CS 370: Introduction to Security
05.16: Advanced web security I, II

Tu/Th 4:00 – 5:50 PM

Sanghyun Hong

sanghyun.hong@oregonstate.edu

Oregon State University

SAIL
Secure AI Systems Lab

Heads-up

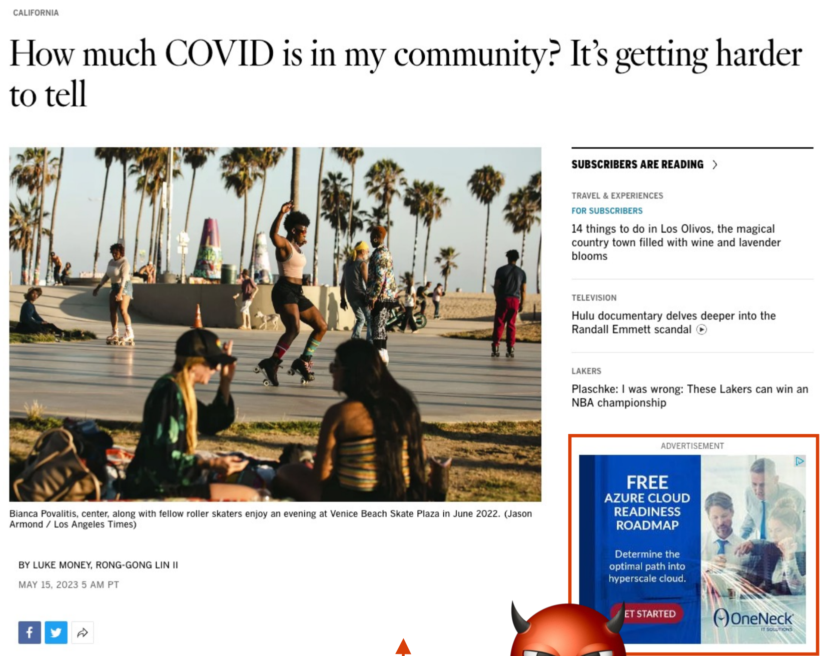
- Updates
  - Quizzes
    - Quiz 2: 5/13-23 → 5/23-30
    - Quiz 3: 6/13-15 → 6/8-15
  - Micro-labs
    - Internet security : 5/23 → 5/30
    - Trustworthy ML : 6/13 → 6/15
    - Usable security : Removed (full points for everyone)

Topics for today

- Advanced web security
  - Same-origin policy
    - Motivation
    - Same-origin policy
    - Weaknesses
  - XSS (Cross-Site Scripting)
    - Motivation
    - XSS attacks
    - Defenses (and potential weaknesses)
  - CSRF (Cross-Site Request Forgery)
    - Cookies
    - Session
    - CSRF attacks
    - Defenses (and potential weaknesses)

Security risks on the internet

- Risk I:
  - Malicious websites should not be able to tamper with our information or interactions on other websites
  - Example:
    - We visit "latimes.com"
    - Malicious folks do "ad" on this site
    - The "ad" runs some JavaScripts and extracts our information from "latimes" (e.g., which type of articles we read)



CALIFORNIA

How much COVID is in my community? It's getting harder to tell

Bianca Povalitis, center, along with fellow roller skaters enjoy an evening at Venice Beach Skate Plaza in June 2022. (Jason Armond / Los Angeles Times)

BY LUKE MONEY, RONG-GONG LIN II

MAY 15, 2023 5 AM PT

SUBSCRIBERS ARE READING

TRAVEL & EXPERIENCES
FOR SUBSCRIBERS
14 things to do in Los Olivos, the magical country town filled with wine and lavender blooms

TELEVISION
Hulu documentary delves deeper into the Randall Emmett scandal

LAKERS
Plaschke: I was wrong: These Lakers can win an NBA championship

ADVERTISEMENT

FREE AZURE CLOUD READINESS ROADMAP

Determine the optimal path into hyperscale cloud.

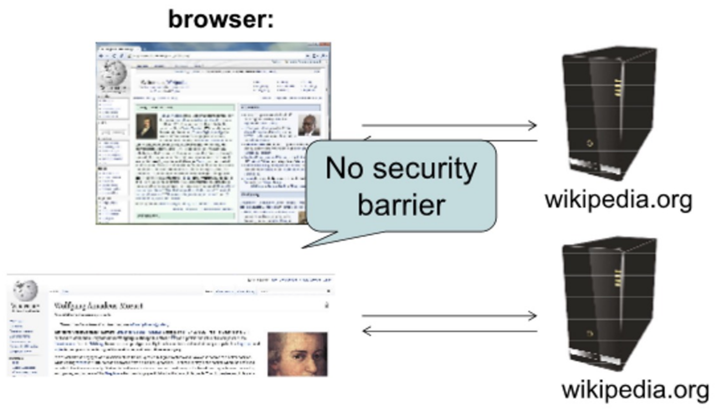
ET STARTED

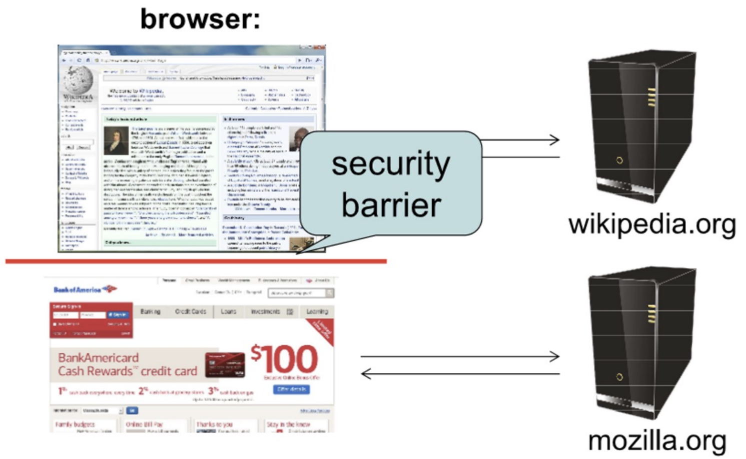
OneNeck

I want to know what you read!

