Attacks from Third-party Scripts in Service Workers Compromise Web Security

XXX
XXX@com
XXX
Rocquencourt, France

XXX
XXX@com
XXX
Rocquencourt, France

ABSTRACT

Service workers (SWs) are becoming popular due to their powerful features; they run asynchronously in the background and enable offline web application features. However, these features can be abused by compromised service workers to intercept requests and responses between users and servers. In recent years, security researchers have uncovered a variety of attacks: attacker-controlled service workers can sniff history, launch botnets, and inject scripts. Most existing threat models assume attackers can inject code to service workers via existing cross-site scripting (XSS) vulnerabilities in the web application.

This paper systematically analyzes APIs accessible to service workers and identifies a novel threat model that assumes attackers can provide third-party scripts imported by service workers. We reveal three new attacks—resource exhaustion attacks, which freeze the browsers; Cache poisoning attacks, which load attacker-controlled pages; and Cookie hijacking attacks. We implement proof-of-concept attacks and successfully test them on all major browsers. We estimate the impact of our threat model by measuring how many websites use importScripts API from Alexa top 100K websites. We discuss mitigation strategies such as using more reputable 3rd party scripts and leveraging Subresource Integrity (SRI).

CCS CONCEPTS

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

KEYWORDS

Service workers, Resource Exhaustion attack, SW Cache poisoning, Cookie hijacking, importScripts

ACM Reference Format:
XXX and XXX. 2022. Attacks from Third-party Scripts in Service Workers Compromise Web Security. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

Conference acronym 'XX, June 03–05, 2022, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Service workers (SW) are JavaScript that executes in the client's origin [42]. Service workers are a key component of the Progressive Web Apps [12, 20]. They shorten the page loading times [10], enable offline access to boost page performance [9, 17], and push notifications to remind users [13, 27]. A service worker can intercept network requests and responses made by a client [24], redirect the request to the service worker cache (SW Cache), and serve the response from the SW Cache [3]. In general, if request assets are stored in SW Cache, the service workers return the assets to the client rather than requesting from the server [4]. With the Background Sync API, service workers can synchronize the data with servers [23]. Thus service workers significantly improve the user's browsing experience. Many websites have started to use service workers to improve performance [19, 26]. According to previous works [39], 5918 (5.9%) out of Alexa top 100k websites register a service worker.

As a result of service workers' capabilities, attackers can launch attacks via compromised service workers. Papadopoulos et al. [37] showed attackers can use service workers to launch botnets and mine cryptocurrency inside victims' browsers. Lee et al. [33], and Karami et al. [32] discovered side-channel attacks to sniff victims' browsing history. Existing threat models assume attackers can inject scripts to service workers via XSS attacks [7].

In this work, we propose a new threat model where attackers host or compromise third-party domains and trick developers into including malicious third-party scripts in service workers via the importScripts API. We systematically analyzed all APIs that service workers have access to and identified three new client-side attacks: Resource exhaustion attacks, SW Cache poisoning attacks, and Cookie hijacking attacks.

For Resource exhaustion attacks, attackers can inject code that contains an infinite loop to the service worker, which will consume enough CPU resources to cause the victim's browser to freeze. As a result, victim users cannot change tabs or close tabs. In some cases, the problem persists even after the browser restarts. For SW Cache poisoning attacks, attackers can inject malicious code to change the SW Cache policies, then poison the SW Cache with malicious content. The malicious SW can intercept all HTTP requests from the client and alter them from SW Cache based on attackers' needs. Previous works [38] show that via Cache API, attackers can steal credentials if attackers have XSS capabilities to the target websites. In comparison, our threat model doesn't rely on existing XSS vulnerabilities, and our proposed attacks still work with their threat model assumptions. For cookie hijacking attacks, attacker-controlled third-party scripts contain code that leverages cookieStore API [5, 6], which is accessible to the service

worker. With the help of cookieStore API, attackers can alter cookies inside the victim user's browser and steal cookies from them. We implemented proof-of-concept attacks on browsers from major vendors (i.e., Safari, Opera, Chrome, Edge, Firefox) on both macOS and Windows. For resource exhaustion attacks where we found that the Opera browser incorrectly handles resource limitation, we reported the issues to the Opera browser.

To evaluate the impact of our threat model, we measure the number of websites that use service workers and the number of websites that import third-party scripts out of Alexa top 100k websites. We found that for 5211 websites that use service workers, 2860 websites use importScripts API [28]. Among 2860 of them, 782 websites import scripts from third-party domains. We also discovered that many developers use unknown domain's third-party scripts rather than APIs provided by large companies like google provided API [1]. To help developers navigate this problem, we implemented a tool called import-checker to show developers information about the third-party domains that they import scripts from. We also propose using the subresource integrity (SRI) [22] to prevent attacks caused by compromised third-party domains. We will open-source our tools and dataset.

This paper makes the following contributions:

- A novel threat model of service workers.
- A systematic analysis of SW security.
- New attacks found based on our threat model.
- A tool to inform developers of third-party domains from which they import scripts.

2 BACKGROUND

In this section, we first introduce service workers, service workers' lifecycle, and browser support. Next, we provide a description for importScripts API, which is an API that can be misused by developers and exploited by attackers. Then, we explain the service worker cache and cookieStore API in our threat model.

