SWAPP: A New Programmable Playground for Web Application Security

Phakpoom Chinprutthiwong
SUCCESS Lab
Texas A&M University

Jianwei Huang
SUCCESS Lab
Texas A&M University

Guofei Gu
SUCCESS Lab
Texas A&M University

Abstract

Client-side web attacks are one of the major battlefields for cybercriminals today. To mitigate such attacks, researchers have proposed numerous defenses that can be deployed on a server or client. Server-side defenses can be easily deployed and modified by web developers, but it lacks the context of client-side attacks such as DOM-XSS attacks. On the other hand, client-side defenses, especially in the form of modified browsers or browser extensions, require constant vendor support or user involvement to be up to date.

In this work, we explore the feasibility of using a new execution context, the service worker context, as a platform for web security defense development that is programmable, browser agnostic, and runs at the client side without user involvement. To this end, we propose and develop SWAPP (Service Worker APplication Platform), a framework for implementing security mechanisms inside a service worker. As the service worker is supported by most browsers, our framework is compatible with most clients. Furthermore, SWAPP is designed to enable the extensibility and programmability of the apps. We demonstrate the versatility of SWAPP by implementing various apps that can mitigate web attacks including a recent side-channel attack targeting websites that deploy a service worker. SWAPP allows websites to offload a part of the security tasks from the server to the client and also enables the possibility to deploy or retrofit emerging security features/prototypes before they are officially supported by browsers. Finally, we evaluate the performance overhead of our framework and show that deploying defenses on a service worker is a feasible option.

1 Introduction

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 [11]. In response to new attacks, researchers have proposed many defense/detection mechanisms that require web browser modification (browser-centric) [34,38,39], manual installation, i.e., as browser extensions (user-centric) [13,15,43], or server-side modifications (server-centric) [17,42]. 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.

For browser-centric defenses, generally, browser developers already put much effort into providing the most secure environment to run a website. Nevertheless, a slight inconsistency between different browser vendors or versions can create a gap that allows attackers to compromise the users. This is because there is usually a time gap for a proposed defense to be officially supported by different browsers and widely deployed. Any outdated client is still at the risk of being compromised. Additionally, proposed prototypes of browser-centric defenses such as BEEP [31] usually require browser modifications. It is not feasible for such prototypes to be widely adopted/deployed without constant support from browser vendors. Even the proposal of Content-Security-Policy took two years before the W3C published the first standard [49].

In the case of user-centric approaches, users are mostly assumed to discover and deploy additional security mechanisms (i.e., browser extensions) by themselves. Unfortunately, only a small amount of users are aware of the latest security risks and deploy the defense mechanisms. As studied by previous works [41], some users are unaware of commercial security tools such as password managers. Some users even ignore security cues and warnings presented by browsers in favor of convenience [24]. Therefore, it is imperative for web developers to be the active party in protecting their users from client-side attacks.

For web developers to mitigate client-side attacks using server-centric approaches, they can deploy a defense mechanism on the server. For instance, Snort and XSSDS [17,32] can be used to set up a server to detect attacks. Nevertheless, server-side defenses lack the context of client-side attacks. For instance, DOM-XSS attacks can include the payload after the URL segment (“#"), which does not get sent to the web-

server. Add-on XSS [26] also executes JavaScript directly in the victim's address bar.

To mitigate the aforementioned limitations, a client-side framework for security functionalities is required. 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 [19] (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.

In this paper, we introduce SWAPP (Service Worker APplication Platform), a new development framework for developing security prototypes and applications. SWAPP is a generalized platform that can be used for any website or in an enterprise setting accessible from specific networks such as business applications. To achieve the first goal, we implement SWAPP to be deployed inside a service worker, which has been supported by all mainstream web browsers [16]. Nonetheless, it is non-trivial to provide a secure environment for apps to run as parts of SWAPP. While the service worker is designed with security as a priority, existing works [28, 29] show that it can still be compromised. 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 4.3).

