An Introduction to Content Security Policy

By Mike West

Published: June 15th, 2012
Updated: September 30th, 2016
Comments: 10

The web's security model is rooted in the *same origin policy*. Code from `https://mybank.com` should only have access to `https://mybank.com`'s data, and `https://evil.example.com` should certainly never be allowed access. Each origin is kept isolated from the rest of the web, giving developers a safe sandbox in which to build and play. In theory, this is perfectly brilliant. In practice, attackers have found clever ways to subvert the system.

Cross-site scripting (XSS) attacks, for example, bypass the same origin policy by tricking a site into delivering malicious code along with the intended content. This is a huge problem, as browsers trust all of the code that shows up on a page as being legitimately part of that page's security origin. The XSS Cheat Sheet is an old but representative cross-section of the methods an attacker might use to violate this trust by injecting malicious code. If an attacker successfully injects *any* code at all, it's pretty much game over: user session data is compromised and information that should be kept secret is exfiltrated to The Bad Guys. We'd obviously like to prevent that if possible.

This tutorial highlights one promising new defense that can significantly reduce the risk and impact of XSS attacks in modern browsers: Content Security Policy (CSP).

Source Whitelists

The core issue exploited by XSS attacks is the browser's inability to distinguish between script that's intended to be part of your application, and script that's been maliciously injected by a third-party. For example, the Google +1 button at the top of this article loads and executes code from `https://apis.google.com/js/plusone.js` in the context of this page's origin. We trust that code, but we can't expect the browser to figure out on it's own that code from `apis.google.com` is awesome, while code from `apis.evil.example.com` probably isn't. The browser happily downloads and executes any code a page requests, regardless of source.

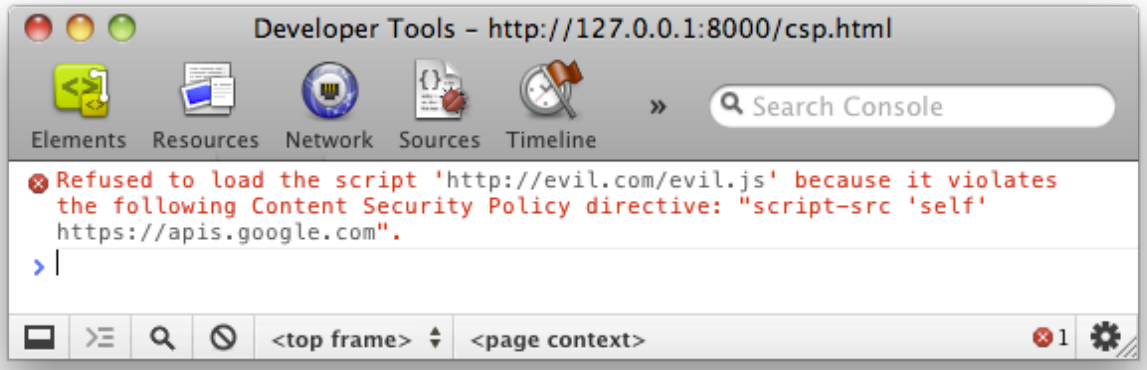
Instead of blindly trusting *everything* that a server delivers, CSP defines the `Content-Security-Policy` HTTP header that allows you to create a whitelist of sources of trusted content, and instructs the browser to only execute or render resources from those sources. Even if an attacker can find a hole through which to inject script, the script won't match the whitelist, and therefore won't be executed.

Since we trust `apis.google.com` to deliver valid code, and we trust ourselves to do the same, let's define a policy that only allows script to execute when it comes from one of those two sources:

```
Content-Security-Policy: script-src 'self' https://apis.google.com
```

Simple, right? As you probably guessed, **`script-src`** is a directive that controls a set of script-related privileges for a specific page. We've specified `'self'` as one valid source of script, and `https://apis.google.com` as another. The browser will dutifully download and execute JavaScript from `apis.google.com` over HTTPS, as well as from the current page's origin.

With this policy defined, the browser will simply throw an error instead of loading script from any other source. When a clever attacker does manage to inject code into your site, she'll run headlong into an error message, rather than the success she was expecting:



Policy applies to a wide variety of resources

While script resources are the most obvious security risks, CSP provides a rich set of policy directives that enable fairly granular control over the resources that a page is allowed to load. You've already seen `script-src`, so the concept should be clear. Let's quickly walk through the rest of the resource directives:

- **`base-uri`** restricts the URLs that can appear in a page's `<base>` element.

