AutoNav: Evaluation and Automatization of Web Navigation Policies

AutoNav: Evaluation and Automatization of Web Navigation Policies

Benjamin Eriksson
Chalmers University of Technology

Andrei Sabelfeld
Chalmers University of Technology

Abstract

Undesired navigation in browsers powers a significant class of attacks on web applications. In a move to mitigate risks associated with undesired navigation, the security community has proposed a standard that gives control to web pages to restrict navigation. The standard draft introduces a new `navigate-to` directive of the Content Security Policy (CSP). The directive is currently being implemented by mainstream browsers. This paper is a first evaluation of `navigate-to`, focusing on security, performance, and automatization of navigation policies. We present new vulnerabilities introduced by the directive into the web ecosystem, opening up for attacks such as probing to detect if users are logged in to other websites or have active shopping carts, bypassing third-party cookie blocking, exfiltrating secrets, as well as leaking browsing history. Unfortunately, the directive triggers vulnerabilities even in websites that do not use the directive in their policies. We identify both specification- and implementation-level vulnerabilities and propose countermeasures to mitigate both. To aid developers in configuring navigation policies, we develop and implement AutoNav[1], an automated black-box mechanism to infer navigation policies. AutoNav leverages the benefits of origin-wide policies in order to improve security without degrading performance. We evaluate the viability of `navigate-to` and AutoNav by an empirical study on Alexa's top 10,000 websites.

CCS Concepts

• **Security and privacy → Web application security**;

Keywords

web application security, csp, web navigations

ACM Reference Format:
Benjamin Eriksson and Andrei Sabelfeld. 2020. AutoNav: Evaluation and Automatization of Web Navigation Policies. In *Proceedings of The Web Conference 2020 (WWW '20), April 20–24, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 12 pages. `https://doi.org/10.1145/3366423.3380207`

1 Introduction

As the power of the web platform grows, attackers increasingly target *client-side vulnerabilities* [3, 9, 12, 16, 18, 37, 39, 43, 50, 56, 57]. Exploiting these vulnerabilities is effective because clients manipulate highly sensitive information, like login credentials, banking, health, and location data, on behalf of the user.

1.1 Motivation

One of the bigger classes of client-side security vulnerabilities on today's web is *cross-site scripting (XSS)* [45] . An XSS vulnerability gives an attacker the power to execute JavaScript code on another website. This can be used to steal user credentials, change the behavior of the application or render the website unusable. A common approach to mitigate this problem is to let servers send extra security policies along with each HTTP response. The web browser will then enforce these policies, for example, by restricting which scripts to allow on the webpage. These security policies have been defined by the web security community as part of *Content Security Policy (CSP)* [63].

Navigation attacks The current CSP standard (level 2 [63]) does not address attacks via *navigation*. Attackers can thus freely redirect users to malicious or inappropriate websites. This type of attack can affect the confidentiality, integrity and availability of the attacked website. For confidentiality, an attacker with injection capabilities can inject the following script to leak the secret cookie.

```
1  <script>
2  window.location = "http://evil.com/?c="+document.cookie;
3  </script>
```

When the script is executed the user will be sent to `http://evil.com`, along with their cookies, potentially allowing the attacker to take over their account. In addition to only stealing the cookie, the attacker could launch a phishing attack by designing `http://evil.com` to look like the attacked website. Here the user could be asked to supply more confidential information or be forced to download malicious software. The availability of the website is also compromised as every user visiting the page containing the injected script will be sent away. Note that while CSP can block scripts, an attacker could also force the user to perform a navigation by using meta tags as shown below. While not valid HTML, modern browsers will follow meta redirects in the HTML body.

```
1  <meta http-equiv="refresh"
2        content="0;URL='http://evil.com/'" />
```

The `navigate-to` *directive* To mitigate these problems the World Wide Web Consortium (W3C) has drafted a standard for the new CSP directive `navigate-to` [61]. This directive has already been implemented in Chrome [36] and Firefox [28]. A common motivation for the directive is to increase the security on websites, as well as, give advertising platforms better control over navigations in ads [40]. We illustrate this in two example scenarios: HTML/JavaScript injection and malicious advertisement.

HTML/JavaScript injection Understanding the space of navigation links on a website can improve security thanks to `navigate-to`. By

[1]Our implementation is available online on `https://www.cse.chalmers.se/research/group/security/autonav/`

WWW '20, April 20–24, 2020, Taipei, Taiwan

ACM ISBN 978-1-4503-7023-3/20/04.
`https://doi.org/10.1145/3366423.3380207`

limiting the possible navigations, attackers will not be able to redirect users. A real-world example of where this policy would have helped is a vulnerability on `blockchain.info` [27]. Attackers were able to inject HTML and JavaScript into the search function on the page. This meant that a URL similar to `blockchain.info/?search=<code>`, which appears to point to `blockchain.info`, could redirect the user to another website. This is known as a reflective XSS vulnerability [48] , as the code in the URL is reflected onto the page. Although `blockchain.info` used CSP to mitigate XSS, it was still possible to inject HTML code that forces a redirect. With the new directive, the following CSP policy can mitigate this type of attack. This policy blocks any navigation attempt to anything but `self`, i.e. `blockchain.info`.

```
navigate-to 'self'
```

Malicious advertisement Advertisement platform providers benefit from ensuring that users who click on their ads end up on the correct page. This is especially important if the pages where the ads are served are sensitive to inappropriate material, e.g. websites for kids, governments, or highly respected financial websites. Using the new directive, advertisers would be able to block navigations leading to incorrect ads. The policy is required because even if the target site for the ad is correct when the ad is bought, the website can at a later stage be hacked or misconfigured. Google Ads could, for example, serve the following policy with an ad from `shoes.com`. This would only allow navigation to `https://shoes.com`, blocking both the HTTP version, as well as, possible deep-links to apps like `app://shoes.com`. The `unsafe-allow-redirects` keyword allows for any number of server-side redirections before reaching `shoes.com`.

```
navigate-to  https://shoes.com 'unsafe-allow-redirects'
```

1.2 Research questions

The standardization [35, 61] and implementation [28, 36] efforts for `navigate-to` are well underway. The time is critical to ask questions on the security, performance, and adoptability of the proposed directive, before its adoption starts on the web. (Our analysis at the time of the writing confirms that the landing pages of Alexa's top 10,000 domains are yet to contain `navigate-to` CSP headers). By pursuing these questions, our goal is to deepen understanding of navigation policies and their impact, contribute to the emergence of the new standard, and to utilize our findings for settling the ongoing discussions by the community [29].

Security While there seems to be much to gain from a navigation policy, what is the impact on the security of the entire web ecosystem? For a fully-fledged security evaluation, we seek to uncover both new vulnerabilities and amplifying effects of known vulnerabilities. Our methodology is thus to investigate possibilities of exploiting the directive by a comprehensive range of attackers defined in the security literature [39]: injection [5], gadget [6], web [2] and passive network [25] attackers. Even though these attackers share some capabilities, they each have unique abilities, e.g. reading network traffic or hosting websites, and as such require individual analysis. This brings us to the questions of security: *Does the new policy "break the web"? Does the new policy introduce security vulnerabilities? How can they be mitigated and by whom?*

Automatization Once the new directive is secured, how can we aid its adoption? CSP has been notoriously hard to adopt, introducing insecure policies or broken websites [56, 57]. To help developers use the new directive, and increase both usability and adoptability, we investigate the possibility of automatically generating navigation policies. Hence, the question: *Can automatic mechanisms be used to help generate the new policy?*

Performance In contrast to CSP directives like `script-src`, intended to whitelist scripts that can be loaded by a webpage, the `navigate-to` directive will whitelist possible navigations. This results in already lengthy response headers becoming even larger, further increasing the overhead of security headers. This brings us to the question of performance: *What are efficient methods for delivering the new policy?*

1.3 Contributions

This paper is a first systematic evaluation of `navigate-to`. Our goal is to both initiate research on navigation security and to affect the emerging standards for navigation policies. We examine the security implications, efficiency, and the possibility of automatic generation of the new `navigate-to` policy.

