Enforcing Content Security by Default within Web Browsers

Christoph Kerschbaumer
Mozilla Corporation
ckerschbaumer@mozilla.com

***Abstract*—Web browsers were initially designed to retrieve resources on the world wide web in a static manner such that adding security checks in select locations throughout the codebase sufficiently provided the necessary security guarantees of the web. Even though systematic security checks were always performed, those security checks were sprinkled throughout the codebase. Over time, various specifications for dynamically loading content have proven that such a scattered security model is error-prone.**

Instead of opting into security checks wherever resource loads are initiated throughout the codebase, we present an approach where security checks are performed by default. By equipping every resource load with a loading context (which includes information about who initiated the load, the load type, etc.), our approach enforces an opt-out security mechanism performing security checks by default by consulting a centralized security manager. In addition, the added load context allows to provide the same security guarantees for resource loads which encounter a server-side redirect.

I. Motivation

Web browsers have become complex software applications that need to be capable of providing security guarantees when loading URIs from the web. For example, a browser needs to ensure that a web page can not access local resources on a user's computer. While web browsers were initially designed for retrieving and displaying information resources on the world wide web in static HTML, modern web browsers need to support the latest and constantly evolving web standards capable of loading resources, like e.g. the `fetch` specification [WHATWG, 2015].

Before loading a URI, web browsers have to perform numerous content security checks, such as evaluating that a script does not violate the same origin policy (SOP) [W3C, 2010b], or ensuring that the requested URI complies with the page's Content Security Policy [W3C, 2014a].

At first, performing security checks in selective locations throughout the codebase sufficed to provide the required security. However, the abundance of new web standards for retrieving resources on the web caused such sprinkled security checks to be error-prone. Even though systematic security checks are performed, a modern browser requires a central API to provide the same security guarantees for the different specifications capable of loading resources.

To complicate things further, the web evolved over time such that almost 12% of resource loads result in a server-side redirect (see Table I). To ensure the required security, browsers have to enforce the same security checks again after a server-side redirect to prevent a malicious page from circumventing the initial security checks by performing a redirect.

Instead of continuing current practice where developers have to opt-in to security checks whenever implementing a new standard for loading resources, we present an approach where content security is applied by default. Adding a central API that relies on an opt-out mechanism provides the needed infrastructure to prevent less security minded engineers from accidentally introducing vulnerabilities.

We first provide background on the different content security checks a browser performs before loading a URI (Section II) and contribute the following:

- We survey content security mechanisms before instantiating a load request and after a server-side redirect, and provide the ratio of redirected URIs on the web (Section III).
- We examine how Firefox has performed content security checks historically (Section IV) and present design and implementation details for enforcing content security by default within Firefox (v.50.0) (Section V).
- We provide an assessment of the engineering needed to retroactively provide an API for enforcing content security by default within a browser (Section VI).

II. Content Security Background

As of today, all major web browsers (Chrome, Edge, Firefox, Internet Explorer, Opera, Safari) are committed to implement most (if not all) of the security specifications defined by the *World Wide Web Consortium* (W3C) to improve security in web browsers. The most important of those specifications are:

- **Same Origin Policy** (SOP) [W3C, 2010b] The same origin policy prevents malicious script on one page from obtaining access to sensitive data on another web page through that page's Document Object Model (DOM) [W3C, 2004]. The SOP defines an origin as a combination of scheme, host and port number.
  More recently, the W3C started discussions about standardizing the concept of **Suborigins** [Joel Weinberger, Devdatta Akhawe, 2016]. Suborigins define a mechanism for programmatically defining origins to isolate different applications running in the same physical origin. User agents can extend the

DOI 10.1109/SecDev.2016.8

same-origin policy with this new namespace plus an origin tuple to create a security boundary between this resource and resources in other namespaces.

