一种多层次的自动化通用 Android 脱壳系统及其应用

简容[1,2]， 黎桐辛[2]， 周渊[3]， 李舟军[1]， 韩心慧[2]
（1. 北京航空航天大学 计算机学院，北京 100191；2. 北京大学 计算机科学技术研究所，
北京 100080；3. 国家计算机网络应急技术处理协调中心，北京 100085）

摘 要：为解决 Android 平台应用程序使用加壳服务后难以进行静态代码分析的问题，研究应用程序自动化通用脱壳技术. 在 Android Dalvik 虚拟机的基础上，设计并实现了一种多层次的自动化通用脱壳系统，提出了多粒度的数据还原方案，保证数据还原的完整性和有效性，能正确还原出加壳应用中被加密的代码内容. 实验表明，该系统适用于市面上主流的加壳服务. 利用该系统，对市场上被加壳的应用程序进行安全性评估，发现加壳应用比未加壳应用存在更多的安全问题，证明了脱壳系统的实际应用价值.

关键词：Android 平台；加壳；脱壳；安全评估
中图分类号：TP309 **文献标志码**：A **文章编号**：1001-0645(2019)07-0725-07
DOI：10.15918/j.tbit1001-0645.2019.07.011

Design and Application of a Multi-Level Automated Universal Unpack System on Android Platform

JIAN Rong[1,2]， LI Tong-xin[2]， ZHOU Yuan[3]， LI Zhou-jun[1]， HAN Xin-hui[2]
（1. School of Computer Science, Beihang University, Beijing 100191, China;
2. Institute of Computer Science and Technology, Peking University, Beijing 100080, China;
3. National Internet Emergency Center, Beijing 100085, China）

Abstract: In order to solve the problem that the Android platform application is difficult to perform static code analysis after using the packer service, an automated universal unpack technology was presented. Firstly, a multi-level automated universal unpack system was designed and implemented based on the Android Dalvik virtual machine. And then, a multi-granularity data restoration scheme was proposed to ensure the integrity and validity of data restoration, and to correctly restore the encrypted code content in the packed application. Experiment results show that this system is suitable for mainstream pack services in the market. Evaluating the security of the packed application with this system in the market, more security issues are found in the packed application than in the normal application, which proves the application value of the unpack system.

Key words: Android platform; packer; unpacker; security evaluation

目前 Android 操作系统已经成为全球市场上所占份额最高的移动终端系统，基于 Android 操作系

收稿日期：2018-11-08
基金项目：国家自然科学基金资助项目(61602025，U1636211，61672081)；国家重点研发计划项目(2016QY04W0802)；北京市成像技术高精尖创新中心项目(BAICIT-2016001)
作者简介：简容(1995—)，男，硕士生，E-mail：jianr1@outlook.com.
通信作者：李舟军(1963—)，男，教授，博士生导师，E-mail：lizj@buaa.edu.cn.

统的应用程序数量也逐年增多. 由于 Android 平台的开放性,应用程序能够较容易的被逆向分析、修改破解、重新打包[1]. 为了防止软件被逆向破解或攻击利用,许多开发者采取了应用加壳的方式[2],对程序关键代码进行加密、隐藏,极大地增加了逆向分析的难度,从而达到对程序保护的效果.

但是,以 Android 平台为目标的恶意软件也利用了加壳技术的特性,对自身携带的恶意代码进行隐藏,用于躲避杀毒引擎的检测和安全研究人员的分析[3-4]. 同时,加壳后的应用也无法利用静态分析工具准确检测应用内部的安全隐患[5]. 因此,实现自动化通用脱壳,在程序分析和恶意软件检测等方面有重要意义.

本文设计并实现了一种多层次的自动化通用脱壳系统,能够应对目前市场上主流加壳服务,还原程序的代码逻辑. 提出了多粒度的数据还原方案,保证数据还原的完整性和有效性. 对应用市场上的加壳程序进行安全性评估,得到了新的发现,证明了脱壳系统的实际应用价值.

