THE SERVICE WORKER HIDING IN YOUR BROWSER: NOVEL ATTACKS AND DEFENSES IN APPIFIED WEBSITES

A Thesis

by

PHAKPOOM CHINPRUTTHIWONG

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Guofei Gu
Committee Members,	Dilma Da Silva
	Jeff Huang
	Alexander Sprintson
Head of Department,	John Keyser

August 2022

Major Subject: Computer Science

ABSTRACT

The service worker (SW) is an emerging web technology that was introduced to enhance the browsing experience of web users. At the core, it is essentially a JavaScript file that runs in an isolated and privileged context separated from the main web page or web workers. Websites can register a service worker to enable native mobile application features including but not limited to supporting offline usage and sending push notifications. With the help of this technology, traditional websites can now act like native mobile apps or become *appified*. Recently, the use of service workers has gained much attention from web developers, security researchers, and even cyber-criminals due to the service worker's unique capabilities, especially the ability to intercept and modify web requests and responses at runtime. Such capabilities inevitably introduce new factors to web security considerations.

The goal of this research is to systematically study both the vulnerabilities and the security enhancement to websites that can come with the introduction of service workers. The contributions of this dissertation are three folds. First, we investigate the service worker lifecycle and uncover a vulnerability allowing cross-site scripts to be executed inside the service worker. We term this novel attack as Service Worker Cross-Site Scripting (SW-XSS) and develop a dynamic taint tracking tool to measure the impact of SW-XSS in the wild. Second, we analyze the communication channels between the service worker and other web contexts. We identify two vulnerable channels, IndexedDB and Push notifications. These channels can be utilized to launch SW-XSS and push hijacking attacks, which can lead to the privacy leakage of users. Third, we propose and develop a framework, SWAPP (Service Worker APplication Platform), for implementing security appliances by leveraging the unique capabilities of a service worker. Not only can SWAPP prevent the aforementioned attacks against service workers but also be used to implement defense mechanisms for traditional web attacks such as Cross-Site Scripting (XSS), data leakage, or side-channel attacks. We develop several defenses for traditional attacks using SWAPP and show that they are easier to develop, have lesser installation requirements, and are effective compared to existing solutions.

DEDICATION

To my family and friends:

Thank you for your support and encouragements.

ACKNOWLEDGMENTS

First and foremost, I would like to express my gratitude to my advisor, Dr. Guofei Gu, especially for his understanding when I was in difficult times and his trust in my capability even when I had self-doubt. Without his trust and guidance, I would have given up on this journey and would not have persevered. He has influenced me to become better professionally and as a person. I hope that I can pass my experience forward to my future students similar to what he has done for me.

I would also like to thank other members of my committee, Dr. Dilma Da Silva, Dr. Alexander Sprintson, and especially Dr. Jeff Huang for his optimism and affability that make me feel at ease when I am thousands miles away from home.

I have also been very fortunate to have a supportive family. I thank my mother and father for bringing me into this world and all the love and encouragements ever since. You have named me with the meaning of "proud", and I am happy that I can once again make you proud.

I would like to thank my friends, Paprapee Buason for being my personal therapist in my darkest and brightest time, Kittipong Somchat for all the valuable experience, and all other friends who supported me during my Ph.D.

Lastly, I would like to thank the SUCCESS lab members, Jialong Zhang, Abner Mendoza, Guangliang Yang, Haopei Wang, Lei Xu, Kevin Hong, Jingyu Hong, Raj Vardhan, Jianwei Huang, Molly Lan, Vicram Rajagopalan, Shreyas Kumar, Huancheng Zhou, Nathan McClaran, and lastly my brother Yangyong Zhang. Their friendships are greatly appreciated, I wish them the best on their own journey.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This dissertation work was supported by a dissertation committee consisting of Professor Guofei Gu (advisor), Professor Dilma Da Silva, and Professor Jeff Huang of the Department of Computer Science & Engineering and Professor Alexander Sprintson of the Department of Electrical & Computer Engineering.

Chapter 3 is originated from a research work co-authored by Raj Vardhan, Dr. Guangliang Yang, and Dr. Guofei Gu, and was published in Proceedings of the 2020 Annual Computer Security Applications Conference (ACSAC'20).

Chapter 4 is originated from a research work co-authored by Raj Vardhan, Dr. Guangliang Yang, Dr. Yangyong Zhang, and Dr. Guofei Gu, and was published in Proceedings of The 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'21).

Chapter 5 is originated from a research work co-authored by Jianwei Huang, and Dr. Guofei Gu, and was published in the 31st USENIX Security Symposium (USENIX'22).

Funding Sources

My graduate study was 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 NSF Grant No. 1617985, 1642129, and 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.

NOMENCLATURE

SW	Service Worker
PWA	Progressive Web App
XSS	Cross-Site Scripting
SW-XSS	Service Worker Cross-Site Scripting
W3C	World Wide Web Consortium
JS	JavaScript
SWAPP	Service Worker APplication Platform
CORS	Cross-Origin Resource Sharing
VAPID	Voluntary Application Server Identification
FCM	Firebase Cloud Messaging
GCM	Google Cloud Messaging
DOM	Document Object Model
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
API	Application Programming Interface
CSP	Content Security Policy
MITM	Man-In-The-Middle
IDB	IndexedDB
UI	User Interface
TCB	Trusted Code Block
CDP	Chrome Devtool Protocol
CLI	Command Line Interface

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Over the past decades, the Internet, web, and mobile have become an integral part of our lives, and our online presence has become as important as our physical presence. Undeniably, the Internet brings us much convenience. Now, we can connect to different people across the world in almost real-time, access and perform banking operations within a few clicks, or anonymously express our opinions that can reach out to millions of people. However, these also make our online presence become a prime target for attackers. For instance, our banking accounts may get stolen, or our anonymous accounts may get de-anonymized by attackers.

A majority of cyber attacks can come in the form of client-side web attacks. These attackers often search for vulnerabilities in a website code and leverage them to compromise sensitive data and privacy of unfortunate visitors who are usually tricked into visiting the vulnerable websites through social engineering. With every web technology being released, a new vulnerability may also be introduced.

In this chapter, I first introduce the service worker, a recent web technology that can transform websites and the web security paradigm. I explain why, at the same time, it can be a threat and also an improvement to the web security ecosystem. Then, I outline the research challenges and clarify the goals I want to achieve, which mainly come in three folds: identifying a vulnerability, measuring the impact on a large scale, and providing security enhancements. Next, I provide an overview of our solution to address the challenges. Finally, I summarize the contributions of this thesis and outline the organization of the remaining chapters.

1.1 Service Worker: Transforming Web into App

Traditionally, a digital product (i.e., website, application, software) needs a different development pipeline. For instance, web developers work on a website, mobile developers work on an app, and desktop developers work on different versions of the same software that would run on different operating systems. Much development effort is wasted on different platforms and they

are not transferable. As a result, several companies are moving toward web-centric products because websites are cross-platform. However, the problems with websites are that they cannot work offline, do not support instant push notifications, and lack native UI like mobile apps or desktop software.

To solve these limitations, the service worker is introduced with the capabilities to intercept network traffic (to provide a cached response when offline) and to be reactivated at any time (to display spontaneous push messages). The service worker (SW) is essentially a JavaScript file that runs in a special web environment, known as the *service worker context*, as part of the background thread of a browser. This background thread runs separately from the web page's rendering thread (known as the *document context*) and the browser interface, thus service workers could be reactivated even when the browser is closed. When a website registers a service worker, its control covers all pages and resources loaded that originate from the same origin. For example, suppose `example.com` registers a service worker, the service worker can control `example.com/pageA.html`. Under a proper setting, if pageA.html tries to fetch `third-party.com/resources` then the resources can also be under the service worker's control. These unique features from service workers enable the *appification* of websites so that they can provide web users an experience similar to that of using a mobile app or desktop software. As a result, I will also refer to SW-enabled websites as *appified* websites.

As a service worker runs in a unique execution context and provides several unique features, it is gaining attention from the research community. Lee et al.[1] were the first to discuss how attackers can abuse malicious service workers to run a crypto-mining script. In this attack, Lee showed that by periodically sending push messages to reactivate the malicious service worker, the attackers can make a profit from using the client's resource to mine cryptocurrency. Similarly, Papadopoulos et al. [2] improved Lee's proposed attack by utilizing the background synchronization API, which can reactivate the malicious service worker automatically. Because the background synchronization API does not produce push messages like Lee's attack, this attack was considered more stealthy. In another type of attack, Watanabe et al. [3]

discovered that re-hosted web services (i.e., web translation and web proxy) are vulnerable to web hijacking attacks. Specifically, all re-hosted websites share the same origin (i.e., `https://rehost.com?target=example.com`), which can be controlled by a single malicious service worker registered for the domain `rehost.com`. Once the attackers register the malicious service worker for a victim's client accessing the re-hosted web service, all different origins re-hosted by the service will be hijacked by the attackers.

Existing studies on service workers have two elements in common. First, they often 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 the malicious service worker, the victim has to keep visiting the malicious website (or the attackers must periodically reactivate the malicious service worker as discussed by Lee and Papadopoulos). Second, they only discussed security threats but never explored the potential of using service workers to enhance the security of websites.

1.2 Research Challenges and Goals

In this dissertation, I intend to explore two research directions that were not discussed in previous work. The goals of this study are as follows:

(G1): I want to show that despite the security considerations of service workers discussed in the W3C specifications[1], benign website's service workers can be compromised and leveraged. This assumption is more generalized than previous work in two aspects. First, Lee and Papadopoulos's assumptions only work when the victim visits a malicious website, and the impact is simply the victim's computational resources being utilized illegitimately. On the other hand, compromising a benign service worker means the attackers can fully control the benign website. The impact of such attacks is at least as potent as persistent Cross-site scripting attacks, which often lead to user accounts or sensitive information (i.e., log-in credentials) being stolen. Second, the attack discussed by Watanabe only works in re-hosted web services, not in any vulnerable benign website.

[1] `https://www.w3.org/TR/service-workers-1/`

Table 1.1: Summary of Service Worker Attack Surfaces and Consequences

Attack Surface	Surface Type	Attack Type	Consequences
S1. SW Registration	Lifecycle	SW-XSS	Persistent code execution, sensitive information leakage, phishing attack, etc.
S2. IndexedDB	Communication Channel	SW-XSS	Persistent code execution, sensitive information leakage, phishing attack, etc.
S3. Push Message	Communication Channel	Push Hijacking	Information leakage such as user locations.
S4. Cache	Communication Channel	Side-Channel	Information leakage such as user browsing history.
S5. postMessage	Communication Channel	XSS	Code execution, sensitive information leakage, phishing attack, etc.

(G2): I want to patch the security flaws in service workers and show that the service worker can be utilized by web developers to deploy security appliances to prevent client-side web attacks. Generally, web defenses are deployed as parts of the web server [4, 5], browser extensions [6, 7, 8], browser modifications [9, 10, 11], or document context [12, 13]. However, the service worker context is never considered despite its unique capabilities that hold several advantages over previous methods. First, the service worker can intercept all requests and responses, which cannot possibly be achieved by web servers without additional costs to deploy an additional (reverse) proxy. Second, the service worker is installed automatically when a user visits the website, unlike browser extensions that require users to manually install in order to be protected. Third, the SW context runs apart from the document context, which hosts several scripts from different origins. Previous work showed that third-party scripts are commonly embedded in the document context (including those that are vulnerable) [14]. Once a script in the document context is compromised, all scripts including the defense mechanisms deployed could also be rendered ineffective.

1.3 Solution Overview

In this dissertation, I divide the solutions into two major parts corresponding to each goal (G1 and G2).

For goal G1, we systematically analyze the service worker lifecycle and communication channels that it utilizes to connect with other contexts. Figure 1.1 shows the overall attack surfaces that come with a service worker. The attacks and their consequences are summarized in Table 1.1.

Surface S1 corresponds to the service worker registration lifecycle. We analyze the registration

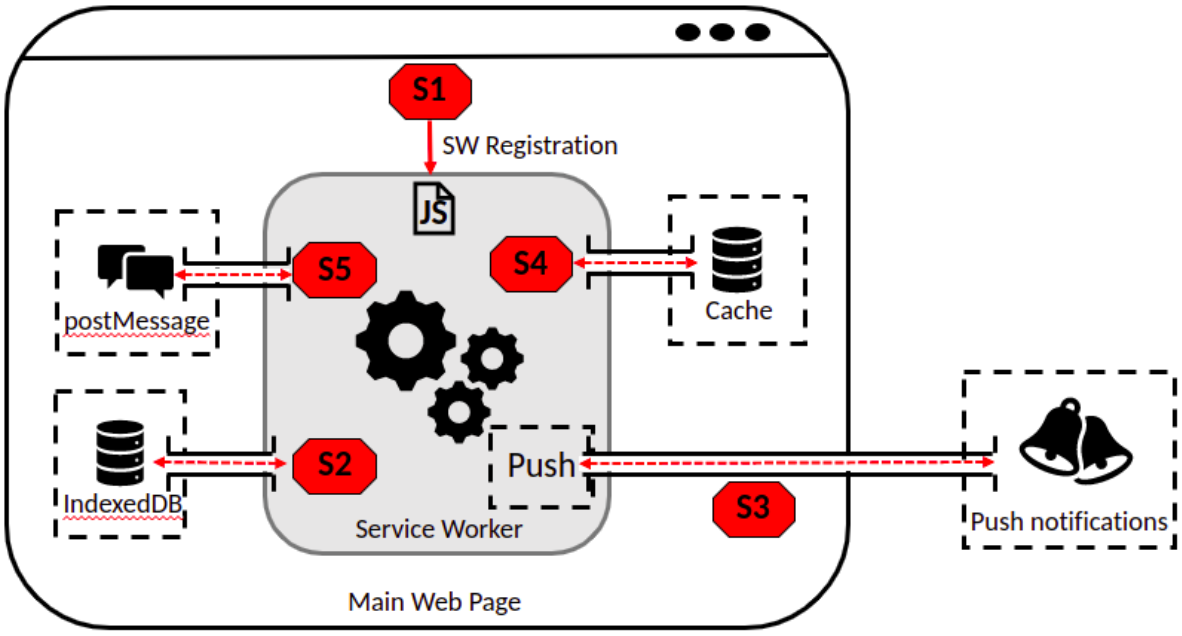

Figure 1.1: An Illustration of Service Worker Attack Surfaces.

process of service workers and discover that attackers can utilize URL search parameters specified in the SW's *register* API to compromise the service worker. We term this attack Service Worker Cross-Site Scripting (SW-XSS). To measure the prevalence of SW-XSS vulnerability, we implement a web crawler to collect service worker files and parameters registered by the top 100K appified websites (based on Tranco ranking [15]). Next, we develop a dynamic taint analysis tool, SW-Scanner, to detect SW-XSS vulnerability in appified websites. SW-Scanner taints the SW registration API parameters and leverages forced execution to improve the code coverage of our scanner.

Surfaces S2 and S3 correspond to the communication channels between the service worker and the main web page. Our manual investigation shows that there are four channels (postMessage, IndexedDB, push notification, and Cache) that can be used to communicate with a service worker. However, Cache and postMessage are studied by previous work [16, 17], thus we focus on the remaining two channels: IndexedDB and push notification. To this end, we extend the Chromium Taint Tracking project [11] to also taint IndexedDB input/output and collect push subscription

information during the analysis. The extended tool identifies potential data flows that can reach inside the secure service worker context.

For goal G2, we propose a platform for developers to implement security appliances in the service worker. We refer to the framework as SWAPP (Service Worker APlication Platform). SWAPP is implemented as a JS library for a service worker. Because the service worker context can be vulnerable through the registration and communication APIs (S1-S5), we enhance the security of service workers using SWAPP. Specifically, we use SWAPP to instrument code to regulate the APIs related to S1-S5. Furthermore, SWAPP provides development interfaces to also implement security apps that can prevent multiple types of web attacks.

1.4 Dissertation Contributions and Organization

In summary, this dissertation has three contributions.

(1) We analyze the service worker registration process and identify a new type of web attack, which we term SW-XSS (Service Worker Cross-Site Scripting). We implement a dynamic taint analysis tool, SW-Scanner, and identify SW-XSS vulnerabilities in the wild. The result shows SW-Scanner can identify 40 vulnerable websites that attackers can fully take control of via the SW-XSS attacks. (Chapter 3).

(2) We analyze the communication channels between the service worker context and the main page. We find two channels that can allow attackers to manipulate the secure service worker context: IndexedDB and Push APIs. We extend a taint tracking tool to identify potential vulnerabilities of these two channels in the wild. The result shows that the extended taint tracking tool can identify 200 vulnerable websites with 1.75M user visits per month. (Chapter 4).

(3) We propose a security application development framework utilizing a service worker, SWAPP (Service Worker APplication Platform). We show that SWAPP can mitigate attacks against the five attack surfaces S1-S5 and be used to implement various types of web defenses with low cost and high programmability. Specifically, there are four web defenses that we develop as examples, which can prevent XSS attacks, side-channel attacks, data leakage, and Autofill abusing attacks. (Chapter 5).

2. BACKGROUND AND RELATED WORK

2.1 Background

In the early stage of web development, a single processor's thread was used to handle all the needs of a website, such as handling UI events and manipulating the DOM. However, modern websites offer rich functionalities which require running several tasks simultaneously, such as processing a large amount of data while keeping the UI responsive. For such needs, a single thread was not enough to ensure a smooth web browsing experience for users. This led to the development of web workers to handle concurrent tasks. Ultimately, a web worker is JavaScript code that runs in a different thread to handle delegated concurrent tasks that do not require user interaction.

A service worker is a type of web worker. It runs in an event-based manner in a background thread that is separated from the main web page. Unlike other types of web workers, the service worker contains a set of unique features as shown in Table 2.1. A service worker supports two core features: offline usage and instant push notifications. As a result, a service worker can modify HTTP requests/responses of the corresponding website to serve an appropriate web page when the network is offline. Furthermore, it can be activated any time, regardless of whether the main page is open, to instantly display a push message that may arrive spontaneously. Additionally, once registered, a service worker can persist across sessions. These are the unique traits of a service worker as compared to other web workers.

2.1.1 Service Worker Lifecycle

For a website to utilize a service worker, it has to first fully operate securely in HTTPS. Then, the website can call the *navigator.serviceWorker.register* API to register a service worker. This API accepts two parameters: the file path of a service worker and the scope that the service worker can control. The first parameter is required, but the second parameter is optional. When the scope parameter is not provided, the default scope is the current path, allowing the registered service worker to control HTTP traffic of web pages under the current path. Once the *register* API is

Table 2.1: A List of Service Worker Events.

Event	Dispatch Condition
install	A service worker is installed
activate	A service worker is activated
fetch	A network request is issued
push	Receive a notification
notificationclick	A notification is click
notificationclose	A notification is closed
sync	Network is available
canmakepayment	A payment request can be handled
paymentrequest	A payment is requested
message	Receive a postMessage
messageerror	Cannot receive a postMessage

called, the browser will download, parse, and execute the specified service worker file. If the file is new or has changed from the previous version, the browser will install the new service worker. Otherwise, the browser will simply reactivate and return the current service worker. A successfully registered service worker will go through the *install* and *activate* lifecycle events. We illustrate the installation process in Figure 2.1. Note that in some cases, the website can install itself as a mobile app, which is also called Progressive Web App (PWA). This only happens when a Manifest file, which is the PWA metadata, is also provided.

Install. This event only occurs once per lifecycle during its initial execution. A website can add the *install* event listener to handle this event and use this opportunity to execute any preliminary tasks such as caching resources. When the browser is installing a new service worker, it allows event handlers to be added to the service worker. These event handlers include *fetch, push,* and *message*, which can be used to control HTTP traffic, handle push messages, and communicate through the *postMessage* API respectively.

Activate. This event is dispatched when the installed service worker is activated and becomes fully functional. Once a service worker is activated, its event handlers will be ready to handle the corresponding events. The activated service worker can operate until it is put into *idle*. When its main page is closed, the service worker will be put idle within a short period of time (usually less

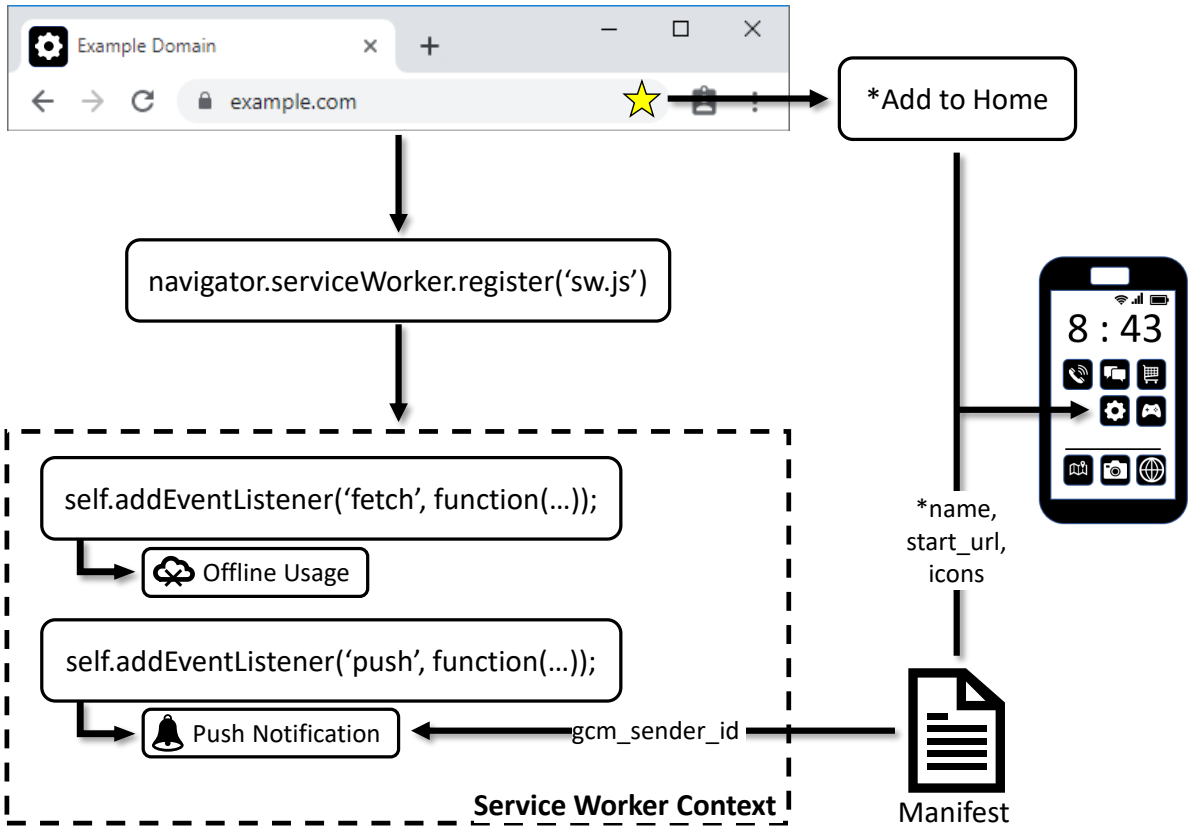

Figure 2.1: Service worker Installation Process.

than a minute). All ongoing tasks will be frozen until an event such as a push message's arrival is dispatched, and then the service worker will be activated again.

