Saarland University
Faculty of Mathematics and Computer Science
Department of Computer Science

Masterthesis

XS-Leaks: How affected are browsers and the web?

submitted by

Jannis Rautenstrauch

on November 10, 2021

Reviewers

Dr.-Ing. Ben Stock

Dr. Giancarlo Pellegrino

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,_____ _____

(Datum/Date) (Unterschrift/Signature)

Acknowledgements

Foremost, I would like to thank my supervisors, Dr.-Ing. Ben Stock and Dr. Giancarlo Pellegrino, for the opportunity to write this thesis with them at the CISPA Helmholtz Center for Information Security. The regular discussions with them shaped the path of this thesis, and their valuable feedback greatly improved this work.

In addition, I would like to thank my friends Florian Hantke and Maja Toebs for proofreading this thesis and for their emotional support.

Abstract

Cyberattacks causing considerable damages to businesses and users are in the news almost every week. Cross-Site information leaks (XS-Leaks) are one type of web attack targeting users. In such attacks, victims visit a malicious website that infers information about them on other sites by abusing browser side-channels. Inferred information can reach from access detection to deanonymization. The scientific community has known XS-Leaks since the 2000s. Still, there is not enough information available to estimate how big of an issue such attacks are for the web ecosystem today. This thesis aims to close this knowledge gap with two approaches. One approach is evaluating how different browser side-channels behave in modern web browsers by observing several channels for 387,072 responses. The other approach is scanning as many websites as possible for vulnerable URLs using a newly created fully automated pipeline. The results show that browser behavior significantly differs for many channels. Studying the differences, we discovered ten security-relevant bugs and reported them to two browser vendors. With the second approach, we detected vulnerable URLs on 258 out of 352 tested sites, presenting the largest study of XS-Leaks to date. Based on the results of this thesis, we conclude that XS-Leaks pose a considerable problem for the web ecosystem, as most websites are currently vulnerable, and millions of users could be heavily affected. However, the results also show that websites have the means to be XS-Leak free and that getting rid of differing edge case behavior in browsers could reduce the attack surface by about 50%. We think that introducing more secure defaults in browsers and increased adoption of new security features by web developers could make XS-Leaks irrelevant in the foreseeable future.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In 2020, an average internet user between the age of 16 and 64 spent 6 hours and 54 minutes on the internet per day [22], more than ever before. All kinds of applications for private activities such as communication platforms, social media, music streaming, blogs, and games use the web as their primary access point. In addition, people perform many professional activities such as writing text and code, reading reports, and attending virtual meetings on the web. In short, our modern life would be unthinkable without the web as we know it today.

With the popularity of the web as an application platform, the web attracts much attention from criminal groups and security researchers. Over the years, researchers discovered many attacks targeting the server-side of the web, such as SQL injections [89] or path traversal [68]. Nowadays, attacks targeting the client-side of the web such as Cross-Site Scripting (XSS) [61] or Cross-Site Request Forgery (CSRF) [60] are common as well. These attacks are not only theoretical vulnerabilities but many of them are actively exploited, causing substantial financial losses for businesses and users. For instance, Cybersecurity Ventures estimates that global cybercrime will deal more than 10 billion US-Dollars of damage by 2025 [59].

One web attack targeting users is called Cross-Site information leak (XS-Leak) attack. Although this attack has already been known under various names since the 2000s such as timing attacks on privacy [31], login detection attacks [41], deanonymization attacks [90] and Cross-Site Search Attacks [38], it only gained traction recently [92, 88, 39]. The general idea of the attack is to abuse browser side-channels to infer user state information on other target sites. First, a victim visits an attacker-controlled website. On this website, the attacker performs a cross-site request to a target site. The browser automatically attaches the victim's information, such as cookies, to the request. The malicious site then observes the browser side-channels to infer state information of the victim. An attacker

can then use the inferred information for various malicious purposes. As one example, consider the following. A user has an account on an illegal video-sharing website and is currently logged in. This user receives an e-mail containing a link in another tab. Next, the user opens the link in the same browser. The link points to an attacker-controlled site. In the foreground of the attack page, there is some non-malicious content. In the background, the website requests the URL */profile/my/img.png* from the video-sharing site. The browser attaches the cookies and performs the request. The server recognizes the user based on the cookies and returns the corresponding profile image. The browser then fires a load event on the malicious website. The attacker now knows that the user has an account on the illegal video-sharing site. The attacker can infer this because requests from users without an account do not return a valid image and would generate an error event. Finally, the attacker blackmails the user and threatens the user to pay money to the attacker. Otherwise, the attacker will leak the information to the police.

Even though XS-Leaks have been known for a long time, many knowledge gaps still exist. With the available information from previous research, it is hard to estimate how big of an issue XS-Leaks are for the web ecosystem. It is also impossible to quantify the effect recent actions of different browser vendors have had on XS-Leaks. Example actions were shifting to SameSite *Lax* by default or introducing new security headers.

Most reports state that a leak method works for at least one example pair of responses in one browser and version. For example, a response with a valid image body can be distinguished from a response that has an HTML body and is otherwise the same by observing the *load* and *error* events on image tags in Firefox 3 [41]. One does not know, however, if the leak method works in other browsers and newer versions. One also does not know the exact properties causing the observable difference. In the above example, one could ask "Will all responses with a valid image body result in a *load* event or are there additional restrictions on the status-codes or the headers?", "Will all responses with a non-image body result in an *error* event?", "Do responses that neither generate a *load* nor an *error* event exist?", and "Is the behavior the same in different browsers and versions?". The answers to these questions enable the creation of groups of responses that result in the same observation for every browser where every two responses from two different groups form a distinguishable response pair. To fill the knowledge gap of when precisely a leak method works, we formulate the following research question:

> **R1:** Which groups of responses can different leak methods distinguish in different browsers?

Additionally, we do not know how big of a problem XS-Leaks are in the wild. Questions such as "How many *dangerous* response pairs do exist on websites?", "Are defenses such

as SameSite cookies or Fetch metadata deployed widely and correctly, thus preventing the problem for many websites?", and "Which methods work often, and which methods do not work in practice?" arise. With these additional questions in mind, we then formulate the second research question:

> **R2:** Which XS-Leak methods work how often on websites in the wild in different browsers?

By answering these two questions, this thesis makes the following contributions:

- We present the first comprehensive analysis of which browsers can distinguish which groups of responses using different XS-Leak methods. The results show that most methods still work in all browsers, but significant differences exist between the exact response groups browsers can distinguish.
- We report ten bugs related to XS-Leaks in two major browsers.
- We release a static tool that shows if and how two responses can be distinguished in different browsers.
- We present the largest study on the prevalence of XS-Leaks in the wild, finding XS-Leaks on a total of 258 sites.
- We release a dynamic pipeline to scan websites for XS-Leaks.
- We provide evidence for the claim that joint work of web developers and browser vendors could greatly reduce the prevalence of XS-Leaks in the wild.

To conclude the introduction, we present the structure of the thesis. In chapter 2, we explain the background on web technologies, XS-Leaks, and browser automation necessary to understand this thesis. In chapter 3, we answer R1 by examining how different XS-Leak methods behave for a large number of responses in several browsers. In chapter 4 we answer R2 by presenting the largest to date study on the prevalence of XS-Leaks in the wild with finding leaky URLs on over 250 websites. Then, in chapter 5, we discuss the ethical consequences of the experiments, the limitations of a fully automatic test pipeline, relate to previous work, and point a way out of the XS-Leak problem. Finally, in chapter 6, we conclude this thesis.

Chapter 2

Background

Cross-Site information leaks (XS-Leaks) can affect every web user in any web browser when visiting a malicious website. Background knowledge about the web and its security model is necessary to understand why and how XS-Leak attacks work and what consequences they can have. Additionally, to understand the chosen methodology, knowledge about browser automation and web crawling is necessary.

In this chapter, we explain the core concepts needed to understand this thesis. The first part explains the general web technologies involved in an XS-Leak attack. Then, the second part explains XS-Leak attacks in detail, including why they are a severe issue in today's web ecosystem. Finally, the last part explains the browser automation techniques needed to perform automated testing of XS-Leaks.

2.1 General web technologies

XS-Leaks are a security vulnerability in web applications that abuse features built into web browsers. Knowledge about the web and its security model is needed to understand why such attacks can occur. First, we explain what resource inclusions are and why including resources from other websites is crucial to make the modern web work. Then, we summarize the main security principle on the web, the same-origin policy (SOP), and present how the web handles state information. Finally, we introduce additional security features in browsers, including cookie security.

Resource inclusions: A modern website does not only consists of a single publicly accessible text document as originally intended by Tim Berners-Lee [8]. It also contains images, executable code, styling directives, and further resources. These interactive

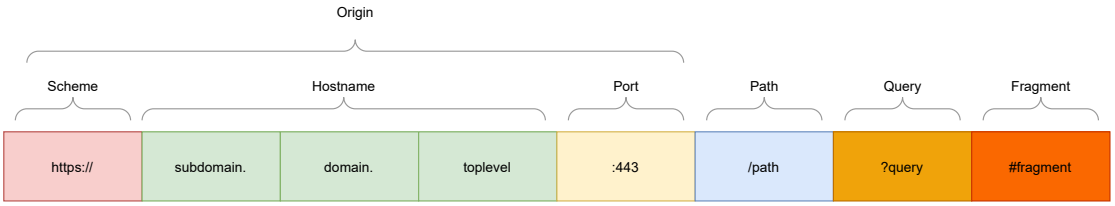


Figure 2.1: Structure of a Uniform Resource Locator (URL).

websites with a mix of resources are not preassembled on the server but built together in the browser and are often tailored to individual users.

When a web browser requests a website, it usually first fetches a HyperText Markup Language (HTML) [45] document from the webserver. This document represents the entry point of the website. This HTML document can include many other resources, such as images or videos, by embedding them with a corresponding HTML tag and source Uniform Resource Locator (URL). The browser then fetches all resources and embeds them accordingly. For example, the browser displays a returned image. Complete websites can also be embedded using the IFrame tag. A document can even include scripts that can execute arbitrary code. These scripts can dynamically add or remove resources, directly request resources using fetch, or open new browser windows or tabs using *window.open*. Table A.1 in the appendix lists all inclusion methods used in this thesis.

All included resources can either belong to the same site as the initial HTML page or external sites. The ability to include resources from any web server makes it possible to easily combine resources to create the interactive experiences we know the web for today. For example, a modern blog can include several social media like and share buttons, several high-quality images from the web, a managed comment system, and external ads.

Same-origin-policy: The ability to include resources from any web server is integral to create the modern web experience. However, it also means that web developers and browser vendors need to consider the security implications of this feature as pages belonging to different websites should not be able to access each other freely.

To reason about the security boundaries of the web, we have to define several terms. Figure 2.1 illustrates the structure of a Uniform Resource Locator (URL). A URL consists of a scheme *https*, a hostname *subdomain.domain.toplevel*, a port *443* (when omitted a default value for each scheme exist), a path */path* (can be empty), and optionally query parameters *?query* and a fragment *#fragment*. An origin is the combination of scheme, hostname, and port and usually belongs to one web application. The hostname can consist of arbitrary many levels, for example, deep.demo.websec.saarland. A site is the

Origin A	Origin B	Origin?	Site?	Explanation
https://demo.websec.saarland:443	https://demo.websec.saarland:443	Same-Origin	Same-Site	Exact match
	https://demo.websec.saarland	Same-Origin	Same-Site	HTTPS default port is 443
	http://demo.websec.saarland:443	Cross-Origin	Same-Site	Different scheme
	https://demo.websec.saarland:**80**	Cross-Origin	Same-Site	Different port
	https://**not**.websec.saarland:443	Cross-Orgin	Same-Site	Different subdomain
	https://**secweb.saarland**:443	Cross-Origin	Cross-Site	Different site

Table 2.1: Possible relations of two origins.

combination of an effective top-level domain (eTLD) (e.g., .com or .github.io[1].) plus the part directly in front of it (websec.saarland in the above example). The site is often called eTLD+1 and is what one business or organization controls.

Table 2.1 explains in what context two origins can be to each other in respect to site and origin. A resource inclusion in a web document can be same-origin, same-site, or cross-site to the origin of the embedding document. In the context of this thesis, we only study cross-site attacks as they can be performed on every website. Same-site attacks are usually more powerful but only work in some instances where an attacker can gain control over the targeted website's sub- or sister domains.

As stated before, content delivered on the web is often tailored towards a specific user or a group of users and should not be available to everybody. For example, only the users themselves should have access to their social media messages. Thus, it is of uttermost importance that malicious websites requesting resources from other websites do not have access to this private content. The most fundamental security mechanism on the web: the same-origin policy (SOP) [63] achieves this isolation.

The same-origin policy dictates that resources from a different origin than the origin of the requesting document cannot be directly accessed from the requesting document. This policy makes it possible that a user can see an included cross-origin image, but the JavaScript code running on the page cannot get its pixel values. In general, the policy achieves its separation goals. However, it is essential to note that browsers share some information with the embedding document for functionality reasons, and some exceptions to the policy exist. For example, the embedding document has access to the dimensions of an included image to adjust its representation. In addition, external scripts run under the scope of the embedding document's origin and not their origin. These intentional relaxations of the SOP and additional unintentional bugs make XS-Leaks possible under certain circumstances.

State on the web: A fundamental aspect of a modern web application is the ability to retain state information. As a simple example, consider requesting a URL of a social

[1]See `https://publicsuffix.org/list/` for a full list of eTLDs.

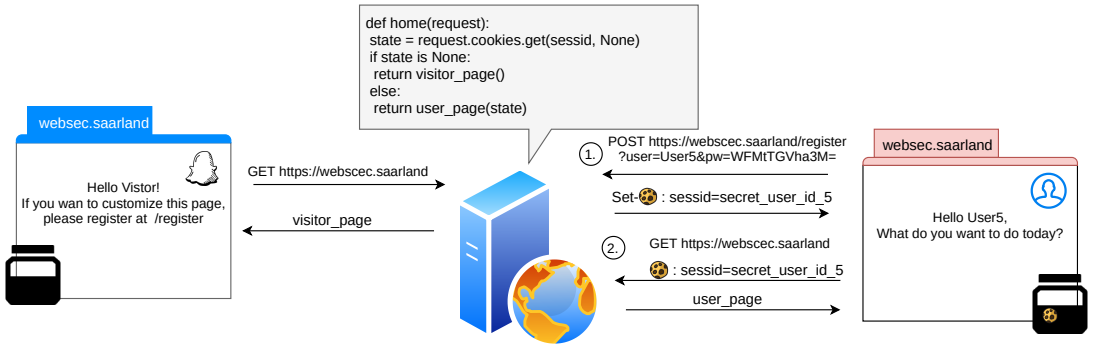


Figure 2.2: Schematic view of how a server can distinguish users on the basis of cookies. The blue browser depicts an anonymous visitor. The red browser depicts a logged-in user after finishing registration.

media platform. Such a request often returns a personal feed for a logged-in user and redirects to the login page for an anonymous visitor. To deliver conditional responses, a web server needs to know whether a user is in the logged-in state or the anonymous state. The default protocol used on the web, the Hypertext Transfer Protocol (Secure) (HTTP(S)) [34], was stateless. Thus, in the early days of the web, a web server could not distinguish between requests from different users, preventing the development of state-full applications.

A state transmission method is necessary to add state information to this protocol. This method allows web servers to determine which state a request belongs to. The default method used for transmitting and saving state information are cookies [7]. Other state transmission methods such as HTTP Authentication exist [37]. However, these other methods are negligible in practice and therefore excluded in this thesis. For cookie-based state transmission, the server usually generates a random string the first time a user visits a website and sends it along with the usual response to the browser as a session cookie via the set-cookie header. The browser saves this cookie for the current site and attaches this cookie to every request of this site. The server stores user information with this cookie and returns different responses for different users. State information can be anything, the preferred language of a visitor, whether the visitor wants to see the website in dark mode or not, or a user identifier of a logged-in user. Figure 2.2 illustrates how a server can distinguish a logged-in user from an anonymous visitor using the cookie mechanism. A visitor that never visited the page before (blue browser) will not have cookies attached to its request. A registered user (red browser) receives cookies when registering or logging in. Later, these cookies are attached to every request enabling the server to return a customized response.

Browser-related web security features: The same-origin policy (SOP) creates a minimal security basis for websites. Many websites, however, have higher security requirements.

For example, they do not want to be embeddable on other sites. This section discusses additional security features of browsers that are either active by default or that websites have to opt-in. These features are usually not primarily designed to prevent or mitigate XS-Leaks but still influence them heavily.

In recent times, browsers have been adding state partitioning as a new secure default. In a state-partitioned browser, every top-level site gets access to its resources, such as caches, instead of sharing them with all other sites. In this way, the partitioning of resources offers protection against cross-site tracking and other attacks relying on shared client-side resources [49, 64]. Another recently introduced feature is Cross-Origin-Read blocking (CORB) [102]. This feature blocks suspicious cross-origin loads before they reach the embedding website. For example, when embedding a JSON file into a script tag, the browser replaces the JSON file with an empty resource. This feature aims at mitigating micro-architectural side-channel attacks such as Spectre [51] as the resource never enters the memory the website can access. These features apply to every response from every site as they are build-in behavior in browsers.

Browsers also add information on the context of a request. A server can then check the received information and not respond with private content when receiving a request from an untrusted source. The first header to add such information was the *referer* header [35]. However, this header introduces privacy issues and can be stripped by proxies or be empty for other reasons. Thus, web servers cannot reliably use it to protect against XS-Leaks. Another header introduced as a better and less privacy-invasive header is the *origin* header [67]. However, this header is also not attached to all requests, and servers should not use it as the only defense mechanism. The third attempt at giving servers more context on received requests is the family of Fetch metadata headers [32]. If Fetch metadata is correctly used, web servers can stop XS-Leaks by not delivering state-dependent resources for untrusted requests. However, larger websites might face challenges when trying to use Fetch metadata correctly as it might be hard to define what trusted and untrusted requests are.

In addition to secure defaults and checking the request context, web servers themselves can add security headers to a response to instruct the browser to behave more safely. An old and well-known feature is the X-Frame-Options (XFO) header [80], which disallows the framing of a document. This header is helpful to prevent clickjacking attacks, where attackers trick users into clicking somewhere to perform unintended actions on their behalf. Another header is X-Content-Type-Options (XCTO) [103] which is used to opt-out of MIME type sniffing for style and script inclusions preventing some MIME Confusion attacks [48]. The Content-Security-Policy (CSP) header [15] originally developed against Cross-Site Scripting (XSS) [61] can now be used for a plethora of features (e.g., it

is also meant to replace the XFO header) and is still evolving. Unfortunately, not only is it complicated for developers to deploy correctly [81], attackers can also use it to detect if a redirect occurred or not [44]. Three new headers enhancing the same-origin policy are Cross-Origin-Resource-Policy (CORP) [33], Cross-Origin-Opener-Policy (COOP) [21], and Cross-Origin-Embedder-Policy (COEP) [20]. CORP can disable cross-origin embedding similar to CORB for additional resources such as images. COOP isolates documents opened using *window.open* from the opener and prevents the usually allowed access between opening and opened site such as postMessages. COEPs is different, as it applies to request performed on behalf of the current site and not on requests that try to embed the resource. It works by blocking all loading of cross-origin resources that do not explicitly opt into cross-origin sharing.

Cookies are the standard state transmission method used by most web applications [7]. By default, cookies get attached to every request belonging to the cookie's site. Moreover, JavaScript code running on that site can access them. The site that sets a cookie can add several flags to change the default behavior. The HttpOnly flag disallows access from JavaScript. The Secure flag makes sure that cookies are only added to HTTPS requests and not to HTTP requests. The difference between HTTP and HTTPS is that the latter uses an encrypted transportation channel. Thus nobody can read or modify the transported data while in transit between browser and web server. This flag prevents cookie hijacking attacks where network attackers try to read or modify the cookies in transit [28]. The new SameSite flag can restrict the requests to which the cookies are attached [58]. This flag is mainly meant to prevent Cross-Site Request Forgery (CSRF) [60] but also affects many other attacks, such as XS-Leak attacks. The old default behavior is *None*, meaning attach the cookies to every request belonging to a site. Another possible value is *Lax*, which only attaches the cookies to same-site requests and top-level GET requests. The last possible value is the *Strict* setting that only adds the cookies to same-site requests. Recently, some browsers switched the default to *Lax* when SameSite is not set explicitly and do not accept *None* without the Secure flag anymore [17].

2.2 XS-Leak attack overview

The goal of every XS-Leak attack is to steal user information cross-site. First, an attacker has to gather information about how a target website behaves. Then, the attacker needs to prepare an attack page based on the gathered data. Finally, the attacker has to lure victims on the created page. The attack page includes a cross-site resource of a target site and executes code to obtain information about the included resource. Depending on

the observations, the attacker then infers the victim's state on the target site, e.g., if the victim is logged in or not.

In the following, we first introduce the web attacker threat model. Then, we list the browser side-channels enabling the attacks and introduce the concept of leak channels. Then, we summarize the concept of state-dependent URLs (SD-URLs) necessary to leak user information using a leak channel. In the end, we describe why XS-Leaks are a severe issue and what harm attackers can do with the inferred information.

Web attacker threat model: We use the web attack threat model formalized by Akhawe et al. [4]. Web attackers can run web servers that deliver arbitrary content to users. An average user cannot easily notice that an attack is ongoing in the background of a site. If the malicious content hides behind good-natured content such as an online game, even professionals have problems identifying the attack. They need to open the devtools and recognize the malicious cross-site requests, which is not easy to do as most websites perform many non-malicious cross-site requests. The attacker can either lure a specific victim to their site by targeted phishing attacks or make their website popular and attack many random users, depending on their goal.

The malicious website has the same capabilities as any other website loaded in the browser. In particular, it can include cross-site resources and execute arbitrary code to gain information about the included resources. For example, the site can embed a cross-origin image and then get the dimensions of the included image. We highlight that the attack does not need the capabilities of a network attacker. Particularly the attacker cannot decrypt any HTTPS encrypted connection, perform DNS spoofing, or similar. Notably, the attacker depends on the victim visiting their malicious site to start the attack and otherwise cannot leak any information.

Leak methods: Due to the SOP, a document cannot directly access cross-site resources. However, for many inclusion types, some limited access or information sharing is possible. For example, a document embedding an external image can determine whether the image loaded correctly or not. The document can use this information to display another image or a custom error indication instead of the default broken image icon of the browser. Table A.2 in the appendix lists all leak methods considered in this thesis. For more straightforward representation, we divide the leak methods into four different groups: events fired (EF), object properties (OP), global properties (GP), and timing (T). In the following, we explain the typical characteristics of each group.

The first group is events fired (EF). For all inclusion methods using HTML tags, developers can implement handler functions that get called when a specific event occurs on the

element. For example, browsers call the load event after a resource has finished loading. The website can then execute code depending on the received event. Many of these events are only fired under certain circumstances depending on the returned response. For example, browsers only fire a load event for an image tag when they successfully render an image. For a successful image rendering, the response must contain a valid image in its body. Additional constraints such as no CORP header apply for cross-site requests. In all other cases, the browser fires an error event. Thus, one can distinguish between a response that returns an image and one that does not, using the events fired.

The second group is object properties (OP). For all inclusions methods using HTML tags, one can get a reference to the included element. For *window.open*, one can get a reference to the opened window. The IFrame element is unique, as one can get both a reference through the HTML tag and directly to the "window" inside. One cannot access all properties of the referenced object when it is cross-origin, but the browser always shares some information with the embedding site. One example is that the image dimensions can be accessed to allow dynamic adjustments on the page related to the size of the embedded image. Another example is that one can access the number of frames of a window reference.

The third group is global properties (GP). A document can access properties or add handler functions to the global window object itself. For instance, an included style sheet can change how a document looks. A document cannot directly access the rules of a cross-origin stylesheet. However, it can query the browser how it is styled using the window.getComputedStyle method and infer what rules are set by the included stylesheet in this way. Another method is the window.onerror handler a document can define. This handler gets called every time an error occurs in the main process. For example, if the document now includes a script tag and the response is not a valid script, the browser throws a parsing error, and the registered handler catches the error.

The fourth group is called timing (T). One can either measure the server and network processing time or the client processing time. For the measurements, one can use events that fire regardless of the returned content. If one measures the time between when a resource is added and when the loading is finished, one knows how long it took to load the resource, which can differ based on the state of the request. Alternatively, one can measure the time between loading is finished and parsing is finished inferring information about the size of the response.

Leak channels: For a more accessible presentation, we define the concept of leak channels. A leak channel is the instantiation of a concrete leak method and an inclusion

method. As an example, events-fired_set_img is the channel that uses the set of events fired on an image tag.

For a leak channel to work, at least two different observable outcomes for the channel must exist. Also, these outcomes must depend only on the returned responses and not on other factors. Some leak channels only have two possible outcomes, e.g., an image tag can either fire a load event when loading was successful or fire an error event otherwise. Other leak channels can have infinitely many outcomes. For example, image dimensions can be any pair of numbers.

State dependent URLs: Many URLs on the web do not deliver static content and change the response on various factors. For example, a news website has different news articles on the front page depending on the time of the visit. Some websites display a *blocked in this country* message depending on the location information of the request. Other URLs return different content depending on the requestor's state in the corresponding web application, such as whether the requestor is logged in. For XS-Leaks, one is usually only interested in the last factor, the requestor's state.

The requestor's state on a website can be whether the requestor is logged in or not. It can also be whether they are a specific user or information about their accounts, such as their age or gender. It can also be information for which no user account is necessary, such as whether someone had visited the page before or consented to the use of third-party cookies, even if they have no account. The web application usually identifies a user by the received cookies and returns the corresponding response. We define state-dependent URLs (SD-URLs) analogous to Sudhodanan et al. [92], as URLs that deliver different responses for requests of different states.

An exploitable URL in the context of XS-Leaks must necessarily be an SD-URL, but not every SD-URL can be exploited. Firstly, the attack page cannot differentiate every difference in two responses as browsers only allow limited access to cross-site resources. Secondly, the URL might be state-dependent for same-site requests only. However, due to SameSite cookies or the use of Fetch metadata or similar mechanisms, the state information is not transmitted to the server for cross-site requests, or the server does not deliver state-dependent responses in these cases.

