XSS and UI Attacks

Today: XSS

- XSS
  - Websites use untrusted content as control data
  - Stored XSS
  - Reflected XSS
  - Defense: HTML sanitization
  - Defense: Content Security Policy (CSP)
- UI attacks
  - Clickjacking
  - Phishing

Cross-Site Scripting (XSS)

Top 25 Most Dangerous Software Weaknesses (2020)

ITIS 6200 / 8200

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

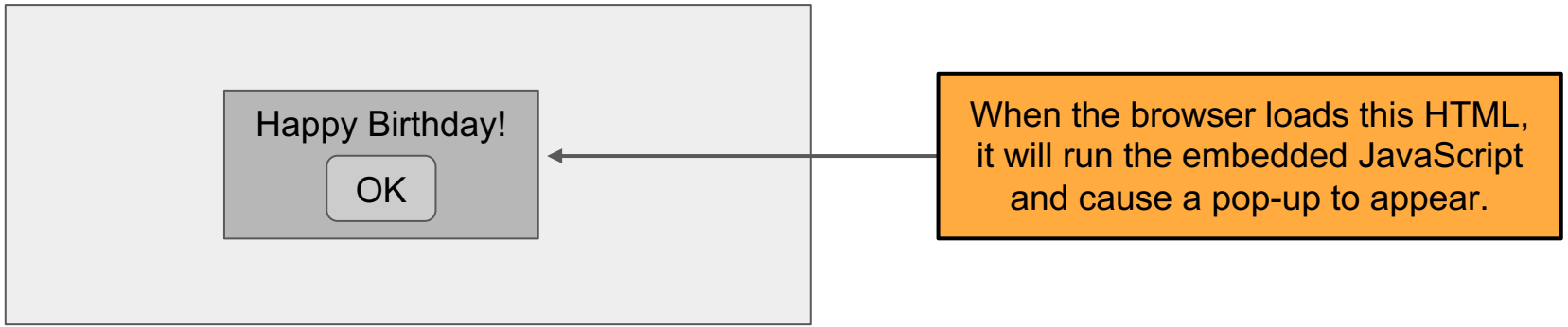
Review: JavaScript

- JavaScript can create a pop-up message

HTML (with embedded JavaScript)

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

Webpage



A Go HTTP Handler

Handler

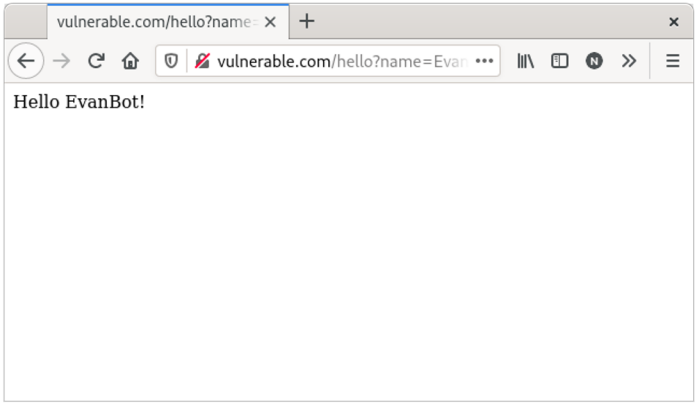
```
func handleSayHello(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Query()["name"][0]
    fmt.Fprintf(w, "<html><body>Hello %s!</body></html>", name)
}
```

URL

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

Response

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



A Go HTTP Handler

Handler

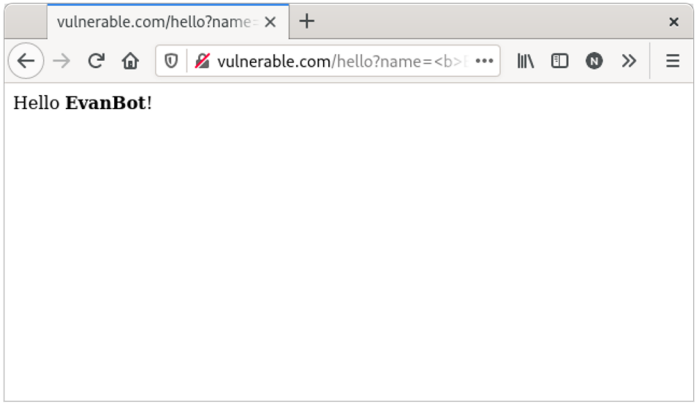
```
func handleSayHello(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Query()["name"][0]
    fmt.Fprintf(w, "<html><body>Hello %s!</body></html>", name)
}
```