1 脱壳系统的设计与实现

本文基于 Dalvik 虚拟机设计并实现了一种多层次自动化通用脱壳系统. 该系统将应用程序的启动执行过程划分为不同的 3 个主要层次,在不同层次对加壳应用的 dex 文件进行不同粒度的数据转储(Dump),并最终对 dex 文件的头部信息,索引结构区,数据区进行还原,生成完整 dex 文件.

图 1 展示了脱壳系统的整体架构. 它包含了多层次状态监控,多粒度数据转储和 Dex 文件重构 3 个模块. 由于目前的加壳技术对 dex 文件的解密不再是一个单一阶段的过程,而是随着 dex 文件的加载与执行逐步解密,因此,在不同时刻内存中的 dex 文件所具备的数据也有所不同. 为此,在加壳程序运行的每一个层次设置监控点,跟踪加壳程序的执行过程. 依照数据类型的不同,对 dex 文件的数据进行不同粒度的区分,并根据监控点的反馈信息,获取当前运行层次下 dex 文件对应的数据内容中的有效部分. 最后,对获取到的有效数据进行 dex 文件重构,实现自动化的脱壳流程.

该系统以加壳后的应用程序作为输入,通过动态执行加壳程序,输出脱壳后包含完整代码的 dex 文件,在脱壳过程中,不需要使用者对加壳服务有逆向经验或对程序进行额外的人工预处理,同时,由于该系统直接基于 Dalvik 虚拟机实现,不会受到加壳服务

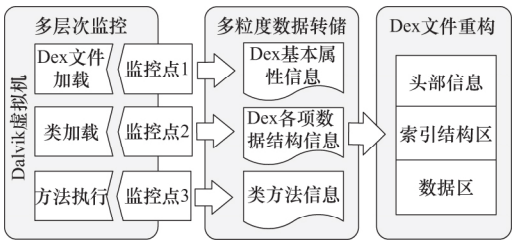

图 1 脱壳系统整体框架

Fig. 1 Architecture of the unpacker system

完整性检查和反调试措施的影响,自动化程度较高.

1.1 多层次状态监控

为了监控壳代码的行为,本文设计了多层次的状态监控方案,该方案不需要对虚拟机整体运行做跟踪,而是在程序执行的不同层次,选取尽可能少的监控点,用于反映壳代码在执行过程中的关键行为.

1.1.1 层次划分

脱壳系统将程序的启动执行划分为以下 3 个层次.

① Dex 文件加载:Dalvik 虚拟机可以通过文件,或二进制字节流的形式,将 dex 文件加载至内存,并生成相应的 DexFile 结构来代表被加载的 dex 文件.

② 类加载:Dalvik 虚拟机中使用 ClassObject 结构代表已加载的类. 根据类加载方式的不同,可以分为显式加载和隐式加载.

③ 方法执行:当类中的方法被调用时,Dalvik 虚拟机将从类加载信息中生成对应的 DexMethod 结构来代表该方法. 方法执行的过程,即为虚拟机对字节码解释执行的过程.

1.1.2 监控点的选取

根据上述层次划分,在每一层次选取监控点,用于获取在该层次下壳代码的行为信息. 在选取监控点时,应当考虑到:选取的监控点数量应该尽量少,以减小系统整体的性能开销;选取的监控点应覆盖当前层次中所有的路径调用情况;选取的监控点不能是导出函数或系统提供的 API,以防被加壳程序检测或修改. 根据上述规则,对每一层次的监控点选取如下.

① 监控 dex 文件加载:为避免 dex 文件的重复加载,Dalvik 虚拟机将已加载的 dex 文件保存在全局变量 gDvm. userDexFiles 中. 选取了 addToDexFileTable 函数作为 dex 文件加载的监控点. 该函数是将 dex 文件增加至 gDvm. userDexFiles 中的唯一途径,因此通过监控该函数,能够得到壳代码加载

dex 文件的信息.

