POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Vulnerability Analysis of Web Push Implementations in the Wild

Supervisors:

Prof. Cataldo Basile

Prof. Jason Polakis

Candidate:

Alberto Carboneri

Academic Year 2022/2023
Torino

Abstract

Web push is a novel technology, supported by all major browsers, which has gained significant traction in the developer community thanks to its ability to engage users efficiently and anonymously. However, security researchers have yet to properly investigate the possible threats arising from its improper use. In this thesis, we explore the capabilities and features of web push, report common usage patterns found in the wild, including an analysis of the inner working of most third-party providers, and present a security analysis of such implementations. We demonstrate a novel history-sniffing attack abusing a common implementation mistake, and a dangerous use case of the well-known CSRF vulnerability. We conduct and show the results of the first large-scale measurement aimed at identifying the prevalence of this technology and the related vulnerabilities on the web. The result of this measurement is also used to quantify the presence of a common negative pattern where websites aggressively ask the user for permission to use web push. Furthermore, we analyze the complexity of efficiently and correctly implementing personalized web push notifications and we report design issues we found on Twitter. We propose a theoretical system that corrects those mistakes and better handles all scenarios. Finally, we present some straightforward countermeasures and good practices to effectively fix the reported vulnerabilities and make the technology safer. Overall this work is intended to remark on the dangers of developing and implementing new technology without considering the security implications and to shine a light on some of the vulnerabilities present in implementations in the wild, possibly leading to a greater interest of the security community and further research on this and related subjects.

Acknowledgment

First, I want to thank my family, whose support and love was essential for me in completing my studies and this experience in Chicago. I also want to thank my thesis committee, Jason Polakis, Jon Solworth, and Cataldo Basile for their guidance and support. I would like to dedicate a special thanks to all the members of the student team pwnthem0le for making me feel part of the family and making me passionate about cybersecurity, as without their support I would not be where I stand. Finally, I want to thank my friends, my roommates Marco and Rebecca, and all the people I met here in Chicago, for making it feel like home.

Table of Contents

List of Figures

List of Tables

Acronyms

CSRF
Cross-Site Request Forgery

OS
Operating System

VAPID
Voluntary Application Server Identification

SOP
Same Origin Policy

CORS
Cross-Origin Resource Sharing

ZAP
Zed Attack Proxy

OWASP
Open Worldwide Application Security Project

MITM
Man In The Middle

Chapter 1

Introduction

In recent years there has been a shift from native applications to web applications, which run on browsers, and provide users with a more dynamic experience and a seamless transfer of data from mobile to desktop. The growth of such technology has been exponential and browsers vendors have been pushed to develop new features that applications can use to provide a better user experience. Given the rapid development and the complexity often involved, it is imperative for the security community to deeply analyze the related issues and possible vulnerabilities. If left unchecked, those new capabilities could lead to dangerous attacks against a great number of users.

In this thesis, we will analyze web push notifications, one feature that allows websites to behave more like desktop applications, sending notifications directly to the user's computer. We will show some issues we found in implementations in the wild, describe our methods for detecting those, and show the results of a measurement we did about the prevalence of the technology and the related vulnerabilities on the web. The work focuses on web push notifications, but the findings and tools are potentially applicable to similar technologies. As this technology is currently lacking proper research on the security aspects of its implementations, the objective of this thesis is to demonstrate the existence of security issues and be a starting point for future analysis of the related threats.

1.1 Motivation

The inspiration for this work comes from a research by S. Karami et al. [1], which analyzed the service worker technology with a focus on the caching ability. Their findings, which showed that, for example, in some cases, such cache can be used by attackers to obtain privacy-sensitive information off a victim, shed light on the potential other issues in this complex browser feature. Service workers are commonly used to handle caching,

background sync, and handling web push notifications, but the above-cited paper mainly analyzed the first of three. Given the promising results obtained on the caching feature, this research intends to deeply analyze the web push notification technology, looking for potential new issues. Moreover, as previously mentioned, the current security research on the subject is lacking with respect to other fields. Even though service workers have been implemented in all major browsers since 2015, security research is only recently catching up, and the majority of it is focusing on either the caching or background sync features, mostly ignoring the widespread use of web push. As this technology is becoming more and more prevalent, the intention behind this work is also to stimulate further research to identify and solve issues before they arise.

1.2 Thesis structure

In Chapter 2, we give a brief overview of the history of web push notifications, and we go through the required background to fully understand the rest of this thesis.

Chapter 3 shows our findings, the found vulnerabilities, and proposes real-world attacks and potential solutions.

Chapter 4 describes an extensive measurement of the prevalence of web push notifications technology on the web and the usage of third-party providers by the top 500 thousand websites. Furthermore, we analyze the number of websites potentially affected by two specific vulnerabilities.

Chapter 5 discuss about limitations of our approaches and ethics of this research.

Chapter 6 illustrates previous studies in the field and related technologies.

Chapter 7 summarizes the work, discusses how new results could be obtained, and potential future works in the field.

Chapter 2

Background

In this chapter, we will provide the reader with the required background to fully understand the rest of this work. First, we will summarize the technology behind service workers and modern push notifications. Then we will provide a brief overview of the classic CSRF vulnerability.

2.1 Service Workers

Service workers are a modern web browser feature that allows developers to manage network requests, caching, background sync, and push notifications. They work in the background, allowing applications to load faster, handle specific situations, and provide an overall better user experience. They take an essential role in modern progressive web applications [2], a newer generation of web applications aiming at providing an app-like experience on any device. For example, by caching an application-specific offline page, whenever the user visits the website without an internet connection, such page can be loaded by the service worker instead of showing the browser's default, providing the user with a seamless experience. Moreover, caching images, stylesheets, and other resources will benefit the page load time in the future, especially for mobile users, for which the internet connection may be slow or unreliable. Finally, all the above with background sync can allow for complete offline website usage. For example, a user may send emails while offline using the cached frontend. Such emails can be stored temporarily in the local browser storage and then sent automatically by the service worker when connectivity is restored.

A service worker is simply a JavaScript file registered by a website on an origin and a path, which works by responding to events and taking action accordingly. For example, a service worker may intercept all network requests to check whether the response has been previously cached and return the cached page instead of forwarding the request to the

target server. They work in a separate thread from the main one rendering the website's DOM, providing non-blocking, fully-async scripting, but without direct access to the page's content [3]. Unlike other workers, they can exist without a reference from the page, so they can be temporarily turned off when not needed and resumed when a specific event occurs. This is especially important in mobile browsers, as energy saving is generally considered more important in such devices.
There are a total of six events that service workers can respond to.

Install event. It is fired when a website first installs a service worker. It is normally used to set up databases such as IndexedDB, cache common resources, and inform the backend of the successful installation. After this event has been completed, the browser detects and deletes older versions of the service worker.

Activate event. It is fired after the install event completes. A typical operation done during this event is the cleanup of leftover data from previous versions of the service worker.

Message event. As the DOM thread cannot communicate directly with the service worker, it must use the postMessage API to do so. When a message is sent using such a method, a message event is fired on the SW side.

Sync event. A common use case of service workers is, as mentioned before, allowing the usage of a web application in an offline context. Using this event, a server can send updated data to the service worker when the website is not open. When the user visits the website later, possibly while offline, the updated data will be available without downloading.

Fetch event. It is fired every time a web application resource is loaded. It can be used to have finer-grained control over network requests and caching. For example, it allows developers to intercept requests to already cached resources and return them with a much shorter delay.

Push event. This event is fired each time the server sends a push notification to the browser. It is used to style and display the notification properly. A push event can be fired anytime and will wake the service worker if it is idle.

The benefit of service workers, with respect to ordinary JavaScript, is the ability to be loaded and kept in memory by the browser. This allows the script to run even in offline contexts and contexts where the target origin is not currently open in a browser tab. With respect to push notifications, the latter is especially important as it allows websites to send messages directly to the user's computer even when not loaded. Finally, one limitation of service workers is the website's requirement to support HTTPS. This is for security reasons, as code injection in a service worker's source code may pose a high risk to the user's privacy as an attacker may be able to use the above event handlers, for example, to intercept and modify requests.

2.2 Web Push Notifications

Push notifications are a form of communication traditionally used by desktop and mobile applications and made available to websites through modern browsers, under the name of web push notifications, in more recent times [4]. They provide website owners with an easy way to send real-time customized messages to users even when such users are currently away. Those messages can be sent to a "mailbox" and automatically delivered to users when they open their browsers. Those messages can contain a title, a description, an image, and a URL. This technology is becoming essential for websites to increase user engagement and traffic to their pages. It has several advantages over traditional email and SMS messages, the most important one being that it allows for the anonymization of the user. In fact, no telephone number or email is required to subscribe to such a service. Moreover, push notifications are more easily seen and interacted with immediately by the target, while the other above-cited delivery methods tend to take some time before being noticed. A 2016 report by pushcrew [5] showed how web push notifications, even though, like all mass communication channels, report a decrease in click rate as the number of recipients increased, still outperformed all other channels used for websites. This is an indicator of the growing importance of such technology.

