安卓系统服务中输入验证型漏洞的初步分析

曹 琛[1], 高 能[1], 向 继[1], 刘 鹏[2]

[1] 中国科学院信息工程研究所 北京 中国 100093
[2] 美国宾夕法尼亚州立大学

摘要 输入验证型漏洞在 Web 安全领域颇受重视, 但其在安卓安全研究领域却在很大程度上被忽视。我们发现由于安卓系统中独特的框架层设计, 安卓设备需要对系统服务(System Service)进行具体的输入验证分析。本文工作对安卓系统中的输入验证型漏洞进行了分析, 1)我们分析了系统服务的攻击面, 介绍了目前系统服务中的输入验证的实现情况; 2)我们开发了一个漏洞扫描器, 通过向系统服务发送带有畸形参数的请求对其进行模糊测试。在对安卓系统中 90 多个服务和 1900 多个函数进行综合的分析后, 我们发现了 16 个系统服务漏洞。最后, 我们把这些漏洞报告给谷歌并得到了谷歌的确认。

关键词 安卓系统服务; 输入验证; Buzzer; 漏洞
中图法分类号 TP309.1

Towards Analyzing the Input Validation Vulnerabilities associated with Android System Services

CAO Chen[1], GAO Neng[1], XIANG Ji[1], LIU Peng[2]

[1] Institute of Information Engineering, CAS, Beijing 100093, China
[2] The Pennsylvania State University, USA

Abstract Although the input validation vulnerabilities play a critical role in web application security, such vulnerabilities are so far largely neglected in the Android security research community. We found that due to the unique Framework Code layer, Android devices do need specific input validation vulnerability analysis in system services. In this work, we take the first steps to analyze Android specific input validation vulnerabilities. In particular, a) we take the first steps towards measuring the corresponding attack surface and reporting the current input validation status of Android system services. b) We developed a new input validation vulnerability scanner for Android devices. This tool fuzzes all the Android system services by sending requests with malformed arguments to them. Through comprehensive evaluation of Android system with over 90 system services and over 1,900 system service methods, we identified 16 vulnerabilities in Android system services. We have reported all the issues to Google and Google has confirmed them.

Key words android system services; input validation; Buzzer; vulnerabilities

1 背景

安卓设备(包括智能手机、平板等)的数量近年来在不断的增长, 2014 年全球智能手机的出货量超过了 10 亿台[5]。与此同时, 针对安卓设备的恶意攻击和安全事件也在不断的增加, 例如, 赛门铁克 2015 年的网络安全威胁报告指出, 赛门铁克发现 17%(近 100 万个)的安卓应用程序是恶意软件。

目前安卓系统中被广泛研究的漏洞和攻击类型主要有权限提升攻击[14,15,18]、恶意软件[34,39]、重打包攻击[22,36,37]以及组件挟持[27]攻击等。

本文工作对安卓设备上输入验证型的漏洞进行了分析。输入验证类型漏洞在 Web 安全领域很受重视, 但其在安卓安全研究领域中却在很大程度上被忽视。目前 Web 应用安全领域的输入验证研究主要针对以下四种问题: 1. 对用户输入处理不当造成 SQL 注入, 最终导致任意 SQL 命令执行[24]; 2. 利用网页验证中的漏洞, 在客户端注入脚本代码造成 XSS(跨站脚本攻击)攻击[26,29]; 3. 服务器端对收到的 HTTP 请求是否由可信客户端发起没有作正确判断,

通讯作者: 高能, 博士, 副研究员, Email: gaoneng@iie.ac.cn。
本课题得到 863 计划-云计算安全体系架构研究(No. 2013AA01A214)资助。
收稿日期: 2015-10-13; 修改日期: 2015-11-30; 定稿日期: 2016-01-03

导致 CSRF(跨站请求伪造)攻击[12]; 4. 对用户输入的数据未作足够的边界值判断引起缓冲区溢出, 可能导致应用被锁、内存中的其他数据被覆盖或使服务端崩溃等问题[16]。

尽管已经有了这四种针对输入数据进行验证的工作, 我们发现安卓设备在系统服务层上还需进行更具体的输入验证工作, 而这样的工作目前还没有在相关论文中出现。系统服务(System Service)是在后台运行的系统或系统应用进程, 它们是安卓设备的重要组成部分。系统服务封装了系统的一些功能, 如蓝牙、通话等, 并给上层的应用程序提供服务。系统服务代码在整个安卓框架层占主要部分, 从这个层面上说, 系统服务也是安卓系统和传统 PC 系统的主要不同之处。传统的 PC 系统没有运行任何类似这种类型的系统服务, 所以传统的输入验证漏洞分析不适用于安卓的系统服务层。

为解决安卓设备系统服务上的输入验证漏洞, 这里先引出两个问题:

问题一: 系统服务是在多大程度上忽略了输入验证?

问题二: 如何开发一个高效的扫描器去发现这些输入验证类型相关的漏洞?

为了解决问题一, 我们对系统服务的代码进行人工审计, 进而去发现那些没有对输入数据进行验证的系统服务函数。最后, 我们发现绝大多数系统服务函数会验证其参数, 但是一些函数没有进行充分的参数检查, 导致仍有一些漏洞出现在系统服务中。

为解决问题二, 我们设计和实现了一个半自动化的工具, 通过向系统服务不断的发送带有畸形参数的请求对其进行模糊测试, 进而去发现那些输入验证类型的漏洞。利用这个工具, 我们在系统服务中发现了 16 个输入验证类型的漏洞, 这些漏洞可以导致系统崩溃、屏幕冻结以及阻止系统卸载软件等问题。恶意应用利用这些漏洞时可能需要特殊的权限, 但是对于大多数攻击(如使系统崩溃或冻结屏幕), 恶意应用不需要任何的权限就可以实施。

我们的工作贡献总结如下:

(1) 分析了安卓系统服务层输入验证类型的漏洞;

(2) 分析了系统服务的攻击面以及介绍了目前的系统服务函数对输入参数进行验证的方式;

(3) 设计实现了一个系统服务层输入验证类型漏洞的扫描器;

(4) 发现了 16 个安卓系统服务层的漏洞。

文章剩余部分的结构如下:

第二部分介绍了安卓的系统服务模型, 第三部分介绍了目前安卓系统服务函数对输入数据进行验证的情况, 第四部分详细阐述了半自动化扫描器的设计以及主要的发现, 对扫描器局限性的讨论在第五部分给出, 第六部分对相关工作进行总结。