② 监控类加载：当一个类从未被加载过时，最终将由 native 层的 Dalvik_dalvik_system_DexFile_defineClassNative 函数完成从 dex 文件加载指定类的任务. 选取该函数作为类加载行为的监控点.

③ 监控类方法执行：当应用程序启动时，Dalvik 虚拟机解释器以 android.app.ActivityThread 类的静态成员函数 main 为入口点执行，在执行过程中遇到函数调用时，会因函数属于 Java 层还是 Native 层而出现 4 种不同的情况. 表 1 展示了这 4 种情况下虚拟机所使用的跳转方法. 可以看到，除去 Native 层到 Native 层的调用之外，都能够在每一个方法被调用前，监控到该方法的调用信息. 由于 Native 层的代码不属于 dex 文件中由 Java 编译而来的字节码，因此可以不用处理这一类情况.

表 1　Dalvik 虚拟机函数调用的 4 种类型

Tab. 1　Four types of function calls in Dalvik virtual machine

主调类型	被调类型	调用方式
Java	Java	GOTO_TARGET(invokeMethod…)
Java	Native	dvmCallJNIMethod()
Native	Native	汇编语句 call/jmp
Native	Java	dvmInvokeMethod(),dvmCallMethodV/A()

1.2　多粒度数据转储(DUMP)

在传统的脱壳方案中，往往会指定一个特定的脱壳时机(通常为加壳程序的第一个 Activity 被创建时)，当这个时机到达后，脱壳程序认为此时壳代码已将 dex 文件在内存中完全解密，并一次性转储 dex 文件，完成整个脱壳流程. 实际上，以这种方式得到的 dex 文件虽然能够被正常反编译，但通常会出现部分数据缺失或偏移错误的情况，这是因为基于特定时机后 dex 文件完全解密的假设本身可能是不准确的. 为了解决这个问题，使用了多粒度数据转储的方案，来保证内存数据的准确性.

Dex 文件由多项不同类型数据结构组成，且通过加载基址加上数据偏移的方式来计算数据存放位置，因此，通常需要根据偏移值进行多次索引，才能获取相应数据内容. 例如，存放于 DexHeader 的 stringIdsOff 代表了字符串类型数据结构 DexStringId 的起始偏移，通过 DexStringId 中 stringDataOff 成员的值，可以获得 MUTF-8 编码的实际字符串内容. 除字符串信息外，dex 文件还包含了类型信息、原型信息、字段信息、方法信息、类信息以及依赖信息等部分. 根据上述索引方式，可以在知道 dex 文件内存基址的情况下，解析所有数据所在内存地址，并获取内容. 然而，并非所有的内容都是有效的，在何时去获取数据内容，取决于壳代码逐级解密的时机，例如，只有当类中的方法被执行时，虚拟机才会去获取方法中的字节码，壳代码可以选择在此时解密出类方法的字节码.

从 1.2.1 节所述的层次划分来看，不同时刻对于 dex 文件中的数据获取的准确程度是不同的. 根据数据获取的粒度不同，将数据分为以下 3 类. ① dex文件的基本属性，如版本标识和内存映射长度；② dex 文件中各项数据结构以及具体数据中与类方法无关的部分；③ 类方法部分. 每一个类方法均由 DexMethod 结构表示，其中包含了可被虚拟机执行的字节码数据.

根据状态监控反馈的信息，该模块将动态的获取不同粒度下的数据内容. 多粒度数据转储基于这样一个准则，即数据转储发生在 Dalvik 虚拟机即将使用该数据之前，在数据转储到虚拟机正常使用数据的过程中，壳代码将没有机会对该数据进行动态修改.

