面向 Java 的高对抗内存型 Webshell 检测技术

张金莉[1,2], 陈星辰[1,2], 王晓蕾[1,2], 陈庆旺[1,2], 代 峰[1], 李香龙[1,2],
冯 云[1], 崔 翔[1,3]

[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
[3] 广州大学网络空间先进技术研究院 广州 中国 510006

摘要 由于 Web 应用程序的复杂性和重要性, 导致其成为网络攻击的主要目标之一。攻击者在入侵一个网站后, 通常会植入一个 Webshell, 来持久化控制网站。但随着攻防双方的博弈, 各种检测技术、终端安全产品被广泛应用, 使得传统的以文件形式驻留的 Webshell 越来越容易被检测到, 内存型 Webshell 成为新的趋势。内存型 Webshell 在磁盘上不存在恶意文件, 而是将恶意代码注入到内存中, 隐蔽性更强, 不易被安全设备发现, 且目前缺少针对内存型 Webshell 的检测技术。本文面向 Java 应用程序, 总结内存型 Webshell 的特征和原理, 构建内存型 Webshell 威胁模型, 定义了高对抗内存型 Webshell, 并提出一种基于 RASP(Runtime application self-protection, 运行时应用程序自我保护)的动静态结合的高对抗内存型 Webshell 检测技术。针对用户请求, 基于 RASP 技术监测注册组件类函数和特权类函数, 获取上下文信息, 根据磁盘是否存在文件以及数据流分析技术进行动态特征检测, 在不影响应用程序正常运行的前提下, 实时地检测; 针对 JVM 中加载的类及对动态检测方法的补充, 研究基于文本特征的深度学习静态检测算法, 提升高对抗内存型 Webshell 的检测效率。实验表明, 与其他检测工具相比, 本文方法检测内存型 Webshell 效果最佳, 准确率为 96.45%, 性能消耗为 7.74%, 具有可行性, 并且根据检测结果可以准确定位到内存型 Webshell 的位置。

关键词 内存型 Webshell; RASP; 动态检测; 静态检测
中图法分类号 TP309.5 DOI 号 10.19363/J.cnki.cn10-1380/tn.2022.11.04

Java-oriented High-adversarial Memory Webshell Detection Technology

Zhang Jinli[1,2], Chen Xingchen[1,2], Wang Xiaolei[1,2], Chen Qingwang[1,2], Dai Feng[1], Li Xianglong[1,2],
Feng Yun[1], Cui Xiang[1,3]

[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
[3] Cyberspace Institute of Advanced Technology, Guangzhou University, Guangzhou 510006, China

Abstract Web application has become one of the main targets of network attacks due to its complexity and importance. After an attacker invades a website, he usually implants a Webshell to control the website persistently. However, with the game between the offense and defense, various detection technologies and terminal security products are widely used, making the traditional Webshell residing in the form of file more and more easily detected, and the memory-based Webshell has become a new trend. Memory-based Webshell does not hava malicious files on disk, but injects malicious code into memory, which is more concealed and difficult to be detected by security devices, and currently there is a lack of detection technology for memory-based Webshell. For Java applications, this paper summarizes the characteristics and principles of memory-based Webshell, constructs a memory-based Webshell threat model, defines a high-adversarial memory-based Webshell, and proposes a high-adversarial memory-based Webshell detection technology based on RASP (Runtime application self-protection) and dynamic and static combination. To user requests, the register component functions and privileged functions are monitored based on RASP technology, and the context information is obtained. The dynamic feature detection is carried out in real-time according to whether there are files in the disk and data flow analysis technology, without affecting the normal operation of the application program. Aiming at the classes loaded in the JVM and the supplement to the dynamic detection method, a deep learning static detection algorithm based on text features is studied to improve the detection efficiency of high-adversarial memory-based webshell. Experiments show that, compared with other

通讯作者: 冯云, 博士, 工程师, Email: fengyun@iie.ac.cn。
本课题得到中国科学院青年创新促进会(No. 2019163); 中国科学院战略性先导科技专项项目(No. XDC02040100); 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: 2022-06-20; 修改日期: 2022-08-17; 定稿日期: 2022-09-07

detection tools, the method in this paper has the best effect in detecting memory-based Webshells, with an accuracy rate of 96.45%, and a performance consumption of 7.74%, which is feasible. Moreover, the location of the memory-based Webshell can be accurately located according to the detection results.

Key words memory webshell; RASP; dynamic detection; static detection

1 引言

当今世界, 互联网已渗透到人们生活的方方面面, 甚至影响着全球经济、政治、文化和社会的发展。Web 作为互联网上最典型、最主流的应用之一, 在社会活动、经济活动中承担着重要角色, 各种高价值的数据和信息经由 Web 接收、处理、存储和转发。Web 给人们带来可观经济效益和广泛社会影响的同时, 也成为网络攻击的主要目标。攻击者在入侵一个网站后, 通常会上传一个 Webshell 来持久化地控制 Web 服务器。

研究表明[1], 将近一半的攻击者在发现 Web 应用上传漏洞后会上传一个 Webshell。攻击者上传的 Webshell 与网站管理员远程管理所使用的功能类似, 通过 HTTP 请求接收命令并提供响应, 可穿越防火墙, 难以检测。Webshell 最初包含的功能强大, 但是文件较大、服务器端代码较多; 针对网站对文件大小上传的限制, 逐渐演变为代码量小、只有单一上传文件功能的小文件; 为了更加隐蔽, Webshell 配合管理工具, 只需简单的脚本执行语句即可; 为了更好的躲避检测, Webshell 还采用了加密、混淆等手段。但是随着近年来攻防双方的博弈, Webshell 获得了越来越多的关注, 其检测方法也更加健全, 包括文本特征检测、行为特征分析、流量检测、日志检测、统计学检测等方法, 并结合机器学习、深度学习等算法提高了检测的准确率, 以及 XDR(Extended Detection and Response, 扩展检测和响应)[2]等防御技术的提出, 使得传统的以文件形式落地的 Webshell 生存空间越来越小。因此, 内存型 Webshell 应运而生。

内存型 Webshell 也被称为无文件 Webshell, 是无文件攻击的一种。无文件攻击由来已久, 但是随着近年来攻防演练的热度再次被提起。据亚信安全的 2020 威胁情报态势分析[3], 病毒攻击的方式发生了很大的变化, 大多数采用无文件攻击技术, 报告数据显示, 在成功入侵的攻击事件中, 有 80%是通过无文件攻击完成的, 传统的防病毒软件对此攻击基本无效。无文件攻击属于一种影响力非常大的安全威胁, 攻击者不会在目标主机的磁盘上写入任何可执行文件, 而是通过各种脚本执行, 因此得名“无文件攻击”。无文件攻击执行后不会留下任何痕迹, 所以难以被检测和清除。

内存型 Webshell 是在内存中写入恶意后门, 不会有文件落地, 利用中间件的进程执行这些恶意代码, 达到远程并持久控制 Web 服务器的目的。早在 2017 年[4], 内存型 Webshell 已经初露头角, 研究人员通过调试 Tomcat 代码, 在运行时动态插入 Filter 和 Valve, 可以达到隐藏 shell 的效果, 只是这两种方法在 Tomcat 重启后会失效。2018 年[5], 研究人员利用 agent 技术通过进程注入方式实现了内存型 Webshell, 并且使用 ShutdownHook 机制实现服务重启后 Webshell 复活的功能, 进一步拓宽了内存马的使用场景。2020 年, 随着攻防演练的兴起, 内存型 Webshell 再次回归视野。乘着 Java 反序列化漏洞的东风, 内存型 Webshell 引发了更多的关注。2020 年[6], 研究人员基于特定框架 Java Spring MVC 的利用, 构造了一个 Spring Controller Webshell, 利用多种技术注入到内存中, 实现无文件攻击。2021 年[7], 研究人员利用 Shiro 的反序列化漏洞, 实现了内存型 Webshell 的注入及 Tomcat 通用回显。除此之外, 常用的 Webshell 管理工具, 如冰蝎、哥斯拉、蚁剑等, 都在最新版本中增加了内存型 Webshell 注入的功能。内存型 Webshell 已经成为了攻击方的必备工具, 针对内存型 Webshell 的检测和防护也越来越受重视。

目前针对内存型 Webshell 的攻击主要活跃在 Java Web 的框架、中间件等上, 且 Java 以其灵活、健壮、跨平台的特点, 是主流的编程语言。因此, 本文主要研究面向 Java 的内存型 Webshell 检测技术。通过收集近年来公开的内存型 Webshell 样本, 进行实验分析, 构建内存型 Webshell 模型, 提出一种高对抗内存型 Webshell, 结合 RASP 动态检测技术和深度学习静态特征检测技术, 对高对抗内存型 Webshell 进行检测。本文的主要贡献如下:

(1) 分析内存型 Webshell 原理, 总结其不同阶段的行为特征, 构建内存型 Webshell 的威胁模型, 并提出一种高对抗内存型 Webshell。

(2) 提出一种基于 RASP 技术和数据流分析的动态行为检测方法, 在不影响目标应用程序正常运行的情况下, 实时检测内存型 Webshell。

(3) 提出一种基于文本特征的深度学习静态检测算法, 与动态方法结合, 提升对高对抗内存型

Webshell 的检测效率。

(4) 设计多组实验, 对本文检测方法进行测试评估。实验表明本文方法检测高对抗内存型 Webshell 准确率达 96.45%, 性能消耗为 7.74%, 具有可行性。

本文的其余部分组织如下: 第二章介绍了相关工作; 第三章讨论了内存型 Webshell 的威胁模型; 第四章详细介绍了针对高对抗内存型 Webshell 的检测方法; 第五章是本文检测方法的实验评估与分析; 第六章是对本文方法的总结与展望。

2 相关工作

2.1 学术界 Webshell 检测方法

内存型 Webshell 是 Webshell 的一种, 学术界针对 Webshell 的检测方法有很多。从检测方式的角度分为静态检测和动态检测两类。从检测对象角度分为文本内容检测、流量检测、日志检测、行为特征检测、统计学检测等五类。如表 1 所示。

