The Remote on the Local: Exacerbating Web Attacks Via Service Workers Caches

Marco Squarcina
TU Wien
marco.squarcina@tuwien.ac.at

Stefano Calzavara
Università Ca' Foscari Venezia & OWASP
calzavara@dais.unive.it

Matteo Maffei
TU Wien
matteo.maffei@tuwien.ac.at

Abstract—**Service workers boost the user experience of modern web applications by taking advantage of the Cache API to improve responsiveness and support offline usage. In this paper, we present the first security analysis of the threats posed by this programming practice, identifying an attack with major security implications. In particular, we show how a traditional XSS attack can abuse the Cache API to escalate into a person-in-the-middle attack against cached content, thus compromising its confidentiality and integrity. Remarkably, this attack enables new threats which are beyond the scope of traditional XSS. After defining the attack, we study its prevalence in the wild, finding that the large majority of the sites which register service workers using the Cache API are vulnerable as long as a single webpage in the same origin of the service worker is affected by an XSS. Finally, we propose a browser-side countermeasure against this attack, and we analyze its effectiveness and practicality in terms of security benefits and backward compatibility with existing web applications.**

I. Introduction

Progressive Web Applications (PWAs) are the latest trend in the tremendous evolution of web applications [27]. PWAs offer a user experience similar to traditional mobile / desktop applications by providing extreme responsiveness and supporting offline usage, e.g., in absence of connectivity, while taking full advantage of inherently online features whenever possible. *Service workers* [28] are the key enabler of PWAs, since they can act as client-side web application proxies able to intercept HTTP requests and immediately serve previously cached HTTP responses. This practice, enabled by the Cache API [22], greatly improves usability, yet its security implications are unclear.

In this paper, we analyze the design of the Cache API available to service workers and we identify new security threats enabled by its adoption. In particular, we discuss how the Cache API can *exacerbate* the dangers normally posed by malicious scripts running in the web application origin, e.g., as the result of a successful XSS exploitation. By tampering with cached HTTP responses, a traditional web attack like XSS can achieve results equivalent to a *person-in-the-middle* attack, which normally requires network capabilities and the lack (or misuse) of transport layer defenses. The attack presented in this paper breaks the confidentiality and the integrity of cached content, which can lead to a wide range of severe threats. Remarkably, the ability to corrupt cached HTTP responses allows the attacker to bypass the protection of security headers and defensive programming practices, which normally mitigate the impact of a successful XSS exploitation. Moreover, since the attack corrupts client-side data, it is persistent and naturally amplified to all pages serving content from the cache on the same origin.

These issues are not purely theoretical, but can easily affect any web application which registers a service worker using the Cache API, as long as the attacker can get active scripting capabilities even on a single page of the web application. While automated testing for script injection is beyond the scope of our study, XSS is the most common web vulnerability, which affects around two-thirds of vulnerable web applications and is routinely discovered even on high-profile websites [13]. Our large-scale analysis on 150,000 websites from the Tranco list identifies that 95.8% of the 3,436 websites which register a service worker using the Cache API are potentially vulnerable to the threats described in our paper, including prominent sites such as Google Developers and WhatsApp. By simulating script injection at scale on these sites via a browser extension, we assess that 65% of the 2,796 sites caching HTML or JavaScript files would be affected by this attack in presence of just a single XSS vulnerability in their codebase.

Based on our security analysis, we argue that the key design flaw of the Cache API is that the cache used by service workers is shared with any script running in the same origin of the web application. To prevent the newly identified threats, we propose as a countermeasure a redesign of the Cache API so that it is only available to service workers. Of course, this major overhaul might lead to the breakage of existing web applications, hence we also discuss a simple mechanism to relax the scope of the cache to include scripts when needed. We experimentally confirm through a web measurement that the large majority of the websites using the Cache API (93.1% out of 3,537) do not need to make the cache accessible to scripts, and they would thus transparently get automated protection against the attack. As to the remaining websites, we identify caching patterns that could be revised to avoid the need of exposing the cache to scripts in most cases. We also discuss a simple programming practice that web developers can immediately implement to detect and discard tampered entries in the service workers cache.

Contributions: To the best of our knowledge, this work presents the first security analysis of the threats posed by the use of the popular Cache API in service workers, proposing an effective and practical browser-side technique to secure its usage in modern web applications. Specifically, we make the following contributions:

1) We demonstrate that the current design of the Cache API poses major security threats, since it allows a web attacker with scripting capabilities to act as a person-in-in-middle against cached content (Section III), thus exacerbating the dangers of standard XSS and enabling new attacks (Section IV).
2) We perform a large-scale empirical study on the top 150,000 websites from Tranco to confirm that the dangers posed by the Cache API are real. Our analysis found that 95.8% of the 3,436 websites using the Cache API in service workers are potentially vulnerable, including high-profile websites, as long as a single webpage in the same origin of the service worker is affected by an XSS vulnerability. By simulating script injection via a browser extension, we confirm that the majority of these sites would be vulnerable to the attack described in this work (Section V).
3) We propose countermeasures to prevent the new threats presented in our work. In particular, we argue that the Cache API should be made accessible only to service workers by default. We quantify the websites that are using the cache outside of service workers (7.2% out of 3,537 sites using the Cache API), and we identify caching patterns that would be affected by the proposed redesign. To ensure compatibility with existing web applications, we outline a server-defined mechanism that allows site operators to instruct the browser to expose the Cache API to other same-origin contexts (Section VI).

II. Background

In this section, we provide an overview of service workers and their use for caching remote content.

A. Service Workers

A service worker is an event-driven and browser-handled JavaScript program, which can be programmatically *registered* by a web application. It plays the role of a client-side proxy, since it can intercept and modify the HTTP requests issued by the web application and the corresponding HTTP responses. Coupled with the Cache API, it can also be used to store HTTP responses and then serve them even in absence of connectivity. Figure 1 shows the proxy-like position of a service worker.

The service worker life-cycle is managed by the browser, which can put it to sleep or wake it up depending on whether there are events to handle or not. Although service workers share the same *origin*[1] of the web application they manage, they execute in a separate and isolated *context*, different from that of other same-origin web pages and workers. For example, service workers do not have access to the DOM.

For security reasons, service workers can only be registered on HTTPS websites. Also, the location of the service worker script used during its registration must be from the same origin of the web application. However, the service worker script can further import and execute third-party scripts within its context. Since service workers are registered to offer functionality to a set of same-origin pages, a site with multiple subdomains may actually need to register multiple service workers (one for each origin of its subdomains).

[1]An origin is a triple including protocol, domain, and port. It operates as the standard security boundary in web browsers, by virtue of the Same Origin Policy (SOP).

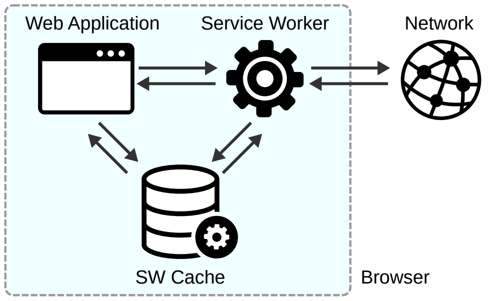

Figure 1: Service workers sit between web applications and web servers. They can store HTTP content in the cache and serve them to web pages, e.g., to speedup the navigation.

