SoK: *Workerounds* - Categorizing Service Worker Attacks and Mitigations

Karthika Subramani*, Jordan Jueckstock†, Alexandros Kapravelos†, Roberto Perdisci*‡
**University of Georgia,* †*North Carolina State University,* ‡*Georgia Institute of Technology*

***Abstract*—Service Workers (SWs) are a powerful feature at the core of *Progressive Web Apps*, namely web applications that can continue to function when the user's device is offline and that have access to device sensors and capabilities previously accessible only by native applications. During the past few years, researchers have found a number of ways in which SWs may be abused to achieve different malicious purposes. For instance, SWs may be abused to build a web-based botnet, launch DDoS attacks, or perform cryptomining; they may be hijacked to create persistent cross-site scripting (XSS) attacks; they may be leveraged in the context of side-channel attacks to compromise users' privacy; or they may be abused for phishing or social engineering attacks using web push notifications-based malvertising.**

In this paper, we reproduce and analyze known attack vectors related to SWs and explore new abuse paths that have not previously been considered. We systematize the attacks into different categories, and then analyze whether, how, and estimate when these attacks have been published and mitigated by different browser vendors. Then, we discuss a number of open SW security problems that are currently unmitigated, and propose SW behavior monitoring approaches and new browser policies that we believe should be implemented by browsers to further improve SW security. Furthermore, we implement a proof-of-concept version of several policies in the Chromium code base, and also measure the behavior of SWs used by highly popular web applications with respect to these new policies. Our measurements show that it should be feasible to implement and enforce stricter SW security policies without a significant impact on most legitimate production SWs.

1. Introduction

Service Workers [1] are a *powerful feature* [2] at the core of *Progressive Web Apps* [3], namely web applications that can continue to function when the user's device is offline and that have access to device sensors and capabilities previously accessible only by native applications. In practice, a Service Worker (SW) is a JavaScript Worker [1] script with the following high-level properties: (i) it is installed by a web application rendered in a browser; (ii) after installation, the SW can act as a proxy for network requests issued by its web application, and can thus control how web content is retrieved (e.g., from a local cache or the network) and what content is eventually passed to the application; (iii) it is an event-driven process that runs in the background, even when its web application is not actively rendered on the browser, and that can be activated by the browser based on events such as receiving a web push message [4] or a request to fetch a web page on behalf of its web application, among others.

Because SWs are a powerful feature, browser developers are mindful of potential security risks that come with them. Therefore, over time browsers have implemented a number of security policies around SWs to limit potential abuse (see Section 2). As an example, SW files can only be requested from a secure first-party origin (essentially, via HTTPS and from the same domain as the installing web application's origin). However, during the past few years, researchers have found a number of ways in which SWs may still be abused to achieve different malicious purposes. For instance, SWs may be abused to build a web-based botnet [5], launch DDoS attacks, or perform cryptomining [6]; they may be *hijacked* to create persistent cross-site scripting (XSS) attacks [7]; they may be leveraged in the context of side-channel attacks to compromise users' privacy [8]; or they may be abused for phishing [6] or social engineering attacks using web push notifications-based malvertising [9].

In this paper, we reproduce and analyze known attack vectors related to SWs, and explore new abuse paths that have not previously been considered (Section 3). We first systematize this information by grouping the attacks into different categories, based on the fundamental SW security weaknesses that make the attacks possible. Afterwards, we analyze whether, how, and estimate when these attacks have been published and mitigated by different browser vendors, and organize this information into an *attacks and mitigations* timeline (see Section 4, Table 1 and Figure 1). Then, we discuss a number of open SW security problems that to the best of our knowledge are currently unmitigated. Accordingly, we propose SW behavior monitoring approaches and new browser policies that we believe should be implemented by browsers to further improve SW security (Section 5). While preventing all types of SW abuse may not be possible, we aim to propose policies that can limit the damage that potential SW attacks can make, while minimizing the impact the proposed browser changes may have on existing legitimate SW code. To demonstrate the feasibility of the proposed browser policy changes, we implement a proof-of-concept version of several policies in the Chromium code base, and also measure the behavior of SWs used by highly popular web applications with respect to these new policies (Sections 6 and 7).

In summary, we make the following main contributions:

- We reproduce previously known attacks that abuse SWs, discuss new paths for abuse that were previously not considered, and systematize these SW attacks into categories based on the fundamental features that make them possible.
- We study whether, how, and estimate when the SW attacks have been mitigated by different browser vendors, and organize this information in an *attacks and mitigations* timeline.
- We discuss open security problems related to SWs and propose new browser policies that aim to reduce the potential for future SW abuse.
- Finally, we implement a proof-of-concept version

of a number of such policies in Chromium. Also, we measure how policy parameters could be tuned to limit SW abuse without significantly impacting legitimate SWs used by popular websites. To this end, we build a *SW forensic engine*, namely an instrumented Chromium browser that allows us to obtain fine-grained information on the behavior of SW code for real-world web applications.

- In addition, we disclosed the new attacks we found to browser vendors and obtained confirmation of their effectiveness. We also release our repository of reproduced and new SW attacks, SW forensic engine, proof-of-concept browser policy implementations and measurements data to the community [**?**], [10].

2. Background

In this section, we provide a brief background on Service Workers (SWs), focusing primarily on properties that are used as part of the attacks and mitigations described in later sections.

2.1. Service Workers

A Service Worker (SW) is a JavaScript Worker [1], namely an event-driven script that runs in the background and that does not have direct access to the DOM. To run in the background, a SW first needs to be registered by a web page. The SW code has to be contained in a JavaScript file hosted under the same origin as the origin of the web page that invokes its registration. Once installed, the SW can be programmed to cache web pages that may be later served to the user even if the browser is offline. This allows a web application running in the browser to behave more like a native application, even when the user's device connectivity to the Internet is unreliable. In addition, SWs can receive push messages and send web push notifications to the user even when the related web application is not open on the browser, in a way similar to native application's notifications.

2.2. SW Lifecycle

Once a website registers a SW, the SW code goes through an installation and activation phase, after which it can *control* web page requests under the website's origin [11]. Before installation completes, the SW can import additional scripts into the worker's context by using the importScripts API. As such, additional code may be imported from any third-party origin. The SW is ready to use only after it is activated. Installed SWs can be updated at any point of time to a new version. Automatic checks for these updates are scheduled by the browser at an interval of 24 hours or whenever a user visits a web page that the SW controls. An update could also be triggered at any point of time by using the *Update* API. Furthermore, a SW can be explicitly de-registered by its web application.

Once installed, the SW is activated immediately, if there is no pre-existing SW installed from the same origin. Otherwise, it needs to wait for a previously installed SW to finish its execution. If required, this wait period can be skipped by using the *skipWaiting* API. Once activated, the status of the SW remains set to *running* until it is terminated by the browser. Each time an event is sent to the SW, the browser activates the SW code and signals the SW about the event.

2.3. SW Scope

Each SW has a *scope* that can be specified during the registration process [12]. The scope represents the URL path under which web pages are controlled by the SW[1]. If no scope is specified, then by default the SW acquires the scope of the URL path under which the SW file is hosted. Currently, a website can have only one SW registered with a given scope. However, multiple SWs can be registered under the same origin if they have different scopes. If a SW, SW_R, is registered with a scope at the root level (i.e., `scope='/'`), it will gain control over all pages of the website. However, if a second SW, SW_A, is registered with a more specific scope (e.g., `scope='/test'`), this SW is given priority over pages under its specific scope. Therefore, any requests made for web pages under this specific scope (e.g., /test/page.html) will be handled by SW_A and not SW_R. However, SW_A will not have access to requests made by web pages outside its scope. Notably, a user doesn't have to visit a web page within the scope of the SW for that particular SW to be registered. For example, when the user visits a web page at the website's root level (e.g., /index.html), that page can register multiple SWs with different scopes.

2.4. Handling Network Requests

Once a SW is activated, it can listen to *fetch* events from web pages under its scope and thus intercept requests for web content. The SW can then make network requests for the requested content and cache them (using the *Cache* API). Later, when a cached resource is requested again, it can be served from the cache, which can help to reduce content load latency and enables a web application to continue working even if the device is offline. As a result, SWs gain a powerful ability that allows them to monitor users' requests and also modify the response sent back to the web page. As we will discuss in later sections, this ability could lead to SW abuse and potential leaks of sensitive information to third party sites (see Sections 3 and 5).

2.5. Push Notifications

A significant component of SWs is the ability to send web push notifications to users who grant permission. To use push notifications, a SW has to subscribe to a push service by using the *PushManager.subscribe* API [13]. This includes adding an *applicationServeKey* to the options. Once subscription is successful, the browser creates an endpoint URL and an *auth* secret key [14] that shouldn't be shared outside the application. These details are later used to steer push messages to the correct SW. Whenever, a push message is sent to the browser, the browser activates the corresponding SW and signals a *push* event that the SW can handle. More details about subscribing to push notifications can be found in [14].

While push messages are received in the background, SWs can also ask the browser to display a visual notification (typically in response to a push message) to the user. To this end, SWs can call *showNotification* to display a message on the user interface. Notice that while push messages and notifications are typically used together, SWs may choose not to call *showNotification* in response to a push message being received (in some browsers, this will trigger a default notification message issued by the browser itself). Similarly, a SW can call *showNotification* independently from receiving a push message.

To send notifications to the user, a SW needs to request a one-time explicit user consent (usually during the SW registration phase), by invoking *Notification.requestPermission()*. However, in addition to the user granting permission to the

---

[1] For instance, a SW registered under origin https://example.com with scope /test has control over all web content requests under https://example.com/test.

SW via the browser UI, the browser itself can only display visual notifications to the user if OS-level permission is granted. Different OSes have their own policies regarding how applications (including the browser) can obtain such permission. As an example, in case of MacOS the permission is disabled by default, unless the user specifically grants the permission (for instance, at the end of the browser software installation process). On the contrary, Windows grant such permission by default.

2.6. Periodic Background Sync

The *Periodic Background Sync* [15] API allows web applications to configure their SWs so to make updates in the background at a periodic time interval. It can be used to trigger `periodicsync` events from the SW without any event being received from a remote server. Effectively, this feature allows a web application to keep its SW and cached content up to date. This API is currently supported by Chrome and other Chromium-based browsers, such as Edge and Opera. However, given its ability to operate in the background, the *Periodic Background Sync* poses a potential security threat that has refrained other browsers, such as Firefox and Safari, from implementing it [16]. To curb its possible abuse, browsers need to enforce a number of restrictions on the API use [17]–[19].

2.7. Security Policies

In general, browsers enforce a number of default security policies to limit potential SW abuse. For instance [20], [21]:

1) Only secure origins (HTTPS sites) can register SWs.
2) The JavaScript file containing SW code must be hosted under the same origin as the website that registers the SW.
3) A SW should be terminated if the SW code has been idle for more than 30 seconds or if an event takes more than 5 minutes to process.
4) Push notifications should trigger a user-visible notification if the SW does not explicitly issue one.
5) The use of some APIs (e.g., *Periodic Background Sync*) should be restricted by permissions that must be granted by the browser (not necessarily via a direct UI request to the user [17]).

