The State of the SameSite: Studying the Usage, Effectiveness, and Adequacy of SameSite Cookies

Soheil Khodayari, Giancarlo Pellegrino
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
soheil.khodayari@cispa.saarland, pellegrino@cispa.de

***Abstract*—Chromium-based browsers now restrict cookies' scope to a same-site context by changing the default policy for cookies, thus requiring developers to adapt their websites. The extent of the adoption and effectiveness of the SameSite policy has not been studied yet, and, in this paper, we undertake one of the first evaluations of the state of the SameSite cookie policy. We conducted a set of large-scale, longitudinal, both automated and manual measurements of the Alexa top 1K, 10K, 100K, and 500K sites across the main rollout dates of the SameSite policies, covering both SameSite usage and cross-site functionality breakage caused by the new default policy. Also, we performed an extensive evaluation of threats against the new Lax-by-default policy's effectiveness, looking at the adequacy of the coverage provided by the Lax policy and bypass caused by website developers' mistakes.**

Our study shows that the growth of sites using a SameSite policy has slowed down considerably after the enforcement dates. Then, the new Lax-by-default policy has affected about 19% of the functionalities implemented via cross-site requests without an explicit SameSite policy, most of which are for online ads. Third, our study observes a significant mismatch between the request contexts covered by Lax and the ones actually used by websites in the wild, making it possible to perform XS attacks also against popular websites such as Tumblr, Twitch, SoundCloud, Mailchimp, and Pixiv. Even when using Lax or Strict policies, much of their effectiveness depends on developers' awareness of SameSite policies' implications, who could introduce vulnerabilities or inconsistent policies, leading to SameSite policy bypasses. For example, we identified bypass in IMDB, Paypal, and Meetup. Also, we discovered a widespread SSO IdP abuse that attackers could use to attack target websites even when using stricter SameSite policies. Finally, in this paper, we also look at SameSite implementations in popular browsers and the default configuration in web frameworks.

***Index Terms*—SameSite Cookies, Cross-Site Request Forgery, Cross-Site Information Leakage**

I. Introduction

Limiting the scope of cookies to first-party context is a long known countermeasure [1] to protect web applications from cross-site attacks (XS attacks), such as cross-site information leakage (XS-Leaks) [2–5] or cross-site request forgery (CSRF) [6–9], by stripping authentication cookies from cross-site requests the user nor the web application intended to initiate. Existing solutions require installing additional components such as HTTP proxies [1] or browser extensions [10], limiting their impact considerably. However, only very recently, Google revamped the idea of same-site policies for cookies by proposing and implementing in Chrome a new cookie attribute [11], the `SameSite` attribute. The `SameSite` attribute introduces three pre-defined same-site policies (None, Lax, and Strict)—one of which is the new default policy—each defining a set of cross-site requests contexts where the browser will not include cookies. By switching to a same-site policy by default, the hope is that XS attacks become old news [2, 12–17].

The radical change introduced by the SameSite attribute is that browsers no longer include cookies in all cross-site requests by default. As such a change can disrupt existing websites and to help developers transition to the new policy, Google rolled out `SameSite`'s features, spreading them over a period of four years, starting from April 2016, where it introduced the support for explicitly-defined SameSite policies, till July 2020 with the enforcement of the new default policy. As the new policies will play a major role to the security of the web platform, in this paper, we take a closer look at the status of `SameSite` attribute before and after the enforcement of the new default SameSite policy. In this paper, we conduct, to the best of our knowledge, the first evaluation of the SameSite cookie policy, systematically covering the trend of its usage, the impact of its new default, and the threats against its effectiveness. We collect and examine the security risks as a result of the way developers are adapting to the new changes, and systematize the threats that can undermine the Lax protection, with the overarching purpose of studying the adequacy of the new default SameSite cookie policy.

Adoption and Breakage. Starting with a longitudinal analysis of the SameSite cookie usage from June 2019 to March 2021 on the top 500K sites, we show that, even with a four-year rollout plan, only 18.94% of sites adopted one of the three policies, with a steep increase of +203.54% and +18.95% at the two dates of the new policy enforcement, respectively. Interestingly, 3.7% of the sites disable the SameSite protection using the None policy, with a significant increase towards more popular websites, i.e., 18% for the top 1K sites. Then, as the new Lax-by-default policy can break functionalities, in this paper, we provide one of the first systematic analyses to identify and quantify functionality breakage in the wild due to the Lax policy. Our results show that, after the rollout of the new policy, about 19% of functionalities implemented via cross-site requests no longer work, most of which (77.5%) are for online advertisement.

Threats to Effectiveness. In this paper, we also take a look

at the adequacy of the Lax policy to protect existing websites. In particular, we explore the tension between the protected contexts and the contexts that are used by existing websites to implement security-sensitive operations. In our evaluation, we first review academic and non-academic literature, and identify ten distinct threats, including three new threats inspired by prior work [18, 19]. Then, we assess their prevalence and practicality in the wild, showing a rather concerning scenario.

For example, we showed that 10.3% of state-changing requests of the top 1K sites (i.e., 721 out of 6,951) are still implemented via GET requests, which are not protected by the Lax policy, and in 2.6% of them, we successfully verified that CSRF attacks are possible (including popular sites like Mailchimp and Pixiv). Then, we discovered 1,302 distinct information leakage vulnerabilities (XS-Leaks) that leak the user's login status or identity leveraging window properties and postMessage, via requests that are not protected by the Lax policy. These XS-Leak attacks affect 40 websites of Alexa top 500, including popular ones such as Tumblr, Twitch, and SoundCloud. When looking at the requests that are protected by the policy, we identified known and new vulnerabilities in web applications hampering the SameSite cookies effectiveness. For example, we discovered that 1.5% of the sensitive state-changing POST requests can be exploited for CSRF attacks by switching the method to GET. Among the vulnerable sites we found IMDB, PayPal, and Meetup. Also, we discovered a wide-spread behavior of Single Sign-On identity providers that can be used to abuse the exceptional SameSite policy used by Chrome (i.e., Lax+POST) to refresh session cookies and perform XS attacks within two minutes of the cookie refresh. The affected IdPs, among which we have Google, Facebook and Linkedin, are used by 49% of the top 10K Alexa sites. Finally, we observed three different incorrect and inconsistent use of the `SameSite` attribute, that can be exploited by attackers in XS attacks, i.e., different policies between mobile and desktop, cookies with different policies across web pages, and duplicated cookies with different `SameSite` attributes.

Browsers and Web Frameworks. Finally, in this paper, we also conduct a comprehensive analysis of 14 popular web browsers (both mobile and desktop) with regards to the SameSite cookies, and observed that none of them fully complies with the RFC 6265bis specification [20], exposing a total of seven divergent behaviours when enforcing the SameSite cookie policy. Even if browsers offer a Lax-by-default SameSite cookie protection, we show that 24% of the top five web frameworks of top five programming languages can downgrade that protection by default when the developer sets a cookie via one of the frameworks' offered APIs (e.g., in Django [21] or Pyramid [22]).

Insights. Overall, our study provides the following insights about the current state of the `SameSite` attribute. As expected, after a rapid increment of sites using one of the SameSite policies around the enforcement dates, we now observe a rather moderate, yet steady, growth. Second, about 19% of the functionalities implemented via cross-site requests without an explicit SameSite policy does not work after the Lax-by-default enforcement, of which the vast majority are requests for advertisement online services. Third, uncustomizable, pre-packaged policies like the SameSite policy are making it particularly challenging for a significant fraction of websites to benefit from their protections without substantially revisiting websites' designs and implementations. For example, while the Lax policy can considerably reduce the attack surface of cross-site exploitations, we observed a significant mismatch between the cross-site request contexts covered by Lax and the ones used by websites in the wild, making it possible to perform XS attacks. Such a mismatch may suggest that a user-customizable, per-contexts same-site policy could be more beneficial for these websites. Forth, even when using Lax or Strict policies, much of their effectiveness depends on developers who may introduce inconsistent or conflicting policies, leading to SameSite policy bypasses. Finally, we observed that popular mobile and desktop browsers exhibit inconsistent behaviors when processing and enforcing SameSite policies and handling exceptional cases.

Contributions. To summarize, this paper makes the following contributions:

- We conduct, to the best of our knowledge, the first security evaluation of the SameSite cookie policy, systematically studying the trend of its usage from June 2019 to March 2021 on the top 500K Alexa sites.
- We study the impact of the new default SameSite cookie policy, and present an overview of the number of affected services and websites by systematically analyzing top 500 Alexa websites.
- We comprehensively review the threats against SameSite cookies, and identify seven known and propose three new threats inspired by prior work. We quantify the impact and prevalence of vulnerabilities of each threat in the wild by designing large-scale experiments.
- We conduct a comprehensive analysis of 14 popular web browsers, identifying seven divergent ways on how browsers enforce the SameSite cookie policy. We analyze top five web frameworks of top five programming languages, and show that 24% of the frameworks offer APIs that, by default, downgrade the new default Lax protection offered by browsers.

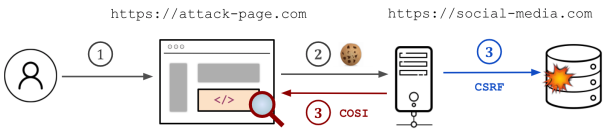
II. Background

Before presenting our research questions, we briefly introduce the building blocks of this paper. §II-A provides an overview of the family cross-site attacks, and then §II-B shows how same-site policies can help to mitigate them.

A. Cross-Site (XS) Attacks

Cross-site attacks are a family of web attacks where attackers lure users into visiting a malicious web page that tricks the user's web browser to send authenticated cross-site HTTP requests to a vulnerable target website. One of the first instances of cross-site attacks is Cross-Site Request Forgery

Fig. 1: Example of a COSI and CSRF attack.



(CSRF) [6–9, 23, 24], where attackers leverage cross-site requests to perform security-sensitive, server-side state-changing operations, such as user credential reset (see, e.g., [25–27]) or money transfers [28], without user's consent or awareness. Malicious cross-site requests can also target the users making these requests by leaking sensitive information about user's login status [29–31], account type [2], age range [32], or user's identity (i.e., deanonymization attack) [2, 5]. These attacks are often called cross-site information leakage (XS-Leaks) or Cross-Origin State Inference (COSI) attacks [2–5, 32–36].

CSRF and COSI attacks have a similar two-phases attack pattern: *preparation* and *attack*. Figure 1 exemplifies the threat model of these attacks. In the preparation step, the attacker prepares a malicious webpage referring to resources from the target site. These can be, for example, a hidden, self-submitting HTML form to reset the user password at the target site or a JavaScript file hosted by the target site. During the attack phase, the user visits the attack page (step 1). As a result of the included resource, the user's browser sends a cross-site request to the target website, and the browser automatically attaches the user's authenticated session cookies to this request (step 2). The target website receives and processes the request (step 3). In the case of CSRF, the server will perform the requested operation, e.g., by resetting the user password with an attacker-controlled one. In case of a COSI attack, the attack page uses browser side-channels to leak sensitive information about the user. For example, consider a cross-origin HTTP request that returns a 200 response code when the user is logged in and a 404 otherwise.

B. SameSite Cookie Policy

Same-Site Policies. A distinctive feature of XS attacks is that browsers include existing valid cookies in all outgoing requests, regardless of the context of the requests (e.g., user click on anchor tags or asynchronous HTTP fetch operation) or the origin of the page performing the requests (e.g., same-site or cross-site requests). An effective solution to XS attacks is limiting the scope of cookies by defining the context in which browsers can include cookies.

Limiting the scope of cookies to prevent XS attacks is not a new idea. For example, in 2006, Johns et al. proposed Request Rodeo [1], an HTTP proxy for browsers that can detect and remove cookies from cross-site requests that the user did not initiate when interacting with the webpage. More recently, in 2016, Google revamped a similar idea by introducing in Chrome a new cookie attribute [11], the `SameSite` attribute for the `Set-Cookie` HTTP response header (see RFC 6265bis [20]), allowing developers to pick one out of three pre-defined cookie policies, namely, the None, Lax, and Strict policies [20, 37].

None	Lax	Strict	Context		Example
✓	✓	-	Anchor	GET	`<a href=u>`
✓	✓	-	Form	GET	`<form method=GET action=u>`
✓	✓	-	Link prerender	GET	`<link rel=prerender href=u/>`
✓	✓	-	Link prefetch	GET	`<link rel=prefetch href=u/>`
✓	✓	-	win.open()	GET	`window.open(u)`
✓	✓	-	win.location	GET	`window.location.href=u`
✓	✓(*)	-	Form	POST	`<form method=POST action=u>`
✓	-	-	Iframe	GET	`<iframe src=u>`
✓	-	-	Object	GET	`<object data=u>`
✓	-	-	Embed	GET	`<embed src=u>`
✓	-	-	Image	GET	`<img src=u>`
✓	-	-	Script	GET	`<script src=u>`
✓	-	-	StyleSheet	GET	`<link rel=stylesheet href=u>`
✓	✓(*)	-	Async Reqs.	Any	`xmlhttp.open("POST", u)`

TABLE I: Contexts where the three `SameSite` policies apply. We use (*) for the Lax+POST exceptional policy, and ✓to show contexts where cookies are included in the cross-site HTTP request.

