Sessions, CSRF and XSS

CS 161 Fall 2024

Last Time: Cookies

- Cookie: a piece of data used to maintain state across multiple requests
  - Set by the browser or server
  - Stored by the browser
  - Attributes: Name, value, domain, path, secure, HttpOnly, expires
- Cookie policy
  - Server with domain X can set a cookie with domain attribute Y if the domain attribute is a **domain suffix** of the server's domain, and the domain attribute Y is not a top-level domain (TLD)
  - The browser attaches a cookie on a request if the domain attribute is a **domain suffix** of the server's domain, and the path attribute is a **prefix** of the server's path
  - Cookie domain: example.com and cookie path: /some/path will be included in request to https://foo/example/com/some/path/index.html

Session Authentication

- **Session**: A sequence of requests and responses associated with the same authenticated user
  - Example: When you check all your unread emails, you make many requests to Gmail. The Gmail server needs a way to know all these requests are from you
  - When the session is over (you log out, or the session expires), future requests are not associated with you
- Naïve solution: Type your username and password before each request
  - Problem: Very inconvenient for the user!
- Better solution: Is there a way the browser can automatically send some information in a request for us?
  - Yes: Cookies!

Session Authentication: Intuition

- Imagine you're attending a concert
- The first time you enter the venue:
  - Present your ticket and ID
  - The doorperson checks your ticket and ID
  - If they're valid, you receive a wristband
- If you leave and want to re-enter later
  - Just show your wristband!
  - No need to present your ticket and ID again

Session Tokens

- **Session token**: A secret value used to associate requests with an authenticated user
- The first time you visit the website:
  - Present your username and password
  - If they're valid, you receive a session token
  - The server associates you with the session token
- When you make future requests to the website:
  - Attach the session token in your request
  - The server checks the session token to figure out that the request is from you
  - No need to re-enter your username and password!

Session Tokens with Cookies

- Session tokens can be implemented with cookies
  - Cookies can be used to save *any* state across requests (e.g. dark mode)
  - Session tokens are just one way to use cookies
- The first time you visit a website:
  - Make a request with your username and password
  - If they're valid, the server sends you a cookie with the session token
  - The server associates you with the session token
- When you make future requests to the website:
  - The browser attaches the session token cookie in your request
  - The server checks the session token to figure out that the request is from you
  - No need to re-enter your username and password!
- When you log out (or when the session times out):
  - The browser and server delete the session token

Session Tokens: Security

- If an attacker steals your session token, they can log in as you!
  - The attacker can make requests and attach your session token
  - The browser will think the attacker's requests come from you
- Servers need to generate session tokens *randomly* and *securely*
- Browsers need to make sure malicious websites cannot steal session tokens
  - Enforce isolation with cookie policy and same-origin policy
- Browsers should not send session tokens to the wrong websites
  - Enforced by cookie policy

Session Token
Cookie Attributes

- What attributes should the server set for the session token?
  - Domain and Path: Set so that the cookie is only sent on requests that require authentication
  - Secure: Can set to True to so the cookie is only sent over secure HTTPS connections
  - HttpOnly: Can set to True so JavaScript can’t access session tokens
  - Expires: Set so that the cookie expires when the session times out

Name	`token`
Value	`{random value}`
Domain	`mail.google.com`
Path	`/`
Secure	`True`
HttpOnly	`True`
Expires	`{15 minutes later}`
(*other fields omitted*)	

Cross-Site Request Forgery (CSRF)

Review: Cookies and Session Tokens

- Session token cookies are used to associate a request with a user
- The browser automatically attaches relevant cookies in every request

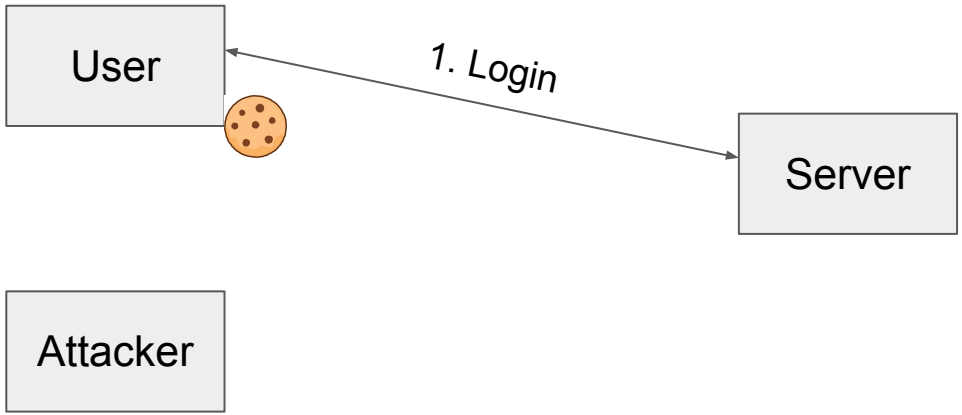
Cross-Site Request Forgery (CSRF)

- Idea: What if the attacker tricks the victim into making an unintended request?
  - The victim's browser will automatically attach relevant cookies
  - The server will think the request came from the victim!
- **Cross-site request forgery (CSRF or XSRF)**: An attack that exploits cookie-based authentication to perform an action as the victim

