DD2525 – Language-Based Security

Cross-Site-Leaks (XS-Leaks)

Tom Sorger, Marco Campione

KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science

June 2024

1 Introduction

Every day, we engage in countless online activities that we wish to keep private. These activities, we assume, are known only to the websites we interact with and their counterparts. However, since the inception of the internet, privacy-invasive leaks of information to other websites open in the same browser have been a persistent issue. These leaks, known as Cross-Site Leaks (XS-Leaks), represent an ongoing challenge for both website developers and browser manufacturers. [1]

XS-Leaks occur when unintended information about a user's activity on one website is exposed to another website through shared browser states or other vulnerabilities. This exposure can happen due to various factors, including the architectural complexities of modern web applications, differences in browser implementations, and the continuous discovery of new observation channels that attackers can exploit. As a result, even with efforts to secure websites against information leaks, these measures often prove to be incomplete, and information can still leak in other parts of the site. [1, 2]

The pervasive nature of XS-Leaks underscores the difficulty in addressing them comprehensively. Websites implement numerous strategies to mitigate these leaks, such as isolating sensitive information, employing advanced security headers, and using robust authentication methods. However, the effectiveness of these measures can vary significantly depending on the browser and its specific handling of web security. [1, 2]

In this paper, we will conduct an in-depth literature review on the different types of XS-Leaks using available online resources. This review will cover various techniques used to exploit XS-Leaks and the vulnerabilities they target, as well as talk about different mitigation strategies. By reviewing existing literature, we aim to build a detailed understanding of the state of XS-Leaks.

After the literature review, we will implement two web servers to demonstrate the different XS-Leaks. This practice component involves the creation of scenarios in which XS-Leaks can be observed and studied in a controlled and simplified environment. The aim is to provide hands-on experience with the techniques used to exploit and enhance our understanding of these vulnerabilities.

Finally, we will conclude with an investigation of open-source projects to identify potential XS-Leaks. This investigation involves the analysis of various open-source projects to identify XS-Leaks, providing real-world examples of these vulnerabilities.

2 XS-Leaks Background

Cross-site Leaks (XS-Leaks) are a class of vulnerabilities in the web platform that exploit side channels to infer information about users. These leaks arise from legitimate interactions between websites, leveraging browser features that allow cross-site communications. Unlike traditional cross-site attacks, XS-Leaks do not directly capture or intercept data but rather deduce the presence or absence of certain states or conditions in a victim's browser. These leaks rely heavily on the unintended exposure of side-channel information, leveraging browser behaviors and resource-loading patterns to exfiltrate data across origins. XS-Leaks are similar to Cross-Site Request Forgery (CSRF) attacks but focus on inferring rather than performing actions on behalf of the user. [1, 2]

2.1 History

The history of XS-Leaks can be traced back to the early 2000s when the web began to evolve into a more dynamic and interactive platform. As web applications grew more complex, so did the attack surface. Initial security efforts focused on direct attacks like Cross-Site Scripting (XSS) and CSRF. However, as defenses against these attacks improved, researchers began to uncover more subtle ways to leak information.

In the mid-2010s, security researchers began systematically identifying and categorizing various XS-Leaks. They showed how seemingly harmless browser behavior can be exploited to infer private data. This led to a deeper understanding of web-specific side-channel vulnerabilities and the realization that traditional security measures were not enough to prevent such leaks. [2, 3]

2.2 Root Cause

At the heart of XS-Leaks is the fundamental web principle of the Same-Origin Policy (SOP), which restricts how documents or scripts loaded from one origin can interact with resources from another origin. However, despite this protective barrier, XS-Leaks exploit minute side channels, so indirect information about resource states or user interactions that browsers inadvertently reveal. Such side channels may include timing discrepancies, error messages, status codes, or even rendering behaviors, which attackers can use to infer sensitive information from cross-origin resources. [1, 2]

2.3 Cross-Site Oracles

The information leveraged in an XS-Leak is typically binary and called "*oracles*". Oracles respond with a *YES* or *NO* to specific, strategically crafted questions in a way that an attacker can observe. For instance, an oracle might be asked:

"*Is the word '**secret**' present in the user's search results on another web application?*"

This can be the same as the query:

"*Does the query containing '**?query=secret**' return a **HTTP 200** status code?*"

Because an HTTP 200 status code can be detected through Error Events, this question essentially becomes:

"*Does loading a resource from '**?query=secret**' in the application trigger the '**onload**' event?*"

An attacker could repeat these queries with various keywords, allowing him to infer sensitive details about the user's data. Web browsers provide numerous APIs that, while intended for legitimate purposes, can unintentionally leak cross-origin information. [2]

2.4 Mitigation Efforts

Addressing XS-Leaks is challenging because it often requires changes at the browser level rather than the application level. [4] Some mitigation strategies will be discussed later in section 3.6.