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.

Allowing fast prototyping of apps through SWAPP is our third goal. To this end, 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 [33] targeting websites with a service worker (Section 5).

Finally, we evaluate the overhead of SWAPP using two popular open-source web applications, WordPress and phpBB, which can integrate SWAPP by modifying a few lines of original code. The results show that the core of SWAPP (without any apps running) incurs 40ms (15.8%) additional home page load time to a vanilla WordPress and 58.8ms (17%) to a vanilla phpBB. With four apps running (Workbox, Cache Guard, Autofill Guard, and DOM Guard), SWAPP incurs 138ms (55%) and 225ms (65.2%) to a vanilla WordPress and phpBB, 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, we find that the largest file (a font) requested by phpBB alone requires SWAPP 20ms to parse 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 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 a service worker can be a feasible option to deploy client-side defenses in the future.

Our main contributions are as follows:

- We propose and implement SWAPP, a new framework for developing security mechanisms inside a service worker. The source code of SWAPP and its apps that we implement are accessible[1]. (Section 4)
- We implement security apps using SWAPP to demonstrate the practicality of the new approach. The developed apps can be easily extended/exported and deployed to mitigate several types of web attacks. (Section 5)
- We evaluate the overhead of SWAPP using two open-source applications (Wordpress and phpBB). The result shows that SWAPP incurs 40ms (15.8%) and 58.8ms (17%) additional page load time to the base Wordpress and phpBB respectively. (Section 6)

[1] https://github.com/successlab/swapp

Table 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

2 Background and Motivation

In this section, we first provide the background on what is a service worker and how it works inside a website. Then, we discuss the limitations of existing web defenses. Finally, we explain why a service worker can become a new playground for implementing security mechanisms.

2.1 Service Worker

A service worker is a type of web worker and can be registered from the document context (i.e., the DOM). Essentially, it is defined as a JavaScript file that must be hosted on the same origin as the website (but additional files can be imported from cross origins). Generally, websites will automatically install a service worker when users visit their home page. Once installed, a service worker runs in an isolated execution context, thus scripts from the document context cannot directly access the service worker. Any information between the service worker and the DOM is normally exchanged through the *postMessage* API.

The service worker has unique capabilities and operates asynchronously based on events. For instance, it can act as a proxy, intercepting a network request, which triggers a *fetch* event for the service worker to handle. The service worker can register an event handler using the *addEventListener* API. Table 1 shows the list of service worker events according to the current W3C specification.

2.2 Existing Defenses and Their Limitations

In this work, 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 [49].

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 [3] with a proof of concept attack shown two years later [2]. 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 framework will allow 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 [43] 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 [1]. 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 our framework is automatically installed by default along with the service worker, it is more controllable by web developers and reachable to the user clients because more than 95% of running browsers support service workers [16].

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 [17, 32], 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, 54]. On the other hand, client-side defenses that run in the document context such as XSS filters [14, 22] share the execution context with attackers. This put them at risk of getting bypassed or manipulated at run-time [37]. Therefore, they are alternatives proposed in the form of defensive JS [25]. Nonetheless, defensive JS solutions may require major changes to the legacy code. Our proposed solution, on the other hand, does not require many changes to the legacy code as we later show in Section 6.2.

2.3 A New Playground: SW-Centric Defense

In this work, we propose a different type of server-centric defense that runs in a service worker, which we call SW-centric defense for simplicity. 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 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 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 6.3.

In order to achieve the three goals, we have to carefully address several challenges while designing our platform. This is because the service worker environment is not initially designed to support multiple apps utilizing the same event handler. We provide details regarding these challenges in Section 4.1.

3 Threat Model

In this work, 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, e.g., attacks discussed by our early work [28, 29] and Steffens et al. [50]. 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 [33]. Further detail on this type of attacker will be discussed in Section 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 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.

