

# Android app反调试与代码保护的一些基本方案

原创：jackycao 腾讯Bugly 3月28日

| 导语 本文介绍Android app代码（java + ndk）的反调试的方法和保护代码增加逆向难度的一些基本方法。

Android自问世以来得以迅速发展，各大手机厂商纷纷投入成本开发、设计开发自己的Android系统，从2016年开始，Android已经超越ios成为全球最有影响力的操作系统。针对于Android app的逆向方法和逆向工具很多，所以反调试对于Android的代码保护扮演着很重要的角色。本文从四个方面介绍一下Android反调试的一些方法。

ps：反调试并不能完全阻止逆向行为，只是在长期的攻防战中给破解人员不断的增加逆向难度。

## Java：

### (1) Proguard

借助 Android studio 的 proguard 工具，对 Java 代码分别进行压缩（Shrink）、优化（Optimize）、混淆（Obfuscate）、检查（Veirfy）。

压缩（Shrink）：去掉代码中无用的类、函数方法和字段。

优化（Optimize）：对Android的可执行文件dex进行优化，去掉无用指令。

混淆（Obfuscate）：用毫无意义的字段对代码的类名、函数名、变量名重命名，比如用a, b, c 这种。

检查（Veirfy）：对混淆后的代码进行检查。

经过Proguard后，代码程序依然可以重新组织和处理，处理后的程序逻辑与之前完全一致，而混淆后的代码即便反编译后依然很难阅读。同时，在混淆过程中对于一些不影响正常运行的信息将永久丢失，这些信息的丢失使得程序更加难以理解。

同时，Proguard还可以控制对某个类混淆，以及对某个类的某些函数方法混淆。

下图是一张混淆前和混淆后的对比图：

混淆前:

```

public RawInputThread(String tag, int port) {
    Log.d(TAG, "RawInputSocketThread: new...");
    this.m_tag = tag;
    this.m_host_ip = getProperty("android.host.server");
    this.m_use_pipe = this.m_host_ip.equals("virtpipe");
    this.m_pipe = null;
    if (!this.m_use_pipe) {
        this.m_run_as_server = this.m_host_ip.length() == 0;
        if (this.m_run_as_server) {
            this.m_port = port;
        } else {
            this.m_port = Integer.parseInt(getProperty("android.host.apps.port"));
        }
        String str = TAG;
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.append("RawInputSocketThread: host_ip=");
        stringBuilder.append(this.m_host_ip);
        stringBuilder.append(", port=");
        stringBuilder.append(this.m_port);
        stringBuilder.append(", run_as_server=");
        stringBuilder.append(this.m_run_as_server);
        Log.d(str, stringBuilder.toString());
    }
    this.m_server_socket = null;
    this.m_socket = null;
    this.m_buffer = new byte[20480];
    this.m_size = 0;
    this.m_stopping = false;
}

public void set_stopping(boolean stopping) {
    if (this.m_server_socket != null) {
        try {
            this.m_server_socket.close();
        } catch (Exception e) {
        }
        this.m_server_socket = null;
    }
    Close();
    this.m_stopping = stopping;
}

public void Sleep(long millis) {
    try {
        Thread.sleep(millis);
    } catch (Exception e) {
    }
}
}