3 Types of XS-Leaks

This section introduces selected types of XS-Leaks. We aim to provide a general introduction to each type and illustrate how each vulnerability can be exploited. At the end of this section, we will also discuss mitigation strategies. For further reading check out the given references.

3.1 XS-Search

Cross-Site Search (XS-Search) exploits the differences in response times or other observable behavior when a website processes search queries. These tiny differences can reveal whether certain data is present in the user's account, putting the user's privacy at risk without direct data access. [5, 6]

3.1.1 Technique

The technique behind this attack relies on timing. It starts by measuring the time required for a request to return results (*hit*) and separately, the time needed for a request with no results (*miss*). The attacker starts by defining a *hit baseline* which indicates whether the used query results in a hit or miss. The attacker then initiates a timing attack on the search endpoint by brute-forcing the first character (e.g., `?q=a`). If the timing measurement is *over* the hit baseline, indicating a *miss*, the attacker tries a new character (e.g., `?q=b`). If the timing measurement is *under* the hit baseline, indicating a *hit*, the attacker appends an additional character (e.g., `?q=aa`). This process is repeated iteratively until the entire secret is revealed. [5, 6]

This attack requires multiple timing measurements to ensure accuracy, which can be enhanced using inflation techniques and statistical analysis. Since relying on timings is often unreliable, you can also combine this attack with other XS-Leaks. Instead of brute-forcing letter by letter, attackers can also target specific words or phrases to leak the presence of results. The fundamental concept of this attack is its versatility, allowing it to be adapted to various XS-Leaks scenarios. [5, 6]

3.1.2 Exploit

A typical XS-Search attack usually follows the following steps:

1. Preparation An attacker creates a malicious web page designed to exploit the XS-Search vulnerability. This page contains scripts that send search queries to the target website where the user is authenticated. [6]

2. Attracting the Victim The attacker lures the victim to visit the malicious web page. This can be done via phishing emails, malicious advertisements, or any other form of social engineering that convinces the user to open the page. [6]

3. Search Query Execution Once the victim visits the malicious page, it automatically sends search queries to the target site. These queries are carefully constructed to probe for the existence of specific data in the user's search results. This technique was described earlier. [5, 6]

4. Measuring Response Times The key to XS-Search is analyzing the timing of responses. The malicious page measures how long it takes for the server to respond to each search query. Differences in response times can indicate whether the searched data exists. For example, a quicker response might indicate no results found, while a slower response might indicate the presence of matching data, possibly revealing sensitive information. As already mentioned, using only timing attacks can be unreliable which means it is possible and suggested to combine an XS-Search attack with other XS-Leaks to leak the same data but in a more reliable manner. [5, 6]

3.2 Frame Counting

Another XS-Leak is the *Frame Counting* attack, a novel method attackers utilize to gather sensitive information about a user's browsing activity. This attack leverages the capabilities of modern web browsers, particularly their rendering engines, to infer the content of cross-origin frames embedded within a web page. [7, 8]

3.2.1 Technique

Window references allow cross-origin pages to access some attributes of other pages, becoming available when using `iframe` and `window.open`. These references provide limited information about the window while still respecting the same-origin policy. One accessible attribute is `window.length`, which indicates the number of frames in the window. This attribute can reveal valuable information about a page to an attacker. While using frames or iframes is common and not inherently insecure, some websites might alter the number of frames based on user-specific information, such as GET parameters and the victim's data. An attacker could potentially infer details about the victim by measuring the `window.length` value on different pages. [7]

3.2.2 Case Scenario

A possible case scenario is a website where a user can search for information using a search engine. If the layout of the page changes depending on whether there are search results, an attacker could use frame counting, also combined with the *XS-Search* (see 3.1) technique, to leak data.

