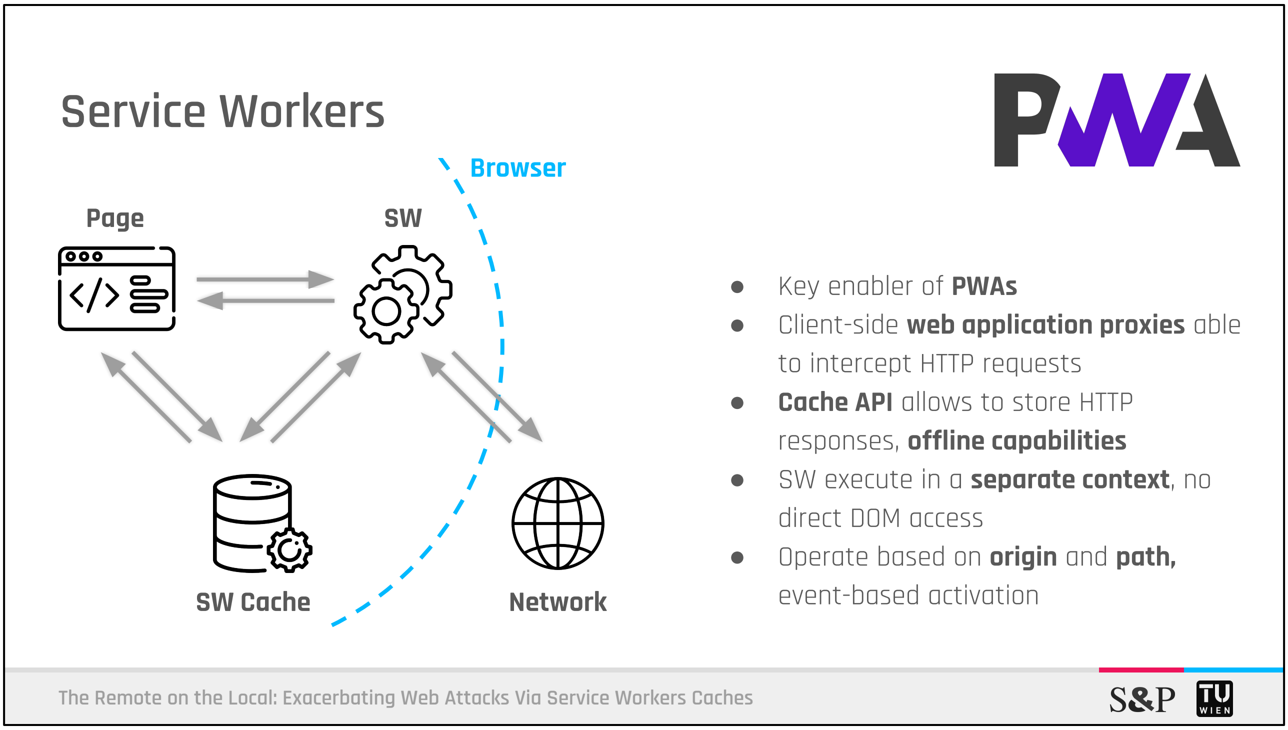
The Remote on the Local

Exacerbating Web Attacks Via Service Workers Caches

Marco Squarcina (TU Wien)
🐦 @blueminimal

15th IEEE Workshop on Offensive Technologies, May 27, 2021

Joint work with
Stefano Calzavara (Università Ca' Foscari Venezia & OWASP)
Matteo Maffei (TU Wien)



Service Workers

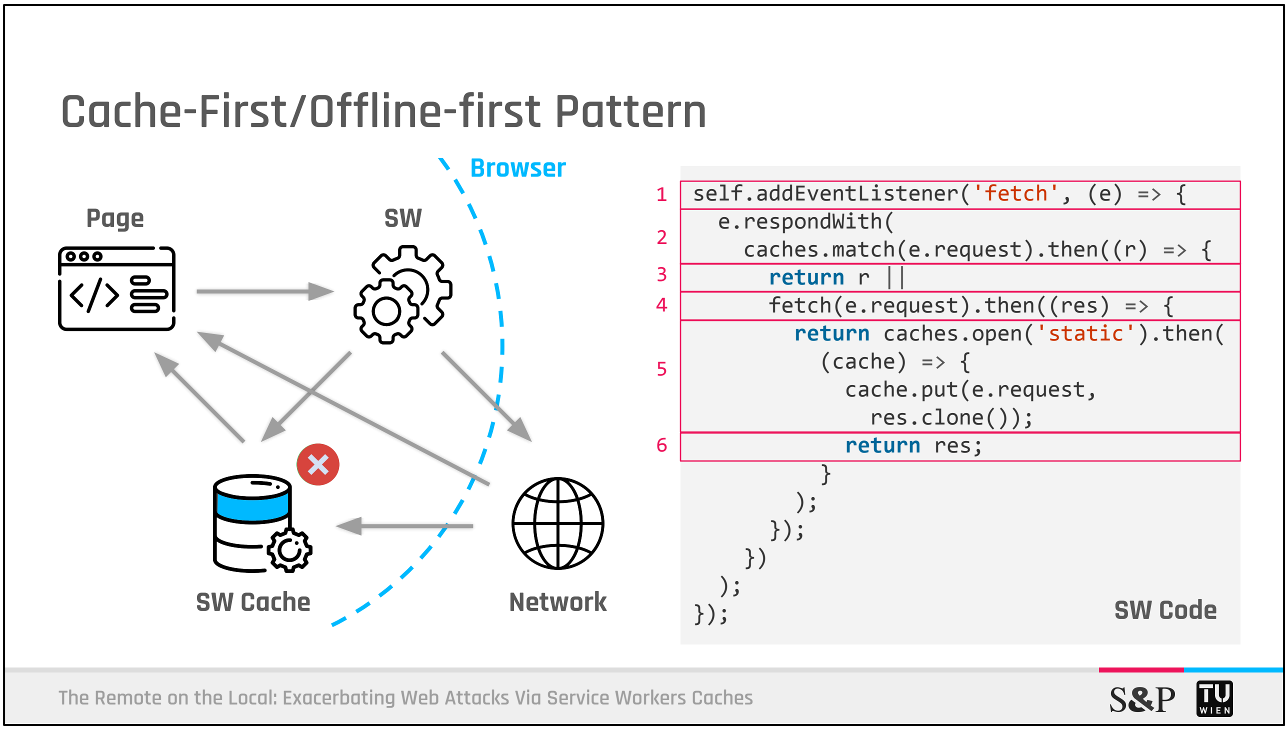
- Key enabler of **PWAs**
- Client-side **web application proxies** able to intercept HTTP requests
- **Cache API** allows to store HTTP responses, **offline capabilities**
- SW execute in a **separate context**, no direct DOM access
- Operate based on **origin** and **path,** event-based activation

