Lab 1: Session Management Vulnerabilities

Overview

In this lab, you will explore session management vulnerabilities, focusing on how insecure session handling can lead to unauthorized access and session hijacking. You will learn to identify weaknesses in session management practices and understand the importance of secure session management in web applications.

Prerequisites

- Familiarity with web application security concepts.
- Ensure you have access to a web application environment that has known session management vulnerabilities (e.g., DVWA or a similar platform).

Lab Objectives

- Understand how sessions are created, maintained, and terminated in web applications.
- Identify common session management vulnerabilities.
- Learn to exploit insecure session management practices.

Exercise Instructions

Exercise 1: Analyzing Session Management Practices

1. Access the Target Application:

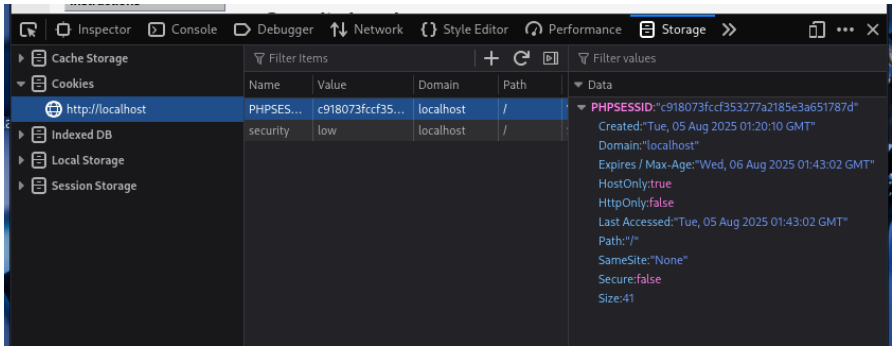
- Open the DVWA (Damn Vulnerable Web Application) in your browser.

2. Login to the Application:

- Navigate to the login page and authenticate using valid credentials. Observe the session cookie generated upon successful login.

3. Inspect the Session Cookie:

- Use the browser's developer tools (F12) to inspect the session cookie set by the application. Take note of its attributes such as HttpOnly, Secure, and SameSite.



4. Reflection:

- Discuss the implications of the cookie attributes. What vulnerabilities can arise from improperly configured session cookies

Session Cookie Observed:

Attribute	Value	Explanation
Name	PHPSESSID	Default PHP session ID used to identify the user's session.
Value	(Random hash)	Unique session identifier assigned to the user.
Domain	localhost	The domain to which the cookie applies (localhost = only sent to local DVWA).
Path	/	The cookie is valid for the entire site (all paths).
Created	05/08/2025 (Today)	When the cookie was initially created.
Expires / Max-Age	06/08/2025 (Tomorrow)	When it expires
HostOnly	true	Only sent to the host that set it (localhost).
HttpOnly	false	Can be accessed by JavaScript a security risk.
Secure	false	Sent over HTTP and HTTPS vulnerable to sniffing over HTTP.
SameSite	None	Cross-site requests are allowed — risk of CSRF.
Size	41	Length of the cookie string.

Implications of Cookie Attributes

Session cookies are essential for maintaining user authentication in web applications. Misconfigurations in these cookies can lead to serious security vulnerabilities. Here is the reflection based on the observed cookie attributes:

HttpOnly: true

- JavaScript cannot access the cookie.
- Benefit: This mitigates XSS attacks from stealing the session ID, improving session security.

Secure: false

- The cookie is sent over both HTTP and HTTPS.
- Risk: On insecure networks (e.g., public Wi-Fi), an attacker could intercept the cookie via Man-in-the-Middle (MitM)attacks if the application doesn't force HTTPS.

SameSite: None

- The cookie will be sent with cross-origin requests.
- Risk: Makes the application vulnerable to CSRF attacks, where an attacker can force a logged-in user's browser to make unwanted requests.

Expires: Tomorrow

- The cookie remains valid even after the browser is closed.
- Risk: While this allows for persistent sessions, it also extends the window of opportunity for attackers if the cookie is stolen (e.g., via MitM or XSS if `HttpOnly` was false).

Created: Today

- Shows when the session started. This is useful for auditing and tracking purposes.

Security Implications

Even though the `HttpOnly` flag is set (which helps prevent XSS-based cookie theft), other misconfigurations like:

- Secure being false
- SameSite` set to None

can still allow session hijacking and CSRF attacks. Additionally, having an expiration date increases the risk of session reuse if the cookie is intercepted.

