基于安卓系统的代码隐藏类规避技术检测框架

马晓凯 杨哲慜
(复旦大学软件学院 上海 201203)

摘 要 随着恶意软件检测和分析技术的发展,大量恶意软件采用规避技术来对抗安全分析。其中,代码隐藏类规避技术将应用代码对静态分析隐藏起来,使分析结果错误或缺失。爆炸式增长的恶意软件数量要求了对代码隐藏类规避技术的自动化检测。通过对 142 个恶意样本进行人工分析,总结出一种代码隐藏类规避技术的检测方法,并实现了一个通用的自动化检测框架。使用检测框架在第三方应用市场 2 278 个样本上进行了实验,发现有 34.9% 的样本使用了代码隐藏类规避技术。

关键词 安卓 规避技术 静态分析 动态分析

中图分类号 TP311 **文献标识码** A DOI:10.3969/j.issn.1000-386x.2017.11.058

A DETECTION FRAMEWORK OF CODE-HIDING EVASION TECHNIQUE BASED ON ANDROID SYSTEM

Ma Xiaokai Yang Zhemin
(*Software School*, *Fudan University*, *Shanghai 201203*, *China*)

Abstract With the development of malware detection and analysis techniques, a large number of malwares use evasion techniques to fight against security analysis. Among these evasion techniques, code-hiding evasion techniques hide application code from static analysis, thus cause analysis results wrong or incomplete. The explosive growth of malware required automated detection of code-hiding evasion techniques. Through manual analysis of 142 malicious samples, this paper summarized an approach for detecting code-hiding evasion techniques and implemented a generic automated detection framework. We use the detection framework to do experiments on 2 278 samples in a third party applications market, and find that 34.9% samples use code-hiding evasion techniques.

Keywords Android Evasion technique Static analysis Dynamic analysis

0 引 言

以移动操作系统为平台,开发者可以开发应用来扩展移动设备的功能。在众多移动操作系统中,安卓系统由于其开放性最受厂商和用户的青睐,占据了移动操作系统 86.8% 的市场份额[1]。与此同时,基于安卓系统的第三方应用数量正呈飞速增长,仅以 Google 官方应用市场 Google Play 为例,其应用数量已超过 260 万个[2],且仍处于飞速增长中。

移动设备中含有大量的用户隐私数据,出于保护用户隐私的目的,安全分析人员利用程序分析方法来分析应用程序行为,识别恶意软件。程序分析技术主要包含静态和动态程序分析两类。其中,静态分析相较于动态分析有代码覆盖全、分析效率高的优点,是目前可用于大规模检测分析的重要手段。

然而,无论是正常应用开发者还是恶意软件作者都具有规避软件检测的需求。其中,正常应用需要对抗逆向工程,保护软件核心代码。而恶意软件作者通过规避分析检测,可以延长其恶意代码的生命周期。此类规避分析技术的方法通常被称为规避技术。其中,代码隐藏类规避技术作为一类主要的规避分析方法,通过把代码对静态分析隐藏起来,使分析结果错误或缺失,降低了分析的准确度。

收稿日期:2017 - 07 - 10。**马晓凯**,硕士,主研领域:移动安全,规避技术。**杨哲慜**,博士。

代码隐藏类规避技术包括动态代码加载技术、代码自修复技术和垃圾代码插入技术三种。动态代码加载技术和代码自修复技术由于将程序关键代码隐藏起来,使得静态分析在中间环节生成的图不完整或错误,导致对样本的误判。垃圾代码插入技术会在静态分析的词法、语法分析阶段对分析工具产生干扰,如果分析工具的实现没有完整地考虑垃圾代码插入技术所带来的威胁,就会有在运行时崩溃的风险。这三种规避分析的技术都具有隐藏程序关键代码,从而使静态程序分析方法无法完整分析应用程序逻辑的能力。

为了解决代码隐藏类规避技术给静态分析带来的不利现状,我们提出了通用的自动化检测框架。该框架通过自动化识别规避检测技术的应用,并以警报的方式提醒分析人员,以辅助分析人员早期发现具有规避行为的恶意软件。

我们对 142 个恶意样本进行了人工分析,归纳了其中代码隐藏类规避技术的使用模式,并提出了一个通用的自动化检测框架。我们的检测框架利用安卓系统中 Dalvik 虚拟机代码加载和执行的模型,使用主动触发类加载和初始化的方法促使代码的加载和修复。我们修改了安卓系统中 Dalvik 虚拟机和 libc 的源代码,对样本中加载和修复代码的行为进行监控,可以有效地识别出样本是否有隐藏代码的行为。

我们从中国第三方应用市场应用宝[3]中收集了 2 278个样本,并对其进行了自动化分析。实验表明,检测框架成功分析了 2 247 个样本,识别出 794 个使用了代码隐藏类规避技术的样本。

1 背景及相关工作

1.1 DEX 文件

由 Dalvik 虚拟机解释执行的字节码也被称作 DEX 字节码(Dalvik Executable Bytecode)[4],由 DEX 字节码所组成的 DEX 文件是运行于安卓系统 Dalvik 虚拟机中的可执行文件,也是安卓应用程序的核心所在。在这一部分,我们简要地介绍 DEX 文件。

安卓应用通常由 Java 语言写成,从 Java 源代码到 DEX 文件的基本映射关系如图 1 所示。Java 编译器将 Java 源代码编译成. class 文件,然后 dx 工具将. class 文件转换为. dex 文件,. dex 文件是. class 文件在安卓系统中的优化。

