# Отражаем атаку на IOMMU

**Денис Молодяков**
Ведущий разработчик группы разработки графики, «Лаборатория Касперского»

kaspersky

# Содержание

# 1 Атаки на драйверы ARM Mali

В данном материале я подробно расскажу о сути атаки на графический стек, а также рассмотрю и другие типы атак через GPU. В заключение я смоделирую подобные сценарии для микроядерной KasperskyOS и покажу, как мы их митигируем.

Как я уже говорил ранее — атаки на графический стек широко распространены. По данным Google Security Team, с 2021 года большинство драйверных атак на Android нацелены на GPU. Для целей этого текста мы сфокусируемся на трех кейсах:

- уязвимости 2022 года, благодаря которой атакующий мог получить полный доступ к устройству и отключить модуль SELinux, воспользовавшись ошибкой управления памятью (более детальное описание атаки можно прочесть тут);
- атаке, эксплуатирующей уязвимость класса use-after-free и предоставляющей контроль за трансляцией графических адресов, но через ошибку в другом механизме драйвера;
- еще одной эксплуатации уязвимости use-after-free, схожей по смыслу и результатам с предыдущей.

Я подробнее расскажу об атаке через первую уязвимость, потому что именно на этом кейсе базируются последующие. Первый этап этой атаки идет на драйвер графического ускорителя ARM Mali. Ее успех — результат сочетания нескольких факторов:

1. наличия уязвимости в коде (нет механизма проверки переполнения целого);
2. особенности прикладного API (без функционала по алиасам, например, выполнить атаку было бы сложнее);
3. немитигированных на тот момент и на том устройстве особенностей slab allocator;
4. отсутствия контроля со стороны ядра за IOMMU (Input/Output Memory Management Unit) аппаратуры.

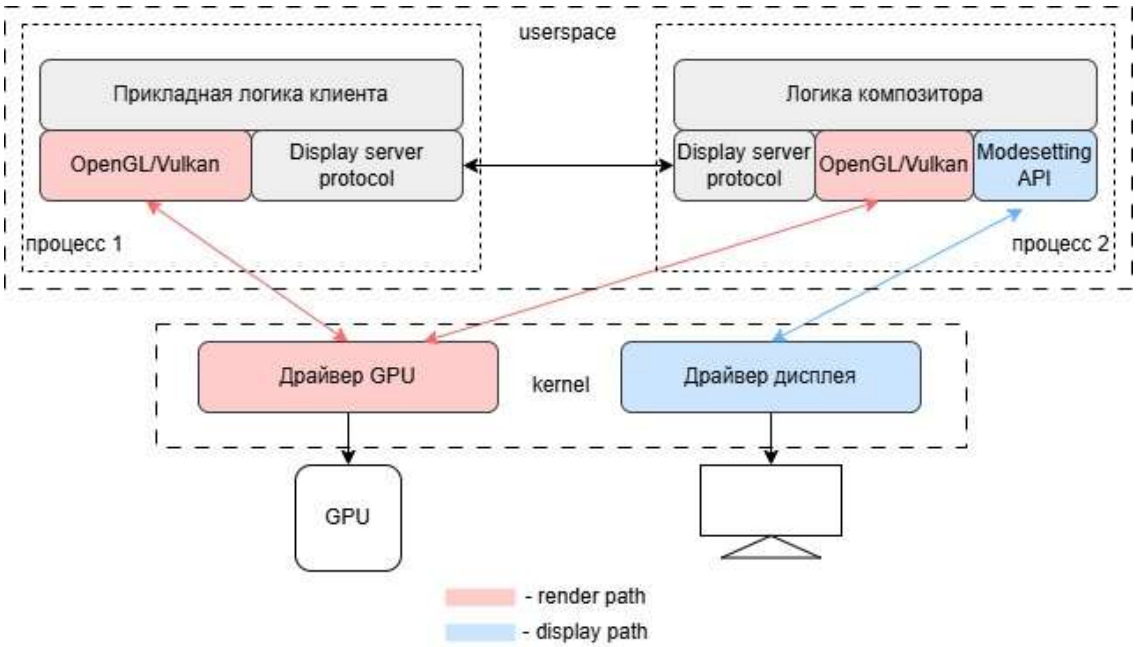
С первым пунктом ничего не поделаешь — уязвимости в коде будут существовать столько, сколько программисты будут писать код. Второй — связан с использованием особенностей конкретного драйвера. Грубо говоря, уязвимость можно эксплуатировать, используя именно такой набор API. На другом драйвере, тоже имеющем уязвимости в коде, но предоставляющем другой API, проэксплуатировать такую же уязвимость не получится. Третий пункт — по сути связанный с дизайном аллокатора памяти — это достаточно специфическая тема, которую стоит обсудить отдельно, потому что мы долго ее исследовали (пишите в комментариях, если хотите прочитать отдельный подробный разбор). Но в этой статье мне было бы интересно поговорить про пункт 4.

