基于功能代码片段的 Java 后门检测方法

刘奇旭[1,2], 王柏柱[1,2], 胡恩泽[1,2], 刘井强[1,2], 刘潮歌[1,2]

[1] 中国科学院信息工程研究所, 北京 中国 100093
[2] 中国科学院大学网络空间安全学院, 北京 中国 100049

摘要 随着软件供应链污染的兴起, Java 开源组件的安全性正面临着越来越严峻的挑战, 近年来也出现了若干起因 Java 开源组件被植入后门而导致大规模的软件污染的安全事件。为了更好地检测 Java 开源组件和 Java 程序的安全性, 本文在大量分析 Java 后门样本的基础上, 构建了 Java 后门的检测模型作为理论基础; 在统计分析实际后门常用 Java API 的基础上, 归纳了一系列适用于检测 Java 后门的规则; 提出了基于功能代码片段的后门分析方法, 并且结合自底向上的数据流分析方法, 实现了首款面向 Java 源码的后门检测系统 JCAT(Java Code Analysis Tool)。以阿里供应链大赛提供的 119 个样本验证 JCAT 的检测能力, 取得了准确率 90.22%的良好效果, 并将漏报率和误报率分别控制在较低水平。

关键词 Java 后门检测; 静态检测技术; 数据流分析
中图法分类号 TP393.0 **DOI 号** 10.19363/J.cnki.cn10-1380/tn.2019.09.04

Java Backdoor Detection Based on Function Code Gadgets

LIU Qixu[1,2], WANG Baizhu[1,2], HU Enze[1,2], LIU Jingqiang[1,2], LIU Chaoge[1,2]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China

Abstract With the rise of software supply chain pollutions, the security of Java open source components is increasingly challenged. In recent years, there have also been some large-scale software pollution incidents caused by Java open source components being implanted backdoors. In order to detect the backdoors in the Java open source components and Java programs more effectively, in this paper, we first build Java backdoor detection models based on analyzing a large number of Java backdoor samples. Next, we summarize a series of rules for detecting Java backdoors on basis of statistics of common Java APIs in backdoors, propose a backdoor analysis method based on *function code gadget*. Combined with the bottom-up data flow analysis method, we develop the first backdoor detection system JCAT(Java Code Analysis Tool). We evaluate the JCAT's detection ability with 119 samples provided by the Ali Supply Chain Competition, and the detection rate reaches 90.22%. Moreover the false positive rate and false negative rate are also controlled at a relatively low level.

Key words Java backdoor detection; static detection technology; data-flow analysis

1 引言

如今开源文化盛行, 从小型信息系统到大型分布式系统都离不开开源组件的支持, 基于开源组件开发的业务系统也遍布各行各业。互联网上丰富的开源软件资源极大地降低了开发成本, 提高了开发效率, "开源共享"不仅是一种互联网精神, 也已成为诸多软件公司赖以生存的生态环境。Forrester 公司在 2017 年发布一份调研报告[1]中曾指出: "如今开发人员以开源组件为基础, 只需使用 10%~20%的新代码就可以创建出他们自己的应用"。

开源组件给开发人员带来便利的同时, 也带来了新的安全威胁——软件供应链攻击[2]: 通过网络攻击、内部威胁等方法在知名开源组件植入后门, 甚至构造带有后门的开源组件并发布, 一旦开发人员使用了上述带有后门的开源组件, 就会将后门引入到二次开发的软件或系统, 给最终交付的用户带来威胁。

Java 开源组件的供应链安全问题尤其值得关注。Java 语言[3]具有简单易用、功能强大和跨平台等特点,

通讯作者: 刘潮歌, 博士, 助理研究员, Email:liuchaoge@iie.ac.cn。
本论文获得国家重点研发计划（No.2016YFB0801604），中国科学院青年创新促进会，中国科学院战略先导 C 类（No.XDC02040100, No.XDC02030200, No.XDC02020200）课题资助；获得中国科学院网络测评技术重点实验室和网络安全防护技术北京市重点实验室资助。
收稿日期: 2019-05-30; 修改日期: 2019-08-15; 定稿日期: 2019-08-20

一直深受开发人员青睐, 因而 Java 平台的开源组件尤其丰富。有调查表明, 2017 年 Maven 仓库[4]中共享的 Java 第三方开源组件数量的增长率达到了 102%[5]。然而, 互联网上大量开源的 Java 组件并不安全, 可能存在严重的安全隐患。

2019 年 4 月, 腾讯安全披露的一起影响广泛的 Java 供应链安全事件[6]就极具典型性: 攻击者在某开源 SDK 之中植入后门, 导致超过 1000 款移动应用被污染, 影响了上千万用户设备。攻击者所开发的后门允许加载任意的远程 Jar 文件, 除了案例中实现的刷广告曝光量和点击量来牟取暴利, 还可以随时更新新的危害更大的 Jar 文件, 这使得被感染设备一直处于严重的安全威胁之中。因此, 在应用 Java 开源组件之前, 对其进行必要的后门检查就显得至关重要。

本文是 Java 源码安全检测方面的研究, 旨在构建相应的 Java 后门检测模型, 并在此基础上设计实现一套自动化检测工具, 既可用于发现第三方开源 Java 组件中的后门, 也可用于检测第一方开发的 Java 项目的安全性。本文的主要贡献如下:

1) 分析了大量 Java 后门并构建了相应的后门检测模型;

2) 提出了基于功能代码片段的分析方法, 并结合数据流分析技术实现了一种 Java 后门检测方法;

3) 基于所提出的检测模型和方法, 实现了 Java 后门自动化检测系统 JCAT(Java Code Analysis Tool)。对 119 个 Jar 样本的实际检测表明, JCAT 系统能够有效发现 Java 开源组件中的后门, 准确率达到 90.22%。

2 相关工作

后门检测历来是软件安全研究的重点, 当前的研究成果主要集中在后门定义和后门检测两个方面。相关的研究成果已汇总于表 1。

1) 后门定义(Backdoor Definition)。2000 年康奈尔大学的 Zhang 等人[7]第一次给出了一种非正式的后门定义, 认为后门是一种被秘密引入系统, 并帮助攻击者非授权访问该系统的机制; 2010 年 Veracode 公司的 Wysopal 等人[8]也类似地认为: 后门是绕过身份验证或其他安全控制以访问计算机系统或该系统上的数据的方法。并且 Wysopal 等还提出了后门的三种分类, 包括系统后门、危害加密算法的加密后门以及应用程序后门。但两者的定义均存在一定的缺陷: Zhang 等人首先给出了后门的一种非正式定义, 但其定义较为广泛, 未深入地针对后门进行分析; Wysopal 等在此基础上给出了后门的三种分类, 但其给出的后门分类并不能有助于单一后门的分析, 即未明确表达出后门的组成。

2018 年伯明翰大学的 Sam 等人[9]在前人工作的基础上, 将后门定义为系统中包含的有意构造, 该构造通过促进对其他特权功能或信息的访问来破坏其预期的安全性。同时, 为更好地检测后门, Sam 等还将后门拆分为输入源、触发器、有效载荷和特权状态 4 类组件。Sam 的研究工作对后门进行了细致的拆分, 并给出任了宽泛的适用于任意编程语言的后门定义。

Sam 对软件后门的定义未区分编程语言, 因而该定义不能很好地支撑某一特定语言的后门检测工作, 因此本文在 Sam 的研究工作基础上, 针对 Java 语言提出了更为细致的后门模型定义, 其中对后门模型定义中的攻击载荷做了具体的分类, 以及明确了在不同攻击载荷下所达到的特权状态。

2) 后门检测技术。因为后门通常由一种或多种程序语言编写而成, 所以针对各种程序语言的分析技术同样适用于后门的检测。大多数的后门检测研究工作均应用了程序分析的方法。从程序分析技术的分类角度入手, 后门检测技术主要分为静态检测技术和动态检测技术, 前者主要使用静态程序分析方法分析源码层面的语义信息, 而后者主要应用动态程序分析技术分析程序运行的过程或结果。此外, 机器学习算法被用于提升各种不同的后门检测方法。