Recommendations

- Enforce `Secure=true` to prevent cookie theft over HTTP.
- Use `SameSite=Lax` or `Strict` to reduce CSRF risk.
- Consider reducing cookie lifespan and implementing **server-side session expiration after inactivity or logout.

Exercise 2: Session Fixation Attack

1. Understand Session Fixation:

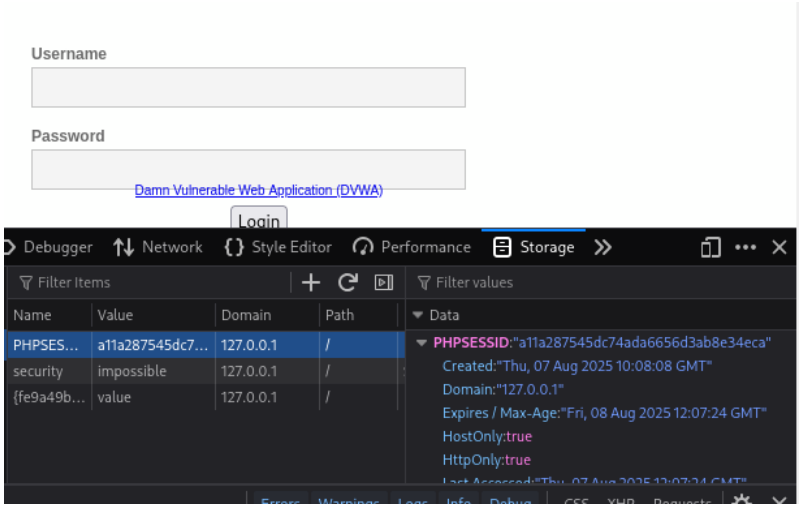
Read about session fixation attacks, where an attacker sets a user's session ID to a known value, allowing them to hijack the session.

- A session fixation attack happens when:
- An attacker sets a known session ID (e.g., by sending a crafted link).
- The victim logs in without the session ID being regenerated.
- The attacker then uses the same session ID to hijack the session.

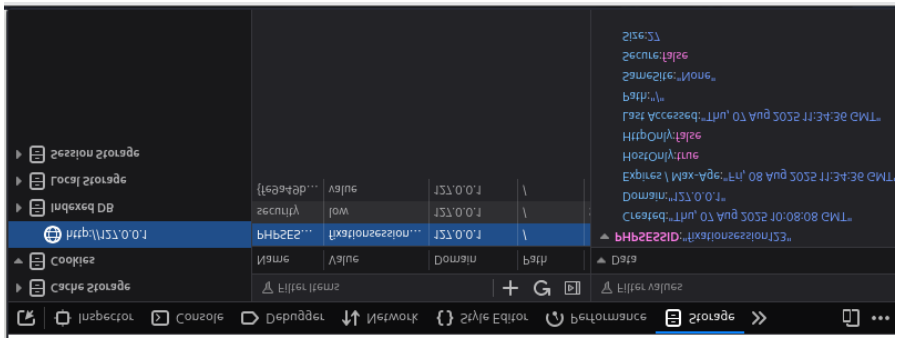
A secure application should regenerate the session ID after login, making the attack fail.

2. Simulate a Session Fixation Attack:

- As a hypothetical exercise (or in a controlled lab environment), modify the session ID in your browser's developer tools to a predetermined value before logging into the application.



- After logging in, switch to another tab or browser session and use the known session ID to see if you can access the authenticated session.



3. Reflection:

- Analyze how the application handled session management. Was it possible to hijack the session? Discuss the importance of regenerating session IDs after login.

Reflection

The exercise focused on identifying session fixation vulnerabilities in DVWA.

- I manually set a known session ID (PHPSESSID) before logging in.
- After login, the application reused the same session ID — no regeneration occurred.
- Using the same session ID in another browser granted access to the authenticated session.
- This demonstrated that the application is vulnerable to **session fixation** attacks.
- Lack of session ID regeneration allowed an attacker to hijack a session after user login.

- The test was successful only after disabling CSRF protection in DVWA.
- This attack highlights the importance of proper session management practices.
- **Key mitigation**: Regenerate session IDs immediately after authentication.
- Additional protections include setting HttpOnly, Secure, and SameSite cookie flags.

