Content Security Policy - An Introduction

November 27, 2014

Content Security Policy is delivered via a HTTP response header, much like HSTS, and defines approved sources of content that the browser may load. It can be an effective countermeasure to Cross Site Scripting (XSS) attacks and is also widely supported and usually easily deployed.

Why do we need CSP?

When your browser loaded this page, it loaded a lot of other assets along with it. There are stylesheets and fonts to load along with quite a few javascript files. One for the Disqus comment system, one for Google Analytics, my social sharing buttons down at the bottom and a few more for good measure too. Your browser loads these assets because it is instructed to do so by the source code of the page. It has no way of knowing whether or not any of those files should or should not be loaded. An attacker could have placed a specially crafted comment in the comment section to load some malicious javascript from a 3rd party domain and this would also be loaded by your browser because it was included along with the page. Your browser has no

reason to not trust the content from `nastyhackers.com` and no way of knowing the content is malicious. This is where CSP comes in.

Whitelisting Approved Sources

A CSP header allows you to define approved sources for content on your site that the browser can load. By specifying only those sources that you wish the browser to load content from, you can protect your visitors from a whole range of issues. Here is a basic CSP response header.

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

Going back to the example above of an attacker using a specially crafted comment to load javascript from another domain, this CSP header would prevent the browser loading content from `nastyhackers.com`. The `script-src` directive specifies the whitelist of sources that the browser may load scripts from. Using the `'self'` keyword is easier than specifying my whole domain and makes the policy a little easier to read once it starts growing. Because the domain `nastyhackers.com` isn't in the script whitelist, the browser will not load the script content from `nastyhackers.com`.

What can we protect?

CSP comes with a wide range of directives that can be used to enforce policy across a whole load of content and circumstances. Here is a list of all of the directives that are available, along with a brief description, courtesy of OWASP.

```
default-src: Define loading policy for all resources type in case of a resource
script-src: Define which scripts the protected resource can execute,
object-src: Define from where the protected resource can load plugins,
style-src: Define which styles (CSS) the user applies to the protected resource,
img-src: Define from where the protected resource can load images,
media-src: Define from where the protected resource can load video and audio,
frame-src: Define from where the protected resource can embed frames,
font-src: Define from where the protected resource can load fonts,
connect-src: Define which URIs the protected resource can load using script inte
form-action: Define which URIs can be used as the action of HTML form elements,
sandbox: Specifies an HTML sandbox policy that the user agent applies to the prc
script-nonce: Define script execution by requiring the presence of the specified
plugin-types: Define the set of plugins that can be invoked by the protected res
```

`reflected-xss: Instructs a user agent to activate or deactivate any heuristics u`
`report-uri: Specifies a URI to which the user agent sends reports about policy v`

Creating A Policy

You can be as specific or as broad as you like when creating a CSP and fine tune it so that it meets your requirements exactly. Here are some example policies to show you what's possible.

`Content-Security-Policy: default-src https:` would allow any assets to be loaded over https from any origin.

`Content-Security-Policy: default-src scotthelme.co.uk` would allow any assets to be loaded from any origin on my domain using any scheme or port.

`Content-Security-Policy: default-src https://scotthelme.co.uk:443` would only allow assets to be loaded from my domain, over https and on port 443.

`Content-Security-Policy: default-src https://scotthelme.co.uk:*` would only allow assets to be loaded from my domain over https on any port.

Wildcards can be used for the scheme, the port and the left most part of a hostname only.

`*://*.scotthelme.co.uk:*` would match any scheme on any subdomain of scotthelme.co.uk, but not scotthelme.co.uk itself, and on any port.

Further to this, you can also use the following keywords alongside the `'self'` keyword mentioned above:

`'none'` blocks the use of this type of resource.

`'self'` matches the current origin (but not subdomains).

`'unsafe-inline'` allows the use of inline JS and CSS.

`'unsafe-eval'` allows the use of mechanisms like eval().

These can be applied in the following manner:

`Content-Security-Policy: default-src 'none'; script-src https://scotthelme.co.uk` would block the loading of any content apart from scripts loaded from my domain over https.

`Content-Security-Policy: default-src 'none'; script-src https://scotthelme.co.uk; style-src 'unsafe-inline'` would be the same as above with the addition of inline CSS.

It's worth noting that you can't specify a directive twice, as the second will be ignored.