Solution to the security risk

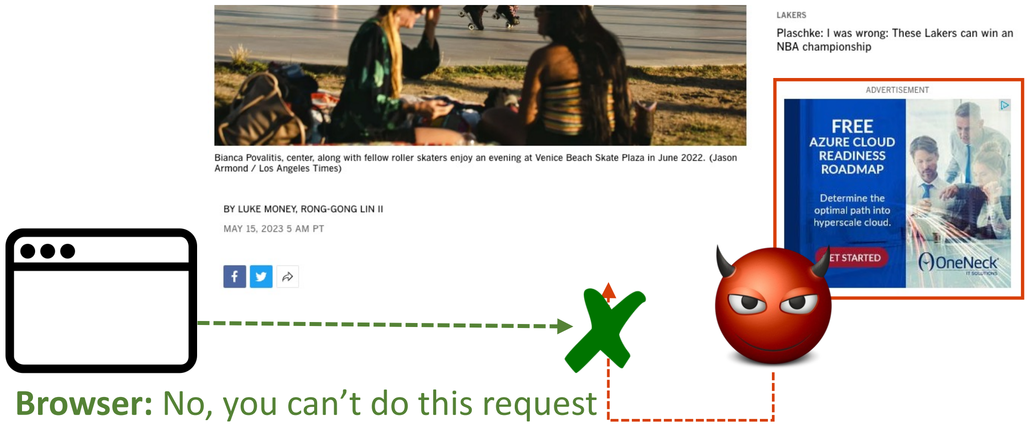
- Same-origin policy
  - A rule that prevents one website from tampering with other *unrelated* websites
    - Enforced by the web browser
    - Prevents a malicious website from running scripts on other websites
    - Pages from the same site don't need to be isolated to each other





Solution to the security risk

- Same-origin policy
  - A rule that prevents one website from tampering with other *unrelated* websites
    - Enforced by the web browser
    - Prevents a malicious website from running scripts on other websites
    - Pages from the same site don't need to be isolated to each other



Bianca Povalitis, center, along with fellow roller skaters enjoy an evening at Venice Beach Skate Plaza in June 2022. (Jason Armond / Los Angeles Times)

BY LUKE MONEY, RONG-GONG LIN II

MAY 15, 2023 5 AM PT

LAKERS

Plaschke: I was wrong: These Lakers can win an NBA championship

Browser: No, you can't do this request

Recap: URLs

- Same-origin policy
  - A rule that prevents one website from tampering with other *unrelated* websites
    - Enforced by the web browser
    - Prevents a malicious website from running scripts on other websites
    - Pages from the same site don't need to be isolated to each other
  - Every webpage has an origin defined by its URL with three parts:
    - Protocol: The protocol in the URL
    - Domain: The domain in the URL's location
    - Port: The port in the URL's location
      (If not specified, the default is 80 for HTTP and 443 for HTTPS)
    - Example:
      - https://computer.science.org/assets/photo.png (default: 443)
      - http://science.org:80/assets/new_photo.png

Solution to the security risk

- Same-origin policy
  - Two websites have the same origin *if and only if*
  - The protocol, domain, and port of the URL all match exactly

Domain I	Domain II	Same-origin?
https://cs.org	http://www.cs.org	No, domain mismatch
http://cs.org	https://cs.org	No, protocol mismatch
http://cs.org:80	http://cs.org:8080	No, protocol mismatch
https://cs.org/photo.png	https://cs.org/data/my.htm	Yes

Solution to the security risk

- Same-origin policy
  - Two websites have the same origin *if and only if*
  - The protocol, domain, and port of the URL all match exactly

Domain I	Domain II	Same-origin?
https://cs.org	http://www.cs.org	No, domain mismatch
http://cs.org	https://cs.org	No, protocol mismatch
http://cs.org:80	http://cs.org:8080	No, protocol mismatch
https://cs.org/photo.png	https://cs.org/data/my.htm	Yes

Reminder: Same-origin policy works with HTTPs!

Solution to the security risk

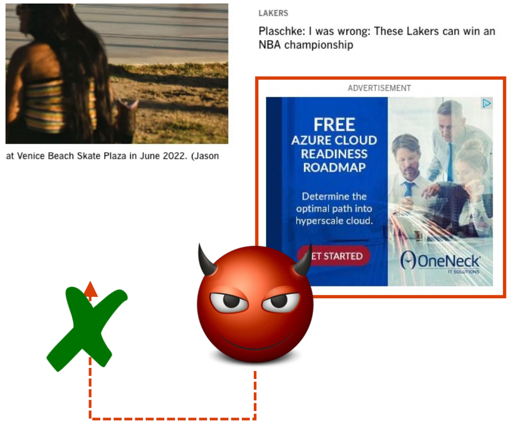
- Same-origin policy
  - Two websites have the same origin *if and only if*
  - The protocol, domain, and port of the URL all match exactly
  - Example scenario:
    - **cs.org** embeds **google.com**
    - The inner frame cannot interact with the outer frame
    - The outer frame cannot interact with the inner one

Same-origin policy exceptions

- Same-origin policy
  - Two websites have the same origin *if and only if*
  - The protocol, domain, and port of the URL all match exactly
  - Example scenario:
    - **cs.org** embeds **google.com**
    - The inner frame cannot interact with the outer frame
    - The outer frame cannot interact with the inner one
  - Exception I:
    - JavaScript runs with the origin of the page that loads it
    - ex. **cs.org** fetches JavaScript from **google.com**:
      - The JavaScript has the origin of **cs.org**
      - **cs.org** has "copy-pasted" JavaScript onto its webpage

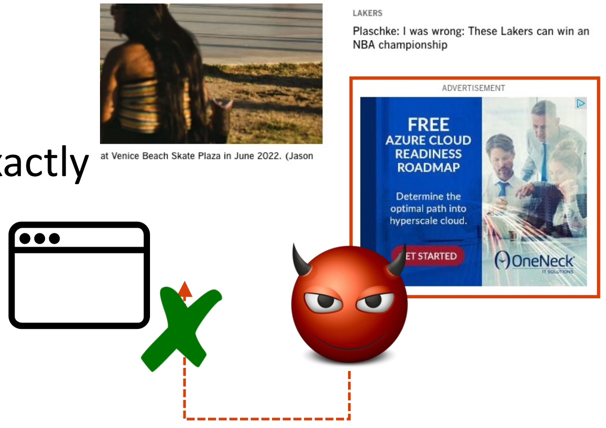
Same-origin policy exceptions

- Same-origin policy
  - Two websites have the same origin *if and only if*
  - The protocol, domain, and port of the URL all match exactly
  - Example scenario:
    - **cs.org** embeds **google.com**
    - The inner frame cannot interact with the outer frame
    - The outer frame cannot interact with the inner one
  - Exception II:
    - Websites can fetch and display images/frames from other origins
    - The website only knows about the image's size and dimensions (restricted info.)
    - The image and the frame has the origin of the page that it comes from (restricted)


at Venice Beach Skate Plaza in June 2022. (Jason
LAKERS
Plaschke: I was wrong: These Lakers can win an NBA championship
ADVERTISEMENT
FREE AZURE CLOUD READINESS ROADMAP
Determine the optimal path into hyperscale cloud.
ET STARTED
OneNeck

