Strenghtening Content Security Policy via Monitoring and URL Parameters Filtering

Dolière Francis Somé
CISPA
Germany
doliere.some@cispa.saarland

Tamara Rezk
Inria
France
tamara.rezk@inria.fr

ABSTRACT

Content Security Policy (CSP) is a security mechanism for mitigating content injection attacks. It makes it possible to specify the origins of content allowed to load in a webpage. Upon enforcement, CSP-compliant browsers would block content not matching the CSP. Previous works have demonstrated limitations of CSP that can lead to security violations. We observe that CSP bypasses (due to JSONP and open redirects) can be linked to the fact that in CSP specification, URL parameters are considered safe by default. In particular, the ability to bypass partially whitelisted origins using HTTP redirections has been rendered possible starting from CSP2 for privacy purposes (not to reveal redirection URLs), while this can lead to security holes. In this work, we discuss 4 extensions to strengthen CSP via a monitoring mechanism: the ability to selectively exclude whitelisted content, express more fine grained checks on URL arguments, explicitly prevent redirections to partially whitelisted origins, and an efficient reporting mechanism to collect content that are allowed by a CSP enforced on a webpage. We show that using CSP along with these extensions improves the security of web applications and overcomes known weaknesses of the current CSP specification. We demonstrate the feasibility of our proposals by an implementation using service workers.

CCS CONCEPTS

• **Security and privacy** → **Browser security**; **Web application security**; **Access control**.

KEYWORDS

Content Security Policy; Same-Origin Policy; Service Workers

ACM Reference Format:
Dolière Francis Somé and Tamara Rezk. 2020. Strenghtening Content Security Policy via Monitoring and URL Parameters Filtering. In *19th Workshop on Privacy in the Electronic Society (WPES'20), November 9, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3411497.3420222

1 INTRODUCTION

Content Security Policy (CSP) is a defense-in-depth mechanism that has been introduced to mitigate the impact of content injection and data exfiltration attacks in web applications [36]. In particular, CSP can be used to mitigate Cross-Site Scripting (XSS), one of the most prevalent attacks in the web [4]. CSP is well supported by browsers, and its adoption by web applications is increasingly growing [14, 27, 29, 32, 34]. CSP is mostly an origin-based whitelisting mechanism [1], the set of origins of content allowed to load in a webpage are declared in the policy. Upon enforcement, browsers allow only content from the whitelisted origins, thereby blocking content not matching the policy. There are 2 main components provided by the specification for declaring policies: directives and directive values. Each directive is linked to a specific type of content (scripts, images, stylesheets, plugins, etc.), and directive values are the trusted origins from which the specific type of content are allowed to load.

```
script-src trusted.com;
object-src 'none';
```

Listing 1: Example of CSP

In the policy above, the directive `script-src` sets restrictions on scripts, and `object-src` sets restrictions on plugins. Scripts are allowed to load from `trusted.com` and no plugin is allowed to load [31, 35, 36].

Problem. Previous studies have demonstrated the limitations of CSP as a whitelisting mechanism, and the attacks that can be mounted to bypass CSP [20, 32]. Thus, they motivated the declaration in policies of individual content or set of content (partially whitelisted origins) instead of whitelisting entire origins, as the latters could host insecure JSONP endoinpts, open redirects or (untrusted) content such as the AngularJS library that can lead to CSP bypasses [20, 32]. Hence, Weichselbaum et al. [32] proposed the use of nonces to mitigate CSP bypasses based on open redirects and unsafe JSONP endpoints. However, this solution comes with the following issues. First, the security of nonces is questionable because they are included in the DOM of webpages [15, 35, 36]. Moreover, the use of nonces does not prevent a script that is already loaded in the webpage from making requests with JSONP parameters, or from redirecting to partially whitelisted origins, especially if the script gets compromised. Second, nonces apply only to scripts and stylesheets, and not to other types of content such as images whose URLs parameters an attacker can leverage to exfiltrate user data for instance. Compositional Content Security Policy (CCSP), a proposal of Calzavara et al. [15], also relies on individual whitelists, and therefore can lead to CSP bypasses in case of HTTP redirections.

[1]Individual content can also be whitelisted by nonces and hashes

We observe that CSP bypasses (due to JSONP and open redirects) can be linked to the fact that in CSP specification, URL parameters are considered safe by default. Therefore attackers can leverage them in order to bypass policies. Furthermore, when an origin (i.e. `trusted.com`) or specific path (i.e. `trusted.com/scripts/`) is whitelisted, it is not possible to exclude specific content (i.e. `trusted.com/untrusted.js` or `trusted.com/scripts/untrusted.js`) in a CSP may result in the policy being more permissive than expected, allowing attacker-injected content to load [14]. Furthermore, browser extensions are widespread on major browsers [2, 6, 8, 10]. In particular, they can inject in webpages their own content that is not always required to comply with the CSP of the page [3, 18]. Nonetheless, extensions content may further inject vulnerabilities in webpages, which are otherwise restricted by CSP.

Proposal. To address these issues, we propose to complement and strengthen CSP by a monitor that can effortlessly be included in client code delivered to the browser in order to disallow redirections, selectively exclude whitelisted content and URL parameters, and improve the feedback CSP reporting mechanism.

Disallowing redirections to partially whitelisted origins Partially whitelisted origins is the more reliable way of whitelisting origins. Nonetheless, this can be bypassed using HTTP redirections. We propose the specification of a new directive `disallow-redirects` that can be used in policies to instruct the monitor to prevent all redirections to partially whitelisted origins.

Adding checks on URLs parameters In CSP specification, URL parameters are considered safe by default, while they can be leveraged by attackers to bypass CSP, in presence of unsafe JSONP endpoints and open redirects. We propose extending the URL matching algorithm of CSP [31, 35, 36], which is used to check whether a URL is allowed by a policy or not, in order to take into consideration URL parameters. The proposed extension is to enable declaring origins, paths, and specific content by specifying the URL arguments that are trusted or untrusted. One can therefore ban parameters from whitelisted origins, whitelist specific parameters and blacklist others.

Selectively exclude whitelisted content Currently CSP specification defines 2 modes: the `report-only` mode in which policies are enforced, but browsers do not block content not allowed by the policy; and the `enforcement` mode in which content that are not allowed by the policy are effectively blocked. We refer to these two modes as CSP whitelisting modes. We propose to introduce a new header, `Content-Security-Policy-Blacklisting` to exclude specific content or set of content on a domain from loading in a webpage. `The blacklisting mode is meant to be used always as a complement of a CSP in whitelisting mode`. This is because, in general blacklisting alone is less reliable than whitelisting. That notwithstanding, CSP in blacklisting mode proves useful when one knows that a whitelisted origin hosts content that are potentially malicious, or endpoints whose URL parameters can be leveraged to bypass a policy. This new mode can also serve to explicitly prevent the loading of sensitive content in a webpage, as they may reveal information about a logged-in user for instance [17].

