Project Deliverables

This deliverable will talk about the security concepts, attacker mindset, and personal reflections I developed throughout this assessment.

ALL inspiration for my CTFs and listed at the appendix section at the bottom.

GitHub for the CTFS and the discord bot:

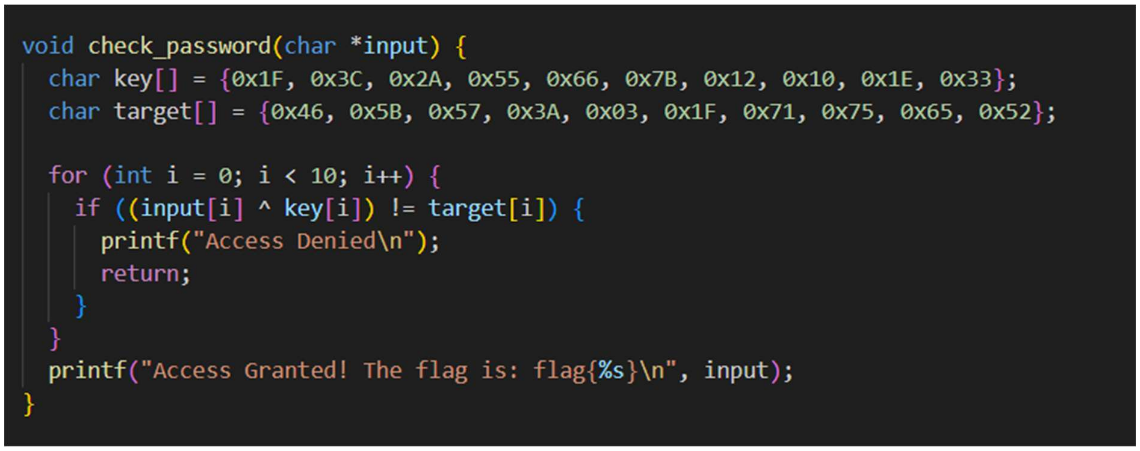
https://github.com/Outpacing82/Ctfs-and-discord-bot

CTF 1: Reverse Engineering

This CTF focuses on Binary Analysis, where the player is expected to reverse XOR operations to derive the key.

Methodology:

- app.c does not print any clear information or flags, but relies on calculations to validate passwords.
- Key and target are hardcoded in a confusing form, so someone without much experience can't crack it easily.



```
void check_password(char *input) {
  char key[] = {0x1F, 0x3C, 0x2A, 0x55, 0x66, 0x7B, 0x12, 0x10, 0x1E, 0x33};
  char target[] = {0x46, 0x5B, 0x57, 0x3A, 0x03, 0x1F, 0x71, 0x75, 0x65, 0x52};

  for (int i = 0; i < 10; i++) {
    if ((input[i] ^ key[i]) != target[i]) {
      printf("Access Denied\n");
      return;
    }
  }
  printf("Access Granted! The flag is: flag{%s}\n", input);
}
```

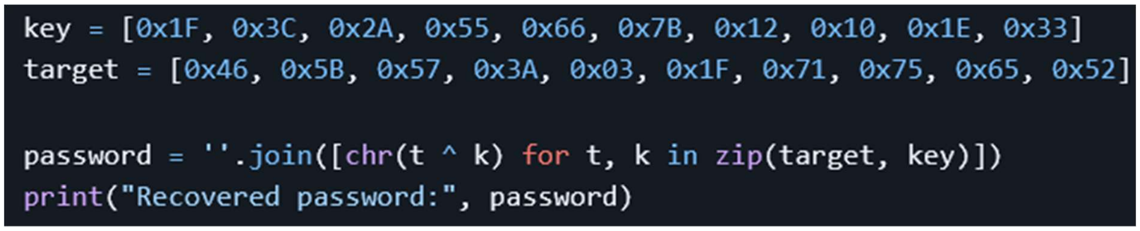
Here, I wanted the attacker to reverse the XOR operation to get the correct key. In the for loop, each input is matched with its corresponding key on an index number.

Example of an XOR operation:

Key [0x1F]	= 0001 1111
Target [0x46]	= 0100 0110
Output	= 0101 1001 (0x59)

From the example operation above, the first correct letter of the password is 0x59, otherwise convertible to 'Y' using ascii conversion. A harder option of the challenge is available, where the user needs to write out code to reverse the CTF instead of reversing the XOR operation by hand.