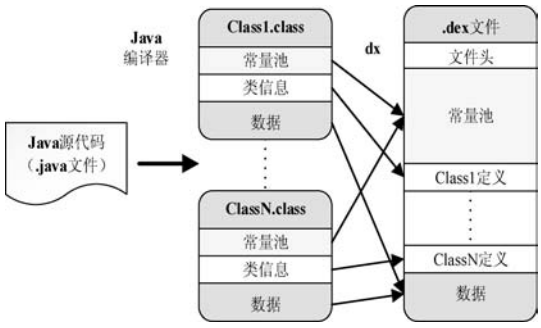

图 1 从 Java 源文件到生成 DEX 文件的基本映射关系

DEX 文件从整体上看是个索引结构,包含了文件头、常量池、类属性表、方法表、类定义表和数据段等部分。文件头包含了魔数、校验值、其他各部分的索引和长度等。常量池、类属性表、方法表和类定义表描述了字符串常量、类的属性、类的方法和类的定义等信息。类名、方法名、常量字符串等都存储在非冗余的常量池中,这样能够充分地减少存储空间。数据段包含了所有的类定义和字节码。一段完整的类方法,其代码必定包含了如表 1 所列的所有字段。

表 1 类方法代码的必含字段

字段	字节数	描述
registersSize	2	使用寄存器个数
insSize	2	参数个数
outsSize	2	调用其他方法时使用的寄存器个数
triesSize	2	try/catch 个数
debugInfoOff	4	指向调试信息的偏移
insnsSize	4	指令个数,以 2 字节为单位
insns	insnsSize * 2	Dalvik 虚拟机指令

代码隐藏类规避技术通常会修改 DEX 文件,来对静态分析产生干扰。较为高级的代码隐藏类规避技术往往会修改表 1 中所描述的字段。例如:1) 将 debugInfoOff 的值修改为非法值,使分析工具运行错误;2) 向 insns 内填充一些空指令和非法指令,使分析工具无法处理该部分代码。

1.2 代码隐藏类规避技术

在安卓系统中,代码隐藏类规避技术通过修改 APK 安装包中的 DEX 文件,将程序的核心代码对静态分析隐藏起来,使静态分析的结果错误或缺失,从而达到保护核心代码、隐藏程序行为的目的。代码隐藏类规避技术包括动态代码加载技术、代码自修复技术和垃圾代码插入技术三种技术。

动态代码加载技术是指进程在运行时额外加载代码执行的技术。利用了额外加载的特性,开发者通常会将需要保护的代码加密,在运行时解密,并通过隐蔽

的手段来加载解密后的代码。也就是说,动态代码加载的过程可以分为获取、解密、加载和执行四个阶段。应用程序可以通过网络或者读取本地文件的方式来获取加密后的代码,而解密和加载的过程可以不接触文件系统,完全在内存中执行,从而提高了安全分析的难度。更复杂的情况是,加载的代码可以分为几段,不同段代码的加载可以以不同的方式触发。

代码自修复技术的程序会在进程执行的过程中动态修改内存中的代码,这意味着即使分析人员在程序启动和执行过程中所观察到的是同一块代码,其内容也可能有所不同。对静态分析来说,代码自修复技术隐藏了部分"真实"的代码,其分析结果会有所缺失。如果开发者将被隐藏的代码替换为无关的程序行为,静态分析还会被误导。

图 2 展示了一段字节码在修改前与修改后的状态。在该示例中,某个类的 interceptSMS( )方法被填充为空指令 nop,即不做任何工作。当应用程序启动后,在 Application. oncreate( )等程序入口处会修改 interceptSMS( )方法的代码,将代码替换为监听短信接收通知的代码。这样,程序在运行过程中通过修改自身代码的行为就可以将其恶意行为伪装起来,从而导致静态分析对样本的误判。

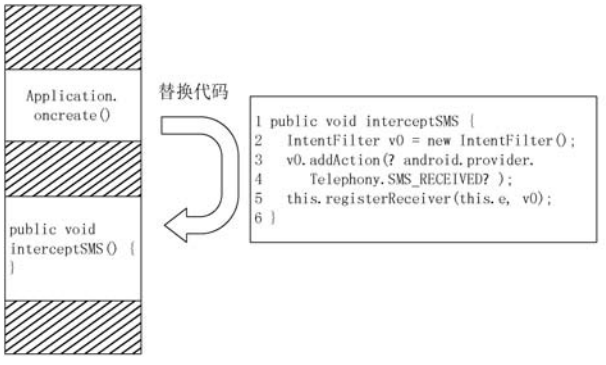

图 2　程序在运行过程中修改自己的代码

垃圾代码插入技术通过向代码的特定位置插入垃圾字节,使反汇编工具生成错误信息,从而对静态分析产生干扰。插入垃圾代码的位置因不同分析工具所使用算法的不同而不同,而插入的代码必然是不能执行的无效代码,否则会导致未知的执行结果。通常的做法是在插入的垃圾代码之前添加一条无条件跳转指令。