SameSite Cookies. The `SameSite` attribute introduces three pre-defined cookie policies. The None policy specifies that cookies are attached to *all* outgoing requests, including cross-site ones. This policy corresponds to the default policy before the introduction of the SameSite attribute. At the other end of the spectrum, we have the Strict policy that stipulates that cookies are restricted to the same-site only, i.e., cookies are never attached to any cross-site request. Finally, the Lax policy is the new *default* policy for cookies, and it defines the contexts where browsers can include cookies for cross-site requests. For example, browsers include cookies to same-site requests and top-level navigation requests (e.g., clicking on an anchor link) with safe HTTP methods [38]. However, browsers will not include cookies to cross-site requests with unsafe HTTP methods. Table I summarizes the context where the same-site policies apply.

New Default Policy and Exception. Starting from July 2020, Google Chrome set the new default policy to Lax [39], meaning that when the `SameSite` attribute is missing, the browser will enforce the Lax policy. Other browser vendors adopted [40, 41] or are planning to adopt [42] the same new default policy.

Unfortunately, enforcing a default Lax policy may break web application functionalities that rely on cross-site communications and cookies. For example, web-based Single Sign-On (SSO) implementations, such as OpenIDConnect or SAML SSO implementations, optimize user experience by avoiding user re-authentication when valid authenticated session cookies are included in the cross-site requests. Such requests are often implemented via POST asynchronous requests or HTML POST forms. The new default Lax policy forbids browsers from sending cookies with these requests, breaking these functionalities. To counter that, Chrome introduced an exception to the Lax policy—called `Lax+POST`—where, for all cookies without the `SameSite` attribute, Chrome applies the None policy only for the first two minutes after the cookie is set. Then, Chrome switches to the Lax policy. Table I lists the contexts covered by SameSite cookies.

III. Problem Statement

Despite the recent surge of attention toward SameSite cookies, little has been done to understand how they are used by application developers (see, e.g., [43]) and the hurdles when using them in practice together with different web application functionalities. This paper takes the first step in this direction and explores the security and effectiveness of SameSite cookies by quantifying their usage in the wild, and also by highlighting known and introducing new web attacks that can circumvent SameSite cookies, and ultimately compromise web applications' security. In this paper, we aim to answer the following questions:

(RQ1) Trend Analysis of SameSite Cookie Usage. The main benefit of the SameSite attribute is the new default cookie policy, which can disrupt existing websites. To help developers transition to the new default policy, Google introduced three gradual changes to Chrome, introduced in 2016, 2019, and 2020. First, in April 2016, Chrome 51 introduced support for the new attribute without modifying the default policy. Later, in September 2019, Chrome 77 started showing console warning messages for cookies without the SameSite attribute. The final step of this transition took place in 2020, with Chrome 80. First, in February 2020, Chrome set Lax+POST the new default policy. However, shortly after, Google rolled it back (April 2020) to ease developers' transition to the new policy in light of the COVID-19 pandemic. The Lax-by-default was then restored in July 2020 with Chrome 84 [39, 43].

When looking at the SameSite rollout timeline, one of the first questions we intend to address is understanding the long rollout approach's effectiveness by quantitatively measuring how website developers adapted to the upcoming new policies across the main rollout milestones. In particular, we intend to quantify the websites that picked one of the three pre-defined SameSite policies and those that rely on the new default behavior. Also, as supporting the SameSite attribute requires modifying the server-side component's code, developers may make mistakes and use non-existing policies (see, i.e., [44]), inadvertently resulting in deploying a different policy than the intended one. We answer these questions in §IV.

(RQ2) Functionality Breakage. Starting from July 2020, the new default policy for cookies without the `SameSite` attribute is Lax [39, 43]. Changing the default policy for cookies can interfere with cross-site communications between web services. For example, services such as Single Sign-On (see, i.e., [45–48]) rely on asynchronous authenticated requests to exchange authentication tokens. Another example is Oracle APEX, a web application that can run inside iframes (see, i.e., [49]). According to the Lax policy, browsers will not include cookies in requests originated in iframes, preventing Oracle APEX from sending authenticated requests. Despite this anecdotal evidence, we lack a comprehensive overview of the websites whose cross-site functionalities may no longer work due to the new default policy. This paper intends to fill this gap, by identifying types of affected functionality and providing a first quantification of the affected websites. We answer these questions in §VI-A.

(RQ3) Lax Adequacy and Threats. The radical change introduced by the SameSite attribute is that browsers no longer include cookies to all cross-site requests by default, but only to those originating from predefined lists of same-site contexts. These lists capture those contexts typically used in XS attacks. For example, the Lax policy forbids including cookies in cross-site POST submissions as they are typically used in CSRF attacks. However, prior works (e.g., [2, 5, 8]) suggest that the Lax policy's coverage may not be adequate. Developers do not strictly obey the distinction between safe and unsafe HTTP methods and implement state-changing or state-leaking operations via, for example, GET requests.

One of the questions that we intend to answer in this paper is about the adequacy of the new same-site policies and their effectiveness. In particular, we want to focus on the tension between the contexts protected by the new policies and the contexts used by existing websites to implement security-sensitive operations. Second, non-academic security reports (see, e.g., [50–52]) have shown that, in certain cases, implementation mistakes can cause a bypass of the protection offered by the new SameSite policy. For example, web applications may not distinguish between different HTTP methods when processing incoming HTTP requests, allowing adversaries to forge protected requests, such as POST, by changing the HTTP method to non-protected methods, such as GET [51, 52]. However, it is somewhat unclear whether these vulnerabilities are outliers or widespread security problems. In this paper, we intend to provide a comprehensive evaluation of threats against web applications that rely on the same-site policies and determine their severity by looking for their prevalence in the wild. We answer these questions in §VI.

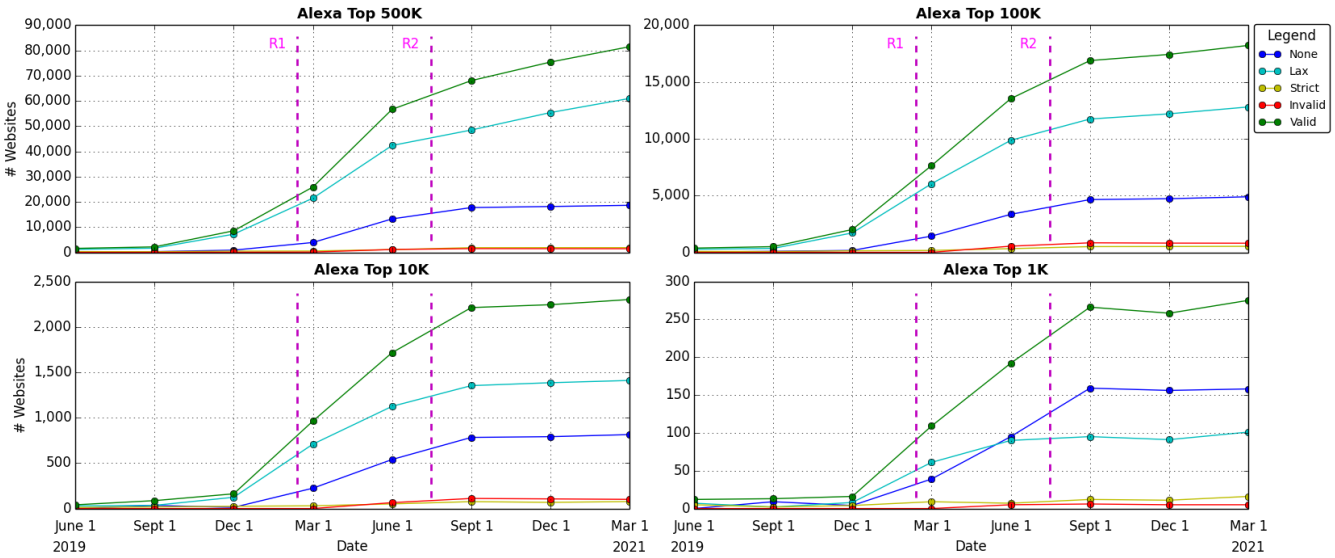
(RQ4) Browser Inconsistencies and Web Frameworks' Defaults. Our last research question investigates how consistent different (i) web browsers and (ii) web frameworks apply SameSite cookies on cross-site requests, and what are the divergent aspects among them. For example, the default policy in Chrome is Lax, whereas Firefox enforces the None policy. Even when browsers enforce a default Lax policy, web frameworks' APIs may downgrade it to None by default.

IV. SameSite Cookie Usage

The first part of this paper addresses RQ1, where we intend to measure how website developers adapted to the new cookie policies. In this section, we first review the methodology that we used to perform our measurements. Then, we present our findings.

Methodology. The methodology of this section consists in grabbing the cookie response headers of the Alexa top 500K sites (fetched in June 2020), and then extracting and counting the number of unique cookie attributes. Instead of conducting live measurements, we used the websites' copies stored by the Internet Archive, allowing us to retrieve past data enabling a longitudinal analysis of the usage of the SameSite.

Fig. 2: SameSite Cookies usage (June 2019 - March 2021). The vertical lines `R1` and `R2` mark the two rollouts of the new default policy by Chrome.



In total, we submitted queries to the Internet Archive for the response headers of the 500K domains archived from June 2019 to March 2021 with a three-months time interval. Fetching headers for 500K took in average six days. We divided the data collection in two periods: September 2020 and March 2021. At the end of each period, we performed a live measurement of the 500K domains to estimate the accuracy of Internet Archive's data. In both measurements, live measurements resulted in at most +0.5% more successful responses.

Trend Analysis. Figure 2 shows the usage of the different SameSite policies from June 2019 to March 2021. As of March 2021, 80.7% of the sites rely on the default cookie policy (i.e., 347,251) whereas 18.94% of them adopted one of the three valid policy. When looking at the trend, the rollout dates R1 and R2 seem to have played a relevant role with a steep increase of SameSite attribute usage, especially for the None policy within the top 1K sites. A small, yet non-negligible fraction of sites set an invalid policy (1,430 sites, i.e., 0.33%). An invalid policy is a string that does not match any of the three known policies, such as `SameSite=1`, which are most likely developers' mistakes. Invalid policies should be treated as the None policy by web browsers according to RFC 6265bis [20]. We refer interested readers to Table XIII of Appendix B that shows the top ten popular invalid policies. Finally, for at most 69,823 sites, a request to Internet Archive failed due to timeout.

None policy. In total, 18,640 sites use the None policy (i.e., 3.7%). However, we observe that the fraction of sites using None policy increases with the sites' popularity, from about one out of 10 among the top 10K sites (8.1%) to one out of five sites within the top 1K sites (i.e., 18%), making the None policy the most used policy among the top 1K sites.

Stricter policies. Then, 62,856 sites make it stricter with either the Strict policy or by explicitly setting the Lax policy. However, we note that the Strict policy is seldomly used (1,854 sites) when compared to the explicitly-set Lax policy, which covers almost all cases (61,002 sites, i.e., 97%).

V. Functionality Breakage

As we have shown in the previous section, 80.7% of the sites did not set the SameSite attribute, thus may rely on the new default policy which could break cross-site functionalities. In this section, we first identify the functionalities implemented via cross-site requests and then we provide a first measurement of the requests and websites that are affected. Before showing our results, we present the methodology of our analysis.

Methodology. Websites can use cross-site requests to implement various functionalities, e.g., advertising, social media buttons, etc. Identifying the functionality starting from a cross-site request is not trivial, and we are not aware of an automated technique able to do that. Accordingly, we design this section's experiments considering a human in the loop.

As we may need to evaluate requests manually, we limit our analysis to Alexa top 500 sites. We exclude duplicate sites (e.g., *google.com* and *google.co.uk*), sites that are not available in English (i.e., language barrier), and sites that do not offer free account creation and user login, resulting in 211 websites. Then, we register a user account for each site and crawl all sites using a browser enforcing the pre-SameSite default policy, searching for cross-site requests with cookies. We developed a JavaScript-enabled web crawler leveraging Puppeteer [53] and Chrome DevTools Protocol (CDP) [54]. Our crawler uses Chrome 83.0.4103.61, which does not enforce the new default Lax policy. Instead, it warns via the CDP Audits [55] when a cookie is attached to the request without specifying the `SameSite`, thus potentially breaking after the enforcement of the new default Lax policy. The seed URLs are the login pages, and after performing the user login with the manually-created credentials, the crawler follows a breadth-first visiting strategy to collect new URLs. The crawler stops

Functionality	# Requests	After R2 # Broken	After R2 # Patched
Advertising / User Tracking	374	93	281
Single-Sign On	81	1	80
Social Media Like / Share	76	11	65
Live Chat Frames	62	8	54
PDF Embed APIs	13	4	9
(Re-) CAPTCHA	12	2	10
Content Servers / CDNs	9	0	9
Survery/Rating Services	6	1	5
Total	633	120	513

Legend: R2= The Second Lax-by-Default Policy Rollout.

TABLE II: Overview of the affected functionalities.

when one of the two criteria is met first: it doesn't find new URLs, or the maximum of 200 URLs is reached. In total, our crawler collected 22,992 cross-site HTTP requests without a `SameSite`, which were initiated from 9,073 unique URLs.

