# Cookie Crumbles: Breaking and Fixing Web Session Integrity

Marco Squarcina
*TU Wien*

Pedro Adão
*Instituto Superior Técnico, ULisboa*
*Instituto de Telecomunicações*

Lorenzo Veronese
*TU Wien*

Matteo Maffei
*TU Wien*

## Abstract

Cookies have a long history of vulnerabilities targeting their confidentiality and integrity. To address these issues, new mechanisms have been proposed and implemented in browsers and server-side applications. Notably, improvements to the Secure attribute and cookie prefixes aim to strengthen cookie integrity against network and same-site attackers, whereas SameSite cookies have been touted as the solution to CSRF. On the server, token-based protections are considered an effective defense for CSRF in the synchronizer token pattern variant. In this paper, we question the effectiveness of these protections and study the real-world security implications of cookie integrity issues, showing how security mechanisms previously considered robust can be bypassed, exposing Web applications to session integrity attacks such as session fixation and cross-origin request forgery (CORF). These flaws are not only implementation-specific bugs but are also caused by compositionality issues of security mechanisms or vulnerabilities in the standard. Our research contributed to 12 CVEs, 27 vulnerability disclosures, and updates to the cookie standard. It comprises (i) a thorough cross-browser evaluation of cookie integrity issues, that results in new attacks originating from implementation or specification inconsistencies, and (ii) a security analysis of the top 13 Web frameworks, exposing session integrity vulnerabilities in 9 of them. We discuss our responsible disclosure and propose practical mitigations.

## 1 Introduction

HTTP cookies are the oldest and most widely used mechanism for state sharing between Web clients and servers. They are a cornerstone of Web sessions and play a crucial role in the authentication and authorization of users. Despite their prominence in Web applications, cookies have a long history of vulnerabilities and several known pitfalls [41,44,64,76,89].

Entire classes of attacks revolve around compromising either the confidentiality or the integrity of cookies [48]. For instance, *session hijacking* attacks aim to leak the value of a session cookie (e.g., via *cross-site scripting*) and use it to obtain unauthorized access to a website [72]. *Session fixation* attacks involve compromising cookie integrity to force an attacker-controlled cookie in the victim's browser, and then impersonate the victim on the target website [61]. *Cross-site request forgery* (CSRF) attacks, instead, are a typical session integrity violation problem where the attacker issues cross-site requests from the victim's browser to execute unwanted actions on a website in which the victim is authenticated [41].

In response to these attacks, new mechanisms have been proposed on both the client and the server side. On the client side, major browsers now support the updated cookie standard RFC6265bis [50] which includes extended security features compared to the original RFC from 2011 [39]. A notable example is the *SameSite* attribute, which has been touted as a robust solution against CSRF attacks [58, 59]. Other changes focused on strengthening cookie integrity against same-site and network attackers, with improvements to the Secure flag and the introduction of __Host- and __Secure- cookie name prefixes [69]. On the server side, traditional protections against CSRF attacks include the usage of a secret token shared between browsers and servers [41]. This approach has been widely adopted by popular Web frameworks and considered an effective defense in the *synchronizer token pattern* variant [62, 71].

In this paper, we question the effectiveness of existing protections and study the real-world security implications of cookie integrity issues. In particular, we focus on network and same-site attackers [44], a class of attackers increasingly becoming a significant threat to Web application security [78]. We show how security mechanisms considered to be robust against these threat models can be bypassed, exposing Web applications to session integrity attacks such as session fixation and *cross-origin request forgery* (CORF). We suggest that these vulnerabilities are due to compositionality challenges between Web standards, browsers, and servers, and we propose a set of countermeasures to reconcile these issues. Overall, our research contributed to 12 CVEs, 27 vulnerability disclosures, and updates to the RFC of the cookie standard [50].

We identified novel attack vectors that bypass modern cookie protections and precisely characterize a class of attacks called *CORF token fixation* that highlights weaknesses in current CSRF protections. We performed a systematic security analysis of the top 13 Web frameworks, exposing session integrity vulnerabilities in 9 of them. We showed that these vulnerabilities are not only implementation-specific bugs but are caused by compositionality issues of security mechanisms or flaws in the standard. We also discussed the response of developers to our responsible disclosure and proposed mitigation strategies to improve the security of the Web ecosystem.

**Contributions.** Our contributions are summarized as follows:

- We extend the work of Squarcina et al. [78] to propose a taxonomy of threat models that describes network and same-site attackers in terms of their capabilities and goals (Sec. 3).

- We perform a thorough cross-browser evaluation of known cookie integrity attacks and introduce new attacks classified along 4 different categories: serialization collisions due to nameless cookies, server-side parsing vulnerabilities, cookie jar desynchronization issues, and broken composition of (compliant) parsers. We present our methodology and discuss the result of a measurement study on nameless and prefixed cookies (Sec. 4).

- In Sec. 5, we precisely define the class of CORF token fixation attacks which captures known and novel bypasses of real-world CSRF protections, including the synchronizer token pattern which is considered robust against same-site and network attackers.

- Sec. 6 presents a systematic security analysis of the top 13 Web frameworks, exposing CORF and session fixation vulnerabilities in 9 of them. We discuss the response of developers to our responsible disclosure and propose a set of practical countermeasures to prevent our attacks.

- We formally verify the correctness of our proposed mitigation to the synchronizer token pattern using the ProVerif protocol verifier [42] (Sec. 7).

We publish all artifacts developed during this research, including the browser test suite (Sec. 4.3), the dataset and processing code of our measurement (Sec. 4.4), the ProVerif models and scripts (Sec. 7), as well as the reproducible proof-of-concept attacks against Web frameworks (Sec. 6) [77].

## 2 Background

In the following, we provide an overview of cookie attributes, including existing mechanisms for cookie integrity, and CSRF protections. We first revise standard notions such as *origins* and *sites* being instrumental to the rest of the paper.

## 2.1 Origins and Sites

The *same-origin policy* (SOP) [40] defines the traditional Web security boundary between websites. The SOP is based on the notion of origin, defined as a tuple of scheme, host, and port. For instance, the origin of https://example.com:443 is `<https, example.com, 443>`. The SOP prevents an origin from reading or modifying the contents of a different origin. However, some components of the Web platform have a different scope. Cookies, for instance, are scoped to the *registrable domain* of the website that set them. A registrable domain is a domain name with one label on the left side of an *effective top-level domain*, as defined by the *Public Suffix List* (PSL) [66]. Hosts sharing the same registrable domain are considered to be same-site, e.g., example.com, auth.example.com, and api.staging.example.com all belong to the same site example.com. Same-site hosts are also called *sibling domains*.

In recent years, the definition of same-site evolved to include the URL scheme [84]. Hence, sibling domains with different schemes are considered same-site, but not *schemeful same-site*. To avoid ambiguities, we maintain both terminologies and refer to same-site only when the scheme is irrelevant.

## 2.2 Cookies

Cookies are the main state management mechanism of the Web, allowing servers to maintain a stateful session over the stateless HTTP protocol [50]. Servers can set a cookie in the browser through the `Set-Cookie` header. This cookie is then automatically attached by the browser to all following HTTP requests to the server via the `Cookie` header. Additionally, JavaScript code running in Web pages can access and set the value of cookies using the traditional Document.cookie property or the new Cookie Store API [68].

**Attributes.** Cookies can be configured with attributes, or flags, which specify additional properties or constraints. The `Path` attribute allows to limit the cookie to a set of URL paths, i.e., the browser will include the cookie in HTTP requests if the path of the request URL matches or is a subdirectory of the Path attribute. The `Domain` attribute broadens the scope of a cookie. The value of this attribute can be assigned to any of the parent domains of the origin that sets the cookie, up to the registrable domain. For instance, a server at foo.example.com can set a cookie with `Domain=example.com` to specify that the cookie should be attached to all subdomains of example.com. If the attribute is omitted, the browser will send the cookie only to the host that set it. `HttpOnly` prevents the cookie from being accessed by JavaScript, e.g., via the Document.cookie property. The `Secure` attribute limits the scope of the cookie to secure connections. Browsers must reject the insertion of a cookie from a non-secure origin if the cookie jar already contains a secure cookie with the same name and scope.

**Same-Site Cookies.** The `SameSite` attribute has been introduced in 2016 as a defense in depth protection against CSRF

attacks by confining cookies to same-site requests [86]. In particular, the standard defines three same-site policies: `Strict`, cookies are attached to same-site requests only, i.e., no cookie is attached to cross-site requests; `Lax`, cookies are attached to same-site requests and cross-site top-level navigations, e.g., clicking on a link, using the GET request method; `None`, cookies are attached to all requests, cross-site included. According to the standard, SameSite cookies follow the *schemeful same-site* definition to determine whether a request is cross-site. This is in contrast to Domain cookies which do not consider the URL scheme, unless used in combination with the Secure attribute. SameSite cookies also represent one of the most effective protection against XS-Leaks, an emerging class of attacks that exploits gaps in the *same-origin policy* (SOP) to infer information such as PII and the authentication status of a user from a cross-site position [55, 73, 79, 81]. The SameSite attribute restricts the ability to initiate authenticated requests to same-site attackers, thus preventing traditional Web attackers from leaking the user's state on a website.

**Cookie Prefixes.** Cookie prefixes, originally introduced in 2015 [85], enable additional security constraints on cookies based on their name. The specification defines two prefixes: when a cookie name begins with `__Secure-`, the cookie must be set with the `Secure` attribute and from a page served over HTTPS; when the name of a cookie starts with `__Host-`, in addition to all restrictions of the `__Secure-` attribute, the `Path` attribute must be explicitly set to `/`, and it must not contain the `Domain` attribute, locking the scope of the cookie to the host that created it. These additional constraints guarantee the integrity of __Host- cookies against same-site attackers, as such cookies are unaffected by shadowing attacks performed from a same-site position (see Sec. 4).

## 2.3 CSRF Protections

CSRF attacks are a well-known class of attacks where the adversary executes unauthorized state-changing actions under the victim's authenticated session. A CSRF attack is always preceded by a setup phase where the attacker prepares a malicious website that silently performs a cross-site request to the target website to execute the unauthorized action, e.g., via an automatic form submission or the fetch API.

