

## 基于运行时检测的Java反序列化漏洞防御技术

李玉林<sup>1</sup>, 陈力波<sup>1</sup>, 刘宇江<sup>2</sup>, 杜文龙<sup>1</sup>, 薛质<sup>1</sup>

(1.上海交通大学网络空间安全学院, 上海 200240; 2.蚂蚁科技集团股份有限公司, 浙江 杭州 310063)

**摘要:** 反序列化漏洞自被发现以来, 便受到安全研究者的广泛关注, 越来越多的漏洞被爆出, 给企业的网络安全带来严重挑战。Java语言多态、反射等特性, 导致其反序列化漏洞利用链更加多变和复杂, 带来了更大的防御和检测难度。因此, 研究如何防御Java反序列化漏洞攻击, 成为网络防御的重要环节。通过对公开的众多Java反序列化漏洞进行研究, 提出了基于运行时检测的Java反序列化漏洞防御技术方案。根据反序列化数据类型, 将反序列化漏洞分为Java原生反序列化漏洞、JSON反序列化漏洞、XML反序列化漏洞、YAML反序列化漏洞4种类型, 并针对每种类型的反序列化漏洞, 归纳其漏洞利用过程中的反序列化入口函数; 通过Java的运行时保护技术, 对Java底层的敏感行为如命令执行进行监控, 获取系统当前的运行时上下文信息; 通过在上文信息中匹配漏洞利用中的反序列化入口函数, 来判断当前行为是否为反序列化漏洞的利用行为。在多个Java应用(如WebLogic、JBoss、Jenkins等)的测试结果表明, 该方案能有效地对Java反序列化漏洞攻击行为进行防御, 且不会对目标系统性能产生较大的影响。同时, 在与其他主流防护方案的比较中, 该方法显示出更好防护效果。

**关键词:** 反序列化漏洞; 运行时保护; 利用链; 漏洞防御

**中图分类号:** TP393

**文献标志码:** A

**DOI:** 10.11959/j.issn.2096-109x.2024021

## Java deserialization vulnerability defense technologybased on run-time detection

LI Yulin<sup>1</sup>, CHEN Libo<sup>1</sup>, LIU Yujiang<sup>2</sup>, DU Wenlong<sup>1</sup>, XUE Zhi<sup>1</sup>

1. School of Cyber Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China

2. Ant Group Company, Limited, Hangzhou 310063, China

**Abstract:** The discovery of deserialization vulnerabilities has garnered significant attention from cybersecurity researchers, with an increasing number of vulnerabilities being uncovered, posing severe threats to enterprise network

收稿日期: 2023-07-12; 修回日期: 2024-03-05

通信作者: 陈力波, bob777@sjtu.edu.cn

基金项目: 国家自然科学基金(No.62372297, No.62272306); 国家广播电视总局实验室项目(No.FYY20230001ZSB011)

**Foundation Items:** The National Natural Science Foundation of China(No.62372297, No.62272306), The National Radio and Television Administration Laboratory Program(No.FYY20230001ZSB011)

**引用格式:** 李玉林, 陈力波, 刘宇江, 等. 基于运行时检测的Java反序列化漏洞防御技术[J]. 网络与信息安全学报, 2024, 10(02): 154-164.

**Citation Format:** LI Y L, CHEN L B, LIU Y J, et al. Java deserialization vulnerability defense technologybased on run-time detection[J]. Chinese Journal of Network and Information Security, 2024, 10(02): 154-164.

security. The Java language's polymorphism and reflection capabilities render its deserialization vulnerability exploitation chains more varied and intricate, amplifying the challenges in defense and detection efforts. Consequently, developing strategies to counter Java deserialization vulnerability attacks has become a critical aspect of network security. Following an examination of numerous publicly known Java deserialization vulnerabilities, a runtime detection-based defense technology solution for Java deserialization vulnerabilities was proposed. Deserialization vulnerabilities were categorized into four types based on the data formats involved: Java native deserialization vulnerability, JSON deserialization vulnerability, XML deserialization vulnerability, and YAML deserialization vulnerability. For each type, the entry function within the exploitation process was identified and summarized. Utilizing Java's runtime protection technology, the solution monitored sensitive behaviors, such as command execution at the Java level, and captured the current runtime context information of the system. By correlating the deserialization entry function with the context information, the system can determine if the current behavior constitutes an exploitation of a deserialization vulnerability. The solution's efficacy was validated through testing on prevalent Java applications, including WebLogic, JBoss, and Jenkins. The results demonstrate that this approach can effectively protect against Java deserialization vulnerability attacks without inflicting a substantial performance penalty on the targeted system. Furthermore, when compared to other mainstream protection solutions, this method exhibits superior protective efficacy.

**Keywords:** deserialization vulnerability, run-time application self-protection, gadget chains, vulnerability defense

## 0 引言

Java反序列化是指将Java对象转成字节序列、JSON字符串、XML字符串等的过程,用于存储或在网络传输,反序列化是序列化的逆过程,将字节序列、JSON字符串、XML字符串等,转换成JAVA对象。二者相结合广泛用于应用程序的开发,随着互联网应用的发展,针对各类应用的攻击呈上升趋势,Java应用程序也暴露出许多漏洞,其中就包含Java反序列化漏洞。反序列化漏洞在2006年就已经被提出,但受到的关注度较低。2015年,Lawrence<sup>[1]</sup>在Apache的Commons Collections基础库中找到了Java反序列化攻击的执行链并首次公开了漏洞利用的方法。Commons Collections基础库作为Apache开源项目的重要组件,被广泛用于各种Java应用的开发(包括WebLogic、WebSphere、JBoss和Jenkins等产品),由于该组件覆盖范围广且危险程度较高,使得Java反序列化漏洞得到广泛关注,随后更多的Java类库被验证存在反序列化漏洞。2017年,MUNOZ<sup>[2]</sup>等对Java和.NET中大量JSON序列化库进行了全面分析,提出了许多基于.NET的库和Java的库的组件都可能存在反序列化漏洞的问题。随着各种类型的Java反序列化漏洞被发现,Java反序列化攻击的防御问题变成了棘手的问题,Java反序列化利用链的复杂性和多变性,导

