OWASP Top 10
Hands-On Lab with DVWA

By Shubodaya Kumar

Date: 15/07/2025

Email: hnshubodaya@gmail.com

Table of Contents

1 Abstract 4

2 Introduction 5

2.1 What is DVWA? 5

2.2 What is the OWASP Top 10? 5

2.3 Why I Did This Project 6

3 Lab Setup 7

3.1 Tools Used 7

3.2 Network Configuration 7

3.3 DVWA Security Levels 7

3.4 Installing and Configuring DVWA 8

4 Testing OWASP Top 10 Vulnerabilities 9

4.1 Broken Access Control 9

4.2 Cryptographic Failures 12

4.3 Injection 14

4.3.1 SQL Injection 14

4.3.2 Cross-Site Scripting (XSS) 17

4.3.3 Command Line Injection 21

4.4 Insecure Design 25

4.5 Security Misconfiguration 28

4.5.1 XML External Entities (XXE) 28

4.5.2 Security Misconfiguration 29

4.6 Vulnerable and Outdated Components 30

4.7 Identification and Authentication Failures 32

4.8 Software and Data Integrity Failures 33

4.9 Security Logging and Monitoring Failures 34

4.10 Server-Side Request Forgery (SSRF) 35

5 Key Takeaways 37

6 Appendix / References 38

Chapter 1

1 Abstract

This document provides a practical, hands-on guide to understanding and exploiting the OWASP Top 10 web application vulnerabilities using Damn Vulnerable Web Application (DVWA). Each section walks through a specific vulnerability explaining what it is, why it matters, and how it can be exploited in a controlled lab environment. The goal is to simulate real-world attack scenarios to build a deeper understanding of how these vulnerabilities work in practice and how attackers leverage them.
Rather than focusing solely on theory, this guide emphasizes actionable knowledge. It's designed for cybersecurity students, professionals, and enthusiasts looking to strengthen both their offensive and defensive skills. The lab was set up locally using DVWA with varying security levels to demonstrate how exploitation techniques change based on application configuration.

By the end of this guide, readers will have practical experience in identifying, exploiting, and thinking critically about each vulnerability on the OWASP Top 10 list laying a strong foundation for penetration testing, secure coding, and threat detection.

Chapter 2

2 Introduction

In cybersecurity, reading about vulnerabilities only takes you so far. To really understand how attacks work and how to stop them you have to get your hands dirty. That's what led me to build a local lab using **Damn Vulnerable Web Application (DVWA)** and walk through the **OWASP Top 10** vulnerabilities one by one.

2.1 What is DVWA?

Damn Vulnerable Web Application (DVWA) is an intentionally vulnerable web app built with PHP and MySQL. It was designed for training and testing web application security tools and techniques. DVWA lets users practice real-world attacks like SQL Injection or Cross-Site Scripting—without breaking any laws or systems in production. It's simple, lightweight, and includes multiple security levels (Low, Medium, High, Impossible), so you can test the same vulnerability under different levels of protection.

2.2 What is the OWASP Top 10?

The **OWASP Top 10 (2021 edition)** is a standard awareness document published by the Open Worldwide Application Security Project. It ranks the ten most common and critical security risks found in web applications. These include issues like:

1. **Broken Access Control**
   - Occurs when users can act outside their intended permissions.
   - In DVWA: Accessing admin functions as a normal user or manipulating IDs in URLs to view other users' data (IDOR).
2. **Cryptographic Failures** *(Previously: Sensitive Data Exposure)*
   - Insecure handling of data like passwords, credit card info, or session tokens.
   - In DVWA: Clear-text password storage, unencrypted login forms, or weak hashing algorithms.
3. **Injection**
   - Classic example is SQL Injection, where untrusted input is interpreted as code by the backend.
   - In DVWA: Bypassing login with ' OR 1=1 --, dumping database contents via crafted inputs.
4. **Insecure Design**
   - Refers to flaws in the application's architecture, not just bad coding.
   - While harder to simulate directly in DVWA, you can talk about lack of rate limiting, weak logic flows, or missing input validation as examples.
5. **Security Misconfiguration**

- Using default credentials, exposing error messages, unnecessary features, or open ports.
- In DVWA: Leaving DVWA security level at "Low," default Apache settings, unnecessary services enabled.

6. **Vulnerable and Outdated Components**
   - Running libraries, frameworks, or servers with known vulnerabilities.
   - In DVWA: Outdated PHP versions or web server software that have unpatched security flaws.
7. **Identification and Authentication Failures**
   - Includes broken login mechanisms, poor session handling, or lack of multi-factor authentication.
   - In DVWA: Session hijacking, bypassing login screens, or weak password policies.
8. **Software and Data Integrity Failures**
   - Trusting software updates, plugins, or data sources without verification.
   - While DVWA doesn't demo this explicitly, you can simulate scenarios like uploading malicious code or tampering with client-side scripts.
9. **Security Logging and Monitoring Failures**
   - Lack of proper logging, alerting, or incident response.
   - In DVWA: Attacks like XSS or SQLi succeed silently with no detection or audit trail.
10. **Server-Side Request Forgery (SSRF)**
    - Exploiting a server's ability to send requests to internal resources.
    - DVWA doesn't natively support SSRF, but you can mention it as a critical issue in misconfigured APIs or file fetchers.

This list is widely recognized by security professionals, developers, and organizations as a baseline for secure development and testing.

2.3 Why I Did This Project

I wanted to take a **hands-on approach** to learning these vulnerabilities by actively exploiting them. Instead of just studying how these attacks work in theory, I built a local lab, ran attacks myself, and documented the impact step by step. This gave me an attacker's perspective, which is key for anyone working in cybersecurity.

By simulating real-world attacks in a safe environment, I was able to see how small mistakes in web applications can lead to serious breaches. I also got to explore what makes each vulnerability dangerous, how attackers exploit them, and what developers or security teams can do to prevent them.

This guide is meant to be a walk-through for anyone who wants to learn the same way through breaking, testing, and analysing. It's part of a bigger effort to sharpen my skills in ethical hacking, threat detection, and secure application design.

Chapter 3

3 Lab Setup

To explore and exploit the OWASP Top 10 vulnerabilities, I set up a local lab using **DVWA (Damn Vulnerable Web Application)**. This environment gave me the flexibility to simulate real-world attacks in a controlled, legal, and repeatable way.

3.1 Tools Used

Here's what I used in this project:

- **DVWA** – The main vulnerable application
- **XAMPP** – To run Apache and MySQL locally (used to host DVWA)
- **Kali Linux** – As the attacker machine, preloaded with tools like Burp Suite, OWASP ZAP, and others
- **Burp Suite (Community Edition)** – For intercepting and manipulating HTTP requests
- **Firefox Developer Edition** – As the browser for testing and debugging web responses
- **OWASP ZAP (Optional)** – For scanning and analysing the application
- **VMware Workstation / VirtualBox** – To run both Kali Linux and DVWA in isolated VMs

3.2 Network Configuration

- Both the **Kali Linux VM** and the **DVWA VM** were set up on **Host-Only Network** to keep traffic isolated from the internet.
- IP addresses were manually set or fetched via DHCP to make sure both machines were on the same subnet.
- I accessed DVWA from Kali by navigating to the DVWA machine's IP address in the browser.

Example:
http://192.168.56.101/dvwa

3.3 DVWA Security Levels

DVWA comes with four built-in security levels:

- **Low** – No security at all (easy to exploit, good for beginners)
- **Medium** – Some basic protections (e.g., simple input validation)
- **High** – More advanced security mechanisms (e.g., prepared statements, token validation)
- **Impossible** – Patched and secure version (used to test detection/monitoring)

For each vulnerability, I tested across multiple levels to understand how the attack technique needs to change as security improves. Most of the screenshots and payloads shown in this document were captured with the **security level set to Low**, but I've included notes on what changes at higher levels.

3.4 Installing and Configuring DVWA

Here's a quick overview of how I set up DVWA on XAMPP:

1. Download and install **XAMPP** for Windows or Linux.
2. Clone or download DVWA from its GitHub repo.
3. Move the DVWA folder to XAMPP's htdocs directory:
    - Example: C:\xampp\htdocs\dvwa
4. Start **Apache** and **MySQL** services via the XAMPP Control Panel.
5. Open http://localhost/dvwa/setup.php in your browser.
6. Click **"Create / Reset Database"** to initialize DVWA.
7. Set the **security level** from the DVWA interface (DVWA Security tab).
8. Modify config.inc.php if needed (usually for DB credentials).

DVWA default login:

- **Username**: admin
- **Password**: password

Chapter 4

4 Testing OWASP Top 10 Vulnerabilities

This section documents the hands-on testing I performed against each of the OWASP Top 10 vulnerabilities using DVWA. For every vulnerability, I've explained what it is, how it can be exploited, what impact it has on a web application, and how it should be mitigated. Each test includes real payloads, step-by-step walkthroughs, and screenshots from my lab environment to show how these attacks work in practice.

4.1 Broken Access Control

Overview

Broken Access Control happens when an application fails to properly enforce user permissions. Attackers can exploit this to access unauthorized pages, data, or functions beyond their role.

Impact

- Access to sensitive or restricted data
- Privilege escalation (e.g., normal user acting as admin)
- Unauthorized actions like deleting or modifying data
- Exposure of internal functionality

Exploitation in DVWA

In DVWA, Broken Access Control can be tested by:

- **URL tampering:** Modifying parameters or URLs to access restricted pages or data.
- **IDOR (Insecure Direct Object Reference):** Changing user IDs in requests to view or manipulate other users' data.
- **Changing DVWA security level without proper authorization.**

For example, at Low security level, changing the user ID parameter in the URL may grant access to other users' profiles or data.

Payloads Used

- Manually changing URL parameters, e.g.:

```
http://dvwa-ip/dvwa/vulnerabilities/exec/?id=2
```

changing id=2 to id=1 to access another user's data.