Over the years, many types of CSRF defenses have been proposed in the literature, including (i) origin/referrer checks, (ii) token-based mechanisms to ensure request unguessability, (iii) the `SameSite` cookie attribute, and (iv) explicit user interaction such as CAPTCHAs [41, 62]. All these protections have some limitations and drawbacks. For instance, Same-Site cookies are not effective against attacks performed from a same-site position. To avoid ambiguity, we use the term *Cross-Origin Request Forgery* (CORF) in the rest of the paper, as it includes the attack scenario of a network or same-site attacker. We focus our analysis on token-based protection techniques as they are the most common defense adopted by

Web frameworks [62], and – as shown in Sec. 7 – can offer robust protection if correctly implemented. The main idea is to send an unguessable parameter $t$, commonly named CSRF token, with every state-changing request, typically as a hidden input field in a form. By ensuring that $t$ remains secret to the attacker, cross-origin forged requests will be discarded by the target website, as the token $t$ is missing. Below, we discuss the two most popular token-based protection patterns [62, 71].

**Synchronizer Token Pattern (STP).** In STP, the server generates CSRF tokens and inserts them in every webpage that may lead to a state-changing operation, e.g., as a hidden field in a form for transferring funds. This token is then bound to the user's session and the server validates newly received tokens by verifying the correctness of this binding. Multiple implementations (see Sec. 6) generate a fixed CSRF secret $s$ per session, and use it to derive CSRF tokens $t(s)$. Other implementations generate a fresh CSRF secret $s$ per request, and derive CSRF tokens $t(s)$ similarly to the previous case. In this pattern, secrets are always linked to the user session, irrespective of whether it is stateful or stateless. In the former case, secrets are stored in the server session, whereas in the latter, client-side storage mechanisms, e.g., cookies, are used to synchronize the secret between the server and the browser.

**Double Submit Pattern (DSP).** In this pattern, the CSRF token is a random value stored in a cookie other than the session cookie. The server typically renders the CSRF token in the HTML page as a hidden input field, and the browser sends it back to the server as part of the authenticated request. The server then verifies the validity of the request by checking the equivalence between the cookie value and the CSRF token. This makes DSP more suitable for stateless sessions, as it does not require the server to store the CSRF secrets or tokens in the session. Notice that CSRF cookies can be encrypted or signed with a fixed key or secret stored on the server. In this case, the server-side validation should account for an additional decryption or validation step before performing the comparison. Additionally, servers could store a CSRF secret in a cookie and use it to derive the CSRF token: whenever the CSRF secret or token is not cryptographically bound to the current session identifier, we still refer to this pattern as DSP.

## 3 Threat Model

In this paper, we aim to investigate the security risks that arise from the interaction between a website and a victim's browser when a network or a same-site attacker can forge cookies scoped to the target website. As shown in recent works, these two threat models are still relevant today. According to Zheng et al. [90], only 0.13% of the top 1M websites in 2015 were protected from network attackers thanks to full HSTS deployment. The situation improved in 2022, although 90% of websites remain vulnerable [46]. Large-scale studies on subdomain takeover vulnerabilities demonstrated the impact of

| Capability | Description |
|---|---|
| `headers` | Control arbitrary HTTP response headers at $w_a$. |
| `js` | Execute arbitrary JavaScript on a page at $w_a$. |
| `https` | The scheme of $w_a$ is `https`. |

Table 1: Capabilities required to set cookies in the victim's browser from a sibling domain of the target ($w_a$).

same-site attackers. In 2016, Liu et al. [63] identified 227 of the Alexa top-10K sites affected by vulnerable subdomains. Borgolte et al. [43] studied deprovisioned cloud instances, finding 700K vulnerable domains. Squarcina et al. [78] estimated 13K potentially vulnerable domains due to deprovisioned cloud instances and discovered 887 sites with other subdomain takeover vulnerabilities among the top 50K sites in the Tranco list. They also discussed the dangers posed by corporate networks, roaming services, and dynamic DNS providers, which put users in a same-site position without carrying out attacks.

We consider a range of threat models corresponding to different levels of control and visibility that an attacker may have over the network and sibling domains of the website. To exclude trivially vulnerable scenarios, we assume that the victim accesses the target website over a correctly-configured secure channel. We do not discuss specific attack vectors that can be exploited to acquire a certain position since they have been extensively covered in the past [43, 53, 63, 78]. We focus, instead, on the capabilities of standard threat models that are relevant to violations of cookie integrity. To do so, we build on the framework introduced by Squarcina et al. [78]. Table 1 outlines the capabilities that are relevant to set cookies, assuming a target website $w$, the set of its sibling domains $S_w$, a website controlled by an attacker $w_a \in S_w$, and the victim's browser $B$. Different combinations of these capabilities enable precise characterization of the threat models considered in this work, as shown in Fig. 1.

**SS-HOST-S** maps to a *same-site attacker*, also called *related-domain attacker*, with full control over a sibling domain of the target with a valid TLS certificate. This attacker can render arbitrary content over a secure channel, having the full set of capabilities `https`, `js`, and `headers`.
**SS-HOST-I** is similar to **SS-HOST-S**, excluding the ability to host pages over a secure channel. This threat model captures the case where an attacker controls a sibling domain of the target but cannot obtain a valid TLS certificate, e.g., due to the presence of a CAA DNS record defining a strict allow-list of permitted CAs [78]. The capabilities are `js` and `headers`.
**SS-XSS-S** is a same-site attacker obtaining indirect control over a sibling domain via a script injection vulnerability (XSS) on a page served via HTTPS. Since the attacker is not in control of the response headers returned by the page, the capabilities are `https` and `js`.
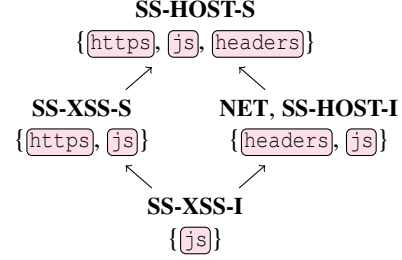


Figure 1: Taxonomy of threat models for cookie integrity violations.

**SS-XSS-I** is an attacker with an XSS vulnerability on a sibling domain served over an insecure connection. The only available capability is `js`.
**NET** maps to a standard network attacker who can fully control cleartext traffic generated by the victim's browser. This attacker is able to intercept, modify, and inject network traffic of any sibling domain of the target domain, including the target domain itself. These capabilities translate into the `headers` and `js`, similarly to the **SS-HOST-I** attacker. Notice that network attackers cannot manipulate cleartext network traffic if the domain enforces a strict HSTS policy that includes the `includeSubDomains` directive [90].

We also formulate a precise definition of *cookie integrity violations*, taking into account the cookie's intended recipient. We assume that the attacker aims to compromise a cookie $c = \langle n, v \rangle$ with name $n$ and value $v$, stored in the victim's browser $B$ for the origin $o$. In a *server-side integrity violation*, the attacker implants a cookie $c' = \langle n', v' \rangle$ in the victim's browser $B$ with the goal of forcing the browser $B$ to send $c'$ to $o$. The server at $o$ parses the `Cookie` header obtaining a cookie with name $n$ but tampered value $v' \neq v$. We refer to a *client-side integrity violation* when the attacker causes the JavaScript Document.cookie property on $o$ to return a key=value pair where the key corresponds to $n$ and the value is chosen by the attacker. Additionally, we consider cookie eviction attacks as integrity violations, i.e., attacks that evict the cookie $c$ from requests to $o$ or remove the cookie from the key=value pairs returned by the Document.cookie API on $o$.

## 4 Violationg Cookie Integrity

In this section, we show how attacker capabilities, and therefore the standard threat models discussed in Sec. 3, map to concrete attacks. First, we systematize known cookie integrity pitfalls and evaluate them on the top 3 Web browsers. Then, we introduce a range of novel attacks along 4 attack classes enabled by inconsistencies between servers, browsers, and the cookie specification. We show that these attacks are possible in practice and can be used to break cookie integrity in unprecedented ways. Finally, we discuss the methodology

| Attack | RFC 📕 | Browsers Chrome | Firefox | Safari |
|---|---|---|---|---|
| Tossing (creation date, oldest first) | ❌ | ✔ | ✔ | ⚠ |
| Tossing (insecure over secure cookie) | ❌ | ✔ | ✔ | ⚠ |
| Eviction (cookie jar overflow) | ✅ | ⚠ | ⚠ | ✔ |
| Eviction (__Host- via secure cookies) | ✅ | ⚠ | ⚠ | ✔ |
| Serialization collision (=a=b→a=b) | ✅$^{\geq 04}$ | ⚠ | ⚠ | ✔ |
| Serialization collision (__Host-) | ❌$^{\geq 11}$ | ⚠$^{<104}$ | ⚠$^{<105}$ | ✔ |
| Cookie jar desynchronization | ❌ | ✔ | ⚠ | ✔ |
| Server-side parsing issues | ❌ | — | — | — |
| Parser-chaining | ✅ | — | — | — |

Table 2: Evaluation of cookie integrity attacks against the cookie standard RFC6265bis-11 and browsers: Chrome (v109), Firefox (v109), and Safari (v16.0). ✅ compliant, ❌ violation, ✔ unaffected, ⚠ vulnerable, — does not apply.

adopted to discover these issues and report on a measurement study performed using the HTTP Archive dataset [37].

## 4.1 Weak Integrity

Due to their legacy design, cookies have a long history of integrity issues, as documented in the cookie specification [50]. A comparison of the top 3 browsers on the integrity pitfalls discussed below is included in Table 2 together with the new attacks introduced in this section.

### 4.1.1 Cookie Tossing

Cookies scoped for a target origin $o$ are sorted by standard-compliant browsers by the most-specific matching `Path` attribute, meaning that cookies set with `Path=/foo` are sent before cookies with `Path=/`. When `Path` attributes are equal, cookies are sorted by creation time, i.e., cookies set first are sent before cookies that are set later. Although the standard states that servers should not rely on the order of cookies sent by browsers, most implementations only consider the first occurrence of a cookie name in the `Cookie` header field [90]. Since attributes are not sent along with cookies, duplicated cookies with the same name but different `Path` attributes are indistinguishable to the server [50, §5.7.3].

Attackers can exploit this behavior to violate cookie integrity. For example, consider a Web application at https://site.tld/login/index.php that sets a cookie via the response header `Set-Cookie: sid=good; Path=/`. Assume also an attacker in control of http://atk.site.tld/. The attacker can set a domain cookie for site.tld with name `sid` and value `evil`. By setting a more specific path in the new cookie, the attacker can cause the victim's browser to send the attacker's controlled cookie first, as in Fig. 2. This specific attack is called *cookie tossing*, or *shadowing*.