Exploitation and impact: Figure 2.3 demonstrates all the steps of a successful XS-Leak attack. First, the attacker chooses a target application (cms.cispa.saarland) and finds URLs that return different responses based on the attacker's state. For this, the attacker tests the website in several states and observes the responses. Next, the attacker chooses

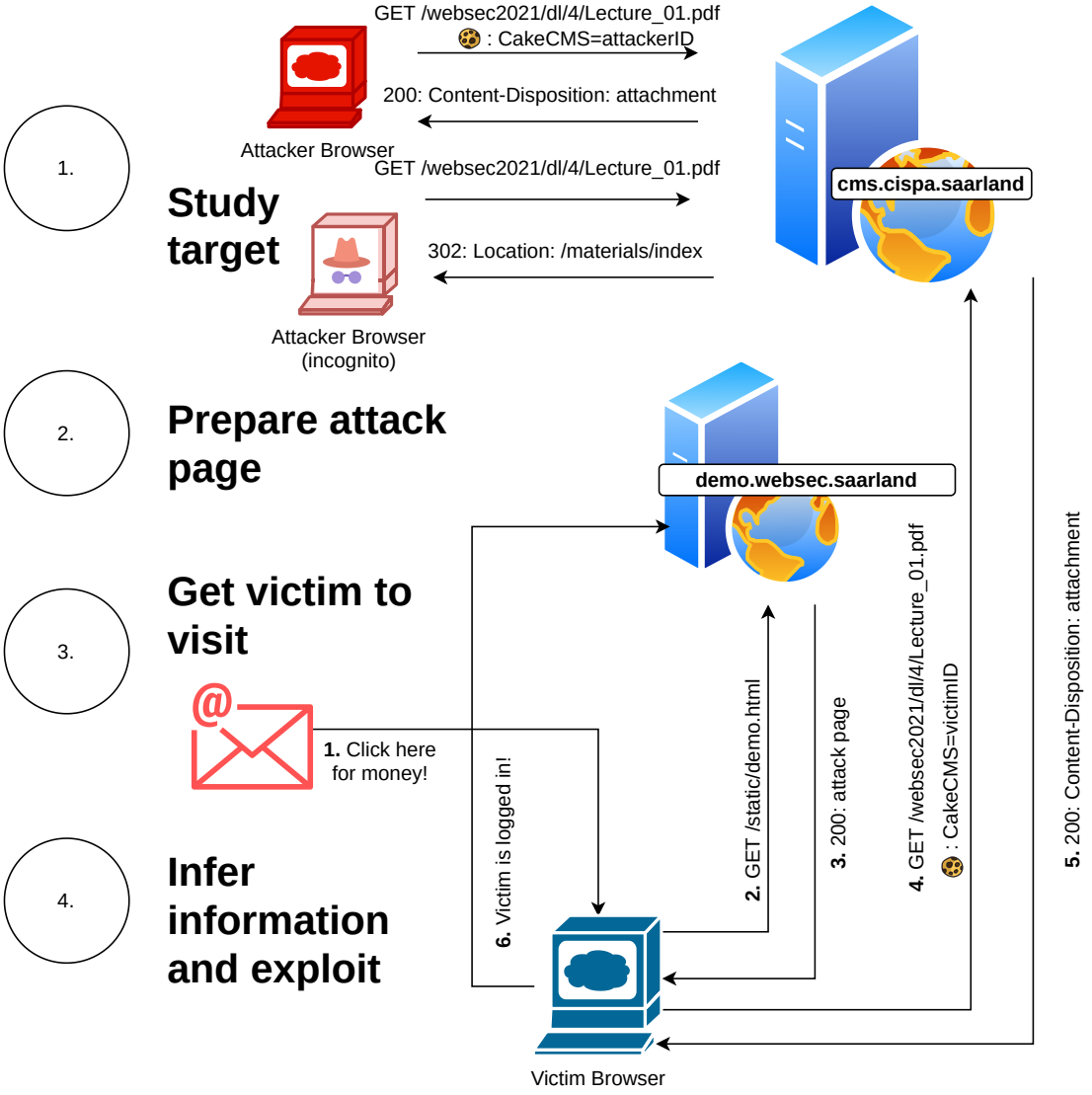


Figure 2.3: Steps of an XS-Leak attack.

a leak channel that can distinguish between the observed responses, prepares an attack page, and hosts it (demo.websec.saarland). Then, the attacker brings a victim to visit the page, e.g., by sending out a phishing mail. Finally, a victim opens the attack page that includes the target URL and infers the state information of the victim based on the browser behavior, and sends the result to the attacker.

Following the successful state inference, the attacker can use this information to achieve mischief. The impact can reach from login detection (used to perform more targeted XSS or CSRF attacks) over targeted tracking and advertisements (e.g., based on your inferred age or gender) to deanonymization (victim is the owner of account X). These attacks are especially critical on privacy-sensitive sites, such as adultery sites, where the gained information can be used for blackmailing. In oppressive countries, an attacker could be a state actor trying to identify people that visited forbidden websites. As the

attack occurs inside the browser, encryption or additional privacy measures such as a VPN [101], or the TOR network [95] do not prevent these attacks.

2.3 Browser automation

This thesis evaluates several browsers comprehensively concerning XS-Leaks and scans thousands of resources on hundreds of websites. It is not feasible to perform this scanning manually. This section explains the techniques and tools necessary to perform a fully automated XS-Leak vulnerability assessment on many websites.

Cross-browser testing and automation: To test thousands of websites in a browser in a reasonable time, one needs an automatable interface to control browsers. Such a framework should at least be able to open specific pages, click on certain elements, and modify cookies. For this purpose, different browser automation frameworks exist.

The general idea is that a driver program accepts commands sent from a client program and uses these commands to control a real browser. Developers often want to test the same website in several browsers to compare them. A framework where one can pluck and exchange the controlled browser without changing the controlling code is a massive benefit as the same code can be used to test several browsers. The Selenium project offers a framework able to control different browsers by scripts [85]. The puppeteer project is another framework to control browsers. However, it only fully supports Chrome as of now [69].

Authenticated crawling: To study the state of the web, researchers need to find and collect resources belonging to many websites. For this purpose, they use web crawlers. Often, the crawler visits a landing page and clicks on all links based on some parameters. Parameters are usually depth-first or breadth-first approach and stop-conditions such as time or number of URLs. Such crawlers mainly find HTML documents. However, all subresources, including scripts, images, or audio files, are potentially exploitable SD-URLs. Putting a proxy between the crawler and the web, all responses, including the ones belonging to the subresources, can be saved. In addition to intercepting the messages, a proxy can also replay and modify requests, e.g., dropping or appending cookies from requests.

To find SD-URLs, the researcher first needs to create state information on target websites and access resources in different states. The considered state is usually logged-in state and anonymous state. This post-login crawling is a challenging problem as it needs

credentials for every tested website and a way to perform the login. Several different methods are available to perform this task. First, one can manually register and login accounts and save and replay the login flows, but this approach does not scale to a large number of websites [55, 92]. Second, one can use single sign-on (SSO) services to log in on the target website using an account of a supported identity provider [104]. A third option is to use manually created or crowd-sourced credentials and a tool to perform the login automatically [47]. The last option is to use a tool that handles registration as well as login fully automatically [28]. These automated options usually rely on a set of heuristics to find login and signup forms, fill them out accordingly, and then check whether the registration and login worked.

Chapter 3

XS-Leaks in browsers

The previous chapter described what an XS-Leak is and how an attacker could abuse a leaky URL. This chapter investigates how XS-Leaks behave in different browsers in more detail.

Most reports introducing new XS-Leak methods only provide two distinguishable responses, an ad-hoc summary of when the leak method works (e.g., the method can distinguish different sized images), or a currently vulnerable SD-URL that can change anytime. These reports prove that an XS-Leak exists. However, their unsystematic approach leads to several issues described in the following.

First, researchers studying an XS-Leak method must find two distinguishable responses based on the provided information. Depending on the quality and age of the report, this can take some time. Even if the report provides two responses, this response pair might not work anymore in an up-to-date version of a browser. The researcher cannot be sure whether browsers fixed the leak method entirely, whether they used incorrect responses, or have an error in their code trying to leak the difference.

Second, such an approach provides no information about how widespread and severe the issue is. In particular, it does not answer the question "Are only these exact two responses vulnerable or all possible responses that differ in property X?". This question is essential to estimate how many response pairs might be vulnerable on the web. It is unlikely to find the exact response pair on an actual website but likely to find response pairs that differ in property X.

Third, browser vendors do not get information on where the underlying cause of the issue might be and how to fix it in their code. If, instead, a report includes that everything except for the value of a property X is irrelevant for the leak to work, this hints that the issue is in the code handling property X. Also, if all browsers behave the same, it hints

that the leak follows the HTML specifications and that the browser vendors cannot easily fix it. If, however, the browsers behave vastly different, or one browser is not vulnerable at all, this suggests that this XS-Leak method primarily relies on unspecified edge case behavior and might be easily fixed by the vendors.

In this chapter, we develop a more systematic approach to overcome the above three issues and answer the question of "Which groups of responses can different leak methods distinguish in different browsers?". The answer to this question does not only help browser vendors to fix underlying issues more efficiently and help researchers studying XS-Leaks, but it also allows for the development of a static XS-Leak checker tool. This tool outputs all leak methods that can distinguish two given responses without rerunning a test for every method. Thus, this tool will be much faster than dynamically confirming every single endpoint of a website for every known method to check if a website is exploitable. It can also be used as an educational tool for developers, warning them about dangerous practices.

The structure of this chapter is as follows. The first section defines the scope, i.e., which methods and responses we tested in which browsers. Then, the second section explains the framework built to answer the research question. Next, the third section evaluates the created framework. Finally, the last section presents the results.

3.1 Scope

There exist at least 225 different browsers [14] and some have more than 90 major releases [62]. Additionally, researchers found dozens of different XS-Leak attack instances over the years [88]. Doing a comprehensive test of all existing XS-Leaks in all browsers and versions is undoubtedly infeasible. Therefore, in this thesis, we aim to cover as much as possible, focusing on aspects relevant to a large share of users and websites while staying within feasible bounds.

3.1.1 Tested browsers

Even though over 225 browsers were created over time, only a small number of them have a significant market share today. Issues found in these main browsers affect many users, and focusing on them has a higher impact than focusing on a large set of mostly unused browsers.

In July 2021, the four major desktop browsers (Chrome, Safari, Edge, and Firefox) had a combined market share of over 93% [23], and most people used an up-to-date version [24].

This fact can be explained by the silent auto-update behavior active in most browsers [30]. We cannot test Safari, only available for macOS, because we only have a Linux server available to run all tests. We decided to test the other three major browsers in the most recent version when starting this project to cover a large share of web users. We describe the details of the exact versions and other settings in appendix B.

Web security studies often use headless browsers, i.e., browsers without graphical users interfaces, as they are faster and need less memory. The pretests found that some XS-Leak behavior (e.g., window.onblur) differed between headful and headless browsers, which is in line with prior findings showing inconsistencies between headless and headful modes of browsers [66]. As we aim to evaluate end-user security, we only consider headful browsers.

3.1.2 Tested leak methods

Many different leak methods were discovered over time [42, 41, 55, 54, 43] starting with the first mention of timing attacks on privacy in the year 2000 [31]. Both the work of Sudhodanan et al. and the collaborative wiki of Sousa et al. attempt to give a comprehensive overview of the known XS-Leaks methods by collecting all available resources and information in one place [92, 88]. The methods listed there, including the ones currently only in the GitHub repository[1], provide a broad baseline of available methods. We use a combined list of methods as the starting point of this research and aim to find variants of these previously known methods. However, not all methods are currently automatically testable. In the following, we provide an overview of the tested methods and excluded methods.

Tested methods: The research in XS-Leaks is still fragmented, and no commonly agreed-on taxonomy exists. The reference works of COSI and the XS-Leaks wiki both primarily classify attack classes by what response property they can distinguish, e.g., CORP leaks or events-fired_content-type_mismatch_script [88, 92]. Unfortunately, this response-centric approach has two shortcomings. First, the same property can often be distinguished by several methods. Second, many methods can distinguish more than one property. As a result, this approach can lead to a high number of leak classes and general confusion. Instead, we focus on the browser side-channel responsible for the information leakage, e.g., events fired on an HTML tag or errors observed using the global onerror handler. As described in chapter 2.2, we group all browser side-channels into four groups: events fired (EF), object properties (OP), global properties (GP), timing (T). Except for

[1] `https://github.com/xsleaks/wiki/issues`

timing, we test every leak method for all inclusion methods, even when it should not work. For example, we check the element dimensions on a script tag. This approach allows us to maintain a smaller codebase and not miss strange edge cases. In total, we test 30 leak methods for 12 inclusion methods described in more detail in appendix A.

Excluded methods: The most representative evaluation would study all known methods. However, not all methods are currently automatically testable or do not fit into our testing approach. Therefore, we decided to only test methods that are reliably testable in a fully automatic manner and do not involve too high engineering hurdles.

Timing and caching-related XS-Leaks are difficult to test at scale as they require timing baselines. Additionally, much research already exist [38, 98, 82]. For these reasons, we decided to excluded them from this work. We also exclude XS-Leaks based on CSS requiring user interaction [57, 50] as they require a user-agent that can do more than just opening a URL. Lastly, we exclude XS-Leaks relying on deprecated features such as AppCache [54]. For a more detailed discussion on how we could add these groups of XS-Leaks in the future, we refer to the discussion in chapter 5.

3.1.3 Tested responses

As described earlier, we want to compute the exact conditions under which an XS-Leak method works in different browsers. To compute these conditions, we need to define a search space of possible conditions and find a computational approach to calculate the exact conditions. The following describes the constructed search space, i.e., all HTTP responses we consider, and the general approach to compute the exact conditions of a leak channel.

Response space: There are infinitely many possible Hypertext Transfer Protocol (Secure) (HTTP(S)) responses as the body content can be arbitrary bytes, and most headers are strings that can be combined in almost any way. Unfortunately, this makes testing the entire space of responses impossible as it is infinitely large. Luckily, past research on XS-Leaks, specifications, and initial experiments, indicated that XS-Leaks rely only on a small set of properties, so not every obscure combination has to be tested as most of them should not influence XS-Leak behavior. The relevant properties are the type of content (e.g., image or non-image), status-code, and relevant headers. The search space constructed is the complete combination of all the considered properties in table 3.1 leading to a total of 387,072 possible responses. We note that we might miss some relevant properties or values with the construction of this limited search space, but we

Property	Count	Options	Notes
Status-Code	63	[100, 101, 102, 103, 200, 201, 202, 203, 204, 205, 206, 207, 208, 226, 300, 301, 302, 303, 304, 305, 307, 308, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 421, 422, 423, 424, 425, 426, 428, 429, 431, 451, 500, 501, 502, 503, 504, 505, 506, 507, 508, 510, 511, 999]	The 62 IANA defined ones [46] and one invalid code 999
Body	12	[ecocnt_html=num_frames=1,input_id=test1, ecocnt_html=num_frames=2, ecocnt_html=post_message=mes1, ecocnt_html=meta_refresh=0;http://172.17.0.1:8000, ecocnt_css=h1 {color: blue}, ecocnt_js=.„., ecocnt_js=var a=5;, ecocnt_img=width=50,height=50,type=png, ecocnt_vid=width=100,height=100,duration=2, ecocnt_audio=duration=1, ecocnt_pdf=a=a, empty]	Special syntax to request content from the echo application
Content-Type	8	[text/html, text/css, application/javascript, video/mp4, audio/wav, image/png, application/pdf, empty]	
Content-Disposition	2	[attachment, empty]	
Location	2	[http://172.17.0.1:8000, empty]	
X-Frame-Options	2	[deny, empty]	
X-Content-Type-Options	2	[nosniff, empty]	
Cross-Origin-Resource-Policy	2	[same-origin, empty]	
Cross-Origin-Opener-Policy	2	[same-origin, empty]	

Table 3.1: Considered properties and options of the response space.

think we have made a good compromise in covering most of the relevant response space while staying feasible. More discussion on the adequacy of the constructed search space follows in chapter 5.2.1.

Table 3.1 lists the number of options for every considered property. For every property, we use at least one positive and one negative value. A positive value could be that the body is a valid image or COOP prevents cross-site pages from having a reference of an opened site. A negative value could be that the body is not a valid image or that COOP does not prevent access to the reference. For all headers and body, we use the special value *empty*, i.e., header not set, or body contains 0 bytes, as the default option. Many of these headers have more than two specified values. Often one value only allows access from the same origin, whereas another value relaxes this to the same site or a specified site. However, in the threat model we consider, the attacker can only host the attack page cross-site. So, from an attacker's perspective, all these values are negative, and it suffices only to test one of these possible values as they should all behave the same on a malicious site.

Exact conditions of XS-Leak methods: There are two general approaches to compute the exact conditions under which an XS-Leak method works within the constructed search space. The first approach starts with two responses that lead to different outcomes, then modifies one of the responses and observes the outcome. For example, one can

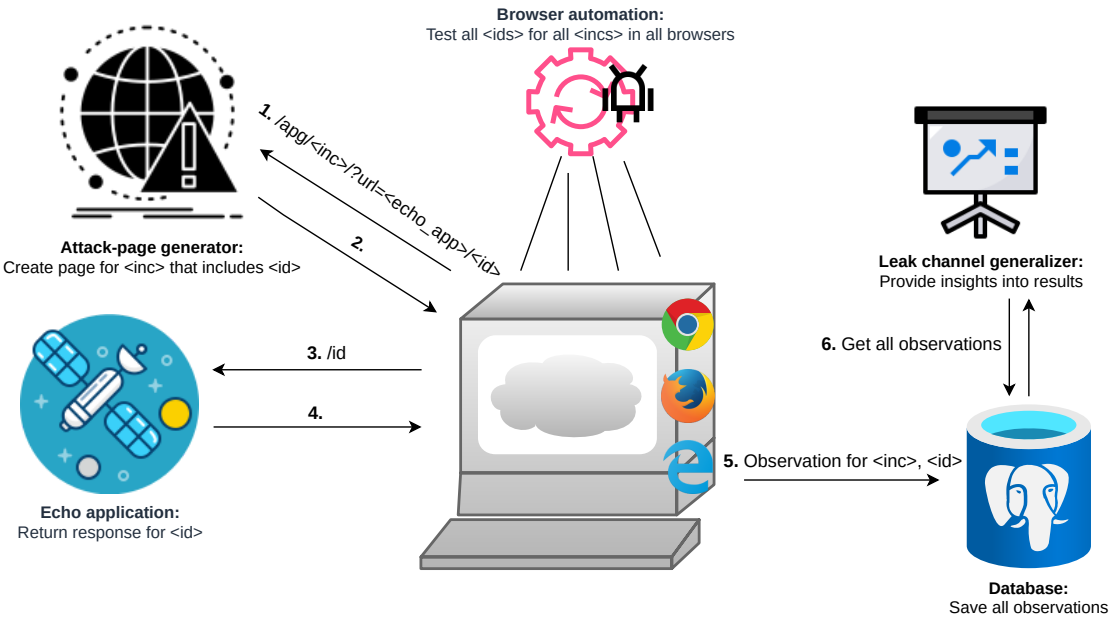


Figure 3.1: Overview of the test browser framework.

add a property, remove a property, or change its value. The idea behind this approach is that if one modifies property A and still observes the same outcome, one can ignore this property. The problem with this approach is that there is no general algorithm one could apply and that there are problems with disjoint groups of responses leading to the same outcome. The second approach is to test the complete space of responses, group responses that generated the same outcome together, and see what the responses in a group have in common. Depending on the algorithm used, this approach can find several disjoint groups leading to the same outcome and test several methods simultaneously. For these reasons, we chose the second approach and tested all leak methods for all responses in the constructed response space.

3.2 Test browser framework

The previous section described the constructed response space and that it is necessary to test every response for every inclusion method for every tested browser. Therefore, a suitable framework is needed to perform millions of tests efficiently. Additionally, a methodology to interpret the results is needed. We created the test browser framework that can automatically test all XS-Leak methods in several browsers over the entire created response space. The framework also generalizes the results using machine learning techniques.

Figure 3.1 gives an overview of the architecture of the framework. The main parts are the echo application representing the response space, the attack-page generator responsible

for creating the attack pages, and the browser automation setup, which visits all attack pages in several browsers. Finally, the leak channel generalizer summarizes all groups of responses that generate the same observations together in a meaningful way.

The general sequence of a test is that an automated browser first requests an attack page for an inclusion method and response id from the attack-page generator. The attack page contains code to observe the outcomes of all leak methods and includes one cross-site resource from the echo application using the specified inclusion method. Then, the automated browser loads the attack page, leading the browser to perform a request to the echo application. The echo application will generate a response based on the received id and return it. The code on the attack page logs all observations, such as which events fired, and sends the results to a database server. Then, the automated browser will request the next attack pages. After all tests are executed, the leak channel generalizer groups all responses that generate the same outcome together and provides a helpful summary.

3.2.1 Echo application

The echo application is responsible for delivering the 387,072 responses of the response space. We constructed it out of two parts. The echo part can return various content bodies and headers by reflecting query parameters to the sender. The leaky part maps all responses of the constructed response space to an id and, when that id is requested, returns the corresponding response by requesting it from the echo part.

Using this application, we can request resources from the application at `/leaks/<id>/`, observe and save the results and later quickly look up what the exact content of a response was. The application uses the Django framework [26] and we deployed it using uWSGI [97] to allow for many parallel connections and support HTTPS. Testing on HTTPS is necessary, because some investigated features such as the Cross-Origin-Opener-Policy require a secure origin [21].

3.2.2 Attack-page generator

The attack-page generator is responsible for delivering the attack pages that perform an XS-Leak attack. It has one endpoint that is called with one inclusion method and one URL. The attack-page generator creates an HTML page where the given URL is embedded according to the requested inclusion method. Additional parameters such as timeouts or the database back-end URL can be specified. Otherwise, it will use built-in default values.

With this implementation and the choice of testing the entire response space, it is possible to test all XS-Leak methods belonging to one inclusion method simultaneously. In total, 13,934,556=12(inclusion method)*387,072(responses)*3(browsers) attack pages are necessary for the complete comprehensive evaluation of the constructed response space. The application uses Django and the Django template language to create the attack pages with a small and maintainable code-base. For the tests, we deploy the application with uWSGI to handle many parallel connections.

3.2.3 Browser automation

An automation methodology is necessary to test all responses with all inclusion methods in all browsers. The automated browser requests every test URL of the attack-page generator consisting of an inclusion method and response id. An example of such a URL is `https://<apg_base>/apg/<inc>/?url=<leaky_base>/leaks/<id>/`. After the attack page successfully gathered and saved the observations, we test the following id. If a test takes too long, we record it as a timeout. Timeouts can occur if the browser or some other part of the framework crash. For some tests, additional steps are necessary before the dynamic automator continues with the following URL. For example, in the case of the download bar detection technique [88], the download bar has to be closed before continuing to the next test.

We use Selenium 4 in the dynamic grid configuration [86] to manage the browsers. In this configuration, every browser is started in its individual docker container [27]. The dynamic grid provides complete isolation between browsers and runs. However, it requires considerable overhead to create and start the docker containers. A python script coordinates the automated tests and starts around 50 browsers to run tests in parallel. We include more details about the used parameters and versions in appendix B.

3.2.4 Leak channel generalizer

The test browser framework generates large amounts of data while performing its 13,934,556 tests. For every leak channel tested, we want to find the common characteristics of responses that result in the same observation. If we have a short description of when a response generates a result, we can expect this to generalize to unseen responses. For example, suppose the description says that as long as the response contains the Cross-Origin-Resource-Policy header, an image inclusion will fire the error event. In that case, this will likely apply to responses outside the constructed response space, such as responses with an unconsidered content-type.

It is easy to get all responses that result in the same outcome for one channel. As we test the entire test space, this is just a database lookup. However, the result that 200 thousand responses result in a load event and the other 200 thousand responses result in an error event does not provide much insight. Therefore, we need an algorithm that summarizes each channel with short representations of each outcome's common features to generate more meaningful insights.

This work uses decision trees in the implementation of h2o random forests as the algorithm to create meaningful representations of the results [25]. Decision trees are good at removing variables that do not provide any information, e.g., irrelevant headers in this case, and can handle several independent paths leading to the same result. If the created decision trees are not too large, they are simple to interpret and come in a representation one can use to predict the outcome of unseen responses. On the other hand, if the created decision trees are large, this suggests that the leak channel is unstable as the trees could not find short characteristics of responses that result in the same outcome.

3.3 Test browser framework evaluation

We used the test browser framework to perform 13,934,556 tests to cover the entire response space constructed. The gathered data shows that several of these tests failed and that different inclusion methods take varying average times in several browser families. Additionally, we can only consider a channel reliable if it always results in the same observation for the same response. Therefore, to check that a leak channel works reliably, we rerun 70,000 tests to evaluate reliability. In the following, we evaluate the time taken by the inclusion methods, the occurred timeouts, and the reliability of the XS-Leak methods.

3.3.1 Timing, timeouts and impossible results

Additionally to the leak methods observations, we also log general timing information. One finding is that chromium-based browsers fire the load event on a page before finishing the parsing and error handling of all subresources. Whereas, Firefox only calls the load event after it finishes parsing and error handling. This knowledge is critical for security researchers as they cannot rely only on the load event for specific tests in chromium-based browsers but need to wait longer. To account for this, we added a timeout period after the load event is received into the test infrastructure and restarted the testing framework. Table 3.2 displays the results of the timing information after adding a delay of at least 150ms before submitting the results. We excluded MicrosoftEdge from the table as the

		Loading time				Completion time			
		mean	std	min	max	mean	std	min	max
Inclusion method	Browser								
audio	Chrome	39.46	15.07	9	959	193.91	15.44	163	1112
	Firefox	59.65	19.04	11	1065	220.12	20.21	170	1248
embed	Chrome	88.98	60.94	11	1052	244.17	60.93	167	1208
	Firefox	51.36	22.72	9	964	212.13	23.18	165	1121
embed-img	Chrome	22.07	19.01	7	960	176.77	19.25	160	1115
	Firefox	51.24	21.52	7	1294	211.34	22.01	163	1452
iframe	Chrome	90.84	45.07	14	958	474.94	46.51	397	1372
	Firefox	130.35	88.11	13	1199	518.61	87.10	399	1619
iframe-csp	Chrome	87.36	33.45	14	996	471.01	36.65	395	2381
	Firefox	125.25	88.01	10	1333	514.43	88.80	395	2557
img	Chrome	15.99	12.87	4	953	170.41	13.25	157	1109
	Firefox	51.41	17.47	9	1009	211.46	18.11	168	1177
link-prefetch	Chrome	4.05	2.05	1	127	158.43	2.95	153	284
	Firefox	7.05	3.90	1	168	167.55	5.94	157	344
link-stylesheet	Chrome	15.40	18.11	3	950	169.84	18.40	156	1104
	Firefox	49.33	16.60	8	1076	209.41	17.22	164	1236
object	Chrome	39.86	33.12	12	981	195.87	33.12	167	1188
	Firefox	52.83	22.70	9	1157	212.91	23.11	166	1323
script	Chrome	18.07	39.06	3	964	172.65	39.29	156	1117
	Firefox	52.07	18.17	11	1175	212.52	18.93	168	1337
video	Chrome	41.04	19.84	16	971	195.46	20.09	169	1124
	Firefox	64.51	21.36	22	1602	225.23	22.53	180	1799
window.open	Chrome	14.82	5.64	6	207	1631.25	458.46	386	2314
	Firefox	54.68	11.31	23	382	1266.36	235.73	412	2650

Table 3.2: Loading and completion times for all inclusion methods in milliseconds.

results are similar to Chrome. The table contains the time it takes the browser to call the load event on the attack page, and the time it took the attack page to finish executing. In addition, we found that chromium-based browsers are slower in completely handling *window.open*. *Window.open* takes the longest time of all inclusion methods, and it is hard to decide how long one should wait, as it is impossible to know when the opened window finished loading, as one does not have access to the load event of opened windows.

Initially, 42,602 out of the 13,934,556 tests, i.e., roughly 0.003%, timed out roughly evenly split over all three browsers. The attack page waits for the load event [65], then executes code to gather information about the included cross-site resource and then sends a request with the observed data to a database server. If the request to the database does not finish in a configured time, we save the test as a timeout. Timeouts can happen for many reasons, primarily related to the testing infrastructure. Therefore, we retest all tests that timed out. After two rounds of retesting the timed out ones, only 4,496 URLs in Firefox remain without results. An investigation discovered that this is due to two bugs in Firefox preventing these pages from ever successfully loading. The first bug is that IFrames get stuck loading if the status-code is 101 or 304, the content-type

is application/pdf, and the body is not empty. The second bug is that many inclusion methods reload a URL infinitely under specific conditions. One example is status-code 203, content-type video/mp4, and empty body for top-level navigation. For more details, we refer to the two created bug reports [77, 78].