# 2 Модель графического стека

В этом материале мы будем смотреть на графический стек, а точнее — на то, как он работает с памятью. По сути, речь идет о многослойной системе, которая отвечает за рендеринг и вывод изображения на дисплей. Именно сбои в выводе изображения на экран могут быть индикатором начавшейся атаки на драйвер.

Чтобы понять происходящее, начнем с описания модели стека. Его можно условно поделить на части по функциональному назначению компонентов: display path и render path. Когда-то эти термины были подсмотрены мной на просторах Интернета, так что в этой статье я использую именно их.

Первая часть отвечает за непосредственно вывод картинки на экран, вторая — за формирование картинок, то есть рендеринг. Иногда обе части реализованы в одном драйвере, например для дискретных видеокарт. Чаще — это разные устройства от разных вендоров под управлением разных драйверов. Мы будем рассматривать как раз такой случай, у нас в фокусе только драйвер GPU, то есть мы смотрим на **render path**.



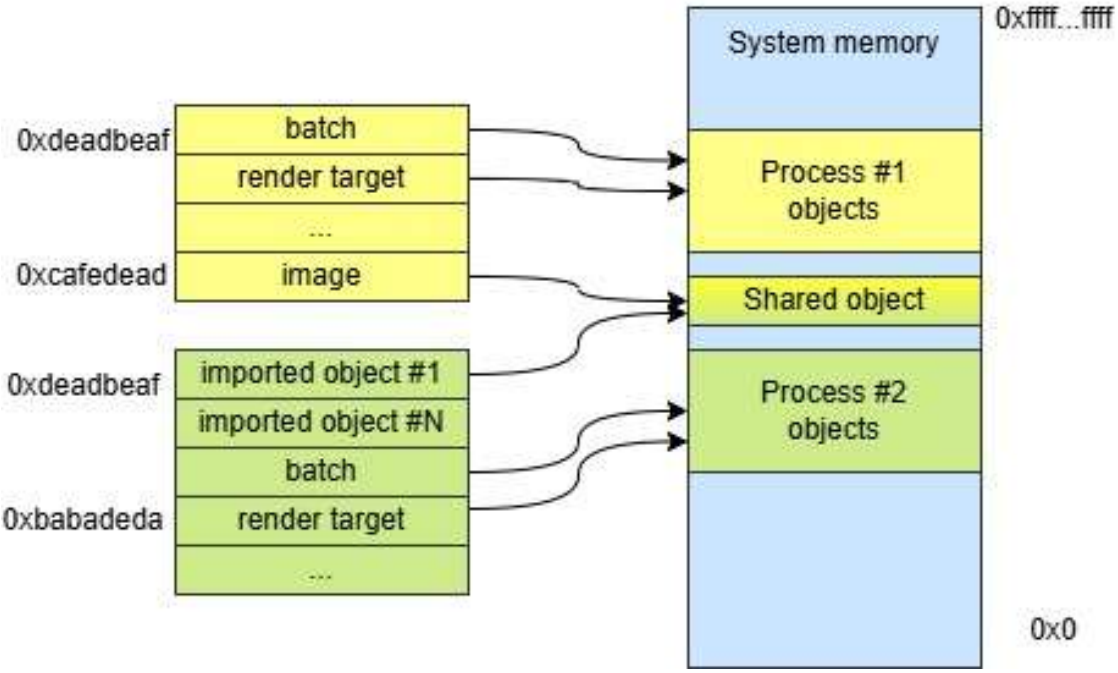

**Деление графического стека**

**Render path** тоже можно условно поделить на две части. Реализацию графического API (OpenGL, Vulkan), компиляцию шейдеров, формирование буфера команд для исполнения на видеокарте обычно выносят в userspace. К тому же, поскольку она может раскрыть внутренний дизайн и особенности видеокарты, эту часть часто закрывают и поставляют в виде библиотек. Как это и происходит в случае с Mali.

Ко второй части относится управление аппаратурой. В Android/Linux оно происходит в ядерном драйвере. Здесь организуются работа с памятью, арбитраж доступа клиентов, постановка команд на исполнение, обработка прерываний от видеокарты и прочие подобные операции.

