Мы тут кое-что обновили. Чтобы «Хабр» работал корректно, обновите страницу.

Перезагрузить страницу

89.36
Рейтинг

Циан

В топ-6 лучших ИТ-компаний рейтинга Хабр.Карьера

Подписаться

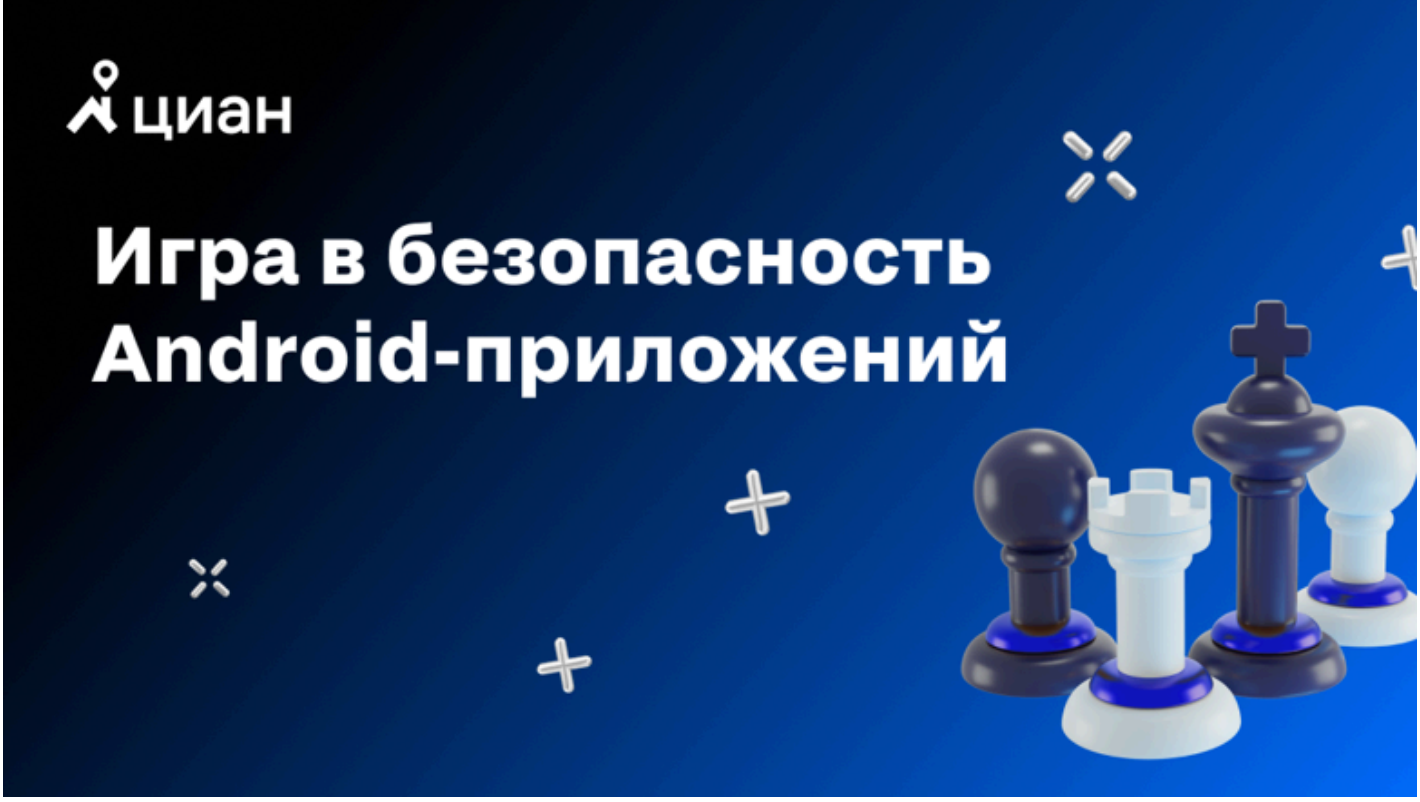
princeparadoxes 3 апр в 14:15

Игра в безопасность Android-приложений

Средний 20 мин 8.4K

Блог компании Циан, Информационная безопасность*, Разработка под Android*

FAQ

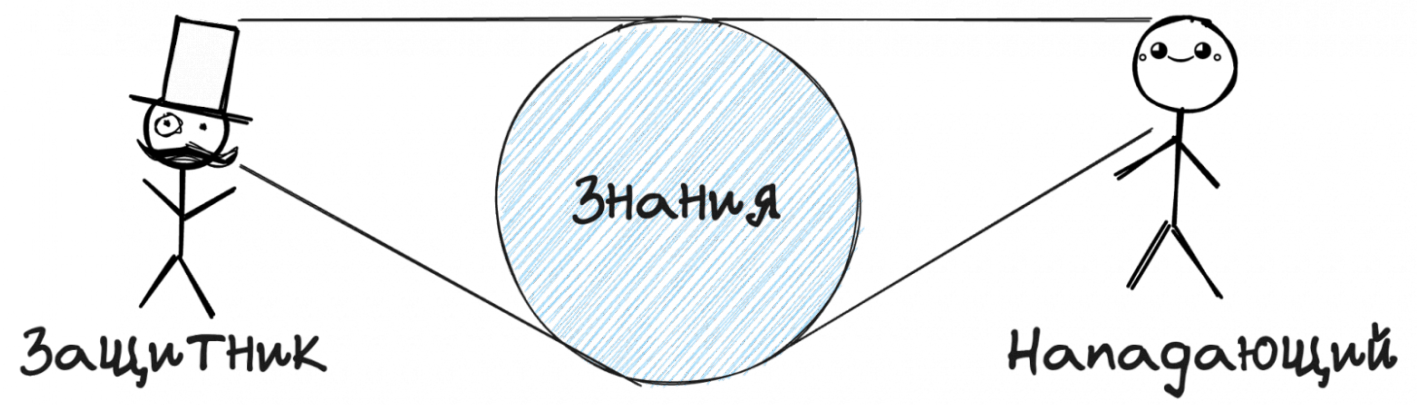


Давайте в общих чертах рассмотрим вопросы взлома и защиты Android-приложений.

В рамках статьи нас интересуют сами процессы взлома и защиты, а не конкретные методики работы с конкретными инструментами. Поэтому разберёмся с этими процессами и постараемся сделать выводы. Чтобы читать было интереснее, я решил попеременно ставить себя и на место нападающего, и на место защищающего приложение человека. Что-то вроде шахмат: сначала ход делает нападающий, а затем защищающийся. Пока кто-то не победит. Пройдём путь, постепенно наращивая сложность, от простого вроде HTTPS — к более сложному, вроде обфускации и деобфускации, изменению поведения. И под конец перейдём к C++ и просмотру его Assembler кода.

По всем правилам приличия представлюсь — меня зовут Перевалов Данил, а теперь давайте перейдём к теме.

Писать буду сразу и о взломе, и о защите, так как одни знания тяжело отделить от других. Если вы знаете, как проникнуть в систему, то вы, скорее всего, представляете, как от этого защититься.

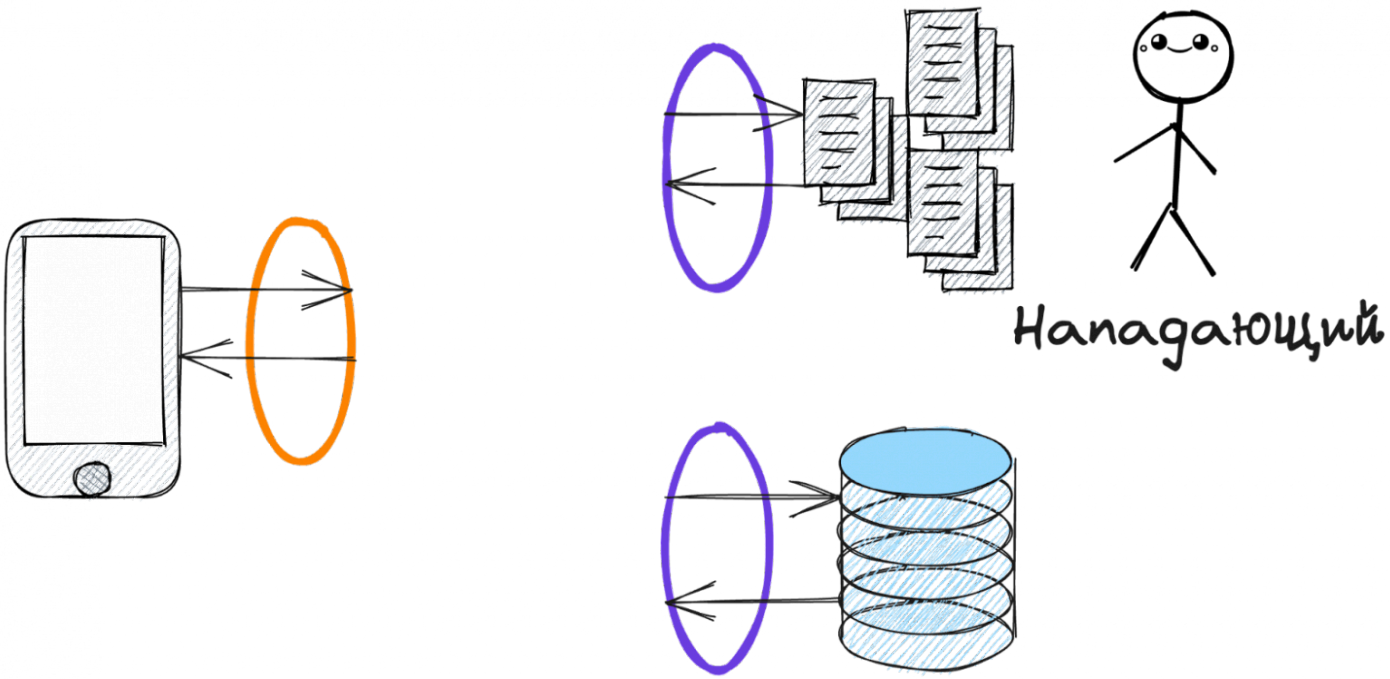


Начнём с самого попсового.

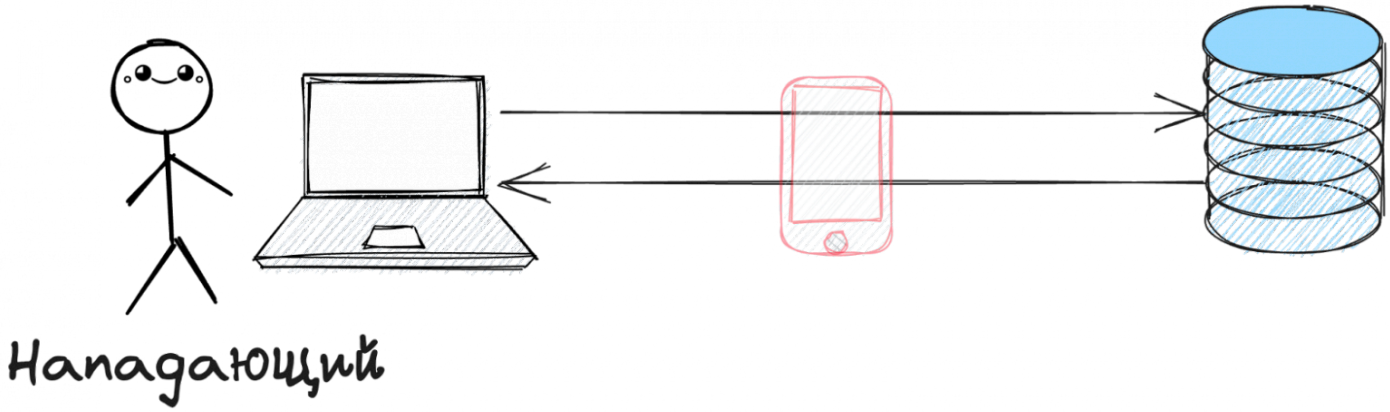
Просмотр данных

Основа данных для современных мобильных приложений — это трафик между сервером и собственно мобильным приложением. Сейчас не очень много приложений, которые работают полностью «офлайн».

Поэтому начнём с самого простого и понятного, с того, чем пользуется, наверное, практически каждый мобильный разработчик и тестировщик — с просмотра трафика. Ведь, по сути, каждому из нас надо понять, правильно ли я всё сделал во взаимодействии с сервером. По этой причине периодически свой трафик тоже приходится смотреть.