4 SWAPP System Design

In this section, we present our framework, SWAPP. First, we discuss three technical challenges for SWAPP. Then, we illustrate and elaborate on our design of SWAPP in response to the challenges. Finally, we show the development interfaces provided to developers and go through the overall workflow of our system.

4.1 Technical Challenges

4.1.1 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 allows 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 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 (more details will be discussed in Section 4.2.1).

4.1.2 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 (further discussed in Section 4.2.2).

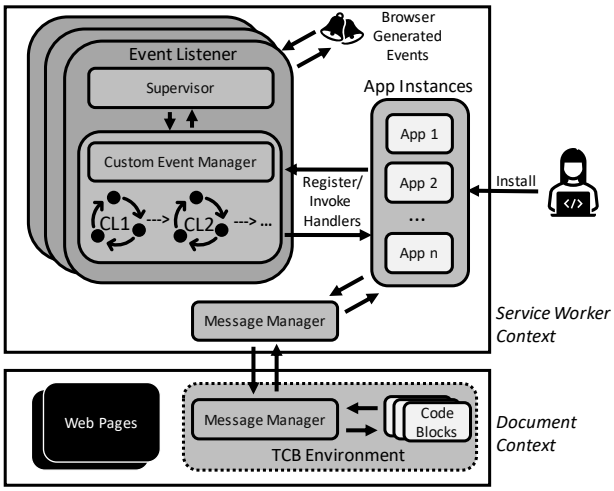


Figure 1: SWAPP Overview Architecture

4.1.3 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, our prior studies [28, 29] have 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. [47] also discussed an attack using the *postMessage* API, which Steffens et al. [51] 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. We further discuss how we reinforce SWAPP against these attack vectors in Section 4.3.

4.2 Overall Design

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.

4.2.1 Supervisor

The Supervisor resides in the service worker context. It is deployed within an event listener. The main purpose of the Su-

pervisor 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. 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 to handle. We call events distributed from the queue to apps 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 [9]. 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 4.2.4).

4.2.2 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 receiving 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.

4.2.3 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 [43] 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 4.3.1.

4.2.4 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 4.3.2). 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.

4.3 SWAPP Security Analysis and Design

SWAPP itself can be the target of attackers (TC3). Here we elaborate on how we enhance SWAPP against threats from different attack vectors. Primarily, there are three channels that attackers in our threat model can leverage against SWAPP: document context, postMessage, and IndexedDB.

4.3.1 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. [43]. Nonetheless, apps may evoke JS APIs that are not originally frozen inside the TCB, and attackers can manipulate these objects [45]. 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. 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.

4.3.2 Security of postMessage

Existing works discussed the Postman attack [47], which utilizes the *postMessage* (PM) to attack a different context such as iFrames, and its prevalence [51]. 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 such attacks 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 2.

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 4.3.1, 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.

4.3.3 Security of IndexedDB

Our previous work [29] shows that despite service workers executing in an isolated context, they can still be compromised through the IndexedDB. 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.

Table 2: 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

4.4 SWAPP Usage and Interface

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 developers. Currently, the core of SWAPP provides four types of primitives[2] (with corresponding interfaces shown in Table 2).

- *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 [21, 30], CSRF detection [6], or proxy-based mechanisms [44, 54].
- *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 [43].
- *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 [50]). Thus, our new primitives can help restrict such unauthorized access and ensure attackers will not be able to manipulate app data.

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

High-level functionalities can then be provided or developed based on these primitives. We demonstrate how these primitives can be combined to construct a security app in Section 5.

5 SWAPP App Examples

In this section, we present four example apps to show how security apps can be constructed using SWAPP interfaces.