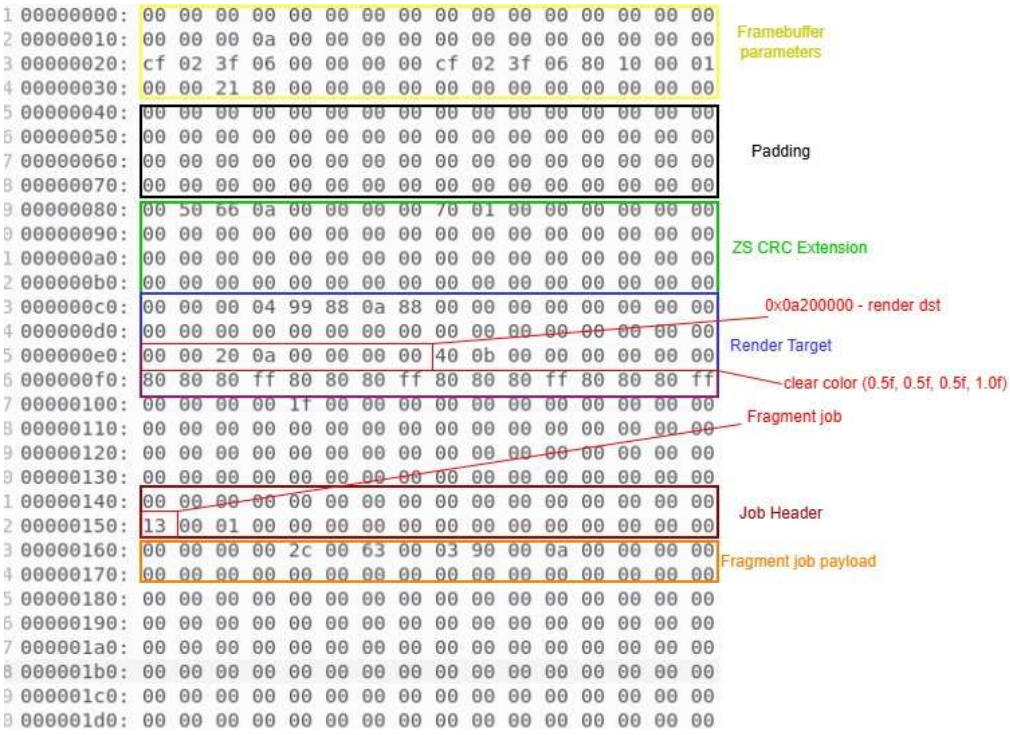
## 2.1 Верхняя часть render path

При выполнении функций графического API создается ряд артефактов: шейдеры компилируются в специфичную для устройства систему команд, текстуры загружаются или создаются в run-time, формируется набор команд для видеокарты, которые описывают «задание» на отрисовку сцены и так далее. Артефакты специфичны для каждого клиента видеокарты, их нужно где-то хранить, а видеокарта должна иметь к ним доступ.




Для этого условно «верхняя» часть стека (реализация OpenGL/Buffer manager) «просит» у «нижней» (драйвер GPU/аллокатор DMA-памяти) выделить память нужного размера под эти объекты и буфер под картинку — результат рендеринга сцены.

В буфере команд для видеокарты лежат инструкции: какие объекты по каким адресам использовать и по какому адресу положить картинку. Ниже приведен фрагмент задания (job) для видеокарты Mali архитектуры Bifrost.




Здесь нужно кратко пояснить специфику исполнения ARM Mali. Возьмем в пример микроархитектуру Bifrost: просто потому, что у нас была под рукой плата с таким GPU и есть опенсорсный драйвер, где описана его система команд с особенностями работы аппаратуры.

Буфер команд для исполнения на видеокарте с Mali GPU делится на несколько задач (jobs), которые объединяются в цепочку (job chain). Вообще задачи на этой видеокарте бывают нескольких типов:

- VERTEX (запускает вершинный шейдер);
- FRAGMENT (запускает фрагментный шейдер и пишет финальное изображение);
- COMPUTE (запускает вычислительный шейдер).

Есть еще несколько специфических задач, рассмотрение которых выходит за рамки данной статьи. Выше приведен пример фрагментной задачи. Она поделена на секции. Нам сейчас интересны секция RENDER_TARGET и поле rgb.base (подробнее с полями секций вы можете ознакомиться здесь). Здесь содержится адрес — 0x0a200000, куда будет помещено изображение. Этот адрес **графический,** как и другие адреса, которыми оперирует видеокарта. Если на видеокарте включено устройство IOMMU, то адрес является виртуальным. Такое виртуальное адресное пространство у каждого клиента свое: это обеспечивают железо и драйвер. Далее рассмотрим подробнее, где и как используется этот адрес.

