计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2018,27(10):112−120 [doi: 10.15888/j.cnki.csa.006596]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于能力依赖图的 SEAndroid 安全策略分析①

曹佳欣[1,2], 程　亮[2], 张　阳[2]

[1](中国科学院大学, 北京 100049)
[2](中国科学院 软件研究所 可信计算与信息保障实验室, 北京 100190)
通讯作者: 曹佳欣, E-mail: caojiaxin@tca.iscas.ac

摘　要: SEAndroid 作为 Android 系统安全机制的重要组成部分, 直接关系到系统的安全性. 在本文中, 我们提出一种基于能力依赖图的 SEAndroid 安全策略分析方法. 能力依赖图描述了实际 Android 系统中用户的能力迁移以及其 SEAndroid 子系统的访问控制配置. 我们首先对 SEAndroid 的具体实现进行分析, 收集安全策略和系统信息, 并进行逻辑建模. 然后, 我们依据 SEAndroid 的策略判定模式设计逻辑推导规则, 并以此利用逻辑编程的方式生成能力依赖图. 基于能力依赖图, 我们提取出可能的攻击路径和攻击模式. 我们对多个 AOSP 发布的不同 Android 版本的 SEAndroid 访问控制系统子进行了评估与分析. 我们发现随着 Android 版本的提升, 其 SEAndroid 安全策略也进行了更新, 新的 SEAndroid 对系统提供了更强的约束和保护. 此外, 我们在实验中发现了一种被黑客在实际攻击中使用到的攻击模式, 从而验证我们方法的有效性.

关键词: SEAndroid; 访问控制; 安全策略; 能力依赖图

引用格式: 曹佳欣,程亮,张阳.基于能力依赖图的 SEAndroid 安全策略分析.计算机系统应用,2018,27(10):112–120. http://www.c-s-a.org.cn/1003-3254/6596.html

Analysis of SEAndroid Policies Based on Capability Dependency Graph

CAO Jia-Xin[1,2], CHENG Liang[2], ZHANG Yang[2]

[1](University of Chinese Academy of Sciences, Beijing 100049, China)
[2](Trusted Computing and Information Assurance Laboratory, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

Abstract: As part of the Android security model, SEAndroid is critical to assure the security of operating systems. In this study, we propose an approach to analyze SEAndroid policies based on capability dependency graph. Capability dependency graph describes attacker's potential capabilities and the dependency relationships among these capabilities. It also describes the configuration of SEAndroid policies. We collect some security related system facts, and encode the collected data to Prolog predicates. We adopt logic programming to automatically compute a capability dependency graph with driving rules. We enumerate all the attack paths from initial nodes to goal nodes in the capability dependency graph, and categorize the attack paths into attack patterns. We apply our approach to analyze and compare some different versions of Android. We find that with the upgrade of the Android version, the SEAndroid security policy has also been updated. The new SEAndroid provides a stronger constraint and protection for the system, and a experimental attack pattern has been verified in the actual system.

Key words: SEAndroid; access control; security policy; capability dependency graph

① 基金项目: 国家自然科学基金 (61471344); 国家重点研发计划 (2017YFB0802902)
Foundation item: National Natural Science Foundation of China (61471344); National Key Research Program of China (2017YFB0802902)
收稿时间: 2018-03-13; 修改时间: 2018-04-03; 采用时间: 2018-04-17; csa 在线出版时间: 2018-09-28

1 引言

Android 操作系统是一个基于 Linux 内核的开放源代码移动操作系统, 它由 Google 成立的开放手持设备联盟 (Open Handset Alliance, OHA) 持续领导与开发. Android 现已成为全球第一大操作系统.

最初的 Android 依赖于其基于 Linux 的自主访问控制 (Discretionary Access Control, DAC) 机制来提供安全边界. 自主访问控制的核心观点是基于用户 ID 和组 ID 的概念, 用户之间保持了相对的隔离性, 文件和程序都有自己的所有者, 也即用户, 用户只有在获得了相应的授权, 才能对其它用户拥有的资源文件或者进程进行相关的操作和通信. 同样的组 ID 是将具有相关属性的用户组合在一起, 同时规定了相应组用户具有操作对应资源的权限. 但是自主访问控制机制存在显著的缺点, 比如有缺陷或恶意的应用会泄露敏感数据、无法限制以 root 权限运行的任何系统守护进程或 setuid 程序等.

为了降低恶意程序对系统带来的损害, 2012 年美国国家安全局 (National Security Agency, NSA) 推出系统安全强化套件 SEAndroid[1], 它是将原来 Linux 平台上的安全子系统 SELinux[2]移植到 Android 平台上. SEAndroid 安全机制从 Android 4.3 版本开始正式引入, 不过此时 SEAndroid 仅是运行在 Permissive 模式之下, 也就是说该机制并没有产生实质功能, 只是对相关操作进行记录. Android 4.4 版本打开了 Enforcing 模式, 但是仅对几个关键的域 (installed, netd, vold, zygote) 进行保护. 从 Android 5.0 版本开始, 所有的进程都运行在 Enforcing 模式下, SEAndroid 正式开始对系统安全起到了保护的作用[3].

