TryHackMe Room Report

Gunasekara D T | IT23621138

IE2062 – Web Security

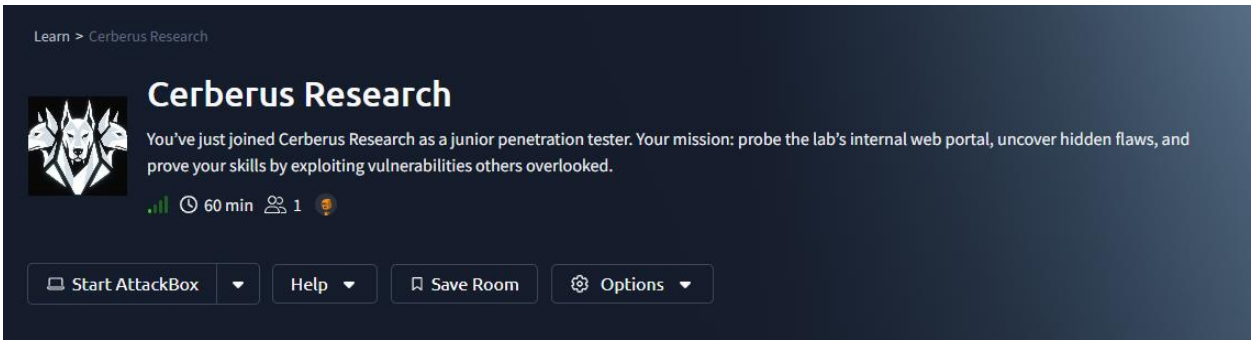
B.Sc. (Hons) in Information Technology specializing Cyber Security

Table of Contents

1. THM Room Introduction

1.1 THM Room Link

- https://tryhackme.com/jr/CerberusResearch



1.2 THM Room Description

"Cerberus Research" is a realistic employee portal for a state-of-the-art Cybersecurity Research & Development lab, intentionally built with vulnerabilities so learners can safely practice offensive and defensive web security skills. This hands-on challenge TryHackMe room teaches how attackers find, exploit, and chain common web flaws — and methods in which defenders should mitigate them.

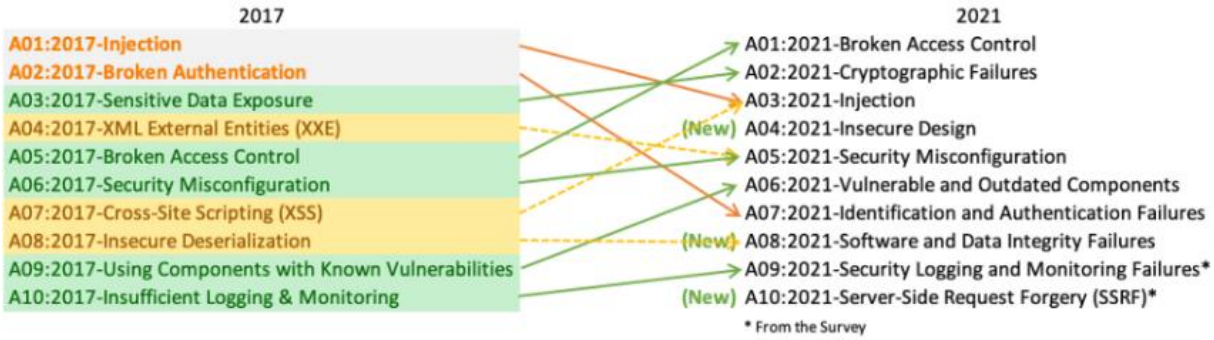
This room focuses on four classic web vulnerabilities, each mapped out to the following categories in OWASP Top 10:

- A01:2021 - Broken Access Control: IDOR (Insecure Direct Object Reference)
- A05:2021 – Security Misconfiguration: Path Traversal
- A03:2021 – Injection: Reflected XSS
- A01:2021 – Broken Access Control: CSRF (Cross-Site Request Forgery)
- A05:2021 – Security Misconfiguration: LFI (Local File Inclusion)

The mission of this room is to perform reconnaissance, authenticate, and exploit common web vulnerabilities to extract several flags.

1.3 What is OWASP top 10?

The OWASP Top 10 is a globally recognized standard for identifying the most critical security risks to web applications. Maintained by the Open Web Application Security Project, it behaves as a introductory awareness document for developers and security professionals.



