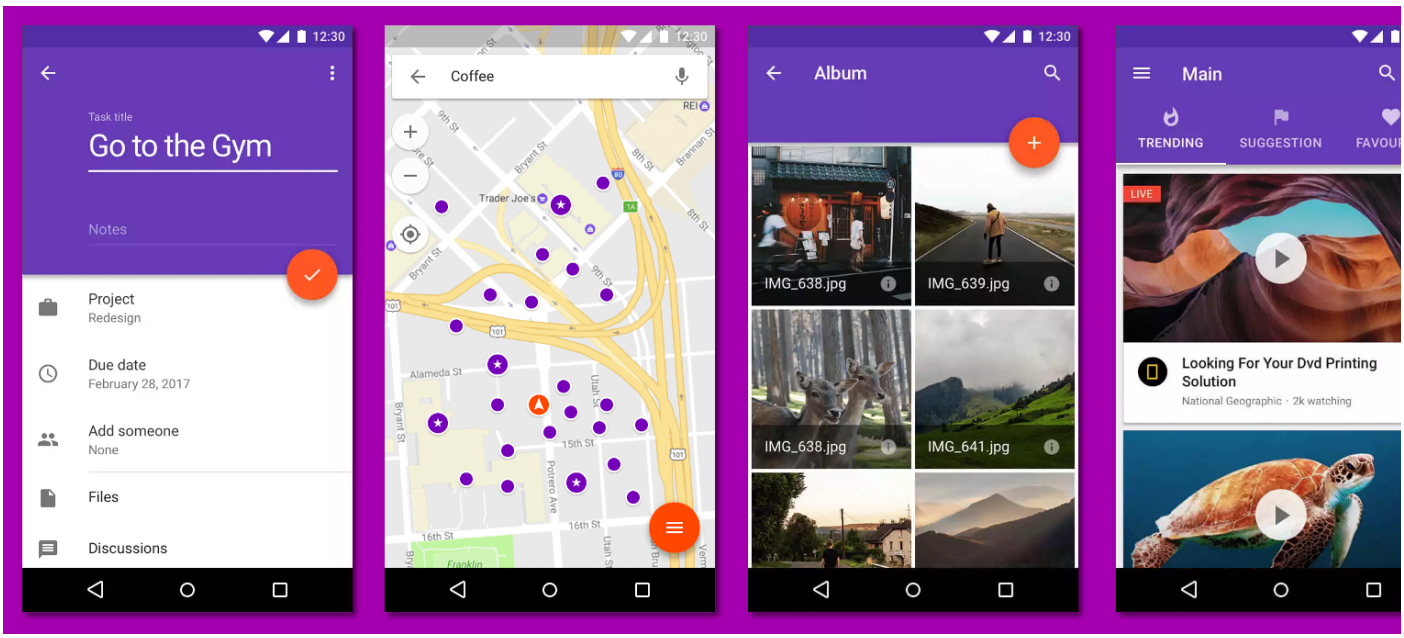
401,35
Рейтинг

Solar Security
Безопасность по имени Солнце

bugaevc 20 сентября в 14:15 Разработка

Как работает Android, часть 2

Разработка под Android, Блог компании Solar Security



В этой статье я расскажу о некоторых идеях, на которых построены высокоуровневые части Android, о нескольких его предшествен и о базовых механизмах обеспечения безопасности.

Статьи серии:

- Как работает Android, часть 1
- Как работает Android, часть 2
- Как работает Android, часть 3
- ...

Говоря про Unix- и Linux-корни Android, нужно вспомнить и о других проектах операционных систем, влияние которых можно прос в Android, хотя они и не являются его прямыми предками.

Я уже упомянул про BeOS, в наследство от которой Android достался Binder.