静态检测是不执行样本而是通过获取的检测对象进行分析, 挖掘对象的特征, 包括文本内容特征、日志访问特征、统计学特性等, 并结合机器学习[8-9]或深度学习[10-11]技术进行检测。静态检测又分为基于文本内容的检测、基于日志的检测、基于统计学的检测等方法。(1)基于文本内容的检测直接针对源代码进行特征工程, 通过提取词法、语法、语义等特征检测 Webshell。该方法对已知的 Webshell 检测准确率高, 快速方便, 部署简单, 但只能检测已知特征的 Webshell, 漏报率、误报率高。(2)基于日志的检测是针对 Webshell 访问网站后在 Web 日志中留下的页面访问数据, 通过对大量的日志文件建立请求模型从而检测出异常文件。该方法产生大量日志, 存在漏报, 且具有一定滞后性。(3)基于统计学的检测是通过一些统计值来区分正常文件与恶意 Webshell。重心在于识别混淆代码, 对未经模糊处理的代码检测机制较为透明, 误报漏报多。

Webshell 传到服务器后, 攻击者总要去执行它, 动态检测是根据 Webshell 执行时刻表现出来的行为模式、网络特性等进行检测。动态检测又分为基于

表 1 Webshell 检测相关工作总结

Table 1 Summary of work related to Webshell detection

检测方式分类	检测对象分类	研究进展	研究团队	存在问题
静态检测	文本内容检测	将 PHP 代码的可执行数据特性与静态文本特性结合起来进行 Webshell 分类	国防大学[12]	只能检测已知特征的 Webshell, 漏报、误报高, 无法检测内存型 Webshell
		源码侧特征提取的启发式方法和 RNN 算法相结合的检测模型	中国科学院大学[13]	
		提取句法和语法特征, 分析与外部通信、运行时环境检测、敏感操作等行为, 来检测 Webshell	莱特州立大学[14]	
	日志检测	基于时间间隔的统计方法来具体识别用户会话, 同时使用 LSTM 深度学习算法检测	四川大学[15]	产生大量日志, 存在漏报和滞后性, 无法检测内存型 Webshell
		基于服务器日志, 从文本特征、统计特征和页面关联特征 3 个角度检测 Webshell	四川大学[16]	
	统计学检测	NeoPI——Webshell 定量检测工具, 从信息熵、最长单词、重合指数、恶意特征、压缩比五个角度综合分析	Ben Hagen 等人[17]	重心在于识别混淆代码, 误报漏报多, 无法检测内存型 Webshell
		提取信息熵、重合指数、最长单词长度、压缩比等统计特征, 结合 php 操作码序列特征, 利用随机森林和梯度提升决策树结合的两层模型, 来检测混淆的 Webshell	电子信息工程学院[18]	
动态检测	流量检测	将 HTTP 请求中的 POST 内容使用 Word2vec 模型以向量的形式表示, 输入到 CNN 进行检测	陆军工程大学[19]	部署成本高, 无法检测加密流量内容, 无法检测内存型 Webshell
		将 HTTP 请求字段转换为长度为 300 的基于字符的向量, 结合 CNN 和 LSTM 算法构建模型	北京邮电大学[20]	
	行为检测	使用 FastText 在 PHP 操作码序列训练得到预测值, 再与统计特征结合作为随机森林检测模型输入	四川大学[21]	资源消耗较大, 检测效率较低, 没有针对内存型 Webshell 的检测
		分别使用 TF-IDF 向量化源码、操作码序列和 TF-IDF 向量化操作码用于 DNN 模型训练, 得到最优检测效果	兰州大学[22]	

流量的检测、基于行为的检测。(1)基于流量的检测对通信过程中Payload流量表现出的异常特征进行提取并分析, 从而实时检测网站变化。该方法部署成本高, 无法检测加密流量内容。(2)基于行为的检测是针对脚本在系统环境中的解析过程进行分析, 通常带有命令执行、文件操作和数据库操作等行为特征, 从而检测异常行为。该检测方法准确率相对较高, 但资源消耗较大, 检测效率较低。

目前学术界对内存型 Webshell 检测的研究极少, 已有的检测方法无法适用于内存型 Webshell。

2.2 内存型 Webshell 检测产品

攻防演练的热度升级备受工业界的关注, 许多工业界与时俱进地在产品中增加内存型 Webshell 的防御功能。

边界无限的靖云甲·RASP 作为一款网络攻防产品[23], 能够对内存型 Webshell 进行防御, 以 Java 为例, 该功能是通过RASP在程序内部获取API接口信息, 利用 Agent 周期性地对 JVM 内存中的 API 进行风险筛查, 然后上报存在风险的类。但是该方法存在一定滞后性, 不能实时检测内存型 Webshell 的注入及存在。

微步在线的 OneEDR 是一款针对主机的入侵检测与响应的终端安全防护平台[24], 利用 Agent 在终端收集系统行为日志, 并结合威胁情报, 实现对主机的入侵发现与响应。针对无文件攻击方式的频繁出现, OneEDR 集成了针对 Java 平台的内存型 Webshell 检测功能。针对 Agent 类内存型 Webshell, OneEDR 同样利用 Java Agent 技术对运行的 JVM 进行检测, 针对可疑高危的 Class 再做进一步检测。OneEDR 也没有对用户实时的请求进行内存型 Webshell 的检测。

河马是一款专注于 Webshell 查杀的检测工具[25], 拥有海量的 Webshell 样本及自主查杀的技术, 采用传统特征, 并结合深度检测、机器学习、云端大数据多引擎检测技术。河马 1.0 内测版本开始支持内存型 Webshell 的检测, 逐渐增加对冰蝎、哥斯拉内存型 Webshell 的检测功能。河马对内存型 Webshell 的检测也依赖于Java Agent技术, 将检测代码注入到Web服务的 Java 进程中对其进行恶意代码检测。河马目前还未发布官方版的内存型 Webshell 检测工具。

工业界针对内存型 Webshell 的检测产品虽然响应较快, 但还处于起步阶段, 检测方案并不全面, 还在持续优化过程中, 并且大多数产品没有开源。

2.3 Java RASP 技术

RASP [26]是一种注入到应用程序内部或应用程序运行时环境的安全技术, 与应用程序融为一体, 能够实时检测和阻断攻击。与传统的 WAF(Web Application Firewall)技术相比, RASP 技术准确性更高, 更可靠。WAF 是基于模式匹配并仅对所有输入流量进行检测, 而 RASP 是在发生攻击的关键节点处同时关注输入和输出, 能够根据数据流分析输入在应用程序内部的行为, 从而精准拦截攻击。RASP 技术已在 Web 安全及漏洞检测领域被广泛应用, 例如在 Web 安全检测[27]、脚本注入安全[28]、Web 框架漏洞检测[29]、智能合约漏洞防护[30]等方面得到了有效解决方案。

对 Java 而言, RASP 技术是通过 Java Agent 方式实现的。Java Agent 允许 JVM 在加载 class 文件之前, 对其字节码进行修改, 通过在程序启动前注入 RASP 逻辑; 同时也支持对已加载的 class 文件进行重新加载, 通过 attach 方式在程序运行时附加 Agent, 动态注入 RASP 逻辑。

Java RASP 的技术实现主要依赖于 JVMTI(JVM Tool Interface)、Instrumentation(Java Agent API)、字节码操作框架等。

JVMTI 是一套由 Java 虚拟机提供的本地编程接口集合, 可以提供 JVM 相关工具开发的接口, 可以对虚拟机内部状态进行监测、分析, 并能控制 JVM 应用程序的执行。JVMTI 的原理是在 JVM 内部的一些事件上进行了埋点, 外部程序通过实现一个 JVMTI Agent, 并将 Agent 注册到 JVM 中, 当事件被触发时, JVM 会回调 Agent 的方法来实现用户逻辑。并非所有 JVM 都支持 JVMTI, 但一些主流虚拟机如 Sun、IBM 等都提供了 JVMTI 实现。但是基于 JVMTI 的开发是通过 C/C++语言编写的 Agent 来实现的, 对于 Java 开发人员不太友好。

JDK 1.5 开始, Java 引入了 Instrumentation 机制。Instrumentation 是由 Java 提供的监测运行在 JVM 程序的 API, 可以基于 Java 编写 Agent 来监控或操作 JVM。Instrumentation 的底层实现是依赖于 JVMTI。Instrumentation 可以实现在已有的类上修改或插入额外的字节码来增强类的逻辑, 但这些实现不会改变原程序的状态或行为。Instrumentation 提供两种方式注入 Agent(即 RASP 逻辑), 一种是 premain 模式在主程序运行前注入, 另一种是 agentmain 模式, 在主程序运行时注入。

Java RASP 除了依赖 JVMTI 和 Instrumentation 来完成 Java Agent 的编写外, 还需要利用字节码操作框架来完成相应的 Hook 操作, 即对已经是字节码的 class 文件进行操作, 主要有两种工具, ASM 和

Javassist。ASM 字节码操作框架, 主要是在指令层面上操作字节码, 门槛高, 但体积小速度快, 性能好。Javassist 重点在源代码层面上操作字节码, 提供了源代码级 API 对字节码操作, 相比 ASM 操作简单, 但执行效率慢。

3 内存型 Webshell 威胁模型

3.1 内存型 Webshell 原理

内存型 Webshell 是无文件攻击的一种, 是将恶意代码注入到内存中, 利用中间件的进程执行某些恶意代码, 没有文件落地, 即服务器上不存在可远程访问的 Webshell 文件。其原理如图 1 所示, 攻击者首先构造与请求处理相关的恶意代码, 包括注册恶意组件的代码、修改已有类逻辑的代码, 通过漏洞等场景执行恶意代码, 成功添加恶意组件到 JVM 进程或修改内存中已有类的逻辑, 将恶意代码驻留在系统内存中, 因此得名内存型 Webshell。注入成功后, 攻击者通过访问特定(带参数的)URL, 达到命令执行、文件操作、数据库操作等远程控制 Web 服务器的目的。Java 的反射机制和 Instrumentation 机制为内存型 Webshell 提供了良好的植入方式。根据内存型 Webshell 的利用技术不同, 将其分为两类, 一类是组件类(也称为非 Agent 类), 一类是 Agent 类。