During the creation of the challenge, coming up with the idea and a flexible solution that will work with any key and target group pair was the hardest part.



```python
key = [0x1F, 0x3C, 0x2A, 0x55, 0x66, 0x7B, 0x12, 0x10, 0x1E, 0x33]
target = [0x46, 0x5B, 0x57, 0x3A, 0x03, 0x1F, 0x71, 0x75, 0x65, 0x52]

password = ''.join([chr(t ^ k) for t, k in zip(target, key)])
print("Recovered password:", password)
```

In my solution, the realization that XOR is reversible in any-way is important (if input ^ key = target then target ^ key = input). They will then combine the target and keys together and convert them into characters.

CTF 2: Binary Exploit

In this CTF, I referenced the week 2 wargames, where users are expected to use techniques like objdump, memory layout and buffer overflows to find the relevant function's address, which is win() in this case.

Methodology:

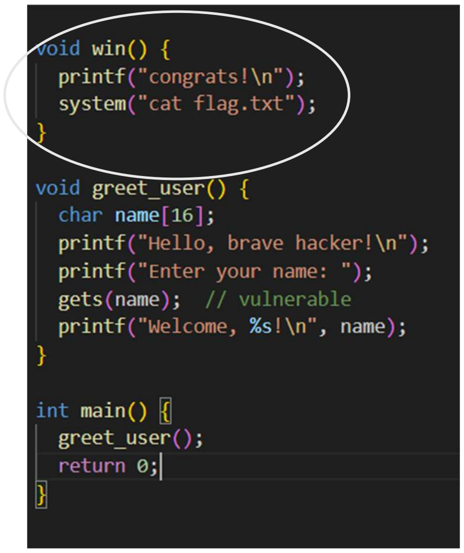
- Used the classic gets() vulnerability, as adjacent memory could get overwritten
- Placed a secret function win(), that prints the flag when successfully called
- README.md with subtle clues to solve the CTF

Once the address is obtained, they should be able to get the little-endian version of the address by reversing every hex number. Combined with the tip(name array has 16 bytes) that I gave in READ.md, they should be able to write a command like:

python3 -c 'import sys; sys.stdout.buffer.write(b"A"*24 + b"\x76\x11\x40\x00\x00\x00\x00\x00")' | ./vuln

The main concept of this CTF is to overwrite the return address so that we can jump to win instead of main when we return.

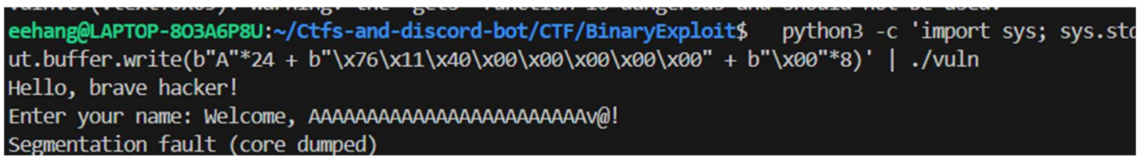
During the challenge, I originally had the flag be in a separate .txt file to avoid spoilers, but segmentation faults happened. I was reminded that these are caused by the return pointer of the program being destroyed, so I could only place the flag directly in the same file to allow the flag to appear.



```
void win() {
  printf("congrats!\n");
  system("cat flag.txt");
}

void greet_user() {
  char name[16];
  printf("Hello, brave hacker!\n");
  printf("Enter your name: ");
  gets(name);  // vulnerable
  printf("Welcome, %s!\n", name);
}

int main() {
  greet_user();
  return 0;
}
```



```
eehang@LAPTOP-8O3A6P8U:~/Ctfs-and-discord-bot/CTF/BinaryExploit$  python3 -c 'import sys; sys.std
ut.buffer.write(b"A"*24 + b"\x76\x11\x40\x00\x00\x00\x00\x00" + b"\x00"*8)' | ./vuln
Hello, brave hacker!
Enter your name: Welcome, AAAAAAAAAAAAAAAAAAAAAAAAv@!
Segmentation fault (core dumped)
```

How the CTF is made harder:

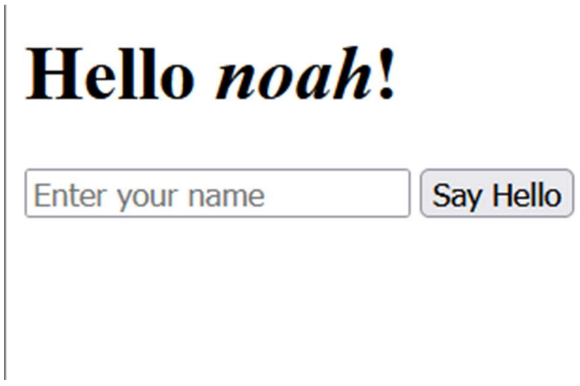
- Avoided calling win() in normal execution, so attackers cannot reach the flag accidentally
- Return address is 8 bytes, and the buffer size is 16 bytes. This requires precision since the buffer size is so small.