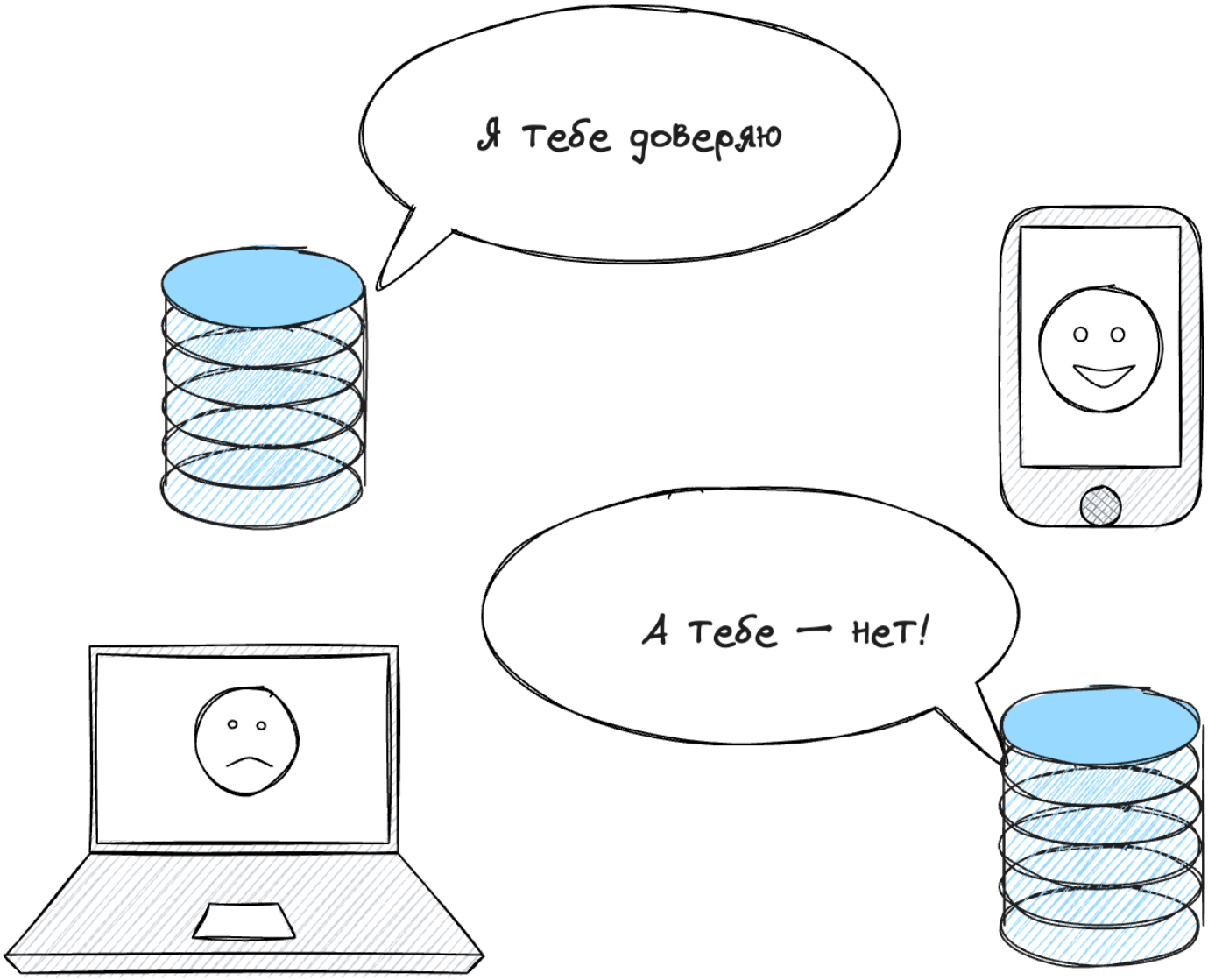
Помимо просмотра трафика для извлечения информации, часто трафик отлавливают, чтобы повторить этот самый трафик. Как бы делая вид, что это приложение пытается запросить данные. А на деле это бот, который, например, парсит наш контент.



Чтобы рассмотреть всё в рамках какого-то примера, предположим, что нападающий пытается написать парсер данных. Будем использовать приложение, так как со стороны веба сервер обычно лучше подготовлен к взломам.

Тут достаточно смешная ситуация — поскольку чаще парсить пытаются через Web, то со стороны мобильных приложений бэкенд нередко защищён куда меньше. Из-за этого

возникают ситуации, когда парсинг, имитирующий мобильное приложение, выгоднее, потому как защиты меньше.



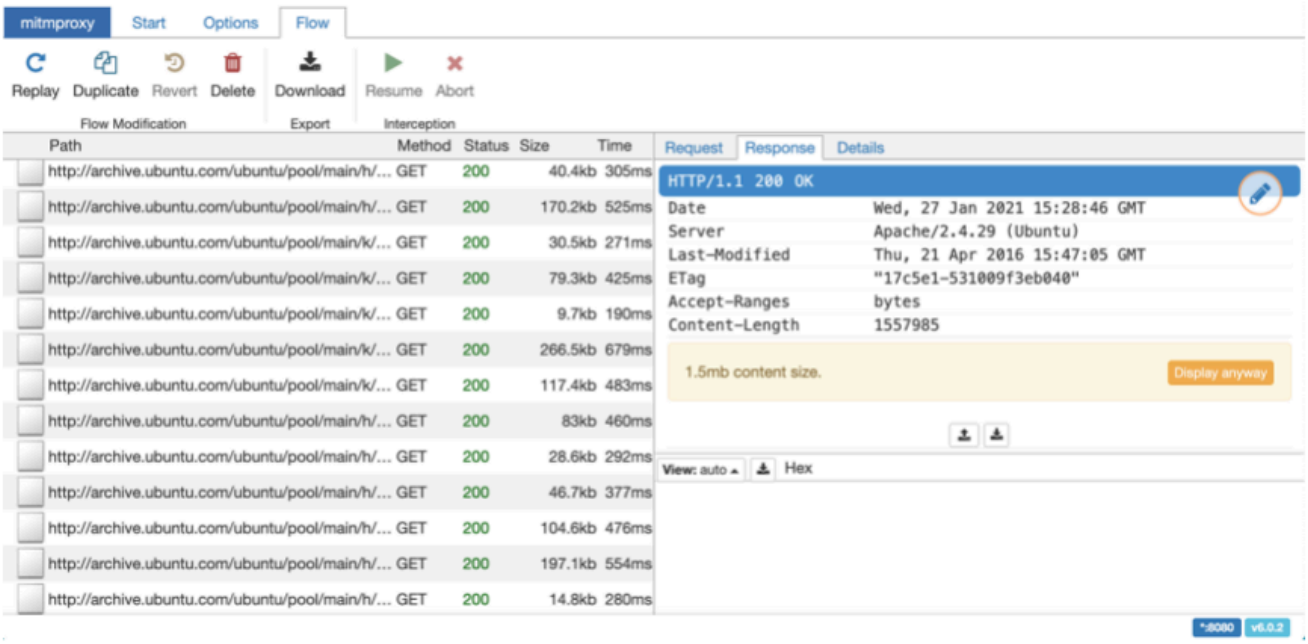
Итак, нападающий делает свой ход.

Ход нападающего. Незащищённый трафик

Есть множество программ для просмотра трафика: Charles, Burp, Mitmproxy и т. п. Я сконцентрирую своё внимание только на последней. Она бесплатная и, как следствие, доступна всем.

Логика таких программ простая: между мобильным приложением и бэкендом «врезается» Proxy-сервер. Трафик сначала идёт на Proxy-сервер в программу-перехватчик. Она, в свою очередь, пересылает данные на бэкенд, оставляя у себя копию, которую мы можем посмотреть.

Сразу обозначу, что любой HTTP-трафик с мобильного приложения не защищён, и его может посмотреть практически кто угодно, было бы желание. Поэтому начну с него. Мало ли, вдруг в нём пересылается что-то «интересное». Устанавливаем Mitmproxy, настраиваем и спокойно смотрим трафик приложения.



Это было очень просто. Теперь пытаемся защититься.

Ход защитника. HTTPS

Но и защититься от этого достаточно легко. Переводим все наши запросы на HTTPS. Это требует небольших усилий на бэкенде.

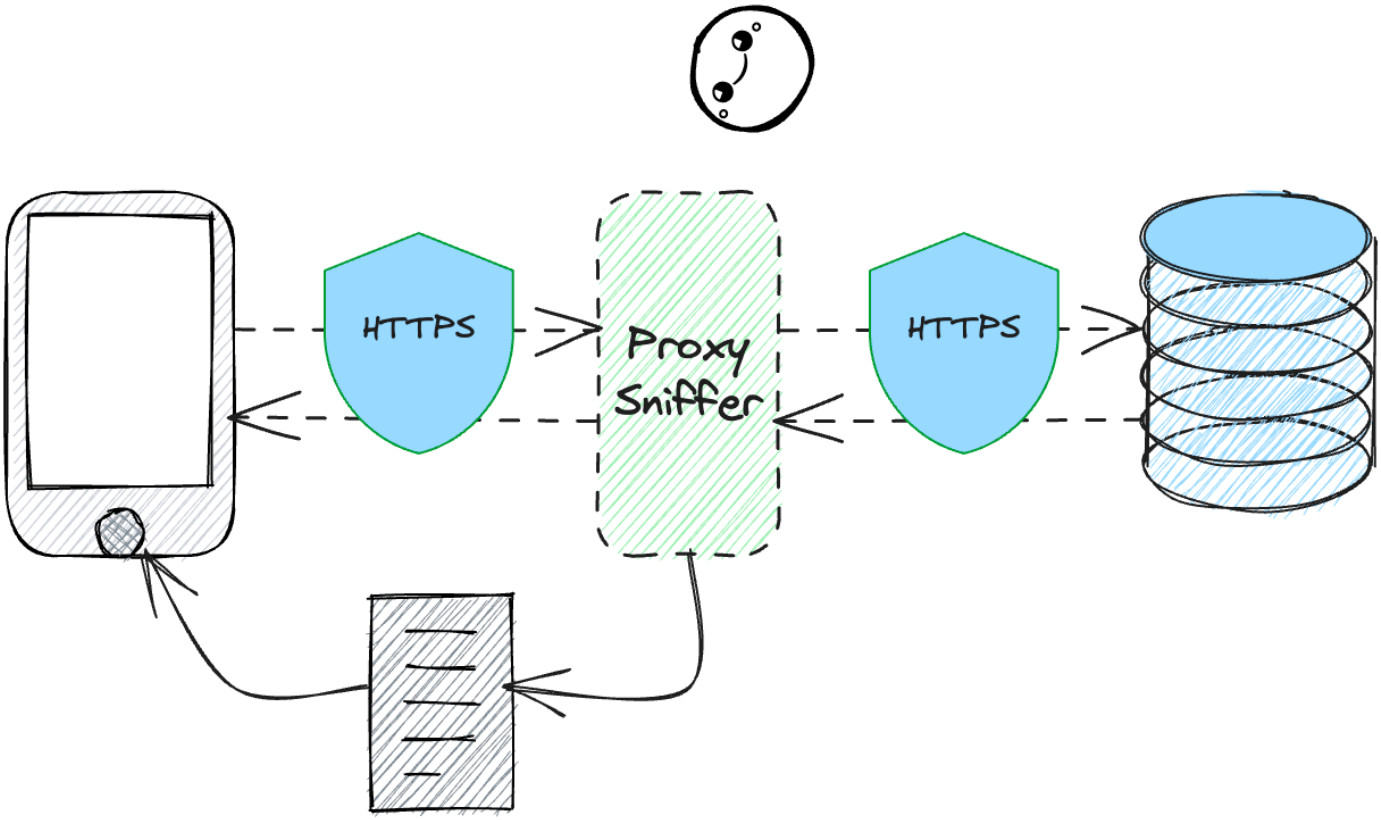
Казалось бы — всё круто, мы защитились. Нападающий не сможет посмотреть наш трафик.

Ход нападающего. Вредоносный сертификат

Но это не так. Мы просто создали дополнительную преграду. Возможность просматривать HTTPS-трафик всё ещё остаётся. Собственно, и Charles, и Mitmproxy предоставляют такую возможность.

Для этого достаточно добавить сгенерированный этими программами сертификат в хранилище сертификатов телефона. Однако если в старых версиях Android (Android 7 и ниже) можно подставить любой сертификат без особых проблем, то в более новых версиях потребуется получение root-прав или модификации Android Manifest приложения. Но в целом, как вы понимаете, получить root тоже достаточно легко. В крайнем случае можно установить эмулятор с Android 7.

После того как вредоносный сертификат был установлен на телефон, нападающий может видеть все запросы, даже HTTPS.