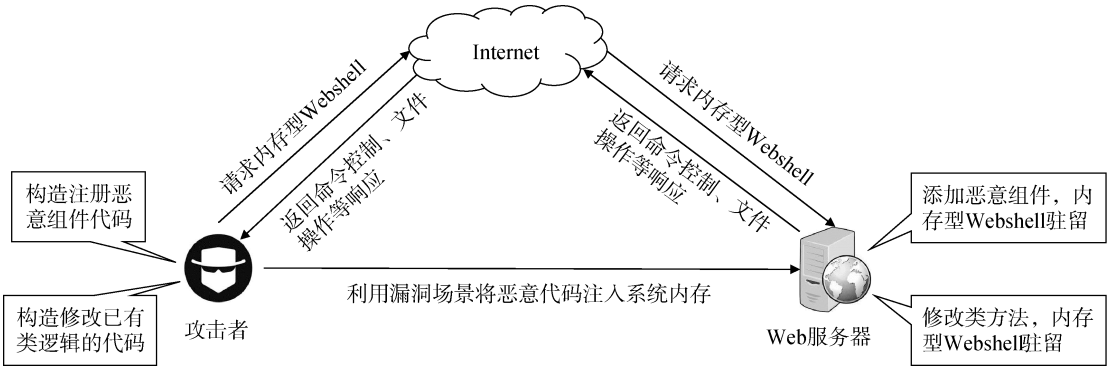

图 1 内存型 Webshell 原理

Figure 1 Memory Webshell principle

(1) 组件类

组件类的内存型 Webshell 主要是通过动态注册一个包含恶意代码的组件到运行内存中, 将恶意代码长期驻留在内存中。常见的组件包括 Java Web 的三大组件, 分别为 Filter、Servlet、Listener。除此之外, 还包括特定框架和容器的特有组件, 例如 Spring 框架的 Controller、Interceptor 组件, Tomcat 容器的 Valve 组件等。

利用场景一般通过反序列化等漏洞直接注入恶意组件到内存, 或通过文件上传漏洞上传一个 JSP 文件(包含恶意组件注入程序), 执行后删除文件, 达到无文件 Webshell 的效果。该类型 Webshell 要求目标环境可以执行任意 Java 代码, 对目标环境依赖性较强, 通用性差。

其实现过程如图 2 所示。首先创建一个组件, 添加恶意逻辑到其中, 然后通过反序列化、代码执行等漏洞将注册组件代码执行, 注入到特定环境的目标应用程序中, 最后通过访问固定 URL 或带固定参数的 URL 来执行 Payload, 达到 Webshell 的特权状态。

(2) Agent 类

Agent 类的内存型 Webshell 主要是利用 Java Agent 技术动态修改 JVM 中加载的类文件, 将恶意代码注入其中, 该类在磁盘上还是正常文件, 只有在运行内存中才会包含恶意代码, 达到无文件落地。

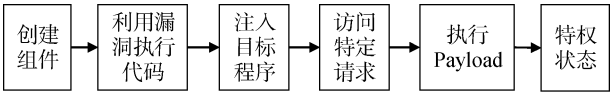

图 2 组件类 Webshell 实现过程

Figure 2 Components type Webshell implementation process

Agent 类的内存型 Webshell 也包括 premain 和 agentmain 两种注入方式, 但是实际环境中, 目标应用程序一般都是已启动状态, 只能使用 agentmain 的方式注入。利用场景通常是通过漏洞上传一个 Jar 文件, 执行成功后删除该文件。该类型 Webshell 在 Java 虚拟机层操作, 通用性强。

其实现过程如图 3 所示。首先通过漏洞上传一个 Jar 包, 用于加载 Java Agent 程序, 将 Jar 包注入到目标程序运行内存后, 就可以删除该文件。然后通过访问固定 URL 或带固定参数的 URL, 恶意逻辑会利用 Agent 技术修改类字节码, 再执行 Payload, 达到

Webshell 的特权状态。

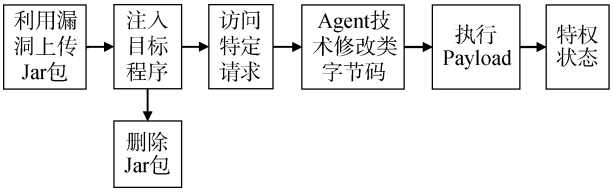

图 3　Agent 类 Webshell 实现过程

Figure 3　Agent type Webshell implementation process

(3) 两种类型异同

组件类和 Agent 类的内存型 Webshell, 相同之处都是利用漏洞将恶意程序注入目标程序内存中, 然后通过访问特定的请求连接 Webshell, 执行 Payload, 达到各种控制服务器的特权状态的目的。

不同之处在于, 前者有创建并注册组件的动态过程, 新增了一个组件, 组件中包含恶意逻辑; 而后者是利用 Agent 技术直接修改 JVM 中已经存在的类的逻辑, 不产生新的类。

3.2　形式化定义

内存型 Webshell 是一种网站后门, 根据研究人员[31-32]对后门的定义, 可以将后门分为输入源、触发器、攻击载荷、特权状态 4 个组件。本节从这 4 个组件出发, 结合内存型 Webshell 的原理, 对其进行形式化定义。

如图 4 所示, 目标程序中存在内存型 Webshell(θ) 可表示为:

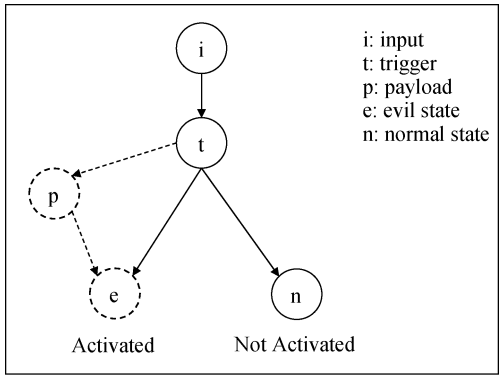

图 4　后门模型

Figure 4　Backdoor model

$$\theta = (S, i, e, t, \Sigma, p),\ i \in S \text{ 且 } e \in S$$

S 表示目标程序的全部有限状态集合。I 表示输入源, 即初始状态集合, $I \subseteq S$。E 表示内存型 Webshell 触发后到达的特权状态集合, $E \subseteq S$。t 表示状态转移函数, 也是触发器, 有 $t(I, c) = S$, c 表示状态转移条件。Σ 表示后门触发条件。p 为攻击载荷, 表示为达到特权状态而采用的有效代码。特别地, 当存在 $i \in I$, $c = \Sigma$, $e \in E$, 且攻击载荷 p 被执行, 使得 $t(i, \Sigma) = e$ 时, 表示后门被触发, 存在内存型 Webshell。n 表示后门没有被触发时的正常转移状态, $n \in S$。

输入源表示激活后门触发器的输入来源。内存型 Webshell 的输入源 I 包括(1)Java Web 组件, 例如 servlet、filter、listener 等; (2)Java 容器组件, 例如 tomcat valve 组件、weblogic 的组件; (3)Java 框架组件, Java 的框架有很多, 包括 spring、springboot 等, 例如 spring 框架的 controller 组件; (4)二进制, 例如 Agent 类内存型 Webshell 的字节码。

触发器在满足一定条件的情况下, 会执行攻击载荷, 进而达到某种特权状态。内存型 Webshell 的触发器一般为反序列化漏洞、文件上传漏洞、RCE 漏洞等。

攻击载荷 p 是一段有效功能代码, 是 Webshell 后续控制服务器的关键部分, 决定了特权状态的存在形式。

内存型 Webshell 触发后到达的特权状态集合 E 与一般的 Webshell 的特权状态相同, 都是为了达到远程控制服务器的目的, 包括系统命令执行、任意操作服务器上的文件、读取数据库数据等状态。

通过对内存型 Webshell 的原理分析及形式化定义, 本文将从内存型 Webshell 实现过程中的输入源、触发器、攻击载荷、特权状态 4 个方面进行分析, 实现内存型 Webshell 的检测。

3.3　威胁模型

内存型 Webshell 具有持久性、隐蔽性, 并且将在上节提到的 4 个组件中不断增强, 本节从攻击载荷、触发器两方面进行分析。

(1) 攻击载荷

目前公开的内存型 Webshell 样本基本都是明文的 Payload, 主要关注点在 Webshell 存在于内存中而非磁盘上, 过于关注输入源、触发器的状态, 而忽略了攻击载荷的形态。随着无文件检测技术的不断发展和提升, 内存型 Webshell 将会在 Payload 上做更多的变形。

我们定义高对抗内存型 Webshell, 为具备加密、编码、混淆等免杀功能的高级内存型 Webshell。我们假设攻击者会使用高级具备免杀功能的内存型 Webshell 样本, 因此本文主要检测高对抗内存型 Webshell 攻击。由于高对抗内存型 Webshell 是基于内存型 Webshell 对攻击载荷的改造升级, 本文的静态检测方法主要检测高对抗内存型 Webshell 攻击,

动态检测方法不对攻击载荷进行分析, 但依然能够对高对抗内存型 Webshell 的输入源、触发器、特权状态等行为特征进行检测, 因此文本主要面向高对抗内存型 Webshell 进行检测。

攻击者通常使用流行的冰蝎、哥斯拉等工具快捷地注入 Agent 类内存型 Webshell, 并且注入的样本所属的包名为工具默认的, 例如 net.rebeyond、com.metasploit 等, 容易被检测到。我们假设高对抗内存型 Webshell 攻击者会修改默认的常见工具包名以绕过简单的字符串匹配等检测。我们避免因包名过滤造成的漏报, 而从其他方面进行检测。

(2) 触发器

我们假设攻击者在植入组件类高对抗内存型 Webshell 时, 触发器的场景包括 1)通过反序列化漏洞等注入恶意字节码, 2)通过 RMI、LDAP 等远程方法调用恶意类, 3)通过文件上传等漏洞注入恶意 JSP 文件。前两种方式是无文件攻击, 目标应用程序的磁盘没有恶意文件落地, 但是也考虑攻击者故意写入磁盘文件的情况, 以绕过无文件这一特性。第三种方式的 JSP 文件在执行过程中, 会先被编译成 class 文件, 然后再执行程序, 即使执行完可以删除文件达到无文件攻击, 但动态检测是在执行过程中进行的, 执行过程中不可避免的有文件落地。检测时需针对以上情况分别考虑。