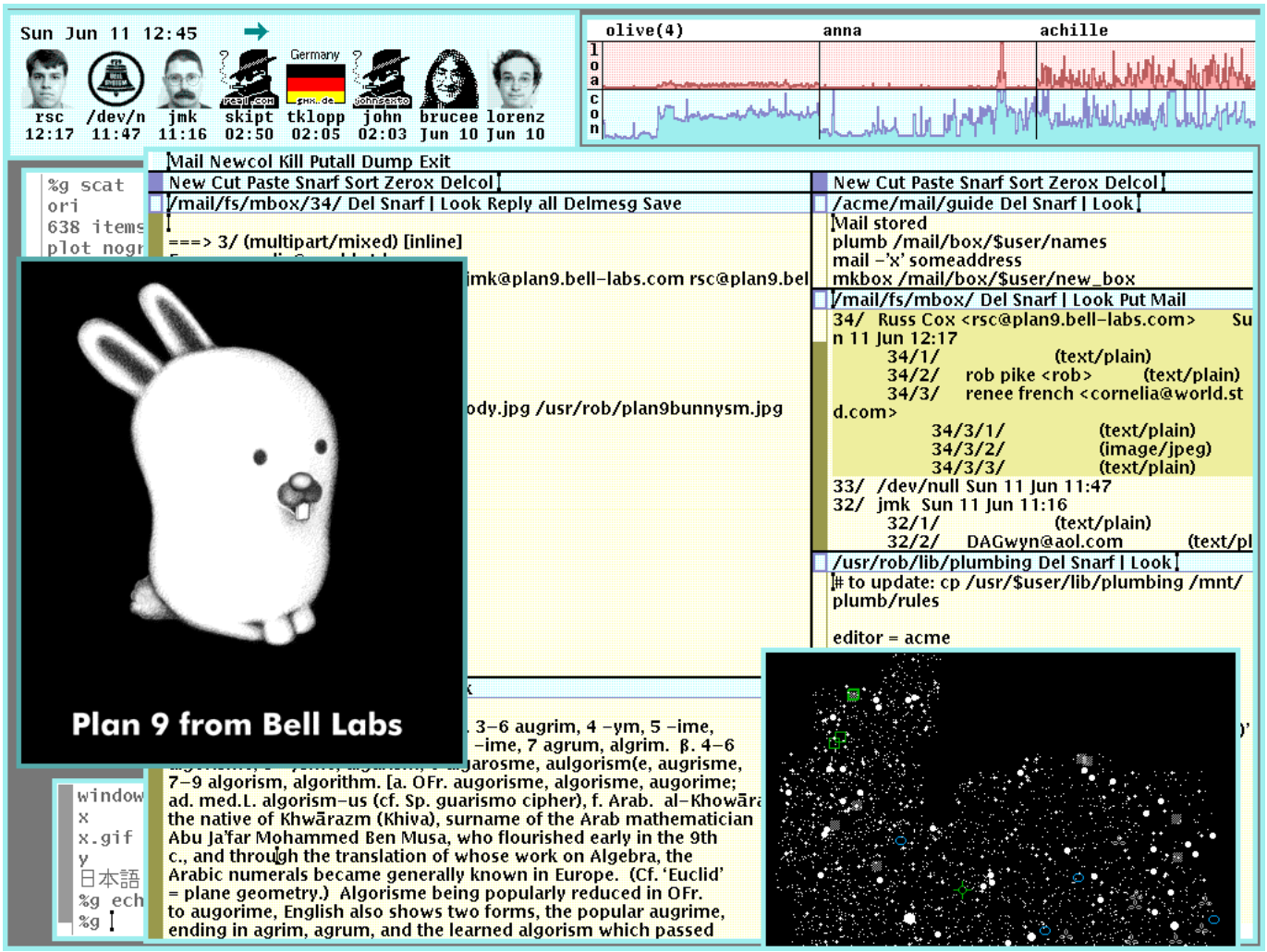
Plan 9 from Bell Labs

Plan 9 — потомок Unix, логическое продолжение, развитие его идей и доведение их до совершенства. Plan 9 был разработан в Bell той же командой, которая создала Unix и C — над ним работали такие люди, как Ken Thompson, Rob Pike, Dennis Ritchie, Brian Kernig Tom Duff, Doug McIlroy, Bjarne Stroustrup, Bruce Ellis и другие.

В Plan 9 взаимодействие процессов между собой и с ядром системы реализовано не через многочисленные системные вызовы и механизмы IPC, а через виртуальные текстовые файлы и файловые системы (развитие принципа Unix **«всё — это файл»**). При этом

каждая группа процессов «видит» файловую систему по-своему (**пространства имён**, namespaces), что позволяет запускать разн части системы в разном окружении.

Например, чтобы получить позицию курсора мыши, приложения читают текстовый файл `/dev/mouse`. Оконная система rio предостав каждому приложению свою версию этого файла, в которой видны только события, относящиеся к окну этого приложения, и используются локальные по отношению к окну координаты. Сама rio читает события «настоящей» мыши через такой же файл `/dev/` — в том виде, в котором его видит она. Если она запущена напрямую, этот файл предоставляется ядром и действительно описывае движения настоящей мыши, но она может быть совершенно прозрачно запущена в качестве приложения под другой копией rio, без какой-то специальной поддержки с её стороны.



Plan 9 полностью поддерживает доступ к удалённым файловым системам (используется собственный протокол 9P, кроме того, поддерживаются FTP и SFTP), что позволяет программам совершенно прозрачно получать доступ к удалённым файлам, интерфейс ресурсам. Такая «родная» **сетевая прозрачность** превращает Plan 9 в *распределённую* операционную систему — пользователь м физически находиться за одним компьютером, на котором запущена rio, запускать приложения на нескольких других, использоват них файлы, хранящиеся на файловом сервере и выполнять вычисления на CPU-сервере — всё это полностью прозрачно и без специальной поддержки со стороны каждой из частей системы.

За счёт красиво спроектированной архитектуры Plan 9 значительно проще и меньше, чем Unix — на самом деле ядро Plan 9 даже в несколько раз меньше известного микроядра Mach.