SEAndroid 安全策略直接决定系统安全状态, 其配置不当将带来严重安全问题, 比如应用程序若被错误地过度授予了不必要的访问权限, 将导致权限提升 (CVE-2015-4640, CVE-2015-4641)[4,5]. 因此分析、检测 SEAndroid 安全策略中的安全漏洞是十分必要的.

目前有大量的 SELinux 访问控制策略的分析研究[6–11], 但这些研究所采用的方法没有将 SEAndroid 的特定方面考虑在内, 没有涉及 SEAndroid 的特定类型及策略结构, 因此不能直接使用这些方法对 SEAndroid 安全策略进行分析. 而目前 SEAndroid 安全策略的分析研究还比较少[12,13], 且现有的关于 SEAndroid 安全策略漏洞检测方法需要大量的手工操作, 要求用户有额外知识储备 (如域、功能测试), 这无疑加重了用户的负担.

有鉴于此, 在本论文中, 我们结合自主访问控制机制提出一种基于能力依赖图的 SEAndroid 安全策略分析方法. 首先, 我们对 SEAndroid 的实现方法进行分析, 同时与 SELinux 进行比较, 构造系统信息和安全策略的抽取工具, 并对抽取的信息进行形式化的描述, 以建立准确而精简的安全策略配置模型. 其次, 以安全策略配置模型为输入, 设计和实现自动化的分析推理引擎, 生成能力依赖图. 最后, 从能力依赖图中提取攻击路径. 本文主要贡献有以下几点:

(1) 我们提出了基于能力依赖图的 SEAndroid 安全策略分析方法. 能力依赖图不仅描述了每个动作需要的前提条件和执行该动作产生的新的能力, 还描述了攻击者攻陷系统的所有可能方法.

(2) 基于我们的方法实现了原型系统. 原型系统由三部分构成: 系统事实收集器、能力依赖图图生成器、攻击路径分析器.

(3) 我们使用原型系统对多个版本 Android 系统的 SEAndroid 安全策略进行了评估与分析. 我们一共发现了 8 种攻击模式, 其中远程攻击模式 2 种, 本地攻击模式 6 种. 通过对比分析, 我们发现随着 Android 版本的提升, SEAndroid 安全策略也进行了更新, 新的 SEAndroid 对系统提供了更强的约束和保护, 并且其中一种远程攻击模式已经在实际系统中得到了验证.

2 SEAndroid

SEAndroid 访问控制包括三种访问控制模型: 类型强制 (Type Enforcement, TE)[14]、角色访问控制 (Role-Based Access Control, RBAC)[15]、多层安全 (Multi-Level Security, MLS)[16]. 在 SEAndroid 中, 每一个主体和客体都有一个安全上下文, 格式描述如下: user: role: type: security_level. 其中, user 代表 SEAndroid 的用户, role 是针对 RBAC 的, type 针对 TE, security_level 针对 MLS. 由于在 SEAndroid 中, 系统只定义了一个用户 u 和两个 RBAC 角色 r 和 object_r, 因此 TE 是 SEAndroid 中最重要的访问控制模型, 本文也将 SEAndroid 中的 TE 模型部分作为我们的主要研究对象.

在 TE 中, 每一个主体 (进程) 和客体 (如文件、TCP 套接字) 都有一个类型 (type), 主体的类型也被成为域 (domain). 具有共性的类型将被归在一起构成一个称为属性 (attribute) 的集合, 比如 appdomain 属性就是一些应用程序相关域的集合. 客体根据性质可以划分为不同的类别 (class), 常见类别包括文件、目录、

套接字等. 系统为每个类别定义了一组权限集合(permission set).

TE 使用 allow 规则来控制主体对客体的访问, 使用 type_transition 规则来管理域转换和为新产生的客体指派类型. allow 规则格式为: allow domain type:class {permission sets}. type_transition 规则格式为: type_transition source_type target_type:class default_type.

Android 系统启动时, 所有的策略配置文件被打包编译成一个二进制文件 sepolicy, 系统将 sepolicy 加载进内核空间的 Linux 安全模块 (Linux Security Module, LSM). LSM 包含访问向量缓存 (Access Vector Cache) 和安全服务器 (Security Server). 当一个进程主体以某种权限访问客体时, SEAndroid 根据进程的域和客体的类型, 首先在 AVC 中查找是否存在该访问向量, 若存在则允许访问, 否则会在安全服务器中继续查找. 若安全服务器中存在该访问向量, 则允许访问并将访问向量写入 AVC, 否则将禁止访问.

3 系统概述

Android 系统访问控制机制是提供用户权限隔离的主要安全机制, 而大多数攻击的前提是权限提升攻击, 因此我们主要关注权限提升攻击. 在整个权限提升攻击过程中, 每一个攻击步骤攻击者都能获取额外的能力, 并为下一步攻击做准备. 我们的目标是分析 Android 的访问控制机制, 实现一个推理系统, 发现当前访问控制配置中潜在的权限提升攻击. 系统包含三部分: 系统事实收集器、能力依赖图生成器、攻击路径分析器, 如图 1 所示. 系统事实收集器收集系统状态信息和 SEAndroid 安全策略信息, 并将这些信息声明为逻辑事实. 图生成器以收集器收集的系统事实、访问控制机制、推导规则的逻辑声明为输入, 给定攻击者的初始能力, 输出能力依赖图. 我们选定 Prolog 作为图生成器的逻辑分析引擎. Prolog 是一种基于一阶谓词的逻辑性语言, 广泛应用于人工智能领域, 通常用来建造专家系统、智能知识库等. 由于 SEAndroid 策略定义了系统中主客体可允许行为的通用规则库, 主体行为在查询规则库并得到许可后方可执行, 而这都可用一阶谓词描述. 因此, Prolog 的描述性语法很适合声明访问控制策略相关信息并查询特定权限是否可以满足. 以权限提升作为目标, 路径分析器遍历能力依赖图得到攻击路径和攻击模式.