Same-origin policy exception (and a weakness)

- Same-origin policy
  - Two websites have the same origin *if and only if*
  - The protocol, domain, and port of the URL all match exactly
  - Example scenario:
    - **cs.org** embeds **google.com**
    - The inner frame cannot interact with the outer frame
    - The outer frame cannot interact with the inner one
  - Exception III:
    - Websites can agree to allow some limited sharing
    - Cross-origin resource sharing (CORS)
    - ex. the **postMessage** function in JavaScript
      - Receiving origin decides if to accept the message based on the origin
      - The correctness is enforced by the browser


LAKERS
Plaschke: I was wrong: These Lakers can win an NBA championship
at Venice Beach Skate Plaza in June 2022. (Jason
ADVERTISEMENT
FREE AZURE CLOUD READINESS ROADMAP
Determine the optimal path into hyperscale cloud.
ET STARTED
OneNeck

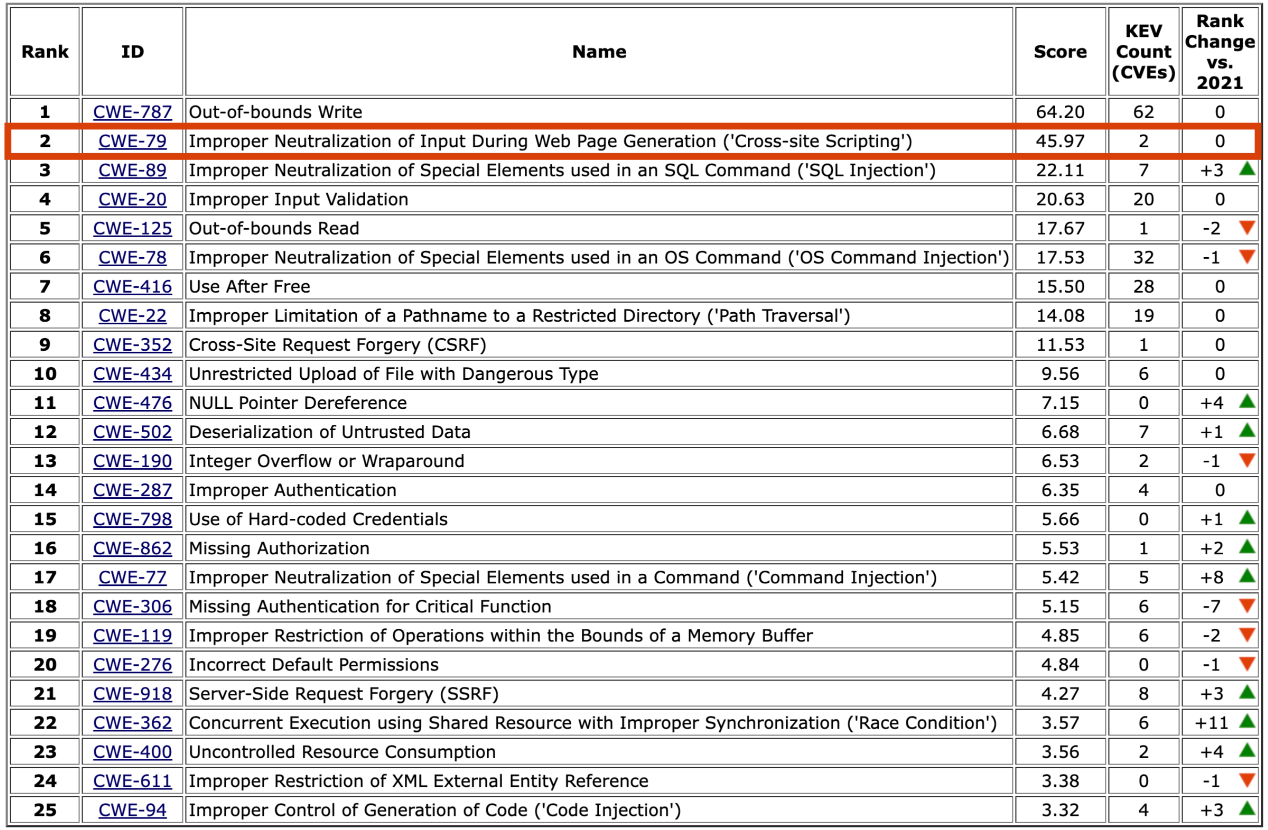
postMessage(“run”, script)

Topics for today

- Advanced web security
  - Same-origin policy
    - Motivation
    - Same-origin policy
    - Weaknesses
  - XSS (Cross-Site Scripting)
    - Motivation
    - XSS attacks
    - Defenses (and potential weaknesses)
  - CSRF (Cross-Site Request Forgery)
    - Cookies
    - Session
    - CSRF attacks
    - Defenses (and potential weaknesses)

Security risks on the Internet

- Risk II



Rank	ID	Name	Score	KEV Count (CVEs)	Rank Change vs. 2021
1	CWE-787	Out-of-bounds Write	64.20	62	0
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.97	2	0
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	22.11	7	+3 ▲
4	CWE-20	Improper Input Validation	20.63	20	0
5	CWE-125	Out-of-bounds Read	17.67	1	-2 ▼
6	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	17.53	32	-1 ▼
7	CWE-416	Use After Free	15.50	28	0
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.08	19	0
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.53	1	0
10	CWE-434	Unrestricted Upload of File with Dangerous Type	9.56	6	0
11	CWE-476	NULL Pointer Dereference	7.15	0	+4 ▲
12	CWE-502	Deserialization of Untrusted Data	6.68	7	+1 ▲
13	CWE-190	Integer Overflow or Wraparound	6.53	2	-1 ▼
14	CWE-287	Improper Authentication	6.35	4	0
15	CWE-798	Use of Hard-coded Credentials	5.66	0	+1 ▲
16	CWE-862	Missing Authorization	5.53	1	+2 ▲
17	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	5.42	5	+8 ▲
18	CWE-306	Missing Authentication for Critical Function	5.15	6	-7 ▼
19	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	4.85	6	-2 ▼
20	CWE-276	Incorrect Default Permissions	4.84	0	-1 ▼
21	CWE-918	Server-Side Request Forgery (SSRF)	4.27	8	+3 ▲
22	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	3.57	6	+11 ▲
23	CWE-400	Uncontrolled Resource Consumption	3.56	2	+4 ▲
24	CWE-611	Improper Restriction of XML External Entity Reference	3.38	0	-1 ▼
25	CWE-94	Improper Control of Generation of Code ('Code Injection')	3.32	4	+3 ▲

