# The Service Worker Hiding in Your Browser: The Next Web Attack Target?

Phakpoom Chinprutthiwong
cpx0rpc@tamu.edu
Texas A&M University
Texas, USA

Raj Vardhan
raj$_v$ardhan@tamu.edu
Texas A&M University
Texas, USA

Guangliang Yang
yanggl@fudan.edu.cn
Fudan University
Shanghai, China

Yangyong Zhang
yangyong@tamu.edu
Texas A&M University
Texas, USA

Guofei Gu
guofei@cse.tamu.edu
Texas A&M University
Texas, USA

## ABSTRACT

In recent years, service workers are gaining attention from both web developers and attackers due to the unique features they provide. Recent findings have shown that an attacker can register a malicious service worker to take advantage of the victim such as by turning the victim's device into a crypto-currency miner. However, the possibility of benign service workers being leveraged is not well studied.

To bridge this gap, we systematically analyze the security of service workers from a new perspective. Specifically, we consider how an attacker can leverage a benign service worker installed in popular websites. To this end, we uncover two attack channels – IndexedDB and Push notification. Through IndexedDB, an attacker can compromise a benign service worker and persistently control the vulnerable website. Likewise, push subscription can also be easily hijacked and used to track a user's location. To understand the prevalence and security impacts of these attack channels, we conduct a measurement study on popular websites that deploy a service worker. Our results show 200 websites that are vulnerable to XSS attacks are also susceptible to push hijacking. We estimate the number of potential victims, who visit these susceptible websites and could be exposed to location tracking, to be up to 1.75 million users per month. Finally, we discuss potential defenses to prevent this problem from growing further.

## CCS CONCEPTS

• **Security and privacy → Web application security**.

## KEYWORDS

service worker, indexedDB, push notification

**ACM Reference Format:**
Phakpoom Chinprutthiwong, Raj Vardhan, Guangliang Yang, Yangyong Zhang, and Guofei Gu. 2021. The Service Worker Hiding in Your Browser: The Next Web Attack Target?. In *24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '21), October 6–8, 2021, San Sebastian, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3471621.3471845

## 1 INTRODUCTION

The service worker (SW) is a type of web worker, a script that runs in a background thread of a browser. It is executed in a new web environment, known as the *service worker context*, which co-exists with (and is isolated from) the original web environment referred to as the *document context*. A website can register a service worker to enable app-like features such as offline mode or instant push notifications. Such unique features from service workers enable the appification of websites so that they can provide mobile users an experience similar to that of using a mobile app. As a result, we refer to all SW-enabled websites as the *appified web*.

As service workers run in a unique execution context and provide several unique features, they are gaining attention from web attackers. Lee et al. [20] were the first to discuss how attackers can abuse malicious service workers to stealthily run a crypto-mining script. Subsequently, Papadopoulos et al. [25] demonstrated that a service worker can execute malicious tasks persistently. Additionally, Watanabe et al. [29] discovered an attack where attackers can register a malicious service worker in re-hosted web services to hijack websites from different origins. These initial studies have explored the security impacts when attackers register or start a malicious service worker in a victim's client. However, such a scenario may not be as practical because service workers cannot run indefinitely. For the attacker to take full advantage of a malicious service worker, the victim has to keep visiting the malicious website.

In this work, we consider an attack from a different perspective. Instead of starting a malicious service worker (i.e., in an attacker-controlled website), the attacker targets and leverages a website's benign service worker. For instance, appified websites often provide a push subscription, which can also be used to legitimately determine a user's location for sending a location-triggered notification. However, we find that an XSS attacker can hijack the push subscription and leverage it to similarly track the victim's location. In fact, as discussed by previous research [19, 24, 26, 28], XSS vulnerabilities are not uncommon. Our own evaluation of the practicality of this threat model (Section 6.2) also confirms that a considerable number of appified websites contains an XSS vulnerability. Furthermore, a recent study [16] identified a novel type of XSS called Service Worker XSS (SW-XSS), which allows web attackers to compromise a benign service worker during the service worker registration process. As XSS attacks may only be the tip of the iceberg, we aim to explore possible attack channels that XSS attackers can utilize to further leverage a benign service worker.

To this end, we first examine four communication channels between the document and SW contexts. We find two potential channels, IndexedDB and push notification, that can be utilized by attackers. For instance, the IndexedDB can be used to inject malicious code into the SW context, allowing attackers to fully compromise the service worker. Additionally, we find that attackers can remotely track the location of the victim who subscribes to push notification. Such attack also does not require attackers to fully compromise the benign service worker (only need to hijack the push subscription), contrary to the requirements of existing attacks on service workers that normally need the attackers to at least run malicious code inside the SW context.

Then, to demonstrate that the attack channels can be utilized in practice, we examine real-world websites that deploy a service worker. In this process, we uncover real cases of vulnerabilities in the two attack channels. First, we find some websites blindly trusting the data from the IndexedDB (IDB), a shared storage space between the document and SW contexts, and use it inside critical functions inside the SW context. Although the SW context is isolated, information from the document context can flow into the SW context through this channel. Second, we find legitimate push services providing location-based notifications, which can similarly be utilized by attackers to track a user's location. Furthermore, the push subscription can be easily hijacked by attackers as there is no security check in the (un)subscription processes.

Finally, to assess the security implications of these attack channels in the wild, we extend a taint tracking tool developed by Melicher et al. [22]. We add two taint sinks (IDB's *put* and *importScripts*) and two taint sources (IDB's *get* and *push message*) to the original tool. Then, we conduct a taint analysis on 7,060 popular appified websites. Our findings show that 5 websites can have their service workers compromised through the IndexedDB. Moreover, 200 websites that are already vulnerable to XSS attacks can have their push subscriptions easily hijacked. Based on the total number of visits to these susceptible websites, we estimate the upper-bounded number of potential victims, whose locations can be exposed to an XSS attacker, to be up to 1.75 million users per month. Note that these problems could not only be exclusively considered taking into account the current percentage of vulnerable websites but also future incorrect implementations. We manually evaluate the results from our tool and estimate the false positives/negatives to be minimal.

In summary, our contributions in this work are as follows:

- We analyze service workers, their communication channels, and the security design to uncover flaws in two channels, IndexedDB and push notification (Section 4).
- We demonstrate how attackers can utilize the flaws to leverage the supposedly secure and isolated SW context of a benign website using real-world examples (Section 5).
- We design a measurement tool and extend an existing taint tracking tool to assess the security of service workers. Our findings estimate up to 1.75 million users of popular appified websites are potentially exposed to location tracking per month, and five appified websites are vulnerable to service worker hijacking through the IndexedDB (Section 6).

## 2 BACKGROUND AND RELATED WORK

In this section, we first provide more background on the appified web. Then, we motivate our research by discussing existing web attacks and briefly review why further study is needed to better enhance the security of service workers.

### 2.1 Appified web

In this work, we refer to the term *appified web* to represent websites with service workers. Appified web includes several basic elements, such as (1) service workers, which serve as the back-end and handle background tasks and events, and (2) manifest files, which provide the configurations for web appification. As per our definition of appified web, a service worker must be present, but a manifest file is optional. These components serve in conjunction to deliver users a smooth browsing experience. Naturally, the Progressive Web App (PWA) is considered an appified website by our definition as a PWA requires *both* a service worker and proper manifest declaration [12]. The overview of how an appified website (or PWA) interacts with a service worker is shown in Figure 1. As our study considers all websites with service workers, the PWA is also in the scope of our evaluation.