总而言之,代码隐藏类规避技术通常会给静态分析带来困扰,有时甚至使其失效。垃圾代码插入技术会在静态分析的词法、语法分析阶段对分析工具产生干扰,越是对 DEX 完整性和规范性有要求的工具越有可能会运行失败。动态代码加载技术和代码自修复技术会使静态分析生成不完整或错误的图,从而导致分析结果不准确。较为高级的商用代码保护方案会将原始安装包中所有的 DEX 字节码加密起来,而解密、加载和修复的逻辑则完全放在了使用混淆技术的本地代码中。在这种情况下,静态分析由于没有实际运行程序,不可能接触到被保护的代码,从而完全无法得知程序的原始行为。

1.3　相关工作

在安卓平台上,对规避技术的研究呈现方兴未艾之势。Strazzere 在文献[5]和文献[6]中介绍了对抗静态分析和动态分析的代码保护技术。Vidas 等[7]总结了一系列检测安卓应用是否运行于虚拟环境的方法,即对抗模拟器或对抗虚拟机技术。

在代码隐藏类规避技术方面,Yu[8]和 Strazzere[9]等对商用代码保护方案的特征做了一些假设,并以这些假设为基础来定位 DEX 字节码。然而随着代码保护方案的发展,这些方法已经失效。DexHunter[10]和 AppSpear[11]分别针对商用代码保护方案提出了提取隐藏代码方法。DexHunter 通过主动触发类加载和初始化进而对代码进行内存转储来提取被隐藏的代码。这种提取隐藏代码的方式只针对特定几种商用代码保护方案,无法全面有效地定位隐藏代码,方法不够通用。AppSpear 修改了 Dalvik 虚拟机的解释器,将解释执行到的相关代码转储到文件中,这种方法存在代码覆盖率不够全的问题。Harvester[12]提出了通过在 Dalvik 虚拟机中对字节码做切片运行来提取应用运行时信息的自动化分析方法,该方法可以在一定程度上对抗规避技术,但对本地代码中的规避行为效果不明显。

本文所提出的框架目的在于对安卓应用中的隐藏代码进行检测,可以准确全面地定位有效代码,具有较强的通用性,并且能应用于自动化分析。

2　恶意样本的人工分析

为了研究代码隐藏类规避技术的使用模式,并提出针对隐藏代码的检测框架,我们人工分析了一批恶意样本。样本的收集时间为 2015 年 08 月 01 日至 2015 年 08 月 03 日,共计 10 827 个。经由第三方检测引擎标注,这些样本可以划分到 107 个家族,170 个变种中去。考虑到同一变种的样本非常相似,我们在每个变种中随机选取了 1 个样本进行人工分析。

在 170 个样本中:有 7 个样本无法安装,这包括 6 个没有有效自签名证书的样本和 1 个为无效 APK 文件的样本;有 15 个样本没有 Activity 或者 Service 等程序入口,无法启动运行,以插件的形式存在;有 6 个样

本由于兼容性或其他问题,在程序启动后崩溃或退出。这 28 个样本均为无效样本,所以没有对其做进一步分析。对剩下的 142 个样本的分析结果如表 2 所示。

表 2 恶意样本的人工分析

分析结果	样本个数	占比
动态代码加载技术	42	29.6%
代码自修复技术	2	1.4%
垃圾代码插入技术	6	4.2%
未使用代码隐藏类规避技术	94	66.2%

在剩下的 142 个样本中,经人工分析,我们发现有 42 个样本使用了动态代码加载技术,有 2 个样本使用了代码自修复技术,有 6 个样本使用了垃圾代码插入技术。同时,使用代码自修复技术的 2 个样本也使用了垃圾代码插入技术。剩下的 94 个样本未发现使用本文所述的代码隐藏类规避技术。我们对这 142 个样本进行了标注,以用来在实验环节验证隐藏代码检测框架的准确性。

3 检测框架

我们对 142 个样本进行了人工分析,总结出一种代码隐藏类规避技术的检测方法,并实现了一个通用的自动化检测框架。其架构如图 3 所示,整个框架处理流程包括预处理、动态分析、后端处理三个阶段。框架的输入为 APK 样本和日志,同时日志也是框架的产出结果。

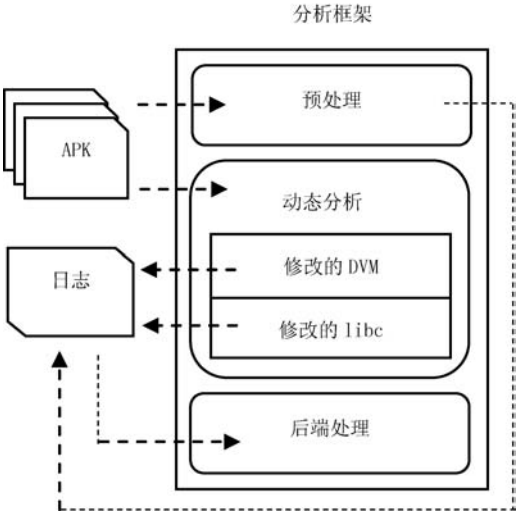

图 3 检测框架架构

3.1 预处理

在分析样本之前,必须能在日志中唯一地确定样本。我们把 APK 文件作为分析样本,使用文件的 MD5 和 SHA1 值来唯一确定样本文件。在安卓应用中,包(package)名和版本可以唯一确定其发布版本,因此我们也记录 APK 文件的包名、版本号、版本名称。