I hope to implement stack canaries next time, which will increase difficulty since players would have to leak the canary.

CTF 3: XSS (Cross-Site Scripting)

This was inspired by week 3's wargames, where we learned to not let user input showed freely as they could make your website do anything.

A clear objective was established from the get-go, where attackers would need to make an alert appear to win this CTF. They are expected to try out various types of input for the app, then realize that the app echoes back whatever we feed them, even **scripts**.

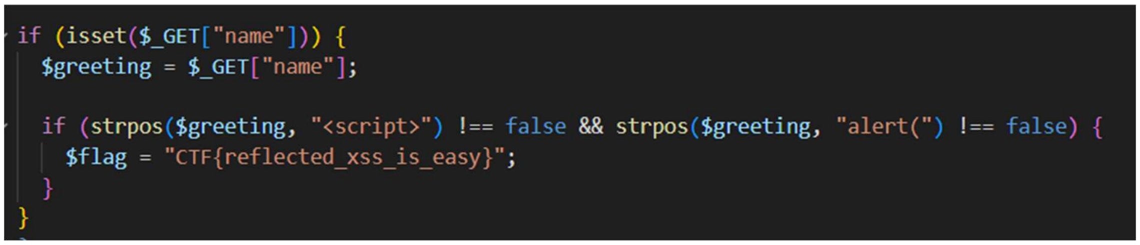

Hello *noah*!

Enter your name

Say Hello

After I entered <i>noah</i> as input

The app currently checks if the input is a script and an alert, and if the flag is empty and empty string is displayed on the webpage.


```
if (isset($_GET["name"])) {
  $greeting = $_GET["name"];

  if (strpos($greeting, "<script>") !== false && strpos($greeting, "alert(") !== false) {
    $flag = "CTF{reflected_xss_is_easy}";
  }
}
```

I learned that the app could be made harder by introducing filtering, and a valid improvement could be:

$name = str_replace(["<", ">", "script"], "", $_GET["name"]);

However, experienced users will just find another way to circumvent this, which is possible using img tags and triggering the onerror tag deliberately, causing an error to pop up: http://localhost:8080/index.php?name=<img src=x onerror=alert(1)>

After some research, I concluded that making CSP headers are the most surefire way to keep websites safe, as they would prevent a script from running in the first place:

Content-Security-Policy: script-src 'self'

(self only allows scripts to be loaded from the same origin as the page itself)

CTF 4: Web Authentication

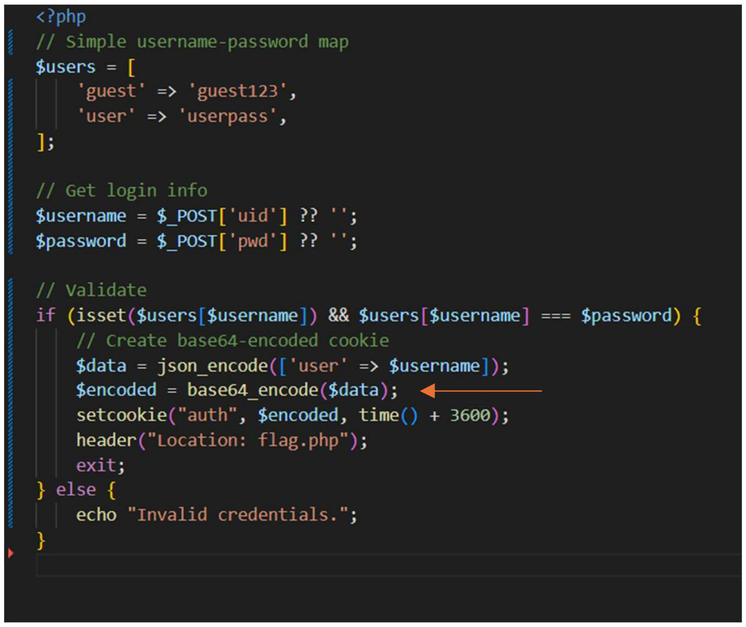
This is one of my harder CTFs, as it combines web authentication, session and cookie-handling, and even base64 data manipulation. The goal is to teach players that even small things like cookies can to serious vulnerabilities.

This challenge was made more difficult in the following ways:

- An admin account doesn't exist in the database, so the attacker will never be able to brute force the CTF with password guessing.

- Error messages don't reveal cookie structure; they are expected to inspect the cookie manually.
- No cookies are mentioned as a hint, so deep inspection is needed.
- The original version of this challenge allowed anyone to log in as "admin" using any password. While this still showed a basic logic flaw, it was far too easy. I improved the challenge by requiring login with valid (non-admin) credentials and expecting the attacker to manipulate a token post-login.