**Service worker.** A service worker is defined in a JavaScript file and runs in a background thread of a browser. It is registered from the document context and tied to a specific path of a host. As shown in Figure 1, primarily, a service worker is used to provide two crucial functions: enabling offline usage by intercepting network requests and serving cached content; and receiving push messages to display push notifications.

To provide an offline mode to users, a service worker can intercept network traffic of the website where it is registered through the *fetch* event handler. The normal usage of this feature is to cache web resources and response when the network is offline. However, it also gives attackers MITM capability when compromised. The attackers can redirect any request to an arbitrary destination and inject any content into the response.

To handle push messages that can arrive spontaneously, a service worker runs in the background thread of the rendering browser, which can execute even when its window is closed. Additionally, once registered, the service worker can exist until the SW file is updated. Attackers can use this feature to run botnets or mine crypto-currency in the background or keep control of the document context for a long period through network interception.

Because these unique capabilities can potentially introduce new risks, web browsers are designed to limit the efficacy of malicious service workers. Previous studies have shown that malicious service workers can be used to turn a victim's client into a botnet [20], a crypto-miner [25], (user's unintended) proxy service [13], or to compromise other websites in a re-hosted environment [29]. To address these attacks (and other foreseen problems), web browsers

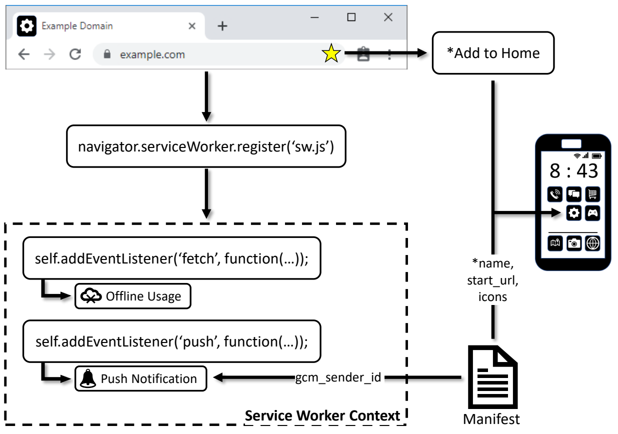


**Figure 1: Service worker registration process and PWA installation.**

have intrinsic defenses that can limit the effect of a malicious service worker. For instance, a service worker can only run for 30 seconds per event, which minimizes the effect of a crypto-mining service worker. Additionally, browsers prevent cross-domain files from being registered as a service worker. This supposedly prevents untrusted scripts from registering a service worker from an arbitrary host to replace an existing benign service worker. Hence, it may be imperative to explore another attack angle where attackers may utilize a benign service worker and prepare countermeasures before such problem arise.

**Manifest.** Similar to Intent declaration in an Android App [1], a manifest file, structured as a JSON, declares different attributes (including notification-related) of the appified web. The manifest file is also well protected as its origin must be the same as the homepage domain (similar to SW script registration) and is usually sent through HTTPS in the appified web. This limits the possibility of attackers manipulating the manifest file of a website.

## 2.2 Existing web attacks

As our work assumes the presence of XSS (Cross-site Scripting) attackers, we first review recent existing work that are germane to our study. Stock et al. [28] studied the history of client-side security and showed the prevalent of XSS attacks over a decade while identifying the root causes of the issue, which can stem from insecure postMessage handling or usage of outdated JavaScript libraries. This is in line with Lauinger et al. [19] finding on the study of outdated JavaScript libraries that many websites use known vulnerable third-party libraries. Lekies et al. [21], Melicher et al. [22], and Steffens et al. [27] also studied an emerging type of client-side XSS attack called DOM-XSS using taint tracking. Despite the effort from the research community over the past decades, XSS is still prevalent in the web nowadays. In this work, we study an orthogonal aspect of how XSS attackers can utilize a relatively new technology, the service worker, for their extended advantages. While XSS attacks are well studied, how it can affect the secure context inside a service worker has not been thoroughly discussed. Our work explores this direction by identifying bridges between

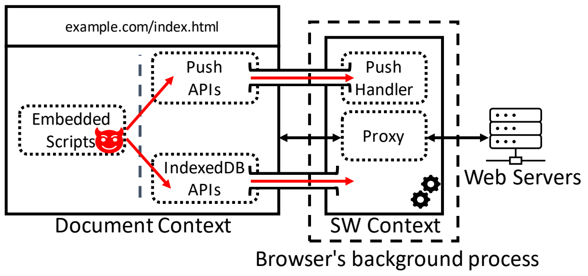


**Figure 2: Appified web threat model & attack channels.**

the document and the SW contexts, and demonstrate how attackers leverage benign service workers in practice.

There are several studies that target the security of service workers. Lee et al. [20] are the first to discuss attacks in Progressive Web Apps. They proposed novel attacks on the Cache and Push notification APIs based on insecure HTTP hosts. Our work assumes a different threat model with HTTPS enabled to hijack push subscription. Papadopoulos et al. explored potential issues with malicious service workers, which can be used by malicious websites to make users become a botnet or mine crypto-currency [25] for the attackers. Our work assumes service workers are benign but may be vulnerable while third-party scripts try to leverage the vulnerability to utilize the benign service worker. While Watanabe et al. [29] shared the same goal of compromising a benign website, our work assumes a different threat model where the target websites are not in a re-hosted environment. Chinprutthiwong et al. [16] shared our threat model and demonstrated how XSS attackers can compromise a benign service worker by manipulating the URL search parameters specified during its registration process. Our work expands upon this threat model and discover novel attack channels to leverage benign service workers.

## 3 THREAT MODEL

In this work, we assume that the service worker and all imported files in the service worker are benign. Additionally, all network links in an appified website are made over HTTPS, and we assume the absence of network attackers. Instead, we assume the presence of XSS attackers who utilize an XSS vulnerability in the document context to further compromise or leverage the service worker as shown in Figure 2. As discussed by Chinprutthiwong et al. [16], this gives the attacker more extended capabilities and attacking options including bypassing certain defenses and controlling the push subscription. Such options are especially worthwhile when the website credentials are well protected or simply do not exist. We further evaluate the practicality of this threat model in Section 6.2.

## 4 ATTACK CHANNELS

In this section, we examine four communication channels (postMessage, Cache, IndexedDB, and push message) between the document and SW contexts. As a result, we identify two potential targets for attackers: the IndexedDB and push APIs. First, we observe that the IndexedDB can be modified from the document context and read inside the SW context, allowing attacker-controlled data to reach sensitive functions. Second, the push subscription can be easily hijacked by XSS attackers as there is no security mechanisms to verify the subscribing party. As a service worker is used to handle the *push* message event, the hijacked push subscription can be used by the attackers to potentially leverage the benign service worker. Therefore, these two channels make it lucrative for attackers to pursue the benign service worker.

