Report on

CSRF

ROBENSIVE

BY Abhishek

Guided By **Rikunj Sindhwad** (R0bensive)

TABLE OF CONTENT

1. Prerequisites

1.1 Same-Origin Policy (Default implemented in browsers)

Same-Origin Policy is a browser-enforced security mechanism that prevents JavaScript from reading or modifying data from another origin unless explicitly permitted, thereby protecting users from data theft. Thus, SOP prevents cross-site data access.

SOP doesn't block sending requests it blocks reading requests.

Components of Origin

Component	Example
Scheme	`https`
Host (domain)	`example.com`
Port	`443`

❌ **Different origin if any one differs:**

- http://example.com ❌ (protocol differs)
- https://api.example.com ❌ (subdomain differs)
- https://example.com:8443 ❌ (port differs)
- https://example.com:443/api ✅ (Every component match)

Note: All components of origin must match for Same-Origin Policy.

1.1.1 What SOP Protects

- Modifying another site's DOM
- Accessing response data

SOP is default deny. CORS is a controlled exception through which resources are shared. CORS is SOP's permission system.

1.1. 2 CORS (SOP Exception Rules)

We know that Same-Origin Policy by default deny the read permission for cross-origins. But what if `frontend.example.com` want to read data from API server (`api.example.com`). We will be blocked.

Thus, CORS is a header-based security mechanism that provides controlled exception mechanism to the Same-Origin Policy. It is a browser-enforced protocol where the server explicitly tells browser which other origins are allowed to read its response.

CORS does not block sending requests; it only restricts JavaScript from reading cross-origin responses. This is why CSRF attacks are still possible—CSRF relies on triggering unintended state-changing actions, not on reading the victim's response data.

Access-Control-Allow-Origin

The `Access-Control-Allow-Origin` header is included in response from one website to a request originating from another website, and identifies the permitted origin of the request.

It can be a:

- **Wildcard(*):** Allowing all domains to access its resources.
- **Specific domain or list of domains:** Only specified domain can access its resources.
- **Null:** No one can access

Access-Control-Allow-Methods: Defines which methods is allowed to get the response.

Access-Control-Allow-Credentials

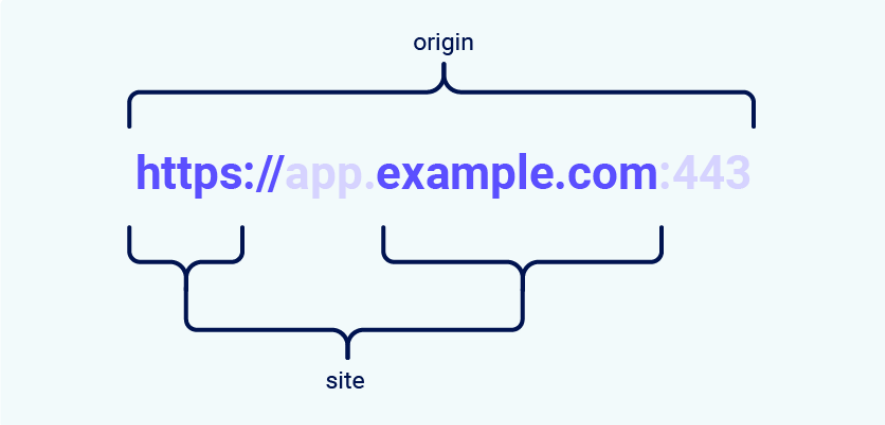
If this header is set to `true` then the following can be included in the response:

- Cookies
- Authorization Header
- Session Tokens

Thus, CORS is browser-only, read-access control.

1.2 SameSite cookie (Cookie Control)

SameSite is a browser security mechanism that determines when a website's cookies are included in requests originating from other websites. It restricts cookie attachment on cross-site requests.



Request from	Request to	Same-site?	Same-origin?
https://example.com	https://example.com	Yes	Yes
https://app.example.com	https://intranet.example.com	Yes	No: mismatched domain name
https://example.com	https://example.com:8080	Yes	No: mismatched port
https://example.com	https://example.co.uk	No: mismatched eTLD	No: mismatched domain name