1.2.1　Dex 文件基本属性获取

当监测到 dex 文件的加载行为时，将获取该 dex 文件的基本属性. 具体而言，通过监测的 addToDexFileTable 函数获得指向内存中 dex 文件的 pDexOrJar 指针，并利用该指针得到 dex 文件对应内存代表的 DexFile 结构. 此时 dex 文件中的各项数据结构对应的内容不一定是有效的，因此仅获取脱壳所需要的一些基本属性，包括：dex 文件在内存中的映射长度，dex 文件是否已被优化为 odex 格式，dex 文件的版本标识和字节序标记.

1.2.2　Dex 文件数据结构获取

当监测到类加载行为时，将获取 dex 文件各项数据结构的信息. 具体而言，通过监测的 Dalvik_dalvik_system_DexFile_defineClassNative 函数，获得当前加载类所使用的 ClassLoader 和当前类所在的 dex 文件信息，然后利用 ClassLoader 主动加载该 dex 文件中的所有类，使得壳代码在类中注入的静态代码块得到执行. 此时，对于 dex 文件中不属于类方法的数据，在类被加载至内存后将处于解密状态，获取当前 dex 文件每一类数据结构的个数和对应的偏移值，根据偏移值计算出数据的内存地址，并依据数据结构中的成员定义，获取数据的真实

内容. 需要注意的是,虽然此时 dex 文件中类方法相关数据不一定处于解密状态,但仍然会在此时根据相关偏移去解析和获取类方法数据.

1.2.3 类方法的动态更新

如 1.2.2 节所述,当完成类加载操作后,类方法数据可能仍处于加密状态,这是由于部分壳代码采取了方法替换的方式,将原有类方法替换为壳代码,并在类方法被调用时动态解密. 为了应对这种情况,在类方法调用时进行类方法的动态更新.

当监测到方法调用行为时,通过函数局部变量 Method,获得当前被调用方法的函数签名,所属类以及其指令集,如果该方法所属类位于已加载的非系统 dex 文件集合中,则获取该方法包含的所有数据. 由于在程序运行期间,并非所有的方法均有机会得到执行,可以使用 Monkey、UI Automator 等动态测试工具,发送随机事件,如键盘输入、屏幕点击、手势滑动等,提高动态代码覆盖率.

1.3 Dex 文件重构

重构 dex,指的是将多粒度数据转储获得的内存数据重新汇编成可供静态分析工具分析的完整 dex 文件,并写入外部存储设备. 进行 dex 重构时,通过广度优先遍历的方式,依照 dex 文件标准规范,以 dex 文件头部为根节点,重构 dex 文件中的各项数据结构. 由于 dex 文件重构的数据完全来源于数据转储的内容,因此 dex 文件重构的时机尤为重要,如果在脱壳时仅进行单一阶段的重构,或者在壳代码对原有数据进行解密之前就进行重构操作,都将导致还原的 dex 文件不完整. 为此,设计了一套数据更新规则,保证 dex 文件重构的完整性和准确性.

壳代码在内存中释放原有 dex 文件是一个逐级操作的过程. 壳代码在对 dex 文件进行动态修改时,必将产生相应数据结构属性或内容的变化. 重构 dex 时,需要考虑相关数据的变化,正确更新,得到准确的 dex 文件.

图 2 展示了一个类方法在逐级解密时的变化路线,其中实线箭头代表壳代码在执行方法前,由于动态修改操作将 Native 方法还原成 Java 方法,虚线箭头代表壳代码在方法执行完毕后,重新将该方法标记为 Native 方法. 虽然通过监控方法调用,能够感知到壳代码的修改行为,但壳代码的重新加密操作,使得会从内存中获得同一方法的不同数据,在重构 dex 文件时,需要选择其中的正确部分.