B. Caching Content in Service Workers

Using the Fetch API, a service worker can intercept all HTTP requests made by the web application it was registered for, and modify responses with arbitrary content. When an HTTP response is obtained from the network, a service worker can cache its content by means of the Cache API, as shown in Figure 1. Therefore, the next time a request for the same resource is made, the service worker can serve the cached response immediately, instead of – or before – fetching it over the network again. More specifically, the `CacheStorage` interface provides a global directory of all the named caches in an origin. Each cache name is mapped to a `Cache` object that represents a list of cached requests/responses.

A typical use case for the Cache API, in combination with the Fetch API, is shown in Listing 1. This strategy, usually referred to as *cache first* [26], is used to minimize network requests and to provide offline capabilities. Intuitively, the service worker intercepts all the HTTP requests (Line 1) and checks whether a corresponding response is already present in a cache (Line 3). If this is the case, the cached response is used (Line 4), otherwise the resource is first fetched from the remote server (Line 5), then added to the cache (Line 7), and served to the web application (Line 8). Notice that if the HTTP response is accessible from the service worker (same-origin or CORS-compliant cross-origin [23]), it can also be arbitrarily modified before caching.

III. Abusing the Cache API

In this section, we present an attack that can affect web applications registering a service worker making use of the Cache API. We first define our threat model and identify the preconditions for the attack, then we explain in detail how the attack works. We discuss its security implications in Section IV and present the results of a large-scale measurement of potential security issues found in the wild in Section V.

```
self.addEventListener('fetch', (event) => {
  event.respondWith(async function() {
    cResponse = await caches.match(event.request);
    if(cResponse) return cResponse;
    response = await fetch(event.request);
    cache = await caches.open('static');
    cache.put(event.request, response.clone());
    return response;
  }());
});
```

Listing 1: Service worker implementing a cache first strategy.

```
  (async () => {
    cache = await caches.open('v1');
    res = await cache.match('/secret.json');
    fetch('https://attacker.com/sniffer', {
      method: "POST",
      headers: {
        "content-type": "application/json"
      },
      body: await res.json()
    });
  }) ();
```

Listing 2: Leaking secrets from the cache. The attacker extracts from the cache the response containing the secrets, then exfiltrates them to a malicious server.

A. Threat Model and Attack Preconditions

We consider a traditional *web attacker* operating a malicious website at attacker.com. The attack we present here is enabled by the following conditions:

1) The attacker has active scripting capabilities on a page p of the target website. For example, this might happen when p suffers from an injection vulnerability and does not correctly deploy Content Security Policy (CSP) to mitigate the potential XSS. This is a reasonable assumption, considering that XSS is still one of the most prevalent web security vulnerabilities and is routinely found even on high-profile websites [13].
2) There exists a service worker using the Cache API registered on the same origin of the page p. This means that the attacker has active scripting capabilities in the same origin where the service worker was registered.

B. Attack Description

The attack is enabled by the fact that the service worker cache is accessible from scripts running in the same origin where the service worker was registered. If the attack preconditions are met, the attacker can abuse the Cache API to get unrestricted read and write access to all the HTTP responses in the cache served from the same origin of the service worker (or from CORS-compliant third-party origins). This means that the attacker can operate as a *person-in-the-middle* against cached HTTP responses, going beyond the traditional capabilities of a web attacker. In particular, the attacker can exfiltrate secrets from the cache or arbitrarily corrupt the cached content before it is served to the target web application.

Secret exfiltration can be performed by abusing the `match` method of the Cache API, which grants access to the response object bound to an arbitrarily chosen request. We exemplify the attack at work in Listing 2. Intuitively, a secret in the response to /secret.json is stored in the cache named *v1*. Hence, the attacker opens the cache *v1*, reads /secret.json and sends it to an attacker's controlled endpoint at https://attacker.com/sniffer by using the `fetch` API. Note that reading cached content is subject to the SOP. Here, the resource /secret.json comes from the same origin as the attacked site, so the attacker can read the secret data.

Arbitrary corruption of the cache content can be done by abusing the `put` method of the `Cache` interface, which allows the attacker to set an arbitrary response object to a cache entry bound to any chosen request. We exemplify the attack at work in Listing 3. In this example, the attacker reads the cached /login.html page, modifies its content in order to inject a keylogger from https://attacker.com, then writes back the modified content in the cache by taking care of stripping out potential security headers from the response. When the user navigates this page, the attacker payload will record and exfiltrate the user login credentials. Moreover, given that the attacker script is injected first in the page, as discussed in the next section, it can break defensive programming practices that rely on the order in which scripts are loaded [6].

```
(async () => {
  cache = await caches.open('v1');
  originalContent = await cache.match('/login.html');
  cParser = document.createElement('html');
  cParser.innerHTML = originalContent
  cParser.getElementsByTagName('head')[0].prepend(
    '<script src="https://attacker.com/keylogger.js"></
       script>');
  await cache.put("/login.html",
    new Response(cParser.outerHTML, {
      status: 200,
      statusText: "OK",
      headers: {
        "content-type": "text/html"
      }
    })
  );
}) ();
```

Listing 3: Corrupting the cache. A keylogger is injected in the cached page login.html and all security headers are stripped out of the tampered response.

IV. Security Implications

In our scenario, we require the attacker to be able to get active scripting capabilities on a page of the target web application. When this happens, the protection offered by SOP is already bypassed, and most security guarantees are voided anyway. However, the presented abuses of the Cache API are particularly dangerous, as they *exacerbate* the threats of traditional XSS.

A. Comparison with Traditional XSS

To understand the threats caused by the presented attack, we start by comparing it against traditional XSS. The first observation we make is that the attack by itself is extremely powerful since it breaks the confidentiality and the integrity of cached content. Technically speaking, the security implications of tampering with cached content are reminiscent of *persistent* *client-side XSS* [14], [35]. In this attack, client-side data like cookies and web storage are corrupted so as to lead to script injection every time the web application is accessed from the same client. As already reported by Vela [37] in 2015 (see Section VII), corrupting the service workers cache similarly gives the attacker the possibility of achieving persistent script injection capabilities. Even an ephemeral attack like reflected XSS can be turned into a persistent client-side XSS, which is triggered every time the victim visits a page fetching content from the cache using the same client where the reflected XSS took place. Moreover, the attacker could mount a persistent denial-of-service attack by manipulating cache content to affect the website functionality.

Note that injections on client-side storage are also dangerous because data therein might be used across multiple pages and sessions (within the same origin), which *amplifies* the attack surface against the web application. For instance, a simple reflected XSS on an error page, where the site operator inadvertently forgot to deploy appropriate CSP headers, might turn into an attack against the cached copy of the login page, where the attacker could inject a script to leak the victim's password. Furthermore, the attacker could exfiltrate sensitive resources cached during an authenticated session, even if the XSS occurs when the user does not hold an active session with the website anymore. In fact, the attacker could access leftover secrets from a previous session, such as personally identifiable information, passwords, security tokens, and multimedia content, just by reading them directly from the cache.