Security The intricate connections between policies together with the growing complexity of the web results in new mechanisms becoming more challenging to incorporate into the ecosystem. This motivates the need to analyze multiple types of attackers, as well as, reexamining existing mechanisms in combination with new ones. We follow a methodology of examining the effects of `navigate-to` on a comprehensive range of attackers: injection [5], gadget [6], web [2] and passive network [25] attackers. By scrutinizing the full attack surface of the new directive, with respect to different types of attackers, we identify specification- and implementation-level vulnerabilities that can be exploited (Section 3). The vulnerabilities allow attackers to probe other websites to detect if users are logged in or have active shopping carts, bypass blocking mechanisms of third-party cookies, leak browsing history, and open up new methods for exfiltration. This demonstrates that the directive "breaks the web" in the sense of introducing vulnerabilities even in otherwise secure websites that do not use the directive in their policies. We present mitigations to security problems, both for web and policy developers (Section 4).

Automatization Looking ahead when the proposed mitigations are in place, our goal is to aid in the adoption of `navigate-to`. We develop AutoNav, an automatic mechanism for navigation policy inference (Section 5). AutoNav crawls websites and generates `navigate-to` policies. The goal of this mechanism is to simplify the deployment of the new directive by helping web developers and security engineers to find fitting policies for their websites. To further improve security, AutoNav can also generate origin-wide policies for the new origin policy delivery mechanism that is currently being drafted [59]. This improves security by applying the policy to the entire origin, covering pages that are easy to forget, like error pages. We implement and evaluate the mechanism by an empirical study (Section 6). In our experiments, we craw 100 pages per domain for 10,000 domains. Based on a subset of 80 pages, AutoNav generates a policy for the remaining 20 pages. For 42% of websites, AutoNav generated a policy which fully covered the 20 pages, and at 59% 19 out the 20 pages were covered. Further investigation into the category of websites shows that shopping websites and adult websites are the easiest to cover.

Performance To evaluate the performance impact of the policy we perform an empirical study (Section 6). Based on 10,000 crawled domains from Alexa's top 10,000, the policy will result in an overhead of 215 bytes for each HTTP response. We create simplification strategies to find a balance between security, performance and maintainability. These simplifications convert complicated policies with multiple subdomains to more manageable policies by using wildcards. For example, instead of including all language-specific subdomains from Wikipedia `navigate-to *.wikipedia.com` would be enough. Our simplification algorithm decreased the overhead by between 40% and 47%. Furthermore, we show that the use of an origin policy would result in an overhead of 1904 bytes in total, as opposed to per HTTP response. This is further decreased to 1004 bytes by using our simplification algorithm. A 900 byte reduction might not seem like much, but it can have a big impact on larger websites [21].

2 Background

Setting the background, we present the threat model in terms of relevant attackers. We describe CSP and how it relates to the origin policy. Finally, we explain navigation methods and how they are treated in the `navigate-to` directive.

2.1 Threat model

The main goal of the `navigate-to` directive is to give web developers control over where users can navigate from their website. The assets that need protecting include confidentiality, integrity and availability. Previous research has already shown how confidential information, such as cookies, can be exfiltrated using navigation [65]. While the new directive is a step in the right direction to address data exfiltration, Zalewski [65] points out that control over navigation is not necessarily enough. Attackers could, for example, inject HTML or JavaScript that change documents from private to public on a website like Dropbox. Forced navigation can also be used for phishing attacks by redirecting users to a similar-looking, but attacker-controlled, website.

Modern web browsers support many different methods for navigation, e.g. by clicking on a link, submitting a form, etc. These navigation methods, and the subset that the `navigate-to` directive is intended to apply to, are explained in Sections 2.4 and 2.5.

As mentioned above, we are interested in a comprehensive security evaluation of the impact of the directive on the entire web ecosystem. Hence, our threat model includes four types of attackers from the security literature [39]: injection, gadget, web and network attackers. In practice there is some overlap between the classes, for example, an attacker with *web attacker* capabilities will usually also have *injection attacker* capabilities. However, the best mitigation strategy might be different depending on which specific class we need to defend against. Therefore it is important to study each distinct class of attacker.

Injection attacker The *injection attacker* [5] is able to inject content into a website. A typical example is a user who can post content on a forum. If the user's post contains JavaScript then that code could be executed by other users on the site, in this scenario, with the goal to force a navigation.

Gadget attacker The *gadget attacker* [6] is similar but more powerful as they are allowed to host code, or gadgets, on other websites. A notable example is JQuery which is a JavaScript snippet that is used by many websites. Since JavaScript do not support any isolation, these gadgets run with the same capabilities as other scripts on the website. A malicious gadget could exfiltrate information from the website it is integrated to, modify content on pages or even navigate the user away from the website.

Web attacker The *web attacker* [2] is able to host and configure a full website. This is especially important for advertisers who want to ensure that the landing page does not redirect to anything other than what was specified in the ad.

Passive network attacker A *passive network attacker* [25] can listen in on all the traffic sent from and to a client but can not decrypt HTTPS. If the traffic is not encrypted, the attacker can read passwords and session cookies being sent to the server.

Note that `navigate-to` is not designed to handle network attacks. Yet we pay attention to network attackers in our effort to analyze the impact of the directive on the entire web ecosystem.

2.2 CSP

CSP is intended to mitigate cross-site scripting (XSS) and other code injection attacks. The current version of CSP, level 2, is supported by all major web browsers [26]. Level 3, which includes the new `navigate-to` directive, is being discussed and drafted [61].

CSP protects the users by specifying which resources and scripts are allowed on a page. The web server sends the CSP policies each time a user requests a page. These policies are then enforced by the browser to, among other things, block XSS. The policy below will only allow scripts to be loaded from the current origin, still blocking any injected inline scripts. In addition, the reporting header `Content-Security-Policy-Report-Only` [33] can be used to report policy violations without enforcing them. These reports are sent as POST requests to the server. They can also be detected using `SecurityPolicyViolationEvent` in JavaScript.

```
1  Content-Security-Policy: script-src 'self'
```

2.3 Origin policy

Today, CSP headers are sent with every HTTP(S) response, which is a concern for both safety and performance [50]. For security, it is easy to forget the policy on special pages, like error pages [59]. It also harms performance because servers need to repeat the same policy for each response, even if the policy should apply to all. To address this, specifications are being drafted [59], implemented [60], and evaluated [50] to enable *origin-wide policies*, known as *origin policies* [59] or *origin manifests* [50]. Using an origin policy, the server only needs to include once which policies should apply to the whole origin.

2.4 Navigation

Navigations can be performed in many different ways by browsers, e.g. by clicking on a link, submitting a form or running JavaScript. Navigation methods can be split into two different categories, user-initiated or document-initiated. While navigation is defined in the Fetch [52] and HTML [4] standards, the exact methods available depend on the web browser implementation. We make an effort to summarize the most common methods in Table 1. The *Automatic* column shows if the navigation method can be performed automatically. This is true for all JavaScript function and, in case JavaScript is allowed, `<a>` and `<form>` tags. It is worth noting that while a web page

cannot read a user's browsing history, it can initiate navigation to go back or forward in the browser history. There are many `.location` functions in JavaScript that can navigate, e.g. `window`, `document`, `parent`, etc. They all use the Location object defined in the HTML standard [4]. Some functions, like `window.navigate`, only works in Internet Explorer [11]. The last column specifies which methods `navgiate-to` affects.