```

腾讯Bugly


混淆后:

```

public a(String str, int i) {
    Log.d("RawInputThread", "RawInputSocketThread: new...");
    this.a = str;
    this.b = c("android.host.server");
    this.i = this.b.equals("virtpipe");
    this.j = null;
    if (!this.i) {
        this.k = this.b.length() == 0;
        if (this.k) {
            this.c = i;
        } else {
            this.c = Integer.parseInt(c("android.host.apps.port"));
        }
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.append("RawInputSocketThread: host_ip=");
        stringBuilder.append(this.b);
        stringBuilder.append(", port=");
        stringBuilder.append(this.c);
        stringBuilder.append(", run_as_server=");
        stringBuilder.append(this.k);
        Log.d("RawInputThread", stringBuilder.toString());
    }
    this.d = null;
    this.e = null;
    this.f = new byte[20480];
    this.g = 0;
    this.h = false;
}

public void a() {
    /* JADX: method processing error */
    /*
    Error: java.lang.NullPointerException
    at jadx.core.dex.visitors.regions.ProcessTryCatchRegions.searchTryCatchDominators(ProcessTryCatchRegions.java:75)
    at jadx.core.dex.visitors.regions.ProcessTryCatchRegions.process(ProcessTryCatchRegions.java:45)
    at jadx.core.dex.visitors.regions.RegionMakerVisitor.postProcessRegions(RegionMakerVisitor.java:63)
    at jadx.core.dex.visitors.regions.RegionMakerVisitor.visit(RegionMakerVisitor.java:58)
    at jadx.core.dex.visitors.DepthTraversal.visit(DepthTraversal.java:31)
    at jadx.core.dex.visitors.DepthTraversal.visit(DepthTraversal.java:17)
    at jadx.core.ProcessClass.process(ProcessClass.java:34)
    at jadx.api.JadxDecompiler.processClass(JadxDecompiler.java:282)
    at jadx.api.JavaClass.decompile(JavaClass.java:62)
    at jadx.api.JavaClass.getCode(JavaClass.java:48)
    */
    /*
    r2 = this;
    r0 = r2.i;    Catch:{ Exception -> 0x001c }
    r1 = 0;       Catch:{ Exception -> 0x001c }
    if (r0 == 0) goto L_0x0011;    Catch:{ Exception -> 0x001c }
    L_0x0005:
    r0 = r2.j;    Catch:{ Exception -> 0x001c }
    if (r0 == 0) goto L_0x001c;    Catch:{ Exception -> 0x001c }
    L_0x0009:
    r0 = r2.j;    Catch:{ Exception -> 0x001c }
    r0.close();  Catch:{ Exception -> 0x001c }
    r2.j = r1;   Catch:{ Exception -> 0x001c }
    */
}

```



## (2) isDebuggerConnected

Android Debug类提供isDebuggerConnected函数，函数原型如下：

```

/**
 * Determine if a debugger is currently attached.
 */
public static boolean isDebuggerConnected() {
    return VMDebug.isDebuggerConnected();
}

```



在VMDebug类里的isDebuggerConnected的具体实现在ndk程序里。

```
/**
 * Determines if a debugger is currently attached.
 *
 * @return true if (and only if) a debugger is connected
 */
public static native boolean isDebuggerConnected();
```

这里暂且不跟进该函数，总之，isDebuggerConnected函数用于检测此刻是否有调试器挂载到程序上，如果返回值为true则表示此刻被调试中。用法很简单，如下：

```
if(android.os.Debug.isDebuggerConnected()) {
    android.os.Process.killProcess(android.os.Process.myPid());
}
```

### (3) android:debuggable属性

在 Android 的 AndroidManifest.xml 清单文件的 application 节点下加入 android:debuggable="false" 属性，使程序不能被调试。在Java程序代码里也可检测该属性的值，如下：

```
if(getApplicationInfo().flag &= ApplicationInfo.FLAG_DEBUGGABLE != 0) {
    Log.i(TAG, "debug debug debug");
    android.os.Process.killProcess(android.os.Process.myPid());
}
```

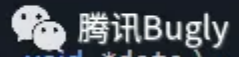
## NDK:

### (1) ptrace函数

Linux内核的ptrace函数原型：

```
#include <sys/ptrace.h>
```

```
long ptrace(enum _ptrace_request request, pid_t pid, void * addr, void *data);
```



ptrace允许A进程控制B进程，并且A进程可以检查和修改B进程的内存和寄存器。但是一个进程只能被一个进程调试，所以根据这个特点，可以让进程自己ptrace自己，传入的request设置为PTRACE\_TRACEME，程序被自己附加调试后，其他的调试操作就会失败了。

## (2) 文件节点检测

一旦程序处于被调试状态，Linux会向进程的节点写入数据，比如/proc/<pid>/status内容中的TracePid会写入调试进程的pid，如果TracePid的值不为0，就表明进程处于被调试状态了。