To determine each request's high-level purpose, we first label our requests using the WebShrinker API [56]—a URL categorization service based on the industry-standard IAB taxonomy. The WebShrinker API is limited in regard to the short-lived, continuously changing advertisement domains. Accordingly, we also match the request URLs against the EasyList [57], Host BlackList [58], and Host BlockList [59]—three popular blocklists specialized for advertisement domains. Finally, we derive a more fine-grained list of functionalities by manually inspecting the requests. We pick 633 random requests out of the 22,992, three from each of the 211 websites, and manually identify the exact functionality implemented by the cross-site request. Then, we forcefully remove the cookies to observe whether the functionality is broken. Broken requests are marked as *affected*.

To precisely evaluate functionality breakage after the new policy's enforcement, we execute our experiments before and after the second rollout R2. In June 2020, before R2, we identified and collected cross-site requests as presented in this section. Then, in February 2021, after R2, we revisited the affected requests to confirm whether the new policy has indeed broken the functionality.

Categorization. The mapping between affected cross-site requests and IAB categories is in Table XII (Appendix B). In total, our crawler identified 22,992 cross-site HTTP requests without a `SameSite`, for a total of 9,073 unique URLs. The mapping identified 16 high-level categories of websites, providing 32 different types of functionalities. The vast majority of affected requests are for those sites offering technology and computing functionalities (e.g., file sharing, or live chat) or business services (e.g., advertising, marketing, or analytics), accounting for over 43.6% and 27.6% of the requests, respectively. For 303 requests of 17 websites, WebShrinker only matches the *uncategorized* category (see Table XII), thus we used EasyList, Host BlackList, and Host BlockList, and observed that all the 303 requests are for undesired content, e.g., ads.

While the IAB categorization provides insights about the type of the service provided by third-party sites, it does not help to identify the exact functionality implemented with cross-site requests. The manual investigation of 633 requests identified eight functionalities, as shown in Table II. Advertising and user tracking is the first type of functionality implemented via cross-site requests, covering 59% of the requests. Then, 12.8% of the requests are for Single Sign-On services, 12% for social media buttons, and 9% for embedded live chat services (e.g., *livechat.com*). The remaining groups are embedded PDF readers (e.g., Adobe Audience Manager), CAPTCHAs, CDNs (e.g., Gigya), and survey/rating services (e.g., *surveymonkey.com*).

Breakage. For each of the 633 requests, we manually confirm whether the new policy for cookies breaks the implemented functionality. We conduct these experiments in February 2021. First, we visit the affected site using a Chrome version enforcing the new Lax-by-default policy. Then we check whether the developers of the affected websites adopted one of the three SameSite policies to avoid service discontinuity. About 81% of the affected cross-site requests are correctly patched, and we do not observe breakage of functionality. However, we observe that the functionalities implemented by 19% of the affected requests are broken. Out of these, 77.5% of the requests are directed to advertisement networks, 9.2% to social media platforms, and 13.3% for the remaining functionalities.

VI. New Default Policy Adequacy and Threats

This section evaluates the adequacy of the new default policy in protecting websites in the wild and an extensive analysis of threats that can hamper the effectiveness of the new same-site policies.

Methodology. The first step of our analysis is identifying a list of threats, distinguishing them in threats leveraging cross-site requests that are not covered by the Lax policy and threats covered by the Lax policy. We systematically review academic literature (i.e., [2, 8, 9, 18, 19, 60–63]), the Stack Exchange [64] and the Dev [65] security communities, and 21 non-academic resources (i.e., [50–52, 66–83]). We searched for non-academic resources via Google search (up to page eight of the results). We used the search term "SameSite cookie" in combination with "bypass", "attack", and "vulnerability", and ignored irrelevant or redundant entries. We consider in scope those threats that can be exploited by a web attacker, and those that are relevant for SameSite cookies according to the identified resources (see, e.g., [2, 9, 23, 50, 51, 84]). Our review identified seven known threats. We also defined three new threats that are inspired by prior work. We present the threats in §VI-A. The second step of our analysis is determining the severity of the threats by looking at their prevalence in the wild. As each threat requires an ad-hoc testing procedure, we describe our tests in §VI-B.

A. Threats

We now present ten threats, of which four are against server-side end-points that are not protected by the Lax policy, and six due to vulnerabilities introduced by developers that can hamper the effectiveness of the SameSite attribute. Table III shows an overview of the threats.

Category	Threat	Attack COSI	Attack CSRF	Reference	Evaluation Testbed	% Vuln.	# Uniq. AV	# Apps
Not Protected By Lax	Replaying State-changing GET	○	●	[52, 72, 75, 76, 79, 80]	Top 1K	2.6% G-SCRs	7	4
	Window Properties Leak	●	○	[2, 78, 85]	Top 500	18.48%	1021	39
	postMessage Leak	●	○	[2, 86, 87]	Top 500	1.9%	11	4
	Pervasive Monitoring	●	○	[37, 88]	Top 500K	0.4%	2,080	2,080
Protected By Lax	Forging State-changing POST	○	●	[51, 73, 74]	Top 1K	1.5% P-SCRs	7	6
	SSC SSO Redirects Bypass	●	●	[50, 60, 83]	Top 10K	49.3%	6	4,935
	SSC Intra-Page Inconsistency*	●	●	[89, 90]	Top 500	1.4%	3	3
	SSC Inter-Page Inconsistency*	●	●	[18]	Top 500	3.3%	11	7
	SSC User-Agent Inconsistency*	●	●	[18, 19, 63]	Top 500K	1.8%	9,215	9,215
	Client-side CSRF vulnerability	○	●	[9, 91]	-	-	-	-

Legend: ● = threat applicable; ○ = threat not applicable; SSC= SameSite Cookie; AV= Attack Vectors ; G/P-SCR= GET/POST-based State-Changing Request.

TABLE III: Overview of threats to SameSite cookies, grouped by those not covered by Lax (top part) and those covered by Lax (bottom part). Threats marked with * are new, yet inspired by prior work.

1) Threats not Covered by the Lax Policy:

Replaying State-changing GET Requests. The new default policy does not prevent the inclusion of cookies in top-level navigation requests. If web applications use GET requests for security-sensitive state-changing operations, attackers can forge *authenticated*, cross-origin HTTP requests on behalf of victims, e.g., leveraging the `window.open()` JavaScript API (see Table I).

Window Properties Leak. Attackers can issue a top-level navigation request, and read the number of frames in a target web page using the `length` property of the opened JavaScript window objects. By comparing the frame count, attackers can leak the user state. For example, using this side-channel, it was to possible leak sensitive information about a user and their friends on Facebook [85], or to determine if a user owns a specific profile in Linkedin (i.e., user deanonymization) [2].

postMessage Leak. Similarly to the previous threat, attackers can issue top-level navigation requests using `window.open()`, listen to broadcasted postMessages from the opened web page, and leak the user state by comparing the set of observed messages [2].

Pervasive Monitoring. Third-party cookies are widely used to track users online, and they often contain sensitive data. If websites do not set the `Secure` attribute for these cookies, a viable threat is pervasive monitoring at network level. To mitigate this issue, Chromium-based browsers reject `SameSite=None` cookies without the `Secure` attribute [37, 84], but other browsers (e.g., Firefox and Safari) do not.

2) Threats Bypassing Protection of Lax or Strict Policy:

Forging State-changing POST Requests. One of the fundamental limitations imposed with the new default Lax policy is that an attack page cannot submit cross-site POST requests to a third-party context with the victim cookies attached. However, some applications are vulnerable in the sense that a state-changing POST request can be replayed and forged with a GET request interchangeably. In other words, the vulnerable application still processes the incoming request regardless of the HTTP verb used to submit the request. In this setting, the new default SameSite policy can be bypassed, e.g., by replaying the request using a top-level navigation GET request.

Single Sign-On HTTP Redirects Bypass. The Lax+POST exceptional policy (see §II) provides a time window of two minutes where Lax protection is not enforced, which is counted starting from the time of setting of a cookie. A possible attack consists of installing new cookies using cross-site requests and using the two-minute window to exploit XS vulnerabilities. Fresh cookies could be installed, for example, by abusing Single Sign-On Identity Providers (IdPs) that allow for user auto re-login via HTTP GET requests and without requiring user interaction (e.g., CAPTCHAs) [50]. The attack against a target site is the following. First, the attacker convinces a user to visit an attack page. Via the `window.open()` API, the page asks the IdP to re-login the user at the target site. As a result of the SSO login, the target site establishes a new authenticated session with the user's browser. Since the cookie is not older than two minutes, Lax protection of the target site is not enforced, enabling the attacker to mount XS attacks.

Listing 1: A vulnerable example of a duplicate cookie setting.

```
// for incompatible clients
Set-cookie: 3pc-legacy=value;
// for newer clients
Set-cookie: 3pc=value; SameSite=Strict;
```

SameSite Cookie Intra-page Inconsistency (new). When developing web applications, providing support for older web browsers that are incompatible with the SameSite cookie policy is challenging. In such incompatible clients, a cookie marked with a `SameSite` attribute or an unsupported SameSite policy may be rejected and not set, thus breaking the application functionality. As a workaround, developers may set redundant cookies, both with and without the `SameSite` attribute [89, 90], or with different `SameSite` policies. However, this can introduce vulnerabilities if not properly applied. For example, Listing 1 shows a vulnerable cookie setting that can be exploited to mount a CSRF attack. In this example, the application sets two duplicate cookies, namely `3pc` and `3pc-legacy`, with Strict and no `SameSite` policy, respectively, and resorts to the `3pc-legacy` if the `3pc` cookie is not included in the request. For a victim vising a CSRF attack page using a modern client, the `3pc` cookie is not attached to the cross-origin CSRF request, but the `3pc-legacy` cookie is still automatically attached to the

request, both when assuming a client enforcing a default None or default Lax policy (i.e., using top-level navigation requests), enabling CSRF on server-side.

SameSite Cookie Inter-Page Inconsistency (new). This vulnerability occurs when a web application sets two different SameSite cookie policies for the same cookie with the same `Path` attribute across two different web pages. For example, if an application sets `3pc=value; SameSite=Strict; Path=/` when visiting URL_1 and `3pc=value; SameSite=None; Path=/` when visiting URL_2, then the Strict policy for this cookie can be bypassed. In this example, the bypass happens by issuing a top-level navigation request to URL_2, which overwrites the cookie with the `SameSite=None` attribute, relaxing the SameSite policy.

SameSite Cookie User-Agent Inconsistency (new). This vulnerability arises when an application set inconsistent cookie policies when using two different user agents. For example, a web application may enforce the Strict or Lax policy for a sensitive cookie when the user is using a desktop browser, but enforce the None policy if the user uses a mobile browser (or vice versa). One reason for such inconsistency is that the mobile and the desktop version are two different applications exposing themselves to the public based on the request user-agent. In such circumstances, CSRF and COSI attacks are possible provided that the victim uses a user agent with the less stricter SameSite policy to visit the target website.

Client-side CSRF. When an attacker-controllable input of client-side JavaScript program is used to generate same-site requests, a web application is exposed to client-side CSRF attacks. As these requests are same-site, the browser will attach cookies, even when using the Strict policy. In this paper, we do not examine the prevalence of this threat as it has been extensively studied in a recent work (see, i.e., [9]).

B. Threats Prevalence in the Wild

Starting from the threats of §VI-A, we now quantify their impact and prevalence in the wild.

Summary of Findings. Our evaluation shows that the coverage of the new default SameSite cookie policy (i.e., Lax) is not sufficient to protect web applications from a noticeable set of CSRF and COSI attacks. More specifically, as we show later, the first category of attacks presented in this section leverage cross-site request contexts that are not covered by the Lax policy. Then, the second category of attacks demonstrate the prevalence of cases where the protection of Lax can be circumvented. Table III summarizes our findings.

Lax Adequacy. In our evaluation, we identified both CSRF and COSI attacks against popular websites. First, GET requests (not covered by the Lax policy) are still prevalently used for state-changing operations—accounting for over 10.3% of the total state-changing requests in top 1K Alexa websites, 2.6% of which can be leveraged for mounting CSRF attacks by replaying the request (e.g., in Mailchimp or Pixiv). Second, we discovered 1,032 distinct information leakage (i.e., COSI) vulnerabilities affecting 40 websites of Alexa top 500, including Tumblr, Twitch and SoundCloud, that leak the user's login status or identity leveraging window properties and postMessage side-channels. We detail these threats in Sections VI-B1 to VI-B3.

Method	POST	GET	Total
# Reqs	6,230 (89.6%)	721 (10.3%)	6,951
# URLs	1,870	251	2,121 / 42,571
# Apps	602 (65.2%)	88 (9.5%)	690 / 922

TABLE IV: State-chaning GET and POST requests in Alexa top 1K websites.