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>
```

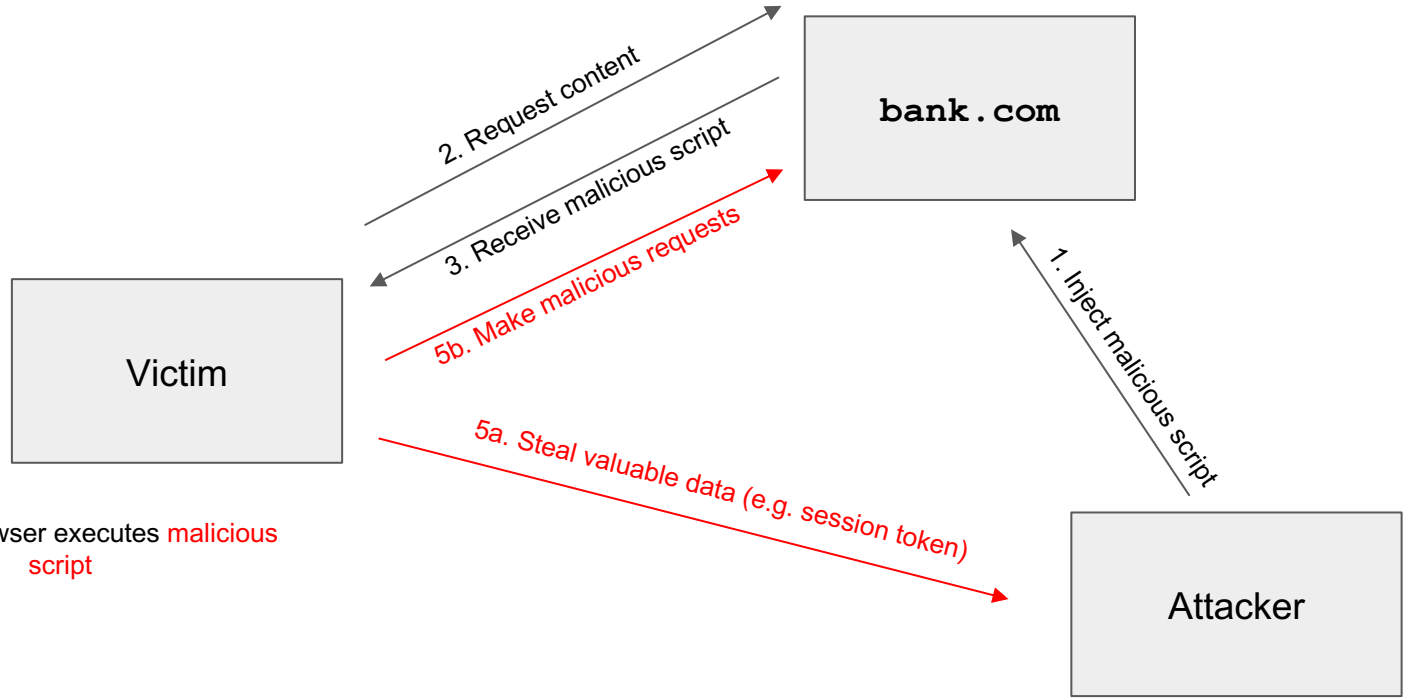
Cross-Site Scripting (XSS)

- 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

Stored XSS

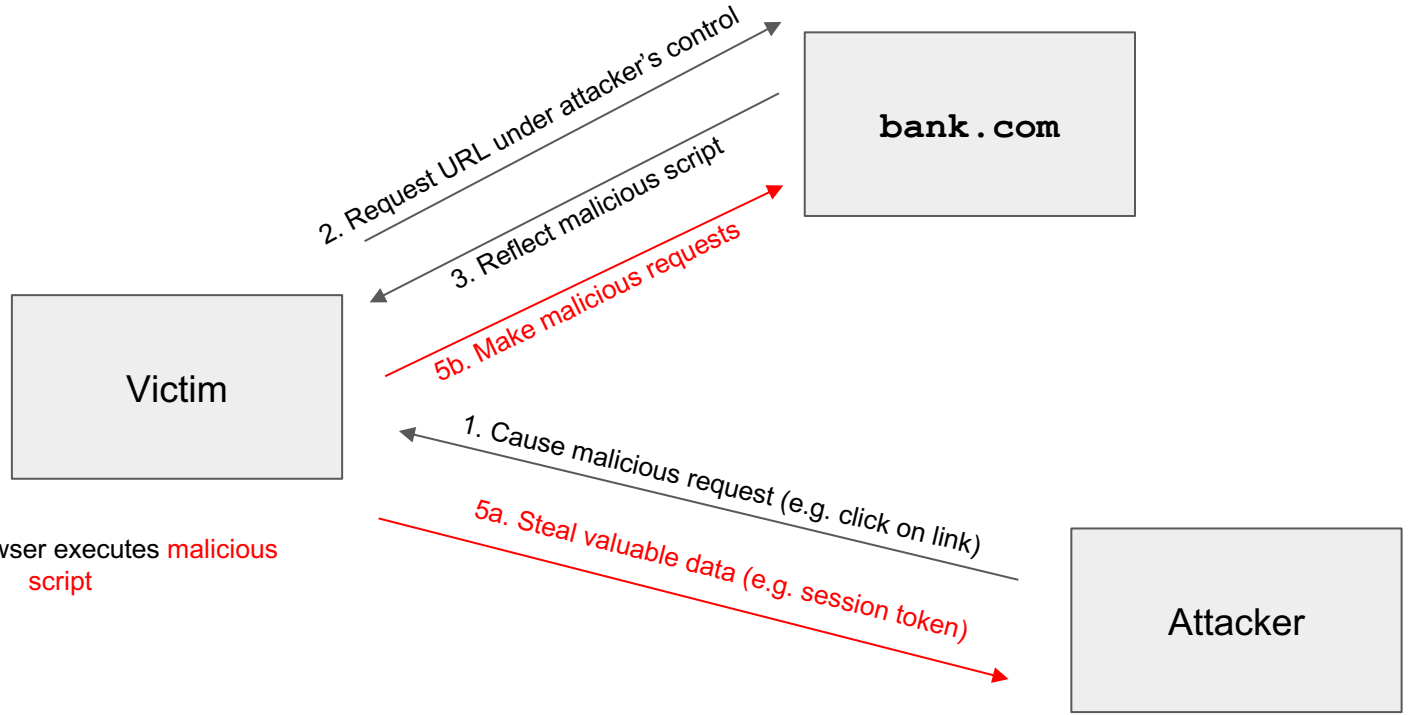


4. Victim browser executes malicious script

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 Defenses

- 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!
- Note: 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 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: 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://test.com/hack.js">`
- Also works with other content (e.g. iframes, images, etc.)
- Relies on the browser to enforce security

UI Attacks

User Interface (UI) Attacks

- General theme: The attacker tricks the victim into thinking they are taking an **intended** action, when they are actually taking a **malicious** action
  - Takes advantage of **user interfaces**: The trusted path between the user and the computer
    - Browser disallows the website itself to interact across origins (same-origin policy), but trusts the user to do whatever they want
  - Remember: Consider human factors!
- Two main types of UI attacks
  - Clickjacking: Trick the victim into clicking on something from the attacker
