2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)

Hardening Firefox against Injection Attacks

Christoph Kerschbaumer
Mozilla Corporation
ckerschb@mozilla.com

Tom Ritter
Mozilla Corporation
tritter@mozilla.com

Frederik Braun
Mozilla Corporation
freddyb@mozilla.com

***Abstract*—Web browsers display content in the form of HTML, CSS and JavaScript retrieved from the world wide web. The loaded content is subject to the web security model and considered untrusted and potentially malicious. To complicate security matters, Firefox uses the same technologies to render its user interface as it does to render untrusted web content which blurs the distinction between the two privilege levels.**

Getting interactions between the two correct turns out to be complicated and has led to numerous real-world security vulnerabilities. We study those vulnerabilities to discover common threats and explain how we address them systematically to harden Firefox.

***Index Terms*—Web Security, Browser Security, Universal Cross-site Scripting, Hardening**

1. Motivation

Historically, web browsers have made a fairly sharp distinction between web content code and browser application code. Content code is part of the web page and hence subject to the web security model [31]–[34], [38], meaning that it is considered malicious and hence unprivileged and sandboxed. Browser code on the other hand represents the actual application code for the browser window and user interface.

While the majority of a browser's core is written in low-level languages such as C++, the front-end code in Firefox is written using standard web technologies such as HTML and JavaScript. This JavaScript code, commonly referred to as chrome code, runs with system privileges. Using internet technologies to write the browser user interface has a multitude of advantages such as rapid prototyping and quicker development cycles. Plus it has benefits for modularity, cross-platform development, and encourages a wider range of people to contribute.

Unfortunately, JavaScript in its dynamic and malleable nature contains a variety of insecure artifacts which provide the functionality to convert arbitrary strings to code at runtime. Including and relying on such insecure artifacts within privileged chrome code opens up an attack vector that potentially allows untrusted content code to trick chrome code into executing untrusted code in the privileged execution scope. And if chrome-privileged code is compromised, an attacker can take over the user's computer.

Over the course of a decade the distinction between untrusted content code and privileged chrome code has become blurry. We attribute this challenging lack of distinction to the fact that web browsers are constantly improving the user experience. Hence, it is not surprising that subtle bugs appear given the complexity of a browser codebase which consists of millions of lines of code and hundreds, if not thousands, of people working on it simultaneously. More precisely, *mozilla-central*[1] the codebase of Firefox, consists of over 272,000 files which contain over 4,000,000 lines of C++ code, around 3,000,000 lines of HTML and over 5,000,000 lines of JavaScript code. In the year 2019, 1,386 unique developers committed a total of 55,914 changesets to mozilla-central.

Expecting developers to write bug-free code on rapid release cycles is unrealistic and one can imagine that even the best code review process in combination with advanced static and dynamic code analysis can not prevent the introduction of bugs to a codebase as complicated as a web browser. Subsequently, Firefox began to suffer from privilege escalation bugs which allowed content code to execute with system privileges (see for example CVE-2017-7795, CVE-2017-7798, CVE-2017-7799, CVE-2018-5124, and CVE-2019-11718 reported to the *Common Vulnerabilities and Exposures*[2] database).

To compensate, we present a layered security approach which removes insecure artifacts at various levels in the codebase which allows hardening the Firefox web browser against all sorts of injection and privilege escalation attacks. While the provided implementation details are specific to the Firefox web browser, the presented hardening techniques apply to all applications which build on top of HTML and/or JavaScript and execute untrusted content alongside trusted.

We first provide background on the fundamental security architecture of the Firefox web browser (Section 2) and contribute the following:

- We survey insecure code fragments which should not be used by any application that executes untrusted code alongside trusted code and relies on HTML and JavaScript for both (Section 3).
- We examine and provide technical insights for the different hardening techniques we have incorporated into Firefox (v.75.0) to prevent privilege escalation attacks (Section 4).
- We provide a practical assessment (Section 5) and practical considerations (Section 6) when retroactively hardening the codebase of Firefox.

1. Download of the Firefox code repository 'mozilla-central' [19] on February 3rd, 2020.
2. https://cve.mitre.org

DOI 10.1109/EuroSPW51379.2020.00094

2. Background on the Security Architecture of Firefox

Like any web browser, Firefox loads JavaScript from untrusted and potentially hostile web pages, and runs it on the user's computer. The security architecture for separating web content and privileged content within Firefox however not only relies on the same-origin policy [31], but additionally builds upon the security boundaries provided by process isolation.

2.1. Separating Privilege Based on Processes

In older versions of Firefox for desktop, predating Firefox 48 from August 2016, the entire browser ran within a single operating system process. Specifically, the JavaScript that powered the user interface (chrome code) and the JavaScript that ran within web pages (content code) were not separated.

In 2016, Firefox was re-architected to run the user interface and trusted operating system operations in a separate process from web content. Due to the user interface being implemented in web technologies, the parent process still needs to be capable of rendering HTML with CSS and JavaScript.

As of February 2020, Firefox uses one privileged process to launch other processes and coordinate activities, eight web content processes, up to two additional semi-privileged web content processes, and four utility processes for web extensions, GPU operations, networking, and media decoding. Ultimately, Mozilla plans to support considerably more web content processes targeting the Site Isolation Architecture presented by Reis et al. [21].

The multi process architecture allows Firefox to separate more complicated or less trustworthy code into processes that have reduced access to operating system resources or user files - an execution model commonly referred to as *Sandboxing*. As a consequence, less privileged code will need to ask more privileged code to perform operations when it itself cannot. For example, a content process will have to ask the parent process to save a download because it does not have the permissions to write to disk. Put differently, if an attacker manages to compromise the content process it must additionally (ab)use one of the APIs to convince the parent process to act on its behalf.

2.2. Separating Privilege Based on Context

From a logical perspective, Firefox relies on the concept of a Principal for representing the security context of code and for performing security checks. A Principal represents an origin (the scheme, host, and port) and additional optional attributes [30]. To evaluate whether two contexts are same-origin, Firefox compares the Principals of the two contexts. In general, Firefox distinguishes between three types of Principals:

- *System Principal:* The System Principal passes all security checks and reflects the security context of all browser user interface (chrome) code.
- *Content Principal:* A Content Principal reflects the security context of web content. For example, when visiting the page `https://foo.com` then the DOM (Document Object Model) [37] window of that page has a content principal defined by the origin of the window.
- *Null Principal:* The Null Principal fails almost all security checks, is only same-origin with itself. The Null Principal uses a custom scheme and host, e.g. `moz-nullprincipal:{0bceda9f-...}`, where the host is represented as a universally unique identifier. The HTML standard names it a unique opaque origin [39]. For example, `iframe` elements with a `sandbox` attribute use a new Null Principal as their security context.