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

Progressive Web Applications are the latest trend in the evolution of the Web. This is mostly due to the fact that they offer a user experience similar to traditional applications by providing responsiveness and by supporting offline usage.

Service Workers are the key enabler of PWAs, since they act as a client-side web application proxy. A Service Worker can indeed intercept all HTTP requests made by the web application and modify the responses with arbitrary content. Using the Cache API, Service Workers can also store HTTP responses and serve them at a later time.

Service Workers execute in a separate context from the JavaScript code running in the page. That means that they have no access to the DOM. They also are not allowed to access cookies via document.cookie and cannot call other synchronous APIs like localStorage or sessionStorage.



Cache-First/Offline-first Pattern

```
1  self.addEventListener('fetch', (e) => {
    e.respondWith(
2     caches.match(e.request).then((r) => {
3      return r ||
4      fetch(e.request).then((res) => {
        return caches.open('static').then(
5        (cache) => {
          cache.put(e.request,
           res.clone());
6         return res;
         }
        );
       });
      })
    );
   });
```

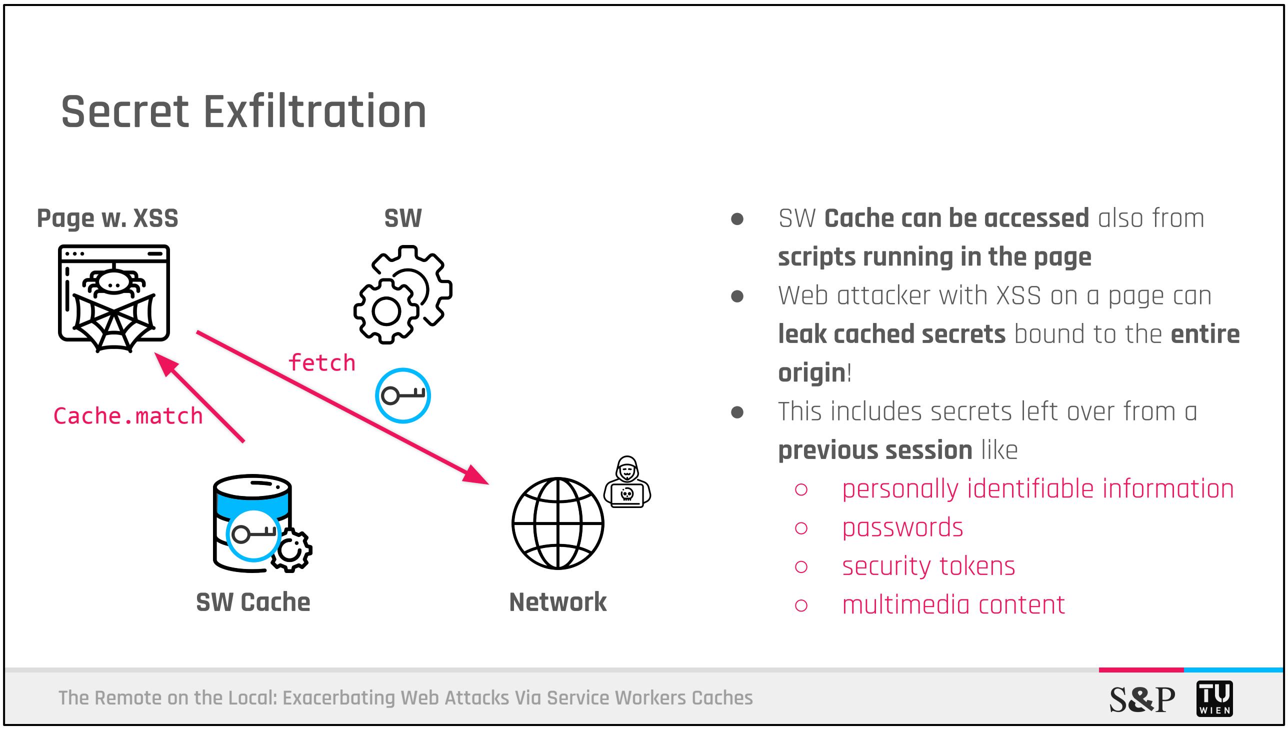
SW Code

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

S&P

