Rethinking the Use of Resource Hints in HTML5: Is Faster Always Better!?

N. Vlajic, X. Y. Shi, H. Roumani and P. Madani

Department of Electrical Engineering and Computer Science,
York University, Toronto, Canada
E-mail: vlajic@cse.yorku.ca; xueshi@my.yorku.ca;
roumani@cse.yorku.ca; madani@cse.yorku.ca

Received 19 November 2017; Accepted 22 November 2017;
Publication 5 December 2017

Abstract

To date, much of the development in Web-related technologies has been driven by the users' quest for ever faster and more intuitive WWW. One of the most recent trends in this development is built around the idea that a user's WWW experience can further be improved by predicting and/or preloading Web resources that are likely sought by the user, ahead of time. *Resource hints* is a set of features introduced in HTML5 and intended to support the idea of predictive preloading in the WWW. Inspite of the fact that *resource hints* were originally intended to enhance the online user experience, their introduction has unfortunately created a vulnerability that can be exploited to attack the user's privacy, security and reputation, or to turn the user's computer into a bot that can compromise the integrity of business analytics.

In this article we outline six different scenarios (i.e., attacks) in which the *resource hints* could end up turning the browser into a dangerous tool that acts without the knowledge of and/or against its very own user. What makes these attacks particularly concerning is the fact that they are extremely easy to execute, and they do not require that any form of client-side malware be implanted on the user machine. While one of the attacks is (just) a new form of the well-known *cross-site request forgery* attacks, the other attacks have not been addressed much or at all in the research literature. Through this work, we ultimate hope to make the wider Internet community critically rethink the way the *resource hints* are implemented and used in today's WWW.

Keywords: Resource hints, Unsolicited Web requests, User privacy, User reputation, Browser forensics, Web attacks, HTML5, Chrome.

1 Introduction

With the ever-growing importance and prevalence of WWW-based services and applications, we are becoming increasing reliant on the use and performance of Web browsers – software applications that allow users to access, traverse and retrieve the WWW resources. And, while in the past Web browsers were almost exclusively built for and used on desktop and laptop computers, nowadays any device capable of connecting to the Internet (e.g., mobile phones, smart watches [1], wearable tracking devices) are likely to host one or multiple Web browsers. In fact, the modern-day dilemma is not so much whether a Web browser should be available on an Internet-enabled device (regardless of it size and capability), but what can be done to make the performance of that browser faster and more user friendly.

Users' quest for ever faster and more intuitive WWW has been the driving force behind the evolution of Web-browser technology as well as numerous Web-related protocols. (Namely, developers and designers of Web-related technology are very well aware of the fact that faster Web-sites lead to better user engagement, ultimately resulting in higher retention and conversion rates [2].) One of the most recent stages in this evolution is driven by the idea that a user's WWW experience can further be improved by predicting and/or preloading Web resources most likely sought by that particular user.

One specific mechanism that was recently introduced in order to make the idea of 'predictive preloading' possible is the so-called *resource hints feature* in HTML5. In particular, *resource hints* is a term that covers four different types of resource (pre)loading: preconnect, dns-prefetch, prefetch and prerender – all four being implemented as a relation (rel) type/attribute of HTML5's Link Element `<link>` [3]. When found in a Web-page (i.e., HTML5 document) *resource hints* are intended to instruct the browser to get hold of resources that are related to or are part of the most likely next-page navigation, ahead of time. Thus, if/when the user actually decides to request the given page, the respective resources will be simply pulled out from the 'background', giving an illusion of instantaneous (near zero-delay) retrieval.

An average Web user is likely to consider the *resource hints* features useful, as they undoubtedly have the potential to facilitate faster browsing experience. As a result, in many browser types, including Google Chrome[1], *resource hints* are enabled 'by default'. This – combined with the fact that users generally tend to keep the default settings of their applications unchanged [5] – further implies that the execution of *resource hints* is likely to be enabled in a significant number, if not the majority, of browser instances currently used in the Internet.

While we do not intend to question the practical usefulness of *resource hints* from the performance/speed point of view, the work presented in this paper seeks to address the potential negative implications of their use. Namely, the *resource hints* are generally designed to be executed without the user's direct involvement (i.e., knowledge or approval) and in an obscure 'behind the scenes' manner. And even though this un-intrusiveness has its obvious advantages when it comes to speed and convenience, it can also be easily misused – both intentionally and unintentionally – by turning a browser into a dangerous tool that acts without the knowledge of and/or against its very own user. The goal of our work is to bring awareness to these possibilities, and to make the wider Internet community rethink the way *resource hints* are implemented and used in today's WWW.

The reminder of this article is organized as follows. First, in Section 2, we discuss the significance and implications of using IP addresses as a means of identifying and tracking WWW users – a common Internet practice that is a precursor to many *resource hints* related problems discussed in this article. Then, in Section 3, we provide a detailed overview of the four major *resource hints* features/tags, while in Section 4 we present some of our experimental results concerning the execution of these tags in Google Chrome. Subsequently, in Section 5, we describe six different scenarios in which *resource hints* have the potential to negatively impact the user's privacy, security, reputation, and/or business profitability. Finally, in Section 6, we close the article with conclusions and recommendations for future research.

---

[1]According to [4], for over 70% of WWW users Google Chrome is the browser of choice. Hence, most of our discussion will revolve around this particular browser. As for the other browsers types, Firefox is used by 16%, Internet Explorer by 5%, and Safari by 3% of Internet users.

2 Background and Motivation

2.1 Relationship Between a User and His Computer/Browser

As today's world grows ever more reliant on the WWW, the boundaries between humans and their respective Internet-enabled devices and browsers are becoming increasingly blurred. Namely, in many disciplines it has become a common practice to assume that a user's device and browser are nothing but a mere extension of the user, and their only mission is to carry out the tasks explicitly requested by the user. Consequently – in all but cases of a verifiable device/browser infection by a computer malware – the user may be considered fully accountable for actions or requests executed by their device/browser.

The concepts of *user tracking* and *Web-related forensics* are perhaps the best illustration of how tight the 'coupling' between users as persons and their device/browser is. For example:

- In *user tracking*, the IP address and cookies[2] associated with a user's device (i.e., browser) are used to identify that particular user in the 'on-line world'. Subsequently, all observed Web requests that happen to carry those particular IP address and/or cookies are assumed to be generated with the full knowledge and intent of the given user and, as such, are used to track the user's online behavior as well as gauge their interest in different product and services [6]. User tracking mechanisms put relatively little (if any) effort in distinguishing between genuine user requests and those that were automatically generated by the user's browser.
- The goal of *Web-related forensics* is to gather information about which Web sites and files a user has accessed while browsing the WWW, in order to prove or disprove a claim of misconduct. The places where forensics-related artifacts are typically collected include one or all of the following: a) the browser history and cache on the user's device (if accessible), b) the log files of the edge gateway that connects the user to the Internet, c) the log files of the Web server(s) hosting the disputed files. If any evidence of the disputed files being accessed through the

[2]IP address is a unique identifier assigned to every computer connected to the Internet. Cookies are small data files that a Web server stores on a user computer to keep track of that user's browsing. Between the two, the use of cookies is a preferred and more accurate mechanism of user tracking in the WWW. However, in cases when cookies are disabled on the client/user side, or are not deployed by the Web-site (according to [11], 50% of Web-sites currently do NOT use cookies), IP addresses are used as an alternative means of user tracking.

user's device/browser (while in the user's possession) is found in either a), b) or c), the user himself could be held responsible – even without an explicit proof that the user, not the browser, was the one who actually initiated those requests.

The study presented in this article is motivated by the fact that the *resource hints* features outlined in the preceding section, when combined with our tendency to assume that devices and browsers are nothing but innocuous and trustworthy 'extensions' of their owners/users, can lead to a number of potential misuses. To lay a foundation for further discussion of this issue, we proceed by providing an outline of a typical WWW client-server architecture and its most significant elements and interactions as pertaining to our study.