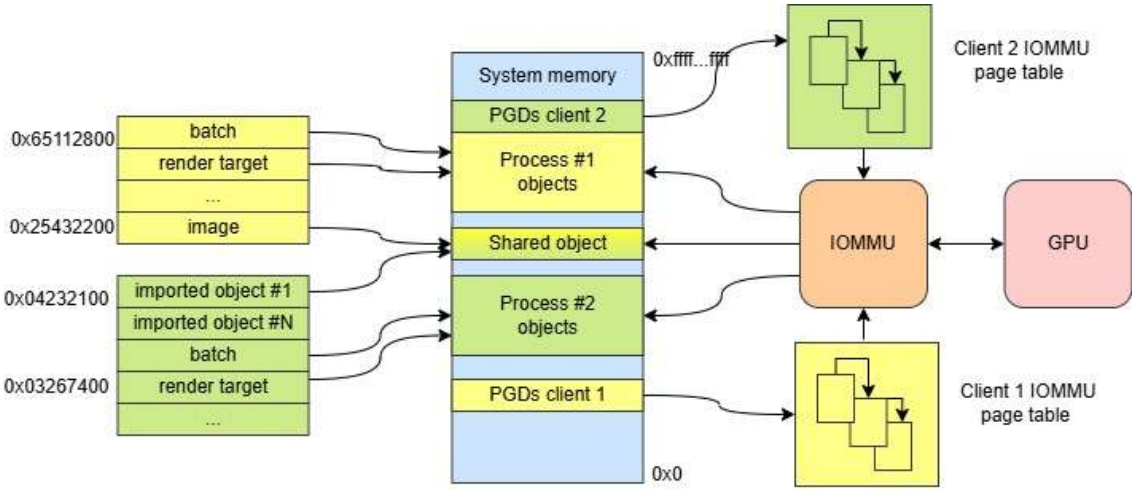
## 2.2 Нижняя часть Render Path

Нижняя часть стека, то есть драйвер видеокарты, одна на всех клиентов. Однако, когда приложение или пользователь начинают работу с ней, для запроса создается уникальный клиентский контекст. Это некоторый набор данных и ресурсов. Указанный выше адрес (0x0a200000) — графический адрес фреймбуфера для конкретного клиента. Чтобы изолировать память, предназначенную для каждого клиента, на стороне железа используется IOMMU.

Объект (обычно используют термин «графический объект» — graphics object) — это, например, буфер команд, он создан по запросу клиента. Под него выделена физическая

память, построена таблица страниц и определен графический адрес, по которому с ним может работать видеокарта. Клиенту нужно записать в этот объект что-то осмысленное, чтобы видеокарта могла это выполнить. Он маппит объект к себе в адресное пространство, получает обычный виртуальный адрес и пишет в него данные.

***Уточнение.*** *На самом деле, физическая память под объект выделяется не сразу в момент его создания. Сначала система может просто создать запись об этом объекте. Непосредственно физическая память выделится позже — когда программа обратится к этому объекту (например, начнет записывать в него данные). Такой подход называется отложенным выделением памяти (lazy allocation).*

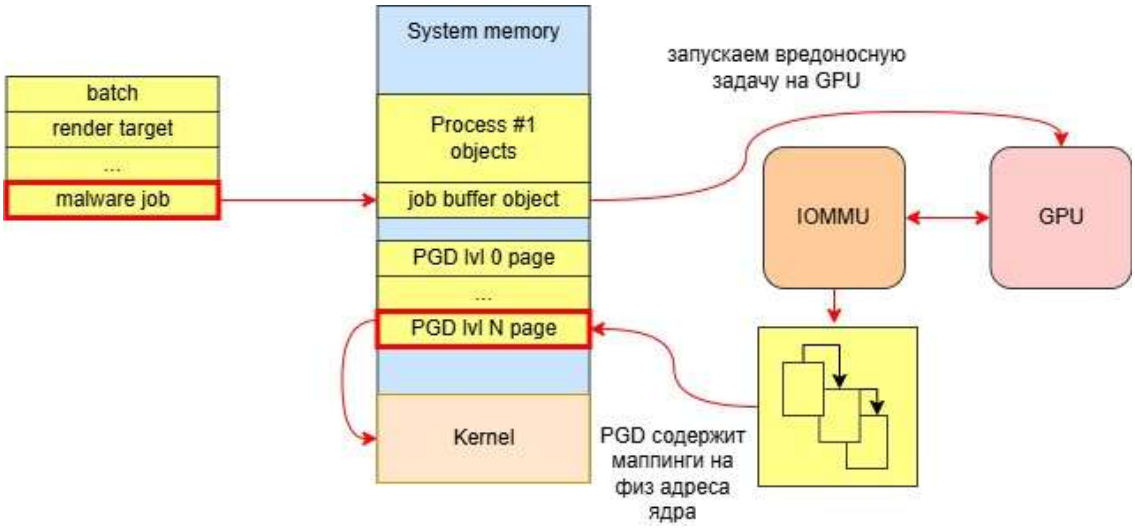



| CPU address | GPU address | PHY address |
|---|---|---|
| 0x03267400 | 0x0a200000 | 0x111FBE000 |
| 0x65112200 | 0x0a009000 | 0x001000100 |

**Объект сопоставлен не только с графическим, но и виртуальным адресом в пространстве клиента. Здесь PGDs — набор физических страниц, содержащих записи для различных уровней трансляции IOMMU**

# 3 Подробнее о сценарии атаки

Теперь перейдем к самому сценарию первой атаки. В ней злоумышленник использует уязвимость в драйвере видеокарты, чтобы создать специальный фрагмент видеопамяти. Он создает условия, чтобы система освободила фрагмент и вернула его в общий пул свободной памяти. При определенных условиях этот фрагмент будет использован под один из уровней таблицы страниц. Цель злоумышленника — моментально перехватить этот только что освобожденный фрагмент и подменить его содержимое, разместив там свою собственную, подконтрольную ему таблицу страниц. Эта таблица отвечает за трансляцию адресов для графического процессора.