As mentioned in Sec. 2, __Host- prefixed cookies are considered to be unaffected by shadowing attacks from a
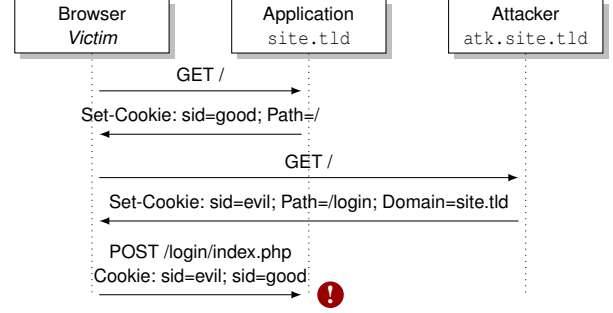


Figure 2: Cookie tossing attack.

same-site position. Furthermore, the standard specifies that secure cookies have strong integrity against non-secure origins. To summarize, cookie tossing requires the `https` capability only for cookies with the `Secure` flag. Otherwise, either the `headers` or the `js` capability is needed.

### 4.1.2 Eviction Techniques

Cookies are evicted from the browser's storage when the storage limit is reached. The eviction policy and precise limits are not specified by the standard, and are left to browser vendors to decide. In practice, recent versions of Firefox and Chrome limit the size of the cookie jar to 180 cookies per schemeful site, while Safari does not enforce any limit. In addition, browsers evict cookies in a least-recently-used (LRU) fashion, i.e., the oldest cookies are evicted first. This is problematic because it allows attackers to control the eviction of cookies by overflowing the cookie jar, and then use cookie tossing to replace the evicted cookies with their own. It is worth mentioning that the `HttpOnly` flag does not provide integrity against an attacker with the `js` capability. Indeed, while HttpOnly cookies cannot be read via JavaScript, they can be evicted by any of the threat models considered in this paper. On the other hand, the `Secure` flag does provide integrity against attackers without the `https` capability, since modern browsers partition cookies by scheme.

## 4.2 Novel Attacks

The cookie standard evolved in recent years to provide stronger integrity guarantees. In particular, the __Host- prefix was proposed in 2015 [85] to prevent cookie tossing attacks. In the following, we present a range of novel cookie integrity attacks that exploit issues in the cookie standard, server and client implementation problems, and the combination of both.
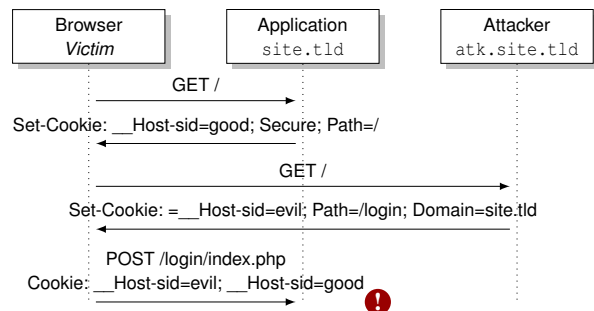
Figure 3: __Host- cookie bypass via nameless cookies.

### 4.2.1 Nameless Cookies and Serialization Collisions

In 2020, a change to the cookie standard[1] added support to *nameless cookies*, i.e., cookies set with empty name and non-empty value. This change was motivated by some servers setting cookies with empty names, and the cookie standard did not specify how to parse them. As a result, the standard now mandates browsers to parse the `Set-Cookie: token` header as a nameless cookie with value `token`. This cookie must be serialized as `Cookie: token`, without any `=` character.

We found that this design introduces a novel attack vector that can bypass even the __Host- prefix. Consider, for instance, a page at site.tld that sets a named cookie `sid=good`. A same-site attacker can set a nameless cookie scoped to site.tld with value `sid=evil`. This can be done via either the Document.cookie property or the HTTP response header `Set-Cookie: =sid=evil; Domain=site.tld`, which is a valid header. According to the standard, the attacker-controlled cookie is serialized as `Cookie: sid=evil`, resulting indistinguishable to the server, or to frontends using the Document.cookie getter, from a cookie named `sid`.

This attack is particularly dangerous because it can violate the integrity guarantees enforced by __Host- cookies. Indeed, any attacker in our taxonomy can shadow a cookie `__Host-<name>=<value>` by forcing in the victim's browser a nameless cookie via `Set-Cookie: =__Host-<name>=<value>; Domain=<domain>`. An example of the attack flow is in Fig. 3.

The same attack vector can shadow arbitrary secure cookies from an insecure origin. As explained in Sec. 2, browsers must reject a cookie set from a non-secure origin if the cookie jar contains a secure cookie matching the name of the new cookie scoped to the same site. Since secure cookies are partitioned differently from insecure ones, the `https` capability is typically required to perform an eviction or a cookie tossing attack against a secure cookie. This attack, however, lowers the preconditions for the integrity violation of secure cookies, requiring only the `headers` or the `js` capability.

**Disclosure.** The attacks above are representative of a larger class of serialization issues that we reported to the IETF HTTP

Working Group on the cookie standard [31] and jointly disclosed the __Host- cookie bypass to the Chrome [29] and Firefox [34] security teams who issued CVE-2022-2860 and CVE-2022-40958, respectively.[2] Chrome fixed the issue in version 104 and Firefox in version 105. Safari is not affected by this vulnerability because it deviates from the standard since it serializes nameless cookies by prefixing the value with `=`. Our contributions and extensive discussion with browser maintainers [30] led to updates to the cookie standard [50, §5.6, point 22] that now mandates browsers to reject nameless cookies with a value starting with a case-insensitive match for `__Host-` or `__Secure-`.

### 4.2.2 Server-Side Parsing Issues

The cookie standard [50, §5.5] describes a set of parsing rules for the `Set-Cookie` header that user agents must follow. Unfortunately, the standard does not clearly specify how servers should parse cookies received via the `Cookie` header. This discrepancy causes server-side cookie integrity violations whenever servers parse two distinct cookies as the same one.

Although the problem is not new per se [90], we discovered a new vulnerability that bypasses __Host- cookies in PHP [35], the server-side language used by 78% of websites [83]. Due to the legacy design derived from `register_globals` [18], PHP replaces spaces, dots, and open square brackets with the underscore symbol `_` in the keys of `$_POST` and `$_GET` superglobal arrays. The same string transformation applies to the keys of the `$_COOKIE` superglobal array. As a result, an attacker can fixate a cookie in the victim's browser via `Set-Cookie: ..Host-sid=evil; Domain=site.tld`, that is parsed by PHP as `Cookie: __Host-sid=evil`. This vulnerability extends integrity concerns to all cookies that contain the underscore symbol, e.g., non-secure origins can use this bug to shadow secure cookies. Similarly, the HTTP server component of the ReactPHP library incorrectly parses the `Cookie` header by url-decoding cookie names [51]. This vulnerability can be exploited to bypass __Host- cookies using percentage-encoded names, e.g., a cookie set via `Set-Cookie: %5F%5FHost-sid=evil; Domain=site.tld` is parsed by ReactPHP as `Cookie: __Host-sid=evil`.

We also discovered a vulnerability in the Werkzeug library, the HTTP middleware used by the popular Flask framework [52]. The `Cookie` header is incorrectly parsed by stripping all leading `=` symbols. To exemplify, a nameless cookie set via `Set-Cookie: ==__Host-sid=evil; Domain=site.tld` is parsed by Werkzeug as a name-value pair corresponding to (`__Host-sid`, `evil`).

All threat models discussed in Sec. 3 can mount these attacks that exploit server-side parsing issues, meaning that

---

[1]RFC6265bis, Accept nameless cookies: https://github.com/httpwg/http-extensions/commit/0178223

[2]The __Host- bypass vulnerability was reported 3 weeks earlier as an independent effort by Axel Chong who is credited on both CVEs. Our issues were merged into the previous vulnerability reports to jointly discuss the mitigation and additional edge cases.

```
1  // Assume an empty cookie jar then set 181 cookies
2  for(let i=1; i<=181; i++)
3      document.cookie = 'a'+i+'=_';
4  // Count the number of cookies
5  document.cookie.split("; ").length
6  > 181 // Higher than the limit of 180 cookies per site
```

Listing 1: Cookie jar overflow desynchronization in Firefox.

only the `headers` or `js` capabilities are required.

**Disclosure.** The PHP vulnerability was assigned CVE-2022-31629 and fixed in PHP 7.4.31, 8.0.24, and 8.1.11. ReactPHP issued CVE-2022-36032 after our report and fixed the vulnerability in version 1.7.0. The Werkzeug vulnerability obtained CVE-2023-23934 and has been patched in version 2.2.3.

### 4.2.3  Cookie Jar Desynchronization

We identified two vulnerabilities in Firefox that cause a desynchronization between the cookies listed by Document.cookie and the actual content of the cookie jar. We experimentally discovered that a cookie jar overflow operated via JavaScript sets more cookies than the maximum number of cookies allowed on a single site. Surprisingly, these cookies can only be retrieved via the Document.cookie API and are not effectively set in the cookie jar, i.e., they are not attached to subsequent HTTP requests [32].

The issue can be easily reproduced using the JavaScript code snippet in Listing 1. This example stores 181 cookies (a1 to a181) in Document.cookie, however, manual inspection of the cookie jar reveals that only 151 cookies are set (a31 to a181). Attempts to clear the cookie jar via the Firefox storage inspector, setting an expiration date in the past via the Set-Cookie header, or using the Clear-Site-Data header [82], fail to remove the first 30 cookies (a1 to a31). This set of cookies survives page reloads and schemeful-same-site navigations. It is also preserved in new schemeful-same-site windows created via the Window.open method. The only way to remove them is to set a past expiration date via JavaScript, or by closing the browser tab.

The described issue can be exploited to violate client-side cookie integrity and requires the `js` capability, with the optional `https` capability if the target website is on a secure origin. Notice also that this inconsistent state could introduce vulnerabilities in applications trusting cookies read from Document.cookie, providing a novel avenue for attacks. For instance, frontends often set custom HTTP headers using the values of specific cookies read via the Document.cookie property. Notable examples are ASP.NET [65] and Angular [36].

The second desynchronization issue happens when there is a secure cookie set by a domain, and a page on a same-site non-secure origin tries to set another cookie with the same name using Document.cookie [33]. We discovered that the insecure cookie is not stored as required by the standard, but it is listed by the Document.cookie property. This inconsistency can create confusion on frontends that rely on Docu-ment.cookie to read cookies. However, the security impact of this second desynchronization issue is limited since it only affects insecure origins that are trivially vulnerable to cookie integrity attacks.

**Disclosure.** We reported both issues to the Firefox security team in June 2022. According to Firefox developers, the root cause of these problems is the composition of cookies' access control policies with Firefox's implementation of Site Isolation, project Fission [67]. The second issue has been fixed in Firefox 112 and obtained CVE-2023-29547, whereas the first one is still under active investigation as of May 2023.