  - Phishing: Trick the victim into sending the attacker personal information

Clickjacking

- **Clickjacking**: Trick the victim into clicking on something from the attacker
- Main vulnerability: the browser trusts the user's clicks
  - When the user clicks on something, the browser assumes the user intended to click there
- Why steal clicks?
  - Download a malicious program
  - Like a Facebook page/YouTube video
  - Delete an online account
- Why steal keystrokes?
  - Steal passwords
  - Steal credit card numbers
  - Steal personal info

Clickjacking: Download buttons

- Which is the real download button?
- What if the user clicks the wrong one?

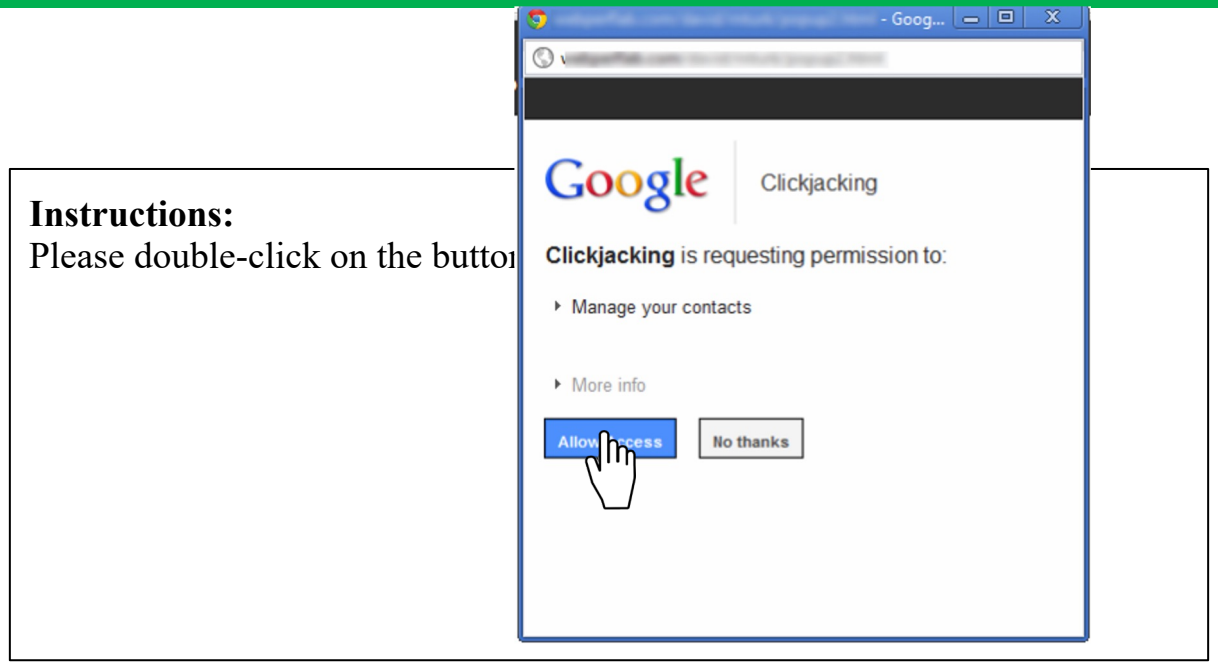
Clickjacking: Invisible iframes

- Variant #1: Frame the legitimate site visibly, *under* invisible malicious content
- Variant #2: Frame the legitimate site invisibly, *over* visible, enticing content
- Variant #3: Frame the legitimate site visibly, *under* malicious content *partially* overlaying the site

Clickjacking: Temporal Attack

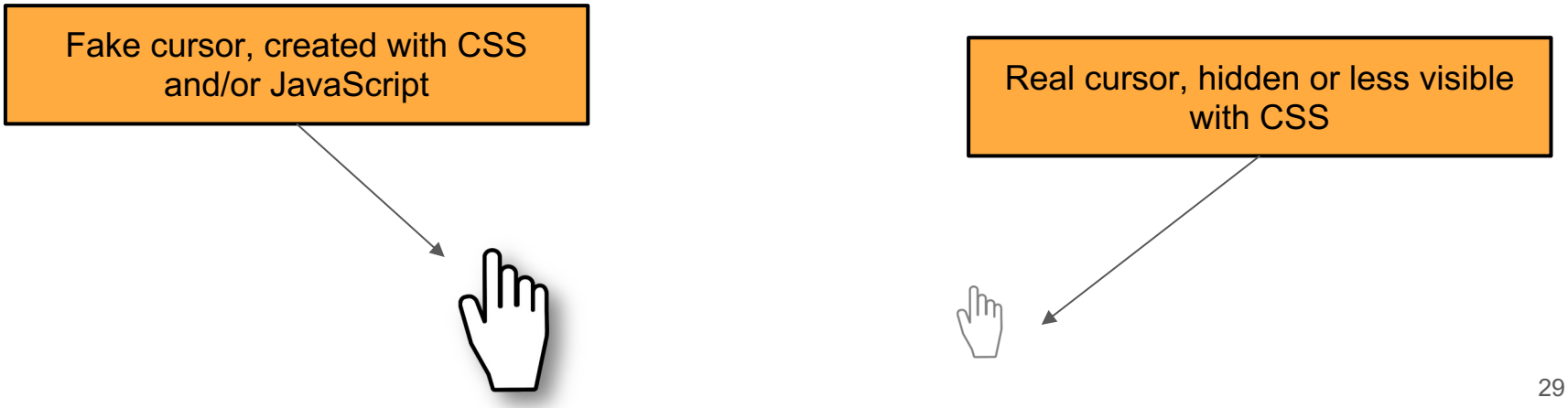
- JavaScript can detect the position of the cursor and change the website right before the user clicks on something
  - The user clicks on the malicious input (embedded iframe, download button, etc.) before they notice that something changed

Clickjacking: Temporal Attack