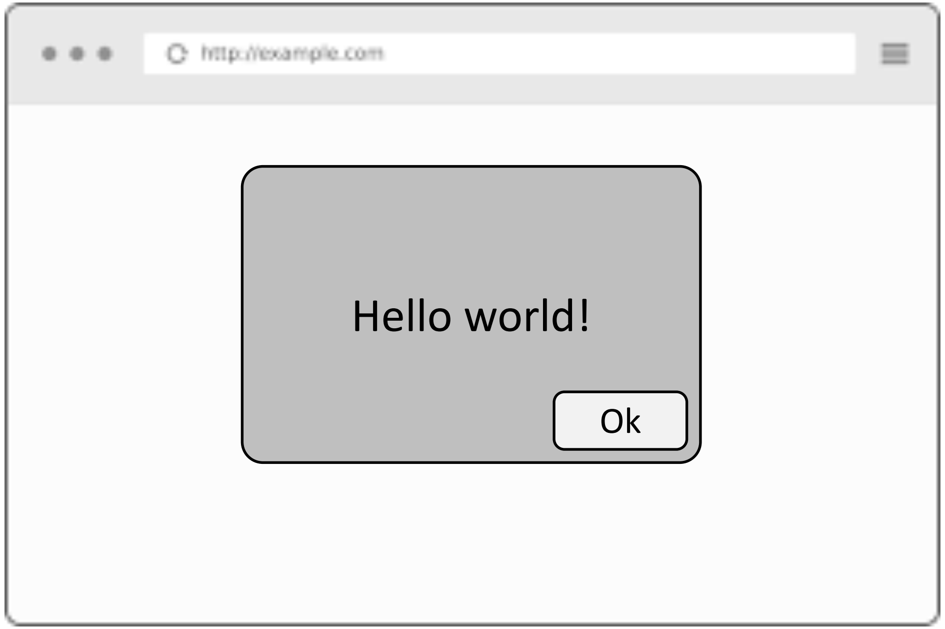
[1]https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

Revisit: JavaScript

- JavaScript
  - A programming language for running code in the web browser
  - Runs on the client-side
    - The server sends code as part of the HTTP response
    - The code runs in the browser, not in 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
  - Supported by all modern web browsers
    - Most modern webpages involve JavaScript

Revisit: JavaScript – cont'd

- JavaScript example
  - Create a pop-up message
  - HTML: <script>alert(”Hello world!")</script>



If the browser loads the HTML, it will run the embedded JavaScript and create a pop-up window.

Revisit: JavaScript – cont'd

- JavaScript in Go
  - Websites runs JavaScript with an (potentially malicious) input
  - Your HTML includes the following script (and hosted on "cs370.com")

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

  - You can use this script to render the website with the given name

```
https://cs370.com/hello?name=Bob
```

  - You will receive the following response (and the browser renders it)

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

Revisit: JavaScript – cont'd

- JavaScript in Go
  - Websites runs JavaScript with an (potentially malicious) input
  - Your HTML includes the following script (and hosted on "cs370.com")

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

  - You can use this script to include HTML tags

```
https://cs370.com/hello?name=<b>Bob</b>
```

  - You will receive the following response (and the browser renders it)

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

Revisit: JavaScript – cont'd

- JavaScript exploitation
  - Websites runs JavaScript with an (potentially malicious) input
  - Your HTML includes the following script (and hosted on "cs370.com")

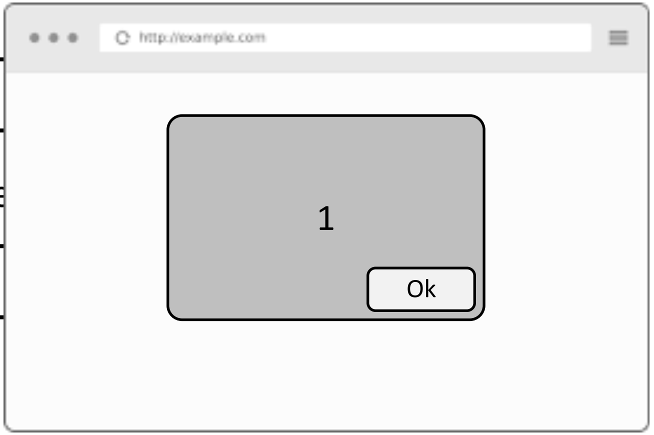
```
func handleSayHello(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Query()["name"][0]
    fmt.Fprintf(w, "<html><body>Hello %s!</body></html>", name)
}
```

  - You can use this script to include HTML tags

```
https://cs370.com/hello?name=<script>alert(1)</script>
```

  - You will receive the following response (and the browse

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



XSS: Cross-site scripting

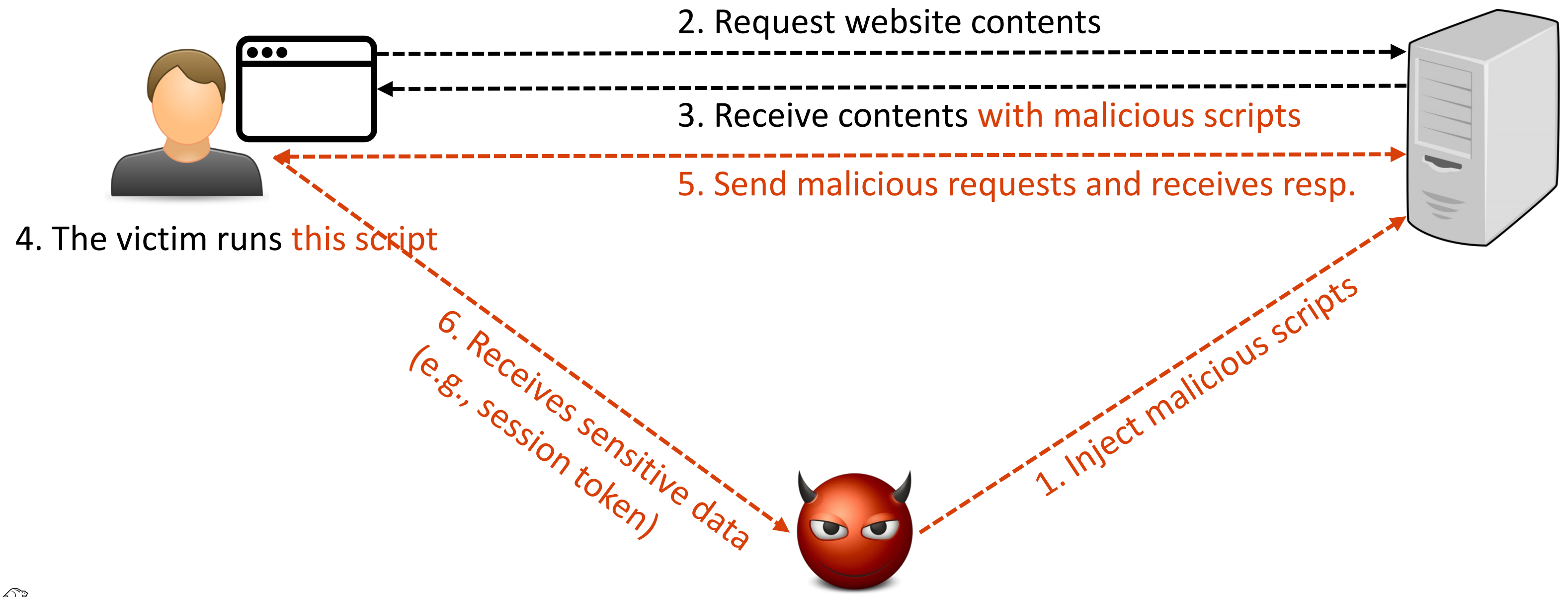
- Cross-site scripting
  - An adversary injects malicious JavaScript to a legitimate website
    - The victim accesses the legitimate website
    - The legitimate website sends the attacker's JavaScript to the victim
    - The victim's browser will run the script with the origin of the legitimate website
    - Now the attacker's JavaScript can access information on the legitimate website
  - It evades the same-origin policy
    - The JavaScript will run with the same origin (as the legitimate website)
  - Two representative XSS attacks
    - Stored XSS
    - Reflected XSS