## 4.1 postMessage

This communication channel is the most direct method. It allows communication between the document and SW contexts and also between different iFrames. As studied by Guan et al. [17] and Son et al. [26], this communication channel can be leveraged by attackers to attack iframes. Regardless, we do not consider this channel as the main target for attackers in our threat model for two reasons. First, service workers are event-based, and their functionalities are heavily tied to event handlers, especially the *fetch* event handler that can intercept network traffic. According to the W3C's specification, the event handlers cannot be registered after the SW installation phase is done. The postMessage also utilizes the *message* handler, in which attackers would already have no way to register another event handler once the postMessage is ready to be used. Second, unlike the push message that can be triggered remotely, the postMessage needs the document context to be active. Therefore, there is no reasons for attackers to use the service worker when they cannot utilize its full potential (i.e., of the *fetch* event), and the document context already runs malicious code. Nevertheless, the attackers can still use this channel as a means to communicate with the malicious code inside the service worker that is established through a different channel (i.e., IndexedDB).

## 4.2 Cache

The *Cache* API is shared by both the document and SW contexts to provide offline usage to an appified website. In another threat model, this channel can potentially be used by attackers, i.e., by determining the state of the victim through Cache [18]. In our threat model, XSS attackers in the document context can modify the Cache and wait for the service worker to run the malicious code inside the service worker. However, we find that the information flow is the opposite in practice. Specifically, it is the service worker who writes to the Cache, and the applied changes affect the document context instead. In fact, the most common usage of the Cache API is to store the document context's static assets, which are already accessible by attackers in the document context. Therefore, we do not consider the Cache API to be the target of attackers in our threat model.

## 4.3 IndexedDB

The IndexedDB is storage that works asynchronously, thus can be used inside the SW context, unlike the *sessionStorage* and *localStorage* APIs. As it is origin-oriented (determined by the protocol/host/-port), it is protected from other website's accesses. Nevertheless, it has no defense mechanisms for service workers against untrusted scripts embedded in the document context. The document and SW contexts may be executed in isolation, but the IndexedDB has shared storage spaces that can be the weak link to let temporary XSS attackers to manipulate critical program states or the variables of service workers. For instance, it is possible for attackers to completely compromise the benign service worker if the data fetched from the IndexedDB is used inside a sensitive function like *importScripts*. Furthermore, the IndexedDB does not need a dedicated event handler like the *postMessage*, making it more available inside the service worker. Once compromised, the service worker can be used by the attackers to extend the initial attack such as to bypass certain client-side defenses or turn a temporary compromised session into a permanent one [16]. We illustrate a practical attack using the IndexedDB in a real-world case study in Section 5.1.

## 4.4 Push message

Notifications in the appified web closely resemble native app push messages, but they are handled by the browser (e.g., Chrome) instead of the OS (e.g., Android). To use push notifications, there are 3 steps that a website must follow. First, the website must explicitly ask for user permission to show a notification. Second, the website can then subscribe a user to a push subscription server, i.e., the Firebase Cloud Messaging (FCM) managed by Google. Third, if the subscription server permits the subscription request, the push credentials, including the *endpoint* working in place of the address of a subscribed user, will be returned to the website. The website can use the credentials to send push messages to the subscribed user. In the case that the website use a third-party push provider (e.g., OneSignal), these three steps are usually handled by the third-party. Normally, the website developers only need to embedded the third-party script and access the third-party web portal to manage subscribed users.

Nevertheless, we observe that it is possible for attackers to hijack the push subscription to leverage a benign service worker. Corresponding to the second step, any script can initiate the subscription and unsubscribing processes. Typically, a script can call the *subscribe* API, which accepts an optional parameter (*applicationServerKey*). When specified, the *applicationServerKey* can act as a means to identify the sender. However, there is no limitation to which key is allowed for the website's push subscription. Additionally, unsubscribing a user can also be done by any script. As a result, attackers can freely call the *subscribe* API using their own key to hijack any legitimate subscription. Because it is the service worker who handles push notifications, attackers can use the hijacked push subscription to potentially leverage the benign service worker.

After attackers successfully hijack the push subscription, they can utilize it to track user locations. We observe that third-party push providers normally offer the demographic of users and location-triggered push messages to their customers. Such features, when used legitimately, can help improve the marketing scheme of the deploying websites. However, attackers can similarly leverage these features to compromise a user's location. For instance, location-triggered messages can infer user locations, which can be as precise as in meters. Some push providers can also report when a user last visits the website that the user is subscribed to. Such information can be used to infer the victim's online behaviors, which can potentially reveal the victim's daily routine. Figure 3 demonstrates what kind of information is possibly available to attackers if the push subscription is hijacked. Therefore, attackers may not necessarily need to re-implement these *stalkerware*[1]-like features and simply leverage a benign service worker that already implements them. We further discuss how attackers can leverage this feature in practice in Section 5.2.

## 5 REAL-WORLD CASE STUDIES

In this section, we demonstrate how attackers can utilize the attack channels to leverage a benign service worker in practice using real-world case studies of appified websites. For the IndexedDB channel, we discuss a website that performs inadequate security checks for an IDB entry used inside a sensitive function inside the service worker that can allow attackers to compromise the service worker. For the push notification channel, we discuss a third-party push provider that lowers the requirements for attackers to utilize this channel to track user locations.

### 5.1 Leveraging IndexedDB

In this case study we show a real-world website that we found using IndexedDB inside the SW context unsafely. We show a simplified and anonymized code snippet of this website in Listing 1. The website initially stores a configuration variable inside the IndexedDB. Then its service worker will read the configuration and process it. As shown at lines (1-5), the service worker opens an IDB instance, fetches an entry called *data*, and obtains the *config* variable from the database. Next, at lines (6-8) the service worker reads the *url* from the *config* variable and finally passes it to the *importScripts* API at line 12. This results in the service worker importing the JavaScript file specified by the *url* variable to its secure context. By manipulating the *url* variable through IndexedDB, attackers can inject an arbitrary code to be executed in the service worker.

```
1  const request = indexedDB.open('db', 1);
2  request.onsuccess = (event) => {
3   const db = event.target.result;
4   const t = db.transaction(['data'], 'readonly')
5   const query = t.objectStore('data').get('config');
6   query.onsuccess = (event) => {
7   const data = event.target.result;
8   url = data.url;
9   var chk = "^https:\/\/(?:[^.]+\.)?example\.com\/.*$"
10  var regex = new RegExp(chk);
11  if(regex.test(url)) {
12   importScripts(url);
13  ...
```

**Listing 1: An example of a vulnerable service worker**

Although this website attempts to sanitize the *url* variable using a regular expression at lines (9-11), we find that it is insufficient. The whole regular expression would match https://example.com/sw.js or https://sub.example.com/sw.js, but it will not match with https://malicious.com/.example.com/sw.js. Hence, the attackers cannot seemingly include another file from a different domain inside this service worker. Nevertheless, the regular expression can be bypassed to inject any arbitrary domain that does not belong to the example.com's subdomain by taking advantage of URL encoding. For example, attackers can encode the "." into "%2E" resulting in https://malicious%2Ecom/.example.com/sw.js. This URL string will naturally pass the regular expression check, and more importantly, decode back correctly by the *importScripts* API allowing attackers to inject a malicious file from their controlled domain into the service worker.

