May I? - Content Security Policy Endorsement for Browser Extensions

Daniel Hausknecht[1], Jonas Magazinius[1,2,3], and Andrei Sabelfeld[1]

[1] Chalmers University of Technology
[2] CISPA, Saarland University
[3] Assured Security AB

Abstract. Cross-site scripting (XSS) vulnerabilities are among the most prevailing problems on the web. Among the practically deployed countermeasures is a“defense-in-depth” Content Security Policy (CSP) to mitigate the effects of XSS attacks. However, the adoption of CSP has been frustratingly slow. This paper focuses on a particular roadblock for wider adoption of CSP: its interplay with browser extensions.
We report on a large-scale empirical study of all free extensions from Google's Chrome web store that uncovers three classes of vulnerabilities arising from the tension between the power of extensions and CSP intended by web pages: third party code inclusion, enabling XSS, and user profiling. We discover extensions with over a million users in each vulnerable category.
With the goal to facilitate a wider adoption of CSP, we propose an extension-aware CSP endorsement mechanism between the server and client. A case study with the Rapportive extensions for Firefox and Chrome demonstrates the practicality of the approach.

1 Introduction

Cross-site scripting (XSS) [27] vulnerabilities allow attackers to inject malicious JavaScript for execution in the browsers of victim users. XSS vulnerabilities are among the most prevailing problems on the web [28]. The *World Wide Web Consortium (W3C)* [38] has proposed a“defense-in-depth” *Content Security Policy (CSP)* [36] to mitigate the effects of XSS attacks. A CSP policy lets websites whitelist a set of URIs which are accepted as the sources for content on a web page. The standard defines CSP to be transmitted in an HTTP header to the client, where it is enforced by a CSP compliant user agent. The browsers enforce the policy by disallowing communication to the hosts outside the whitelist. The majority of the modern browsers support CSP [35].

The web application security community largely agrees on the usefulness of CSP [26,40] as an effective access control policy not only to mitigate XSS but also other cross-domain attacks such as *clickjacking* [25]. However, the adoption of CSP has been frustratingly slow. Major companies lead the way, e.g. with Google introducing CSP for Gmail in December 2014 [14], yet, currently, only 402 out of the top one million websites actually specify a policy for their websites [6].

CSP and browser extensions. This paper focuses on what we believe is a serious roadblock for wider adoption of CSP: its interplay with browser extensions. Browser extensions often rely on communication with websites, fetching their own scripts, external libraries, and integrating independent services. For example, the extension *Rapportive* [20], with over 320,000 users, shows information from LinkedIn about the contacts displayed in Gmail. The functionality of this extension depends on the ability to communicate with LinkedIn. This is in dissonance with Gmail's CSP, where, not surprisingly, LinkedIn is not whitelisted.

As mentioned above, Google recently started enforcing a CSP policy for Gmail. To be more precise, Google changed CSP from *report-only* to enforcement mode [14]. This resulted in immediate consequences for both the Firefox and Chrome versions of the Rapportive extension. Interestingly, the consequences were different for Firefox and Chrome. The Firefox extension no longer runs. As we will see later in the paper this is related to Firefox' conservative approach to loading external resources by extensions. After an update, the Chrome version of the extension has been adapted to the change that allows it to run. It is possible because the new CSP for Gmail includes neither a `connect-src` nor a `default-src` directive, which allows scripts injected by browser extensions to open connections to remote servers. Rapportive uses this permissiveness in Gmail's CSP to load the LinkedIn profile information into the web page rendered in Chrome. This different behaviors of Rapportive with Firefox and Chrome exemplify the differences in the browser extension frameworks, motivating the need to investigate these differences deeper.

The above highlights the tension between the power of extensions and CSP intended by web pages. Website administrators are in a difficult position to foresee what browser extensions the users have installed. This indeed hampers the adoption of CSP.

Research questions The paper focuses on two main research questions: Q1: What is the state of the art in resolving the tension between the power of browser extensions and restrictions of CSP? and Q2: How to improve the state of the art as to leverage the security benefits of CSP without hampering the functionality of browser extensions?

The state of the art To answer Q1, we perform an in-depth study of practices used by extensions and browsers to deal with CSP. Within the browser, browser extensions have the capabilities to modify HTTP headers and content of the page. Injecting content to the page is common among extensions to add features and functionality. CSP applies also to this injected content, which may break or limit the functionality of the extension. To maintain the functionality and added user experience, extensions require the needs for relaxing the policy of the page. Because extensions have the capability to modify page headers, and to execute before a page is loaded, extensions have the window of opportunity to relax or even disable the CSP policy before it is applied. Changing the CSP header undermines high security requirements of a web service, e.g. for online banking, or simply bypass the benign intentions of a web application provider. Relaxing or removing the CSP of a web page disables this last line of defense.

Empirical study We address Q1 by a large-scale empirical study of all free 25835 extensions from Google's Chrome web store [10]. We have analyzed how browser extensions make use of their capability to modify the CSP header. We are also interested in how the presence of a CSP header affects content injection through browser extensions, i.e. the practical effects of CSP on extensions.

To understand the prevalence of invasive modifications, we have developed tools to identify such extensions and analyze their behavior in the presence of a CSP policy. The results are two-fold, they show that invasive modifications are very common in extensions, but at the same time manipulation of the CSP headers are still rare.

Vulnerability classes uncovered With the insights from the empirical study at hand, we categorize the findings to identify three classes of vulnerabilities that arise from invasive modifications that relax or disable the CSP policy. First, the extension injects *third party content* that increases the attack surface and introduces attack vectors that can be used for further attacks previously not present in the page. Second, it opens up for *XSS attacks* that would have been mitigated in otherwise hardened web pages. Third, the extension injects code that allows its developer to perform *user tracking* during the browser session. The invasive modifications described in these scenarios constitute a risk to both the user and the web service. Because extensions are applied either to a specific set of pages or all browsed pages, the impact varies. Naturally, an extension that is applied to every page puts the user at greater risk.

