Blacklist Filters

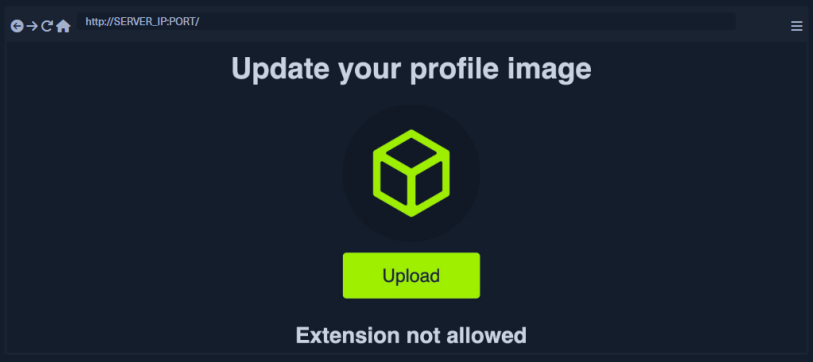
In the previous section, we saw an example of a web application that only applied type validation controls on the front-end (i.e., client-side), which made it trivial to bypass these controls. This is why it is always recommended to implement all security-related controls on the back-end server, where attackers cannot directly manipulate it.

Still, if the type validation controls on the back-end server were not securely coded, an attacker can utilize multiple techniques to bypass them and reach PHP file uploads.

The exercise we find in this section is similar to the one we saw in the previous section, but it has a blacklist of disallowed extensions to prevent uploading web scripts. We will see why using a blacklist of common extensions may not be enough to prevent arbitrary file uploads and discuss several methods to bypass it.

Blacklisting Extensions

Let's start by trying one of the client-side bypasses we learned in the previous section to upload a PHP script to the back-end server. We'll intercept an image upload request with Burp, replace the file content and filename with our PHP script's, and forward the request:



As we can see, our attack did not succeed this time, as we got `Extension not allowed`. This indicates that the web application may have some form of file type validation on the back-end, in addition to the front-end validations.

There are generally two common forms of validating a file extension on the back-end:

1. Testing against a `blacklist` of types
2. Testing against a `whitelist` of types

Furthermore, the validation may also check the `file type` or the `file content` for type matching. The weakest form of validation amongst these is `testing the file extension against a blacklist of extension` to determine whether the upload request should be blocked. For example, the following piece of code checks if the uploaded file extension is `PHP` and drops the request if it is:

Code: php

```
$fileName = basename($_FILES["uploadFile"]["name"]);
$extension = pathinfo($fileName, PATHINFO_EXTENSION);
$blacklist = array('php', 'php7', 'phps');

if (in_array($extension, $blacklist)) {
    echo "File type not allowed";
    die();
}
```

The code is taking the file extension (`$extension`) from the uploaded file name (`$fileName`) and then comparing it against a list of blacklisted extensions (`$blacklist`). However, this validation method has a major flaw. `It is not comprehensive`, as many other extensions are not included in this list, which may still be used to execute PHP code on the back-end server if uploaded.

> **Tip:** The comparison above is also case-sensitive, and is only considering lowercase extensions. In Windows Servers, file names are case insensitive, so we may try uploading a `php` with a mixed-case (e.g. `pHp`), which may bypass the blacklist as well, and should still execute as a PHP script.

So, let's try to exploit this weakness to bypass the blacklist and upload a PHP file.

Fuzzing Extensions

As the web application seems to be testing the file extension, our first step is to fuzz the upload functionality with a list of potential extensions and see which of them return the previous error message. Any upload requests that do not return an error message, return a different message, or succeed in uploading the file, may indicate an allowed file extension.

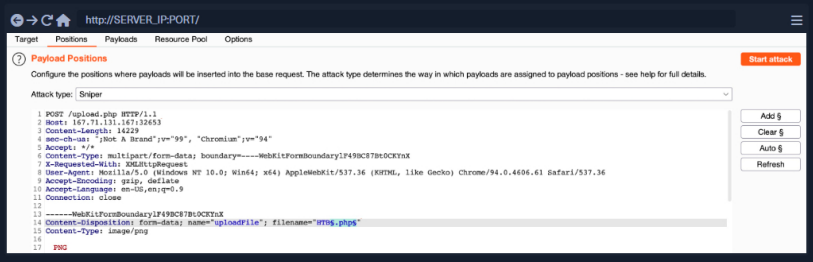
There are many lists of extensions we can utilize in our fuzzing scan. `PayloadsAllTheThings` provides lists of extensions for PHP and .NET web applications. We may also use `SecLists` list of common Web Extensions.

