Security Study of Service Worker Cross-Site Scripting.

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

Raj Vardhan
Texas A&M University
raj_vardhan@tamu.edu

Guangliang Yang
Texas A&M University
guangliang.yang11@gmail.com

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

ABSTRACT

Nowadays, modern websites are utilizing service workers to provide users with app-like functionalities such as offline mode and push notifications. To handle such features, the service worker is equipped with special privileges including HTTP traffic manipulation. Thus, it is designed with security as a priority. However, we find that many websites introduce a questionable practice that can jeopardize the security of a service worker.

In this work, we demonstrate how this practice can result in a cross-site scripting (XSS) attack inside a service worker, allowing an attacker to obtain and leverage service worker privileges. Due to the uniqueness of these privileges, such attacks can lead to more severe consequences compared to a typical XSS attack. We term this type of vulnerability as Service Worker based Cross-Site Scripting (SW-XSS). To assess the real-world security impact, we develop a tool called SW-Scanner and use it to analyze top websites in the wild. Our findings reveal a worrisome trend. In total, we find 40 websites vulnerable to this attack including several popular and high ranking websites. Finally, we discuss potential defense solutions to mitigate the SW-XSS vulnerability.

CCS CONCEPTS

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

KEYWORDS

Service Worker, Cross-Site Scripting

ACM Reference Format:
Phakpoom Chinprutthiwong, Raj Vardhan, Guangliang Yang, and Guofei Gu. 2020. Security Study of Service Worker Cross-Site Scripting.. In *Annual Computer Security Applications Conference (ACSAC 2020), December 7–11, 2020, Austin, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3427228.3427290

ACSAC 2020, December 7–11, 2020, Austin, USA

ACM ISBN 978-1-4503-8858-0/20/12…$15.00
https://doi.org/10.1145/3427228.3427290

1 INTRODUCTION

To improve the browsing experience of web users, modern websites are utilizing service workers (SW) to enable app-like features such as offline working mode and push notifications. Such features require a service worker to run in a special execution context which is isolated from the main page. This allows a service worker to intercept and modify network traffic of the corresponding website to provide a cached HTTP response when the network is offline. Additionally, as a service worker does not require a browser's window to be open for its functionalities to execute, it can listen to and handle push messages which can arrive spontaneously.

As a service worker provides such unique functionalities and execution environment, its security is critical. Generally, web browsers enforce several rules to ensure a service worker will be safe from outside tampering. For instance, only a same-origin file is allowed to be registered as a service worker. Despite existing safeguards, we find a new XSS vulnerability that allows an external source to execute malicious code inside a service worker.

In this work, we discover a considerable number of websites introduce a questionable programming practice and break the security assumptions in favor of configurability and flexibility of their service workers. These websites 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 based 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.

With the service worker's capabilities, an attacker can gain several advantages. Because a service worker runs in the background and its lifetime lasts until a new service worker is provided or the website's data is manually cleared, the attacker can stealthily utilize a compromised service worker for an extended period of time. The attacker can also use the service worker to persistently monitor the victim's actions or inject malicious content into the web page. In some cases, an attacker only needs to send victims a URL to compromise the service worker. The compromised service worker

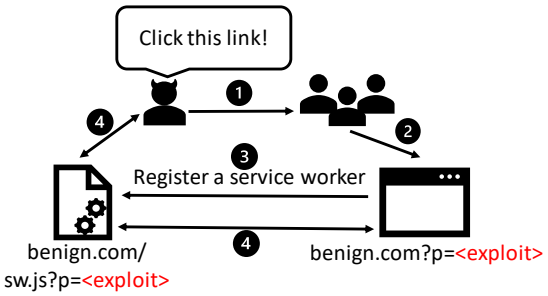

Figure 1: A motivating example demonstrating how cross-site scripting can also occur inside a service worker. A remote attacker can compromise and control a benign service worker to steal victim's sensitive data.

can then inject malicious content into the web page before forwarding sensitive information to the attacker as shown in Figure 1. We discuss this attack in more detail in Section 4.3.