致安全人员很难对Java反序列化漏洞攻击进行有效防御,给公司和企业的安全问题带来巨大隐患。Java反序列化漏洞也成为危害最严重的漏洞之一,2017年,Owasp发布了危害最严重的前10种安全漏洞,反序列化漏洞就在其中,因此反序列化漏洞的防御问题逐渐变成各大厂商非常关心的问题。

由于反序列化功能是程序的一项正常功能,所以在进行反序列化漏洞防御时,不能简单地关闭应用的反序列化功能,因此,工业界对Java反序列化漏洞的防御策略是使用黑名单机制,即将已知的反序列化漏洞攻击利用链添加到黑名单中,禁止其执行,但是黑名单机制有很明显的缺陷,容易被攻击者绕过,如果攻击者能找到一条新的反序列化漏洞利用链,那么攻击者依然可以对目标进行反序列化攻击。

针对目前Java反序列化漏洞防御问题,本文提出一种基于运行时检测的Java反序列化漏洞防御方案。该方案可以实现对目标程序的非入侵式保护,获取程序内部敏感操作函数运行时的上下文信息,通过对反序列化漏洞利用过程中的反序列化入口函数的判断,准确地识别Java反序列化漏洞的攻击,进而有效拦截,并且能快速、高效地对漏洞点进行分析定位。通过大量的实验验证了该方案对于防御Java反序列化漏洞攻击的有效性,此外,由于在同类Java反序列化漏洞利用模式上具有相似性,该方案在理论上能对未公开的

反序列化漏洞利用链进行检测防御。

## 1 相关研究

目前, Java 反序列化漏洞防御的研究主要有反序列化利用链的自动挖掘和基于运行时应用程序自我保护的防护机制 2 种。

反序列化利用链的自动挖掘, 是主动挖掘利用链, 进行修补防御。王柏柱<sup>[3]</sup>提出了基于代码属性图的 Java 反序列化漏洞链检测技术, 通过构建代码属性图检索从危险函数 Sink 到触发函数 Source 的可联通路程, 获取最终的 Java 反序列化利用链。武永兴等<sup>[4]</sup>提出了基于混合分析的 Java 反序列化漏洞利用链挖掘方法, 通过同时进行面向指针和污点变量的混合信息流分析<sup>[5-9]</sup>, 构建混合信息流图, 判断外部污点数据传播到危险函数的可达性, 构造相应的反序列化利用链。Cao 等<sup>[10]</sup>提出了新的混合解决方案—ODDFUZZ 进行轻量级静态污点分析识别可能的反序列化利用链, 并使用定向灰盒模糊验证可用的利用链。此种方式能主动发现系统中可能存在的反序列化利用链, 但是容易造成漏报, 无法保证找到系统中存在的所有利用链, 难以做到有效准确的防御。

运行时应用程序自我保护 (runtime application self protect, RASP)<sup>[11-15]</sup>的防护机制是在业务系统运行的过程中对漏洞攻击进行防御, 但目前该方向研究主要侧重于整个 Web 的防御, 单独对 Java 反序列化漏洞防御研究得较少, 何作鑫等<sup>[16]</sup>提出了基于 RASP 的 Webshell 检测方案, 通过利用 JavaAgent 技术<sup>[17-19]</sup>和白名单方式对敏感函数的集中监控, 但是该方案是在对攻击成功后的 Webshell 后门进行检测, 没有对漏洞攻击的过程进行防护。余航等<sup>[20]</sup>提出了基于 RASP 的 Web 安全检测方法, 通过在 Web 应用程序的关键函数埋下 RASP 探针防护指定类型的攻击, 但目前攻击种类较少, 只能进行简单的命令注入类、SQL 注入类的攻击检测防护, 没有 Java 反序列化攻击的防御。邱若男等<sup>[21]</sup>提出了基于 RASP 技术的 Java Web 框架漏洞通用检测与定位方案, 主要对 Spring、Struts2、Tomcat 等 Java Web 框架的远程代码执行 (remote code execution, RCE) 漏洞、反序列化漏洞、SQL 注入类漏洞和服务器配置不

当 4 类漏洞进行监控防护, 其中 Java 反序列化漏洞的防护实现比较简单, 仅考虑 Java 原生反序列化函数 readObject() 带来的漏洞, 在反序列化漏洞攻击特征的判断上, 也只是简单地判断数据流中是否存在 Magic Bytes 标识头, 并未考虑各种复杂状况 (如 XML、JSON 等第三方库的反序列化问题) 对反序列化攻击链的防御采用黑名单机制, 不能对新出现的利用链进行防御。

此外, 百度公司基于运行时应用程序自我保护技术开发了防御工具 OpenRASP, 在一定程度上能对 Java 反序列化攻击进行识别和防御, 它通过对 Java 反序列化相关函数 resolveClass() 进行 Hook 判断反序列化类是否为历史出现的恶意类, 如果是, 则阻止反序列化。其本质是一种黑名单机制, 通过将历史出现的反序列化漏洞利用类加入黑名单来阻止 Java 反序列化漏洞的攻击, 此种方式并不能防御新出现的 Java 反序列化漏洞利用链的攻击, 同时这种方式只能处理 Java 原生反序列化漏洞的情形, 不适用于 JSON、XML 等反序列化漏洞的情况。

除了以上 2 种方式外, 互联网上也有许多开源的 Java 反序列化漏洞检测工具, 如 Java-Deserialization-Scanner。这类工具侧重于漏洞的检测与发现, 能检测出一些特定的反序列化漏洞。但该类工具有 2 个主要问题: 依赖于公开的反序列化漏洞利用链, 可能会导致无法检测出因为新利用链带来的漏洞; 需要知道目标 url, 且发送的数据如 http 头部数据、post 数据等相对比较固定, 只能检测一些逻辑简单的漏洞。

## 2 反序列化漏洞原理分析

### 2.1 反序列化漏洞分类

针对以上状况, 本文对 Java 反序列化漏洞进行分类。Java 程序在对数据进行反序列化处理时, 通常需要处理的数据有字节流、XML 字符串、JSON 字符串和 YAML 字符串 4 种类型。根据不同的处理类型, 本文将 Java 反序列化分为 Java 原生反序列化、XML 反序列化、JSON 反序列化和 YAML 反序列化 4 种类型。