5.1 Cache Guard

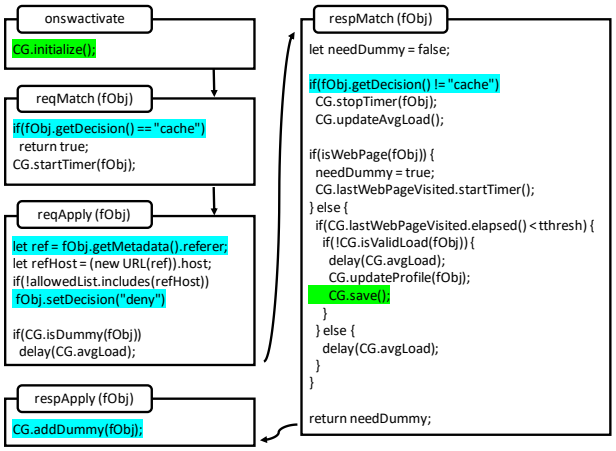


Figure 2: Workflow of Cache Guard. The coloring represents the usage of different groups of SWAPP interfaces (either as a direct invocation or a function containing the interface). Green represents a secure storage management interface. Blue represents the network manipulation interface. For instance, *CG.initialize* utilizes our secure storage to load settings and previous statistics thus is labeled in green.

Karami et al. have recently discussed privacy-invasive attacks on websites utilizing service workers for caching [33]. 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) [36]. Note that this attack can work even when the X-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 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.

5.2 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. [46] and Stock et al. [52] 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 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. [52], 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,

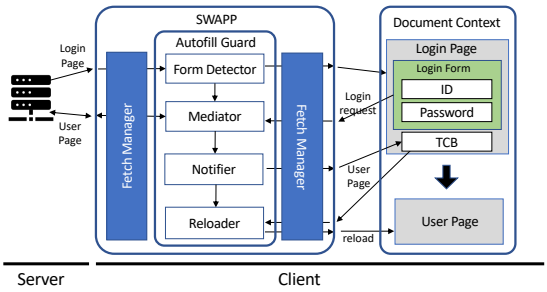


Figure 3: Workflow of Autofill Guard

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.

5.3 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 in many websites. For resources that contain privacy (e.g. URL to a private file on the website or privileged operations [8] [7]), 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 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 4.

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 1. For

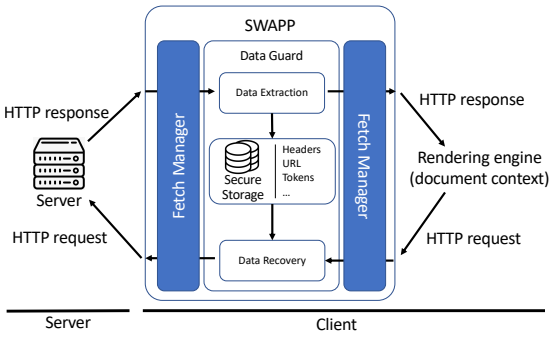


Figure 4: Workflow of Data Guard

each element identified by 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 in 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.

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

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

Listing 1: Data Guard Undocumented Data Extension Template

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 [12]. 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 [27]. 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 [23]. 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.

6 Evaluation

In this work, we aim to demonstrate an alternative method to enhance website security. As a result, our evaluation will focus on four aspects: adoptability (Section 6.1), compatibility (Section 6.2), extensibility/programmability (Section 6.3), and efficiency (Section 6.4).

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 [4]. 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 [35] list obtained in April 2021[3]. 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 3 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 countermeasures 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.

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 5.1.

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 [20] 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 2. Finally, we modify the generated service worker

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

Table 3: 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)

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.

```
[WorkboxApp.js]
+ var wbApp = new Object();
+ wbApp.reqMatch = function(fObj) {
+         return true;
+ };

+ self.importScripts("workbox-sw.js");
+ f2fInst.addApp(wbApp);

[workbox-sw.js]
addFetchListener() {
-  self.addEventListener("fetch", (e => {
-    const {
-        request: t
-    } = e, s = this.handleRequest({
-        request: t,
-        event: e
-    });
-    s && e.respondWith(s)
-  }))

+  let ref = this;
+  wbApp.reqApply = async function(fObj) {
+    let e = fObj.getMetadata();
+        const {
+      request: t
+    } = e, s = await ref.wbHandleRequest({
+      request: e,
+      event: e
+    });
+    let b = await s.text();
+    fObj.setMeta(s);
+    fObj.setBody(b);
+    fObj.setDecision("cache");
+    return fObj;
  }
}
```

Listing 2: Migrating Workbox to SWAPP

6.3 Extensibility and Programmability

In Section 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 4, SWAPP provides a unified platform that can be used to develop various defense solutions against different types of web attacks such as side-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.