There exist, however, cases of content injections with which a web service would comply. For example, a web service provider that trusts the provided content of another service agrees to allow the modified CSP. By default the service does not include the third party content and therefore does not include the origin of the content in its CSP to be as restrictive as possible and thus to obtain the best protection for the web page. In this case, a relaxation of the CSP made by an extension would be acceptable. This brings out to the second research question and motivates a mechanism for detecting and endorsing CSP modifications to detect and agree on a policy acceptable by the web service.

CSP endorsement To address Q2, we propose a mechanism that enables extension-aware CSP endorsement by the server of the client's request to modify CSP. We expand the event processing of extensions in web browsers to detect CSP modifications made by an extension. On detection, the browser collects the necessary information and sends a request to the server which decides over accepting or rejecting the CSP modification. The browser eventually enforces the CSP policy respecting the server's decision. Additionally to the basic mechanism, we also propose an optimization with which the browser itself is able to make decisions based on origins labeled as acceptable by the web server, in order to obviate the need of sending CSP endorsement messages.

Note that the mechanism provides higher granularity than simply including a whitelist in the original CSP that attempts to foresee what extensions might be installed and relax CSP accordingly. Such an over-approximating whitelist would not be desirable to transport for performance reasons. Instead, our CSP endorsement allows making on-the-fly decisions by the server depending on the context and grants the flexibility of not having to send a complete whitelist in the first phase. We have implemented the CSP endorsement mechanism for Firefox and Chrome, and an endorsement server using *Node.js* [17].

Rapportive case study We have conducted a case study to analyze the usefulness of the CSP endorsement mechanism. For this, we have implemented a Firefox and a Chrome extension that models the behavior of Rapportive and used it to report the performance of the prototype implementation.

Contributions In summary, the paper's contributions are as follows:

– Large-scale empirical study to analyze the behavior of Chrome browser extensions in the context of CSP (Section 2).
– Identification of vulnerability classes stemming from modifications of CSP policies by extensions (Section 2).
– Analysis of browser extension framework behavior and the implications for resource loading in the presence of a CSP (Section 3).
– Development and prototype implementation of an extended CSP mechanism which allows the endorsement of CSP modifications (Section 4).
– Case study with the Rapportive extension to demonstrate the practicality of the approach (Section 5).

The program code for our prototype implementation is available online[4].

2 Empirical study

Browser extensions are pieces of software that can be plugged into web browsers and extend its basic functionality. Depending on the development interfaces and used technologies, the capabilities of extensions vary depending on the respective browser but powerful in general. For example, all major browsers enable extensions to intercept HTTP messages and to modify their headers, or to tweak the actual page content. Though this allows of course augmenting a user's browsing experience, this can willingly or unwillingly affect a web page's security.

In the following section, we analyze all 25835 free extensions from Google Chrome store in order to learn how browser extensions modify web pages in practice and how these modifications affect a page's security, in particular with respect to CSP policies. We classify our findings and identify popular real world examples while following the principle of responsible disclosure.

[4] `http://www.cse.chalmers.se/~danhau/csp_endorsement/`

2.1 Extension analysis

Many extensions modify the browsed page when loaded, however some do it more invasively than others. In order to understand how web pages are affected by extensions and their relation to CSP we perform an empirical study. The aim of the study is to see how many extensions are doing invasive modification to the web page source. An invasive modification is here defined as injecting content that poses a threat to the confidentiality or integrity of the page in the relation to user assets. Examples of such invasive modification are inclusion of scripts referring to potentially malicious external code, inclusion of scripts designed to track the user's browsing habits, and modifications that disable the browser's built in protection, e.g., against cross-site scripting attacks.

Large-scale study This study was performed by downloading (on December 20, 2014), extracting, and analyzing the source of each of the complete set of free extensions available in the Google Chrome store at the time of the analysis.

To perform the study we developed simple tools to automate the process of downloading extensions from the store and extracting their sources.

Only the scripts of the extension itself (called *content scripts*) can do invasive modifications to page content. Therefore, the analysis was limited to the subset of extensions that had content scripts defined. For each extension in this set each individual content script was analyzed to find invasive modifications that manifest themselves by injecting scripts or references to external resources. At last, we split the set into those that are applied to specific pages, versus every page. Due to the large number of extensions that modify the page, the analysis was to a large part automated.

To find extensions that modifies the CSP the set of extensions that in any form mentions *content security policy* or *CSP*. Each of these were manually inspected to see exactly how CSP is used. Because these extensions were manually inspected the numbers are exact.

A total of 25853 extensions were downloaded and analyzed. Out of these, about 1400 (5 %) of the existing extensions do invasive modifications to browsed pages, less than 0.2 % of all downloaded extensions take CSP into consideration. This suggests that the currently low adoption of CSP among major web sites makes invasive page modifications relatively rare and modification of the CSP header largely superfluous. If the technology reaches more wide-spread use, this will pose an issue for the large part of these extensions, who would in turn have to adapt. Table 1 summarizes these results.

Extension categories The results have been categorized in two main categories: extensions that invasively modify pages, and extensions that modify the CSP-header. The first category includes extensions that modify pages in a way that is restricted by the CSP policy if one is in place. In the later category we distinguish between different ways in which the CSP policy is modified, restricting, relaxing, replacing, or removing the policy, as can be seen in Table 1.

	Specific pages	All pages	Total
Modify page	651	781	1432
Modify CSP	20	25	45
- *Restrict*	1	4	5
- *Relax*	11	18	29
- *Replace*	2	2	4
- *Remove*	6	1	7

Fig. 1: Extension behavior with respect to page content and CSP modification

The main set of extensions that modify the CSP relaxes the policy to include a few additional domains. This is typically required for the extension to load external resources from these sources. A small number of extensions were found to make the policy stricter. These restrictions are generally made by extensions that allow the user to control what resources should be loaded and allowed to execute. Some extensions replace the policy. Here the new policy were either an "allow all" or "allow none" policy. Lastly, some extensions removed any CSP policy entirely. This allows the extension to add any content without restrictions.

Taking into account that extensions can be applied to different pages differently, the categories are further divided into the set of extensions that perform modifications on a single or small number of pages, and those that apply themselves to every page. The latter set of extensions are more of a concern as they potentially introduce vulnerabilities on every page visited by the user.