This part of my code is the core logic of the authentication system. After combining the username and password, I obscured it with base64, then finished it off by setting it as cookie that will expire in 1 hour (3600 seconds).

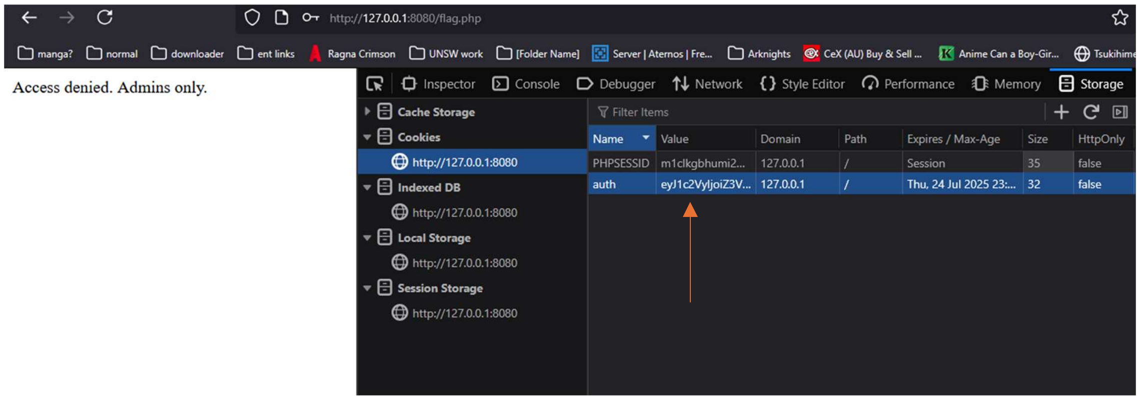


```php
<?php
// Simple username-password map
$users = [
    'guest' => 'guest123',
    'user' => 'userpass',
];

// Get login info
$username = $_POST['uid'] ?? '';
$password = $_POST['pwd'] ?? '';

// Validate
if (isset($users[$username]) && $users[$username] === $password) {
    // Create base64-encoded cookie
    $data = json_encode(['user' => $username]);
    $encoded = base64_encode($data);
    setcookie("auth", $encoded, time() + 3600);
    header("Location: flag.php");
    exit;
} else {
    echo "Invalid credentials.";
}
```

For the attacker path, after using the username and password I gave them, they receive a valid auth cookie but are denied access to the admin panel. This tells advanced players that some part of their session or identity still needs to be modified, which is most commonly done using DevTools.

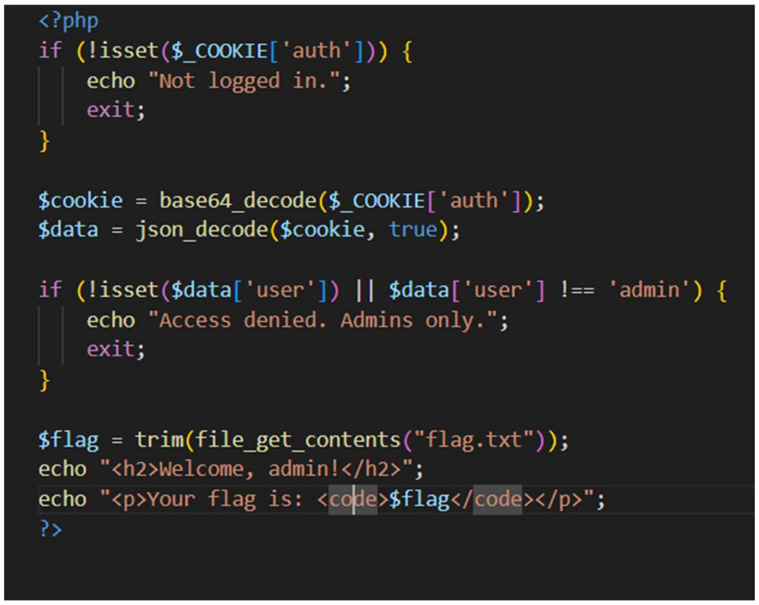


Access denied. Admins only.

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly
PHPSESSID	m1clkgbhumi2...	127.0.0.1	/	Session	35	false
auth	eyJ1c2VyIjoiZ3V...	127.0.0.1	/	Thu, 24 Jul 2025 23:...	32	false

After they obtain the token (ey11… that translates to {“user”:”guest”} after base64 decoding), they are to modify it to {“user”:”admin”} and re-encode it, replacing the cookie with an admin login.

