Web Security II: Sessions and Requests, CSRF

CS1660 Introduction to Computer Security

What we know so far

- HTTP and Browsers
- Cookies (and what happens if you steal them)
- “Client-side controls”

Today

- More about requests: same-origin/cross-origin
- CSRF attacks
- Session token entropy

A generic web architecture

Review: Cookies

Key-value pairs (stored in browser) that keep track of certain information

- User preferences, session ID, session expiration, etc.
- Key attributes (so far):
  - **`Domain`**: eg. cs.brown.edu .brown.edu

Review: Cookies

Key-value pairs (stored in browser) that keep track of certain information

- User preferences, session ID, tracking, ad networks, etc.
- Key attributes (so far):
  - **`Domain`**: eg. cs.brown.edu .brown.edu

When a request is made, all cookies with a matching domain are sent with it
…subject to certain other browser restrictions (today's topic!)

Cookies: examples

- Session ID: cookie used for authentication
- App state: Shopping cart, page views
- Ad networks/tracking

…

Javascript

- Scripting language interpreted by browser
- Fetched as part of a page (just like HTML, images)

Javascript

- Scripting language interpreted by browser
- Fetched as part of a page (just like HTML, images)

Capabilities

- Read/modify web pages
  - DOM: Document Object Model
- Make requests asynchronously => dynamic content

Essential to all modern webpages

Javascript

```
<script type="text/javascript">
   function hello() { alert("Hello world!");}
</script>
```

Examples

- Read / modify elements of the DOM
  - “Look for all <p> tags and return their content”
  - “Change the content within all <img> tags to ______”
  - “Fetch resource at <URL> and add it to the page”
- Make web requests: `fetch()`, `XMLHTTPRequest()`
- Read cookies

```
alert(document.cookie);
```

Examples: Requests

Example: our demo site

A really poor website

PHP

Server-side web scripting language, first released 1993

```
index.php:
<!DOCTYPE html>
<html>
<head> <title>PHP "Hello, World!" program</title> </head>
  <body>
    <?php echo '<p>Hello, World!</p>'; ?>
 </body>
</html>
```

PHP

Server-side web scripting language, first released 1993

```
index.php:
<!DOCTYPE html>
<html>
<head> <title>PHP "Hello, World!" program</title> </head>
  <body>
    <?php echo '<p>Hello, World!</p>'; ?>
 </body>
</html>
```

⇒ Archaic, but still widely used

⇒ Same concepts apply to others!

According to a study by W3Techs:

As of 2024, PHP was in use by 76.5% of websites where the backend programming language could be detected

58.8% of these were using known-insecure PHP versions

Used by: Facebook, Wikipedia, Wordpress, …

Problems?

Problems?

Just like all software, modern pages are built from many components

- Load external objects from other sites (images, CSS)
- Load code from other sites
- Make requests to other sites

Also, we visit a lot of sites!

How to enable pages to load external resources?

How to keep code/data/cookies from one page from interfering with another?

How to enable pages to load external resources?

How to keep code/data/cookies from one page from interfering with another?

(… except when that's what we want)

Same origin policy (SOP): so far

- Limits how a site can set cookies
- Limits which cookies are sent on each request

In general, “origin” must match:

```
https://site.example.com[:443]/some/path
```

SOP: Requests

Websites can submit requests to another site (e.g., sending a GET / POST request, image embedding, `Javascript requests (XMLHttpRequest, fetch)`

- Can generally embed (display in browser) cross-origin response
  - Embedding an image
  - Opening content / opening the response to a request in an iframe

What can we do with this?

