Безопасная разработка kvadraOS - Enterprise системы на базе AOSP.

Александр Дубинин
YADRO

О компании YADRO

YADRO — российская технологическая компания (входит в «ИКС Холдинг»), объединяющая направления разработки и производства вычислительных платформ, систем обработки и хранения данных, телекоммуникационного и сетевого оборудования, персональных и «умных» устройств, микропроцессорных ядер и fabless-разработку микропроцессоров. R&D центры расположены в Москве, Санкт-Петербурге, Екатеринбурге, Нижнем Новгороде и Минске.

О компании YADRO: yadro.com/ru/company

Продукты YADRO: yadro.com/ru/products

Портал инженерной культуры: engineer.yadro.com

Парадигма - «Продукты должны быть безопасны!»

Составляющие качественного и безопасного продукта это:

- Инженерная культура - минимализм и конструктивизм, ничего лишнего;
- Сначало дело, потом формальности, принцип «делай, как я!»;
- Подробная документация - от архитектуры до реализации в коде, активно внедряем и применяем технологии «as a Code»;
- Знаем каждый байт нашего красивого и надежного кода (SCA/SAST/DAST);
- Управление уязвимостями и дефектами - документированный и понятный процесс, разработаны и внедряются собственные инструменты и методики;
- Изолированная сборка всегда и везде.

Несколько наших продуктов (СХД, UEFI BIOS) прошли сертификацию ФСТЭК.

kvadraOS - мобильная Enterprise OS на базе AOSP

Да, мы тоже хотим свой, проверенный и безопасный Android! Почему?

- Кроссплатформенный, огромное количество ПО;
- Нам не нужны ни китайские, ни американские «закладки»;
- Устали от отсутствия нормальной поддержки у китайфонов/планшетов;

Делаем, потому что можем!

- **Своя ОС для своего «железа»** - как на ARM, так и на RISC-V;
- **Следуем принципам и практикам безопасной разработки**;
- **Все собираем сами** из кода: SDK, NDK, AOSP, свои приложения и сервисы.

Мифы и легенды про AOSP

Миф №1: Android Open Source Platform - проприетарное ПО;

- Как бы «*Open Source*» и лицензия Apache 2.0 говорят сами за себя.

Мифы и легенды про AOSP

Миф №1: Android Open Source Platform - проприетарное ПО;

- Как бы «*Open Source*» и лицензия Apache 2.0 говорят сами за себя.

Миф №2: Сервисы Google нельзя отключить/извлечь;

- В AOSP нет сервисов Google, для наших клиентов мы добавили свои сервисы.

Мифы и легенды про AOSP

Миф №1: Android Open Source Platform - проприетарное ПО;

- Как бы «*Open Source*» и лицензия Apache 2.0 говорят сами за себя.

Миф №2: Сервисы Google нельзя отключить/извлечь;

- В AOSP нет сервисов Google, для наших клиентов мы добавили свои сервисы.

Миф №3: Собрать что-то на основе AOSP могут только инженеры Google, т.к. среда разработки закрытая;

- Все прекрасно собирается полностью из исходного кода, в изолированной среде, в обычном Linux. Вся документация есть, воспроизвести - лишь вопрос профессионализма инженеров.

Мифы и легенды про AOSP

Миф №1: Android Open Source Platform - проприетарное ПО;

- Как бы «*Open Source*» и лицензия Apache 2.0 говорят сами за себя.

Миф №2: Сервисы Google нельзя отключить/извлечь;

- В AOSP нет сервисов Google, для наших клиентов мы добавили свои сервисы.

Миф №3: Собрать что-то на основе AOSP могут только инженеры Google, т.к. среда разработки закрытая;

- Все прекрасно собирается полностью из исходного кода, в изолированной среде, в обычном Linux. Вся документация есть, воспроизвести - лишь вопрос профессионализма инженеров.

Миф №4: AOSP содержит закладки, т.к. Google - американская компания;

- Тогда и ядро Linux использовать нельзя - т.к. Free Software Foundation тоже в США.

Данная «проблема» решается применением **принципов и методик безопасной разработки** грамотными инженерами.

Архитектура: MILS, K.I.S.S, Loose Coupled и все-все-все

Как многие знают, безопасная разработка начинается с архитектуры.

