CHAMELEON

FOR OUR SMARTER WORLD

Content Security Policy Tutorial

By Lucas Kocon, 218510242
Chameleon Security

Contents

Introduction: What is a Content Security Policy?

The **Content Security Policy** (also known as a **CSP**) is a front-end/HTTP security system that maintains authority over what code runs on a given website. It is implemented inside HTML code and it constrains what dynamic code is allowed to run.

This feature is not a policy in that it is a way for people to do things (unlike policies in businesses or politics that dictate how things should be done in service of given goals and objectives), but it is a technical system created in HTML and programmable to restrict how much code is executed on the given website. It is a list of rules that describe what operations are allowed to run on the website, where its absence means that any code that appears is allowed to run. The **W3C** notes in their **Content Security Policy Level 3** technical document that its utility does not make this a “first line of defense against content injection attacks” but is “best used as defense-in-depth”.

What does a Content Security Policy offer?

The **Content Security Policy** puts the authority back into the server where it manages what dynamic code will run on the website. Where a given page may use external JavaScript code to provide utility for the website (in fact with the rollout and wide adoption of the AJAX platform it’s expected), the **CSP** works to only use resources the developers prescribe within the **CSP** as the website runs requests for the user and otherwise blocking unforeseen requests.

These unforeseen requests can appear as external JavaScript resource being referenced or a novel script created from scratch, both of which are examples of cross-site scripting attacks. Either way this resource will appear in any HTML element that creates HTTP requests with an open input field (such as a textbox for a search function or contact form) as a means of loading this code onto the server. The attack appears when the server receives the request but instead of following its developer-prescribed process, it performs a new process described in the user input, of which this attack can be prevented by the **Content Security Policy**.

Why does Chameleon need a Content Security Policy?

Each of Chameleon’s web projects include plans for dynamic features, least of which is a contact form that exists without function on the MOP site. These elements are the attack surfaces for cross-site scripting that will need to be protected as they gain function. But implementing this system is different to most other security features we recognise.

Most other technical security features feel like complete systems, where they seem fully-formed and ready to use without a demand for customisation. **Encryption** is one example where one solution to secure communications is a **certificate** to provide end-to-end secure communication, and that each different certificate does not differ much in how they work nor do they provide better or worse security (technically they do but it’s marginal); this unchanging practice creates the impression of a solved security problem. By contrast, the **Content Security Policy** is a solution that needs to be crafted for the project to provide security.

A **Content Security Policy** is a security asset that, just like the code base of the website, must update alongside to protect the project. Each website needs various tools to provide complex utility as it lives during production, but without updating the **CSP** it will block its integration (that is normal for whitelisting systems when they are not updated).

How was the Content Security Policy system introduced?

Websites were initially designed to offer a textual service in furtherance with their initial goal of sharing academic information on static webpages, and as it became a publicly available service many websites innovated with more dynamic content.

These dynamic utilities were the backbone for all Internet commerce towards the dot-com bubble and continues to this day for many more Internet services that we have previously used or remain popular today such as social media (Facebook, MySpace and X/Twitter), entertainment streaming (Youtube, Spotify and Netflix) and digital storefronts (eBay, Amazon and Steam). But as these services became the bedrock of financial success due to their widespread adoption and usage, the fundamental rules of how these websites operate (as all these services remain as websites) were abused and attacked to harm online and business operations.

This created a demand for website security systems so that existing systems continue to operate as their developers and business owners intend and continue to make money.

Why doesn't Chameleon's projects already have a CSP?

There are two differing explanations for why there is no existing Content Security Policy system for any of Chameleon's projects.

One explanation is that it is currently very difficult to include into the current Node.js framework that composes and creates each of Chameleon's projects. I had spent many weeks consulting online resources and tutorials on how one could implement a Content Security Policy in Node.js (refer), which work in teaching how to write a CSP but there is no consideration for how to implement it in the project's current environment. This will be a needed task for future Chameleon contributors. This difficulty of integration is also not an old problem, as a previous version of the MOP website that was composed in Python had a skeleton of a Content Security Policy built in.

Another explanation is how recognising and creating a **CSP** requires skills in security to recognise, and web development to implement a solution. It is not unusual to have experts specialised in a single domain with not all the necessary skills to combine. One such shortcoming I have in my development experience is in programming JavaScript and Node.js, where I could not find a solution to integrate a CSP into the project.

The key requirement is what file or Node.js module leads to its integration, and a follow-up requirement is how this CSP is integrated into the **<head>** element of the project's HTML pages so it can work immediately.