2 系统模型

2.1 安卓应用程序和系统服务的接口

应用程序向系统服务发送请求通常需要调用 API 函数, 例如如果一个应用想要设置 WiFi, 必须调用 *Context.getSystemService(Context.WIFI_SERVICE)* 和 *WifiManager.setWifiEnabled(boolean enabled)*这两个函数, 通过两步完成。但是, 一些系统服务函数并没有提供这些 API 函数给应用程序, 我们发现最好的发送请求的方法是用一个名为 *transact* 的 API, 它仅需一步就可以完成调用且可以调用任意公开的系统服务函数。

一个请求本质上是几个输入参数的集合, 为发送一个请求, 应用程序首先调用 *transact*, 它需要四个参数。参数一是一个整型, 指定了被调用的系统服务函数; 参数二是一个 *Parcel* 对象, 它填充了对应系统服务接口字符串以及对应函数所需要的参数; 第三个参数是一个 *Parcel* 对象, 如果有返回数据时, 会保存在这个对象中; 最后一个参数是一个整型, 指定了对应的系统服务函数是否需要返回数据。当这个函数得到执行时, 应用程序进程将通过 binder 驱动发送这些输入参数到系统服务层。系统服务层通过 *onTransact* 这个函数接收参数。*onTransact* 函数可以处理各种类型的请求, 它会做两件事情: 首先, 从第二个参数对象中解析所需要的参数, 接着根据第一个参数调用对应的系统服务函数。

以 Wifi 系统服务为例, 图 1 描述了应用程序和系统服务之间的接口。应用发送的请求由 *transact* 这个 API 通过 binder 驱动发送到 *onTransact* 函数。虚线表示两个函数的参数是相同的, 在 *onTransact* 解析出对应的参数后, 调用系统服务函数 *WifiServiceImpl.setFrequencyBand* 并设定对应的整型参数 *band* 以及布尔型参数 *persist*。

2.2 威胁模型

我们假定输入验证类型攻击是由恶意应用发起的, 且恶意应用已经安装在目标设备中。为实现这种攻击, 恶意应用可能需要某种权限, 如攻击 Wifi 系统服务可能必须获取 *CHANGE_WIFI_STATE* 权限。但对其他类型的攻击(如使系统崩溃、冻结屏幕), 恶

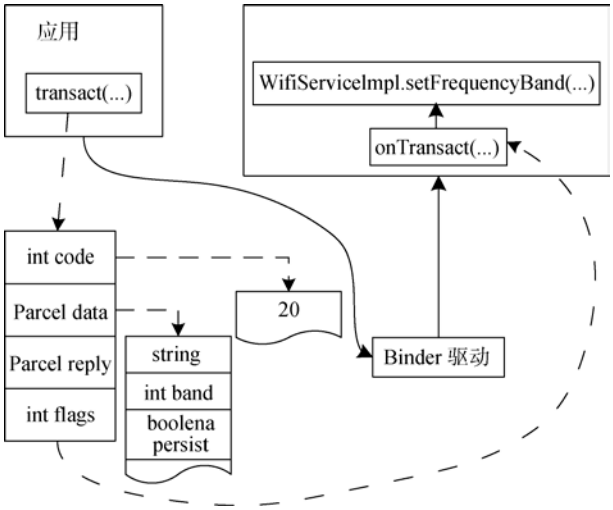

图 1　应用程序和系统服务之间的接口

意应用只需要发送一个特殊的请求给系统服务，不需要任何权限。值得注意的是恶意请求与合法请求看起来非常相似，这也是为什么这种攻击难以被检测到的原因。

2.3　问题描述

本文工作基于 Android 5.0.1(最新版本)，该版本系统包含 96 个系统服务，共计 1972 个接口函数(我们把 onTransact 调用的函数称为“接口函数”)。平均每个接口函数包含 1.8 个输入参数，一个应用程序通过 *transact* 能够发送带四个参数的请求，而接口函数的参数都存储于其第二个参数中，因此输入验证类型攻击面包含 1972×1.8+3 *IAVR (Input Argument Value Range)* 个向量，每个 *IAVR* 对应特定参数的取值范围。

为解决安卓系统服务中的输入验证漏洞，这里引出两个问题:

问题一: 系统服务是在多大程度上忽略了输入验证?

问题二: 如何开发一个高效的扫描器去发现这些输入验证类型的漏洞?

3　问题一的解决

安卓系统服务存在如此多的攻击面，因此，如果系统服务函数没有对输入数据进行充分的验证，攻击者可以构造畸形数据，从而导致系统崩溃或功能不可用。我们主要通过手工检查系统服务中的接口函数去研究函数体对参数的验证情况。我们定义有输入验证的函数应该满足以下条件之一: a) 函数至少检验了一个参数; b) 函数需要应用程序满足某种条件。之所以把 b)作为一个条件是因为如果应用程序不满足某种条件(如没有特定的权限)，提交给系统服务的参数就会被忽略。另外，如果一个函数在它自己的函数体里未作输入验证，但是直接调用了另一个包含参数验证的函数，那么我们认为它作了参数验证。图 2 给出了有输入验证和无输入验证的函数比例。对于没有参数的函数，我们把它归到一个特定的分类，另外，对于函数体为空或参数未使用或仅仅返回无效值的函数，我们把它们归为同一类。通过图 2 我们可以看出，绝大多数系统服务函数都做了输入验证。

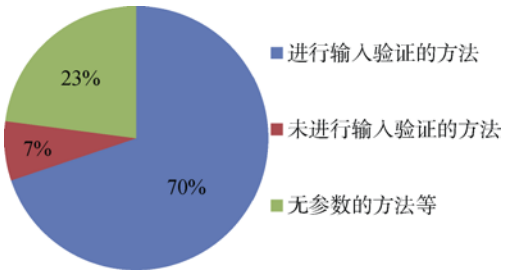

图 2　不同类型的系统服务函数的分布

由此看来，安卓系统服务在输入验证上做的比较好。例如，大多数函数在收到一个字符串类型的包名时，都会验证包名的真实性，如果这个字符串不是包名，函数将返回错误信息。此外，很多系统服务函数需要应用程序有某种权限，如 *STATUS_BAR_SERVICES* 和 *BLUETOOTH* 等，第三方的应用程序可以申请一些权限，但是不会授权得到其他特殊的权限。

尽管很多函数都会验证收到的参数，但有一些函数没有作充足的验证。例如，*BluetoothManagerService.registerStateChangeCallback* 函数虽然需要应用程序拥有 *BLUETOOTH* 权限，但是它未作充分验证，导致第三方应用程序可以发起请求，只需要设置参数为特定值，那么就会在系统蓝牙状态改变时，系统崩溃。

总体来说，大多数系统服务函数都作了输入验证，但是一些函数没有作充足的验证。所以需要更多的工作去研究这些函数的输入验证以及由这些验证问题引发的漏洞，我们因此开发了一个高效的漏洞扫描器去帮助发现这些漏洞。