A service worker is a web worker which runs in the background and provides offline features. A web worker contains three types: dedicated workers, shared workers, and service workers. It runs on a different thread from the main thread, so it is asynchronous and non-blocking. Even if the browser is closed, the service worker can still run in the background. Service workers can act as a proxy between browsers and web servers. A service worker can intercept network requests and responses made by a client, redirect the request to the service worker cache, and serve the response from the service worker cache. It can enable offline features and improve user experience [17]. However, it can not access DOM directly, but it can communicate with DOM via MessageChannel API [11].

A service worker can only control web pages whose URLs are in the same domain as the service worker. The default scope of the service worker is the './' of the relative URL. As a result, attackers cannot simply inject service worker scripts from attackers' domains into target websites. Moreover, the service worker only runs over HTTPS, except on LocalHost. Therefore, it is hard for a network attacker to directly compromise service workers.

Figure 1 shows the service worker's lifecycle. The first step is registration, coded in the application's JavaScript file, so registration happens whenever the web page is loaded. A service worker is

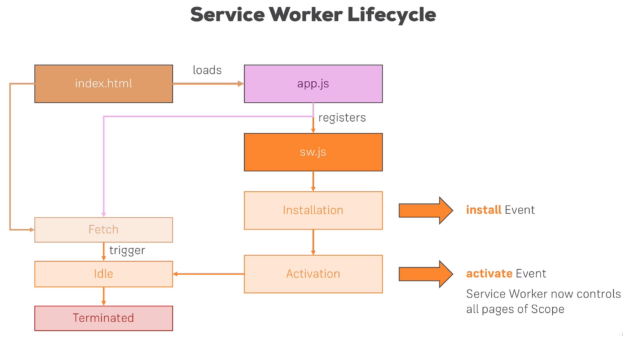

Figure 1: Service worker lifecycle

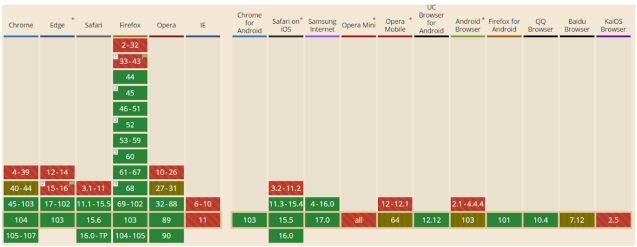

Figure 2: Browser support for service workers

downloaded when the registration cycle is finished. Moreover, if the service worker fails to download, the website will not use the service worker and will act normally in the user's browser. After the completion of registration, the installation will be triggered once the service worker is executed. In the installation steps, the developer can cache everything they need to service worker cache. It is worth mentioning that installation only happens if the service worker has never been registered or the script is changed, even with only 1 byte. Otherwise, it will not be re-install if there are no changes in the service worker. The last step is activation, which does not happen immediately after installation. The activation will only happen when no other old version service workers remain active. Developers can use self.skipWaiting() to prevent the waiting, which means the service worker activates as soon as it is finished installing [16]. Service workers use event listeners to catch events like fetch, notification events, and response with developer-controlled actions.

Service workers can also be unregistered by users and developers. Users can stop, unregistered, or block service workers as their needs because the service worker runs in the background asynchronously, sometimes costing CPU. A service worker can run in the background even when the browser is closed, so users can either stop the service worker or close the running process by using an activity monitor.

Service workers are supported on the most dominant modern browsers like Chrome, Edge, Safari, Firefox, and Opera. Also, Safari on IOS and Chrome on Android support service worker, and they are mainly used in progressive web apps. Even if a browser does not support service workers, it will not crash the websites, and users can still browse the context without service worker features. A detailed of browser support can be found in Figure 2.

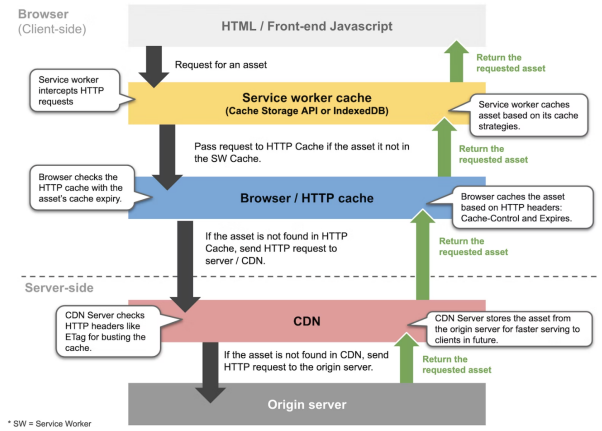

Figure 3: Service workers Cache

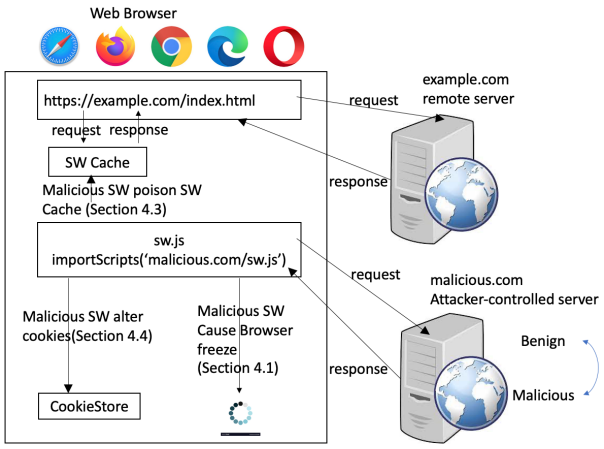

Figure 4: Threat Model

	MacOS		Windows10	
	Killed by OS	Time to be killed	Killed by OS	Time to be killed
Chrome v.103	—	>10 min	Yes	3 min
Firefox v.103	Yes	1min 30 sec	—	>10 min
Edge v.103	Yes	9 min	Yes	2 min 45 sec
Opera v.89	Yes	10 min	Yes	10 min
Safari v.15.6	Yes	3 min	NA	NA