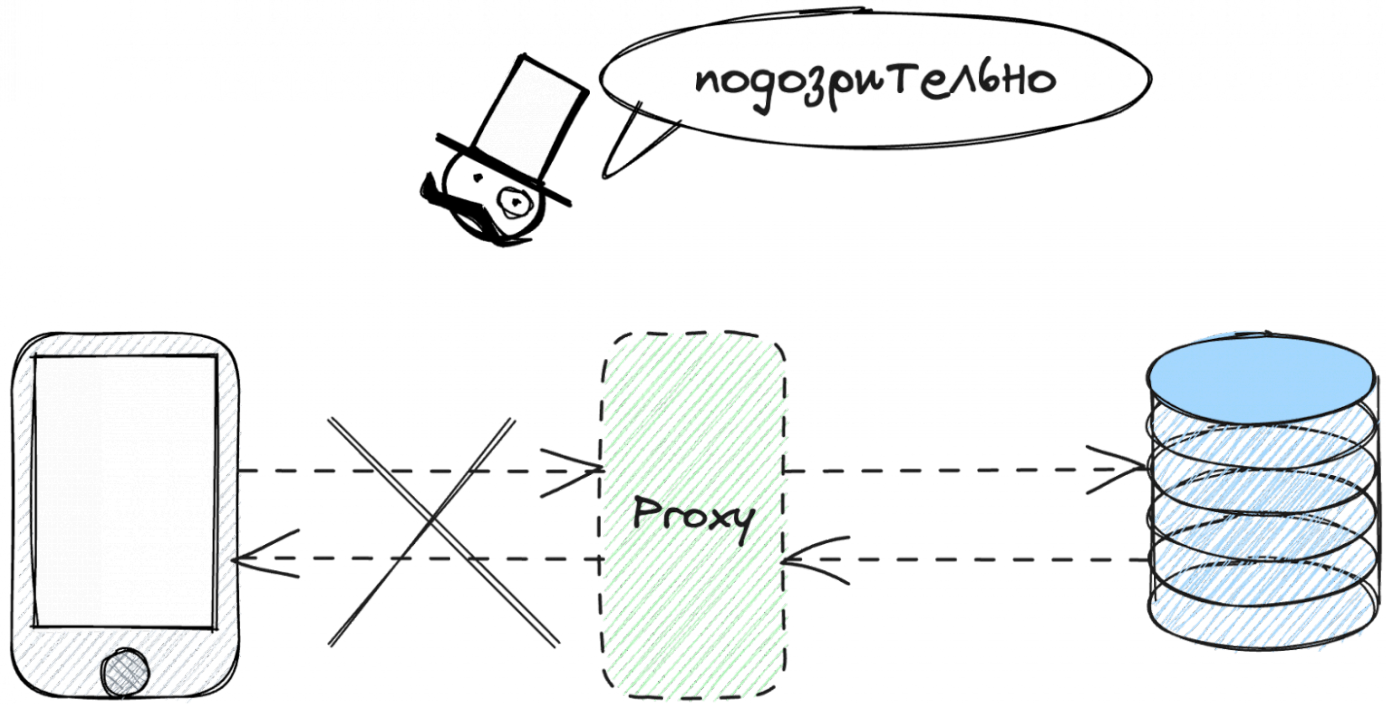
Ход защитника. Усложняем

На первый взгляд, ситуация патовая. Наш трафик смотрят, и сделать с этим мы вроде как ничего не можем. Всё верно, но. Мы можем усложнить жизнь тому, кто хочет наш трафик посмотреть. Злоумышленнику придётся сделать несколько дополнительных шагов, чтобы понять, какой трафик шлёт приложение.

Прокси

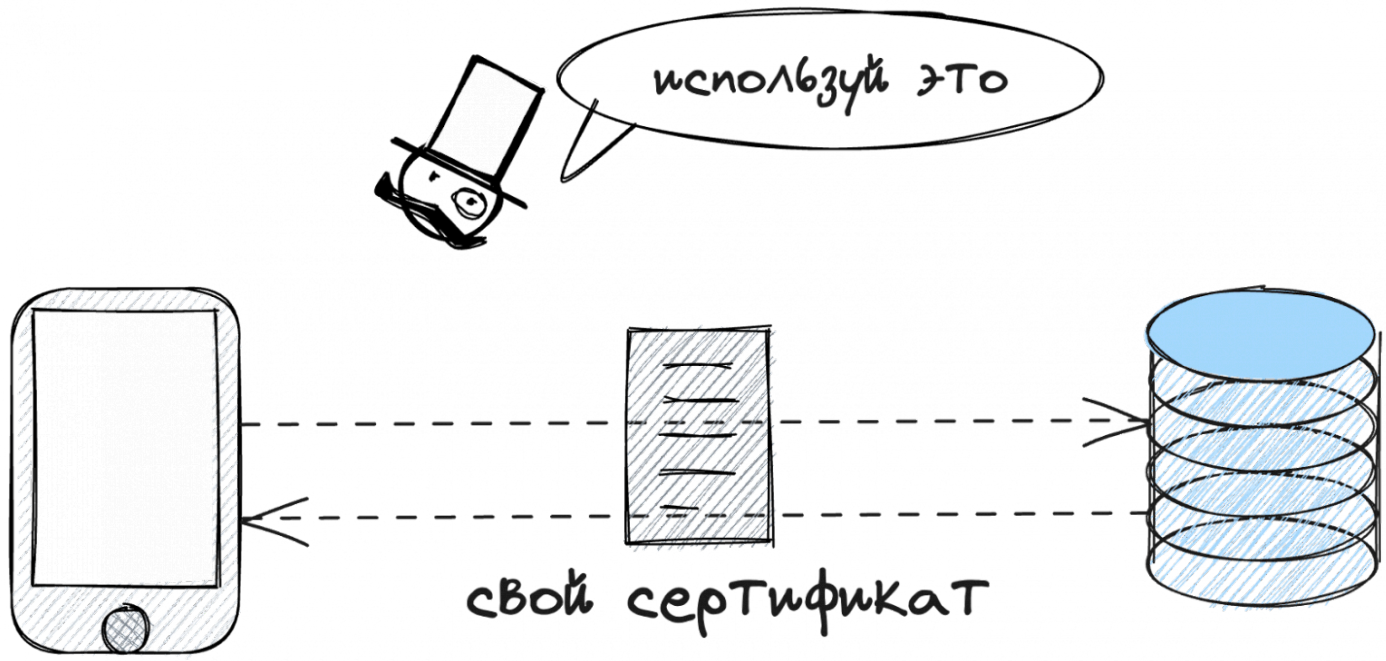
Для начала нужно понять, включён ли на телефоне Proxy. И если он включён, то вообще не шлём никакие запросы к серверу.

Могут возникнуть некоторые проблемы с тем, что часть пользователей использует прокси в повседневной жизни и у них тоже перестанет работать наше приложение. С другой стороны, вы повысите защищённость. Тут уж решать вам, какая чаша весов перевесит. В любом случае это заставит злоумышленника либо сделать проксирование трафика через роутер, либо лезть в код.



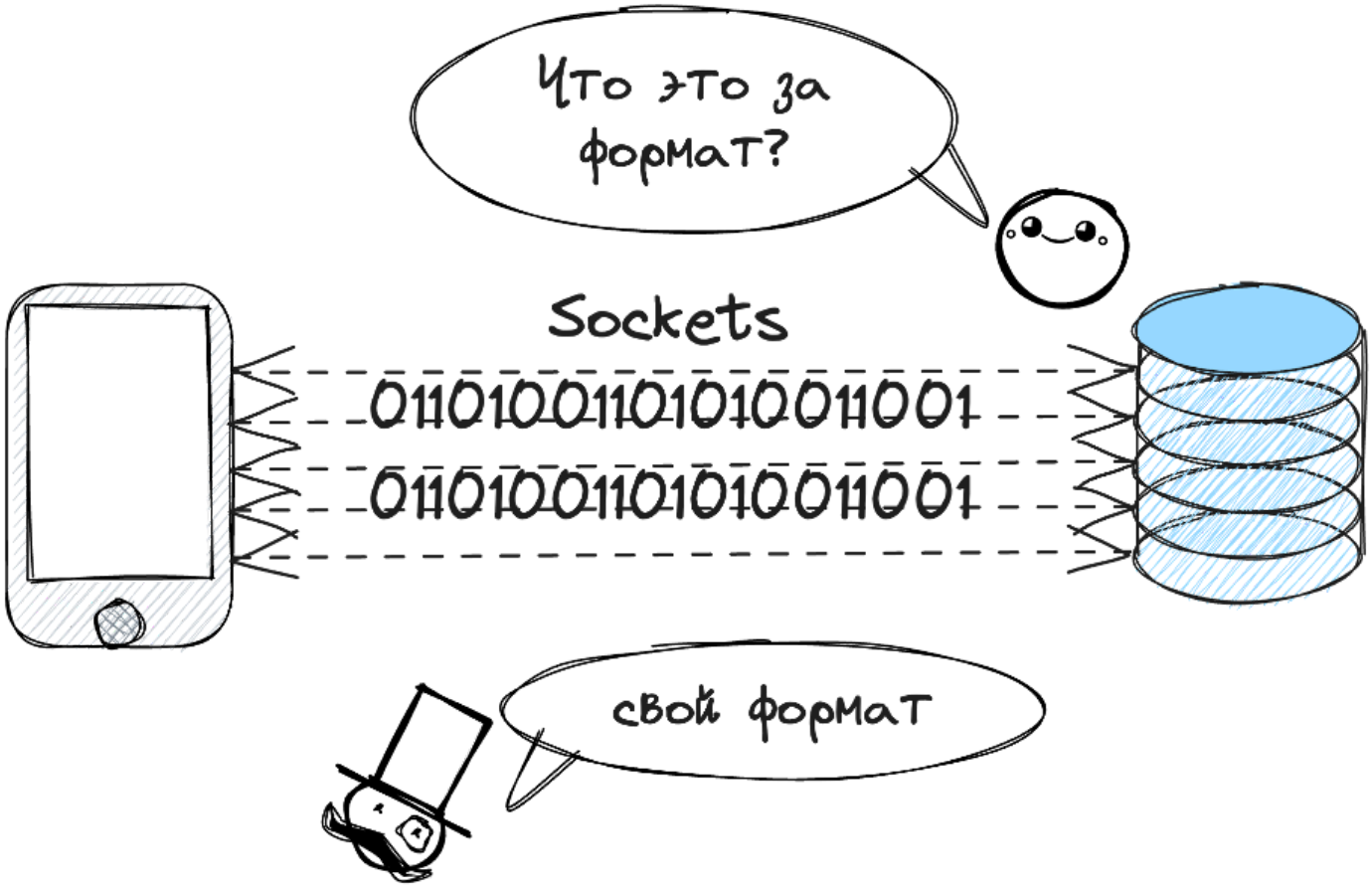
SSL Pinning

SSL Pinning позволяет нам сказать приложению: «Ты не иди в общее хранилище сертификатов, а используй только вот этот конкретный сертификат». Таким образом, подмена сертификата в общем хранилище никак не повлияет на наш сертификат. Злоумышленнику придётся распаковывать наше приложение, чтобы достать сертификат, с которым он потом и будет имитировать трафик приложения.



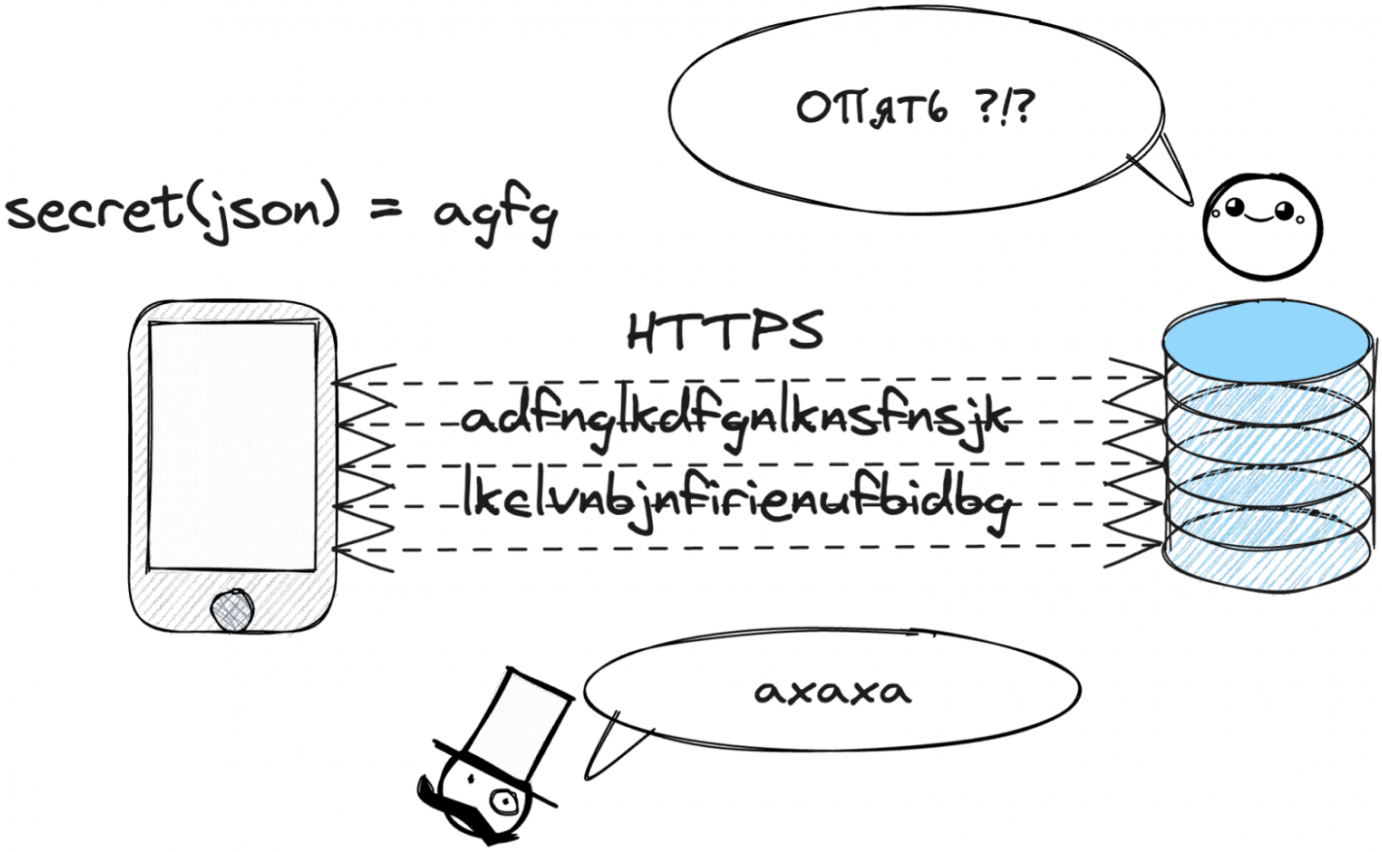
Sockets

Самые секретные методы можно перевести на сокеты. Дело в том, что они общаются, по сути, простым набором байтов, формат которых определяете вы сами. Определить по набору данных, что там за формат, почти нереально. Злоумышленник обязан опять лезть в код, чтобы понять формат сокет-сообщений и имитировать их.