Because this vulnerable code is executed before the legitimate code gets to register event handlers, the attackers can initialize the *fetch* event handler first and elevate the initial XSS attack into a kind of persistent Man-In-The-Middle (MITM) attack. By controlling the *fetch* event, which can inspect and modify all requests/responses of the website, the attackers naturally take full control of the website persistently until the service worker is replaced. Although the service worker will be replaced once a new service worker file is detected, a previous study [16] found that appified websites take 40 days on average to update their service workers. Therefore, the attackers can potentially leverage this benign service worker for a considerably long period of time.

```
1  let p = [Input manipulable by an attacker];
2  let t = decodeURIComponent(p);
3
4  if (new URL(t,location.href).host === location.host) {
5    ...
6    self.importScripts(t),
7    ...
8  }
```

**Listing 2: An example of a robust input sanitization**

Using importScripts with (non-static) parameters are not uncommon among appified websites, and robust sanitization is crucial to ensure the security of service workers. Here, we show another real-world example that uses importScripts with a sensitive parameter but with proper sanitization. The code snippet in Listing 2 shows a shorten and generalized service worker code provided by Akamai, a Cloud service provider. In this case, the variable *p* (line 1) holds a value that is manipulable by an attacker. However, this service worker reconstructs the input, obtains the origin, and compares the input origin with its own origin at line 3 before importing the result at line 5. Therefore, an attacker will not be able to leverage this service worker to import a cross-origin file. As it can be difficult to thoroughly check the correctness or completeness of a regular expression, we recommend developers to use alternative approaches similar to this example instead.

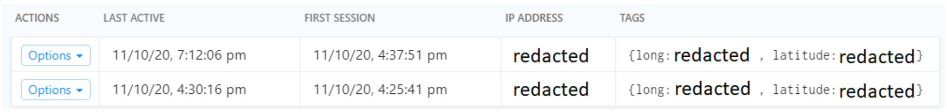
### 5.2 Leveraging push subscription

In this example, we demonstrate how attackers can leverage a benign service worker to track user locations easily through the provided functionalities of third-party push providers. The overview of the attack is illustrated in Figure 4.

Normally, an appified website can use the primitive *pushManager.subscribe* API to register for push notifications. However, in practice, a large number of appified websites utilizes a third-party push library to handle push messages. As a result, we choose the most popular (based on our measurement) third-party push library, OneSignal, as our case study. The generalized code snippet of our appified website, which we create as a proof-of-concept, is shown in Listing 3.

OneSignal (and generally any push libraries) follows a similar push subscription process with their own abstractions. At lines (1-12), our website subscribes a visitor through the *init* function, specifying the *appId* that works in place of the *applicationServerKey*.

[1]Privacy-invasive malicious software or code that tracks and monitors victim's activities, which is becoming a worrisome problem [7, 15]



| ACTIONS | LAST ACTIVE | FIRST SESSION | IP ADDRESS | TAGS |
|---|---|---|---|---|
| Options ▾ | 11/10/20, 7:12:06 pm | 11/10/20, 4:37:51 pm | redacted | {long: redacted , latitude: redacted} |
| Options ▾ | 11/10/20, 4:30:16 pm | 11/10/20, 4:25:41 pm | redacted | {long: redacted , latitude: redacted} |

**Figure 3: A screenshot from our OneSignal account page illustrating what user information can be made accessible once subscribed.**

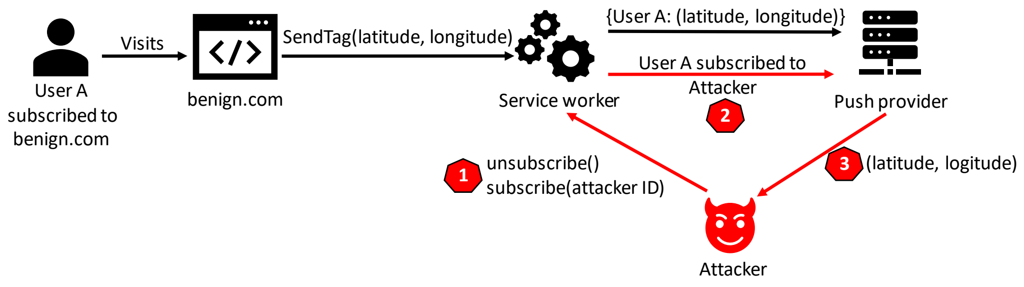

User A subscribed to benign.com
Visits
benign.com
SendTag(latitude, longitude)
Service worker
{User A: (latitude, longitude)}
User A subscribed to Attacker
2
Push provider
1
unsubscribe()
subscribe(attacker ID)
3
(latitude, logitude)
Attacker

**Figure 4: An illustration of how attackers can leverage push subscription to track user locations. Users who subscribe to legitimate websites with location-triggered notification can be vulnerable. Attackers can re-subscribe the user and persistently access user location or other information.**

The *init* function will then register a service worker with the URL search parameter *appId* set to *BENIGN_APP_ID*.

To demonstrate how attackers can hijack a push subscription, we create two OneSignal accounts, a benign account and an attacker account. We also enable location-triggered notifications as suggested by OneSignal [8] at lines (16-19). This *sendTags* function will send a visitor's location to OneSignal (if the visitor has previously granted the location permission). This information can be accessed through OneSignal account page as shown in Figure 3.

```
 1 <head>
 2 <script src="OneSignalSDK.js" async=""></script>
 3 <script>
 4   var OneSignal = window.OneSignal || [];
 5   OneSignal.push(function() {
 6     OneSignal.init({
 7       appId: "BENIGN_APP_ID"
 8     });
 9     OneSignal.registerForPushNotifications();
10   });
11 </script>
12 </head>
13 <body>
14 <script>
15   //Normal Operations
16   OneSignal.sendTags({
17     latitude: latitude,
18     long: longitude
19   });
20   ...
21   //Injected by reflected XSS
22   subscription.unsubscribe();
23   serviceWorker.unregister();
24   OneSignal.init({
25     appId: "ATTACKER_APP_ID"
26   });
27 </script>
28 </body>
```

**Listing 3: A generalized code snippet of our proof-of-concept website demonstrating how attackers can hijack OneSignal subscription and track user's location.**

Then at lines (22-26), we assume that the code is injected through an XSS attack. First, the code unsubscribes us from the benign account. Second, it un-registers the current service worker, which is tied to the benign account. Third, it re-subscribes through the *init* API with the attacker account's appId, which will automatically register a new service worker (but of the same JS file) tying to the attacker account. These steps (though produce some errors/warnings to our console) allow the attacker account to replace the push subscription from the benign account. As OneSignal provides all the implementations and also an easy-to-use web portal to access the subscribed user information, the attackers only need to run a few lines of code inside the document context to easily track victim locations.

When we navigate through other pages of our test website or close the web browser, we find that we are still subscribed to the attacker account. This is because a service worker will only get replaced/reinstall when a different service worker file is detected or the *(un)register* API is deliberately invoked. As the attacker-bound service worker uses the same legitimate file (but with a different *appID* as a URL parameter), the service worker will survive until a web page or the victim specifically requests the browser to reinstall/remove it.

Note that the steps from lines (22-26) are tested to work on OneSignal, but the problem does not tie to OneSignal's implementation. The outcome would have been the same had we use a different library or even using the native APIs, albeit the steps may be slightly different. This is because the underlying problem is with the push protocol not having any mechanism to check a list of allowed *applicationServerKey*. In the case that the target website do not use a third-party push provider, the attackers may have to implement the back-end server to handle push subscriptions and an alternative function to track geo-location instead. While this can increase the attack requirements, it does not completely repel persistent attackers. Nonetheless, we have notified OneSignal and are in contact with their developers regarding this issue.