4　问题二的解决

我们设计实现的扫描器是一个半自动化工具，它通过构造一些畸形数据向系统服务发起请求，从而对系统服务进行模糊测试。我们把这个扫描器称为 Buzzer (Binder Fuzzer)，在介绍它的设计和实现之前，我们首先解释 Buzzer 中做的前提条件，这有助于对后边关于 Buzzer 的设计和实现细节的理解。

4.1 前提

Buzzer 在安卓系统中表现为一个普通的第三方应用程序, 它可以得到第三方应用能获得的所有权限。

目标测试系统是纯净的 Android 5.0.1 系统, 系统镜像的获得有两个地方, 一个是直接由安卓源码编译, 另一个是从谷歌官网下载的 Nexus 镜像。

4.2 设计

由于安卓系统服务可以收到来自应用程序的请求, Buzzer 利用这个方法不断的向这些服务发送请求。

在 Buzzer 的设计中存在两个挑战。第一个挑战是如何使 Buzzer 具有可扩展性。虽然我们在模糊测试的时候, 目标系统版本选为最新的 5.0.1, 但我们希望在以后出现新版本时, Buzzer 能够很容易的扩展。所以我们的方案是把不同的功能模块化, 对于一个系统服务, 仅有一个对应的模块, 这个模块提供了该系统服务需要的所有参数, 即在 Buzzer 中不同的系统服务对应不同的模块, 请求的发送和记录也在不同的模块进行。另外一个挑战是在对系统服务模糊测试时, 怎样记录和分析产生的日志。因为我们假定目标系统未经过任何改动, 所以不能在系统中添加代码。我们的解决方法是利用安卓日志系统, 使 Buzzer 产生尽可能多的日志信息。这些日志信息可以在系统内部进行分析, 也可以在系统外部分析, 但考虑到系统内部分析可能会增加系统的压力, 所以我们选择在系统外部分析。

图 3 展示了 Buzzer 的架构。Buzzer 由四部分组成: 服务模块、服务选择、请求发送和记录、日志分析。

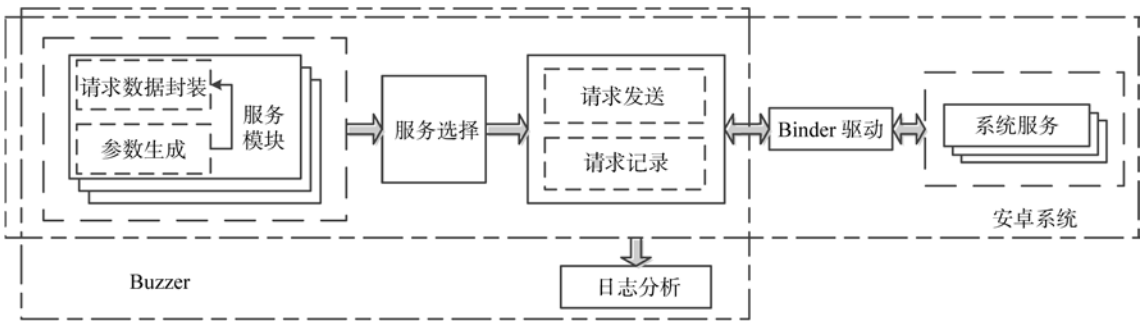

图 3 Buzzer 的架构

一个服务模块对应安卓系统中的一个系统服务, 它会产生对应接口函数所需的所有参数, 此外每一个模块提供一个统一的接口给服务选择部分。

服务模块由两部分组成: 参数生成和请求数据封装。参数生成部分生成了系统服务函数需要的所有参数, 并将其提供给请求数据封装部分。如果某种类型参数会被很多系统服务用到, 这些参数的生成函数将被封装成一个统一的模块。在请求数据封装部分收到参数后, 会把所有参数封装成请求发送部分可接收的对象。

服务选择模块担任 Buzzer 中的管理员角色, 它负责选择服务模块测试系统服务, 另外被测试的系统服务函数数量也是由该部分产生一个随机数决定。

请求发送部分通过 binder 驱动向系统服务发送请求, 由于请求数据已经由之前的模块封装成对象, 这部分不需要再对不同的系统服务函数进行处理。

请求记录部分结合了安卓的日志系统, 它会记录包括系统状态在内的所有日志, 同时也会输出系统服务函数返回的内容。

日志分析是离线进行的, 它在测试进程结束后在系统外部进行分析。目前日志分析是半自动化的, 它仅仅进行了简单的过滤, 需要人工审计。

4.3 实现

我们在最新安卓系统中实现了 Buzzer, 并在实现过程中解决了以下难题。

4.3.1 参数的产生

Buzzer 能够高效的生成参数绕过系统服务函数的初始检查, 这也是 Buzzer 的优势之一。比如, 一些函数会检查收到的数据是否为 *null*, 如果参数为 *null*, 这些函数会抛出异常。通过手工分析系统服务函数的源代码, 并记录对应所需的参数, Buzzer 能够有效产生绕过初始检查的参数。

对于那些原始类型的参数, Buzzer 用 *java.util.Random* 类去产生。*String* 不是原始类型, 但是大量的参数都是 *String* 型, 除了诸如包名、文件路径等类型的特殊字符串外, Buzzer 随机产生一个 *100×1024* 长度内的字符串, 太长的字符串会使 Buzzer 崩溃。

对于其他类型的参数, Buzzer 尽可能的产生多个值。以 *android.location.Criteria* 为例, 该类有 9 个字段需要初始化, 每个字段都有不同的取值范围, Buzzer 通过遍历每一个字段的所有取值范围产生参数。即便如此, 一些复杂类型的参数也很难生成多个

数据, 如对于 *Intent* 类型的参数, Buzzer 只能依据安卓文档经验性的产生。

在安卓系统中, 很多类并没有公开给第三方应用, 这些类被称为隐藏类。*Location* 服务中的接口函数收到一个参数, 它的类型是 *android.location.LocationRequest*, 这就是一个隐藏类。有两种方式去调用隐藏类, 一种是用 Java 的反射机制, 另一种是编译安卓源码, 用自编译的包含隐藏类的库去替换安卓 SDK 中的库。由于第一种方法需要更多的 Java 代码, Buzzer 采用了第二种方法。

对于无法生成的参数, 这里用 *null* 代替。此外, 对于一些非 *null* 的参数, 可能要求应用程序拥有某种权限, 但第三方应用程序不可获取该权限, 对于这种类型的参数, 也用 *null* 代替。

概括而言, Buzzer 使用半智能化的方法产生畸形参数, 并尽可能遍历所有参数的取值范围空间, 尽力深入测试系统函数。

4.3.2 请求处理

Buzzer 通过 binder 驱动发送请求给系统服务, 它借鉴了安卓命令——service, service 命令可用于列出所有的系统服务, 它也可以封装简单的参数请求调用系统服务, 请求中的 *parcel* 对象在 native 代码中完成封装。