3.3.2 Reliability evaluation

One can only successfully exploit a leak channel if the leak channel always results in the same observations for the same responses. Unfortunately, not every channel is reliable. For example, other uncontrolled factors not depending on the returned responses can influence the tests resulting in *random* observations. For example, the load event could not be fired in time because the response got delayed due to a global hiccup and not because the browser would not fire a load event for the response. We retest a random sample of URLs to measure the reliability of the leak channels in the testing infrastructure. A channel is regarded as reliable only if we observe the same results in the original runs and the reruns for most tests. For this, we rerun 70,000 tests, i.e., roughly 1,950 retests per browser and inclusion method combination. Many channels are perfectly reliable and always result in the same observation. Other channels are mostly stable with some rare incorrect results. At the same time, some channels are volatile and give different results in many runs. The instability has several reasons explained in the following.

Some instability is due to problems in the testing infrastructure. For example, the global-properties_downloadbarheight method requires the precondition that the download bar is closed before the test starts. In Selenium, there is no method to close the download bar directly. By design, one can only interact with the web page and not with the browser's user interface [96]. To close the download bar using the available features, we visit `chrome://downloads` and then execute JavaScript on the page to click on every *X* button to remove the download bar. Unfortunately, this method does not work every time. In addition, the WebDriver automated browsers display a notice that the browser is automated [100]. Chrome and Edge do this by displaying a banner at the top. Unfortunately, this banner sometimes does not appear or disappears randomly, leading to incorrect observations. A solution for the first problem would be to open a new browser for every single test. However, as every browser is isolated in its docker container by the dynamic grid, this would lead to unacceptably high overhead costs. Additionally, it would not change anything for the second problem.

Other instabilities seem to be connected to insufficient waiting time, e.g., if a postMessage is delayed, the test might miss the message if it does not wait long enough. We mainly

observed this phenomenon for *window.open*, but occasionally also for IFrame and link-prefetch. One solution would be to use longer waiting times. However, it is unclear what a suitable waiting time is. The higher the timeout, the longer the tests take, and the more difficult it would be to exploit in the real world as a victim might already have closed the malicious website. In the initial results, all leaks based on the *window.open* inclusion method were unstable. We then realized that the initial timeout period of two seconds for a test is not always enough for the *window.open* inclusion method and redid all *window.open* test with a new timeout period of four seconds. After this adjustment, most XS-Leak methods using *window.open* were stable.

For other methods, we could not find any explanation. These methods appear to behave randomly, e.g., the window.onblur method in Firefox appears to fire randomly unrelated to any of the controlled factors.

We argue that the unstable methods are hard to exploit for an attacker in the real world. For example, a user might already have the download bar open, or several tries would be needed to make sure the result is correct. Hence, we remove the unstable methods from the following tests.

3.4 Leak channel results

Finally, we present the results for the main question: "Which groups of responses can different leak methods distinguish in different browsers?". We calculated how many different outcomes a channel has and how many responses resulted in every outcome for all leak channels. Table 3.3 shows the binarized results of the outcomes. The columns represent all tested inclusion methods, the rows represent all tested leak methods, and the cells represent all leak channels for both browsers. Every cell with a ● represents a leak channel that has more than one observation, and every cell with a ◯ represents a leak channel that only has a single observation. Except for some cases, such as the third method *global-properties_downloadbarheight*, which only has more than one observation in Chrome, most channels have the same number of possible observations in both browsers. We ignored all channels that only have one outcome as they do not provide any information. For example, the last column shows that many leak methods such as event-list cannot work for *window.open*. The other channels can work or behave randomly, as seen in the previous section. Therefore, we created decision trees to summarize all reliable channels and manually analyzed them to find interesting patterns. Using this strategy, we found several discrepancies in browsers and bugs described in the following. We also present the created trees and how one can use them as a static pruning tool.

Leak method	Browser	audio	embed	embed-img	iframe	iframe-csp	img	link-prefetch	link-stylesheet	object	script	video	window.open
	Inclusion method												
event-list	Firefox	●	●	●	●	●	●	●	●	●	●	●	○
	Chrome	●	●	●	●	●	●	●	●	●	●	●	○
event-set	Firefox	●	●	●	●	●	●	●	●	●	●	●	○
	Chrome	●	●	●	○	○	●	●	●	●	●	●	○
gp-download-bar-height-bin	Firefox	○	○	○	○	○	○	○	○	○	○	○	○
	Chrome	○	●	●	●	●	○	○	○	●	○	●	●
gp-securitypolicyviolation	Firefox	○	○	○	○	●	○	○	○	○	○	○	○
	Chrome	○	○	○	○	●	○	○	○	○	○	○	○
gp-window-getComputedStyle	Firefox	○	○	○	○	○	○	○	●	○	○	○	○
	Chrome	○	○	○	○	○	○	○	●	○	○	○	○
gp-window-hasOwnProperty	Firefox	○	○	○	○	○	○	○	○	○	●	○	○
	Chrome	○	○	○	○	○	○	○	○	○	●	○	○
gp-window-onblur	Firefox	●	●	●	●	●	●	●	●	●	●	●	●
	Chrome	●	●	○	●	○	○	○	○	○	●	○	●
gp-window-onerror	Firefox	○	○	○	○	○	○	○	○	○	●	○	○
	Chrome	○	○	○	○	○	○	○	○	○	●	○	○
gp-window-postMessage	Firefox	○	●	●	●	●	○	○	○	●	○	○	●
	Chrome	○	●	○	●	●	○	○	○	●	○	○	●
op-el-buffered	Firefox	●	○	○	○	○	○	○	○	○	○	●	○
	Chrome	●	○	○	○	○	○	○	○	○	○	●	○
op-el-contentDocument	Firefox	○	○	○	●	●	○	○	○	○	○	○	○
	Chrome	○	○	○	●	●	○	○	○	●	○	○	○
op-el-duration	Firefox	●	○	○	○	○	○	○	○	○	○	●	○
	Chrome	●	○	○	○	○	○	○	○	○	○	●	○
op-el-height	Firefox	○	○	○	○	○	●	○	○	○	○	○	○
	Chrome	○	○	○	○	○	●	○	○	○	○	○	○
op-el-media-error	Firefox	●	○	○	○	○	○	○	○	○	○	●	○
	Chrome	●	○	○	○	○	○	○	○	○	○	●	○
op-el-naturalHeight	Firefox	○	○	○	○	○	●	○	○	○	○	○	○
	Chrome	○	○	○	○	○	●	○	○	○	○	○	○
op-el-naturalWidth	Firefox	○	○	○	○	○	●	○	○	○	○	○	○
	Chrome	○	○	○	○	○	●	○	○	○	○	○	○
op-el-networkState	Firefox	●	○	○	○	○	○	○	○	○	○	●	○
	Chrome	●	○	○	○	○	○	○	○	○	○	●	○
op-el-paused	Firefox	○	○	○	○	○	○	○	○	○	○	○	○
	Chrome	○	○	○	○	○	○	○	○	○	○	○	○
op-el-readyState	Firefox	●	○	○	○	○	○	○	○	○	○	●	○
	Chrome	●	○	○	○	○	○	○	○	○	○	●	○
op-el-seekable	Firefox	●	○	○	○	○	○	○	○	○	○	●	○
	Chrome	●	○	○	○	○	○	○	○	○	○	●	○
op-el-sheet	Firefox	○	○	○	○	○	○	○	○	○	○	○	○
	Chrome	○	○	○	○	○	○	○	○	○	○	○	○
op-el-videoHeight	Firefox	○	○	○	○	○	○	○	○	○	○	●	○
	Chrome	○	○	○	○	○	○	○	○	○	○	●	○
op-el-videoWidth	Firefox	○	○	○	○	○	○	○	○	○	○	●	○
	Chrome	○	○	○	○	○	○	○	○	○	○	●	○
op-el-width	Firefox	○	○	○	○	○	●	○	○	○	○	○	○
	Chrome	○	○	○	○	○	●	○	○	○	○	○	○
op-frame-count	Firefox	○	○	○	●	●	○	○	○	○	○	○	●
	Chrome	○	○	○	●	●	○	○	○	○	○	○	●
op-win-CSS2Properties	Firefox	○	○	○	●	●	○	○	○	○	○	○	●
	Chrome	○	○	○	●	●	○	○	○	○	○	○	●
op-win-history-length	Firefox	○	○	○	●	●	○	○	○	○	○	○	●
	Chrome	○	○	○	●	●	○	○	○	○	○	○	●
op-win-opener	Firefox	○	○	○	○	○	○	○	○	○	○	○	●
	Chrome	○	○	○	○	○	○	○	○	○	○	○	●
op-win-origin	Firefox	○	○	○	●	●	○	○	○	○	○	○	●
	Chrome	○	○	○	●	●	○	○	○	○	○	○	●
op-win-window	Firefox	○	○	○	●	●	○	○	○	○	○	○	●
	Chrome	○	○	○	●	●	○	○	○	○	○	○	●

Table 3.3: Matrix of all tested leak methods and inclusion methods.

MicrosoftEdge behaves almost identical to Chrome and is omitted for brevity.
○: Only one observation exists. ●: At least two observations exist.

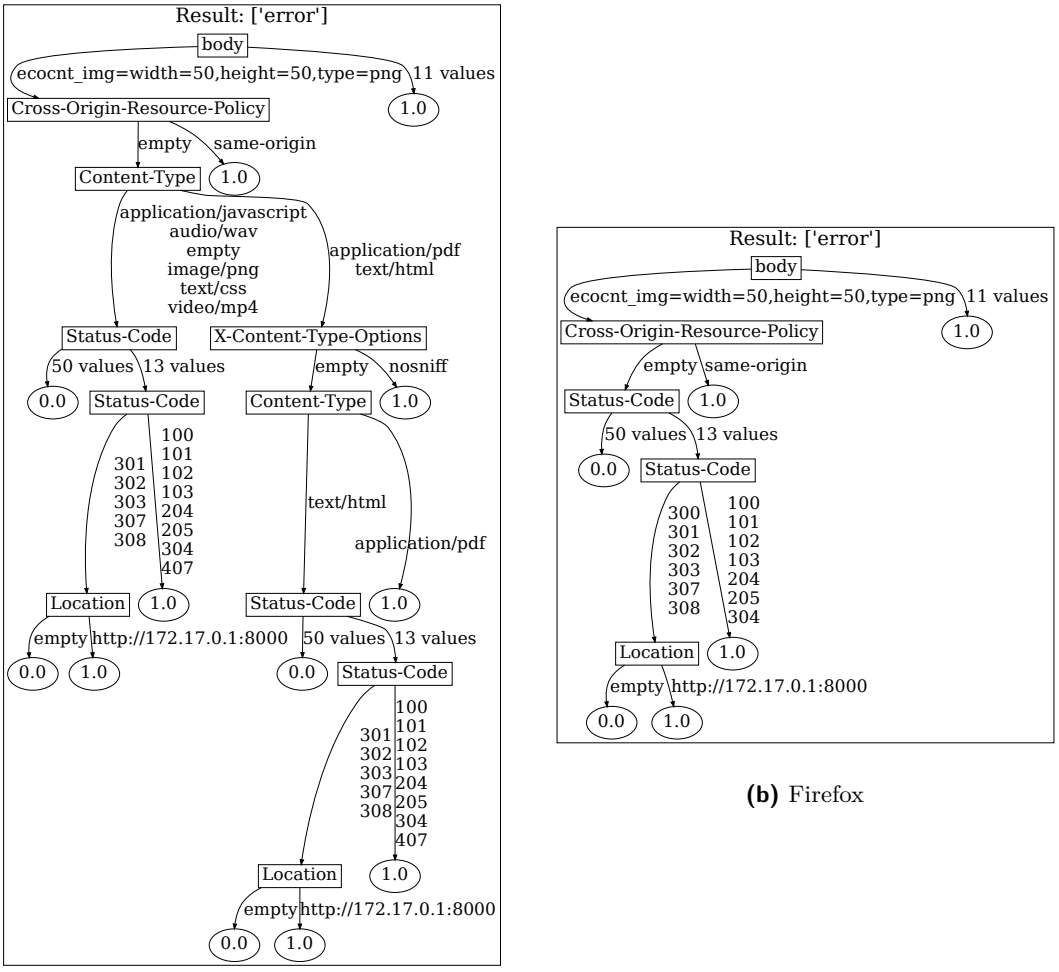
3.4.1 Browser inconsistencies

The first result is that Chrome and MicrosoftEdge behave identically except for the unstable global-properties_downloadbarheight method that is unreliable in Chrome and never works in other browsers. This observation suggests that almost all code responsible for XS-Leaks belongs to Chromium and not the code changed by Chrome and MicrosoftEdge. Consequently, we removed MicrosoftEdge from further tests. Another observation is that Chrome and Firefox behave quite differently, but it is not always clear which browser behaves correctly. For example, object and embed behave almost identical to IFrame in Chrome, which is not the case for Firefox. However, this is known, and Firefox might adjust its behavior in the future [99]. Another example, where it is even less clear what the correct behavior is, is that Firefox performs redirections for responses with status-code 300, whereas Chrome does not. Websites could use this behavior to detect which browser a user is using, even if JavaScript is disabled.

3.4.2 Working leak channels and trees

Most leak channels work more or less as expected. However, there are significant variations between the browsers. For example, which status-codes are allowed or which content-types are allowed differs for many methods. In the following, we present two instances of the created decision tree and show which differences between the browsers exist in the handling of edge-case behavior.

Figure 3.2 presents the decision trees created for events-fired_set_img. This leak channel observes the events fired on image tags. The image tag either fires a *load* event when an image was loaded successfully or an *error* event otherwise. The leaf nodes in the tree with *1.0* belong to responses that fire an *error* event, and the leaf nodes with *0.0* belong to responses that fire a *load* event. The tree for Chrome is located on the left and the tree for Firefox on the right. In general, both browsers fire *load* for a successful image load and *error* otherwise. However, the definitions of a successful load are slightly different. Most importantly, the body needs to contain a valid image for a successful load. However, even when the body contains a valid image, browsers do not fire a load event under several circumstances. Both browsers do not load the image if CORP is set or the status-code prevents loading. Both browsers do not load for status-codes 100, 101, 102, 103, 204, 205 and 304. Chrome additionally fails for status-code 407. They also do not load the image if the status-code is a redirection code and a valid location header redirects to a non-image location. Firefox accepts one additional code for redirections: 300. In Chrome, the load also fails if the content-type header is application/pdf or if the XCTO header is set and the content-type header is text/html. The CORB implementation of



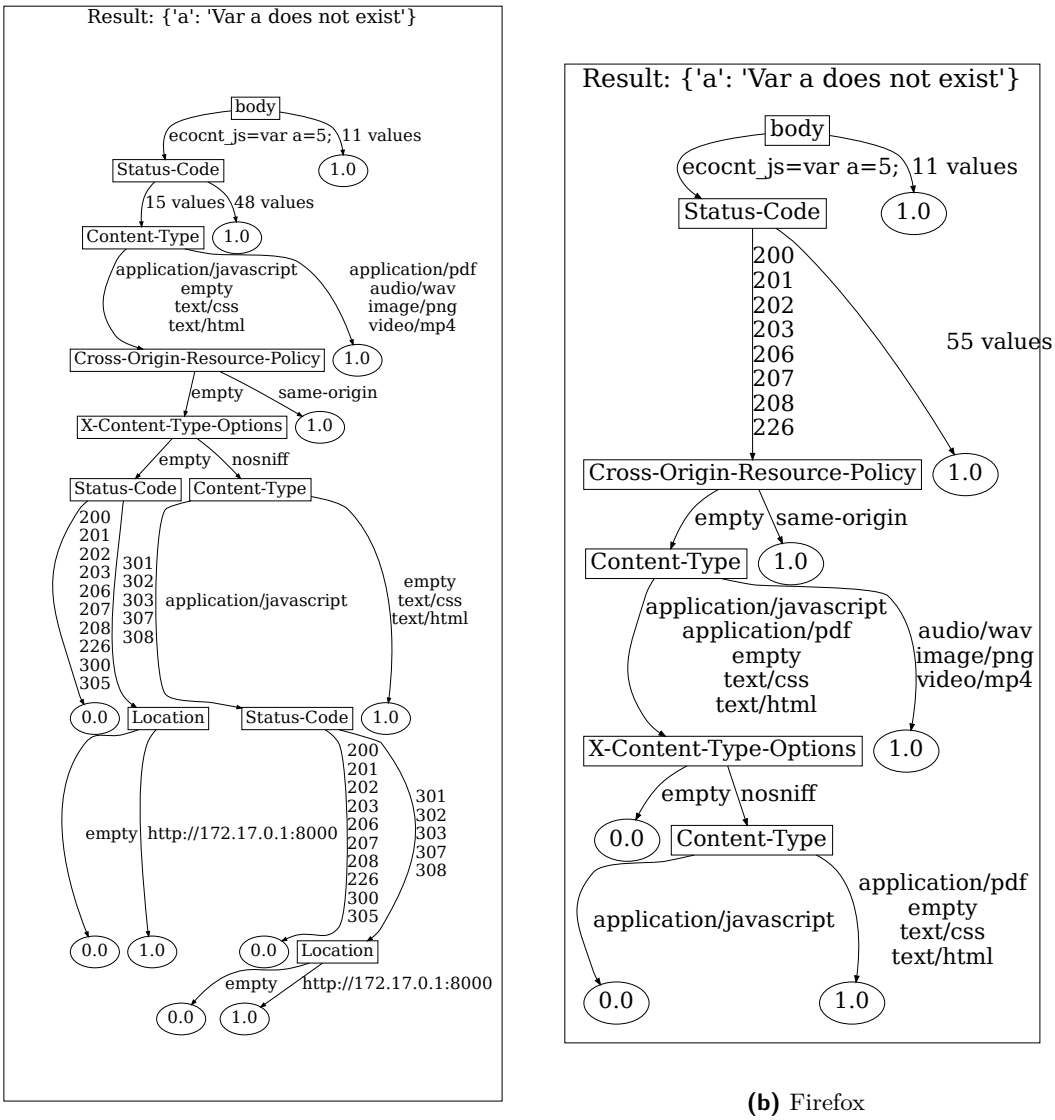
(a) Chrome

(b) Firefox

Figure 3.2: Decision trees for the leak channel events-fired_set_img.

Chrome explains this behavior as Chrome replaces images by an empty body in these cases [5].

Figure 3.3 presents the trees created for global-properties_hasOwnProperty_script (Cross-Site Script Inclusion (XSSI)). This time, the *1.0* leaf nodes represent cases where no variable is set, and the *0.0* leaf nodes represent cases where the tested variable is set. To detect that the variable is set, the body content needs to contain the variable. However, even if the body is valid JavaScript containing the variable, the script parsing can fail for many other reasons resulting in a not-set variable. First, not every status-code is allowed. Firefox, only accepts codes 200, 201, 202, 203, 206, 207, 208, and 226. Chrome additionally accepts the redirection codes 300, 301, 202, 303, 305, 307, and 308. For all of the redirection codes except 300 and 305, no location header is allowed as otherwise a redirect would occur. In both browsers, CORP stops the parsing of the resource. The allowed content-types are application/javascript, text/css and text/html



(a) Chrome

(b) Firefox

Figure 3.3: Decision trees for the leak channel global-properties_hasOwnProperty_script.

in Chrome. Firefox additionally allows application/pdf. These content-types are allowed if the X-Content-Type-Options header is not set. If the header is set, both browsers only allow application/javascript as the content-type.

The patterns observed in the two examples above regarding allowed status-codes or content-types apply to all leak channels. These results highlight that Chrome and Firefox have major differences in how they handle the parsing of responses. These differences can potentially lead to many XS-Leaks only existing in one browser. All created trees are available in the associated online material referenced in appendix B.

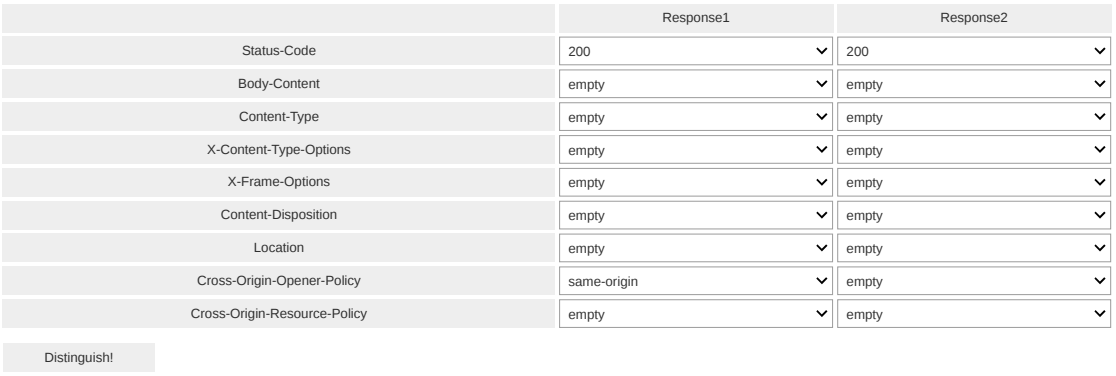


Figure 3.4: Screenshot of the test responses application.

3.4.3 Test responses application

One can use the created decision trees to predict for every possible response what the observed outcome will be. However, to predict the outcome of a response using the trees, one first needs to map it to the response space. For example, check the response's body type, and if it is a valid image, map it to ecocnt_img=width=50. Then, check if the CORP header has a prohibiting value and map it to CORP=same-origin in that case. After mapping the response, one can follow the corresponding path in the trees and predict the outcomes as the value of the reached leaf nodes in every tree. We note that the prediction for the same browser and version is correct as long as the mapping is correct, no unconsidered property changes the observation, and the server does not use defenses such as Fetch metadata.

Analyzing all the created trees or the raw data is time-consuming and difficult to process for humans. Thus, we release an application to check if two responses are distinguishable to make the results more accessible. Figure 3.4 displays a screenshot of this application. First, users can configure two responses using drop-down menus. Then, the application will show all leak channels that can distinguish the two responses and the individual outcomes of the responses. Browser vendors can use this tool to check which leak channels

still leak information in their browsers. Additionally, web developers can use this tool to check if they have potentially leaky endpoints. The example shows two responses that only differ in the COOP header. This difference can be detected with several different leak methods when including the URL with *window.open*. In this example, we also see that in Chrome, one additional vector exists. This vector exits because the framecount of responses with COOP is not accessible in Chrome, whereas it is *0* in Firefox.

3.4.4 Security relevant bugs

A manual analysis of the created decision trees revealed odd paths where the observed result was unexpected. After finding such paths in a tree, we manually confirmed that the unexpected result was happening, checked if it broke any specifications and could cause trouble to users, and then reported it to the browser vendors. In the following, we summarize the security-relevant bugs discovered while analyzing the data of this chapter.

X-Frame-Options bug: In certain cases, Firefox ignores the X-Frame-Options header and displays the response, despite that it should block the response due to the presence of the XFO header. A response with XFO set to deny will not get blocked if the status-code is 300, 301, 302, 303, 307, or 308. In addition, the location header is not allowed to be set, as otherwise, the browser would perform a redirection first. If such a response is included using the IFrame tag, users can see the content, which attackers could use for clickjacking attacks. In terms of XS-Leaks, it can be abused for the leak channels global-properties_securitypolicyviolation_iframe-csp and global-properties_postMessage_iframe. For them to work, the body needs to contain code to perform a redirection or broadcast a postMessage. The content-type needs to be either text/html or empty. This finding is related to the known behavior that browsers do not enforce XFO if the response is a server-side redirection [52]. For more details, see the created bug report [74].

Cross-Origin-Resource-Policy inconsistencies: A second finding is that Firefox enforces CORP on the embed and object tags, whereas Chrome does not enforce CORP on embed and object. However, CORP should only be enforced on *no-cors* requests and not on *navigate* requests [33]. IFrame, object and embed all issue *sec-fetch-mode: "navigate"* requests in both Chrome and Firefox. Therefore, Firefox should not apply CORP on these, but it does on the object and embed tags. For more details, see the created bug report [76].

mediaError bugs: We discovered additional bugs in handling the MediaError object of audio and video resources for both browsers. Abusing the MediaError object to leak information cross-site was first discovered in 2018 [3]. According to the original bug reports, this leak should be fixed in Chrome, and Firefox [2, 1]. However, the fixes implemented are both incomplete. We found several bypasses that still work in current versions highlighting the need to take a more comprehensive approach to XS-Leaks instead of only testing with a single response pair if a leak works or not, as this can lead to insufficient bug fixes.

In Firefox, the implemented fix is only applied to cross-site pages and not to same-site pages. This loophole makes it possible to leak all information that was possible before on subdomains or different ports. For more details, see the created bug report [73].

Figure 3.5 illustrates the incorrect behavior in Chrome. For each of the four possible outcomes, a binary sub-tree exists. The tree class of every sub-tree indicates the positive outcome of the tree. All paths that lead to a leaf node with *1.0* result in this outcome. All other paths result in another outcome. The first sub-tree shows when the result will be *null*, i.e., no error occurred. The result is only *null* for status-code 200, no CORP, and a valid video (or audio) in the body. Also, the content-type header has not to trigger CORB protection. The second sub-tree shows the responses resulting in the empty error message that should apply to all invalid responses according to their fix. The third sub-tree shows how one can detect CORP or status-codes 100, 101, 102, 103, and 407, resulting in an error message with content *MEDIA_ELEMENT_ERROR: Format error*. The last sub-tree shows another error message: *PIPELINE_INITIALIZATION_FAILED* that was reached for the valid but empty audio response in case status-codes, CORP or CORB did not block the resource. For more details, see the created bug report [71].

CSP path matching bug: While building the framework, we discovered an additional bug. The CSP matching algorithm should ignore the path component after a redirection [16]. Chrome follows the algorithm when a redirection status-code causes a redirection. However, Chrome does not follow it when a meta-refresh tag or JavaScript causes the redirection. One can use this behavior to find out that a same-origin redirection happened cross-origin. More details can be found in the created bug report [70].

Fetch metadata inconsistency: During the manual confirmation of the results, we discovered another bug. In a newer version, Firefox introduced Fetch metadata. However, it is sending *sec-fetch-site "cross-site"* instead of *sec-fetch-site "same-site"* if only the ports differ on localhost or IPs, which is against the standard [32]. For more details, we refer to the created bug report [75].