2.2 Typical WWW Client-Server Architecture

The below figure outlines the most significant elements of a typical WWW client-server architecture, and those include:

a. The *client*, which in the case of the WWW is a Web browser running on the user' device. The device could be either 'fixed' (e.g., a desktop computer) or 'mobile' (e.g., a laptop, tablet or smartphone), and is uniquely identified either with a static IP address (common scenario in fixed enterprise networks) or a dynamic IP address (common scenario in cellular and public WiFi networks).
b. The *edge network*, which provides physical connectivity between the user device and the rest of the Internet. This could be either an enterprise

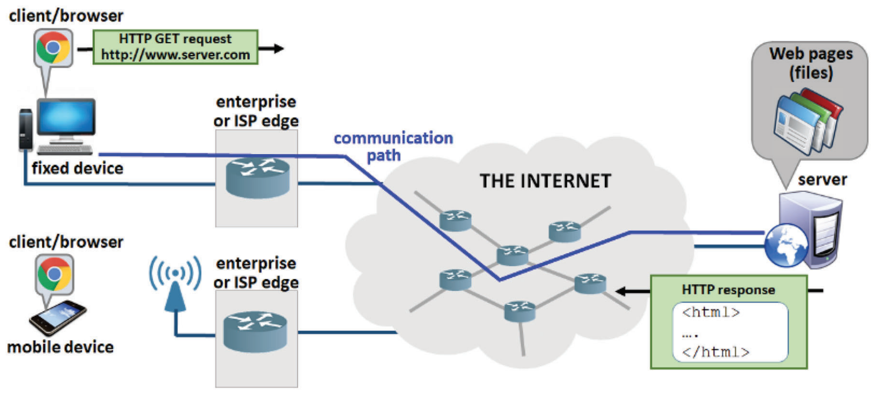


Figure 1 Typical WWW client-server architecture.

edge network (e.g., when the device is used at work), or an ISP edge network (e.g., when the device is used at home). In either case, the edge network typically contains one or multitude of specialized devices which engage in monitoring and/or logging of the passing traffic (e.g., gateway routers, firewalls, proxies, . . .).

c. The *Internet core*, which is responsible for routing packets, including those that carry client-server HTTP requests and responses, from their source to the intended destination.
d. The *server*, which in the case of the WWW is a machine capable of hosting and sharing Web-pages (i.e., files) over the Internet, and typically performs continuous and detailed logging of all incoming traffic.

Now, whenever a particular client requests a Web-page from a particular server (by means of a GET HTTP request), various types of 'artifacts' related to this event get recorded at various points along the communication path between the two entities. For example:

i. On the client side, the URL of the requested page gets recorded in the *browser history*, while the resources that the requested page is composed of get stored in the *browser cache* (once they actually arrive from the server). As earlier indicated, browser history and cache are of great significance from the perspective of Web forensics, since they can help prove that a particular Web request has taken place. Nevertheless, the main challenge of relying on browser history and cache as forensics evidence is that they are owned by and directly accessible to the user, and as such could be easily modified or deleted (intentionally or unintentionally), or simply rendered unavailable if the user decides to deny access (in which case a search warrant is required to be able to access these resources).
ii. The given HTTP request is likely recorded, together with the traffic of other users, in the logs of the specialized devices in the edge network (gateway, firewall or proxy). It should be noted, however, that edge networks are not always mandated to record these logs, hence from the forensics point they may have limited practical relevance.
iii. The intermediate routers in the Internet core could also keep a record of the given HTTP request in their own traffic logs. However, due to the high volume of passing/recorded traffic, these logs are generally kept for a very short interval of time. Consequently, their practical use as forensics evidence is rather limited, similar to ii.

iv. The server logs is the final place where the given HTTP request gets recorded[3]. In general, server logs have particularly important significance from the forensics point of view, for two main reasons. Firstly, most organization tend to retain their Web server logs over long periods of time. Secondly, in most organizations Web server logs are well protected and could only be altered by the site administrator. Hence, when a record of a Web request arriving from a particular client/host (i.e., IP address) is found in these logs, it is impossible to deny the authenticity of the given event – unless one can prove that the logs were altered (e.g.) by a malicious site administrator or some form of malware implanted on the server system.

With the above facts in mind, we further focus on the following fundamental question: for an HTTP request generated by the client/browser, is there a way of determining whether the given request was generated a result of an intentional action by the user, or perhaps it was generated without the user's knowledge and approval (e.g., due to the execution of a *resource hint* tag/command found in a rendered Web-page)? Put another way, we are set to examine whether the artifacts collected along the given communication path, and specifically on the client and server end, provide enough information to tell these two different types of requests apart.

3 Resource Hints Execution in Chrome

In this section we provide a more detailed look at the four different types of *resource hints* mechanisms that can prompt a browser to perform various forms of resource preloading, without the user's explicit knowledge and intervention.

3.1 Resource Hints in HTML5

Hypertext Markup Language (HTML) is a well-known and widely used interpreted tagged markup language that enables creation of Web-pages (hypertext documents). In the most recent version of the protocol (HTML5), which was first published in 2014, a special new set of features have been introduced in order to support the idea of 'instant' (zero-delay) Web-page load.

[3]Web server logs collect a wealth of data, including which specific pages/resources were requested, at what time, and from which IP address. This data is then used to deduce information about the overall number of visitors to the given site as well as to analyze their browsing behavior.

Namely, as pointed in [8] and [9], a browser that starts downloading a Web-page only after the page has been explicitly requested by the user will inevitably result in substandard browsing experience that is riddled with various types of network delays. (These delays include: DNS lookup delay, TCP handshake delay, SSL negotiation delay, delay to obtain base HTML page ... [2, 8].) The only way to spare the user from experiencing the browsing/network delays is by trying to anticipate their requests ahead of time, and then preload the most critical resources associated with those requests even before the actual 'click on the link' action occurs. That way, the resources will be readily available when the user actually requests them, giving an illusion of an instantaneous (zero-delay) download.

Now, the idea of 'instant' browsing is not entirely new. This concept was originally supported through the implementation of *Web-cache* – a memory location where the resources of previously visited Web-pages are stored, allowing that these resources be instantaneously retrieved whenever the user decides to subsequently revisit them. Unfortunately, as such, Web-cache is of no use when it comes to the retrieval of new pages that have not been previously requested. To enable zero-delay browsing of pages that are to be visited for the first time, or pages that have been purged or expired from the cache, HTML5 has come up with a set of features commonly referred to as *resource hints*.

According to [2], there are four different types of *resource hints* provisions in HTML5.

a) ***dns-prefetch*** is a resource hint option that can be used to suggest a browser to perform a DNS prefetch (i.e., IP lookup) for a particular hostname. The following is a situation where this feature might be useful in practice. Imagine the user is currently visiting `page_A.html` hosted on `server_1.com,` and there is a high likelihood that the Web-page the user is going to visit next is `page_B.html` located on another server (`server_2.com`) – as illustrated in Figure 2. To expedite the loading of `page_B.html` (if and when the user requests it), the below tag could be placed in the <`head`> section of `page_A.html`:

```
<link rel="dns-prefetch" href="//server_2.com">
```

That way, the browser would start performing the DNS lookup for `sever_2.com` right away (while the user is still viewing `page_A.html`), making sure that the IP address of `server_2` is obtained even before the user actually clicks on `http://server_2.com/page_B.html`.

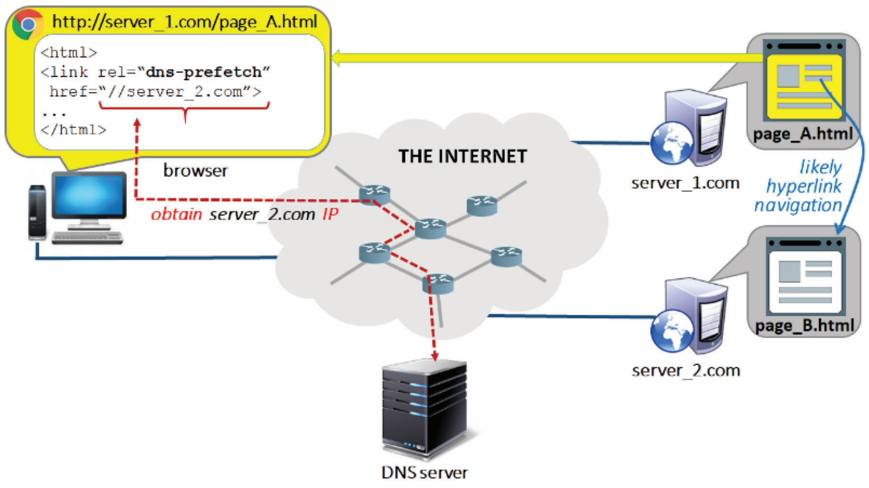


Figure 2 Linked pages hosted by different servers.

b) ***preconnect*** is a resource hint option that can be used to initiate an early connection with a Web server, which includes the DNS lookup, TCP handshake, as well as optional TLS negotiation. As such, preconnect clearly goes step further in minimizing/masking networking delays relative to *dns-prefetch*.
In the example of Figure 2, the following tag placed in the <head> section of `page_A.html` would prompt the user's browser to establish an early (pre)connection with `server_2.com`.

```
<link rel=''preconnect'' href=''//server_2.com''>
```

Also, in the given example, the decision whether to use *preconnect* or just *dns-prefetch* for `server_2.com` should be closely tied to the actual probability that the user navigates to `page_B.html` from `page_A.html`. Clearly, the higher this probability, the more reasonable it would be to place the *preconnect resource hints* option referring to `server_2.com` in the HTML head/body of `page_A.html`.

c) ***prefetch*** is a resource hint option that further builds on the functionality of a) and b). Namely, in addition to performing the DNS resolution and establishing a connection with a particular server, *prefetch* also allows that some actual resources (e.g., the base HTML file of a Webpage, images, JavaScript-s, CSS-s, etc.) be downloaded from this server ahead of time and stored in the browser cache. For example, in the scenario of Figure 2, the following tag placed in the <head> section of page_A.html would prompt the user's browser to download and cache the base HTML file of page_B.html – the key Web resource (and the first one to be retrieved) during the rendering of this page.

```
<link rel="prefetch" href="//server_2.com/
                page_B.html">
```