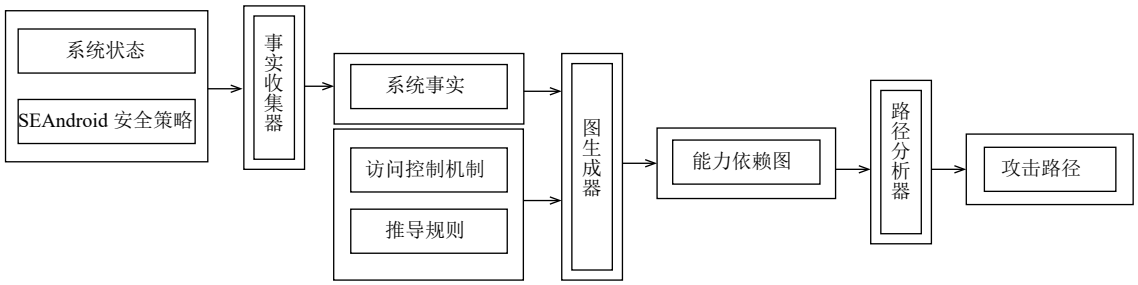

图 1 系统框架图

4 系统设计与实现

在本部分, 我们将分别对系统的事实收集器、图生成器、路径分析器的具体设计实现进行详细介绍.

4.1 事实收集器

事实收集器收集系统状态信息和 SEAndroid 安全策略配置信息, 并将其声明为 Prolog 事实.

一般与权限提升攻击相关的系统状态信息包括用户信息、文件信息、网络状态、进程信息和进程打开文件等. 需特别指出的是在 Android 系统中 UID 的含义不同于 Linux, UID 在 Linux 中用来唯一确定某个用户的身份, 而早期 Android 是单用户系统, 不需要支持多用户登陆. Android 为每一个应用程序分配一个 UID, 用 UID 对应用程序进行管理. Android 中的 UID 分为系统 UID 和应用程序 UID 两部分, 系统 UID 包括 root、WIFI、shell 等, 可从 AOSP 提供的源码文件 android_filesystem_config.h 中获取. 应用程序 UID 包括 calendar、email 等, 可以从 Android 系统的 packages.list 中获取.

SEAndroid 安全策略配置信息包括安全上下文和安全策略. 在 SEAndroid 安全机制中, 主体一般就是进程, 而客体一般就是文件. 故我们主要收集文件和进程的安全上下文. SEAndroid 的安全策略主要存贮在以

te 为后缀的文件中, 而所有 te 文件最终会被系统打包成一个二进制文件——sepolicy. 我们借助 SETools 将 sepolicy 解析成文本文件. 在所有安全策略中, 我们主要关注规则名称为 allow 的访问向量和 type_transition 类型规则.

我们将上述收集到的信息编码为 Prolog 事实, 格式如表 1 所示. 其中, *Owner*, *Group*, *Setuid*, *Setgid*, *Sticky*, *Uid*, *Pid*, *Port* 和 *Gid* 是数字. *Type*∈{dir, regular, ⋯}. *Protocol*∈{tcp, udp}. *Uper*, *Gper*, *Oper* 是用 0、1 代表 read、write、execute 权限的三元组. *Gids*, *OpList* 是编码为 Prolog 表格式的集合. *Path*, *Se_user*, *Se_role*, *Se_type*, *Username*, *Program*, *Domain*, *Filename*, *Class*, *Old_dom*, *New_dom*, *Attribute* 是字符串.

表 1 系统事实声明格式

Prolog 事实	含义
file_info(*Path*, *Type*, *Owner*, *Group*, *Uper*, *Gper*, *Oper*, *Setuid*, *Setgid*, *Sticky*, *Se_user*, *Se_role*, *Se_type*)	文件信息
user_info(*Username*, *Uid*, *Gids*)	用户信息
process_network(*Pid*, *Protocol*, *Port*)	网络端口信息
process_running(*Pid*, *Uid*, *Gid*, *Program*, *Se_user*, *Se_role*, *Domain*)	运行进程信息
process_reading(*Pid*, *Filename*)	进程打开文件
se_domain(*Domain*)	域信息
se_type(*Se_type*)	类型信息
dom_priv(*Domain*, *Se_type*, *Class*, *OpList*)	权限信息
se_typetrans(*Old_dom*, *New_dom*, *Se_type*)	类型转换
se_attribute(*Se_type*, *Attribute*)	属性信息

4.2 图生成器

能力依赖图生成器以系统事实、访问控制机制和推导规则为输入, 生成能力依赖图.

4.2.1 能力依赖图

能力依赖图描述了每个动作执行所需的前提条件和执行后生成的新的能力, 该图形象刻画了在 Android 访问控制机制限制下, 具有初始能力的攻击者可以获得的所有能力. 我们的 SEAndroid 安全策略分析就是基于能力依赖图的.