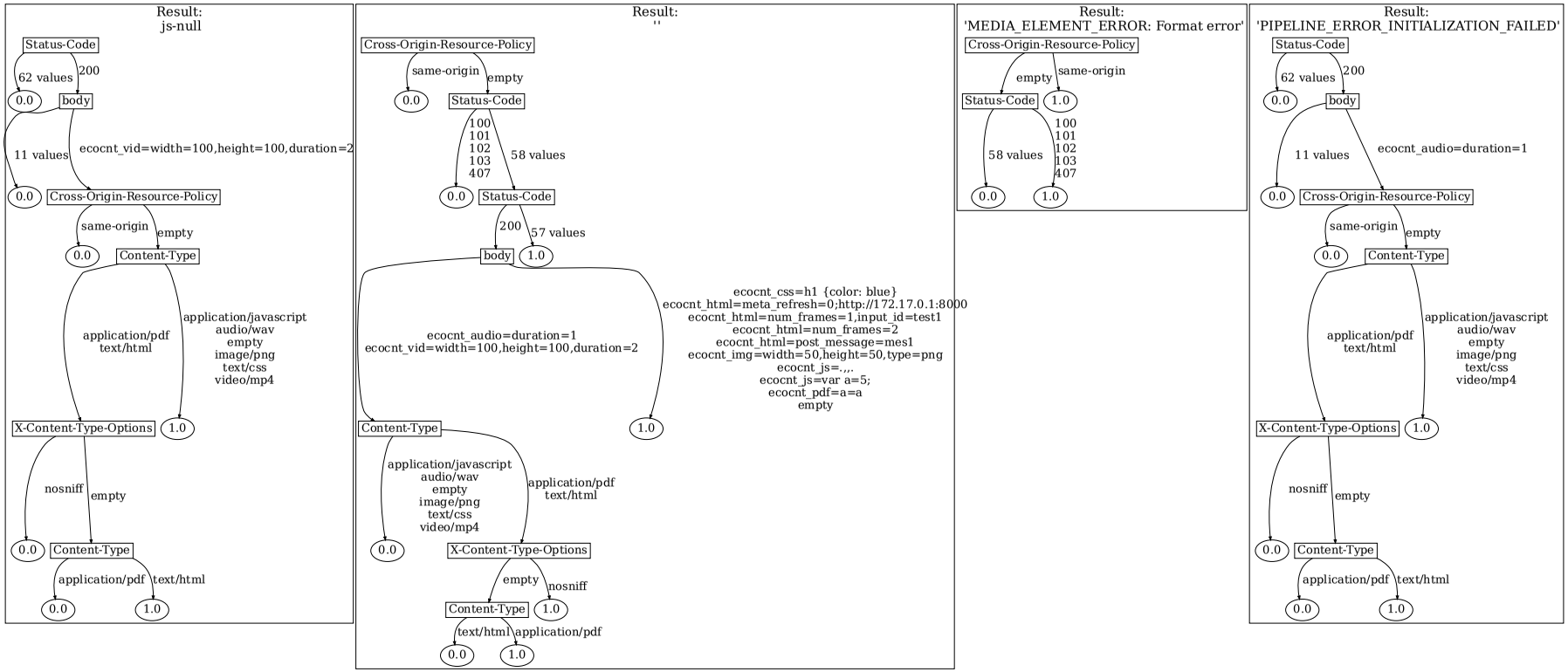


Figure 3.5: Decision trees for the leak channel object-properties_mediaError_audio in Chrome.

Chapter 4

XS-Leaks in the wild

Previous research has shown that XS-Leaks existed even on top-class websites such as Google or Facebook potentially affecting millions of users [94, 56]. Additionally, we know that most XS-Leak methods still work in modern browsers from the results of the previous chapter. However, we do not know how big of an issue XS-Leaks are for the web on a greater scope.

For an XS-Leak method to work, a URL has to return a response that leads to one observation in state A and a response that leads to another observation in state B. Unfortunately, there are no satisfactory estimates on how many websites possess SD-URLs that are exploitable. Also, there exists no information on which inclusion methods and leak methods work well in practice and which do not. Such information could guide further development and allow more targeted action as methods that do not work in practice have less urgency to be eliminated. The work by Sudhodanan et al. suggests that XS-Leaks are quite a serious issue as they found at least one vulnerable URL on all 58 websites they have tested [92]. The small number of tested websites and many recent developments make it unclear if their results still apply today and generalize to the larger web. Web developers might have learned from past vulnerability reports and nowadays try to deliver indistinguishable pages for logged-in users and anonymous visitors. Additionally, new web security features such as SameSite cookies or Fetch metadata allow servers to return the same indistinguishable response for cross-site requests while delivering different responses that would be distinguishable for same-origin requests. Finally, bug fixes and other improvements in browsers can also reduce the number of leaks.

In this chapter, we investigate the above-illustrated knowledge gap in-depth and answer the research question: “Which XS-Leak methods work how often in the wild in different browsers?”. Additionally, we investigate how many SD-URLs exist and how new security

features are used and influence the prevalence of XS-Leaks. We also investigate whether the differences between the browsers discovered in the previous chapter result in different attack surfaces of the browsers. With the gathered information, such as which methods often work in practice or primarily in one browser, browser vendors can prioritize their countermeasures. Additionally, improved education can prevent web developers from making common mistakes reducing the overall impact of XS-Leaks.

The first section of this chapter sets the scope of the investigation. The second section introduces the fully automatic does-it-leak pipeline developed to find vulnerable URLs on any website. The third section evaluates how reliable the different parts of the created pipeline are and which stages need improvement. Finally, the last section summarizes the results and shows that most websites are still vulnerable to XS-Leaks, but there are considerable differences between the browsers and methods.

4.1 Scope

Analyzing XS-Leaks in the wild requires us to define what we consider the wild, which state-differences we want to examine, and which leak channels we test. In this section, we first describe which websites we tested and which crawl settings we used to cover a sufficiently large part of the web. Then, we explain which state-differences we consider and why it is hard, but not necessary, to qualify the exact state-difference leaked. In the end, we list the leak channels studied in this chapter which are mostly the same as the ones studied in the previous chapter.

4.1.1 Tested websites and crawl settings

Testing a sufficiently large sample of websites is necessary to understand how much of an issue XS-Leaks are in the wild. For selecting websites, we need a selection criterion. We decided on using the Tranco ranking [53] as the basis of website selection to test high-ranked websites as vulnerabilities on them affect many users. A state-full crawling approach is necessary to investigate XS-Leaks. Additionally, one needs to investigate every tested website in-depth to find as many vulnerable URLs as possible instead of only testing the landing page. These two considerations drastically decrease the possible scale for XS-Leak studies in comparison to other web security studies.

We decided to seed the testing pipeline with the top 20,000 websites, according to Tranco [53], to cover a broad set of high to medium popularity websites. As the tool used to create state information on the websites only has a relatively low success rate

of roughly 2%, we decided to additionally test all possible websites from the Tranco top 50 with a semi-manual state creation step to cover more of the top websites. In total, we successfully created state information on 430 websites and found 258 sites to be vulnerable in at least one browser.

For every website we test, we first have to find all URLs belonging to that site that might be vulnerable. In general, the more URLs, the higher the chance to find a vulnerable URL. However, more time is needed to test more URLs, and the chance that the crawler gets stuck in an infinite loop increases. These considerations suggest that a limit on the crawling process is necessary. Therefore, we decided to crawl a maximum of 100 URLs per website or to a maximum depth of three, whichever the crawler reaches first. It is important to note that this does not mean only 100 URLs are tested per website, as we also collect and test all requests issued on the visited documents, such as included images or fetch requests to APIs.

Websites might also load slowly due to high load on their end or because they detected a crawler. In addition, various other errors can also occur, such as connection failures or browser crashes. To deal with these issues, we defined a maximum page loading time of 20 seconds and decided to test every URL a second time in case of an error or timeout. This setting applies to both the crawling step as well as the dynamic confirmation step. We describe the exact settings of the different programs in appendix B.

4.1.2 Considered state information

In principle, XS-Leaks can leak any possible state difference that exists between visitors of a website. How many and what kind of states exist heavily depends on the tested website. In addition, different URLs on a site might leak other state differences. For example, a website might differentiate between first visit visitors and visitors that already visited a page (Access detection), between logged-in users and anonymous visitors (Login detection), between a specific user and everybody else (Deanonymization), between a regular user and a premium user (Account type detection), and many more. These various state differences differ in severity and in how hard it is to test for them. The following describes the state difference considered in this thesis and the issues with verifying what a detected state difference means.

To detect a state difference, one must test a URL in both state A and state B, e.g., a logged-in premium user A and an anonymous visitor. If a difference is detected, one could say that the website is vulnerable to login detection. However, this is not necessarily correct. For example, it could also be a difference between user A and anybody else (Deanonymization), when no difference between a logged-in user B and an anonymous

visitor can be detected with this URL. To identify the reason for a detected state difference, one must enumerate all states that an application has, test every URL in all states, and compare the results. Enumerating a complete set of states is impossible without access and understanding of a website's application logic, as states could even be something like users created on a specific day. For example, imagine a URL only accessible to users on their birthday on a social networking site. In general, it is impossible to verify the exact nature of an identified state difference automatically, and getting a reasonable estimate by testing in many different states is impractical. For the above reasons, we only test two states for every website, one logged-in user A and one anonymous visitor B. These states are comparatively easy to create automatically. The pipeline will then tell whether it could distinguish state A from state B or not. Most of the time, the detected state difference will probably be login detection. However, we cannot verify this. It could also be anything else that differs between the two states, such as the user id (deanonymization) or that the logged-in user has accepted the cookie policy and the anonymous visitor has not.

We argue that a creative attacker can abuse any detectable state difference and the presence of any state difference hints at the existence of other, more problematic state differences. Thus, we exclude the qualification of the actual state-property we distinguished on each URL from the scope of this thesis and will only report that we can distinguish between states A and B.

4.1.3 Considered leak channels and browsers

As seen in the previous chapter, many leak channels exist, and some are harder to test reliably than others. Overall, we decided to test the leak channels that we found reliable in the previous chapter. In the following, we describe small changes made.

In addition to the channels of the previous chapter, we added the leak channel *load count* to the events fired category. As observed previously, the order of the events can sometimes be non-deterministic, and to account for this, we used the set of events. However, if, for example, IFrames perform client-side navigations, more than one load event is fired. The *load count* channel accounts for such cases as it counts the number of fired load events.

In the previous chapter, we included the leak channels global-properties_hasOwnProperty_script and global-properties_getComputedStyle_stylesheet to show that they work in general. These methods, however, require analyzing the content of the two returned scripts or CSS resources to see if there are any differences in them and then add specific checks to verify if a variable or a style only set in one state is present. The former attack can often leak more than a single bit of state difference, and sometimes even directly leak

a password or user name and is known under the name of Cross-Site Script Inclusion [55]. The latter can be called Cross-Site-Style-Inclusion [40, 29]. Both of these channels would require significant engineering efforts, and several studies focusing on them already exist [55, 29]. For these reasons, we decided not to cover them in this part of the thesis. It is important to note that the primitive cases of these attacks, where one response is a valid script or style, and the other is not, are already covered by other leak methods. Valid stylesheets fire a load event, whereas other responses such as HTML documents will result in an error event. If one response returns a valid JavaScript file and the other does not, an onerror handler can detect this, and the global-properties_window.onerror method covers this case.

We have seen no real difference in behavior between Chrome and Edge except for the unstable global-properties_downloadbarheight method. Thus, we exclude MicrosoftEdge from the tests in this chapter. In the end, we test the other two browsers considered in the previous chapter: Firefox and Chrome, to reuse parts of the created testing infrastructure and the created decision trees.

4.2 Does-it-leak pipeline

Fully automatically finding URLs vulnerable to XS-Leaks on a website is a challenging task, consisting of many steps. To be able to solve this task, we built a pipeline able to perform all needed steps. Here, we give a high-level overview of the pipeline, and the following sections explain the specific implementation created for this thesis in more detail.

Figure 4.1 provides an overview of the components of the does-it-leak pipeline. It consists of a state generator, a stateful crawler, a static pruner, and a dynamic confirmator. Every tested website runs through all these components in sequence, and in the end, the pipeline outputs all vulnerable URLs. The following explains the task and general structure of each component.

First, there needs to be a state generator component. This component is responsible for creating at least two states on each tested website to be compared later. As explained in the previous section, we consider the state of one logged-in user and one anonymous visitor, but in principle, every state difference would be possible. The reached states must be passed to the following components somehow. One could either rerun the state creation actions in every component before any other action is taken or rely on the fact that the state information is usually bound to the cookies of a session and share these

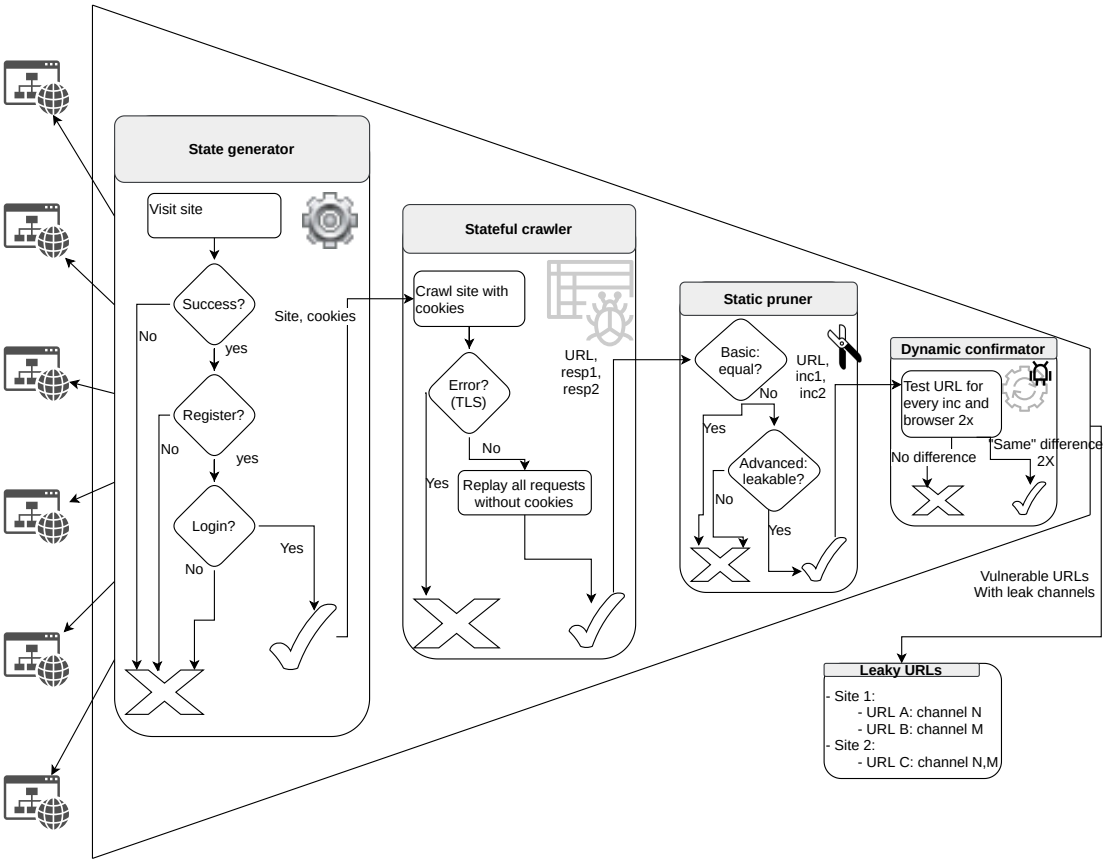


Figure 4.1: Overview of the does-it-leak pipeline.

with the other components. For the pipeline, we decided on the latter and passed the cookies along.

Second, we need a crawler to find URLs belonging to a site that might be vulnerable. In principle, to find URLs on a website, every stock web crawler could be used. However, we do not only want to find URLs included in $<a>$ tags or similar on the visited websites. We also want to record all requests made while visiting the site, including image resources and dynamically created fetch requests, as all these URLs can also be vulnerable. The crawler component does not necessarily have to be stateful. However, a stateful crawler provides several benefits, such as increased coverage and generating the data needed for a static pruning component. Suppose the crawler starts from a non-default state, such as a logged-in user. In that case, it is more likely to detect potentially vulnerable URLs as it has the chance of entering a protected area not accessible by anonymous visitors where every URL might be vulnerable. A standard non-stateful crawler that only crawls the website as an anonymous visitor would not find such URLs. Additionally, a stateful crawler can visit every discovered URL in several states and collect the responses. The collected responses can be used for pruning and quantifying which response pairs exist in the wild.

The third part is the static pruning component. This component is not strictly necessary as one could test all found URLs for every inclusion method. However, it can significantly reduce the time of the experiment and server load. The general idea is to exclude URLs that returned two responses that cannot possibly be distinguished from each other by the considered attack model and remove them from the set of URLs that will be dynamically confirmed. We use a basic pruner that disregards all URLs that returned the *same* response for both states, as well as a more advanced tree pruner built upon the results from the previous chapter.

The last component is the dynamic confirmator. This component confirms that a URL is vulnerable for a specific leak channel in a specific browser. First, it receives the list of potentially vulnerable URLs, the list of leak channels to test, and the state information. Then it tests every URL for each corresponding inclusion channel in all states in all supported browsers using the automation setup and attack-page generator from the test browser framework of the previous chapter. This step is necessary as vulnerable URLs reported by the static pruning part might not be vulnerable cross-site due to SameSite cookies, Fetch metadata, and other features. After the pipeline finishes all tests, it checks which leak URLs (combination of target URL and inclusion channel) resulted in different observed properties, making them distinguishable. It then retests all leak URLs found to be vulnerable to reduce the chance of false positives. False positives can occur when the request has timed out for one state, or if the number of frames on a page is random, or for many other reasons.

4.2.1 State generator

Successfully creating at least two states on every tested website is a fundamental requirement when testing for XS-Leaks. We used two different approaches to create the initial state on target websites. One is a modified version of cookiehunter [28], the other is manual registration of accounts with replays of Selenium IDE scripts for login [85]. Both approaches register one user, log the user in and then share the user state by exporting all cookies. The other state is a user without any cookies set and does not need any preparation. The following explains the structure and changes to cookiehunter and how to create and use selenium login scripts.

The cookiehunter tool performs the complete process of registering user accounts, logging in, and confirming that the login was successful without user interaction. The tool uses a set of regular expressions and manually defined rules to find registration and login forms and single sign-on (SSO) buttons on websites. If the tool finds a registration form, it attempts to automatically fill it and then uses several oracles, such as if a mail is received

or the registration form is not accessible anymore, to check if it was successful. It then performs a similar process for the login procedure. For SSO, the researcher has to provide accounts on the identity providers Google and Facebook, and the tool will click on the SSO buttons and try to create a new account on the tested site. In the end, it performs a request without cookies to check that the reported success is not a false positive. For more details on the tool and its procedure, we refer to the original paper [28].

We modified the cookiehunter tool to make it work again in 2021. It used several outdated libraries calling external APIs such as Gmail and Google translate that were not available anymore, making the tool crash at startup. We ported the code from python2 to python3 and replaced all libraries with newer working versions. The SSO process for Google and Facebook has changed, so we adapted the SSO steps. We additionally added one regular expression but mainly left the original registration and login heuristics untouched. We added additional code to pull the tested sites from the Tranco list [53] and send the gathered cookies to the stateful crawler in case of a successful login.

The cookiehunter code is not open-source, and the success rate of roughly 2% is relatively low. Therefore, we added another method of performing the state generation that we can use as an alternative to cookiehunter. For this thesis, we manually tried to register accounts on the top 50 websites, according to Tranco. Then, for each website where we successfully registered an account, we recorded the login process with Selenium IDE [85] including clicking on something only available for logged-in users, usually the profile icon, to make sure that the login was successful. Selenium IDE can export the recordings to python scripts. However, these exported scripts are not identical to the recording in Selenium IDE, e.g., timing and fail-over strategies are different. Therefore, we created a script to automatically make the exported python scripts more similar to the actual recordings and add the cookie gathering step. Finally, we replayed these login scripts on the server, and in case of a successful login, started the stateful crawler with the collected cookies.

4.2.2 Stateful crawler

The stateful crawler is responsible for finding as many potentially vulnerable URLs belonging to a site as possible. We use a crawler build upon the puppeteer automation library [69]. The crawler first sets the cookies from the previous step and then visits `https://<site>/`. It then enters all links found on the website's landing page in a queue and visits them until a maximum of 100 pages or a depth of three to not run into an endless loop. As this process only finds URLs in $<a>$ tags, we route all traffic of the browser through a MITM proxy [18] to also save all URLs belonging to subresources, such

as images and dynamically created fetch requests. For all first-party GET requests, i.e., the potentially vulnerable URLs, we duplicate the requests and perform them without cookies to get the response for an anonymous visitor.

The crawler should output not only a list of URLs belonging to a site but also save all responses of every state as the static pruning component needs them. We decided to only crawl in the logged-in state and perform the requests of the anonymous state with the proxy. We reason that it is likely that the logged-in crawler finds resources only accessible to logged-in users, such as profile pages. In contrast, it is unlikely that an anonymous crawler will find resources only available for anonymous visitors. The alternative would be to crawl in both states independently and then test every URL only found in one state in the other. However, this would require high synchronization efforts.

4.2.3 Static pruner

The previous step generates thousands of potential URLs for most websites. Many of these URLs belong to publicly accessible static documents, and testing them would waste time. Therefore, we created a static pruning tool to reduce the load on the tested websites and speed up the testing process. This tool analyzes the collected responses of both states for every URL and removes all URLs where the dynamic confirmator should not be able to distinguish the states. The pruning consists of three stages explained in the following.

The first stage removes all URLs where we only have one observation. As the stateful crawler only replays first-party GET requests, these removed URLs are mostly requests to third parties and requests using other HTTP verbs such as POST or OPTIONS. In addition, we removed a small number of requests where the first-party GET request failed in one state. One reason for failing could be that a timeout occurred.

The second stage, called basic pruning, checks if a URL delivered the *same* response for both states. Common sense tells us that it should be impossible for any dynamic confirmator to distinguish two identical responses, so such a URL should not be leaky and does not have to be tested. However, it is improbable that the identical response was recorded twice due to ever-changing properties such as the date header. For this reason, we have a more broad understanding of what it means that two responses are the *same*. We first map the responses to an extended version of the response space we defined in chapter 3. After that, for every response, only the status-code, the body hash, and the smoothed value of the seven considered headers (location, content-type, X-Frame-Options, content-disposition, Cross-Origin-Opener-Policy, X-Content-Type-Options, and Cross-Origin-Resource-Policy) remain. In addition, we add the non-smoothed version of

the location header as a difference here is enough to warrant further investigation. If these smoothed responses are the same, we consider the original URL to be unleaky and abort. Otherwise, we continue to the next pruning stage.

The third and final pruning stage uses the decision trees we introduced in the previous chapter. To use the created trees and predict the outcome for every response, we first have to convert the responses to fit the trees. The status-code automatically fits. Most headers have simple transformation functions, can stay as they are, or in the backup case, are treated as empty. For the body, this is more complicated as we cannot rely on the content-type header as it can be empty or incorrect. Instead, we use the Linux file command [36] as well as Apache Tika [6] to get the most likely actual content-type of the document body. After determining the body type, we must map it to a body of the created response space. This mapping works well for PDFs, images, audios, and videos. For HTML resources, we only get the information that the body is of type HTML. However, we cannot statically determine how many frames an HTML document has or if it sends a postMessage. Therefore, we duplicate every response with inferred body-type HTML for all four possible HTML body values in the response space to account for every possibility.

In general, only the methods that have different predicted outcomes should potentially be vulnerable. For some cases, such as image width, this is not the case. Even if the tree predicted that both responses would lead to width *50*, this does not automatically mean that the URL is not vulnerable. It only means that we mapped both actual responses to the training responses with an image of width *50* in the body, and the width is observable. We add a post-processing step for these cases and check if the body hash is different, and if yes, mark them as potentially vulnerable.

This stage goes further than saying a URL is potentially vulnerable but includes information on which inclusion methods this URL might be vulnerable. This information is crucial as otherwise, one would need to test every remaining URL for every inclusion method. The static pruner passes all URL-inclusion method combinations, called leak URLs, left after this final pruning stage, to the dynamic confirmator.

4.2.4 Dynamic confirmator

The static pruning component outputs a list of leak URLs. This list of potentially vulnerable URLs with possible inclusion methods is much smaller than the total combination of URLs multiplied by the number of inclusion methods. However, the list of leak URLs still contains many non-leaky URLs. For example, the preprocessing step of body content can introduce non-leaky URLs. One possible explanation is that not every HTML page sends

a postMessage. Likewise, the post-processing step can also introduce non-leaky URLs. For instance, if two responses have different body hashes but return images of the same dimensions, they are not distinguishable by the tested methods. Additionally, non-leak URLs can be introduced by defense mechanisms such as SameSite cookies or Fetch metadata, resulting in not receiving the same responses cross-origin that we observed for the same-origin requests. Last but not least, a URL might be invulnerable when other factors such as time or randomness cause the differences in the returned responses. For example, a site could return responses with a nondeterministically changing number of frames due to different ads.

To only report true positives, i.e., URLs that leak state differences cross-site, we need to confirm each potential URL dynamically. The dynamic confirmator takes a list of leak URLs for every browser and tests every leak URL in both states using the attack-page generator from the test browser framework. If the confirmator finds a difference, it tests the leak URL a second time to ensure this was not due to chance or other errors. If the leak works both times in the *same* way, we report the URL as vulnerable. Otherwise, the does-it-leak pipeline will not confirm the URL as vulnerable. The definition of *same* includes the case of having the same observations twice for both states, but also additional less restrictive rules for some leak methods. For example, we accepted two times zero frames in one state and two times non-zero frames in the other state as *same* for the framecount method.

4.3 Pipeline Evaluation

For this thesis, we have created a complicated multi-step pipeline to answer the research questions and detect XS-Leaks on many websites in a reasonable time. The pipeline consists of adapted pre-existing tools as well as modules created from scratch. The results show that the pipeline manages to find XS-Leaks in at least one browser on 258 websites. However, they do not show in which step and for which reasons the pipeline failed.

In this section, we evaluate the different steps of the pipeline regarding false negatives and false positives. First, we discuss the shortcomings of the state creator and why its success rate decreased since its introduction by Drakonakis et al. Then, we analyze why a successful login according to the state creator does not necessarily mean the website gets crawled properly. After that, we show the effectiveness and applicability of both main pruning steps.

4.3.1 State creator

Checking for XS-Leaks on a website without access to the source code requires creating and comparing at least two states on the tested website. As described earlier, we use the states logged-in user and anonymous visitor in this thesis. Getting the crawler to the state anonymous visitor is easy as this is the default state of every site. Automatically getting to the state of a logged-in user is challenging. Out of 20,236 attempted sites, the tools to create state information only succeeded on 430 sites. In the following, we highlight several reasons for this.

We primarily used the cookiehunter tool [28] to register and login on roughly the Tranco Top 20,000 [53]. With the modified version we achieved success on 18 of the Tranco top 1,000 (1.8%) and success on 412 on all 20,236 attempted sites (2%). The original paper [28] contains an extensive error evaluation on why the success rate is low, so we refrain from redoing it and only summarize their results. Reasons range from no account functionality existing on the site (on 86.6% of sites, the tool did not find one), over input constraints (e.g., complicated password rules or valid credit card is required), and anti-bot measures (e.g., captchas), to the login or registration oracles not being able to confirm that they succeeded.

In the original paper, the authors report success on 95 of their top 1,000 (9.5%) and an overall success rate of 1.6% on the complete set of 1,585,964 tested sites. Their results suggest that higher ranked sites should have a higher success rate, and one reason might be that most popular sites have account systems, whereas many long-tail websites do not have account systems. Here, we will list reasons why the success rate in our runs on the top 1,000 was significantly lower than what they reported. The tool uses several outdated libraries, and some of these are not supported by the external APIs anymore (e.g., the translation service used). As a result, the tool either misses functionality or crashes. We ported the tool to recent versions and tried to fix such issues, but some problems remain. Additionally, many websites now use bot detection frameworks, making it impossible to register and log in with the current cookiehunter tool. Notably, during the experiments for this thesis, the bot detection of Google SSO changed. Thus, after the first few thousand sites, no login with Google worked anymore. Other sites in the Tranco top 1,000 are only redirects and result in duplicates such as fb.me. Some sites also did not load successfully or timed out. Additionally, although the SSO login with Facebook and the email connection with Gmail worked in general, it was reset several times during the experiment with no automatic option to re-enable it. Until we manually repaired them, cookiehunter continued without these features. Also, many websites now use more complicated or non-standard login and registration techniques, making them undetected or unfillable by cookiehunter. In addition, aggressive cookie policy banners

make some sites unusable until they are dealt with, and cookiehunter does not attempt to close these banners to interact with the page.