Vulnerability classes Many extensions rely on being able to inject content in the pages they are applied to. For these extensions a restrictive CSP policy prevents them from functioning as intended. Some of them bypass this restriction by modifying the policy to suit their needs. Extensions that modify the CSP header can inadvertently, or intentionally, weaken the security of a web page. Attacks that would have been prevented by the policy again pose a threat as a result of disabling or relaxing the policy.

We identify three classes of vulnerabilities to potentially expose web pages, and in that also the user, as a direct result of installing an extension that modifies the CSP. All three cases can be mitigated by a solid CSP policy.

Third party code inclusion: As documented by Nikiforakis et al. [24], including third party code weakens the security of the web page in which it is included. A real-life example is the defacement of the Reuters site in June 2014 [31], attributed to "Syrian Electronic Army", which compromised a third-party widget (Taboola [34]). This shows that even established content delivery networks risk being compromised, and these risks immediately extend to all web sites that include scripts from such networks.

Upon loading a page, certain extensions inject and load third party code in the page itself, out of the control of the page. The included third party code can be benign in itself, but broadens the attack surface in an unintended way.

Or, it contains seemingly benign code that may be malicious in the context of a specific page. For external resources the included code may change over time, making it next to impossible to tell for certain what code will actually execute.

A prominent example of such an extension has at present around 1.9 million users. The extension allows the user to select a section of the page to be saved for viewing offline at a later point. In order to do so, it injects a script tag in every browsed page from a domain under the control of the developers. Should the developer have malicious intentions, or the domain be hacked, it would take no effort to exploit every web page visited by their 1.9 million users. By inspecting its code and stated purpose, it is most certainly believed to be benign, yet it displays the vulnerable behavior described above.

Enabling XSS: Recall that CSP is intended to be a "last line of defense" against XSS attacks by optionally disabling inline code and restricting resource origins to a small set of trusted domains. Extensions that modify the CSP policy open up for otherwise prevented attacks, in particular extensions that add the `unsafe-inline` or `unsafe-eval` to the `script-src`, `style-src` or even `default-src` directives. This allows to directly inject arbitrary code into the web page and to execute it.

One high-profile extension, at the time of writing having more than 2.8 million users, allows the user to define scripts that will execute when certain pages are browsed, e.g., to load aggregated information or dynamically set the background color of pages during night time. To allow the user defined scripts full freedom to execute, the extension removes the CSP-header on every browsed page. While perhaps written with good intentions, the extension subjects the user to additional exposure on pages that are normally protected by a CSP policy.

User profiling: Profiling a user's habits can be a lucrative source of information. A large number of extensions add content that allows its developer to track the user's movements across the Internet, page by page. In an extreme form of tracking, some extensions add Google Analytics to *every* browsed web page with their own Google Analytics ID, enabling comprehensive user profiling.

One extension, with possibly unsuspecting 1.2 million users, stands out in this respect. The official description states that the extension offers an extended set of smileys on a small and specific hardcoded set of web pages. Aside from this, the extension injects itself in every page and adds a Google Analytics script with own ID.

3 Extension framework analysis

An important goal for our work is understanding the behavior of extensions in different browsers. Our attention is focused on the current stable release versions of Firefox (v35.0.1), Chrome (v40.0.2214.111), Opera (v27.0.1689.66) and Safari (v8.0.3) whose extensions are especially popular. In particular, we want to know how browsers restrict loading of resources initiated by an extension.

In this respect, we first describe in this section a simple scenario for loading a sequence of different resources which we use to examine the behaviors of the afore mentioned browsers. We then demonstrate the real world implications of the different browser behaviors with a case study on LinkedIn's browser extension Rapportive for Firefox and Chrome.

3.1 Resource loading through content scripts

There are two ways of loading resources: first, the content is loaded by the extension itself directly into the extension. Second, a target web page is modified to load the content as part of the page itself. In Chrome, e.g., loading a script is done by injecting an HTML element either in the extension's internal background page or the web page, respectively. Loading within an extension is in Chrome restricted through a CSP defined in its manifest file [12]. This CSP is defined by the developer herself and applies only to the extension, not to browsed web pages. Since extension security is already extensively discussed in literature [18,1,32,4,5,7], we focus on loading of resources in the context of web pages deploying CSP policies.

We have set up a simple scenario in various browsers to test possible content script behavior. We illustrate the setup in Figure 2. In the first step, an extension injects an HTML script element with a script `script1.js` from a server `server1` into the visited web page's DOM. This causes the browser in a second step to load the script from the given URI. Third, the code of `script1.js` injects yet another HTML script element with a script `script2.js` from a server `server2` into the same web page (step 3). The browser again loads the script from the server (step 4) but this time generates a dialog message indicating that the loading of `script2.js` was successful (step 5).

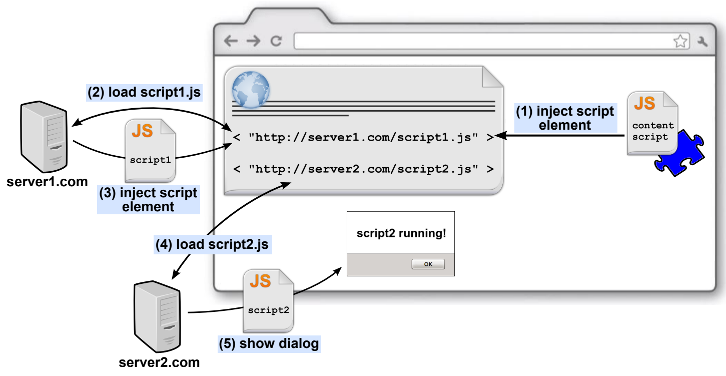

Fig. 2: Experiment set-up for evaluating the behavior of browsers for resource loading initiated by extensions

We have implemented the same scenario for various browsers and have tested the content script behavior of the respective extensions in the presence of CSP. We have chosen `example.com` as our target page. Since it does not define a CSP, we have added the most restrictive policy `default-src 'none'` to the HTTP headers. The content scripts of Firefox extensions become part of the web page they are injected in. Consequently, the page's CSP directly applies to the content scripts and the loading of `script1.js` (step 2) is already blocked. We have observed a different behavior for Chrome, Opera and Safari. In these browsers, the content script successfully injects and loads the first script (steps 1-3). Step 4 and 5, however, are not executed since `script1.js` is full part of the web page and thus requesting `script2.js` is blocked by its CSP. Surprisingly, this behavior has been observed regardless of the web page's or the extension's internal CSP. This implies that even with a most restrictive CSP policy for the web page and browser extensions, an extension is always able to send requests to arbitrary servers, potentially leaking sensitive user information. Thus, extensions for these browsers can actively circumvent the CSP security mechanism. We show our results in Figure 3.