This case study demonstrates how attackers can leverage a benign service worker (implemented by OneSignal in this case), instead of starting their own malicious service worker. The attackers simply re-subscribe the victim using their push account to utilize the location-triggered notification feature to track the victim. We find that a number of push providers are starting to advertise similar features to improve user experience [6, 11] and expect that such features will be more common in the future. In any case, further study is required to understand how many users would grant the permission for location-triggered notifications. We leave this direction to future work. Nevertheless, if attackers can also fully hijack the service worker (i.e., through the IDB channel), then they can directly use the compromised service worker to inject the location tracking code into the document context to persistently track the victim locations.

## 6 SECURITY MEASUREMENT

In this section, we present our assessment of the security of service workers. First, we discuss how we conduct a measurement study (Section 6.1). Second, corresponding to our threat model, we evaluate the prevalence of XSS vulnerabilities in appified websites (Section 6.2). Third, we assess the prevalence of the IndexedDB attack channel (Section 6.3). Last, we assess the prevalence of the push attack channel (Section 6.4). Note that the attacks discussed in our paper mostly come from design flaws and cannot be directly fixed by web developers. We have taken appropriate measures with our best effort to notify those potentially affected parties that could have the problems alleviated from the web developer side (i.e., IDB attacks and OneSignal).

### 6.1 Measurement overview

We developed a measurement tool to conduct a large-scale study on the popular websites that deploy a service worker. Initially, we use the data set provided by Chinprutthiwong et al [16], which shows the Alexa top 7,060 websites that utilize a service worker. These 7,060 websites are the target of our measurement study.

Our tool can be divided into two components. The first component contributes to collecting static data from the websites. The goal of this component is to verify that the website registers a service worker and to identify the push subscription metadata (i.e., what third-party library is used). The second component contributes to analyzing the IndexedDB usage of the website and to identify if there is any flow from the document context to the SW context. This component is extended from the Chromium Taint Tracking open source project developed by Melicher et al [22]. We add two additional taint sources, IDB Get and Push message, and two additional taint sinks, IDB Put and importScripts, to the original tool.

### 6.2 XSS vulnerability in appified websites

Previous works have reported that regular websites do embed vulnerable JS libraries that are prone to XSS attacks [19, 24, 28]. Here, we aim to evaluate whether such a trend also applies to appified websites. To identify vulnerable JS libraries in appified websites, we use vulnerability reports from OpenBugBounty [10], a public bug bounty platform that allows security researchers to submit a bug report to a vulnerable website. As OpenBugBounty contains all types of vulnerabilities, we filter out other vulnerabilities and focus on the XSS bug reports.

**Table 1: A table of XSS reports in appified websites.**

| Report type | # of websites | # of reports |
|---|---|---|
| Unpatched | 934 | 1646 |
| Patched | 1636 | 3550 |
| Onhold | 169 | 251 |
| Total | 2739 | 5447 |

We query OpenBugBounty for the bug reports of all 7,060 appified websites. The result is shown in Table 1. The reports are divided into three categories: unpatched, patched, and onhold. A report is labeled onhold for 30 days after the initial report, which also limits access to the detail of the vulnerability to prevent other attackers from leveraging the vulnerability. The result shows that there are 934/7,060 (13.23%) appified websites with an unpatched XSS report.

To verify if the bugs are still applicable, we manually inspect 30 of these reports. We confirm that the reports contain a vulnerable URL that can be easily followed to attack the vulnerable websites. Although the number of unpatched reports may be alarming, the majority of these websites do not provide login sessions or payment system (i.e., news websites or web blogs). As login credentials and payment information are the main targets of XSS attackers, we speculate that the web developers simply ignore the reports since they believe the cost to fix the bug outweigh the risks. Nevertheless, there are always associated risks even in websites that do not provide login or payment mechanisms because now websites can be equipped with a service worker that can provide extended capabilities for attackers to leverage. As we discussed in Section 5.2, XSS attackers can still leverage these types of websites for other purposes such as to track user locations. As more features are implemented into the service worker, this problem can only get worse if appropriate protections are not implemented correspondingly. We further discuss the number of XSS-vulnerable websites tied to each attack channel in their respective subsections, i.e., the IndexedDB channel in Section 6.3 and the push channel in Section 6.4.

### 6.3 Prevalence of IDB attack channel

Our measurement tool reported that there are 3,813 (of 7,060) websites with IDB access and 21% (828/3,813) of these websites load an IDB entry to use inside their service workers. More importantly, there are 40 information flows that reach a sensitive sink in the SW context. We find 5 flows reach the *importScripts* API and 35 flows reach the *setInterval* API.

**Confirming vulnerabilities.** We manually check these 40 sensitive flows and confirm that all 5 flows (corresponding to 5 different websites) that reach the *importScripts* API are vulnerable. We find that these 5 websites save a URL into the IndexedDB, and the corresponding IDB entry is read and passed into the *importScripts* API. We use Chrome's DevTools to test that when the URL is modified

**Table 2: A list of variable types of IndexedDB entries loaded inside a service worker.**

| Type | # Entries | Examples |
|---|---|---|
| Bool | 63 | true/false |
| Flag | 485 | persistNotification<br>emailAuthRequired<br>isPushEnabled |
| URL | 88 | https://www.eazydiner.com/<br>https://via.batch.com/2.1.0/worker.min.js<br>Albertonews.com |
| Push Key | 13 | dwRH5VdycN4:APA91bEgqyRo0t9R1hW9oqwJAjLk6MUL9QNQ<br>7fLhMSrXxS0-MWdkZBV3tqIbfMl633itH8bakis3L6HTIOZJ51<br>o_tAST-ogHg1XJTBHnJvY_E3sNSz0OdJvNEgCfOg2gfya-Ely2p_Mi |
| ID | 306 | 83AEAB70-31DF-2ADC-98F3-F0F365A753A1<br>f45438cb19044fd78277994b2231ddea<br>NY0C-5Skyo1ijcRfgddX_w |
| Title | 11 | Discover The Latest Fashion Trends |
| Numeric | 101 | 2.2, 1.2.0, 224 |
| Email | 3 | vibethemes@gmail.com |
| Others | 92 | America/Chicago<br>Chrome/77.0.3818.0 Safari/537.36<br>Sun Dec 22 2019 14:53:47 GMT-0600 (Central Standard Time) |

to our own host, we are able to hijack the service worker of these websites. Note that our test does not actually affect the websites as the test was done locally, in which we ourselves are the victim. We further confirm the other 35 sensitive flows regarding the *setInterval* API. Fortunately, these 35 flows are safe due to the tainted data being numerical and used solely to specify an interval for the API.