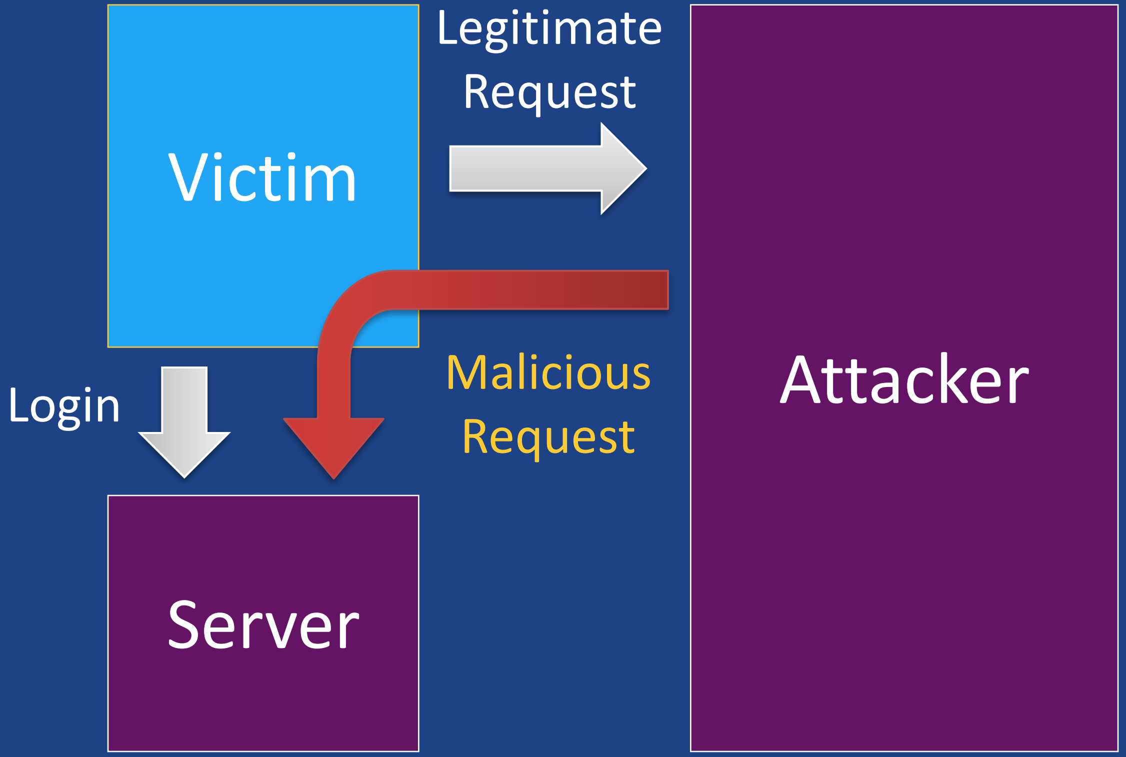
Break!

CSRF attacks

Browser performs unwanted action while user is authenticated

CSRF Mechanics

- Server trusts victim (login)
- Victim trusts attacker enough to click link/visit site
- Attacker could be a hacked legitimate site



CSRF: via GET

```
bad-site.com:

   <a href="http://bank.com/transfer.php&acct=1234?amt=1000.00?...
```

- Bad practice: state change info encoded in GET request
- Can easily "replay" request

CSRF: via POST

```
bad-site.com:
    <form action="https://bank.com/wiretransfer" method="POST"
          id=”bank">
    <input type="hidden" name="recipient" value="Attacker">
    <input type="hidden" name="account" value="2567">
    <input type="hidden" name="amount" value="$1000.00">
    …
    </form>
    document.getElementById(”bank").submit();
```

Is user is logged in, this will work!

CSRF Demo

How can we restrict which origins can make requests?

How can we restrict which origins can make requests?

Multiple mechanics, implemented at different layers of the system

=> Defense in depth!

Server-side: CSRF token

Server sends unguessable value to client, include as hidden variable in POST

```
<form action="/transfer.do" method="post">
<input type="hidden" name="csrf_token" value=”aXg3423fjp. . .">
[...]
</form>
```

On POST, server compares against expected value, rejects if wrong or missing

What does this prove?

CSRF Token: Mechanics

Different web frameworks handle tokens differently

- Set token per-session or per-request?
- Can include token directly in generated HTML, or use JS to set via cookie

CSRF Token: Mechanics

Different web frameworks handle tokens differently

- Set token per-session or per-request?
- Can include token directly in generated HTML, or use JS to set via cookie

How to generate the tokens?

