When Push Comes to Shove: Empirical Analysis of Web Push Implementations in the Wild

Alberto Carboneri
University of Illinois Chicago
acarbo4@uic.edu

Mohammad Ghasemisharif
University of Illinois Chicago
mghas2@uic.edu

Soroush Karami*
Paypal, Inc.
skarami@paypal.com

Jason Polakis
University of Illinois Chicago
polakis@uic.edu

ABSTRACT

Web push notifications are becoming an increasingly prevalent capability of modern web apps, intended to create a direct communication pipeline with users and increase user engagement. The seemingly straightforward functionality of push notifications obscures the complexities of the underlying design and implementation, which deviates from a near-universal practice in the web ecosystem: the ability to access an account (and the associated functionality) from practically *any* browser or device upon successful completion of the authentication process. Instead, push notifications create a communication endpoint for a *specific* browser instance. As a result, the challenges of deploying push notifications are further exacerbated due to the integration obstacles that arise from other aspects of web apps and user browsing behaviors (e.g., multi-device environments, account and session management). In this paper, we conduct an empirical analysis of push notification implementations in the wild, and identify common deployment pitfalls. We also demonstrate a series of attacks that target push notification functionality, including a novel subscription-sniffing attack, through a selection of use cases. To better understand current practices in push notifications implementations, we present a large-scale measurement of their deployment and also provide the first, to our knowledge, exploration and analysis of third-party service providers. Finally, we provide guidelines for developers and propose an approach for correctly handling push notifications in multi-browser, post-authentication settings.

1 INTRODUCTION

Web applications have rapidly grown in popularity over the last decade, as they improve the user experience by eliminating the need for manual software installation, and can seamlessly synchronize data between mobile and desktop platforms. Even common tasks that were previously conducted using native applications, such as sending emails, or editing documents and spreadsheets, are now predominantly performed through web applications (or "web apps"). This shift has driven browser vendors to continuously deploy new browser functionality to enable and ease the development of complex features and close the gap between traditional desktop applications and web apps. However, these features and their implementations often carry associated security and privacy risks [10, 29, 34, 37] which, if left unchecked, can expose users to severe threats.

For instance, Progressive Web Apps (PWAs) constitute a major evolution in web app capabilities and have garnered significant traction within the developer community. PWAs are powered by the Service Worker API (released in 2015), which allows websites to cache resources, better handle background sync operations, operate offline, and manage web push notifications. Service Workers operate by enabling websites to register an event-driven script that remains inactive until specific events occur, even after the originating website has been closed. Web Push notifications are an additional powerful capability intended for increasing user engagement by allowing websites to deliver custom messages to a target user (in actuality, they target a specific *browser* instance). Push notifications work differently from traditional communication channels, such as SMS and e-mails, as the message is sent directly to the user's device and shown as soon as it is online, resembling mobile and desktop notifications. This effectively reduces the time required for users to notice messages, resulting in an increased click rate. According to reports [39, 40], push notifications have been shown to outperform all other commonly used communication channels in terms of click rates. Moreover, web push notifications, unlike e-mails and SMS, allow for anonymous opt-in, potentially increasing the privacy of users.

The basic implementation of Web Push notification requires complex operations and multiple interactions between different systems. The inherent complexity of such systems creates the potential for implementation flaws that can result in security and privacy threats, and affected QoS deterioration that leads to user dissatisfaction or even revenue loss for vendors. While many websites have developed custom implementations of Web Push notification functionality, others rely on third-party providers as that allows them to sidestep the development complexities and overhead through a straightforward script inclusion. However, both approaches face challenges. Custom implementations are found throughout the web and a great number of popular websites have opted for this strategy. While this approach allows websites to have more control over the usage and handling of more complex scenarios, incorrect implementations pose a privacy threat to end users, especially when the notifications' content is sensitive (e.g., private messages). While third-party providers offer the same basic features with fewer implementation barriers to a wide range of websites, any vulnerability in their source code will impact a larger number of websites.

Prior work on PWAs has mainly focused on examining the security aspects of Service Workers' caching or background synchronization mechanisms [26, 38, 43], yet push notifications have received little scrutiny from the security community. Existing works explored

*Work was done while at the University of Illinois Chicago.

the correlation between Web Push notifications and phishing attacks [31], the de-anonymization threat posed by web push notifications [36], and the use of the technology to deliver malicious ad campaings [44]. However, while push notifications are *conceptually* straightforward, correct implementations pose various interesting challenges that have been overlooked, including correctly implementing third-party service providers, bounding push notifications to users' sessions and properly accounting for session changes (e.g., users logging out).

To bridge this gap, in this paper we provide an in-depth empirical analysis of potential vulnerabilities, and a large-scale measurement of Web Push notifications in the wild. Our work first presents an overview of challenges when incorporating push notification functionality in modern web apps, while also shedding light on the ecosystem of third-party service providers for websites that want to avoid implementing custom push notification capabilities. To study the current state of Web Push notification adoption and analyze its prevalence we develop an automated framework built on top of an instrumented Chromium. Our tool crawls websites and automatically extracts information related to push notification functionality intercepting all calls to relevant APIs, such as `PushManager` and `Notification` and analyzing the content of installed service workers. To better understand the implications of different attack scenarios and the current state of the web ecosystem, our tool needs to differentiate between custom and third-party push notification implementation. To that end we have created a set of heuristics that identifies the presence of a wide-range of third-party push notification providers.

Our empirical analysis highlights the complexities of properly integrating push notifications into a modern website and adhering to secure practices in regards to properly handling session-related changes. For instance, we find two major web apps (namely, Twitter and Poshmark) that, respectively, incorrectly handle multiple browsers, leading to push notifications not being sent in some situations, or incorrectly handle user logout, possibly leaking private data in the case, for example, of a shared computer. Accordingly, we propose an information workflow for bypassing pitfalls when deploying push notifications in authenticated contexts, providing a blueprint for web app developers to correctly handle such scenarios. Next, we introduce `Subscription Sniffing`, a novel variant of a history-sniffing attack that leverages websites improperly using of the `postMessage` API for inferring which websites a user has subscribed to for push notifications. We find that four of the 40 most prevalent third-party push notification providers are vulnerable to our attack. We also present a more severe variant of this attack, which also affects existing providers, that allows attackers to obtain more sensitive information (For example previous notifications sent to the user or notifications the user has previously clicked on) and also forcefully unsubscribe users from a target website. Next, we highlight an additional pitfall when deploying push notification functionality and the importance of integrating them with existing security mechanisms. Specifically, we present a case study of a website where the lack of the necessary anti-CSRF tokens allows attackers to silently hijack the victim's push notifications.

Finally, we conduct a large-scale measurement and analysis of web push notification in the wild. We find over 18,566 sites in the Tranco top 500K currently employ Web Push notifications. When taking into account that many additional websites may only expose push functionality post-authentication, which will be missed by our crawler, it is evident that push notifications are becoming increasingly common across the web. Notably, among those sites we find that 7,388 (40%) utilize custom implementations and 3,117 (42%) of those aggressively request push permissions from users. We also found at least 1,187 websites that are potentially vulnerable to our proposed subscription-sniffing attack, 479 of which are exposed to the more dangerous variant that allows attackers to obtain more sensitive information about the user or stealthily deleting the user's push subscription.

In summary, this research presents the following contributions:

- We conduct an empirical analysis of the security and privacy risks introduced by the adoption of web push notifications, an integral part of progressive web apps. We present and demonstrate a series of attacks that target push notifications, that pose significant privacy risks to users.
- We present a large-scale measurement of push notification adoption in the wild that highlights both the prevalence of push notification functionality, as well as the dependence of websites on third-party providers. Our analysis highlights the extent of the risks that users face, and a series of use cases that better illuminate the real-world implications of implementation flaws in web push notifications.
- We present a detailed workflow for personalized push notifications in multi-browser, post-authentication settings, as a blueprint for developers to avoid the pitfalls we uncovered.