Besides all these threats, there is a distinctive feature of this attack that uniquely exacerbates the dangers of traditional XSS, i.e., the ability of the attacker to perform person-in-the-middle attacks against cached HTTP responses. This opens up novel attack scenarios, e.g., the attacker can bypass the protection of security headers and defensive programming practices, which normally mitigate the impact of a successful XSS exploitation. We discuss this below.

B. Bypassing Security Headers

By corrupting the service workers cache, the attacker does not just get access to the HTML of the cached pages, which they can already control upon XSS by interacting with the DOM, but to entire *response objects*. This means that the attacker can inspect and arbitrarily modify the content of HTTP headers, which is normally not possible unless the attacker can control HTTP traffic or exploit an HTTP response splitting vulnerability [29]. This is a major concern, especially given the increasing popularity of client-side security mechanisms based on HTTP headers. Prominent security headers such as `Content-Security-Policy`, `Feature-Policy`, and `X-Frame-Options` can be tampered with by an attacker to nullify their effect on protected webpages. We discuss concrete examples of attacks in the following. None of these attacks can be mounted via traditional XSS, since the attacker has no access to security headers in that scenario.

1) Content Security Policy: CSP was originally designed to mitigate the dangers of XSS by restricting script execution. Since the presented attack already requires active scripting capabilities to take place, one may think that the attacker has no gain in manipulating CSP headers. However, attackers might be prevented from executing malicious payloads injected into cached pages, whenever the corresponding response objects are protected by a CSP. In this case, attackers must also strip or modify the `Content-Security-Policy` header from the cached response to allow for the inclusion of the malicious content. Furthermore, CSP has evolved to support many more use cases [32], which can be targeted by the attacker. We discuss selected examples:

- Let us assume a page sets the CSP directive `frame-ancestors` to `'none'` to prevent framing on any page. An attacker abusing XSS could strip away the CSP directive from the cached copy of the page to void the protection enforced by the security header. A similar attack is discussed below for the `X-Frame-Options` header.
- The CSP `sandbox` directive, configured without the option `allow-same-origin`, can be deployed on a page to isolate it from other pages by creating a unique origin. Since CSP-sandboxed pages can still be cached, the attacker can tamper with cached content to inject malicious content in normally sandboxed contexts.
- CSP can be used to monitor security violations to a given policy without actually enforcing it, e.g., to test compliance with a policy without breaking functionality. If a policy is not enforced but only monitored, attackers abusing XSS could remove the policy from cached pages to hide the presence of policy violations and go incognito in their attack attempts.

2) Feature Policy: The *Feature Policy* [11], very recently renamed to *Permission Policy*, is a relatively new security mechanism that allows one to control which features are enabled on a page and in embedded frames. Policy directives set via the `Feature-Policy` header are defined as a combination of a feature name and a list of origins that can use that feature. Feature Policy is typically used to selectively disable security critical APIs (such as *Media Capture and Streams* [16] or the *Generic Sensor API* [38]) to prevent abuses. Although the activation of some of these features requires the explicit permission of the user, e.g., by clicking on a popup, disabling the `Feature-Policy` header may have severe security and privacy consequences, given that the attacker can potentially escalate privileges to, e.g., get control of webcams, microphones or other devices.

3) X-Frame-Options: The `X-Frame-Options` header, even though deprecated by the `frame-ancestors` directive of CSP, is still widely used for framing control in order to fight *UI Redressing* attacks [33], [15] and to mitigate certain classes of *XS-Leaks* [36]. For example, by setting its value to `DENY`, a webpage is normally ensured that it will not be framed, not even by same-origin pages. Removing this security header disables the protection and, interestingly, allows an attacker to also embed the cached page on an arbitrary origin. This is a peculiarity of our attack scenario, given that any request in the scope of the service worker is intercepted, even if it is caused by the inclusion of an iframe from a cross-origin position.

Discussion: The previous list of threats is not intended to be exhaustive, yet it is worth mentioning a few notable exclusions. The `Set-Cookie` header is not strictly speaking a security header, yet it contains security attributes for cookies, hence the attacker might profit from tampering with it; however, this header is not accessible from the Fetch API, which defines a list of forbidden response header names [5]. Moreover, we experimentally observed that HSTS headers of cached responses cannot be modified in major browsers, including Google Chrome and Mozilla Firefox. All the attacks discussed in the present section have been confirmed to work correctly by implementing appropriate proofs of concept.

C. Bypassing Defensive Programming

By tampering with cached response objects, the attack also allows one to bypass defensive programming [9]. In particular, popular APIs and coding conventions used to improve the security and the robustness of JavaScript code can be circumvented. Examples of such practices include:

- *Frozen objects* obtained by calling the `Object.freeze` method are immutable objects which can no longer be changed. This method prevents adding or removing properties, changing property values, or altering the object's prototype.
- *Sealed objects*, as created by the `Object.seal` method, are a weaker variant of frozen objects. The main difference with respect to frozen objects is that changing the values of existing properties is still possible.
- Other methods, such as `Object.preventExtensions` or even the descriptors of `Object.defineProperty` method, can serve to protect sensitive JavaScript objects from manipulations by malicious code.

By redefining methods like `Object.freeze` in cached HTTP responses, an attacker can void all the protections intended to defend sensitive objects. This cannot be achieved by traditional XSS, since the presented techniques build on language features specifically designed to constrain the execution of JavaScript code. The reason why this attack sidesteps such protection mechanisms is its privileged person-in-the-middle position on cached responses, which allows it to arbitrarily modify code before it is executed. This implies that attacker-controlled scripts run first and can perform damage before defensive programming practices are enabled.

We show the attack in Listing 4, where `Object.freeze` is used as a hardening measure against *prototype pollution* [8], [18]. This dangerous class of attacks refers to the ability of an attacker, under certain circumstances, to overwrite the properties of an object to execute malicious code once the tampered property is used by the application. The code listed between lines 2 and 5 represents a standard object creation procedure in JavaScript taking advantage of `Object.freeze` to prevent the prototype of the object from being modified.

```
1  Object.freeze = (x) => x; // Cache-based injection
2  let objProto = Object.freeze({
3    foo() { console.log('foo'); }
4  });
5  let obj = Object.create(objProto);
6  obj.__proto__.foo = () => alert("XSS"); // Injection
7  obj.foo();
```

Listing 4: Sidestepping protections against prototype pollution.

Therefore, the injection at line 6 has no effect on the prototype of the object, and the attacker payload is not executed. Conversely, an attacker with the ability to run code at the beginning of the snippet (Line 1) can redefine `Object.freeze` with an arbitrary function and disable the intended protection mechanism. As a result, the prototype of the object is modified by the injection at line 5, and the attacker's payload is executed.

