CS 578: CYBER-SECURITY

PART V: WEB SECURITY

Sanghyun Hong
sanghyun.hong@oregonstate.edu

SAIL
Secure AI Systems Lab

WHAT ARE THE ATTACKS IN THE WEB?

DICTIONARY ATTACK

- The security guarantee assumes
  - We choose the password randomly!

- In reality
  - (12345678) Easy to memorize and type
  - (OregonBeaverRocks) Some phrases familiar
  - (Oregon1234) Add numbers on the phrase
  - (password1234!!) Add special characters at the end
  - …

DICTIONARY ATTACK

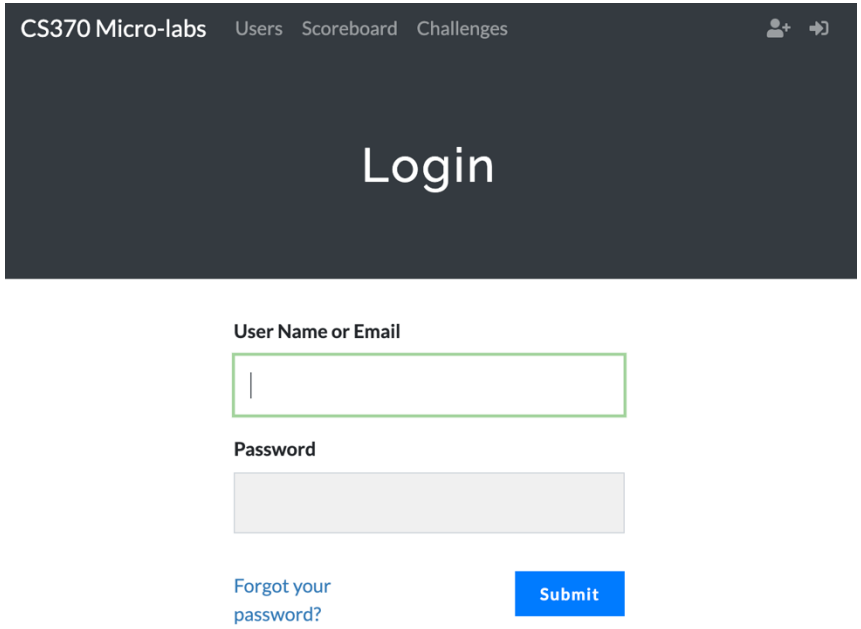
- Search space is significantly reduced
  - Suppose that the password is
    - 13 characters and consists of [A-Za-z0-9]
    - = 62^{13} possible combinations ($2.002854e^{23}$)
  - Suppose that
    - We know the password starts from ‘Portland’
    - = 62^{5} possible combinations ($9.1613283e^{8}$)
    - = 10^{15} smaller

SQL INJECTION

SQL INJECTION

- Exploit the system's weakness
  - SELECT (username, password) FROM users WHERE username = 'neuronoverflow' and password = SHA256(secret + "my-super-secure-password!@#$11")

- SQL injection
  - We supply 'or 'a'='a as a password
  - SELECT (username, password) FROM users WHERE username = 'neuronoverflow' and password = '' or 'a' ='a'
  - THIS IS ALWAYS TRUE!!!



SQL INJECTION

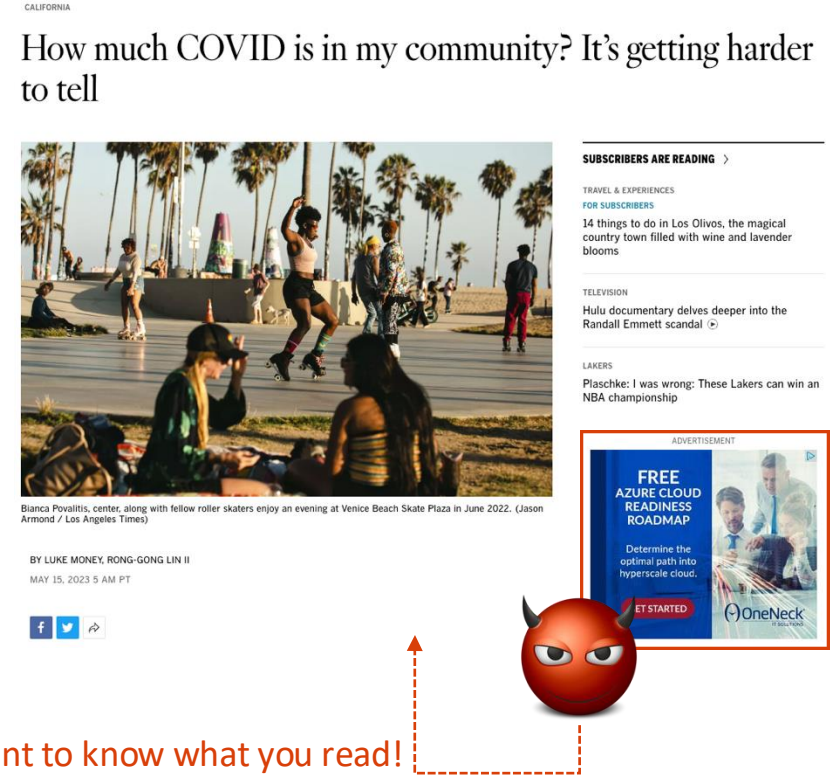
- What if we supply ‘ union select (‘admin’, ‘a’) where ‘a’=‘a as a password?
  - SELECT (username, password) FROM users WHERE
  - username = ‘neuronoverflow’ and password = ‘’ union select (‘admin’, ‘a’) where ‘a’=‘a’

- You will have the admin
  - None for the first select statement
  - and the 2nd statement will query
    - Username = ‘admin’
    - Password = ‘a’
    - Always return true ‘a’ = ‘a’

SAME ORIGIN POLICY

SECURITY RISKS ON THE INTERNET

- Risk I:
  - Malicious websites should not be able to tamper with our information or interact-ions 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)



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

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

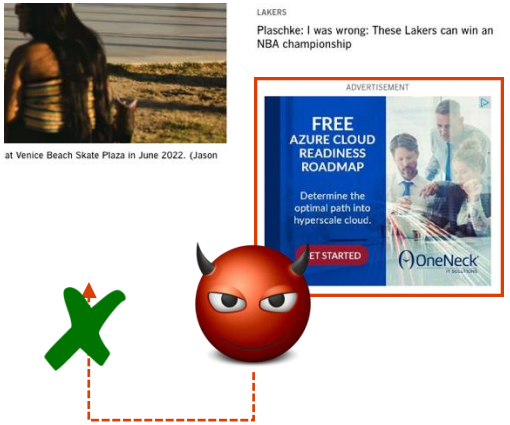
- Same-origin policy
  - Two websites have the same origin *if and only if*
  - The protocol, domain, and port of the URL all match exactly
  - Example scenario:
    - **cs.org** embeds **google.com**
    - The inner frame cannot interact with the outer frame
    - The outer frame cannot interact with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



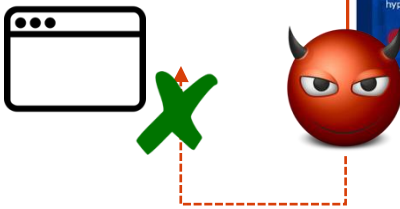
SAME-ORIGIN POLICY EXCEPTION (AND A WEAKNESS)

- 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

at Venice Beach Skate Plaza in June 2022. (Jason

LAKERS

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



postMessage("run", script)

CROSS-SITE SCRIPTING (XSS)

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 ▲

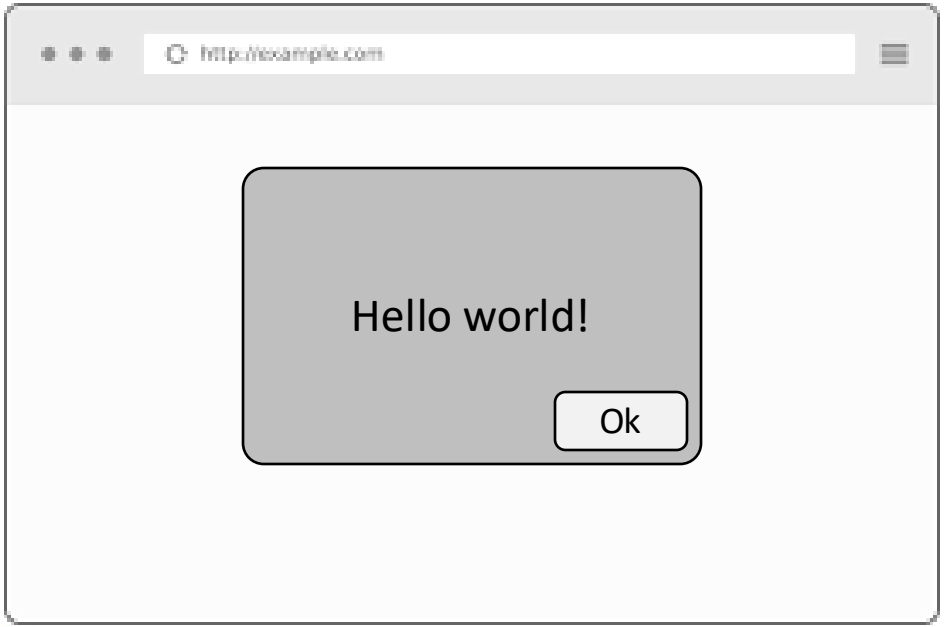
[1]https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

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")

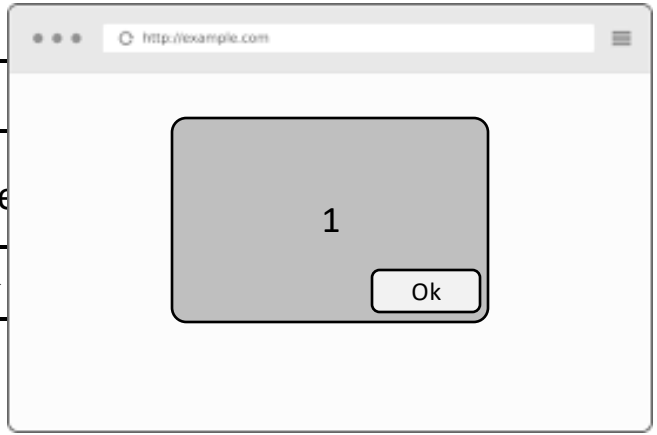
```
func handleSayHello(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Query()["name"][0]
    fmt.Fprintf(w, "<html><body>Hello %s!</body></html>", name)
}
```

  - 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

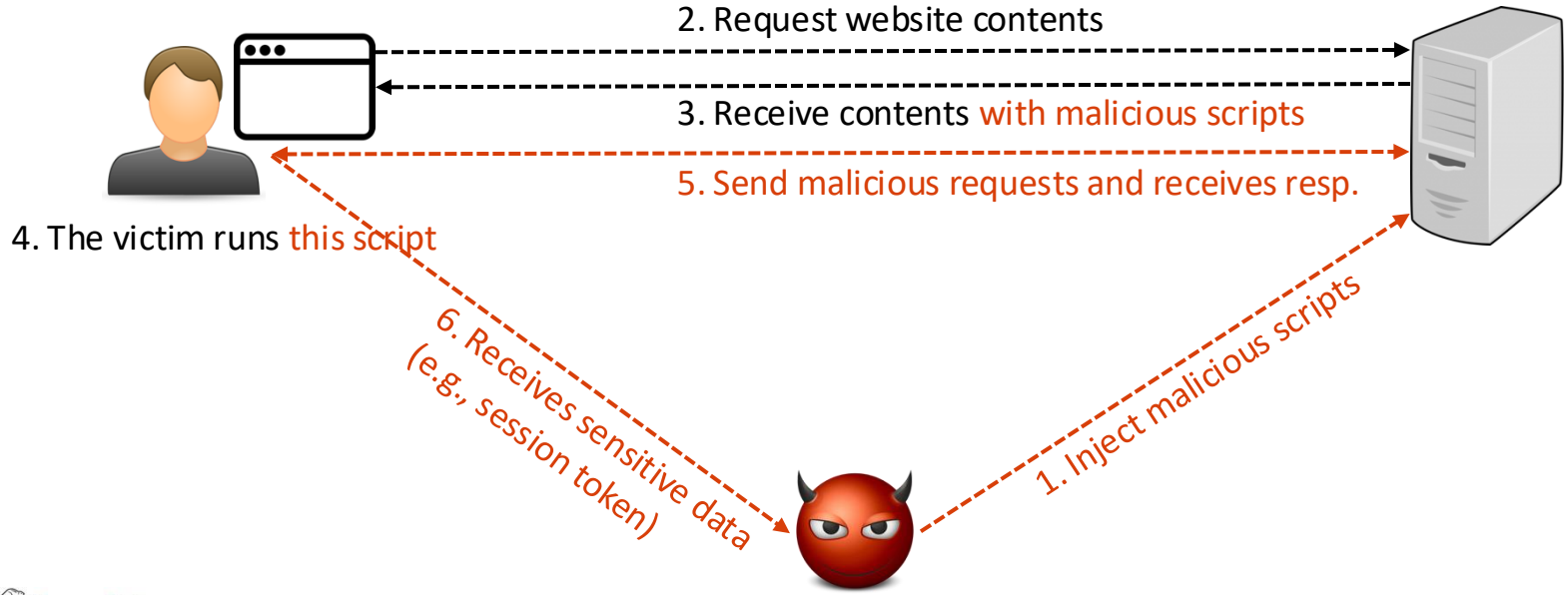
- Cross-site scripting
  - An adversary injects malicious JavaScript to a legitimate website
    - The victim accesses the legitimate website
    - The legitimate website sends the attacker's JavaScript to the victim
    - The victim's browser will run the script with the origin of the legitimate website
    - 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

