Original Browser

- Static content presented to clients
- Servers were responsible for dynamic parts
- Security attacks were focused on servers
  - Malformed URLs, buffer overflows, root paths, unicode attacks
Today’s Browsers – Complexity Creeps In

• JavaScript – allows code execution
• Document Object Model (DOM) & Cascading Style Sheets (CSS) – change appearance of page
• XMLHttpRequest (AJAX) – asynchronously fetch content
• WebSockets – open interactive communication session between JavaScript on a browser and a server
• Multimedia support - `<audio>`, `<video>`, `<track>`
  – MediaStream recording (audio and video), speech recognition & synthesis
• Geolocation
WebAssembly (Wasm) & NaCl

• **Google Native Client (NaCl)**
  – Download binary software and run it in your browser
  – Sandboxing and load-time code verification for safety

• **WebAssembly**
  – Execution of compiled code by a browser via a processor virtual machine
  – Simple, stack-based virtual machine
    • Sandboxed & designed with security in mind … but so was Java
    • Control flow hijacks and heap buffer overflows have been demonstrated
  – Harder to detect malware & more opportunities to disguise malware
  – Has been great for cryptominers
    • Malicious web pages can run cryptomining software far more efficiently than with JavaScript
Complexity creates a huge threat surface

• More features → more bugs
• Browsers experienced a rapid introduction of features
• Browser vendors don’t necessarily conform to all specs
• Check out quirksmode.org
Web Security Model
Most desktop & mobile apps come from one place
- They may use external libraries, but those are linked in and tested

Web pages usually have components from different places

E.g., www.cnn.com has
- Fonts from www.i.cdn.cnn.com
- Images from cdn.cnn.com, analytics.twitter.com, d.agkn.com, cm.everesttech.com, sb.scorecardresearch.com, ping.charbeat.net, ...
- Scripts fagility.conn.com, beacon.krxd.net, beacon.s-onetag.com, cdn.boomtrain.com, cdn.cookielaw.org, cdn.segment.com, cnd3.optimizely.com, cnn.bouceexchange.com, connect.facebook.net, ...
- XMLHttpRequests from wmff.warnermedia.com, geo.ngtv.io, telemetry.api.wmcdp.io, zone-manager.izi, container-manager.html, ...
- Other content from s-onetag.com
What should code on a page have access to?

- Can analytics code access JavaScript variables from a script loaded from jQuery.com on the same page?

  Scripts are from different places
  ... *but the page author selected them so shouldn’t that be OK?*

- Can analytics scripts interact with event handlers?

- How about embedded frames?
Background: Frames and iFrames

• **Browser window may contain embedded frames**
  – Each frame contains independent web or media content that may come from different sources
  – **Frame** = rigid division as part of frameset (no longer used)
  – **iFrame** = floating inline frame

• **Why use them?**
  – Delegate screen area to content from another source
  – Browser provides isolation based on frames
  – Parent can continue to function even if frame is broken
Web application security policy goals

• Safe to visit a web site

• Safe to visit two pages at one time
  – Address bar distinguishes them

• iFrame inside a parent frame?
  – We want to allow safe delegation
  – Each frame = origin of the HTML content within it

  **Same-origin policy:**
  - a.com cannot access b.com’s content
  - b.com cannot access a.com’s content
  …if a.com and b.com have different origins
Web application security model: **same-origin policy**

A browser permits scripts in one page to access data in a second page **only if** both pages have the same origin.

**Origin** = \{ URI scheme, hostname, port number \}

- **Same origin**

- **Different origin from above**
  - http://poopybrain.com/index.html – different host

JavaScript in a frame is **not** be able to access any data of other frames that do not share the same origin.
How the same-origin policy works

• Each frame is assigned the origin of its URL

• Each origin has access to its own client-side resources
  – **Cookies**: simple way to implement state (sets of *name*, *value* tuples)
    • Browser sends cookies associated with the origin
  – **DOM storage**: key-value storage per origin
  – **JavaScript namespace**: functions & variables
  – **DOM tree**: JavaScript version of the HTML structure

• JavaScript code executes with the authority of its frame’s origin
  – If cnn.com loads JavaScript from jQuery.com, the script runs with the authority of cnn.com

• Passive content (CSS files, images) has *no* authority
  – It doesn’t (and shouldn’t) contain executable code
Can two different frames communicate?