Table 1: Browsers vulnerable to Resource Exhaustion for different OSs (Notes: time varies for different version of OSs and Browsers)

ImportScripts API is designed for web workers, including service workers. ImportScripts API can be used to import other scripts to load additional scripts to the service worker context before the installation lifecycle completes. Compromised service workers can use importScripts API to import additional attacker-controlled third-party scripts.

The service worker manages its own cache via Cache Storage API [4]. Service workers with this API can be used to load content faster. A general communication between client and server is shown in the Figure 3 [14, 21]. If the client sends an HTTP request, the service worker intercepts the request and checks the asset hit inside the service worker cache; if the asset exists, it is returned to the client directly. If the asset does not exist, the service worker will pass it to the browser's HTTP cache and check for availability, and if it still does not exist, the browser will send the HTTP request to the server and wait for responses.

The Cookie Store API is an asynchronous API to manage cookies, exposing cookies to service worker [5, 6]. Traditional cookie API is accessed by document.cookie [8], which is synchronous and single-threaded. Moreover, since service workers cannot access DOM directly, they cannot access cookies directly. Therefore cookieStore API provides an alternative way that service workers can access and change cookies inside service workers.

3 THREAT MODEL

In our threat model (Figure 4), we assume attackers can provide malicious service worker scripts to a target website (e.g., example.com) either by hosting or compromising the third-party domain (e.g., sw-provider.com) that provides service workers' scripts that the target website imports. We assume developers import the third-party scripts using `importScripts` API. We assume developers would test the correctness and safety of service workers before publishing them. While developers are not maintaining and checking the source code constantly, attackers can change their scripts from benign to malicious. These target websites are susceptible to attacks once they import attacker-controlled scripts. When users visit target websites that import attacker-controlled service worker scripts, attackers can compromise the secrecy and integrity of users' data by stealing cookies and credentials and tamper with the webpage resources, and run unauthorized computation, consuming resources on users' machines.

4 ATTACKS FROM IMPORTED SW SCRIPTS

In this section, we describe three novel attacks, Resource Exhaustion/IP blocked attacks, SW Cache poisoning attacks and Cookie hijacking attacks.

4.1 Resource Exhaustion attack

For Resource Exhaustion attacks, attackers can inject malicious code that contains an infinite loop to the service worker, which causes the victims' browsers to freeze. When Resource Exhaustion attacks happen, victim users cannot change or close tabs. For some browsers, victim users can't close the frozen browser. Even If the victim closes the browser, the attacks still run in the background and consume CPU resources.

We run our proof of concept attack on different browsers and the results are shown in Table 1. We measure the time it took the OS to kill the browser process due to heavy resource usage. We only observe up to 10 minutes. Google Chrome has the largest number of users of all browsers [25]. Resource Exhaustion attacks on Google Chrome running on macOS last more than 10 minutes, which exists longer than other browsers on macOS.

We found that Resource Exhaustion attacks persist across browser shutdowns on the Opera browser. This is due to the fact that if Opera is terminated abnormally (e.g., by Operating System or Activity

Monitor), when users re-open Opera, Opera will restore the previous tabs. This issue happens on both macOS and Windows. We discover that even using activity monitor or rebooting machines will not stop Resource Exhaustion attacks on Opera. When the user re-opens the browser, the vulnerable websites will be restored, and Resource Exhaustion will be re-activated. We disclosed this issue to Opera.

4.2 IP blocked attack

Attacker can also inject code that sends a large number of HTTP requests to target websites and cause the victim's IP address to be blocked by those websites. As a result, when the user visits those websites they will get error pages (e.g., Too Many Requests, Access Denied) instead of normal pages.

Our experiments show that 7 (i.e., netflix.com, microsoft.com, adobe.com, tiktok.com, sharepoint.com, intuit.com, blogspot.com) out of the Alexa top 100 websites will block the user's IP. In some cases, the IP was blocked by these websites for at least an hour. In our experiment, service workers can send 1000 requests within 3 seconds using Fetch API. All these seven websites block our IP within 1000 requests. Many lower-ranking websites will block the victim's IP within 400 requests (e.g., pacsun.com).

4.3 SW Cache poisoning attack

In this attack, we assume victims visit a benign website (i.e., example.com), which imports attacker-controlled third-party scripts using importScripts API. When user's browser makes an HTTP request for main HTML pages, normally browser sends the request to example.com (remote server) and, fetches responses from the server, then provides them to the users. With SW Cache poisoning attacks, as shown in Figure 4, attackers fetch attacker-controlled pages and cache them in the victim's browser. When victims make requests, the SW Cache will provide attacker-controlled pages to victims.