As high-profile sites are particularly interesting, we added a second semi-manual login strategy. We manually registered 26 accounts on the top 50 websites, according to Tranco. On the other 24 websites, either no registration option was available (8), registration required credit card or phone information (8), it failed due to language problems (5), or it was a duplicate URL for another service in the list (e.g., youtu.be) (3). Replaying the 26 created login scripts worked in 18 cases. The other seven failed due to risk-based authentication requiring additional steps such as clicking on a link or solving a captcha.

4.3.2 Stateful crawler

The state creator passes every site for which the state creation was successful to the stateful crawler, including the corresponding cookies. However, the stateful crawler did not crawl all of these 430 sites successfully. We list reasons why the crawl process sometimes fails and general statistics on the crawling process in the following.

The crawler itself saved how many URLs it tried to access, and the proxy saved all network requests routed through it, including all subresources. Unfortunately, due to data loss, we only have information of the crawler itself for 396 of the 430 sites. For 85 sites, the crawler only tried the initial URL meaning some error occurred. For another 290 sites, the crawler crawled 100 URLs. For the remaining 21 sites, the crawler crawled between one and 100 URLs, meaning it reached its maximum depth limit. The static pruner component takes the URLs saved by the proxy as input, and even if the crawler only tried the initial URL, there still can be several vulnerable URLs recorded by the proxy. In the following, we focus on the data created by the proxy part of the stateful crawler.

On 29 sites, the proxy did not record a single URL. On 25 sites, this is due to TLS problems. These sites either delivered incorrect certificates or did not support HTTPS at all. We decided not to ignore such errors and skip websites with TLS problems in the stateful crawler as a secure origin is necessary for some of the studied features. Moreover, modern browsers will display a severe warning before accessing insecure sites. However, such sites still can get passed to the crawler as the state creator component ignores all certificate errors due to legacy reasons. For the other 4 unsuccessful sites, the proxy timed out. These timeouts could be due to bad luck, such as server maintenance or bot detection deployed by the site.

	Original URLs	Basic pruning URLs	Leak URLs	Chrome tests	Firefox tests
mean	535.83	82.37	988.43	252.39	279.50
std	641.28	88.39	1060.73	433.58	451.04
min	1.00	0.00	0.00	0.00	0.00
50%	352.00	72.00	864.00	147.50	174.50
max	6252.00	580.00	6960.00	6745.00	6750.00

Table 4.1: URLs and tests per site before and after pruning.

On an additional 29 sites, the proxy recorded less than 20 unique URLs. The low number of recorded URLs indicates that errors occurred as most tested sites should have more URLs. Cross-site redirects on the landing page are one reason that occurred on at least six sites. For example, the state creator reported that it managed to log in on readthedocs.io, but the initial page redirects to readthedocs.org, leading to a stop of crawling as we configured it to crawl readthedocs.io. Another reason is that some sites detected the crawler as a bot and responded with a *no access for bots page* or the crawler timed out or crashed on one of the initial pages. It is important to note that a site can still be vulnerable even if there is a cross-site redirect on the initial page, so we did not remove them.

4.3.3 Static pruner

Testing all combinations of found URLs with all inclusion methods in all browsers would take a long time and would require many requests. Therefore, to make the search for XS-Leaks more efficient, we proposed a static pruning step that heavily reduces the number of performed tests. We can analyze the pruning in two regards: is it effective, i.e., how many requests does it save, and is it correct, i.e., does it introduce false negatives. We evaluate both questions against the basic pruning step and the advanced pruning step in the following.

Table 4.1 provides the efficiency measurements for this evaluation. The first column shows the number of URLs per site. The second column shows the number of URLs remaining after the basic pruning step. The third column shows the number of tests needed if every URL remaining after the basic pruning step would be tested for every inclusion method. The last two columns show how many tests per site remain for both browsers after the advanced pruning step. The basic pruning step reduces the average number of potentially vulnerable URLs from 536 to 82, or by 85% in total and 79% on average per site. The maximum value is reduced from 6,252 to 580, or by 90%. Without the advanced tree pruning, we need to test every remaining URL in all inclusions methods. The advanced pruning reduces the average number of tests from 988 to 252 in Chrome and 280 in Firefox, which is an additional 74% and 72% reduction.

Grouping	URLs	P-URLs	GT	TP	FP	FPR	FN	TN	FNR
Target URLs									
Basic-all	32,952	5,766	487	382	5,384	0.17	105	27,081	0.22
Firefox	16,476	2,883	230	175	2,708	0.17	55	13,538	0.24
Chrome	16,476	2,883	257	207	2,676	0.16	50	13,543	0.19
Advanced-all	5,766	5,705	382	382	5,323	0.99	0	61	0.00
Firefox	2,883	2,857	175	175	2,682	0.99	0	26	0.00
Chrome	2,883	2,848	207	207	2,641	0.99	0	35	0.00
Leak URLs									
Basic-all	395,424	69,192	981	781	68,411	0.17	200	326,032	0.20
Advanced-all	69,192	17,052	781	747	16,305	0.24	34	52,106	0.04
Firefox	34,596	9,062	350	342	8,720	0.25	8	25,526	0.02
Chrome	34,596	7,990	431	405	7,585	0.22	26	26,580	0.06
audio	5,766	475	4	0	475	0.08	4	5,287	1.00
embed	5,766	1,340	16	12	1,328	0.23	4	4,422	0.25
embed-img	5,766	1,382	14	14	1,368	0.24	0	4,384	0.00
iframe	5,766	2,296	172	169	2,127	0.38	3	3,467	0.02
iframe-csp	5,766	2,304	163	161	2,143	0.38	2	3,460	0.01
img	5,766	466	6	6	460	0.08	0	5,300	0.00
link-prefetch	5,766	386	15	2	384	0.07	13	5,367	0.87
link-stylesheet	5,766	539	50	48	491	0.09	2	5,225	0.04
object	5,766	504	15	14	490	0.09	1	5,261	0.07
script	5,766	2,376	73	73	2,303	0.40	0	3,390	0.00
video	5,766	386	5	0	386	0.07	5	5,375	1.00
window.open	5,766	4,598	248	248	4,350	0.79	0	1,168	0.00

Table 4.2: Static pruner false positive evaluation.

The efficiency measurements show that the pruning step drastically reduces the time and requests needed. However, if the pruning misses too many vulnerable URLs, it is not worth it. To test for the applicability of the pruning, we reran the pipeline without pruning on 50 sites where we found leaky URLs for both Chrome and Firefox. On 36 out of 50, the re-login worked according to the state creator. For the others, the login failed. The re-login failed for several reasons. First, Google changed their SSO deployment in the meantime and now detects the creator as a bot. Additionally, sites can change all the time resulting in non-working logins. Moreover, sometimes the state creator manages to log in and register in one go but cannot log in independently. Out of these 36 sites, the pipeline crawled 34 successfully and found 30 vulnerable in at least one browser again.

We retroactively applied both pruning steps on these unpruned runs to analyze if the pruning steps would introduce any false negatives. Table 4.2 shows the results of this evaluation. The upper half shows the results for the target URLs and the lower half for the leak URLs. Both halves are further divided into basic pruning on top and advanced

pruning below. The first column shows which subset of data is considered. The second column lists the number of URLs or leak URLs without pruning. The third column shows the number after pruning. The fourth column shows the discovered vulnerabilities (ground truth). The remaining columns show the true positives, false positives, false positive rate, false negatives, true negatives, and false negative rate.

The does-it-leak pipeline would have missed 105 of the 487 vulnerable URLs and 234 of the 981 leak URLs with both pruning steps. The first row shows that all of the 105 vulnerable URLs missed are due to the basic pruning step. The first two rows in the second half show that out of the 234 leak URLs missed, 200 are due to the basic pruning step, and 34 are due to the advanced pruning step. The other rows show how the false negatives are distributed over the browsers and inclusion methods. Out of these 34 missed leak URLs, 13 use link-prefetch in Chrome which we concluded to be unstable in the previous chapter and thus removed. The other 19 have unclear reasons. It is important to note that for most websites, the pruning steps did not miss any vulnerable URLs and leak URLs. Instead, only a couple of sites are responsible for the majority of the false negatives, with one site being responsible for 90 out of the 234 missed leak URLs.

The advanced pruning based on the decision trees is highly effective without adding many false negatives. The basic pruning is practical as well. However, it adds a false negative rate of 22% with a false positive rate of 17% for the number of vulnerable URLs found. The basic pruning step is a necessary preprocessing step, and it is impossible to use the tree pruner without it. Furthermore, the basic pruning step only removes responses that do not differ in any of the seven considered headers, status-code, or body hash. The 105 URLs distributed over 200 leak URLs missed by the basic pruner can have various reasons explained next.

One reason for false negatives is that the pruning might remove information responsible for the distinguishable observations. For example, we do not consider the CSP header, which can be responsible for different observations if the frame-ancestors directive is used inconsistently in both states. Another problem is collecting the data input for the basic pruner with one same-origin visit per state for each URL in one browser only. It might, however, be that a URL is only leaky for cross-site requests. This behavior is possible as the server can check Fetch metadata and similar and only responds unsafely if the request is cross-site. Another explanation might be that the responses are based on the user-agent. The responses are not leaky for the user-agent used in the crawling step but only for the user-agent used in the dynamic confirmation part. Other possible reasons are server-side randomness and timeshift. In such cases, the server delivered unleaky responses during the initial crawl but leaky responses during the dynamic confirmation.

This behavior can either be due to time difference, e.g., a resource was deleted in the meantime, or random behavior of the server, for example, caused by load balancers or A/B testing. It might also be that other server-processing changes related to the crawl occurred, e.g., it might be that one state got rate-limited and the other did not. In such a case, the states in the pruning step and confirmation step differ, as we are now comparing a rate-limited state with a non-rate-limited state. We already confirm every leak URL twice to reduce this problem, but two times might not always be enough.

4.4 Results

The main question in this chapter and the complete thesis is "Which XS-Leak methods work how often in the wild in different browsers?" With the created pipeline, we could find vulnerable URLs on 258 out of 352 sites where we dynamically tested for them. These results show that the issue of XS-Leaks still prevails. Furthermore, we can use the collected data not only to tell which sites are vulnerable to XS-Leaks but also to create other insights. For example, the data reveals which dangerous response patterns exist, which inclusion channels are particularly powerful, and which differences between the browsers exist.

In this section, we present the essential results of collected data. First, we investigate the responses collected by the stateful crawler and uncover that some security headers are rarely used, whereas others are often used insecurely. We also show considerable differences between the responses to the requests of the logged-in state and the anonymous state, hinting at dangerous coding practices. Then, we investigate response pairs in more detail and find several anti-patterns of common but dangerous response combinations. We also investigate the security settings of the cookies collected by the pipeline and show that most websites rely on the default behavior of browsers for the important SameSite flag. Then, we answer the main question and split up the vulnerable results by browsers, sites, inclusion, and leak methods. We show that there are considerable differences between the studied browsers and methods and the tested sites. Finally, we present potential issues that can affect the validity of the presented results.

4.4.1 Headers and responses

The crawling framework records all outgoing requests from all visited websites using a proxy. The saved data includes all kinds of requests such as GET, POST, OPTIONS, CONNECT to a range of URLs as most websites include third-party content. However, as explained earlier, we are only concerned with first-party GET requests and only replay

Cookies	GET	POST	OPTIONS	CONNECT	HEAD	PUT	PATCH
True	811,700	64,353	6,246	1,558	289	30	3
False	217,654	0	0	0	0	0	0

Table 4.3: All requests saved by the stateful crawler by HTTP verb and state.

these without cookies. Many URLs are requested several times, e.g., many websites include a company logo on every HTML document. In the database, we only saved the first response to a request from each state and outgoing site as long as the HTTP version and the returned status-code did not change and otherwise increased a counter that the URL was requested several times. In this section, we investigate what information the collected response data contains.

Table 4.3 shows all requests excluding the retest of some sites for the evaluation section. A total of 1,101,834 responses is in the database. Excluding all third-party requests and all non-GET requests, a total of 464,411 responses remain. Next, we exclude all requests that only have an observation for one state, i.e., the other request failed for some reason. Finally, we removed the requests that have returned more than one status-code for the same state, mostly these are requests that got rate-limited during the crawl and observed the code *429* once and another code once. In the end, 215,276 unique URLs with two observed responses remain, and we consider these pairs in the following.

On all of these 215,276, we can get statistics on how often various status-codes and headers occurred. These statistics reveal if there are any differences between logged-in users and anonymous visitors and other observations. Table 4.4a shows the most commonly observed headers on all responses. The most common header *date* occurs on almost every response. The next most common header *content-type* occurs on only slightly fewer responses. However, overall, only nine headers occur on more than half of the responses. Table 4.4b shows the number of unique values for both states for every considered property. Most of the studied features have the same number of unique values for both states. The features body and CSP have more unique values for the logged-in state, and the location feature has more unique values for the anonymous state. In the following, we investigate the seven studied headers, status-codes and content bodies in more depth.

Content-type: Table 4.5a shows the most common content-types observed for both states. The by-far largest group of all responses is images. For most image types, the logged-in state observed slightly more images except image/webp, which was more oftenly observed by the anonymous state. The second-largest group of responses is HTML, with

Header	Count
date	425,780
content-type	418,644
server	382,989
cache-control	351,697
content-length	306,993
last-modified	290,722
etag	263,371
vary	235,971
accept-ranges	227,764
expires	179,119
age	177,221
content-encoding	151,954
strict-transport-security	151,116
x-cache	141,071
x-content-type-options	135,813

(a) The 15 most common headers observed.

Property	Cookies	No cookies
url	215,276	215,276
site	403	403
status-code	25	26
body	183,183	182,054
content-type	164	165
x-frame-options	27	26
content-disposition	64	64
cross-origin-opener-policy	5	5
x-content-type-options	6	6
cross-origin-resource-policy	3	3
content-security-policy	1,445	1,362
location	5,544	6,817

(b) Number of unique values for all relevant properties in both states.

Table 4.4: General statistics of the collected responses.

slightly more results in the anonymous state. The third-largest type of responses is JavaScript. In the fourth place, we have CSS followed by the *Empty* content-type directly after. The *Empty* content-type occurs roughly in 200 more cases for the anonymous visitor. Finally, there is a large tail with other content-types usually occurring the same number of times for both states.

X-Frame-Options (XFO): Table 4.5b shows the most common X-Frame-Options values observed for both states. Roughly 23% of responses use X-Frame-Options. There is almost no difference in the occurrences of *SAMEORIGIN*, but *DENY* occurs more often for the logged-in state. A couple of *ALLOW-FROM* values are invalid, including *ALLOW-FROM ** or *ALLOW-FROM 'self'*, suggesting that developers are mixing up headers as this syntax is valid for *CSP frame-ancestors*. Several responses also contained several X-Frame-Options headers (folded with commas here). Sometimes specifying the same value twice, and sometimes specifying *DENY* once, and *SAMEORIGIN* once, probably resulting in unwanted edge case behavior.

Content-disposition: Table 4.5c displays the most common content-disposition values for both states. The content-disposition header occurs more often for the anonymous visitor. However, this is mostly due to *inline*. The attachment value is more prevalent for the logged-in user, suggesting login-protected downloads. We have trimmed all values until the first *;*, to group all responses with different names together. However, several entries did not specify anything in front of the *;* or only a filename.

Value	Cookies	No cookies	Sites
image/jpeg	56,972	56,048	290
image/png	28,886	27,551	335
image/webp	22,081	23,935	108
text/html; charset=utf-8	16,404	16,699	269
text/html; charset=UTF-8	13,353	13,649	270
application/javascript	10,756	10,746	301
image/svg+xml	10,170	10,166	265
image/gif	7,025	6,899	219
text/css	6,510	6,483	303
Empty	5,833	6,075	222

(a) Content-type

Value	Cookies	No cookies	Sites
Empty	165,450	165,951	395
SAMEORIGIN	32,539	32,546	240
DENY	9,693	9,189	146
sameorigin	2,593	2,627	30
SAMEORIGIN, SAMEORIGIN	1,259	1,230	17
	1,081	1,081	2
deny	860	952	27
Sameorigin	301	301	1
ALLOW-FROM *	259	259	1
ALLOWALL	259	259	13

(b) X-Frame-Options

Value	Cookies	No cookies	Sites
Empty	201,480	199,739	403
inline	13,028	14,799	90
attachment	668	638	29
	29	29	1
filename="jpeg"	7	7	1
filename=pixel.png	5	5	1
filename="png"	2	2	1
filename=310148.jpeg	1	1	1
filename=310078.jpeg	1	1	1
filename=310121.jpeg	1	1	1

(c) Content-disposition

Value	Cookies	No cookies	Sites
Empty	214,898	215,003	403
same-origin	154	154	2
same-origin-allow-popups;report-to="coop"	191	86	1
same-origin-allow-popups	31	31	2
unsafe-none	2	2	1

(d) Cross-Origin-Opener-Policy

Value	Cookies	No cookies	Sites
Empty	147,037	147,702	388
nosniff	67,155	66,551	288
nosniff, nosniff	1,076	1,015	18
nosniff, nosniff, nosniff	5	5	1
Nosniff	2	2	1
"nosniff"	1	1	1

(e) X-Content-Type-Options

Value	Cookies	No cookies	Sites
Empty	210,358	210,353	403
cross-origin	4,916	4,921	12
same-site	2	2	1

(f) Cross-Origin-Resource-Policy

Value	Cookies	No cookies	Sites
Empty	208,889	207,108	397
https://www.instagram.com/accounts/login/	0	169	1
https://accounts.adafruit.com/users/sign_in	74	88	1
/login	27	96	17
/	49	44	36
https://faucetcrypto.com/login	0	56	1
https://authorize.feedbooks.com/user/login	26	26	1
https://www.tomford.com	44	0	1
/features/qualified-electronic-signatures	22	22	1
https://www.inspectlet.com/signin	21	22	1

(g) Location

Value	Cookies	No cookies	Sites
empty	6,122	7,069	307
GIF image data, version 89a, 1 x 1	606	606	19
GIF image data, version 89a, 1 x 1	424	424	4
GIF image data, version 89a, 1 x 1	349	349	1
GIF image data, version 89a, 1 x 1	297	274	8
GIF image data, version 89a, 2 x 2	323	192	5
GIF image data, version 89a, 1 x 1	222	222	1
ASCII text, with no line terminators	216	216	1
ASCII text, with no line terminators	244	184	61
HTML document, UTF-8 Unicode text	203	198	1

(h) Response bodies

Table 4.5: Ten most common values occurring for both states for the considered properties.

Cross-Origin-Opener-Policy (COOP): Table 4.5d reveals that the Cross-Origin-Opener-Policy is not used a lot yet. Mainly a secure value is set for both states. However, one site sets a value for the logged-in state leading to potential XS-Leaks. The unsafe value of *unsafe-none* only occurred two times.

X-Content-Type-Options (XCTO): Table 4.5e presents the observations for the X-Content-Type-Options header. The X-Content-Type-Options header is set on around 31% of responses with a slightly higher number on the logged-in state and several responses setting the header several times.

Cross-Origin-Resource-Policy (CORP): The Cross-Origin-Resource-Policy header is almost never used yet as can be seen in table 4.5f. Only two responses use a secure value *same-site*, whereas a higher number of around 4,920 responses use the insecure value of *cross-origin*. We explain this observation by the need to set the value to use it on other pages using COEP [20] to make use of features such as SharedArrayBuffer.

Location: Table 4.5g shows the observed locations. For the location header, many sites seem to redirect visitors on protected resources to their login page. For some sites, both states are redirected roughly the same number of times to a login page. These are probably sites where the session sharing via cookies failed, so both states are the same.

Response bodies: Table 4.5h displays the most commonly found bodies. For the body, 6,122 times it was empty in the logged-in state, and 7,069 times it was empty in the anonymous state. Most other bodies only occurred once in both states or only once in one state and never in the other. Most of the often occurring bodies appear to be 1x1 Pixel images that websites probably use as tracking pixels.

Status-code: As seen in the previous chapter, status-codes are one of the most important properties for XS-Leaks, and some status-code differences can be distinguished regardless of other properties. Table 4.6 displays the most common observed status-codes. Both states have the standard status-code *200* for most responses. However, the anonymous visitor has around 3,400 occurrences less of it than the logged-in user. Most redirection codes occurred significantly more often for the anonymous visitor, with a difference of 1,623 for code *302* and 171 for code *307*. Client error responses also occur more often for the anonymous visitor, with relatively small differences for *404* and *403* and large differences for *429*, *401*, and *400*. The latter two seem to be normal server behavior if visitors access content meant for logged-in users. The *429* rate-limiting status-code is

Value	Cookies	No cookies	Sites
200	204,522	201,177	397
302	3,050	4,673	305
301	3,327	3,351	337
204	1,008	979	23
206	826	825	50
404	796	840	158
403	520	720	85
429	152	732	8
101	437	247	36
401	31	649	60
202	260	259	6
400	27	313	27
307	72	243	22
303	66	106	17
500	62	72	27
520	59	10	1
503	18	26	10
308	18	18	6
502	9	10	3
504	7	6	4
304	4	4	1
410	2	4	3
205	0	5	1
440	0	5	1
451	1	1	1
550	1	1	1
405	1	0	1

Table 4.6: All status-codes for both states.

interesting. Both states request every URL the same number of times with similar speeds, and the logged-in state even performs additional requests to the server (e.g., POST). This observation suggests that some websites have stricter rate limits for anonymous visitors than for users of their service. If the rate-limiting is done server-wide, every URL of the service might be used as an XS-Leak oracle if an attacker counts the number of requests until the rate limit is detected. We note that we do not perform such tests and only accidentally got rate-limited. Some other results are that the non-standard *440 Login Time-Out* code only occurred for the anonymous visitor, and another non-standard code: *520 Web Server Returned an Unknown Error*, occurred almost six times more often for the logged-in state. A *101* code suggesting a successful creation of a websocket appeared twice as likely for logged-in users. For the other *2XX* codes and *5XX* codes, there are no noteworthy differences between both states. We did not observe other codes such as *419* or *100*.

Complete response	Minimal pruning	Basic pruning	Count
non SD	non SD	non SD	4,972
SD	non SD	non SD	38,267
SD	SD	non SD	138,822
SD	SD	SD	33,215

Table 4.7: State-dependent URLs according to different definitions.

4.4.2 Response pairs

In the above section, we presented the overall statistics of all observed responses. However, of particular interest are pairs of responses belonging to the same URL as only SD-URLs with two different responses are possible candidates for XS-Leaks. This section gives an overview of how many SD-URLs exist, and within the SD-URLs, which combinations occurred, e.g., code 200 switched to code 401.

State-dependent URLs (SD-URLs): For a URL to be vulnerable, it has to be an SD-URL, but not every SD-URL is exploitable. The crawling framework requested every first-party GET request without cookies immediately after every request with cookies. Using this methodology, we collected two entries for every URL. It is important to note that the following statistics are only estimates of SD-URLs. For a real SD-URL, the difference in the responses has to be caused by the state difference studied. However, the response of a URL can differ for many other reasons as well. One reason is timeshift, as resources can change all the time. The replay of requests in the proxy minimizes the time difference between the two requests. However, a small time difference still exists. Another reason is server-side randomness in the responses. Such randomness can be caused by A/B testing or load balancers and similar not related to the state. Lastly, the GET request might not be idempotent. For example, a server implementing a `/delete/<id>` GET endpoint might return a 200 status-code for the first request and a 404 status-code for a second request as the resource is already deleted. Due to all the above reasons, responses can differ without depending on the state difference we control. Thus, the SD-URLs statistics likely overestimate the actual number of SD-URLs.

Table 4.7 lists the number of state-dependent and non-SD-URLs according to different definitions of what counts as the same response. Checking for complete equality, i.e., all headers, status-code, and body have to be equal, reveals that only 4,972 out of all 215,276 are non-SD-URLs according to this definition. This fact can be explained by the fact that headers that introduce much randomness such as date, server, cache-control, last-modified, etag, vary, expires, and age are set for most responses as can be seen in table 4.4a. Removing this set of high variance headers that should not influence XS-Leak

Browser	mean	std	min	median	max
Firefox	3.61	3.44	0	2.0	12
Chrome	3.26	3.47	0	2.0	12

Table 4.8: Average number of leak URLs per basic pruned URL.

behavior an additional 38,267 URLs are regarded as non SD-URLs. Using the basic pruning step, i.e., ignoring all headers other than the seven headers we tested in the previous chapter, an additional 138,822 URLs are marked as non-SD, and only 33,215 URLs remain as SD-URLs. Out of the remaining 33,215 URLs, only 1,008 URLs have the same body for both responses suggesting that SD-URLs that only differ in the headers or code are rare.

Not every SD-URL is distinguishable by the considered leak methods. For example, two responses that both have XFO=deny and COOP=same-origin set and differ in the body cannot be distinguished by the framecount method as no access is possible for both responses. We can use the decision trees to get a better estimate of which SD-URLs the tested methods can distinguish. In this case, the tree pruning module would correctly remove a SD-URL with such a response pair from the dynamic tests as the tree correctly predicts that no access is possible for both responses.

The additional tree pruning step reduces the number of URLs from 33,215 to 30,485 in Chrome and 30,614 in Firefox. The additional reduction of the trees might seem small at first. However, the trees mainly decide which inclusion channel should be tried for a URL and not that a URL should not be tried at all. Table 4.8 displays the average number of tested leak URLs per basic pruned URL. On average, we only tested 3.4 out of 12 inclusion methods for every remaining URL. All responses that redirect to different locations are tested in all 12 inclusion methods, as we do not know the features of the final response.

Response pairs: In the previous section, we have seen which values exist how often for both states without considering the exact response pairs. These results indicate server behavior. For example, if the anonymous visitor observes code 401 more often than the logged-in user, it indicates that some resources are only accessible by logged-in users. However, this is not necessarily the case as one state could get response code 401 for 500 URLs, and the other state could get the code 401 for 500 different URLs leading to skewed statistics. To investigate the changes further, we list how often a property changed between the states for the URLs in the following.