- **Cross Origin Resource Sharing** (CORS) [W3C, 2010a] To relax the same origin policy, the W3C specified CORS, which creates a whitelist of trusted domains by extending HTTP with a new origin request header. Hence CORS provides a mechanism which allows restricted resources (e.g., fonts) on a web page to be requested from a domain outside of the originating domain.
- **Mixed Content Blocking** [W3C, 2014b] The Mixed Content Blocker blocks insecure content on web pages that are supposed to be secure. HTTP [W3C, 2016] itself is not secure, which means connections are open for eavesdropping and man-in-the-middle attacks. If the main page is served over HTTPS but includes HTTP content, then the HTTP portion can be read and modified by attackers, even though the main page is served over HTTPS. When an HTTPS page has HTTP content, then the mixed content blocker can block such 'mixed' content.
- **Subresource Integrity** (SRI) [W3C, 2014c] SRI provides a mechanism for website authors to provide a cryptographic hash to a resource in addition to the location of the resource. Web browsers compare the hash provided by the website author with the computed hash from the fetched resource and load the resource only if the hashes match.
- **Content Security Policy** (CSP) [W3C, 2014a] The Content Security Policy allows web authors to define a whitelist in a HTTP header (or HTML meta element) to specify trusted sources for delivering content. For example, a CSP of `script-src https://good.com` permits the user agent to load script only when it is sourced from `https://good.com`. Three CSP directives are particularly important and should be highlighted:
  - **Upgrade Insecure Requests** [W3C, 2015b] The CSP directive `upgrade-insecure-requests` instructs user agents to upgrade all insecure HTTP URIs to their secure HTTPS equivalent. The directive targets web sites with large numbers of insecure legacy URIs that would otherwise need to be rewritten to secure links.
  - **Strict Mixed Content Blocking** [W3C, 2015a] The CSP directive `block-all-mixed-content` provides a stricter variant of mixed content checking which will block optionally-blockable mixed content in addition to blockable mixed content and prevents users from overruling the browsers decision.
  - **Require SRI for** [W3C, 2016] This CSP directive `require-sri-for` provides a mechanism for website authors to load resources only when SRI is defined for the resource load.

In addition to these specifications major browsers also implement security standards defined by the *Internet Engineering Task Force* (IETF):

HTTP Strict Transport Security (HSTS) [IETF, 2012] HSTS protects against protocol downgrade attacks by providing a mechanism for web servers to declare that web browsers should only interact with it using secure HTTPS connections and never allow connecting via the insecure HTTP protocol.

HSTS Priming [Mike West, Richard Barnes, 2016] HSTS priming proposes modifications to the behavior of HSTS to mitigate the risk that mixed content blocking will prevent migration from HTTP to HTTPS. Before blocking a third party subresource as mixed content, HSTS priming would perform an anonymous 'preflight' request to the subresource in question to check if the subresource is marked HSTS. If the subresource isn't available over HSTS it would be blocked by the mixed content blocker

Ultimately, browsers need to enforce **Access Permission Checks** to guarantee a webpage can not access local files. For example, a browser needs to block requests trying to load an image from the local file system, e.g. `file://home/secdev/conf.png`.

III. Performing Content Security Checks

Whenever a browser fetches a resource from the web, the browser has to perform all, or at least a subset of the security checks described in Section II.

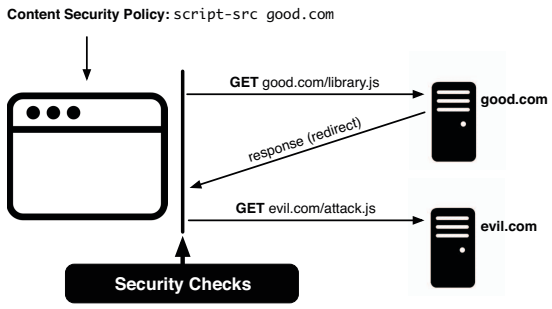

Fig. 1: Browser performing Content Security Checks (including checks after redirects).

Figure 1 shows a web browser that performs a `GET` request of the URI `good.com/library.js`. Before the browser actually initiates the network load, it performs at least an access permission check, or applies (if applicable) the more restrictive same origin policy. Additionally, the browser constrains mixed content and also enforces the page's Content Security Policy. As illustrated in Figure 1, the web page ships a Content Security Policy of `script-src good.com`, which instructs the browser to only load scripts originating from `good.com`. The browser now checks if the host portion of the resource matches the host defined in the CSP. In the example, the host of the URI matches the whitelisted host in the CSP,

hence the browser initiates the `GET` request for the given resource.

