International Cybersecurity and Digital Forensic Academy

PROGRAMME: CYBERSECURITY AND ETHICAL HACKING INTERNSHIP

ASSIGNMENT

PRESENTED BY

AYILARA BUSARI DARE

IDEAS/24/28133

COURSE CODE: INT308

COURSE TITLE: Session Management and Web Security

COURSE FACILITATOR: Mahmoud Kani

February, 2024

Lab 1: Session Management Vulnerabilities

Exercise 1: Analyzing Session Management Practices

The session cookie details from the image above are as follows:

- **Name:** PHPSESSID
- **Value:** 56op8dkk027s50ik4i3qtdqrr7
- **Domain:** 192.168.201.129
- **Path:** /
- **Expires / Max-Age:** Session
- **Size:** 35
- **HttpOnly:** false
- **Secure:** false
- **SameSite:** None
- **Last Accessed:** Thu, 10 Feb 2025 12:14:35 GMT

Cookie attributes play a crucial role in web security by controlling how cookies are stored, transmitted, and accessed. Improperly configured session cookies can lead to significant vulnerabilities, potentially compromising user data and session integrity. Below is a discussion of key cookie attributes and the vulnerabilities that can arise from misconfiguration:

Key Cookie Attributes and Their Implications

1. Secure Attribute

Purpose: Ensures the cookie is only sent over HTTPS, protecting it from being transmitted over unencrypted channels.

Implications of Misconfiguration:

- If the `Secure` attribute is not set, the cookie can be sent over HTTP, making it susceptible to interception via man-in-the-middle (MITM) attacks.

- Attackers can steal session cookies and hijack user sessions.

2. HttpOnly Attribute

Purpose: Prevents client-side scripts (e.g., JavaScript) from accessing the cookie, mitigating cross-site scripting (XSS) attacks.

Implications of Misconfiguration:

- If `HttpOnly` is not set, attackers can exploit XSS vulnerabilities to steal cookies and impersonate users.
- Sensitive session data becomes accessible to malicious scripts.

3. SameSite Attribute

Purpose: Controls whether the cookie is sent with cross-site requests, helping to prevent cross-site request forgery (CSRF) and cross-origin attacks.

Values:

- `Strict`: Cookie is only sent in a first-party context.
- `Lax`: Cookie is sent with top-level navigations (e.g., clicking a link).
- `None`: Cookie is sent with all requests (requires `Secure` attribute).

Implications of Misconfiguration:

- If `SameSite` is not set or set to `None` without the `Secure` attribute, the cookie is vulnerable to CSRF attacks.
- Attackers can trick users into making unauthorized requests, potentially performing actions on their behalf.

4. Domain and Path Attributes

- Purpose: Define the scope of the cookie, specifying which domains and paths can access it.
- Implications of Misconfiguration:
- If the `Domain` attribute is too broad (e.g., `.example.com`), the cookie is accessible across subdomains, increasing the attack surface.

- If the `Path` attribute is too broad, the cookie is accessible on unrelated parts of the site, potentially exposing it to unauthorized access.

5. Expires and Max-Age Attributes

-Purpose: Define the lifetime of the cookie.

- Implications of Misconfiguration:

- If the expiration is set too far in the future, the cookie remains valid for an extended period, increasing the risk of theft and misuse.

- If the expiration is not set, the cookie becomes a session cookie, which is deleted when the browser is closed. However, this can still be risky if the session is not properly invalidated on the server side.

Vulnerabilities from Improperly Configured Session Cookies

1. Session Hijacking

- If session cookies lack the `Secure` and `HttpOnly` attributes, attackers can intercept them via MITM attacks or steal them via XSS, allowing them to impersonate the user.

2. Cross-Site Request Forgery (CSRF)

- Without the `SameSite` attribute, attackers can forge requests from other sites, tricking the browser into sending session cookies and performing unauthorized actions.

