基于全局储存的新思路 | Tomcat的一种通用回显方法研究

原创 Litch1 长亭安全课堂 2020-03-19 17:35

收录于话题
#原创技术文章

26个

> 作者：Litch1，给phith0n师傅递茶的小弟

起因

对于不出网的反序列化回显，有利用Linux文件描述符的方法：《linux下java反序列化通杀回显方法的低配版实现》，思路比较巧妙，但有一些局限性，对比起来感觉response直接写回显会比较方便，但是其在通用性上也存在痛点：

- **很多框架对于Serlvet进行了封装，不同框架实现不同，同一框架的不同版本实现也可能不同，因此我们无法利用一种简单通用的方法去获取当前请求的response。**

针对这一点，前几天在先知社区看到了@kingkk师傅的**《Tomcat中一种半通用回显方法 》**学习了一下，师傅的思路很巧妙：**通过反射修改控制变量，来改变Tomcat处理请求时的流程，使得Tomcat处理请求时便将request,response存入ThreadLocal中，最后在反序列化的时候便可以利用ThreadLocal来取出response。**师傅的寻找过程主要是从Tomcat处理请求的调用栈中寻找有机会的代码流程。我在学习完师傅的文章之后也试着跟着调用栈寻找了一下，看是否有别的方法，但是发现这种方法比较掣肘于Tomcat处理请求时现有的流程，没有更好的发现。于是尝试换一种思路，不再寻求改变代码流程，而是找找有没有Tomcat全局存储的request或response。

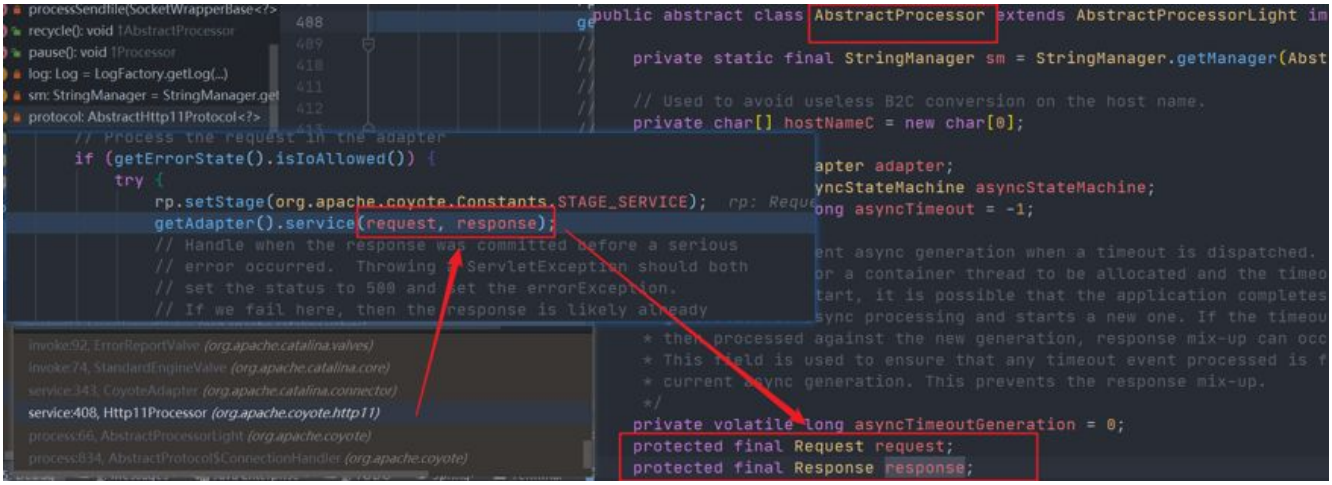
前提

个人感觉要忽略框架来寻找request，response的关键点是**寻找当前运行代码的上下文环境与Tomcat运行上下文环境之间的联系**。比如@kingkk师傅所用的ThreadLocal就是这样的联系。