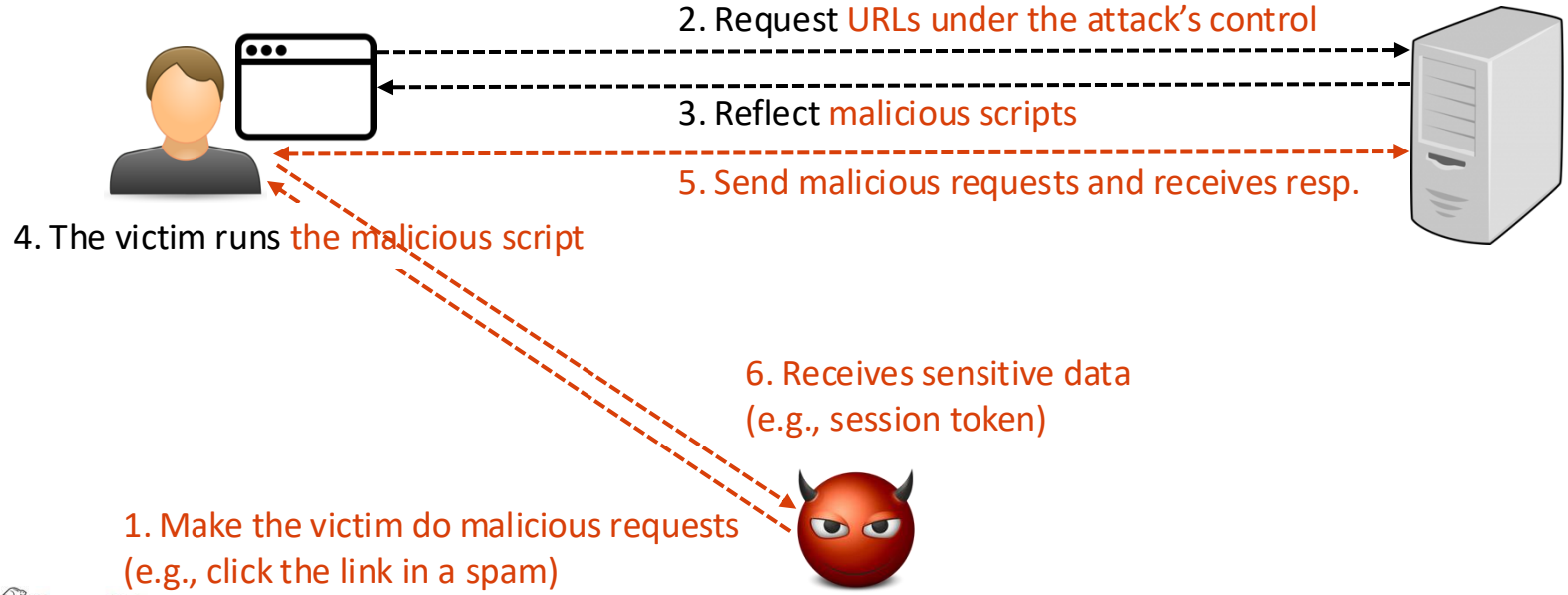


XSS: CROSS-SITE SCRIPTING

- 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



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) …

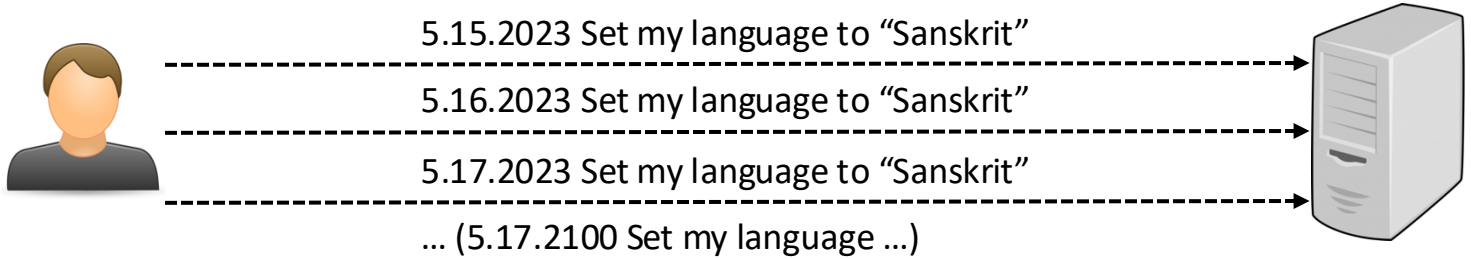
XSS: CROSS-SITE SCRIPTING

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

COOKIES

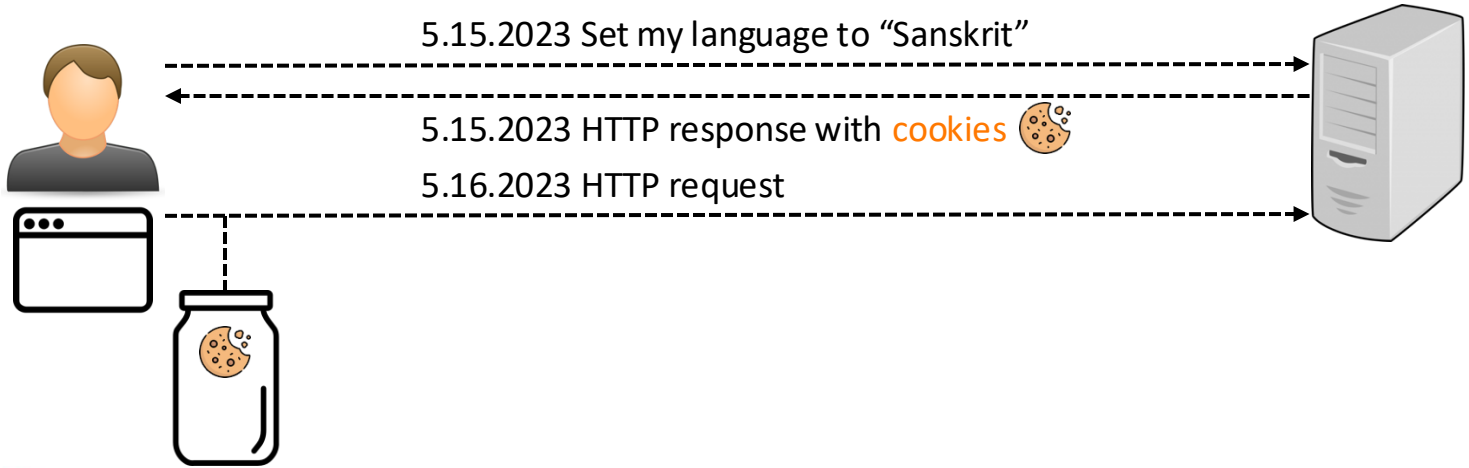
MOTIVATION

- 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)



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



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

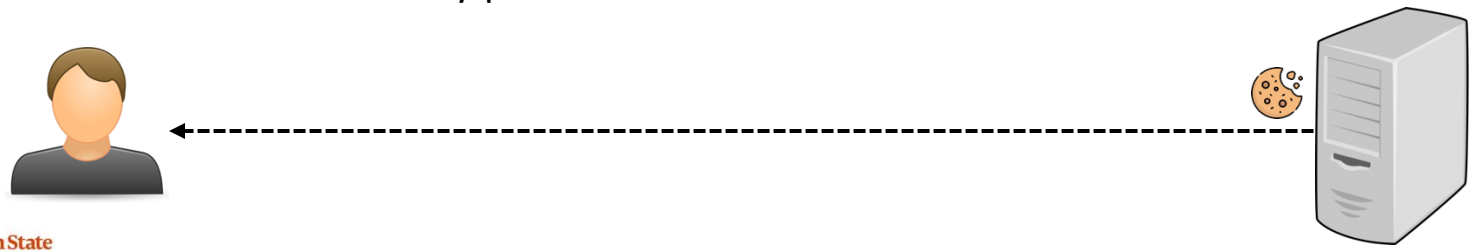
- Cookie scope
  - A small blocks of data
    - The server sends cookies as a part of their HTTP response (no cookies at the first time)
    - HTTP Header:
      - Set-cookie: name = value;
      - Domain = (when to send);
      - Path = (when to send);
      - Secure = (only send over HTTPS);
     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

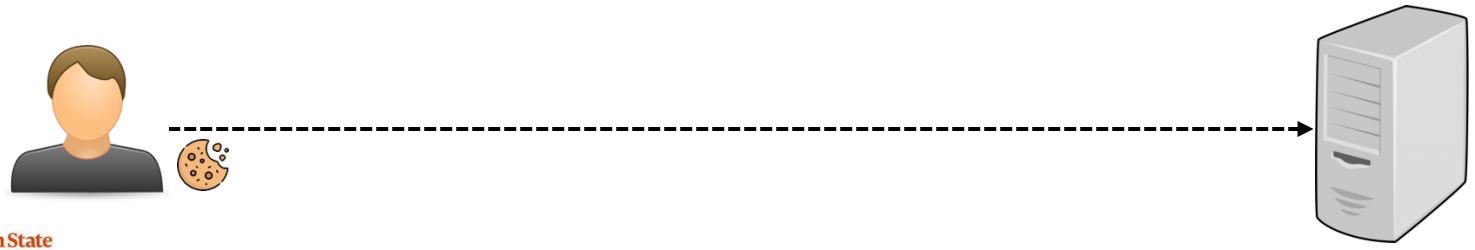
COOKIES

- 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



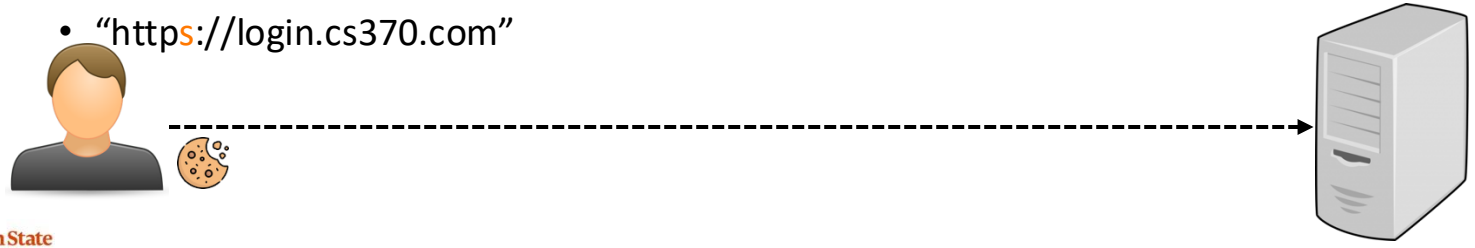
COOKIES

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

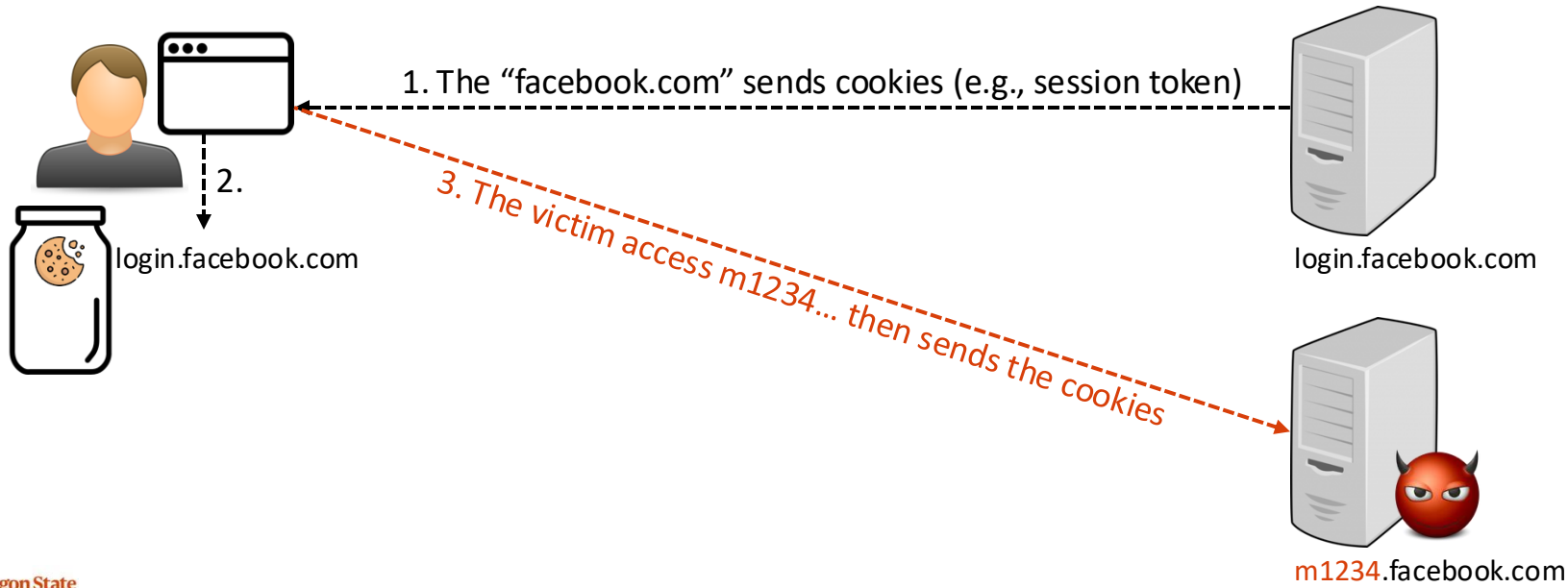


COOKIES

- 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



SESSION AUTHENTICATION

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

CROSS-SITE REQUEST FORGERY (CSRF)

