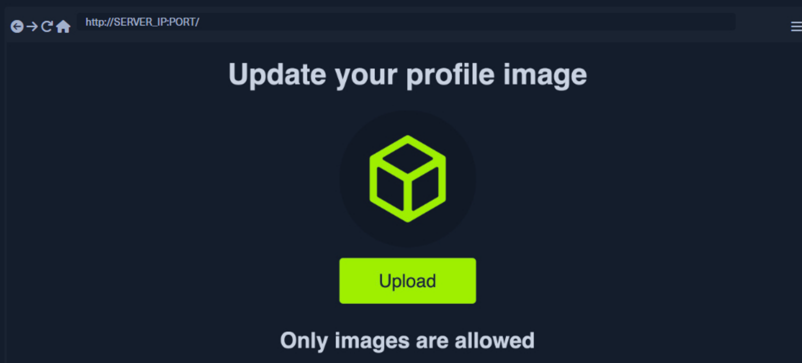
Whitelist Filters

As discussed in the previous section, the other type of file extension validation is by utilizing a `whitelist of allowed file extensions`. A whitelist is generally more secure than a blacklist. The web server would only allow the specified extensions, and the list would not need to be comprehensive in covering uncommon extensions.

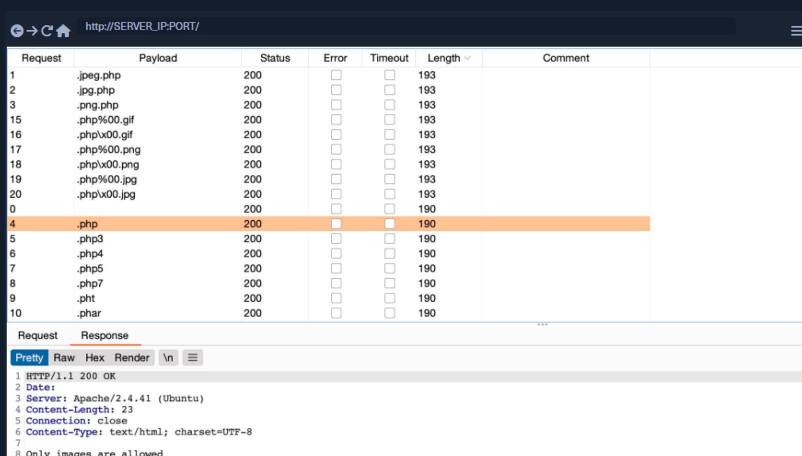
Still, there are different use cases for a blacklist and for a whitelist. A blacklist may be helpful in cases where the upload functionality needs to allow a wide variety of file types (e.g., File Manager), while a whitelist is usually only used with upload functionalities where only a few file types are allowed. Both may also be used in tandem.

Whitelisting Extensions

Let's start the exercise at the end of this section and attempt to upload an uncommon PHP extension, like `.phtml`, and see if we are still able to upload it as we did in the previous section:



We see that we get a message saying `Only images are allowed`, which may be more common in web apps than seeing a blocked extension type. However, error messages do not always reflect which form of validation is being utilized, so let's try to fuzz for allowed extensions as we did in the previous section, using the same wordlist that we used previously:



Request	Payload	Status	Error	Timeout	Length	Comment
1	.jpeg.php	200	☐	☐	193	
2	.jpg.php	200	☐	☐	193	
3	.png.php	200	☐	☐	193	
15	.php%00.gif	200	☐	☐	193	
16	.php\x00.gif	200	☐	☐	193	
17	.php%00.png	200	☐	☐	193	
18	.php\x00.png	200	☐	☐	193	
19	.php%00.jpg	200	☐	☐	193	
20	.php\x00.jpg	200	☐	☐	193	
0		200	☐	☐	190	
4	.php	200	☐	☐	190	
5	.php3	200	☐	☐	190	
6	.php4	200	☐	☐	190	
7	.php5	200	☐	☐	190	
8	.php7	200	☐	☐	190	
9	.pht	200	☐	☐	190	
10	.phar	200	☐	☐	190	

Request Response

Pretty Raw Hex Render \n

```
HTTP/1.1 200 OK
Date:
Server: Apache/2.4.41 (Ubuntu)
Content-Length: 23
Connection: close
Content-Type: text/html; charset=UTF-8

Only images are allowed
```

We can see that all variations of PHP extensions are blocked (e.g. `php5`, `php7`, `phtml`). However, the wordlist we used also contained other 'malicious' extensions that were not blocked and were successfully uploaded. So, let's try to understand how we were able to upload these extensions and in which cases we may be able to utilize them to execute PHP code on the back-end server.