### 2.2 反序列化漏洞利用分析

Java 反序列化漏洞攻击过程中, 攻击者一般

通过HTTP向存在漏洞的Web应用程序发送特定攻击载荷即反序列化漏洞利用链，虽然每次发送的漏洞利用链的执行路径存在差别，但最终达到的攻击结果基本相似，如执行系统命令、读写文件等。

为进一步分析反序列化漏洞的特性，本文首先选择主流的Java框架或应用（如JBoss、Jenkins、Weblogic、Spring、Struts2等）这些应用有较为广泛的用户基础，在实际生产环境中经常遇到。通过fofa空间搜索引擎可知，JBoss、Spring和Struts2每种应用均有百万级别的匹配结果，其他应用也有几万到几十万的匹配结果，用户量巨大，相关漏洞造成的危害性也较高。近几年爆出的反序列化漏洞如表1所示。通过对34个的反序列化漏洞分析可知，被漏洞攻击程序的内部和同类别的反序列化漏洞的攻击过程存在相似的执行路径，这也是本文方案设计的基础。

下面以CVE-2017-7504为例，分析展示Java反序列化漏洞利用过程。

该漏洞是2017年JBoss爆出的漏洞，HTTPServerILServlet.Java中JBossMQ功能在实现上未限制执行反序列化的类，允许远程攻击者通过精心设计的序列化数据执行任意代码。CVE-2017-7504漏洞代码如图1所示。程序从HTTP协议接收数据流，在processRequest()函数中接收数据流并调用readObject()函数对数据流进行反序列化操作，造成了Java原生反序列化漏洞的存在。使用开源框架ysoserial生成一段反序列化利用链，对该漏洞进行攻击测试，调试跟踪反序列化漏洞的攻击过程，CVE-2017-7504调用

栈如图2所示，系统首先调用org.jboss.mq.il.http.servlet.HTTPServerILServlet.doPost()函数接收POST请求的攻击数据；然后，调用org.jboss.mq.il.http.servlet.HTTPServerILServlet.processRequest()函数处理恶意的攻击数据，数据进入反序列化入口函数Java.io.ObjectInputStream.readObject()进行反序列化处理；接着，执行反序列化利用链中间函数；最后，执行敏感操作函数Java.lang.Runtime.exec()，实现了执行系统命令的攻击效果。

```
protected void processRequest(HttpServletRequest request, HttpServletResponse response){
    .....
    ObjectInputStream inputStream=new ObjectInputStream((InputStream).request.getInputStream());
    HTTPILRequest=(HTTPILRequest)inputStream.readObject();
    .....
}
```

图1 CVE-2017-7504漏洞代码  
Figure 1 Vulnerability code of CVE-2017-7504

```
javax.servlet.http.HttpServlet.service()
org.jboss.mq.il.http.servlet.HTTPServerILServlet.doPost
org.jboss.mq.il.http.servlet.HTTPServerILServlet.processRequest()
java.io.ObjectInputStream.readObject()
java.io.ObjectInputStream.readObject()
java.io.ObjectInputStream.readOrdinaryObject()
java.io.ObjectInputStream.readSerialData()
.....
org.apache.commons.collections.functors.ChainedTransformer.transform()
org.apache.commons.collections.functors.InvokerTransformer.transform()
java.lang.reflect.Method.invoke()
.....
com.payload.RunCommand.<init>()
java.lang.Runtime.exec()
java.lang.ProcessBuilder.start()
java.lang.ProcessImpl.start()
```

图2 CVE-2017-7504调用栈  
Figure 2 Stack of CVE-2017-7504

通过分析各种类型反序列化漏洞利用过程，总结反序列化漏洞通用利用模式如图3所示。其中POST/GET处理数据过程是Web系统通用流程，可以忽略，也就是说任何一次的反序列化漏洞利用必然会从反序列化入口函数执行到底层的

表1 反序列化漏洞  
Table 1 Deserialization vulnerability

| 漏洞类型        | 漏洞编号                                                                                                                                                                                        |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Java 原生反序列化 | CVE-2013-4810,CVE-2015-7501,CVE-2015-4852,<br>CVE-2016-3510,CVE-2017-7504,CVE-2017-1000353<br>CVE-2018-1000861,CVE-2017-12149                                                               |
| JSON 反序列化   | CVE-2017-18349,CVE-2020-8840,CVE-2022-25845,<br>CVE-2017-7525,CVE-2017-15095,CVE-2017-17485,<br>CVE-2019-12384,CVE-2020-36188,CVE-2020-7961                                                 |
| XML 反序列化    | CVE-2016-0792,CVE-2017-3506,CVE-2017-10271,<br>CVE-2017-10352,CVE-2019-2725,CVE-2023-24162,<br>CVE-2013-7258,CVE-2020-26217,CVE-2020-26259,<br>CVE-2021-21351,CVE-2021-21344,CVE-2021-21345 |
| YAML 反序列化   | CVE-2016-9606,CVE-2017-3159,CVE-2016-8744,<br>CVE-2021-36162,CVE-2022-1471                                                                                                                  |

表2 反序列化分类  
Table 2 Deserialization classification

| 类型          | 反序列化入口函数                                                                                                                                                                        |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Java 原生反序列化 | Java.io.ObjectInputStream.readObject()                                                                                                                                          |
| JSON 反序列化   | Fastjson<br>com.alibaba.fastjson.JSON.parseObject()<br>Jackson<br>com.fasterxml.jackson.databind.ObjectMapper.readValue()<br>FlexSon<br>flexjson.JSONDeserializer.deserialize() |
| XML 反序列化    | XMLDecoder<br>Java.beans.XMLDecoder.readObject()<br>XStream<br>com.thoughtworks.xstream.XStream.fromXML()                                                                       |
| YAML 反序列化   | SnakeYaml<br>org.yaml.snakeyaml.Yaml.load()                                                                                                                                     |

敏感操作函数（命令执行、文件读写等），所以通过反序列化入口函数和敏感操作函数结合可以确定一次反序列化攻击。而不同类型的反序列化漏洞利用过程，除了反序列化入口函数不同之外，其他过程基本一致。而对于反序列化入口函数，Java 原生反序列化漏洞入口函数是固定的，其他 JSON 反序列化、XML 反序列化和 YAML 反序列化的入口函数与使用的第三方库相关，本文整理了 4 种类型反序列化分类如表 2 所示。