2 BACKGROUND

In this section, we provide pertinent background information about Web Push Notifications (WPNs) that provides the necessary contextual information for the subsequent sections.

Push Notification. Generally, push notifications have been a feature commonly available in native applications, which have become available on the web as well in recent years. Such notifications appear in the form of pop-up messages sent by the browser to a user's device and typically include a title, message, image, and URL. Note that websites can send push notifications at any time, which will be delivered to the user's device even if the corresponding website is *not* currently open in the browser. WPNs are permission-based, meaning that users have to explicitly opt-in to receive them. There are two methods for requesting users' permission. In the first approach, a browser-based prompt will be shown to users and ask for permission of showing notifications. In the second approach, which is called "soft ask", the website will provide more information via a custom message before showing the browser prompt. For example, it may explain the value of receiving push notifications and if the user accepts the "soft ask", the website will proceed with displaying the browser prompt. Unlike other subscription and marketing methods, WPN provides an anonymous opt-in approach where users do not have to share their personal information, such as their name, email, or phone number to receive notifications. Instead, this communication channel is bound to the specific browser instance; as such, if the web service does not have information about the user's identity or account, this communication will essentially be anonymous.

Service Workers. Service workers are a special type of script installed by websites that operate independently in the background,

```
self.addEventListener('push', function(event) {
  var title = event.data.notification.title;
  var message = event.data.notification.message;
  var icon = event.data.notification.icon;
  event.waitUntil(
  self.registration.showNotification(title, {
    body: message,
    icon: icon
  }));});
```

Listing 1: An example use of the `push` event in service workers for receiving and handling push notifications.

separate from the DOM rendering process. Web push is built on service workers, allowing them to listen for push events and display incoming messages as notifications. Put simply, service workers enable websites to show notifications regardless of whether the website is currently open or not.

Push Services. WPNs work by maintaining a persistent connection to a push service provided by browser vendors (e.g., Mozilla's `autopush` [4] service for Firefox). Upon opening the browser, it contacts its push service and keeps the connection open via a WebSocket. When a user subscribes to notifications from a specific website, the push service creates a new "mailbox" which is a secret URL referred to as the `endpoint`. When a website attempts to send a notification, its server contacts the browser's push service by sending an HTTP POST request to the push service of the recipient's browser (i.e., `endpoint`). The push service then stores the data and delivers the new notifications when the browser establishes a connection. It's important to note that a single browser can have multiple endpoints simultaneously, each belonging to different websites. Moreover, the websites do not have any control over the push services and browsers can use any random push services. The push service APIs are standardized [7], which means websites are not required to know the underlying push service implementation.

Client-side implementation. There are two methods adopted by websites for requesting notification permissions. The first approach uses the `Notification` interface. The returned `promise` from calling `Notification.requestPermission()` includes the user's response to the permission request. Websites can check the status of current permission via the `Notification.permission` property, whose value can be `default`, `granted`, or `denied`. The second approach is to use the `PushManager` interface. The `PushManager.subscribe()` function subscribes the user to a push service. Before obtaining the subscription, it will verify the current notification permission status. If permission has not been granted, it prompts the user for permission through the browser prompt. Otherwise, it works with a push service to generate the subscription. Ultimately, this function returns a `promise` that resolves to a `PushSubscription` object containing all the information the application server needs for sending a push message to the user. A new push subscription is created if the current service worker does not have an existing subscription. This is all done in JavaScript using the Push API. An example using `PushSubscription` is provided in Listing 2. To protect the generated endpoint, a VAPID [45] key must be used. When the `PushManager.subscribe()` function is called, the corresponding public key must be passed as an argument. The generated subscription data is tied to that key, and can only be used in conjunction with the private key. This ensures confidentiality in the case the subscription data is stolen. Moreover, since service workers can

```
{
  "endpoint": "https://a-push-service.com/unique-id/",
  "keys": {
    "p256dh": "BNcRdreALRFXTkOOUHK1E ... e8QcYP7DkM=",
    "auth": "tBHItJI5svbpez7KI4CCXg=="
  }
}
```

Listing 2: In this push subscription example, the "endpoint" specifies the browser that will receive the notifications and the keys are being used for the cryptographic operations.

only be installed in an HTTPS context, websites must use HTTPS to be able to successfully send push notifications (or outsource push notifications to a third-party provider).

The service workers incorporate a `push` event listener to receive notifications. When a push message is received, a push event is triggered within the worker. This event is then directed to the `push` event listener, allowing the website to process the message and display a notification. Listing 1 shows an example using the `push` event listener. The `self.registration.showNotification()` function calls the operating system's notification API to show the notification.

Server-side implementation. Push services provide APIs that allow websites to send push notifications to their users. Using the APIs, the website specifies the recipient of the message, the message content and any relevant information which is then transmitted to the push service. The push service receives the data and validates it using the corresponding cryptographic keys. It then finds the appropriate browser using the `endpoint` and delivers the push message.

3 DEPLOYMENT & INTEGRATION PITFALLS

In this section, we first provide an overview of the basic information workflow of WPNs. Subsequently, we discuss deployment and integration pitfalls, based on the findings from our empirical analysis of existing custom implementations in popular websites, and the code and documentation of third-party push notification service providers. Specifically, we focus on the challenges of relying on third-party service providers for handling notification functionality, as well as the additional complexity of correctly integrating push notifications into web apps' authenticated functionality.

Basic deployment. Websites can use the Push and Notification APIs to enable the delivery of WPNs. The Push API allows web applications to receive messages pushed from a server, regardless of whether the web app is currently active or loaded in the browser. This empowers developers to deliver asynchronous notifications and updates to their subscribers. The Notifications API also allows web apps to control the display of system notifications to the end user. Since these are outside of the top-level browsing context viewport they can be displayed even when the user has switched tabs or moved to a different page.

Figure 1 illustrates information flow in Web Push: ❶ the webpage requests Notification permission through the browser-specific prompt and ❷ if the user grants permission, the Push Service generates a Push Subscription for the browser. After obtaining a Push Subscription, ❸ the webpage will forward it to its backend server, which then stores the subscription in a database. This stored subscription allows the server to send push messages to the respective user at a later time. When the server intends to send a push message to its users,

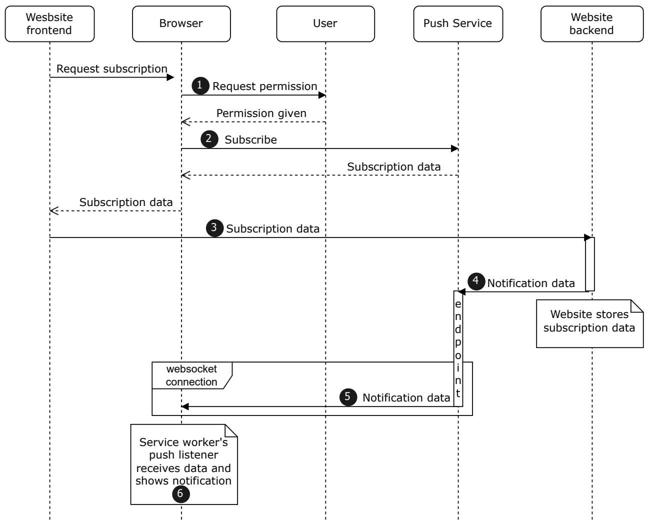

Figure 1: The workflow of Web Push communication between user's browser and server. The webpage uses a JavaScript API to get the push subscription information and send them to the server.

❹ it initiates an API call to a specific endpoint of the push service. The endpoint information and the required authentication keys are included in the subscription data as shown in Listing 2. This API call includes details regarding the data to be sent, the recipient of the message, and any specific criteria regarding the delivery of the message.

When a push service receives a network request, ❺ it verifies its authenticity and delivers a push message to the intended browser. If the browser is offline, the message is stored in a queue until the user's device becomes online and the push service can deliver the messages. When a message is successfully delivered, the browser receives and decrypts the data, triggering a push event in the service worker. Finally, ❻ the service worker uses the Notification API to display the notification to the user.