3. Cross-Site Scripting (XSS)

- If the `HttpOnly` attribute is missing, attackers can exploit

Exercise 2: Session Fixation Attack

1. Understand Session Fixation:

Session Fixation is an attack where an attacker sets a user's session ID to a known value before they log in. If the web application does not regenerate a

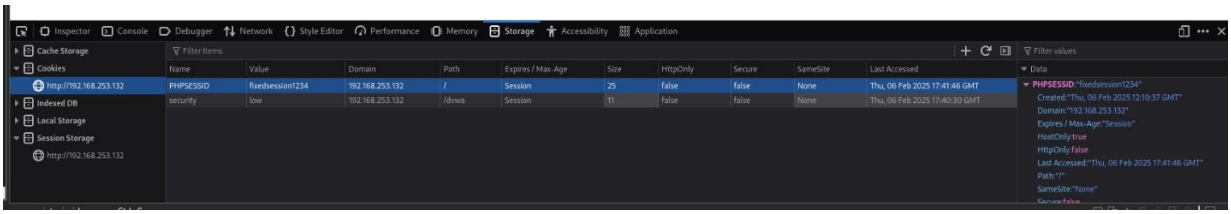
new session ID after authentication, the attacker can use the same session ID to gain unauthorized access to the victim's account.

- **Attack Vector:** The attacker forces a user to use a **pre-determined session ID**, which remains the same even after login.
- **Impact:** The attacker can impersonate the user and access their account.
- **Mitigation:** The web application must **regenerate the session ID** after login to prevent reuse.

2. Simulate a Session Fixation Attack:

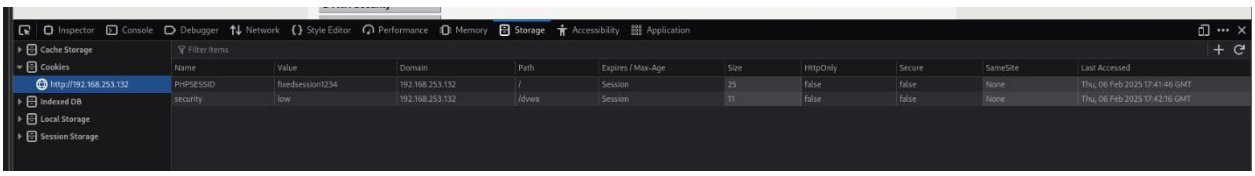
Step 1: Modify the Session ID Before Login

1. Open your browser's **Developer Tools (F12)**.
2. Go to the **Storage** tab and find the session cookie (e.g., PHPSESSID).
3. Change the **Value** of the PHPSESSID to a **predefined session ID** (e.g., fixedsession1234).
4. Press **Enter** to save the new session ID.



Step 2: Log into the Application

1. Open a new tab or refresh the page.
2. Login with a valid username and password.
3. Observe whether the session ID remains **unchanged** or if a **new session ID** is generated.



Step 3: Use the Fixed Session ID in Another Browser 1.

Open another browser (or an incognito/private window).

2. Open Developer Tools → **Storage** → **Cookies**.

3. Manually set the PHPSESSID value to fixedsession1234.

4. Try accessing the website.

I was able to access the authenticated session in a new private window which proves the application is vulnerable because the web application didn't regenerate a new session session ID.

3. Reflection & Analysis

Importance of Regenerating Session IDs After Login

Regenerating session IDs after login is a critical security practice that protects web applications from various session-related attacks, including session fixation and session hijacking. When a user logs in, a new session ID should be assigned to ensure that no one else can take over the session.

1. Prevents Session Fixation Attacks

- Session fixation occurs when an attacker forces a user to use a pre-set session ID, allowing them to hijack the session after login.
- How ID regeneration helps:
    - When the session ID is regenerated after authentication, the old ID becomes invalid. ○ Even if an attacker had set a session ID before login, they cannot use it afterward.

Example Fix: session_regenerate_id(true); // Generates a new session ID and deletes the old one

2. Mitigates Session Hijacking

- Session hijacking happens when an attacker steals a user's session ID (e.g., via MITM attacks, XSS, or malware).