**Как только это происходит, у злоумышленника появляется доступ к IOMMU**

В этот момент начинается атака со стороны GPU. Злоумышленник может сформировать записи как угодно, используя любые физические адреса, и при определенном мастерстве и удаче — достигнуть ядра, иначе говоря «получить рут».

Теперь вернемся к тому адресу — 0xa200000, который мы видели в дампе фрагментной задачи. Если составить таблицу страниц таким образом, что этому адресу будет соответствовать физический адрес памяти, где лежит ядро — его можно переписать серым фоном.

Если использовать вычислительный шейдер и вычислительную задачу (COMPUTE JOB), то можно переписать нужные физические адреса не значением цвета, а чем-нибудь полезным. Например, инжектировать собственный shell-код, отключить SELinux и так далее.

Итак, мы разобрали ход первой атаки. В этом случае первым под удар попал драйвер, после чего злоумышленники эксплуатировали IOMMU видеокарты. Здесь надо отметить: несмотря на сложность атаки, она не уникальна. Подобные инциденты с драйвером и эксплуатация аппаратуры со злым умыслом хорошо исследованы и описаны.

Ниже рассмотрим подробнее похожие варианты атак и их митигации.

# 4 Атаки на драйвер и их митигации

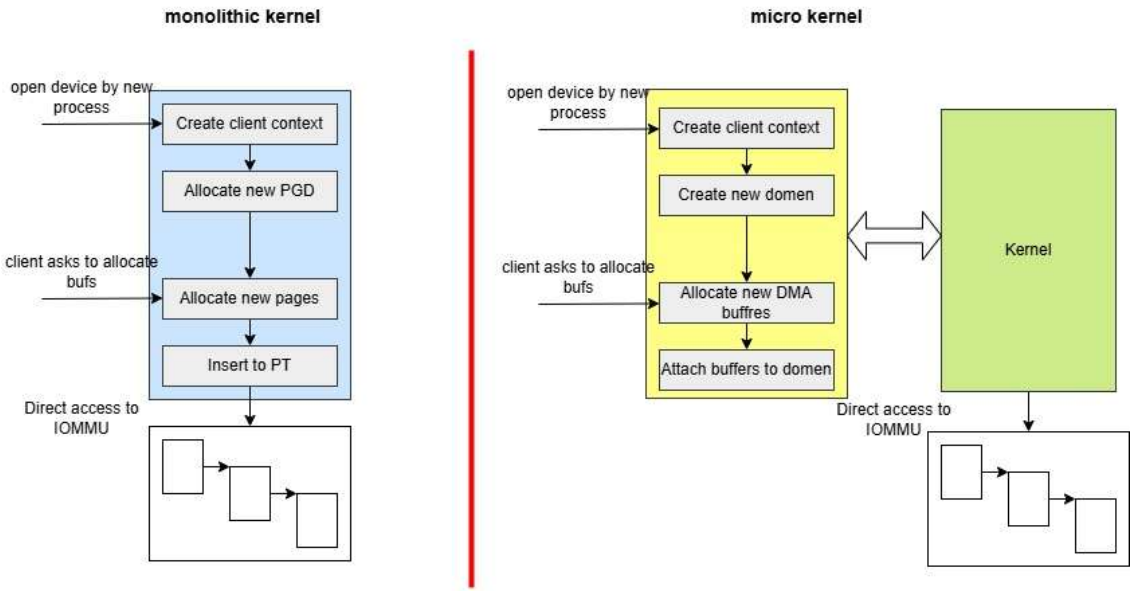
Успешность и ущерб атаки на драйвер зависят как от дизайна драйвера, так и операционной системы в целом. Драйвер управляет всеми клиентскими контекстами. Работает принцип: чем больше функциональности в него заложено, тем больше поверхность атаки.

Мы не будем останавливаться на описании атак, эксплуатирующих логические ошибки и уязвимости в коде (типа «переполнение целого», «ошибка на единицу» и пр.) Статей на тему SDL и практик безопасного программирования написана масса, да и методы митигации известны. Нам интересна особенность дизайна — полномочия драйвера управлять таким могучим механизмом, как механизм трансляции графических адресов.

Критерием конструктивной безопасности и хорошего дизайна графического драйвера могло бы стать уменьшение поверхности атаки. Существует три варианта для «прокачки» этого показателя.

## 4.1 Выносим из драйвера критически важную для безопасности системы функциональность

В описанной выше атаке на драйвер использовалась функция управления MMU (IOMMU) видеокарты. Получение злоумышленником доступа к IOMMU может стать фатальным для работы всего устройства. В монолитных ОС этот функционал никуда не спрячешь, а вот в микроядерных системах управление IOMMU можно вынести в отдельный изолированный и доверенный процесс. Также мы можем избавить драйвер от необходимости управления физической памятью (аллокацией/деаллокацией), сосредоточив все в одной точке — ядре или отдельном доверенном процессе.