В процессе разработки мы:

- **«Грокнули»* архитектуру** AOSP и возрадовались - «Красивая!»
- Хорошо и детально ее **описали в документации**;
- **Осознано проектируем** наши сервисы и компоненты для ОС;
- Проработали **модель угроз** и определили **поверхность атаки**.

* «Грокнуть» (от англ. to grok) — сленговое выражение, означающее: понять что-либо на глубоком, интуитивном уровне, вникнуть в самую суть, «познать до основания». Из «Stranger in the strange land» Р.Хайнлайна.

Архитектура: MILS, K.I.S.S, Loose Coupled и все-все-все

Как многие знают, безопасная разработка начинается с архитектуры.

В процессе разработки мы:

- **«Грокнули»* архитектуру** AOSP и возрадовались - «Красивая!»
- Хорошо и детально ее **описали в документации**;
- **Осознано проектируем** наши сервисы и компоненты для ОС;
- Проработали **модель угроз** и определили **поверхность атаки**.

Архитектура AOSP содержит почти все известные приемы повышения надежности и безопасности:

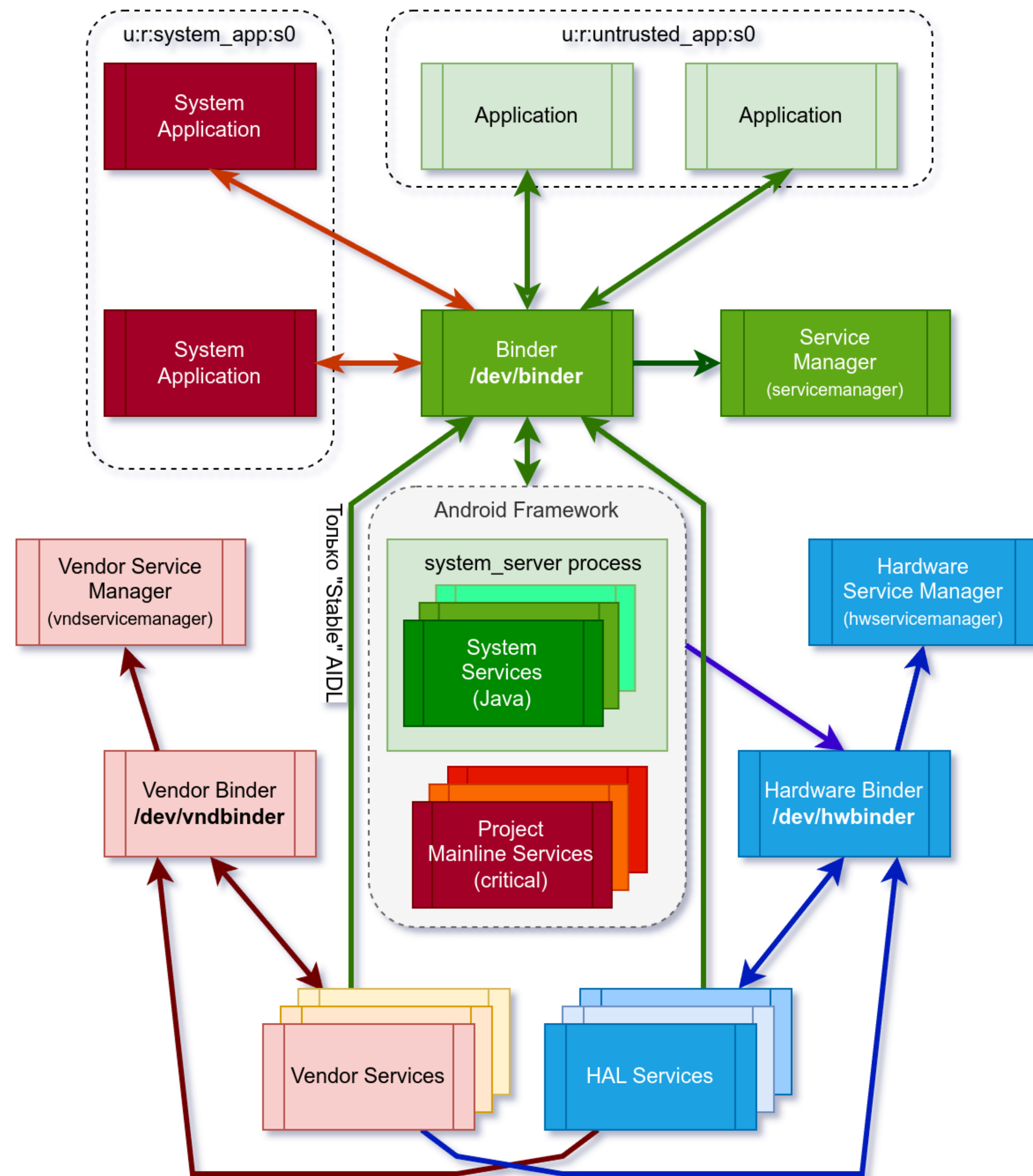
- **MILS** : Multiple Independent Layers Of Security;
- **K.I.S.S.** : Keep It Simple Stupid;
- **Loose coupled** services;
- **UNIX Way;**
- **Isolation and privilege separation.**

