一种针对 Tomcat Filter 型的 MemShell 检测技术研究

蔡国宝[1], 张 昆[2], 曲 博[3], 李 俊[4], 袁 方[5], 李振宇[1], 丁 勇[1]

[1]桂林电子科技大学计算机与信息安全学院 桂林 中国 541000
[2]国家信息中心 北京 中国 100000
[3]鹏城实验室 深圳 中国 518000
[4]国家工业信息安全发展研究中心 北京 中国 100000
[5]外交部通信总台 北京 中国 100000

摘要 近些年来, 随着计算机技术的不断发展和应用, Web 应用技术也在快速更迭, 与其一起发展的还有木马后门技术, 但传统的木马后门技术已经不能满足攻击者的需求, 因而基于内存攻击的方式不断涌现, 包括 powershell 内存载入攻击、.NET assembly 托管代码注入攻击以及内存马(Memory WebShell, MemShell)攻击等, 这些攻击方式为现有的安全防御检测机制带来了极大的挑战。因而业界对面向解决基于内存的攻击尤其是内存马的攻击展现出了强烈的需求。但当前业内针对内存马的检测能力较弱, 学术界也缺乏对该领域的研究工作, 所以本文提出了一种针对 Tomcat Filter 型的内存马检测方法。通过研究发现, 内存马其最核心技术便是无文件(Fileless)及不落地(Living off the Land), 但尽管如此, 内存马最终会在内存中展现其功能并执行命令, 所以内存是所有威胁的交汇点, 因此本文将 Java 虚拟机(Java Virtual Machine, JVM)作为起始点, 首先利用 JVM 内存扫描技术遍历出 JVM 内存中加载的所有 Filter 类型对象, 但需要注意的是这些对象并非都是有威胁的, 并且每一个对象都具有一定的特征, 所以可以对这些特征通过人工经验进行分类并且筛选出具有代表性的特征向量, 然后获取每一个 Filter 类型对象的所有代表特征向量, 并根据特征向量的值梳理出异常表现序列; 最后, 利用朴素贝叶斯算法将大量正常和异常的 Filter 对象的异常表现序列作为训练样本, 计算出对应项的条件概率并形成贝叶斯分类器。利用训练出的贝叶斯分类器就可以构建出一个内存马检测模型, 该模型能够有效得针对该类型的内存马进行检测。实验结果表明, 本文提出的方法针对 Tomcat Filter 型内存马的检测, 实现了零误报率和 94.07%的召回率。

关键词 远程控制; 内存马; 无文件后门; 朴素贝叶斯分类算法; 异常表现序列
中图法分类号 TN92 **DOI 号** 10.19363/J.cnki.cn10-1380/tn.2023.07.11

Research on MemShell Detection Technology for Tomcat Filter

CAI Guobao[1], Zhang Kun[2], Qu Bo[3], Li Jun[4], Yuan Fang[5], Li Zhenyu[1], Ding Yong[1]

[1]School of Computer and Information Security, Guilin University of Electronic Technology Guilin 541000, China
[2]State Information Center, Beijing 100000, China
[3]Peng Cheng Laboratory, Shenzhen 518000, China
[4]China Industrial Control Systems Cyber Emergency Response Team, Beijing 100000, China
[5]Communication center of the Ministry of Foreign Affairs, Beijing 100000, China

Abstract In recent years, with the continuous development and application of computer technology, web application technology is also changing rapidly, along with the development of Trojan back door technology. Apart from the attacks using traditional Trojan back door technology, memory-based attacks are emerging, including PowerShell memory loading attacks, .NET assembly managed code injection attacks and Memory WebShell (MemShell) attacks, all of which can bring great challenges to the existing security defense and detection mechanism. Therefore, there comes a great demand for solutions to memory-based attacks, especially MemShell ones. While the industry is presently faced with a lack of MemShell detection means, little academic research has been carried out. Under the circumstances, this paper proposes an approach to detecting MemShell of Tomcat Filter. Research shows that the core

通讯作者: 丁勇, 博士, 教授, stone_dingy@126.com

本课题得到国家重点研发计划(No. 2020YFB1006003, No. 2020YFB1006004)、国家自然科学基金(No. 61772150, No. 61862012, No. 61962012, No. 62002184)、广东省重点领域研发计划项目(No. 2020B0101090002)、鹏城实验室网络空间安全研究中心网络仿真项目(No. PCL2018KP004)的资助。
收稿日期: 2021-12-21; 修改日期: 2022-02-28; 定稿日期: 2023-04-17

technology of MemShell is fileless and living off the land. However, MemShell will eventually show its functions and execute commands in memory, so memory is the intersection of all threats. Therefore, this paper takes Java virtual machine (JVM) as the starting point. Firstly, use the JVM memory scanning technology to traverse all Filter type objects loaded in the JVM memory, whereas it should be noted that these objects are not all threatening, and each object has certain characteristics, so these characteristics can be classified through human experience and representative feature vectors can be filtered. Then, get the representative eigenvector of each filter type object, and sort out the abnormal performance sequence according to the value of the eigenvector. Finally, the naive Bayesian algorithm is used to take a large number of abnormal performance sequences of normal and abnormal filter objects as training samples to calculate the conditional probability of corresponding items and then form a Bayesian classifier. Using the trained Bayesian classifier, a MemShell detection model can be constructed, which can effectively detect this type of MemShell. Shown by the experimental results, the method proposed in this paper achieves zero false positive rate and 94.07% recall rate for the detection of Tomcat filter MemShell.