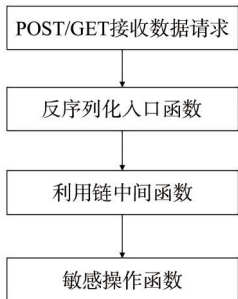


图3 反序列化漏洞通用利用模式

Figure 3 General exploitation pattern of deserialization vulnerability

### 2.3 敏感操作函数分析

敏感操作函数即 Java 反序列化漏洞利用最终想要执行的操作，一般有命令执行和文件读写 2 种类型，通常是读取敏感文件或写入网站后门 WebShell 文件等。Java 中命令执行和文件读写的方法很多，有些方法存在嵌套关系，工程实现上不能对每个函数进行监控，那样会造成大量的重复工作，需要通过对这些函数进行跟踪分析，观察这些函数最终调用的是哪些函数，然后对其进行 Hook 监控，减少重复工作，提高系统效率。

Java 中执行命令函数主要有 Java.lang.Runtime()、Java.lang.ProcessBuilder()和 Java.lang.ProcessImpl()，通过动态跟踪可知这 3 种执行命令最

终都调用了 Java.lang.ProcessImpl()函数。

Java 中文件读写的函数就比较多了，比如 FileWriter()、BufferedWriter()、FileOutputStream()、BufferedReader()和 FileInputStream()等，通过分析梳理，所有文件读写的方式最终调用以下 5 个函数：Java.io.FileOutputStream()、Java.io.FileInputStream()、Java.io.RandomAccessFile()、Java.nio.channels.Channels.newOutputStream()和 Java.nio.channels.Channels.newInputStream()。

## 3 方案设计

### 3.1 防御思路提出

由第 2.2 节分析可知，反序列化入口函数和敏感操作函数可以确定一次 Java 反序列化漏洞攻击行为，攻击路径如图 4 所示，流经链路 1→2→3 或 1→2→4 过程即为反序列化攻击过程。正常对象或数据在反序列化过程中是不会进入敏感操作函数的，如命令执行或文件操作，而反序列化漏洞的攻击却在反序列化过程中进行了这些危险操作，这给 Java 反序列化漏洞防御提供了方向。

如果能获取系统每次敏感操作前的控制流，则能判断敏感操作是从图 4 中哪条链路过来的。通过在控制流中匹配反序列化入口函数，如果匹配某种反序列化入口函数，则认为此次行为是来自某反序列化漏洞的攻击行为，没有匹配，则认为此次行为为正常行为。

### 3.2 防御方案设计与实现

通过对 Java 反序列化漏洞的总结以及本文防御思路，设计了防御方案的整体框架，本文方案框架设计如图 5 所示。该防御框架共分为 3 个模块，分别为上下文信息搜集模块、漏洞判定模块

和漏洞预警模块。其中上下文信息搜集模块通过Java的Agent机制,将RASP探针嵌入Java的底层敏感操作函数中,在执行敏感操作前,通过Web访问所依赖的线程副本,可完整记录访问线程经过的函数流程。上下文信息搜集后,通过漏洞判定模块对搜集的上下文信息进行处理,判断是否出现攻击行为,然后将判定结果发给漏洞预警模块。如果出现漏洞攻击行为,则拦截并记录该攻击行为和上下文信息;如果是正常行为,则不做任何处理。

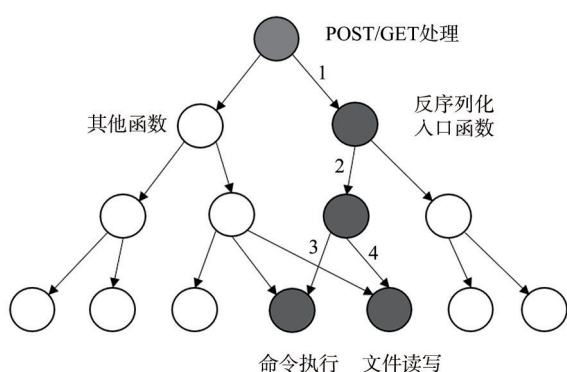


图4 攻击路径  
Figure 4 Attack path

### 3.2.1 上下文信息搜集模块

上下文信息搜集模块使用了Java的RASP技术,它能以非入侵方式监控并拦截目标程序内部运行信息,本文方案使用了蚂蚁公司基于RASP技术封装的开源框架Ranger-Java来监控函数。

Ranger-Java是Java语言面向切面编程的实现,基于字节码注入技术,可以在不参与业务系统的情况下对业务进行数据内视和逻辑干预,支持多Module编程,通过Module编程机制,用户可以快速地对指定函数进行挂钩,并在函数执行前添加自定义代码,获取用户关心的数据或改变函数的执行流程。

上下文信息搜集模块主要拦截Java的底层敏感操作,对其进行监控。主要有监控命令执行、文件写和线程创建3种行为,使用Ranger-Java获取对应函数执行前的上下文信息。

漏洞攻击者的首要目标是命令执行,Java提供了几种命令执行方式供攻击者直接调用,所以命令执行是漏洞防御时重点监控的操作,由第2.3节可知,命令执行最终调用函数为Java.lang.ProcessImpl.start(),所以在监控系统命令执行行为时,该模块对Java.lang.ProcessImpl.start()函数监控,并对其调用获取当前的上下文信息,从而得知执行命令的控制流,进而通过匹配反序列化入口函数,判断此行为是否具有攻击性。

文件读写是系统常用功能,也是攻击者常用手段,通过调用Java自带的读函数,可以读取目标系统上的敏感文件,包括密码文件、配置文件等;通过写操作,可以在目标系统中修改关键文件或放置后门WebShell等。所以文件读写操作也是防御中需要重点关注的操作,Java中读写文件