Browser	script1.js form server1.com	script2.js from server2.com
Firefox	blocked	blocked
Chrome/Opera	loaded	blocked
Safari	loaded	blocked

Fig. 3: Different browser behavior for content loading in the scenario from Fig. 2

3.2 Case study: Rapportive

Rapportive [20] is LinkedIn's browser extension which augments Google's email service Gmail [13] with LinkedIn profile information. This extension modifies the Gmail web page such that when a user composes an email, it automatically tries to look up the recipient's LinkedIn account information and presents it as part of the Gmail web page to the user. More technically, Rapportive dynamically loads scripts from LinkedIn within the extension and injects them through a content script into the Gmail web page. The injected scripts are then responsible for fetching the addressee's LinkedIn profile information.

Rapportive as an extension contains only scripts to dynamically download other scripts from `rapportive.com`, `linkedin.com` and LinkedIn subdomains, which provide the actual functionality of the extension, the fetching and displaying of profile data. In Rapportive for Firefox and for Chrome, user scripts are responsible for injecting HTML elements which load the respective online code into the web page. In case of Firefox, this is done by injecting an HTML `script` element from `rapportive.com`. However since content scripts are treated as part of the web page, Firefox immediately blocks its loading and consequently breaks the functionality of Rapportive. Ironically, users blamed LinkedIn, not Gmail, for breaking the extension [30]. Rapportive for Chrome, on the other hand, has been updated in reaction to Gmail's deployment of a CSP. The extension makes active use of Chromes behavior to load resources directly injected by user scripts and injects an iframe with a resource from `api.linkedin.com`. In accordance with the standard, the CSP of a host page does not apply to the content of an iframe. Therefore, every subsequent loading is done within the iframe.

4 CSP endorsement

The implementations of Rapportive relies on the fact that the CSP policy of Gmail only restricts script, frame and plugin sources but is otherwise fully permissive, e.g. it does not hinder loaded script code to load resources from servers through, e.g., `XMLHttpRequest`. A web page's CSP policy is, however, most effective only if it is most restrictive. This can conflict with the injection behavior of extensions and eventually break their functionality. In this section we develop a mechanism that allows web applications to deploy a most restrictive CSP policy while guaranteeing the full functionality of browser extensions which inject resources from trusted domains into a web page. We first introduce a general mechanism to allow requesting endorsement of a new CSP by a web server if it is required for the seamless functioning of installed extensions. After that, we present our prototype implementation for Firefox and the Chrome browser.

4.1 Endorsement workflow

A web browser's main purpose is to request web pages usually via HTTP or HTTPS from a web server. In most major web browsers, these requests as well as their respective responses can be intercepted by extensions. In order to do so, an extension registers a callback function for the respective events, e.g. the event that occurs on receiving a response message. An extension can then modify the HTTP CSP header in any possible way or even remove it from the HTTP response. But since browsers are responsible for calling an extension's registered event handler function, the browser can also analyze their returned data. In particular, a browser can detect when the CSP header in an HTTP response was modified by an extension. On detection, we want to send a message to a specified server to inform about the CSP modification and request a server-side decision if the change is acceptable. This of course requires a server mechanism that allows the processing of CSP endorsement requests. In the following, we describe the workflows and interplay of both, the web browser and the web server, for CSP endorsements.

Browser improvement On detection of a CSP header modification, we need to notify the server which accepts and processes CSP endorsement requests. To make use of existing features and to ensure backwards compatibility with the current standard, we use the URI provided in the existing CSP directive `report-uri` for determining the CSP endorsement server. Additionally, we introduce a new keyword `'allow-check'` that can be added to the `report-uri` directive. If this keyword is set, the HTTP server notifies the browser that the report server is capable of handling endorsement request messages. Otherwise, the browser behaves as without the endorsement mechanism in place.

The overall extended browser behavior is as follows: if the CSP is set to be enforced in the browser and the `'allow-check'` flag is set in the `report-uri` directive, the browser sends a CSP endorsement request message to the report server whenever a CSP modification has been detected. In case the flag is missing, it is assumed that the server does not accept endorsement requests and the browser falls back to the current standard behavior. The same fall-back behavior is applied in report-only mode, even when the flag is set. The reason is, because the CSP is not enforced, the extension functionality is not affected by the CSP and endorsement requests are thus redundant. Note that in any case the standard behavior for CSP is not affected at all. For example even when `'allow-check'` is defined, reports are sent out for every CSP violation as defined by the standard.

We show the basic workflow for the browser and the endorsement server in Figure 4. For our protocol to work, we assume a browser that implements our endorsement mechanism and additionally has at least one extension installed that modifies the CSP in the HTTP headers of received messages. The initial situation is that the browser sends a page request to an HTTP server and receives a response including a CSP header. The browser checks the CSP in the received header if the policy is enforced, if it includes the `report-uri` directive with a report URI and the `'allow-check'` directive. In case of a positive check, the browser stores a copy of the received CSP. Subsequently, the browser triggers the *on-headers-received*[5] event which allows the installed extensions to access and modify the header fields. If any of the checks so far has been negative, the browser continues just as without the endorsement mechanism. If however all checks before the event call have been positive, the modified CSP headers are compared with the original ones. When a CSP modification is detected, the browser sends the updated CSP policy to the endorsement server. The server's URI is retrieved from the `report-uri` of the original CSP to prevent malicious extensions from redirecting endorsement requests to the attacker's server. The endorsement server decides whether to accept or reject the CSP modification and transfers the result back to the browser. In case the modified CSP is accepted, the browser continues with the new policy. In case of rejection, the browser falls back to the initially received CSP and discards the modifications. Any subsequent page handling, e.g. the page rendering, is kept unchanged regardless the server's decision. This means in particular that the CSP is enforced normally and violations are reported as defined by the current standard.