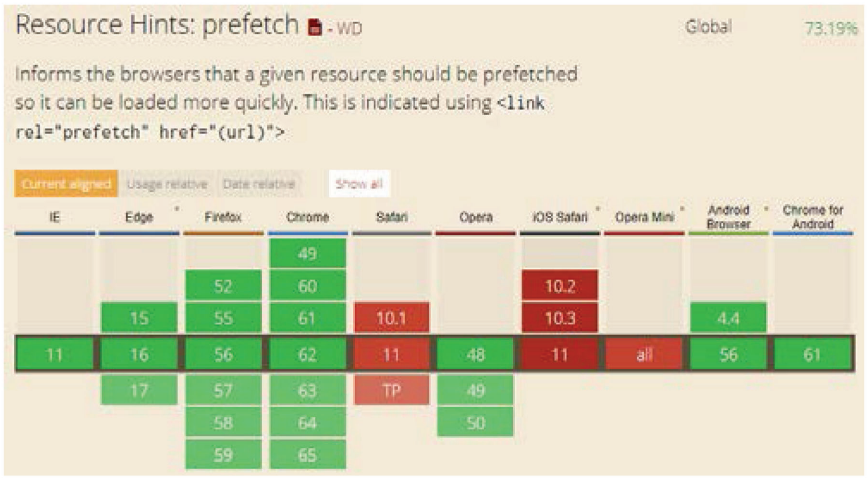
Clearly, by allowing that whole parts of a page be obtained by the browser - even before the page gets actually requested – *prefetch* enables even further reduction in networking/browsing delays. However, given the communication and storage overhead associated with *prefetch*, it is recommended that this *resource hints* option be used only in cases when the probability that the user actually navigates to a specific page is greater than in the case of a) or b).

d) ***prerender*** is the most encompassing *resource hints* option – it allows not only that the base HTML file and all other components of a page get preloaded ahead of time, but also that the page itself gets fully laid out, its respective CSS-s applied and JavaScript-s executed. Put another way, it is as if the page is open in a hidden tab, and the moment the user navigates to the page's URL, the hidden tab is immediately swapped into view [2]. As such, *prerender* is the only *resource hints* option that can truly cut the browsing delay down to zero, giving an illusion of truly instantaneous browsing.
In the scenario of Figure 2, the following tag placed in the <head> section of page_A.html would prompt the user's browser to prerender (i.e., preload and preassemble) the entire page_B.html.

```
<link rel="prerender" href="//server_2.com/
                page_B.html">
```

Now, it should be pretty clear that out of all four *resource hints* options, the use of *prerender* is associated with the most significant communication, storage and processing overhead. Consequently, the use of this option should be reserved only for cases when the navigation to a specific page is highly probable if not absolutely certain.

The above suggestions are merely recommendations pertaining to the *resource hints* options in HTML5 as outlined by World Wide Web Consortium (W3C) [3]. Unfortunately, the actual implementation of the *resource hints* options in real-world browsers has neither been standardized nor mandated. As a result, there has been a significant variation in the number and actual implementation of different *resource hints* options by different browser types. (For more see [2, 8, 9]). Given that for the majority of Internet users


Resource Hints: prefetch - WD

Global 73.19%

Informs the browsers that a given resource should be prefetched so it can be loaded more quickly. This is indicated using `<link rel="prefetch" href="(url)">`

IE	Edge	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Chrome for Android
			49						
		52	60			10.2			
	15	55	61	10.1		10.3		4.4	
11	16	56	62	11	48	11	all	56	61
	17	57	63	TP	49				
		58	64		50				
		59	65						

Figure 3 Support for HTML5 prefetch option by different browser types as of November 2017 [10].

Google Chrome happens to be the browser of choice [4], our discussion largely focuses on this particular browser type. Specifically, in the proceeding section, we present some of our experimental results pertaining to the behavior of Google Chrome when encountering different *resource hints* options in the browsed pages. The experimentations were originally performed on Google Chrome v.52, which supported all four types of resource hints options discussed in this paper. For the most recent support of different *resource hints* options by various browser types the reader is referred to the site https://caniuse.com/. (E.g., according to his site, the current support for *prefetch resource hints* option by various browser types is as show in Figure 3.)

4 Experimental Set-Up and Results

In order to gain a better understanding of how Google Chrome deals with different HTML5 *resource hints* options when encountering them in a browsed page, we built an experimental client-servers framework as outlined in Figure 4. The 'client' in this framework was the latest version of Google Chrome (Chrome v.52) running on a laptop PC. The 'server' was set up on the Amazon Cloud (http://ec2-54-186-72-100.us-west-2.compute.amazonaws.com) and was hosting a repository of test Web-pages.

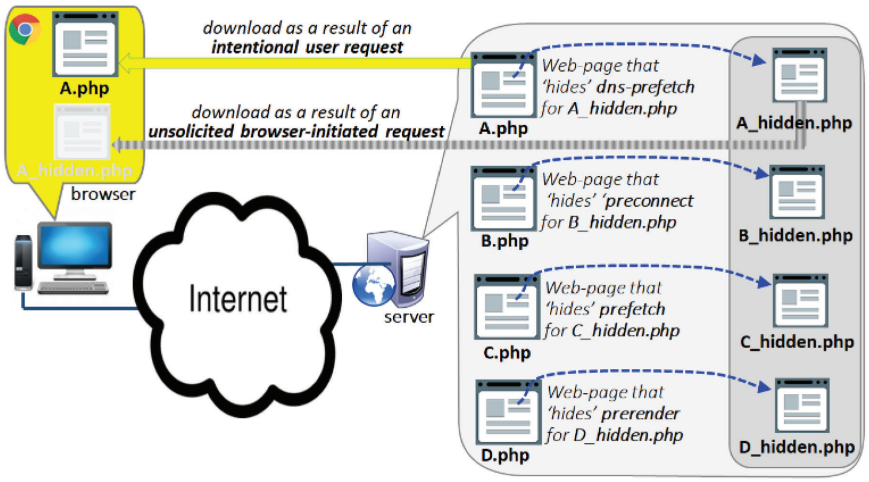

download as a result of an intentional user request
A.php
download as a result of an unsolicited browser-initiated request
A_hidden.php
browser
Internet
server
Web-page that 'hides' dns-prefetch for A_hidden.php
A.php
A_hidden.php
Web-page that 'hides' 'preconnect for B_hidden.php
B.php
B_hidden.php
Web-page that 'hides' prefetch for C_hidden.php
C.php
C_hidden.php
Web-page that 'hides' prerender for D_hidden.php
D.php
D_hidden.php

Figure 4 Experimental framework for evaluation of Chrome behavior when browsing pages with *resource hints* options.

We chose to code the pages of this repository in php instead of plain html in order to be able to prevent their caching on the client side, as well as to be able to implement and examine the general impact of cookies on pages referenced in *resource hints* tags.

The test pages of our framework were grouped into two sets. The pages of the first set were are designed to be directly visible/accessible to the user, and each of them hid one particular *resource hints* option in its respective php/html code (A.php, B.php, C.php, D.php). The other set was comprised of pages referenced in the *resource hints* tags of the first set, and this set was not intended to be directly visible/accessible to the user (A_hidden.php, B_hidden.php, C_hidden.php, D_hidden.php). With this structure, if the pages of the second set – or their respective resources – ever got requested, that was a clear indication that the browser itself (not the user) had triggered those requests while processing the *resource hints* tags in the pages of the first set. (Note that, because of the way *resource hints* are intended to work as well as the way our framework was designed, requests for the pages of the second set not only got generated without the user's direct knowledge and involvement, but the user also never got to know when those resources actually arrived at their browser.)

In our experimentation, we first performed intentional requesting/retrieval of pages A.php to D.php (Figure 4) through the client – Chrome v.52 browser operating on a machine in our departmental network. Subsequently, we examined the collected artifacts pertaining to these requests both on the client and on the server side. The most significant of our observations are presented in Table 1, and can be summarized as follows:

1. The requesting of pages A.php and B.php (i.e., pages that contained DNS-prefetch and preconnect resource hint options in their respective HTML5 code) did not leave any permanent artifacts related to A_hidden.php and B_hidden.php – either on the client or on the server side. Such a result could have been expected, as these two particular *resource hints* options do not 'trigger' application-level preloading of resources referenced in their <link> tags. Instead, DNS-prefetch and preconnect facilitate only 'lower level' (DNS and TCP) domain-name resolution and connection set-up.
2. On the other hand, the requesting of pages C.php and D.php (i.e., pages that contained prefetch and prerender resource hint options in their respective HTML5 code), did leave a number of artifact related to C_hidden.php and D_hidden.php on the client and on the server side. In particular:

   2a. On the client side, both (prefetched) C_hidden.php and (prerendered) D_hidden.php were not only retrieved but also ended up being stored in the browser cache. Furthermore, a cookie associated with each of these pages was created and placed in the browser's cookie cache. Finally, a DNS record pertaining to both pages was stored in the browser's DNS cache. All in all, the way the browser went about retreiving C_hidden.php and D_hidden.php was not much different from the way A.php to D.php were retrieved – even though the latter group of pages was explicitly requested by the user, while the user had no way of knowing that the former group of pages was ever requested and/or retrieved. (The only noticeable difference between the two groups is that the retrieval of A.php to D.php was recorded in the browser history, which was not the case for C_hidden.php and D_hidden.php).