XSS: Cross-site scripting

- Stored XSS (Persist XSS)
  - The attacker's JavaScript is stored on the legitimate server
  - Example: Facebook pages
    - Anyone can load a Facebook page with content provided by users
    - An adversary puts some JavaScript on their Facebook page
    - Anyone who loads the attacker's page will run JavaScript (with the origin of Facebook)
  - **Note:** stored XSS requires the victim to load the page with injected JavaScript

XSS: Cross-site scripting

- Stored XSS illustration
  - The attacker’s JavaScript is stored on the legitimate server
  - **Note:** stored XSS requires the victim to load the page with injected JavaScript

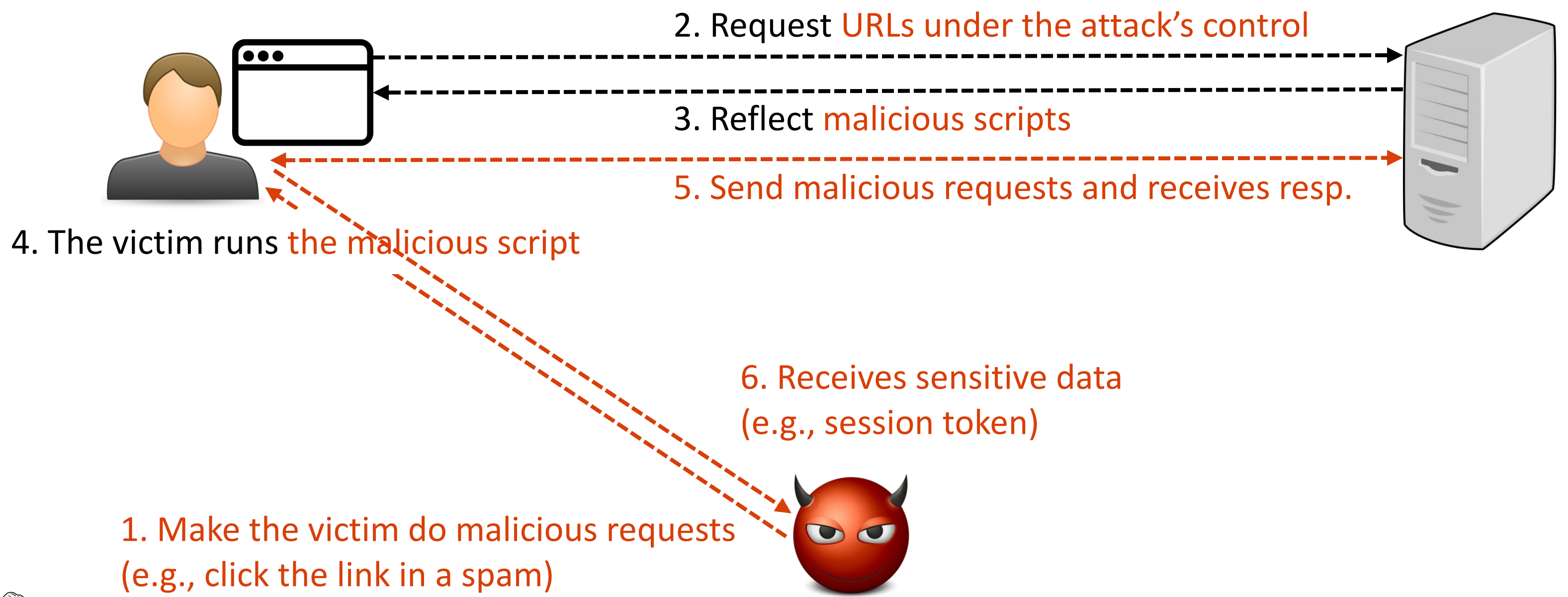

2. Request website contents
3. Receive contents with malicious scripts
5. Send malicious requests and receives resp.
4. The victim runs this script
6. Receives sensitive data (e.g., session token)
1. Inject malicious scripts

XSS: Cross-site scripting

- Reflected XSS
  - The attacker has the victim input JavaScript into a request
  - The content is reflected (copied) in the response from the server
  - Example: Search
    - The victim makes a request to http://google.com/search?q=Bob
    - The response will be “XYZ results for Bob”
    - The victim makes a request to http://google.com/search?q=<script>alert(1)</script>
    - The response will be “XYZ results for <script>alert(1)</script>”
  - **Note:** reflected XSS requires the victim to make a request with injected JavaScript

XSS: Cross-site scripting

- Reflected XSS illustration
  - The attacker has the victim input JavaScript into a request
  - The content is reflected (copied) in the response from the server


2. Request URLs under the attack's control
3. Reflect malicious scripts
5. Send malicious requests and receives resp.
4. The victim runs the malicious script
6. Receives sensitive data
(e.g., session token)
1. Make the victim do malicious requests
(e.g., click the link in a spam)

XSS: Cross-site scripting

- Reflected XSS (Practicality)
  - How do we make the victim to make such malicious requests?
    - Make a malicious website that includes an embedded iframe which makes the request
      - 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 the link
      - Posting a link on social media
      - Sending a text (Here is a new photo from your friend XYZ…)
      - Sending a phishing email
    - The link will load the attacker's website and redirects to the reflected XSS link
    - … (Good luck then) …

XSS: Cross-site scripting

- Defenses
  - HTML sanitization
  - HTML escaping
  - Content security policy (CSP)