Efficient feedback reporting mechanism Finally, we complement CSP with a mechanism for collecting content that match a policy (aka feedback), similarly to CSP violations reports that can be collected using the `report-uri` and `report-to` directives. To specify the endpoints for collecting feedback, we introduce the directives `monitor-uri` and `monitor-to`. This feedback can be useful for many reasons. First, even if an application has been heavily tested, it is not excluded that an attacker can find a vulnerability and inject malicious content in the application. Moreover, an error

Implementation and evaluation. Our proposals add to the security of web applications since they allow to further restrict a policy in whitelisting mode, by preventing bypasses due to URL parameters or unsafe content. We implement the proposed extensions using service workers. Service workers (we refer to them as a monitor) intercept HTTP requests initiated by browsers to load content in a webpage. As such, they act like a proxy for content included in the page [12]. We deploy a service worker with an example web application, which deploys a CSP in enforcement mode and another policy in blacklisting mode to complement the policy in enforcement mode. The CSP in blacklisting mode is enforced by the monitor, and the CSP in whitelisting mode is enforced by the browser. Once the URL of a content matches a policy, the browser makes a request to fetch its content. Then, the request is sent to the service worker, which further checks its URL against the blacklisting policy. If the URL matches the blacklisting policy (either because it is a blacklisted content or carries untrusted arguments), then the request is blocked, otherwise it is effectively made. For open redirects, as HTTP redirections are not intercepted by service workers for security reasons [36], we could not fully implement the new `disallow-redirects` directive. Nevertheless, to prevent redirections to partially whitelisted origins, we assumed that all open redirects are known by the developer. Then we either used the new blacklisting mechanism to blacklist the open redirects, or prevented them from carrying URL parameters by filtering them out with the new URL parameters filtering mechanism we introduced. Finally, the monitor also logged all the URLs of content that it intercepted and reported them to the developer, as a feedback of the runtime enforcement of CSP. We evaluated the overhead that these extensions would introduce to web applications. The blacklisting mode has no overhead since it fully relies on CSP in whitelisting mode, already supported by browsers. The sole difference is in the final decision: a URL matching a CSP in blacklisting mode is blocked, while in a whitelisting mode it is allowed. Preventing redirections to partially whitelisted origins, in presence of the `disallow-redirects` directive, is achieved by implementing CSP1 [31] specification regarding redirections, without reporting the blocked redirection URL. We implemented the algorithm for checking URL parameters using a dozen lines of codes. Finally, to support the feedback mechanism, browsers would have to keep track of all content that match a policy, and report this to whatever endpoint specified by the `monitor-uri` or `monitor-to` directives.

Contributions. In summary, this paper contributes with four new extensions to address previous known and important weaknesses of the Content Security Policy. It aims at improving the security of web applications, by (i) expressing policies in blacklisting mode, (ii) filtering URL arguments, (iii) disallowing redirects to partially whitelisted origins and finally (iv) providing developers with an

Directive	Content type
`script-src`	scripts
`object-src`	plugins
`style-src`	stylesheets (CSS)
`connect-src`	XMLHttpRequest, ...
`img-src`	images

Table 1: Examples of CSP Directives

efficient way for collecting feedback about the runtime enforcement of their policies in an application.

2 MOTIVATING OUR PROPOSALS

A problem with the first proposal of CSP, CSP1, is that browsers cannot distinguish between legitimate and attacker-controlled inline scripts. To address this issue, starting from CSP2 [36], nonces and hashes have been introduced to allow the whitelisting of individual legitimate inline scripts, so as to distinguish them from malicious ones. So far, however, there is no way to whitelist individual DOM event handlers using nonces, or hashes [2]. Such scripts have to be externalized in files, and hosted on origins that will be further whitelisted in the CSP of the application. Nonces can also be used to whitelist individual external or remote scripts. One of the reasons that hindered a wider adoption of CSP was the lack of a mechanism for easily accommodating dynamically injected scripts. Weichselbaum et al. [32] then proposed the `'strict-dynamic'`, a keyword to be used to allow scripts whitelisted with nonces or hashes, to further dynamically inject additional scripts even if they are not explicitly whitelisted in a policy.

There are 2 main components in CSP for declaring policies. Directives target a specific type of content (scripts, images, plugins, etc.) (See Table 1), and directive values are the trusted origins from which content can be loaded. This includes origins (i.e. `trusted.com`, `*.trusted.com`), schemes (i.e. `https:`, `data:`), keywords (i.e. `'self'`, `'unsafe-inline'`, `'strict-dynamic'`), nonces and hashes [35, 36].

For the sake of simplicity and throughout the rest of this work, we describe in more details the issues we addressed by considering mostly the `script-src` directive, which sets restrictions on the origins from which trusted scripts can load. Nonetheless this work is more general, and concerns CSP as a whole, its other directives and content types. To illustrate the limitations of CSP and motivate our proposals, we consider the following policies.

```
script-src https://trusted.com https://redirect.
    com https://partials.com/scripts/;
img-src trusted.com/image.png;
```

Listing 2: Example of an origin-based CSP

```
default-src 'none'; form-action 'none';
frame-ancestors 'none'; report-uri /allcontent
```

Listing 3: Restrictive policy in report-only mode to get the list of content loaded in a webpage

[2]There is draft proposal in the current CSP3 [35], to be able to use hashes for whitelisting individual DOM event handlers. This mechanism is however not yet implemented by browsers

```
script-src 'nonce-random1234' 'strict-dynamic'
```

Listing 4: Example of CSP with nonces, or nonce-based policy

Listing 2 presents a CSP where trusted scripts are whitelisted by their origins. We refer to them as origin-based policies. Only scripts from the explicitly specified origins are allowed to load in the webpage on which this policy will be deployed. The injection of a script with the URL https://trusted.com/script.js in the webpage is allowed since the script comes from the whitelisted origin https://trusted.com. Listing 3 presents a restrictive policy that basically prevents a page from loading content. Listing 4 presents a nonce-based policy. When a nonce-based policy makes use of the `'strict-dynamic'` keyword, we refer to the overall policy as a strict CSP. Nonces are used to whitelist individual scripts. To allow a script to load, one injects `<script src="https://trusted.com/script.js" nonce="random1234"></script>` in the page. Note the use of the `nonce` attribute on the script tag. Its value is the nonce whitelisted in the policy (See Listing 4). With the presence of the `'strict-dynamic'` keyword, whitelisted scripts (with nonces) that load in the page can further dynamically inject additional scripts, even though the additional scripts are not assigned whitelisted nonces [3]. Hence, contrary to the origin-based CSP in Listing 2 where one knows the exact origins from which content can load, in the case of strict CSP in Listing 4, scripts that effectively load are known only at runtime (when the page is loaded in a browser and the policy enforced).

2.1 Partially whitelisted origins

In the CSP of Listing 2, from the domain https://partials.com, only scripts with the path `/scripts/` (for instance https://partials.com/scripts/a.js) are trusted. Hence, trying to inject https://partials.com/script.js will fail. Nonetheless, one can get this script loaded and executed if it is loaded as the result of an HTTP redirection [35, 36]. To illustrate the bypass of partially whitelisted origins, let's assume that the origin https://redirect.com, which is also whitelisted in the CSP of Listing 2, hosts an open redirect endpoint https://redirect.com/r. In other words, instead of directly injecting https://partials.com/script.js, one passes the URL of the script as an argument to the open redirect endpoint by injecting a script with the URL https://redirect.com/r?url=https://partials.com/script.js. Instead of returning a script to be executed, the open redirect generates an HTTP redirection `Location: https://partials.com/script.js`. Since this is an HTTP redirection, the browser will not check whether the whole URL matches the CSP. It is sufficient that the origin of the request be fully or partially whitelisted in the CSP of the page for the script to be allowed via the HTTP redirection. And since this is the case (See Listing 2), then the script https://partials.com/script.js is allowed to load, even though its URL does not match the CSP. This bypass works in CSP2 [36] and CSP3 [35].

2.2 Excluding content from whitelisted origins

Now let's assume that from the CSP of Listing 2, most of the scripts from the https://trusted.com origin are trusted. However, the origin also hosts the insecure script https://trusted.com/untrusted.js and hosts sensitive scripts in the `/admin/` folder (scripts which paths start with https://trusted.com/admin/) that must be not be loaded in

[3]Nonces and hashes work quite similarly [35, 36]

the current page. CSP does not provide a mechanism for excluding content from an origin. When an origin is whitelisted, it is trusted in its entirety.

2.3 URL parameters

URL parameters are not taken into consideration when browsers match a URL against a policy. Consider the CSP of Listing 2, the URLs https://trusted.com/script.js and https://trusted.com/script.js?func=eval&arg=1 all match the CSP if they are used to inject a script, and the URL https://trusted.com/image.png?data=someuserdata&cookie=usercookies matches the policy it is the URL of an image injected in the page. In the first case, the URL does not have any parameter. In the second case, the same URL is provided the parameters `func` with the value `eval` and `arg` with the value `1`. If according to CSP, these 2 URLs are exactly the same, in practice they may result in the execution of completely different content. Considering the second case, if the parameters provided are used to generate the response which is returned back, we run into JSONP requests which can lead to CSP bypass [20, 32]. In the third case, the URL to load the image is passed some user data and cookies as parameters, so that they are exfiltrated to `trusted.com`.