To evaluate the impact of SW-XSS vulnerability on real-world websites, we develop a service worker scanning tool called SW-Scanner. We use SW-Scanner to crawl and analyze the top 100,000 websites in the wild. SW-Scanner applies taint information on URL search parameters to track how they are used inside a service worker and reports when a tainted value reaches a sensitive function. We find the SW-XSS vulnerability in 40 of these websites which includes some popular and high profile websites with more than a hundred million combined visitors per month. With growing adoption of service workers, we believe this trend will only get worse if web developers keep overlooking this problem. We hope that our work will help raise awareness regarding the importance of service worker's security and provide useful insights for web developers regarding the secure implementation of service workers in the future.

Our main contributions are as follows.

- We discover a new XSS vulnerability caused by a questionable programming practice followed by some websites to install a service worker. We analyze this class of vulnerability (Section 4) and term it as SW-XSS. To the best of our knowledge, we are the first to identify XSS in a service worker and show a practical attack that can compromise a benign service worker.
- We develop a service worker scanning tool called SW-Scanner (Section 5) to evaluate the real-world impact of the problem (Section 6) and find that 40 websites are vulnerable. This includes several popular websites that have more than a hundred million combined visitors per month. We open source the tool and collected data to help the research community of this domain[1].

2 BACKGROUND

In this section, we first provide an introduction to web workers. Next, we discuss some unique traits of service workers which makes their purpose different from other web workers. Then, we describe the steps involved in a service worker's lifecycle. Finally, we discuss the existing forms of cross-site scripting (XSS) attacks.

2.1 Web Workers

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 requires running several tasks simultaneously, such as processing a large amount of API 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.

2.2 Service Worker

A service worker is a type of web worker. Like any web worker, it runs in a background thread that is separate from the main web page. However, it contains a set of unique features that makes the purpose of service workers different from other web workers. 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.

2.3 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 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.

Install. This event only occurs once per service worker 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.

[1] https://u.tamu.edu/sw-scanner

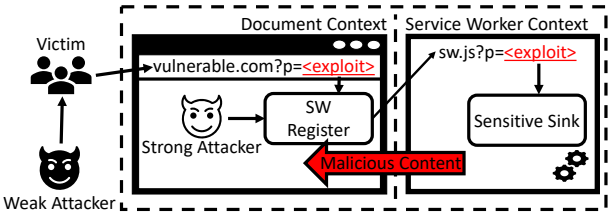

Figure 2: An illustration of SW-XSS attack threat model

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 into idle within a short period of time (usually less 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.

2.4 Cross-Site Scripting

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 get 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 in 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.

3 THREAT MODEL

Generally, web attacks consider two separate contexts of client and server. However, as shown in Figure 2, 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 client-side in a traditional web attacker's threat model, which covers the main page's execution context or the DOM. 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. In this work, we consider two types of attackers which we term as weak attackers and strong attackers.

Weak Attackers represent a threat model consistent with the existing Web attackers present 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.

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 (discussed further in Section 4.1.2). 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.[2]

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 paper.

4 SW-XSS ATTACK

In this section, we analyze the SW-XSS vulnerability. First, we discuss the motivation of an attacker to conduct a SW-XSS attack, i.e., address the question of why an attacker would target and compromise a benign service worker. For both weak and strong attackers, we examine the additional advantages or special privileges provided by a service worker and how the attackers may utilize them. Then, we discuss the challenges of compromising a service worker and examine why existing safeguards may not be adequate. Finally, we demonstrate how an attacker can compromise a benign service worker through SW-XSS vulnerability and discuss the differences of this attack compared to traditional XSS attacks.

4.1 Motivation

A service worker provides several unique functionalities that are not available in other contexts, thereby making it a new target for

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

attackers. As we assume two types of attackers, we discuss the motivation for each type of attacker as follows.

4.1.1 Weak attacker. For a weak attacker, the most prominent aspect of a service worker is that it creates a new attack vector in the form of a new sensitive function called *navigator.serviceWorker.register*. As we will later demonstrate in Section 4.3, 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 open a new attack vector for launching an XSS attack, it can also provides 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.