- If the session ID remains the same after login, an attacker could reuse it to gain unauthorized access.
- How ID regeneration helps:
    - A new session ID invalidates the old one, rendering stolen IDs useless.

3. Enhances User Authentication Security

- Some users might reuse public or compromised session IDs unknowingly.
- Regenerating the session ID ensures that a fresh, unique session is used upon login, reducing the risk of unauthorized access.

4. Limits Session Replay Attacks

- In a session replay attack, an attacker captures a valid session ID and reuses it later.
- By regenerating session IDs frequently, attackers cannot replay old sessions to regain access.

Best Practices for Session ID Regeneration

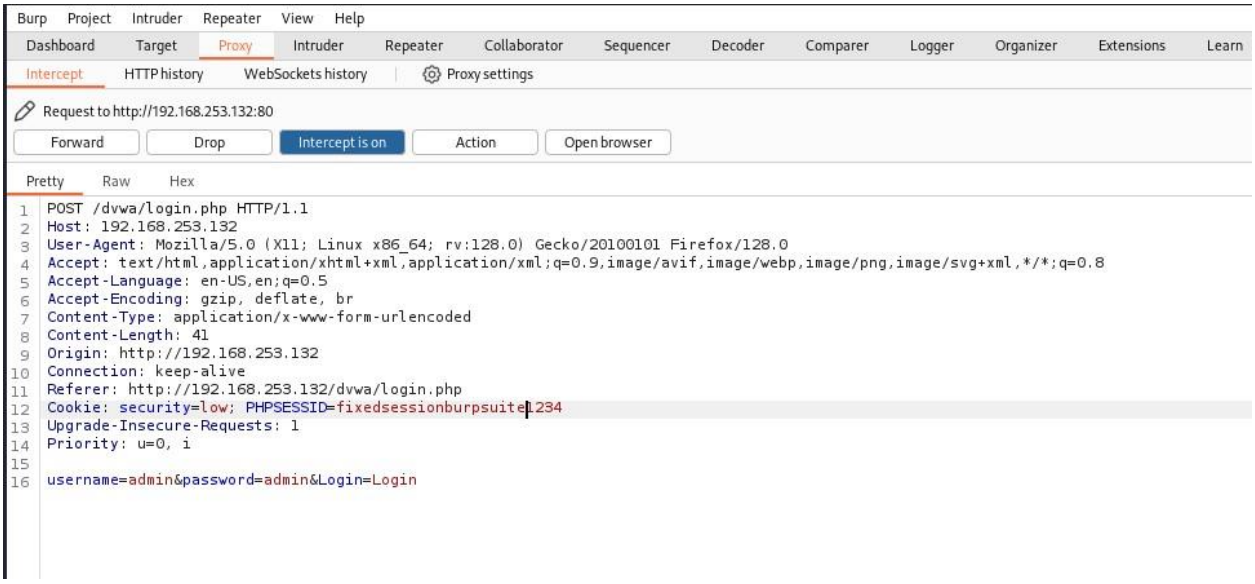
- Regenerate session IDs on every login: Prevents fixation and hijacking.
- Regenerate periodically during a session: Adds extra protection.
- Use HTTPS: Prevents attackers from intercepting session data.
- Enable Secure and HttpOnly cookies: Protects against XSS and MITM attacks.

Conclusion

Regenerating session IDs after login is a simple but powerful security measure that significantly reduces session fixation and hijacking risks. It should be combined with secure cookie settings, session expiration, and MFA for optimal protection.

Exercise 3: Session Hijacking Using Burp Suite

I modified the session ID to fixedsessionburpsuite1234, forwarded the request and I was able to to gain unauthorized access to the authenticated session.



Effectiveness of Session Management in the Application

Session management plays a crucial role in securing web applications by ensuring that users' sessions remain private, authenticated, and protected against unauthorized access. In your application, the session management mechanisms in place (such as secure cookies and session regeneration) contribute to a strong security posture.