D. Examples

In the rest of the paper, we report on the results of a large-scale measurement in the wild of the dangers of the Cache API in combination with service workers. To better motivate our study, we provide in Appendix A examples of real-world attacks enabled by abuses of the Cache API that we found on prominent sites which, in presence of an XSS vulnerability, would satisfy our attack preconditions. All the reported vulnerabilities have been tested in an ethical manner: XSS execution on a page was simulated by injecting a script through a browser extension called TamperCache, without attempting to exploit cross-site scripting vulnerabilities on the analyzed websites. The code used to mount the attack is directly adapted from Listing 3. Moreover, we developed a mock web application called SafeNotes to showcase the attack at work and help understanding its security implications. We make both TamperCache and SafeNotes publicly available, also providing short videos of attack simulations.[2]

V. Web Measurement

We perform an empirical study to assess the deployment of service workers and identify websites that fall into our threat model, because they use the Cache API and do not deploy CSP to mitigate XSS. We first explain how we evaluated the CSP deployment on the crawled sites and present the data collection methodology. Then, we report on the security impact of the presented attack in the wild.

A. Assessing CSP Deployment

To clarify the minimum requirements that a CSP should satisfy to mitigate XSS, we introduce a definition of *safe* CSP inspired by prior work [10], yet adapted to the latest version of CSP. The definition provides a baseline to evaluate the robustness of policies, meaning that a CSP that is not safe can be trivially bypassed. Nevertheless, a safe CSP does not ensure protection against the full spectrum of bypasses, including script gadgets [21] and the presence of allow-listed JSONP endpoints.

[2]Companion site: https://swcacheattack.secpriv.wien/

Definition (Safe CSP)**.** A CSP is *safe* iff it contains a `script-src` directive (or a `default-src` directive in its absence) bound to a value v satisfying these conditions:

1) v does not contain the `'unsafe-inline'` keyword, unless nonces or hashes are also present in v;
2) v does not contain the wildcard `*` or any of the `http:`, `https:` and `data:` schemes, unless the `'strict-dynamic'` keyword is also present in v.

Intuitively, clause 1 prevents XSS attacks based on the injection of inline scripts by means of `<script>` tags, event handlers, and `javascript:` URLs. Clause 2, instead, ensures that `<script>` tags are subject to meaningful restrictions on what they can load, e.g., it prevents script inclusion from arbitrary HTTPS websites. The side-conditions of the "unless" form account for subtleties in the CSP semantics, coming from the existence of multiple CSP versions with backward-compatible syntax. We assume here the adoption of a modern browser supporting the latest version of CSP (Level 3).

B. Data Collection

Our data collection procedure consists of 3 phases. First, we use a browser automation framework for Chrome to crawl websites and to identify those registering service workers. Then, we extend the code of service workers with Mitmproxy [3] to track access to the Cache API. Finally, for those sites which are potentially vulnerable because they do not deploy CSP correctly, we perform a controlled experiment where we simulate an XSS attack abusing the Cache API. This allows us to identify those cases that would be exploitable upon XSS.

1) Websites Deploying Service Workers: We consider the top-ranked sites (domains) according to the Tranco list [31], as well as subdomains found on the Bing search engine. We visit the homepage of each origin to extract links related to the website. We combine those links with those found on Bing. We group the links per origin, then we randomly select and navigate up to 50 links per origin. In the end, we isolate origins deploying service workers and monitor them.

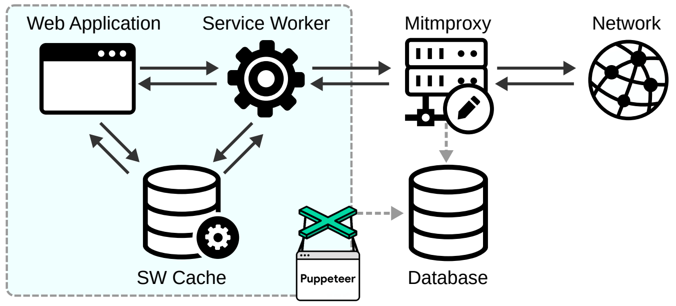

Figure 2: Automation framework to record accesses to the Service Worker Cache.

2) Monitoring Service Workers: The monitoring methodology is summarized in Figure 2. For each origin deploying service workers, we use Puppeteer [4] with Chrome v79 to simulate a navigation under the scope of the service worker, with the browser configured to redirect all HTTP(S) communications to a local Mitmproxy instance. Mitmproxy is an interactive HTTPS proxy for intercepting and manipulating HTTP requests and responses [3]. In our case, we are interested in intercepting and modifying service workers code. More specifically, we instruct Mitmproxy to prepend our own JavaScript code (referred to as the *service worker monitor*) to the original code of the intercepted service workers. Our monitor runs first within the service worker context and records all the operations performed by the original service worker (i.e., read/write to the Cache API). The monitor is largely implemented by means of the JavaScript Proxy API [25], which allows one to passively watch other JavaScript APIs without changing their original semantics. By doing so, we fully preserve the original functionality of service workers, while still effectively monitoring the different operations they perform, i.e., cache accesses.

In addition to monitoring the cache accesses from service workers contexts, we also monitor cache accesses from web pages. This is specifically done to support the discussion on the countermeasures we propose (see Section VI).

3) Controlled Cache Attacks: For origins that register service workers making use of the Cache API without deploying CSP to mitigate XSS, we carry out a controlled experiment by simulating an XSS attack to assess the magnitude and impact of the threats reported in this work. The simulation consists of the following steps:

(i) We navigate a random selection of links under the scope of the service workers using the Cache API. This step is performed to preliminary fill the cache with content.
(ii) Then, we infect the cache by replacing all originally cached HTML and JavaScript content with our own content (*attacker-controlled content*) and by removing all security headers from originally cached HTML pages.
(iii) Once the cache is infected, we navigate the tampered HTML pages to load our injected content and the infected JavaScript files. If the service worker returns our attacker-controlled content, the attack is successful.
(iv) To measure persistence, we also uninstall and reinstall the service worker, and navigate again the previously infected pages. If the service worker still returns the attacker-controlled content, then we achieve persistence.

The attacks are performed in a controlled setting, using a methodology similar to the one used for the service workers monitoring (see Figure 2). Our sole goal is to probe the service workers, locally installed in our test environment, to check whether they would serve payloads that we carefully crafted to replace the original HTML and JavaScript content in the cache. The simulated attack happens locally, and no malicious payload is sent to remote servers.

There are a number of reasons for which our approach may not succeed in loading attacker-controlled content in tampered pages. By manually investigating a subset of websites, we noticed that some service workers are constantly refreshing the cached content, thus overwriting our tampered files. Furthermore, read operations on the cache may be performed exclusively after specific user actions, e.g., clicking a button on the page. This is a limitation of our navigation pattern that simply visits URLs without interacting with elements in the page. Despite these limitations, we observe that our strategy is quite effective in practice (see below).