在实现过程中有两个问题需要解决。首先是怎样把 *parcel* 对象由 Java 层传到 native 代码层。这个问题通过阅读安卓源码, 学习代码结构来解决。在安卓系统中, 当一个 *parcel* 对象在 Java 空间创建时, 其 *mNativePtr* 字段由原生 native 层创建的变量初始化, 这个特殊的变量是一个长整型的指向 native 层 *parcel* 对象的指针。当调用任意函数时, Java 中的 *parcel* 对象把这个值传给原生代码层, 即 Java 空间的任何操作都是由 native 层完成的。因此, Buzzer 在 native 层创建了一个未初始化的 *parcel* 对象, 当 Java 层的 *parcel* 对象填充参数后, 这个对象的 *mNativePtr* 字段值将会分配给 Buzzer 的 native 层 *parcel* 对象, 之后 *parcel* 对象再封装在请求数据中, 通过 binder 驱动发送给系统服务。另外, 由于返回值也是一个 native *parcel* 对象, Buzzer 可以输出这个对象在内存中的数据。第二个问题是如何编译 Buzzer。由于 Buzzer 很大程度上依赖这些包含私有 API 的原生代码, 普通的安卓 NDK 不能编译 Buzzer, 我们通过编译安卓源码, 提取必要的库和头文件配置目标版本的库, 如 *libandroid_runtime.so*, *libbinder.so* 等。综上, Buzzer 用安卓原生私有 API 去发送请求, 我们为 Buzzer 部署了一个特殊的编译环境。

此外, Buzzer 能够发送畸形的数据给 servicemanager 这个特殊的系统服务。每一个系统服务在 binder 驱动中都有一个对应的 32 位整型的 *token*, 系统服务必须在 servicemanager 中注册。任何应用想要给系统服务发送请求, 必须获取一个由 servicemanager 产生的对应服务的 *handler*, 这个 *handler* 包含了一个 *token* 值。Servicemanager 是一个 *token* 值为 0 的特殊系统服务。通常所有发送给 servicemanager 的请求必须按照严格的格式在特定函数中封装, Buzzer 分两步绕过这些函数。首先 Buzzer 从 native 层获取由 servicemanager 产生的任意一个系统服务 *handler*, 接着 Buzzer 把获得的 *handler* 的 *token* 设为 0, 然后用这个畸形的对象去调用 *transact*。

4.3.3 大量的接口函数

安卓 5.0.1 系统有 96 个服务, 1972 个接口函数可以接收请求, 如果对所有的接口函数进行分析, 会浪费很多时间, 因为一些函数仅仅返回错误信息。我们用两种方式提高 Buzzer 的效率。首先, 大量的系统服务函数要求应用程序具有某些特殊的权限, 但是第三方应用程序不可能被授予这些权限, 同样 Buzzer 也不能, 因此 Buzzer 忽略了这种类型的 703 个函数。其次, 很多系统服务中的函数没有函数体或者仅仅返回空值, 如 *ConnectivityService.pendingListenForNetwork* 接口, Buzzer 同样忽略了 8 个这种类型的函数。所有忽略掉的函数都是人工发现的, 即通过人工审查了 1972 个函数, 标记出那些可以被忽略的函数。总而言之, Buzzer 过滤了那些不能被模糊测试或者没有内容的函数。

4.4 发现的漏洞

本工作使用 Nexus 6(32 位)和 Nexus 9(64 位)两部设备进行实验, 两部设备都是安卓 5.0.1 系统。在 Buzzer 的帮助下, 我们发现了 16 个漏洞, 将其分为 5 个类别, 此外我们还有一些其他的发现。

4.4.1 自动生成代码引发的漏洞

一些系统服务可接收类型为特定接口的参数, 该类型的接口是由 *AIDL (Android Interface Definition Language)* 系统产生。以 *IPackageManager* 接口为例。代码片段 1 中的代码是由 IPackageManager.aidl 文件生成。

```
public static android.content.pm.IPackageManager
asInterface(android.os.IBinder obj)
{
  if ((obj==null)) {
  return null;
  }
  android.os.IInterface iin =
```

```
            obj.queryLocalInterface(DESCRIPTOR);
        if (((iin!=null)&&(iin instanceof
            android.content.pm.IPackageManager))) {
            return
            ((android.content.pm.IPackageManager)iin);
    }
    return new android.content.pm.
            IPackageManager.Stub.Proxy(obj);
    }
```

代码片段 1 AIDL 系统产生的代码

由代码片段 1 可看出, *asInterface* 函数参数为 IBinder 对象, 一个非空的 IBinder 对象通过检查后会被映射到一个 *android.content.pm.IPackageManager* 接口, 此外通过封装 IBinder 对象, 会产生一个 *IPackageManager* 代理, 即只要 IBinder 对象不为空, *asInterface* 函数总会返回一个 *IPackageManager* 接口。这就是漏洞所在, 攻击者可以通过一个任意非空 IBinder 对象伪造特定的接口。

一些系统服务函数可接收一些接口作为它们的参数, 但是它们并未对这些接口进行充分的验证。通过上述的漏洞, 一个恶意应用可以利用一个伪造的对象(假设对应系统服务 X)伪造一个接口, 然后将其发送给系统服务 A, 当 A 的函数去调用这个接口时, 由于未处理 *SecurityException* 异常, 系统服务 A 将会崩溃。这个函数用了一个伪造接口的字符串描述符去填充一个 *parcel* 对象, 但是真正的接口由系统服务 X 实现, 当系统服务扫描 *parcel* 对象中的字符串描述符时, 发现不同后会抛出一个 *SecurityException* 异常给系统服务 A。图 4 给出了整个过程。

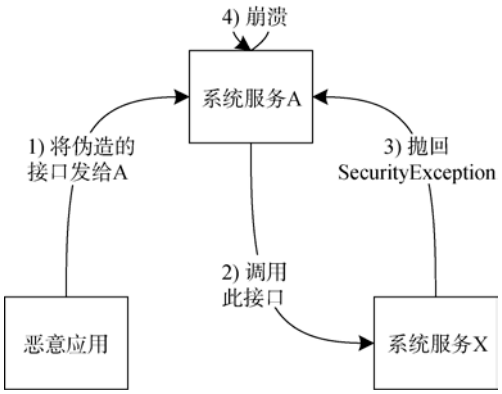

图 4 攻击系统服务

第一个这种类型的漏洞是通过手工分析 *AIDL* 系统以及手工构造恶意接口发现的, 之后我们把这种畸形参数的生成函数放到 Buzzer 的参数生成模块中, 在安卓系统服务中作测试后, 发现了其他相似的系统服务漏洞, 表 1 给出了这些系统服务的名字以及对应的函数名和接口。