2.1.2 Service Worker Communication Channels

There are four communication channels (postMessage, Cache, IndexedDB, and push message) between the document and SW contexts.

postMessage. This communication channel is the most direct method. It allows communication to flow between the document and SW contexts and also between different iFrames. As studied by Guan et al. [18] and Son et al. [17], this communication channel can be leveraged by attackers to attack iframes, in which the same techniques can be applicable to the SW context. The attackers can also 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). While we do not

identify novel attacks utilizing postMessage, we take the lessons learned from previous attacks and provide an enhanced postMessage API in our proposed framework SWAPP's Message Manager module.

Cache. The Cache is shared by both the document and SW contexts to provide offline usage to an appified website. This channel can be used by attackers, i.e., to determine the state of the victim through response timings [16]. Generally, the service worker stores static assets in the Cache and the stored assets will be loaded in the document context to improve performance or support offline access. However, the difference in loading time between cached and non-cached resources can be used to infer a user's browsing history. Therefore, we take this observation and develop a countermeasure as a SWAPP app, Cache Guard (further discussed in Section 5.5.1).

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 another website's access. 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 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*. 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. We illustrate a practical attack using the IndexedDB in a real-world case study in Section 4.2.

Push message. Notifications shown in appified websites closely resemble native app push messages, but they are handled by the browser (e.g., Chrome) instead of the operating system (e.g., Android). For a website to provide push notifications, it must ask for a user's permission before subscribing the user through a specific push subscription protocol. Generally, the subscription protocol and the website's code to handle push messages are provided through third-party services.

We find that such practice can lead to user privacy leakage, especially leaking the user locations as discussed later in Section 4.3.

2.1.3 Service Worker Security

As a service worker contains unique privileges that other contexts do not have, it is designed with security as one of its priorities. Naturally, an attacker cannot easily compromise a benign service worker due to the service worker's built-in safeguards. Here we discuss the built-in security that an attacker could face while targeting a benign service worker and how the attacker may circumvent the corresponding protections.

First-party only registration. A browser only allows a first-party file to be registered as a service worker. This ensures third-party scripts embedded in the document context will not register their own script as a service worker. However, it does not prevent the registered service worker from importing an additional script from an external domain through the *importScripts* API. Therefore, this API can still create an opportunity for an attacker to launch an SW-XSS attack as we later discuss in Chapter 3.

Order of execution. A service worker runs mostly in an event-based environment, thus the privileges are provided in the form of events that can be handled. For instance, the *fetch* event is used to handle network traffic, and the *push* event is used to handle push messages. The list of all events currently supported by service workers is shown in Table 2.1. Most of these event handlers can only be added (using the *addEventListener* API) during the *install* lifecycle. Once the installation is finished, the browser will deny any attempt to register a new event listener. Similarly, an event (except the *message* event) cannot have more than one listener attached to it. Therefore, the goal of attackers is to add event handlers before the legitimate code adds its own handlers.

When an attacker fails to add an event listener, the impact of the attack is greatly limited. The injected malicious code would not gain any privileges and it will only get executed when the service worker is activated (i.e., when the website itself is visited), which is no different than compromising the document context. In this scenario, to indirectly influence the handler, the malicious code could still try overriding existing functions inside the service worker that will be

called by an event handler.

While the SW-XSS attack heavily relies on the order of execution of the malicious code, we find that it is not difficult to launch an attack in practice. As we will later show in Section 3.4.3, websites with service workers often add event listeners at a later stage after having imported additional scripts. This action of importing additional scripts is actually the root cause of the SW-XSS vulnerability. As a result, the current trend in how websites implement their service workers surprisingly favors the SW-XSS attackers.

Service worker's freshness. Generally, a web browser will constantly check a registered service worker and compare it to the hosted service worker file to make sure the service worker is up-to-date. When there is a different version available (i.e., a byte difference between the files is detected), the old service worker will be replaced. Therefore, an attacker who may have hijacked the old service worker will lose control of it.

Although this security mechanism can theoretically help prevent an attacker from keeping control of a hijacked service worker for a long period of time, there are two reasons why it is insufficient in practice. First, this check of freshness does not include the imported files. Even when an attacker manipulates or replaces an imported file, the browser will not replace the service worker as long as the service worker file itself does not change. Second, websites may not update the service worker as often as expected. As we later show in Section 3.4.4, our measurement indicates most websites are stale and rarely update their service workers in practice. This provides attackers an opportunity to circumvent this safeguard and compromise a benign service worker.

2.2 Cross-Site Scripting Attack

Cross-site scripting or XSS attack is one of the most common types of web attacks due to the simplicity with which it can be launched (e.g., requires minimal interaction with the victim) and its immediate impact. As a result, several forms of XSS attacks and the corresponding countermeasures were proposed.

Typically, XSS attacks are a type of code injection generally in the form of client-side scripts (e.g., JavaScript), which come from a malicious cross-domain source. The XSS attackers exploit a

flaw that allows inputs, usually in the form of URL parameters, to reach a sensitive function (such as *eval*) without proper sanitization. There are three common types of XSS.

Stored-XSS. An attacker crafts and navigates to a URL with a parameter that will get stored in a server database. The parameter, in the form of malicious JavaScript code, may normally be represented as a message in a forum or the description of a user's public profile. When a victim visits the page with the malicious code, the code can get executed in the victim's browser, allowing the attacker to steal sensitive information from the victim.

Reflected-XSS. An attacker lures or redirects a victim to visit a URL with a malicious parameter, which will then get forwarded to the corresponding web server. In this case, the parameter does not get stored, but it is immediately reflected (or *echo-ed*) back to the victim and gets executed in the victim's browser.

DOM-XSS. Similar to Reflected-XSS, an attacker first lures or redirects a victim to visit a malicious URL. However, the specified parameter will not get forwarded to the corresponding server, and the attack occurs entirely on the client-side. A prime example of DOM-XSS vulnerability is when a website reads its URL (using *document.location*) and writes the URL parameters onto its page (i.e., using *document.write*) without proper sanitization.

2.3 Existing Attacks on Service Worker

The security of service workers is an emerging research topic that recent researchers have discussed. Lee et al. [1] and Papadopoulos et al. [2] demonstrated how a malicious service worker can be utilized by attackers to run malicious background tasks (i.e., crypto-currency mining or botnet client). Watanabe et al. discussed how a malicious service worker can be injected into a benign re-hosted website [3]. Karami et al. [16] and Squarcina et al. [19] discussed how attackers can leverage the cache API, which is supported in the service worker, to leak user privacy or escalate the initial XSS attack. These prior studies tried to leverage or manipulate service workers for malicious purposes. Our work is orthogonal to them as we take the lessons learned to enhance the security of service workers and demonstrate how service workers can become a unified platform to provide security for websites.

2.4 Existing Defenses Against Web Client-Side Attacks

We categorize defense techniques into three categories based on how the proposed defenses can be deployed.

Browser-Centric solutions require browser modifications to implement a defense mechanism. This type of defense is the most robust as it runs in the lowest level, the browser code. Bypassing browser-centric defenses usually implies the attackers can tamper with the browser's binary, or the defense's design has a critical flaw. Once the proposed defense is acknowledged in the community, it may be put into the web standard such as in the case of CSP [20].

Nevertheless, the limitation of browser-centric defenses is that there is a significant time lag before a proposed solution becomes official, and until then, it is difficult for the prototype to be widely deployed. For instance, autofilling hidden fields in websites was first reported to Chromium as early as January 13th, 2015 [21] with a proof of concept attack shown two years later [22]. Since then, Chrome has constantly improved its autofill security such as disabling autofill insecure forms in Chrome 87 (October 2020) or showing explicit prompts when autofilling an address in Chrome 95 (October 2021). It could take years before a security feature is developed, tested, and deployed. Considering that the web is fast progressive, new attacks may already evolve into a different variant that is more resistant to the proposed solution. Even when a new feature has been supported, not all users will use the latest version of their browsers, which further delays the deployment of these features. Therefore, although browser-centric defenses are robust, they are too rigid for the current web development. Our proposed platform, SWAPP (discuss in Chapter 5), addresses these limitations by allowing developers to deploy new prototypes without being officially integrated into the web standard to keep up with new attacks.

User-Centric solutions usually take the form of browser extensions that users can manually install to provide protection. For instance, Schwarz et al. proposed JSZero [6] to help prevent micro-architectural side-channel attacks. A more popular example of a user-centric solution is AdBlocker, which can prevent unwanted advertisements. Such user-centric defenses are usually easier to deploy than browser-centric solutions, i.e., installing an AdBlock extension only takes a

few clicks. However, it is unclear whether the prototype, developed as a browser extension, will be widely deployed by users. For instance, even Adblocker, one of the most common browser extensions, is reportedly installed in less than 50% of clients [23]. There is at least another half of the population that is not used to (or decide not to) utilizing browser extensions. This may also apply to any other user-centric defenses in general.

Additionally, users are known to be the Achilles heels in security. As reported by Akhawe et al. [24], users may even ignore security warnings such as the SSL error. Therefore, web developers should only treat user-centric solutions as optional when considering the security of web users. Because a service worker is automatically installed by default during a user visit, we propose SWAPP as a service worker library as it is more controllable by web developers and reachable to the user clients. We find that more than 95% of running browsers support service workers [25].

Server-Centric solutions are deployed by web developers at the back-end server, as a proxy, or as parts of the websites. Depending on where a server-centric defense is deployed, there can be limitations. For solutions that run in the server like network firewall [5, 26], they lack the context of the client at run-time, hence they may not detect an attack that occurs exclusively on the client. Solutions that are deployed as a proxy also require additional infrastructures, which can incur additional cost and complexity [27, 28]. On the other hand, client-side defenses that run in the document context such as XSS filters [12, 13] share the execution context with attackers. This put them at risk of getting bypassed or manipulated at run-time [29]. Therefore, they are alternatives proposed in the form of defensive JS [30]. Nonetheless, defensive JS solutions may require major changes to the legacy code. SWAPP, on the other hand, does not require many changes to the legacy code compared to existing methods.

3. SW-XSS: SERVICE WORKER CROSS-SITE SCRIPTING ATTACK*

As a service worker provides such unique functionalities and execution environment, its security is critical. In this study, we systematically analyze the service worker lifecycle. Generally, web browsers enforce several rules to ensure a service worker will be safe from outside tampering. Despite existing safeguards, we find that many websites introduce a questionable practices during the installation process that can jeopardize the security of a service worker.

These websites introduce a questionable programming practices and break the security assumptions in favor of configurability and flexibility of their service workers. They usually install a service worker with URL search parameters as internal configurations, which are blindly trusted inside the service worker. When a malicious parameter is fed and reaches a sensitive function, it can allow an attacker to execute a cross-site script and compromise the service worker. We term this type of vulnerability as Service Worker Cross-Site Scripting (SW-XSS). Unlike other types of XSS, SW-XSS attackers do not necessarily leverage a web page's vulnerable parameters. Instead, they target the vulnerable parameters of a service worker and gain access to extra capabilities from the service worker that are not available to other XSS attackers.

In summary, we discuss the motivation of the attackers to launch SW-XSS attacks and the consequences. Then, we demonstrate how SW-XSS vulnerability can occur in appified websites. To assess the real-world security impact, we propose and develop a tool called SW-Scanner and use it to analyze top websites in the wild. Our findings reveal a worrisome trend. In total, SW-Scanner finds 40 websites vulnerable to this attack including several popular and high-ranking websites. We estimate the number of monthly visitors to the vulnerable websites to be up to 300M users. We report the vulnerabilities to all affected websites. Finally, we discuss potential defense solutions to mitigate the SW-XSS vulnerability.

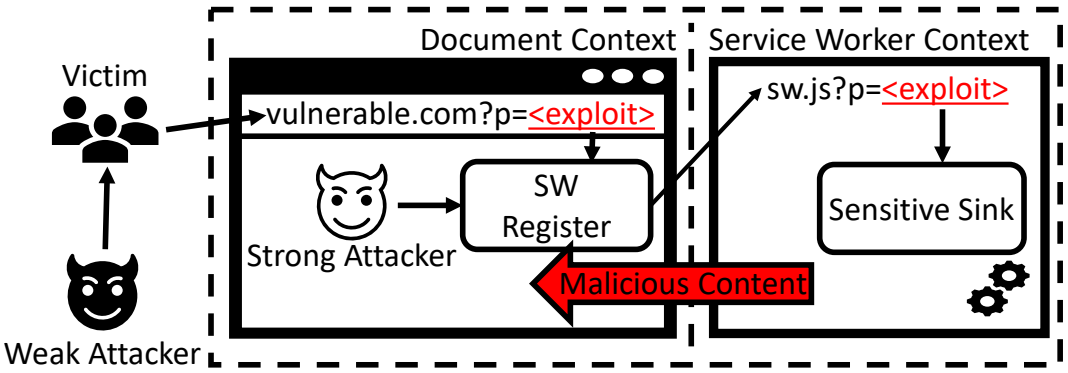

Figure 3.1: An illustration of SW-XSS attack threat model

3.1 Threat Model

Generally, web attacks consider two separate contexts of client and server. However, as shown in Figure 3.1, we extend the web attack model and divide the client-side into two contexts: document context and service worker context. Document context can be regarded as the usual scope of the client-side in a traditional web attacker's threat model, which covers the main page's execution context or the DOM. The service worker context, which was not accounted for in the previous literature, can be regarded in a similar manner to the server-side in a traditional web attack model, where an attacker cannot directly tamper with it but can still leverage a vulnerability in the service worker context to compromise it. A service worker provides several unique functionalities that are not available in other contexts, thereby making it a new target for attackers. In this attack, we consider two types of attackers which we term weak attackers and strong attackers.

3.1.1 Weak Attackers

represent a threat model consistent with the existing Web attackers presented in any typical XSS attack. This type of attacker can craft a URL that exploits certain vulnerable code in the target website. When a victim navigates to the URL or visits a malicious website that includes an iFrame pointing to the URL, the victim's service worker will be immediately compromised. The attackers can use the service worker's *fetch* event to inject malicious code into the document context and carry out malicious tasks that any typical XSS attacker can perform.

The most prominent aspect of the weak attackers model is that a service worker creates a new attack vector in the form of a new sensitive function called *navigator.serviceWorker.register*. This function plays an important role in the SW-XSS attack as it can potentially allow URL parameters to pass into the service worker, which can then be used to inject malicious code back into the document context. Therefore, the unsafe usage of this function can *at least* lead to similar consequences as other sensitive functions such as *document.write* or *innerHTML* utilized by a DOM-XSS attacker. To the best of our knowledge, we are the first to identify the service worker's *register* API as a sensitive function.

Not only can a service worker opens a new attack vector for launching an XSS attack, but it can also provide several unique functionalities that can be leveraged by an attacker. For a weak attacker, these features are a bonus that can be used to escalate the initial attack, but they are the main goal for a strong attacker. Therefore, we will explore these functionalities while discussing the motivation for a strong attacker.

3.1.2 Strong Attackers

are present in the form of JavaScript code executing in the document context. This type of attacker has access to document context's other unprotected scripts and APIs, thus they can already launch a wide range of attacks such as cookie stealing, phishing, etc. Their goal is to infect and take control of the presumably secure service worker to obtain additional capabilities from the service worker context. Such attackers can still greatly benefit from compromising a service worker, given that, as stated by the W3C service worker's security consideration, service workers create the opportunity for a bad actor to turn a bad day into a bad eternity.[1]

As a strong attacker already resides in the document context, the motivation is different from a weak attacker's. A strong attacker mainly wants to compromise a benign service worker to utilize its features to escalate or strengthen the initial attack. We discuss the features unique to the service worker context and how an attacker may utilize them as follows.

Network traffic interception. Unlike the document context, a service worker has access to the

[1] https://www.w3.org/TR/service-workers/#security-considerations

network traffic of the website. It can intercept network traffic of the files under its scope and modify any HTTP header and content. This type of interception can be used to inject malicious content, and it is not subjected to the monitoring or security enforcement of existing defenses. For example, when a malicious third-party script in the document context is prohibited from modifying other DOM elements (e.g., by other proposed defenses that limit the access of third-party origins [31, 32]), it can use the service worker to directly modify the web page's DOM content. Therefore, an attacker can potentially use a compromised service worker to circumvent certain types of defenses in the document context and execute the actual payload.

Persistent across sessions. Once successfully registered, the service worker's content (e.g., event listeners) will persist until a newer service worker replaces the old one. Similarly, a malicious payload stored in a service worker can last across sessions. An attacker can use this capability in conjunction with network traffic manipulation to fully take control of the target website for an extended period of time. This can especially benefit a temporary strong attacker (i.e., in the case of reflected XSS attacks) as she can turn the attack into a permanent one by hijacking the service worker.

Instant push notification. One feature of a service worker is that it allows a service provider (or an attacker) to remotely activate a push event and display a push message at any time regardless of whether the browser is open. This feature brings about two advantages for an SW-XSS attacker. First, the attacker is not required to wait for a victim to visit the website on her own accord to launch a phishing attack. The attacker can initiate the attack at any time through a push message. Second, the push message's sender is shown as coming from the website, which is normally a legitimate website. Therefore, the phishing message will appear more realistic compared to a message that comes from a different and unknown website.

Nevertheless, both types of attackers share an important requirement. The target service worker must use URL search parameters inside a sensitive function without proper sanitization during the registration process, allowing code execution inside the service worker. This basis defines what we consider a vulnerability throughout this section.

3.2 SW-XSS: Service Worker Cross-Site Scripting Attack

Despite the built-in security mechanisms of the service worker, it is possible for an attacker to compromise a benign service worker. Due to a bad practice followed by a number of SW-enabled websites, an attacker can leverage it to import an arbitrary script into the target service worker.

Bad practice in service worker registration. When registering a service worker, a website can specify two parameters: a service worker's path and scope. The path specified can forward URL search parameters into the installation. For instance, if a website registers 'sw.js?userid=bob' as the path, the service worker's URL will become `'https://example.com/sw.js?userid=bob'`. This search parameter is accessible through the *self.location* API from the service worker context (equivalent to the *window.location* in the document context). Such practice is becoming popular and frequently used by websites as a way to correctly initialize service workers based on visiting users. This is due to the limitation of service workers in which they cannot directly access the document context information, causing websites to utilize search parameters in the service worker registration process to forward necessary data.

Typically, HTTP GET is a commonly used method for websites to make a request to a server. It is not too surprising that a website would also utilize URL search parameters to communicate with its service worker. However, for web servers to blindly trust information sent through the parameters, they face associated risks that the parameters may be maliciously crafted as studied by Saxena et al. [33] and Mendoza et al [34]. Similarly, we observe that service workers encounter the same issue considering that the search parameters may originate from an untrusted or vulnerable source in the document context, which is not uncommon in practice [35, 14].

Cross-site script injection in service worker. Although using URL search parameters in a service worker does not necessarily lead to code execution in the service worker context, we find that many websites use the parameters in sensitive functions. In the following example, we demonstrate an SW-XSS attack using a real-world sports website with more than 50 million visits each month. We refer to the website in this example as vulnerable.com.

Listing 3.1 shows the vulnerable HTML page of vulnerable.com and its corresponding service

worker. We can observe that vulnerable.com hosts a vulnerable page called sw.html. At lines (1-5), sw.html adds the load event, which will be executed upon page load. This event handler reads and directly forwards the whole URL parameters into the *register* API. Then, at lines (7-12), the service worker will read the parameters from its URL, extract a specific parameter called *resourceHost*, and directly uses it in the *importScripts* API. Throughout the whole process, the parameters from the original HTML page can reach the *importScripts* API, which is a sensitive function, without any sanitization. This kind of practice is questionable. Unfortunately, we find that it exists on several websites including high-profile websites such as this sports website.

```
1  <sw.html>
2  window.addEventListener("load", function() {
3    navigator.serviceWorker.register("/sw.js"
4      +location.search);
5  });
6
7  <sw.js>
8  (function() {
9    self.param = parseParams(location.search);
10   var host = self.param.resourceHost;
11   self.importScripts(host+"/sw_fn.js");
12 }())
```

Listing 3.1: A simplified code from a vulnerable HTML page and service worker code allowing malicious code injection from web attackers

Based on this kind of practice, an attacker can leverage it to launch an SW-XSS attack. Figure 3.2 illustrates the attack on vulnerable.com. First, an attacker needs to make the victim's browser visit vulnerable.com with exploitable URL search parameters. For example, the attacker can craft a URL as `'https://vulnerable.com/sw.html?resourceHost=attacker.com'` and either tricks the victim into clicking the URL or includes an iFrame to the URL in an attacker-controlled website. By visiting this URL, the victim's browser will automatically register `'https://vulnerable.com/sw.js?resourceHost=attacker.com'` as vulnerable.com service worker. Consequently, the service worker will extract the parameter and import `'attacker.com/sw_fn.js'` into the service worker context. The attacker can host the

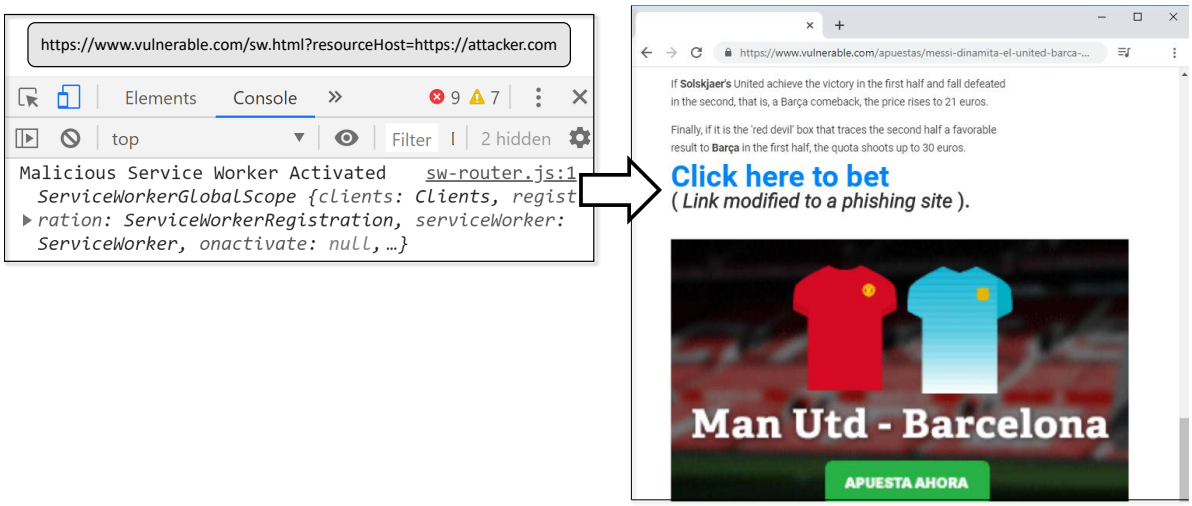

Figure 3.2: A screenshot of an SW-XSS attack targeting vulnerable.com.

sw_fn.js in her own domain to import event listeners into the service worker and take control of the website.