`Content-Security-Policy: script-src scotthelme.co.uk; script-src google.com` would result in no scripts being loaded from google.com. The correct policy would be:

```
Content-Security-Policy: script-src scotthelme.co.uk google.com
```

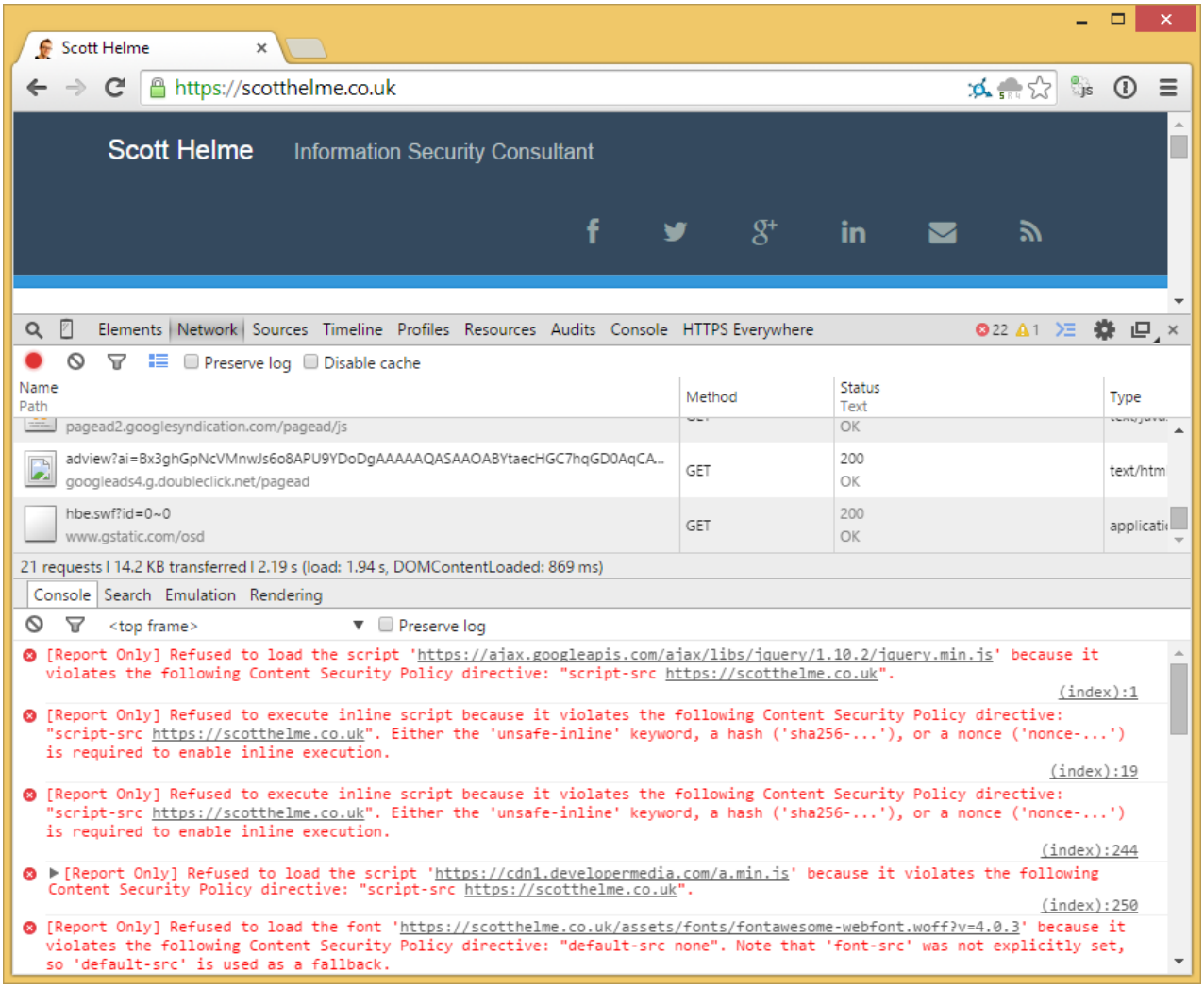
There is also no inheritance from the default source directive.