Подпись

Этот способ не спасёт от просмотра, но дополнительно усложнит жизнь недоброжелателю при попытке имитации. В приложении будет алгоритм, который на основе данных запроса сможет сгенерировать уникальный хеш этого запроса. Далее на сервере происходит проверка хеша для запроса. Если проверка не пройдена, то откидываем запрос. Это заставит злоумышленника залезть в код, чтобы посмотреть алгоритм генерации хеша.



Шифрование

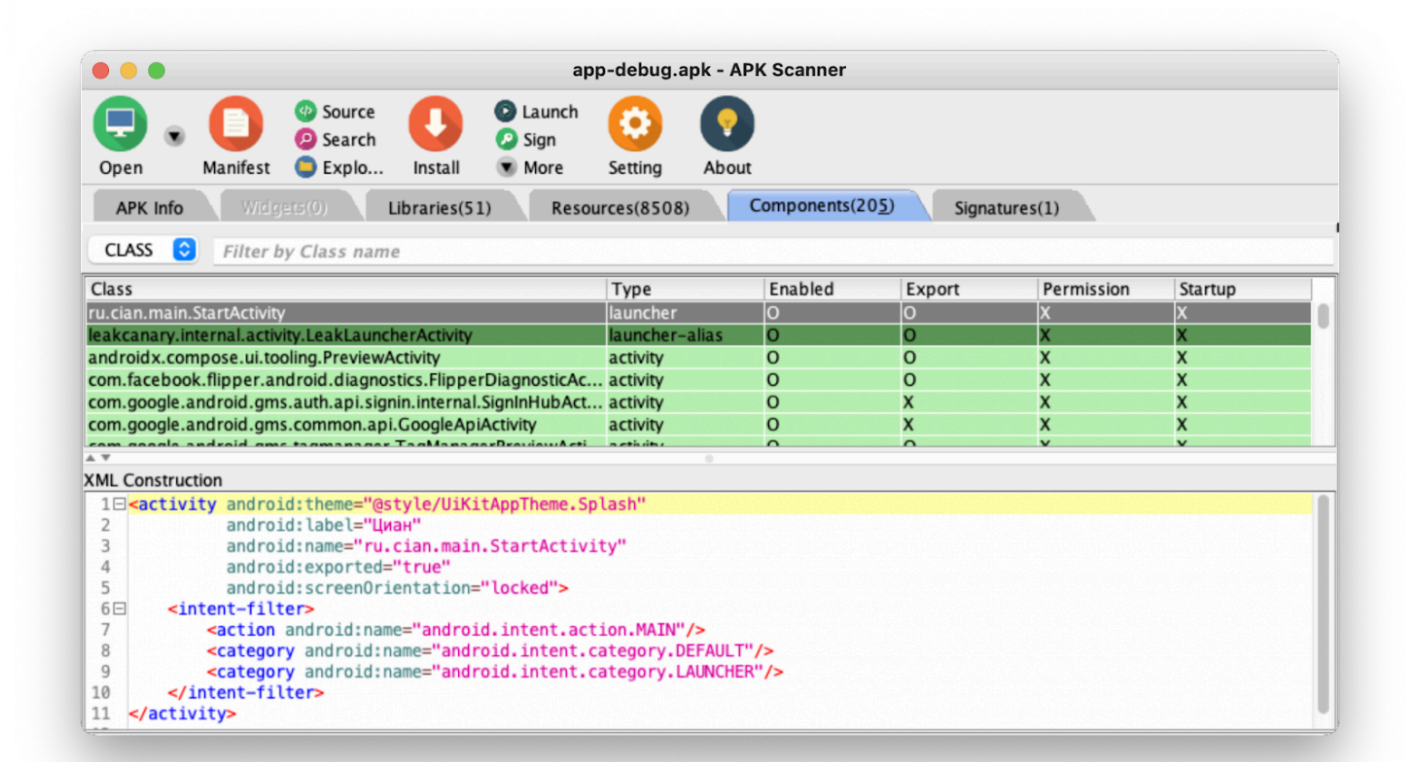
И напоследок можно конкретные запросы или вообще весь трафик дополнительно шифровать алгоритмом со стороны мобильного приложения. Затем на бэкенде расшифровывать обратно. Это наиболее геморройный способ, особенно с точки зрения обновления алгоритма и поддержки старых версий, но он точно заставит нападающего лезть в код приложения.

Можно применить и все способы. Как вы могли заметить, так или иначе, каждый из способов заставляет нападающего лезть в .apk.

Ход нападающего. Просмотр Android Manifest

Выбора нет. Лезем в .apk. Можно воспользоваться Android Studio, но в ней это делать не очень удобно. Мне больше по душе APK Scanner. Поэтому будем рассматривать на его примере. Тем не менее все эти действия можно повторить и в других инструментах.

Просто открываем интересующий нас .apk в APK Scanner. Без лишних телодвижений в нём можно посмотреть Android Manifest.



В Android Manifest мы можем увидеть все компоненты приложения. Например, найти все Activity, у которых exported=true, а значит, их можно открыть из другого приложения. Это уже даёт некоторые просторы для взлома. Особенно если на таком Activity есть важные данные. Такие Activity можно открывать просто через adb.

Или банально можно из вредоносного приложения кидать Intent со стартовым Activity вашего приложения. Если на Intent кто-то может ответить, значит, приложение установлено на

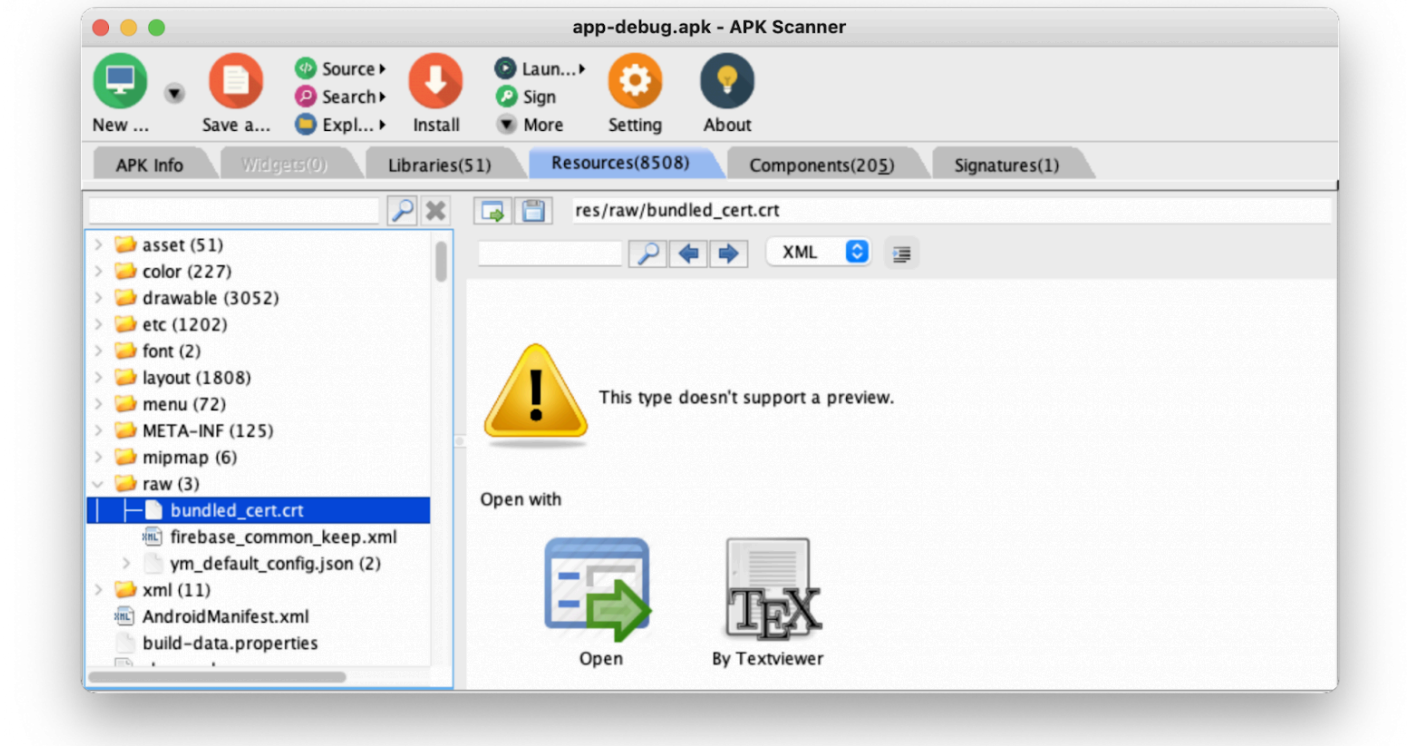
телефоне. Таким способом можно, например, собирать информацию о том, установлено ли на смартфоне определённое приложение.

Ход защитника. Закрываем компоненты

Всё банально. Проставляем exported=false у всех компонентов, которые не должны быть доступны извне. Свежий AGP по умолчанию проставляет его как false, так что надо просто следить, чтобы true не стоял на тех компонентах, которым это не нужно.

Ход нападающего. Просмотр ресурсов и кода

Теперь углубимся и взглянем на ресурсы приложения.

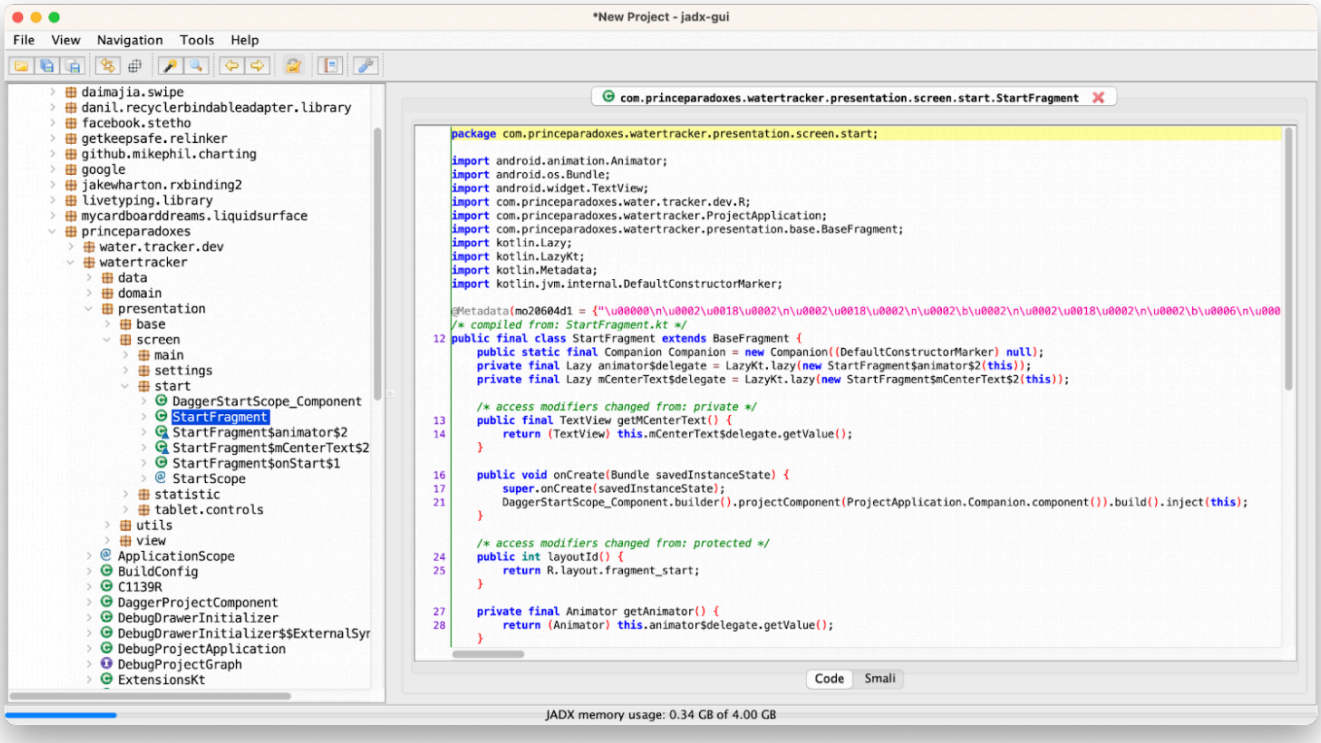


Обычно в них не очень много полезного, но иногда можно найти ключи в strings. Да и сертификат для SSL Pinning, вероятнее всего, будет лежать тут в папке raw. Смело крадём его.

