Developing an isolated in-browser platform for security applications against malicious browser extensions

Karampelas Apostolos-Paraschos

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: *Ioannidis Sotiris*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

University of Crete
Computer Science Department

Developing an isolated in-browser platform for security applications against malicious browser extensions

Thesis submitted by
Karampelas Apostolos-Paraschos
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: APOSTOLOS KARAMPELAS Digitally signed by APOSTOLOS KARAMPELAS Date: 2021.08.02 16:26:28 +03'00'

Karampelas Apostolos-Paraschos

Committee approvals: Polyvios Pratikakis Digitally signed by Polyvios Pratikakis Date: 2021.08.17 12:21:15 +03'00'

Pratikakis Polyvios
Assistant Professor, Thesis Supervisor

SOTIRIOS IOANNIDIS Digitally signed by SOTIRIOS IOANNIDIS Date: 2021.07.30 18:43:40 +03'00'

Ioannidis Sotiris
Associate Professor Technical University of Crete, Committee Member

KONSTANTINOS MAGOUTIS Digitally signed by KONSTANTINOS MAGOUTIS Date: 2021.08.02 00:56:30 +03'00'

Magoutis Kostas
Associate Professor, Committee Member

Departmental approval: Polyvios Pratikakis Digitally signed by Polyvios Pratikakis Date: 2021.08.17 12:21:38 +03'00'

Pratikakis Polyvios
Assistant Professor, Director of Graduate Studies

Heraklion, July 2021

Developing an isolated in-browser platform for security applications against malicious browser extensions

Abstract

Modern web browsers offer developers a wide variety of powerful features, enabling them to push web application logic to the user side increasingly. This paradigm shift aims to improve end-user quality of experience by minimizing the latency and increasing the scalability of web services.

At the core of these features lie browser extensions, which have access to a rich set of tools so that they can satisfy unique user needs, like customizing the user interface or blocking ads. Extensions have also seen wide adoption in the industry, becoming a very popular avenue for companies in the web ecosystem to deploy and maintain the client side logic of their services. Unfortunately, malicious actors often exploit extensions to launch Man-in-the-Browser attacks, where they serve as a vehicle for spying, phishing and fraud at the expense of unknowing users. In some cases, compromising a privileged user opens up a more potent attack vector against the web service or its broad userbase.

Motivated by the lack of effective countermeasures by major browser vendors, this thesis proposes WRIT, a practical framework that enables websites and web service providers to protect critical functionality from malicious extension abuse. WRIT's primary objective is to establish and maintain a trusted execution environment isolated both from conventional client-sided code and extensions, where security sensitive code can be deployed and run safely. WRIT then provides the necessary tools to attest the integrity of outgoing web requests and verify their authenticity, ensuring they were triggered by a user's action and not by a malicious extension.

We evaluate WRIT's security properties by analyzing the possible attacks extensions can launch against a web service's client-sided code and WRIT itself. Each attack scenario is executed and tested against WRIT in practice through an individual custom extension. We also conduct a performance evaluation testing WRIT's prototype implementation under varying network conditions. Our experimental results show that it adds a negligible 7.29 ms latency to sensitive actions triggered by users, such as posting a message on social media.

Contents

List of Figures

1. Introduction

Browser extensions enable developers to augment the baseline functionality provided by browsers and enhance the user experience. Yet, the very same rich capabilities of theirs can provide a powerful vehicle for attackers to perform malicious actions [29, 8, 55]. There are various cases discovered over the years, where malicious browser extensions were used by attackers to infect users' browsers in order to gain control of the code that web servers sent to the users during browsing [13, 41, 32, 61].

Focusing on recently reported attacks though, we can see a new trend in their functionality, in which the malicious extensions do not simply aim to steal user credentials or passwords, but instead mask their malicious actions under the guise of typical user activity [1, 3, 5]. A recent example involved a cluster of scam extensions in Google's Chrome Web Store with combined installations of more than 500,000, which were able to generate fake ad clicks in an automated way, without user knowledge or consent [3]; the malicious extension works by creating the same web requests that would have been created if the advertisement was clicked by a real user. A similar but more advanced and targeted attack scenario was reported in early 2021, where a malicious extension was able to perform several actions on the Gmail accounts of its victims (including reading, forwarding, deleting, and sending e-mails) [5]. The implementation of such sophisticated types of malicious extensions makes it very challenging for web services to distinguish the legitimate actions of real users from malicious actions performed by extensions within the users' browser.

Before addressing these as well as many other threats posed by browser extensions, there is a *fundamental necessity for an isolated environment* that will allow a webpage to run its script(s) without the possibility of data access or code injection from any third-party code. Otherwise, any and all defensive code deployed in the client side by the web service is susceptible to intereference by resident malicious extensions and third party scripts.

Motivated by the lack of any existing effective solutions, in this work we present a novel and practical framework named WRIT (Web Request Integrity and aTtestation). WRIT's purpose is to aid website and web service owners overcome two key challenges:

1. running security-sensitive code without hindrance in the untrusted and potentially infected browser of clients

2. ensuring that critical requests have been created via a benign control-flow path (i.e., request triggered by actual user) and not generated artificially by malicious third-party/browser extension code.

In order to tackle these challenges, WRIT provides the necessary front-end security building blocks to (i) establish an isolated execution environment to protect sensitive data and/or code and (ii) verify the execution integrity of selective code snippets to identify artificial requests. WRIT is implemented completely in JavaScript and utilizes APIs built into all modern browsers exclusively, thus not requiring *any* modification of the user's browser. To assess the effectiveness and feasibility of our approach, we implemented a prototype of WRIT (available in [2]) and evaluated its performance. The evaluation's results show our approach incurs a low overhead in terms of latency and throughput.

Contributions: In summary, the contributions of this thesis are:

1. We design a front-end mechanism that generates and maintains a protected environment against untrusted third-party code and/or browser extensions. We also design an attestation technique for JavaScript execution that verifies the integrity of outgoing web requests at function-level granularity.

2. We implement and provide an open source prototype of our approach. Further, we integrate our approach within Axios [9], a lightweight HTTP client API for creating web requests that is typically used in combination with many popular modern web frameworks, such as ReactJS [31] and Angular [28].

3. We conduct a thorough analysis to evaluate the security properties and performance overheads of our approach. Preliminary performance evaluation results show that the latency added by WRIT to protect user actions considered as security-critical[1] (i.e., post a message) is practically negligible to the user experience (7.29 ms).

[1]WRIT operates only on marked-as-sensitive web requests, leaving the rest (such as those for getting normal web content, object fetches, and asynchronous updates) unaffected.

2. Threat model

We assume an adversary who manages to run malicious JavaScript code at the client side either by a malicious imported third-party library or by a malicious extension installed in the victim's browser, enabling them to hijack the browsing sessions of the victim to perform specific actions (e.g., web requests or transactions) on their behalf. We also assume that neither the browser nor the operating system have been modified (e.g., by installing malicious software on the underlying system that is able to extend the capabilities of the attacker beyond the browser context[1]) and thus, the capabilities of the attacker are limited to the context of the browser.

Recently reported attacks can be categorized in two popular attack vectors: (i) *click frauds*, in which malicious extensions cover their click traffic under the guise of ordinary user activity, in order to look as benign as possible [3]; and (ii) *account hijacking*, in which the extensions perform several actions while users are already signed in to their accounts (e.g., message posts, reading and sending of e-mails, etc), as it was in the case for Facebook [1] and Gmail [5]. We note a common pattern in both vectors: extensions do not only have access to powerful (and dangerous) built-in APIs, but also actively seek to benefit from their position in the user's browser, where they are already authenticated and traffic is decrypted. This makes it very difficult for web services to distinguish the actions of a real user signed in to their personal account from the actions performed by an extension within the user's browser.

Infection. A successful infection can be achieved in many different ways: (i) by deceiving the user that the extension is harmless (typically providing a useful feature at the surface), (ii) by attackers purchasing popular benign extensions and then updating them with malicious operations, (iii) by side-loading from a local archive and not the official extension store of a browser vendor, or (iv) by compromising popular extensions or JavaScript libraries and subsequently having them serve malicious code [15, 46, 51, 14].

Capabilities. A malicious extension can monitor, disrupt, tamper, or block *any* incoming or outgoing traffic from and to the web service, as well as inject JavaScript code to the web page or tamper *any* JavaScript snippet sent by a web

[1] An adversary capable of compromising a user's browser or operating system can launch significantly more potent attacks than running malicious JavaScript code.

server before running on the user's browser. A browser extension can execute code —without any further interaction with the user past initial installation— through background scripts or content scripts. A background script [17] runs continuously in the browser's background (as the name implies), as long as the browser is running. In this global context, a malicious extension can perform general purpose attacks (e.g., monitor all outgoing HTTP requests), as well as have access to a plethora of information regarding every open tab and leverage it to perform more targeted attacks. In addition, a malicious extension can leverage the browser API for web extensions, i.e., `executeScript` [21], in order to deploy code segments that can run as content scripts directly into one or more open tabs. Alternatively, content scripts can be declared into the extension's manifest to run in tabs matching a URL pattern (a wildcard enables injecting to all tabs) at page load by default or after the page finishes loading; either way, a malicious extension gains the ability to interact with a specific tab's page's context through content scripts. Threat actors can use a malicious extension to send a web request from a user's browser using one of the following methods:

1. By synthetically crafting a web request and manually sending it to the server. The request may include any cookies or session identifiers needed, which can easily be acquired by a malicious extension.

2. By hooking a pre-existing JavaScript function that resides either within a page's DOM[2] or is browser built-in, in which the malicious code is injected. This way, the malicious code will be executed every time the hooked function is invoked.

3. By mimicking a user action that ends up sending a seemingly benign request. For example, auto-completing a form and generating an artificial click event on the button that submits the form, instead of using a programmatic function to submit it directly, i.e., `form.submit()`. The resulting request will be sent through an ordinary program flow, even though it has been created artificially via malicious code clicking a button.

Last but not least, it is important to note that WRIT does not aim to defend against cases where the browser extension tricks the user into performing an action through clickjacking or UI redressing attacks. Instead, WRIT aims to assess the humanness of an action regardless if this was intended or not. There are many works already dealing with such kind of user action-jacking attacks [39, 6, 49].

[2]The Document Object Model (DOM) is the data representation of the objects that comprise the structure and content of a document on the web.

3. Building blocks

In this section, we describe the tools that we mainly use to build WRIT. By design, WRIT restricts itself to built-in browser features and APIs as well as standard JavaScript features to ensure wide compatibility with modern browsers.