Bypassing Lax. We identified a wide range of attacks that hamper the effectivenss of the Lax policy. For example, we discovered that 1.5% of the sensitive state-changing POST requests can be exploited for a CSRF attack by using the GET HTTP method instead. We found instances of these CSRF attacks in popular websites, e.g., IMDB, PayPal, or Meetup. Also, we discovered six unprotected SSO IdPs, including Google, Facebook and Linkedin, that enable trivial bypass of the Lax policy on over 49% of the top 10K Alexa sites, leveraging the exceptional Lax+POST policy. When looking at SameSite cookie policy inconsistencies, our evaluation of popular Alexa top 500 websites revealed seven vulnerable sites with inter-page policy inconsistencies, including Vimeo, AliExpress and Office365, as well as three vulnerable sites with intra-page policy inconsistencies, i.e., GitHub, CNN, and Yahoo. Finally, we discovered 9,951 vulnerable websites with inconsistent SameSite cookie policies for different user-agents, by systematically testing half a million Alexa sites, which can be used to bypass the constraints of the new SameSite cookie setting. We detail these threats in Sections VI-B4 to VI-B8.

1) Replaying State-changing GET Requests: In this section, we show that state-changing requests that use the GET method and that are *not* protected by the new default SameSite cookie policy are prevalent, and we demonstrate real-world CSRF exploitations in popular websites leveraging such requests.

Quantification of Request Types. Our methodology to quantify the prevalence of state-changing GET requests are as follows. We use the web crawler of JAW [9] to crawl Alexa top 1K websites. The crawler stores a JavaScript-enabled, DOM snapshot of the web page after ten seconds. To identify state-changing requests at large-scale, we create a script that finds all HTML forms in the DOM snapshots with an anti-forgery token—an indication that the HTTP request would change the server-side state when the form is submitted. Then, the script filters all forms based on their HTTP method, quantifying their prevalence. We note that, even if these requests are seemingly protected by a CSRF token, the implementation of the defense may have been wrong (e.g., faulty token verification when overriding the HTTP request method [92, 93])—a mistake that may have acted as a contributing factor for the introduction of the new SameSite cookie setting. Finally, we compare our findings with prior work, i.e., the data of Mitch [8].

Table IV summarizes the results. In total, the crawler finds 42,571 URLs for 922 websites. In these pages, our script identifies a total of 6,951 state-changing requests. Out of this number, the majority, i.e., 6,230 are POST-based requests.

Rank	Website	Replay Req.	Forge Method	Total Req.
58	imdb.com	0	2	2
81	fandom.com	0	2	2
102	paypal.com	0	1	1
289	ilovepdf.com	0	1	1
300	investing.com	2	0	2
427	meetup.com	0	2	2
524	mailchimp.com	1	1	2
586	brilio.net	3	0	3
627	pixiv.net	1	0	1
Total Vuln.	9 / 690	7 / 264	9 / 602	16 / 866

TABLE V: Summary of CSRF vulnerabilities discovered for a set of randomly selected requests of Alexa top 1K websites.

Still, a noticeable fraction of all identified state-changing requests are based on the GET HTTP method, i.e., over 10.3%, which as we will show, is in line with prior research [8]. Specifically, we use the dataset of Mitch [94] to confirm our findings. The dataset contains a total of 58,828 HTTP requests for 60 popular Alexa websites. Out of this number, we observe that 938 requests contain an anti-forgery token, an indication that the request is state-changing. From 939 requests, 121 use the GET HTTP method, i.e., 12.8%, which is statistically close to our finding of 10.3%. Therefore, GET requests are still used in practice for state changes, despite the fact that they are not protected with the default SameSite cookie policy.

Exploitations. We manually explored the collected data to detect concrete GET-based CSRF exploitations. Given the scale of our data, we randomly selected three GET requests from each web application for which we detected a GET request, i.e., 88 applications (see Table IV), resulting in a total of 264 requests. Then, for each selected request, we checked if the CSRF token verification is performed correctly by replaying it. Table V presents our findings. In total, we discovered that seven out of the 264 GET requests (i.e., 2.6%) are forgeable due to faulty CSRF token verification, affecting four websites, i.e., Mailchimp, Brillo, Investing, or Pixiv. We created a working proof-of-concept exploit for each vulnerable web application. The exploits allow an attacker to delete user sketches in Pixiv, delete articles, videos and pictures (i.e., user-generated content) in Brillo, create or remove user portfolios in Investing, and finally change user settings' defaults (e.g., notifications) in Mailchimp.

2) Window Properties and postMessage Leaks: We investigate the prevalence of window properties and postMessage XS-Leaks using the data of our crawl of §V on the Alexa top 500 websites, i.e., 9,073 URLs of 211 websites. We automatically explore the presence of login detection and user deanonymization attack vectors leveraging dynamic analysis. Specifically, we create a script that, for each URL in our dataset, loads a candidate test web page inside the browser in two different user states, i.e., logged and not logged for login detection and logged as two different users for deanonymization. For window properties leak, the test web page opens the URL in a new window leveraging the `window.open()` API, and reads the `length` property of the opened window, repeating this process for both user states under test. Similarly, the same process is performed for postMessage leaks, but instead of reading the number of frames in the opened window, the test web page listens for broadcasted postMessages using the `window.addEventListener` API. Finally, the script compares the values collected at the two different states, and outputs the set of state-dependent, leaky URLs.

XS-Leak		Login Det.	Deanonym.	Total
PM	# Apps	4	1	4
	# URLs	9	2	11
WP	# Apps	39	8	39
	# URLs	986	35	1,021
Total	# Apps	40	8	40
	# URLs	995	37	1,032

TABLE VI: Summary of Window Properties (WP) and postMessage (PM) information leakage vulnerabilities discovered in Alexa top 500 websites.

	# Vuln. Alexa Websites			
Crawl	**Top 1K**	**Top 10K**	**Top 100K**	**Top 500K**
June 2019	0	0	3	12
Sept. 2019	9	34	70	113
Dec. 2019	1	4	35	135
March 2020	6	29	227	644
June 2020	12	74	556	1,918
Sept. 2020	12	82	609	2,076
Dec. 2021	12	71	597	1,987
March 2021	12	82	634	2,148

TABLE VII: Summary of the discovered security risks due to the missing `Secure` flag in `SameSite=None` cookies.

Table VI summarizes the results of our experiment. In total, we discovered 1,302 vulnerable URLs, belonging to a total of 40 distinct websites. Out of 1,302, 37 URLs can be exploited for deanonimyzing the user's identity in eight websites, i.e., Tumblr, Twitch, AliExpress, Blogger, Office365, Tokopedia, Ebay, and SoundCloud, and the rest (i.e., 995) can be trivially exploited for mounting login detection attacks in all 40 vulnerable websites, including privacy-sensitive sites, such as PornHub. Note that being logged in implies having an account, which may be problematic for privacy-sensitive websites. Overall, we observe that 18.4% and 1.9% of the tested web applications are vulnerable to the window properties and postMessage side-channels, respectively.

3) Pervasive Monitoring: We reuse the data we collected from Internet Archive between June 2019 to March 2021 (§IV) to identify cookies marked with `SameSite=None` that miss the `Secure` flag. In total, we detected 2,148 websites who are at risk of compromising user's privacy, 12 and 82 of which belong to the top 1K and 10K websites, respectively. Table VII summarizes the results of our analysis. We observe that there is an increasing trend on the instances of this security risk in the wild.

4) Forging State-changing POST Requests: We reuse the data we collected in §VI-B1 to assess the prevalence of forgeable state-changing POST requests. We manually explored the collected data to detect concrete instances of forgeable POST requests, where attackers can bypass the Lax protection by changing the HTTP method. As shown in Table IV, in total, we identified 6,230 state-changing POST requests in 602 web applications of Alexa top 1K websites. Given the scale of the data, we randomly selected one state-changing POST request per web application, resulting in 602 requests. For each selected request, we checked the susceptibility to a

CSRF attack by replaying the request using a different HTTP method, i.e., GET. In addition to the HTTP method change, we encode the key-value pairs in the POST request body, if any, in the form of GET request query parameters. Table V summarizes our findings. In total, we discovered that nine out of the 602 requests (i.e., 1.5%) are forgeable, affecting six popular websites (e.g., PayPal, IMDB, or Meetup). We created a proof-of-concept exploit for each of the six vulnerable web applications. The exploits allow an attacker to add or remove movies from a user watchlist in IMDB, change user settings (e.g., name, gender, or profile title) in Fandom, modify user invoices and extend the user session in PayPal, editing a user's signature in iLovePDF, and finally creating or removing notification alerts in Meetup.

5) Single Sign-On HTTP Redirects: To identify SSO IdPs that enable bypass of the Lax policy, we create a web application that integrates SSO using 13 different popular IdPs, i.e., Google, Facebook, Amazon, Apple, Microsoft, Linkedin, Github, Twitter, VK, Mail.ru, Twitch, Instagram, and Yahoo. To derive the list of popular IdPs, we manually review Alexa top 500 sites, and list the IdPs they use for SSO. We investigate if each IdP can be leveraged to bypass Lax by checking if it offers a cross-origin GET-based auto re-login feature that does not require any user interaction (e.g., CAPTCHA). For each affected IdP, we find websites from Alexa top 500 that integrate a SSO feature via that IdP, and verify if the attack still works in the real-world setting.

To quantify the impact of affected IdPs on websites, We built a JavaScript-enabled, Chrome-based web crawler on the top of XDriver [62], and used it to detect the IdPs each website is using for the SSO. The detection of the IdPs is similar to that of [62], and is based on a set of fine-grained static probes and regular expressions that we design for each IdP. We use our crawler to detect the IdPs in Alexa top 10K websites, examining tens of thousands of web pages.

Results. In total, we found six SSO IdPs that enable trivial bypass of the new default policy, i.e., Google, Facebook, Microsoft, Linkedin, VK and GitHub. These IdPs are integrated in 4,935 websites, accounting for more than 49% of the top 10K Alexa sites. To identify the affected websites, our crawler examined a total of 208,464 web pages of 9,485 sites in a period of around two weeks, designating 6,638 login pages, out of which 5,180 are login pages with an SSO. From these pages, 4,935 are login pages with at least one of the six affected SSO. For 515 sites, our crawler failed because either the website was unresponsive or XDriver failed when looking for DOM elements. Table XIV in Appendix B summarizes our findings.

False Positives. To evaluate the potential false positives (FPs) of our automated SSO detection mechanism, we randomly selected 500 websites, and manually verified the detected IdPs. This resulted in a total of seven FP IdP instances for four websites. In all cases, the underlying reason for the FP was that the heuristic used to match the existence of the IdP was present in the website for non-SSO usecases, e.g., in websites containing tutorials, or documentation about an SSO. Accordingly, our crawler exhibits an estimated false positive rate of 7/(13×500) IdP instances, or 4/500 websites (0.8%).

Rank	Website	Policy Downgrade	# Vuln. Cookies
38	aliexpress.com	Lax to None	2
148	kompas.com	Lax to None	3
151	office365.com	Lax to None	1
170	canva.com	Strict to None	1
176	vimeo.com	Lax to None	1
191	abs-cbn.com	Lax to None	2
199	aliyun.com	Strict to None	1
Total	7		11

TABLE VIII: Summary of inter-page SameSite cookie inconsistencies in Alexa top 500 websites.

6) SameSite Cookie Inter-Page Inconsistency: We investigate inter-page policy inconsistencies using the data of our crawl of §V on the top 500 Alexa websites. We create a script that compiles a list of cookies set on each website together with the URL of web pages on which the cookie was set. Then, the script looks for redundant cookie entries across web pages, and for each matching case, it checks if the value of the `SameSite` attribute is consistent in all cases, and otherwise, it reports the inconsistency. Finally, for each reported case by the automated script, we manually confirm the inconsistency on the live instance of the application.

In total, out of the 211 websites of the top 500 Alexa, this process led to the detection of seven vulnerable sites having a total of 11 cookies with policy inconsistencies. This includes, among others, popular websites such as AliExpress, Vimeo, and Office365. In all cases, an attacker can downgrade the Lax or Strict policy to None. Table VIII summarizes the results.

7) SameSite Cookie Intra-Page Inconsistency: We explore the presence of duplicate cookies having inconsistent SameSite cookie policies with a semi-automated approach, leveraging the data of our crawl of §V on the Alexa top 500 websites. Specifically, we create a script that compares all the cookies set in a web page with each other. The script checks if it can find a pair of cookies that have the same value. If a match is found, it checks if their specified `SameSite` policy is different, i.e., no `SameSite` or None policy for one cookie, and Lax or Strict for the other. Since two session cookies may trivially have the same value (e.g., an integer), yet do not encode the same semantics, the script also apply certain heuristics, e.g., the length or type of the strings. Finally, it reports all cookie pairs that match these properties. For each cookie reported, we manually review and confirm the existence of a vulnerability to eliminate false positives.