3.1 Integrating Third-Party Push Providers

WPNs were initially introduced for sending first-party messages. However, to facilitate adoption, WPNs can be deployed by integrating third-parties that provide this feature as a service. Service Providers facilitate the process of integrating web push to the website by providing a simple script to be served by the websites. After including the script, the website owners can use the service provider's dashboard to compose messages (including title, message, image, and URL) and send notifications. There are two main approaches for integrating this service. In the first approach, service providers create service worker scripts for their customers and instruct them to *host* these scripts. Additionally, they request website owners to include a script tag that registers the service worker and manages user subscriptions.

However, there are some caveats associated with this approach that can pose challenges for customers. First, if the website currently uses service workers, it may create a conflict with the provider's script. However, this can be managed by registering the service worker on a different scope. Second, the websites' contents must be served via HTTPS to be able to use such services. To expand their customer base, service providers employ an alternative approach that circumvents these limitations. For each customer, they create a third-party domain that is served via an HTTPS protocol. They use this page to request permission and subscribe to notifications. Since the origins are different, there will be no conflicts for service workers. Under this approach, when a user decides to subscribe to push notifications, a pop-up window will be launched to install a service worker, use the Push API and manage the push subscription. Next, the service providers create a unique URL for a script, which is hosted on their servers, and instruct the website's owner to include it. This script verifies the user's notification subscription status, and if not subscribed, it will display a subscribe button which opens the above-mentioned pop-up window to ask for permission.

As the subscription is completed on a third-party domain, no cookie can be saved on the customer's website to mark a successful subscription. The script thereby loads a specific page of the third-party domain into an `iFrame`, which uses the `postMessage` API to return the current subscription status. In the following section, we examine the potential security risks that may arise from these design decisions and their corresponding implementations.

3.2 Implementing Personalized WPNs

Push notifications serve as a direct communication channel between web applications and users. Web applications use push notifications to inform users about a set of events that users may find important, and certain notifications may contain sensitive information which is tailored to individual users. For instance, email users (e.g., Gmail) may receive notification messages containing the sender's name and email title when they receive an email. In these scenarios, the implementation of WPNs requires careful consideration, as compositionality issues can arise when other security mechanisms need to be integrated into the WPN pipeline.

Session management. As discussed earlier, when a user logs into a website, a session ID is assigned to the user. The session ID has a limited lifespan and gets revoked when the user logs out from the web application or expires after a certain duration. In either case, the server must stop sending notifications to the user. However, if the server does not verify the validity of the session for each endpoint, or does not correctly invalidate sessions on the server side, it will send notifications to users that are already logged out. In a correct design, only the users with a valid session who have subscribed should receive the notifications. To test the correctness of an implementation, we consider the following rules, which capture pitfalls that can occur when implementing personalized WPNs :

(1) Having multiple active sessions should not impact correctly receiving notifications.
(2) Even after logging out and logging back in, the user should continue to receive notifications.
(3) The notifications should be received correctly on active sessions when there are both active and inactive sessions.

4 EXPLOITING IMPLEMENTATION PITFALLS

As discussed in Section 3, developers have various options for implementing push notifications, and we highlighted various flaws that can occur during the integration and deployment process. In this section, we continue our empirical analysis and further explore the security and privacy risks posed by this technology. To that end, we first detail the threat model that will guide our analysis. Next, we present two different attacks that target WPNs. Initially, we introduce a novel history-sniffing attack, and then provide an in-depth analysis of the security implications of CSRF in WPN deployments.

Threat model. For our empirical analysis, we consider two different attacker scenarios. First, we employ the *web attacker* threat model, which is typical for studies exploring web technologies. Specifically, we assume that the victim visits a website under the control of the attacker, who is then able to execute JavaScript code in the user's browser. Nonetheless, several of the attacks explored in this work could be executed at a larger scale through other attack vectors (e.g., ads). Additionally, in recent years the security community has increasingly studied intimate partner violence and highlighted how abusers exploit technology [20]. As such, due to the idiosyncrasies of WPNs (which use specific browser instances as communication endpoints) we also consider a malicious individual that can obtain physical access to the victim's browser when studying push notifications in conjunction with session management.

Subscription sniffing. History sniffing attacks, in which attackers cross-check a list of target websites against the user's browser to (partially) infer the user's browsing history, pose a severe privacy threat as they can reveal a plethora of sensitive user information (e.g., age, gender, sexual orientation, medical issues, and political affiliation). While websites cannot directly access the browsing history, prior research has demonstrated a wide variety of techniques for inferring visited websites via side-channel attacks [48] or abusing certain browser features [26, 42].

Here we present subscription sniffing, a novel variant of history-sniffing attacks that targets third-party push notification providers. To illustrate the attack, Figure 2 outlines the information flow of a push notification service when it is "outsourced" to a third-party provider. In this example, the website uses an `iFrame` to load the third-party page ② and check whether the user is subscribed. This information is saved into a cookie on the third-party domain. The `iFrame` then checks the presence of the cookie ③ and sends back the information using the `postMessage` method ④. The first-party then registers an event listener to receive the information and if the user is not subscribed yet, it opens the third-party domain as a pop-up to ask for the necessary permission ⑤ - ⑨.

In this method, each website has a unique domain for providing the third-party notification service. A common mistake in using the `postMessage` method is not restricting the target origin of the sent messages. Specifically, as an `iFrame` may not be able to correctly identify the including origin, the `postMessage` API requires a second parameter that specifies which origins should be allowed to receive the sent message. The browser will then deliver the message only to the correct destination. However, a value of "*" is allowed, mainly for development, which allows any origin to receive the sent message. If this is the case, an attacker can also use the same `iFrame` ⑪ to check whether the user has subscribed to notifications of a specific website by listening to the sent messages ⑬. By finding the list of websites that a user has subscribed to, the attacker can perform a subscription-sniffing attack and infer sensitive user information.

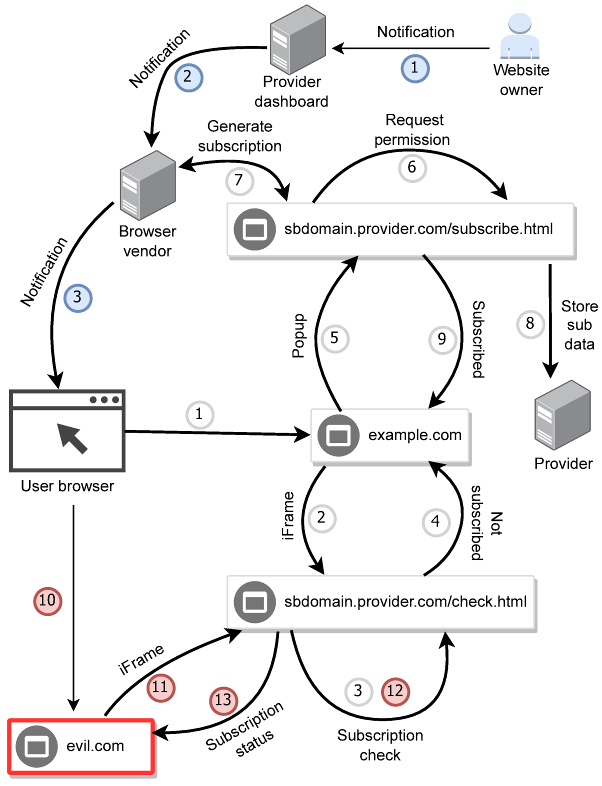

Figure 2: Subscription-sniffing attack against third-party WPN providers.