3.1 Service Workers

Service Workers [24] are non-blocking (i.e., fully asynchronous) modules that reside in the user's browser, network-wise positioned between a loaded web page and its remote server. They are designed to intercept and handle network requests originating in their registered web page(s) as programmable network proxies and caches, giving developers the necessary tools to make their website available under poor network conditions or even offline. Other popular use cases include pre-caching for performance, background syncing and pushing notifications.

Registration. Page scripts can register a new Service Worker using the `serviceWorkerContainer.register()` function. Its main argument requires the URL of the Service Worker's script, which is passed internally to the browser and then fetched over HTTPS. As a result, no browser extension or any in-browser entity can have access to the browser's C++ implementation that handles the Service Worker's retrieval and registration with the first-party domain [50]. Moreover, this JavaScript file can only be fetched from the first-party domain (i.e., it cannot be hosted in a CDN or any other third-party server) and cannot be registered from an iframe or third-party script.

Scope. Unless explicitly specified as an optional argument during registration, the Service Worker's scope begins at the current location (i.e., currently visited URL) and includes all possible sub-paths. The registered Service Worker will then have access to requests originating only in the set of paths ultimately defined by the scope. Therefore, it is possible to have multiple Service Workers in operation at the same time, as long as they are are registered with different scopes; when scopes conflict, the Service Worker with the longest matching scope prevails (i.e., handles the request). In practice WRIT benefits from this design decision, as website owners aren't outright forced to tweak or replace their existing Service Worker scripts and can instead opt to utilize WRIT's reference script in a different scope.

Lifecycle. A Service Worker is typically registered the first time the user visits a website and runs in the background, in an isolated execution context that is independent from its registered web page. While Service Workers cannot access the page's DOM directly, they can communicate with pages under their scope directly via the `postMessage` interface [18] or indirectly via custom responses to intercepted requests. After a period of inactivity or when the user browses away from a website, its Service Worker is paused by the browser and its associated thread is terminated; it is then reactivated and its thread is spun back up once the registered domain is visited again.

3.2 JavaScript Closures

Functions in JavaScript can form closures[1] [19], which combine functions with the lexical environment they are created in; the lexical environment includes any variables or other functions that share the same scope with the function forming the closure. The closure allows them to live on (i.e., to remain operational and accessible) past their original scope through the function that formed the closure. This property makes closures a powerful tool that allows us to emulate private methods, in order to regulate access to sensitive library script functions and variables.

JavaScript closures make it feasible to generate a function that contains one or more private variables that store secret or sensitive data (e.g., secret tokens, private keys, etc.). At execution time, private variables are kept hidden and cannot be accessed by any other JavaScript scope, except for the single public function we make available which cannot be used to reveal private variables. Crucially, by deploying WRIT's library code as a closure within the visited page's execution context guarantees that malicious extensions cannot access WRIT's sensitive private variables and functions. We also note that while a malicious extension could override native JavaScript APIs in order to extract secrets from a closure, this tampering can be easily detected and then restored via an iframe [4].

3.3 Our approach: WRIT

WRIT utilizes both JavaScript closures and Service Workers in conjunction to form a composite, safe environment that can verify if a request was created through a benign execution path and not by any form of automation, like originating from a malicious browser extension infecting the user's browser.

Additionally, our approach leverages request/response flows for the internal communication between a JavaScript closure in the visited page and its respective Service Worker. The behavior of these flows (explained in a Chromium bug

[1] In layman's terms, a JavaScript closure can be described as a code block wrapped in an anonymous function that executes itself immediately upon definition, thus locking itself in a scope that is only accessible by a variable or function that it provides. An example is shown in Listing 4.2.

report [12]) has been standardized across all modern Chromium-based browsers (e.g., Chrome, Opera, Edge, Brave, etc.). Figure 3.1 showcases request/response flow behavior with both a Service Worker and an extension operating in the visited page's context. Browsers already do not allow extensions to use or interact with Service Workers (e.g., monitor their execution, inspect their variables, etc.), except for detecting their registration. Recent browser versions restrict this further by not allowing extensions to communicate with Service Workers or monitor their messages with the webpage, i.e., requests originated from extensions' background or content scripts do not pass through Service Workers.

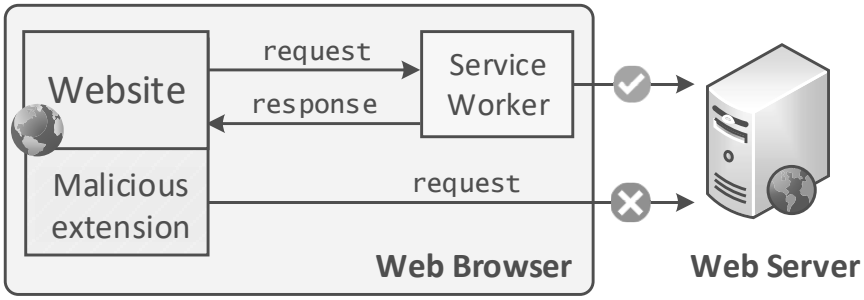

Figure 3.1: The Service Worker interacts with the website, the web server and possible browser extensions. The inability of browser extensions to communicate with the Service Worker or monitor their messages with the website allows WRIT to detect requests that have been issued by a browser extension.

4. System Overview

WRIT is comprised of three parts, as shown in Figure 4.1. The first part is the JavaScript code that resides within the web page that the server wants to protect; the second is the Service Worker residing in the user's browser and the third is a server-side component, whose primary role is to verify that any received request is properly signed. The in-page script[1] is the essential link that allows interaction with the page's context (i.e., JavaScript and DOM), since Service Workers cannot access the content of web pages under their registered domain by design.

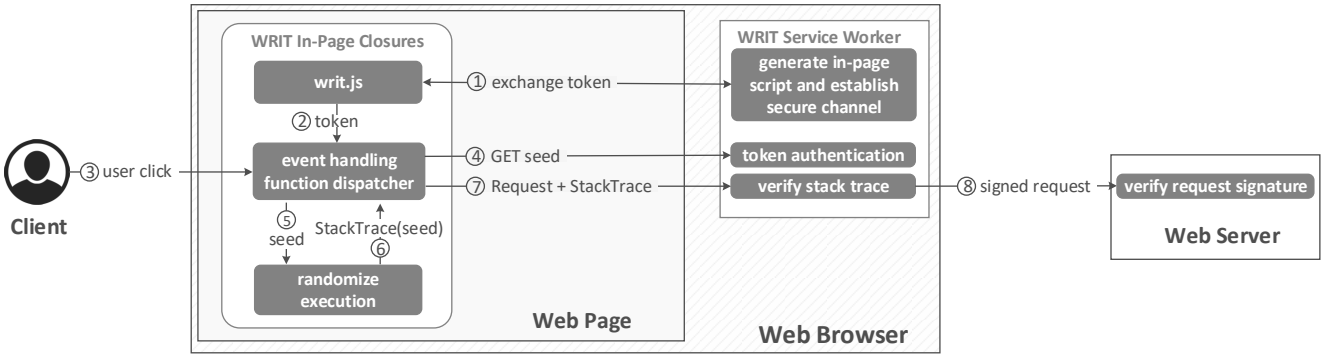

Figure 4.1: High-level overview of our approach. WRIT generates a separate in-page script at each user session, which contains a unique, hard-coded identifier. This identifier is used to establish a communication channel, through which the exchanged messages can be securely authenticated. Every time the user performs an action, WRIT's in-page component uses a secret token to request a random seed from the Service Worker. The seed is used to randomize the execution of the event handling function trace, which can be verified by the Service Worker using the exact same seed; if the verification succeeds, the Service Worker signs the corresponding request using a pre-established key and forwards it to the back end web server.

[1] In the context of the visited page's JavaScript execution environment, the term "in-page script" refers to WRIT's library script i.e., `writ.js`. The term "in-page component" refers to WRIT's closure containing the library script.

4.1 Isolated environment

WRIT utilizes Service Workers in order to guarantee a persistent environment, completely isolated and protected against any possible malicious extensions that may be running on the user's browser.

Setup phase: The setup phase occurs *only once*, typically when the user signs in for the first time. For the registration of the Service Worker, we require a setup from a clean environment. The simplest way to achieve this is by prompting the user to manually disable all extensions during the installation of the Service Worker script. Alternatively, the registration of the Service Worker can be completed even in the presence of malicious extensions, by using a continuous code update technique such as those presented in [11, 16]. After the registration, the Service Worker lives in the browser's background context and is activated automatically every time the user visits the website.

As described in Section 3.1, the Service Worker is registered through the `serviceWorkerContainer.register()` function, which requires the URL of the Service Worker's script as input. I's worth noting that even though the corresponding JavaScript file can be fetched from the first-party domain only, its filename can be arbitrary. That aspect allows us to utilize a unique one-time URL per user in order to fetch the Service Worker script from the server. The one-time URL consists of a unique *id* that the server provides to the client. This unique *id* should be appended as a string at the end of the registration URL every time a new client requests the script of the Service Worker; the server will not respond to Service Worker requests if they do not contain a valid *id*. Likewise, the server will respond to Service Worker requests *only once*: the first time such a request is made. If the server receives a request with a given *id* for a second time, it will trigger an alert for an abnormal situation (i.e., a malicious extension installed in the user's browser tried to access it or has already accessed it before). In such cases, the user needs to re-run the setup phase from the beginning. The Service Worker uses this *id* to exchange a key k with the web server, that will be used for the signing of each request. The server associates each issued key k to the respective client that requested it and binds it internally with their corresponding session cookie. As a result, even in the case where a malicious party has managed to acquire the unique key of a real user, the incident will be detected by the server.

Attestation phase: After the Service Worker has been successfully installed on the client's browser, a safe working environment has been established. The Service Worker is running in the background and monitors every web request that is exchanged between the webpage and the server. As each outgoing request arrives at the Service Worker, it undergoes validation checks; if deemed valid, the Service Worker signs it using the unique key k. In particular, the Service Worker employs the SubtleCrypto [25] API for the signing process using one of the available options for digital signature production, e.g., HMAC with SHA512 (configurable). When the server component receives one of these requests, it can verify its integrity using its signature. The requests should also include a counter in order to defend against

replay attacks, maintained on a per-user basis so that the same count never repeats twice. If the signature generated by the server differs from the one in the received request or if there is a mismatch in the expected counter value, then the server can safely assume a malicious extension has tampered with the Service Worker's request.

Our experiments indicate that even though a malicious web extension cannot access either the context of the Service Worker or its key k that is required for the signing process of the outgoing requests, it can still unregister the Service Worker. First, it must get a hold of the installed Service Worker through the API-provided `getRegistration` function; then it can call the `unregister` function using the previously obtained registration object as the sole parameter. Even in that case though, any subsequent web request received by the server will not be signed by the key k, which can in turn trigger an alert at WRIT's server-side component.
Updating the Service Worker: Typically the web browser will occasionally check for Service Worker updates in certain circumstances [26] and update the Service Worker's script automatically in case a new version is available. In order to prevent this default behavior, which could endanger new Service Worker script versions to interception by malicious extensions, the server component never updates it directly (e.g., responds with a 404). Instead, updated Service Worker code segments can be encrypted by the server using the unique key k that is stored within the Service Worker script; the latter can then use k to decrypt the update and use it as necessary.