`Content-Security-Policy: default-src https:; script-src http://scotthelme.co.uk` would result in no scripts being loaded over https from scotthelme.co.uk. The correct policy would be:

```
Content-Security-Policy: default-src https:; script-src https://scotthelme.co.uk http://scotthelme.co.uk
```

Testing A Policy

Once you've created your policy, there's a really great feature you can take advantage of to test it. Instead of sending the header `Content-Security-Policy:`, you can send `Content-Security-Policy-Report-Only:`. This means the browser will receive and act upon the policy, but instead of enforcing it, it will give you feedback on what the effects of the policy would have been.



Now you can deploy, test and fine tune your policy without risking a disaster like accidentally blocking all JS on the page! Another great feature to take advantage of at this point is the ability to send both headers. Once you're happy with your policy and are sending the `Content-Security-Policy:` header, there will no doubt come a time when you want to make changes to that policy. What you can do is implement the changes you need and send the new policy in the `Content-Security-Policy-Report-Only:` header. The browser will continue to enforce the existing policy but you can also test and get feedback on the changes from the new policy without having any negative impact. I should probably also mention that CSP is only enforced on a per page basis. The browser will not cache a CSP header and continue to enforce it so you must send the header with every response that you want the policy to be enforced on.

Reporting Violations

It's all good and well having these policies in place, but it would be even better if we knew when the browser was taking action based on them. Perhaps an asset is being referenced in production that shouldn't be, or perhaps an attacker has found an XSS attack vector and is actively trying to exploit it. Without any form of active reporting, we have no way of knowing what's happening. For this, we have the `report-uri` directive. This directive specifies a location that the browser should POST a JSON formatted violation report to in the event it has to take action based on the CSP.

If I used a basic policy on my site to only allow assets to be loaded over https, but a reference to a http asset managed to find a way in, the browser would block the asset from being loaded and deliver a report to me.

The policy: `Content-Security-Policy: default-src https:; report-uri https://report.scotthelme.co.uk`

The offending item: `<img src="http://scotthelme.co.uk/logo.png">`

The resulting report:

```
{
    "csp-report": {
        "document-uri": "https://scotthelme.co.uk",
        "referrer": "",
        "blocked-uri": "http://scotthelme.co.uk",
        "violated-directive": "default-src https:",
        "original-policy": "default-src https:; report-uri https://report.scotth
    }
}
```

Breaking down the values in the report we have:

`document-uri`: The page the violation occured on.

`referrer`: The page's referrer.

`blocked-uri`: The origin of the blocked URI.

`violated-directive`: The specific directive that was violated.

`original-policy`: The entire policy.

A Word Of Caution

Though I haven't ventured much into the area of CSP Violation Reporting, I'd say that the implementation of such a feature would need to be approached with great care. The authenticity of incoming reports can never be assured and their contents could well be maliciously crafted by an attacker themselves. Any of the URI values in the report may link to malicious content, and visiting a unique URI could divulge information about the admin that received the report. It would also be highly susceptible to fuzzing and DDoS attacks. Implemented correctly and safely, violation reports can be a source of valuable data. Implemented incorrectly they could potentially do a great deal of harm.

Real World Limitations

Due to CSP working on the basis of whitelisting origins for content, using inline JS or CSS is considered as unsafe. The very foundation of an XSS attack is when an attacker finds a way to inject script into the page. Your typical XSS POC would look something like `alert('XSS')` and being inline in the page, the browser can't be sure whether to run the script. For this reason, inline JS and CSS are blocked by default and we have the `unsafe-inline` directive to allow it. If you don't allow inline JS, an attacker would be forced to try and load a script resource from another domain and then hit problems as they aren't in the whitelist. To be able to accommodate this, you need to externalise the content of any `<script>` tags. Instead of a block of code for your Google Analytics, you would have to create an external file and reference it like `<script src="/assets/js/ga.min.js"></script>` instead. That's not such a big problem, and I've externalised all of my JS and CSS. This also means inline event handlers like `onClick"doStuff();"` have to go and be replaced with `addEventListener()` calls instead. Still, not too much of an issue. That is, until something you depend on uses inline JS or CSS like your blogging platform, a plugin or a library.

My Disqus comment system includes the use of inline CSS and my advertising system uses inline JS too. If I use the `unsafe-inline` directive to allow these, I reduce the effectivenes of my CSP so the only other options are to use a hash or a nonce. The basic premise of these two methods is to allow the use of inline scripts or styles but to still control them with the CSP.