2. Challenge Walkthroughs

2.1 IDOR (Insecure Direct Object Reference) Challenge

This IDOR challenge tests whether players can discover and exploit an exposed user identifier to access another user's profile (admin) and retrieve a hidden flag.

What is IDOR?

This vulnerability occurs when an application exposes internal object references (like user IDs, file names, or order numbers) without proper access control. Attackers can operate these references to access or modify data belonging to other users.

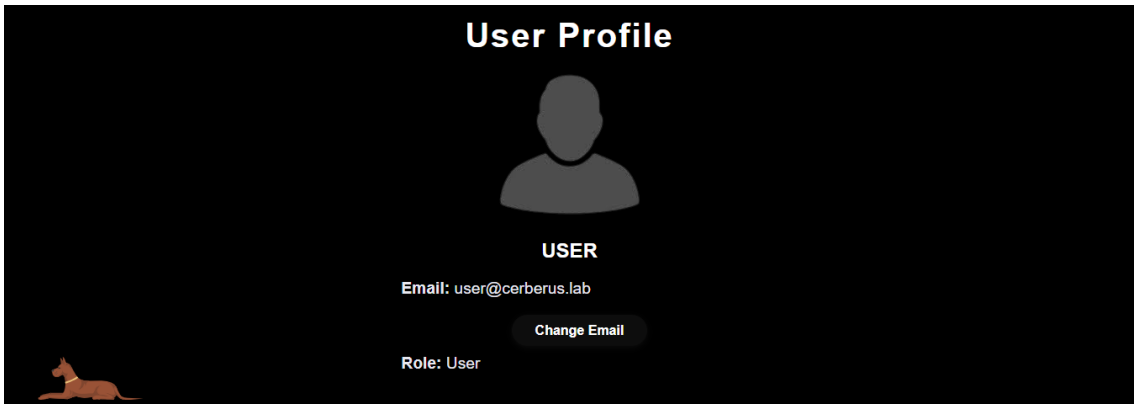
How does it work?

If a web application depends only on user-supplied input (for example, user_id=1') without checking permissions, an attacker can change the ID to another value and access resources they shouldn't in the application.

Steps to complete the challenge:

The goal of this challenge is to access the admin's profile and get the flag, while being logged in as a user.

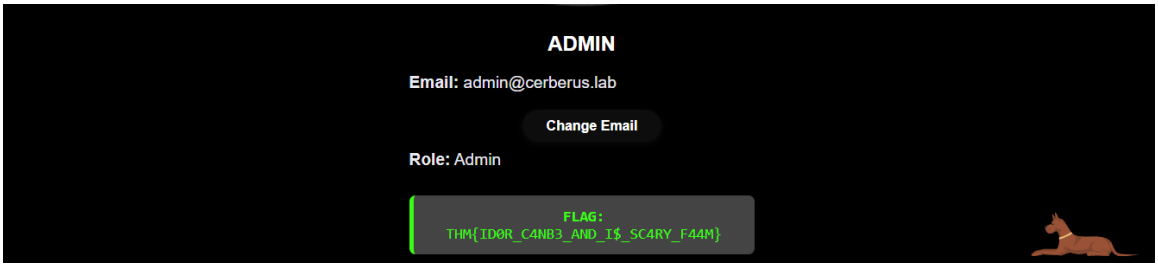
- Login to the website using the credentials (*user/ user321*)
- Now, you are currently logged in as a user in Cerberus research.
- Next, Navigate to the *Profile Page* using the Navbar.



- If you observe the URL of the Profile page, you'll notice that the user_id is directly visible as user:
    - *https://cerberusresearch-h8b6budqgkh2ajbu.southeastasia-01.azurewebsites.net/profile?user_id=**user***

- Now, change the user_id value to admin:
    - *https://cerberusresearch-h8b6budqgkh2ajbu.southeastasia-01.azurewebsites.net/profile?user_id=**admin***

- The Profile Page will change to the profile of the admin, revealing the flag.



Flag Explanation:

The flag is a secret token stored on the admin user's profile page. Because the application exposes a direct object reference (in this case, the user_id) in the URL and does not verify that the logged-in user is allowed to view the referenced account. So changing user_id=user to user_id=admin returns the admin's profile and reveals the flag.

The root cause of this vulnerability is because the server trusts the client-supplied identifier (user_id) and uses it to fetch and reload profile data without enforcing authorization checks.