C. Prevalence of the Attack

The experimental evaluation was performed between May 24 and June 18, 2020. We found service workers deployed on 9,153 origins from 6,709 (4.6%) websites out of 150,000 sites that we crawled from the Tranco List generated on 23 May 2020.[3] Among those, 3,436 sites (51.2%) make use of the Cache API in a service worker for storing and reading all sorts of content, including HTML pages, JavaScript files, AJAX data, images, stylesheets, etc. Out of the 3,436 websites registering a service worker using the Cache API, the very large majority of them (95.8%) contains at least one page which does not deploy a safe CSP, and hence is vulnerable to the threats described in this work if the attacker can find an injection therein. Only a few prominent websites such as Google News, Twitter, or Pinterest covered all their crawled pages with safe CSPs, appropriately protecting against content injection attacks. The other sites either did not use CSP, used it for other use cases than script content restriction, or failed to properly restrict scripts, e.g., due to the use of `'unsafe-inline'`. We also found a few websites deploying a combination of safe and unsafe CSPs on their pages [34].

Considering the controlled experiments, we were able to successfully mount the attack against 65% of the 2,796 sites caching HTML or JavaScript. This means that the related service workers blindly served the cached HTML or JavaScript content we have tampered with. This is an under-approximation of the sites where the attack would be possible in presence of XSS, due to the limitations of our automated testing strategy. We now provide more details about our findings.

D. Security Implications

We explained that websites vulnerable to the attack described in this work are already exposed to significant danger since the confidentiality and the integrity of cached content can be arbitrarily compromised, which leads to persistent client-side XSS. However, we also mentioned two new capabilities enabled by the attack, i.e., bypassing security headers and circumventing defensive programming practices, which we investigate below.

1) Bypassing Security Headers: Because security policies are deployed on web pages, we consider only the results of the attack against the 2,040 sites with service workers caching HTML resources. For those cases, by tampering with the cached HTML content, we successfully simulated the attack by injecting our own scripts and by stripping out security headers on 792 (38.8%) sites. This percentage is already significant, yet it provides an under-approximation of the potentially exploitable sites as previously discussed.

Figure 3 presents the security policies on the cached pages belonging to origins whose service workers have been

[3]https://tranco-list.eu/list/J32Y/full

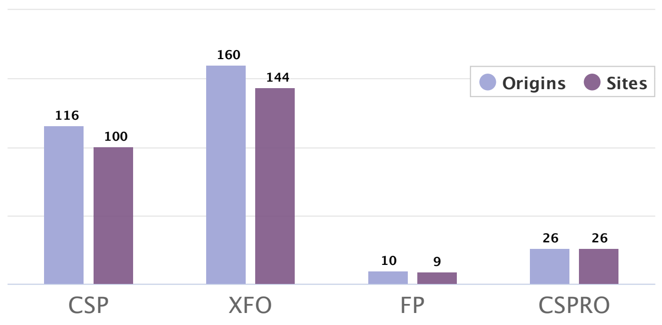

Figure 3: Security policies found on cached HTML content. These policies can be bypassed by altering cached responses, thereby voiding their effects on the security of the page. The most widespread policies include `Content-Security-Policy` (CSP), `X-Frame-Options` (XFO), `Feature-Policy` (FP) and `Content-Security-Policy-Report-Only` (CSPRO).

successfully attacked. Table I further breaks down the policies into a selection of their most used directives. We comment on these numbers in the following. It is important to note that the relatively low number of policies found in the cache of service workers reflects general statistics about the usage of those policies in the wild [32]. The more these security mechanisms gain traction, the more the practical significance of the presented attack increases.

CSP is primarily used for framing control, as shown by the prevalence of the `frame-ancestors` directive on 53 sites (6.7%). The popularity of XFO confirms that framing control is considered important by site operators: we found 137 sites (17.3%) using the header, out of which 108 only allow same-origin framing and 29 completely block framing. By stripping out these security headers, normally protected pages may become vulnerable to clickjacking and other frame-based attacks [33], [15]. We also found cases where CSP is deployed according to its original design, i.e., to mitigate script injection: we discussed how attackers can circumvent this protection on cached HTTP responses, so as to amplify the attack surface against the web application. Finally, we found a small number of sites caching HTML pages with a `Feature-Policy` header. This policy is very recent, which

Directive	Value	Sites	Origins
CSP	`script-src`	24	29
	`default-src`	27	28
	`frame-ancestors`	53	61
X-Frame-Options	`SAMEORIGIN`	108	116
	`DENY`	29	31
Feature-Policy	`gyroscope`	3	4
	`geolocation`	5	6
	`camera`	4	5
	`payment`	4	5
	`fullscreen`	3	4

Table I: Directives and Values of Cached Security Policies.

API	Sites	Origins
`freeze`	1,411	1,677
`seal`	895	1,026
`preventExtensions`	993	1,149
Total	**(90.5%) 1,472**	**(89.6%) 1,748**

Table II: Usage of defensive APIs on sites rendering tampered cached JavaScript content. Total represents the sites using at least one of the considered methods.

explains why it is not yet widespread among existing sites. Nonetheless, we have observed websites using it to restrict access to the camera or the geolocation, which are privacy-critical resources.

2) Bypassing Defensive Programming: Among the 2,148 sites caching JavaScript files, we found that 1,626 (75.7%) blindly serve JavaScript content that we have tampered with. Among those, 1,472 (90.5%) make use of at least one form of the defensive APIs shown in Table II.

As one can observe, websites are freezing objects in order to make them non-modifiable by bugs or potentially malicious scripts. Recall that freezing objects puts drastic restrictions on them since properties cannot be added, changed, or removed, likewise the prototype. Methods for sealing and preventing extensions of objects are also widely used and provide similar protections against sensitive objects. Since in our attack simulation we have successfully replaced the cached scripts with content that we controlled, an attacker in the same position could entirely bypass the protections of those APIs.

An assessment of what those defensive APIs are used for in the vulnerable websites would be extremely challenging and beyond the scope of this paper, since it would require the definition of a custom static analysis for JavaScript. Yet, the prevalence of the use of defensive APIs, coupled with an evaluation of service workers blindly serving attacker-controlled content, demonstrates the dangerous impact of the presented attack on defensive programming practices.

VI. Countermeasures

All the attacks discussed in this work are related to the fact that the service workers cache is accessible from the entire origin, and thus can be read or modified by potentially malicious scripts running in same-origin web pages. This issue could be prevented by making the Cache API inaccessible from other same-origin browsing contexts, i.e., by allowing only service workers to access the cache. This simple solution would effectively address all the attacks discussed in this work.

A. Caching Patterns from Web Pages

Clearly, such redesign of the service workers cache would introduce compatibility issues on existing web applications that are making use of the Cache API from web pages. As part of the automated large-scale analysis presented in Section V, we quantified the websites that access the cache outside of a service worker. For each of the origins that we found, we performed a manual assessment to better understand the access pattern for the cache and possibly identify other usages of the Cache API that we could not detect with the automated scan. This is the case, for instance, of amaro.com: the automated analysis and the manual navigation of the website only detected calls to `caches.keys()`, giving scarce information on its behavior. A manual assessment, instead, revealed that the site performs a cleanup of outdated cached content by deleting caches older than 30 days.