> Perfection is achieved not when there is nothing more to add, but when there is nothing left to take away.

Несмотря на техническое превосходство и наличие слоя совместимости с Unix, Plan 9 не получил широкого распространения. Тем менее, многие идеи и технологии из Plan 9 получили распространение и были реализованы в других системах. Самая известная из — кодировка UTF-8, которая была разработана в Plan 9 для обеспечения полной поддержки Unicode при сохранении обратной совместимости с ASCII — стала общепринятым стандартом.

Больше всего идей и технологий из Plan 9 реализовано в Linux:

- файловая система `/proc` (procfs)

- системный вызов `clone` (аналог `rfork` из Plan 9)
- поддержка пространств имён монтирования (mount namespaces)
- поддержка файловых систем, реализованных в пользовательском пространстве (filesystem in userspace, FUSE)
- поддержка протокола 9P

Многое из этого используется, в том числе, и в Android. Кроме того, в Android реализован механизм intent'ов, похожий на plumber и 9; о нём я расскажу в следующей статье.

Про Plan 9 можно узнать подробнее на сайте plan9.bell-labs.com (сохранённая копия в Wayback Machine), или его зеркале 9p.io

Inferno

Plan 9 получил продолжение в виде проекта Inferno, тоже разработанного в Bell Labs. К таким свойствам Plan 9, как простота и распределённость, Inferno добавляет *переносимость*. Программы для Inferno пишутся на высокоуровневом языке Limbo и выполняк с использованием just-in-time компиляции — встроенной в ядро Inferno виртуальной машиной.

Inferno настолько переносим, что может запускаться

- на процессорах разных архитектур: ARM, x86, IBM PowerPC, Sun SPARC, 6SGI MIPS и HP PA-RISC,
- как самостоятельная операционная система или как программа под Plan 9, Unix, Windows 95 и Windows NT.

При этом приложениям, запущенным внутри Inferno, предоставляется совершенно одинаковое окружение.

Inferno получил ещё меньше распространения и известности, чем Plan 9. С другой стороны, Inferno во многом предвосхитил Androi самую популярную операционную систему на свете.

Danger

Компания Danger Research Inc. была сооснована Энди Рубином (Andy Rubin) в 1999 году, за 4 года до сооснования им же Android In 2004 году.

В 2002 году Danger выпустили свой смартфон Danger Hiptop. Многие из разработчиков Danger впоследствии работали над Android, поэтому неудивительно, что его операционная система была во многом похожа на Android. Например, в ней были реализованы:

- «всегда запущенные» приложения, написанные на Java,
- полноценный веб-браузер,
- веб-приложения,
- мессенджер,
- email client,
- облачная синхронизация,
- магазин сторонних приложений.

Подробнее о Danger можно прочитать в статье Chris DeSalvo, одного из разработчиков, под названием The future that everyone forg

Java

Хотя использование высокоуровневых языков для серьёзной разработки сейчас уже никого не удивляет, из популярных операцион систем только у Android «родной» язык — высокоуровневая Java (с другой стороны, здесь можно вспомнить веб с его JavaScript, .N для Windows и относительно высокоуровневый — но полностью компилируемый в нативный код и не использующий сборку мусора Swift).

Несмотря на кажущиеся недостатки («Java сочетает в себе красоту синтаксиса C++ со скоростью выполнения питона»), Java обла множеством преимуществ.

Во-первых, Java — самый популярный (с большим отрывом) язык программирования. У Java огромная экосистема библиотек и инструментов разработки (в том числе систем сборки и IDE). Про Java написано множество статей, книг и документации. Наконец, существует множество квалифицированных Java-разработчиков.

Программы на Java, как и на многих других высокоуровневых языках, переносимы между операционными системами и архитектур процессора («Write once, run anywhere»). Практически это проявляется, например, в том, что приложения для Android работают *бе перекомпиляции* на устройствах любой архитектуры (Android поддерживает ARM, ARM64, x86, x86–64 и MIPS).

В отличие от низкоуровневых языков вроде C и C++, использующих ручное управление памятью, в Java память автоматически управляется средой времени выполнения (runtime environment). Программа на Java даже не имеет прямого доступа к памяти, что автоматически предотвращает несколько классов ошибок, часто приводящих к падениям и уязвимостям в программах, написанны низкоуровневых языках — невозможны «висячие ссылки» (из-за которых происходит use-after-free), разыменование нулевого указ (при попытке это сделать выбрасывается `NullPointerException`), чтение неинициализированной памяти и выход за границы массив

Использование полноценной сборки мусора (по сравнению с automatic reference counting) избавляет программиста от всех пробле сложностей с циклическими ссылками и позволяет реализовывать ещё более продвинутые (advanced) зависимости между объекта

Это делает разработку под Android более приятной, чем разработку с использованием низкоуровневых языков, а приложения под Android гораздо более надёжными, в том числе и точки зрения безопасности.