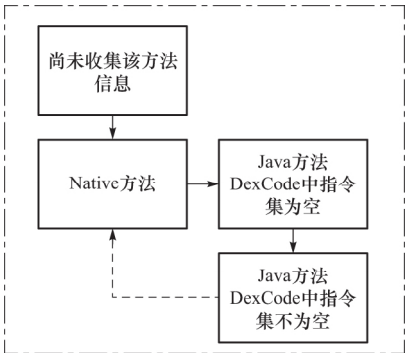

图 2 解密过程中类方法在内存中的变化

Fig. 2 Class methods transformation during decryption

为此,通过设计单向的数据更新规则,规定了在程序运行中可被接受的数据变化操作,当变化操作出现时,对 dex 文件中对应的数据结构进行更新. 由于整个更新过程是单向的,因此壳代码尝试重新加密或破坏原本有效数据的行为将不会导致数据更新. 以图 2 为例,仅接受一个方法从 Native 类型转换成 Java 类型的行为,并认为该转换是方法的解密过程,而当壳代码重新将方法标记为 Native类型时,将不会更新之前已获取的方法数据. 对于其他类型的数据,也设计了与之类似的单向转换规则.

1.4 脱壳有效性分析

为了验证脱壳系统的有效性,选取了市场上主流加壳厂商的加壳产品进行脱壳测试. 从应用市场中,选取了爱加密,阿里,百度,梆梆,360,腾讯 6 家厂商加壳的应用程序各 10 个,使用脱壳系统对其进行脱壳处理.

从如下两个标准来衡量脱壳的有效性.

① 脱壳系统给出的还原后 dex 文件,能够使用 BakSmali、JEB 等反编译工具进行处理,以证明 dex 文件能够满足后续静态分析的需要.

② 还原失败的类方法在所有方法中的比例. 还原失败的类方法,指的是在脱壳过程中,被壳代码替换成 Native 方法,且未能成功还原为字节码的类方法. 这一比例应当尽可能低.

最终的测试结果如表 2 所示. 由于加壳程序中本身存在真实的 Native 函数,因此通过人工分析的方法,排除了这部分真实 Native 函数,并给出了所有 Native 方法在总方法数的比例,以及还原失败的类方法占总方法数的比例.

表 2　加壳应用样本脱壳测试结果

Tab. 2　Unpack results of packed application samples

加壳厂商	反编译工具成功处理数量	Native方法比例/%	还原失败的类方法比例/%
爱加密	10 / 10	0.26	0.05
阿里	10 / 10	0.13	0
百度	10 / 10	0	0
梆梆	10 / 10	1.08	0.12
360	10 / 10	1.19	0.30
腾讯	10 / 10	0.08	0

可以看到,针对目前主流加壳服务,脱壳系统还原的 dex 文件均能用于后续静态分析,且还原失败类方法比例均在 0.3%以下,表明本脱壳系统具有良好的通用性和有效性.

1.5　性能开销和对比

为了测试脱壳系统所带来的额外性能开销,在型号为 Nexus 5 的智能手机上,使用了 CF-Bench 来进行对比试验. 通过在安装有脱壳系统和未安装脱壳系统的情况下,分别运行 CF-Bench 10 次并取平均值,得到最终结果如图 3 所示. 可以看到,脱壳系统引入的额外性能开销约为 11.7%,在可以接受的范围之内.

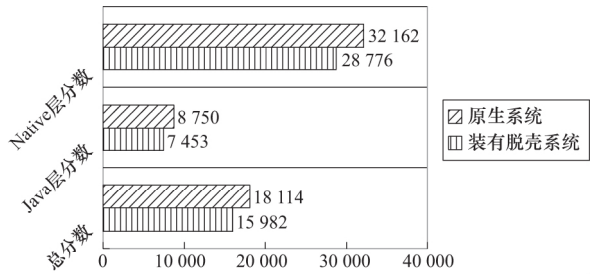

图 3　性能开销对比

Fig. 3　Performance overhead comparison

同时,选取了 drizzleDumper 和 DexHunter 这两项提供了开源代码的脱壳方案,与本文的脱壳系统进行实验对比. 为判断脱壳后 dex 文件的完整性和准确性,从互联网上选取了 5 个开源的 Android 应用程序进行编译,并利用加壳服务进行加壳处理.

