jenkins 无限制 rce 分析

orich1 / 2019-01-27 00:11:00 / 浏览数 8662 安全技术 漏洞分析 顶(2) 踩(0)

声明：仅用于技术研究，不恰当使用造成的危害后果自负

简言

首先，文章里没有直接 rce 的exp，想要exp的大哥抱歉了23333
但是由于执行了 groovy 代码，所以瞎j2搞的话我也不知道会出啥问题

挖jenkins好几个月了，一直莫得比较好用的洞，12.05 公布越权动态调用就搞得心里痒痒，给重新挖了下，此文章写的比较随意，大佬轻喷，欢迎交流

分析过程

先讲讲偷窥思路

从 orange 大佬12.20公布发现了一个 jenkins 未授权 rce 开始，我就一直在试图将其挖掘出来，一直到 1.16 大佬公布第一部分 Jenkins 动态路由利用这篇文章，才真正拿到触发链，不过到最后 groovy sandbox 绕过实在是不会了.....

在 orange 的文章中，其实帮助最大的还是贴出了官方的一个漏洞通告链接（没有收集漏洞通报的良好习惯2333）和对 Descriptor 的理解还有利用
官方在 1.8 号公布了一个通告：https://jenkins.io/security/advisory/2019-01-08/#jenkins-security-advisory-2019-01-08

大致讲的是 pipeline 这个插件里对groovy脚本进行解析的时候会出现 sandbox 被绕过的情况以及 REST API 也会直接访问到，但是他是需要 Overall/Read 权限的

这里我思考了一会儿，虽然这通告涉及到 bypass groovy sandbox，一定程度上和 RCE 有关联，但是它需要权限的。Overall/read 在jenkins 中属于比 ANONYMOUS 权限高一丁点的权限，但是它默认是 FALSE 的，就是默认配置安装的 Jenkins 是没有 Overall/read 权限的，不登录的情况下只有 ANONYMOUS 权限，然后呢我们需要的是一个未授权 RCE
，所以这里很有可能是一个最后的一个任意代码执行的地方，那么还得需要去寻找绕过权限检查的触发路由（这里我没有对 REST API 直接访问导致的 RCE 做研究）

在更早的时候，官方有通告如下：
https://jenkins.io/security/advisory/2018-12-05/
这也是 CVE-2018-1000861 ，造成的影响呢是能够一定程度上调用 Jenkins 中的任意 getter 函数，造成了越权的情况

将两者结合起来的话就很有可能是未授权 RCE
首先因为我之前一直在搞 Jenkins
的反序列化黑名单，所以对其路由解析过程算是比较熟悉（因为不熟悉ACL机制，甚至当时根本没听说过，导致没有察觉到这个任意 getter 调用的漏洞点，只是觉得jenkins的路由映射做的很奇怪233333），所以能在拿到官方通报后就能猜到整体触发流程，接下来我会先对 CVE-2018-1000861 做一点简单的分析，然后再从寻找 RCE 的角度去分析挖掘方式

然后还有想说的就是，从通告中可以直接拿到插件代码diff，可以很方便的找到漏洞点

动态路由形成过程

CVE-2018-1000861 其实还是和之前 orange 大佬发现的 Jenkins 任意文件读取相关核心： Stapler 有关系
它对于 Jenkins 来说就是一个小型路由生成器
完完全全可以直接从 web.xml 开始跟入 Stapler 类中 service 函数的，因为官方补丁diff的话，反而找不到 Jenkins 路由生成过程，对后面的漏洞挖掘会造成一定理解上的困难
我不贴 service 函数的代码，简单说说就好，service 函数中最后调用到了 invoke 函数，一直跟着 invoke 关键字的话，会进入到 Stapler 中的 tryInvoke 函数中，关键代码块如下：

```
729
730     MetaClass metaClass = webApp.getMetaClass(node);
731
732     try {
733         for( Dispatcher d : metaClass.dispatchers ) {
734             if(d.dispatch(req,rsp,node)) {
735                 if(LOGGER.isLoggable(Level.FINER))
736                     LOGGER.finer( msg: "Handled by "+d);
737                 return true;
738             }
739         }
740     } catch (IllegalAccessException e) {
```