Но не забываем, что нашей первоочередной целью всё-таки является код. Android имеет собственный формат упаковки кода под названием .dex. Поэтому, чтобы потом спокойно смотреть код, для начала стоит перевести .dex в .jar.

APK Scanner обладает несколькими вариантами просмотра кода: JD-GUI, JADX-GUI, BytecodeViewer. Но по большей части разница между ними только в UI. Все они для конвертации .dex в .jar по умолчанию используют утилиту dex2jar.

Я для просмотра кода рекомендую JADX-GUI. Он, как мне кажется, самый удобный, но это не точно.



С его помощью можно просматривать весь код приложения. Но кода-то много. Не просматривать же его весь.

И правда, весь смотреть точно не стоит. Благо есть «крючки». Так как большинство Android-приложений используют для работы с сетью OkHttp и Retrofit, то найти методы, помеченные аннотациями Retrofit, будет несложно. Даже если это будут какие-то другие библиотеки вроде Ktor, нам это не помешает. Это всё равно будет какая-то популярная библиотека. Мало кто будет писать собственный сетевой клиент.

Таким образом, простым поиском импортов классов сетевого клиента мы можем найти и получить доступ к коду всех методов работы с сервером.

К тому же в Android мире принято пользоваться сериализаторами/десериализаторами в/из Json наподобие Kotlin Serialization, Moshi, Gson, а значит, мы ещё и сможем посмотреть все запросы и ответы в удобном виде. Сказка!

На этом этапе можно попробовать пересобрать .apk, внеся некоторые изменения в код. Правда, это больше актуально в иных контекстах. Например, так убирают рекламу из игр или разблокируют возможности расширенной/платной версии приложения, то есть когда вам нужна «особенная» версия приложения.

Но есть случаи, когда это может помочь и при взломе. Часто для облегчения разработки и тестирования в приложении имеется debug-меню с широкими возможностями. Если разработчики не удаляют его код из release-версии, то можно разблокировать его точку входа либо вообще добавить новую кнопку, ведущую к нему.

Настало время защищаться.

Ход защитника. Обфускация

Самым главным, популярным и гламурным на районе способом защитить свой код от исследования является обфускация. К тому же она неплохо так чистит код, уменьшая его размер, и выпиливает ненужный код. Сейчас в Android основным обфускатором является R8. Он по умолчанию включён для всех release-билдов. Есть и альтернативы — старый добрый бесплатный ProGuard и платный DexGuard.

В целом статей об обфускации — вагон и приличных размеров тележка. Поэтому подробно расписывать все правила я не буду. Главное, что нам сейчас стоит знать — обфускатор превратит такой код:

```
interface RetrofitService {
   @POST("/path-to-method/")
   suspend fun sendData(@Body body: Body): Boolean
}
```

— в такой:

```
interface A {
   @POST("/path-to-method/")
   suspend fun b(@Body c: D): Boolean
}
```

Понять, что же тут такое написано, можно. Но затруднительно.

Настал ход нападающего.

Ход нападающего. «Крючки» к деобфускации

Мы открываем код и видим, что он обфусцирован. Что же делать? Нам ведь нужно как-то понять, что делает этот код. С ходу догадаться довольно проблематично. Поэтому проведём процедуру восстановления. По сути, просто переименовываем переменные и классы в более понятный и человекочитаемый вид.

Здесь у нас есть несколько «крючков».

Проверки на null в Kotlin

Если код изначально написан на Kotlin, то по умолчанию он генерирует в телах методов проверки на null. При этом в сообщении об ошибке Kotlin любезно пишет оригинальное, необфусцированное имя. В данном примере легко догадаться, что изначальные имена параметров — xValues и yValues.

```
public int mo1207a(float[] fArr, float[] fArr2) {
   C1913g.m540e(fArr, "xValues");
   C1913g.m540e(fArr2, "yValues");
   ...
}
```

Аннотации сериализаторов/десериализаторов Json

При использовании обфускации в их аннотациях остаются оригинальные имена. Иначе общение с сервером будет невозможно.

Тут мы видим, что оригинальное название property — это searchRequest.

```
@Serializable
data class D11e(

   @SerialName("searchRequest")
   val p234: String,
)
```

Логи

Часто разработчики оставляют логи в коде, чтобы в случае сложной ошибки, пришедшей в Firebase Crashlytics или куда-то ещё, была возможность понять, что вообще произошло. Часто логи достаточно подробные и могут сильно помочь в понимании того, что же делает конкретный класс или метод.

```
public final void mo2419U() {
   C2065g<R> e = this.mo14131I(new C1470c(new C1236m(this)));
   C2065g<R> n = this.mo14133n(new C1471d(C1232i.f1811e));
   C1913g.m5293d(n, "addWaterObservable / 2 < it.second")
   ...
}
```

В целом логи — это хорошая точка входа для взлома. Конечно, большинство разработчиков используют Timber, который позволяет не писать логи в Logcat в релизном приложении, но попадаются приложения и с обычным Log. К тому же заставить Timber писать логи в Logcat можно с помощью изменения поведения, про это будет ниже.

Особенно стоит следить за логами, в которых может содержаться информация о пользователях: номера телефонов, email и пр. Лучше такого вообще не допускать.

Аналитика

Я думаю, у всех что-то используется для продуктовой аналитики. Зачастую уже по названию события можно понять, что делает метод, который кидает эту аналитику. Также само событие содержит параметры с необфусцированными именами, которые дают больше контекста.

Exceptions

Мы ведь все хотим понять, из-за чего произошла ошибка? Поэтому стоит получше расписать сообщение об ошибке. Вот только для нападающего это дополнительная информация.

Есть ещё много «крючков», но уже по тем, которые я привёл, можно понять, что мы сами при написании кода неплохо помогаем во взломе. По-настоящему избавиться можно только от первого крючка — сообщений из проверок на null. От остального вряд ли кто-то захочет просто так избавляться, так как пользы это приносит реально много. Конечно, можно запариться и зашифровать логи и тексты ошибок, поменять названия ресурсов и прочее. Но, откровенно говоря, этим мало кто будет заниматься.

В итоге нападающий проделает восстановление кода, пока окончательно не поймёт, что делает нужный кусок кода. Долго и нудно, но это работает. Обфускация в контексте безопасности лишь усложняет восприятие кода человеческим глазом, не более. Поэтому не стоит думать, что обфусцировал — и забыл о безопасности.

«Штож», давайте пробовать защищаться.

Ход защитника. Проблемы dex2jar

Мы осознаём, что наш прекрасный код рано или поздно всё равно разберут и поймут, что в нём написано. Так как же нам, собственно, от этого защититься?

Мы ведь не хотим, чтобы кто-то, не очень нам известный, мог использовать запросы к серверу, притворяясь нашим приложением.

Очевидно, что dex2jar — неидеальная утилита, и не всегда ей удаётся правильно сконвертировать байткод. По сути, для защиты от него мы просто добавим в код конструкции, с которыми у dex2jar не очень. В итоге он выдаст что-то вроде: Code restructure failed.

В целом открываем список issues и выбираем любую понравившуюся проблему. Здесь я покажу только одну — использование битовых сдвигов.

Представим, что мы, чтобы защититься от имитации запросов, генерируем подпись тела нашего запроса. Для этого создадим класс SignatureHelper. Затем эту подпись добавим в заголовок запроса. Пусть подпись генерируется за счёт битового сдвига на 2 для первых десяти символов.

```
public String sign(String input) {
    StringBuilder ouputBuilder = new StringBuilder();
    for (int i = 0; i < 10; i++) {
        char oldChar = input.charAt(i);
```

```
            char newChar = (char) (oldChar << 2);
            ouputBuilder.append(newChar);
        }
        return ouputBuilder.toString();
    }
```

Иногда надо добавить несколько сдвигов, но, я думаю, суть вам ясна. При попытке декомпилировать этот код dex2jar, скорее всего, выдаст ошибку. И любезно приложит bytecode метода.

Для нападающего это неприятный момент. Копаться в bytecode затруднительно. Хотя это, конечно же, возможно. Он куда понятнее ассемблера, на мой взгляд.

«Город засыпает. Просыпается ~~мафия~~ нападающий».

Ход нападающего. Альтернативы dex2jar

Мы осознаём, что dex2jar выдал нам какую-то нерабочую фигню. Что же делать? На самом деле dex2jar — не единственный способ превратить .dex в .jar. Есть альтернативный Enjarify. Поэтому в Bytecode Viewer выбираем именно его.

Он уже нормально сможет сконвертировать байтовую конструкцию. Но и он не идеален. Где-то лучше справляется dex2jar, где-то — Enjarify. В любом случае нам этого достаточно, чтобы воспроизвести непонятный код. Что-то восстанавливаем одним инструментом, что-то — другим.

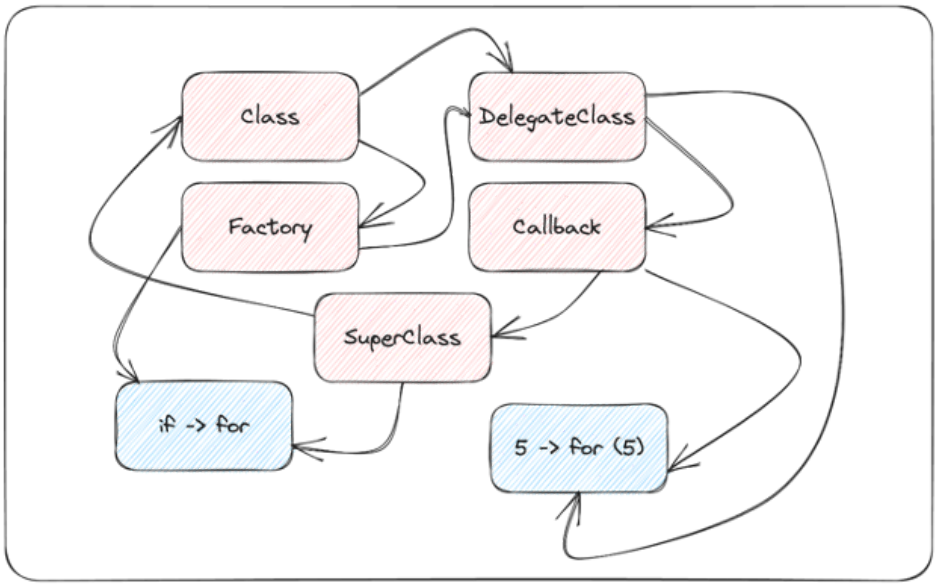
В крайнем случае ничто не мешает самостоятельно попытаться восстановить код из bytecode. Его формат называется Smali (как мило) и в целом широко известен. Можно посмотреть тут.

Пробуем защититься.

Ход защитника. Секретные техники

Ну и, как бы это странно ни звучало, лучшая защита — это говнокод. Будем намазывать на один слой говнокода ещё один слой, и ещё, и ещё.

Простую функцию разнесём на несколько классов. Для сдвига создадим фабрику, которая будет возвращать делегат. Можно добавить побольше обратных вызовов. Ещё лучше, чтобы каждый класс имел свой интерфейс с несколькими реализациями и только одна из них рабочая. Побольше лямб и анонимных классов. Вместо if с проверкой на пустоту использовать for. Если надо прибавить 5, то почему бы не прибавить 0.2 в цикле 25 раз.