The server `good.com` responds with a server-side URI redirect (e.g., `301 Moved Permanently`) indicating that the URI will be redirected to `evil.com/attack.js`. Before the browser now initiates another `GET` request to the redirected URI, the browser has to perform the same content security checks again to make sure the redirect is not trying to mount an attack, e.g., by trying to load a local file.

In the example, the browser performs another CSP check and detects that script is not allowed to be loaded from `evil.com` because the page's CSP only whitelists scripts to be loaded from `good.com`. The browser blocks the load and logs an error message to the browser's console.

HTTP Response Status Codes incl. Description	*%*	*%*
2xx Success		**61.86**
200 OK	61.86	
3xx Redirection		**11.82**
301 Moved Permanently	0.76	
302 Found	7.66	
307 Temporary Redirect	3.33	
308 Permanent Redirect	0.07	
xxx Other responses		**26.32**
4xx, 5xx, ...	26.32	

TABLE I: HTTP connection responses [Mozilla, 2016]

As illustrated in Table I, roughly 62% of resource loads receive a status code of `200 Success` indicating that the resource is available for loading. The fact that almost 12% of URI loads are redirected again highlights the need for a unified API to enforce content security checks. We find that web sites using redirects has become de facto standard, which again motivates our work. Observe, that the data gathered in this paper covers a time frame of two months, taken on June 15th, 2016, where we collected and analyzed 2.4 billion URI loads.

IV. Enforcing Content Security within Firefox Historically

Terminology: The name of Firefox' layout engine is Gecko which reads web content, such as HTML, CSS, JavaScript, etc. and renders it on the user's screen. For loading resources over the internet, Firefox relies on the network library called Necko. Necko is a platform-independent API and provides functionality for several layers of networking, ranging from transport to presentation layers.

For historical reasons, Necko was developed to be available as a standalone client. That separation also caused security checks to happen in Gecko rather than Necko and caused Necko to be agnostic about load context. This separation in turn caused security checks after redirects to prove complicated. Nowadays, the separation of Gecko and Necko has vanished which allows to pass information about the load context from Gecko down into Necko.

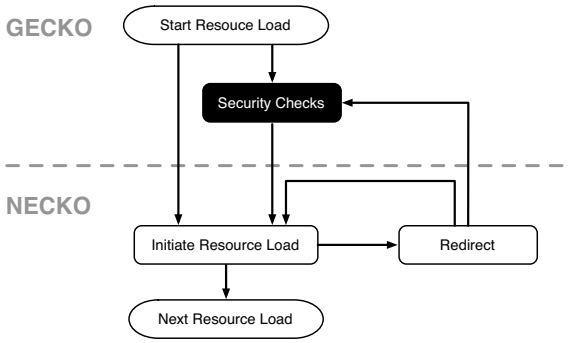

Fig. 2: Content Security Checks within Firefox historically: Gecko performs opt-in security checks and requests resources through Necko.

As illustrated in Figure 2, Gecko performs all content security checks before resources are requested over the network through Necko. The downside of this legacy architecture is, that all the different subsystems in Gecko need to perform their own security checks before resources are requested over the network. For example, `ImageLoader` as well as `ScriptLoader` have to opt into the relevant security checks before initiating a *GET* request of the image or script to be loaded, respectively.

Given that a browser needs to enforce numerous security policies, such a decentralized enforcing mechanism is error-prone. More generally, each location within the codebase that initiates a network load has to perform its own security checks. Worse, less security minded developers are more likely to forget the required security checks because there is no uniform API. To complicate things further, since Necko is agnostic to load context, each time a URI gets redirected, Necko has to call back into Gecko to perform any kind of security checks.