In total, the script reported 22 cookie pairs of eight websites out of the 211 sites under test. However, manual investigation revealed that only three websites (nine cookie pairs) are vulnerable, i.e., GitHub, CNN, and Yahoo, accounting for 1.4% of the tested websites. For example, CNN sets a pair of duplicate cookies named `obuid` and `OB-USER-TOKEN` with the exact same value but with different SameSite cookie policies, i.e., no `SameSite` attribute and the Strict policy, respectively. Similarly, Yahoo sets duplicate session cookies with different policies, i.e., None and Lax. Finally, GitHub uses a pair of cookies named `user-session` and `__Host-user-session-same-site` with inconsistent

Crawl	(M, N)	(M, L)	(M, S)	(M, NS)	(M, I)	Total
June 2020						5,719
(D, N)	-	2,263	70	213	0	
(D, L)	2,382	-	244	157	0	
(D, S)	72	244	-	9	0	
(D, NS)	217	133	5	-	0	
(D, I)	0	0	0	0	-	
Sept. 2020						9,215
(D, N)	-	3,262	299	1,282	0	
(D, L)	3,167	-	381	759	0	
(D, S)	234	378	-	26	0	
(D, NS)	172	268	13	-	0	
(D, I)	0	0	0	1	-	
April 2021						9,951
(D, N)	-	3,572	328	1,166	0	
(D, L)	3,516	-	431	781	0	
(D, S)	302	432	-	35	0	
(D, NS)	135	278	33	-	0	
(D, I)	0	0	0	1	-	

Legend: D= Desktop; M= Mobile; N= None; L= Lax; S= Strict; NS= Not Set; I= Invalid.

TABLE IX: Summary of SameSite cookies' inconsistencies across mobile and desktop clients of Alexa top 500K websites. The total column shows the number of vulnerable sites where a policy downgrade can occur.

policies, i.e., no `SameSite` policy and Strict, respectively.

8) Inconsistent Policy for Different User-Agents: To determine inconsistent policies based on the user-agent, we performed three web crawls on live instances of Alexa top 500K websites on June 2020, September 2020, and April 2021 using two different User-Agents for mobile and desktop clients (see Appendix B). Accordingly, we compare if the SameSite cookie policy is set differently for the same cookie across the HTTP responses captured for desktop and mobile clients.

Table IX summarizes our findings. In total, we identified 5,719, 9,215, and 9,951 vulnerable websites that allow a policy downgrade in the three web crawls, respectively. Note that not all entries in Table IX may lead to a policy downgrade, i.e., pairs that have the Lax policy in one client, and do not set any policy in the other client do not lead to a policy downgrade assuming the new default policy. Finally, out of the 9,951 vulnerable websites, 138 are among the top 1K Alexa websites, showing that such inconsistencies are prevalent among popular sites. We refer interested readers to Table XV in Appendix B which shows the number of policy inconsistencies grouped by site popularity.

VII. Web Browsers and Web Frameworks

The final analysis of our paper looks at the inconsistency between browsers when handling and enforcing the SameSite attribute properties (§VII-A), and it looks at the default policies used by popular web frameworks (§VII-B).

A. Evaluation of Web Browsers

Browsers exhibit a variety of behaviours when applying SameSite cookies. For example, the default policy in Chrome and Opera is Lax, whereas Firefox and Safari enforce the None policy by default. Even with regards to Chrome, the latest IOS version (87.0.4280.77) still uses the None policy by default [88]. Also, such inconsistencies not only apply to the default setting, but also to other corner cases where a request is sent in cross-site context, e.g., when `SameSite=None` cookies are used without a `Secure` flag, or when the `SameSite` attribute has an invalid or even the Lax or Strict value.

Methodology. We conducted our analysis against 14 web browsers and investigated their compliance with the new RFC 6265bis specification [20]. The list of popular browsers for testing is from MDN [37], and we add the iOS Chrome and Tor Browser. Furthermore, for Safari, we consider three different versions that are frequently used by the three recent macOS operating systems, since Safari cannot be upgraded standalone [95]. We automated the analysis by developing three webpages, two in the same origin and the third page in a different origin, where the first page performs same-site and cross-site requests from different contexts towards the second and third page, respectively. Then, the analysis of the logs of the web servers reveals which request was submitted with cookies.

Results. Table X summarizes our findings. In total, we identified seven distinct ways on how browsers enforce the SameSite cookie policy in same-site and cross-site context. We observed that, to date, none of the 14 tested browsers are fully compliant with the new RFC 6265bis specification [20], including Chrome. The most RFC-compliant browsers are Chrome, Chrome on Android, Opera, Opera on Android, and Edge, which comply for 11 out of the 12 possible cases of how the SameSite cookie attribute can be set in same-site and cross-site contexts, as shown in Table X.

As of today, web application developers need to be aware of all these seven behaviors if they want their website to (i) work with all these browsers and (ii) provide the same security guarantees. One way to achieve that is using user-agent-dependent SameSite policies. While this may seem a valid solution, we have seen in the past that header inconsistencies can be the root cause of vulnerabilities (see, e.g., [19] or our SSC User Agent Inconsistency vulnerabilities in §VI-B8).

B. Evaluation of Web Frameworks

Even when browsers enforce a default Lax policy, web frameworks' built-in APIs can downgrade it to the None policy by default. Accordingly, we examined the top five frameworks of top five programming languages, with the overarching goal of identifying frameworks that relax the browser's default SameSite cookie policy when a cookie is set.

Methodology. First, we select the top five web programming languages based on GitHub's 2020 Octoverse report [123] (i.e., JavaScript, Python, Java, PHP, and C#). Then, we compile a list of frameworks for each language, and quantify their popularity based on a series of criteria (ordered): number of tagged questions in Stack Overflow [124], number of uses by other GitHub repositories, number of GitHub stars, forks and watches, and the number of downloads in package managers of each language. Accordingly, we pick the top five frameworks of each language, resulting in a total of 25 frameworks (see Table XVI of Appendix B). Then, we resort to the documentation of each framework to see if it has built-

Browser/ Spec	Version	Scope	Set-Cookie HTTP Header					
			K=V	K=V; SS=None; Secure	K=V; SS=None	K=V; SS=Invalid	K=V; SS=Lax	K=V; SS=Strict
Specification	RFC 6265bis [20]	Same-Site	●	●	○	●	●	●
		Cross-Site	○	●	○	●	○	○
Tor Browser	10.0.12	Same-Site			✗			
		Cross-Site		✗		✗		
Chrome	89.0.4389.82	Same-Site						
		Cross-Site				✗		
Opera	74.0.3911.218	Same-Site						
		Cross-Site				✗		
Edge	89.0.774.54	Same-Site						
		Cross-Site				✗		
Firefox	86.0	Same-Site			✗			
		Cross-Site	✗		✗			
Safari	14.0.3	Same-Site			✗			
		Cross-Site	✗		✗			
Safari	12.0.3, 13.1.1	Same-Site			✗			
		Cross-Site	✗	✗		✗		
IE	11.0	Same-Site			✗			
		Cross-Site	✗		✗		✗	✗
Andr. Chrome	84.0.4147.124	Same-Site						
		Cross-Site				✗		
Andr. Opera	61.2.3076.56749	Same-Site						
		Cross-Site				✗		
Andr. Firefox	79.0.5	Same-Site			✗			
		Cross-Site	✗		✗			
iOS Safari	14.4	Same-Site			✗			
		Cross-Site	✗		✗			
Samsung Int.	13.2.1.70	Same-Site			✗			
		Cross-Site	✗		✗			
Andr. WebView	84.0.4147.105	Same-Site			✗			
		Cross-Site				✗		
iOS Chrome	87.0.4280.77	Same-Site			✗			✗
		Cross-Site	✗		✗			

Legend: K= Key; V= Value; SS= SameSite; ● = Cookie Sent; ○ = Cookie Not Sent;
✗ = Divergent From/Not Compliant with Specification.

TABLE X: Overview of web browser's compliance with RFC 6265bis [20]. Browsers with similar behaviours are grouped with the same color. The table highlights a total of seven distinct browsers' implementations when enforcing the SameSite cookie policy, each marked by a different color.

in support for SameSite cookies. If so, we create a basic web application using the default configuration of the framework, and use the frameworks' cookie APIs to set a cookie. Finally, we run the application and investigate if the framework did set a `SameSite` attribute on the cookie by default.

Results. Table XI summarizes our findings. First, out of the 25 frameworks, 21 frameworks provide built-in APIs to control the SameSite policy when setting a cookie, out of which in three frameworks, not all the three SameSite policies are supported. Then, we observe that six out of the 25 frameworks (i.e., 24%) specify the None policy by default for all cookies set. For example, when a developer uses the API `set_cookie(k, v)` in Django [21] or Pyramid [22], the framework sets the cookie `k=v; SameSite=None`, adding a None policy semi-transparently to the developer.

VIII. Discussion

The Hidden Costs of Pre-packaged Policies. In this paper, we quantified a significant fraction of the attack surface that remains unprotected by the SameSite policy and exposed to XS attacks. Protecting such a fraction of the attack surface is a considerably harder and more costly task, requiring developers to revisit the design and implementation of their systems (e.g., removing state-changing GET) and being aware of both precise corner case behaviors of browsers and web frameworks. To date, developers must adapt their existing

Language	Framework	Version	SameSite Cookies Support	SameSite Cookies Default	Reference
Python	Flask	1.1.2	◓	Not Set	[96]
	Django	3.1.7	●	None	[21]
	Tornado	6.1	◓	Not Set	[97]
	Pyramid	2.0	●	None	[22]
	Web.py	0.62	●	None	[98]
JavaScript	Express	4.17.1	●	Not Set	[99]
	Meteor	2.1	○	Not Set	[100]
	Sails	1.4.1	●	None	[101, 102]
	Koa	6.1.0	●	Not Set	[103]
	Hapi	20.1.0	●	Strict	[104]
PHP	Laravel	8.16.1	◓	Lax	[105]
	Symfony	5.2	●	Lax	[106, 107]
	CakePHP	4.2.4	●	Not Set	[108, 109]
	Zend	1.12	●	Not Set	[110]
	Slim	4.7.0	●	Lax	[111, 112]
C#	ASP WebForms	4.7.2	●	None	[113]
	ASP MVC	4.7.2	●	None	[114]
	ASP Core	5.0	●	Not Set	[115]
	Nancy	1.4.4	○	Not Set	[116]
	Service Stack	5.1	●	Lax	[117]
Java	Spring	5.3.4	●	Lax	[118]
	Play	2.8	●	Lax	[119]
	Vaadin	8.0	○	Not Set	[120]
	Vert.x-Web	4.03	●	Not Set	[121]
	Spark	3.1.1	○	Not Set	[122]

Legend: ● = Fully Supported; ◓ = Partially Supported; ○ = Not Supported;

TABLE XI: Evaluation of SameSite cookie policy in top five frameworks of top five programming languages.

web applications to three predefined sets of admitted contexts, which is in stark contrast with other web security policies (e.g., CORS and CSP) where developers have fine-grained options to customize a security policy to their needs. We believe that such flexibility and customization could help developers fully protect their web applications. We hope that our work encourages researchers to take on the challenges of going beyond static, pre-packaged policies and exploring more flexible and customizable SameSite policies.

Correct and Secure Use Require Awareness. While the Lax-by-default policy is a relatively new mechanism that could help protect from XS attacks, it requires developers to know the precise cross-site request contexts that are and are not protected. In this paper, we identified six cross-site contexts that are not covered by the Lax policy (Table I), which are exposed to XS attacks. For example, we observed that over 10.3% of state-changing operations in Alexa top 1K sites use the GET method, and 2.6% of them can be trivially exploited to mount a CSRF attack.

SameSite for Defense in Depth. Switching to the new Lax policy requires further care by developers as even the contexts covered by the Lax policy can be still be abused for XS attacks. For example, we showed that 1.5% of POST requests of top 1K Alexa sites that are seemingly protected by Lax can be successfully forged by replaying the request with the GET HTTP verb that is not protected by Lax. Also, SameSite policies should be used consistently across pages and across website versions to avoid introducing security gaps.

Advertisement Services Affected the Most. Our functionality breakage analysis showed that as of Februrary 2021, 19% of cross-site requests without the SameSite attribute are no longer working, affecting the most the advertising services (77.5%).

Browsers Diverge on SameSite Cookie Policy. Our analysis of 14 different web browsers uncovered seven distinct types of behaviours when enforcing SameSite cookies. These divergent enforcements may urge application developers to implement ad-hoc solutions to handle cookies of different user clients differently, e.g., by dynamically generating the SameSite policy per client, or by setting duplicate cookies, one for each intended client. For example, we discovered that the Lax policy can be bypassed in 1.4% and 3.3% of the tested top 500 sites due to cookies with inconsistent SameSite policies, either within a web page (duplicate cookies), or across multiple pages, respectively. We believe such divergence will narrow down over time.

Change of the Browser's Flag Is the First Step. SameSite cookies are a robust defense-in-depth mechanism against some classes of XS attacks. However, developers needed to opt-into its protections by explicitly specifying a `SameSite` attribute. Accordingly, changing the default browser's SameSite cookie flag to Lax helps transition from an "opt-in" to an "opt-out" solution. While such change is a promising first step, it is not enough to complete this transition. For example, we observed that 24% of the top 25 web frameworks set the None policy by default when a cookie is set, which can downgrade the browser's default SameSite cookie policy, and requires developers to explicitly opt-into stricter policies. In addition, external functionalities, such as the integration of the application to third-party services, may be leveraged to compromise the Lax protection, and thus need to be reviewed. For example, for the top 10K Alexa sites, the Lax policy can be trivially bypassed in over 49% of the sites due to their integration with vulnerable identity providers.