Steps of a CSRF Attack

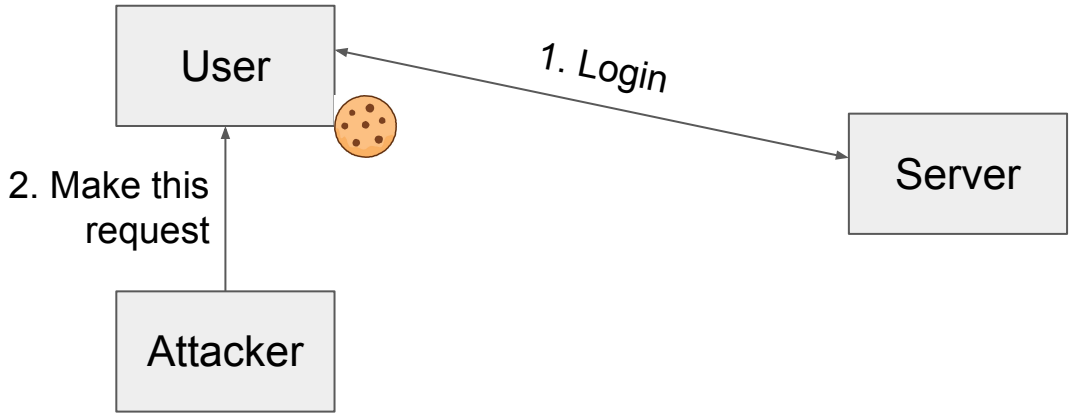
Steps of a CSRF Attack

1. User authenticates to the server
    - User receives a cookie with a valid session token



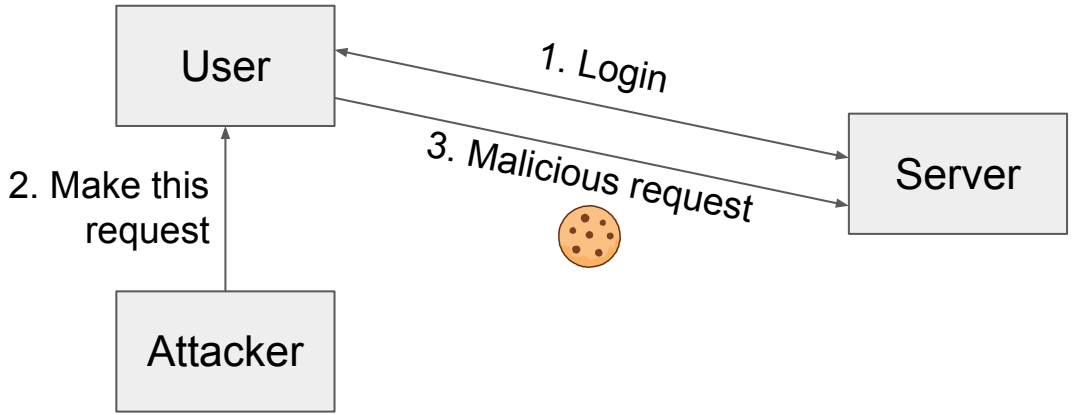
Steps of a CSRF Attack

1. User authenticates to the server
    - User receives a cookie with a valid session token
2. Attacker tricks the victim into making a malicious request to the server



Steps of a CSRF Attack

1. User authenticates to the server
    - User receives a cookie with a valid session token
2. Attacker tricks the victim into making a malicious request to the server
3. The server accepts the malicious request from the victim
    - Recall: The cookie is automatically attached in the request



Steps of a CSRF Attack

1. User authenticates to the server
    - User receives a cookie with a valid session token
2. Attacker tricks the victim into making a malicious request to the server
3. The server accepts the malicious request from the victim
    - Recall: The cookie is automatically attached in the request

Executing a CSRF Attack

- How might we trick the victim into making a GET request?
- Strategy #1: Trick the victim into clicking a link
  - Later we'll see how to trick a victim into clicking a link
  - The link can directly make a GET request:
    `https://www.bank.com/transfer?amount=100&to=Mallory`
  - The link can open an attacker's website, which contains some JavaScript that makes the actual malicious request
- Strategy #2: Put some HTML on a website the victim will visit
  - Example: The victim will visit a forum. Make a post with some HTML on the forum
  - HTML to automatically make a GET request to a URL:
    `<img src="https://www.bank.com/transfer?amount=100&to=Mallory">`
    - This HTML will probably return an error or a blank 1 pixel by 1 pixel image, but the GET request will still be sent...with the relevant cookies!

Executing a CSRF Attack

- How might we trick the victim into making a POST request?
  - Example POST request: Submitting a form
- Strategy #1: Trick the victim into clicking a link
  - Note: Clicking a link in your browser makes a GET request, not a POST request, so the link cannot directly make the malicious POST request
  - The link can open an attacker's website, which contains some JavaScript that makes the actual malicious POST request
- Strategy #2: Put some JavaScript on a website the victim will visit
  - Example: Pay for an advertisement on the website, and put JavaScript in the ad
  - Recall: JavaScript can make a POST request

Top 25 Most Dangerous Software Weaknesses (2020)

Rank	ID	Name	Score
[1]	CWE-79	Improper Neutralization of Input During Web Page Generation (’Cross-site Scripting’)	46.82
[2]	CWE-787	Out-of-bounds Write	46.17
[3]	CWE-20	Improper Input Validation	33.47
[4]	CWE-125	Out-of-bounds Read	26.50
[5]	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	23.73
[6]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command (’SQL Injection’)	20.69
[7]	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	19.16
[8]	CWE-416	Use After Free	18.87
[9]	CWE-352	Cross-Site Request Forgery (CSRF)	17.29
[10]	CWE-78	Improper Neutralization of Special Elements used in an OS Command (’OS Command Injection’)	16.44
[11]	CWE-190	Integer Overflow or Wraparound	15.81
[12]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory (’Path Traversal’)	13.67
[13]	CWE-476	NULL Pointer Dereference	8.35
[14]	CWE-287	Improper Authentication	8.17
[15]	CWE-434	Unrestricted Upload of File with Dangerous Type	7.38
[16]	CWE-732	Incorrect Permission Assignment for Critical Resource	6.95
[17]	CWE-94	Improper Control of Generation of Code (’Code Injection’)	6.53

CSRF Example: YouTube

- 2008: Attackers exploit a CSRF vulnerability on YouTube
- By forcing the victim to make a request, the attacker could:
  - Add any videos to the victim's "Favorites"
  - Add any user to the victim's "Friend" or "Family" list
  - Send arbitrary messages as the victim
  - Make the victim flag any videos as inappropriate
  - Make the victim share a video with their contacts
  - Make the victim subscribe to any channel
  - Add any videos to the user's watchlist
- **Takeaway**: With a CSRF attack, the attacker can force the victim to perform a wide variety of actions!

