Cross Site Request Forgery (CSRF)

Kameswari Chebrolu

Outline

- What is CSRF Vulnerability?
- Defenses:
  - REST Principle
  - CSRF token; flawed implementations
  - Same Site Cookies; flawed implementations
  - Referer Header; flawed implementations
  - Reauthentication
- Summary of Good Practices
- Real Life Examples

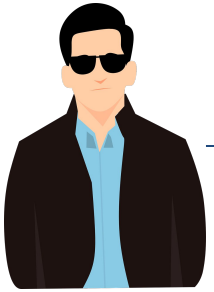
Cross Site Requests

- What is a cross site request?
- When user visits website,
  - Requests are made to the same website domain (same origin)
  - Requests are also made to different domains (cross site request)
    - Very common to use Content Delivery Networks (CDNs) to store static content
      - Improves latency!
      - Static files from CDN will be cross site
    - Common to also present ads in mainsite from advertising sites (cross site)!

CSRF Vulnerability

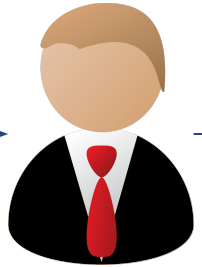
- An **attacker website** instructs a **victim's web browser** to submit a web request to another **(victim) website** (cross site)
  - Without victim's intent but under the identity of victim
  - From victim website's perspective, request indistinguishable from a legitimate and intentional request made by victim browser
    - Victim's browser will automatically include any cookies/session-tokens associated with the site

- Requests are not simple “resource fetch” requests (e.g. get static files) but requests that change state
  - Change victim’s email address to Attacker’s
  - Purchase items using victim’s account
  - Transfer funds from victim account to attacker’s
- Attackers have used CSRF to steal Gmail contact lists, forge add friend requests, update profiles, trigger one-click purchases on Amazon, and change router configurations!
- Used to be in OWASP top 10 in past!



Attacker

1. Share link to attacker.com through spam



Victim

2. Open link in browser

Victim's browser

3. Load page with malicious payload

attacker.com

4. Send request including user session tokens/cookies

vulnerable.com

5. Request is accepted and processed as legit user request

(Contrived) Example

- Suppose a bank uses GET requests to perform bank transfers
  - https://www.bank.com/transferMoney.php?amount=10000&toID=137392
- Attacker:
  - Launch a malicious website and craft an ordinary link to mask actual action
  - Click to claim your gift card!
  - Note: toID is attacker's account

- Victim:
  - Browsing the web, lands on attacker website (in one tab)
  - Further is logged into the bank website (in another tab)
  - Clicks the malicious link in attacker website
  - Victim browser sends above get request with all relevant victim cookies
- Bank website:
  - Valid request since session-token/cookies are correct
  - Executes request!

Another Attack: Login

- Malicious website: Has code that instructs victim user to authenticate to victim site as attacker (not as user)
- Victim user is unaware of above and may think he/she is logged in and may enter sensitive information
  - Attacker gets this info since it is entered in his account

Points to Note

- For a successful CSRF three conditions needed!
  - A relevant action that is advantageous to attacker
    - E.g. changing user's email address or password or stealing cookies
  - Cookie-based session handling
    - Server relies solely on (session) cookies to identify the user
  - Predictable request parameters
    - All parameters in the request can be determined or guessed by attacker
    - E.g. if field is new password but function requires inputting current password as well → attack not easy!

- Victim does not need to even click on a link; Just visiting the website can trigger the request
  - Can be achieved via certain HTML tags and JavaScript
  - E.g. <img> tag will automatically generate a GET request to retrieve the image resource
  - `<img src="https://vulnerable.com/change-email?email=attacker@evil.com">`
- Not just GET, works will POST as well!
  - POST requests often contain data submitted through a form
  - Can use a FORM tag that is executed automatically using JavaScript.