SECURITY RISKS ON THE INTERNET

- Risk III

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 ▲

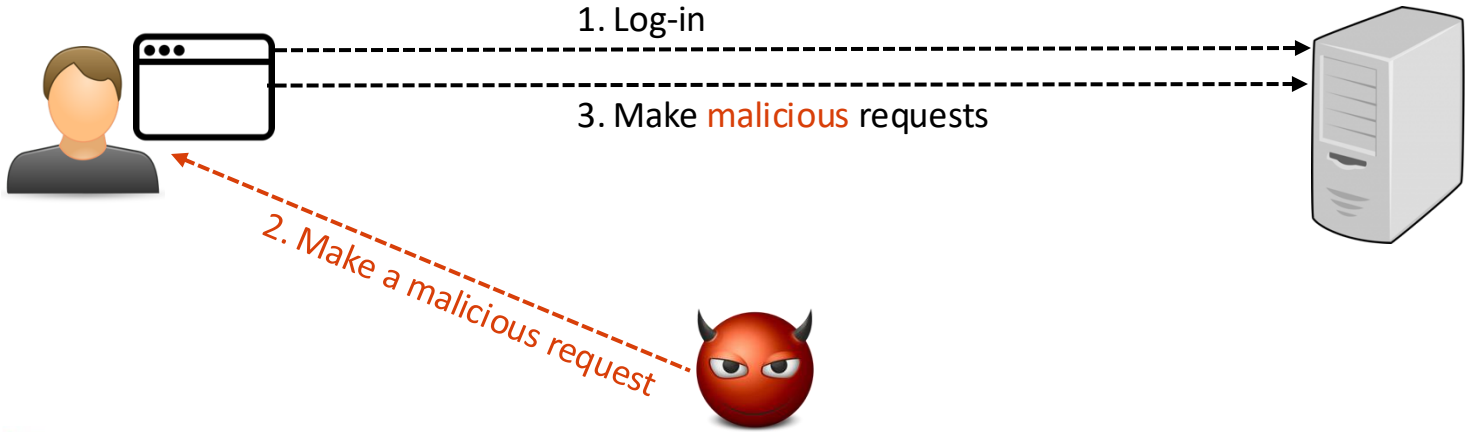
[1]https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

CSRF: CROSS-SITE REQUEST FORGERY

- CSRF (one-click attack or session riding)
  - Make legitimate users to send malicious requests to the server
    - The attacker impersonates a legitimate user
    - The user's browser will automatically attach (malicious) cookies (It exploits the cookie-based authentication mechanism)

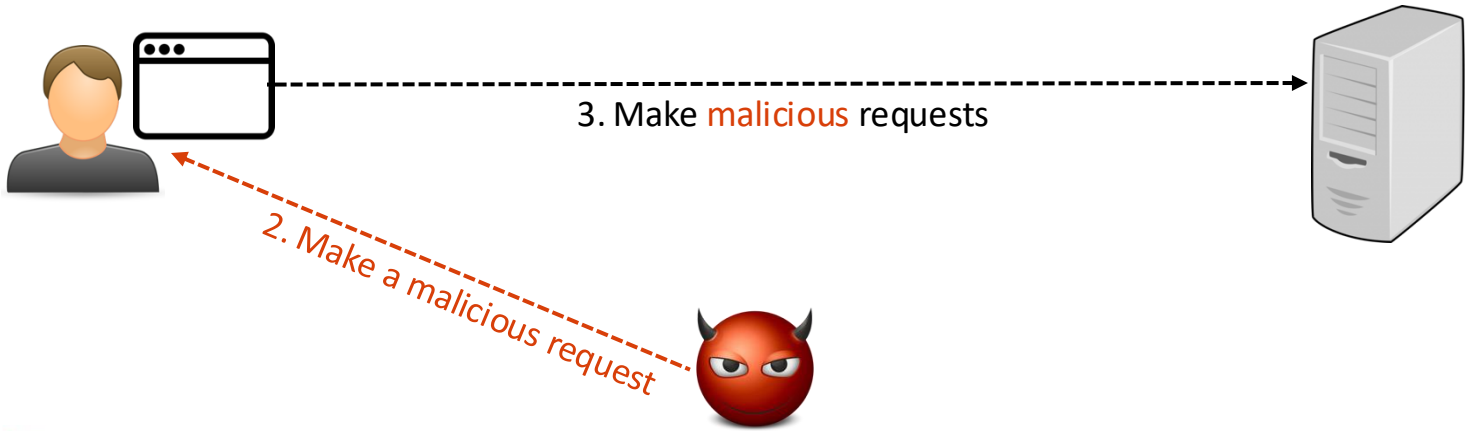
CSRF: CROSS-SITE REQUEST FORGERY

- CSRF (one-click attack or session riding)
  - Attack Illustration
    - A user authenticates to the server
    - The attacker tricks the user into making a malicious request
    - The server accepts the malicious request from the legitimate user
    - The server is the target!



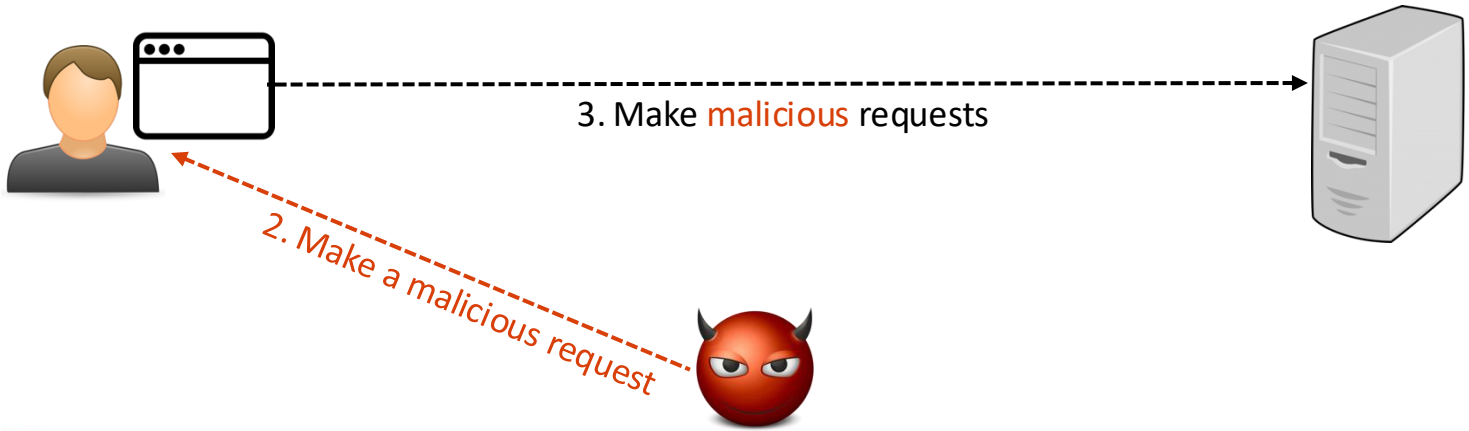
CSRF: CROSS-SITE REQUEST FORGERY

- CSRF (one-click attack or session riding)
  - How can an adversary trick the user?
    - GET request:
      - Make the user into clicking a link (SMS, Spam, …)
      - `https://bank.com/transfer?amount=10000&to=Mallony`
      - Put some html on a website the victim will visit (1x1 pixel image with a request)
      - `<img src="https://bank.com/transfer?amount=10000&to=Mallony">`



CSRF: CROSS-SITE REQUEST FORGERY

- CSRF (one-click attack or session riding)
  - How can an adversary trick the user?
    - Post request:
      - Make the user into clicking a link (run JavaScript on the website a user opens)
      - ex. The link opens an attacker's website, and it runs some JavaScript code
      - Put some JavaScript on a website the user will visit
      - ex. The attacker pays for an ad. and put JavaScript code there

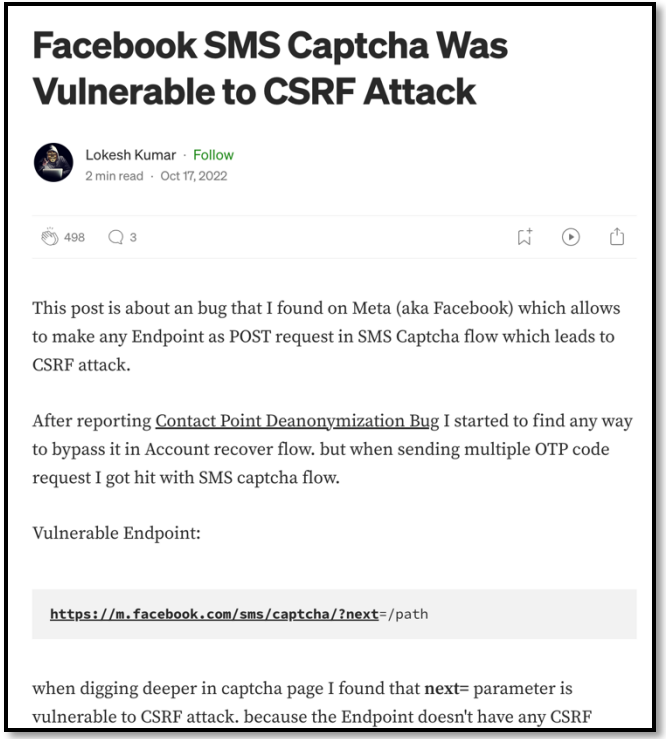


CSRF: CROSS-SITE REQUEST FORGERY

- CSRF != Reflected XSS
  - Reflected XSS: Make the user (victim) run malicious scripts
  - CSRF : Make the server run malicious scripts

CSRF: CROSS-SITE REQUEST FORGERY

- Real-world examples (Facebook, YouTube)



Facebook SMS Captcha Was Vulnerable to CSRF Attack

Lokesh Kumar · Follow
2 min read · Oct 17, 2022

498 3

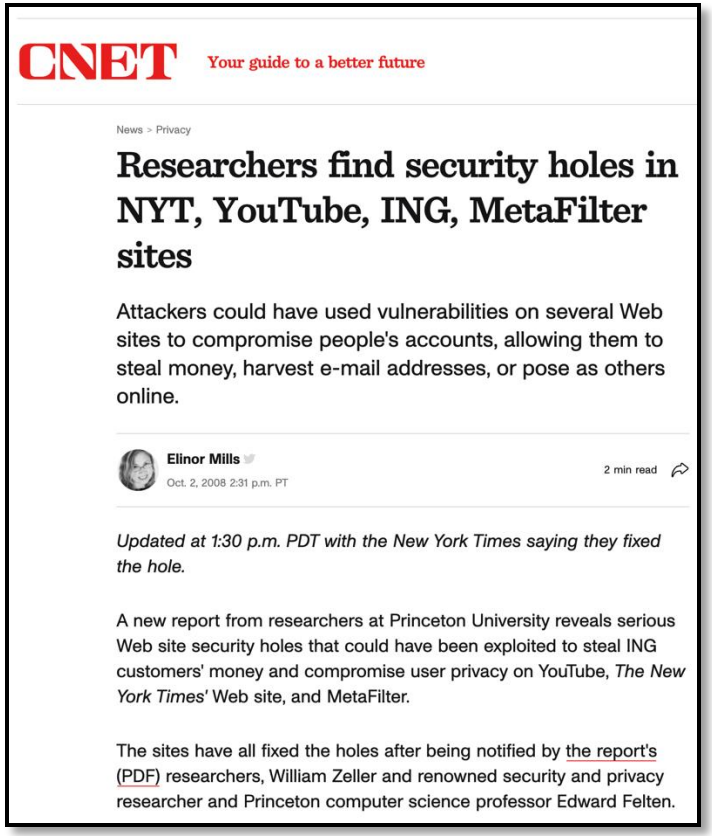
This post is about an bug that I found on Meta (aka Facebook) which allows to make any Endpoint as POST request in SMS Captcha flow which leads to CSRF attack.

After reporting Contact Point Deanonymization Bug I started to find any way to bypass it in Account recover flow. but when sending multiple OTP code request I got hit with SMS captcha flow.

Vulnerable Endpoint:

```
https://m.facebook.com/sms/captcha/?next=/path
```

when digging deeper in captcha page I found that **next=** parameter is vulnerable to CSRF attack. because the Endpoint doesn't have any CSRF



CNET Your guide to a better future

News > Privacy

Researchers find security holes in NYT, YouTube, ING, MetaFilter sites

Attackers could have used vulnerabilities on several Web sites to compromise people's accounts, allowing them to steal money, harvest e-mail addresses, or pose as others online.

Elinor Mills
Oct. 2, 2008 2:31 p.m. PT
2 min read

Updated at 1:30 p.m. PDT with the New York Times saying they fixed the hole.

A new report from researchers at Princeton University reveals serious Web site security holes that could have been exploited to steal ING customers' money and compromise user privacy on YouTube, *The New York Times*' Web site, and MetaFilter.

The sites have all fixed the holes after being notified by the report's (PDF) researchers, William Zeller and renowned security and privacy researcher and Princeton computer science professor Edward Felten.