1.2.1 SameSite Cookie Modes

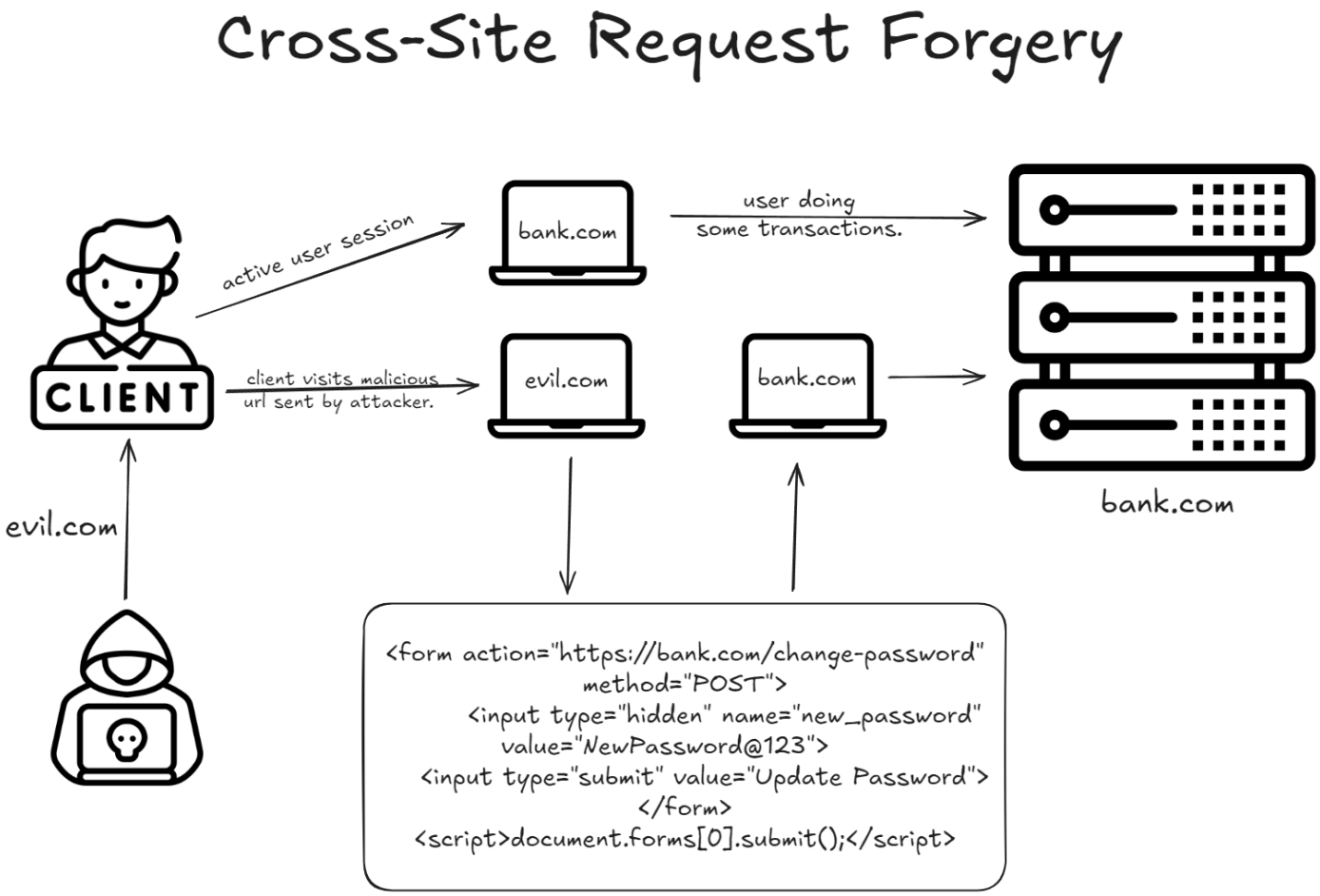
1. **Strict**
    - **How it works**
        - Cookies are sent only if the request originates from the same site.
        - It is strongest CSRF protection at browser level.
    - **Best Application**
        - Admin Panels
        - Internal tools
        - Sensitive Dashboards
    - **Disadvantages**
        - Even legitimate navigation via links gets blocked. (i.e., a secure bank link embedded in another bank link for reference).
        - Even bookmarked links gets blocked.
        - Even email links get blocked.
2. **Lax (Default used in browser)**
    - Cookies are blocked on cross-site state-changing requests, but allowed on top-level safe navigations.
    - **How it works**
        - Cookies are sent when
            - Same-site requests
            - Cross-site top-level navigation
                - `evil.com → GET bank.com` ✅
                - `evil.com → POST bank.com` ❌
        - Cookies are not sent when:
            - Cross-Site POST
            - Cross-site PUT / PATCH / DELETE
            - Request from iframes
            - AJAX / fetch from another site

- **Disadvantages**
    - If a web application is allowing `GET /logout` or `GET /delete-account` then CSRF is still possible.

3. **None**
    - Cookie is sent in all contexts, including cross-site.
    - No CSRF Protection at browser level.
    - Why it is required:
        - Third-party authentication
        - Cross-site SSO
        - Embedded apps
        - Payment gateways