Running Java is ART

В отличие от большинства других высокоуровневых языков, программы на Java не распространяются в виде исходного кода, а компилируются в промежуточный формат (**байткод**, bytecode), который представляет собой исполняемый бинарный код для специального процессора.

Хотя делаются попытки создать физический процессор, который исполнял бы Java-байткод напрямую, в подавляющем большинств случаев в качестве такого процессора используется эмулятор — **Java virtual machine** (JVM). Обычно используется реализация от Oracle/OpenJDK под названием HotSpot.

В Android используется собственная реализация под названием **Android Runtime** (ART), специально оптимизированная для работы мобильных устройствах. В старых версиях Android (до 5.0 Lollipop) вместо ART использовалась другая реализация под названием **D**

И в Dalvik, и в ART используется собственный формат байткода и собственный формат файлов, в которых хранится байткод — DEX executable). В отличие от class-файлов в «обычной джаве», весь Java-код приложения обычно компилируется в один DEX-файл `classes.dex`. При сборке Android-приложения Java-код сначала компилируется обычным компилятором Java в class-файлы, а потом конвертируется в DEX-файл специальной утилитой (возможно и обратное преобразование).

И HotSpot, и Dalvik, и ART дополнительно оптимизируют выполняемый код. Все три используют just-in-time compilation (JIT), то есть время выполнения компилируют байткод в куски полностью нативного кода, который выполняется напрямую. Кроме очевидного выигрыша в скорости, это позволяет оптимизировать код для выполнения на конкретном процессоре, не отказываясь от полной переносимости байткода.

Кроме того, ART может компилировать байткод в нативный код заранее, а не во время выполнения (ahead-of-time compilation) — пр система автоматически планирует эту компиляцию на то время, когда устройство не используется и подключено к зарядке (напри ночью). При этом ART учитывает данные, собранные профилировщиком во время предыдущих запусков этого кода (profile-guided optimization). Такой подход позволяет дополнительно оптимизировать код под специфику работы конкретного приложения и даже особенности использования этого приложения именно этим пользователем.

В результате всех этих оптимизаций производительность Java-кода на Android не сильно уступает производительности низкоуров кода (на C/C++), а в некоторых случаях и превосходит её.

Java-байткод, в отличие от обычного исполняемого кода, использует объектную модель Java — то есть в байткоде явно записываю такие вещи, как классы, методы и сигнатуры. Это делает возможной компиляцию *других языков* в Java-байткод, что позволяет написанным на них программам исполняться на виртуальной машине Java и быть в той или иной степени совместимыми (interopera Java.

Существуют как JVM-реализации независимых языков — например, Jython для Python, JRuby для Ruby, Rhino для JavaScript и диале Lisp Clojure — так и языки, исходно разработанные для компиляции в Java-байткод и выполнения на JVM, самые известные из кото Groovy, Scala и Kotlin.

Самый новый из них, **Kotlin**, специально разработанный для идеальной совместимости с Java и обладающий гораздо более приятн синтаксисом (похожим на Swift), поддерживается Google как официальный язык разработки под Android наравне с Java.

Несмотря на все преимущества Java, в некоторых случаях всё-таки желательно использовать низкоуровневый язык — например, д реализации критичного по производительности компонента, такого как браузерный движок, или чтобы использовать существующу нативную библиотеку. Java позволяет вызывать нативный код через Java Native Interface (JNI), и Android предоставляет специальны средства для нативной разработки — Native Development Kit (NDK), в который входят в том числе заголовочные файлы, компилятор (Clang), отладчик (LLDB) и система сборки.

Хотя NDK в основном ориентирован на использование C/C++, с его помощью можно писать под Android и на других языках — в том числе Rust, Swift, Python, JavaScript и даже Haskell. Больше того, есть даже возможность портировать iOS-приложения (написанные Objective-C или Swift) на Android практически без изменений.

О безопасности

Классический Unix

Модель безопасности в классическом Unix основана на системе **UID/GID** — специальных номеров, которые ядро хранит для каждо процесса. Процессам с одинаковым UID разрешён доступ друг к другу, процессы с разным UID защищены друг от друга. Аналогич

ограничивается доступ к файлам.

По смыслу каждый UID (user ID) соответствует своему пользователю — во времена создания Unix была нормальной ситуация, когда компьютер одновременно использовался множеством людей. Таким образом, в Unix процессы и файлы разных *людей* были защищ друг от друга. Чтобы разрешить общий доступ к некоторым файлам, пользователи объединялись в группы, которым и соответствов (group ID).

При этом всем программам, запускаемым пользователем, даётся полный доступ ко всему, к чему есть доступ у этого пользователя Собственно, поскольку пользователь не может общаться с ядром напрямую, а взаимодействует с компьютером через shell и други процессы — права пользователя *и есть* права программ, запущенных от его имени.

Такая модель подразумевает, что пользователь полностью доверяет всем программам, которые использует. В то время это было логично, потому что программы чаще всего либо были частью системы, либо создавались (писались и компилировались) самим пользователем.

