# Try Hack Me Room - Walkthrough

Lakshan Bandara

# Table of Contents

# Overview of the Challenge

## Room Information

- Room Name: SecureShop - Web Security Challenge
- Room URL: https://tryhackme.com/jr/secureshop79
- Vulnerable Site: http://secureshop.infinityfreeapp.com/
- Difficulty Level: Beginner
- Number of Challenges: 4
- Total Flags: 4

## Room Description

**SecureShop** is a vulnerable e-commerce web application intentionally designed with security flaws for educational purposes. The application simulates a small online store selling electronics and gadgets. Players must identify and exploit four different web vulnerabilities to capture hidden flags.

## Learning Objectives

By completing this room, participants will:

- Understand common web application vulnerabilities from the OWASP Top 10.
- Learn manual exploitation techniques without automated scanners.
- Practice using web security testing tools.
- Develop skills in vulnerability identification and exploitation.
- Understand proper remediation techniques.

## Challenges Included (Brief)

1. **SQL Injection (A03:2021 - Injection)** - Unparameterized queries allow authentication bypass or data disclosure.
2. **Broken Access Control / IDOR (A01:2021)** - Direct object identifiers (e.g., profile.php?id=) are used without server-side authorization checks.
3. **Cross-Site Scripting - Stored XSS (A03:2021)** - Unsanitized user reviews/comments are rendered, enabling script execution in other users' browsers.
4. **Sensitive Data Exposure (A02:2021 / Cryptographic Failures)** - Publicly accessible backup files reveal database dumps, credentials, and configuration data.

## Technology Stack

- **Backend:** PHP
- **Database:** MySQL
- **Web Server:** Apache
- **Frontend:** HTML, CSS, JavaScript
- **Hosting:** InfinityFree (Free Hosting Platform)

## Vulnerabilities Mapped to OWASP Top 10 (2021)

| Challenge | Vulnerability Type | OWASP Category |
|---|---|---|
| 1 | SQL Injection | A03:2021 - Injection |
| 2 | Broken Access Control (IDOR) | A01:2021 - Broken Access Control |
| 3 | Cross-Site Scripting (XSS) | A03:2021 - Injection |
| 4 | Sensitive Data Exposure | A02:2021 - Cryptographic Failures |

## Lab Setup — Accessing the TryHackMe Room & Vulnerable Site

- Open the TryHackMe room page: https://tryhackme.com/jr/secureshop79 and click **Start** to initialize the lab.
- Once the room is running, open the vulnerable web application in your browser: http://secureshop.infinityfreeapp.com/
- Follow the room tasks and use the lab site to test vulnerabilities and capture flags.

# Challenge 1: SQL Injection

## Overview

- Vulnerability Type: SQL Injection
- OWASP Category: A03:2021 – Injection
- Severity: High
- Location: Login Page (/login.php)
- Flag: FLAG{SQL_1nj3ct10n_1s_d4ng3r0us}

## What is SQL Injection?

SQL Injection is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It occurs when user input is directly concatenated into SQL queries without proper sanitization or parameterization. This vulnerability can allow attackers to:

- Bypass authentication mechanisms.
- Retrieve sensitive data from the database.
- Modify or delete database records.
- Execute administrative operations on the database.

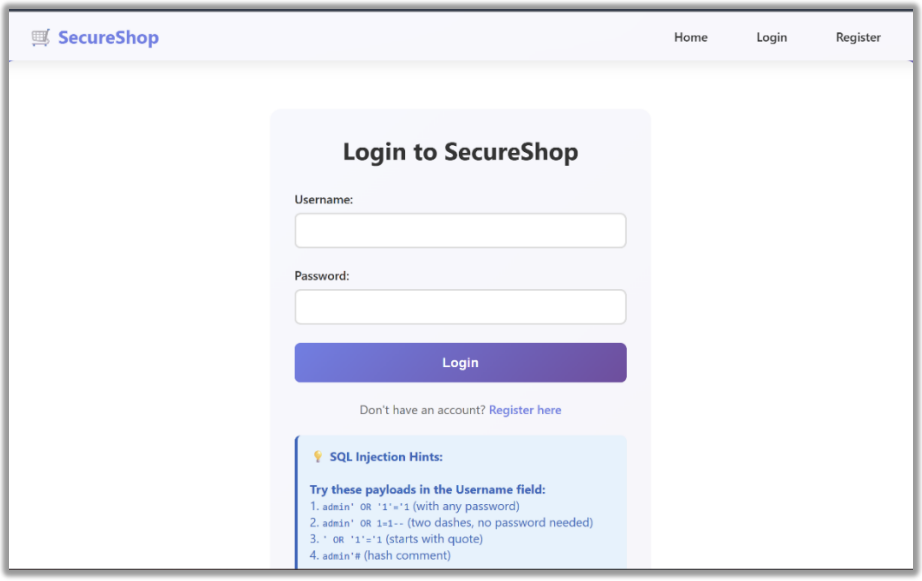
## What is happening here

The login form directly inserts user input into an SQL query without validation or parameterization. By entering a crafted payload like admin' OR '1'='1, the attacker forces the query's condition to always evaluate as true. This tricks the application into authenticating the attacker as a valid user (in this case, admin), revealing sensitive data and the flag.

## Steps to Exploit

### Step 1: Navigate to the Login Page

- Open your web browser and go to:
- http://secureshop.infinityfreeapp.com/login.php
- You will see a standard login form with two input fields:
- Username field
- Password field




## Step 2: Analyze the Vulnerability

The login form is vulnerable because it likely constructs SQL queries using string concatenation. A typical vulnerable query looks like this:

*SELECT * FROM users WHERE username='[USER_INPUT]' AND password='[USER_INPUT]'*

When user input is directly inserted into this query without proper escaping or parameterization, we can manipulate the SQL logic.

## Step 3: Craft the SQL Injection Payload

We will use a basic SQL injection payload that exploits the OR logical operator to always return true.