After the attacker successfully injects malicious code into the target service worker, she can register for any event handler inside the service worker context. The most important event that the attacker needs to focus on to fully take advantage of the service worker's capabilities is the *fetch* event. A *fetch* event is generated for every resource request. The *fetch* event handler has access to the request's HTTP headers, which it can freely modify. More importantly, the handler also has access to the corresponding responses and can easily modify or replace their HTTP headers or bodies.

By using the *fetch* event handler, the attacker can inject a malicious payload into the document context. The malicious payload is usually for stealing cookie, launching a phishing attack, or performing any task normally done in a typical XSS attack. As shown in Figure 3.2, the attacker can easily use the *fetch* event to modify a betting page of vulnerable.com to launch a phishing attack. When the victims click on the link, they will be redirected to another phishing page that can steal sensitive data, especially payment information.

It is worth noting that during the whole process, the victims may not even realize that they are

under attack. Because service worker registration does not require any permission from users and occurs silently in the background, when the attacker registers a malicious service worker inside a benign website, especially through an iFrame, the victims are given no visual cues. Additionally, even after the victims close the browser, the malicious service worker can stealthily infect the victims for as long as vulnerable.com does not update the service worker file or the victims manually remove the service worker.

SW-XSS in comparison with existing XSS. Although SW-XSS shares some similarities with existing XSS attacks such as DOM-XSS, there are certain differences that make the SW-XSS novel. We highlight the main differences between this attack and the existing XSS as follows.

1. **XSS entry point.** In traditional XSS, an attacker normally initiates the attack by crafting a malicious URL of a vulnerable web page, which may be in the form of HTML or PHP. We consider such a URL as an XSS entry point. While it is true that a weak attacker can also initiate the SW-XSS attack in a similar fashion, the actual entry point of SW-XSS comes from the URL of the registered service worker, which is *strictly* a JavaScript file. A weak attacker may be able to launch a normal XSS attack, but it does not necessarily lead to SW-XSS if the service worker and its URL are not vulnerable.

2. **XSS target.** While traditional XSS can compromise a web page or other web workers, to the best of our knowledge, we are the first to identify XSS in a service worker. Naturally, a service worker does not have direct access to the DOM, thus it is conflicting to regard this attack as DOM-XSS. Additionally, a service worker has unique features, such as network manipulation, that other types of web workers or web pages do not have. Therefore, we distinguish and regard this type of attack as SW-XSS.

3.3 SW-Scanner: Detecting SW-XSS Through Dynamic Taint Analysis

In this section, we introduce our SW-XSS detection tool, SW-Scanner. First, we discuss the goal of SW-Scanner in detecting SW-XSS in the wild. Then, we present the design of SW-Scanner and its implementation.

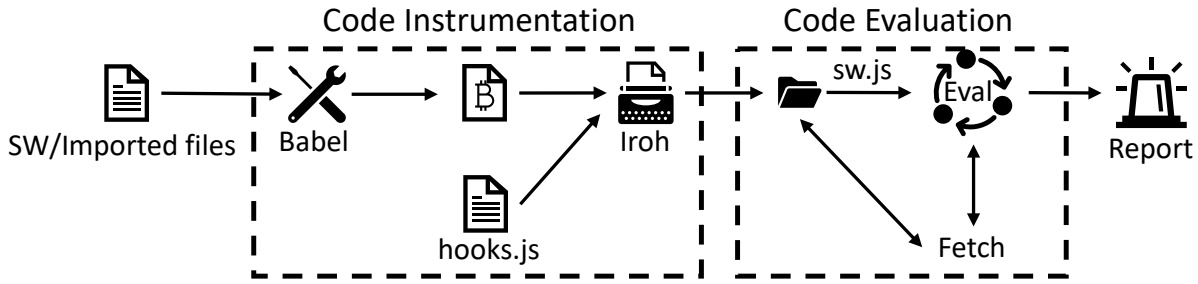

Figure 3.3: An illustration of SW-Scanner's pipeline.

Ultimately, the SW-XSS vulnerability stems from the unsafe usage of URL parameters in a sensitive function inside a service worker. Therefore, to search for SW-XSS vulnerability in real-world websites, we need to track how a service worker consumes a given URL search parameter. To accomplish this goal, we develop SW-Scanner as a taint tracking tool that can taint the URL search parameters of a service worker and report when a tainted value reaches a sensitive function. Specifically, the taint source is the *self.location* API and the taint sinks are the *importScripts, Function, eval, setTimeout*, and *setInterval* APIs. SW-Scanner mainly consists of two modules: the Code Instrumenter module can add taint tracking capability onto the target script; the Code Evaluation module acts as the controller and will execute the instrumented code and ensure that the taint tracking runs and reports correctly.

3.3.1 Code Instrumenter Module

This module accepts a JavaScript file as an input. Then it checks the input's validity using Babel [36], a JavaScript compiler. When the input JavaScript code is malformed, this module will use Babel to try fixing the code before rejecting it if Babel cannot do so. After the code is validated and normalized, the instrumenter will instrument the code to add the taint tracking capability using an existing dynamic analysis library called Iroh [37]. Iroh uses a JavaScript parser to read the target's code and transforms it into an intermediate representation, which can easily locate and instrument key locations such as the variable declaration, conditional check, or function's enter/exit. The full list of such locations is presented on Iroh's Github website [38]. Once one of these predefined locations is reached during execution, Iroh generates a corresponding event that can be handled.

This allows SW-Scanner to instrument JavaScript code into the predefined key locations.

For the purpose of tracking URL search parameters, SW-Scanner instruments taint information (by adding object's properties) into the taint source. The information includes a *tainted* label and a list of tainted words. For example, when a tainted string "example.com" is concatenated with a static string "/index.html", the resulting string "example.com/index.html" will have the *tainted* label and a list ["example.com"].

To correctly propagate the taint information, SW-Scanner adds hooks to the following events: the Function and API call events, the *New* operator event, and the Binary operation event. In the case of functions and API calls, when the calling object or the parameters contain a tainted value, the hook will taint the resulting object. Similarly, when a *New* operator is called, SW-Scanner checks the parameters and taints the resulting object if a parameter is tainted. For a binary operation event, SW-Scanner will check the left and right operands and taint the result if at least one of the operands is tainted. When a tainted value reaches a sensitive sink, SW-Scanner will log the tainted value.

3.3.2 Code Evaluation Module

This module is developed as a website. It accepts the instrumented files as input and reports the tainted result. The workflow of SW-Scanner follows these simple steps. First, SW-Scanner prepares its environment to mimic that of the target website. It overrides the *self.location* object and modifies all origin-related properties into the target's origin. SW-Scanner also registers its own service worker file using the same search parameters as the target service worker. Next, the target's instrumented service worker and imported files are saved in a folder, and SW-Scanner strips off all directory hierarchy from each file's path. By overriding the *importScripts* API, SW-Scanner can redirect all fetch requests to the local copies to avoid CORS-related errors. After the environment is set, SW-Scanner proceeds to *eval* the target's instrumented service worker file inside the service worker context. This will reenact the registration process and report the taint tracking result upon completion.

Adding taint information can affect the execution path of the service worker because primitive

Table 3.1: A table summary of the taint tracking analysis result.

Taint Source		Taint Sink	
Parameter Type	Count	importScripts	Function
Hash	367	4	0
URL	141	80 (35)	0
Code	1	0	1

data types in JavaScript (such as String or Number) can transform into an Object when the taint properties are added. When the service worker checks a variable's type and finds the type is mismatched, it can essentially alter the execution path. SW-Scanner ensures that this does not happen by executing the target service worker twice during the analysis. For the first execution, SW-Scanner does not add the taint information to the sources. Instead, SW-Scanner adds hooks to path-related events such as the If-Else and Switch-Case events. When the target service worker is *eval-ed* the first time, SW-Scanner records the path and the order that the target service worker has taken. Then during the second *eval-ed*, SW-Scanner adds the taint information and forces the path according to the first execution.

3.4 Evaluating SW-Scanner

In this section, we conduct an evaluation of the security impact of the SW-XSS vulnerability in real-world websites. First, we describe the data collection process and the overall statistics of service workers and their parameter usage on top websites. Next, we uncover the SW-XSS vulnerabilities in the wild, present the results of SW-Scanner, and discuss the responsible disclosure of the vulnerabilities discovered. Then, we evaluate the practicality of attackers utilizing the persistency of service workers by measuring the service worker's "freshness." Finally, we provide a case study of a vulnerable popular shopping website.

3.4.1 Data Collection and Overall Statistics

We first crawl the top 100,000 websites, based on Tranco's list created in December 2019 [15], using a custom Chromium build that we slightly modify to log the service worker registration and

importScripts API calls. We record the path, including the URL search parameters, used in these APIs. After this step, we are left with 7,060 websites with a service worker registered.

Next, we use Puppeteer's headless browser to revisit the websites in the list and download the JavaScript files. Then, we use Babel, a JavaScript compiler, to check the code's validity and possibly fix small syntax issues. If Babel is unable to parse the files, then we consider the files corrupted or protected from external download requests, and disregard these websites. After this step, we are left with 6,182 websites.

From the 6,182 websites, we measure the URL search parameter usage in the registration process. Specifically, we check the log files obtained from the data collection and analyze the service worker's paths. We use a regular expression to match the '?[key]=[value]&...' patterns in the path. Overall, We find that 2,525 of 6,182 websites (40.84%) specify at least one parameter in the registration API, and each website includes 1.29 URL search parameters on average.

3.4.2 SW-XSS Vulnerabilities in the Wild

For the 2,525 websites with parameter usage in a service worker, we use SW-Scanner to identify the SW-XSS vulnerability. For the taint source, we use heuristics to further categorize the parameter types and count the number of websites with a corresponding parameter type as shown in Table 3.1. We originally divided parameters into six types (Hash, URL, Version, Flag, Key, and Code), but only three types associated with at least one vulnerable website are reported here. Note that the numbers on the Taint Source column only represent the numbers of websites with a corresponding parameter type (not necessarily used in a sensitive sink). Instead, the Taint Sink column shows the number of websites that have at least one taint flow from the taint source reaching a corresponding sink.

We find that there are 367 websites with hashed parameters. Mostly, these parameters do not represent sensitive information. We manually analyze a set of sample websites that utilize these hashed variables and find that most of the samples used the variables as public API keys or a visitor's public information like username, which poses no immediate threat in our threat model. Nevertheless, we find four websites reported by SW-Scanner that hash a URL path used in the

importScripts API.

The URL type is the most dangerous as it is used mostly to interact with external sources, and it can be manipulated to point to an attacker's host. We find that 141 websites pass URLs as a parameter. Although the majority of websites use them in a non-sensitive sink, there are 80 websites originally reported by SW-Scanner that use it in the *importScripts* API. However, some of these reports contain parameters that cannot be leveraged by an attacker. For example, the parameter "?target=production" used in a website reaches the *importScripts* API, but the string is concatenated to a static domain, thus the attacker will not be able to import a cross-domain script into this website. SW-Scanner performs filtering based on whether the tainted value can affect the imported file's origin by checking the list of tainted words. Unless the list contains a domain, the report is removed. In total, SW-Scanner automatically removes 45 reports, leaving 35 websites. Lastly, there is one website directly passing JavaScript code into the URL search parameters, which we will further discuss in Section 3.4.5.

In total, SW-Scanner reports vulnerabilities in 40 websites. As our threat model assumes two types of attackers, further categorization of these vulnerable websites is required. For each of the 40 vulnerable websites, we manually inspect its source code to find all *window.location* and *register* API usages. When we locate a function that may allow URL search parameters to get executed as source code or reach the service worker registration API, we try launching an XSS attack on our client to verify the vulnerability. If the malicious URL search parameters can reach the registration API in these vulnerable websites, we label the attack's requirement as Weak, corresponding to the Weak Attacker Model. Otherwise, the attack's requirement is labeled Strong.

From the 40 vulnerable websites, 11 of them can be attacked by the Weak Attacker model, with the highest rank being in the top 20,000 websites. We use SimilarWeb [39] to measure the number of visitors to these websites and find that there are approximately 95M monthly visitors for the 11 websites in total. We do not claim that these visits represent vulnerable users, but any one of these visits can be a potential target for the attackers. Figure 3.4 summarizes the number of all 40 vulnerable websites and their monthly visits based on the category of websites. The Media category

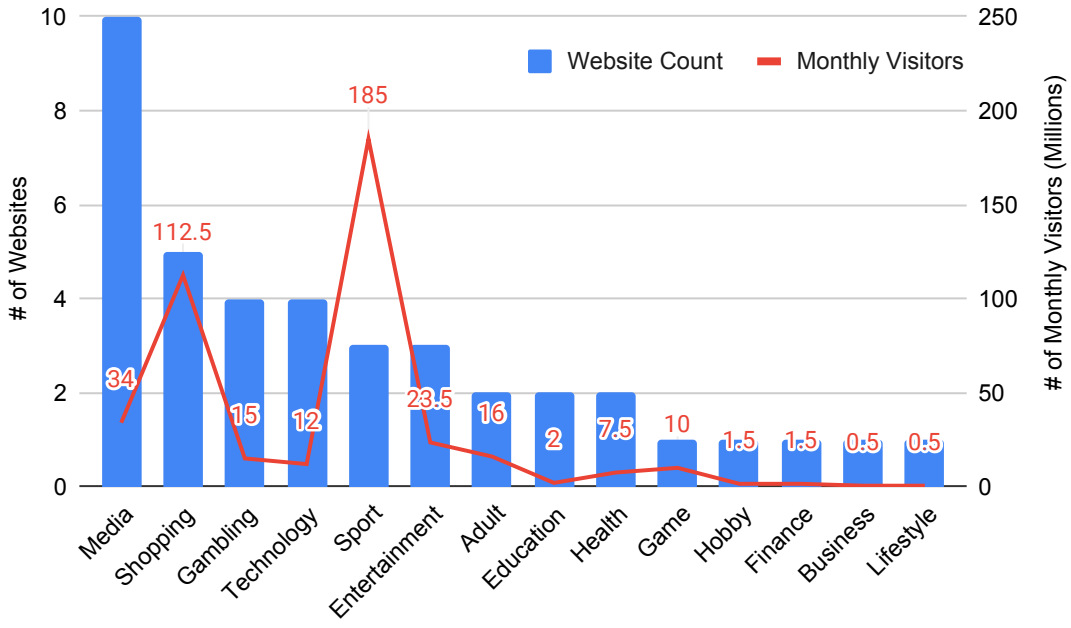

Figure 3.4: A chart representing the number of vulnerable websites by category and monthly visitors.

has the highest number of vulnerable websites, followed by the Shopping category. However, based on the numbers of monthly visits, the Shopping and Sports categories may actually be the most affected as there are 112.5 and 185 million monthly visits to the affected websites respectively. From this result, we can see that even though the number of vulnerable websites may appear to be low, the actual impact may affect a lot of users in practice.

3.4.3 SW-Scanner Performance

Here we discuss how we confirm the vulnerabilities reported by SW-Scanner and further address the impact of unexplored paths in the taint analysis on the number of vulnerabilities reported.

Confirming vulnerabilities. We manually inspect the 40 reported websites to confirm the vulnerabilities. For each website, we use Chrome's DevTools to inspect the target website and put a breakpoint at the reported sink. Then, we call the *register* API to re-install a service worker using a parameter that we specifically modify from the original value to point to another domain that

we control. In the other domain, we prepare a JavaScript file that would simply add event listeners. When the parameter reaches the breakpoint without its value being altered, which essentially allows the imported file to register the event listeners, we can confirm that the website is indeed vulnerable. From our analysis, we find that all of the 40 websites can be confirmed as vulnerable and we do not have any false-positive reports.

Unexplored paths. It is possible that some websites had a vulnerable path to a sensitive function that was left unexplored by SW-Scanner. To study the likelihood of such cases, we randomly select 100 websites that were not originally reported as vulnerable by SW-Scanner for further analysis. Then, for each of these websites, we use SW-Scanner to instrument instructions that can force the exploration of all branches of the website's service worker. SW-Scanner keeps re-executing the service worker and tries taking different paths until all paths have been exhausted. Finally, SW-Scanner reports websites that contain an invocation of a sensitive function, and we use Chrome's DevTools to manually inspect them. This entire process takes 10 minutes on an average per website. Due to the time and manual effort involved, it was not feasible to inspect all the 2485 websites that were not reported as vulnerable.

From the 100 websites, we find 81 websites with *importScripts*, 39 websites with *eval*, 66 websites with *setTimeout*, 11 websites with *setInterval*, and 37 websites with *Function*. The numbers are not mutually exclusive as one website may contain several sensitive functions. Our manual analysis aided by SW-Scanner for these specific functions helped us in uncovering some interesting trends in developer practices related to service workers.

For 79/81 websites with the *importScripts* API, we notice that the API is invoked within the first 40 instructions of the service worker with no branch happening before the API invocation. The other 2 websites include a packed website and an obfuscated website. Before importing any other file, the packed website performs an unpacking process and the obfuscated website performs a deobfuscation process. We reverse engineer the obfuscated website, which turns out to be using a static key that can be recovered, and find that it has a similar structure to the packed website. Specifically, both websites first unpack/deobfuscate the service worker, and then pro-

ceed to invoke the *importScripts* within the next 40 instructions similar to the other 79 websites. Based on such real-world observations from the 81 websites we manually inspect, we find that a service worker execution normally follows a basic sequence of operations structured as [unpack-/deobfuscate(optional)][short setup][import scripts][add event listeners and other functions]. The unpacking/deobfuscation process and the short setup normally do not depend on any input parameter, thus their execution will always follow the same path. Based on this observed basic structure of service worker's execution sequence in these real-world websites, typically there would not be an unexplored path for SW-Scanner that leads to an *importScripts* API. That is because such straightforward paths to the API are easily covered by SW-Scanner.

Additionally, we manually check each instance of *eval, setTimeout, setInterval*, and *Function* found in the 100 websites. All of the 39 websites with *eval* and the 37 websites with *Function* use the corresponding function simply to obtain the global service worker object (e.g., by calling (0, eval)('this')). Also, the setTimeout and setInterval are used safely among these websites (e.g., the parameter is a static function). We believe that because these APIs are well-known sensitive functions targeted by attackers (especially DOM-XSS attackers), web developers put more emphasis on the safety of these APIs. This is in line with our findings as we find almost no vulnerable websites with these APIs. In any case, when these APIs are used unsafely, SW-Scanner will be able to detect the vulnerability as shown in one case among the 40 vulnerable websites involving the *Function* API. Therefore, from our overall manual inspection of the 100 randomly selected websites, we find that the impact caused by unexplored paths is minimal.

While this basic structure of service worker's sequence of operations may hold today, it is possible that service workers will evolve in the future with new functionalities added. This could in turn make their usage more varied and thereby causing the structure to change. We plan to improve our tool to accommodate this change in the future by adding symbolic execution capability to SW-Scanner so that we can automatically traverse all paths and decide whether a path is vulnerable based on the possible values of the parameters. This will significantly reduce the need for any manual intervention while ensuring the likelihood of false negatives is low.

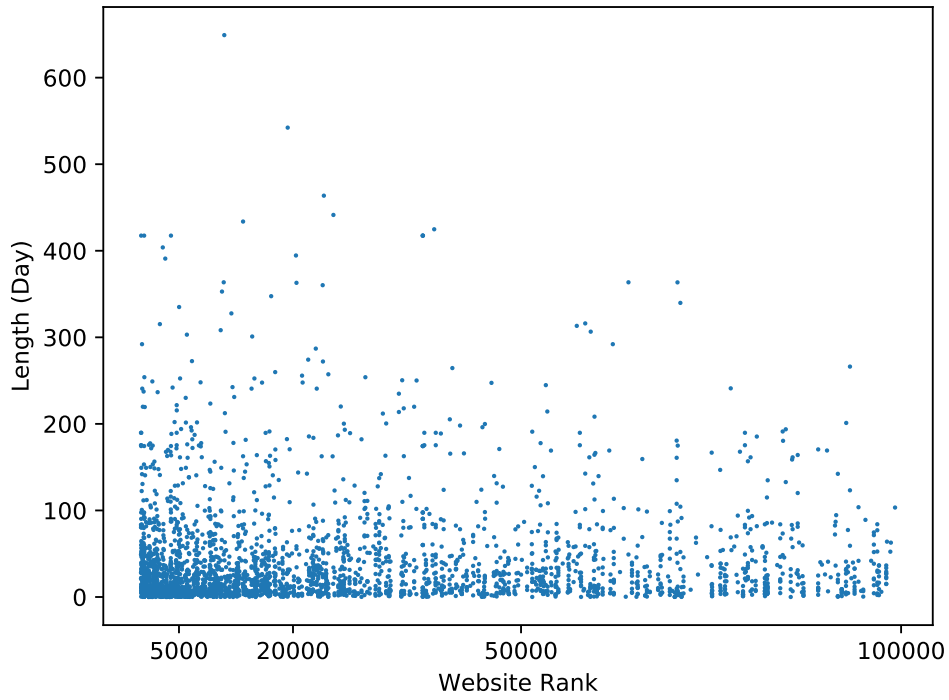

Figure 3.5: A scatter plot illustrating the length between updates of service worker files based on website ranking

3.4.4 Service Worker Freshness

Earlier, we claimed that a temporary strong attacker can benefit from the persistence of a service worker. However, it remains questionable whether a strong attacker can actually utilize the persistency in practice as the compromised service worker could get replaced. Therefore, we aim to measure how often each website updates its service worker to deduce the upper bound for how long an attacker can infect users.