The following is an example of a file extension whitelist test:

Code: php

```php
$fileName = basename($_FILES["uploadFile"]["name"]);

if (!preg_match('^.*\.(jpg|jpeg|png|gif)', $fileName)) {
    echo "Only images are allowed";
    die();
}
```

We see that the script uses a Regular Expression (`regex`) to test whether the filename contains any whitelisted image extensions. The issue here lies within the `regex`, as it only checks whether the file name `contains` the extension and not if it actually `ends` with it. Many developers make such mistakes due to a weak understanding of regex patterns.

So, let's see how we can bypass these tests to upload PHP scripts.

Double Extensions

The code only tests whether the file name contains an image extension; a straightforward method of passing the regex test is through `Double`

Cheat Sheet

? Go to Questions

Table of Contents

My Workstation

OFFLINE

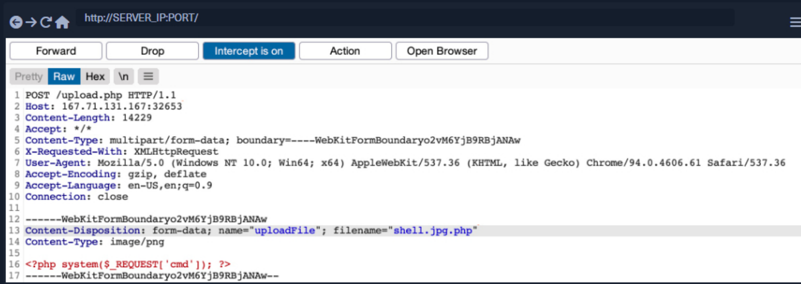
Start Instance

∞ / 1 spawns left

The code only tests whether the file name contains an image extension; a straightforward method of passing the regex test is through `Double Extensions`. For example, if the `.jpg` extension was allowed, we can add it in our uploaded file name and still end our filename with `.php` (e.g. `shell.jpg.php`), in which case we should be able to pass the whitelist test, while still uploading a PHP script that can execute PHP code.

> **Exercise:** Try to fuzz the upload form with This Wordlist to find what extensions are whitelisted by the upload form.

Let's intercept a normal upload request, and modify the file name to (`shell.jpg.php`), and modify its content to that of a web shell:



Now, if we visit the uploaded file and try to send a command, we can see that it does indeed successfully execute system commands, meaning that the file we uploaded is a fully working PHP script:

However, this may not always work, as some web applications may use a strict `regex` pattern, as mentioned earlier, like the following:

Code: php

```php
if (!preg_match('/^.*\.(jpg|jpeg|png|gif)$/', $fileName)) { ...SNIP... }
```

This pattern should only consider the final file extension, as it uses (`^.*\.`) to match everything up to the last (`.`), and then uses (`$`) at the end to only match extensions that end the file name. So, the `above attack would not work`. Nevertheless, some exploitation techniques may allow us to bypass this pattern, but most rely on misconfigurations or outdated systems.

Reverse Double Extension

In some cases, the file upload functionality itself may not be vulnerable, but the web server configuration may lead to a vulnerability. For example, an organization may use an open-source web application, which has a file upload functionality. Even if the file upload functionality uses a strict regex pattern that only matches the final extension in the file name, the organization may use the insecure configurations for the web server.