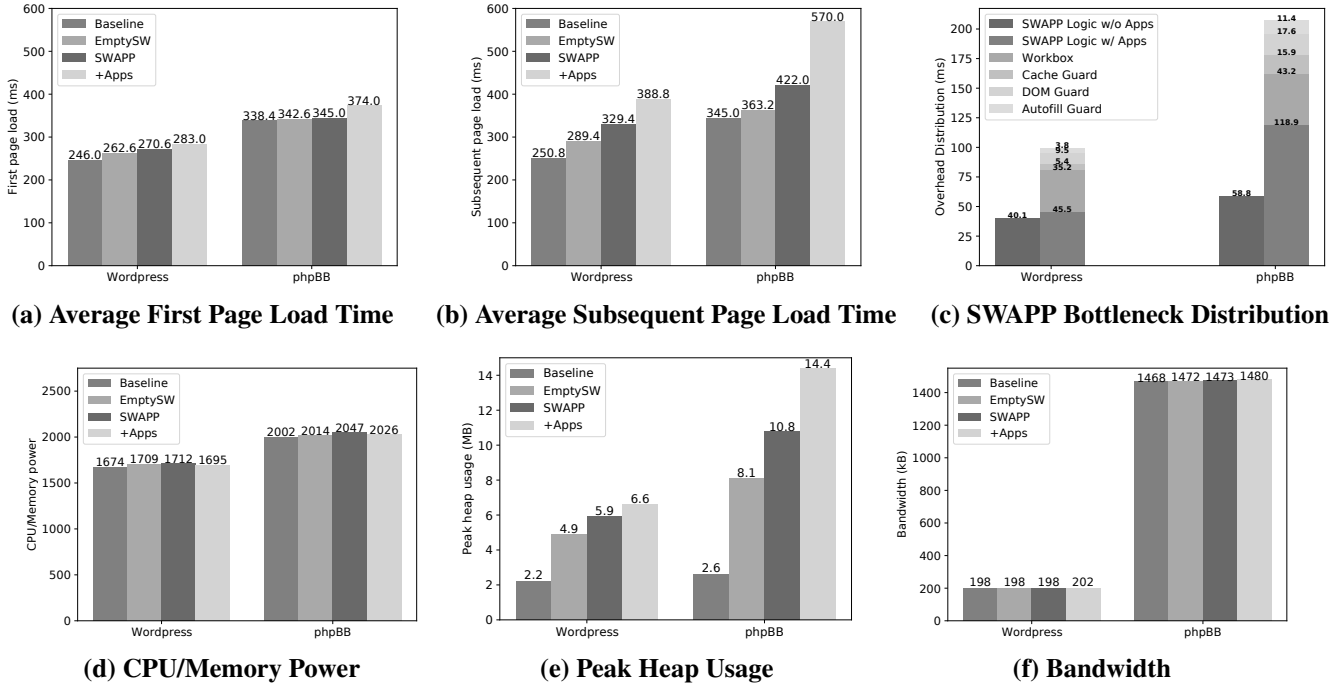
Furthermore, we roughly quantify the easiness of programmability on our framework by comparing the lines of code (LoC) app developers need to implement the same functionalities (or equivalence) of existing defense mechanisms. From Table 4, we can see that the number of LoC of apps in SWAPP is noticeably smaller than that in traditional platforms for most defenses, which suggests that SWAPP can reduce the cost of implementing applicable defense mechanisms.