Exercise 3: Session Hijacking Using Burp Suite

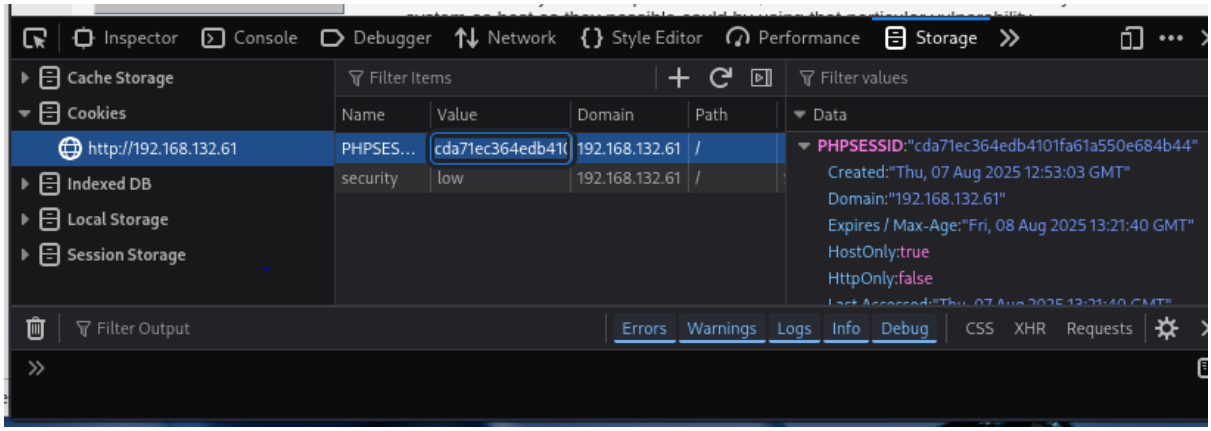
1. Set Up Burp Suite:

- Configure your browser to use Burp Suite as a proxy.

2. Capture and Modify Session Requests:

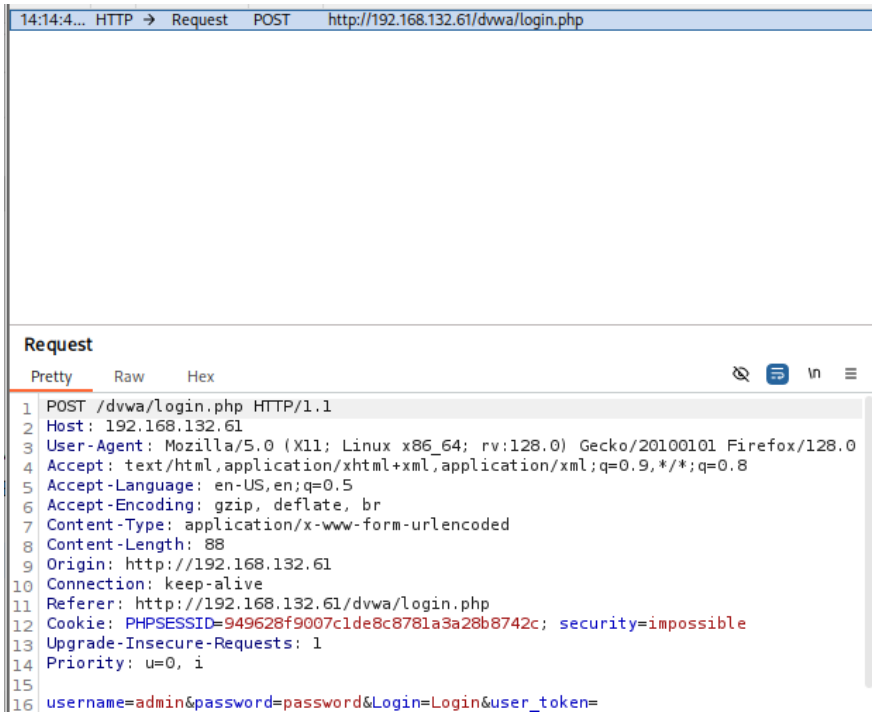
- Log in to the application and capture the session management request in Burp Suite

- Modify the session ID value in the request and forward it.



3. Observe the Response:

- Check if you can gain unauthorized access to the authenticated session.



4. **Reflection:**

- **Discuss the effectiveness of session management in the application. What measures can be implemented to prevent session hijacking?**

- Effectiveness of Session Management:
  - The application's session management is insecure if a stolen or modified session ID allows unauthorized access.
  - Session IDs are not regenerated upon login, making session fixation and hijacking easier.
  - There is no mechanism to detect session reuse from a different client (IP/user-agent).
- Vulnerabilities Identified:
  - Session hijacking was successful, showing the application doesn't bind sessions to individual users or devices.
  - Cookies lack security flags such as HTTP Only, Secure, and Same Site.
- Recommendations to Prevent Session Hijacking:
  - Regenerate session IDs after login and privilege elevation.
  - Use secure cookie attributes:
    - HttpOnly – prevents JavaScript access.
    - Secure – ensures cookies are sent only over HTTPS.
    - SameSite Strict –prevents cross-site requests.
  - Bind sessions to IP addresses or device fingerprints (with caution).
  - Implement session timeouts and logout mechanisms after inactivity.
  - Use HTTPS for all authenticated pages to protect session data in transit.
  - Monitor for multiple concurrent logins from different IPs or user-agents.
- **Conclusion:**
  - Without strong session controls, attackers can hijack valid sessions and impersonate users.