4 高对抗内存型 Webshell 检测技术

本节针对内存型 Webshell 的原理和特征, 提出一种基于 RASP 技术的动静态结合的高对抗内存型 Webshell 检测方法, 其总体流程如图 5 所示。首先对应用程序进行监测, 在有用户请求的情况下, 利用 RASP 监测技术获取监测函数的上下文信息。根据获取的行为特征, 结合磁盘是否存在文件、黑白名单过滤、数据流分析技术进行动态特征检测; 获取 JVM 中所有加载的类, 通过特征过滤筛选出高危类, 利用深度网络 ResNet50 分类模型进行静态文本特征检测。动态检测中监测到的非 JS 场景的新增组件类及 defineClass 函数输入到静态检测中, 动静态检测方法结合判断请求或系统内部是否存在高对抗内存型 Webshell。最后根据判定结果进行处置并发出告警信息。

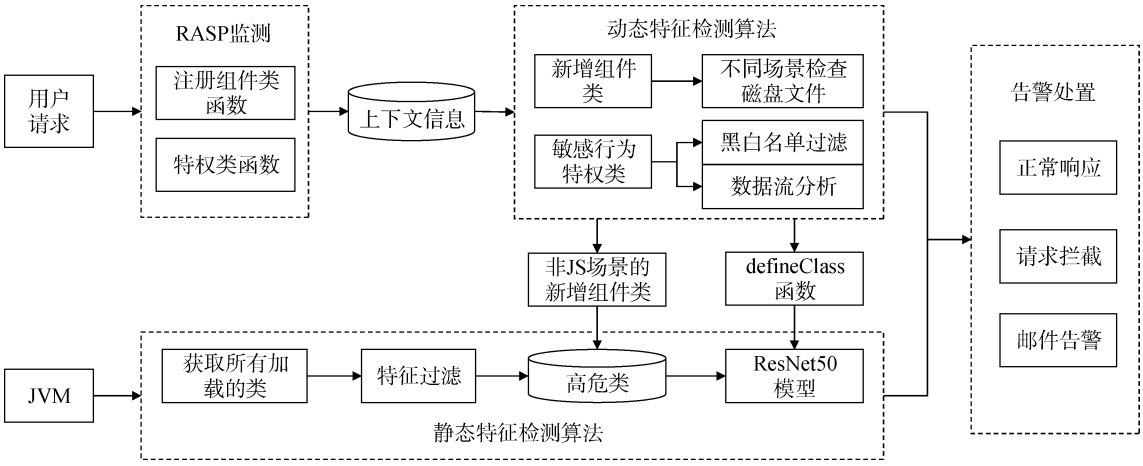

图 5 高对抗内存型 Webshell 检测框架

Figure 5 High-adversarial memory Webshell detection framework

4.1 RASP 监测技术

RASP 技术是一种动态实时监测技术, 可以在不影响目标应用程序正常运行的前提下, 对其请求过程中的函数进行字节码增强。RASP 通过 Hook 函数对其设置探针来进行监测, 从而收集函数运行时的上下文信息, 包括函数参数、函数返回值、函数调用栈等信息。针对内存型 Webshell, RASP 主要监测两类函数, 一类是与输入源中的组件相关的注册组件类函数, 另一类是为达到特权状态所请求的函数, 称为特权类函数。结合内存型 Webshell 的原理可知, 前者主要关注其在注入过程中的函数请求, 后者主要关注注入成功后访问 Webshell 时的函数请求。

(1) 注册组件类函数

内存型 Webshell 的输入源中包含组件, 该类型的 Webshell 需要依靠组件将 Webshell 驻留在运行内存中, 所以会通过调用注册组件类的方法以达到目的, 而其他的正常行为或攻击行为通常不会通过该方式添加一个组件。注册组件类函数通过调用某个类可以直接在内存中生成一个新的组件, 不同类型的组件注册的方式不同, 调用的函数也不同。以

Tomcat 添加 Filter 为例, 在内存型 Webshell 的实际环境中, Filter 并不会按照在 web.xml 中注册的方式添加, 而是通过反射机制进行动态注册。

注册 Tomcat 的 Filter 组件关键步骤是:

1) 通过反射获取 ServletContext 对象。

2) 实例化一个包含 Payload 的恶意 Filter 对象。

3) 利用 FilterDef 对 Filter 进行封装, 定义其名称和类名等, 将 FilterDef 添加到 FilterDefs 中。

4) 创建 FilterMap, 设置 Filter 和拦截的 URL 的映射关系, 并将该 FilterMap 作为第一个过滤器添加到 FilterMaps 中。

5) 创建 FilterConfig, 封装 FilterDef 对象, 并添加到 FilterConfigs 中。

6) 最后把 FilterDefs、FilterMaps、FilterConfigs 注入到 StandardContex 即可。

整个过程中与 Filter 组件注册紧密相关的函数在第 3 步中, 如图 6 所示, 包括 setFilter() 和 setFilterClass()方法, 这两个方法可以通过参数信息分别获取实例化的 Filter 对象和 Filter 的类名。

```
FilterDef filterDef = new FilterDef();
filterDef.setFilter(filter);
filterDef.setFilterName(name);
filterDef.setFilterClass(filter.getClass().getName());
standardContext.addFilterDef(filterDef);
```

图 6 Filter 组件注册相关函数

Figure 6 Filter component registers related functions

注册组件的方法根据不同容器或框架使用不同的函数, 这些方法根据参数等信息可获取新增的组件类名、类实例等信息。通过 Hook 这些函数, 可以在内存型 Webshell 注入过程中就检测和拦截。表 2 总结了部分注册组件类函数监测点。

(2) 特权类函数

内存型 Webshell 与普通的 Webshell 相似, 最终目标都是持久化控制目标服务器, 达到一定的特权状态。该特权状态包括任意系统命令执行、操作服务器文件、对数据库操作等恶意行为状态, 涉及的特权类函数包括命令执行函数、文件操作函数、数据库操作函数、编解码函数等敏感操作函数。通过 Hook 相关函数, 可以检测和拦截 Webshell 对系统造成危

表 2 注册组件类函数

Table 2 Register component class functions

组件类型	监测类#方法
Filter	org.apache.tomcat.util.descriptor.web.FilterDef#setFilter org.apache.tomcat.util.descriptor.web.FilterDef#setFilterClass org.apache.catalina.core.ApplicationContextFacade#addFilter weblogic.servlet.internal.FilterManager#registerFilter weblogic.utils.collections.ConcurrentHashMap#put
Servlet	org.apache.catalina.Wrapper#setServlet org.apache.catalina.Wrapper#setServletClass weblogic.servlet.internal.ServletStubImpl#ServletStubImpl weblogic.servlet.internal.URLMatchHelper#URLMatchHelper weblogic.servlet.utils.ServletMapping#put weblogic.utils.collections.ConcurrentHashMap#put
Listener	org.apache.catalina.core.StandardContext#addApplicationEventListener org.apache.catalina.core.StandardContext#applicationEventListenersList weblogic.servlet.internal.EventsManager#createListener weblogic.servlet.internal.EventsManager#addEventListener weblogic.servlet.internal.WebAppServletContext#registerListener weblogic.servlet.internal.WebAppServletContext#addListener
Valve	org.apache.catalina.Pipeline#addValve org.apache.catalina.core.StandardContext#addValve
Controller	org.springframework.web.servlet.handler.AbstractHandlerMethodMapping$MappingRegistry#register org.springframework.web.servlet.mvc.condition.PatternsRequestCondition#PatternsRequestCondition org.springframework.web.servlet.mvc.condition.RequestMethodsRequestCondition#RequestMethodsRequestCondition org.springframework.web.servlet.mvc.method.RequestMappingInfo#RequestMappingInfo org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping#registerMapping
Interceptor	org.springframework.core.MethodParameter#set org.springframework.core.MethodParameter#MethodParameter java.util.List#add java.util.ArrayList#add

表 3 特权类函数

Table 3 Privileged functions

类型	监测类#方法
命令执行	java.lang.ProcessImpl#start java.lang.ProcessBuilder#start java.lang.Runtime#exec java.lang.reflect.Method#invoke
文件操作	org.apache.commons.fileupload.FileUploadBase.FileItemIteratorImpl.FileItemStreamImpl#<init> javax.servlet.http.Part#getInputStream java.io.File#delete java.nio.file.Files#newInputStream/newOutputStream/newBufferedReader/newBufferedWriter java.io.FileOutputStream#<init> java.io.FileInputStream#<init>
数据库操作	java.sql.Connection#prepareStatement java.sql.Statement#executeQuery java.sql.DriverManager#getConnection java.sql.Driver#connect javax.naming.spi.DirectoryManager#getObjectInstance
编解码函数	java.util.Base64#Decoder java.util.Base64#Encoder
特殊函数	java.lang.Runtime#addShutdownHook weblogic.management.configuration.DomainMBean#createShutdownClass weblogic.management.configuration.DomainMBean#createStartupClass java.lang.ClassLoader#defineClass

表 4 请求类函数

Table 4 Request functions

类型	监测类#方法	上下文信息
请求函数	javax.servlet.http.HttpServlet#service	HttpServletRequest 对象(请求 IP、URI、请求方法、请求头、时间戳等) HttpServletResponse 对象(响应内容)

害行为的操作。除此之外, 针对内存型 Webshell 的特性, 还包括一类特殊函数。例如, 一般情况下, 内存型 Webshell 会随着组件的生命周期在 Web 服务器关闭或重启后, 也会随之销毁, 但是攻击者针对这一缺陷, 通过添加 JVM 的关闭钩子 ShutdownHook, 使得服务器在关闭时启动一个线程将内存型 Webshell 落地磁盘, 当服务器重启后, 磁盘文件自动执行再删除, 内存型 Webshell 就可以长期驻留在运行内存中。在 startUpClass 中植入内存型 Webshell, 同样可以做到持久化的效果。还有些内存型 Webshell 为了更隐蔽地达到特权状态, 在新增的组件类或 Agent 方式修改的类中通过 defineClass 方法加载字节码文件产生一个新的类, 在新的类中注入恶意代码, 使得检测难度增加, 攻击更加隐蔽。