   2b. On the server side, HTTP GET requests for both C_hidden.php and D_hidden.php appeared in the server logs. More importantly, these two requests looked identical to the requests for pages A.php to D.php, in terms of their (HTTP) content. In other words, based on what was recorded in the sever logs, it was impossible to distinguish between the user's intentional requests – for A.php to D.php – and the requests that were issued automatically by the browser without the user's knowledge and approval (for C_hidden.php and D_hidden.php).

Following the experimentation with the framework outlined in Figure 4, we conducted another experimental study, where the Web objects referenced in A.php to D.php were pages hosted on another server. The observations concerning the recorded artifacts in this experiment were identical to the ones presented hereinabove (i.e., in Table 1).

Our experimentation also looked at the use of multiple prerender and prefetch tags inside the same Web-page. Our observation is that in case of multiple prerender tags in a Web-page, only one of these tags is executed at the time, while the respective (prerendered) page gets placed in the browser's RAM[4]. (The likely reason why Chrome and other browser do not allow simultaneous prerendering of multiple pages is to prevent potential overloading of the browser's RAM, which would degrade the overall browser performance.) On the other hand, there seem to be no limit on the number of prefetch tags that get executed in a Web-page. Once retrieved, each of the prefetched resources ends up being stored in the browser's cache.

5 Resource Hints Implications on User Privacy, Reputation and Business Performance

In this section, we present six different scenarios in which *resource hints* are used as the main attack vector against a targeted Web user. The names of these attacks and their respective targets (i.e., user 'assets' that they are ultimately impacting) are summarized in Table 2. The key characteristics of all six attacks is the fact that they are extremely easy to execute, as they do not require that any form of client-side malware be implanted on the victim machine. The only precondition for their successful execution is to be able to lure the targeted user (victim) into visiting a specially crafted decoy Web-page. As indicated in [12], there are numerous well-known and very effective techniques which the attacker could deploy to lure a victim into visiting a decoy Web-page – ranging from various site-promotion techniques (e.g., in blogs and social media sites) to the use of targeted phishing emails.

[4]Our research has shown that, theoretically, it would be possible to have multiple prerender tags, from one single Web-page, executed. Though, this would require that each of the prerendered Web-pages comes with the auto-refresh functionality, and a relatively short auto-refresh interval.

Table 1 Artifacts collected on client and server side when resource hint options found in a Web-page

	Browser-side Artifacts				Server-side Artifacts
Resource Hints Option	**Effect on Chrome History**	**Effect on Chrome Cache**	**Effect on Chrome DNS Cache**	**Effect on Cookies**	**Server Side Log**
DNS-prefetch	no effect	no effect	no effect	no effect	no GET request received at the server
preconnect	no effect	no effect	no effect	no effect	no GET request received at the server
prefetch	no effect	prefetched page/ resource showed up in cache	showed up as a sub-resource of the calling web-site	cookies created	a GET request for prefetched resource/page received at the server (unless page/resource found in cache)
prerender	no effect	prerendered page showed up in cache	showed up as a standalone record (same as a user initiated visit)	cookies created	a GET request for prerendered page received at the server (unless page found in cache)

Table 2 Impact of different attacks performed using prefetch/prerender on user reputation, security, privacy and business profitability

Attack Performed Using Prefetch/Prerender	**Attack's Primary Target**
Framing Attack	user reputation
Targeted DoS	user security (i.e., availability of a site which user wants to access)
CSRF	user security (i.e., integrity of requests issued by user's browser]
Data-Analytics Pollution	business profitability
ETag Tracking	user privacy
Cookie Stuffing	business profitability

Scenario 1: ***Framing Attack.***

The term 'framing attack'[5] was introduced in [12], and it refers to a scenario in which false (digital) evidence is planted on the victim's computer, without requiring physical or remote access to their machine and without involving any form of client-side malware. The sole goal of this attack is to incriminate or discredit the victim in the context of their social, workplace, business or political life.

To provide an illustration of how a framing attack could be accomplished by means of HTML5 resource hints, imagine a situation where Trudy is a disgruntled employee working at a research company. Trudy holds a special grudge towards Bob – a manager that she directly reports to. As a form of revenge against Bob, Trudy decides to format one of her upcoming reports as an HTML5 document. Inside this document, she 'hides' several dozens (or more) of resource hint tags – each prefetching[6] a highly inappropriate (e.g., child pornography or terrorism-related) Web-page. By means of JavaScript, Trudy also ensures that the execution of each prefetch tag occurs at a different point in time, thus mimicking the way a human user would go about retrieving a sequence of such Web-pages.

The 'reporting' day has come, and Bob opens the document that Trudy has referred him to. The (visible) content of the document seems very relevant, and Bob spends quite some time viewing the document in his browser. Clearly, while Bob is reading the visible content, his browser (in the background)

[5]Please note that in a few research works, such as [13], the term 'framing attack' was used to refer to a version of 'clickjacking attack'. However, the type of attack discussed in [12] as well as this paper has a very different context and outcome.

[6]In this case, prefetching of a Web-page/URL would mean that its respective top-level resource (most often a base HTML file) is requested and retrieved by the browser.

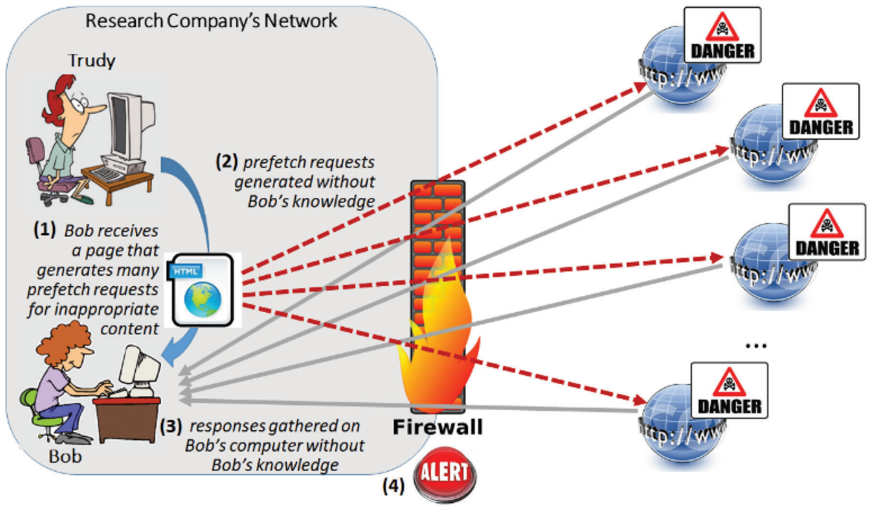

Research Company's Network
Trudy
(1) Bob receives a page that generates many prefetch requests for inappropriate content
(2) prefetch requests generated without Bob's knowledge
(3) responses gathered on Bob's computer without Bob's knowledge
Bob
Firewall
(4) ALERT
DANGER
DANGER
DANGER
DANGER
...

Figure 5 Framing attack.

retrieves/prefetches the inappropriate pages (i.e., hidden resource hints) one-by-one, as illustrated in Figure 5. Bob, obviously, remains completely unaware that these downloads are taking place.

At the same or later point in time, the company's Web-content firewall generates an alert pointing to Bob's machine (i.e., his machine's IP) as the source of requests for inappropriate content. The company's authentication system verifies that the requests were generated while Bob was logged in and using the machine. From the forensics point of view, these pieces of evidence are often enough to 'point fingers' to Bob, and hold him accountable.

Now, depending on how severe the company's policy pertaining to inappropriate use of resources is, Bob could experience a whole range of possible outcomes – from receiving a simple warning to facing serious disciplinary actions and possibly termination. The only way Bob could avoid these repercussions and clear his name is by providing aggregate browsing-related artifacts from his computer (spanning over a period of time before and after the actual incident) to relevant authorities. While an adequate expert analysis of these artifacts could potentially succeed in putting 'all the pieces of the puzzle together', and identify the actual cause of the inappropriate requests, the implications on Bob's privacy could be significant – especially if Bob had used his own personal device (as in the case of BYOD) to view Trudy's page. In addition, by the very virtue of being linked with actions that are considered ethically and/or legally unacceptable, Bob is likely to experience unnecessary scrutiny with all the accompanying negative implications on his professional and personal life. (The best illustration of this are the cases of Julie Amero [14] and Michael Fiola [15]. These two people, in two different instances, were wrongly charged with downloading of child pornography. In both cases it was ultimately proven that the downloading of the inappropriate material was caused by malicious software and their respective names were cleared. Still, as stated by both people, the conducted trials have had lasting negative effect not only on their lives but also on the lives of their families.)