For example, the `/etc/apache2/mods-enabled/php7.4.conf` for the `Apache2` web server may include the following configuration:

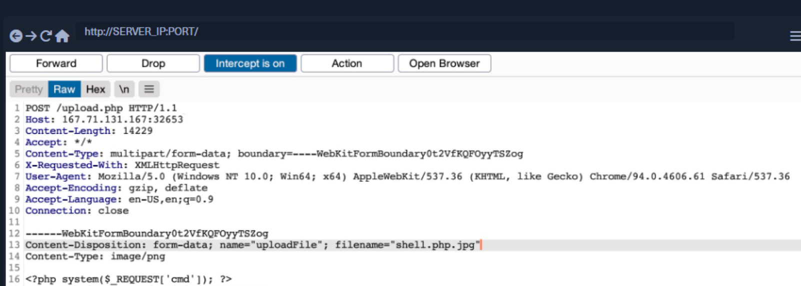
Code: xml

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

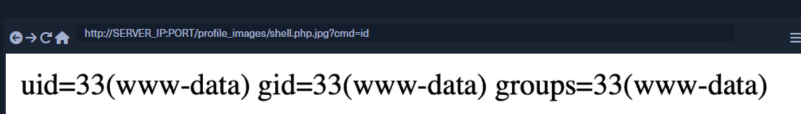
The above configuration is how the web server determines which files to allow PHP code execution. It specifies a whitelist with a regex pattern that matches `.phar`, `.php`, and `.phtml`. However, this regex pattern can have the same mistake we saw earlier if we forget to end it with (`$`). In such cases, any file that contains the above extensions will be allowed PHP code execution, even if it does not end with the PHP extension. For example, the file name (`shell.php.jpg`) should pass the earlier whitelist test as it ends with (`.jpg`), and it would be able to execute PHP code due to the above misconfiguration, as it contains (`.php`) in its name.

> **Exercise:** The web application may still utilize a blacklist to deny requests containing `PHP` extensions. Try to fuzz the upload form with the PHP Wordlist to find what extensions are blacklisted by the upload form.

Let's try to intercept a normal image upload request, and use the above file name to pass the strict whitelist test:



Now, we can visit the uploaded file, and attempt to execute a command:



As we can see, we successfully bypassed the strict whitelist test and exploited the web server misconfiguration to execute PHP code and gain control over the server.

Character Injection

Finally, let's discuss another method of bypassing a whitelist validation test through `Character Injection`. We can inject several characters before or after the final extension to cause the web application to misinterpret the filename and execute the uploaded file as a PHP script.

The following are some of the characters we may try injecting:

- `%20`
- `%0a`
- `%00`
- `%0d0a`
- `/`
- `.\`
- `.`
- `…`
- `:`

Each character has a specific use case that may trick the web application to misinterpret the file extension. For example, (`shell.php%00.jpg`) works with PHP servers with version `5.X` or earlier, as it causes the PHP web server to end the file name after the (`%00`), and store it as (`shell.php`), while still passing the whitelist. The same may be used with web applications hosted on a Windows server by injecting a colon (`:`) before the allowed file extension (e.g. `shell.aspx:.jpg`), which should also write the file as (`shell.aspx`). Similarly, each of the other characters has a use case that may allow us to upload a PHP script while bypassing the type validation test.

We can write a small bash script that generates all permutations of the file name, where the above characters would be injected before and after both the `PHP` and `JPG` extensions, as follows:

Code: bash

```bash
for char in '%20' '%0a' '%00' '%0d0a' '/' '.\\' '.' '…' ':'; do
    for ext in '.php' '.phps'; do
        echo "shell$char$ext.jpg" >> wordlist.txt
        echo "shell$ext$char.jpg" >> wordlist.txt
        echo "shell.jpg$char$ext" >> wordlist.txt
        echo "shell.jpg$ext$char" >> wordlist.txt
    done
done
```

With this custom wordlist, we can run a fuzzing scan with `Burp Intruder`, similar to the ones we did earlier. If either the back-end or the web server is outdated or has certain misconfigurations, some of the generated filenames may bypass the whitelist test and execute PHP code.

> **Exercise:** Try to add more PHP extensions to the above script to generate more filename permutations, then fuzz the upload functionality with the generated wordlist to see which of the generated file names can be uploaded, and which may execute PHP code after being uploaded.

Questions

Answer the question(s) below to complete this Section and earn cubes!

Cheat Sheet

Target(s): Click here to spawn the target system!

+ 2 The above exercise employs a blacklist and a whitelist test to block unwanted extensions and only allow image extensions. Try to bypass both to upload a PHP script and execute code to read "/flag.txt"

HTB{1_wh173l157_my53lf}

Submit Hint

Previous Next

Mark Complete & Next