For example, in the case of a website adapting the layout of user profile pages based on factors such as gender or personally identifiable information (PII). Through simple page inspection and frame counting, an attacker can exploit this variance to reveal private details. [7]

3.3 ID Attribute

Another XS-Leak vulnerability takes advantage of the element ID attribute of a web page.

3.3.1 Technique

The id attribute is commonly used to identify HTML elements. However, cross-origin websites can exploit the focus event and *URL fragments* to check if a particular ID is present on a page. For instance, when `http://victim.local/foo#bar` is accessed, the browser tries to scroll to the element with `id="bar"`. This behavior can be detected cross-origin by loading `http://victim.local/foo#bar` in an iframe; if an element with `id="bar"` exists, the focus event triggers. The blur event can also serve this purpose.

Some web applications assign ID attributes to focusable elements, potentially exposing user information. These IDs may directly contain sensitive information (e.g., a secret) or indicate user-specific details (e.g., account status). [8, 9, 10]

3.3.2 Case Scenario

A bank offers its clients the option to generate short numeric One-Time PINs (OTPs) via its browser application to authenticate sessions on mobile devices. The bank implemented it the way that they use the OTP as the identifier for the button that reveals the PIN to the client. So an attacker could exploit the OTP by brute-forcing all possible OTP values until triggering the focus event. He would get to know the right OTP and therefore also gain unauthorized access to user accounts. [9]

Another example is a web application that utilizes predefined IDs and HTML elements for accounts with premium status or specific user genders. An attacker can identify the presence of these IDs on a victim's page, enabling them to extract account information. [9]

3.4 Error Events

When a web page sends a request to a server it processes the request and decides whether it should succeed (e.g., `HTTP 200`) or fail (e.g., `HTTP 404`) based on the given context. If the server responds with an error status, the browser triggers an error event for the page to handle. These error events also occur when the parser fails, such as when HTML content is mistakenly embedded as an image. [11]

3.4.1 Technique

Error events can originate from various HTML tags, and their behavior may differ across browsers. Factors influencing this behavior include the loaded resources, HTML tags, the presence of specific headers (e.g., `nosniff`, `Content-Type`), and the enforcement of default browser protections. The concept of leaking information through error events can be generalized and applied to numerous XS-Leaks. For instance, one technique involves cache probing, which uses error events to determine if a particular image has been cached by the browser. [11]

3.4.2 Case Scenario

Attackers can exploit this mechanism to determine whether a user is logged into a service by checking access to resources restricted to only authenticated users. The severity of this XS-Leak depends on the application but can facilitate sophisticated attacks, including user de-anonymization.

A real-world example involved a Twitter API endpoint that only the specified user could access. This endpoint would return an error for all other users except the owner, allowing attackers to de-anonymize the user.

Another similar bug involved exploiting an image authentication mechanism in private messages to de-anonymize the user. [11]

3.5 Cache Probing

Cache Probing is a technique used to detect whether a resource has been cached by a user's browser. This concept, known since the early days of the web, originally relied on detecting timing differences. When users visit a website, resources such as images, scripts, and HTML content are cached by the browser to speed up future visits. By determining which resources are cached, attackers can infer if a user has visited a specific page previously.

A more advanced form of cache probing uses error events to enhance accuracy and impact, offering a sophisticated method to leverage browser caching behavior for potential information leaks. [12]

3.5.1 Technique

Basic Cache Probing The basic principle involves an attacker determining if a resource is served from the cache or fetched from the network. The steps involved are:

1. **Resource Caching**: When a user visits a website, certain subresources get cached. This process helps reduce load times during subsequent visits by serving resources directly from the browser's local storage.
2. **Attacker-Controlled Page**: The user visits a page controlled by the attacker, which requests a resource typically cached by the target website. This resource could be anything commonly stored by the target site, like a logo or a script.
3. **Timing Analysis**: The attacker measures the response time. A quicker response indicates the resource was served from the cache, suggesting the user has visited the target site before. This method leverages the inherent differences in response times between cached and non-cached resources. [12]

Cache Probing with Error Events This advanced technique involves several steps to invalidate and detect cached resources using error events:

1. **Invalidate Cache**: Ensure the resource isn't previously cached by making the server return an error when fetching the subresource. This step is crucial to avoid false positives from previous sessions.
2. **Trigger Conditional Caching**: Initiate a request that causes specific items to be cached based on the user's state, such as being logged in. This can be achieved by embedding the target site in an `iframe` or by opening a new window using `window.open`.
3. **Error Detection**: Make another request that the server rejects, triggering an error event. If the resource was cached in the previous step, this request will succeed instead of causing an error. This method provides a more accurate assessment compared to timing-based techniques. [12]