CSRF. Cross-Site Request Forgery [28] attacks remain a severe threat that can manifests in different contexts of web app functionality [11, 13, 16, 27, 50]. Here we explore the implications of such an attack in the context of push notifications. As discussed previously, after obtaining the `PushSubscription` the website will send the necessary information to the server. Since the request sent to the server carries the user's cookies, the server will be able to match it to the corresponding user and store the subscription information. For simplicity we assume that the web app does not employ any form of CSRF protection [12]; nonetheless, prior work has demonstrated methods for bypassing CSRF protections [50]. Once the user visits a website controlled by the attacker, the attacker's page can "force" the victim's browser into generating a request with the attacker's `PushSubscription`, which carries all the necessary cookies for the server to consider this a valid request actually issued by the user. This will result in the server storing the attacker's `PushSubscription` instead of the victim's, and the web app sending all subsequent notifications intended for the user to the attacker. It is important to emphasize that this attack is completely "silent" and the victim does not even need to grant the notification permission. In essence, the server receives a seemingly valid `PushSubscription` from the request and will store the attacker's associated endpoint for sending notifications. The result of the attack in the situation when the user is already subscribed depends on the underlying web application.

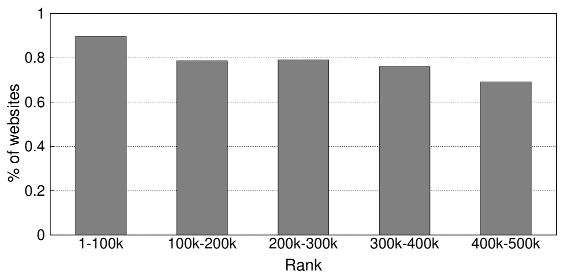

Figure 3: Percentage of sites requesting direct permission to send WPNs.

The legitimate user subscription may get overwritten, especially in the case when the attack is executed on the same browser where the subscription was made from, or it may simply be added into a list.

5 EXPERIMENTS AND MEASUREMENTS

In this section, we present our large-scale measurement study on the prevalence and state of WPNs in the wild. We also present a series of case studies that highlight the pitfalls of deploying push notification functionality and demonstrate the impact of our attacks.

5.1 Large-Scale Measurement

Collection methodology. To collect the data related to push notifications, we instrumented Chromium to capture all calls to `ServiceWorkerContainer`, `ServiceWorkerGlobalScope`, `ServiceWorkerRegistrationPush`, `Notification`, and `PushManager` APIs alongside all events dispatched inside a service worker such as `register`, `install`, `sync`, and `fetch`. The instrumentation was made at the browser level by patching the chromium source code to log relevant data during calls to the above APIs. Moreover, to enable the automatic detection of websites requesting notification permission directly, we also introduced a Chromium patch to automatically accept notification requests after a short amount of time. This was achieved by editing the flow used to trigger the permission popup. We use dynamic and static approaches for detecting custom implementation of push notifications. By analyzing API call logs, particularly by examining the occurrence of `Notification.requestPermission()` or `pushManager.subscribe()` function calls, which are necessary for obtaining user permission for notifications, we can dynamically identify websites that invoke these functions to acquire notification permission and send notifications. However, there are cases where the dynamic approach cannot capture the APIs, particularly when these APIs require user interaction to be invoked. For instance, a user has to visit the website's settings and explicitly ask the website to send them notifications. This action triggers the notifications permission and invokes the APIs. Therefore, we also developed an approach to statically analyze the service worker scripts and determine whether a website uses the push API. Typically, websites have only one main service worker script that may include multiple scripts. In our static approach, we use the list of APIs to recursively download all scripts and analyze them. We use a straightforward keyword-matching technique on the source code of the service workers to identify relevant APIs. To construct the list of keywords, we performed a smaller experiment using the dynamic approach until we obtained a list of 500 websites using WPN, examined their service workers and manually extracted relevant keywords from the source code to minimize the occurrence of false negatives to zero. The list of obtained keywords contains common strings such as 'Notificationclick', 'handlePushEvent', and 'push' events registration in 'addEventListener'.

Data collection. We used Puppeteer [6] and the instrumented browser to visit the top 500k Tranco websites and utilized the aforementioned approaches to obtain information about the website's push notifications. For each website, we open a new browser instance waiting for the page to load (we wait a maximum of 60 seconds or until no network requests occur for at least 500ms); after that we wait for an additional 30 seconds to ensure that everything has been fetched and loaded. To quantify the prevalence of third-party providers, and the associated flaws, we took the top 40 providers (according to builtWith [8]) and built a tool for determining whether a website utilizes a particular provider. The tool loads the website, as aforementioned, and then detects the use of providers through custom-tailored heuristics, such as by examining the DOM content for domains and tags present in the scripts distributed by the providers or verifying the existence of a specific service worker in a specific path which is required in order for that provider to function.

Our findings are reported in Table 1. The table differentiates between websites detected by our dynamic, static, and third-party detection approaches. We note that the total number of websites reported is greater than the total as some websites were flagged by more than one approach. In total, both our scripts successfully run on 396,154 websites. We discovered 18,566 (4.68%) unique websites that use push notifications. Among them, 3,117 (0.78%) were found to request permission immediately upon a user visiting the page. A breakdown of those websites, where each bucket is normalized based on the number of sites analyzed within the given bucked so as to avoid bias, is shown in Figure 3. While we find such invasive practices slightly more often in popular websites, their prevalence is comparable across all ranks. Next, we explore the prevalence of custom implementations, as shown in Figure 4. Surprisingly, we find that relying on third-party WPN service providers is common practice regardless of a website's popularity as, overall, 11,178 (2.82%) websites employed a third-party provider while 7,388 (1.86%) had a custom implementation. As discussed in §3, the complexity of implementing WPNs may incentivize websites to employ a service provider. Despite the obvious benefits, however, any flaws in the third-party's implementation will result in widespread vulnerabilities across the web ecosystem. Overall, our script failed to run on 103,846 websites mainly due to missing DNS records or anti-bot features detecting and blocking our automated framework..

Manual validation. Next, we conduct a manual analysis on a subset of the websites, to verify the accuracy of our heuristics. The dynamic approach does not have any false positives (websites not using WPN but being flagged as so), but it may have false negatives (websites using WPN but not being flagged). We note that even though the static approach helps reduce false negatives, it may still fail to detect websites with a heavily obfuscated service worker code. Both approaches are unable to detect websites that install a service worker only after a user interaction has occurred as both relies on API calls made during or after the service worker installation. To verify our third-party detection approach, we collected at most 10 websites per provider

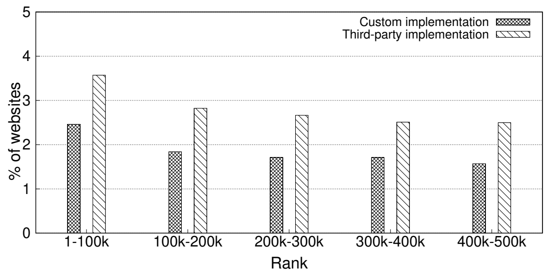

Figure 4: Custom and third-party implementation usage, grouped by global rank.

Table 1: Breakdown of WPN detection per technique.

Technique	Websites
Dynamic approach	3,117
Static approach	8,750
Third-party detection	11,178

Table 2: False negatives in third-party provider detection.

Issue	Websites
User action required	12
Cloudflare anti-bot detection	3
Code obfuscation	1

from builtWith which we manually classified and checked to be loading correctly. We note that for some providers we were unable to find 10 different valid websites and for this reason the total number of analyzed websites is lower. We also randomly selected 20 websites from the Tranco top 500k where our system did not detect push notifications. In total, we manually classified 343 websites and compared our findings with the ones obtained from our system. For our third-party detection approach we found 5 (1.4%) false positives, all due to leftover code in the website, and 16 (4.7%) false negatives, which can be attributed to factors such as code obfuscation, websites blocking our automated framework, or situations where user interaction is necessary. The number of websites affected per issue are reported in Table 2. Overall, our system reported a 5.6% FPR and a 5.8% FNR.