Смотрите, какая красота. Тут с ходу и не поймёшь, что делает код, а ведь затем он ещё и обфусцирован будет. Мы как минимум выбесим нападающего так, что ему понадобится асбестовый стул, что, согласитесь, уже неплохо.

Нападающий подумает, что код писал или конченый идиот, или конченый гений. Но ему явно придётся потрудиться, чтобы понять, как всё это работает, и зачем оно нужно. Он, конечно, может попробовать вообще не разбираться в коде — просто вытащить ваши .class и засунуть их в jar. А затем просто вызвать итоговый метод класса. Поэтому не забывайте про обратные вызовы! Так сделать это будет сложнее.

Теперь надо пытаться как-то разобраться с этим...

Ход нападающего. Изменение поведения

Есть код, который настолько плох, что в нём не хочется разбираться. Что же делать?

Лучший вариант — вообще не разбираться. В приложении-то этот код работает. Почему бы не попытаться изменить поведение приложения так, чтобы оно само работало как парсер. В крайнем случае с помощью изменения поведения разгадать алгоритм шифрования, не углубляясь в говнокод.

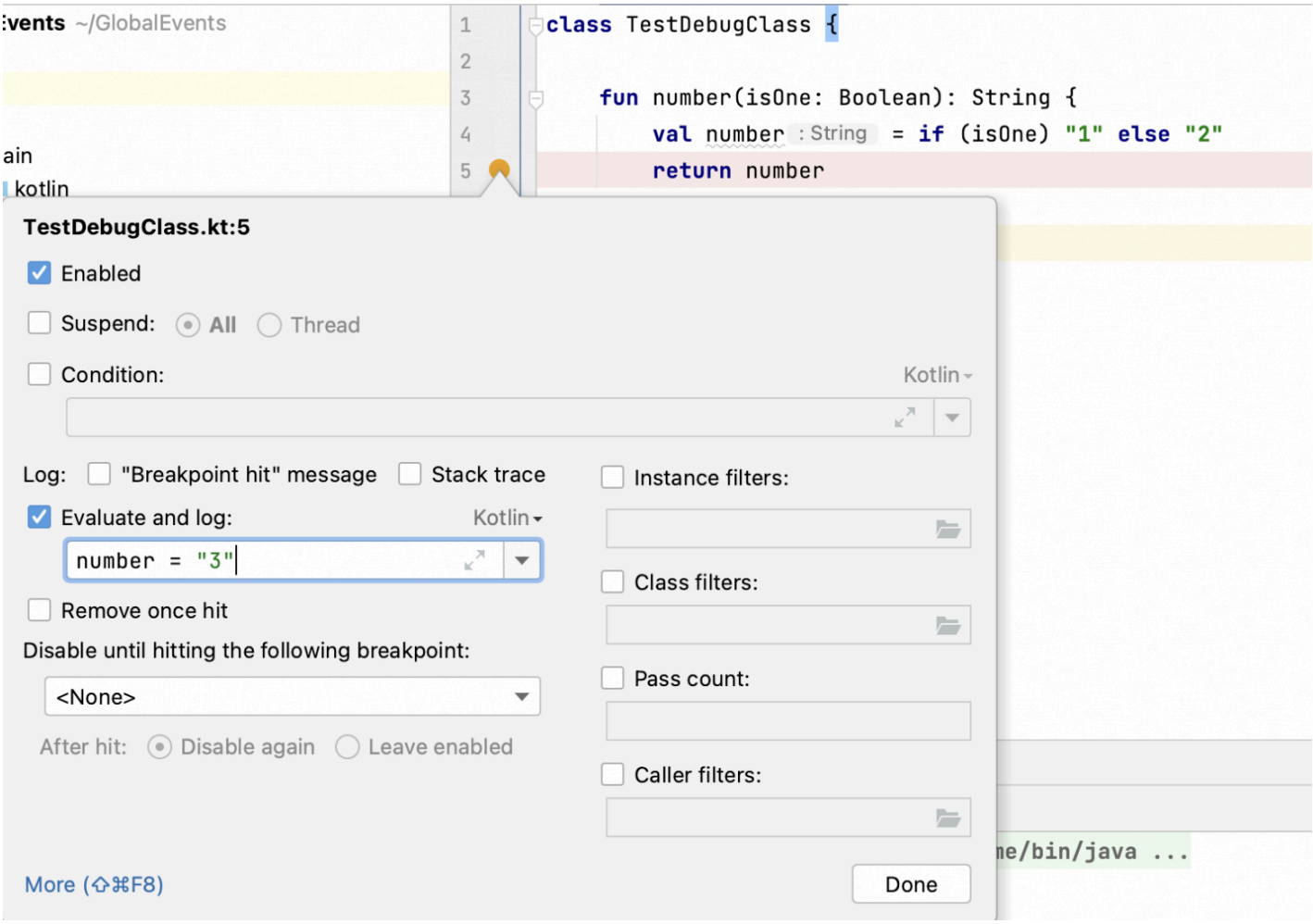
Что за «изменение поведения»?

Допустим, у нас есть метод, которому на вход подаётся boolean флаг. Если он true, то метод возвращает строку “1”, иначе — “2”.

```
fun number(isOne: Boolean): String {
    val number = if (isOne) "1" else "2"
    return number
}
```

Это очень простая и понятная логика. Но можно изменить поведение, если в этот метод «внедрить скрипт», который, например, заставит метод возвращать строку “3”. По коду у нас так не может быть, но мы внедряем скрипт, и логика меняется.

Наверное, самым распространённым примером является Breakpoint. По сути, Breakpoint работает как инъекция кода в ваше приложение. Мы можем поставить breakpoint на return и указать, что number у нас теперь равен “3”.

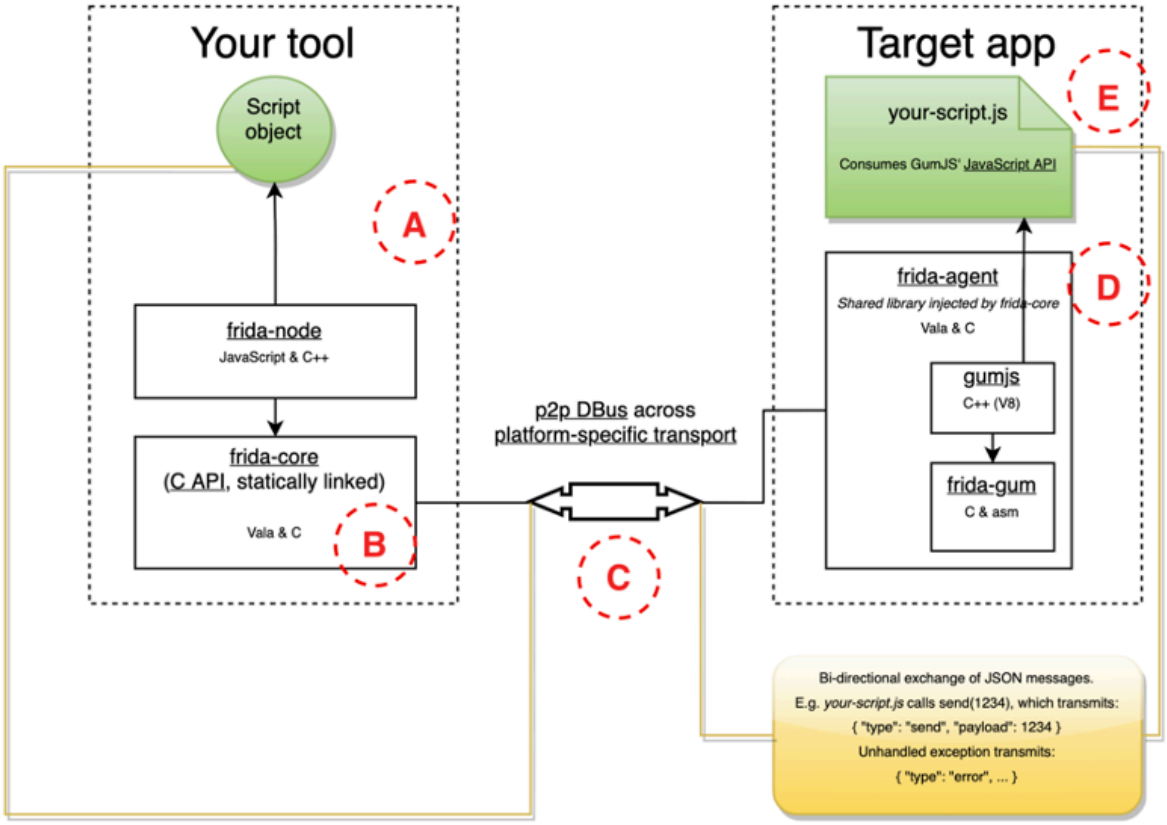


В результате мы получим “3”, что совершенно неожиданный результат для этого метода. «Так не должно быть, но так есть».

Но очевидно, что Android Studio работает с debuggable версией приложения. Там понятно, наверное, само приложение как-то так подстроено под то, чтобы в него происходили инъекции. А нам-то нужно производить инъекции в release версию. Для этого есть Frida.

Frida

Frida — это очень мощный инструмент, который позволяет делать инъекции уже в релизное приложение. Собственно, как оно работает?



По сути, у нас есть четыре части:

1. Frida-agent (D), что находится на устройстве и непосредственно выполняет инъекции. Может находиться на уровне системы, если на устройстве есть root. Либо же может быть встроен в apk, если root недоступен. Правда, придётся перепаковать .apk, что может вызвать дополнительные проблемы.
2. Frida-core (B) находится у вас на компьютере. Он управляет работой frida-agent.
3. Шина общения между frida-agent и frida-core (C). Тут, я думаю, всё понятно.
4. Непосредственно сам скрипт инъекции (A, E). Скрипты пишутся на JavaScript.

Применяем знания

Итак, что мы можем сделать с помощью Frida? Чисто для примера. Мы можем взять и достаточно просто переопределить реакцию на клик в Activity. Допустим, у нас есть MainActivity, в котором в методе onClick обрабатываются нажатия. Мы хотим помимо обычной реакции вывести что-то в лог.

```
Java.perform(() => {
  const MainActivity = Java.use('ru.cian.MainActivity');

  const onClick = MainActivity.onClick;
  onClick.implementation = function (view) {
    onClick.call(this, view);
    // Выводим что-то в лог
```

```
    var log = Java.use("android.util.Log")
    log.e("Что-то")
  };
});
```

В данной функции происходит следующее:

1. С помощью Java.use мы получаем класс MainActivity.
2. Сохраняем оригинальный метод MainActivity.onClick в переменную onClick.
3. Переопределяем реализацию метода onClick на свою, в которой сначала вызываем оригинальный метод, а затем вызываем логирование.

Ещё пример. В прошлом примере мы подменяли работу класса нашего приложения. На самом деле мы с Фридой можем сделать гораздо больше, мы даже можем логику работы классов Android SDK немножко подправить. Например, нам захотелось подправить класс Location, который используется для хранения данных о геопозиции. В основном текущей.

Пишем простенький скрипт, который переопределяет поведение getter-методов. Теперь они будут возвращать заданные нами значения.

```
Java.perform(() => {
    var Location = Java.use('android.location.Location');
    Location.getLatitude.implementation = function() {
        return LATITUDE;
    }
    Location.getLongitude.implementation = function() {
        return LONGITUDE;
    }
})
```

Эти два примера показывают весь огромный перечень возможностей, что даёт Frida.

Теперь попробуем применить её для решения нашей проблемы. Напомню, мы остановились на том, что защитник применил секретные техники говнокода для защиты.

Можно для начала попытаться как-то распутать говнокод. Чтобы это был не абстрактный набор классов, а полноценная логика вызовов. Для этого в нужной точке можно создать Exception, получить от него stackrace и вывести его в консоль Frida.

```
Java.perform(() => {
      var log = Java.use("android.util.Log"),
      var exception = Java.use("java.lang.Exception");
```

```
        console.log(log.getStackTraceString(exception.$new()));
    })
```

Теперь мы можем видеть последовательность вызова методов, а значит, часть хаков, вроде ложных классов или лишних фабрик и делегатов, нам нипочём. Это сильно облегчит понимание кода.

Можно зайти с другой стороны. Зачем нам вообще лезть в этот код? У нас же есть общепризнанные библиотеки: OkHttp, Retrofit, Android SDK, стандартная библиотека Java. Они-то написаны хорошо. Любой говнокод вынужден будет обращаться к этим библиотекам в ходе своей работы.

Для примера возьмём OkHttp. Прекрасная библиотека, которая имеет прекрасную фичу — Interceptors. Они позволяют перехватывать и модифицировать все запросы и ответы. Нам достаточно добавить свой Interceptor в OkHttp, и мы получаем полный контроль над всеми запросами и ответами. Хоть логируй, хоть модифицируй. Это позволяет нам с лёгкостью обходить проверку на включённость Proxy. Через свой Interceptor мы сможем достать даже больше данных, чем через mitmproxy.