Cache Invalidation Techniques

- **Using Errors**: To invalidate a resource from the cache, the attacker must force the server to return an error when fetching that subresource. There are a few methods to achieve this:
  - **Overlong Referrer Header**: Send a request with an overly long referrer header. This can cause the server to reject the request, invalidating the cached resource.
  - **Specific Request Headers**: Use request headers like `Content-Type` or `Accept` that may cause the server to fail. These headers can vary based on the application and server configuration.

- **Without Errors**: It's also possible to remove resources from the cache without server errors.
  - **AbortController**: Perform a fetch request with a `cache:'reload'` option and abort it before new content is received. This technique relies on the `AbortController` interface to cancel the request after it's initiated.
  - **POST Requests**: Use `POST` requests with `no-cors` to purge the resource using the top-level site's key, bypassing the partitioned HTTP cache.
  - **Cache Limit**: Exceed the browser cache limit, forcing the eviction of older resources to make room for new ones. [12]

3.5.2 Case Scenario

Social Network Cache Detection An attacker wants to determine if a user has visited a particular social network.

- **Resource Caching**: When the user visits the social network, specific subresources are cached, like profile pictures or scripts.
- **Visit Attacker's Page**: The user navigates to a page controlled by the attacker, which fetches a resource typically cached by the social network. The resource should be something unique and identifiable.
- **Timing Attack**: The attacker measures the response time to determine if the resource was cached. A faster response time indicates a cached resource, confirming the user's previous visit to the social network. [12]

Error Event Utilization

- **Cache Invalidation**: The attacker forces the server to return an error, ensuring the resource isn't cached from a previous visit. This step eliminates any historical data that could affect the attack.
- **Conditional Caching**: The user loads a page that caches a specific resource if logged in. For example, a unique image that only appears on a logged-in user's profile.
- **Error Detection**: A subsequent request triggers an error if the resource wasn't cached, allowing the attacker to infer the user's state. This method is highly accurate as it uses server-side behavior to confirm the cache status. [12]

3.6 Mitigation Strategies

Defending against XS-Leaks Attacks is challenging due to their diverse nature, impacting various web and browser components uniquely. Some bug bounty programs, e.g. *Google VRP*, have even stopped payments for new XS-Leaks reports, focusing instead on implementing systemic changes to counter XS-Leaks comprehensively. Google and other companies believe that investing in large-scale mitigations and platform changes to address entire categories of XS-Leaks is the right approach to mitigate these vulnerabilities in the future. Also, browsers offer opt-in mechanisms for mitigation, but they lack universal support. Some effective defense strategies are detailed further below. [4]

3.6.1 Opt-in Mechanisms

SameSite Cookies *SameSite Cookies* are a powerful security feature that addresses cross-site request security issues. They allow applications to control whether browsers include cookies in requests made to the same site. There are three modes for SameSite Cookies: `None`, `Strict`, and `Lax`.

- **None**: Disables all SameSite protections, reverting to the old cookie behavior. This mode is not recommended and must be used with the Secure flag.
- **Strict**: Prevents the browser from including cookies in any cross-site requests, such as requests made by `<script src=...>`, `<img src=...>`, `fetch()`, and `XHR`. Even when a user clicks a link to the resource, the cookies are not included.
- **Lax**: Similar to strict but allows cookies to be included in requests triggered by cross-site top-level navigations. This makes Lax easier to implement without breaking incoming links to your application. However, an attacker can still exploit this by triggering top-level navigation using `window.open`, which lets them maintain a reference to the window object. [13]

Cross-Origin Resource Policy (CORP) The *Cross-Origin Resource Policy* (CORP) is a web security feature that helps websites prevent their resources from being accessed by other sites. Unlike *Cross-Origin Read Blocking* (CORB), which automatically blocks some cross-origin reads, CORP is an optional defense that developers can enable. CORP helps protect against speculative execution attacks and XS-Leaks by ensuring that sensitive resources cannot be accessed by attacker-controlled processes. Developers can specify which groups of origins (such as `"same-site"`, `"same- origin"`, or `"cross-site"`) are permitted to read their resources. By setting the CORP header to `"same-site"` or `"same-origin"`, developers can prevent attackers from accessing those resources, offering robust and highly recommended protection. [14]