大致意思是，会得到 MetaClass 的一个对象，然后对请求包进行轮询调用其中的 dispatcher.disspatch，这个 dispatcher 就是一个个"节点"下的小路由，比如说一个请求的url：
http://target/123/abc
那么它先对 123 这个节点匹配对应的 dispatcher ，然后在 dispatch 中进行反射调用具体函数，然后再对 abc 这个节点做匹配以此类推

那么关键就是 dispatcher 的生成了，跟进 getMetaClass 函数

```
198
199     public MetaClass getMetaClass(Klass<?> c) {
200         if(c==null)     return null;
201         synchronized(classMap) {
202             MetaClass mc = classMap.get(c);
203             if(mc==null) {
204                 mc = new MetaClass( webApp: this,c);
205                 classMap.put(c,mc);
206             }
207             return mc;
208         }
209     }
```

主要是根据传入的 node 变量生成了一个 MetaClass 对象，node 变量经过了一定包装，大致是获取了类相关信息，继续跟进 MetaClass 看看

```
87  @   /*package*/ MetaClass(WebApp webApp, Klass<?> klass) {
88          this.clazz = klass.toJavaClass();
89          this.klass = klass;
90          this.webApp = webApp;
91          this.baseClass = webApp.getMetaClass(klass.getSuperClass());
92          this.classLoader = MetaClassLoader.get(clazz.getClassLoader());
93          buildDispatchers();
94      }
95
```

记录 node 的各种信息，然后调用 buildDispatchers 函数，函数体太长总共301行就不贴出来了，这个函数主要功能就是对 node 对应的 class 、此 class的父类、此 class继承类（简单来说就是继承家族树中的所有类）进行一个函数信息提取，然后获取指定的函数相关信息，做一个函数反射调用和 url 节点名称的存储，存储在 MetaClass.dispatchers 中，这就是制作路由的过程了，也是 orange 文章中提到的，如下：

整個漏洞要從 Jenkins 的 動態路由 講起, 為了給開發者更大的彈性, Jenkins(嚴格來講是 Stapler)使用了一套 Naming Convention 去匹配路由及動態的執行! 首先 Jenkins 以 / 為分隔將 URL 符號化, 接著由 jenkins.model.Jenkins 為入口點開始往下搜尋, 如果符號符合 (1) Public 屬性的成員或是 (2) Public 屬性的方法符合下列命名規則, 則調用並繼續往下呼叫:

1. get()
2. get(String)
3. get(Int)
4. get(Long)
5. get(StaplerRequest)
6. getDynamic(String, …)
7. doDynamic(…)
8. do(...)
9. js(...)
10. 擁有 @WebMethod 標註的方法
11. 擁有 @JavaScriptMethod 標註的方法

只要在继承家族树中，任意类满足在以上 11 种规则的函数，统统可以直接在 URL 中访问到（get的意思是以 get 开始的名字，do 和 js 同理）

然后呢，我随便截一个 dispatcher 的生成过程，如下图：

那么这里整个就是一次对当前 node 进行动态路由制作的过程了，如果思路延展一下，可以发现整个是一个迭代的过程，我稍微描述下：
访问 http://target/123/abc

先解析 root 节点（也就是第一次传入的 node ），然后对 123 做适配，匹配到的是满足上图中 11 中规则的并且存在于当前 root 节点家族树的函数，并对他进行反射调用，目标函数流程走完了后，根据返回结果类型进行下一步处理，因为在上图中的 req.getStapler().invoke 调用中，目标函数的返回结果传递给了第三个形参，如下：