定义 1. 能力依赖图 (Capability Dependency Graph): 一个能力依赖图是一个有向图 $G=(C_a \cup C_o, A, E)$, 其中, C_a 是能力节点集合, C_o 是条件节点集合, A 是动作节点集合, E 是有向边集合, $E \subseteq ((C_a \cup C_o) \times A) \cup (A \times C_a)$.

能力依赖图的有向边有两种类型: 指向动作节点的边和指向能力节点的边. 有向边可以从能力节点和条件节点指向动作节点, 表示只有这些能力和条件前提都满足时该动作才能执行. 有向边也可以从动作节点指向能力节点, 表示该动作执行后产生的新能力.

如图 2 所示, dac_can_access(*Uid*, *Gid*, *FileName*, *FileType*, write) 和 se_can_access(*Domain*, *FileName*, write) 是条件节点, control_process_code(proc(*Uid*, *Gid*, *Domain*)) 和 control_file_data(*FileName*) 是能力节点, WriteFile(*FileName*) 是动作节点. 指向 WriteFile(*FileName*) 的三条边表示要对文件 *FileName* 进行写操作必须满足三个前提: (1) 控制了某个进程, 该进程安全上下文为 proc(*Uid*, *Gid*, *Domain*); (2) DAC 允许该安全上下文写文件 *FileName*; (3) SEAndroid 允许该安全上下文写文件 *FileName*. 指向 control_file_data(*FileName*) 的边表示当对文件 *FileName* 进行写操作后产生的新的能力.

4.2.2 访问控制机制逻辑声明

访问控制机制和系统状态信息一起构成了能力依赖图中的条件节点. 我们对自主访问控制和 SEAndroid 的类型强制访问控制机制进行逻辑声明. 权限提升攻击的攻击对象主要是文件和进程, 所以我们主要考虑与进程和文件相关的访问控制规则. 我们的研究目标在于发现 SEAndroid 访问控制策略的漏洞, 实际有些权限提升攻击确实不违反我们声明的 SEAndroid 访问控制规则, 例如利用内核的缓冲区溢出漏洞 (如 CVE-2017-13293[17]) 进行提权, 但这些攻击并不能通过修改 SEAndroid 安全策略来缓解, 这超出了我们的研究范围.

通常进程作为主体, 文件作为客体, 与之相关的权限包括读、写、执行. 由此, 以 dac 前缀代表自主访问控制, se 前缀代表 SEAndroid 的类型强制访问控制, 我们声明的格式如下所示:

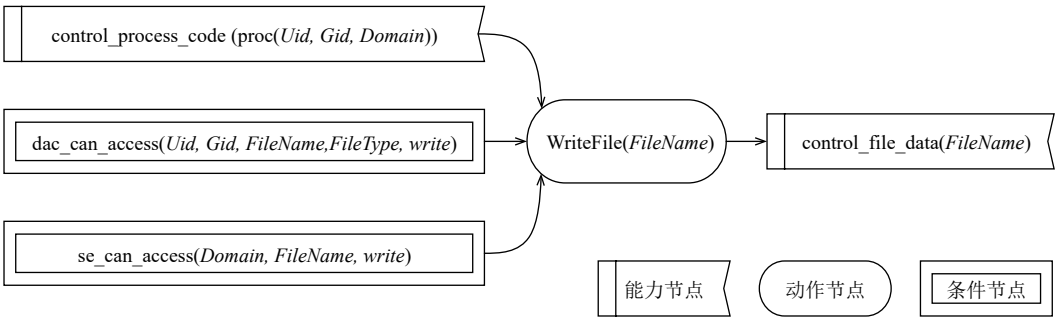

图 2 能力依赖图示例

dac_can_access(*Uid*, *Gid*, *FileName*, *Mode*): 检查自主访问控制下用户 id、组 id 为 *Uid*、*Gid* 的主体是否能对 *FileName* 客体进行 *Mode* 操作;

dac_execv(*OUid*, *OGid*, *NUid*, *NGid*, *Program*) : 检查自主访问控制下用户 id、组 id 为 *OUid*、*OGid* 的主体是否能通过执行 *Progrom* 程序克隆出一个用户 id、组 id 为 *NUid*、*NGid* 的新进程;

se_domain_privilege(domain(*Domain*), type(*Type*), class(*Class*), op(*Op*)): 检查类型强制访问控制下域为 *Domain* 的主体能否对类型为 *Type*、类别为 *Class* 的客体进行 *Op* 操作;

se_can_access(*Domain*, *FileName*, *Mode*): 检查类型强制访问控制下域为 *Domain* 的主体能否对文件 *FileName* 执行 *Mode* 操作;

se_execv(*Domain*, *NewDomain*, *Type*) : 检查类型强制访问控制下域为 *Domain* 的主体能否通过执行类型为 *Type* 的客体克隆出域为 *NewDomain* 的新进程;

privilege_enhancing(*Uid*, *Gid*, *Domain*, *NewUid*, *NewGid*, *NewDomain*): 检查产生新的进程后, 攻击者的能力是否有所提升. 检查的依据: 1) 原 *Uid*、*Gid* 不为 0(root) 或 1000(system) 且用户 id、组 id 发生改变; 2) 原 *Domain* 的属性不为 unconfineddomain 且域发生改变.