- `child-src` lists the URLs for workers and embedded frame contents. For example: `child-src https://youtube.com` would enable embedding videos from YouTube but not from other origins. Use this in place of the deprecated `frame-src` directive.
- `connect-src` limits the origins to which you can connect (via XHR, WebSockets, and EventSource).
- `font-src` specifies the origins that can serve web fonts. Google's Web Fonts could be enabled via `font-src https://themes.googleusercontent.com`
- `form-action` lists valid endpoints for submission from `<form>` tags.
- `frame-ancestors` specifies the sources that can embed the current page. This directive applies to `<frame>`, `<iframe>`, `<embed>`, and `<applet>` tags. This directive can't be used in `<meta>` tags and applies only to non-HTML resources.
- `frame-src` deprecated. Use `child-src` instead.
- `img-src` defines the origins from which images can be loaded.
- `media-src` restricts the origins allowed to deliver video and audio.
- `object-src` allows control over Flash and other plugins.
- `plugin-types` limits the kinds of plugins a page may invoke.
- `report-uri` specifies a URL where a browser will send reports when a content security policy is violated. This directive can't be used in `<meta>` tags.
- `style-src` is `script-src`'s counterpart for stylesheets.
- `upgrade-insecure-requests` Instructs user agents to rewrite URL schemes, changing HTTP to HTTPS. This directive is for web sites with large numbers of old URLs that need to be rewritten.

By default, directives are wide open. If you don't set a specific policy for a directive, let's say `font-src`, then that directive behaves by default as though you'd specified `*` as the valid source (e.g. you could load fonts from everywhere, without restriction).

You can override this default behavior by specifying a `default-src` directive. This directive, as you might suspect, defines the defaults for most directives you leave unspecified. Generally, this applies to any directive that ends with `-src`. If `default-src` is set to `https://example.com`, and you fail to specify a `font-src` directive, then you can load fonts from `https://example.com`, and nowhere else. We specified only `script-src` in our earlier examples, which means that images, fonts, and so on can be loaded from any origin.

The following directives don't use default-src as a fallback. Remember that failing to set them is the same as allowing anything.

- `base-uri`
- `form-action`

- `frame-ancestors`
- `plugin-types`
- `report-uri`
- `sandbox`

You can use as many or as few of these directives as makes sense for your specific application, simply listing each in the HTTP header, separating directives with semicolons. You'll want to make sure that you list *all* required resources of a specific type in a *single* directive. If wrote something like

```
script-src https://host1.com; script-src https://host2.com
```

the second directive would simply be ignored. Something like the following would correctly specify both origins as valid.

```
script-src https://host1.com https://host2.com
```

If, for example, you have an application that loads all of its resources from a content delivery network (say, `https://cdn.example.net`), and know that you don't need framed content or any plugins at all, then your policy might look like the following:

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

Implementation Details

You will see `X-WebKit-CSP` and `X-Content-Security-Policy` headers in various tutorials on the web. Going forward, you can and should ignore these prefixed headers. Modern browsers (with the exception of IE) support the unprefixed `Content-Security-Policy` header. That's the header you should use.

Regardless of the header you use, policy is defined on a page-by-page basis: you'll need to send the HTTP header along with every response that you'd like to ensure is protected. This provides a lot of flexibility, as you can fine-tune the policy for specific pages based on their specific needs. Perhaps one set of pages in your site has a +1 button, while others don't: you could allow the button code to be loaded only when necessary.

The source list in each directive is fairly flexible. You can specify sources by scheme (`data:`, `https:`), or ranging in specificity from hostname-only (`example.com`, which

matches any origin on that host: any scheme, any port) to a fully qualified URI (`https://example.com:443`, which matches only HTTPS, only `example.com`, and only port 443). Wildcards are accepted, but only as a scheme, a port, or in the leftmost position of the hostname: `*://*.example.com:*` would match all subdomains of `example.com` (but *not* `example.com` itself), using any scheme, on any port.

Four keywords are also accepted in the source list:

- `'none'`, as you might expect, matches nothing.
- `'self'` matches the current origin, but not its subdomains.
- `'unsafe-inline'` allows inline JavaScript and CSS (we'll touch on this in more detail in a bit).
- `'unsafe-eval'` allows text-to-JavaScript mechanisms like `eval` (we'll get to this too).