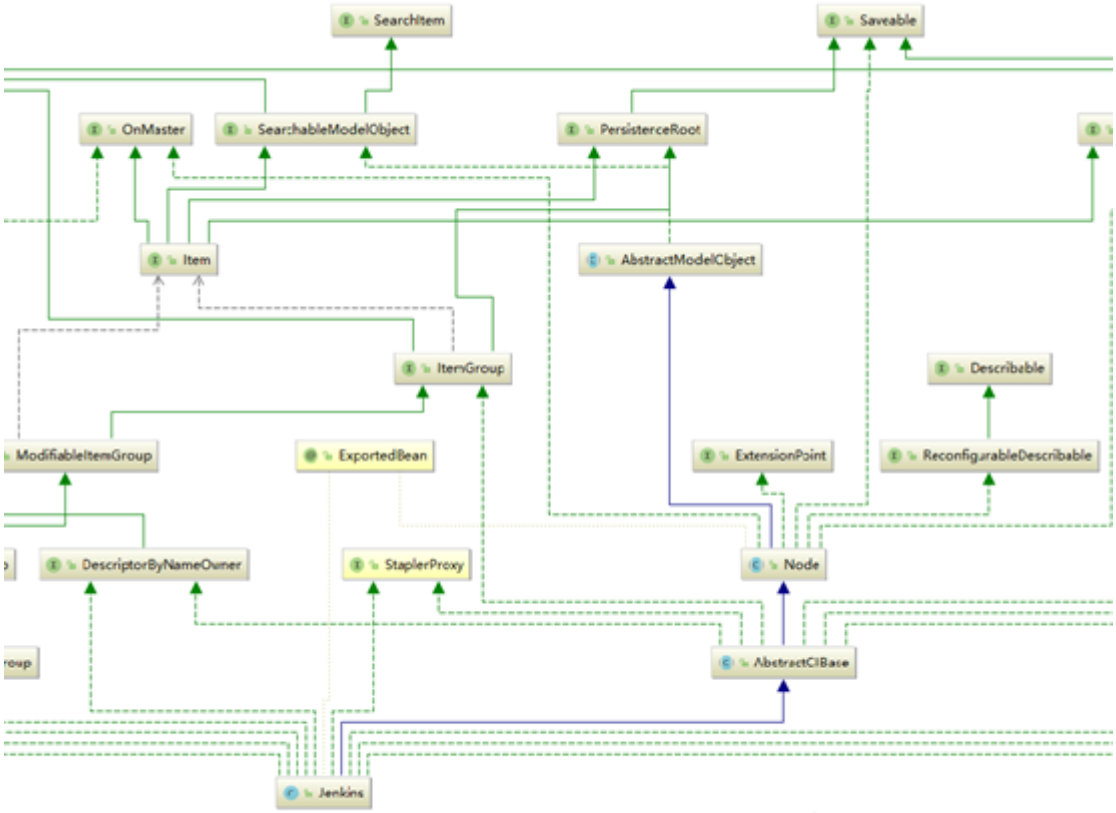
```
 */
void invoke(RequestImpl req, ResponseImpl rsp, Object node ) throws IOException, ServletException {
    if(node==null) {
        // node is null
        if(!Dispatcher.isTraceEnabled(req)) {
```

其实就是一个新的 node ，那么对 abc 做适配的，就是新 node 的家族树中的函数了，以此递归下去，直到匹配出错或者所有节点匹配完成

那现在我们看一看，谁是 root node ：

- 如果 url 是以 /$stapler/bound/ 开头的话，就 org.kohsuke.stapler.bind.BoundObjectTable 为 root
- 如果其他的话就是以 Jenkins.model.Jenkins 为 root