- "Synchronizer token": server picks random value, saves for checking
- "Encrypted token": server sends encrypt/MAC of some value that can be checked without saving extra state (eg. user ID)

Limit cookie sharing

More info: Mozilla MDN

SameSite attribute: control how cookie is shared when origin is a different site:

```
Set-Cookie: sessionid=12345; Domain=b.com; SameSite=None
```

Limit cookie sharing

More info: Mozilla MDN

SameSite attribute: control how cookie is shared when origin is a different site:

```
Set-Cookie: sessionid=12345; Domain=b.com; SameSite=None
```

- **None**: No restrictions*
- `Strict`: Send cookie only when request originates from site that sent the cookie
- `Lax` (default since 2021): allow cross-site requests for requests *initiated by user (eg. clicking a link, but not Javascript)*

Limit cookie sharing

More info: Mozilla MDN

More important attributes:

```
Set-Cookie: sessionid=12345; . . . HttpOnly=true, Secure
```

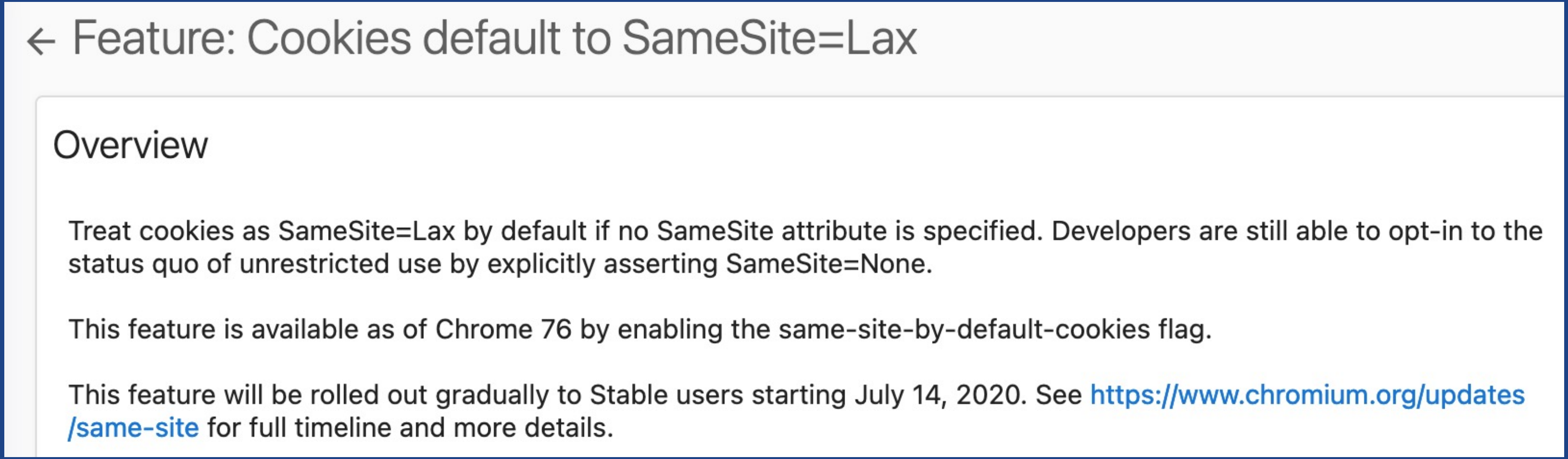
Limit cookie sharing

More info: Mozilla MDN

More important attributes:

```
Set-Cookie: sessionid=12345; . . . HttpOnly=true, Secure
```

- `Secure` (true/false): Only send this cookie when using HTTPS
- `HttpOnly` (true/false): If true, cookie can’t be read by Javascript (but can still be sent by requests)