4.2 In-Page Access

Given that the Service Worker can't access the DOM of the webpage or its JavaScript context (functions or variables), it is necessary to also have a JavaScript component running in the page. The primary role of this in-page counterpart is to act as WRIT's agent within the page's scope, interacting with the page's HTML elements, JavaScript code and communicating important information to/from the Service Worker. We present its design and responsibilities in this section and further detail its integrity preserving mechanism in Section 4.3.

The in-page component's major design challenge is how to safely initialize and execute it within the page scope, given the staggering power that browser extensions have over the page's DOM (potential threats detailed in Section 2). Obviously it is not sufficient to request the in-page WRIT component from the web server, because a malicious extension could easily tamper its code e.g., using the webRequest API [27]. For instance, the Firefox browser provides the `filterResponseData()` function[2], which allows any extension to monitor and modify the body of a HTTP response before the page's DOM tree is built. To protect against such cases we follow a different approach, in which we utilize the Service Worker to serve WRIT's

[2]Apart from Mozilla Firefox, the other contemporary browsers have removed this feature, mainly for performance reasons.

JavaScript code that resides in the webpage. The communication between the web page and the Service Worker (both ways) cannot be accessed by browser extensions employing webRequest monitoring [27], as such a malicious extension is not able to communicate with Service Workers or monitor their exchanged traffic. In other words, none of WRIT's in-page scripts can be requested, accessed or modified by a malicious extension as long as they are served by the Service Worker. The Service Worker is responsible for periodically synchronizing with the web server and acquiring any possible software updates in order to always have the last version of WRIT. These updates can be performed securely, using the key that they have exchanged at the bootstrap phase (Section 4.1).

At the beginning of every session, WRIT's in-page JavaScript component is requested and installed immediately as shown in Listing 4.2. The request is handled by the Service Worker instead of the web server. The in-page script deploys a closure that is crucial in safeguarding any sensitive or secret data (i.e., WRIT's library), as well as a unique hard-coded identifier used to provide distinguishability between WRIT's in-page counterpart and the Service Worker. This identifier is used to exchange a secret *token* generated by the Service Worker that provides mitigation against impersonation attacks, where a malicious extension crafts and sends messages to the Service Worker through a page-side script, in an attempt to imitate the page component and/or probe the Service Worker for information. This type of impersonation attack leverages the extension's ability to inject code within the page and generate requests, which blend in with the page's ordinary requests and become indistinguishable to a Service Worker that intercepts them. Since the extensions' probing attempts will not have the correct *token* however, they will fail to authenticate with the Service Worker.

```
1  (async () => {
2     // the WRIT library is fetched as a string from
3     // the Service Worker and NOT from the web server
4     let lib_string = await fetch("/writ_lib.js");
5     let writ, ua = navigator.userAgentString;
6     // Function() not fully supported in Safari; use
7     // eval() instead
8     if (ua.indexOf("Safari") != -1 &&
9             ua.indexOf("Chrome") == -1)
10      writ = eval(lib_string);
11    else
12      writ = (new Function("return " + lib_string))();
13    // reveal public library function(s) to the page
14    window.WRIT = writ;
15 })();
```

Listing 4.2: Initialization of WRIT's in-page component within a closure. The closure ensures that any sensitive data is kept safe against a malicious browser extension or third-party library that resides in the page's scope.

The *token* is stored within the secure confines of WRIT's closure. As described in Section 3.2, the closures allow us to regulate access to sensitive functions and variables. WRIT exposes a single public function to the page's general JavaScript

context (i.e., `window`, out of the closure) for other scripts to use, namely `post()`, whose sole argument is a user-defined callback function to be executed internally. The callback function should contain code that performs the web service's security or privacy sensitive actions (e.g., process user data, transactions), which eventually leads to creating a web request and sending it to the server. We detail the design and operation of `post()` in Section 4.3. As shown in Listing 4.3, this public function thinly wraps around inner/private function invocations to provide other page scripts access to WRIT's core functionality, without leaking any information about their inner workings. The private scope prevents sensitive data from being accessed by a malicious extension and contains the code for mutually exchanging the secret *token* with the Service Worker. This *token* will be used to attest the execution of the user-defined callback function, as described in Section 4.3.

```
async function post(callback, args, e, funcs=10) {
  // check if the event was triggered manually
  if (e && e.isTrusted === false)
    return new Error("Artificial event fired!");
  // get a new seed from the SW
  let seed = await fetch("/seed",
          {body: funcs, method: "POST"});
  // run the page's protected function (callback)
  // save the request it produces & capture stack trace
  // add the randomly generated functions to the trace
  let package = gen_trace(seed, callback, args, funcs);
  // forward final trace & page's request to the SW
  return fetch("/trace",
          {body: JSON.stringify(package), method: "POST"});
}
```

Listing 4.3: WRIT's dispatch function for sending requests.

In addition, we follow a serve-once policy for the in-page script and also make sure that it is the first JavaScript script that will be requested and executed at page load. This ensures that only the page will execute the in-page script and not an untrusted third-party entity within the page or a malicious browser extension; a malicious extension that hijacks the page loading process and tries to request the in-page script will fail, as the in-page script will have already been requested from the Service Worker. Indeed, an extension's content script can actually execute before the in-page script when run at `document_start`, because at that point in time the construction of the page's DOM tree has only just began. Even then, that is if the page rendering process is abruptly halted by a malicious extension, the corresponding request for the in-page script has already been sent and received.

The previously described procedure enables WRIT to achieve its primary objective in a way that is easily incorporated into any piece of existing client-sided code, while remaining as private and robust as possible in the page context wherein malicious extensions may operate. In practice, this means that an attacker cannot access or change any internal (i.e., in closure) WRIT library function or variable that stores critical system information. It could be the case though that a malicious extension overwrites (hooks) the library's public functions, with new versions that

execute malicious code before, after or instead of executing the original function. WRIT's library within the closure can repel this attack by verifying the integrity of its public functions upon use, restoring them to their original state if necessary and optionally raising an alarm notifying the user and/or the server for malicious activity [43, 4]. Finally, the majority of the WRIT's JavaScript code is structured around and makes heavy use of the Promise API [23], a hard requirement for many of the Service Worker related functions (e.g., fetch) and also crucial in ensuring that the system performs efficiently without blocking function calls.

4.3 Execution Sequence of Function Calls

Having secured WRIT's foothold in the page, our focus shifts to ensuring that the code provided to WRIT (i.e., callbacks into `post()`) has received no tampering from malicious extensions.

Initially, we confirm that a static analysis of the corresponding code snippets does not suffice, since an attacker can change JavaScript on the client at will (e.g., dynamically hook functions, inject code, etc.). Instead it is necessary to monitor the JavaScript execution at runtime, which is challenging for several reasons. First, the execution model of modern web environments should be taken into consideration, that is based in event-driven programming. In this model functions often execute asynchronously in response to events that are triggered by e.g., , network activity or user input. Second, the monitoring needs to be performed within the webpage, hence it should be implemented in a way that cannot be tampered by a malicious browser extension, either by hooking a monitoring function or by directly changing the variable(s) where the monitoring information is stored.

WRIT aims to take a snapshot of the current stack trace at a critical point of execution, typically within the `post()` function shown in Listing 4.3, that executes the (web service) developer-defined function responsible for collecting some required parameters before crafting a request. The main motivation for using stack traces is that by generating them within the provided functions that we want to protect, we can verify the integrity of their operation and detect if they have been called from a benign execution flow or if they are the product of a potentially malicious action.

The majority of modern web browsers provide at least one native implementation for stack trace generation, e.g., through the Error standard built-in object [20] in Firefox and Chrome. The correct function call sequence must be known in advance, so that any injection or alteration (caused by a malicious browser extension) of the sequence can be spotted by checking the function call sequence. The benign sequence can either be extracted manually by the developer (e.g., via a debugger [36]) or automatically (using a dynamic analysis tool [7]). However, the inspection of the stack trace alone as provided by the browser API is not enough, mainly because browser APIs do not provide any further details about the active stack frames (such as the program counter) besides the function names. Hence,

WRIT is not able to defend against attack cases where, an adversary injects a malicious function that (intentionally) has the same name as a benign function residing in a different scope to avoid naming conflicts, or in the same scope effectively overwriting that function. As a result, the attacker would successfully spoof a naive inspection of the stack trace, simply because in both cases the stack trace will contain a seemingly innocuous sequence of function calls.

To overcome this, we further enhance WRIT with the ability to diversify the original execution trace of critical functions, by creating **a series of pseudo-randomly generated redundancy layers** in the form of empty JavaScript functions. These new functions form a chain by calling one another in a particular sequence, dictated by a *seed* that has been used to randomly generate them. The diagram of Figure 4.4 showcases this mechanism.

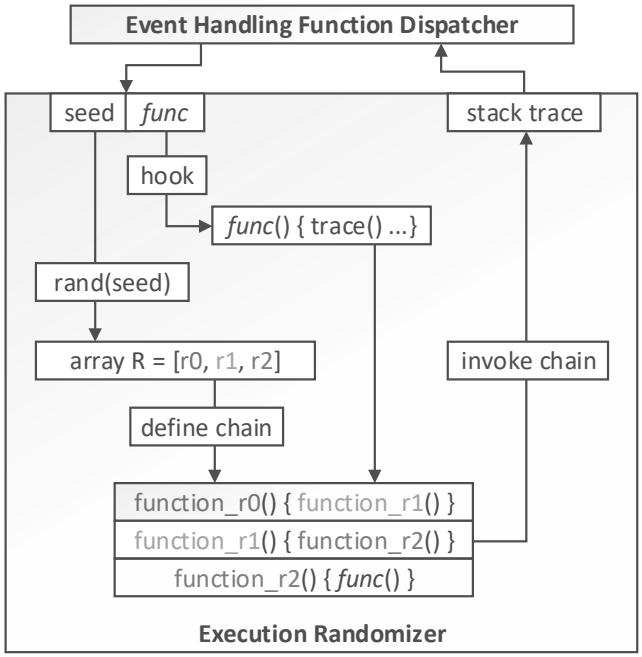

Figure 4.4: Overview of WRIT's random chain mechanism.