静态检测技术直接通过从源代码或编译后的文件中提取程序的语义信息来检测潜在的后门, 而无需运行程序。2003 年 Wisconsin 大学的 Christodorescu 等人[10]开发了 SAFE 框架来发现恶意代码, 该框架实现了几种简单的静态程序分析技术用于推测变量的值内容, 以此来检测恶意代码(不论该恶意代码是否被混淆处理)。Wysopal 等人[11]使用了静态程序分析技术对二进制可执行文件进行后门检测。他们对多种类型的恶意行为的代码进行了分析, 包括特定认证、隐藏的命令执行、信息泄漏、rootkit 行为、反调试和时间炸弹。针对 Java 的应用程序, 2005 年斯坦福大学的 Livshits 等人[11]使用了上下文敏感的指针分析方法检测并定位 Java 源码中的多种安全漏洞。2012 年北京邮电大学的王一岚等[12]提出基于数据流分析和缺陷模式匹配的方法来检测 Java 语言中的后门。

与静态检测技术相对应的是动态检测技术, 其利用程序运行过程中的动态信息来分析其行为和特性。通常动态检测技术需要模拟真实或可运行的沙

盒环境[13]来运行待检测程序, 记录并分析其恶意行为[14]或其 API 调用序列[19]来检测后门。但由于后门行为的预测非常困难, 动态检测技术很难发现某些隐蔽的后门行为。

除此之外, 机器学习算法也被广泛地应用于后门检测, 但一般需要静态或动态检测技术的配合。2015 年印度 Trivandrum 工程学院的 Shijo 等[20]提出了多种技术相融合的后门检测方法。在静态检测和动态检测的基础上, 提取经过分析后的二进制代码和动态的后门行为作为特征向量, 然后利用机器学习算法训练分类模型。但该方法存在着所有动态检测技术共有的缺陷, 即需要预先创建可运行的环境。Java 开源组件往往为某一软件或程序的一部分, 无法创建通用的运行环境来观察其行为特征, 因此 Shijo 等提出的方法并不适用于检测开源组件中的后门。

为了更好地检测基于 Java 语言开发的软件或开源组件中的后门, 本文在分析大量 Java 后门的基础上构建了 Java 后门检测模型, 并且利用静态检测技术构建了相关语义模型, 提出了基于功能代码片段的检测方法, 设计并实现了自动化 Java 恶意后门检测系统 JCAT(Java Code Analysis Tool)。

表 1 后门检测相关研究工作总结

Table 1 Summary of researches on backdoor detection

研究方向	研究进展	研究团队	存在问题
后门定义	提出后门是一种被秘密引入系统, 并帮助攻击者非授权访问该系统的机制	康奈尔大学[7]	不明确的后门组成和分类
	提出后门是绕过身份验证或其他安全控制以访问计算机系统或该系统上包含的数据的方法, 并且提出了后门的三种分类	Veracode, Inc.[8]	缺乏后门的组成分析
	提出后门是系统中包含的有意构造, 该构造通过促进对其他特权功能或信息的访问来破坏其预期的安全性, 并且提出了后门的 4 种组成	雷恩大学[9]	针对特定语言, 拆分后的后门组成仍需进一步细化分类
后门静态检测技术	使用静态程序分析技术对二进制可执行文件进行后门检测, 并分析了多种类别的后门	Veracode, Inc.[8]	针对二进制文件, 不适用于面向对象的语言
	开发了 SAFE 框架来发现相关恶意代码的模式, 该框架实现了几种简单的静态程序分析技术用于推测变量的值内容, 以此来检测恶意代码	Wisconsin 大学[10]	针对二进制文件, 不适用于面向对象的语言
	使用上下文敏感的指向分析方法检测 Java 源码中的多种安全漏洞	斯坦福大学[11]	关注点在于安全漏洞发现
	提出基于数据流分析与缺陷模式匹配方法检测 Java 恶意后门	北京邮电大学[12]	未归纳出后门检测模型, 缺乏硬编码类型的数据流分析
	模拟真实或可运行沙盒环境执行恶意后门的程序	国立台湾科技大学[13]	
后门动态检测技术	通过分析恶意行为检测恶意后门 通过分析数据泄露行为的恶意软件检测方法 通过行为监控和特征分析检测恶意木马 通过改进的 HMM 算法区分恶意的程序行为 通过记录并分析 API 调用序列来检测恶意后门	丹麦奥尔堡大学[15] 软件工程国家重点实验室(武汉大学)[16] 北京大学[17] 海军指挥学院[18] 韩国大学[19]	动态检测条件苛刻, 无法构建一通用环境检测 Java 组件, 并且后门行为预测很难, 难以发现隐蔽的后门行为
	结合动静态技术分析可执行文件, 利用机器学习进行分类	印度 Trivandrum 工程学院[20]	需预先创建可运行环境, 不适用开源组件的后门检测

3 Java 后门检测模型

本节将从输入源、触发机制、攻击载荷和特权状态这四个方面描述 Java 后门。

根据 John E. Hopcroft 等[21]的理论, 使用一个软件程序来检测另一个软件程序是否存在缺陷是不可判定问题, 因为不存在一个通用的算法可以判断任何给定的程序能否终止。但是检测软件中是否存在某种特定漏洞却是可行的, 例如检测某个危险函数的调用。

后门的表现形式多种多样, 为了能更全面清晰地描述后门, 本节基于有限状态向量机(Finite-state machine, FSM)的思想来描述某个 Java 函数 F 中的后门 θ。

3.1 后门模型

假设某个给定函数 F 存在有限个变量节点 $[v_1,v_2,v_3,\cdots,v_n]$，且每个变量节点都存在独立的状态取值集合 $S(v_i)$。由于该有限个变量节点在不同时刻的状态不同，本文将函数 F 在 t 时刻的状态 F_t，以全部变量在该时刻的状态值所构成的集合表示，即 $F_t=\{s_1,s_2,\cdots,s_n \mid s_1\in S(v_1),s_2\in S(v_2),\cdots,s_n\in S(v_n)\}$。

根据上述定义，判定函数 F 中是否存在后门 θ 的问题，可以转化为：对于某个 $i\in I$ 和 $e\in E$，能够找到一个恰当的 Σ，使 $\delta(i,\Sigma)=e$ 成立并且能够执行某个攻击载荷 p。即，后门 θ 可以描述为：

$$\theta=(N,i,e,\delta,\Sigma,p),i\in N且e\in N$$

N 表示函数 F 的全部状态集合，即 F_t 的全部取值空间。I 表示函数 F 全部可能的初始状态所构成的集合，也称之为函数 F 的输入状态合集，显然有 $I\subseteq N$；E 表示函数 F 中全部的后门触发状态所构成的集合，显然有 $E\subseteq N$。一般来讲，当函数 F 处于某个 $e\in E$ 状态时，后门被触发，函数 F 也将相应获得原本不应具备的某种特权，因此后文也将这种 $e\in E$ 的状态称为“特权状态”；δ 表示函数 F 的状态转移函数，有 $\delta(F_n,c)=F_m,F_n\in N且F_m\in N$，其中 c 表示 δ 的转移条件；Σ 表示后门触发条件，特别地，当 $c=\sum$ 时，有 $\delta(F_n,\sum)=e,F_n\in N且e\in E$；$p$ 表示后门能够加载的攻击载荷，是后门之于攻击者价值的体现。

3.2 输入源

从控制流的角度来看，Java 程序可视为多叉树，仅当满足一定条件时，程序才会沿某条路径向下执行。同理，若攻击者想触发某个后门，也需要满足相应的条件。触发某条路径执行的直接条件，既可以由程序计算产生，也可以从程序内部或外部直接输入，但从根本上讲，前者是在后者的基础上衍生而来的。

因此，本文只关注程序内部或外部的直接输入，并把这些数据定义为程序的输入源。输入源的类型可以有很多种，为了简化处理 Java 中的输入源，本文将其抽象为 4 类：文件类(包括本地文件和外部文件)、硬编码字符串类、系统信息类以及 Socket 类。其中，硬编码字符串类为程序内部输入源，其他三类为程序外部输入源。Java 中各类输入源及其示例如表 2 所示。

3.3 触发机制

从理想的后门工作机制来看，仅当后门触发条件满足时，才会载入特定的攻击载荷以达到某种特权状态，如图 1 所示。根据触发条件 Σ 的可见性，本文将触发机制分为显式和隐式两种。后文将分别给出两种触发机制的代码实例，以及相应的有限状态向量机描述。

表 2 Java 输入源及示例

Table 2 Java input sources and samples

类名	输入源类型
File	文件类
URL/URI	文件类
FileWriter/FileReader	文件类
Paths	文件类
String/StringBuffer	硬编码字符串类
byte[]	硬编码字符串类
System	系统信息类
Socket	Socket 类
ServerSocket	Socket 类