В Unix есть и исключение из ограничений доступа — UID 0, который принято называть **root**. У него есть доступ ко всему в системе, никакие ограничения на него не распространяются. Этот аккаунт использовался системным администратором; кроме того, под UID запускаются многие системные сервисы.

В современном Linux эта модель была значительно расширена и обобщена, в том числе появились capabilities, позволяющие «полу часть root-прав», и реализующая мандатное управление доступом (mandatory access control, MAC) подсистема SELinux, которая позволяет дополнительно ограничить права (в том числе права root-процессов).

Всё изменилось

За несколько десятков лет, прошедших с создания Unix до создания Android, практика использования компьютеров («вычислителей значительно изменилась.

Вместо машин, рассчитанных на параллельное использование многими пользователями (через *терминалы* — то, что сейчас эмули *эмуляторы терминалов*), появились **персональные компьютеры**, рассчитанные на использование одним человеком. Компьютеры перестали быть лишь рабочим инструментом и стали центром нашей цифровой жизни. С появлением **мобильных устройств** — сна КПК, потом смартфонов, планшетов, умных часов и т.п. — эта тенденция только усилилась (потому что заниматься рабочими вопро на мобильных устройствах относительно неудобно).

На таких устройствах хранятся гигабайты персональной информации, доступ к которой должен быть защищён и ограничен. В то же время расцвёл рынок сторонних приложений, которым у пользователя нет никаких оснований доверять.

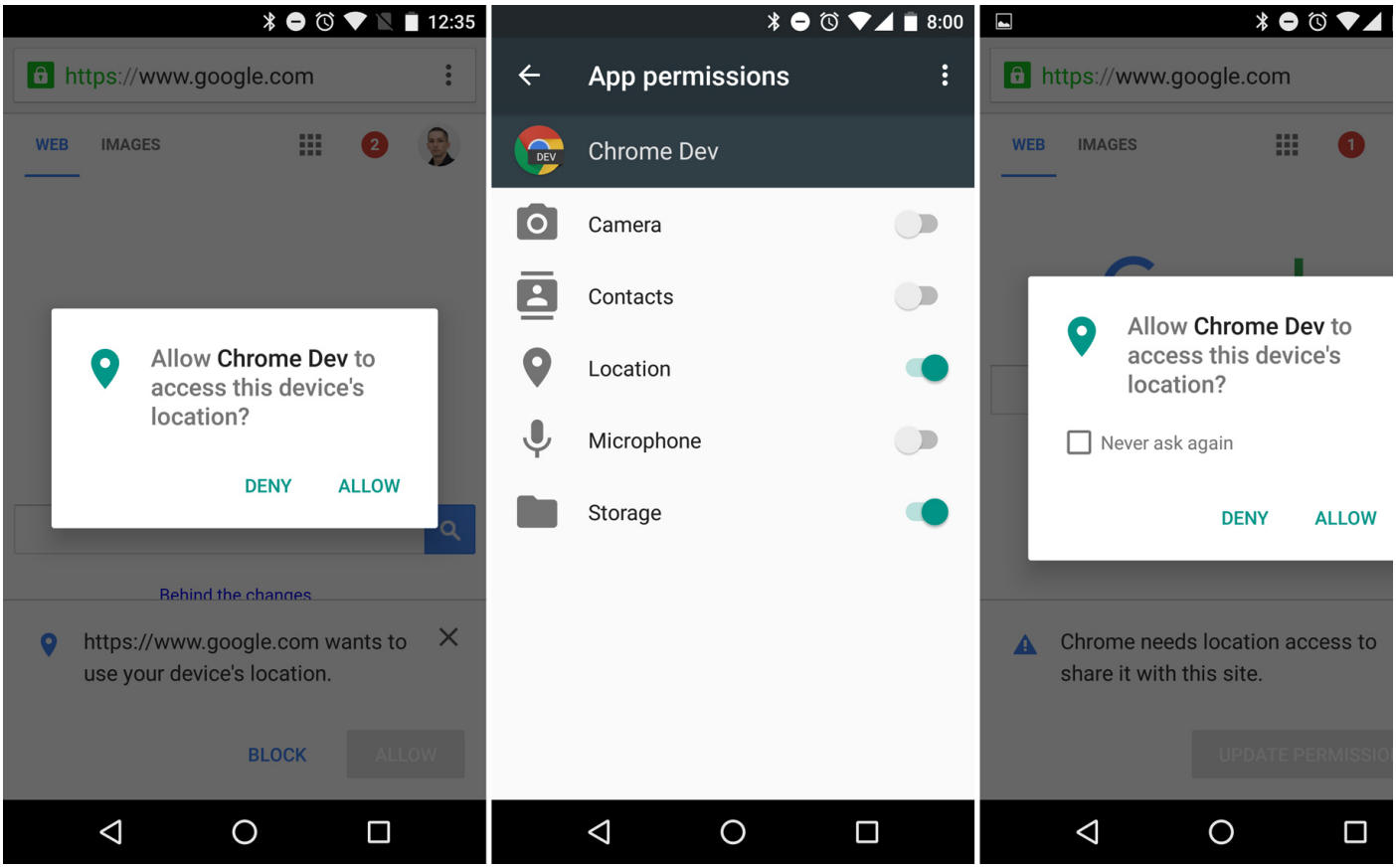
Таким образом, в современных условиях вместо защиты разных пользователей друг от друга необходимо защищать от приложени другие приложения, пользовательские данные и саму систему. Кроме того, широкое распространение получили вирусы, которые о используют уязвимости в системе — для защиты от них нужно дополнительно защищать части системы друг от друга, чтобы использование одной уязвимости не давало злоумышленнику доступ ко всей системе.