2.3. Differentiation of Chrome and Content Code

The two privilege distinctions - a privileged parent process and other lesser privileged child processes; and the privileged System Principal Context and the lesser privileged Content and Null Principal Contexts - both describe mechanisms of privilege distinction within Firefox. However, the actual mechanism by which privilege is separated is generally unimportant unless noted otherwise. Dating back to when Firefox had only a single process; 'chrome' code refers to the privileged code and related privileged user interface elements such as the address bar and 'content' code refers to the unprivileged code and is generally associated with a webpage.

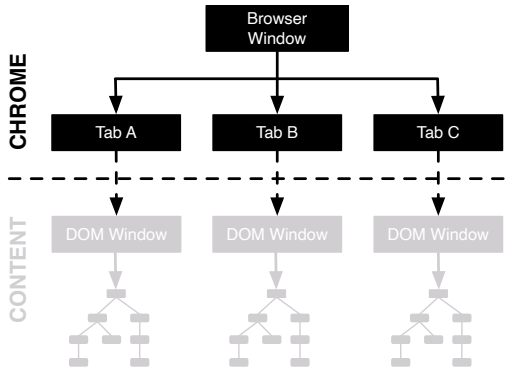

Figure 1: Firefox Security Architecture showing the separation of privileged chrome code and the Tabs A, B, and C including DOM windows exhibiting unprivileged content code.

As illustrated in Figure 1, the Browser Window represents the Firefox application window, including the URL Bar, all other toolbars and menus, as well as as all tabs that contain web pages. Underneath the Browser Window the tabs Tab A, Tab B, and Tab C represent the tabs that a user keeps open. Between any Tab and the corresponding DOM Window is the boundary between privileged and unprivileged code. Objects in the upper, privileged side are part of Firefox itself and allowed complete access to objects in a less privileged scope. They are granted a restricted view of such objects, specifically designed to prevent them from being tricked by the untrusted code (see Section 4.4). Objects in the lower, unprivileged side are loaded from the web as HTML, CSS and JavaScript.

3. Insecure Code Fragments liable for the majority of Injection Attacks

Building desktop applications using HTML and JavaScript and loading untrusted web content within such applications can bear privilege separation problems. In particular, because JavaScript supports the ability to take dynamically built strings and execute them as code which represents extremely valuable flaws for attackers to exploit. Enough so that even a small weakness, or one that does appear exploitable, is worth significant effort on the attacker's part to investigate.

Inspecting, evaluating and grouping security vulnerabilities recorded within Mozillas *Bug Tracking System* reveals that the top attack vectors responsible for injection attacks are the following three insecure code fragments. Relying on any of these fragments within the trusted code opens up an attack vector which potentially allows untrusted code to trick chrome code to execute untrusted code within the privileged scope.

3.1. DOM construction from arbitrary strings

Assignments to the property `innerHTML`, `outerHTML` and also `insertAdjacentHTML()` provide similar functionality on a DOM element, and the same security semantics discussed within this section hold true for all of them.

Description: The Element property `innerHTML` gets or sets the HTML or XML markup contained within the element. The `innerHTML` property of the document grants access to the current HTML source of the page, including any changes since the initial page load. Setting the value of `innerHTML` on an element removes all of the descendants within the element and replaces them with nodes constructed by parsing the provided string as HTML.

Security Risk: Even though browsers do not allow the execution of code inserted through `<script>` elements when assigning to `innerHTML`, there are numerous ways to cause JavaScript to execute.

For example, the following simplified code snippet:

```
1 <img src="x" onerror="alert('attack-code')">
```

will cause a failing image load and allow the injected code in the error event handler attribute to run. Again, the supplied code executes in the security context of the current runtime. For that reason, we recommend to abstain from using the insecure fragment `innerHTML`.

Alternative: Fortunately, there are alternatives to `innerHTML` which mitigate the security risks. For example, the fragment `Node.textContent` does not parse content from a string, but instead inserts it as raw text and hence eliminates the risk of dynamically parsing and executing JavaScript in the HTML markup.

3.2. JS execution of arbitrary strings

The JavaScript function `eval()` — along with the similar `new Function()`, `setTimeout()` and `setInterval()` — provides a powerful yet dangerous tool.

Description: The function `eval()` represents a global function which evaluates or executes a specified string argument.

Security Risk: The function `eval()` and aforementioned relatives parse and execute an arbitrary string in the same security context as itself. This feature conveniently allows executing code generated at runtime or stored in non-script locations like the DOM. Unfortunately the use of `eval()` also opens up the code to injection attacks which cause the injected code to be evaluated and executed in the security context of the current runtime. In other words, if an attacker manages to provide code which ends up within `eval()`, then the malicious code ends up running on the user's machine with the privileges of the current security context.

Besides the security risk, we discourage the use of `eval()` to dynamically generate code because the argument to `eval()` is not known a-priori and hence the compiler can not fully optimize variable lookup in the surrounding code which can result in a performance downgrade.

Alternative: Developers have often used `eval()` for legacy browser support in JSON parsing and keep it as a fallback if the global `JSON.parse()` function is not available. We recommend removing all of those fallbacks. The global JSON object has been available since the standardization of ECMAScript 5 [28] and can be considered widely supported.

3.3. Code execution from URIs

Description: The `javascript:` resource identifier scheme encodes executable JavaScript code within contexts that support resource identifiers. When navigating to the `javascript:` URL scheme, the browser does not actually seek a remote resource but will instead execute the supplied JavaScript code. If the code returns a string, the browser will parse and render it as a dynamically generated HTML document.

Security Risk: Navigations to `javascript:` URLs have been made popular in the early days of active HTML documents. If an attacker gains control over the code within the `javascript:` URL, it allows the attacker to execute arbitrary JavaScript code in the current context.

