Статья 6 способов спрятать данные в Android-приложении

xss.is/threads/27460

Привет, дорогой читатель, уже достаточно давно я изучаю мобильные приложения. Большинство приложений не пытаются как-то скрыть от меня свой «секретный» функционал. А я в это время радуюсь, ведь мне не приходится изучать чей-то обфусцированный код.

В этой статье я хотел бы поделиться своим видением обфускации, а также рассказать про интересный метод сокрытия бизнес-логики в приложениях с NDK, который нашел относительно недавно. Так что если вас интересуют живые примеры обфусцированного кода в Android — прошу под кат.

Под обфускацией в рамках этой статьи будем понимать приведение исполняемого кода Android-приложения к трудному для анализа виду. Существует несколько причин затруднять анализ кода:

1. Ни один бизнес не хочет, чтобы в его «внутренностях» ковырялись.
2. Даже если у вас приложение-пустышка, интересное там можно найти всегда (пример с инстаграмом).

Многие разработчики решают проблему простым форком конфига ProGuard. Это не лучший способ защиты данных (если вы первый раз слышите об этом, то см. вики).

Хочу привести показательный пример, почему предполагаемая “защита” с помощью ProGuard не работает. Возьмем любой простенький пример из Google Samples.

```
buildTypes {
    release {
        debuggable false
        minifyEnabled true
        shrinkResources true
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        signingConfig signingConfigs.Application
    }
    debug {
        debuggable true
        minifyEnabled true
        shrinkResources true
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
    }
}
```

Подключив к нему ProGuard со стандартным конфигом, получим декомпилированный код:

```
public class a
extends h {
    public void g() {
        com.example.android.common.logger.a.a(new c());
        com.example.android.common.logger.a.a("SampleActivityBase", "Ready");
    }

    @Override
    protected void onCreate(Bundle bundle) {
        super.onCreate(bundle);
    }

    @Override
    protected void onStart() {
        super.onStart();
        this.g();
    }
}
```

«Ооо, ничего непонятно» – скажем мы и успокоимся. Но через пару минут переключения между файлами найдём подобные кусочки кода:

```
private void Z() {
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.append("Setting up a VirtualDisplay: ");
    stringBuilder.append(this.ad.getWidth());
    stringBuilder.append("x");
    stringBuilder.append(this.ad.getHeight());
    stringBuilder.append(" (");
    stringBuilder.append(this.V);
    stringBuilder.append(")");
    com.example.android.common.logger.a.a("ScreenCaptureFragment",
    this.aa = this.Z.createVirtualDisplay("ScreenCapture", this.ad.
    this.ac.setText(2131427364);
}
```

В этом примере код приложения выглядит затрудненным довольно слабо (логирование данных, создание видео захвата), поэтому некоторые методы, использованные в оригинальном коде, легко понятны и после обработки конфигом ProGuard.

Дальше больше, взглянем на data-классы в Kotlin. Data-класс по умолчанию создает метод “toString”, который содержит в себе названия переменных экземпляра и название самого класса.

Исходный data-класс:

```
data class Category(
    val id: String,
    val title: String,
    val slug: String
) {
    companion object
}
```

Он может превратиться в лакомый кусочек для реверсера:

```
public String toString() {
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.append("Category(id=");
    stringBuilder.append(this.f3594b);
    stringBuilder.append(", title=");
    stringBuilder.append(this.f3595c);
    stringBuilder.append(", slug=");
    stringBuilder.append(this.f3596d);
    stringBuilder.append(")");
    return stringBuilder.toString();
}
```

(автогенерация метода toString в Kotlin)

Выясняется, что ProGuard прячет далеко не весь исходный код проекта.

Если я все еще не убедил вас в нецелесообразности защиты кода таким способом, то давайте попробуем оставить в нашем проекте атрибут “.source”.