Android

Хотя часть Android-приложений поставляется с системой — например, такие стандартные приложения, как Калькулятор, Часы и Ка — большинство приложений пользователи устанавливают из сторонних источников. Самый известный из них — Google Play Store, н есть и другие, например, F-Droid, Amazon Appstore, Яндекс.Store, китайские Baidu App Store, Xiaomi App Store, Huawei App Store и т. Кроме того, Android позволяет вручную устанавливать произвольные приложения из APK-файлов (это называют sideloading).

Как и другие Unix-подобные системы, Android использует для ограничения доступа существующий механизм UID/GID. При этом — отличие от традиционного использования, когда UID соответствуют пользователям — в Android разные UID соответствуют разным *приложениям*. Поскольку процессы разных приложений запускаются с разными UID, уже на уровне ядра приложения защищены и изолированы друг от друга и не имеют доступа к системе и данным пользователя. Это образует **песочницу** (Application Sandbox) и позволяет пользователю устанавливать любые приложения без необходимости доверять им.

Чтобы всё-таки получить доступ к пользовательским данным, камере, совершению звонков и т.п., приложение должно получить от пользователя разрешение (**permission**). Некоторые из разрешений существуют в виде GID, в которые приложение добавляется, ког получает это разрешение — например, получение разрешения `ACCESS_FM_RADIO` помещает приложение в группу `media`, что позволяе получить доступ к файлу `/dev/fm`. Остальные существуют только на более высоком уровне (в виде записей в файле `packages.xml`) и проверяются другими компонентами системы при обращении к высокоуровневому API через Binder.



Небольшая часть системных сервисов в Android запускается под UID 0, то есть root, но большинство используют специально выдел номера UID, повышая при необходимости свои права с помощью Linux capabilities. Кроме того, Android использует SELinux — использование SELinux в Android называют SEAndroid — для ещё большего ограничения того, какие действия разрешено выполнять приложениям и системным сервисам.

Обычно Android не предоставляет пользователю прямой доступ к root-аккаунту, но в некоторых случаях у него есть возможность э доступ получить. Как это происходит, зачем это нужно и какими опасностями это грозит я расскажу позднее.

В следующей статье (которая выходит уже через неделю) я расскажу о компонентах, из которых состоят приложения под Android, и идеях, которые стоят за этой архитектурой.

Метки: android internals, android, linux, plan 9, inferno, danger, java, kotlin, bytecode, dalvik, jit

+71 310 34,5k 34

Solar Security 401,35
Безопасность по имени Солнце

36,0 Карма 101,7 Рейтинг 92 Подписчики

Сергей Бугаев @bugaevc
Разработчик

Facebook

3 августа в 14:46

Как работает Android, часть 1

+90 76,9k 783 49

1 ноября 2016 в 11:08

Поиск уязвимостей в байткоде Java: что делать с результатами?

+10 4,4k 32 7

Комментарии 34

IvanOne 20.09.17 в 15:44

Продолжение ждал с нетерпением, уже думал что не выйдет! Спасибо, и пишите чаще :-)

nitrodenov 20.09.17 в 17:05

ART использует AOT компиляцию

> И HotSpot, и Dalvik, и ART дополнительно оптимизируют выполняемый код. Все три используют just-in-time compilation (JIT), то есть во вре выполнения компилируют байткод в куски полностью нативного кода, который выполняется напрямую.

bugaevc 20.09.17 в 17:11

Да, я про это написал дальше:

> Кроме того, ART может компилировать байткод в нативный код заранее, а не во время выполнения (ahead-of-time compilation)

До 7.0 Nougat ART не поддерживала JIT, только AOT (в отличие от Dalvik, который поддерживал только JIT), поэтому во многих статьях пе с Dalvik на ART описывали как переход с JIT на AOT. На самом деле это полностью новый рантайм с гораздо более продвинутой инфраструктурой для оптимизации; просто начали с реализации AOT, а потом реализовали и JIT, и, как они сами это называют, all-the-tim

nitrodenov 20.09.17 в 20:34

Не знал это, спасибо за инфу!