Key words remote control, memory webshell, fileless webshell, naive bayesianclassification algorithm, abnormal performance sequence

1 引言

随着 Web 技术的快速发展, 基于浏览器/服务器(Browser/Server, B/S)架构的站点已然成为了当代互联网 Web 应用的主流, 但随之而来的还有各种 Web 攻击。根据国家互联网应急响应中心(National Internet Emergency Center, CNCERT)发布的《2020 年上半年我国互联网网络安全监测数据分析报告》[1]指出, 今年国家信息安全漏洞共享平台(China National Vulnerability Database, CNVD)收录通用型安全漏洞 11073 个, 同比大幅增长 89.0%, 其中 Web 应用漏洞排名第二, 占比 26.5%。当攻击者利用这些漏洞取得权限后, 通常会留下一个被称之为 WebShell 的恶意文件, 攻击者可以通过该文件达到窃取数据、修改数据或者发起 DDOS 攻击的目的。但近些年来, 随着攻守双方的技术博弈, 诸如流量分析、端点检测与响应(Endpoint Detection and Response, EDR)、蜜罐、入侵检测等专业的监测和防护手段被防守方广泛使用, 使得传统上以文件形式持续驻留在目标服务器的 WebShell 的方式逐渐失效, 攻击难度也逐步加大。这时内存马(Memory WebShell, MemShell)便重回攻击方的视野。

内存马也被称为无文件马(Fileless WebShell, FWShell), 是无文件攻击的一种常用手段。据波尼蒙研究所(Ponemon Institute, PI)发现[2], 2018 年的网络攻击中有 35%的攻击是无文件攻击, 其中在所有数据泄露的攻击中, 无文件攻击占比 50%。同时根据趋势科技(Trend Micro, TM)网络安全报告[3]数据显示, 无文件攻击在 2019 年上半年增长了 256%。在此背景下, 对于内存马的检测就显得格外重要。传统的 WebShell 的检测方法通常是基于有文件的检测技术, MemShell 的无文件特性使得它在躲避检测方面具有天然的优势, 因此检测 MemShell 是否存在, 对防止攻击者进一步控制服务器, 防止数据泄漏, 具有重要意义。

目前内存马多数存在于 Java 应用程序中, 且大多数都是基于 Tomcat 服务器, 因而也诞生了基于 Tomcat 三大组件 Servlet、Filter 以及 Listener 类型的内存马, 本文提出的检测方法便是针对 Tomcat Filter 型 MemShell 的检测。同时, 本文基于该方法实现了一个 MemShell 检测工具, 该脚本可以实时扫描 Java Virtual Machine(JVM)内存中加载的所有 Filter 类型对象。本文的主要贡献如下:

1) 本文基于目前主流的方法以及研究经验总结归纳出了若干特征值, 并利用这些特征值生成了异常表现序列, 同时给出了异常表现序列生成的算法。

2) 从实用性以及工程量方面考虑, 本文基于朴素贝叶斯分类算法构建了一个专门针对 MemShell 的检测模型, 利用该模型能够输出高风险的 Filter 类对象及其相关信息。

3) 基于上述方法本文实现了对 Tomcat Filter 型的 MemShell 的检测, 实验分析表明, 该方法实现了较高的检测准确率和召回率。

2 背景与相关工作

JavaWeb 中有三大组件, 分别是 Servlet、Filter 以及 Listener, 这三个组件在 JavaWeb 开发中担任着不同的职责: Servlet 主要用于生成动态内容; Filter 主要用于修改并调整资源的请求或响应[4]; Listener 主要根据事件状态进行更有效的资源管理和自动处理[5]。在这三个组件中, Filter 组件因为不与具体的 URL 耦合且能够接受客户端请求中数据的特性导致其被攻击者频繁地用于构造 MemShell。此外, 由于 Filter 型的 MemShell 能够被攻击者动态注入, 导致其攻击的痕迹难以通过日志和流量追寻, 因而像一些基于流量检测的方法[6-7], 基于日志检测的方法[8-9]对

MemShell的检测有所不足, 又由于MemShell无文件的特性, 基于机器学习的有文件检测方法[10-12]在失去检测目标的情况下无法进行检测, 因而对于 Filter 型 MemShell 的检测研究显得十分重要。

Filter 型 MemShell 有如下四点让其难以检测。

第一, Filter 型的 MemShell 无逻辑结构边界, 难以被发现。通常 MemShell 仅存在于进程的内存空间中, 与正常的代码和数据混淆在一起。MemShell 与传统恶意代码的不同之处在于它在磁盘上不存在文件, 会导致传统的检测防护手段失效。

第二, Filter型的MemShell缺乏稳定的静态特征, 难以被识别。MemShell 缺乏结构化的静态形式, 它依附在进程运行期间的输入数据进入进程, 数据可能被加密混淆。因此, 无法通过常规的静态特征去识别 MemShell。

第三, Filter 型的 MemShell 注入方式多样, 注入后攻击手法多样, 难以有针对性的防御方法。从最常用的利用反序化漏洞进行注入, 到利用代码执行漏洞注入, 再到文件上传漏洞注入, 不同 MemShell 的注入方式、执行方式、恶意代码触发机制各不相同, 例如有些 MemShell 在完成 Filter 的注入后, 结合其他攻击手法如采用冰蝎的 AES 动态加密、使用伪造的 HTTPS 证书进行流量加密或自定义编解码方式, 进而达到流量混淆规避监测等效果, 难以被安全产品拦截防御。

第四, Filter 型的 MemShell 本身就镶嵌在在应用系统中, 处于管理和运维难度较高的位置, 非企业的高级管理人员无法访问到 Filter 中的代码片段, 因此就算察觉到Filter可能已经遭到恶意的注入也难以查找到具体的问题。

图 1 说明了一个 Filter 型的 MemShell 从动态注入进 JavaWeb 中再到 Filter 响应请求并执行命令的过程。首先, 攻击者通过攻击手段将内存马注入到 Tomcat 容器中, 并向浏览器发起带有攻击 payload 的请求, 经过 Listener 监听器流向 Filter 过滤器; 然后, 被注入在容器中的 MemShell 在 Filter 过滤器中被激活, 执行完传入的攻击 payload 后, 向 Servlet 发起逻辑调用; 最后, Servlet 再经过一层层的 Filter 过滤器将逻辑请求的结果返回给攻击者, 完成一次攻击。