CSRF: CROSS-SITE REQUEST FORGERY

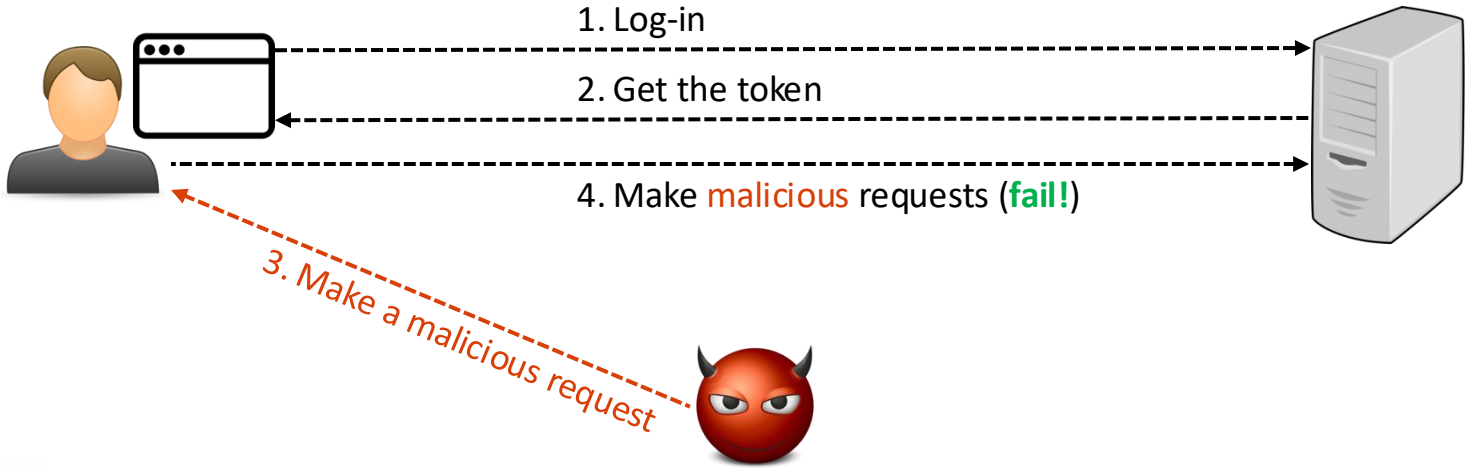
- Defenses
  - CSRF tokens
  - Referer validation
  - Same-site cookie attribute

CSRF: CROSS-SITE REQUEST FORGERY

- Defenses
  - CSRF tokens
    - A secret value that the server provides to the user
    - The user must include the same value in the request for the server
  - Note
    - The token should not be sent to the server in a cookie
    - The token must be sent somewhere else and stored to a separate storage
    - The token shouldn't be like a session token (it should expire after 1-2 requests)
  - Example:
    - HTML forms: vulnerable to CSRF (the attacker can do a POST request with their forms)
    - If a user requests from a form, the server attaches a CSRF token as *a hidden form* field
    - The attacker's JavaScript won't be able to create a valid form

CSRF: CROSS-SITE REQUEST FORGERY

- Defenses
  - CSRF tokens
    - A secret value that the server provides to the user
    - The user must include the same value in the request for the server



CSRF: CROSS-SITE REQUEST FORGERY

- Defenses
  - CSRF tokens
  - Referer header
    - A header in an HTTP request that shows which webpage made the request
    - In CSRF, the user makes malicious requests from a different website
      - "Referer" is a 30-year typo in the HTTP standard…
      - If we make a request from "facebook.com" then the header is "https://www.facebook.com"
      - If an "img" tag on a forum makes your browser to make a request then the Referer header will be "the forum's URL"
      - If JavaScript on an attacker's website makes your browswer to make a request then the header will be "the attacker's website URL"
    - The server checks the Referer header
      - Reject if it's not from the same-site
      - Accept if it's from the the same-site

CSRF: CROSS-SITE REQUEST FORGERY

- Defenses
  - CSRF tokens
  - Referer header
    - A header in an HTTP request that shows which webpage made the request
    - Potential issues:
      - The server can “observe” the user’s private info. from the header (ex. “facebook.com/<your-friend-name>/posting_1234”)
      - Oftentimes, network firewalls (or your browswers) remove this header…
      - The header is optional; some requests can come without the header (what should we do…)

CSRF: CROSS-SITE REQUEST FORGERY

- Defenses
  - CSRF tokens
  - Referer header
  - Same-site cookies
    - Set a flag on a cookie unexploitable by CSRF attacks
    - The browser will send requests when the domain of the cookie = that of the origin
      - SameSite = none
      - SameSite = strict: check if the domain matches
    - Potential issue: not all browsers implements this attribute

UI ATTACKS

OVERVIEW

- UI attacks
  - What is it?
    - The attacker tricks the victim into thinking
    - They are taking an intended action when they are actually taking a malicious action
  - What to exploit?
    - User interfaces: the trusted path between the user and the computer
    - Your browser blocks the website to interact across different origins (SOP)
    - But trusts the user to do whatever they want
  - Two representative attacks
    - Clickjacking: Trick the victim into clicking on something from the attacker
    - Phishing: Trick the victim into sending the attacker personal information

CLICKJACKING

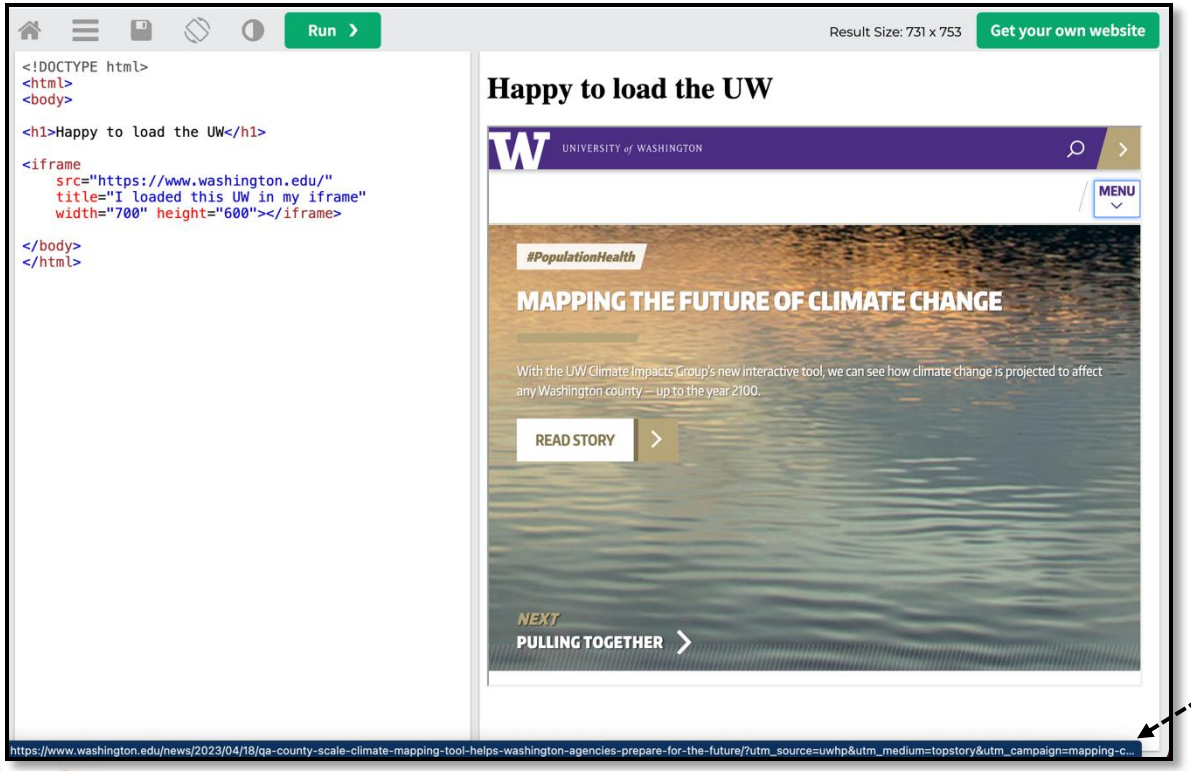
- Clickjacking
  - What is it?
    - Trick the victim into clicking on something from the attacker
  - What to exploit?
    - User interfaces: the trusted path between the user and the computer
    - Your browser trusts “your clicks”
    - If you click something, the browser believes you intend to click that
  - What can the attacker do?
    - Download a malicious program
    - Like a YouTube video(s), Instagram pages, or Amazon products
    - Steal keystrokes (once sth is downloaded)
      - Good luck to your credit card numbers, passwords, or any personal info.

CLICKJACKING EXAMPLE

- Download buttons
  - What is the *right* button?
  - What happens if I click the *wrong* button(s)?

CLICKJACKING EXAMPLE

- "iframe" can be vulnerable

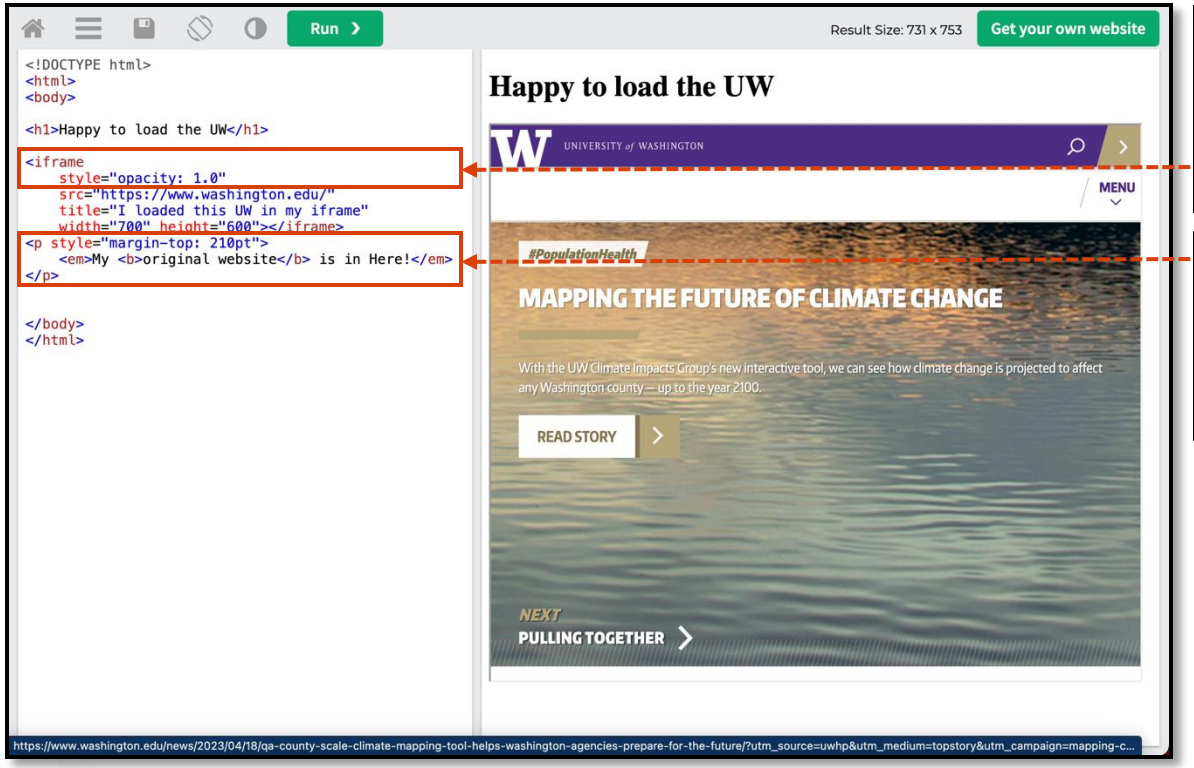


Note: any links on the website in the iframe are "washington.edu"

Users can click it, but we cannot make the website automatically click this link due to the same origin policy

CLICKJACKING EXAMPLE

- "iframe" can be vulnerable – let's change the code a bit

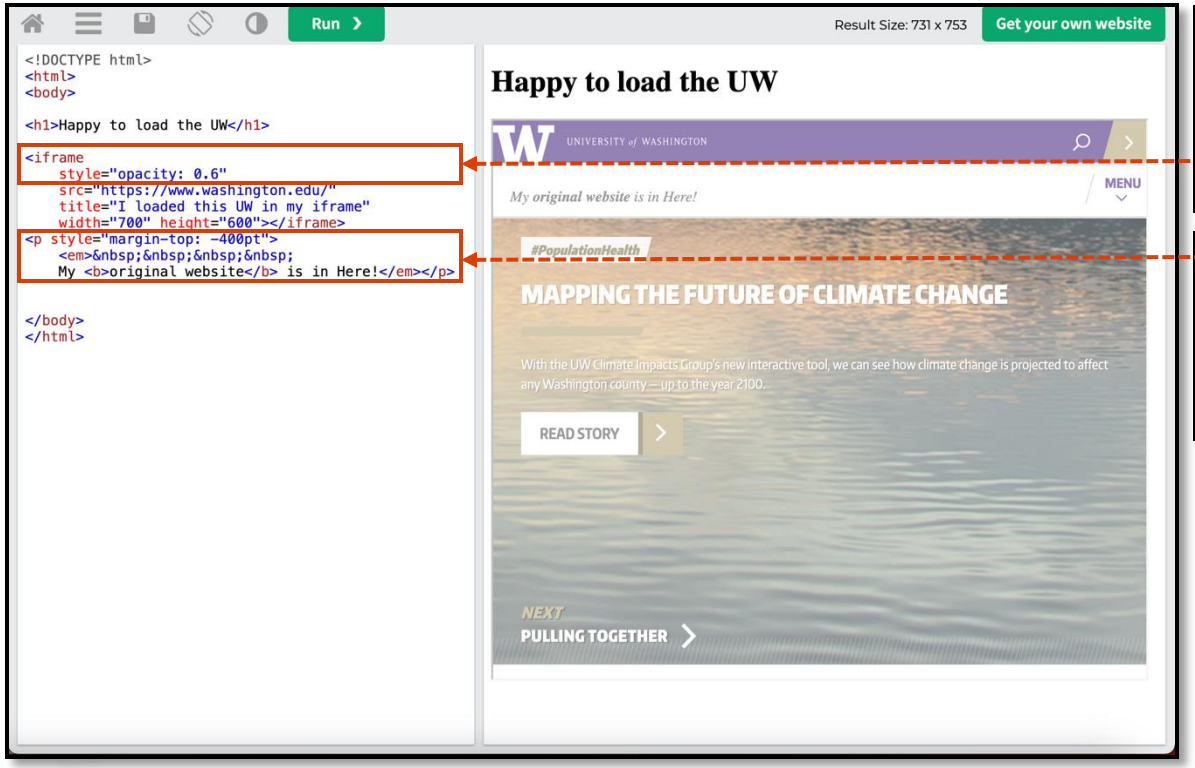


Put style: opacity to control the "opacity"