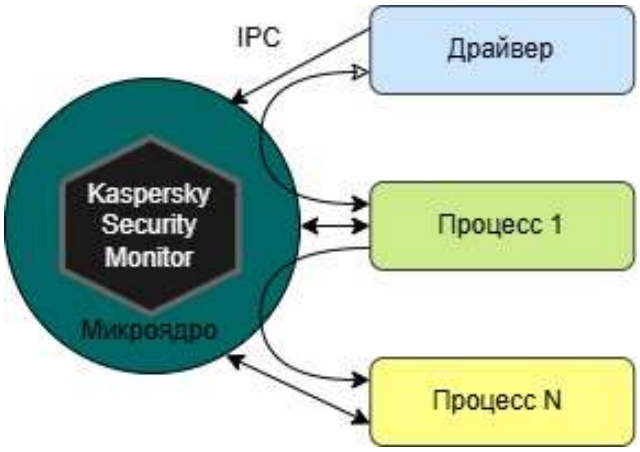


## 4.2 Изолируем драйвер в отдельном адресном пространстве

Логическим продолжением и дополнением предыдущего пункта будет изоляция драйвера в отдельном адресном пространстве. Функционирование драйвера в отдельном изолированном домене позволяет предотвратить распространение атаки. Опять же, эта опция доступна лишь для микроядерных (и прочих нано-, экзо- и т. д.) ОС. При изоляции драйвера успешная атака на него не приводит автоматически к компрометации всего ядра. Кроме того, графический драйвер защищен от других драйверов, что снижает риск их компрометации.

## 4.3 Контролируем использование функциональности драйвера

В KasperskyOS, помимо изоляции драйверов, есть возможность контролировать взаимодействия между ними и другими процессами через reference monitor, работающий на основе политик. На уровне политик можно дать больше прав доверенным клиентам и предоставить минимально необходимые привилегии недоверенным.




В Linux для графических драйверов после многих атак и долгих исследований в конце 2025 года предложили фильтрацию доступа на уровне IOCTL путем расширения SELinux-политики.

Перечисленные выше варианты — как комплексно, так и по отдельности — позволяют избежать сценария из вышеописанной атаки. Но, увы: атака на драйвер — не единственный вариант проникновения.

# 5 Атаки без графического драйвера

Атаки на графический стек увлекательны тем, что проникновение может идти не только через зловредный или недостаточно качественный драйвер. Расскажу подробнее о других вариантах.

## 5.1 DMA-атаки

DMA (Direct Memory Access) — механизм, позволяющий получить доступ к системной памяти периферии без участия CPU. Очень хороший обзор DMA-атак с использованием IOMMU можно найти здесь. Когда оборудование обращается к системной памяти, IOMMU работает в том числе как контролер: преобразует адреса устройства в реальные физические адреса и «проверяет», разрешен ли доступ. Всю эту логику настраивает драйвер. Если в драйвере или в самом блоке IOMMU есть уязвимость, злоумышленник может использовать ее, чтобы «врезаться» в процесс трансляции адресов. В результате устройство получает возможность читать и записывать данные в защищенные области системной памяти — например, в память ядра или других процессов. Так драйвер становится лишь первой ступенью в цепочке атаки, конечная цель которой — полный контроль над системой через скомпрометированное устройство.

**Митигация.** Защита от подобных атак должна быть комплексной. Для рассмотренной выше атаки достаточно делегировать функционал по управлению IOMMU ядру или отдельному доверенному компоненту и, конечно, следовать практикам SDL. Большая часть решений, которая позволила бы митигировать такие атаки, лежит в плоскости дизайна системы. В разделе «Как мы защищаемся в KasperskyOS» я попытаюсь обобщить их.

## 5.2 Подмена firmware

Многие видеокарты используют специализированное ПО (firmware), исполняемое внутри железа. Оно может управлять планированием задач, питанием и вообще иметь полный доступ к видеокарте. Как, например, прошивка GuC для Intel. Многие ОС предоставляют firmware практически неограниченный доступ к физической памяти. Таким образом, получив контроль над прошивкой, можно добраться до важных данных, включая и само ядро.

**Митигация.** Для борьбы с подменой firmware обычно реализуют подписывание и верификацию прошивки при загрузке (сценарии Trusted Boot) и обновлении (Secure Update). Однако не стоит забывать, что эти общесистемные механизмы достаточно сложны и в них самих могут быть уязвимости.

## 5.3 Нарушение изоляции контекстов

Следующая атака становится возможной при совместном использовании двумя процессами графической памяти. Для экономии видеопамяти драйвер может размещать небольшие графические объекты от разных процессов в рамках одной физической страницы (например, 4 KiB). Однако это нарушает базовую изоляцию процессов. Если оба процесса имеют доступ к одной странице памяти видеокарты (GPU), то шейдер одной программы может прочитать или изменить данные другой, а это серьезная уязвимость. Еще опаснее ситуация, когда в общем пуле памяти находятся и системные (ядерные) объекты. В этом случае возможны утечка критических адресов или даже прямой несанкционированный доступ к данным ядра из непривилегированного процесса.