We use the Internet Archive's Wayback Machine to retrieve the old service worker files [40]. Since some websites are not archived in the Wayback machine, we cannot obtain the complete data. In total, we can retrieve 3,166 data points from 777 websites with service workers that contain more than one archived service worker file as illustrated in Figure 3.5. For each website, we pick the oldest, newest, and eight randomly archived files, resulting in at most ten files per website. Finally, we sort the files based on the timestamps and compare each file if they are different. When the

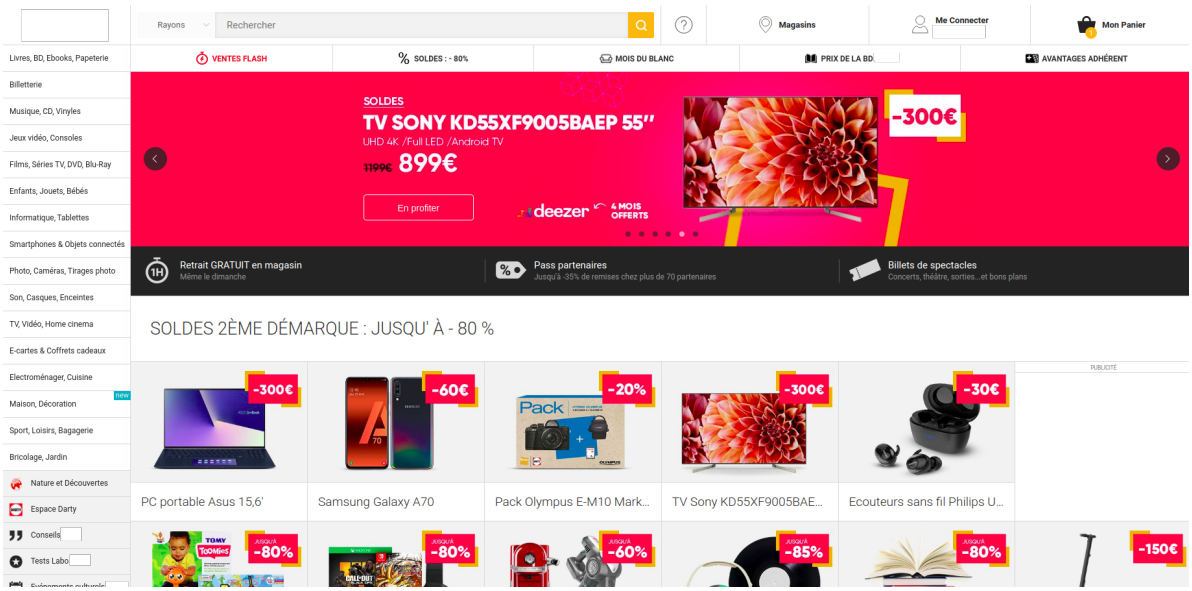

Figure 3.6: A screenshot of a vulnerable shopping website

adjacent timestamp files are different, we approximate the update time to be the mean value of the two timestamps and compute the length based on the update time. As a result, we find that websites update their service worker files on average every 40 days (while the median is 20 days), and the longest time a service worker file is not updated is 649 days. This shows a strong attacker can take advantage of the service worker's persistency for 40 days on average, supporting our claim that service workers are not as "fresh" in practice.

Additionally, we find a high-profile shopping website with 50M monthly visits did not update its service worker file from April 2018 to at least the end of 2019. During this period, we find that this website had an XSS vulnerability reported by OpenBugBounty [41] in which the bug was resolved after a few months. Because there is no change to the service worker file, any XSS attack from back then could theoretically last in the victim's machine for more than a year had the attackers also leveraged the SW-XSS attack. This illustrates the practicality of SW-XSS as it can be used in conjunction with other XSS attacks and further shows the importance of keeping service workers updated.

3.4.5 Case Study

We discuss a case study of another high-profile shopping website (Figure 3.6) with approximately 40M monthly visits that SW-Scanner reported. This website is the only vulnerable case involving direct code execution through the *Function* API rather than indirect code execution through the *importScripts* API like the majority of the vulnerable websites reported. Furthermore, this website has compressed and packed its service worker file making it difficult to analyze its source code both manually and automatically. Nevertheless, we demonstrate that SW-Scanner can effectively discover this case despite the complexity created by the unpacking process. The partial code of the website's service worker is shown in Listing 3.2.

```
1  ...
2  function i(t) {
3    var e = /^MATCH PATTERN$/.exec(t);
4    if (!e)
5      throw new TypeError('Err');
6    var n, r = o()(e, 4), i = r[1], u = r[2], a = r[3];
7    c = unescape(a);
8    ...
9    n = decodeURIComponent(escape(atob(c)));
10   ...
11   return new Function(n)
12 }
13 ...
14 var a = o.value; // o is service worker's URL
15 ...
16 f = new URL(a.uri,location);
17 ...
18 i(f.href)()
19 ...
```

Listing 3.2: A partial of service worker's code of a vulnerable website showing direct code execution from URL search parameters.

Starting at line 14, the service worker obtains its URL parameters and uses them to craft a URL object with its own origin at line 16. Afterward, the crafted URL, stored as *f*, is passed into the function *i()*. In the function, the URL pattern is tested at line 3, but the test does not affect the

attack in any way as it simply checks if the URL contains certain tags indicating that JavaScript code is specified in the parameters. From lines (6-9) the code is extracted from the parameters and returned at line 11, which later gets executed on line 18. This process happens before any event handler is registered. Therefore, an attacker can specify JavaScript code in the service worker's URL parameter to register her own event handlers and hijack the service worker.

From this case study, we illustrate that SW-XSS can be found even on high-profile websites and can occur in a complicated manner making it hard to be detected. Therefore, such problems may be overlooked by web developers. We hope that our work will help raise awareness regarding the importance of service worker security and provide useful insights for web developers to implement secure service workers in the future.

3.4.6 Responsible Disclosure

We directly contacted all affected developers of the vulnerable libraries and received replies from 7 websites, which have also fixed the problem. As not all websites have been fixed yet, all examples and results related to a vulnerable website's identity will be anonymized.

3.5 Countermeasures

As the main cause of SW-XSS comes from the unsafe/unsanitized usage of URL search parameters in service workers, the most natural solution is to properly check how the parameters are used inside the service workers. Nevertheless, we notice that the reason why websites follow the bad practice in the first place is that the service worker lacks a way to initially communicate with other contexts while being installed. Note that the *postMessage* API itself cannot be accessed until after the installation process is finished and the service worker is successfully activated. Therefore, viable options are to restrict the URL search parameters of a service worker, provide another way for the document context or web server to communicate with the service worker during the installation, limit script inclusion in the SW context, and prevent the registration API from being accessed after initial installation.

Restrict the URL search parameters of a service worker. We suggest a method involving

the manifest file, which is normally already included in SW-enabled websites. While the *worker-src* directive of the Content-Security-Policy (CSP) can limit the domains and paths that can be registered as a service worker, our attack utilizes the parameters of the same service worker file. According to the CSP3 specification [42, 43], the path does not include the parameters. Therefore, this CSP directive is currently not effective (unless there is a new specification that includes URL search parameters for source lists). In any case, we notice that the Manifest used to have the *serviceworker* property that can tell the browser which service worker the developers intend to install. Although this property has become obsolete [44], we believe that such a method could help mitigate the SW-XSS vulnerability as the intended URL search parameters can be specified as the service worker *src* property. One downside of this method is that the Manifest file is usually static, so the web server may need to provide multiple versions of the Manifest files if the URL search parameters need to be varied for each visitor. This leads to our second suggestion to use cookie, which can provide more dynamic values.

Provide an alternative for service worker to instantiate with data. Even though cookie is currently not accessible by a service worker, there is an active development of the Cookie Store API, which allows cookie access to a service worker. This can help web servers communicate with the service worker during the installation. For instance, the web server can specify the parameters along with a cookie. However, an attacker in the document context could still launch an SW-XSS attack by manipulating a service worker's cookie. Therefore, we suggest that the service worker's cookie should be isolated (or at least give an option/flag) from the document's cookie. For instance, an additional *SWOnly* flag can limit access from the document context but allows the Cookie Store API from the service worker to access it. One downside of this method is that it may require browsers to change their implementation to additionally check the calling context of the cookie API (whether it is from the service worker context). This could lead to additional overhead.

Limit script inclusion in SW context. Another defense option for the SW-XSS attack is to limit script inclusion through the importScripts API. To this end, web developers can utilize the CSP *script-src* directive in the service worker to specify which domain names can be imported

inside the SW context. This can effectively prevent SW-XSS attackers from importing malicious cross-domain files to hijack the service worker. However, there are two downsides to this solution. First, it cannot prevent SW-XSS attacks when the payload can be specified directly through the URL search parameters because the attackers do not need to use the importScripts API. This requires web developers to also implement a defense for URL search parameters (i.e., by using the Manifest as we suggested) to fully prevent SW-XSS attacks. Second, CSP is not widely deployed [45] and can be hard to configure correctly or can be bypassed [46]. Although specifying the *script-src* for service workers is seemingly simple and effective, we cannot guarantee that it is impossible for attackers to find a way to bypass this directive in the future.

Disallow SW registration after installation. Because most mitigation methods we discussed require changes to the browser implementation, it may take a long time for them to be officially supported or they may not even be picked up officially. Here, we propose another alternative that does not need browser modification. It is possible to use our proposed framework, SWAPP, to mitigate SW-XSS attacks. The intuition is that SW-XSS attacks occur during an installation. Therefore, we can simply disable the *register* API once the service worker is installed legitimately. SWAPP disables the registration API after SWAPP is installed in the service worker by instrumenting the *register* API. This makes further attempts to launch an attack against the service worker impossible. While this approach requires the users to have SWAPP installed prior to an attack, the assumption is very much practical because the service worker is automatically installed upon user visits. The user simply has to visit the website once before an attack happens. Note that the website can still update or replace the service worker by sending HTTP's Clear-Site-Data header, which will remove the instrumentation.

3.6 SW-Scanner Internet Distribution

We open-source our tool and the collected data, which can be found at `https://u.tamu.edu/sw-scanner`, to support more research in this direction.

3.7 Summary

In this chapter, we discussed a novel attack, Service Worker Cross-Site Scripting (SW-XSS), which is a new variant of XSS attacks. Furthermore, we described our dynamic taint analysis tool, SW-Scanner, which we developed to measure the prevalence of SW-XSS vulnerability in appified websites. The result showed that SW-XSS is emerging with 40 websites vulnerable to this attack. Lastly, we explored mitigation options against SW-XSS attacks.

4. SECURITY ANALYSIS OF SERVICE WORKER COMMUNICATION CHANNELS*

In our first study (Chapter 3), we learn that despite the intrinsic security mechanisms deployed in service workers, it is still possible to compromise or leverage a benign-but-vulnerable service worker. In this study, we extend our security analysis from the service worker lifecycle to the communication channels of service workers.

As discussed in Section 2.1.2, there are four communication channels. In this study, we focus on utilizing IndexedDB and Push to leverage a benign service worker, and we discuss the security enhancements of postMessage and Cache in the next chapter. There are two observations regarding the IndexedDB and Push channels. 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 are 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.

In summary, we first discuss the threat model. Then, we demonstrate the SW-XSS via IndexedDB and Push hijacking attacks using real-world examples. To assess the impact of these attacks, we extend a dynamic taint analysis tool and use it to measure the prevalence of both attacks. Our findings estimate up to 1.75M users can be affected by the Push hijacking attack. Finally, we discuss the mitigation methods.

4.1 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

* Reprinted with permission from "The Service Worker Hiding in Your Browser: The Next Web Attack Target?" by Phakpoom Chinprutthiwong, Raj Vardhan, Guangliang Yang, Yangyong Zhang, Guofei Gu, 2021. The 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '21), Copyright 2021 by Chinprutthiwong et al., publication rights licensed to ACM.

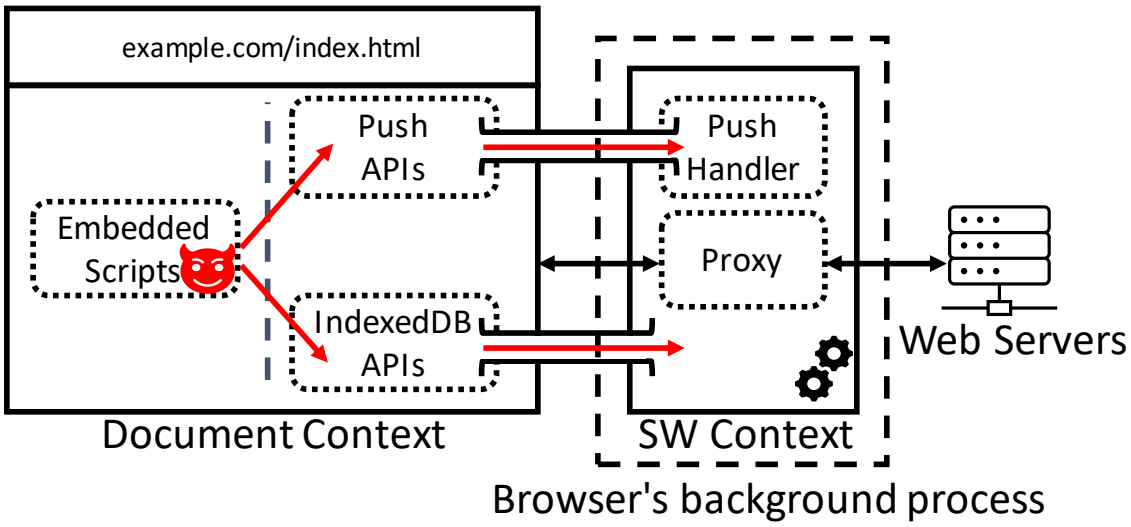

Figure 4.1: Appified web threat model & attack channels.

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 4.1. As discussed in Chapter 3, 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 4.5.1.

4.2 IndexedDB Attack

We illustrate the SW-XSS via IndexedDB attacks using a real-world website that we found, caused by the unsafe usage of IndexedDB inside the SW context. We show a simplified and anonymized code snippet of this website in Listing 4.1. The website initially stores a configuration variable inside the IndexedDB. Then its service worker will read the configuration and process it. As shown in 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 4.1: 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, it could take appified websites 40 days on average to update their service workers as discussed in Section 3.4.4. 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 4.2: An example of a robust input sanitization

Using importScripts with (non-static) parameters is 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 4.2 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 in line 3 before importing the result in 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 use alternative approaches similar to this example instead.

4.3 Push Hijacking Attack

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

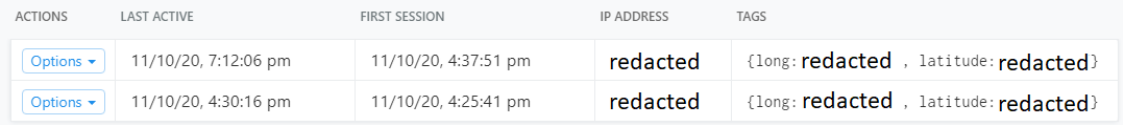

Figure 4.2: A screenshot from our OneSignal account page illustrating what user information can be made accessible once subscribed.

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 uses 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 embed 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 in-

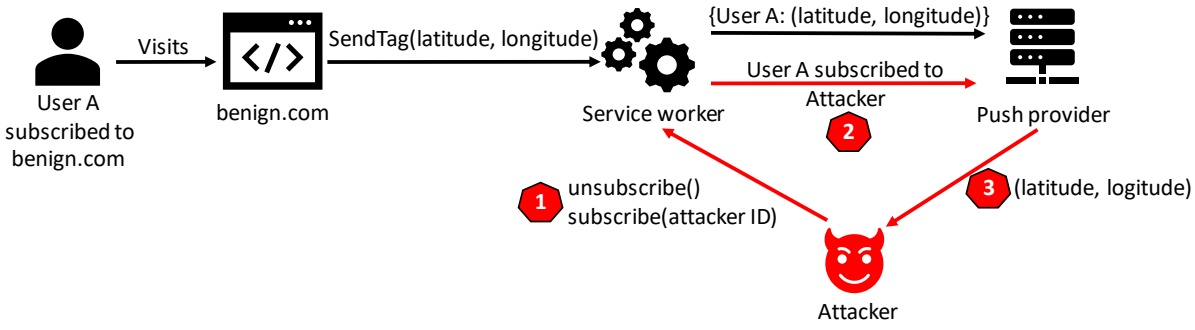

Figure 4.3: An illustration of how attackers can leverage push subscription to track user locations.

fer the victim's online behaviors, which can potentially reveal the victim's daily routine. Figure 4.2 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.

Here, we use a real-world example to 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 4.3.

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 utilize 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 4.3.

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*. 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

[1]Privacy-invasive malicious software or code that tracks and monitors victim's activities, which is becoming a worrisome problem [47, 48]

as suggested by OneSignal [49] 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 the OneSignal account page as shown in Figure 4.2.

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

Then on 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) tied 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's account. This is because a service worker will only get replaced/reinstalled 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 used a different library or even used 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 does not use a third-party push provider, the attackers may have to implement the backend 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 [50, 51] 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 per-

mission 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's locations.

Improving Stealthiness and Persistency. Generally, a website home page will contain the code to check and subscribe to push notifications. As a result, even if an attacker hijacks the push subscription, when the user revisits the website again, the malicious subscription will be replaced with a legitimate subscription. However, there is a way for the attackers to prolong the attack.

We observe that a website can have multiple service worker registrations and multiple push subscriptions. Specifically, each website's path can register a unique service worker and push subscription. For instance, the path /home/ can register "homesw.js" and the path /work/ can register "worksw.js". These service workers can co-exist and work separately. The homesw.js will control all paths under the /home/ folder. Similarly, the "worksw.js" will control /work/ sub-folders.

Based on this observation, the attacker can register a service worker and subscribe to push notifications in a different sub-folder than the home page. Listing 4.4 shows the example attack payload. Note that the path of a service worker can be different than the path that it is registered to. This makes the push hijacking attack more stealthy and persistent.

```
<script>
navigator.serviceWorker.register('onesignal.js?param=attackerID', {'scope': 'others/'});

navigator.serviceWorker.ready.then(function(reg) {
  reg.pushManager.subscribe({
    applicationServerKey: urlB64ToUint8Array(attacker_key)
  });
});
</script>
```

Listing 4.4: An example attack payload to register a service worker in a different path and subscribe to push notification.

4.4 Detecting Vulnerability Through Taint Tracking

Our extensions to further identify SW-XSS attacks via IndexedDB and Push hijacking attacks can be divided into two components. The first component corresponds to the data collection. Specifically, we extend the crawler discussed in Section 3.4.1. It now collects IndexedDB usage and push subscription metadata in addition to service worker registration information. The IndexedDB usage statistic can be used as a heuristic to identify whether the website could be vulnerable to SW-XSS attacks via IndexedDB. Further taint analysis will need to be performed on websites that show IndexedDB usage. The push subscription metadata includes a third-party library that is used to handle push subscriptions, subscription parameters, etc. This information is used to determine the prevalence of push hijacking vulnerability.

The second component corresponds to the dynamic taint analysis. Specifically, we extend the Chromium Taint Tracking project developed by Melicher et al [11]. It is extended to also consider IndexedDB data flow as both a taint source and sink. Specifically, we add two additional taint sources, IDB Get and Push message, and two additional taint sinks, IDB Put and importScripts, to the original tool. The intuition behind this change is that considering the document context, the IndexedDB is a taint sink because attackers can save malicious code inside the IndexedDB. On the other hand, considering the service worker context, the IndexedDB is a taint source because the service worker can read a malicious code from the IndexedDB and execute it in another taint sink such as *eval*.

With these extensions, we revisit the 7,060 appified websites that we identified in Section 3.4.1 to collect the Push metadata. Then, we perform taint analysis on the set of websites that show signs of IndexedDB usage.

4.5 Evaluation

In this section, we present our assessment of the security of service workers. First, corresponding to our threat model, we evaluate the prevalence of XSS vulnerabilities in appified websites (Section 4.5.1). Second, we assess the prevalence of the IndexedDB attack channel (Section

Table 4.1: 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

4.5.2). Last, we assess the prevalence of the push attack channel (Section 4.5.3). Note that the attacks 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).

4.5.1 XSS vulnerability in appified websites

Previous works have reported that regular websites do embed vulnerable JS libraries that are prone to XSS attacks [45, 14, 35]. 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 [41], 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.

We query OpenBugBounty for the bug reports of all 7,060 appified websites. The result is shown in Table 4.1. 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 systems (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 outweighs 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 4.3, XSS attackers can still leverage these types of websites for other purposes such as tracking 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 4.5.2 and the push channel in Section 4.5.3.

4.5.2 Prevalence of IDB attack channel

Our measurement study 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 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 con-

Table 4.2: A list of variable types of IndexedDB entries loaded inside a service worker.

Type	# Entries	Examples
Bool	63	true/false
Flag	485	persistNotification emailAuthRequired isPushEnabled
URL	88	https://www.eazydiner.com/ https://via.batch.com/2.1.0/worker.min.js Albertonews.com
Push Key	13	dwRH5VdycN4:APA91bEgqyRo0t9R1hW9oqwJAjLk6MUL9QNQ 7fLhMSrXxS0-MWdkZBV3tqIbfMl633itH8bakis3L6HTIOZJ51 o_tAST-ogHg1XJTBHnJvY_E3sNSz0OdJvNEgCfOg2gfya-Ely2p_Mi
ID	306	83AEAB70-31DF-2ADC-98F3-F0F365A753A1 f45438cb19044fd78277994b2231ddea 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 Chrome/77.0.3818.0 Safari/537.36 Sun Dec 22 2019 14:53:47 GMT-0600 (Central Standard Time)

tain 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 conducting 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 [11] that requires automated web crawling similarly faced the code coverage issue. While several techniques were proposed to improve web crawlers, efficient and exhaustive web exploration under time-bound constraints remains a challenge, especially for rich web applications that require login credentials [52]. 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 4.2. For instance, a string with only numbers and dots is considered numerical, a true/false string is Boolean, a string containing only alphabets 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 is more well-defined. We can use regular expressions 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.