- Challenge Flag: *THM{ID0R_C4NB3_AND_I$_SC4RY_F44M}*

Mitigation strategies for IDOR vulnerability:

Listed are some simple but effective methods to keep web applications from being vulnerable to IDOR attacks:

- Enforce backend authorization for every request.
- Use non-guessable IDs like UUIDs instead of sequential numbers.
- Never trust client-side logic for access control.

2.2 Path Traversal Challenge

This Path Traversal challenge tests whether players can manipulate file path input (e.g., file=../../etc/passwd) to traverse the filesystem of the web application and read sensitive files that contain the flag.

What is Path Traversal?

This is a vulnerability where attackers change the file paths in user input to access files or directories outside the intended scope. The goal of such attacks is to access sensitive files, source code, configuration files, or other restricted resources within the web application.

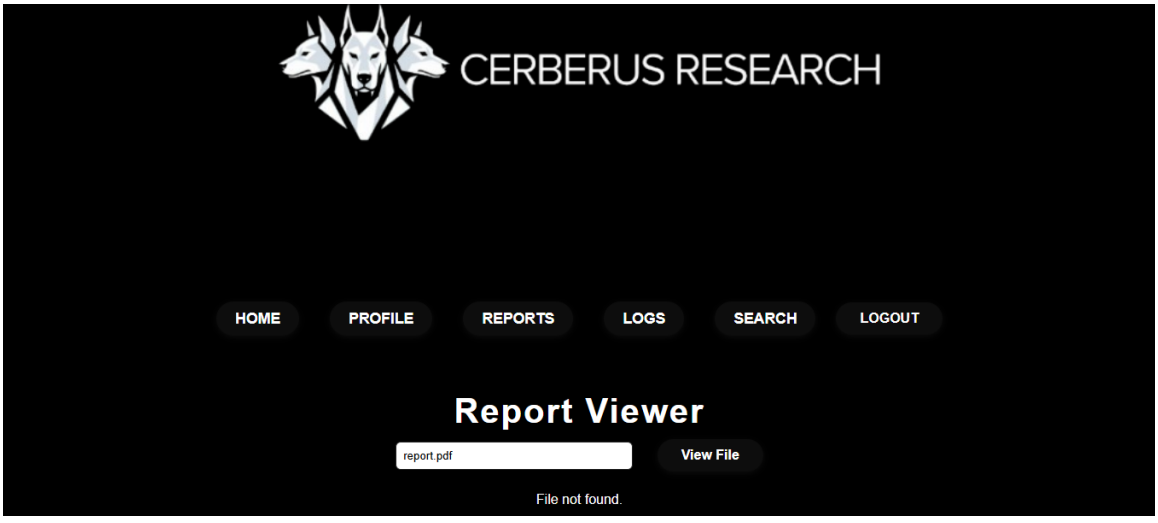
How does it work?

The user input is directly concatenated into a file path without proper validation or restriction and then the attacker modifies the resource path to escape the allowed directory or access unintended files within the web application.

Steps to complete the challenge:

The goal of this challenge is to read an encoded secret in a file outside the intended directory, to get the key to finding the flag.

- Login to the website using the credentials (*user/ user321*)
- Now, you are currently logged in as a user in Cerberus research.
- Next, navigate to the *Reports Page* using the Navbar.



- The flag is hidden in a .txt file named *secret*, which is in a directory called *files*, that is not visible in the front-end.