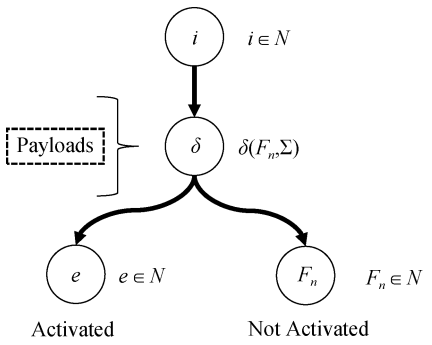


图 1 后门触发流程图

Figure 1 Backdoor trigger flowchart

3.3.1 显式触发机制

后门的显式触发机制是指触发条件硬编码于源代码中的情况。如图 2 所示，后门的触发条件为“程序当前所运行的系统环境为 Windows 系统”，当且仅当该条件满足时，后门被触发并加载后续的攻击载荷，使程序处于某个特权状态 $e\in E$。此处的后门触发条件是以字符串的形式硬编码于源码中，对代码维护人员来说是可见的，因此称之为“显式触发机制”。

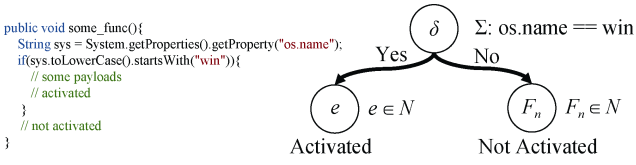


图 2 显式触发机制实例

Figure 2 Example of explicit trigger mechanism

3.3.2 隐式触发机制

同显式触发机制相对应的, 后门的隐式触发机制是指触发条件隐含于输入源的情况。这是隐蔽的、对抗性的后门触发机制, 因为若只分析程序源码的表面含义, 难以找到后门风险点, 后门是否触发、怎样触发, 完全取决于攻击者使用后门时的输入。更糟糕的是, 这一输入点可能是攻击者与正常用户共享的, 只不过攻击者触发后门的输入更加特殊。此类后门触发机制虽然隐蔽, 但也有一定的特征: 通常需要配合特定的函数使用, 并且后门的触发必须依赖于特定的输入源。

图 3 所示的实例代码描述了一个典型的隐式触发机制: SpelExpressionParser 类是 Spring 框架[22]中 SpEL 表达式的解析类, 它可以在软件运行过程中动态查询和操作对象图。示例代码中的输入 *input* 未作严格的安全处理, 这将导致程序接受任意的 SpEL 表达式, 包括执行软件开发者所不希望看到的危险系统命令。因此, 可以将高危的 SpEL 表达式视为后门的隐式触发条件。一旦恶意 SpEL 表达式被执行, 程序将获得特权状态, 其状态也将产生相应的变化。

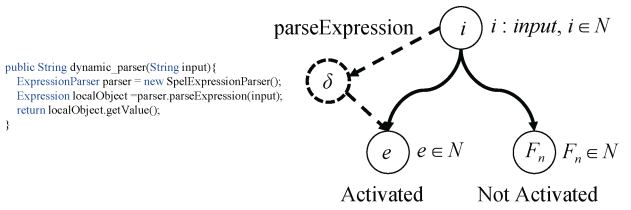


图 3 隐式触发机制实例

Figure 3 Example of implicit trigger mechanism

显式触发机制和隐式触发机制只建立了最简单的后门触发模型, 现实中的情况会更加复杂, 往往是两种触发机制的融合使用。例如先判断程序是否运行在 Windows 系统, 再调用 SpelExpressionParser 类解析 SpEL 表达式。

3.4 攻击载荷

攻击载荷是后门的关键部分, 它直接决定了后门能够造成的危害。类似于触发机制的分类, 本节根据攻击载荷在源码中的可见性, 将其分为显式攻击载荷和隐式攻击载荷两类。此外, 本节还提出功能代码片段的思想, 用于更加细化地判断攻击载荷的危害。

3.4.1 功能代码片段

定义 1.功能代码片段(Function Code Gadget)。功能代码片段由一行或包含多行代码的代码块组成, 目的是完成某个完整独立的功能逻辑。

功能代码片段可以有单个或多个输入源, 但有且仅有单一输出, 该输出的类型既可以是状态输出也可以是数据输出。一段完整的攻击载荷一般可以划分为多个功能代码片段, 各功能代码片段之间通过输入输出产生逻辑关联。攻击载荷的最后一个功能代码片段将输出一个特权状态, 本文将其称为后置功能代码片段; 攻击载荷中其他功能代码片段仅输出数据, 本文将其统称为前置功能代码片段。

在大量分析 Java 后门的基础上, 本文将构造后门相关的功能代码片段划分为 5 类, 即数据传输、代码执行、命令执行、文件操作以及辅助函数。每个攻击载荷可以由多个功能不同的功能代码片段组成, 也可以由多个功能相同的功能代码片段组成。下文将详细解释各类功能代码片段:

1) 数据传输功能代码片段。是一类具备同程序外部交互数据的代码片段, 交互内容包括从某个网络地址下载数据或将本地数据发送到某个网络地址。从数据流向来看, 前者从程序外部获取数据传输到本地, 后者将数据从本地传输到程序外部。通常, Java 语言中 Socket 类、URL 相关类或 HTTP 操作类可以完成此类数据交互。

2) 代码执行功能代码片段。是一类具备动态执行程序代码的功能代码片段。Java 语言支持解析执行多种程序语言, 包括 Python 语言、Ruby 语言、Groovy 脚本语言、EL 表达式、SpEL 表达式以及 JeXL 表达式等。以 Python 语言为例, Java 语言中使用 PythonInterpreter 类就可以动态执行 Python 语言。除此之外, Java 语言的类加载器机制[23]和反射机制[24]也可以使程序具备间接动态执行 Java 代码的能力。后门常利用 Java 语言的这种特性动态加载执行远程 Jar 或 Class 文件, 赋予攻击者动态调整攻击载荷以完成不同攻击意图的能力。

3) 命令执行功能代码片段。是一类具备执行系统命令的代码片段。Java 语言中执行系统命令主要依赖 ProccessBuilder 类和 Runtime 类。在程序中执行系统命令一般认为是危险行为, 但正常的 Java 应用程序确实有执行系统命令的需求, 因此区分程序正常的系统命令执行动作和后门恶意的系统命令执行动作是关键。

4) 文件操作功能代码片段。是一类具备操作本地文件能力的功能代码片段。污染本地文件一直是软件后门中常见的行为, 动作方式也多种多样, 包括读写系统敏感文件、污染包依赖管理源以及替换常用系统命令文件等等。

5) 辅助函数功能代码片段。是一类具备解析字符串或其他操作的代码片段。现实案例中, 为逃避后

门检测, 后门关键代码通常要经过混淆处理, 例如使用 BASE64 编码。此类后门将包含解析编码后的字符串的功能代码片段。此外, 辅助函数功能代码片段还包括一些不常见的特殊功能, 如调用 Webcam 类[25]打开电脑摄像头或调用 AudioSystem 类[26]录音等。

为了更准确地描述各类功能代码片段, 本文枚举了典型的 Java 类, 将其分别与各功能代码片段对应, 并形成表 3。图 4 示例了某个后门的功能代码片段分析, BASE64Decoder 是前置的辅助函数功能代码片段, DefiningClassLoader 是后置的代码执行功能代码片段, 两者结合使得硬编码字符串被动态解析执行。