Table 1: Navigation methods together with initiator and possibility to automatically navigate.

Method	Initiator	Automatic	Affected
`<a>` tag	Document	With JavaScript	✓
`<form>` tag	Document	With JavaScript	✓
`<meta>` tag	Document	Yes	✓
`<iframe>` tag [51]	Document	Yes	✓
`window.open` [53]	Document	Yes	✓
`*.location` [4]	Document	Yes	✓
`window.navigate` [11]	Document	Yes	✓
Typing the URL	User	No	
History buttons	User & Document	Yes	
Home button	User	No	

2.5 Navigate-to directive

The *navigate-to* directive gives developers the power to control the navigations a document can initiate. Document initiated navigations are discussed in Section 2.4. This directive makes it harder for attackers to inject code to redirect users from legitimate websites. For example, if an attacker manages to inject links on `disney.com` then Disney's reputation is at stake if links lead to inappropriate websites. To tackle this, Disney could add the following to their CSP policy:

```
navigate-to *.disney.com *.thewaltdisneycompany.com
```

This would instruct the browser to only accept navigations to subdomains of `disney.com` and `thewaltdisneycompany.com`, and block all navigations to other websites. The standard also introduces the new keyword `unsafe-allow-redirects`, which allows any redirects as long as the final destination is allowed by the policy. It is deemed less safe since it does not have full control over all the sites in the redirect chain. However, it is still better than nothing in terms of limiting navigations.

The `navigate-to` directive is currently being standardized by W3C [61] and implemented in Chrome [36] and Firefox [28]. It is available in the current version of Chrome (version 77.0.3865), and other Chromium-based browsers like Edge and Brave, behind a flag that enables experimental features. It is also available in Firefox Nightly (Version 71.0a1) behind a flag [28].

3 Vulnerabilities

This section presents vulnerabilities and security concerns related to the `navigate-to` policy. These vulnerabilities are not navigation attacks, but rather vulnerabilities that become possible due to `navigate-to`. Except for the last vulnerability in Section 3.3.3, where we rather want to show that a small improvement to `navigate-to` can solve an existing problem. The policy introduces new methods for acquiring privacy-sensitive information, circumvention of security mechanism and data exfiltration. All the attacks described in this section have been tested in practice. While some of the vulnerabilities, like the data exfiltration, relies on the existence of other vulnerabilities, like content injection, the `navigate-to` adds a new layer to the attacks. This possibility of combining attacks shows the importance of reexamining existing ones when introducing new mechanisms.

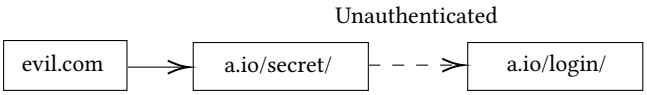

Figure 1: A user visiting `evil.com` will be navigated to `a.io/secret/`. If they are not logged in, they are further redirected to `a.io/login/`.

3.1 Methodology

To systematically find vulnerabilities we distinguish vulnerabilities relating to the specification of the `navigate-to` directive and vulnerabilities related to its implementation. For each category, we divide the investigation of vulnerabilities pertaining to confidentiality, integrity and availability, in accordance with the CIA triad. We draw on our threat model and examine vulnerabilities with respect to injection, web, gadget, and passive network attackers. Finally, we analyze how the directive can be used to circumvent modern countermeasures, such as third-party cookie blocking.

The presentation of the vulnerabilities is ordered by our estimate of their impact, from high to low. Table 2 lists the vulnerabilities we discover together with their corresponding attacker model. Interesting to note is that resource probing and Google Search profiling can be exploited to attack websites that themselves do not use the `navigate-to` directive. This results in previously security websites becoming insecure.

Table 2: The uncovered vulnerabilities together with corresponding attacker model.

Vulnerability / Attacker	Injection	Web	Gadget	Passive network
Resource probing		✓		
Google Search profiling		✓		
Third-party cookie bypass		✓		
History sniffing		✓		
Data exfiltration	✓		✓	
Ads leaking data				✓

3.2 Specification

The following vulnerabilities are present in the specification. This means that any browser following the specification correctly will be vulnerable.

3.2.1 Resource probing In cases where web applications redirect based on sensitive resources, these resources could be probed. For example, probing for the existence of Dropbox files. The probing attacks in this section are deterministic, as opposed to other attacks that rely on timings [55]. The attacks are also general and could potentially be used on any website, not solely on advertiser platforms such as the attack presented by Venkatadri et al. [54].

A malicious website, i.e. a web attacker, can navigate a user to `dropbox.com/preview/wallet.txt` to detect if a user has a file named `wallet.txt`. If no such file exists then the user is redirected to `dropbox.com/home/wallet.txt`, making it possible to craft a policy which blocks `/preview/` but not the redirection to `/home/`, like the following. Note here that we only use path-sensitivity to

block `/preview/`. If we are redirected, then path-sensitivity is no longer available and we only have to allow `dropbox.com`. The main difference compared to previous work on CSP redirections [23] is that we only need path-sensitivity for the first request, not the redirects.

```
navigate-to 'unsafe
      -allow-redirects' https://www.dropbox.com/not_preview/;
```

By utilising invisible iframes multiple files can be checked in parallel, without the user being navigated away from the malicious website.

One specific application of resource probing that has been researched before is login detection. Previous methods [20, 32] relies on third-party cookies, which can be blocked by the user or by the proposed default SameSite policy [62]. Instead, note that a navigation to `facebook.com/settings/` will redirect the user to the login page, `facebook.com/login.php`, if they are not authenticated, similar to Figure 1. By allowing only one of these URLs in the policy, the attacker can differentiate between a successful navigation and a blocked one. This feature makes our method more powerful and general.

We have also found that on some E-commerce websites it is possible to detect if a customer has anything in their shopping cart. This is because navigating directly to the shopping cart or checkout page sometimes redirects the user depending on the content of the cart. PrestaShop, which is an E-commerce platform used on hundreds of thousands of websites [8], does exactly this. By visiting `example.com/en/order` a user will be redirected to `example.com/en/cart`, assuming `example.com` uses PrestaShop.

Some of the probing attacks can leak more data if they are done in an active fashion. The PrestaShop attack can be improved to, in theory, enumerate the full cart. This is due to a Cross-Site Request Forgery (CSRF) [47] vulnerability in PrestaShop, currently being disclosed, which allows an attacker to add and remove items. Using this method an attacker can repeatedly remove items and then check if the cart is empty.

These are only a few examples we have found where redirects are based on sensitive data. We believe that many more such redirects currently exists on the web. Furthermore, navigations can bypass `lax` SameSite cookies, making the attack possible on sites where previous CSRF attacks were not possible.

3.2.2 Google Search profiling Google Search relies on *personalized search* [19], meaning that the results of a search query are based on the users' previous interactions with Google. A recent study [24] shows that users are put into so-called "filter bubbles" by Google, resulting in varying result when searching for political terms such as "gun control" or "immigration". A web attacker can craft a malicious website which uses the `navigate-to` directive together with Google's *I'm feeling lucky* function to extract top results from visitors. This type of extraction attack is called cross-site search and has previously been successfully mounted against Gmail and other websites [17]. The main difference is that previous methods have relied on timing, whereas our method is fully deterministic. Castelluccia et al. [10] were also able to infer sensitive information about users based on Google Searches. However, their approach required network attacker capabilities and assumed the traffic was unencrypted, which is not the case anymore. Our attack can be mounted by anyone with the capability to set up a website.