**Payload:** admin' OR '1'='1

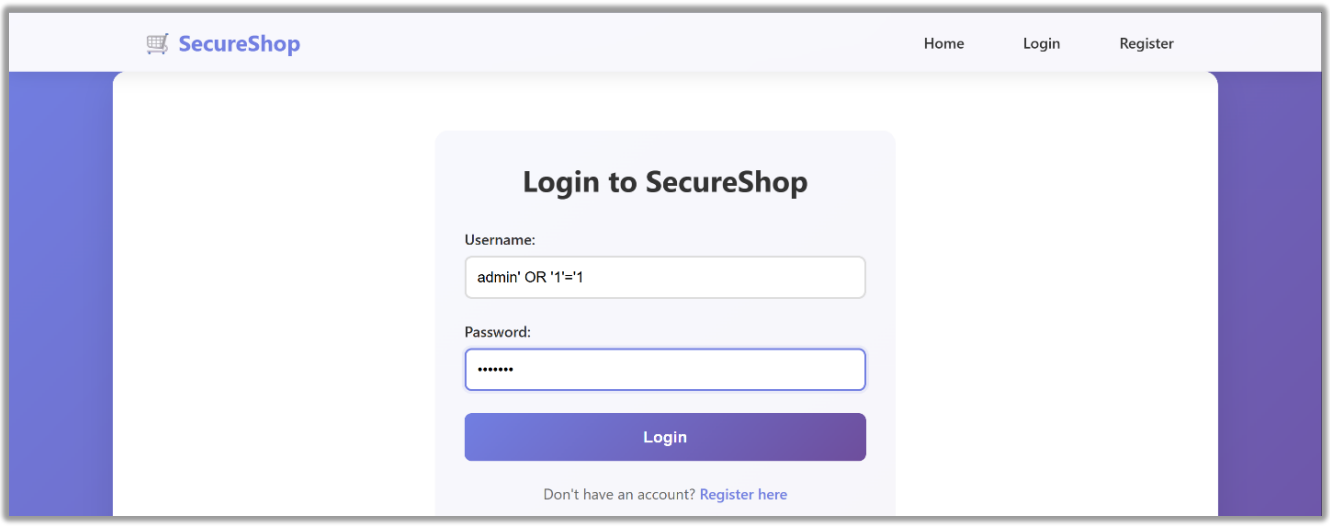
This payload works by:

1. Closing the username string with a single quote (').
2. Adding an OR condition that is always true ('1'='1').
3. Effectively bypassing the password check.

## Step 4: Execute the Attack

In the login form:

1. Username field: Enter admin' OR '1'='1
2. Password field: Enter anything (e.g., test or leave empty)
3. Click the "Login" button




## Step 5: Understand What Happened

When the application processes our input, the SQL query becomes:

SELECT * FROM users WHERE username='admin' OR '1'='1' AND password='test'

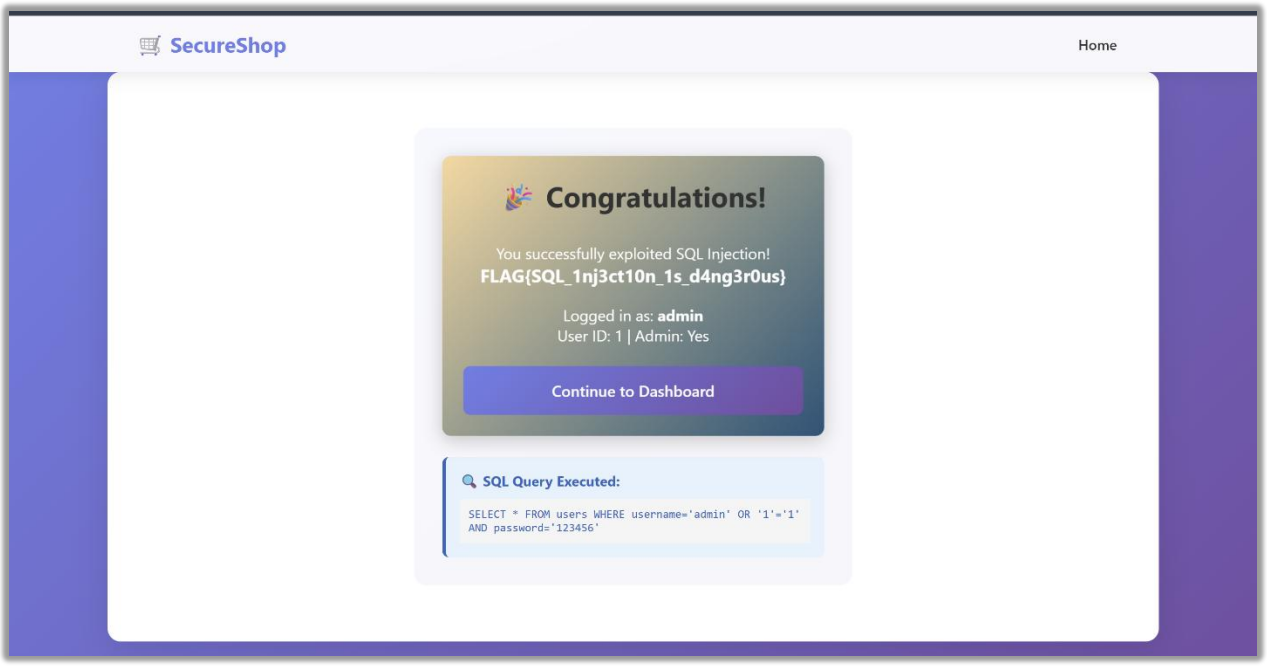
Breaking this down:

- username='admin' - Checks if username is admin.
- OR - Logical OR operator.
- '1'='1' - This is always TRUE.
- The password check is bypassed because the OR condition is satisfied.

Since '1'='1' always evaluates to true, the query returns results regardless of the password, bypassing authentication.

### Step 6: Capture the Flag

Upon successful exploitation, you will be logged into the application as the admin user. The flag will be displayed on the screen:




**Flag Captured:** FLAG{SQL_1nj3ct10n_1s_d4ng3r0us}

## Explanation of the Flag

The flag format FLAG{SQL_1nj3ct10n_1s_d4ng3r0us} emphasizes the critical nature of SQL injection vulnerabilities. It serves as a reminder that:

- SQL injection can completely bypass authentication.
- It's one of the most dangerous web vulnerabilities.
- Proper input validation and parameterized queries are essential.

## Impact Assessment

Severity: High

### Potential Impact:

- Complete authentication bypass.
- Unauthorized access to all user accounts including administrative accounts.
- Full database access (read, modify, delete).
- Potential for data exfiltration.
- Possibility of executing operating system commands (in some configurations).
- Complete compromise of application security.

## Security Measures:

1. Use Prepared Statements: Always use parameterized queries (prepared statements)
2. Input Validation: Validate and sanitize all user inputs.
3. Least Privilege: Database users should have minimal necessary permissions.
4. Password Hashing: Store passwords using strong hashing algorithms (bcrypt, Argon2).
5. Web Application Firewall: Implement WAF rules to detect SQL injection attempts.
6. Error Handling: Don't expose database errors to users.

# Challenge 2: Broken Access Control (IDOR)

## Overview

- Vulnerability Type: Insecure Direct Object Reference (IDOR)
- OWASP Category: A01:2021 - Broken Access Control
- Severity: High
- Location: Profile Page (/profile.php?id=)
- Flag: FLAG{1ns3cur3_D1r3ct_0bj3ct_R3f3r3nc3}

## What is IDOR?

Insecure Direct Object Reference (IDOR) is a type of access control vulnerability that occurs when an application provides direct access to objects based on user-supplied input. The application fails to verify whether the user is authorized to access the requested resource. This allows attackers to bypass authorization and access data belonging to other users by simply modifying parameter values.