6.4 Overhead

We have shown that SWAPP can help develop security prototypes at 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, 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.

Table 4: 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



(a) Average First Page Load Time **(b) Average Subsequent Page Load Time** **(c) SWAPP Bottleneck Distribution**

(d) CPU/Memory Power **(e) Peak Heap Usage** **(f) Bandwidth**

Figure 5: Overhead of SWAPP

6.4.1 Page Load Time

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(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 [10], 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(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(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.

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(d), we observe no differences between each configuration, thus the CPU and memory utilization overhead is 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(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.

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 6.4.2. The amount of resources loaded is shown in Figure 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.

7 Limitation and Discussion

Nowadays, most websites (94% according to a recent report [18]) 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 [5]. 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.

8 Related Work

In addition to existing web defenses discussed in Section 2.2, the security of service workers is an emerging research topic that recent researchers have discussed. Lee et al. [36] and Papadopoulos et al. [40] 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 [53]. In more recent work [28], we showed a novel type of Cross-site scripting called SW-XSS that can let attackers execute malicious code inside a benign service worker and potentially hijack it. Karami et al. [33] and Squarcina et al. [48] 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.

9 Conclusion

In this paper, we propose SWAPP, a unified client-side security framework deployed on the service worker. SWAPP allows web developers to offload a part of security tasks from server to client and enables the possibility to deploy emerging security features before they are supported by mainstream browsers and widely deployed on user devices. We developed example apps to show the high programmability and extensibility of SWAPP. Compared to traditional server-side implementation, security features can be implemented in SWAPP and easily deployed by developers in a more timely and flexible manner. The evaluation results prove that SWAPP can introduce its benefits to the ecosystem with reasonable overhead. We believe that SWAPP offers a powerful new framework for prototyping and delivering innovative security applications into the rapidly evolving world of web development.

ACKNOWLEDGEMENTS

We want to thank our shepherd Yinzhi Cao and the anonymous reviewers for their valuable comments. This material is based upon work supported in part by the National Science Foundation (NSF) under Grant No. 1700544 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.

References

[1] Adblock Usage Statistics. https://backlinko.com/ad-blockers-users.

[2] Autofill Attack. https://github.com/anttiviljami/browser-autofill-phishing.

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

[4] Can I use. https://caniuse.com/.

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

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

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

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

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

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

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

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

[13] HTTPS Everywhere. https://www.eff.org/https-everywhere.

[14] JS-XSS. https://github.com/leizongmin/js-xss.

[15] NoScript. https://noscript.net/.

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

[17] Snort. https://www.snort.org/.

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

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

[20] Workbox CLI. https://developers.google.com/web/tools/workbox/modules/workbox-cli.

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

[22] XSS Filters. https://github.com/YahooArchive/xss-filters.

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

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

[25] Karthikeyan Bhargavan, Antoine Delignat-lavaud, and Sergio Maffeis. Defensive javascript building and verifying secure web components.

[26] Yinzhi Cao, Chao Yang, Vaibhav Rastogi, Yan Chen, and Guofei Gu. Abusing browser address bar for fun and profit - an empirical investigation of add-on cross site scripting attacks. volume 152, pages 582–601, 11 2015.

[27] Pooja Chaudhary, B. B. Gupta, Chang Choi, and Kwok Tai Chui. Xsspro: Xss attack detection proxy to defend social networking platforms. In Sriram Chellappan, Kim-Kwang Raymond Choo, and NhatHai Phan, editors, *Computational Data and Social Networks*, pages 411–422, Cham, 2020. Springer International Publishing.

[28] Phakpoom Chinprutthiwong, Raj Vardhan, GuangLiang Yang, and Guofei Gu. Security study of service worker cross-site scripting. ACSAC '20, page 643–654, New York, NY, USA, 2020. Association for Computing Machinery.