Code:

```
-keepattributes SourceFile
```

Эта строчка есть во многих opensource проектах. Она позволяет просматривать StackTrace при падении приложения. Однако, вытащив “.source” из smali-кода, мы получим всю иерархию проекта с полными названиями классов.

По определению, обфускация – это “приведение исходного кода в нечитаемый вид для того, чтобы противодействовать разным видам ресерча”. Однако, ProGuard (при использовании со стандартным конфигом) не делает код нечитаемым – он работает как минификатор, сжимающий названия и выкидывающий лишние классы из проекта.

Такое использование ProGuard – это легкое, но не совсем подходящее для хорошей обфускации решение на ”авось”. Хорошему разработчику нужно заставить ресерчера (или злоумышленника) испугаться “китайских символов”, которые трудно деобфусцировать.

Что прячем

Теперь давайте посмотрим, что обычно прячут в приложениях.

Ключи шифрования:

```
String str = sessToken;

String aesKey =
        "BeCWvhSlKvvVWGsiYm8/4u3kAWDaDb3Vp7UuUwloIh/iu7qG6TP5q9aRQcpZEM0I0UezP46YuWh7";

byte[] _aesKey = Base64.getDecoder().decode(aesKey);
```

Специфическую логику приложения:

```
private void initializeMatcher(@NonNull String string2) {
    UriMatcher uriMatcher;
    matcher = uriMatcher = new UriMatcher(-1);
    uriMatcher.addURI(string2, "account", 0);
    matcher.addURI(string2, "account_extended", 1);
    matcher.addURI(string2, "package", 2);
    matcher.addURI(string2, "lib", 3);
    matcher.addURI(string2, "retail", 4);
}
```

В коде часто может быть спрятано что-то более неожиданное (наблюдения из личного опыта), например:

- Имена разработчиков проекта
- Полный путь к проекту
- “client_secret” для протокола Oauth2
- PDF-книга “Как разрабатывать под Android” (наверное, чтобы всегда была под рукой)

Теперь мы знаем, что может прятаться в Android-приложениях и можем переходить к главному, а именно к способам сокрытия этих данных.

Способы сокрытия данных

Вариант 1: Ничего не скрывать, оставить все на виду

В таком случае я просто покажу вам эту картинку

“Помогите Даше найти бизнес-логику”

Это нетрудозатратное и совершенно бесплатное решение подойдет для:

- *Простых приложений, которые не взаимодействуют с сетью и не хранят чувствительную пользовательскую информацию;*
- *Приложений, которые используют только публичное API.*

Вариант 2: Использовать ProGuard с правильными настройками

Это решение все-таки имеет право на жизнь, потому что, в первую очередь, оно является простым и бесплатным. Несмотря на вышеупомянутые минусы, у него есть весомый плюс: при правильной настройке ProGuard-правил приложение может действительно стать обфусцированным.

Однако, нужно понимать, что такое решение после каждой сборки требует от разработчика декомпиляции и проверки, все ли нормально. Потратив несколько минут на изучение APK файла, разработчик (и его компания) могут стать увереннее в безопасности своего продукта.

Как изучать APK-файл

> Проверить приложение на наличие обфускации достаточно просто. Для того, чтобы достать APK-файл из проекта существует несколько путей:
> (*) взять из директории проекта (в Android Studio обычно название папки “build”);
> (*) установить приложение на смартфон и достать APK с помощью приложения “Apk Extractor”.
> После этого, пользуясь утилитой Apktool, получаем Smali-код (инструкция по получению здесь https://ibotpeaches.github.io/Apktool/documentation) и пытаемся найти что-нибудь подозрительно читаемое в строках проекта. Кстати, для поиска читаемых кодов можно запастись уже заранее готовыми bash-командами.