XSS: Cross-site scripting

- Defenses
  - HTML sanitization
    - Remove the potentially-exploitable HTML code
      - Example:
        - » `<b> <i> <u> <em>` are allowed
        - » `<script> <object> <embed> <link>` will be removed
        - » `onClick` attribute will be removed
        - » … many more
      - Important:
        - » Need to escape all dangerous characters (lists of them can be found)
        - » Otherwise, we will still be vulnerable
    - You should always rely on trusted libraries to do this for you

XSS: Cross-site scripting

- Defenses
  - HTML escaping
    - Treat the request as data, not a script (not a HTML)
      - Transform certain characters into some other format
      - Example:
        - » Make it start with an ampersand (&) and end with a semicolon (;)
        - » Instead of <, use &lt;
        - » Instead of ", use &quot;
        - » Instead of , use &nbsp;
        - » … many more
      - Important:
        - » Need to escape all dangerous characters (lists of them can be found)
        - » Otherwise, we will still be vulnerable
    - You should always rely on trusted libraries to do this for you

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

XSS: Cross-site scripting

- Defenses
  - HTML escaping

```
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))
}
```

  - A malicious URL of the attack's

```
https://cs370.com/hello?name=<script>alert(1)</script>
```

  - You will receive the following safer response

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

XSS: Cross-site scripting

- Defenses
  - HTML escaping – cont'd
    - Tackles the challenge:
      - Suppose a developer has to take an action for every request
      - It's unlikely to be manageable by the developer
    - Smart ways
      - Templating
      - You declare in your HTML what data goes where
      - The templating engine handles all the escaping internally
      - The HTTP library encourages you to use templates

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

XSS: Cross-site scripting

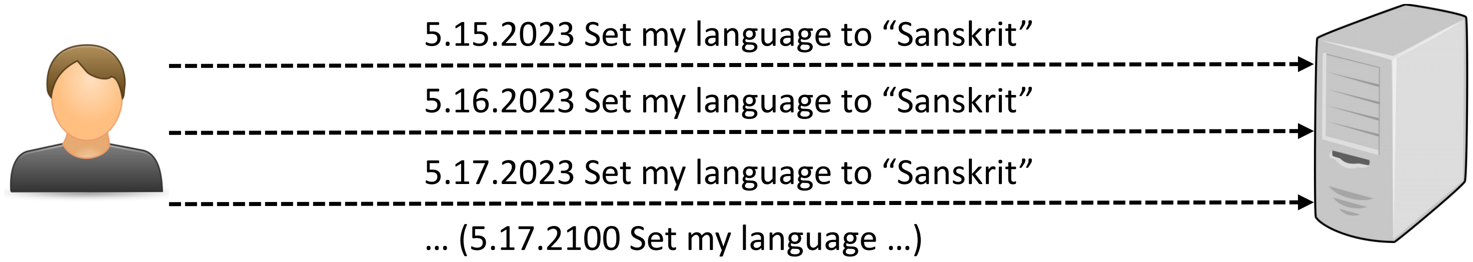
- Defenses
  - Content security policy (CSP)
    - Instruct the browser to only use resources loaded from specific places
    - Implement additional headers to specify such policy
    - Standard approaches:
      - Disallow all inline scripts, which prevents inline XSS
      - Ex: Disallow `<script>alert(1)</script>`
      - Only allow scripts from specified domains, which prevents XSS from linking to external scripts
      - Ex: Disallow `<script src="https://cs370.com/infostealer.js">`
    - Note:
      - CSP is also compatible with other contents (e.g., iframes, images, etc.)
      - Relies on the browser to enforce security, so more of a mitigation for defense-in-depth

Topics for today

- Advanced web security
  - Same-origin policy
    - Motivation
    - Same-origin policy
    - Weaknesses
  - XSS (Cross-Site Scripting)
    - Motivation
    - XSS attacks
    - Defenses (and potential weaknesses)
  - CSRF (Cross-Site Request Forgery)
    - Cookies
    - Session
    - CSRF attacks
    - Defenses (and potential weaknesses)

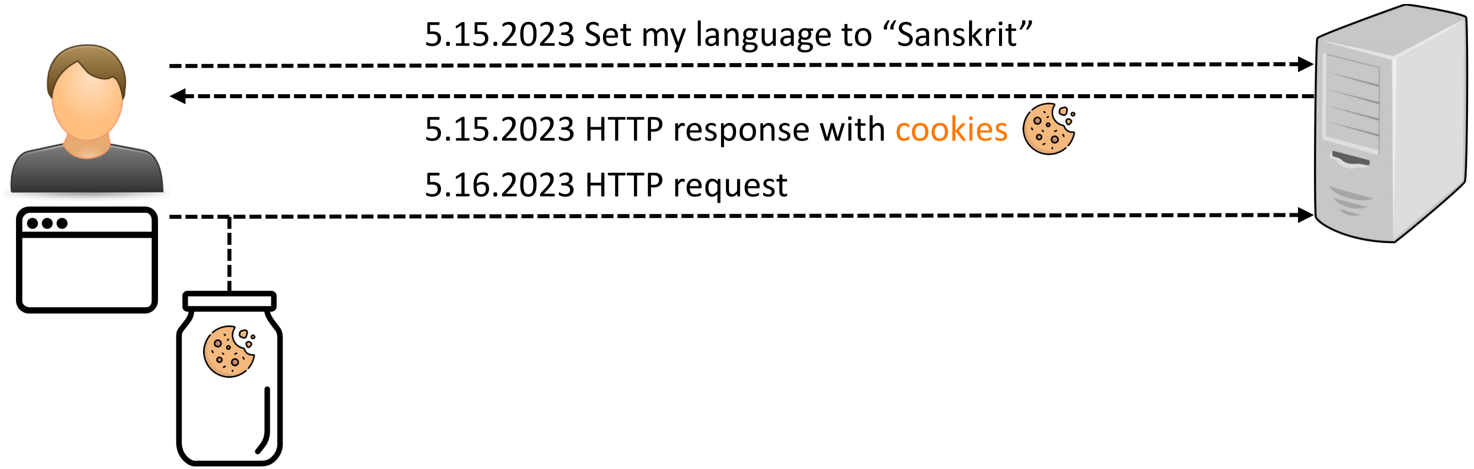
Motivation

- HTTP is state-"less"
  - Illustrating example
    - Yesterday, Bob visited "facebook.com" and set the language pref. to "Sanskrit"
    - Today, Bob visited "facebook.com" and found that the language is "English"
    - Bob sets it to "Sanskrit"
    - … (do this unlimited times)


5.15.2023 Set my language to "Sanskrit"
5.16.2023 Set my language to "Sanskrit"
5.17.2023 Set my language to "Sanskrit"
… (5.17.2100 Set my language …)