4.1.2 Strong attacker. 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 network traffic of the website. It can intercept network traffic of the files under its scope and modify any HTTP's 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 scripts that wrap sensitive functions like *document.write* with security checks or by an extended in-browser defense mechanism [25, 28] that limits the access of third-party origins), 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 the 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.

4.2 Challenges

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 challenges that an attacker could face while targeting a benign service worker and how the attacker may circumvent the corresponding protections. We also discuss why certain safeguards may be inadequate in preventing the attacker.

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.

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. 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 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 this attack in practice. As we will later show in Section 6.2.1, 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

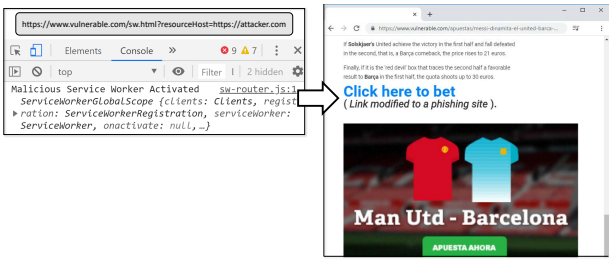

Figure 3: A screenshot of an SW-XSS attack targeting vulnerable.com, allowing attackers to steal the victim's sensitive information.

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. That is, 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, we find that most websites rarely update their service workers in practice. In Section 6.3, we measure the service worker freshness and show that service workers deployed by websites are generally stale. Such practices provide attackers an opportunity to circumvent this safeguard and compromise a benign service worker.

4.3 Hijacking Service Worker

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. [22] and Mendoza et al [19]. 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 [14, 20].

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 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 in 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 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 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 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 capabilities is the *fetch* event. A *fetch* event is generated for every resource request. The *fetch* event handler has access to the request's

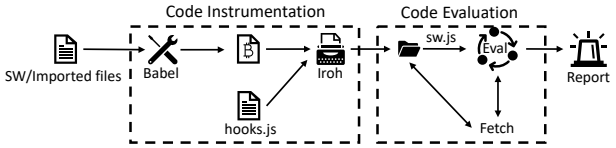

Figure 4: An illustration of SW-Scanner's pipeline.

HTTP headers, in 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, 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 regarding 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.

4.4 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 which make the SW-XSS novel. We highlight the main differences between this attack and the existing XSS as follows.

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 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.

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.

5 DETECTING SW-XSS IN THE WILD

In this section, we introduce our tool called 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. 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.

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 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 like the controller and will execute the instrumented code and ensure that the taint tracking runs and reports correctly.

5.1 Code Instrumenter Module

This module accepts a JavaScript file as an input. Then it checks the input's validity using Babel [1], 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 [2]. Iroh uses 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 in Iroh's Github website [3]. Once one of these predefined locations is reached during an 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 operand is tainted. When a tainted value reaches a sensitive sink, SW-Scanner will log the tainted value.

Table 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

5.2 Code Evaluation Module

This module is developed as a website. It accepts the instrumented files as an input and reports the taint 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 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 mismatch, 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.

6 EVALUATION

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 worker and its parameter usage in top websites. Next, we uncover the SW-XSS vulnerabilities in the wild, present the results of SW-Scanner, and discuss the responsible disclosure we made 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.

6.1 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.

6.2 SW-XSS Vulnerabilities in the Wild

For the 2,525 websites with parameter usage in 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 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's keys or 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 type 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 URL 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 a 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

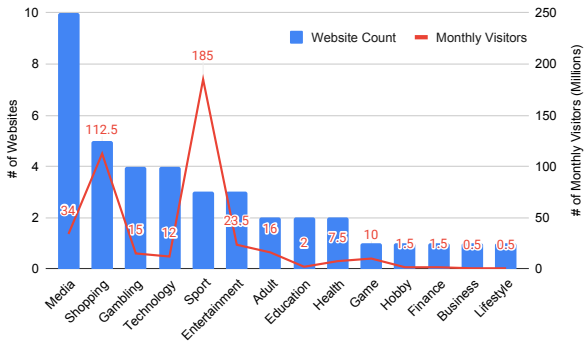

Figure 5: A chart representing the number of vulnerable websites by category, and showing their aggregated monthly visitors.