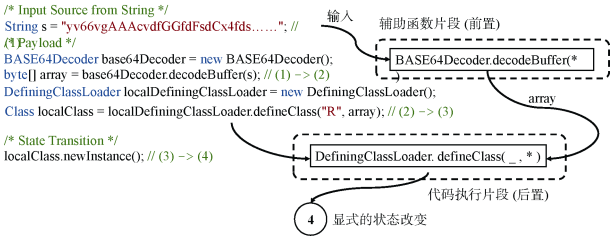


图 4　功能代码片段实例

Figure 4　Example of function code gadget

表 3　Java 功能代码片段示例

Table 3　Examples of Java function code gadget

功能代码片段类型	片段组成类名	片段组成类函数
数据传输功能代码片段	HttpURLConnection(*) URL	OutputStream getOutputStream() InputStream getInputStream()
	Socket(*,*) String|int|InetAddress	OutputStream getOutputStream() InputStream getInputStream()
	InetAddress()	InetAddress getByName(*) String InetAddress getAllByName(*) String InetAddress getByAddress(*) byte[] InetAddress getByAddress(*,*) String|byte[]
	ServerSocket(*) int	Socket accept()
代码执行功能代码片段	DefiningClassLoader() DefiningClassLoader(*) ClassLoader Method(*) ClassLoader	Class defineClass(_,*) byte[] Class loadClass(*) String Object invoke(*,*) Object,Object[]
	Class	Class forName(*) String Object newInstance() Method getDeclaredMethod(_,*) Class[] Method getMethod(_,*) Class[]
	URLClassLoader(*) URL[] URLClassLoader(*,_) URL[]	Class defineClass(_,*) byte[] Class loadClass(*) String
	SimpleCompiler()	void cook(_,*) StringReader|String void cookFile(*,_) File|String void cookFile(*) File|String ClassLoader getClassLoader()
	Expression(*,*,*) Object|String|Object[] Expression(_,*,*,*)Object|String|Object[]	void execute() Object getValue()
	SpelExpressionParser()	SpelExpression parseRaw(*) String
命令执行功能代码片段	Runtime()	Runtime getRuntime() Process exec(*) String|String[] Process exec(*,_) String|String[] Process exec(*,_,_) String|String[]
	ChannelExec()	void setCommand(*) String|byte[] void connect()
	ProcessBuilder(*) List<String>|String[]	Process start()
文件操作功能代码片段	BufferedWriter(*) Writer BufferedWriter(*,_) Writer	void write(*) int void write(*,_,_) char[]|String
	File(*) File|String|URI|StringBuffer File(*,_) File|String|URI|StringBuffer	boolean exists() boolean delete() File[] listFiles() String[] list() String getName() File createTempFile(*,_,_) String

续表

功能代码片段类型	片段组成类名	片段组成类函数
文件操作功能代码片段	RandomAccessFile(*,_) String|File	int read(*) byte[] void readFully(*) byte[] String readLine() String readUTF() void write(*) byte[] void writeBytes(*) String void writeChars(*) String void writeUFT(*) String
辅助函数功能代码片段	Base64.Encoder Base64.Decoder Webcam()	byte[] decode(*) byte[]|String byte[] encode(*) byte[] String encodeToString(*) byte[] Webcam getDefault() boolean open() BufferedImage getImage()

3.4.2 显式攻击载荷

显式攻击载荷是指攻击载荷硬编码于源码的情况, 此时攻击载荷对任何人都是可见、可知的, 攻击载荷执行后程序所处的特权状态也是可以预测的。图 5 示例了一个典型的显式攻击载荷: 攻击载荷以本地文件作为输入源, 由文件操作功能代码片段污染 authorized_keys 文件, 整个后门动作都显式地硬编码在源码中。显式攻击载荷执行后程序所处的特权状态是明确的, 称之为“显式特权状态”。

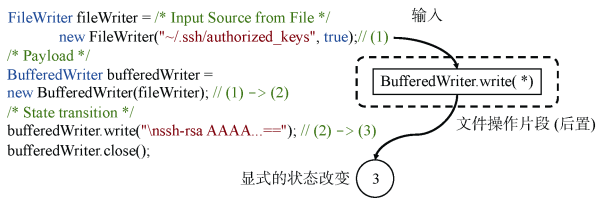


图 5 显式攻击载荷实例

Figure 5 Example of explicit payload

3.4.3 隐式攻击载荷

隐式攻击载荷是指动态加载代码的攻击载荷, 此时部分攻击载荷对任何人都不可见、不可知。不同于显式攻击载荷, 此类攻击载荷所能造成的威胁不可预测, 取决于攻击者的即时动作。

图 6 示例了一个典型的隐式攻击载荷: 后门利用 Weblogic XMLDecoder 类构造了一个反序列化漏洞, 具备执行任意代码的能力。但是分析这段后门代码, 仅能获知后门可执行任意攻击载荷, 而无法预知攻击者将具体使用什么样的攻击载荷, 程序将达到何种特权状态。隐式攻击载荷执行后程序所处的特权状态是不明确的, 称之为“隐式特权状态”。

3.5 特权状态

一旦攻击者触发后门并执行攻击载荷, 程序状态也将随之改变, 此时程序便拥有了预期之外的权限(如读写系统文件、执行任意代码等), 处于特权状态。根据攻击载荷状态的不同, 特权状态可分为显式特权状态和隐式特权状态。从控制流图(Control Flow Graph, CFG)角度看, 显式特权状态是一个可达的节点, 即在源码中能具体找到发生状态变化的函数或类, 也可明确程序拥有何种特权; 隐式特权状态在 CFG 中是一个不可达的节点, 因为此时程序所拥有的特权与攻击载荷相关, 而攻击载荷又是非确定的。图 7 左侧以有限状态向量机表示了图 5 所示的攻击载荷导致的程序状态迁移过程; 图 7 右侧则以有限状态向量机表示了图 6 所示的攻击载荷导致的程序状态迁移过程; 。

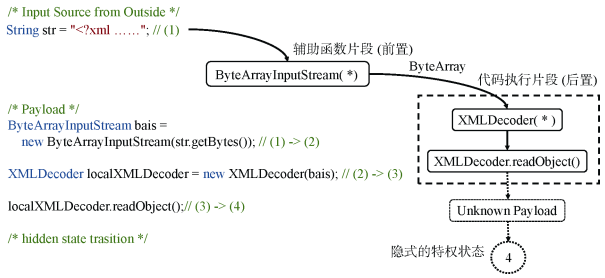


图 6 隐式攻击载荷实例

Figure 6 Example of implicit payload

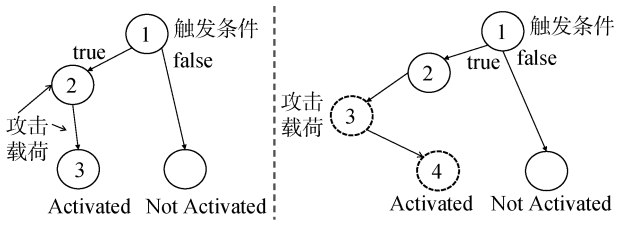


图 7 攻击载荷作用下的程序状态迁移

Figure 7 Program state migration caused by payload

4 系统设计与实现

本章的目标是面向 Java 语言, 设计一个可动态扩展规则的自动化后门检测系统——JCAT(Java Code Analysis Tool, Java 代码分析系统)。系统以 Jar 文件为输入, 以后门定位和后门可疑分值为输出。本章首先讨论后门的规则化描述, 再详细介绍 JCAT 系统的实现。

4.1 规则化后门描述

规则化后门描述(Regularized Backdoor Description, RBD)由后门规则和后门可疑分值计算两部分内容组成, 前者将已知的后门转化为可用于后门检测的规则描述, 后者则量化给出后门的危险程度。

4.1.1 后门规则

后门规则主要包括 Sink 函数规则、输入源规则和攻击载荷规则, 本节将详述以上 3 类规则。

1) Sink 函数规则。本文将后门中可能用到的关键函数称为 Sink 函数。在 Java 代码中调用此类函数通常被视作潜在的危险行为, 如调用 Runtime 类中用于执行任意命令的 exec 函数。因此, 检测程序源代码中的 Sink 函数是发现后门威胁的关键, 这种基于启发式思想的检测方法可以有效减少代码文件的分析工作量从而提高检测系统的效率。在检测系统实现中, 使用正则表达式的形式描述 Sink 函数, 以命令执行类 Runtime 类为例, 其 Sink 规则为"\bexec\s*\(", 枚举出所有包含 exec 函数的 Java 文件。

2) 攻击载荷规则。攻击载荷规则由前置功能代码片段、后置功能代码片段和关键字(后文称为 Token)流组成。前置功能代码片段和后置功能代码片段将形成一个描述后门功能逻辑的代码片段链。后门检测过程中, 将从后置功能代码片段开始经过前置功能代码片段向上递归查找输入源。此时, 攻击载荷规则将用于比对待检测的各功能代码片段是否与已知的后门功能代码片段链相吻合。功能代码片段的描述由变量类型、调用函数及其调用函数的参数组成。Token 流将用于描述某一后门以出现顺序排列的关键字集合, 即由连续的调用函数或类的关键字出现顺序组成的 Token 集合。

3) 输入源规则。输入源规则描述了后门的输入源信息, 包括类型、敏感字符、构造函数参数内容等等。输入源规则的类型指代 3.2 小节中输入源的 4 种类型, 包括硬编码字符串类、文件类、Socket 类以及系统信息类。针对不同类型的输入源, 分别检查其内容的敏感字符或其构造函数的参数内容的敏感字符。例如硬编码字符串类需要直接检查其内容是否包含规则中的相关敏感字符(表示为_sensitive 字段); 而 Socket 类则需要检查构造函数的参数内容是否包含敏感字符(表示为_params 字段)。除此之外, 输入源规则包含相应的分值用于后文的计分规则。