2.2.1 Technical Overview

Web push notifications work by using the Push API, which is built on top of the Service Worker API [6]. This service is permission-based, which implies that when a website wants to start sending push notifications to the user, it should ask the user for permission to do so. In particular, it should first check whether the user previously granted permission, and if not, it should ask the user for it by using a special browser prompt, which guarantees the clarity of the request. To ask for such permission, two different approaches can be used. In the first approach, the request is directly made to the browser, which shows the prompt to the user. This is generally considered a bad practice, as it is annoying and invasive, leading to a bad first-time experience on the website and potentially leading customers away. A study by J. Hofmann [7] showed how, of the 18 million notification request prompts shown to users on Firefox Beta from Dec 25, 2018, to Jan 24, 2019, only about 3% got accepted, and almost 19% caused the users to immediately navigate away from the site. In the second approach, a so-called "soft ask" mechanism is employed, where a button is placed on the website page, which, when clicked, triggers the browser prompt. The latter allows websites to describe the purpose of web push notifications in the website context. For example, an online store may tell the user that web push notifications are used to remind them of abandoned carts. This allows users to be shown the browser prompt only if they are interested in accepting notifications, reducing the negative effect and increasing the effective click rate. Some browsers, including Firefox and Safari, forbid the former method and require user interaction before showing the prompt [8] [9]. Whether the permission was previously granted can be detected programmatically by checking the

Notification.permission property, which will contain granted if true, default or denied if not. It is important to note that such permission is related to a specific origin, and accepting it on a specific website does not mean granting it globally. The permission can then be requested in one of two ways, by either calling Notification.requestPermission() or pushManager.subscribe(). The former can be directly called from the JavaScript running on the main page, will request the permission, and return the result without actually subscribing the user if such permission has been accepted. Websites can use this to install a service worker only when the permission was accepted beforehand. The latter can only be called in the context of a service worker, as it requires the pushManager interface, which is accessed only by the ServiceWorkerRegistration interface. This API checks the current status of the permission and first asks the user for it if it was not previously accepted. Once the permission has been granted, it will return a Promise, which resolves to a PushSubscription object containing the subscription data, enabling the website to send push notifications. Such subscription data will contain a pair of encryption keys and an endpoint at the browser's vendor server, specific for that user and origin, as seen in 2.2. This vendor server acts as a temporary storage for all notifications waiting to be delivered. Each time a subscription is created, a specific endpoint on such server is generated, to which websites can make authenticated requests to add a notification to the user's "mailbox". One browser will have multiple endpoints leading to it, one for each origin allowed to send web push notifications to it.

As seen before, this whole process lacks an identifier directly related to the user, such as name, telephone number, or email, and allows for anonymous opt-in simply by clicking a button. The website only receives a random endpoint on the browser's vendor server, which is the only one capable of identifying the related browser. An added level of security is the need for a VAPID public key to be sent to the vendor during the subscription process. This ensures that in the scenario where an attacker obtains the subscription data of a user, it cannot be used to send notifications without the corresponding VAPID private key.

The subscription data needs to be sent to the website's server to be stored and used to send notifications in the future.

A WebSocket connection is kept open by the browser to its vendor servers, where the push service is hosted (e.g., Mozilla autopush). This connection is opened when the browser is started. When a web push notification needs to be sent, the server contacts the push service at the target endpoint for a specific recipient with an HTTP POST request containing the encrypted notification data and any criteria for delivery, such as the priority of the message. APIs to make such POST requests are standard [10], so the backend does not need to know which push service it is talking to. The push service receives and stores it until the target browser comes online. As soon as the user's browser connects, the service sends the notification data to it. The browser then checks whether the target website's service worker is alive or not. In the latter case, it starts it and then delivers a push event with the related subscription data. Therefore, a push event listener needs to be implemented in the service worker code, which will receive the event containing the required pieces of information about the notification, style it properly, and show it

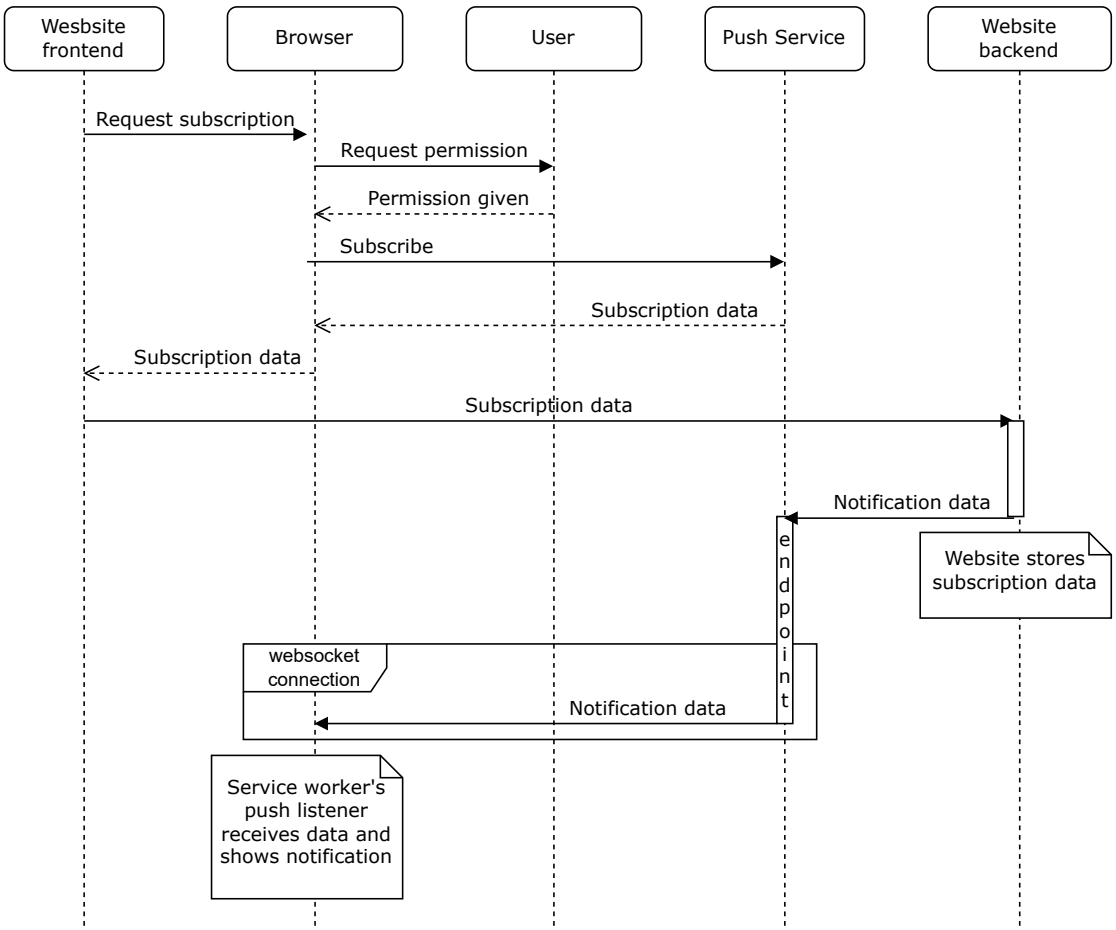

Figure 2.1: Flow of a successful subscription and subsequent notification.

```
{
    endpoint: 'https://fcm.googleapis.com/fcm/send/
cwEL8oM02a8:APA91 ... cg',
    expirationTime: null,
    keys: {
        p256dh: 'BPm4u ... vvNrY',
        auth: 'bv1SgdyGs9vT30x2-5BoRQ'
    }
}

```

Figure 2.2: An example of push subscription data.

```
1         self.addEventListener ( ' push ', function ( event ) {
2             var title = data.notification.title;
3             var message = data.notification.URL;
4             var image = data.notification.image;
5             event.waitUntil (
6                 self.registration.showNotification ( title , {
7                     body: message ,
8                     image: image
9                 })
10            );
11        }) ;
12
```

Figure 2.3: An example of push event listener.

using the method self.registration.showNotification(). This API calls the low-level OS API, which shows the notification.

2.2.2 Third party providers

The code required to successfully implement web push notifications, as seen in previous sections, is quite complex, and many websites lack the capabilities to write and execute it correctly. Many companies were born in the last decade which sells web push notifications as a service, allowing more straightforward implementation of this technology. By subscribing to their service, the website's owner gets access to a complete dashboard from where they can see statistics and send new notifications, and a small JavaScript file, which, when included in the website's source, will handle all the complexity. One example of such a provider is oneSignal[1], which dashboard is shown in 2.4.