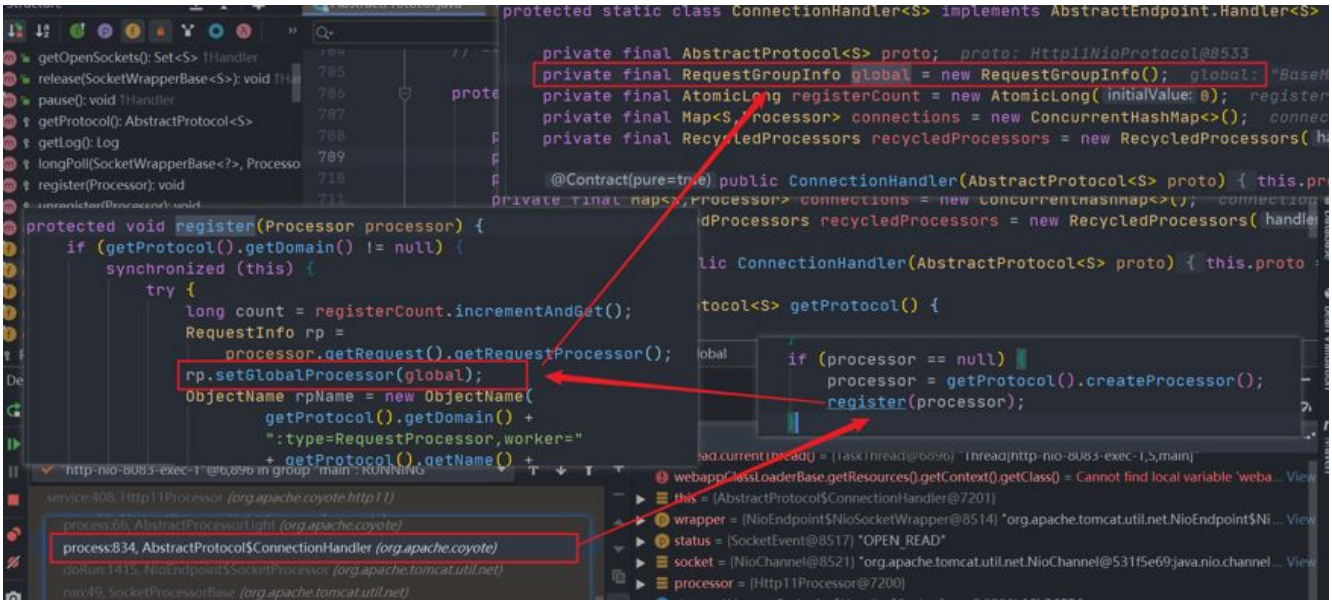
寻找

我们先看看Tomcat中哪个类会存储Request以及Response。

起一个SpringBoot调试一下看看，在调用链中发现继承Http11Processor的AbstractProcessor中有Request以及Response的Field，而且这两个Field都是final类型的，也就是说其在赋值之后，对于对象的引用是不会改变的，那么我们只要能够获取到这个Http11Processor就肯定可以拿到Request和Response。



这时候已经有了request，response，接下来往前寻找有没有哪里存储了这个Processor？或者是哪里对于Processor的Request等信息进行了存储？可以发现在之前的调用链中的AbstractProtocol的内部类ConnectionHandler中在处理的时候就将当前的Processor的信息存储在了global中。



其中这个RequestGroupInfo类型的核心就是一个存储所有RequestInfo的List。

```
public class RequestGroupInfo extends BaseModelMBean {
    private final List<RequestInfo> processors = new ArrayList<>();
```

那么到现在已经有了**AbstractProtocol$ConnectoinHandler------->global-------->RequestInfo------->Request-------->Response。**

再往后看调用栈，现在要寻找有没有地方有存储AbstractProtocol(继承AbstractProtocol的类)。

在CoyoteAdapter的service方法中，我们发现CoyoteAdapter的connector这个Field有很多关于Request的操作。

这个类中就有与AbstractProtocol有关的字段：**protocolHandler**，这个field的类型为ProtocolHandler,可以看一下继承了ProtocolHandler的类，其中与HTTP11有关的也都继承了AbstractProtocol。

处理请求的部分我们就寻找完了。为**Connector------>AbstractProtocol$ConnectoinHandler------->global-------->RequestInfo------->Request-------->Response。**