当对样本进行动态分析时,需要将其安装到安卓模拟器中并启动它的 Main Activity 使其运行。因此,我们还需要提取样本的 Main Activity 名称。除此以外,使用同一代码保护方案的样本往往具有统一的 Application 名称。因此,我们将这一字段记录下来,以便于对日志做进一步分析。

对于与安卓应用相关的信息,使用安卓 SDK 中的 aapt 工具进行提取。工具 aapt 的全名为 Android Asset Packaging Tool,即安卓资源打包工具,在 SDK 的 build-tools 目录下。该工具可以查看,创建,更新 ZIP 格式的文档附件(zip, jar,apk)。这里使用 aapt 来提取样本中的信息是因为 aapt 工具的代码实现与安卓系统相同,因而能够成功安装并运行的应用都可以用 aapt 工具进行分析,aapt 工具具有良好的兼容性。

在提取了安卓应用的相关信息后,还需要将样本 APK 文件进行解压缩,以用来在后端处理阶段对安装包中的文件进行处理。

3.2 动态分析

我们修改了安卓系统源码中 Dalvik 虚拟机和 libc 部分的代码,编译成安卓模拟器作为动态分析的环境。考虑到 APK 版本的兼容性,我们使用了较低版本的安卓系统源码,其版本号为 4.4.4。为了保证样本之间互不影响,每一次启动的安卓模拟器镜像都是预先配置好的镜像的一份拷贝。我们修改的 Dalvik 虚拟机和 libc 会在分析时记录下样本的运行时信息,并将其写入到日志中。

3.2.1 主动触发类加载和初始化

动态分析可以分为以符号执行为代表的白盒测试和基于真实分析环境和虚拟分析环境的黑盒测试。一般来说,动态分析都期望被分析的程序尽可能多地执行代码路径来保证分析有足够的代码覆盖率。在基于虚拟机环境的自动化分析环境中,一种较为流行的保证代码覆盖率的方法是以随机模拟事件的方式来驱动代码的执行,如 AppsPlayground[13]。安卓 SDK 也提供了名为 MonkeyRunner[14] 的类似的自动化测试工具。

然而,在分析恶意样本时我们发现,有大量样本在启动之后会直接退出用户界面并驻留在后台运行。在这种情况下,如果采用随机模拟事件的方式来驱动程序的执行,可能会启动系统内的其他应用,甚至导致内

存不足,Activity Manager 杀死当前需要分析的进程。与一般动态分析中行为分析不同的是,分析框架的目的是尽可能地触发进程中代码的加载和修复,而不是执行完所有的代码路径,我们使用了主动触发类加载和初始化的方法来保证分析有足够的代码覆盖率,这一方法由 DexHunter[10] 提出,我们做了部分改进。

我们的方法是:监控 Dalvik 虚拟机中加载 DEX 字节码的过程,每当有一段 DEX 字节码被加载时就启动一个新的线程。图 11 展示了该线程接下来的执行流程。新创建的线程会先睡眠一段时间,然后根据 DEX 文件的索引结构查找该 DEX 文件中所有类的名字,用类名作为参数调用相关 API 来触发 Dalvik 虚拟机对类的加载。这里启动一个新的线程并睡眠一段时间是为了保证程序不会运行崩溃。因为当类加载的顺序与预先设定的执行顺序不一致时,类的部分属性会初始化为与预期不同的值,从而使主线程抛出未捕获的异常,导致进程崩溃。而当起动一个新的线程并睡眠一段时间后,主线程的初始化工作已经完成,这时新启动的线程就可以通过强制捕获所有抛出异常的方式来保证类的加载,并不会对主线程的执行造成影响。

基于以下结论,该方法能够起到良好的效果:1) Dalvik 虚拟机对代码的加载是以类为基本单位的,当虚拟机加载一个类的时候,会验证该类所有方法的代码,并将该类进行初始化,也就是创建一个类对象;2) 对隐藏代码的加载和修复往往会出现在程序的入口和类的初始化方法 < clinit > ( ) 中。第 2 条结论由人工分析的恶意样本得出。出现这种模式与 Java 语言要求的类在初始化前必须先加载并初始化其依赖的类的机制有关。这是因为隐藏代码的加载和修复的行为如果不出现在 < clinit > ( ) 方法中,那么就必然出现在非静态方法中。这就要求对代码进行复杂的调用依赖分析,否则虚拟机可能会抛出 ClassNotFoundException 异常。而复杂的调用依赖分析会在代码保护的离线处理中占用非常多的资源,对程序的性能也会造成影响。

3.2.2　修改的 DVM

对 Dalvik 虚拟机的修改主要用来记录进程中类对象的加载过程,以用来判断样本是否使用了动态代码加载技术。

对于动态代码加载技术,检测方法的基本思想是:Dalvik 虚拟机是一个解释执行模型,其解释器引用到的 DEX 字节码一定是有效代码。为了详细地阐述这一思想,首先需要解释一下 Dalvik 虚拟机中类对象和 DEX 文件在内存中的关联关系,如图 4 所示。