4.5.3 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 labeling 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 4.3 (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 its own sub-domains, and 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 4.3 (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 4.4 (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 [41] 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 interacts more with the website, we have to manually inspect this sample

Table 4.3: 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

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.

To estimate the number of potential victims, we use SimilarWeb [39] to get the number of monthly visits to the 200 vulnerable websites. We find that there are over 1 billion visits in one month. According to a report from OneSignal [53], 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

Table 4.4: 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)

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.

4.6 Discussion

4.6.1 Key observation from IDB attack channel study.

Although we can only confirm 5 vulnerable websites in this study, 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 scripts as third-party service worker providers or SW providers 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 of 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 4.2 that a security check could be bypassed), the IndexedDB attack channel can become more prevalent in the future.

4.6.2 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) [54, 55], we speculate that the same trend will follow push notifications 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 of the API designs). For example, OneSignal prevents its subscribe API from being invoked twice. This should prevent

Table 4.5: A table of execution overhead occurred in the defense prototypes.

	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)

attackers in our threat model from re-subscribing using the attacker's 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.

4.6.3 Possible mitigation/defense

There are two directions that we try in order to provide mitigation against the IndexedDB and Push hijacking attacks. First, we implement ad-hoc changes into the Chromium browser to provide exclusive storage for service workers and to extend the current push APIs. Second, we discuss how SWAPP can be used to easily provide secure storage and push subscriptions.

IndexedDB. To prevent attackers from utilizing the IndexedDB against the service worker, the most effective method is to sanitize the IndexedDB entries before using them inside a sensitive function. However, it is extremely difficult to perfectly sanitize all inputs, which is why XSS attacks are still prevalent nowadays. Therefore, an improvement that can help enhance the security of service workers is to provide 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 additional dedicated storage.

As an ad-hoc solution, 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 IndexedDB API can be extended to provide 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 about 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 opens an IndexedDB and executes 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 4.5 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 dedicated storage for service workers is probable. However, further optimization may be needed and planned out by the browser developer community to better provide a robust solution on a larger scale.

On another note, we also 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 to 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

an 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 cookie access. However, such a design was soon proven wrong, and the Cookie Store API [56], which allows cookie access to the service worker, is under development to satisfy the needs of the developer community. Therefore, we expect that a mechanism like the SWOnly cookie may be supported in the future, though it may take some time before it is officially released.

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 [57], but it was rejected due to possible usability issues. Nevertheless, given that push providers start to incorporate location-based messages 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 an ad-hoc solution, we implement a prototype for this improvement in Chromium. The changes we make to Chromium are 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 4.5. Although the push's *subscribe* API is rarely called, there is a considerable overhead that could have a significant impact on a website. 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 subscriptions as discussed in Section 4.3.

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 4.3, 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.

SWAPP as a defense. Because our ad-hoc solutions require code changes to the browser, we propose an alternative to enhancing these two channels using SWAPP. For the IndexedDB, SWAPP uses the Trusted Code Block (TCB) module to instrument IndexedDB APIs to reserve certain storage names that cannot be opened in the document context. As a result, scripts in the service worker can use this reserved storage internally, while attackers in the document context do not have access to this storage. For the Push subscription, we implement Push Guard as a SWAPP app to check the push subscription ID. If the ID does not match, Push Guard will reject the subscription request. We discuss the details of SWAPP and its apps in Chapter 5.

4.7 Summary

In this chapter, we discussed the security analysis of service worker communication channels. We found two channels that can be leveraged by attackers: IndexedDB and Push subscription. Specifically, The IndexedDB can introduce SW-XSS attacks similar to what we discussed in Chapter 3. The Push subscription can lead to Push hijacking attacks where user locations and browsing behaviors can be leaked to attackers. We extended a taint analysis tool to conduct a large-scale study of these vulnerabilities in appified websites. The analysis found five additional websites exposed to the SW-XSS attacks via IndexedDB and approximately 1.75M users are vulnerable to the Push hijacking attacks. Lastly, we discussed several mitigation methods against these attacks.

5. SWAPP: A NEW PROGRAMMABLE PLAYGROUND FOR WEB APPLICATION SECURITY*

Ever since the introduction of the Internet, cybersecurity threats have always been relentless, especially regarding client-side web attacks. For example, one of the most prevalent attacks, Cross-site scripting (XSS), costs more than $4M a year in bug bounty rewards [58]. In response to new attacks, researchers have proposed many defense/detection mechanisms. While each of the methods has been proven reasonably effective in its rights, there are corresponding limitations based on where the mechanisms are mainly deployed as discussed in Section 2.4.

To mitigate the aforementioned limitations, a client-side framework for security functionalities is required. In this study, we explore an option to develop apps and deploy them in a new context, the service worker. To the best of our knowledge, we are the first to provide a platform to deploy apps inside service workers. Nonetheless, there are several challenges including patching the five service worker attack surfaces that we discussed in Chapter 3 and Chapter 4. We discuss the challenges in detail later in Section 5.2.

There are three goals that we want to realize with this platform.

- (G1) **Adoptability.** We want to provide browser-agnostic security functionalities that can be quickly adopted by web developers with minimal changes to the legacy code and without user involvement and continuous support from browser vendors.
- (G2) **Compatibility.** We want to provide a unified environment for different functionalities, including non-security libraries such as Workbox [59] (for cache management), to be compatible and run coherently in the same environment.
- (G3) **Fast prototyping.** We want to provide the extensibility and programmability with the platform for developers to implement security apps against existing and future attacks.

To achieve the first goal, we implement SWAPP (Service Worker APplication Platform) to be

deployed inside a service worker, which has been supported by all mainstream web browsers[25]. SWAPP is a new development framework for developing security prototypes and applications. It is a generalized platform that can be used for any website or in an enterprise setting accessible from specific networks such as business applications. Nonetheless, realizing SWAPP is non-trivial, especially to provide a secure environment for apps to run as parts of SWAPP. While the service worker is designed with security as a priority, it can still be compromised as discussed in Chapter 3 and Chapter 4. In consequence, we harden service worker APIs that can be leveraged as an attack vector and systematically evaluate the security of SWAPP against possible attacks (Section 5.4).

For the second goal, the heterogeneity of apps running inside SWAPP can be a problem. As we envision SWAPP to be a platform for future security prototype development, SWAPP needs to handle different apps (including legacy ones like Workbox) that try to handle the same resource coherently. However, the service worker is designed to work homogeneously. Because it runs asynchronously, each key resource is provided as an event that can only be handled by a single event listener. As a result, only one party has a monopoly on each type of resource, i.e., only Google's Workbox can handle the *fetch* event. To address this issue, SWAPP promotes a new sub-event queuing system by extending the original event handling mechanism. SWAPP will generate corresponding sub-events from the original event to allow different apps to sequentially handle a copy of the original event based on the app priority levels. The results will then be combined by SWAPP. This allows multiple security apps to run cohesively without any conflicts.

To address the third goal, we provide four interfaces based on the most crucial functionalities: network manipulation, document context access, secure communication, and secure storage. We develop several example security apps to show that SWAPP can be used to implement security apps against various types of attacks, including a recent side-channel attack [16] targeting websites with a service worker (Section 5.5).

Based on the limitations of existing defenses and our goals (G1-G3), there are three key reasons why we implement our platform in the service worker.

- **Adoptability.** Corresponding to the first goal (G1), SW-centric defenses are easy to deploy

and update because a service worker is automatically installed/updated by web browsers. Users do not have to make an extra effort to be protected compared to defenses deployed as a browser extension. Nonetheless, there are certain requirements that the clients and servers have to meet in order to utilize our platform. We evaluate the adoptability in Section 5.6.1.

- **Compatibility.** For our second goal (G2), the service worker runs in a different context than the main page, thus it minimally affects legacy code in the document context. Additionally, the service worker runs in an event-based manner in which a library may occupy an event handler. If the library utilizes a different set of events than what our platform requires (i.e., the *fetch* event), it will be compatible. In the case that the legacy code utilizes the same event handler, our proposed platform can encapsulate them as an app to run alongside other apps as we will discuss in Section 5.6.2.

- **Locality.** Regarding our third goal (G3), SW-centric defenses are deployed at an advantageous location, without requiring additional infrastructures. The service worker context provides rich capabilities especially allowing apps to act as a proxy for the website. With our provided interfaces for the proposed platform, developers can quickly implement, adopt, or update new prototypes. We evaluate the extensibility and programmability of our platform in Section 5.6.3.

In summary, we first discuss the threat model of SWAPP, when SWAPP can provide protection and when it does not. Then, we discuss the technical challenges to realizing SWAPP as a unified platform for service workers. Next, we provide the system design of SWAPP and evaluate the security of such design against different types of attacks. Then, we demonstrate how SWAPP can be used to implement several security apps. Lastly, we evaluate the performance of SWAPP and discuss its limitations.

5.1 Threat Model

We regard the service worker context as our root of trust. Therefore, all scripts included as parts of the service worker and SWAPP apps are benign. We assume the presence of XSS attackers

who may utilize the communication channels from the document context to compromise our root of trust as discussed in Chapter 3 and Chapter 4. This includes overriding native JS APIs to execute malicious code inside the protected scopes in the document context, i.e., prototype pollution attack. We also assume side-channel attackers who trick a user into visiting their websites, in which they insert iFrames pointing to the target websites to measure the page load timing and infer the user browsing history [16]. Further detail on this type of attacker will be discussed in Section 5.5.1.

Because service workers can only *partially* resist MITM attackers (e.g., browsers will never replace or update service workers when there is an SSL error despite the user clicking through the error to visit the web page), we do not assume attackers are able to obtain a legitimate certificate. This protection makes service workers resistant to the evil twin attackers with a self-signed certificate. Nonetheless, it is still vulnerable to capable MITM attackers who have a legitimate certificate (e.g., a compromised cloud edge serving first-party web content or a compromised first-party server). Such capable attackers will bypass any defenses with the same assumptions as SWAPP, which implements purely in JavaScript without browser modifications, browser extensions, or an additional proxy.

Furthermore, we assume SWAPP has been installed in the victim's browser prior to an attack. This is a reasonable assumption in all modern browsers because service workers are automatically installed on the first *normal* visit. If the victim visits the website exclusively in incognito mode (or other equivalences) before and during an attack, then SWAPP will not be activated because the incognito mode disables several features including the service worker. Additionally, changing the browser profile, device, or clearing the website data will remove the service worker, thus SWAPP will need to be reinstalled prior to an attack.

Note that we design SWAPP to be deployed by first-party developers. Most of our developed apps only need to access first-party scripts and exclude third-party content. This is because when an app needs to intercept cross-origin requests, there may be complications regarding the Cross-Origin-Resource-Sharing (CORS) protocol. We further discuss the limitation of SWAPP (and its apps) with CORS mode in Section 5.7.

Lastly, attackers in the forms of malicious browser extensions (or malware that can control web browsers) installed in the clients can remove any installed service worker, thus they are beyond the scope of our protection. This assumption holds true for any defenses implemented purely in JavaScript.

5.2 Challenges

In order to achieve the three goals (G1-G3), we have to carefully address three technical challenges (TC1-TC3) while designing our platform. This is because the service worker environment is not initially designed to support multiple apps utilizing the same event handler.

TC1: Homogeneous SW Environment. The service worker context is designed to mostly work homogeneously. Based on the W3C specification of the service worker, most crucial service worker events (i.e., *fetch*) can only be managed by a single handler, unlike the document context events such as *postMessage*, which allow multiple handlers. Because the *fetch* event can be crucial to a variety of apps, this design may prevent multiple apps from sharing the handler. For example, an XSS defense may want to perform ingress filtering to detect an XSS payload, while a CSRF defense may need egress filtering to check the HTTP's referer header. The *fetch* event allows network interception to perform both ingress and egress filtering. However, the problem arises when the XSS and CSRF defenses are developed independently by different groups of developers. This can cause conflicts, and only one defense may be allowed to run as the *fetch* event handler.

As a first step toward providing a unified platform for the service worker environment, the design of SWAPP must first accommodate and promote heterogeneity. To this end, we introduce a new event queue for the *fetch*, *activate*, and *message* event handlers. A Supervisor is assigned for each event to keep track of which app gets to execute and in which order.

TC2: Limitation of Original SW Events/APIs. While the initial service worker events provide unique capabilities that do not exist in the document context, they are still rather limited in the granularity to enable the development of security applications that are rich in diversity. For instance, the *fetch* event is dispatched during a network request, but the network response is treated as the byproduct of the request instead of having a dedicated event separately. To this end, SWAPP

utilizes the Supervisor to provide a custom event for apps to handle. The custom event system can improve the granularity of the original events especially by decoupling the request-response pairing from the *fetch* event into a request event and a response event.

TC3: SWAPP Security Against Attackers Because SWAPP is designed to act as a centralized controller to protect the website, it is unavoidable that SWAPP itself will be subjected to web attacks. For instance, Chapter 3 and Chapter 4 discussed attacks against service workers using the APIs that can propagate information from the document context to the service worker such as *serviceWorker.register* and IndexedDB. Son et al. [17] also discussed an attack using the *postMessage* API, which Steffens et al. [60] later highlighted the prevalence of this attack in a large scale. While the attack originally targets iFrames, it is also applicable to service workers. Because there are no built-in capabilities to reinforce security or accommodate web developers to utilize these APIs securely, we have to enhance the security of all channels and APIs that can reach the service worker.

5.3 SWAPP: System Design and Implementation

The components of SWAPP reside in both the service worker and document context. There are four key components that make up SWAPP: Supervisor, Custom Event Manager, Trusted Code Block, and Message Manager.

5.3.1 Supervisor

The Supervisor resides in the service worker context. It is deployed within an event listener. The main purpose of the Supervisor is to provide a heterogeneous execution environment inside the service worker (TC1). In our current implementation, we have put the Supervisor in three events: *activate*, *message*, and *fetch*. While we do not deploy the Supervisor in all events, these three events are sufficient to implement several apps as we later discuss in Section 5.5. This method can also be extended to support other events such as *push* or future events that are not yet released.

The Supervisor acts like a mediator between an originally dispatched event and apps. When it receives an original event (e.g., *fetch*) from the browser, it creates an event queue for SWAPP apps

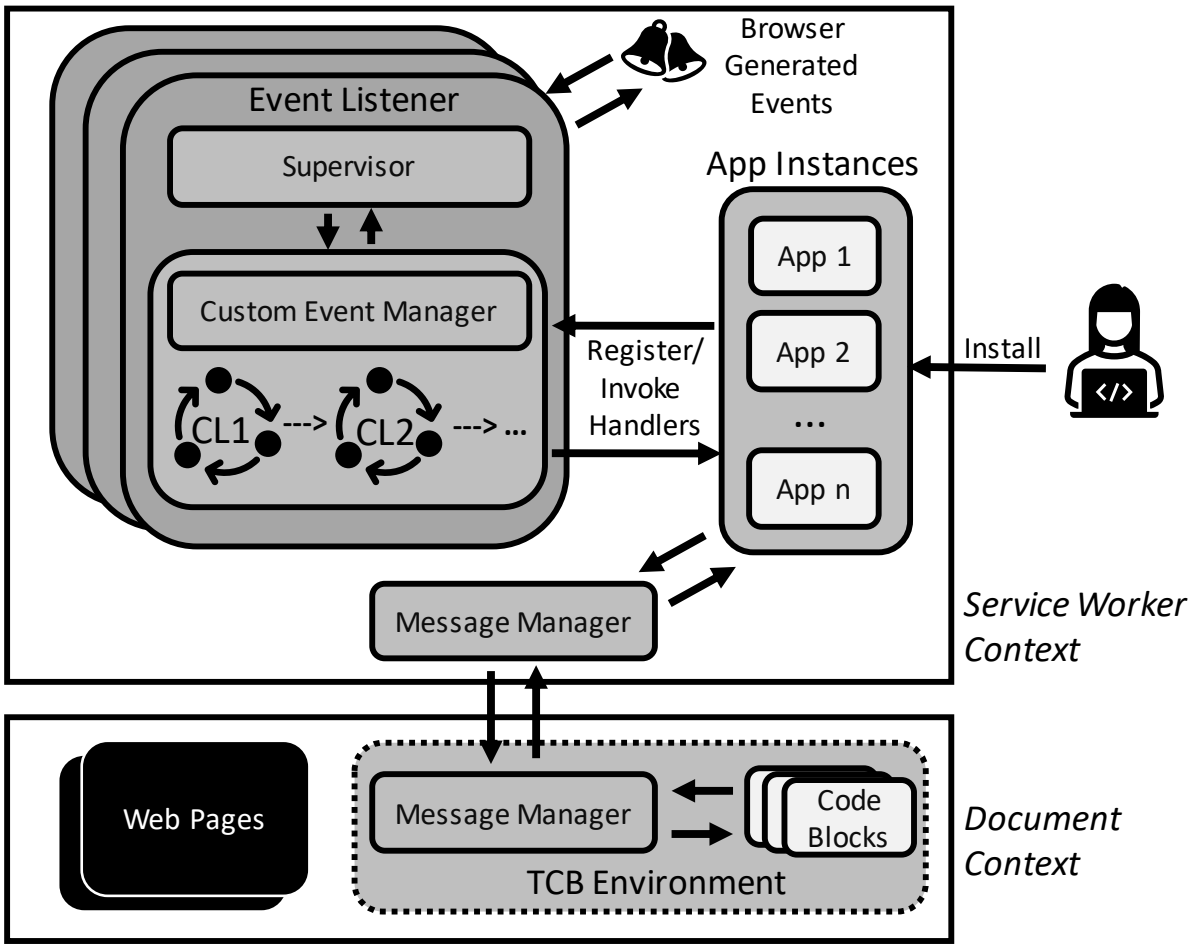

Figure 5.1: SWAPP Overview Architecture

to handle. We call events distributed from the queue to an app as subevents. Each app can register a subevent handler through the Supervisor, which will manage the execution order between apps and combine the execution results before making the final decision regarding the original event.

There are mainly two types of subevents for apps to handle.

- The *match* subevent tells the apps about the information of the original event. For example, if the original event is a *fetch* event, the information will include the HTTP headers and body of a request. Note that the available HTTP headers are still limited by the list of Forbidden header [61]. The handler of this event should tell SWAPP whether the app is interested in manipulating the event by returning a Boolean value.

- The *action* subevent is dispatched for apps interested in the original event after the Supervisor receives the answer from the *match* event. In general, the handler of the *action* event will have access to the final object, *fObject*, which is passed as a parameter that must be returned to be fed to the next handler as a parameter. The *fObject* contains a clean copy of the original event and a dirty version, which other apps may have modified.

For the Supervisor to manage the execution order of each app and to decide what to do with an event, an app can be assigned two parameters: execution order value (*eOrder*) and decision priority level (*pLevel*). A lower *eOrder* implies the app is ahead in the line and will execute earlier. A higher *pLevel* implies the app has a higher priority and can override the decision. If these parameters are not specified, the Supervisor will follow the app's installation order (i.e., the first app installed executes first and has the highest decision priority).

Furthermore, the possible values for a decision will be based on the original event. For instance, the decision of a *fetch* event can be *original* (proceed with the original), *dirty* (proceed with the modified version), *cache* (respond with specified cached content), or *drop*. The *activate* event cannot specify a decision as they are mainly provided for apps to initialize their variables when the service worker is activated. For the *message* event, we wrap it in an additional layer to provide enhanced security (further discuss in Section 5.3.4).

5.3.2 Custom Event Manager

The Custom Event Manager is closely tied to the Supervisor and can be considered as an extension of the Supervisor. Its main purposes are to define custom events, manage the transition between each custom event loop/queue (CL), and mediate between the Supervisor and apps. These are to provide more granularity to the original service worker events (TC2). Currently, we implement the Custom Event Manager only for the *fetch* event, but this concept can be extended to other events as needed.

Primarily, we use the Custom Event Manager to decouple the *fetch* event into the *request* and *response* custom events. Specifically, the Custom Event Manager divides the original *fetch* event into two stages. The first stage is similar to the original *fetch* event, which is triggered upon receiv-

ing a network request. However, unlike the original, this event ends when a decision regarding the request is made, not when a response is specified. The second stage starts immediately after the first stage if the decision of the *request* custom event is not to drop. Apps will then be notified to modify the response accordingly.

5.3.3 TCB Environment

The Trusted Code Block (TCB) Environment is injected into every web page by the Supervisor inside the *fetch* event listener. It is essentially located in the document context to provide a secure environment for apps that need to execute a piece of code in the document context. By design, the service worker cannot directly execute code in the document context. This leads to both advantages and disadvantages when considering implementing a defense. The advantage is that attackers, in the form of malicious scripts injected into the main web page, cannot directly access the service worker. However, the opposite is also applied that the service worker may not be able to enforce certain restrictions to the malicious script before the malicious operation is already in-flight to the *fetch* event. Several proposed defenses and techniques [6] rely on having trusted code running in the document context.

Similar to how the Supervisor operates, SWAPP allows app to listen to the TCB's *match* and *action* subevents. When SWAPP attempts to inject the TCB environment to a web page, the *match* event is dispatched. The handler will receive the web page information (e.g., the URL, HTTP headers, and the response body) and can decide whether the app wants to inject any code along with the TCB environment. The code will then be invoked when the TCB environment has finished initializing. We further elaborate on how the TCB provides a secure environment in Section 5.4.1.

5.3.4 Message Manager

The Message Manager runs both inside the *message* event listener in the service worker and inside the TCB. It provides a secure communication channel between the service worker and document context within SWAPP (more details of its security are discussed in Section 5.4.2). For an app to send a message, it can call our internal API, *broadcastMsg*, which will send a message to

a dedicated message port. This API accepts two parameters: the message content and the list of tags. The tags can be used to identify which apps are the intended recipients. An app can register a tag list along with a handler with the Message Manager. When the Message Manager receives a message, it will notify all apps that have a matching tag and invoke the registered handler.

5.4 SWAPP: Security Analysis

SWAPP itself can be the target of attackers (TC3). Here we elaborate on how we enhance SWAPP against threats from different attack vectors. In this section, we discuss the security enhancement of three attack surfaces: document context (S1), postMessage (S5), and IndexedDB (S2). For other surfaces S3/S4, we implement two apps corresponding to each surface (later discuss in Section 5.5.5 and Section 5.5.1 respectively).