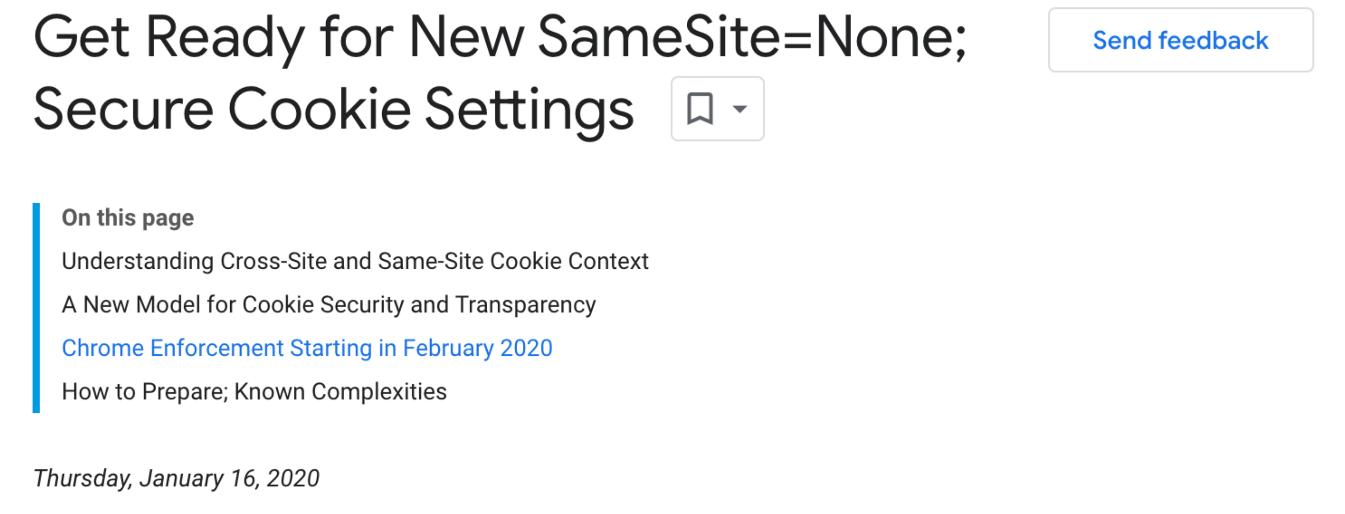
← Feature: Cookies default to SameSite=Lax

Overview

Treat cookies as SameSite=Lax by default if no SameSite attribute is specified. Developers are still able to opt-in to the status quo of unrestricted use by explicitly asserting SameSite=None.

This feature is available as of Chrome 76 by enabling the same-site-by-default-cookies flag.

This feature will be rolled out gradually to Stable users starting July 14, 2020. See https://www.chromium.org/updates/same-site for full timeline and more details.