Это решение подойдет для:

- *Приложений игрушек, приложений интернет-магазинов и т.п.;*
- *Приложений, которые действительно являются тонкими клиентами, и все данные прилетают исключительно с серверной стороны;*
- *Приложений, которые не пишут на всех своих баннерах “Безопасное приложение №1”.*

Вариант 3: Использовать Open Source Obfuscator

К сожалению, реально хороших бесплатных обфускаторов для мобильных приложений я не знаю. А обфускаторы, которые можно найти в сети могут принести вам много головной боли, поскольку собрать такой проект под новые версии API будет слишком сложно.

Исторически сложилось, что существующие крутые обфускаторы сделаны под машинный код (для C/C++). Хорошие примеры:

- Obfuscator-LLVM, https://github.com/obfuscator-llvm/obfuscator
- Movfuscator, https://github.com/xoreaxeaxeax/movfuscator

Например, Movfuscator заменяет все opcodes mov-ами, делает код линейным, убирая все ветвления. Однако, крайне не рекомендуется использовать такой способ обфускации в боевом проекте, потому что тогда код рискует стать очень медленным и тяжелым.

Это решение подойдет для приложений, у которых основная часть кода — NDK.

Вариант 4: Использовать проприетарное решение

Это самый грамотный выбор для серьезных приложений, так как проприетарное ПО:
а) поддерживается;
б) всегда будет актуально.

Пример обфусцированного кода при использовании таких решений:

```
Manifest | Resources | Certificate | Assembly | Decompiled Java | Strings | Constants | Notes

        while(true) {
            switch(1) {
                case 0: {
                    goto label_51;
                }
                case 1: {
                    goto label_54;
                }
            }
        }

    label_54:
        this.ba04300430aa04300430(arg8.b0436043604360436ж0436ж(mmmmpp.a0436043604360436ж0436ж(v0, v1, v6)));
        this.ba0430a0430a04300430(arg8.b0436043604360436ж0436ж(mmmmpp.mb_string_decrypt("[illegible]",
                '!', 'ß', '\u0000')));
        this.baaaa043004300430(arg8.bжжж0436ж0436ж(mmmmpp.a0436043604360436ж0436ж("¿°®«¾¯\u0097¹¬³¶¯\u0098¿·¬¯¾",
                '%', '\u0000')));
        this.b0430aaa043004300430(arg8.bжжж0436ж0436ж(mmmmpp.a0436043604360436ж0436ж("ēĝïėċēĖíċĔĞğĜdïĘċČĖdĎ",
                'Ü', '\u0000')));
    }

    public void b04170417ЗЗ04173(String arg4) {
    label_2:
        switch(0) {
            case 0: {
                break;
            }
            case 1: {
                goto label_2;
            }
            default: {
            label_3:
                switch(1) {
                    case 0: {
                        goto label_2;
                    }
                    case 1: {
                        goto label_5;
                    }
                    default: {
                        goto label_3;
```

В этом фрагменте кода можно увидеть:

1. Максимально непонятные названия переменных (с наличием русских букв);
2. Китайские символы в строчках, не дающие понять, что реально происходит в проекте;
3. Очень много добавленных в проект ловушек (“switch”, “goto”), которые сильно меняют codeflow приложения.

Это решение подойдет для:

- *Банков;*
- *Страховых компаний;*
- *Мобильных операторов, приложений для хранения паролей и т. д.*

Вариант 5: Использовать React-Native

Я решил выделить этот пункт, так как написание кроссплатформенных приложений сейчас стало действительно популярным занятием.

