CVE-2018-11776：如何使用Semmle QL在Apache Struts中找到5个RCE

一叶飘零 / 2018-11-05 09:21:00 / 浏览数 2465 技术文章 技术文章 顶(0) 踩(0)

■■■■■https://lgtm.com/blog/apache_struts_CVE-2018-11776

前言

2018年4月，我向Struts安全小组报告了在Apache Struts新发现的一个远程代码执行漏洞，漏洞已被标记为2018-11776
(S2-057)，在某些配置下,如果一个服务器上运行了Struts，那么这个漏洞就会出现，或者访问特定的URL，这个漏洞也会出现。有关Struts的版本和配置受影响的详细信息、

这一发现是我对ApacheStruts安全性研究的一部分，在这篇文章中，我将介绍发现漏洞的过程。我将解释如何使用已知的漏洞来获取Struts内部运行的信息，并创建封装Str

探寻攻击面

许多安全漏洞涉及到的数据，一方面可能是它们的来源不受信任，比如有的数据来源于用户的输入，也可能是数据使用方式的问题，例如，SQL查询、反序列化、其他一些解
library就能完成所有的工作。对于一个特定的项目来说，查看该软件较旧版本的已知漏洞，就可以让你轻松地找到你想要的各种资料。

在这次调查中，我首先查看了rce漏洞S2-032 (CVE-2016-3081), S2-033 (CVE-2016-3687)和S2-037
(CVE-2016-4438)。与Struts中的许多其他RCEs一样，这些用户们不可靠的输入被认定为OGNL表达式，该表达式允许攻击者在服务器上运行任意代码。这三个漏洞特别有趣

这三个问题都是让methodName作为OgnlUtil::getValue()的参数，从而传递远程输入。

```
String methodName = proxy.getMethod();    //<--- untrusted source, but where from?
LOG.debug("Executing action method = {}", methodName);
String timerKey = "invokeAction: " + proxy.getActionName();
try {
   UtilTimerStack.push(timerKey);
   Object methodResult;
   try {
       methodResult = ognlUtil.getValue(methodName + "()", getStack().getContext(), action); //<--- RCE
```

这里的proxy包括ActionProxy，这是一个接口。从它的定义看，除了getMethod()(它在上面的代码中用来表示受污染的变量methodName)还有各种各样的方法，比如get

现在，我们可以开始使用QL对这些不受信任的源进行建模：

```
class ActionProxyGetMethod extends Method {
 ActionProxyGetMethod() {
   getDeclaringType().getASupertype*().hasQualifiedName("com.opensymphony.xwork2", "ActionProxy") and
   (
     hasName("getMethod") or
     hasName("getNamespace") or
     hasName("getActionName")
   )
 }
}

predicate isActionProxySource(DataFlow::Node source) {
  source.asExpr().(MethodAccess).getMethod() instanceof ActionProxyGetMethod
}
```

识别OGNL接收器

现在我们已经确定了一些不受信任的来源，下一步是对接收器执行相同的操作。如前所述，许多StrutsRCEs将远程输入解析为OGNL表达式。Struts中有许多函数都将其参数
(CVE-2017-5638)中,
TextParseUtil::translateVariables()被使用了。我们可以寻找用于执行OGNL表达式的通用函数，而不仅仅是将这些方法名描述为QL中的单独接收器。我认为OgnlUtil::com

我在一个QL的predicate中对它们进行了描述，如下所示：

```
predicate isOgnlSink(DataFlow::Node sink) {
 exists(MethodAccess ma | ma.getMethod().hasName("compileAndExecute") or ma.getMethod().hasName("compileAndExecuteMethod") |
   ma.getMethod().getDeclaringType().getName().matches("OgnlUtil") and
   sink.asExpr() = ma.getArgument(0)
 )
}
```