Endorsement server The endorsement server is the entity which accepts messages reporting CSP policy modifications and makes the decision if the modified

[5] The actual name of the event depends on the browser implementation.

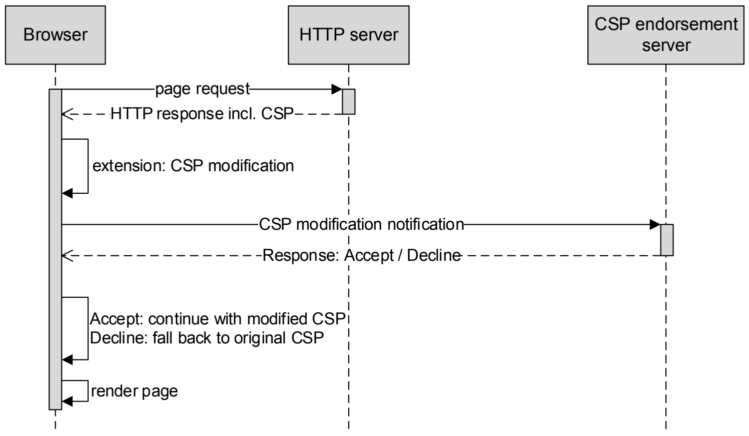
Browser
HTTP server
CSP endorsement server
page request
HTTP response incl. CSP
extension: CSP modification
CSP modification notification
Response: Accept / Decline
Accept: continue with modified CSP
Decline: fall back to original CSP
render page

Fig. 4: CSP endorsement workflow

policy should be applied or discarded by the browser. On receiving an endorsement request message, the server must return a message containing either `Accept` or `Reject`. Otherwise, there are no restrictions on how to implement the server's decision making process. However, we suggest a server-side whitelisting as an intuitive and relatively easy to implement concept as the basis for the decision making process. For the remainder of this paper, we assume a server implementation using the whitelisting strategy.

One possible way to obtain a proper whitelist is to evaluate received CSP violation reports and endorsement request messages. A server administrator can, for example, extract the most frequent domains and analyze them in terms of trustworthiness. Depending on the administrator's decision, she can then update the whitelist with the validated domains. This method allows a half-automated and incremental maintenance of the CSP endorsement whitelist.

Optimization The CSP policy is read out and fixed before page rendering. This implies that the modification and endorsement of CSPs must be finished before page rendering, i.e. the browser must interrupt the page loading process until a decision is made. This blocking behavior comes with an obvious performance and user experience problem. Intuitively, the loading is delayed for a full round-trip time (endorsement request message plus server response) and the computation time for making the decision itself.

To address this issue, we optimize the endorsement approach by introducing a separate whitelist sent in addition to the CSP policy. This allows for decision making in the browser on behalf of the endorsement server. The whitelist reflects in essence the list used for server-side decision making. But since the server-side whitelist might be relatively large, it is sufficient to send only a sublist with the most frequently endorsed trusted domains. Before sending an endorsement request, the browser is now enabled to accept or reject a CSP modification itself based on this list. The main motivation for a separate list and for not including trusted domains directly in the CSP, is to keep the actual CSP policy as restrictive as possible while still informing the browser which URIs are acceptable to be added to the CSP.

We do not require the whitelist to be complete. Therefore, if a modification is rejected by the browser, it must still send the endorsement request to the server because the rejected domains might be included in the complete server-side list. In case the modification is accepted, the whitelist is sufficient and the browser can immediately proceed with processing the page. Thus, for a positive client-side evaluation of CSP modifications the endorsement request must not be sent resulting in an effective performance gain. We evaluate the performance of our prototype implementation and the improvement through the optimization in Section 5.

4.2 Prototype implementation

Browser modification We have implemented our approach for Firefox Nightly version 38 [23] and Chrome version 41 [11]. For Firefox, we have adjusted the browser's observer mechanism to detect CSP header modifications, to store the original header for potential later client-side decision making and to subsequently trigger the CSP endorsement mechanism. For Chrome, we have modified the browsers event handler in a way that it triggers our mechanism on returns of the `onHeadersReceived` event. The return value of this API function contains the modified CSP header value. The storage of the initially received header for later comparison is done automatically by Chrome. Both browser implementations have in common that whenever a CSP header has been modified, they check for the CSP enforcement mode and the presence of the `'allow-check'` flag in the `report-uri` directive of the original CSP. If both checks succeed, the browsers try too extract the whitelist from the `csp-whitelist` HTTP header. If one is provided, the browsers try to make a decision without any further server requests. However if the check is negative, a CSP endorsement request is sent to the first server given in the `report-uri` directive. On receiving a reply from the server with `Accept`, the browsers proceeds as without our code extension and eventually replace the initially received CSP with the modified version. Otherwise, the CSP header is modified in the usual way.

The implementation of the browser internal decision making expects a JSON formatted whitelist in which the attributes match the directive names and their values define a list of URIs which are accepted to be added to the respective directive in the CSP. Additionally, there can be the attribute `general` which value denotes a list of URIs accepted to be added to any directive in the CSP. If any of the attributes is missing it is treated as it would contain an empty list, i.e. no modification is permitted for the respective directive. We show whitelist examples in Listing 1.1 and 1.2. The first example allows adding `https://platform.linkedin.com/` to every directive and defines specific URIs allowed to be added to the `script-src` and the `frame-src` directive, respectively. The second example does not allow any URI to be added to any directive which effectively rejects all CSP modifications. Note that removing URIs from the policy is not forbidden since that would make the policy only more restrictive but does not introduce any potential security risks.