The section 'security-considerations' within the draft of *The 'javascript' resource identifier scheme* [12] recommends extreme caution in deciding where resource identifiers are recognized and permitted because of the in-context evaluation. Instead of recommending extreme caution of where to use the `javascript:` resource identifier scheme we recommend to eliminate it entirely from applications that rely on JavaScript for trusted and untrusted code to avoid potential injection and privilege escalation attacks.

Alternative: Instead of relying on `javascript:` URIs we recommend using static JavaScript code that sets pre-defined event handlers on HTML elements using the `addEventListener()` method.

4. Hardening Firefox

One fundamental difference between writing browser chrome code and web content code is that all the browser code can be defined upfront and hence be packaged and shipped with the browser itself. Restricting execution to code originating from that static code repository allows us to make a number of assumptions about browser chrome code, each enabling us to apply more constraints than we can apply to normal websites. In particular, browser chrome code:

- should never need to convert arbitrary strings to code, because we can define all the code directly in our codebase. Recognizing this property allows us to harden Firefox at two levels: (a) never execute inline Javascript, including eval(), and (b) apply a CSP to ensure injected JavaScript does not execute (See Section 4.1 and Section 4.2).
- should never need to load remote content (unlike real websites), because we can package all content within the browser code (see Section 4.3).
- will almost never want to interact with content-defined objects, and thus should ignore them by default (Section 4.4).

4.1. Hardening internal pages

Firefox not only renders web pages on the internet but also ships with a variety of built-in pages, commonly referred to as *about* pages. An about page provides an interface to reveal internal state of the browser.

Since such internal pages are also implemented using HTML and JavaScript they are subject to the same security model as regular web pages and therefore not immune from code injection attacks.

More figuratively, if an attacker manages to inject code into such an internal page, it potentially allows the attacker to execute the injected script code in the security context of the browser itself, hence allowing the attacker to perform arbitrary actions on the behalf of the user.

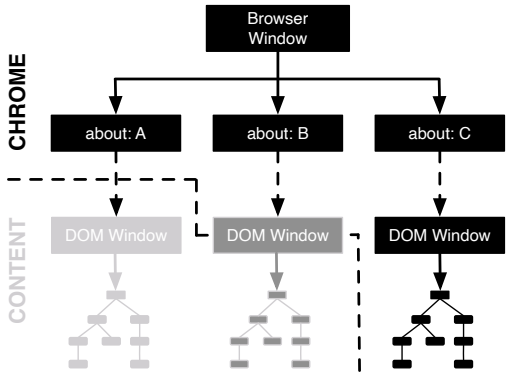

Figure 2: Firefox Security Architecture showing separation of privileged chrome code and the tab A, showing a *content privileged* about page; tab B, showing a *semi privileged* about page and tab C of a *system privileged* about page.

4.1.1. The privilege levels of internal pages. As illustrated in Figure 2, the Firefox web browser relies on three different privilege levels for building and exposing the built-in pages:

***Content Privileged* page:** A Content Privileged page and a regular web page loaded over the internet share the same security semantics. More precisely, there is absolutely no difference from a security perspective whether the page `https://foo.com` or the content privileged about page `about:credits`, which displays a list of contributors to Mozilla, loads within Firefox. In both cases, a Content Principal defines the security context, which completely separates the content code from the system code.

***Semi Privileged* page:** A Semi Privileged about page relies on a Content Principal for defining its security context. Additionally, the web browser grants special permissions to semi privileged about pages.

For example, the semi privileged page `about:privatebrowsing` is allowed to open a new window in *Private Browsing Mode*. To allow the page to actually switch to that privacy enhancing mode, Firefox exposes an API to semi privileged pages. This reduced API permits the page to actually flip an internal setting and allows the end user to open a new window using *Private Browsing Mode*.

***System Privileged* page:** A System Privileged page needs full access to Firefox internals and hence executes within the same security context as the browser chrome code itself, defined by a System Principal. Most prominently, `about:config`, which exposes an API to inspect and update preferences and settings which allows Firefox users to tailor their Firefox instance to their specific needs and desires.

For obvious reasons, system privileged pages expose the highest value target for attackers, because being able to inject code into a system privileged page instantly allows an attacker to run code with system privileges. To mitigate the risk of code injections and to add an additional layer of security to all of the built-in pages, we not only applied our hardening techniques to system privileged pages, but extended our protections to all 45 internal pages.

4.1.2. Rewriting all occurrences of inline JavaScript. Before adding a prevention mechanism that blocks the execution of injected script, we have to rewrite all inline event handlers because our proposed prevention mechanism can not distinguish between regular inline script and attacker injected inline script. For example, the following button element code defines some function to execute whenever the button is clicked:

```
<button id="myBut" onclick="myFunc(e)">clickme</button>
```

Can be reduced to the following HTML markup:

```
<button id="myBut">click me</button>
```

And the actual JavaScript code:

```
let myBut = document.getElementById("myBut");
myBut.addEventListener("click", (e) => { myFunc(e) });
```

The JavaScript code was moved into an external file which we package and ship with the browser itself. For example, for our running example, we create a file named `buttonCode.js` which we load relying on the pseudo protocol `chrome:` which loads packaged resources within Firefox. We then rely on the standard loading mechanism of external script within the HTML markup:

```
<script src="chrome://path/buttonCode.js"></script>
```

Moving all inline event handlers as well as all JavaScript code within `<script>` tags into the newly created external files allows us to remove all inline code within all of the internal pages.

4.1.3. Preventing the execution of injected JavaScript.

The removal of all inline JavaScript allows us to apply a strong Content Security Policy (CSP) [32] such as

```
default-src chrome:;
```

to all inernal pages. Please note that a CSP which does not include the keyword `'unsafe-inline'` causes the browser to block all inline JavaScript. Hence, our applied CSP ensures that JavaScript code only executes when loaded from a packaged resource using the internal `chrome:` protocol. Additionally, we added commit guards to assure that newly introduced pages within Firefox need to provide at least a CSP including a `default-src` directive. Those safe guards ensure that no new internal pages can be introduced to Firefox without a proper defense mechanism.

Not allowing any inline script in any of the internal pages limits the attack surface of arbitrary code execution and hence provides a strong first line of defense against code injection attacks.