Unfortunately, not all browsers implement all policies and, when implemented, differences exist among browser vendors. In the rest of the paper, we discuss both previously known and new ways in which an attacker could still abuse SWs to achieve malicious goals despite the SW constraints listed above.

2.8. SWs vs. Extensions and Page-Level Scripts

The security model and policies that apply to SWs differ significantly from those of browser extensions and page scripts. For instance, while extensions need explicit user permission to be installed, the installation of SWs is completely transparent to users. Also, extension icons are typically visible next to the URL bar, whereas SW installations provide no visual information to users. Yet, like extensions, SWs run in the background and have powerful privileges that may be abused by attackers. Extensions are first inspected by browser web stores before publication to curtail the distribution of malicious code, whereas we are not aware of any trusted service that vets the security of SW code. Third-party code inclusions in SWs are different from third-party scripts imported in a web page because third-party SW code will run in the background in a way that is completely transparent to the user, whereas page scripts will stop running after the related tab is closed. Additionally, Third-party code included in a SW automatically inherits all privileges of the SW. Thus, the third-party code can be used to intercept any web request and response for all URLs under the SW's origin and scope, could be awakened by push messages to perform potentially malicious actions, or to send notifications to the user at any time (even when the browser is closed, in the case of mobile devices).

3. Service Worker Abuse

In this section, we describe and categorize a number of known and new attacks that can be launched by abusing Service Workers (SWs) in different ways. The discussion is based on both a review of previous academic work on web security, as well as the review of web specifications, browser documentation, and the analysis of browser source code.

3.1. Academic Literature Review

To systematically select academic papers related to SW abuse, we first explored papers published in the top four security conferences (IEEE S&P, USENIX Security, ACM CCS, and NDSS) and presented in their web security-related conference sessions over the past 6 years. During this first step we reviewed a total of 144 papers and found 4 papers [5], [6], [8], [23] that were directly related to SW security. Next, for each of these papers, we explored their related work sections and cited references. This led us to review another 119 papers from which we identified 6 additional papers related to SW abuse [7], [9], [22], [24]–[26]. Additionally, we used Google Scholar with search keywords such as "Service Worker," "Progressive Web Apps," "security," "attacks," "abuse," etc. The search results included papers we already found in the previous two literature review steps, plus 5 more papers [27]–[31] that discuss the performance and efficiency of Progressive Web Apps, rather than focusing on security issues. As these latter 5 papers do not discuss SW abuse, they are discussed only briefly in Section 8. Overall, we reviewed over 260 papers and identified 10 relevant works [5]–[9], [22]–[26] that are related to SW abuse. In the remainder of this section, we analyze these 10 papers and categorize the SW abuse they describe.

3.2. Abuse Categories

We group the attacks discovered via our literature review, as well as new attacks found, into different categories based on the root SW features that make them possible. For most of the attacks we discuss, we (re-)produce our own proof-of-concept implementations [10], which we tested on a large number of browser versions from five major browser vendors.

Scope of Threats. In our categorization of SW threats, we mainly focus on attacks that *exploit weaknesses in SW-specific security policies and functionalities* and in which *SWs themselves are the primary subject or enabler of the attack*. These include the use of malicious SW code installed by visiting an adversary-owned website, malicious code injected into legitimate third-party SWs, the direct abuse of SWs to launch side-channel attacks that may reveal private information about a user's browsing activities, and the use of SWs to launch social engineering attacks.

A summary of the attacks we analyze is provided in Table 1, which includes a reference to relevant previous publications or online resources in which an attack was described. To the best of our knowledge, some of the attack variants we discuss were not previously considered and are thus marked as "New." In addition, Table 1 provides detailed information on different browser features or APIs that are

TABLE 1: Overview of attacks and impacted browser versions. Legend: (●) first attack impact; (○) fix released; (◐) partial fix released; (⊠) no fix released yet; (☑) always possible if notifications are supported/enabled; (✡) attack not possible.

		Abuse Vectors									Browsers				
SW Abuse Categories	**Attacks**	Push API	Notification API	Sync API	Performance API	Update API	ImportScripts	*iframe* inclusion	3rd-party code	IndexDB API	Chrome	Firefox	Edge	Safari	Opera
Continuous Execution	WebBot [5]			✓		✓					● v69.0 ○ v70.0	● v57.0 ◐ v60.0	- ○ v80.0	✡	● v56.0 ○ v57.0
	PushExe [6]	✓									✡	● v59.0 ⊠	✡	✡	✡
	StealthierPushExe [New]	✓	✓								● v85.0 ⊠	✡	● v.85.0 ⊠	✡	● v71.0 ⊠
Side-Channels	OfflineOnload [6]							✓			✡	● v59.0 ⊠	✡	● v11.1 ⊠	✡
	PerformanceTiming1 [8]				✓			✓			● v79.0 ○ v83.0	● v72.0 ◐ v73.0	● v79.0 ○ v83.0	● v12.1 ◐ v14.0	● v66.0 ○ v69.0
	PerformanceTiming2 [8]				✓			✓			● v79.0 ⊠	● v72.0 ⊠	● v79.0 ⊠	● v12.1 ⊠	● v66.0 ⊠
SW Hijacking	XSSHijack [7]						✓		✓		⊠	⊠	⊠	⊠	⊠
	ExtensionHijack [New]								✓		✡	● v82.0 ○ v95.0	✡	✡	✡
	LibraryHijack [New]						✓		✓		⊠	⊠	⊠	⊠	⊠
	IndexDBHijack [22]						✓		✓	✓	● v55.0 ⊠	● v52.0 ⊠	● v80.0 ⊠	● v11.1 ⊠	● v42.0 ⊠
Push API and Notifications Abuse	Phishing [6]	✓	✓								☑	☑	☑	☑	☑
	Malvertising [9]	✓	✓								☑	☑	☑	☑	☑
	Stalkerware [22]	✓									☑	☑	☑	☑	☑

exploited for each attack and information about what browser versions were first affected and what version provided a fix, if any. In this section we focus on categorizing the attacks, whereas browser mitigations are discussed in Section 4.

While we primarily focus on attacks that directly exploit SWs' features (Sections 3.2.1 through 3.2.4), for completeness in Section 3.2.5 we also discuss different types of abuse in which SWs are used to augment other types of web attacks, rather than SWs being the main subject or enabler of the attack.

3.2.1. Continuous Execution

Because SWs can run in the background, can issue network requests, and *can be activated at any time* (even when their related website origin is closed), they can be abused to *stealthily* run unwanted or malicious code. For instance, SWs could be abused by a malicious website to run *cryptomining* code [6] or to build a *web-based botnet* [5]. Such types of attacks are generally enabled by artificially prolonging the amount of execution time granted by the browser to SW code running in the background, thus (approximately) achieving *continuous execution*.

WebBot: Papadopoulos et al. [5] describe how to build a SW-based botnet. If a victim visits a malicious website M, this website can register a SW, S_M, which can run in the background. When executed, S_M could implement code that (i) reaches out to a command-and-control (C&C) website to receive commands and (ii) execute the received command to perform actions such as participating in DDoS attacks, *distributed password cracking*, function as a *relay proxy*, etc.

For the botnet to function properly, S_M needs to be periodically (frequently) activated. Papadopoulos et al. [5] mention that this would be possible by using the *BackgroundSync* API [32]. Based on this information alone, we were initially unable to fully reproduce the attack. However, we found an online discussion about the attack by Chromium developers [19], which stated that the attack was possible primarily due to a bug that allowed the *Update* API (see Section 2) to be invoked from a SW's *activate* listener. By combining the information provided in [5] and [19], we were able to reproduce the attack as explained below. For the attack to work, the following components are required:

- SW script whose content keeps changing at server side to appear fresher than the SW already registered.
- Leveraging the *Update* API, which checks if there are any changes made to the SW file and fetches the updated version of the SW script from the server.
- Leveraging the *BackgroundSync* API to activate the SW every time the browser is re-opened.

```
function wait(ms) {
  const tmp = setInterval(() => { /* do bad */ }, 100);
  return new Promise(res => setTimeout(res, ms));
}
self.addEventListener(’activate’, event => {
  self.registration.sync.register(’foo’);
  // Wait < 30s
  event.waitUntil(wait(25000).then(() => {
    self.registration.update();  }));
});
```

Listing 1: Example of SW self-update on *activate*

As shown in Listing 1, as the SW is activated it registers a *BackgroundSync* and then calls the *Update* API after a predefined timeout. In general, the SW execution is supposed to terminate after a fixed period of time (a few minutes). Since the SW file in the server keeps changing, calling the *update* method will fetch the newer version of the SW script. This action is followed by the browser raising the *activate* event, which causes the corresponding listener function in the SW code to be executed, where malicious code can be invoked. This cycle repeats, until the browser is closed (or the SW is explicitly unregistered). When the browser is re-opened, the *BackgroundSync* triggers a sync event and the SW will be activated again restarting the execution cycle.

PushExe: Lee et al. [6] demonstrated that if an attacker was successful in registering a SW and obtaining push notification permission from the user, she could then leverage the *Push* API to activate the SW code at any moment. In some browsers, the attack could be rendered stealthy if the SW code does not explicitly invoke the *showNotification* API when a push notification is received. Using this approach, the authors were able to keep the SW running continuously in the background for long periods of time, for instance to perform *cryptomining*.

This attack was found to work in Firefox, Edge, and the UC Browser [6], though the attack is not stealthy in Chrome because the browser displays a default notification message for every push event, which may alert the user about the presence of a malicious SW running in the background.

By independently reproducing and testing this attack, we verified that in Firefox and Edge the browser revokes the push subscription of a SW (i.e., the SW cannot receive new push events), if the SW fails to show a notification after receiving a push message for 15 and 3 times respectively, thus blocking the attack, as also mentioned in [6]. However, the attack could still be made continuously stealthy in Firefox (whereas Edge does not appear to be affected) by simply renewing the SW registration in the background, after a few push messages are received (i.e., before exceeding the browser's limit for "silent" push events), as shown in Listing 7 (in Appendix).

```
async function DisplayHideNotifications()
{self.registration.showNotification().then(
  async()=>{   const notifications =
       await self.registration.getNotifications();
       for(let i = 0; i < notifications.length; i++) {
              notifications[i].close();}
 }))}
self.addEventListener(’push’,async function(event){
  event.waitUntil( DisplayHideNotifications());
});
```

Listing 2: Immediately hiding notifications on push events

[New] StealthierPushExe: While working to attain SW Continuous Execution, we discovered a variant of **PushExe** that can overcome the limitation of default notifications being displayed to the user, which would then prevent potentially alarming the user of suspicious activity. For instance, Chrome's implementation requires that a call to *showNotification* be explicitly invoked by a SW upon receiving a push message. If *showNotification* is not explicitly called, a default notification stating *'This site was updated in the background.'* will be displayed. To avoid this, we found it was possible to first invoke *showNotification* and then immediately invoke a function that parses through all notifications displayed by the browser for the SW's origin (using *getNotifications*) and closes them using *Notification.close*, as shown in Listing 2 (notice that the code can be modified to selectively close only the attack-related notifications, if desired). As a result, at every push event the SW could display a notification and then immediately close it, making it unnoticeable by the user. This would allow for frequently activating the SW in a stealthier way (i.e., with no visible UI signal), making it possible to achieve stealthy continuous execution. We verified that this new attack work on both desktop (Windows 10) and mobile (Android 11) devices, and have disclosed it to the Chrome developers (see Section 3.3).