Motivation

- Solution: Cookies
  - Illustrating example
    - Yesterday, Bob visited “facebook.com” and set the language pref. to “Sanskrit”
    - The server sends HTTP response with small blocks of data containing the language pref.
    - The browser stores the data to its cookie jar
    - Today, Bob visited “facebook.com” and see the “Sanskrit” version


5.15.2023 Set my language to “Sanskrit”
5.15.2023 HTTP response with cookies
5.16.2023 HTTP request

Cookies

- Cookies
  - A small blocks of data
    - The server sends cookies as a part of their HTTP response (no cookies at the first time)
    - HTTP Header:
      - Set-cookie: name = value;
      - (It's a name-value pair with some extra metadata)
      - Example:

```
» HTTP/1.1 200 OK
» Content-Type: text/html
» Set-Cookie: items=16
» Set-Cookie: headercolor=blue
» Set-Cookie: footercolor=green
» Set-Cookie: screenmode=dark, Expires=Sun, 1 Jan 2023 12:00:00 GMT
```

    - Let's take a look (Chrome: view > developer > developer tools > application > Cookies)

Cookies

- Cookie scope
  - A small blocks of data
    - The server sends cookies as a part of their HTTP response (no cookies at the first time)
    - HTTP Header:
      - Set-cookie: name = value;
      - Domain = (when to send);
      - Path = (when to send);

    Scope
    - The server *automatically* attaches the cookies in scope

Cookies

- Cookie scope
  - A small blocks of data
    - The server sends cookies as a part of their HTTP response (no cookies at the first time)
    - HTTP Header:
      - Set-cookie: name = value;
      - Domain = (when to send);
      - Path = (when to send);
      - Secure = (only send over HTTPS);
    - The server *automatically* attaches the cookies in scope
    - The cookies can *only* be sent via secure communication (using TLS)

Cookies

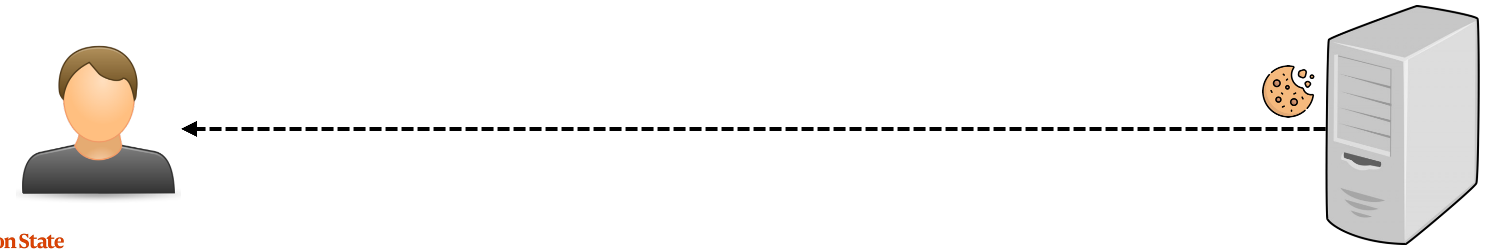
- Cookie scope
  - A small blocks of data
    - The server sends cookies as a part of their HTTP response (no cookies at the first time)
    - HTTP Header:
      - Set-cookie: name = value;
      - Domain = (when to send);
      - Path = (when to send);
      - Secure = (only send over HTTPS);
      - Expires = (when expires)
      - HttpOnly
    - The server *automatically* attaches the cookies in scope
    - The cookies can *only* be sent via secure communication (using TLS)
    - The browser *should delete* the cookies after a certain expiration date
    - HttpOnly: cookies cannot be accessed by JavaScript; only for HTTP requests

Cookies

- Cookie policy
  - The server sets the scope (domain and path) on cookies
  - The browser sends the cookies based on the scope

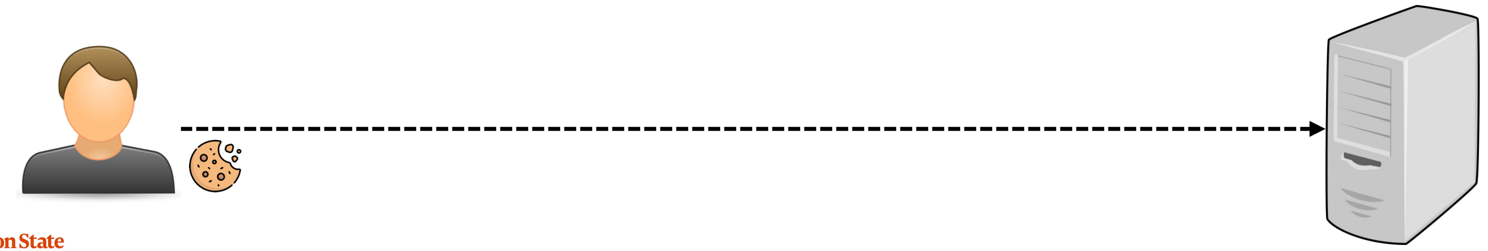
Cookies

- Cookie policy
  - The server sets the scope (domain and path) on cookies
    - Domain can be any domain-suffix of URL-hostname (not a TLD)
    - Example:
      - The server "login.cs370.com" sends cookies; can it
      - set cookies in the browser for "cs370.com"?
      - set cookies in the browser for ".cs370.com"?
      - set cookies in the browser for "secret.cs370.com"?
      - set cookies in the browser for ".com"?
      - set cookies in the browser for "osu-cs370.com"?
    - Path can be set to any path



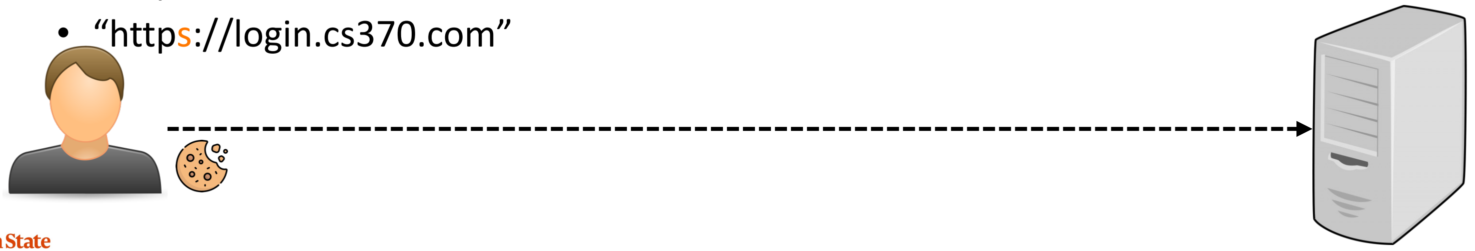
Cookies

- Cookie policy
  - The server sets the scope (domain and path) on cookies
  - The browser sends the cookies based on the scope
    - Suppose the cookie we have is
      - domain: “cs370.com”
      - path : “/micro-labs”
    - The browser can include the cookies in the request to:
      - http://login.cs370.com/micro-labs/week1/sanity-check