Vulnerability Notification. In February 2021, we started the notification process to browsers, web frameworks, IdPs and websites following the best practices of vulnerability disclosure [125]. We prioritized our reports by severity. We sent an initial notification followed by an additional reminder every month, including a detailed description of the security risk, or a proof-of-concept exploit for vulnerabilities, e.g., the user deanonymization leaks in Tumblr and Twitch, or CSRF vulnerabilities in Meetup, PayPal, and Mailchimp, who confirmed the vulnerabilities, and patched them. We detail the latest status of our notification campaign in Appendix C.

IX. Related Work

HTTP Cookies. The study of cookies in Web security has primarily focused on cookie integrity attacks (e.g., [126–130]), and third-party cookies (e.g., [131, 132]). As a response to the nefarious role of third-party cookies in XS attacks, previous research also proposed multiple approaches to automatically strip session cookies from cross-site requests, e.g., using server-side proxies [24, 133], browser extensions [1, 10, 134, 135], or both [136]. Franken et. al. [60] proposed a framework to evaluate the correct enforcement of these cookie stripping policies on cross-site requests, by analyzing the security mechanisms of browsers and browser extensions. Other works studied the usage of cookie security attributes. For example, Sivakorn et. al. explored the adoption of the `Secure` flag [137], and Singh et. al. measured its usage [138]. Similarly, Zhou and Evans studied the usage of the `HTTPOnly` attribute [139]. In contrast to these works, we focus on the `SameSite` attribute. Closely related to our work, Calvano [44, 140] analyzed the usage of SameSite cookie policies using HTTP archive. Similarly, the proprietary BuiltWith website reports the usage of SameSite cookies [141]. Our work completes the missing pieces from these analyses, systematically studying the trend of the adoption of valid and invalid SameSite cookie policies, and the impact and effectiveness of Lax-by-default cookies.

Web Inconsistencies. Analysis of Web inconsistencies has been considered by several researchers in the past. Most notably, previous research studied the inconsistent deployment of HTTP headers between the desktop and mobile variants of a website [19, 63]. Other works addressed incoherencies in browser access control policies and SOP (see, e.g., [2, 138, 142, 143]). More closely related to the attacks presented in our work, Calzavara et. al. studied the inconsistent adoption of security mechanisms across different web pages of an application [18]. As opposed to these works, in this paper, we uncovered inconsistencies of the browsers with regards to the SameSite cookie policy. Also, we studied the protective coverage of the new default Lax policy, and systematically identified and proposed attacks that can bypass it, primarily based on incoherencies in the `SameSite` cookie attribute.

X. Conclusion

In this paper, we performed, to the best of our knowledge, the first security evaluation of SameSite cookie policy, systematically covering the trend of its usage, the impact of the new default policy, and the threats against it, with the overarching goal of studying how effectively SameSite cookies can mitigate XS attacks. We quantified the prevalence of vulnerabilities of each threat in the wild, showing that (i) XS attacks can still be mounted in popular web applications leveraging requests that are not protected by the default Lax policy, thus requiring developers to be aware of the unprotected requests and the additional security risks, and (ii) even if developers use the default Lax policy correctly and as a defense-in-depth, application-level vulnerabilities, such as forgeable state-changing POST requests, or intra-page and inter-page cookie policy inconsistencies can continue to cause XS attacks, despite the presence of the Lax policy. Finally, we showed that browsers diverge when enforcing SameSite cookies, and that web frameworks' default APIs can undermine the browser's enabled-by-default Lax protection. Overall, we believe SameSite cookies are a powerful defense-in-depth that can help reduce the attack surface for XS attacks. However, their correct and secure use require developer's awareness and expertise.

REFERENCES

[1] M. Johns and J. Winter, "RequestRodeo: Client-side Protection Against Session Riding," 2006, https://www.owasp.org/images/4/42/RequestRodeo-MartinJohns.pdf.

[2] A. Sudhodanan, S. Khodayari, and J. Caballero, "Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks," in *Network and Distributed Systems Security Symposium*, 2020.

[3] XS-Leaks Wiki. https://xsleaks.com/.

[4] S. Lekies, B. Stock, M. Wentzel, and M. Johns, "The Unexpected Dangers of Dynamic JavaScript," in *USENIX Security Symposium*, 2015.

[5] C. A. Staicu and M. Pradel, "Leaky Images: Targeted Privacy Attacks in the Web," in *USENIX Security Symposium*, 2019.

[6] A. Sudhodanan, R. Carbone, L. Compagna, and N. Dolgin, "Large-scale Analysis & Detection of Authentication Cross-site Request Forgeries," in *IEEE European Symposium on Security and Privacy*, 2017.

[7] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow, "Deemon: Detecting CSRF with Dynamic Analysis and Property Graphs," in *ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[8] S. Calzavara, M. Conti, R. Focardi, A. Rabitti, and G. Tolomei, "Mitch: A Machine Learning Approach to the Black-Box Detection of CSRF Vulnerabilities," in *IEEE European Symposium on Security and Privacy*, 2019.

[9] S. Khodayari and G. Pellegrino, "JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals," in *30th USENIX Security Symposium*, 2021.

[10] P. D. Ryck, L. Desmet, W. Joosen, and F. Piessens, "Automatic and Precise Client-Side Protection against CSRF Attacks," in *European Symposium on Research in Computer Security (ESORICS)*, 2011.

[11] M. West, "Same-site Cookies," 2016. [Online]. Available: https://tools.ietf.org/html/draft-west-first-party-cookies-07

[12] S. Helme, "CSRF is (really) dead," https://scotthelme.co.uk/csrf-is-really-dead/.

[13] S. Rees-Carter, "CSRF is dead, long live SameSite=Lax (or is it?)," https://stephenreescarter.net/csrf-is-dead-long-live-samesite-lax/.

[14] Using the Same-Site Cookie Attribute to Prevent CSRF Attacks. https://www.netsparker.com/blog/web-security/same-site-cookie-attribute-prevent-cross-site-request-forgery/.

[15] R. Sharma, "Preventing Cross-Site Attacks Using SameSite Cookies," https://dropbox.tech/security/preventing-cross-site-attacks-using-same-site-cookies.

[16] P. K. Riramar. OWASP: SameSite Attribute. https://owasp.org/www-community/SameSite.

[17] M. West, "Incrementally Better Cookies," 2019. [Online]. Available: https://tools.ietf.org/html/draft-west-cookie-incrementalism-00

[18] S. Calzavara, T. Urban, D. Tatang, M. Steffens, and B. Stock, "Reining in the Web's Inconsistencies with Site Policy," in *Network and Distributed Systems Security Symposium*, 2021.

[19] A. Mendoza, P. Chinprutthiwong, and G. Gu, "Uncovering HTTP Header Inconsistencies and the Impact on Desktop/Mobile Websites," in *World Wide Web Conference*, 2018.

[20] "Cookies: HTTP State Management Mechanism," 2020. [Online]. Available: https://tools.ietf.org/html/draft-ietf-httpbis-rfc6265bis-05

[21] Django HttpResponse.set_cookie() API. https://docs.djangoproject.com/en/3.1/ref/request-response/.

[22] Pyramid Response.set_cookie() API. https://docs.pylonsproject.org/projects/pyramid/en/latest/api/response.html.

[23] A. Barth, C. Jackson, and J. C. Mitchell, "Robust Defenses for Cross-site Request Forgery," in *ACM Conference on Computer and Communications Security*, 2008.

[24] A. Czeskis, A. Moshchuk, T. Kohno, and H. J. Wang, "Lightweight Server Support for Browser-based CSRF Protection," in *International Conference on World Wide Web*, 2013.

[25] Account Take Over in US Dept of Defense. https://hackerone.com/reports/410099.

[26] Critical CSRF Vulnerability on Facebook. https://www.acunetix.com/blog/web-security-zone/critical-csrf-vulnerability-facebook/.

[27] WordPress CVE-2014-9033. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-9033.

[28] W. Zeller and E. W. Felten, "Cross-Site Request Forgeries: Exploitation and Prevention," in *Princeton University*, 2008, https://www.cs.utexas.edu/~shmat/courses/cs378/zeller.pdf.

[29] M. Cardwell, "Abusing HTTP Status Codes to Expose Private Information," 2011, https://www.grepular.com/Abusing_HTTP_Status_Codes_to_Expose_Private_Information.

[30] T. Yoneuchi, "Detect the Same-Origin Redirection with a bug in Firefox's CSP Implementation," 2018, https://diary.shift-js.info/csp-fingerprinting/.

[31] R. Linus, "Your Social Media Fingerprint," https://github.com/RobinLinus/socialmedia-leak.

[32] T. Van Goethem, W. Joosen, and N. Nikiforakis, "The Clock is Still Ticking: Timing Attacks in the Modern Web," in *ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[33] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow," in *IEEE Symposium on Security and Privacy*, 2010.

[34] A. Janc and M. West, "How do we Stop Spilling the Beans Across Origins," 2018. [Online]. Available: https://www.arturjanc.com/cross-origin-infoleaks.pdf

[35] J. Mao, Y. Chen, F. Shi, Y. Jia, and Z. Liang, "Toward Exposing Timing-Based Probing Attacks in Web Applications," in *International Conference on Wireless Algorithms, Systems, and Applications*, 2016.

[36] Z. Weinberg, E. Y. Chen, P. R. Jayaraman, and C. Jackson, "I Still Know What You Visited Last Summer: Leaking Browsing History via User Interaction and Side Channel Attacks," in *IEEE Symposium on Security and Privacy*, 2011.

[37] SameSite Cookies. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite.

[38] Safe HTTP Methods. https://developer.mozilla.org/en-US/docs/Glossary/safe.

[39] (2020) Feature: Cookies default to SameSite=Lax. https://www.chromestatus.com/feature/5088147346030592.

[40] Site compatibility-impacting changes coming to Microsoft Edge. https://docs.microsoft.com/en-us/microsoft-edge/web-platform/site-impacting-changes.

[41] Can I use SameSite cookie attribute? https://caniuse.com/?search=samesite.

[42] (2019) Intent to implement: Cookie SameSite=lax by default and SameSite=none only if secure. https://groups.google.com/forum/#!msg/mozilla.dev.platform/nx2uP0CzA9k/BNVPWDHsAQAJ.

[43] The Chromium Projects: SameSite Updates. https://www.chromium.org/updates/same-site.

[44] P. Calvano, "SameSite Cookies - Are you Ready?" 2020, https://dev.to/httparchive/samesite-cookies-are-you-ready-5abd.

[45] Impact of the Changes to the SameSite Cookie Flag Default Behavior in Chrome. https://wiki.resolution.de/doc/saml-sso/latest/all/knowledgebase-articles/technical/impact-of-the-changes-to-the-samesite-cookie-flag-default-behavior-in-chrome.

[46] F. Skokan, "Upcoming Browser Behavior Changes: What Developers Need to Know," 2020, https://auth0.com/blog/browser-behavior-changes-what-developers-need-to-know/.

[47] B. Geesink, "Default cookie SameSite attribute behaviour change," 2020, https://wiki.surfnet.nl/display/surfconextdev/Default+cookie+SameSite+attribute+behaviour+change.

[48] PhenixID: SameSite cookie patch. https://document.phenixid.net/m/87804/l/1201413-samesite-cookie-patch.

[49] J. Dixon and M. Paine, "Upcoming SameSite cookie changes and the impact for APEX Apps running in an iframe," 2020, https://www.jmjcloud.com/blog/upcoming-samesite-cookie-changes-and-the-impact-for-apex-apps-running-in-an-iframe.

[50] Bypass SameSite Cookies Default to Lax and get CSRF. https://medium.com/@renwa/bypass-samesite-cookies-default-to-lax-and-get-csrf-343ba09b9f2b.

[51] Defending against CSRF with SameSite cookies. https://portswigger.net/web-security/csrf/samesite-cookies.

[52] J. Rabal, "Same-Site cookies against CSRF attacks analysis," 2017, https://www.tarlogic.com/en/blog/samesite-cookies-analysis/.

[53] Puppeteer. https://github.com/puppeteer/puppeteer.

[54] Chrome DevTools Protocol. https://chromedevtools.github.io/devtools-protocol/.

[55] Chrome DevTools Protocol Audits. https://chromedevtools.github.io/devtools-protocol/tot/Audits/.

[56] Web Shrinker API. https://www.webshrinker.com/.

[57] EasyList. https://easylist.to/.

[58] Host BlackList. https://github.com/anudeepND/blacklist.

[59] Host BlockList. https://github.com/notracking/hosts-blocklists.