There's a text behind the iframe loaded the "washington.edu"

CLICKJACKING EXAMPLE

- "iframe" can be vulnerable – let's add some opacity

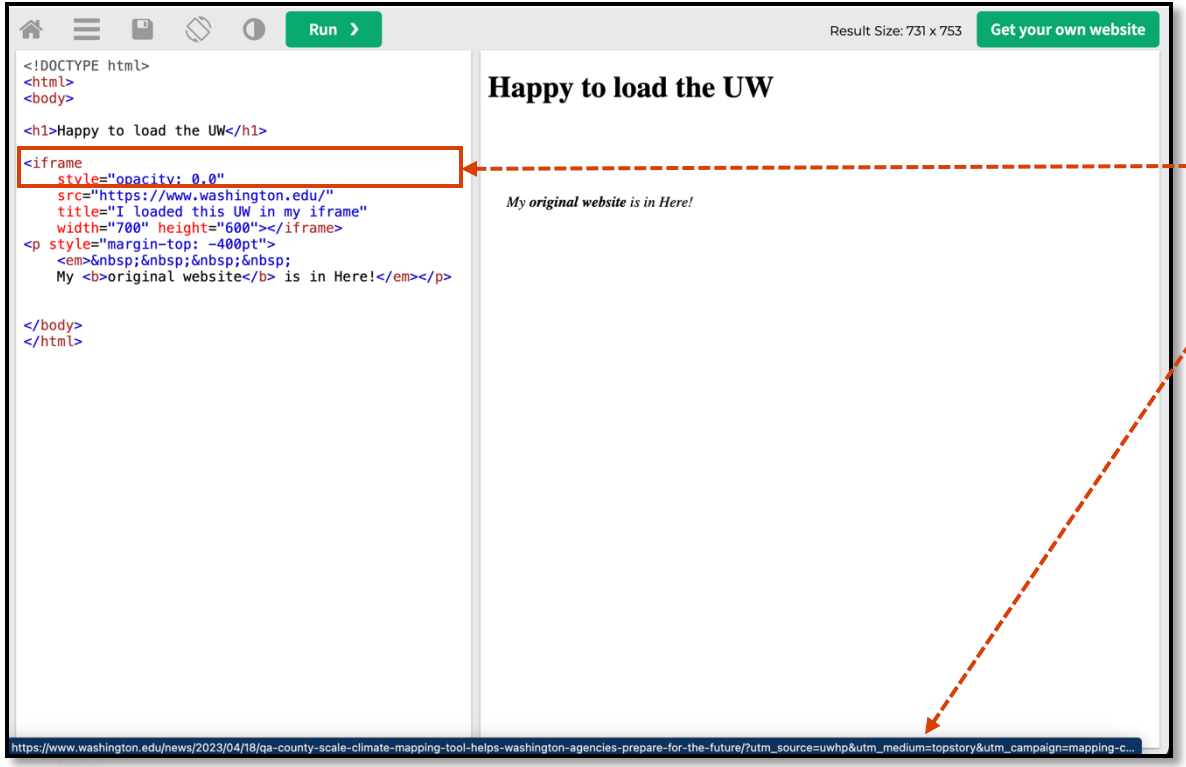


Put style: opacity to control the "opacity"

There's a text behind the iframe loaded the "washington.edu"

CLICKJACKING EXAMPLE

- "iframe" can be vulnerable – some more (or do extremely)

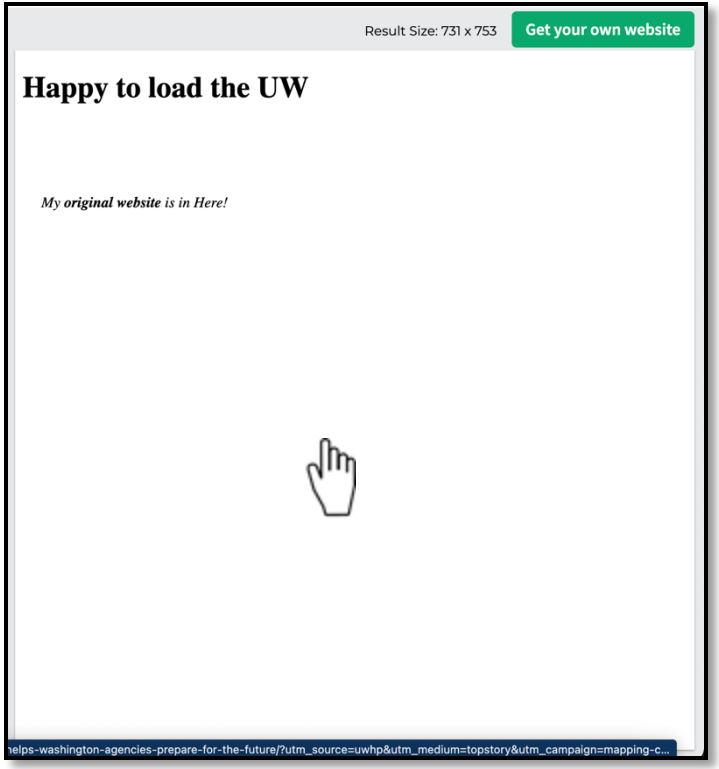


Now the website is completely opaque

But you can still click something on this website … ?!

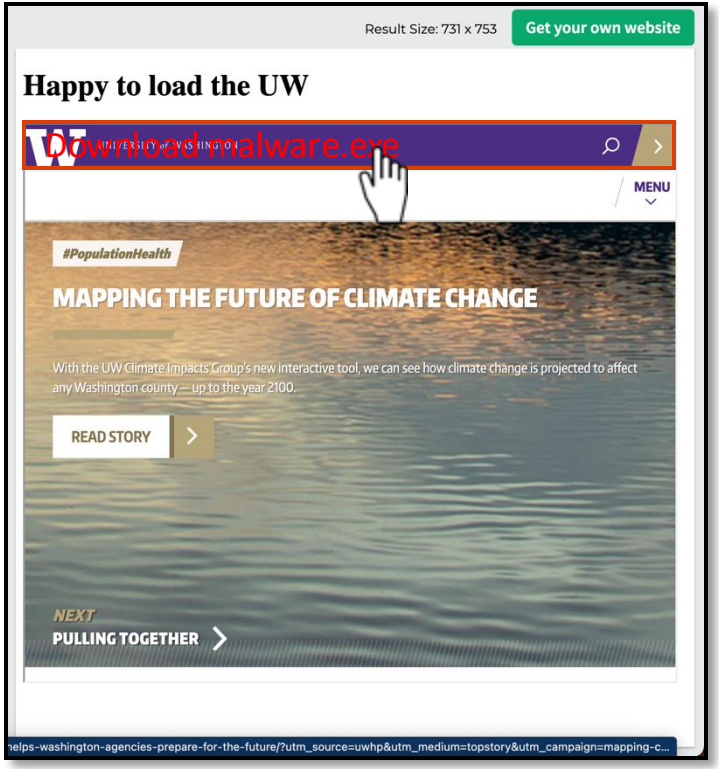
CLICKJACKING EXAMPLE

- Invisible “iframe”s
  - The attacker puts an iframe onto the attacker’s site invisibly, over visible, enticing content
  - Users (victims) think they click on the attacker’s website
  - But the click is actually happened on the legitimate website
  - ex. You click sth, but it’s the Facebook like btn



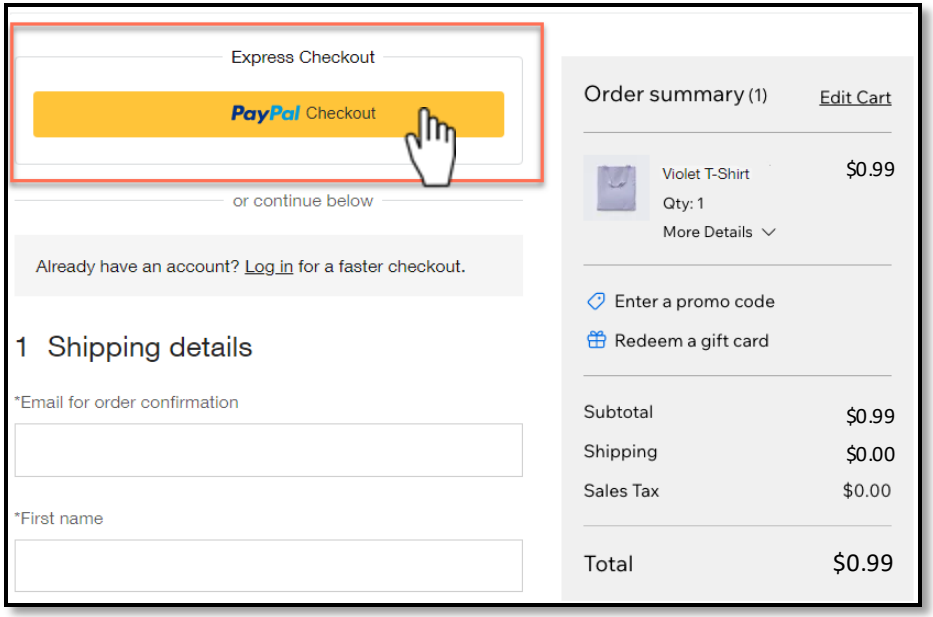
CLICKJACKING EXAMPLE

- Invisible “iframe”s – cont’d
  - The attacker puts an iframe onto the legitimate site invisibly, under invisible, malicious content
  - Users (victims) think they click on the legitimate website
  - But the click is actually happened on the attacker’s website
  - ex. You click sth, and it downloads malware



CLICKJACKING EXAMPLE

- Invisible “iframe”s – cont’d



  - The attacker frames the legitimate site, with the visible malicious contents
  - ex. You click the checkout, and I wish you the best!

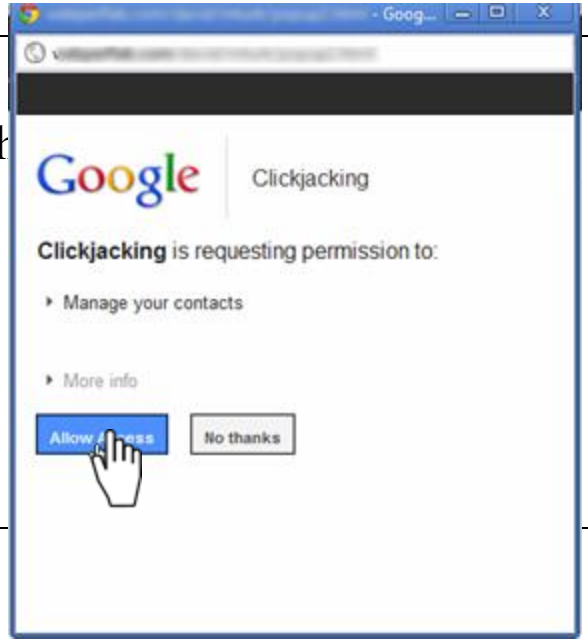
CLICKJACKING EXAMPLE

- Temporal attack
  - Process
    - The attacker uses JavaScript
    - that detects the position of your cursor
    - and change the website right before you click on sth.

CLICKJACKING EXAMPLE

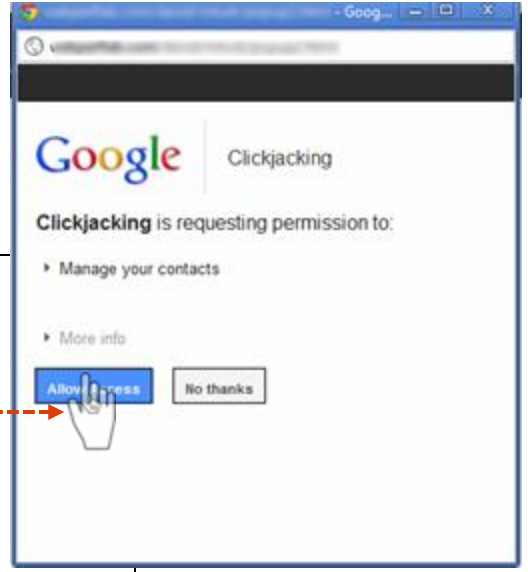
- Temporal attack
  - Example:

Instructions:
Please double-click on th your content



CLICKJACKING EXAMPLE

- Cursorjacking
  - CSS can style the appearance of your cursor
  - JavaScript can track a cursor's position
  - We can create a fake cursor to trick users into clicking on sth.