Clickjacking: Cursorjacking

- CSS has the ability to style the appearance of the cursor
- JavaScript has the ability to track a cursor's position
- If we change the appearance a certain way, we can create a fake cursor to trick users into clicking on things!

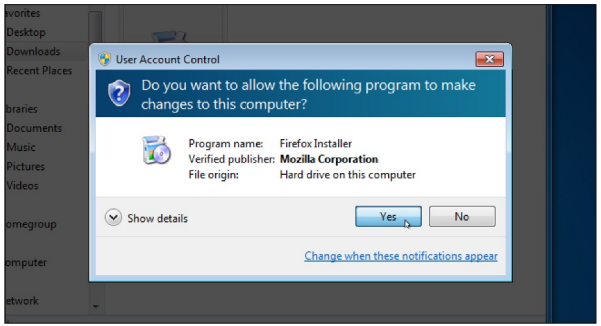


Clickjacking: Cursorjacking

What do you think you're clicking on?

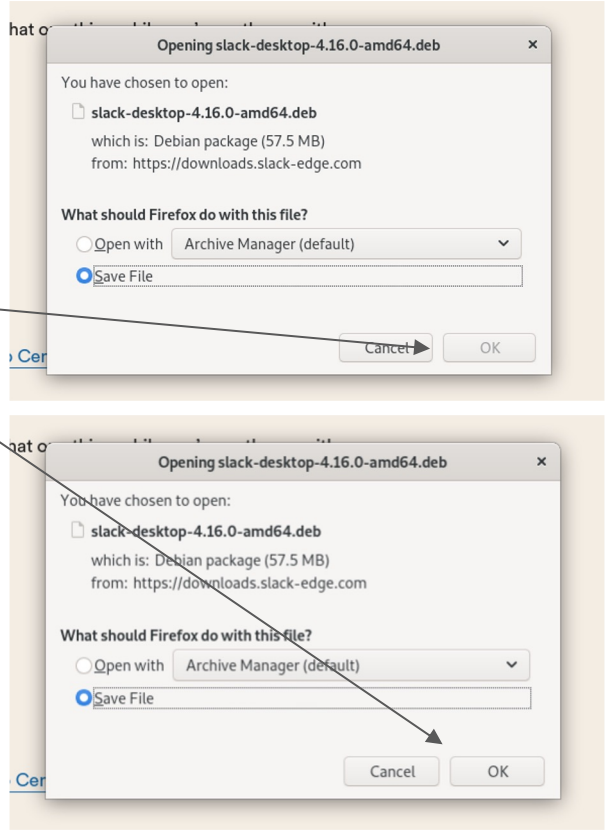
Clickjacking: Defenses

- **Enforce visual integrity**: Ensure clear visual separation between important dialogs and content
  - Notice: Windows User Account Control darkens the entire screen and freezes the desktop



Clickjacking: Defenses

- **Enforce temporal integrity**: Ensure that there is sufficient time for a user to register what they are clicking on
  - Notice: 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
  - Drawbacks: Asking for confirmation annoys users (consider human factors!)

- **Frame-busting**: The legitimate website forbids other websites from embedding it in an iframe
  - Defeats the invisible iframe attacks
  - Can be enforced by Content Security Policy (CSP)
  - Can be enforced by X-Frame-Options (an HTTP header)

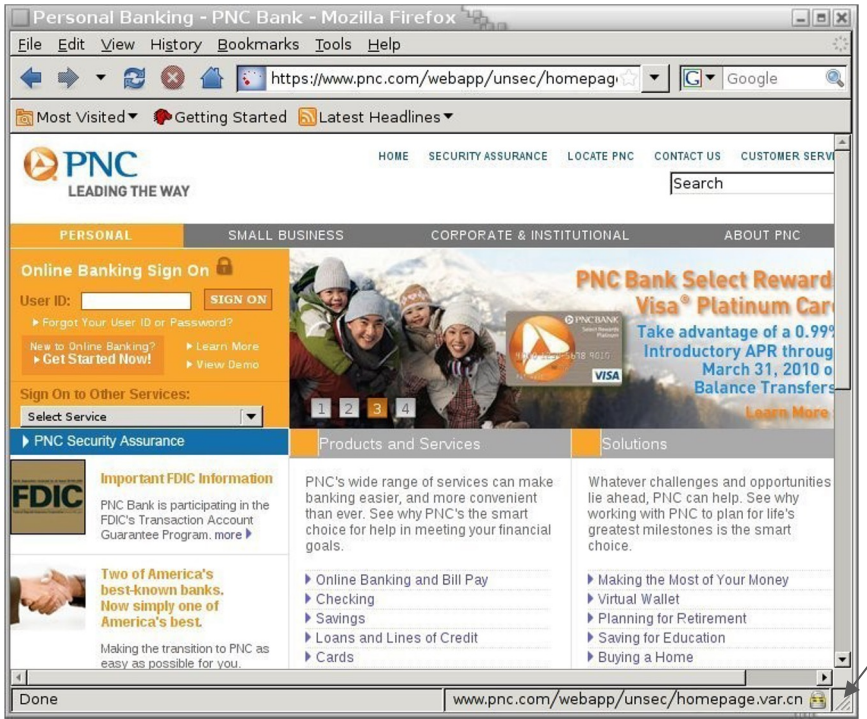
Phishing

- **Phishing**: Trick the victim into sending the attacker personal information