## What is happening here

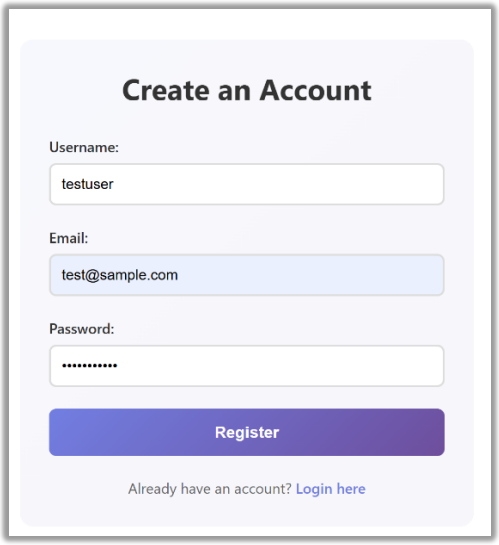
The web application exposes object references through a URL parameter (for example, profile.php?id=2) without verifying whether the logged-in user is allowed to access that resource. By simply changing the ID value, the attacker can view another user's (such as the admin's) private data and retrieve sensitive information, including the flag.

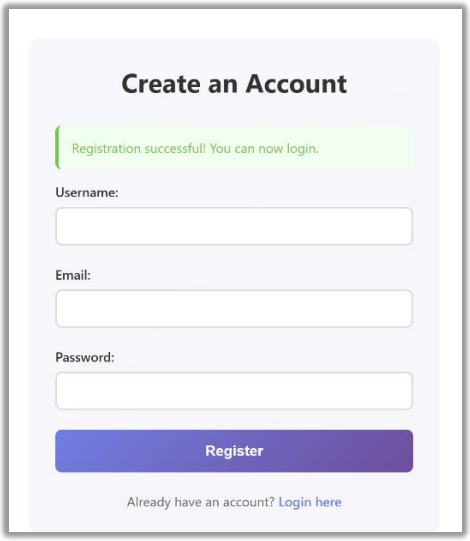
## Steps to Exploit

### Step 1: Create a User Account

First, we need a regular user account to observe the normal behavior:

1. Navigate to: http://secureshop.infinityfreeapp.com/register.php
2. Fill in the registration form:
    - Username: testuser (or any username you prefer)
    - Email: test@example.com
    - Password: password123
3. Click "Register".
4. You should see a success message.



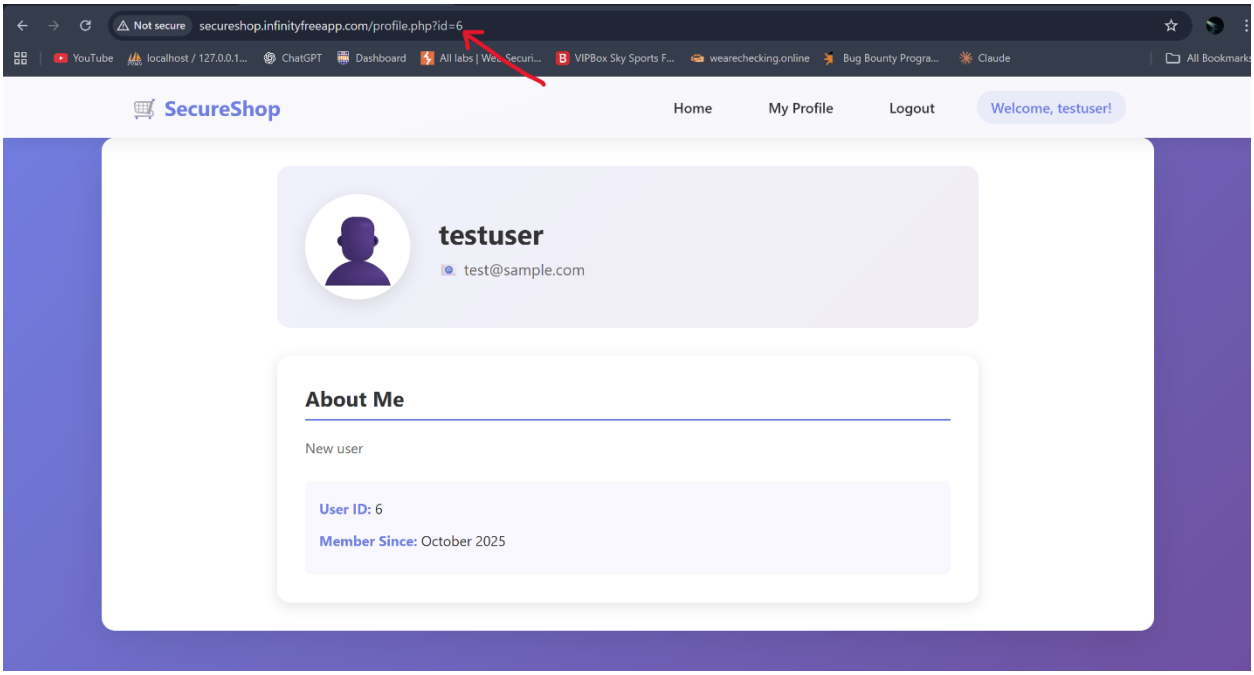

## Step 2: Login with Your Account

1. Go to: http://secureshop.infinityfreeapp.com/login.php
2. Login with your credentials:
    - Username: testuser
    - Password: password123
3. Click "Login"

## Step 3: Access Your Profile

After logging in:

1. Click on "My Profile" in the navigation menu
2. Observe the URL in your browser's address bar
3. You'll see something like: http://secureshop.infinityfreeapp.com/profile.php?id=6




The id parameter in the URL represents your user ID in the database.

## Step 4: Understand the Vulnerability

The application displays profile information based solely on the id parameter in the URL without checking if the currently logged-in user has permission to view that profile. This is the IDOR vulnerability.

## Step 5: Identify the Admin User ID

In most applications, the first user created is typically the administrator, and their user ID is usually 1. We'll test this hypothesis.