Instructions:
Please double-click on the [illegible] to content

Click here

Real cursor: created by JavaScript or with CSS

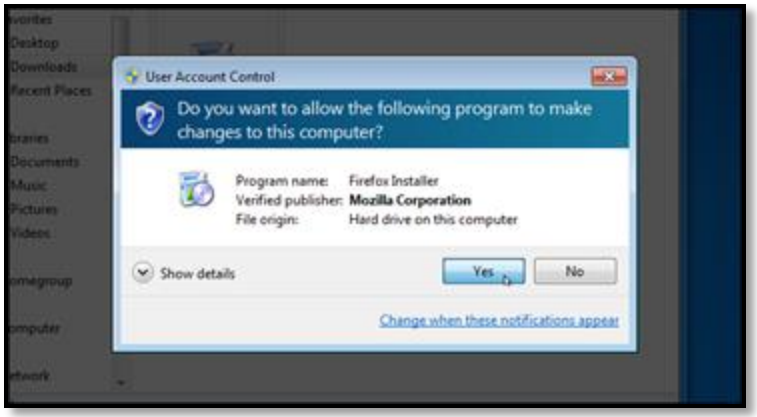
Fake cursor: created by JavaScript or with CSS

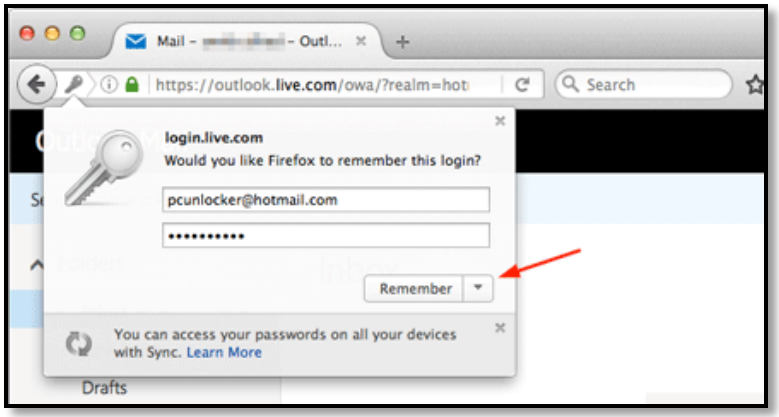
CLICKJACKING EXAMPLE

- Cursorjacking
  - CSS can style the appearance of your cursor
  - JavaScript can track a cursor's position
  - We can create a fake cursor to trick users into clicking on sth.

CLICKJACKING DEFENSES

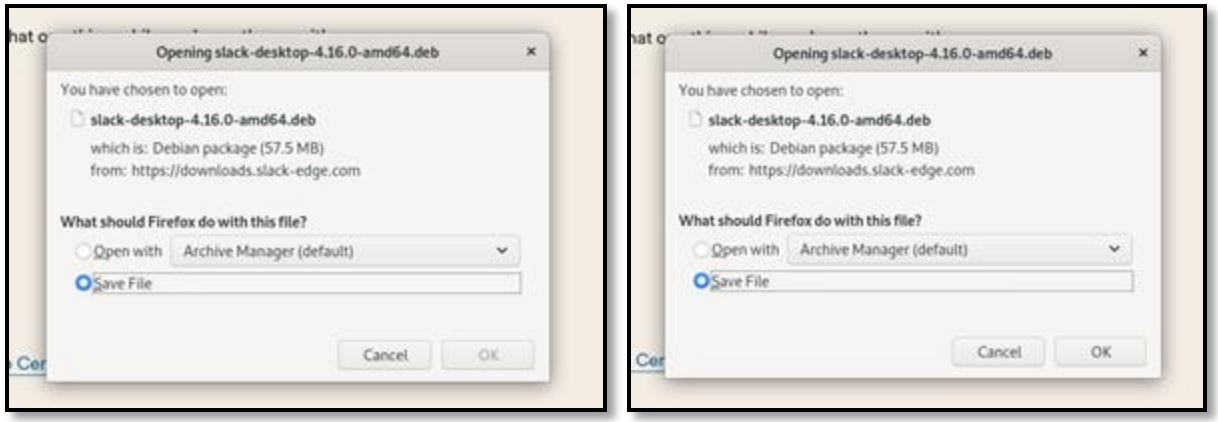
- Enforce visual integrity
  - Clear visual separation between important alerts and content
  - Examples:
    - Windows “User Account Control” darkens the entire screen and freezes the desktop
    - Firefox dialogs “cross the boundary” between the URL bar and content
      (Only the valid dialog can do this!)





CLICKJACKING DEFENSES

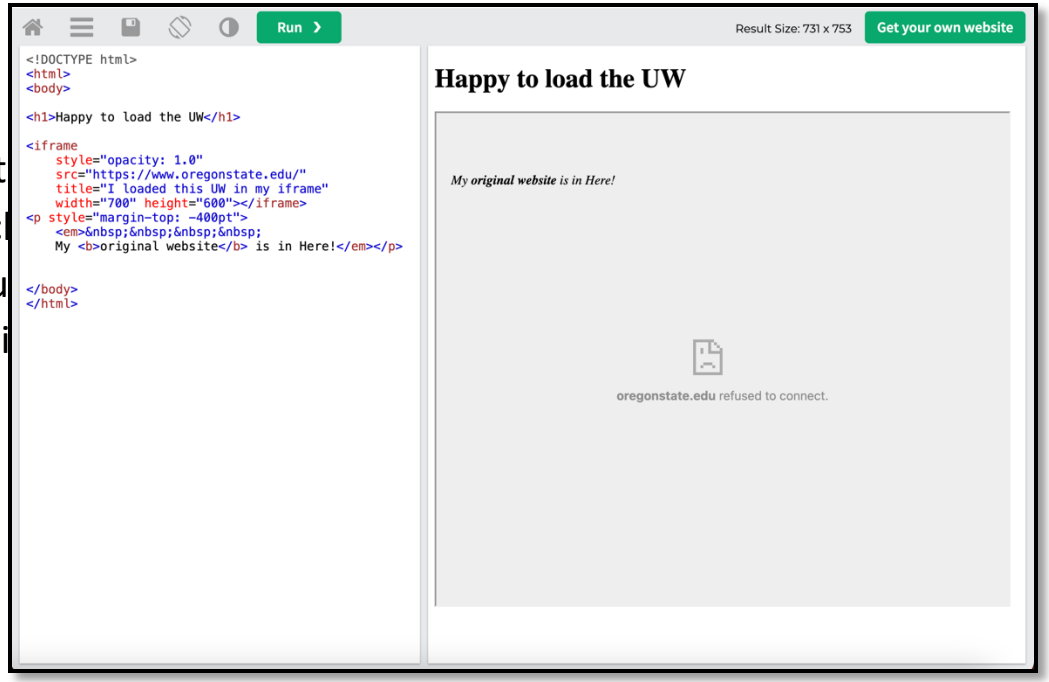
- Enforce temporal integrity
  - Sufficient time for a user to register what they are clicking on
  - Example:
    - Firefox blocks the “OK” button until 1 second after the dialog has been focused



CLICKJACKING DEFENSES

- Require confirmation from users
  - The browser needs to confirm that the user's click was intentional
  - Downside: asking for confirmation annoys users

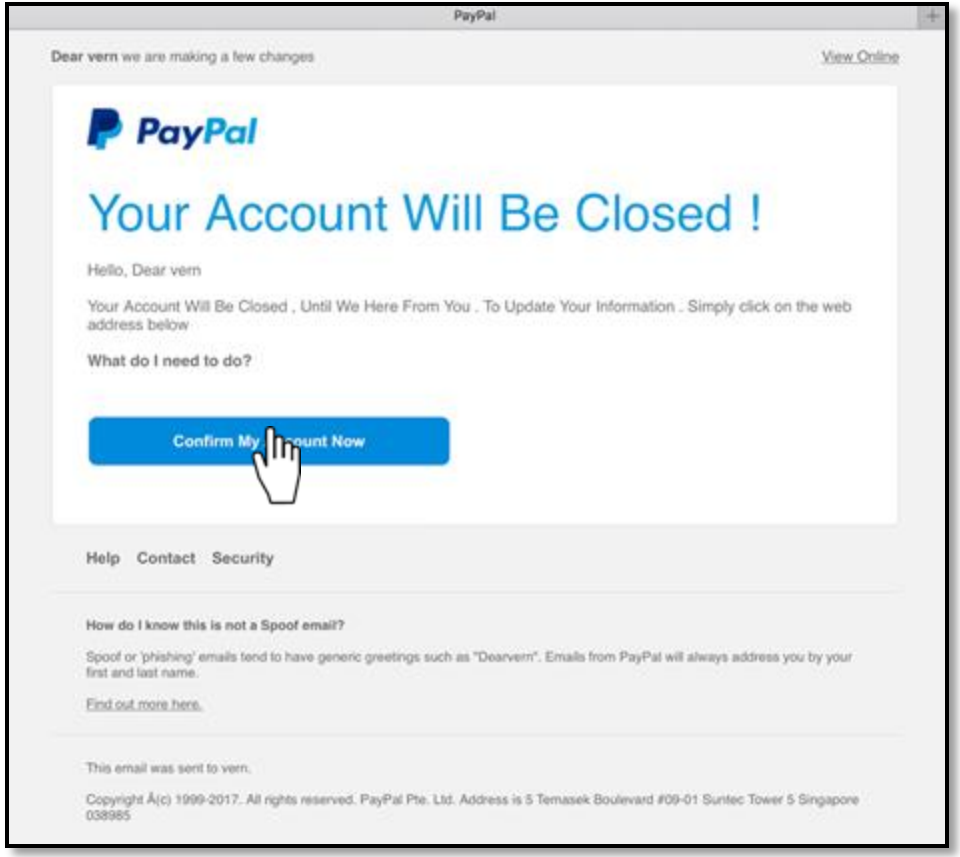
- Frame-busting
  - The legitimate website forbids ot
  - Defeats the invisible iframe attac
  - Can be enforced by Content Secu
  - Can be enforced by X-Frame-Opti



PHISHING

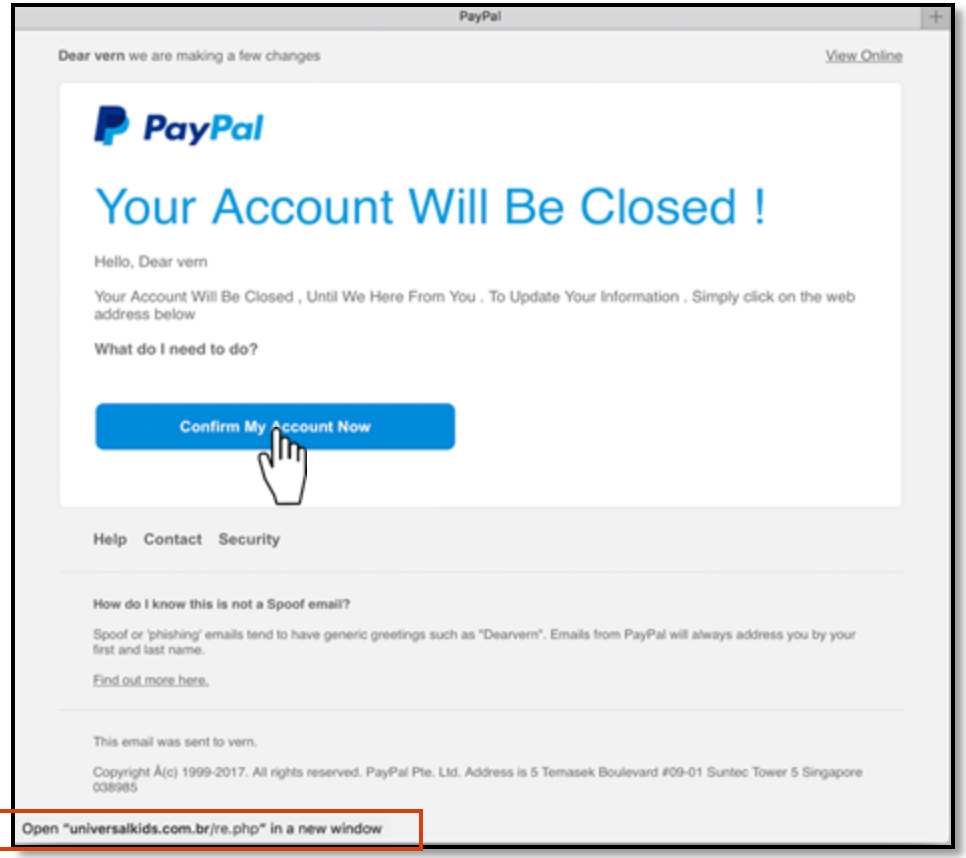
PHISHING

- Your account will be closed!