To integrate the service worker, two approaches are used. In the first approach, the worker is given to the owner as a JavaScript file, which needs to be uploaded to a specific directory on the website. This is the easiest one, but it has some limitations. For example, a website may already have a service worker, and integrating the scripts may be challenging, or the website may not support HTTPS, which is required for web push notifications to work. Moreover, it requires the website's owner to have the possibility to store files on the disk, which some hosting providers do not guarantee. In the second approach, a custom subdomain is created for the customer on a domain controlled by the provider. Then, a script is included in the main website, which creates a soft-ask window that opens such a subdomain on a pop-up when clicked. This pop-up request permission and, if

[1] https://onesignal.com/

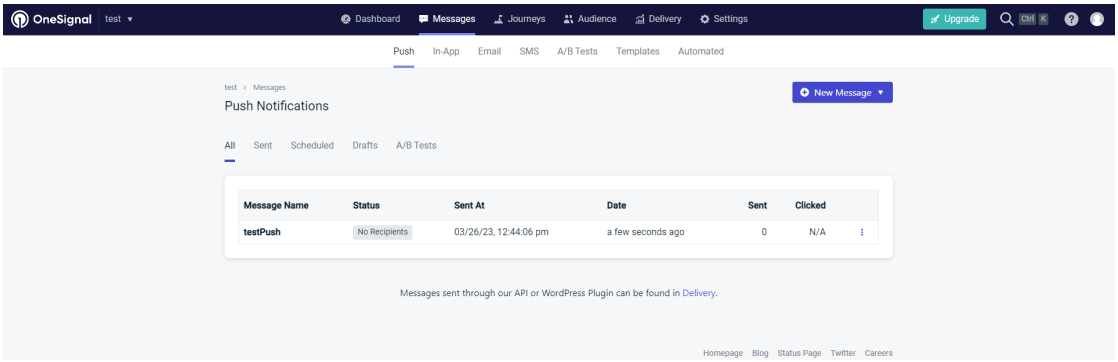

Figure 2.4: oneSignal dashboard, used for testing the service.

accepted, installs the service worker. Then automatically sends the subscription data to the third-party provider. It is important to note how, by following this method, notifications are effectively sent from the third-party subdomain instead of the customer's website, as in the first case. For checking whether the user is subscribed or not, an iFrame is used, which is included in the website by the script, loads a specific page in the third-party subdomain, and sends back a message using the postMessage API after checking for the presence of a cookie. The latter method requires more technical work to handle subscription, unsubscription, and subscription changes, but it allows providers to target more potential customers and is a common option.

2.3 CSRF

CSRF is a well-known vulnerability that allows an attacker to have a victim perform an action on a website without their knowledge. It works by taking advantage of session cookies. When a user logs into a website, one or more session cookies are saved in the browser and are sent back with any request to the same domain to allow for the identification of the user. This is standard across the web, as cookies are the most common way to keep track of a specific user. Cookies may have different attributes, one of which is SameSite, which is used to instruct the browser on which cases the cookie should be sent. Requests can have two different origins:

- SameOrigin is the case for every request made to the same origin as the one it is starting from.
- CrossOrigin identifies requests made to a different origin.

The attribute SameSite can have three different values.

- Strict

- Lax
- None

When set to Strict, cookies are sent only in SameOrigin requests. For example, a request from example.com to target.com would not include cookies set on target.com with the attribute SameSite set to Strict, while a request from target.com to target.com would. A value of Lax allows cookies to be sent only in SameOrigin requests and when a user navigates to the target website. This is the default value and was made so to specifically protect against some classes of CSRF attacks. Finally, a value of None allows cookies to be sent in every request.

In the most common scenario, cookies should have the property set to one of the first two options. However, specific cases may require session cookies to be sent across origins, requiring using the None value. For example, a website whose frontend resides on an origin but whose backend is hosted intentionally on a different subdomain would be an example of an exception, requiring the usage of the less secure None value. Whenever this is done, additional protection should be ensured as this attack could take place easily. An attacker

```
1    <html>
2      <body>
3        <form action="https://mybank.com/transfer" method="POST">
4          <input type="hidden" bsb="43123" accountNo="832848293" amount=
   "100" />
5        </form>
6        <script>
7          document.forms[0].submit();
8        </script>
9      </body>
10   </html>
11
```

Figure 2.5: An example of malicious page abusing a CSRF vulnerability.

can craft a malicious page, hosted, for example, at https://attacker.com, which contains, as a source, code similar to what is shown in 2.5. When a victim visits such website, an automatic POST request will be made to the target URL, including the hidden data and cookies whose SameSite attribute was set to None. This may trigger the intended action with all the consequences. For example, in the case of a web application that allows for changing the user's name with an HTTP request and is vulnerable to CSRF, when the victim has logged on to the web application and visits an attacker-controller website, it may find its name changed on the target website. The impact of this vulnerability can vary, depending on the application affected and the attacker's ability to have the user visit a controlled website. A common prevention is the usage of CSRF tokens, unpredictable,

unique strings which are sent by the server to the client in a previous request and need to be sent to the server with the target request to make it valid. This approach relies on the SOP to avoid the JavaScript on the attacker-controlled website reading the CSRF token on the target. This is a security protection implemented in modern browsers which forbids the JavaScript running on a page from reading responses to requests made to a different origin unless the target origin explicitly allows it by using the correct CORS response headers. A poorly configured CORS policy may allow a CSRF attack to happen even when a token is used. For example, a web server that, to enable cross-origin requests from a specific website, sets the Access-Control-Allow-Origin to * in all responses accidentally allows access to all resources from all origins, making it vulnerable to CSRF in any case.

2.4 Threat model

The threat model proposed in this thesis is similar to models commonly used in other works about history sniffing, CSRF, and other privacy-invasive attacks. For all attacks related to the above topics, we assume the attacker can run JavaScript in the user's browser in the context of an attacker-controlled origin by having the user visit such origin. However, most of those attacks could be theoretically implemented at a larger scale, for example, by using ads as an attack vector.

Chapter 3

Vulnerabilities

In this chapter, we will illustrate the issues and vulnerabilities we found during this research and describe the methods used to detect those in the wild. Based on the known applications of the technology, we identified three distinct areas to start our research from:

- Personalized web push.
- Third-party providers.
- CSRF issues.

For each of the above research areas, we analyzed the related potential problems and identified vulnerabilities. Then, with the help of partially automated frameworks, we found real examples in the wild.

3.1 Personalized notifications

Push notifications are commonly used to send identical messages to a broad audience of subscriptors. However, the nature of the technology allows for sending personalized web push notifications targeted at a specific user. For example, a social network may send notifications about new friendship requests, which must be sent only to the targeted user. This kind of notification is more privacy sensitive and more complex to automate, which is why many websites, and almost all third-party providers, do not implement those. In this section, we will first propose a basic system for handling such notifications, show possible implementation errors, and then describe the possible impact of such errors. A basic design of this system can be similar to what is shown in 3.1. Only a column to the user table may be added to correctly identify the user, simplifying lookup and reducing the

Table 3.1: A POSSIBLE STRUCTURE OF THE USER TABLE INCLUDING SUBSCRIPTION DATA.

userid	username	password	subscription
1	Mark	spFRgWTNLjfx . . . nkH4Gh4DeJKOJM	endpoint,keys
2	David	DpRRgSDVjfd . . . nkGTDh4De68VMP	endpoint,keys
3	Maria	DURgSEWNLHfx . . . nk4EW3FxrV4M	endpoint,keys

impact required to implement it to the minimum. However, in reality, more than this approach is needed, as a user may have multiple browsers and sessions, each of which may have different subscription data. A whole new table to store the multiple relations needs to exist in this case, as without it, only one browser at a time may receive notifications, and a new browser will disable the permission on every previous login. When a notification needs to be sent, the server must look up all the subscription data linked to this user and send the notification to all of them. However, the backend must be careful to detect when a user logs out or his session expires, as such events must disable a subscription temporarily until a login session on the same browser is created again. This is to avoid notifications being sent to the browser which, for example, may be running on a shared computer. So the database must also store, in that table, a relation to the sessions table for each subscription. Each time a notification needs to be sent, the server must check the current user's subscription data table, filter out all invalid or expired sessions, and send the notification to the remaining ones. This operation should not delete the subscription data when the user logs out unless a proper mechanism is implemented to request the service worker to send the data again after each login. It has to be noted that this is generally more complex than simply keeping track of the related data in the database. This approach requires the backend to update this table each time a new session is created or expired and correctly link that session to the current browser's subscription. For example, when a user logs in with a browser on which he has already accepted notifications for that website, the session will be new, but the subscription data will be the same. The server must therefore recognize this and store the information correctly. Moreover, this approach requires that each time a new subscription is generated, it is returned with the session cookies. The server then must handle the request correctly, fingerprint the browser, and store the correct data in a database table for future usage.

What we explained is a basic idea of how to implement such a system. A more complex scenario may require more complex database tables and checks to be correctly implemented. 3.1 shows an example of correctly handling logging out from a browser. As we can see, the whole system is non-trivial to develop and is prone to errors. In the following sections, an overview of problems caused by the improper use of this technology is discussed.