CSRF Example: Facebook

InternetNews.com

Link

Facebook Hit by Cross-Site Request Forgery Attack

Sean Michael Kerner *August 21, 2009*

Nevertheless, that Facebook accounts were compromised in the wild is noteworthy because the attack used a legitimate HTML tag to violate users' privacy.

According to Zilberman's disclosure, the attack simply involved the malicious HTML image tag residing on any site, including any blog or forum that permits the use of image tags even in the comments section.

"The attack elegantly ends with a valid image so the page renders normally, and the attacked user does not notice that anything peculiar has happened," Zilberman said.

Takeaway: The HTML image tag can be used to execute a CSRF attack

CSRF Defenses

CSRF Defenses

- CSRF defenses are implemented by the server (not the browser)
- Defense: CSRF tokens
- Defense: Referer validation
- Defense: SameSite cookie attribute

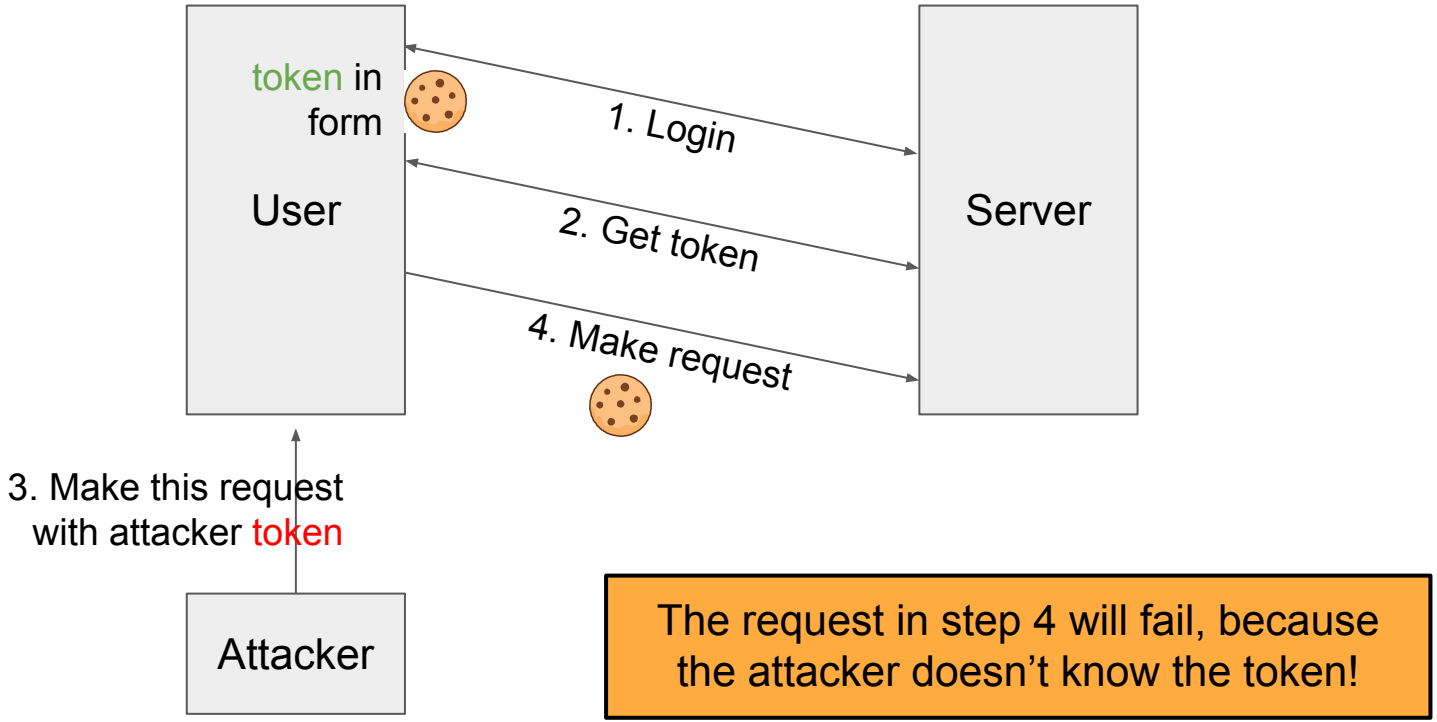
CSRF Tokens

- Idea: Add a secret value in the request that the attacker doesn’t know
  - The server only accepts requests if it has a valid secret
  - Now, the attacker can’t create a malicious request without knowing the secret
- **CSRF token**: A secret value provided by the server to the user. The user must attach the same value in the request for the server to accept the request.
  - CSRF tokens cannot be sent to the server in a cookie!
    - The token must be sent somewhere else (e.g. a header, GET parameter, or POST content)
  - CSRF tokens are usually valid for only one or two requests

CSRF Tokens: Usage

- Example: HTML forms
  - Forms are vulnerable to CSRF
    - If the victim visits the attacker's page, the attacker's JavaScript can make a POST request with a filled-out form
- CSRF tokens are a defense against this attack
  - Every time the user requests a form from the legitimate website, the server attaches a CSRF token as a *hidden form field* (in the HTML, but not visible to the user)
  - When the user submits the form, the form contains the CSRF token
  - The attacker's JavaScript won't be able to create a valid form, because they don't know the CSRF token!
  - The attacker can try to fetch their own CSRF token, but it will only be valid for the attacker, not the victim

CSRF Tokens: Usage



The request in step 4 will fail, because the attacker doesn't know the token!