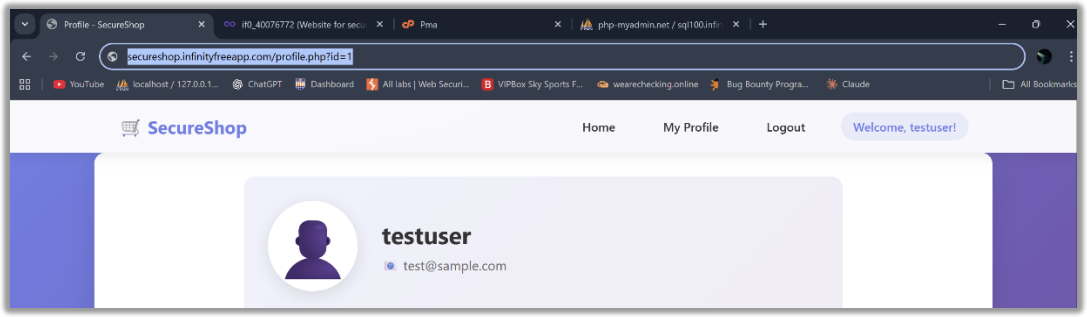
## Step 6: Exploit the IDOR Vulnerability

Manually modify the URL to access the admin profile:

1. In your browser's address bar, change the URL to:

http://secureshop.infinityfreeapp.com/profile.php?id=1

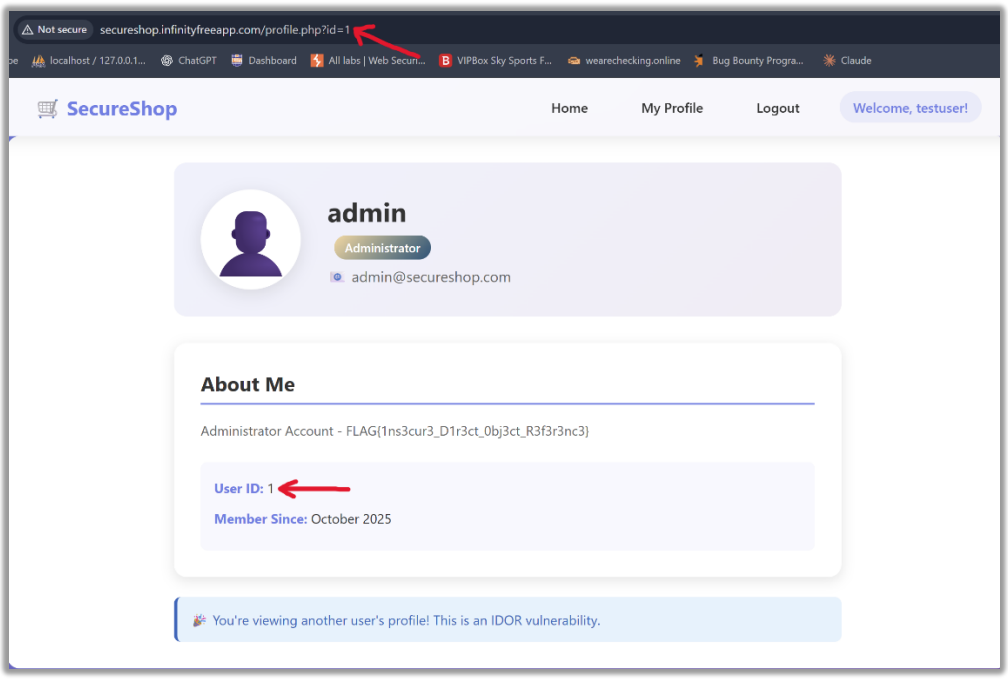
2. Press Enter




## Step 7: Access Unauthorized Data

You should now see the administrator's profile page, even though you're logged in as a regular user. This page displays:

- Admin username
- Admin email address
- Admin status badge
- Admin bio (which contains the flag!)




## Step 8: Capture the Flag

Look in the administrator's bio section. You will find:

**Flag Captured:** FLAG{1ns3cur3_D1r3ct_0bj3ct_R3f3r3nc3}

### Additional Testing

You can also try accessing other user profiles by incrementing the ID:

- profile.php?id=2 - User 2 (john_doe)
- profile.php?id=3 - User 3 (jane_smith)
- profile.php?id=4 - User 4 (bob_wilson)

This demonstrates that ALL user profiles are accessible without proper authorization checks.

## Explanation of the Flag

The flag FLAG{1ns3cur3_D1r3ct_0bj3ct_R3f3r3nc3} directly references the vulnerability name (Insecure Direct Object Reference). It highlights that:

- Object references (like user IDs) shouldn't be directly exposed
- Authorization checks are critical
- Users can access data they shouldn't be able to see

## Impact Assessment

**Severity:** High

### Potential Impact:

- Privacy violations - unauthorized access to user data
- Data leakage - sensitive information exposed
- Compliance issues - GDPR, CCPA violations
- Identity theft potential
- Business logic bypass
- Privilege escalation possibilities

## Security Measures:

1. Access Control Lists (ACL): Implement proper authorization checks
2. Use UUIDs: Instead of sequential IDs, use random UUIDs
3. Session Management: Verify user permissions on every request
4. Indirect Reference Maps: Use temporary random tokens instead of direct IDs
5. Logging: Log all access attempts for monitoring
6. Rate Limiting: Prevent automated enumeration attacks

# Challenge 3: Cross-Site Scripting (XSS)

## Overview

- Vulnerability Type: Stored Cross-Site Scripting (XSS)
- OWASP Category: A03:2021 – Injection
- Severity: High
- Location: Product Review Section (/product.php?id=1)
- Flag: FLAG{XSS_c4n_st34l_c00k13s}

## What is XSS?

Cross-Site Scripting (XSS) is a web security vulnerability that allows attackers to inject malicious scripts into web pages viewed by other users. When the application includes untrusted data in a web page without proper validation or escaping, attackers can execute arbitrary JavaScript code in victims' browsers. This can lead to:

- Session hijacking
- Cookie theft
- Keylogging
- Phishing attacks
- Website defacement
- Malware distribution

## Types of XSS

1. Stored XSS (Persistent) - The malicious script is permanently stored on the server
2. Reflected XSS (Non-Persistent) - The script is reflected off the web server
3. DOM-based XSS - The vulnerability exists in client-side code

This challenge involves Stored XSS because the payload is saved in the database and executed whenever the page is loaded.