在使用上述脱壳工具进行脱壳后,将能被正常反编译,且其中包含的方法代码与对应开源项目能一一对应的 dex 文件视作成功脱壳. 最终的测试结果如表 3 所示. drizzleDumper 通过特征搜索来定位 dex 文件,无法处理破坏了 dex 文件头部的情况;DexHunter 需要人为确定脱壳目标文件的名称和路径,同时对于不属于类数据的部分进行了连续的内存转储,使得最后的 dex 文件存在不能正常反编译的情况. 因此,针对目前的加壳服务,本文提出的系统能达到更好的脱壳效果.

表 3　不同脱壳方案实验结果对比

Tab. 3　Comparison results of different unpacker systems

加壳厂商	drizzleDumper	DexHunter	本文脱壳系统
爱加密	×	×	√
阿里	×	√	√
百度	×	×	√
梆梆	×	×	√
360	×	×	√
腾讯	×	√	√

2　实际应用

目前,静态检测领域存在大量优秀的分析工具,如基于流分析的 ScanDroid,基于代码相似性的 ViewDroid[6] 等等. 脱壳系统与静态分析工具相结合,通常被应用到恶意代码的检测和分析领域,并取得了良好的效果. 本文从另一个角度出发,利用脱壳系统,对应用市场中使用了加壳服务的程序进行了安全性评估.

2.1　样本收集

应用程序在被加壳后,其文件结构会发生不同程度的变化. 加壳服务通常会在 APK 文件中加入动态加载库,并通过 JNI 接口调用其中的函数. 通过分析不同类型的加壳服务,归纳了这些加壳服务自身的动态库名称,如表 4 所示.

表 4　加壳服务动态库名称列表

Tab. 4　Dynamic library name of different packer services

加壳服务	动态库名称
阿里	libmobisec.so, libdemolish.so
百度	libbaiduprotect.so
梆梆	libSecShell.so, libsecexe.so
爱加密	libexecmain.so, libexec.so
腾讯	libmain.so, libshell.so
360	libjiagu.so

Janus 是一个移动应用安全分析社区化平台[7],它收集了主流应用市场中应用程序的信息,并提供了一种自定义的规则语言来对数据库进行检索. 通过搜索检测 APK 文件中是否包含上述特征动态

库，来判断应用程序是否使用了加壳服务，以及加壳服务的具体名称，然后选取部分加壳程序下载分析.

历史下载量是应用市场排名算法中的一个重要影响因素. 根据特征匹配结果，随机选取了下载量处于不同区间的加壳应用程序共3 500个作为实验样本，实验样本收集于2017年5月. 为了确保下载量的准确性，将应用程序在百度、腾讯、360三家应用市场中被下载次数的平均值作为最终的下载量. 表5展示了这些应用下载量的分布情况.

表5 加壳应用样本下载量分布

Tab. 5 Download distribution of packed application samples

下载量区间	加壳应用样本数
<100	513
$10^2\sim10^3$	740
$10^3\sim10^4$	754
$10^4\sim10^5$	718
$10^5\sim10^6$	661
$10^6\sim10^7$	102
$>10^7$	6

2.2 样本分析

在软件开发过程中，由于开发人员的疏忽，可能为应用程序引入潜在的安全问题. 这些安全问题可能导致程序在某些极端边界条件下产生异常运行，也有可能被恶意攻击者利用，造成更为严重的后果. 为了批量分析应用程序中的安全问题，使用了AndroBugs[8]框架作为Android应用程序漏洞扫描的基础工具. 待扫描的应用程序被分为以下3类.

① 加壳后的应用：从应用市场上下载的加壳程序样本；② 脱壳后的应用：使用脱壳系统还原出原有dex文件后，重新打包的应用程序；③ 未加壳的应用：针对每一个加壳程序样本，选取出下载量偏差在10%以内的不加壳应用程序.

