VulnBox: 1

Gz

Contents

1 About the box

This is a **b2r** (*boot2root*) easy/medium machine challenge which intends to show some common but less known features like `sg`, `doas` and `cmp`.

Requirements:

- Web enumeration
- Medium SQLi exploitation
- Linux knowledge
- System enumeration
- Hash cracking

2 Solution

The machine is an easy level, but not that straightforward, it's intended to work with tools or techniques I've seen very little out there so far.

2.1 Reconnaissance and Enumeration

Let's get started by discovering the IP address of our target:

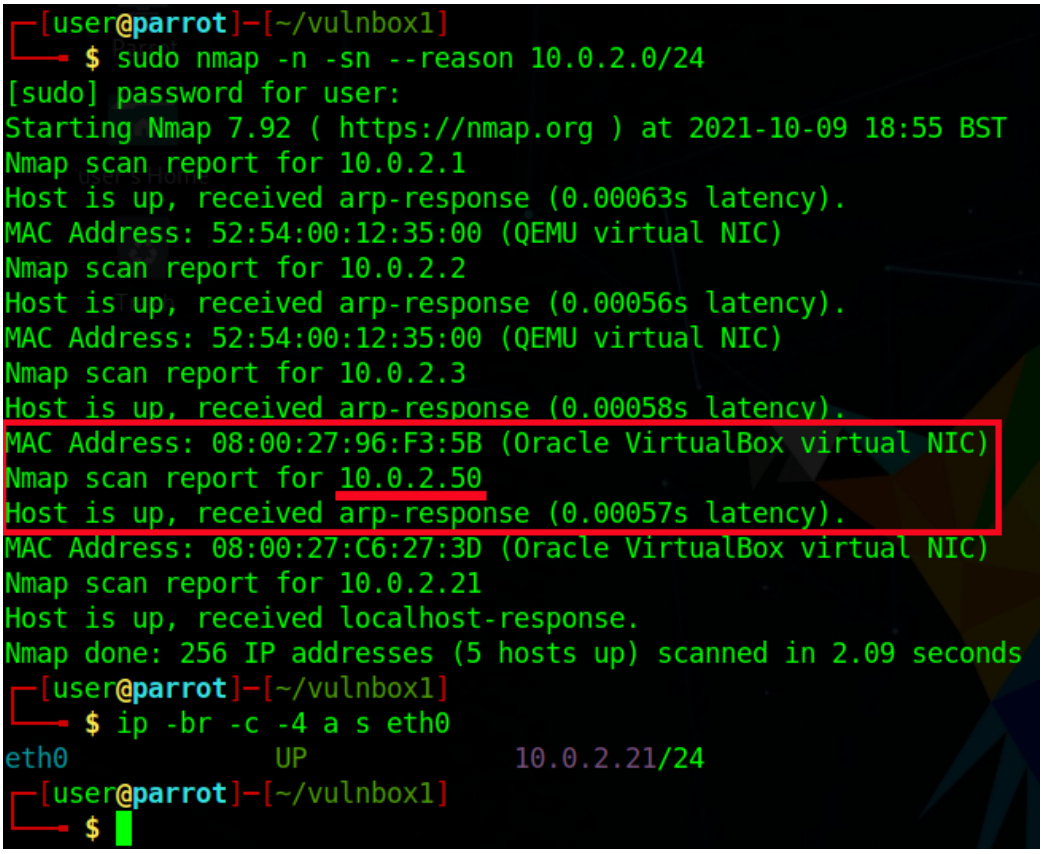


Figure 1: Host Discovery with `nmap`

My IP address is `10.0.2.21`, the first three are from *VirtualBox*, and the remain one is our target's (`10.0.2.50`).

Let's see what ports it has open. I start with **TCP** ports as most services leverage this protocol because of its reliability. If we didn't find anything interesting we'd move on and would scan the **UDP** ports.

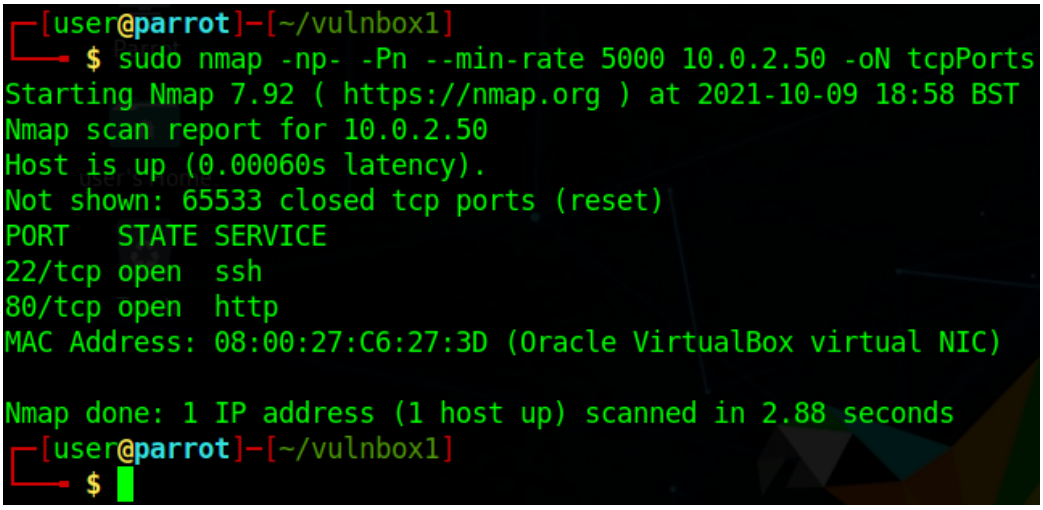


Figure 2: Port Scanning: **TCP**

We know now what ports are open, and they might be our possible way in. `nmap` shows those ports with the *default* protocol associated. Which in most cases matches the reality, but in some others, they don't. They may be changed to be "*hidden*"[1].

However, `nmap` can perform more advanced scans to fingerprint the services according to the responses received:

[1]This is known as "*Security through obscurity*".

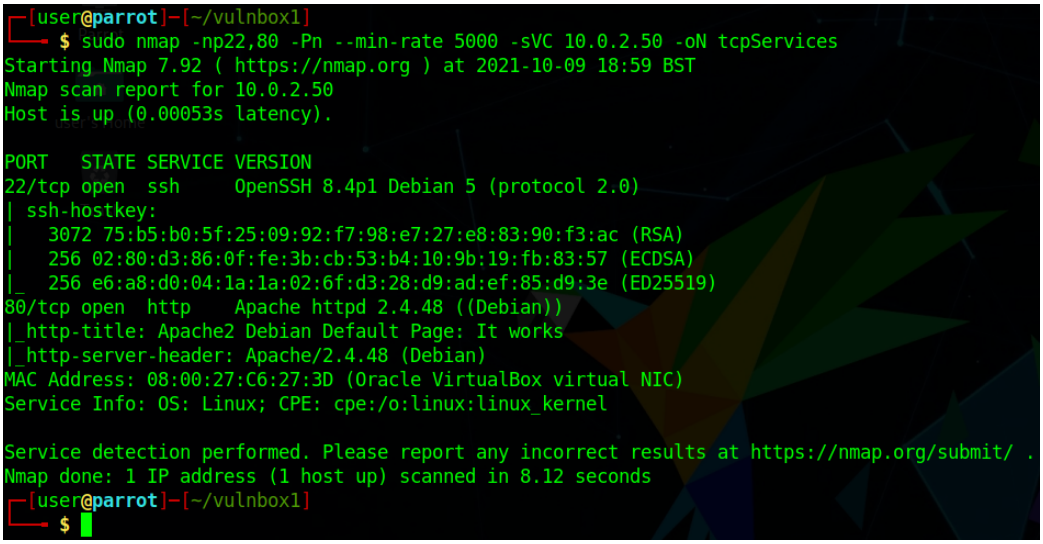


Figure 3: Service Fingerprinting: **TCP**

We've got some important information already:

- `22/tcp: OpenSSH 8.4p1 Debian 5 (protocol 2.0)`
- `80/tcp: Apache httpd 2.4.480 (Debian)`

We're before, what it looks like, a *Debian* distro. And we know the versions of both the **SSH** and web server.

Since there's a web server, we can use a web browser to see the contents:

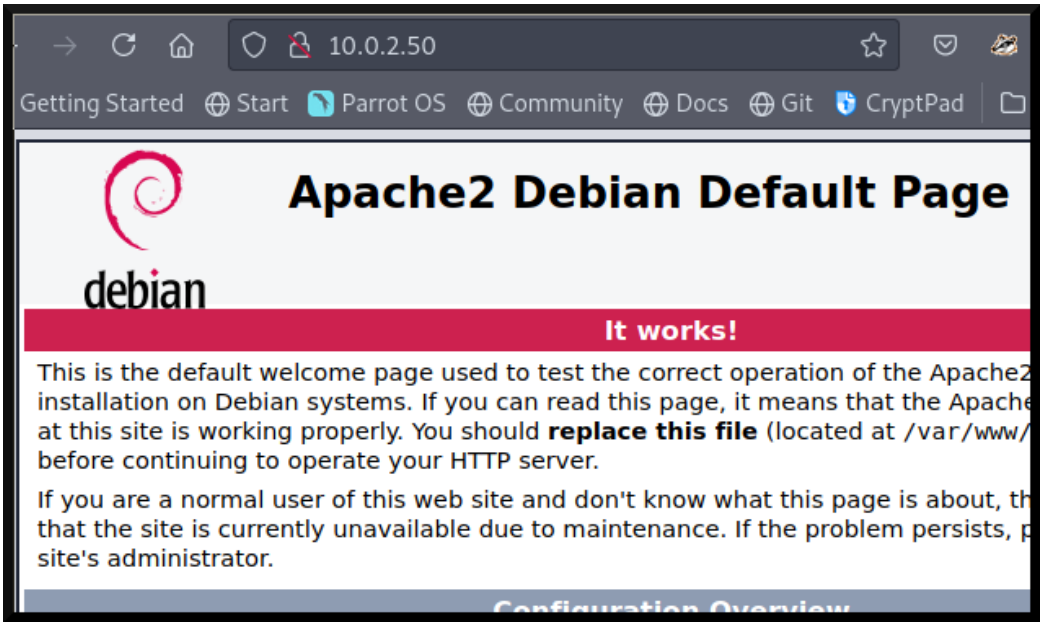


Figure 4: Initial page: *Apache*'s default page

This is the *Apache*'s default page. We can use `whatweb` to see quickly if there's a technology in use we don't see:

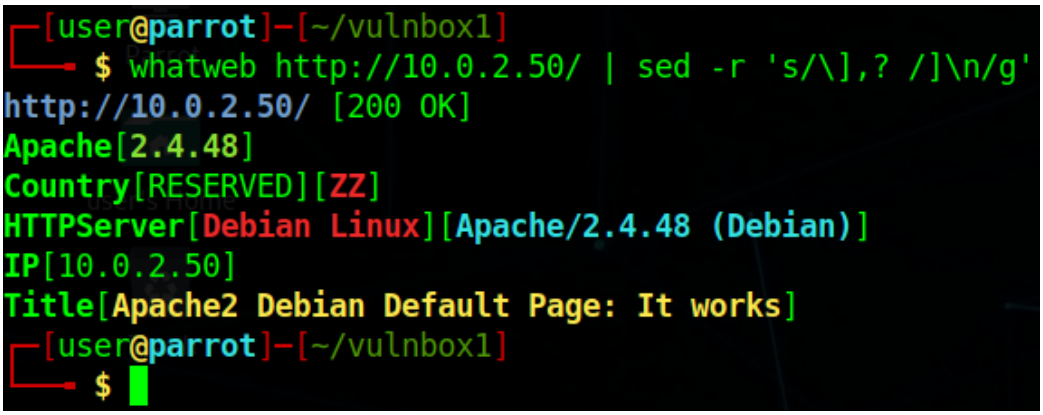


Figure 5: `whatweb` output on the initial page

Nothing out of the ordinary.

We have no choice but start fuzzing. Let's start with possible resources. To save time I try to search with the extensions `.txt`, as it's a very common extension in any operating system I can think of, and `.php`, as the web server is *Apache*, and its web applications are commonly written in this language.

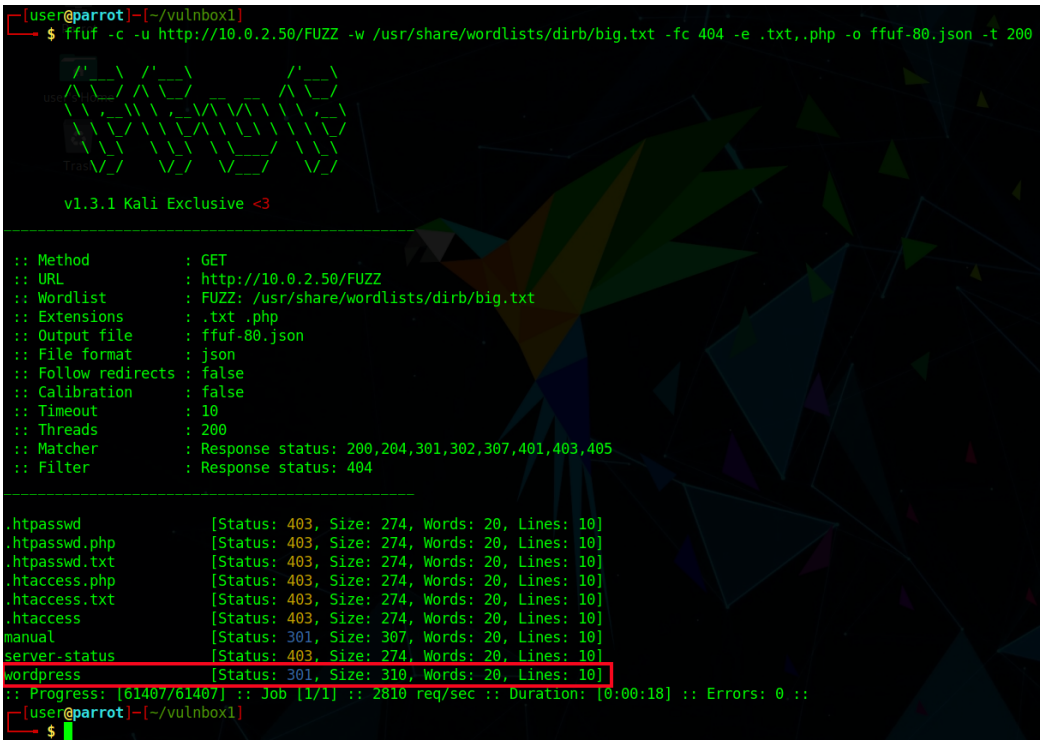


Figure 6: Fuzzing "hidden" resources with `ffuf`

The **wordpress** resource catches my eye immediately. We see its contents on a web browser:

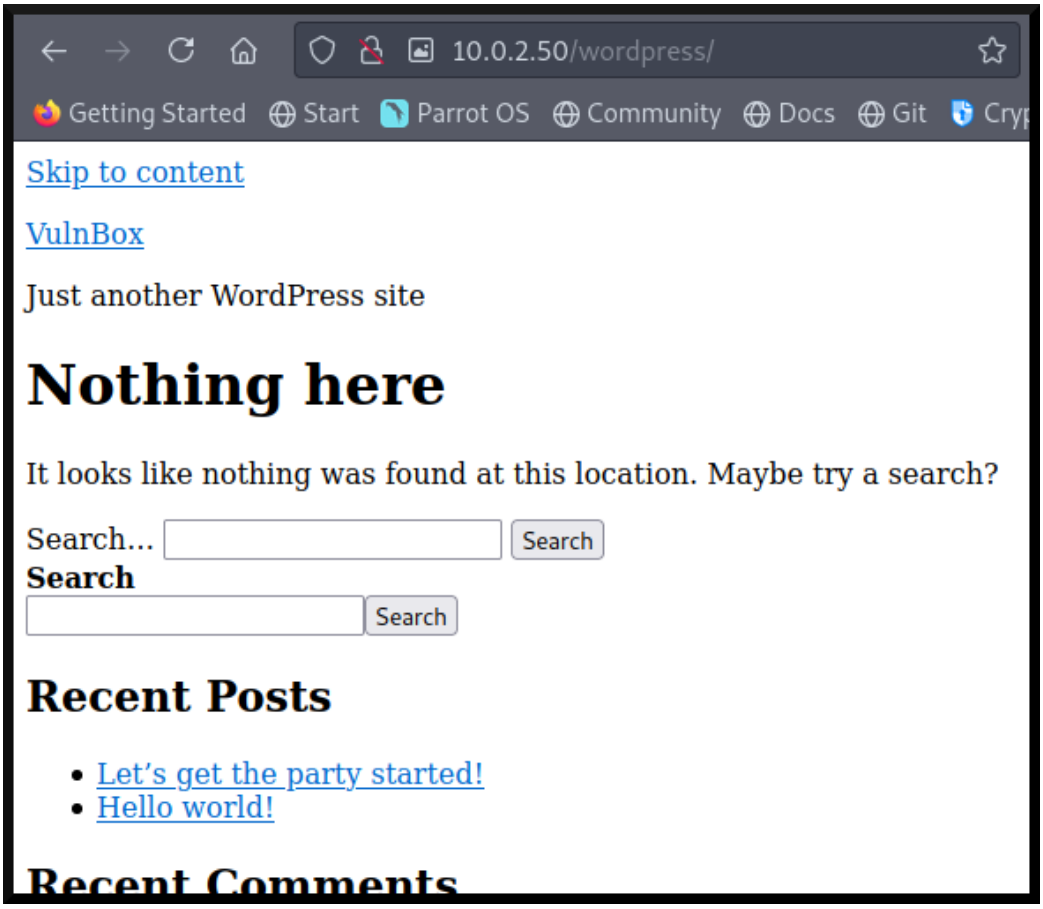


Figure 7: *WordPress* main page

It doesn't look right. It might be using *virtual hosting*, we can see this either by hovering on the links or by inspecting the source page:

Figure 8: *WordPress* main page **HTML** source code

It seems it makes requests to `vulnbox.ctf`, let's check its behaviour with a plain requests and another specifying the domain found in the `Host` header:

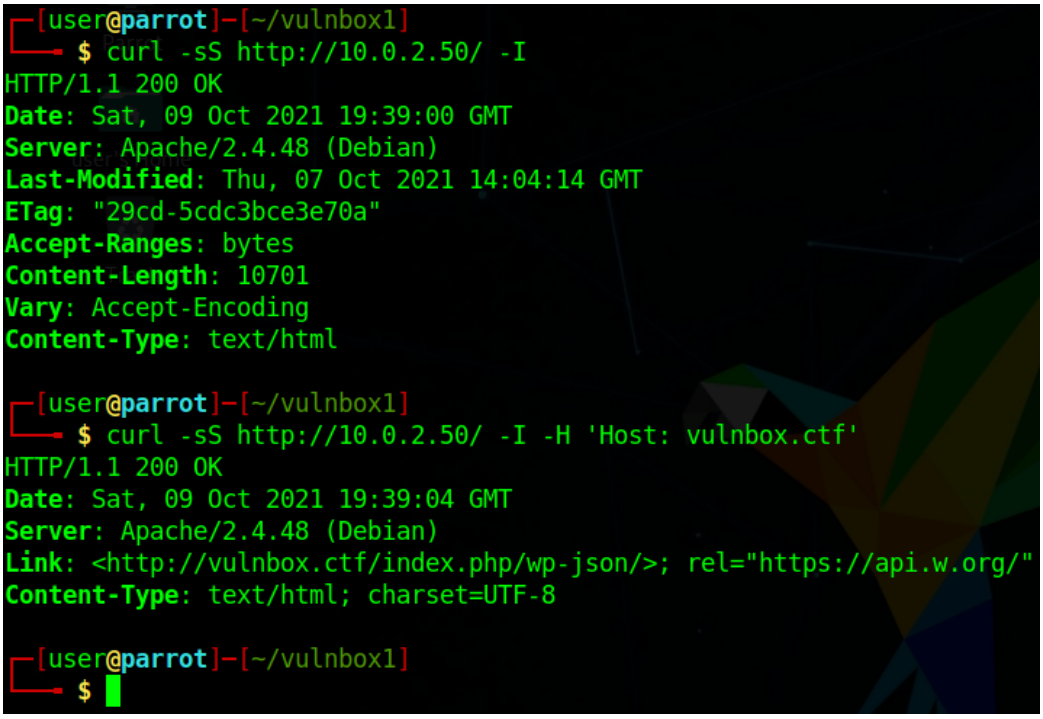


Figure 9: Difference between using the IP or the domain

The response is clearly different. This may mean *virtual hosting* is being applied. We should add it to `/etc/hosts` so the domain is resolved to our target IP address.

This way we can browse the web through our web browser seeing the contents intended to be shown.

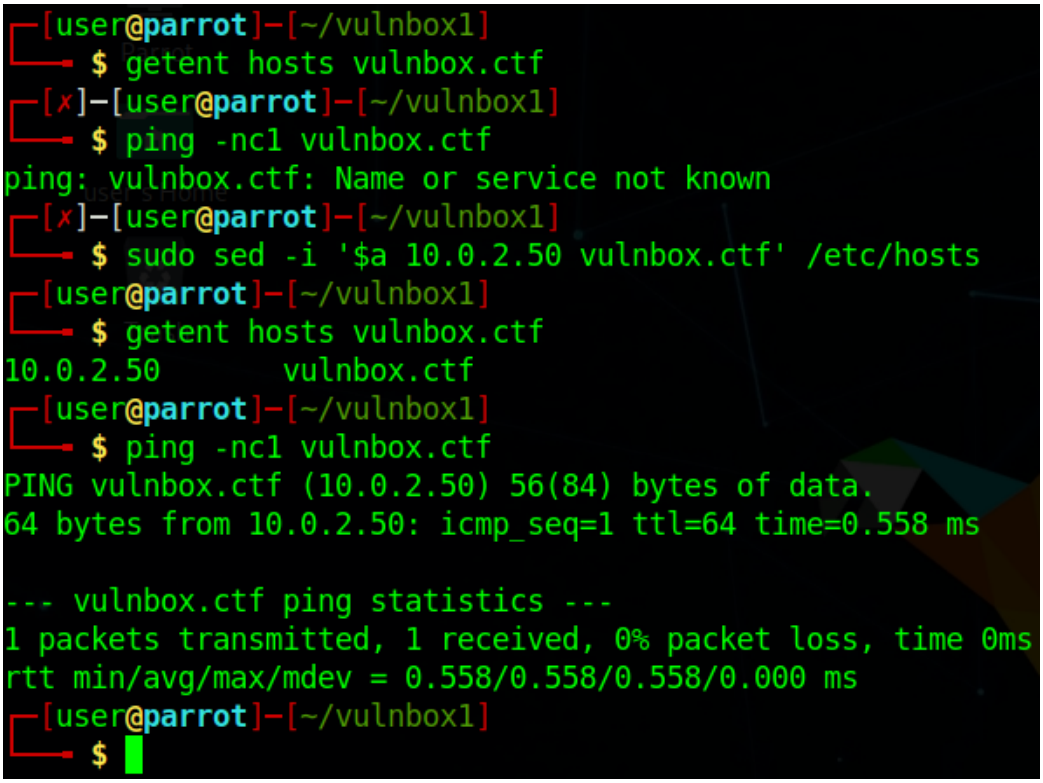


Figure 10: Adding the domain to `/etc/hosts`

Navigating to `vulnbox.ctf` on the web browser we see:

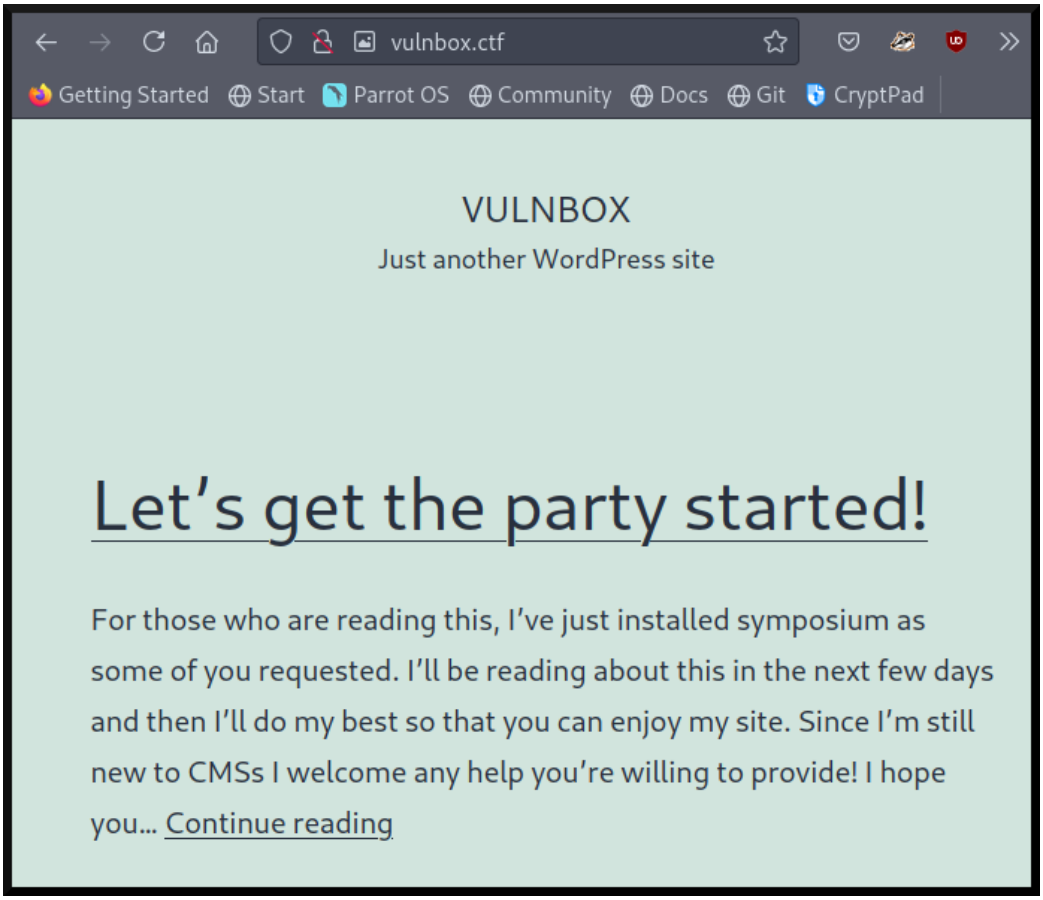


Figure 11: *WordPress* blog

Et voilà ! we see the page correctly.

Within this blog post, we can extract some crucial information. On the one hand, we have a username, namely, **albert**, who said he's installed something. That should make us think he's got some sort of privileges. On the other hand, he's saying what he's installed: **symposium**:

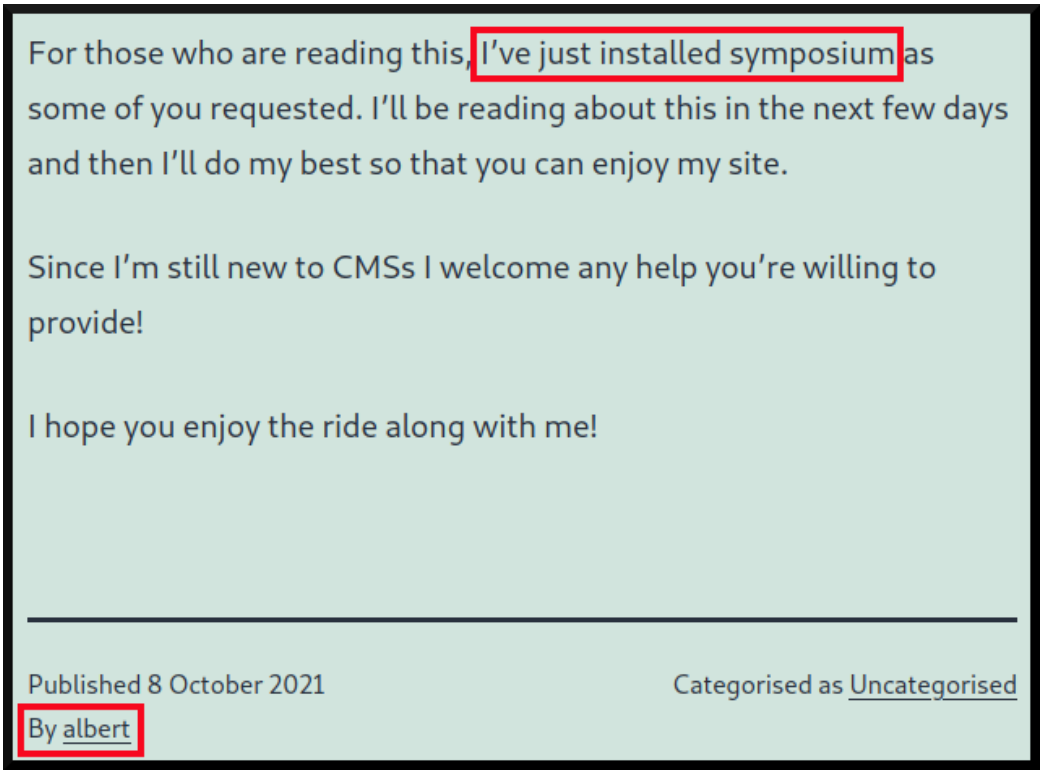


Figure 12: Information Gathering from the blog post

To confirm if the username found is valid, we should try to log in, the usual path for *WordPress* log in is in `{WORDPRESS}/wp-login.php`, if we go there we'll see objective panel.

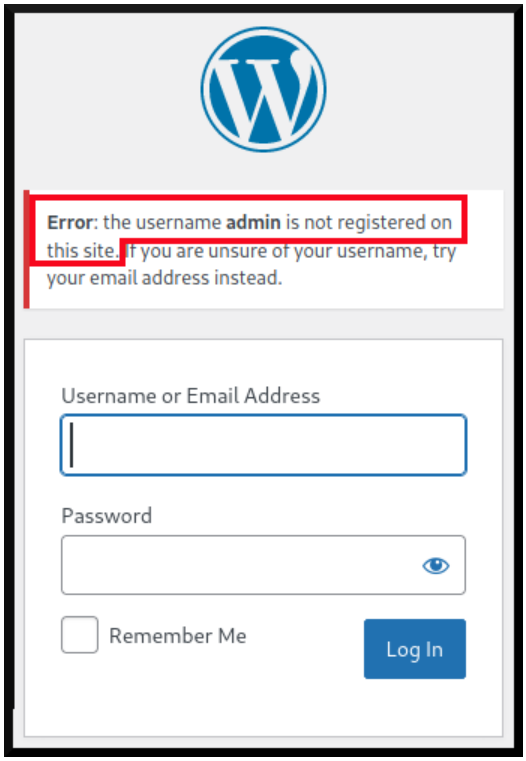


Figure 13: Checking the existence of the user **admin**

WordPress warns us that this user, **admin**, doesn't exist. Let's check the username found on the blog post:

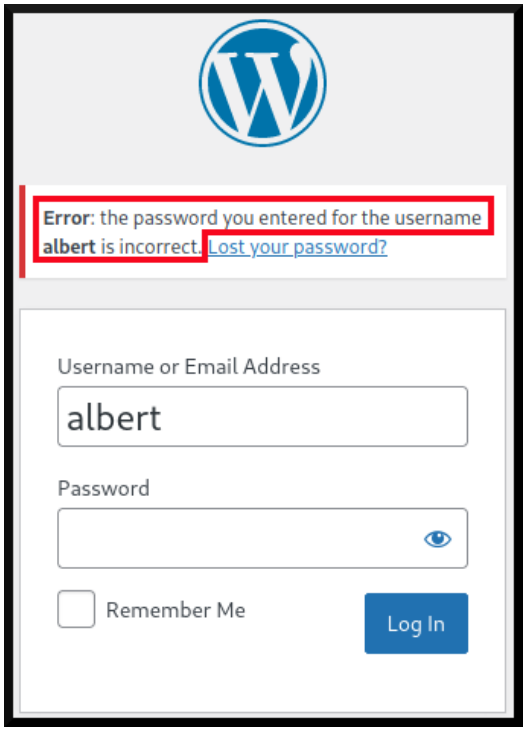


Figure 14: Checking the existence of the user **albert**

The message got is completely different from the previous one. This tells us the password is incorrect, but the user exists! However, we don't have a clue of what the password is, and as brute force is the last thing we should apply, let's look up on the Internet what **symposium** is.