To measure the false-negative rate, we manually inspect 60 websites. We randomly select 30 websites from the set of websites that do not have IDB access based on our tool report. Then we randomly select another 30 websites from the set of websites that access IDB but do not contain a sensitive flow. We use Chrome's DevTool to interact with these 60 websites and inspect their source code. This process takes us approximately 15 human hours in total, which limits us to only perform this evaluation on a handful of websites. Overall, we find 7 websites from the first set actually access the IDB but only after we subscribe for push notifications or create a login account. This limitation is not specific to our tool, and previous work [22] that requires automated web crawling similarly faced the code coverage issue. While several techniques were proposed to improve web crawler, efficient and exhaustive web exploration under time-bound constraint remains a challenge, especially for rich web applications that require login credentials [23]. Nonetheless, we do not find any additional sensitive flow that our tool missed from these 60 websites. Therefore, we estimate that our false-negative rate is minimal.

Based on the 5 vulnerable websites, we notice the potential problem with the URL data type used in the IndexedDB. Therefore, we further investigate 828 websites that load an IDB entry inside their service worker. We use string-based heuristics to identify whether the data stored is a URL as summarized in Table 2. For instance, a string with only numbers and dots is considered numerical, a true/false string is Boolean, a string with only alphabet is likely a flag, a string with multiple spaces is textual or title, or a string with no spaces and special characters except underscore or dash is likely an ID. On the other hand, the patterns of URL, Push key, or email are more well-defined. We can use regular expression to match these data types more narrowly.

We find that there are 88 IDB entries from 88 websites that read a URL from the IndexedDB to use inside their service workers. We use our taint tracking tool, which is practically a web browser, to visit and further interact with each website. We check the taint information to see if there is any additional taint flow that can come from an unexplored path in the original analysis and use the Chrome Devtool to inspect the service worker's execution. Fortunately, there is no additional vulnerable website found.

We use OpenBugBounty to query the past records of reported XSS vulnerabilities on these five websites. We find that one website has an unpatched XSS vulnerability and three websites have records of XSS vulnerabilities that were patched. Such XSS vulnerabilities naturally allow XSS attackers to compromise the service worker through the IDB attack channel. In total, there are 5 appified websites that we can confirm as vulnerable (in which one is also exploitable) to the IDB attack channel. We have notified the five websites regarding the attack, and four of them eventually fixed the issues.

## 6.4 Prevalence of push attack channel

There are two types of push protocols: legacy and VAPID. First, the legacy protocol uses *gcm_sender_id* to identify the sender. The sender ID is normally shared between users of the same third-party library, thus we collect and measure the most common *gcm_sender_id*. Second, the VAPID protocol uses the *applicationServerKey* to identify the sender. However, the key is normally different between users of the same third-party library. Therefore, we identify the third-party library by grouping similar JavaScript files together and manually label them.

**Legacy push protocol.** Although Google has deprecated the Google Cloud Messaging (GCM), which utilizes the legacy protocol, in April 2018, there are still 359 websites supporting the legacy protocol. Among these websites, there are 4 popular libraries that they used to handle push notifications. The libraries are Aimtell, Insider, Feedifly, and Rich as shown in Table 3 (right). The websites that use these libraries are potentially vulnerable as the same *gcm_sender_id* may be shared among all accounts of these public push services, which attackers can also create an account. In the case of Youtube, its *gcm_sender_id* is shared within their own sub-domains, thus cannot be leveraged by any attackers.

**VAPID push protocol.** Websites that utilize the VAPID protocol are intrinsically vulnerable. Because there is no security policy that can regulate the allowed key used in the VAPID subscription, any third-party script can register its own key and easily hijack the victim website's push service. From Table 3 (left), the most popular VAPID push libraries are OneSignal, Izooto, Pushowl, Firebase, and Pushly. As a result, we mark these 5 libraries for further investigation.

Considering push libraries in both the legacy and VAPID protocols, there are 9 public libraries that may be leveraged by attackers. We further survey these libraries' account creation process and find that 5 libraries offer free account registration without requiring any personal identification as shown in Table 4 (Free-tier). Furthermore, they offer an equivalent feature to the location-triggered notification in their services (i.e., geo-segmenting). We regard these libraries as vulnerable as attackers can easily utilize the web interfaces of these libraries to track victims.

In total, there are 993 websites that utilize these 5 push libraries. We use OpenBugBounty [10] to search for websites with an XSS vulnerability among these 993 websites. Surprisingly, we find 200/993 websites with an unpatched XSS vulnerability. Attackers can easily hijack the push subscription of users who visit these websites by leveraging these XSS vulnerabilities.

We randomly select 40 websites from the 200 vulnerable websites to further verify how many websites could be used for location tracking. Because a website often does not ask for user location unless the user logs in or interact more with the website, we have to manually inspect this sample of websites and cannot automatically verify all 200 websites. We use our best effort to manually interact with them to see whether they will ask for the location permission. For example, we try to register an account and subscribe for push notifications (using Google translate when the website is non-English). Eventually, there are 14/40 (35%) websites that ask for location permission. These websites can allow attackers to utilize the location-triggered APIs to send user locations when the push subscription is hijacked and users grant the permissions.

**Table 3: A list of five most popular VAPID push libraries (left) and a list of five most common gcm_sender_id (right) in popular appified websites.**

| VAPID | Count |
|---|---|
| OneSignal | 854 |
| Izooto | 126 |
| Pushowl | 59 |
| Firebase | 37 |
| Pushly | 34 |
| Total | 1110 |

| Legacy | Count |
|---|---|
| 71562645621[Aimtell] | 28 |
| 912856755471[Insider] | 14 |
| 343259482357[Feedifly] | 9 |
| 402845223712[Youtube] | 7 |
| 361246025320[Rich] | 4 |
| Total | 62 |

**Table 4: A table of the pricing for top 7 push libraries.**

| Name | Free tier | Paid tier (per month) |
|---|---|---|
| OneSignal | 30K (devices) | $99 (unlimited) |
| Izooto | - | $85/30K (devices) |
| Pushowl | 500 (messages) | $19/10K (messages) |
| Firebase | Unlimited | - |
| Pushly | 100 (devices) | $15 (base) + 0.005/device |
| Aimtell | - | $49/10K (devices) |
| Feedify | 10 (message) | $25/3K (devices) |

To estimate the number of potential victims, we use SimilarWeb [14] to get the number of monthly visits of the 200 vulnerable websites. We find that there are over 1 billion visits in one month. According to a report from OneSignal [9], the most popular push library in our list, around 10% of visitors would subscribe to a push service, and 5% of subscribed users would interact with a push message. As subscribing for push notifications can be an indicator that these users are well-engaged with the websites and may also grant the location permission, we estimate the number of victims to be approximately 1.75M users (derived from 1 billion x 10% x 5% x 35%) per month. Note that this number does not represent the actual vulnerable users but only an *upper bound* estimation since the number of visits counts repeated users, and attackers still have to launch an XSS attack against these users.

Nonetheless, this estimation only includes the top 5 push libraries that attackers can easily utilize, and there are more than a thousand websites that use other libraries or implement their own. These websites can also be targeted, but they require different steps to reproduce the same attack based on the detailed implementation of each push library. As we cannot manually confirm the attack on all push libraries, we leave these websites out of our estimation.

## 7 DISCUSSION

### 7.1 Key observation from IDB attack channel study.

Although we can only confirm 5 vulnerable websites at this moment, we observe a worrying trend regarding this attack channel. We observe that the dynamic configuration of service workers, which is designed to be more or less static, is the root cause of the vulnerability.