上者和 Object 绑定功能相关，默认情况下是不会有什么利用点的
稍微看看 Jenkins 类的一部分家族树，如下图：



所以我们访问 Jenkins 的时候，第一个解析的路由节点，就是上图中满足那11个规则的函数

现在问题来了，这些类中所有满足规则的函数都能访问吗，那岂不是几乎没有任何限制了。

绕过路由访问限制

那么限制在哪儿呢？
还是在 Stapler 类中 tryInvoke 函数中，函数一开头就做了一个操作如下：

```
boolean tryInvoke(RequestImpl req, ResponseImpl rsp, Object node ) throws IOException, ServletException {
    if(traceable())
        traceEval(req,rsp,node);

    if(node instanceof StaplerProxy) {
        if(traceable())
            traceEval(req,rsp,node, prefix: "((StaplerProxy)", suffix: ").getTarget()");
        Object n = null;
        try {
            n = ((StaplerProxy)node).getTarget();
        } catch (RuntimeException e) {
            if (function.renderResponse(req,rsp,node,e))
                return true; // let the exception serve the request and we are done
            else
                throw e;    // unprocessed exception
        }
```

如果是 StaplerProxy 的实现类，那么就会调用当前 node 的 getTarget() 函数，从家族树中看见 Jenkins 这个类确实实现了 StaplerProxy，那么它的 getTraget 函数如下：

```
public Object getTarget() {
    try {
        checkPermission(READ);
    } catch (AccessDeniedException e) {
        if (!isSubjectToMandatoryReadPermissionCheck(Stapler.getCurrentRequest().getRestOfPath())) {
            return this;
        }

        throw e;
    }
    return this;
}
```

上图中会检查当前用户是否拥有 READ 权限，如果没有的话会抛出异常，然后进入 isSubjectToMandatoryReadPermissionCheck 函数中，并且带入了当前访问的路由节点名，如下：

```
public boolean isSubjectToMandatoryReadPermissionCheck(String restOfPath) {
    for (String name : ALWAYS_READABLE_PATHS) {
        if (restOfPath.startsWith(name)) {
            return false;
        }
    }

    for (String name : getUnprotectedRootActions()) {
        if (restOfPath.startsWith("/" + name + "/") || restOfPath.equals("/" + name)) {
            return false;
        }
    }

    // TODO SlaveComputer.doSlaveAgentJnlp; there should be an annotation to request unprotected access
    if (restOfPath.matches( regex: "/computer/[^/]+/slave-agent[.]jnlp")
        && "true".equals(Stapler.getCurrentRequest().getParameter( s: "encrypt"))) {
        return false;
    }

    return true;
}
```

满足上面三个条件的路由节点名，都会放行当前请求通过，如果都不满足则记录请求并转到 login 窗口

查看一下 ALWAYS_READABLE_PATHS ：

```
/**
 * Urls that are always visible without READ permission.
 *
 * <p>See also:{@link #getUnprotectedRootActions}.
 */
private static final ImmutableSet<String> ALWAYS_READABLE_PATHS = ImmutableSet.of(
        "/login",
        "/logout",
        "/accessDenied",
        "/adjuncts/",
        "/error",
        "/oops",
        "/signup",
        "/tcpSlaveAgentListener",
        "/federatedLoginService/",
        "/securityRealm",
        "/instance-identity"
);
```

上图中的节点路由都可以通过，不过这数量少得可怜。但是其中 securityRealm 就是绕过 ACL 限制的跳板入口。

这里不禁想起来自己挖洞的时候记录2333333，如下：

```
ALWAYS_READABLE_PATHS = ImmutableSet.of(
        "/login",
        "/logout",
        "/accessDenied",
        "/adjuncts/",
        "/error",
        "/oops",
        "/signup",
        "/tcpSlaveAgentListener",
        "/federatedLoginService/",
        "/securityRealm",
        "/instance-identity"
    );
以上的路由都访问过，没有大作用
```