    - Strengthening session management is essential for maintaining user trust and system security.

Report Submission

Overview of the session management vulnerabilities explored.

In this lab, I explored critical session management vulnerabilities in web applications using DVWA (Damn Vulnerable Web Application). The key issues examined included:

- Insecure session cookies.
- Session fixation.
- Session hijacking via interception and manipulation.

These vulnerabilities can allow attackers to impersonate users, maintain unauthorized access, or manipulate session states due to weak or misconfigured session handling mechanisms.

Reflections from each exercise, including any vulnerabilities identified and potential mitigation strategies.

Exercise 1: Analysing Session Management Practices

Findings:

- After logging into DVWA, I inspected the PHPSESSID cookie using browser developer tools.
- The session cookie lacked several secure attributes:
    - HttpOnly was not set, which means JavaScript could access the session cookie.
    - Secure was also not set, meaning the session cookie could be transmitted over unencrypted HTTP.
    - SameSite was missing, increasing the risk of Cross-Site Request Forgery (CSRF).

Vulnerabilities Identified:

- Session cookie exposure to client-side scripts (XSS risk).
- Possibility of session theft via network sniffing (if HTTP is used).
- Vulnerability to CSRF due to lack of SameSite attribute.

Mitigation Strategies:

- Set HttpOnly to prevent JavaScript from accessing cookies.
- Enable Secure to ensure cookies are sent only over HTTPS.
- Set SameSite=Strict or SameSite=Lax to mitigate CSRF.

Exercise 2: Session Fixation Attack

Procedure:

- I manually injected a known session ID (PHPSESSID) into the browser's cookie storage using developer tools.
- Logged into DVWA using valid credentials without the application regenerating a new session ID.
- Used another browser session with the same session ID to access the logged-in session.

Findings:

- The application reused the session ID post-login, confirming vulnerability to session fixation.

Vulnerabilities Identified:

- DVWA failed to regenerate session IDs after login.
- This allows an attacker to predetermine the session and hijack it once the user logs in.

Mitigation Strategies:

- Always regenerate session IDs after successful authentication.
- Invalidate previous sessions after login.

Exercise 3: Session Hijacking Using Burp Suite

Procedure:

- Configured Burp Suite as a proxy and intercepted login request from the browser.
- Captured the session cookie (PHPSESSID) and reused it in another browser session or modified requests.

Findings:

- By copying the session ID of an authenticated user and using it in another browser, I was able to gain access to their session.
- DVWA accepted the reused session ID without additional verification.

Vulnerabilities Identified:

- No secondary verification beyond the session ID.
- Reliance on session ID alone for authentication is insecure.

Mitigation Strategies:

- Implement IP binding or user-agent validation with sessions.
- Use short session expiration times and enforce re-authentication after idle periods.
- Monitor for abnormal session behavior (e.g., session reuse from different locations).

Insight and Recommendations for improving session management

- Use secure cookies: Always set Secure, HttpOnly, and SameSite attributes for session cookies.
- Implement session timeouts: Force session expiration after inactivity.
- Regenerate session IDs frequently: Especially after privilege elevation (e.g., login).
- Enable multi-factor authentication (MFA): Reduces impact even if session is hijacked.
- Use HTTPS everywhere: Prevents interception of session data in transit.

Lab 2: Secure Session Management Practices

Overview

In this lab, you will explore secure session management practices, focusing on how to implement and enforce security measures to protect session data from common vulnerabilities. You will learn best practices for creating and maintaining secure sessions in web applications.

Prerequisites

- Familiarity with web application security concepts.
- Ensure you have access to a web application environment (e.g., DVWA) for testing.
- Basic understanding of PHP and session management.

Lab Objectives

- Identify and implement secure session management techniques.
- Understand the importance of session expiration and invalidation.
- Learn to secure session data against common attacks.