4.2. Restricting calls to eval()-like Functions

```
1  bool CheckAllowEval(nsIPrincipal* aPrincpal,
2                      JSContext* aCtx) {
3
4    // if not in chrome content -> ALLOW
5    if (!aPrincpal->IsSystemPrincipal(){
6      return true;
7    }
8
9    // See Section 5.2 for practical adjustments
10
11   // loading eval() in chrome content -> DENY
12   ASSERT(false, "error: eval() in chrome context");
13   return false;
14 }
```

Listing 1: Restricting eval() within privileged contexts.

As discussed within Section 3.2, the JavaScript function `eval()`, along with the eval() family of functions expose a high impact attack surface. Hence, we remove all usages of it in privileged code, and enforce that any new calls to it will be ineffectual and not execute any argument provided to the function.

The internal browser implementation of the Content Security Policy within Firefox already provides the necessary security infrastructure for preventing the execution of eval(). More precisely, in case the keyword `'unsafe-eval'` is not present in the CSP to enforce then a browser's CSP implementation will block the execution of eval. We add our additional security restrictions on top of the already existing security infrastructure and ensure that eval() can not appear in privileged contexts.

As illustrated within Listing 1 we implement a security function `CheckAllowEval()` which asserts and returns false if a call to eval() occurs in system privileged contexts. While not exemplified, we enforce a similar check for any usage in the privileged parent process.

Before enforcing our new security monitoring system for calls to eval() in system privileged contexts we rewrote occurrences of eval() in privileged code in 54 distinct locations. In the majority of cases we could replace the call to eval() with a call to `JSON.parse()`. We attribute these legacy uses of eval() to the fact that `JSON.parse()` was not available when those calls to eval were introduced in the Firefox codebase. In most other situations, the offending code was a call such as `setTimeout("done()")` that we replaced with e.g. `setTimeout(function() { done(); })`.

As illustrated on Line 5, if the call to eval() does not happen in a privileged security context, reflected through a System Principal, then the eval()-call happens in regular web content. Since our eval restrictions do not apply to web content, only to privileged contexts, our introduced security function can return true at this point and allow the call to eval().

As indicated at Line 9, we make some practical adjustments to our newly introduced `CheckAllowEval` function. We refer the reader to Section 5.2 and 5.3 which discusses those adjustments. At this point, we know the call to eval() happens in a system privileged context and our dynamic eval() monitor will raise an assertion (see Line 12) and will return false and deny execution.

Removing all occurrences of eval() in system privileged contexts reduces the attack surface of arbitrary code execution and hence provides an additional layer of security against code injection attacks.

4.3. Restricting loads in privileged contexts

```
1  bool CheckAllowLoadInChromeCtxt(nsIChannel* aChannel) {
2
3    nsILoadInfo* loadInfo = aChannel->GetLoadInfo();
4
5    // if not in chrome content -> ALLOW
6    if (!loadinfo->LoadingPrincipal->IsSystemPrincipal(){
7      return true;
8    }
9
10   // if safe scheme -> ALLOW
11   nsIURI* uri = aChannel->GetURI();
12   if (uri->SchemeIs("chrome") ||
13       uri->SchemeIs("resource") ||
14       uri->SchemeIs("about") {
15     return true;
16   }
17
18   // See Section 5.3 for practical adjustments
19
20   // loading potentially dangerous code, e.g.
21   // remote content in the form of http(s)
22   // resources or javascript: URIs
23   // into chrome content -> DENY
24   ASSERT(false,
25     "error: loading insecure code in chrome context");
26   return false;
27 }
```

Listing 2: Restricting resource loads within system privileged contexts.

Higher privileged contexts within Firefox have access to APIs that are capable of modifying the browser settings and its user interface. We ensure that these APIs are restricted to HTML documents and scripts that are part of the Firefox source code. More specifically, system privileged code within Firefox should never need to load remote web content, because all the user interface code is packaged and shipped within the browser itself.

Firefox enforces a Security by Default [13] loading mechanism for all resource loads. Building on top of these efforts we add runtime assertions to ensure that Firefox can only load resources into system privileged context or the parent process if allow-listed by our introduced runtime monitor.

As illustrated in Listing 2 we call this monitor `CheckAllowLoadInChromeCtxt()` which returns true if the load is allowed, or raises an assertion and returns false if the monitor detects that the resource is not allowed to be loaded into system privileged contexts. While not exemplified, we enforce similar restrictions of resource loads in the parent process.

As illustrated on Line 6, the `nsILoadInfo` object holds information about the loading context that caused the resource load to occur. If the load is not going to happen within privileged chrome context, as represented through the check `LoadingPrincipal->IsSystemPrincipal()`, then the result of the resource load will not be evaluated in system privileged context and it is safe for our monitor to return true at this point,

Inspecting the scheme within the URI of the loaded resource allows Firefox to determine whether it is about to load a packaged resource, see Line 10. If the resource is of scheme `chrome:`, `resource:`, or `about:` then it is safe for our monitor to return true at this point, because all of these schemes represent packaged and allowed resources which are safe to load into system privileged contexts.

As indicated on Line 18, we made some practical adjustments to our monitoring system. We refer the reader to Section 5.4 which discusses those adjustments to our newly introduced `CheckAllowLoadInChromeCtxt` function.

If however, we end up at Line 24 within Listing 2, then our runtime monitor will raise an assertion and return false because we are about to load an unexpected scheme. For example, chrome privileged code should never need to load web content (iframe, image, script, ...) using a remote scheme like `http(s)`. Additionally, as discussed within Section 3.3, the insecure resource identifier `javascript:` should never need to occur in system privileged context. Summing it up, if our monitoring system implemented in `CheckAllowLoadInChromeCtxt()` detects a non allow-listed scheme, then it will raise an assertion and prevent the resource from loading by returning false.

4.4. Hardening using X-Ray Vision

Firefox runs JavaScript from a variety of different sources and at a variety of different privilege levels, and at times these privilege levels must interact. The security machinery in Firefox ensures that there is asymmetric access between code at different privilege levels. In more detail, Firefox ensures that content code can not access objects created by chrome code, but chrome code can access objects created by content. However, even the ability to access content objects can be a security risk for chrome code. X-Ray vision solves that confused deputy problem [10], wherein content leverages the high semantic flexibility of JavaScript and mutates JavaScript objects to behave differently than the privileged actor expects.

```
1 // Vulnerable Code in privileged context:
2 function getContents(element) {
3   let elementDoc = element.ownerDocument
4   return elementDoc.innerHTML;
5 }
6 // ...
7 getContents(searchbox);
```

One example of a confused deputy vulnerability is illustrated in the above code snippet. The vulnerable code in the above Listing inspects an element from content code (see Line 2). The code first accesses the defined property `ownerDocument` (Line 3) and then returns the containing document's HTML (Line 4).