* «Грокнуть» (от англ. to grok) — сленговое выражение, означающее: понять что-либо на глубоком, интуитивном уровне, вникнуть в самую суть, «познать до основания». Из «Stranger in the strange land» Р.Хайнлайна.

Как именно все это работает?

Архитектура безопасности AOSP основывается на:

- **Binder** - весьма «продвинутом» асинхронном механизме IPC;
- **SELinux** - мандатный контроль доступа (не MLS модели Белла-Ла-Падула!);
- выделении своего **уникального UID каждому приложению**;
- фильтрации системных вызовов через **seccomp-bpf**
- **AIDL** - четко определенном способе взаимодействия процессов в системе.



«У мене внутре... гм... не... неонка»* Вызовы композиционного анализа AOSP

Из чего состоит AOSP?

- Android Common Kernel («ванильное» ядро Linux + патчи AOSP) + патчи вендора HW + патчи YADRO
- Нативный код (**NDK**) - glibc, библиотеки, SELinux, бинарные блобы firmware, libart.so
- HAL & Vendor services + Android Framework + **SDK**
- Applications (Java, Kotlin)

* А. и Б. Стругацкие, «Сказка о Тройке» https://strugacki.ru/book_27/1159.html

«У мене внутре... гм... не... неонка»*
Вызовы композиционного анализа AOSP

Из чего состоит AOSP?

- Android Common Kernel («ванильное» ядро Linux + патчи AOSP) + патчи вендора HW + патчи YADRO
- Нативный код (**NDK**) - glibc, библиотеки, SELinux, бинарные блобы firmware, libart.so
- HAL & Vendor services + Android Framework + **SDK**
- Applications (Java, Kotlin)

Готового инструмента или решения для SCA нет. Что делать?

- Перехватывать процесс сборки, анализировать логи и находить транзитивные зависимости;
- Использовать промежуточные файлы, содержащие результаты анализа разных этапов сборки;
- Проверять: то, что входит в SBOM - должно однозначно соответствовать образу системы;
- Обогащать CycloneDX JSON зависимостями и другой информацией;
- Попробуем использовать BUILDOGRAPH.

* А. и Б. Стругацкие, «Сказка о Тройке» https://strugacki.ru/book_27/1159.html

Анализ уязвимостей AOSP

Недостатки инструментов

- Dependency Track не распознает компоненты AOSP по CPE/PURL;
- В базе NVD нет CPE для компонентов, только ОС: **cpe:2.3:o:google:android:xx.x:*:*:*:*:*:*:***
- Нет поддержки БДУ ФСТЭК (и автоматизации тоже)

Анализ уязвимостей AOSP

Недостатки инструментов

- Dependency Track не распознает компоненты AOSP по CPE/PURL;
- В базе NVD нет CPE для компонентов, только ОС: **cpe:2.3:o:google:android:xx.x:*:*:*:*:*:*:***
- Нет поддержки БДУ ФСТЭК (и автоматизации тоже)

Готового инструмента или решения снова нет. Что делать?

- Реализовали свой собственный инструмент для работы с уязвимостями и создали собственную консолидированную базу уязвимостей с поддержкой БДУ, NVD, GHSA, OSV и т.д.;
- Добавили автоматизацию на основе конфигурации ядра, определения патчей в исходном коде с помощью **vex-kernel-checker** и **vanir**;
- Анализируем известные CVE для AOSP нужной версии (**cpe:2.3:o:google:android:12.1:**);
- Добавляем анализ используемого ядра Linux с учетом его конфигурации;

Анализ уязвимостей AOSP

Недостатки инструментов

- Dependency Track не распознает компоненты AOSP по CPE/PURL;
- В базе NVD нет CPE для компонентов, только ОС: **cpe:2.3:o:google:android:xx.x:*:*:*:*:*:*:***
- Нет поддержки БДУ ФСТЭК (и автоматизации тоже)

Готового инструмента или решения снова нет. Что делать?