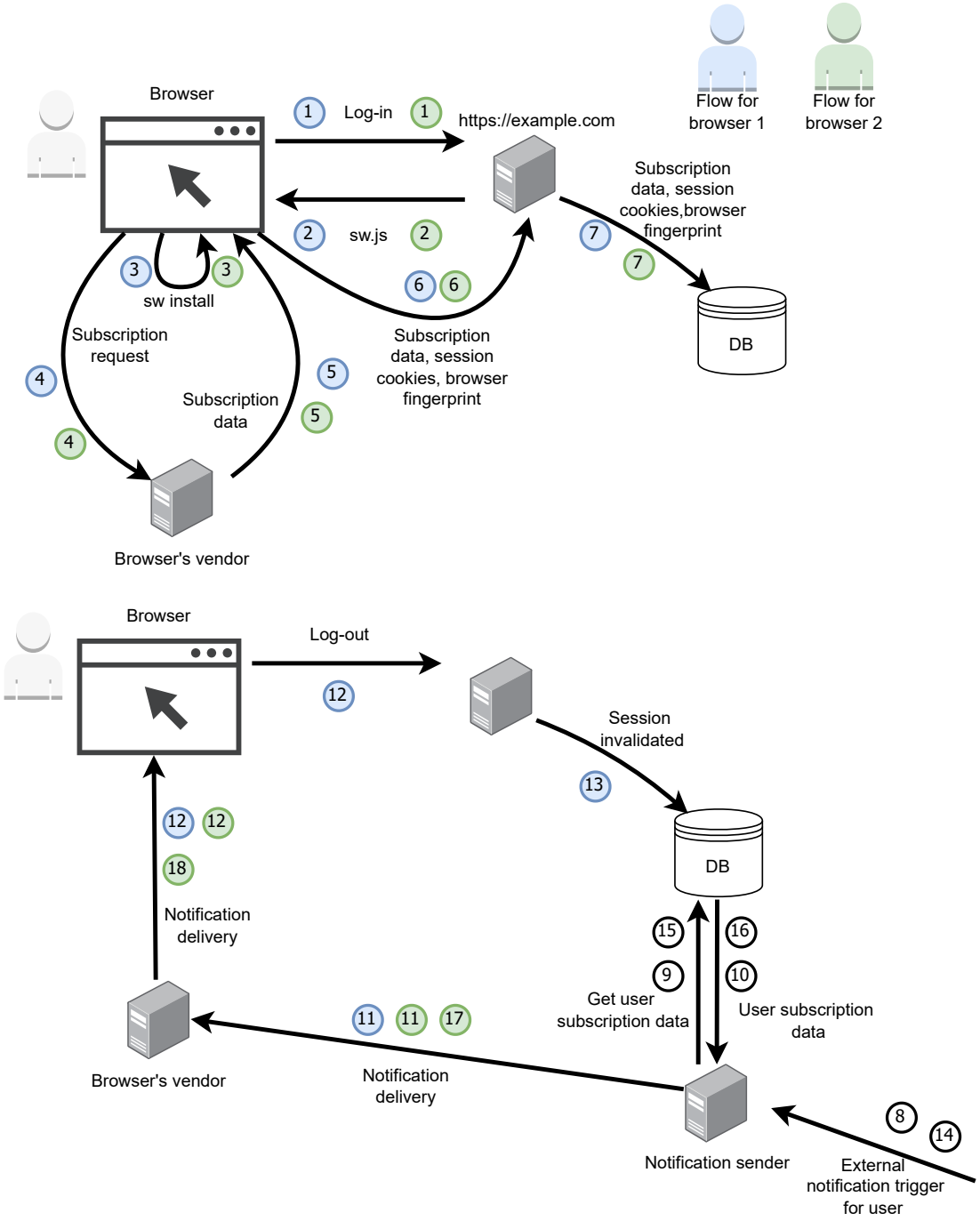

Figure 3.1: System to handle personalized web push notifications showing correct behaviour on logout.

3.1.1 Incorrect user logout handling

As said before, when a user logs out or his session expires, the server must detect it and react accordingly. Specifically, it should invalidate the subscription until a new session on the same browser is created. If this is not done correctly, notifications may still be sent and delivered to unintended people currently using such browsers. For example, a user may share a laptop with other people. In such cases, no notifications should be shown when the user is not logged in, as other people may read them.

By manual analysis, we found that poshmark.com has this issue. Poshmark is an e-commerce website to buy and sell used or new clothing, where users may send messages to each other. Those messages will also be shown as web push notifications if the user has subscribed. If the user logs out, notifications are still sent and may leak personal information to others.

3.1.2 Other design issues

Implementation of the system must work well in all cases and be robust to unexpected actions of the user. There are many examples of correct behaviors which should be followed but may not be in some circumstances. Some of those are listed here.

- Having multiple sessions simultaneously should not affect which sessions receive notifications.
- The action of logging in, logging out, and logging in again possibly multiple times, should work seamlessly for the user without the need to re-subscribe to notifications.
- The expiration of a specific session, for any reason, should not affect how notifications are sent to other currently active sessions.

One example of a website that doesn't currently implement the system correctly is Twitter. After comprehensively testing its behavior, we found that some scenarios exist where the above rules are not respected. First, when a user logs out and then logs in again, notifications are not sent anymore, and a manual interaction is required to restart the subscription process. Second, if a user logs into his account from two different machines, which have the same OS and browser (regardless of their version), the server will only send web push notifications to the most recent one.

The above issues are probably caused by improper storing of the session in the subscription table and incorrectly identifying a browser by the OS and the vendor instead of another more strong identifier, which should be used instead when referring to the subscription table. According to our research, Twitter's database's table to store web push subscription data may be similar to 3.2. A simple solution to both issues is creating a

Table 3.2: A POSSIBLE STRUCTURE OF THE SUBSCRIPTION TABLE IN TWITTER'S DATABASE.

userid	session	OS	browser	subscription
user1	nJW3mn3Su9E . . . BHQY4u7HjKBwY	MAC OS	Firefox	endpoint,keys
user2	spRRgWNLjfx . . . nk4Eh4DexrVMM	Ubuntu	Chrome	endpoint,keys
user3	2VrU7RTr4pU . . . eU8cCh4qDBccT	Windows	Brave	endpoint,keys

proper fingerprint for the current browser, for example, a unique cookie stored locally, which is sent on every log-in, and which uniquely identifies such browser's subscription.

3.2 Third party providers

As discussed before, the technology behind web push notifications allows for third-party providers to exist. Of the two ways of operation, the one where a pop-up is shown to request permission on a subdomain of the provider may present some vulnerabilities in specific cases where the security of the code running on such subdomain is not correctly handled.

3.2 shows the flow for a subscription with the above method. In this case, for the main website to check whether a subscription is already present, an iFrame loading a specific page of the third-party subdomain is used. This page's JavaScript code will then check for the presence of a cookie and send back the result using the postMessage API. This API is the only way for websites to communicate with the code in a loaded iFrame but requires the receiving origin to declare a message event listener. This ensures that messages are only received and handled intentionally, and no code can be executed on a different website. Another security feature of the postMessage API is the targetOrigin parameter, which specifies to which website the message should be delivered. In fact, for a page loaded into an iFrame, it can be hard to correctly identify the parent page whose it is currently loaded into. However, such a feature is often unused, and, for practical reasons, the targetOrigin value is often left to *, indicating any origin. Another common practice is to get the target origin through an URL parameter, set by the including page. In such circumstances, an attacker may include the iFrame in his website and listen for messages. Then, the iFrame would tests the presence of target cookies on the subdomain and return the information by sending a message. The attacker can receive the message and detect whether the user previously subscribed to that specific website. This attack allows for history sniffing and is shown in 3.2.

The sent message commonly includes a unique identifier for the user's subscription, potentially leading to further attacks.