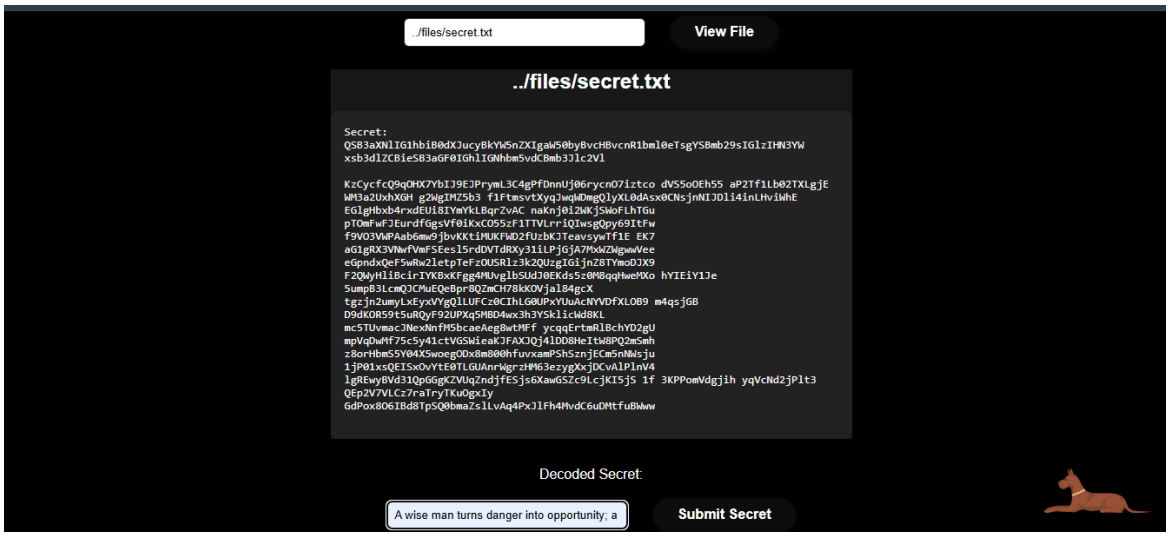
- Try viewing a normal report (e.g., report1.txt, report2.txt...etc) in the provided search bar or in the URL: *https://cerberusresearch-h8b6budqgkh2ajbu.southeastasia-01.azurewebsites.net/download?file=report1.txt*

- After entering a report name, the contents of the report are displayed.

- Now, instead of report names, try entering *secret.txt* to see if anything shows up.

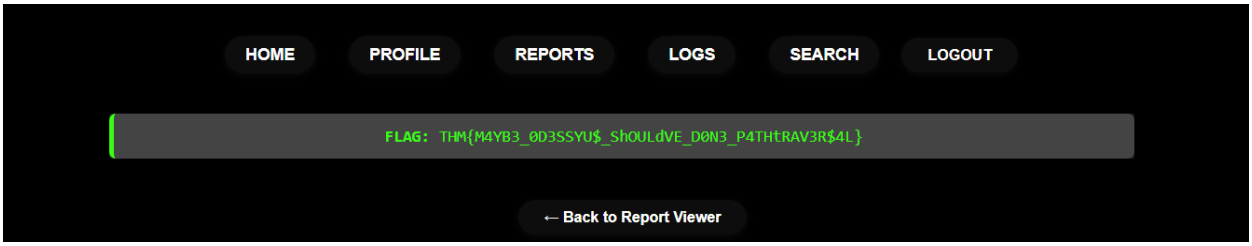
- Since no files are found in that directory, attempt to path traversal using the URL or the search bar provided

- Append the following to the end of the URL (just after =) to obtain the contents of the secret file: *../files/secret.txt*



- Now, you will be able to see the secret in the *.txt* along with some other content.

- Notice how the secret is encoded? decode that to get the secret phrase and enter it into the submission bar to obtain the flag. (Quick tip, try CyberChef to decode this Base64 encoded secret)



Flag Explanation:

The flag is a secret token stored in a sensitive file on the server's filesystem (in this case, ../files/secret.txt). Because the application accepts a file path from the client (e.g., file=report1.txt) and concatenates it into a filesystem lookup without proper validation or normalization, an attacker can supply traversal sequences (like ../../) to navigate outside the intended directory and read arbitrary files.

So, changing file=reports/report1.txt to file=../files/secret.txt returns the contents of the system file and reveals the flag. The main cause of this vulnerability is that the server trusts client-supplied path input and fails to canonicalize/validate the path or enforce access restrictions before reading files.

- Challenge Flag: *THM{M4YB3_0D3SSYU$_ShOULdVE_D0N3_P4THtRAV3R$4L}*

Mitigation strategies for Path Traversal vulnerability:

Listed are some simple but effective methods to keep web applications from being vulnerable to Path traversal attacks:

- Never trust user-supplied file paths.
- Use whitelisting (predefined allowed filenames).
- Resolve to canonical path before access checks, then verify it stays within the intended directory.
- Use OS APIs or libraries that enforce sandboxed/jail paths.
- Encode/escape output to prevent directory climbing.

2.3 Reflected XSS Challenge

This Reflected XSS challenge tests whether players can discover and exploit input that is reflected unsafely back into a page (like a search or error parameter) to execute JavaScript in the victim's browser and steal a flag or session.

What is Reflected XSS?