Cross-Origin-Opener-Policy (COOP) The *Cross-Origin-Opener-Policy* (COOP) allows you to ensure that a top-level window is isolated from other documents by putting them in a different browsing context group so that they cannot directly interact with the top-level window. For example, if a document with COOP opens a pop-up, its `window.opener` property will be `null`. The COOP header takes three possible values:

- **same-origin**: Documents that are marked `same-origin` can share the same browsing context group with `same-origin` documents that are also explicitly marked `same-origin`.
- **same-origin-allow-popups**: A top-level document with `same-origin-allow-popups` retains references to any of its popups that either don't set COOP or opt out of isolation by setting a COOP of `unsafe-none`.
- **unsafe-none**: `unsafe-none` is the default and allows the document to be added to its opener's browsing context group unless the opener itself has a COOP of `same-origin`. [15]

3.6.2 Application Design

Cache Protections via Random Tokens To defend against cache probing-based XS-Leaks, applications can employ *Cache Protections via Random Tokens*. Instead of disabling caching, this method involves adding random tokens to URLs. By appending a unique token to the URL of each subresource, attackers cannot easily determine if an item is in the cache, as they cannot guess the token. For instance, instead of directly referencing a user's profile photo at `/user/<USERNAME>.png`, the application would use `/user/<USERNAME>.png?cache_buster=<RANDOM_TOKEN>`. This token does not need to be processed by the server but ensures that attackers cannot probe the cache without it.

This approach is compatible with all major browsers and maintains caching functionality, though it can be challenging to implement. [16]

Subresource Token-Based Protections Including a user-specific token in every request offers strong protection against most XS-Leak techniques. This method requires a token to be included with resource requests, which the server verifies as valid for the current user. For example, in a search application, the server provides a secure token when the main page loads. The token is then included in search requests, and the server verifies it. If the token is invalid, the request is rejected. This approach prevents attackers from making legitimate requests without a valid token. It is applicable to both authenticated and unauthenticated subresources, though other strategies like same-site cookies or specific cache protections may also be used. [17]

3.6.3 Secure Defaults

Cross-Origin Read Blocking (CORB) **Cross-Origin Read Blocking** (CORB) is a security mechanism designed to prevent attackers from loading certain cross-origin resources, thereby protecting against speculative side-channel attacks. These attacks allow an attacker to read the memory of a process where both cross-site pages are embedded. CORB prevents the loading of sensitive cross-origin resources into an attacker-controlled process. For instance, if an attacker attempts to load cross-origin HTML, XML, or JSON into an image tag, CORB intervenes, treating the response as though no data was returned.

To classify resources, CORB relies on the `Content-Type` header, the `nosniff` header, and various other heuristics. [18]

Partitioned HTTP Caches To defend against cache probing attacks, browser developers are implementing partitioned HTTP cache functionality to ensure that each website has a distinct cache. Cache probing exploits the shared nature of a browser's HTTP cache across different websites. By using partitioned caches, this defense mechanism assigns cache keys based on tuples ensuring that the cache is unique to the requesting site. This approach makes it difficult for attackers to interact with the cached contents of other sites. [19]

4 XS-Leaks Lab (XSLL)

We implemented two web servers to showcase the different types described in section 3. One of them acts as a malicious attacker, while the other functions as a victim leaking information. We opted for this approach because it allows us to control both sides, attacker and victim, and to keep the scenario as simple as possible, since at this stage, the lab is intended to serve as a straightforward demonstration. Additionally, we aim to avoid attacking other web pages without permission, making this an ethical approach that complies with legal standards. The implementation of the *XS-Leaks Lab* (XSLL) can be found at [20].

4.1 Web Server

We decided to use the official `nginx` docker image [21] as a starting point for the web servers. As previously mentioned, we implemented two web servers: one acts as the attacker, while the other as a victim leaking data, allowing the attacker to gain information.

Both servers have a quite similar setup which is a landing page referring to all demonstrated XS-Leaks. On each page of a specific leak, you can see the attack demonstration.

4.2 XS-Search

4.2.1 Preparation

The victim web server implements a Query-Based Search System for pictures stored on the web server. This means that the website provides a search bar in which the user can enter a query. The system then looks up if there is a picture with the same name or starts with the specified query. If so the web page will load the pictures in iframes, if not it will load no pictures.