Кроме очень большого community, JS имеет очень большое количество открытых обфускаторов. Например, они могут превратить ваше приложение в смайлики:

```
ﾟωﾟﾉ= /｀m´）ﾉ ~┻━┻ //*´∇｀*/ ['_']; o=(ﾟｰﾟ) =_=3; c=(ﾟΘﾟ) =(ﾟｰﾟ)-(ﾟｰﾟ); (ﾟДﾟ) =(ﾟΘﾟ)= (o^_^o)/
(o^_^o);(ﾟДﾟ)={ﾟΘﾟ: '_' ,ﾟωﾟﾉ : ((ﾟωﾟﾉ==3) +'_') [ﾟΘﾟ] ,ﾟｰﾟﾉ :(ﾟωﾟﾉ+ '_')[o^_^o -(ﾟΘﾟ)] ,ﾟДﾟﾉ:
((ﾟｰﾟ==3) +'_')[ﾟｰﾟ] }; (ﾟДﾟ) [ﾟΘﾟ] =((ﾟωﾟﾉ==3) +'_') [c^_^o];(ﾟДﾟ) ['c'] = ((ﾟДﾟ)+'_') [ (ﾟ
ｰﾟ)+(ﾟｰﾟ)-(ﾟΘﾟ) ];(ﾟДﾟ) ['o'] = ((ﾟДﾟ)+'_') [ﾟΘﾟ];(ﾟoﾟ)=(ﾟДﾟ) ['c']+(ﾟДﾟ) ['o']+(ﾟωﾟﾉ +'_')[ﾟ
Θﾟ]+ ((ﾟωﾟﾉ==3) +'_') [ﾟｰﾟ] + ((ﾟДﾟ) +'_') [(ﾟｰﾟ)+(ﾟｰﾟ)]+ ((ﾟｰﾟ==3) +'_') [ﾟΘﾟ]+((ﾟｰﾟ==3) +'_')
[(ﾟｰﾟ) - (ﾟΘﾟ)]+(ﾟДﾟ) ['c']+((ﾟДﾟ)+'_') [(ﾟｰﾟ)+(ﾟｰﾟ)]+ (ﾟДﾟ) ['o']+((ﾟｰﾟ==3) +'_') [ﾟΘﾟ];(ﾟДﾟ)
['_'] =(o^_^o) [ﾟoﾟ] [ﾟoﾟ];(ﾟεﾟ)=((ﾟｰﾟ==3) +'_') [ﾟΘﾟ]+ (ﾟДﾟ) .ﾟДﾟﾉ+((ﾟДﾟ)+'_') [(ﾟｰﾟ) + (ﾟｰﾟ)]+
((ﾟｰﾟ==3) +'_') [o^_^o -ﾟΘﾟ]+((ﾟｰﾟ==3) +'_') [ﾟΘﾟ]+ (ﾟωﾟﾉ +'_') [ﾟΘﾟ]; (ﾟｰﾟ)+=(ﾟΘﾟ); (ﾟДﾟ)[ﾟ
εﾟ]='\\'; (ﾟДﾟ).ﾟΘﾟﾉ=(ﾟДﾟ+ ﾟｰﾟ)[o^_^o -(ﾟΘﾟ)];(oﾟｰﾟo)=(ﾟωﾟﾉ +'_')[c^_^o];(ﾟДﾟ) [ﾟoﾟ]='\"';(ﾟДﾟ)
['_'] ( (ﾟДﾟ) ['_'] (ﾟεﾟ+(ﾟДﾟ)[ﾟoﾟ]+ (ﾟДﾟ)[ﾟεﾟ]+(ﾟΘﾟ)+ (ﾟｰﾟ)+ (ﾟΘﾟ)+ (ﾟДﾟ)[ﾟεﾟ]+(ﾟΘﾟ)+ ((ﾟｰﾟ) +
(ﾟΘﾟ))+ (ﾟｰﾟ)+ (ﾟДﾟ)[ﾟεﾟ]+(ﾟΘﾟ)+ (ﾟｰﾟ)+ ((ﾟｰﾟ) + (ﾟΘﾟ))+ (ﾟДﾟ)[ﾟεﾟ]+(ﾟΘﾟ)+ ((o^_^o) +(o^_^o))+
((o^_^o) - (ﾟΘﾟ))+ (ﾟДﾟ)[ﾟεﾟ]+(ﾟΘﾟ)+ ((o^_^o) +(o^_^o))+ (ﾟｰﾟ)+ (ﾟДﾟ)[ﾟεﾟ]+((ﾟｰﾟ) + (ﾟΘﾟ))+
(c^_^o)+ (ﾟДﾟ)[ﾟεﾟ]+((o^_^o) +(o^_^o))+ (c^_^o)+ (ﾟДﾟ)[ﾟεﾟ]+((ﾟｰﾟ) + (ﾟΘﾟ))+ (ﾟΘﾟ)+ (ﾟДﾟ)[ﾟoﾟ])
(ﾟΘﾟ)) ('_');
```

Мне бы очень хотелось посоветовать вам данное решение, но тогда ваш проект будет работать самую малость быстрее черепахи.

