Cookie Crumbles: Breaking and Fixing Web Session Integrity

Marco Squarcina
TU Wien

Pedro Adão
Universidade de Lisboa

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 8 CVEs, 23 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 12 Web Frameworks, exposing session integrity vulnerability in 7 of them. We discuss our responsible disclosure and propose practical mitigation strategies.

1 Introduction

HTTP cookies are the oldest and most widely used mechanism used to share state 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 [34,37,56,67,77].

Entire classes of attacks revolve around compromising either the confidentiality or the integrity of cookies [41]. 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 [65]. *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 [53]. *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 [34].

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 [43] which includes extended security features compared to the 2011 version [32]. A notable example is the *SameSite* attribute, which has been touted as a robust solution against CSRF attacks [50, 51]. 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 [61]. On the server side, traditional protections against CSRF attacks include the usage of a secret token shared between browsers and servers [34]. This approach has been widely adopted by popular Web frameworks and considered an effective defense in the *synchronizer pattern* variant [54, 64]. As a result of these changes, CSRF attacks fell outside the OWASP top 10 list in 2017 [63].

In this paper, we question the effectiveness of these new protections and study the real-world security implications of cookie integrity issues. In particular, we focus on network and same-site attackers [37], a class of attackers that has been largely overlooked in the literature but is increasingly becoming a significant threat to Web application security [68]. 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 8 CVEs, 23 vulnerability disclosures, and updates to the RFC of the cookie standard [43]. We identified novel attack vectors that bypass modern cookie protections and introduced a new attack against state-of-the-art CORF prevention mechanisms, called *CORF token fixation*. We performed a systematic security analysis of the top 12 Web frameworks, exposing session integrity vulnerabilities in 7 of them. We showed how these vulnerabilities are not only implementation-specific bugs but are caused by compositionality issues of security mechanisms or vulnerabilities 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. [68] to propose a taxonomy of threat models that precisely characterize 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 introduce the CORF token fixation, a new attack that bypasses real-world implementations of the synchronizer token pattern, which is considered a robust defense against CORF.

- Sec. 6 presents a systematic security analysis of the top 12 Web frameworks, exposing CORF and session fixation vulnerabilities in 7 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 [35] (Sec. 7).

We plan to release all artifacts developed during this research, including the browser test suite in 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).

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) [33] 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) [58]. 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 [72]. Hence, sibling domains with different schemes are considered same-site, but not *schemeful same-site*. To avoid ambiguities in this work, 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 [43]. 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 [60].

Attributes. Cookies can be configured with attributes, or flags, which specify additional properties or constraints on the cookie. The `Expires` and `Max-Age` attributes specify when cookies expire and should be removed from the browser storage (or cookie jar). 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 default value for the cookie Path is the directory part of the path of the URL that set the cookie. 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 set 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 cookie with the same name and scope marked as Secure.

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 [74]. 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. Note that 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. Starting from 2020, browsers started enforcing the `Lax` policy by default on cookies which do not specify a `SameSite` attribute [69]. Due to compatibility issues, however, the specification was changed to allow a new enforcement mechanism, named `Lax-Allowing-Unsafe` [43], for such default-lax cookies. In this mode the `Lax` policy is relaxed to allow cross-site POST requests in the first 2 minutes.

Cookie Prefixes. Cookie prefixes, originally introduced in 2015 [73], 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-`, it is subject to all restrictions of the `__Secure-` attribute. Additionally, 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 class of attacks where the adversary executes unauthorized state-changing actions under the victim's authenticated session on a target website. 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. The ways this request is performed depend on the capabilities available to the adversary, and include, for instance, auto-submitting a form, or using the fetch API.

To exemplify the attack flow, assume that an attacker wants to perform a transaction on behalf of a victim at bank.com. The attacker prepares a malicious website at evil.com that includes a form with all the data needed to execute the transaction, e.g., desired amount and destination account. The at-

tacker then lures the victim to visit the malicious website, which automatically submits the form to bank.com. Assuming that the victim has an active session with the bank, the user's credentials, i.e., the session cookie, are attached to the request. Upon receiving the request with the valid session cookie, bank.com performs the transaction forged by the attacker under the victim's session.

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 [34, 54]. 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) attacks 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 [54], 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 two of the most popular token-based protection patterns, the *Synchronizer Token Pattern* and the *Double Submit Pattern* [54, 64].

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 verifies the correctness of this binding on newly received tokens before considering them valid. Multiple implementations (see Sec. 6) generate a fixed CSRF secret s per user session, and use it to derive CSRF tokens $t(s)$. Other implementations opt for generating 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 pattern is similar to STP, but it does not require CSRF tokens to be bound to the user session. This makes

3

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).

3 Threat Model

In this work, we aim to investigate the security risks that arise from the interaction between a website and a victim's browser when an attacker can forge cookies scoped to the target website. We focus on violations of cookie integrity, i.e., the ability of an attacker to force cookies in the victim's browser to override the values of authentic cookies.

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 are extensively covered by previous work [46, 55, 68].