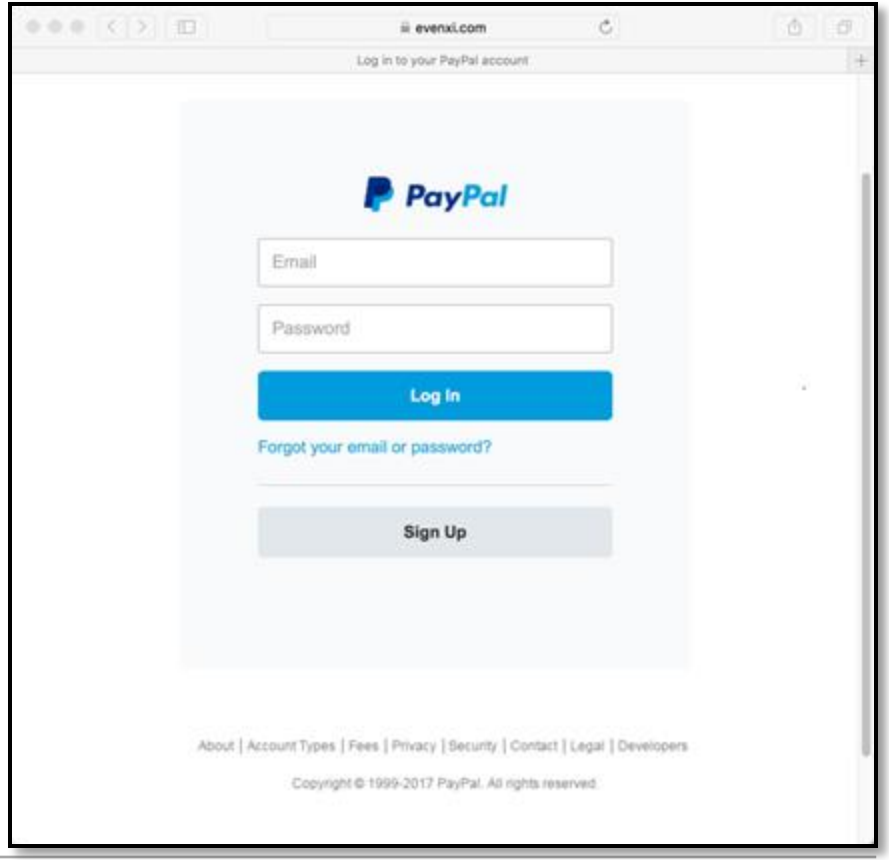
PHISHING

- Your account will be closed!
- … is it?



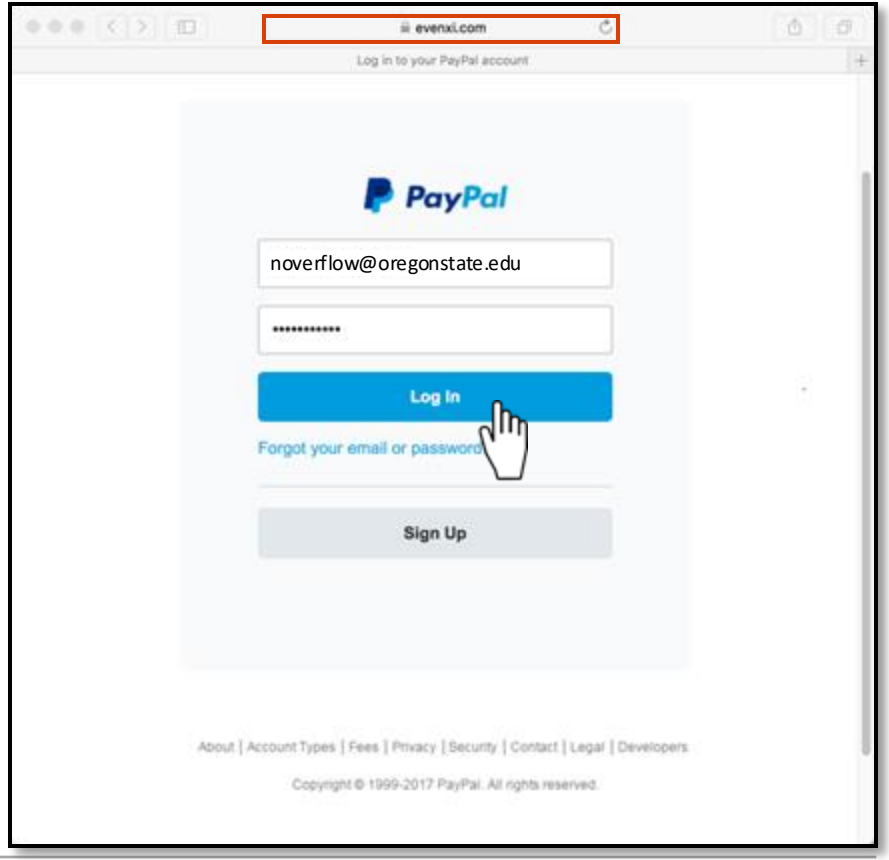
PHISHING

- You need to log-in to PayPal



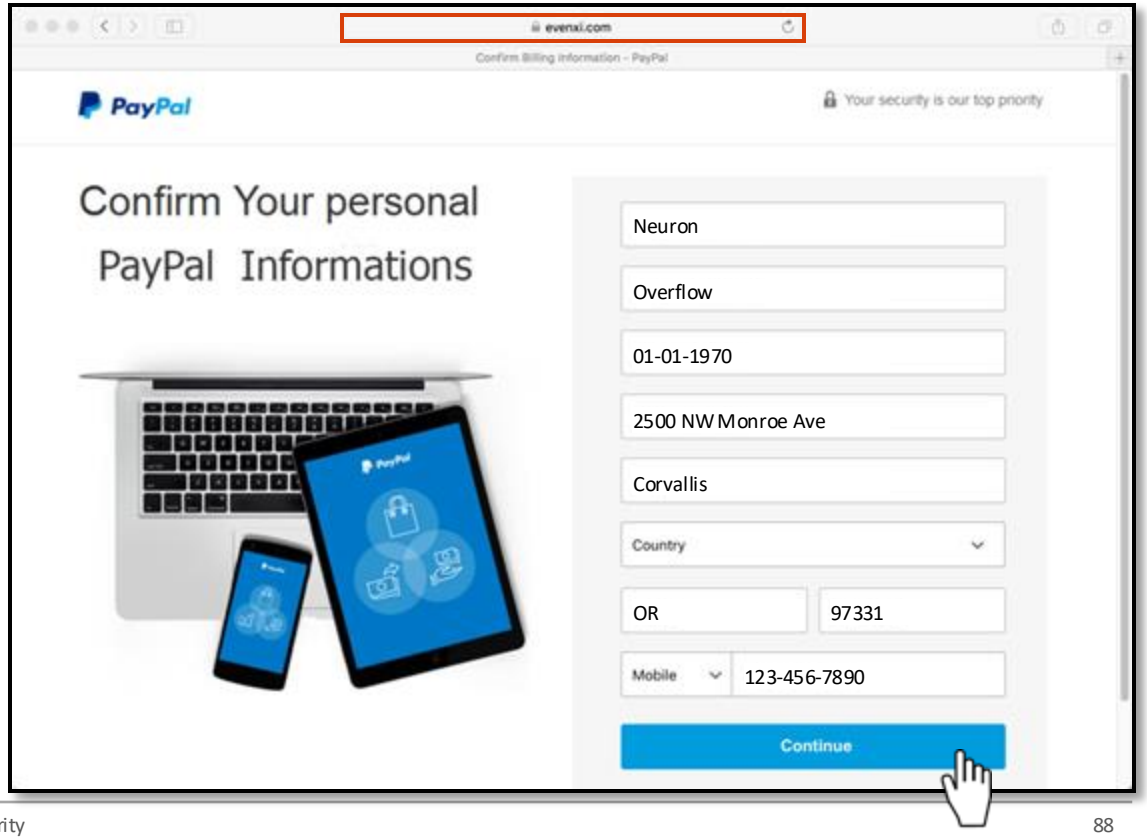
PHISHING

- You need to log-in to PayPal
- … is it?



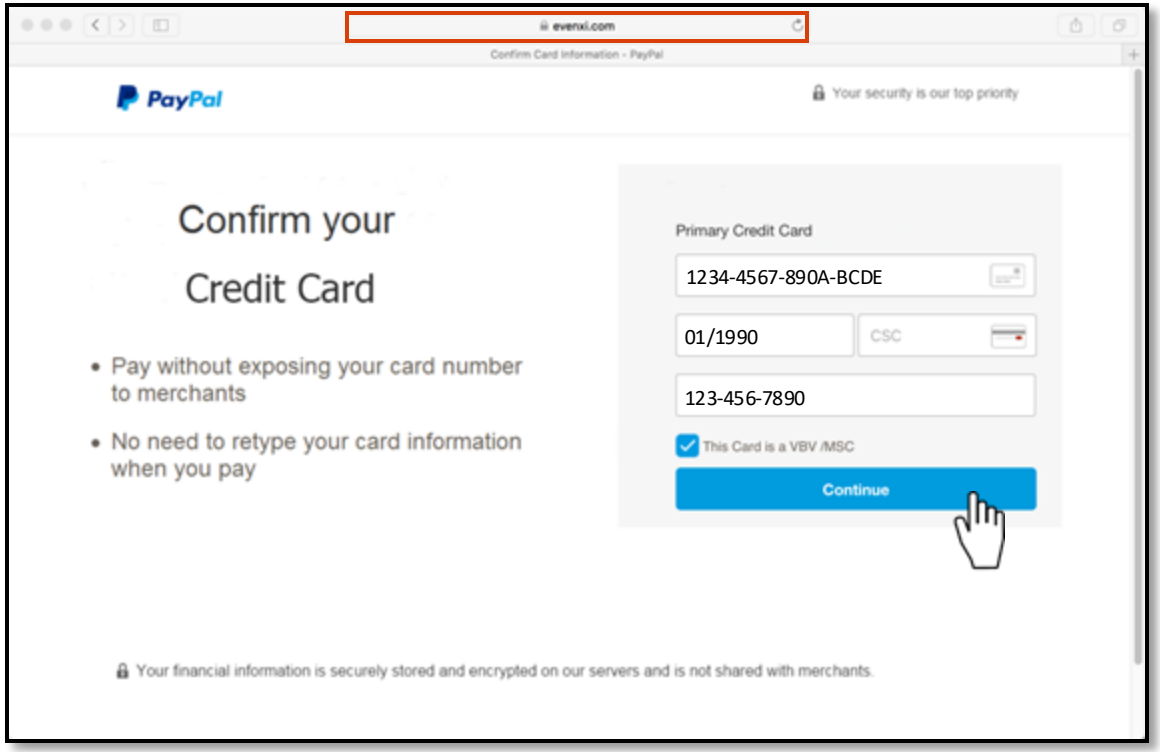
PHISHING

- You need to log-in to PayPal
- … is it?
- … is it?



PHISHING

- You need to log-in to PayPal
- … is it?
- … is it?
- … is it?

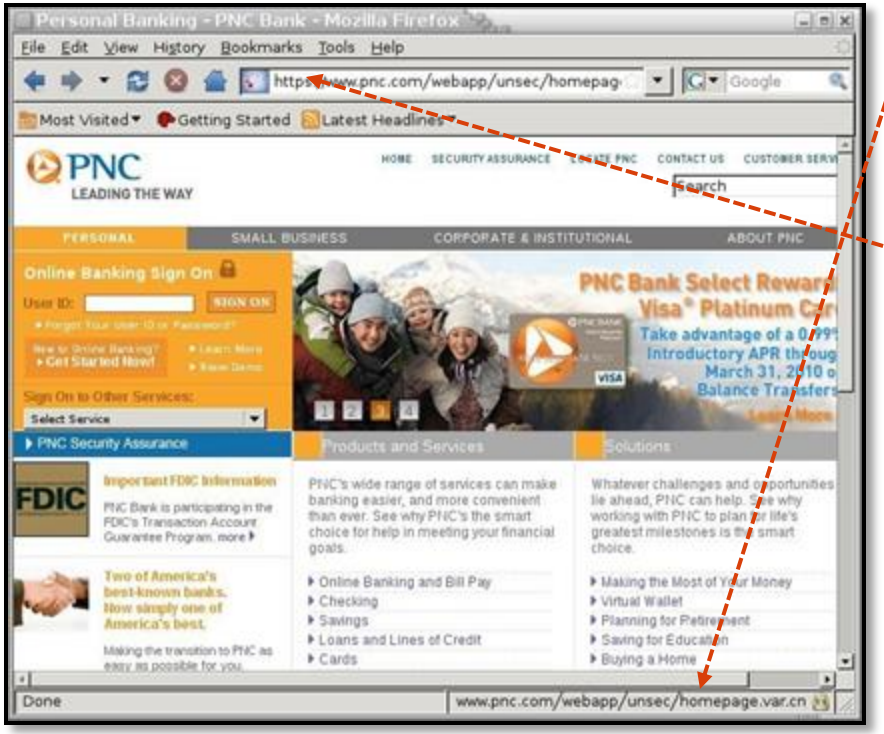


PHISHING

- Phishing
  - Trick the victim into sending the attacker personal information
  - Exploit:
    - Users can't distinguish between a legitimate website and a website impersonating the legitimate website

PHISHING: CHECK THE URL?

- Is this website real?



"www.pnc.com/webapp/unsec/homepage.var.cn" is an entire domain!

The attacker can also register an HTTP certificate for this valid domain

PHISHING: CHECK THE URL?

- Is this website real?

These letters come from the Cyrllic alphabet, not the Latin alphabet! They're rendered the same but have completely different bytes!

