Experiment No. 3

Aim: To simulate and protect against Cross-site Request forgery (CSRF) in a web application.

Learning Objective: To simulate and protect against Cross-site Request forgery (CSRF) in a web application

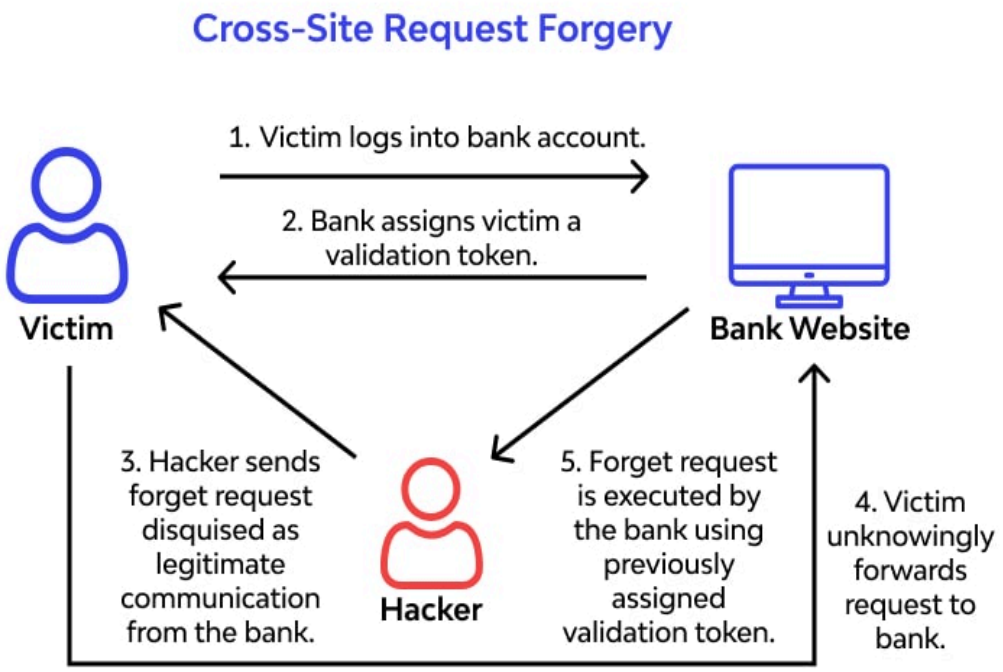
Theory:

Cross-Site Request Forgery (CSRF), also known as a "one-click attack" or "session riding," is an attack that tricks a user's browser into making an unintended and malicious request to a web application where the user is already authenticated. Since the browser automatically includes any relevant cookies (like session cookies) with the request, the vulnerable application treats the malicious request as a legitimate action performed by the authenticated user.

How a CSRF Attack Works

The attack exploits the trust a web application has in a user's browser. Here's the typical flow:

1. **Authentication**: A user logs into a trusted, but vulnerable, web application (e.g., your-bank.com). The application places a session cookie in the user's browser to keep them logged in.
2. **The Lure**: The attacker tricks the user into visiting a malicious website (e.g., evil-site.com). This site could contain an innocent-looking image, link, or button.
3. **The Forged Request**: Unbeknownst to the user, the malicious page contains hidden code (like an HTML form) that automatically sends a request to the vulnerable application (your-bank.com). This request is designed to perform a state-changing action, like transferring money or changing the user's password.
4. **Exploiting Trust**: When the browser sends this forged request to your-bank.com, it helpfully includes the session cookie. The bank's server sees a valid session cookie and processes the malicious request as if the user genuinely initiated it.