wp-symposium is a *WordPress* plugin, let's see if it has known vulnerabilities:

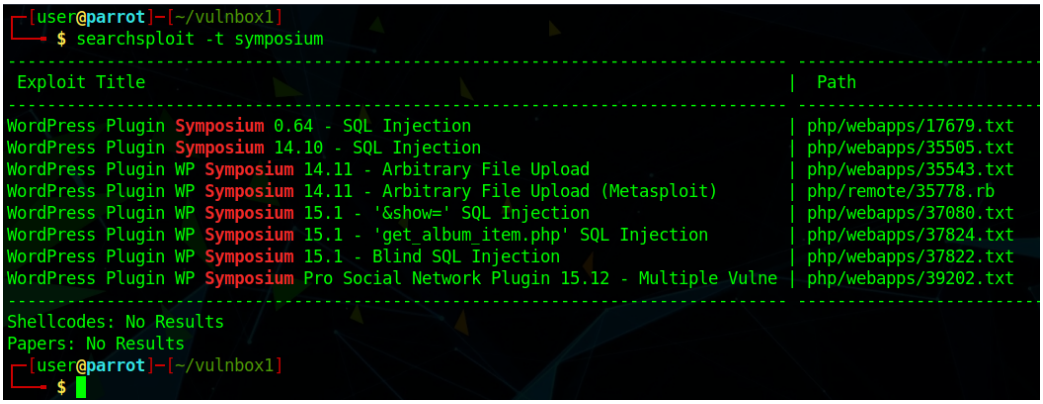


Figure 15: Looking for known **symposium** vulnerabilities

There are quite a few of vulnerabilities. We don't know how useful they are until we know the version. Looking for its source code on *GitHub*:

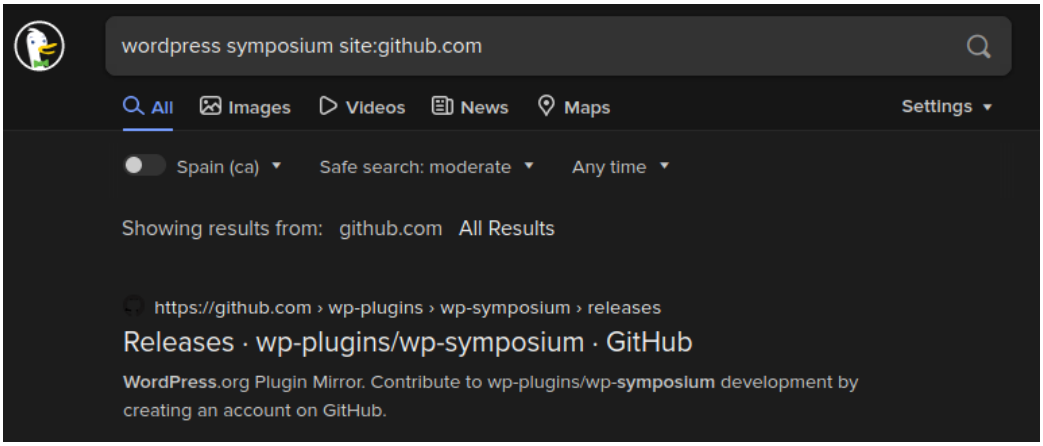


Figure 16: Looking up for **symposium** source code

I decided to download the latest version to see where I can get information about the version:

Figure 17: Downloading **wp-symposium** source code

We can download the source code and seek for patterns, like *version* or common files where the version might be, like *changelog* or *readme*, and so on:

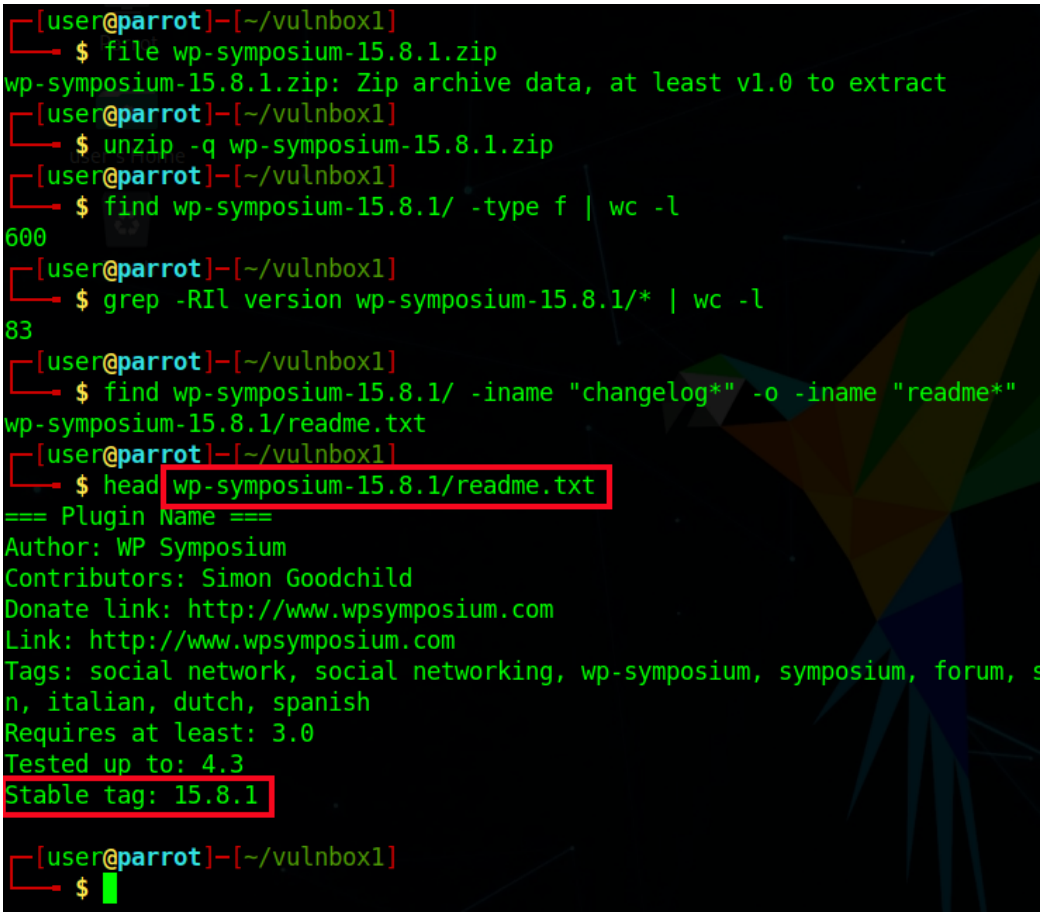


Figure 18: Looking for what file may contain the version

As we can see, there are 600 files, 83 of them have the word *version*. It's quite a lot. Searching only for some common files we can find **readme.txt** and within its first ten lines we see the version: *15.8.1*

We know that's the version because it's the one we've downloaded.

Now we know what file to aim in order to get the version and see whether or not is vulnerable.

WordPress stores its plugins on `{WORDPRESS}/wp-content/plugins/`, we should see if we can reach this path on the remote system:

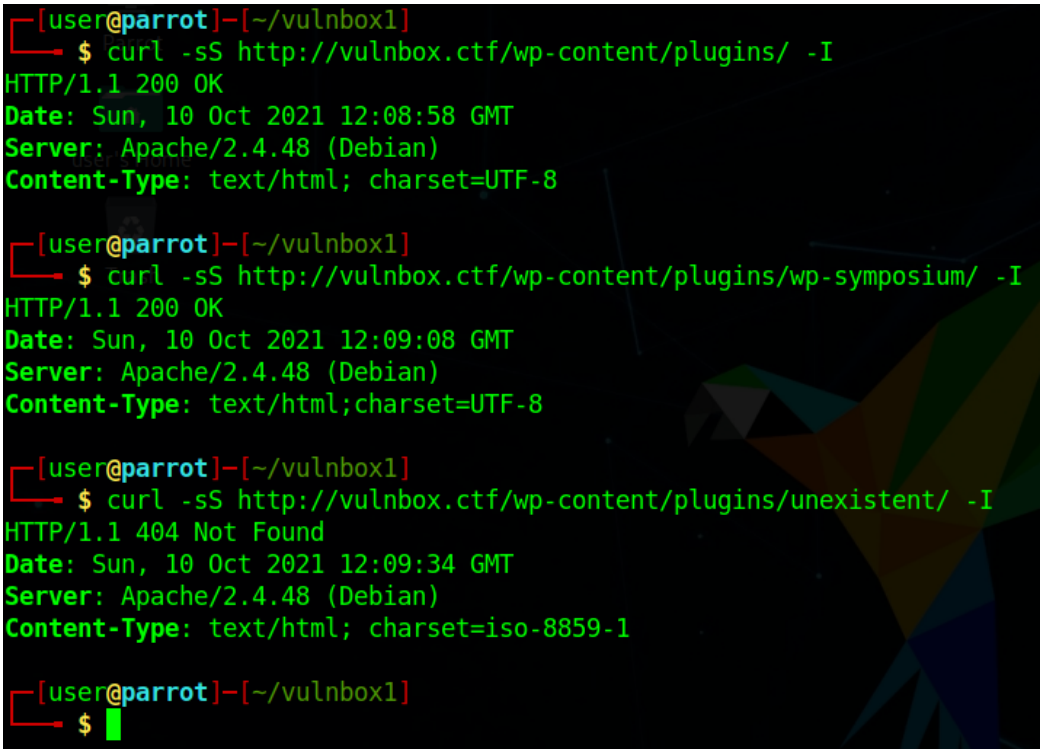


Figure 19: Checking for the existence of **wp-symposium** path

We see we can reach both the plugins path and the **wp-symposium** location[2], and if we try to get another path, like *unexistent* we get the infamous *404 Not Found* response.

Let's see if the **readme.txt** file is present:

[2]They both return a **200 OK** HTTP response code.

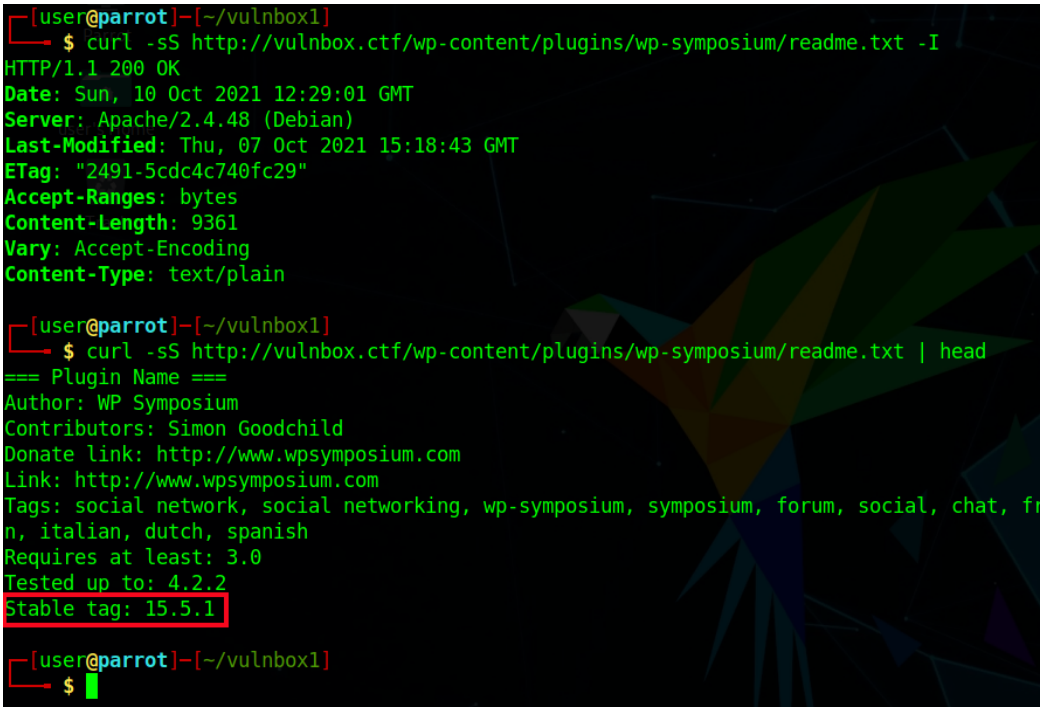


Figure 20: Discovering our target's **symposium** version

Not only does it exist, but it also gives us the current version installed, as expected.

Going back to the results of `searchsploit` we have at least three possible exploits to test. I'll try the second one. We should mirror it to get it into our current working directory:

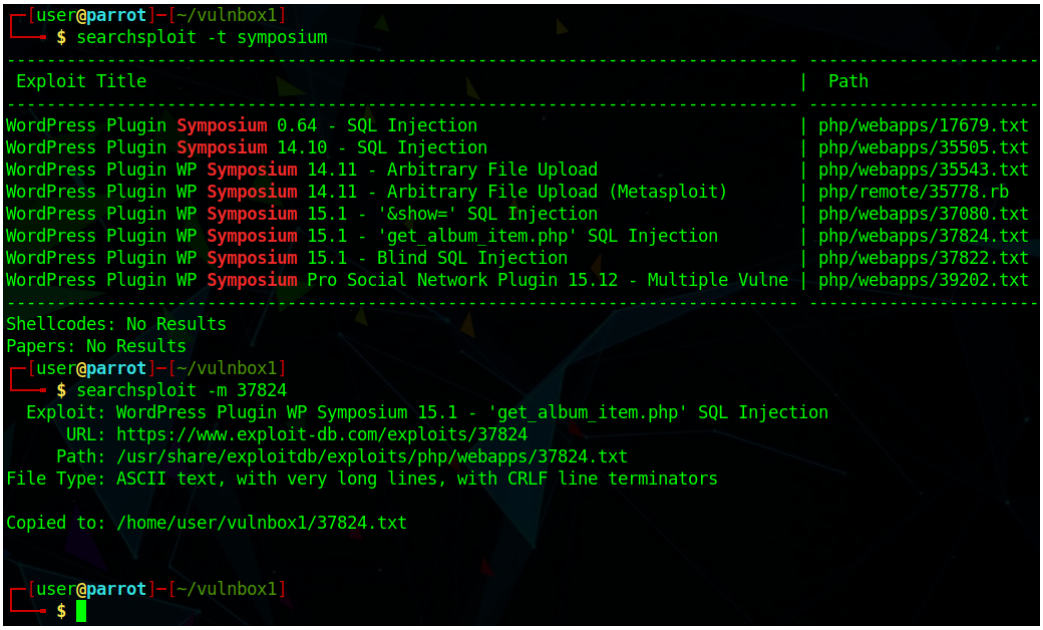


Figure 21: Mirroring the exploit

Read it, and get the part we need the most, the exploitation one:

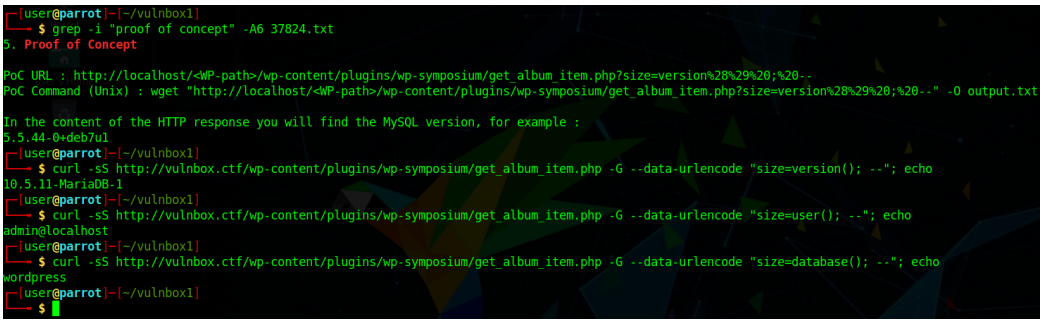


Figure 22: Testing the *PoC*

It works just fine!