However, session hijacking remains a major threat, where attackers steal a user's session ID to gain unauthorized access. While the current implementation includes secure cookie settings and session regeneration, additional security measures can further strengthen the system. Measures to Prevent Session Hijacking

1. Use Secure and HTTP-Only Cookies

Ensure cookies are only sent over HTTPS:

- This prevents attackers from intercepting session cookies via man-in-themiddle (MITM) attacks.
- Implemented using: session_set_cookie_params(['secure' => true, 'httponly' => true]);

Prevent JavaScript from accessing cookies:

- The httponly flag stops malicious scripts from stealing cookies via XSS attacks.

Restrict cookie use to the same site:

- samesite => 'Strict' prevents cookies from being sent in cross-site requests, reducing CSRF risks.

2. Implement Session ID Regeneration

Regenerate session IDs after login and at set intervals

- Prevents session fixation attacks, where an attacker sets a victim's session ID before login.
- Implemented using: session_regenerate_id(true);

3. Set Session Expiration and Inactivity Timeout

Limit the session duration to reduce attack windows.

4. Implement IP and User-Agent Binding

Restrict session usage to the same IP and browser

- Helps detect session hijacking if the session suddenly appears from another device or IP address.

5. Enable Multi-Factor Authentication (MFA)

Even if a session is hijacked, MFA adds an extra verification step, making unauthorized access more difficult.

6. Monitor and Log Sessions

Track active sessions and notify users of suspicious activity

- Implement session activity logs to detect unusual behavior (e.g., login from a new device).

Conclusion

While the current session management setup is effective, it can be further strengthened with IP binding, session timeouts, logging, and MFA. These measures reduce the risk of session hijacking and enhance the security of your application.

Lab 2: Secure Session Management Practices

Exercise 1: Configuring Secure Session Cookies

Modify Server-Side Configuration:

Impact of These Changes on Session Security

The PHP session security measures implemented in the code significantly enhance protection against common web application threats such as **session hijacking, fixation, and cross-site request forgery (CSRF)**. Here's a breakdown of how each change contributes to security:

1. Secure Cookie Parameters (session_set_cookie_params())

Setting	Impact on Security

lifetime => 0	The session expires when the browser is closed, preventing long-lived sessions that could be stolen.

Setting	Impact on Security
path => '/'	Ensures the session cookie is accessible across the entire domain, preventing path-based attacks.
domain => 'yourdomain.com'	Restricts the session cookie to a specific domain, mitigating **session fixation attacks** on subdomains.
secure => true	Ensures cookies are only sent over **HTTPS**, preventing **man-in-the-middle (MITM) attacks**.
httponly => true	Prevents JavaScript from accessing session cookies, mitigating **cross-site scripting (XSS) attacks**.
samesite => 'Strict'	Blocks cookies from being sent with cross-site requests, reducing **CSRF vulnerabilities**.

2. Regenerating Session IDs (session_regenerate_id(true))

- **Prevents session fixation:** Attackers cannot force users to use a pre-set session ID.
- **Mitigates session hijacking:** If an attacker somehow gains access to a session ID, regenerating it after login renders the old session ID useless.
- **Enhances security after authentication:** A new session ID is assigned after login, making it harder for attackers to exploit stolen credentials.

How These Changes Mitigate Common Vulnerabilities

Vulnerability	Mitigation
Session Hijacking	Secure cookies and session regeneration ensure attackers cannot steal and reuse session IDs.
Session Fixation	Regenerating session IDs prevents attackers from fixing a victim's session ID before login.

CSRF Attacks The SameSite=Strict attribute blocks session cookies from being used in cross-site requests.

Vulnerability	Mitigation
XSS Attacks	The httponly flag prevents malicious JavaScript from accessing session cookies.
MITM Attacks	Enforcing HTTPS (secure=true) prevents attackers from intercepting session data over unencrypted connections.

Final Reflection

These **session security controls** significantly improve the integrity and confidentiality of user sessions. However, additional measures such as **session timeout enforcement, IP address binding, and multi-factor authentication (MFA)** can further enhance security.