2.4 CSP violations

CSP can be used in 2 modes. In the report-only mode, policies are delivered to browsers using the `Content-Security-Policy-Report-Only` header. In this mode, content not matching the policy are not blocked by the browser. They are simply reported as CSP violations to the developer. In the dual enforcement mode on the other hand, policies are delivered to browsers using the `Content-Security-Policy` header. When browsers enforce such a policy, content that do not match the policy are effectively blocked, and a violation report is also sent. CSP allows to combine policies in different modes, or even deploy multiple policies in the same mode. Multiple policies are all enforced individually. In this case, a resource is allowed to load if it is allowed by all the policies. The directives `report-uri` (in CSP1, CSP2) and `report-to` (in CSP3) are used in a policy to indicate where CSP violations will be submitted to [35, 36].

The violations report mechanism of CSP can be used to build the list of content that load in a webpage. To do so, one has to deploy 2 policies: a policy in enforcement mode (as the ones in Listing 2 and 4), and a policy in report-only mode that does not allow any content, as the policy shown in Listing 3. Since the policy is in report only mode, any content that attempts to load in the webpage will trigger a CSP violation. Hence, every content triggers a violation. It is therefore impossible to distinguish between malicious and trusted content by analyzing the violations reported by a single policy in report-only mode. So one has to also collect violations triggered by the enforcement of the policy in enforcement mode deployed to effectively prevent malicious content from loading (i.e. policies shown in Listing 2 and Listing 4). Hence violations in this policy are triggered only by content not matching the policy. Computing the difference between the 2 reports then gives the content that effectively loaded because they are allowed by the CSP of the webpage. It is worth mentioning the case of browser extensions, whose content are not always subject to the CSP of the page [3, 18]. For instance, if the policy in Listing 2 is deployed on a webpage, this does not prevent a browser extension from injecting a script with the URL https://untrusted.com/vulnerable.js, even if this URL is not allowed by the policy. Worryingly, the browser extension may be injecting a content that introduces vulnerabilities in the webpage. Moreover, the injection of this script will not trigger a CSP violation report, even in presence of a CSP in report-only mode as the one in Listing 3.

2.5 (In-)Security of CSP Nonces

To help mitigate the CSP bypasses due to JSONP and open redirects, Weichselbaum et al. [32] suggested the use of nonces for whitelisting individual scripts instead of whitelisting the origins, URLs or path to the scripts. Nevertheless, recent studies question the security of nonces, mostly because nonces are included in the DOM of webpages, and thereby are subject to leakage by scriptless attacks [15]. Moreover, the use of nonces does not prevent a script which is already loaded in a webpage from making requests with unsafe JSONP parameters, using open redirects, or loading untrusted content. If a whitelisted script gets compromised by an attacker, then he can bypass the CSP at will. The use of nonces apply only to scripts (`script-src`) and stylesheets (`style-src`) content types, and not to other types of content. The violation reporting mechanism is more indicated for violations that occur from time to time, and is not suited for efficiently collecting feedback about content that load in a webpage. Moreover, it does not report content injected by browser extensions. Also when the page deploys a strict CSP, collecting feedback is useful because one cannot know in advance the content allowed by the strict CSP before it is enforced.

2.6 Our proposal

To successfully address the aforementioned issues, we propose to complement CSP with (i) a blacklisting mode, (ii) a URL arguments filtering mechanism, (iii) new directives for preventing redirections to partially whitelisted origins and (iv) a mechanism for efficiently collecting feedback about the runtime enforcement of the policy of a webpage by browsers.

3 MONITOR EXTENSION FOR CSP

In this section, we introduce our extensions on top of CSP.

3.1 CSP in blacklisting mode

Similarly to the headers used for deploying a CSP in enforcement and report-only modes respectively, we propose a new header `Content-Security-Policy-Blacklisting` for deploying CSP in blacklisting mode. Semantically, the blacklisting mode is the exact opposite of the enforcement mode. Hence, when a URL matches a CSP in blacklisting mode, then it is not allowed to load. Consider the policy in Listing 5.

```
script-src cdn.cloudfare.com/angular.js;
```

Listing 5: A CSP in blacklisting mode to exclude `angular.js`

In enforcement mode, this policy would have allowed the `angular.js` script to load. By deploying the policy in blacklisting mode, then the script is blacklisted and hence not allowed to load. One can therefore combine this policy with another policy in enforcement mode to prevent only `angular.js` from loading, while allowing any other content from cdn.cloudfare.com to load. Listing 6 presents the two policies.

```
Content-Security-Policy: script-src cnd.cloudfare.
    com
Content-Security-Policy-Blacklisting: script-src
    cdn.cloudfare.com/angular.js
```

Listing 6: Combining 2 policies: one in enforcement mode, and the other one in blacklisting mode

3.2 Checks on URL arguments

To illustrate this proposal, let's consider the following scenario. Listing 7 shows an example of a JSONP endpoint which expects the parameter `callback` and uses it to generate a function call, and passes it data.

```
Content-Security-Policy: script-src jsonp.com
```

Listing 7: CSP with insecure JSONP endpoint

If an attacker injects http://jsonp.com/?callback=eval in a webpage, the returned response is a function call to `eval(...)`. Note that the URL argument is used to generate the function call. To prevent the URL from loading when it is passed the `callback` parameter, one could also deliver a CSP in blacklisting mode as shown in Listing 8 in addition to the CSP in Listing 7 that allows parameters to be passed to the insecure JSONP endpoint.

```
Content-Security-Policy-Blacklisting: script-src
    jsonp.com/?callback
```

Listing 8: Supporting URL parameters in CSP

The CSP in blacklisting mode mandates that, for URLs to load content from jsonp.com, they must not have the argument `callback`. While the first policy only (Listing 7) would have allowed http://jsonp.com/?callback=eval to load, deploying also the second policy in blacklisting mode (Listing 8) would block it. By enforcing the CSP in blacklisting mode, one detects that the URL of the resource to load has an argument, whose name is `callback`. Therefore the URL is blocked. Note that this design does not prevent the webpage from loading other content from jsonp.com. For instance, it is completely possible to load http://jsonp.com/script.js, but not to load http://jsonp.com/script.js?callback=foo. If the script has some other arguments, then they are allowed to load. For instance, loading http://jsonp.com/script.js?foo=bar is allowed by the policy above.

We have shown how to prevent URLs with a specific argument (`callback` in our example). Now we illustrate additional scenarios on how to filter URLs parameters.

Blocking all URLs with arguments To prevent URLs with arguments, one simply ends their origins (paths or URLs) with ? in a blacklisting CSP.

```
Content-Security-Policy: script-src jsonp.com
Content-Security-Policy-Blacklisting: script-src
    jsonp.com/?
```

Listing 9: Blocking all URLs with arguments

The policy in Listing 9 stipulates that arguments are not allowed on URLs of scripts from jsonp.com. Without the CSP in blacklisting mode, any URL from jsonp.com would have been allowed. Now, the second policy in blacklisting mode will block URLs with parameters.

Blacklisting URLs when they have specific argument value One can go even more fine-grained, by blocking URLs only when they have specific parameters that have specific values. Listing 10 shows how to blacklist URLs from jsonp.com when the have the argument `callback` with the value `eval`.

```
Content-Security-Policy: script-src jsonp.com
Content-Security-Policy-Blacklisting: script-src
    jsonp.com/?callback=eval;
```

Listing 10: Blacklisting URLs with specific argument names and values

While the CSP in blacklisting mode selectively prevents http://jsonp.com/script.js?callback=eval to load, it will however not prevent http://jsonp.com/script.js?callback=alert from loading.