Every time WRIT needs to protect a critical function call $func$, which e.g., sends a request to the server, it creates one such chain and appends $func$ to the tail of the chain followed by a stack trace capture. By doing so, the generated stack trace will be enriched by the sequence of newly created functions, which uniquely identifies that particular execution of the critical function. More importantly, the random functions add extra levels of differentiation and entropy in the generated stack trace, which becomes exponentially hard for an adversary to spoof or predict (due to the added randomness). Hence, if a malicious extension manages to inject an extra function within the original execution path, the resulting stack trace will

not match the benign one.

The random functions are generated through `seedrandom` [10] and a *seed* that is known only to the Service Worker and WRIT's library within the page closure, so that both parties end up generating the same functions. Every time a new function chain has to be created, WRIT's in-page counterpart requests a new *seed* from the Service Worker. The reason we request a new seed every time is to reduce synchronization logic and the state that would otherwise be needed to be kept in the Service Worker in order to keep track of the asynchronous requests made from different parts of the web site.

We note that the communication between page and the Service Worker cannot be tampered by any browser extension, as it is securely performed via the in-page closure, as we describe in more detail in Section 4.5. Once the *seed* has arrived safely inside WRIT's in-page component, malicious extensions are unable to read the *seed* from within it because, as we described in Section 4.2 extensions do not have access to WRIT's interior scope. The *seed* is then used to generate a sequence of random numbers, each of which is appended to the name of a newly created dummy function. Each function contains a single instruction, which is set to invoke another function that has been assigned the next number of the sequence. The last function of the sequence is set to invoke the real function $func$ that WRIT wants to protect instead, which is hooked to produce a stack trace before it runs. This approach allows to control the randomization levels by adjusting the number of the resulted permutations $P(n, r) = n!/(n-r)!$, where n is the number of different function names and r is the number of random functions that we actually use each time. As we will see in Section 7, choosing a number for r between 10 and 100 incurs minor performance overhead while the probability of guessing the correct permutation is in the order of 10^{-18} for e.g., $r = 20$.

At this point the chain is formed and critical function $func$ is run by setting off the chain reaction that WRIT instrumented. After executing the dummy functions and $func$ successfully, WRIT's library sends the stack trace that was produced to the Service Worker, who will in turn use the same *seed* to generate a sequence of numbers and compare it against the sequence contained in the random functions previously added to the stack trace. The Service Worker's crucial role must be stressed for this part of the process: **validation is performed within the Service Worker, where no extension can observe or hinder it**. If the sequences are identical, execution of $func$ was successful without complications incurred by malicious extension intervention. In case of sequence mismatch, it is assumed that either the code segments have not executed as expected or have been tweaked by an extension, resulting in validation failure. In either case of success or failure, the Service Worker is configured to remove the random functions from the stack trace, sign the remainder and send it to the end server including a single bit indicating the outcome of validation. The outcome bit is essential in case of failure where a server would want to know something went wrong, but it could also prove useful in case of success as an indication that the Service Worker is still functional. This feature could be baked into the Service Worker as a more

secure approach, assuming that a clever malicious extension could entirely block the Service Worker's signed requests to the server.

4.4 Distinguishing Event Origins

The stack trace monitoring methodology described in Section 4.3, allows us to track conformant program execution and verify if a request has been generated through a benign control flow path (e.g., when a user clicks a button). We discover however that if a malicious extension generates e.g., the `MouseEvent` [22] that triggers the critical function $func$ artificially (i.e., programmatically), our methodology is unable to distinguish if the resulting request was created by the user or an extension. Further investigation highlights that the stack trace WRIT captures in the artificial event generation scenario does not contain any unexpected or malicious functions, simply because $func$'s execution path was not tampered in any way.

Generally, in order to successfully complete the generation of a web request, client-sided code often requires the completion of some user action (e.g., input text in a text form, selection of an item from a menu list, etc.). Malicious extensions though can perform such actions programmatically thanks to their direct and unrestricted access to the page's DOM and all the elements in it. More importantly, as this malicious code executes **before** firing the event that leads to running the client-sided function $func$ WRIT wants to protect, it ultimately eludes WRIT's stack trace. To protect against this type of malicious behavior, it becomes evident that we need to distinguish between human and non-human actions triggering events on DOM elements, e.g., button click events.

The metadata that are attached to each generated event object typically include several properties, some of which we find to have different values when the event is triggered by a human and non-human action. For instance, an event triggered programmatically via JavaScript code has a negative value in its aptly named `isTrusted` field. Similarly, two common properties that store the cursor's position when the event fires i.e., `pageX` and `pageY` have a value of zero. As per vendor documentation, the browser protects these metadata implicitly by restricting their access as read-only. We can attest to this protection functioning properly through experiments, in which we were unable to alter (or remove) these read-only properties. Therefore by checking the value of a combination of properties of a triggered event, WRIT can distinguish the source of DOM element interaction. These checks are performed within the private scope of WRIT's closure (Section 4.2) before initiating the attestation process as a preliminary filter.

4.5 Out-of-band Communications

We define two distinct communication channels in WRIT, as shown in Figure 4.1. One lies between the web page and the Service Worker, and the other between the

Service Worker and the web server. In the following two sections, we discuss in detail the role each channel plays in the context of WRIT's operation.

Communication between the web page and the Service Worker: As we describe in Section 3, the Service Worker has the ability to intercept and handle any request originating from the web page. In WRIT, we use custom URL requests in order to distinguish between normal HTTP traffic and requests that are meant for internal communication between WRIT's in-page component and the Service Worker. There are several unique URLs, each linked to a specific operation that either needs to request input from or send output to the Service Worker. The most significant operations are: (a) sending the *id* to the Service Worker (after it has been installed, during initial setup), (b) requesting a new *seed* from the Service Worker for stack trace generation and (c) sending a newly generated stack trace to the Service Worker for validation. The Service Worker is aware that these specific requests are meant for internal communication and responds back to the page accordingly without implicating the server. To mitigate against page-side impersonation attacks and guarantee secure communication between the page and the Service Worker, we authenticate the exchanged messages using the *token* that is exchanged between the Service Worker and WRIT's in-page component (see Section 4.2).

Communication between the Service Worker and the Web Server: The communication between the Service Worker and the web server is taking place over the network, as such it is susceptible to monitoring, interception and even blocking by malicious extensions through the webRequest API [27]. To overcome this threat scenario, we force the Service Worker to explicitly sign every protected request, so that the server can verify them and safely detect if a malicious extension has tampered with the Service Worker's request. In particular, the Service Worker validates that a protected request is benign (using the procedure that is described in Section 4.3) and then signs it using the secret key k that is stored within the Service Worker's isolated environment, provided by the server during the initial setup (see Section 4.1).

By doing so, the server can verify them and safely detect if a signed request has been tampered by a malicious extension or not. We note though, that a malicious extension has the power to completely block a request (as well as completely un-registering the Service Worker) and disrupt the normal operation of the web service/website. To detect such attacks, WRIT can be configured to use periodic heartbeats that are exchanged between the Service Worker and the server (signed by the secret key k). However, it would still not be easy to distinguish between cases that a user went offline due to a legitimate but unfortunate event (e.g., system crash, network failure, etc.) or due to a malicious action. As the main purpose of WRIT is only to attest the integrity of web requests, providing mitigation against these attacks is out of the scope of this work.

5. End-to-End Example

In this section we present a complete, step-by-step example scenario of a user visiting a website that uses WRIT. The corresponding steps are outlined in Figure 5.1 and Figure 5.2.

If this is the first time that the user visits the website, WRIT initiates the setup phase from a separate webpage (Figure 5.1). This webpage (i) asks for a randomly generated unique *id* from the server, (ii) it subsequently uses *id* to fetch the Service Worker via a one-time URL, (iii) then runs `register()` to install the new Service Worker and finally, (iv) uses *id* to exchange a secret key k with the web server that is kept safe in a local (private) variable.

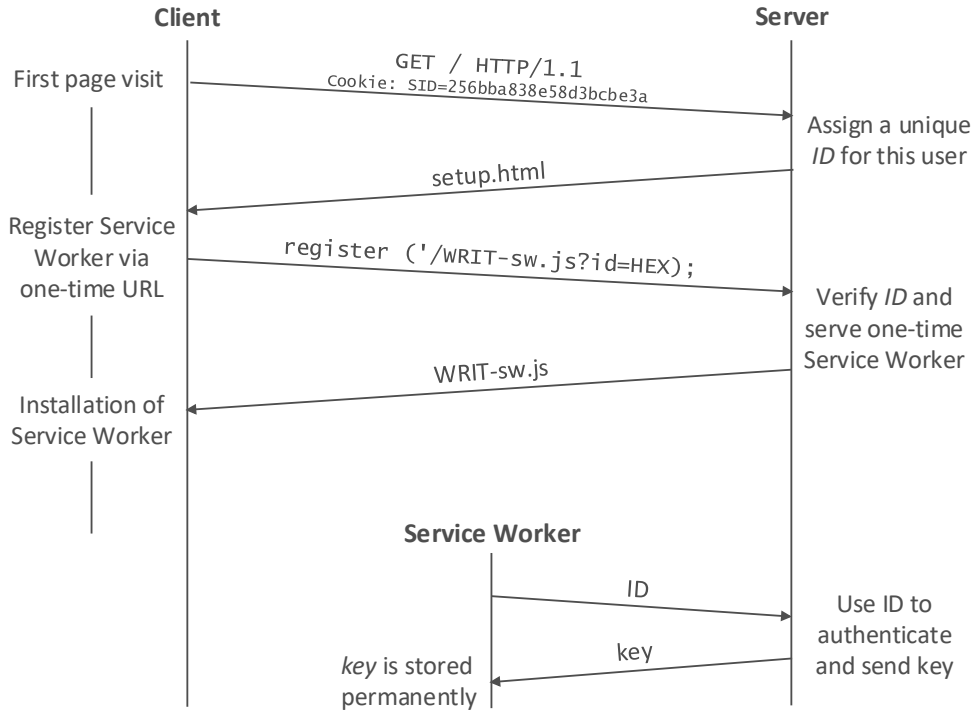

Figure 5.1: The initial setup of WRIT. The setup needs to run *only once*, typically when the user visits the website for the first time.

On every subsequent user visit to the website (Figure 5.2), WRIT's page component first needs to authenticate the communication channel between itself and the Service Worker. It fetches a new in-page script, then it sends its hardcoded unique id to the Service Worker, which is finally used between the page closure and the Service Worker to exchange a secret token.

From this point on, every time the user interacts with an element and triggers one of these events, (e.g., the user clicks on a button) the following actions take place before the original event handling function $func$: (i) a random *seed* is obtained from the Service Worker and is used to generate a sequence of functions forming a call chain with $func$, (ii) the chain is invoked and a stack trace then captures the random function sequence and $func$, (iii) the trace is sent to the Service Worker for validation of the random function sequence generated in the page against the one generated in the Service Worker using the same *seed*, (iv) lastly the Service Worker signs the request and sends it to the server.