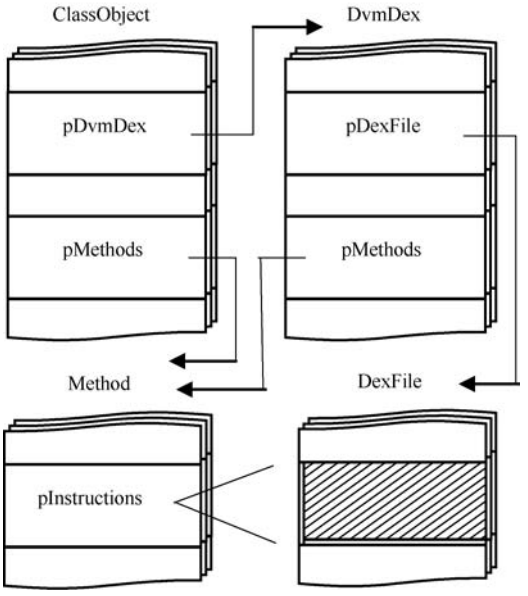

图 4　Dalvik 虚拟机中类对象和 DEX 文件间的关联

Dalvik 虚拟机中维护着一张表,表里保存着当前所有已加载的类对象,即 ClassObject。ClassObject 对应于 java. lang. Class 的对象,每个类只有一个,只有当类被加载时才会创建。ClassObject 中保存着该类所有方法的指针。DexFile 是 DEX 文件在内存中的映射。由于 DEX 文件是索引结构,不能直接维护所有的类对象和方法对象,虚拟机会为每一个 DexFile 维护一个名为 DvmDex 的对象,ClassObject 对其对应的 DexFile 的引用是间接的,即通过指向一个 DvmDex 对象来寻找其对应的 DexFile 对象。

Dalvik 虚拟机维护这样一种对象关系与其代码加载机制有关,其对代码的加载遵循 Java 虚拟机中类加载的默认委派机制。即启动类加载器由虚拟机的本地代码实现,负责对 Java 核心类的加载。扩展类加载器同样由虚拟机实现,负责对虚拟机代码库中非核心代码的加载。应用程序类加载器是虚拟机提供给应用的默认类加载器,负责加载开发者实现的代码。除此之外,开发者还可以自行实现自定义类加载器,来加载私有的代码。一个类的加载是从下到上以委派的模式加载的。即虚拟机先调用与当前类关联的类加载器,查找所需要加载类的代码,如果找不到则向上一级委派。上一级类加载器重复这一过程,直到执行到启动类加载器。如果启动类加载器也无法加载类的代码,就会抛出 ClassNotFoundException。在 Dalvik 虚拟机中,一个类加载器往往关联着几个 DEX 文件。

在 Dalvik 虚拟机中,当有一个 DEX 文件被加载时,就会创建一个 DexFile 对象来维护 DEX 文件的索引结构。然后,虚拟机会为该 DexFile 对象创建一个 DvmDex 对象以用来维护类、方法等内存中的对象。

当虚拟机解释执行时,它会通过全局表来来查找方法所在的类,即 ClassObject。如果类不存在,则通过委派加载机制依次查找到途经类加载器所关联的 DexFile 对象,直至找到所需的类。当把该类加载和初始化后,虚拟机通过 ClassObject 所引用的 Method 找到对应的字节码,并开始解释执行。

如果开发者想要对代码进行保护,那么他们可以实现私有的类加载器,或者通过硬编码等方式调用 Dalvik 虚拟机的代码,从而绕过 Dalvik 虚拟机所提供的调用接口。更高级的规避方法是将代码分段,在进程执行的过程中修改 Dalvik 虚拟机对象的引用指针,如将 ClassObject 的 pDvmDex 改为空指针,或将 Method 的 pInstructions 指向一块堆内的地址,这时 ClassObject 与 DvmDex 所指向的 Method 对象必然不一致,从而对分析人员造成困扰。

然而,Dalvik 虚拟机是一种解释执行模型,其解释器引用到的代码一定是有效代码,否则将导致程序行为异常。在解释器运行的过程中,总是通过 Dalvik 虚拟机所维护的已加载类表来寻找对应的 ClassObject,通过解引用其指向的方法代码来解释执行。由于进程的执行流程是无法预判的,所以可以认为 ClassObject 所引用的代码是进程加载和恢复的原始代码。我们将所有的 ClassObject 和与之相关的对象记录到日志中,在后端处理阶段对其进行分析。

3.2.3 修改的 libc

通过对使用了代码自修复技术的恶意样本进行人工分析,我们发现了代码自修复技术的使用模式:Dalvik 虚拟机是一种解释执行模型,其解释执行的 DEX 文件映射在内存中的页属性是只读的。进程为了修改 DEX 字节码,会调用系统调用中的 mprotect( ) 函数来修改 DEX 字节码在内存中的页保护属性,然后通过内存拷贝的方式来修改 DEX 字节码,最后调用再 mprotect( ) 函数将其内存页保护属性改为原值。

为了分析样本中代码自修复技术的使用情况,我们在分析框架中对 libc 中 mprotect( ) 函数的调用情况进行了监控,每当进程调用 mprotect( ) 函数修改内存页为可写时,分析框架就将其调用参数的起始地址和终止地址记录到日志中。同时,当进程执行一段时间后,分析框架将进程的内存布局从/proc/self/maps 文件输出到日志中。/proc/self/maps 的内存布局如下所示:

4003a000-40049000 r-xp 00000000 b3:15 138 /system/bin/linker

40049000-4004a000 r-p 0000e000 b3:15 138 /system/bin/linker

4004a000-4004b000 rw-p 0000f000 b3:15 138 /system/bin/linker

...