PHISHING: HOMOGRAPH ATTACK

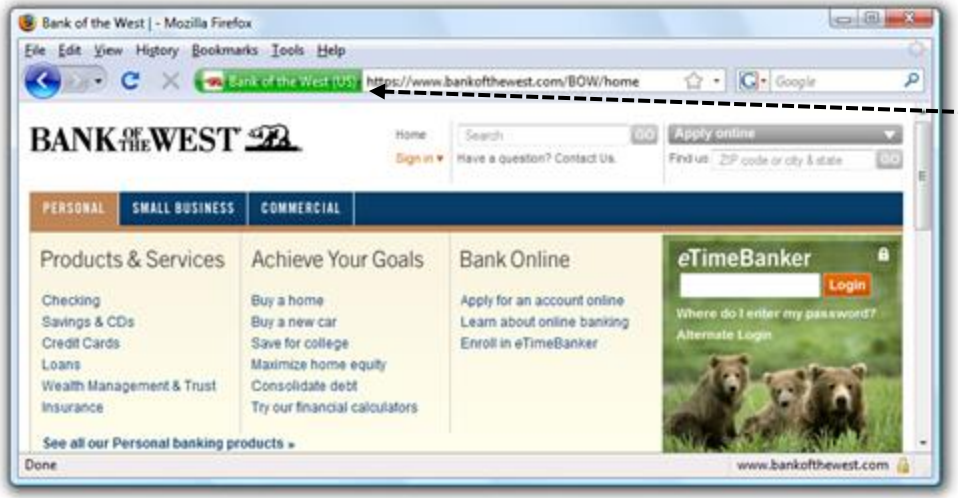
- Homograph attack
  - Create malicious URLs that look similar (or the same) to legitimate URLs
  - Homograph: two words that look the same, but have different meanings

PHISHING: HOMOGRAPH ATTACK

- Homograph attack
  - Create malicious URLs that look similar (or the same) to legitimate URLs
  - Homograph: two words that look the same, but have different meanings

PHISHING: CHECK EVERYTHING

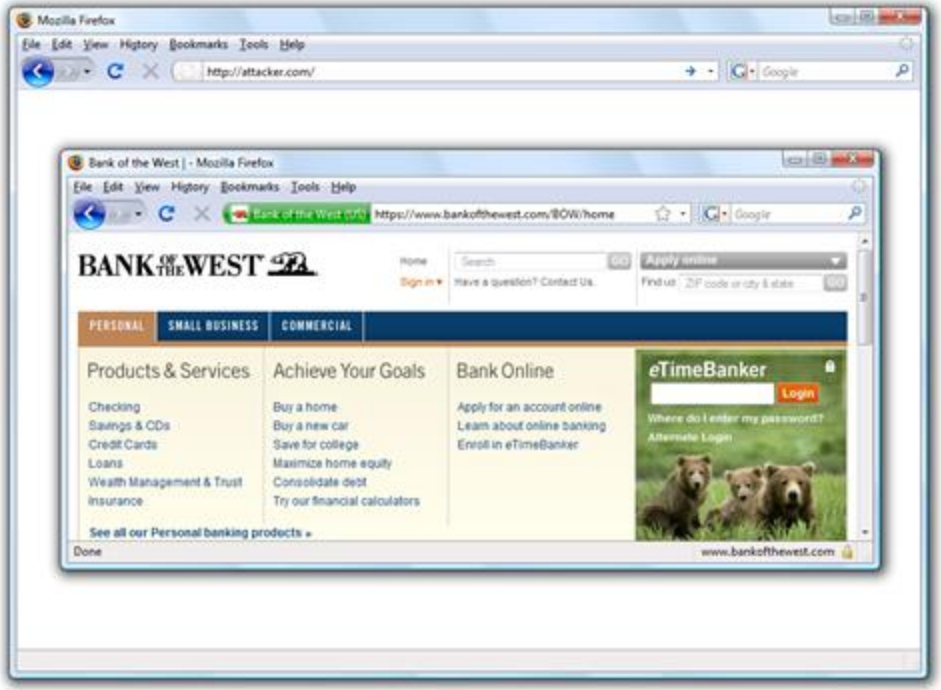
- … hmm it looks legit!



Extended Validation: CA verified the identity of the site (not just the domain)

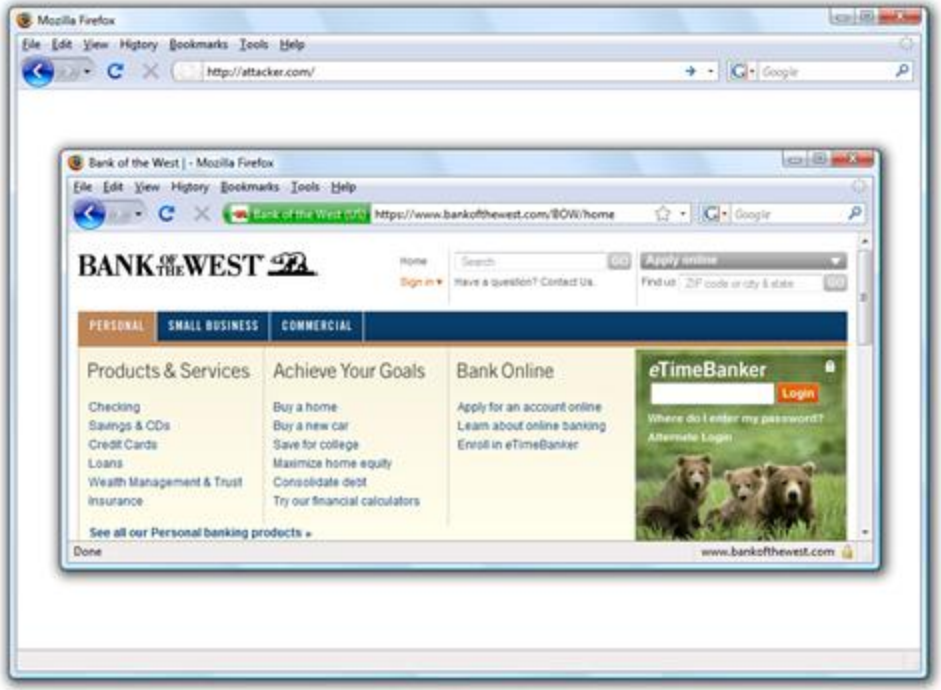
PHISHING: CHECK EVERYTHING

- … hmm it looks legit!
  - Is it?



PHISHING: BROWSER-IN-BROWSER ATTACK

- Browser-in-browser attack
  - The attacker simulates the entire web browser with JavaScript



PHISHING: NOW WHO'S THE FAULT?

- Let's not blame the users
  - They are not security experts
  - Attacks are rare
    - Users do not always suspect malicious action
    - Detecting phishing is hard, even if you're on the lookout for attacks
    - Legitimate messages often look like

Title: Your Final Grades
Sender: Hóng (sanghyun@oregonstate.com)

Hey Guys,

There are some corrections on your final exam scores. I need you to confirm your scores immediately from here.

Thanks,
Sanghyun

PHISHING DEFENSE: 2FA

- Two-factor authentication
  - Motivation
    - Phishing attacks may expose your passwords to the attackers
    - You want to make that the password is not sufficient for logging in
  - Two-factor authentication (2FA)
    - Prove their identity in two different ways before successfully logging-in
  - Three main ways for a user to prove their identity
    - Something the user knows: password, security questions
    - Something the user has: mobile devices, security keys
    - Something the user is: fingerprint, face ID
  - Stealing one factor (password) is not enough

PHISHING DEFENSE: 2FA

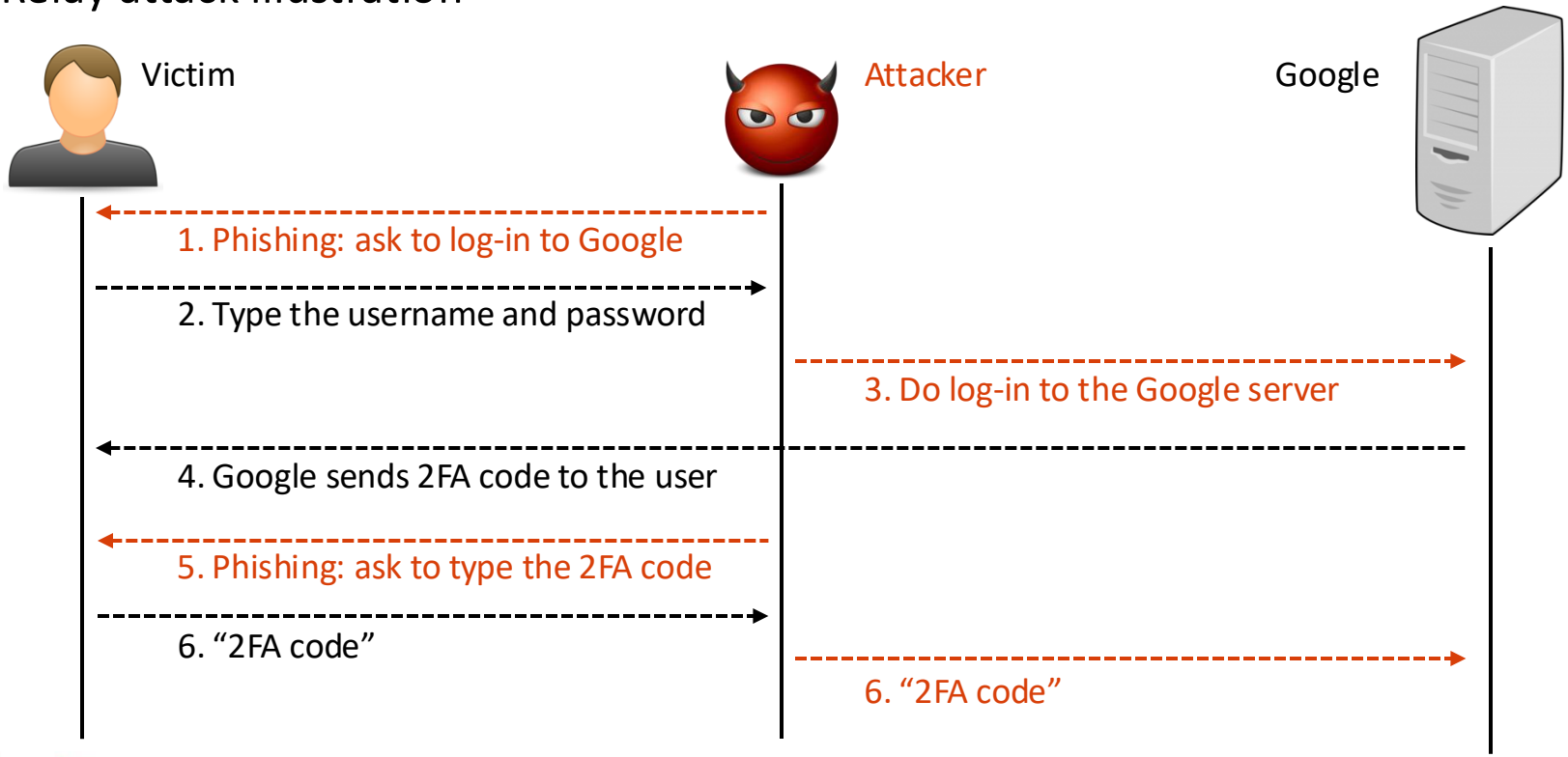
- Two-factor authentication
  - Protection scenarios
    - An attacker steals the password file and performs a dictionary attack
    - The user re-uses passwords on two different websites.
      The attacker compromises one website and tries the same password on the 2nd one

PHISHING DEFENSE: 2FA WEAKNESS

- Relay attack
  - The attacker steal both factors in a single phishing (one stone for two birds)
  - Attack example
    - User uses 2FA
    - 1st : Password (something the user knows)
    - 2nd: A code sent to the user's mobile device (something the user owns)
  - Procedure
    - The phishing website asks the user to input their password (1st factor)
    - The attacker immediately tries to log-in to the actual website as the user
    - The actual website sends a code to the user
    - The phishing website asks the user to enter the code (2nd factor)
    - The attacker enters the code to log in as the user

PHISHING DEFENSE: 2FA WEAKNESS

- Relay attack illustration



PHISHING DEFENSE: 2FA WEAKNESS

- Social engineering
  - Hijacking your phone
    - Attackers can call your phone provider (e.g., T-mobile)
    - Tell them to activate the attacker's SIM card, and will be done
    - They receive your texts
    - 2FA via SMS is not great but better than nothing
  - Bypassing customer service
    - Attackers can call customer support and ask them to deactivate 2FA!
    - Companies should validate identity if you ask to do this (but not all do)

PHISHING DEFENSE: AUTHENTICATION TOKEN

- Auth token
  - A device that generates secure second-factor codes
  - Examples:
    - RSA SecurID and Google Authenticator
  - Usage
    - The token and the server share a common secret key k
    - When the user wants to log in, the token generates a code HMAC(k, time)
    - The time is often truncated to the nearest 30 seconds for usability
    - The code is often truncated to 6 digits for usability
    - The user submits the code to the website
    - The website uses its secret key to verify the HMAC
  - Downside(s):
    - Vulnerable to relay attacks
    - Vulnerable to online brute-force attacks
    - Possible fix: add a max number of times you can request!

PHISHING DEFENSE: SECURITY KEY

- Security key
  - A 2nd factor designed to defeat phishing
  - User owns the security key
  - Usage scenario
    - User signs up for a website; the security key generates a new public/private key pair
    - User gives the public key to the website
    - If the user wants to log in, the server sends a nonce to the security key
    - The security key signs the nonce, website name (from the browser), and key ID
    - User gives the signature to the server
  - Security keys prevent phishing
    - In phishing, the security key generates a signature with the attacker's website name, not with the legitimate website name
    - Impervious to relay attacks!

Thank You!

Sanghyun Hong

https://secure-ai.systems/courses/Sec-Grad/current

SAIL
Secure AI Systems Lab