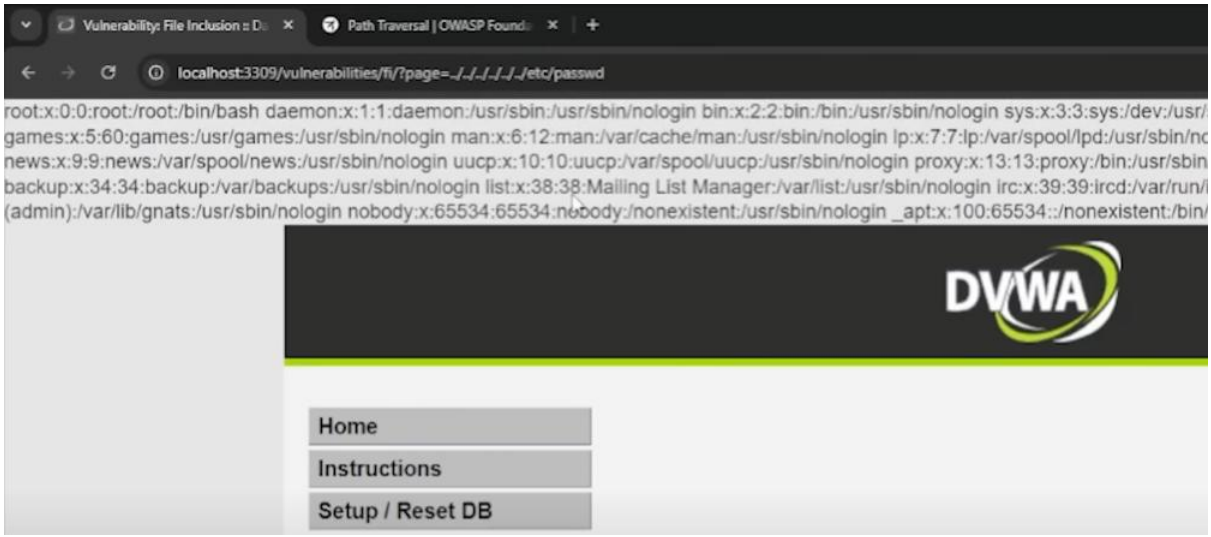

Figure 4.1: Exploiting Broken Access Control by manipulating the URL parameter to access /etc/passwd, revealing system user account information due to lack of proper input validation and path restriction.

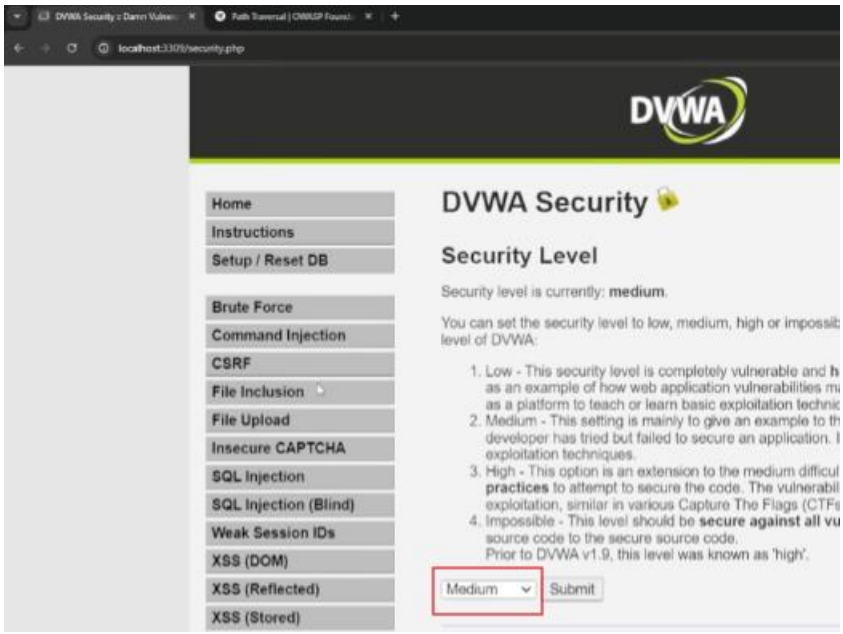

Figure 4.2: The DVWA security level was changed to Medium to test how the application's behaviour changes with stricter security controls. This helps assess whether the vulnerability persists or is mitigated at higher levels.

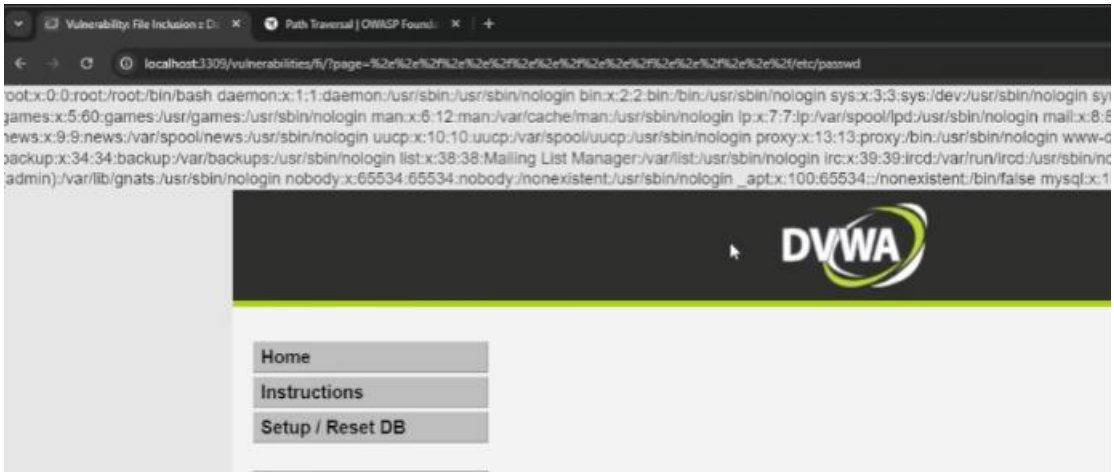

Figure 4.3: The application blocked the standard ../ path traversal pattern, so the encoded version %2e%2e%2f was used to bypass input filters and successfully trigger the path traversal vulnerability.

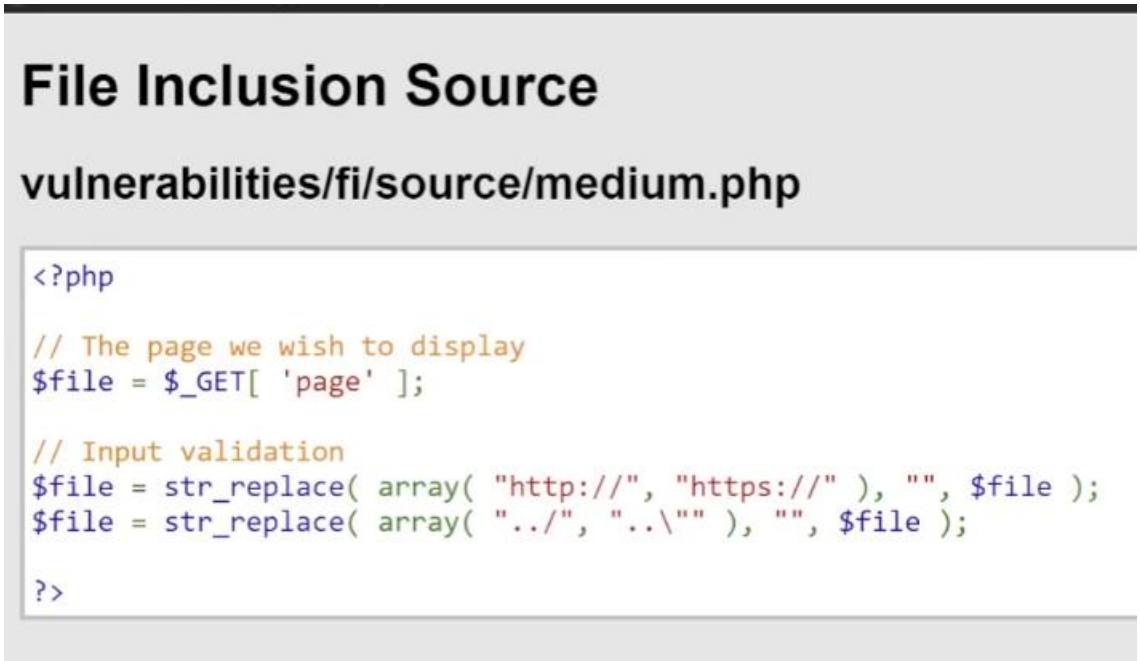

Figure 4.4: Source code snippet showing basic input validation logic used to block path traversal attempts by filtering specific patterns like ../.

Mitigation

- Implement **server-side authorization checks** on every request.
- Enforce **least privilege principles** users only access what they should.
- Avoid exposing sensitive data in URLs or client-side code.
- Use **role-based access control (RBAC)** or attribute-based access control (ABAC).
- Regularly audit and test access controls.

Note: DVWA's higher security levels restrict these attacks by enforcing proper checks.

4.2 Cryptographic Failures

Overview

Sensitive Data Exposure happens when an application fails to adequately protect critical information such as passwords, credit card numbers, or personal data. This can occur through insecure storage, weak encryption, or data transmitted over unencrypted channels.

Impact

- Theft of personal information
- Account compromise
- Financial fraud
- Loss of user trust and legal consequences

Exploitation in DVWA

In DVWA, sensitive data exposure is demonstrated mainly through:

- **Weak password storage:** Passwords are stored in plain text or using weak hashing.
- **Unencrypted connections:** Since DVWA runs locally over HTTP, data sent between client and server is unencrypted and can be intercepted.

Example: By sniffing network traffic with tools like Wireshark or Burp Suite, you can capture login credentials in clear text.

Payloads Used

- No specific payloads are needed here since the flaw is in how data is stored or transmitted.
- Simply logging in or intercepting HTTP requests reveals sensitive information.