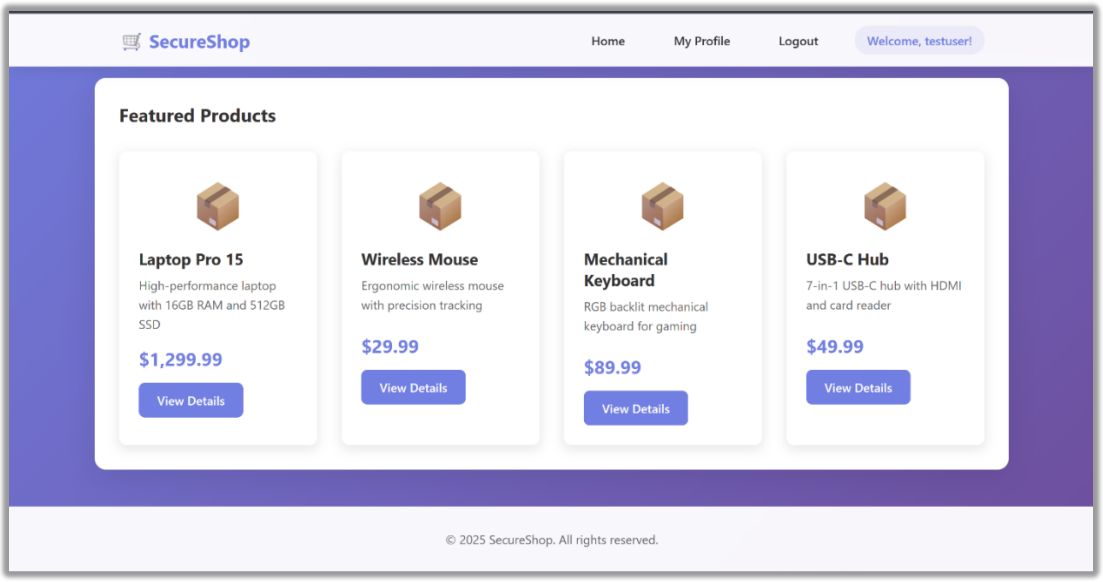
## What is happening here

User input from a comment or review form is saved to the database and displayed on pages without being sanitized or escaped. When an attacker submits a JavaScript payload (like <script>alert('XSS')</script>), it executes automatically for any visitor who views that page. This proves a stored cross-site scripting flaw, which could be used to steal cookies, hijack sessions, or perform actions as other users.

## Steps to Exploit

### Step 1: Navigate to a Product Page

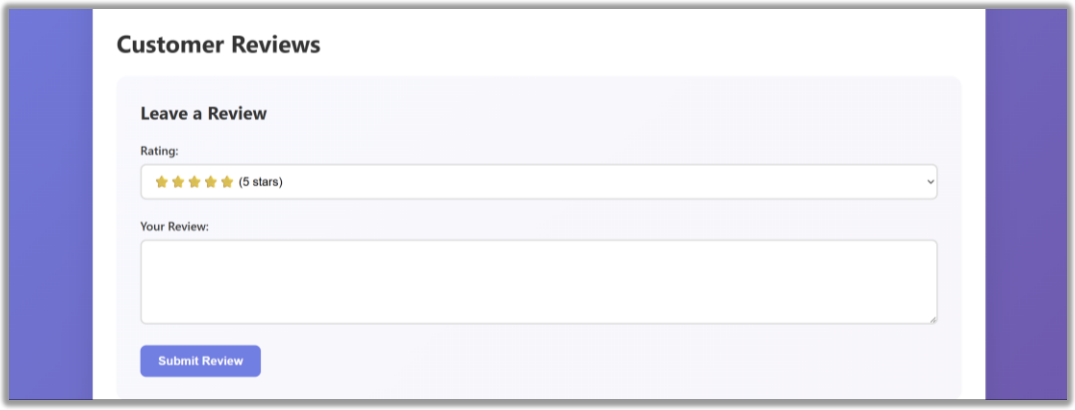
1. Make sure you're logged into the application
2. Go to: http://secureshop.infinityfreeapp.com/product.php?id=1
3. You'll see a product detail page with product information and customer reviews




### Step 2: Locate the Review Section

Scroll down the page until you see:

- "Customer Reviews" heading
- "Leave a Review" section with a form




## Step 3: Test for XSS Vulnerability

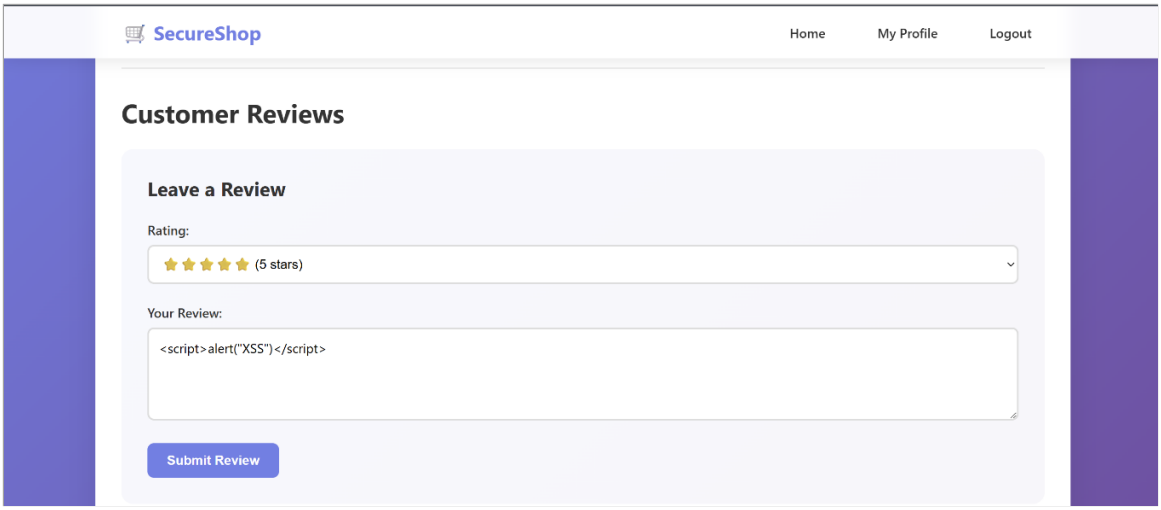
We'll start with a basic XSS payload to test if the application is vulnerable:

Basic XSS Payload:

<script>alert('XSS')</script>

In the review form:

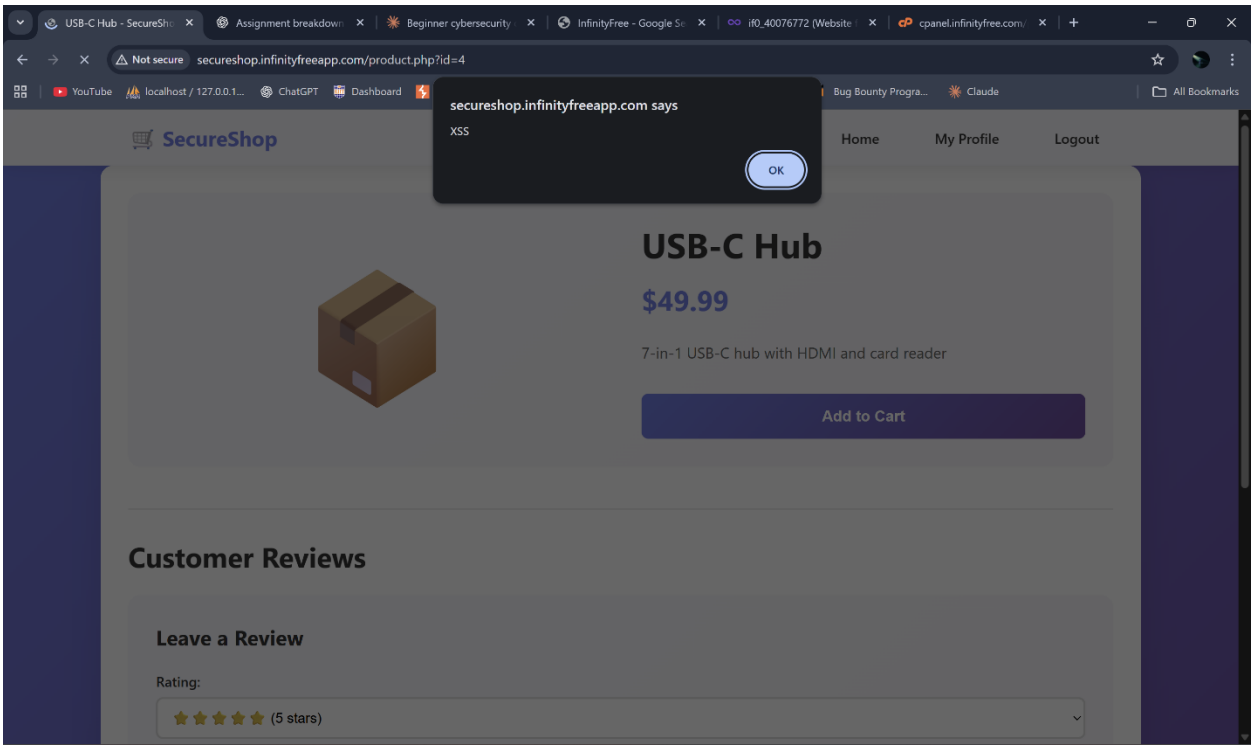
1. Rating: Select any rating (e.g., 5 stars)
2. Your Review: Enter <script>alert("XSS")</script>
3. Click "Submit Review"



## Step 4: Verify XSS Execution

After submitting the review:

1. The page will reload.
2. If the application is vulnerable, you should see an alert box popup with the text "XSS".
3. This confirms that the application is executing JavaScript from user input.



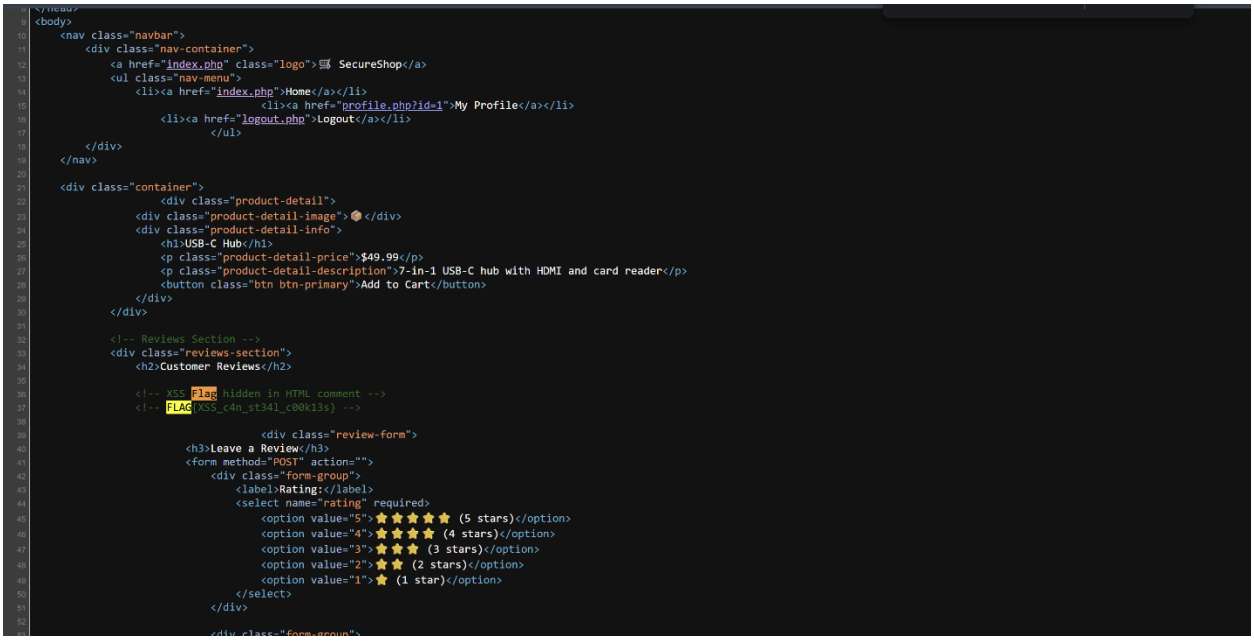

## Step 5: Locate the Flag

The flag for this challenge is hidden in the page's HTML source code. Here's how to find it:

### *Method 1: View Page Source*

1. Right-click anywhere on the page.
2. Select "View Page Source".
3. Search for "FLAG" using Ctrl+F (Windows) or Cmd+F (Mac).
4. You'll find an HTML comment containing the flag:

*<!-- FLAG{XSS_c4n_st34l_c00k13s} -->*




### *Method 2: Developer Tools*

1. Right-click on the page and select "Inspect" or press F12
2. Go to the "Elements" or "Inspector" tab
3. Press Ctrl+F (Windows) or Cmd+F (Mac) to search
4. Search for "FLAG"
5. You'll find the comment with the flag

## Step 6: Capture the Flag

```
<!-- XSS Flag hidden in HTML comment -->
<!-- FLAG{XSS_c4n_st34l_c00k13s} -->
```

**Flag Captured:** FLAG{XSS_c4n_st34l_c00k13s}

## Understanding the Vulnerability

The application is vulnerable because:

1. User input (review text) is stored in the database without sanitization
2. When displaying reviews, the application outputs the text directly into HTML without encoding
3. This allows JavaScript code to be executed in the browser

## Explanation of the Flag

The flag FLAG{XSS_c4n_st34l_c00k13s} emphasizes one of the most dangerous aspects of XSS attacks - the ability to steal session cookies. With stolen cookies, attackers can:

- Hijack user sessions
- Impersonate legitimate users
- Access sensitive accounts without passwords
- Perform actions on behalf of victims

## Impact Assessment

**Severity:** High

### Potential Impact:

- Session Hijacking: Steal authentication tokens and impersonate users
- Credential Theft: Capture keystrokes and login credentials
- Phishing: Display fake login forms to steal passwords
- Malware Distribution: Redirect users to malicious sites
- Website Defacement: Modify page content
- Data Theft: Access sensitive information displayed on the page
- Social Engineering: Manipulate user trust
- Worm Propagation: Self-propagating XSS attacks

### Remediation techniques for XSS

- Escape/encode user input – Properly encode data before outputting in HTML, attributes, JavaScript, or URLs.
- Use safe frameworks/libraries – Rely on templating engines or frameworks (e.g., React, Angular) that auto-escape output.
- Validate and sanitize inputs – Allow only expected formats/characters and use trusted sanitizers (e.g., DOMPurify) if HTML is required.
- Apply Content Security Policy (CSP) – Enforce a strict CSP to block inline scripts and unauthorized sources.