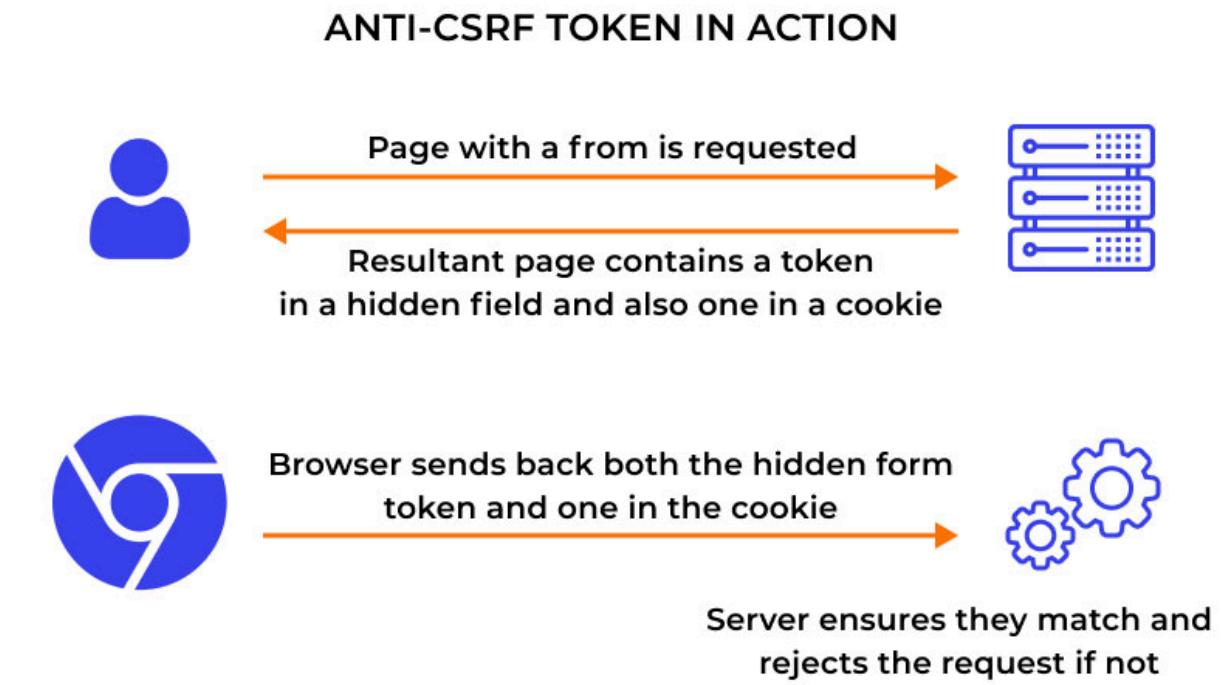
The Defense: Anti-CSRF Tokens

The most common and effective defense against CSRF is the **Synchronizer Token Pattern**.

1. **Token Generation**: When a user requests a page with a form, the server generates a unique, secret, and unpredictable token called an anti-CSRF token.
2. **Token Embedding**: The server embeds this token as a hidden field in the form and also stores it in the user's session on the server side.
3. **Token Validation**: When the user submits the form, the token from the hidden field is sent back to the server. The server then compares this submitted token with the one stored in the user's session.
    - **If they match**, the request is considered legitimate and is processed.
    - **If they don't match (or the token is missing)**, the request is rejected as it's likely a forged request.

An attacker cannot guess this token, and because of the browser's Same-Origin Policy, the malicious site cannot read the token from the trusted site's page to include it in the forged request.

wallarm



Procedure / Implementation:

This procedure outlines the steps to simulate a CSRF attack and then implement a defense.

1. **Set Up the Vulnerable Application**: First, create and run a simple web application using Python and Flask. This app should have a login system and a form that performs a sensitive, state-changing action, like changing a user's email address. Crucially, this form should not have any CSRF protection.
2. **Create the Attacker's Webpage**: Develop a separate, malicious HTML page. This page will contain a hidden form that targets the vulnerable application's email-change endpoint. Use JavaScript or a body onload event to automatically submit this form when a victim visits the page.
3. **Execute the Attack**: Run the vulnerable server. In a browser, log into the vulnerable application. Then, in a new tab, open the attacker's HTML file. The malicious form will submit in the background. Return to the vulnerable application's tab and refresh the page to confirm that the email has been changed, proving the attack was successful.