4.2.3 能力集合逻辑声明

由于进程和文件多被用来实现权限提升攻击, 我们关注与进程和文件相关的权限, 而攻击者获得的权限表述为对文件和进程具有的能力. 自主访问控制使用用户 id 和组 id 来标识主体, 类型强制访问控制使用域来标识进程, 因此我们使用 proc(*Uid*, *Gid*, *Domain*) 来描述进程的安全上下文. 针对 Android 的权限提升攻击中存在大量的远程攻击, 而远程攻击的前提是攻击者可以控制某个网络端口的输入. 由此, 我们声明的能力如下所示:

control_network_input(*Port*): 攻击者控制端口 *Port* 的输入, 并以此远程访问系统;

control_process_code(proc(*Uid*, *Gid*, *Domain*)): 攻击者控制一个用户 id、组 id 为 *Uid*、*Gid*, 域为 *Domain* 的进程;

control_file_data(*FileName*): 攻击者控制文件 *FileName* 的文件内容.

4.2.4 攻击动作与推导规则逻辑声明

为了实现权限提升, 攻击者尝试执行不同的攻击动作来获取新的能力. 我们分析权限提升漏洞的利用方式, 得到如下影响系统安全性的操作, 即攻击动作:

WriteFile(*FileName*): 攻击者写 *FileName* 文件;

Execute(*Program*): 攻击者执行 *Program* 程序;

CompromiseNetwork(*Pid*, *Port*):攻击者将恶意制作的数据包通过端口 *Port* 发送到进程号为 *Pid* 的程序来攻陷该进程;

CompromiseRead(*Pid*, *FileName*): 攻击者恶意篡改进程号为 *Pid* 的程序读取的文件 *FileName* 的内容来攻陷该进程;

WaitExecute(*Program*): 攻击者等待正在被某个进程执行的程序 *Program* 以相同的安全上下文再次被执行;

HopeExecute(*Program*): 攻击者希望特权用户执行程序 *Program*.

当一个攻击动作 action 执行时, 必须满足一定的前置条件 pre-conditions, 该动作执行后, 攻击者将获取新的能力. 我们声明推导规则来描述这种能力转移关系.推导规则被声明为形如 (pre-conditions, action, capability) 的三元组. 我们声明的推导规则如表 2 所示.

Android 访问控制子系统的一个重要功能是实现进程隔离, 即使某个脆弱程序被攻陷, 访问控制系统依然能够阻止权限提升攻击. 因此, 我们在声明推导规则

时引入”what-if”分析, 我们假设每个可执行程序都可能存在漏洞, 都可能被攻击者利用执行任意代码.

表 2 推导规则逻辑声明

Pre-conditions	Action	Capability
control_process_code (proc(*Uid*, *Gid*, *Domain*))dac_can_access(*Uid*, *Gid*, *FileName*, *FileType*, write)se_can_access(*Domain*, *FileName*, write)	WriteFile(*FileName*)	control_file_data(*FileName*)
user_info(_, *OldUid*, _)process_running(*OldPid*, *OldUid*, *OldGid*, _, *Old_SE_User*, *Old_SE_Role*, *OldDomain*)dac_can_execute(*OldUid*, *OldGid*, *Program*)se_can_execute(*OldDomain*, *Program*, *Type*)dac_execv(*OldUid*, *OldGid*, *Uid*, *Gid*, *Program*)se_execv(*OldDomain*, *Domain*, *Type*)privilege_enhancing(*OldUid*, *OldGid*, *OldDomain*, *Uid*, *Gid*, *Domain*)	Execute(*Program*)	control_process_code (proc(*Uid*, *Gid*, *Domain*))
control_network_input(*Port*), receiving_data(*Pid*, *Port*)	CompromiseNetwork(*Pid*, *Port*)	control_process_code(proc(*Uid*, *Gid*, *Domain*))
process_reading(*Pid*, *FileName*), control_file_data(*FileName*)process_running(*Pid*, *Uid*, *Gid*, _, _, _, *Domain*)	CompromiseRead(*Pid*, *FileName*)	control_process_code(proc(*Uid*, *Gid*, *Domain*))
process_running(*Pid*, *Uid*, *Gid*, *Program*, _, _, *Domain*)control_file_data(*Program*)	WaitExecute(*Program*)	control_process_code(proc(*Uid*, *Gid*, *Domain*))
is_executable(*Program*)control_file_data (*Program*)	HopeExecute(*Program*)	control_process_code(*)

4.2.5 能力依赖图生成

要生成能力依赖图, 还需声明攻击者的初始能力和攻击目标. 攻击者分为远程攻击和本地攻击两种类型. 远程攻击者主要通过网络远程访问 Android 系统, 其初始能力被声明为 control_network_input(*Port*), *Port* 是系统开放的任意有效的端口号. 本地攻击者拥有一普通账户, 其初始能力被声明为 control_process_code(*Process*), *Process* 代表攻击者可以运行的进程. 攻击者的目标是获取 root 或 system 权限, 声明格式为 control_process_code (proc(*Uid*, *Gid*, *Domain*)), Uid 为 0(root) 或 1000(system).