To exemplify the intended usage of the Cache API, we provide a code snippet that implements a basic version of a pattern called cache-first that is used to minimize network traffic and provide offline capabilities.

1. The Service Workers intercepts all the requests by registering a listener on the "fetch" event
2. It checks if the corresponding resource is found in the cache using the "match" method
3. If this is the case, the cached response is used
4. Otherwise, the resource is fetched from the network
5. It is then added to the cache, first by opening a given cache and then by using the "put" method
6. Finally, it is served to the web application

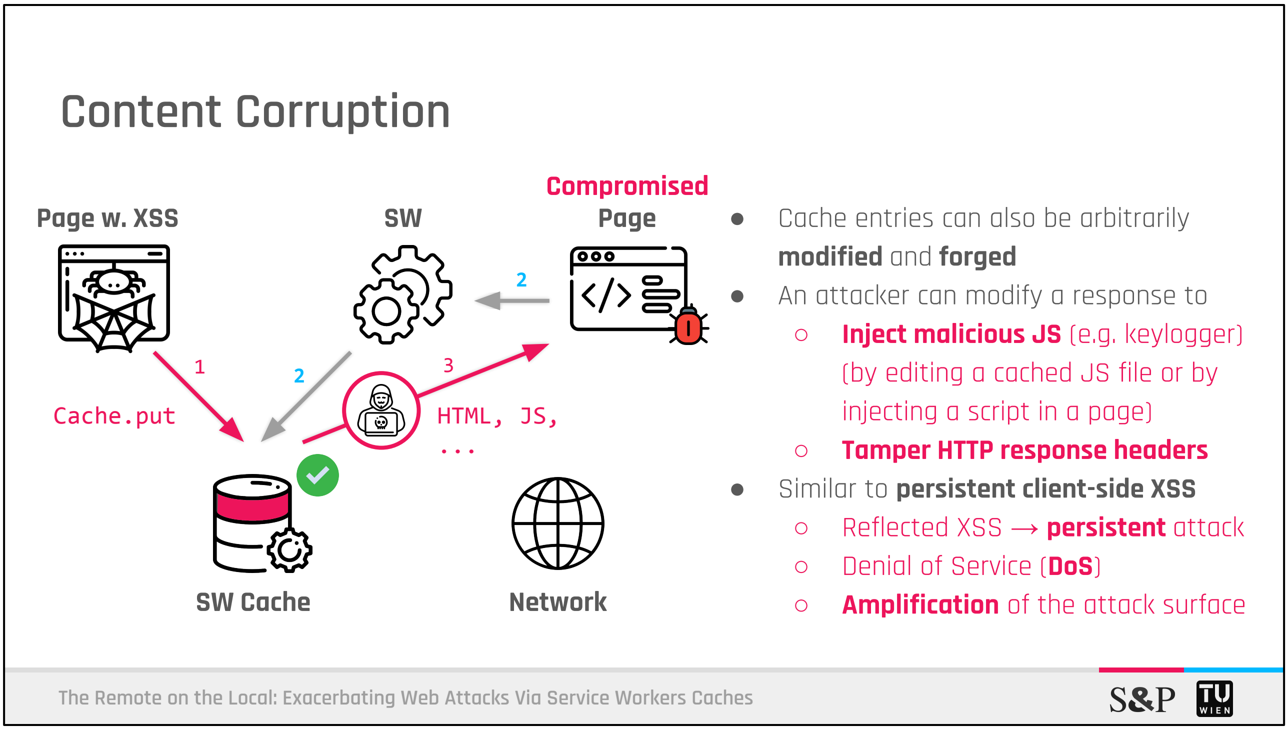


Secret Exfiltration

- SW **Cache can be accessed** also from **scripts running in the page**
- Web attacker with XSS on a page can **leak cached secrets** bound to the **entire origin**!
- This includes secrets left over from a **previous session** like
  - personally identifiable information
  - passwords
  - security tokens
  - multimedia content

Our attack is enabled by the fact the Service Worker Cache API is accessible from scripts running in the same origin where the Service Worker was registered. This means that an attacker with an XSS on a page can get arbitrary read and write access to the Cached content for the whole origin.

From security standpoint this means that an attacker can leak secrets from the cache, i.e., sensitive data left over from a previous authenticated session. This includes personal information, multimedia content, security tokens, passwords, and so on.