Value cookies	Value no-cookies	URLs	Sites
image/png	image/webp	1,254	27
image/jpeg	image/webp	591	23
text/html; charset=UTF-8	text/html; charset=utf-8	133	22
text/html; charset=utf-8	Empty	81	18
text/html;charset=UTF-8	Empty	40	9
Empty	text/html; charset=utf-8	20	9
text/html; charset=utf-8	text/html; charset=UTF-8	58	8
application/json; charset=utf-8	text/plain; charset=utf-8	27	8
application/json	text/html; charset=UTF-8	19	8
text/html; charset=utf-8	text/html	16	8
application/json; charset=utf-8	text/html; charset=utf-8	134	7
text/html	text/html; charset=utf-8	51	7
image/gif	image/webp	11	7
application/json; charset=utf-8	Empty	30	5
image/gif	text/html; charset=utf-8	27	5
application/json	text/html; charset=utf-8	9	5
Empty	text/html	79	4
text/plain; charset=UTF-8	application/json	50	4
application/json; charset=UTF-8	text/html; charset=UTF-8	49	4
text/html; charset=UTF-8	text/html	9	4

(a) Content-types

Value cookies	Value no-cookies	URLs	Sites
SAMEORIGIN	Empty	631	38
Empty	SAMEORIGIN	201	18
DENY	Empty	86	10
Empty	DENY	51	7
sameorigin	Empty	9	5
DENY	SAMEORIGIN	432	3
SAMEORIGIN, SAMEORIGIN	Empty	24	3
SAMEORIGIN, SAMEORIGIN	SAMEORIGIN	5	2
deny	Empty	3	2
SAMEORIGIN,deny	deny	57	1
SAMEORIGIN, sameorigin	sameorigin	43	1
DENY	deny	37	1
SAMEORIGIN,SAMEORIGIN,deny	deny	1	1

(b) X-Frame-Options

Value cookies	Value no-cookies	URLs	Sites
Empty	inline	1,821	31
inline	Empty	50	6
attachment	Empty	30	4

(c) Content-disposition

Value cookies	Value no-cookies	URLs	Sites
same-origin-allow-popups;report-to="coop"	Empty	105	1

(d) Cross-Origin-Opener-Policy

Value cookies	Value no-cookies	URLs	Sites
nosniff	Empty	858	35
Empty	nosniff	193	21
nosniff, nosniff	nosniff	61	2

(e) X-Content-Type-Options

Value cookies	Value no-cookies	URLs	Sites
Empty	cross-origin	15	1
cross-origin	Empty	10	1

(f) Cross-Origin-Resource-Policy

Value cookies	Value no-cookies	URLs	Sites
Empty	/login	60	12
/	Empty	10	6
Empty	/	9	6
Empty	/login.php	7	2
/dashboard	Empty	2	2
Empty	/login?ref=/myaccount	2	2
Empty	/login?ref=%2Fmy-groupons%2Fend-points%2Fvoucher_...	2	2
Empty	/login?ref=%2Fmystuff	2	2
Empty	/login?ref=%2Fmystuff%2Fbuy_it_again	2	2
Empty	/login?ref=%2Fwishlist	2	2
Empty	/login?return_to=/subscription_center	2	2
Empty	/signin	2	2
Empty	https://www.instagram.com/accounts/login/	168	1
Empty	https://faucetcrypto.com/login	55	1
https://www.tomford.com	Empty	44	1
Empty	https://theoldreader.com/users/sign_in	37	1
https://ahrefs.com/user/login	Empty	32	1
/browse	Empty	27	1
Empty	https://portal.nofraud.com/users/sign_in	26	1
Empty	/auth/login/sanchez-mendoza/	25	1

(g) Location

Value cookies	Value no-cookies	URLs	Sites
200	302	2,140	194
302	200	497	99
200	401	497	46
200	403	266	28
302	403	20	16
200	400	210	15
200	307	164	13
200	404	73	12
200	301	45	11
403	200	83	9
200	303	41	9
301	200	31	9
200	500	9	8
200	429	568	6
404	403	12	6
307	200	8	5
200	503	10	4
403	302	5	4
101	400	76	3
302	301	30	3

(h) Status-code pairs

Table 4.9: Twenty most common differing response pairs for all considered properties.

Content-type: Table 4.9a displays the most common content-type changes for all response pairs. The largest group of changes is from *image/(png|jpeg)* to *image/webp*. This change seems to be an artifact of first versus second request and not connected to the actual state difference. Other changes that occur relatively often, however, behave as expected. There are relatively many changes from various content-types such as *application/json* to *text/html*, *text/plain* or *Empty*. Interestingly, several responses changed from *Empty* to *text/html*. These could be non-idempotent GET requests that return an error page when the action is not possible anymore.

X-Frame-Options (XFO): Table 4.9b displays the changes for the X-Frame-Options header. The settings are usually more secure for the cookie state, but it can also be the other way round. 631 URLs on 38 sites change from *SAMEORIGIN* to *Empty*, whereas only 201 URLs on 18 sites change from *Empty* to *SAMEORIGIN*. Six of these sites are in both sets and change the value in both directions. It is also interesting to note that on one site, the value changed from *DENY* to *deny*. These results suggest that several applications do not set the XFO flag in a single place in their application and sometimes forget to set it at all for some resources.

Content-disposition: Table 4.9c displays the changes for the content-disposition header. For many URLs, *inline* is only set for the anonymous state, suggesting it might be set only on an error page. However, the default behavior of this header is *inline*, so this difference cannot be leaked if it is the only difference. For four sites, the difference between content-disposition *attachment* and *Empty* is detectable.

Status-code: The results for status-codes can be seen in table 4.9h. There are many changes from code 200 to redirection codes (302, 307, 301) and various error codes (401, 403, 400, 404, 429, 500). The high number of URLs where the status-code changed from 200 to 429 indicates that these six sites have lower rate-limiting thresholds for anonymous visitors. There are also some changes from error or redirection codes to code 200, but these are less common than in the other direction.

Remaining properties: Table 4.9d shows that only one site has state-dependent COOP settings. Table 4.9e depicts that 35 sites set the nosniff value only for the logged-in state for some URLs. Ten out of these 35 sites and another eleven sites only set the nosniff value for the anonymous state for some URLs. Table 4.9f shows that CORP is only set for the logged-in state on one site and only set on requests without cookies on another site. Table 4.9g demonstrates that the location header is often empty for the logged-in

Security flag			Cookies	Sites
sameSite	secure	httpOnly		
Lax	False	False	855	225
		True	43	21
	True	False	67	38
		True	80	60
None	False	False	74	13
		True	14	6
	True	False	423	117
		True	220	99
Not set	False	False	3,613	345
		True	231	129
	True	False	261	133
		True	262	144
Strict	False	False	16	16
		True	1	1
	True	False	80	44
		True	6	6

Table 4.10: Usage of security flags for cookies by individual cookies and sites.

state and redirects to `/login` or `/signin` or similar in the other state. For some URLs, it only redirects logged-in users, however.

4.4.3 Cookie statistics

The pipeline also collects all the cookies on each of the tested sites. We can use this information to gather information about the cookie usage of these sites. Cookies are essential for XS-Leaks as they are the default state-transmission method used by servers. If the cookies are not attached to the requests induced by the attacker, the server cannot distinguish between requests belonging to different states. In such cases, the server will reply with the same response regardless of the user's actual state, preventing the successful execution of an XS-Leak.

There are three security flags for cookies: *httpOnly*, *secure*, and *SameSite*. The last one is of particular interest for XS-Leaks as a setting of *Lax* will cause modern browsers to not send them along with most cross-site requests except for requests caused by *window.open*. A setting of *Strict* will instruct the browser never to send them along with cross-site requests. For more details, we refer to [58].

In total, we analyze 6,246 cookies for 382 sites. Unfortunately, due to a server crash, we lost the cookie information for another 21 tested websites. Table 4.10 summarizes the cookies found according to the different security-relevant attributes. Out of these 6,246

Values	Sites
Lax, Not set	103
Lax, None, Not set	83
Not set	81
None, Not set	33
Lax, None, Not set, Strict	27
Lax, Not set, Strict	22
Lax, None	10
None	9
Lax	5
Not set, Strict	5
None, Not set, Strict	2
None, Strict	1
Lax, None, Strict	1

Table 4.11: SameSite settings for cookies observed on sites.

cookies, 857 set the httpOnly flag to True, and 1,399 set the Secure flag to True. For SameSite, we have 1,045 cookies with the *Lax* value, 731 with the *None* value, and 103 with the *Strict* value. Another 3,613 cookies did not set any flag.

For CSRF [60], the recommendation is to set all session cookies to *Lax* or better even to *Strict* such that attackers cannot cause damage. For other cookies, the SameSite attribute is sometimes not set at all, or for better cross-origin interoperability, set to *None*. We highlight that even if all session cookies are set to *Strict*, and browsers thus do not attach them to any cross-origin requests, XS-Leaks might still be possible due to other cookies with less protective settings. To illustrate, imagine a website with a dark and a light theme. The website saves the user preference in a non-session cookie set to SameSite="None" such that even if other websites frame the website, the user preference is complied with. If the dark and light themes differ in the number of frames on the page, an attacker can easily infer the state with the object-properties_framecount method. If the default setting on the website is *light* and one observes the number of frames for *dark*, one knows that the victim changed this setting. If the button to change it is only accessible to logged-in users, we can even reason that the victim is logged in as it is unlikely that they changed the value of the cookie in the developers' console. Other similar use cases could exist on a variety of websites.

Next, we investigate how different websites are using the SameSite setting and if they use different values on different cookies. Table 4.11 displays all configurations of SameSite cookies on sites found in the tests. The most common setting is that the website sets *Lax* on some cookies and nothing on others (103). The next most common behavior is that some cookies are *Lax*, some are *None*, and some are not set (83). The third place is taken by sites not setting any SameSite flag on all cookies (81). In addition, nine

websites set *None* on all cookies, and five websites set *Lax* on all cookies, and no website sets *Strict* on all cookies. Most of the other possible combinations exist as well in varying occurrences.

4.4.4 Vulnerable endpoints

The pipeline's primary goal is to automatically find and confirm XS-Leaks on an extensive range of websites. The previous sections described potential security issues on sites such as misconfigured headers and cookies. However, such issues do not automatically mean that a site is vulnerable to XS-Leaks. This section presents how many XS-Leaks the last part of the pipeline, the dynamic confirmator, has found on all tested sites and analyzes the difference between browsers, inclusion methods and leak methods.

Both state creation mechanisms together managed to login on 430 websites. Out of these, at least 29 websites could not be crawled correctly, mainly due to TLS problems. Due to an unfortunate hardware issue on the server, we could not save the results of the dynamic confirmator for 67 sites. After fixing the problem and rerunning the pipeline on these sites, the state creation failed for 21 sites, and we could not retest them. In the end, for 383 sites, at least one URL was crawled and saved in the database. Out of these sites, 352 sites had at least one potentially vulnerable URL according to the static pruning component, and we started the dynamic confirmator to confirm the potential leaks. Out of these, 318 sites remained after the first step with at least one URL that observed distinguishable responses. These URLs got retested to lower the chance of false positives due to non-state-dependent differences in the responses. After the second confirmation, a leak URL, i.e., a combination of inclusion channel and target URL, is only considered vulnerable if the observed property differed the *same* way twice. As an example, we consider the framecount method. This method records the number of frames N for the first test and the number M for the second test, and we denote it with (N, M). Observing a framecount of (0, 0) for one state and (3, 4) for the other state would be considered *same* way, but an observed framecount of (4, 3) for one state and (3, 4) for the other state would not be considered as *same* way as both values occur for both states. After testing the potential leak URLs a second time, the pipeline found XS-Leaks on 258 sites on either browser. It found 215 vulnerable sites on Chrome and 230 vulnerable sites on Firefox. 189 sites have at least one vulnerable URL in both browsers.

Vulnerability overview: Table 4.12 gives an overview of all vulnerabilities found. On the 258 sites with at least one found vulnerability a total of 3,625 unique vulnerable URLs and a total of 7,559 vulnerable leak URLs in either browser exist. These numbers heavily vary between sites, browsers, inclusion methods, and leak methods. Out of the

Group	URLs any browser	URLs both browsers	URLs only one browser	URLs Firefox	URLs Chrome	Sites both browsers	Sites only one browser	Sites Firefox	Sites Chrome
all	3,625	1,638	1,987	2,976	2,287	187	71	230	215
audio	40	2	38	18	24	2	10	7	7
embed	661	8	653	623	46	6	38	35	15
embed-img	969	0	969	954	15	0	55	53	2
iframe	1,001	119	882	918	202	48	60	89	67
iframe-csp	550	116	434	470	196	39	65	85	58
img	10	4	6	6	8	2	0	2	2
link-prefetch	64	0	64	42	22	2	14	8	10
link-stylesheet	68	0	68	9	59	0	12	2	10
object	937	13	924	929	21	8	39	45	10
script	851	28	823	822	57	5	37	36	11
video	33	2	31	13	22	1	8	3	7
window.open	2,375	1,465	910	1,816	2,024	125	63	159	154
event-list	1,182	50	1,132	1,111	121	17	53	60	27
gp-securitypolicyviolation	176	6	170	165	17	5	23	23	10
gp-window-onerror	814	0	814	811	3	0	37	35	2
gp-window-postMessage	444	95	349	365	174	25	34	52	32
op-el-media-error	23	0	23	0	23	0	6	0	6
op-el-naturalWidth	10	4	6	6	8	2	0	2	2
op-frame-count	2,321	1,328	993	1,734	1,915	126	58	156	154
op-win-history-length	413	239	174	322	330	76	57	106	103
op-win-opener	52	44	8	48	48	3	0	3	3
op-win-origin	255	108	147	233	130	12	27	34	17

Table 4.12: Vulnerable URLs by inclusion methods and leak methods for browsers and sites.

	URLs any browser	URLs both browsers	URLs only one browser	URLs Firefox	URLs Chrome
mean	14.05	6.35	7.70	11.53	8.86
std	40.54	17.41	29.43	39.52	19.64
min	1.00	0.00	0.00	0.00	0.00
50%	3.00	1.00	2.00	2.00	2.00
max	560.00	136.00	424.00	560.00	136.00

Table 4.13: Summary of vulnerable URLs discovered by site.

3,625 vulnerable URLs only 1,638 URLs were confirmed in both browsers and the other 1,987 URLs were only confirmed in one browser. Table 4.13 provides a summary of the data for each site showing that there are large differences between sites, with many sites only having one vulnerable URL and one site contributing 560 vulnerable URLs.

Inclusion methods and SameSite differences: Looking at the different inclusion methods, one can see that Firefox found more vulnerable URLs for most of them. In Chrome, most methods did not work on a large set of URLs, except for *window.open*, making up more than 88% of the discovered vulnerable URLs (61% in Firefox). We explain this difference by the fact that except for *window.open* all inclusion methods do not work if the state-defining cookie has the SameSite setting *Lax*. As shown earlier, many cookies do not set the SameSite flag. Chrome defaults to *Lax* in this case, whereas Firefox defaults to *None*. With the collected data, we can estimate how many sites would not be vulnerable in Firefox if they would switch to the new default *Lax* and would stop accepting *None* cookies that do not set the *Secure* flag. Every site that has zero vulnerable URLs for all inclusion methods except *window.open* in Chrome but has at least one vulnerable URL in Firefox is a likely candidate of a site that would be more secure if Firefox would switch their default behavior. This approach produces only an estimate as other issues such as failed session sharing only in Chrome, or different parsing behavior can also be why a site is only vulnerable in Firefox. This approach estimated that 60 out of 258 sites are insecure due to the SameSite inconsistencies. Subtracting these sites from the dataset, 1,175 URLs are vulnerable in both browsers, and 1,078 URLs are vulnerable in only one browser. This result indicates that many issues arise because the different leak methods work slightly differently in both browsers, as explored in more depth in the previous chapter.

Some exceptions where Chrome is more vulnerable than Firefox are link-stylesheet, video, and audio. For link-stylesheet, this is due to Chrome allowing almost any content-type for stylesheets, whereas Firefox applies strict mime-type checking by default and only allows text/css as a valid content-type. For video and audio, this is due to the mediaError bug discovered in the previous chapter [71]. The image inclusion method rarely works for both browsers, suggesting that the well-known case of images only available for authenticated users [42, 90] is not a big issue anymore. The differences between the browsers for the inclusion methods embed, embed-img, object, and script are particularly large and cannot only be explained by SameSite cookies. For embed, embed-img, and object, one reason is the fact that they behave almost identical to IFrame or image in Chrome, whereas they behave differently in Firefox [99]. For script, one primary reason is the CORB implementation of Chrome [19].

Working leak methods: Next, we present the results on the leak methods. First, we focus on the methods mainly related to *window.open*. The framecount method is the best working one in both browsers with almost 2,000 URLs reported as vulnerable each. The object-properties_window-origin method works on 255 URLs. The method works on around 100 more URLs in Firefox. This difference is mainly because these methods additionally often work for the inclusion method IFrame-CSP and not only for *window.open* in Firefox and seldom in Chrome as can be seen in table 4.14. This difference can be due to SameSite behavior on one hand. On the other hand, it can be since the blocked frame replacement is same-origin in Firefox, making it similar to global-properties_securitypolicyviolation, which is not the case in Chrome where the blocked frame is not accessible. The object-properties_window-opener method works almost identically in Chrome and Firefox. The object-properties_window-history-length method works slightly more often in Chrome. The global-properties_postMessage method works more often in Firefox, this is mainly due to SameSite cookies as it also works for the inclusion methods IFrame, embed and object in Firefox.

Next, we focus on the methods not possible with *window.open*. Most of these methods work more often in Firefox. This difference is, on the one hand, due to the different SameSite defaults. On the other hand, it is due to differing parsing oddities and edge-case handling in the browsers. One exception to the rule is the mediaError channel that only works in Chrome due to the discovered bug [71]. Another recognizable difference is that the global-properties_window.onerror method worked three times in Chrome and 811 times in Firefox. Here, the explanation cannot only be SameSite. Instead the CORB implementation in Chrome stopped several of these leaks as it replaces JSON or HTML bodies with empty content that does not trigger an error [19]. Several methods do not occur in table 4.12. We decided to drop several methods from the previous chapter that, according to the decision trees, are identical for our purposes. For example, we only included the dimension properties object-properties_el-naturalHeight and object-properties_el-naturalWidth in the analysis and dropped object-properties_el-Height and object-properties_el-Width. Additionally, out of the 18 tested methods in this chapter, the three methods object-properties_el-duration, object-properties_el-videoHeight, and object-properties_el-videoWidth did not work a single time. This non-occurrence suggests that it is rare for video or audio resources to only be accessible by logged-in users or that it is hard to find such resources.

Working leak channels: The leak channel results in table 4.14 present additional results not seen in the above data. First, we focus on the channels not using *window.open* as the inclusion method. All of these methods should, in general, be more likely in Firefox

Leak channel	URLs any browser	URLs both browsers	URLs only one browser	URLs Firefox	URLs Chrome	Sites both browsers	Sites only one browser	Sites Firefox	Sites Chrome
Audio									
event-list	23	2	21	18	7	2	7	7	4
op-el-media-error	20	0	20	0	20	0	5	0	5
Embed									
event-list	576	2	574	574	4	1	16	16	2
gp-securitypolicyviolation	26	4	22	21	9	3	8	9	5
gp-window-postMessage	163	4	159	128	39	4	21	17	12
Embed-img									
event-list	845	0	845	830	15	0	25	23	2
gp-securitypolicyviolation	55	0	55	55	0	0	10	10	0
gp-window-postMessage	181	0	181	181	0	0	31	31	0
IFrame									
event-list	557	2	555	557	2	1	10	11	1
gp-securitypolicyviolation	32	4	28	29	7	3	6	8	4
gp-window-postMessage	271	14	257	209	76	10	28	33	15
op-frame-count	430	73	357	400	103	7	37	37	14
op-win-history-length	100	31	69	78	53	36	32	52	52
op-win-origin	4	1	3	4	1	1	2	3	1
IFrame-CSP									
event-list	125	2	123	125	2	1	18	19	1
gp-securitypolicyviolation	120	5	115	112	13	4	17	17	8
gp-window-postMessage	166	23	143	117	72	8	25	27	14
op-frame-count	303	75	228	266	112	9	34	36	16
op-win-history-length	75	18	57	56	37	22	34	42	36
op-win-origin	100	1	99	100	1	1	15	16	1
Image									
event-list	10	4	6	6	8	2	0	2	2
op-el-naturalWidth	10	4	6	6	8	2	0	2	2
Link-prefetch									
event-list	64	0	64	42	22	2	14	8	10
Link-stylesheet									
event-list	68	0	68	9	59	0	12	2	10
Object									
event-list	846	7	839	836	17	5	22	24	8
gp-securitypolicyviolation	60	5	55	59	6	4	8	11	5
gp-window-postMessage	152	4	148	149	7	3	18	20	4
Script									
event-list	815	28	787	787	56	5	31	30	11
gp-window-onerror	814	0	814	811	3	0	37	35	2
Video									
event-list	15	2	13	13	4	1	4	3	3
op-el-media-error	18	0	18	0	18	0	4	0	4
Window.open									
gp-window-postMessage	120	72	48	87	105	13	8	16	18
op-frame-count	2,188	1,296	892	1,619	1,865	116	59	144	147
op-win-history-length	281	188	93	227	242	19	17	30	25
op-win-opener	52	44	8	48	48	3	0	3	3
op-win-origin	158	108	50	136	130	12	14	21	17

Table 4.14: Vulnerable leak channels by browser and site.

due to the different SameSite defaults. For some channels, the difference is particularly striking, warranting further investigation.

The first finding is that the global-properties_securitypolicyviolation method worked 186 times for inclusion methods other than IFrame-CSP. Investigating this oddity, we could confirm that this is due to the embedding documents specifying a CSP frame-ancestor policy that does not allow the attack page to frame them. In such a case, the browser throws a securitypolicyviolation event on the parent page. As this leaks cross-site information, browsers should not throw an event on the parent page. The issue was already discovered and fixed independently in Chrome [83] but we reported the issue to Firefox where it is still possible to exploit [79].

As a second channel, we investigate the event-list channel. For some inclusion methods such as audio and image, the results between Chrome and Firefox are similar and can mainly be explained by the SameSite behavior. The inclusion methods that found the most vulnerable URLs for event-list, are IFrame-CSP, object, embed, and embed-img. For these inclusion methods, the two browsers behave fundamentally differently. Embed and object behave mostly similar to IFrame in Chrome, whereas they behave vastly differently on Firefox. Embed-img behaves similar to embed in Firefox and similar to img in Chrome. The IFrame behavior is also not identical for both browsers. All these differences, in addition to the differing SameSite default, result in both browsers finding almost complete disjoint sets of URLs vulnerable. Firefox has drastically more vulnerable URLs for these cases but still does not include all cases found in Chrome.

Next, we highlight some of the behavior of both browsers and which responses they can distinguish. For the IFrame, object, and embed inclusion methods, both Chrome and Firefox can distinguish responses where only one response performs a redirection using the meta refresh tag, the refresh header, or by JavaScript code. In Firefox, embed-img works as well. Such redirections, however, are rare in practice and only account for two vulnerable URLs in the collected data. In Firefox, responses that only set *CSP: frame-ancestors* on one response can also be distinguished by all four inclusion methods. For IFrame one load event less is fired, for embed(-img) no event is fired at all, and for object an error event is fired in such cases. Firefox can also distinguish responses that have an error status-code such as 401 and responses with a success code such as 200 for object, embed, and embed-img. Firefox can also distinguish responses that set X-Frame-Options from responses that do not set X-Frame-Options using the events fired for object, embed, and embed-img. In Chrome, the distinction for *CSP: frame-ancestors* and X-Frame-Options works as well, but only for the object inclusion. The behavior in Chrome is unwanted as object should not leak more information as IFrame. Thus we created another bug report [72]. The behavior in Firefox threatens more websites as

a larger attack surface exists. However, Firefox plans to rewrite their complete object and embed code to make it behave more like IFrame [99]. Therefore, we refrained from opening additional bug reports until we clarified more details about the exact nature of this issue and their planned changes. Embed-img works the same way as img in Chrome.

The global-properties_postMessage method works similarly for *window.open*, roughly three times more often on Firefox for most other inclusion methods and zero times for embed-img in Chrome.

In the end, we want to highlight one aspect of the best working method overall framecount. For the *window.open* inclusion method, the leak method works for 1,619 URLs in Firefox and 1,865 URLs in Chrome. One reason that accounts for 48 of the URLs only found in Chrome is the decision of Chrome to not return any framecount for broken window references instead of just returning 0 as a safe default. Window references can break for two reasons. One reason is the setting of a Cross-Origin-Opener-Policy that cuts off the connection, which occurred on two sites. The other reason occurs when the response causes a download, e.g., through content-disposition=attachment, which occurred on one site.

4.4.5 Potential issues

Every tool that automatically deals with finding vulnerabilities faces the problem of false positives and false negatives. False negatives are due to missing methods or insufficient coverage of the tested site. They mean the tool reports a site as invulnerable even though it is or reports not all vulnerable URLs. False negatives are harmful as they convey a false sense of security. However, if one knows about the limitation and does not only rely on one tool to check for the security of a website, they do not change the fact that the tool found real vulnerabilities. False positives, on the other hand, are a more serious issue for a tool like the does-it-leak pipeline. First, they skew the scientific reports as sites and URLs are reported as vulnerable that are, in fact, secure. Second, suppose a developer uses an automatic tool to find vulnerabilities, and the tool presents many wrong cases. In that case, the developer loses much time and loses interest in the tool, often leading to less secure sites [93]. In the following, we try to estimate whether we have false positives in the data and how serious of a problem it is. Then, we will discuss the potential issue of unpredictable URLs. Finally, we will discuss the issue that the found state differences are not necessarily between a logged-in user and an anonymous visitor.

False positives: We created the pipeline with the goal of not producing false positives and sacrificed this for a higher number of false negatives. For example, every URL reported as vulnerable has to be leaky the *same* way twice. Still, we cannot guarantee that the pipeline did not report any false positives. False positives can have two main reasons. First, two repetitions were not enough, and due to bad luck, an actual random response is reported as vulnerable. Second the definition of *same* is not strict enough. Investigating false positives in the pipeline is challenging for three main issues. First, the number of reported vulnerabilities is too high to check manually. Second, as seen previously, it can be that the necessary state creation component does not work anymore. Third, even if one manually attempts to confirm the vulnerable URLs and does not manage to confirm them, it does not automatically mean that the reported URL is a false positive. Instead, it could also be that the response to the URL has changed as we are attacking a moving target, and websites change how they behave every day.

We came up with a different method to see if a reported vulnerability might be a false positive and which methods are more prone to them. We investigate the observed results for every leak channel to compare whether the same result was observed for the logged-in state and for the anonymous state or if we only observed every single value for one state. Applying this method to the complete dataset is not helpful as the negative case for one site can be positive for the other. Table 4.9g shows that, for example, some sites redirect logged-in users, whereas other sites redirect anonymous visitors. Getting the data per site is not without issues either, as one site could, for example, redirect logged-in users on one URL and anonymous visitors on another URL. Still, this gives some insight into potentially problematic methods as well as into methods that appear stable.

Table 4.15 summaries the observation pairs for each inclusion channel by browser. Several methods such as event-set are dropped from this analysis as they behave almost exactly as other methods. An observation pair consists of the values observed for both states. We aggregate the pairs per site for the table, i.e., we count the same value pair twice if it occurs on two different sites. The first column shows the number of unique pairs, and the second column shows the number of URLs for which we observed these pairs. The other two columns show the number of pairs and URLs left if we only consider the pairs observed in both permutations for a site. There are 2,267 value pairs observed, and only 168 of these occur in both permutations on a site.