- Main vulnerability: The user can’t distinguish between a legitimate website and a website *impersonating* the legitimate website

Phishing: Check the URL?

Is this real?



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

The attacker can still register an HTTPS certificate for the perfectly valid domain

Phishing: Check the URL?

Is *this* real?

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

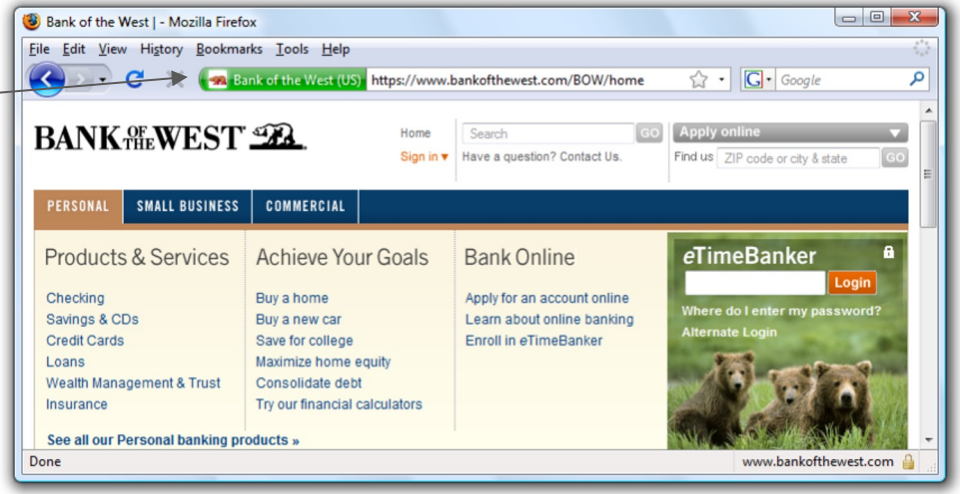
Phishing: Homograph Attacks

- Idea: Check if the URL is correct?
- **Homograph attack**: Creating 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*

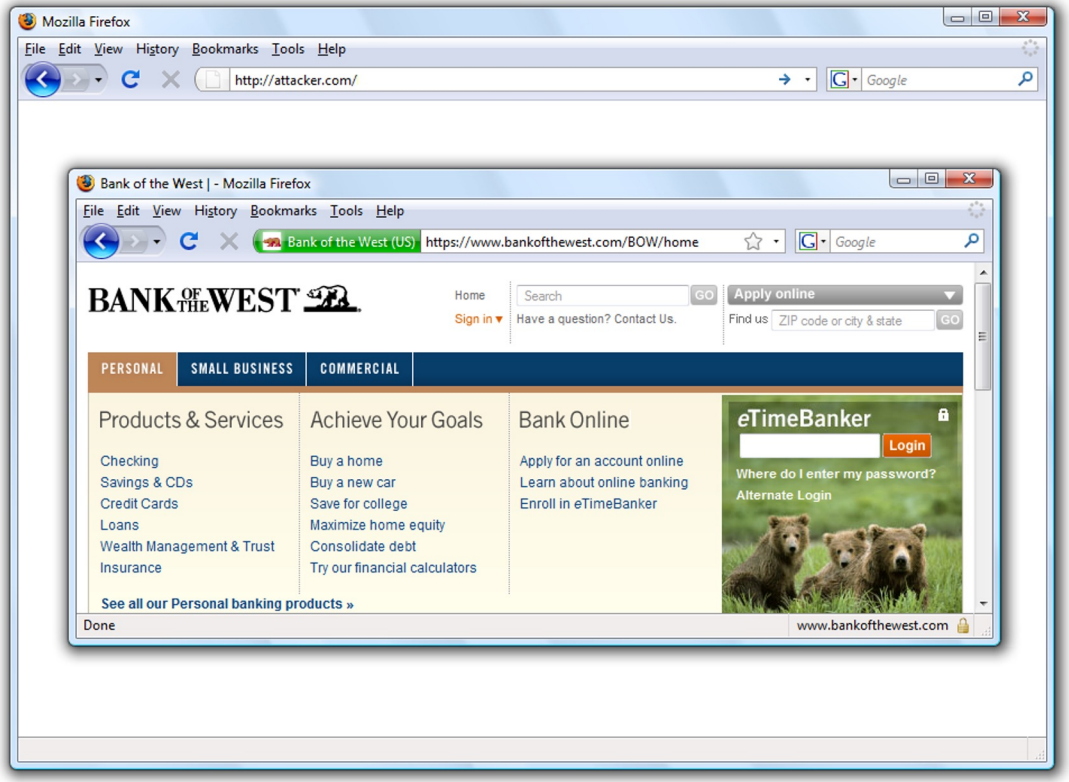
Is *this* real?

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



Phishing: Check *Everything*

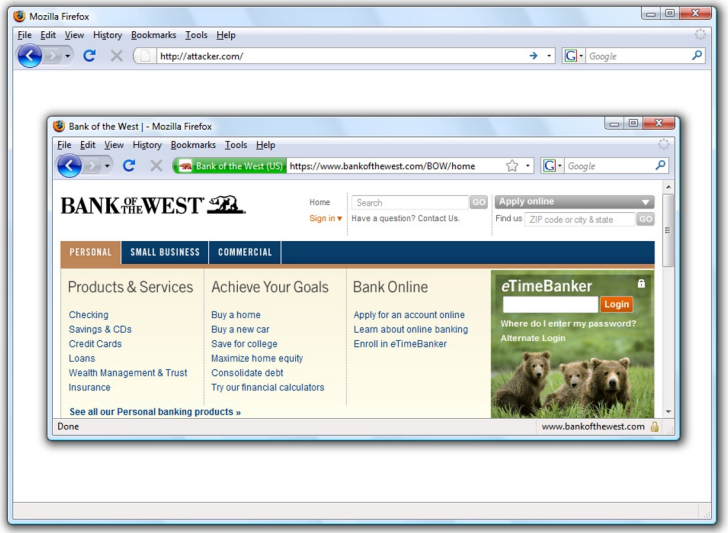
Is *this* real?



Phishing: Browser-in-browser Attacks

- Idea: Check for a green padlock icon in the browser's address bar, or any other built-in browser security feature

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



Phishing: Don’t Blame the Users

- Most users aren’t security experts
- Attacks are uncommon: users don’t always suspect malicious action
- Detecting phishing is hard, even if you’re on the lookout for attacks
  - Legitimate messages often look like phishing attacks!

Two-Factor Authentication