这 10 个漏洞是同一种类型, 但是漏洞导致的结果不同, 最后崩溃的进程包括 *system_server*、*com.android.settings* 等。另外它们的触发条件也不相同, 这些跟对应的接口函数有关。

表 1 受影响的系统服务和接口

服务	存在漏洞的函数	接口
bluetooth manager	registerAdapter	android.bluetooth.IBluetoothManagerCallback
bluetooth manager	registerStateChangeCallback	android.bluetooth.IBluetoothStateChangeCallback
launcherapps	addOnAppsChangedListener	android.content.pm.IOnAppsChangedListener
lock settings	registerObserver	com.android.internal.widget.ILockSettingsObserver
wallpaper	getWallpaper	android.app.IWallpaperManagerCallback
fingerprint	startListening	android.service.ngerprint.IFingerprintServiceReceiver
window	watchRotation	android.view.IRotationWatcher
display	createVirtualDisplay	android.hardware.display.IVirtualDisplayCallback
mount	mountObb	android.os.storage.IObbActionListener
audio	startWatchingRoutes	android.media.IAudioRoutesObserver

display 服务和 mount 服务中的漏洞无需任何操作就可以使系统服务崩溃, 因为这两个服务在收到请求后会直接调用接口。其他的系统服务的漏洞触发条件列举如下: 当启动器中显示的应用发生变化时, 可能会导致 launcherapps 服务崩溃, 这个漏洞可以阻止系统卸载应用程序; 蓝牙状态改变时可能引起 bluetooth_manager 服务崩溃, 恶意应用需要有访问蓝牙的权限; 应用程序发送一个请求给 fingerprint 服务时, 可能引起其崩溃, 如发送一个删除指纹 ID 的请求, 关于这个的更多细节将会在 4.4.6 介绍; Window 系统服务崩溃由设备屏幕方向旋转触发; lock_settings 服务崩溃由用户改变屏幕锁设置触发; wallpaper 服务崩溃由墙纸设置改变触发; 媒体频道由当前设备换为外置扬声器或插入耳机等设备时, 可能引起 audio 服务崩溃。

总之, 这些漏洞的触发条件取决于具体的函数

接口, 漏洞可导致系统服务以及对应的主进程崩溃。

4.4.2 Servicemanager 漏洞

Servicemanager 是一个 *token* 值为 0 的特殊的系统服务, 它只能接收 *flags* 参数为 0 的请求, 如果收到 *flags* 值为 1 的请求时, Servicemanager 会退出并重新启动。

Servicemanager 运行在*/system/bin/servicemanager* 进程中, 它是由 *init* 进程启动的一个核心进程, Servicemanager 的重启会造成其他几个进程也重启, 包括 *healthd,*、*zygote*、*media*、*surfaceinger* 和 *drm*[2,3] 等。由于 *zygote* 也是一个重要的进程, 它的重启会导致所有的应用程序退出, 系统应用程序重启。另外, Servicemanager 的重启对*/system/bin/keystore* 进程无影响, 这会导致后文描述的一个问题, 在这之前, 我们先解释为什么 Servicemanager 不能接受 *flags* 为 1 的请求。

当 Servicemanager 收到 *flags* 为 1 的请求时, 在 binder 驱动中对应的 *binder_thread* 中的 *transaction_stack* 值被设为空, *binder_thread* 中的 *return_error* 值被设为 *BR_FAILED_REPLY*。当 *binder_thread* 函数被调用时, binder 驱动检测到 *return_error* 的值异常, 于是返回对应错误代码给 Servicemanager 用户空间, 最终 Servicemanager 在收到错误代码后退出并重启。

/system/bin/keystore 进程中运行 *android.security.keystore* 系统服务, 当这个进程启动时, 会在 Servicemanager 中注册, 也就是说如果该进程比 Servicemanager 进程启动得早, *android.security.keystore* 就不会在 Servicemanager 中注册。由于所有的应用程序在和系统服务通信时必须从 Servicemanager 拿到一个句柄, 如果上述情况存在, 即 *android.security.keystore* 未在 Servicemanager 中注册, 应用程序就无法使用 *android.security.keystore* 这个服务。

由于 *android.security.keystore* 服务负责安卓系统中的密钥操作, 因此任何与密钥有关的行为都无法进行, 如卸载应用、添加新用户等, 在“设置”选项中, “安全”和“VPN”两个选项以及它们的子菜单都无法点击。

这个漏洞是手工发现的, 当阅读 Servicemanager 源码时, 我们发现每个函数发送的请求中 *flags* 参数都为 0, 所以我们就尝试研究在利用 Buzzer 发送一个 *flags* 为 1 的请求给 servicemanager 时, Servicemanager 的状态会是什么, 之后我们继续分析它的源码和 binder 驱动部分, 最终找到了根本原因, 同时也发现了其对 *android.security.keystore* 的影响。

4.4.3 WiFi 系统服务漏洞

现在的安卓设备都支持双频率 Wifi, 即 *2.4GHz* 和 *5GHz*, 安卓系统允许用户选择不同的频段。默认情况下, WiFi 频段的索引 *band* 值只能为 0(自动)、1(*5GHz*)和 2(*2.4GHz*)这三种取值, 当 *band* 的值大于 2 且 Wifi 驱动解析为某一个不是当前连接热点的频率时, 设备就不能连上当前热点。除了这种攻击方式外, 在设置中还存在一个漏洞。当 *band* 被设为大于 2 时, 如果用户想在设置中更改频段, 设置就会崩溃。结合这两种方式, 攻击者可以使设备永远无法连接上一个特定 WiFi 热点。

系统设置应用的漏洞存在 *AdvancedWifiSettings* 这个类中, 代码片段 2 给出了这个类的漏洞函数片段, 函数中的 *index* 值是从 Wifi 系统服务中获取的, *fummaries* 这个数组是从 *arrays.xml* 资源文件中获取的, 该数组有 Automatic、5 GHz only 和 2.4 GHz only 这三个项。因此当 *index* 值大于 2 时将会触发 *StringIndexOutOfBoundsException* 异常, 而函数没有处理这个异常, 所以导致设置崩溃。

```
private void updateFrequencyBandSummary(
      Preference frequencyBandPref, int index) {
String[] summaries = getResources().
  getStringArray(
    R.array.wifi_frequency_band_entries);
frequencyBandPref.setSummary (summaries
  [index]);
}
```

代码片段 2 存在漏洞的函数

总之, 利用这个漏洞可以把 Wifi 频段索引设为某一个特定值, 如果 *band* 值大于 2 且 Wifi 驱动解析该值后设置的频率与 Wifi 热点频率不同, 设备将失去和 Wifi 热点的连接, 如果用户此时打开设置程序, 设置将崩溃, 因此设备将无法连上该 Wifi 热点。