5.4.1 Security of Document Context

Our threat model assumes that attackers can execute malicious code in the document context. This allows attackers to target the TCB directly. To this end, SWAPP puts the TCB inside a Closure and freezes sensitive JS objects used by the core of SWAPP similar to the method discussed by Schwarz et al. [6]. Nonetheless, apps may evoke JS APIs that are not originally frozen inside the TCB, and attackers can manipulate these objects [62]. For instance, if a SWAPP app registers a mouse click event listener that will call *console.log* to print out some information, attackers can override the *log* function to perform malicious tasks such as sending internal SWAPP messages to manipulate the service worker. Because the TCB is an anchor for SWAPP in the document context, we need to ensure that the TCB will not be compromised.

The security of TCB can be considered in two cases. The first case is the initial execution of the TCB. In this case, attackers cannot launch an attack nor modify any code. This is because the scripts in the document context will execute sequentially. SWAPP can intercept a web page and insert the initialization script at the topmost of the header (if available) or body to ensure the TCB is established before other code runs. Therefore, any attacks in this phase are not a threat.

The second case is after the initial execution of the TCB. In this case, SWAPP does not know

in advance what APIs the apps will use that need to be frozen. Furthermore, SWAPP cannot preemptively freeze all JS objects due to possible conflicts with existing libraries. Instead, SWAPP has to identify when there is a code tampering with an object executing inside the TCB. A way to check the integrity of a callable object is to inspect its definition through the *toString* function. A native object would return "[Native code]" upon inspecting, while a modified object would return the code that redefines it. However, there are many evasion techniques that can trick the *toString* function to return "[Native code]".

To avoid the cat and mouse game with the attackers in the document context entirely, our solution is to pass the target object to a fresh iFrame before calling the *toString* function. With this method, the malicious code that tries to trick the *toString* function (including its prototype chain) will not apply to the iFrame context. This is because the evasion techniques will tamper with a certain point in a prototype chain. By sending the object to a fresh iFrame, the object can be executed without the attacker's manipulation.

We provide a helper function, *checkIntegrity*, to help verify the integrity of the list of given objects. This function will create a fresh iFrame, go through the list of native API calls that the app wants to use, pass each API reference to the iFrame, obtain the object definition, and return the value to the original context. The returned value will then be hashed for future comparisons. Because the iFrame is newly created and destroyed after each use (and the API to manipulate iFrames will be frozen), attackers will not be able to affect the iFrame.

Now that we obtain the hashed API definition, we can check if it is malicious. The list of APIs to be checked is given by the app developers. They can inspect their own code and give SWAPP the list of hashed benign API definitions when installing the app. When an unmatched is found for an API definition, we know that an unexpected (and potentially malicious) code overrides the API and we alert the app to prevent the code from executing. In this way, we can protect the TCB while minimizing the effect on other legitimate libraries.

Because the service worker (un)registration APIs are also accessible in the document context, we must also prevent attackers from removing the service worker, which contains SWAPP's core

functionalities. To this end, SWAPP disables the register (and unregister) API entirely. This also prevents SW-XSS attacks (S1) as the attackers can no longer call the *register* API. If the website wants to legitimately execute the *register* API, then it can send the Clear-Site-Data HTTP header to remove SWAPP. Once SWAPP is removed, the APIs will not be overridden, and the website can invoke the register API again.

We test these enhancements by launching prototype pollution attacks with multiple bypassing techniques in our example website. We find that with a correct list of sensitive APIs defined, malicious code cannot be executed inside the TCB.

5.4.2 Security of postMessage

Existing works discussed the Postman attack [17], which utilizes the *postMessage* (PM) to attack a different context such as iFrames, and its prevalence [60]. The results show that websites often neglect or perform inadequate origin checks regarding the message sender, leading to code execution inside the targeted context. This type of attacker is especially potent in our threat model where the service worker is treated as the root of trust. Therefore, SWAPP's Message Manager must provide a mechanism to prevent this attack surface (S5) by default.

To mitigate against this type of attacker, we enhance and extend the original PM APIs. In the service worker, multiple PM (also referred to as *message* event) handlers can be registered. We register an instance of the handler and deploy a Message Manager inside the service worker (SW-PM). Correspondingly, the document context also has a Message Manager deployed (DOC-PM) in the TCB. The message operations within SWAPP are accessed through SWAPP's dedicated APIs as shown in Table 5.1.

Intuitively, we provide security in SWAPP's internal messaging system by limiting the sources of messages from the document context. When the DOC-PM is instantiated inside the TCB, it will also use the *MessageChannel* API to create communication ports. Then, it will send the port information to the SW-PM, who will keep the port information for records. Further communication will be made using this port. The SW-PM will reject messages from unauthorized message channels. As discussed in Section 5.4.1, the ports cannot be accessed by attackers outside the TCB. With

these enhancements, we mostly limit the sender origins of *postMessage* communications to only within SWAPP.

To test these enhancements, we try to launch the Postman attack (in a local environment) by sending post messages to the service worker. We find that without the apps leaking the dedicated port or other libraries running inside the service worker being vulnerable, attackers will not be able to successfully contact the service worker.

5.4.3 Security of IndexedDB

Chapter 4 shows that despite service workers executing in an isolated context, they can still be compromised through the IndexedDB (S2). Considering SWAPP apps run inside the service worker, which only has access to IndexedDB as a storage space, it is especially crucial to enhance the security of IndexedDB. This is because apps may need to store sensitive statistics, state information, or configurations locally.

In order to prevent attackers from utilizing the IndexedDB, SWAPP provides an isolated storage space dedicated to the SW context (SW_DB) and to SWAPP (SWAPP_DB). SWAPP overrides the IndexedDB APIs when initializing the TCB such that these two database names are restricted. SWAPP_DB is used internally as parts of SWAPP, and other scripts outside of the TCB will not be able to access it. The SW_DB is used specifically in the SW context, and even the TCB will not have access to it.

We try to launch an attack against SWAPP with these enhancements by attempting to access the private IndexedDB. We find that without apps (un)intentionally leaking the IndexedDB's transaction that opens the private database, attackers will not be able to access the secure storage.

5.5 Case Studies

Considering that both attacks and defenses are evolving, it could be almost impossible to implement a solution that can satisfy all types of defenses out-of-the-box. Therefore, SWAPP aims to provide a framework that abstracts generic security primitives and enables the extensibility for

Table 5.1: A List of Example SWAPP Interfaces.

Category	Interface	Description
Network Manipulation	reqMatch	Check if a request matches interception criteria
	reqAction	Perform the modification to a request
	reqPriority	Specify the priority level of the app to a request
	reqOrder	Specify the execution order of the app to a request
	respMatch	Check if a response matches interception criteria
	respAction	Perform the modification to a response
	respPriority	Specify the priority level of the app to a response
	respOrder	Specify the execution order of the app to a response
	setDecision	Set the decision for a request/response
	get/setMeta	Get/Set the metadata of a request/response
	get/setBody	Get/Set the body of a request/response
Document Context Access	tcbMatch	Check if a web page matches injection criteria
	tcbAction	Perform the modification to the document context
	tcbOrder	Specify the execution order of the app in the TCB
Secure Communication	msgLabel	Specify message labels of the app's interest
	msgHandler	Specify a message handler
	broadcastMsg	Send a message to all apps
Secure Storage Management	get	Get stored data from secure database
	set	Save data to secure database
	delete	Delete data from secure database
	createTable	Create a new table in secure database
	removeTable	Remove a table in secure database

developers. Currently, the core of SWAPP provides four types of primitives[1] (with corresponding interfaces shown in Table 5.1).

- *Network Manipulation* enables apps to inspect and modify any network requests and responses in-flight. Such capability can be leveraged by many types of defenses, i.e., XSS filter [63, 64], CSRF detection [65], or proxy-based mechanisms [27, 66].
- *Document Context Access* allows apps to execute code securely in the document context. This is crucial to defenses that require code instrumentation to enforce security policy at run time in the document context [6].
- *Secure Communication* provides a secure channel for the communication between the service worker context and the document context. As SWAPP spans in both contexts, an app may have parts of its logic located in different contexts or want to communicate with another app. This primitive helps alleviate these tasks for developers.
- *Secure Storage Management* ensures that data stored by apps will not be tampered with by unauthorized scripts from the document context. The only existing reliable storage provided to service workers is the *IndexedDB*, which is also shared with the document context (and can even be utilized for an XSS attack [67]). Thus, our new primitives can help restrict such unauthorized access and ensure attackers will not be able to manipulate app data.

5.5.1 Cache Guard

Karami et al. have recently discussed privacy-invasive attacks on websites utilizing service workers for caching (S4) [16]. The attackers lure victims into visiting their website, where the target web pages (or resources) will be loaded in iFrames, and the load times are measured to determine whether they are served through cache. If certain resources are served through cache, the attackers can infer that the victims have visited privacy-sensitive pages before. This includes inferring the victim's WhatsApp social graph. Such side-channel attacks (determining cached content to infer the victim's browsing history) are common threats for websites that want to utilize a service worker (or cache in particular) [1]. Note that this attack can work even when the X-

[1]Primitives and associated interfaces could be extended in future.

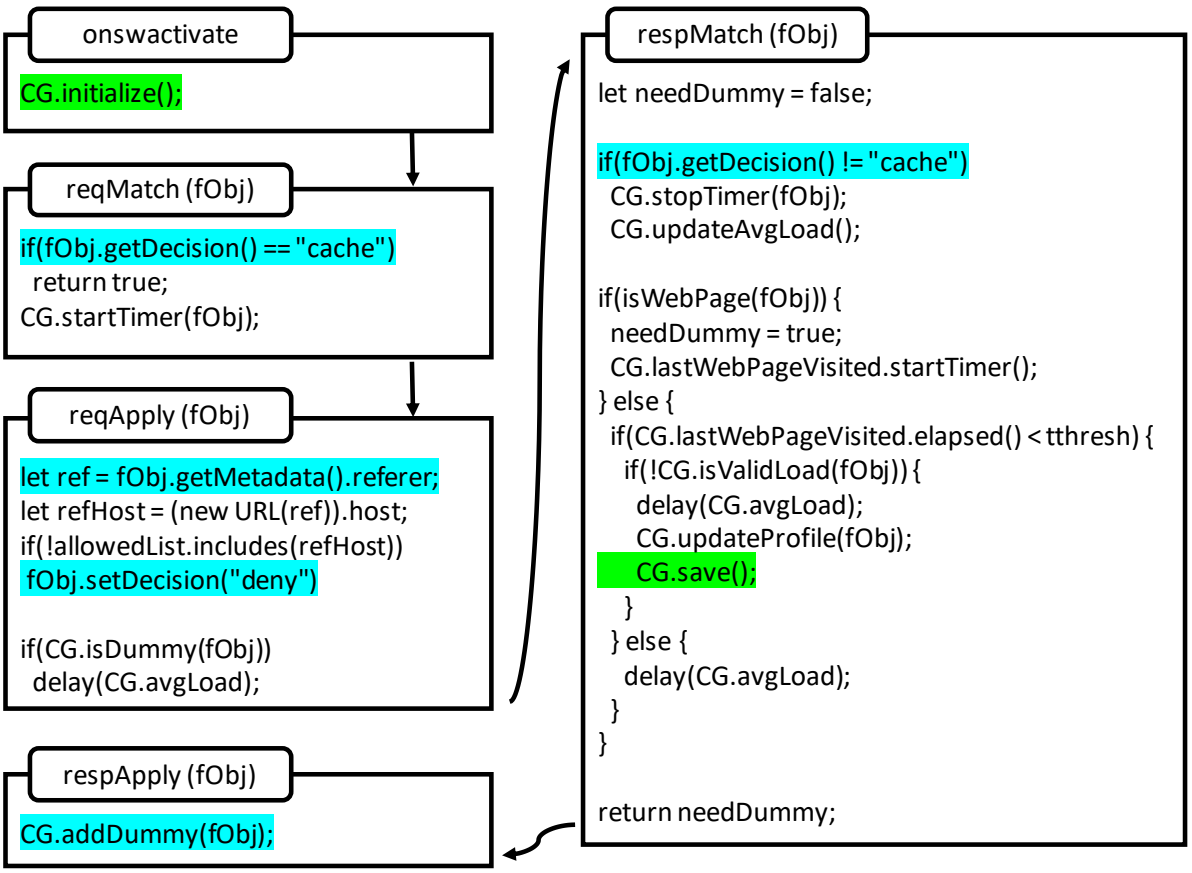

Figure 5.2: Workflow of Cache Guard.

Frame-Options or CSP is specified on certain browsers such as Firefox. This is because even when the browser does not show the iFrame content, the load time can still be measured.

To mitigate against this type of attack, Karami et al. implemented a helper tool for developers, in which the tool will instrument the fetch handler to check the referer HTTP header. If the header is not specified in the allowed list, the request will be dropped. However, this cannot prevent an attack where the website supports open redirection because the attackers can essentially forge the referer using the same-origin redirection page. To illustrate that SWAPP can be used by researchers as a platform to develop new defense mechanisms, we develop Cache Guard using SWAPP in response to this attack with two goals in mind. First, Cache Guard should be easily implemented and distributed by researchers and deployed by web developers. Second, Cache Guard should further prevent the attacks even when the website supports open redirection.

In addition to Karami's proposed defense (checking the referer header), which we implement using SWAPP in a few lines of code, we further improve Cache Guard to additionally protect when the website has open redirection. The final version is implemented in 258 LoC, and the overview is shown in Figure 5.2. The intuition behind this improvement is to delay the cached response to make it looks like it is loaded over the network when necessary. Because the cache is introduced to reduce load time and provide the offline capability, Cache Guard cannot simply delay all cached responses. As a result, we consider two scenarios for delaying cached responses.

First, if the resource being loaded is a web page, Cache Guard will attach a dummy resource request using the *respAction* event (the custom subevent for a response action). By using the *reqAction* event (the custom subevent for a request action), the dummy will be delayed to make the page load time similar to network loading. Cache Guard keeps the load time of prior resources to calculate the average network delay over time. The timer starts in the *reqMatch* (request match) event and stops in the *respMatch* (response match) event. In this way, users will not experience the delay and attackers cannot accurately determine the cache usage because the page load time is measured when the page has finished loading *all* resources in the page.

Second, if the resource being loaded is not a web page (and not our dummy resource), Cache Guard will delay it unless there is a prior web page request that Cache Guard knows will need the resource (i.e., legitimate resource requests). The intuition is that non-page requests mostly originate from a web page. Therefore, Cache Guard will cumulatively build a resource loading profile for each web page and check when there is a non-page request. If it does not match a prior profile, Cache Guard will delay it first before updating the profile. These are mostly done in the *respMatch* event.

Because attackers typically rely on measuring the first resource load time by adding random URL parameters, our approach will nullify this type of attack. We evaluate the effectiveness of Cache Guard by launching the side-channel attack discussed by Karami et al. We develop a demonstration website locally (accessible in our Github directory) and use Chrome's DevTools to measure the average resource load time across multiple runs. We use Chrome's *Fast3G* throt-

tling network profile to emulate the network delay. We find that with Cache Guard enabled, the first cached resource load time is within 10% of the average network load time. Furthermore, subsequent access to the resource is still as fast as the normal cache load time (within 50% of the average network load time). Therefore, Cache Guard is effective against this side-channel attack.

5.5.2 Autofill Guard

Nowadays, websites provide a login mechanism for users, usually as a form that users can type in manually or be autofilled by browsers or external tools. The login credentials are often a target for attackers, who could steal these sensitive login credentials from a login form that is automatically filled by browsers or external tools. For instance, Silver et al. [68] and Stock et al. [69] show that auto-filled forms are vulnerable to MITM and XSS attacks respectively. In this example, we propose an alternative defense, called Autofill Guard, that can protect login forms from XSS attackers. Autofill Guard can work complementary to and in conjunction with other existing defense mechanisms.

Autofill Guard mainly provides protection through isolation (by using iFrames). By putting a form inside an iFrame, which is isolated from the main context, XSS attackers will not be able to access the form anymore. Furthermore, to prevent attackers from creating an invisible form (different from the legitimate one) and tricking password managers to give them the credentials, Autofill Guard can also override JavaScript APIs and disallow form creation. The overview of Autofill Guard is illustrated in Figure 5.3.

When a user requests a website, Autofill Guard's Form Detector automatically detects a sensitive form and encapsulates it inside an iFrame. Then, if the form is submitted, Autofill Guard's Mediator will forward the request to the webserver to log in. Upon receiving the response, Autofill Guard's Notifier will notify the TCB to reload the main page. These processes are done automatically, thus there will be no differences from the user perspective.

We test the effectiveness of Autofill Guard by constructing a similar attack discussed by Stock et al. [69], where attackers inject malicious JavaScript code that tries to read the input of a login form. We perform the mock attack in a local environment with Chrome 80 to access the mock

website. We verify that Chrome automatically fills in the login form, but the malicious script we add cannot access the form information.

It is worth noting that there are two limitations in the current Autofill Guard version. First, as Autofill Guard is intended to protect against attacks targeting autofill in static login forms, it has to disable APIs that can create new forms, which can possibly introduce false positives (i.e., blocking legitimate form creation).

Second, the JS code that accompanies the forms would not be able to access the original page's DOM. In practice, web developers could modify Autofill Guard to include the necessary JS code along with the iFrame, i.e., to validate the correctness of the filled values or to handle an onclick event. However, the JS code would not have direct access to the main page's DOM due to the isolation provided by Autofill Guard. The developers could establish a postMessage channel between the iFrame and the main page, but this would defeat the purpose of Autofill Guard. This is because Autofill Guard is developed to isolate the forms from malicious scripts in the DOM and establishing such communication could expose the iFrame to the attackers. In any case, if the forms and the accompanied JS code do not rely on the main page's DOM, they would be compatible with Autofill Guard.

Autofill Guard is developed as a proof of concept to demonstrate how SWAPP can enable new directions and use cases. Our implementation of Autofill Guard utilizes iFrames and the service worker to manage extra requests/responses to allow the login to work despite the form being submitted in an iFrame instead of the main context. We hope this method can spark new ideas to implement more complicated defenses in the future.

5.5.3 Data Guard

HTML resources are used in websites for displaying various content to users. However, due to the complexity of designing and implementing access control policies, broken access control vulnerabilities exist on many websites. For resources that contain privacy (e.g. URL to a private file on the website or privileged operations [70][71]), if an access control vulnerability exists and attackers are able to steal the URL, the privacy could be leaked. To protect such data from being

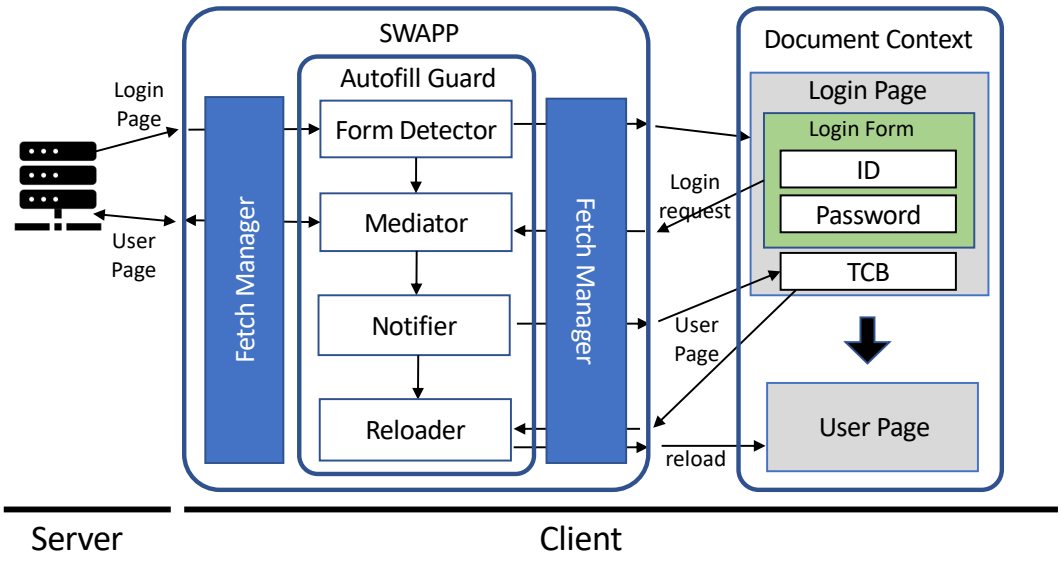

Figure 5.3: Workflow of Autofill Guard

compromised, we designed Data Guard to automatically preserve the data in a service worker and add them back to HTTP requests according to the context. The overview of Data Guard is illustrated in Figure 5.4.

```
...
function dg_init(){
    ...
    add_undoc_data_type("data_name", extraction_cb);
    ...
}

function extraction_cb(body, headers){
    // define the data extraction strategy here
}
...
```

Listing 5.1: Data Guard Undocumented Data Extension Template

Data Guard will first perform static analysis and find all predefined data types on the web page. Web developers can also define their customized data extraction strategies to support other types of data. To enable the customization in Data Guard, we provide a template for web developers to define their own data extraction strategies, as shown in Listing 5.1. For each element identified by

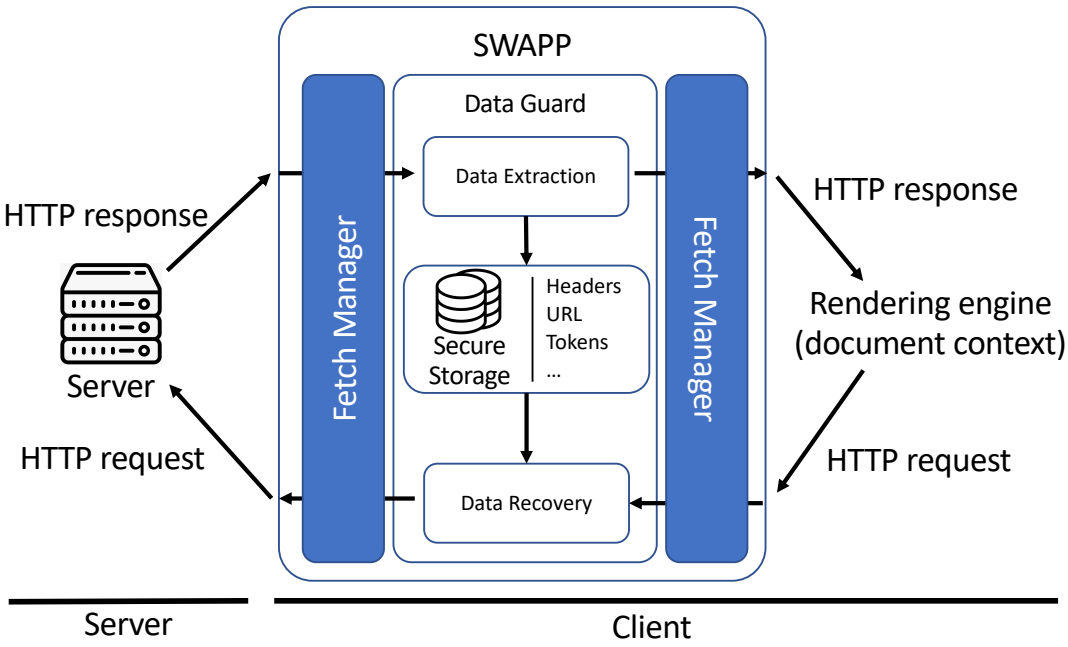

Figure 5.4: Workflow of Data Guard

Data Guard, we will replace the sensitive data with a unique string, which will be an SHA-256 hash string generated from the element. The original sensitive data and the corresponding unique string will be stored in secure storage provided by SWAPP as a key-value pair. Whenever the unique string is detected in any outgoing message, Data Guard will replace the string with the original sensitive data. Currently, Data Guard will replace all URLs on the web page as a proof of concept. Based on our observation, such practice will not harm the normal workflow of the websites we have tested.