We notice that the vulnerable websites utilize a third-party script to handle all of the service worker's implementation. These websites start a simple service worker that does not contain any functionality other than importing another third-party script. We refer to such third-party script as third-party service worker provider or SW provider in short. We speculate that the vulnerable appified websites use the IndexedDB to specify the path of the file being imported because the SW provider encourages them to do so. The provider likely has several service worker configurations corresponding to different service worker files that fit different types of customers (i.e., provider.com/sw-conf1.js and provider.com/sw-conf2.js). Therefore, instead of providing different starting SW files that import a different static URL to each customer, the providers use a common (but vulnerable) starting service worker file and let the customer dynamically choose the configuration through the IndexedDB.

Based on our further verification on the 88 websites that load a URL inside a service worker, we notice that the 5 vulnerable websites are not the only ones following this practice. Fortunately, the other SW providers have properly sanitized the IDB entry before passing it to the *importScripts* API. Once this type of service becomes more popular, and if SW providers do not take caution in sanitizing IDB entries (as we show in Section 5.1 that a security check could be bypassed), the IndexedDB attack channel can become more prevalent in the future.

## 7.2 Key observation from push attack channel study.

Based on our manual investigation of popular push providers, we find that major push providers do offer or advertise location-based features. For example, subscriber demographic can help provide the statistics needed to improve the business campaign, and location-triggered notifications can increase subscriber engagement especially for limited time/location events. Nevertheless, we only see such features currently implemented in a relatively small fraction of appified websites (i.e., 14/40 websites). As location-based features are widely used in other domains (i.e., for marketing and advertising) [2, 3], we speculate that the same trend will follow push notification and the number of appified websites utilizing such features will increase in the future.

Interestingly, we find that a large number of websites that we manually investigate currently use push messages abusively instead (i.e., to promote phishing messages or illegal services). Although attackers in our threat model can also hijack push subscriptions from legitimate websites to use push messages abusively, we do not consider this direction in this work. This is mainly because abusive messages will likely make users unsubscribe, causing attackers to lose control of the hijacked subscription. Nonetheless, as Chrome (starting from version 80) has started blocking push notification permission by default (instead of the "ask by default") for some users or websites, we believe that this kind of problem may be rather pervasive. Therefore, this problem may be a worthwhile research direction for future work.

Additionally, we observe that some push providers implement some forms of protection against push hijacking attacks (albeit it may be coincidental and a by-product from the API designs). For example, OneSignal prevents its subscribe API from being invoked twice. This should prevent attackers in our threat model from re-subscribing using the attacker account. However, by removing the service worker, we observe that the subscribe API can be invoked again, even though it produces some error/warning messages. Therefore, such client-side checks may not be enough to prevent this kind of attack and server-side checks may be a more reliable mitigation method.

## 7.3 Possible mitigation/defense

**IndexedDB.** To prevent attackers from utilizing the IndexedDB against the service worker, the most effective method is to sanitize the IndexedDB entries before using inside a sensitive function. However, it is extremely difficult to perfectly sanitize all inputs, which is why XSS attacks are still prevalent nowadays. Therefore, for a long term solution, we suggest an alternative method to initialize the service worker. For instance, a new type of cookie, SWOnly (Service-worker-only) cookie, that only allows access exclusively in a service worker can potentially mitigate the attack through the IndexedDB. Unlike the HTTPOnly cookies, which completely disallow script access, the SWOnly cookies would simply disallow script access from the document context (or normal web workers). We observe that the attacks against service workers usually occur during the installation phase, which still allows attacker to add sensitive event listeners (i.e., *FetchEvent*). By initializing the service worker through SWOnly cookies instead of the IndexedDB, attackers will not have a way to manipulate the internal SW variables during the installation phase anymore.

Originally, the service worker was designed to not need the cookie access. However, such design was soon proven wrong, and the Cookie Store API [5], which allows cookie access in the service worker, is under development to satisfy the needs from the developer community. Therefore, we expect that a mechanism like the SWOnly cookie may not be too far fetched in the future.

Another improvement that can help enhance the security of service workers is to provide a dedicated storage for service workers. Currently, service workers have to use the IndexedDB, which is shared between different contexts of the same origin. While this is useful for sharing data, especially for web workers that may need to sync parallel computation results, it limits service workers from storing sensitive data as untrusted scripts from a different context can freely access the IndexedDB. Although the SW context is isolated, the IndexedDB can be a weak link that invalidates the context isolation. In the future, it is possible that service workers may be used for a wider range of applications and require sensitive data to be stored locally, especially to still support offline usage. Therefore, it may be crucial for service workers to have an additional dedicated storage.

While we cannot provide a dedicated storage for service workers as a solution that can be easily deployed, we try to implement a prototype in Chromium to understand the feasibility and side effects of this improvement on web browsers. To this end, we manually inspect the source code of Chromium and find that the existing

**Table 5: A table of execution overhead occurred in the defense prototypes. In each entry, the first number represents the stock version execution time, and the second number (in parenthesis) shows the additional overhead of our implementation (taken as an average across 10 testing runs).**

| | Average (ms) | Min (Ms) | Max (ms) | Median (ms) |
|---|---|---|---|---|
| Push[Subscribe] | 156 (+40) | 141 (+22) | 185 (+46) | 154 (+35) |
| IDB[Open New] | 36 (+8) | 30 (+10) | 47 (+4) | 36 (+8) |
| IDB[Open Existed] | 2 (+7) | 2 (+4) | 2.5 (+12) | 2.3 (+6.8) |
| IDB[Store] | 0.5 (+2.5) | 0.4 (+1.1) | 1.1 (+4.6) | 0.5 (+1.8) |
| IDB[Read] | 0.2 (+0.4) | 0.1 (+0.3) | 0.8 (+0.9) | 0.2 (+0.4) |

IndexedDB API can be easily extended to provide a dedicated storage for service workers. We create another copy of an IDB Factory (back-end of the IndexedDB API). We link this copy with a new API that can only be accessed from the SW context. We name this API as *privateIDB*. This modification to Chromium requires less than 1K LoC to get a working version of the new API. Note that we only test the new API on basic usages in which it can provide the isolation without crashing. As our implementation is a proof-of-concept, the actual implementation may require more changes to the source code and more intensive testing.

To calculate the overhead of the new dedicated storage, we compare the modified Chromium with a baseline version. Specifically, we visit our website that simply open an IndexedDB and execute the *privateIDB* API (i.e., open, read, write). We record the run time of each API call and take the average between ten runs. Then, we repeat the same tasks using the baseline Chromium to execute the original IndexedDB API. Table 5 rows (2-4) show the overhead incurred by this modification. The first number in each cell refers to the baseline average run time, and the number in the parenthesis refers to the added time using the modified Chromium. Based on these numbers from our crude prototype, we believe that providing a dedicated storage for service workers is feasible. However, further optimization may be needed and planned out by the browser developer community to better provide a robust solution in a larger scale.

**Push subscription.** To prevent any script from using an arbitrary key for the subscription, one possible solution is to allow a website to specify allowed keys that can be used to subscribe for push notifications. For instance, web browsers can reserve a Manifest entry (i.e., *Allowed-Application-Server-Keys*), which contains a list of allowed keys. Then, when the *subscribe* API is used, the browsers can check if the specified key is allowed in the Manifest entry. However, a similar suggestion was raised in the developer community [4], but it was rejected due to possible usability issues. Nevertheless, given that push providers start to incorporate location-based message as a new standard, which can be utilized by attackers to track user locations, we believe the benefit is worth the adoption cost of this improvement.