3.2.2. Side-Channels

This category of abuse includes attacks that allow unauthorized parties to leverage SWs to gain sensitive information by bypassing browser isolation.

OfflineOnload: In [6], Lee et al. propose a history-sniffing attack that works as follows. A user first visits the attacker's website, which registers a SW. At a later time, if the user again opens the attacker's website in *offline mode*, the SW will intercept the request and return a page that includes a number of `iframe`s whose URL points to third-party target sites. The attacker's goal is to determine if the user previously visited those sites. Lee et al. found that in some browser versions, such as Firefox 59.0.2 and Safari 11.1, if the browser is in offline mode, the top page (i.e., the attacker's page) is sent an *onload* event related to an embedded `iframe` only if the target site had already been visited by the user and a corresponding SW (with offline support) had been registered. Therefore, the attackers can register an *onload* event handler to sense if a third-party site embedded in an `iframe` was previously visited by the user.

PerformanceTiming: In a recent paper by Karami et al. [8], the authors propose two different history-sniffing attacks. Both approaches involve the user visiting the attacker's website, which includes an *iframe* that loads content from a third-party target site. Also, the attacks assume that the target website was previously visited by the user, and that it registered a SW. Furthermore, the *iframe*'s source URL must fall within the scope of the targeted website's SW.

The first attack (**PerformanceTiming1**) identifies the presence of a previously registered third-party SW by monitoring two attributes of *PerformanceResourceTiming* API, namely *workerStart* and *nextHopProtocol*. The values of these attributes change depending on whether the resource is being loaded when the target page request is served via a SW, compared to when no SW is yet registered, and can thus be used to infer whether the page was previously visited by the user. While working to reproduce this attack, we additionally found that in Firefox there exists another property of *PerformanceResourceTiming*, called *initiator*, that can also be used to identify the presence of a SW in a similar way.

The second attack (**PerformanceTiming2**) is a timing-based side-channel attack that measures the loading time for the requested `iframe` resource on the user's machine, which can be compared to a pre-calculated loading time of the resource without the presence of a SW. Because SWs often cache resources to optimize performance and enable offline browsing, the difference in the loading times can help determine the presence of a SW [8].

3.2.3. SW Hijacking

We now discuss attacks that involve *hijacking* SW functionalities, by either injecting malicious code into a legitimate SW or by injecting a malicious SW into a benign origin.

XSSHijack: In [7], Chinprutthiwong et al. present an XSS attack that can be used to hijack a legitimate site's SW. They found that the URL path of a SW script can in some cases be manipulated to inject an attacker's script into the SW code. This is possible because some websites use dynamic URL query parameters in the SW's URL path that depend on the *window.location* API. The authors demonstrate that the attacker could modify the URL parameters by tricking the users to visit a carefully crafted target URL. Although the user ends up visiting the legitimate target website, failure to validate the URL parameters could result in the injection of attacker-controlled code into the SW context during the registration of a legitimate SW. Such an attack is stealthy in that it would go unnoticed by the user or the targeted website.

[New] ExtensionHijack: We discovered another possible approach to hijack a legitimate website's SW. Specifically, we found that browser extensions can be used to inject malicious SW code in the scope of any benign origin.

Specifically, Firefox is vulnerable to SW hijacking by extensions because they are allowed to use the *FilterResponse* API, which enables them to modify the request made to fetch a SW script file during its registration phase. This API is unique to Firefox and we leverage it to demonstrate this new attack.

To this end, we developed a basic Firefox extension that has the capability to intercept requests using *WebRequest* and *WebRequestBlocking* permissions, which are commonly used by popular extensions, including ad blockers. Next, we need to filter requests made for obtaining the SW script. To achieve this, our extension uses the *OnBeforeSendHeaders* API to intercept requests and obtain their HTTP headers. We identify SW script requests, as well as scripts that are imported by the SW, by looking for the header parameter *name* with value *Service-Worker*. Once a request for SW code is identified, we read the SW file's content using the API *FilterResponse*. Before sending the file's data, we can insert at the beginning of the file a malicious code snippet (although in our proof-of-concept extension we inject harmless code) as shown in Listing 8 (in Appendix).

The advantage of this attack is that the extension itself does not explicitly execute malicious code. Rather, the extension uses allowed APIs to inject additional code to be executed in the context of a SW. This may make it more difficult for extension stores to classify the extension itself as malicious in the first place. Additionally, because malicious extensions often go unnoticed for long periods of time [33], the impact of this attack may be significant. Even if the extension is detected as malicious and removed from the browser after installation, it may be too late, as the extension may have already injected malicious SW code under many highly popular website origins, which will continue to execute on a potentially large number of browsers even after the extension is removed from the store and the browser, until the SW code is updated.

[New] LibraryHijack: Website owners can leverage third-party libraries from "push providers" (e.g., OneSignal.com, SendPulse.com, iZooto.com, etc.) to conveniently enable and manage push notification campaigns. Typically, this entails including third-party code to run within the SW of a website, W. As a result, the provider of the third-party code gains complete access to W's SW, whose capabilities go much beyond providing push notifications. For example, the SW script could be modified to intercept all fetch requests and inject new page content that may harvest sensitive user information and relay it back to an unauthorized server. Currently, there are no restrictions posed on functionalities of third-party SW libraries, and in Section 6 we discuss our findings on a third-party library that indeed seems to misuse imported push service code to track all web pages visited by the user on W.

IndexDBHijack: In a recent paper by Chinprutthiwong et al. [22], the authors propose a page-based XSS that could be used to inject malicious code into a SW. This attack leverages the IndexDB client-side storage API, which can be accessed both via scripts running in the context of a page DOM as well as via SW code running under the same origin. Notably, the proposed page-level XSS attack poisons the IndexDB storage in a manner such that, if the SW relies on content stored in IndexDB to import additional URLs, malicious scripts may be injected into an otherwise legitimate SW, thus hijacking the SW's code.

3.2.4. Push API and Notifications Abuse

SWs' permissions to send notifications can also be abused to launch different attacks. In the attacks described below, a website must first register a SW and then obtain permission from the user to send notifications.

Phishing: Lee et al. [6] discussed the possibility of launching phishing attacks via WPNs. For instance, a malicious SW could issue a notification that displays the Chrome icon and a message such as "Google Chrome Premium," and a "DOWNLOAD" button, which when clicked on could lead the user to installing malicious code. Furthermore, the authors discuss how in some cases an attacker could extract the *PushSubscription* object from network traffic [6], and then use it to spoof push messages as arriving from a legitimate domain.

Malvertising: Although web push notifications (WPNs) were initially meant to be used to send first-party messages to users to keep them engaged with a website's own content, WPNs have since become an alluring platform for advertisers to reach users even when a given publisher website is not being visited. For instance, ad networks such as VWO Engage (formerly PushCrew), Roost, PushAd, etc., provide software that allows web developers to easily include WPN-based ads to their websites. To this end, web developers typically include third-party SW code provided by these companies to their websites. Besides potentially exposing a website's SW to the LibraryHijack attack described earlier, the website may also be responsible for exposing users to malicious ads via their WPNs, as reported in [9].

Stalkerware: In a recent paper [22], the authors describe how attackers could hijack the *PushSubscription* object from a benign website to spy on the website's users, for instance by revealing private information such as the users' location, age, etc. The attack starts with a page-level XSS attack and proceeds by unsubscribing the current Push subscription and subscribing again using the attacker's app ID under the legitimate website's origin [22].

3.2.5. Augmenting Other Web Attacks

The attack categories discussed earlier directly abuse SWs to launch different types of attacks and thug represent the main focus of this paper. However, for completeness, we also describe additional examples of SW abuse whose main purpose is to augment different types of web attacks, rather than directly exploiting SWs.

CachePoisoning: Squarcina et al. [24] describe how to launch a man-in-the-middle attack against cached HTTP responses by augmenting a page-level XSS attack using SWs. Specifically, the attack assumes that an adversary can inject JavaScript code within a web page under a victim third-party origin, V. If V has registered a legitimate SW that makes use of the Cache API [34], the attacker can use her XSS code to read any page content previously cached by V's SW, which may include highly sensitive user information. In addition, the attacker's page-level code can also write into the cache shared with the SW, thus potentially extending the XSS attack to any cached page under V. This attack augmentation strategy is only possible because page-level scripts and SW code under the same origin share the same cache via the Cache API. Thus, the authors propose that SW cache resources should be isolated from the page-level cache [24].

PersistentMITM: Watanabe et al. [23] present five different attacks against rehosted websites, of which one leverages SWs to enable a persistent man-in-the-middle attack. These attacks are possible only due to a vulnerability that the authors identified in website rehosting services. Specifically, the vulnerability is due to web rehosting services using the same domain name to rehost different third-party domains. This effectively "disables" the isolation imposed by the same origin policy. Consequently, if an attacker can force a vulnerable web rehosting service to rehost her malicious website under the same origin as other legitimate third-party sites, the attacker may then be able to register a SW that will act as a proxy for all web requests from all third-party websites rehosted under the same origin. However, this attack only affects users who visit the third-party sites under the rehosted domain, rather than their respective true domain names.

CacheInference: In [25], [26], Van Goethem et al. study how side-channel attacks can leverage a number of modern web features to learn private information about a user, such as learning the user's behavior on social media sites. Among several different attack variants, one variant leverages side-channel attacks against cache entries stored by a SW. However, these attacks are mostly independent from SW code and are instead intimately linked to measuring cache access time or leveraging browser-imposed cache size limits. Furthermore, these attack variants rely on the Fetch and Cache APIs, which are available to Web Workers in general, not only SWs. Thus, these attacks can be implemented without the use of SW code.

3.3. Ethical Considerations and Disclosure

All attacks were tested using our own websites and lab client machines. No real user or production website was affected.

We disclosed our findings to affected browser vendors. First, on April 23rd, 2021, we reported the StealthierPushExe attack (see Section 3.2.1) to the Chrome, Opera and Edge developers. After about one month, we received confirmation that the attack affects Chromium-based browsers (i.e., including Chrome, Opera, Edge, and likely several other less popular browsers). At the time of writing, the Chromium developers are still discussing (in a private online forum) possible fixes. Some of the steps described in the the discussion for patching the issue follow an approach similar to some of the recommendations we propose in this paper for restricting SW execution (notice that we developed our proposed mitigation ahead of disclosing the attack to the Chromium team). To the best of our knowledge, the StealthierPushExe attack has not yet been mitigated.

Furthermore, we disclosed the Extension Hijack attack (see Section 3.2.3) to the Firefox developers on June 2nd, 2021, who confirmed that Firefox was indeed still vulnerable to this attack. Recently (on December 7, 2021), this vulnerability was fixed and the bug report has been closed accordingly [**?**].

4. Existing Mitigations

In this section, we discuss existing mitigations to some of the attacks discussed in Section 3. As mentioned earlier, we have (re-)produced a proof-of-concept version of the attacks. We then tested the attacks using multiple versions of five different browsers from different popular vendors, namely Chrome, Firefox, Edge, Safari and Opera. To make testing with multiple combinations of browser version and operating systems easier, we made use of BrowserStack.com. Our purpose was to estimate *when* (at what browser version) an attack was fixed, whether it was *fully or partially mitigated*, or if the attack is *still feasible* in some or all browsers.