[29] Phakpoom Chinprutthiwong, Raj Vardhan, GuangLiang Yang, Yangyong Zhang, and Guofei Gu. The service worker hiding in your browser: The next web attack target? RAID '21, page 312–323, New York, NY, USA, 2021. Association for Computing Machinery.

[30] Shashank 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*, 41, 10 2015.

[31] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, page 601–610, New York, NY, USA, 2007. Association for Computing Machinery.

[32] Martin Johns, Björn Engelmann, and Joachim Posegga. Xssds: Server-side detection of cross-site scripting attacks. In *2008 Annual Computer Security Applications Conference (ACSAC)*, pages 335–344, 2008.

[33] Soroush Karami, Panagiotis Ilia, and Jason 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.

[34] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16. USENIX Association, 2016.

[35] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter 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, February 2019.

[36] Jiyeon Lee, Hayeon Kim, Junghwan Park, Insik Shin, and Sooel 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, pages 1731–1746, New York, NY, USA, 2018. ACM.

[37] Sebastian Lekies, Krzysztof Kotowicz, Samuel Groß, Eduardo A. Vela Nava, and Martin 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, page 1709–1723, New York, NY, USA, 2017. Association for Computing Machinery.

[38] William Melicher, A. Das, Mahmood Sharif, L. Bauer, and Limin Jia. Riding out domsday: Towards detecting and preventing dom cross-site scripting. In *NDSS*, 2018.

[39] Leo Meyerovich and Ben Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. 01 2010.

[40] Panagiotis Papadopoulos, Panagiotis Ilia, Michalis Polychronakis, Evangelos P. Markatos, Sotiris Ioannidis, and Giorgos 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.

[41] Sarah Pearman, Shikun Aerin Zhang, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Why people (don't) use password managers effectively. In *Proceedings of the Fifteenth USENIX Conference on Usable Privacy and Security*, SOUPS'19, page 319–338, USA, 2019. USENIX Association.

[42] Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight self-protecting javascript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, 2009.

[43] Michael Schwarz, Moritz Lipp, and D. Gruss. Javascript zero: Real javascript and zero side-channel attacks. In *NDSS*, 2018.

[44] Hossain Shahriar, Sarah North, Wei-Chuen Chen, and Edward Mawangi. Design and development of anti-xss proxy. In *8th International Conference for Internet Technology and Secured Transactions (ICITST-2013)*, pages 484–489, 2013.

[45] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval 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, pages 2863–2880. USENIX Association, January 2021.

[46] David Silver, Suman Jana, Dan Boneh, Eric Chen, and Collin Jackson. Password managers: Attacks and defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 449–464, San Diego, CA, August 2014. USENIX Association.

[47] Sooel Son and Vitaly Shmatikov. The postman always rings twice: Attacking and defending postmessage in html5 websites. In *NDSS*, 2013.

[48] Marco Squarcina, Stefano Calzavara, and Matteo Maffei. The remote on the local: Exacerbating web attacks via service workers caches. In *15th IEEE Workshop on Offensive Technologies (WOOT 21)*. IEEE, 2021.

[49] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, page 921–930, New York, NY, USA, 2010. Association for Computing Machinery.

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

[51] Marius Steffens and Ben Stock. Pmforce: Systematically analyzing postmessage handlers at scale. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 493–505, New York, NY, USA, 2020. Association for Computing Machinery.

[52] Ben Stock and Martin 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, page 183–194, New York, NY, USA, 2014. Association for Computing Machinery.

[53] T. Watanabe, Eitaro Shioji, Mitsuaki Akiyama, and T. Mori. Melting pot of origins: Compromising the intermediary web services that rehost websites. In *NDSS*, 2020.

[54] Peter Wurzinger, Christian Platzer, Christian Ludl, Engin Kirda, and Christopher Kruegel. Swap: Mitigating xss attacks using a reverse proxy. In *2009 ICSE Workshop on Software Engineering for Secure Systems*, pages 33–39, 2009.