而在Tomcat启动过程中有这样的方法，可以看到会将Connector放入Service中。

```
public void setConnector(Connector connector) {
    Service service = getService();
    boolean found = false;
    for (Connector serviceConnector : service.findConnectors()) {
        if (connector == serviceConnector) {
            found = true;
        }
    }
    if (!found) {
        service.addConnector(connector);
    }
}
```

这里的Service为StandardService，所以串起来就是：**StandardService--->Connector--->AbstractProtocol$ConnectoinHandler--->RequestGroupInfo(global)--->RequestInfo------->Request-------->Response。**

关键

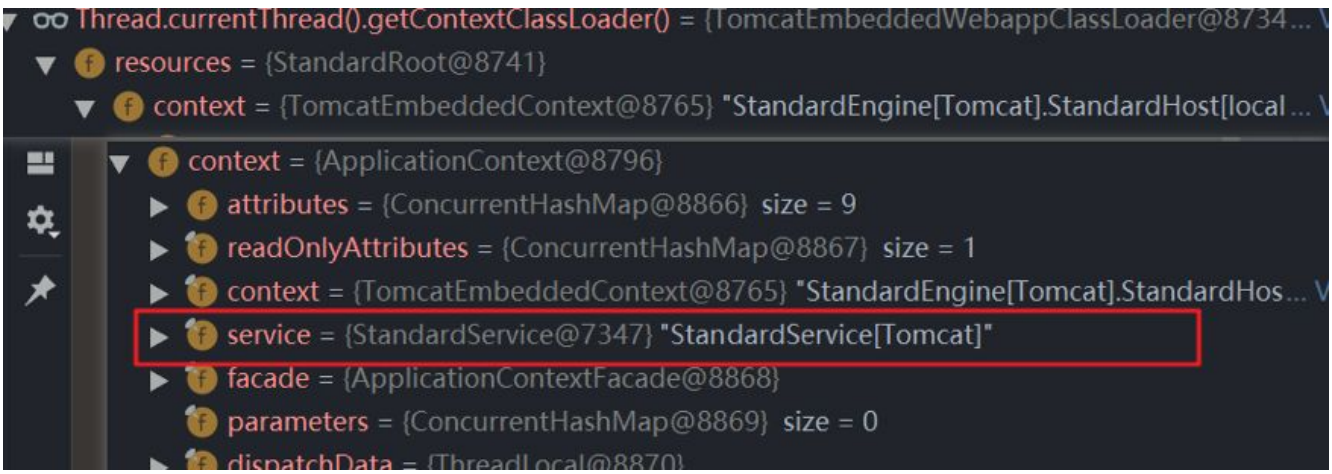
Tomcat的类加载机制并不是传统的双亲委派机制，因为**传统的双亲委派机制并不适用于多个Web App的情况。**

假设WebApp A依赖了common-collection 3.1，而WebApp B依赖了common-collection 3.2 这样在加载的时候由于全限定名相同，不能同时加载，所以必须对各个webapp进行隔离，如果使用双亲委派机制，那么在加载一个类的时候会先去他的父加载器加载，这样就无法实现隔离，tomcat隔离的实现方式是每个WebApp用一个独有的ClassLoader实例来优先处理加载，并不会传递给父加载器。**这个定制的ClassLoader就是WebappClassLoader。**

那么如何破坏Java原有的类加载机制呢？如果上层的ClassLoader需要调用下层的ClassLoader怎么办呢？就需要**使用Thread Context ClassLoader，线程上下文类加载器。**Thread类中有getContextClassLoader()和setContextClassLoader(ClassLoader cl)方法用来获取和设置上下文类加载器，如果没有setContextClassLoader(ClassLoader cl)方法通过设置类加载器，那么线程将继承父线程的上下文类加载器，如果在应用程序的全局范围内都没有设置的话，那么这个上下文类加载器默认就是应用程序类加载器。对于Tomcat来说ContextClassLoader被设置为WebAppClassLoader（在一些框架中可能是继承了public abstract WebappClassLoaderBase的其他Loader)。

说了那么多，其实**WebappClassLoaderBase就是我们寻找的Thread和Tomcat 运行上下文的联系之一。**

我们debug看看Thread.currentThread.getContextClassLoader()里面都有啥东西，这里只要稍微搜寻一下就会发现有很多Tomcat有关的运行信息。我们只要寻找我们上文提到的需要的Service就可以了。