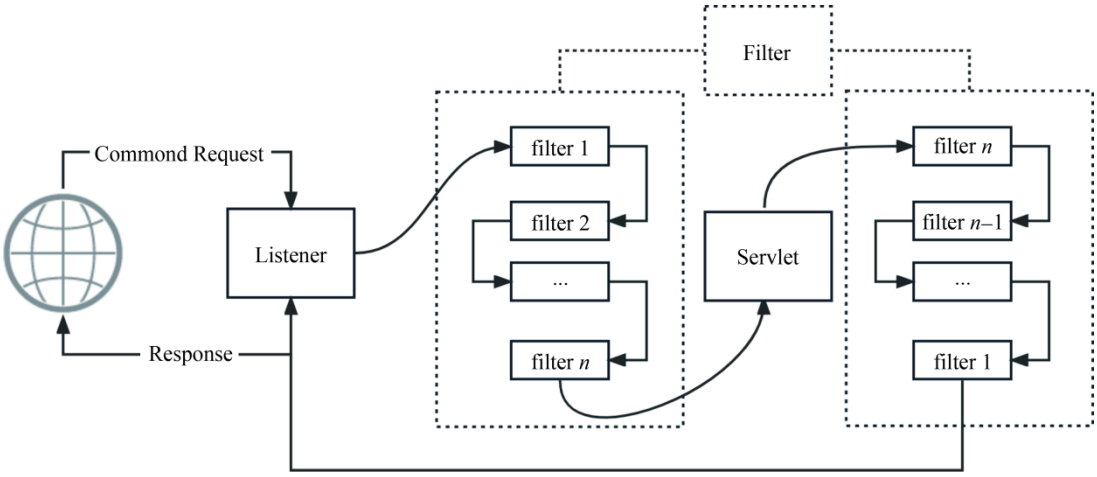

图 1 MemShell 执行命令的过程

Figure 1 MemShell executes the command process

针对上面提出的难点, 本文提出了具有针对性的检测算法, 虽然 Filter 的型 MemShell 属于无文件攻击的一种类型且难以被检测, 但也正是由于其无文件的特性, 使得 Filter 的型 MemShell 更容易被针对。对于 Java Web 中普通的 Filter 对象, 其在磁盘上都有着对应的 Class 文件, 当使用内存扫描技术去扫描JVM加载的Filter对象时, 可以通过判断相应对象在磁盘中有无 Class 文件来进一步检测。因此, 本文将通过两个阶段来达到对 Tomcat Filter 型的 MemShell 检测目的, 其中第一个阶段是遍历出 JVM 内存中加载的所有 Filter 类型对象, 然后再获取每一个 Filter 类型对象的四个特征值, 并利用这四个特征值的多种判断特征去构建异常表现序列。当异常表现序列构建完成后, 在下一个阶段, 再基于朴素贝叶斯分类算法模型, 通过构建完成的异常表现序列归纳总结出朴素贝叶斯分类器(Naive Bayesian Classifier, NBC), 利用归纳的分类器对待测 Filter 类对象进行分类判断, 并输出分类结果。

下面本文利用两个章节分别介绍这两个阶段的具体研究方法, 其中第 3 章节主要阐述了异常表现

序列的构建过程和主要算法, 第 4 章节主要介绍了基于朴素贝叶斯算法的分类模型构建过程。最后, 在第 5 章本文根据提出的算法模型进行了实验并对结果进行了分析。

3 基于异常表现序列的构建方法

构建异常表现序列首先需要做的就是对象的确认, 因此在本章中本文要首先确定对象, 然后根据确定的对象来筛选特征和特征值, 最后根据特征与特征值之间的关联, 根据特定算法来构建异常表现序列。

3.1 Filter 对象扫描

Tomcat 的 Filter 对象和其他 Class 一样都被加载到了 JVM 中, 因此想要对 Filter 对象进行扫描首先需要从 Tomcat JVM 中获取所有加载的 Filter 类, 但从 Tomcat JVM 中提取出所有 Filter 对象, 并根据 Filter 对象筛选出特征值是一项挑战, 目前获取的方式主要有以下三种方式。

3.1.1 Serviceability Agent

SA(Serviceability Agent)是 JDK 提供的一个强大的调试工具集, 它适用于语言层和虚拟机层, 支持调试运行着的 Java 进程、core 文件和虚拟机 crash 之后的 dump 文件, 使其适合调试 VM(Virtual Machine)本身和生产中的 Java 程序[13], 因此可以通过 SA 提供的 Java API, 如 sa-jdi.jar 去查看当前 JVM 中已经加载的类[14]。此外, 利用 SA 还可以获取到 JVM 中运行的 Filter 类的字节码以便进一步分析。

3.1.2 Arthas

Arthas 是 Alibaba 开源的 Java 诊断工具[15], 利用 Arthas 可以达到监控到 JVM 的实时运行状态的目的。因此利用 Arthas 能够快速的将 Tomcat JVM 中所有的 classLoader 的信息统计出来, 并可以展示继承树、urls 等。

3.1.3 VisualVM

VisualVM 是一款全能型性能监控和故障分析工具[16], 自 JDK 1.6.0_10 版本以来, JVisualVM 成为 JDK 的一部分, 利用 JVisualVM 能够对 JVM 堆内存消耗、线程、类加载的实时监控。

以上三个方式是目前比较流行的检测方式, 但若使用 SA 方式, 通常要求拥有 root 权限, 否则会报错, 且若向线上业务加载 javaagent 可能会直接导致目标 JVM 出现异常而导致服务宕机; 若使用 VisualVM 方式, 必须在服务端项目进程中配置相关的监控参数。然后 VisualVM 通过远程连接到项目进程才可以获取相关的数据。但当线上环境的网络是隔离的时, 本地的监控工具根本连不上线上环境, 导致 VisualVM 无法使用。此外, VisualVM dump 和得到实例的速度很慢, 影响扫描的效率。但使用 Arthas 的方式, 不但能够快速帮完成扫描, 还可以导出 JVM 中加载类的字节码, 进行后续分析。但 Arthas 的功能面向的人员主要是开发者, 因而导致其有很多功能对于检测 MemShell 来说是冗余的, 因此需要对 Arthas 进行二次开发以更大的程度完成 Filter 对象扫描的目的。