5.2 Case Studies

To check a website in an automated manner, we should be able to log in and properly evaluate the efficacy of the notification system. However, we have opted for a manual process due to the various challenges of automating the testing pipeline. Apart from the difficulty of automating the account-creation and authentication process, enabling WPNs may be hidden within an account's settings. More importantly, even if those challenges are overcome, actually triggering push notifications can require site-specific actions (e.g., another account sending a message) or only occur periodically when triggered by the server. As such, here we present a series of use cases that detail vulnerabilities and flaws that we uncovered during our empirical analysis, highlighting the challenging nature of deploying WPNs and the threat posed by the threats we present in our work. A summary of the findings of our case study and vulnerable websites and service providers is provided in Table 3. Note that in this table the entry 'Session Mismanagement' includes informations about both 'Session termination' and 'Session management' issues.

Session termination. As mentioned in Section 4, if websites do not verify session validity, personalized notifications can be received even when the user is logged out. During our empirical analysis we manual inspected `poshmark.com`, a popular social commerce marketplace for clothing and accessories (Ranked 1,703 on Tranco). While Poshmark has incorporated WPNs into their authenticated workflow and send account-specific notifications, they do not correctly handle session changes. Specifically, they continue to send personalized notifications even after the user has logged out. This can reveal sensitive information to *other* individuals that gain or have access to that browser. We further discuss the conceptual disconnect that exists between typical account-functionality and WPNs in §6.

Session Management. Upon comprehensive testing of Twitter's behavior, we discovered that there are certain scenarios where their push notification service does not function as intended due to flaws in integrating WPNs into their authentication workflow and correctly handling session changes. First, if users log out and subsequently log in again, they will no longer receive notifications. In fact, users need to repeat their push subscription to be able to get notifications, indicating that session terminations in Twitter lead to WPN push subscription termination. Second, If a user logs in from two separate machines that have the same OS (regardless of its version) and use the same browser vendor (regardless of its version), Twitter will send notifications only to the most recent browser that logged in. To better illustrate this, suppose that a user has two valid sessions on two devices: DeviceA running macOS v12 and using Chrome v109, and DeviceB running macOS v13 and Chrome v114. The user has also subscribed to Twitter notifications from both devices, and DeviceA was the last to subscribe to Twitter. In that case, the user will only receive notifications on DeviceA.

Upon further experimentation with Twitter's notification system, we deduced that it likely stores subscriptions in a custom table, alongside a sessionID, browser vendor, and OS. When a user subscribes to notifications, the system checks if the user's OS and browser combination already exists in the table. If it does not, a new record is inserted. However, if the OS and browser combination matches an existing record, the system updates the session and subscription fields of that record with the new information. In other words, Twitter infers the endpoint based on sessionID, OS and browser fields. As a result, if the user uses a similar OS and browser on another system, Twitter will lose track of the previous subscription information and just push the notifications to the latest.

A correct design must validate sessions before sending notifications, which would prevent sending them to endpoints with invalid sessions. Additionally, Twitter does not update the corresponding sessionIDs when users get a new one (e.g., logging out and logging in). As a result, users are compelled to resubscribe to notifications after each login. Overall, the issues that we have uncovered in Twitter highlight the true complexity of incorporating WPNs in authenticated environments where session management needs to be incorporated into the life cycle of notification management. Moreover, the presence of such issues on a site as popular as Twitter, is a good indication that such flaws may also affect many other services.

Table 3: Example of push notification vulnerabilities discovered in various websites.

Flaws	Service
Session Mismanagement	Twitter, Poshmark
Subscription Sniffing	Webpushr, LetReach, Cleverpush, Pushalert, VWO
CSRF	gama.ir

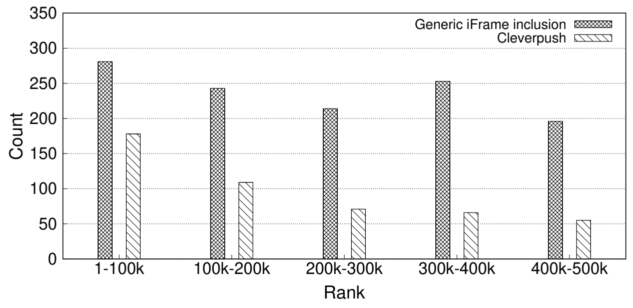

Figure 5: Rank breakdown of websites affected by the iFrame inclusion vulnerability and the Cleverpush flaw.

Subscription sniffing. LetReach [3] is a push notification provider that uses the aforementioned `iFrame` method for subscribing users. The `iFrame` uses the `postMessage` method to inform the first-party website whether the user is subscribed for receiving notifications. Since the `iFrame` should only be loaded by the original website, the messages should only be sent to that specific origin. However, LetReach incorrectly sets the `targetOrigin` of the `postMessage` to "`*`", thereby enabling any website to receive these messages. Consequently, an adversary can exploit the absence of proper destination validation and obtain the list of websites to which the user has subscribed by including the `iFrames` into a website they are in control of and registering an event listener for messages sent by the third-party domain. This attack can also target a specific customer (i.e., website) since each website that uses this functionality has a custom subdomain generated by the provider (Section 3).

Additionally, these messages contain a unique identifier for the user's subscription if the user has accepted receiving notifications on the target website, which can potentially reveal additional information about the victim to the attacker. Our in-depth analysis of the JavaScript included in the `iFrames` revealed numerous endpoints for managing the user's subscription. There is also an unsubscribe endpoint that accepts unique identifiers sent by the `iFrame` to the original page as a parameter, potentially allowing an attacker to unsubscribe users. However, at the time of writing, that endpoint does not work correctly even for legitimate uses. Several other providers, including Webpushr [9] (the third most commonly used according to BuiltWith), have the same flaw in their implementation. However, unlike LetReach, most providers only offer subscriptions through a subdomain upon explicit request, resulting in potentially fewer websites being vulnerable to this issue.

Cleverpush [2], another notification provider, is also vulnerable to this issue and includes an `iFrame` for all websites, even when HTTPS is directly supported on the customer's domain. Moreover, the JavaScript code loaded within the `iFrame` exposes a `postMessage` API that lacks the necessary origin checks. An attacker can include an `iFrame` in their website and gain access to information such as whether the user has visited and accepted notifications, the internal subscription ID, a list of notifications the user has interacted with, and even forcibly unsubscribe the user from push notifications. Most importantly, the unsubscribing process is never revealed to the target website, which will not prompt the user again for permission to send push notifications unless the browser data associated with both the target and the `iFrame` origin is deleted. In total, we identified 1,187 websites that could potentially be vulnerable to this history-sniffing attack and we discovered 479 websites affected by the vulnerability present in Cleverpush.

An analysis of third-party providers affected by this vulnerability, whose results are shown in Figure 5, revealed that, while the generic issue is almost equally distributed between websites' ranks, Cleverpush is mostly used by high-ranking websites, leading to more users being affected. All those websites are mainly categorized as "General News", "Blogs/Wiki", "Entertainment", and "Business", and more than 30 websites classified as "Finance/Banking".

CSRF. During our manual analysis, we found that `gama.ir` does not include protection against CSRF attacks on the subscription endpoint. Gama is an Iranian website for teachers to share and publish educational content, including exam questions. When a user subscribes to the push notifications of Gama, no protection against CSRF is implemented. An attacker can deceive the victim into visiting their website, where they can add their subscription to the victim's account and gain access to their notifications.

6 AUTHENTICATED WORKFLOW DESIGN

Conceptually, WPNs constitute a straightforward communication mechanism between two endpoints, i.e., the web server and the user's browser. However, as discussed and demonstrated in the previous sections, various flaws can occur during implementation, as well as during the integration of third-party service providers. Our empirical analysis revealed that the most complex aspect is correctly handling WPNs in conjunction with session management. Web applications use push notifications to inform users about a set of events that users may find important or interesting, and some may contain sensitive information tailored to the individual user. For instance, email providers may push notification messages containing the sender's name and email title. In these scenarios, the implementation of WPNs requires careful consideration. To that end, this section explores various strategies and possible approaches for designing a custom push notification system for settings where authentications is required.