通过监测特权类函数可以获取其上下文信息, 例如命令执行函数可以获取执行的系统命令, 文件操作函数可获取操作的文件类型、文件名, 数据库操作函数可获取执行的数据库语句, 编解码函数可获取编解码的内容, 特殊函数可获取类线程、类名、执行的类字节码等信息。表 3 总结了部分特权类函数监测点。

除了对注册组件类函数和特权类函数监测外, RASP 还需要监测 Web 请求类函数。一方面组件类的内存型 Webshell 一般是在有请求的情况下发生的, 需要对请求进行标记; 另一方面需要获取 Web 请求的上下文信息用于告警处置。表 4 是请求类函数及其上下文信息。

4.2 动态特征检测算法

根据内存型 Webshell 的原理, 一般会注册一个新的组件到内存中, 达到磁盘上无文件的目的。所以可以根据新增组件和磁盘无文件两个特性判断是否为内存型 Webshell。因为正常的服务请求或者其他的攻击行为一般不会新增一个 Java Web 组件, 这是内存型 Webshell 的独有特性。可以根据 RASP 监测的注册组件类函数, 获取上下文信息, 从拦截的函数参数中获取新增的组件类。然后检测磁盘上是否存在对应类文件, 根据触发器的不同场景, 分 3 种情况讨论。1)如果新增组件是通过字节码注入或远程方法

调用实现, 并且磁盘上没有对应类文件, 则判定为内存型 Webshell。2)新增组件是通过字节码注入或远程方法调用实现, 但是, 有的攻击者为了规避磁盘无文件这一特征, 故意将字节码文件写入到 ClassPath 路径下, 并在 web.xml 文件中配置相关信息, 来绕过检测。针对这种情况, 如果检查到磁盘上存在新增的组件类, 不能确定为非内存型 Webshell, 需要针对存在的新增类, 结合静态特征检测进一步判断, 将在下一节讨论。3)新增组件是通过 JSP 文件实现, 则 JSP 文件执行时, 即新增组件注册时, 一定会产生磁盘类文件, 动态检测是在新增组件注册时进行检测的, 无需检查磁盘是否有文件, 直接判定为内存型 Webshell。

内存型 Webshell 在注入到目标应用程序之后, 也跟正常 Webshell 一样, 最终会触发特权类函数来操控目标服务器, 例如操作文件、执行系统命令等行为或执行关闭钩子函数使 Webshell 更加持久化。但是对于应用程序内部也有可能执行特权类函数, 所以需要根据 RASP 监测获取的上下文信息判断是否为具有敏感行为的特权类函数以及是否有请求。应用程序内部的特权类函数调用, 例如命令类函数调用是程序内部函数的相互调用, 而内存型 Webshell 通常是在有请求的条件下进行的。通过设置黑白名单进一步过滤敏感行为, 如文件类特权函数设置上传文件后缀白名单, Webshell 常访问的路径范围设置黑名单等。即使是敏感的行为操作也可能是其他攻击类型, 根据内存型 Webshell 的特性, 设计动态特征检测算法, 如图 7 所示。

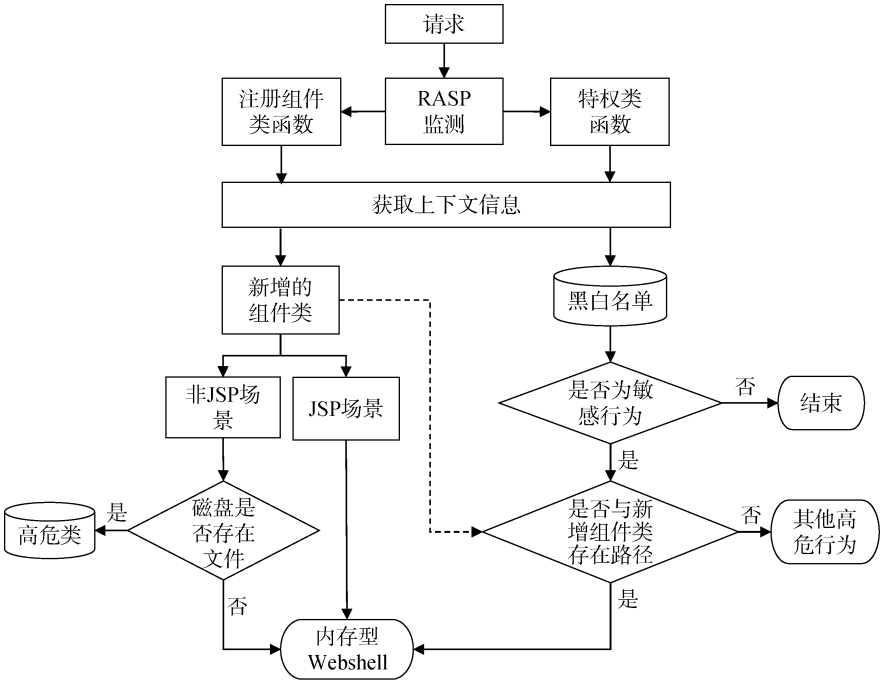

图 7 动态特征检测算法

Figure 7 Dynamic feature detection algorithms

动态特征检测是在有请求的情况下对其行为进行检测的, 通过 RASP 对注册组件类函数和特权类函数的监测来检测内存型 Webshell。当请求发生时, 如果触发了注册组件类函数, 则获取其上下文信息, 主要通过参数信息分析得到新增的组件类, 结合无文件的特性, 判断磁盘上是否存在新增的类文件。针对非 JSP 文件注册的组件类, 如果不存在相应的类文件, 则判定该组件类即为内存型 Webshell, 如果存在, 则将该组件类标记为高危类需要进行静态特征检测。针对 JSP 文件注册的组件类, 无需检测类文件是否存在, 直接判定该组件类即为内存型 Webshell。

如果在请求下触发了除 defineClass 之外的特权类函数的探针, 首先根据参数等上下文信息进行黑白名单的过滤, 如果是敏感行为, 则进一步获取调用栈等信息, 利用自下而上的数据流分析技术[33-34]计算特权类函数到注册组件类函数新增的组件之间是否可达。如果可达, 根据内存型 Webshell 的原理, 说明该后门首先通过注册组件类函数注册了一个组件到内存中, 然后再次利用了该后门并触发了特权类函数, 则判定新增的组件类为内存型 Webshell; 如果不可达, 则极有可能是其他危险类攻击触发了特权类函数。如果是 defineClass 的特权类函数, 不能根

据参数等上下文信息直接判断是否为敏感行为, 需要由下一节的静态特征检测算法进一步根据内容进行检测。

动态特征检测算法的伪代码如算法 1 所示。输入包括请求上下文、注册组件类函数的上下文、特权类函数的上下文, 以及用于过滤敏感行为的黑白名单列表。输出为算法的判定结果。

算法 1. 动态特征检测算法.

输入: 请求上下文 *reqContext*、注册组件类函数上下文 *compContext*、特权类函数上下文 *priContext*、黑白名单列表 *list*

输出: 判定结果 *result*

```
1: IF !Empty(reqContext)
2:     IF !Empty(compContext)
3:         获取新增组件类 newCompClass
4:         IF ClassLoader.getResource (newCompClass) == jsp 文件
5:             RETURN true, result(Webshell)
6:     ELSE IF ClassLoader.getResource (newCompClass)没有获取到类文件
7:             RETURN true, result(Webshell)
8:         ELSE
9:             newCompClass 标记为 riskClasses
10:            静态特征检测(riskClasses)
11:        END IF
12:    END IF
13:    IF !Empty(priContext)
14:        Get priContext.参数
15:        Get priContext.调用栈
16:        IF priContext.参数 MATCH list
17:            IF priContext.调用栈 Contains newCompClass
18:                RETURN true, result(Webshell)
19:            ELSE
20:                RETURN true, result(Other)
21:            END IF
22:        ELSE
23:            RETURN false
24:        END IF
25:    END IF
26:END IF
```

4.3 静态特征检测算法

RASP 针对内存型 Webshell 的特权状态以及组件类内存型 Webshell 的输入源进行了监测, 并根据其特性进行了动态特征检测。由于 Agent 类内存型 Webshell 不会动态注册组件植入后门, 而是直接修改 JVM 中已经存在的类, 因此针对 Agent 类内存型 Webshell 需要获取 JVM 中已加载的类文件, 然后结合静态特征对其攻击载荷进行检测。

由于 JVM 中加载的类很多, 如果全部获取并检测则会影响检测效率, 因此根据内存型 Webshell 的原理和特性, 我们尝试在正常请求的完整函数调用栈上使用 javassist 框架对函数依次进行 Agent 字节码修改, 修改成功的类可能被攻击者利用为内存型 Webshell 攻击, 经过多轮请求测试, 提取了 JVM 中五种类型的高危类, 包括类名、父类、接口、ClassLoader、注解。

(1) 类名, 包含容易被修改的类的名字以及 Agent 类 Webshell 常使用的特殊的类名字。

(2) 父类, Webshell 如果是自定义的类, 则需要继承父类以重写相应的方法执行恶意功能。

(3) 接口, 内存型 Webshell 为了最终能够执行恶意功能, 会实现某些接口。

(4) ClassLoader, 内存型 Webshell 使用的 ClassLoader 一般与反序列化、代码执行等漏洞相关, 与正常的类加载器有所区别。

(5) 注解, 内存型 Webshell 实现时会使用一些注解, 例如 Spring 的 Controller 类型的 Webshell 通常使用 Spring 的注解来声明。

表 5 列举了 JVM 中部分内存型 Webshell 相关的高危类。

静态特征检测算法如图 8 所示。首先获取 JVM 中加载的所有类, 然后根据特征筛选出 5 种类型的高危类, 对高危类进一步检测。除此之外, 高危类还包括动态特征检测中, 通过请求新增的组件类, 并且该组件类是通过非 JSP 注册的在磁盘上存在 class 文件的情况, 该新增的组件类也被归为高危类。