This is a type of vulnerability where an attacker injects malicious JavaScript code into a website's request, and that code is immediately reflected in the response. The vulnerability occurs when a website does not properly sanitize user input, allowing it to be executed in a victim's browser.

XSS vulnerabilities occur when an application includes untrusted data on a web page without proper validation or escaping, allowing attackers to execute malicious code.

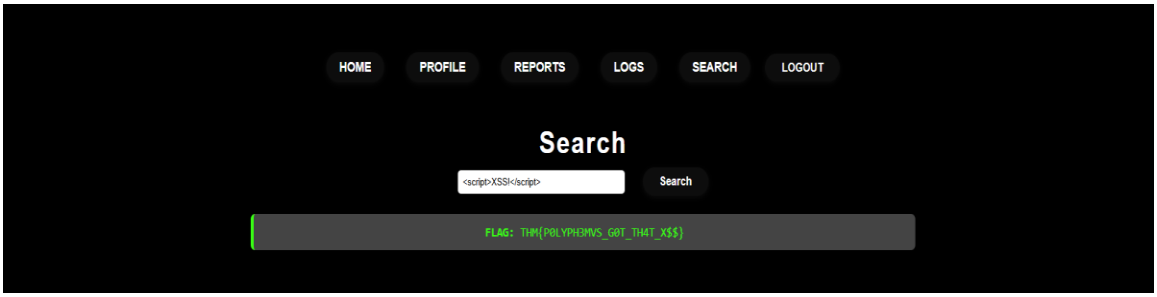
How does it work?

XSS works by injecting malicious scripts into a trusted website or web application. These scripts are executed by the victim's browser when they visit the compromised site. The attack typically exploits the browser's trust in the content of the website, leveraging it to perform actions on behalf of the user, such as stealing session cookies, redirecting users, or altering page content.

Steps to complete the challenge:

The goal here is to trigger a simple reflected XSS attack using a malicious payload to get the flag.

- Login to the website using the credentials (*user/ user321*)
- Now, you are currently logged in as a user in Cerberus research.
- Next, navigate to the *Search Page* using the Navbar.
- Now deliver a simple Javascript payload to the submission bar to see Refleced XSS in action: *<script>XSS!</script>*



Flag Explanation:

The flag is a secret token present on a page that renders user-supplied content. Because the application takes input from the client and injects it directly into the HTML response without proper sanitization, an attacker can create a URL containing malicious JavaScript which will execute in the victim's browser when they open it.

Traveling to the crafted URL causes the injected script to run in the victim's session and expose the flag back to the attacker, revealing the secret. The primary cause of this vulnerability is that the server trusts and reflects client input into the page output without validating or encoding it for the HTML/JavaScript context.

- Challenge Flag: *THM{P0LYPH3MVS_G0T_TH4T_X$$}*

Mitigation strategies for Reflected XSS vulnerability:

Listed are some simple but effective methods to keep web applications from being vulnerable to Reflected XSS attacks:

- Input Validation: Sanitize and validate user inputs.
- Output Encoding: Encode special characters to prevent them from being interpreted as code.
- Content Security Policy (CSP) Header: Restrict script sources.
- Use Secure Frameworks: Use frameworks that handle XSS prevention automatically

2.4 CSRF (Cross Site Request Forgery) Challenge

This CSRF challenge tests whether players can implement a simple cross-site request that forces an authenticated victim to perform an action (e.g., change an email or reveal account data) which results in flag exposure.

What is CSRF?

Cross-Site Request Forgery (CSRF) is a type of attack where the attacker tricks an authenticated user's browser into making an unwanted request to a web application on which the user is authenticated.

CSRF attacks rely on the fact that web browsers automatically include authentication tokens (like cookies) with requests. The attacker leverages this by creating a malicious request that executes actions the user did not intend. If the server does not properly validate the source of the request (i.e., no verification methods used), the attacker can execute unauthorized actions.

How does it work?