## Challenge 4: Sensitive Data Exposure

### Overview

- Vulnerability Type: Sensitive Data Exposure / Information Disclosure
- OWASP Category: A02:2021 - Cryptographic Failures
- Severity: High
- Location: Backup Directory (/backups/)
- Flag: FLAG{b4ckup_f1l3s_4r3_d4ng3r0us}

### What is Sensitive Data Exposure?

Sensitive Data Exposure occurs when applications fail to adequately protect sensitive information such as:

- Database backups
- Configuration files
- Source code
- API keys and credentials
- Customer data
- Authentication tokens

This vulnerability often results from:

- Leaving backup files in publicly accessible directories
- Enabling directory listing
- Poor access controls
- Insecure file storage practices
- Lack of encryption

## What is happening here

The website stores backup or database dump files inside a publicly accessible directory (such as /backups/). These files can be discovered through directory enumeration or brute forcing. Since they are not protected, anyone can download them and extract critical information like usernames, hashed passwords, and configuration details—resulting in full data disclosure and exposure of the flag.

## Steps to Exploit

### Step 1: Directory Enumeration

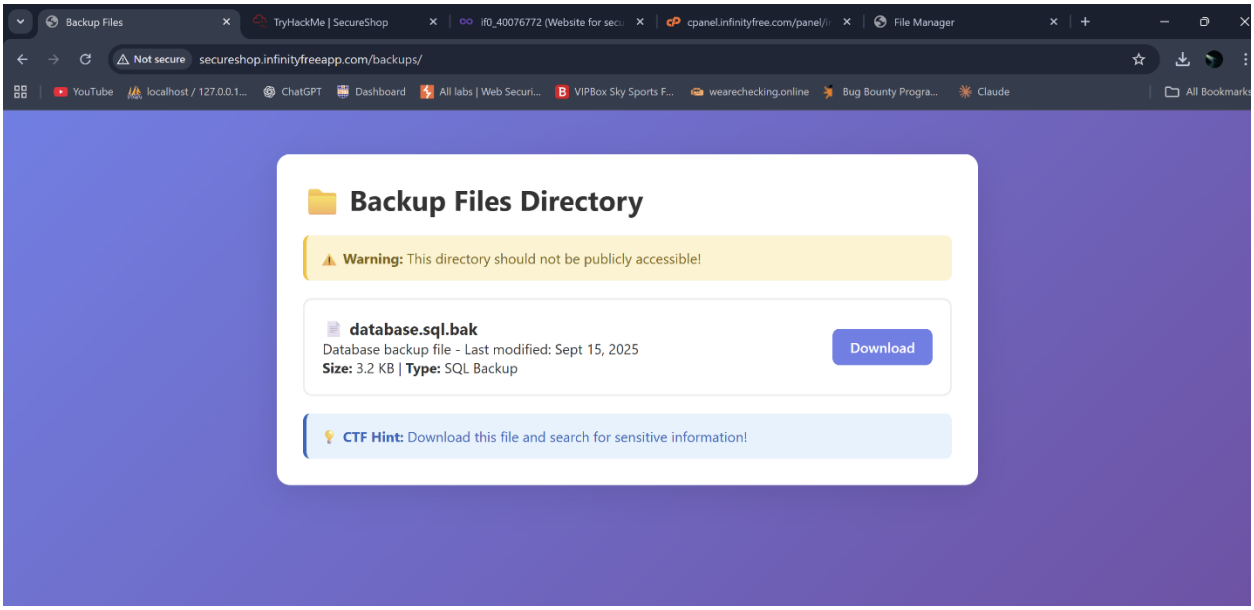
The first step is to discover hidden directories and files. Common backup directory names include:

- /backup/
- /backups/
- /.backup/
- /old/

### Step 2: Attempt to Access Backup Directory

Try accessing the backups directory:

http://secureshop.infinityfreeapp.com/backups/



**What might happen:**

- **Option A:** Directory listing is enabled, showing all files
- **Option B:** An index page lists available files
- **Option C:** 403 Forbidden error (need to guess filenames)

## Step 3: Identify Backup Files

Common backup file extensions to look for:

- .bak
- .backup
- .sql
- .sql.bak
- .old
- .zip
- .tar.gz

## Step 4: Access the Backup File

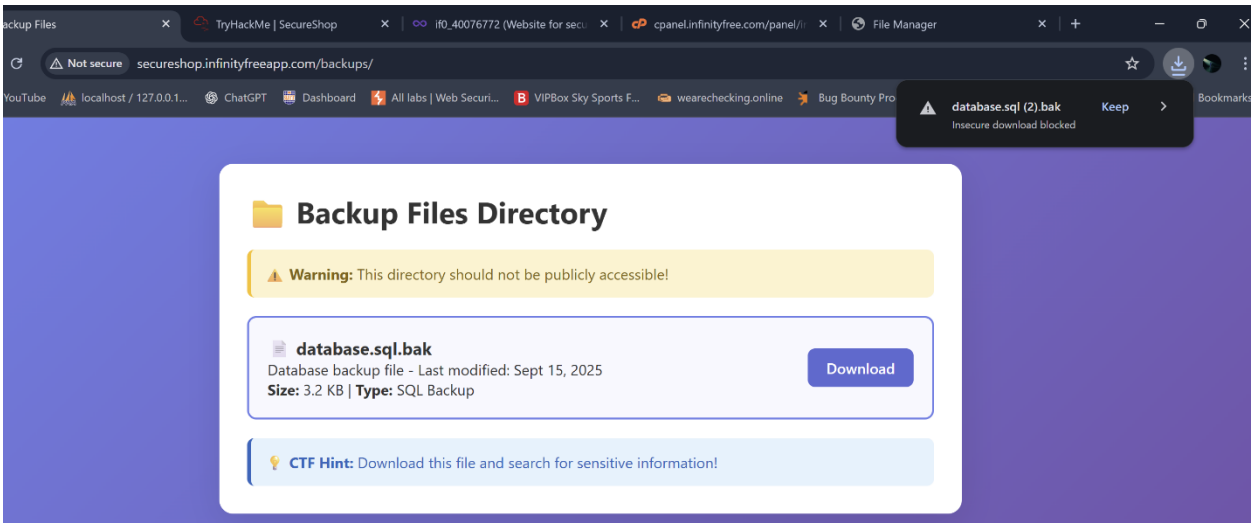
Try accessing the specific backup file:

http://secureshop.infinityfreeapp.com/backups/database.sql.bak

## Step 5: Download or View the Backup File

Once you've found the backup file:

1. Click on it (if directory listing is enabled)
2. Or access it directly via the URL
3. Your browser will either:
    - Display the file content
    - Download it to your computer