Specifying multiple unsafe arguments The different scenarios presented above can be combined to filter out URLs with a set of unsafe arguments. If multiple arguments are specified for an origin in a blacklisting policy, then a URL is blocked if it has all the blacklisted arguments. Listing 11 is a policy which blocks URLs having both the `callback` and `arg` arguments with any values.

```
Content-Security-Policy: script-src jsonp.com
Content-Security-Policy-Blacklisting: script-src
    jsonp.com/?callback&arg;
```

Listing 11: Blacklisting URLs with multiple unsafe arguments

This policy will block http://jsonp.com/script.js?callback=eval&arg=1, but not http://jsonp.com/script.js?callback=eval, because the first URL has both unsafe parameters, while the second one does not.

Blocking URLs with at least one untrusted argument To block URLs if they have at least one argument among a set of unsafe arguments, one can declare the blacklisting policy as shown in Listing 12.

```
Content-Security-Policy: script-src jsonp.com
Content-Security-Policy-Blacklisting: script-src
    jsonp.com/?callback jsonp.com/?arg;
```

Listing 12: Blacklisting URLs with at least one of multiple unsafe arguments

A URL will be blocked if it has either the argument `callback` or the `arg` with any values. Hence, http://jsonp.com/script.js?callback=eval and http://jsonp.com/script.js?arg=1 will be blocked, while http://jsonp.com/script.js?foo=bar will not be blocked.

3.3 Preventing redirections

In CSP1, when an origin is partially whitelisted, then only content that match the partially whitelisted origin can load. In CSP2 and CSP3 however, any content from partially whitelisted origins are allowed to load as the result of HTTP redirections [35, 36]. Here, we propose to extend the CSP specification so as to allow developers to explicitly prevent redirections to partially whitelisted origins. To do so, we propose a new directive `disallow-redirects`. When this directive is present in a policy, it prevents redirections to partially whitelisted origins. Consider the following policy

```
script-src https://partials.com/scripts/.js;
```

```
disallow-redirects;
```

Listing 13: CSP that uses `disallow-redirects` to explicitly prevent redirections to partially whitelisted origins

In case of an HTTP redirection (using an open redirect endpoint for instance), script from https://trusted.com/script.js would be allowed in CSP2 and CSP3. The new `disallow-redirects` directive in the policy instructs the browser to prevent these redirections to the partially whitelisted origins.

3.4 Reporting runtime enforcement of CSP

In CSP, to collect violation reports sent by the browsers, one must use the `report-uri` directive in CSP1 and CSP2, and the `report-to` directive in CSP3. As we have shown, only violations are reported to developers. Nonetheless, the content that actually load in webpages represent valuable information that can be used to improve the security of the application, by helping to deploy more secure policies. Therefore, following the semantics of the `report-to` and `report-uri` directives used for reporting violations, we propose the `monitor-uri` and `monitor-to` directives for reporting to developers content that effectively load within a webpage. When content are allowed to load within the page upon enforcement of a CSP, browsers would generate a report, following similar algorithm used for generating violations [31, 35, 36], and send this to the developer to whatever endpoint is specified in the `monitor-uri` or `monitor-to` directives. Listing 14 shows an example of a policy deployed to get feedback on the runtime enforcement of the policy, as well as collect CSP violations.

```
script-src trusted.com; object-src 'none';
report-uri /reports/violations;
monitor-uri /reports/feedback;
```

Listing 14: Policy to collect violations and feedback

In this example, the `monitor-uri` directive follows exactly the semantics of `report-uri` directive of CSP1 and CSP2.

3.5 Unsafe URL parameters

The algorithm for filtering out unsafe URL parameters is rather simple to implement. We provided an implementation of this additional algorithm in Section 4 using the URLSearchParams JavaScript API [11]. We refer to it as the URL parameters checker. It consists of a dozen lines of code. In doing so, we did not modify the URL matching algorithm itself. We rather implemented a dedicated function for matching CSP against URLs parameters. As such, we preserve backwards compatibility in browsers. An implementation of the URL parameters checker can be plugged into an already existing implementation of the URL matching algorithm [31, 35, 36] of CSP in order to further apply filtering on URLs arguments. Hence, after matching a URL against a request, the URL is passed to the parameters checker which further checks that the URL does not carry unsafe parameters. Otherwise the related content is blocked from loading.

4 IMPLEMENTATION

In this section, we demonstrate an implementation of the proposed extensions using service workers. Our implementation can be deployed on real world applications. We measure the overhead associated with deploying a service worker, which applies CSP in blacklisting mode, filters URLs arguments, and reports feedback, by using an example of web application.

In our implementation, we deploy a monitor. It acts like a proxy as shown by Figure 1. It intercepts requests made by the browser to load content in a web application. It is seamlessly and easily integrated to the application by the developer from the server-side, without requiring users or browsers to undertake any particular action. In addition to deploying a CSP in enforcement mode that will be enforced by the browser, the developer also deploys a CSP in blacklisting mode. In this policy, the developer can express fine-grained policies regarding unsafe URL parameters, partially whitelisted origins, blacklisted content and provide an endpoint where to submit feedback. So the CSP in enforcement mode is enforced by the browser, while the one in blacklisting mode is enforced by the monitor. When a content is injected in a webpage, first the CSP in enforcement mode is enforced by the browser. When a URL matches the policy, the browser makes a request to fetch its content. This request is intercepted by the monitor, and the CSP in blacklisting mode is applied. Upon enforcement, the monitor checks whether the URL is not a blacklisted one and does not carry unsafe parameters. In the particular case of partially whitelisted origins, HTTP redirections are not visible to service workers for privacy reasons [36]. Nevertheless, to implement the semantics of the `disallow-redirects` directive, one can use the blacklisting and URL filtering mechanisms to either blacklist open redirects endpoints or filter out their unsafe parameters. Then once a URL is allowed by the monitor upon enforcement of the policy in blacklisting mode, the request is made. Otherwise, it is blocked. The monitor also logs all requests that it intercepts, and reports them as feedback about the enforcement of CSP in the browser.

The monitor is not meant to replace the CSP enforcement provided by browsers. It rather complements it by filling the CSP expressiveness gap at fully mitigating bypasses. It adds a reporting mechanism for developers to get the set of content being loaded in the application.

4.1 URL filtering algorithm

Following we provide an example of implementation of URL arguments checker algorithm.

```
function unsafeArguments(origin, url){
  if(origin.indexOf("?") == -1)
    return true;
  var oArgs = origin.split("?").slice(1).join("?")
      ,
    uArgs = url.split("?").slice(1).join("?");
  if(!oArgs && !uArgs)
    return false;
  var oparams = new URLSearchParams(oArgs || ""),
    uparams = new URLSearchParams(uArgs || "");
  for(var it of oparams.keys()){
    if(!uparams.has(it)){
```

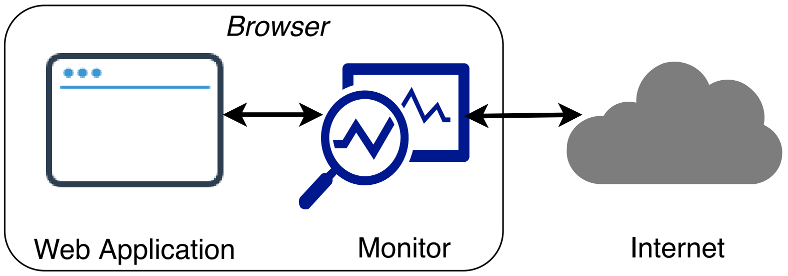

Figure 1: Monitoring CSP enforcement and applying additional checks on resources loaded in webpages after the browser has enforced CSP. In more details, the browser enforces the CSPs deployed on webpages and issues HTTP requests. The monitor (service worker) intercepts those requests and further checks and blocks requests whose URLs are blacklisted, or URLs that carry unsafe arguments (JSONP endpoints and open redirects). The monitor also records all intercepted HTTP requests and responses and reports them to the web application server as a feedback of the runtime enforcement of the CSP on the web application.

```
        return false;
      }else{
        var ovalue = oparams.get(it) || "",
          uvalue = uparams.get(it) || "";
        if(ovalue && ovalue != uvalue){
          return false;
        }
      }
    }
    return true;
}
```

Listing 15: Implementation of the URL arguments matching algorithm using the URLSearchParams API [11] in JavaScript

The implementation is done in JavaScript, using the URLSearch-Params API [11]. If the function returns `true`, then the URL is blocked, otherwise it is allowed. Recall that blacklisting URL arguments is meant to be used with CSP in blacklisting mode. So, the URL matching algorithm is first applied. Then, when the URL matches an origin in the blacklisted CSP, it is further passed to the arguments checker. If the origin of URL does not declare any unsafe arguments, it means that the URL must be blocked since it already matches the blacklisted origin. We can say that the blacklisting is at the origin level. Otherwise if the blacklisted origin specifies unsafe arguments, then the URL is blocked if its arguments match the blacklisted arguments of the origin. In this case, we can say the blacklisting is at the arguments level. Section 3.2 gives all the details about filtering out URL parameters.