第一次尝试追踪污点

现在我们已经在QL中定义了sources and sinks，我们可以在一个进行污染跟踪的查询中使用这些定义。我们通过定义一个DataFlow Configuration，并使用DataFlow library来做到它:

```
class OgnlTaintTrackingCfg extends DataFlow::Configuration {
 OgnlTaintTrackingCfg() {
   this = "mapping"
 }

 override predicate isSource(DataFlow::Node source) {
   isActionProxySource(source)
 }

 override predicate isSink(DataFlow::Node sink) {
   isOgnlSink(sink)
 }

 override predicate isAdditionalFlowStep(DataFlow::Node node1, DataFlow::Node node2) {
   TaintTracking::localTaintStep(node1, node2) or
   exists(Field f, RefType t | node1.asExpr() = f.getAnAssignedValue() and node2.asExpr() = f.getAnAccess() and
     node1.asExpr().getEnclosingCallable().getDeclaringType() = t and
     node2.asExpr().getEnclosingCallable().getDeclaringType() = t
   )
 }
}

from OgnlTaintTrackingCfg cfg, DataFlow::Node source, DataFlow::Node sink
where cfg.hasFlow(source, sink)
select source, sink
```

在这里，我使用先前定义的isActionProxySource和isOgnlSinkpredicate。

注意，我还重写了一个名为isAdditionalFlowStep的predicate，它允许我可以囊括一些污染数据。例如，它允许我将特定项目的信息合并到流程配置中。比如说，如果我有
Library跟踪受污染的数据。

对于这个特定的查询，我为DataFlow library添加了两个额外的流程步骤。第一项：

```
TaintTracking::localTaintStep(node1, node2)
```

包括通过标准Java library调用、字符串操作等，跟踪标准QLTaintTracking library。第二个附加部分是一个近似值，它允许我通过字段跟踪受污染的数据：

```
exists(Field f, RefType t | node1.asExpr() = f.getAnAssignedValue() and node2.asExpr() = f.getAnAccess() and
 node1.asExpr().getEnclosingCallable().getDeclaringType() = t and
 node2.asExpr().getEnclosingCallable().getDeclaringType() = t
)
```

这意味着，如果一个字段被分配给某个受污染的值，只要这两个表达式都是由相同类型的方法调用的，那么访问该字段的话也就将被认为是污染的。大概来说，这包括以下情

```
public void foo(String taint) {
 this.field = taint;
}

public void bar() {
 String x = this.field; //x is tainted because field is assigned to tainted value in `foo`
}
```

如您所见，访问this.field in bar()并不一定会被污染。例如，在bar()之前，if
foo()并不会被调用。因此，默认情况下不包括DataFlow::Configuration这个流程步骤，因为我们不能保证数据总是以这种方式流动。但是，对于搜索漏洞，我发现这个附加

初始结果和查询细化

之前我对最新版本的源代码运行了查询，并查看了结果，我注意到引起s2-032，s2-033和s2-037的原因仍然被查询标记。在查看它发现的其他结果之前，我想了解在代码固

如果最初通过净化输入来修复漏洞，在s2-037之后，Struts团队决定用OgnlUtil::callMethod()替换OgnlUtil::getValue().

```
methodResult = ognlUtil.callMethod(methodName + "()", getStack().getContext(), action);
```

callMethod()覆盖了对compileAndExecuteMethod()的调用:

```
public Object callMethod(final String name, final Map<String, Object> context, final Object root) throws OgnlException {
 return compileAndExecuteMethod(name, context, new OgnlTask<Object>() {
   public Object execute(Object tree) throws OgnlException {
```

```
    return Ognl.getValue(tree, context, root);
  }
 });
}
```

在执行它们之前,compileAndExecuteMethod()将会执行对表达式的附加检查：

```
private <T> Object compileAndExecuteMethod(String expression, Map<String, Object> context, OgnlTask<T> task) throws OgnlExcept
 Object tree;
 if (enableExpressionCache) {
   tree = expressions.get(expression);
   if (tree == null) {
     tree = Ognl.parseExpression(expression);
     checkSimpleMethod(tree, context); //<--- Additional check.
   }
```