How do we write a Content Security Policy for Chameleon?

Let us assume a solution exists to integrating a Content Security Policy, so that we can address how to write a working **Content Security Policy** to bring to the project. These instructions are primarily sourced from **W3C's Content Security Policy** document that is also sourced by Mozilla's **MDN web docs** website that shares details that are helpful for website development.

The Content Security Policy should be written in the **<head>** section of the HTML code and it should be the first element included so that it can operate immediately. Within that section, it is included in a **<meta>** tag as it involves what data is included in the website. The CSP is defined within the **http-equiv** attribute and its rules must be written within the **content** attribute.

Now within the **content** field we work on what rules to apply. The format of these rules is to call a directive and then list all the trusted sources. These sources can compose of domains that one can refer by name or IP address, and certain symbols (note that the single-quotes are part of the phrase):

- **'none'**: this can explicitly not allow any resource to load if needed. Useful to remove any possible attack surface for cross-site scripting.
- **'self'**: only resources that pass the same-origin policy will load (same scheme, host and port).
- **'strict-dynamic'**: the directive will load resources when a paired **strict-dynamic** resource matches (more on that later).
- **'report-sample'**: creates a copy of the violation to send to the reporting endpoint that can be defined, and aid in making security improvements or evidence in incident response logs.
- **'unsafe-inline'**: allows inline resources to be resolved on the site. This is markedly unsafe as it enables user insertion of inline HTML code if it subverts data sanitation and opens an attack surface. Enable if inline resolutions are used by the website, but work to resolve so that their utility is not offered via inline. The self-retweeting tweet worked by automatically navigating a user's inline elements and automatically click the retweet button.
- **'unsafe-eval'**: allows dynamic code evaluation within the browser, which is definitely unsafe via creating novel JavaScript code that may also subvert sanitation. The Samy worm was injected and propagated via inline and needed **eval()** to subvert MySpace's attempts to block it.
- **'http:*.example.com/path/to/file.js'**: any web domain can be included in the **CSP** whitelist. Individual files can be sourced, while the origin domain can include all resources and prefacing with an asterisk ***** will include all the sub-domain's resources.

The fallback directive is **default-src,** where every element can adopt a general allow-list of resources. Any other written directives will apply and take priority over **default-src** for web security and functionality, but this is a safe cover-all-bases directive to include in any CSP.

The other directives that follow can be individually crafted to provide a different set of allowed resources to produce the webpage. These directives are:

- **connect-src**: this controls whatever outbound communication the page can make, useful for sending and receiving data from external services. If left open, attackers can use this lack of definition as a vector for data exfiltration.
- **font-src**: fonts are important partly for a webpage's style and mostly for maintaining readability. It is ideal to keep a list of sources that one trusts and can be modified at the project's discretion, in case of a web-defacing attack involving changing of fonts.
- **frame-src/object-src**: frames are elements where elements of other websites can be imported into the subject website, and objects are similar embed that utilises plugins that need to be loaded. Previously this existed in **<embed>** but it has been deprecated and its utility is dispersed between **<img>**, **<video>**, **<audio>** and **<iframe>** where that final element is the subject for this directive.
  If the project will use more frame elements to build up its website, it will need a more sophisticated **frame-src** policy to prevent clickjacking attacks. Objects also need to be controlled to prevent any arbitrary plugins from loading and become a new attack vector via exploited plugins.
- **img-src**: images are another key resource for websites to make it visually appealing. However if they are hosted externally and not controlled in the **CSP**, attackers can exploit this weakness by uploading viruses manipulated into image files. This highlights that as websites are built upon the resources found on other websites, these other websites become a part of the subject website's security environment.
- **script-src**: the most important directive to describe as it will be the source for all dynamic code that will run on the website. This will be the source of all major cross-site scripting attacks if the policy is not broadly covered, but it must be crafted carefully to not block the website's access to its own JavaScript resources.
- **style-src**: finally there is the style directive that manages visual elements throughout the whole webpage. If this resource is not managed by the **CSP**, attackers can deface the website.

Bibliography

- https://www.w3.org/TR/CSP3/
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy
- https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP
- https://cheatsheetseries.owasp.org/cheatsheets/Content_Security_Policy_Cheat_Sheet.html
- https://www.imperva.com/learn/application-security/content-security-policy-csp-header/
- https://www.imperva.com/learn/application-security/content-security-policy-csp-header/
- https://www.stackhawk.com/blog/nodejs-content-security-policy-guide-what-it-is-and-how-to-enable-it/