3.2 特征值及多类检测分支

利用 Arthas 扫描一个类获得的信息如图 2 所示, 包括类名、加载类对应 java 文件及字节码的路径、类的类型、父类、加载类和加载类的 hash 等。

```
class-info        org.apache.jsp.shell_jsp
 code-source       /Users/panda/Library/Caches/JetBrains
                   /IntelliJIdea2020.1/tomcat
                   /Tomcat_9_0_36_shellCheckDemo/work
                   /Catalina/localhost/shellCheckDemo_war_exploded/
 name              org.apache.jsp.shell_jsp
 isInterface       false
 isAnnotation      false
 isEnum            false
 isAnonymousClass  false
 isArray           false
 isLocalClass      false
 isMemberClass     false
 isPrimitive       false
 isSynthetic       false
 simple-name       shell_jsp
 modifier          final,public
 annotation
 interfaces        org.apache.jasper.runtime.JspSourceDependent,
                   org.apache.jasper.runtime.JspSourceImports
 super-class       +-org.apache.jasper.runtime.HttpJspBase
                     +-javax.servlet.http.HttpServlet
                       +-javax.servlet.GenericServlet
                         +-java.lang.Object
 class-loader      +-org.apache.jasper.servlet.JasperLoader@d47b5aa
                     +-ParallelWebappClassLoader
                         context: shellCheckDemo_war_exploded
                         delegate: true
                       ----------> Parent Classloader:
                       java.net.URLClassLoader@6f75e721

                       +-java.net.URLClassLoader@6f75e721
                         +-sun.misc.Launcher$AppClassLoader@764c12b6
                           +-sun.misc.Launcher$ExtClassLoader@759ebb3d
 classLoaderHash   d47b5aa
```

图 2 Arthas 扫描类获取信息

Figure 2 Arthas scan class to get information

这些类的信息并非全部是本文在检测 MemShell 时所需, 因此需要从获得的这些信息中选择检测时必须的要素当成特征值。特征值应根据 Filter 型的 MemShell 注入应用程序时伪装的信息、必须条件以及无法改变的信息去选择。当选择完特征值后, 将这些特征值详细分化, 对于每一个分化后的特征值预设多类检测分支, 然后将这些多类检测分支当成判断特征去构建异常表现序列。

经过大量的对 Filter 型 MemShell 人工检测的经验总结, 本文选取如表 1 所示的信息为特征值。

表 1 特征值的选择

Table 1 Selection of eigenvalues

代表参数	特征值	描述
N	Filter Name	加载 Filter 类的名称
C	Filter Class	加载的具体 Filter 类
L	ClassLoader	Filter 类的类加载器
P	File Path	Filter 类对应文件路径

下面本文对特征值进行详细分化。对于 N 和 C, 这两个特征值完全是由用户自定义的字段值, 取决于 Filter 型的 MemShell 构造者的习惯, 但一些构造者在构造时往往喜欢留下具有明显代表意义的个人标志信息, 因此针对这些关键信息可以把它当做识别特征去预设检测分支; C 与 N 不同的是, Filter 型 MemShell 的 C 通常会继承一些指定的接口类, 所以这些接口类也应当做识别特征。此外, C 的继承类同样会有指定类, C 本身所属包名也可能存在常用包名, 因此这些也可以当做识别特征去预设分支; 对于 L, Filter 也是 Class, 必定有特定的 classLoader 去加载, 通常情况下的 Filter 都是由中间件 WebappClassLoader 所加载, 但如果是通过远程类加载、任意代码执行以及文件上传等方式注入的, 那么其 classLoader 就会不同, 因此这也是一个值得检测的预设分支。对于 P, 因为 MemShell 无文件的特性, 因此 MemShell 不存在于物理磁盘空间, 该识别特征是十分重要的预设分支。

经过上述分析, 可以得到如表 2 所示的多类检测分支。

表 2　多类检测分支

Table 2　Multi-type detection branch

代表参数	所属特征值	描述
n_1	Filter Name	过滤器名 Flag 分支
c_1	Filter Class	**实现类检测分支**
c_2	Filter Class	继承类检测分支
c_3	Filter Class	类名 flag 分支
c_4	Filter Class	包名分支
l_1	ClassLoader	类加载器分支
p_1	File Path	有文件检测分支

3.3　构建异常表现序列

在上一步骤中已经得到了多类检测分支, 现在本文将每一个类检测分支的结果认作是一个关联数据流。具体如下。

假设 L 的 c_1 检测分支判断结果为真, 则定义如下:

$$L \rightarrow c_1@1$$

那么每一个类的类检测分支应该有七个关联数据流。图 3 所示的是在 Tomcat JVM 中获得某个 Filter 类的信息。

```
Filter Name   : index
Filter Class  : org.apache.jsp.index
      Filter Class Name:  org.apache.jsp.index
      Interfaces       :  javax.servlet.Filter
      Extends Class    :  org.apache.jasper.servlet.JasperLoader@16c7c466
      package Name     :  org.apache.catalina.Context;java.lang.refelct.*;...;
classLoader   : org.apache.jasper.servlet.JasperLoader
```

图 3　Filter 类信息

Figure 3　The information of Filter Class

假设其七个关联数据流表示如下:

$$N \rightarrow n_1@0$$
$$C \rightarrow c_1@0$$
$$C \rightarrow c_2@0$$
$$C \rightarrow c_3@1$$
$$C \rightarrow c_4@1$$
$$L \rightarrow l_1@1$$
$$P \rightarrow p_1@1$$

那么该 Filter 类异常表现序列为: $\{n_1=0, c_1=0, c_2=0, c_2=0, c_3=1, c_4=1, l_1=1, p_1=1\}$。

需要注意的是, 每一个对象的多类检测分支都是各不相同的, 因此在构建异常表现序列之前首先要确定多类检测分支判断集合 EvalList, 该集合的每个子集对应着不同的检测分支, 当确定完多类检测分支判断集合后, 再从 Tomcat JVM 扫描结果中获取到 *FilterMap* 集合。遍历 *FilterMap* 集合就可以得到所有 Filter 对象的全部特征值。通过判断每个检测分支与对应所属特征值的关系, 就能够根据多类检测分支的判断结果来构建异常表现序列。整个构建过程如算法 1。