We focus, instead, on the capabilities of standard threat models by building on the framework introduced by Squarcina et al. [68]. 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`.
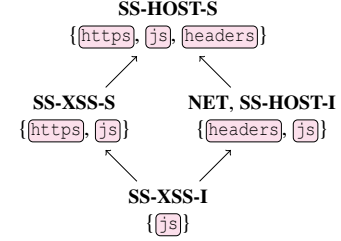

Figure 1: Taxonomy of threat models for cookie integrity violations.

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 [68]. 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`.
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 [78].

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.

	Standard	Browser		
Attack	📕 RFC6265bis	🅖 ≤v109	🦊 ≤v109	🧭 ≤v16.0
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)	●≥04	✔	🐞	✔
Serialization collision (__Host-)	⊗≥11	🐞<v104	🐞<v105	✔
Cookie jar desynchronization	⊗	✔	🐞	✔
Server-side parsing issues	⊗	—	—	—
Parser-chaining	●	—	—	—

Table 2: Cross-browser evaluation of cookie integrity attacks. ● = standard compliant, ⊗ = standard violation, ✔ = unaffected, 🐞 = vulnerable, — = does not apply.

Figure 2: Cookie tossing attack.

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 adopted to discover these issues and report on a measurement study performed using the HTTP Archive dataset [30].

4.1 Weak Integrity

Due to their legacy design, cookies have a long history of integrity issues, as documented in the cookie specification [43]. 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 a `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 [78]. Since attributes are not sent along with cookies, duplicated cookies with the same name but different `Path` attributes are indistinguishable to the server [43, §5.7.3].

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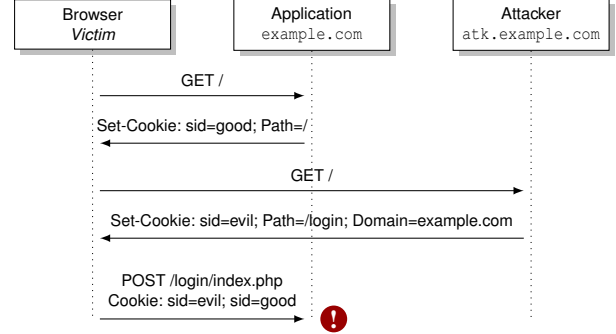
sume also an attacker in control of `http://atk.example.com/`. The attacker can set a domain cookie for example.com with name `sid` and value `evil`. By setting a more specific path in the new cookie, the attacker can cause the victim's browser to send the attacker's controlled cookie first, as illustrated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

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

Despite these known issues, the cookie standard evolved in recent years to provide stronger integrity guarantees. In partic-
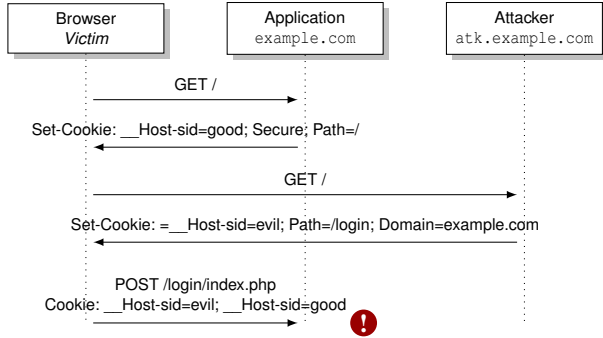
Figure 3: __Host- cookie bypass via nameless cookies.

ular, the __Host- prefix was proposed in 2015 [73] 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.

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 example.com that sets a named cookie `sid=good`. A same-site attacker can set a nameless cookie scoped to example.com with value `sid=evil`. This can be done via either the Document.cookie property or the HTTP response header `Set-Cookie: =sid=evil; Domain=example.com`, 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 Document.cookie, 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 be used to 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. The attack we just presented, 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 [23] and jointly disclosed the __Host- cookie bypass to the Chrome [21] and Firefox [26] security teams who issued CVEs 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 a =. Our contributions and extensive discussion with browser maintainers [22] led to updates to the cookie standard [43, §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 [43, §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 incoming cookies received via the `Cookie` header. This discrepancy can cause server-side cookie integrity violations whenever the server parses two distinct cookies as the same cookie.

Although the problem is not new per se [78], we discovered a new vulnerability that bypasses __Host- cookies in PHP [27], the server-side language used by 78% of all websites according to recent measurements [71]. Due to the legacy design derived from `register_globals` [11], 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=example.com`, that is parsed by PHP as `Cookie: __Host-sid=evil`. This vulnerability raises further integrity concerns when legitimate cookies contain the underscore symbol. For instance, non-secure origins can use this bug to shadow secure cookies.