**Митигация.** Этой атаке можно противодействовать усложнением алгоритма размещения с учетом подлежащих абстракций. В примере из абзаца выше достаточно учитывать границы физической страницы, но в некоторых случаях (особенно при работе в облачных средах) систему радикально разделяют, например через запрет на совместное использование одного GPU разным клиентам (виртуальным машинам). Другой пример изоляции — NVIDIA MIG.

## 5.4 Исполнение вредоносной программы на GPU

Еще одна разновидность атак производится на буфер команд. Вариантов реализации буфера команд (batch-буфера, буфера задания и пр.) множество, и все они зависят от конкретной видеокарты. Вариант исполнения задачи для mali-g72 был приведен выше. Если не включен IOMMU и отсутствует механизм изоляции клиентских контекстов, то у злоумышленника есть возможность прочесть и изменить команды любого клиента. Если базовый механизм защиты (IOMMU) включен, то злоумышленник может «повесить» видеокарту (спровоцировать ее сбой), реализовав механизм отказа. В определенный момент планировщик задач для видеокарты поймет, что произошло, и попробует сбросить задачу. В зависимости от дизайна планировщика (на стороне драйвера или в прошивке на стороне аппаратуры) и железа, это можно сделать сильно по-разному. Драйвер panfrost, например, просто отресетит видеокарту и сделает рестарт всех исполнявшихся задач. Это может привести к чему угодно, от мерцания картинки до полной потери изображения на экране. И в рамках работы бизнеса такая неполадка может стоить немалых потерь…

*Насколько это критично? Как-то один коллега сказал мне: «Подломали GPU без получения рута? Вывели неприличное изображение на экран? Ничего страшного, переживем!». Я вспомнил его слова, когда стоял ночью в очереди возмущенных людей у единственной автоматической кассы (обычная ночью не работала, из персонала в магазине был только охранник), которая вместо чека и пробитого товара показывала рекламу. Конечно, ее не взломали, просто сбой в ПО, но могло быть и иначе... Но найдется ли человек, который скажет владельцу бизнеса, потерявшему деньги: «Ничего страшного, друг, переживешь!» :)*

Для драйвера i915 в Linux исполняемый на видеокарте буфер команд, сформированный клиентом, дополняется специальной привилегированной частью, которую пишет сам драйвер. После исполнения привилегированной части выполняется команда на «прыжок» в пользовательский буфер команд с понижением привилегий. Если попытаться записать привилегированные команды в ту часть буфера, которая заполняется на стороне пользователя (не привилегированную), то при исполнении аппаратура их проигнорирует (например, интерпретирует как NOOP). При этом возможность «подвесить» видеокарту по-прежнему останется.

Однако у злоумышленника появляется большое поле для маневра, если ему удалось подломать драйвер и получить доступ к привилегированной части буфера команд, например поменять маппинги физических страниц для глобальной таблицы трансляции адресов (GGTT). В этом случае вопрос лежит уже в плоскости защиты самого драйвера.

**Митигация.** Митигация атаки сильно зависит от возможностей конкретной видеокарты. На какой-то видеокарте можно сбросить задачу «сбойного» клиента без влияния на других, и это не повлияет на общую картину. На другой — может потребоваться полный сброс аппаратуры. Тогда придется решать проблему на уровне ПО: отслеживать «зависших» клиентов и как-то исправлять «фризы» (зависание изображения), тиринги (смещение частей картинки) и прочие артефакты изображения. При этом то, что выводит на экран финальную картинку (обычно это «композитор»), должно быть максимально защищено. Про атаки на высокоуровневые компоненты мы поговорим в другой раз. Также всегда остается теоретическая возможность run-time-санитайзинга буферов команд от

клиентов на стороне драйвера, но это с большой долей вероятности «убивает» производительность.

## 5.5 Атака через IOTLB

Вообще атаки через IOTLB можно отнести к DMA-атакам, но мы разберем их отдельно, как особенно экзотический, но все-таки возможный вариант. IOTLB выполняет те же функции для графической карты, что и TLB (Translation Lookaside Buffer) для обычного MMU — хранит кэш трансляций. Каждый раз, когда перестраивается таблица страниц, IOTLB обычно сбрасывают. Его чистка может быть довольно затратной по времени операцией, и при асинхронном ее выполнении — когда управление возвращается до фактического сброса кэша — может возникнуть ситуация, при которой освобожденная страница уже используется кем-то, но при этом еще находится в кэше. Тогда со стороны видеокарты на короткое время появляется канал для доступа к чужой памяти.

**Митигация.** Митигируется эта атака не слишком хорошо, снижая производительность ресурса из-за частого сбрасывания TLB или изоляции компонентов с разным уровнем доверия. Также для митигации важно знать точно — сбросился ли кэш трансляций перед построением новой таблицы страниц или нет. Определить вариант подходящей митигации можно только с учетом рассмотрения особенностей конкретной аппаратуры.