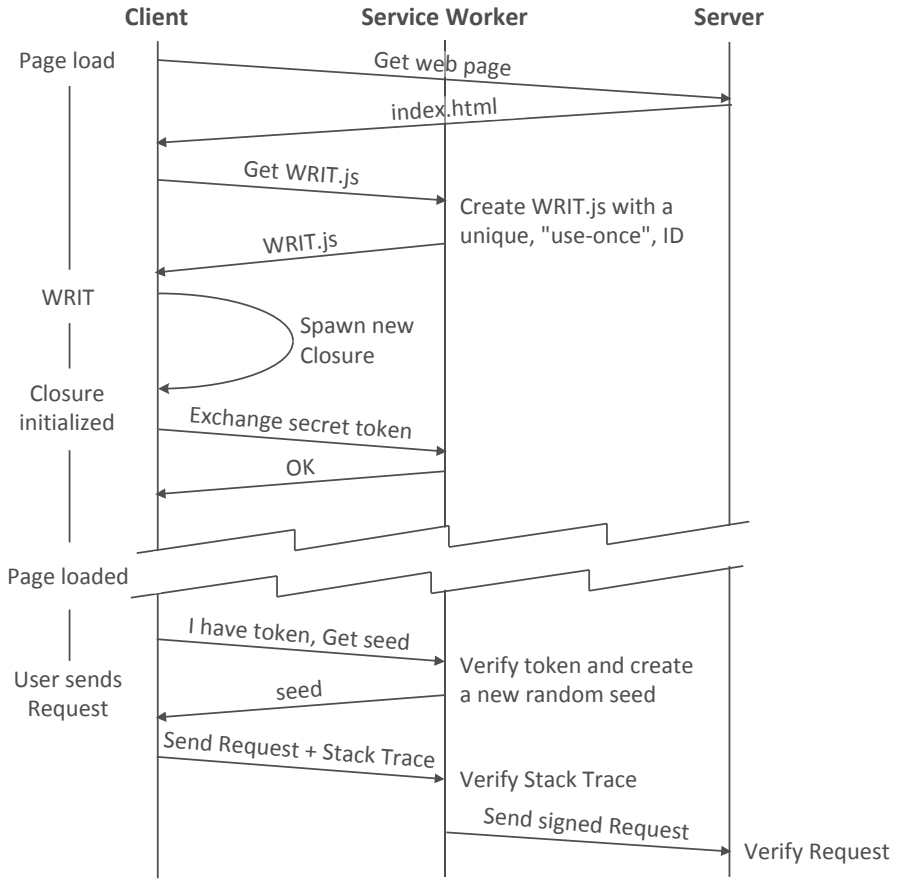

Figure 5.2: The messages exchanged during a user's session with WRIT.

6. Implementation

We developed a prototype implementation of WRIT, available at [2], in order to evaluate our approach and demonstrate its feasibility. Listing 6.1 presents a code snippet that shows how WRIT can be used to attest the requests that originate from an input element of the web page. As we can see, the developer only needs to provide a function that is responsible for getting the data from the corresponding DOM element(s) and crafting the request that needs to be sent to the web server. This function is passed as an argument to the `post()` function together with the user event that triggered this action. We also integrate our approach into Axios version v0.19.2; Axios [9] is a lightweight HTTP client API for creating web requests. We chose Axios as it offers many desirable features: (i) it offers a user friendly API for creating web requests, on top of the `XMLHttpRequest` API, (ii) it has become very popular in web development and is typically used in combination with the majority of modern web frameworks such as ReactJS, Angular and (iii) it is compatible with most modern browsers.

```
<body>
  <textarea id="tx">Hello World!</textarea>
  <script>
    function create_request() {
      let text = document.getElementById("tx").value;
      let config = {method: "POST", body: text};
      return {path: "/some_path", config};
    }
  </script>
  <button onclick='WRIT.post(create_request, e=event);'>
    Send POST via WRIT
  </button>
</body>
```

Listing 6.1: A simple page composed of a text input field, a function that creates a POST request with the input and a button that runs the function through WRIT's protection.

The changes needed to integrate WRIT within `axios.js` required about five lines of code in the `dispatchXhrRequest()` function, which is responsible for creating and sending a customized `XMLHttpRequest` according to user input, passed via a `config` object. The `config` object specifies, among others, the request's method type (GET or POST) and includes any request parameters or body content. Axios performs several tasks and checks after the creation of a new request inside `dispatchXhrRequest()`. The last task before the request is sent, is to check whether or not a "cancel token" is present in the supplied `config` object; if one is found, then the request is cancelled as per user request. We enrich that check with the result of an invocation to our WRIT public function which produces an embellished stack trace that must be verified by the currently operating Service Worker. If the latter verifies the trace successfully, Axios' request is sent, otherwise it is cancelled. The modified, WRIT-enabled Axios library is served to the client through our Service Worker, as described in Section 4.2.

Listing 6.2 shows an example of how the WRIT-enabled Axios API can be used to attest user requests *transparently*, even for legacy web applications that already utilize the Axios library.

```
1  <body>
2    <textarea id="tx">Hello World!</textarea>
3    <script>
4      function create_request() {
5        let text = document.getElementById("tx").value;
6        let config = {method: "POST", body: text};
7        return {path: "/some_path", config};
8      }
9    </script>
10   <button onclick='axios.post("", event, create_request);'>
11     Send POST via WRIT-enabled Axios
12   </button>
13 </body>
```

Listing 6.2: Similar to Listing 6.1, but this time using WRIT-enabled Axios.

7. System Evaluation

In this section, we evaluate our proposed architecture in terms of performance and security. Our base setup consists of two different machines: one server that hosts a simple web site and one machine that acts as a web client. The server is equipped with an AMD R5-3600 and 16GB of RAM, the client is equipped with an Intel I5-6300U and 8GB of RAM. The two machines are connected over a 1 GbE connection, which we shape accordingly to evaluate how the performance of WRIT scales on different network environments that represent different kind of web users, such as 3G, 4G, and LAN connections. The server that is used to host our website is running Flask v1.1.1 and uses Python v3.6.8. We have also configured our server so that it can serve over HTTPS, a requirement for the Service Worker API to expose itself in the page's context and become available to the client.

Browser Compatibility: Our approach is based on built-in browser components like Service Workers, which are supported natively by the majority of popular web browsers (Chrome, Firefox, Opera, Edge, Safari), as well as by most of their mobile counterparts (Samsung Internet, Android Chrome, iOS Safari, Android Firefox). A complete list of compatible browsers can be found in [47].

7.1 Performance Evaluation

We now evaluate WRIT in terms of performance. In particular, we measure the added overhead of WRIT in two scenarios: one under different network connection types and one in which the client increases the number of functions that are used for stack generation. For each scenario, we measure the time needed to perform a simple POST request on top of WRIT with all of its security mechanisms enabled and we compare it with the vanilla case (in which the request is simply sent to the server, with and without a Service Worker installed). For all our experiments we use Chrome with caching manually disabled. We also break WRIT's overhead down to four key components, listed in order of occurrence: the request to the Service Worker for a new seed, the generation of the stack trace in WRIT's closure, the processing and signing that takes place in the Service Worker and finally the signed request that is sent to the server.

Figure 7.1 shows the end-to-end (round trip) time of a POST request in the vanilla case versus the case where WRIT is enabled. We also plot a case where we place an empty Service Worker in the vanilla setup, to show the overhead added by the Service Worker alone. As shown, an empty Service Worker adds an overhead of about 2.97 ms on average (1.6 ms in the case of LAN). On top of that, we find that WRIT adds an additional overhead of as low as 5.69 ms in the case of a typical LAN setup (or 13.63 ms in the case of wifi). As a consequence, the overall end-to-end latency that WRIT adds to protect a sensitive POST request is as low as 7.29 ms.

In Figure 7.2, we break down the above overhead across all network presets and we see that latency is clearly dominated by the time needed to send the final request to the server. This was expected as it is the only network-bound operation. In contrast, given that WRIT runs with a baseline of 10 added stack functions in this scenario, the three remaining components remain nearly constant, within our margin of error (0.1ms). Overall, WRIT's network overhead increases as network conditions degrade across the different presets.

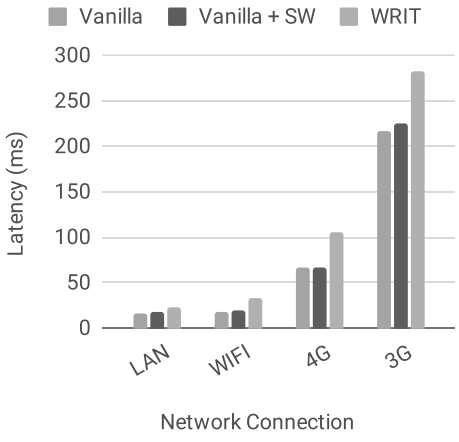

Figure 7.1: Network Overhead

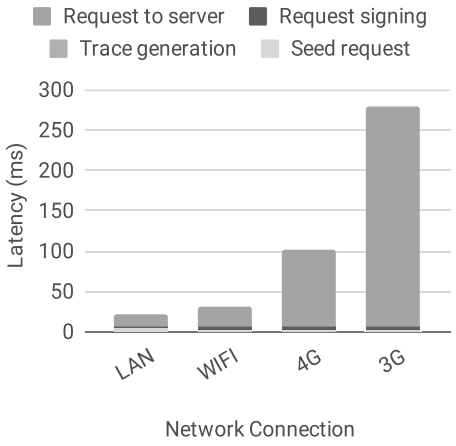

Figure 7.2: Network Breakdown

This limitation becomes even more evident as we move on to our second testing scenario, where by increasing the number of functions added to the stack trace, the trace's size directly increases along with the final request's body size[1]. The results of this scenario are shown in Figure 7.3, illustrating how end-to-end request latency is affected by the number of functions in the stack trace, in the LAN preset. What we see is that WRIT's baseline of 10 added functions can be easily expanded to 50 and 100 functions for a 0.7 ms and 1.5 ms latency increase respectively. Alternatively, halving the number of functions to 5 yields a 1 ms latency decrease.

[1]For the sake of completeness, we also measured (but didn't plot) the final request's body size: it starts at 1.8 KB with 10 functions, then grows to 6 KB with 50 functions and finally reaches 11.3 MB with 100 functions.