The following attack code shows how to exploit the above vulnerability:

```
1 // Exploit Code in content context:
2 <iframe src="http://victim.com"></iframe>
3 <div id="searchbox"></div>
4 <script>
5   var sb = document.getElementById('searchbox');
6   Object.defineProperty(sb, 'ownerDocument', {
7     value: frames[0].document
8   });
9 </script>
```

Using the above attacker provided code, an adversary could trick the exposed function `getContents()` to return the markup of a cross origin iframe. Please note that the content script can not access the contents of the cross origin iframe on Line 2 directly due to the Same Origin Policy. However, the mutability of JavaScript allows the adversary to confuse the privileged code by redefining the trusted `ownerDocument` property (line 5). In turn, this confusion would allow the attacker to get access to a cross-origin iframe.

The fundamental principle of X-Ray Vision is that dynamic content-controlled data is only exposed through well-defined access points for a given object type. When a privileged script accesses a DOM object, it sees only the native version of the object. If any properties of the object have been redefined by web content, it sees the original implementation, not the redefined version. So in the example above, chrome code accessing the content element's `ownerDocument` would get the original version of `ownerDocument`, not the redefined version of the adversary.

X-Ray Vision is implemented by identifying the separation of execution contexts of JavaScript. Once this identification is made, we can extend the checks logically, beyond the confused deputy problem. If content code somehow got a reference to an object from a higher privilege level, here is where access would actually be blocked. And similarly, although two origins are the same privilege level insomuch as they are both content-privileged; they are separate execution contexts and are also blocked.

5. Practical Assessment of Hardening Firefox

Within this section we outline a practical assessment of the difficulties and unexpected behaviors we encountered when deploying the hardening techniques described in Section 4 to the Firefox codebase.

5.1. Some internal pages require remote content

As discussed within Section 4.1 Firefox ships with a variety of built-in pages. While it is possible to limit content loading to the packaged and shipped resources within Firefox for the majority of internal pages, some of the internal pages need to load content from the internet as well. For example, when entering `about:addons` into the URL Bar of Firefox, the page offers a variety of extensions and themes which allow to customize Firefox. The offered apps provide functionality to additionally protect passwords, download videos, customize browser themes and much more. Since the offered software programs are often developed by third parties, Firefox needs to provide a way to safely load a limited set of resources from the addons website.

Applying a custom Content Security Policy only allowing resources to load over the `chrome:` protocol as suggested within Section 4.1 would not work and would block remote resources. Instead `about:addons` applies a CSP similar to the following:

```
default-src chrome:; frame-src chrome: https:;
```

which allows filling iframes using not only the `chrome:` protocol but additionally supports loading of sub documents using the `https:` protocol.

We want to apply the most restrictive CSP to every internal page and only allow well defined remote resources, like in the case of `about:addons` iframes for displaying the extension, to load. Most important for our hardening efforts is that a CSP of any of the internal pages does not include the keyword `'unsafe-inline'` within the `script-src` or the fallback directive `default-src`. Again, the absence of that keyword within a CSP is what makes sure that any injected JavaScript code does not execute in the privileged context and hence provides a strong defense mechanism against code injection attacks.

5.2. Internal tests and debugging require eval()

```
1  bool CheckAllowEval(nsIPrincipal* aPrincpal,
2                      JSContext* aCtx) {
3
4    ...
5
6    static nsString evalAllowlist[] = {
7      // Test-only utility
8      NS_STRING("resource://testing/content-task.js"),
9      // The Browser Toolbox/Console
10     NS_STRING("debugger"),
11   };
12
13   // if allow-listed -> ALLOW
14   nsString fileName = aCtx.getFileName();
15   for (nsString allowlistEntry : evalAllowlist) {
16     if (fileName.Equals(allowlistEntry)) {
17       return true;
18     }
19   }
20   ...
21 }
```

Listing 3: Practical adjustments to restricting eval() within chrome privileged contexts.

When running our test suite, we encountered a number of usages of the eval() family which we attribute to our testing infrastructure. No attacker can influence the string to be evaluated and also no attacker can craft a file relying on the Firefox internal `resource:` protocol, hence we decided to allow-list such occurrences of eval() and focus our efforts on replacing and updating more alarming use cases of eval() in the Firefox codebase which affect Firefox end users.

As illustrated on Line 8 and Line 10 within Listing 3, we allow-list the filename `content-task.js` and `debugger` respectively. Calls to eval() in first filename, `content-task.js`, can only occur within our testing infrastructure as the file is not shipped to end users. Hence we consider it safe to allow-list the calls originating in that file. Additionally, we allow eval() when it is called from the Browser Toolbox, which is a development and debugging aid which can not be influenced by any web page. If the call to eval() occurs in one of the two contexts, then our newly added runtime assertions allow the call to eval by returning true (see Line 17).

While removing eval() support from the parent process, we had to make exceptions for two extremely common idioms for gaining access to the global `this` object:

```
1 eval("this");
```

and

```
1 Function("return this")
```

both of which execute constant strings and can not be injected into, hence we consider those two exceptions safe.

5.3. Discovering legacy eval() through Telemetry

We started to encounter differences between the Firefox test environment and in-the-wild Firefox configurations with respect to eval() usage, hence we began to collect Event Telemetry in pre-release builds which reports limited information about eval usage, while simultaneously allowing those usages to occur. This telemetry information provides insights of eval() usages that our introduced runtime monitor would have blocked, before potentially downgrading end user experience.

Evaluating Telemetry results allowed us to discover situations where eval() was being used but at the same time was not visible from the mozilla-central codebase, for example in situations where the runtime executes user supplied JavaScript. Historically Firefox supported a mechanism which allowed the user to execute user-supplied JavaScript in the execution context of the browser. Back then this feature, now considered a stability and security risk, provided a mechanism to customize Firefox at start up time and was called `userChrome.js`. When Firefox removed support for executing user-supplied JavaScript, users found a way to accomplish some of the goals using a mechanism that closely mimicked `userChrome.js`.