Nonce

The policy to utilise a nonce value would look something like this:

```
Content-Security-Policy: script-src 'self' 'nonce-SomeRandomNonce'
```

Any inline script on the page that you wanted to be executed would use that value like so:

```
<script nonce="SomeRandomNonce">...</script>
```

This method is more suited to pages that are generated dynamically and requires the use of a sufficiently random and long nonce to prevent an attacker predicting the value for their own scripts. You can use the same nonce value for every script on the page, but it has to be re-generated and inserted into the header and all script elements on each response. A static nonce value would be useless.

Hash

The next method is to place a hash of the script or style in the CSP header. More suited to static content, the browser will hash any inline JS or CSS and see if the digest matches a value found in the header. If it does, the content is safe for use.

```
Content-Security-Policy: script-src 'sha256-HashDigestHere='
```

The contents of any `<script>` tags on the page would then be hashed and compared to the value found in the CSP header. If the values matched, the script would be allowed to execute.

The Problem

As both of these scripts load 3rd party content, I can't go down the hash route as everything would break as soon as they made the tiniest change to any of the content. This leaves me with the nonce approach. Being quite difficult to implement and maintain, I'm effectively left with the choice of not having adverts and Disqus, or reducing the constraints the policy applies. For the time being, that effectively leaves me with the following policy:

```
Content-Security-Policy-Report-Only: default-src https: 'unsafe-inline'
'unsafe-eval'
```

All this does is ensure that all content on the page will be loaded over a secure connection, or not loaded at all, preventing any mixed content warnings. It doesn't offer any protection against XSS. I think this highlights some of the difficulties of implementing CSP and that building a site with CSP in mind from the beginning would be much better. A lot of the requirements, like externalising JS and CSS, are good to implement anyway, outside of the need for CSP. For now though, whilst I have a CSP in place, I would have much preferred something more secure.

Browser Support

Fortunately, support for the CSP header is widespread but there is one thing to watch out for, good old Internet Explorer. The `Content-Security-Policy` header is supported in the latest and greatest versions of Chrome, FireFox, Safari (OSX and iOS), Opera (but not Mini), the Android Browser and Chrome for Android. Internet Explorer, however, requires the `X-Content-Security-Policy` header instead. This means that if you want to have the most widespread support for your CSP header, you will need to issue it twice! I have to admit I'm not a great fan of that prospect but hopefully that will change in IE 12.

Read More: CSP Cheat Sheet - HPKP - HTTP Public Key Pinning - HSTS - The missing link in Transport Layer Security

---

CSP, *JS*, *XSS*, *Content Security Policy*

Previous Post : Getting an A+ on the Qualys SSL Test - Windows Edition

Next Post : PageSpeed - Performance optimisation made easy

The Author

Hi, I'm Scott Helme, a Security Researcher, international speaker and author of this blog. I'm also the founder of the popular securityheaders.com and report-uri.com, free tools to help you deploy better security!

Follow

Support

Enjoy my blog or find it useful? Please consider supporting me on Patreon, Flattr or PayPal.

Upcoming Events

Everything is Cyber-Broken!
23rd March

Hack Yourself First Workshop (Australia TZ)
26th - 27th March

Hack Yourself First Workshop (EU TZ)
1st - 2nd April

Hack Yourself First Workshop (EU TZ)
21st - 22nd April

Subscribe

You can use this IFTTT recipe to get an email when I publish a new blog!

There's also my RSS Feed.

Projects

Report URI

Security Headers

Why No HTTPS?

Crawler.Ninja

HTTP Forever

Cheat Sheets

CSP Cheat Sheet

HSTS Cheat Sheet

HTTPS Cheat Sheet

Performance Cheat Sheet

Projects

https://report-uri.com

Real-time security reporting for your site.

https://securityheaders.com

Analyse your HTTP response headers.

https://crawler.ninja

Daily crawl of top 1 million sites.

Popular Tags

HTTPS CSP HSTS HPKP HSTS-Preload Let's Encrypt securityheaders.com report-uri.com 2FA 2SV CSRF XFO XXSSP XCTO CDN Wardriving RSA ECDSA Table Storage Performance

Must Read

Alexa Top 1 Million Crawl - August 2017

29th August 2017

Year In Review | 2017

29th December 2017

My week in Vegas

14th August 2017