• Generally, no – they’re isolated if they’re not the same origin
• But `postMessage()` allows a script to send a message to the window
  – A receiver in another frame can pick up an `onmessage` event
• Both sides must opt in
Mixed content: http & https

• HTTPS page may contain HTTP content:
  `<script src="http://www.mysite.com/script.js"> </script>`
  – Active network attacker may now hijack the session
  – Content over the network is plain text

• Safer approach: use HTTPS and don’t specify the scheme for content
  `<script src="/www.mysite.com/script.js"> </script>`
  – Served over the same protocol as the embedding page (frame)

• Some browsers block mixed content
  – But this behavior can be disabled
Passive content has no authority

Makes sense … but why does it matter?

Usually it doesn’t … but …

**MIME sniffing attack**

- Old versions of IE would examine leading bytes of object to fix wrong Content-Type headers
- Suppose a malicious user uploaded an image (passive content) to a web server
  - The content would really be HTML & JavaScript to the content
- IE would allow the download but reclassify the content as HTML with JavaScript
Cross-origin weirdness

• **Images**
  – A frame can load images from anywhere
  – But … same-origin policy does not allow it to inspect the image
  – However, it can infer the size of the rendered image

• **CSS**
  – A frame can embed CSS from any origin but cannot inspect the text in the file
  – **But:**
    It can discover what the CSS does by creating DOM nodes and seeing how styling changes

• **JavaScript**
  – A frame can fetch JavaScript and execute it … but not inspect it
  – But … you can call `myfunction.toString()` to get the source
  – Or … just download the source via a `curl` command and look at it
Cross-Origin Resource Sharing (CORS)

• Browsers enforce the **same-origin policy**
  – JavaScript can only access content from the same origin
    • Images, CSS, iframes within the page, embedded videos, other scripts, …
    • It cannot make asynchronous requests to other origins (e.g., via XMLHttpRequest)

• But a page will often contain content from multiple origins
  – Images, CSS, scripts, iframes, videos

• **CORS** allows a server to define other origins as equivalent
  – Example, a server at `service.example.com` may respond with
    
    `Access-Control-Allow-Origin: http://www.example.com`

    Stating that it will treat `www.example.com` as the same origin
CORS Summary

CORS allows a server to define other servers as having the same origin. Those origins can access the requested content as if it was their own origin.
Names & IP addresses

• A frame can send http & https requests to hosts that match the origin

• The security of same origin is tied to the security of DNS
  – Recall the DNS rebinding attack
    • Register attacker.com; get user to visit attacker.com
    • Browser generates DNS request for attacker.com ⇒ DNS response contains a really short TTL
    • After the first access, attacker reconfigures the DNS server
    • Binds attacker.com to the alternate IP address
  
  – JavaScript on a site can fetch a new object from a different address
    • Web browser only sees the domain name and thinks request goes to an external site
    • Really, it goes to a server in the victim’s network
  
  – The attacker can access data within the victim’s servers and send data back to an attacker’s site … all by dynamically changing the name-address mapping
DNS Rebinding attacks

• Solution – no foolproof solutions
  – Don’t allow DNS resolutions to return internal addresses
  – Force longer TTL even if the DNS response has a short value
HTTP Cookies

Mechanism created to allow websites to manage browser state
- Cookie = small chunk of data sent by a server to a browser with a page
- Browser sends the cookie back for future requests to the server
- A browser may have an arbitrary # of cookies for a site

When a browser generates an HTTP request it sends all matching cookies
What are cookies used for?

1. **Session management (authentication cookies)**
   - Track a user's activity on a web site
     - Manage a shopping cart even if a user isn't logged in
   - Track whether a user is logged into a site
     - Upon successful login, the server sends a session ID cookie
     - This is sent with every future request to the site so it knows you’re logged in
   - Allows sites like Amazon, eBay, Instagram, Facebook to not prompt you for repeated logins
What are cookies used for?

2. Personalization
   - User preferences
   - Page rendering options
   - Content
   - Form data
What are cookies used for?