Prior research has shown that developers struggle with integrating additional mechanisms into their authentication workflows and correctly handling session changes [21]. Similarly, incorporating push notification functionality into a web app's post-authentication workflow introduces considerable complexity as it combines two different communication abstractions: that of a user's account and that of a user's browser. To put it differently, account-specific functionality and data are not tied to a specific device and are available to *any* browser running on *any* device once the authentication process completes successfully (i.e., the user is able to log into their account). On the other hand, push notification functionality is tied to the *specific* browser instance from which the user subscribed, and is not available to any other browsers or devices from which the user may log in. As such, the workflow of push notifications in a post-authentication setting is significantly more complicated. To that end, here we present a workflow for handling notifications for authenticated users, and also provide additional implementation-related details that will allow developers to sidestep deployment pitfalls and correctly incorporate push notifications into their apps.

Figure 1 provides a detailed overview of the workflow for an authenticated notification system. The server has to store the information representing the relationship between the Push Subscription and the user account which can be a table in the database. This enables the server to find the recipient of the notification once an event is triggered. It may appear straightforward to assume that we can simply store this information by creating a table where each row of the table stores the user_id and the PushSubscription. However, a user can have multiple active sessions at the same time on the same or even different devices. In this case, we may need to have multiple rows for the same user_id to store different PushSubscriptions. Once there is an event for this user, the server has to locate the information for that specific user and send the notification to all of them.

While this approach may seem correct, it does not factor in a case in which a user logs out from only one session or when only one session expires while another sessions remain active. The discussed design does not have any mechanism to consider such cases which will result in sending notification messages to all sessions even when they are inactive. An alternative approach is to store the session IDs, which are unique identifiers assigned to each session, in addition to the notification information. Once a push event is triggered, the server will query the table and only sends the notifications to the endpoints that have a valid session ID. However, including session IDs adds more complexity to the process where the servers must update the table every time the session ID is renewed. For example, if a user logs in and out of her account from the same browser, the session ID will change whereas the `PushSubscription` will remain intact. Therefore, the server needs to know what specific row belongs to this specific browser and update the row by replacing the previous session ID with the new one.

Figure 6 depicts the correct workflow for managing personalized WPNs in the event of a logout or session expiration. In the top section ①→⑦, the same user subscribes to the push service on two different browsers. The notification server ② sends back the new endpoint that is generated for the website. When the website receives this information, ⑥ it will send it back to its server along with the cookies and a proper browser fingerprint. The server utilizes the session ID cookies to find the `user_id` and then ⑦ stores them in a database next to the PushSubscription information. In the bottom section, the workflow demonstrates ⑧ the initial triggering of an event. The server ⑨ ⑩ locates the valid endpoints from the database and ⑪ ⑫ sends the message to those endpoints through the browser's vendor servers. The endpoints, which are managed by the browser vendor, will contact the browser and send the notification to the user. Then ⑬ the user logs out from the first browser. The session invalidation ⑭ must be correctly updated in the database so that a ⑮ future notification ⑯→⑱ will only be sent to the still logged-in browsers.

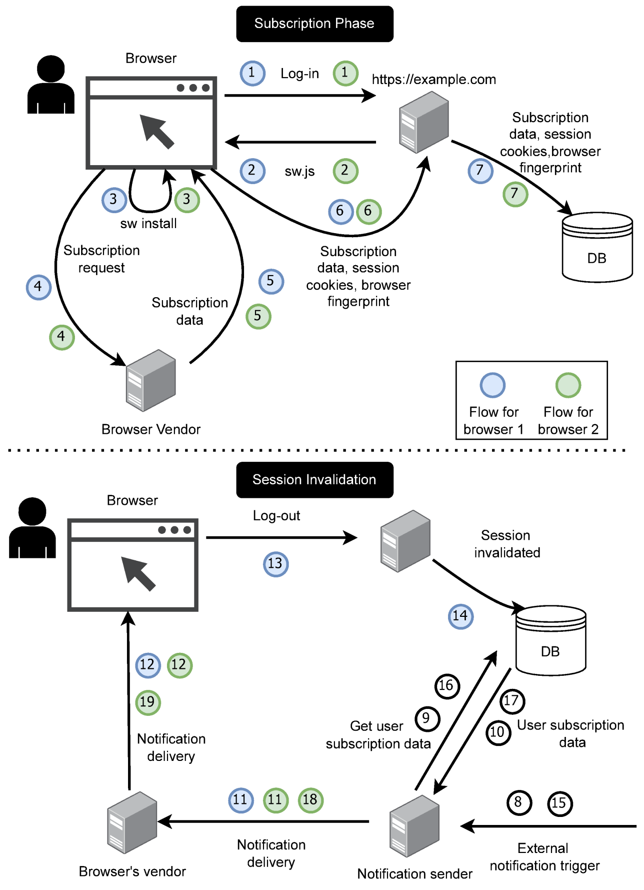

Figure 6: Server stores the push subscription data in a table, along with the cookie and a browser identifier. Once an event is dispatched, this data is used to determine the endpoints with valid sessions to which the notification should be sent.

We note that the workflow proposed here is a guideline for implementing subscription management in multi-browser authenticated environments, highlighting the nuances and pitfalls that can occur. Designing such a system is not a straightforward task, and developers may need to adapt their design to better match the intricacies of their web app architecture and information workflow.

7 DISCUSSION AND LIMITATIONS

In section we further discuss our research, outline the limitations of our work and detail possible future directions.

Realistic crawling. Recent studies have studied the countermeasures deployed by web services for mitigating bot traffic, and explored strategies for automated crawlers to more realistically resemble actual user traffic [25, 49]. As such, while our measurements rely on the processing of a large number of websites, in practice certain websites may behave differently when being visited by our tool. First, running Chromium in *headless* mode may affect code execution and page rendering in certain websites, potentially causing some websites being inadvertently excluded or overlooked. This also includes websites that detect and respond differently to automated browsers. Furthermore, our approach sets a finite time limit for the website to load, so as to reduce overhead in our large-scale measurement study, which although sufficient for most websites, may not be adequate for others (particularly slower websites). Since our system only visits home pages on websites, it may overlook cases where websites install service workers *only* on internal pages rather than the home page. Additionally, our framework does not emulate any user actions and thus if the tested features require user interaction, our framework cannot capture its presence on those websites. Based on our experimental observation and previous research [26], the number of websites that require either visiting additional pages or requires user action to install a service worker is relatively small, and therefore, does not significantly affect our studies findings; nonetheless, we consider our numbers to be a lower bound. An interesting future direction would be a measurement study that employs a more realistic browsing environment and user interactions.

Post-authentication measurements. We note that websites may require account creation for enabling push notifications, especially for personalized notifications. However, our system does not automate account creation or login. While we found a large number of websites that only allow subscribing to notifications after logging in, we observed that in almost all of those cases the website installed the required service worker on the landing page, thus enabling our static approach detecting them. Additionally, our empirical analysis also relies on extensive manual experiments. Nonetheless, future work could leverage an automated account creation and authentication framework (e.g., [14, 17]) for conducting a more comprehensive post-authentication analysis.

Automated vulnerability discovery. Our tool cannot automatically assess whether websites suffer from some of the flaws that we discovered manually (i.e., on Twitter and Poshmark). This is primarily due to the multitude of distinct behaviors that would need to be tested and the complexity involved in developing a system that can automatically trigger a notification on an unknown website. We also note that detecting websites that are vulnerable to CSRF attacks on their subscription endpoint is outside the scope of this work. Nonetheless, multiple vulnerability scanners already exist for such tasks (e.g., ZAP [5], w3af [1]), and recent academic studies have also proposed systems for detecting CSRF vulnerabilities [13, 27].