Content Corruption

- Cache entries can also be arbitrarily **modified** and **forged**
- An attacker can modify a response to
  - **Inject malicious JS** (e.g. keylogger) (by editing a cached JS file or by injecting a script in a page)
  - **Tamper HTTP response headers**
- Similar to **persistent client-side XSS**
  - Reflected XSS ⟶ **persistent** attack
  - Denial of Service (**DoS**)
  - **Amplification** of the attack surface

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

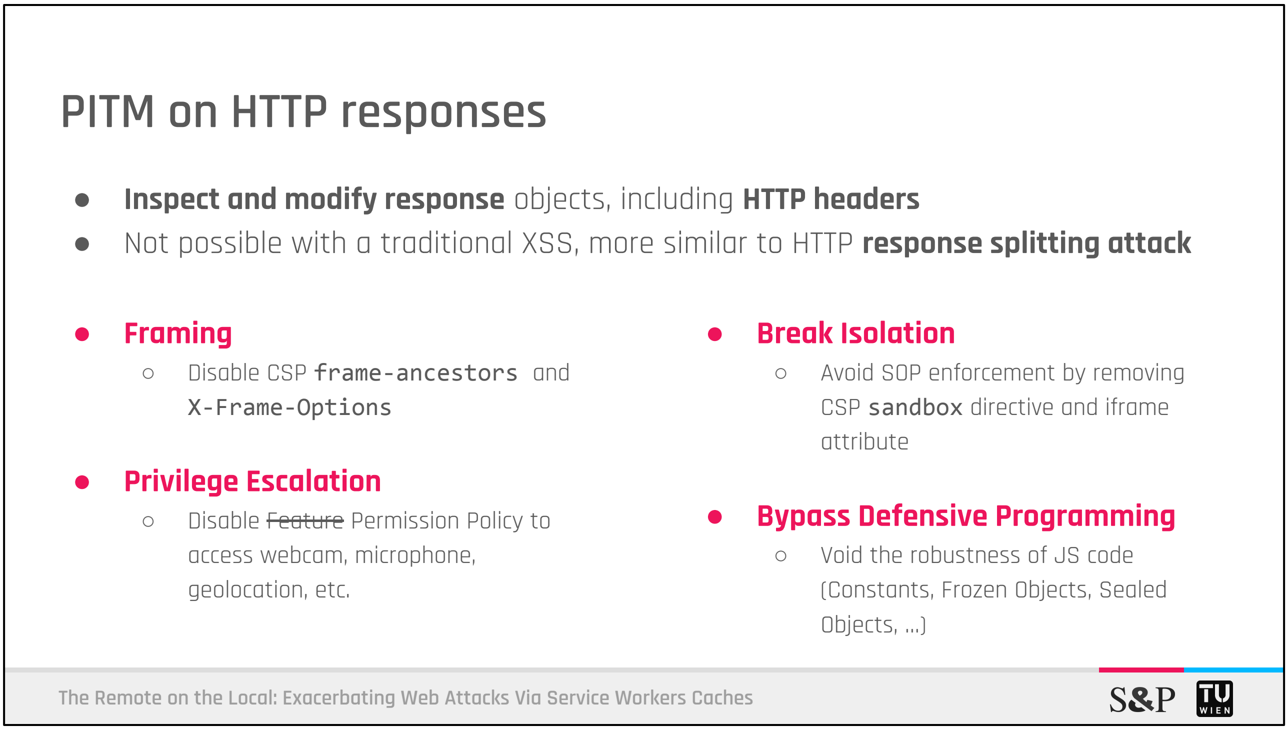
S&P

The attacker can also perform arbitrary corruption of the Cache content. In this example the attacker has a reflected XSS on a page, but wants to spread a malicious payload to other cached resources.

1. The attacker tampers with cached responses in order spread the injection of a malicious script, like a keylogger
2. Then, the victim performs an access to a legitimate page such as login.html, that happens to be in the cache
3. At this point, the Service Worker returns a page containing the keylogger, which exfiltrates the login credentials

This attack resembles a persistent client-side XSS, because it turns an ephemeral reflected XSS into a persistent attack. Notice that the Cache does not expire unless explicitly cleaned. Even if the Service Worker is reinstalled, the cache is not altered by default.

This technique can also be used to amplify the attack surface and to mount a DoS attack by preventing a user from accessing the website.



PITM on HTTP responses

- **Inspect and modify response** objects, including **HTTP headers**
- Not possible with a traditional XSS, more similar to HTTP **response splitting attack**