Referer Header

- Idea: In a CSRF attack, the victim usually makes the malicious request from a different website
- Referer header: A header in an HTTP request that indicates which webpage made the request
  - “Referer” is a 30-year typo in the HTTP standard (supposed to be “Referrer”)!
  - Example: If you type your username and password into the Facebook homepage, the Referer header for that request is `https://www.facebook.com`
  - Example: If an `img` HTML tag on a forum forces your browser to make a request, the Referer header for that request is the forum’s URL
  - Example: If JavaScript on an attacker’s website forces your browser to make a request, the Referer header for that request is the attacker’s URL

Referer Header

- Checking the Referer header
  - Allow **same-site requests**: The Referer header matches an expected URL
    - Example: For a login request, expect it to come from `https://bank.com/login`
  - Disallow **cross-site requests**: The Referer header does not match an expected URL
- If the server sees a cross-site request, reject it

Referer Header: Issues

- The Referer header may leak private information
  - Example: If you made the request on a top-secret website, the Referer header might show you visited `http://intranet.corp.apple.com/projects/iphone/competitors.html`
  - Example: If you make a request to an advertiser, the Referer header gives the advertiser information about how you saw the ad
- The Referer header might be removed before the request reaches the server
  - Example: Your company firewall removes the header before sending the request
  - Example: The browser removes the header because of your privacy settings
- The Referer header is optional. What if the request leaves the header blank?
  - Allow requests without a header?
    - Less secure: CSRF attacks might be possible
  - Deny requests without a header?
    - Less usable: Legitimate requests might be denied
  - Need to consider fail-safe defaults: No clear answer

SameSite Cookie Attribute

- Idea: Implement a flag on a cookie that makes it unexploitable by CSRF attacks
  - This flag must specify that **cross-site** requests will not contain the cookie
- **SameSite flag**: A flag on a cookie that specifies it should be sent only when the domain of the cookie **exactly** matches the domain of the origin
  - SameSite=None: No effect
  - SameSite=Strict: The cookie will not be sent if the cookie domain does not match the origin domain
  - Example: If **`https://evil.com/`** causes your browser to make a request to **`https://bank.com/transfer?to=mallory`**, cookies for bank.com will not be sent if SameSite=Strict, because the origin domain (**`evil.com`**) and cookie domain (**`bank.com`**) are different
- Issue: Not yet implemented on all browsers

Cookies: Summary

- Cookie: a piece of data used to maintain state across multiple requests
  - Set by the browser or server
  - Stored by the browser
  - Attributes: Name, value, domain, path, secure, HttpOnly, expires
- Cookie policy
  - Server with domain X can set a cookie with domain attribute Y if the domain attribute is a **domain suffix** of the server's domain, and the domain attribute Y is not a top-level domain (TLD)
  - The browser attaches a cookie on a request if the domain attribute is a **domain suffix** of the server's domain, and the path attribute is a **prefix** of the server's path

Session Authentication: Summary

- Session authentication
  - Use cookies to associate requests with an authenticated user
  - First request: Enter username and password, receive session token (as a cookie)
  - Future requests: Browser automatically attaches the session token cookie
- Session tokens
  - If an attacker steals your session token, they can log in as you
  - Should be randomly and securely generated by the server
  - The browser should not send tokens to the wrong place

CSRF: Summary

- Cross-site request forgery (CSRF or XSRF): An attack that exploits cookie-based authentication to perform an action as the victim
  - User authenticates to the server
    - User receives a cookie with a valid session token
  - Attacker tricks the victim into making a malicious request to the server
  - The server accepts the malicious request from the victim
    - Recall: The cookie is automatically attached in the request
- Attacker must trick the victim into creating a request
  - GET request: click on a link
  - POST request: use JavaScript

CSRF Defenses: Summary

- CSRF token: A secret value provided by the server to the user. The user must attach the same value in the request for the server to accept the request.
  - The attacker does not know the token when tricking the user into making a request
- Referer Header: Allow same-site requests, but disallow cross-site requests
  - Header may be blank or removed for privacy reasons
- Same-site cookie attribute: The cookie is sent only when the domain of the cookie exactly matches the domain of the origin
  - Not implemented on all browsers

Cross-Site Scripting (XSS)

Textbook Chapter 22

Top 25 Most Dangerous Software Weaknesses (2020)

Rank	ID	Name	Score
[1]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46.82
[2]	CWE-787	Out-of-bounds Write	46.17
[3]	CWE-20	Improper Input Validation	33.47
[4]	CWE-125	Out-of-bounds Read	26.50
[5]	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	23.73
[6]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	20.69
[7]	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	19.16
[8]	CWE-416	Use After Free	18.87
[9]	CWE-352	Cross-Site Request Forgery (CSRF)	17.29
[10]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	16.44
[11]	CWE-190	Integer Overflow or Wraparound	15.81
[12]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	13.67
[13]	CWE-476	NULL Pointer Dereference	8.35
[14]	CWE-287	Improper Authentication	8.17
[15]	CWE-434	Unrestricted Upload of File with Dangerous Type	7.38
[16]	CWE-732	Incorrect Permission Assignment for Critical Resource	6.95
[17]	CWE-94	Improper Control of Generation of Code ('Code Injection')	6.53

Review: Same-Origin Policy

- Two webpages with different origins should not be able to access each other's resources
  - Example: JavaScript on **`http://evil.com`** cannot access the information on **`http://bank.com`**