### 4.2.4  Parser Chaining Vulnerabilities

The serialization collision previously discussed introduces a new attack vector against chains of cookie parsers. We investigated the presence of this configuration in real-world applications by studying the AWS API Gateway, a service that acts as a frontend for other AWS services. The AWS Lambda proxy integration for HTTP APIs enables developers to bridge an API route with a Lambda function, passing request payloads to the Lambda function using a JSON message exchange format. According to the documentation [75]: "Format 2.0 includes a new cookies field. All cookie headers in the request are combined with commas and added to the cookies field. In the response to the client, each cookie becomes a set-cookie header."

From our tests, this proxy introduces an additional parser that serializes the cookies in the request payload. As a result, a cookie attached to a request, such as `Cookie: =__Host-sid=evil` corresponding to a nameless cookie with value `=__Host-sid=evil`, is serialized by the AWS Lambda proxy as `{"cookies": ["__Host-sid=evil"], ...}`, resulting indistinguishable from a legitimate cookie named `__Host-sid`. Notice that this specific attack is not prevented by recent Chrome and Firefox mitigations against __Host- cookie collisions, since the cookie value starts with the = symbol.

**Disclosure.** We reported the issue to the AWS security team in October 2022 that deployed a fix in November 2022. The mitigation consists of discarding key-value cookie entries starting with the = symbol followed by a case-insensitive match for `__Host-` or `__Secure-`. This approach, combined with modern browsers that adhere to the latest draft of the cookie standard [50], effectively protects against the threat described in this section.

## 4.3  Discovering Cookie Integrity Issues

The methodology used to discover these attacks consisted of three main stages.

**Browser Testing.** We performed a comprehensive evaluation of known cookie integrity attacks across the top-3 browsers (Chrome, Firefox, and Safari). Inspired by the WPT project [25], we developed a suite of test cases that simulated

various types of attacks and evaluated the behavior of the browsers. The test cases were designed to cover all possible combinations of secure and insecure origins between the victim and a same-site attacker. We also tested different ways to set cookies, i.e., via the Set-Cookie header or using the JavaScript Document.cookie property. The test cases were run on the latest browser versions, and the results were analyzed to identify any inconsistencies between the browsers. Additionally, we used BrowserStack[3] to test all releases from January 2021 to January 2023 of the three major browsers against our test suite and identify any changes in the behavior over time. This phase was crucial to uncover little-known discrepancies between the browsers. For instance, Safari sorts cookies by placing the most recent one first, while Firefox and Chrome serialize cookies starting from the oldest one, as mandated by the specification. We also verified that Safari does not prevent cookie tossing of secure cookies from non-secure cookies, which is a violation of the standard [50]. Additionally, we experimentally verified that Safari does not enforce limits on the maximum number of cookies stored for a single site. Finally, the test suite enabled the automatic discovery of the cookie jar desynchronization issue in Firefox, which was previously unknown to the security community.

**Reviewing the Cookie Standard.** Whenever a discrepancy was found between the browsers, we manually reviewed the cookie standard [50] to determine what was the expected behavior. During this phase, we learned that the standard introduced support to nameless cookies in 2020 and we discovered the serialization collision issues. We engaged with the IETF HTTP Working Group and browser vendors to address the problems as we found them.

**Testing Server-Side Parsers.** As a third stage of the analysis, we investigated the presence of inconsistencies in the cookie parsers of the server-side languages and core HTTP handling libraries used by the frameworks discussed in Sec. 6. For each target considered in our analysis, we developed a small *reflector* program that parses the Cookie header and returns pairs of cookie names and values. Then, we wrote a simple fuzzer to generate variations of the Cookie request header and automatically assessed how the header was parsed by our programs. We acknowledge that this approach does not constitute a systematic evaluation of server-side parsing inconsistencies. However, our initial analysis provided strong evidence of the pervasiveness of the issue. We leave such comprehensive study as future work.

### 4.4 Measurement of Cookie Name Prefixes and Nameless Cookies

We present the results of our measurement of the prevalence of cookie name prefixes and nameless cookies in the top 100K websites. We based our evaluation on the public HTTP

Archive dataset [37] and performed all queries against the database provided by the Web Almanac initiative [46]. We considered the website popularity rank in the Chrome User Experience Report (CrUX) [56], which distinguishes the popularity of origins by orders of magnitude (top 1K, 10K, 100K, etc.). CrUX introduced the rank metric in February 2021 [60], thus we restricted the measurement to the last 2 years to avoid any bias due to mixing different ranking metrics. We also excluded third-party cookies from our analysis and focused instead on first-party cookies to avoid popular CDNs and analytics services from affecting the results.

Table 3 reports the outcome of our measurement performed on the dataset from June 2022. The table shows the number of origins that use the Secure attribute, the __Host- and __Secure- prefix, and nameless cookies. Fig. 4 provides a direct comparison between July 2021 and June 2022 of the adoption of cookies on the top 100K origins. As expected, prominent websites are more inclined towards well-established security features such as the Secure attribute. We found that more than 70% origins in the top 1K range are using secure cookies, while the percentage decreases to 60% in the top 100K range. Interestingly, while the adoption of secure cookies remained overall stable in the last 2 years for the top 1K websites, lower-ranked origins are increasingly adopting the Secure attribute. This trend becomes even more evident by focusing on the adoption of the __Host- prefix. Despite numbers being still low, the popularity of __Host- prefix is growing rapidly in the top 10K and top 100K ranges. Overall, 77 origins used the __Host- prefix in 2021, in contrast to the 133 origins that used it in 2022, which corresponds to a 72% increase in one year. On the other hand, the distribution of nameless cookies is more stable over time and does not show a clear correlation with the website rank.

Table 4 provides a characterization of __Host- and nameless cookies, showing the most common names and values, respectively, across the top 100K origins. Intuitively, the names adopted by __Host- cookies suggest that they are used to store sensitive data such as session identifiers or CSRF tokens. Nameless cookies, instead, are likely to be the result of misconfigurations on the server side, since the most common values match cookie attribute identifiers. A manual analysis of the full collection of nameless cookies did not reveal any clear intended usage. To the best of our knowledge, our study is the first to measure the prevalence of nameless cookies in the wild. The results suggest that nameless cookies are a byproduct of misconfigurations and are not actively used by websites. For these reasons, we advocate for the removal of nameless cookies from the cookie standard and browsers to eradicate this source of confusion and the serialization collision vulnerabilities discussed in Sec. 4.2. Conversely, we believe that the increasing adoption of __Host- cookies is a positive trend that should be further promoted among Web developers and security practitioners.

---

[3]https://www.browserstack.com/

| Rank | Origins | Secure | __Host- | __Secure- | Nameless |
|------|---------|--------|---------|-----------|----------|
| 1K | 732 | 537 (73.4%) | 6 (0.8%) | 1 (0.1%) | 1 (0.1%) |
| 10K | 5952 | 4005 (67.3%) | 14 (0.2%) | 19 (0.3%) | 6 (0.1%) |
| 100K | 58068 | 35098 (60.4%) | 113 (0.2%) | 109 (0.2%) | 86 (0.1%) |

Table 3: Number of origins from the 2022-06-01 dataset setting cookies, and the percentage of origins using the Secure attribute, cookie prefixes, and nameless cookies.
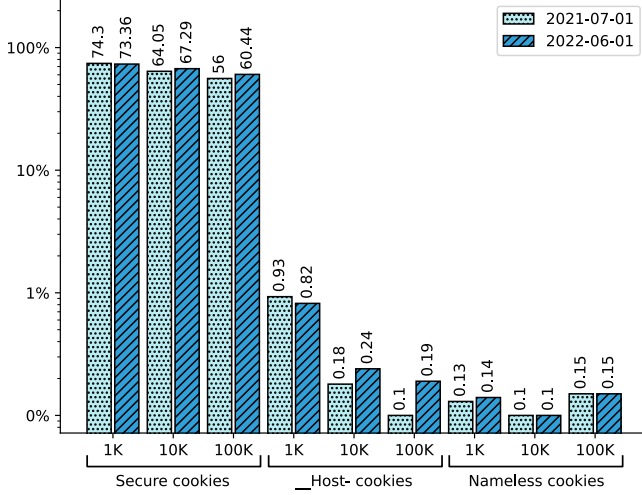


Figure 4: Deployment of cookies between 2021 and 2022.

# 5 CORF Token Fixation

We present a class of attacks that we call *CORF Token Fixation* that undermine implementations of the synchronizer token pattern in the presence of network or same-site attackers. The synchronizer token pattern is considered a robust CSRF protection against the same-site threat model [71] and is widely used in Web applications [62]. However, as we show in Sec. 6, common implementations are vulnerable to CORF attacks. The term *CSRF Token Fixation* has been used in the past to refer to a vulnerability affecting the Devise authentication library [80]. Although this vulnerability is an instance of our attack class, we provide for the first time a precise characterization of the attack flow and discuss a more general instance of the problem. Moreover, by factorizing the attacks into *fixation* and *replay* phases, we show how known bypasses to the double submit pattern can be framed in this class.

## 5.1 Token Fixation Attacks

Fig. 5 shows an instance of a token fixation attack (pre-login) that performs a state-changing request to a token-protected endpoint (`/action`). User sessions are represented as $sess[loggedin\text{-}status, csrf\text{-}secret]$, where $sess$ is the identifier for a session containing the *loggedin-status* and the *csrf-secret* value. Sessions can be stored on the server or the client side:

| __Host- cookie names | # | Nameless cookie values | # |
|----------------------|---|------------------------|---|
| __Host-next-auth.csrf-token | 26 | HttpOnly | 50 |
| __Host-GAPS | 23 | `<empty string>` | 16 |
| __Host-csrf-token | 13 | Secure | 6 |
| __Host-PHPSESSID | 10 | = | 5 |
| __Host-SESSION_LEGACY | 5 | ACookieAvailableCrossSite | 4 |
| __Host-SESSION | 5 | =0 | 3 |
| __Host-sess | 4 | secure | 1 |
| __Host-SWAFS | 3 | * | 1 |
| __Host-session | 3 | ^(.*)$ $1 | 1 |
| __Host-js_csrf | 3 | =1 | 1 |

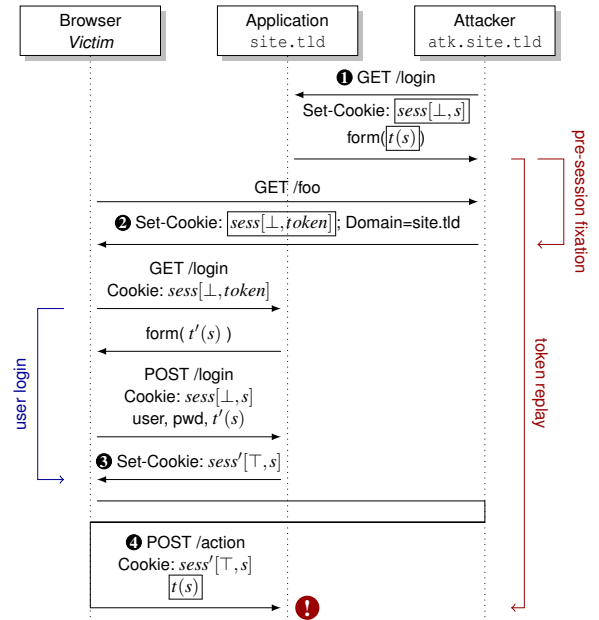Table 4: Top-10 __Host- cookie names and nameless cookie values from 2022-06-01.