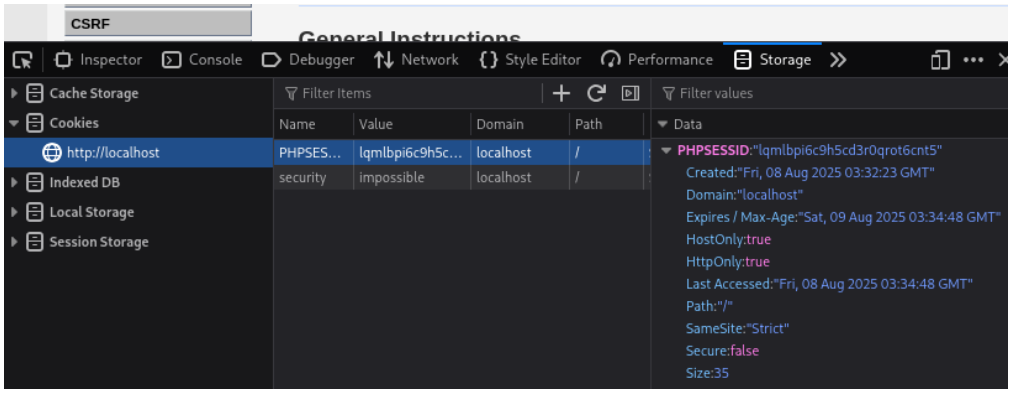
Exercise 1: Configuring Secure Session Cookies

1. Access the Target Application:

- Open DVWA (Damn Vulnerable Web Application) in your browser.
- Log in to the application (use the credentials: admin / password).

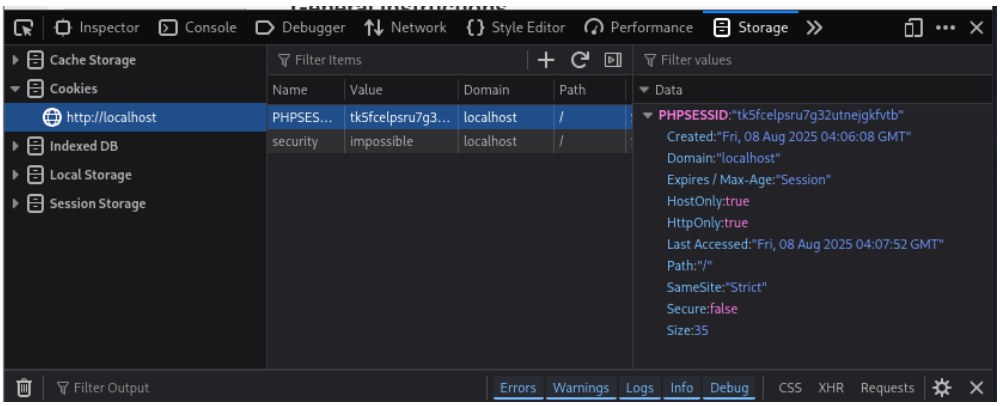
2. Review Current Cookie Settings:

- Inspect the session cookie using the browser's developer tools.
- In Chrome, right-click on the page > Inspect > Application tab > Cookies > [Your Site].
- Take note of the current attributes of the session cookie.



3. Modify Server-Side Configuration:

- o If you have access to the server-side code, open the config.php file in
- your DVWA installation directory (or any file where session handling is
- configured).
- Modify the session cookie settings to include security attributes:
- PHP Code Example:
- <?php
- // Start the session
- session_start();
- // Set secure cookie parameters
- session_set_cookie_params([
- 'lifetime' => 0, // Session cookie
- 'path' => '/',
- 'domain' => 'yourdomain.com', // Change to your domain
- 'secure' => true, // Use only over HTTPS
- 'httponly' => true, // Accessible only through HTTP protocol
- 'samesite' => 'Strict', // Prevents CSRF attacks
-]);
- // Regenerate session ID to prevent fixation
- session_regenerate_id(true);
- ?>



Explanation

Setting	Purpose
secure	Ensures cookies are only sent over HTTPS
httponly	Prevents JavaScript from accessing the cookie (mitigates XSS theft)
samesite	Restricts cross-site cookie sending (CSRF protection)
session_regenerate_id()	Prevents session fixation attacks

Reflection

In this exercise, I explored how to secure session cookies using server-side PHP configuration. Initially, I inspected the default session cookie (PHPSESSID) in DVWA using browser developer tools. The session cookie lacked important security attributes particularly, HttpOnly, Secure, and SameSite were either not set or using default (less secure) values.