The attacker can then use these top results from Google to infer these filter bubbles. Using the URL `https://www.google.com/search?q=QUERY&btnI`, Google will automatically redirect the user the top result for term "QUERY". Therefore the *I'm feeling lucky* function acts as an open redirector, which is something both OWASP [46] and Google [34] themselves warn about. It is well known that Google has this problem but so far they choose to accept the risk [1]. However, the `navigate-to` directive adds a new dimension to the problem as it enables attackers to infer data about users.

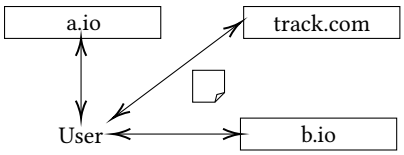

Figure 2: When a user visits `a.io` or `b.io`, they can force the user to obtain first-party cookies from `track.com`.

To exploit this the attacker can specify a report-only policy that only allows `google.com`, as shown below. The redirect will violate the policy and the browser will dutifully report which domain was in violation to the malicious website. The attacker can iteratively update the query to get more results. Assuming searching for "news" would return `news.com`, then the next query would be "news -site:news.com", which excludes `news.com` and perhaps returns `reports.com` instead. Another attack vector would be other search engines using this approach to directly copy personalized search results from Google, similar to what Bing did [41].

```
Content-Security-Policy-Report
      -Only: navigate-to 'unsafe-allow-redirects' google.com
```

3.2.3 Third-party cookie bypass A cookie is a piece of data that websites can save locally on users' machines. [31] Depending on how the cookie is acquired, it will either be considered a first-party cookie or a third-party cookie. A navigation will result in first-party cookies while image request and similar results in third-party cookies.

Third-party cookies are useful for advertisers [14] as it allows them to use small tracking pixels [15] for tracking users. Modern browsers allow users to block third-party cookies or do it by default [42].

Previous work has demonstrated how Cookie Synchronization [7, 38] can be used by ad platforms to effectively break the same-origin policy. Privacy-aware users can mitigate this by blocking third-party cookies altogether. However, the `navigate-to` directive introduces a new method for advertisers to circumvent this by using navigations. As it requires control over the CSP headers, web attackers are the main threat. Figure 2 shows a user visiting `a.io`, then being forcibly navigated to `track.com` and acquiring a first-party cookie. Using the following policy, the redirection will be blocked, making the attack unnoticeable to the user.

```
navigate-to 'unsafe-allow-redirects'
```

3.3 Implementation

The following vulnerabilities are due to implementation decisions. We focus on Chrome's [36] and Firefox's [28] implementations of `navigate-to`,

3.3.1 History sniffing The `navigate-to` policy can, in some cases, be exploited by a web attacker with a malicious website to probe

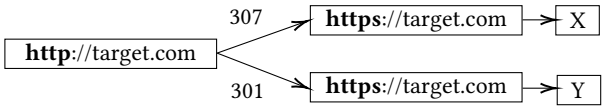

Figure 3: If `target.com` uses HSTS, and the user has visited the site before, then the browser will automatically upgrade the connection to HTTPS using a 307 redirect instead of a server side 301.

which websites a user has visited. The attack uses the fact that websites using HSTS force the browser to remember and upgrade insecure connections. Previous methods exploiting this have relied on timing attacks which are now mitigated [64].

Using `navigate-to`, a malicious website can make a POST request to another site which uses HSTS but is not preloaded. If the site redirects based on the POST data then the attacker might be able to detect if a user has visited the site before. This is possible because if the user has visited the site before it will result in an internal redirect (HTTP 307), which keeps the POST data. Otherwise, the server will redirect (HTTP 301/302), which drops the POST data. If the server specifically performs a 307 redirect then the attack will not work. By crafting a CSP that does not allow the redirect, the attacker can differentiate between the two cases, denoted X and Y in Figure 3. This is, for example, possible using the login function on the popular social media website VK.

3.3.2 Data exfiltration and communication Previous research has shown that data exfiltration is possible in the face of CSP [49]. The usage of forms and links to exfiltrate data has also been studied [65]. However, the `navigate-to` policy introduces an improved method for exfiltration, and two-way communication, based on JavaScript together with navigation. This works in Chrome, but not Firefox, as Chrome does not unload the page for `navigate-to` violations.

Consider a website using `connect-src 'none'` and `frame-src 'none'` to limit external loads as much as possible. The `connect-src` directive protects against some exfiltration methods including XHR, fetch and `<a ping>`, while `frame-src` will block exfiltration to iframes. Assume the website uses `unsafe-allow-redirects` followed by a list of allowed URLs. Note here that we show that *unsafe* has implications beyond the scope of restricting navigation. An attacker capable of injecting JavaScript, i.e. either an injection or gadget attacker, can now use `window.location`, as shown in the listing below, to exfiltrate arbitrary data. Each navigation request will exfiltrate data, then be blocked by the policy, as the attacker can choose a website outside the CSP whitelist. Furthermore, by adding a `SecurityPolicyViolation` event listener the attacker can inspect the blocked URI in the violation. To send a response, `evil.com` would redirect the request to a subdomain like `<msg>.evil.com`.

```
function exfiltrate (data) {
  window.location = "http://evil.com/?d=" + data;
}
```

The main difference between not using `navigate-to` and using the policy described is that by blocking the navigations, the control is returned to the attacker, allowing for further stealth exfiltration and communication.

3.3.3 Ads leaking data We have found that ads served over HTTPS can still leak the final landing page to a passive network attacker if an ad in the redirection chain is unencrypted. While network-level eavesdropping is outside of CSP's threat model, the `navigate-to` directive presents a great opportunity to fix this problem. The problem stems from the fact that when a user clicks on an ad they can be channeled through multiple tracking websites. Listing 1 shows a chain where the user is redirected to three different websites before the landing page. We performed a small empirical study using the same dataset as in Section 6. We extracted all iframes and compared their source URL to a list of known advertisement platforms, e.g. DoubleClick. If the URL matched we followed it and recorded the redirects. This resulted in 24650 unique ads, of which 26.7% have a website between the advertisement platform and the landing page. This highlights the need for advertisement platforms to consider potential redirects from tracking websites and further motivates the need for the `navigate-to` directive.

```
https://www.googleadservices.com/...
http://www.kqzyfj.com/...
http://cj.dotomi.com/...
http://www.emjcd.com/...
https://<landing page>/...
```

Listing 1: Example of an ad chain containing three different unencrypted domains between the encrypted ad platform and landing page.

As can be seen in Listing 1, both the first and last websites use HTTPS but there exist sites between that are unencrypted. This is very hard for a user to detect as both the ad and the landing page seems secure. The problem with having HTTP in the chain is that an eavesdropper can follow the request and find the landing page. Our empirical experiments show 10.6% of the ads follow this pattern. As ads become more personal this becomes a privacy concern. Advertisements related to economic status or specific diseases might be leaked without the user's knowledge.

4 Countermeasures

This section presents countermeasures to the vulnerabilities in 3. The countermeasures cover the specification, mitigations for web developers, as well as, implementation improvements in web browsers. Similarly to the vulnerabilities in Section 3 we distinguish specification- and implementation-level countermeasures.

4.1 Specification

4.1.1 Resource probing Previous login detection methods have forced web developers to rewrite their applications to avoid special types of redirections. As mentioned in [13], Google added an extra regex check to make sure the redirection did not lead to resources that could be loaded cross-origin, e.g. "jpg", "js" and "ico".

The `navigate-to` policy circumvents this by being able to block and report different paths in the URL, i.e. it is possible to block `example.com/settings/` and allow `example.com/login/`. In this case, if `/settings/` redirects to `/login/` for unauthenticated users, then the CSP report log can be inspected to discern between authenticated and unauthenticated users.