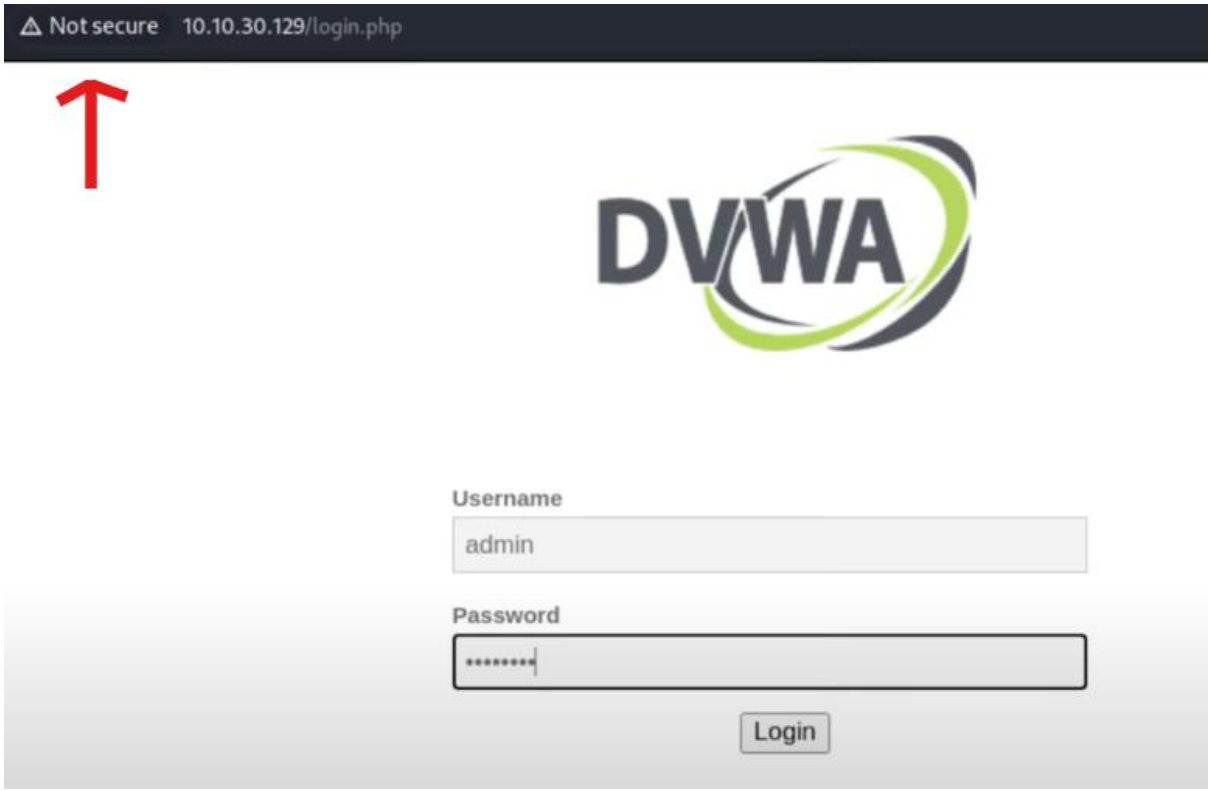

Figure 4.5: The browser displays a "Not Secure" warning for the DVWA site, indicating it's being served over HTTP instead of HTTPS highlighting a lack of transport layer security. This exposes data in transit to potential interception or tampering.

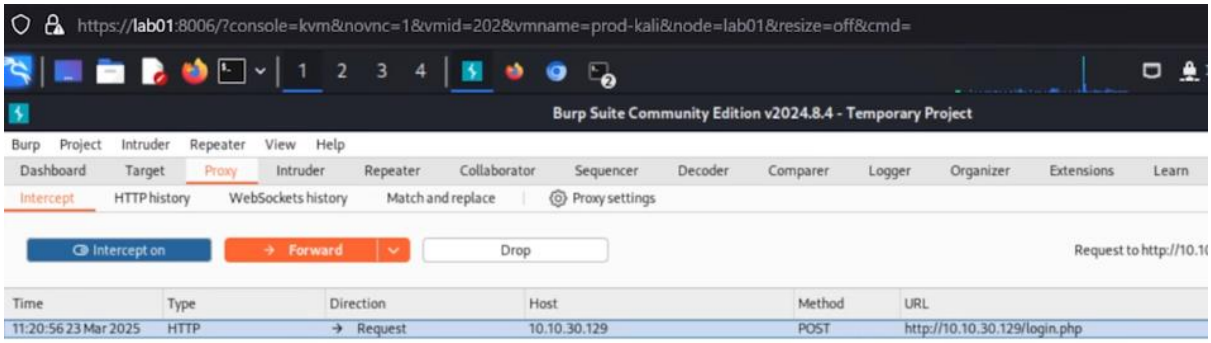

Figure 4.6: Burp Suite intercepting HTTP traffic between the browser and DVWA, allowing inspection and manipulation of requests demonstrating how attackers can analyse and exploit insecure communication flows.

Figure 4.7: Burp Suite intercepts the HTTP POST request from the DVWA login page, revealing the username and password in plaintext. This highlights the risk of transmitting credentials over unencrypted HTTP.

Mitigation

- Use **strong encryption** for data at rest (hashed and salted passwords, encrypted databases).
- Enforce **HTTPS/TLS** to protect data in transit.
- Avoid storing sensitive data unless absolutely necessary.
- Implement secure key management and regularly audit data handling.
- Use modern cryptographic algorithms and libraries.

Note: DVWA's default setup is intentionally insecure to highlight these weaknesses. Real-world applications must enforce strict data protection standards.

4.3 Injection

4.3.1 SQL Injection

Overview

SQL Injection is a vulnerability that occurs when user-supplied input is improperly handled and executed as part of a SQL query. If an application doesn't sanitize inputs, an attacker can manipulate queries to bypass authentication, extract data, or even modify the database.

Impact

- Bypass login authentication
- Access or dump sensitive database records (users, passwords, emails, etc.)
- Delete or modify database contents
- In severe cases, gain shell access or escalate privileges

Exploitation in DVWA

In DVWA, the SQL Injection vulnerability exists under:
Vulnerabilities → SQL Injection

1. Set DVWA security level to **Low** from the DVWA Security tab.
2. Navigate to the SQL Injection page.
3. You'll see a simple input field asking for a user ID.
4. Instead of entering a valid user ID like 1, enter the payload:

```
1' OR '1'='1
```

5. The app responds by displaying **all user records**, proving that the SQL query was altered.

Payloads Used

```
1' OR '1'='1

' OR 1=1 --

admin' --
```

These payloads take advantage of unescaped user input and alter the SQL query logic.

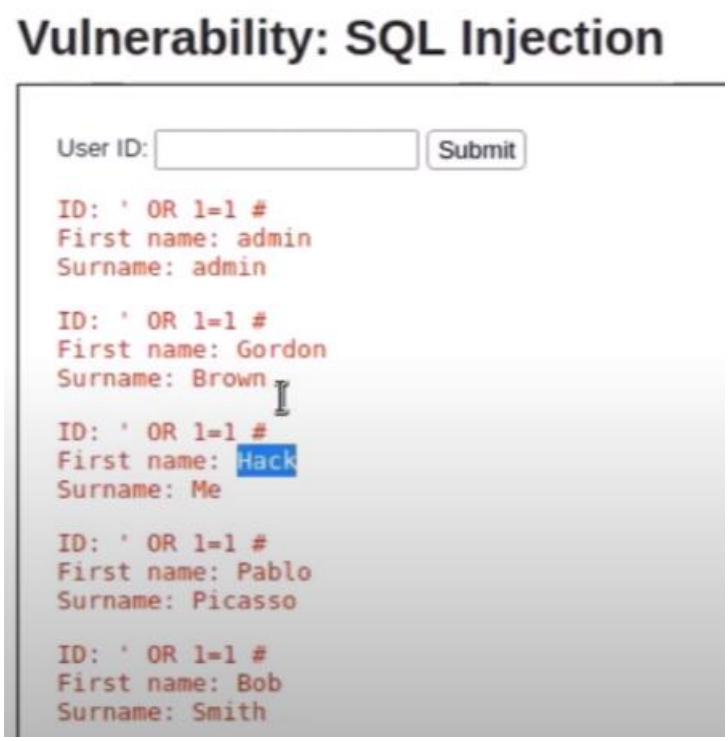

Figure 4.8: Demonstration of SQL Injection vulnerability on DVWA with the security level set to Low, allowing easy exploitation through unsanitized user input.

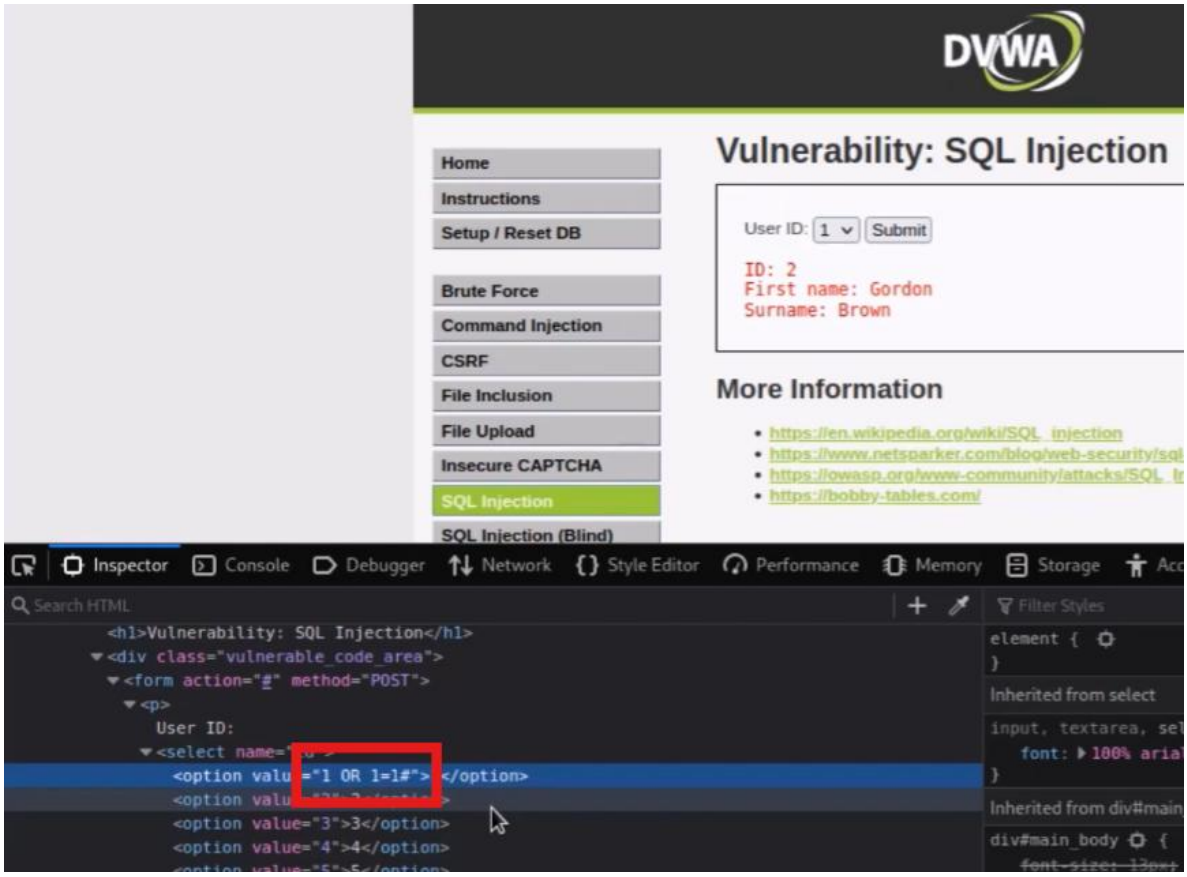

Figure 4.9: With DVWA security level set to Medium, manual inspection and manipulation of page parameters by changing the option value.