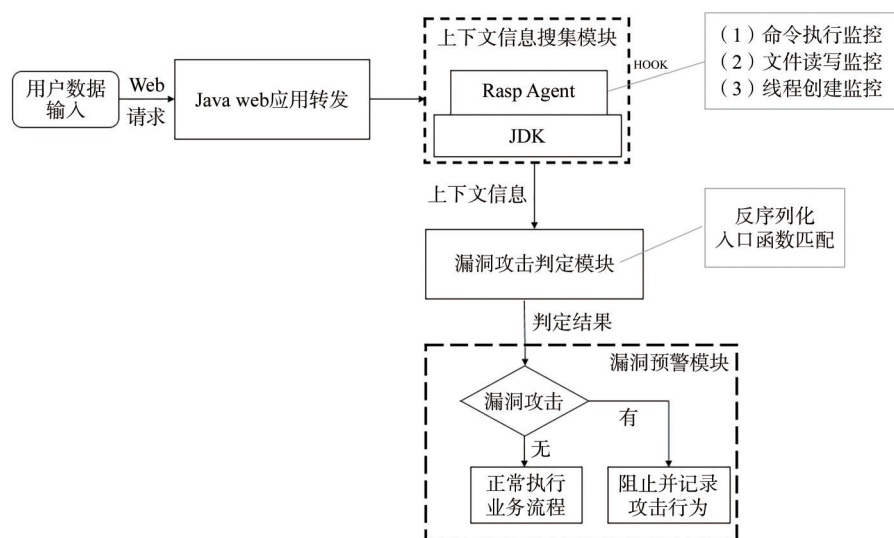


图5 本文方案框架设计

Figure 5 The framework design of the proposed scheme

方式众多,由第2.3节内容可知,读写操作最终都会执行以下5个函数,所以上下文信息搜集模块对以下5个函数进行监控:Java.io.FileOutputStream()、Java.io.FileInputStream()、Java.io.RandomAccessFile()、Java.nio.channels.Channels.newOutputStream()和Java.nio.channels.Channels.newInputStream()。在每个读写函数执行前获取上下文信息,方便判断是否为攻击行为。

线程创建行为是工程实现上需要考虑的问题,因为上下文信息的获取是不能跨线程的,正常情况程序只能获取当前线程的上下文信息。如果攻击者构造的多线程攻击代码如图6所示。命令执行代码在线程中(该段代码对应的调用堆栈如图7所示),此时的调用堆栈并不包含完整的攻击过程,因而无法直接通过反序列化入口函数判断该行为是否具有攻击性。所以针对通过创建线程进行攻击的行为,需要进行单独处理。众所周知,Java反序列化是处理对象或数据的传输和存储,在反序列化过程中很少执行Java的各种方法函数,而是进行简单的数据转化,更不可能存在创建线程的行为。通过对大量漏洞和对应利用Payload的研究分析可以发现,攻击者在使用常规攻击行为被目标系统防御拦截后,会尝试将命令执行或文件读写等敏感攻击行为放在线程中,试图绕过目标的防御机制。所以在反序列化中如果出现了创建线程行为,则可认为此次反序列化行为是一次攻击行为。因此上下文信息搜集模块需要监控线程创建函数Java.lang.Thread.start(),在线程函数调用之前获取当前的上下文信息,然后使用漏洞判定模块进行行为判定。

```
public class Payload_Thread {
    public Payload_Thread(){
        new Thread(){
            @Override
            public void run() {
                try {
                    Runtime.getRuntime().exec("calc");
                }
                catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }.start();
    }
}
```

图6 多线程攻击代码  
Figure 6 Multi-threading attack code

```
Payload_Thread$1.run()
java.lang.Runtime.exec()
java.lang.ProcessBuilder.start()
java.lang.ProcessImpl.start()
java.lang.ProcessImpl.<init>()
```

图7 调用堆栈  
Figure 7 Call stack

### 3.2.2 漏洞判定模块

由第2.2节可知,每种类型的反序列化漏洞利用都会执行固定的反序列化入口函数,这是漏洞判定模块进行漏洞攻击判断的前提。该模块首先接收从上下文搜集模块传递的调用链信息;然后,利用检测算法进行反序列化入口函数匹配。如果在调用链中匹配到反序列化入口函数,则判定此行为是Java反序列化漏洞攻击行为;如果未匹配到反序列化入口函数,则判定此次行为是正常用户行为。漏洞判断框架如图8所示。

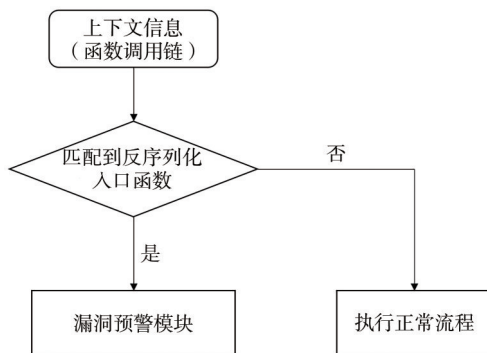


图8 漏洞判断框架  
Figure 8 The framework for vulnerability assessment

如果函数调用链中匹配到Java.io.ObjectInputStream.readObject(),则认为攻击为Java原生反序列化漏洞攻击;如果匹配到com.alibaba.fastjson.JSON.parseObject(),则认为攻击为Fastjson反序列化漏洞攻击;如果匹配到flexjson.JSONDeserializer.deserialize(),则认为攻击为Flexjson反序列化漏洞攻击;如果匹配到com.fasterxml.jackson.databind.ObjectMapper.readValue(),则认为攻击为Jackson反序列化漏洞攻击;如果匹配到Java.beans.XMLDecoder.readObject(),则认为攻击为XmlDecoder反序列化漏洞攻击;如果匹配到com.thoughtworks.xstream.XStream.fromXML(),则认为攻击为XStream反序列化漏洞攻击;如果

匹配到 `org.yaml.snakeyaml.Yaml.load()`，则认为攻击为SnakeYaml反序列化漏洞攻击。如果均未匹配到，则此次行为为正常的访问行为。