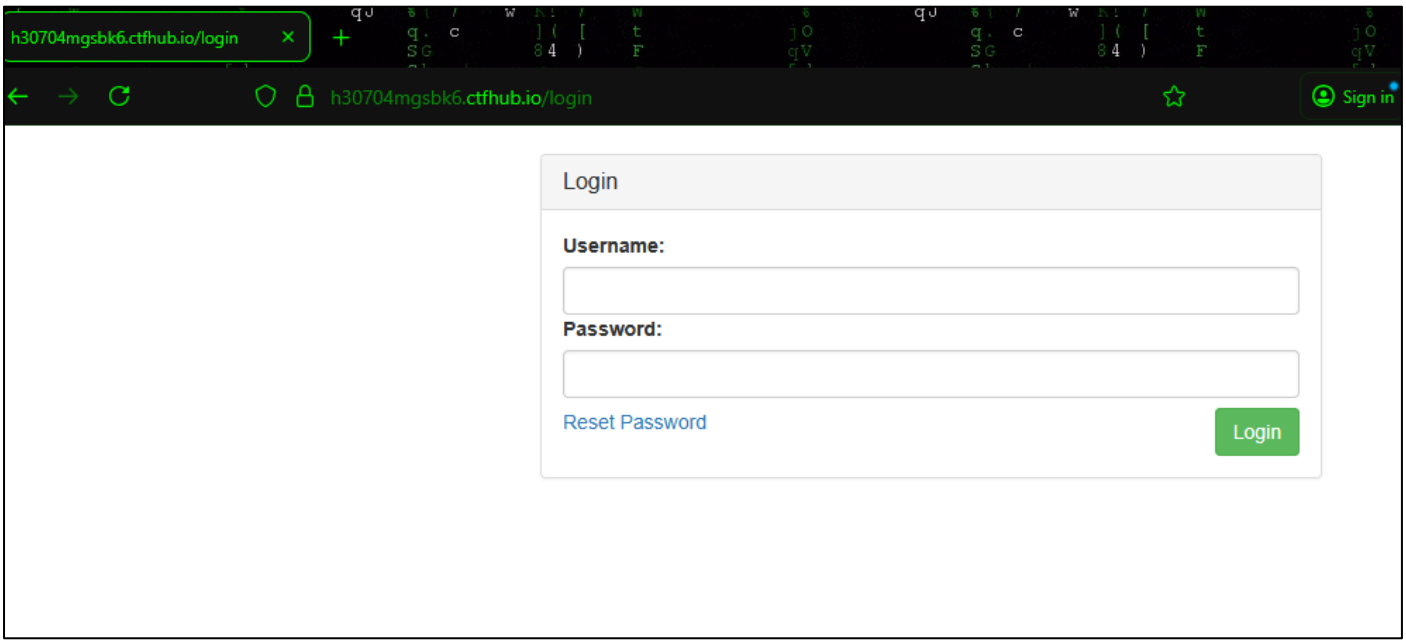
2. What is CSRF ?

CSRF is a client-side vulnerability which falls under Broken Access Control because it occurs when the server blindly trusts a user's active session and processes sensitive actions—such as changing settings, updating data, or performing transactions—without verifying that the request was deliberately initiated by the authenticated user and not triggered by a malicious third-party website.