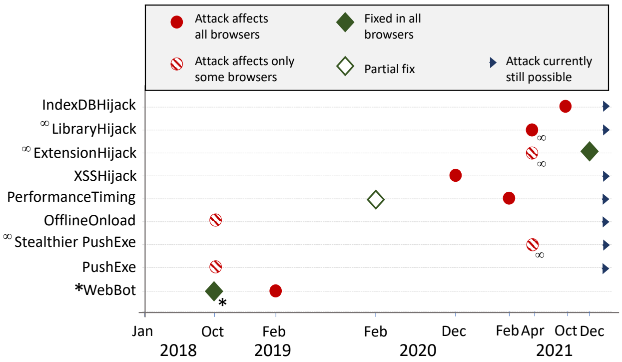


Figure 1: Approximate timeline of attacks publication and mitigations. The ∞ symbol denotes new attacks discovered in this paper, which are not yet mitigated. Notice that some mitigations were implemented before official attack publication, perhaps thanks to responsible disclosure (the '*' next to WebBot denotes that a mitigation was described in online documentation related to Firefox, but our own tests show the attack appears to still be possible on that browser).

Table 1 provides an overview of browser versions that are vulnerable to the attacks, and what browser version (if any) fixed or mitigated the vulnerability. Furthermore, Figure 1 visualizes an approximate timeline of when the attack was made public and when a mitigation for the attack was released (if any). Overall, we found that some of the attacks have been mitigated by some browsers, but also that most of the attacks are still possible on at least some of the latest browser versions. Furthermore, some of the attacks introduced in Section 3 have not yet been considered for mitigation. Notice also from Figure 1 that some attacks appear to have been mitigated by some browsers before the attack was officially published, perhaps as a result of responsible disclosure processes. Below we discuss what mitigations have already been implemented or planned so far by browsers, whereas in Section 5 we discuss open problems and propose new mitigations.

4.1. Mitigating Continuous Execution

The continuous execution attacks described in Section 3.2.1 all require the victim to visit a malicious website that will register an attacker-controlled SW on the victim's browser. By design, to avoid overwhelming the user with UI alerts for SW registration requests, the action of registering a SW is silently allowed by default and the user is not informed by the browser that the site she visited registered a SW. The mitigations discussed below still assume that SWs can be silently registered, but have the effect of limiting the number of users granting notification permissions, limiting the frequency with which a service worker is activated, or making SW execution due to web push events less stealthy.

Termination Delay Limits. This existing mitigation has the effect of fully preventing the **WebBot** attack. As explained in Section 3.2.1, the *WebBot* attack exploited a self-update behavior by continuously invoking the *Update* API, causing the SW to self-update and continue executing malicious code. To defend against this vulnerability, major browsers such as Chrome, Edge and Opera have implemented a limit of up to three minutes on the SW termination timeout (e.g., `kMaxSelfUpdateDelay` is set to three minutes in the Chromium source code), when *update* is invoked while handling an *activate* or *install* event [19]. In case of Firefox, according to an online bug report [35] a fix was implemented in v70.0. However, while testing our reproduced attack code we were able to keep the SW running for hours. On the other hand, Safari terminates the SW as soon as the related website is closed and is therefore not affected by this attack. According to [19], this fix was implemented before the **WebBot** attack was officially published in [5], as reflected in Figure 1.

Notification UI Changes. Bilogrevic et al. [36] recently showed that although 74% of all browser permission prompts that users receive are about notification permissions, only 10% of these requests are granted by users on desktop devices (21% on mobile devices). Because notification permission requests cause unwanted interruptions during normal user browsing, a new and more quiet notification prompt has been introduced in Chrome (starting from version 80) and in Firefox [36]–[38]. In Chrome, the quiet notification permission prompt is shown within the URL bar when either of two conditions are met: (i) the website requesting notification permission has a high average deny rate across its visitors, or (ii) the user recently denied notifications multiple times (e.g., 3 consecutive time) on different websites within a given timeframe (e.g., 28 days).

While we did not find indications that the quiet notification implemented by Chrome was intended to mitigate specific types of SW abuse, it may have some mitigating effect on the **PushExe** and **StealthierPushExe** attacks described in Section 3.2.1. Because these attacks require victims to grant notification permission to the attacker's website, the Chrome UI change may cause the attacker's website to be selected for the quiet notification prompt, potentially decreasing the number of users that will actually grant the permission, thus mitigating the attacks. However, as it is not a direct mitigation to those two attacks, we do not consider the quiet notifications UI as a fix for the purpose of Table 1. Furthermore, in Section 5 we also discuss how this mitigation may be easily circumvented by the attacker.

Default Notifications. We have confirmed that, at the time of writing, the *PushExe* attack described in Section 3.2.1 still works in the latest versions of Firefox (v91.0). In Chrome, the *PushExe* is not stealthy, because a default notification message[2] is shown after Chrome detects that no notification is explicitly shown by the SW. However, we verified that the *StealthierPushExe* attack that we introduced in Section 3.2.1 remains unmitigated in the latest versions of Chrome (v93.0), Edge (v93.0) and Opera (v78.0).

4.2. Mitigating Side-Channels

Event Signaling. To mitigate the **OfflineOnload** attack mentioned in Section 3.2.2, Chrome (at least since v50) ensures that the `iframe` *onload* event is triggered regardless of the presence of a SW. However, we were able to verify that this attack is still possible even for the latest version of Firefox (v91.0) and Safari (v14.0).

Site Isolation. The **PerformanceTiming1** attack summarized in Section 3.2.2 can be mitigated by making sure that meta-data related to a given origin is not revealed to third-party `iframes`. This has been fixed in Chrome since version 83.0. However, as mentioned in Section 3, we found that a variant of this attack appears to be still possible in the latest Firefox browser (v91.0) by monitoring the *initiator* property from a third-party `iframe`. Also, our tests with reproduced attack code for the **PerformanceTiming2** attack confirmed that it still remains unmitigated in the latest versions of all major browsers (see Table 1).

4.3. Mitigating Other Attacks

Besides a recent mitigation for the ExtensionHijack that followed our attack disclosure to Firefox (see Section 3.3), we are not aware of specific mitigations that have already been implemented by affected browsers to counter other hijack attacks (Section 3.2.3) or social engineering attacks (Section 3.2.4). We will discuss open problems and potential new mitigations to some of these attacks in Section 5.

5. Open Problems and New Mitigations

In this section, we revisit some of the attacks presented in Section 3 and highlight open problems that, to the best of our knowledge, have not yet been addressed by browsers. Furthermore, we also propose new mitigations that we believe should be implemented in future browser versions to address the problems we identified.

5.1. Limiting SW Execution

Open Problem: In Section 3.2.1, we discussed different ways (both previously known as well as new ones) to (silently) extend the execution time of SWs, to approximately achieve continuous execution. Although some mitigations specific to the attacks in Section 3.2.1 have been employed by some browsers, it may still be possible to create similar attack conditions that exploit existing or future SW features. For instance, to circumvent existing mitigations related to always showing notification messages to users every time a push event occurs (see Section 4.1), the SW code could be activated only at a time when the user may not be paying attention to the screen (e.g., many users leave the browser always open, even at night), as also discussed in [6]. Furthermore, even if the SW is activated a large number of times in a row using many consecutive push messages, the SW can prevent the browser from showing multiple notifications from the same website, which may make the user suspicious. To make sure that the user will only see one single notification, the SW can keep reusing the same `tag` parameter value, as shown in Listing 9 in Appendix. Fundamentally, we found that browsers do not currently put any constraints on the number of push messages a SW can receive or on the amount of execution time granted to any given SW. This leaves open possible abuse paths, as exemplified above.

Proposed Mitigation: To defend against present and future *continuous execution* attacks, we need a more generalized defense that can dynamically monitor the SW execution time and throttle it when abuse is suspected. This can be accomplish with additional browser policies. Specifically:

1) Monitor and limit the overall background execution time for which a SW runs every time it is activated. This will prevent known and unknown ways of artificially elongating the time a SW remains active (e.g., this would mitigate abuse vectors similar to the self-update exploit used in the **WebBot** attack).
2) Limit the number of push events received by a SW within a predefined time window. This would have the effect of limiting the frequency with which a SW can be remotely activated, thus throttling continuous execution attacks.
3) Ensure that a SW notification displayed to the user remains visible until the user interacts with it (e.g., by clicking on it or closing it explicitly). This would help to mitigate stealthy activations via push events.
4) Limit the volume of third-party network requests issued in the background by a SW. While this is not strictly a

---

[2] Message: "The site has been updated in the background."

limitation on execution time, it can be useful to mitigate possible "bursty" bandwidth-exhaustion DDoS attacks (e.g., by issuing many background network requests in a short execution time) against third-party websites.

In Section 7, we discuss how we implemented a proof-of-concept version of some of these policies in Chromium. In Section 6, we measure how SWs are currently used by popular websites and propose concrete thresholds to limit SW execution with limited or no impact on legitimate SW functionalities.

5.2. Limiting Malicious SW Permissions

Because by design SWs can be silently registered by any website, preventing the registration of an arbitrary SW may not be possible[3]. However, notice that without being granted the notification permission the SW cannot receive push messages and the attacker is unable to launch effective *continuous execution* attacks (Section 3.2.1) or *social engineering* attacks (Section 3.2.4), thus limiting the damage that a malicious SW may cause. The new quiet notification permission requests described in [37] (see also Section 4.1) could therefore be seen as a way of greatly restricting the damage a malicious SW can do. The reason is that, presumably, only few users would grant notification permission to an untrusted website (notice also that the permission grant rate is already low in general for most websites [36]). Thus, it is likely that a malicious site that asks its visitors for notification permission would rapidly meet the criteria to qualify for the quiet notification UI. In turn, this may have the effect of further reducing the number of users who grant permission and whose browser can be meaningfully abused by the malicious SW.

Open Problem: Unfortunately, the quiet notifications UI is not in itself an effective mitigation for limiting the number of victims that may grant notification permission to a malicious SW. One reason for this is that malicious SWs can leverage the same *double permission* prompt pattern [40], [41] that is recommended as a good practice to legitimate web developers. The *double permission* prompt consists in asking the user twice whether they would like to receive notifications from a website. The first time, the website uses JavaScript code [40], [41] to create a notification permission dialog box within the page's context (see Figure 5 in Appendix). Only if the user confirms, typically by clicking on a custom "Yes" or "Sign up" button, the SW will go ahead an request the actual notification permission through the browser UI. The reason why legitimate websites often use this pattern is because they want to avoid being blocked from asking the user for notification permission again in the near future. Since the website controls the JavaScript dialog box, the browser will not be aware that the user may want to block notifications from this site, and therefore the website gets to ask again every time the user visits it. The net effect of this legitimate (and recommended) web development pattern is that the browser may grossly overestimate the notifications allow-rate for a given website. Intuitively, it is highly likely that users who do not want to receive notifications from a website will click on the "Not now" or "Dismiss" (or equivalent) button on the JavaScript dialog box and they will not be presented with the real SW's request for notification through the browser. Ultimately, given that one of the criteria to enable quiet notifications UI is a higher denial rate, malicious sites can evade this by simply adopting the double permission pattern. In general, the very recommendation to legitimate web developers on adopting the *double permission* prompt may make the newly introduced quiet notification UI much less effective than anticipated.