Writing that query is quite time consuming, modifying it is a little cumbersome and error prone. It's better if we write our own exploit:

Figure 23: Exploit written in *Python*

We must test it:

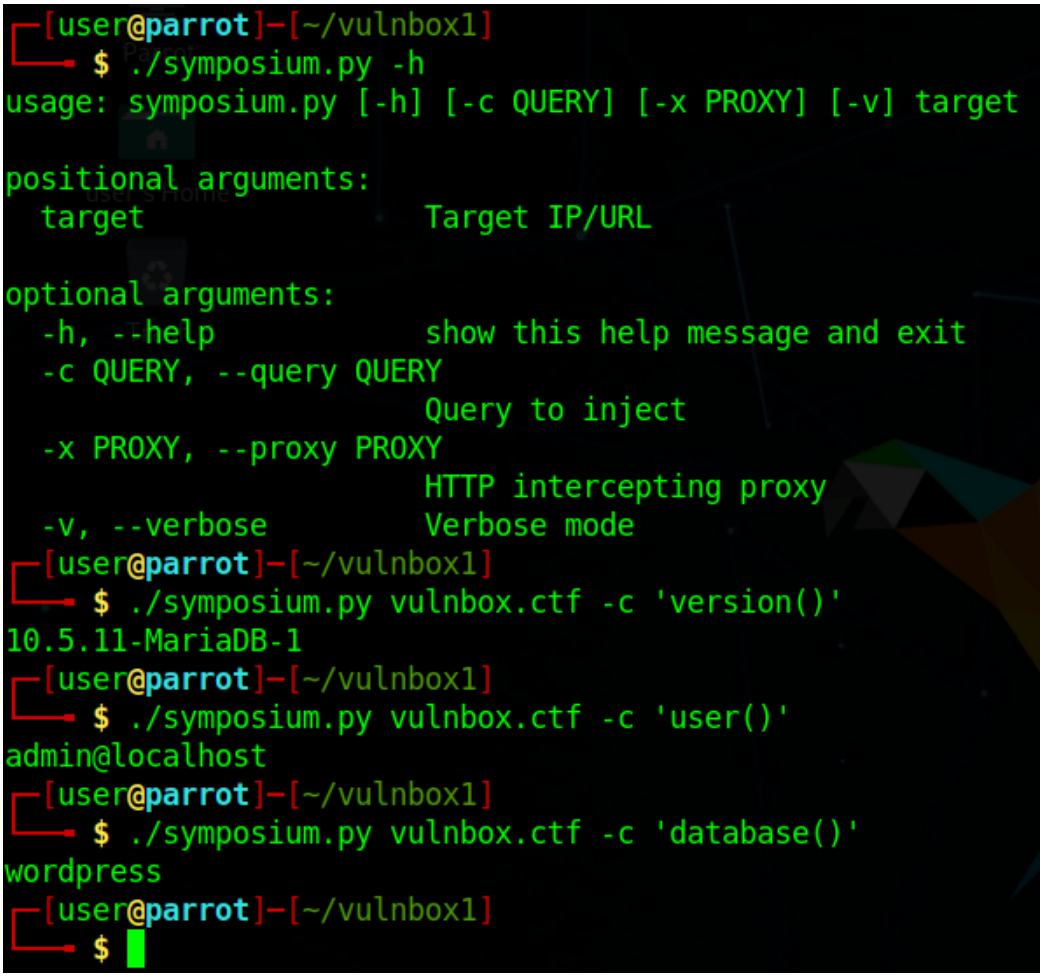


Figure 24: Testing the exploit

It seems to work right. Let's enumerate the databases:

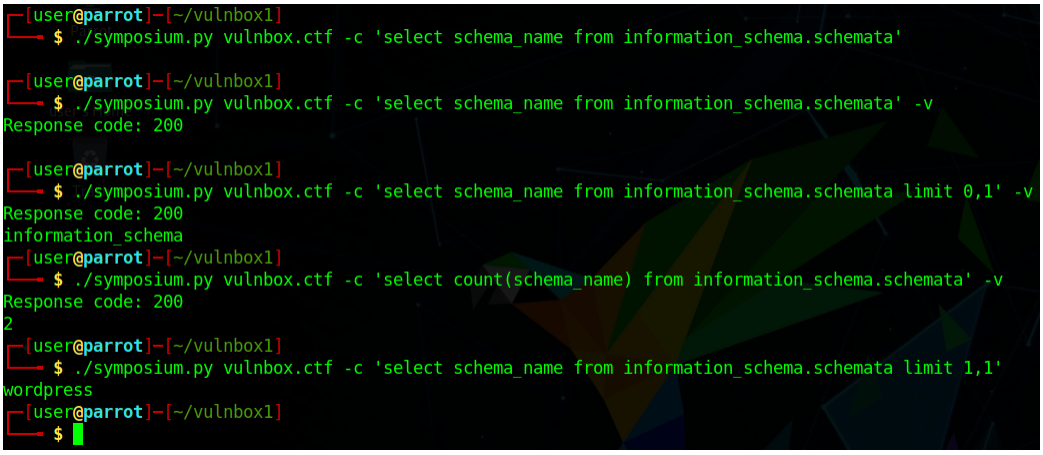


Figure 25: Enumerating databases

I'd like to break down the process I've made here:

1. I try to get the databases from **information_schema.schemata**.
2. As I don't get any output, I try to see the response code.
3. As the response code seems to have fetched an empty page, it might have happened because the query can't return more than one result at a time. So I limit the number of results to 1.
4. Since it worked, we are shown the first database from the output, I want to know how many of them there are by counting them.
5. There are only two databases, so I only try to get the next one: **wordpress**.

The important data is actually on the **wordpress** database, we're using **information_schema** to get information about the schemas and their structure.

Let's try to get the tables from our target database:

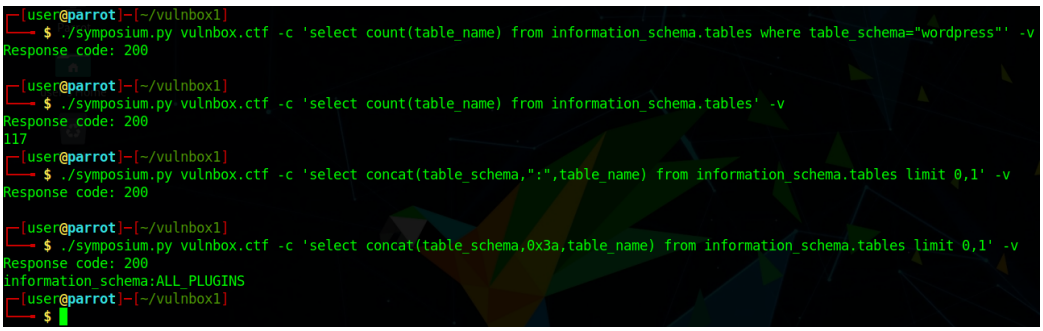


Figure 26: Enumerating tables: a first approach

1. I try to retrieve the tables in the **wordpress** database. But I get an empty response instead.

2. I remove the `where` clause to see if there's any problem with the filtering. And I receive the amount of tables on the whole DBMS.

3. As I won't able to tell which tables belong to which database, I decide to concatenate the database along with the table name. Separated by a colon. It didn't work though.

4. The `concat` function should exist, and as the filtering clause hadn't worked either, that makes me think there's a problem with the double quotes[3].
   I then use the colon's hexadecimal representation (`0x3a`) and it worked!

There are two approaches I can think of right now to enumerate the tables. The first of them is to do exactly what we have just done and filtering with `grep` to get solely the tables in the **wordpress** database, like this:

[3]The same happen if we try with single quotes.

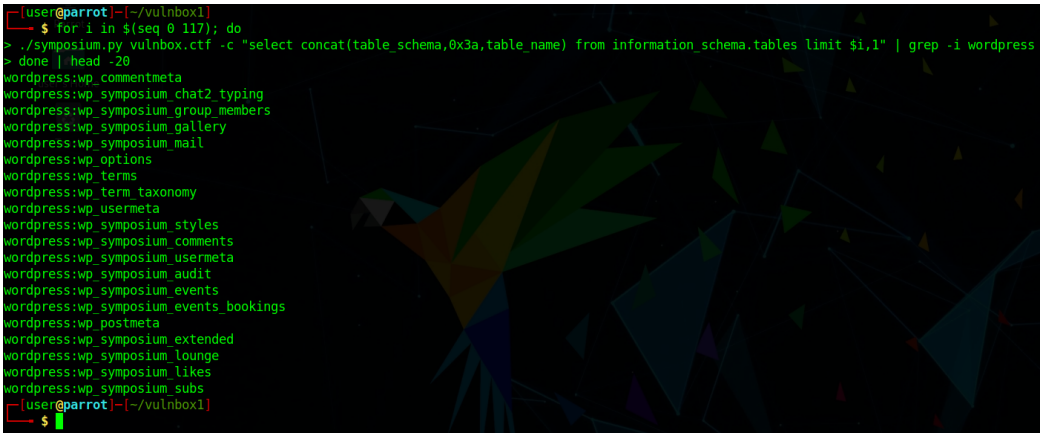


Figure 27: Enumerating tables: getting the tables the slow way

It works! But the output is kind of slow, as it has to make 118 requests[4] and filter the ones we're interested in.

The thing is, we know what the issue with the queries was: the quotes! And we know we can bypass it using hexadecimal, as we've proved with `0x3a`.

Let's try to do the same with the database name:

[4]from 0 to 117, this last response should be empty

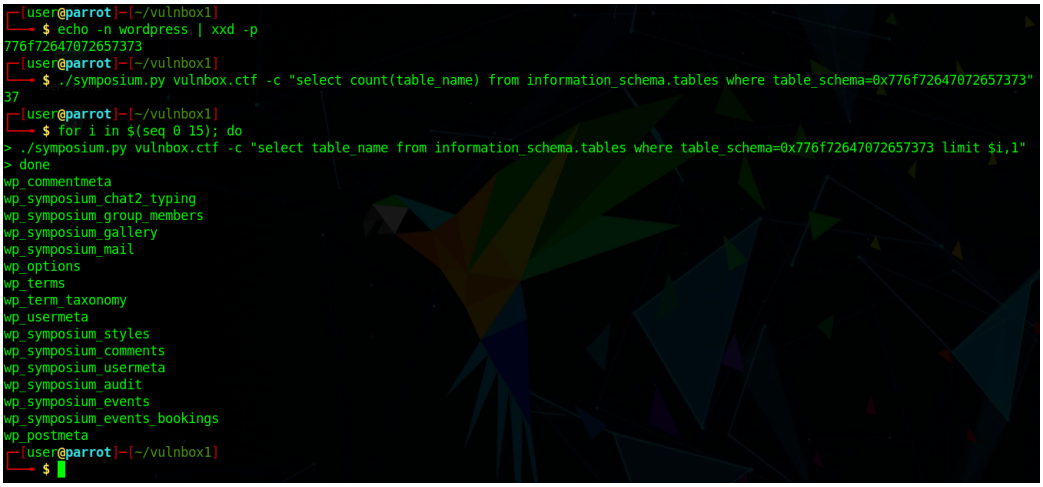


Figure 28: Enumerating tables: getting the tables, a better way

1. I see the hexadecimal representation of the word **wordpress**. Note I've removed the line feed with the option `-n`.
2. I tested to see if it worked and it did indeed!
3. We can now retrieve the table names in a more comfortable and quick way. Note I have requested for the first 16 ones, just to show how it worked and not to have a huge output.

If we performed the previous command with `seq 0 37` we would get all the table names we were aiming.

We're specially interested in credentials, therefore the table **wp_users**[5] catches our eye.

We can get the columns from **wp_users** filtering by both the database and table:

[5]This table isn't shown in the screenshot but we can get it when getting all the tables.

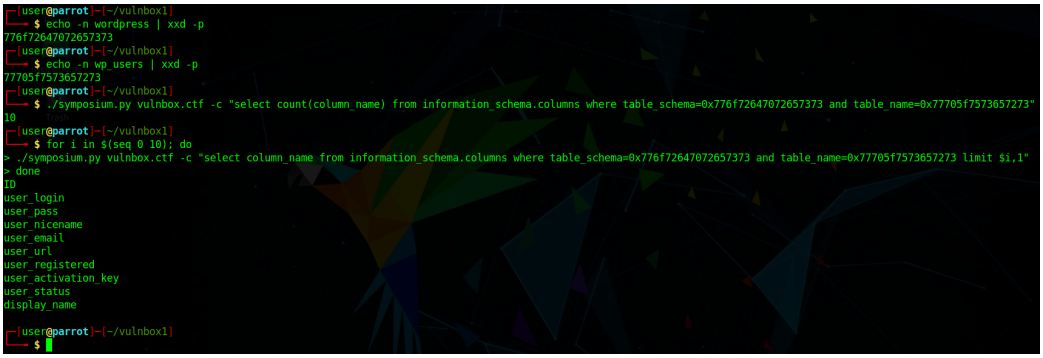


Figure 29: Enumerating columns from **wp_users**

Getting the table contents now is trivial. Let's get what we are interested in: **user_login** and **user_pass**

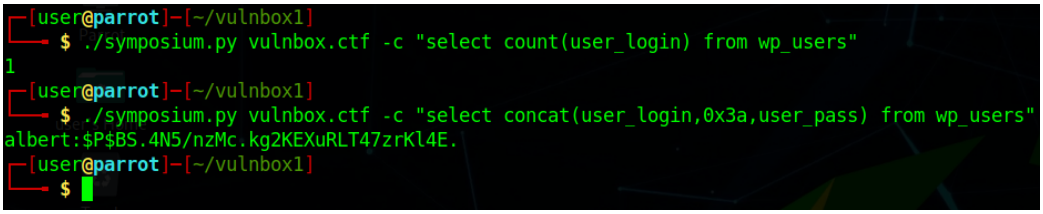


Figure 30: Getting credentials from **wp_users**

As the **user_pass** is likely to be hashed, let's try to break it with `john`:

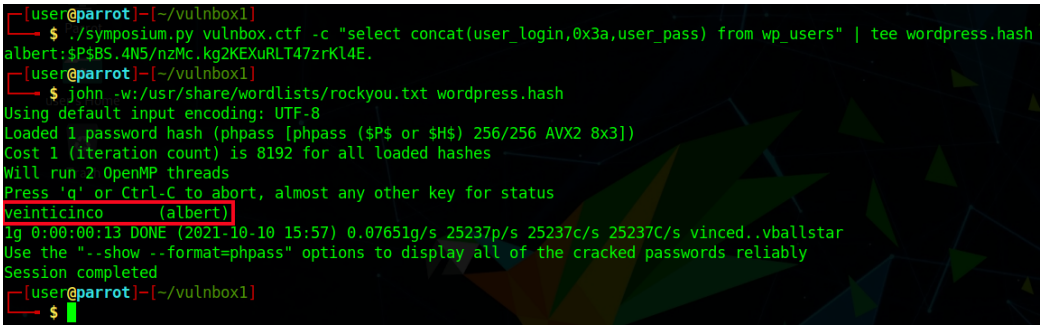


Figure 31: Cracking *WordPress* credentials

2.1.1 Gaining Access

With the credentials gathered:

`albert:veinticinco`

we can access the CMS as administrator:

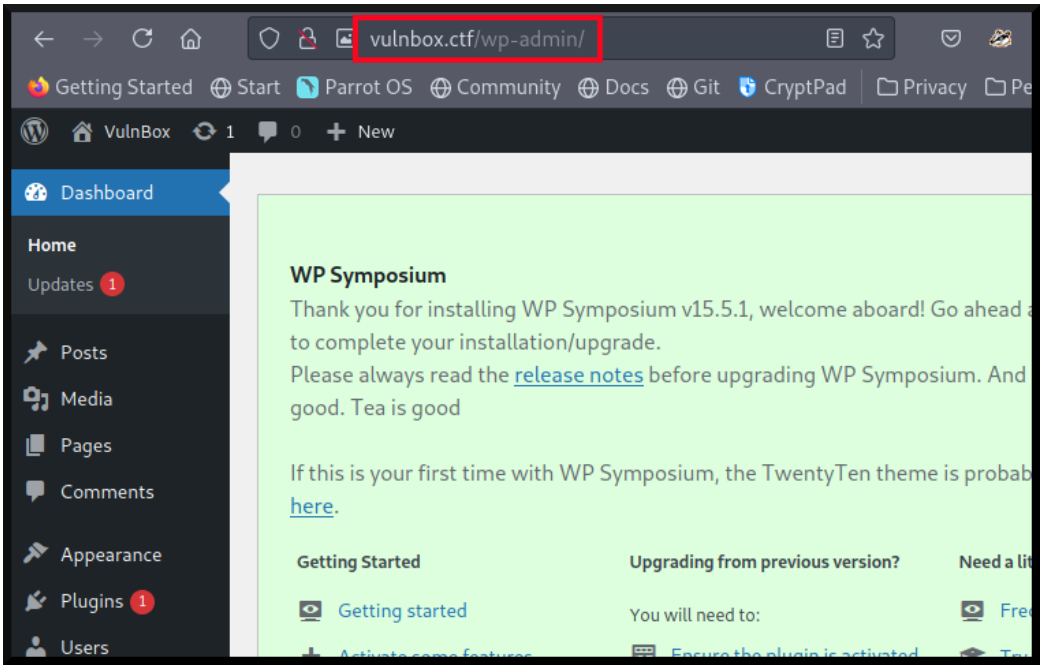
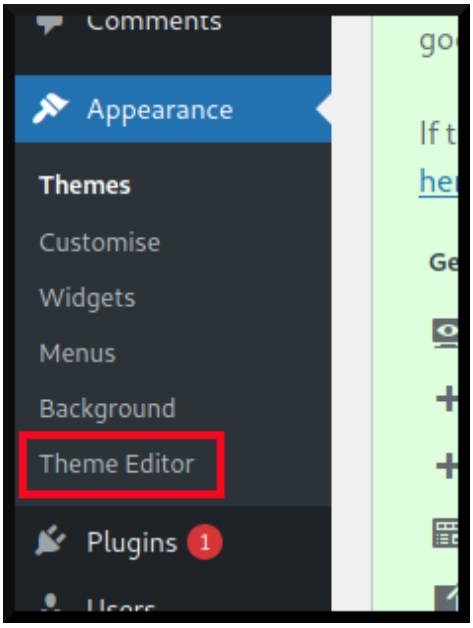


Figure 32: *WordPress* admin panel

To gain access to the system we can try the usual way: modifying the theme.

Let's go to **Appearance** → **Theme Editor**



The common template we use is the `404.php`, we have to take into account what theme we're modifying, in our case is the **twentytwentyone**, the active one[6], it's important to know this because we are going to run system commands through the *webshell* by specifying the path and a query string.

[6]There are three themes in this *WordPress*: **twentynineteen**, **twentytwenty** and **twentytwentyone**. All of them have similar structures, and we can use any of them regardless of whether or not is active.

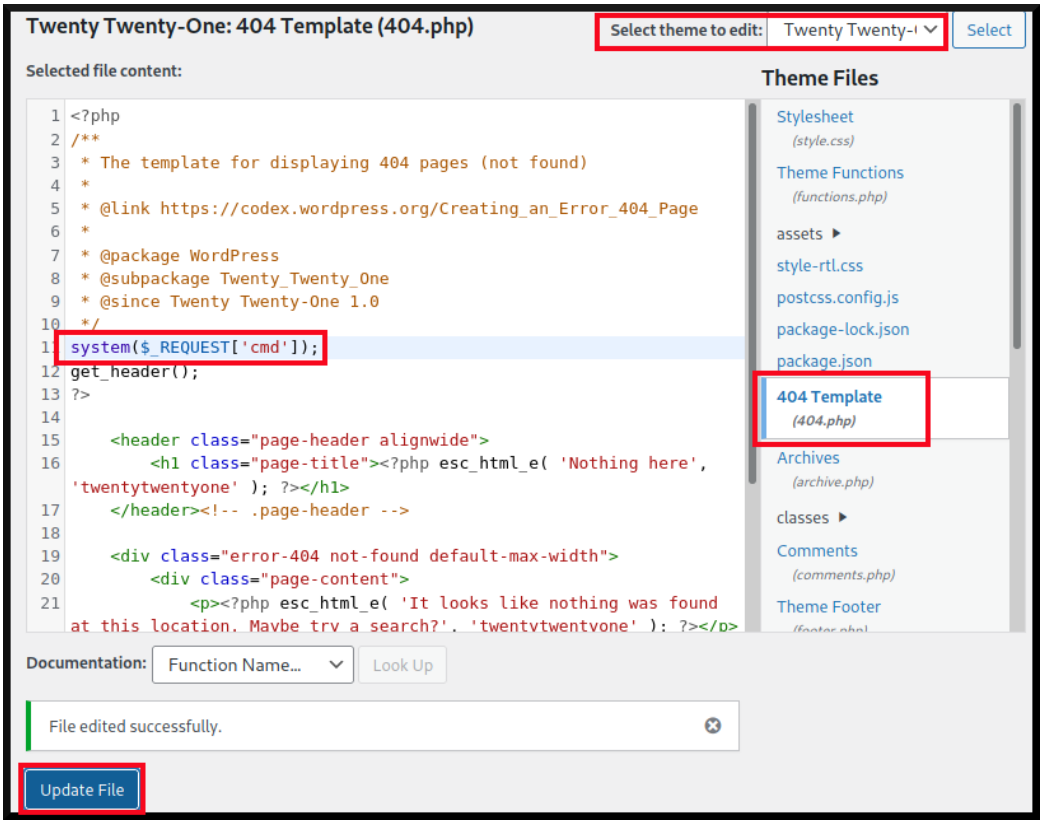


Figure 33: Setting a *Webshell*

Once updated successfully, we ought to test its functionality:

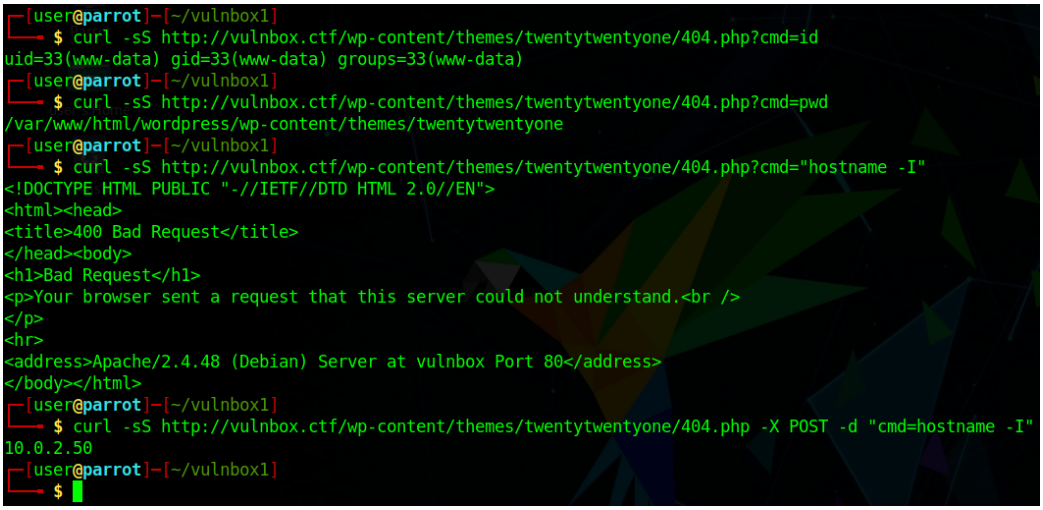


Figure 34: Testing our *webshell*

We've got **RCE**!

Note I'm able to run commands with both **GET** and **POST** requests, due to `$_REQUEST['cmd']`. If I had used `$_GET['cmd']` I would only be able to make **GET** requests, likewise with `$_POST['cmd']` I'd only be able to make **POST** requests.

The next step is to get a shell to get into the system.

There are a number of tools and techniques we can use to achieve this, the most well-known is `netcat`, even though, it's installed in the system we'll be leveraging the one it's install for sure: `bash`.

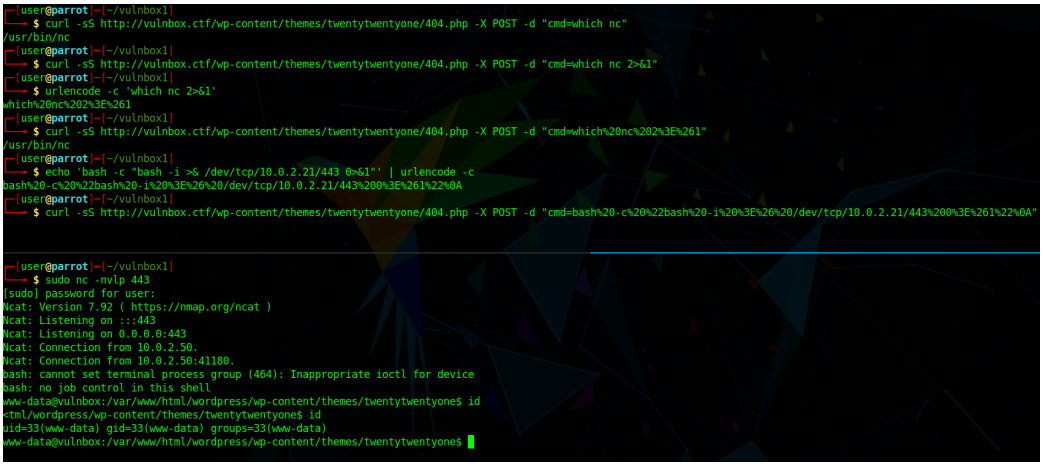


Figure 35: Getting a *reverse shell*

1. We check for the existence of `netcat`, and it's installed[7]
2. As we know we have a positive output from the previous command, we may try to do it again redirecting the **stderr** to **stdout**, this should make no difference, but it does. The reason is because the ampersand (`&`) is used to separate parameters, and this request raises an error and we receive no output at all.
3. To overcome this issue, we have to encode the parameter in **URL format**[8] You can also do this with
   `php -r "echo urlencode('which nc 2>&1');"`.
4. Once we know the parameters encoded work, we can try to establish a reverse shell.

As we can see, we've got a reverse shell, to work more comfortable, we should upgraded:

[7]Even though it's installed, the version may not support the `-e` or `-c` options, which run a command on the destination.

[8]The tool I'm using I created myself, you won't find it on your system. You can encode it in other ways, like searching a web page for this purpose, or use **python**, or **php** among others.

Figure 36: Upgrading the shell

The steps I make are:

1. `script /dev/null -c bash`
2. I stop the current session with **Ctrl**+**Z**.
3. `stty raw -echo; fg`
4. `reset`
5. The prompt asks for the type of shell, even when we specify it, it won't set that value. So we have to do it ourselves.
6. `export TERM=xterm`
7. `export SHELL=bash`
8. `stty rows 39 columns 191`[9]

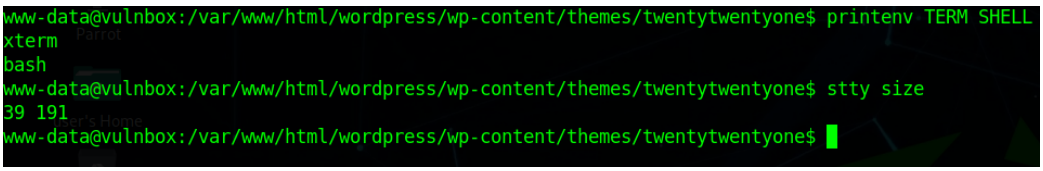


Figure 37: Shell upgraded

[9]This is the size of my console, but it may be different in your case, try `stty size` within another console to take the values from.

2.1.2 Lateral Movement: albert

Let's enumerate the users on the machine to know which ones could be our main targets, apart from **root**, of course:

```
www-data@vulnbox:/$ ls -lhFA /home
total 8.0K
drwxr-xr-x 3 albert  albert  4.0K Oct  8 16:39 albert/
drwx------ 3 jessica jessica 4.0K Oct  7 21:29 jessica/
www-data@vulnbox:/$ awk -F: '/sh$/{print $1}' /etc/passwd | xargs -n1 id
uid=0(root) gid=0(root) groups=0(root)
uid=1000(albert) gid=1000(albert) groups=1000(albert),24(cdrom),25(floppy),29(audio),30(dip),44(video),46(plugdev),109(netdev)
uid=1001(jessica) gid=1001(jessica) groups=1001(jessica),1002(reader)
www-data@vulnbox:/$
```

Figure 38: Enumerating system users

With this information the user **albert** seems to be our main target, as he's on more groups and **jessica** doesn't seem to be in an important group.

On the other hand, **jessica**'s permissions to her home directory are more restrictive and it would be harder to get more information than with **albert**.

```
www-data@vulnbox:/$ cd ~albert && ls -lhFA
total 40K
-rw-r--r--  1 albert root      0 Oct  7 22:42 .bash_history
-rw-r--r--  1 albert albert  220 Oct  7 15:05 .bash_logout
-rw-r--r--  1 albert albert 3.5K Oct  7 15:05 .bashrc
-rw-r--r--  1 albert albert   50 Oct  7 16:32 .gitconfig
-rw-r--r--  1 albert root      0 Oct  7 22:42 .mysql_history
-rw-r--r--  1 albert albert  807 Oct  7 15:05 .profile
-rw-------  1 albert albert 9.4K Oct  7 23:05 .viminfo
drwxr-xr-x  3 albert albert 4.0K Oct  7 17:15 devel/
-r--r-----  1 albert albert   33 Oct  7 15:09 flag1.txt
----r-----+ 1 albert albert   33 Oct  7 15:10 flag2.txt
www-data@vulnbox:/home/albert$ cat flag1.txt
cat: flag1.txt: Permission denied
www-data@vulnbox:/home/albert$ cat flag2.txt
cat: flag2.txt: Permission denied
www-data@vulnbox:/home/albert$
```

Figure 39: **albert**'s home directory's contents

We can't read neither `flag1.txt` nor `flag2.txt`. Both `.bash_history` and `.mysql_history` are empty. Let's see what's in `devel`:

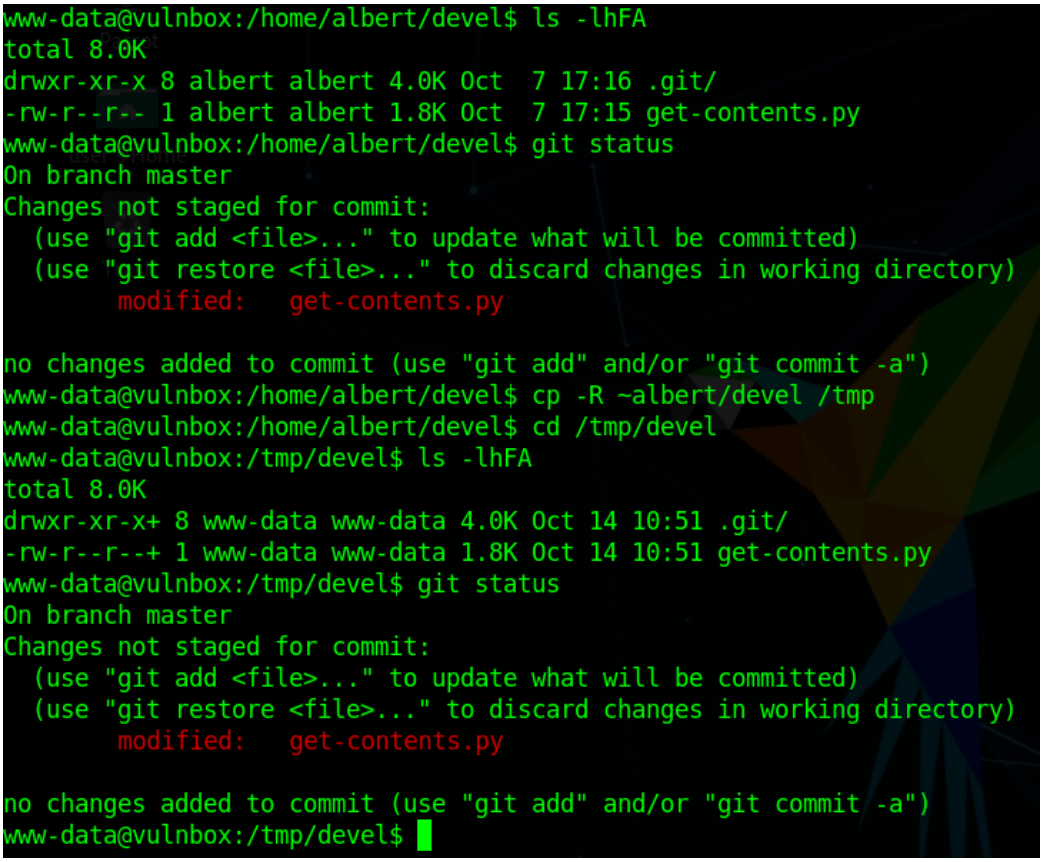


Figure 40: `devel` folder contents

It's a project versioned with `git`. If we run `git status` we can see there are changes from the previous commit. There's nothing interesting in the python script we can see.