Overall, we identified 293 origins on 254 different sites that make use of the Cache API from page context. A significant part of these sites is due to a small number of companies that deployed the same web framework on multiple websites under their control. For instance, we discovered a network of 40 British online newspapers that are sharing the same cache access pattern as found on mirror.co.uk. Localized versions of airbnb.com count up to 39 websites, while the same portal used by developers.google.com has been found in other 18 websites hosted by Google, such as tensorflow.org and developer.android.com. Similarly, websites such as banggood.com have separate subdomains for each language, all of them sharing the same cache access pattern. This explains the discrepancy between the number of origins and sites.

On the one hand, we found several websites making use of the Cache API from page context that could be easily migrated to service worker context only. In particular, 49 websites have been found clearing the cache from a script instead of deleting outdated caches during the service worker activation phase [1]. On the other hand, we noticed legitimate use cases that require script access to complement the functionalities offered by service workers. We also investigated online web development resources to identify additional caching patterns of this kind and to provide a more systematic overview of the ones discovered in the wild.

We discuss below the most relevant caching patterns which would be affected by the proposed redesign of the Cache API. The following is by no means an exhaustive list of all possible use cases for the service workers cache in page context, but it provides evidence of the usefulness of this practice, and it motivates the relaxation mechanism in Section VI-B.

Cache on user interaction: This is a pattern described in [7] that does not require the presence of a service worker. It is useful to make specific resources available offline instead of caching the whole site. Users can be given a *save offline* button that, when clicked, fetches the resource from the network and adds it to the cache to make it available at a later time. It is worth noting that, since this flow does not make use of service workers, requests cannot be intercepted to provide a fallback page while the user is offline. Assuming that a page of the website is still open in the user's browser, a script running on that page must render the cached resources.

Cache then network: The idea of this pattern is to display the cached data first and then update the page when fresh content is retrieved from the network [7]. More specifically, at first a script in the page attempts to fetch a resource from the network. While the network request is being processed by the service worker, the requested resource is loaded from the cache, if available. In this case, the cached resource is rendered immediately by the script, while the page is updated with fresh content retrieved from the network at a later time. This is useful to render content that is frequently updated, as in the case of social media timelines or news feeds.

Cache on network response: Similarly to the *cache on user interaction* approach, this pattern does not necessarily require a service worker. Resources that are not found in the cache are fetched from the network and then cached. If they are requested again, then they are loaded from the cache to lower the rendering time and to avoid the same resource from being fetched from the network multiple times. This pattern is typically used to serve resources that are not frequently updated. For instance, we noticed that the popular Mapbox GL JS library [2] takes advantage of this approach to render interactive maps from vector tiles that are cached and drawn using WebGL.

B. Cache Access Relaxation

Out of 4,679 origins using the Cache API, only 6.3% have been found to access the cache from scripts in web pages. Although this percentage is low, we identified some legitimate cache patterns adopted by web applications that are not implemented via service workers. Hence, restricting the Cache API to service workers would lead to breakage in existing websites. To prevent the attacks described in this paper, while ensuring compatibility with existing web applications, we propose to 1) expose by default the Cache API exclusively to service workers, but 2) relax the scope of the cache to other same-origin contexts upon request. To this end, we envision a simple server-defined mechanism that allows site operators to instruct the browser to make the Cache API accessible from other same-origin contexts.

The mechanism could be implemented via a custom HTTP response header that loosens the Cache API restrictions on the origin of the requesting website. By incorporating this change in web browsers, the large majority of websites would be protected by default, while other sites that need to use the Cache API from page context could choose to opt in for the current behavior by adding the custom header on the affected pages. Notice that browsers already support an HTTP response header that directly affects the service workers cache: `Clear-Site-Data` mandates the browser to clear data associated with the requesting website, such as cookies, storage (including the service workers cache), and the browser cache.

Alternatively, the directive could land into one of the emerging mechanisms that are used to enforce server-provided security policies in supporting browsers. For instance, the Feature Policy [11] allows website operators to selectively disable certain features, e.g., to lock down their applications by avoiding security critical APIs from being abused, or to enable browser features and APIs that might be disabled by default. Interestingly, the Feature Policy does not only apply to the top-level page received from the origin that specified the security header, but it is also enforced on embedded content. For the Cache API, this would allow a malicious website to enable access to the cache on cross-origin pages which are framed by the attacker's website, thus voiding the protection of embedded origins not under the control of the attacker. This issue is specifically addressed by the newborn Document Policy [12], which is similar to the Feature Policy, but does not automatically propagate to embedded browsing contexts. A document served with a `Document-Policy` HTTP header may embed other same or cross-origin documents, but embedded documents are not affected by the parent's policy, unless they explicitly comply with it. Although the Document Policy is perfectly tailored to include our proposal, this mechanism is still under development and only supported by Google Chrome. In this regard, we did not provide any specific implementation details for our approach at the moment. Instead, we outlined possible directions that browser vendors and web standards developers could follow to mitigate the attack discussed in this work.

Security Considerations: The proposed redesign of the Cache API would transparently protect all the websites that are accessing the cache exclusively from service workers (93.7% of 4,679 origins, covering 93.1% of the 3,537 websites using the Cache API). It is worth noting that our attack scenario requires the target origin to register a service worker which interacts with the Cache API. For this reason, origins accessing the cache exclusively from page context (2.6% of the origins) are protected by design against the attack and could benefit from the relaxation mechanism without suffering from additional security issues. The remaining part of the origins which have been found to perform mixed access to the cache from service workers and other same-origin contexts (3.6% of the origins) should enable the relaxation mechanism with care and deploy other countermeasures against XSS attacks, such as a strict CSP [39] or Trusted Types [19].

C. Self-protecting Service Workers

Besides the Cache API redesign discussed in this section, we point out a simple countermeasure that web developers can adopt to protect their websites against malicious cache modifications from the page context. As explained in Section III, corrupting the cache implies the creation of a synthetic `Response` object that replaces the original entry in the service worker's cache. Since the `Response` constructor does not allow to instantiate the `url` attribute with an arbitrary value [24], the `url` attribute is set to the empty string. Therefore, a service worker could put a restriction before rendering cached responses, such that on matched cache objects, the URL of the response in the cache is compared against the URL of the request: the cached response is rendered if the URLs are equivalent, otherwise the service worker discards the cached entry and fetches the resource from the network. A practical demonstration of this programming practice is available on our companion website.

This simple solution has already been proposed in the past,[4] but our experimental evaluation shows that this practice has not

[4]See https://github.com/w3c/ServiceWorker/issues/698

been adopted at scale (see Section V). Although this approach is effective against tampering from the page context, it does not prevent malicious scripts from violating the confidentiality of cached contents. Furthermore, this practice is incompatible with websites using caching patterns based on synthetic responses.