Exercise 2: Implementing Session Expiration and Invalidation

1. Understand Session Expiration:

Session expiration is a critical aspect of session management in web applications. It determines how long a user's session remains active before it is automatically terminated. Properly configuring session expiration is essential for balancing security and usability.

What is a Session?

- A session is a way to store user-specific data on the server between HTTP requests.
- When a user logs in, a unique session ID is generated and stored in a cookie on the client side. This session ID is used to associate the user with their session data on the server.

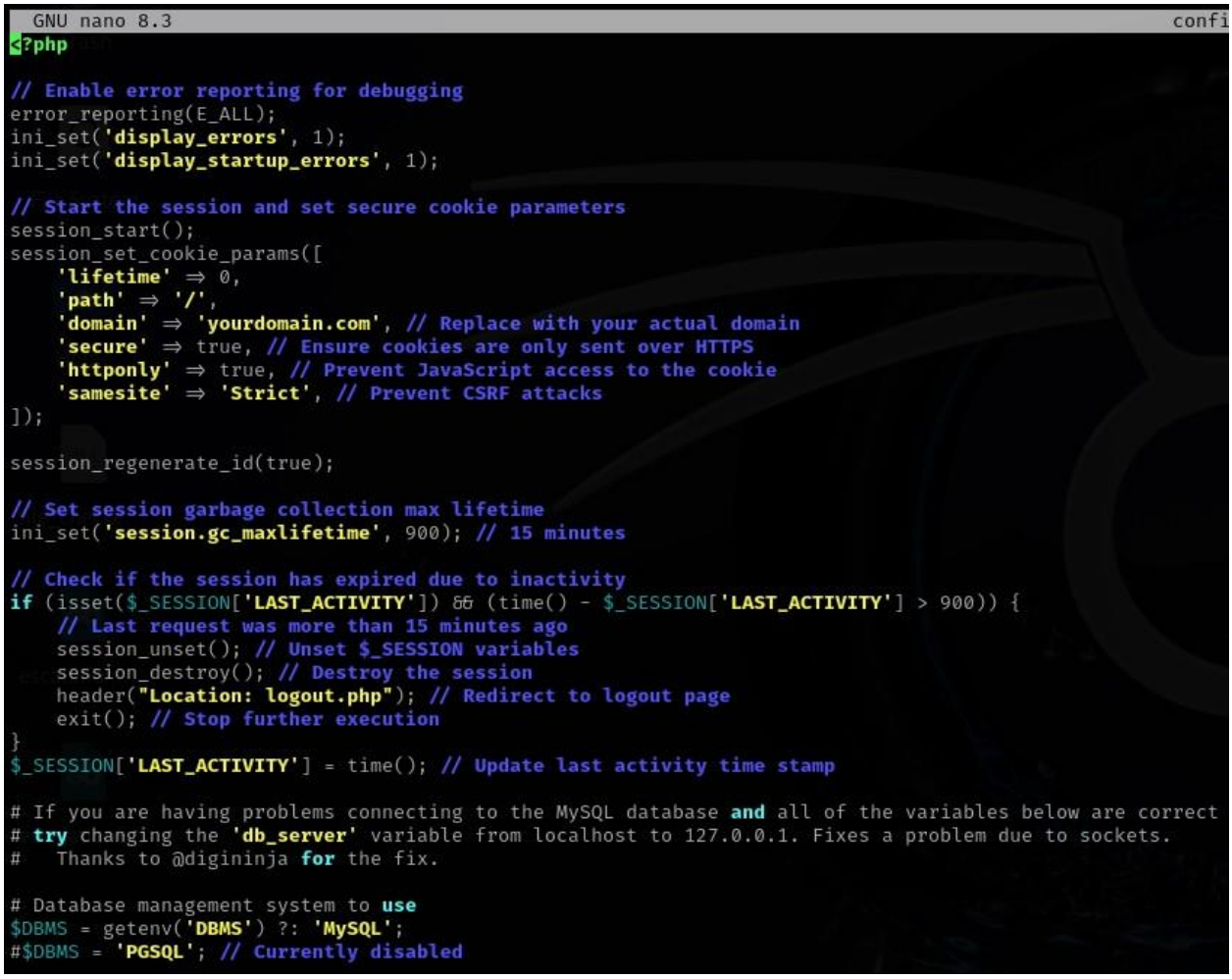
Why is Session Expiration Important?