Review: JavaScript

- **JavaScript**: A programming language for running code in the web browser
- JavaScript is **client-side**
  - Code sent by the server as part of the response
  - Runs in the browser, not the web server!
- Used to manipulate web pages (HTML and CSS)
  - Makes modern websites interactive
  - JavaScript can be directly embedded in HTML with `<script>` tags
- Most modern webpages involve JavaScript
  - JavaScript is supported by all modern web browsers
- You don’t need to know JavaScript syntax
  - However, knowing common attack functions helps

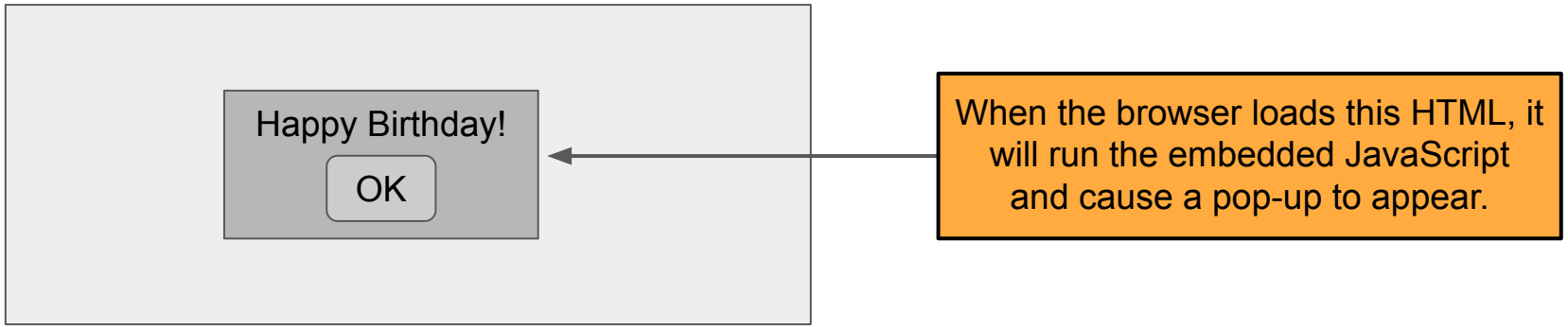
Review: JavaScript

- JavaScript can create a pop-up message

HTML (with embedded JavaScript)

```
<script>alert("Happy Birthday!")</script>
```

Webpage



A Go HTTP Handler

Handler

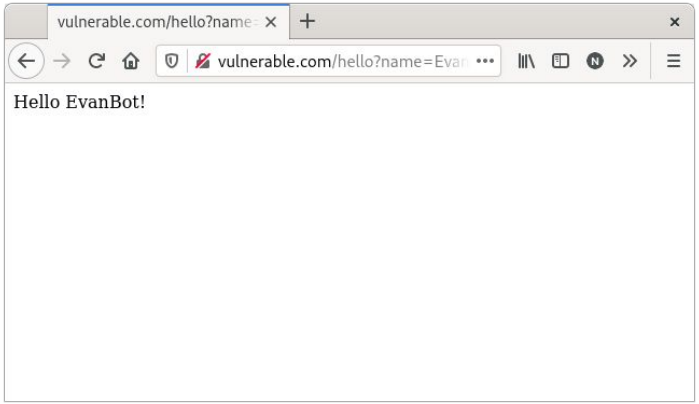
```
func handleSayHello(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Query()["name"][0]
    fmt.Fprintf(w, "<html><body>Hello %s!</body></html>", name)
}
```

URL

```
https://vulnerable.com/hello?name=EvanBot
```

Response

```
<html><body>Hello EvanBot!</body></html>
```



A Go HTTP Handler

Handler

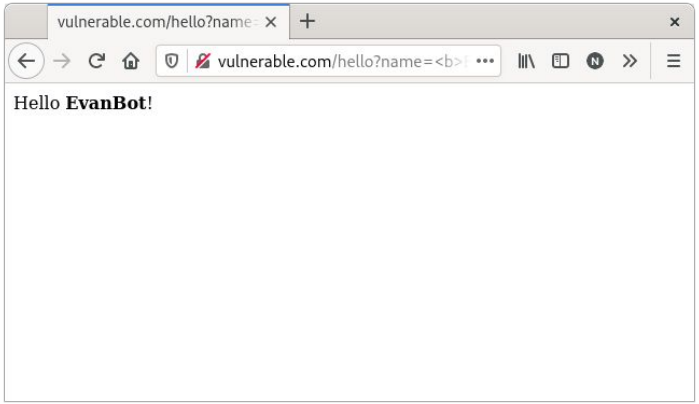
```
func handleSayHello(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Query()["name"][0]
    fmt.Fprintf(w, "<html><body>Hello %s!</body></html>", name)
}
```

URL

```
https://vulnerable.com/hello?name=<b>EvanBot</b>
```

Response

```
<html><body>Hello <b>EvanBot</b>!</body></html>
```



A Go HTTP Handler

Problem: This input represents control data (HTML), not just text!