To fix this, path precision could be removed from the policy. If an origin as a whole can not be trusted, it seems to add little security to trust certain paths on the origin. Since these vulnerabilities affect websites that do not use `navigate-to`, we also present countermeasures web developers can implement. We recommend avoiding redirection based on secrets. Instead, by showing an error page or rendering the login form on the same page the website is guaranteed to not leak any data, as there will be no redirections. If redirection is necessary, encoding paths in GET parameters, e.g. from `example.com/files/` to `example.com/?path=/files/`, also mitigates the problem.

4.1.2 Google Search profiling For vulnerabilities like Google Search profiling, as presented in Section 3.2.2, the key countermeasure is to avoid open redirects [46]. One possible way for Google to accomplish this without removing the *I'm feeling lucky* function is to use a CSRF token [47].

4.1.3 Third-party cookie bypass The navigation path through the redirection chain can depend on the user's cookies. For this reason, it is not possible to block cookies while checking if the navigation is allowed. Instead, we suggest that cookies attained during the check are temporarily sandboxed and then removed if the navigation is blocked.

4.2 Implementation

4.2.1 History sniffing Privacy problems related to HTTP Strict Transport Security (HSTS) [22] has been researched before [44]. However, they focused on tracking mechanisms similar to cookies but harder to remove.

The solution is to ensure that an attacker can not differentiate between the paths in Figure 3. Again, it becomes the web developers responsibility to either use an internal redirect or not redirect on post data.

4.2.2 Data exfiltration What makes this attack extra powerful is its ability to regain execution control after the navigation fails. It is not specified what should happen when the `navigate-to` policy blocks a navigation attempt. Currently, Chrome seems to simulate a 204 response [58], resulting in the continuation of the script, and the possibility to exfiltrate more data. Firefox, on the other hand, uses a full-page error that unloads the original document. By using this strategy the script will stop executing, blocking further exfiltration. The attack can also be mitigated by avoiding `unsafe-allow-redirects`, as this will block the exfiltration during the pre-navigation check.

4.2.3 Ads leaking data The `navigate-to` directive could block redirect chains which contain HTTP websites. Currently, the policy `navigate-to https:` allows navigation to any website using HTTPS. However, combined with `unsafe-allow-redirects` HTTP is allowed in the chain, as long as the landing page is HTTPS. One solution is to add a value `unsafe-allow-https-redirects` which would only allow redirection by HTTPS. A more general solution is to split the policy into `navigate-to` and `navigate-by`, where the latter would apply as long as the request is redirected. When no redirect is received, the landing page is checked against the `navigate-to` policy. By using this method, the following policy would allow any HTTPS redirections which lead to `https://example.com`.

```
navigate-to https://example.com
navigate-by https:
```

5 AutoNav

We present AutoNav, an automatic mechanism to aid web developers in inferring policies for their websites. The mechanism crawls the website and creates a map of where pages can navigate. This mapping is used to generate and simplify the policies. AutoNav can generate both per-page policies, where each page on a website gets its own policy, and origin-wide policies [59].

5.1 Inference

We use a key-value map from the crawler to infer the policies. The page is used as a key, and a list of all possible navigations from the page is used as a value. Listing 2 shows an example.

```
{
  "example.com/a.html": [facebook.com, google.com],
  "example.com/b.html": [twitter.com, google.com]
}
```

Listing 2: Example of a key-value map generate from crawling two pages on `example.com`

Using the key-value map, AutoNav can generate separate policies for each page on the website. This is shown in Listing 3. AutoNav can also generate an origin-wide policy based on the union of all the URLs, as shown in Listing 4. These policies are then simplified, using the method described in Section 5.2, to reduce the size and improve maintainability.

```
{
  "a.html": "navigate-to facebook.com google.com",
  "b.html": "navigate-to twitter.com google.com"
}
```

Listing 3: Per-page policies generated from Listing 2.

```
{
  "*": "navigate-to facebook.com twitter.com google.com"
}
```

Listing 4: Origin-wide policy generated from Listing 2.

5.2 Policy generation

The navigation policy is a whitelist of URLs that the user is allowed to navigate to. In the most secure setting, the policy should contain the full URLs to each allowed target. While secure, this creates big and hard to maintain lists of URLs requiring much bandwidth. Take Wikipedia for example, their policy could consist of all subdomains like `en.wikipedia.org`, `es.wikipedia.org`, etc. for each language. A more compact policy is `*.wikipedia.org`. This simplification results in both less data being transmitted and a more maintainable policy, however, it does decrease security as it also allows `evil.wikipedia.org`.

AutoNav supplies developers with best-effort policies that aim to help them harden their websites. Using our parameterized simplification algorithm, developers get a slider style method for finding a trade-off between maintainability, performance and security. The simplification algorithm looks for evidence that all subdomains are trusted. The two sources used are the number of URLs that point to the subdomains (denoted t_1) and the number of subdomains that are pointed to (denoted t_2). The motivation for t_2 is that even if multiple links are found to `a.example.com` it does not imply

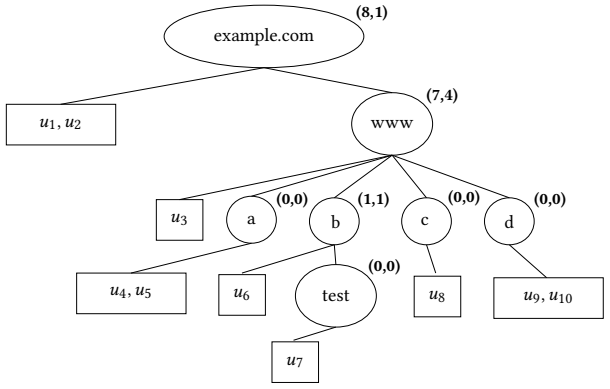

Figure 4: Tree representation of 10 URLs collected from `example.com` and its subdomains. The tuples corresponds to the (t_1,t_2) thresholds.

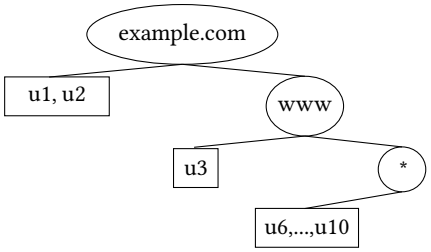

Figure 5: Result of applying the simplification algorithm, using a treshold of $(2,2)$, to the tree in Figure 4. Resulting in the following policy, `navigate-to example.com www.example.com *.www.example.com`.

that `b.example.com` should be allowed. Similarly, t_1 is motivated by the notion that the more URLs that point to `*.example.com`, the more it can be trusted. Figure 4 shows the tree representation of 10 URLs pointing to `example.com` and its subdomains. u_i in the figure represents one URL, e.g. u_7 points to a resource on `test.b.www.example.com`. Furthermore, the figure also includes tuples of the threshold values (t_1,t_2). Figure 5 shows the tree after simplification using a threshold of $(2,2)$. Using this method the policy will only contain 3 entries instead of 7 entries.

Figure 6 shows the result from crawling five pages on `ebay.com` and generating a policy. The crawler was only supplied with the start page and then found the other four using the crawling algorithm from Section 5.3. The five pages crawled are shown in the middle of the figure in grey with integer labels. The arrows from these nodes indicate that a possible navigation was found between two nodes. The colors correspond to which part of the policy covers the navigation. As shown, `*.ebay.com` covers a lot of the subdomains, thus they all share the same color. Using the figure, an origin policy could be generated by taking the union of all the colors.

This method of generating policies guarantees that the functionality of the website will remain intact. This is because, if a domain is in the list of possible navigations, then it will be included in the policy. Similar to other policies, the generated policy would need to be recalculated if the website was updated to include new possible navigations. For security, the method guarantees that if a domain is not in the list, then it will not be added to the policy. However, subdomains of domains in the list can be added to the policy.