3. Tracking

- Server creates a cookie with a unique ID if browser doesn't provide a cookie
- Cookie will be sent for each page requested from the web site
- Server tracks requested URL & time of request
- Correlate activity with user if (when) user logs in
Types of cookies

- **Session cookies**
  - Only stored in memory
  - Disappear when browser closes
  - No expiration date

- **Persistent cookies**
  - Stored to disk – persists across browser invocations
  - Have an expiration date

**Session cookie:** Set-Cookie: name=paul;

**Persistent cookie:** Set-Cookie: name=paul; expires=Sun, 24 Apr 2022 17:30:00 GMT;
When & where are cookies sent?

- **Cookies are sent only to the domain & path associated with them**
  - **domain**: domain to send the cookie with each HTTP request
    - Default: cookie belongs to the domain of the origin
    - Server can specify domain for a cookie
      - Tail component pattern match for domain name
        - **domain=poopybrain.com**
        - Will match `www.poopybrain.com`, `419.poopybrain.com`, `public.poopybrain.com`, etc.
  - **path**: path in the URL for which to send the cookie
    - Leading substring match. Browser will send the cookie to all paths under the root:
      - `Set-Cookie: name=paul; path=/`
      - Browser will send the cookie to `/419`, `/419grades`, `/419-backup`, etc.
      - `Set-Cookie: name=paul; path=/419`
Securing Cookies

Cookies are often used to track server sessions
If malicious code can modify the cookie or give it to someone else, an attacker may be able to
• View your shopping cart
• Get or use your login credentials
• Have your web documents or email get stored into a different account

• **HttpOnly** flag: disallows scripts from accessing the cookie
• **Secure** flag: send the cookie only if there is an https session

```
Set-Cookie: username=paul; path=;/; HttpOnly; Secure
```
Third-party cookies: tracking

Third-party cookies: cookie that belongs to a domain other than the one on your URL bar

Common with pages containing content from other sides, such as banner ads

Because it belongs to the tracker’s domain

- … the cookie will be sent whenever you visit any other website that uses the same tracking server
- The website will see the same ID in the cookie so it can correlate what sites you visited

Most browsers allow you to block third-party cookies

- But trackers find ways to track you without using cookies
Third-party cookies: tracking cookies

• First-party cookies: sent by the domain of the page you loaded

• Third-party cookies:
  – sent by content in iFrames, media, and scripts (ads, Facebook, Google)

• Page = collection of first-party and third-party content
  – Third-party servers get their cookie whenever you visit a page they have a presence on
  – Record parent page, time of visit, user ID (if you have a cookie for one)
Cookies and privacy

• Cookies are essential but their use for tracking can also be invasive

• EU ePrivacy Directive
  – Receive consent from users
  – Provide info about each cookie
  – Store the user's consent
  – Provide the service without consent
  – Allow users to change their minds
Web-based Attacks
Malicious JavaScript & drive-by downloads

• Most web pages load JavaScript files
  – Malicious pages or iFrames may load malicious JavaScript
  – Visiting a malicious page that loads scripts is called a **drive-by download**

• What's the harm?
  – The script redirects you to another site & downloads an exploit kit
  – Exploit kit from the site probes, OS, browser, and other software to find vulnerabilities
  – Exploit kit downloads malware payload

• Other actions
  – Present ads, generate ad click-through, generate *likes* for social media content, mine cryptocurrency.
Cross-Site Request Forgery (CSRF)

A browser sends cookies for a site along with each server request

• If an attacker gets a user to access a site
  … the user’s cookies will be sent with that request

• If the cookies contain the user’s identity or session state
  – The attacker can create actions on behalf of the user

  http://mybank.com?action=transfer&to=attacker_account&amount=1000.00

• Plant the link in forums, email, ads, …

  <a href=http://mybank.com?action=transfer…>Important notice!</a>

The user sees: Important notice!
Some past CSRF attacks

- Create accounts on behalf of user
- Transfer funds out of user's accounts
- Add videos to Favorites
- Add attacker to a user's Friends or Family list
- Send messages as user, share video with user's contacts
- Subscribe a user to a channel
- Find email address of arbitrary user
- Add a movie to a user's queue
- Take over any Facebook user account
CSRF Defenses