The number of extra functions directly affects the related WRIT components as shown in Figure 7.4. The stack trace generation itself grows from 0.18 ms latency using 10 functions, to 0.53 ms in case of 50 functions and 0.68 ms in case of 100 functions. By reducing down to 5 functions from 10, we get a latency of 0.24 ms, which is however within our margin of error. The Service Worker's seed generation procedure sees minor increases, taking up to 3 ms latency at both 10 and 100 added functions. Likewise, the signing procedure starts at 1.96 ms latency with 10 functions and peaks at 2.37 ms with 100 functions. As expected, the final request to the server remains constant at about 15-16 ms latency throughout all four function tiers. Finally, there is a constant 2.5 ms latency not accounted for in the plot, which is composed of miscellaneous Service Worker and browser tasks that are out of our control.

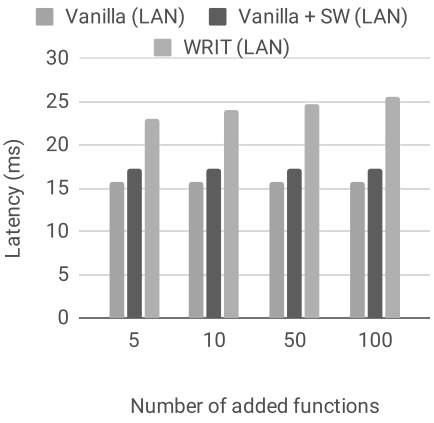

Figure 7.3: Compute Overhead

Figure 7.4: Compute Overhead Breakdown

Discussion on UI/UX: WRIT increases the latency of the web requests that have been issued explicitly through its API by 5-57 ms depending on network and browser conditions. We note though that this increase affects only security-critical web operations (e.g., submitting a form), which are usually generated through human-triggered actions: the effects in responsiveness are negligible, as the added latency remains well below the limits of having the user feel that the system is reacting instantaneously [48]. Finally, the latency of any other web request (such as those for getting normal web content, object fetches and asynchronous updates) is not affected by WRIT, hence neither the corresponding user-experience.

7.2 Security Analysis

We now evaluate the security properties of our proposed design by describing possible threat scenarios or attacks and addressing how WRIT protects against them. For many of the attacks described below, we have used custom browser

extensions or the browser built-in developer tools and debugger to get more insights and low-level operational details.

7.2.1 Tampering WRIT's Service Worker

As described in Section 3.3, browser extensions cannot interact with deployed Service Workers (e.g., monitor their execution, inspect their variables, etc.). That means that they cannot access the secret key k that is used for the attestation between the Service Worker and the end web server. In addition, the Service Worker script with the accompanied key is served *only-once*. Even though a malicious browser extension can request the corresponding JavaScript file after the user or access it by other means (e.g., through a legitimate sign process), it will not obtain the targetted user's key. On the other hand, requesting it before the user is not possible because the Service Worker is registered within a clean environment, typically when the user signs in for the first time as described in Section 4.1. Any future updates are initiated only within its script instead of replacing it entirely with a new one and are always signed with the corresponding key. Furthermore the Service Worker can be served over HTTPS only, hence its script is also well protected against MITM attacks.

Finally, a malicious extension that has been successfully installed on the user's browser can unregister WRIT's Service Worker. This can happen by getting a hold of the installed Service Worker through the API-provided `getRegistration` function and then calling the `unregister` function on it. Even in that case though, WRIT ensures that no malicious requests will be performed on the user's behalf, as any subsequent requests received by the server will not be signed by the key k, which in turn trigger an alert.

7.2.2 Tampering WRIT's in-page component

Similar to Service Workers, a browser extension cannot interact with a closure, interfere with its execution, access its context and the data stored within. However, contrary to the Service Worker which is permanent after its first registration, a closure is instantiated every time the user visits the web page. The most critical part of its establishment is when the in-page script is fetched from the server, due to the fact that browser extensions can hook the web page in various ways in order to tamper with the content that is received from the web server. As described in Section 2, browser extensions can deploy malicious code within the visited web page's context either directly through a content script or indirectly through a call to `executeScript` from the background script. Besides that, they can also utilize JavaScript code that is available in the page's context, just like any regular script fetched from the server, including WRIT's in-page script.

To protect WRIT's in-page script from tampering of its code when fetched from the server, we follow a different approach: instead of having the web server serve the script, we assign this task to our trusted Service Worker. By doing so,

the transfer is invisible to any browser extension, thus any malicious action on it is preemptively abolished. Moreover, it eliminates the possibility of a browser extension requesting the script on the user's behalf from the end server and using it to craft non-benign requests. Since browser extensions cannot communicate with the Service Worker, neither background nor content scripts of theirs can obtain a valid in-page script.

Furthermore, any critical data in WRIT i.e., the in-page script's *id*, the *key*, *seed*s and *stack trace*s, are kept private within appropriate closures as described in Section 4.2. WRIT's library only exposes a small set of public functions[2] in the page's global scope, which must be used to interact with WRIT. A malicious extension could overwrite (hook) the library's public functions with new versions that execute malicious code before executing the original function and vice versa. The library can repel this attack by verifying the integrity of its public functions upon use, restoring them to their original state if necessary and optionally raising an alarm notifying the user and/or the server for malicious activity. Alternatively, if a malicious extension prints them to probe for information, each public function simply reveals a call to the corresponding private library function, which is inaccessible due to the library's closure. If they are outright deleted, there is no repercussion to the library's operation beyond denial of access to the library for other (benign) page scripts, that can then re-fetch a fresh copy of the library's script (from the Service Worker).

7.2.3 Traffic Monitoring

Browser extensions can observe, intercept and modify requests and responses exchanged between the client and remote web servers via the `webRequest`, the `devtools.network` and the Chrome-only `chrome.debugger` APIs. All these three options grant extensions similar capabilities but require different permissions, which must be claimed accordingly in their manifests.

After experimentation, we have discovered that the `webRequest` API in Chrome does not provide any means of accessing the full responses i.e., only the response headers and not the bodies. Even though this can be helpful as it decreases the network monitoring threat posed by extensions, this behavior does not extend to Firefox specifically, which provides the `filterResponseData()` function in its implementation of the `webRequest` API. This function allows any extension to monitor and modify the body of a HTTP response received by the web server. WRIT is not vulnerable to this feature though, because it uses the Service Worker to serve any sensitive JavaScript code and data at the beginning of each session. Any communication between the web page and the Service Worker cannot be accessed by browser extensions employing `webRequest`, as such a malicious extension would be unable to monitor their exchanged traffic (both ways). Moreover, the

[2]WRIT's current design proposes that a single public function is exposed, creating a single point of defense, albeit providing WRIT's services in an inflexible and non modular way. Therefore, more could be carefully added according to developers' needs as they arise in practice.

Service Worker uses a secret key that has been securely exchanged with the web server, as described in Section 4. This key can be used to sign each and every request that enters the network and as such, can sufficiently protect web traffic against any kind of tampering or spoofing.

Besides `webRequest`, many popular browsers like Chrome and Firefox offer `devtools` APIs that grant the extensions extra capabilities (such as access to the console, network and performance tabs). These capabilities are typically available only in the developer tools panel and once obtained they grant extensions control over different aspects of the browser, originally only intended for development and debugging. However, extensions claiming the `devtools.network` permission can gain access to the respective API only while the browser's developer tools panel is open, due to the fact that the API is exposed only to very specific pages and scripts that are used to implement new tabs for the panel. While very powerful, the danger it poses is severely limited as any user is bound to be alarmed by their browser's devtools panel opening randomly (assuming extensions are or become capable of programmatically opening the panel, a task we have been unable to achieve so far). A counterargument can be made here, suggesting that an extension could open the dev console panel for a split second to execute its malicious code, although as soon as the panel closes the extension loses access to the relevant `devtools` API commands[3]. Finally, in case an extension attempts to trick the user into opening the panel manually, many websites and popular web applications have been inserting warnings directly into the browser's console panel in order to mitigate self-hacking (e.g., Facebook as shown in Figure 7.5).

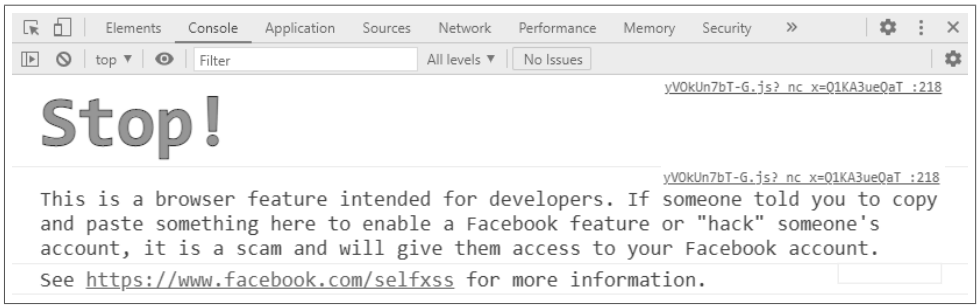

Figure 7.5: Facebook's warning against self-induced cross site scripting in Chrome's DevTools console tab.

Similarly, the Chrome-specific `chrome.debugger` API provides extensions with functions that have elevated access to a visited page's network activity, DOM elements and script execution. In essence, it provides elevated access to response

[3]In practice, it's hard to accurately estimate how much damage an extension could cause at so short a time frame such that a user doesn't detect the anomaly, especially since the malicious code's execution time is restricted by hardware capabilities and the complexity of the task.

(and request) handling, in turn allowing the inspection and modification of response bodies. When an extension that claims the `chrome.debugger` permission attaches Chrome's debugger to an open tab via `chrome.debugger.attach()`, the browser displays a screen-wide banner below the user's bookmark bar; text within this banner indicates that an extension (whose name is displayed) is debugging the browser, accompanied by a button to cancel its operation. This visual indication is at least as strong as the previously discussed opening of the devtools panel, alarming any user to the possibility of their browser being compromised, especially after an extension's installation. There are also JS antidebugging tricks [33] that we can leverage in order to detect the presence of the debugger and act accordingly.

7.2.4 Replay attacks

We defend against the replay of signed requests by including a non-repeating count in every request (Section 4.1). Each request is signed using the string representation of the request, combined with the value of the current count. A malicious extension that is eavesdropping benign requests cannot replay them in the future, as the server will notice the repetition of an old count. However, a malicious extension could block a specific signed request from reaching the web server and storing it locally; this signed request could be sent later on to the server successfully. Even though we are not aware —to the best of our knowledge— of a real-world scenario that can be exploited from this hiccup, this can still be fixed by further applying one-time passwords or even timestamps to the signing process.

7.2.5 Blocking web requests and responses

A malicious extension has the ability to block any network traffic from and to the web service. To detect such types of attacks, we could configure the Service Worker to periodically send heartbeats (signed by the secret key k). However, it would still not be easy to distinguish between cases where a user goes offline due to a legitimate but unfortunate event (e.g., system crash, network failure, etc.) or due to a malicious action. In a case where a malicious extension selectively blocks the web traffic flowing to the website, the heartbeats would operate successfully. Even though WRIT is not able to provide further protection to the already infected client, it can still ensure that no malicious requests will be performed on the user's behalf. Overall, the main purpose of WRIT is only to attest the integrity of web requests, defending against these attacks is out of the scope of this work.