V. Enforcing Content Security within Firefox By Default

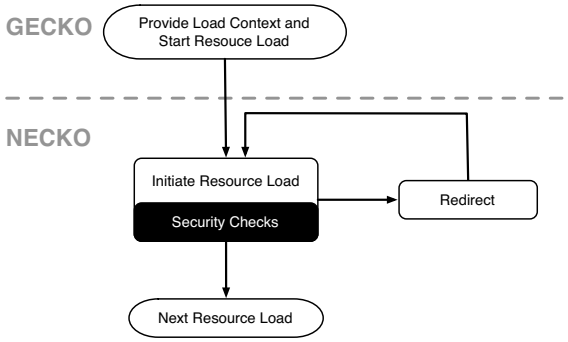

Fig. 3: Content Security Checks within Firefox by Default: Gecko provides load context and Necko performs security by default providing an opt-out mechanism to skip security checks.

As illustrated in Figure 3, we revamped the security landscape of Firefox providing an API that centralizes all the security checks within Necko.

Instead of performing ad hoc security checks for each network request within Gecko, our approach enables Gecko to provide information about the load context so Necko can perform the relevant security checks in a centralized manner. Whenever data (script, css, image,...) is about to be requested from the network, our technique creates an immutable `LoadInfo`-object which remains assigned to a network load throughout the whole loading process, across redirects.

```
1 LoadInfo {
2   nsIPrincipal* loadingPrincipal;
3   nsContentPolicyType contentPolicyType;
4   nsSecurityFlags securityFlags;
5 };
```

Listing 1: `LoadInfo`-object attached to each network request within Firefox.

The **`loadingPrincipal`** provides the key element within every `LoadInfo`-object and can not be null. The `loadingPrincipal` represents the origin context of the resource. For example, the web page `https://www.example.com` requests an image from `https://foo.com/bar.jpg` then the `loadingPrincipal` of that image load will be a principal object of the origin `https://www.example.com`. Ultimately, the `loadingPrincipal` is responsible for making the decision whether the load is allowed or denied.

In general, there are three types of principals:

1) *Content Principal:* A content principal is associated with some web content and reflects the origin of this content. Typically, a DOM window has a content principal defined by the origin of the window.
2) *System Principal:* The system principal passes all security checks and is attached to network loads that are triggered by the browser (system). For example, the *Safe Browsing* mechanism within Firefox updates its blacklist in the background every 30 minutes. A webpage can not manipulate the URI for updating that blacklist, hence it's safe that such a network request bypasses all security checks because the load is triggered by the browser (system).
3) *Null Principal*: The null principal fails almost all security checks and can not be accessed by anything other than itself (except system code). Such a null principal represents a resource that is only same-origin with itself. Commonly a null principal represents the origin when loading a *Binary Large Object* (`Blob`). Blobs represent data that is not necessarily in a JavaScript-native format.

The **`contentPolicyType`** describes the type of data that is about to be loaded over the network, e.g. a `contentPolicyType` might be script, image, style, etc.

As mentioned before, the values within the `LoadInfo` are immutable and can not be modified throughout the loading process of a resource.

The `contentPolicyType` for example allows the Content Security Policy to identify which CSP directory applies to the load and also indicates whether the load needs to be classified as optionally-blockable or blockable mixed content within the mixed content blocker.

The **`securityFlags`** determine what security checks need to be performed before data is fetched over the network. For example, the security flag `enforce_sop` indicates that the same-origin policy needs to be enforced. As mentioned in Section II the same origin policy is defined as scheme, host and port and is very restrictive. If the same origin policy does not need to be enforced, e.g. for image loads, then a security-flag of `allow_cross_origin` indicates that information can be loaded cross origin, but still triggers access permission checks before loading the resource to make sure local resources can not be loaded. Another common security flag is `enforce_cors` which indicates that cross origin loads are allowed, but the cross origin resource sharing headers need to be inspected before granting the load to succeed.

Important to mention is that Firefox' security by default mechanism enforces the most restrictive security checks before loading a resource over the network. The newly added opt-out mechanism requires developers to pass specific security flags to bypass certain security checks.

Ultimately, we added a ContentSecurityManager that bundles all content security checks within a single file and allows us to remove sprinkled security checks from each location initiating a resource load in the codebase. Such a centralization of security checks further enables novices to get an overview of security enforcing mechanisms by looking at a single file instead of browsing the whole codebase to spot relevant security checks.