• **For the user**
  – Log off sites when you’re done with them
  – Don’t allow browsers to store authentication cookies for sites

• **On the server**
  – The server can create a unique random token for every session
    • Token sent via hidden fields or headers
    • Validated with each page request from the user
  – Add a token that's a HMAC(request, timestamp)
  – Identify the origin of the request (*Origin* or *Referer* header)
Screen sharing attack

• HTML5 added a screen sharing API
• Normally: no cross-origin communication from client to server
• This is violated with the screen sharing API
  – If a frame is granted permission to take a screenshot, it can get a screenshot of the entire display (monitor, windows, browser)
  – Can also get screenshots within the user's browser without consent
• User might not be aware of the scope of screen sharing

http://dl.acm.org/citation.cfm?id=2650789
Input sanitization

Remember command injection attacks?

- Any user input must be parsed carefully
  
  `<script> var name = "untrusted_data"; </script>`

- Attacker can set `untrusted_data` to something like:
  
  ```html
  hi"; </script> <h1>Hey!</h1> <script> malicious code ...
  ```

- **Sanitization** should be used with any user input that may be part of
  
  - HTML
  - URL
  - JavaScript
  - CSS
SQL Injection

- Many web sites use a back-end database
- Queries may be constructed with user input

```javascript
username = getRequestString("uname");
pwd = getRequestString("passwd");

query = 'select * from Users where name = "' + username + '" and pwd = "' + pass + '"'
```

User Name: ramesh
Password: letmein

```sql
select * from Users where name = "ramesh" and pwd = "letmein"
```
SQL Injection

- Many web sites use a back-end database
- Queries may be constructed with user input

```python
username = getRequestString("uname");
pwd = getRequestString("passwd");

query = 'select * from Users where name = ''
        + username + '' and pwd = '' + pass + '''
```

```
select * from Users where
    name = '' or ''='' and pwd = '' or ''='' 
```

will return all rows from the Users table
Servers that use web form data in an OS command may be vulnerable to OS command injection
Arbitrary code execution vulnerability in bash

- Function export feature is buggy, allowing functions defined in one instance of bash to be available to other instances via environment variable lists

- **Web servers using CGI scripts** *(Common Gateway Interface)*
  - HTTP headers get converted to environment variables
  - Command gets executed by the shell via `system()`

  ```
  env x='()' { ;}; echo vulnerable' bash -c "echo this is a test"
  ```

- **Bogus function definition in bash**
  - Bash gets confused while parsing function definitions and executes the second part (“echo vulnerable”), which could invoke any operation

- **Discovered in 2014 … but existed since 1989!**
Cross-Site Scripting (XSS)

**Code injection attack**

- Allows attacker to execute JavaScript in a user’s browser
- Exploit vulnerability in a website the victim visits
  - Possible if the website includes user input in its pages
  - Example: user content in forums (feedback, postings)

- **Main types of attack:**
  - Stored XSS
  - Reflected XSS
Stored (Persistent) XSS

- Website stores user input and serves it back to other users at a later stage
- Victims do not have to click on a malicious link to run the payload
- Example: forum comments & postings

Item Description

```html
<h1>Official Radioactive Man collectible doll.</h1>
<p><strong>Brand new in box</strong>.</p>
<p style="color:red">Comment</p>

<script function loadurl(url){
    window.creator.location=url;
} </script>
```
Reflected XSS

- **Malicious code is not stored on the server**
  - It is returned as part of the HTTP response
  - **Attack string is part of the link**
    - HTTP query parameters used without sanitization and contain code
    - Distributed as links on spam email or web sites
    - Links look legitimate because the domain name is a valid, trusted server
    - Attacks may take advantage of existing cookies that will authenticate a user

- **Web application passes unvalidated input back to the client**
  The script in the link is returned in its original form inside the page & executed

www.mysite.com/login.asp?user=<script>malicious_code</script>
What's the harm?