Listing 1.1: whitelist policy accepting CSP modifications for Rapportive

```
{ "general": ["https://platform.linkedin.com/"],

  "script-src": ["https://rapportive.com/",
                 "https://www.linkedin.com/"],

  "frame-src": ["https://api.linkedin.com/"] }
```

Listing 1.2: modification acceptance policy rejecting any modification

```
{ "general":  [] }
```

Endorsement server implementation We have implemented a CSP modification endorsement server using the Node.js [17] runtime environment. We have implemented the same whitelist behavior as for the client-side decision making in the browser. This means that the server implementation accepts the same JSON formatted whitelist as the server configuration and uses the same algorithm to decide whether to accept or reject a policy modification.

5 Evaluation

We have used Rapportive in a case study to empirically gain experience regarding the applicability and effectiveness of our approach. In the following, we introduce the general setup and report on the results collected from our experiments.

5.1 Experiment set-up

For all our experiments we have used a Dell Latitude with an Intel i7 CPU and Ubuntu 14.10 operating system. Since we have implemented our approach for Firefox and Chrome, we have been able to analyze the implementations of Rapportive for both browsers.

In reaction to Gmail's CSP change from report-only to enforcement mode [14], LinkedIn has adjusted Rapportive for Chrome to not conflict with the policy. However, at the time of writing the paper, the Firefox counterpart has no longer been functioning since the dynamic loading of the necessary scripts is blocked by the CSP policy. We have implemented extensions for both browsers with the exact same functionality as Rapportive, except that our extension also modifies the CSP header and adds the necessary URIs to the policy. For convenience, we refer to our extension implementations as "Rapportive" in the remainder of this paper since they behave otherwise exactly the same.

Gmail deploys a CSP policy whitelisting resources for the `script-src`, `frame-src` and the `object-src` directive, respectively. The policy does not include the `default-src` directive which implies that there are no restrictions on other ways of loading content than the just mentioned ones, e.g. loading of content with `XMLHttpRequest`. Violation reports are sent to the URI defined in the CSP's `report-uri` directive. The complete CSP policy which had been in place during our experiments is provided in Appendix A.

The implementation of our approach uses the first report URI as the URI of the CSP endorsement server. In order to conduct experiments with Gmail, we have therefore installed a local proxy server, using mitmproxy [22], which replaces Gmail's report URI with the one of our CSP endorsement server and appends the `'allow-check'` keyword. Depending on the experiment, we have also added the `csp-whitelist` header with the respective whitelist. Any other header, including the rest of the CSP header, has been left unchanged and forwarded to the browsers.

As the endorsement server, we have installed our Node.js based server implementation on the same machine as we have run our browser experiments. This allows easier repetition of the experiments and avoids misleading network latencies. At the same time we believe this set-up to be sufficiently expressive for analyzing the general performance of our implementation.

5.2 Results

We have conducted experiments with the three possible execution modes of our approach: sending of endorsement requests with full server-side decision making, receiving the modification acceptance whitelist with full client-side decision making, and mixed decision making, i.e. the additional whitelist sent with the HTTP response is not sufficient for making a client-side decision and an endorsement request is sent subsequently.

In all experiments, Rapportive relaxes Gmail's CSP by adding the three URIs `https://rapportive.com/`, `https://platform.linkedin.com/` and `https://www.linkedin.com/` to the `script-src` directive, and to the `frame-src` directive the URI `https://api.linkedin.com/`.

For each scenario we have measured the time overhead of the overall endorsement process, the browser internal decision making process and the round-trip time needed to request a decision from the CSP endorsement server. The results for both browsers are summarized in Figure 5 and depict the average times of 200 samples.

Server-side decision making In our first experiment the endorsement server accepts all CSP modifications using the policy shown in Listing 1.1. The main observation is that the most time is consumed by the server-side processing which itself is almost completely the time for sending and receiving the endorsement messages. For Firefox, the browser internal processing is even so small that it is hardly noticeable. Note that the transmission times are relatively short because all components are located on the same machine. For an external endorsement

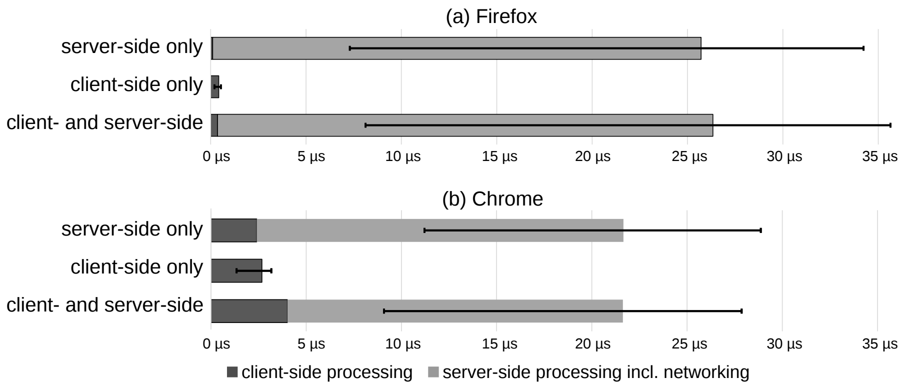

Fig. 5: Time overhead of client-side only, server-side only, and server-and client-side decision making with the respective standard deviations

server the message round-trip times increase accordingly. The results are shown in the first bars of each diagram for the respective browsers in Figure 5.

Client-side decision making In the second experiment, the proxy injects the whitelist from Listing 1.1, i.e. it matches exactly the URIs added by Rapportive. The resulting overhead is exactly the time required to come to a client-side decision. For a human user, this delay is not noticeable and the browsing experience is not affected at all. The results are shown in the second bar of each diagram for the respective browsers in Figure 5.

Mixed decision making The last experiment in essence combines both previous ones. However, the whitelist added by the proxy is not sufficient to come to a positive decision on the client side. As a result, an endorsement request is sent to the server subsequently. The time overhead is, similar to the first experiment, dominated by the communication with the endorsement server. Though in this last scenario the browsers also try to make a decision based on the received `csp-whitelist` header, the measured times are similar to the ones in the second experiment and the delays not noticeable for a human user. The results are shown in the third bars of each diagram for the respective browsers in Figure 5.