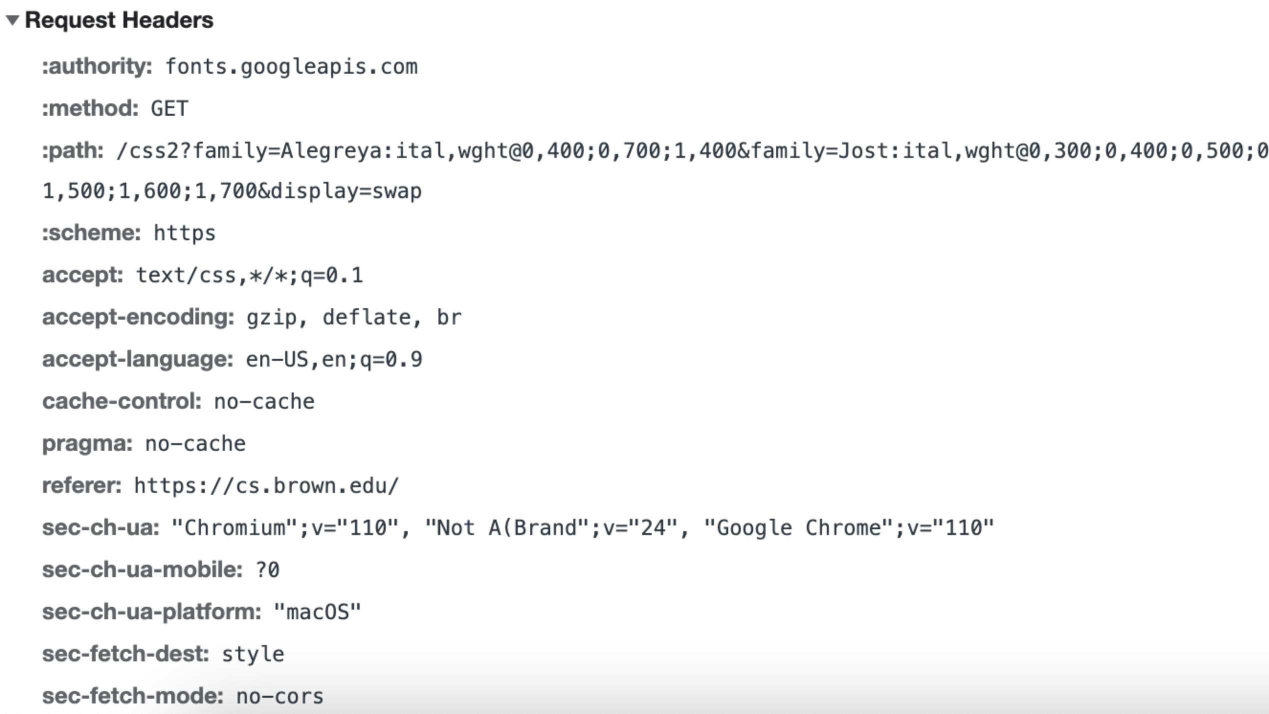


Get Ready for New SameSite=None; Secure Cookie Settings

Thursday, January 16, 2020

Another way: checking headers

"Referer" [sic] header: URL from which request is sent



▼ **Request Headers**

```
:authority: fonts.googleapis.com
:method: GET
:path: /css2?family=Alegreya:ital,wght@0,400;0,700;1,400&family=Jost:ital,wght@0,300;0,400;0,500;0
1,500;1,600;1,700&display=swap
:scheme: https
accept: text/css,*/*;q=0.1
accept-encoding: gzip, deflate, br
accept-language: en-US,en;q=0.9
cache-control: no-cache
pragma: no-cache
referer: https://cs.brown.edu/
sec-ch-ua: "Chromium";v="110", "Not A(Brand";v="24", "Google Chrome";v="110"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "macOS"
sec-fetch-dest: style
sec-fetch-mode: no-cors
```

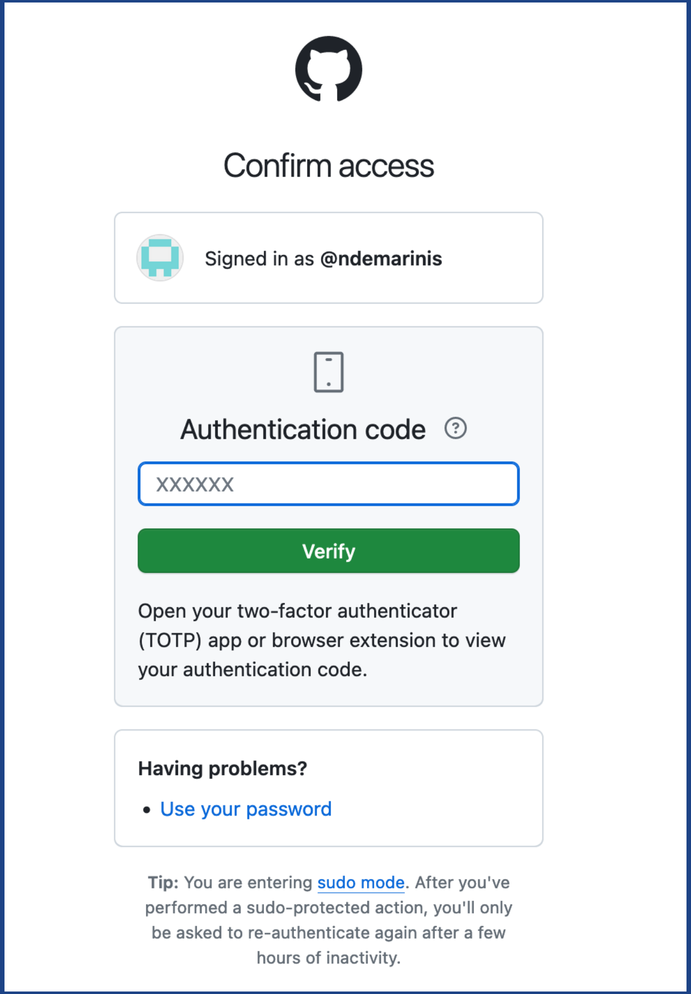
Another way: checking headers

- Check Referer header on request, see if it matches expected origin
- Browser limits how Referer header can be changed