这意味着我们可以将·compileAndExecuteMethod()从我们的接收器中移除。

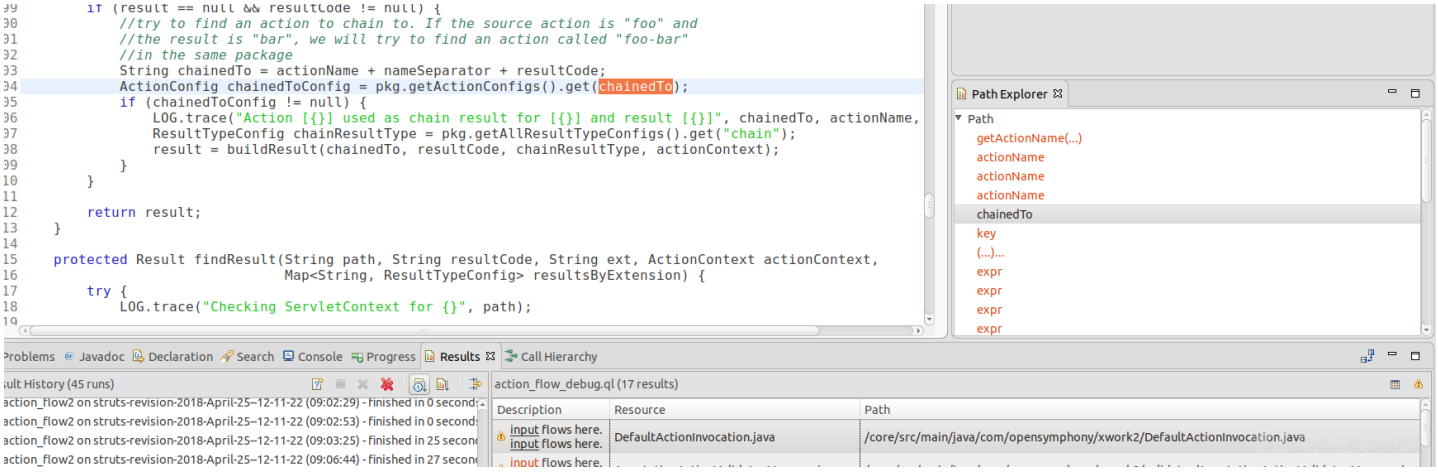
在重新运行查询后，我之前曾强调过调用getMethod()作为接收器，但是现在这个结果消失了。然而，仍然有一些结果突出了DefaultActionInvocation.java代码，这些代码

路径探索与查询细化

为了研究为什么要标记这个结果，我需要能够看到DataFlow
library生成这个结果的每个单独的流程。QL允许您编写特殊的path-problem，从而生成长度可变的路径的查询，这些路径可以逐节点查看，DataFlow
library允许您编写输出此数据的查询。

在撰写这篇博文时，lgtm本身还没有查询路径问题的用户界面，因此我需要使用另一个应用程序Semmle：Eclipse
QL。这是一个Eclipse插件，它包含了一个可视化工具，可以帮助您完成污染跟踪中的各个步骤。你可以按照指示免费下载并安装此Eclipse插件。它不仅允许离线分析LGTM
Git存储库中找到。您可以按照README.md在Eclipse插件中运行它们的文件。从现在开始，我将囊括来自QL for Eclipse的截图。

首先，在initial.ql中执行查询。在QL for Eclipse中，一旦从DefaultActionInvocation.java中选取了结果，你就可以看到从源到接收器的详细路径。



在上面的图像中，您可以看到经过几步之后，调用getActionName()返回的值，将会继续流入调用pkg.getactionconfigs ( )返回的对象的get ( )的参数:

```
String chainedTo = actionName + nameSeparator + resultCode; //actionName comes from `getActionName` somewhere
ActionConfig chainedToConfig = pkg.getActionConfigs().get(chainedTo); //chainedTo contains `actionName` and ended up in the `g
```

点击下一步，key，进入ValueStackShadowMap::get()：

```
public Object get(Object key) {
 Object value = super.get(key);  //<--- key gets tainted?

 if ((value == null) && key instanceof String) {
   value = valueStack.findValue((String) key);  //<--- findValue ended up evaluating `key`
 }

 return value;
}
```

由于pkg.getActionConfigs()返回了一个Map，并且ValueStackShadowMap实现了Map的连接，从理论上讲，pkg.getActionConfigs()返回的值可能是ValueStackShado
DataFlow library显示了从变量chainedTo，到类value stackshadowmap中，实现get( )的潜在流程。在实践中，类ValueStackShadowMap属于jasper
reports插件，并且这种类只有在某些情况下会被创建，而这其中没有一个是由pkg.getActionConfigs()返回的。在我看到了这个问题之后，我明白ValueStackShadowMap

```
override predicate isBarrier(DataFlow::Node node) {
 exists(Method m | (m.hasName("get") or m.hasName("containsKey")) and
```

```
   m.getDeclaringType().hasName("ValueStackShadowMap") and
   node.getEnclosingCallable() = m
 )
}
```