With Data Guard, attackers will not be able to send valid requests to the server with the stolen data since it has been replaced with unique strings that can only be recognized by SWAPP in the victim's browser.

5.5.4 DOM Guard

Cross-site scripting (XSS) attacks have been one of the most prevalent web attacks. In recent years, an emerging type of XSS called DOM-XSS is becoming a severe problem. DOM-XSS is unique to existing XSS in that it occurs mostly at the client-side, making server-side solutions

ineffective. DOM Guard, however, can utilize existing techniques such as server-side firewall and apply it to the client-side through SWAPP.

Our implementation of DOM Guard allows a plug-and-play strategy where different types of techniques can be switched. Currently, as proof of concepts, we use a filtering technique as the detection strategy. To detect DOM-XSS payload from executing on the client-side, DOM Guard will check the URL segment of every request for potential payload using an HTTP encoder [72]. DOM Guard will compare the encoded URL segment with the original segment to determine a potential attack. Nonetheless, this can be improved by applying other existing detection techniques in DOM Guard. For example, Chaudhary et al. proposed a proxy-based technique to validate network responses [28]. In this case, DOM Guard can act as the proxy to lessen the requirement of the technique that needs a physical proxy to be deployed.

We demonstrate the effectiveness of DOM Guard using an existing XSS payload [73]. We create a website with a vulnerable page that will read the value of its URL segment and directly write it into the DOM. Then, we install SWAPP with DOM Guard activated. We test to see if the payload is executed by checking if the alert function is called. After we have visited the vulnerable page with the given payload list, we find that the filtering is effective.

As DOM Guard is currently designed to apply existing XSS detection techniques, it will inherit their limitations. Additionally, techniques that require heavy computation can potentially affect the clients. Nevertheless, the advantage of DOM Guard is that once a new technique is developed, DOM Guard can potentially make use of it. DOM Guard as a DOM-XSS detection app that can easily switch to different techniques/strategies demonstrates the flexibility of SWAPP.

In any case, it is also possible to implement XSS filtering in the TCB. However, implementing a filter in the document context may require additional native objects related to the filtering such as the input sources (e.g., Document.location) or sinks (e.g., appendChild) to be instrumented. There are two disadvantages to this approach. First, it could conflict with legacy code that utilizes these objects due to code instrumentation. Second, the instrumented code directly affects the page's responsiveness as it introduces delays to users when interacting with the web page. Implementing

the filter inside the service worker does not have the same disadvantages.

First, the service worker context is separated from the document context, thus does not conflict with the legacy code in the document context. Furthermore, URL filtering is native to the fetch event in which apps can easily check the outgoing request whether it contains a suspicious parameter. Instead of instrumenting several sources and sinks in the document context, our DOM Guard app requires only a few lines of code to achieve the same result.

Second, the overhead incurred in the service worker context affects user experience less because the service worker runs asynchronously in a different thread. The overhead is added to an already lengthy network delay and is non-blocking (i.e., does not block user interaction with the web page). For instance, a Fast 3G configuration used in Chrome's DevTools would normally add 300-500 ms to a network request. The filtering takes 10-20 ms (based on our measurement), so the overhead is likely not noticeable to the user. However, if a page needs an additional 10-20 ms for DOM Guard before it can be interactable, then this could affect user experience especially when more apps are deployed.

Nevertheless, we do not intend DOM Guard to completely replace or supersede existing XSS defenses. We simply demonstrate an alternative option for implementing an XSS defense in a new platform. We hope that our example will provide evidence for security researchers and practitioners to utilize SWAPP for more security apps in the future.

5.5.5 Push Guard

To prevent push hijacking attacks (S3) discussed in Section 4.3, we implement a simple app that checks push subscription ID against a list of allowed IDs that can be predefined by developers. When a push subscription is initiated, if the ID specified does not match, Push Guard will reject the request. This is possible because the Push ID of the provider is static and tied to one owner. Typically, the developers can obtain the ID from a third-party push provider and inform Push Guard. When the developers want to switch push providers, they can also easily configure the allowed list of IDs again.

5.6 Evaluation

In this section, we aim to demonstrate an alternative method to enhance website security. As a result, our evaluation will focus on four aspects: adoptability (Section 5.6.1), compatibility (Section 5.6.2), extensibility/programmability (Section 5.6.3), and efficiency (Section 5.6.4).

5.6.1 Adoptability

Our first goal is to provide a security framework that is easily deployable by web developers (G1). We evaluate the adoptability of our framework using two studies. First, we focus on browser clients and survey popular browsers to check if there are any vendors/versions that cannot adopt our framework. Second, we focus on web servers and measure the number of websites that meet the requirements to adopt our framework.

Client. To measure the adoptability of SWAPP within client devices, we first list out APIs that are utilized by SWAPP such as *serviceWorker, Fetch,* and *IndexedDB*. Then we refer to the statistics provided by an open-source project, CanIUse [74]. The project gathers front-end web APIs and provides usage statistics, which are regularly updated and maintained by the web developer community. According to the statistics, 95% of web users are using a browser that supports all APIs used by SWAPP.

Server-side. As our framework requires a service worker, we first need to measure how many websites are ready to install it. To this end, we conduct a measurement study on the top 10,000 websites based on the Tranco [15] list obtained in April 2021[2]. We develop a custom web crawler based on Node's Puppeteer and Chrome Devtools Protocol (CDP). Our crawler will visit each website's home page (with a 60s timeout), and the CDP will collect all network requests/responses and service worker updates. If the crawler fails to visit a website, it will retry three times before logging the error message.

Table 5.2 shows the configurations we use for our crawler. Because several websites apply different techniques to detect web crawler and trap it in an infinite loop, we have to utilize coun-

[2]https://tranco-list.eu/list/2QV9

Table 5.2: The settings and environment information for crawling SW-enabled websites.

Chromium	Version 92.0.4515.159
Puppeteer Puppeteer-extra-plugin-stealth Arguments	Version 10.2.0 Version 2.7.8 [headless: false, 'no-sandbox', 'disable-setuid-sandbox', 'disable-infobars', 'window-position=0,0', 'ignore-certifcate-errors', 'ignore-certifcate-errors-spki-list', 'start-maximized']
Chrome Devtools Protocol Events listened	Version 0.0.901419 ['Network.requestWillBeSent', 'Network.requestWillBeSentExtraInfo', 'Network.responseReceived', 'Network.responseReceivedExtraInfo', 'ServiceWorker.workerVersionUpdated']
Operating System	Ubuntu 18.04 (64-bit)

termeasures such as using the Puppeteer stealth plugin and trying different crawling arguments. While there are still cases that our crawler fails to visit, the numbers we report should sufficiently represent the adoption trend of service workers among top websites. Further optimization can potentially lower the failed cases, but our aim is simply to understand the adoption trend without being more disruptive than necessary to the crawled websites. Our crawler is not invasive and simply visits the home page of a website once without scraping the website data.

In total, our crawler successfully visits 9,293 websites. According to the error messages, 491 websites are not reachable, 183 websites are timed out, 11 websites have certificate errors, and 22 websites have other errors. We manually check 50 domain registration information of the 491 websites and find that they are mostly domain names that are not supposed to serve a web page such as googleusercontent.com. Furthermore, we manually visit 50 of the 183 timed-out websites and find that many websites are loaded within 1 minute outside our crawler despite using the same browser version (and setting), indicating bot prevention mechanisms may have been deployed to trap the web crawlers from finishing loading. In any case, we do not try to further collect information from these websites and refer to the 9,293 websites as our baseline.

Among 9,293 websites, 8,361 websites (90%) fully use HTTPS for all their requests, which

is a strict requirement to use a service worker. On another note, 694 websites (7.5%) are already using a service worker, which may require slight modification to work with SWAPP.

5.6.2 Compatibility: Working with an Existing Service Worker

While the number of SW-enabled websites is still small (7.5%), we believe service workers will be increasingly adopted by top websites in the future. Because SWAPP may conflict with existing service worker libraries, we need to illustrate that SWAPP can be deployed with minimal changes.

To this end, we discuss how Workbox, a library that provides the cache-ability to SW-enabled websites, can be encapsulated and run as a SWAPP app. We choose Workbox as an example for two reasons. First, Workbox is one of the most popular libraries embedded inside a service worker. Among the 694 SW-enabled websites, 174 websites (25%) use Workbox. We obtain this number through static analysis of the service worker file and identify Workbox's API calls using regular expressions. Second, Workbox mainly utilizes the *fetch* event for cache, which has a direct relation with our example app, Cache Guard, discussed in Section 5.5.1.

In order for SWAPP to work with Workbox, we have to encapsulate Workbox as a SWAPP app, which requires less than 30 lines of code modification to the original Workbox file. We utilize the Workbox CLI and follow the instruction provided by Google [75] to generate a service worker file (workbox-sw.js) with the default setting. Next, we create an app wrapper (WorkboxApp.js) for the generated service worker shown in Listing 5.2. Finally, we modify the generated service worker file and import it inside the app. This process preserves the caching policy provided by Workbox, and the SWAPP's Workbox app we create also works in conjunction with Cache Guard as expected.

```
1 [WorkboxApp.js]
2 + var wbApp = new Object();
3 + wbApp.reqMatch = function(fObj) {
4 +         return true;
5 + };
6
7 + self.importScripts("workbox-sw.js");
```

```
8  + f2fInst.addApp(wbApp);
9
10 [workbox-sw.js]
11 addFetchListener() {
12 -  self.addEventListener("fetch", (e => {
13 -    const {
14 -        request: t
15 -    } = e, s = this.handleRequest({
16 -        request: t,
17 -        event: e
18 -    });
19 -    s && e.respondWith(s)
20 -  }))
21
22 +  let ref = this;
23 +  wbApp.reqApply = async function(fObj) {
24 +    let e = fObj.getMetadata();
25 +        const {
26 +      request: t
27 +    } = e, s = await ref.wbHandleRequest({
28 +      request: e,
29 +      event: e
30 +    });
31 +    let b = await s.text();
32 +    fObj.setMeta(s);
33 +    fObj.setBody(b);
34 +    fObj.setDecision("cache");
35 +    return fObj;
36   }
37 }
```

Listing 5.2: Migrating Workbox to SWAPP

5.6.3 Extensibility and Programmability

In Section 5.5, we demonstrate the programmability of SWAPP in accordance to the goal G3. Here, we further show how easy to develop various security apps on our platform, compared with existing defense solutions. As shown in Table 5.3, SWAPP provides a unified platform that can be used to develop various defense solutions against different types of web attacks such as side-

Table 5.3: Extensibility and Easiness of Programmability of SWAPP

Attack type	Defense Name	Description	Defense Platform	# of LoC
Side-channel Attack	Cache Guard Instrumented JS APIs	Selectively delay cached response on suspicious requests Instrument sensitive sensor APIs to apply policy	SWAPP Chrome Extension	258 >400
Autofill Abusing Attack	Autofill Guard Insecure Form Warning	Mitigate password stealing attacks in log in forms by isolation with iFrames Disable Autofill for forms to insecure url and send alert to users	SWAPP Chromium	223 ~160
Data Stealing	Data Guard Access Control Management	Reserve sensitive data in secure storage to prevent it from beling stolen Enhance the access control policies to deny unauthorized access	SWAPP Sever-side	325 Manual work
DOM-XSS Attack	DOM Guard DOMPurify.js	Inspect URL parameters to filter XSS payload XSS sanitizer for HTML, MathML, and SVG	SWAPP Client-side	406 1542

channel attacks, autofill abusing attacks, data stealing, and DOM-XSS attacks. These apps in our system can be instantly deployed or updated without waiting for months/years for browsers to officially support the same features.

Furthermore, we roughly quantify the easiness of programmability on our framework by comparing the lines of code (LoC) app developers need to implement the same functionalities (or equivalence) of existing defense mechanisms. From Table 5.3, we can see that the number of LoC of apps in SWAPP is noticeably smaller than that in traditional platforms for most defenses, which suggests that SWAPP can reduce the cost of implementing applicable defense mechanisms.

5.6.4 Overhead

We have shown that SWAPP can help develop security prototypes on the client-side. However, it is also imperative to show that the clients do not suffer as a result. To this end, we evaluate the overhead of SWAPP imposed on a client in four different aspects: the page load time, computational power, heap usage, and network bandwidth. We perform the testing and measurement in a commodity laptop running Ubuntu 18.04 with Intel Core i7-8565U CPU, onboard Intel UHD Graphics 620, and 16GB of memory. We set up four testing configurations, each based on WordPress and phpBB on a local webserver. The first configuration, referred to as the Baseline, represents the original WordPress or phpBB. The second configuration, referred to as EmptySW, has a service worker that simply registers a *fetch* handler without other functionalities. The third configuration has SWAPP installed but does not contain any apps. We refer to this configuration as SWAPP. The fourth configuration has SWAPP installed with four apps activated: Autofill Guard,

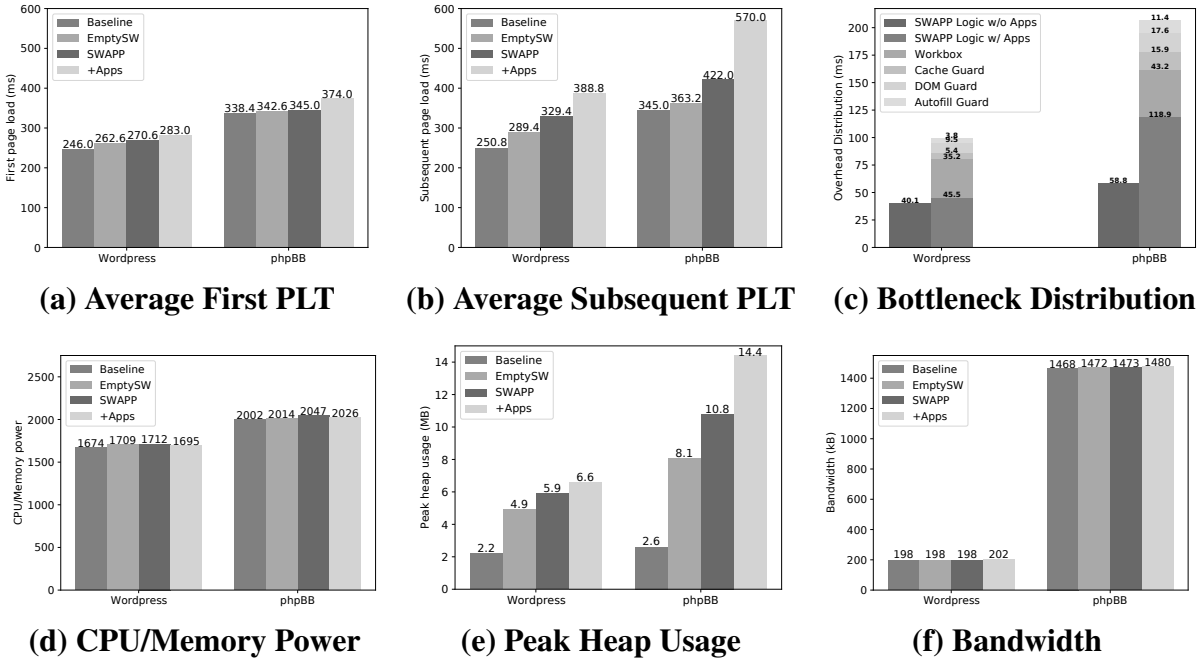

Figure 5.5: Overhead of SWAPP

DOM Guard, Workbox, and Cache Guard. This configuration is referred to as +Apps. Then, we use Chrome's DevTools to measure website visit traces. Both the client and server are running on the same machine, thus the network delay does not need to be taken into account. The results are shown in Figure 5.5.

5.6.4.1 Page Load Time (PLT)

In this evaluation, we compare the average page load time between each configuration in phpBB and WordPress. We make sure that the browser caching is disabled and all site data is cleared for every measurement. The average page load time is calculated among five runs, and it is shown in Figures 3(a) and 3(b).

First Page Load. This scenario represents the first time a user visits the website. As shown in Figure 5.5(a), we observe a gradual increment in the page load time across different settings due to the parsing of additional JavaScript files. Note that the service worker and SWAPP functionalities may not be fully activated during the first-page load. For instance, the initial *fetch* events for the

first-page load will not go through the service worker.

Subsequent Page Load. In the following visits, the service worker will be fully activated. In addition to extra JavaScript files being parsed, we observe an overhead incurred by service worker *activation cost*. Specifically, even when a service worker is empty, the browser still needs to activate the service worker before a request can be fetched. The difference between the EmptySW settings from Figures 3(b) and 3(a) represents this activation cost. On average, an additional 26.8ms (10.26%) and 20.6ms (6.01%) activation cost is incurred to WordPress and phpBB respectively. This behavior is also documented in Google's blog [76], in which navigation preload is used to reduce the activation cost. However, SWAPP cannot preload a resource as prefetching a malicious request can mean the attack is already successful (i.e., the information from the victim already reaches the attacker's server).

SWAPP Overhead. There are two types of overhead introduced by SWAPP: application logic and SWAPP logic. The application logic overhead scales with the number of requests because an app may need to process every request/response. On average, Workbox, Cache Guard, DOM Guard, and Autofill Guard introduce 35.22ms, 5.4ms, 9.5ms, and 3.8ms to WordPress (and 43.2ms, 15.9ms, 17.64ms, and 11.44ms to phpBB) respectively. The distribution of this overhead is illustrated in Figure 5.5(c). Note that the numbers of requests for loading WordPress and phpBB home pages are 18 and 48 respectively. In total, the four apps introduce 53.92ms (20.28%) to the original WordPress and 88.18ms (25.5%) to the original phpBB.

The SWAPP logic overhead scales with the size of each response's body. This is because when an app needs to inspect a response, SWAPP needs to parse the body of the response. We can observe this overhead in phpBB shown in Figure 5.5(c). When there are no apps, SWAPP's logic incurs 58.8ms (17%) overhead. When there are four apps installed, SWAPP's logic incurs 118.8ms (34.4%). On the other hand, WordPress only has 40ms (15.8%) and 45.4ms (18.1%) overhead when there are no apps and when there are four apps, respectively. Note that WordPress requires 50.1kB for loading its home page and phpBB requires 187kB. We observe that parsing the largest response of phpBB incurs almost 20ms alone.

In total, when deploying SWAPP in the original WordPress and phpBB with the four apps, there are 138ms (55%) and 225ms (65.2%) overhead, respectively. Note that for the purpose of evaluation, we enable the four apps for all types of requests. In practice, the developers can configure the apps to selectively activate them for certain pages or types of resources. This could help reduce the overhead of SWAPP and its apps. For instance, the largest file requested by phpBB is a font, which SWAPP spends 20ms parsing but none of the four apps actually takes any action on it. Furthermore, the measurement was conducted in a local environment, and the calculations do not consider the network delay. The network latency depends on several factors, but Google's DevTools would add approximately 300-500ms when the Fast3G setting is applied in our testing environment. Considering an actual user experience with a 400ms network delay, the overhead of SWAPP would be 12% for WordPress without apps and 21.2% with four apps. Similarly, the overhead would be 10.3% for phpBB without apps and 30.2% with four apps. Therefore, we believe SWAPP can be a considerable option for web developers, researchers, and practitioners to quickly develop new prototypes in the future.

5.6.4.2 CPU/Memory Power and Heap Usage

To measure the CPU and memory usage, we utilize Lighthouse, a gadget provided by Chrome's Devtools for measuring the website's overall performance. Lighthouse will load the website and give a score for the CPU and memory power as a numerical value. While this might not be the most accurate method to measure, it gives a value that can be easily compared. Based on the result shown in Figure 5.5(d), we observe no differences between each configuration, thus the CPU and memory utilization overhead are minimal.

To measure the heap usage, we manually define a set of actions that normal users would do (i.e., visit the home page, log in, post a forum's topic or publish a blog, etc.). Then, we collect the heap sampling records. The numbers shown in Figure 5.5(e) represent the peak heap usage during the actions. Note that using samplings may not yield the absolute peak usage, thus we average the numbers across five trials. Overall, we observe a strictly increasing trend in heap usage in both WordPress and phpBB. The inclusion of SWAPP can bring additional 20-30% to the heap usage

compared to an empty service worker, and the apps can introduce 10-30% overhead compared to when there is no app. Nonetheless, the idle heap usage of WordPress and phpBB are similar across all settings with 4.6Mb (+/-5%) and 5.1Mb (+/-7%) respectively.

5.6.4.3 Network Bandwidth

Because we run the client and server for our testing on the same machine, there is no actual network bandwidth. Regardless, Chrome's DevTools can calculate the network bandwidth during a page load, which can correctly measure the amount of bandwidth caused by SWAPP. We collect the network bandwidth of a set of page navigation similar to Section 5.6.4.2. The amount of resources loaded is shown in Figure 5.5(f).

We observe that the amount of additional resources loaded is mostly negligible across all settings. The size of SWAPP is less than 1kB, and it is only loaded on the first page as expected. Although we observe almost a double amount of the number of requests, it does not incur additional resource loaded. The DevTools simply counts a request twice, once for the original request, and second for when the request is handled by the service worker. This also explains why the page load time increases when a website has a running service worker with a *fetch* event handler. For every request, now the browser will have to wake up the service worker to handle the request, which can incur additional computational overhead. Regardless, the amount of extra network bandwidth is still negligible.