Unfortunately we have no control of what users put in these customization files, but our runtime checks confirmed that in some cases, these `userChrome.js`-like scripts included eval. Please note that our threat model targets injection attacks and leaves powerful attackers out

of scope. If an attacker could place an allow-listed file on the users computer, then the attacker is already so powerful rendering our injection prevention techniques obsolete. Anyway, when we detect that the user has enabled such tricks, we disable our blocking mechanism and allow usage of eval().

Our introduced telemetry will continue to inform the Mozilla Security Team of newly introduced and/or yet unknown instances of eval() which we will closely audit and evaluate and restrict as we further harden the Firefox Security Landscape.

5.4. Not all remote content is unwanted

When removing support for loading remote web content in privileged contexts and the parent process (Section 4.3) we identified a number of situations we needed to account for.

```
1  bool CheckAllowLoadinChromeCtxt(nsIChannel* aChannel) {
2
3    ...
4
5    static nsString loadAllowlist[] = {
6      NS_STRING("http://detectportal.firefox.com"),
7      NS_STRING("https://safebrowsing.google.com"),
8      NS_STRING("http://ocsp.digicert.com"),
9    };
10
11   // if allow-listed -> ALLOW
12   nsIURI* uri = aChannel->GetURI();
13   nsString uriSpec = uri->GetSpec();
14   for (nsString allowlistEntry : loadAllowlist) {
15     if (uriSpec.Equals(allowlistEntry)) {
16       return true;
17     }
18   }
19
20   ...
21 }
```

Listing 4: Allow-listed set of safe origins in the remote content filter to maintain existing Firefox functionality.

As illustrated within Section 4 on Line 5, we add an allow-list of resources that may load in system privileged context. The list maintains existing Firefox functionality while lowering default privileges. In particular, the code to detect captive portals has to issue a request to a HTTP page (Line 6) to open an opportunity for networks to intercept and redirect. Our added restriction can not simply block the load, but we limit the possible requests to uses in `fetch()` rather than allowing document, image and script loads. This removes exposure of media parsing and JavaScript code execution code from insecure HTTP. The captive portal detection URI is hardcoded within Firefox and web pages can not influence that URI, hence we consider it safe to load as a data resource in the system privileged context.

Another example of data being downloaded is the *Safe Browsing* [6] mechanism within Firefox, which updates its malware list in the background every 30 minutes. A webpage can not manipulate the URI for updating that list, and the response is not used with extensive parsing, display or code execution, hence it is safe to allow-list the URL `https://safebrowsing.google.com`, see Line 7.

Further, the resource load `http://ocsp.digicert.com` is connecting to the OCSP (Online Certificate Status Protocol) [23] responder to determine revocation status of a certificate. Again, a webpage can not influence the URL for OCSP and we expose only a limited subset of our parsing code. Hence we allow the online certificate status protocol on Line 8.

It's important to emphasize that the presented allow-list mechanism provides a secure-by-default mechanism, which blocks loading of remote resources in privileged context as well as the parent process. Any exceptions to it will be well documented and allows the security team to create a mental model around those exceptions. Trying to load any other remote resources will be blocked by default.

5.5. Adapting X-Ray Vision

Initially, the implementation of X-Ray vision for JavaScript objects was limited to interfaces specified with an accompanying WebIDL definition [4]. More precisely, the WebIDL provides expectations about how objects operate in a programmatic way and hence fundamentally enabled the architecture of X-Ray vision. Unfortunately not all objects have such a definition, e.g. those native to JavaScript are lacking a WebIDL definition.

The lack of a safe interaction channel with objects that are native to the ECMAScript language but not specified as WebIDL (e.g., `Date`) opened up a new attack vector. Additionally, some WebIDL types like `any` and `object` do not allow for strict type checking which also limited the capabilities of the X-Ray vision architecture. In more detail, the DOM standard specifies the `detail` property of the `CustomEvent` interface as type `any`, which allows to pass arbitrary JavaScript [37]. Consequently, such arbitrary strings can include cross-origin objects or obscure objects with custom, attacker-defined getter and setter functions.

Addressing the above shortcomings in the design of X-Ray vision allowed us to additionally harden the architecture by building on secure-by-default principles. Previously, properties defined as objects in WebIDL were not properly screened and thus transparent to the X-Ray concept. We added default support for these properties as well as for built-in objects that are not defined in WebIDL by leveraging the JavaScript engine's ability to treat such objects (e.g., `Date`). Furthermore, the types `Object` and `Array` do not have well-defined semantics. As they are frequently used to express simple dictionary-like structures, we allow traversal by accessing a property that itself is an `Object`, and present it as an X-Ray object with the same restrictions.

For all other types the X-Ray design bypasses any mutated behavior in the object and instead returns the original implementation. The design disallows access to accessor properties (getters and setters), as this would lead to security issues like unexpected code execution in a separate privileged context or execute user code that potentially modifies the object in place while native code is reading from it at the same time, leading to data races and memory safety issues.

If an object cannot be viewed through an X-Ray wrapper, our security by default mechanism turns it opaque so privileged code cannot view its attributes or interact with it, hence avoiding any confused deputy problems in privileged chrome code.

6. Implementation Considerations

Within this section we will highlight engineering effort as well as a comparison to other browsers. While the provided implementation and practical effort was specific to the Firefox web browser, the emergence of browser-powered desktop applications, security problems hitherto reserved for the web domain become increasingly relevant and pressing for desktop apps.

6.1. Engineering Effort

In order to retroactively harden the Firefox codebase against code injection and privilege escalation attacks we landed (up to February 3rd, 2020) 263 changesets with a total diff of +14,830 and -10,666 lines of code. Over a period of 18 months, three engineers worked part time (20 hours a week) on hardening the security landscape of Firefox and dozens of others provided feedback, guidance and reviewed the code. Note that Firefox is organized as modules [18], which means that one of the peers responsible for the code quality within each of the `Desktop Firefox`, `Security UI`, `Privacy UI`, etc. modules needs to review and accept a changeset before it can be merged into the codebase. Since this project modified code throughout the entire codebase, we had numerous discussions with reviewers from all those parts of the codebase. We have invested approximately 4,000 hours to retroactively harden the codebase of Firefox against privilege escalation attacks.