4.1.2 后门可疑分值计算

后门的功能代码片段链比对完成后, 为了更好地区分不同后门的严重程度, 本文提出了如下公式的计分规则, 使得危害程度越高的后门其可疑分值越高。公式中 M 表示最终判定的后门可疑分值, 由各单个规则子分值 G_i 累加得到。

$$M=\sum_{i}^{n}G_i$$

其中, $G_i=\lambda\times(g'+g_1''+g_2''+\cdots+g_j'')$, $\lambda\in[0,1]$

1) 单规则子分值 G_i。单规则子分值主要有两部分组成, 包括概率系数和输入源规则总分值。首先, 每一条输入源规则有一代表该输入源的基础分值 base_score, 公式中以 g' 表示, 其取值范围为正整数集。除此之外, 输入源规则的_sensitive 字段和_params 字段中包含了多条敏感字符规则, 每一条敏感字符规则中包含额外的分值 e_score, 在公式中以 g'' 表示, 其取值范围为正整数集。在检测过程中, 如果待检测片段中的输入源命中了某一输入源规则, 并且该输入源的内容又命中了该输入源规则的_sensitive 或_params 字段中的若干条敏感字符规则, 那么其输入源规则部分的分值为基础分值与额外分值之和。其次, 每一条规则存在一概率系数 λ(取值范围为 0 到 1), 该系数代表当前规则的可靠程度。

最后, 单规则子分值 G_i 为输入源规则部分的总分值与概率系数的乘积。

2) 多规则总分值 M。在检测过程中, 单个 Java 后门可能触发多条检测规则, 这意味着该后门存在多个单规则子分值 G_i。为了更好地区分后门的危害程度, 本文将多个单规则子分值求和得到多规则总分值 M。若后门实现的功能越多, 则其可能触发的后门规则越多, 最终其可疑分值越高。

3) 实例分析。为了更好地描述上述计分规则, 以图 4 中的实例代码为例, 生成了如图 8 的后门规则文件。如果后门命中了敏感字符 ssh-rsa 并且字符长度大于 50, 则其额外分值为 4, 单规则子分值为基础分值与额外分值之和, 共 6 分。当前规则的概率系数 λ 为 1, 表示可靠程度为 100%, 则最后的单规则子分值 G_i 为 6。

图 8　规则化后门描述实例

Figure 8　Example of RBD

除此之外，如果该后门实现了多种恶意功能，最终的多规则总分值 M 还需累加其他规则的子分值。因为图 4 的实例代码仅实现了任意代码执行的功能，所以其最后的总分值 M 等同于 G_i 为 6。

4.2　JACT 系统实现

Java 后门检测系统 JCAT 由预处理子系统、中间层生成子系统和检测子系统组成(如图 9)。预处理子系统的任务是反编译 Jar 文件，输出包括 Sink 点的 Java 文件；中间层生成子系统的任务是解析 Java 源码，并输出其中的功能代码片段；检测子系统通过规则比对的方式，识别疑似后门的功能代码片段序列，并输出最终的 Jar 文件后门检测结果。

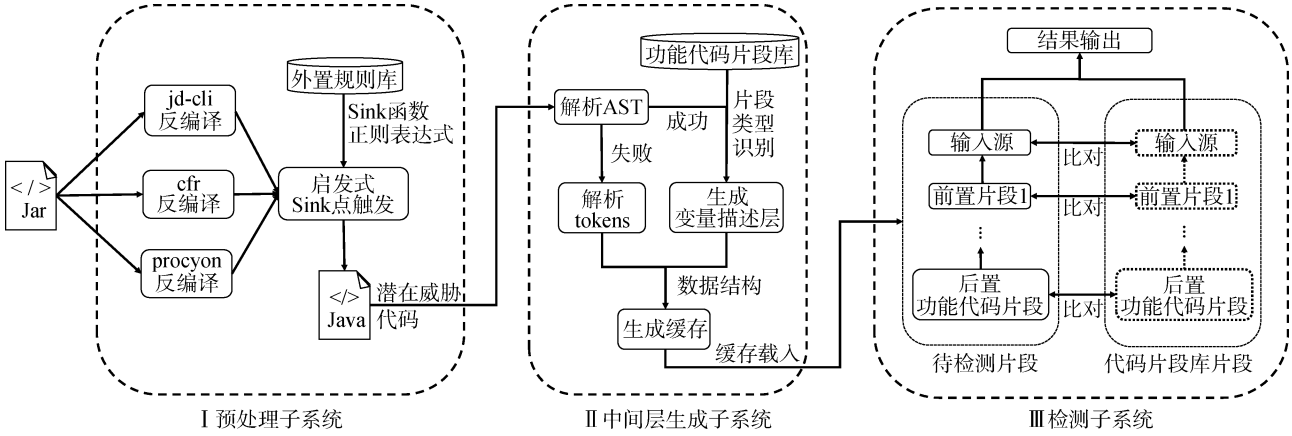


图 9　JCAT 系统构架图

Figure 9　JCAT System Architecture

4.2.1　预处理子系统

预处理子系统以待检测的 Jar 文件为输入，输出可疑的 Java 源文件，完成两项功能：第一，接受输入的待检测 Jar 文件，并将其反编译为多个 Java 源文件；第二，载入 Sink 函数描述规则，识别疑似的 Sink 函数并定位其所在的 Java 文件。与上述功能相对应的是反编译模块和启发式 Sink 函数触发模块。

反编译模块是将 Jar 文件反编译为 Java 源代码，反编译效果在很大程度上决定了后续检测的效果。反编译的效果高度依赖于反编译工具的能力，因此 JCAT 系统并联使用 cfr[27]、jd-cli[28]和 procyon-decompiler[29]三款知名的反编译工具，各自独立生成一份 Java 源代码。

启发式 Sink 函数触发模块的功能是应用 JCAT 系统集成的 52 条 Sink 函数触发规则，快速定位 Java 文件中的 Sink 函数。被检测出含有 Sink 函数的 Java 文件进入后续子系统做进一步分析，其他的文件则被排除嫌疑。启发式[30]是一种依据有限的知识在有限的搜索空间内快速找到问题解决方案的技术，适合在一定规则的配合下做模糊检测。经过该模块的预处理，可以排除掉相当一部分正常的 Java 文件，中间层生成子系统仅需处理可疑的 Java 文件，从而大幅提高系统检测效率。

4.2.2　中间层生成子系统

中间层生成子系统以含有 Sink 点的高危 Java 源文件作为输入，根据已定位的 Sink 点，分析包含 Sink 点的函数，输出该函数的功能代码片段集合及其 Token 序列，具体包括两项功能：第一，将静态的函数源码解析为方便内存载入的数据结构；第二，将相应的数据结构序列化存储，供检测子系统多次载入分析。与上述功能相对应的是语法解析模块和缓存模块。

语法解析模块的功能是解析 Java 文件，生成一个适合的数据结构来描述其代码逻辑。该模块对 Java 文件的处理分为两个层次，抽象语法树(Abstract Syntax Tree, AST)层和变量描述层。

中间层生成子系统调用第三方库 Javalang[31]自动化生成抽象语法树。但是，由于预处理子系统使用

的三款反编译工具并不能百分百地还原出没有语法错误的源码, 一旦源码存在语法错误, Javalang 将无法正确解析形成 AST, 这将会影响后续检测工作。为了解决这个问题, 本文归纳了出现语法错误的情况: 一是包含 Sink 点的函数出错, 二是不包含 Sink 点的位置出错。后一种错误并不影响对包含 Sink 点函数的进一步分析, 因此可以直接忽略。而针对第一种错误, 本文引入了词法流分析, 将出错函数解析成关键字(Token)流, 再做下一步处理。除此之外, 纯 AST 结构不易分析, 因此还需变量描述层对生成的 AST 结构做进一步处理。

定义 2. 变量描述层。变量描述层由若干条变量描述组成, 每条变量描述包含变量类型、变量内容、关联函数集合、所属功能代码片段类型等信息。

关联函数集合和所属功能代码片段类型是变量描述中最重要的内容。关联函数集合描述了某一变量所调用的函数集合和所调用函数的参数集合。该参数集合由不同的变量构成, 并且这些变量同当前的变量产生了关联关系。因此, 变量的关联函数集合还能用于搜索不同变量之间的关联关系。所属功能代码片段类型表明该变量属于前置功能代码片段还是后置功能代码片段。对于前置功能代码片段, 还需判断该功能代码片段是否存在对输入源的操作。如果存在, 则另外将其记录为输入源, 用作后续检测的结束节点。