Since each website using this approach has a custom subdomain on the third-party

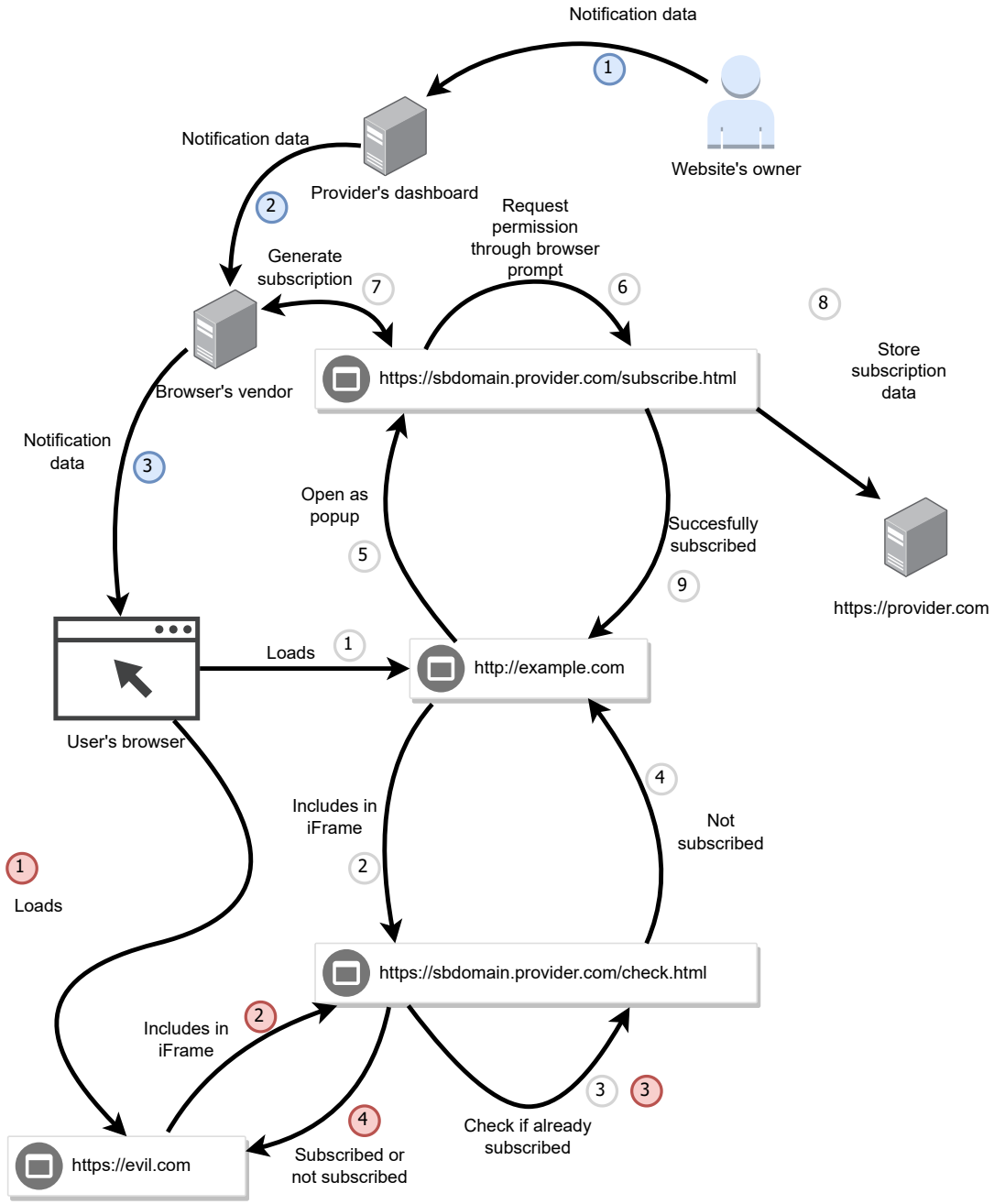

Figure 3.2: Flow of a subscription and subsequent attack using a third-party provider.

provider, the attack allows targeting specific websites by only including the specific iFrame for that customer. Such subdomains can be easily collected from the scripts loaded into the target website, which necessarily includes the required URL. Moreover, it is possible

to target multiple websites simultaneously by including multiple iFrames and checking the origin of the received messages. 3.3 shows an example of a malicious page targeting a specific provider and customer.

```
<html>
  <head><title>letReach history sniffing</title><script>
      window.addEventListener('message', function(event) {
          if (event.origin == "https://letreach.com" || event.origin
 == "https://subdomain.letreach.com" || event.origin == "https://
 subdomain.ltrch.com") {
              console.log("event " + event.data);
              const res = document.getElementById("result")
              if(event.data.length==1){
                  res.innerText="User did not visit/accept
 notifications on website"
              }else{
                  res.innerText="User did visit and accept
 notifications on website"
              }
          }
      }, !1)
  </script></head>
  <body>
  <iframe src="https://subdomain.letreach.com/ask.html?
 checkPermission=1" visibility="hidden" style="width: 0px; height: 0
 px; border: 0px; display: none;"></iframe>
  <div id="result"></div></body>
</html>

```

Figure 3.3: An example of malicious page abusing the history sniffing vulnerability on letReach.

By finding the list of websites the victim has previously visited, the attacker may infer some additional information that could lead to further attacks.

A simple set of countermeasures to this kind of attack is correctly using the targetOrigin parameter of the postMessage API by not using the value * and having either a single website allowed to include the iFrame or a proper whitelist. Moreover, data sent to the including website should be minimal, containing only a boolean flag indicating the current status of the subscription, to further reduce the impact of a possible leak.

An extensive analysis conducted on the top fifty third-party providers [11] shows that the issue does exist and is common. For Example, Webpushr, the third most commonly used provider [11] suffers from this vulnerability.

Cleverpush is another provider which suffers from a more dangerous variant of this issue. In any mode of operation, whether directly included in the website or loaded into an iFrame, it saves a cookie on a subdomain of mycleverpush.com after a successful subscription. When loading a specific page of that subdomain into an iFrame, it exposes a custom API over the postMessage API, which can be used by the including website to query the status of the subscription. This API allows, as before, for detecting whether the user previously accepted notifications on a target website, but also to receive a list of notifications waiting to be sent to the user, a list of previously clicked-on notifications, and the possibility to unsubscribe the user from the service. Moreover, the unsubscribe is transparent to both the user and the target website, leading to it not requesting again for permission unless all browser's data, in both the original website and the mycleverpush.com subdomain, are deleted.

3.3 CSRF

In the context of web push notifications, CSRF is a critical vulnerability to protect against. In this section, we want to emphasize the severity of the issue. As shown before in 2.1, after a successful subscription, the frontend needs to send the subscription data to the backend for it to be able to store it and send web push notifications to it in the future. As this request, when the notifications are personalized, necessarily contains the user cookie to allow for the link of the subscription data with the current session, it is critical to protect. If CSRF protection is missing, it is possible, whenever a user visits an attacker-controller website, for the attacker to trick it into sending such request with the attacker subscription data instead. In this event, the user's private notifications will be sent to the attacker. The attack is entirely silent, as the victim does not need to have accepted notifications previously. The server receives the subscription data and treats it as if the user accepted the permission itself. 3.4 shows the flow of this attack.

To find an example of this vulnerability, first, we detected all websites which install a service worker and then removed from the list the websites which ask directly for permission without a previous login, as such websites are unlikely to send personalized notifications. The remaining list was analyzed using ZAP[1], an automated analysis framework developed by OWASP. This was used to find websites in the list which did not use CSRF tokens in other requests, as those websites were more likely to not use tokens in the push subscription request too. Finally, with manual analysis of the obtained list, we found that gama.ir has this vulnerability as it does not include CSRF tokens and makes use of the None value for the property SameSite of the session cookies. gama.ir is an Iranian website for teachers to share and publish educational content, including exam questions. As it does not protect against CSRF, an attacker can send his own subscription after tricking the victim into visiting a controlled website. This allows such attacker to receive and read

[1] `https://www.zaproxy.org/`

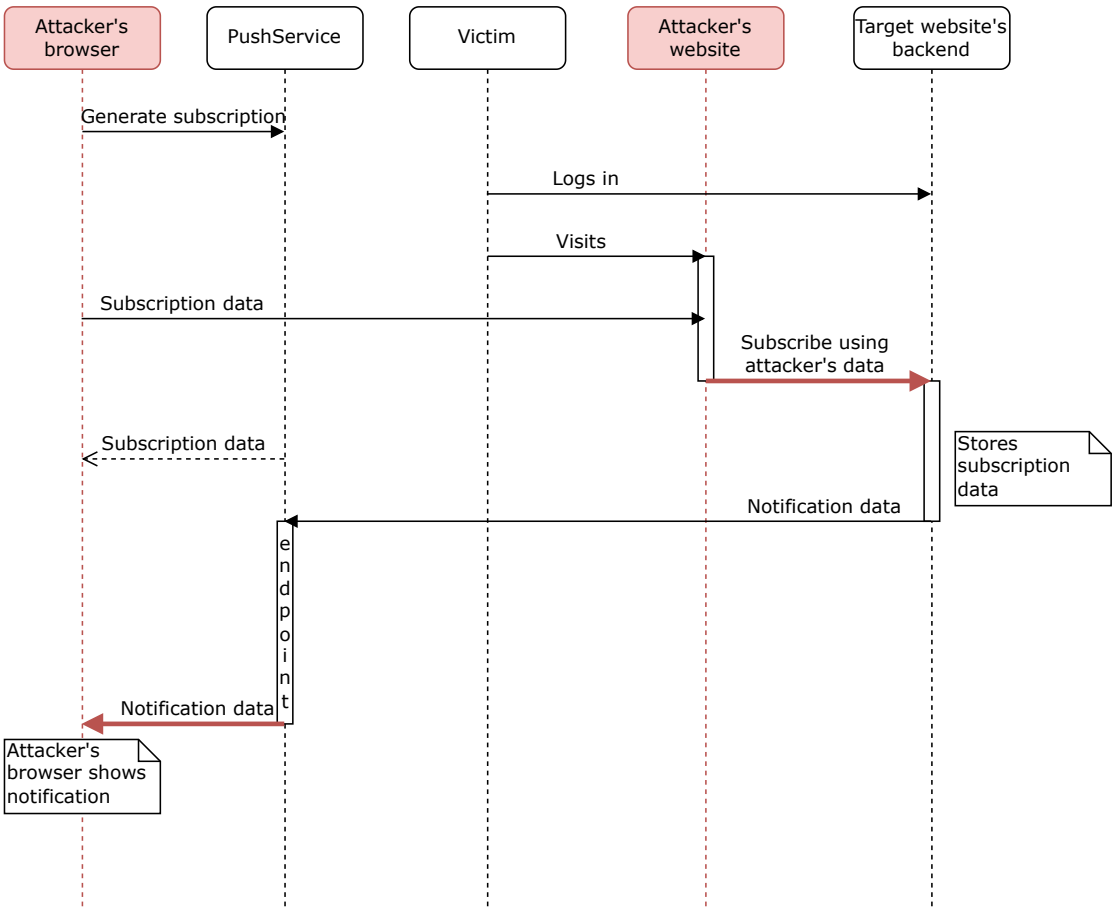

Figure 3.4: Flow of a CSRF attack.

Table 3.3: LIST OF THREATS.