另外动态特征检测过程中, 如果触发的特权类函数是 defineClass 函数, 则根据 RASP 监测获取其上下文信息, 根据其加载的字节码文件进一步检测。

为了识别高对抗内存型 Webshell 的攻击载荷, 对高危类或 defineClass 获取的字节码文件进行深度学习模型检测。使用深度学习, 通过学习文件中的鲁棒性特征, 实现对目标文件快速准确的识别。本文使用深度模型的核心思想是将文件转化成灰度图像进行学习, 从图像的角度捕获文件中的恶意特征, 从而实现对 Webshell 的识别。由于内存型 Webshell 与普通 Webshell 的 Payload 基本一致, 而内存型 Webshell 的样本较少, 因此样本集中包含了两者, 并

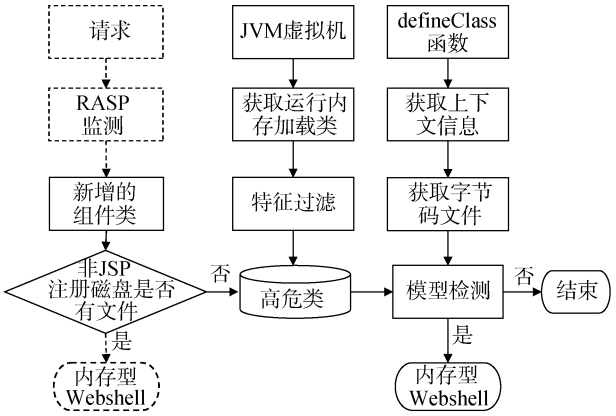

图 8　静态特征检测算法

Figure 8　Static feature detection algorithms

且包含了具备免杀功能的普通 Webshell 样本, 以及自定义编写的具备免杀功能的内存型 Webshell 样本。最后深度模型所使用的数据集包含 606 个 Webshell 样本和 819 个白样本, 样本格式为“.jsp”或“.java”。整个数据集按照 8:2 划分训练集和测试集, 使用经典的深度网络 ResNet50[35]作为分类模型。

模型训练之前, 先进行数据预处理。将样本以二进制的形式读取, 并以灰度图的形式保存。最终数据集中的每一个文件都对应一张灰度图, 用灰度图进行模型训练和测试。在训练阶段, 用训练集对应的灰度图进行网络的学习, 使用 Adam 优化器训练 150 轮次(Epoch), 其中学习率(Learning_rate)设置为 0.001、批量大小(Batch_size)为 32。每训练 1 个轮次, 都使用测试集对应的灰度图进行测试, 并记录准确率。经过不断的学习, 深度模型具备了对 Webshell 样本图像和白样本图像的分类能力。保存测试结果最优的那一轮次的模型用于最终任务的检测。在实际检测中, 首先将要检测的“.class”文件或字节码文件反编译为“.java”文件, 然后将“.java”文件转化成的灰度图输入深度检测模型中, 模型输出是 Webshell 样本或白样本。

静态特征检测算法的伪代码如算法 2 所示。以 JVM 中加载的所有类作为输入, 以筛选出高危类, 同时高危类中也包含动态检测过程中输出的高危类; 输入还包括 defineClass 函数中的字节码文件。最后输出为算法的判断结果。

算法 2. 静态特征检测算法.

输入: JVM 加载类 *classes*、defineClass 的字节码文件 *bytecode*

输出: 判定结果 *result*

```
1: Get JVM all classes
2:     FOR class in classes
3:         IF class MATCH 特征过滤
4:             class 标记为高危类 riskClasses
5:         END IF
6:     END FOR
7: FOR class in riskClasses
```

表 5　内存型 Webshell 高危类

Table 5　High-risk classess of memory Webshell

类型	实例
类名	org/springframework/web/servlet/handler/AbstractHandlerMapping
父类	javax/servlet/http/HttpServlet org/apache/catalina/valves/ValveBase
接口	javax/servlet/Filter javax/servlet/FilterChain javax/servlet/Servlet javax/servlet/ServletRequestListener org/apache/coyote/Adapter
ClassLoader	com/sun/org/apache/xalan/internal/xsltc/trax/TemplatesImpl$TransletClassLoader com/sun/org/apache/bcel/internal/util/ClassLoader java/net/URLClassLoader java/lang/ClassLoader
注解	org.springframework.stereotype.Controller org.springframework.web.bind.annotation.RestController org.springframework.web.bind.annotation.RequestMapping org.springframework.web.bind.annotation.GetMapping org.springframework.web.bind.annotation.PostMapping org.springframework.web.bind.annotation.PatchMapping org.springframework.web.bind.annotation.PutMapping org.springframework.web.bind.annotation.Mapping

```
8:    模型检测 f()
9:        RETURN result
10:EDN FOR
11:IF !Empty(bytecode)
12:   模型检测 f()
13:       RETURN result
14:END IF
```

4.4 告警处置

告警处置用于对内存型 Webshell 检测结果进一步处理。针对有请求的动态检测结果, 在 RASP 监测模块已经对请求类函数进行了监测, 如果检测结果为正常, 则返回原始响应页面, 如果检测结果为内存型 Webshell 或者其他高危行为, 则对请求拦截并返回一个自定义的页面, 同时使用 SimpleEmail[36]发送邮件告警的通知。针对 JVM 加载类的静态检测结果, 直接通过邮件方式发出告警通知管理人员。

告警通知的内容根据两种检测结果的不同, 如表 6 所示。

表 6 告警信息内容

Table 6 Alarm information content

类型	告警信息
动态检测结果	请求时间戳
	唯一识别码 UUID
	检测方式(动态检测)
	攻击类型
	攻击信息
	RASP 上下文信息
	堆栈信息
静态检测结果	检测时间戳
	唯一识别码 UUID
	检测方式(静态检测)
	攻击类型
	攻击信息
	高危类

5 实验与分析

本章通过内存型 Webshell 请求和正常请求对本文提出的基于 RASP 动静态结合的高对抗内存型 Webshell 检测方法进行评估。从准确率、精确率、召回率、F1 值 4 个指标评估本文方法的检测能力, 并结合实际利用场景对两种类型 Webshell 的检测结果分析, 从时间开销和资源消耗两方面评估本文方法的检测性能, 最后对检测方法进行讨论。

5.1 实验环境与数据

本文的检测方法是运行时检测, 因此需要针对易被内存型 Webshell 注入的 Java 常见中间件和框架搭建 Web 测试环境进行测试, 包括 Tomcat、Spring Web、Weblogic 等环境。测试环境使用 1 台服务器, 操作系统为 Ubuntu 20.04, 配置为双核处理器和 16G 内存。

实验采取的数据集来自于 GitHub 上的样本, 以及 Webshell 的管理工具, 例如冰蝎、哥斯拉等。样本覆盖组件类和 Agent 类的内存型 Webshell, 组件类包括 Filter、Servlet、Listener、Valve、Controller、Interceptor 等。部分数据集来源如表 7 所示。

5.2 检测能力分析

本实验对正常请求和内存型 Webshell 的请求分别测试, 从多个检测指标检测本文方法的检测能力。内存型 Webshell 请求分为两部分, 分别对其注入过程和利用过程进行测试。组件类是实时检测, 能够检测注入过程, Webshell 注入成功后会被再次访问, 所以包含注入过程和利用过程两部分。Agent 类是静态检测, 只包含利用过程。注入过程包括组件类的 31 个样本, 利用过程包括组件类和 Agent 类的全部样本, 共 40 个, 目前 Agent 样本比较少, 实验数据共 9 条。内存型 Webshell 共 71 个样本; 正常请求共 70 个样本, 如表 8 所示。本次实验对 Tomcat 容器、Spring 框架、Weblogic 容器进行了测试。

实验从准确率(Accuracy)、精确率(Precision)、召回率(Recall)、F1 值(F1-score)4 个指标测试评估。这些指标由以下 4 个基础指标计算, 如表 9 的混淆矩阵所示。左侧表示样本的真实值, 上侧表示样本的预测值。TP(True Positive)表示 Webshell 被正确检测为 Webshell 的个数; FN(False Negative)表示 Webshell 被错误检测为正常请求的个数, FP(False Positive)表示正常请求被错误检测为 Webshell 的个数; TN(True Negative)表示正常请求被正确检测为正常请求的个数。

准确率表示检测正确的结果占总样本的百分比, 即预测正确的概率, 计算公式如下:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

精确率是检测为 Webshell 的样本中实际为 Webshell 的概率, 表示对 Webshell 检测的准确程度, 计算公式如下:

$$Precision = \frac{TP}{TP + FP}$$

召回率是实际为 Webshell 的样本中被检测为 Webshell 的概率, 计算公式如下:

Precision 和 Recall 都越高越好, 但两者是一对矛

表 7 部分数据来源

Table 7 Partial data sources

类型	来源
注册组件类	https://github.com/jweny/MemShellDemo https://github.com/ce-automne/TomcatMemShell https://github.com/bitterzzZZ/MemoryShellLearn https://github.com/dk47os3r/SpringMemShell https://github.com/su18/MemoryShell
Agent 类	https://github.com/WisteriaTiger/JundeadShell https://github.com/keven1z/weblogic_memshell https://github.com/rebeyond/Behinder https://github.com/threedr3am/ZhouYu

表 8 实验数据

Table 8 Experimental data

类型	分类	个数
注入过程	Servlet-api(Filter、Servlet、Listener) Tomcat(Valve) Srping(Controller、Interceptor) Weblogic	31
利用过程	组件类	31
	Agent 类	9
正常请求	/	70

表 9 混淆矩阵

Table 9 Confusion Matrix

	Positive	Negative
Positive	TP	FN
Negative	FP	TN

$$Recall = \frac{TP}{TP + FN}$$

盾的度量, $F1$ 值是 Precision 和 Recall 的一种调和平均, 同时权衡这两个指标, $F1$ 值越高越好。计算公式如下:

$$F1 - score = \frac{2 * Precision * Recall}{Precision + Recall}$$

本文选取三个同类型的工具进行对比和分析。百度开源项目 OpenRASP[37], 该工具是 RASP 技术的开源实现, 能够对应用进行全面的监控和防护, 覆盖 Webshell 行为的部分场景。Copagent 工具[38], 是目前比较主流的开源内存型 Webshell 检测工具, 很多研究是在该工具基础上进行的。某 Beta 版工具(以下简称某工具), 该工具专门用于检测 Webshell, 也添加了内存型 Webshell 的检测功能。在测试内存型 Webshell 利用过程请求时, 由于请求会依次经过 Listener、Filter、Servlet 组件, 为避免先经过的组件类 Webshell 样本对后经过的组件类样本检测时的影响, 每轮请求完后会重启服务。