As for the verification aspect, my code would expect the user to be modified to admin. Note that it must also be in base64 before reaching my backend.



```php
<?php
if (!isset($_COOKIE['auth'])) {
    echo "Not logged in.";
    exit;
}

$cookie = base64_decode($_COOKIE['auth']);
$data = json_decode($cookie, true);

if (!isset($data['user']) || $data['user'] !== 'admin') {
    echo "Access denied. Admins only.";
    exit;
}

$flag = trim(file_get_contents("flag.txt"));
echo "<h2>Welcome, admin!</h2>";
echo "<p>Your flag is: <code>$flag</code></p>";
?>
```

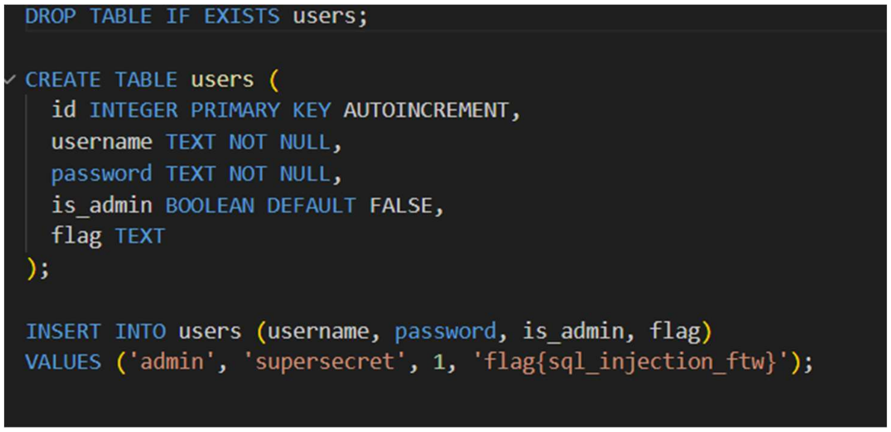
I learned a lot during the building of this CTF:

- Strengthened defensive and offensive intuition for identifying logic-flaws
- Base64 is actually not a security mechanism, it actually instils false confidence and only works on beginners.

- Session design must not only rely on cookies as they can easily be changed.

CTF 5: SQL Injection

This is a simple CTF that simulates a weak login system, where the attacker must exploit improper input sanitisation to obtain a hidden flag.

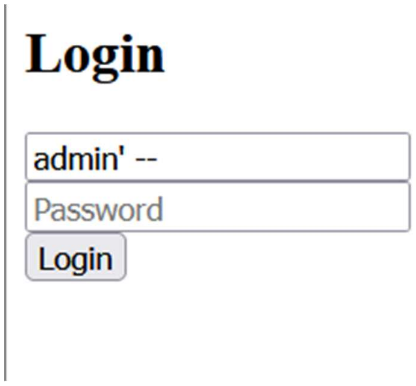


```sql
DROP TABLE IF EXISTS users;

CREATE TABLE users (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  username TEXT NOT NULL,
  password TEXT NOT NULL,
  is_admin BOOLEAN DEFAULT FALSE,
  flag TEXT
);

INSERT INTO users (username, password, is_admin, flag)
VALUES ('admin', 'supersecret', 1, 'flag{sql_injection_ftw}');
```

The attacker is expected to try out different sql injection methods and find out what works best. In this case, options like ' OR 1=1; -- and “admin' -- work well.



Login

admin' --

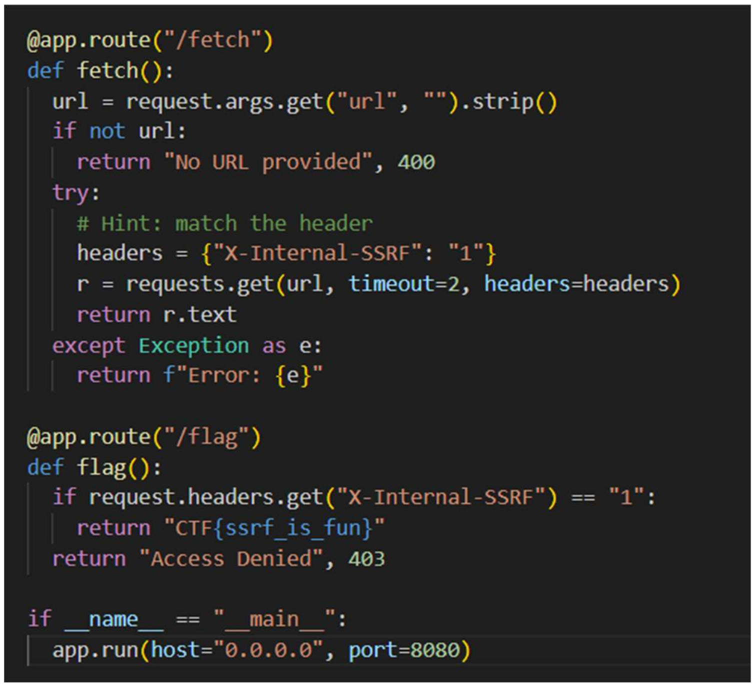
One of the errors I made was using DROP TABLE IF EXISTS users; in my SQL script, when SQLite3 does not support it.