根据安全问题的严重程度，AndroBugs划分出了4种不同的安全等级：① Critical：代码符合某些已知的安全问题特征，需要进一步排查；② Warning：可能存在的安全隐患；③ Notice：检测到存在某些敏感操作，提供了额外的信息以供分析；④ Info：没有检测到安全问题.

需要注意的是，虽然并非所有被标记为Critical的代码都存在被攻击的安全漏洞，但是扫描结果的统计信息能够有效地对应用程序进行整体安全评估.

2.3 实验结果分析

图4展示了对上述3类应用程序进行漏洞扫描后，其不同安全等级的统计对比信息. 从实验数据可以看出，加壳程序被脱壳后，暴露出了更多的安全问题. 同时，下载量大致相同的应用程序中，加壳应用往往比未加壳应用存在更多的安全问题. 脱壳后的加壳程序样本Critical标记统计数目是未加壳程序样本的约1.6倍. 通过进一步的人工分析，总结出产生该结果的两个主要因素.

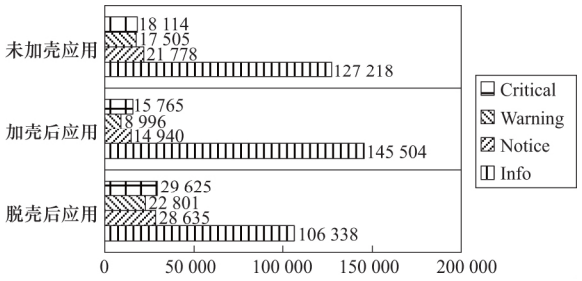

图4 应用程序漏洞扫描结果对比

Fig. 4 Application vulnerability scan results comparison

① 壳本身所引入的潜在风险. 加壳服务不可避免的需要在原有应用程序中引入自身代码，为了实现代码的解密和动态加载，部分壳代码使用了较为敏感的API，例如Runtime. getRuntime(). exec来执行特定命令，或者使用mprotect函数在内存中映射可读可写可执行的区域. 同时，加壳程序会通过修改APK中的AndroidManifest. xml文件，注册新的组件，并且获取额外的运行权限，以保障各项服务的正常运行. 例如，为了收集运行时产生的异常或者崩溃信息，需要增加读取日志、外部存储读写和网络通信等权限. 额外权限的提升，使得应用程序一旦被攻击，也就能够使得恶意代码在一个较高权限的环境中运行，增加了恶意代码的威胁程度.

② 开发者对于加壳服务的过于信任，导致在采用加壳服务后，在一定程度上忽视了安全编码的重要性. 加壳服务的核心目的在于对程序的代码和资源进行保护，防止暴露核心敏感逻辑和二次篡改. 虽然加壳提高了对应用程序进行分析的难度，但是却无法修复程序中潜在的安全问题.

以下3类是加壳程序样本脱壳后，增长数目较多的威胁类型.

① 安全套接层SSL的不合理使用. SSL为数据在不可信网络中的安全传输提供了保障. 许多应用程序均使用了包含SSL子层的通信协议(例如

HTTPS)完成与服务器之间的数据传输. 但是错误的调用相关函数,或者对参数的设置不合理,将导致整个通信过程不再安全. 例如,在扫描的脱壳后样本中,有 67% 设置了 ALLOW_ALL_HOSTNAME_VERIFIER 属性,使得应用在与远程服务器建立连接时,不会检查 SSL 证书的 Common Name,任何持有有效证书的攻击者将能进行中间人攻击[9],在通信过程中进行数据的窃取和伪造.

② 文件属性全局可读或全局可写. Android 系统沙箱机制保证了每个应用程序无法随意访问其他应用程序的资源. 然而当应用程序错误的设置自身私有数据属性时,将极大降低沙箱隔离的有效性,导致其它任何程序都能读取或改写其文件内容. 脱壳后样本中有 73% 设置了 MODE_WORLD_READABLE 或 MODE_WORLD_WRITEABLE 文件属性.