## 5.6 Атаки на кэш GPU

Атаки на кэш GPU так же сложны, как и атаки на кэш CPU, но при этом ограничены в применении конкретной картой. В рамках подобной разновидности атак при замещении данных в кэше исследуется набор данных, с которыми работала «жертва». В этой статье приведен пример подобной атаки на видеокарты NVIDIA с использованием фреймворка CUDA.

**Митигация**. Здесь интересная ситуация. Команды для видеокарты формируются внутри библиотеки с реализацией графического API. Часто это проприетарная библиотека с закрытыми исходниками. Если вы детально изучили систему команд видеокарты, научились управлять кэшами GPU и знаете, какой командой можно, например, принудительно сбросить кэши, и даже знаете, в какой момент, куда вы поставите эту команду? Будете патчить буферы команд каждого клиента в run-time или как-то научите драйвер делать это? Как понять, что такая инвазия не порушит логику работы конвейера отрисовки в какой-нибудь ситуации? Эта проблема требует комплексного решения — как со стороны аппаратуры, так и со стороны ПО.

# 6 Как мы защищаемся в KasperskyOS

Как видно из перечисленных выше атак и возможных митигаций, устранить многие из них можно через «редизайн» операционной системы и продукта. Например, полноценную изоляцию драйвера от ядра крайне сложно реализовать в монолитной ОС. Нельзя обеспечить контроль доступа к сервисам/драйверам, не имея для этого механизмов, заложенных в фундаменте ОС.

KasperskyOS справляется с большинством вышеописанных уязвимостей (кроме тех, что требуют митигаций на уровне аппаратуры) благодаря следующим факторам.

1. Микроядерная архитектура позволяет нам изолировать драйвер видеокарты в отдельном процессе, благодаря чему **его компрометация не означает**

**компрометацию ядра** (как я уже упоминал выше в разделе про атаки на драйвер).

2. Контроль взаимодействия между процессами (IPC) с помощью политик безопасности позволяет нам реализовать принцип наименьших привилегий. Например, мы можем явно указать единственный процесс, которому можно будет выполнять управление выводом картинки на экран (mode setting). Тогда, даже если он не запущен и не принял управление дисплейным контроллером, никто другой не сможет подобрать управление видеокартой, «проходя мимо».
3. Обязательное включение IOMMU (если он имеется) на как можно более раннем этапе. Здесь вроде бы тоже все понятно — если есть механизмы защиты, то надо их включить. Однако так поступают не все и не всегда. Зачастую какой-нибудь DMA-контроллер можно заставить работать без IOMMU, он будет использовать обычные физические адреса. Работать будет, но какова цена ошибки?
4. Управление IOMMU централизовано и реализовано внутри ядра. Ядро управляет созданием доменов, пишет в регистры IOMMU и заполняет таблицы страниц.
5. Единое и централизованное управление аллокацией физической памяти. Драйверы не имеют возможности оперировать аллокацией физической памяти напрямую, все действия осуществляются через IPC под строгим присмотром reference monitor. Повторюсь, чем меньше свободы у драйвера, тем меньше возможностей его сломать. Однако не стоит перебарщивать — из драйвера в отдельный доверенный процесс или ядро нужно выносить только критически важный и наиболее общий функционал, такой как управление памятью.
6. Изоляция контуров рендеринга и вывода изображения на экран, обработка нештатного поведения клиентов. Эту особенность следует отнести к архитектуре всей системы, включая продуктовую логику, но я не могу не упомянуть про это здесь, так как безопасности архитектуры всего решения целиком мы тоже уделяем внимание.
7. Обеспечение минимальной гранулярности аллокации DMA-памяти. То есть если выделяется 64 объекта размером 64 байта, при минимальном размере страницы 4 KiB будет затрачено много памяти, но объекты никогда не будут размещены на одной странице. Что это значит? Когда в драйвер постучатся клиенты и попросят выделить память под свои объекты, они гарантированно будут на разных физических страницах. Конечно, всегда нужно соотносить параметры пейджинга для IOMMU и гранулярность аллокаций физической памяти. Если размер страницы будет, например, 2 MiB, то order аллокации физической памяти должен совпадать.
8. Использование KASLR (технологии «перемешивания» расположения ключевых частей ядра в памяти при каждой загрузке компьютера) и, конечно, всех возможных харденингов. Это позволяет затруднить атаку, если адреса ядра все же каким-то образом утекли.

При этом остается ряд вещей, которые требуют устранения на уровне аппаратуры. Например, мы не можем заставить производителей видеокарт унифицировать свои решения, чтобы общие схемы были применимы ко всем типам аппаратуры.

# 7 По итогу…

Наверное, на этот момент читателю уже стало понятно, насколько важно учитывать требования безопасности на самых ранних этапах разработки системы. Правильные решения, заложенные на этапе проектирования в фундаменте системы, значительно облегчают митигации угроз в дальнейшем. А какие-то аспекты безопасности вступят в конфликт с требованиями по производительности, и здесь потребуются усилия и искусство архитекторов.