Another issue is due to the fact that the attacker's site could also launch social engineering attacks similar to the ones mentioned in [42] to encourage the user to explicitly click on the quiet UI's icon and explicitly grant permission.

Proposed Mitigation: Unfortunately, preventing users from granting notification permission to a malicious SW may be difficult, as discussed above. In addition, once a SW is registered and has been granted permission, the SW will persist until the user explicitly removes them following a cumbersome process that involves going through the browser preferences and settings. If little or no constraints are imposed, this may lead to significant abuse, as described in Section 3.

As a mitigation, we argue that the browser should monitor each SW's behavior for signs of abuse. The browser could then explicitly offer the user to de-register a SW (with a specific UI dialog box) when anomalous behavior is detected, or it could automatically de-register the SW.

For instance, consider the following scenario:

- The user visits a malicious website once, at which point a SW is registered and notification permission is requested.
- Assume that the user grants notification permission at first visit (e.g., due to a social engineering attack), and that the user never visits the site again.
- Afterwards, the SW runs frequently in the background (e.g., due to frequent push events) to achieve continuous execution using one of the approaches described earlier.

In this example scenario, the browser could detect that the SW is violating one or more of the policies we proposed in Section 5.1, which will automatically limit SW execution. At the same time, the browser could detect that the user has not visited the site again since the first time the SW was registered, or that the site has a very low engagement score as defined by Chromium [43]. In this case, the browser could inform the user and ask whether she would like to de-register the SW. To make the decision easier, the browser could let the user know that the SW has been running anomalously and frequently, and that the user has visited the website only once (or very rarely). As an alternative, if the SW is not explicitly de-registered by the user and the browser continues to observe that the SW abuses execution limit policies, it may simply de-register the SW outright (notice that the SW could always be re-registered next time the user visits the same site, if the user so desires).

5.3. Restricting Third-Party Code Inclusions

Open Problem: It is well known that third-party JavaScript code inclusions come with security risks [44]. As discussed in Section 3.2.3, the common practice of including third-party code into SWs could lead to *hijacking* attacks. Content Security Policies [20] (CSPs) can be used to defend against *SW hijacking attacks* such as **XSS, LibraryHijack** and **IndexDBHijack**, for instance by using the `script-src` to restrict imported code into a SW to be loaded only from authorized domains. However, implementing this defense is up to web developers, and in Section 6 we show that only a small fraction of websites express SW-specific CSP restrictions

---

[3] Obviously, blocking a known malicious website, for instance by using URL blocklists, would also prevent any related SW to be registered. Unfortunately, threat feeds and blacklists often have gaps and may not block a malicious site for a prolonged time [39], during which many victims could visit the SW and have a SW installed.

(also, low CSP adoption is a known issue in general [45]). Unfortunately, when CSP policies are missing, the browser poses no restrictions to importing third-party code into a SW.

Proposed Mitigation: We argue that, given the potential for abuse related to SWs, the browser should follow the *fail-safe defaults* principle and deny the ability to import third-party code by default. Namely, the browser should always assume a default `script-src: 'self'` policy for SWs. The web developer could then express exceptions to this default policy by explicitly listing authorized third-party origins in the `script-src` directive (this CSP directive would need to be sent to the browser with every SW file response, which can be easily configured in modern web servers). In Section 6, we will also show that the number of different origins that would need to be authorized in current production SWs is very small (only one or two, if any).

Unfortunately, `script-src: 'self'` does not prevent `eval()` to be used in SW code [46], leaving a door open to potential code hijacking attacks such as variants of the **XSS** attack proposed in [7]. Instead, the use of `eval()` should be disabled by default and enabled explicitly by adding the directive `script-src: 'unsafe-eval'`, as for page JavaScript code. Notice also that the `worker-src` CSP directive [47] can be used to restrict what URLs may be used to load a SW file, but does not apply to the *importScripts* API. Furthermore, `worker-src` does not have any effect on blocking the use of `eval()` either.

5.4. Restricting the Scope of Third-Party Libraries

Open Problem: In some cases, web developers may want to explicitly allow third-party services, such as *push services*, to include code into their SWs. For instance, assume that website W wants to make use of push service P (e.g., OneSignal.com, iZooto.com, etc.). In this case, W would want to import P's third-party code into its SW, S_W (notice that P's origin can be easily specified in S_W's `script-src` CSP directives, as discussed earlier). Unfortunately, once P's code is imported in the context of S_W, there is no way to restrict what APIs P's code can use, thus potentially enabling a **LibraryHijack** attack (see Section 3.2.3).

```
if (’serviceWorker’ in navigator) {
  // proposed register()
  options to sonly enable use of push notifications
 // while prevent the use of other sensitive APIs
  // such as cookie store, cache, fetch events, etc.
  navigator
 .serviceWorker.register(’/pushservice_sw.js’,
        {scope: ’./pushservice_sw_scope/’,
         capabilities: ’push’, ’notifications’})
 }
```

Listing 3: Proposed change to register SW with limited capabilities.

To attempt to isolate their third-party code from the first-party website's S_W, web developers could register a separate SW, S_P, with a different *scope*, instead of running the SW under the root path of W. This would allow S_P to coexist with other SWs registered under W. In addition, S_P would not be able to intercept network requests related to content outside of its scope, thus effectively isolating the third-party SW code. However, while this would be an improvement, it does not prevent S_P from being able to directly accessing W's cookie store [48]. Furthermore, S_P would also be able to access the cache [34] and thus any content previously stored by S_W, since there is currently no cache isolation for SWs registered with different scopes under the same origin. Consequently, the third-party code could still potentially access highly sensitive information related to W.

Proposed Mitigation: To mitigate the risk of **LibraryHijack** attacks, a possible approach is to explicitly limit the *capabilities* that a given SW script can have. This list of capabilities could be expressed at registration time. At the moment, when registering a SW under a given origin (via `ServiceWorkerContainer.register()` [12]), only the *scope* of the service worker can be limited. However, we argue that the `options` parameter should be extended to allow expressing additional constraints. For instance, we could express what set of functionalities or APIs the SW is allowed to access, or what set of events it is allowed to listen to. This way, we could restrict the capabilities of a third-party SW (i.e., a SW that imports third-party code) to using the *push* and *notifications* APIs while denying the use of the cookie store, the cache, or the fetch API, as in the example code in Listing 3. On the other hand, first-party SW scripts could be registered without restrictions, so that they can use any functionality made available to SWs by the browser.

More fine-grained changes would also be useful, such as expressing whether the SW is allowed to access the cache but at the same time indicate whether the cache for this SW should be isolated by scope (notice that this proposed cache isolation mechanism could be implemented in a way similar to the cache isolation approach used for the now-deprecated *AppCache* API [49]). These browser modifications would still allow push services to provide a convenient way of managing push notification campaigns on behalf of a website W while limiting exposure of potentially sensitive information belonging to W's users. Notice also that the proposed fine-grained SW policies would be somewhat analogous to *Feature Policy* for `iframes` [50].

6. Measuring In-The-Wild SW Behavior

In this section, our objective is to measure the behavior of in-the-wild SW code used by popular websites. The main goal is to learn how production SW code may be impacted by some of the mitigations we discussed in Section 5. We focus mostly on mitigations against continuous execution attacks (Sections 5.1 and 5.2) and potentially malicious third-party code inclusions (Sections 5.3 and 5.4), and aim to learn how policy enforcement thresholds may be tuned to curtail possible attacks while minimizing their impact on legitimate SW behaviors.

6.1. Browser Instrumentation for SW Forensics

To obtain fine-grained information on the behavior of SW code for real-world web applications, we first developed an instrumented version of the Chromium browser (v84.0.4147.121) with an embedded *Service Worker Forensics* engine. Our SW forensics engine logs fine-grained information regarding the following:

- The occurrence of SW life-cycle events such as *install*, *activate*, *update*, *uninstall*, and *termination*.
- Any permissions that were requested by the SW code, such as push notifications and geolocation. In addition, we automatically grant these permissions to monitor how their related APIs are utilized.
- Detailed information about network requests issued by SWs, including the URL of resources being fetched.
- API calls made by SW code with respect to caching, push, and notification APIs.
- CPU and memory consumption, and network usage (e.g., number of third-party network requests and related URLs) for each SW instance.

- We also simulate user interactions with the browser that are required to trigger events to be handled by a SW.

The logs generated by our forensics engine are then analyzed offline to measure useful properties about the behavior of SWs in the wild. Since Chromium serves as a basis for many popular browsers, such as Chrome, Opera, Edge, etc., the measurement results we obtained can be considered as representative of SW code running on a variety of browsers.

6.2. Experimental Setup

Because the main objective of our measurements is to understand how real-world SW code behaves with respect to the mitigations we proposed in Section 5, we focus on analyzing SWs registered by the most popular websites according to Alexa.com [51]. We organize our measurement results by dividing the top Alexa websites into different *bands*, based on their ranking (e.g., 0-1K, 1K-5K, 5-10K, etc.), as shown in Table 2. Different rank bands provide insights into the behavior of websites at different levels of popularity. We visited the home page of the top 100k websites and found 5,309 sites that registered a SW. As our crawler interacted with these sites, in some cases the automated clicks led to new pages hosted on third-party domains being opened. As a result, the crawler visited an additional 609 sites with rank >100k that also registered a SW. Thus, overall we visited 5,918 websites with SWs. Of these, 1,750 websites requested to be granted notifications permission.

Since we are especially interested in mitigations against continuous execution attacks, we focused our investigation on the 1,750 (out of 5,918 websites that register a SW) whose SW code requested notifications permission, and created a small farm of automated instrumented browsers (see Section 6.1) to interact with these web pages. For each of these 1,750 web applications, we continued interacting with them and monitored their SW behavior for 3 days. To drive our instrumented browser to automatically visit and interact with these web pages we made use of custom Puppeteer [52] scripts.

Notice that our data collection and analysis does not include websites such as social media and messaging apps that may require login to send notifications to users. The reason is that for these websites the behavior of their SWs, such as the number and frequency of web push notifications, is highly dependent on user activities and social network. We exclude these sites from our measurements, as it would be highly challenging to simulate a realistic network of users that send messages to each other in a way that is representative of a large and diverse user population. However, it should be noted that SW security policies that aim to impose limits on push notifications may also include a customizable allow-list for popular social media and messaging websites.

TABLE 2: Ranking bands for Alexa's top 100K websites

Ranking Bands	Band_1	Band_2	Band_3	Band_4	Band_5	Band_6	Total
Ranking Range	0-1K	1K-5K	5K-10K	10K-50K	50K-100K	100K -1M	
#Registered SWs	160	426	437	1917	2369	609	5918
#Analyzed SWs	56	151	145	585	204	609	1750

6.3. SW Behavior Results

In Section 5.1 we discussed a number of restrictions that could be imposed on SWs to limit their execution time and reduce potential damage that a malicious SW may cause due to *continuous execution* attacks (see also Section 3.2.1). Specifically, among other mitigations (see Section 5.1 for details) we proposed to (a) limit overall background execution time, (b) limit the number of push events within a given time window, (c) ensure notifications are visible to users, and (d) limit the volume of third-party network requests.