5.3 Crawling

Our implementation of AutoNav uses selenium with a Chrome instance to crawl the pages on a website. By only supplying AutoNav with the first URL it will automatically collect and crawl new URLs that it finds. When a URL from the same website is found it is added to a set of unvisited URLs, from which the next URL is picked. For each page on a domain, all the JavaScript is executed, then the URLs from links and forms are saved. When the crawling session is over, the inference method described in Section 5.1 is used to generate the policy.

5.4 Limitations

We did not take special care to crawl behind the login. However, it is trivial for a site owner to add a session cookie to the crawler. The more pages AutoNav can crawl the more the policy will cover. Crawling too few pages will result in an incomplete yet secure policy. The policy is secure because AutoNav will never add a domain to the policy that has not been seen.

We use static links to infer the policies, which will miss possible redirections. While not a security concern, we would produce more precise policies if each link was followed dynamically and the redirections recorded.

User-agent sniffing is a common problem for crawling studies. Since the AutoNav is designed for developers we think they can manually add entries like `languages.mysite.org` and use the AutoNav to detect everything else.

6 Empirical Study

This section presents an empirical study to evaluate the performance impact of the new directive, as well as, how different delivery methods and simplifications can reduce the impact. Next, we evaluate AutoNav in how well automatically generated policies based on a subset of the website cover the full website.

To test how the new `navigate-to` policy will function on common websites we utilize AutoNav in a crawling experiment. For calculating the performance impact in Section 6.1, we use Alexa's top 10,000 websites. For evaluating AutoNav itself we use Alexa's top 14,000, ensuring we have 10,000 domains which all have more than 100 pages each.

6.1 Policy tradeoffs

This section presents the performance tradeoffs between per-page and origin-wide policies together with the delivery methods of HTTP headers and origin policy.

The costs in Table 3 are based on a user visiting n pages on a website, thus the cost of HTTP headers need aggregation over all pages, i.e. $\sum_{i \leq n}$. The cost of sending a single CSP policy depends on the number of URLs it contains. We defined the cost of the policy based on the set of URLs, i.e. $|U_i|$, U_i being the set of URLs on page i. Further, we can define a set of all URLs as the union of the sets of URLs on each page as $\bigcup_{j \leq n} U_j$, with corresponding

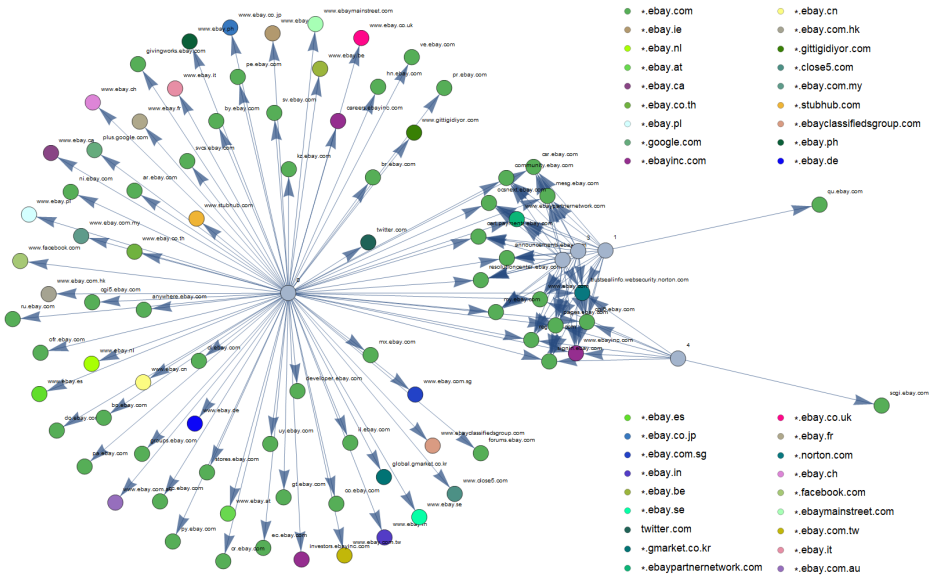

Figure 6: Generated policies for `ebay.com`. The nodes with outwards pointing arrows are the five pages that we crawled. All the other nodes correspond to a possible navigation. The color indicates which part of the policy covers the navigation.

Empirical performance Based on the 10,000 crawled domains, a per-page policy, without any simplifications, would increase the header size with 215 bytes, per response. A more maintainable origin-wide policy results in a size increase to 1904 bytes. This cost can be decreased by using the origin policy for delivery, in which case the user only downloads the policy once. Note, as shown in Table 3, that an origin policy outperforms a per-page policy after only 9 responses. While per-page policies might seem better, they are difficult to use since they require knowledge about the content on each page. As such, some website, e.g. Facebook, use origin-wide policies, motivating the need for an origin policy delivery method.

In addition to the comparison between per-page and origin policy, we also evaluated the cost benefits of using our policy simplification algorithm. Using maximum simplifications, i.e. $t_1 = 1, t_2 = 1$, the average size of the origin wide policy decreases from 1904 to 1004 bytes, a decrease of 47%. Similarly, the per-page policy decreases from 215 bytes to 129 bytes, which is a 40% decrease. For some websites, the benefit of simplification is much greater. In particular, this is the case when websites allow navigation to numerous subdomains. For example, `spravker.ru` would require a 20438 byte origin policy without simplification, but only 61 bytes after simplification. The big difference stems from the fact that `spravker.ru` have 954 subdomains.

Table 3: Empirical costs for different policy models.

	HTTP	Origin Policy
Per-page	$\sum_{i \leq n} 215$	-
Origin-wide	$\sum_{i \leq n} 1904$	1904

We also performed a more in-depth analysis of three websites, `ebay.com`, `wikipedia.org` and `stackexchange.com`, to see how

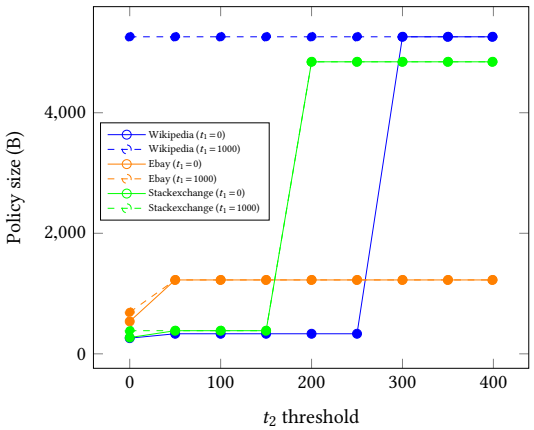

Figure 7: Cost of origin policy for different domains and simplification thresholds. The y-axis shows policy size in bytes and the x-axis shows the t_2 threshold. The legend shows the t_1 threshold

the threshold affect performance. Fixing t_1 to 0, we only focus on the number of subdomains when deciding if wildcards should be used. Figure 7 shows these domains as solid lines, together with the corresponding costs for their origin policies. As can be noted, after the t_2 threshold reaches 280 subdomains Wikipedia can no longer use the wildcard and the policy quickly increases in size. By increasing t_1 to 1000, more URLs are required before simplifications can take place. As can be seen in the dashed lines in Figure 7, the crawled data from Wikipedia did not contain enough URLs to the same domain for a simplification. This would be the desired behavior if Wikipedia required high assurance before introducing wildcards.

6.2 Coverage

While full coverage may be desirable, the goal of AutoNav is to help even if the coverage is not complete, by providing a useful baseline policy for developers to build on.