Given an origin and a URL, the URL arguments checker checks whether the blacklisted arguments of the origin are found among the arguments of the URL. Table 2 shows the application of this algorithm on different origins and URLs. When there is a match between the origin and the URL, then the URL is blocked.

4.2 Service workers

Service workers [12] are implemented in major browsers including Chrome, Firefox, Opera, Microsoft Edge and Safari [4]. They act as a proxy, part of the application itself, which can however intercept all HTTP requests made by the browser to load content in an application. Service workers are deployed as part of the application, but once executed, will reside in the browser and intercept all requests going out of the application, as well as all incoming responses destined to the application. Service workers have been introduced among other things, to enable web applications to provide users with an offline experience when network is unavailable. It appears that they perfectly fit the needs of our monitor, and we use them to implement the latter. This quotation from Mozilla Developer Network defines very well service workers [12]

A service worker is an event-driven worker registered against an origin and a path. It takes the form of a JavaScript file that can control the web page/site it is associated with, intercepting and modifying navigation and resource requests, and caching resources in a very granular fashion to

[4]https://jakearchibald.github.io/isserviceworkerready/

Table 2: Matching arguments in an origin against arguments in a URL

Origin	URL	Match
a.com/?	a.com/s.js	no
a.com/?	a.com/s.js?func=eval	✓
a.com/?	a.com/s.js?func=eval&arg=hello	✓
a.com?func	a.com/s.js	no
a.com?func	a.com/s.js?arg=hello	no
a.com?func	a.com/s.js?func=alert	✓
a.com?func	a.com/s.js?func=alert&arg=hello	✓
a.com?func=eval	a.com/s.js	no
a.com?func=eval	a.com/s.js?arg=hello	no
a.com?func=eval	a.com/s.js?func=alert	no
a.com?func=eval	a.com/s.js?func=eval	✓
a.com?func=eval	a.com/s.js?func=eval&arg=hello	✓
a.com?func=eval&arg	a.com/s.js	no
a.com?func=eval&arg	a.com/s.js?func=eval&arg=hello	✓
a.com?func=eval&arg	a.com/s.js?func=alert&arg=hello	no

give you complete control over how your app behaves in certain situations, (the most obvious one being when the network is not available.)

First of all, the service worker itself is a JavaScript file which makes use of the specific APIs made available to it by browsers. Then the service worker is deployed by referencing it in the application whose requests it intercepts and manages.

Intercepting requests Following is how service workers intercept requests made from an application.

```
self.addEventListener('fetch', function(event) {
    url = event.request.url;
    content_type = event.request.destination;
    page = event.request.headers.referer

  }
);
```

Listing 16: Intercepting requests in service workers

To intercept the URLs of requests, service workers listen for `fetch` events, which are triggered each time that a request is initiated by the browser to load content in the application. Note that those requests are done after the browser enforces the CSP of the application on the URL of the content to load. The `request` object of the event contains all the information necessary to make the request (URL of the request, type of content being loaded, the specific page from which the request is being made, data sent along the request in case of HTTP POST requests, ...) [12]. As shown in Listing 16, `url` represents the URL of a request intercepted by the service worker, `content_type` the type of content that the URL will load (script, image, ...) [5], and `page`, the specific page of the application from which the request is being made. This helps for instance, to deploy a single service worker for an entire application made of multiple pages. The monitor specifically makes use of this three categories of information (URL of request, content type and URL of the page), which are sufficient for it to check whether the request of the particular type should be allowed or not, in the specific page. When the request is allowed, the monitor lets it proceed using the `fetch` API [5], as shown in the following Listing 17.

```
event.respondWith(
  return fetch(event.request).then(function(
      response) {
    return response;
  })
);
```

Listing 17: Making a request from the service worker

Otherwise, the request is blocked. This is achieved by generating and returning an empty response in the monitor, using the `Response` API [6].

```
event.respondWith(
  return new Response();
);
```

Listing 18: Blocking a request

[5]This information is not available on Firefox service workers
[6]https://developer.mozilla.org/en-US/docs/Web/API/Response

Deployment To deploy the service worker (monitor), one has to indicate to the browser the location (URL) of its code on the application server. Additionally, one indicates whether the monitor is deployed for a specific page or for an entire application (origin). To deploy service workers for an entire application, one can simply modify the main (HTML) page of the application, and the service worker will be deployed for the entire application.

```
...
<script>
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js', {
      scope: '/'})
  .then(function(reg) {
    console.log('Service Worker Started');
  }).catch(function(error) {
    console.log('Service Worker Failed');
  });
}
</script>
...
```

Listing 19: Deploying a service worker for an application

In Listing 19 above, `sw.js` is the JavaScript file of the service worker located on the application server, and the service worker is registered for the entire application `scope: '/'`.

4.3 Enforcement of the CSP in blacklisting mode

We have already described an implementation of the URL checker algorithm (See Listing 15), included in the service worker file that is deployed. When a web server sends a CSP in blacklisting mode with the response to a request to load a webpage, the monitor intercepts and saves on the fly the CSP in blacklisting mode. Then, when a request to load content on the page is intercepted (Listing 16), the URL of the request is checked against the blacklisting policy of the page. If the URL of request does not match the blacklisting policy, the request is normally made (Listing 17). Otherwise, the request is blocked. In this case, the monitor (service worker) returns an empty response (Listing 18).

In any case, requests that are intercepted, are logged. Requests that are blocked (because they are blacklisted content or because of their unsafe parameters) are also logged. This is submitted to the endpoint specified by the developer in the monitor policy. In our implementation, the reports are sent every 15 seconds [7]. That is the feedback about the enforcement of CSP on the application. In our implementation, we fix the URL where the feedback is sent to.

We provide online (Location blinded for review) our ready-to-deploy monitor and guidelines on how to easily integrate it to web applications.

5 EVALUATION

We deployed the monitor on an example web application (located on the localhost), with different types of content (scripts, images, stylesheets, fonts, XMLHttpRequests, etc.).

[7]We have chosen this number randomly, as all content on our example pages are loaded before this delay expires

The CSP of the page is shown in the following Listing 20.

```
script-src 'self' http://localhost:5000 http://
    localhost:7000
```

Listing 20: CSP in enforcement mode deployed on the webpage

It allows scripts of the site own origin, and from 2 third party origins. To simulate third party origins, we deployed multiple example applications on different ports of the localhost (ports 5000 and 7000). The application itself is deployed on port 8000. To further apply additional checks on URLs of requests to these origins (the 2 third party origins in particular), we deployed the CSP in blacklisting mode shown in Listing 21

```
script-src http://localhost:5000/ajax/libs/
    angular.js/ http://localhost:7000/?
```

Listing 21: Blacklisting CSP used to apply further checks on the content allowed by the CSP in enforcement mode

It blacklists (excludes) scripts whose paths start with http://localhost:5000/ajax/libs/angular.js/ from the origin http://localhost:5000 whitelisted in the CSP of Listing 20. URLs to http://localhost:7000 that carry any arguments will also be blocked by the monitor.