In the following, we measure how current production SWs behave, compared to the limitations listed above. This will help us in two ways: (i) determine how different limits may impact existing SW behaviors, and (ii) inform the choice of thresholds that could be used in the implementation of new SW security policies. Detailed measurement results are reported in Figure 2 and Table 3, which we discuss below.

6.3.1. Frequency of Push Events

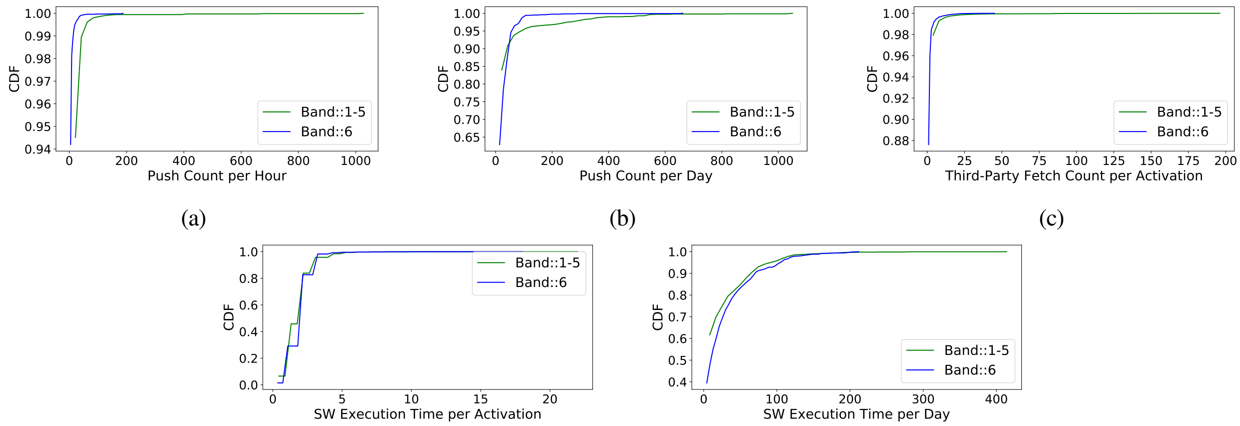
Among the 1,750 websites we monitored, 518 of them have a SW that received at least one push event during our analysis period (i.e., 3 days). To estimate the frequency with which push events are received by our instrumented browser, we divide the timeline into slots of one hour, and count the number of push events per hour for each SW. Figure 2a shows the distribution (a CDF) of the number of push events per hour across all monitored SWs. Also, from Table 3 we can see that at the 90th percentile, SWs receive 14 push events or less per hour.

From Table 3, we can see that if we implemented a SW security policy that limits the number of push events per hour to ≤ 14, this policy would affect (i.e., throttle) the SWs of 49 different websites (49 is the sum of the number of SWs under each ranking band), with almost half of the impacted websites having a ranking above 100k (Band-6). While at a first glance this result may look like a potentially significant impact on production SWs, a detailed manual analysis of the push notifications that would be curtailed by the new policy reveals something different. In fact, we found that all 49 websites that would be potentially impacted sent notifications that could be considered abusive. Specifically, the notifications we recorded from those sites are related to (potentially malicious) WPN ads, which were previously also identified by other researchers in [9] as being often abusive (some example notifications reconstructed from our logs are reported in Figure 3 in appendix). Thus, it appears that the proposed limit on the frequency of push messages would at most throttle the number of push-based ads received by users, without significantly affecting most legitimate SWs. At the same time, limiting the number of push events that can activate a SW can help to decrease the potential for continuous execution attacks that may be used for instance to perform cryptomining, DDoS attacks, or other malicious tasks, as further discussed later.

6.3.2. Execution Time

For each SW, we also measured the maximum execution time per activation. Namely, we measure the time for which a SW ran without releasing control or being forced to stop by the browser (as before, these measurements were performed throughout our 3 days of monitoring per each web application's SW code).

As it can be seen in Figure 2d and Table 3, at 99% of the instances, SWs were alive for a maximum duration of 5 minutes per activation. At the same time, we also found that 20 websites had a SW that at some point remained active beyond the maximum limit (5 minutes) allowed by the browsers. The maximum continuous execution time per activation that we observed was 22 minutes. These cases of long continuous execution were possible because the SW termination was delayed by the browser as the SW received multiple events (e.g., multiple consecutive push events) in close succession, with the next event arriving and being handled before the SW finished handling the previous event. This again demonstrates that the



(a) (b) (c)

(d) (e)

Figure 2: SW behavior measurements. Each graph displays the distribution (CDF) of occurrences of an event within a specific time window: a) Push count per hour; b) Push count per day; c) Third-party fetch count per SW activation; d) SW execution time per activation (in minutes); e) SW execution time per day (in minutes).

possibility of abusing SWs to perform malicious tasks such as cryptomining and DDoS attacks remains open. As in Section 5.1, we argue that the browser should impose stricter limits to continuous SW execution. For instance, it could be limited to 5 minutes, since 99% of all SWs activations we measured never exceeded this threshold. Longer continuous execution times should be considered as anomalous and potentially dangerous.

Besides the execution time per activation, we also calculate the overall SW execution time per day, as the sum of the execution time spent during all activations of a given SW for a day of observation. As shown in Figure 2e, 95% of SWs were active for less than 90 minutes per day. However, we found 17 websites whose SWs were active from 146 up to 400 minutes (over 6 hours). By analyzing the logs, we found that these websites “spammed” the browser with a large number of potentially malicious notifications (similar to Figure 3 in appendix). As an example of a website whose SW exhibited long execution times, we found that waploaded.com (ranked 51,299) registered a SW that sent over 50 push events per hour in multiple time windows, and as a result activated its SW and kept it running for long periods (e.g., 22 minutes in one single activation and over 4 hours in a single day).

6.3.3. Third-party Background Network Requests

To reduce the risk and impact of SWs participating in DDoS attacks, in Section 5.1 we proposed to limit the volume of third-party network requests that the SW could issue while in the background (i.e., when the related web application is not rendering on a browser tab).

To understand what may be a good volume threshold, we measured the number of fetch requests that were made to third-party origins by each SW while running *in the background*. Specifically, to identify these requests we perform two different checks: (a) first, we make sure that the request was issued by a SW by checking whether the JavaScript execution context that issued the fetch request belongs to a SW script[4], then (b) we make sure the request was executed in the background by checking if it was made from inside a *fetch* event listener, which would indicate that it was invoked when a page request is handled by a SW and thus it was not issued in the background. To verify (b), we should notice that whenever a *fetch* listener is started, it invokes the `StartFetchEvent` method and at its completion it invokes `DidHandleFetchEvent` under `ServiceWorkerGlobalScope`. We log these calls and filter out any fetch requests made between these two events, since they are not background requests. At the end of this filtering process, we are left with the *background network requests* made by SWs.

To account for network requests to explicitly authorized third-parties, such as push services that are intentionally imported into a SW’s code, we first determine the domain name associated with all URLs in *importScripts* calls and exclude them from our background network requests statistics (i.e., network requests from a SW to its push service domains are effectively counted as first-party requests). After the filtering explained above, we found that 99% of all SWs issued no more than 5 background network requests to third-party origins per each activation.

Although we did not find any evidence of in-the-wild SWs that performed malicious attacks such as DDoS attacks, we were able to reproduce attack code that can indeed send a large number (e.g., 50 per second) of third-party background network requests with no browser limitations. Therefore, we believe that limiting the number of such background requests (e.g., to ≤ 5) per activation, in combination with limiting the frequency of activations due to push events, as discussed in Section 6.3.1, is necessary to significantly reduce the risk for SW-based DDoS attacks while having minimal or no impact on the vast majority of legitimate SWs.

6.3.4. Third-Party Code Inclusions

To measure whether it is possible to limit the potential for *hijacking* attacks (see Section 5.3), we analyze the number of third-party scripts imported by SWs. To this end, Figure 4a shows the count of third-party scripts imported per SW, whereas Table 4b shows the top 10 origins related to imported scripts.

The vast majority of origins recorded in our logs belong to third-party push services (just among the top 10 origins, 7 are related to web push services). Also, as we can see from Figure 4a, the vast majority of SWs that import third-party code load it from at most one or two origins. Therefore, we believe that the *fail-safe defaults* approach we proposed in Section 5.3,

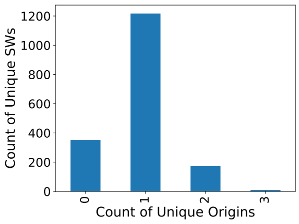
---

[4] As determined by calling `IsServiceWorkerGlobalScope()` of `ExecutionContext`.

TABLE 3: Event counts at specific distribution percentiles – The *threshold value* is the percentile value from the corresponding CDF (see Figure 5), whereas B-n represents the Alexa ranking band n.

Event Count	No. of SW Origins	Number of SW Origins above threahold value																				
		90%							95%							99%						
		Threshold Value	B-1	B-2	B-3	B-4	B-5	B-6	Threshold Value	B-1	B-2	B-3	B-4	B-5	B-6	Threshold Value	B-1	B-2	B-3	B-4	B-5	B-6
Push Count per Hour	518	14	0	1	5	11	10	22	22	0	1	3	7	5	8	43	0	0	2	6	2	5
Push Count per Day	518	38	1	2	5	13	9	43	88	1	1	3	8	4	11	392	0	0	0	3	0	1
Third Party Fetch Count per Activation	416	1	12	10	13	35	30	44	2	8	8	10	25	26	36	6	5	4	8	12	19	17
SW Execution Time Per Activation	761	3	2	1	6	16	16	38	3	2	1	6	16	16	38	5	1	1	2	5	5	6
SW Execution Time Per Day	761	64	1	2	7	9	10	33	90	1	0	3	6	4	26	146	1	0	2	3	2	9

whereby the browser should set a `script-src: 'self'` default CSP for every service worker, could be implemented with no significant impact on existing SWs. This is because the developers of existing SWs would only need to add one or two authorized origins from which additional SW code can be imported (e.g., in the Apache web server this could be done with a minimal `.htaccess` file associated with the SW file hosted on the website's first-party origin [53]).



(a)

Third-Party Domains	# of unique SW Domains
api.pushnami.com	196
www.gstatic.com	140
cdn.onesignal.com	95
app.najva.com	63
s3.amazonaws.com	50
cdn.subscribers.com	40
cdn.aimtell.com	35
storage.googleapis.com	33
cdn.izooto.com	31
cdn.p-n.io	28

(b)

Figure 3: Third-Party(TP) Imports (a) Count of TP imported domains per SWs (b) Top 10 TP domains imported by SWs

We also measured whether CSPs are currently used in relation to SW code. To this end, we analyzed the HTTP response for every SW file fetched from the 100K Alexa websites. To identify the request (and related response) for a SW file, we look for the *Service-Worker* request header, which is explicitly found when a page fetches a SW file. We found that 4.8% of all SW files have a CSP headers in the response. However, only 0.8% include the `script-src` directive. Applying `script-src: 'self'` as a default policy would reduce the risk of potential hijacking attacks, such as the **XSS** attacks presented in [7].