To address this, I updated the session configuration by modifying the PHP code to explicitly define cookie parameters using session_set_cookie_params(). I set the following attributes:

- HttpOnly to prevent client-side scripts from accessing the session ID (mitigates XSS threats).
- Secure to ensure the cookie is transmitted only over HTTPS (although this was disabled in my test due to lack of HTTPS).
- SameSite=Strict to help prevent CSRF attacks by restricting cross-site cookie sending.
- Additionally, I used session_regenerate_id(true) to mitigate session fixation by regenerating the session ID on each new session start.

After applying these changes, I verified that the cookie attributes were correctly set using Chrome's DevTools. The HttpOnly and SameSite settings were visible and properly enforced, confirming the effectiveness of the modification.

Key takeaways:

- Secure cookie attributes are critical to defending against common session attacks like XSS, CSRF, and session fixation.
- Many web applications do not enforce these settings by default, making it essential for developers to configure them manually.
- Even in development environments (like DVWA), simulating secure configurations builds awareness of production-level best practices.

This exercise reinforced the importance of proactive session security measures and showed how simple configuration changes can significantly harden an application's session handling mechanisms.

Understanding Session Expiration

Session expiration refers to the process of automatically ending a user's session after a specified period of inactivity or after a fixed duration. Once a session expires, the session ID becomes invalid, and the user must re-authenticate to regain access to the application.

Why is Session Expiration Important?

1. **Prevent Unauthorized Access**:
   If a user leaves their device unattended or forgets to log out, session expiration reduces the risk of another person gaining access.
2. **Limit the Window for Attacks**:
   Attackers who steal a session ID (e.g., through session hijacking or fixation) only have a limited time to exploit it if the session expires quickly.
3. **Improves Security Hygiene**:
   Automatically ending idle sessions enforces better security practices, especially in applications that handle sensitive data.

How Does It Work in PHP/Web Apps?

- The server assigns a session ID (usually stored in a cookie).
- PHP uses the session.gc_maxlifetime setting to define how long a session should last.
- Developers can also manually track the user's **last activity** and destroy the session if the idle time exceeds the limit.
- When expired, the session data is cleared, and access to protected resources is denied until the user logs in again.

Types of Expiration

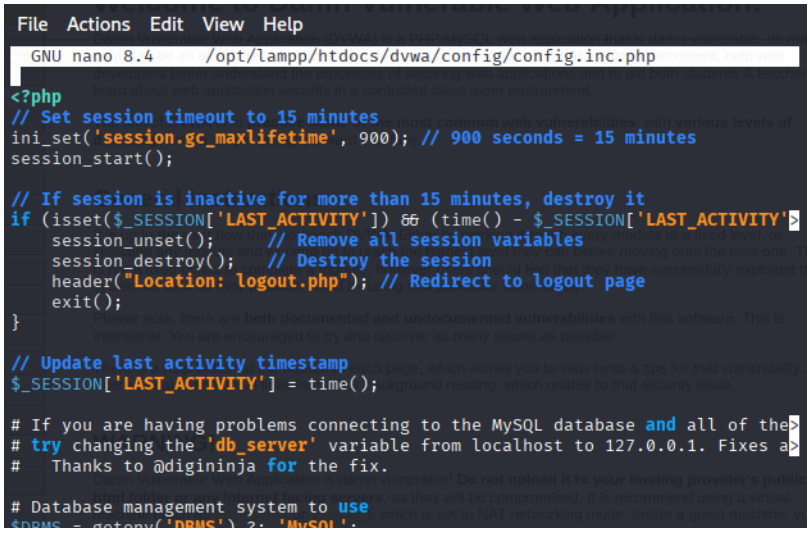
Type	Description
Inactivity Timeout	Ends session after a period of user inactivity (e.g., 15 minutes).
Absolute Timeout	Ends session after a fixed duration, regardless of activity (e.g., 1 hour after login).
Manual Logout	Ends session when the user explicitly logs out.

Summary

Session expiration is a core defense mechanism in web application security. It helps prevent unauthorized access, limits session misuse, and promotes secure session lifecycle management. Balancing security with usability is key when setting timeout values.