40055000-40075000 r-xp 00000000 b3:15 749 /system/lib/libc.so

40075000-40077000 rwxp 00020000 b3:15 749 /system/lib/libc.so

40077000-4009c000 r-xp 00022000 b3:15 749 /system/lib/libc.so

...

41555000-41883000 rw-p 00000000 00:04 6911 /dev/ashmem/dalvik-zygote (deleted)

...

6eae7000-6ec54000 r-p 00000000 b3:17 185112 /data/dalvik-cache/system@framework.jar

...

747d7000-747d8000 r-s 00009000 b3:17 48292 /data/app/com.example-1.apk@classes.dex

...

如 app_process、linker 以及 libc.so 等可执行文件在内存中会映射为三段,分别具有可执行、只读、读写的属性,对应于.text、.rodata、.data/.bss 字段。形如/dev/ashmem/xxxxxx (deleted)的是匿名共享内存 Ashmem,经由 Binder 进行进程间通信后将内存销毁。形如 system@framework.jar 为系统提供的 Java 代码库。/data/app/com.example-1.apk@classes.dex 为第三方应用的 DEX 文件在内存中的映射,其属性为只读。我们需要监控的即是对/data/app/com.example-1.apk@classes.dex 所要进行的修改。

3.3 后端处理

后端处理以动态分析输出的日志为输入,将代码隐藏类规避技术的检测结果作为输出写到日志中。

为了判断样本是否使用了动态代码加载技术,一种取巧的做法是遍历日志中记录的所有的 ClassObject,并统计其中不同的 DvmDex 的个数。如果 DvmDex 的个数超过 1 个,即使其值为空,也仍然使用了动态代码加载技术。此外,我们还在日志中记录了 pDvmDex 运行时的值,有效代码是否在 ClassObject 间接引用的 DexFile 内存块中等运行时数据来进行验证。

通过比较日志中调用 mprotect( ) 函数修改的内存与内存布局中 DEX 文件映射区域是否有重合,即可得知进程在运行的过程中是否有修改 DEX 字节码的意图,从而可以判断样本是否使用了代码自修复技术。

为了判断样本是否使用了垃圾代码插入技术,最

直观的方法是使用反汇编工具直接对 APK 文件的 DEX 字节码进行处理,通过分析工具输出的日志来判断样本的代码是否合乎规范。在这里,我们使用了 baksmali[15] 作为分析框架的反汇编的工具,版本号是 2.1.2。当 baksmali 在处理 DEX 文件时,如果发现文件不符合规范或者代码指令非法,即会输出错误信息。

在表 3 中列出了经过后端处理阶段后,对一个样本的分析结果的所有字段。不符合规范或者代码指令非法,即会输出错误信息。

表 3　检测结果的所有字段

字段	描述
MD5	APK 安装包文件的 md5,与 sha1 结合来唯一标识样本文件
SHA1	APK 安装包文件的 sha1,与 md5 结合来唯一标识样本文件
Package Name	安卓应用的包名,用来唯一标识该应用的名称
Version Code	安卓应用的版本号,用来维护版本升级,即当版本号高于当前版本且签名一致时,对当前版本作替换
Version Name	安卓应用的版本名称,对应于版本号
Application Name	安卓应用中 android. app. Application 或其子类的名称,每个应用只有一个。该字段可以用来辅助分析样本是否使用了商用的代码保护方案
Main Activity	安卓应用的 Launchable Activity,即显示在抽屉桌面可以由用户点击启动的 Activity,为程序分析的主要入口之一
Analysis Start Time	动态分析的开始时间
Analysis End Time	动态分析的结束时间
Analysis Status	动态分析的最终状态,可以是:(1) 分析成功;(2) 安装失败;(3) 应用无法启动;(4) 进程自动退出或崩溃;(5) 分析超时
Dynamic Code Loading	样本是否使用了动态代码加载技术
Self-Modifying Code	样本是否使用了代码自修复技术
Junk Code Insertion	样本是否使用了垃圾代码插入技术

在动态分析的过程中,应用可能会在运行很长一段时间后仍然没有记录足够的数据。比如样本需要联网下载文件,但是分析时的网络状况不是很好。这时,应将当前正在运行的安卓模拟器关闭,并将分析任务加入到任务队列中等待重新分配。因此我们记录了动态分析的起始时间和终止时间。除此以外,我们还记录了样本分析的最终状态,这些状态可能是:分析成功、安装失败、应用无法启动、进程自动退出或崩溃,分析超时。导致安装失败的原因可能是 APK 文件不符合规范,没有有效的自签名证书,或者工具对样本的信息提取不够完整。应用无法启动的原因可能是应用没有 Main Activity 或者分析工具无法提取出样本的 Main Activity 名称。除此之外,我们还会以轮询的方式查看进程是否在稳定运行,如果不是,则程序有可能已经退出或者运行崩溃。

4　实验评估

为了评估分析方法的准确性,我们收集了两组实验样本。

第一组样本为第三节所述的人工标注的 142 个恶意样本,收集时间为 2015 年 08 月 01 日至 08 月 03 日。

第二组样本共计 2 278 个,于 2016 年 05 月 12 日在中国第三方应用市场应用宝中收集。我们在应用宝官方划分的购物、阅读、新闻、视频、旅游、工具、社交、音乐、美化、摄影、理财、系统、生活、出行、安全、教育、健康、娱乐、儿童、办公和通信 21 个分类中,将所有的推荐应用进行了抓取。这些应用在官方网站中按照下载量、打分等因素排序,是该应用市场中最流行的应用。考虑到游戏应用对安卓模拟器的兼容性较差,我们未对游戏类应用进行抓取。