In terms of how I made the challenge harder, common false payloads like admin ' – will fail, so the attacker must understand SQL parsing. In the future, I would implement login throttling, that will track failed attempts, forcing the attacker to think carefully before an attempt.

The hardest part of creating this CTF is admittedly the docker and sh part, a lot of research and documentation reading was needed to understand them.

CTF 6: SSRF (Server-Side Request Forgery)

This CTF is mainly inspired by the newest wargames in week 7, where I had trouble understanding how etc/hosts affect internal network accesses, and that lead to me designing a CTF that expects the player to manipulate URLS to access the flag.

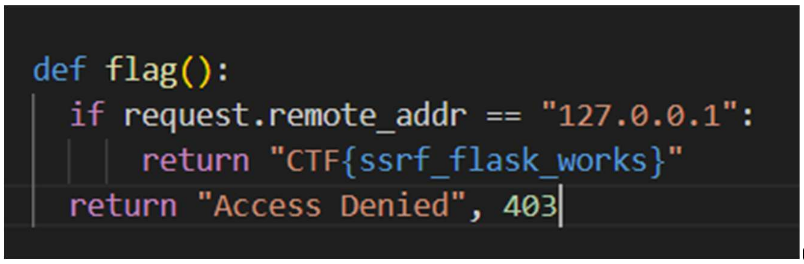


```
@app.route("/fetch")
def fetch():
    url = request.args.get("url", "").strip()
    if not url:
        return "No URL provided", 400
    try:
        # Hint: match the header
        headers = {"X-Internal-SSRF": "1"}
        r = requests.get(url, timeout=2, headers=headers)
        return r.text
    except Exception as e:
        return f"Error: {e}"

@app.route("/flag")
def flag():
    if request.headers.get("X-Internal-SSRF") == "1":
        return "CTF{ssrf_is_fun}"
    return "Access Denied", 403

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8080)
```

Initially, my plan was to check whether the request originated from the loopback IP address (127.0.0.1) before disclosing the flag. However, I quickly discovered a problem: since both the browser and the Flask server run on the same machine, all traffic naturally appears to originate from 127.0.0.1. This made it impossible to distinguish between internal and external requests, defeating the challenge's purpose.



```
def flag():
    if request.remote_addr == "127.0.0.1":
        return "CTF{ssrf_flask_works}"
    return "Access Denied", 403
```

(original flag check)

To circumvent this, I modified the flag endpoint to check for a custom HTTP header (X-Internal-SSRF) that is only included by internal requests triggered via the /fetch route. This also serves as an additional defence mechanism since typical SSRF-vulnerable apps like the above have no URL checks at all, and normal users can't set internal headers on requests to my server.

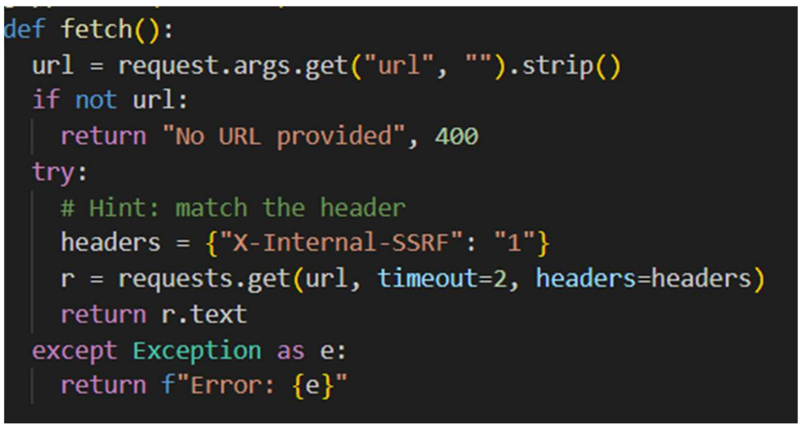
This header acts as an internal access token, simulating a situation where certain backend services are only accessible from within the server.