Figure 5: CORF token fixation attack (pre-login).

in the first case, typically referred to as *stateful*, the cookie includes only the session identifier; in the latter, known as *stateless*, the content of the session is used as the cookie value, possibly after being encoded and signed. The attack flow is identical in both scenarios.

The attack has the following preconditions: (i) the target application uses the synchronizer token pattern, storing the CSRF secret in the session; (ii) the application constructs a pre-session for guest users (i.e., not logged-in) and has at least one CSRF token-protected form visible to guests. Alternatively, the CSRF token can be derived from information present in the pre-session. In the diagram, $t(s)$ represents the token that is attached to forms and derived (e.g., hashed or encoded) from the CSRF secret $s$; (iii) the CSRF secret is shared unchanged between the pre-session and the session.

When these preconditions are satisfied, the attack is per-

formed as follows: ❶ the attacker visits the target application and obtains the value of the pre-session cookie and the CSRF token that is bound to that pre-session; ❷ the attacker performs a pre-session fixation attack [61], setting the victim pre-session cookie to the value previously obtained by the attacker; ❸ by logging into the application, the user has an authenticated session $sess'$ which shares the CSRF secret $s$ with the attacker-known pre-session $sess$; ❹ the attacker causes the victim's browser to execute a crafted request towards the /action endpoint, attaching the value of the token $t(s)$ obtained in the first step. Given precondition (iii), the secret was preserved during the login process, so a valid token for the pre-session is accepted as a valid token for the authenticated session. This allows the attacker to perform a CORF attack that bypasses the CSRF token protection.

Note that the encoding/serialization mechanism used to derive a token from the secret $s$ may generate different tokens ($t(s)$ and $t'(s)$ in the figure) for different requests, e.g., by including an expiration date. In such cases, a server could disallow expired tokens or only accept the last token that was generated. Still, an attacker could bypass this protection by executing again step ❶ before constructing the request ❹ to obtain a valid token. Furthermore, the attack can be performed even if the victim has an already established authenticated session with the website. Besides setting a more specific path in the injected cookie, as described in Sec. 4.1.1, the attacker can forcibly logout the victim from the website using a cookie eviction technique (see Sec. 4.1.2) before fixating the pre-session cookie.

**Post-Login Variant.** The double submit pattern typically stores the CSRF secret in a separate cookie from the session. Hence, overwriting/shadowing this cookie (fixation phase) is sufficient to perform the attack, assuming that the attacker subsequently crafts a request to the protected endpoint with a CSRF token that matches the value of the overwritten cookie (replay phase). Notice that this attack variant does not require fixating the pre-session, thus lowering the set of preconditions compared to the STP bypass. Additionally, the post-login attack can be commonly performed without prior knowledge of a valid CSRF token. Still, whenever the server performs additional validation checks, an attacker can obtain a valid cookie for the application and its related CSRF token and use them to carry out the attack.

## 5.2 Mitigations

Token fixation attacks are enabled by cookie integrity violations from network and same-site attackers. Hence, preventing cookie tossing from sibling domains, i.e., via the __Host- cookie prefix, would trivially prevent the attacker from executing the fixation phase (step ❷). However, __Host- cookies may introduce compatibility issues on applications that use multiple origins. For instance, sharing the same session at accounts.example.com, where users log in, with the rest of the application at example.com, requires setting domain cookies.

**Token Secret Refresh for STP.** A robust mitigation for token fixation attacks for websites that implement the synchronizer token pattern consists in refreshing the value of the CSRF secret in the user session upon login. This update has the effect of using different secrets in the pre-session and in the authenticated session, so that precondition (iii) of the pre-login attack is no longer satisfied. This leads to the rejection of pre-session tokens in authenticated sessions and prevents the attacker from executing step ❹ of Fig. 5, since the token obtained at step ❶ is not valid for the new user session.

**Mitigating Attacks Against DSP.** In 2012, Wilander [87] proposed a variation of the double submit pattern named *triple submit cookies* to address a specific version of the attack. The mechanism employ random identifiers for both the name and value of the cookie, attaching only the random value to forms, and leveraging HttpOnly cookies to not disclose the random name with client-side scripts. The server-side validation of the submitted token may require storing the random name in the user session (stateful triple submit), or enforcing that the request contains only a single cookie with a random name, discarding the request otherwise (stateless). The stateful variant is equivalent to a synchronizer token pattern, where the random name acts as the CSRF secret and is stored in the user session. The stateless variant relies on the assumption that cookies cannot be erased since, otherwise, the attacker can forge a request with a single random-name cookie [64]. This assumption is only valid for Safari (see Sec. 4.1.2), thus the stateless triple submit is not effective in the general case. Consequently, the post-login attack can only be mitigated by (i) using __Host- prefix cookies, which are subject to compatibility issues, or (ii) switching to the synchronizer token pattern and refreshing the secret upon login.

## 6  Systematic Evaluation of Web Frameworks

We perform a study of Web development frameworks aimed at detecting session integrity vulnerabilities that may derive from the composition of security libraries, focusing on session management and CSRF protection components. In particular, we apply the threat models defined in Sec. 3 and leverage the techniques described in Sec. 4 to conduct the CORF token fixation attacks presented in Sec. 5. Albeit developers are ultimately responsible for securing their Web applications, we believe security abstractions should provide defaults that ensure safe composition. Hence we conducted the study on the default settings enabled by each framework. Moreover, we discuss relevant opt-in options that are listed in the documentation and assess how they affect security. As part of our work, we responsibly performed a coordinated disclosure of all the identified issues.

**Selection Criteria.** The selection criteria for the analyzed Web development frameworks follow the approach adopted

by Likaj et al. in their comprehensive study [62]. First, we considered the top 5 languages used for Web development in 2022 according to [54], i.e., JS, Python, Java, C#, and PHP, and then selected the most used frameworks from this pool. For this purpose, we used the GitHub metrics *watch*, *fork*, and *stars*, collected on April 8, 2022. We then picked the top 10 of each category. This selection resulted in a total of 13 frameworks. We refer the reader to Appendix A for the complete framework list and the associated GitHub metrics.

## 6.1 Frameworks Analysis Methodology

We conducted a manual security analysis to expose Web session integrity vulnerabilities in the selected frameworks. For each framework, we followed the official documentation to develop a toy application that includes a login form and a state-changing endpoint protected by a token-based CSRF mechanism. The login and CSRF functionalities were implemented using the official libraries provided by the framework. When official libraries were not available, we used external libraries that are widely used by the community, thus being considered the *de facto* standards. In two cases, we had to implement the session management functionality at the application level following the instructions provided in the documentation since no standard libraries were available. For each framework, we also developed an automated routine to simulate the attacker's website and to mechanize the CORF token fixation attacks.

We performed a coordinated disclosure of the identified vulnerabilities, and assisted framework developers to understand the threat model and to implement appropriate solutions that would improve the baseline security of their frameworks. We focused our disclosure on unsafe defaults, avoiding reports that would have been perceived by developers as potentially deceptive. For instance, we reported vulnerabilities on the double submit pattern only when this CSRF protection mechanism was set as default or it was the only one available. Double submit is indeed known to be vulnerable against same-site attackers, although it provides some protection against standard Web attackers.

Table 5 summarizes the results of our analysis categorizing each framework by language and including the selection of the libraries used to implement the login and CSRF functionalities, as well as the adopted CSRF protection mechanisms and the tested versions. The table also shows the outcome of our disclosure, denoted with an arrow symbol. Out of the 13 analyzed frameworks, we identified 12 supporting the synchronizer token pattern, among which 7 were found vulnerable to CORF token fixation attacks (pre-login). Furthermore, 6 frameworks implemented the double submit pattern, resulting vulnerable to the post-login attack variant. We also discovered 3 frameworks vulnerable to session fixation attacks, thus allowing an attacker to fully compromise the victim's account.

## 6.2 Synchronizer Token Pattern Bypasses

In the following, we present the security analysis of vulnerable real-world implementations of the synchronizer token pattern. All vulnerable frameworks, excluding CodeIgniter 4, failed to refresh the CSRF secret after a successful login, thus allowing an attacker to perform a CORF token fixation (pre-login) by reusing the CSRF token issued for the attacker's session following the steps described in Fig. 5.

### 6.2.1 Passport-Based: Express, Koa, Fastify

Several frameworks based on Node.js integrate with the Passport authentication middleware to support authenticated user sessions. Express natively integrates with Passport, Koa requires an additional Passport middleware (koa-passport), and Fastify provides its own port of Passport (fastify-passport). The CSRF protection is implemented by the csurf CSRF token middleware in Express, while Koa uses a different middleware called koa-csrf; Fastify, instead, provides CSRF protection via the csrf-protection plugin. All implementations support the synchronizer token pattern with the CSRF secret being stored in the session object. The login and user validation functions are performed by the `authenticate` function of Passport (and fastify-passport). We discovered that this function does not clear, nor reinitializes, the attributes in the session object other than those specific to Passport, e.g., the `passport` attribute. Hence, the session attribute `csrfSecret` (`secret` in Fastify) is not renewed upon successful authentication, satisfying the condition (iii) of our attack. Consequently, CSRF tokens issued to the attacker during the pre-session fixation step can be used to forge CORF requests after the victim authenticates on applications developed using these frameworks.

