Extending the browser to secure applications

Highlights from W3C WebAppSec Group

Deian Stefan

Modern web apps have many moving pieces & parties

- Application code & content itself
- User provided content (e.g., mail, comments)
- 3rd party libraries (e.g., jQuery)
- 3rd party content (e.g., ads, widgets)
- 3rd party hosting (e.g., CDNs)

Hard to secure

(especially if you’re not starting from scratch)

Browsers can help

when extended with security mechanisms

Some work being done in the W3C WebAppSec group

- Secure contexts
- Mixed content
- Upgrade insecure requests
- SRI
- Referrer policy
- CSP
- Suborigins
- COWL
- Credential mgmnt
- UI Security

Moving the web to HTTPS

Secure contexts: definitional

- Why? Expose potentially dangerous features (e.g., Service workers) only if context is secure
- Nutshell: content is loaded over HTTPS and doesn't have HTTP opener

Moving the web to HTTPS

Mixed content: making the 🔒 great again

- **Goals:** Authentication, encryption, data integrity
- **Nutshell:** Disallow loading content via HTTP in secure contexts

```
Content-Security-Policy: block-all-mixed-content
```

Moving the web to HTTPS

- Transitioning to HTTPS is painful
  - Don't have too much control over 3rd party hosts
  - Often have hardcoded links in content
- Browsers to the rescue:
  - Don't block images, videos or audio by default. Use CSP directives to block (e.g., `img-src https:`)
  - `Content-Security-Policy: upgrade-insecure-requests`

HTTPS is not enough

- Authenticates connection but not content
  - Protects against (coffee shop) network attacker
- Don’t know if content you’re loading is the one you meant to load
  - Malicious/compromised CDN can serve bad content

Subresource integrity

- Page author specifies hash of (sub)resource they are loading; browser checks integrity
  - E.g., integrity for link elements

```
<link rel="stylesheet" href="https://site53.cdn.net/style.css"
      integrity="sha256-SDfwewFAE...wefjijfE">
```

  - E.g., integrity for scripts

```
<script src="https://code.jquery.com/jquery-1.10.2.min.js"
   integrity="sha256-C6CB9UYIS9UJeqinPHWTHVqh/E1uhG5Tw+Y5qFQmYg=">
```

Subresource integrity

- Limitations
  - Only scripts and stylesheets supported in current version
  - Roadmap: downloads and other elements
- Challenges
  - May dynamically load scripts; too expensive to load all hashes ahead of time
  - Roadmap: support signature-like scheme

Some work being done in the W3C WebAppSec group

- Secure contexts
- Mixed content
- Upgrade insecure requests
- SRI
- Referrer policy
- CSP
- Suborigins
- COWL
- Credential mgmnt
- UI Security

CSP: Content Security Policy

- Motivation: prevent or limit damage of XSS
- Idea: restrict resource loading to a white list
  - Only load content from origins you trust
  - Only leak data to these origins (in case of XSS)
  - Limit inline scripts to whitelist via hash/nonce
- Example: allow loads from CDN, but disallow frames and plugins

```
Content-Security-Policy: default-src https://cdn.example.net;
                         child-src 'none'; object-src 'none'
```

Example directives

- connect-src: limits the origins you can XHR to
- font-src: where to fetch web fonts form
- form-action: where forms can be submitted
- child-src: where to load frames/workers from
- frame-ancestors: sources that can embed this page
- default-src: default whitelist

CSP: Content Security Policy

- Limitations
  - Leaks via navigation, postMessage, DNS prefetching, etc. are easy
  - Roadmap: new directives to cover some of these
- Challenges
  - Hard to know if your CSP policy will break libraries
  - Roadmap: CSP Embedded Enforcement
  - Whitelisting origins is too coarse-grained

Why are origins too coarse grained?

- Consider whitelisting your CDN
  - Attacker can put malicious script on CDN and escalate simple XSS to arbitrary code
- New extension to CSP: strict-dynamic (WIP)
  - Idea: use nonce and hash to whitelist scripts, ignore origins

```
Content-Security-Policy: script-src 'nonce-abcdefg' 'strict-dynamic'
```

CSP is not enough

- Multiple web apps are hosted on same physical origin
  - E.g., google.com/maps and google.com/mail
    - XSS on one logical site can easily affect the other (own maps, you own mail)
- ~~Solution: maps.google.com mail.google.com~~
  - Actually hard to deploy
- Solution: suborigins

Suborigins

- Goal: provide simple, backwards-compatible privilege separation mechanism
- Idea: extend notion of origin to (origin, namespace)
  - E.g., (google.com, maps)
  - Treat suborigin different from origin: as if diff origin

Suborigins

- Deployment challenges
  - Can't access cookie jar of origin
  - postMessage origin comparison is suborigin unaware
- Roadmap: unsafe-* directives to make it easier to deploy suborigins on existing sites

Still not enough

- Can’t actually drop privileges, so scripts in suborigin have the privileges of that origin
  - E.g., may want to load page and then drop privileges so XSS won’t leak/corrupt user data
- Even when combined with CSP: have no control of what code can do with sensitive data

COWL: Confinement with Origin Web Labels

- Goals:
  - Allow developers to privilege separate their app
  - Allow developers to run their own code with least privilege
  - Allow developers to confine 3rd party code by restricting what it can do with data it shares

COWL framework

- Policy specification
  - Labels: generalization of origin to conjunctions and disjunctions of origins and tags
  - Privileges: makes authority of pages explicit with labels
- Policy enforcement
  - Enforcement of policy end-to-end via confinement

Labels

- Specifying policy in the browser
  - Labels are associated with contexts
  - Labels can be associated with clonable objects (use with postMessage and XHR)
- **Specifying policy server-side**
  - Labels can be associated with XHR responses

Privileges

- Explicit control over page privilege with JS
  - Page has default privilege = its origin
- **Controlling default page privilege**
  - E.g., to ensure page doesn't have authority of origin

Confinement enforcement

- In browser and browser-server communication
  - Restrict postMessage and requrest to prevent leaking browser data
  - Restrict responses if label of response is more sensitive than context label (non labeled-json request)

COWL use cases

- Confining untrusted third-party services
- Sharing data with third-party mashups
- Content isolation (via privilege separation)
  - Similar to suborigin use case
- Running content with least privileges

COWL limitations & challenges

- Limitations
  - Similar to suborigins: COWL-confined iframe doesn't have access to storage
  - Disallowed APIs: web sockets, service workers
  - Roadmap: add labeled storage and enable other APIs
- Challenges
  - Need to compartmentalize app into many least-privileged iframes to get most out of confinement

Stepping back

- Secure contexts
- Mixed content
- Upgrade insecure requests
- SRI
- Referrer policy
- CSP
- Suborigins
- COWL
- Credential mgmnt
- UI Security

Stepping back

New mechanisms can make it easier to secure new and existing web apps: hack the browser!