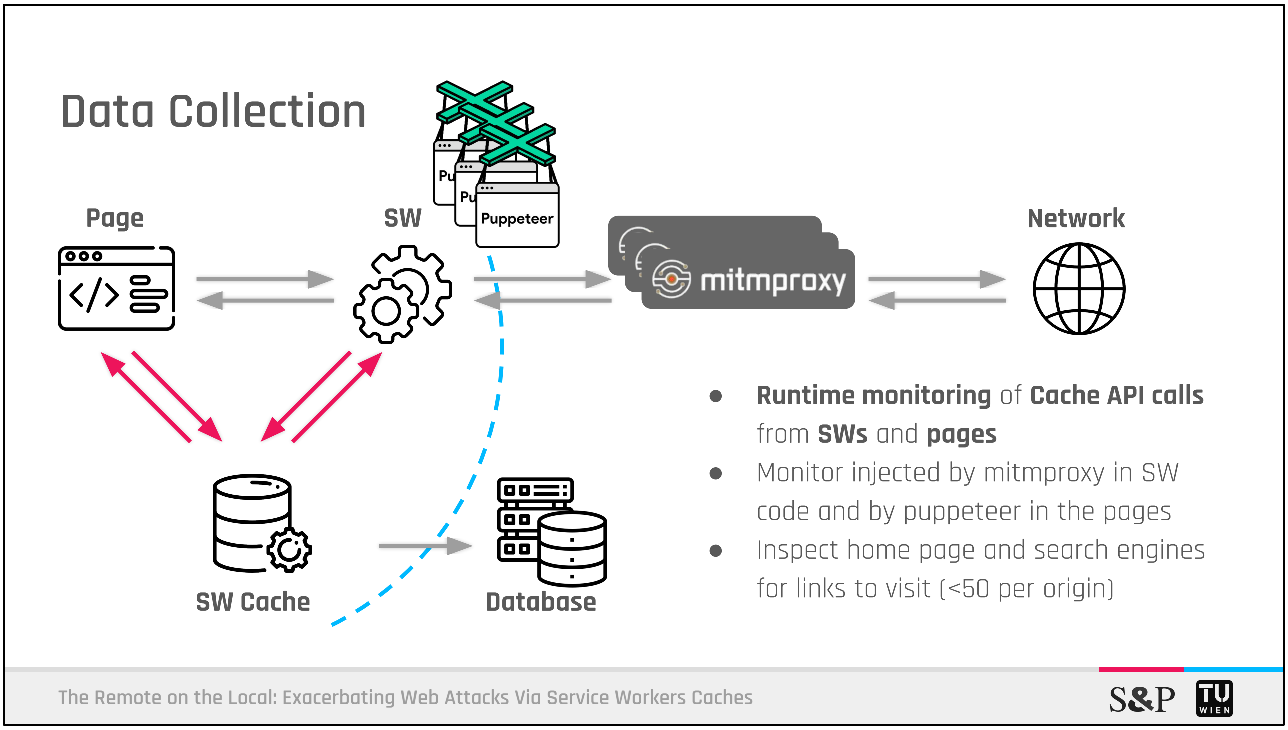
- **Framing**
  - Disable CSP `frame-ancestors` and `X-Frame-Options`
- **Privilege Escalation**
  - Disable ~~Feature~~ Permission Policy to access webcam, microphone, geolocation, etc.
- **Break Isolation**
  - Avoid SOP enforcement by removing CSP `sandbox` directive and iframe attribute
- **Bypass Defensive Programming**
  - Void the robustness of JS code (Constants, Frozen Objects, Sealed Objects, ...)

Other than tampering with the body of HTTP responses, an attacker can also inspect and manipulate the HTTP headers. This grants much more power than a persistent client-side XSS.

For example, by disabling the frame-ancestors directive of the Content-Security Policy or the X-Frame-Options header, an attacker could frame a page that was previously protected and mount attacks like clickjacking or exploit certain classes of xs-leaks.

Other examples include breaking the document's isolation by removing the "sandbox" directive from iframes and CSP.
Also, stripping out the Permission Policy header - in case it is enforced - gives the ability to access the webcam and the microphone of the user.

Furthermore, the injected script can be executed anywhere in the page. This enables the attacker to bypass defensive programming practices such as freezing objects, which is a standard protection against prototype pollution attacks.