4.2.2 Attack

Testing the initial approach of using a timing-based attack showed us how unreliable the timings were. So after testing how we could measure the delays of queries with loading pictures and queries without loading pictures, we opted for combining the *XS Search* with *Frame Counting*.

Since we can use the URL to specify our query, e.g. the query "`http://victim.local/xs-leaks/xs-search.html?q=secret`" opens the website and automatically queries for the search term "`secret`", we can automate this to find existing queries.

As described in 3.1.1, we brute-force all possible combinations and count the frames on each result. Finding at least one iframe indicates a hit while finding no iframe indicates a miss. To get more information about how frame counting works please look into the previous sections.

4.3 Frame Counting

4.3.1 Preparation

As preparation for the victim web server, we created the scenario revealing the gender of the user as described in 3.2.2. We created a small form containing three radio buttons on which the users can select their gender. When selecting the radio button saying *male* the form will show one iframe, when selecting *female* two iframes, and when selecting *prefer not to say* zero iframes.

4.3.2 Attack

On the attacking web server, we implement a function that creates a reference to the window we want to count the frames of. In our case, that's "`http://victim.local/xs-leaks/frame-counting.html`".

After waiting for 2 seconds to allow the page to load, we log the number of iframes present (see code 1). [7]

```
// Get a reference to the window
var win = window.open('http://victim.local/xs-leaks/frame-counting.html');

// Wait for the page to load
setTimeout(() => {
    // Read the number of iframes loaded
    console.log("%d iframes detected", win.length);
}, 2000);
```

Code 1: Frame Counting Attack.

Since the attacker knows the web page of the victim he also knows the meaning of the returned number. With this number, he automatically knows what gender the attacked user selected.

4.4 ID Attribute

4.4.1 Preparation

The preparation for the victim web server is quite simple. All we have to do is add an HTML tag with an id as seen in code 2.

```
<input id=42>
```

Code 2: Preparation for Leaking ID Attribute.

4.4.2 Attack

The goal of the attacking web server is now to figure out the id which was specified in code 2; `42` so to be precise.

As described in 3.3.1 we can use the `onblur` event to get "notified" whether it found a given id (see code 3).

```
onblur = () => {
    alert(`Element ID "${id}" found!`);
}
```

Code 3: Listening on `onblur` event.

This code snippet will raise an alert telling the attacker the ID that was found when losing focus. In case there is no hit, no alarm will be raised.

To find the ID and trigger the `onblur` event we can use the code snippet seen in code 4.

```
let ifr = document.createElement('iframe');
ifr.src = 'http://victim.local/xs-leaks/id-attribute.html#42';
document.body.appendChild(ifr);
```

Code 4: iFrame Creation.

This code creates an iframe element and sets its source to `http://example.local/id-attribute.html#42`. It then appends this iframe to the body of the current HTML document, effectively embedding the specified URL into the page. In case the embedded web page has an element with the specified ID it will trigger the event. [9]

Automating this script gives us an automated way to look for ID attributes.

4.5 Error Events

4.5.1 Preparation

The preparation for the victim web server is quite simple. We just need to implement a web page that we will later use during the attack.

4.5.2 Attack

On the attacker web server, we implemented a simple web page with two buttons that execute the function in code 5. [11]

```
function probeError(url) {
    let script = document.createElement('script');
    script.src = url;

    script.onload = () => {
        console.log('Onload event triggered');
    }

    script.onerror = () => {
        console.log('Error event triggered');
    }

    document.body.appendChild(script);
}
```

Code 5: Error Events Attack.

In conclusion, the `onload` event is triggered when the script has finished loading and executed successfully. Similarly, the `onerror` event is triggered if an error occurs while loading or executing the script.

By clicking the first button the `probeError` function will be called with the URL parameter "`http://victim.local/xs-leaks/error-events.html`" and will result in triggering an `onload` event because the URL passed to the function exists. Whereas if the second button is clicked the `probeError` function will be called with the URL parameter "`http://victim.local/xs-leaks/404`" and will result in triggering an `error` event because the URL passed to the function does not exist.

4.6 Cache Probing

4.6.1 Preparation

The preparation for the victim web server is quite simple. We just need to implement a web page containing an image, that we will later use during the attack.