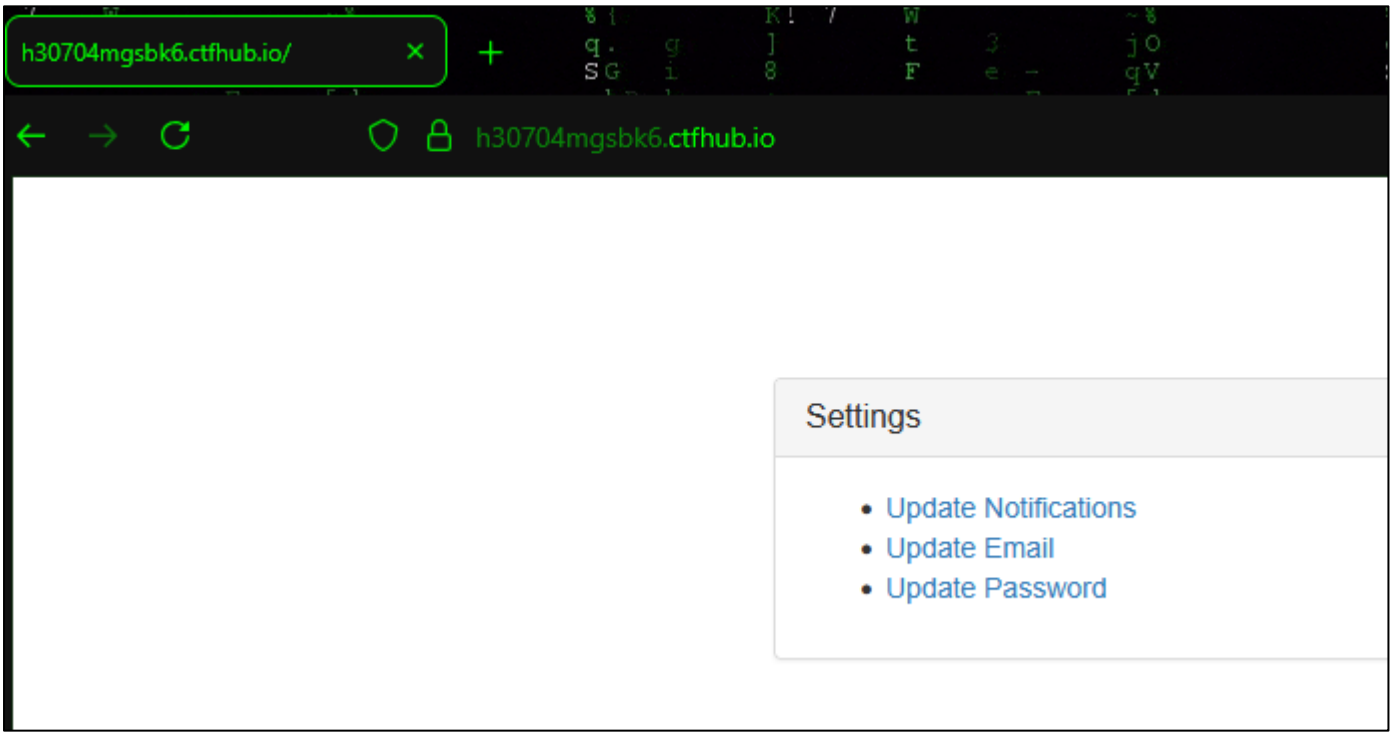
2.1 Exploiting CSRF

2.1.1 Target

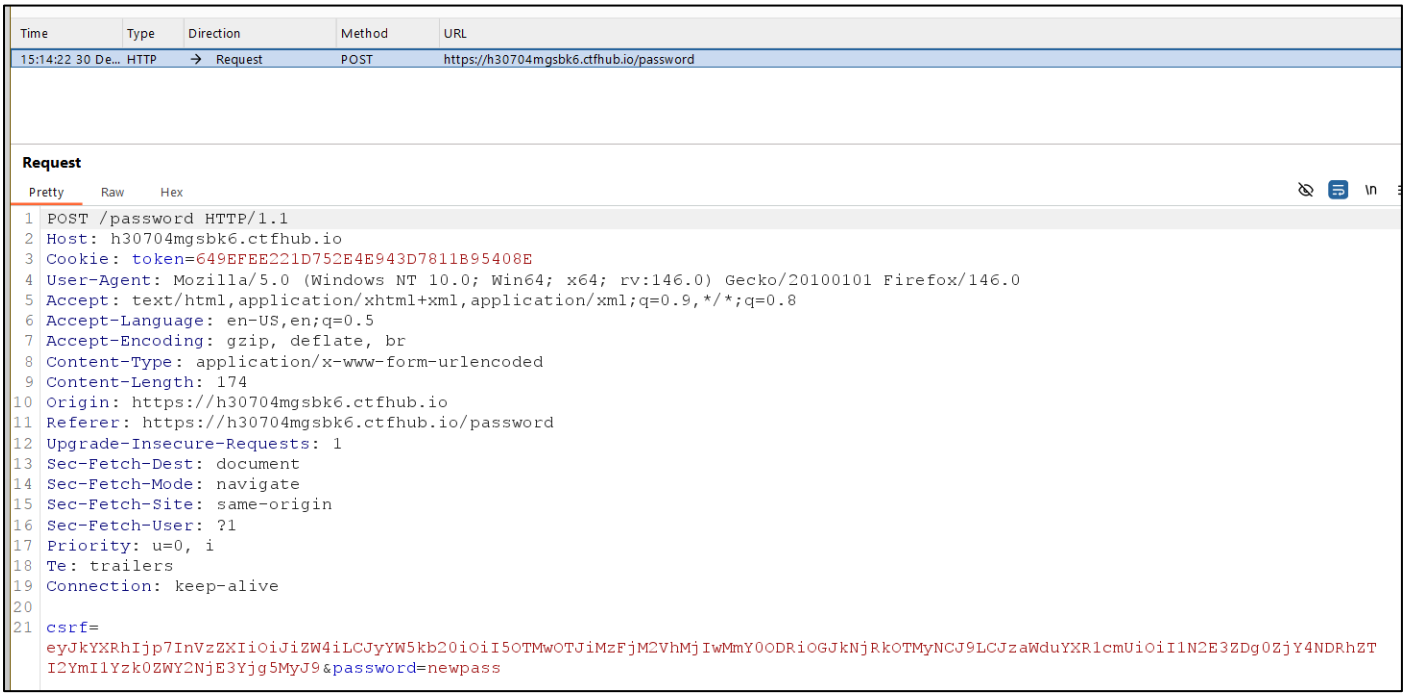


The target lab has provided two credentials. Let's login with one of the credentials and check the functionality.

2.1.2 Functionality



Thus, a user can update notifications/email/password from his login. Let's intercept the password update request and check it in Burp.