POST based Payload

```
<html>
    <body>
        <form action="https://vulnerable.com/change-email"
method="POST">
            <input type="hidden" name="email"
value="attacker@evil.com" />
        </form>
        <script>
            document.forms[0].submit();
        </script>
    </body>
</html>
```

When victim visits, the form is auto submitted

HTTP response back to the browser can be hidden via an iFrame using the display:none attribute

```
<iframe src="form.html"
width="400" height="300"
style="display:
none;"></iframe>
```

- GET requests more vulnerable to CSRF attacks
  - Request's contents in the URL itself → easy to craft
  - E.g. In old Twitter, tweets were created via GET
    - Attacker created URL when clicked, would post on a victim user's timeline; tricked a few twitter users to click on it!
    - Viral worm: When readers of tweet clicked the link, they too were tricked into tweeting the same thing on their timeline

Defense: Follow REST Principles

- REST stands for Representational State Transfer
- REST states one should map website operations to appropriate HTTP methods
  - Fetch data or pages with GET requests
  - Create new objects on the server with PUT requests
  - Modify objects on the server with POST requests
  - Delete objects with DELETE requests
- Defusing your GET requests (i.e. do not use them to change state) shuts the door on most CSRF attacks
  - Other methods require more work!
  - Determined attacker would however exploit other methods

Defence: CSRF Token

- Token is unique and unpredictable
- Generated by server and shared with client via hidden fields on sensitive pages
  - Sometimes shared as json data or custom headers
- Client must retrieve the token and include it in the request sent
  - This is handled by javascript based on specific implementation
  - Attacker cannot predict this token or read it from a legitimate page sent to victim → hence cannot create the right payload
- Most popular web frameworks provide this feature

Server Form

```
<form name="change-email-form"
action="/my-account/change-email" method="POST">
    <label>Email</label>
    <input required type="email" name="email"
value="user@example.com"/>
    <input required type="hidden" name="csrf"
value="50FaWgdOhi9M9wyna8taR1k3ODOR8d6u"/>
    <button class='button' type='submit'> Update email
</button>
</form>
```

Client POST

```
POST /my-account/change-email HTTP/1.1
Host: normal-website.com
Content-Length: 70
Content-Type: application/x-www-form-urlencoded

csrf=50FaWgdOhi9M9wyna8taR1k3ODOR8d6u&email=
user@example.com
```

Some applications place CSRF tokens in HTTP headers as well!

```
HTTP/1.1 200 OK
Content-Type: text/html
X-CSRF-Token:50FaWgd
Ohi9M9wyna8taR1k3ODO
R8d6u
```

```
POST /my-account/change-email
HTTP/1.1
Host: normal-website.com
Content-Length: 70
Content-Type:
application/x-www-form-urlencoded
X-CSRF-Token:50FaWgdOhi9M9w
yna8taR1k3ODOR8d6u

email=user@example.com
```

- Some servers generate a randomized token and put it as a hidden field in a form (like before)
- Further, server also includes the same token in the HTTP response headers
  - Set-Cookie: _xsrf=5a78f21f41h417n2
- When user submits form, web server validates token from form and token in return Cookie header matches!

Flawed Implementations

- Validation depends on request method
  - Validate correctly if POST but skip validation if GET
  - Attacker converts POST requests to GET
- Validate if token present, skip if token absent
  - Attacker doesn't have to do much!
- CSRF token is not tied to user session
  - Server maintains a pool of issued tokens and accepts any token from the pool
  - Attacker will get a valid token for self and use that token on behalf of victim

More complex Attack

- Server uses different framework for session handling and CSRF protection
  - Sets a CSRF cookie (e.g. csrfKey=htyUbdsJV12KLHJmqnLKJaqN812ML)
  - Sets a CSRF token (e.g csrf=ojNQCN113mqTK812MLN56KSAW)
  - Both CSRF cookie and token are tied to a user but not tied to session cookie
  - Expects client to send cookie and token and checks if they belong together

```
POST /my-account/change-email  HTTP/1.1
Host: vulnerable.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 68
Cookie: session=pSJYSScWKpmC60LpFOAHKixuFuM4uXWF;
csrfKey=htyUbdsJV12KLHJmqnLKJaqN812ML

csrf=ojNQCN113mqTK812MLN56KSAW&email=user@example.
com
```