Ethics and disclosure. All of the experiments with web apps and third-party service providers that required a user account, were completed using test accounts. We did not interact, target, or affect any actual users. Additionally, due to the significant implications of our findings, we have disclosed them to all of the affected websites and third-party providers via dedicated channels when present or customer support, and are currently waiting for a response. Specifically, we disclosed the discovered subscription sniffing vulnerability to Webpushr, LetReach, Cleverpush, Pushalert, and VWO and notified Twitter, Poshmark, and gama.ir regarding the issues we found on their respective websites. At the time of writing, only VWO and Cleverpush have acknowledged the issues and have informed us that they are actively working on a fix. Moreover, we note that the issue that we uncovered in Twitter regarding the mishandling of session changes was fixed prior to our disclosure; the second reported issue, which allows for breaking the notification system by enabling it on two computers with the same OS and browser vendor, is still present at the time of this writing.

8 RELATED WORK

Service Workers. Service workers are quite a novel functionality in modern browsers, and while their security has been tested, further research is required to fully understand the related issues. Papadopoulos et al. [38] analyzed their usage for stealthy cryptomining on client-side applications. Franken et al. [19] revealed how requests initiated by service workers are almost never checked and blocked by browser privacy extensions when carrying third-party cookies, possibly allowing tracing of the user even in such conditions. An novel, persistent, MITM attack was proposed by Watanabe et al. [46]. This attack abuses the `fetch` event listener in the context of a rehosting website and can manipulate requests and responses to such website. Squarcina et al. [43] analyzed their implications in escalating XSS attacks and found an attack vector through the Cache API which allow for performing a MITM attack.

Web push notifications. Research in this field, as previously mentioned in this paper, is currently lacking. However, a few notable mentions do exist. Lee et al. [31] analyzed the connection between web push notifications and phishing attacks, and found multiple websites abusing the notification icon by using the logo of a well-known website such as WhatsApp or YouTube to increase the click rate. Moreover they found how, at the time, Firefox desktop and mobile in certain environments, and Samsung Internet browser in all cases, didn't show the origin of the notification, potentially allowing attackers to mount an even more realistic phishing attack. Finally, they found how third-party providers making use of subdomains to send notifications make the problem worse by confusing the user. Unrelated to push notifications, they also showed how to exploit the offline browsing feature in PWAs using a cache to infer the victim's history of visited PWAs. Loreti et al. [36] researched the privacy issues related to mobile push applications and found that an attacker, if able to trigger a notification to a device, can successfully detect whether such device is currently located on the same network. Finally, Subramani at al. [44] developed a crawler and instrumented Chromium to analyze WPN notifications and discover malicious ad campaigns. While their patching strategy is similar to the system we developed, there are differences in the API calls collected. Moreover, their research is focused on the content of sent notifications and their use for malicious advertisement, while our work aims at identifying pitfalls that occur during the implementation and deployment of push notifications and, thus, focuses more on the service workers' source code.

Mobile push notifications. While the underlying technology and implementation of push notifications in mobile platforms differ significantly to those of web push notifications due to the inherent differences between web app and mobile app capabilities, and we, prior work has explored various aspects of Android push notifications. One of the first such security studies was by Li et al. [33], who conducted a security analysis of popular mobile push messaging services, under the assumption that the attacker was able to install a malicious app on the user's device. Liu et al. [35] explored aggressive push notification practices, and proposed a system for automatically exercising Android apps and detecting aggressive push notification behaviors. Prior work by Hyun et al. [24] and Lee et al. [30] explored how botnets could leverage push notifications for establishing stealthy C&C channels.

History sniffing. Many attacks on CSS properties have been proposed during the years [15, 22, 23, 42, 47] which enabled attackers to know the browser's history of a visiting user. Felten and Schneider [18] analyzed web timing attacks with the objective of compromising user data, in particular their browsing history. By measuring the time needed to access a resource the attacker could detect whether it was previously cached. Side-channel attacks, as demonstrated by Wu et al. [48] in the specific case of browser rendering, and service workers, as shown by Karami et al. [26], are another vector which an attacker may use to obtain information about users' browsing history. Finally, Lee et al. [32] demonstrated how AppCache, a novel functionality of HTML5 could allow an attacker to detect the status of a target URL and infer previous logins on such websites.

CSRF. CSRF vulnerabilities have been extensively investigated by the security community and numerous prior studies have explored the threat that they pose [11, 13, 16, 27, 28, 50]. Semastin et al. [41] analyzed existing measures and proposed a combined solution, Barth et al. [12] proposed new techniques, while Zheng et al. [50] analyzed methods to bypass existing protections. However, detecting such flaws remains a complex task and is an active research topic. Calzavara et al. [13] proposed an approach based on machine learning for detecting CSRF vulnerabilities in a black-box scenario, while Khodayari et al. [27] focused on client-side CSRF and released a tool to test for its presence.

9 CONCLUSIONS

Web push notifications present a departure from the multitude of web technologies of the past, as they obviate users' ability to access account functionality regardless of the device or browser being used. Instead, push notification create a notification channel whose client-side endpoint is attached to a specific browser instance. By deviating from the de-facto user abstraction of an *account* to that of a *browser*, push notifications create an incompatibility that significantly complicates deployment and integration with other web app functionality (e.g., session management). In this paper we presented an in-depth empirical analysis of push notifications in the wild, and highlighted a series of implementation pitfalls. We also demonstrated a series of attacks, including a novel history-sniffing attack, that affect popular web services and push notification providers. To that end we provided a series of guidelines for developers, including a detailed workflow for correctly handling push notifications when users need to authenticate and use multiple devices to access their account. Overall, push notifications constitute a uniquely interesting web technology that can expose users to significant threats, and should be further scrutinized by the security community.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers and Shepherd for their helpful feedback. This project was supported by the National Science Foundation (CNS-2211574, CNS-2143363). The views in this paper are only those of the authors and may not reflect those of the US Government or the NSF.