语法解析模块完成上述动作后, 每一个含有 Sink 点的函数都将转化成相应的变量描述数据结构或词法流数据结构。为了满足持续分析的需求, 中间层生成子系统调用 Python 的第三方 Pickle 库, 将上述数据结构序列化后存入缓存文件, 供检测子系统多次载入分析。

4.2.3 检测子系统

检测子系统以中间层生成子系统缓存的数据结构为输入, 结合相应的规则检测后门, 最终输出后门判定结果, 该结果以分值的形式量化给出。检测子系统的核心是片段序列比对算法。

检测子系统首先从中间层生成子系统生成的缓存文件中读取缓存的变量描述数据结构或词法流数据结构, 还原其中的功能代码片段序列, 作为片段序列比对算法的输入。同时, 检测子系统还将加载后门检测规则作为输入, 其中后门检测规则为若干条已知后门片段序列。

片段序列比对算法采用自底向上的数据流分析方法, 实现了同功能代码片段库比对的功能。利用中间层生成子系统生成的数据结构, 实际算法的实现仅需判断功能代码片段类型即可。因为变量描述层已经根据功能代码片段库, 完成了片段类型的识别, 即已对待检测片段进行了比对。

算法的第一步是检索变量描述层的后置功能代码片段, 即包含 Sink 函数调用的变量描述。其次, 根据后置功能代码片段的调用函数的参数变量, 用递归的方式(*check* 函数)自底向上查找最终的输入源, 并在查找过程中计算可疑分值。算法 1 给出了这部分功能的伪代码描述:

算法 1. 片断序列比对算法.

输入: 待检测功能代码片断序列、检测规则(后门功能代码片断序列)

输出: 可疑分值

```
1: FUNCTION check(变量集合,规则)
2:   FOR 变量 IN 变量集合 DO
3:     IF 变量片段类型 MATCH 输入源 THEN
4:       RETURN 可疑分值
5:     ELSE IF 变量片段类型 MATCH 前置 THEN
6:       FOR 变量函数 IN 变量调用函数集合 DO
7:         check(变量函数参数集合,规则)
8:       END FOR
9:     END IF
10: END FOR
11: RETURN 0
12:
13: FOR 变量 IN 待检测变量集合 DO
14:   IF 变量片段类型 MATCH 后置 THEN
15:     FOR 变量函数 IN 变量调用函数集合 DO
16:       IF 变量函数名 MATCH sink 函数名 AND 变量函数参数类型 MATCH sink 函数参数类型 THEN
17:         RETRUN check(变量函数参数集合, 规则)
18:       END IF
19:     END FOR
20:   END IF
21: END FOR
```

除此之外, 当中间层生成子系统生成变量空间失败时, 检测子系统将依靠词法流分析的方式来判断是否存在恶意后门。我们利用规则文件中预定义的 Token 流匹配待检测的 Token 流。如果完全匹配, 则认为该函数内部存在恶意后门。

5 实验与分析

本章通过 Java 后门样本对 JCAT 系统作详细评估, 将从准确率、漏报率和误报率三个方面评估 JCAT 系统的检测能力, 从时间开销随代码规模的变化关系评估 JCAT 系统的检测性能。此外, 本章还对大量 Java 后门做了技术性统计分析, 讨论了 JCAT 系统的缺陷并提出下一步改进方向。

5.1 实验数据与环境

实验采用的数据集来自 2018 年“攻守道”阿里供应链安全大赛[32]预赛提供的 119 个 Java 组件样本(119 个 Jar 文件)。每个样本以某一常见的 Java 开源组件为基础, 随机嵌入了一个或多个后门。并且每个样本都经过了混淆处理, 无法通过与原开源组件对比得到后门位置。

参与实验的样本的代码量超过 10000 行的样本数高达 73.94%(这里代码量指代纯 Java 代码), 代码量小于 1000 行的样本数据则不足 4%, 由此可见实验工作量很大。图 10 给出了样本代码量的统计。

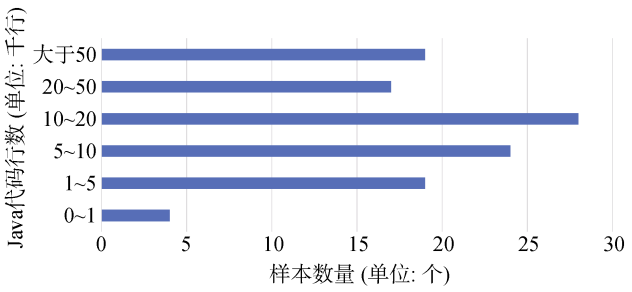


图 10 样本代码量统计

Figure 10 Statistics of code quantity in samples

结合赛后公布的解题数据, 经人工分析确认: 119 个 Jar 样本中共有 133 个后门, 其中部分样本含有多个后门。后文将以上述数据作为基础事实, 以后门准确率、漏报率、误报率评估系统的 JCAT 系统的检测效果。

JCAT 系统的测试环境为两台 PC 主机, 配置均为双核处理器和 2G 内存, 操作系统分别为 Ubuntu 16.04 和 Windows7。JCAT 系统运行过程中无其他消耗资源的程序运行。

5.2 检测能力分析

全部待检测的 119 个 Jar 样本中共含有 133 个后门。以全部 119 个样本作为 JCAT 系统的输入, 成功检测出 120 个后门; 有 13 个后门未检出, 形成漏报; 有 6 处将正常代码误判为后门, 形成误报。JCAT 系统的后门准确率达到 90.22%, 漏报率 9.8%, 误报率 4.5%。由此可见, JCAT 系统在检测 Java 后门方面表现突出, 不仅具有较高的准确率, 还将漏报率和误报率控制在合理水平。

JCAT 系统是首个直接针对 Java 后门检测的工具, 尚无同类型工具可对比。但是, 本文仍尝试将 JCAT 系统与若干款知名的 Java 代码静态审计工具做比较。参与实验的工具包括开源工具 FindBugs[33]和 SpotBugs[34], 商业工具 fortify[35]和 360 代码安全卫士[36]等。由于上述工具的主要关注点为 Java 代码规范和漏洞发现, 它们对 119 个样本的后门准确率为 0。由此可见, 本文工作对于提升 Java 源码的安全性意义重大。

此外, JCAT 系统还经历了 2018 年“攻守道”阿里供应链安全大赛决赛阶段的考验, 在对 50 个未知决赛样本检测中, JCAT 系统检出了其中 98%的后门, 再次验证了 JCAT 系统在 Java 后门检测方面的能力。

5.3 系统性能分析

系统性能也评估 JCAT 系统能力的重要方面。考虑到在实际 Java 项目开发场景中, 项目的源码量可能较庞大并且会快速增长, 因此需要从算法设计和系统实现两方面严格控制 JCAT 系统的检测速度, 不能因为代码量的增加而呈现指数增长的情况。

为此, 模拟代码量增长过程, 以 20 个样本为初始条件, 每次增加 20 个样本, 分别测试 JCAT 系统的检测时间开销。为排除偶然因素干扰, 在每个样本量级上重复实验 10 次, 并计算平均值。实验结果如图 11 所示, 可见 JCAT 系统的检测时间开销随样本规模增加呈线性增长, 即 JCAT 系统很好地控制了时间增长规模。

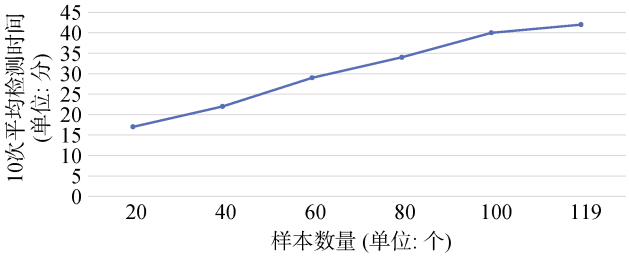


图 11 检测时间统计

Figure 11 Statistics of detection time costs

5.4 后门代码片段分析

结合 JCAT 的检测结果, 经人工确认: 119 个 Jar 样本中的 133 个后门分布于 147 个后门代码片段之中。对这 147 个后门代码片段的统计分析表明, 数据传输、代码执行、文件操作是最主要的后门功能(具体统计结果如图 12 所示)。