VII. Related Work

The Cache API has been reported as a potential attack vector by Vela [37] in a blog post soon after the introduction of service workers in 2015. At the time of submission of the present paper, we were not aware of that research which was pointed out by one of the anonymous reviewers. Although Vela described a cache pollution attack that is reminiscent of persistent client-side XSS, he did not discuss the exclusive capabilities that are enabled by this attack vector, i.e., bypassing security headers and defensive programming practices (see Section IV). Compared to our work, Vela also did not perform an evaluation of the threat at scale and did not propose a countermeasure to the attack. Instead, he investigated an instance of the attack where persistent XSS execution can be obtained on targets caching the response of open redirectors. This specific attack has been mitigated by restricting redirected responses inside service workers.[5] However, the design of the Cache API remained unchanged, leaving it vulnerable to the security issues discussed in this paper.

Since then, there has been little research around service workers. A recent paper by Steffens et al. [35] quantified the prevalence of persistent client-side XSS in the wild by taint tracking. However, their work did not consider the Cache API as a possible sink and, remarkably, even proposed the use of service workers to secure problematic programming patterns related to caching. Unfortunately, our analysis shows that the use of the Cache API in service workers is not a silver bullet against XSS and might even exacerbate its impact in specific cases.

The first and most notable security analysis of service workers is the one by Lee et al. [20]. Their work extensively focused on the Push Notification API, which they have shown could be used to mount phishing attacks. They also abused the lifespan of service workers to mine cryptocurrencies (Monero), and used push messages to distribute transactions. Finally, the paper considered the Cache API in a privacy setting to discover whether a victim visited a target PWA from a pool. To do so, the victim is tricked into visiting an attacker-controlled PWA that, when the victim is offline, opens multiple iframes whose sources are the URLs of the target PWAs. Frames whose content successfully loads while the victim is offline, identify PWAs which have been visited by the victim in the past. This attack has been recently improved by Karami et al. [17], who presented techniques based on timing information to probe whether specific resources have been cached by the victim's browser. These techniques rely on realistic assumptions, i.e., the victim is not required to be offline, and enable to infer sensitive application-level information.

[5]See https://crbug.com/669363

Another work on the security implications of service workers by Papadopoulos et al. [30] demonstrated a sophisticated resource abuse scenario, where a remote entity makes use of current web technologies to perform harmful computations and operations, such as mining cryptocurrencies. The authors leveraged the fact that the life cycle of service workers is not tied to that of webpages to demonstrate how to turn browsers into bots, remotely controlled by the attacker. In particular, they leveraged the Push and Sync APIs to keep the service workers alive, so that computations can continue until the user closes the browser. However, there is no mention of particular attacks on the Cache API of service workers.

VIII. Conclusion

In PWAs, service workers make use of the Cache API in order to improve the responsiveness of web applications and possibly offer an offline browsing experience when the network is unavailable. In this work, we showed that since the cache is accessible from the entire origin of a service worker, malicious scripts running in web pages can achieve person-in-the-middle capabilities against cached content. This allows an attacker with scripting capabilities to mount an attack reminiscent of persistent client-side XSS, and yet more powerful. In particular, the person-in-the-middle position of the attacker exacerbates the threats of traditional XSS to a new level since the attacker obtains the ability to bypass security headers and circumvent defensive programming practices. These vulnerabilities represent a new class of attacks that are only possible with service workers caching content. We performed an empirical study on 150,000 sites from the Tranco list and found that the large majority of the sites which register a service worker using the Cache API are vulnerable to the presented attack, as long as it is possible to identify an XSS vulnerability on them. As a countermeasure, we suggested a simple redesign of the Cache API to avoid exposing it to scripts and same-origin contexts other than service workers. We quantified that the majority of sites would get immediate security benefits from this change and proposed a relaxation mechanism to ensure backward compatibility with websites which deliberately accept the security risks, or mitigate them by deploying appropriate XSS countermeasures.

Acknowledgments

We thank Dolière Francis Somé (CISPA Helmholtz Center for Information Security) for contributing to a previous version of this work. We also thank the anonymous reviewers for their helpful suggestions and for pointing out missing related works that have been discussed to improve this paper. This work has been partially supported by the European Research Council (ERC) under the European Union's Horizon 2020 research (grant agreement 771527-BROWSEC); by the Austrian Science Fund (FWF) through the project PROFET (grant agreement P31621); by the Austrian Research Promotion Agency (FFG) through the Bridge-1 project PR4DLT (grant agreement 13808694) and the COMET K1 SBA.

References

[1] "Caching Files with Service Worker - Google Developers," https://developers.google.com/web/ilt/pwa/caching-files-with-service-worker.

[2] "Mapbox gl javascript library," https://docs.mapbox.com/mapbox-gl-js/.

[3] "Mitmproxy - a free and open source interactive HTTPS proxy," https://mitmproxy.org/.

[4] "Puppeteer Automation Framework," https://pptr.dev/.

[5] "Fetch Living Standard," 2021, https://fetch.spec.whatwg.org/.

[6] D. Akhawe, P. Saxena, and D. Song, "Privilege separation in html5 applications," in *Proceedings of the 21st USENIX Security Symposium*, 2012.

[7] J. Archibald, "Google Web Fundamentals: The Offline Cookbook," 2019, https://developers.google.com/web/fundamentals/instant-and-offline/offline-cookbook.

[8] O. Arteau, "Prototype pollution attack," 2018, https://github.com/HoLyVieR/prototype-pollution-nsec18.

[9] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis, "Language-based defenses against untrusted browser origins," in *Proceedings of the 22th USENIX Security Symposium*, 2013.

[10] S. Calzavara, A. Rabitti, and M. Bugliesi, "Content security problems?: Evaluating the effectiveness of content security policy in the wild," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[11] I. Clelland, "Feature Policy, W3C Public Working Draft," 2019, https://www.w3.org/TR/feature-policy-1/.

[12] ——, "Document Policy, Editor's Draft," 2020, https://w3c.github.io/webappsec-feature-policy/document-policy.html.

[13] HackerOne, "The HackerOne Top 10 Most Impactful and Rewarded Vulnerability Types," 2020, https://www.hackerone.com/top-10-vulnerabilities.

[14] S. Hanna, R. Shin, D. Akhawe, A. Boehm, P. Saxena, and D. Song, "The emperor's new apis: On the (in) secure usage of new client-side primitives," in *Proceedings of the Web 2.0 Security and Privacy (W2SP)*, 2010.

[15] L. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson, "Clickjacking: Attacks and Defenses," in *Proceedings of the 21th USENIX Security Symposium*, 2012.

[16] C. Jennings, B. Aboba, J.-I. Bruaroey, H. Boström, and Y. Fablet, "Media Capture and Streams - W3C Candidate Recommendation," 2021, https://www.w3.org/TR/mediacapture-streams/.

[17] S. Karami, P. Ilia, and J. Polakis, "Awakening the web's sleeper agents: Misusing service workers for privacy leakage," in *28th Annual Network and Distributed System Security Symposium (NDSS)*, 2021.

[18] I. Koishybayev and A. Kapravelos, "Mininode: Reducing the Attack Surface of Node.js Applications," in *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[19] K. Kotowicz and M. West, "Trusted Types, Editor's Draft," 2020, https://w3c.github.io/webappsec-trusted-types/dist/spec/.