此外，通用的检测逻辑还有检测调试的端口号，Linux的文件节点/proc/net/tcp会记录着正在运行的进程的本地的端口号，调试工具IDA的默认的调试端口是23946，通过读取/proc/net/tcp内容，检测是否有23946，如果找到了就表明进程处于被调试状态了。

## (3) Inotify

Linux的Inotify用于检测文件系统变化。它可以检测单个文件，也可以检测整个目录。

逆向最常做的一件事就是dump 内存，使用dd命令（或者如果使用gdb的话为gcore命令），dump掉/proc/<pid>/mem或/proc/<pid>/mpas或/proc/<pid>/pagemap的内容。

这里，就可以使用Inotify API对上述三个文件监控，如果有发现打开、读写操作，极大概率就是进程正在被破解。

## (4) so文件hash值检测

so文件在被JNI\_Onload加载后，so文件的函数的指令是固定的，若被调试器挂载，下了断点后指令会发生改变（断点地址会被改写为bkpt指令），计算内存中加载的so的hash值，进行校验检测函数是否被修改或被下断点即可判断出是否被调试状态。

## (5) 时间差检测

一个取巧的方法，正常情况下，一段程序在两条代码之间的时间差是很短的，而对于调试程序来说，单步调试中的程序两条代码之间的时间差会比较大，检测两条代码之间的时间差，可以大概率判断程序是否被调试。



## Resource资源文件：

Android资源文件经常被恶意篡改，植入各种广告，插件，严重影响了Android app生态平衡。

## APK签名检测

Android SDK 中有 apk 签名检测的方法，Framework 的 PackageManager 类提供了 getPackageInfo()函数，函数原型：

```

/**
 * Retrieve overall information about an application package that is
 * installed on the system.
 * <p>
 * Throws {@link NameNotFoundException} if a package with the given name can
 * not be found on the system.
 *
 * @param packageName The full name (i.e. com.google.apps.contacts) of the
 *     desired package.
 * @param flags Additional option flags. Use any combination of
 *     {@link #GET_ACTIVITIES}, {@link #GET_GIDS},
 *     {@link #GET_CONFIGURATIONS}, {@link #GET_INSTRUMENTATION},
 *     {@link #GET_PERMISSIONS}, {@link #GET_PROVIDERS},
 *     {@link #GET_RECEIVERS}, {@link #GET_SERVICES},
 *     {@link #GET_SIGNATURES}, {@link #GET_UNINSTALLED_PACKAGES} to
 *     modify the data returned.
 * @return Returns a PackageInfo object containing information about the
 *     package. If flag GET_UNINSTALLED_PACKAGES is set and if the
 *     package is not found in the list of installed applications, the
 *     package information is retrieved from the list of uninstalled
 *     applications (which includes installed applications as well as
 *     applications with data directory i.e. applications which had been
 *     deleted with {@code DONT_DELETE_DATA} flag set).
 * @see #GET_ACTIVITIES
 * @see #GET_GIDS
 * @see #GET_CONFIGURATIONS
 * @see #GET_INSTRUMENTATION
 * @see #GET_PERMISSIONS
 * @see #GET_PROVIDERS
 * @see #GET_RECEIVERS
 * @see #GET_SERVICES
 * @see #GET_SIGNATURES
 * @see #GET_UNINSTALLED_PACKAGES
 */
public abstract PackageInfo getPackageInfo(String packageName, int flags)
    throws NameNotFoundException;

```

第二个参数传入GET\_SIGNATURES时，返回对象的signature字段就是签名信息，计算其hash值，前后对比hash值。实际可用的两种方案：

- (1) 在本地Java代码里进行校验，不一致则强退应用；
- (2) 把签名信息发到服务器后台，服务器后台记录着正确的签名信息，比对后不一致则返回一个错误给错误。

上述即为对于Android app的反调试，代码保护的一些基本策略。

如果您觉得我们的内容还不错，就请转发到朋友圈，和小伙伴一起分享吧~

有趣 | 有料 | 有收获



长按此处关注[腾讯bugly](#)