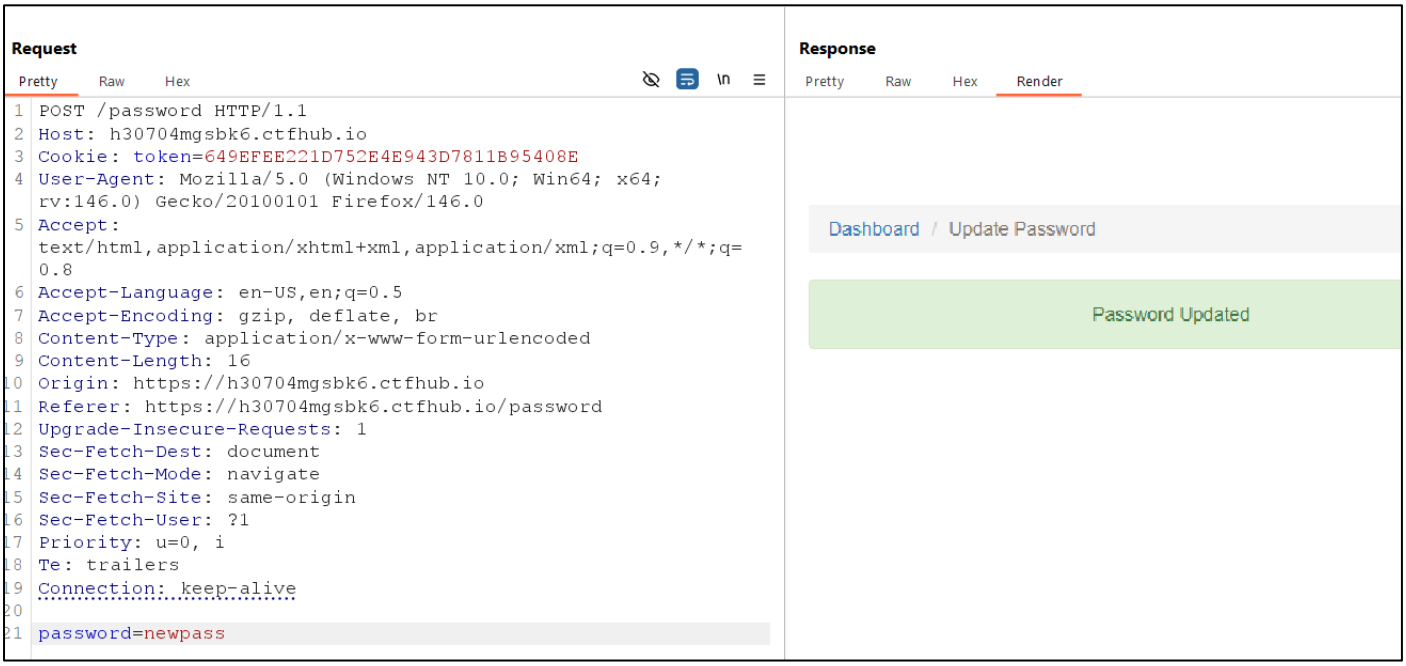
Time	Type	Direction	Method	URL
15:14:22 30 De...	HTTP	→ Request	POST	https://h30704mgsbk6.ctfhub.io/password

Request

Pretty Raw Hex

```
1  POST /password HTTP/1.1
2  Host: h30704mgsbk6.ctfhub.io
3  Cookie: token=649EFEE221D752E4E943D7811B95408E
4  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:146.0) Gecko/20100101 Firefox/146.0
5  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
6  Accept-Language: en-US,en;q=0.5
7  Accept-Encoding: gzip, deflate, br
8  Content-Type: application/x-www-form-urlencoded
9  Content-Length: 174
10 Origin: https://h30704mgsbk6.ctfhub.io
11 Referer: https://h30704mgsbk6.ctfhub.io/password
12 Upgrade-Insecure-Requests: 1
13 Sec-Fetch-Dest: document
14 Sec-Fetch-Mode: navigate
15 Sec-Fetch-Site: same-origin
16 Sec-Fetch-User: ?1
17 Priority: u=0, i
18 Te: trailers
19 Connection: keep-alive
20
21 csrf=
   eyJkYXRhIjp7InVzZXIiOiJiZW4iLCJyYW5kb20iOiI5OTMwOTJiMzFjM2VhMjIwMmY0ODRiOGJkNjRkOTMyNCJ9LCJzaWduYXR1cmUiOiI1N2E3ZDg0ZjY4NDRhZTI2YmI1Yzk0ZWY2NjE3Yjg5MyJ9&password=newpass
```

Let's send the same request in the repeater and check if we can forge the request by sending it without csrf token and other practices.



Request

Pretty Raw Hex

```
1  POST /password HTTP/1.1
2  Host: h30704mgsbk6.ctfhub.io
3  Cookie: token=649EFEE221D752E4E943D7811B95408E
4  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64;
   rv:146.0) Gecko/20100101 Firefox/146.0
5  Accept:
   text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=
   0.8
6  Accept-Language: en-US,en;q=0.5
7  Accept-Encoding: gzip, deflate, br
8  Content-Type: application/x-www-form-urlencoded
9  Content-Length: 16
10 Origin: https://h30704mgsbk6.ctfhub.io
11 Referer: https://h30704mgsbk6.ctfhub.io/password
12 Upgrade-Insecure-Requests: 1
13 Sec-Fetch-Dest: document
14 Sec-Fetch-Mode: navigate
15 Sec-Fetch-Site: same-origin
16 Sec-Fetch-User: ?1
17 Priority: u=0, i
18 Te: trailers
19 Connection: keep-alive
20
21 password=newpass
```