基于上述访问控制机制、能力集合、攻击动作集合、推导规则的逻辑声明, 我们按照算法 1 所示生成能力依赖图.

算法 1. 能力依赖图生成算法

1) 在当前条件和能力下依据推导规则搜索能够执行的攻击动作;
2) 得到能够执行的攻击动作后, 依据推导规则计算该攻击动作执行后得到的新能力;
3) 构造 (pre-conditions, action, new capability) 日志记录;
4) 将步骤 2 中得到的新能力加入当前能力中;
5) 重复步骤 1 直到不再生成新的日志记录;
6) 添加 pre-conditions 指向 action 的边和 action 指向 new capability 的边.

4.3 路径分析器

路径分析器以能力依赖图作为输入, 使用深度优先搜索算法 (DFS) 搜索所有从初始能力节点到目标节点的攻击路径, 如算法 2 所示. 算法首先遍历初始能力节点集合, 以每一个初始能力节点 v_i 作为入口参数, 调用 DFS, 将得到的路径集合 *Paths* 与攻击路径集合 *APS* 合并得到新的 *APS*. DFS 搜索过程中为减少不必要动作我们做了一些限制: (1) 攻击路径上不存在环路; (2) 攻击路径的长度不超过 10; (3) 攻击路径的中间节点不含目标节点. 我们选择 10 作为攻击路径长度的阈值是因为基于我们的实验, 我们发现 Android 系统中大部分攻击路径长度是 7 和 9, 如果减小阈值, 将损失大量攻击路径, 导致较高的漏报率; 增大阈值后. 新增的攻击路径数目只有几十条, 且这些新增的长攻击路径相比大量的短攻击路径, 对于攻击者来说工作量太大, 中间步骤多容易失败且容易被审计系统发现, 从而对攻击者而言缺乏吸引力. 而选择 10 作为攻击路径长度的阈值既可以满足性能需求, 也足够描述系统所承受的攻击面.

算法 2. 攻击路径生成

输入: *INS*: 初始能力节点集合
GNS: 目标节点集合
输出: *APS*: 攻击路径集合
Function ATTACK_PATHS_GENERATION(*INS*, *GNS*)
1. $APS \leftarrow \phi$
2. **for all** $v_i \in INS$ **do**
3. *path* ← null
4. DFS(v_i, *path*)
5. $APS \leftarrow APS \cup Paths$
6. **end for**
7. **return** *APS*
Function DFS(*v*, *path*)
1. **if** *v in path* **then** /*避免环路*/

2. **return**
3. **end if**
4. **if** *path.length*>10 **then**
5. **return**
6. **end if**
7. *path*.push_back(*v*)
8. **if** *v*∈*GNS* **then**
9. *Paths* ← *Paths* ∪ *path*
10. **else**
11. **for all** node v_i adjacent to *v* **do**
12. DFS(v_i, *path*)
13. **end for**
14. **end if**

能力依赖图中存在大量的攻击路径, 因此我们引入攻击模式. 通过去掉攻击路径中的参数, 每一条攻击路径可以被抽象成一种攻击模式, 而每种攻击模式代表一类使用相同攻击动作的攻击路径.

5 实验结果分析

针对权限提升攻击, 为检测不同版本的 Android 系统当前 SEAndroid 安全策略的安全漏洞, 我们利用我们的方法和工具对 Android 6.0 到 Android 7.1 这 3 个版本的 SEAndroid 安全策略进行评估与分析, 并将我们的实验结果与现有权限提升漏洞的利用方式进行比较以验证其有效性. 在本部分, 我们将介绍我们的实验环境和实验结果分析.

5.1 实验环境与步骤

在我们的实验中, 系统状态信息在测试的 Android 系统上收集, 其余部分在安装有 Ubuntu 16.04 的机器上完成, 机器配置为 Intel Core I7-6700 3.4 GHz, 8 GB 内存.

我们的实验步骤如下:

(1) 信息收集: 将相关程序和脚本拷贝进 Android 系统并执行得到临时文件, 从 Android 系统中拷贝出临时文件和 sepolicy, 执行相关脚本后得到系统中的访问控制元数据, 保存为 Prolog 谓词形式;

(2) Prolog 推理: 根据 Prolog 推理规则和第一步生成的 Prolog 文件, 在 XSB[18]逻辑推理引擎中执行相关查询, 进行逻辑推理, 记录推理轨迹, 生成能力依赖图;

(3) 攻击路径分析: 运行攻击路径分析程序, 以生成的能力依赖图作为输入, 生成图中的全部攻击路径, 并生成攻击路径抽象而成的攻击模式.

5.2 实验结果分析

我们使用 SETools[19]工具对多个 Android 版本的 sepolicy 二进制文件进行解析, 得到的类型、属性、allow 规则、type_transtion 规则的数目如表 3 所示. 从表 3 中, 我们可以发现, SEAndroid 策略的类型、属性、allow 规则、type_transtion 规则的数目随着 Android 系统版本提升而增加. 从 Android 6.0 到 Android 7.0, 类型数目增加了 23.2%, 属性增加了 21.7%, allow 规则增加了 37.3%, type_transition 规则增加了 30.6%. 从 Android 7.0 到 Android 7.1, 类型数目增加了 2.9%, 属性增加了 3.6%, allow 规则增加了 3.3%, type_transition 规则增加了 7.0%.