[60] G. Franken, T. Van Goethem, and W. Joosen, "Who Left Open

the Cookie Jar? A Comprehensive Evaluation of Third-party Cookie Policies," in *27th USENIX Security Symposium*, 2018.
[61] G. Franken, T. Van Goethem, and W. Joosen, "Exposing Cookie Policy Flaws Through an Extensive Evaluation of Browsers and Their Extensions," in *IEEE Symposium on Security and Privacy*, 2019.
[62] K. Drakonakis, S. Ioannidis, and J. Polakis, "The Cookie Hunter: Automated Black-box Auditing for Web Authentication and Authorization Flaws," in *ACM SIGSAC Conference on Computer and Communications Security*, 2020.
[63] T. Van Goethem, V. Le Pochat, and W. Joosen, "Mobile Friendly or Attacker Friendly? A Large-Scale Security Evaluation of Mobile-First Websites," in *ACM Asia Conference on Computer and Communications Security*, 2019.
[64] StackExchange Security Community. https://security.stackexchange.com/.
[65] Dev Security Community. https://dev.to/t/security.
[66] Issue 831725: SameSite cookie bypass via prerender. https://bugs.chromium.org/p/chromium/issues/detail?id=831725.
[67] Cookies with SameSite=None or SameSite=invalid treated as Strict. https://bugs.webkit.org/show_bug.cgi?id=198181.
[68] Mozilla CVE-2018-12370. https://www.cvedetails.com/cve/CVE-2018-12370/.
[69] CVE-2018-18351. https://nvd.nist.gov/vuln/detail/CVE-2018-18351.
[70] CVE-2019-5880: SameSite cookie bypass. https://bugzilla.redhat.com/show_bug.cgi?id=1762378.
[71] SameSite cookies aren't sent on credentialed CORS requests. https://github.com/whatwg/fetch/issues/769.
[72] C. Sabol, "It's Okay, We're All On the SameSite," 2020, https://securityboulevard.com/2020/02/its-okay-were-all-on-the-samesite/.
[73] S. Rees-Carter. SameSite Cookies Deep Dive / CSRF is dead (or is it?). https://stephenreescarter.net/talks/samesite-cookies/.
[74] V. Li, "Bypassing CSRF Protection," 2020, https://vickieli.dev/csrf/bypass-csrf-protection/.
[75] SameSite cookies. https://makandracards.com/makandra/71018-samesite-cookies.
[76] J. Walton, "Avoiding CSRF Attacks with API Design," 2020, http://www.thedreaming.org/2020/05/26/avoid-csrf-attacks-with-api-design/.
[77] SameSite Cookies and CSRF Attacks. https://symfonycasts.com/screencast/api-platform-security/samesite-csrf.
[78] XS-Leaks Wiki: SameSite Cookies. https://xsleaks.dev/docs/defenses/opt-in/same-site-cookies/.
[79] SameSite Cookie Attribute and Synchronizer Token Pattern. https://security.stackexchange.com/questions/201396/samesite-cookie-attribute-and-synchronizer-token-pattern.
[80] How is the lack of the "SameSite" cookie flag a risk? https://security.stackexchange.com/questions/154106/how-is-the-lack-of-the-samesite-cookie-flag-a-risk.
[81] Will same-site cookies be sufficent protection against CSRF and XSS? https://security.stackexchange.com/questions/121971/will-same-site-cookies-be-sufficent-protection-against-csrf-and-xss.
[82] D. Jubeau, "Secure your cookies to the next level with SameSite attribute," 2017, https://dev.to/damienjubeau/secure-your-cookies-to-the-next-level-with-samesite-attribute.
[83] F. Valverde, "Everybody hates CSRF," 2020, https://dev.to/fdoxyz/everybody-hates-csrf-4fek.
[84] Feature: Reject insecure SameSite=None cookies. https://www.chromestatus.com/feature/5633521622188032.
[85] R. Masas. (2018) Patched Facebook Vulnerability Could Have Exposed Private Information About You and Your Friends. [Online]. Available: https://www.imperva.com/blog/facebook-privacy-bug/
[86] C. Guan, K. Sun, Z. Wang, and W. Zhu, "Privacy Breach by Exploiting postMessage in HTML5: Identification, Evaluation, and Countermeasure," in *ACM Asia Conference on Computer and Communications Security*, 2016.
[87] A. Ballarano, F. Colace, M. De Santo, and L. Greco, "The Postman Always Rings Twice": Evaluating E-Learning Platform a Decade Later," *International Journal of Emerging Technologies in Learning*, 2016.
[88] SameSite Frequently Asked Questions (FAQ). https://www.chromium.org/updates/same-site/faq.
[89] SameSite Cookie Attribute explained. https://cookie-script.com/documentation/samesite-cookie-attribute-explained.
[90] R. Merewood, "SameSite cookie recipes," https://web.dev/samesite-cookie-recipes/.
[91] (2018) Client-side CSRF. https://www.facebook.com/notes/facebook-bug-bounty/client-side-csrf/2056804174333798/.
[92] L. Caretton, "Node.js Connect CSRF Bypass Abusing Method Override Middleware," http://blog.nibblesec.org/2014/05/nodejs-connect-csrf-bypass-abusing.html.
[93] Often Misused: HTTP Method Override. https://vulncat.fortify.com/en/detail?id=desc.dynamic.xtended_preview.often_misused_http_method_override.
[94] Mitch Dataset. https://github.com/alviser/mitch/tree/master/dataset.
[95] Update or reinstall Safari for your computer. https://support.apple.com/en-us/HT204416.
[96] Flask Response.set_cookie() API. https://tedboy.github.io/flask/generated/generated/flask.Response.set_cookie.html.
[97] Tornado RequestHandler.set_cookie() API. https://www.tornadoweb.org/en/stable/web.html?highlight=set_cookie#tornado.web.RequestHandler.set_cookie.
[98] Web.py setcookie() API. https://webpy.org/cookbook/cookies.
[99] Express SameSite Cookie Attribute. https://expressjs.com/en/resources/middleware/session.html.
[100] Meteor Cookie.set() API. https://docs.meteor.com/.
[101] Sails Response.cookie() API. https://sailsjs.com/documentation/reference/response-res/res-cookie.
[102] Sails SameSite Cookies. https://github.com/balderdashy/sails/issues/6942.
[103] Koa SameSite Attribute. https://github.com/koajs/session/issues/174.
[104] Hapi Server.state() API and isSameSite Option. https://hapi.dev/api/?v=20.1.0.
[105] Laravel SameSite Cookie Attribute. https://laracasts.com/discuss/channels/laravel/some-cookies-are-misusing-the-recommended-samesite-attribute.
[106] Symfony Cookie API. https://symfony.com/doc/current/components/http_foundation.html#setting-cookies.
[107] Symfony Default SameSite Cookie Attribute. https://github.com/symfony/symfony/blob/c377a795f579e5417d106c94ae5d5fe4b4300dca/src/Symfony/Component/HttpFoundation/Cookie.php.
[108] CakePHP withCookie() API. https://book.cakephp.org/4/en/controllers/request-response.html#setting-cookies.
[109] CakePHP Default SameSite Cookie Attribute. https://github.com/cakephp/cakephp/blob/d4b68a6dd2404d0b8cc7431838a39ec44b3f5f6b/src/Http/Cookie/Cookie.php.
[110] Zend Default SameSite Cookie Attribute. https://github.com/zendframework/zend-http/commit/0d99103d391f4f746e267a00507d753660550f7b.
[111] Slim setCookie() API. https://www.slimframework.com/docs/v2/response/cookies.html.
[112] Slim SameSite Cookie Attribute. https://github.com/bryanjhv/slim-session/issues/54.
[113] ASP.NET WebForms HttpCookie. https://docs.microsoft.com/en-us/aspnet/samesite/csharpwebforms.
[114] ASP.NET HttpCookie. https://docs.microsoft.com/en-us/aspnet/samesite/csmvc.
[115] ASP.NET Core CookieBuilder. https://docs.microsoft.com/en-us/aspnet/core/security/samesite?view=aspnetcore-5.0.
[116] Nancy Web Framework. https://github.com/NancyFx/Nancy.
[117] Service Stack UseSameSiteCookies Configuration. https://docs.servicestack.net/sessions.
[118] Spring SameSite Cookie Attribute. https://docs.spring.io/spring-session/docs/current/reference/html5/guides/java-custom-cookie.html.
[119] Play Cookie.builder() API. https://www.playframework.com/documentation/2.8.x/Migration26#SameSite-attribute,-enabled-for-session-and-flash.
[120] Vaadin Cookie.SetCookie() API. https://vaadin.com/docs/v8/framework/articles/SettingAndReadingCookies.
[121] Vert.X-Web setCookieSameSite API. https://github.com/vert-x3/vertx-web/blob/f7902ccd4f5da70908a68611119d77ef4aa3f8d4/vertx-web/src/main/java/io/vertx/ext/web/handler/SessionHandler.java.
[122] Spark Response.cookie() API. https://sparkjava.com/documentation#getting-started.
[123] GitHub 2020 Octoverse Report. https://octoverse.github.com/.
[124] Stackoverflow Tags. https://stackoverflow.com/help/tagging.
[125] B. Stock, G. Pellegrino, C. Rossow, M. Johns, and M. Backes, "Hey, you have a problem: On the feasibility of large-scale web vulnerability notification," in *USENIX Security Symposium*, 2016.
[126] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, and T. Wan, "Cookies

Lack Integrity: Real-World Implications," in *USENIX Security Symposium*, 2015.
[127] A. Bortz, A. Barth, and A. Czeskis, "Origin Cookies: Session Integrity for Web Applications," in *ACM Transactions on Internet Technology (TOIT)*, 2012.
[128] N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen, "SessionShield: Lightweight protection against session hijacking," in *International Symposium on Engineering Secure Software and Systems*, 2011.
[129] M. Bugliesi, S. Calzavara, R. Focardi, and W. Khan, "CookiExt: Patching the browser against session hijacking attacks," in *Journal of Computer Security*, 2015.
[130] S. Calzavara, A. Rabitti, and M. Bugliesi, "Sub-session hijacking on the web: Root causes and prevention," 2019.
[131] F. Roesner, T. Kohno, and D. Wetherall, "Detecting and defending against third-party tracking on the web," in *9th USENIX Symposium on Networked Systems Design and Implementation*, 2012.
[132] M. Dhawan, C. Kreibich, and N. Weaver, "Priv3: A third party cookie policy," in *W3C Workshop: Do Not Track and Beyond*, 2012.
[133] F. Kerschbaum, "Simple Cross-site Attack Prevention," in *Third International Conference on Security and Privacy in Communications Networks*, 2007.
[134] P. D. Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen, "CsFire: Transparent Client-Side Mitigation of Malicious Cross-Domain Requests," in *International Symposium on Engineering Secure Software and Systems*, 2010.
[135] Z. Mao, N. Li, and I. Molloy, "Defeating Cross-Site Request Forgery Attacks with Browser-Enforced Authenticity Protection," in *13th International Conference on Financial Cryptography and Data Security*, 2009.
[136] S. Lekies, W. Tighzert, and M. Johns, "Towards Stateless, Client-side Driven Cross-site Request Forgery Protection for Web Applications," *SAP Research*, 2012.
[137] S. Sivakorn, I. Polakis, and A. D. Keromytis, "The Cracked Cookie Jar: HTTP Cookie Hijacking and the Exposure of Private Information," in *IEEE Symposium on Security and Privacy*, 2016.
[138] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee, "On the Incoherencies in Web Browser Access Control Policies," in *IEEE Symposium on Security and Privacy*, 2010.
[139] Y. Zhou and D. Evans, "Why aren't HTTP-only cookies more widely deployed," in *Proceedings of 4th Web 2.0 Security and Privacy Workshop*, 2010.
[140] P. Calvano, "SameSite Cookies Analysis," 2020, https://discuss.httparchive.org/t/samesite-cookies-analysis/1988.
[141] SameSite Strict Usage Statistics. https://trends.builtwith.com/docinfo/SameSite-Strict.
[142] J. Schwenk, M. Niemietz, and C. Mainka, "Same-Origin Policy: Evaluation in Modern Browsers," in *Proceedings of the 26th USENIX Conference on Security Symposium*, 2017.
[143] G. Aggarwal, E. Bursztein, C. Jackson, and D. Boneh, "An Analysis of Private Browsing Modes in Modern Browsers." in *USENIX security symposium*, 2010.
[144] X. Likaj, S. Khodayari, and G. Pellegrino, "Where We Stand (or Fall): An Analysis of CSRF Defenses in Web Frameworks," in *24th International Symposium on Research in Attacks, Intrusions and Defenses*, 2021.
[145] A. Parsovs, "Practical Issues with TLS Client Certificate Authentication." in *Network and Distributed Systems Security Symposium*, 2014.
[146] Hackerone. https://hackerone.com.
[147] Bugcrowd. https://www.bugcrowd.com.

APPENDIX A
ADDITIONAL DETAILS FOR THREATS

Window Properties and postMessage Leak. In a window properties XS-Leak, the attacker can issue top-level navigation requests via, for example, `w=window.open()`, and then count the number of frames in a webpage (note that `w.frames.length` is the leaking channel). For example, if the victim is logged in, the attacker will count x frames, and zero otherwise. Top-level navigation requests are not covered by the Lax policy, meaning that top-level requests will include cookies, leaving the leaking channel observable by the attacker. On the contrary, if developers set the SameSite policy to strict, this XS-Leak is mitigated. For the postMessage leak, the attack pattern is similar. The only difference is that the observable leaking channel is no longer the number of frames but the attacker is listening for broadcasted postMessages. For instance, if the victim is logged in, a postMessage m is observed, and no messages otherwise. Also, in this case, should the cookie not be included in top-level navigation requests, the attacker would not be able to observe differences across user states (e.g., logged in vs logged out) [2].