通过对 71 个 Webshell 和 70 个正常请求的测试, 4 个工具的检测结果如表 9 所示。

表 10 实验检测结果

Table 10 Experimental test results (%)

方案	Accuracy	Precision	Recall	F1-score
OpenRASP	72.34	100.00	45.07	62.14
Copagent	69.50	100.00	39.44	56.57
某工具	19.86	28.57	39.44	33.14
本文方法	96.45	98.53	94.37	96.40

实验表明, 本文方法具有较高的准确率, 且精确率和召回率的综合 $F1$ 值优于其他检测工具。

OpenRASP 是基于 RASP 的动态检测工具, 目前只能在程序启动时指定 Java Agent 以 premain 的方式对程序进行检测, 在程序运行时无法嵌入检测程序, 因此运行时如果检测需要重启服务。该工具只能检测内存型 Webshell 的利用过程, 对注入过程检测不到, 且实验测试发现, 对于 Listener、Valve 组件的命令执行后的利用过程也检测不到, 准确率和召回率较低。该工具无法识别内存型 Webshell, 只能通过触发后门的方式拦截并报警, 本文将该工具对内存型 Webshell 攻击的拦截报警认为是检测到了内存型 Webshell, 因此精确率为 100%。该工具只收集了目标程序的运行时信息, 无法定位到存在后门的类。$F1$ 值也较低。

Copagent 是一款基于规则匹配的静态检测工具。

首先获取 JVM 中所有加载的类, 通过黑名单筛选出高危类, 再对高危类进行简单的字符串匹配来判断内存型 Webshell。该工具无法对目标程序实时检测, 无法对内存型 Webshell 的注入过程进行检测。该工具设定的字符串匹配比较单一, 只针对 Webshell 常使用的字符串, 所以精确率高。但设定的规则比较简单, 无法检测到 Controller、Interceptor、Valve 类的内存型 Webshell, 并且实验可以绕过该工具匹配的字符串, 该工具无法检测高对抗内存型 Webshell, 准确率、召回率、*F*1 值都比较低。

某工具也是一款静态检测工具, 无法对目标程序实时检测, 无法对内存型 Webshell 的注入过程检测。分析发现, 该工具也是基于黑名单筛选出高危类, 然后基于高危字符串规则匹配来判断是否为内存型 Webshell。该工具的匹配规则比 Copagent 多, 匹配了很多的特权函数字符串、编解码方法字符串、加解密方法字符串, 导致正常请求容易被误判, 准确率、精确率比较低。但该工具对高危类和高危字符串的匹配也不全, 实验中可以绕过该工具的匹配规则, 例如通过 HEX 编码的高对抗内存型 Webshell 无法检测到, 导致准确率和召回率比较低。该工具目前对 Java 项目的兼容性不太好, 无法检测 Spring、Weblogic 的 Agent 类内存型 Webshell。

本文方法结合动静态检测技术, 动态检测方法中函数监测点比较全面, 能够在内存型 Webshell 注入阶段和利用阶段及时发现后门存在; 并且本文方法既能在程序启动前以 premain 方式检测, 也能够在程序运行时以 agentmain 的方式进行检测。静态检测方法中, 通过对内存型 Webshell 请求调用栈上函数的全面筛查, 筛选出比较全的高危类, 并结合深度学习模型训练, 能够很好地检测高对抗内存型 Webshell, 准确率较高。本文方法在检测过程中记录多个监测点上下文信息, 能够准确的定位到 Webshell 后门存在的类。

5.3 案例分析

本小节搭建内存型 Webshell 实际利用场景, 分析本文检测方法对组件类和 Agent 类 Webshell 后门的检测过程案例分析。搭建反序列化漏洞的 Web 服务, 测试对组件类内存型 Webshell 的检测过程; 搭建任意文件上传漏洞的 Web 服务, 测试对 Agent 类内存型 Webshell 的检测过程。

(1) 组件类内存型 Webshell

搭建 Tomcat 环境和 Shiro 框架, 利用 Shiro 的反序化漏洞注入组件类内存型 Webshell。首先使用 Shiro_Attack 工具[39]爆破密钥, 然后使用反序列化攻击解除 Shiro 对 header 的长度限制, 之后选择一个 Filter 类内存型 Webshell, 对其进行序列化、AES 加密、base64 编码, 最后将其作为 cookie 的字段值发送到目标应用程序中。

目标应用程序在本文检测方法的保护下, 能够发现并拦截到内存型 Webshell 的类并发送邮件。结果获取了 RASP 拦截的信息, 包括特权类函数信息、注册组件类函数信息, 以及请求的上下文信息, 同时也获取了函数调用栈信息。内存型 Webshell 调用的注册组件函数是 org.apache.catalina.core. Application-Context@addFilter, 注册的组件类名为 com.shiro.vu-ln.Controller.TomcatMemShellInject, Webshell 注入过程中的调用栈信息如图 9 所示。

图 9 组件类检测结果

Figure 9 Component type detection result

(2) Agent 类内存型 WebShell

搭建 Tomcat 任意文件上传漏洞的测试环境, 改造冰蝎中的 Agent 内存型 Webshell, 并注入混淆的高对抗代码, 通过冰蝎上传改造后的 jar 包, 注入 Webshell 后门。注入成功后, 可以访问到 Webshell 后门, 并且磁盘上不存在该文件。

用本文检测方法防御后, 成功检测到被内存型 Webshell 修改的系统类并发送邮件。结果显示内存型 Webshell 存在的类是 org.apache.jasper.servlet.JspServlet, 高危类是 javax/servlet/Servlet 接口。如图 10 所示。

图 10 Agent 类检测结果

Figure 10 Agent type detection result

5.4 性能分析

因为本文动态检测方法是在应用程序运行时进行检测, 为了不影响用户的正常访问, 性能评估尤为重要。由于 Copagent 工具和某工具采用的静态检测方法, 不做性能测试分析。本节对本文检测方法和 OpenRASP 进行性能测试分析和对比。使用 Apache 公司的 JMeter[40]性能测试工具进行测试。采用平均响应时间、性能消耗作为评估指标。以 Tomcat 服务作为实验环境, 每次请求将执行 1000 次哈希计算操作, 使系统响应时间更加接近实际情况下网站响应时间的用户体验度。在没有任何安全防护的情况下, 测试服务的平均响应时间, 然后在服务上部署本文检测方法和 OpenRASP 后分别测试服务的平均响应时间。

设置 JMeter 的线程数为 2000, Ramp-up period 为 60s, 来模拟 2000 个用户在 60s 内随机进行访问, 将此过程循环 10 次, 总体请求数为 20000 个。

假如运行检测工具前后的平均响应时间分别是 T_1、T_2, 则性能消耗的计算公式 T 如下所示。

$$T=\frac{T_2-T_1}{T_1}\times 100\%$$

检测结果如表 11 所示。检测工具运行前系统的平均响应时间是 1.55s, 本文检测方法运行后的平均响应时间是 1.67s, 平均响应时间延迟 0.12, 性能消耗是 7.74%, 在可接受范围。OpenRASP 运行后的平均响应时间是 1.98s, 平均响应时间延迟 0.43s, 性能消耗是 27.74%, 较为明显。

根据运行检测工具前后的请求响应时间绘制了响应时间分布曲线图, 如图 11 所示。整体上, 运行本文检测方法和 OpenRASP 后系统的响应时间均略长, 但本文检测方法的分布曲线图更接近无安全防护情况下的系统响应时间分布, 且性能消耗小于 OpenRASP。本文检测方法对目标应用程序的影响不大。

表 11 性能测试结果

Table 11 Performance test results

测试方案	平均响应时间(s)	性能消耗(%)
无安全防护	1.55	/
本文检测方法	1.67	7.74
Open RASP	1.98	27.74

5.5 讨论

本文检测方法是动静态结合检测内存型 Webshell, 动态方法可以实时检测目标程序, 静态方法需要获取 JVM 中加载的类再用于模型判定, 目前

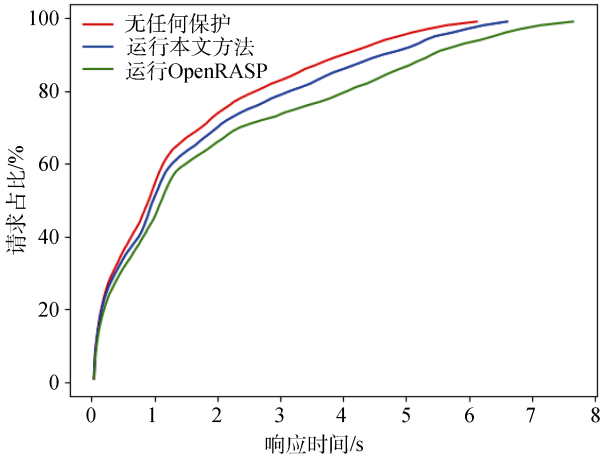

图 11 运行检测工具前后的响应时间分布曲线图

Figure 11 Response time distribution curve before and after running the detection tool

无法做到实时检测, 可以根据目标应用程序需求, 定期或周期性检测, 以更全面地防护目标受内存型 Webshell 的入侵。

由于 Copagent 工具和某工具无法检测高对抗内存型 Webshell, 为了测试其静态检测能力, 只对部分 Agent 类内存型 Webshell 改造为高对抗内存型 Webshell 进行测试。受限于内存型 Webshell 样本的稀缺, 无法达到更全面的检测效果。

6 总结与展望

本文对 Java 内存型 Webshell 进行了研究, 分析两种类型的内存型 Webshell 原理, 构建威胁模型, 并提出高对抗内存型 Webshell 定义, 设计了一种动静态相结合的方法检测高对抗内存型 Webshell。从内存型 Webshell 的输入源、触发器、特征状态方面出发, 利用 RASP 技术对输入源中的注册组件类函数, 以及特权类函数进行监测, 根据内存型 Webshell 的特性, 针对不同的触发器场景分析, 结合数据流分析技术进行动态检测; 针对攻击载荷改造的高对抗内存型 Webshell, 基于深度学习模型训练, 进行静态文本特征的检测。对本文检测方法进行了实验评估, 对 71 个内存型 Webshell 样本和 70 正常样本进行测试, 结果表明本文检测方法的准确率达到 96.45%, 性能消耗为 7.74%, 具有可行性。

下一步工作中, 将完善针对 Java Web 内存型 Webshell 的检测方案, 本文检测方案中在内存型 Webshell 注入过程中监测了注册组件类的函数, 没有对内存型 Webshell 注入过程中的可利用漏洞处进行监测, 后续将针对这部分内容进行补充, 增加内存型 Webshell 在注入时的检测效率。另外研究其他

语言 Web 服务的内存型 Webshell 检测技术。

参考文献

[1] Canali D, Balzarotti D. Behind the Scenes of Online Attacks: An Analysis of Exploitation Behaviors on the Web [C]. *20th Annual Network & Distributed System Security Symposium*. 2013.