Threat	Example	Additional notes
Notifications after logout	poshmark.com	alexa rank: 1,296
Design issues	twitter.com	
3rd-party providers	Webpushr, Cleverpush	
CSRF	gama.ir	alexa rank: 4,316

private messages and other information sent over notifications to the user.

Chapter 4

Measurements

In this chapter, we will present the methodology used to collect data from a large number of websites and analyze the results obtained.

4.1 Chromium-based analysis framework

To collect enough data and properly analyze the results, we created an instrumented version of chromium that can collect and log to file all calls to the Notifications and PushManager APIs. A full list of logged APIs is shown in A, Appendix A. Moreover, as neither chromium allows for automatic acceptance of web push notification requests nor puppeteer allows for quick interaction with such pop-ups, we also modified the browser so that whenever a website makes a request, it will be automatically accepted after a short amount of time. This allows our scripts to distinguish websites that directly ask for permission from websites that require user interaction.

4.2 Methodology

In our first measurement, the objective was to detect all websites which use web push notifications. To do so, we developed two approaches. In the first approach, the so-called dynamic one, we detect websites that directly ask for permission without user interaction. To do so, we launch an instance of the instrumented chromium on the target website, wait for it to load, which is detected by waiting up to a minute until the network was idle for more than half a second, wait an additional 30 seconds, close it, and look at the logged API calls looking for calls to Notification.requestPermission or PushManager.subscribe. This approach does not have any false positives but may have false negatives. The 30

seconds wait time was obtained empirically as a trade-in between speed and accuracy. No user interaction like scrolling or moving the mouse was implemented as no different behavior was found on a test run.
The second approach consists of statically analyzing the service worker's scripts. Service workers must be installed from a path in the same origin but can import other scripts from other origins using the importScripts method. We collect the URLs of the installed and imported scripts through the instrumented chromium by logging all API calls to ServiceWorker.install and importScripts. Our script then looks up all URLs and downloads all the sources. Static analysis of the source code of such scripts allows for detecting most of the remaining websites. The list of keywords for the static analysis was derived from the results of the dynamic one. This approach may have false positives as some service worker scripts may be flagged as containing web push notifications code even if they don't. The drawback of those two approaches is that websites that only install a service worker after the user accepts notification requests are not flagged as using the technology.

To try and reduce the number of such websites, we developed a third approach, which detects third-party providers. We manually developed a distinguisher for each of the first top fifty providers to detect if a website uses it. Such a list of distinguishers is shown in B, Appendix B. It was built by registering testing accounts on each provider, analyzing the instructions to include it in a website, and refining the obtained detection method by checking it against a list of known websites until almost zero false negatives were reported. Such lists were obtained from builtWith[1] after filtering, as with manual analysis we found some false positives, especially with lesser-used providers. Then, our script visits each website in the instrumented chromium, waits for loading to complete plus an additional 15 seconds and runs all distinguishers on the generated document. The wait time was chosen to be lower than the first approach as in a test run we found no differences in the reported results. To further increase the detection rate, we analyze the document for the usage of Google Tag Manager, a framework to manage tags on websites commonly used to include the specific scripts distributed by providers. If such a framework is detected, we download all tags included in the page by it and run the distinguisher on those too.

Other than complementing the previous approach by providing a list of some websites that use push notifications without installing a service worker before, this approach allows us to analyze the usage of such providers and detect the prevalence of the related vulnerabilities.

4.3 Validation

In order to evaluate the performance of our scripts we applied different approaches. The dynamic approach can be manually reported as having 0 false positives, as only websites

[1] https://trends.builtwith.com/widgets/push-notifications

Table 4.1: RESULTS REPORTED BY EACH INDIVIDUAL DETECTION APPROACH.

Source	Number of websites
Dynamic approach	3117
Static approach	8750
Third-party detection	11178

which make use of either the Notification or the PushManager APIs are flagged. By using the results of this approach we found that all websites for which it reported positive were also flagged by our static approach, leading to 0 false negatives on it. However, as previously mentioned, websites that do not install a service worker directly will still be reported as not using the technology by both methods.

To evaluate the third-party detection script, at most ten random websites (excluding unreachable ones) were selected from builtWith for each analyzed provider. As the list obtained was unlikely to contain many websites not using any provider, 20 external websites were added, taken randomly from the top 500k Tranco list. Then a manual analysis of each website was carried on to identify the used provider, and confronted with the reported result of the detection script. Overall we found, on 343 tested websites, 5 false positives (1.4%) and 16 false negatives (4.7%). A manual check of those revealed how the false positives were all due to old leftover code, while the false negatives were caused by either obfuscated code, cases where the code was downloaded and executed only after a certain user action, or websites using anti-bot features blocking our framework.

4.4 Results

The reported results for each approach are shown in 4.1. Overall our script succesfully run on 396154 unique websites and found 18566 websites that potentially use web push notifications (4.68%), of which 3117 (0.78%) directly ask the user for permission without any interaction. The amount, divided by rank, is shown in 4.2. Of all found websites, 11178 (2.82%) use a third-party provider. The complete list of providers with their amount is shown in 4.2. By analyzing those findings, we found that 1165 websites are potentially vulnerable to our proposed iFrame inclusion vulnerability, of which 469 are vulnerable to the more dangerous issue found on Cleverpush.
4.1 shows the amount of websites using web push, divided by custom or third-party implementation and by rank.

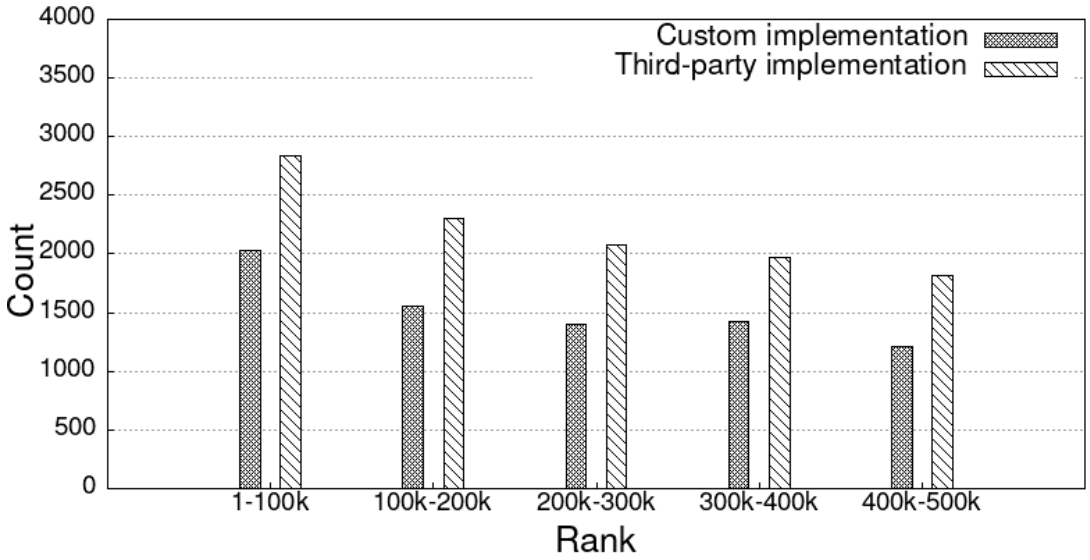

Figure 4.1: Web push custom and third-party implementations, divided by website rank.

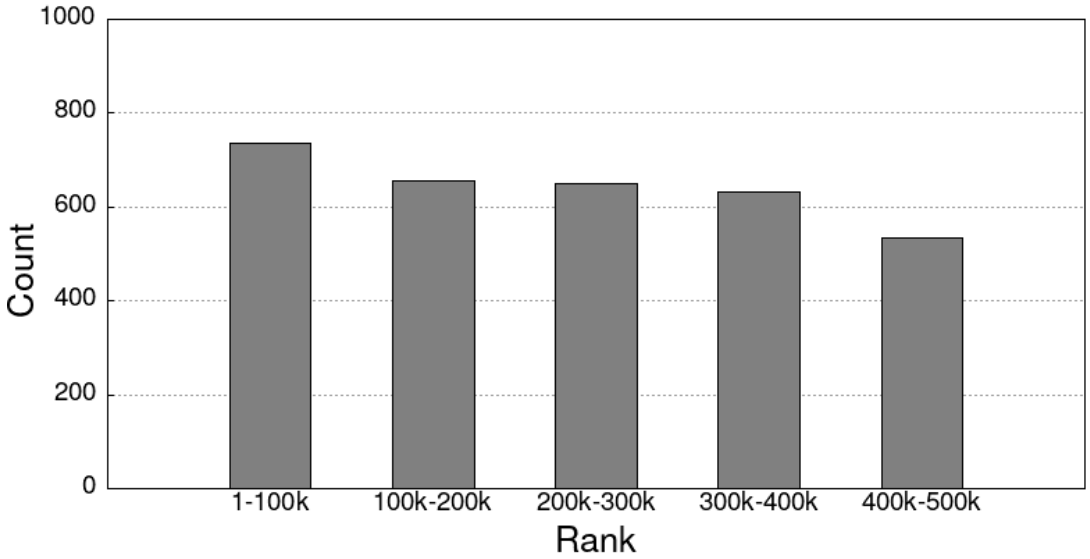

Figure 4.2: Amount of websites requesting permission to use web push directly.