Или можно не пытаться деобфусцировать и понять алгоритм работы SignatureHelper. Можно просто попросить его сгенерировать подпись для нужного нам запроса и вернуть результат нам в консоль.

```
Java.perform(() => {
    var signatureHelper = Java.use("ru.cian.SignatureHelper"),
    console.log(signatureHelper.sign(INPUT_STRING));
})
```

Ну и напоследок помните: мы делали защиту через SSL-пиннинг. С помощью Frida можно простым движением обратиться к TrustManager и попросить доверять всем сертификатам. SSL-пиннинг отключён таким простым скриптом.

```
Java.perform(() => {

    var array_list = Java.use("java.util.ArrayList");
    var ApiClient = Java.use('com.android.org.conscrypt.TrustManagerImpl');

    ApiClient.checkTrustedRecursive.implementation = function(a1, a2, a3, a4, a5, a6)
        var trusted = array_list.$new();
        return trusted;
    }

});
```

И вот как от этого защищаться?

Ход защитника. C++

Мы приходим к выводу, что, по сути, любой написанный нами код на JVM можно в итоге восстановить или изменить его поведение. Пусть и со всей проделанной нами работой это стало значительно сложнее.

Что ещё можно придумать? Любой JVM-код восстанавливается достаточно просто. А как насчёт применения C++? Пытаться восстановить два языка явно сложнее, чем один. Так и есть.

У нас есть SignatureHelper, который занимается генерацией подписи. Почему бы нам не переписать его на C++? Сдвиги нам теперь не нужны, и для простоты понимания изменим алгоритм. Пусть он просто добавляет единицу к входному символу.

```
std::string SignatureHelper::createSignature(std::string inputString) {
    std::string outputString = std::string();
    for (int i = 0; i < inputString.size(); i++) {
        outputString[i] = inputString[i] + 1;
    }
    return outputString;
}
```

Мы создаём outputString, проходим по inputString и к каждому символу добавляем единичку. В итоге у нас всего три действия: создание строки, цикл с изменением и возврат результата. Для входной строки 1234 мы на выходе получим 2345.

Напомню, что для обращения к методам C++ из JVM вам придётся писать обёртку JNI.

```
Java_com_princeparadoxes_myapplication_MainActivity_signature(
        JNIEnv* env,
        jobject jobj,
        jstring inputString) {
    SignatureHelper helper = SignatureHelper();
    jboolean isCopy;
    const char *convertedValue = (env)->GetStringUTFChars(inputString, &isCopy);
    std::string signature = helper.createSignature(convertedValue);
    return env->NewStringUTF(signature.c_str());
}
```

Этот код занимается тем, что конвертирует объекты Java в объекты C++, выполняет C++ код и делает обратную конвертацию. Вроде должно сработать.

Теперь нападающий вынужден разбираться с C++.

Ход нападающего. Подключаем библиотеку

Для начала стоит попробовать вообще не лезть в C++ код. Алгоритм генерации подписи сделан чистой функцией. Значит, мы можем просто вызвать метод signature из своего кода. Для этого надо найти все обращения к C++ библиотеке в оригинальном коде. В Kotlin они помечены как external, в Java — как native. Соответственно, находим вообще все методы по этим ключевым словам.

Благо, что из-за особенностей JNI имена таких методов часто не обфусцируются без дополнительных настроек, как и имена параметров этих методов. А значит, мы с ходу понимаем, что нужно передать на вход JNI функции. Дальше просто копируем себе .so и вызываем метод signature из своего кода.

Ход защитника. Дополнительные данные

Думаю, очевидно, что функции шифрования на C++ не стоит делать чистыми. Иначе нападающему даже не надо разбираться с кодом. Он просто будет использовать библиотеку у себя.

Поэтому добавим дополнительные данные. Они могут любыми, что придут вам в голову. Я для простоты возьму версию Android API. В Android NDK для этого есть метод android_get_device_api_level(). Теперь вместо единички будет прибавляться именно она.

```
std::string SignatureHelper::createSignature(std::string inputString) {
    std::string outputString = std::string();
    for (int i = 0; i < inputString.size(); i++) {
        outputString[i] = inputString[i] + android_get_device_api_level();
    }
    return outputString;
}
```

При 31 версии Android API для входной строки 1234 мы на выходе получим PQRS.

Дальше пусть версия Android API дополнительно шлётся в одном из заголовков, например, User-Agent. Сервер на своей стороне сделает дешифровку за счёт значения из заголовка. С ходу нападающий не догадается о таком алгоритме. Разве что случайно «потыркает» значения в User-Agent, но это маловероятно.

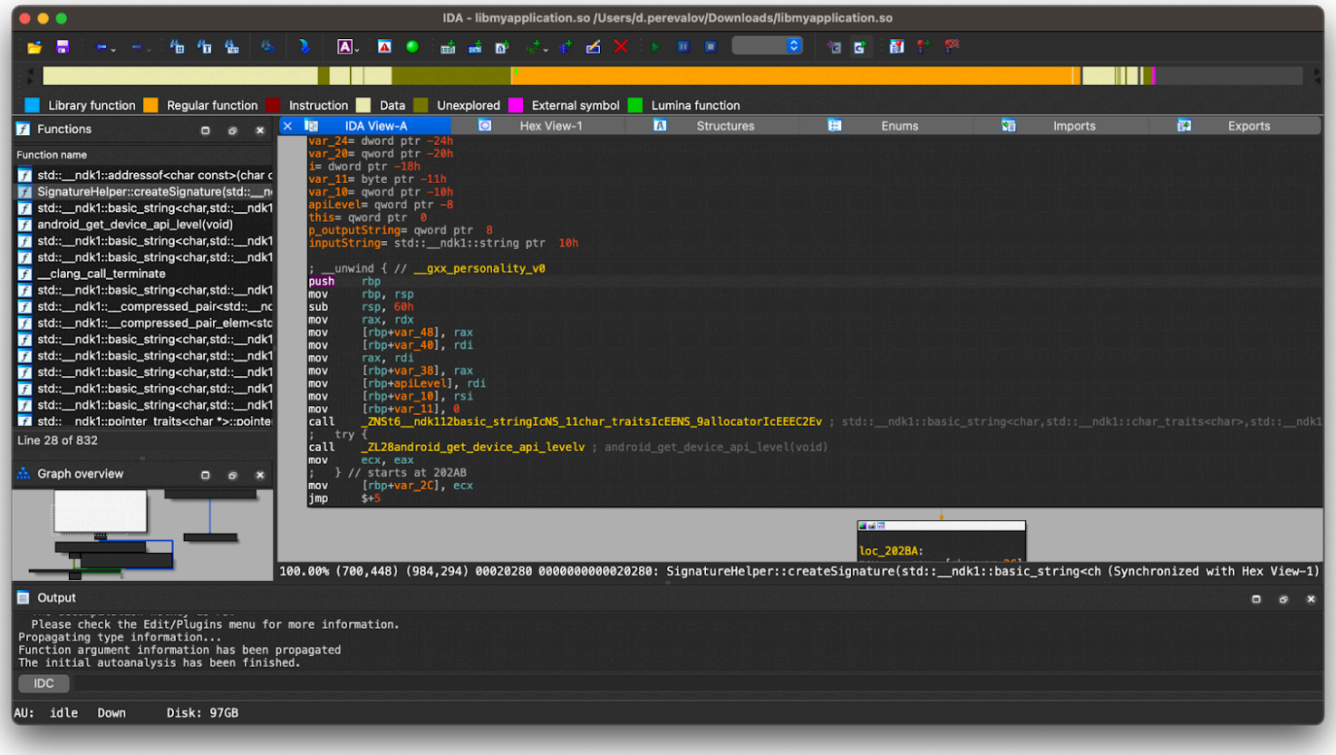
Теперь нападающий не сможет просто так использовать библиотеку, особенно если данные будут сложнее, чем версия Android API. Например, какой-нибудь уникальный идентификатор устройства или значения из BuildConfig. Также хорошей практикой станет использование хеша сертификата, которым было подписано приложение. Это позволяет легко понять, что приложение было перепаковано.

Ещё лучше совместить несколько наборов данных. Из-за этого нападающему придётся дольше возиться с кодом.

Ход нападающего. Декомпиляция

Java/Kotlin код мы спокойно декомпилировали, а что с этим у C++? Для декомпиляции C++ есть IDA. Она платная, но у неё есть и бесплатная версия. Бесплатная весьма ограничена, но в целях демонстрации её вполне хватит.

Загружаем в IDA наш .so и на выходе получаем список методов в библиотеке. Правда, большинство из них нас не интересует, так как они относятся к Android NDK. Тем не менее найти в списке нужные методы можно. При попытке посмотреть содержимое метода IDA выдаст нам код Assembler.

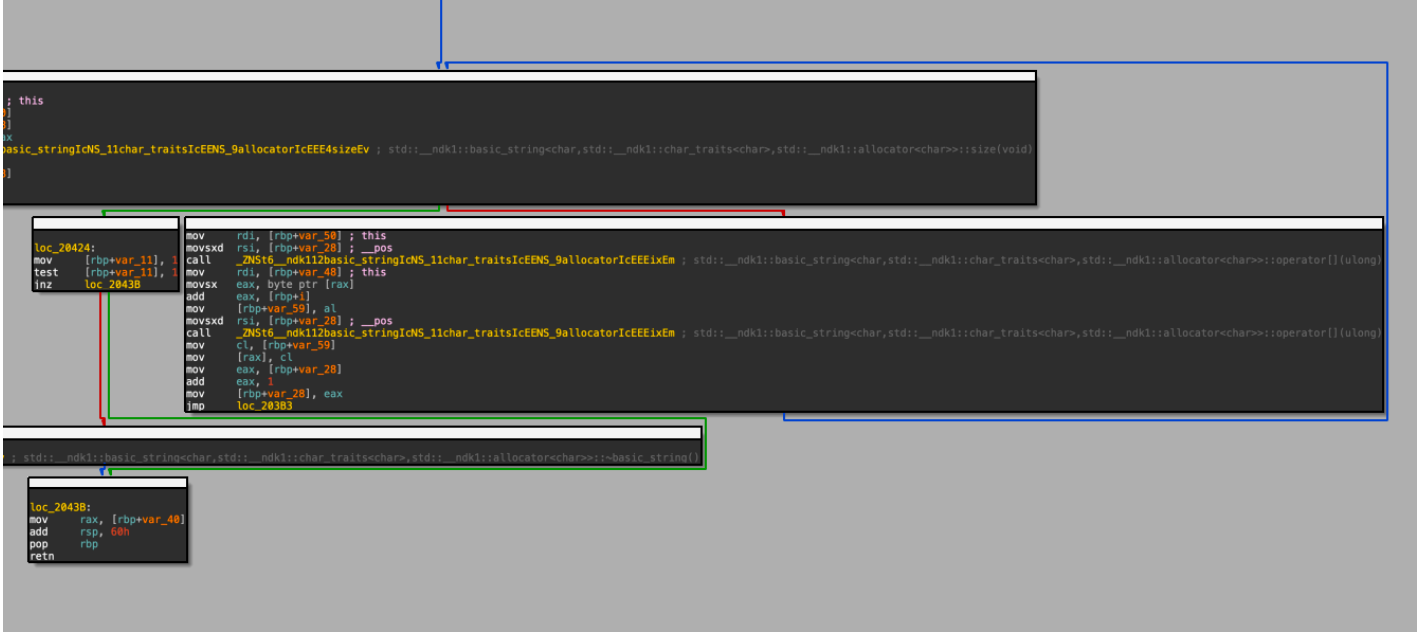


В целом Assembler не самый сложный с точки зрения логики и синтаксиса язык, но зато весьма сложен для чтения. Особенно если сравнивать со Smali.

Однако если приглядеться, то можно увидеть логику. Например:

```
mov     eax, [rbp+var_28]
add     eax, 1
mov     [rbp+var_28], eax
jmp     loc_203B3
```

Здесь идёт чтение из rbp+var_28 в регистр eax. Затем к нему прибавляется единица. Далее результат из eax записывается обратно в rbp+var_28, и происходит прыжок к другому блоку кода. Сразу в голове всплывает цикл. Благо, что это не надо каждый раз глазами высматривать, и IDA заботливо может показать код в виде блок-схемы, на которой видны и ветвления, и циклы.



Также можно разглядеть и вызов метода android_get_device_api_level.

```
call    _ZL28android_get_device_api_levelv
```

В сочетании с циклом уже можно составить примерную картину работы алгоритма формирования подписи.

