【Struts2-命令-代码执行漏洞分析系列】S2-001

chybeta / 2018-02-08 17:41:00 / 浏览数 7993 技术文章 技术文章 顶(2) 踩(0)

前言

最近学习java安全，在分析s2-001的时候发现了一些问题和心得。

一方面网上关于s2-001的漏洞分析很少，基本上都是poc+利用而已。

另一方面在调试过程中感觉apache官方通告有不准确的地方，这点见后面的■■■■部分。

有不准确的地方望各位师傅指出，谢谢。

漏洞信息

漏洞信息页面： https://cwiki.apache.org/confluence/display/WW/S2-001

漏洞成因官方概述：Remote code exploit on form validation error

漏洞影响：

WebWork 2.1 (with altSyntax enabled), WebWork 2.2.0 - WebWork 2.2.5, Struts 2.0.0 - Struts 2.0.8

环境搭建

源码结构：

几个主要文件（待会用到）：
index.jsp

```
<html>
<head>
 <title>■■■■</title>
</head>
<body>
<h1>■■■■</h1>
<s:form action="login">
 <s:textfield name="username" label="username" />
 <s:textfield name="password" label="password" />
 <s:submit></s:submit>
</s:form>

</body>
</html>
```

struts.xml:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
       "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
       "http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
   <package name="s2-001" extends="struts-default">
       <action name="login" class="com.test.LoginAction">
           <result name="success">/success.jsp</result>
           <result name="error">/index.jsp</result>
       </action>
   </package>
</struts>
```

web.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLocation="
```

```
    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
    </filter>

    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>

</web-app>
```

完整源码见附件。

漏洞利用

最简单poc：

```
%{1+1}
```

任意命令执行:

```
%{#a=(new java.lang.ProcessBuilder(new java.lang.String[]{"pwd"})).redirectErrorStream(true).start(),#b=#a.getInputStream(),#c
```

将`new java.lang.String[]{"pwd"})`中的pwd替换为对应的命令，即可执行。

漏洞分析

在经过tomcat容器的处理后，http请求会到达struts2，从这里开始调试吧。

在 /com/opensymphony/xwork2/interceptor/ParametersInterceptor.java:158 接受我们输入的参数值，之后会调用对应的set方法(这个省略)：

```
ValueStack stack = ac.getValueStack();
        setParameters(action, stack, parameters);
    } finally {
```

继续执行，执行完interceptor部分，也即对`return invocation.invoke();`进行step over，到达/com/opensymphony/xwork2/DefaultActionInvocation.java:252

跟进`executeResult()`，到达 /com/opensymphony/xwork2/DefaultActionInvocation.java:343

跟进`result.execute(this)`，到达 /org/apache/struts2/dispatcher/StrutsResultSupport.java:175：

此后在`dispatcher.forward(request, response);`跟进，此处省略一些过程，相关调用栈如下：

进入 /org/apache/struts2/views/jsp/ComponentTagSupport.java:47，

这里会对jsp标签进行解析，但这时的标签并不包含我们的payload，我们可以在这里step over，直到解析到对应的标签：

如上图，我们提交的password值为`%{1+1}`，因此着重关注对`<s:textfield name="password" label="password" />`解析。回到`doStartTag`，执行完后会再次回到`index.jsp`，此时遇到了相应的闭合标签`/>`，会跳转到`doEndTag`:

```
public int doEndTag() throws JspException {
        component.end(pageContext.getOut(), getBody());
        component = null;
        return EVAL_PAGE;
}
```

跟进`component.end()`，到达 /org/apache/struts2/components/UIBean.java:486：

跟入`evaluateParams`后一直执行到如下图

由于开启了`altSyntax`，expr变为为`%{password}`

跟入`addParameter("nameValue", findValue(expr, valueClazz));`中的`findValue`，来到 /org/apache/struts2/components/Component.java:318

```
protected Object findValue(String expr, Class toType) {
       if (altSyntax() && toType == String.class) {
           return TextParseUtil.translateVariables('%', expr, stack);
       }
```

开启了altSyntax且toType为class.java.lang.string，跟入TextParseUtil.translateVariables，在/com/opensymphony/xwork2/util/TextParseUtil.jav

```
public static String translateVariables(char open, String expression, ValueStack stack) {
       return translateVariables(open, expression, stack, String.class, null).toString();
   }
```