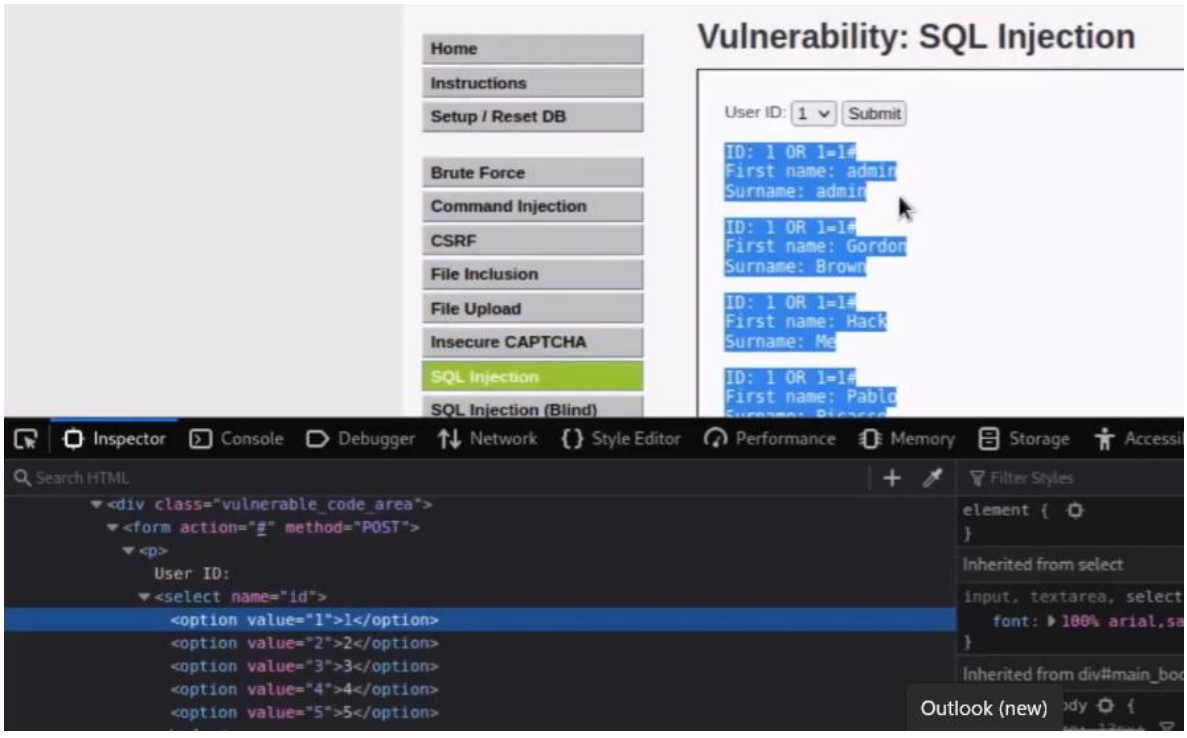

Figure 4.10: Manual inspection and manipulation of page parameters allowed successful SQL Injection, demonstrating that security controls at this level are still by passable.

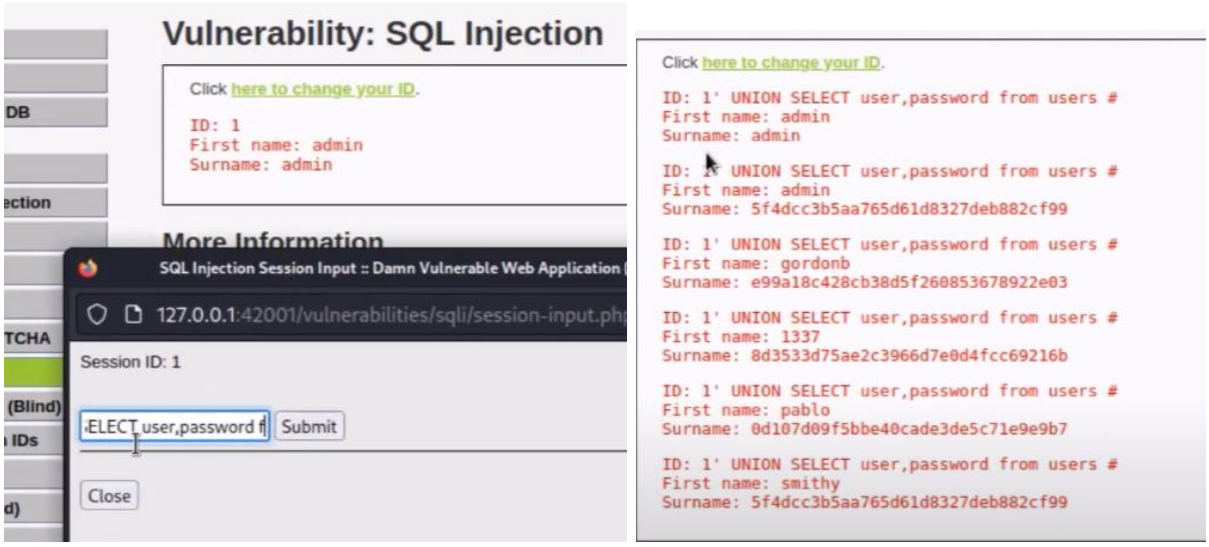

Figure 4.11: On DVWA set to High security level, an input window appears to change the user ID. Entering the payload *1 UNION SELECT user, password FROM users#* successfully exploits SQL Injection, with the retrieved usernames and passwords displayed in the right image.

Mitigation

To fix SQL Injection:

- Use **prepared statements** (parameterized queries) to separate code from data.
- Sanitize and validate all user inputs never trust raw input.
- Use **ORMs** (Object-Relational Mappers) that handle queries safely.
- Avoid displaying database errors directly in the browser.
- Implement least privilege access for database accounts.

Note: In DVWA, increasing the security level to **High** protects against this attack using prepared statements. Try the same payload at higher levels to see how it gets blocked.

4.3.2 Cross-Site Scripting (XSS)

Overview

Cross-Site Scripting (XSS) allows attackers to inject malicious scripts into web pages viewed by other users. These scripts run in the victim's browser and can steal cookies, capture keystrokes, redirect users, or manipulate the page.

There are three main types:

- **Stored XSS** – malicious code is saved on the server and served to users
- **Reflected XSS** – payload is part of the request and reflected in the response
- **DOM-Based XSS** – executed via client-side scripts and the DOM, without server involvement

Impact

- Theft of session cookies (account hijacking)
- Redirecting users to malicious sites
- Keylogging or form data capture
- Defacement or altering page content
- Full takeover if session or auth tokens are exposed

Exploitation in DVWA

DVWA has a dedicated **XSS (Reflected)** and **XSS (Stored)** module. Here's how I exploited both at Low security:

1. Reflected XSS

1. Go to:
   Vulnerabilities → XSS (Reflected)

2. Enter the following payload in the input field:

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

3. The script runs immediately in the browser—proof of injection.

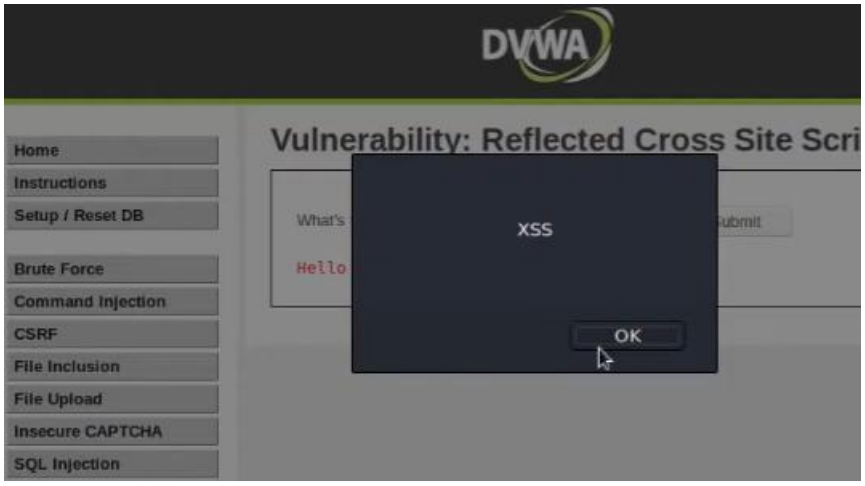

Figure 4.12: The XSS script injected into the input field at Low security level is executed on the page, demonstrating a successful stored Cross-Site Scripting attack.

2. Stored XSS

1. Go to:
   Vulnerabilities → XSS (Stored)

2. Enter this in the message or name field:

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

3. Submit the form, refresh the page, and the alert runs again—this time stored in the backend and triggered on every load.