- Access a user's cookies related to that website
- Hijack a session (using session authentication cookie)
- Create arbitrary HTTP requests with arbitrary content via XMLHttpRequest
- Make arbitrary modifications to the HTML document by modifying the DOM
- Install keyloggers
- Download malware – or run JavaScript ransomware
- Try phishing by manipulating the DOM and adding a fake login page or redirecting
XSS Defenses

• **Key defense is sanitizing ALL user input**
  – E.g., Django templates: `<b> hello, {{name}} </b>`
  – Use a less-expressive markup language for user input (e.g., markdown)

• **One of the problems in preventing XSS is character encoding**
  – Filters might check for “*<script>” but not “%3cscript%3e”*

• **Privilege separation**
  – Use a different domain for untrusted content
    • E.g., googleusercontent.com for static and semi-static content
    • Limits damage to main domain

• **Content Security Policy (CSP)**
  – Designed to prevent XSS & clickjacking
  – Allows website owners to **identify approved origins & types** of content the user can access
Deception: Typographic Attacks
Homograph attacks
Unicode confusion

Unicode represents virtually all the worlds glyphs

Some symbols look the same (or similar) but have different values

*Potential for deception*

They’re totally different to software but look the same to humans

/ = solidus (slash) = U+002F
/ = fraction slash = U+2044
/ = division slash = U+2215
\ = combining short solidus overlay = U+0337
/ = combining long solidus overlay = U+0338
/ = fullwidth solidus = U+FF0F

Yuck!
Paul ≠ Paul
Paul ≠ Paul

This is an uppercase i
This is a Greek υ (upsilon)
This is a Cyrillic a
This is a Greek Π
Homograph (Homoglyph) Attacks

• Some characters may look alike:
  – 1 (one), l (L), l (i)
  – 0 (zero), O

• Homograph attack = deception
  – paypal.com vs. paypaI.com (I instead of L)

• It got worse with internationalized domain names (IDN)
  – wikipedia.org
    • Cyrillic a (U+0430), e (U+435), p (U+0440)
    • Belarusian-Ukrainian i (U+0456)
  – Paypal
    • Cyrillic P, a, y, p, a; ASCII l

Check out the Homoglyph Attack Generator at https://www.irongeek.com/homoglyph-attack-generator.php

https://en.wikipedia.org/wiki/IDN_homograph_attack
URL Hijacking
URL Hijacking – Typosquatting

• Misspelled domain names
  – Confuse people into thinking the domain is something else
  – Hope that users make a typo when they type a URL

• Where do they take you?
  – Non-malicious content (usually)
    • Parked page (non-configured site) – accounts for most typosquats
    • Advertising
    • Brand-damaging content
    • Redirect to a different site (e.g., competitor)
    • The legitimate site
  – Malicious content (credential stealing, malware)

• Domain names can also be used in email-based phishing campaigns

www.googlec.om

Combosquatting
chase-bank.com
In October 2019, Digital Shadows’ Photon Research Team embarked on an adventure involving election typosquats that could potentially affect the presidential election and its candidates. If you haven’t read our original report, I’ll fill you in on a brief recap:

We detected over 550 typosquats for the 34 candidate- and election-related domains from open-source research. Not every single domain was interesting; most of the time, the typosquat was parked and not hosting content. Still, there were some worthwhile areas to dig into deeper: Misconfigured or illegitimate sites, non-malicious sites, and website redirects.
Examples

- winrde.com
- tulsi2020.co
- elizabethwarren.com
- donaldtrump.digital

https://www.cyberscoop.com/dhs-bulletin-typosquatting-2020-election-officials/
Examples

• 160 domains containing appleid registered between October 2019-April 2020
  – e.g., www-appleid[.]com
  – None owned by apple
  – 28% had malicious content

• Some famous examples:
  – MikeRoweSoft.com → targeted Microsoft (sort of)
  – hotmail.com variations → targeted hotmail.com
  – Fallwell.com → targeted Jerry Falwell (falwell.com)
  – PETA.org → targeted PETA