- Problem: Phishing attacks allow attackers to learn passwords
- Idea: Require more than passwords to log in
- **Two-factor authentication** (**2FA**): The user must prove their identity in two different ways before successfully authenticating
- Three main ways for a user to prove their identity
  - **Something the user knows**: Password, security question (e.g. name of your first pet)
  - **Something the user has**: Their phone, their security key
  - **Something the user is**: Fingerprint, face ID
- Even if the attacker steals the user's password with phishing, they don't have the second factor!

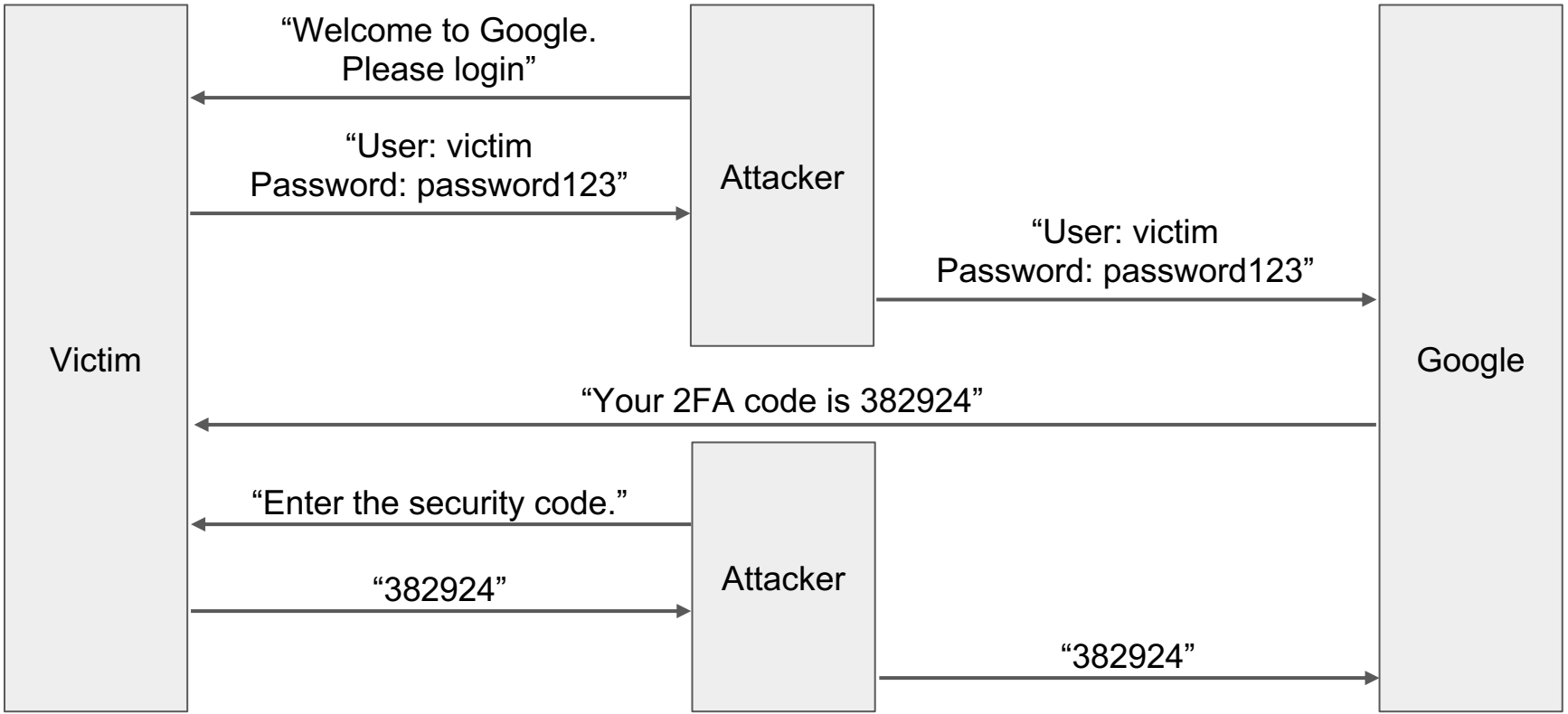
Two-Factor Authentication

- Two-factor authentication also defends against other attacks where a user's password is compromised
  - Example: An attacker steals the password file and performs a dictionary attack
  - Example: The user reuses passwords on two different websites. The attacker compromises one website and tries the same password on the second website
  - With 2FA, the password alone is no longer enough for the attacker to log in!

Subverting 2FA: Relay Attacks

- **Relay attacks** (**transient phishing**): The attacker steals both factors in a phishing attack
- Example
  - Two-factor authentication scheme