**Disclosure.** We reported this issue to the Passport developer, who promptly fixed it in version 0.6.0 by clearing all attributes from the session object after login, effectively solving the vulnerability on Express. However, for backward compatibility, Passport 0.6.0 supports the `keepSessionInfo` option that enables Web developers to opt out from the new safe behavior, and preserve the session attributes between pre-sessions and authenticated sessions. This option is set to `false` by default. CVE-2022-25896 was issued for this vulnerability. Fastify developers promptly fixed the vulnerability in version 2.3.0 by clearing all attributes from the session object after login and assigned CVE-2023-29020 to this vulnerability. The release also introduced support to the `clearSessionIgnoreFields` option that enables Web developers to define a set of session attributes to be preserved between pre-sessions and authenticated sessions. On the other hand, the new version of Koa middleware (6.0.0, published on February 2023) does not benefit from the best practices implemented in Passport 0.6.0 and remains vulnerable. We are currently in touch with the developers to identify an effective mitigation.

| Framework | Lang. | Auth. Library | CSRF Library | CSRF Protection | | CORF Token Fixation | | Session Fixation |
|---|---|---|---|---|---|---|---|---|
| | | | | STP | DSP | Pre-L | Post-L | |
| Express (4.18.1) [5] | JS | passport (0.5.3) [57] | csurf (1.11.0) [6] | ◐ | ◔ | ☠ → ✔ | ☠ | ☠ → ✔ |
| Koa (2.13.4) [14] | JS | koa-passport (4.1.3) [16] | csrf (3.0.8) [15] | ◐ | – | ☠ | – | ✔ |
| Sails (1.5.3) [20] | JS | in cookies as in docs | csurf (1.10.0) [6] | ◐ | – | ☠ → ☠ | – | ☠ → ☠ |
| Fastify (4.13.0) [8] | JS | fastify/passport (2.2.0) [10] | csrf-protection (6.1.0) [9] | ◔ | ◐ | ☠ → ✔ | ☠ → ✔ | ☠ → ✔ |
| Django (3.2.13) [4] | Python | built-in | built-in | ◔ | ◐ | ✔ | ☠ | ✔ |
| Flask (2.1.2) [11] | Python | flask-login (0.6.1) [12] | flask-wtf (1.0.1) [13] | ◐ | – | ☠ | – | ✔ |
| Tornado (6.2.0) [26] | Python | in cookies as in docs | built-in | – | ◐ | – | ☠ → ☠ | ✔ |
| Laravel (9.1.5) [17] | PHP | built-in | built-in | ◐ | – | ✔ | – | ✔ |
| Symfony (5.4.19) [23] | PHP | built-in | security-csrf (5.3.0) [24] | ◐ | – | ☠ → ✔ | – | ✔ |
| CodeIgniter 4 (4.2.1) [2] | PHP | shield (1.0.0-beta) [3] | built-in | ◔ → ◐ | ◐ → – | ☠ → ✔ | ☠ → – | ✔ |
| Yii2 (2.0.45) [28] | PHP | built-in | built-in | ◔ | ◐ | ✔ | ☠ | ✔ |
| ASP.NET Core (6.0.4) [1] | C# | built-in | built-in | ◐ | – | ✔ | – | ✔ |
| Spring (5.3.19) [21] | Java | Spring Security (5.6.3) [22] | Spring Security (5.6.3) | ◐ | – | ✔ | – | ✔ |

Table 5: Analyzed Web frameworks, and their respective authentication and CSRF libraries. ◐ default, ◔ available, ✔ unaffected, ☠ vulnerable, – not implemented. ✔ safe (insecure options available), ☠ vulnerable (secure options available).

### 6.2.2 Symfony

Symfony provides user management natively and relies on the official library security-csrf for CSRF protection. Symfony supports three different ways to handle session identifiers and session content while authenticating users, called *strategies*. The default strategy (MIGRATE) regenerates the session identifier upon login, but preserves the remaining session attributes. As the CSRF secret is not refreshed, the framework is vulnerable to the pre-login CORF token fixation attack. One specificity of Symfony is that the granularity of the CSRF mechanism can be configured to support distinct CSRF secrets depending on the endpoint. In this case, the pre-login attack still succeeds against all endpoints where it is possible to obtain a valid CSRF token under a pre-login session. The attacker simply needs to execute step ❶ towards all these endpoints to populate a pre-session with the corresponding CSRF secrets before executing step ❷.

**Disclosure.** This vulnerability was reported to the Symfony developers who updated the MIGRATE strategy to clear the CSRF storage in new versions of the library (v4.4.50, v5.4.20, v6.0.20, v6.1.12, v6.2.6). We stress that the two other strategies are either insecure or could introduce compatibility problems on websites based on Symfony: NONE preserves the same session after authentication, leading to session-fixation attacks, whereas INVALIDATE regenerates the session identifier and deletes all other attributes in the session. CVE-2022-24895 was issued after our disclosure.

### 6.2.3 Sails

Sails does not implement a login handler function, however it ships with a generator [19] that bootstraps a template application providing a user-management service based on express-session [7]. Sails can be configured to enable CSRF protection out of the box via the csurf library. Given that the user-management logic is hard-coded at the application level and that the session object is not refreshed upon login, any token generated before authentication is still valid after the user authenticates, thus satisfying the precondition (iii) of the attack. We expect Web developers to build their applications starting from the generated template application. For this reason, we consider this unsafe code pattern to be likely inherited by real-world websites.

**Disclosure.** The unsafe code pattern was reported to the Sails development team. As a result, a new version of the generator was released (2.0.7) with support for __Host- cookie prefixes in production mode (non-default). Using a __Host- cookie for the session addresses the vulnerability, although Web developers must be aware of cookie scope restrictions that may hamper the deployment of the protection, as discussed in Sec. 5.2.

### 6.2.4 Flask

Flask-based applications supporting user authentication often rely on the Flask-Login library for session management and Flask-WTF to provide CSRF protection using WTForms [27]. Login and user validation are performed by the login_user function that, similarly to Passport, does not clear nor reinitialize the attributes in the session object other than those specific for Flask-Login, thus satisfying precondition (iii) of the attack.

**Disclosure.** This vulnerability was disclosed to the developers of Flask and Flask-login, proposing a fix that would allow developers to define a set of opt-in attributes to be preserved upon login and to clear all others. Given that the two libraries operate separately, developers proposed instead to clear all attributes from the session and let application developers explicitly copy the attributes that should be preserved. A pull request for this issue is still open.

### 6.2.5 CodeIgniter 4

CodeIgniter 4 provides user management via the (official) library Shield [3], while CSRF protection is included natively and can be easily enabled. CodeIgniter 4 offers the synchronizer token pattern and double submit as CSRF protections, with the latter being the default option. For both mechanisms, the framework supports the option to regenerate the CSRF secret upon each CSRF-protected action (default), or to preserve the secret per session, via the option `security.regenerate = true` and `false` respectively. Similarly to the previous cases, CodeIgniter 4 is vulnerable to the CORF token fixation (pre-login) attack when the CSRF secret is not refreshed at login. However, we discovered that CodeIgniter 4 is also vulnerable when the CSRF secret is regenerated at login via the `security.regenerate = true` setting.

CodeIgniter 4 sessions objects are stored on the server and contain CSRF secrets as attributes called `csrf_test_name`. When a user accesses the application, a session object *sess* is created with secret *s*, and upon login, a new session *sess'* is created with secret *s'*. However, while creating *sess'*, the attribute `csrf_test_name` of *sess* is also updated to *s'*. Thus, the attack illustrated in Fig. 5 is still possible as the attacker, knowing *sess*, can perform an additional request between steps ❸ and ❹ to, e.g., `/login`, and obtain a fresh token $t'(s')$ that is valid for both the pre-session *sess* and the authenticated session *sess'*.

**Disclosure.** This vulnerability was communicated to the developers of Shield, who released a new fixed version of the library (1.0.0-beta.2) that (i) always refreshes the CSRF secrets at login, (ii) deletes pre-sessions upon login, and (iii) discontinues the double submit pattern in combination with Shield. CVE-2022-35943 was issued for this vulnerability.

## 6.3 Double Submit Pattern Issues

All analyzed frameworks implementing the double submit pattern were vulnerable to CORF token fixation attacks (post-login). Although this pattern is known to enable same-site attackers to bypass CSRF protections, our study aimed at identifying if any of the frameworks was applying mitigations such as the __Host- cookie prefix. We concluded that none of the frameworks applied the above mitigation. Fastify tried to mitigate this attack by including information related to the logged-in user in the CSRF token in order to prevent cookie tossing. It turns out that the attack was still possible if the `userInfo` associated with the target was predictable.

**Disclosure.** As discussed in Sec. 6.1, we did not contact developers of frameworks that were already applying safe defaults (Express) or were already aware of the risks associated with the double submit pattern (Django). The other vulnerabilities were communicated to the developers of the 4 remaining frameworks. Fastify addressed the vulnerability by performing an HMAC of the `userInfo` in order to pre-vent cookie tossing. CVE-2023-27495 was issued for this vulnerability. The CodeIgniter 4 Shield library disallowed the combination with the double submit pattern, relying now only on the synchronizer token pattern as a more robust CSRF protection. Tornado added optional support for the __Host-prefix to the CSRF cookie in version 6.3.0 [4]. Yii2 developers initially replied to our disclosure but, to the best of our knowledge, did not follow up on the issue.

## 6.4 Session Fixation Vulnerabilities

We also found 3 frameworks vulnerable to session fixation attacks. Session fixation attacks happen when pre-session cookies are preserved after authentication, thus allowing an attacker to hijack the session of an authenticated user violating its confidentiality and integrity [61]. The attack flow is the following: (i) the attacker obtains an unauthenticated session cookie `session_cookie=S` by visiting https://example.com; (ii) the victim is lured into visiting https://atk.example.com that sets a domain cookie for https://example.com/ in the victim's browser, such that `session_cookie=S`; (iii) the victim authenticates on https://example.com/; (iv) the attacker uses the session cookie `session_cookie=S` to hijack the victim's session at https://example.com/. Notice that regenerating the session cookie prevents session fixation, but it is not enough to mitigate CORF token fixation attacks if CSRF secret values still propagate unchanged to the authenticated session.

### 6.4.1 Passport

We identified a session fixation vulnerability in Passport stemming from the fact that the session attribute `sessionId` of the pre-session was not cleared nor reinitialized upon login, but rather preserved after user authentication.

**Disclosure.** This vulnerability was disclosed to the developers of the Passport library and was fixed in version 0.6.0 using the `Session.regenerate` method of the express-session module to generate a new `sessionId` after a successful login. CVE-2022-25896 was issued for this vulnerability.

### 6.4.2 Fastify

A session fixation attack similar to the one in Passport was also identified in Fastify when using the fastify/session plugin as the underlying session management mechanism (stateful).

**Disclosure.** This vulnerability was disclosed to the developers of fastify-passport and was fixed in version 2.3.0 using the `session.regenerate` method of fastify/session to generate a new `sessionId` after a successful login. CVE-2022-29019 was issued for this vulnerability.

---

[4]https://www.tornadoweb.org/en/stable/releases.html

### 6.4.3 Sails

A session fixation attack similar to the one in Passport was also identified in Sails. We recall that, although Sails does not implement a native login interface, it provides an application template that bootstraps a project. Consequently, unsafe code patterns embedded in the application template could be inherited by real-world websites.