4.6.2 Attack

On the attacking web server, we implement a function called `ifCached` that checks if a URL is cached using a fetch request with a very short timeout (see code 6). It handles fetch success and errors to determine caching status. [12]

```
async function ifCached(url) {
    var controller = new AbortController();
    var signal = controller.signal;
    var timeout = setTimeout(() => {
        controller.abort();
    }, 9);
    try {
        let options = {
            mode: "no-cors",
            credentials: "include",
            signal: signal
        };
        await fetch(url, options);
    } catch (err) {
        console.log("The resource is not cached");
        return false;
    }
    clearTimeout(timeout);
    console.log("The resource is cached");
    return true;
}
```

Code 6: Cache Probing Attack.

In case the execution of `fetch` in code 6 takes too long the function will throw an error and therefore assume that the resource is not cached. Whereas when the `fetch` finishes in time the function assumes that the resource is cached. Varying the time we allow `fetch` to finish before throwing an error might be crucial depending on the system for this function to work.

5 Open-Source Research

5.1 Methodology

During the Open-Source investigation for XS-Leaks vulnerabilities, we began by establishing a methodology to guide the research process.

This included defining the scope, objectives, and specific techniques to identify and analyze potential XS-Leaks vulnerabilities. We decided to utilize some of the most popular repository hosting services, *GitHub* and *GitLab*, making them an ideal starting point for finding open-source projects.

We focused on projects that were actively maintained and high-starred, indicating their popularity and extensive user base. This strategic choice was made because identifying vulnerabilities in widely used projects would potentially impact a larger number of users, thereby maximizing the practical benefits of the research.

We also opted for those that contain login pages. This way we also have the possibility of finding projects with potential administration pages, which could be interesting for data leakage.

5.2 Investigation and Results

The investigation started with an in-depth examination of multiple projects, for example, *speedtest-tracker* [22] and *firefly iii* [23] to name two. All projects are widely used in the self-hosting community, e.g., *speedtest-tracker* offers statistics about your internet connection and *firefly iii* is a budget planner and wealth manager, both meeting our requirements, as they respectively have 2.3k and 14.6k stars on GitHub at the current time, have an admin page, and can also be self-hosted.

However, despite all efforts, no significant XS-Leaks vulnerabilities were discovered in the open-source projects analyzed.

This outcome can be attributed to several factors. Firstly, the time allocated for the research was relatively short, which is a critical limitation given the complex nature of XS-Leaks. These types of vulnerabilities are often subtle and require deep, time-consuming analysis to identify. Additionally, the security landscape has improved significantly, with many of these leaks being previously identified and fixed by the open-source community and security experts. Moreover, modern web browsers have implemented robust security measures to block many common XS-Leaks, adding another layer of difficulty to the task of finding new vulnerabilities.

Despite the lack of immediate results, the rapidly evolving nature of web technologies means that new vulnerabilities can emerge at any time. Given the depth of exploration required, extending this research into a thesis project could be highly beneficial. A more extended time frame would allow for a more thorough investigation, potentially uncovering more sophisticated XS-Leaks that shorter studies might miss. It would also provide an opportunity to develop and refine new methodologies and tools for detecting such vulnerabilities.

6 Conclusion

In this comprehensive study, we delved into the complex and evolving domain of Cross-Site Leaks (XS-Leaks). Our exploration began with a thorough literature review, establishing the fundamental principles, historical context, and inherent challenges posed by XS-Leaks. We identified the root causes, traced the evolution of these vulnerabilities, and dissected the nuanced mechanisms through which XS-Leaks exploit web interactions to infer sensitive information.

Through our research, we categorized and analyzed various types of XS-Leaks, including XS-Search, Frame Counting, ID Attribute, Error Events, and Cache Probing. Each type presents unique exploitation techniques and scenarios, demonstrating the diverse methods attackers employ to compromise user privacy and security. The detailed examination of these leaks underscored the importance of understanding both the technical specifics and the broader implications for web security.

Our practical component involved the development of the XS-Leaks Lab (XSLL) to simulate and study XS-Leaks in controlled environments. This hands-on approach provided valuable insights into the mechanics of these vulnerabilities and the effectiveness of current mitigation strategies. XSLL served as a vital tool for demonstrating how these attacks can be executed.