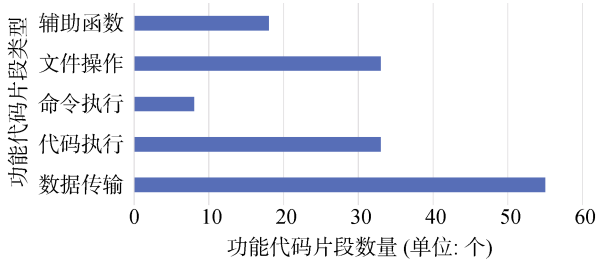


图 12　样本功能代码片段统计

Figure 12　Statistics of function code gadgets in samples

后门的实现离不开调用 Java API, 例如为了执行系统命令, 一般需要调用 Runtime 类的 exec 函数和 ProcessBuilder 类的相关函数; 为了动态执行任意代码, 一般需要调用 ClassLoader 类及其相关子类中的 defineClass 函数和 loaderClass 函数。因此, 从 Java API 调用的角度对后门代码片段做进一步分析, 统计分析后门中常见的 API 调用, 有助于更好地设计检测规则。

图 13 统计了全部 133 个后门中个出现频次排在前 20 的 Java API, 其中文件操作类占比 40%, 代码执行类占比 17.9%, 输入源类占比 17.3%, 数据传输类占比 22.5%, 命令执行类占比 2.3%。由此可见: 第一, 软件供应链攻击后门更倾向于使用文件操作类、数据传输类, 调用这两类 API 的比例高达 62.5%, 其背后的目的主要是污染系统文件或传输敏感文件; 第二, 任意代码执行是多数后门常采用的方式, 此类 API 的调用占比达到 17.9%, 因为动态加载本地或远程代码有助于逃避安全软件的查杀, 并且此种隐式触发机制也有助于攻击者灵活选择攻击载荷。

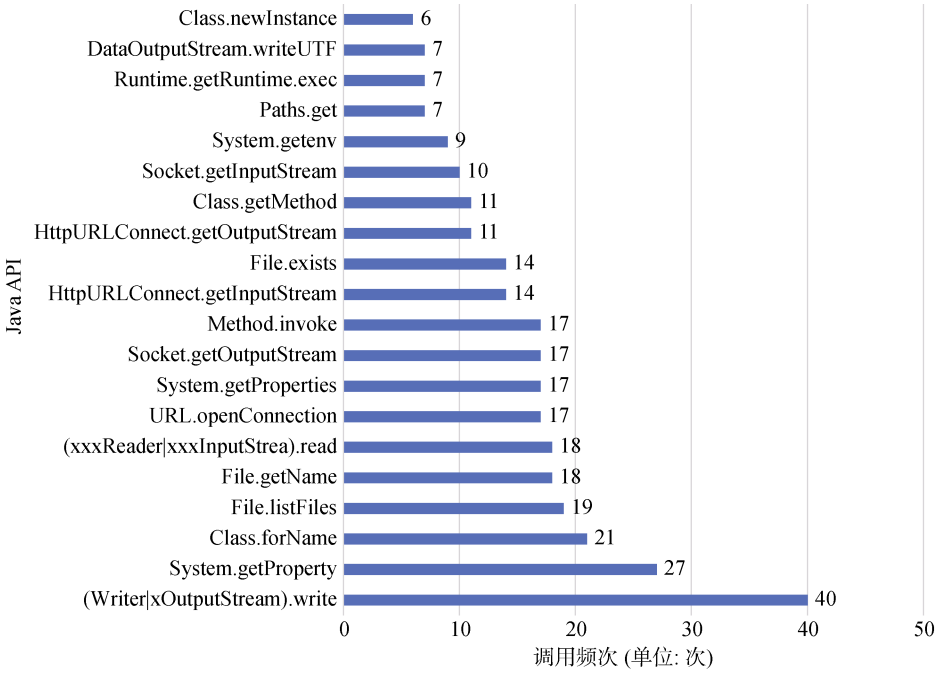


图 13　后门中调用排名前 20 的 Java API

Figure 13　Top 20 Java API calls in backdoors

5.5　案例分析

某 Jar 样本的后门由 96 行代码构成, 可以将自身提升到多种特权状态。经 JCAT 系统分析后, 该后门被划分为 9 个功能代码片段, 包括 8 个前置功能代码片段(其中 1 个是输入源)和 1 个后置功能代码片段, 图 14 展示了各功能代码片段间的逻辑关联。

该后门实现了数据传输功能、任意文件读取功能、任意文件目录枚举功能等。根据外部 Socket 所传递的数据, 显式的触发相应的攻击载荷, 达到任意文件信息传输(包括内容和目录信息)的特权状态。该后门触发了 JCAT 系统多个检测规则, 包括文件读取规则、Socket 数据传输规则、文件目录枚举规则等。这导致该后门累计的可疑分值达到了 10 分。

5.6　缺陷分析

JCAT 检测结果中包括 13 处漏报和 6 处误报, 分析其产生上述漏报、误报的原因可概括为三类: 1)反编译失败; 2)规则未命中; 3)无法检测跨函数后门。反编译失败和规则未命中是形成 13 处漏报的原因, 而 JCAT 系统无法检测跨函数后门是形成 6 处误报的原因。针对上述缺陷, 本文未来将对 JCAT 系统做进一步优化。

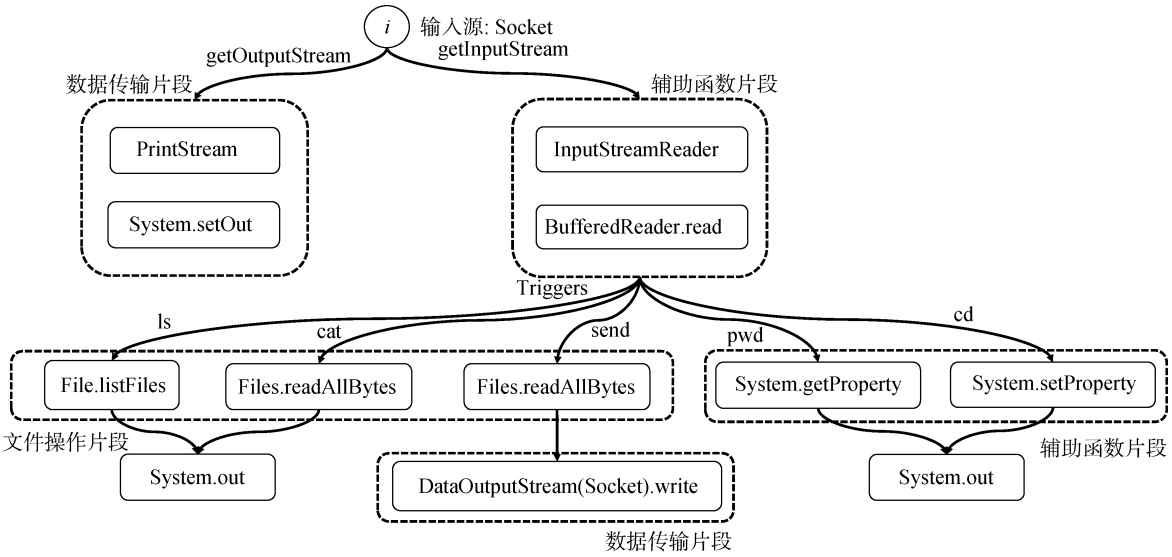


图 14　后门实例分析

Figure 14　Case study of a backdoor

针对反编译失败问题, 考虑在 JCAT 系统中集成 cfr、jd-cli 和 procyon-decompiler 之外更高性能的反编译工具, 或尝试改变现 JCAT 系统算法, 从字节码入手进行检测以减少反编译失败所造成的后门漏报。此外, 检测规则是 JCAT 系统的核心资源, 系统维护人员可以通过快速扩展机制进一步完善检测规则, 从而解决规则未命中的问题。

无法检测跨函数后门是 JCAT 系统亟待解决的能力缺陷。目前, JCAT 系统仅能在函数内部做功能代码片段的划分和检测规则的匹配, 不能综合各函数的检测结果进一步检测跨函数的后门, 因此检测跨函数的后门是未来工作的重点。

6　结论

随着 Java 开源组件的广泛应用, 开源组件的安全问题也逐渐凸显。本文针对 Java 源代码安全性检测做了相关研究, 提出并构建了 Java 后门检测模型。此外, 本文提出基于功能代码片段并结合数据流分析的静态程序分析方法用于检测函数内的潜在后门。基于此, 本文设计并实现了面向 Java 源代码安全性检测的自动化系统 JCAT。利用该系统, 本文还对阿里供应链大赛的相关样本做了检测, 后门的准确率达到 90.22%, 功能代码片段的准确率达到了 96.6%。并且对于决赛的未知样本, JCAT 系统检出了 98%的后门。

本文的研究工作仍有很大的提升空间, 拟重点解决 JCAT 系统形成漏报、误报的原因, 即反编译失败、规则未命中以及无法检测跨函数后门问题。特别地, 最后一个问题亟待解决。

参考文献

[1] “The Forrester Wave™: Software Composition Analysis, Q1, 2017”, Forrester Research, Inc., https://www.forrester.com/report/The+Forrester+Wave+Software+Composition+Analysis+Q1+2017/-/E-RES136463, Feb. 2017.