By requiring players to craft a Server-Side Request Forgery that leverages the /fetch endpoint to indirectly access /flag, the challenge becomes a more realistic and educational demonstration of how SSRF vulnerabilities can allow attackers to pivot inside a network and reach otherwise protected internal endpoints.

Any attacker would know that an app that makes server-side HTTP requests based on any user input is a huge red flag. Knowing this, they would try out multiple routes to try and hit the flag route, especially since I gave a hint that the route is at /flag.

Once they searched http://localhost:8080/flag in their browser, "Access , Missing X-Internal-SSRF Header" will be shown. This tells the attacker that only the links entered through the input would have the header.

Finally, once the user enters http://localhost:8080/flag into the input box instead of the browser, the flag will be obtained. This is because all URLs that went through my fetch function will contain the header.



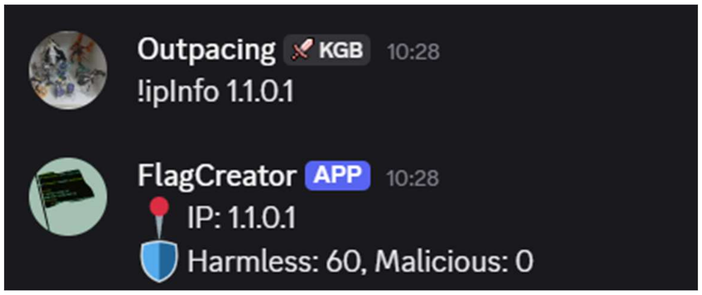
```python
def fetch():
    url = request.args.get("url", "").strip()
    if not url:
        return "No URL provided", 400
    try:
        # Hint: match the header
        headers = {"X-Internal-SSRF": "1"}
        r = requests.get(url, timeout=2, headers=headers)
        return r.text
    except Exception as e:
        return f"Error: {e}"
```

Discord bot

The discord bot is where the CTF challenges are sent out, it also contains some security measures and checks.

Commands:

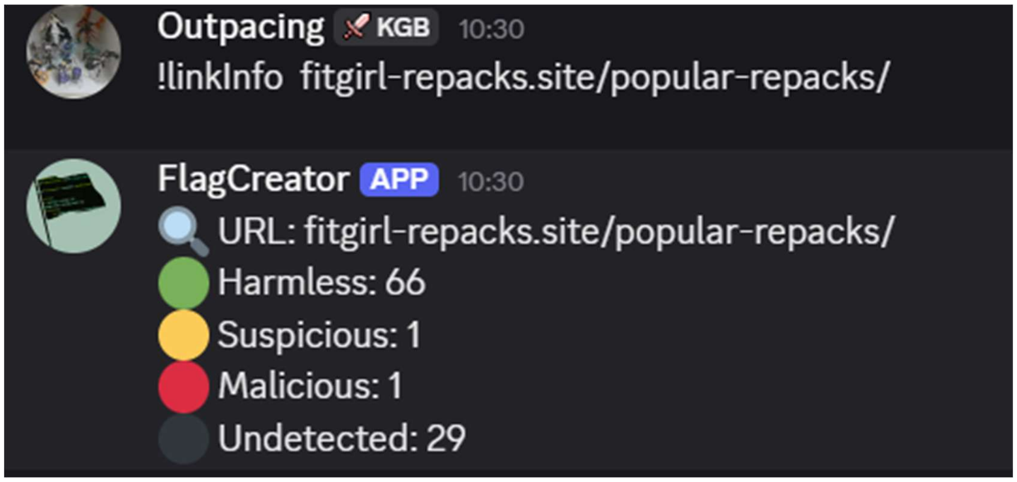
!ipInfo: Puts the given ip into VirusTotal, and separates the checks of antivirus companies into two categories, Harmless and Malicious



Outpacing KGB 10:28
!ipInfo 1.1.0.1

FlagCreator APP 10:28
📍 IP: 1.1.0.1
🛡️ Harmless: 60, Malicious: 0

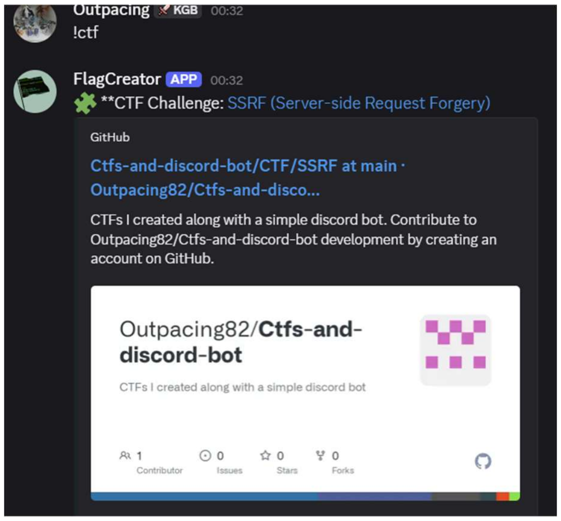
!linkInfo: Puts the given link into VirusTotal, and separates the checks of antivirus companies into four categories, Harmless, Suspicious, Malicious and Undetected.



