

Computer Security

14. Web Security

Paul Krzyzanowski

Rutgers University

Spring 2018

Original Browser

- Static content on clients
- Servers were responsible for dynamic parts
- Security attacks were focused on servers
 - Malformed URLs, buffer overflows, root paths, unicode attacks

Today's Browsers

Complex!

- **JavaScript** – allows code execution
- **Document Object Model (DOM)** – 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**
- **NaCl** – run native code inside a browser (sandboxed)

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

Multiple sources

- Most desktop & mobile apps come from one place
 - They may use external libraries, but those are linked in and tested
- Web apps usually have components from different places
- E.g., www.cnn.com has
 - **Fonts** from cdn.cnn.com
 - **Images** from turner.com, outbrain.com, bleacherreport.net, chartbeat.net
 - **Scripts** from amazon-adsystem.com, rubiconproject.com, bing.com, krx.net, gigya.com, krx.net, livefyre.com, fyre.co, optimizely.com, facebook.net, cnn.com, criteo.com, outbrain.com, sharethrough.com, doubleclick.net, googletagservices.com, ugdturner.com
 - **XMLHttpRequests** from zone-manager.izi, optimizely.com, chartbeat.com, cnn.io, rubiconproject.com
 - **Other content** from scorecardresearch.com, imnworldwide.com, facebook.com

What should code on a page have access to?

- Can analytics code access JavaScript state from a script from jQuery.com on the same page?
 - Scripts are from different places ... but the page author selected them
- Can analytics scripts interact with event handlers?
- How about embedded frames?

Same-origin Policy

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
 - <http://www.poopybrain.com/419/test.html>
 - <http://www.poopybrain.com/index.html>
- Different origin
 - <https://www.poopybrain.com/index.html> – different URI scheme (https)
 - <http://www.poopybrain.com:8080/index.html> – different port
 - <http://poopybrain.com/index.html> – different host

Ideas behind the same-origin policy

- Each origin has client-side resources
 - **Cookies**: simple way to implement state
 - 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
- Each frame is assigned the origin of its URL
- 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 two independent frames to communicate
- Both sides have to opt in

Passive content has no authority

Makes sense ... but why does it matter?

Usually no ... but ...

MIME sniffing attack

- Chance of security problems if browser parses object incorrectly
- Old versions of IE would examine leading bytes of object to fix wrong file types provided by the user
- Suppose a page contained passive content from an untrusted site
- Attacker could add HTML & JavaScript to the content
 - IE would reclassify the content

Cross-origin weirdness

- **Images**

- A frame can load images from anywhere
- 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 inside 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)

- A page can contain content from multiple origins
 - Images, CSS, scripts, iframes, videos
- XMLHttpRequests are not permitted
 - **CORS** – allows servers to define allowable origins
 - Example, a server at `service.example.com` may respond with
`Access-Control-Allow-Origin: http://www.example.com`
 - Stating that it will allow treating `www.example.com` as the same origin

Cookies

- Cookies are identified with a domain & a path
`pk.org/419`

All paths in the domain have access to the cookie

- Whoever sets the cookie chooses what domain & paths looks like
 - JavaScript can set
`document.cookie = "username=paul";`
 - Server can set cookies by sending them in the HTTP header
`Set-Cookie: username=paul`

When a browser generates an HTTP request
it sends all matching cookies

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
 - Sent in a `Set-Cookie` HTTP response header
- **Secure** flag: send the cookie only over https
 - `Set-Cookie: username=paul; path=/; HttpOnly; Secure`

Cross-Site Request Forgery (XSRF)

- A browser sends cookies for a site along with a 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
- Planting the link
 - Forums or spam
 - http://mybank.com/?action=transfer&amount=100000&to=attacker_account

Cross-Site Request Forgery (XSRF)

Defenses

- Validate the *referrer* header at the server
- Require unique tokens per request
 - Add randomness to the URL that attackers will not be able to guess
 - E.g., legitimate server can set tokens via hidden fields instead of cookies
- **Default-deny** browser policy for cross-site requests (but may interfere with legitimate uses)

Clickjacking

- Attacker overlays an image to trick a user to clicking a button or link
- User sees this



- 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 the Flash plugin
- Defense
 - JavaScript in the legitimate code to check that it's the top layer
`window.self == window.top`
 - Set `X-Frame-Options` to not allow frames from other domains

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>

<http://mews.sv.cmu.edu/papers/oakland-14.pdf>

Input sanitization

- Remember SQL injection attacks?
- Any user input must be parsed carefully

```
<script> var x = "untrusted_data"; </script>
```

- Attacker can set `untrusted_data` to something like:

```
hi"; </script> <h1> Hey, some text! </h1> <script> malicious code... </script>
```

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

Shellshock attack

- Discovered in 2014 Existed since 1989!
- **Privilege escalation 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

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)
- What's the harm?
 - Access cookies related to that website
 - Hijack a session
 - 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

Types of XSS attacks

- **Reflected XSS**

- Malicious code is not stored anywhere
 - It is returned as part of the HTTP response
 - Only impacts users who open a malicious link or third-party web page
 - **Attack string is part of the link**
- **Web application passes unvalidated input back to the client**
 - The script is in the link and is returned in its original form & executed

[www.mysite.com/login.asp?user=<script>malicious_code\(...\) </script>](http://www.mysite.com/login.asp?user=<script>malicious_code(...) </script>)

- **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

XSS Defense

- One of the problems in preventing XSS is **character encoding**
 - Filters might check for "<script>" but not "%3cscript%3e"
- Key defense is **sanitizing ALL user input**
 - E.g., Django templates: hello, {{name}}
- Use a less-expressive markup language for user input
 - E.g., markdown
- **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** of content & **types** of content

SQL Injection & pathnames

We examined these earlier

SQL Injection

- Many web sites use a back-end database
- Links contain queries mixed with user input

```
query = "select * from table where user=" + username
```

Pathnames

- Escape the HTML directory

```
//mysite/images/../../../../etc/shadow
```


GIFAR attack

- Java applets are sent as JAR files
 - This is just a zip format
 - Header is stored at the end of the file
- GIF files are images
 - Header is stored at the beginning of the file
- We can combine the two files: gif + jar
- GIFAR attack
 - Submit a GIFAR file ([myimage.gif](#)) to a site that only allows image uploads
 - Use XSS to inject `<applet archive:"myimage.gif">`
 - Code will run in the context of the server
 - Attacker gets to run with the authority of the origin (server)

Network 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 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 victim's IP address
 - Web site can now fetch a new object via AJAX
 - Web browser thinks request goes to an external site
 - Really, it goes to a server in the victim's network
 - The attacker is now accessing data within the victim's servers and can send data back to an attacker's site

Network addresses

- Solution – no foolproof solutions
 - Don't allow DNS resolutions to return internal addresses
 - Force longer TTL

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