③ WebView 中使用 addJavascriptInterface 带来的威胁. 应用程序经常使用 WebView 类来在应用中内置一个浏览器组件,同时也可以通过 addJavascriptInterface 函数来往 WebView 中注入 Java 对象,这使得 JavaScript 代码能调用注入 Java 对象中的公有方法. 在 Android 4.2 以及之前的系统中,攻击者能利用这个特性,通过反射的方式访问被注入 Java 对象的所有公有域,甚至以应用程序的权限执行任意代码[10].

综上所述,对应用加壳虽然起到了防止应用被逆向破解和恶意窜改的作用,但由于壳代码本身不会改变应用原本的代码逻辑,存在于应用内的安全问题并不能得到修补,开发者不能因使用了加壳服务而忽视了安全编码的重要性. 同时,加壳本身也为应用引入了一定风险,一旦应用被恶意攻击,所造成的危害也会更大. 这与通常认为加壳提升了安全的直觉是相矛盾的.

3　结　论

本文设计并实现了一种多层次的自动化通用 Android 脱壳系统,能正确还原出加壳应用中被加密的代码逻辑,供静态分析工具使用. 实验表明,该系统能对市面上主流加壳服务保护后的应用进行正确脱壳. 利用该系统,本文对市场上被加壳的应用程序进行安全性评估,证明了脱壳系统的实际应用价值. 实验发现加壳应用比未加壳应用存在更多的安全问题,其原因在于加壳技术对原应用程序引入了更多的安全风险,及开发者可能过于信任加壳服务后忽视了安全编码的重要性. 加壳厂商和程序开发者应引起重视,进一步提高相关代码的安全性.

参考文献:

[1] Xu R, Saïdi H, Anderson R J. Aurasium: practical policy enforcement for Android applications[C]// USENIX Security Symposium. Bellevue, Washington: USENIX Association, 2012.

[2] 徐君锋,吴世忠,张利. Android 软件安全攻防对抗技术及发展[J]. 北京理工大学学报,2017,37(2):163-167.
Xu J F, Wu S Z, Zhang L. Survey on attack and defense technologies of Android software security[J]. Transactions of Beijing Institute of Technology, 2017, 37(2): 163-167. (in Chinese)

[3] Rastogi V, Chen Y, Jiang X. Droidchameleon: evaluating android anti-malware against transformation attacks[C]// Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security. [S. l]: ACM, 2013: 329-334.

[4] 林佳萍,李晖. 安卓恶意软件检测研究综述[J]. 信息网络安全,2016 (10):80-88.
Lin J P, Li H. Review of Android Malware detection[J]. Netinfo Security, 2016 (10): 80-88. (in Chinese)

[5] Feng Y, Anand S, Dillig I, et al. Apposcopy: semantics-based detection of Android Malware through static analysis[C]// Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. [S. l]: ACM, 2014: 576-587.

[6] Zhang F, Huang H, Zhu S, et al. ViewDroid: towards obfuscation-resilient mobile application repackaging detection[C]// Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks. [S. l]: ACM, 2014: 25-36.

[7] Anon. Janus[EB/OL]. [2018-06-12]. https://www.appscan.io.

[8] Anon. AndroBugs[EB/OL]. [2018-06-12]. https://github.com/AndroBugs/AndroBugs_Framework.

[9] Fahl S, Harbach M, Muders T, et al. Why Eve and Mallory love Android: an analysis of Android SSL (in) security[C]// Proceedings of the 2012 ACM Conference on Computer and Communications Security. [S. l.]: ACM, 2012: 50-61.

[10] Luo T, Hao H, Du W, et al. Attacks on Web View in the Android system[C]// Proceedings of the 27th Annual Computer Security Applications Conference. [S. l]: ACM, 2011: 343-352.

(责任编辑:刘芳)