Handler

```
func handleSayHello(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Query()["name"][0]
    fmt.Fprintf(w, "<html><body>Hello %s!</body></html>", name)
}
```

URL

```
https://vulnerable.com/hello?name=<script>alert(1)</script>
```

Response

```
<html><body>Hello <script>alert(1)</script>!</body></html>
```

A Go HTTP Handler

Not just `%s`: It can happen with any string manipulation

Handler

```
func handleSayHello(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Query()["name"][0]
    content := "<html><body>Hello "+name+"!</body></html>"
    fmt.Fprint(w, content)
}
```

URL

```
https://vulnerable.com/hello?name=<script>alert(1)</script>
```

Response

```
<html><body>Hello <script>alert(1)</script>!</body></html>
```

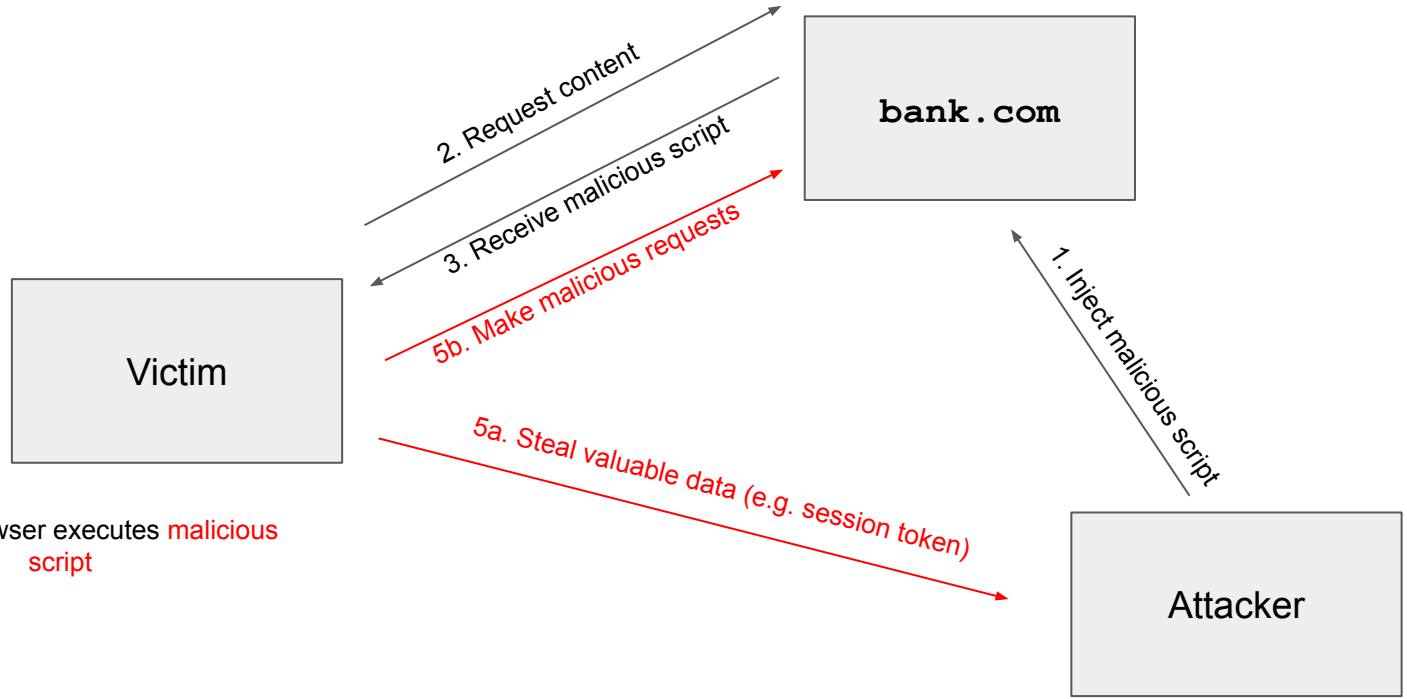
Cross-Site Scripting (XSS)

- Idea: The attacker adds malicious JavaScript to a legitimate website
  - The legitimate website will send the attacker's JavaScript to browsers
  - The attacker's JavaScript will run with the origin of the legitimate website
  - Now the attacker's JavaScript can access information on the legitimate website!
- **Cross-site scripting** (**XSS**): Injecting JavaScript into websites that are viewed by other users
  - Cross-site scripting subverts the same-origin policy
- Two main types of XSS
  - Stored XSS
  - Reflected XSS

Stored XSS

- **Stored XSS** (**persistent XSS**): The attacker's JavaScript is **stored** on the legitimate server and sent to browsers
- Classic example: Facebook pages
  - Anybody can load a Facebook page with content provided by users
  - An attacker puts some JavaScript on their Facebook page
  - Anybody who loads the attacker's page will see JavaScript (with the origin of Facebook)
- Stored XSS requires the victim to load the page with injected JavaScript
- Remember: Stored XSS is a server-side vulnerability!