SW Cache poisoning attacks can alter images, CSS style, and JavaScript code in DOM and even the entire HTML page. For images, attackers can alter benign images with phishing ads or illegal contexts to harm users. With attacker-controlled CSS style, attackers can use customized CSS style to exploit CSS-related vulnerabilities. For instance, in 2015, CSS was found to have the problem of allowing attackers to track users' browsing history. With malicious JavaScript code, attackers can launch persistent-XSS attacks within victims' browsers. Most importantly, attackers can alter the whole HTML page through this attack. In attacker-controlled pages, attackers can sniff victims' history by leveraging history API and monitor victims' screens using navigator.mediaDevices.getDisplayMedia() API, or get victims' locations, credentials by manipulating the web pages. Attacker-controlled web pages can ask victims for permission to access cameras, microphones, locations, file editing, and other sensitive permissions. Most of these permissions are only asked one time, and when victims allow such permissions to compromised web pages, victims' secrecy and privacy are compromised. Although service workers cannot access DOM, with this attack, attackers can compromise DOM directly and alter DOM attributes and properties.

Our SW Cache poisoning attacks have fewer restrictions than related works. Squarcina et al. [38] show that their cache poisoning can steal users' credentials. They assume attackers host the same domain website as the benign websites, while we do not have this restriction. Our proposed attack still works on their attack model assumption. In our attacks, as long as benign websites finish importing third-party scripts using importScripts API, the SW Cache poisoning attacks can be launched even offline. Because of this offline feature, victims still visit malicious pages that are served from the poisoned SW Cache. If attackers leverage background sync features in a service worker, an offline action performed by victim users will take effect when the application is online. Attackers can combine the SW Cache poisoning attacks with background sync features to get more assets.

Our study shows that malicious context stored in SW Cache can last for 30 days [15]. Our experiments demonstrate that we can successfully alter all loaded resources, including the entire HTML pages on all five browsers on macOS and Windows. In theory, this attack can be launched on all browsers that support service workers and SW Cache API.

4.4 Cookie hijacking

Attacker-controlled third-party scripts can use cookieStore API, which is accessible to the service worker. With the help of cookieStore API, attackers can alter cookies for benign websites that import attacker-controlled scripts. Previous works did not explore this API that service workers can access. We divide cookie hijacking attacks into two categories, cookie stealing attacks and cookie tampering attacks. Note that cookies let websites remember users, their website logins, or other credentials (e.g., Single sign-on (SSO) cookie [2]). With victims' session cookies, attackers can launch session hijacking attacks, login into benign websites without victims' usernames and passwords, or steal victims' identities [8]. Therefore, Cookies are sensitive data that should be protected.

For cookie stealing, attackers can get all cookies under the benign websites that use importScripts API to import attacker-controlled scripts from victim users. They can get a single cookie value with their corresponding key name, or they can get all cookies to their needs. For most websites, if users allow cookies, they do not have to type in their login information every time they log in. Cookies can store their information and send them to remote servers. With these cookies, attackers can pretend to be victim users and perform user-unwanted actions (e.g., purchasing unwanted items). For cookie tampering, attackers can change current cookies' settings, for example, cookie name/value pairs. Other sensitive settings like expiration time and same site settings. Another scenario is that attackers can put their own cookie values inside the compromised websites. Therefore, the user's action is not equal to their identity.

In our cookie tampering (stuffing) attacks, we can change the victim's cookies to the attacker's, and the victim's action is now connected to the attacker's identity. In our cookie stealing attacks, we successfully steal victims' all cookies and use them to log in to real-world websites with the victim's identity (i.e., Amazon). One caveat is that if the operation is sensitive (e.g., changing passwords, buying and selling), Amazon still needs the user to provide the password.

	Attacker can read and write cookies
Chrome	Yes
Opera	Yes
Firefox	No
Safari	No
Edge	Yes

Table 2: Browsers vulnerable to cookie hijacking

Finally, we discuss browsers that are vulnerable to cookie hijacking attacks. We show our result in Table 2. For all five browsers that we use for macOS and Windows, we find that Chrome, Opera, and Edge are vulnerable to such attacks. Firefox and Safari are not vulnerable because they do not support cookieStore API.

5 MEASUREMENT OF SW USAGE

In this section, we present the results of the service worker usage and importScripts API usage measurement studies and discuss the impact of our threat model.

5.1 Service worker usage

We crawled the top 100k websites based on the Alexa ranking list, and identify websites using service workers by checking service workers' registration (Listing 1).