### 3.2.3 漏洞预警模块

漏洞预警模块具备以下关键功能：详细记录Java反序列化漏洞攻击类型和攻击过程的上下文信息；阻止Java反序列化漏洞攻击行为，并将攻击行为报告管理员。

该模块接收漏洞判定模块的结果，如果判定结果为漏洞攻击行为，则阻止该攻击行为，并记录到文件中搜集模块搜集的上下文信息，向管理员发送攻击预警信息和攻击上下文信息，其中上下文信息可供管理员分析漏洞使用。此外，该模块还具有判断此次反序列化攻击利用链是新的利用链，还是网上已经爆出的利用链，搜集整理网上已经爆出的Java反序列化漏洞利用链，提炼出利用链中使用的危险反序列化类，将其入库。如果漏洞预警模块判定了某次反序列化漏洞攻击行为，则将此次攻击链中使用的反序列化类与库中已有反序列化类进行匹配；如果没有匹配到反序列化类，则认为此次攻击使用了新的反序列化利用链，然后记录详细信息，报告管理员。

## 4 实验分析

### 4.1 防御效果测试

本文提出的防御框架使用Java编写，可运行

在Windows或Linux系统。从34个漏洞中选取了12个漏洞进行测试，选择的漏洞覆盖了不同框架和4种反序列化类型。实验环境配置Javaagent参数，加载防御框架Ranger-Java，并使用各个漏洞对应的Payload以及各种变形Payload。攻击测试时，观察框架是否能有效检测和防御漏洞攻击，反序列化漏洞检测结果如表3所示。由表3可知，本文方案能有效地检测Java反序列化漏洞的攻击行为，并且能进行有效防御，成功率达100%。

### 4.2 与现有方案OpenRASP对比

在目前实际生产环境中，针对Java反序列化防御产品较少，选取百度开源的同类型产品OpenRASP（V1.37版本）与本文方案进行对比，由于OpenRASP的防御机制覆盖面较广（包括SQL注入、文件上传、命令执行、反序列化等），所以本文实验只选择OpenRASP的反序列化漏洞防御的功能进行比较。实验选择了32个不同类型的攻击Payload，分别对使用本文方案和OpenRASP方案进行攻击测试，与OpenRASP比较结果如表4所示。

从实验结果可知，本文方案的Java反序列化漏洞防御效果明显优于OpenRASP，一是OpenRASP是相对综合的防御框架，针对的类型不仅包括Java反序列化，还包括其他多种类型的漏洞，所以对Java反序列化漏洞防御处理得不太理想；二是OpenRASP中，Java反序列化漏洞防御

表3 反序列化漏洞检测结果  
Table 3 Results of deserialization vulnerability detection

| 序号 | 漏洞编号             | 漏洞类型              | 框架             | 危害类型 | 防御结果 |
|----|------------------|-------------------|----------------|------|------|
| 1  | CVE-2013-4810    | Java 原生反序列化漏洞     | JBoss          | RCE  | √    |
| 2  | CVE-2015-7501    | Java 原生反序列化漏洞     | JBoss          | RCE  | √    |
| 3  | CVE-2017-12149   | Java 原生反序列化漏洞     | JBoss          | RCE  | √    |
| 4  | CVE-2017-3506    | Xmldecoder 反序列化漏洞 | WebLogic       | RCE  | √    |
| 5  | CVE-2017-10271   | Xmldecoder 反序列化漏洞 | WebLogic       | RCE  | √    |
| 6  | CVE-2019-2725    | Xmldecoder 反序列化漏洞 | WebLogic       | RCE  | √    |
| 7  | CVE-2016-0792    | XStream 反序列化漏洞    | Jenkins        | RCE  | √    |
| 8  | CVE-2017-1000353 | Java 原生反序列化漏洞     | Jenkins        | RCE  | √    |
| 9  | CVE-2017-18349   | Fastjson 反序列化漏洞   | JFinal         | RCE  | √    |
| 10 | CVE-2017-7525    | Jackson 反序列化漏洞    | Apache Struts2 | RCE  | √    |
| 11 | CVE-2020-7961    | Flexjson 反序列化漏洞   | Liferay        | RCE  | √    |
| 12 | CVE-2022-1471    | SnakeYaml 反序列化漏洞  | Spring boot    | RCE  | √    |

机制以黑名单为主,即只针对已公开的攻击链进行防御,如果黑名单搜集不全或出现新的漏洞利用链,OpenRASP都无法进行有效防御。而本文方案以漏洞攻击过程中的反序列化入口函数为检测点,不管攻击链如何变化,攻击过程都会执行反序列化入口函数,从而更有效地进行防御。此外,本文方案还记录攻击过程的上下文信息,可辅助管理员更快、更高效地定位漏洞细节,从而更有效地对系统进行防护。

表4 与OpenRASP比较结果  
Table 4 Results of comparison with OpenRASP

| 方案       | 实验 Payload | 防御 Payload | 成功率    |
|----------|------------|------------|--------|
| 本文方案     | 32         | 32         | 100%   |
| OpenRASP | 32         | 25         | 78.13% |

#### 4.3 性能测试

实验选择 Jenkins 和 JFinal 两种 Java 框架进行性能测试,观察本文方案对原框架的性能影响程度。性能测试采用 Apache 公司提供的 JMeter 性能测试工具,分别对 Jenkins 和 JFinal 进行访问测试,观察和记录吞吐量和响应时间。

实验中,首先将正常的 Jenkins 程序作为性能测试对象,使用 JMeter 对其进行访问测试,统计吞吐量和响应时间等数据;然后,使用本文方案框架对 Jenkins 程序进行保护,再使用 JMeter 对其进行访问测试,统计吞吐量和响应时间等数据。实验设置 JMeter 活跃线程数为 200,10 s 内随机启动完成,循环 20 次,代表 200 个用户在 10 s 内同时访问系统循环 20 次,因此,样本总数为 4 000。对 JFinal 也同样的性能测试。某个 10 s 内, Jenkins 的数据请求响应时间如图 9 所示, JFinal 的数据请求响应时间如图 10 所示。

假设在运行防御程序前、后系统的平均响应时间分别用  $T_1$  和  $T_2$  表示,定义性能消耗为:

$$\text{性能消耗} = \frac{T_2 - T_1}{T_1} \cdot 100\% \quad (1)$$

Jenkins 性能消耗如表 5 所示, JFinal 性能消耗如表 6 所示。由表 5 和表 6 可知,运行防御程序后 Jenkins 和 JFinal 的性能消耗均在 4% 左右,表明防御框架并未造成较大的性能消耗,基本不会影响用户使用。