WiFi 系统服务的漏洞是由 Buzzer 测试发现。我们发现测试结束后即使设备重启也仍然无法连接特定 WiFi 热点。之后通过手工分析记录的测试函数和日志信息, 再加上查看对应的源码, 最终我们找到了出现漏洞的位置和原理。

4.4.4 搜索服务中的漏洞

搜索服务负责展示搜索界面以及维护可进行搜索的 activity 的注册。自安卓 5.0 以后, 搜索服务中增加了 *launchAssistAction* 这个函数, 这个函数负责打开一个注册 *ACTION_ASSIST* 广播的 activity。谷歌 Nexus 系统镜像中包含一个名为 *com.google.android.googlequicksearchbox* 的系统应用, 这个应用在接收 *ACTION_ASSIST* 广播后会打开一个 activity。当攻击者持续不断的发送请求给搜索服务时, 这个系统应用的 activity 不断被打开, 之后整个屏幕会冻结, 用

户点击任何地方都无效，直到电池电量耗尽，手机关机。或者只能通过长按电源键强制重启。

这个漏洞是手工发现的，这个漏洞会造成界面被搜索 activity 覆盖，我们怀疑如果漏洞函数被不断调用时，界面可能无法操作。于是利用 Buzzer 不断发送请求，最终证实了之前的想法，发现该漏洞。

4.4.5 NULL 引用导致的漏洞

media.audio_policy 服务中的 *registerClient* 函数未检测收到的参数是否为 *NULL* 就直接使用，如果收到参数为 *NULL* 时，该服务就会崩溃，由于这个服务运行在*/system/bin/mediaserver* 进程中，该进程也会崩溃，同样运行在该进程上的 media.audio_flinger、media.player、media.camera 和 media.sound_trigger_hw 等服务也会重启，这意味着用户无法看视频、听音乐和使用相机。如果一个恶意程序不断的发送请求使 media.audio_policy 服务和*/system/bin/mediaserver* 进程崩溃，那么与 media 相关的服务都将无法正常使用。media.player 服务中的 decode 函数具有同样的漏洞。

SurfaceFlinger 服务中的 *getActiveConfig* 函数同样未检测参数是否为 *NULL* 就直接调用，SurfaceFlinger 服务运行在 *system_server* 进程空间中，因此该服务的崩溃会导致整个手机系统的重启。

总之，利用这几个漏洞会导致对应的服务和进程重启，而且恶意的应用不需要任何权限就可以实施攻击。

这三个漏洞是由 Buzzer 自动扫描发现，由于该类型漏洞的利用结果很明显，很容易分析日志信息定位到漏洞的具体位置。在利用 media.player 服务中漏洞时使用了一个小技巧。Buzzer 发送了一个空的请求，对应系统服务从 *Parcel.readCString* 函数获得一个字符串，这个函数不像 *Parcel.readString8* 函数和 *Parcel.readString16* 函数那样返回一个空字符串，它返回一个 *NULL* 字符串，系统服务是通过这种方式才获得一个 *NULL* 参数，因此 *Parcel.readCString* 这个函数的设计实现还需再斟酌。

4.4.6 其他的发现

安卓 5.0.1 系统有 96 个系统服务，但是在某些设备中一些系统服务没被用到，如 Nexus 6 和 Nexus 9 没有用到 consumer_ir 服务(控制设备的红外线)，Nexus 9(Wifi 版本) 没有用到和电话相关的服务。另外一些系统服务虽然具有检查权限的函数声明，但是没有具体的函数实现，这导致了一些潜在的问题。例如 fingerprint 服务在内核没有硬件支持，*checkPermission* 作为其权限检查函数应该检查发起请求的应用是否有 *USE_FINGERPRINT* 和 *ENROLL_FINGERPRINT* 权限，但这个函数仅仅有一个声明却没有函数体。如 4.4.1 部分描述的那个漏洞，恶意应用不需要任何权限就可以伪造 *android.service.fingerprint.IFingerprintServiceReceiver* 这个接口发起请求。当攻击者发送一个请求给 fingerprint 服务去删除一个指纹 ID 时，由于 *nativeRemove* 函数仅仅返回一个错误代码，对应系统服务就会崩溃。因此，安卓系统应当减少不在系统中使用的系统服务，减少被攻击的可能。

攻击者在滥用一些系统服务时可能会困扰用户。例如 statusbar 服务用于管理设备屏幕上的状态栏，这个系统服务的一个函数可以显示最近使用过的应用列表，如果攻击者不断的去调用这个函数，会一直显示最近使用过的应用列表界面，使用户无法正常操作，而恶意的应用程序发起这种攻击不需要任何的权限。

4.5 漏洞利用攻击场景

在 4.4 小节中，我们分析了发现的漏洞，并简要描述了每个漏洞被利用后造成的攻击效果。本节中，我们将这些漏洞利用能够产生的攻击综合起来描述一个场景，能够更易看出漏洞的危害性。

一位普通安卓设备用户想听音乐，他想使用蓝牙耳机，因此打开蓝牙，系统崩溃重启(利用蓝牙服务中的漏洞)。他转而使用普通的耳机，当把耳机插入设备时，系统崩溃重启(利用 Audio 服务中的漏洞)。最终他放弃听音乐，想看一部电影。当他把设备屏幕翻转的时候，系统崩溃重启(利用 Windows 服务中的漏洞)。他想连接 Wifi 热点，阅读新闻，但是发现无法连接热点(利用 Wifi 服务中的漏洞)，而且当他想查看修改 Wifi 设置的时候，设置应用直接崩溃，无法使用。同时他发现照相无法使用，每当打开应用的时候，都会崩溃(利用 media.player 中的漏洞使得 media 进程崩溃)。最终设备不断崩溃重启，无法终止这种情况(利用 mount 服务中的漏洞)。

以上攻击场景中，恶意应用利用六个漏洞产生一系列的攻击效果最终导致安卓设备的完全不可用。这里值得说明的是，恶意应用必须预先安装在设备中，此外，该应用必须申请 *BOOT_COMPLETED* 权限以使得应用在设备重启的时候都会启动，从而发动攻击。

4.6 谷歌对漏洞的回应

我们把这些漏洞的详细细节报告给谷歌，谷歌的安全团队在分析了我们的报告后为这些问题创建了跟踪编号，分别为 ANDROID-20076875、ANDROID-20643294、ANDROID-21117978、ANDROID-215-23339、ANDROID-21585255 和 ANDROID-22489397。另外，