    - First factor: The user's password (something the user knows)
    - Second factor: A code sent to the user's phone (something the user owns)
  - Attack
    - The phishing website asks the user to input their password (first 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 (second factor)
    - The attacker enters the code to log in as the user

Subverting 2FA: Relay Attacks



Subverting 2FA: Social Engineering

- Some 2FA schemes text a one-time code to a phone number
  - Attackers can call your phone provider (e.g. Verizon) and tell them to activate the attacker's SIM card, so they receive your texts!
  - 2FA via SMS is not great but *better than nothing*

- Some 2FA schemes can be bypassed with customer support
  - 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)

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

Summary: Clickjacking

- Clickjacking: Trick the victim into clicking on something from the attacker
- Main vulnerability: the browser trusts the user's clicks
  - When the user clicks on something, the browser assumes the user intended to click there
- Examples
  - Fake download buttons
  - Show the user one frame, when they're actually clicking on another invisible frame
  - Temporal attack: Change the cursor just before the user clicks
  - Cursorjacking: Create a fake mouse cursor with JavaScript
- Defenses
  - Enforce visual integrity: Focus the user's vision on the relevant part of the screen
  - Enforce temporal integrity: Give the user time to understand what they're clicking on
  - Ask the user for confirmation
  - Frame-busting: The legitimate website forbids other websites from embedding it in an iframe

Summary: Phishing

- Phishing: Trick the victim into sending the attacker personal information
  - A malicious website impersonates a legitimate website to trick the user
- Don’t blame the users
  - Detecting phishing is hard, especially if you aren’t a security expert
  - Check the URL? Still vulnerable to homograph attacks (malicious URLs that look legitimate)
  - Check the entire browser? Still vulnerable to browser-in-browser attacks
- Defense: Two-Factor Authentication (2FA)
  - User must prove their identity two different ways (something you know, something you own, something you are)
  - Defends against attacks where an attacker has only stolen one factor (e.g. the password)
  - Vulnerable to relay attacks: The attacker phishes the victim into giving up both factors
  - Vulnerable to social engineering attacks: Trick humans to subvert 2FA
  - Example: Authentication tokens for generating secure two-factor codes
  - Example: Security keys to prevent phishing