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 6.4.

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 in 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 [4] 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 5 summarizes the number of all 40 vulnerable websites and their monthly visits based on the category of websites. The Media category 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.

6.2.1 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 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 includes 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 proceed 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 on 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 true today, it is possible that service workers will evolve in 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.

6.2.2 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 in this work.

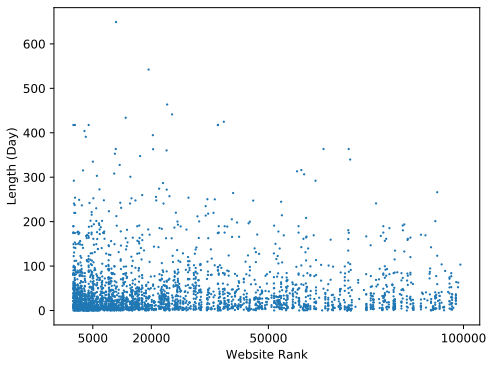

Figure 6: A scatter plot illustrating the length between updates of service worker files based on website ranking

6.3 Service Worker Freshness

In Section 4.1.2, we claimed that a temporary strong attacker can benefit from the persistency 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.

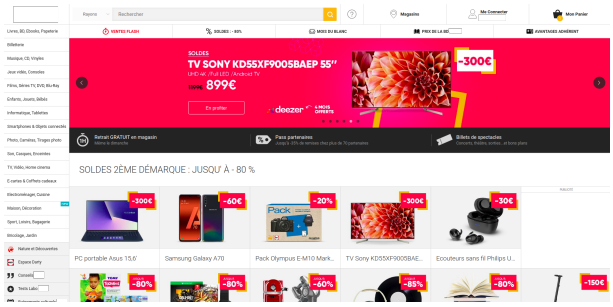

Figure 7: A screenshot of a vulnerable shopping website

We use the Internet Archive's Wayback Machine to retrieve the old service worker files [5]. 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 files as illustrated in Figure 6. 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 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 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 visit 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 [6] 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.

6.4 Case Study

We discuss a case study of another high-profile shopping website (Figure 7) 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 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 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 use it 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 line (6-9) the code is extracted from the parameters and returned at line 11, which later gets executed at 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 in high-profile websites and can occur in a complicated manner making it hard to be detected. Therefore, such problem may be overlooked by web developers. We hope that our work will help raise awareness regarding the importance of service worker's security and provide useful insights for web developers to implement secure service workers in the future.

7 POTENTIAL DEFENSE SOLUTIONS

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 because 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 URL search parameters of a service worker, to provide another way for the document context or web server to communicate with the service worker during the installation, or to limit script inclusion in the SW context.

To 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 [7, 8], the path does not include the parameters. Therefore, this CSP directive is currently not effective (unless a new specification includes URL search parameters for source lists). In any case, we notice that the Manifest used to have the *service-worker* property that can tell the browser which service worker the developers intend to install. Although this property has become obsolete [9], 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 needs to be varied for each visitor. This leads to our second suggestion that is to use cookie, which can provide more dynamic values.

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. However, an attacker in the document context could still launch SW-XSS attack by manipulating a service worker's cookie. Therefore, we suggest that 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 an additional overhead.

Another feasible defense solution 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 [24] and can be hard to configure correctly or can be bypassed [27]. 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.

Lastly, we suggest a mitigation approach in addition to other previously discussed solutions that could be helpful in the long term. We notice that while the service worker gives better experience for users, it also gives attackers a new attack surface and additional privileges. For example, web attacks used to happen when a victim opens a malicious or compromised web page, but now service workers can execute malicious payload off-screen and enable several novel attacks [21, 26]. By simply visiting a website, users are exposed to potential risks of a service worker. Therefore, we suggest that web browsers could provide an indicator when a website has a service worker installed (possibly similar to the

lock icon for HTTPS websites). This could help users be aware of the risk when visiting an untrusted website and prompt them to clear the website's content or remove unnecessary service workers more often. While this approach may not yield any result at this moment, with increasing adoption of service worker, this approach may prove to be useful. In any case, such an approach will need a user study in the future to fully understand its effectiveness.