8. Related Work

Since their inception over 20 years ago, browser extensions have always been a very attractive vehicle for malicious code deployment. Sometimes installed by millions of users [44, 40], they infect end-user web browsers to exfiltrate sensitive data and perform many other malicious actions during the browsing sessions of unsuspecting users.

Analysis: Kapravelos et al. [44] analyzed 48K extensions from the Chrome Web store and identified several large classes of malicious behavior, including affiliate fraud, credential theft, ad injection or replacement, and social network abuse. In [40] the authors analyze multiple browser extensions and indentify a set of 9,523 malicious ones. By using both static and dynamic analysis techniques, they show that extensions typically abuse `contentScript` permissions to perform malicious activities, such as Facebook hijacking, ad injection, search leakage and user tracking. Another analytical avenue proposed by past studies is to employ machine learning techniques offline, in order to detect malicious behavior. For example, in [60], the authors develop a deep learning framework that detects malicious JavaScript code with an accuracy of about 94%. Even though such frameworks can provide high accuracy, they always need to feed on new data in order to keep the accuracy high, as new obfuscation/malicious JavaScript techniques are developed.

Hardware Enclaves: With the advent of trusted execution environments on commodity hardware (e.g., Intel SGX, Android Trusty), some approaches show how they can be utilized to provide a safe and protected space for sensitive data storage and code execution [35, 30]. TrustJS [35] enables trustworthy execution of security-sensitive JavaScript inside commodity browsers, using Intel's SGX. Fidelius [30] utilizes secure SGX enclaves provided by Intel SGX to offer an end-to-end secure path between the I/O devices and the browser. This approach requires the establishment of secret keys between the keyboard and the monitor and ensures that both input and output data cannot be intercepted by malicious software. However, the specialized hardware features that these approaches are based on (i.e., Intel SGX) are currently available only in vendor-specific models (i.e., sixth generation Intel Core microprocessors and onwards in the case of SGX or specific ARM SoCs for Trusty). This makes such approaches difficult to scale, given the diversified types of devices that users use to access the Internet (from handheld devices, to smart IoT equipment and laptops). Also keeping the Web ecosystem's constant

and rapid evolution in mind, widespread adoption of new systems is only possible through features and APIs that are available to most or all users. To that end, platform-specific hardware features are a definite barrier to entry.

Older Defences for Untrusted Clients: In [62], authors propose ZigZag to strengthen JavaScript-based web applications against client-side validation attacks. Ripley [59] tries to tackle the problem of untrusted clients by automatically replicating the execution of client-side JavaScript on a trusted server tier, thus preserving the integrity of a distributed computation. Every triggered client-side event is transferred to the server's replica for execution. Ripley observes the results of the computation, both as computed on the client-side and on the server-side, so possible discrepancies are flagged as potential violations of computational integrity. However, Ripley imposes network and memory overhead, as it transfers and replays every client event to the server. Web Tripwires [54] is client-side JavaScript code that can detect most in-flight modifications to a web page, and prevent changes in the received changes (e.g., popup blocking scripts, advertisements, malicious code that can cause harm). Similary, Glasstube [37] uses a lightweight approach that protects the integrity of web applications against network MITM attacks (such as session hijacking, reordering and replay attacks). However, with the great adoption of the HTTPS in the Internet nowadays, such techniques have become outdated.

Newer Defenses for Untrusted Clients: DOMtegrity [58] is an approach that ensures the integrity of web content in the client's browser, by using client-side code to verify that the DOM structure has not been altered by a malicious browser extension. However, malicious extensions or third party applications can still perform malicious actions, such as crafting requests on a user's behalf. While WRIT cannot attest the integrity of visited pages' structure, it is able to attest all requests and detect any abnormal or malicious requests that have not been created with the user's consent. Caja [34] is a security sandbox that allows to safely embed third-party HTML, CSS and JavaScript in a website and enables developers to control the permissions of code over a user's data. However, a browser extension can still perform malicious actions on a user's behalf. WRIT is able to further protect against malicious extensions, as well as third-party libraries and code.

Untrusted Servers: Many works aim to maintain client-side integrity when facing a compromised server. Mylar [53], ShadowCrypt [38], and CryptDB [52] aim to provide confidentiality against a corrupted web server using encryption. A different approach by Verena [45] provides a trusted client-side component in the user's browser, that can verify the integrity of a web page by verifying the results of queries on data stored at the server.

Browser/OS Modifications: While WRIT's design is intentionally restricted to standardized in-browser features and APIs to avoid browser modification, there are numerous alternative approaches proposed in the last two decades that involve modifying the client's browser or underlying operating system. For example, SubOS [56] proposes a kernel-level protection mechanism that changes how applications treat malicious incoming objects, restricting their access to system resources like an operating system would. Secure Browser [57] further leverages

support from the underlying operating system to design a secure web browser that protects against malicious incoming objects.

9. Conclusion

The current state-of-the-art shows a significant lack of attention to client-sided integrity and attestation. This thesis presented WRIT, as a first step of a new line of security mechanisms: a lightweight framework capable of verifying the integrity of web requests that operate in an isolated environment malicious browser extensions cannot breach or tamper. At its core, WRIT strives to ensure that protected web requests have been created through a benign execution path and not generated by malicious third-party JavaScript code or a malicious browser extension. WRIT is designed with ease of adoption and integration in mind as it uses standard browser and language features exclusively, without requiring any browser modifications or the installation of a browser extension. Developers can utilize WRIT's API to protect security-critical web requests at a practically negligible cost of about 7.29 ms latency.

10. Discussion and Future Work

While WRIT achieves the goals set at its inception, it is also important to discuss future work that may enhance its capabilities or address some of its weaknesses.

10.1 Unregistration

As described in Section 4.1, extensions cannot interact with a Service Worker but can still unregister it to disrupt WRIT's operation. Following that, two possible scenarios take form:

- If the critical function(s) protected by WRIT (e.g., powering a payment submission on user click) were hardcoded to only run through WRIT's API, then they won't be able to run at all and the user suffers a denial of service. In response, the client-sided code can alert the remote web server to WRIT being unavailable.
- Alternatively if the critical function is coded to run even when WRIT's API is dysfunctional, then the function runs but the produced request will reach the web server unsigned by WRIT's Service Worker. Here the server can raise an alarm upon these unexpected arrivals.

In both cases, the server should issue a full reset of WRIT in the client coupled with a warning. This process would require disabling all of the user's extensions and running WRIT's initial setup process again.

Even though this is an attack WRIT resists, it highlights WRIT's weakest point and ultimately its susceptability to disruption. The source of the issue i.e., the extensions' access to the `unregister()` function of the Service Worker API lies in every browser's implementation of said API and possibly even their implementation of browser extensions. Unless browser vendors revoke this access in the future, we consider the elimination of this attack vector an important part of future work.

10.2 Browser Reliance

As a security mechanism deployed in the user's browser, WRIT finds itself in a precarious position where it is directly dependant on the underlying browser for

its efficacy, its reliability and even its existence in the first place.

The ever-evolving nature of the modern browser landscape has already caused a few minor and major changes throughout WRIT's design and implementation. Firefox for example gives extensions access to HTTP response bodies (see Section 4.2), a deviation from Chrome and the other Chromium-based modern browsers. This one seemingly minor difference between browsers caused significant changes in WRIT's design. Similarly, we analyzed in Section 10.1 how extensions having access to a single function opens up a potent attack vector against WRIT, one that we are incapable of fully mitigating without modifying the browser itself.

Worst of all, there is no guarantee that modern browser vendors will not eventually increase the tools available to extensions, in turn making them even more powerful. For instance, extensions already have access to the primary communication method intended for Service Workers i.e., `postMessage()`, which forces WRIT to use custom web request/response flows instead (see Section 3.3). If they are allowed to monitor and intercept the flows between web page and Service Worker, WRIT's design would have to adapt in response, likely employing an alternative means of communication that is more complex and/or cumbersome. Additionally, there is the possibility of undermining WRIT by altering or removing core features it relies on; major examples being of course Service Workers (which are thankfully gaining adoption by the day), the SubtleCrypto API and JavaScript closures.

On the other hand, WRIT also stands to benefit from changes in browser features, APIs and the implementation of extensions, bringing about positive changes in WRIT's design and implementation i.e., simplifying the setup and attestation phases or limiting extensions' access to powerful APIs etc. Since we are unable to influence these decisions however, maintaining WRIT's correctness and effectiveness in the future will be a constant work in progress.

10.3 Platform Expansion

This thesis has focused on developing a secure execution environment in the browser, specifically for the purpose of providing protection against malicious browser extensions. We strongly believe however that WRIT's Service Worker powered architecture could serve as a launching pad for many other security and privacy applications. Due to time constraints, we only tested the feasibility of this concept anecdotally with two relatively simple applications, alongside WRIT's baseline attestation mechanism.

2FA: The first one is a two-factor authenticator that runs in WRIT's Service Worker, using the JS-OTP [42] library that imeplements HOTP (hash-based) and TOTP (time-based) one-time passwords in JavaScript. Our simple proof of concept consists of a webpage that requires a one-time password in order to authenticate a user (that has signed up for two-factor authentication), before allowing them to access e.g., their account or a service feature. Nowadays most services and

websites encourage or require users to install an authenticator app on their phone. In our approach the user doesn't need a secondary device or application, because the website's client-sided code can retrieve the one-time password directly from the installed Service Worker. As an added benefit, the critical function that fetches the one-time password can also be protected by WRIT's attestation procedure.

This approach might initially seem less secure, given that a secondary device inherently provides an additional layer of protection (the second factor) in case the user's credentials are compromised. On the contrary, the proposed authenticator leverages WRIT as the second factor; thus even if the user's credentials are compromised by a malicious actor, the latter would not have WRIT's Service Worker registered in their browser and as a result would not be able to obtain the one-time password required to log into the victim's account.

Click Fraud: The second application is a slight modification of WRIT targetting a particular type of web service: advertisement providers. Malicious extensions have become very popular among ad fraud perpetrators (see Section 1), often found injecting artificial ads in users' visited pages, artificially clicking benign ads in them or even sending artificial requests to the ad domains directly, without clicking an ad at all. To challenge this malicious activity, we fork WRIT towards two different implementations depending on whether the ad provider's ads are served in `iframe` elements (industry norm) or in other types of elements:

- In the case of iframes, the ad is located within the iframe which has the ad provider's domain as its source. This is an important detail, as Service Workers intercept requests from pages that are part of their registration scope, as described in Section 3.1. Since the iframe's source indicates the ad provider's domain, the first party website containing the iframe (and ads) must direct its users to the ad provider in order to perform WRIT's initial setup; notably, WRIT's Service Worker is served by the ad provider in this case.