我们也因这些漏洞获得了 5 个 CVE 编号, 我们将会持续跟踪这些漏洞的状态。

5 讨论

本文工作是为了理解安卓系统服务中无效输入带来的安全威胁。该工作中最影响效率的是大部分前期工作需要手工完成。我们手工分析了安卓系统服务源码, 进而发现做了输入验证和不做输入验证的各个函数的比例, 最后我们开发了一个半自动化的系统服务模糊测试工具。它不断的向系统服务发送请求并记录系统日志, 虽然工具用脚本过滤掉不必要的日志信息, 但对应系统服务和工具产生的信息通过人工检测完成。未来工作可以放在通过静态分析工具分析源码, 用自动化的方法检查日志信息。

本文工作针对安卓 5.0.1 系统。不同版本安卓系统中相同系统服务可能具有不同数量不同种类的函数。例如在 2.0 版本中, wallpaper 服务提供 8 个函数, 而在 5.0.1 版本中提供了 12 个函数。因此 Buzzer 将测试每个系统服务的功能封装为一个模块, 便于针对以后安卓版本的扩展。目前 Buzzer 没有做到兼容多个版本, 这是由于 Buzzer 很大程度上依赖于 native 库, 为编译 Buzzer 需要特定的 NDK 版本, 不过其他版本也可以用同样的方法配置, 即 Buzzer 的设计不依赖于安卓系统版本, 实现兼容多个版本的 Buzzer 只需简单的工作即可。

针对本文工作中发现的漏洞, 我们认为最好的修复方法是在接口函数中添加白名单, 即接口函数在收到参数时, 应当确保参数的取值在一个限定的范围内, 任何不符合匹配条件的参数都应当被丢弃。另外, 对于那些过于复杂无法限制取值的参数, 至少应该丢弃那些明显是畸形数据的参数。此外, 在对安卓系统服务进行分析的过程中, 我们发现很多系统函数需要应用具有某些特殊权限, 但是这些特殊权限在安卓开发者文档中却没有记录, 而且第三方应用无法申请到这些权限, 有些权限就给我们造成了一些困扰, 因此我们认为安卓文档应该记录这些权限, 给予开发者更好的系统说明。

6 相关工作

6.1 输入验证

大多数的输入验证研究是在 Web 应用[10,30]上进行, 这些输入验证造成的问题包括跨站脚本攻击、SQL 注入等。通常客户端和服务端是两个分离的信息系统, 如果服务端被攻击后, 客户端访问会受到影响, 但攻击方不会有损失。但本文研究的输入验证类型问题存在一个安卓系统中, 如果系统服务被攻击, 恶意应用也会受到影响。

6.2 安卓和 Linux 的模糊测试

Buzzer 的设计思想源于一个 Linux 系统调用模糊测试工具——Trinity。Trinity 利用一些技巧将半智能化的参数传递给系统调用函数, 其背后的思想是减少无效测试的时间, 尽可能深入的到达测试代码片段, 把重点放在那些可能出现意外错误的测试点上。例如, Trinity 启动时会创建一个文件描述符列表, 当某个系统调用函数需要一个文件类型参数时, Trinity 会从中随机选取一个传给该系统调用函数。Buzzer 同样会传递半智能化的参数给接收对应请求的系统服务。我们通过人工方式检查所有系统服务函数的参数, 例如很多系统服务会接收一个整型的用户 ID 作为参数, 并在使用前会检测其是否有效, Buzzer 搜集系统中所有的用户 ID, 随机选取一个传递给这种类型的系统服务函数。另外, 很多应用程序会拒绝没有特殊权限的应用发起的请求, 因此 Buzzer 直接过滤这些函数。

针对安卓系统也有一些模糊测试工具, 如 intent fuzzer[7,28,35], 它传递一些空字段给应用程序暴漏出来的组件进行测试。Maji 等人[28]在这个简单的 fuzzer 上作了扩展, 他们根据安卓文档中的 intent 对象, 经验性的构造了一个半智能化的测试集去测试组件间通信的稳定性。DroidFuzzer 利用 *URI* 数据发起中间人攻击来发现 activity 的问题, 它利用正常数据作为种子, 产生一些畸形的数据注入到 intent 中。组件之间的通信是通过 binder 机制进行的, 之前的一些文献中提到的这类工作对 Buzzer 生成参数方式有一定帮助, 如 activity 系统服务会收到很多参数为 Intent 类型的请求。

6.3 安卓安全

安卓安全研究领域已经有了很多文献, 以下是一些研究热点实例。

提权攻击[14,15,18]利用安卓权限机制中的安全缺陷, 允许一个低权限的应用去访问一个需要更多权限才可以访问的组件。信息泄露攻击窃取了用户的隐私数据。重打包攻击[22,36,37]通过反编译安卓应用, 插入恶意的代码后再重新编译成新的应用程序。所有的这些例子聚焦在应用本身或应用之间的安全。

Wu 等人[32], Zhou 等人[38]和 Grace 等人[21]研究了安卓定制化系统的安全问题。Xing 等人[33]致力于研究安卓系统升级机制可能导致的提权问题。Li 等人[25]分析了流行的消息推送服务问题。他们发现这些服务极易出错, 允许一个未授权方去推送任意消息给

一个应用程序, 在本地和远程都可以实现。Bhoraskar 等人[13], Georgiev 等人[20]和 Shekhar 等人[31]研究了安卓应用中第三方组件和库的安全, 包括广告库和混合应用程序框架。

目前, 据我们了解, 现有的工作很少关注系统服务层, 尤其是系统服务中的输入验证类型漏洞, 本文研究了系统服务中潜在的安全问题, 发现了与之相关的输入验证类型漏洞。

7 结论

本文工作对安卓输入验证类型的漏洞进行了分析, 1)我们分析了系统服务的攻击面, 介绍了目前系统服务中的输入验证实现情况; 2)我们开发了一个输入验证漏洞扫描器, 通过向系统服务发送一些带有畸形参数的请求对其进行模糊测试。通过对安卓系统中 90 多个服务和 1900 多个函数进行综合的分析, 我们发现了 16 个系统服务中的漏洞。

参考文献

[1] Android aidl guide. http://developer.android.com/guide/components/aidl.html.

[2] Android init language. https://android.googlesource.com/platform/system/core/+/master/init/readme.txt.

[3] Android init.rc file. https://android.googlesource.com/platform/system/core/+/master/rootdir/init.rc.

[4] Android open source project. https://android.googlesource.com/.

[5] Android shipments in 2014. http://www.cnet.com/news/android-shipments-exceed-1-billion-for-first-time-in-2014/.

[6] Factory images for nexus devices. https://developers.google.com/android/nexus/images.

[7] Intent fuzzer. https://www.isecpartners.com/tools/mobile- security/intent-fuzzer.aspx.

[8] Symantec's threat report. http://know.symantec.com/LP=1123.

[9] Trinity - a linux system call fuzz tester. http://codemonkey.org.uk/projects/trinity/.