算法 1. 异常表现序列构建

输入: *FilterMaps* 集合, *EvalList* 集合

输出:异常表现序列 *AbnormalSequence*

```
1:function SETTRUE(parameter)
2:parameter ← result
3:end function
4: function CHECKMETHOD(CheckFiled, CheckList, parameter)
5:parameter ← 0
6:if CheckFilednotinCheckList then
7:SETTRUE(CheckFiled, parameter)
8:end if
9:return parameter
10:end function
11:k=SIZE(FilterMaps)
12: function FILTERCHECK(FilterMaps, k, EvalList)
13:FilterNameFlag ← EvalList[0]
14:EvalIInterfaces ← EvalList[1]
15:EvalExtends ← EvalList[2]
16:FilerClassFlag ← EvalList[3]
17:EvalPackage ← EvalList[4]
18:isExstFile ← EvalList[5]
19:while k-- do
20:N, C, L, P ← FilterMaps[k]
21:CHECKMETHOD(N, EvalList[0], n1)
```

22:CHECKMETHOD(C, *EvalList*[1], c_1)
23:CHECKMETHOD(C, *EvalList*[2], c_2)
24:CHECKMETHOD(C, *EvalList*[3], c_3)
25:CHECKMETHOD(C, *EvalList*[4], c_4)
26:CHECKMETHOD(L, *EvalList*[5], l_1)
27:CHECKMETHOD(P, *EvalList*[6], p_1)
28:*AbnormalSequence*[i] = {n_1, c_1, c_2, c_3, c_4, l_1, p_1}
29:**end while**
30:**return** *AbnormalSequence*
31:**end function**

算法的输入来自从 Tomcat JVM 扫描结果中获取到的集合 *FilterMap*, 以及多类检测分支判断集合 *EvalList*, 首先初始化检测分支(第 5 行), 令所有的检测分支为待检测状态, 然后判断每个检测分支与对应所属特征值的关系(第 6~8 行), 再将扫描出的 *FilterMaps* 传入该判断函数进行判断(第 13~18 行), 如果判断符合 *EvalList* 中的特征条件, 那么确定 *N, C, L, P* 四个特征值的多类检测分支检测的结果(第 21~27 行), 最终生成并返回异常表现序列(第 28 行)。

本文提出的异常表现序列构建算法能够有效的针对筛选出的 Tomcat Filter 型对象的特征进行快速判断。并且由于其算法属性对规律性强、特征明显的对象有极强的针对性, 因而不会导致误判等问题。此外, 在经过多类检测分支各个所属特征值的映射关系向上向下追踪, 可以非常准确的构建出异常表现序列以供后续朴素贝叶斯分类器生成时使用。

4 基于朴素贝叶斯分类的检测模型

异常表现序列构建完毕后, 就需要用朴素贝叶斯分类模型对构建的序列进行训练学习, 因此本章节主要介绍了如何利用朴素贝叶斯算法构建分类模型。

4.1 基于朴素贝叶斯的分类模型

序列异常通常是对离散异常时序事件的检测, 常应用于工业设备检测, 生物界中的氨基酸序列或基因组序列检测, 用户行为分析等方面。序列异常分为两类。第一类为位置异常, 即序列是否异常取决于位置上的实际值与模型预测值间的偏差。第二类是组合异常, 以符号组合为考量, 对整个序列进行判断, 如果其与绝大多数不同, 则被作为异常找出来。异常表现序列便是属于第二类。通常针对于序列异常检测的方法是马尔科夫链(Markov chain)模型, 马尔科夫链描述的是状态空间从一个状态到另一个状态转换无后效性的随机过程, 在该过程中, 下一状态的概率分布仅由当前概率决定。但在 MemShell 的检测中, 当确定了一个 Filter 类对象后, 本文所选的特征值便不再变动, 因而在上一时刻和下一时刻均不能影响其状态, 也就是说对于 Filter 对象而言是没有状态的。这也就导致无法利用马尔科夫链去检测。

综上考虑, 本文选择了朴素贝叶斯分类算法来构建分类模型。朴素贝叶斯分类算法是一种应用广泛的分类算法, 对于给出的待分类项, 求解在此项条件下各个类别出现的概率, 哪个最大, 那么就认为此待分类项属于哪个类别。基于朴素贝叶斯分类算法的检测技术属于有监督机器学习的一种, 通过对已经训练的序列进行学习, 归纳出朴素贝叶斯分类模型, 然后根据改分类模型分出待测的 Filter 对象。分类模型如图 4 所示。

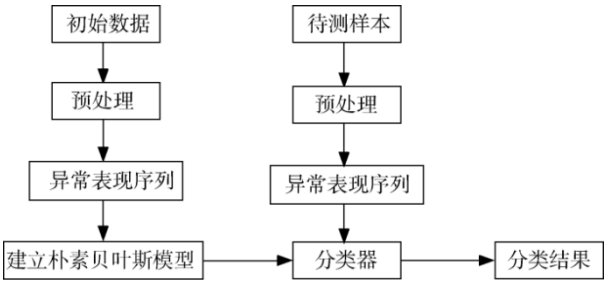

图 4 基于朴素贝叶斯的分类模型
Figure 4 Classification model based on Naive Bayes

该模型主要分为三个阶段, 第一阶段是预处理准备阶段, 主要梳理出正常和异常 Filter 对象的异常表现序列。第二阶段是训练阶段, 将大量的正常和异常 Filter 对象的异常表现序列作为训练样本, 计算出对应项的条件概率并形成贝叶斯分类器。第三阶段是检测阶段, 将陌生 Filter 对象的异常表现序列作为待测分类项输入到第二阶段进行检测。

4.2 朴素贝叶斯检测分类器的构建

在第三节中已经介绍了异常表现序列生成的方法, 这里本文主要介绍朴素贝叶斯分类器的生成。

设异常表现序列待分类项集合为: $X=\{x_1,x_2,x_3,x_4,x_5,x_6,x_7\}$, 分别表示 n_1、c_1、c_2、c_3、c_4、l_1、p_1 七个多类检测分支。