## Step 6: Analyze the Backup File

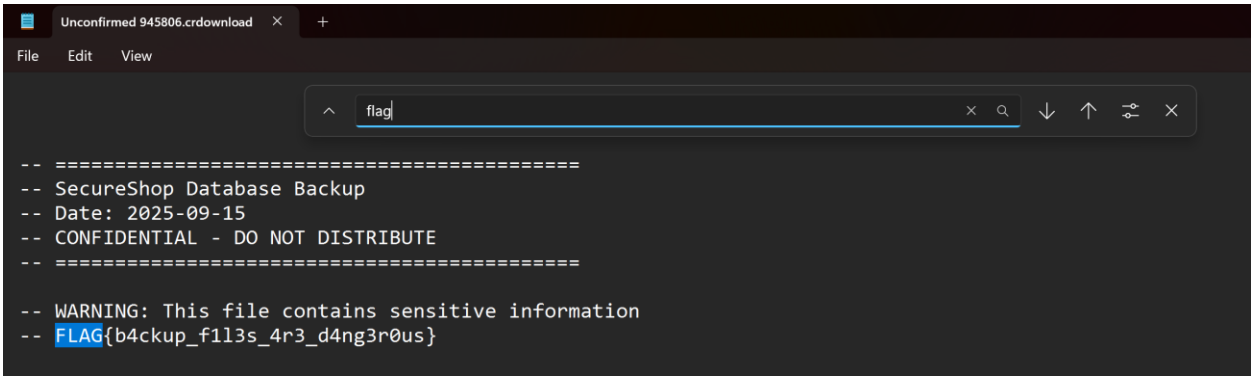
Open the downloaded file with a text editor (Notepad++, VS Code, Sublime Text). The file contains:

- Complete database structure
- All table schemas
- User credentials (in plain text!)
- All data from the database
- And importantly... the flag!

## Step 7: Search for the Flag

In the backup file:

1. Use the search function (Ctrl+F or Cmd+F)
2. Search for "FLAG"
3. You'll find the flag in the file content




**Flag Captured:** FLAG{b4ckup_f1l3s_4r3_d4ng3r0us}

### *What's in the Backup File?*

The database.sql.bak file contains sensitive information including:

- User Credentials:
    - INSERT INTO users VALUES
    - (1,'admin','admin123','admin@secureshop.com',1,'Administrator Account');
- Database Configuration:
    - *DB_HOST: localhost*
    - *DB_USER: root*
    - *DB_PASS: (empty for XAMPP default)*
- API Keys (in comments):
    - *SESSION_SECRET: 7f9a8b6c5d4e3f2a1b0c9d8e7f6a5b4c*
    - *API_KEY: sk_test_4eC39HqLyjWDarjtT1zdp7dc*
- And the Flag:
    - *FLAG{b4ckup_f1l3s_4r3_d4ng3r0us}*

## Explanation of the Flag

The flag FLAG{b4ckup_f1l3s_4r3_d4ng3r0us} directly communicates the risk:

- Backup files often contain complete database dumps
- They expose credentials, user data, and system configuration
- They should NEVER be publicly accessible
- This is a common but critical security mistake

## Impact Assessment

**Severity:** High

### Potential Impact:

- Complete Data Breach: Access to entire database dump.
- Credential Exposure: Usernames and passwords in plain text.
- API Key Leakage: Exposed authentication tokens.
- Configuration Disclosure: Database connection details.
- Business Logic Exposure: Understanding of application structure.
- Privacy Violations: Customer data exposed.
- Compliance Issues: GDPR, CCPA, PCI-DSS violations.
- Reputational Damage: Loss of customer trust.
- Financial Loss: Potential fines and legal action.

### Real-World Examples

1. Facebook (2019): 540 million records exposed in publicly accessible AWS S3 buckets
2. Capital One (2019): 100 million customer records exposed due to misconfigured firewall
3. MongoDB Databases: Thousands of databases left publicly accessible without authentication
4. Git Repositories: Many companies accidentally expose .git folders with complete source code

## Remediation techniques for Sensitive Data Exposure / Information Disclosure

- Use HTTPS/TLS everywhere – Encrypt all data in transit and disable weak protocols (SSL, TLS 1.0/1.1).
- Encrypt sensitive data at rest – Store passwords with strong hashing (bcrypt/Argon2) and use AES for files/databases.
- Avoid exposing sensitive info – Remove stack traces, debug messages, API keys, and version numbers from production responses/logs.
- Restrict access & least privilege – Limit access to sensitive files, databases, and backups; disable directory listing.
- Secure key management – Store encryption keys in a secure vault (e.g., AWS KMS, HashiCorp Vault) instead of hardcoding them.

## Conclusion

This walkthrough documented the exploitation of four critical web vulnerabilities in the SecureShop application:

1. SQL Injection - Allowed complete authentication bypass
2. IDOR - Enabled unauthorized access to user data
3. XSS - Permitted arbitrary code execution in users' browsers
4. Data Exposure - Revealed sensitive backup files with credentials

All four vulnerabilities represent real-world security issues commonly found in web applications and are part of the OWASP Top 10, the industry standard for web application security risks.

## Design Process

The design of the TryHackMe room was focused on creating a realistic and educational web security environment that demonstrated multiple OWASP Top 10 vulnerabilities. I first designed the application structure to resemble a simple e-commerce platform (SecureShop) that users could easily navigate. Each challenge was then mapped to a specific OWASP category, ensuring that every vulnerability represented a real-world attack scenario. The main goal during the design process was to balance *learnability* and *security realism* — making the room engaging for beginners while still technically accurate.
To make the experience authentic, I included flags, detailed instructions, and hints that guided players through discovery, exploitation, and remediation. The use of tools like Burp Suite, OWASP ZAP, and manual testing ensured the environment behaved like a live web application, helping players understand how vulnerabilities appear in production systems.

## Challenges Faced

During the development of this room, I faced several technical and configuration challenges. One of the biggest difficulties was fixing multiple coding and SQL errors while setting up the web application. Creating proper database backup files and restoring them correctly took time, especially when linking the .sql data with the application. Hosting the website on InfinityFree was also a bit long and tedious — certain configurations needed manual adjustments, and handling the remote database connection added more complexity. I also had to troubleshoot issues with file paths, PHP errors, and some XSS payloads not triggering as expected. Despite these challenges, each problem helped me understand how backend systems interact with databases and how small coding mistakes can lead to vulnerabilities.

## Learning Outcomes

This project strengthened my understanding of how insecure coding practices directly lead to exploitable vulnerabilities. I gained practical experience using professional security testing tools and writing structured security documentation. I also learned how to design a CTF-style environment that encourages active learning and critical thinking rather than just following steps. Most importantly, this task improved my ability to identify, exploit, and remediate common web vulnerabilities in a controlled and ethical way. The overall process deepened my appreciation of secure development principles and the importance of applying OWASP guidelines throughout the software lifecycle.