[10] M. A. Alkhalaf. "Automatic Detection and Repair of Input Validation and Sanitization Bugs. [Ph.D. dissertation], " *University of California, Santa Barbara*, 2014.

[11] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. " In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. pp. 259-269, 2014.

[12] A. Barth, C. Jackson, and J. C. Mitchell. "Robust defenses for cross-site request forgery. " In *Proceedings of the 15th ACM conference on Computer and communications security. (CCS'08).* pp. 75-88, 2008.

[13] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall. "Brahmastra: driving apps to test the security of third-party components." In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)*, pp. 1021-1036. 2014.

[14] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.R. Sadeghi. "Xmandroid: A new android evolution to mitigate privilege escalation attacks." *Technische Universit□at Darmstadt, Technical Report TR-2011-04*, 2011.

[15] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.R. Sadeghi, and B. Shastry. "Towards taming privilege-escalation attacks on android." In *19th Annual Network & Distributed System Security Symposium. (NDSS'12)*, 2012.

[16] G. Chen, H. Jin, D. Zou, B. B. Zhou, Z. Liang, W. Zheng, and X. Shi. "Safestack: automatically patching stack-based buffer overow vulnerabilities." *Dependable and Secure Computing, IEEE Transactions* on 10, no. 6 pp. 368-379, 2013.

[17] Q. A. Chen, Z. Qian, and Z. M. Mao. "Peeking into your app without actually seeing it: Ui state inference and novel android attacks." In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)*, pp. 1037-1052, 2014.

[18] L. Davi, A. Dmitrienko, A.R. Sadeghi, and M. Winandy. "Privilege escalation attacks on android." In *Information Security*. Springer Berlin Heidelberg, pp. 346-360. 2011.

[19] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. "Taintdroid: an information-ow tracking system for realtime privacy monitoring on smartphones." In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, (OSDI'10)*, pp. 1-6, 2010.

[20] M. Georgiev, S. Jana, and V. Shmatikov. "Breaking and fixing origin-based access control in hybrid web/mobile application frameworks." In *Proceeding of the Network and Distributed System Security Symposium*, (*NDSS'14)*, 2014.

[21] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. "Systematic detection of capability leaks in stock android smartphones." In *Proceeding of the Network and Distributed System Security Symposium, (NDSS'12)*, 2012.

[22] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. "Juxtapp: A scalable system for detecting code reuse among android applications." In *Detection of Intrusions and Malware, and Vulnerability Assessment, (DIMVA'13).* pp. 62-81, 2013.

[23] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center. "Scandal: Static analyzer for detecting privacy leaks in android applications." In *Proceedings of Mobile Security Technologies (MoST'12)*, 2012.

[24] Y. Kosuga, K. Kono, M. Hanaoka, M. Hishiyama, and Y. Takahama. "Sania: Syntactic and semantic analysis for automated testing against sql injection." In *Computer Security Applications Conference*, (*ACSAC'07)*, pp. 107-117 2007.

[25] T. Li, X. Zhou, L. Xing, Y. Lee, M. Naveed, X. Wang, and X. Han. Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. (CCS'14)*. pp. 978-989, 2014.

[26] M. T. Louw and V. Venkatakrishnan. "Blueprint: Robust prevention of cross-site scripting attacks for existing browsers." In *Security and Privacy. (S&P'09)* pp. 331-346, 2009.

[27] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. “Chex:statically vetting android apps for component hijacking vulnerabilities.” In *Proceedings of the 2012 ACM conference on Computer and communications security, (CCS’12)*, pp. 229-240, 2012.

[28] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer. “An empirical study of the robustness of inter-component communication in android.” In *Dependable Systems and Networks, (DSN’12)*. pp. 1-12, 2012.

[29] Y. Nadji, P. Saxena, and D. Song. “Document structure integrity: A robust basis for cross-site scripting defense.” In *Proceeding of the Network and Distributed System Security Symposium, (NDSS’09)*, 2009.

[30] T. Scholte, D. Balzarotti, and E. Kirda. “Have things changed now? an empirical study on input validation vulnerabilities in web applications.” *Computers & Security*, 31, no. 3, pp. 344-356, 2012.

[31] S. Shekhar, M. Dietz, and D. S. Wallach. “Adsplit: Separating smartphone advertising from applications.” In *Proceedings of the 21rd USENIX Security Symposium (USENIX Security’12)*, pp. 553-567, 2012.

[32] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. “The impact of vendor customizations on android security.” In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, (CCS’13)*. pp. 623-634, 2013.

[33] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang. “Upgrading your android, elevating my malware: Privilege escalation through mobile os updating.” In *Security and Privacy (S&P’14)*, pp. 393-408, 2014.

[34] L.-K. Yan and H. Yin. “Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis.” In *Proceedings of the 21rd USENIX Security Symposium (USENIX Security’12)*, pp. 569-584, 2012.

[35] H. Ye, S. Cheng, L. Zhang, and F. Jiang. “Droidfuzzer: Fuzzing the android apps with intent filter tag.” In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia, MoMM’13)*. pp. 6, 2013.

[36] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. “Viewdroid: Towards obfuscation-resilient mobile application repackaging detection.” In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks, (WiSec’14)*. pp. 25-36, 2014.

[37] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. “Detecting repackaged smartphone applications in third-party android marketplaces.” In *Proceedings of the second ACM conference on Data and Application Security and Privacy, (CODASPY’12)*. pp. 317-326, 2012.

[38] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. “The peril of fragmentation: Security hazards in android device driver customizations.” In *Security and Privacy (S&P’14)*. pp. 409-423, 2014.

[39] Y. Zhou and X. Jiang. “Dissecting android malware: Characterization and evolution.” In *Security and Privacy (S&P’12)*, pp. 95-109, 2012.

曹琛　于 2011 年在中国矿业大学信息安全专业获得学士学位。现在中国科学院大学信息安全专业攻读博士学位。研究领域为系统安全。研究兴趣包括：云计算安全、分布式系统安全。
Email: caochen11@mails.ucas.ac.cn

高能　于 2006 年在中国科学院研究生院获得博士学位。现为中国科学院信息工程研究所第三研究室副研究员，现任信息安全国家重点实验室副主任。研究领域为网络安全，系统安全。
Email: gaoneng@iie.ac.cn

向继　于 2009 年在中国科学院研究生院获得工学博士学位。现任中国科学院信息工程研究所第四工程部主任。研究领域为 Web 安全, Android 安全。
Email: xiangji@iie.ac.cn

刘鹏　于 1999 年在美国 George Mason University 获得博士学位。现为美国宾夕法尼亚州立大学信息科学与技术学院教授，信息安全实验室主任。研究领域为系统安全。
Email: pliu@ist.pst.edu