本文定义的有类别集合为 $C=\{C_1,C_2\}$, 其中 C_1 为正常 Filter 对象, C_2 为异常 Filter 对象。然后需要计算各个条件概率, 即 $P(C_1|X)$, $P(C_2|X)$, 并且如果 $P(C_j|X)=\max\{P(C_1|X), P(C_2|X)\}$, 那么 $X\epsilon C_j$。

计算各个条件概率首先需要统计各类别下各个特征属性的条件概率估计, 即

$$\mathrm{P}\left(x_1|C_i\right),\mathrm{P}\left(x_2|C_i\right),\ldots,\mathrm{P}\left(x_m|C_i\right) \tag{1}$$

$$\mathrm{P}\left(x_1|C_j\right),\mathrm{P}\left(x_2|C_j\right),\ldots,\mathrm{P}\left(x_m|C_j\right) \tag{2}$$

由于各个特征属性是条件独立的, 那么根据贝叶斯定理有如下推导

$$\mathrm{P}\left(C_j|X\right)=\frac{\mathrm{P}\left(X|C_j\right)\mathrm{P}\left(|C_j\right)}{\mathrm{P}\left(X\right)} \tag{3}$$

本文定义对于未知类型 Filter 对象集合 D, 朴素贝叶斯分类器将预测在异常表现序列待分类项集合 X 出现条件下具有最高后验概率的类别。朴素贝叶斯分类器要预测集合 D 属于类 C_1, 需满足

$$\mathrm{P}\left(C_i|X\right)>\mathrm{P}\left(C_j|X\right),\ \mathrm{i,j=1,2} \tag{4}$$

根据式(4)可知, 需要计算出最大的 $\mathrm{P}\left(\mathrm{C_j}|\mathrm{X}\right)$ 的值, 而在式(3)中, 其分母对于所有类别为常数, 且由于各个特征属性都是相互独立假设提出的, 因此各特征属性条件独立, 所以只需要将分子最大化, 有

$$\begin{aligned}&\mathrm{P}\left(X|C_j\right)\mathrm{P}\left(C_j\right)\\&=\mathrm{P}\left(x_1|C_j\right)\mathrm{P}\left(x_2|C_j\right)\ldots\mathrm{P}\left(x_m|C_j\right)\\&=\mathrm{P}\left(C_j\right)\prod_{i=1}^{m}\mathrm{P}\left(x_i|C_j\right)\end{aligned} \tag{5}$$

综上, 要判断集合 D 的类型, 对于每个类 C_j, 只需要计算每个类别对应的 $\mathrm{P}\left(X|C_j\right)\mathrm{P}\left(C_j\right)$ 的值, 取概率最高的类别作为待测样本集合 D 的类别。

5 实验分析

5.1 实验环境

本文选取了一台使用 2.7GHz 主频英特尔四核 Core i7 处理器、内存为 16 GB 的主机进行试验, 系统为 macOS Big Sur 11.3.1。选取了 Github 和 Gitlee 上部分以 SSH 和 SSM 框架开发的项目, 总计检测对象 202 517 个, 从中筛选出 239 个 Filter 训练对象。

本文将这些项目中的Filter对象进行预处理和特征处理后形成了异常表现序列, 并以此建立了朴素贝叶斯模型。如图 5 为训练集散点图, 每一个点都代表着异常表现序列中某一个特征值所属分类, 其中 c_1、c_2 是大多数 MemShell 所具有的表现序列, 而对于 P, 由于 MemShell 无文件的特性, 导致大多数的 MemShell 并未出现该特征, 但由于某些 MemShell 功能上的问题, 虽然其不以.jsp 的形式存在于项目的路径下, 但依旧以.class 的形式存在。

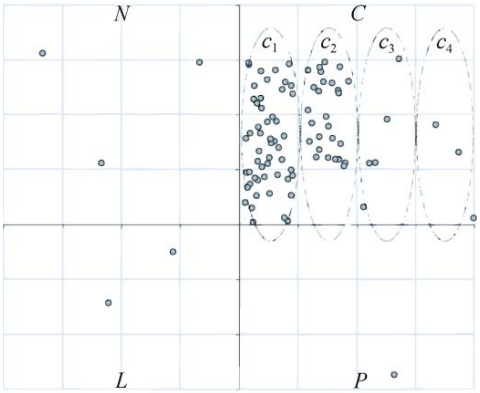

图 5 部分 filter 对象表现序列散点分布

Figure 5 Part of the filter objects represent the scattered point distribution of the sequence

同时, 为了验证该模型对于 MemShell 的检测准确性, 本文整理了三个不同的 Java Web 测试项目, 然后利用本文的检测工具从这三个 Java Web 项目分别整理出 13899、21329、34200 个类对象, 并从中分别筛选出了 27、31、35 个 Filter 对象类。此外, 本文还收集并归纳了共计 20 种不同功能的 Filter 类对象 MemShell, 然后模拟了主流内存马注入的攻击手段, 通过 Java 反序列化攻击、文件上传攻击、JavaAgent 等不同的方式将其注入到三个 Java Web 测试项目中作为负样本。

5.2 实验结果

根据上述实验环境进行实验测试, 得到实验数据如表 3 所示。

表 3 检测结果

Table 3 Experimental result

样本总量	Filter 对象类型总量	负样本总量	注入方式	分类正确	误报数	漏报数
13899	47	20	反序列化	13897	0	2
			文件上传	13897	0	2
			JavaAgent	13897	0	2
21329	51	20	反序列化	21328	0	1
			文件上传	21328	0	1
			JavaAgent	21328	0	1
34200	55	20	反序列化	34199	0	1
			文件上传	34199	0	1
			JavaAgent	34199	0	1

从表 3 中可以看出, 注入方式对于本实验的检测结果没有造成影响, 此外为了能够准确评价分类器的性能, 本文使用了精确率(Precision)、召回率(Recall)和 F1-score 三个指标来评估检测效果。具体利用的是混淆矩阵真正(TP)、真负(TN)、假正(FP)、假负(FN)来计算, 具体如表 4 所示。