最后的路径：

WebappClassLoaderBase ---> ApplicationContext(getResources().getContext()) ---> StandardService--->Connector--->AbstractProtocol$ConnectoinHandler---->RequestGroupInfo(global)--->RequestInfo------->Request-------->Response。

核心代码在修改完的ysoserial的util/Gardgets.java中。

加入ysoserial

我对ysoserial不是很熟，直接仿造@kingkk师傅把自己写的利用createTemplatesImpl加了进去。

新添加的链：

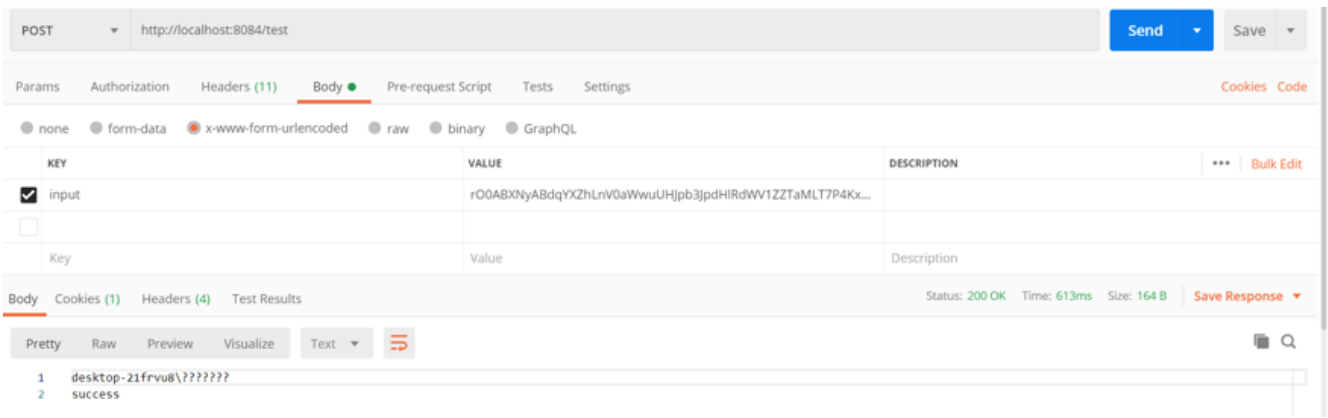
- CommonsBeanutils1TomcatHeader（用于解除tomcat对于request header的大小限制）
- CommonsBeanutils1TomcatEcho2
- CommonsCollections2TomcatEcho2
- CommonsCollections3TomcatEcho2
- CommonsCollections4TomcatEcho2

用法：

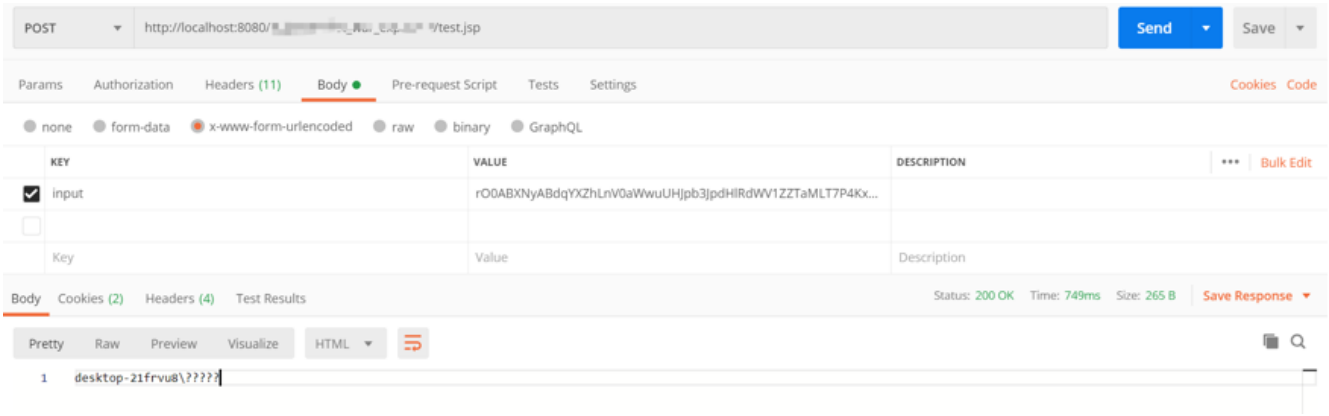
//为了区别出与普通request，需要在打的时候在request header加上 tomcat: tomcat
java -jar ysoserial-0.0.6-SNAPSHOT-all.jar CommonsBeanutils1TomcatEcho2 yourCommand。