The deployed monitor has been able to successfully enforce the blacklisting policy on content we injected in the application. We also tested the monitor with nonce-based strict CSPs, and with toy browser extensions injecting content in the webpage. All such content have been successfully intercepted by the monitor and applied the CSP in blacklisting mode. Finally, the monitor was also able to log and report all content intercepted, blocked, and loaded. On a real web application, by analyzing the reported feedback, one may be able to detect potentially malicious or untrusted content loaded as a result of errors or misconfigurations, or as a result of an attacker exploiting a content injection vulnerability in the application. This also includes content dynamically injected in strict CSPs, and content injected by browser extensions.

The following is a report sent by the monitor upon enforcement of the blacklisting policy.

```
[
   {
      "url": "http://localhost:8000/script.js?
          BNfWB8tsrM",
      "type": "script",
      "blocked": false
   },
   {
      "url": "http://localhost:8000/script.js?
          K3Vc6ksIeV",
      "type": "script",
      "blocked": false
   },
   {
      "url": "http://localhost:7000/scripts/
          cspinclusion.js?callback=zleYLgrXNQ",
      "type": "script",
      "blocked": true
   },
   {
      "url": "http://localhost:5000/ajax/libs/
          angular.js/1.7.2/angular-animate.js",
      "type": "script",
      "blocked": true
   },
   {
      "url": "http://localhost:7000/scripts/
          cspinclusion.js",
      "type": "script",
      "blocked": false
   },
   {
      "url": "http://localhost:8000/script.js?
          AyhR4pkCaJ",
      "type": "script",
      "blocked": false
   },
   {
      "url": "http://localhost:8000/script.js?
          VPJS8xfJbn",
      "type": "script",
      "blocked": false
   },
   {
      "url": "http://localhost:7000/scripts/
          cspinclusion.js?callback=QZ4d2uiq8Y",
      "type": "script",
      "blocked": true
   },
   {
      "url": "http://localhost:7000/scripts/
          cspinclusion.js?callback=cmkRJvjPih",
      "type": "script",
      "blocked": true
   }
]
```

Listing 22: Feedback reported by the monitor

Entries in the report array with the `"blocked": true` property are content that are allowed by the CSP of the page as enforced by the browser, but blocked by the monitor after applying the blacklisting CSP.

5.1 Performance overhead

We evaluated the overhead introduced in a web application, with the use of a monitor (See Figure 2). To do so, our example webpage is embedding a set of content of different types. In particular it has 3 scripts for measuring the load time of the application:

- A first script which, when executed, registers the start time. It is the first script loaded in the webpage.
- A second script is responsible for dynamically loading in the webpage many content of different types (scripts, stylesheets, fonts, images, etc.).
- A third script injected at last, is responsible for measuring the end time. It is the last script executed in the webpage.

The page is composed of the following content:

- 20 scripts, each further making 1 synchronous XMLHttpRequest;
- 20 stylesheets, further loading 1 font each;
- 20 (JPG) images.

The application is served from the localhost to avoid latency and delay introduced with the network if it was deployed on a remote server. Time is measured using the `performance` [8] API, 100 times in different browsers (Chrome, Firefox, Opera, and Brave). All resources loaded are never cached, so that all measurements are done in the same conditions.

A Figure 2 shows, a first measurement is done when the application is not deploying any monitor (`No Monitor`). Then, another measurement is done when the monitor does not perform any action apart from simply forwarding all the requests that it intercepts (`Unenforced Monitor`) without applying the monitor policy. This is done to measure the overhead introduced by the use of service workers. Finally, a last measurement is done when the monitor enforces a blacklisting policy (`Enforced Monitor`).

The different times are shown in Figure 2 for Chrome browser, version 66, on an Intel(R) Core(TM) i7-4710MQ CPU @ 2.50GHz, 64 bits, with 16GB of RAM. Results in other browsers are similar and therefore omitted.

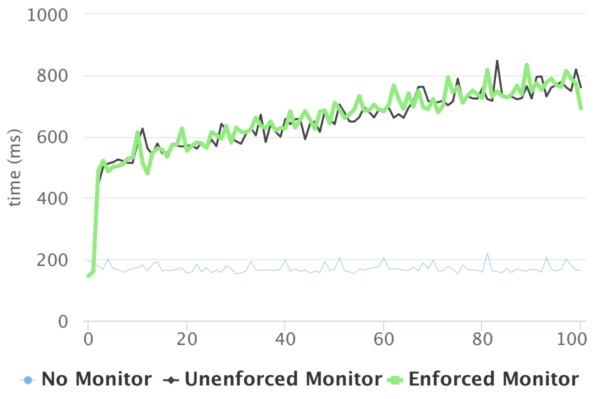

Figure 2: Performance overhead of deploying the monitor. As one can obverse, there is an observable overhead due to the use of service workers themselves (`No Monitor` vs. `Unenforced Monitor`). However the implementation of our monitor within the service worker itself induces very little overhead (`Unenforced Monitor` vs. `Enforced Monitor`).

As one may observe from Figure 2, the main overhead is due to the use of service workers to implement the monitor (`Unenforced Monitor` vs. `No Monitor`). Comparatively, enforcing the monitor policy itself introduced a negligible overhead (`Enforced Monitor` vs. `UnEnforced Monitor`). We think that this is an acceptable overhead, not only due to the security benefits that one gains with the deployment of the monitor, but because in our evaluation, we are not leveraging the full capabilities of service workers. In fact, in real world web applications, the Cache API of service workers can be used to cache resources, therefore reducing, or even completely removing this overhead.

[8]https://developer.mozilla.org/en-US/docs/Web/API/Performance

5.1.1 *Overhead of applying the CSP in blacklisting mode.*

We further measured the specific overhead of applying CSP within the monitor to blacklist content. To do so, we collected the CSP and scripts of 100k Alexa sites (home pages and up to 100 pages related [9] to the site). When the page had a CSP, we extracted the whitelisted origins of its `script-src` directive. This resulted in 6,481 unique sets of `script-src` directives and the associated values. We further gathered all the origins whitelisted in all `script-src` directives into a single set of unique origins, totaling 11,982 of them.

Then we randomly selected 6,481 scripts, corresponding to the number of unique `script-src` directives. To each script, using our implementation of the CSP URL matching algorithm, we applied all the 11,982 unique origins to check whether there was a match or not, and saved the time it took for the algorithm to terminate. Then, we compute the average time for applying all the 11,982 origins to the script. Figure 3 presents the results.

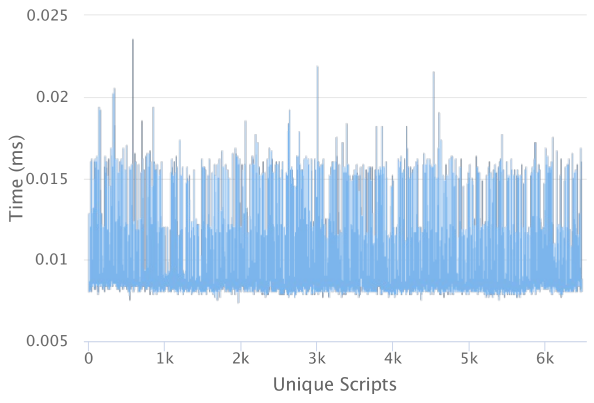

Figure 3: Overhead introduced by applying CSP to intercepted requests within the monitor. As one can observe, our implementation does not introduce any observable overhead (less than 2.5ms) in our examples evaluations.

The overhead introduced with applying CSP is really negligible. Assume that a directive contains 100 origins, in the worst case, the overhead introduced by applying CSP is less than 2.5 ms, which in our opinion, is acceptable, compared to the security benefits gained with deploying CSP in blacklisting mode.

6 DISCUSSIONS AND LIMITATIONS

Here we discuss the limitations of the service workers we used to implement the monitor in this work.