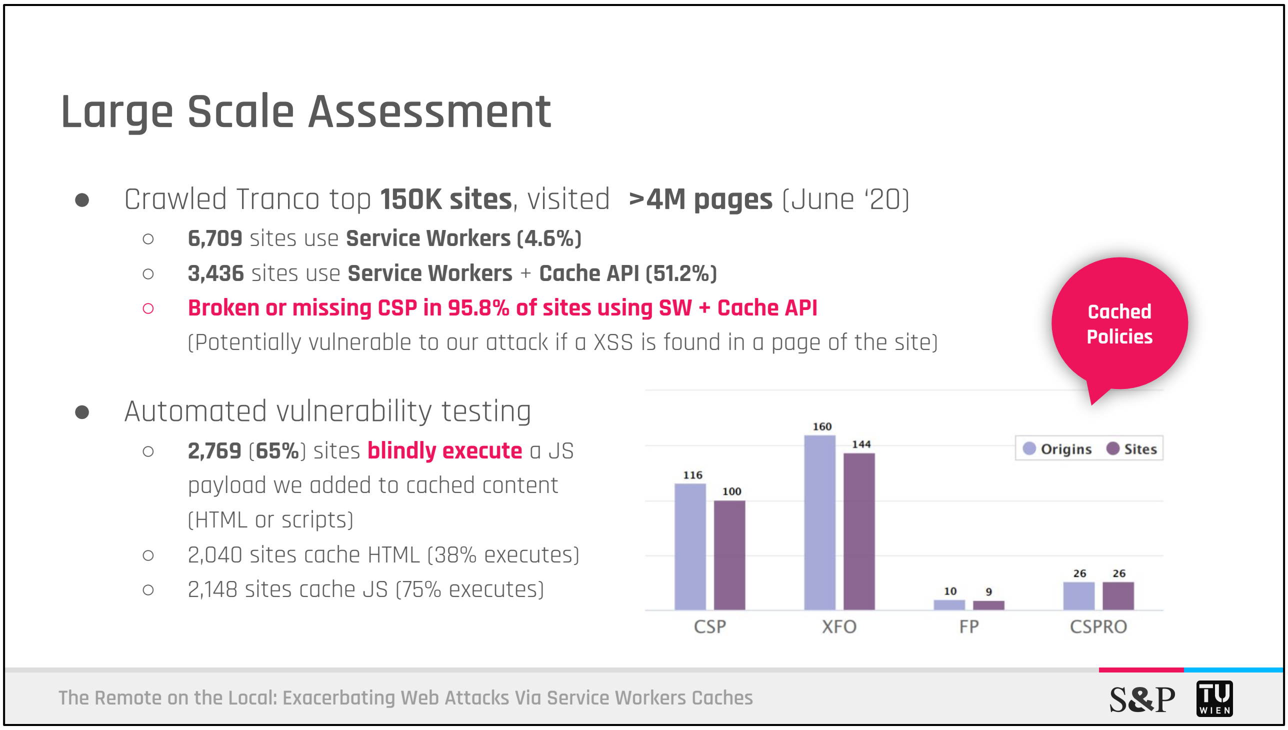


Data Collection

- **Runtime monitoring** of **Cache API calls** from **SWs** and **pages**
- Monitor injected by mitmproxy in SW code and by puppeteer in the pages
- Inspect home page and search engines for links to visit (<50 per origin)

To evaluate the pervasiveness of this threat, we performed a large scale assessment of the Cache API usage by websites. Mitmproxy and puppeteer for Google Chrome were used to inject a runtime monitor in the browser that records accesses to the cache performed in the page context and inside Service Workers. All tracked API calls and their arguments were then saved into a database for later analysis.

For each one of the origins that we processed, we visited up to 50 different links in the same origin to maximise the coverage of our analysis. We also investigated the main subdomains of the sites in our dataset.



Large Scale Assessment

- Crawled Tranco top **150K sites**, visited **>4M pages** (June '20)
  - **6,709** sites use **Service Workers (4.6%)**
  - **3,436** sites use **Service Workers** + **Cache API (51.2%)**
  - **Broken or missing CSP in 95.8% of sites using SW + Cache API**
    (Potentially vulnerable to our attack if a XSS is found in a page of the site)
- Automated vulnerability testing
  - **2,769** (**65%**) sites **blindly execute** a JS payload we added to cached content (HTML or scripts)
  - 2,040 sites cache HTML (38% executes)
  - 2,148 sites cache JS (75% executes)

Cached Policies

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

S&P TU WIEN

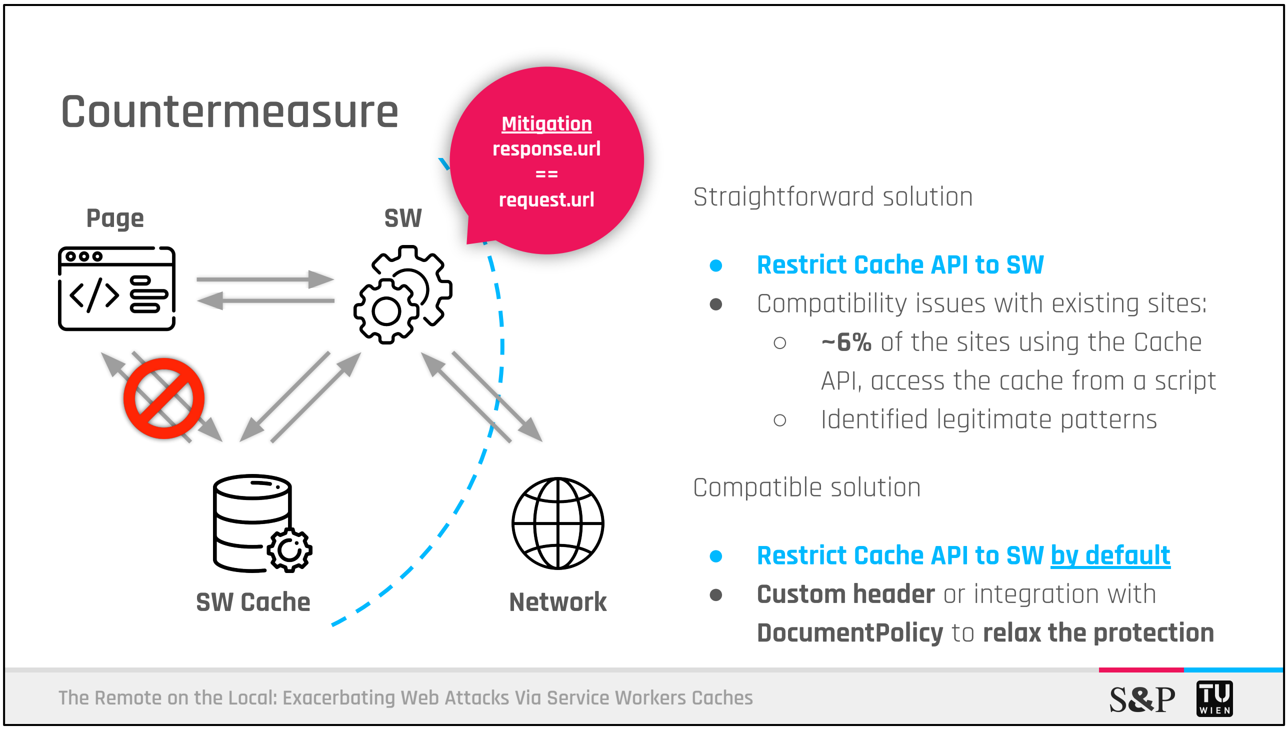
Overall we started from the Tranco list and we crawled 150K sites, for a total amount of 4M distinct pages. We found that Service Worker adoption as of June 2020 is still low (4.6%), but more than half of these sites are using the Cache API.