表 3 不同 Android 版本的 SEAndroid 策略

Android 版本	类型	属性	allow 规则	type_transition 规则
Android 6.0	504	23	4464	121
Android 7.0	621	28	6129	158
Android 7.1	639	29	6329	169

我们的实验一共生成了 8 种攻击模式, 如下所示:

P1：control_network_input→CompromiseNetwork
→control_process_code→Execute
→control_process_code

P2：control_network_input→CompromiseNetwork
→control_process_code→WriteFile
→control_file_data→WaitExecute
→control_process_code

P3：control_process_code→Execute
→control_process_code

P4：control_process_code→WriteFile
→control_file_data→HopeExecute
→control_process_code

P5：control_process_code→WriteFile
→control_file_data→CompromiseRead
→control_process_code

P6：control_process_code→WriteFile
→control_file_data→CompromiseRead
→control_process_code→Execute
→control_process_code

P7：control_process_code→WriteFile
→control_file_data→CompromiseRead
→control_process_code→WriteFile
→control_file_data→HopeExecute
→control_process_code

P8：control_process_code→WriteFile
→control_file_data→CompromiseRead

→control_process_code→WriteFile
→control_file_data→CompromiseRead
→control_process_code

其中, P1、P2 是远程攻击模式, P3 到 P8 是本地攻击模式. 各 Android 版本中每种攻击模式对应的攻击路径数目如表 4 所示.

表 4 不同 Android 版本的攻击模式

Android 版本	P1	P2	P3	P4	P5	P6	P7	P8	总计
Android 6.0	44	0	76	21	4	552	117	9	823
Android 7.0	52	208	36	0	6	292	3	0	597
Android 7.1	14	48	42	0	6	196	0	6	312

从表 4 中, 我们可以看出 Android 6.0 有 7 种攻击模式, 823 条攻击路径, Android 7.0 有 6 种攻击模式, 597 条攻击路径, Android 7.1 有 6 种攻击模式, 312 条攻击路径. Android 7.0 比 Android 6.0 多了一种远程攻击模式, 少了两种本地攻击模式, 整体攻击路径数量减少了 27.4%. Android 7.1 与 Android 7.0 的攻击模式很相似, 但攻击路径减少 47.7%.

结合表 3 和表 4, 我们可以得出结论: 随着 Android 版本的提升, SEAndroid 安全策略也进行了更新. 新的 Android 版本的 SEAndroid 安全策略拥有更多的类型和规则, 与之对应的是系统中攻击路径的减少, 这说明系统受到了更强的约束和保护. 而 CVE 关于 Android 系统的漏洞统计[20−22]也说明了这一点: Android 6.0 的漏洞数为 558, Android 7.0 的漏洞数为 492, Android 7.1 的漏洞数为 228.

值得一提的是, 在对实验得到的攻击模式进行验证时, 我们发现 P2 模式中包含对 SwiftKey 的利用, 具体方式为: 攻击者通过发送恶意制作的数据包攻陷 SwiftKey 进程, 并改写一个可执行文件, 当该可执行文件被再次执行时, 攻击者获得特权进程. 而这种权限提升方式, 在实际系统中已经得到了验证[23]: 攻击者首先利用编号为 CVE-2015-4640 漏洞, 通过 80 端口向输入法应用程序 SwiftKey 发送精心构造的更新包, 然后利用 CVE-2015-4641 文件系统遍历漏洞使用该恶意更新包内的可执行文件将系统文件进行替换, 当该文件被执行时, 得到 system 权限进程. 这充分说明了我们方法的有效性.

6 相关工作

目前, 有大量关于 SELinux 安全策略的研究: Jaeger 等人设计实现了 GOKYO[6,7]来分析安全策略中的各种冲突, 试图找到缺失或不正确的约束; Zanin 和 Mancini 定义了形式化模型 SELAC[8]来分析 SELinux 的安全策略配置; Kissinger 和 Hale 设计实现了 Lopol[9]来进行策略分析; Cheng 等人设计实现了 ACVAL[10]来评估和比较不同操作系统访问控制机制提供的保护质量; Han[11]等人提出一种基于能力依赖图和 MaxSAT 的访问控制安全策略配置自动化强化方法. 但这些都没有将 SEAndroid 的特定方面考虑在内.

Wang 等人设计实现了 EASEAndroid, EASEAndroid[12]提出了一种半监督式机器学习方法来改进 SEAndroid 安全策略. 因为 Android 的持续更新和不断有新的攻击出现, SEAndroid 安全策略需要持续改进. EASEAndroid 提取大规模审计日志中的相关信息, 利用此信息构建访问模式, 以 AOSP 的参考策略和自定义少数恶意访问模式作为初始知识库, 利用半监督的机器学习方法对访问模式进行聚类, 根据聚类结果, 将良性访问模式转换为 allow 规则, 恶意访问模式转换为 neverallow 规则. 但是获得这个数据量的审计日志是很困难的, 因为它需要收集来自数百万个 Android 设备的日志文件, 而这可能涉及用户隐私. 并且默认情况下, 审计日志只记录被策略拒绝的系统层的操作, 对太过宽松的策略, 恶意访问可能被允许而没有记录, 对应用层的操作更是没有涉及. 最后生成的访问策略的安全性也有待考证.