В реальности, понятное дело, алгоритм будет сложнее. И если защитник усложнит логику формирования сдвига, к примеру, добавив дополнительные данные и вынеся получение сдвига в отдельный метод, например, getShift(), то всё это мы также увидим.

```
mov     [rbp+shift], rdi
mov     [rbp+var_11], 0
mov     rdi, [rbp+var_38] ; this
call    __ZN15SignatureHelper8getShiftEv
mov     ecx, eax
mov     [rbp+var_2C], ecx
jmp     $+5
```

Понятно, что алгоритм может быть слишком сложным для его анализа через Assembler целиком. Это в целом возможно, но будет крайне тяжело. Значит, нам нужен ещё какой-то помощник...

Дополнительная помощь

Здесь к нам на помощь опять придёт Frida. На удивление, она может работать и с C++. Не так легко, как с Java, но может.

Какие варианты? Помните, я говорил, что мы можем анализировать говнокод с помощью стандартных библиотек. Можно, как в этом примере, сделать трассировку на стандартные

методы работы со строками вроде strcmp, strlen и пр.

Для этого напишем функцию, которая позволит нам трассировать методы с нужным именем.

```
function hook(name, count) {
    Interceptor.attach(Module.findExportByName(«libc.so», name), {
        onEnter: function(args) {
            let bt = DebugSymbol.fromAddress(Thread.backtrace(this.context, Backtrace
            let arg = [];
            for (var i = 0; i < count; i++){
                try {
                    arg.push(Memory.readCString(args[i]));
                } catch (e) {}
            }
            if (bt.moduleName.indexOf(«libchallenge.so») !== -1) {
                console.log(name + '(»' + arg.join('», «') + '») ' + bt);
            }
        }
    });
 }
```

Далее просто перечисляем желаемые методы работы со строками:

```
function makeHooks() {
    hook(«strcmp», 2);
    hook(«strlen», 1);
    ...
 }
```

Таким образом, по обращениям к методам мы сможем больше понять о логике работы алгоритма. Трассировку можно вешать и на методы в рамках самого алгоритма.

Хотя, конечно, возможности Frida в C++ скуднее, чем в JVM. Там за счёт его интерпретируемости можно вообще почувствовать себя богом, делая, по сути, что угодно.

Так или иначе, это станет большой помощью в распутывании логики работы.

Рано или поздно нападающий сможет понять логику работы алгоритма шифрования.

Ход защитника. Что дальше?

Можно, конечно, попробовать зайти на «новый круг». Применить обфускацию и говнокод в C++, вставлять запутывающие вставки на Assembler. Думаю, описывать это смысла нет, ибо, по сути, буду повторяться.

Также можно пробовать обнаруживать, что Frida подключена к приложению через проверку открытости типичного для неё порта, но это можно обойти.

Думаю, где-то тут пора переходить к выводам.

Выводы

Если ставить вопрос ребром: «Можно ли защитить мобильное приложение?». На мой взгляд, нет.

Рано или поздно человек, если захочет, раскопает всё, что ему нужно. Поэтому изначально стоит воспринимать мобильное приложение как скомпрометированное. И, кстати, Frida и на IOS работает, если вдруг у вас такой вопрос возник.

Но тут какая штука. Допустим, у вас интернет-магазин с десятью конкурентами. Кто-то из них решил поиграть нечестно и всегда ставить цену ниже, чем у вас. Прибегнув для этого к взлому приложения. Дело в том, что вы у него не один такой конкурент, и нападающий будет пытаться взломать вообще всех конкурентов. Если он видит, что с вашим приложением ему придётся повозиться, то он, скорее всего, «забьёт» и пойдёт взламывать следующего конкурента. Или искать обходной путь через Web.

Как следствие, наращивать слои защиты нужно до определённого момента, когда вы поймёте, что сейчас достаточно. Лезть куда-то там в дебри и даже в C++ вовсе не обязательно. Каждый слой просто увеличивает количество времени, которое нападающему придётся на вас потратить. Нужно найти баланс, когда для него количество времени, которое ему придётся потратить на ваш взлом, станет нецелесообразным. Основа защиты, как ни крути, должна быть на бэкенде. Ведь даже взламывая ваше приложение, нападающий, по сути, хочет добраться до бэкенда.

В этой статье, естественно, не вся информация о безопасности Android-приложений. По сути, мы рассмотрели всего один сценарий, хоть и более-менее глубоко. Если вас заинтересовала эта тема, то есть такая штука, как OWASP. Это целый набор вообще всего, что связано с безопасностью мобильного приложения. Целых две книги. Первая посвящена безопасности в целом, а вторая — тестированию вашего приложения на безопасность. В них есть информация обо всём, что я писал выше, включая Frida и типичные скрипты к ней. А также многое-многое другое.

Теги: security, https, android, ida, android development, owasp, frida, dex2jar

Хабы: Блог компании Циан, Информационная безопасность, Разработка под Android

Подписаться

Сайт Хабр Карьера ВКонтакте

56 Карма 0 Рейтинг

Данил Перевалов @princeparadoxes

Android developer

Подписаться

ВКонтакте Github

Комментарии 9

akk0rd87 3 апр в 23:25

Спасибо за статью. Очень познавательно. Но признайтесь, что нижеприведенный метод Вы не тестировали.

```
std::string SignatureHelper::createSignature(std::string inputString) {
    std::string outputString = std::string();
    for (int i = 0; i < inputString.size(); i++) {
        outputString[i] = inputString[i] + android_get_device_api_level();
    }
    return outputString;
}
```

0 Ответить

princeparadoxes 4 апр в 13:37

Спасибо. Дополнительно проверил - вроде работает)

0 Ответить

akk0rd87 4 апр в 19:45

Я предлагаю Вам задуматься о трех моментах:

1) Какого размера буффер выделяется под данные, при выполнении

```
std::string outputString = std::string();
```

2) Куда Вы записываете очередной символ операцией присваивания

```
outputString[i] =
```

3) Насколько безопасна такая операция, причем тут UB и переполнение буффера?

+1 Ответить

princeparadoxes 4 апр в 23:13

А, Вы про это. Да, есть такое, в проде такой код использовать нельзя. Он тут скорее для иллюстрации логики подмешивания дополнительных данных. У меня в нём ещё и корнер-кейсы, вроде того, что android_get_device_api_level может вернуть -1, не учтены.

Я на C++ давно уже не писал, так что это точно не хороший код)

0 Ответить

Spyman 4 апр в 04:21

Статья прекрасна. Когда копал в ту же сторону находил ещё один способ защиты - можно было после проведения покупки через google play запрашивать целостность приложения. Но не помню точно как это работало((вроде можно было запросить у гугла проверку покупки и там заодно сверить хеш от установленного пакета, чем занимался плеймаркет на пользовательском устройстве. Получалось что для взлома приложения дополнительно требовалось бы ещё и сделать иньекцию в google play, чтобы он обманывал сервера гугла о целостности приложения.

С выводами полностью согласен - после передачи приложения пользователю его уже не защитить никак. Даже на ios с jailbeake можно вытащить ipa в расшифрованном виде для дальнейшей декомпиляции.

+2 Ответить

VADemon 4 апр в 13:07

> Если надо прибавить 5, то почему бы не прибавить 0.2 в цикле 25 раз.

What Every Computer Scientist Should Know About Floating-Point Arithmetic

Ответить

princeparadoxes 5 апр в 14:59

Есть такое) В статье это больше для примера того, как усложнить восприятие кода для атакующего.
И насколько я понимаю, при 25 или даже 100 итерациях погрешность из-за чисел с плавающей запятой всё равно будет слишком мала и при округлении "нивелируется".
Это больше актуально когда все вычисления во float (особенно умножение) или когда итераций цикла миллионы или миллиарды.

Ответить

Wesha 4 апр в 21:29

> Если ставить вопрос ребром: «Можно ли защитить мобильное приложение?». На мой взгляд, нет.

Уже давно всю плешь проклевали: если устройство, на котором крутится защищённое ПО, находится в руках у атакующего, а ключи находятся на нём же, то взлом *неизбежен*. Чисто потому, что тогда атакующий тупо подключается к схеме тем или иным путём и считывает всё, что ему надо.

> наращивать слои защиты нужно до определённого момента, когда вы поймёте, что сейчас достаточно.

Ваистену так. "Чтобы убежать от медведя, достаточно бежать быстрее одного из прочих убегающих." (с)

Ответить

НЛО прилетело и опубликовало эту надпись здесь

Вы можете оставлять комментарии только к свежим публикациям

Публикации

ЛУЧШИЕ ЗА СУТКИ ПОХОЖИЕ

Flammmable 23 часа назад

Электроника в вопросах и ответах

13 мин 5.8K

+53 53 47 +47

slava_rumin 21 час назад

Я ушел с маркетплейсов, закрыл производство, продаю на 25 млн в год, и живу в 6-местном хостеле. А как прошел ваш год?

Простой 12 мин 59K

Интервью

+40 56 52 +52

ru_vds 22 часа назад

Спасите меня из ада данных

Средний 10 мин 3.3K

Обзор Перевод

+33 12 10 +10

 **true-grue** 3 часа назад

Толкаем байты, или Простейший эмулятор своими руками

Простой 11 мин 1.1K

Туториал

+30 11 1 +1

 **karen07** 23 часа назад

WireGuard и QUIC

Средний 4 мин 14K

Туториал

+29 104 26 +26

 **shiru8bit** 2 часа назад

Новогодний DIY: В лесу крутилась ёлочка

Простой 17 мин 1K

Кейс

+26 6 1 +1

1 час назад

Утренняя история: праздничный виммельбух для уставших, но не сдавшихся

2 мин 1.3K

+23 1 51 +51

 **Thomas_Hanniball** 6 часов назад

Scrum is dead или почему Kanban намного эффективнее Scrum

Простой 6 мин 1.6K

Мнение

+21 21 15 +15

 **donovanrey** 2 часа назад

Как мы построили SIEM для Холдинга «Газпром-Медиа» и научились подключать новые активы к SOC за сутки

12 мин 444

Кейс

+18 3 0

Ne_Palimsa 5 часов назад

Как мигрировать данные между разными StorageClass в Kubernetes и зачем это делать

Простой 7 мин 518

Туториал

+18 23 0

Получи ответ на свой вопрос в сфере патентования

Турбо

Показать еще

МИНУТОЧКУ ВНИМАНИЯ

Карманная Llama: запускаем языковую модель в браузере

Исследуем новые миры: Хабр и ЭКОПСИ изучают IT-рынок РБ

Один анализ крови, чтобы править всеми

КУРСЫ

Архитектура ИТ-решения: проектирование и реализация MVP
12 января 2025 · Systems Education

Веб-приложения на Python при партнерстве с ВМК МГУ им. М.В. Ломоносова
По факту набора · Coddyschool

JS Intensive Course
По факту набора · IT INCUBATOR

Кем стать? Бесплатная профориентация

По факту набора · Skillbox

«Управление персоналом организации. Обеспечение эффективного функционирования системы управления персоналом» с присвоением квалификации «Специалист по управлению персоналом»

По факту набора · НАДПО

Больше курсов на Хабр Карьере

ИНФОРМАЦИЯ

Сайт	www.cian.ru
Дата регистрации	15 июля 2019
Дата основания	21 сентября 2001
Численность	1 001–5 000 человек
Местоположение	Россия
Представитель	Zina Bezzabotnova

БЛОГ НА ХАБРЕ

29 ноя в 10:00

Как ускорить проверку приложения с помощью Impact-анализа: Часть 1 — Статические анализаторы

973 5 +5

19 ноя в 14:58

Как мы попробовали Apache Iceberg в связке со Spark и что из этого вышло

3K 9 +9

12 ноя в 09:30

5 шагов адаптации тимлида в новой компании

1.2K 0

16 окт в 10:30

Tuist: добавляем генерацию проекта в текущее приложение

1.4K 4 +4

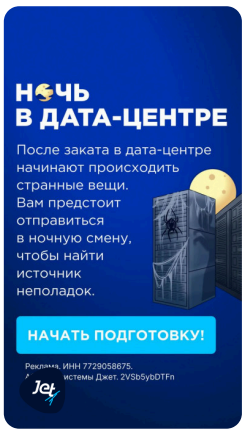
3 окт в 14:09

Как понять продукт и зачем это нужно разработчику

1.6K 2 +2

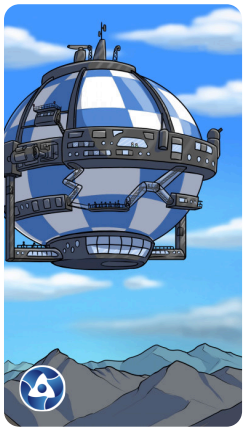
ИСТОРИИ

Спасибо, КЭП

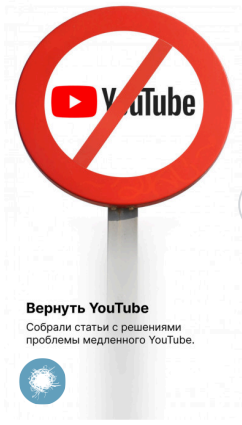


Неполадки в ночном дата-центре

Магия тестировщиков зовет



С высоты аналитического полета



Вернуть YouTube