1. **Security**:
    - Prevents unauthorized access to a user's session if the session ID is compromised.

- Reduces the risk of session hijacking and session fixation attacks.

2. **Resource Management**:
    - Frees up server resources by terminating inactive sessions.
    - Prevents the server from being overwhelmed by stale sessions.
3. **User Experience**:
    - Balances security with usability by allowing users to remain logged in for a reasonable amount of time.

2. Set Session Lifetime:



1. Force Logout on Expiration:

```
 GNU nano 8.3
<?php

define( 'DVWA_WEB_PAGE_TO_ROOT', '' );
require_once DVWA_WEB_PAGE_TO_ROOT . 'dvwa/includes/dvwaPage.inc.php';

dvwaPageStartup( array( ) );

if( !dvwaIsLoggedIn() ) {       // The user shouldn't even be on this page
        // dvwaMessagePush( "You were not logged in" );
        dvwaRedirect( 'login.php' );
}

dvwaLogout();
dvwaMessagePush( "You have logged out" );
dvwaRedirect( 'login.php' );

session_start();
session_unset(); // Unset all session variables
session_destroy(); // Destroy the session
header("Location: index.php"); // Redirect to login page
exit();
?>
```

2. Reflection:

How Session Expiration Contributes to Overall Application Security

Session expiration is a critical security mechanism that ensures user sessions do not remain active indefinitely, reducing the risk of unauthorized access. Here's how it enhances security:

1. Prevents Session Hijacking
    - If an attacker gains access to a valid session ID, session expiration limits the window of opportunity to misuse it.
2. Mitigates Session Fixation Attacks
    - If a session ID is fixed before login, expiration ensures the session is not usable indefinitely.
3. Reduces Risk of Unattended Sessions
    - Users who leave their devices unattended (e.g., in public places) without logging out are automatically signed out after a set time.
4. Protects Against Stale Sessions

- Ensures old sessions do not persist, preventing unauthorized re-use after a logout or crash.

Challenges in Enforcing Session Expiration

1. User Experience Disruptions
   - Strict session timeouts may log out users unexpectedly, causing frustration, especially during long forms or transactions. ○ Solution: Implement session renewal mechanisms on activity detection.
2. Balancing Security and Usability
   - Short timeouts improve security but may inconvenience users who frequently need to log in. ○ Solution: Allow users to opt-in for "Remember Me" sessions with strict security controls.
3. Potential Data Loss
   - Users filling out forms or working on critical tasks may lose progress if their session expires mid-action. ○ Solution: Use AJAX-based session keep-alive techniques to refresh sessions only when users are active.
4. Increased Authentication Overhead
   - Frequent reauthentication can slow down workflows, especially for business users.
   - Solution: Implement adaptive session expiration, where critical actions require re-authentication, but browsing remains active.
5. Bypassing Expiration via Persistent Cookies
   - If users rely on persistent cookies to auto-login, attackers with access to a stolen device can still gain entry.
   - Solution: Combine session expiration with device verification and MFA.

Best Practices for Effective Session Expiration

- Set reasonable timeouts (e.g., 15-30 minutes for sensitive applications, 60+ minutes for less critical ones).
- Use rolling expiration (extend session only on user activity, not idle time).
- Force logout on inactivity but allow users to save work.
- Log session activity to detect unusual patterns.