According to our knowledge, [12] is the only other research work that, in addition to introducing, has also studied the actual mechanisms of executing a framing attack. The idea specifically suggested in [12] is similar to the one outlined in Figure 4, except that the obscure/decoy requests are not generated via *resource hints* tags (prefetch of prerender) but instead by means of two better known and more widely used HTML tags – <iframe> and <img>. However, as indicated in [12], for these framing attacks to actually be successful, the attacker needs to take extra measures towards 'obscuring' the objects/Web-pages referenced in the decoy <iframe> and <img> tags (i.e., make sure that they go unnoticed by the victim once they are retrieved/rendered by the browser). Possible approaches to ensuring that the decoy <iframe> and <img> objects remain 'invisible' include: 1) minimizing their size to 0x0 pixels, 2) hide them under another overlaid iframe/image, 3) make them invisible through CSS (e.g., by setting their *display* attribute to *none*). It should be noted, though, that the same object obfuscation techniques are required and deployed by many other types of browser-based attacks, such as *clickjacking* and *cross-site request forgery*. These specific attacks have been around for more than a decade, and as a result, the majority of today's Web-vulnerability scanning tools (e.g., Burp [16]) are programmed to spot and block Web-pages suspected of object obfuscation. Consequently, a framing attack based on the use of <iframe> and <img> decoy tags (as proposed in [12]) could potentially be detected and prevented by these tools. On the other hand, a framing attack based on the use of HTML5 *resource hints* (as suggested in this work) would virtually go unnoticed by these same scanning tools. Namely, while a 'malicious' <iframe> and <img> could be detected (i.e., labeled as such) by looking for signs of obfuscation, there are no clear mechanisms or indicators which could help in distinguishing between a benign and a malicious <prerender> or <prefetch> tag. (Recall, the very purpose of these tags is to facilitate 'invisible' preloading of Web object. Furthermore, the preloaded objects are supposed to remain hidden until explicitly requested by the user.)

Scenario 2: ***Targeted DoS Attack*.**
Now, imagine that in the previously depicted story, instead of tarnishing Bob's reputation within their organization, Trudy decides to execute her revenge by affecting the 'outside' reputation of Bob's machine (i.e., IP address), with the ultimate goal of having Bob's IP address blacklisted and denied service.

In particular, imagine that Trudy knows of a Web-site that Bob likes to frequently visit, such as the Web-site of his bank or a specific news-agency Web-site. In that case, Trudy could hide a very large number of prefetch references targeting this particular Web-site (its various pages/resources) inside her 'malicious' Web-page, as illustrated in Figure 6. As many other similar organizations, Bob's bank is likely to perform comprehensive intrusion detection monitoring of the incoming Web traffic, in order to spot and blacklist all misbehaving users. Given that the avalanche of requests coming from Bob's machine is very reminiscent of a denial of service (DoS) attack, it is quite possible that Bob's IP would end up on the bank's blacklist, at least for a period of time. Consequently, during that period of time, even Bob's legitimate requests would be rejected (since they originate from the same IP), and Bob would be cut off from the online services of his bank. We refer to this attack as 'targeted DoS', as it ensures that one specific user is denied (i.e., not able to access) service of one particular Web-site.

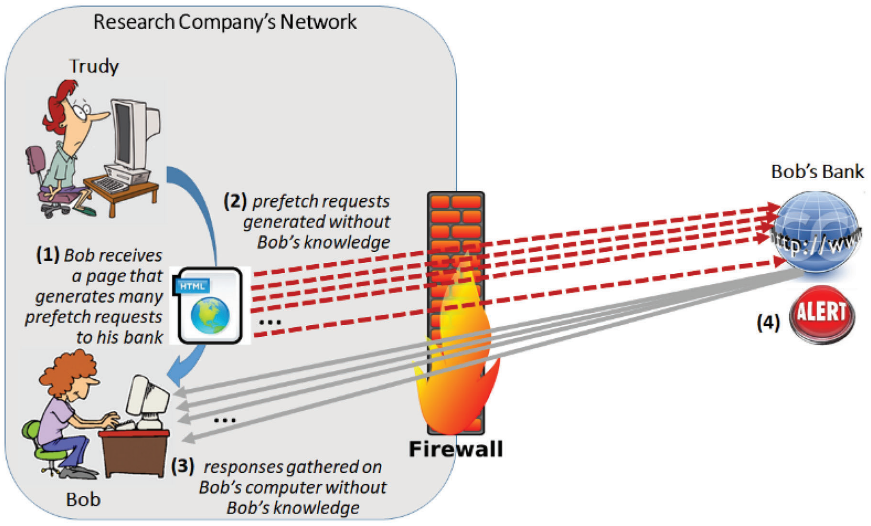

Research Company's Network
Trudy
(1) Bob receives a page that generates many prefetch requests to his bank
(2) prefetch requests generated without Bob's knowledge
Bob's Bank
(4)
ALERT
Firewall
Bob
(3) responses gathered on Bob's computer without Bob's knowledge

Figure 6 Targeted DoS attack.

Scenario 3: ***Cross-Site Request Forgery Attack*.**
Cross-site request forgery (CSRF) is a well-known type of attack that occurs when a malicious Web-site causes a user's browser to perform an unwanted action on a trusted site for which the user is currently authenticated [17]. More specifically, CSRF attack requires that the user first gets successfully authenticated to a legitimate Web-site (e.g., by means of cookies), as illustrated in Figure 7. If following that action the user visits a malicious (Trudy's) Web-page,(as shown in Figure 8, the malicious page can force the user's browser to make unsolicited request towards the site for which the user is currently authenticated. By default, the browser will attach the legitimate previously set

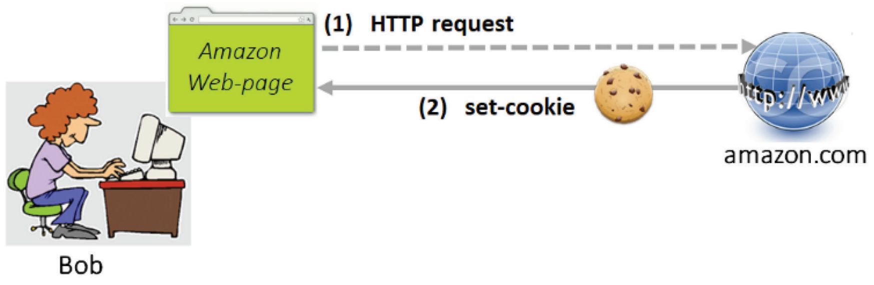
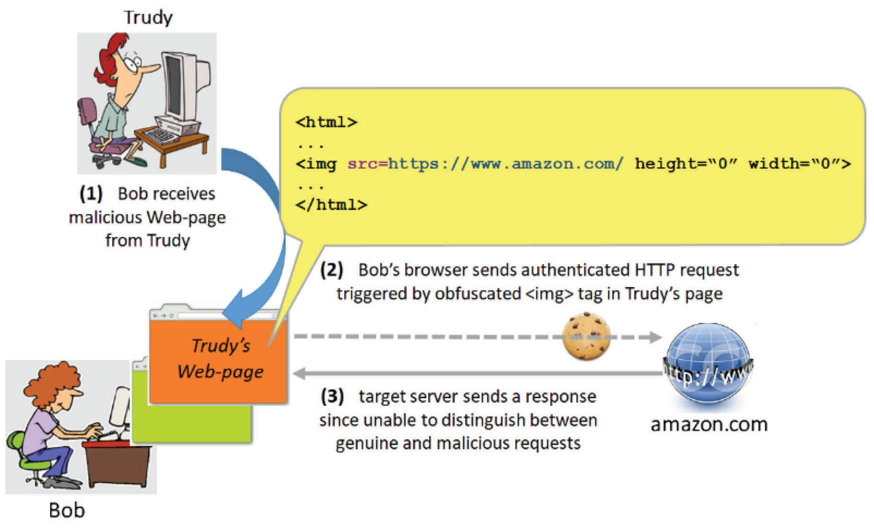


Figure 7 User authentication by means of cookies.


Trudy
(1) Bob receives malicious Web-page from Trudy
(2) Bob's browser sends authenticated HTTP request triggered by obfuscated <img> tag in Trudy's page
Trudy's Web-page
http://www
(3) target server sends a response since unable to distinguish between genuine and malicious requests
amazon.com
Bob

```
<html>
...
<img src=https://www.amazon.com/ height="0" width="0">
...
</html>
```

Figure 8 CSRF attack following user authentication.

cookie(s) to each of the unsolicited/malicious requests, which will make the server's job of distinguishing between genuine user requests and those that were triggered by the malicious Web-page hard if not impossible.

CSRF attack have been traditionally accomplished by 'hiding' the unsolicited HTTP requests of the malicious page inside <img> and <iframe> HTML tags, as in the case of the framing attack described in [12]. However, we have already explained that many of today's Web-vulnerability scanning tools are capable of detecting such 'basic' variants of CSRF attacks, simply by looking for signs of <img> or <iframe> obfuscation. Consequently, from the attacker's point of view, hiding the unsolicited CSRF HTTP requests inside <prefetch> and <prerender> tags is a viable and far more lucrative alternative, as today's Web-vulnerability scanning tools generally do not look for signs of misuse in any of the four resource hints options.