**Disclosure.** This unsafe code pattern was disclosed to the Sails team. No particular action was taken to mitigate this unsafe pattern in the template application, although the addition of the optional `__Host-sails.sid` in production mode described before mitigates the impact of this attack.

## 7  Formal Verification of Web Frameworks

We complement the analysis of the top Web frameworks (Sec. 6) with the formalization of their session management mechanism and CSRF protections. The goal of our formalization is to verify the correctness of the mitigation to vulnerable synchronizer token patterns, i.e., the CSRF secret refresh discussed in Sec. 5.2. To this end, we use the WebSpi [38] library for the ProVerif [42] protocol verifier, which enables automated security proofs for Web applications.

Our formalization focuses on the 7 frameworks that are vulnerable to the pre-login token fixation attack and resulted in 4 different framework models that differ depending on whether the session is stored on the client or the server side, and on implementation details of the synchronizer token pattern adopted by the framework. This is the case since most JavaScript frameworks share the user management mechanism based on the Passport library, and, for instance, Express and Sails implement CSRF protection with the csurf library. The framework models implement a common API used by a generic application model to implement login and protected form elements. The application is run in parallel with a powerful same-site attacker that can overwrite any cookie on its sibling domains, independently from path or flags/prefixes. This attacker model over-approximates the threat models in Sec. 3, essentially considering cookies with no integrity and resulting more powerful than **SS-HOST-S**. This over-approximation ensures stronger security proofs, which are valid irrespectively of integrity assumptions on cookies.

A CSRF attack results from an unauthorized authenticated request to a protected endpoint performed by the attacker, thus we define our expected security property as follows.

**Invariant.** Every action executed by a token-protected endpoint must be explicitly initiated by an honest user by performing a request containing the token.

We encode the invariant as a *correspondence assertion* [88] between the two events (i) *app-action-successful*, that happens when the server successfully validates the CSRF token and performs the token-protected state-changing action, and (ii)

*app-action-begin*, that happens when the honest user submits the form that contains the CSRF token.

$$\forall (c : Cookie)(b : Browser)(token : CSRFToken).$$
$$\textbf{\textit{event}}(\textit{app-action-successful}(c, token)) \Rightarrow \textbf{\textit{event}}(\textit{app-action-begin}(b, token))$$

Intuitively, the correspondence requires that every instance of the *app-action-successful* event must be preceded by the *app-action-begin* event. This property explicitly forbids execution traces where the attacker successfully executes the protected action without the honest user submitting the form.

ProVerif confirms that the property does not hold on any of the four models, producing counterexamples that closely resemble the token fixation attack of Fig. 5. We then update the models to include the token refresh mitigation, i.e., generate a new CSRF secret upon user login (Sec. 5.2). Additionally, we refresh the session identifier on the model for Sails, Express, and Fastify (see session fixation attacks, Sec. 6.4). With these modifications, we obtain four fixed models for which ProVerif proves that our correspondence property is valid. Notice that in the presence of a session fixation attack, refreshing the CSRF secret is not enough for the property to hold, as the attacker can perform a full session hijacking attack and execute the token-protected action.

This analysis shows that refreshing the CSRF secret upon login makes the synchronizer token pattern a robust mitigation for CORF attacks, even in presence of same-site or network attackers who can fully compromise cookie integrity. We refer the reader to the extended version of our paper [77] for additional details on the formalization of Web frameworks.

## 8  Related Work

Several studies have focused on cookie integrity issues, with a particular emphasis on session integrity [44, 45, 47, 49, 70, 90]. In their seminal work, Bortz et al. [44] introduce the related-domain attacker model and propose a mechanism, named *origin cookies*, to bind cookies to specific origins. The `__Host-` prefix builds on this proposal and has been integrated into modern browsers [50]. Other studies suggest browser extension, e.g., to transparently strip session (cookie) identifiers from network requests to avoid session hijacking [45, 70]; Calzavara et al. [47] focus on the server-side by proposing a type system for verifying session integrity of PHP code against a variety of attackers, including network and related domain attackers. These works, except for [90], do not assess the implications of the lack of cookie integrity for real world application. Zheng et al. [90] present an empirical assessment of cookie injection attacks on the Web, taking into account both browser-side and server-side cookie handling inconsistencies, and discovering attacks on popular Web sites (e.g., Google, Amazon). Similarly to our work, the authors discover browser implementation differences in storage limits for cookies and cookie ordering in requests, and inconsistencies in server-side languages such as the automatic percent decoding

of cookie names in PHP. Our findings uncover that, even after seven years, these types of cross-browser/language inconsistencies are still relevant and also affect newly introduced security mechanisms such as `__Host-` prefix cookies.

Recently, Squarcina et al. [78] measured and quantified the threats posed by same-site attackers to Web application security. In their study of cookies, they discovered that the majority of the cookies on sites vulnerable to subdomain takeover has no integrity against related domain attackers. The authors highlight that the `__Host-` prefix was used only once in their dataset. Our measurement (Sec. 4.4) confirms the infrequent usage of the prefix in the wild, but shows a promising positive trend on its adoption in the last 2 years, especially on lower-ranked websites. Sanchez-Rola et al. [74] performed a large-scale measurement to characterize cookie-based Web tracking. The study shows that third-party script inclusion enables cookie sharing in the context of first-party cookies, thus enabling third parties to set cookies on behalf of the visited website. Additionally, the authors report on instances of *cookie collisions*, where scripts from different actors in the same website access cookies created with the same name but different semantics. This setting matches our **SS-XSS-S** threat model (Sec. 3), where different parties gain control of a domain on a page served via HTTPS. However, unlike the study of Sanchez-Rola et al., which does not consider domain cookies, we focus on cookie integrity violations from attacker-controlled subdomains.

Concerning the analysis of Web frameworks, Likaj et al. [62] evaluated the mechanisms implemented by major Web frameworks, quantifying their exposure to CSRF attacks as a result of implementation mistakes, cryptography-related flaws, cookie integrity violations, or leakage of CSRF tokens. The authors discover that 37 out of 44 frameworks are affected by such issues. Our analysis of Web frameworks (Sec. 6) shows that further implementation issues in the synchronizer token pattern (deemed secure in [62]), originating from the composition of different libraries, lead to a bypass of the protection in the presence of same-site attackers. For instance, the CORF token fixation attack sidesteps the Flask framework protection, which was considered secure in previous work.

## 9 Conclusion

This study is a modern look at cookie integrity issues and their impact on Web application security. Our research showed that integrity vulnerabilities are not limited to implementation bugs, but are a pervasive threat across the Web due to compositionality problems at multiple levels. We engaged with browser vendors, the IETF HTTP Working Group, and Web framework developers to address the discovered issues, which resulted in several high-impact updates, e.g., Chrome and Firefox, PHP (the server-side language powering 78% of all websites), major authentication libraries such as Passport (2M weekly downloads), and the cookie standard [50].

## References

[1] ASP.NET. https://dot.net.

[2] CodeIgniter 4. https://codeigniter.com/user_guide/index.html.

[3] CodeIgniter Shield. https://codeigniter4.github.io/shield/.

[4] Django Framework. https://www.djangoproject.com/.

[5] Express. https://expressjs.com/.

[6] Express csurf: CSRF token middleware. https://github.com/expressjs/csurf.

[7] Express Session. https://github.com/expressjs/session.

[8] Fastify. https://www.fastify.io/.

[9] Fastify csrf-protection. https://github.com/fastify/csrf-protection.

[10] Fastify Passport. https://github.com/fastify/fastify-passport.

[11] Flask. https://flask.palletsprojects.com/.

[12] Flask Login. https://flask-login.readthedocs.io/.

[13] Flask WTF. https://flask-wtf.readthedocs.io/.

[14] Koa. https://koajs.com.

[15] Koa CSRF. https://github.com/koajs/csrf.

[16] Koa Passport. https://github.com/rkusa/koa-passport.

[17] Laravel Framework. https://laravel.com/.

[18] PHP Manual: Using Register Globals. https://web.archive.org/web/20201205183413/https://www.php.net/manual/en/security.globals.php.

[19] Sails Generate. https://sailsjs.com/documentation/reference/command-line-interface/sails-generate.

[20] Sails.js. https://sailsjs.com/.

[21] Spring. https://spring.io/.

[22] Spring Security. https://spring.io/projects/spring-security.

[23] Symfony. https://symfony.com/.

[24] Symfony CSRF. https://github.com/symfony/security-csrf.

[25] The Web Platform Tests Project. https://web-platform-tests.org/.

[26] Tornado Web Server. https://www.tornadoweb.org/.

[27] WTForms. https://wtforms.readthedocs.io/.

[28] Yii PHP framework. https://www.yiiframework.com/.

[29] Chromium Bugs. Issue 1351601: Cookie prefixes bypass via nameless cookies (rfc6265bis). https://bugs.chromium.org/p/chromium/issues/detail?id=1351601, 2022.

[30] Chromium Bugs. Issue 1354090: post-CVE-2022-2860 security limitations of cookie prefixes and nameless cookies. https://bugs.chromium.org/p/chromium/issues/detail?id=1354090, 2022.

[31] IETF HTTP Working Group, HTTP Extensions. Issue 2229: [rfc6265bis] nameless cookies, client/server inconsistencies #2229. https://github.com/httpwg/http-extensions/issues/2229, 2022.

[32] Mozilla Bugzilla. Issue 1782561. document.cookie desynchronization after cookie jar overflow. https://bugzilla.mozilla.org/show_bug.cgi?id=1782561, 2022.

[33] Mozilla Bugzilla. Issue 1783536. document.cookie in an insecure origin process allows setting an insecure cookie in that process that has the same name as a secure one. https://bugzilla.mozilla.org/show_bug.cgi?id=1783536, 2022.

[34] Mozilla Bugzilla. Issue 1783982: Cookie prefixes bypass via nameless cookies (rfc6265bis). https://bugzilla.mozilla.org/show_bug.cgi?id=1783982, 2022.

[35] PHP Bug Tracker. Issue 81727: cookie integrity vulnerabilities. https://bugs.php.net/bug.php?id=81727, 2022.

[36] Angular. HTTP: Security - XSRF Protection. https://angular.io/guide/http#security-xsrf-protection, 2022.

[37] HTTP Archive. The HTTP Archive. https://httparchive.org/.

[38] C. Bansal, K. Bhargavan, and S. Maffeis. Discovering Concrete Attacks on Website Authorization by Formal Analysis. In *CSF*. IEEE, 2012.

[39] A. Barth. HTTP State Management Mechanism. RFC 6265, IETF, 2011.

[40] A. Barth. The Web Origin Concept. RFC 6454, IETF, 12 2011.

[41] A. Barth, C. Jackson, and J. C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *CCS*. ACM, 2008.

[42] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *WCSF*. IEEE, 2001.