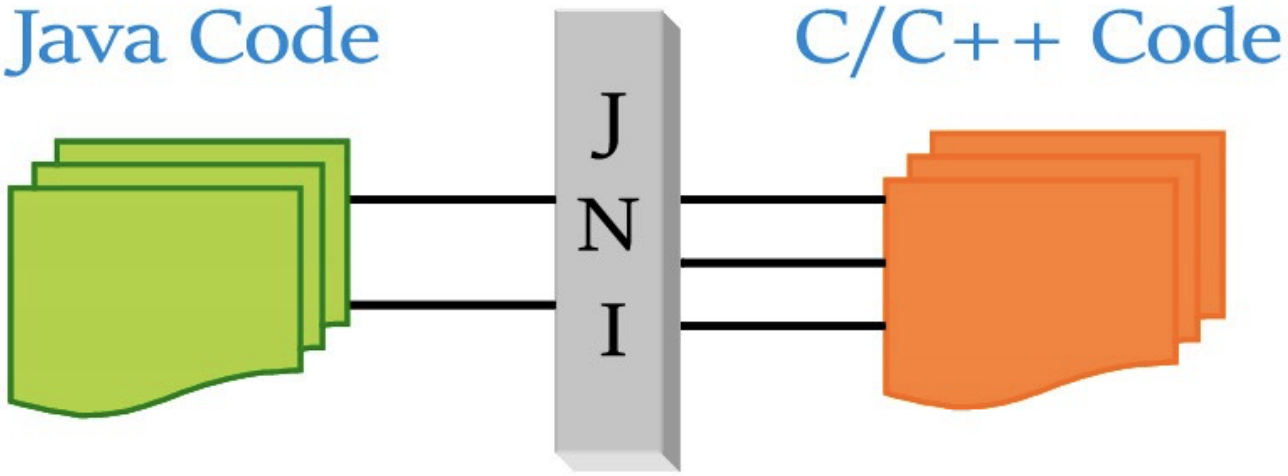
Зато, уменьшив требование к обфускации кода, мы можем создать действительно хорошо защищенный проект. Так что гуглим “js obfuscator” и обфусцируем наш выходной bundle-файл.

Это решение подойдет для тех, кто готов писать кроссплатформенное приложение на React Native.

Xamarin
Было бы очень интересно узнать про обфускаторы на Xamarin, если у вас есть опыт их использования – расскажите, пожалуйста, о нем в комментариях.

Вариант 6: Использовать NDK

Мне самому часто приходилось использовать NDK в своем коде. И я знаю, что некоторые разработчики считают, что использование NDK спасает их приложение от реверсеров. Это не совсем так. Для начала нужно понять, как именно работает сокрытие с помощью NDK.



Оказывается, очень просто. В коде есть некоторая JNI-договоренность, что при вызове C/C++ кода в проекте он будет преобразовываться следующим образом.

Нативный класс NativeSummator:

```
public class NativeSummator {
    native int sum(int a, int b);
    static native int staticSum(int a, int b);
}
```

Реализация нативного метода sum:

```
int
Java_com_m039_study_Summator_sum(JNIEnv* env,
                                 jobject thiz,
                                 int a,
                                 int b) {
    return a + b;
}
```

Реализация нативного статичного метода sum:

```
int
Java_com_m039_study_StaticSummator_sum(JNIEnv* env,
                                       jclass thiz,
                                       int a,
                                       int b) {
    return a + b;
}
```

Становится понятно, что для вызова нативного метода используется поиск функции Java_<package name>_<Static?><class>_<method> в динамической библиотеке.