Listing 1: Code snippet for checking service workers registration

```
const puppeteer = require('puppeteer');

(async () => {
  const url = "https://www.example.com";
  const browser = await puppeteer.launch({
    headless: false,
  });
  await page.goto(url);
  const swTarget = await browser.waitForTarget(
    (target) => {
      return target.type() === 'service_worker';
    },
    { timeout: 30000}
  );
})();
```

Table 3 summarizes results from our crawl and comparing it with prior work. We observe that the amount of websites of service workers is decreasing. There are two possible reasons. The first reason is the variability of the sites crawled. All of these studies use the Alexa top 100K websites, which is maintained by Amazon. However, Amazon retired Alexa list on May 1, 2022. As a result, some websites on that list cannot be accessed anymore. Compared to previous results [30], we found that the second reason is that some websites that supported service workers do not support service workers anymore. For instance, on the top Alexa 100 websites list, pinterest.com and www.okezone.com supported service workers, but they do not support service workers anymore. It could be that

Source	# of websites with SW	# % of websites with SW	Year
Phakpoom et al. [30]	6182	6.18%	2019
Subramani [39]	5918	5.90%	2021
This paper	5211	5.21%	2022

Table 3: SW usage

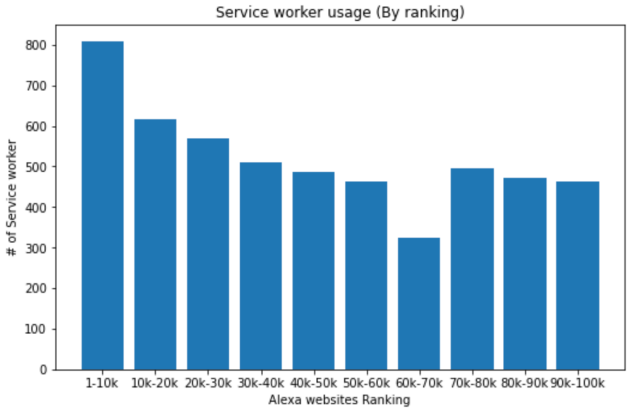

Figure 5: SW usage by ranking

these developers moved away from using service workers because more code means more bugs, and misusing service workers can be a problem. Since the other two works did not provide their ways of measuring the number of service workers and other results, we cannot make a detailed comparison.

Overall, we measure that 5.21% of the Alexa top 100k websites use service workers. We show them by their ranking in Figure 5. The x-axis is the ranking, and the y-axis is the number of websites that have service workers. The top 10k websites on Alexa list have more service workers than other ranges. As shown in Figure 5, websites ranking between 60k to 70k have a dip; this is due to many websites are not reachable in this range. Moreover, this is also the reason that they have less websites that use importScripts in their service workers in Figure 6.

5.2 ImportScripts measurement and feasibility discussion

Our work found that for 5211 websites that use service workers, there are 2860 websites using importScripts API. For example, example.com import 3party.com script in their service worker source code. Among 2860 of them, 782 websites import third-party domain scripts. 231 of them import more than one third-party scripts. Some of them import six different third-party scripts.

Table 4 lists the domain owners for these third-party scripts. Domain owners can indicate which organization own these third-party scripts, which can serve as an indicator of how trustworthy these scripts are.

We rank the domain owners by the number of websites that import scripts from them. In Table 4, we observe that Google LLC is the most popular one, accounting for 42.7% of the third-party

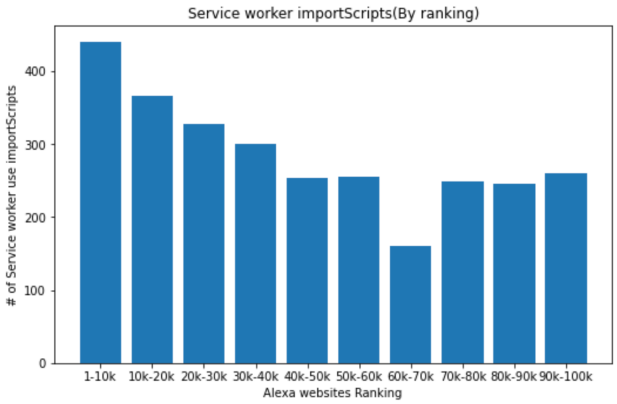

Figure 6: SW importScript by ranking

Org	# of websites	% of websites
Google LLC	461	42.7%
None	305	28.2%
Domain registrar	194	18.0%
Other companies	114	10.5%
Amazon Inc	7	0.6%

Table 4: Domain owners for third-party scripts

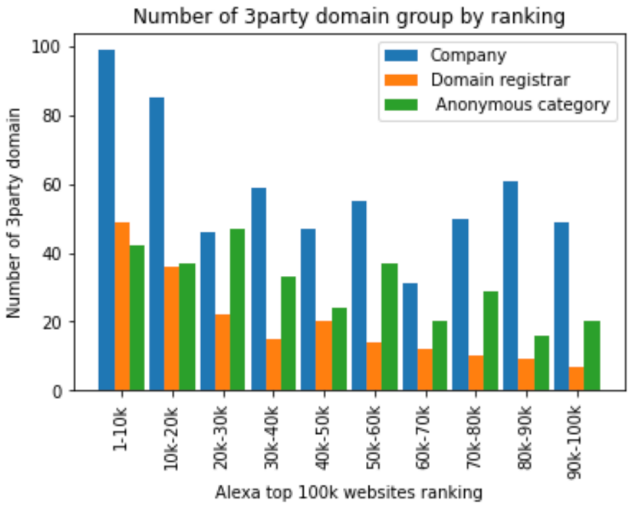

Figure 7: Third-party domain owners by ranking

Org	# of websites	% of This category
Companies	582	100%
Google LLC	461	79.2%
Other companies	114	19.6%
Amazon Inc	7	1.2%

Table 5: Domain owners for Companies

scripts. Then, there are some domain owners that are anonymous or lacking information about the owners, which accounts for 28.2% of the third-party scripts. After that, Domain registrar, the domain owners that register domain names for others such as Domains By Proxy LLC, Whois Privacy Service, accounts for 18% of the scripts. Then we have a long tail of companies that account for under 0.6% of the third-party scripts, so we lump them together into the "Other companies" category. Finally, Amazon owns the second largest amount of third-party scripts after Google at 0.6%. A detailed breakdown of all the companies is shown in Table 5.