[20] J. Lee, H. Kim, J. Park, I. Shin, and S. Son, "Pride and prejudice in progressive web apps: Abusing native app-like features in web applications," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[21] S. Lekies, K. Kotowicz, S. Groß, E. A. V. Nava, and M. Johns, "Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[22] MDN web docs, "CacheStorage API," https://developer.mozilla.org/en-US/docs/Web/API/CacheStorage.

[23] ——, "Cross-Origin Resource Sharing (CORS)," https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

[24] ——, "Fetch API - Response," https://developer.mozilla.org/en-US/docs/Web/API/Response.

[25] ——, "JavaScript Proxy API," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy.

[26] ——, "Making PWAs work offline with Service workers," https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Offline_Service_workers.

[27] ——, "Progressive web apps (PWAs)," https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps.

[28] ——, "Service Workers," https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers.

[29] MITRE, "CWE-113: Improper neutralization of crlf sequences in http headers," https://cwe.mitre.org/data/definitions/113.html.

[30] P. Papadopoulos, P. Ilia, M. Polychronakis, E. P. Markatos, S. Ioannidis, and G. Vasiliadis, "Master of web puppets: Abusing web browsers for persistent and stealthy computation," in *26th Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[31] V. L. Pochat, T. V. Goethem, S. Tajalizadehkhoob, M. Korczynski, and W. Joosen, "Tranco: A research-oriented top sites ranking hardened against manipulation," in *26th Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[32] S. Roth, T. Barron, S. Calzavara, N. Nikiforakis, and B. Stock, "Complex security policy? a longitudinal analysis of deployed content security policies," in *27th Annual Network and Distributed System Security Symposium (NDSS)*, 2020.

[33] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson, "Busting frame busting: a study of clickjacking vulnerabilities at popular sites," *IEEE Oakland Web*, vol. 2, no. 6, 2010.

[34] D. F. Somé, N. Bielova, and T. Rezk, "On the content security policy violations due to the same-origin policy," in *Proceedings of the 26th International Conference on World Wide Web (WWW)*, 2017.

[35] M. Steffens, C. Rossow, M. Johns, and B. Stock, "Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild," in *26th Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[36] A. Sudhodanan, S. Khodayari, and J. Caballero, "Cross-origin state inference (COSI) attacks: Leaking web site states through xs-leaks," in *27th Annual Network and Distributed System Security Symposium (NDSS)*, 2020.

[37] E. Vela, "[service workers] new apis = new vulns = fun++," 2015, http://sirdarckcat.blogspot.com/2015/05/service-workers-new-apis-new-vulns-fun.html.

[38] R. Waldron, "Generic Sensor API - W3C Candidate Recommendation," 2019, https://www.w3.org/TR/generic-sensor/.

[39] L. Weichselbaum, "Mitigate cross-site scripting (XSS) with a strict Content Security Policy (CSP)," 2021, https://web.dev/strict-csp/.

Appendix

We discuss examples of potential attacks identified in real-world sites which do not deploy a safe CSP on all their pages. These attacks have been confirmed by simulating script injection via a browser extension, i.e., they would work in presence of an XSS vulnerability.

A. *Persistence and DoS*

The web client of WhatsApp (web.whatsapp.com) enforces a surprisingly weak CSP which allows for the execution of unsafe scripts in the page, due to the `unsafe-inline` directive and the lack of nonces or hashes. An excerpt of the policy is reported in Listing 5. Despite the extreme popularity of this web application, the risk of XSS is real, and the lack of appropriate mitigations could lead to vulnerabilities being exploited in practice.[6] We found that the service worker registered by the website stores a number of resources in the cache to increase the responsiveness of the application. Cached resources include icons, images, fonts, stylesheets, and scripts. The presence of scripts in the cache is particularly dangerous, given that an attacker could modify the code stored in the cache and persistently infect all the pages which are including them. As shown in Listing 6, we experimentally verified that tampering with one of the cached scripts causes our payload to be persistently included by the web application, even after closing and restarting the browser.

[6]https://www.perimeterx.com/tech-blog/2020/whatsapp-fs-read-vuln-disclosure/

```
default-src 'self' data: blob:;
script-src  data: blob: 'self' 'unsafe-eval'
            'unsafe-inline'
            https://ajax.googleapis.com
            https://api.search.live.net
            https://maps.googleapis.com
...
```

Listing 5: CSP cached on web.whatsapp.com.

```
caches.open("wa2.2025.6").then((cache) => {
  cache.put("https://web.whatsapp.com/app2.0
      edf20df3a4aa9395226.js",
    new Response("alert('CACHE-XSS');", {
      status: 200,
      statusText: "OK",
      headers: {
        'Content-Type': 'text/javascript'
      }
    })
  );
});
```

Listing 6: Attack on web.whatsapp.com. The modified script is included from the cache by the website, whichs persistently executes the malicious payload.

This allows an attacker to subvert the intended behavior of the web application for an indefinite amount of time, i.e., until the user manually clears the service worker cache or until the service worker script invalidates the cached resources due to a version upgrade. A well-orchestrated attack could silently monitor the entire activity of a WhatsApp user, circumventing end-to-end encryption to exfiltrate all the exchanged messages and the network of contacts of the user for weeks. Alternatively, the attacker could DoS the web application to divert the user towards other less secure communication platforms.

B. *Privilege Escalation*

The website computerbase.de makes use of a service worker to provide offline capabilities to its users. It stores a set of pages in the cache which are served to the browser when the user is offline. Listing 7 shows the Feature Policy set for the website, which is also attached to cached pages. This policy disallows all scripts running in the page and iframes from requesting access to critical APIs and devices. Unfortunately, an XSS on any of the pages in the site's origin would enable an attacker to arbitrarily modify the cached resources and strip away the `Feature-Policy` header, thereby removing the restrictions during offline navigation. This would cause the attacker to achieve persistent access to the webcam or the microphone when the user is offline, and potentially exfiltrate the captured data as soon as the user is online again. This can be done, for instance, by injecting a script that runs in the offline page which continuously attempts to upload the recording to an attacker's controlled endpoint.

C. *Framing Protection Bypass*

The hosting provider accuwebhosting.com adopts the cache first strategy outlined in Listing 1. To minimize network requests, the service worker on this website returns cached pages first, instead of loading fresh resources from the network. If the page is not available, it is loaded from the network and then cached. The CSP enforced by the website (Listing 8) is clearly not intended as a countermeasure against XSS attacks, but it prevents same and cross-origin framing. A malicious script could abuse the Cache API to entirely remove the `Content-Security-Policy` header from cached resources and disable the framing protection, clearing the way for UI Redressing attacks. Furthermore, framing resources from a same-origin position allows an attacker to amplify the scope of the vulnerability by accessing the DOM to exfiltrate secrets from framed pages, even if those pages are not cached.

```
camera 'none'; document-domain 'none';
geolocation 'none'; microphone 'none';
payment 'none'; sync-xhr 'none'
```

Listing 7: Feature Policy cached on computerbase.de.

```
frame-ancestors 'none'
```

Listing 8: CSP cached on accuwebhosting.com.