As we don't have write permissions we won't be able to go to previous commits. So, we should copy the whole project to where we can do it, like `/tmp`, and make sure we've copied everything exactly as it is.

To see what are the differences between the current script and the previous version, we can issue `git diff get-contents.py`. But, once again, we don't get anything interesting. Maybe there were other files removed or more branches.

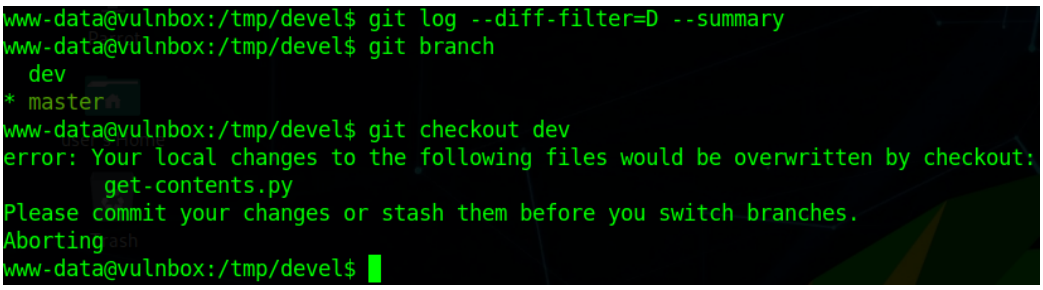


Figure 41: Looking for deleted items and more branches

Filtering by deleted files/folders, we don't get anything. Checking for the existence of more branches, we find out the **dev** branch. But we can't switch to it because our current branch, **master**, has uncommitted changes.

With `git status` we could also see there were files which weren't added for the commit, so we should add them and then make the commit:

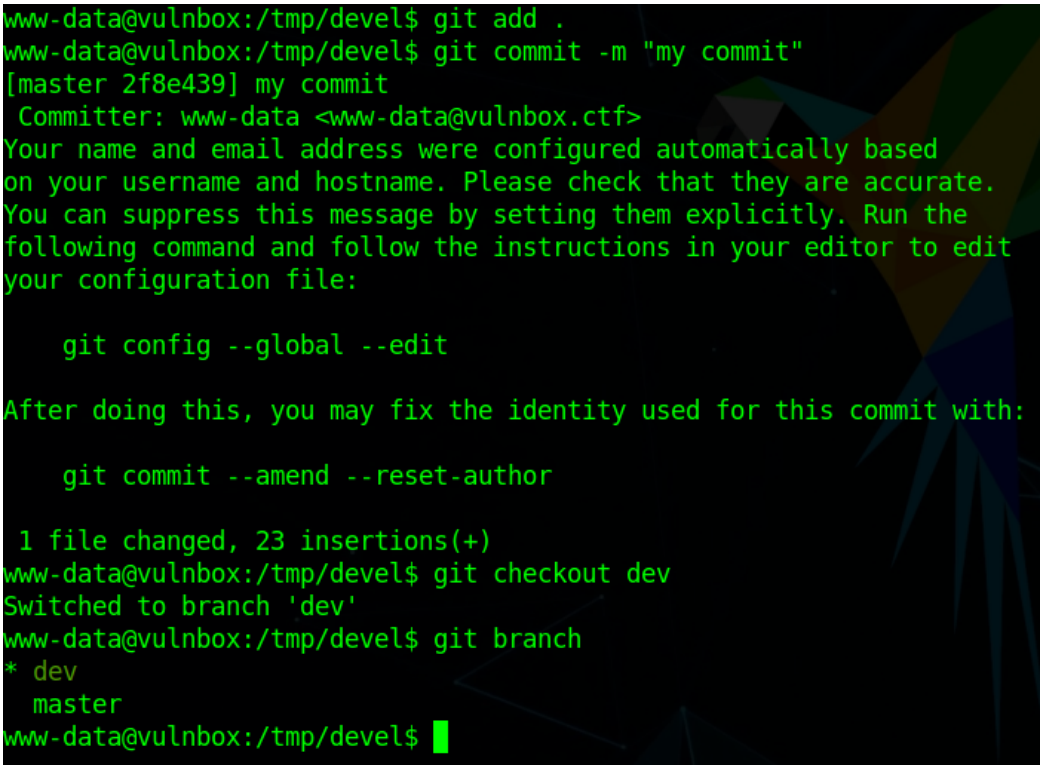


Figure 42: Committing and moving to **dev** branch

And we're in **dev** branch now. If we see the contents, there's a new file, namely `jessica.txt`:

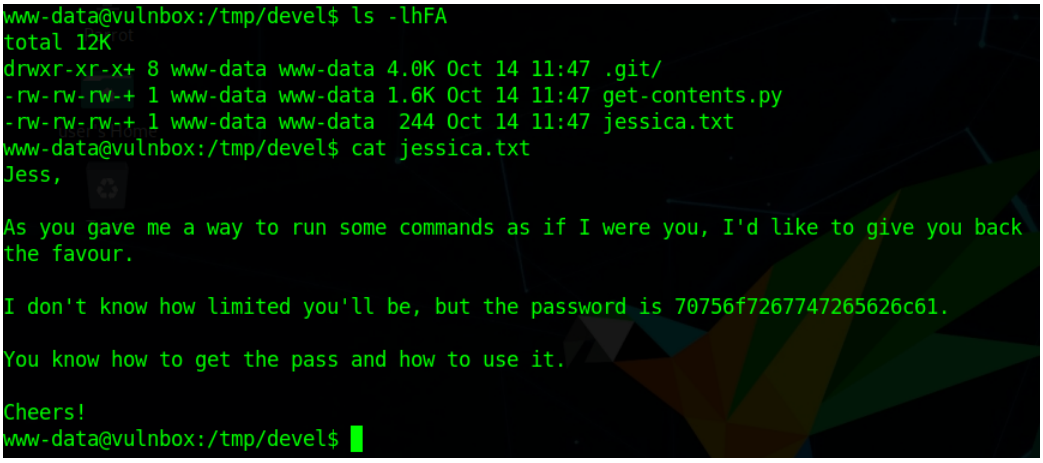


Figure 43: Reading `jessica.txt`

As we can read, we have a password in hexadecimal which **albert** is providing to **jessica**, then we try to log in with this to **albert**, but it fails. Although it makes little sense, we have to try with every single user (*password spraying*). We don't succeed.

If we reverse the password found, we see a hint: *albertgroup*:

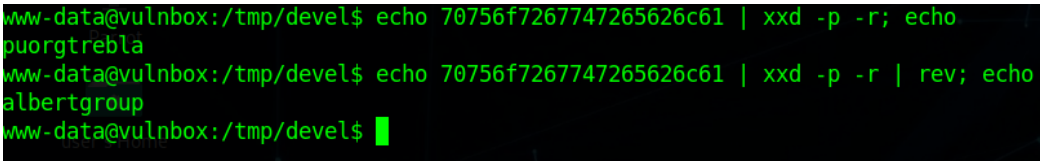


Figure 44: Revealing the password and the hint

We may already know we can start a shell with another user, as long as we know the password, with `su - <user>`, but we can do the same with a group! It has to have a password assigned though:

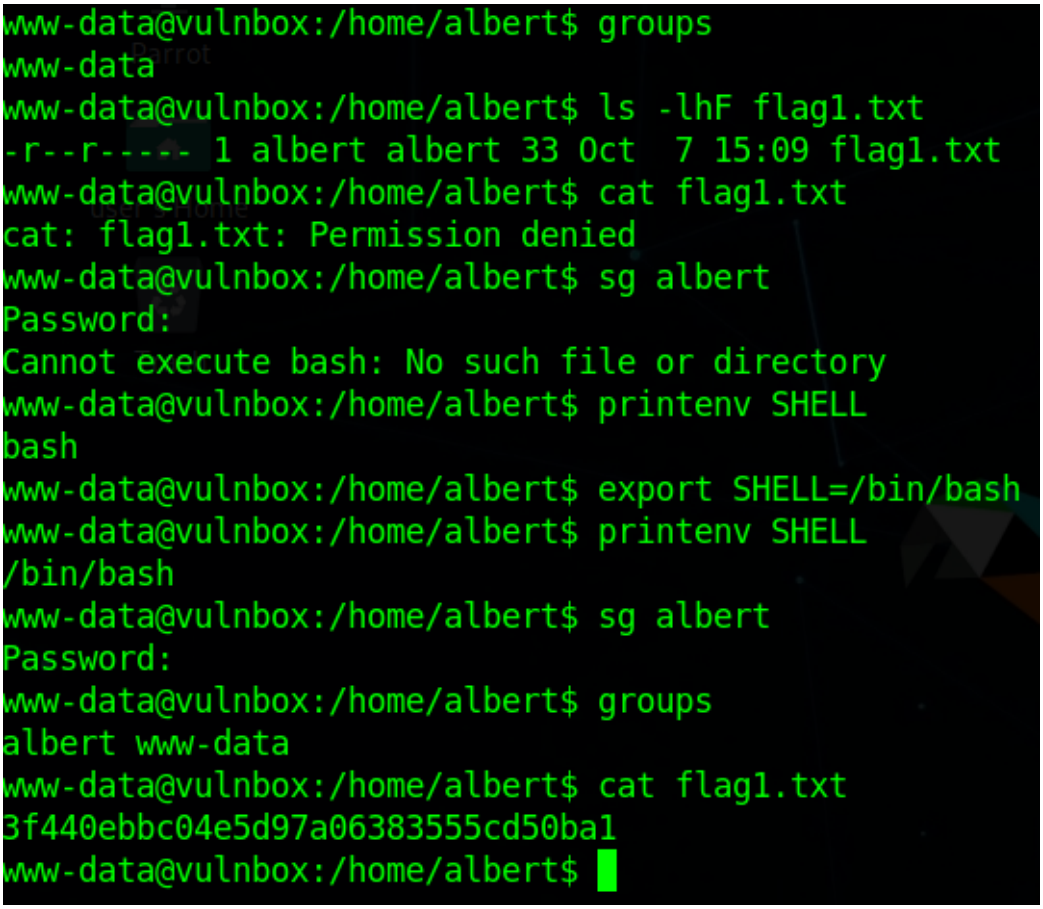


Figure 45: Changing groups and reading `flag1.txt`

- With `groups` we list the groups we belong to.
- With `sg <group>` we can add ourselves to another group.
- `printenv <variable>` prints exported environmental variables.

We had a problem trying to get into **albert** group because it didn't find the *shell* specified. It needed the full path to it, once provided we got into our target group and could read the first flag.

Nevertheless, we can't read `flag2.txt`, even though we are in the group **albert**:

Figure 46: We can't read or change `flag2.txt` permissions

Neither can we change its permissions.

Inspecting the file further, we were able to see a plus sign within the permissions. This indicates it has **ACL**s[10], let's see them:

[10] **ACL**: *Access Control Lists*

Figure 47: Listing the **ACL**s of `flag2.txt`

It seems neither **albert** is able to read the file, unless he changes the permissions as he's the owner, but **jessica** can.

2.1.3 Lateral Movement: jessica

Enumerating schedule tasks (**cronjobs**), we have an interesting finding:

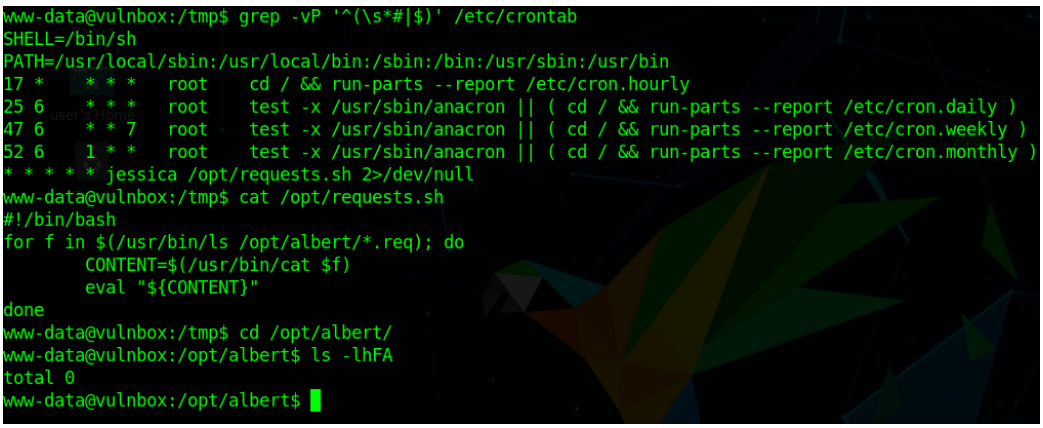


Figure 48: **cronjobs** enumerating with finding

The last line of `/etc/crontab` tells us the script `/opt/requests.sh` is being run by **jessica** every minute.

The contents of the previous script shows how any file within `/opt/albert` with the extension `.req` is being executed.

And this folder is empty. As we don't know what is inside **jessica**'s home folder, we can start by enumerating its contents, or starting with something easier, reading `flag2.txt`:

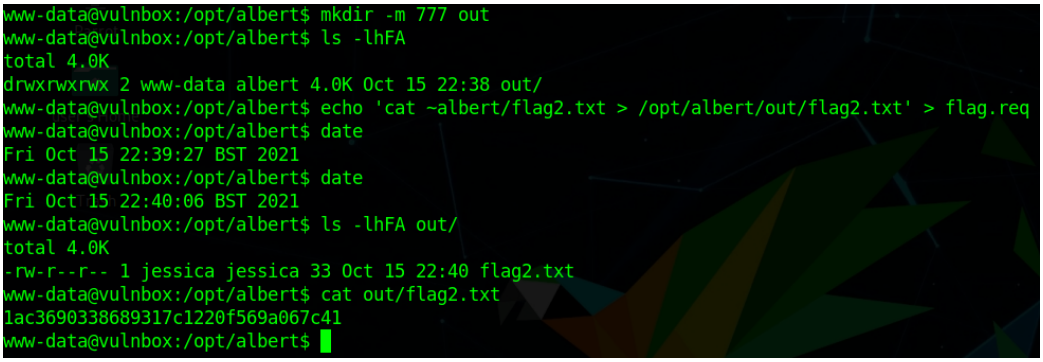


Figure 49: `flag2.txt`

And we were able to read `flag2.txt`. Now, let's try to take advantage of this situation to get into **jessica**'s account.