As a proof of concept, we also implement a prototype for this improvement in Chromium. The changes we make to Chromium are no more than a few hundred lines of code, and we manually verify that it can successfully prevent a random key from being used without crashing. We repeat a similar evaluation (done with the *privateIDB*), and the overhead is shown in the first row of Table 5. Although the push subscribe API is rarely called and the overhead may not have a significant impact on a website, we acknowledge that such changes may introduce more complications on other aspects. Regardless, we urge the developer community to re-evaluate the push notification APIs given that service workers can enable new ways for attackers to utilize hijacked push subscription as discussed in Section 5.2.

Another possible method that push providers may employ to mitigate push hijacking is to check when two accounts are tied to the same website. During our manual investigation, we observe that push providers will ask for the website URL that wants to provide push notifications, while creating a new project (or app). However, as discussed in Section 5.2, we are able to make two separate accounts (a benign and an attacker account) link to the same website. By preventing another account from linking it to a website that is currently tied to another account, attackers will not be able to easily utilize the push providers anymore. Although this only prevents attackers from utilizing the push providers, it forces the attackers to use the native API such that the attackers have to implement the back-end push server by themselves. These extra steps can potentially chase away attackers due to the gain is not worth the extra effort. We suggest push providers consider this method in addition to any existing client-side checks to further enhance the defense against future attacks.

## 8 CONCLUSION

In this work, we examined the communication channels of service workers. We discovered two attack channels, namely, the IndexedDB and push notification that attackers can use to leverage a benign service worker. To measure the impacts of these attack channels, we conducted an analysis on 7,060 popular appified websites. Our findings showed that approximately 1.75M users per month can have their locations tracked by an attacker who leverages benign service workers of popular websites. As more websites start adopting service workers without considering potential risks, we believe this problem can worsen. Therefore, we discussed our key observations and possible defense solutions to help mitigate this problem from growing in the future.

## ACKNOWLEDGMENTS

This material is based upon work supported by the NSF/VMware Partnership on Software Defined Infrastructure as a Foundation for Clean-Slate Computing Security (SDI-CSCS) program under Award Title "S2OS: Enabling Infrastructure-Wide Programmable

Security with SDI" and No. 1700544. It is also supported in part by ONR Grant No. N00014-20-1-2734. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF and ONR.

## REFERENCES

[1] [n.d.]. Android Intent. https://developer.android.com/reference/android/content/Intent.
[2] [n.d.]. AWS Location-based Marketing Report 2018. https://s3.amazonaws.com/factual-content/marketing/downloads/LocationBasedMarketingReport$_F$actual.pdf.
[3] [n.d.]. AWS Location-based Marketing Report 2019. https://s3.amazonaws.com/factual-content/marketing/downloads/Factual-2019-Location-Based-Market-Report.pdf.
[4] [n.d.]. Chromium Push Issue. https://bugs.chromium.org/p/chromium/issues/detail?id=803106.
[5] [n.d.]. Cookie Store API. https://wicg.github.io/cookie-store/.
[6] [n.d.]. Geofencing on push notification. https://retailtouchpoints.com/features/executive-viewpoints/geofencing-and-mobile-push-notifications-a-match-made-in-customer-engagement-heaven.
[7] [n.d.]. Kaspersky Report on Stalkerware. https://www.kaspersky.com/about/press-releases/2019$_c$ould-someone-be-spying-on-you-through-your-phone.
[8] [n.d.]. Location-triggered notification. https://documentation.onesignal.com/docs/location-triggered-event#section-web-setup.
[9] [n.d.]. OneSignal Report. https://onesignal.com/blog/increase-opt-in-rates-for-push-notifications/.
[10] [n.d.]. OpenBugBounty. https://openbugbounty.org/.
[11] [n.d.]. Pushwoosh geo-based notification. https://www.pushwoosh.com/blog/geo-based-push-notifications/.
[12] [n.d.]. PWA Checklist. https://developers.google.com/web/progressive-web-apps/checklist.
[13] [n.d.]. Shadow Worker. https://shadow-workers.github.io/.
[14] [n.d.]. SimilarWeb. https://www.similarweb.com//.
[15] [n.d.]. Stalkerware. https://www.cyberscoop.com/stalkerware-pandemic-coronavirus-domestic-violence/.
[16] Phakpoom Chinprutthiwong, Raj Vardhan, GuangLiang Yang, and Guofei Gu. 2020. Security Study of Service Worker Cross-Site Scripting.. In *Annual Computer Security Applications Conference* (Austin, USA) *(ACSAC '20)*. Association for Computing Machinery, New York, NY, USA, 643–654. https://doi.org/10.1145/3427228.3427290
[17] Chong Guan, Kun Sun, Zhan Wang, and Wen Tao Zhu. 2016. Privacy Breach by Exploiting postMessage in HTML5: Identification, Evaluation, and Countermeasure. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, Xiaofeng Chen, XiaoFeng Wang, and Xinyi Huang (Eds.). ACM, 629–640. https://doi.org/10.1145/2897845.2897901
[18] Soroush Karami, Panagiotis Ilia, and Jason Polakis. 2021. Awakening the Web's Sleeper Agents: Misusing Service Workers for Privacy Leakage. In *NDSS*.
[19] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2017. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society.
[20] Jiyeon Lee, Hayeon Kim, Junghwan Park, Insik Shin, and Sooel Son. 2018. Pride and Prejudice in Progressive Web Apps: Abusing Native App-like Features in Web Applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. ACM, New York, NY, USA, 1731–1746. https://doi.org/10.1145/3243734.3243867
[21] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 Million Flows Later: Large-Scale Detection of DOM-Based XSS. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA, 1193–1204. https://doi.org/10.1145/2508859.2516703
[22] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. 2018. Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society.
[23] Seyed M Mirtaheri, Mustafa Emre Dinçktürk, Salman Hooshmand, Gregor V Bochmann, Guy-Vincent Jourdan, and Iosif Viorel Onut. 2014. A brief history of web crawlers. *arXiv preprint arXiv:1405.0749* (2014).
[24] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. You are what you include: large-scale evaluation of remote javascript inclusions. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, Ting Yu, George Danezis, and Virgil D. Gligor (Eds.). ACM, 736–747. https://doi.org/10.1145/2382196.2382274
[25] Panagiotis Papadopoulos, Panagiotis Ilia, Michalis Polychronakis, Evangelos P. Markatos, Sotiris Ioannidis, and Giorgos Vasiliadis. 2019. Master of Web Puppets: Abusing Web Browsers for Persistent and Stealthy Computation. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society.
[26] Sooel Son and Vitaly Shmatikov. 2013. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society.
[27] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. 2019. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society.
[28] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. 2017. How the Web Tangled Itself: Uncovering the History of Client-Side Web (In)Security. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 971–987.
[29] Takuya Watanabe, Eitaro Shioji, Mitsuaki Akiyama, and Tatsuya Mori. 2020. Melting Pot of Origins: Compromising the Intermediary Web Services that Rehost Websites. https://doi.org/10.14722/ndss.2020.24140