Table 4.2: THIRD PARTY PROVIDER USAGE.

Third-party provider	Number of websites
aimtell.com	204
batch.com	258
cleverpush.com	479
foxpush.com	116
frizbit.com	348
getpushmonkey.com	12
gravitec.net	344
izooto.com	290
jeeng.com	88
killtarget.com	16
letreach.com	0
magicbell.com	4
najva.com	187
notix.co	234
onesignal.com	5679
panneaupocket.com	1
popify.app	1
push.world	37
push4site.com	44
pushalert.co	246
pushbird.com	4
pushe.ru	22
pushengage.com	488
pushly.com	436
pushnami.com	304
pushnews.eu	90
pushpad.xyz	84
pushpanda.io	356
pushpros.com	8
pushpushgo.com	172
pushtoast.com	4
pushwoosh.com	64
quickblox.com	3
subscribers.com	159
titanpush.com	2
truepush.com	215
vwo.com	235
webpushr.com	463
wonderpush.com	137

Chapter 5

Discussion and limitations

In this chapter, we will further discuss about the research, outline the limitations of this work and analyze possible future directions.

5.1 Realistic crawling

During our measurements, we detected some websites blocking our detection approaches or on which our approach failed. In particular, while our measurement tools rely on analyzing a large number of websites in a standardized way, some of those may behave differently when visited and browsed by a real user. First, running Chromium in headless mode may break some websites by affecting the rendering process or code execution, possibly leading to exclusion from the results. This may also be caused by websites detecting automated frameworks and refusing to load or load differently in such situations. Furthermore, due to time constraints, our approach defines and uses a time limit while rendering and analyzing the page which, while sufficient for the majority of websites, may not be enough for others. Finally, our approach only visits the home page of the target and does not follow links or complete any user action so it may miss websites that, for example, only install a service worker in different paths. Based on previous research [1] and our observations the number of such websites is relatively small and does not affect significantly our results which, for the above reasons, are considered a lower bound.

5.2 Post-authentication

Some websites, particularly when using personalized web push notifications, require user registration and login in order to enable the use of the technology. However, our

measurement framework does not automate this process and, in the case where websites install the service worker only after this action, will miss the website. While we found a large amount of websites that require login to enable web push notifications we found almost all install the service worker as soon as the user visits the home page, thus enabling our static approach to detect them. Future work may leverage a custom account creation and login script to detect more websites.

5.3 Ethics and disclosure

All experiments involving services that require a user account were tested with an account created ad-hoc, not interacting, targeting, or affecting any real user. Additionally, we disclosed all found vulnerabilities to the affected vendors and are currently waiting for a response. For issues related to third-party providers, the providers themselves were informed, and not the affected websites. Finally, we note that the issue we detected on Twitter regarding the wrong handling of sessions was fixed before our disclosure.

Chapter 6

Related Works

In this chapter, we discuss previous work in the field and related research on the vulnerabilities described in this thesis.

6.1 Service Workers

Service workers, as mentioned before, are quite a novel functionality whose security has not been properly tested yet. P. Papadopoulos et al. [12] analyzed their usage for malicious client-side computations, for example, crypto-mining, while G. Franken et al. [13] showed how SW-initiated requests are often not checked nor blocked by privacy extensions looking for third-party cookies. A persistent MITM attack was proposed by T. Watanabe et al. [14], which abuses the service workers fetch listener. By registering a service worker in the scope of a rehosting website, the attacker can manipulate requests and responses issued to the target website. J. Lee et al. [15] further analyzed service workers, especially in the context of progressive web applications, and found that their caching functionality can be used, in specific cases, to complete a history-sniffing attack. However, such an attack had unrealistic assumptions, such as requiring the victim to visit the website to install a service worker and revisit it while offline. Finally, M. Squarcina et al. [16] analyzed how XSS can be used to perform a MITM attack by abusing the Cache API of service workers to escalate the threat.

6.2 History Sniffing

Many articles have been published in the years about history sniffing, but new vectors to mount this attack are constantly found. A. Clover [17] in 2002, in one of the first articles

regarding this vulnerability, reported an exploit that used the visited style property for detecting whether a user previously visited a link. JavaScript code running in the browser could use the getComputedStyle method to check such property value. Since then, the vulnerability has gone a long way, with browser vendors adding more and more layers of privacy. Even with the added protections, novel attacks exploiting CSS properties have been constantly found during the years [18][19][20][21]. R. Kotcher et al. [22] showed how an attacker could reveal data such as text tokens by exploiting time differences in rendering different DOM trees. E. W. Felten and M. A. Schneider [23] analyzed the impact of web timing attacks as a tool for compromising user data, specifically their browsing history. An attacker could detect whether the user previously accessed such a resource by measuring the time needed to access data on third-party websites. In more recent times, S. Lee et al. [24] revealed that by abusing AppCache, a new functionality of HTML5, an attacker could detect the status of a target URL and use that information to detect previous logins on such website.

6.3 Push Notifications

As previously mentioned, research in the field of security of web push notifications is scarce. However, a few notable mentions do exist. J. Lee et al. [15] showed how this technology could be used to mount phishing attacks. Multiple websites were found using icons of well-known websites, such as YouTube and WhatsApp, to increase user interaction with their notifications. Moreover, at the time, Firefox desktop in some Linux versions, Firefox mobile in some instances, and Samsung Internet browser did not show the notification's origin domain, enabling websites to mimic notifications from other websites successfully. Finally, they found issues with third-party providers. First, the notifications are sent from the third-party subdomain, which leads to further confusion on the origin of those, and second, they proposed two ways for an attacker to steal the push subscription data. However, both attacks require the attacker to be able to intercept the requests between the victim and the provider, which, while not entirely unrealistic, is uncommon and usually not suitable for a large-scale attack. P. Loreti et al. [25] analyzed mobile push notifications and found that if an attacker can trigger a push notification to a victim and can passively sniff over a Wi-Fi network, it is possible to detect whether the victim is currently located on such network. This attack can be used either actively, by sniffing while the notification is sent, or passively, by sniffing and correlating such data with data from a notification sent later.

6.4 CSRF

Since the first appearance of the term in an e-mail by Peter Watkins in 2002 [26], the research on this vulnerability has led to many defenses and good practices to help websites

avoid this threat. X. Likaj et al. [27] analyzed such protections, their usage in the wild, and their effectiveness. The research pointed out how many web frameworks do not implement by default protections against CSRF and require either enabling it or installing another library to handle it correctly. This leads to websites using such frameworks being often not protected. L. Compagna et al. [28] researched the SameSite attribute as protection against CSRF and found how correct usage of such mechanism does protect in the majority of cases. Moreover, they built a list of guidelines and correct behaviors for the community to make such protection as strong as possible. However, they reported some cases where this protection does not work, and as such, it does not entirely substitute other mechanisms.

Chapter 7

Conclusions

In this thesis, we analyzed web push notifications, a novel technology that is being used more and more by websites. We explored the potential threats arising from the improper use of this technology and analyzed implementations found in the wild to analyze the prevalence of such issues. First, we analyzed the complexity required to successfully implement a system to send web push notifications to a specific user in response to certain events. We reported some dangerous privacy issues which could arise, such as websites possibly leaking private notifications. Then, regarding the same system, we found implementation issues in websites such as Twitter, potentially reducing the number of users of this system. An extensive analysis of third-party providers was then conducted, pointing out how the misuse of browser APIs, namely postMessage, is common and may lead to further vulnerabilities. We presented a novel history sniffing attack that uses such an issue to infer information about the user. Moreover, we found examples of how, even when an issue is not expected to exist, as is the case of Cleverpush while being used by direct inclusion of the service worker script in the customer's website, attempting to make development easier by standardizing workflows between two different approaches, can lead to the spreading of a vulnerability to a usually secure context. Then we showed another example of a CSRF vulnerability in the context of this technology, further proving the necessity of correctly protecting all websites from well-known issues.

Finally, we presented a large-scale measurement of the current usage of this technology and third-party providers. We analyzed the top 500K websites and found that about 0.6% of those websites ask the user for permission without presenting the user with an explanation for the necessity to use such technology, an approach proven to lead customers away. Of those websites, 1165 were found using a vulnerable third-party provider, exposing customers to privacy threats.

Overall this work revealed several threats in this relatively new browser feature, which we hope will incentivize further research in the field.

Appendix A

Logged API calls

Here are listed the API calls logged by the instrumented chromium alongside some notes where relevant to the research.

Table A.1: LIST OF LOGGED API CALLS.

API	Additional notes
Notification.permission	Property to check current notification permission status
Notification.requestPermission	Method to request notification permission
PushManager.subscribe	Method to request permission and then subscribe to push service
PushManager.getSubscription	Method to get existing subscription
PushManager.permissionState	
PushManager.getSubscription.Create	Method to create a new subscription
ServiceWorkerRegistrationPush.pushManager	
ServiceWorkerContainer.registerServiceWorker	
ServiceWorkerGlobalScope.importScripts	Method to import external scripts in a service worker
ServiceWorkerGlobalScope activate event	
ServiceWorkerGlobalScope backgroundfetchabort event	
ServiceWorkerGlobalScope backgroundfetchclick event	
ServiceWorkerGlobalScope backgroundfetchfail event	
ServiceWorkerGlobalScope backgroundfetchsuccess event	
ServiceWorkerGlobalScope canmakepayment event	
ServiceWorkerGlobalScope fetch event	
ServiceWorkerGlobalScope install event	
ServiceWorkerGlobalScope message event	
ServiceWorkerGlobalScope notificationclick event	
ServiceWorkerGlobalScope notificationclose event	
ServiceWorkerGlobalScope paymentrequest event	
ServiceWorkerGlobalScope sync event	
ServiceWorkerGlobalScope periodicsync event	
ServiceWorkerGlobalScope push event	Fired when a push notification is received
ServiceWorkerGlobalScope pushsubscriptionchange event	

Appendix B

Third-party detection

Here is provided the list of distinguishers used for each third-party provider.