Our coverage was generated similarly to the method used in CSPAutoGen [37]. We generate the policy based on a training set of 80% of the pages on a domain and then test how well they match the other 20%. We define U as the set of URLs in the training set. For the n pages in the validation set, we check if URLs on the page are covered, i.e. $p_i \subseteq U$, where p_i is the set of all the URLs on page p_i. Finally the coverage of a website is calculated as: $c = \frac{|\{p_i : p_i \subseteq U\}|}{n}$

Using this formula, c is calculated for all the websites that were crawled. In total 42% of all the websites were fully covered and for 59% of the websites 95% or more were covered. Note that these results come from only crawling 100 pages, deeper crawls can greatly increase this coverage.

7 Related work

Automatic methods for generating CSP policies have been studied before [12, 16, 18, 37]. deDacota by Doupe et al. [12] performs static analysis of ASP.NET code in order to separate JavaScript code from data. After the JavaScript has been separated into files, a CSP policy was generated for the file. AutoCSP by Fazzini et al. [16] takes a similar approach by analysing server-side code, PHP in this case. However, AutoCSP uses dynamic taint tracking instead of static analysis, allowing it to create policies for inline JavaScript events and CSS code. While both AutoCSP and deDacota were successful, they required access to the source code of the application. In contrast, AutoNav uses a black-box approach which removes the need for the source code. Furthermore, the aforementioned methods focus on JavaScript and CSS, while our focus is on navigation and URLs. In addition, static analysis of source code will miss many URLs since modern web applications, like WordPress, store content in the database and not in the code.

In addition, research has been done on generating policies without access to the source code. Golubovic's autoCSP [18] method utilizes a reverse proxy and the report function in CSP to run an application in *learning mode*. In this mode, the tool externalizes inline code and generates policies for the scripts that should be allowed. A drawback is that autoCSP requires manual navigation through the application to ensure all scripts are triggered. While this works well for scripts, it becomes challenging when all possible links need to be navigated. A similar approach based on the report function in CSP was utilized by King's Firefox extension Laboratory [30]. Laboratory is impressive as it enables users to record and generate CSP policies in real-time while visiting a website. Starting with a strict policy, it gradually weakens it as violation reports are received. While this method could be extended to include navigations, it would require the user to initiate all possible navigations on each page. Instead of relying on the reporting functionality, our method uses a combination of static and dynamic analysis to record the navigations a document can initiate. By doing this we avoid the problem of having to initiate all navigations to generate a report. We also improve on the manual aspect of traversing a website by implementing an automatic crawler, as suggested by Golubovic, in future works.

CSPAutoGen Pan et al. [37] is also intended to automatize CSP generation. CSPAutoGen uses a crawler to analyze websites and try to infer which scripts should be allowed. Similar scripts are also generalized into abstract syntax trees, based on how many similar scripts are found. Once a policy has been inferred, CSPAutoGen functions as a proxy between the client and the server. This enables CSPAutoGen to rewrite requests and responses in real-time, without needing any CSP configurations on the website. This is a great feature when a server needs to be secured without any direct modification. While a similar approach could be used for URLs and navigation, our goal is to generate CSP policies that can be used by the server directly.

In addition to policy generation, we benefit from origin-wide policies [59]. Similarly to the work on evaluating general origin-wide policies by Van Acker et al. [50], our results also indicate that an origin-wide policy provides additional security without degrading performance.

8 Conclusion

Security We have performed a security analysis of the emerging CSP directive `navigate-to`. Our findings show that the current specification and implementations introduce new vulnerabilities. The vulnerabilities include methods for resource probing, login detection, circumventing blockage of third-party cookies, as well as, history enumeration. To mitigate these problems we propose countermeasures to both the specification and implementation of the directive. We demonstrate that the directive triggers vulnerabilities even in websites that do not use the directive in their policies. Thus, we also propose countermeasures web developers can make to their applications in order to mitigate the possibilities of being exploited.

Automatization We have evaluated the possibility of automatically generating policies to help developers adopt the policy, we created AutoNav. AutoNav uses a black-box approach to crawl websites and generate CSP policies that can be directly applied to the website. Our results show that in total 42% of all the websites were fully covered and for 59% of the websites 95% or more were covered. We further simplify the process by identifying categories of websites which the policy better fits. Our research shows that shopping and adult websites are best covered. These websites have a high incentive to keep the users on their site, with the exception of linking to sponsors or partners, which AutoNav's policies cover.

Performance To analyze the performance of `navigate-to` we have conducted an empirical study of Alexa's top 10,000 websites. For each website, we have crawled 100 pages and based on theses generated policies. We show that on average this directive would increase the header size by 215 bytes per request. However, using our simplification algorithm we produce more maintainable policies which were also 40% smaller on average. Our results indicate that using an origin policy would require a one time cost of 1904 bytes, or 1004 using simplifications, as opposed to 215 bytes per request. Thus we show that the performance hit from the increased security can be efficiently mitigated by adopting an origin policy with suitable simplifications.

Coordinated disclosure We are in the process of disclosing the discovered vulnerabilities to the affected vendors, including Google where both Chrome's implementation of `navigate-to` directive and the Google Search website are affected. Based on our recommendations Firefox chose to harden their implementation against exfiltration attacks, as explained in Section 4.2.2.

Acknowledgments

Thanks are due to Mike West, Christoph Kerschbaumer, and Daniel Hausknecht for helpful discussions on the topic of `navigate-to`. This work was partly funded by the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (VR).

References

[1] Feross Aboukhadijeh. 2011. Is Google an Open Redirector? https://feross.org/is-google-an-open-redirector/

[2] Devdatta Akhawe, Adam Barth, Peifung E Lam, John Mitchell, and Dawn Song. 2010. Towards a formal foundation of web security. In *Computer Security Foundations Symposium (CSF)*. IEEE, 290–304.

[3] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. 2018. {NAVEX}: Precise and Scalable Exploit Generation for Dynamic Web Applications. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 377–392.

[4] Apple, Google, Mozilla, Microsoft. 2019. HTML Living Standard. https://html.spec.whatwg.org/multipage/history.html#the-location-interface

[5] Adam Barth, Collin Jackson, and John C Mitchell. 2008. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 75–88.

[6] Adam Barth, Collin Jackson, and John C. Mitchell. 2009. Securing frame communication in browsers. *Commun. ACM* 52, 6 (2009), 83–91. https://doi.org/10.1145/1516046.1516066

[7] Muhammad Ahmad Bashir, Sajjad Arshad, William Robertson, and Christo Wilson. 2016. Tracing information flows between ad exchanges using retargeted ads. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 481–496.

[8] BuiltWith Pty Ltd. 2018. PrestaShop Usage Statistics. https://trends.builtwith.com/shop/PrestaShop

[9] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. 2018. Semantics-Based Analysis of Content Security Policy Deployment. *ACM Trans. Web* 12, 2, Article 10 (Jan. 2018), 36 pages. https://doi.org/10.1145/3149408

[10] Claude Castelluccia, Emiliano De Cristofaro, and Daniele Perito. 2010. Private information disclosure from web searches. In *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 38–55.

[11] Dottoro. 2019. navigate method (window). http://help.dottoro.com/ljuhrdju.php

[12] Adam Doupé, Weidong Cui, Mariusz H. Jakubowski, Marcus Peinado, Christopher Kruegel, and Giovanni Vigna. 2013. deDacota: toward preventing server-side XSS via automatic code and data separation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (Berlin, Germany) *(CCS '13)*. ACM, New York, NY, USA, 1205–1216. https://doi.org/10.1145/2508859.2516708

[13] Ahmed Elsobky. 2016. Novel Techniques for User Deanonymization Attacks. https://0xsobky.github.io/novel-deanonymization-techniques/

[14] Steven Englehardt and Arvind Narayanan. 2016. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 1388–1401.