github地址：https://github.com/buptchk/ysoserial/

- **测试SpringBoot（内置Tomcat8.0.30）：**



- **测试普通的JSP（Tomcat 9.0.7）：**

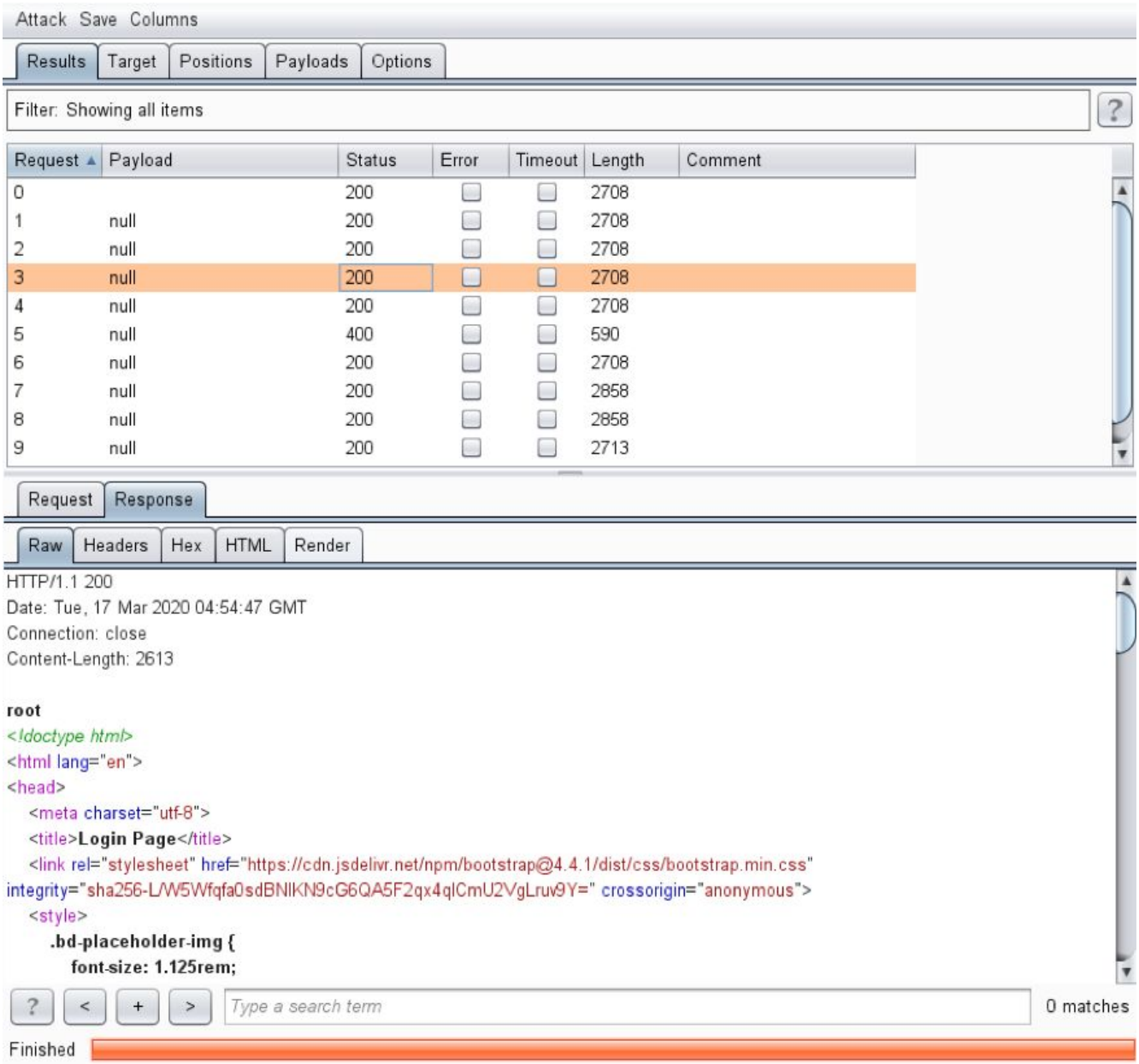


- **测试shiro（vulhub中的环境）：**

测试shiro的时候，发现一个问题，生成的payload太长了 ，已经超过了Tomcat默认的max header的大小，经过一再缩减也没有成功，于是考虑通过改变Tomcat max header的大小解除限制，思路是改变org.apache.coyote.http11.AbstractHttp11Protocol的maxHeaderSize的大小，这个值会影响新的Request的inputBuffer时的对于header的限制，但是由于request的inputbuffer会复用，所以我们在修改完maxHeaderSize之后，需要多个连接同时访问，让tomcat新建request的inputbuffer，这时候的buffer的大小限制就会使用我们修改过后的值。

具体的攻击流程：

//生成改变max header的较短payload，并发送过去
java -jar ysoserial-0.0.6-SNAPSHOT-all.jar CommonsBeanutils1TomcatHeader yourSize
//生成较长的回显payload，使用多个线程同时发送，注意控制发送payload的数量，一般来说在连接少的情况下发送10个之内就可以成功
java -jar ysoserial-0.0.6-SNAPSHOT-all.jar CommonsBeanutils1TomcatEcho2 yourCommand


Attack Save Columns
Results Target Positions Payloads Options
Filter: Showing all items
Request Payload Status Error Timeout Length Comment
0 200 2708
1 null 200 2708
2 null 200 2708
3 null 200 2708
4 null 200 2708
5 null 400 590
6 null 200 2708
7 null 200 2858
8 null 200 2858
9 null 200 2713
Request Response
Raw Headers Hex HTML Render
HTTP/1.1 200
Date: Tue, 17 Mar 2020 04:54:47 GMT
Connection: close