(According to our knowledge, no previous work has looked at the use of resource hints options in the context of CSRF attacks. Our group has recently conducted a study on the feasibility of CSRF attacks on amazon.ca and ebay.ca by means of resource hints, which resulted in the discovery of open vulnerabilities in both sites. The findings of this study are currently under submission to [18].)

Scenario 4: ***Data-Analytics Pollution Attack.***
(Similar to Framing and Targeted DoS, this particular type of attack has also not been previously discussed in the literature. Its main aim is to impact the performance of an on-line business, and ultimately its financial profitability, by distorting its Web-site based data analytics.)

As the premise for this attack, we imagine Trudy to be the owner of a small business, and Alice to be her direct business competitor. Both businesses have online presence which is critical for the success of their operation. Namely, not only that the two businesses advertise and sell their products through their respective Web-sites, but they also heavily rely on the Web (server-log) analytics to better understand where their customers come from and what they are looking for.

In order to 'pollute' the logs of Alice's Web server, and thus negatively impact Alice's business intelligence, Trudy has come up with the following plan: In the Web-page(s) of her own Web-site, Trudy has hidden numerous prerender and prefetch tags referencing various (strategically chosen) pages from Alice's Web-site. Thus, whenever Trudy's customers visit her Web-site, their respective browsers end up generating a slew of 'polluting' requests towards Alice's Web server – see Figure 9. Obviously, because of the way

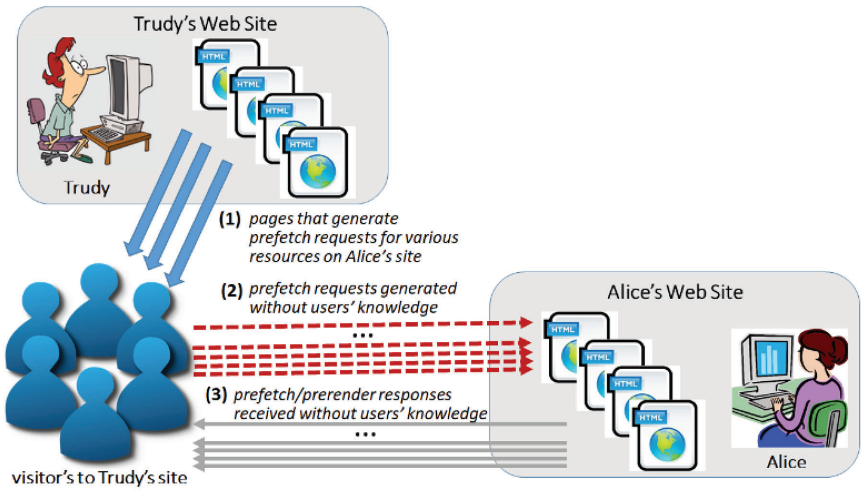

Trudy's Web Site
Trudy
(1) pages that generate prefetch requests for various resources on Alice's site
(2) prefetch requests generated without users' knowledge
(3) prefetch/prerender responses received without users' knowledge
Alice's Web Site
Alice
visitor's to Trudy's site

Figure 9 Data polluting attack.

the *resource hints* are intended to work, Trudy's customers will be completely unaware that their browsers have participated in a 'data polluting' attack. At the same time, the performance of Trudy's Web-site will remain entirely unimpacted by the attack, as the retrievals of prerender/prefetch resources from Alice's Web-site will always take a lower priority and occur only during the browser's idle times.

As for Alice's ability to detect this attack and identify all the polluting requests – the only piece of information that she possibly could rely on is the *referer* field in the incoming HTTP requests. (*Referer* field identifies the address of the Web-page from which the user/browser has accessed, or moved to, the current Web-page.) In the case of Trudy's attack, this field would be referring to the pages of her Web-site, thus indirectly revealing the true origin of the 'polluting' requests targeting Alice's site. Though, if Trudy wanted to make her attack particularly stealthy, she could implement the following meta tag in the <head> section of her Web-pages used to refer to resources in Alice's Web-site:

```
<meta name=''referrer'' content=''none''>
```

That way, HTTP requests arriving to Alice's Web-site by means of Trudy's Web-pages would not contain any referral data. Consequently, Trudy's attack would remain virtually undetectable.

<u>Scenario 5</u>: *ETag Tracking Attack*.

As explained in Section 2, two most commonly deployed techniques of user tracking in the WWW are: (1) tracking that utilizes the IP-address of the user's device, and (2) tracking that utilizes Web-cookies stored in the user's browser. According to [19], (1) falls in the category of the so-called *fingerprinting-based*, while (2) falls in the category of the so-called *storage-based* user tracking techniques. The third lesser known category of user tracking are the so-called *cache-based* techniques, with ETag tracking[7] being its best-known representative.

Cache-based is somewhat similar to *storage-based* tracking, as it also makes use of persistent client-side storage to achieve its objective. However, the main difference between the two is that *cache-based* tracking avoids storing explicit tracking data (e.g., Web cookies) in the client's memory, but instead relies on data/objects that get implicitly and automatically stored in the browser's cache memory during the standard process of Web-page retrieval. Namely, in most browser types, after a Web-page with all of its inline objects is downloaded and displayed to the user, the browser will identify objects that are marked 'cacheable' and place them in its Web cache in order to speed up their delivery next time they are requested. Now, to facilitate the process of user tracking, there is nothing preventing the server from adding one 'dumb' cacheable tracking object (e.g., a small or invisible image or a javascript) to all its hosted pages, while hiding some unique user-related ID in this object's meta-data. Subsequently, the given object together with its meta-data could be exploited to track the respective user. One of the most convenient places for hiding such user-related meta-data is the so-called Entity Tag (ETag) field of HTTP Response Header. ETag is an opaque non-mandatory identifier assigned by the host server to a specific version of an object found at a particular URL. If the given object ever changes, while keeping the same URL, a new and different ETag should be assigned to it. As such, ETag is intended to be an additional HTTP field/feature that allows Web browsers to verify whether the objects stored in their respective Web cache are up-to-date. Nevertheless, there are no specific rules about how ETag values should be set or initialized. Hence,

[7]The first practical use of ETag tracking was noted in 2011 [19]. Initially this technique created a lot of controversy, since it was seen as an easily obscurable way of implementing cross-domain user tracking that worked even for user sessions in private browsing mode. (Recall, cross-domain user tracking generally should NOT be used unless explicitly consented to by the user.) Today, however, this technique is seen as 'mainstream', and the privacy policies of many major Web-sites mention the use of ETags as a possible means of user tracking implemented by these sites.

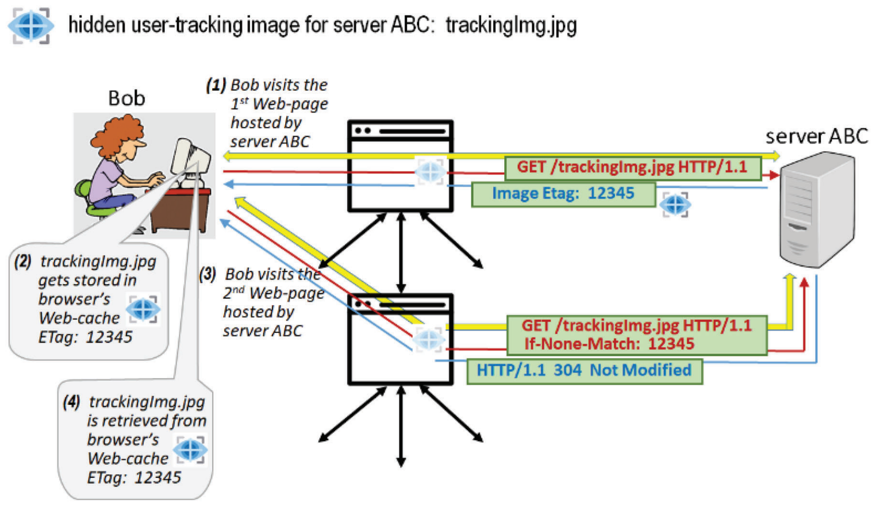

hidden user-tracking image for server ABC: trackingImg.jpg
Bob
(1) Bob visits the 1st Web-page hosted by server ABC
server ABC
GET /trackingImg.jpg HTTP/1.1
Image Etag: 12345
(2) trackingImg.jpg gets stored in browser's Web-cache ETag: 12345
(3) Bob visits the 2nd Web-page hosted by server ABC
GET /trackingImg.jpg HTTP/1.1 If-None-Match: 12345
HTTP/1.1 304 Not Modified
(4) trackingImg.jpg is retrieved from browser's Web-cache ETag: 12345

Figure 10 User tracking by means of ETag.

it is possible to send a completely different ETag value to every single visitor retrieving exactly the same object. This, ultimately, can be exploited to turn ETag into an effective means of user tracking. The actual idea of using ETag for the purpose of user tracking is illustrated in Figure 10.

