Статья Защищаем приложение для Android от отладчиков, эмуляторов и Frida

xss.is/threads/45371

Когда задумываешься о защите приложения от реверса, в первую очередь на ум приходят такие слова, как обфускация и шифрование. Но это только часть решения проблемы. Вторая половина — это детект и защита от самих инструментов реверса: отладчиков, эмуляторов, Frida и так далее. В этой статье мы рассмотрим техники, которые мобильный софт и зловреды используют, чтобы спрятаться от этих инструментов.

Важный момент: я приведу множество разных техник защиты, и у тебя может возникнуть соблазн запихнуть их все в один класс (или нативную библиотеку) и с удобством для себя запускать один раз при старте приложения. Так делать не стоит, механизмы защиты должны быть разбросаны по приложению и стартовать в разное время. Так ты существенно усложнишь жизнь взломщику, который в противном случае мог бы определить назначение класса/библиотеки и целиком заменить его на одну большую заглушку.

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:
Code:

```
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;
}
```

Либо использовать такую функцию, позаимствованную из приложения rootinspector:
Code:

```
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:
Code:

```
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 диалог 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:
Code:

```
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;
    }
}
```

Искать можно и по косвенным признакам. Например, SuperSU, некогда популярное решение 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 тестовыми ключами. Это не всегда подтверждает наличие root, но точно говорит о том, что на устройстве установлен кастом:
Code:

```
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 ты не столкнешься с устройством, рутованным с помощью Magisk. Это так называемый systemless-метод рутинга, когда вместо размещения компонентов для root-доступа в файловой системе поверх нее подключают другую файловую систему (оверлей), содержащую эти компоненты.

Такой механизм работы не только позволяет оставить системный раздел в целости 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.

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, а исходный код proof of concept выложен на GitHub. Способ работает до сих пор на самой последней версии Magisk — 20.4.

ЭМУЛЯТОР

Реверсеры часто используют эмулятор для запуска подопытного приложения. Поэтому нелишним 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 их значения:
Code:

```
ro.hardware=goldfish
ro.kernel.qemu=1
ro.product.model=sdk
```

Прочитав их значения, можно предположить, что код исполняется в эмуляторе:
Code:

```
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});
}
```

Обрати внимание, что класс android.os.SystemProperties скрытый и недоступен в SDK, поэтому для обращения к нему мы используем рефлексию.

В других эмуляторах значения системных переменных могут быть другими. На этой странице 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.

ОТЛАДЧИК

Один из методов реверса — запуск приложения под управлением отладчика. Взломщик может декомпилировать твое приложение, затем создать в Android Studio одноименный проект, закинуть в него полученные исходники и запустить отладку, 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:
Code:

```
public static boolean checkDebuggable(Context context){
    return (context.getApplicationInfo().flags & ApplicationInfo.FLAG_DEBUGGABLE) != 0;
}
```

Чуть более надежный способ — напрямую спросить систему, подключен ли отладчик:
Code:

```
public static boolean detectDebugger() {
    return Debug.isDebuggerConnected();
}
```

То же самое в нативном коде:
Code:

```
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 помогут обнаружить отладчик на базе протокола JDWP (как раз тот, что встроен в Android Studio). Но другие отладчики работают по-другому, и методы борьбы с ними будут иными. Отладчик GDB, например, получает контроль над процессом с помощью системного вызова ptrace(). А после использования ptrace флаг TracerPid в синтетическом файле /proc/self/status изменится с нуля на PID отладчика. Прочитав значение флага, мы узнаем, подключен ли к приложению отладчик GDB:
Code:

```
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. Ее аналог на языке C будет нетрудно найти на Stack Overflow.

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) и затем попытаться вызвать системный вызов ptrace:
Code:

```
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 способ не работает.

В старых версиях Android также можно было прямо искать процесс отладчика в системе, когда приложение просто проходит по дереву процессов в файловой системе /proc и ищет строки 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 с его помощью устанавливают системные модификации и твики приложений, существует масса модулей, которые могут быть использованы для реверса и взлома твоего приложения. Это и различные модули для отключения SSL Pinning, и трассировщики вроде inspeckage, и самописные модули, которые могут как угодно изменять приложение.

Есть три действенных способа обнаружения Xposed:

- поиск пакета de.robv.android.xposed.installer среди установленных на устройство;
- поиск libexposed_art.so и xposedbridge.jar в файле /proc/self/maps;
- поиск класса de.robv.android.xposed.XposedBridge среди загруженных в рантайм пакетов.

В статье Android Anti-Hooking Techniques in Java приводится реализация третьего метода одновременно для поиска Xposed и Cydia Substrate. Подход интересен тем, что мы не ищем напрямую классы в рантайме, а просто вызываем исключение времени исполнения и затем ищем нужные классы и методы в стектрейсе:
Code:

```
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! Этот изумительный инструмент позволяет перехватить вызов любой функции подопытного приложения, прочесть все ее аргументы и заменить тело собственной реализацией на языке JavaScript. Frida не только занимает почетное 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 приводится три (на самом деле четыре, но первый уже неактуален) способа это сделать.

1. Поиск библиотек frida-agent и frida-gadget в файле /proc/self/maps:

Code:

```
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":
Code:

```
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. Проход по всем открытым TCP-портам, отправка в них dbus-сообщения AUTH и ожидание ответа Frida:
Code:

```
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 (этот способ обычно применяют, когда невозможно получить root на устройстве).

В статье Detect Frida for Android автор приводит еще три способа:

1. Поиск потоков frida-server и frida-gadget, которые Frida запускает 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.

КЛОНИРОВАНИЕ

Некоторые производители встраивают в свои прошивки функцию клонирования приложения (Parallel Apps в OnePlus, Dual Apps в Xiaomi и так далее), которая позволяет установить на смартфон копию выбранного приложения. Прошивка создает 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: создают изолированную среду для приложения и устанавливают его в собственный приватный каталог.

С помощью второго метода твое приложение могут запустить 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:
Code:

```
/data/user/0/com.example.app/files
```

При создании клона с помощью одного из приложений из маркета путь будет уже таким:
Code:

```
/data/data/com.ludashi.dualspace/virtual/data/user/0/com.example.app/files
```

А при создании клона с помощью встроенных в прошивку инструментов — таким:
Code:

```
/data/user/999/com.example.app/files
```

Соберем все вместе и получим такой метод для детекта изолированной среды:
Code:

```
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.

ВЫВОДЫ

Конечно же, это далеко не все способы борьбы с реверсом и анализом поведения приложений. Существует множество менее эффективных или слишком узкоспециализированных способов. Поэтому ниже я приведу список литературы и проектов на GitHub, с которыми обязательно нужно ознакомиться. Там ты найдешь более глубокое объяснение некоторых описанных в этой статье методов защиты, а также множество других техник.

WWW

- Android Anti-Reversing Defenses — глава из свободной книги Mobile Security Testing Guide о защите от реверса;
- Android Anti-Hooking Techniques in Java — статья о способах обнаружить Xposed и Cydia Substrate;
- The Jiu-Jitsu of Detecting Frida — описание способов обнаружить Frida;
- Detect Frida for Android — еще несколько способов обнаружить Frida;
- SafetyNet: Google's tamper detection for Android — статья о принципе работы инструмента SafetyNet, использующего многие из описанных техник определить компрометацию устройства;
- RootBeer — готовая библиотека, помогающая обнаружить права root;
- anti-emulator — репозиторий с несколькими техниками детекта эмулятора и дебаггера;
- DetectMagiskHide — готовое приложение для детекта Magisk.

Автор: Евгений Зобнин
zobnin.github.io
взято с хакер.ру