Most inclusion channels have zero potential false positives, according to this estimation. The framecount method, however, has a high potential number in both browsers. In addition, the object-properties_window-history-length has a high potential number in Chrome. Other methods that have some potential false positives are global-properties_postMessage, mediaError in Chrome, and event-list. It is important to note that these numbers

			Complete Data		Potential FPs	
			Pairs	URLs	Pairs	URLs
Inclusion method	Leak method	Browser				
audio	event_list	Chrome	4	7	0	0
		Firefox	8	18	2	2
	op_el_media_error	Chrome	6	20	2	2
embed	event_list	Chrome	2	4	0	0
		Firefox	17	574	2	3
	gp_securitypolicyviolation	Chrome	5	9	0	0
		Firefox	9	21	0	0
	gp_window_postMessage	Chrome	37	39	2	2
		Firefox	28	128	0	0
embed-img	event_list	Chrome	3	15	2	12
		Firefox	24	830	0	0
	gp_securitypolicyviolation	Firefox	10	55	0	0
	gp_window_postMessage	Firefox	57	181	0	0
iframe	event_list	Chrome	1	2	0	0
		Firefox	11	557	0	0
	gp_securitypolicyviolation	Chrome	4	7	0	0
		Firefox	8	29	0	0
	gp_window_postMessage	Chrome	46	76	0	0
		Firefox	57	209	0	0
	op_frame_count	Chrome	18	103	0	0
		Firefox	94	400	2	2
	op_win_history_length	Chrome	52	53	0	0
		Firefox	59	78	2	2
	op_win_origin	Chrome	1	1	0	0
		Firefox	3	4	0	0
iframe-csp	event_list	Chrome	1	2	0	0
		Firefox	20	125	2	2
	gp_securitypolicyviolation	Chrome	8	13	0	0
		Firefox	100	112	0	0
	gp_window_postMessage	Chrome	41	72	0	0
		Firefox	52	117	2	2
	op_frame_count	Chrome	25	112	2	6
		Firefox	62	266	2	2
	op_win_history_length	Chrome	36	37	0	0
		Firefox	43	56	2	2
	op_win_origin	Chrome	1	1	0	0
		Firefox	17	100	2	2
img	event_list	Chrome	3	8	2	5
		Firefox	2	6	0	0
	op_el_naturalWidth	Chrome	7	8	2	2
		Firefox	5	6	0	0
link-prefetch	event_list	Chrome	10	22	0	0
		Firefox	8	42	0	0
link-stylesheet	event_list	Chrome	11	59	2	25
		Firefox	2	9	0	0
object	event_list	Chrome	9	17	0	0
		Firefox	26	836	2	2
	gp_securitypolicyviolation	Chrome	5	6	0	0
		Firefox	11	59	0	0
	gp_window_postMessage	Chrome	6	7	0	0
		Firefox	32	149	0	0
script	event_list	Chrome	12	56	2	22
		Firefox	31	787	2	7
	gp_window_onerror	Chrome	2	3	0	0
		Firefox	38	811	6	18
video	event_list	Chrome	3	4	0	0
		Firefox	3	13	0	0
	op_el_media_error	Chrome	5	18	2	8
window.open	gp_window_postMessage	Chrome	39	105	6	15
		Firefox	31	87	2	5
	op_frame_count	Chrome	419	1,850	50	163
		Firefox	388	1,596	40	138
	op_win_history_length	Chrome	30	242	6	113
		Firefox	37	227	4	6
	op_win_opener	Chrome	3	48	0	0
		Firefox	3	48	0	0
	op_win_origin	Chrome	19	130	4	8
		Firefox	22	136	2	3

Table 4.15: Number of unique observation pairs by site for each channel for the complete data and the potential false positives.

are only estimated as the observed duplicate values belong to distinct URLs. It might be that the different URLs leak in opposite ways, e.g., a site might redirect a logged-in user to `/account` when trying to access `/login` and redirect an anonymous visitor to `/login` when trying to access `/account`. When it is indeed a false positive, it can have several reasons. For example, a not strict enough definition of *same* way, not enough repetitions due to server-site randomness, or that the method is not stable enough in general, or the attack page does not wait long enough for stable results.

Studying the different methods in more detail gives some more insights into these problematic cases. For framecount, there seem to be several sites that have URLs with 0 frames for visitors and N frames for logged-in users, and the other way round. This observation suggests that these are not false positives, but different parts of the website behave differently for both states. At the same time, there are several sites where it is unclear. For future experiments, we suggest increasing the retry number to three for the framecount and postMessage method. In addition, if both states receive messages, we could use a distance function and disregard too similar messages for the cost of potentially including additional false negatives. Finally, for the methods based on *window.open* in Chrome, it seems that the timeouts were not high enough in some cases, even though we vastly increased them from the last chapter.

Unguessable URLs: Many websites use random-looking and unguessable URLs to protect resources. This method of unguessable URLs is, for example, often used for image sharing sites such as Google photos [10]. These secret URLs can be shared among all users in the service, or every user can have their secret URL. In addition, these URLs can be protected by another means of authentication. This concept was studied in depth by Staicu and Pradel. In their paper, they have shown that many sites are vulnerable to so-called leaky images attacks [90]. For XS-Leaks, all cases where the secret URL is the only protection measure cannot be leaky as regardless of the requestor's state, the server returns the same response. Cases where the secret URL is shared between all users and authentication is required result in working XS-Leaks. The last case of interest is where the URLs are secret per user, and the server additionally checks authentication. In this case, only one user has access to the resource using the specific URL. The server will verify if the URL and the authentication details fit together and otherwise block the access. This kind of defense is deployed on several sites, including Facebook [90] and does work for every inclusion method, not only images. Suppose users that are not the URL owner and anonymous visitors receive the same response when accessing it. In that case, the URL cannot be exploited as an XS-Leak as attackers can only attack themselves using these URLs.

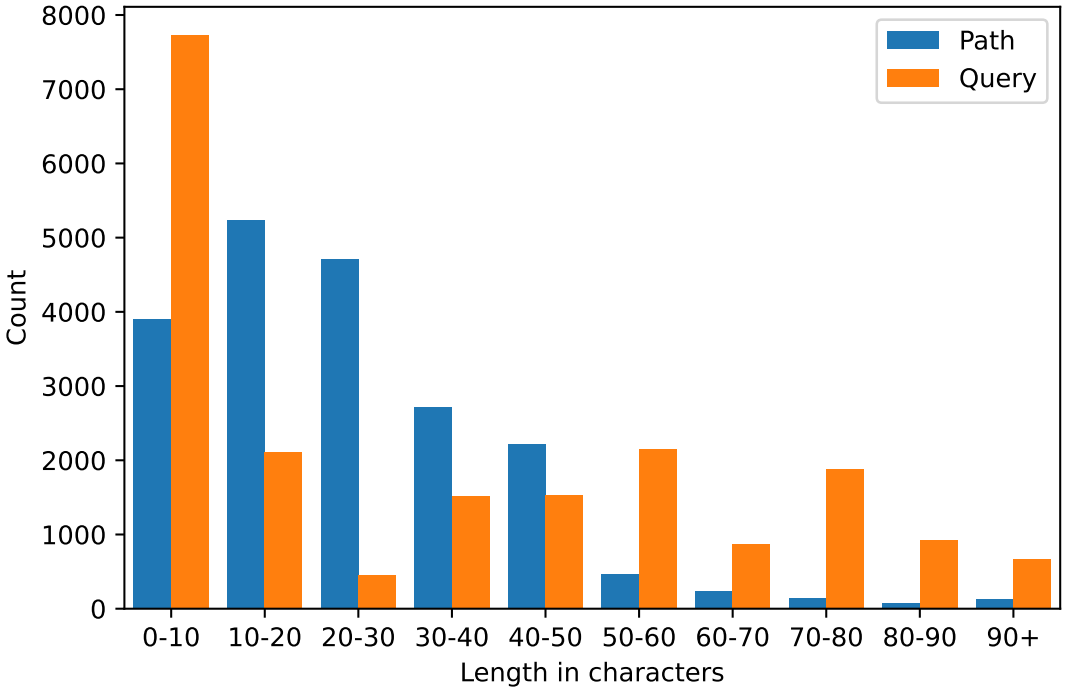


Figure 4.2: URL path and query length histogram of all reported vulnerable URLs.

Due to the way the pipeline is structured, we only created one user per tested website, searched for potentially vulnerable URLs using this account, and then distinguished the created user from an anonymous visitor. This procedure means we were attacking ourselves, and we included such non-exploitable URLs in the results presented in the previous section. It is infeasible to check all found vulnerable URLs regarding this matter and would require testing every URL with another account. Still, we can estimate how big of an issue this is for the presented results. An essential factor of this defense is that the URL needs to have enough entropy, such that attackers cannot predict or brute force the URLs needed to attack users other than themselves based on the URLs they observe for their accounts. The shorter a URL is, the less likely it is that the URL is not user-specific and unpredictable. However, the other way round is not necessarily the case. While testing several very long URLs, we have observed many public URLs that specify a long search query and are exploitable. We did not find a single URL that uses the introduced defense during a manual check, but the sample size was too small to generalize the results. Instead, we want to highlight that many vulnerable URLs are short, meaning they do not have enough entropy to use this defense.

Figure 4.2 presents the distribution of path and query lengths of all reported vulnerable URLs. The median path length is 21 characters, and the median query length is 20 characters. This distribution shows that even if all URLs would use session-specific URLs, many would still be exploitable as they are too short to do so securely.

Leaked state difference: The user-specific URLs described above mean a URL is not exploitable by an attacker. However, they are still working XS-Leaks that can distinguish between the user we created and everybody else. This fact relates to the issue mentioned at the beginning of this chapter: we do not know which state difference information we are leaking. Investigating the leak channel data per site can partly solve this for some sites. For example, we noticed that the framecount for the logged-in state was always zero for one site, whereas it varied for the anonymous state. After manual inspection, we could confirm that the website always redirected logged-in users to a welcome survey. On other sites, we expect other similar behaviors. Such behaviors mean we do not only distinguish between logged-in users and anonymous visitors but often between users of a specific group, e.g., users created today, users not having filled the welcome form, and users not in this group.

Chapter 5

Discussion

In the previous chapters, we have shown that most XS-Leaks still work in major browsers and that many websites are still vulnerable. We created a pipeline to automatically scan websites for XS-Leaks using fake accounts and reported real vulnerabilities. With these facts in mind, we have to discuss the ethical considerations of this work. Additionally, we have to be honest about the limitations of this work. For example, we did not study all XS-Leak methods. Just because we did not find a vulnerability on a website does not mean that the website is invulnerable. Neither does a found vulnerability automatically mean that real users are facing actual risks. We also have compare this work to other research, showing what is new and different and what was already known. Lastly, we would like to make a call to action to get rid of this threat as soon and as effectively as possible.

5.1 Ethics

With this research, we want to make the web a more secure place and do not want to put users at risk or disturb the functionality of the tested websites. We crawled for URLs with only one browser per site. In contrast, the dynamic confirmation of potentially vulnerable URLs has had a maximum of four concurrent browsers testing a site, one per state in both Firefox and Chrome. As we chose the tested sites from the top 20,000 websites and configured relatively high waiting times for the state inference on a site, the requests should not have influenced the service level of any tested website. We have, however, encountered several *429 too many requests* responses, and as a future improvement, we should honor these responses and slow down testing of such sites. Also, we only tried to leak the state difference between the created users and a controlled

visitor. We did not collect any data on real users nor tried to break or steal information from the websites themselves.

A different aspect lies in the creation of the user accounts necessary to perform XS-Leaks vulnerability scanning. Many websites require acknowledging their terms of service (TOS) before creating an account or logging in. Naturally, we could not read any of these TOS, and the creation pipeline just accepted anything. These TOS often only allow real human users to create accounts and sometimes ban bots and vulnerability scanning of their site. We note that the pipeline also does not take anti-bot notices such as *robots.txt* into account and actively tries to prevent being detected as a bot to estimate the extent of the issue for real users more accurately. From specific interest are the regulations of Google and Facebook, which we used for both Gmail access and single sign-on (SSO). These organizations prohibit bot accounts, and the state creation part of the pipeline is breaking several of their TOS. We did not perform any active actions with the created accounts, except for accessing additional websites using SSO, and only created one account per website. We argue that this research benefit outweighs breaking TOS, and testing as many websites as possible will result in more accurate results. As the created accounts are primarily passive and only one per website, they should not influence any of the factors why websites included the bans of bots in their TOS in the first place.

Another aspect lies in the reported vulnerabilities in this thesis and the release of a tool to find XS-Leaks on any website. All bugs found in browsers were reported to the vendors using their official, secure channels as they could potentially put thousands of users at risk. We, however, refrained from reporting the found XS-Leaks to the affected websites. Reporting the found vulnerabilities, sometimes several dozens per site, would require significant effort. It would require manually confirming every reported leak and an educated guess of which state difference is leaked. It also requires finding a suitable reporting point and time for the reporting and answering of follow-up questions. Previous research showed that reporting is a complex task, and only a small amount of websites get fixed quickly following a report [91, 13]. In addition, we did not evaluate what actual state difference is leaked and how it could be abused. Also, websites change every day, meaning many vulnerable endpoints might not be vulnerable anymore. All these reasons resulted in the decision that the cost of notifying every affected website stands in no comparison to the benefits. To further lower the chance of putting users and websites at risk, we decided not to report the vulnerable URLs in this thesis.

The tool released can be used to find XS-Leaks on websites automatically. However, the user needs to provide their state creation scripts as the automatic state creation part is not publicly available. Therefore, it prevents the automatic testing of a high number of

websites. In the end, we believe that the tool is more beneficial for website developers improving their site and other security researchers than for attackers trying to use it for malicious purposes.

5.2 Limitations

As with every scientific work, this work comes with its own set of limitations. Some of them are engineering problems, and we would be able to overcome them with more time and resources. However, other limitations are more fundamental due to the chosen approaches or limited inherently. We first present the limitations of the first framework described in chapter 3 which are mainly related to the chosen scope. Then, we present the limitations of the second pipeline described in chapter 4 which are mainly false negatives and false positives.

5.2.1 XS-Leaks in browsers

The pipeline created to answer the first research question has three main shortcomings: the number of tested browsers and versions, the arbitrary response space, and the set of tested XS-Leak methods. In the following, we explain each point in more detail.

Set of browsers: The first limitation is that we only tested three major browsers in one version each. Testing other browsers, especially ones designed for privacy, such as Tor and Brave, or mobile browsers, could provide additional insights. Studying the exact leak method behavior over versions can also give insights into how XS-Leaks change over time and find incomplete bug fixes. However, there is some trouble in setting up the testing infrastructure. One needs a Selenium 4 driver to use the existing code and not for every browser and version a selenium 4 driver exists. However, nothing fundamentally hinders the addition of additional browsers and versions, and we could add them with more time and resources.

Adequacy of tested response space: The second limitation is the question of the appropriateness of the tested response space. We only tested a limited number of properties with a limited number of values each. We used headers known to influence XS-Leaks and one positive and one negative value each. Using this approach, we tried to cover as much of the entire space of relevant paths in the browser's code while keeping the number of necessary tests reasonable. We did not consider all headers that influence XS-Leaks. Notably, we did not include the Content-Security-Policy header that currently

can influence XS-Leaks. We also did not study a variety of possible values that could lead to all kinds of edge case behavior. For example, one browser might autocorrect an invalid value, whereas another chooses to ignore it.

This problem cannot be solved by just adding resources. One can easily increase the tested response space and add a single additional property or value. However, the growth is exponential, and there are infinitely many possible values for headers as they are strings. One more suited approach could be taking the source code of browsers into account and covering all possible paths in the relevant code. Another approach could be sampling responses from the wild and using a response space constructed out of recorded values to only test combinations that occur in the wild. A third approach would be to start with the created decision trees and only add new values in interesting paths instead of going for the complete combination of properties. If, for example, the decision trees show that responses with CORP will never render images, one does not have to add the new values to the responses with CORP. Still, we think that we covered most of the relevant response space. Additionally, the results in chapter 4 show that the limited response space did not introduce many false negatives, and the created does-it-leak pipeline could find many vulnerable URLs.

Missing methods: The third limitation is the exclusion of several known XS-Leak methods, mainly concerning timing and caching. Unfortunately, the results in this thesis do not say anything about whether these methods still work and which response groups they can distinguish. Adding some of these excluded methods to the pipeline is not easy. They would need severe adjustments and are often influenced by other factors not considered in the current response space, such as the server-processing time.

The timing-based ones need several requests to create accurate timing measurements and establish a timing baseline. These many requests add a burden on the tested sites and the testing infrastructure. In addition, they leak either server-processing time or client-processing time related to resource size. Both factors are currently not represented in the response space and it is unlikely that many of the currently included properties influence the timing. The caching-based methods additionally need a reliable method to purge the cache. Unfortunately, most cache purging methods rely on browser bugs or race conditions, making them unreliable and labor-intensive. Another method only possible in a test environment would be to restart the browser between every request. In the current docker-based framework, this generates massive overhead, and native browsers should be used instead. In addition, the introduction of site isolated cache in modern browsers should mitigate most caching-based methods as different sites do not share a cache anymore [87].

Some XS-Leak methods based on CSS behavior need user interaction. For example, they require a user to click on a button. Users are trained to click on certain elements, such as cookie confirmation buttons. Therefore, these methods can work [57, 50]. However, if the user clicks nowhere or on the wrong space, this might cause incorrect results. In addition, these methods are not automatically exploitable without problems as one needs to find out where the bot has to click. Such automatic clicks are not possible in the current tooling. They would require piping the screen output into a computer vision model and then sending back the coordinates to Selenium to click.

The last removed group consists of methods relying on features not supported in modern browsers anymore, such as AppCache attacks [54]. We could add these methods by testing in a browser version where the feature is still supported. However, studying them does not result in impactful findings as most users use reasonably recent browser versions [11]. Also, it is unlikely that the leaks will get reintroduced as the features were removed entirely and will likely not come back.

5.2.2 XS-Leaks in the wild

The pipeline created to answer the second question also has its own set of shortcomings. First, it inherits all shortcomings from above as it builds upon the results from the first question and shares part of the testing infrastructure. Second, the leak channels global-properties_hasOwnProperty_script and global-properties_getComputedStyle_stylesheet are not properly included as they require high additional engineering effort and have already been studied by others [55]. Third, it also inherits the limitations from other tools it builds on, such as cookiehunter [28]. In the following, we will discuss some of the shortcomings and potential improvements for the future.

State-creation and stateful crawling problems: We could not test many websites as the pipeline failed to register and log in on these sites successfully. In the future, we hope to have a more efficient tool to automatically perform this task or have a standardized way of websites offering test accounts usable by security researchers to scan their websites for vulnerabilities. A more powerful tool would at least need the capability to dodge bot detection measures better and fill captchas. Currently, web developers can manually create a login script for their site or even pass cookies directly to the scanning part of the pipeline. Manual approaches, however, do not scale for security researchers trying to get an overview of the state of the web.

We could not successfully test all websites where the login worked. Reasons include that no session sharing via cookies was possible or the session was invalidated, e.g., by clicking

on a logout button or because the website detected that several users use the same session simultaneously. Other websites detected the crawler as a bot, had insufficient certificates, or used rate-limiting, preventing us from successfully examining them. Some of these problems could be addressed in the future by better monitoring and slower or no parallel requests. Unfortunately, due to insufficient logging and data loss related to a server crash, we cannot precisely determine how often which explanation caused the failure of the crawl or test, which we want to improve in the future.

False positives: We created the pipeline with the explicit goal of producing no false positives. Still, false positives can exist in the collected data due to server-side randomness and other issues. With the false positive estimates in the previous chapter, we could show that the number of false positives is probably relatively small. Still, we suggest testing methods with higher variability, such as global-properties_postMessage and framecount, more than twice per URL in the future. A problem with increasing the confirmation number is increased time, and that it could introduce additional false negatives.

False negatives: False negatives are another problem. We only crawl a limited area of each website and only test the set of supported methods described. Consequently, only because we found no vulnerable URLs on a tested website does not mean that the website is not vulnerable. We suggest that developers change the crawling parameters or directly provide the crawler with a list of all available URLs to achieve higher coverage. If time is not of concern, the pruning module can also be deactivated to test every URL for all inclusion methods.

Future improvements: The primary issue of the problems mentioned above is not that they exist but that we cannot correctly quantify them. One problem is that we are studying a moving target. Due to this, it is not possible to replicate the results or add additional tests after the data analysis step. For example, the websites might have changed already, the login might not work anymore, or server-side randomness interferes.

There are two distinct approaches to how one could address this meta-problem in the future. The first approach is to save all traffic during the dynamic confirmation step. This additional traffic data should make it retroactively possible to study many of the above questions and replay the responses for manual confirmation. In addition, this data should give more insights into SameSite cookies, defenses such as Fetch metadata, and general server-side randomness. Another approach would be to use the does-it-leak pipeline in a more controlled environment that disables most sources of randomness, such as A/B testing and load balancers. One idea could be to test self-hosted versions of

commonly used web applications such as the ones from Bitnami [9] instead of scanning popular websites. As the source code is available for these applications, most of the above issues can be detected or mitigated. Additionally, insecure defaults in commonly used libraries could be detected and reported to the library maintainers.

5.3 Related work

This thesis builds upon a variety of works ranging from the discoveries of the used XS-Leak methods, over studies of cross-browser behavior and measurement studies on the web, to works that are studying the state of XS-Leaks in browsers or the wild. In the following, we introduce the most relevant related work and compare the findings of this thesis with previous results.

Detection techniques: The general problem of XS-Leaks has been known since the early 2000s when Felten and Schneider described access detection attacks exploiting cache behavior via a timing side-channel [31]. In 2006, Hansen reported that the error event on img-tags could be used to determine if a user is authenticated. This method works because some sites serve HTML error pages or redirect unauthenticated users trying to access protected images [42]. Later, similar leaks on other tags (e.g., script or video) and other events (e.g., onloadeddata) were discovered [41, 12, 40]. In 2012, Grossman described that it is possible to leverage the detection of properties of dynamic JavaScript files (XSSI) and CSS files, as well as the SOP-conform access on the frame.length property of IFrames for login detection [40]. In 2013, Homakov discovered that Content-Security-Policys, invented to mitigate XSS-Attacks, can be used to detect redirects which can be used to detect the OAuth status of a user [43]. Recently, all these different detection techniques and attack scenarios have been grouped under the label XS-Leaks in a wiki [88]. The creators gather information on all the different leak techniques and defenses on the continuously evolving wiki.

In this work, we have shown that most methods still work today. Additionally, we provide the first comprehensive data on the prevalence of different methods. We found that some methods work more often than expected, whereas others, such as authenticated images, rarely occur in the wild. These findings mean that the web landscape has changed, or the old estimates were invalid from the beginning as they were primarily anecdotal and not based on web scan data.

Cross-browser behavior: It is a well-known fact that different browsers behave differently. These inconsistencies bring many web developers to use cross-browser testing tools

to ensure their website works as intended on different browsers such as selenium [85]. In 2017, Schwenk et al. showed that cross-browser inconsistencies do not only concern design functionality, but also critical web security mechanisms such as the Same-Origin-Policy [84]. They developed many tests to check which DOM accesses are possible cross-origin and discovered that the behavior differed in 23% of their tests. They also found one new XS-Leak method only possible in Edge and Internet Explorer of that time.

We show that even though almost all methods work in all tested browsers, there is a significant difference between the main types of browsers. We discovered that every single leak method behaves slightly or drastically different in Chrome and Firefox. These findings align with previous work and suggest that browser vendors should enhance their cooperation to make the web a safer place.

Real world measurements: Large-scale measurements are necessary to show that XS-Leaks are a real threat to the web. Testing XS-Leaks is more complicated than many other web security issues as one must first create state information on the tested sites. Usually, this is done by registering accounts and then testing the site one time in the logged-in state and one time in the anonymous state. These registration and login efforts can either be manual or automatic.

In 2015, Lekies et al. manually created accounts on 150 top-ranked websites to search for dynamic JavaScript (XSSI) [55]. They found that user state information was leakable on 40 of the tested sites due to dynamic JavaScript. In 2019, Sanchez-Rola et al. came around the account creation and login problem by studying access detection instead [82]. They searched for server-side processing time differences depending on whether the page was visited before or not by timing requests with and without cookies attached. Out of the 10,000 websites they tested, they found around half of them vulnerable to their attack. In 2020, Drakonakis et al. solved part of the register and login problem by creating a tool that can do this automatically [28]. They managed to register and log in to 25,242 domains fully automatically. However, they used it to study a different attack, namely cookie-based account hijacking. The recent work of Jonker et al. attempted only to automate the login process and falls back to crowd-sourced user credentials for the registration step [47].

We mainly used the cookiehunter tool [28] to create state information in this thesis. Unfortunately, due to various changes on the web, the tool does not work as well as advertised anymore. Still, we managed to register and login on 412 sites with it and additionally used manually created accounts for 18 sites. We believe that better and more reliable tools are necessary for future studies of authenticated areas of websites.

State of XS-Leaks: In 2020, Sudhodanan et al. were the first to test many known XS-Leak methods in three mainstream browsers. In addition, they manually created accounts for 58 tested websites and found at least one leaky URL on all of them [92]. In 2021, Goethem et al. proposed a new taxonomy for XS-Leaks. They also tested for several methods which versions of Firefox and Chrome were vulnerable. In addition, they studied different defense mechanisms regarding which leak method they work against [39].

These works only tested a subset of the described methods in their automatic tests as they had similar problems in automating the other methods. Sudhodanan et al. tested several responses for every leak channel. However, the considered response space was small. They attempted to generalize the observations into two most general responses, A and B. Then, they introduced the concept of leak classes and presented 40 leak classes, out of which only 14 worked in all tested browsers. Many of these leak classes belong to the same leak channel and describe different paths or subtrees of the decision trees in the representation used in this thesis. The need for many leak classes relates to their incomplete generalizations. For example, not every response fits either response A or B, and it is unclear what the observation of such a response would be. In such cases, they often created several leak classes for non- or partially overlapping response sets or ignored other responses entirely. Also, they overgeneralized several properties and assumed that all 2XX, 3XX, and 4XX responses behave the same. In this thesis, we showed that different status-codes from the same group behave quite differently. This thesis makes it more transparent that usually, a method not only works in one browser or another but that the sets of responses that a leak channel can distinguish differ for each browser. This thesis also shows that a static pruning step is feasible and efficient if the generalizations are adequate. Additionally, we provide insights into the actual response patterns observed in the wild and how often different methods work. Finally, we also release the created tools making it possible for other researchers to scan sites for XS-Leaks and browser vendors to investigate edge case behavior.

Goethem et al. only used one example response pair for all of the *attacks* they tested. Several of these attacks are different response pairs for the same leak channel. For at least one attack: *server redirect (CSP violation)*, they have chosen a pair that only worked in Chrome and concluded that the attack does not work in Firefox. In this thesis, we have shown that the global-properties_securitypolicyviolation event works in Firefox and that it is essential to test a larger response space for every leak channel as the exact behavior significantly differs between browsers. These differences are often bugs in browsers. For many response pairs in the wild, these bugs are decisive for whether they are vulnerable or not. In addition, only focusing on single pairs often leads to incomplete bug fixes. To give evidence, we have discovered several bypasses of previously fixed leak channels where only a single pair regression test case existed.

5.4 Defenses and call to action

XS-Leaks are preventable. Many defenses exist, and if used correctly, it is impossible to infer user information cross-site using the techniques considered in this thesis. However, currently, many websites are vulnerable, and browser vendors should do more to prevent XS-Leak by default.

We have seen that many leaks found in chapter 4 are only possible due to inconsistently set security headers. Therefore, we strongly advise every web developer who uses security headers such as XCTO, XFO, CORP, COOP, and similar to set the same value regardless of the state of the request. We could not concretely study the effect of Fetch metadata, but servers can also use this information to prevent XS-Leaks, and we encourage them to use it. Adjusting both states to have the same outcome generally works but does not scale well and often misses some URLs. Another approach could be to add noise to responses, e.g., a randomly changing amount of invisible IFrames makes the framecount method hard or impossible to use reliably. Browser vendors could also use a noisy approach for opened documents without a reference anymore, e.g., due to COOP, instead of behaving like *about:blank* or raising a security exception.

SameSite cookies are one of the most effective and straightforward defenses against XS-Leaks. A setting of *Lax* prevents most XS-Leaks without breaking many websites. We highlight that not only the session cookies need to be protected but *all* cookies carrying any state information. In addition, websites should set the COOP header to prevent leaks based on *window.open*. We have observed many leaky URLs in Firefox that are not attackable in Chrome due to differing SameSite defaults. Therefore, we appeal to Firefox to switch to the new recommended default of *Lax* and not accepting *None* without the Secure flag as soon as possible.