Content-Length: 2613
root
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Login Page</title>
<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@4.4.1/dist/css/bootstrap.min.css"
integrity="sha256-L/W5Wfqfa0sdBNIKN9cG6QA5F2qx4qICmU2VgLruv9Y=" crossorigin="anonymous">
<style>
.bd-placeholder-img {
font-size: 1.125rem;
Type a search term
0 matches
Finished

最后

从@kingkk师傅的文章中学到了很多，也因此在加入ysoserial的时候少踩了不少坑。

Thread.currentThread.getContextClassLoader()最终获取到request的办法应该不止一种，但有的可能有版本问题。

同时这个方法也有一些局限性，已知的局限性：

- 由于是从很多连接中筛出自己当前的那一条，连接较多时，可能有性能问题,并且对于shiro拓展header的问题，在连接特别多的情况下可能不适用（猜测有可能不会再新建inputBuffer，还没有进行测试）。
- 我自己测试的Tomcat版本较少，由于涉及到较多的Tomcat内部类，所以Tomcat实现改变的话就会有问题（我其实原来是用的另一种写法，但后来发现有版本问题）。

最后的最后

@threedr3am师傅刚发了一篇**《基于tomcat的内存 Webshell 无文件攻击技术》**,思路也很赞，感觉也可以基于Thread.currentThread.getContextClassLoader()来动态注册Filter，有兴趣的师傅可以试一下哈。

本人水平有限，如果师傅发现文章中的疏漏，欢迎指出来可以一起探讨~

References：

- *linux下java反序列化通杀回显方法的低配版实现*

https://xz.aliyun.com/t/7307

- *Tomcat中一种半通用回显方法*

https://xz.aliyun.com/t/7348

- *基于tomcat的内存 Webshell 无文件攻击技术*

https://xz.aliyun.com/t/7388

收录于话题 #原创技术文章 26

上一篇
对PowerShell免杀脚本的分析

下一篇
ERC20代币合约新型漏洞预警及分析，可导致无限授权转账

喜欢此内容的人还喜欢

漏洞风险提示 | Spring 远程代码执行漏洞
长亭安全课堂