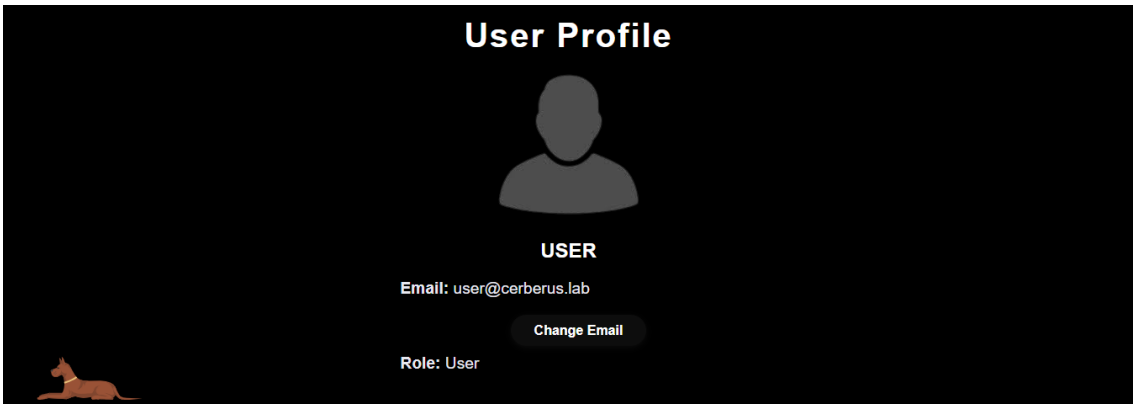
A CSRF exploit usually involves embedding the malicious payload in a place where the target user will interact with it:

- Malicious Links: Send a link containing a malicious payload (often embedded in a URL).
- Phishing Emails: Send an email with a link that automatically submits a CSRF payload when clicked.
- Malicious Web Pages: Place the CSRF exploit on a third-party website that the user trusts, causing them to unknowingly submit a request when they visit.

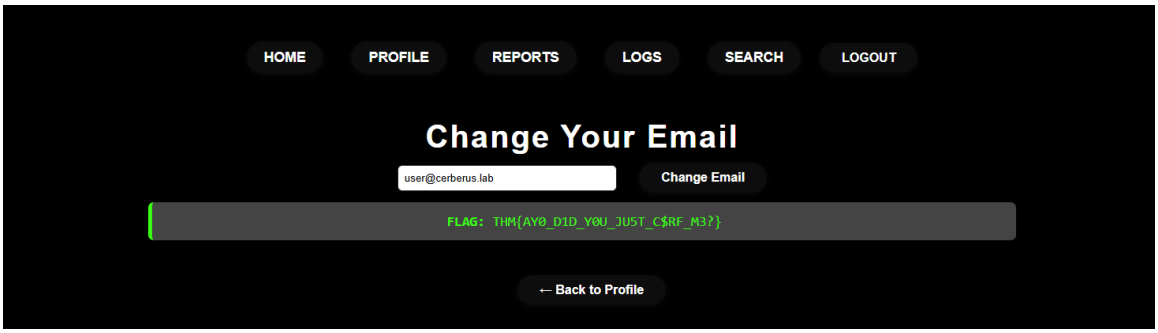
Steps to complete the challenge:

The goal here is to change the mail of the user and get the flag.

- Login to the website using the credentials (*user/ user321*)
- Now, you are currently logged in as a user in Cerberus research.
- Next, navigate to the *Profile Page* using the Navbar.



- You will notice that just below the email of the user there is a 'change mail' option.
- Click that and enter a different email to obtain the flag.
- Please take note that here you act as an attacker on a victim's valid session; In real life situations, this works because the app accepts POSTs without CSRF tokens, and the browser attaches the victim's session cookie.



Flag Explanation:

The flag is a secret token that can be exposed when an authenticated user is tricked into performing a state-changing action that the application accepts without verifying request intent. Because the application relies purely on the user's session cookie and does not enforce an anti-CSRF token or same-site protection, an attacker can craft a malicious page or hidden form that issues a request which the victim's browser will execute while logged in.

The primary cause of this vulnerability is that the server trusts authenticated requests without validating that the request was intentionally made by the user (missing CSRF protections such as synchronizer tokens, SameSite cookie settings, or origin checks).

- Challenge Flag: *THM{AY0_D1D_Y0U_JU5T_C$RF_M3?}*

Mitigation strategies for CSRF:

Listed are some simple but effective methods to keep web applications from being vulnerable to CSRF attacks:

- Input Validation: Sanitize and validate user inputs.
- Output Encoding: Encode special characters to prevent them from being interpreted as code.
- Content Security Policy (CSP) Header: Restrict script sources.
- Use Secure Frameworks: Use frameworks that handle XSS prevention automatically

2.5 LFI (Local File Inclusion) Challenge

This LFI challenge tests whether players can abuse a local file inclusion to execute server-side template code or read sensitive files, allowing flag disclosure or remote code execution.

What is LFI?