These keywords require single-quotes. `script-src 'self'` (with quotes) authorizes the execution of JavaScript from the current host. `script-src self` (no quotes) allows JavaScript from a server named "`self`" (and *not* from the current host), which probably isn't what you meant.

Sandboxing

There's one more directive worth talking about: `sandbox`. It's a bit different than the others we've looked at, as is places restrictions on actions the page can take, rather than on resources that the page can load. If the `sandbox` directive is present, the page will be treated as though it was loaded inside of an `iframe` with a `sandbox` attribute. This can have a wide range of effects on the page: forcing the page into a unique origin, and preventing form submission, among others. It's a bit beyond the scope of this article, but you can find full details on valid sandboxing attributes in the "sandboxing flag set" section of the HTML5 spec.

The meta Tag

CSPs preferred delivery mechanism is an HTTP header. It can be useful, however, to set a policy on a page directly in the markup. Do that using a meta tag with an http-equiv attribute:

```
<meta http-equiv="Content-Security-Policy" content="default-src
https://cdn.example.net; child-src 'none'; object-src 'none'">
```

This can't be used for frame-ancestors, report-uri, or sandbox.

Inline Code Considered Harmful

It should be clear that CSP is based on whitelisting origins, as that's an unambiguous way of instructing the browser to treat specific sets of resources as acceptable and to reject the rest. Origin-based whitelisting doesn't, however, solve the biggest threat posed by XSS attacks: inline script injection. If an attacker can inject a script tag that directly contains some malicious payload (`<script>sendMyDataToEvilDotCom();</script>`), the browser has no mechanism by which to distinguish it from a legitimate inline script tag. CSP solves this problem by banning inline script entirely: it's the only way to be sure.

This ban includes not only scripts embedded directly in `script` tags, but also inline event handlers and `javascript:` URLs. You'll need to move the content of `script` tags into an external file, and replace `javascript:` URLs and `<a ... onclick="[JAVASCRIPT]">` with appropriate `addEventListener` calls. For example, you might rewrite the following from:

```
<script>
  function doAmazingThings() {
    alert('YOU AM AMAZING!');
  }
</script>
<button onclick='doAmazingThings();'>Am I amazing?</button>
```

to something more like:

```
<!-- amazing.html -->
<script src='amazing.js'></script>
<button id='amazing'>Am I amazing?</button>
```

```
// amazing.js
function doAmazingThings() {
  alert('YOU AM AMAZING!');
}
document.addEventListener('DOMContentReady', function () {
  document.getElementById('amazing')
          .addEventListener('click', doAmazingThings);
});
```

The rewritten code has a number of advantages above and beyond working well with CSP; it's already best practice, regardless of your use of CSP. Inline JavaScript mixes structure and behavior in exactly the way you shouldn't. External resources are easier for browsers

to cache, more understandable for developers, and conducive to compilation and minification. You'll write better code if you do the work to move code into external resources.

Inline style is treated in the same way: both the `style` attribute and `style` tags should be consolidated into external stylesheets to protect against a variety of surprisingly clever data exfiltration methods that CSS enables.

If you really, absolutely must have inline script and style, you can enable it by adding `'unsafe-inline'` as an allowed source in a `script-src` or `style-src` directive. You can also use a nonce or a has (see below). But please don't. Banning inline script is the biggest security win CSP provides, and banning inline style likewise hardens your application. It's a little bit of effort up front to ensure that things work correctly after moving all the code out-of-line, but that's a tradeoff that's well worth making.

If You Absolutely Must Use It...

CSP Level 2 offers backward compatibility for inline scripts by allowing you to whitelist specific inline scripts using either a cryptographic nonce (number used once) or a hash. Although this may be cumbersome in practice, it is useful in a pinch.

To use a nonce, give your script tag a nonce attribute. Its value must match one in the list of trusted sources. For example:

```
<script nonce=EDNnf03nceIOfn39fn3e9h3sdfa>
  // Some inline code I can't remove yet, but need to asap.
</script>
```

Now, add the nonce to your script-src directive appended to the nonce- keyword.

```
Content-Security-Policy: script-src 'nonce-
EDNnf03nceIOfn39fn3e9h3sdfa'
```

Remember that nonces must be regenerated for every page request and they must be unguessable.