6.1 Service workers

Service workers is still a working draft at the W3C [10], even though it is already supported by major browsers, including Firefox, Chrome, Opera, Microsoft Edge and Safari. They are backwards compatible: browsers not supporting them will not deploy the monitor, without breaking the application. Developers do not have to serve specific versions of the application for browsers not supporting service

[9]Pages from the same origin as the site, and pages from a subdomain
[10]https://w3c.github.io/ServiceWorker/

workers. Deploying the monitor as shown in Listing 19 (Section 4) ensures its backwards compatibility. The only modification required is in the entry page to the application, where the monitor should be indicated, using an HTML script tag. Even though, using service workers introduce an overhead, there are many improvements which can compensate this overhead, in addition to the security benefits that one would gain by deploying them. Responses to requests can be cached in the monitor. Later on, when the application makes a request for the same content, it is retrieved from the cache and returned to the application. In this work, we have shown an implementation of the monitor using service workers. The monitor presents the advantage of laying outside of the browser enforcement of CSP. It only intercepts requests after they are allowed by the browser upon enforcement of the CSP of the page. This allows to further check for blacklisted content or content with unsafe arguments. In the monitor, we log requests, and can delay the report time. Nonetheless, service workers have limitations. They work only for secure (HTTPS) web applications (and the localhost). They do not work in Firefox private browsing mode. Service workers cannot intercept requests to load cross-origin iframes, for security reasons. Nonetheless, the monitor can be successfully deployed for content which executes in the context of the application such as scripts, plugins, stylesheets, images, etc.

Alternative methods can also be used to implement the monitor especially for browsers not yet supporting service workers: JavaScript proxies [7], or redefining JavaScript objects to intercept the injection of content [28]. To get all content that are injected in a webpage, similarly to a restrictive CSP in report-only mode (See Section 2), one can make use of Mutation Observers [9]. They allow to watch all changes made to the DOM of a webpage. As such, one can record all content that are injected in the webpage. These methods have many drawbacks. The first one is that they can potentially interfere with CSP enforcement as done by browsers. Moreover, contrary to these methods, service workers are easy to deploy, and can monitor all pages of entire web applications.

6.2 Browser extensions

Contrary to CSP in report-only mode, our monitor intercepts all content even those injected by browser extensions directly in the context of web applications. Content that browser extensions directly inject in the context of web applications [11], are usually not subject to the CSP of the page, and browsers would not block them. Such content are also intercepted in the monitor. If the browser allows them to load, even though they do not match the CSP of the page, the monitor instead would block them if they are not allowed by the blacklisting policy. Blocking such requests may however break extensions functionality. We did not assess how widespread this practice is among extensions, but applications developers have to take this into consideration when deploying our monitor. Should extensions content be blocked or not? Developers have to find a trade-off between the security of their applications and preserving the functionality of extensions [26].

[11] These are not content scripts, as content scripts execute in their own contexts. These are content further injected by content scripts directly in the context of web pages (See https://developer.chrome.com/extensions/content_scripts). In Chrome and Opera, even web accessible resources are also intercepted (See https://developer.chrome.com/extensions/manifest/web_accessible_resources)

6.3 Privacy implications of the reporting mechanism

In general, our proposal of monitoring the runtime enforcement of CSP, has to be discussed in the scope of browser extensions. Currently, a developer can observe content injected by browser extensions in web applications, by inspecting the DOM, setting up a Mutation Observer [9] or deploying a service worker as we have done. Since content injected in web pages by browser extensions are visible to web pages, we argue that browser vendors may also report such content to developers when reporting the runtime enforcement of CSP. Therefore reporting content does not leak any further information than what could already be obtained with the examples of techniques given above. Our proposal is just an efficient way for getting feedback, without relying on the techniques presented here, given their limitations.

There are however cases where extension developers would like to hide their injected content from web applications. For instance, in Firefox, injecting browser extensions own content, called web accessible resources, leak the extension unique identifier, which is unique on a per user basis. If this identifier is leaked to the web application, it can serve to uniquely identify and track her in future browsing sessions, as the identifier is unique for the extension and does not change throughout browsing sessions [25]. In general, it is difficult to hide this identifier from web applications. Setting up a mutation observer allows to intercept the identifier. We think that such content must also be reported as they can already be observed by different means.

There is however one case, recommended to prevent leaking extension's identifiers, in the particular case of iframes injection, as discussed on Bugzilla [1].

```
var f = document.createElement("iframe");
document.body.appendChild(f);
f.contentWindow.location = chrome.extension.getURL
    ("iframe.htm");
```

As shown in the listing above, one can inject an iframe without leaking the unique identifier of the extension. From a mutation observer, is is not possible to observe the URL of such an iframe, and scripts running in the page cannot also observe it. Finally, service workers cannot intercept the URLs of cross-origin iframes. In this situation, and for the sake of user privacy, one may argue that the monitor of the runtime enforcement of CSP must not report the URLs of iframes included as such. We think that since such content are included in the webpage, they must also be reported.

7 RELATED WORK

Content Security Policy has been first proposed by Stamm et al. [30], and standardized by the W3C. Since its introduction, CSP has gained significant consideration from the research community, with propositions aimed at improving its effectiveness and security [31, 36, 36]. It is supported by major browsers, and its adoption on websites in the wild is growing, even though slowly [14, 29, 32, 34]. To help improve CSP adoption, many tools have been proposed [21, 23, 24].

Martin Johns [20] then Weichselbaum et al. [32] reported CSP bypasses due to JSONP, open redirects, and symbolic execution enabled by libraries such as AngularJS library. Weichselbaum et al. [32] proposed the use of nonces for whitelisting scripts. Because an attacker cannot guess in advance which nonces will be assigned to trusted scripts, they cannot inject untrusted content, or abuse JSONP and open redirects to load arbitrary content in the application. They also first proposed the `'strict-dynamic'` keyword to be used in `script-src` directive, for easily loading dynamic scripts. In this setting, a whitelisted script is allowed to further load any additional script. However, nonces are included in the DOM, and their security is questionable. Furthermore, the trust propagation enabled by `'strict-dynamic'` only applies to scripts and stylesheets, and is too liberal since it allows any whitelisted script to further inject any other script without restrictions. To limit this trust propagation, Calzavara et al. [15] proposed Compositional Content Security Policy (CCSP). In their proposal, scripts which are included in the application are individually whitelisted in the CSP of the application, instead of whitelisting their origins. Furthermore, each of them is assigned an upper bound in the additional content it can further inject in the application. The upper bound is a CSP specifying which additional content a whitelisted script can further load. Besides requiring significant modifications in the current CSP specification, CCSP also requires content providers to declare all the dependencies needed by content that they host. This helps developers build the upper bounds when including such content in their policies. CSP is a page-specific policy. Collin Jackson and Adam Barth [19] have shown that page-specific policies can be bypassed by origin-wide policies. Somé et al. [29] demonstrated that this also applies to CSP, by analyzing the interactions between CSP and the Same Origin Policy (SOP). They showed that because CSP is a page-based policy, while the SOP allows all same-origin pages to access each other's execution context, CSP can be bypassed by scripts running in other same origin pages, even though they are not explicitly whitelisted in the CSP-protected page. Many studies have analyzed the effectiveness of CSP at mitigating content injection attacks. Most of them report that the majority of CSP deployed, are ineffective because they are liberal, allow inline scripts and related unsafe APIs [14], insecure JSONP endpoints or more subtle bypasses [32], DNS and resource prefetching [13]. Hausknecht et al. [18] found many browser extensions tampering with CSP, leading to enforced policies being more or less permissive, and thus ineffective against attacks. More recently, Calzavara et al. [16] presented a formalization of CSP semantics, especially the directives values.

8 CONCLUSION

In this work, we propose four new extensions to complement via monitoring the current CSP specification: a new blacklisting mode, the ability to blacklist content based on unsafe URL arguments, new directives for explicitly preventing redirections to partially whitelisted origins and an efficient monitoring mechanism for collecting feedback of the runtime enforcement of CSP. We implement the new extensions using service workers, to monitor and intercept content that load on the policy, and apply additional checks on the URLs of content. We then evaluated the overhead of deploying such a policy on a web application. The monitor is easily integrated to the application by the developer from the server-side, without requiring users or browsers to undertake any particular action.