表 4　混淆矩阵

Table 4　Confusion matrix

混淆矩阵		实际值	
		Positive	Negative
预测值	Positive	TP	FN
	Negative	FP	TN

True Positive(TP)是将正类预测为正类数, 为准确预测; True Negative(TN)是将负类预测为负类数, 为准确预测; False Positive(FP)是将负类预测为正类数, 为误报; False Negative(FN)是将正类预测为负类数, 为漏报。计算精确率、召回率、F1-score 的公式如(5)、(6)、(7)所示。

精确率: 分类结果中为 C_j 的样本数占分类结果中所有分为 C_j 的样本数, 用于衡量分类的查准率。

$$\text{Precision}=\frac{\text{TP}}{\text{TP}+\text{FP}} \tag{6}$$

召回率: 分类结果中正确分类为 C_j 的样本数占所有类 C_j 的样本数的比例, 用于衡量分类的查全率。

$$\text{Recall}=\frac{\text{TP}}{\text{TP}+\text{FN}} \tag{7}$$

F1-score: 基于精确率和召回率的基础上的概念, 用于反映模型的稳健性, 对精确率和召回率进行整体评价。

$$\text{F1}=\frac{2\text{TP}}{2\text{TP}+\text{FN}+\text{FP}} \tag{8}$$

即等同于

$$\text{F1}=\frac{2\cdot\text{Precision}\cdot\text{Recall}}{\text{Precision}+\text{Recall}} \tag{9}$$

根据上述计算公式, 本文最终实现了零误报率和 94.07%的召回率, 并且衡量分类效果的 F1 值高达 96.94%, 实验结果表明, 本模型对于检测 Tomcat Filter 型的 MemShell 实现了较高的准确率和召回率。同时本文也对检测耗时做了统计, 三份样本耗时与检测数量的折线图如图 6 所示, 对于每个 classLoader 加载类对象的检测时间约为 1.05ms, 表明了本文的检测方法也具有较高的检测效率。

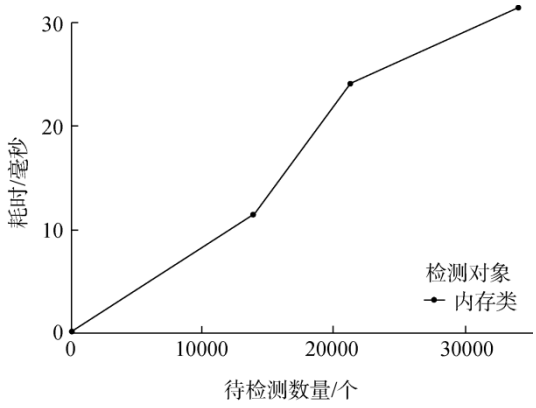

图 6　MemShell 检测耗时

Figure 6　MemShell detection time-consuming

最后, 本文还调研了目前市场上主流的 WebShell 检测工具以及其他研究者提出的研究方法是否支持对 MemShell 的检测, 具体如表 5 所示。

表 5　本文与现有相关工作对 MemShell 检测的支持情况

Table 5　This paper and the existing related work support memshell detection

产品名称	是否支持 MemShell 的检测	支持语言	兼容性
本文方法	√	JAVA	通用
D-Shield	×	PHP/ASP/ASPX/JAVA	仅 Windows
WEBDIR+	×	PHP/ASP/ASPX/JAVA	通用
WebShell Detector	×	PHP/CGI/ASP/ASPX	通用
CloudWalker	×	PHP	通用
PHP Malware Finder	×	PHP	仅 Linux
WebShell Analyzer	×	PHP/ASP/ASPX/JAVA	通用
WebShell AIHunter	×	PHP/ASP/JAVA	通用
Loki	√	PHP/CGI/ASP/ASPX	通用
WTA	×	PHP	通用
FindBot	×	PHP/ASP/ASPX/JAVA	通用

本文选取了十一种当前流行的 WebShell检测工具来观察其对于 MemShell 的检测能力, 他们是 D-Shield[17]、WEBDIR+[18]、WebShell Detector[19]、CloudWalker[20]、PHP Malware Finder[21]、WebShell Analyzer[22]、WebShell AIHunter[23]、Loki[24]、WTA[25]、FindBot[26]。通过分析发现这些检测工具中仅有 Loki 能够通过分析内存字符串和 MFT 分析的方式从侧面对 MemShell 进行扫描检测, 其余检测工具并不支持对于 MemShell 的检测。此外, 进一步研究发现 Loki 的内存字符串分析功能比较依赖扫描规则, 开源版

本的 Loki 扫描规则有限, 对于隐蔽性强的 MemShell 无法探测, 且 Loki 主要功能并非是针对 MemShell 进行检测, 因而只能得出可疑的检测数据, 无法对得到的检测数据进行判定, 这导致用户可能会更多的去关注 Loki 得到判定结论的扫描结果, 而忽略这些混杂在扫描过程中的检测数据信息。

通过对比分析发现, 本文的检测方法也存在一定的局限性。首先, 本文中针对 Filter 型 MemShell 的检测算法过于依赖现有类型 MemShell 的已知特征和对于 MemShell 识别的人工判断经验。如果想要对应用了新技术或者使用了新免杀思路的 Filter 型 MemShell 进行检测, 就必须拥有此类型的已知样本, 然后针对该类型样本基于新思路和新技术的原理进行人工分析、总结经验、提取新特征等一些列工作, 最后再通过异常表现序列的构建和训练, 归纳出新的朴素贝叶斯检测分类模型, 从而达到检测的效果。这样一来就使得检测方案的可扩展性降低, 因此后续的工作需要考虑检测方案的拓展性问题。其次, 本文的检测方法虽然能够快速有效的针对 MemShell 进行检测, 但是检测类型有限, 仅对 Tomcat Filter 型的 MemShell 具有可检测性, 对于其他类型如 Servlet 型、Listener 型或者其他容器中的 MemShell 如 SpringBoot、Jetty、Resin 等不具有可检测性, 需要完善对于其他类型 MemShell 和其他容器中 MemShell 的检测算法。

6 结束语

内存马是近些年出现较为频繁的新型无文件攻击方式。本文提出了构建异常表现序列的新颖方法, 然后基于朴素贝叶斯分类模型成功实现了对于 Tomcat Filter 型 MemShell 的检测。实验表明, 本方案实现了较高的准确率和召回率, 对于内存马的检测具有积极的意义。但在实际测试中发现, 虽然本文的方案不会产生误报的情况, 但是会存在漏报, 下一步工作将对多类检测分支判断集合 EvalList 进行优化, 并同时拓展对于 Tomcat Servlet 型 MemShell、Tomcat Listener 型 MemShell 的检测算法, 最后还需要对整体方案的可拓展性进行改善。

参考文献

[1] Analysis report on China's Internet network security monitoring data in the first half of 2020. China National Internet Emergency Center. http://www.cac.gov.cn/2020-09/26/c_1602682854845452. htm. Sept 2020.
(2020 年上半年我国互联网网络安全监测数据分析报告. 中国国家互联网应急中心. http://www.cac.gov.cn/2020-09/26/ c_1602682854845452.htm. Sept 2020.)