We may use any of the above lists for our fuzzing scan. As we are testing a PHP application, we will download and use the above PHP list. Then, from `Burp History`, we can locate our last request to `/upload.php`, right-click on it, and select `Send to Intruder`. From the `Positions` tab, we can `Clear` any automatically set positions, and then select the `.php` extension in `filename="HTB.php"` and click the `Add` button to add it

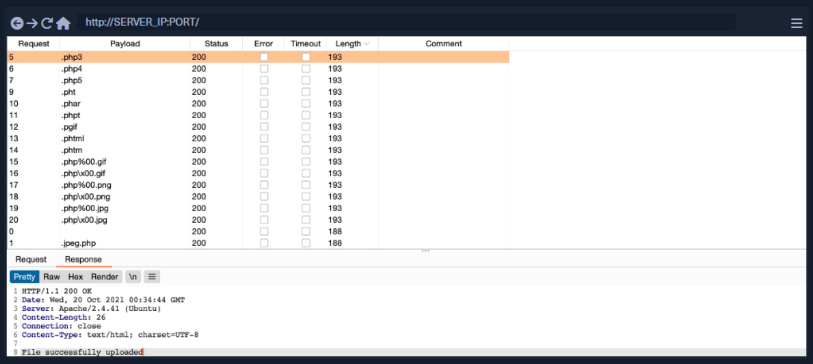
Cheat Sheet
Go to Questions

Table of Contents
Intro to File Upload Attacks
Basic Exploitation
Absent Validation
Upload Exploitation
Bypassing Filters
Client-Side Validation
Blacklist Filters
Whitelist Filters
Type Filters
Other Upload Attacks
Limited File Uploads
Other Upload Attacks
Prevention
Preventing File Upload Vulnerabilities
Skills Assessment
Skills Assessment - File Upload Attacks

as a fuzzing position:



We'll keep the file content for this attack, as we are only interested in fuzzing file extensions. Finally, we can `Load` the PHP extensions list from above in the `Payloads` tab under `Payload Options`. We will also un-tick the `URL Encoding` option to avoid encoding the (`.`) before the file extension. Once this is done, we can click on `Start Attack` to start fuzzing for file extensions that are not blacklisted:

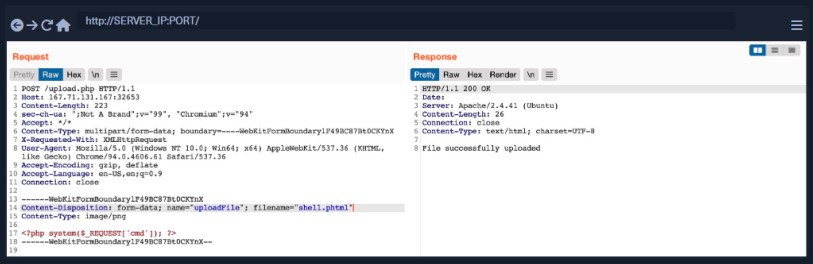


We can sort the results by `Length`, and we will see that all requests with the Content-Length (`193`) passed the extension validation, as they all responded with `File successfully uploaded`. In contrast, the rest responded with an error message saying `Extension not allowed`.