Since our threat model requires to have an XSS on the site, we analysed CSP adoption and found out that the deployment is broken or completely missing in 96% of the websites that are using Service Workers and the Cache API.

We implemented an automated testing strategy to verify the potential exploitability of the issue on websites with no CSP or with a trivially broken CSP:

1. We visited a random selection of links to fill the cache with some content
2. Then, we infected all JavaScript and HTML files with our payload, and we also stripped out all security headers
3. We visited the pages again and checked whether our payload was executed

We performed this injection using a browser extension that is available on our website. We found out that our payload was blindly executed by 65% of the tested sites.



Countermeasure

Straightforward solution

- **Restrict Cache API to SW**
- Compatibility issues with existing sites:
  - **~6%** of the sites using the Cache API, access the cache from a script
  - Identified legitimate patterns

Compatible solution

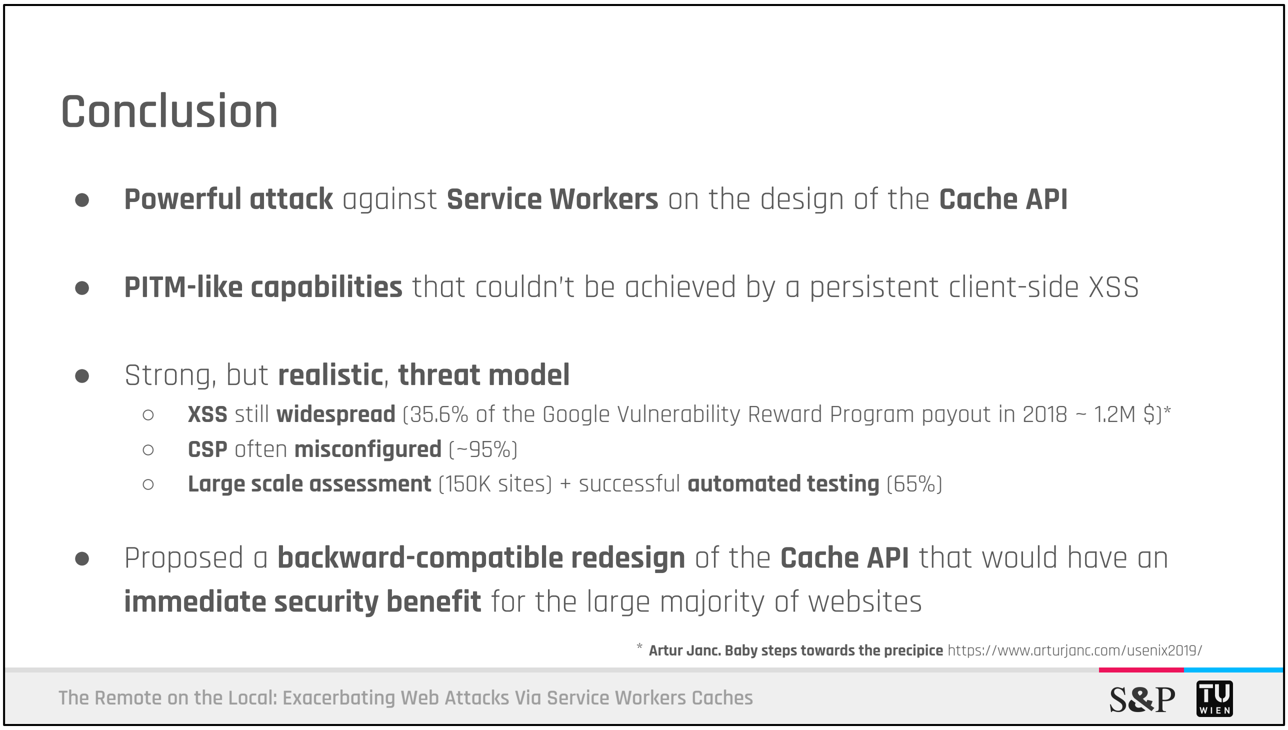
- **Restrict Cache API to SW by default**
- **Custom header** or integration with **DocumentPolicy** to **relax the protection**

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

S&P

A straightforward solution is to prevent scripts in the page to access the Cache API, basically restricting it to the Service Worker context only. Although secure, this solution would introduce compatibility issues on sites that are using of the Cache API from page context. From our measurements, we discovered that around ~250 sites would be affected by this change. Furthermore, we identified some legitimate caching patterns that require cooperation between Service Workers and scripts in the page.