Stored XSS

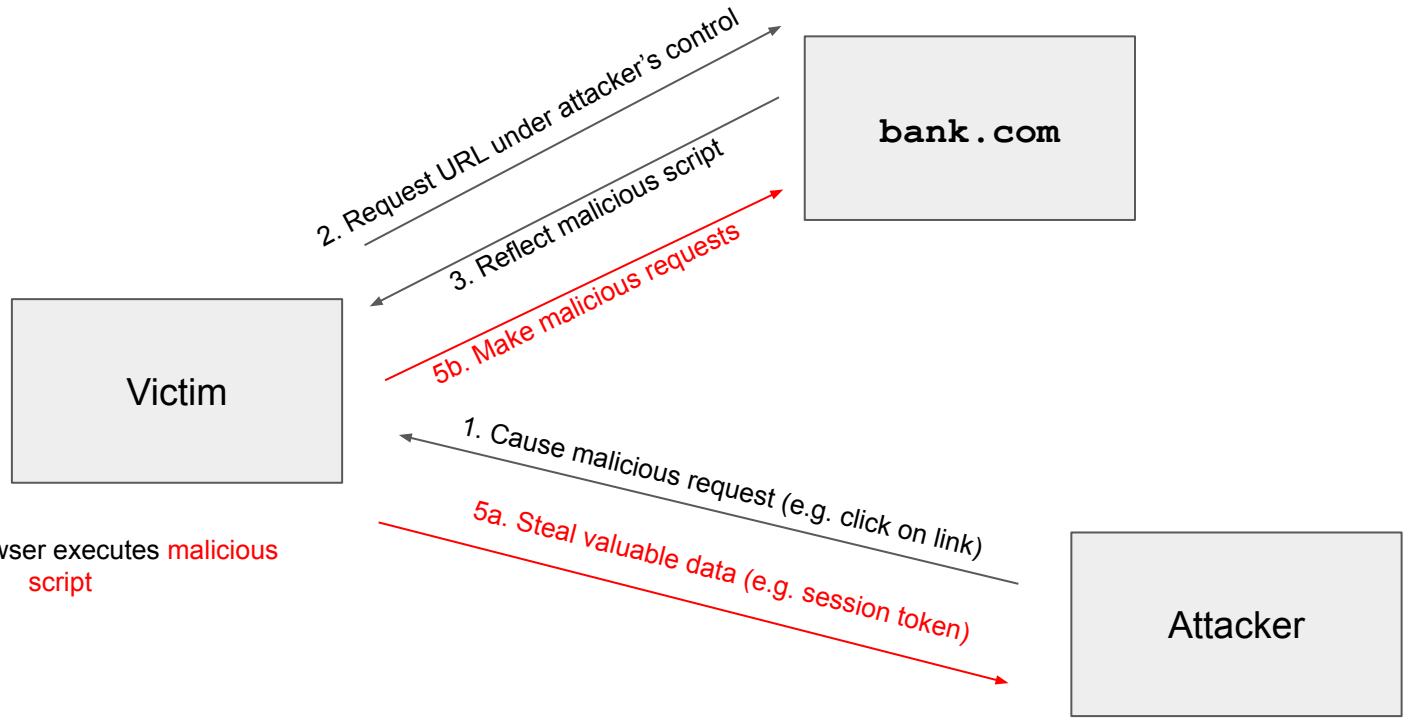


4. Victim browser executes malicious script

Reflected XSS

- **Reflected XSS**: The attacker causes the victim to input JavaScript into a request, and the content is **reflected** (copied) in the response from the server
- Classic example: Search
  - If you make a request to `http://google.com/search?q=evanbot`, the response will say "10,000 results for evanbot"
  - If you make a request to `http://google.com/search?q=<script>alert(1)</script>`, the response will say "10,000 results for `<script>alert(1)</script>`"
- Reflected XSS requires the victim to make a request with injected JavaScript

Reflected XSS



4. Victim browser executes malicious script

Reflected XSS: Making a Request

- How do we force the victim to make a request to the legitimate website with injected JavaScript?
  - Trick the victim into visiting the attacker's website, and include an embedded iframe that makes the request
    - Can make the iframe very small (1 pixel x 1 pixel), so the victim doesn't notice it:

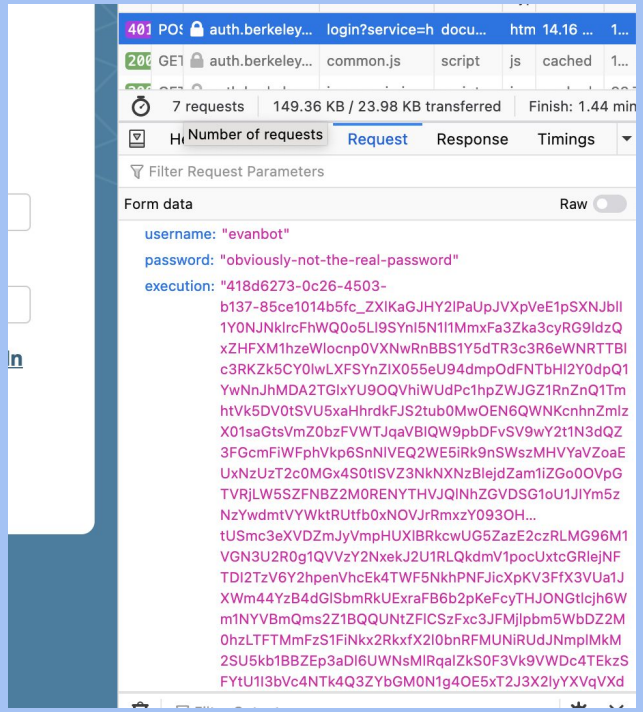
      ```
      <iframe height=1 width=1
      src="http://google.com/search?q=<script>alert(1)</script>">
      ```
  - Trick the victim into clicking a link (e.g. posting on social media, sending a text, etc.)
  - Trick the victim into visiting the attacker's website, which redirects to the reflected XSS link
  - … and many more!

Reflected XSS is not CSRF

- Reflected XSS and CSRF both require the victim to make a request to a link
  - XSS: An HTTP response contains maliciously inserted JavaScript, executed on the client side
  - CSRF: A malicious HTTP request is made (containing the user's cookies), executing an effect on the server side

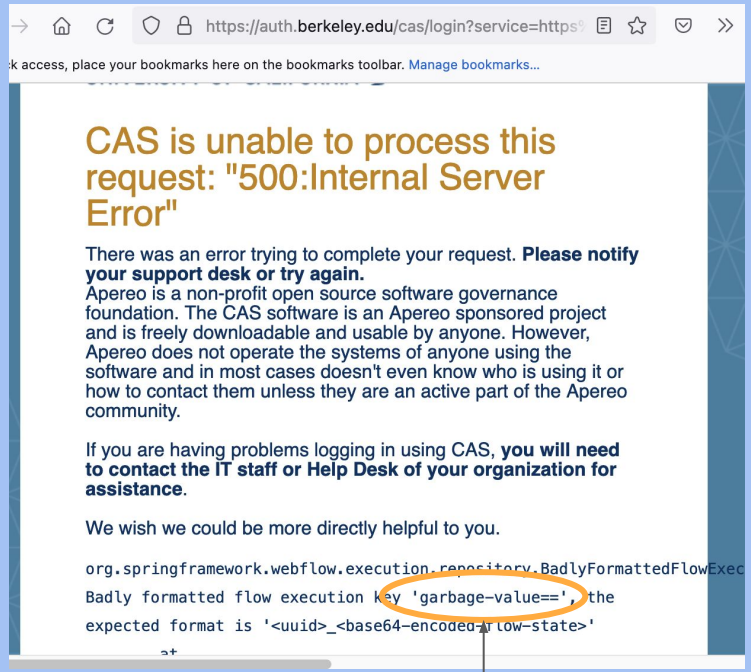
XSS in the Wild… On CalNet?!