6.2. Comparison to other Browsers

While the Chrome Browser is an exception, many Web Browsers (Edge, Edgium, Opera, Safari, Vivaldi) are based on open source components, but have a closed-source User Interface. Unfortunately, this access restriction to source code makes a direct comparison difficult. However, individual reported vulnerabilities provide some insight into the nature of their design and reveals that other browsers suffer from similar privilege escalation problems.

For example, Golubovic [9] details a flaw in Vivaldi that opens up a universal Cross Site Scripting vulnerability. When a SVG image with embedded JavaScript is opened from the Notes feature, the JavaScript will be executed in the privileged JavaScript context, allowing access to the browser's storage object which contains and hence grants access to the user's browsing history.

Similarly, a recent vulnerability disclosed in Edgium (Chromium based Edge) reveals that a website could exploit a Cross Site Scripting vulnerability in the New Tab page to execute JavaScript in a privileged context[3]. After exploring the objects available, the author discovered that the method `chrome.qbox.navigate()` passes unsanitized values to C++, resulting in a likely-exploitable crash.

Chromium on the other hand is entirely open source, and lends itself to a more detailed case study and inference to the shipped Chrome browser. Chrome builds upon a limited implementation of privileged UI using web technologies, primarily limited to their equivalent of *about* pages, which are called *chrome* pages, not to be confused with chrome meaning privileged context in this work. They place similar restrictions on these pages as Firefox does (see Section 4.1), preventing them from embedding and fetching remote resources of web content.

In contrast to Firefox, Chrome does not have a concept of X-Ray Vision; instead they use Isolated Worlds which do not permit access outside the world. However, two differently-privileged worlds (such as an extension's content script and a web page's JavaScript) can share access to the same window and document. In this situation, custom defined properties on accessible objects are not visible, but changes to built-in properties like the element's ID are. Nonetheless; Chrome encountered situations where it was possible to gain access to another world's objects[4].

Additionally, Isolated Worlds are not intended to provide absolute security by default, but rather to create a base upon which secure scripts can be written. It is possible for a lesser privileged script to confuse a more privileged script by polluting the more privileged script's world with unexpected variables; this was used to confuse the LastPass Extension and resulted in Remote Code Execution[5].

Electron is built upon Chromium, but it is intended for developers to write privileged UI using web technologies. It provides extensive security guidelines about writing privileged UI[6] and has two privileged sets of APIs known as Node.js integration and the remote module; these APIs include the `shell.openExternal()` API for executing a program of the author's choosing. To limit the impact of a script injection attack; Electron strongly recommends disabling both Node.js integration and the remote module on any browser window that loads remote content (as opposed to pre-packaged UI). This effort to prevent remote code loading aligns with the efforts of Firefox and Chrome to prevent privilege escalation attacks.

Similar to Firefox, Electron does support applying a Content Security Policy to one's UI. Finally, Electron does inherit the Isolated World concept, and uses it as a way to protect Electron APIs and preloaded scripts. However, Electron does have a couple of very concerning settings and behaviors: by default any permission prompt that a web browser would show (e.g. Camera or Geolocation access) is automatically granted unless the developer specifically chooses to intercept the request and handle it in some way. Additionally, it is possible to entirely disable the same origin policy for a browser window.

7. Related Work

Our presented work emphasizes techniques for hardening the security landscape of Firefox and was influenced by various approaches to prevent Cross Site Scripting. For example, *Securing the Tangled Web* [5] focuses on preventing script injection vulnerabilities through software design. Our work removes insecure artifacts in the privileged chrome code and hence also prevents vulnerabilities through software design. The approach of *Trusted Types* [15] is also comparable to our work. The idea

3. https://leucosite.com/Edge-Chromium-EoP-RCE/

4. https://bugs.chromium.org/p/chromium/issues/detail?id=85158
5. https://bugs.chromium.org/p/project-zero/issues/detail?id=1225
6. https://www.electronjs.org/docs/tutorial/security

behind Trusted Types is that it allows web applications to instruct the browser to only accept non-spoofable, typed values in place of strings for known DOM XSS sinks and hence provides a mechanism to prevent DOM based XSS in web applications. Since in our case we had control over both, the user agent and the web application code in form of the user interface code we did not have to annotate any of our web application code. Instead we modified sinks within the user agent directly to not allow any calls to eval() and also to not allow any non-browser packaged resources to load in system privileged context.

In addition, our work was inspired by numerous surveys and evaluations of browser security mechanisms ranging from the problematic situation of granting third-party script access to application internals [20] to highlighting JavaScript security mechanisms within a browser [1].

In particular, our efforts focused on preventing privilege escalation attacks which would allow an attacker to fully take control of a users computer. Even though systems like *Fidelius* [8] - an architecture which protects user secrets even if the entire underlying browser is fully controlled by an attacker - would at least protect an end users private data, any privilege escalation attacks would still allow an attacker to perform arbitrary actions. To lower the risk of privilege escalation attacks, our presented work was further inspired by our colleagues who have provided a multitude of DOM based XSS prevention mechanisms [16], [17], [26], [27], [29]. In principle, privilege escalation prevention in a browser setting is fundamentally equivalent to XSS prevention techniques.

Even though server side techniques for preventing XSS do not directly apply to our presented hardening efforts, they still provided valuable input for our work [7].

Before applying a strong CSP to Firefox internal pages we evaluated the effectiveness on the different strategies to apply a CSP, ranging from allow-list based CSPs to nonce-based CSPs. Ultimately our hardening techniques require a guard which blocks all inline script from execution, hence we settled on a scheme-based CSP which does not allow any scripts outside of our shipped product, much less `'unsafe-inline'` and hence does not allow any injected script to execute. Still, works from our colleagues on CSP [2], [3], [11], [14], [22], [24], [25], [35], [36] heavily motivated and supported our undertaking of hardening the security landscape of Firefox.

8. Conclusion

We have presented techniques which harden the Firefox web browser against privilege escalation attacks. While the provided implementation and described practical effort was specific to the Firefox web browser, the presented hardening techniques apply to all applications which load and interact with untrusted code like JavaScript. As more and more applications are built on top of web technologies the presented techniques target any application that displays or executes untrusted content alongside trusted and relies on HTML and JavaScript for both. We have shown that removing insecure code fragments from a codebase limits the attack surface against code injection attacks and hence provides a strong first line of defense against arbitrary code execution.