Payloads Used

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

"><script>alert(1)</script>

<svg onload=alert(1)>
```

Additionally, tools like Burp Suite and BeEF can be used to intercept and manipulate traffic targeting the stored XSS vulnerability. By injecting BeEF's raw JavaScript payloads, an attacker can gain persistent control over the victim's browser, enabling advanced exploitation and browser-based attacks.

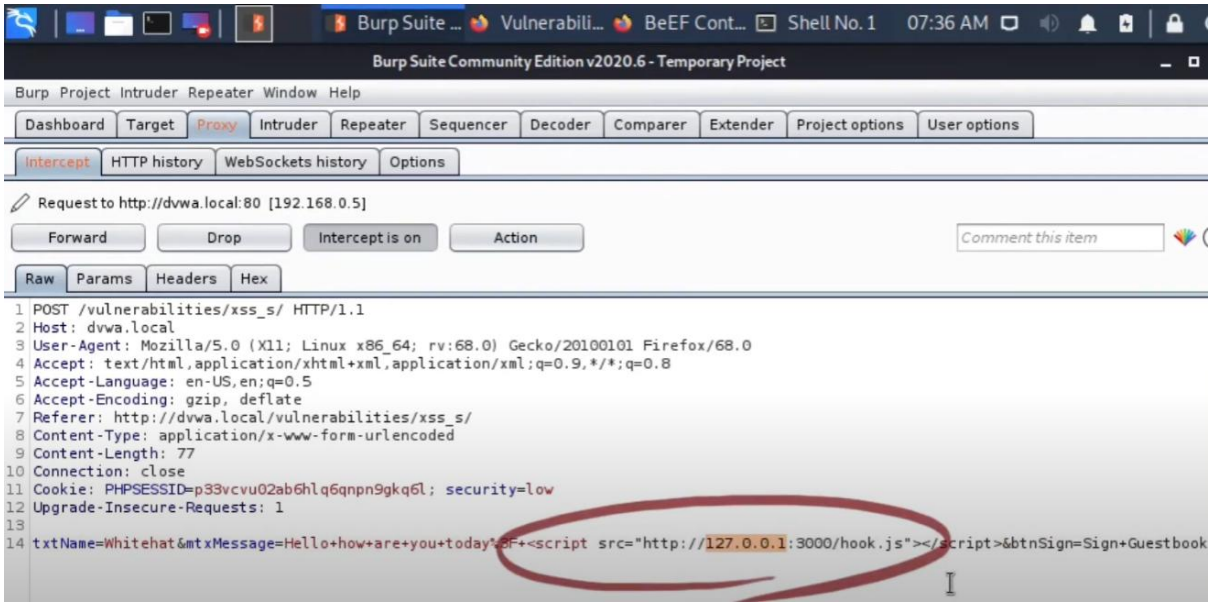

Figure 4.13: Burp Suite intercepts the HTTP request, showing the manipulated input with the local IP (192.168.0.10) instead of 127.0.0.1.

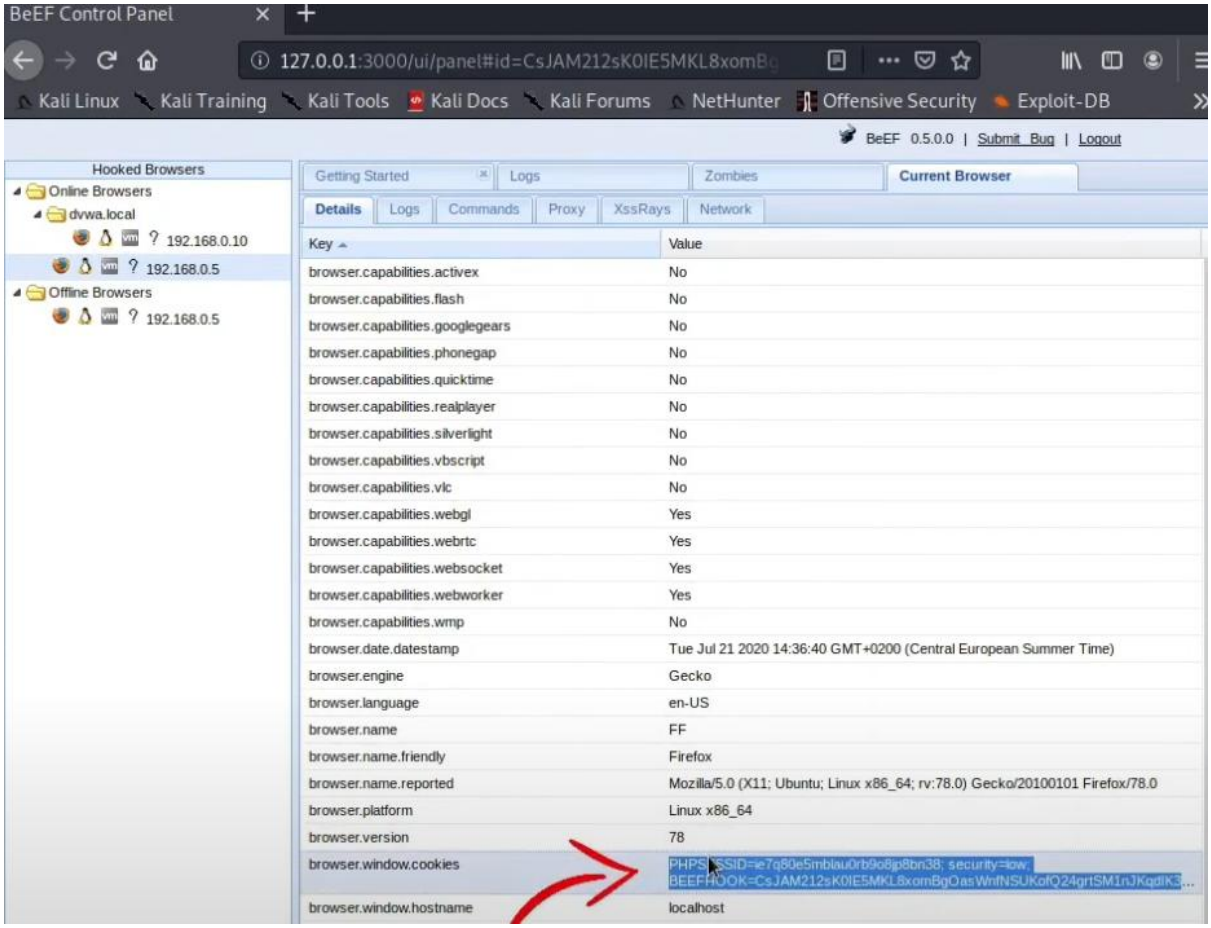

Figure 4.14: BeEF control panel displaying captured cookies and detailed information about the hooked victim's browser, demonstrating successful exploitation through stored XSS.

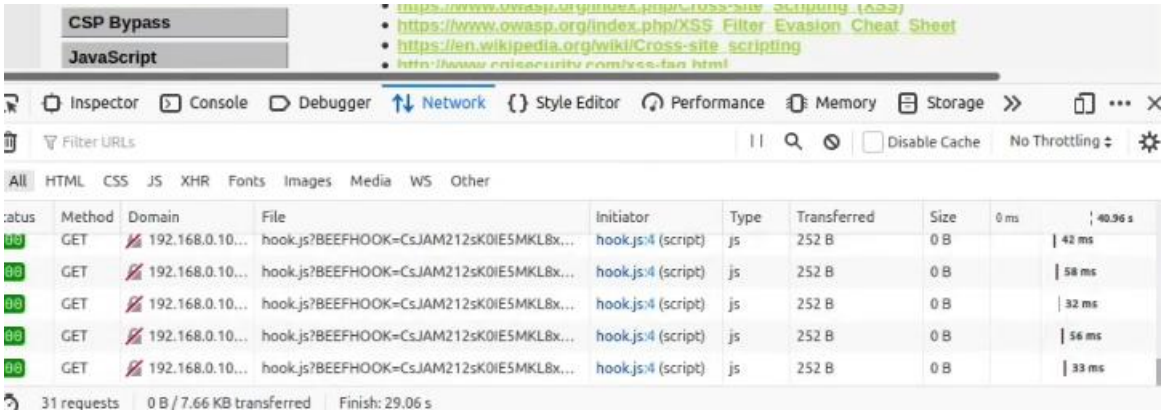

Figure 4.15: On the victim machine, the browser's developer tools show an active session communicating with the BeEF server at 192.168.0.10, confirming successful session hijacking via the injected BeEF hook cookie.

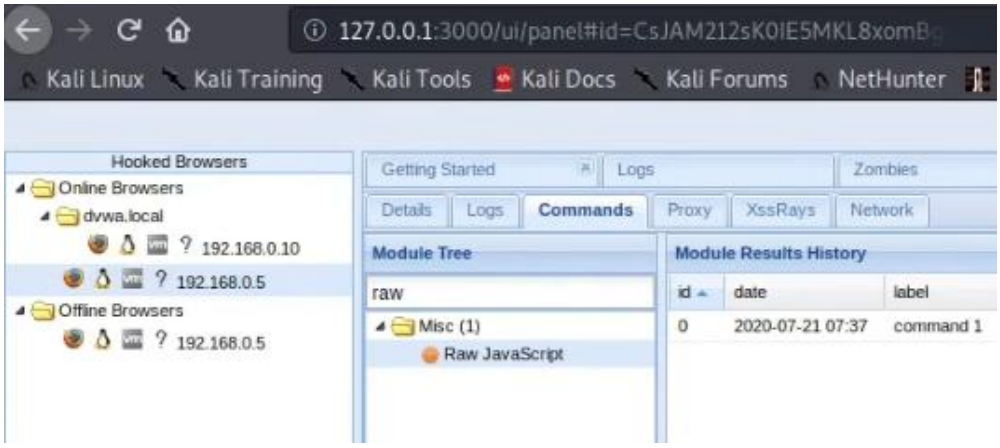

Figure 4.16: BeEF control panel interface, allowing the attacker to send raw JavaScript payloads directly to the hooked victim's browser for advanced exploitation.

Figure 4.17: The JavaScript prompt triggered on the victim's browser as a result of executing a payload sent from the BeEF control panel, confirming successful remote code execution via XSS.

Mitigation

- **Escape output** in HTML, JavaScript, and attributes
- Use secure frameworks that auto-sanitize input (e.g., React, Angular)
- Implement **Content Security Policy (CSP)** to limit script execution
- Sanitize user inputs using libraries (e.g., DOMPurify)
- Set HttpOnly flag on cookies to prevent JavaScript access

DVWA's High and Impossible levels neutralize these attacks with proper input handling and output encoding.

4.3.3 Command Line Injection

Overview

Command Line Injection (also known as OS Command Injection) occurs when an application passes unsafe user input directly into a system shell or command interpreter without proper

sanitization or validation. This allows attackers to execute arbitrary commands on the host operating system.

It's especially dangerous because it can lead to full system compromise if the web server runs with elevated privileges.

Impact

- Execute arbitrary system commands
- Read or modify sensitive files
- Launch reverse shells to gain remote access
- Pivot to other machines within the network
- Potential full system takeover

Exploitation in DVWA

DVWA's **Command Execution** module is a perfect example of this vulnerability. Here's how I exploited it:

How I Exploited It

1. Navigate to Command Execution module.
2. Input a valid IP address (e.g., 127.0.0.1) in the form.
3. Append a command using shell operators like ;, &&, or |:

Payload Example:

```
127.0.0.1; whoami
```

4. Submit the form and observe the response. The result of the injected command (e.g., the current user) is displayed in the browser.

Additional Payloads:

```
127.0.0.1; ls -la

127.0.0.1 && cat /etc/passwd

127.0.0.1 | uname -a
```

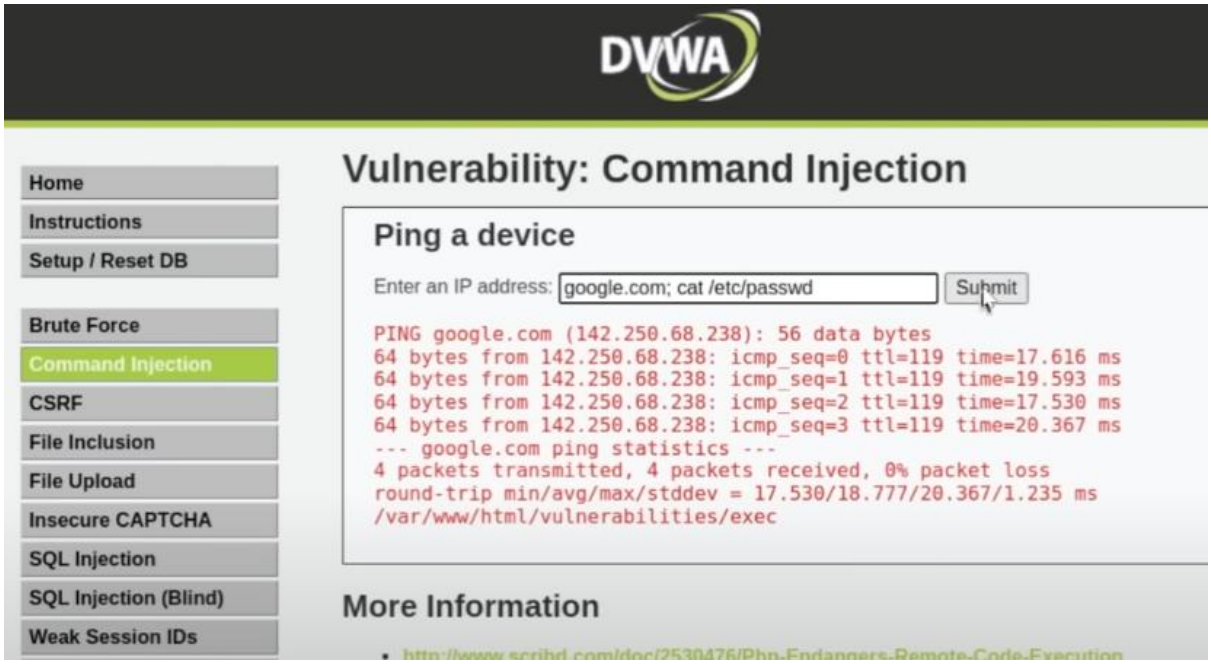

Figure 4.18: Command Injection using the payload *; cat /etc/passwd*, which chains commands to display the contents of the system's password file on the DVWA server.

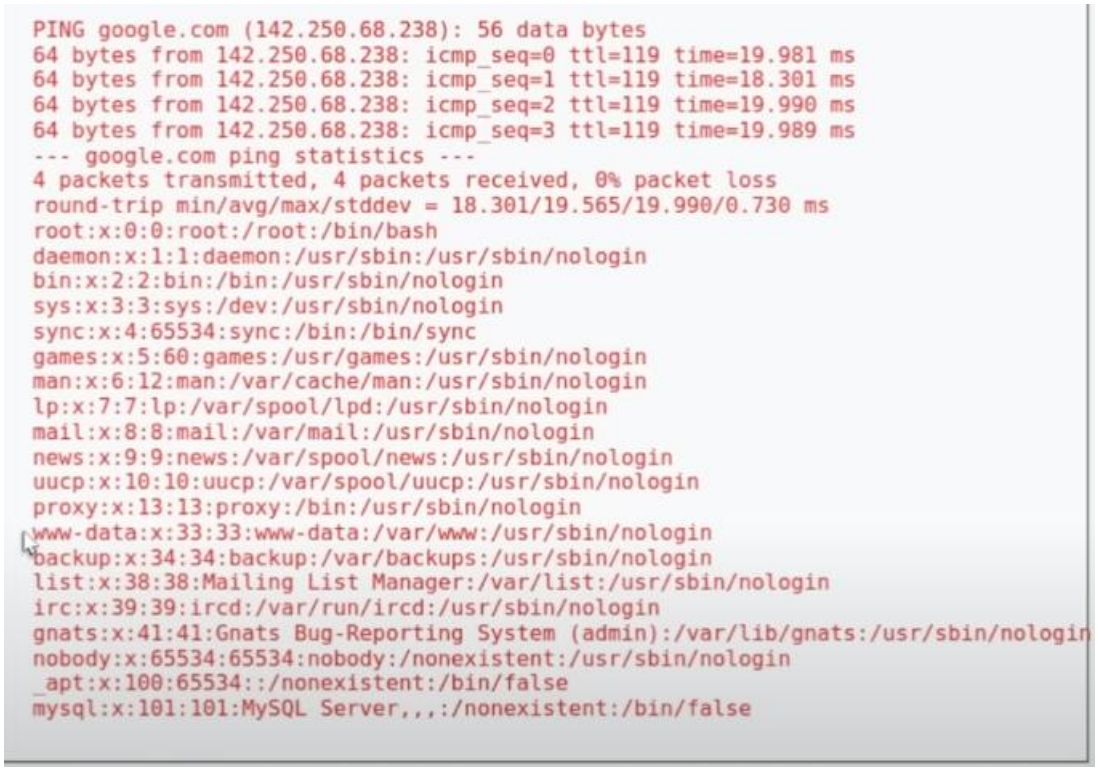

Figure 4.19: Output of the cat /etc/passwd command executed via Command Injection, revealing the contents of the system's password file on the DVWA server.

Command Injection Source

vulnerabilities/exec/source/medium.php

```php
<?php

if( isset( $_POST[ 'Submit' ]  ) ) {
    // Get input
    $target = $_REQUEST[ 'ip' ];

    // Set blacklist
    $substitutions = array(
        '&&' => '',
        ';'  => '',
    );

    // Remove any of the charactars in the array (blacklist).
    $target = str_replace( array_keys( $substitutions ), $substitutions, $target );

    // Determine OS and execute the ping command.
    if( stristr( php_uname( 's' ), 'Windows NT' ) ) {
        // Windows
        $cmd = shell_exec( 'ping  ' . $target );
    }
    else {
        // *nix
        $cmd = shell_exec( 'ping  -c 4 ' . $target );
    }