So we propose to restrict the Cache API to Service Workers by default, but at the same time we suggest a way to relax the protection using either a custom header or the new Document Policy, so to preserve compatibility. As a mitigation, web developers can also harden the code of their Service Workers to protect against malicious modifications of the cache. This can be done by checking that the url attribute of cached responses corresponds to the url of the request. By doing so it's possible to spot tampering attempts, because synthetic responses have the "url" attribute set to the empty string. Notice that this change would be incompatible with websites using synthetic responses.



Conclusion

- **Powerful attack** against **Service Workers** on the design of the **Cache API**
- **PITM-like capabilities** that couldn't be achieved by a persistent client-side XSS
- Strong, but **realistic**, **threat model**
  - **XSS** still **widespread** (35.6% of the Google Vulnerability Reward Program payout in 2018 ~ 1.2M $)*
  - **CSP** often **misconfigured** (~95%)
  - **Large scale assessment** (150K sites) + successful **automated testing** (65%)
- Proposed a **backward-compatible redesign** of the **Cache API** that would have an **immediate security benefit** for the large majority of websites

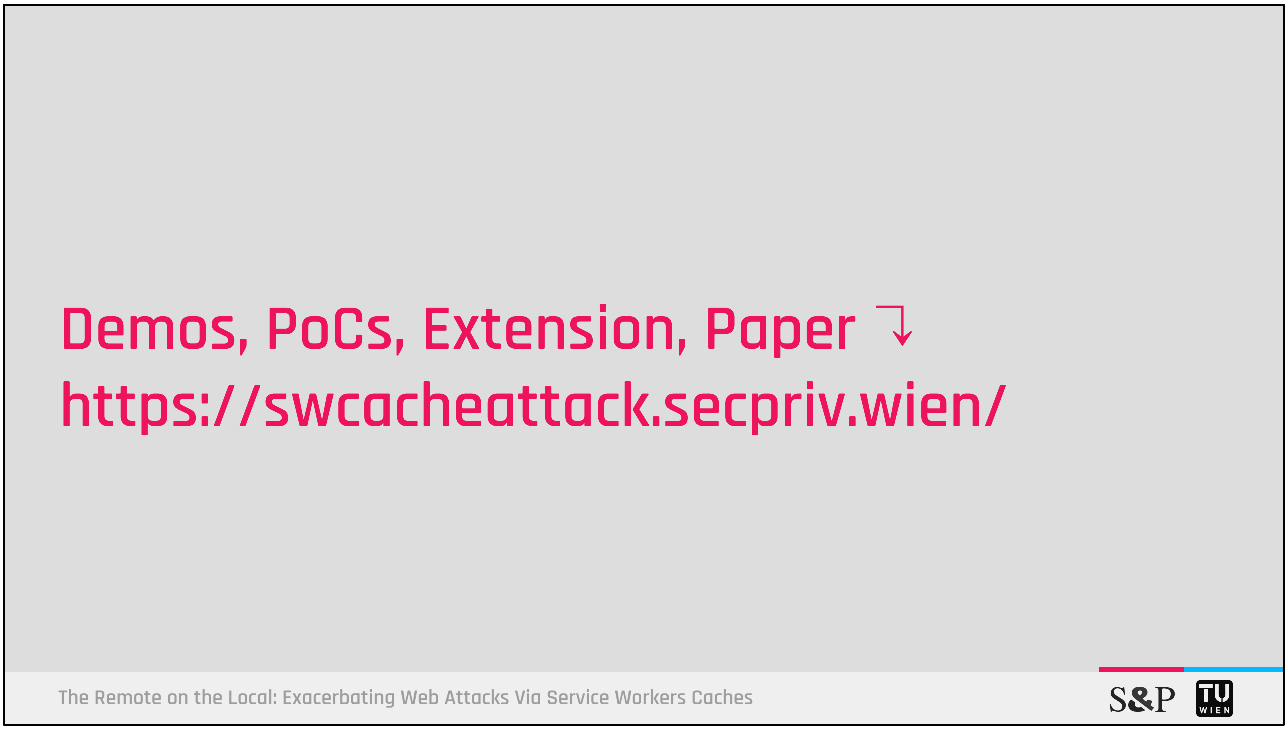
* **Artur Janc. Baby steps towards the precipice** https://www.arturjanc.com/usenix2019/

To summarize our contributions, we introduced a powerful attack on Service Workers that revolves around the design of the Cache API.

We showed that this attack grants person-in-the-middle capabilities that are more powerful than a persistent client-side XSS.

Although our threat model is strong, we proved that it's realistic considering that XSS are still widespread and mitigations such as CSP are not correctly implemented. Our large scale assessment showed 65% of the sites that are using Service Workers with the Cache API are affected.

Lastly, we proposed a backward-compatible redesign of the Cache API that would immediately secure the large majority of the sites affected by the issue discussed in this talk.



Demos, PoCs, Extension, Paper ↴
https://swcacheattack.secpriv.wien/

Icons from https://www.flaticon.com

Thank you!

https://swcacheattack.secpriv.wien/

Marco Squarcina (TU Wien)
✉ marco.squarcina@tuwien.ac.at
🐦 @blueminimal

Q+A?