Если заглянуть в Dalvik/ART код, то мы найдём следующие строки:

```
/*
 * First, we try it without the signature.
 */
preMangleCM =
    createJniNameString(meth->clazz->descriptor, meth->name, &len);
if (preMangleCM == NULL)
    goto bail;

mangleCM = mangleString(preMangleCM, len);
if (mangleCM == NULL)
    goto bail;

ALOGV("+++ calling dlsym(%s)", mangleCM);
func = dlsym(pLib->handle, mangleCM);
```

(источник)

Сначала сгенерируем из Java-объекта следующую строку Java_<package name>_<class>_<method>, а затем попытаемся разрезолвить метод в динамической библиотеке с помощью вызова “dlsym”, который попытается найти нужную нам функцию в NDK.

Так работает JNI. Его основная проблема в том, что, декомпилировав динамическую библиотеку, мы увидим все методы, как на ладони:

0xa40	P	JNI_OnLoad
0xa50	P	Java_io_card_payment_CardScanner_nUseNeon
0xa71	P	sub_a71
0xa73	P	sub_a73
0xa80	P	Java_io_card_payment_CardScanner_nUseTegra
0xaa1	P	sub_aa1
0xaa3	P	sub_aa3
0xab0	P	Java_io_card_payment_CardScanner_nUseX86
0xac0	P	android_getCpuFamily
0xb00	P	sub_b00
0xe00	P	android_getCpuFeatures
0xe41	P	sub_e41
0xe43	P	sub_e43
0xe50	P	android_getCpuCount
0xe90	P	android_setCpu
0xf00	P	sub_f00

Значит, нам нужно придумать такое решение, чтобы адрес функции был обфусцирован.

Сначала я пытался записать данные напрямую в нашу JNI-таблицу, но, понял, что механизмы ASLR и разные версии Android просто-напросто не позволят мне сделать этот способ работающим на всех устройствах. Тогда я решил узнать, какие методы NDK предоставляет разработчикам.

И, о чудо, нашелся метод “RegisterNatives”, который делает ровно то, что нам нужно (вызывает внутреннюю функцию dvmRegisterJNIMethod).

Определяем массив, описывающий наш нативный метод:

```
static JNINativeMethod methods[] = {
        {"getNativeKey1", "()Ljava/lang/String;", (void *)&hideFunc},
};
```

И регистрируем наш объявленный метод в функции JNI_OnLoad (метод вызывается после инициализации динамической библиотеки, тыц) :

```
jclass clz = (*env)->FindClass(env, "com/tyagiabhinav/hellosecretkeys/MainActivity");
res = (*env)->RegisterNatives(env, clz, methods, sizeof(methods) / sizeof(methods[0]));
if (res != JNI_OK) {
    return JNI_ERR;
}
```

Ура, мы самостоятельно спрятали функцию “hideFunc”. Теперь применим наш любимый llvm-обфускатор и порадуемся безопасности кода в конечном виде.

Это решение подойдет для приложений, которые уже используют NDK (подключение NDK несет в проект большое количество сложностей, поэтому для не-NDK приложений это решение не так актуально).

Вывод

На самом деле, в приложении не должно храниться никаких чувствительных данных, либо они должны быть доступны только после аутентификации пользователя. Однако, бывает, что бизнес-логика принуждает разработчиков к хранению токенов, ключей и специфических элементов логики кода внутри приложения. Надеюсь, эта статья поможет вам, если вы не хотите делиться такими чувствительными данными и быть “открытой книгой” для ресерчеров.

Я считаю, что обфускация – важная структурная часть любого современного приложения.

Обдуманно подходите к вопросам сокрытия кода и не ищите простых путей!

автор (c) inwady
взято с habr'a