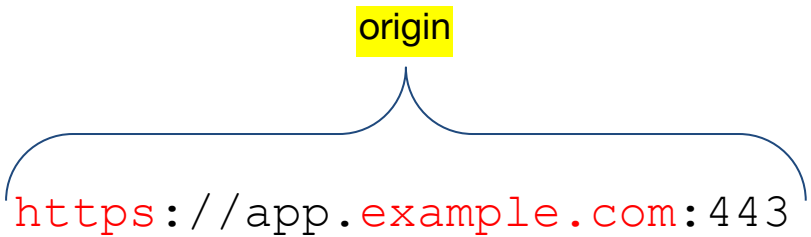
Server will compare the cookie and the token, they should be a valid pair

- Difficult to exploit unless website has a vulnerability that lets attacker set a cookie
  - E.g if a search can result in server setting a cookie to remember last search item
    - Attacker can do /?search=dogs%0d%0aSet-Cookie:%20csrfKey=Token%3b
      - Encoding: Newline, space and semicolon
  - Attacker can log in using own account and obtain a valid token and associated cookie pair
  - And use payload as shown

```
<html>
    <body>
        <form action="https://vulnerable.com/my-account/change-email"
method="POST">
            <input type="hidden" name="email" value="attacker@evil.com" />
    <input required type="hidden" name="csrf"
value="50FaWgdOhi9M9wyna8taR1k3ODOR8d6u" />
    <input type="submit" value="Update Email" />
    </form>
    <img
src="https://vulnerable.com/?search=%0d%0aSet-Cookie:%20csrfKey=YOUR-KEY%3b%20Same
Site=None" onerror="document.forms[0].submit()">
    </body>
</html>
```

Defence: SameSite Cookies

- SameSite Cookies: An attribute set by server and implemented by browser
- Tells the browser whether or not to include a website's cookies in a cross site request
  - Note cookie is still being sent to the right end point



Request from	Request to	Same-site?	Same-origin?
https://example.com	https://example.com		
https://app.example.com	https://intranet.example.com		
https://example.com	https://example.com:8080		
https://example.com	https://example.co.uk		
https://example.com	http://example.com		

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
https://example.com	http://example.com	No: mismatched scheme	No: mismatched scheme

All major browsers currently support the following

- Strict: browsers will not send any cookies in any cross-site requests
  - Target site for the request does not match the site currently in browser's address bar
  - Good to set it for pages where user can modify data or perform other sensitive actions
  - Set-Cookie: session=0F8tgdOhi9ynR1M9wa3ODa; SameSite=Strict
  - Most secure, but can negatively impact user experience
    - Many third party analytics require this cross-site functionality (tracking cookies)

- Lax: browsers will send the cookie in cross-site requests, if and only if
  - The request uses the GET method
  - The request resulted from a top-level navigation by the user
    - TOP LEVEL navigation changes the URL in the address bar
      - Clicking on a link is ok
      - Background requests like those initiated by scripts, iframes, reference to images etc are not ok
  - Default of Chrome browser

- None: disables SameSite restrictions altogether
  - Will send cookie in all requests
  - Default of most browsers
  - There are legitimate reasons for disabling SameSite (e.g. tracking cookies)
  - Set-Cookie: trackingId=0F8tgdOhi9ynR1M9wa3ODa; SameSite=None; Secure
    - Secure flag ensure cookies is only sent over HTTPs

Points to Note

- Good to set SameSite attribute on all cookies
  - Not just for CSRF protection
- Setting SameSite attribute to strict may however have usability issues!
  - Cookie of requests to your site generated from other websites will be stripped
  - E.g. Inbound links to your site will force users to log in again
    - Log back into Facebook every time somebody shared a video (on another website; video hosted on facebook)!
  - More useful to set SameSite attribute to Lax
    - At least allows GET requests from other sites to send cookies
    - Need to ensure GET requests are not vulnerable!