Our efforts eliminated a set of vulnerabilities by enforcing security by default after server-side redirects. As previously mentioned within Section IV, enforcing security checks after redirects has been historically complicated within Firefox. One very illustrative example that our efforts resolved was a CSP bypass of server-side redirected images when loaded within a CSS file [1]. Instead of fixing the problem specifically for the one call-site that initiated the load, our architecture allows to fix the same security risk for all specifications capable of loading resources at the same time. Our approach allows to do so because all resource loads have to pass the same security checks centralized within the ContentSecurityManager.

VI. Evaluation

A. Engineering Effort

Firefox initiates network loads in around 100 locations in its codebase, ranging from image, script and style loads, over

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=949706

updating the blacklist for *SafeBrowsing* to `view:source` loads. In addition, Firefox' internal testing framework initiates over 400 network loads which all needed to be updated and equipped with the accurate `LoadInfo`-object.

In order to retroactively provide such a *Security by default mechanism* we landed (up to date) 518 changesets with a total diff of 126,322 lines of code. Please note that all landed changesets are generated using `hg diff -p -U 8` which provides eight lines of context and shows the relevant function for the block.

Over a period of 20 months, one engineer worked full time on revamping the security landscape of Firefox and dozens of others provided feedback, guidance and reviewed the code. Firefox is organized as modules, which means that one of the peers responsible for the code quality within the `ScriptLoader`, `StyleLoader`, `FontLoader`, etc. needs to review and accept the code before it can be merged into the codebase. Since this project modified code in all corners of the codebase, we had numerous discussions with reviewers from all those parts of the codebase. We have invested approximately 3,500 man-hours to retroactively patch Firefox to provide a security by default mechanism.

B. Regressions and Web Compatibility

As outlined in Section I enforcing security checks by default for every resource load has the benefit of providing safe defaults. That also means that all resource loads are now subject to the same security checks, including those resource loads that have been exempt from certain checks previously.

After deploying the newly created ContentSecurityManager and having the first resource loads rely on our security by default mechanism community members and volunteers started to report regressions because certain resources were blocked from loading on their website. Turns out, in almost all of the cases our security by default mechanism did not introduce regressions, but rather web compatibility issues. Those are issues where web pages started to rely on wrong behavior of web browsers. A few very illustrative examples of such web compatibility issues:

(1) The method link prefetch has never been subject to CSP, but in fact, that method can be used as a channel to exfiltrate sensitive user data from a webpage and hence CSP should provide a directive to restrict link prefetch[2]. For this particular example we raised an issue with the W3C working group [W3C, 2014a] and propose a new CSP directive to govern link prefetch.

(2) Before resource loads became subject to our security by default mechanism, addons were allowed to load Firefox internal document type definition (DTD) files. When converting the nsExpatDriver to enforce security by default, such DTD loads became subject to more security checks which then broke several addons[3]. Even though the new security checks would be correct to enforce, we decided to relax security checks for DTD loads[4] to keep such legacy addons working with the latest version of Firefox. Please note, that addons can not perform any critical actions when loading internal DTD files. Relaxing those security checks is simply a matter of keeping those legacy addons up and running.

(3) Loading favicons and performing the right security checks for favicons has always been crucial within web browsers. When we converted favicons to rely on our security by default mechansim we had to update our internal APIs and extend argument lists to account for the `loadingPrincipal` so Firefox can perform accurate security checks[5]. Addons, which implement that API using JavaScript would start breaking, hence we provided a default argument for five release cycles before we started to block favicon loads if addons do not explicitly provide the argument `loadingPrincipal`[6]. Please note that addons implementing that API in C++ would encounter a compile error.

In general, whenever we encountered web compatibility issues, we reached out to high volume sites/addons and let them know in advance to update their sites/addons so they remain working without any breakage.

