**Experiment 9**

**Aim**: Session Management for Web Application.

**Theory:**

**Session management** is a critical aspect of web application security, handling the process of maintaining the state of user interactions across multiple requests. In web applications, HTTP is a stateless protocol, meaning that each request from a client (e.g., a web browser) to a server is treated as an independent transaction. To create a seamless and secure user experience, session management techniques are used to identify and authenticate users while keeping track of their actions.

### **Key Concepts in Session Management**

1. **Session**:
   * A session represents a series of interactions between a user and a web application during a specific time frame. It begins when a user accesses the application and ends when they log out or the session expires.
2. **Session Identifier (Session ID)**:
   * A unique identifier assigned to each session, typically a long, random string. The session ID allows the server to retrieve the session data associated with that user. It is crucial to ensure that the session ID is unpredictable to prevent session hijacking attacks.
3. **State Management**:
   * Since HTTP is stateless, web applications must implement state management techniques to store and manage session data on the server. This can include user preferences, authentication information, and other data necessary for maintaining the user experience.

### **Session Management Process**

1. **Session Creation**:
   * When a user logs in or starts interacting with the application, a session is created on the server. A unique session ID is generated and sent to the user's browser, often stored in a cookie.
2. **Session Storage**:
   * The server stores session data (e.g., user credentials, preferences, shopping cart contents) associated with the session ID. This data can be stored in memory, databases, or distributed caches.
3. **Session Validation**:
   * For each subsequent request made by the user, the web application validates the session ID sent by the browser. If the session ID is valid, the application retrieves the corresponding session data to fulfill the request.
4. **Session Termination**:
   * Sessions can be terminated in several ways:
     + **User Logout**: When a user explicitly logs out, the session is destroyed, and the session ID becomes invalid.
     + **Timeout**: Sessions often have a timeout period. If a user is inactive for a specified duration, the session is automatically terminated to prevent unauthorized access.
     + **Server-side termination**: An administrator can also terminate sessions manually if necessary.

### **Session Management Techniques**

1. **Cookies**:
   * Cookies are commonly used to store session IDs in the user's browser. They can be set with specific attributes:
     + **Secure**: Ensures that the cookie is sent only over HTTPS connections.
     + **HttpOnly**: Prevents JavaScript from accessing the cookie, reducing the risk of cross-site scripting (XSS) attacks.
     + **SameSite**: Controls whether the cookie is sent along with cross-site requests, helping to prevent cross-site request forgery (CSRF) attacks.
2. **URL Rewriting**:
   * If cookies are disabled, session IDs can be appended to URLs (e.g., example.com/page?sessionid=abc123). However, this method can expose session IDs in server logs and browser history, leading to security risks.
3. **Token-Based Authentication**:
   * In modern web applications, especially those using APIs, token-based authentication (e.g., JSON Web Tokens, or JWTs) is increasingly common. Upon authentication, the server issues a token that the client includes in subsequent requests, allowing for stateless session management.

### **Security Considerations**

1. **Session Hijacking**:
   * Attackers may attempt to steal a user's session ID to impersonate them. To mitigate this risk:
     + Use secure, unpredictable session IDs.
     + Implement secure cookie attributes (Secure, HttpOnly, SameSite).
     + Monitor for unusual activity or IP address changes.
2. **Cross-Site Scripting (XSS)**:
   * XSS vulnerabilities can allow attackers to execute scripts in a user's browser, potentially stealing session IDs. To protect against XSS:
     + Sanitize and validate user input.
     + Use content security policies (CSP) to restrict script execution.
3. **Cross-Site Request Forgery (CSRF)**:
   * CSRF attacks trick users into performing actions without their consent by sending unauthorized requests. To prevent CSRF:
     + Use anti-CSRF tokens that are included in forms and verified by the server.
     + Implement SameSite cookie attributes.
4. **Session Expiration**:
   * To minimize risks, implement session timeouts that log users out after a period of inactivity. Users should also be given the option to log out manually.
5. **Session Fixation**:
   * An attacker can fixate a session ID before the user logs in. To prevent this:
     + Regenerate the session ID upon successful login to ensure that the new ID is used for the session

⦁Select a request anywhere in Burp Suite Professional that you want to test or exploit.

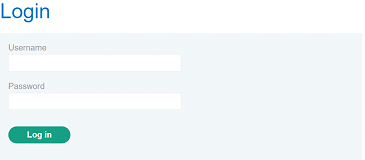
⦁From the right-click context menu, select Engagement tools / Generate CSRF PoC.