Wang 等人[13]实现了 SPOKE 来识别过于宽松的策略规则. SPOKE 从 Android 系统功能测试中自动收集用户层的功能踪迹和内核层的访问模式, 根据 pid、uid 和带时间戳的包名, 在访问模式与功能踪迹间建立关联, 并将结果导入数据库; 查询数据库, 将不能与数据库中访问模式匹配的策略规则识别为过于宽松的策略规则. SPOKE 的缺点是依赖于通常不完整的应用程序功能测试, 而且功能测试中也可能包含不恰当的访问.

7 结语

在本文中, 我们结合自主访问控制机制提出一种

基于能力依赖图的 SEAndroid 安全策略分析方法. 我们对 SEAndroid 访问控制机制的实现方法进行分析, 同时与 SELinux 进行比较, 构造系统信息和安全策略的抽取工具, 并对抽取的信息进行逻辑声明. 基于系统信息、访问控制规则、推导规则等的逻辑声明, 我们生成了能力依赖图, 通过深度优先搜索能力依赖图得到了攻击路径. 我们设计实现了原型系统, 并对多个版本 Android 系统的 SEAndroid 安全策略进行了评估与分析, 我们发现随着 Android 版本的提升, SEAndroid 安全策略也进行了更新, 对系统提供了更强的约束和保护. 值得一提的是我们在实验中得到了一种已在实际系统中得到验证的攻击模式. 在未来工作中, 我们将对更广泛的 Android 手机固件进行试验, 发现其中的攻击模式, 实施进一步的对比与验证, 并尝试对引起权限提升漏洞的安全策略进行自动化定位和修复.

参考文献

1 Smalley S, Craig R. Security enhanced (SE) Android: Bringing flexible MAC to Android. NDSS, 2013, 310: 20–38.

2 Loscocco P, Smalley S. Integrating flexible support for security policies into the Linux operating system. Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference. Boston, MA, USA. 2001. 29–42.

3 Security-Enhanced Linux in Android. https://source.android.com/security/selinux.

4 CVE-2015-4640. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-4640.

5 CVE-2015-4641. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-4641.

6 Jaeger T, Zhang XL, Edwards A. Policy management using access control spaces. ACM Transactions on Information and System Security, 2003, 6(3): 327–364. [doi: 10.1145/937527]

7 Jaeger T, Sailer R, Zhang X L. Analyzing integrity protection in the SELinux example policy. Proceedings of the 12th Conference on USENIX Security Symposium. Volume 12. Washington, DC, USA. 2003. 5.

8 Zanin G, Mancini LV. Towards a formal model for security policies specification and validation in the selinux system. Proceedings of the Ninth ACM Symposium on Access Control Models and Technologies. Yorktown Heights, NY, USA, 2004: 136–145.

9 Kissinger A, Hale JC. Lopol: A deductive database approach to policy analysis and rewriting. Proceedings of the Second Annual Security Enhanced Linux Symposium. Baltimore, MD, USA. 2006.

10 Cheng L, Zhang Y, Han ZH, *et al*. Evaluating and comparing the quality of access control in different operating systems. Computers & Security, 2014, 47: 26–40.

11 Han ZH, Cheng L, Zhang Y, *et al*. Operating system security policy hardening via capability dependency graphs. Lopez J, Wu YD. Information Security Practice and Experience. Cham: Springer, 2015. 3–17.

12 Wang RW, Enck W, Reeves DS, *et al*. EASEAndroid: Automatic policy analysis and refinement for security enhanced Android via large-scale semi-supervised learning. Proceedings of the 24th USENIX Conference on Security Symposium. Washington, DC, USA. 2015. 351–366.

13 Wang RW, Azab AM, Enck W, *et al*. SPOKE: Scalable knowledge collection and attack surface analysis of access control policy for security enhanced Android. Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security. Abu Dhabi, United Arab Emirates. 2017. 612–624.

14 Badger L, Sterne DF, Sherman DL, *et al*. A domain and type enforcement UNIX prototype. Proceedings of the 5th Conference on USENIX UNIX Security Symposium. Volume 5. Salt Lake City, UT, USA. 1995. 12. 47–83.

15 Sandhu RS, Coyne EJ, Feinstein HL, *et al*. Role-based access control models. Computer, 1996, 29(2): 38–47. [doi: 10.1109/2.485845]

16 Sandhu RS. Lattice-based access control models. Computer, 1993, 26(11): 9–19. [doi: 10.1109/2.241422]

17 CVE-2017-13293 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-13293. [2017-08-23].

18 Sagonas K, Swift T, Warren DS, *et al*. The XSB system version 3.0-volume 1: Programmer's manual, Volume 2: libraries. Interfaces, and Packages. 2006.

19 SETools.https://github.com/TresysTechnology/setools.[2017-11-10].

20 Google Android version 6.0: Security vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-19997/version_id-187788/Google-Android-6.0.html.

21 Google Android version 7.0: Security vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-19997/version_id-201744/Google-Android-7.0.html.

22 Google Android version 7.1.0: Security vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-19997/version_id-204495/Google-Android-7.1.0.html.

23 Remote Code Execution as System User on Samsung Phones. https://www.nowsecure.com/blog/2015/06/16/remote-code-execution-as-system-user-on-samsung-phones/.