In addition, we urge browser vendors and specification bodies to unify edge case behavior. If one only considers the intersection where both tested browsers work, the attack surface for XS-Leaks is heavily reduced. Luckily, unifying edge case behavior should not introduce any missing functionality as the individual quirks responsible for the URLs only vulnerable in one browser do not exist in the other browser.

Chapter 6

Conclusion

This thesis aimed to study XS-Leaks in more depth to give a better insight into the state of XS-Leaks on the web and in browsers. Furthermore, we wanted to use these insights to estimate more accurately how big of an issue XS-Leaks are for the web ecosystem.

Looking back at the first research question, "Which groups of responses can different leak methods distinguish in different browsers?", we state that most methods still leak information in major browsers. Many leaks follow directly from the specifications and cannot be easily solved without changing how the web works. However, the exact conditions of each leak method differ for every single method between Firefox and Chromium-based browsers. To provide more insight, we created decision trees for every leak channel in every browser, that illustrate in an easy-to-understand manner which groups of responses lead to which outcomes. Using the decision trees, we were able to discover several bugs in browsers and find other considerable differences between them. We have released the decision trees and an application to check which methods can distinguish two given responses to foster future research and more secure websites and browsers.

The second question, "Which XS-Leak methods work how often on websites in the wild in different browsers?" is challenging to answer due to the complex task of automatically creating state information on websites and confirming that a URL leaks information. Still, we managed to test for XS-Leaks on 352 sites and found 258 sites vulnerable in at least one browser. Some methods such as object-properties_element-height work only rarely and behave almost identically in both browsers. However, other methods such as object-properties_framecount are widespread and behave considerably different in both browsers. In total, only around half of the vulnerable endpoints affect both browsers. This significant difference is partly due to the tested browsers' different SameSite and

Fetch metadata behavior, but also due to the high variability in the handling of edge cases discovered while answering the first question.

In summary, we conclude that XS-Leaks exist on most websites and are a considerable threat for web users on a large share of the web in all browsers. However, there is hope for an XS-Leak free future. We could not find vulnerable URLs on every tested website and found evidence that modern defenses can effectively prevent XS-Leaks when deployed correctly. However, it is difficult for web developers to deploy these defenses correctly. Additionally, we discovered that a large part of the attack surface is due to edge case behavior that browser vendors can fix without changing the fundamentals of the web. Therefore, we suggest that browser vendors take additional steps to introduce more secure defaults and unify their edge case behavior to increase the security and privacy of users. For the future, we want to observe the evolution of browsers regarding XS-Leaks continuously. Regression bugs are common, and every new feature can introduce new leaks or stop leaks from working. Additionally, a better method to create state information on websites is necessary to give more accurate estimates on the state of the web ecosystem regarding issues such as XS-Leaks.

Bibliography

[1] G. Acar. *1450853 - (CVE-2020-15666) MediaError Message Property Leaks Cross-Origin Response Status*. 2018. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=1450853 (visited on 08/23/2021).

[2] G. Acar. *828265 - MediaError Message Property Leaks Cross-Origin Response Status*. 2018. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=828265 (visited on 08/23/2021).

[3] G. Acar, D. Y. Huang, F. Li, A. Narayanan, and N. Feamster. "Web-Based Attacks to Discover and Control Local IoT Devices". In: *Proceedings of the 2018 Workshop on IoT Security and Privacy*. SIGCOMM '18: ACM SIGCOMM 2018 Conference. Budapest Hungary: ACM, Aug. 7, 2018, pp. 29–35. DOI: 10.1145/3229565.3229568.

[4] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. "Towards a Formal Foundation of Web Security". In: *2010 23rd IEEE Computer Security Foundations Symposium*. 2010 23rd IEEE Computer Security Foundations Symposium. July 2010, pp. 290–304. DOI: 10.1109/CSF.2010.27.

[5] L. Anforowicz. *More CORB-Protected MIME Types - Adding Protected Types One-by-One. · Issue #860 · Whatwg/Fetch*. GitHub. 2019. URL: https://github.com/whatwg/fetch/issues/860 (visited on 08/25/2021).

[6] *Apache Tika*. URL: https://tika.apache.org/ (visited on 10/12/2021).

[7] A. Barth. *RFC6265*. Apr. 2011. URL: https://httpwg.org/specs/rfc6265.html (visited on 09/23/2021).

[8] T. J. Berners-Lee. *Information Management: A Proposal*. 1989.

[9] *Bitnami*. URL: https://bitnami.com/ (visited on 10/23/2021).

[10] R. Brandom. "Google Photos and the Unguessable URL". In: *The Verge* (June 23, 2015). URL: https://www.theverge.com/2015/6/23/8830977/google-photos-security-public-url-privacy-protected (visited on 10/16/2021).

[11] *Browser Version Market Share Worldwide*. StatCounter Global Stats. URL: https://gs.statcounter.com/browser-version-market-share (visited on 08/24/2021).

[12] M. Cardwell. *Abusing HTTP Status Codes to Expose Private Information*. Grepular. 2011. URL: https://www.grepular.com/Abusing_HTTP_Status_Codes_to_Expose_Private_Information (visited on 03/08/2021).

[13] O. Cetin, C. Ganan, M. Korczynski, and M. van Eeten. "Make Notifications Great Again: Learning How to Notify in the Age of Large-Scale Vulnerability Scanning". In: *Workshop on the Economics of Information Security (WEIS)*. 2017.

[14] *Complete List of Web Browsers*. URL: https://www.webdevelopersnotes.com/browsers-list (visited on 09/24/2021).

[15] *Content Security Policy Header*. URL: https://w3c.github.io/webappsec-csp/#csp-header (visited on 09/23/2021).

[16] *Content Security Policy Paths and Redirects*. URL: https://www.w3.org/TR/CSP11/#source-list-paths-and-redirects (visited on 08/23/2021).

[17] *Cookies Default to SameSite=Lax - Chrome Platform Status*. URL: https://www.chromestatus.com/feature/5088147346030592 (visited on 08/19/2021).

[18] A. Cortesi, M. Hils, T. Kriechbaumer, and contributors. *Mitmproxy: A Free and Open Source Interactive HTTPS Proxy*. Version 7.0.2. 2021. URL: https://mitmproxy.org/.

[19] *Cross-Origin Read Blocking (CORB)*. URL: https://chromium.googlesource.com/chromium/src/+/master/services/network/cross_origin_read_blocking_explainer.md (visited on 04/28/2021).

[20] *Cross-Origin-Embedder-Policy*. URL: https://html.spec.whatwg.org/multipage/origin.html#coep (visited on 09/23/2021).

[21] *Cross-Origin-Opener-Policy*. URL: https://html.spec.whatwg.org/multipage/origin.html#cross-origin-opener-policies (visited on 09/23/2021).

[22] DataReportal. *Digital 2021: Global Overview Report*. DataReportal – Global Digital Insights. 2021. URL: https://datareportal.com/reports/digital-2021-global-overview-report (visited on 09/28/2021).

[23] *Desktop Browser Market Share Worldwide July 2021*. StatCounter Global Stats. URL: https://gs.statcounter.com/browser-market-share/desktop/worldwide/ (visited on 09/24/2021).

[24] *Desktop Browser Version Market Share Worldwide*. StatCounter Global Stats. URL: https://gs.statcounter.com/browser-version-market-share/desktop/worldwide/ (visited on 09/24/2021).

[25] *Distributed Random Forest (DRF)*. URL: https://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/drf.html (visited on 08/23/2021).

[26] *Django*. Version 3.2.4. 2021. URL: https://www.djangoproject.com/ (visited on 08/23/2021).

[27] *Docker*. URL: https://www.docker.com/ (visited on 09/24/2021).

[28] K. Drakonakis, S. Ioannidis, and J. Polakis. "The Cookie Hunter: Automated Black-Box Auditing for Web Authentication and Authorization Flaws". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security. Virtual Event USA: ACM, Oct. 30, 2020, pp. 1953–1970. DOI: 10.1145/3372297.3417869.

[29] C. Dresen, F. Ising, D. Poddebniak, T. Kappert, T. Holz, and S. Schinzel. "CORSICA: Cross-Origin Web Service Identification". In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. ASIA CCS '20. New York, NY, USA: Association for Computing Machinery, Oct. 5, 2020, pp. 409–419. DOI: 10.1145/3320269.3372196.

[30] T. Duebendorfer and S. Frei. "Why Silent Updates Boost Security". In: *TIK, ETH Zurich, Tech. Rep* 302 (2009), p. 98.

[31] E. W. Felten and M. A. Schneider. "Timing Attacks on Web Privacy". In: *Proceedings of the 7th ACM Conference on Computer and Communications Security*. CCS '00. New York, NY, USA: Association for Computing Machinery, Nov. 1, 2000, pp. 25–32. DOI: 10.1145/352600.352606.

[32] *Fetch Metadata Request Headers*. URL: https://w3c.github.io/webappsec-fetch-metadata/#sec-fetch-site-header (visited on 08/25/2021).

[33] *Fetch Standard CORP*. URL: https://fetch.spec.whatwg.org/#cross-origin-resource-policy-header (visited on 08/25/2021).

[34] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. *RFC2068: Hypertext Transfer Protocol–HTTP/1.1*. RFC Editor, 1997.

[35] R. Fielding and J. Reschke. *RFC7231*. June 2014. URL: https://httpwg.org/specs/rfc7231.html (visited on 10/17/2021).

[36] *File*. URL: https://man7.org/linux/man-pages/man1/file.1.html (visited on 10/12/2021).

[37] P. J. Franks, P. Hallam-Baker, L. C. Stewart, J. L. Hostetler, S. Lawrence, P. J. Leach, and A. Luotonen. *HTTP Authentication: Basic and Digest Access Authentication*. Request for Comments RFC 2617. Internet Engineering Task Force, June 1999. 34 pp. DOI: 10.17487/RFC2617.

[38] N. Gelernter and A. Herzberg. "Cross-Site Search Attacks". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS'15: The 22nd ACM Conference on Computer and Communications Security. Denver Colorado USA: ACM, Oct. 12, 2015, pp. 1394–1405. DOI: 10.1145/2810103.2813688.

[39] T. V. Goethem, G. Franken, I. Sanchez-Rola, D. Dworken, and W. Joosen. "Understanding Cross-Site Leaks and Defenses". In: *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 2021, p. 16. URL: https://secweb.work/papers/2021/vangoethem2021leaks.pdf.

[40] J. Grossman. *I Know What Websites You Are Logged-In To (Login-Detection via CSRF)*. WhiteHat Security. 2012. URL: https://web.archive.org/web/20160317054027/https://www.whitehatsec.com/blog/i-know-what-websites-you-are-logged-in-to-login-detection-via-csrf/ (visited on 03/08/2021).

[41] J. Grossman. *Login Detection, Whose Problem Is It?* 2008. URL: https://blog.jeremiahgrossman.com/2008/03/login-detection-whose-problem-is-it.html (visited on 03/08/2021).

[42] R. Hansen. *Detecting States of Authentication With Protected Images*. ha.ckers. 2006. URL: https://web.archive.org/web/20150417095319/http://ha.ckers.org/blog/20061108/detecting-states-of-authentication-with-protected-images/ (visited on 03/08/2021).

[43] E. Homakov. *313737 - Disclose Domain of Redirect Destination Taking Adventadge of CSP*. 2013. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=313737 (visited on 03/08/2021).

[44] E. Homakov. *Using Content-Security-Policy for Evil*. Jan. 13, 2014. URL: https://homakov.blogspot.com/2014/01/using-content-security-policy-for-evil.html (visited on 03/09/2021).

[45] *HTML Standard*. URL: https://html.spec.whatwg.org/multipage/ (visited on 09/23/2021).

[46] *Hypertext Transfer Protocol (HTTP) Status Code Registry*. URL: https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml (visited on 08/24/2021).

[47] H. Jonker, S. Karsch, B. Krumnow, and M. Sleegers. "Shepherd: A Generic Approach to Automating Website Login". In: *Proceedings 2020 Workshop on Measurements, Attacks, and Defenses for the Web*. Workshop on Measurements, Attacks, and Defenses for the Web. San Diego, CA: Internet Society, 2020. DOI: 10.14722/madweb.2020.23008.

[48] C. Kerschbaumer. *Mitigating MIME Confusion Attacks in Firefox*. Mozilla Security Blog. 2016. URL: https://blog.mozilla.org/security/2016/08/26/mitigating-mime-confusion-attacks-in-firefox (visited on 08/19/2021).

[49] E. Kitamura. *Gaining Security and Privacy by Partitioning the Cache*. 2020. URL: https://developers.google.com/web/updates/2020/10/http-cache-partitioning (visited on 08/19/2021).

[50] S. Kobes. *712246 - Security: CSS :Visited with Mix-Blend-Mode Can Leak Browser History*. 2017. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=712246 (visited on 08/20/2021).

[51] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019 IEEE Symposium on Security and Privacy (SP). May 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[52] J. Kokatsu. *835465 - X-Frame-Options and CSP Frame-Ancestors Is Ignored When Location Header Is Present*. 2018. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=835465 (visited on 08/23/2021).

[53] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczynski, and W. Joosen. "Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation". In: *Proceedings 2019 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2019. DOI: 10.14722/ndss.2019.23386.

[54] S. Lee, H. Kim, and J. Kim. "Identifying Cross-Origin Resource Status Using Application Cache". In: *Proceedings 2015 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2015. DOI: 10.14722/ndss.2015.23027.

[55] S. Lekies, B. Stock, M. Wentzel, and M. Johns. "The Unexpected Dangers of Dynamic JavaScript". In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 723–735. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lekies.

[56] R. Masas. *Patched Facebook Vulnerability Could Have Exposed Private Information About You and Your Friends*. Imperva. 2018. URL: https://www.imperva.com/blog/facebook-privacy-bug/ (visited on 03/08/2021).

[57] R. Masas. *The Human Side Channel*. 2021. URL: https://ronmasas.com/posts/the-human-side-channel (visited on 08/20/2021).

[58] R. Merewood. *SameSite Cookies Explained*. web.dev. 2019. URL: https://web.dev/samesite-cookies-explained/ (visited on 10/14/2021).

[59] S. Morgan. *Cybercrime To Cost The World $10.5 Trillion Annually By 2025*. Cybercrime Magazine. Nov. 10, 2020. URL: https://cybersecurityventures.com/cybercrime-damage-costs-10-trillion-by-2025/ (visited on 10/18/2021).

[60] Mozilla. *Cross-Site Request Forgery (CSRF)*. URL: https://developer.mozilla.org/en-US/docs/Web/Security/Types_of_attacks#cross-site_request_forgery_csrf (visited on 09/23/2021).

[61] Mozilla. *Cross-Site Scripting (XSS)*. URL: https://developer.mozilla.org/en-US/docs/Web/Security/Types_of_attacks#cross-site_scripting_xss (visited on 09/23/2021).

[62] Mozilla. *Mozilla Firefox Release Notes*. 2021. URL: https://www.mozilla.org/en-US/firefox/releases/ (visited on 09/24/2021).

[63] Mozilla. *Same-Origin Policy*. URL: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy (visited on 08/19/2021).

[64] Mozilla. *State Partitioning*. 2021. URL: https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Privacy/State_Partitioning (visited on 04/28/2021).

[65] Mozilla. *Window: Load Event*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Window/load_event (visited on 08/24/2021).

[66] J. Müller. *Inconsistency between Headless and Headful Browser · Issue #3940 · Puppeteer/Puppeteer*. GitHub. 2019. URL: https://github.com/puppeteer/puppeteer/issues/3940 (visited on 08/25/2021).

[67] *Origin Header*. URL: https://fetch.spec.whatwg.org/#origin-header (visited on 10/17/2021).

[68] *Path Traversal*. URL: https://owasp.org/www-community/attacks/Path_Traversal (visited on 10/18/2021).

[69] *Puppeteer*. Version 10.2.0. 2021. URL: https://pptr.dev/ (visited on 08/19/2021).

[70] J. Rautenstrauch. *1251534 - Security: CSP Matching Algorithm Does Not Ignore Paths for Client-Side Redirections*. 2021. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=1251534 (visited on 09/24/2021).

[71] J. Rautenstrauch. *1251921 - Security: MediaError Messages Still Leak Cross-Origin Informatio*. 2021. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=1251921 (visited on 09/24/2021).

[72] J. Rautenstrauch. *1260366 - Security: X-Frame-Options and CSP: Frame-Ancestor Information Leaks Cross-Origin Using Object Tag*. 2021. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=1260366#makechanges (visited on 10/15/2021).

[73] J. Rautenstrauch. *1731614 - MediaError Message Property Leaks Information on Cross-Origin Same-Site Pages*. 2021. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=1731614 (visited on 09/24/2021).

[74] J. Rautenstrauch. *1732012 - X-Frame-Options Is Ignored on Redirection Status-Codes (without a Location Set)*. 2021. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=1732012 (visited on 09/24/2021).

[75] J. Rautenstrauch. *1732069 - Sec-Fetch-Site Inconsistent on Localhost/IPs*. 2021. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=1732069 (visited on 09/24/2021).

[76] J. Rautenstrauch. *1732106 - Cross-Origin-Resource-Policy Incorrectly Applied on Object and Embed Tags.* 2021. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=1732106 (visited on 09/24/2021).

[77] J. Rautenstrauch. *1732141 - Request Loads Forever If Code Is 101 or 304 and Ct=application/Pdf*. 2021. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=1732141 (visited on 09/24/2021).

[78] J. Rautenstrauch. *1732199 - Infinite Reload of 201, 203, 204 Responses*. 2021. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=1732199 (visited on 09/24/2021).

[79] J. Rautenstrauch. *1735856 - Securitypolicyviolation Leaks Cross-Origin Information for Frame-Ancestors Violations*. 2021. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=1735856 (visited on 10/15/2021).

[80] D. Ross and T. Gondrom. *HTTP Header Field X-Frame-Options*. Request for Comments RFC 7034. Internet Engineering Task Force, Oct. 2013. 14 pp. DOI: 10.17487/RFC7034.

[81] S. Roth, T. Barron, S. Calzavara, N. Nikiforakis, and B. Stock. "Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies". In: *Proceedings 2020 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2020. DOI: 10.14722/ndss.2020.23046.

[82] I. Sanchez-Rola, D. Balzarotti, and I. Santos. "BakingTimer: Privacy Analysis of Server-Side Request Processing Time". In: *Proceedings of the 35th Annual Computer Security Applications Conference*. ACSAC '19: 2019 Annual Computer Security Applications Conference. San Juan Puerto Rico USA: ACM, Dec. 9, 2019, pp. 478–488. DOI: 10.1145/3359789.3359803.

[83] A. Sartori. *1186611 - Securitypolicyviolation Event Leaks Cross-Origin Information for Frame-Ancestors Violations*. 2021. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=1186611&q=frame-ancestors&can=1 (visited on 10/15/2021).

[84] J. Schwenk, M. Niemietz, and C. Mainka. "Same-Origin Policy: Evaluation in Modern Browsers". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 713–727. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schwenk.

[85] *Selenium*. Version 4. 2021. URL: https://www.selenium.dev/ (visited on 08/19/2021).

[86] *Selenium Grid*. Version 4. 2021. URL: https://www.selenium.dev/documentation/en/grid/ (visited on 04/15/2021).

[87] *Site Isolation*. URL: https://www.chromium.org/Home/chromium-security/site-isolation (visited on 10/17/2021).

[88] M. Sousa, terjanq, R. Clapis, D. Dworken, NDevTK, 1lastBr3ath, Brasco, rick.titor, C. Fredrickson, and jub0bs. *XS-Leaks Wiki*. 2020. URL: https://xsleaks.dev/ (visited on 03/08/2021).

[89] *SQL Injection*. URL: https://owasp.org/www-community/attacks/SQL_Injection (visited on 10/18/2021).

[90] C.-A. Staicu and M. Pradel. "Leaky Images: Targeted Privacy Attacks in the Web". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 923–939. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/staicu.

[91] B. Stock, G. Pellegrino, F. Li, M. Backes, and C. Rossow. "Didn't You Hear Me? - Towards More Successful Web Vulnerability Notifications". In: *Proceedings 2018 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2018. DOI: 10.14722/ndss.2018.23171.

[92] A. Sudhodanan, S. Khodayari, and J. Caballero. "Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks". In: *Proceedings 2020 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2020. DOI: 10.14722/ndss.2020.24278.

[93] L. Suto. "Analyzing the Accuracy and Time Costs of Web Application Security Scanners". In: *San Francisco, February* (2010).

[94] terjanq. *Mass XS-Search Using Cache Attack*. 2019. URL: https://terjanq.github.io/Bug-Bounty/Google/cache-attack-06jd2d2mz2r0/index.html (visited on 10/01/2021).

[95] *The Tor Project*. URL: https://torproject.org (visited on 09/24/2021).

[96] A. Tolfsen. *CONTROL Key Chords Not Working to Open New Tabs and Windows · Issue #786 · Mozilla/Geckodriver*. GitHub. 2017. URL: https://github.com/mozilla/geckodriver/issues/786#issuecomment-321046966 (visited on 08/23/2021).

[97] *uWSGI*. Version 2.0.19. 2020. URL: https://uwsgi-docs.readthedocs.io/en/latest/ (visited on 08/23/2021).

[98] T. Van Goethem, W. Joosen, and N. Nikiforakis. "The Clock Is Still Ticking: Timing Attacks in the Modern Web". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS'15: The 22nd ACM Conference on Computer and Communications Security. Denver Colorado USA: ACM, Oct. 12, 2015, pp. 1382–1393. DOI: 10.1145/2810103.2813632.

[99] A. van Kesteren. *1595491 - Make <embed> and <object> Behave More like <iframe>*. 2019. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=1595491 (visited on 09/24/2021).

[100] *WebDriver Standard*. 2021. URL: https://www.w3.org/TR/webdriver/#security (visited on 08/25/2021).

[101] *What Is a VPN? - Virtual Private Network*. Cisco. URL: https://www.cisco.com/c/en/us/products/security/vpn-endpoint-security-clients/what-is-vpn.html (visited on 09/24/2021).

[102] WHATWG. *Fetch Standard CORB*. URL: https://fetch.spec.whatwg.org/#corb (visited on 08/19/2021).

[103] *X-Content-Type Options*. X-Content-Tpye-Options. URL: https://fetch.spec.whatwg.org/#x-content-type-options-header (visited on 09/23/2021).

[104] Y. Zhou and D. Evans. "SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities". In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 495–510. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/zhou.

Appendix A

Inclusion methods and leak methods

Inclusion methods Table A.1 lists all inclusion methods considered in this thesis with example inclusion code. We omitted the event callbacks for clarity.

Inclusion method	Inclusion code
script	<script id='test' src=URL></script>
link-stylesheet	<link id='test' href=URL rel=stylesheet></link>
link-prefetch	<link id='test' href=URL rel=prefetch></link>
img	<img id='test' src=URL></img>
iframe	<iframe id='test' src=URL></iframe>
video	<video id='test' src=URL></video>
audio	<audio id='test' src=URL></audio>
object	<object id='test' data=URL></object>
embed	<embed id='test' src=URL></embed>
embed-img	<embed id='test' src=URL type=image/jpg></embed>
window.open	win = window.open(URL)
iframe-csp	<meta http-equiv="Content-Security-Policy" content="default-src 'self' 'unsafe-inline' URL"> <iframe id='test' src=URL></iframe>

Table A.1: List of inclusion methods considered in this thesis.

Leak methods The leak methods either work by observing the events fired (EF), by accessing a property on a tag element *el* or a window element *win* (OP), or by defining a callback function on the global window object or reading a global property (GP). The following code shows how to get access to the tag element *el*, the window element *win*, which events we considered and what *css_test* is.

```
var el = document.getElementById("test");
var win = el.contentWindow; // or win = window.open(URL)
```

```
var events = ["error", "load", "loadedmetadata", "stalled", "suspend"];
// element that the stylesheet applies to
var css_test = document.getElementById("css_test");
```

Table A.2 lists all leak methods tested in this thesis together with a summary of how each method works. The methods defining functions observe that the function was called together with the calling parameters. Many of the methods accessing properties on *win* test whether the element is accessible (often *null*) or if the access throws a security error. The other methods record the value of the property.

Leak method	Leak description
event_list	list of events fired
event_set	set of events fired
load_count	number of load events fired
op_frame_count	win.length
op_win_window	win.window
op_win_CSS2Properties	win.CSS2Properties
op_win_origin	win.origin
op_win_opener	win.opener
op_win_history_length	win.location.replace('about:blank'); win.history.length
op_el_height	el.height
op_el_width	el.width
op_el_naturalHeight	el.naturalHeight
op_el_naturalWidth	el.naturalWidth
op_el_videoWidth	el.videoWidth
op_el_videoHeight	el.videoHeight
op_el_duration	el.duration
op_el_networkState	el.networkState
op_el_readyState	el.readyState
op_el_buffered	el.buffered
op_el_paused	el.paused
op_el_seekable	el.seekable
op_el_sheet	el.sheet
op_el_media_error	el.error
op_el_contentDocument	el.contentDocument
gp_window_onerror	window.onerror = function()
gp_window_onblur	window.onblur = function()
gp_window_postMessage	window.addEventListener('message', funcction())
gp_window_getComputedStyle	window.getComputedStyle(css_test).getPropertyValue('color')
gp_window_hasOwnProperty	window.hasOwnProperty('a')

gp_download_bar_height_bin	start = window.innerHeight;
	load();
	bar = start - window.innerHeight

gp_securitypolicyviolation	window.addEventListener('securitypolicyviolation', function())

Table A.2: List of leak methods considered in this thesis.

Appendix B

Online materials and browser settings

Online materials All code used and created for this thesis can be accessed online. In addition to the code, the material contains all created decision trees. The code is distributed over the following three git repositories:

- `https://github.com/JannisBush/xs-leaks-browser-web`
  - Main repository
  - All code for R1 and the test browser framework.
  - Most parts for R2/does-it-leak pipeline except for the stateful crawler and the cookiehunter option of the state generator. The code in this repository can be used standalone to test own websites.
  - All analysis code and output including all created decision trees and tables used in this thesis.
- `https://projects.cispa.saarland/c01jara/cookiehunter`
  - State generator
  - Modified version of cookiehunter for registration and login.
- `https://projects.cispa.saarland/c01jara/node-crawler`
  - Stateful crawler
  - Modified version of node-crawler to crawl websites with cookies.

Browser setting details

- Settings and versions for R1:
  - Selenium Grid image used: `https://github.com/SeleniumHQ/docker-selenium/releases/tag/4.0.0-beta-3-20210426`
  - Browser versions: Chrome 90.0.4430.85, Edge 91.0.864.1, Firefox: 88.0
  - Selenium driver: 4.0.0b3, Python version: 3.9.2
  - Timeouts: page load 1s, page execute 2s (for window.open retest: 4s, additional delay time after load 150ms, (for window.open and iframe 1.5 * 150ms), maximum number of URLs before browser restart: 500 (for window.open retest: 100)
- Settings and versions for R2:
  - State generator (cookiehunter): Chrome 92.0.4515.131 headfull with xvfb, Tranco date: 2021-08-05, maximum time per try: 1800s, maximum tries per site: 2.
  - Stateful crawler (node-crawler): Chromium 92.0.4512.0 headless with stealth plugin, maximum depth of crawl: 3, maximum URLs per site: 100, maximum URLs before browser restart: 10, maximum number of retries per URL: 2, page load timeout: 20s, execution time timeout: 5s.
  - Does-it-leak pipeline dynamic confirmator: browsers: same as R1, page load timeout: 20s, execution time timeout: 10s, additional delay time: 2s, maximum number of URLs before browser restart: 100, maximum number of retries per URL: 2.