4. **Implement CSRF Protection**: Modify the web application to defend against this attack. The protection involves generating a unique, secret **anti-CSRF token** for each user session. This token must be included as a hidden input in the email-change form. The server-side code must be updated to validate that the token submitted with the form matches the one stored in the user's session. If they don't match, the request must be rejected.
5. **Verify the Protection**: Run the newly secured application. Log in again. Attempt the attack by opening the attacker's webpage. This time, the attack will fail because the malicious form cannot provide the correct anti-CSRF token. Refresh the application page to confirm that the email address remains unchanged. The server terminal should show an error indicating a token mismatch, confirming the defense is working.

1. Vulnerable Application (vulnerable_app.py)

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

app = Flask(__name__)
app.secret_key = 'demo-key'
# Simple in-memory user data
users = {'testuser': {'email': 'original@example.com'}}
LOGGED_IN_USER = None

PROFILE_PAGE = """
<h2>Vulnerable Profile</h2>
<p>Current Email: {{ email }}</p>
<form method="post" action="/change_email">
    New Email: <input type="text" name="new_email">
    <input type="submit" value="Change">
</form>
"""

@app.route('/')
def index():
    # Simple login simulation for demo
    global LOGGED_IN_USER
    LOGGED_IN_USER = 'testuser'
    return render_template_string(PROFILE_PAGE, email=users[LOGGED_IN_USER]['email'])

@app.route('/change_email', methods=['POST'])
def change_email():
    new_email = request.form.get('new_email')
    users[LOGGED_IN_USER]['email'] = new_email
    print(f"Email changed to: {new_email}")
    return redirect('/')

if __name__ == '__main__':
    app.run(port=5000)
```

2. Attacker's Site (attacker.html)

```html
<!DOCTYPE html>
<html>
<head>
    <title>You've Won a Prize!</title>
</head>
<body onload="document.forms[0].submit()">
    <h1>Loading your prize...</h1>
    <form action="http://127.0.0.1:5000/change_email" method="POST" style="display:none;">

        <input type="hidden" name="new_email" value="hacked@evil.com" />
    </form>
</body>
</html>
```

3. Protected Application (protected_app.py)

```python
import os
from flask import Flask, request, render_template_string, session, redirect, abort

app = Flask(__name__)
app.secret_key = os.urandom(24) # Use a real secret key
users = {'testuser': {'email': 'original@example.com'}}

# Note the new hidden input for 'csrf_token'
PROFILE_PAGE_SECURE = """
<h2>Protected Profile</h2>
<p>Current Email: {{ email }}</p>
<form method="post" action="/change_email">
    <input type="hidden" name="csrf_token" value="{{ session.csrf_token }}">
    New Email: <input type="text" name="new_email">
    <input type="submit" value="Change">
</form>
"""

@app.before_request
def setup_session():
    # Set up user and CSRF token in session
    session['user'] = 'testuser'
    if 'csrf_token' not in session:
        session['csrf_token'] = os.urandom(16).hex()

@app.route('/')
def index():
    return render_template_string(PROFILE_PAGE_SECURE, email=users[session['user']]
['email'])

@app.route('/change_email', methods=['POST'])
def change_email():
    # *** Protection Step: Validate the token ***
    if request.form.get('csrf_token') != session.get('csrf_token'):
        abort(403, "CSRF token is invalid or missing.")

    new_email = request.form.get('new_email')
    users[session['user']]['email'] = new_email
    print(f"Email changed to: {new_email}")
    return redirect('/')

if __name__ == '__main__':
    app.run(port=5001)
```

Learning Outcome: Upon completing this experiment, you will understand how Cross-Site Request Forgery (CSRF) attacks exploit browser trust to perform unauthorized actions on behalf of a user. You will also be able to implement the Synchronizer Token Pattern using anti-CSRF tokens to effectively defend a web application against this common vulnerability.

Conclusion: This experiment proves that CSRF is a significant threat, capable of hijacking a user's authenticated session to perform unwanted actions. The most effective defense is using anti-CSRF tokens, which act as a unique, secret password for each form submission. Because an attacker's site cannot guess this secret token, the protected application can easily distinguish a legitimate request from a forged one and block the attack, demonstrating that this token-based defense is essential for web security.

Name:
Class: BE-CSE
Roll No.:

For Faculty Use

Correction Parameters	**Formative Assessment [40%]**	**Timely completion of Practical [40%]**	**Attendance / Learning Attitude [20%]**	
Marks Obtained				