[15] Facebook Inc. 2018. Use Facebook Pixel. https://www.facebook.com/business/help/952192354843755

[16] M. Fazzini, P. Saxena, and A. Orso. 2015. AutoCSP: Automatically Retrofitting CSP to Web Applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 336–346. https://doi.org/10.1109/ICSE.2015.53

[17] Nethanel Gelernter and Amir Herzberg. 2015. Cross-site search attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1394–1405.

[18] Nicolas Golubovic. 2013. autoCSP - CSP-injecting reverse HTTP proxy. https://golubovic.net/thesis/bachelor.pdf

[19] Google Inc. 2009. Personalized Search for everyone. https://googleblog.blogspot.com/2009/12/personalized-search-for-everyone.html

[20] 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. *CoRR* abs/1808.07359 (2018). arXiv:1808.07359 http://arxiv.org/abs/1808.07359

[21] Scott Helme. 2018. Optimising Twitter's CSP header. https://scotthelme.co.uk/optimising-twitters-csp-header/

[22] J. Hodges, C. Jackson, and A. Barth. 2012. *HTTP Strict Transport Security (HSTS)*. RFC 6797. RFC Editor. http://www.rfc-editor.org/rfc/rfc6797.txt

[23] Egor Homakov. 2014. Using Content-Security-Policy for Evil. http://homakov.blogspot.com/2014/01/using-content-security-policy-for-evil.html

[24] DuckDuckGo Inc. 2018. Measuring the "Filter Bubble": How Google is influencing what you click. https://spreadprivacy.com/google-filter-bubble-study/

[25] Collin Jackson and Adam Barth. 2008. Forcehttps: protecting high-security web sites from network attacks. In *Proceedings of the 17th international conference on World Wide Web*. ACM, 525–534.

[26] John Karahalis. 2018. Content Security Policy (CSP). https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

[27] Kasper Karlsson. 2017. 179426 Reflected XSS on blockchain.info. https://hackerone.com/reports/179426

[28] Christoph Kerschbaumer. 2018. 1529068 - Implement CSP 'navigate-to' directive. https://bugzilla.mozilla.org/show_bug.cgi?id=1529068

[29] April King. 2016. Allow navigation to only whitelisted URLs via navigate-to 125. https://github.com/w3c/webappsec-csp/issues/125

[30] April King. 2018. april/laboratory. https://github.com/april/laboratory

[31] D. Kristol and L. Montulli. 2000. *HTTP State Management Mechanism*. RFC 2965. RFC Editor.

[32] Robin Linus. 2017. Your Social Media Fingerprint. https://robinlinus.github.io/socialmedia-leak/

[33] Joe Medley. 2018. Content-Security-Policy-Report-Only. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy-Report-Only

[34] Jason Morrison. 2009. Open redirect URLs: Is your site being abused? https://webmasters.googleblog.com/2009/01/open-redirect-urls-is-your-site-being.html

[35] Andy Paicu. 2018. CSP 'navigate-to' directive: Consensus & Standardization. https://www.chromestatus.com/feature/6457580339593216

[36] Andy Paicu. 2018. Implement the 'navigation-to' directive. https://bugs.chromium.org/p/chromium/issues/detail?id=805886

[37] 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. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. ACM, New York, NY, USA, 653–665. https://doi.org/10.1145/2976749.2978384

[38] Panagiotis Papadopoulos, Nicolas Kourtellis, and Evangelos Markatos. 2019. Cookie synchronization: Everything you always wanted to know but were afraid to ask. In *The World Wide Web Conference*. ACM, 1432–1442.

[39] Philippe De Ryck, Lieven Desmet, Frank Piessens, and Martin Johns. 2014. *Primer on Client-Side Web Security*. Springer.

[40] Google Blink Security. 2018. Communication with Google's Blink security team.

[41] Ryan Singel. 2011. Google Catches Bing Copying; Microsoft Says 'So What? https://www.wired.com/2011/02/bing-copies-google/

[42] Nick Statt. 2017. Advertisers are furious with Apple for new tracking restrictions in Safari 11. https://www.theverge.com/2017/9/14/16308138/apple-safari-11-advertiser-groups-cookie-tracking-letter

[43] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. 2019. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild.. In *NDSS*.

[44] Mark Stockley. 2015. Anatomy of a browser dilemma - how HSTS supercookies make you choose between privacy or security. https://nakedsecurity.sophos.com/2015/02/02/anatomy-of-a-browser-dilemma-how-hsts-supercookies-make-you-choose-between-privacy-or-security/

[45] The OWASP Foundation. 2017. OWASP Top 10 - 2017. https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf

[46] The OWASP Foundation. 2017. Unvalidated Redirects and Forwards Cheat Sheet. https://www.owasp.org/index.php/Unvalidated_Redirects_and_Forwards_Cheat_Sheet

[47] The OWASP Foundation. 2018. Cross-Site Request Forgery (CSRF). https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)

[48] The OWASP Foundation. 2018. Cross-site Scripting (XSS). https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)#Stored_and_Reflected_XSS_Attacks

[49] 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*. ACM, 853–864.

[50] Steven Van Acker, Daniel Hausknecht, and Andrei Sabelfeld. 2018. Raising the Bar: Evaluating Origin-wide Security Manifests. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. 342–354. https://doi.org/10.1145/3274694.3274701

[51] Anne van Kesteren. 2018. Fetch Standard. https://fetch.spec.whatwg.org/#concept-request-destination

[52] Anne van Kesteren. 2019. Fetch Living Standard. https://fetch.spec.whatwg.org/#concept-request-destination

[53] Mehdi Vasigh. 2018. Window.open(). https://developer.mozilla.org/en-US/docs/Web/API/Window/open

[54] G. Venkatadri, A. Andreou, Y. Liu, A. Mislove, K. P. Gummadi, P. Loiseau, and O. Goga. 2018. Privacy Risks with Facebook's PII-Based Targeting: Auditing a Data Broker's Advertising Interface. In *2018 IEEE Symposium on Security and Privacy (SP)*. 89–107. https://doi.org/10.1109/SP.2018.00014

[55] Takuya Watanabe, Eitaro Shioji, Mitsuaki Akiyama, Keito Sasaoka, Takeshi Yagi, and Tatsuya Mori. 2018. User Blocking Considered Harmful? An Attacker-Controllable Side Channel to Identify Social Accounts. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 323–337.

[56] 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. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1376–1387.

[57] Michael Weissbacher, Tobias Lauinger, and William Robertson. 2014. Why is CSP failing? Trends and challenges in CSP adoption. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 212–233.

[58] Mike West. 2016. Allow navigation to only whitelisted URLs via navigate-to 125. https://github.com/w3c/webappsec-csp/issues/125#issuecomment-251975791

[59] Mike West. 2017. Origin Policy. https://wicg.github.io/origin-policy/

[60] Mike West. 2017. Origin Policy. https://bugs.chromium.org/p/chromium/issues/detail?id=751996

[61] Mike West. 2018. Content Security Policy Level 3. https://www.w3.org/TR/CSP3/

[62] Mike West. 2019. Incrementally Better Cookies. https://tools.ietf.org/html/draft-west-cookie-incrementalism-00

[63] Mike West, Adam Barth, and Dan Veditz. 2016. Content Security Policy Level 2. https://www.w3.org/TR/CSP2/

[64] Yan (bcrypt). 2015. @bcrypt - Advanced Browser Fingerprinting - Toorcon 2015. https://www.infosecurity.us/blog/2015/11/8/toorcon-2015-advanced-browser-fingerprinting

[65] Michal Zalewski. 2011. Postcards from the post-XSS world. http://lcamtuf.coredump.cx/postxss/