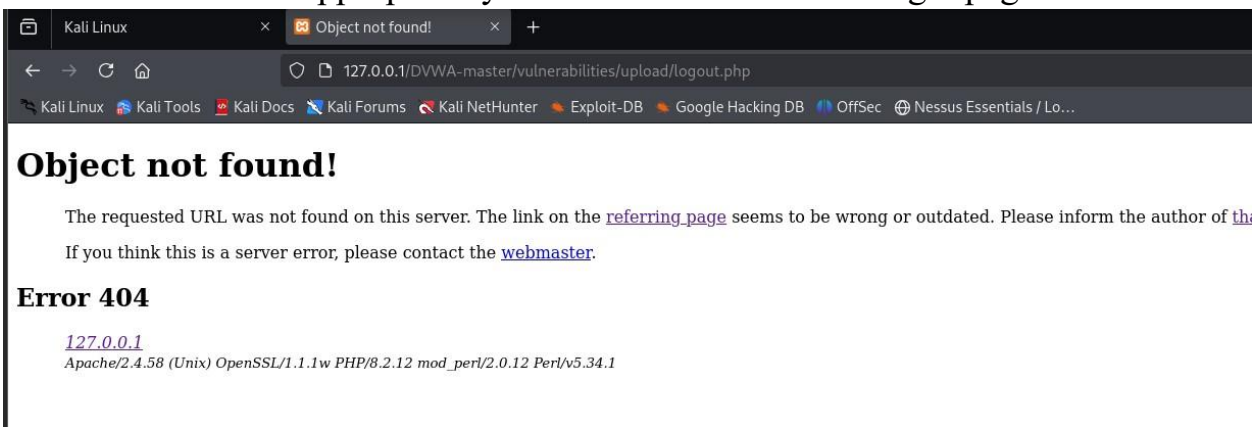
By balancing security with usability, session expiration remains a powerful tool in protecting web applications from unauthorized access while maintaining a smooth user experience.

Exercise 3: Testing Session Management Controls

1. Simulate an Attack:

2. Evaluate Security Measures:

Yea it denied access appropriately and redirected me to the login page.



3. Reflection:

Effectiveness of the Session Management Controls in Place

Effectiveness of the Session Management Controls in Place

The session management controls implemented in your application enhance security by protecting user sessions from common attacks such as session hijacking, session fixation, and cross-site request forgery (CSRF). The following measures contribute to their effectiveness:

1. Session Cookie Security

- Secure flag (secure => true) ensures that cookies are only sent over HTTPS, preventing interception via man-in-the-middle (MITM) attacks.
- HTTPOnly flag (httponly => true) protects session cookies from being accessed by JavaScript, mitigating cross-site scripting (XSS) attacks.
- SameSite Attribute (samesite => 'Strict') prevents cookies from being sent with cross-site requests, reducing CSRF risks.

2. Session Regeneration
    - session_regenerate_id(true); ensures that a new session ID is generated after login, preventing session fixation attacks.
3. Session Expiration & Forced Logout
    - Setting lifetime => 0 ensures the session expires when the browser is closed, reducing the window for session hijacking. ○ Logout function (session_unset(); session_destroy();) ensures that inactive sessions are properly terminated.

Additional Measures to Strengthen Security

1. Implement Session Timeout ○ Set an inactivity timeout using a timestamp:
    - This limits session duration, reducing the risk of stolen session reuse.
2. Use Strong Session IDs ○ Enable session entropy to generate stronger session IDs: ○ This prevents predictable session IDs, making brute-force attacks harder.
3. Bind Session to IP Address and User-Agent
    - Prevent session hijacking by verifying the client's IP and browser fingerprint:
4. Database-Backed Session Management
    - Store session data in a database instead of flat files to prevent session theft:

5. Implement Multi-Factor Authentication (MFA)
    - Require a second authentication factor (OTP, email verification) after login to prevent unauthorized access.
6. Monitor and Log Sessions
    - Keep logs of active sessions and detect unusual login patterns to prevent account takeovers.

By implementing these additional controls, session security can be significantly strengthened, reducing risks associated with unauthorized access, session hijacking, and CSRF attacks.