- Реализовали свой собственный инструмент для работы с уязвимостями и создали собственную консолидированную базу уязвимостей с поддержкой БДУ, NVD, GHSA, OSV и т.д.;
- Добавили автоматизацию на основе конфигурации ядра, определения патчей в исходном коде с помощью **vex-kernel-checker** и **vanir**;
- Анализируем известные CVE для AOSP нужной версии (**cpe:2.3:o:google:android:12.1:**);
- Добавляем анализ используемого ядра Linux с учетом его конфигурации;

Результат:

- Автоматически маркируется порядка 70% CVE (либо исправлены, либо нерелевантны)

Фаззинг ядра... Быстро, или то, что надо?

Фаззинг - способ выявления дефектов в работающем коде, обеспечения качества.

- Целесообразно фаззить код ядра целевой архитектуры (надо ARM64), а также имеет смысл фаззить код, работающий на целевой платформе (наше «железо»), иначе просто не достигнем нужного кода;
- Фаззить на целевом устройстве - медленно.

Что делать?

- Современная версия syzcaller позволяет «из коробки» работать с Android устройствами!
- Современная версия syzcaller умеет работать с «фермами» устройств;

Наша фаззинг-ферма (пока маленькая):

- Стойки для планшетов, напечатанные на 3D принтере
- Управляющий ПК
- Активный USB HUB и провода USB<>RS232

Фаззинг ядра на «живом» железе - вызовы:

Что было сложного:

- При создании фермы нужно использовать активный USB hub с дополнительным питанием - т.к. при нагрузке планшеты разряжаются быстрее, чем получают энергию по USB;
- Без знания платформы - не получить информацию. Но у нас свое «железо», пришлось подпаяться(!!!) к нужны контактам, чтобы получить последовательный вывод с информацией от ядра через RS232 и внести соответствующие настройки в конфиг;
- Иногда планшет приходит в состояние, когда нужно либо вручную нажимать «reset», либо придумывать автоматику.

Фаззинг ядра на «живом» железе - вызовы:

Что было сложного:

- При создании фермы нужно использовать активный USB hub с дополнительным питанием - т.к. при нагрузке планшеты разряжаются быстрее, чем получают энергию по USB;
- Без знания платформы - не получить информацию. Но у нас свое «железо», пришлось подпаяться(!!!) к нужны контактам, чтобы получить последовательный вывод с информацией от ядра через RS232 и внести соответствующие настройки в конфиг;
- Иногда планшет приходит в состояние, когда нужно либо вручную нажимать «reset», либо придумывать автоматику.

Результат:

- Фаззинг ядра на планшетах ARM64 идет примерно с той же скоростью, что и x86 на мощном сервере в QEMU;
- Находим дефекты, связанные не только с кодом, но и с оборудованием;
- Фаззинг-ферму можно легко масштабировать во всех измерениях.

Такие себе эмуляторы...

Конечно, мы пробовали запускать фаззинг ядра в эмуляторах. Но есть проблемы:

- **QEMU**:
  - Нет адекватной ARM64 конфигурации для запуска, соответствующей «железу». Система банально «валится» на запуске первых же сервисов, нужно отдельно конструировать userlevel чтобы фаззить.
  - Попробовали с «ванильным» ядром и busybox/libc - невероятно медленно, медленнее чем на планшете.

Такие себе эмуляторы...

Конечно, мы пробовали запускать фаззинг ядра в эмуляторах. Но есть проблемы:

- QEMU:
  - Нет адекватной ARM64 конфигурации для запуска, соответствующей «железу». Система банально «валится» на запуске первых же сервисов, нужно отдельно конструировать userlevel чтобы фаззить.
  - Попробовали с «ванильным» ядром и busybox/libc - невероятно медленно, медленнее чем на планшете.
- **Cuttlefish:** «классический» эмулятор для AOSP (QEMU «под капотом»). Одна проблема - не то «железо», совсем другая конфигурация ядра. Пригоден для фаззинга ART.