In Figure 10, *server ABC* has placed a tracking image (*trackingImg.jpg*) in all its hosted pages. At the time of Bob's visit to the first (of possibly several) *server ABC*'s pages, Bob's browser ends up making HTTP GET requests for all inline objects of this page, including *trackingImg.jpg*. As a response, *server ABC* sends *trackingImg.jpg* back to Bob's browser together with an assigned ETag value that is unique to Bob (12345) and a maximum possible cache-control age, which will ensure that the tracking image gets stored and kept in the browser's Web-cache for a sufficiently long period of time. As a result, if/when Bob clicks on (i.e., requests) any other page from *server ABC*[8], the browser will attempt to revalidate the copy of *trackingImg.jpg* already stored in its cache by sending a HTTP GET request to *server ABC* with 'If-None-Match' field set to the previously received value of ETag (12345). When this re-validation request arrives at *server ABC*, the server will not only learn that the request for the respective/originating Web-page has been made by a

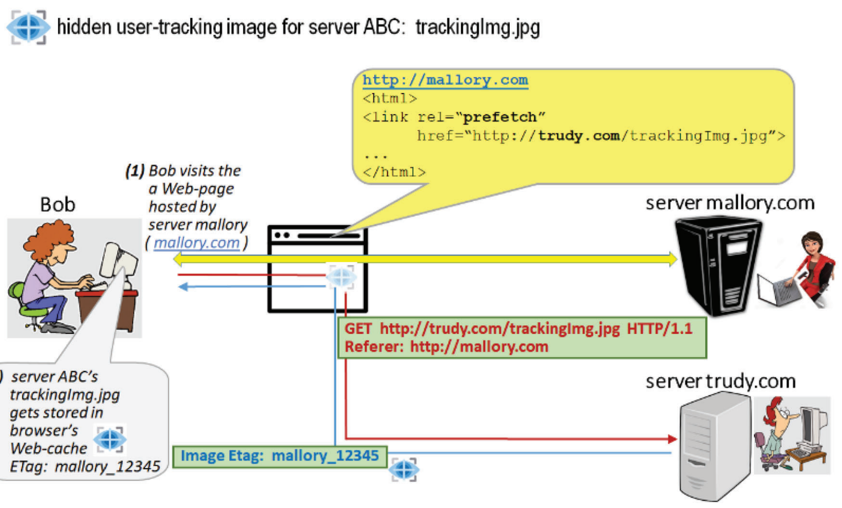
[8]Recall, all other pages also contain a copy of trackingImg.jpg. Hence, to properly render any of these pages, the browser will have to re-fetch a copy of trackingImg.jpg – either from the Web cache, if the cached copy of the image is still valid, or otherwise from the server.

returning visitor, but will also be able to reveal the visitor's actual (previously assigned) identity –12345.

To understand how HTML5 prefetch or prerender functionality can be combined with ETag caching for the purpose of covert cross-domain tracking, let us imagine the scenario outlined in Figure 11. Here, Trudy – the owner of the site *trudy.com* – is not only interested in knowing whether the visitors to the pages of her site are 'new' or 'returning', but also whether at **any** point prior to coming to *trudy.com* these visitors have also visited the Web domain *mallory.com* owned by her friend Mallory. Trudy wants to gather this information surreptitiously, without the use of cookies. To achieve her objective, Trudy has asked Mallory to add the following simple prefetch reference into all Web-pages of her site: `<link rel=''prefetch'' href=''http://trudy.com/trackingImg.jpg>.` Now, assume Bob is one such user who has come to *http://mallory.com* prior to visiting *trudy.com*. Given that *http://mallory.com* contains the embedded prefetch reference to Trudy's *trackingImg.jpg*, Bob's browser will request (obtain and cache) this image from *trudy.com* during the rendering of http://mallory.com. Moreover, since Bob's HTTP GET request for *trackingImg.jpg* also contains `'Referer: http://mallory.com'` in itself, server *trudy.com* will be able to extract and embed this information into ETag value of the returned *trackingImg.jpg* image (observe the returned and cached ETag value in Figure 11 – `mallory_12345)`. That way, when Bob eventually visits a Web-page from *trudy.com* – which will also contain *trackingImg.jpg*[9] – Bob's browser will recognize that the given image has already been retrieved and stored in its cache. Subsequently, the browser will attempt to re-validate this image by sending a GET request to *trudy.com* with `'If-None-Match'` set to `mallory_12345`. By receiving such request *trudy.com* will be easily able to deduce that the visitor to the respective/originating Web-page (i.e., Bob) is one of the visitors who has previously, at some point, also visited *mallory.com*.

It should be quite obvious that with very little additional effort the scenario from Figure 11 can be extended to allow Trudy: a) to learn/extract the exact time of Bob's visit to *mallory.com* (e.g., by putting the encoded/encrypted timestamp associated with the first arrival of prefetch request for *trackingImg.jpg* into ETag value), and b) to track the movement of her visitors across more than one other 'partner' site of interest (e.g., by placing the tracking image to all such sites, and by monitoring the `Referer` field of all revalidation requests for *trackingImg.jpg*).

[9]Recall, Trudy also wants to be able to track returning users on her own domain, hence all pages of *trudy.com* will also contain trackingImg.jpg.



```
http://mallory.com
<html>
<link rel="prefetch"
      href="http://trudy.com/trackingImg.jpg">
...
</html>
```

```
GET http://trudy.com/trackingImg.jpg HTTP/1.1
Referer: http://mallory.com
```

hidden user-tracking image for server ABC: trackingImg.jpg
Bob
(1) Bob visits the a Web-page hosted by server mallory (mallory.com)
server mallory.com
server trudy.com
(2) server ABC's trackingImg.jpg gets stored in browser's Web-cache ETag: mallory_12345
Image Etag: mallory_12345

Figure 11 Cross-domain user tracking with ETag and HTML5 prefetch.

Scenario 6: ***Cookie Stuffing Attack*.**

Online affiliate marketing is a form of online marketing that is particularly popular among businesses which, in addition to advertising their products/services, also sell these products/services online [20]. The three key players in any online affiliate marketing program include: 1) online retailer or merchant – the person or company selling something (e.g., *amazon.com*), 2) publisher or affiliate – the person or company which promotes the retailer's product(s) in exchange for a commission on the sale of those products (e.g., the owner of a popular blog site), and 3) user or customer – the person that buys the product(s) based on the affiliate's referral.

From the technical standpoint, there are two key enablers of online affiliate marketing (see Figure 12): 1) *unique affiliate identifier* which is used to create a special redirect link between the web-site of a particular affiliate and the web-site of a particular retailer, and 2) *affiliate cookie* which is created by the retailer and stored in the browser of each users that has arrived to the retailer's site by following the affiliate's redirect link. The affiliate cookie allows the retailer to identify and give credit to the right affiliate in case of an actually accomplished sale. For example, in the scenario illustrated in Figure 12, the cookie planted on Bob's computer during his visit to *http://www.amazon.com/productXYZ* via *trudy_blogger.com* will ensure that Trudy gets credit for Bob's potential purchase of this product irrespective of the exact time when such a purchase takes place – right away, or at some later point in time (as long as the cookie does not get overwritten by another affiliate's cookie).

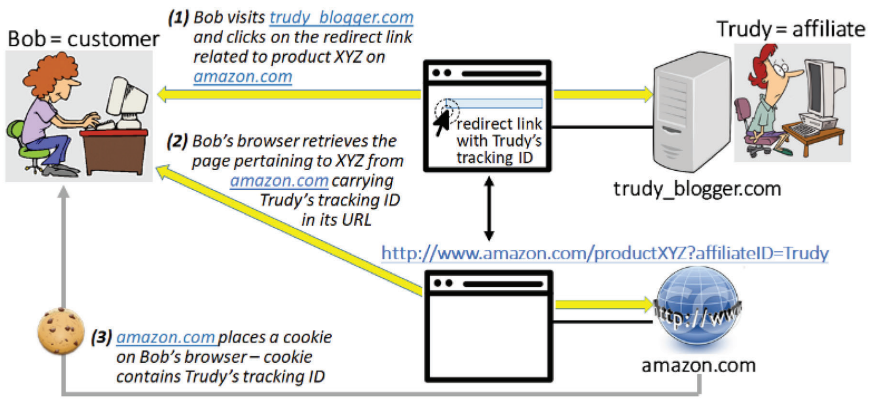
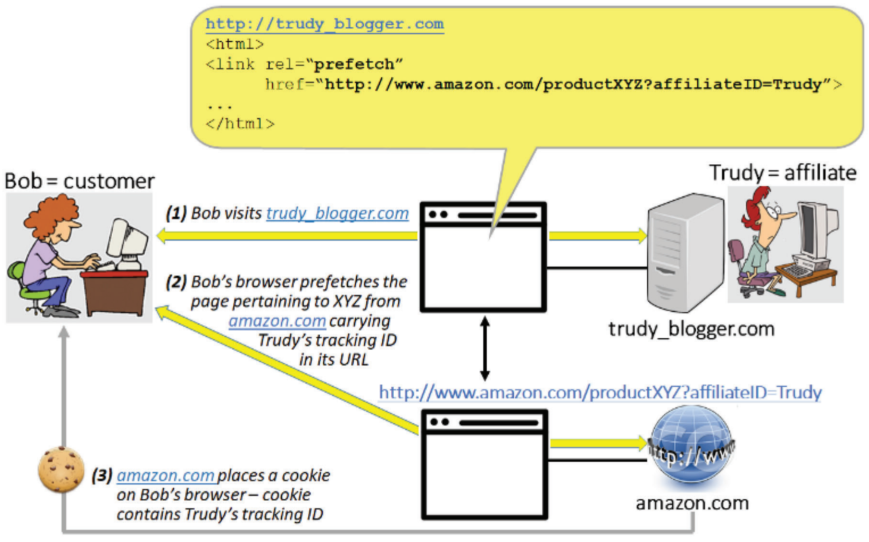