We categorize domain owners into three categories: Known Companies, Domain registrars, and Unknown. The known company category is for companies with a known presence that host these scripts. For example, Google LLC is the largest domain owner in this category; 461 of Alexa top 100k websites use Google-provided scripts. Other companies like Amazon, Adobe, Quora, Netease are also in this category. Domain registrars Category is for the Domain registrars (e.g., Whois Privacy Protection Service Inc.), which can hide domain owners' information. The Unknown category is when domain owners are unknown (i.e., None). For the domain owner with None, it's hard to track down these domain owners, and 305 websites use these third-party scripts as their source code. Attackers in our threat model could lie in both the Domain registrar and Unknown category to hide their identity.

Our results in Figure 7 show that even top 10K ranking websites import Domain registrar and Unknown category's script.

6 COUNTER MEASURES

The fundamental problem that our attack model points out is the lack of trust in imported third-party scripts. To mitigate these issues, we recommend two approaches that developers can take to gauge and improve the trustworthiness of the scripts that they import. The first approach is to know and understand who and where the imported scripts come from. To help developers, we developed a tool (import-checker) that can display the scripts' domain and the domain owners of their imported scripts. Our tool import-checker also provides a domain owner list from our crawl of the Alexa top 100k websites. Developers can decide whether they want to trust the third-party scripts based on a number of websites that import from the same (types of) domains.

Second, we recommend that developers monitor the third-party scripts' changes. To do so, we propose using Subresource Integrity (SRI) to prevent attackers from changing imported scripts. Subresource Integrity is a security feature that enables browsers to verify that the resources they fetch (for example, from a CDN) are delivered without unexpected manipulation [22]. Developers can compute the integrity hash for their trusted service workers' code and put it in script tags inside DOM. If attackers tamper with service workers, the SRI hash will change, and integrity violation occurs, then the browser will refuse to execute the scripts [22].

Table 6: Current works for SW

SW Abuse Categories	Attacks	Abuse Vectors											AT capabilities	AT Impact	AT exists when Offline
		Push API	Notification API	Sync API	Performance API	Update API	IndexDB API	Cache API	Fetch API	CookieStore API	ImportScripts	iframe inclusion			
Continuous Execution	WebBot [37]			✓		✓							a	Botnet, cryptomining	
	PushExe [33, 39]	✓	✓										a, c	Reactivate sw	✓
	Resource Exhaustion								✓		✓		a,b,d,e	Browser freeze	✓
Side Channels	OfflineOnload [33]											✓	a	History sniffing	
	PerformanceTiming1 [32]				✓							✓	a	History sniffing	
	PerformanceTiming2 [32]				✓							✓	a	History sniffing	
Push API and Notification Abuse	Phishing [33]	✓	✓										b, c	Click malicious url	✓
	Malvertising [40]	✓	✓										b, c	Read malicious ads	✓
	Stalkerware [31]	✓											b, c	Track user's location	
Cache API and Cookie hijacking	Cache API Abuse [38]							✓	✓				e	XSS	
	SW Cache Poisoning							✓	✓		✓		a,b,d,e	Persistent XSS	✓
	Cookie Hijacking								✓	✓	✓		a,b,d,e	Cookie hijacking/Re-writing	✓

Bold name attacks are our attacks
Assumption:
a. Victims visited attacker-controlled websites. b. Attackers control SW. c. Victims allow push notification permission to attackers. d. Attackers host benign third-party SW scripts. e. Benign websites allow attackers to host a malicious website on the same domain.
AT exists:
With a check mark, it means this kind of attack doesn't need network connections. Otherwise, the attack needs network connections to be consistent.

7 RELATED WORK

7.1 SW security

Previous work Threat model Papadopoulos et al. [37] assume target websites' service workers are compromised (e.g., by importing malicious third-party libraries inside service workers). In their threat model, another case is that attackers can register malicious service workers to compromised websites. Therefore, users who visit these websites are vulnerable to their attacks. Lee et al. [33] and Subramani et al. [39] assume that attackers control the malicious PWAs and their service workers.

Karami et al. [32] assume attackers have capabilities to inject code to target websites. Squarcina et al. [38] assume attackers can inject code to target websites or websites that are the same domain as target websites. Subramani et al. [40] assume service workers are compromised. Our attacks also work under these assumptions. Compared to their works, we assume target websites are benign. Because if target websites are malicious, we do not need service workers to perform the subsequent attacks.

Other threat models are attackers are web attackers [30, 31, 39], most of them are fixed by developers or service worker maintainers.

Previous works' attacks We summarize our and previous works' attacks in Table 6. We divided them into four categories: Continous Execution, Side Channels, Push API and Notification Abuse, Cache API and Cookie Hijacking.