Cookies

- Example:
  - Cookie 1:
    - name = neuronoverflow
    - value = ctf-admin
    - domain = login.cs370.com
    - path = /
    - non-secure
  - Cookie 2:
    - name = test
    - value = ctf-player
    - domain =.cs370.com
    - path = /
    - non-secure
  - Which cookies will be sent?
    - "http://checkout.cs370.com"
    - "http://login.cs370.com"
    - "http://osu-cs370.com"
    - "https://login.cs370.com"

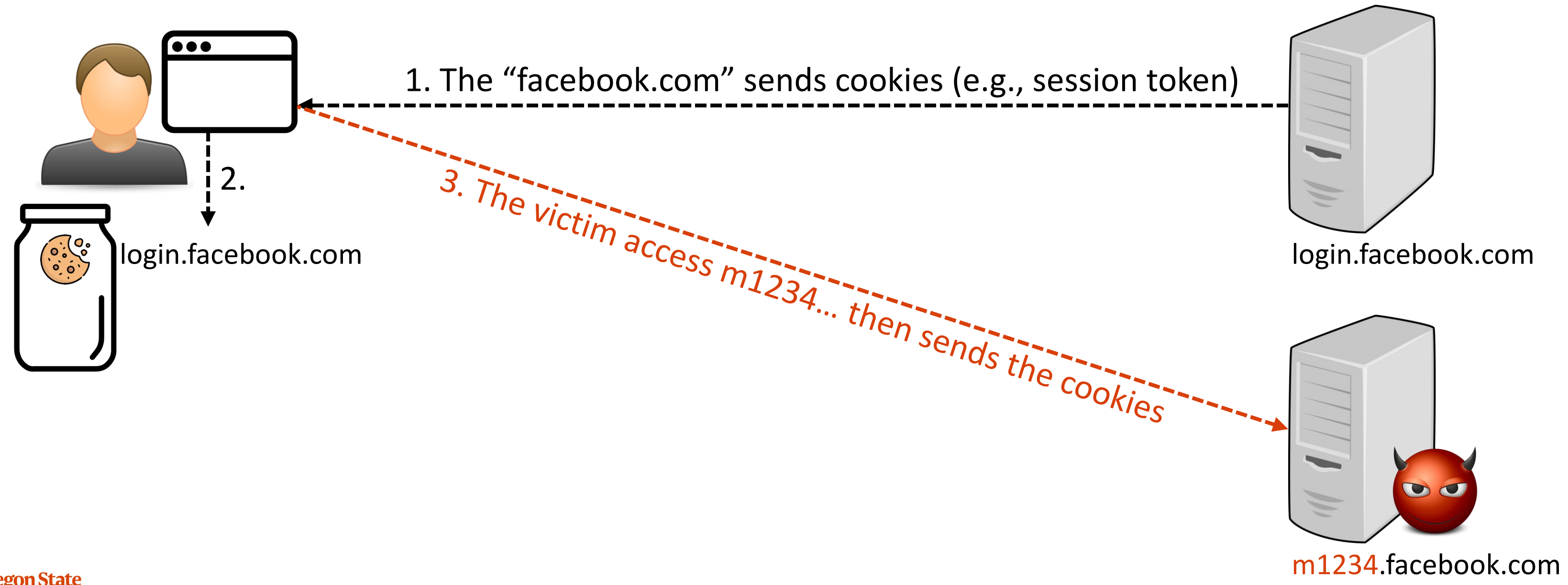


Cookies

- Cookies vs. same-origin policy
  - SOP requires an exact match between domains
  - Cookies do not always require an exact match; scope matters
  - Example:
    - Suppose we have a cookie:
      - name = neuronoverflow
      - value = ctf-admin
      - domain = login.cs370.com
      - path = /
      - non-secure
    - “http://users.cs370.com”
      - JavaScript on this URL can access the cookie above…

Cookies

- Bypass same-origin policy
  - SOP requires an exact match between domains


1. The "facebook.com" sends cookies (e.g., session token)
2.
login.facebook.com
3. The victim access m1234… then sends the cookies
login.facebook.com
m1234.facebook.com

Motivation

- Session authentication
  - Motivating example
    - Bob visited “oregonstate.com” and login with their username, password
    - Bob, 5-min later, visit “oregonstate.edu”
    - The website asks their usernamd and password
    - Bob is very happy…

MOTIVATION

- Session authentication
  - Motivating example
    - Bob visited “oregonstate.com” and login with their username, password
    - Bob, 5-min later, visit “oregonstate.edu”
    - The website asks their usernamd and password
    - Bob is very happy…
  - Session token
    - A secret value for associating requests with a legitimate user
    - In the first visit to the website:
      - Type the username and password
      - The browser receives a session token (the server remembers this token)
    - The subsequent visits to the website
      - Include the session token in the requests
      - The server checks if the token is valid and is not expired
      - Then the server processes the request

Session token

- Session authentication
  - Session token + cookies
    - A secret value for associating requests with a legitimate user
    - In the first visit to the website:
      - Type the username and password
      - The server sends cookies with a session token
      - The browser receives a session token (the server remembers this token)
    - The subsequent visits to the website
      - Include the session token cookie in the requests
      - The server checks if the token is valid and is not expired
      - Then the server processes the request
    - If one logs-out
      - The browser and server delete the session token

Session token

- + cookies
  - Security
    - Suppose that the session token is stolen:
      - The attacker can impersonate you in any request
      - … You are friendly-up!
    - To ensure the security
      - The server needs to generate session tokens *randomly* and *securely*
      - The browser requires to
        - » Check if malicious website cannot steal tokens (GSB)
        - » Make sure they do not send session tokens to malicious websites

Topics for today

- Advanced web security
  - Same-origin policy
    - Motivation
    - Same-origin policy
    - Weaknesses
  - XSS (Cross-Site Scripting)
    - Motivation
    - XSS attacks
    - Defenses (and potential weaknesses)
  - CSRF (Cross-Site Request Forgery)
    - Cookies
    - [Next lecture!] Session
    - [Next lecture!] CSRF attacks
    - [Next lecture!] Defenses (and potential weaknesses)

Thank You!

Tu/Th 4:00 – 5:50 PM

Sanghyun Hong

sanghyun.hong@oregonstate.edu

Oregon State University

SAIL

Secure AI Systems Lab