Bob = customer
(1) Bob visits trudy_blogger.com and clicks on the redirect link related to product XYZ on amazon.com
redirect link with Trudy's tracking ID
Trudy = affiliate
trudy_blogger.com
(2) Bob's browser retrieves the page pertaining to XYZ from amazon.com carrying Trudy's tracking ID in its URL
http://www.amazon.com/productXYZ?affiliateID=Trudy
(3) amazon.com places a cookie on Bob's browser – cookie contains Trudy's tracking ID
amazon.com

Figure 12 Mechanism of affiliate tracking.

Cookie stuffing is a form of fraudulent activity occurring within the framework of affiliate marketing [20]. In particular, the term refers to a range of scenarios and techniques by which an affiliate – without any knowledge or consent of the users visiting her Website – manages to trick the browsers of these users into following the redirect link to the retailer's site and, ultimately, get them 'injected' with her affiliate cookie. A situation in which cookie stuffing is performed by means of HTML5 prefetch/prerender functionality is shown in Figure 13. Here, by simple virtue of visiting



```
http://trudy_blogger.com
<html>
<link rel=“prefetch”
      href=“http://www.amazon.com/productXYZ?affiliateID=Trudy”>
...
</html>
```

Bob = customer
(1) Bob visits trudy_blogger.com
Trudy = affiliate
trudy_blogger.com
(2) Bob's browser prefetches the page pertaining to XYZ from amazon.com carrying Trudy's tracking ID in its URL
http://www.amazon.com/productXYZ?affiliateID=Trudy
(3) amazon.com places a cookie on Bob's browser – cookie contains Trudy's tracking ID
amazon.com

Figure 13 Cookie stuffing by means of prefetch/prerender.

(i.e., rendering) *trudy_blogger.com*, Bob's browser is 'lured' into requesting the redirect page with Trudy's tracking ID from *amazon.com*, and as a result it gets injected with Trudy's affiliate cookie. Through all this action, Bob remains completely unaware that his browser has ever had any interaction with *amazon.com*. Moreover, Bob also remains clueless to the fact that Trudy might rake in financial benefit from his potential purchase of `productXYZ` if, for example, he stumbles upon and buys this product through a direct search on *amazon.com*.

It is worth pointing here that in the scenario of Figure 13, Bob is neither solely nor the primary victim of the outlined attack. Namely, with potentially thousands of users visiting *trudy_blogger.com*, *amazon.com* would be the one ultimately experiencing the most significant financial impact of Trudy's cookie stuffing ploy.

6 Conclusions and Future Work

The goal of this article is to raise awareness of a slew of vulnerabilities that have been created with the introduction of HTML5 *resource hints*. We have provided examples of specific threats and attacks that are easy to mount and can have serious implications.

In order to mitigate these risks, further work is warranted and it can be structured within the general framework of handling threats; namely, to deter and block, and failing that, to be able to recover from an attack. These can be achieved by a combination of one or more of the following measures:

1. Browsers should have an option to disable *resource hints* so users can block potential attacks. Chrome provides such an option but is set to "allow" by default.
2. Browsers should make *resource hints* transparent, so that users are aware of them, without impacting the user experience.
3. Discriminating browser-initiated loads from user-initiated ones is currently done through the HTTP *Purpose* header, which is not logged by most servers. We propose that this be elevated to a request parameter (i.e., ?purpose=prefetch) so that it can be readily available during forensics investigations.
4. Increase awareness, particularly amongst expert witnesses and analysts, of the footprint left by *resource hints*. For example, if a page appears in a browser's cache but not in its history is a telltale sign that this was not a deliberate user-initiated retrieval.

We plan to pursue some of these directions in future works.

References

[1] Web Browser for Android Wear (2017). Google Play. Available at: https://play.google.com/store/apps/details?id=com.appfour.wearbrowser&hl=en

[2] Grigorik, I. (2013). *High performance browser networking: What every web developer should know about networking and web performance*. "O'Reilly Media, Inc."

[3] Resource Hints (2016). W3C Working Draft. Available at: https://www.w3.org/TR/resource-hints/

[4] StatCounter Global Stats (2015). Top 5 Desktop, Tablet & Console Browsers. Available at: http://gs.statcounter.com/?PHPSESSID=oc1i9oue7por39rmhqq2eouoh0

[5] Arthur, C. (2013). Why the default settings on your device should be right first time. theguardian.com, Available at: https://www.theguardian.com/technology/2013/dec/01/default-settings-change-phones-computers

[6] Bichler, M. (2001). *The future of e-markets: Multidimensional market mechanisms*. Cambridge University Press.

[7] February 2016 Web Server Survey. Netcraft (2016). Available at: https://news.netcraft.com/archives/2016/02/22/february-2016-web-server-survey.html

[8] Grigorik, I. (2013). High Performance Networking in Google Chrome. Available at: https://www.igvita.com/posa/high-performance-networking-in-google-chrome/

[9] Jackson, B. (2016). *Resource Hints – What is Preload, Prefetch and Preconnect? KeyCDN Blog*. Available at: https://www.keycdn.com/blog/resource-hints/

[10] W3Tech Web Technology Surveys (2016). Usage of Cookies for Websites. Available at: https://w3techs.com/technologies/details/ce-cookies/all/all

[11] Deveria, A. (2017). Can I use _____? Available at: https://caniuse.com/

[12] Gelernter, N., Grinstein, Y., and Herzberg, A. (2015). Cross-site framing attacks. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15),* CA, USA, 161–170. ACM.

[13] Rydstedt, G., Bursztein, E., Boneh, D., and Jackson, C. (2010). Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites. In *IEEE Symposium on Security and Privacy (S&P'10)*. Oakland, California.

[14] PC World. (2008). *The Julie Amero Case: A Dangerous Farce*. Available at: http://www.pcworld.com/article/154768/julie_amero.html
[15] The Register. (2009). How malware frames the innocent for child abuse. Available at: https://www.theregister.co.uk/2009/11/09/malware_child_abuse_images_frame_up/
[16] Burp. Available at: https://portswigger.net/vulnerability-scanner/
[17] Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet (2017). OWASP, Available at: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet
[18] Basit, A., and Vlajic. N. (2017). CSRF Attack Using HTML5 Resource Hints: A New Face of an Old Enemy. In *IEEE Cyber Science and Technology Congress*.
[19] Bujlow, T., Carela-Español, V., Solé-Pareta, J., and Barlet-Ros, P. (2017). A Survey on Web Tracking: Mechanisms, Implications, and Defenses. In *Proceedings of the IEEE*, 105.
[20] Snyder, P., and Kanich, C. (2015, December). No Please, After You: Detecting Fraud in Affiliate Marketing Networks. In *WEIS,* Amsterdam, Netherlands. Available at: https://www.cs.uic.edu/∼ckanich/papers/snyder2015noplease.pdf

Biographies

Natalija Vlajic is an Associate Professor at the Lassonde School of Engineering, York University. The main areas of her research include: user privacy and anonymity, DDoS, Internet bots and botnets, network and application-layer security, IoT security, machine learning. Prof. Vlajic has co-authored numerous journal and conference articles on a range of topic pertaining to computer security and privacy. She currently serves as an Associate Editor of IEEE Communication Magazine.

Xue Ying Shi is currently working for Tier1CRM Inc. as a full stack software developer developing CRM related applications. She received B.Eng. in Computer Engineering from York University in June 2017. She was a recipient of the Undergraduate Student Research Award from Lassonde School of Engineering, at York University, in the Summer of 2016.

Hamzeh Roumani received his Ph.D. in Theoretical Particle Physics in 1980 from the University of Illinois and has since been in academia at various Physics and Computer Science departments. His main area of interest is computer security and quantum computing. Hamzeh is a 3M Fellow and a recipient of numerous awards including the Ontario Leadership, the York University-Wide Award, the Faculty of Science Excellence in Teaching, the Lassonde Educator of the Year, and the Computer Science Mildred Baptist award.

Pooria Madani is a Ph.D. candidate at the Lassonde School of Engineering, York University, specializing in the areas of computer security and privacy, as well as adversarial machine learning. He obtained his M.Sc. from University of New Brunswick in 2015.