Такие себе эмуляторы...

Конечно, мы пробовали запускать фаззинг ядра в эмуляторах. Но есть проблемы:

- QEMU:
  - Нет адекватной ARM64 конфигурации для запуска, соответствующей «железу». Система банально «валится» на запуске первых же сервисов, нужно отдельно конструировать userlevel чтобы фаззить.
  - Попробовали с «ванильным» ядром и busybox/libc - невероятно медленно, медленнее чем на планшете.
- Cuttlefish: «классический» эмулятор для AOSP (QEMU «под капотом»). Одна проблема - не то «железо», совсем другая конфигурация ядра. Пригоден для фаззинга ART.
- **ARM Emulator**: те же проблемы, что и QEMU - но чуть быстрее.

Такие себе эмуляторы...

Конечно, мы пробовали запускать фаззинг ядра в эмуляторах. Но есть проблемы:

- QEMU:
  - Нет адекватной ARM64 конфигурации для запуска, соответствующей «железу». Система банально «валится» на запуске первых же сервисов, нужно отдельно конструировать userlevel чтобы фаззить.
  - Попробовали с «ванильным» ядром и busybox/libc - невероятно медленно, медленнее чем на планшете.
- Cuttlefish: «классический» эмулятор для AOSP (QEMU «под капотом»). Одна проблема - не то «железо», совсем другая конфигурация ядра. Пригоден для фаззинга ART.
- ARM Emulator: те же проблемы, что и QEMU - но чуть быстрее.

Результаты:

- **Смотрим в сторону Cuttlefish, но это не исключает «железного» фаззинга**. Преимущество в скорости работы пока неочевидно, но масштабировать кажется легче.

Фаззинг Android Runtime (libart.so) - «сердца» AOSP

LibArtClassVerifier:

- Основан на libfuzzer;
- Фаззит только одну функцию - парсинг DEX, но код не выполняет;
- Дает покрытие около 12%.

Фаззинг Android Runtime (libart.so) - «сердца» AOSP

LibArtClassVerifier:

- Основан на libfuzzer;
- Фаззит только одну функцию - парсинг DEX, но код не выполняет;
- Дает покрытие около 12%.

Google dexfuzzer (нуждается в доработке):

- Написан на Java, работает крайне медленно;
- Фаззинг DEX парсера на каждый проход генерит файлик 12MB;
- Не сохраняется корпус - каждый раз фаззинг начинается с первичного корпуса;
- Очень медленно растет покрытие (~20%), хотя вроде как мутации производятся.

Фаззинг Android Runtime (libart.so) - «сердца» AOSP

LibArtClassVerifier:

- Основан на libfuzzer;
- Фаззит только одну функцию - парсинг DEX, но код не выполняет;
- Дает покрытие около 12%.

Google dexfuzzer (нуждается в доработке):

- Написан на Java, работает крайне медленно;
- Фаззинг DEX парсера на каждый проход генерит файлик 12MB;
- Не сохраняется корпус - каждый раз фаззинг начинается с первичного корпуса;
- Очень медленно растет покрытие (~20%), хотя вроде как мутации производятся.

Выводы:

- Публичной информации о фаззинге ART мы не нашли, так что вроде бы мы первые.

Фаззинг Android Framework и приложений

Jazzer - идеальное решение!

- Можно запускать где угодно - в том числе и на «железе»;
- Единственный инструмент для фаззинга Java/Kotlin кода;
- Скорость работы сравнима с libfuzzer для нативного кода.

Выводы:

- Тут особых сложностей нет, все идет по плану :)

Выводы

AOSP - прекрасная основа для надежной и безопасной мобильной (и не только :) Операционной Системы.

Делать ОС на AOSP - можно и нужно, но осознанно и с РБПО.

Полезные ссылки

- Официальный сайт AOSP: https://source.android.com/docs/core/architecture
- Как используется ядро Linux в AOSP: https://source.android.com/docs/core/architecture/kernel
- Отличное видео про Binder: https://www.youtube.com/watch?v=Jgampt1DOak
- Определение релевантности CVE ядра по конфигу: https://github.com/Laerdal/vex-kernel-checker
- Определение статуса AOSP CVE в исходном коде: https://github.com/google/vanir

yadro.com

123376, Москва г., Рочдельская ул., дом 15, строение 15

a.dubinin@yadro.com