此外,实验统计了 Jenkins 和 JFinal 使用防御

框架后服务器每秒处理的事务数 (transactions per second, TPS)。某个 10 s 内, Jenkins 的 TPS 如图 11 所示, JFinal 的 TPS 如图 12 所示。由图 11 和图 12 可知,在使用防框架前、后,系统的 TPS 几乎一样,影响不大。

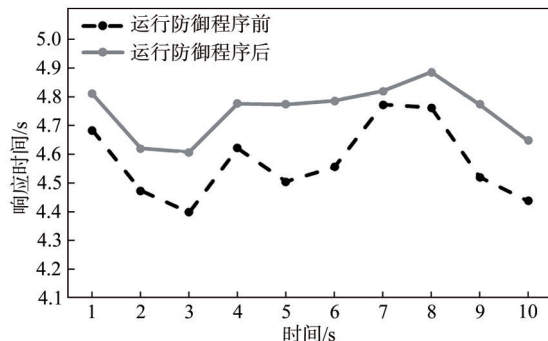


图9 Jenkins 的数据请求响应时间  
Figure 9 Response time for data requests in Jenkins

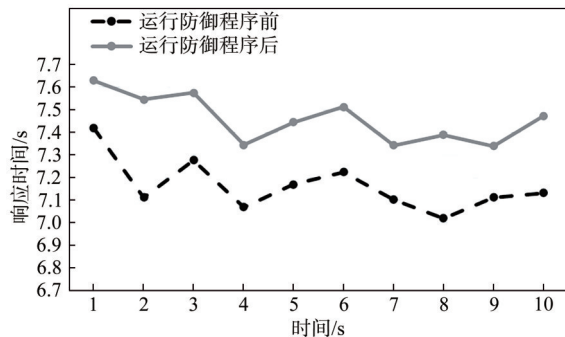


图10 JFinal 的数据请求响应时间  
Figure 10 Response time for data requests in JFinal

表5 Jenkins性能消耗  
Table 5 Performance consumption of Jenkins

| 测试系统         | 平均响应时间/s | 性能消耗  |
|--------------|----------|-------|
| 运行前( $T_1$ ) | 4.57     | 3.93% |
| 运行后( $T_2$ ) | 4.75     |       |

表6 JFinal性能消耗  
Table 6 Performance consumption of JFinal

| 测试系统         | 平均响应时间/s | 性能消耗  |
|--------------|----------|-------|
| 运行前( $T_1$ ) | 7.16     | 4.19% |
| 运行后( $T_2$ ) | 7.46     |       |

综上所述,防御框架对目标应用程序的总体性能影响不大,几乎不会影响用户的实际体验。

#### 4.4 问题与改进

目前,本文方案对有些行为的判断粒度较粗,相对简单,可能存在误报的情况。如目标框

架或应用中存在 ThreadObjectTest 代码, 如图 13 所示。ThreadObjectTest 重写了 readObject 函数, 并调用了创建线程操作且目标框架恰好需要对 ThreadObjectTest 进行反序列化处理, 则监控 Java.lang.thread.start 线程函数获取的上下文信息存在 Java.io.ObjectInputStream.readObject, 会造成误报并且影响程序的正常反序列化过程。由第 3 节内容分析可知, 该情况并不常见, 但是存在导致本文方案误报的可能性。

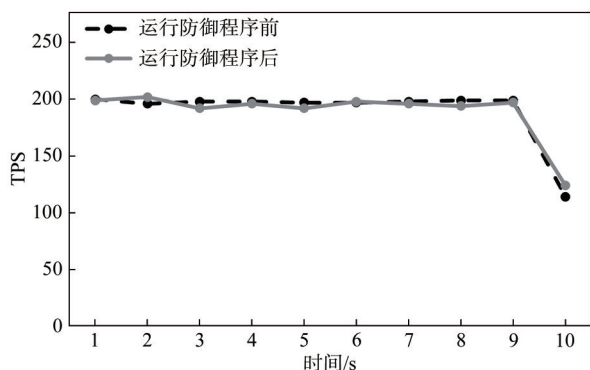


图 11 Jenkins 的 TPS  
Figure 11 TPS of Jenkins

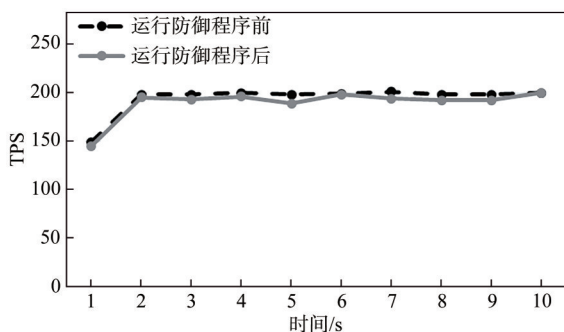


图 12 JFinal 的 TPS  
Figure 12 TPS of JFinal

```
public class ThreadObjectTest implements Serializable {
    private String username;
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    private void readObject(ObjectInputStream in) throws Exception {
        in.defaultReadObject();
        new Thread() {
            @Override
            public void run() {
                try {
                    System.out.println(getUsername());
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }.start();
    }
}
```

图 13 ThreadObjectTest 代码  
Figure 13 Code of ThreadObjectTest

同时, 本文方案针对的漏洞类型较单一, 只能处理反序列化漏洞。在后续工作中, 可通过分析其他类型的 RCE 漏洞, 如对象图导航语言注入漏洞、表达式语言注入漏洞、Velocity 模板注入漏洞等。这类漏洞在利用过程中, 先对输入的内容进行解析, 然后执行攻击命令, 可以总结归纳关键的漏洞利用环节, 拓展本文方案, 进一步覆盖更多类型的漏洞检测防御。

## 5 结束语

本文通过分析总结现有 Java 反序列化漏洞, 将 Java 反序列化漏洞按照不同情况分为 4 个大类别和 7 个小类别, 总结了 7 种类型反序列化漏洞的通用利用模式, 在其基础上提出了基于运行时检测的 Java 反序列化漏洞防御方案。针对这些类型反序列化漏洞的 Java 框架和程序, 进行攻击测试, 检测本文方案对漏洞攻击的检测和防御能力, 并与百度开源的 OpenRASP 产品进行对比实验和性能测试, 实验结果证明了本文方案的有效性。