这个predicate标明，如果污染数据流入get()或containsKey()方法名ValueStackShadowMap，那就不要继续追踪它。(我添加了containsKey()方法名，因为它存在相同的

在给ActionMapping::toString()进一步增加barrier后(toString()再被任意对象调用时它都将产生问题)，我重新运行查询，但这只给我们留下了很少的结果。您也可以尝试使

新的漏洞

只有10对相关资料，因此这很容易通过手工检查。查看这些路径，我发现有一些路径是无效的，有的是因为它们在测试用例中，所以我在查询中添加了一些障碍以过滤掉这些

就拿ServletActionRedirectResult.java中的来说：

在第一步中，来自调用getNamespace()的源通过变量namespace流入函数ActionMapping的构造参数：

```
public void execute(ActionInvocation invocation) throws Exception {
 actionName = conditionalParse(actionName, invocation);
 if (namespace == null) {
   namespace = invocation.getProxy().getNamespace();  //<--- source
 } else {
   namespace = conditionalParse(namespace, invocation);
 }
 if (method == null) {
   method = "";
 } else {
   method = conditionalParse(method, invocation);
 }

 String tmpLocation = actionMapper.getUriFromActionMapping(new ActionMapping(actionName, namespace, method, null)); //<--- nam

 setLocation(tmpLocation);
```

进一步可以看到，getUriFromActionMapping()返回了使用namespace构造的URL字符串ActionMapping，然后通过变量tmpLocation进入setLocation()的参数中:

setLocation()然后在super class StrutsResultSupport中设置字段location:

```
public void setLocation(String location) {
   this.location = location;
}
```

然后，在ServletActionResult中调用execute():

```
String tmpLocation = actionMapper.getUriFromActionMapping(new ActionMapping(actionName, namespace, method, null));

setLocation(tmpLocation);

super.execute(invocation);
```

通过location字段调用conditionalParse():

```
public void execute(ActionInvocation invocation) throws Exception {
   lastFinalLocation = conditionalParse(location, invocation);
```

```
   doExecute(lastFinalLocation, invocation);
}
```

接着conditionalParse()经过location进入translateVariables()，这将param评价为引擎盖下的OGNL表达式:

```
protected String conditionalParse(String param, ActionInvocation invocation) {
   if (parse && param != null && invocation != null) {
       return TextParseUtil.translateVariables(
           param,
           invocation.getStack(),
           new EncodingParsedValueEvaluator());
   } else {
       return param;
   }
}
```

所以,当ServletActionRedirectResult中没有设置namespace参数时，代码从ActionProxy获取namespace，然后将其作为OGNL表达式进行评估。为了测试这个，在演示应

```
<struts>
   <package name="actionchaining" extends="struts-default">
       <action name="actionChain1" class="org.apache.struts2.showcase.actionchaining.ActionChain1">
          <result type="redirectAction">
            <param name = "actionName">register2</param>
          </result>
       </action>
   </package>
</struts>
```

然后我在本地运行showcase应用程序，然后访问一个URL，该URL旨在触发此漏洞并执行shell命令,从而在我的计算机上打开计算器应用程序。

这起了作用(花了一些时间绕过OGNL沙箱之后)。在这个阶段，更多的细节我将会在稍后的一个恰当的时机揭露它们。

不仅如此，还有来自ctionChainResult,
PostbackResult和ServletUrlRenderer等的一些不可信的来源，它们也起作用了！PortletActionRedirectResult可能也能用，但我没有测试。四个RCEs已经足以证明问题的

结论

在这篇文章中，我展示了通过使用已知的漏洞来帮助构建应用程序的污染模型，而你只需将困难的工作留给QL DataFlow
library就可以发现新的漏洞。特别是，通过研究Struts中前三个RCEs，我们最终发现了另外四个!

鉴于S2-032、S2-033和S2-037都是在短时间内发现的，安全研究人员研究了S2-032以寻找类似的问题，并发现了S2-033和S2-037。所以这里最大的问题是：考虑到我在这

如果你认为这更像是一种侥幸，因为我假设ActionProxy中的namespace字段是突然被污染的，那么请继续关注下一篇文章，这是我接下来将要详细介绍的，并根据首要原则

■■■■■https://lgtm.com/blog/apache_struts_CVE-2018-11776