Pervasive Monitoring. Assume a website $W1$ that set a privacy-sensitive cookie with `SameSite=None` and another website $W2$ that performs cross-site requests to $W1$. Because of the policy set by $W1$, browsers will include cookies in all requests from $W2$ to $W1$. This is the typical setting of third-party cookies widely used for tracking users. Pervasive (network) monitoring is a threat to these scenarios because if cookies are not securely transported (i.e., over TLS), they can reveal sensitive information about user identity. For this reason, browsers like Chrome and Opera reject cookies that do not set the `Secure` flag together with `SameSite=None` policy [84]. However, other browsers such as Firefox and Safari do not reject these cookies (see Table X), exposing users of these websites to pervasive monitoring attacks.

Cookie-less Request Authentication. While cookies are one of the most prevalent forms of request authentication, they are not the only one (see, e.g., [144]). SameSite cookies can protect those class of request forgery attacks that perform ambient HTTP request authentication with cookies. Accordingly, other forms of request authentication, such as HTTP authentication, client certificate authentication [145], or network-based authentication are not protected by SameSite cookies.

APPENDIX B
ADDITIONAL EVALUATION DETAILS

User-Agents for Web Crawls. To determine inconsistent SameSite cookie policies across mobile and desktop deployments of a website, we performed three large-scale crawls on Alexa top 500K websites using two different user-agents (i.e., Chrome Desktop and iOS Safari) on June 2020, September 2020 and April 2021, as per methodology of §VI-B8. The exact user-agent strings used by our crawler are detailed below.

- Chrome Desktop: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/84.0.4147.89 Safari/537.36.
- iOS Safari: Mozilla/5.0 (iPhone; CPU iPhone OS 9_1 like Mac OS X) AppleWebKit/601.1.46 (KHTML, like Gecko) Version/9.0 Mobile/13B137 Safari/601.1.

Request Timeout Threshold for Internet Archive. We submitted queries to the Internet Archive to collect the HTTP response headers of the top 500K Alexa sites from June 2019 to March 2021 (eight crawls, each separated by a three-month

IAB ID	Category	Sub-categories	# Requests	# Cookies	# Websites
IAB19	Technology & Computing	File Sharing, Web Search, Email / Chat / Messaging, Data Centers, Desktop Publishing	10,026	37,612	188
IAB3	Business	Advertising, Marketing, Business Software	6,354	19,917	186
IAB12	News / Weather / Information	News / Weather / Information	1,969	10,795	32
IAB9	Hobbies & Interests	Video & Computer Games, Freelance Writing / Getting Published, Photography	1,365	10,508	34
IAB5	Education	Distance Learning	1,120	10,705	23
IAB1	Arts & Entertainment	Books & Literature, Movies, Music & Audio, Television & Video	867	4,341	25
IAB4	Careers	Job Search	399	2,005	21
IAB6	Family & Parenting	Babies & Toddlers	351	1,918	18
IAB24	Uncategorized	Uncategorized	303	1,025	17
IAB21	Real Estate	Buying / Selling Homes	62	489	10
IAB22	Shopping	Content Server, Streaming Media, Adult Content, Contests & Freebies	57	411	17
IAB14	Society	Social Networking, Weddings	46	283	13
IAB18	Fashion	Jewelry, Clothing	40	142	16
IAB11	Law, Government, & Politics	Politics	12	55	8
IAB13	Personal Finance	Credit / Debit & Loans	9	22	5
IAB2	Automotive	Buying/Selling Cars	8	14	4
IAB7	Health & Fitness	Exercise / Weight Loss	4	9	3
Total	16	32	22,992	89,743	211

TABLE XII: Overview of the categorization of the affected cross-site requests and types of third-party functionalities.

Invalid Policies	# Websites
SameSite=secure	287
SameSite=1	245
SameSite=true	138
SameSite=undefined	124
SameSite=;	106
SameSite=	72
SameSite=false	68
SameSite=-1	55
SameSite:Lax	53
SameSite=0	40

TABLE XIII: Top ten invalid SameSite cookie policies in Alexa top 500K sites.

IdP	Vuln.	# Websites
Google	●	3,450
Amazon	○	679
Facebook	●	3,328
Apple	○	1,593
Microsoft	●	1,921
Linkedin	●	983
GitHub	●	198
Twitter	○	2,591
VK	●	1,241
Mail.ru	○	49
Twitch	○	168
Yahoo	○	379
Instagram	○	1,485
Total Vuln.	6	4,935
Total	13	9,485

Legend: ● = vuln. ; ○ = not vuln.

TABLE XIV: Overview of the IdPs that enable bypass of the new default SameSite cookie policy and the number of affected websites.

gap). To scale up our analysis, we conservatively set a 45 seconds connection timeout for each request.

Chrome on iOS and Lax-by-default. iOS/iPadOS browsers are required to use WebKit for rendering web pages, possibly limiting browser developers' liberty in changing the default SameSite cookies handling. However, even when browsers are required by the AppStore policy to use iOS WebKit, we observed different behaviors between iOS Safari and iOS Chrome. With `SameSite=Strict`, Safari attaches cookies only for SameSite requests (as per RFC 6265bis specification [20]) whereas iOS Chrome does not do that (Table X). Based on that, we do not know whether Chrome developers have some form of liberty to control browser's behavior when handling SameSite cookies, or if Chrome developers are using a different version of iOS WebKit from the one used by Safari.

Testbed	June 2020	Sept. 2020	April 2021
Alexa Top 500K	5,719	9,215	9,951
Alexa Top 100K	1,903	2,949	3,242
Alexa Top 10K	339	565	645
Alexa Top 1K	64	128	138

TABLE XV: SameSite cookie policy inconsistencies for different user-agents grouped by site popularity.

The New Default and Invalid Policy Values. Lax being the new default policy means that if the HTTP response `Set-Cookie` header does not contain the `SameSite` attribute, browsers are expected to enforce the Lax policy as per RFC 6265bis [20]. However, when this attribute is wrongly set with an invalid value, the selected policy depends on the browser. RFC 6265bis says that browsers should set the policy to None. However, our results show that this does not always happen. For example, Chromium-based browsers and Tor Browser set the policy to Lax (which is the new default policy). Other browsers follow the specification.

SameSite Cookies on Protected Pages. In this paper, we did not evaluate *at scale* pages after the login step accurately. To the best of our knowledge, Cookie Hunter [62] is the only recent approach able to handle the sign-up and sign-in automatically. However, the sign-up success rate is quite unsatisfactory, with 88% fail rate in creating accounts. In addition, Cookie Hunter relies on pattern matching which is too brittle to minor changes in the UIs, requiring creating and maintaining new patterns throughout the longitudinal analysis. For these reasons, we evaluated SameSite adoption on a smaller scale by creating ad-hoc login scripts, focusing on the protected pages on the Top 500 sites having the login functionality (211 sites). We observed that 88% of the sites that do not use SameSite for their cookies on the home page also do not use it on their protected pages.

APPENDIX C
VULNERABILITY NOTIFICATION

The security issues we identified in this paper affects websites (Table III), IdPs (Table XIV), web frameworks (Table XI)

Framework	StackOverflow Questions	GitHub Used By	GitHub Stars	GitHub Forks	GitHub Watches	Weekly Downloads
Flask	38.3k	462k	51k	13.6k	2.3k	2.9m
Django	236k	398k	50.4k	21.8k	2.3k	1.2m
Tornado	3.5k	98.2k	19.2k	5.4k	1.1k	2.7m
Pyramid	2.2k	9.7k	3.7k	865	174	43k
Web2py	2.1k	38	1.9k	849	229	85
Express	65.9k	6.4m	49.1k	8.1k	1.8k	10.7m
Meteor	28.6k	-	41.7k	5.1k	1.7k	-
Sails	6.5k	24.7k	21.4k	1.9k	709	28k
Koa	1.1k	124k	29.5k	2.8k	882	438k
Hapi	523	-	12.5k	1.3k	439	207k
Laravel	37.5k	488k	59.9k	18.8k	4.7k	243k
Symfony	16.5k	59.3k	26k	7.6k	1.3k	24.9k
CakePHP	7.7k	10k	8.2k	3.4k	617	4.2k
Zend	5.1k	6.3k	5.7k	2.8k	542	4.5k
Slim	650	25.1k	10.7k	1.9k	564	10.2k
ASP.NET WebForms	357k	322k	606	290	77	-
ASP.NET MVC	357k	322k	606	290	77	-
ASP.NET Core	48k	10k	18.1k	5.1k	1.5k	-
Nancy	1.1k	-	7.1k	1.5k	452	-
Service Stack	5k	2.1k	4.8k	1.6k	542	-
Spring	171k	162k	38k	25.7k	3.5k	-
Play	16.8k	-	11.6k	3.9k	712	-
Vaadin	5k	10.3k	1.6k	730	151	-
Vert.x-Web	1.9k	16.2k	767	361	87	-
Spark	534	19k	8.8k	1.5k	436	-

TABLE XVI: Popularity of the top five web frameworks of top five programming languages. The list contains a reordered, updated version of [144] on April 2021, and is shortlisted to five frameworks. Weekly download statistics are derived from PIP, NPM, and Packagist for Python, JavaScript, and PHP-based frameworks, respectively.

and browser vendors (Table X). We started the process of notifying the affected parties in February 2021. We examined multiple communication channels until we found a valid point of contact. Specifically, we used the vulnerability disclosure programs on (i) HackerOne [146] and Bugcrowed [147], (ii) contact forums and valid email addresses we found on vulnerable websites themselves, and the website of web frameworks, (iii) WHOIS lookups [125], and finally (iv) Git issues for web frameworks. At the time of writing this paper, the status of our notification campaign is:

- 1,032 XS-Leaks affecting 40 websites: 27 confirmed, of which 16 patched and the rest closed as informative/ acceptable risk, four websites are currently under review, and nine are unresponsive.
- 16 CSRF vulnerabilities affecting nine websites: eight confirmed and patched, and one is unresponsive.
- 14 inter/intra-page SSC inconsistency affecting 10 websites: six confirmed, of which five patched, one under review, and three unresponsive.
- SSO Redirects Bypass affecting six IdPs: all confirmed, of which three patched and the rest considered it as an acceptable risk.
- SameSite=None by default in six web frameworks: all confirmed, of which four patched and two said it is a developer issue.
- Browser vendors: all confirmed, of which one patched (Tor Browser), and the rest is either future work or calculated risk.
- For the remaining two types of vulnerabilities, we need to contact 9,951 websites for the user-agent inconsistency and 2,148 for pervasive monitoring. In April 2021, we sent the first reports to 138 sites vulnerable to user-agent inconsistency and 82 sites vulnerable to pervasive monitoring. To contact the remaining sites, we sought the assistance of the national CSIRT.

APPENDIX D
CASE STUDIES

In this section, we report on a few manually vetted case studies of the confirmed attacks. We note that the affected parties have been promptly informed of the discovered vulnerabilities, and have already patched them (see Appendix C).

Tumblr. We discovered a user deanonymization vulnerability in Tumblr leveraging the cross-domain window properties XS-Leak. This attack vector can be used for targeted attacks, i.e., deanonymizing a single individual, or a group of people across site origins. In particular, assuming the target user has the account with username `u1` in Tumblr, the attacker exploits the state-dependent resource $R =$`/blog/u1/review`. If the user is not logged in, Tumblr returns the login page which has $c = 1$ frame. If the user is logged in but is not the target user, Tumblr redirects to a different webpage, which also has $c = 1$ frame. However, when the user is logged and owns the account `u1`, the user has proper permissions to access the private resource R. In this case, the webpage contains $c = 2$ frames. While Tumblr uses SameSite cookies, attackers can observe the difference in the number of webpage frames leveraging top-level requests, deanonymizing the user.

Meetup. We found a CSRF vulnerability in Meetup that allows adding or removing notification alerts for users. This attack vector affects the endpoint `/api`. To add or remove an alert, Meetup normally sends a POST request to the aforementioned affected endpoint. This request has several parameters in its body, e.g., an anti-forgery token `-csrf-token`, and a `method` parameter specifying the request action, i.e., adding or removing the alert. As shown in Table I, POST request contexts are protected by the new default Lax policy. However, an attacker could bypass this protection by changing the request method to GET. For a CSRF exploitation, the attacker replays the key-value parameters in the POST request body in the form of GET query parameters. For the anti-CSRF parameter, the attacker uses the `-csrf-token=0` for all requests. We observed that Meetup fails to perform the CSRF verification correctly when the method is changed from POST to GET.

Mailchimp. We discovered a CSRF vulnerability that enables a web attacker to create arbitrary surveys in Mailchimp across origins on behalf of a victim user. The vulnerability affects the endpoint `/lists/surveys/create`. For creating a survey, Mailchimp sends a request to the aforementioned endpoint using the POST method. The body of this request contains an anti-CSRF token and other key-value parameters that encode the details of the created survey (e.g., title, questions, etc). We found that an attacker can forge this request by switching the HTTP method to GET, and replaying the POST body parameters in the form of GET query parameters. Similarly to Meetup, Mailchimp does not perform the CSRF verification when the method is changed to GET. Therefore, the anti-CSRF token can be omitted from the forged request.