Response

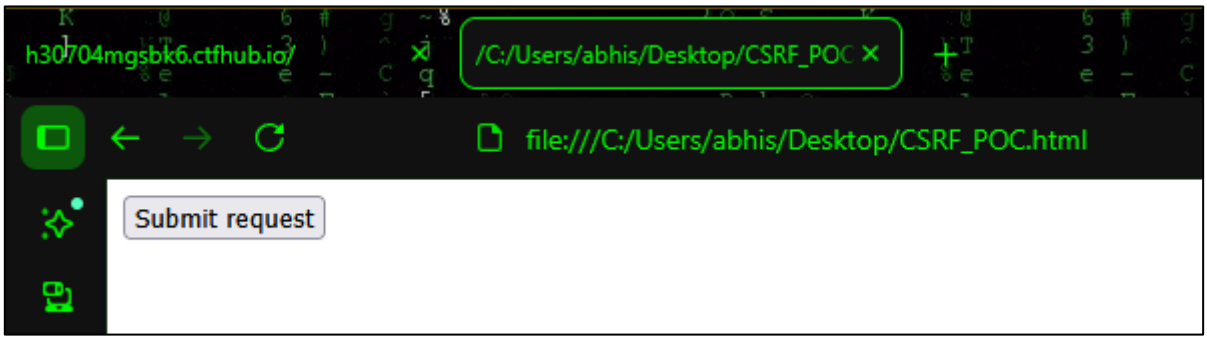
Pretty Raw Hex Render

Dashboard / Update Password

Password Updated

We can see that the token is not being validated. Thus now let's login to another user account which is "admin" and open CSRF_PoC.html generated by burp in it to change his password.

2.1.3 Exploitation



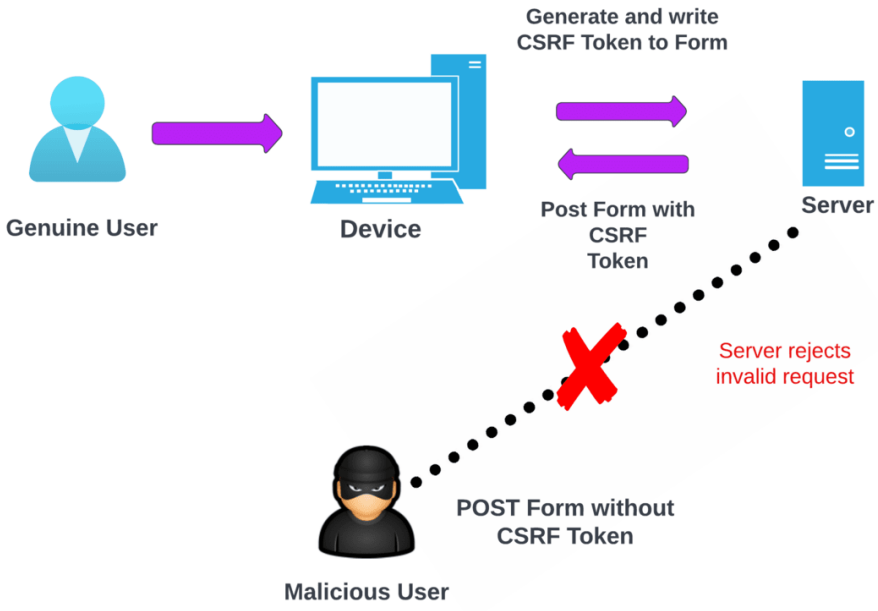
After clicking on “submit request” as an admin the form will be submitted and the password will be changed.

Portswigger Solved labs

3. CSRF Mitigation Techniques

3.1 Synchronizer Token Pattern (Anti-CSRF Token)

A token is a random, unpredictable value generated by the server, stored on the server and required in every sensitive request.



There are two types for it:

1. **Per-Session CSRF Token (Most Common)**
    - Token is generated once at login.
    - Stored in user's server-side session.
    - Same token is embedded in every form and validated for every request.
2. **Per-Request (One-Time) CSRF Token (Stronger)**
    - New token generated for each form/request.
    - Token is single use and are invalidated after use.
    - *It increases complexity and ease of use as navigating back make it an invalid session because that token is already used.*

3.1.1 Common Disadvantage of using CSRF Token (Either Per-Session or Per-Request)

1. **CSRF Tokens Do NOT Protect Against XSS. (Biggest Limitation).**
    - An XSS vulnerability inside the application can allow an attacker to read the DOM and extract the CSRF token allowing them to send valid authenticated requests. Fix XSS before trusting CSRF Protection.

2. **Implementation Complexity and Developer Errors**
    - Common mistakes done by developers:
        - Token generated but never validated.
        - Token validated only for single method (i.e., POST), not GET / PUT / PATCH / DELETE.
        - Token not tied correctly to a user session.
        - Forget application of token in every sensitive request.

3.1.2 CSRF Token Implementation – MUST requirements

1. Always generate unique, random, unpredictable and large token (minimum 128 bit).
2. A single Token is always tied to a single user session.
3. Always send token in request body or Custom-Header instead of URL / query string or cookie.

```
Request Body:    <input type="hidden" name="csrf_token" value="……")

Custom-Header:   X-CSRF-Token: abc123
```

4. The server must strictly validate the CSRF token against the value stored for the authenticated user session, and reject all requests with missing, invalid, or mismatched tokens.
5. CSRF protection must be enforced on all state-changing HTTP methods, including POST, PUT, PATCH, and DELETE and CSRF token validation must be performed on server-side only.
6. Regenerate token when:
    - User logs in
    - User logs out
    - Session is rotated
7. Use `SameSite=Lax` with no GET state-changing request in the application.