LFI (Local File Inclusion) occurs when an application includes or renders files based on user input without proper validation. Depending on how the content is handled, LFI can lead to information disclosure or even code execution (especially if the file gets interpreted as template code).

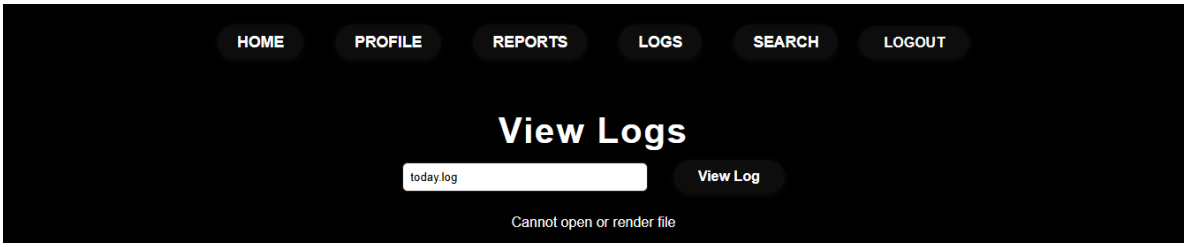
How does it work?

User-supplied path or filename is used by the app to read and *include* content in server-side rendering (or to execute template evaluation). Attackers can use traversal to include sensitive files (configs, passwd, logs) or place file content that triggers template evaluation (SSTI).

Steps to complete the challenge:

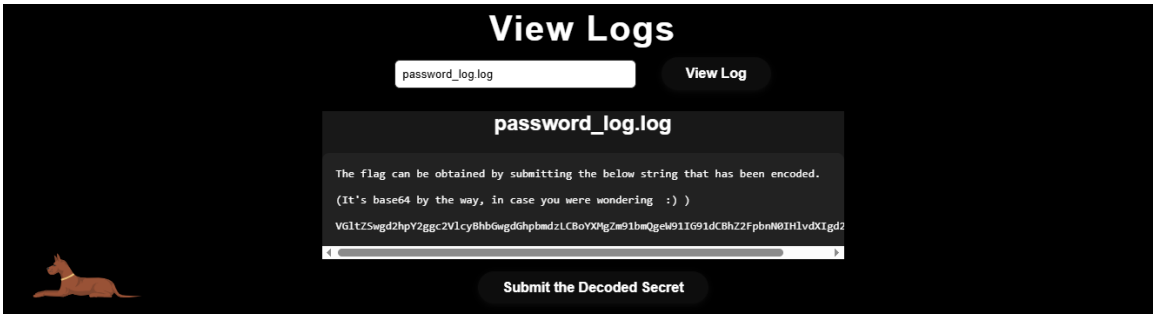
The goal here is to change the mail of the user and get the flag.

- Login to the website using the credentials (*user/ user321*)
- Now, you are currently logged in as a user in Cerberus research.
- Next, navigate to the *Logs Page* using the Navbar.

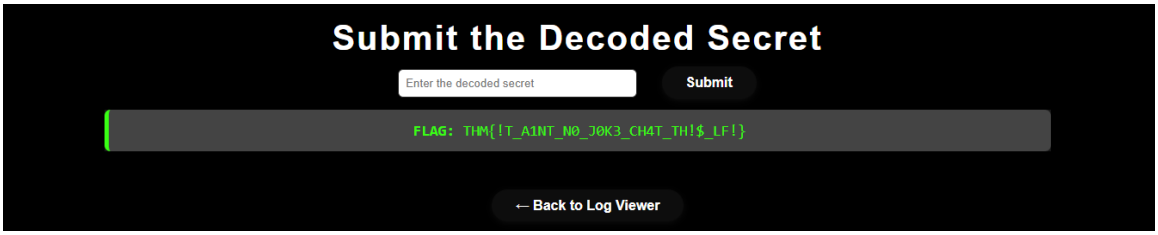


- The secret is stored in a log file named ‘password_log.log’.
- So, enter the file name into the search bar or append it to the web pages URL.
- You will see that in this challenge the content of the log file displayed and that the secret is encoded as well.

- Decode the encoded secret (don't worry it's encoded in Base64; same as the Path traversal challenge) and submit it to the submission bar to find the flag.



- Mind you, this is a very simple LFI challenge designed to show the difference between LFI and path traversal.
    - LFI lets you include and view files from a specific directory (here, the logs folder), but you cannot use **../** or path traversal to escape the folder.
    - In this lab, you can only view files that exist in the logs directory.