6.3.5. Third-Party Code Behavior

Although the default CSP restrictions discussed above may help to mitigate some hijacking attacks, such as **XSS** attacks [7], we need to be mindful that third-party libraries explicitly allowed by web developers may still behave maliciously. For example, as we studied the results of our measurements on SW code imports, we found a few cases of potential unauthorized tracking implemented in third-party libraries. For instance, we found that code imported from *coinPush* (a third-party push service) listens to *fetch* events, and can track all URLs visited on the importing website, even though this may not be necessary to enable the advertised push notification services. Listing 4 shows the (simplified) code that appears to track all visited URLs and send them to a remote site.

Such examples demonstrate that it is possible for third-party services to potentially abuse their privileged access to SW code, and should therefore be subjected to stricter default policies by browsers and much more attention by developers. Therefore, our recommendation is that the browser should implement more fine-grained SW policies (similar to *Feature Policy* for `iframes`) and web developers should carefully isolate SWs that import third-party code by correctly setting their *scope*, as we proposed in Section 5.4.

```
//Unauthorized Fetch handler to track request URLs
self.addEventListener('fetch', function (e) {
  if (e.request.url.indexOf
     (location.origin) === 0 && isDocument(e.request)) {
    trackingUrl(e.request.url); }  });
//simplified version of trackingURl method
function trackingUrl(url){
    var body = {registration_id: obj.token,
                sender_id: obj.senderId,
                logs: {url: url, timestamp: timestamp}}
    return fetch(TRACKING_SERVER + '/tracking_url
     ', { method: 'POST', mode: 'no-cors', body: body }
}
```

Listing 4: SW Code of 3rd Party Push service

7. Implementing New SW Policies

To demonstrate that implementing the new policies proposed in Section 5 is possible with reasonable effort, in the following we discuss our own proof-of-concept implementation in the Chromium (v84.0.4147.121) browser of some of those policies.

```
{    "name": <policy-name>,
     "severity": <value>,
     "threshold": <threshold>,
     "duration_in_minutes":<duration>   }
```

Listing 5: Template for count-based policies

To implement the new policies, we developed a new class called `SWPolicies` within the *Blink* rendering engine. In case of policies concerned with limiting the frequency of events that activate a SW, we follow a template similar to the one shown in Listing 5. Each time an event such as *push* or *fetch* occurs, we invoke a corresponding method to update the related counter (e.g., *push_count_per_hour*) and check if the count falls within a predefined *threshold*, which could be selected based on the trade-offs we discussed in our SW behavior measurements results (see Section 6). When a policy violation occurs (i.e., the *threshold* is exceeded), we log the violation and increase a *severity* indicator. Then, if the severity level reaches a predefined maximum value, the browser immediately terminates the SW (and could also deregister it, if the user engagement score for the SW's origin is very low).

```
  void SWPolicies::OnSWActivated(ExecutionContext* ex)
  { //start timer
    swpolicy_info->sw_timer->Start(
        FROM_HERE, kServiceWorkerRunningDelay,
        base::BindOnce(&SWPolicies::OnSWTimeout
     , base::Unretained(this), To<SWGlobalScope>(ex)));
  }
  void SWPolicies::OnSWTimeout(SWGlobalScope* gs)
  { // immediately terminate SW
    gs->SetIdleDelay(base::TimeDelta::FromSeconds(0));
  }
```

Listing 6: Example code for limiting SW execution time

As a simplified example of how to terminate a SW, to stop a SW that exceeds a given execution time we can start a timer whenever the service worker is activated, as shown in Listing 6, and attach a callback method that will be called once the timer expires. At this point, the callback can check the state of the SW and terminate it (if it is still running) by calling Blink's `SetIdleDelay` method with delay set to 0 seconds.

We use an approach similar to that described above to implement and enforce the policies proposed in Section 5.1. Furthermore, we tested these policies against a number of SW attacks (using the approaches we described in Section 3) that attempt to perform DDoS attacks, cryptomining, notification spam, etc., and verified that we are indeed able to greatly throttle such attacks, rendering them ineffective.

7.1. Discussion

As we consider the implementation of new browser policies that would restrict SW behaviors, as discussed earlier, we should carefully consider how they could impact legitimate SWs. In our measurements (Section 6), we showed that it would be possible to find enforcement thresholds whose effect is to greatly limit abuse while interfering with the behavior of only few actual legitimate SWs. In addition, we should consider that the implementation of the proposed policies could include a customizable allow-list that can be pre-populated by the browser vendor and extended with help from the user, if preferred. For instance, consider the limits on the frequency of push notifications proposed in Section 5.1. If a popular website (e.g., a social media platform) legitimately needs to send a large number of notifications (e.g., many tens or hundreds of notifications per day), such an application could be added to this allow-list.

The effect of the proposed policies against continuous execution attacks (Section 5) is that they may also limit the execution time for a small fraction of legitimate SWs. However, we should notice that browsers already implement a mechanism to terminate a SW's execution after a certain amount of execution time. Therefore, SW developers already need to take into account that their code could be forced into an idle state. Unfortunately, as we demonstrated in Section 3, attackers can trick the browser into executing SWs for much longer than it would be otherwise allowed, which motivates the new policies we proposed Section 5. Ultimately, we believe that limiting the execution time of SWs would have a small to negligible impact on legitimate SW code, while drastically reducing the risk for SW abuse. Additionally, before enforcing the new policies, browsers could grant a grace period during which an alert is issued every time a SW policy is violated without strictly enforcing the policy itself. During this transition period, developers will then have the time to adjust their SW code to make sure the new policies are not violated moving forward.

In general, because the vast majority of legitimate web applications do not require a completely unfettered access to push events, web notifications, background third-party requests, etc., as shown in Section 6, and considering the potential damage that SW abuse could cause given its powerful features (see Section 3), we believe browsers should follow an approach akin to the *least privileges* principle as much as possible and limit those and other SW privileges.

8. Related Work

Throughout the paper we have discussed a number of previous works that focus primarily on attacks against SWs or in which SWs play a fundamental role. In this section, we briefly discuss other works related to multiple aspects of web security, including some additional attacks and mitigation measures.

While our paper focuses on SW abuse, other studies focus more generally on PWAs, for instance measuring their performance compared to traditional sites, and applications in both desktop and mobile environments [27]–[30]. More works [33], [54]–[57] are dedicated to measuring privacy leakage due to third-party code, such as extensions, external libraries and ad injections. In [58], the authors discuss issues related to blind trust between cross origins and explain the extent of damage it could lead to. Other works that focus on client-side web security are [59]–[61], which include policy-based access to restrict Javascipt APIs (e.g., the Performance API) to mitigate timing based side-channel attacks. Similarly, [62] presents a cost-benefit analysis to enable restrictions per site without affecting its legitimate use. In [63], Jackson and Barth argue that the concept of "fine-grained origins" (FGO) is a flawed solution to curb origin contamination. In this paper (Section 5.4), we discussed the use of scopes as a way to isolate third-party SWs and proposed additional measures to restrict the capabilities of third-party SWs to mitigate possible origin contamination issues for SW scripts.

Other systematization of knowledge papers that consider both attacks and mitigation measures have focused on web security [64]–[66], mobile security [67], and IoT security [68]. Our work is different because we focus on attacks on Service Workers specifically, discuss existing mitigations, present a timeline of when attacks and mitigations were introduced, present open security problems, and proposed new policies that browsers could adopt to limit the damage that SW abuse could cause.

9. Conclusion

In this paper, we reproduced and analyzed known attack vectors related to Service Workers and explored new abuse paths that have not previously been considered. We systematized the attacks into different categories, and analyzed whether, how, and when these attacks have been published and mitigated by different browser vendors. Then, we discussed a number of open SW security problems that are currently unmitigated, and propose SW behavior monitoring approaches and new browser policies that we believe should be implemented by browsers to further improve SW security. Furthermore, we implement a proof-of-concept version of several policies in the Chromium code base, and also measure the behavior of SWs used by highly popular web applications with respect to these new policies. Our measurements show that it is feasible to implement and enforce stricter SW security policy without a significant impact on most legitimate production SWs.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive comments and suggestions on how to improve this paper. This material is based in part upon work supported by the National Science Foundation (NSF) under grants No. CNS-2126641 and CNS-2047260, and by the Office of Naval Research (ONR) under grant N00014-21-1-2159. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or ONR.

References

[1] "Mdn web docs: Service worker," https://developers.google.com/web/fundamentals/primers/service-workers.

[2] "The chromium projects," https://www.chromium.org/Home/chromium-security/prefer-secure-origins-for-powerful-new-features.

[3] "Web.dev: Progressive web apps," https://web.dev/progressive-web-apps/.

[4] "Mdn web docs:push api," https://developer.mozilla.org/en-US/docs/Web/API/Push_API.

[5] P. Papadopoulos, P. Ilia, M. Polychronakis, E. Markatos, S. Ioannidis, and G. Vasiliadis, "Master of web puppets: Abusing web browsers for persistent and stealthy computation," *ArXiv*, vol. abs/1810.00464, 2019.

[6] J. Lee, H. Kim, J. Park, I. Shin, and S. Son, "Pride and prejudice in progressive web apps: Abusing native app-like features in web applications," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1731–1746.

[7] P. Chinprutthiwong, R. Vardhan, G. Yang, and G. Gu, "Security study of service worker cross-site scripting." in *Annual Computer Security Applications Conference*, ser. ACSAC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 643–654. [Online]. Available: https://doi.org/10.1145/3427228.3427290

[8] S. Karami, P. Ilia, and J. Polakis, "Awakening the web's sleeper agents: Misusing service workers for privacy leakage," in *Network and Distributed System Security Symposium (NDSS)*, 2021.

[9] K. Subramani, X. Yuan, O. Setayeshfar, P. Vadrevu, K. H. Lee, and R. Perdisci, "When push comes to ads: Measuring the rise of (malicious) push advertising," in *Proceedings of the ACM Internet Measurement Conference*, ser. IMC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 724–737. [Online]. Available: https://doi.org/10.1145/3419394.3423631

[10] "Sw abuse demos," https://demopwa.github.io/sw_index.

[11] J. Archibald, "The service worker lifecycle," https://developers.google.com/web/fundamentals/primers/service-workers/lifecycle.

[12] "Mdn web docs: Serviceworkerregistration," https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerRegistration.

[13] "Mdn web docs:pushmanager api," https://developer.mozilla.org/en-US/docs/Web/API/PushManager/subscribe.

[14] "Mdn web docs:push subscription api," https://developer.mozilla.org/en-US/docs/Web/API/PushSubscription/getKey.

[15] "Mdn web docs : Periodic background synchronization api," https://developer.mozilla.org/en-US/docs/Web/API/Web_Periodic_Background_Synchronization_API.

[16] "Google groups: Intent to ship: Periodic background sync," https://groups.google.com/a/chromium.org/g/blink-dev/c/KSJViFp3hMc/m/6gVYzjg_BAAJ?pli=1.

[17] "Web.dev : Richer offline experiences with periodic background sync api," https://web.dev/periodic-background-sync/.