Outpacing KGB 10:30
!linkInfo fitgirl-repacks.site/popular-repacks/

FlagCreator APP 10:30
🔍 URL: fitgirl-repacks.site/popular-repacks/
🟢 Harmless: 66
🟡 Suspicious: 1
🔴 Malicious: 1
⚫ Undetected: 29

!ctf: gives out the repo of one of the six CTFs that I made at random



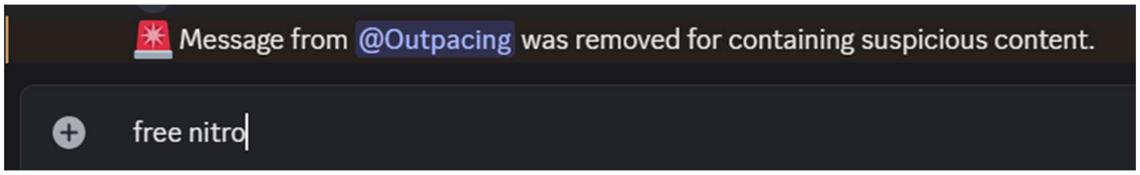
Outpacing KGB 00:32
!ctf

FlagCreator APP 00:32
🧩 **CTF Challenge: SSRF (Server-side Request Forgery)

GitHub
Ctfs-and-discord-bot/CTF/SSRF at main · Outpacing82/Ctfs-and-disco...
CTFs I created along with a simple discord bot. Contribute to Outpacing82/Ctfs-and-discord-bot development by creating an account on GitHub.

Outpacing82/**Ctfs-and-discord-bot**
CTFs I created along with a simple discord bot
1 Contributor · 0 Issues · 0 Stars · 0 Forks

Simple scam message filter: commonly used phishing terms would be automatically removed, and the person who sent the message will be given a warning.


🚨 Message from @Outpacing was removed for containing suspicious content.

free nitro

I believe my bot is extremely secure, as I have hidden my discord and virustotal API keys in my .env file and gitignored it. This measure is necessary to prevent unauthorized access to my bot.

Overall Conclusion and Reflection

Over the course of this project, I explored and implemented a wide range of security vulnerabilities through carefully crafted CTF challenges. Each challenge was unique and taught me specific exploitation concepts. The journey from conception to execution of these CTFs took a lot of brainstorming, planning and implementation. All these perfectly match the thought process encouraged by the lecturer, which is recon-planning-exploit-persistence-reporting-recon.

Security Concepts Learned

Through building challenges around reverse engineering, binary exploitation, web vulnerabilities (XSS, SQLi, SSRF), and authentication flaws, I developed an understanding of:

- Memory safety vulnerabilities and low-level system operations
- Common web application flaws and their real-world impact
- How poorly validated input can lead to complete application compromise
- The dangers of over-reliance on insecure encoding techniques (e.g., base64, unsalted hashes)

- The role of internal network structures in enabling SSRF attacks

Designing these challenges also required thinking like an attacker:

- How might a user manipulate input?
- What shortcuts or exploits would an experienced hacker try?
- What unintended behaviours can be triggered under edge cases?

This mindset will always be invaluable in both challenge design and testing.

Reflections on Tools and Ecosystem

- I gained significant fluency in tools like objdump, gdb, Python, SQLite3, and web tech stacks like Flask and PHP.
- Dockerizing the environment for consistent CTF deployment was difficult at first, but it also showed me why it is a better option for hosting, when compared to doing it locally.

Creating a Discord bot to distribute and interact with these CTFs extended the learning into practical cybersecurity tooling. It helped reinforce the value of automation and secure API integration (e.g., with VirusTotal) and forced me to consider real-world security practices like hiding API keys and sanitizing user commands. (leaking discord bot token would be disastrous for big bots like MEE6 and EpicRPG)

Final Thoughts

Moving forward, I plan to:

- Expand these CTFs with more advanced protections (e.g., ASLR, stack canaries, CSP enforcement, WAF simulation)
- Contribute to open-source security challenge platforms
- Explore bug bounty platforms to apply these skills to real-world scenarios

Appendix:

Reverse Engineering CTF inspiration: https://medium.com/@0xMr_Robot/nahamcon-ctf-2024-reverse-engineering-challenges-b397296721c1

HTML and frontend knowledge: COMP6080