ArtRoman 21.09.17 в 14:08

В самых ранних версиях андроида (до 2.2) отсутствовал даже JIT, а его появление было большим событием.

jatx 20.09.17 в 23:37

Спасибо за статью. Хоть я и обладаю некоторым опытом в практической разработке под Андроид, но многое из написанного узнаю впервы

bukov_georgiy 21.09.17 в 08:05

Наконец то вторая часть, очень интересно. Будут ли подобные статьи про ios?

bugaevc 21.09.17 в 11:27

Спасибо!

Нет — к сожалению планов писать про iOS пока нет, но если вы интересуетесь, могу порекомендовать книгу Mac OS X and iOS Internals: T Apple's Core, автор Jonathan Levin.

bukov_georgiy 22.09.17 в 03:35

Спасибо большое.
Ну тогда ждем еще много статей про android

QtRoS 21.09.17 в 09:23

Отличная статья, пожалуйста продолжайте!

yamilov 21.09.17 в 09:51

Кратко, структурированно, интересно. Буду с нетерпением ждать третью часть.

kreo_OL 21.09.17 в 09:51

Круто! Ждал вторую статью, теперь подожду и третью.
На сколько статей рассчитан цикл?

> **bugaevc** 21.09.17 в 11:24
>
> Спасибо! У меня есть идеи для пяти статей (то есть ещё трёх), но может получиться и больше.

a40 21.09.17 в 11:02

Очень интересно, спасибо!

SPoket 21.09.17 в 11:02

Действительно очень хорошая статья. Спасибо, жду продолжения! :)

UMAX 21.09.17 в 15:17

Спасибо, очень интересно всё написано (хотя я и не разработчик). А как пользователя (ещё с ANdroid 1.6) меня крайне интересует, почему практически убрали в андроиде возможность устанавливать/переносить приложения на карту памяти.
С точки зрения безопасности данных приложений не вижу в этом смысла, ведь можно данные шифровать на случай кражи карты памяти.
С точки зрения возможного замедления работы приложений тоже проблем не вижу — загрузка приложения на 1 секунду дольше меня не напрягла бы, в отличие от того, что у меня в телефоне есть карточка на 32 Гб, а смартфон ругается, что ему не хватает памяти и нет возможности установить ещё хотя бы одно маленькое приложение.
Как вариант, остаётся только маркетинг (покупайте более дорогие телефоны с большим количеством встроенной памяти), но для самого Гу разработчиков это плохая практика — я не могу себе позволить купить некоторые платные приложения и игры не потому, что не хочу или у нет денег, а потому что понимаю, что мне их просто не установить наряду с теми, которые мне нужны постоянно.

> **bugaevc** 21.09.17 в 16:51
>
> Возможность устанавливать приложения на SD-карту никуда не убрали, но это opt-in со стороны приложения — по умолчанию `installLocation="internalOnly"`. Да, при этом данные приложения помещаются в специальный зашифрованный asec-контейнер на карт
> Как написано в документации, это в основном актуально для больших игр.
>
> > **UMAX** 21.09.17 в 16:56
> >
> > Ну насколько я понял, это не только от разработчика зависит. Некоторые приложения на одних устройствах и прошивках переносятся карту (но только частично), а на другом устройстве или с другой прошивкой уже не переносятся.
> > У меня на старом HTC Desire на Android 2.3.3 была прошивка, на которой приложения переносились на карту памяти полностью. В итог было одновременно установлено под полторы сотни приложений и игр. Сейчас на Samsung J3 2016 я никак не могу найти прошивку, которая позволила бы делать то же самое. С приложениями типа Data2SD и с рутованными прошивками, через некоторое время карть памяти умирают или возникают постоянные ошибки — разделы карты внезапно теряются, приложения отваливаются и т.п. Пользовать нормальном режиме не получается.
> >
> > > **bugaevc** 21.09.17 в 17:03
> > >
> > > > через некоторое время карты памяти умирают или возникают постоянные ошибки — разделы карты внезапно теряются, приложен отваливаются и т.п. Пользоваться в нормальном режиме не получается.
> > >
> > > Вот именно поэтому это и opt-in. Да, кастомные сборки могут насильно переносить приложение, даже если оно не рассчитано на т его части начнут внезапно отваливаться.
> > >
> > > > **UMAX** 21.09.17 в 17:06
> > > >
> > > > Вот в том то и дело. А на HTC в той прошивке всё работало абсолютно стабильно. И на x-pda постоянные вопросы в темах про прошивки именно про переносимость приложений. Т.е., людям это нужно. Почему по умолчанию в стандартной поставке андрои сделать это включённым и нормально работающим?
> > > >
> > > > > **nikolayv81** 22.09.17 в 22:17
> > > > >
> > > > > Вроде несколько лет назад озвучивалась позиция отказа от sd, основная идея насколько помню защита от взлома приложений принципе та же причина что и отсутствие root по умолчанию), да при этом обычно говорят о защите пользователей но что-то подсказывает что дело в другом.