8 RELATED WORK

Web Attack. Generally, web attacks can be categorized into either client- or server-side. Saxena et al. and Mendoza et al. show that on the server-side, a bad or malicious parameter controlled by the attackers can potentially compromise users' sensitive data [19, 22]. Our work, on the other hand, shares similarities in terms of how the attackers can craft a malicious parameter to subvert the security. However, SW-XSS does not involve the server-side and occurs in the client-side instead.

Cross-Site Scripting attacks are one of the most infamous client-side attacks. Stock et al. study the history of XSS attacks over a decade and find that script inclusion or data access from cross-domain plays a role in the website's security, which is also in line with Nikiforakis et al. findings [20, 24]. Our attack also utilizes cross-domain file inclusion to launch the SW-XSS attack, thus we share the same sentiment regarding this issue. In recent years, a variant of XSS called DOM-XSS is emerging [17, 18, 23]. DOM-XSS can be similar to our attack in a sense that it allows attackers to execute remote code on the client-side. However, SW-XSS does not execute the payload in the DOM but in the service worker unlike DOM-XSS.

Service worker security is rarely studied in the past but is attracting more attention. Lee et al. are possibly the first to discuss attacks related to Progressive Web App and service worker [16]. However, they assume that the vulnerable website runs in HTTP while our threat model assumes full HTTPS. Papadopoulos et al. also analyze the impact of when a service worker runs a malicious code in which the attackers can mine crypto-currency in the background or control a botnet inside the victim's browser [21]. Nevertheless, Papadopoulos et al. assume that the target website and the service worker are already malicious or compromised but does not discuss a way to compromise a service worker. Watanabe et al. discuss how an attacker can register a malicious service worker for a re-hosted website to compromise other re-hosted websites of the same service provider [26]. We look at the service worker in a different angle and assume the service worker is benign while the goal is to compromise it instead of registering a malicious service worker. Stuart Larsen discovers a bug allowing a vulnerable JSONP endpoint to be used to register arbitrary code for a service worker [10]. Our work shows an alternative way to compromise a benign service worker through URL search parameters of a service worker.

JavaScript Analysis. Static analysis tools such as JSHint or SonarJS can help identify generic coding issues [11, 12], but JavaScript is an extremely dynamic language, so the report generated by static analysis will contain a lot of false negative or false positive, and they cannot detect sophisticated attacks such as XSS. Therefore, most recent studies focus on utilizing dynamic analysis. Saxena et al., Melicher et al., and Lekies et al. propose dynamic analysis tools based on browser or JavaScript engine modification [17, 18, 22]. However, service worker development is still in an early stage and its specification changes frequently. Tools that are based on browser modification cannot naturally keep up with the changes, and they do not take the service worker context into account. While Jueckstock et al. concurrently propose a light-weight in-browser dynamic analysis tool that can monitor JavaScript's native APIs usage and quickly adapt into a new browser version, it cannot currently perform taint tracking [13]. Therefore, we implement SW-Scanner in JavaScript, which provides taint tracking capability and can run in any browser.

9 CONCLUSION

In this work, we found a growing problematic practice in SW-enabled websites. These websites use URL search parameters during their service worker's installation and blindly trust those parameters. This allows attackers to feed a malicious parameter into a benign service worker to compromise it. We termed this attack as SW-XSS. We developed a tool called SW-Scanner to evaluate the impact of SW-XSS in real-world websites. Our findings showed 40 websites to be vulnerable, wherein more than a hundred million users could potentially be affected per month. We reported our findings to all affected developers. With growing adoption and forthcoming additional features of service workers, more vulnerabilities or new types of attacks may emerge if web developers neglect this problem. We hope that this work will provide useful insights that can help minimize such outcomes in future.