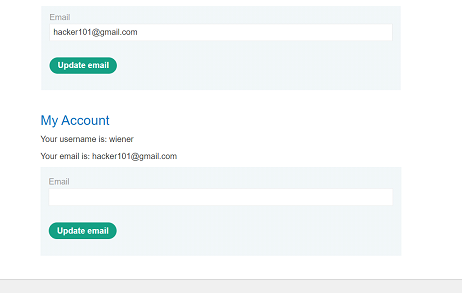
⦁Burp Suite will generate some HTML that will trigger the selected request (minus cookies, which will be added automatically by the victim's browser).

⦁You can tweak various options in the CSRF PoC generator to fine-tune aspects of the attack. You might need to do this in some unusual situations to deal with quirky features of requests.

⦁Copy the generated HTML into a web page, view it in a browser that is logged in to the vulnerable web site, and test whether the intended request is issued successfully and the desired action occurs.

Open Burp's browser and log in to your account. Submit the "Update email" form, and find the resulting request in your Proxy history.





Use the following HTML template and fill in the request's method, URL, and body parameters. You can get the request URL by right-clicking and selecting "Copy URL".

<form method="$method" action="$url">

<input type="hidden" name="$param1name" value="$param1value">

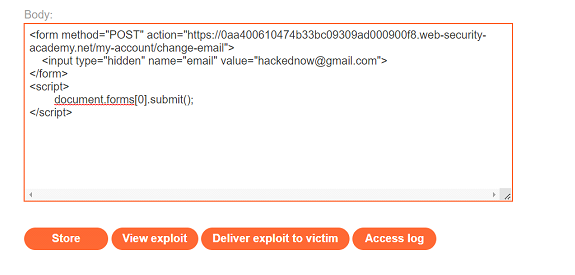
</form>

<script>

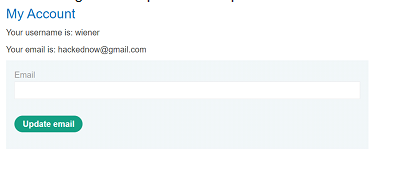
document.forms[0].submit();

</script>

Go to the exploit server, paste your exploit HTML into the "Body" section, and click "Store".



To verify that the exploit works, try it on yourself by clicking "View exploit" and then check the resulting HTTP request and response.



CSRF Attack using DVWA

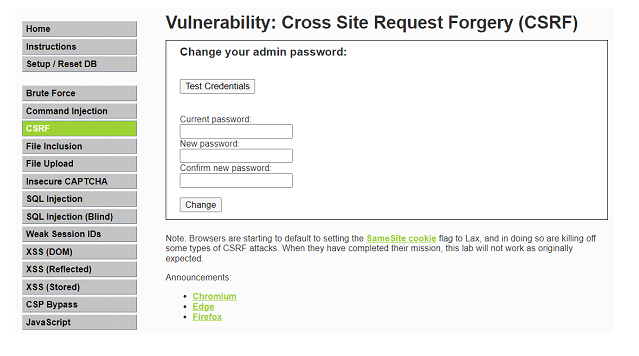
Here, we will use the Damn Vulnerable Web Application (DVWA). It’s a web app developed in PHP and MySQL and intentionally made to be vulnerable.

Use the default credentials below:

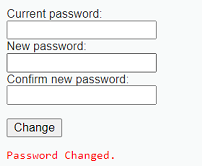
Username: admin

Password: password

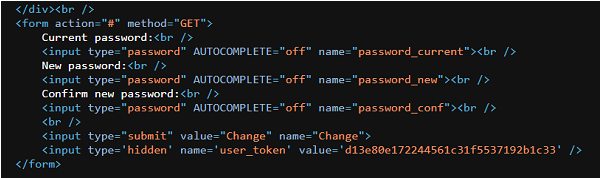
After a successful login, you will see the DVWA main page. Now click on the CSRF tab on the left pane.



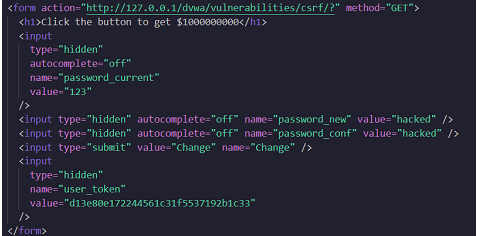
If we change the password of the login credentials from “password” to “123”, the password gets changed. Now performing the CSRF attack.



Now we right click on the page and go to the Page Source. Copy the form tag given below.



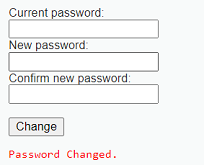
Now open any text editor and edit the following form tag code into the following text. Save it as mysite.html.



On opening the HTML file, it will redirect to the following page.



On clicking the change button, it will redirect to the DVWA CSRF Credentials page. Now the password is changed to “hacked”.



**Conclusion:**

Thus we have studied how to validate Session Management for Web Application and performed CSRF attacks using DVWA and Portswigger.