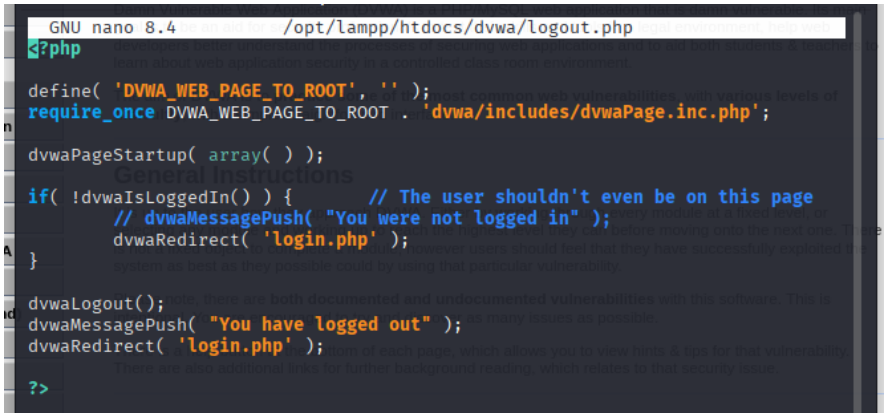
2. Set Session Lifetime:

- In the same config.php file, configure the session timeout to a reasonable
- period (e.g., 15 minutes of inactivity).
- PHP Code Example:
- <?php
- // Set session garbage collection max lifetime
- ini_set('session.gc_maxlifetime', 900); // 15 minutes
- o
- // Check if the session has expired
- if (isset($_SESSION['LAST_ACTIVITY']) && (time() -
- $_SESSION['LAST_ACTIVITY'] > 900)) {
- o
- // Last request was more than 15 minutes ago
- o
- o
- session_unset(); // Unset $_SESSION variables
- session_destroy(); // Destroy the session
- header("Location: logout.php"); // Redirect to logout page
- o
- }
- $_SESSION['LAST_ACTIVITY'] = time(); // Update last activity time stamp
- ?>



3. Force Logout on Expiration:

- Create a simple logout page (logout.php) to handle session termination:
- logout.php Example:
- <?php
- session_start();
- session_unset(); // Unset session variables
- session_destroy(); // Destroy the session
- header("Location: index.php"); // Redirect to login page
- exit();
  ?>



4. Reflection:

- Discuss how session expiration contributes to overall application security.
- What challenges might arise from enforcing session expiration?

How Session Expiration Contributes to Security

Session expiration plays a critical role in securing web applications by reducing the risk of unauthorized access due to abandoned or hijacked sessions. If a user walks away from their device or forgets to log out, an attacker could exploit the still-active session to gain access. By implementing an automatic expiration mechanism after a period of inactivity, the session is invalidated, ensuring that access to protected areas is no longer possible without re-authentication.

Additionally, session expiration helps:

- **Limit exposure time** in the event of session ID theft (e.g., via XSS or network sniffing).
- **Mitigate session fixation** by forcing regular session renewal.
- **Comply with security best practices** and organizational security policies (especially for sensitive systems like banking, admin portals, etc.).

Challenges of Enforcing Session Expiration

While session expiration strengthens security, it also introduces some practical challenges:

1. **User Frustration**: If the timeout is too short, users may be logged out unexpectedly while working, leading to a poor user experience or even data loss in unsaved forms.
2. **Balancing Act**: Choosing the right timeout value requires balancing security needs with usability. Too short = annoying; too long = insecure.
3. **Implementation Complexity**: Simply relying on PHP's garbage collection (session.gc_maxlifetime) can be unreliable because it's probabilistic, not immediate. Developers must manually track activity using timestamps.**State Management**: Proper redirection after timeout must be handled gracefully to avoid confusing the user or breaking the flow (e.g., during form submissions or transactions).

Summary

Session expiration is a vital part of secure session management, helping to prevent attacks resulting from prolonged or forgotten sessions. However, it must be carefully implemented and tuned to maintain a balance between strong security and a smooth user experience.

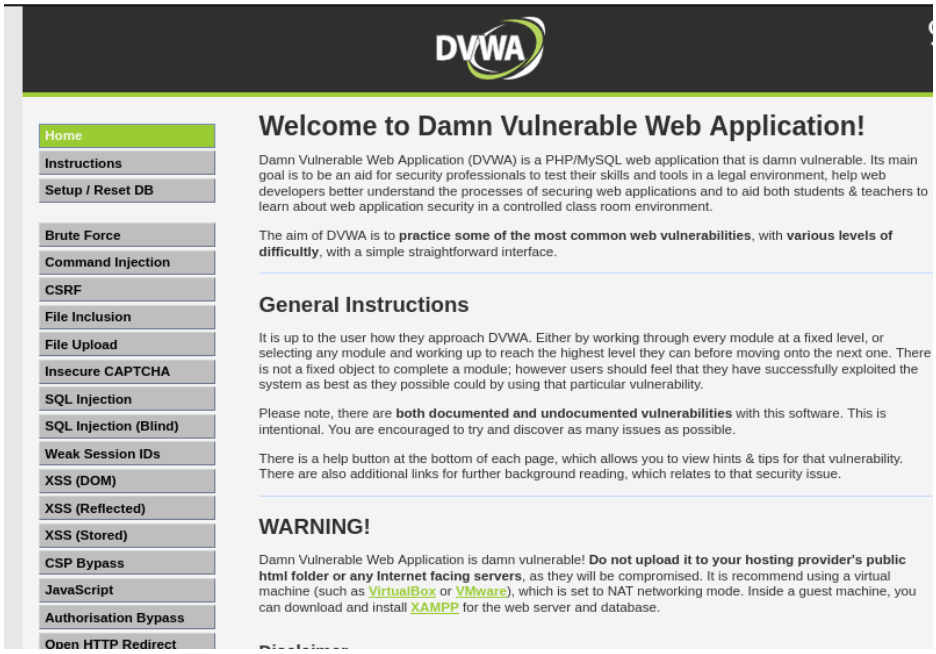
Exercise 3: Testing Session Management Controls