Although our current monitor is completely independent of CSP and browser implementations, these extensions have the potential to be integrated in future versions of CSP and be implemented directly in browsers. Our extensions would only introduce a few modifications to the implementations of CSP in browsers. For instance, the blacklisting mode does not require browsers to support a new algorithm, but uses exactly the URL matching algorithm already implemented by browsers [35, 36]. There is only need for supporting a new CSP header. If the browser already implements CSP, it just enforces a blacklisting policy as a normal one. The sole difference is in the final decision: when a URL matches a blacklisting policy, the URL is not allowed, while in whitelisting mode, it is allowed. The only modifications needed to the CSP URL matching algorithm are those to support the URL parameters filtering mechanism and the new directives. These extensions are all backwards compatible. Hence they do not break the current state of the specification, nor do they require significant modifications from current browsers implementations of the specification.

REFERENCES

[1] [n.d.]. Bug 1372288 - WebExtensions UUID can be used as user fingerprint. https://bugzilla.mozilla.org/show_bug.cgi?id=1372288
[2] [n.d.]. Chrome Extensions. https://chrome.google.com/webstore/category/extensions?hl=en-US
[3] [n.d.]. Chrome Extensions API - Content scripts and Content Security Policy. https://developer.chrome.com/extensions/contentSecurityPolicy
[4] [n.d.]. Cross-Site-Scripting. https://www.owasp.org/index.php/Cross\protect\discretionary{\char\hyphenchar\font}{}{}site_Scripting_(XSS).
[5] [n.d.]. Fetch Specification. https://fetch.spec.whatwg.org/.
[6] [n.d.]. Firefox Add-ons. https://addons.mozilla.org/en-US/firefox/
[7] [n.d.]. JavaScript Proxy. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy
[8] [n.d.]. Microsoft Edge Extensions. https://www.microsoft.com/en-us/store/collections/edgeextensions/pc
[9] [n.d.]. MutationObserver API . https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver
[10] [n.d.]. Opera Add-ons. https://addons.opera.com/en/extensions/
[11] [n.d.]. URLSearchParams API. https://developer.mozilla.org/en-US/docs/Web/API/URLSearchParams
[12] [n.d.]. Using Service Workers. https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers
[13] Steven Van Acker, Daniel Hausknecht, and Andrei Sabelfeld. 2016. Data Exfiltration in the Face of CSP. In Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016, Xiaofeng Chen, XiaoFeng Wang, and Xinyi Huang (Eds.). ACM, 853–864. https://doi.org/10.1145/2897845.2897899
[14] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. 2016. Content Security Problems?: Evaluating the Effectiveness of Content Security Policy in the Wild, See [33], 1365–1375. https://doi.org/10.1145/2976749.2978338
[15] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. 2017. CCSP: Controlled Relaxation of Content Security Policies by Runtime Policy Composition, See [22], 695–712. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/calzavara
[16] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. 2017. Semantics-Based Analysis of Content Security Policy Deployment. ACM Trans. Web 12, 2, Article 10 (Jan. 2017), 36 pages. https://doi.org/10.1145/3149408
[17] Gábor György Gulyás, Dolière Francis Somé, Nataliia Bielova, and Claude Castelluccia. 2018. To Extend or not to Extend: on the Uniqueness of Browser Extensions and Web Logins. In To appear in the Proceedings of the 2018 ACM on Workshop on Privacy in the Electronic Society, WPES@CCS 2018, Toronto, Canada, October 15 - 19, 2018.
[18] Daniel Hausknecht, Jonas Magazinius, and Andrei Sabelfeld. 2015. May I? - Content Security Policy Endorsement for Browser Extensions. In Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings (Lecture Notes in Computer Science), Magnus Almgren, Vincenzo Gulisano, and Federico Maggi (Eds.), Vol. 9148. Springer, 261–281. https://doi.org/10.1007/978-3-319-20550-2_14
[19] Collin Jackson and Adam Barth. 2008. Beware of Finer-Grained Origins. In Web 2.0 Security and Privacy (W2SP 2008). https://seclab.stanford.edu/websec/

origins/fgo.pdf

[20] Martin Johns. 2014. Script-templates for the Content Security Policy. J. Inf. Sec. Appl. 19, 3 (2014), 209–223. https://doi.org/10.1016/j.jisa.2014.03.007

[21] Christoph Kerschbaumer, Sid Stamm, and Stefan Brunthaler. 2016. Injecting CSP for Fun and Security. In Proceedings of the 2nd International Conference on Information Systems Security and Privacy, ICISSP 2016, Rome, Italy, February 19-21, 2016., Olivier Camp, Steven Furnell, and Paolo Mori (Eds.). SciTePress, 15–25. https://doi.org/10.5220/0005650100150025

[22] Engin Kirda and Thomas Ristenpart (Eds.). 2017. 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. USENIX Association. https://www.usenix.org/conference/usenixsecurity17

[23] Xiang Pan, Yinzhi Cao, Shuangping Liu, Yu Zhou, Yan Chen, and Tingzhe Zhou. 2016. CSPAutoGen: Black-box Enforcement of Content Security Policy upon Real-world Websites, See [33], 653–665. https://doi.org/10.1145/2976749.2978384

[24] Kailas Patil and Braun Frederik. 2016. A Measurement Study of the Content Security Policy on Real-World Applications. I. J. Network Security 18, 2 (2016), 383–392. http://ijns.femto.com.tw/contents/ijns-v18-n2/ijns-2016-v18-n2-p383-392.pdf

[25] Iskander Sánchez-Rola, Igor Santos, and Davide Balzarotti. 2017. Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies, See [22], 679–694. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/sanchez-rola

[26] Alexander Sjösten, Steven Van Acker, and Andrei Sabelfeld. 2017. Discovering Browser Extensions via Web Accessible Resources. In Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ, USA, March 22-24, 2017, Gail-Joon Ahn, Alexander Pretschner, and Gabriel Ghinita (Eds.). ACM, 329–336. https://doi.org/10.1145/3029806.3029820

[27] Dolière Francis Somé, Nataliia Bielova, and Tamara Rezk. [n.d.]. Use of Content Security Policy and Cookies in Top 10,000 Alexa Sites. https://webstats.inria.fr/popsecurity.php

[28] Dolière Francis Somé, Nataliia Bielova, and Tamara Rezk. 2017. Control What You Include! - Server-Side Protection Against Third Party Web Tracking. In Engineering Secure Software and Systems - 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings (Lecture Notes in Computer Science), Eric Bodden, Mathias Payer, and Elias Athanasopoulos (Eds.), Vol. 10379. Springer, 115–132. https://doi.org/10.1007/978-3-319-62105-0_8

[29] Dolière Francis Somé, Nataliia Bielova, and Tamara Rezk. 2017. On the Content Security Policy Violations due to the Same-Origin Policy. In Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017, Rick Barrett, Rick Cummings, Eugene Agichtein, and Evgeniy Gabrilovich (Eds.). ACM, 877–886. https://doi.org/10.1145/3038912.3052634

[30] Sid Stamm, Brandon Sterne, and Gervase Markham. 2010. Reining in the web with content security policy. In Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010, Michael Rappa, Paul Jones, Juliana Freire, and Soumen Chakrabarti (Eds.). ACM, 921–930. https://doi.org/10.1145/1772690.1772784

[31] Brandon Sterne and Adam Barth. 2012. Content Security Policy 1.0. W3C Candidate Recommendation. http://www.w3.org/TR/2012/CR-CSP-20121115/

[32] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. 2016. CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy, See [33], 1376–1387. https://doi.org/10.1145/2976749.2978363

[33] Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). 2016. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. ACM. http://dl.acm.org/citation.cfm?id=2976749

[34] Michael Weissbacher, Tobias Lauinger, and William K. Robertson. 2014. Why Is CSP Failing? Trends and Challenges in CSP Adoption. In Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings (Lecture Notes in Computer Science), Angelos Stavrou, Herbert Bos, and Georgios Portokalidis (Eds.), Vol. 8688. Springer, 212–233. https://doi.org/10.1007/978-3-319-11379-1_11

[35] Mike West. 2016. Content Security Policy Level 3. W3C Working Draft. http://www.w3.org/TR/CSP3/

[36] Mike West, Adam Barth, and Dan Veditz. 2015. Content Security Policy Level 2. W3C Candidate Recommendation.