Though the third scenario represents the "worst case", adding the time for the server communication to the time needed for browser internal decision making, the overhead for the client-side decision making is small enough and thus negligible compared to the networking overhead. Therefore, the mixed decision making scenario performs roughly the same as with server-side decision making only and comparably, an insufficient whitelist does not introduce an affecting disadvantage. However, the second experiment shows that the optimization through possible client-side decision making introduces a significant improvement and makes our approach practicable.

6 Related work

Compared to the CSP standard 1.0 [36], the successor standard 2.0 [37] includes several new features and eliminates certain shortcomings. For example, the new `plugin-types` directive restricts the possible MIME-types of embedded media, the `child-src` replaces the `frame-src` directive to cover both, iframes and workers, or the `frame-ancestor` which attempts to supersede the X-Frame-Options HTTP request header. However, both standards note that they do not intent to influence the workflow of extensions. Our approach only detects policy modifications and is widely independent from the CSP specification. This makes the endorsement mechanism compatible with both CSP 1.0 and CSP 2.0.

Weissbacher et al. [39] measure a low deployment rate of CSP and conduct studies to analyze the practical challenges for deploying CSP policies. They point out that it is difficult to define a policy for a web page that utilizes the full potential of CSP. One of their studies is on inferring policies by running the server in the report-only mode for CSP, collecting the reports and helping developers to define and revise policies. Weissbacher et al. note the conflict of browser extension and web page functionality and suggest exempting extensions from the CSP policies altogether. Our mechanism offers flexibility on the server side, where exempting, denying or selectively granting are all possible alternatives.

Fazzini et al. propose AutoCSP [9], a PHP extension that automatically infers a CSP policy from web pages on HTML code generation. In our approach web pages are queried normally and a server is initially unaware of any installed extensions and possible CSP modifications. In fact, even after a modification, the server does not learn anything about installed extensions but only receives the modified CSP policy for endorsement. In this way, AutoCSP and our approach complement each other.

UserCSP [29] is a Firefox extension that allows a user to manually specify a CSP policy for web pages herself. Besides that this approach requires a certain level of expertise and a certain degree of insight into the web pages functionality, it cannot protect from non-compliant CSP policy modifications by other extensions. Other implementations infer a CSP policy based on the in the browser rendered page [16,33]. These approaches assume an untampered version of the web page, i.e. unmodified by browser extensions or untouched by web attackers. Therefore, they are helpful for finding a suitable CSP policy but the results give no guarantees and should be manually inspected by a web administrator.

The analysis of browser extensions has recently received more attention. The focus lies either on the detection of maliciously behaving browser extensions [18,1,32], infection and protection of extensions from dynamically loaded third party code [4] or the protection of web pages from malicious browser extensions [5,7]. Orthogonal to this, we do not analyze the extension behavior itself but rather observe how extensions affect a web page's security for the particular case of CSP policies.

In a line of work to secure JavaScript in browser extensions, Dhawan and Ganapathy [8] develop Sabre, a system for tracking the flow of JavaScript objects as they are passed through the browser subsystems. Bandhakavi, et al. [2]

propose a static analysis tool, VEX, for analyzing Firefox extensions for security vulnerabilities. Heule et al. [15] discuss the risks associated with the trust that browsers provide to extensions and look beyond CSP for preventing privacy leaks by a confinement system.

Other works study the different development kits for extensions of common web browsers [19,3,7]. Though we have observed the effective behavior of content scripts in browsers, our interest has been only common practices of browser extensions on the market and the enforcement of CSP policies in case of content injections through content scripts into web pages.

7 Conclusion

We have investigated the empirical and conceptual aspects of the tension between the power of browser extensions and the CSP policy of web sites. We have shown that the state of the art in today's practice includes both invasive page modification and the modification of the CSP policy itself. This leads to three classes of vulnerabilities: third party code inclusion, enabling XSS, and user profiling. We have presented an empirical study with all free Chrome extension from Chrome web store identifying extensions with over a million of users in each category.

With the goal to facilitate a wider adoption of CSP, we have presented an endorsement mechanism that allows extensions and servers to amend the CSP policy on the fly. We have evaluated the mechanism on both the Firefox and Chrome versions of the popular Rapportive extension, indicating the practicality of the approach.

Following responsible disclosure, we have reported the results of our empirical study to Google. Since the time of the study, three extensions with invasive CSP modifications have been removed from the Chrome store, including the one with 1.2 million users that we discuss in the user profiling category.

Future work includes exploring the possibilities of user involvement in the CSP policy amendments. A GUI notification might be useful to allow ignoring the endorsement rejects from the server.

In this context, an empirical study along the lines of Maggi et al. [21] may reveal the real-world impact of restrictions imposed by CSP policies as described in this paper, together with their perception by human users.

Acknowledgments Thanks are due to Federico Maggi, Adrienne Porter Felt, and the anonymous reviewers for the helpful comments and feedback. This work was funded by the European Community under the ProSecuToR and WebSand projects and the Swedish research agencies SSF and VR.

References

1. S. Van Acker, N. Nikiforakis, L. Desmet, F. Piessens, and W. Joosen. Monkey-in-the-browser: malware and vulnerabilities in augmented browsing script markets. In *ASIA CCS '14*, 2014.