Table B.1: LIST OF THIRD-PARTY DISTINGUISHERS.

third-party	Type	Distinguisher
aimtell.com	Service worker includes from	cdn.aimtell.com
batch.com	Path exists and contains proper data	/batchsdk-worker-loader.js
cleverpush.com	Web page source content	cleverpush.com
foxpush.com	Web page source content	foxpush
frizbit.com	Path exists and contains proper data	/FrizbitServiceWorker.js
getpushmonkey.com	Web page source content	pushmonkey.com
gravitec.net	Web page source content	cdn.gravitec.net
izooto.com	Path exists and contains proper data	/izooto.html
jeeng.com	Web page source content	jeeng.com
killtarget.com	Path exists and contains proper data	/kt-messaging.js
letreach.com	Web page source content	ltr-btn-allow
magicbell.com	Web page source content	magicbell
najva.com	Path exists and contains proper data	/najva-messaging-sw.js
notix.co	Path exists and contains proper data	/sw.enot.js
onesignal.com	Web page source content	OneSignalSDK
panneaupocket.com	Web page source content	app.panneaupocket.com
popify.app	Web page source content	popify.app
push.world	Web page source content	push.world
push4site.com	Web page source content	push4site
pushalert.co	Web page source content	cdn.pushalert.co
pushbird.com	Web page source content	cdn.pushbird.com
pushe.ru	Web page source content	pushe.co
pushengage.com	Service worker source content	pushengage.com
pushly.com	Web page source content	pushly

third-party	Type	Distinguisher
pushnami.com	Service worker source content	api.pushnami.com
pushnews.eu	Web page source content	pushnews or pn.vg
pushpad.xyz	Service worker source content	pushpad.xyz
pushpanda.io	Path exists and contains proper data	/PushPandaWorker.js
pushpros.com	Web page source content	pushpros.tech
pushpushgo.com	Web page source content	pushpushgo.com
pushtoast.com	Web page source content	pushtoast
pushwoosh.com	Web page source content	Pushwoosh
quickblox.com	Web page source content	quickblox
subscribers.com	Web page source content	cdn.subscribers.com
titanpush.com	Web page source content	titanpush
truepush.com	Web page source content	truepush
vwo.com	Web page source content	pushcrew
webpushr.com	Web page source content	webpushr
wonderpush.com	Web page source content	wonderpush.com

Bibliography

[1] Soroush Karami, Panagiotis Ilia, and Jason Polakis. «Awakening the Web's Sleeper Agents: Misusing Service Workers for Privacy Leakage». In: *In Proc. of the Network and Distributed System Security Symposium ({NDSS})* (February 2021).

[2] Zac Johnson. «How Progressive Web Apps Will Change Online Business». In: *Forbes* (2019). [Online; accessed 2023-03-14].

[3] Mozilla. *Service Worker API*. `https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API`. [Online; accessed 2023-03-14].

[4] Google. *Chrome 42 Release*. `https://chromereleases.googleblog.com/2015/04/stable-channel-update_14.html`. [Online; accessed 2023-03-14]. 2015.

[5] pushcrew. *The State of Web Push Notifications*. `https://gallery.mailchimp.com/fccee8d27b3a55c46b81ce8ae/files/The_State_of_Web_Push_Notifications_2016.pdf`. 2016.

[6] Mozilla. *Push API*. `https://developer.mozilla.org/en-US/docs/Web/API/Push_API`. [Online; accessed 2023-03-14].

[7] J. Hofmann. *Reducing Notification Permission Prompt Spam in Firefox*. `https://blog.nightly.mozilla.org/2019/04/01/reducing-notification-permission-prompt-spam-in-firefox/`. [Online; accessed 2023-03-27]. 2019.

[8] Mozilla. *Require user gestures for push notifications*. `https://bugzilla.mozilla.org/show_bug.cgi?id=1524619`. [Online; accessed 2023-03-27]. 2019.

[9] Apple. *Safari 12.1 Release Notes*. `https://developer.apple.com/documentation/safari-release-notes/safari-12_1-release-notes`. [Online; accessed 2023-03-27]. 2019.

[10] M. Tomson. *Generic Event Delivery Using HTTP Push*. RFC 8030. RFC Editor, Dec. 2016. URL: `https://www.rfc-editor.org/rfc/rfc8030.txt`.

[11] builtWith. *Push Notifications Usage Distribution in the Top 1 Million Sites*. `https://trends.builtwith.com/widgets/push-notifications`. [Online; accessed 2023-03-27]. 2023.

[12] Panagiotis Papadopoulos, Panagiotis Ilia, Michalis Polychronakis, Evangelos P. Markatos, Sotiris Ioannidis, and Giorgos Vasiliadis. «Master of Web Puppets: Abusing Web Browsers for Persistent and Stealthy Computation». In: 2019. DOI: 10.14722/ndss.2019.23070.

[13] Gertjan Franken, Tom Van Goethem, and Wouter Joosen. «Who left open the cookie jar? A comprehensive evaluation of third-party cookie policies». In: 2018.

[14] Takuya Watanabe, Eitaro Shioji, Mitsuaki Akiyama, and Tatsuya Mori. «Melting Pot of Origins: Compromising the Intermediary Web Services that Rehost Websites». In: 2020. DOI: 10.14722/ndss.2020.24140.

[15] Jiyeon Lee, Hayeon Kim, Junghwan Park, Insik Shin, and Sooel Son. «Pride and prejudice in progressive web apps: Abusing native app-like features in Web applications». In: 2018. DOI: 10.1145/3243734.3243867.

[16] Marco Squarcina, Stefano Calzavara, and Matteo Maffei. «The Remote on the Local: Exacerbating Web Attacks Via Service Workers Caches». In: Institute of Electrical and Electronics Engineers Inc., May 2021, pp. 432–443. ISBN: 9781728189345. DOI: 10.1109/SPW53761.2021.00062.

[17] Andrew Clover. *CSS visited pages disclosure*. https://lists.w3.org/Archives/Public/www-style/2002Feb/0039.html. [Online; accessed 2023-03-21]. 2002.

[18] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. «Scriptless attacks - Stealing the pie without touching the sill». In: 2012. DOI: 10.1145/2382196.2382276.

[19] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk. «Scriptless attacks: Stealing more pie without touching the sill». In: *Journal of Computer Security* 22 (4 2014). ISSN: 0926227X. DOI: 10.3233/JCS-130494.

[20] Craig Disselkoen, Shravan Narayan, Michael Smith, Deian Stefan, and Fraser Brown. «Browser history re : visited». In: *Usenix Security* (1 2018).

[21] Gilbert Wondracek, Thorsten Holz, Engin Kirda, and Christopher Kruegel. «A practical attack to de-anonymize social network users». In: 2010. DOI: 10.1109/SP.2010.21.

[22] Robert Kotcher, Yutong Pei, Pranjal Jumde, and Collin Jackson. «Cross-origin pixel stealing: Timing attacks using CSS filters». In: 2013. DOI: 10.1145/2508859.2516712.

[23] E. W. Felten and M. A. Schneider. «Timing attacks on web privacy». In: 2000. DOI: 10.1145/352600.352606.

[24] Sangho Lee, Hyungsub Kim, and Jong Kim. «Identifying Cross-origin Resource Status Using Application Cache». In: 2015. DOI: 10.14722/ndss.2015.23027.

[25] Pierpaolo Loreti, Lorenzo Bracciale, and Alberto Caponi. «Push attack: Binding virtual and real identities using mobile push notifications». In: *Future Internet* 10 (2 2018). ISSN: 19995903. DOI: 10.3390/fi10020013.

[26] Peter Watkins. *Cross-Site Request Forgeries*. http://web.archive.org/web/20020204142607/http://www.tux.org/~peterw/csrf.txt. [Online; accessed 2023-03-27]. 2001.

[27] Xhelal Likaj, Soheil Khodayari, and Giancarlo Pellegrino. «Where we stand (or Fall): An analysis of CSRF defenses in web frameworks». In: Association for Computing Machinery, Oct. 2021, pp. 370–385. ISBN: 9781450390583. DOI: 10.1145/3471621.3471846.

[28] Luca Compagna, Hugo Jonker, Johannes Krochewski, Benjamin Krumnow, and Merve Sahin. «A preliminary study on the adoption and effectiveness of SameSite cookies as a CSRF defence». In: Institute of Electrical and Electronics Engineers Inc., Sept. 2021. ISBN: 9781665410120. DOI: 10.1109/EuroSPW54576.2021.00012.