[18] "W3c periodic background sync specification," https://wicg.github.io/background-sync/spec/PeriodicBackgroundSync-index.html#security.

[19] "Periodic background sync has serious security risks," https://github.com/WICG/background-sync/issues/169.

[20] "Service worker security policies," https://chromium.googlesource.com/chromium/src/+/master/docs/security/service-worker-security-faq.md.

[21] "W3c: Service worker," https://www.w3.org/TR/service-workers/#security-considerations.

[22] P. Chinprutthiwong, R. Vardhan, G. Yang, Y. Zhang, and G. Gu, "The service worker hiding in your browser: The next web attack target?" in *24th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 312–323. [Online]. Available: https://doi.org/10.1145/3471621.3471845

[23] T. Watanabe, E. Shioji, M. Akiyama, and T. Mori, "Melting pot of origins: Compromising the intermediary web services that rehost websites," in *NDSS*, 2020.

[24] M. Squarcina, D. F. Some, S. Calzavara, and M. Maffei, "The remote on the local: Exacerbating web attacks via service workers caches," ser. USENIX WOOT '21.

[25] T. Van Goethem, W. Joosen, and N. Nikiforakis, "The clock is still ticking: Timing attacks in the modern web," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1382–1393. [Online]. Available: https://doi.org/10.1145/2810103.2813632

[26] T. Van Goethem, M. Vanhoef, F. Piessens, and W. Joosen, "Request and conquer: Exposing cross-origin resource size," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 447–462.

[27] T. Steiner, "What is in a web view: An analysis of progressive web app features when the means of web access is not a web browser," in *Companion Proceedings of the The Web Conference 2018*, ser. WWW '18. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2018. [Online]. Available: https://doi.org/10.1145/3184558.3188742

[28] Y. Liu, E. Xu, Y. Ma, and X. Liu, "A first look at instant service consumption with quick apps on mobile devices," in *2019 IEEE International Conference on Web Services (ICWS)*, 2019, pp. 328–335.

[29] I. Malavolta, G. Procaccianti, P. Noorland, and P. Vukmirovic, "Assessing the impact of service workers on the energy efficiency of progressive web apps," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2017, pp. 35–45.

[30] A. Gambhir and G. Raj, "Analysis of cache in service worker and performance scoring of progressive web application," in *2018 International Conference on Advances in Computing and Communication Engineering (ICACCE)*. IEEE, 2018, pp. 294–299.

[31] A. Biørn-Hansen, T. A. Majchrzak, and T.-M. Grønli, "Progressive web apps: The possible web-native unifier for mobile development," in *WEBIST*, 2017.

[32] "Mdn web docs: Syncmanager api," https://developer.mozilla.org/en-US/docs/Web/API/SyncManager.

[33] N. Pantelaios, N. Nikiforakis, and A. Kapravelos, "You've changed: Detecting malicious browser extensions through their update deltas," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 477–491.

[34] "Mdn web docs: Cache api," https://developer.mozilla.org/en-US/docs/Web/API/Cache.

[35] "Bugzilla : Self-update service worker to stay alive," https://bugzilla.mozilla.org/show_bug.cgi?id=1432846.

[36] I. Bilogrevic, B. Engedy, J. Porter, N. Taft, K. Hasanbega, A. Paseltiner, H. Lee, E. Jung, M. Watkins, P. McLachlan, and J. James, ""shhh...be quiet!" reducing the unwanted interruptions of notification permission prompts on chrome," in *30th USENIX Security Symposium (USENIX Security 21)*, Vancouver, B.C., 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/bilogrevic

[37] "Quieter ui for notificationss," https://blog.chromium.org/2020/01/introducing-quieter-permission-ui-for.html.

[38] "Mozilla : Web push notifications," https://support.mozilla.org/en-US/kb/push-notifications-firefox#w_how-does-it-work.

[39] X. Bouwman, H. Griffioen, J. Egbers, C. Doerr, B. Klievink, and M. van Eeten, "A different cup of TI? the added value of commercial threat intelligence," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 433–450. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/bouwman

[40] "Webfundamentals: Permission ux," https://developers.google.com/web/fundamentals/push-notifications/permission-ux#double_permission.

[41] "Investigating the chrome ux report: How to boost web push opt-in rates," https://onesignal.com/blog/boost-your-web-push-opt-in-rates/.

[42] P. Vadrevu and R. Perdisci, "What you see is not what you get: Discovering and tracking social engineering attack campaigns," in *Proceedings of the Internet Measurement Conference*, ser. IMC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 308–321. [Online]. Available: https://doi.org/10.1145/3355369.3355600

[43] "Chromium projects : Site engagement," https://sites.google.com/a/chromium.org/dev/developers/design-documents/site-engagement.

[44] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You are what you include: Large-scale evaluation of remote javascript inclusions," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 736–747. [Online]. Available: https://doi.org/10.1145/2382196.2382274

[45] M. Weissbacher, T. Lauinger, and W. Robertson, "Why is csp failing? trends and challenges in csp adoption," in *Research in Attacks, Intrusions and Defenses*, A. Stavrou, H. Bos, and G. Portokalidis, Eds. Cham: Springer International Publishing, 2014, pp. 212–233.

[46] “Mdn web docs: Using web workers,” https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers#content_security_policy.

[47] “Mdn web docs: Csp worker-src,” https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/worker-src.

[48] “W3c cookie store api,” https://wicg.github.io/cookie-store/.

[49] “Appcache api,” https://blog.chromium.org/2020/01/appcache-scope-restricted.html.

[50] “Mdn web docs: Iframe,” https://developer.mozilla.org/en-US/docs/Web/HTTP/Feature_Policy/Using_Feature_Policy.

[51] “Alexa top sites,” https://www.alexa.com/topsites.

[52] Google, “Puppeteer: Chormium browser automation tool,” http://liwc.wpengine.com/compare-dictionaries/, 2019, (Last accessed Nov.11, 2019).

[53] “Adding csp using .htaccess,” https://content-security-policy.com/examples/htaccess/.

[54] O. Starov and N. Nikiforakis, “Extended tracking powers: Measuring the privacy diffusion enabled by browser extensions,” in *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW ’17. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2017, p. 1481–1490. [Online]. Available: https://doi.org/10.1145/3038912.3052596

[55] R. Perrotta and F. Hao, “Botnet in the browser: Understanding threats caused by malicious browser extensions,” *IEEE Security Privacy*, vol. 16, no. 4, pp. 66–81, 2018.

[56] K. Thomas, E. Bursztein, C. Grier, G. Ho, N. Jagpal, A. Kapravelos, D. McCoy, A. Nappa, V. Paxson, P. Pearce *et al.*, “Ad injection at scale: Assessing deceptive advertisement modifications,” in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 151–167.

[57] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, “Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web,” *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[58] G. Meiser, P. Laperdrix, and B. Stock, “Careful who you trust: Studying the pitfalls of cross-origin communication,” *ACM AsiaCCS*, 2021.

[59] B. Stock, M. Johns, M. Steffens, and M. Backes, “How the web tangled itself: Uncovering the history of client-side web (in)security,” in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 971–987. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/stock

[60] J. Jueckstock and A. Kapravelos, “Visiblev8: In-browser monitoring of javascript in the wild,” in *Proceedings of the Internet Measurement Conference*, 2019, pp. 393–405.

[61] M. Schwarz, M. Lipp, and D. Gruss, “Javascript zero: Real javascript and zero side-channel attacks.” in *NDSS*, 2018.

[62] P. Snyder, C. Taylor, and C. Kanich, “Most websites don’t need to vibrate: A cost-benefit approach to improving browser security,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 179–194. [Online]. Available: https://doi.org/10.1145/3133956.3133966

[63] C. Jackson and A. Barth, “Beware of finer-grained origins,” in *In Web 2.0 Security and Privacy (W2SP 2008)*, 2008.

[64] R. Rogowski, M. Morton, F. Li, F. Monrose, K. Z. Snow, and M. Polychronakis, “Revisiting browser security in the modern era: New data-only attacks and defenses,” in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017, pp. 366–381.

[65] A. Bulazel and B. Yener, “A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web,” in *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*, 2017, pp. 1–21.

[66] M. Conti, N. Dragoni, and V. Lesyk, “A survey of man in the middle attacks,” *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, pp. 2027–2051, 2016.

[67] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith, “Sok: Lessons learned from android security research for appified software platforms,” in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 433–451.

[68] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose, “Sok: Security evaluation of home-based iot deployments,” in *2019 IEEE symposium on security and privacy (sp)*. IEEE, 2019, pp. 1362–1380.

Appendix

1. Additional Example Code Snippets

```
self
 .addEventListener(’push’, async function (event) {
    // listen
  to push event and perform any computation here
    // increase push count
    push_count +=1
    // do not call ShowNotification() API

    if(push_count>10){
    //renew subscription
        var options = {
          userVisibleOnly: true,

        applicationServerKey: <applicationServerKey>
        };
        self.registration.pushManager
 .getSubscription().then(function(subscription) {
          subscription
 .unsubscribe().then(function(successful) {
             // You’ve successfully unsubscribed
             // subscribe again

    self.registration.pushManager.subscribe(options)
          }).catch(function(e) {
             // Unsubscribe failed
        })
    }
});
```

Listing 7: Example code to avoid showing notifications on push events

```
function modifySW(url, reqId) {
      let filter
  = browser.webRequest.filterResponseData(reqId);
      ...
      filter.ondata = event => {
          let
  str = decoder.decode(event.data, {stream: true});
          let code_snippet = "self
 .addEventListener(’push’, async function (event) {

       console.log(’Extension:: Received push’)} );"
          filter
 .write(encoder.encode(malicious_code+’\n’+str));
          filter.disconnect();
      }
}
```

Listing 8: Hijacking SW code from an extension

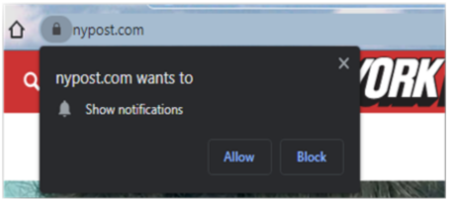
```
self
 .addEventListener(’push’, async function (event) {
    var notificationTitle = ’Same Notification!’;
    var notificationOptions = {
        body: event.data.text(),

        //same tag used for all incoming push events
        tag: ’notification-update-tag’
    };
    // replaces current displayed notification

  // with new notification due to using the same tag
    self.registration.showNotification
 (notificationTitle, notificationOptions)
}
```

Listing 9: Reusing the same notification for multiple push messages

Get notifications from The New York Post

Click 'Sign Up' then 'Allow'

(a) JavaScript-based prompt



nypost.com wants to

Show notifications

(b) Browser native prompt, after user clicked on 'Sign Up'

Figure 4: Example of double permission prompt in use on a popular website.

Amazon

$1,000 to your Amazon Account...

Join.super-samples.com

Zelle -

$750. Click Immediately

Join.super-samples.com

Viruses found (3)

Warning - Your Computer Is Infected!..

SYSTEM ERROR

Warning - Your Computer Is Infected!

Click here to turn on Virus Defender.

surveydeet.com

Figure 5: Examples of spam/malicious notifications