2. S. Bandhakavi, N. Tiku, W. Pittman, S. T. King, P. Madhusudan, and M. Winslett. Vetting browser extensions for security vulnerabilities with vex. *Commun. ACM*, 2011.
3. A. Barth, A. Porter Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *NDSS*, 2010.
4. A. Barua, M. Zulkernine, and K. Weldemariam. Protecting web browser extensions from javascript injection attacks. In *ICECCS*, 2013.
5. L. Bauer, S. Cai, L. Jia, T. Passaro, and Y. Tian. Analyzing the dangers posed by Chrome extensions. In *IEEE CNS*, 2014.
6. BuiltWith. Content security policy usage statistics. `http://trends.builtwith.com/docinfo/Content-Security-Policy`. accessed: Feb 2015.
7. W. Chang and S. Chen. Defeat information leakage from browser extensions via data obfuscation. In *Information and Communications Security*, 2013.
8. M. Dhawan and V. Ganapathy. Analyzing information flow in javascript-based browser extensions. In *ACSAC*, 2009.
9. M. Fazzini, P. Saxena, and A. Orso. AutoCSP: Automatically Retrofitting CSP to Web Applications, 2015.
10. Google. Chrome web store. `https://chrome.google.com/webstore/category/extensions`. accessed: Feb 2015.
11. Google. Chromium. `http://dev.chromium.org/Home`. accessed: Feb 2015.
12. Google. Content security policy (csp) - google chrome. `https://developer.chrome.com/extensions/contentSecurityPolicy`. accessed: Feb 2015.
13. Google. Gmail. `https://www.gmail.com/`. accessed: Feb 2015.
14. Google. Reject the unexpected - content security policy in gmail. `http://gmailblog.blogspot.se/2014/12/reject-unexpected-content-security.html`. accessed: Feb 2015.
15. S. Heule, D. Rifkin, D. Stefan, and A. Russo. The most dangerous code in the browser. In *HotOS*, 2015.
16. A. Javed. CSP AiDer: An automated recommendation of content security policy for web applications. Poster at IEEE Symposium on Security & Privacy 2011.
17. Joyent. Node.js. `http://www.nodejs.org/`. accessed: Feb 2015.
18. A. Kapravelos, Ch. Grier, N. Chachra, Ch. Kruegel, G. Vigna, and V. Paxson. Hulk: Eliciting Malicious Behavior in Browser Extensions. In *USENIX Sec.*, 2014.
19. R. Karim, M. Dhawan, V. Ganapathy, and Ch. Shan. An analysis of the mozilla jetpack extension framework. In *ECOOP*, 2012.
20. LinkedIn. Rapportive. `https://rapportive.com`. accessed: Feb 2015.
21. F. Maggi, A. Frossi, S. Zanero, G. Stringhini, B. Stone-Gross, Ch. Kruegel, and G. Vigna. Two years of short urls internet measurement: security threats and countermeasures. In *WWW*, 2013.
22. mitmproxy. `https://mitmproxy.org/`. accessed: Feb 2015.
23. Mozilla. Firefox nightly. `https://nightly.mozilla.org/`. accessed: Feb 2015.
24. N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, Ch. Kruegel, F. Piessens G., and Vigna. You are what you include: Large-scale evaluation of remote javascript inclusions. In *CCS*, 2012.
25. OWASP. Clickjacking. `https://www.owasp.org/index.php/Clickjacking`. accessed: Feb 2015.
26. OWASP. Content security policy. `https://www.owasp.org/index.php/Content_Security_Policy`. accessed: Feb 2015.
27. OWASP. Cross-site scripting. `https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29`. accessed: Feb 2015.

28. OWASP. Top 10 2013. https://www.owasp.org/index.php/Top_10_2013. accessed: Feb 2015.
29. K. Patil, T. Vyas, F. Braun, M. Goodwin, and Z. Liang. Poster: UserCSP - User Specified Content Security Policies. In *SOUPS*, 2013.
30. Rapportive :: Reviews :: Add-ons for firefox. https://addons.mozilla.org/en-US/firefox/addon/rapportive/reviews/. accessed: Feb 2015.
31. Syrian Electronic Army uses Taboola ad to hack Reuters (again). https://nakedsecurity.sophos.com/2014/06/23/syrian-electronic-army-uses-taboola-ad-to-hack-reuters-again/.
32. H. Shahriar, K. Weldemariam, M. Zulkernine, and T. Lutellier. Effective detection of vulnerable and malicious browser extensions. *Computers & Security*, 2014.
33. B. Sterne. Content security policy recommendation bookmarklet. http://brandon.sternefamily.net/2010/10/content-security-policy-recommendation-bookmarklet/. accessed: Feb 2015.
34. Taboola. Taboola — drive traffic and monetize your site. http://www.taboola.com/. accessed: Feb 2015.
35. Can I Use. Content security policy 1.0. http://caniuse.com/#feat=contentsecuritypolicy. accessed: Feb 2015.
36. W3C. Csp 1.0. http://www.w3.org/TR/CSP/. accessed: Feb 2015.
37. W3C. Csp 2.0. http://www.w3.org/TR/CSP2/. accessed: Feb 2015.
38. W3C. World wide web consortium. http://www.w3.org/. accessed: Feb 2015.
39. M. Weissbacher, T. Lauinger, and W. Robertson. Why Is CSP Failing? Trends and Challenges in CSP Adoption. In *RAID*, 2014.
40. WhiteHat. Content security policy - whitehat security blog. https://blog.whitehatsec.com/content-security-policy/. accessed: Feb 2015.

A Gmail CSP policy (12. January 2015)

```
script-src https://*.talkgadget.google.com 'self' 'unsafe-
    inline' 'unsafe-eval' https://talkgadget.google.com https:
    //www.googleapis.com https://www-gm-opensocial.
    googleusercontent.com https://docs.google.com https://www.
    google.com https://s.ytimg.com https://www.youtube.com
    https://ssl.google-analytics.com https://apis.google.com
    https://clients1.google.com https://ssl.gstatic.com https:
    //www.gstatic.com blob:;
frame-src https://*.talkgadget.google.com https://www.gstatic.
    com 'self' https://accounts.google.com https://apis.google
    .com https://clients6.google.com https://content.
    googleapis.com https://mail-attachment.googleusercontent.
    com https://www.google.com https://docs.google.com https:
    //drive.google.com https://*.googleusercontent.com https:
    //feedback.googleusercontent.com https://talkgadget.google
    .com https://isolated.mail.google.com https://www-gm-
    opensocial.googleusercontent.com https://plus.google.com
    https://wallet.google.com https://www.youtube.com https://
    clients5.google.com https://ci3.googleusercontent.com;
object-src https://mail-attachment.googleusercontent.com;
report-uri /mail/cspreport
```