Flag Explanation:

The flag is a secret token stored in a file on the server's filesystem (for example, a config or uploads directory). Because the application accepts a file path or template name from the client (e.g., page=home.html or template=report) and directly includes or renders that file without validating or restricting the input.

Requesting the crafted path returns the file contents or causes the template engine to expose secrets, revealing the flag. The primary cause of this vulnerability is that the server trusts client-supplied file identifiers and fails to canonicalize, validate, or restrict which files may be read or rendered.

- Challenge Flag: *THM{!T_A1NT_N0_J0K3_CH4T_TH!$_LF!}*

Mitigation strategies for LFI:

Listed are some simple but effective methods to keep web applications from being vulnerable to LFI attacks:

- Do not render user-supplied files as templates. Treat included files as plain text and escape before rendering.
- Canonicalize and validate file paths; use allowlists and map IDs to files.
- Disable uploading of files that could be interpreted as code and sanitize file contents.

A Quick Comparison: Path Traversal & LFI:

Are they the same?

No, but they overlap as they both abuse file-handling, but they're different in intent and impact. This lab's LFI challenge is a weak variant to show the conceptual gap. The differences are listed in simple point forms below.

Path Traversal

- The goal is to read files outside allowed directories.
- The effect of the attack is that the file content is returned as-is.
- Example: ../../files/secret.txt.

LFI (Local File Inclusion)

- The goal here is to include a file into server processing, as a template or script.
- The effect of the attack is that files are executed or rendered, can reveal extra info or even allow code execution.
- Example: including a log with {{ 7*7 }} rendered by render_template_string().

Impact:

- Path Traversal mostly leads to read-only disclosure.
- LFI leads to disclosure plus potential code execution.

3. Reflections on Room Creation

3.1 Design Process and Decision Making

Creating "Cerberus Research" was driven by the goal of providing players and learners with realistic, hands-on experience in web security testing. The design process began with a simple idea of a hypothetical vulnerable cybersecurity R&D lab, and it later came to life with careful consideration of which vulnerabilities would provide the most educational value while maintaining practical relevance to real-world scenarios as well as the OWASP Top 10.

Equally important was preserving the excitement and narrative of CTFs: this room was intentionally built to be a story-driven, interactive experience that stays true to CTF philosophy of immersive challenges and clear learning objectives so learners can practice concrete skills while staying engaged.

Vulnerability Selection Basis:

- **IDOR and CSRF** were chosen to demonstrate broken access control mechanisms (OWASP A01:2021), as these are commonly found in enterprise applications.
- **Path Traversal and LFI** were included to showcase security misconfigurations (OWASP A05:2021) that often result from insufficient input validation.
- **Reflected XSS** was selected to represent injection flaws (OWASP A03:2021), which remain one of the most prevalent vulnerability classes.

3.2 Technical Implementation Challenges

During the room creation process, several technical challenges were encountered:

- The most significant challenge was creating vulnerabilities that were realistic enough to reflect real-world scenarios while remaining accessible to learners of varying skill levels, as well as implementing a progressive difficulty structure where each vulnerability builds upon previous knowledge.

- Implementing proper access controls and directory restrictions to ensure learners must use the intended exploitation methods, while still maintaining the realistic nature of the vulnerabilities.

- Creating flags that felt integrated into the storyline rather than arbitrary strings required careful planning. Each flag needed to support the learning objectives while maintaining the engaging experience.

- Realtime learning and implementation of programming languages to create the vulnerable web application and managing the IDE proved tedious and time-consuming.

3.3 Learning Outcomes and Personal Development

This TryHackMe room creation significantly enhanced my understanding of what web security is from both defensive and offensive perspectives:

Technical Learning Outcomes:

- Gained deeper insight into how seemingly minor implementation decisions can create major security vulnerabilities.
- Developed appreciation for the interconnected nature of web security issues and how vulnerabilities can be chained together.
- Enhanced understanding of the OWASP Top 10 framework and its practical application in vulnerability assessment.

Educational Design Outcomes:

- Learned to balance challenge difficulty with accessibility for players.
- Developed skills in creating engaging, scenario-based learning experiences.
- Improved ability to explain complex technical concepts in clear, step-by-step instructions.
- The process of creating intentionally vulnerable applications provided valuable insights into common developer mistakes and security oversights as well.