C. Performance Impact

Firefox consists of millions of lines of code where volunteers and staff land code on a daily basis. Implementing the outlined security by default mechanism spanned over a period of 20 months, hence it's not feasible to provide meaningful performance measurements, because an increase or decrease in performance can not clearly be attributed to our security by default mechanism. For example, running JS benchmarks before starting the project and after finishing the project would not provide any accurate measurements, because so many parts of the codebase changed within that timeframe and changes within ScriptLoader or also the ImageLoader might have a bigger performance impact than revamping the security landscape.

To summarize, the architectural change we performed in fact just moved security checks from being performed opt-in to now being performed opt-out. Hence we argue that introducing a unified API for performing security checks by default has a negligible performance impact.

D. Comparison to other Browsers

Colloquially speaking we think that web browsers were initially designed to retrieve and display information resources on the web. We argue that the fundamental architecture of web browsers were not guided by the same security principles which have become de facto standard in the latest software products. Looking at the initial release dates of well known web browsers: Internet Explorer (1995), Firefox (2002), Safari (2003), Chrome (2008) lets us adumbrate that our approach might directly translate to their products as well. Since we

[2]https://bugzilla.mozilla.org/show_bug.cgi?id=1242902
[3]https://bugzilla.mozilla.org/show_bug.cgi?id=1226869
[4]https://bugzilla.mozilla.org/show_bug.cgi?id=1228116
[5]https://bugzilla.mozilla.org/show_bug.cgi?id=1119386
[6]https://bugzilla.mozilla.org/show_bug.cgi?id=1227289

do not have access to closed source based browsers we do not know how their internal security landscape looks like. WebKit based browsers [WebKit, 1998] however are facing similar issues we have described in this paper. Especially, enforcing the right security checks after a server-side redirect have been known issues. We think that our presented approach of enforcing security checks by default might very well be applicable to such browsers and we believe that such browsers would have to invest a similar engineering effort (see Section VI-A) to revamp their security landscape to perform content security checks by default.

VII. Related Work

Our work emphasizes on enforcing security mechanisms by default within a web browser, but was also inspired by numerous surveys and evaluations of browser security mechanisms ranging from the problematic situation of granting third-party script access to to application internals [Nikiforakis et al., 2012] to highlighting JavaScript security mechanisms within a browser [Bielova, 2013].

There has been substantial work on CSP [Stamm et al., 2010] and why CSP will succeed or fail [Kerschbaumer et al., 2016], [Fazzini et al., 2015], [Weissbacher et al., 2014], [Schwenk et al., 2015] and also large scale analysis of mixed content websites focusing and highlighting the pitfalls of mixed content blocking [Chen et al., 2015].

Also relevant, but not directly applicable are the approaches guiding us towards a formal verification of browser security [Akhawe et al., 2010] and formal shim verification for browsers [Jang et al., 2012].

VIII. Conclusion and Outlook

We have presented an approach that allows to perform the most restrictive content security checks by default within a web browser. Our approach attaches immutable information about load context to every network. The so called `LoadInfo`-object remains attached to the resource load throughout the whole loading process which further enables unified security checks after redirects.

We have proven that anchoring a security API retroactively into a browser rendering engine is possible through enormous engineering effort and conclude that implementing new standards for resource loading can hook into the provided API without the risk of jeopardizing a browser's security reputation.

Acknowledgment

Thanks to everyone in Security Engineering at Mozilla for their feedback, reviews, and provoking discussions. In particular, thanks to Jonas Sicking, Tanvi Vyas, Boris Zbarsky, Olli Pettay, Dan Veditz, Paul Theriault, Steve Workman, Doug Turner, and Eric Rescorla. Finally, also thank you to Sid Stamm and Stefan Brunthaler for their insightful comments.

References

[Akhawe et al., 2010] Akhawe, D., Barth, A., Lam, P. E., Mitchell, J., and Song, D. (2010). Towards a Formal Foundation of Web Security. In *Computer Security Foundations Symposium*. IEEE.

[Bielova, 2013] Bielova, N. (2013). Survey on JavaScript security policies and their enforcement mechanisms in a web browser. In *The Journal of Logic and Algebraic Programming*. Elsevier.

[Chen et al., 2015] Chen, P., Nikiforakis, N., Huygens, C., and Desmet, L. (2015). A Dangerous Mix: Large-scale analysis of mixed-content websites. In *Information Security*. Springer.