- Otherwise, the ad is placed within any other conventional element i.e., a `div` and its origin is the first party website the user visits. Therefore, WRIT can be set up normally and its Service Worker is served by the first party website.

In either case, the client-sided function that runs upon clicking the ad should be wrapped by WRIT's `post()` function. WRIT's Service Worker can then distinguish between artificial and user-initiated ad clicks. Furthermore, any artificial requests that were created programmatically will reach the server unsigned, making it trivial for the ad provider to discard them.

Bibliography

[1] Trojan:JS/Kilim is a family of malicious browser extensions that post unauthorized content to the user's Facebook Wall. https://www.f-secure.com/v-descs/trojan_js_kilim.shtml.

[2] WRIT Project. https://anonymous.4open.science/r/df11f608-44c0-4e70-9dba-7196bb66510b/.

[3] Malicious chrome extensions enable criminals to impact half a million users and global businesses. https://atr-blog.gigamon.com/2018/01/18/malicious-chrome-extensions-enable-criminals-to-impact-half-a-million-users-and-global-businesses/, 2018.

[4] Chromium issue 793217: "document_start" hook on child frames should fire before control is returned to the parent frame. https://bugs.chromium.org/p/chromium/issues/detail?id=793217, 2019.

[5] Ta413 leverages new friarfox browser extension to target the gmail accounts of global tibetan organizations. https://www.proofpoint.com/us/blog/threat-insight/ta413-leverages-new-friarfox-browser-extension-target-gmail-accounts-global, 2021.

[6] Devdatta Akhawe, Warren He, Zhiwei Li, Reza Moazzezi, and Dawn Song. Clickjacking Revisited: A Perceptual View of UI Security. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)*, USENIX WOOT, 2014.

[7] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A Survey of Dynamic Analysis and Test Generation for JavaScript. *ACM Comput. Surv.*, 50(5), September 2017.

[8] Sajjad Arshad, Amin Kharraz, and William Robertson. Identifying Extension-based Ad Injection via Fine-grained Web Content Provenance. In *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions, and Defenses*, RAID, 2016.

[9] Axios. The axios project. https://github.com/axios/axios.

[10] David Bau. Javascript random number generator "seedrandom" repository. https://github.com/davidbau/seedrandom.

[11] Mariano Ceccato and Paolo Tonella. CodeBender: Remote Software Protection Using Orthogonal Replacement. *IEEE Software*, 28(2):28–34, 2011.

[12] Chromium Bugs. Figure out how Service Worker and Web Request API should interact. https://bugs.chromium.org/p/chromium/issues/detail?id=766433.

[13] Catalin Cimpanu. Google removes 500+ malicious Chrome extensions from the Web Store. https://www.zdnet.com/article/google-removes-500-malicious-chrome-extensions-from-the-web-store/.

[14] Catalin Cimpanu. "Particle" Chrome Extension Sold to New Dev Who Immediately Turns It Into Adware. https://www.bleepingcomputer.com/news/security/-particle-chrome-extension-sold-to-new-dev-who-immediately-turns-it-into-adware/, 2017.

[15] Tomer Cohen. Game of Chromes:Owning the Web with Zombie Chrome Extensions. https://www.blackhat.com/docs/us-17/thursday/us-17-Cohen-Game-Of-Chromes-Owning-The-Web-With-Zombie-Chrome-Extensions-wp.pdf.

[16] Christian Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra. Distributed Application Tamper Detection via Continuous Software Updates. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12. Association for Computing Machinery, 2012.

[17] MDN contributors. Background. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/background.

[18] MDN contributors. Client.postmessage(). https://developer.mozilla.org/en-US/docs/Web/API/Client/postMessage.

[19] MDN contributors. Closures. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures.

[20] MDN contributors. Error. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error.

[21] MDN contributors. Javascript apis. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API.

[22] MDN contributors. Mouseevent. https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent.

[23] MDN contributors. Promise. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

[24] MDN contributors. Service worker api. https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API.

[25] MDN contributors. Subtlecrypto. https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto.

[26] MDN contributors. Updating the service worker. https://developers.google.com/web/fundamentals/primers/service-workers/lifecycle\#updates.

[27] MDN contributors. webrequest. https://developer.mozilla.org/en-US/Add-ons/WebExtensions/API/webRequest.

[28] Google Developers. Angular: One framework. mobile & desktop. https://angular.io.

[29] M. Dhawan and V. Ganapathy. Analyzing Information Flow in JavaScript-Based Browser Extensions. In *2009 Annual Computer Security Applications Conference*, ACSAC, 2009.

[30] Saba Eskandarian, Jonathan Cogan, Sawyer Birnbaum, Peh Chang Wei Brandon, Dillon Franke, Forest Fraser, Gaspar Garcia Jr., Eric Gong, Hung T. Nguyen, Taresh K. Sethi, Vishal Subbiah, Michael Backes, Giancarlo Pellegrino, and Dan Boneh. Fidelius: Protecting User Secrets from Compromised Browsers. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, IEEE SP, 2019.

[31] Facebook Developers. React: A javascript library for building user interfaces. https://reactjs.org.

[32] Nicholas Fearn. Nearly 80 chrome extensions caught spying – how to protect yourself. https://www.tomsguide.com/news/chrome-extension-spyware, 2020.

[33] Juan Manuel Fernández. Javascript antidebugging tricks. https://x-c3ll.github.io/posts/javascript-antidebugging/, 2020.

[34] Gogle Develoopers. Caja Project. https://developers.google.com/caja/.

[35] David Goltzsche, Colin Wulf, Divya Muthukumaran, Konrad Rieck, Peter Pietzuch, and Rüdiger Kapitza. Trustjs: Trusted client-side execution of javascript. In *Proceedings of the 10th European Workshop on Systems Security*, ACM EuroSec, 2017.

[36] Google Developers. chrome.debugger. https://developer.chrome.com/extensions/debugger.

[37] Per A. Hallgren, Daniel T. Mauritzson, and Andrei Sabelfeld. GlassTube: A Lightweight Approach to Web Application Integrity. In *Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, ACM PLAS, 2013.

[38] Warren He, Devdatta Akhawe, Sumeet Jain, Elaine Shi, and Dawn Song. ShadowCrypt: Encrypted Web Applications for Everyone. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ACM CCS, 2014.

[39] Lin-Shung Huang, Alex Moshchuk, Helen J Wang, Stuart Schecter, and Collin Jackson. Clickjacking: Attacks and defenses. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, USENIX Security, 2012.

[40] Nav Jagpal, Eric Dingle, Jean-Philippe Gravel, Panayiotis Mavrommatis, Niels Provos, Moheeb Abu Rajab, and Kurt Thomas. Trends and Lessons from Three Years Fighting Malicious Extensions. In *Proceedings of the 24th USENIX Security Symposium*, USENIX Security, 2015.

[41] Richi Jennings. Chrome web store fail: 300+ more scam browser extensions. `https://securityboulevard.com/2020/08/chrome-web-store-fail-300-more-scam-browser-extensions/`, 2020.

[42] Allan Jiang. 100authentication. `https://github.com/jiangts/JS-OTP`, 2021.

[43] Jordan Jueckstock and Alexandros Kapravelos. VisibleV8: In-browser Monitoring of JavaScript in the Wild. In *Proceedings of the ACM Internet Measurement Conference*, ACM IMC, 2019.

[44] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. Hulk: Eliciting Malicious Behavior in Browser Extensions. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, USENIX Security, 2014.

[45] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun. Verena: End-to-End Integrity Protection for Web Applications. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 895–913, 2016.

[46] Maxime Kjaer. Malware in the browser: how you might get hacked by a chrome extension. `https://kjaer.io/extension-malware/`, 2016.

[47] Alexis Deveria Lennart Schoors. Can i use service workers? `https://caniuse.com/\#feat=serviceworkers`.

[48] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[49] Marcus Niemietz and Jörg Schwenk. Out of the Dark: UI Redressing and Trustworthy Events. *Cryptology and Network Security*, pages 229–249, 2018.

[50] 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*, NDSS, 2019.

[51] Alex Perekalin. Why you should be careful with browser extensions. `https://www.kaspersky.com/blog/browser-extensions-security/20886/`, 2018.

[52] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP b••11, page 85b•"100, New York, NY, USA, 2011. Association for Computing Machinery.

[53] Raluca Ada Popa, Emily Stark, Jonas Helfer, Steven Valdez, Nickolai Zeldovich, M. Frans Kaashoek, and Hari Balakrishnan. Building Web Applications on Top of Encrypted Data Using Mylar. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDIb••14, page 157b•"172, USA, 2014. USENIX Association.

[54] Charles Reis, Steven Gribble, Tadayoshi Kohno, and Nicholas Weaver. Detecting in-flight page changes with web tripwires. In *5th USENIX Symposium on Networked Systems Design & Implementation*, NSDI 2008, pages 31–44, 01 2008.

[55] Guido Schwenk, Alexander Bikadorov, Tammo Krueger, and Konrad Rieck. Autonomous Learning for Detection of JavaScript Attacks: Vision or Reality? In *Proceedings of the 5th ACM Workshop on Artificial Intelligence and Security*, ACM AISEC, 2012.

[56] Jonathan M. Smith Sotiris Ioannidis, Steven M. Bellovin. Sub-operating systems: A new approach to application security. `https://dl.acm.org/doi/10.1145/1133373.1133394`, 2002.

[57] Steven M. Bellovin Sotiris Ioannidis. Building a secure web browser. `https://www.usenix.org/legacy/event/usenix01/freenix01/full_papers/ioannidis/ioannidis_html/`, 2001.

[58] Ehsan Toreini, Siamak F. Shahandashti, Maryam Mehrnezhad, and Feng Hao. DOMtegrity: ensuring web page integrity against malicious browser extensions. *International Journal of Information Security*, 18(6):801–814, 2019.

[59] K. Vikram, Abhishek Prateek, and Benjamin Livshits. Ripley: Automatically Securing Web 2.0 Applications Through Replicated Execution. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ACM CCS, 2009.

[60] Yao Wang, Wan-dong Cai, and Peng-cheng Wei. A deep learning approach for detecting malicious JavaScript code. In *Security and Communication Networks*, 2016.

[61] Michael Weissbacher, Enrico Mariconti, Guillermo Suarez-Tangil, Gianluca Stringhini, William Robertson, and Engin Kirda. Ex-ray: Detection of history-leaking browser extensions. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 590–602, 2017.

[62] Michael Weissbacher, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Zigzag: Automatically hardening web applications against client-side validation vulnerabilities. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 737–752, 2015.