PayPal-related domains registered in June 2020

paypalticket91661[.]info
paypal-team[.]space
paypal-service[.]website
paypalticket91644[.]info
mypaypal[.]online
paypal-service[.]site
team-paypal[.]space
paypalticket91640[.]info
paypal-service[.]space
paypal-updateconfirmationsaccounts[.]com
paypal-updateconfirmationsaccount[.]com
paypal-support[.]space
team-paypal[.]website
paypalticket91645[.]info
paypalticket91642[.]info
paypalticket91664[.]info
paypal-updateconfirmationaccount[.]com

Typosquatting data feed: https://typosquatting.whoisxmlapi.com

http://www.circleid.com/posts/20200609-typosquatting-domains-every-appleid-owner-should-avoid/
Protection against typosquatting

- It's a race – legitimate domain owners race to get domain names before adversaries

- Example:
  A WHOIS search shows 455 Instagram-related names that belong to Facebook (who owns Instagram)

You can't register every possible domain!

• For example, IBM detected these registrations:
  – copyright-lnstagram[.]ml
  – instagram-verifybadge-support[.]ml
  – lnstagram-copyright-help-a3623vas336-va6f63a6ogsa824[.]ml

• The letter before "nstagram" in the first domain is an L, not an I

Legal remedies
  – 1999 U.S. Anticybersquatting Consumer Protection Act (ACPA)
    • Prove good-faith use of URL
    • Have a domain that is not similar to existing trademarks or brands
  – World Intellectual Property Organization (WIPO)
    • Petition the court that a domain is confusingly similar to yours
    • Holder had no rights to your brand
    • Site is used in bad faith

Typosquatting: Not just a URL problem!

- Package managers (PyPi, npm, ...) often contact public repositories of source
  - Anyone can add new packages

- 50% of packages are installed with admin privileges

- Attacker can create fake packages with similar names to legitimate ones and hope victims make grammatical mistakes or typos when installing

See https://incolumitas.com/2016/06/08/typosquatting-package-managers/
Typosquatting: Dependency Confusion

• Example – PayPal Node.js source code on GitHub
  – Meant for internal PayPal use (other internal package names published on Internet forums)
  – Package (package.json) contained a mix of public & private dependencies
    • Public ones are probably hosted from npm
    • Private ones are hosted internally

• An attacker (white hat – with permission!) uploaded malicious Node packages to npm with those private component names

• Software ended up loading these components instead of the private ones

• Apple, Shopify, Yelp, & Tesla were a few of the companies that were exposed!

See https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610
Image-based attacks & tracking
Clickjacking

- Attacker overlays an image to trick a user to clicking a button or link
- User sees this
  ![FREE iPad](image)
  ![Click Here](image)
- Not realizing there’s an invisible frame over the image
- Clicking there could generate a Facebook like
  ... or download malware ... or change security settings for a plugin
- Defense
  - JavaScript in the legitimate code to check that it’s the top layer
    ```javascript
    window.self == window.top
    ```
  - Set `X-Frame-Options` to not allow frames from other domains
HTML image tags

- Images are static content with no authority
- Any problems with images?

```html
<img src="http://pk.org/images/balloons.jpg" height="300" width="400"/>
```
<img src="http://pk.org/images/balloons.jpg?extra_information" height="300" width="400"/>

- URL may pass arguments
  - Communicate with other sites
- Hide resulting image
  <img src="..." height="1" width="1"/>
- Request will send cookies
  - server response can set them

Almost 25% of mail messages contain a tracking link
Of popular sending domains, about 50% perform tracking
Ad retargeting

- **Origin** = www.facebook.com

- **Accessing the web page with this pixel will**
  - Contact Facebook to load the image
  - Send Facebook cookies from your browser to Facebook
  - Enable Facebook to record the fact that you visited this page
Deception via image tags

Social engineering: add logos to fool a user

- Impersonate site
- Impersonate credentials
Mouseover on a link shows link target

https://www.paypal.com/signin/

Trivial to spoof with JavaScript

```
<a href="http://www.paypal.com/signin"
    onclick="this.href='http://www.evil.com/';">PayPal</a>
```
The situation is not good

• HTML, JavaScript, and CSS continue to evolve
• All have become incredibly complex
• Web apps themselves can be incredibly complex, hence buggy
• Web browsers are forgiving
  – You don’t see errors
  – They try to correct syntax problems and guess what the author meant
  – Usually, *something* gets rendered
The End