Our investigation into XS-Leaks vulnerabilities in popular open-source projects did not reveal XS-Leaks vulnerabilities, likely due to the complexity of these vulnerabilities and improved security measures. Nevertheless, the evolving nature of web technologies indicates that extending this research could uncover new vulnerabilities. This means a longer-term project would allow for a deeper and more thorough analysis, enhancing the detection capabilities.

7 Future Work

Expanding the XS-Leaks Lab (XSLL) and including Capture The Flag (CTF)-like exercises can provide a practical way to deepen understanding and improve skills related to XS-Leaks. By designing CTF scenarios that mimic real-world XS-Leaks attacks, users can gain hands-on experience in identifying these vulnerabilities, fostering a more robust learning environment.

Another critical area of future work involves conducting more detailed and extensive searches for XS-Leaks within open-source projects. This process is inherently time-consuming, requiring one to familiarize with each project's structure and codebase to effectively identify vulnerabilities. However, this thorough approach is essential for uncovering XS-Leaks, ultimately contributing to the development of more secure web applications. By dedicating resources to these efforts, the research community can enhance both theoretical knowledge and practical skills, driving forward the field of web security.

References

[1] Jannis Rautenstrauch, Giancarlo Pellegrino, and Ben Stock. "The Leaky Web: Automated Discovery of Cross-Site Information Leaks in Browsers and the Web". In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023, pp. 2744–2760. DOI: 10.1109/SP46215.2023.10179311.

[2] *XS-Leaks Wiki*. URL: https://xsleaks.dev/.

[3] Nethanel Gelernter and Amir Herzberg. "Cross-Site Search Attacks". In: Oct. 2015. DOI: 10.1145/2810103.2813688.

[4] *Defense Mechanisms*. URL: https://xsleaks.dev/docs/defenses/.

[5] *XS-Search*. URL: https://xsleaks.dev/docs/attacks/xs-search/.

[6] Nethanel Gelernter and Amir Herzberg. "Cross-Site Search Attacks". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 1394–1405. ISBN: 9781450338325. DOI: 10.1145/2810103.2813688. URL: https://doi.org/10.1145/2810103.2813688.

[7] *Frame Counting*. URL: https://xsleaks.dev/docs/attacks/frame-counting/.

[8] *Cross-site leaks Cheat Sheet*. URL: https://cheatsheetseries.owasp.org/cheatsheets/XS_Leaks_Cheat_Sheet.html.

[9] *ID Attribute*. URL: https://xsleaks.dev/docs/attacks/id-attribute/.

[10] Teo Selenius. "XS-Leaks (Cross-Site Leaks) Attacks and Prevention". In: *AppSec Monkey* (Feb. 2021). URL: https://www.appsecmonkey.com/blog/xs-leaks.

[11] *Error Events*. URL: https://xsleaks.dev/docs/attacks/error-events/.

[12] *Cache Probing*. URL: https://xsleaks.dev/docs/attacks/cache-probing/.

[13] *SameSite Cookies*. URL: https://xsleaks.dev/docs/defenses/opt-in/same-site-cookies/.

[14] *Cross-Origin-Resource-Policy*. URL: https://xsleaks.dev/docs/defenses/opt-in/corp/.

[15] *Cross-Origin-Opener-Policy*. URL: https://xsleaks.dev/docs/defenses/opt-in/coop/.

[16] *Cache Protections*. URL: https://xsleaks.dev/docs/defenses/design-protections/cache-protections/.

[17] *Subresource Protections*. URL: https://xsleaks.dev/docs/defenses/design-protections/subresource-protections/.

[18] *Cross-Origin Read Blocking*. URL: xxx.

[19] *Partitioned HTTP Cache*. URL: https://xsleaks.dev/docs/defenses/secure-defaults/partitioned-cache/.

[20] URL: https://github.com/t-sorger/xs-leaks-lab.

[21] URL: https://hub.docker.com/_/nginx.

[22] URL: https://github.com/alexjustesen/speedtest-tracker.

[23] URL: https://github.com/firefly-iii/firefly-iii/.

A Contribution

Tom was responsible for handling several XS-Leaks, including *XS Search*, *Frame Counting*, and *ID Attribute*. Marco, on the other hand, managed the XS-Leaks related to *Error Events* and *Cache Probing*. Tom also took the lead in establishing the infrastructure for *XSLL* (XS-Leaks Lab). Both were equally active in the open-source part of the project as well as researching mitigation techniques.