Hashes work in much the same way. Instead of adding code to the script tag, create a SHA hash of the script itself and add it to the script-src directive. For example, let's say your page contained this:

```
<script>alert('Hello, world.');</script>
```

Your policy would contain this:

```
Content-Security-Policy: script-src 'sha256-
qznLcsROx4GACP2dm0UCKCzCG-HiZ1guq6ZZDob_Tng='
```

There are a few things to note here. The `sha*-` prefix specifies the algorithm used to generate the hash. In the example above, sha256- is used. CSP also supports sha384- and sha512-. When generating the hash do not include the `<script>` tags. Also capitalization and whitespace matter, including leading or trailing whitespace.

A Google search on generating SHA hashes will lead you to solutions in any number of languages. Using Chrome 40 or later you can open DevTools then reload your page. The Console tab will contain error messages with the correct sha256 hash for each of your inline scripts.

Eval Too

Even when an attacker can't inject script directly, she might be able to trick your application into converting otherwise inert text into executable JavaScript and executing it on her behalf. `eval()`, `new Function()`, `setTimeout([string], ...)`, and `setInterval([string], ...)` are all vectors through which injected text might end up executing something unexpectedly malicious. CSP's default response to this risk is, unsurprisingly, to block all of these vectors completely.

This has more than a few impacts on the way you build applications:

- Parse JSON via the built-in `JSON.parse`, rather than relying on `eval`. Native JSON operations are available in every browser since IE8, and they're completely safe.
- Rewrite any `setTimeout` or `setInterval` calls you're currently making with inline functions rather than strings. For example:

```
setTimeout("document.querySelector('a').style.display =
'none';", 10);
```

would be better written as:

```
setTimeout(function () {
  document.querySelector('a').style.display = 'none';
}, 10);
```

- Avoid inline templating at runtime: Many templating libraries use `new Function()` liberally to speed up template generation at runtime. It's a nifty application of dynamic programming, but comes at the risk of evaluating malicious text. Some frameworks support CSP out of the box, falling back to a robust parser in the absence of `eval`; AngularJS's ng-csp directive is a good example of this.

You're even better off, however, if your templating language of choice offers precompilation (Handlebars does, for instance). Precompiling your templates can make the user experience even faster than the fastest runtime implementation, and it's safer too. Win, win! If eval and its text-to-JavaScript brethren are completely essential to your application, you can enable them by adding `'unsafe-eval'` as an allowed source in a `script-src` directive. But, again, please don't. Banning the ability to execute strings makes it much more difficult for an attacker to execute unauthorized code on your site.

Reporting

CSP's ability to block untrusted resources client-side is a huge win for your users, but it would be quite helpful indeed to get some sort of notification sent back to the server so that you can identify and squash any bugs that allow malicious injection in the first place. To this end, you can instruct the browser to `POST` JSON-formatted violation reports to a location specified in a `report-uri` directive.

```
Content-Security-Policy: default-src 'self'; ...; report-uri
/my_amazing_csp_report_parser;
```

Those reports will look something like the following:

```
{
  "csp-report": {
    "document-uri": "http://example.org/page.html",
    "referrer": "http://evil.example.com/",
    "blocked-uri": "http://evil.example.com/evil.js",
    "violated-directive": "script-src 'self'
https://apis.google.com",
    "original-policy": "script-src 'self' https://apis.google.com;
report-uri http://example.org/my_amazing_csp_report_parser"
  }
}
```

It contains a good chunk of information that will help you track down the specific cause of the violation, including the page on which the violation occurred (`document-uri`), that

page's referrer (referrer, note that the key is *not* misspelled), the resource that violated the page's policy (`blocked-uri`), the specific directive it violated (`violated-directive`), and the page's complete policy (`original-policy`).

Report-Only

If you're just starting out with CSP, it makes sense to evaluate the current state of your application before rolling out a draconian policy to your users. As a stepping stone to a complete deployment, you can ask the browser to monitor a policy, reporting violations, but not enforcing the restrictions. Instead of sending a `Content-Security-Policy` header, send a `Content-Security-Policy-Report-Only` header.

```
Content-Security-Policy-Report-Only: default-src 'self'; ...;
report-uri /my_amazing_csp_report_parser;
```

The policy specified in report-only mode won't block restricted resources, but it will send violation reports to the location you specify. You can even send *both* headers, enforcing one policy while monitoring another. This is a great way to evaluate the effect of changes to your application's CSP: turn on reporting for a new policy, monitor the violation reports and fix any bugs that turn up, then start enforcing the new policy once you're satisfied with its effect.