=> Useful if you trust browser; but ultimately can be controlled by client

User Interaction

Force certain high-value operations to require use input



Confirm access

Signed in as **@ndemarinis**

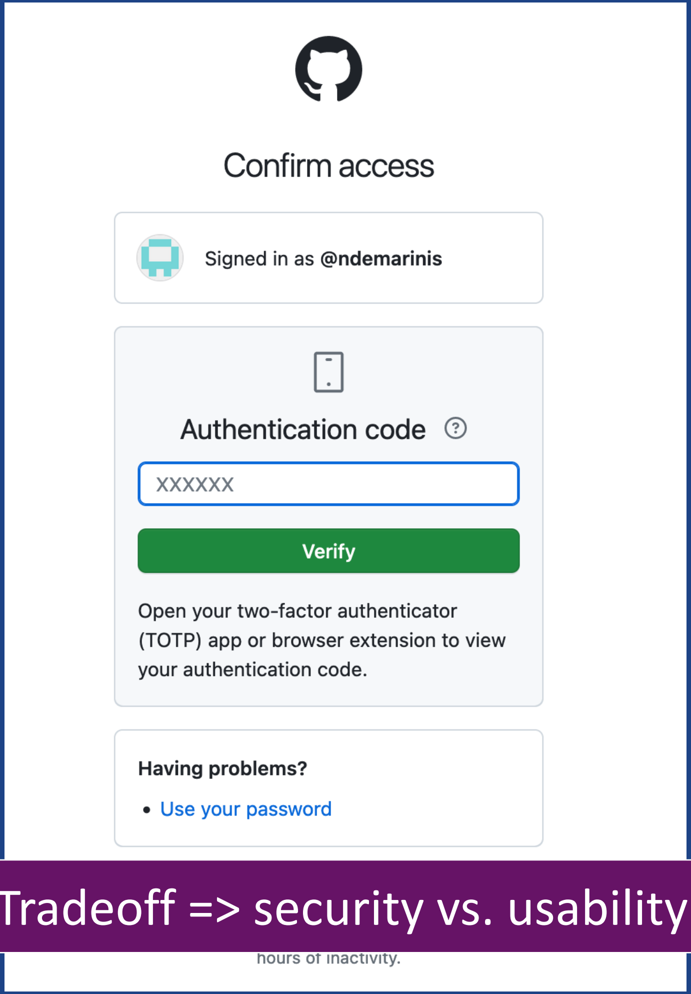
Authentication code

Open your two-factor authenticator (TOTP) app or browser extension to view your authentication code.

Having problems?

- Use your password

Tip: You are entering sudo mode. After you've performed a sudo-protected action, you'll only be asked to re-authenticate again after a few hours of inactivity.



Confirm access

Signed in as **@ndemarinis**

Authentication code

XXXXXX

Open your two-factor authenticator (TOTP) app or browser extension to view your authentication code.

Having problems?

- Use your password

hours of inactivity.

Tradeoff => security vs. usability

CORS: Cross-Origin Resource Sharing

Systematic way to set permissions for cross-origin requests for most dynamic resources (Javascript and others)

CORS: Cross-Origin Resource Sharing

Systematic way to set permissions for cross-origin requests for most dynamic resources (Javascript and others):

```
# Allow origin example.com to use resources from here
Access-Control-Allow-Origin: https://example.com

# Allow any origin to use resources from here
Access-Control-Allow-Origin: *
```

If Origin not allowed by header,
browser prevents page from using resource
=> Browser must implement this properly!

CORS: Further reading

Overview here: Mozilla MDN

- Gained adoption in major browsers 2009-2015
- Requires site owners to define *policies* for how resources are used
- For some requests, browser will do a “preflight” request to see if authorized first
- Extra nuances for requests that send cookies “credentialed” requests

What We Have Learned

- Motivation and specifications for session management
- Session ID implementations
  - Cookie
  - GET variable
  - POST variable
- Cross-Site Request Forgery (CSRF) attack
- CSRF mitigation techniques

Potential issues

- SameSite attribute set to Strict:
  - the browser will not include the cookie in any requests that originate from another site.
- A logged-in user follows a third-party link to a site:
  - they will appear not to be logged in, and will need to log in again before interacting with the site in the normal way
- Potential problems for usability and user tracking (e.g. Ads)