[2] Firstbrook P, Lawson C. Innovation insight for extended detection and response[J]. Gartner ID G00718616, 2021.

[3] 2020 年无文件攻击无处不在. 亚信安全. https://www.asiainfo-sec.com/upload/files/2021/02/2020eybdzs.pdf. Feb. 2021.

[4] Tomcat 源代码调试笔记 - 看不见的 Shell. N1nty. https://mp.weixin.qq.com/s/x4pxmeqC1DvRi9AdxZ-0Lw. Jun. 2017.

[5] 利用“进程注入”实现无文件不死 webshell. Rebeyond. https://www.cnblogs.com/rebeyond/p/9686213.html. Aug. 2018.

[6] 基于内存 Webshell 的无文件攻击技术研究. LandGrey. https://www.anquanke.com/post/id/198886. Feb. 2020.

[7] ShiroVulnEnv. KpLi0rn. https://github.com/KpLi0rn/ShiroVulnEnv. Apr. 2021.

[8] Guo Y, Marco-Gisbert H, Keir P. Mitigating Webshell Attacks through Machine Learning Techniques[J]. *Future Internet*, 2020, 12(1): 12.

[9] Yang J, Processing C A. A webshell detection model based on Bayes[C]. *2021 2nd International Conference on Computer Communication and Network Security*, 2021: 71-74.

[10] Wang Z, Yang J, Dai M, et al. A method of detecting webshell based on multi-layer perception[J]. *Academic Journal of Computing & Information Science*, 2019, 2(1).

[11] Tao F J, Cao C J, Liu Z H. Webshell Detection Model Based on Deep Learning[M]. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019: 408-420.

[12] Pan Z L, Chen Y C, Chen Y, et al. Webshell Detection Based on Executable Data Characteristics of PHP Code[J]. *Wireless Communications and Mobile Computing*, 2021, 2021: 1-12.

[13] Zhao Z H, Liu Q X, Song T T, et al. WSLD: Detecting Unknown Webshell Using Fuzzy Matching and Deep Learning[M]. Information and Communications Security. Cham: Springer International Publishing, 2020: 725-745.

[14] Li Y, Huang J, Ikusan A, et al. ShellBreaker: Automatically Detecting PHP-Based Malicious Web Shells[J]. *Computers & Security*, 2019, 87: 101595.

[15] Wu Y X, Sun Y Q, Huang C, et al. Session-Based Webshell Detection Using Machine Learning in Web Logs[J]. *Security and Communication Networks*, 2019, 2019: 1-11.

[16] Shi L Y, Fang Y. Webshell Detection Method Research Based on Web Log[J]. *Journal of Information Security Research*, 2016, 2(1): 66-73.
(石刘洋, 方勇. 基于 Web 日志的 Webshell 检测方法研究[J]. *信息安全研究*, 2016, 2(1): 66-73.)

[17] Scott, B., and H. Ben. Web shell detection using neopi[R/OL]. 2011.

[18] Cui H D, Huang D L, Fang Y, et al. Webshell detection based on random forest-gradient boosting decision tree algorithm[C]. *2018 IEEE Third International Conference on Data Science in Cyberspace*, 2018: 153-160.

[19] Tian Y F, Wang J B, Zhou Z J, et al. CNN-webshell: Malicious web shell detection with convolutional neural network[C]. *The 2017 VI International Conference on Network, Communication and Computing - ICNCC 2017*, 2017: 75-79.

[20] Zhang H, Guan H C, Yan H B, et al. Webshell Traffic Detection with Character-Level Features Based on Deep Learning[J]. *IEEE Access*, 6: 75268-75277.

[21] Fang Y, Qiu Y Y, Liu L, et al. Detecting Webshell Based on Random Forest with FastText[C]. *The 2018 International Conference on Computing and Artificial Intelligence*, 2018: 52-56.

[22] Yong B B, Liu X, Liu Y, et al. Web behavior detection based on deep neural network[C]. *2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation*, 2018: 1911-1916.

[23] 边界无限三周年 | 极客精神 不断挑战 做网安行业的弄潮儿. 边界无限. https://mp.weixin.qq.com/s/sVLtfioGaG1bLFvaQ1-slQ. May. 2022.

[24] 饱受无文件攻击之苦？一文详解内存马攻击防范关键点. https://mp.weixin.qq.com/s/qUqrHzJmAfodVA1Y6xbucg. May. 2022.

[25] SHELLPUB.COM 专注查杀. 河马. https://www.shellpub.com.

[26] Čisar P, Čisar S M, Bioengineering, et al. The framework of runtime application self-protection technology[C]. *2016 IEEE 17th International Symposium on Computational Intelligence and Informatics*, 2017: 081-086.

[27] Hang Y U, Shuai W, Huamin J I N. RASP based Web security detection method[J]. *Telecommunications Science*, 36(11): 113.

[28] Yin Z X, Li Z F, Cao Y. A Web Application Runtime Application Self-Protection Scheme Against Script Injection Attacks[M]. Cloud Computing and Security. Cham: Springer International Publishing, 2018: 566-577.

[29] 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.
(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.)

[30] Peng J. *Research and implementation of runtime self-protection technology in smart contracts*[D]. Beijing: Beijing University of Posts and Telecommunications, 2020.
(彭婧. 智能合约运行时自我保护技术的研究与实现[D]. 北京: 北京邮电大学, 2020.)

[31] Thomas S L, Francillon A. Backdoors: Definition, Deniability and Detection[M]. Research in Attacks, Intrusions, and Defenses. Cham: Springer International Publishing, 2018: 92-113.

[32] Liu Q X, Wang B Z, Hu E Z, et al. Java Backdoor Detection Based on Function Code Gadgets[J]. *Journal of Cyber Security*, 2019, 4(5): 33-47.
(刘奇旭, 王柏柱, 胡恩泽, 等. 基于功能代码片段的 Java 后门检测方法[J]. *信息安全学报*, 2019, 4(5): 33-47.)

[33] Allen F E, Cocke J. A Program Data Flow Analysis Procedure[J]. *Communications of the ACM*, 1976, 19(3): 137.

[34] Wang L, Li F, Li L, et al. Principle and Practice of Taint Analysis[J]. *Journal of Software*, 2017, 28(4): 860-882.
(王蕾, 李丰, 李炼, 等. 污点分析技术的原理和实践应用[J]. *软件学报*, 2017, 28(4): 860-882.)
[35] He K M, Zhang X Y, Ren S Q, et al. Deep residual learning for image recognition[C]. *2016 IEEE Conference on Computer Vision and Pattern Recognition*, 2016: 770-778.
[36] Class SimpleEmail. Apache. https://commons.apache.org/proper/commons-email/apidocs/org/apache/commons/mail/SimpleEmail.html.
[37] LandGrey/Copagent. GitHub. https://github.com/LandGrey/copagent. Feb. 2022.
[38] OpenRASP. 百度安全. https://rasp.baidu.com. Jan. 2022.
[39] SummerSec/ShroAttack2. Github. https://github.com/SummerSec/ShiroAttack2. Jun. 2022.
[40] APACHE JMeter. APACHE. https://jmeter.apache.org. Feb. 2022.

张金莉 于 2016 年在北京邮电大学信息安全专业获得硕士学位。现任中国科学院信息工程研究所助理研究员。研究领域为网络攻防技术。研究兴趣包括: Webshell 检测、恶意代码分析。Email: zhangjinli@iie.ac.cn

陈星辰 于 2020 年在东北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博士学位。研究领域为 Web 安全和程序分析。Email: chenxingchen@iie.ac.cn

王晓蕾 于 2019 年在西安电子科技大学信息安全专业获得学士学位。现在中国科学院大学计算机技术专业攻读硕士学位。研究领域为 web 安全。Email: wangxiaolei@iie.ac.cn

陈庆旺 于 2021 年在东北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博士学位。研究领域为漏洞挖掘和机器学习。研究兴趣包括: 隐私保护、数据挖掘。Email: chenqingwang@iie.ac.cn

代峰 于 2021 年在北京工商大学计算机应用技术专业获得硕士学位。现任中国科学院信息工程研究所助理工程师。研究领域为恶意代码检测、机器学习。研究兴趣包括: 恶意代码检测、Webshell 检测。Email: daifeng@iie.ac.cn

李香龙 于 2020 年在长春理工大学网络工程专业获得学士学位。现在中国科学院大学电子信息专业攻读硕士学位。研究领域为 web 安全、代码分析。研究兴趣包括: Java 代码分析、攻防技术。Email: lixianglong@iie.ac.cn

冯云 于 2021 年在中国科学院大学网络空间安全专业获得博士学位。现在中国科学院信息工程研究所工程师。研究领域为究领域为网络攻防技术、网络攻击追踪溯源。Email: fengyun@iie.ac.cn

崔翔 于 2012 年在中国科学院计算技术研究所信息安全专业获得博士学位。现任广州大学网络空间先进技术研究院研究员。研究领域为网络攻防技术。Email: cuixiang@gzhu.edu.cn