- In 2021, 61C student Rohan Mathur was exploring the CalNet login page
- Fields submitted when logging in:
  - **`username`**: What user am I logging in as?
  - **`password`**: What's the user's password?
  - **`execution`**: Server-side state (like a CSRF token with extra data)



XSS in the Wild… On CalNet?!

- The server expects **`execution`** to be in a specific format that contains its server-side state
  - What if you muck with **`execution`**?
- The “corrupted” execution key is placed into the error message in the HTML to aid debugging
  - … and CalNet at the time did not escape the execution key



Constructing an Attack on CalNet

Attack: Force a POST request to CalNet!

```
<html>
  <head>
    <script>
      // When the malicious page finishes loading, automatically submit the form!
      document.addEventListener('DOMContentLoaded', () => {
        document.getElementById('form').submit();
      });
    </script>
  </head>
  <body>
    <!-- Malicious form containing our malicious execution data. -->
    <form id="form" action="https://auth.berkeley.edu/cas/login" method="POST">
      <input name="username" type="text" value="evanbot" />
      <input name="password" type="text" value="obviously-not-the-real-password" />
      <input name="execution" type="text" value="<script>alert('XSS!')</script>" />
    </form>
  </body>
</html>
```

So What Happened?

- CalNet also accepts **`execution`** parameters over URL query parameters
  - A link like **`https://auth.berkeley.edu/cas/login?execution=<script>alert('XSS!')</script>`** would result in the same attack
- It would only fire if you clicked the button to show the error
- … But if clicked, the injected JavaScript could
  - Present a fake login prompt
  - Steal your CalNet password
  - Log you in as the attacker's account
  - Steal your authentication session token
- Root cause: A >5 year old sample page modified by CalNet
  - CalNet uses software from Apereo, which comes with sample pages for logging in
  - Apereo fixed that bug many years before, but sample pages don't get included in bug fixes!

XSS Defenses

```
<html>
<body>
Hello &lt;script&gt;alert(1)&lt;/script&gt;!
</body>
</html>
```

- Defense: **HTML sanitization**
  - Idea: Certain characters are special, so create sequences that represent those characters as data, rather than as HTML
- Start with an ampersand (`&`) and end with a semicolon (`;`)
  - Instead of `<`, use `<`
  - Instead of `"`, use `&quot;`
  - And many more!
    - It is important to escape all dangerous characters (lists of them can be found), or you will still be vulnerable!
- Note: You should always rely on trusted libraries to do this for you!

XSS Defenses: Escaping

Handler

```
func handleSayHello(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Query()["name"][0]
    fmt.Fprintf(w, "<html><body>Hello %s!</body></html>", html.EscapeString(name))
}
```

URL

```
https://vulnerable.com/hello?name=<script>alert(1)</script>
```

Response

```
<html><body>Hello &lt;script&gt;alert(1)&lt;/script&gt;!</body></html>
```

XSS Defenses: Escaping

```
<html>
<body>
Hello {{.name}}!
</body>
</html>
```

Example: Golang HTML template

- If a programmer has to take an action for every usage…
  - They are *going* to screw up (e.g. CalNet)
  - Recall: Consider human factors!
- Nowadays, escaping is generally achieved through **templating**
  - HTML templates are essentially their own language, where you declare what data goes where
  - The templating engine handles all the escaping internally
  - The HTTP library gets very angry if you don't use templates
  - Recall (again): Consider human factors!

XSS Defenses: CSP

- Defense: **Content Security Policy** (**CSP**)
  - Idea: Instruct the browser to only use resources loaded from specific places
  - Uses additional headers to specify the policy
- Standard approach:
  - Disallow all inline scripts (JavaScript code directly in HTML), which prevents inline XSS
    - Example: Disallow `<script>alert(1)</script>`
  - Only allow scripts from specified domains, which prevents XSS from linking to external scripts
    - Example: Disallow `<script src="https://cs161.org/hack.js">`
- Also works with other content (e.g. iframes, images, etc.)
- Relies on the browser to enforce security, so more of a mitigation for defense-in-depth

Summary: XSS

- Websites use untrusted content as control data
  - `<html><body>Hello EvanBot!</body></html>`
  - `<html><body>Hello <script>alert(1)</script>!</body></html>`
- Stored XSS
  - The attacker's JavaScript is stored on the legitimate server and sent to browsers
  - Classic example: Make a post on a social media site (e.g. Facebook) with JavaScript
- Reflected XSS
  - The attacker causes the victim to input JavaScript into a request, and the content it's reflected (copied) in the response from the server
  - Classic example: Create a link for a search engine (e.g. Google) query with JavaScript
  - Requires the victim to click on the link with JavaScript

Summary: XSS Defenses

- Defense: HTML sanitization
  - Replace control characters with data sequences
    - < becomes `<`
    - " becomes `&quot;`
  - Use a trusted library to sanitize inputs for you
- Defense: Content Security Policy (CSP)
  - Instruct the browser to only use resources loaded from specific places
  - Limits JavaScript: only scripts from trusted sources are run in the browser
  - Enforced by the browser