Real World Usage

CSP 1 is quite usable in Chrome, Safari, and Firefox, and has (very) limited support in IE 10. You can view specifics at caniuse.com. CSP Level 2 was available in Chrome with version 40. Massive sites like Twitter and Facebook have deployed the header (Twitter's case study is worth a read), and the standard is very much ready for you to start deploying on your own sites.

The first step towards crafting a policy for your application is to evaluate the resources you're actually loading. Once you think you have a handle on how things are put together in your app, set up a policy based on those requirements. Let's walk through a few common use-cases, and determine how we'd best be able to support them within the protective confines of CSP:

Use Case #1: Social media widgets

- Google's +1 button includes script from `https://apis.google.com`, and embeds an `iframe` from `https://plusone.google.com`. You'll need a policy that includes both

these origins in order to embed the button. A minimal policy would be `script-src https://apis.google.com; child-src https://plusone.google.com`. You'll also need to ensure that the snippet of JavaScript that Google provides is pulled out into an external JavaScript file. If you had an existing policy using child-src, you would need to change it to child-src.

- Facebook's Like button has a number of implementation options. I'd recommend sticking with the `iframe` version, as it's safely sandboxed from the rest of your site. That would require a `child-src https://facebook.com` directive to function properly. Note that, by default, the `iframe` code Facebook provides loads a relative URL, `//facebook.com`. Please change that to explicitly specify HTTPS: `https://facebook.com`. There's no reason to use HTTP if you don't have to.
- Twitter's Tweet button relies on access to a script and frame, both hosted at `https://platform.twitter.com` (Twitter likewise provides a relative URL by default: please edit the code to specify HTTPS when copy/pasting it locally). You'll be all set with `script-src https://platform.twitter.com; child-src https://platform.twitter.com`, as long as you move the JavaScript snippet Twitter provides out into an external JavaScript file.
- Other platforms will have similar requirements, and can be addressed similarly. I'd suggest just setting a `default-src` of `'none'`, and watching your console to determine which resources you'll need to enable to make the widgets work.

Including multiple widgets is straightforward: simply combine the policy directives, remembering to merge all resources of a single type into a single directive. If you wanted all three, the policy would look like:

```
script-src https://apis.google.com https://platform.twitter.com;
child-src https://plusone.google.com https://facebook.com
https://platform.twitter.com
```

Use Case #2: Lockdown

Assume for a moment that you run a banking site, and want to make very sure that only those resources you've written yourself can be loaded. In this scenario, start with a default policy that blocks absolutely everything (`default-src 'none'`), and build up from there.

Let's say the bank loads all images, style, and script from a CDN at `https://cdn.mybank.net`, and connects via XHR to `https://api.mybank.com/` to pull various bits of data down. Frames are used, but only for pages local to the site (no third-

party origins). There's no Flash on the site, no fonts, no nothing. The most restrictive CSP header that we could send in this scenario is:

```
Content-Security-Policy: default-src 'none'; script-src
https://cdn.mybank.net; style-src https://cdn.mybank.net; img-src
https://cdn.mybank.net; connect-src https://api.mybank.com; child-
src 'self'
```

Use Case #3: SSL Only

A wedding-ring discussion forum admin wants to ensure that all resources are only loaded via secure channels, but doesn't really write much code; rewriting large chunks of the third-party forum software that's filled to the brim with inline script and style is beyond his abilities. The following policy would be effective:

```
Content-Security-Policy: default-src https:; script-src https:
'unsafe-inline'; style-src https: 'unsafe-inline'
```

Even though `https:` was specified in `default-src`, the script and style directives don't automatically inherit that source. Each directive overwrites the default completely for that specific type of resource.

The Future

Content Security Policy Level 2 is a Candidate Recommendation. The W3Cs Web Application Security Working Group isn't lounging around, patting itself on the back; work has already begun on the specifications next iteration. The next version is already under active development.

If you're interested in the discussion around these upcoming features, skim the public-webappsec@ mailing list archives, or join in yourself.