继续跟入translateVariables，源码如下：

```
public static Object translateVariables(char open, String expression, ValueStack stack, Class asType, ParsedValueEvaluator eva
       // deal with the "pure" expressions first!
       //expression = expression.trim();
       Object result = expression;

       while (true) {
           int start = expression.indexOf(open + "{");
           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 = stack.findValue(var, asType);
               if (evaluator != null) {
                   o = evaluator.evaluate(o);
               }


               String left = expression.substring(0, start);
               String right = expression.substring(end + 1);
               if (o != null) {
                   if (TextUtils.stringSet(left)) {
                       result = left + o;
                   } else {
                       result = o;
                   }

                   if (TextUtils.stringSet(right)) {
                       result = result + right;
                   }

                   expression = left + o + right;
               } else {
                   // the variable doesn't exist, so don't display anything
                   result = left + right;
                   expression = left + right;
               }
           } else {
               break;
           }
       }

       return XWorkConverter.getInstance().convertValue(stack.getContext(), result, asType);
   }
```

此时expression为%{password}

经过while循环，确定start和end定位后，此时var为password，到达：

stack.findValue(var, asType);会返回password的值%{1+1}，这个就是我们传入的payload：

此后o为%{1+1}，再对o进行了一番处理后，payload经过result变量，最终成为expression的值：

在完成后，进入下一个循环：

并且在Object o = stack.findValue(var, asType);中完成了对payload的执行

因此究其原因，在于在translateVariables中，递归解析了表达式，在处理完%{password}后将password的值直接取出并继续在while循环中解析，若用户输入的pa

漏洞修复

XWork 2.0.4中，改变了ognl表达式的解析方法从而不会产生递归解析，用户的输入也不会再解析执行。

对应源码如下：

```
public static Object translateVariables(char open, String expression, ValueStack stack, Class asType, ParsedValueEvaluator eva
       // deal with the "pure" expressions first!
       //expression = expression.trim();
       Object result = expression;
       int loopCount = 1;
       int pos = 0;
       while (true) {

           int start = expression.indexOf(open + "{", pos);
           if (start == -1) {
               pos = 0;
               loopCount++;
               start = expression.indexOf(open + "{");
           }
           if (loopCount > maxLoopCount) {
               // translateVariables prevent infinite loop / expression recursive evaluation
               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 = stack.findValue(var, asType);
               if (evaluator != null) {
                   o = evaluator.evaluate(o);
               }


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

```
                    result = left + middle;
                }

                if (TextUtils.stringSet(right)) {
                    result = result + right;
                }

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

    return XWorkConverter.getInstance().convertValue(stack.getContext(), result, asType);
}
```

当解析完一层表达式后，如图，此时`loopCount > maxLoopCount`，从而执行break，不再继续解析`%{1+1}`:

diff的结果如下：

一个说明

回到漏洞信息和开头环境搭建部分，漏洞信息中的图有画上了几个红圈。基于此问一个问题：表单验证错误是在哪里触发的？

查阅apache2官方文档(https://struts.apache.org/core-developers/validation.html)提到

就本次测试源码而言，并没有相应的`validation.xml`或其他validation方法。为更直观起见，我们修改`LoginAction.java`源码：

```
public void setPassword(String password) {
    this.password = "%{1+1}";
}

public void setUsername(String username) {
    this.username = "chybeta";
}

@Override
public String execute() throws Exception {
    if (username.equals("chybeta") ) {
        return "error";
    } else {
        return "success";
    }
}
```

当submit时，会执行相应的set方法，直接设置username和password分别为`chybeta`与`%{1+1}`。修改后的源码中的`execute`方法中若`username`为`chybeta`则返回`erro`

所以这里没有涉及到我们传入的参数，不存在表单验证失败，也不存在逻辑上的验证失败。执行poc如下：

所以可以说表单验证错误并不是该漏洞的产生的原因，但表单验证错误是这个漏洞出现的场景之一。在struts2框架中，配置了Validation，倘若验证出错会往往会原样返回

以struts2的showcase为例：

在本测试环境的源码中，没有表单验证，但同样把用户的输入留在了页面里，从而在解析的时候执行了。

Refference

- https://cwiki.apache.org/confluence/display/WW/S2-001
- https://github.com/vulhub/vulhub/tree/master/struts2/s2-001
- http://www.freebuf.com/column/156344.html

点击收藏 | 5 关注 | 3

上一篇：Java反序列化漏洞之殇 下一篇：渗透测试之cisco路由器在渗透中的利用

1. 1 条回复

三顿 2018-02-24 15:04:24

见附件？？？附件呢？？？

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录