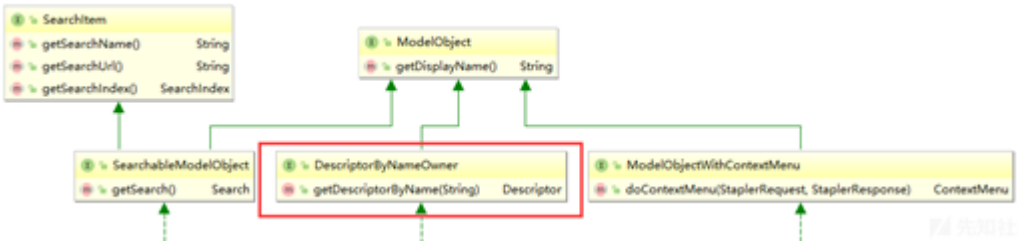
疯狂打脸233333，因为当时草草看了下，一心对反序列化黑名单着迷，没有对相关函数的返回类型做家族树调查

跳-跳-跳

对 /securityRealm 访问时进行动态调试发现，返回的是 Hudson.security.HudsonPrivateSecurityRealm 类，我们跟过去，查看其中的 getUser 函数，如下：

```
    public User getUser(String id) {
        return User.getById(id, create: true);
    }
```

这里根据传入的下一节点名当做 id，然后生成一个 User 出来，稍微测试了下，不存在的用户也能正常生成 User，未对这个原因进行深究，此时目标函数返回的是 User 类。我们看看 User 类的家族树，找到一个关键点如下：



查看 getDescriptorByName 如下：

```
public interface DescriptorByNameOwner extends ModelObject {
    default Descriptor getDescriptorByName(String id) {
        return Jenkins.getInstance().getDescriptorByName(id);
    }
}
```

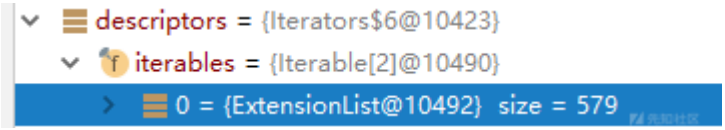
其实就是调用了 Jenkins.getDescriptorByName，这个函数主要根据传入的 id（String），然后获取到程序中所有继承了 Descriptor 的子类

总结下这里的利用类连续跳动过程：
`Jenkins -> HudsonPrivateSecurityRealm -> User -> DescriptorByNameOwner -> Jenkins -> Descriptor`

Descriptor 绕过 ACL 的主角

Descriptor ，从这个类名都能感觉到，是描述功能相关，并且其中拥有大量的 getter ，从设计上思考的话，这个 Descriptor
很有可能是会对很多功能点的相关描述

动态调了下，默认配置的 Jenkins 拥有约579个 Descriptor



如何寻找 RCE

其实在研究动态路由的过程中，就发现了，想要 RCE 还是要依靠插件中的一些脚本解析功能才行，但是突然懒癌发作，看着一堆插件就不想动手去分析了23333

这里我们还是简单一点，根据官方漏洞通告寻找补丁diff：
https://github.com/jenkinsci/workflow-cps-plugin/commit/d09583eda7898eafdd15297697abdd939c6ba5b6

从中看见几个修改的类文件：
src/main/java/org/jenkinsci/plugins/workflow/cps/CpsFlowDefinition.java
src/main/java/org/jenkinsci/plugins/workflow/cps/CpsGroovyShellFactory.java
src/main/java/org/jenkinsci/plugins/workflow/cps/replay/ReplayAction.java

稍微筛选下，就能找出 CpsFlowDefinition 才是主角（虽然通过 ReplayAction 也能够触发，但是根据我的跟踪中发现需要一定权限才可以）
查看关键点 CpsFlowDefinition$DescriptorImpl 如下：

```
@Extension
public static class DescriptorImpl extends FlowDefinitionDescriptor {

    @Override
    public String getDisplayName() { return "Pipeline script"; }

    public FormValidation doCheckScript(@QueryParameter String value, @QueryParameter boolean sandbox) {
        return sandbox ? FormValidation.ok() : ScriptApproval.get().checking(value, GroovyLanguage.get());
    }

    public JSON doCheckScriptCompile(@QueryParameter String value) {
        try {
            CpsGroovyShell trusted = new CpsGroovyShellFactory( execution: null).forTrusted().build();
            new CpsGroovyShellFactory( execution: 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)
    }

}
```