For Continous Execution, Papadopoulos et al. [37] discuss how a webBot can be built inside a service worker. They assume victims visit malicious websites that register a malicious SW, and attackers use SW to control the victim's machines to make DDoS attacks on remote servers or do cryptocurrency mining in the victim's machines. Both of These attacks run dedicated works behind the browser via service workers. However, the Chromium maintainer claims that [18] service worker will be terminated if dedicated works are running in the background. Their threat model describes attackers can control victims' machines through victims' service workers and launch distributed denial-of-service (DDoS) to remote servers. At the same time, our resource exhaustion attack happened on the client side directly with the effect of freezing the victims' browsers. For PushExe attacks, Lee et al. [33] and Subramani et al. [39] describe attacks in that attackers can reactivate the SW using showNotification API and then perform cryptocurrency mining even though SW is stopped. It requires attackers to register malicious service workers to vulnerable websites and trick users into allowing push notifications to these vulnerable websites. As far as we know, these attacks are fixed or mitigated by most browsers.

For side-channel attacks, Lee et al. [33], and Karami et al. [32] propose side-channel attacks to sniff users' browsing history.

For Push API and Notification Abuse, Lee et al. [33] propose an attack model in that attackers first register malicious SW that have malicious push notifications, then trick victims into clicking push notifications to download malicious code or click the URL in push notifications using phishing attacks. Subramani et al. [40] show malicious advertisements can be included in push notifications for malvertising. Moreover, Chinprutthiwong et al. [30] show that attackers can leverage push notifications to get victims' locations by using third-party libraries like OneSignal.

For Cache API and Cookie Hijacking, Squarcina et al. [38] propose cache poisoning attacks against SW cache. In their attacks,

they focus on injecting malicious scripts to HTML pages, whereas our attacks alter the whole HTML pages.

Other than the above attacks, related works discuss how to compromise service workers. Chinprutthiwong et al. find vulnerable websites that let URL parameters [31] or context in indexDB [30] passed directly to service workers' importScripts API, which leads attacker-controlled input to be service workers' code. They claim that these vulnerabilities are fixed. Subramani et al. [39] propose an attack in which an attacker-controlled Firefox extension can inject malicious SW code in the scope of any victim origin. This vulnerability was fixed on December 7, 2021, by Firefox [39]. They also propose a library hijacking attack, while benign websites include third-party code to manage push notifications. They found a third-party library that misused push notifications to track all web pages visited by the user.

7.2 Third-party script

Third-party scripts are commonly used in practice. Agent et al. [29] show that more than half of the Alexa top 10k websites include scripts from more than Five different origins. However, third-party scripts are not always benign. Mayer et al. [35] reveal that third-party scripts can track users' privacy and sell them to advertising companies.

Previous researchers have presented many tools to sandbox third-party JavaScript code or detect vulnerable code. Terrace et al. [41] present tools to allow third-party scripts to be executed inside a sandbox and isolated environment. Agent et al. [29] propose a sandboxing framework that enforces the server-specified policy on third-party scripts and ensures the scripts carry no risks. Musch et al. [36] propose a tool that prevents third-party scripts to perform unsafely string-to-code conversions. Luo et al. [34] present a framework that restricts third-party script execution under the host web page's instructions. These tools may not work for service workers as they would restrict service workers' key functionalities.

8 CONCLUSION AND FUTURE WORK

We proposed a new threat model for service works, where attackers control scripts imported via importScripts API. We identified three new attacks and tested proof-of-concept attacks on browsers from major vendors. We found that a large fraction of websites that import scripts from domains that attackers may be able to compromise. For mitigation, we implemented a tool that presents developers with useful information about the third-party domains that they import scripts from. We also propose to leverage SRI to improve the trustworthiness of the imported scripts. As future work, we plan to examine service work security in the context of mobile devices: progressive web app (PWA) specifically. PWAs have access to additional phone resources, which could allow the attacks to cause more damage compared to the desktop environment.

REFERENCES

[1] 2022. API Reference. https://developer.chrome.com/docs/workbox/reference/. Accessed 5 July, 2022.

[2] 2022. Automatic cookie single sign on on multiple domains - like google. https://stackoverflow.com/questions/11434866/automatic-cookie-single-sign-on-on-multiple-domains-like-google. Accessed 5 July, 2022.

[3] 2022. Cache. https://developer.mozilla.org/en-US/docs/Web/API/Cache. Accessed 1 June, 2022.

[4] 2022. Caching. https://web.dev/learn/pwa/caching/. Accessed 5 July, 2022.

[5] 2022. Cookie Store API. https://developer.mozilla.org/en-US/docs/Web/API/Cookie_Store_API. Accessed 5 July, 2022.

[6] 2022. CookieStore. https://developer.mozilla.org/en-US/docs/Web/API/CookieStore. Accessed 5 July, 2022.

[7] 2022. Cross Site Scripting (XSS). https://owasp.org/www-community/attacks/xss. Accessed 5 July, 2022.

[8] 2022. Document.cookie. https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie. Accessed 5 July, 2022.

[9] 2022. Making PWAs work offline with Service workers. https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Offline_Service_workers. Accessed 5 July, 2022.

[10] 2022. Measuring the Real-world Performance Impact of Service Workers. https://web.dev/service-worker-perf. Accessed 5 July, 2022.

[11] 2022. MessageChannel. https://developer.mozilla.org/en-US/docs/Web/API/MessageChannel. Accessed 5 July, 2022.

[12] 2022. Progressive web apps (PWAs). https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps. Accessed 5 July, 2022.