Flawed Implementations

- Lax Restriction: Convert POST to GET
  - Many servers support both GET or POST at a given endpoint (even form submission)
  - If however GET request isn't allowed, some frameworks allow override via _method parameter
  - However this still requires a top-level navigation

(Assumes no CSRF token being used)

```
<form
action="https://vulnerable.com/account/transfer-payment"
method="POST">
    <input type="hidden" name="_method" value="GET">
    <input type="hidden" name="recipient" value="hacker">
    <input type="hidden" name="amount" value="1000000">
</form>
```

Server
The form sends a POST request, but the server interprets the "_method" field and treats the request as if it were GET

GET /my-account/change-email?email=user%40example.com&_method=POST HTTP/1.1

(Browser will send cookie since it is GET; Server views this request as POST in case it is configured to accept only POST)

```
<script>
    document.location =
"https://vulnerable.com/my-account/change-email?email=user@example.com&_me
thod=POST";
</script>
```

(Above is the HTML payload that results in above GET; this conforms with top level navigation since it is changing the URL)

- Strict Restriction: Can overcome if there is a button that results in a secondary request within the same site
  - E.g. client-side redirect that dynamically constructs the redirection target using attacker-controllable input like URL parameters
  - Details in next slide

- Suppose ed-tech platform has a video and a discussion forum below to discuss the video
- When you add your comment, a POST request is sent “POST /fill/comment HTTP/1.1”
  - Body of post will contain message details as well as videoID (say 7)
- Response to POST, server redirects user (via GET) to a page /fill/comment/thankyou?videoId=7
  - This GET will result in a html page with a thank you message
  - Further the page has an embed javascript which will get the videoID (via url.searchParams.get("postId")) and constructs a URL “https://vulnerable.com/video/7
  - The javascript then changes the window location to this URL → GET request will go to this URL

Points to Note

- If anyone sends a GET request (first GET) to fill/comment/thankyou?videoId=7, they will get redirected to https://vulnerable.com/video/7 (second GET)
  - This is being done by embed javascript in response to the first GET
  - No need to post a comment for this action!
- If directory traversal is possible, and a GET request gets sent as fill/comment/thankyou?videoId=7/../../my-account/
  - One can land on the my-account page instead of video page
- Importantly the second GET request (constructed by javascipt) is originating from same site!!!

- Attacker Payload:

```
<script>
    document.location =
"https://vulnerable.com/fill/comment/thankyou?
videoId=1/../../my-account/change-email?email=
user%40example.com";
</script>
```

Defense: Referer Header

- Referer header: optional request header that contains the URL of the web page that contains the object which is being requested
  - Mispelled in standard :-)
  - Automatically added by browsers
- Verify that the request originated from the application's own domain via checking this field
- Least effective and subject to bypass
  - Various methods exist to instruct browser not to include this header
  - Often done for privacy reasons.

Defense: Reauthentication

- Require re-authentication for sensitive actions
  - Some websites force you to reconfirm your login details when you perform sensitive actions
    - Change your password or initiate a payment
- Gives user a clear indication that you're about to do something significant and potentially dangerous
- Also helps protect users if they accidentally leave themselves logged in on shared or stolen devices
- Particularly important for website that handle financial transactions or confidential data

Summary of Good Practices

- CSRF Tokens should satisfy the following
  - Unpredictable with high entropy (cryptographically secure pseudo-random number generator - CSPRNG)
  - Tied to user's session
  - Strictly validated in every case

- CSRF tokens should be treated as secrets and handled in a secure manner
  - Can send token to the client within a hidden field of an HTML form
    - Ideally be placed as early as possible within the HTML document
      - Before any locations where user-controllable data is embedded
  - Avoid placing token in the URL query string
    - Often logged in various locations on the client and server side
    - Is liable to be transmitted to third parties within the HTTP Referer header; and
    - Can be displayed on-screen within the user's browser.

- CSRF tokens should be validated as follows
  - When a CSRF token is generated, it should be stored server-side within the user's session data
  - When client presents a token,
    - Server should verify it matches the value that was stored in the user's session
    - Must be performed regardless of the HTTP method or content type of the request
    - If the request does not contain any token at all, it should be rejected as if an invalid token

- In addition to CSRF, use Strict SameSite
  - Use strict policy by default, lower to Lax only if really needed!
  - Avoid SameSite None unless you're fully aware of the security implications!
- As an end-user: Logout after work done

Real Life Examples

- Twitter Disconnect CSRF: https://hackerone.com/reports/111216
- Badoo’s Full account takeover using CSRF https://hackerone.com/reports/127703/

References

- https://portswigger.net/web-security/csrf
- https://owasp.org/www-community/attacks/csrf