[2] 2020 State of Endpoint Security Final.Ponemon Institute. https://www.morphisec.com/hubfs/2020%20State%20of%20Endpoint%20Security%20Final.pdf. Jan 2020.

[3] Trend Micro Security Predictions for 2020.Trend Micro.https://www.trendmicro.com/vinfo/us/security/research-and-analysis/predictions/2020. 2020.

[4] Java™ Servlet Specification.Oracle Corporation. https://javaee.github.io/servlet-spec/downloads/servlet-4.0/servlet-4_0_FINAL.pdf. JUL 2017.

[5] Application Server Containers for J2EE Servlet Developer's Guide. Oracle Corporation. https://docs.oracle.com/cd/B14099_19/web.1012/b14017.pdf. JUL 2005.

[6] Yang W, Sun B, Cui B. A webshell detection technology based on http traffic analysis[C]. *International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, 2018: 336-342.

[7] Zhang H, Guan H C, Yan H B, et al. Webshell Traffic Detection with Character-Level Features Based on Deep Learning[J]. *IEEE Access*, 2018, 6: 75268-75277.

[8] 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(8): 1-11.

[9] 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.)

[10] Ai Z, Luktarhan N, Zhou A J, et al. WebShell Attack Detection Based on a Deep Super Learner[J]. *Symmetry*, 2020, 12(9): 1406.

[11] Cui H, Huang D, 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.

[12] Cui Y P, Shi K X, Hu J W. Research of Webshell Detection Method Based on XGBoost Algorithm[J]. *Computer Science*, 2018, 45(B06): 375-379.
(崔艳鹏, 史科杏, 胡建伟. 基于 XGBoost 算法的 Webshell 检测方法研究[J]. *计算机科学*, 2018, 45(B06): 375-379.)

[13] The HotSpot™ Serviceability Agent: An out-of-process high level debugger for a Java™ virtual machine. Kenneth Russell and Lars Bak. https://www.usenix.org/legacy/event/jvm01/full_papers/russell/russell_html/. Feb 2001.

[14] Charlie Hunt, Monica Beckwith, Poonam Parhar, Bengt Rutisson. Java Performance Companion[M].Addison-Wesley Professional, 2016,5: 45-46.

[15] Arthas Documentation.Alibaba Middleware Group.https://arthas.aliyun.com/doc/. 2018.

[16] VisualVM All-in-One Java Troubleshooting Tool. J. Sedlacek and H. Thomas. https://visualvm.github.io/. 2017.

[17] D-Shield. D-Shield. Available online: http://www.d99net.net/(accessed on 5 January 2022).

[18] WEBDIR+. BaiDu. Available online: https://scanner.baidu.com/#/pages/intro(accessed on 2021-0628-1400).

[19] WebShell Detector. ShellDetector. Available online: https://github.

com/emposha/PHP-Shell-Detector.
[20] Cloudwalker. Chaitin. Available online: https://github.com/chaitin/cloudwalker/.
[21] PHP Malware Finder. jvoisin. Available online: https://github.com/jvoisin/php-malware-finder.
[22] WebShell Analyzer. tstillz. Available online: https://github.com/tstillz/webshell-analyzer.
[23] WebShell AIHunter. ColdSnap. Available online: https://github.com/Coldwave96/WebShell-AIHunter.
[24] Loki. Neo23x0. Available online: https://github.com/Neo23x0/ LoKi/.
[25] Yu, L. . WTA: A Static Taint Analysis Framework for PHP Webshell[J]. *Applied Sciences*, 2021,11.
[26] FindBot.pl. CBL. Available online: https://www.abuseat.org/findbot.pl.

蔡国宝 于 2015 年在阜阳师范大学软件工程专业获得学士学位。现在桂林电子科技大学计算机技术专业攻读硕士学位。研究领域为 Web 安全、代码分析等。研究兴趣包括: 代码审计、漏洞挖掘。Email: lyxc0821@163.com

曲博 于 2017 年获得荷兰代尔夫特理工大学博士学位, 现任鹏城实验室助理研究员, 主要研究方向为应用网络分析、网络表示学习、网络安全等。Email: qub@pcl.ac. cn

袁方 北京航空航天大学硕士, 主要研究方向专用通信建设、信息化应用、信息安全、密码应用、网络安全等。Email: yuan_f_2008@163.com

丁勇 桂林电子科技大学教授, 博士生导师, 鹏城实验室兼职教授, 主要研究方向为密码学、区块链、网络安全等。Email: stone_dingy@126.com

张昆 于 2009 年 12 月在北京航空航天大学项目管理专业获得硕士学位。现任单位国家信息中心, 高级工程师。研究领域为网络空间安全、数据安全、政务信息化。主要研究领域为智慧城市、大数据、区块链等。Email: zhk1200@163.com

李俊 国家工业信息安全发展研究中心高级工程师, 主要研究方向为工控安全、工业互联网安全、新一代信息技术安全。Email: lijun@cics-cert.org.cn

李振宇 于 2015 年在桂林电子科技大学信息科技学院获得学士学位。现在桂林电子科技大学计算机技术专业攻读硕士学位。研究领域为网络空间安全、网络工程。研究兴趣包括：区块链、物联网、软件定义网络等。Email: lizhenyu.0@163.com