[13] 2022. Push API. https://developer.mozilla.org/en-US/docs/Web/API/Push_API. Accessed 5 July, 2022.

[14] 2022. Service worker caching and HTTP caching. https://web.dev/service-worker-caching-and-http-caching. Accessed 5 July, 2022.

[15] 2022. Service worker caching and HTTP caching. https://web.dev/service-worker-caching-and-http-caching. Accessed 5 July, 2022.

[16] 2022. The service worker lifecycle. https://web.dev/service-worker-lifecycle. Accessed 5 July, 2022.

[17] 2022. Service worker overview. https://developer.chrome.com/docs/workbox/service-worker-overview. Accessed 5 July, 2022.

[18] 2022. Service Worker Security FAQ. https://www.chromium.org/Home/chromium-security/security-faq/service-worker-security-faq. Accessed 5 July, 2022.

[19] 2022. Service Worker Usage Statistics. https://trends.builtwith.com/javascript/Service-Worker. Accessed 5 July, 2022.

[20] 2022. Service workers. https://web.dev/learn/pwa/service-workers. Accessed 5 July, 2022.

[21] 2022. Strategies for service worker caching. https://developer.chrome.com/docs/workbox/caching-strategies-overview. Accessed 5 July, 2022.

[22] 2022. Subresource Integrity. https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity. Accessed 5 July, 2022.

[23] 2022. Synchronize and update a PWA in the background. https://learn.microsoft.com/en-us/microsoft-edge/progressive-web-apps-chromium/how-to/background-syncs. Accessed 5 July, 2022.

[24] 2022. Use Service Workers to manage network requests and push notifications. https://learn.microsoft.com/en-us/microsoft-edge/progressive-web-apps-chromium/how-to/service-workers. Accessed 5 July, 2022.

[25] 2022. User population of selected internet browsers worldwide from 2014 to 2021 (in millions). https://www.statista.com/statistics/543218/worldwide-internet-users-by-browser. Accessed 5 July, 2022.

[26] 2022. What Are Service Workers and How They Help Improve Performance. https://www.keycdn.com/blog/service-workers. Accessed 5 July, 2022.

[27] 2022. What You Should and Shouldn't Be Using Push Notifications For. https://onesignal.com/blog/what-you-should-be-using-push-notifications-for. Accessed 5 July, 2022.

[28] 2022. WorkerGlobalScope.importScripts(). https://developer.mozilla.org/en-US/docs/Web/API/WorkerGlobalScope/importScripts. Accessed 5 July, 2022.

[29] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H Phung, Lieven Desmet, and Frank Piessens. 2012. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference*. 1–10.

[30] Phakpoom Chinprutthiwong, Raj Vardhan, GuangLiang Yang, and Guofei Gu. 2020. Security Study of Service Worker Cross-Site Scripting.. In *Annual Computer Security Applications Conference*. 643–654.

[31] 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*. 312–323.

[32] Soroush Karami, Panagiotis Ilia, and Jason Polakis. 2021. Awakening the Web's Sleeper Agents: Misusing Service Workers for Privacy Leakage. In *Network and Distributed System Security Symposium*.

[33] 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*. 1731–1746.

[34] Wu Luo, Xuhua Ding, Pengfei Wu, Xiaolei Zhang, Qingni Shen, and Zhonghai Wu. 2022. ScriptChecker: To tame third-party script execution with task capabilities. (2022).

[35] Jonathan R Mayer and John C Mitchell. 2012. Third-party web tracking: Policy and technology. In *2012 IEEE symposium on security and privacy*. IEEE, 413–427.

[36] Marius Musch, Marius Steffens, Sebastian Roth, Ben Stock, and Martin Johns. 2019. Scriptprotect: mitigating unsafe third-party javascript practices. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. 391–402.

[37] Panagiotis Papadopoulos, Panagiotis Ilia, Michalis Polychronakis, Evangelos P Markatos, Sotiris Ioannidis, and Giorgos Vasiliadis. 2018. Master of web puppets: Abusing web browsers for persistent and stealthy computation. *arXiv preprint arXiv:1810.00464* (2018).

[38] Marco Squarcina, Stefano Calzavara, and Matteo Maffei. 2021. The remote on the local: exacerbating web attacks via service workers caches. In *2021 IEEE Security and Privacy Workshops (SPW)*. IEEE, 432–443.

[39] Karthika Subramani, Jordan Jueckstock, Alexandros Kapravelos, and Roberto Perdisci. 2022. SoK: Workerounds-Categorizing Service Worker Attacks and Mitigations. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 555–571.

[40] Karthika Subramani, Xingzi Yuan, Omid Setayeshfar, Phani Vadrevu, Kyu Hyung Lee, and Roberto Perdisci. 2020. When push comes to ads: Measuring the rise of (malicious) push advertising. In *Proceedings of the ACM Internet Measurement Conference*. 724–737.

[41] Jeff Terrace, Stephen R Beard, and Naga Praveen Kumar Katta. 2012. {JavaScript} in {JavaScript}(js. js): Sandboxing {Third-Party} Scripts. In *3rd USENIX Conference on Web Application Development (WebApps 12)*. 95–100.

[42] W3C. 2022. Service Workers. https://www.w3.org/TR/service-workers. Accessed 12 July, 2022.