我们的实验平台为 64 位 Linux 系统,内核版本 3.16.0。处理器为 40 核,CPU 频率 2.2 Hz,物理内存大小 128 GB。动态分析采用的安卓系统源码版本为 4.4.4。

4.1　标注样本的自动化分析

我们将检测框架应用于人工标注的 142 个样本中,以评估检测方法的准确性。对人工标注样本的自动化分析结果如表 4 所示。

表 4　人工标注样本的自动化分析结果

分析结果	样本个数	检出个数	占比
动态代码加载技术	42	36	85.7%
代码自修复技术	2	2	100%
垃圾代码插入技术	6	6	100%

实验结果显示,在 42 个标注为使用了动态代码加载技术的样本中,检测框架识别出了 36 个,使用了代码自修复技术的 2 个样本和垃圾代码插入技术的 6 个样本全部能识别出来。在未识别的 6 个样本中,有 1 个样本需要用户登录才能触发对隐藏代码的加载,其他 5 个样本会在启动阶段判断是否在安卓模拟器中运行,如果是,则立即退出。这 5 个样本有对抗模拟器的

意图。

4.2 第三方市场应用的自动化分析

我们对 2 278 个第三方市场应用样本进行了自动化分析,其结果如表 5 所示。

表 5 第三方市场中样本的自动化分析结果

分析结果	样本个数	占比
无法安装	4	0.2%
无法直接运行	25	1.1%
启动后崩溃或退出	2	0.1%
动态代码加载技术	773	33.9%
代码自修复技术	0	0.0%
垃圾代码插入技术	23	1.0%
未检出使用代码隐藏类规避技术	1 453	63.8%

在 2 278 个样本中:有 4 个样本无法安装到安卓模拟器中,经分析发现这 4 个样本依赖于 Google Maps 框架,而我们的定制版安卓模拟器中没有 Google 服务框架;有 25 个样本使用了较新的安卓 SDK,我们的工具在预处理阶段无法提取其完整信息,动态分析不能继续进行;有 2 个样本由于兼容性问题在进程启动后崩溃或退出。

在剩下的 2 247 个样本中:有 773 个样本使用了动态代码加载技术;有 0 个样本使用了代码自修复技术;有 23 个样本使用了垃圾代码插入技术。其中,有 2 个样本同时使用了动态代码加载和垃圾代码插入技术,剩下的 1 453 个样本未发现使用本文所述的三种规避技术。

在 2 247 个样本中未发现任何一个样本使用了代码自修复技术是出人意料的。在 Going Native[16] 中指出,调用 mprotect( )对 DEX 代码做修改的样本无一例外地使用了 apkprotect 这一早期代码保护工具。我们对 2 247 个样本中的文件进行了静态扫描,发现确实没有任何一个样本使用了 apkprotect 工具,一个可能的原因是该工具已经不再对外提供服务了。判定样本是否使用了 apkprotect 工具可以基于以下特征:1) 在 APK 文件中有 libapkprotect. so 这一本地代码文件;2) DEX 文件中有 apkprotect 这样一个类,该类在初始化阶段会加载 libapkprotect. so。

如表 6 所示,我们在 773 个检测为使用了动态代码加载技术的样本中随机选取了 50 个样本进行人工验证,发现这 50 个样本确实全部使用了动态代码加载技术。检测为使用了垃圾代码插入技术的 23 个样本全部进行了人工验证,未发现有错误的检测结果。在 1 453 个未检出使用了代码隐藏类规避技术的样本中,选取了 50 个样本进行人工验证,发现有 1 个样本由于安卓模拟器中地理位置信息不符合实际情况未触发动态代码加载的过程。

表 6 第三方市场样本分析结果的人工验证

规避技术	阳性样本数	假阳率	阴性样本数	假阴率
动态代码加载技术	50	0%	50	2.0%
代码自修复技术	0	0%	0	0.0%
垃圾代码插入技术	23	0%	50	0.0%

动态代码加载技术在推荐应用中有 33.9% 的普及率,略高于人工分析的恶意样本中 29.6% 的比率。虽然两者的良恶性质、抓取时间皆不相同,但是比率较为接近,说明了动态代码加载技术在良性、恶性样本中都有着显著的普及率。

通过对使用了动态代码加载技术的样本进行日志分析,我们发现有 287 个样本的 Dalvik 虚拟机内存对象引用不正确,在使用动态代码加载技术的样本中比例达到了 37.1%。这说明样本有隐藏其规避行为的意图,其动态代码加载的过程具有隐蔽性。需要指出的是,这里所列的数据是真实情况的下界。

相反地,代码自修复技术粒度粗、隐蔽性弱,垃圾代码插入技术只能稍微提高安全分析的门槛,相较于动态代码加载技术,其使用比例较低。这体现了规避技术的发展是一个对抗的过程,其在技术实现上越来越复杂且隐蔽。

5 结 语

通过对 142 个恶意样本进行人工分析,本文提出了一个通用的自动化检测框架,可用于识别样本是否使用了代码隐藏类规避技术。该检测框架将代码隐藏类规避技术的模式统一起来,利用安卓系统中 Dalvik 虚拟机的代码加载和执行模型,对样本加载和修改代码的行为进行监控,可以全面有效地定位隐藏代码,可应用于自动化分析。实验表明,该检测框架有着较高的准确率。

参 考 文 献

[1] Smartphone OS market share[OL]. 2016. http://www.idc.com/promo/smartphone-market-share/os.