[Fazzini et al., 2015] Fazzini, M., Saxena, P., and Orso, A. (2015). AutoCSP: Automatically Retrofitting CSP to Web Applications. In *Software Engineering*. IEEE.

[IETF, 2012] IETF (2012). HTTP Strict Transport Security (HSTS). https://tools.ietf.org/html/rfc6797. (checked: August, 2016).

[Jang et al., 2012] Jang, D., Tatlock, Z., and Lerner, S. (2012). Establishing Browser Security Guarantees through Formal Shim Verification. In *USENIX Security Symposium*, pages 113–128. USENIX.

[Joel Weinberger, Devdatta Akhawe, 2016] Joel Weinberger, Devdatta Akhawe (2016). Suborigins. https://w3c.github.io/webappsec-suborigins/. (checked: August, 2016).

[Kerschbaumer et al., 2016] Kerschbaumer, C., Stamm, S., and Brunthaler, S. (2016). Injecting CSP for Fun and Security. In *Information Systems Security and Privacy*. Springer.

[Mike West, Richard Barnes, 2016] Mike West, Richard Barnes (2016). HSTS Priming. https://mikewest.github.io/hsts-priming/. (checked: August, 2016).

[Mozilla, 2016] Mozilla (2016). Telemetry Dashboards (Keyword=HTTP_RESPONSE_STATUS_CODE). https://telemetry.mozilla.org/. (checked: August, 2016).

[Nikiforakis et al., 2012] Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., and Vigna, G. (2012). You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In *Computer and Communications Security*. ACM.

[Schwenk et al., 2015] Schwenk, J., Heiderich, M., and Niemietz, M. (2015). Waiting for CSP: Securing Legacy Web Applications with JSAgents. In *European Symposium on Research in Computer Security*. Springer.

[Stamm et al., 2010] Stamm, S., Sterne, B., and Markham, G. (2010). Reining in the Web with Content Security Policy. In *World Wide Web*. ACM.

[W3C, 2004] W3C (2004). Document Object Model (DOM). http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/DOM3-Core.pdf. (checked: August, 2016).

[W3C, 2010a] W3C (2010a). Cross-Origin Resource Sharing (CORS). http://www.w3.org/TR/cors. (checked: August, 2016).

[W3C, 2010b] W3C (2010b). Same-Origin Policy (SOP). https://www.w3.org/Security/wiki/Same_Origin_Policy. (checked: August, 2016).

[W3C, 2014a] W3C (2014a). Content Security Policy (CSP). http://www.w3.org/TR/CSP2/. (checked: August, 2016).

[W3C, 2014b] W3C (2014b). Mixed Content. https://www.w3.org/TR/mixed-content/. (checked: August, 2016).

[W3C, 2014c] W3C (2014c). Subresource Integrity (SRI). https://www.w3.org/TR/SRI/. (checked: August, 2016).

[W3C, 2015a] W3C (2015a). CSP - Strict Mixed Content Blocking. https://www.w3.org/TR/mixed-content/#strict-checking. (checked: August, 2016).

[W3C, 2015b] W3C (2015b). Upgrade Insecure Requests. https://www.w3.org/TR/upgrade-insecure-requests/. (checked: August, 2016).

[W3C, 2016] W3C (2016). CSP - Require SRI for. https://github.com/w3c/webappsec-subresource-integrity/pull/32. (checked: August, 2016).

[W3C, 2016] W3C (2016). HTTP. http://www.w3.org/Protocols/. (checked: August, 2016).

[WebKit, 1998] WebKit (1998). Open source web browser engine). https://webkit.org/. (checked: August, 2016).

[Weissbacher et al., 2014] Weissbacher, M., Lauinger, T., and Robertson, W. (2014). Why Is CSP Failing? Trends and Challenges in CSP Adoption. In *Research in Attacks, Intrusions and Defenses*. Springer.

[WHATWG, 2015] WHATWG (2015). Fetch Standard. https://github.com/whatwg/fetch. (checked: August, 2016).