To achieve this, if possible, we should enumerate what she has or can do. As her permissions to her home directory were very restrictive, we may want to start to see what's in there:

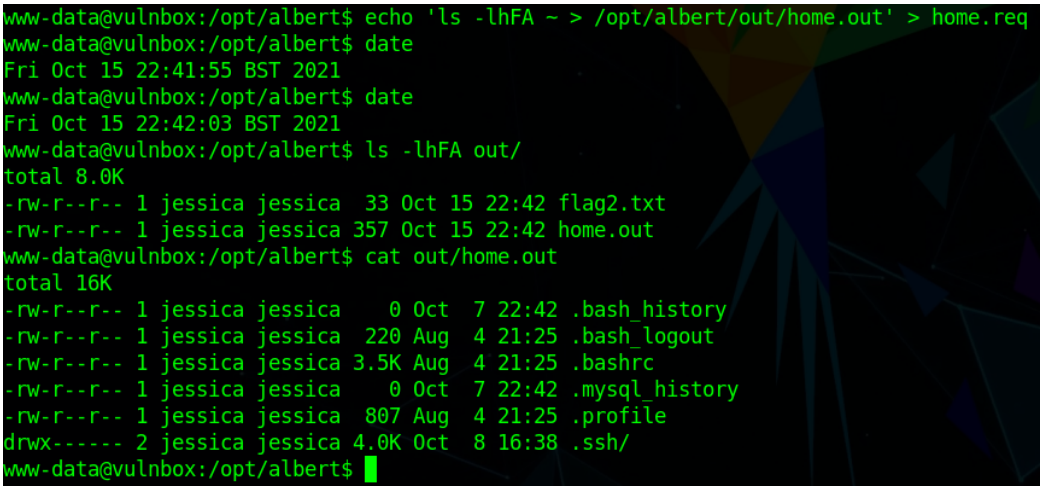


Figure 50: Listing **jessica**'s home contents

There's a `.ssh` directory, which might have a private key to her own account. If we list the contents:

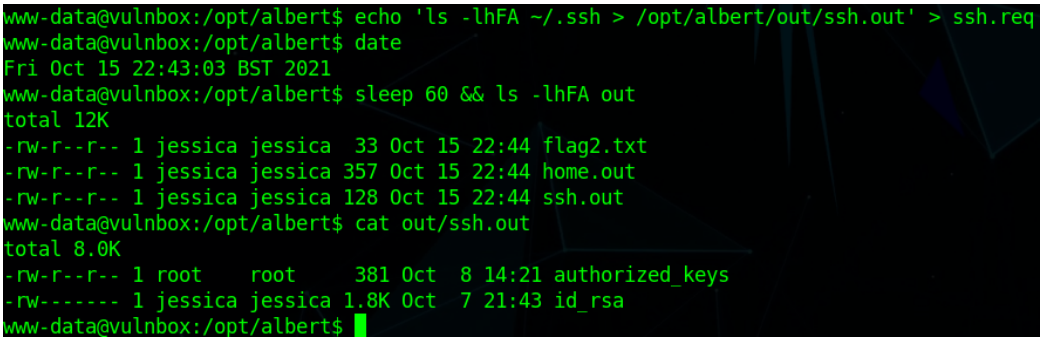


Figure 51: Listing **jessica**'s `.ssh` folder

There's a `id_rsa` file, which is used to store private **SSH** keys. Let's get its contents:

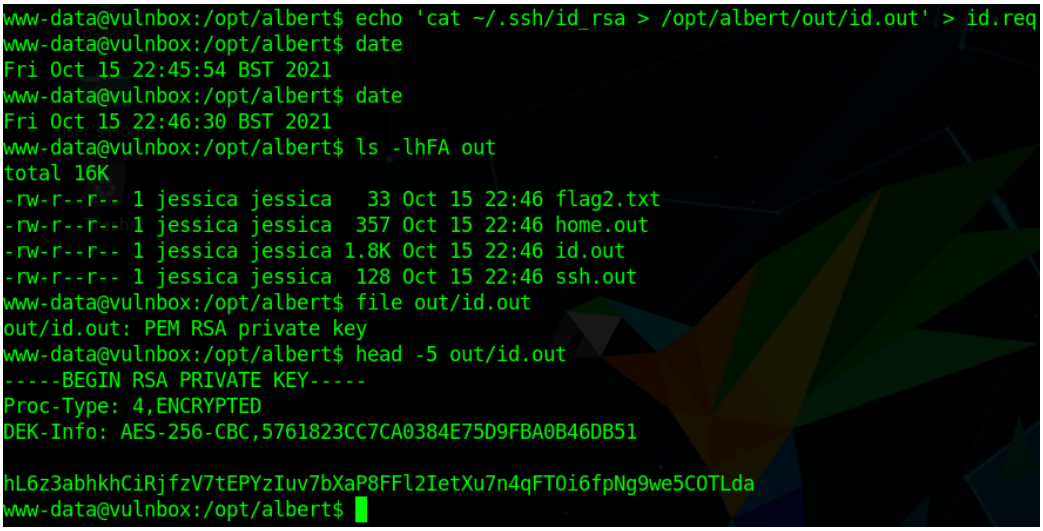


Figure 52: Getting **jessica**'s `id_rsa` private key

Since **jessica** is the owner of the files we're creating through her **cronjob**, we cannot modify the file, and we need to change the permissions to be readable exclusively by the owner to use it. However, we can copy the file, as we can read it, and then we're going to be the owner of the copy:

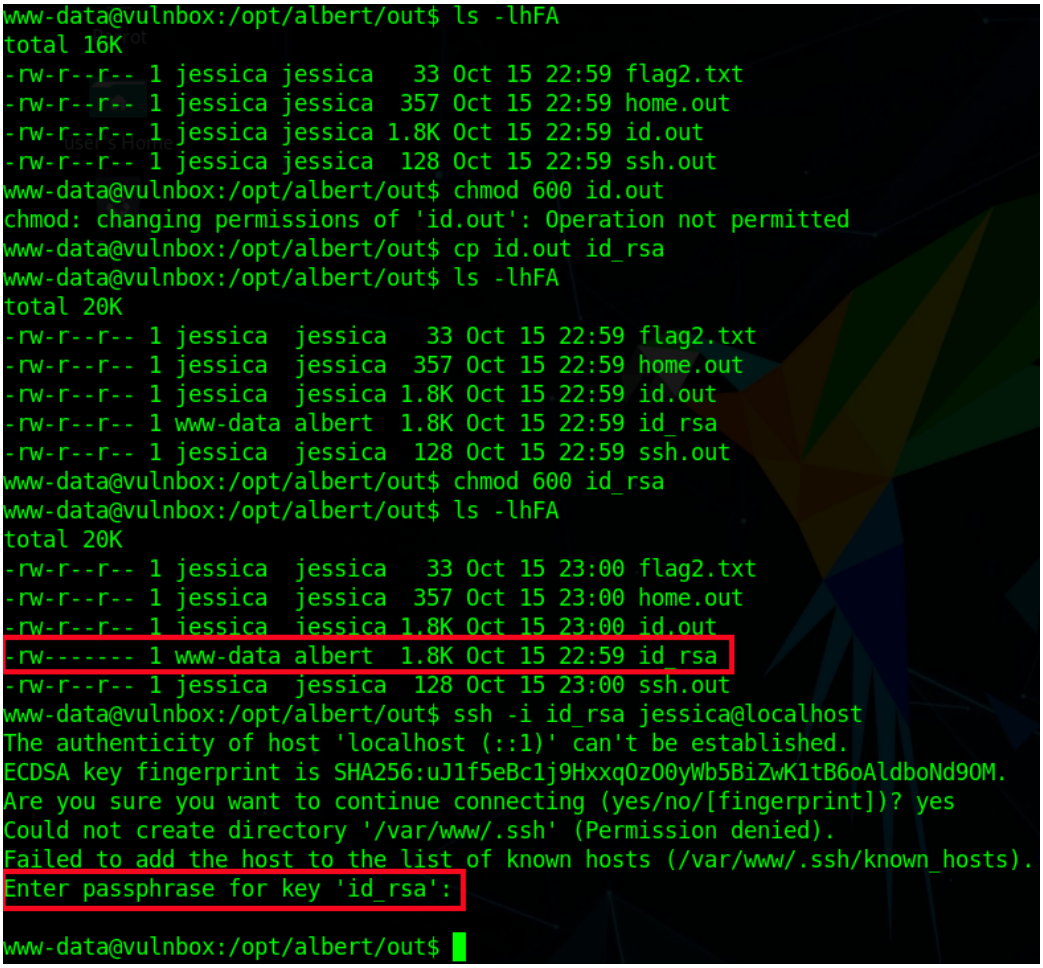


Figure 53: Trying to log in with `id_rsa`

At the end of the previous screenshot, I tried to log in with the private key, but it asked me for a passphrase[11], so I cancelled it.

In this case, we should try to crack it, a well-known option to achieve this is by using `ssh2john` which will give us a hash to crack with the `john` utility. As those utilities won't probably be on the victim's, we should transfer to our attacking machine and crack it there:

[11] We were able to tell so because in the second line of `id_rsa` there was the word `ENCRYPTED`.

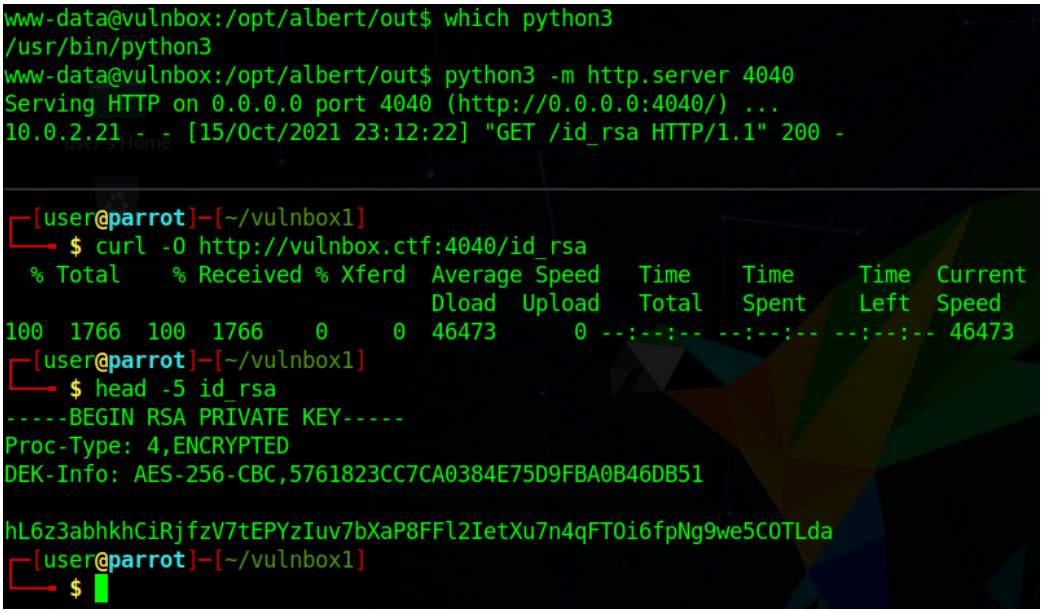


Figure 54: Transferring `id_rsa`

Figure 55: Getting a `john`-friendly hash from the private key

And we finally crack it:

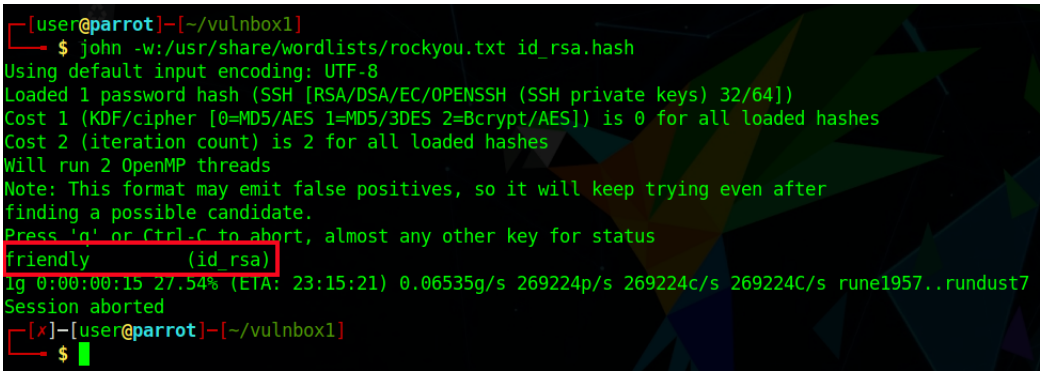


Figure 56: Cracking the password-protected `id_rsa`

Now, we can log into the machine as **jessica** through her private key:

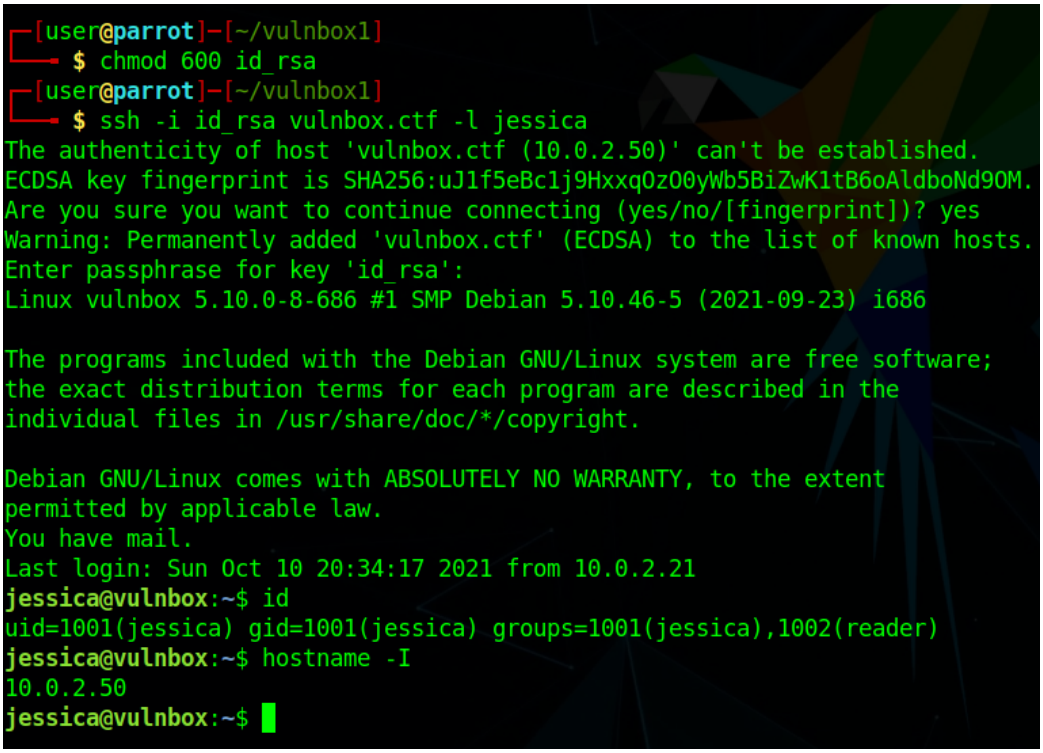


Figure 57: Logging as **jessica** through **SSH**

2.1.4 Privilege Escalation

Note: we're in a restricted bash shell (`rbash`), which is a little uncomfortable, as this machine wasn't secured enough, we can bypass it by spawning a bash shell:

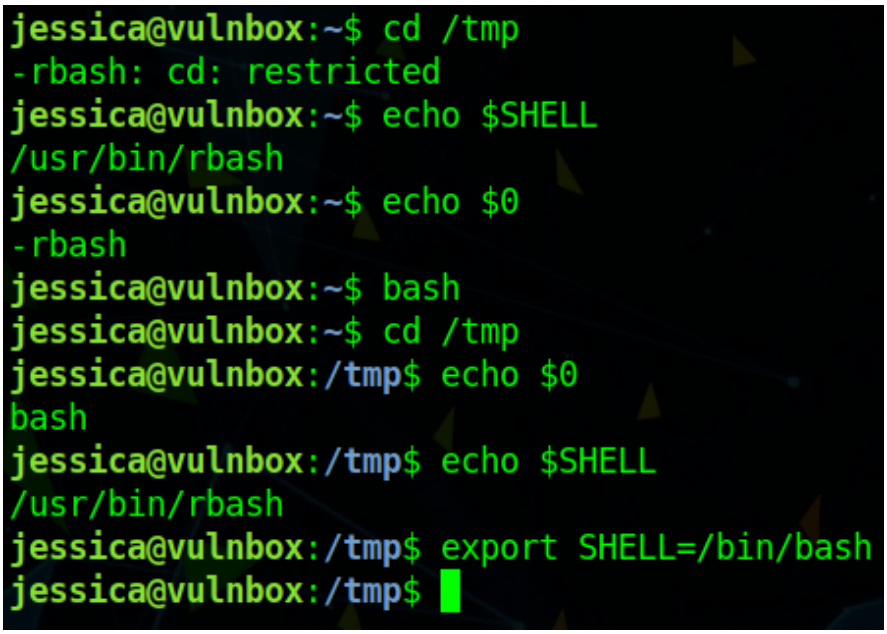


Figure 58: Bypassing a poorly secured `rbash`

Enumerating the system as this new user, we find out there's an internal service listening on port 25, which is a default port for **SMTP**, so, we check our inbox and we have an unread mail:

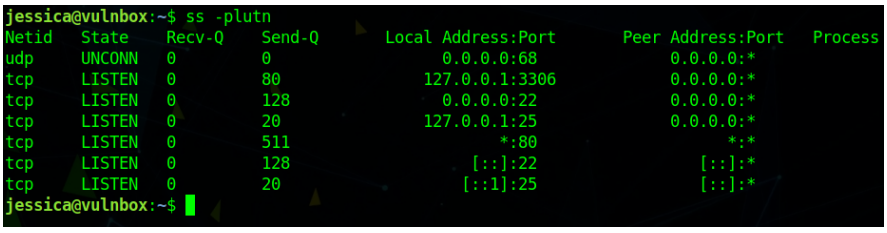


Figure 59: Listing all open ports in the victim machine

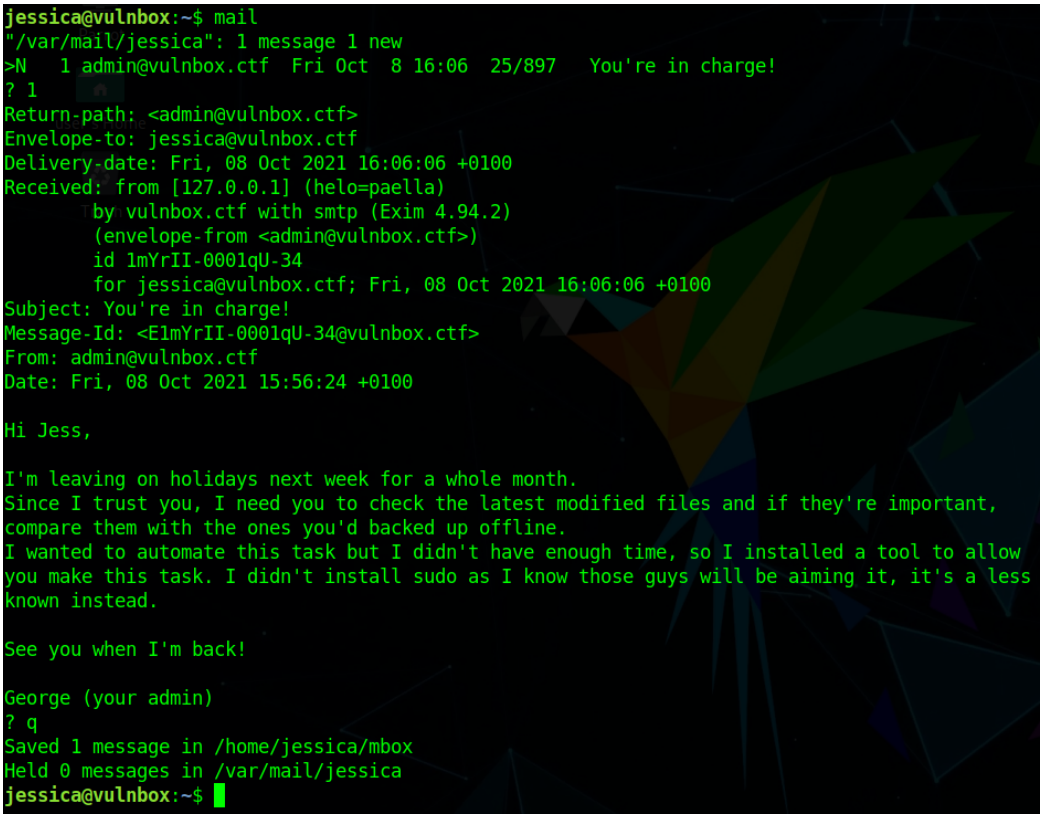
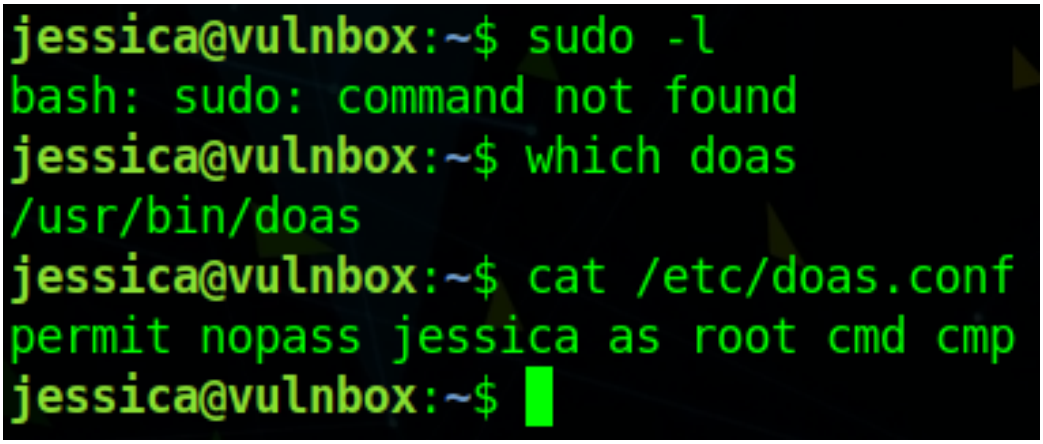


Figure 60: Reading a mail

From the mail we know `sudo` is not installed, but there's a similar utility, like `doas`:



From the configuration file we see the user **jessica** can run as **root** the command `cmp` without providing any password.

How to abuse this? There's a great resource called GTFOBins which shows different ways to abuse *Linux* binaries depending on certain situations:

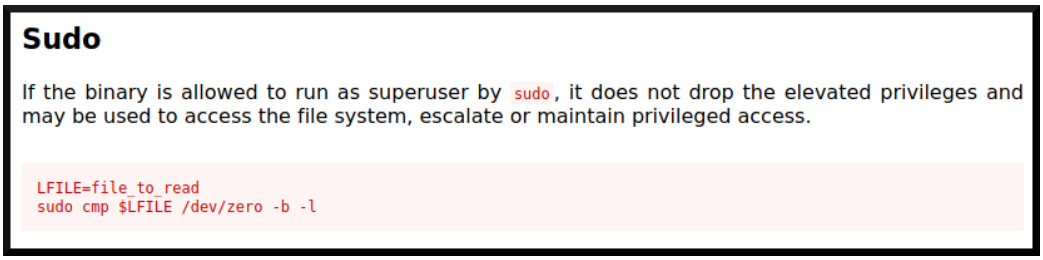


Figure 61: Getting info from **GTFOBins** on how to abuse the binary

Basically we can compare two files and get the output. With this, we can think already what we should do: read the flag.

I'm not going to do so right now, I have two better purposes. The first one is to get shell as **root**, and the second one is actually explaining why I've done what I've done.

To make the most of it, I think I should start by explaining what I'm going to do and why. What I want to do is easy: to read a privileged file, like `/etc/shadow`, where all the password hashes from system users are stored,

and see if any of them is crackable.

Take the next image as a simple example on how to read a file with `cmp`:

Figure 62: Simple example on how to read a file with `cmp`

About the output format of `cmp -bl`:

1. The first column is the character position to compare.
2. The second column is the decimal code of the character at the position to compare from the first file.
3. The third column is the **ASCII** representation of the character at the position to compare from the first file.
4. The fourth column is the decimal code of the character at the position to compare from the second file.
5. The fifth column is the **ASCII** representation of the character at the position to compare from the second file.

I know it seems a mouthful, but basically the order of the arguments matter, and we are more interested in the **ASCII** representation rather than

their decimal code. And you may have noticed there were some missing positions at the far left...

Notes to take into account:

1. When the same character is found in the same position at both files, the output of this is omitted.
2. `cmp` stops comparing when a **EOF**[12] is reached.
3. The *newline* character is represented as `^J`.

Knowing this, if we want to leak the whole `/etc/shadow` contents, we need to use a file larger than that one, and make sure any of the characters on them match at any position.

Find a file with this characteristics seems hard, but we can craft one at will. We can use the `\0` character which is meant to indicate the end of a string and we won't find it in a text file. And we can make it as large as we want:

```
jessica@vulnbox:/tmp/privesc$ dd if=/dev/zero of=zero.dd bs=1M count=1
1+0 records in
1+0 records out
1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.00218397 s, 480 MB/s
jessica@vulnbox:/tmp/privesc$ file zero.dd
zero.dd: data
jessica@vulnbox:/tmp/privesc$ xxd zero.dd | head
00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000080: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000090: 0000 0000 0000 0000 0000 0000 0000 0000  ................
jessica@vulnbox:/tmp/privesc$
```

Figure 63: Crafting a file a special file

We don't know the size of `/etc/shadow`, but we know it's not 1 megabyte.

[12] **EOF**: *End Of File*.

To get the `/etc/shadow` contents:

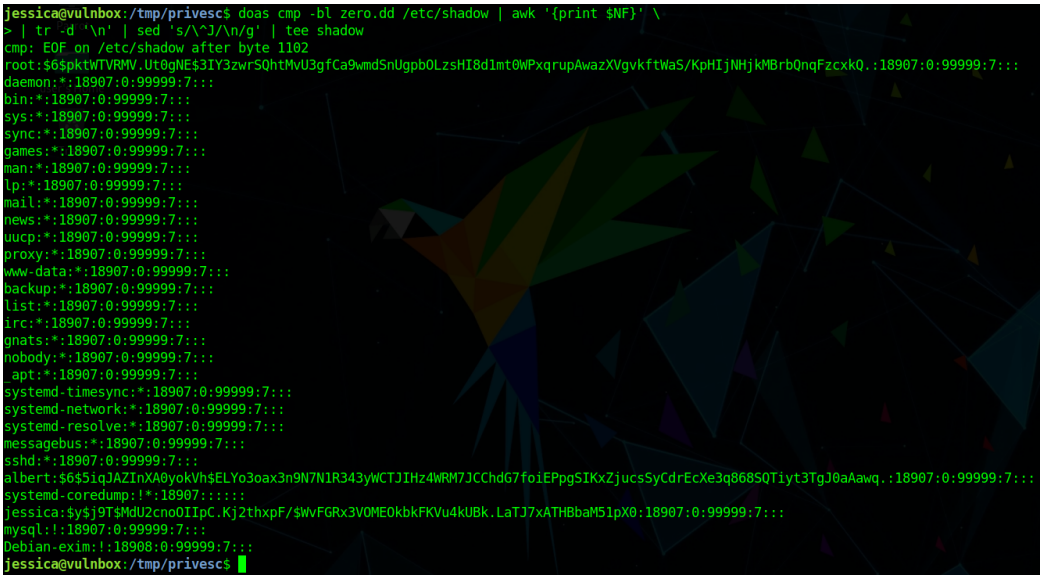


Figure 64: Leaking `/etc/shadow` with `cmp`

1. `doas cmp -bl zero.dd /etc/shadow`
   We compare, as **root**, our crafted file with our target one.

2. `awk '{print $NF}'`
   We filter the last column, which contains the **ASCII** value of our target file.

3. `tr -d '\n'`
   We remove the *newline* characters. Otherwise, we would end up with one character by line.

4. `sed 's/\^J/\n/g'`
   We substitute the `^J` character with a *newline*, as it's what it actually represents.

5. `tee shadow`
   We save the output in a file named `shadow` at the same time the output is shown on the screen.

Let's try to see if the **root** password is within `rockyou.txt`. But first, we should transfer the file:

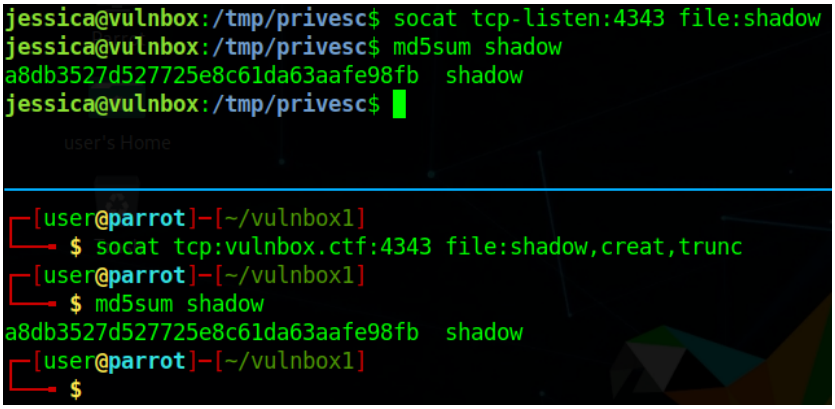


Figure 65: File transfer with `socat`

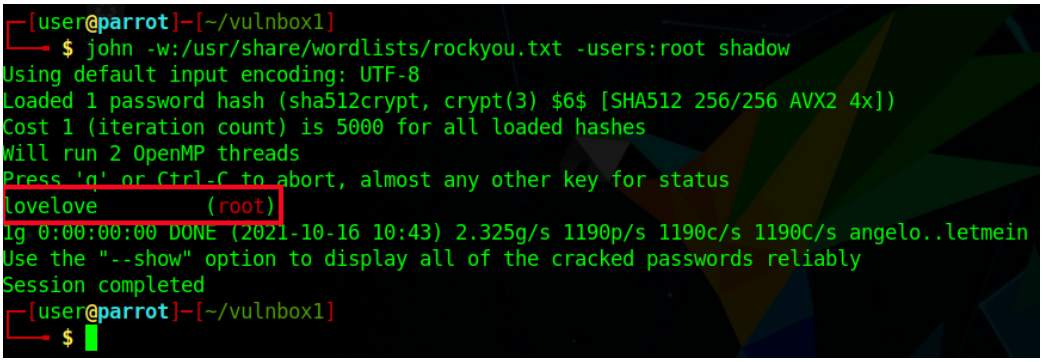


Figure 66: Cracking **root** password from `shadow`

And we have the **root** password: **lovelove**.

Let's get access as **root** and read the last flag:

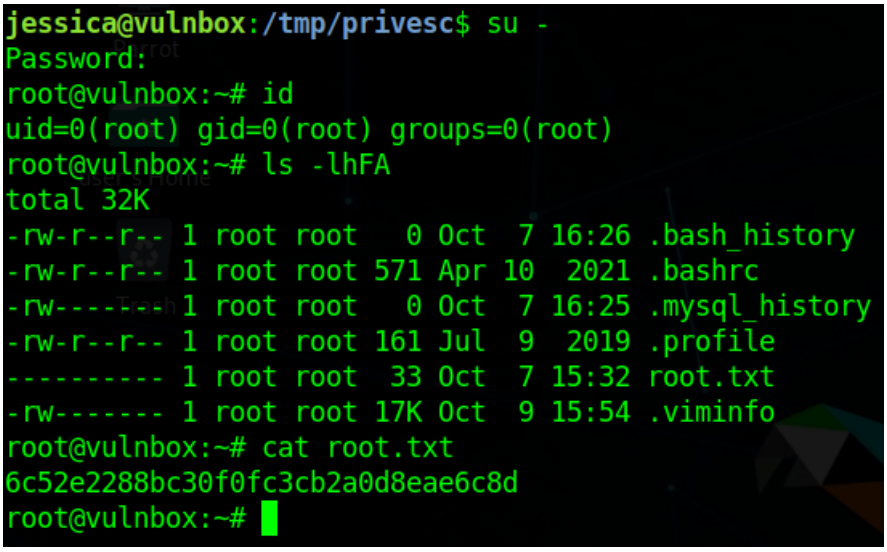


Figure 67: Gaining **root** access and reading `/root/root.txt`

2.2 Flags

albert's flag:
`3f440ebbc04e5d97a06383555cd50ba1`
jessica's flag:
`1ac3690338689317c1220f569a067c41`
root's flag:
`6c52e2288bc30f0fc3cb2a0d8eae6c8d`