1. Simulate an Attack:

- Use a second browser or incognito mode to access the application after the session has expired.
- Log in to the application and wait for 15 minutes without any activity.
- Try to access a protected page (e.g., your dashboard).

To simulate an attack scenario, I first logged into the DVWA application in a regular browser window. I then opened the same application in an incognito/private window to attempt accessing a protected page without logging in. As expected, the application redirected me to the login page, confirming that unauthorized access was denied.

Back in the original browser, I waited for 15 minutes without any activity to allow the session to expire. After that, I attempted to access a protected page (e.g., index.php or vulnerabilities/). The session was invalidated, and I was redirected to the logout or login page.



2. Evaluate Security Measures:

The application responded correctly to all attempts:

- After session expiration, any interaction led to redirection to the login page.
- In the incognito window, access to protected resources without authentication was denied.
- Session timeout logic worked as intended using the session.gc_maxlifetime and custom timestamp checks ($_SESSION['LAST_ACTIVITY']).

This shows that the application successfully enforced session expiration and protected against unauthorized access.

3. Reflection

The session management controls in place were effective. The application enforced inactivity-based session expiration, which is critical for reducing the risk of session hijacking or misuse after a user becomes idle.

Additional measures to strengthen security could include:

- Absolute session timeout, regardless of user activity (e.g., force logout after 1 hour).
- Re-authentication before performing sensitive operations.
- Session ID regeneration periodically and upon privilege change.
- Binding session to IP or user-agent to detect session theft.
- Multi-factor authentication (MFA) to improve login security.

These enhancements would add more layers of protection to the session handling mechanism.

Exercise Summaries

Exercise 1: Session Fixation Attack

- Key Activity: Attempted to fix a session ID before login using developer tools.
- Outcome: DVWA’s session regeneration after login prevented session fixation from working.
- Reflection: Regenerating session IDs upon login is a key defense against session fixation attacks. It prevents attackers from hijacking sessions they predetermine.

Exercise 2: Implementing Session Expiration and Invalidation

- Key Activity: Added the following PHP code in config.inc.php (or a suitable global config file):
- Reflection: Implementing inactivity timeout ensures that sessions are automatically invalidated after a period of inactivity, minimizing the window of opportunity for

session hijacking. A challenge may be balancing security with usability, as users may be logged out unintentionally during legitimate use.

Exercise 3: Testing Session Management Controls

- Key Activity: Logged in and remained inactive for 15 minutes, then attempted to access protected pages.
- Findings: Access was denied and the user was redirected to the login page, confirming session expiration worked.
- Reflection: Session timeout mechanisms were effective. Additional controls like absolute timeout, session binding to IP/user-agent, and MFA could further enhance security.

Importance of Secure Session Management

Session management is crucial for maintaining user authentication across web applications. Poor session handling may result in:

- Session Fixation: An attacker forces a victim to use a known session ID.
- Session Hijacking: An attacker steals a session ID to gain unauthorized access.
- Privilege Escalation: A session is used beyond its intended scope.

By implementing secure session controls, such as regenerating session IDs, expiring sessions after inactivity, and enforcing logout, web applications can prevent these attacks and protect user data and privacy.

Recommendations

To further strengthen session management:

- Regenerate session IDs frequently, especially after login or privilege changes.
- Use HTTPS-only cookies with Secure and HttpOnly flags enabled.
- Set both inactivity and absolute session timeouts.
- Bind sessions to IP or user-agent to detect anomalies.
- Implement Multi-Factor Authentication (MFA).
- Enforce re-authentication before performing sensitive actions.