REFERENCES

[1] 2013. w3af. http://w3af.org/.
[2] 2023. *Cleverpush*. https://cleverpush.com/
[3] 2023. *LetReach*. https://www.letreach.com/
[4] 2023. *Mozilla autopush*. https://autopush.readthedocs.io/
[5] 2023. OWASP ZAP. https://www.zaproxy.org/.
[6] 2023. *Puppeteer*. https://pptr.dev/
[7] 2023. *Push API*. https://www.w3.org/TR/push-api/
[8] 2023. *Push Notifications Usage Distribution in the Top 1 Million Sites*. https://trends.builtwith.com/widgets/push-notifications
[9] 2023. *Webpushr*. https://www.webpushr.com/
[10] Mir Masood Ali, Binoy Chitale, Mohammad Ghasemisharif, Chris Kanich, Nick Nikiforakis, and Jason Polakis. 2023. Navigating Murky Waters: Automated Browser Feature Testing for Uncovering Tracking Vectors.. In *NDSS*.
[11] Elham Arshad, Michele Benolli, and Bruno Crispo. 2022. Practical attacks on Login CSRF in OAuth. *Computers & Security* 121 (2022), 102859.
[12] Adam Barth, Collin Jackson, and John C Mitchell. 2008. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*. 75–88.
[13] Stefano Calzavara, Mauro Conti, Riccardo Focardi, Alvise Rabitti, and Gabriele Tolomei. 2019. Mitch: A machine learning approach to the black-box detection of CSRF vulnerabilities. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 528–543.
[14] Stefano Calzavara, Hugo Jonker, Benjamin Krumnow, and Alvise Rabitti. 2021. Measuring web session security at scale. *Computers & Security* 111 (2021), 102472.
[15] Andrew Clover. 2002. *CSS visited pages disclosure*. https://lists.w3.org/Archives/Public/www-style/2002Feb/0039.html
[16] Luca Compagna, Hugo Jonker, Johannes Krochewski, Benjamin Krumnow, and Merve Sahin. 2021. A preliminary study on the adoption and effectiveness of SameSite cookies as a CSRF defence. In *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 49–59.
[17] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. 2020. The cookie hunter: Automated black-box auditing for web authentication and authorization flaws. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1953–1970.
[18] E. W. Felten and M. A. Schneider. 2000. Timing attacks on web privacy. *Proceedings of the ACM Conference on Computer and Communications Security*. https://doi.org/10.1145/352600.352606
[19] Gertjan Franken, Tom Van Goethem, and Wouter Joosen. 2018. Who left open the cookie jar? A comprehensive evaluation of third-party cookie policies. *Proceedings of the 27th USENIX Security Symposium*.
[20] Diana Freed, Jackeline Palmer, Diana Minchala, Karen Levy, Thomas Ristenpart, and Nicola Dell. 2018. "A Stalker's Paradise" How Intimate Partner Abusers Exploit Technology. In *Proceedings of the 2018 CHI conference on human factors in computing systems*. 1–13.
[21] Mohammad Ghasemisharif, Chris Kanich, and Jason Polakis. 2022. Towards automated auditing for account and session management flaws in single sign-on deployments. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1774–1790.
[22] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. 2012. Scriptless attacks - Stealing the pie without touching the sill. *Proceedings of the ACM Conference on Computer and Communications Security*. https://doi.org/10.1145/2382196.2382276
[23] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk. 2014. Scriptless attacks: Stealing more pie without touching the sill. *Journal of Computer Security* 22 (2014). Issue 4. https://doi.org/10.3233/JCS-130494
[24] Sangwon Hyun, Junsung Cho, Geumhwan Cho, and Hyoungshick Kim. 2018. Design and analysis of push notification-based malware on android. *Security and Communication Networks* 2018 (2018).
[25] Jordan Jueckstock, Shaown Sarker, Peter Snyder, Aidan Beggs, Panagiotis Papadopoulos, Matteo Varvello, Ben Livshits, and Alexandros Kapravelos. 2021. Towards Realistic and Reproducible Web Crawl Measurements. In *Proceedings of The Web Conference (WWW)*.
[26] Soroush Karami, Panagiotis Ilia, and Jason Polakis. 2021. Awakening the Web's Sleeper Agents: Misusing Service Workers for Privacy Leakage. In *NDSS*.
[27] Soheil Khodayari and Giancarlo Pellegrino. 2021. JAW: Studying client-side CSRF with hybrid property graphs and declarative traversals. In *USENIX Security Symposium*.
[28] KirstenS. 2021. Cross Site Request Forgery (CSRF) | OWASP Foundation.
[29] Brian Kondracki, Assel Aliyeva, Manuel Egele, Jason Polakis, and Nick Nikiforakis. 2020. Meddling middlemen: Empirical analysis of the risks of data-saving mobile browsers. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 810–824.
[30] Hayoung Lee, Taeho Kang, Sangho Lee, Jong Kim, and Yoonho Kim. 2014. Punobot: Mobile botnet using push notification service in android. In *Information Security Applications: 14th International Workshop, WISA 2013, Jeju Island, Korea, August 19-21, 2013, Revised Selected Papers 14*. Springer, 124–137.
[31] Jiyeon Lee, Hayeon Kim, Junghwan Park, Insik Shin, and Sooel Son. 2018. Pride and prejudice in progressive web apps: Abusing native app-like features in Web

applications. *Proceedings of the ACM Conference on Computer and Communications Security.* https://doi.org/10.1145/3243734.3243867

[32] Sangho Lee, Hyungsub Kim, and Jong Kim. 2015. Identifying Cross-origin Resource Status Using Application Cache. https://doi.org/10.14722/ndss.2015.23027

[33] Tongxin Li, Xiaoyong Zhou, Luyi Xing, Yeonjoon Lee, Muhammad Naveed, XiaoFeng Wang, and Xinhui Han. 2014. Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security.* 978–989.

[34] Xu Lin, Panagiotis Ilia, and Jason Polakis. 2020. Fill in the blanks: Empirical analysis of the privacy threats of browser form autofill. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security.* 507–519.

[35] Tianming Liu, Haoyu Wang, Li Li, Guangdong Bai, Yao Guo, and Guoai Xu. 2019. Dapanda: Detecting aggressive push notifications in android apps. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 66–78.

[36] Pierpaolo Loreti, Lorenzo Bracciale, and Alberto Caponi. 2018. Push attack: Binding virtual and real identities using mobile push notifications. *Future Internet* 10 (2018). Issue 2. https://doi.org/10.3390/fi10020013

[37] Francesco Marcantoni, Michalis Diamantaris, Sotiris Ioannidis, and Jason Polakis. 2019. A large-scale study on the risks of the html5 webapi for mobile sensor-based attacks. In *The World Wide Web Conference.* 3063–3071.

[38] Panagiotis Papadopoulos, Panagiotis Ilia, Michalis Polychronakis, Evangelos P. Markatos, Sotiris Ioannidis, and Giorgos Vasiliadis. 2019. Master of Web Puppets: Abusing Web Browsers for Persistent and Stealthy Computation. https://doi.org/10.14722/ndss.2019.23070

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

[40] PushEngage. 2021. Email vs Push Notifications: Statistics & Expert Strategies. https://www.pushengage.com/email-vs-push-notifications-statistics.

[41] Emil Semastin, Sami Azam, Bharanidharan Shanmugam, Krishnan Kannoorpatti, Mirjam Jonokman, Ganthan Narayana Samy, and Sundresan Perumal. 2018. Preventive measures for cross site request forgery attacks on Web-based Applications. *International Journal of Engineering and Technology(UAE)* 7 (2018). Issue 4. https://doi.org/10.14419/ijet.v7i4.15.21434

[42] Michael Smith, Craig Disselkoen, Shravan Narayan, Fraser Brown, and Deian Stefan. 2018. Browser history re:visited. In *12th USENIX Workshop on Offensive Technologies (WOOT 18).* USENIX Association, Baltimore, MD. https://www.usenix.org/conference/woot18/presentation/smith

[43] Marco Squarcina, Stefano Calzavara, and Matteo Maffei. 2021. The Remote on the Local: Exacerbating Web Attacks Via Service Workers Caches. *Proceedings - 2021 IEEE Symposium on Security and Privacy Workshops, SPW 2021*, 432–443. https://doi.org/10.1109/SPW53761.2021.00062

[44] Karthika Subramani, Xingzi Yuan, Omid Setayeshfar, Phani Vadrevu, Kyu Hyung Lee, and Roberto Perdisci. 2020. When Push Comes to Ads: Measuring the Rise of (Malicious) Push Advertising. *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC.* https://doi.org/10.1145/3419394.3423631

[45] M Thomson and P Beverloo. 2017. Voluntary Application Server Identification for Web Push. *IETF Tools* (2017).

[46] Takuya Watanabe, Eitaro Shioji, Mitsuaki Akiyama, and Tatsuya Mori. 2020. Melting Pot of Origins: Compromising the Intermediary Web Services that Rehost Websites. https://doi.org/10.14722/ndss.2020.24140

[47] Gilbert Wondracek, Thorsten Holz, Engin Kirda, and Christopher Kruegel. 2010. A practical attack to de-anonymize social network users. *Proceedings - IEEE Symposium on Security and Privacy.* https://doi.org/10.1109/SP.2010.21

[48] Shujiang Wu, Jianjia Yu, Min Yang, and Yinzhi Cao. 2022. Rendering Contention Channel Made Practical in Web Browsers. In *31st USENIX Security Symposium (USENIX Security 22).* 3183–3199.

[49] David Zeber, Sarah Bird, Camila Oliveira, Walter Rudametkin, Ilana Segall, Fredrik Wollsén, and Martin Lopatka. 2020. The representativeness of automated web crawls as a surrogate for human browsing. In *Proceedings of The Web Conference 2020.* 167–178.

[50] Xiaofeng Zheng, Jian Jiang, Jinjin Liang, Haixin Duan, Shuo Chen, Tao Wan, and Nicholas Weaver. 2015. Cookies Lack Integrity: Real-World Implications. In *24th USENIX Security Symposium (USENIX Security 15).*