[43] K. Borgolte, T. Fiebig, S. Hao, C. Kruegel, and G. Vigna. Cloud Strife: Mitigating the Security Risks of Domain-Validated Certificates. In *NDSS*, 2018.

[44] A. Bortz, A. Barth, and A. Czeskis. Origin Cookies: Session Integrity for Web Applications. In *W2SP*, 2011.

[45] M. Bugliesi, S. Calzavara, R. Focardi, and W. Khan. CookiExt: Patching the browser against session hijacking attacks. *Journal of Computer Security*, 2015.

[46] Web Almanac by HTTP Archive. Http archive's annual state of the web report. https://almanac.httparchive.org/, 2022.

[47] S. Calzavara, R. Focardi, N. Grimm, M. Maffei, and M. Tempesta. Language-Based Web Session Integrity. In *CSF*. IEEE, 2020.

[48] S. Calzavara, R. Focardi, M. Squarcina, and M. Tempesta. Surviving the Web: A Journey into Web Session Security. *CSUR*, 50(1):13:1–13:34, 2017.

[49] S. Calzavara, A. Rabitti, and M. Bugliesi. Sub-Session Hijacking on the Web: Root Causes and Prevention. *Journal of Computer Security*, 2019.

[50] L. Chen, S. Englehardt, M. West, and J. Wilander. Cookies: HTTP State Management Mechanism (IETF Draft). RFC 6265bis, IETF, 2022.

[51] GitHub Advisory Database. ReactPHP's HTTP server parses encoded cookie names so malicious __Host- and __Secure- cookies can be sent. https://github.com/advisories/GHSA-w3w9-vrf5-8mx8, 2022.

[52] GitHub Advisory Database. Incorrect parsing of nameless cookies leads to __Host- cookies bypass. https://github.com/pallets/werkzeug/security/advisories/GHSA-px8h-6qxv-m22q, 2023.

[53] Detectify. Hostile subdomain takeover using heroku/github/desk + more. https://labs.detectify.com/2014/10/21/hostile-subdomain/, 2014.

[54] GitHub. Top programming languages. https://octoverse.github.com/2022/top-programming-languages, 2022.

[55] T. Van Goethem, C. Pöpper, W. Joosen, and M. Vanhoef. Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections. In *USENIX Security*, 2020.

[56] Google. Chrome UX Report. https://developer.chrome.com/docs/crux/.

[57] J. Hanson. Passport – Simple, unobtrusive authentication for Node.js. https://www.passportjs.org/.

[58] S. Helme. Cross-Site Request Forgery is dead! https://scotthelme.co.uk/csrf-is-dead/, 2017.

[59] S. Helme. CSRF is (really) dead. https://scotthelme.co.uk/csrf-is-really-dead/, 2019.

[60] J. Henkel and B. Pollard. Adding Rank Magnitude to the CrUX Report in BigQuery. https://developer.chrome.com/blog/crux-rank-magnitude/, 2021.

[61] M. Kolšek. Session fixation vulnerability in web-based applications. https://acrossecurity.com/papers/session_fixation.pdf, 2002.

[62] X. Likaj, S. Khodayari, and G. Pellegrino. Where We Stand (or Fall): An Analysis of CSRF Defenses in Web Frameworks. In *RAID*. ACM, 2021.

[63] D. Liu, S. Hao, and H. Wang. All Your DNS Records Point to Us: Understanding the Security Threats of Dangling DNS Records. In *CCS*. ACM, 2016.

[64] R. Lundeen. The Deputies are Still Confused. https://media.blackhat.com/eu-13/briefings/Lundeen/bh-eu-13-deputies-still-confused-lundeen-wp.pdf, 2013.

[65] Microsoft. Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core. https://learn.microsoft.com/en-us/aspnet/core/security/anti-request-forgery, 2022.

[66] Mozilla. Public Suffix List. https://publicsuffix.org/.

[67] Mozilla. Project Fission. https://wiki.mozilla.org/Project_Fission, 2022.

[68] Mozilla Developer Network. Cookie Store API. https://developer.mozilla.org/en-US/docs/Web/API/Cookie_Store_API.

[69] Mozilla Developer Network. Set-Cookie. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie.

[70] N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen. SessionShield: Lightweight Protection against Session Hijacking. In *Engineering Secure Software and Systems*. Springer Berlin Heidelberg, 2011.

[71] OWASP. Cross-site request forgery prevention cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html.

[72] OWASP. Session hijacking attack. https://owasp.org/www-community/attacks/Session_hijacking_attack.

[73] I. Sanchez-Rola, D. Balzarotti, and I. Santos. BakingTimer: Privacy Analysis of Server-Side Request Processing Time. In *ACSAC*. ACM, 2019.

[74] I. Sanchez-Rola, M. Dell'Amico, D. Balzarotti, P. Vervier, and L. Bilge. Journey to the Center of the Cookie Ecosystem: Unraveling Actors' Roles and Relationships. In *S&P*. IEEE, 2021.

[75] Amazon Web Services. Working with AWS Lambda proxy integrations for HTTP APIs. https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-develop-integrations-lambda.html, 2022.

[76] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. On the Incoherencies in Web Browser Access Control Policies. In *S&P*. IEEE, 2010.

[77] M. Squarcina, P. Adão, L. Veronese, and M. Maffei. Cookie Crumbles: Breaking and Fixing Web Session Integrity – source code, artifacts, and extended version of the paper. https://github.com/SecPriv/cookiecrumbles, 2023.

[78] M. Squarcina, M. Tempesta, L. Veronese, S. Calzavara, and M. Maffei. Can I Take Your Subdomain? Exploring Same-Site Attacks in the Modern Web. In *USENIX Security*, 2021.

[79] A. Sudhodanan, S. Khodayari, and J. Caballero. Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks. In *NDSS*, 2020.

[80] J. Valim. CSRF token fixation attacks in Devise. https://blog.plataformatec.com.br/2013/08/csrf-token-fixation-attacks-in-devise/, 2013.

[81] T. Van Goethem, G. Franken, I. Sanchez-Rola, D. Dworken, and W. Joosen. SoK: Exploring Current and Future Research Directions on XS-Leaks through an Extended Formal Model. In *ASIA CCS*. ACM, 2022.

[82] W3C. Working Draft: Clear Site Data. https://www.w3.org/TR/clear-site-data/, 2017.

[83] W3Techs. Usage statistics of PHP for websites. https://w3techs.com/technologies/details/pl-php, 2023.

[84] web.dev. Schemeful Same-Site. https://web.dev/schemeful-samesite/.

[85] M. West. Cookie Prefixes. https://tools.ietf.org/html/draft-west-cookie-prefixes-05.

[86] M. West and M. Goodwin. RFC6265: Same-site Cookies draft-west-first-party-cookies-07, 2016.

[87] J. Wilander. Advanced CSRF and Stateless Anti-CSRF. https://owasp.org/www-pdf-archive//AppSecEU2012_Wilander.pdf, 2012.

[88] T. Y. C. Woo and S. S. Lam. A semantic model for authentication protocols. In *S&P*. IEEE, 1993.

[89] M. Zalewski. The tangled web: A guide to securing modern web applications. No Starch Press, 2011.

[90] X. Zheng, J. Jiang, J. Liang, Haixin Duan, S. Chen, T. Wan, and N. Weaver. Cookies lack integrity: Real-world implications. In *USENIX Security*, 2015.

## A  Web Framework Analysis

Table 6 lists the entire pool of Web frameworks considered for this study. We restricted the analysis to the top 10 frameworks according to the GitHub metrics *watch*, *fork*, and *stars*, obtaining the final set of 13 frameworks.

| Framework | Language | GH Watch | GH Fork | GH Star |
|-----------|----------|----------|---------|---------|
| ASP.NET MVC | C# | 75 | 329 | 739 |
| ASP.NET Core | C# | **1.4k** | **7.7k** | **27.8k** |
| Service Stack | C# | 515 | 1.6k | 5k |
| Nancy | C# | 438 | 1.5k | 7.2k |
| Spring | Java | **3.4k** | **33.3k** | **47.1k** |
| Play | Java | 683 | 4k | 12.1k |
| Spark | Java | 413 | 1.6k | 9.3k |
| Vert.x-web | Java | 79 | 470 | 955 |
| Vaadin | Java | 53 | 59 | 361 |
| Dropwizard | Java | 398 | 3.4k | 8.2k |
| Blade | Java | 302 | 1.1k | 5.6k |
| ZK | Java | 46 | 169 | 350 |
| Apache Struts | Java | 124 | 737 | 1.1k |
| Apache Wicket | Java | 61 | 354 | 616 |
| Express | JS | **1.8k** | **9.6k** | **56.6k** |
| Meteor | JS | 1.6k | 5.2k | 42.9k |
| Koa | JS | 847 | 3.2k | **32.5k** |
| Hapi | JS | 422 | 1.4k | 13.8k |
| Sails | JS | 667 | 2k | **22.2k** |
| Fastify | JS | 281 | 1.7k | **22.7k** |
| ThinkJS | JS | 268 | 643 | 5.3k |
| Total.js | JS | 218 | 459 | 4.1k |
| AdonisJS | JS | 229 | 579 | 12.3k |
| Laravel | PHP | **4.6k** | **22.4k** | **69.3k** |
| Symfony | PHP | **1.2k** | **8.6k** | **26.7k** |
| Slim | PHP | 525 | 1.9k | 11.3k |
| CakePHP | PHP | 573 | 3.5k | 8.5k |
| Zend/Laminas | PHP | 18 | 56 | 1.4k |
| CodeIgniter | PHP | **1.6k** | **7.8k** | 18.2k |
| FuelPHP | PHP | 107 | 287 | 1.4k |
| Yii2 | PHP | **1.1k** | **7k** | 13.9k |
| Phalcon | PHP | 658 | 1.9k | 10.6k |
| Li3 | PHP | 91 | 247 | 1.2k |
| CodeIgniter4 | PHP | 278 | 1.6k | 4.2k |
| Flask | Python | **2.2k** | **15k** | **58.5k** |
| Django | Python | **2.3k** | **26.9k** | **63.3k** |
| Tornado | Python | **1k** | **5.4k** | 20.5k |
| Bottle | Python | 320 | 1.4k | 7.6k |
| Pyramid | Python | 160 | 878 | 3.7k |
| Falcon | Python | 273 | 872 | 8.7k |
| Zope | Python | 91 | 99 | 288 |
| Masonite | Python | 57 | 104 | 1.7k |
| TurboGears2 | Python | 32 | 76 | 777 |
| Web2py | Python | 220 | 866 | 2k |

Table 6: Web development frameworks from [62] ranked according to GitHub metrics as of April 8, 2022.