5.7 Limitation

Nowadays, most websites (94% according to a recent report [77]) embed at least one third-party resource, e.g., through <script> or <img> tags. Similarly, many websites also make use of cross-origin AJAX requests (XHR). Generally, for a cross-origin XHR to be fetched correctly, the CORS (Cross-Origin Resource Sharing) protocol of the resource must be correctly configured.

Our approach requires network interception, which may include intercepting cross-origin requests. Because the service worker also adheres to the CORS protocol, SWAPP will not be able to read certain request headers or modify the content of cross-origin responses depending on the

CORS configuration [78]. For example, when it is the *no-cors* mode, only simple HTTP headers are accessible and the response properties will be inaccessible to SWAPP. If an app attempts to access such response, it will result in an error.

Among the apps that we developed, Autofill Guard and Data Guard do not access cross-origin resources and only modify HTTP pages, which are from the same origin, while simply forwarding other requests/responses. In the case of Cache Guard, it sets the timer of each request (including cross-origin) when forwarding non-page requests for computing the average network delay. Therefore, it requires CORS-enabled responses, in which the *no-cors* mode would suffice because Cache Guard does not read the response properties.

While the restriction from the CORS protocol limits the direct applicability of SWAPP to seamlessly work with cross-origin resources, we envision SWAPP as a first step toward providing first-party developers a fast-prototyping framework for deploying security applications. With more websites adopting service workers, SWAPP will have better protection coverage. We leave the full integration of SWAPP with third-party resources for future work.

5.8 SWAPP Internet Distribution

We open-source SWAPP, which can be found at `https://github.com/successlab/swapp`, to support more research in this direction.

5.9 Summary

In this chapter, we proposed and open-sourced a framework, SWAPP, to develop security applications on the client side. SWAPP runs in the service worker context and enhances all the attack surfaces (S1-S5). Furthermore, we demonstrated that SWAPP can be used to implement several types of apps including defenses against traditional attacks like XSS, data leakage, side-channel, and autofill abusing attacks. Lastly, we evaluate SWAPP based on the adoptability, compatibility, extensibility/programmability, and efficiency. The result showed that SWAPP can be an alternative for web developers to implement security defenses that can be easily deployed without many requirements.

6. LESSONS LEARNED

In this chapter, we plan to answer the following questions: How are our systems related to and different from each other? What lessons have we learned from designing these systems?

6.1 Summary of Appified Web Security

In this dissertation, there are three types of systems that we provide.

First, systematic analysis of service worker attack surfaces. To this end, we manually investigate the web specification, implementation, and how the service worker works in practice. Furthermore, we survey previous work that utilizes service workers abusively. The accumulation of the obtained knowledge becomes the systematic view of the service worker attack surfaces.

Second, we propose and develop detection systems to identify real vulnerabilities in top appified websites. For instance, SW-Scanner can be used to detect SW-XSS attacks based on surface S1 and extended to detect attacks based on surfaces S2 and S3. Because surfaces S4 and S5 are discussed by previous work, which includes their detection tools, we do not re-implement the wheel and simply refer to their finding results.

Third, we propose and develop defense mechanisms for each attack surface. Our system is called SWAPP. Not only SWAPP can patch the attack surfaces, but it can also provide generic protection against traditional or emerging client-side web attacks. We summarize the attack surface and the corresponding defense in the following.

Table 6.1 shows the summary of all attack surfaces and how our systems can mitigate them. In total, there are five attack surfaces for the service worker.

Table 6.1: Summary of our Attacks and Defenses

Attack Surface	Surface Type	Attack Type	Corresponding SWAPP Defense
S1. SW Registration (§3.2)	Lifecycle	SW-XSS	TCB Module preventing SW registration after SWAPP installation (§5.4.1)
S2. IndexedDB (§4.2)	Communication Channel	SW-XSS	Enhanced IndexedDB (§5.4.3)
S3. Push Message (§4.3)	Communication Channel	Push hijacking	Push Guard (§5.5.5)
S4. Cache [16]	Communication Channel	Side-Channel	Cache Guard (§5.5.1)
S5. postMessage [17]	Communication Channel	XSS	Enhanced postMessage API (§5.4.2)

S1 Attack. The first attack surface, S1, is due to the flaw in the service worker registration lifecycle that allows attackers to manipulate URL search parameters. We find that appified websites blindly trust these parameters and use them inside the service worker leading to SW-XSS vulnerability.

S1 Defense. In order to prevent SW-XSS attacks via SW registration, SWAPP's TCB module disables the registration API entirely once SWAPP is installed. If the website wants to reinstall, it must send the "Clear-Site-Data" HTTP header. While this method does not prevent the attacks that are attempted prior to SWAPP installation, we argue that the attackers will not obtain any meaningful information in this scenario. This is because SWAPP is installed upon the first visit. If the victim never visits the target website, there is likely not any worthwhile data for the attackers to steal.

S2 Attack. Based on the first attack surface, we extend our analysis beyond the lifecycle into the communication channels of a service worker. The second surface S2 extends our SW-XSS attacks onto the IndexedDB channel. While the concept is similar, it is non-trivial to study the prevalence of the vulnerability of this channel in the wild. This is because the IndexedDB entries are used for different purposes than the SW registration parameters. We need to extend another taint tracking tool and utilize real IndexedDB entry to discover the vulnerability. As a result, we find five additional websites that are vulnerable to SW-XSS.

S2 Defense. To prevent SW-XSS via IndexedDB, our system, SWAPP, enhance the IndexedDB API. SWAPP instruments the API to reserve specific database names that cannot be opened in the document context. All scripts in the service worker can use the reserved IndexedDB. In this case, attackers will not be able to manipulate a service worker's internal variables anymore.

S3 Attack. The third surface S3 does not lead to SW-XSS, unlike the first two surfaces. While the impact of this channel is more limited, it still leads to user locations being leaked to attackers. Furthermore, there are more websites that contain this vulnerability due to the popularity of third-party push providers, thus making this attack surface the most prevalent.

S3 Defense. To address this attack surface, we implement a simple SWAPP app called Push

Guard. This app instruments the push subscription API to check for the registering subscription ID. If the ID does not match an allowed list that the developers provide, Push Guard will reject the call.

S4 Attack. This attack surface was discussed by Karami et al. [16]. The attackers utilize differences in page load timing between cached and non-cached resources to determine the user's history. We thoroughly studied this attack surface following Karami's idea and came up with a mitigation method that can be easily deployed and effective.

S4 Defense. We develop a SWAPP app called Cache Guard to prevent side-channel attacks through Cache timing. Cache Guard selectively extends the page load time without affecting the user experience. We show that Cache Guard works effectively and without jeopardizing the website's performance.

S5 Attack. This attack surface was discussed by Son et al. [17]. The attack targets communication between different iFrames. However, the same can be applied to service workers. Therefore, we take the lessons learned from Son's idea to implement an effective mitigation method by enhancing the postMessage API.

S5 Defense. To provide intrinsic security to postMessage API, SWAPP utilizes a secret port established inside the TCB. This port is used when communicating within SWAPP (including both in the document and service worker contexts). By limiting the senders to within a trusted group, attackers can no longer compromise the service worker context.

6.2 Lessons Learned

A benign service worker can be compromised. In the earlier studies of service worker security, the common threat model involves a malicious website starting a service worker to utilize the victim's computational power. Nevertheless, such a model is limited in its impact especially when most previous attacks utilize the computational power to mine crypto-currency. Nowadays, it is much more difficult to mine crypto-currency without a specialized machine. Therefore, we aim to show that it is also possible to compromise a benign service worker. This assumption is more generic than the previous threat model.

As we have shown throughout this dissertation, there are several attack surfaces that can allow attackers to compromise the service worker. Once compromised, the attackers can also steal sensitive information leading to more severe consequences than in previous attacks. Hence, we learn that despite security mechanisms in service workers, it is possible to compromise or leverage a benign service worker.

While the service worker introduces new attack vectors, it can also be used to enhance client-side security. The service worker introduces several attack surfaces to the client-side. However, once these surfaces are patched, the unique capabilities of service workers can be utilized by developers to implement innovative applications. As shown in Chapter 5, SWAPP allows several security apps to be implemented and deployed easily. Furthermore, SWAPP can even implement non-security apps such as the wrapped Workbox app discussed in Section 5.6.2. As we are the first to utilize service workers to improve client-side security, we envision that SWAPP will be the first step toward more innovations in the future.

Service worker adoption is increasing. Since the start of our study on service workers (2018), we have found that the service worker adoption rate increases every year. By the time we do our most recent data collection (2020), we found that the number of appified websites increased from 2,700 to 7000 websites. Based on this growth, we expect service workers to be adopted by more websites in the future.

One potential reason why more websites start to adopt service workers could be that new features are also being implemented for service workers. We noticed that initially, the service worker did not support many event types, thus many websites did not find service workers attractive. Because service workers are constantly being improved, we see new features (such as payment APIs) being added and tested by browser developers. We expect the service worker to be more powerful in the future, which in turn makes its security even more important.

7. CONCLUSION AND FUTURE WORK

7.1 Conclusion

The service worker is a recent web technology that enables appification of websites turning traditional websites into native mobile apps or desktop software. It comes with unique capabilities including network traffic interception and spontaneous activation of its code. These both bring new attack surfaces for attackers to target and new opportunities for developers to utilize the service worker for innovation.

In this dissertation, we systematically analyzed the security model of the service worker. First, we analyzed the service worker lifecycle and discovered a new variant of XSS attacks, called SW-XSS. Second, we extended our analysis to the communication channels used by service workers. We identified two channels that can be utilized by attackers to launch SW-XSS attacks and push hijacking attacks. Third, we measured the prevalence of these attack surfaces in the top 100,000 websites using our developed taint tracking tool called SW-Scanner. The large-scale study showed that 40, 5, and 200 benign websites could be compromised through SW-XSS (via registration), SW-XSS (via IndexedDB), and push hijacking attacks respectively. As a result, we satisfied the goal G1 as discussed in Section 1.2.

Furthermore, we proposed a platform utilizing the service worker as the root of trust on the client-side to implement security applications. Our platform, SWAPP, patched all attack surfaces exposed to service workers, thus preventing the aforementioned attacks against service workers. More importantly, SWAPP can also be used to implement innovative defenses that can mitigate against other traditional web attacks such as DOM-XSS, data leakage, and side-channel attacks. Therefore, we satisfied goal G2.

7.2 Future Work

There are two paths that I intend to work on in the future. First, I want to expand my knowledge of web security to the Web3, which incorporate blockchain mechanisms with traditional websites

to provide decentralization. To this end, I am looking toward phishing attack studies in the domain of Web3 as the interaction between users and Web3 providers is different than traditional websites. Furthermore, I want to potentially provide mitigation methods against Web3-specific phishing attacks, especially regarding anti-phishing wallet. Second, I want to continue my research on service worker security as follows.

- **Improvement on the SWAPP efficiency and implementation of other applications using SWAPP.** Currently, SWAPP incurs an acceptable but considerable overhead. We plan to further optimize SWAPP performance by selectively parsing resources that are useful while skipping other resources. Furthermore, we want to fully support third-party resources on the website. Due to the limitation of CORS, SWAPP cannot fully control third-party content. An alternative is to build SWAPP natively as a browser extension or modification. Lastly, we want to develop more apps using SWAPP as SWAPP apps can easily be installed. We believe that future studies on new SWAPP apps will be useful for future researchers.

- **Unexpected service worker registration using first-party scripts.** One research direction I am considering is using existing first-party scripts as gadgets and registering the initial gadget as a service worker to import malicious code into the service worker context. The intuition is that the service worker does not allow third-party codes to be registered. However, it does not limit normal first-party scripts running in the document context to being registered. We notice that much first-party code running in the document context contains code importing mechanisms such as importScripts (in normal web workers) and module import. If such scripts allow the importing URL to be controlled, it is possible to forcefully register the script as a service worker and control the importing URL to point to the attacker's server. We aim to explore the use of these script gadgets to compromise service workers in the future.

- **Formal method for analyzing service workers.** Another direction is to apply formal methods for analyzing service workers similar to what Fett et al. did on OAUTH2 [79]. In my

work, I have studied the attack surfaces and defenses of service workers. However, service workers do not just interact with a website's code but also it can interact with other aspects such as web browsers. I plan to extend my knowledge and provide formal verification of service worker security in all aspects.

REFERENCES

[1] J. Lee, H. Kim, J. Park, I. Shin, and S. Son, "Pride and prejudice in progressive web apps: Abusing native app-like features in web applications," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, (New York, NY, USA), pp. 1731–1746, ACM, 2018.

[2] P. Papadopoulos, P. Ilia, M. Polychronakis, E. P. Markatos, S. Ioannidis, and G. Vasiliadis, "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, 2019.

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

[4] P. H. Phung, D. Sands, and A. Chudnov, "Lightweight self-protecting javascript," in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, 2009.

[5] "Snort." `https://www.snort.org/`.

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

[7] "NoScript." `https://noscript.net/`.

[8] "HTTPS Everywhere." `https://www.eff.org/https-everywhere`.

[9] L. Meyerovich and B. Livshits, "Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser," in *Proceedings - IEEE Symposium on Security and Privacy*, January 2010.

[10] D. Kohlbrenner and H. Shacham, "Trusted browsers for uncertain times," in *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, USENIX Association, 2016.

[11] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia, "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, 2018.

[12] "XSS Filters." `https://github.com/YahooArchive/xss-filters`.

[13] "JS-XSS." `https://github.com/leizongmin/js-xss`.

[14] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, The Internet Society, 2017.

[15] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen, "Tranco: A research-oriented top sites ranking hardened against manipulation," in *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, NDSS 2019, Feb. 2019.

[16] S. Karami, P. Ilia, and J. Polakis, "Awakening the web's sleeper agents: Misusing service workers for privacy leakage," in *28th Annual Network and Distributed System Security Symposium, NDSS 2021, San Diego, California, USA, February 21-24, 2021*, The Internet Society, 2021.

[17] S. Son and V. Shmatikov, "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, 2013.

[18] C. Guan, K. Sun, Z. Wang, and W. T. Zhu, "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* (X. Chen, X. Wang, and X. Huang, eds.), pp. 629–640, ACM, 2016.

[19] M. Squarcina, S. Calzavara, and M. Maffei, "The remote on the local: Exacerbating web attacks via service workers caches," in *15th IEEE Workshop on Offensive Technologies (WOOT 21)*, IEEE, 2021.

[20] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, (New York, NY, USA), p. 921–930, Association for Computing Machinery, 2010.

[21] "Autofill Report." `https://bugs.chromium.org/p/chromium/issues/detail?id=448539`.

[22] "Autofill Attack." `https://github.com/anttiviljami/browser-autofill-phishing`.

[23] "Adblock Usage Statistics." `https://backlinko.com/ad-blockers-users`.

[24] D. Akhawe and A. P. Felt, "Alice in warningland: A large-scale field study of browser security warning effectiveness," in *22nd USENIX Security Symposium (USENIX Security 13)*, (Washington, D.C.), pp. 257–272, USENIX Association, Aug. 2013.

[25] "Service Worker Support." `https://caniuse.com/?search=service%20worker`.

[26] M. Johns, B. Engelmann, and J. Posegga, "Xssds: Server-side detection of cross-site scripting attacks," in *2008 Annual Computer Security Applications Conference (ACSAC)*, pp. 335–344, 2008.

[27] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel, "Swap: Mitigating xss attacks using a reverse proxy," in *2009 ICSE Workshop on Software Engineering for Secure Systems*, pp. 33–39, 2009.

[28] P. Chaudhary, B. B. Gupta, C. Choi, and K. T. Chui, "Xsspro: Xss attack detection proxy to defend social networking platforms," in *Computational Data and Social Networks* (S. Chellappan, K.-K. R. Choo, and N. Phan, eds.), (Cham), pp. 411–422, Springer International Publishing, 2020.

[29] S. Lekies, K. Kotowicz, S. Groß, E. A. Vela Nava, and M. Johns, "Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, (New York, NY, USA), p. 1709–1723, Association for Computing Machinery, 2017.

[30] K. Bhargavan, A. Delignat-lavaud, and S. Maffeis, "Defensive javascript building and verifying secure web components."

[31] T. Tran, R. Pelizzi, and R. Sekar, "Jate: Transparent and efficient javascript confinement," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, (New York, NY, USA), pp. 151–160, ACM, 2015.

[32] Y. Zhou and D. Evans, "Understanding and monitoring embedded web scripts," in *Proc. IEEE Symp. Security and Privacy*, pp. 850–865, May 2015.

[33] P. Saxena, S. Hanna, P. Poosankam, and D. Song, "FLAX: systematic discovery of client-side validation vulnerabilities in rich web applications," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*, The Internet Society, 2010.

[34] A. Mendoza and G. Gu, "Mobile application web API reconnaissance: Web-to-mobile inconsistencies & vulnerabilities," in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pp. 756–769, IEEE, 2018.

[35] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. V. Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "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* (T. Yu, G. Danezis, and V. D. Gligor, eds.), pp. 736–747, ACM, 2012.

[36] "Babel." `https://babeljs.io/`.

[37] "Iroh." `https://maierfelix.github.io/Iroh/`.

[38] "Iroh events." https://github.com/maierfelix/Iroh/blob/master/API.md.

[39] "Similar Web." https://www.similarweb.com/.

[40] "Wayback Machine." https://web.archive.org/.

[41] "Open Bug Bounty." https://www.openbugbounty.org/.

[42] "CSP3." https://www.w3.org/TR/CSP3/#framework-directive-source-list.

[43] "Absolute Path." https://tools.ietf.org/html/rfc3986#section-3.3.

[44] "Manifest File." https://developer.mozilla.org/en-US/docs/Web/Manifest.

[45] B. Stock, M. Johns, M. Steffens, and M. Backes, "How the web tangled itself: Uncovering the history of client-side web (in)security," in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. (E. Kirda and T. Ristenpart, eds.), pp. 971–987, USENIX Association, 2017.

[46] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, "Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, (Vienna, Austria), 2016.

[47] "Kaspersky report on stalkerware." https://www.kaspersky.com/about/press-releases/2019_could-someone-be-spying-on-you-through-your-phone.

[48] "Stalkerware." https://www.cyberscoop.com/stalkerware-pandemic-coronavirus-domestic-violence/.

[49] "Location-triggered notification." https://documentation.onesignal.com/docs/location-triggered-event#section-web-setup.

[50] "Geofencing on push notification." https://retailtouchpoints.com/features/executive-viewpoints/geofencing-and-mobile-push-notifications-a-match-made-in-customer-engagement-heaven.

[51] "Pushwoosh geo-based notification." https://www.pushwoosh.com/blog/geo-based-push-notifications/.

[52] S. M. Mirtaheri, M. E. Dinçktürk, S. Hooshmand, G. V. Bochmann, G.-V. Jourdan, and I. V. Onut, "A brief history of web crawlers," *arXiv preprint arXiv:1405.0749*, 2014.

[53] "OneSignal Report." https://onesignal.com/blog/increase-opt-in-rates-for-push-notifications/.

[54] "AWS Location-based Marketing Report 2018." https://s3.amazonaws.com/factual-content/marketing/downloads/LocationBasedMarketingReport_Factual.pdf.

[55] "AWS Location-based Marketing Report 2019." https://s3.amazonaws.com/factual-content/marketing/downloads/Factual-2019-Location-Based-Market-Report.pdf.

[56] "Cookie store api." https://wicg.github.io/cookie-store/.

[57] "Chromium Push Issue." https://bugs.chromium.org/p/chromium/issues/detail?id=803106.

[58] "HackerOne Report." https://www.hackerone.com/top-ten-vulnerabilities.

[59] "Workbox." https://developers.google.com/web/tools/workbox.

[60] M. Steffens and B. Stock, "Pmforce: Systematically analyzing postmessage handlers at scale," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, (New York, NY, USA), p. 493–505, Association for Computing Machinery, 2020.

[61] "Forbidden HTTP headers." https://developer.mozilla.org/en-US/docs/Glossary/Forbidden_header_name.

[62] A. Shusterman, A. Agarwal, S. O'Connell, D. Genkin, Y. Oren, and Y. Yarom, "Prime+probe 1, javascript 0: Overcoming browser-based side-channel defenses," in *Proceedings of the 30th USENIX Security Symposium*, Proceedings of the 30th USENIX Security Symposium, pp. 2863–2880, USENIX Association, Jan. 2021.

[63] "XSS Auditor Deprecation." https://www.chromium.org/developers/design-documents/xss-auditor.

[64] S. Gupta and B. B. Gupta, "Xss-safe: A server-side approach to detect and mitigate cross-site scripting (xss) attacks in javascript code," *ARABIAN JOURNAL FOR SCIENCE AND ENGINEERING*, vol. 41, October 2015.

[65] "CSRF Prevention." https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html.

[66] H. Shahriar, S. North, W.-C. Chen, and E. Mawangi, "Design and development of anti-xss proxy," in *8th International Conference for Internet Technology and Secured Transactions (ICITST-2013)*, pp. 484–489, 2013.

[67] M. Steffens, C. Rossow, M. Johns, and B. Stock, "Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild," in *NDSS*, 2019.

[68] D. Silver, S. Jana, D. Boneh, E. Chen, and C. Jackson, "Password managers: Attacks and defenses," in *23rd USENIX Security Symposium (USENIX Security 14)*, (San Diego, CA), pp. 449–464, USENIX Association, Aug. 2014.

[69] B. Stock and M. Johns, "Protecting users against xss-based password manager abuse," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, (New York, NY, USA), p. 183–194, Association for Computing Machinery, 2014.

[70] "CVE-2021-43175." https://nvd.nist.gov/vuln/detail/CVE-2021-43175.

[71] "CVE-2021-41277." https://cve.mitre.org/cgi-bin/cvename.cgi?name=2021-41277.

[72] "HTML Encoder." https://github.com/mathiasbynens/he.

[73] "XSS Payload." https://github.com/payloadbox/xss-payload-list.

[74] "Can I use." https://caniuse.com/.

[75] "Workbox CLI." https://developer.chrome.com/docs/workbox/modules/workbox-cli/.

[76] "Google's Blog Service Worker Boot Up Delay." https://developers.google.com/web/updates/2017/02/navigation-preload/.

[77] "Third-party script usage." https://almanac.httparchive.org/en/2021/third-parties.

[78] "CORS modes." https://developer.mozilla.org/en-US/docs/Web/API/Request/mode.

[79] D. Fett, R. Küsters, and G. Schmitz, "A comprehensive formal security analysis of oauth 2.0," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1204–1215, 2016.