    // Feedback for the end user
    echo "<pre>{$cmd}</pre>";
}
```

Figure 4.20: Source code snippet explaining the altered behaviour of the application when DVWA security level is set to Medium, highlighting added input validation and filtering mechanisms.

Figure 4.21: Command Injection using the pipe (|) operator instead of semicolon (;) to chain commands and execute arbitrary system commands on DVWA.

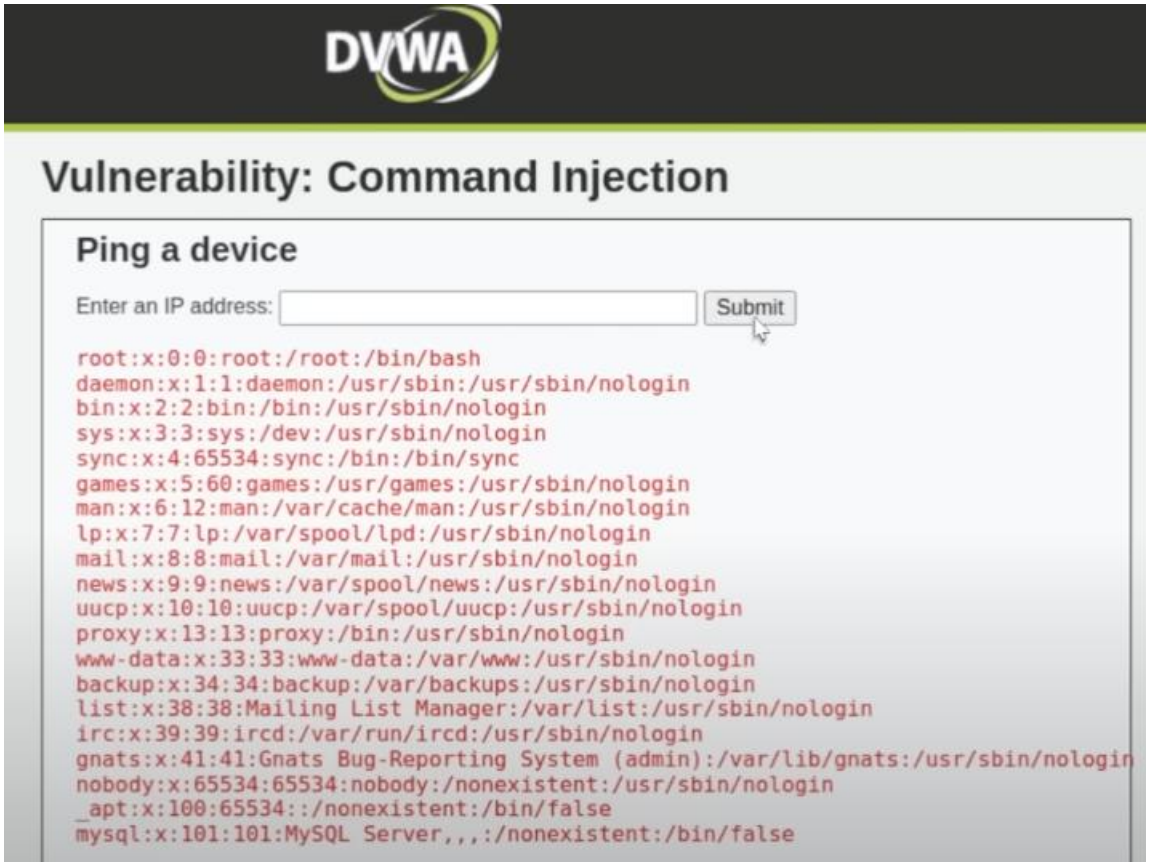

Figure 4.22: Output of the command injected via the pipe (|) operator, displaying the results of the executed system command on the DVWA server.

Mitigation

- **Avoid passing user input to system commands** wherever possible
- Use **safe API calls** instead of shell execution (e.g., Python's subprocess.run() with list inputs and no shell=True)
- Implement **input validation and allow-lists**
- Run web applications with the **least privileges**
- Use **Web Application Firewalls (WAFs)** to detect and block malicious payloads

Command injection is extremely dangerous and often leads to full compromise. If you must use system calls, make absolutely sure user input is tightly controlled.

4.4 Insecure Design

Overview

Insecure Design refers to flaws in the architecture or design decisions of an application that make it inherently unsafe, even if the code is implemented correctly. It's not just about coding mistakes it's about the *absence of security by design*.

Examples include:

- No limits on failed login attempts
- Lack of business logic validation
- Overly permissive access flows
- Missing threat modeling during development

This category focuses on whether the application was **designed** to be secure in the first place.

Impact

- Business logic abuse (e.g., bypassing payment, manipulating workflows)
- Unrestricted access to sensitive operations
- Exploitable flows that don't violate code but violate intent
- Inability to enforce user roles or permissions

Exploitation in DVWA

DVWA demonstrates insecure design in several places:

- **Brute Force login page** allows unlimited login attempts with no account lockout or CAPTCHA
- **File Upload module** lacks file type restrictions in lower security levels
- **No CSRF protections** in critical actions
- **No audit logging or access tracking**

These are all design-level flaws not just missing input validation, but missing protective *thought*.

Example Flow: Brute Force Login

1. Navigate to Brute Force module
2. Attempt repeated logins with different password guesses
3. There is no lockout or alert triggered this is a design flaw

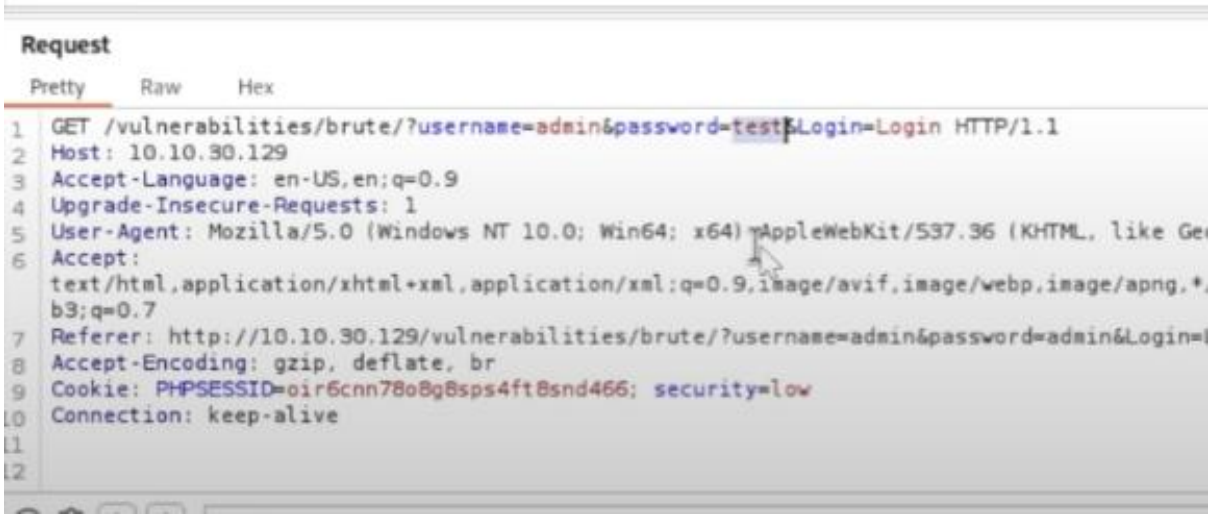

Figure 4.23: Burp Suite intercepting the HTTP POST request during login to DVWA using "admin" as username and password as "test", exposing credentials in plaintext due to lack of HTTPS encryption.

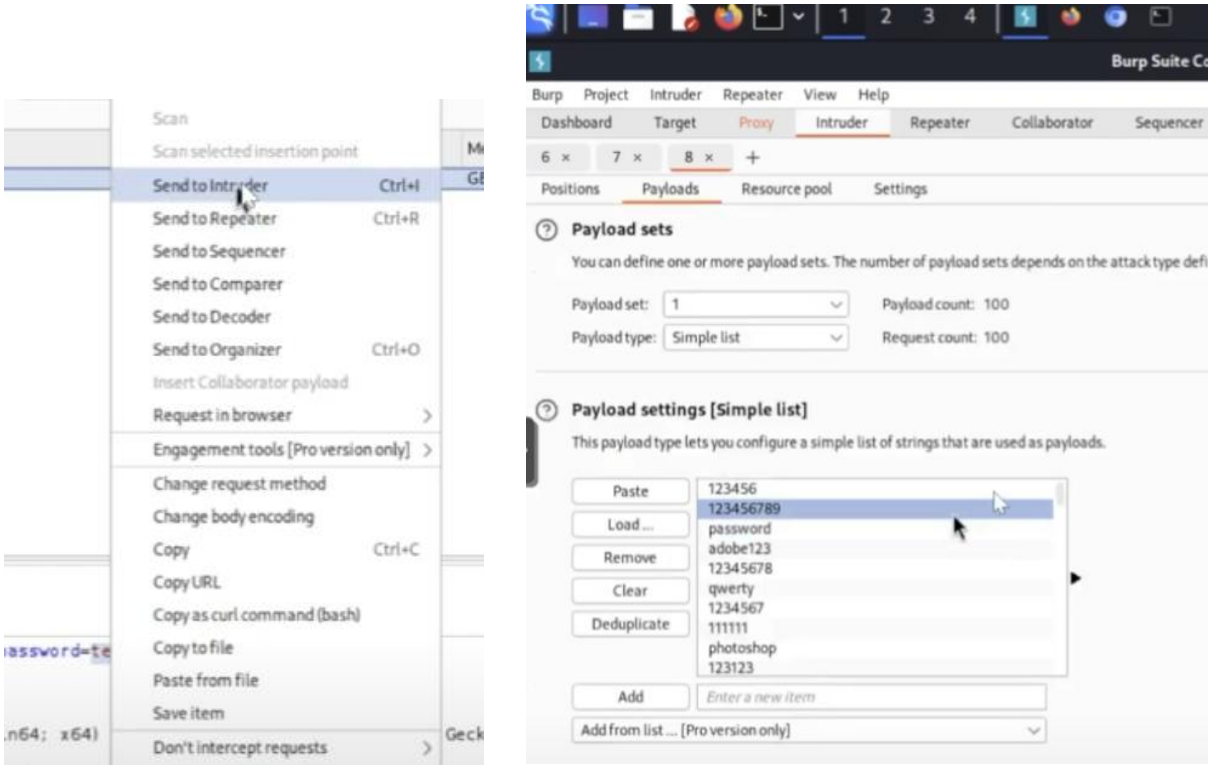

Figure 4.24: On the left, Burp Suite's context menu is used to send the login request to the Intruder module. On the right, the Intruder tab shows the payload position and a simple wordlist loaded to initiate a brute-force attack on the password field.

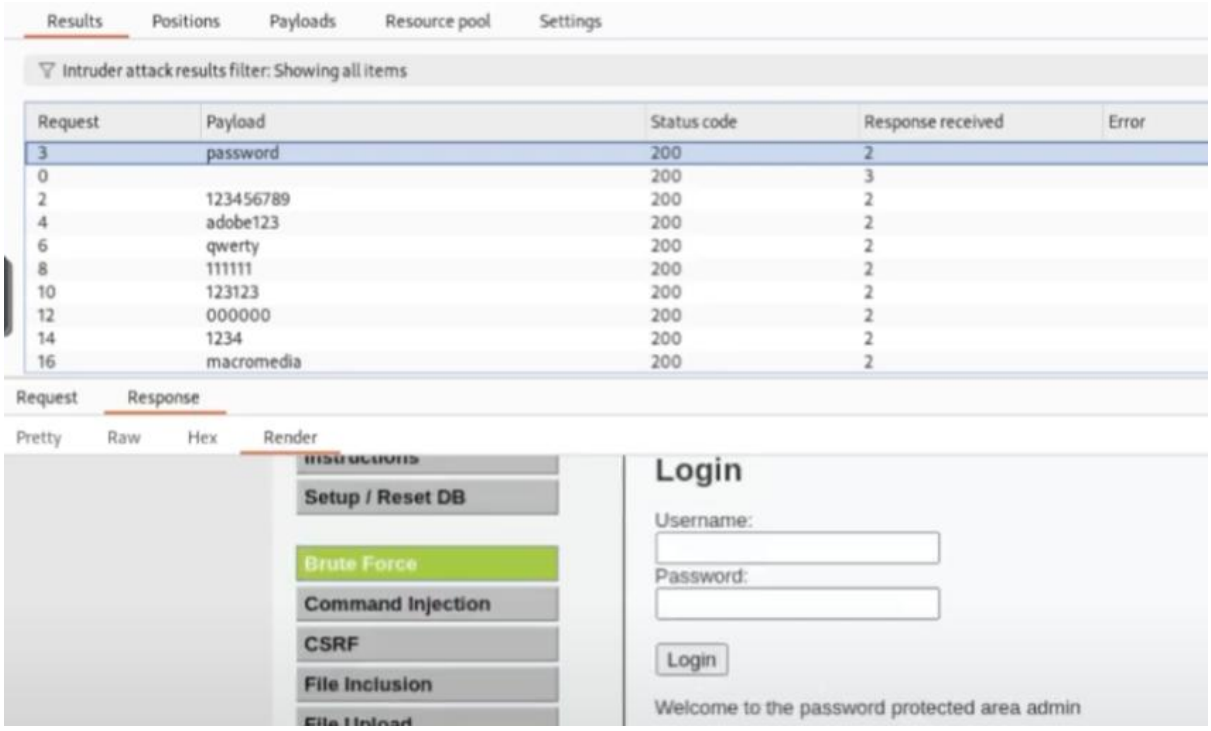

Figure 4.25: Burp Suite Intruder attack results sorted by response length. The successful login is identified when the password "password" returns a longer response, confirming valid credentials through response size analysis.

Mitigation

- Apply **secure design principles** like Least Privilege, Defence in Depth, Fail-Safe Defaults
- Implement rate limiting, lockout, and CAPTCHA for auth flows
- Include security architects or security engineers during the planning and design phase
- Conduct **threat modeling** during feature design to anticipate abuse
- Review business logic from an attacker's perspective—not just code correctness

Insecure design is hard to patch later. Secure architecture decisions must happen early, not after deployment.

4.5 Security Misconfiguration

4.5.1 XML External Entities (XXE)

Overview

XML External Entities (XXE) attacks exploit vulnerabilities in XML parsers that process external entity references. When improperly configured, an attacker can trick the parser into reading local files, causing denial of service, or even executing malicious code.

Impact

- Disclosure of sensitive files (e.g., /etc/passwd)
- Server-side request forgery (SSRF)
- Denial of service (application crashes)
- Remote code execution in severe cases

Exploitation in DVWA

DVWA does not directly include an XXE vulnerability or XML processing functionality. However, understanding XXE is important since many web apps still parse XML data without proper security controls.

If you want to test XXE, consider standalone vulnerable apps like **bWAPP** or custom vulnerable XML parsers.

Mitigation

- Disable external entity processing in XML parsers.
- Use fewer complex data formats like JSON where possible.
- Validate and sanitize all XML inputs.
- Keep libraries and parsers updated to latest secure versions.

4.5.2 Security Misconfiguration

Overview

Security Misconfiguration is one of the most common vulnerabilities. It happens when security settings are left at default, misconfigured, or incomplete giving attackers easy opportunities to exploit services, apps, or servers.

This can include anything from directory listing being enabled, verbose error messages revealing stack traces, open ports/services, or using outdated software.

Impact

- Unauthorized access to internal functionality or data
- Exposure of sensitive information (debug messages, API keys, etc.)
- Increased attack surface (unnecessary services, outdated software)
- Full system compromise if admin panels or credentials are exposed

Exploitation in DVWA

DVWA is intentionally misconfigured to demonstrate these issues:

1. **Default credentials**:
   Logging in with admin:password works out of the box.

2. **Verbose error messages**:
   At Low security, SQL errors are displayed directly in the browser, leaking internal query structure.

3. **Directory browsing**:
   If Apache directory listing is enabled, attackers can view file structures directly in the browser.

4. **Exposed configuration files**:
   Accessing files like /config/config.inc.php can reveal database credentials if server permissions are misconfigured.

Payloads / Tests Performed

- Accessing default paths and config files manually
- Deliberately triggering SQL errors
- Attempting directory traversal or enumeration
- Checking XAMPP for exposed admin interfaces like phpmyadmin

Mitigation

- Disable default accounts and change default credentials
- Disable directory listing in the web server
- Suppress detailed error messages in production
- Keep all software (web server, CMS, plugins) updated
- Regularly review and harden server, app, and database configurations
- Use automated scanning tools to detect misconfigurations

DVWA makes these flaws obvious, but in real-world scenarios, they're often subtle and easy to miss without regular audits.

4.6 Vulnerable and Outdated Components

Overview

This vulnerability occurs when applications use outdated libraries, frameworks, or software components with known security flaws. If these components are not patched or monitored, attackers can exploit known CVEs (Common Vulnerabilities and Exposures) to compromise the system.

What makes this dangerous is that exploitation doesn't require new techniques—attackers just reuse existing public exploits.

Impact

- Full system compromise through known exploits
- Privilege escalation
- Data breaches
- Exposure of internal APIs, logic, or secrets
- Lateral movement across environments

Exploitation in DVWA

DVWA is intentionally built using outdated software to demonstrate insecure configurations and vulnerabilities:

- **DVWA uses PHP versions and MySQL packages with known security issues**.
- **XAMPP** (used to host DVWA) often includes unpatched components like outdated versions of Apache, PHPMyAdmin, etc.
- **Client-side libraries** (JavaScript) are not updated.

For example, you can:

- Run tools like **Nmap**, **Nikto**, or **OWASP Dependency-Check** on the DVWA server.
- Identify software versions and match them with CVEs on websites like CVE Details or Exploit DB.

Mitigation

- Maintain an **inventory of all third-party libraries and dependencies**
- Regularly check for CVEs affecting your stack
- Use automated tools (e.g., OWASP Dependency-Check, Retire.js, Snyk, npm audit)
- Apply patches and updates as soon as they're available
- Remove unused components and services

Real-world attackers often go after old, forgotten packages it's low-effort, high-reward for them if you're not keeping things up to date.

4.7 Identification and Authentication Failures

Overview

Broken Authentication occurs when an application fails to properly protect user credentials, session IDs, or authentication logic. This allows attackers to bypass login, hijack sessions, or impersonate other users. It's a critical issue that can lead to full account takeover.

Impact

- Unauthorized access to user accounts
- Privilege escalation to admin accounts
- Session hijacking
- Identity theft and data leakage

Exploitation in DVWA

DVWA has several ways to demonstrate broken authentication, especially at **Low** security levels:

Login Bypass via SQL Injection

At login.php, you can bypass authentication by injecting SQL in the username field:

```
Username: admin' --

Password: (anything)
```

This works because the query becomes something like:

```
SELECT * FROM users WHERE username = 'admin' -- ' AND password
= '...'
```

Weak Passwords

DVWA users have extremely weak default passwords like admin:password. Try brute-forcing or guessing passwords to gain access.

Session ID Predictability

After logging in, the session ID (PHPSESSID) can be guessed or reused, allowing session hijacking in poorly implemented environments.

Payloads Used

```
admin' --

' OR 1=1 --
```

Also tested simple passwords like password, admin123, etc.

Mitigation

- Implement **multi-factor authentication (MFA)**
- Enforce **strong password policies**
- Use **secure session management**: regenerate session IDs after login, set HTTPOnly and Secure flags
- Limit failed login attempts (rate limiting, account lockout)
- Use hashed + salted passwords (e.g., bcrypt, Argon2)
- Validate and sanitize all user input to prevent login bypass

Note: As you raise the DVWA security level, these flaws are gradually mitigated. At High or Impossible, login bypass is blocked with prepared statements and better session handling.

4.8 Software and Data Integrity Failures

Overview

Insecure Deserialization occurs when untrusted data is used to recreate objects or data structures without proper validation. If an application deserializes user-supplied input, an attacker can manipulate the serialized data to execute arbitrary code, escalate privileges, or tamper with application logic.

This is a dangerous vulnerability because even if remote code execution isn't possible, it can still allow replay attacks, injection, or data tampering.

Impact

- Remote Code Execution (RCE)
- Authentication or authorization bypass
- Data tampering
- Denial of service

Exploitation in DVWA

DVWA does **not** include insecure deserialization scenarios by default. However, you can simulate or practice this vulnerability in intentionally vulnerable apps like:

- **bWAPP**
- **Java applications using insecure object deserialization (e.g., Apache Commons Collections gadgets)**
- **PHP Object Injection labs**

In those environments, you might:

- Intercept serialized objects with **Burp Suite**
- Modify serialized strings to inject malicious payloads
- Exploit insecure unserialize() functions in PHP or similar methods in Java/.NET

Payload Example (PHP)

```
O:8:"ExploitMe":1:{s:4:"data";s:13:"malicious_code";}
```

Mitigation

- Never deserialize data from untrusted sources
- Use safer data formats (e.g., JSON, XML with strict parsing)
- Apply integrity checks (e.g., HMAC) to detect tampering
- Use allow-lists for classes that can be deserialized
- Keep deserialization libraries patched and up to date

While DVWA skips this, it's still crucial to understand, especially in Java or PHP-heavy environments where insecure deserialization is a common issue.

4.9 Security Logging and Monitoring Failures

Overview

This vulnerability refers to the failure to log security-relevant events or monitor those logs effectively. Without proper logging and monitoring, attacks can go unnoticed, giving attackers time to escalate, pivot, and exfiltrate data without detection.

It’s not just about collecting logs it's about knowing what to log, detecting anomalies, and responding in time.

Impact

- Missed early signs of breaches (e.g., brute-force attempts, SQL injection attempts)
- Delayed incident response or no response at all
- Compliance failures (e.g., GDPR, HIPAA, PCI-DSS)
- Attackers remaining undetected for long periods (dwell time)

Exploitation in DVWA

While DVWA itself doesn't simulate logging failures explicitly, its design shows what happens in their absence:

- No audit logs for failed login attempts
- No alerts triggered on repeated SQLi or XSS payloads
- No session tracking or alerts for privilege abuse
- Attacks (even successful ones) leave no footprint in logs by default

To demonstrate this practically, I integrated **Microsoft Sentinel** in my broader lab setup (see previous project) to detect these kinds of events on a real Windows 10 VM. Sentinel flagged things like brute force attempts, failed RDP logins, and malicious input patterns—none of which DVWA would catch alone.

Mitigation

- Log authentication attempts (both success and failure)
- Log input validation errors, access control failures, and system errors
- Store logs securely and protect against tampering
- Use centralized log management and correlation tools (e.g., SIEMs like Sentinel, Splunk, ELK)
- Set up alerting rules for suspicious patterns (e.g., 5 failed logins in 1 min)
- Regularly review and test your logging setup during incident response drills

Logging and monitoring won't stop an attack but without them, you'll never even know it happened.

4.10 Server-Side Request Forgery (SSRF)

Overview

Server-Side Request Forgery (SSRF) happens when an attacker tricks a server into making requests to internal or external resources the attacker shouldn't have access to. The attacker abuses a functionality that fetches remote URLs like webhooks, file downloads, or metadata lookups.

This can allow access to internal services (e.g., AWS metadata), bypass firewalls, and even lead to remote code execution.

Impact

- Internal network scanning and service enumeration
- Access to internal APIs or cloud metadata (e.g., AWS EC2 instance credentials)
- Sensitive data exposure
- Potential remote code execution depending on response handling

DVWA Support

DVWA does **not natively support SSRF**. However, you can experiment with SSRF in other vulnerable applications like:

- **SSRF Labs** by PortSwigger
- **bWAPP** (has limited SSRF cases)
- **Hackazon**, **WebGoat**, or custom-built test environments

Example Exploit Flow

If a form allows users to supply a URL for the server to fetch:

```
POST /fetch

Host: vulnerable-site.com

Content-Type: application/json



{

  "url": "http://169.254.169.254/latest/meta-data/"

}
```

This could return AWS EC2 metadata if running in a cloud instance without proper protections.

Mitigation

- Never trust user-supplied URLs or destinations
- Enforce URL allow-lists and block internal address ranges (127.0.0.1, 169.254.169.254, etc.)
- Validate and sanitize all user input used in outbound requests
- Disable unused protocols or internal resolvers
- Monitor outbound traffic from the server to spot suspicious requests

SSRF is increasingly common in cloud-native apps, APIs, and microservices—especially with metadata services and misconfigured proxies.

Chapter 5

5 Key Takeaways

Working through the OWASP Top 10 on DVWA gave me a deeper understanding of how these vulnerabilities actually play out not just in theory, but hands-on. Here's what stood out:

Real Exploits Make Concepts Stick

Reading about SQL Injection is one thing. Actually, bypassing a login form with ' OR '1'='1 makes it real. Seeing reflected XSS pop up in your own browser hits differently. Practical testing helps internalize the risk.

Security Isn't Just About Code

A lot of vulnerabilities exist not because of bad code, but because of missing security design. Brute force login without rate-limiting? That's not a bug, that's bad design. Misconfigured permissions or default creds? Same story.

Automation Can Miss Logic Bugs

Tools like scanners are great, but they often miss business logic flaws or insecure design issues. Thinking like an attacker and testing like one fill that gap.

Common Beginner Mistakes

- Focusing only on low-hanging fruit like SQLi and XSS
- Ignoring higher-level issues like access control or insecure design
- Forgetting about post-exploitation steps like persistence or data exfiltration
- Assuming the presence of HTTPS or input validation means an app is "secure"

How This Helps Attackers and Defenders

If you're defending systems, knowing how attacks work is essential. You can't monitor what you don't understand. If you're attacking for learning or red teaming, this gives a safe way to test skills and build muscle memory.

What's Next

This lab was phase 1 offensive testing. Next, I'll shift to **defensive hardening** by setting up a **Web Application Firewall (WAF)** to block these same attacks. This full-circle approach helps connect both sides of cybersecurity.

Chapter 6

6 Appendix / References

Tools Used

- **DVWA (Damn Vulnerable Web Application)** – Vulnerable PHP/MySQL app for security testing
- **XAMPP** – Local server stack (Apache, MySQL, PHP) to host DVWA
- **Kali Linux** – Penetration testing distro used for attacks
- **Burp Suite Community Edition** – Intercepting proxy for testing web app flows
- **OWASP ZAP** – Optional scanner and testing tool
- **Nmap** – Network scanning
- **Nikto** – Basic web vulnerability scanner

Learning Resources

- OWASP Top 10 Official Page
- PortSwigger Web Security Academy – Free training on web app vulnerabilities
- TryHackMe – Practical cybersecurity labs
- Hack The Box – Offensive security practice
- PayloadAllTheThings GitHub – Go-to resource for exploitation payloads
- OWASP Cheat Sheet Series – Mitigation-focused references
- CVEs on websites like CVE Details or Exploit DB.

Supporting Files in This Repo

- /screenshots/ – Visual proof and walk-throughs of each vulnerability tested
- DVWA.pdf – Full step-by-step guide and analysis document (this file)