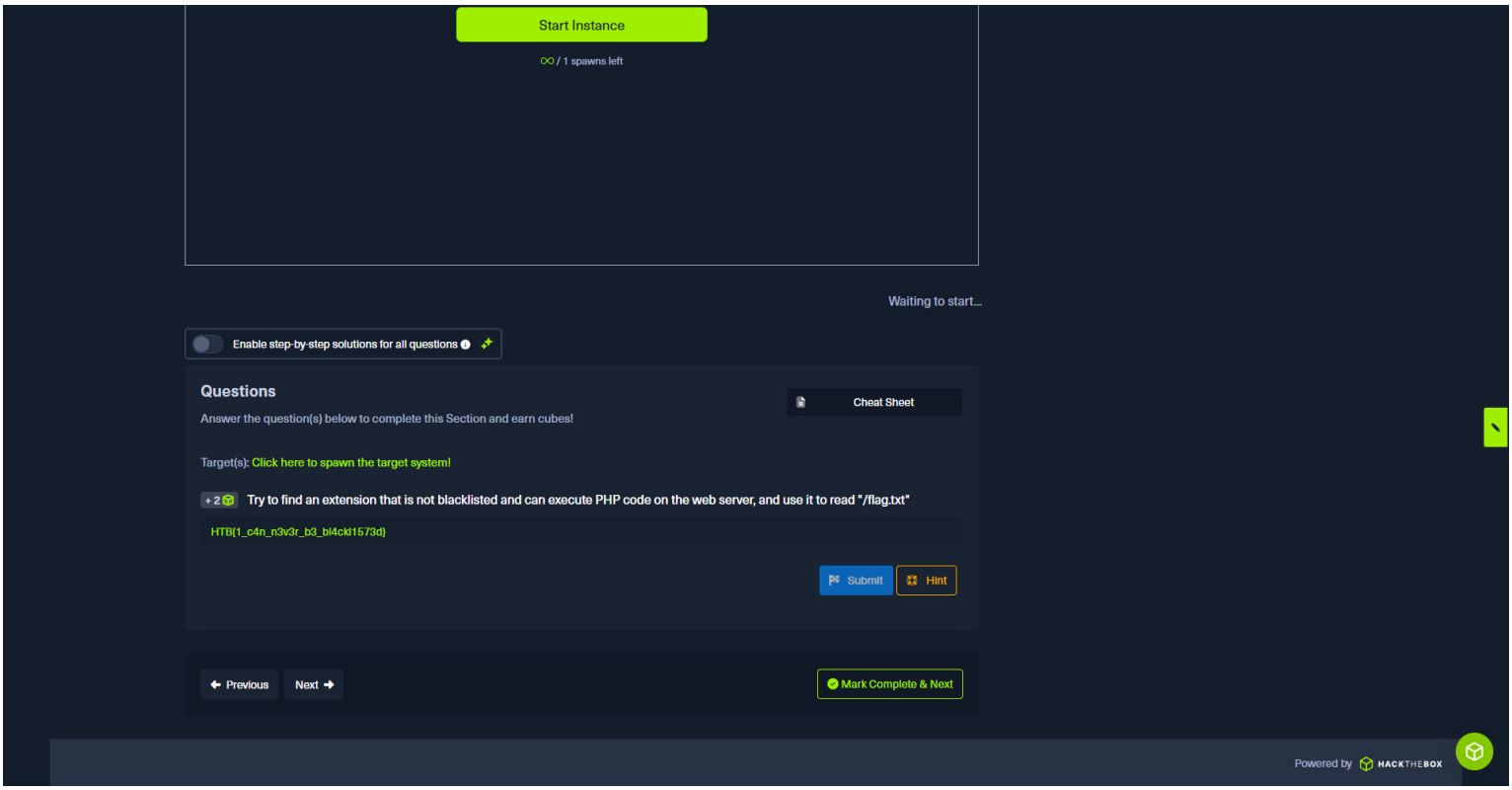
Non-Blacklisted Extensions

Now, we can try uploading a file using any of the `allowed extensions` from above, and some of them may allow us to execute PHP code. `Not all extensions will work with all web server configurations`, so we may need to try several extensions to get one that successfully executes PHP code.

Let's use the `.phtml` extension, which PHP web servers often allow for code execution rights. We can right-click on its request in the Intruder results and select `Send to Repeater`. Now, all we have to do is repeat what we have done in the previous two sections by changing the file name to use the `.phtml` extension and changing the content to that of a PHP web shell:



As we can see, our file seems to have indeed been uploaded. The final step is to visit our upload file, which should be under the image upload directory (`profile_images`), as we saw in the previous section. Then, we can test executing a command, which should confirm that we successfully bypassed the blacklist and uploaded our web shell:


Start Instance
∞ / 1 spawns left
Waiting to start...
Enable step-by-step solutions for all questions
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
Try to find an extension that is not blacklisted and can execute PHP code on the web server, and use it to read "/flag.txt"
HTB{1_c4n_n3v3r_b3_bl4ckl1573d}
Submit
Hint
Previous
Next
Mark Complete & Next
Powered by HACKTHEBOX