Note: *Cross-Site Scripting (XSS) is the most critical threat to CSRF defenses, as successful XSS execution allows an attacker to bypass most CSRF mitigation techniques by operating within the same origin and accessing CSRF tokens directly.*

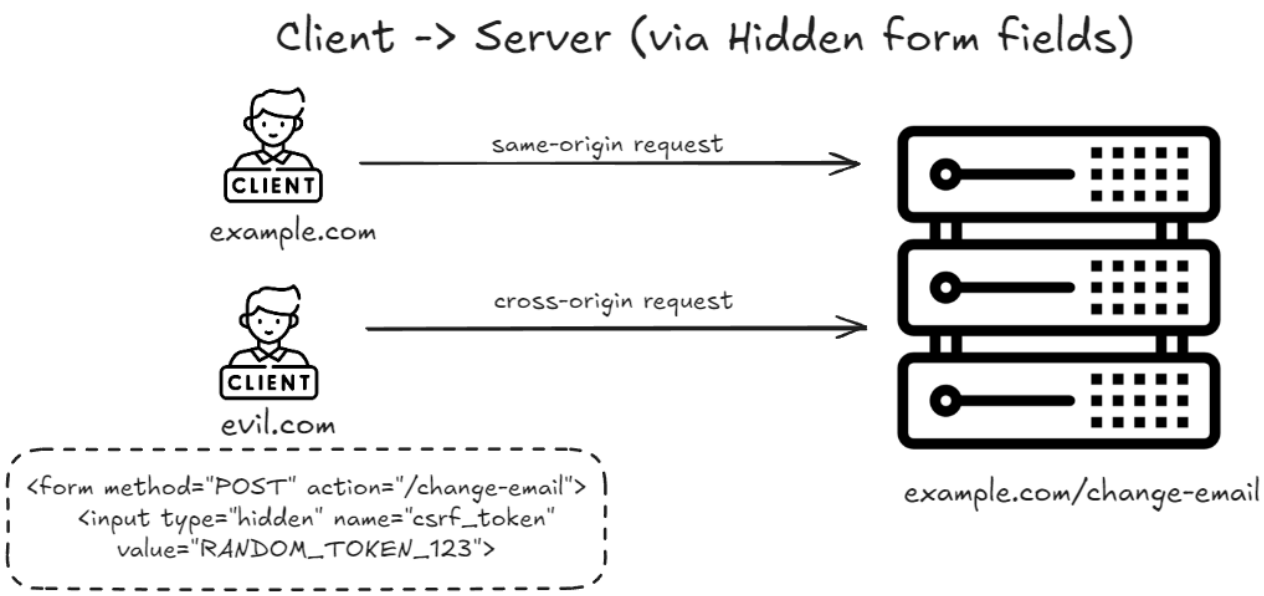
3.1.3 Transmitting CSRF Tokens in Synchronized Patterns

Server → Client via hidden form fields (response payloads) or JSON response.