ACKNOWLEDGMENTS

This material is based upon work supported by the NSF/VMware Partnership on Software Defined Infrastructure as a Foundation for Clean-Slate Computing Security (SDI-CSCS) program under Award Title "S2OS: Enabling Infrastructure-Wide Programmable Security with SDI" and No. 1700544. It is also supported in part by 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, VMware and ONR.

REFERENCES

[1] [n.d.]. https://babeljs.io/.
[2] [n.d.]. https://maierfelix.github.io/Iroh/.
[3] [n.d.]. https://github.com/maierfelix/Iroh/blob/master/API.md.
[4] [n.d.]. https://www.similarweb.com/.
[5] [n.d.]. https://web.archive.org/.
[6] [n.d.]. https://www.openbugbounty.org/.
[7] [n.d.]. https://www.w3.org/TR/CSP3/#framework-directive-source-list.
[8] [n.d.]. https://tools.ietf.org/html/rfc3986#section-3.3.
[9] [n.d.]. https://developer.mozilla.org/en-US/docs/Web/Manifest/serviceworker.
[10] [n.d.]. https://c0nradsc0rner.com/2016/06/17/xss-persistence-using-jsonp-and-serviceworkers/.
[11] [n.d.]. https://jshint.com/.
[12] [n.d.]. https://github.com/SonarSource/SonarJS.
[13] Jordan Jueckstock and Alexandros Kapravelos. 2019. VisibleV8: In-browser Monitoring of JavaScript in the Wild. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.
[14] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2017. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society.

[15] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS 2019)*. https://doi.org/10.14722/ndss.2019.23386

[16] Jiyeon Lee, Hayeon Kim, Junghwan Park, Insik Shin, and Sooel Son. 2018. Pride and Prejudice in Progressive Web Apps: Abusing Native App-like Features in Web Applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 1731–1746. https://doi.org/10.1145/3243734.3243867

[17] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 million flows later: large-scale detection of DOM-based XSS. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM, 1193–1204. https://doi.org/10.1145/2508859.2516703

[18] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. 2018. Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_07A-4_Melicher_paper.pdf

[19] Abner Mendoza and Guofei Gu. 2018. 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*. IEEE, 756–769. https://doi.org/10.1109/SP.2018.00039

[20] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. You are what you include: large-scale evaluation of remote javascript inclusions. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, Ting Yu, George Danezis, and Virgil D. Gligor (Eds.). ACM, 736–747. https://doi.org/10.1145/2382196.2382274

[21] Panagiotis Papadopoulos, Panagiotis Ilia, Michalis Polychronakis, Evangelos P. Markatos, Sotiris Ioannidis, and Giorgos Vasiliadis. 2019. Master of Web Puppets: Abusing Web Browsers for Persistent and Stealthy Computation. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/master-of-web-puppets-abusing-web-browsers-for-persistent-and-stealthy-computation/

[22] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. 2010. 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. https://www.ndss-symposium.org/ndss2010/flax-systematic-discovery-client-side-validation-vulnerabilities-rich-web-applications

[23] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. 2019. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/dont-trust-the-locals-investigating-the-prevalence-of-persistent-client-side-cross-site-scripting-in-the-wild/

[24] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. 2017. How the Web Tangled Itself: Uncovering the History of Client-Side Web (In)Security. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 971–987. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/stock

[25] Tung Tran, Riccardo Pelizzi, and R. Sekar. 2015. JaTE: Transparent and Efficient JavaScript Confinement. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC 2015)*. ACM, New York, NY, USA, 151–160. https://doi.org/10.1145/2818000.2818019

[26] Takuya Watanabe, Eitaro Shioji, Mitsuaki Akiyama, and Tatsuya Mori. 2020. Melting Pot of Origins: Compromising the Intermediary Web Services that Rehost Websites.

[27] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. 2016. 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.

[28] Y. Zhou and D. Evans. 2015. Understanding and Monitoring Embedded Web Scripts. In *Proc. IEEE Symp. Security and Privacy*. 850–865. https://doi.org/10.1109/SP.2015.57