[2] Android statistics: number of android applications[OL]. 2016. https://www.appbrain.com/stats/number-of-android-apps.

[3] Myapp mobile application market[OL]. 2016. http://sj.qq.com.

[4] Dalvik Execuatble Format[OL]. 2016. https://source.android.com/devices/tech/dalvik/dex-format.html.

[5] Strazzere T. Dex education: Practicing safe dex[M]. Black Hat, USA, 2012.

[6] Strazzere T. Dex education 201: anti-emulation[M]. HITCON, 2013.

[7] Vidas T, Christin N. Evading android runtime analysis via sandbox detection[C]//Proceedings of the 9th ACM symposium on Information, computer and communications security. ACM, 2014:447 - 458.

[8] Rowland Yu. Android Packers: Facing the challenges, Building solutions[C]//Proc. of the 24th Virus Bulletin International Conference, 2014.

[9] Strazzere T, Sawyer J. Android Hacker Protection Level 0 DEF CON 22[Z]. 2014.

[10] Zhang Y, Luo X, Yin H. Dexhunter: toward extracting hidden code from packed android applications[M]//European Symposium on Research in Computer Security. Springer International Publishing, 2015:293 - 311.

[11] Yang W B, Zhang Y Y, Li J R, et al. AppSpear: Bytecode Decrypting and DEX Reassembling for Packed Android Malware[M]//Research in Attacks, Intrusions, and Defenses. Springer International Publishing, 2015.

[12] Rasthofer S, Arzt S, Miltenberger M, et al. Harvesting runtime values in android applications that feature anti-analysis techniques[C]//Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS). 2016.

[13] Rastogi V, Chen Y, Enck W. AppsPlayground: automatic security analysis of smartphone applications[C]//Proceedings of the third ACM conference on Data and application security and privacy. ACM, 2013:209 - 220.

[14] Monkey Runner[OL]. 2016. https://developer.android.com/studio/test/monkeyrunner/index.html.

[15] Smali and baksmali: Dalvik bytecode toolkit[OL]. 2016. https://github.com/JesusFreke/smali.

[16] Afonso V, Bianchi A, Fratantonio Y, et al. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy[OL]//Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS), 2016.

(上接第 269 页)

参 考 文 献

[1] Wagner D, Schneier B. Analysis of the ssl 3[J]. Proceedings of the Second Unix Workshop on Electronic Commerce, 1996, 28(28):29 - 40.

[2] Giesen F, Kohlar F, Stebila D. On the security of TLS renegotiation[C]//Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM, 2013: 387 - 398.

[3] Fardan N J A, Paterson K G. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols[C]// IEEE Symposium on Security and Privacy. IEEE Computer Society, 2013: 526 - 540.

[4] Möller B, Duong T, Kotowicz K. This POODLE bites: exploiting the SSL 3.0 fallback[OL]. 2014. https://www.openssl.org/~bodo/ssl-poodle.pdf.

[5] Duong T, Rizzo J. Here Come The ⊕ Ninjas[J]. Unpublished Manuscript, 2011.

[6] 强小辉, 陈波, 陈国凯,等. OpenSSLHeartBleed 漏洞分析及检测技术研究[J]. 计算机工程与应用, 2016, 52(9): 88 - 95.

[7] Brumley D, Dan B. Remote timing attacks are practical[J]. Computer Networks, 2005, 48(5):701 - 716.

[8] Georgiev M, Iyengar S, Jana S, et al. The most dangerous code in the world: validating SSL certificates in non-browser software[C]// Proceedings of the 2012 ACM conference on Computer and communications security. ACM, 2012: 38 - 49.

[9] Freier A, Karlton P, Kocher P. The SSL 3.0 Protocol[Z]. November 1996.

[10] Allen C, Dierks T. The TLS protocol version 1.0 [Z]. 1999.

[11] Dierks T, Rescorla E. The Transport Layer Security(TLS) Protocol Version 1.1[R]. RFC 4346 , DOI 10.17487/RFC4346 , April 2006.

[12] Dierks T, Rescorla E. The Transport Layer Security(TLS) Protocol Version 1.2[R]. RFC 5246 , DOI 10.17487/RFC5246 , August 2008.

[13] Rescorla E. The Transport Layer Security (TLS) Protocol Version 1.3[R]. draft-ietf-tls-tls13 - 18, 2016.

[14] Salowey J. Transport Layer Security (TLS) Session Resumption without Server-Side State[Z]. Transport, 2006.

[15] Clark J, Oorschot P C V. SoK: SSL and HTTPS: Revisiting Past Challenges and Evaluating Certificate Trust Model Enhancements[C]// Security and Privacy. IEEE, 2013:511 - 525.

[16] Bhargavan K, Fournet C, Kohlweiss M, et al. Proving the TLS Handshake Secure (as it is)[J]. Lecture Notes in Computer Science, 2014, 8617:235 - 255.

[17] Dowling B, Fischlin M, Nther F, et al. A Cryptographic Analysis of the TLS 1.3 Handshake Protocol Candidates[C]// The, ACM Sigsac Conference. ACM, 2015:1197 - 1210.