[2] “Software Supply Chain Attacks”, National Institute of Standards and Technology, U.S. Department of Commerce, https://csrc.nist.gov/CSRC/media/Projects/Supply-Chain-Risk-Management/documents/ssca/2017-winter/NCSC_Placemat.pdf, Dec. 2017.

[3] “TIOBE Index for May 2019”, TIOBE, https://www.tiobe.com/tiobe-index/, May. 2019.

[4] “Maven Repositorty”, Maven, https://mvnrepository.com/, 2019.

[5] “The State of open source security-2019”, Snyk Ltd., https://snyk.io/opensourcesecurity-2019/, Feb. 2019.

[6] “神秘 SDK 暗刷百度广告 植入数千款 APP”, 雷锋网, https://www.leiphone.com/news/201904/pPCnES607Nd6khjl.html, Apr. 2019.

[7] Zhang Yin, Vern Paxson. “Detecting Backdoors,” Conference on Usenix Security Symposium USENIX Association, 2000.

[8] Wysopal, Chris, Chris Eng, and Tyler Shields, “Static detection of application backdoors,” *Datenschutz und Datensicherheit-DuD*, vol. 34, pp. 149-155, 2010.

[9] Thomas S. L. and Francillon A., “Backdoors: Definition, Deniability and Detection,” International Symposium on Research in Attacks, Intrusions, and Defenses (*RAID'2018*), pp. 92-113, 2018.

[10] Christodorescu, Mihai, and Somesh Jha. “Static analysis of executables to detect malicious patterns,” WISCONSIN UNIV-

MADISON DEPT OF COMPUTER SCIENCES, 2006.

[11] Livshits V B and Lam M S. "Finding Security Vulnerabilities in Java Applications with Static Analysism," *USENIX Security Symposium*. vol. 14, pp. 18-18, 2005.

[12] Wang Yilan, and Guo Song. "Backdoor Detection for Java Language based On Static Analysis," *Netinfo Security*, vol. 7, pp. 43-45 (in Chinese), 2012.
(王一岚, 郭嵩, "基于静态分析的 Java 源代码后门检测技术研究", *信息网络安全*, 2012(7): 43-45。)

[13] Lin Chih-Hung, Hsing-Kuo Pao, and Jian-Wei Liao, "Efficient dynamic malware analysis using virtual time control mechanics," *Computers & Security*, vol. 73, pp. 359-373, 2018.

[14] Shibahara T., Yagi T., Akiyama M., Chiba D., Yada T, "Efficient dynamic malware analysis based on network behavior using deep learning," 2016 IEEE *Global Communications Conference* (GLOBECOM). IEEE, pp. 1-7, 2016.

[15] Hansen S. S., Larsen T. M. T., Stevanovic M., Pedersen J. M., "An approach for detection and family classification of malware based on behavioral analysis," 2016 International Conference on Computing, *Networking and Communications* (ICNC). IEEE, pp. 1-5, 2016.

[16] Wang Lina, Tan Cheng, Yu Rongwei, and Yin Zhengguang, "The Malware Detection Based on Data Breach Actions," *Journal of Computer Research and Development*, vol. 54, pp. 1537-1548 (in Chinese), 2017.
(王丽娜, 谈诚, 余荣威, 尹正光, "针对数据泄漏行为的恶意软件检测", *计算机研究与发展*, 2017, 54: 1537-1548。)

[17] Hao Zengshuai, Guo Ronghua, Wen Weiping, and MENG Zheng, "Research and Implementation on Unknown Trojan Detection System Based on Feature Analysis and Behavior Monitoring," *Netinfo Security*, vol. 15, pp. 57-65 (in Chinese), 2015.
(郝增帅, 郭荣华, 文伟平, 孟正, "基于特征分析和行为监控的未知木马检测系统研究与实现", *信息网络安全*, 2015(2): 57-65。)

[18] Wu Xin, Yan Yuesong, and Liu Xiaoran, "Program Behavior Anomaly Detection Method Based on Improved HMM," *Netinfo Security*, vol. 16, pp. 108-112, 2016.
(吴鑫, 严岳松, 刘晓然, "基于改进 HMM 的程序行为异常检测方法", *信息网络安全*, 2016(9): 108-112。)

[19] Ki Youngjoon, Eunjin Kim, and Huy Kang Kim, "A novel approach to detect malware based on API call sequence analysis," *International Journal of Distributed Sensor Networks*, vol. 2015, pp. 1-9, 2015.

[20] Shijo P. V., and A. Salim, "Integrated static and dynamic analysis for malware detection," Procedia Computer Science, vol. 46, pp. 804-811, 2015.

[21] Hopcroft, John E, "Introduction to automata theory, languages, and computation," Pearson Education India, 2008.

[22] "Spring: the source for modern java," Spring.io, https://sping.io, May. 2019.

[23] "Using Java Reflection," Oracle, https://www.oracle.com/technetwork/articles/java/javareflection-1536171.html, Jan. 1998.

[24] "深入探讨 Java 类加载器", IBM, https://www.ibm.com/developerworks/cn/java/j-lo-classloader/index.html, Mar. 2010.

[25] "Webcam Capture API," SarXos, http://webcam-capture.sarxos.pl/, May. 2019.

[26] "AudioSystem API," Oracle, https://docs.oracle.com/javase/7/docs/api/javax/sound/sampled/AudioSystem.html, May. 2019.

[27] "CFR–another java decompiler," benf.org, https://www.benf.org/other/cfr/, May. 2019.

[28] "jd-cmd–Command line Java decompiler," GitHub, Inc., https://github.com/kwart/jd-cmd, May. 2019.

[29] "Procyon/Java Decompiler," ATLASSIAN, https://bitbucket.org/mstrobel/procyon/wiki/Java%20Decompiler, May. 2019.

[30] "Heuristic," Wikimedia Foundation, Inc., https://en.wikipedia.org/wiki/Heuristic, May. 2019.

[31] "Javalang: Pure Python Java parser and tools," GitHub, Inc., https://github.com/c2nes/javalang, May, 2019

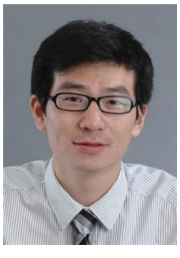
[32] "2018 年'功守道'阿里软件供应链安全大赛", Alibaba.com, https://softsec.security.alibaba.com, May. 2019

[33] "FindBugs–Find Bugs in Java Programs," http://findbugs.sourceforge.net/, May. 2019.

[34] "SpotBugs–Find Bugs in Java Programs," https://spotbugs.github.io/, May. 2019.

[35] "Fortify Software," Wikimedia Foundation, Inc., https://en.wikipedia.org/wiki/Fortify_Software, May. 2019.

[36] "360 代码卫士," 北京奇虎测腾安全技术有限公司, http://www.codesafe.cn/, May. 2019.



刘奇旭 于 2011 年在中国科学院研究生院信息安全专业获得博士学位。现任中国科学院信息工程研究所副研究员、中国科学院大学网络空间安全学院副教授。主要研究方向为网络攻防技术、网络安全评测。Email: liuqixu@iie.ac.cn

王柏柱 于 2017 年在浙江工商大学信息安全专业获得学士学位。现在中国科学院大学网络空间安全专业攻读硕士学位。研究领域为 Web 安全和程序分析。Email: wangbaizhu@iie.ac.cn

胡恩泽 于 2017 年在电子科技大学软件工程(网络安全方向)专业获得学士学位。现在中国科学院大学计算机技术专业攻读硕士学位。主要研究领域为 Web 安全和程序分析。Email: huenze@iie.ac.cn

刘井强 于 2017 年在哈尔滨工业大学计算机技术专业获得硕士学位。现任中国科学院信息工程研究所研究实习员。主要研究领域为软件供应链安全、Web 安全。Email: liujingqiang@iie.ac.cn

刘潮歌 于 2019 年在中国科学院大学信息安全专业获得博士学位。现任中国科学院信息工程研究所助理研究员、中国科学院大学网络空间安全学院讲师。主要研究领域为网络攻击追踪溯源、Web 安全和恶意代码。Email: liuchaoge@iie.ac.cn