Ktulkhu_Triediniy 21.09.17 в 15:44

> над ним работали такие люди, как Ken Thompson, Rob Pike, Dennis Ritchie, Brian Kernighan, Tom Duff, Doug McIlroy, Bjarne Stroustrup, Bruce E и другие.

Не переводить спец. термины и названия — норма. Не переводить имена — моветон. В остальном, цикл статей великолепен. С нетерпением продолжения. Спасибо.

bugaevc 21.09.17 в 16:29

Я специально в этой серии статей почти везде пишу имена в оригинале — на мой взгляд, так получается лучше. Другими словами, не баг фича :)

Vitls 22.09.17 в 10:29

Хм… вот какая мысль. Технологиий JIT и AOT известны достаточно давно. Почему никто до сих пор не реализовал идею: компилировать бай в нативный на этапе инсталляции приложения в систему и затем по требованию запускать уже скомпилённый код. Это же сэкономит кучу машинного времени за такую дорогостоящую операцию как запуск приложения.

bugaevc 22.09.17 в 10:29

То, что вы описали, и есть AOT.

ArtRoman 29.09.17 в 01:51

Да, но тот обязательный AOT при установке аукнулся долгой установкой приложений (и их обновлений), и очень долгим процессом «оптимизации» всех установленных приложений при обновлении прошивки. Сейчас же (Android 7.0 и выше) получился универсальный вариант.

Vitls 22.09.17 в 15:53

Погодите, в случает AOT — приложение компилируется в нативный код перед запуском, точнее перед передачей управления, например при старте системы. Если приложение выгружено из памяти, при повторном старте оно либо будет заново компилироваться, либо будет взято и некоего кеша (кеш тоже может протухнуть и быть почищен). Я же говорю о том, что приложения, включая системные, компилировать именн этапе установки. Один раз скомпильнули и далее пользуемся только нативными бинарями.

bugaevc 22.09.17 в 17:03

AOT означает, что приложение компилируется до (ahead of) выполнения. Это можно делать сразу же при установке (и именно так это работало в Lollipop и Marshmallow) или в какое-то другое время между установкой и запуском (не обязательно прямо перед запуском; та работает, начиная с Nougat). Новый способ заметно лучше старого — в том числе и потому, что позволяет потом перекомпилировать приложение с учётом данных профилирования.

xoxol_89 23.09.17 в 10:39

Отлично написано! Подолжайте в том же духе!)

DieSlogan 25.09.17 в 15:32

> Некоторые из разрешений существуют в виде GID, в которые приложение добавляется, когда получает это разрешение — например, получение разрешения ACCESS_FM_RADIO помещает приложение в группу media, что позволяет ему получить доступ к файлу /dev/fm. Остальные существуют только на более высоком уровне (в виде записей в файле packages.xml)

Подскажите пожалуйста, с чем связан такой зоопарк и есть ли планы в roadmap по сведению всего в единую систему? Или там что-то системное, что дает именно такое решение ситуации?

bugaevc 25.09.17 в 19:03

Почему зоопарк? С точки зрения разработчика приложений все разрешения выглядят и работают одинаково.

С точки зрения низкоуровневой реализации часть разрешений проверяется ядром (разрешение на доступ к сети, на доступ к файлам устройств, на доступ к файлам пользователя…), поэтому они должны быть экспортированы на уровень ядра в виде GID (в packages.xml он тоже записаны, конечно). Для остальных разрешений это не нужно и неудобно (при изменении GID приложение нужно перезапустить, чт делается прозрачно для пользователя, но всё-таки лучше делать это реже).

bugaevc 25.09.17 в 19:02

(не туда)

Doomland 26.09.17 в 15:44

Огромное СПАСИБО за статьи про внутреннюю кухню Андроида. Очень познавательно и жду, как и многие здесь, продолжения.

Crandel 26.09.17 в 16:43

Уже есть продолжение :)

> Только полноправные пользователи могут оставлять комментарии. Войдите, пожалуйста.

САМОЕ ЧИТАЕМОЕ

Сутки | Неделя | Месяц

WhatsApp, что внутри?
+64 17,7k 119 41

Как Яндекс учит искусственный интеллект разговаривать с людьми
+80 14,7k 52 171

44 урока управления технарями
+31 13,6k 179 75

Как собеседовать инженеров-программистов
+25 11,3k 77 19

Как мы обманываем клиентов. Софт как Сервис
+19 9,9k 50 10

ИНТЕРЕСНЫЕ ПУБЛИКАЦИИ

Моделирование объектов, функций и операций. Мереологические отношения между объектами данного типа
+6 417 7 0

На пути к естественному интеллекту
+11 1,4k 19 4

Хаскель — ход конем II GT
+9 1,3k 6 4

Простая Scada на Python и Arduino
+11 2,7k 29 4

Sportiduino. Система электронной отметки для спортивного ориентирования. Часть 2 GT
+18 1,7k 16 0