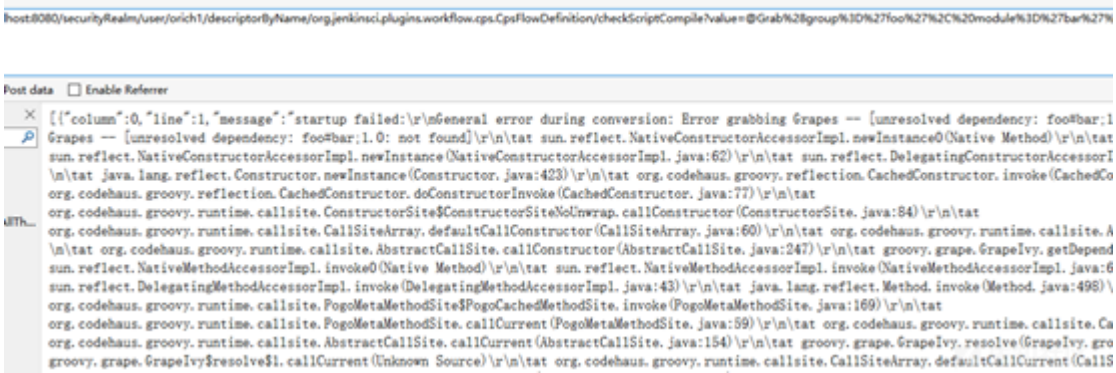
继承的 FlowDefinitionDescriptor ，这个类继承自 Descriptor，上图中有两个满足那11个规则的函数，其中带上 @QueryParameter 注解的参数都可以通过参数请求传递进来

OK，到此为止已经拿到了 Jenkins 的无限制 RCE 触发链，但是最终它是解析 Groovy 脚本的，并且似乎上了沙盒，虽然官方补丁diff中含有一点 bypass sandbox 的技术点，但是我对 groovy 是一窍不通，搞了好几天都没办法，各位师傅如果有经验的话，试试呢？

目前为止的效果

都是官方补丁diff的bypass

使用 Grab 注解如下：



使用 ASTTest 注解如下：

总结（我菜如狗）

Stapler 的动态路由制作过程

Jenkins 本身的白名单路由

Descriptor 的利用，这里的利用过程相当曲折，从 Jenkins 入口跳出去最终再跳到 Jenkins 自己这里获取 Descriptor ，然后再从各种继承类中寻找到 groovy 解析的利用点，膜 orange

Groovy sandbox 的绕过，实在是不会弄了，Orz，求大哥们教教

Links：

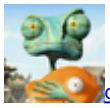
hacking-Jenkins-part1-play-with-dynamic-routing：
https://devco.re/blog/2019/01/16/hacking-Jenkins-part1-play-with-dynamic-routing/
jenkins官方通告：
https://jenkins.io/security/advisory/2019-01-08/#jenkins-security-advisory-2019-01-08
https://jenkins.io/security/advisory/2018-12-05/
pipeline-groovy插件相关：
https://plugins.jenkins.io/workflow-cps
https://github.com/jenkinsci/workflow-cps-plugin/commit/d09583eda7898eafdd15297697abdd939c6ba5b6

上一篇：过D盾webshell分享 下一篇：安全工具——wfuzz
1. 2 条回复


leveryd 2019-01-29 14:45:42

大佬有没有挖别的Java框架的？

0 回复Ta


orich1 2019-02-09 10:59:01

@leveryd 挖不动挖不动

0 回复Ta