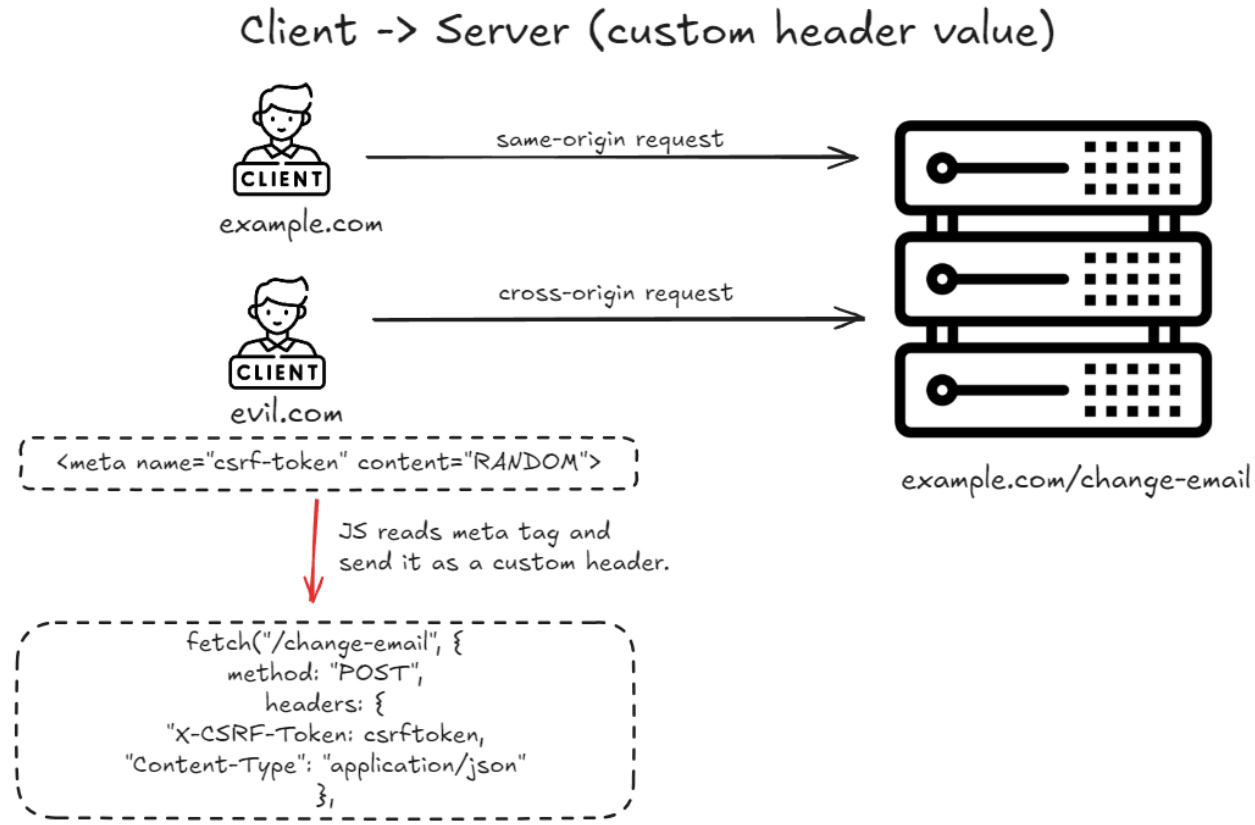
Client → Server

There are two ways to send CSRF token from client to server.

1. **Hidden form field (Request Body)** **(Less Secure)**



2. **AJAX request (custom header value)** **(More Secure)**



3.2 Double Submit Cookie pattern

Here the server doesn't validate the token as the token is not stored on the server side. The browser simply put same CSRF value in two places, and verify if they match.

Those two places are:

1. Cookie (automatically sent by the browser)
2. Request parameter or header (manually added by JS/forms)

The server simply checks if both the token matches and if it is then it validates the request.

If cookie.csrf == request.csrf → OK

Else → BLOCK

3.2.1 Disadvantage of using Double Submit Cookie

1. It does not verify the token's origin. If an attacker gains control of a subdomain (via subdomain takeover), they can set forged cookies using Set-Cookie and have them sent with the user's requests.
2. XSS as always is the biggest limitation.

3.3 HMAC Double Submit Validation

An HMAC CSRF token is a stateless CSRF protection mechanism where the token is cryptographically signed using a server secret and session-specific data, allowing the server to verify authenticity without storing CSRF state.

3.3.1 Token Construction

The server generates a CSRF token by computing a Hash-based Message Authentication Code (HMAC) over a message that includes:

- A session-dependent value (e.g., server session identifier or JWT ID)
- A cryptographically secure random nonce
- A server-side secret key used exclusively for HMAC computation

CSRF_Token = HMAC(secret, session_value || nonce) || "." | nonce

3.3.2 Validation

1. **Presence and equality validation**
    - The server first verifies if csrf_token of request match with that of cookie.

2. **Cryptographic authenticity validation**
    - After equality is confirmed, the server verifies who created the token.
    - The server recomputes the expected HMAC using its secret key, the current session identifier, and the nonce from the request, then compares it with the received HMAC using a constant-time comparison; if they match, CSRF validation succeeds.

3.3.3 Design Considerations

- Tokens must be regenerated on **session creation and rotation**.
- Static identifiers (user ID, email) must not be used as session-binding inputs.
- HMAC must be preferred over simple hashing to ensure integrity and authenticity.
- Token expiration timestamps are unnecessary; session termination invalidates the token.

3.4 Origin & Referrer header validation

It is a very important secondary defence mechanism which should be implemented along with the Synchronizer Token Pattern or HMAC Double Submit Cookie validation for better prevention against CSRF.

You should add a section explaining:

- Validate Origin header for state-changing requests
- Fallback to Referer if Origin is absent
- Reject requests with:
    - Missing header & Mismatched origin

3.5 Fetch Metadata headers

Fetch Metadata headers are browser-supplied HTTP headers that describe the origin and context of a request, allowing servers to identify and block cross-site requests and thereby mitigate CSRF attacks as a defenses-in-depth mechanism. Core headers are:

1. **Sec-Fetch-Site (Most Important)**

Value	Meaning
same-origin	Request from the same origin
same-site	Same registrable domain
cross-site	Different site (potential CSRF)
none	User-initiated (e.g., bookmark, address bar)

CSRF attacks almost always use `cross-site`.

2. **Sec-Fetch-Mode, Sec-Fetch-Dest, Sec-Fetch-User** - additional headers that provide context about the request (such as the request mode, destination type, or whether it was triggered by a user navigation).

4. Conclusion

CSRF is not a browser flaw but an application trust flaw. Effective CSRF prevention requires a layered defense strategy combining secure application design, robust server-side validation, and modern browser-provided security mechanisms. This report demonstrates not only how CSRF vulnerabilities are exploited in real-world scenarios but also how they can be systematically mitigated using industry-recommended practices.

Thank You!

ROBENSIVE

BY Rikunj

Rob's Offensive Security Training and Services