## 参考文献:

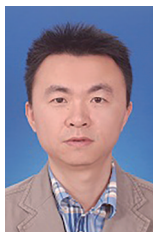
- [1] LAWRENCE G. Marshalling pickles[EB]. 2015.
- [2] MUNOZ A. Friday the 13th: JSON attacks[EB]. 2017.
- [3] 王柏柱. 基于代码属性图的 Java 反序列化利用链检测技术研究[D]. 北京: 中国科学院大学, 2021.  
WANG B Z. Research on Java deserialization utilization chain detection technology based on code attribute graph[D]. Beijing: University of Chinese Academy of Sciences, 2021.
- [4] 武永兴, 陈力波, 姜开达. 基于混合分析的 Java 反序列化利用链挖掘方法[J]. 网络与信息安全学报, 2022, 8(2): 160-174.  
WU Y X, CHEN L B, JIANG K D. Java deserialization gadget chain discovery method based on hybrid analysis[J]. Chinese Journal of Network and Information Security. 2022, 8(2): 160-174.
- [5] TRIPP O, PISTOIA M, FINK S J, et al. TAJ[J]. ACM SIGPLAN Notices, 2009, 44(6): 87-97.
- [6] HUANG W, DONG Y, MILANOVA A. Type-based taint analysis for Java web applications[C]//Proceedings of the International Conference on Fundamental Approaches to Software Engineering. Berlin: Springer, 2014: 140-154.
- [7] SRIDHARAN M, ARTZI S, PISTOIA M, et al. F4F: taint analysis of framework-based web applications[C]//Proceedings of the 2011 ACM international Conference on object Oriented Programming Systems Languages and Applications. New York: ACM press, 2011: 1053-1068.
- [8] LERCH J, HERMANN B, BODDEN E, et al. FlowTwist: efficient context-sensitive inside-out taint analysis for large codebases [C]//Proceedings of the 22nd ACM SIGSOFT International Sym-

- posium on Foundations of Software Engineering. New York: ACM press, 2014: 98-108.
- [9] 王蕾, 李丰, 李炼, 等. 污点分析技术的原理和实际应用[J]. 软件学报, 2017, 28(4): 860-882.  
WANG L, LI F, LI L, et al. Principle and practice of taint analysis[J]. Journal of Software. 2017, 28(4): 860-882.
- [10] CAO S C, HE B, SUN X B, et al. ODDFuzz: discovering Java deserialization vulnerabilities via structure-aware directed grey-box fuzzing[C]//Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP). Piscataway: IEEE Press, 2023: 2726-2743.
- [11] HUANG Y W, YU F, HANG C, et al. Securing web application code by static analysis and runtime protection[C]//Proceedings of the 13th International Conference on World Wide Web. New York: ACM press, 2004: 40-52.
- [12] AMOROSO E. Recent progress in software security[J]. IEEE Software. 2018, 35(2): 11-13.
- [13] SETH A, BHATTACHARYA S, ELDER S, et al. Comparing effectiveness and efficiency of interactive Application Security testing (IAST) and runtime application self-protection (RASP) tools in ALarge java-based system[J]. SSRN Electronic Journal, 2022.43-44.
- [14] ČISAR P, ČISAR S M. The framework of runtime application self-protection technology[C]//Proceedings of the 2016 IEEE 17th International Symposium on Computational Intelligence and Informatics (CINTI). Piscataway: IEEE Press, 2016: 81-86.
- [15] 奚阳. Java 程序运行时保护与监控技术研究[D]. 南京: 南京大学, 2008.  
XI Y. Research on runtime protection and monitoring technology of Java programs[D]. Nanjing: Nanjing University, 2008.
- [16] 何作鑫. 基于 RASP 的 Webshell 检测程序设计 with 实现[D]. 重庆: 西南大学, 2022.  
HE Z X. Design and implementation of Webshell detection program based on RASP[D]. Chongqing: Southwest University, 2022.
- [17] BINDER W, HULAAS J, MORET P. Advanced Java bytecode instrumentation[C]//Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java. New York: ACM, 2007: 135-144.
- [18] MASTRANGELO L, HAUSWIRTH M. JNIF: Java native instrumentation framework[C]//Proceedings of the Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools. New York: ACM press, 2014: 194-199.
- [19] BINDER W, HULAAS J, MORET P. A quantitative evaluation of the contribution of native code to Java workloads[C]//Proceedings of the 2006 IEEE International Symposium on Workload Characterization. Piscataway: IEEE Press, 2006: 201-209.
- [20] 余航, 王帅, 金华敏. 基于 RASP 的 Web 安全检测方法[J]. 电信科学, 2020, 36(11): 113-120.  
YU H, WANG S, JIN H M. RASP based Web security detection method[J]. Telecommunications Science, 2020, 36(11): 113-120.
- [21] 邱若男, 胡岸琪, 彭国军, 等. 基于 RASP 技术的 Java Web 框架漏洞通用检测与定位方案[J]. 武汉大学学报(理学版), 2020, 66(3): 285-296.  
QIU R N, HU A Q, PENG G J, et al. A general detection and location scheme for Java Web framework vulnerability based on RASP technology[J]. Journal of Wuhan University (Natural Science Edition). 2020, 66(3): 285-296.

## [作者简介]



李玉林 (1990-), 男, 安徽阜阳人, 上海交通大学硕士生, 主要研究方向为 Web 安全、IoT 设备安全。



陈力波 (1985-), 男, 江西广丰人, 上海交通大学高级工程师、硕士生导师, 主要研究方向为软件与系统安全、物联网安全、虚拟化安全。



刘宇江 (1990-), 男, 江西樟树人, 蚂蚁集团高级安全专家, 主要研究方向为企业安全架构、安全基础设施、网络安全攻防对抗、数据安全治理体系。



杜文龙 (1994-), 男, 安徽宿州人, 上海交通大学硕士生, 主要研究方向为设备的漏洞挖掘与利用。



薛质 (1971-), 男, 江苏常州人, 上海交通大学教授、博士生导师, 主要研究方向为计算机通信网、网络攻防与评测、安全主动网。