Acknowledgments

Thanks to everyone in *Security, Privacy, Networking Engineering* at Mozilla for their feedback, reviews, and thought-provoking discussions. In particular, thanks to Steven Englehardt, Thyla van der Merwe, Ethan Tseng, Wennie Leung, Selena Deckelmann, Vinothkumar Nagasayanan, Jonas Allmann, and Bobby Holley. Finally, also big thanks to Stefan Brunthaler and Mario Heiderich for their insightful comments.

References

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

[2] S. Calzavara, A. Rabitti, and M. Bugliesi. Content security problems?: Evaluating the effectiveness of content security policy in the wild. In *Proceedings of the Conference on Computer and Communications Security*. ACM, 2016.

[3] S. Calzavara, A. Rabitti, and M. Bugliesi. Semantics-based analysis of content security policy deployment. *ACM Transactions on the Web*, 2018.

[4] Cameron McCormack. WebIDL Level 1. https://www.w3.org/TR/WebIDL-1/, 2016. (checked: February, 2020).

[5] Christoph Kern. Securing the Tangled Web. https://queue.acm.org/detail.cfm?id=2663760, 2014. (checked: February, 2020).

[6] G. Corporation. Google Safe Browsing. https://safebrowsing.google.com/, 2007. (checked: February, 2020).

[7] A. Doupé, W. Cui, M. Jakubowski, M. Peinado, C. Kruegel, and G. Vigna. deDacota: toward preventing server-side XSS via automatic code and data separation. In *Proceedings of the Conference on Computer and Communications Security*. ACM, 2013.

[8] S. Eskandarian, J. Cogan, S. Birnbaum, P. Brandon, D. Franke, F. Fraser, J. Garcia, E. Gong, H. Nguyen, T. Sethi, V. Subbiah, M. Backes, G. Pellegrino, and D. Boneh. Fidelius: Protecting User Secrets from Compromised Browsers. In *Proceedings of the Symposium on Security and Privacy*. IEEE, 2018.

[9] N. Golubovic. Attacking Browser Extensions. https://golubovic.net/thesis/master.pdf, 2016. (checked: February, 2020).

[10] N. Hardy. The Confused Deputy: (or why capabilities might have been invented). In *SIGOPS Operating Systems Review 22*. ACM, 1985.

[11] D. Hausknecht, J. Magazinius, and A. Sabelfeld. May I?-Content Security Policy Endorsement for Browser Extensions. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer-Verlag, 2015.

[12] B. Hoehrmann. The 'javascript' resource identifier scheme. https://tools.ietf.org/html/draft-hoehrmann-javascript-scheme-03, 2010. (checked: February, 2020).

[13] C. Kerschbaumer. Enforcing Content Security by default within Web Browsers. In *Proceedings of Cybersecurity Development Conference*. IEEE, 2016.

[14] C. Kerschbaumer., S. Stamm., and S. Brunthaler. Injecting CSP for Fun and Security. In *Proceedings of the International Conference on Information Systems Security and Privacy*. Springer, 2016.

[15] Kotowicz, Krzysztof and West, Mike. Trusted Types - Editor's Draft, 15 January 2020. https://w3c.github.io/webappsec-trusted-types/dist/spec/, 2020. (checked: February, 2020).

[16] S. Lekies, B. Stock, and M. Johns. 25 Million Flows Later: Large-Scale Detection of DOM-Based XSS. In *Proceedings of the Conference on Computer and Communications Security*. ACM, 2013.

[17] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia. Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting. In *NDSS*, 2018.

[18] Mozilla. Firefox Modules. https://wiki.mozilla.org/Modules/All, 2020. (checked: February, 2020).

[19] Mozilla. Getting Mozilla Source Code Using Mercurial. https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Source_Code/Mercurial, 2020. (checked: February, 2020).

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

[21] C. Reis, A. Moshchuk, and N. Oskov. Site Isolation: Process Separation for Web Sites within the Browser. In *Proceedings of the USENIX Conference on Security Symposium*. USENIX, 2019.

[22] S. Roth, T. Barron, S. Calzavara, N. Nikiforakis, and B. Stock. Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies. NDSS, 2020.

[23] B. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet Public Key Infrastructure - Online Certificate Status Protocol - OCSP. https://tools.ietf.org/html/rfc6960, 2013. (checked: February, 2020).

[24] D. F. Some, N. Bielova, and T. Rezk. On the Content Security Policy Violations Due to the Same-Origin Policy. In *Proceedings of the International Conference on World Wide Web*. ACM, 2017.

[25] S. Stamm, B. Sterne, and G. Markham. Reining in the Web with Content Security Policy. In *Proceedings of the International Conference on World Wide Web*. ACM, 2010.

[26] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns. Precise Client-Side Protection against DOM-Based Cross-Site Scripting. In *Proceedings of the USENIX Conference on Security Symposium*. USENIX, 2014.

[27] B. Stock, S. Pfistner, B. Kaiser, S. Lekies, and M. Johns. From Facepalm to Brain Bender: Exploring Client-Side Cross-Site Scripting. In *Proceedings of the Conference on Computer and Communications Security*. ACM, 2015.

[28] B. Terlson, B. Farias, and J. Harband. Ecmascript language specification. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf, 2020. (checked: February, 2020).

[29] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS*, 2007.

[30] T. Vyas, A. Marchesini, and C. Kerschbaumer. Extending the Same Origin Policy with Origin Attributes. In *Proceedings of the International Conference on Information Systems Security and Privacy*, 2017.

[31] W3C. Same-Origin Policy (SOP). https://www.w3.org/Security/wiki/Same_Origin_Policy, 2010. (checked: February, 2020).

[32] W3C. Content Security Policy (CSP). http://www.w3.org/TR/CSP2/, 2014. (checked: February, 2020).

[33] W3C. Mixed Content. https://www.w3.org/TR/mixed-content/, 2014. (checked: February, 2020).

[34] W3C. Subresource Integrity (SRI). https://www.w3.org/TR/SRI/, 2014. (checked: February, 2020).

[35] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc. CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In *Proceedings of the Conference on Computer and Communications Security*. ACM, 2016.

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

[37] WHATWG. DOM. https://dom.spec.whatwg.org/. (checked: February, 2020).

[38] WHATWG. Fetch. https://fetch.spec.whatwg.org/. (checked: February, 2020).

[39] WHATWG. HTML5. https://html.spec.whatwg.org/. (checked: February, 2020).