We also reported a similar vulnerability to the ReactPHP library [44]. The HTTP server component incorrectly parses the `Cookie` header by url-decoding cookie names. This vulnerability can be exploited to bypass __Host- cookies using percentage-encoded cookie names, e.g., a cookie set via

---

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

[2]The __Host- bypass vulnerability was reported 2 weeks before us as a parallel effort by an independent researcher who is credited on both CVEs. Our issues were merged into the precedent vulnerability reports to jointly discuss the mitigation.

```
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.

Set-Cookie: %5F%5Host-sid=evil; Domain=example.com is parsed by ReactPHP as Cookie: __Host-sid=evil.

We also discovered a vulnerability in the werkzeug library web, the HTTP middleware used by the popular Flask framework [45]. The Cookie header is incorrectly parsed to strip all leading = symbols. To exemplify, a nameless cookie set via Set-Cookie: ==__Host-sid=evil; Domain=example.com 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 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, at the time of writing (January 2023), has been patched in the development branch. A new release is pending.

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 [24].

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, or using the Clear-Site-Data header [70], 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 [57] and Angular [29].

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 [25]. 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 the Document.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 [59]. At the time of writing (January 2023), the issues are still under active investigation, since a fix requires some architectural changes without degrading the security guarantees provided by Site Isolation.

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 for building, deploying, and managing APIs. It acts as a frontend for accessing backend services and integrates with 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 [66]: "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 shown in Listing 2. The resulting cookie is 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 who deployed a fix in November 2022. The

```
1  { "version": "2.0",
2    "routeKey": "ANY /{proxy+}",
3    "rawPath": "/",
4    "rawQueryString": "",
5    "cookies": [
6        "__Host-sid=evil"
7    ], ... }
```

Listing 2: AWS Lambda proxy __Host- cookie collision.

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 [43], effectively protects against the threat described in this section.

4.3 Methodology

The methodology used to discover the attacks discussed in this section consisted of three main stages.

First, we were interested in performing a comprehensive evaluation of known cookie integrity attacks across the top-3 browsers (Chrome, Firefox, and Safari). Inspired by the WPT project [18], 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 versions of Chrome, Firefox, and Safari, 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 of the browsers over time.

This phase was crucial to uncovering 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 [43]. 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. The most significant results of the evaluation are presented in Table 2.

Second, whenever a discrepancy was found between the browsers, we manually reviewed the cookie standard [43] 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

---

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

issues. We engaged with the IETF HTTP Working Group and browser vendors to address the problems as we found them.

It appeared evident that correct cookie handling could not be achieved by relying on the browsers alone. Therefore, 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 [30] and performed all queries against the optimized tables provided by the Web Almanac initiative [39]. We considered the website popularity rank in the Chrome User Experience Report (CrUX) [48], which distinguishes the popularity of origins by orders of magnitude (top 1K, 10K, 100K, etc.). CrUX introduced the rank metric in February 2021 [52], 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

Rank	w/ cookies	w/ Secure	w/ __Host-	w/ __Secure-	nameless
1000	732	537 (73.36%)	6 (0.82%)	1 (0.14%)	1 (0.14%)
10000	5952	4005 (67.29%)	14 (0.23%)	19 (0.32%)	6 (0.10%)
100000	58068	35098 (60.44%)	113 (0.19%)	109 (0.19%)	86 (1.15%)

Table 3: Deployment of cookies per origin from the 2022-06-01 dataset: Secure , __Host-, __Secure, and nameless. Percentages refer to the total number of origins using cookies.

__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 4: Comparison of cookie deployment between 2021 and 2022.

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

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.

5 CORF Token Fixation

In this section, we present a novel attack named *CORF Token Fixation* that bypasses synchronizer-based CSRF protections in presence of network or same-site attackers. This attack is enabled by the lack of cookie integrity guarantees described in Sec. 4. The synchronizer token pattern is considered to be robust against the same-site threat model [64] and is widely used in Web applications [54]. However, common implementation practices of the pattern (see Sec. 6) open Web applications to CORF attacks. Moreover, by factorizing the attack into a *fixation* and a *replay* phase, we show that known bypasses to the double-submit pattern are instances of our attack.

5.1 Token Fixation Attacks

Fig. 5 shows an instance of a token fixation attack (pre-login) allowing one to perform an authenticated state-changing request to a token-protected endpoint (/action). User sessions

are represented as *sess*[*loggedin-status*, *csrf-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: 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 successful execution of 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 CSRF secret s. (iii) The CSRF secret is shared between the pre-session and the session.

When the preconditions are satisfied, the attack is performed as follows. ❶ By visiting the target application, the attacker obtains both the value of the pre-session cookie and the token that is attached to forms for that pre-session (e.g., the CSRF token of the login form). ❷ The attacker performs a pre-session fixation attack, setting the victim pre-session cookie to the same value as the attacker's one. 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 attaches the value of the token $t(s)$, that was obtained at step ❶, to a request performed by the victim to the /action endpoint. The application validates the received token against the corresponding secret s in the session, accepting the request. 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 post-login token fixation attack variant affects applications that employ the double submit pattern for CSRF protection. Similarly to the pre-login variant, the second precondition (ii) must be valid. Conversely, there is no need for the secret to be shared between pre-session and session (iii). In particular, given that the double submit pattern typically relies on storing the secret in a separate cookie from the session, overwriting/shadowing this cookie (fixation) is enough to perform the attack with a valid CSRF token (replay), w.r.t. the overwritten cookie.

5.2 Mitigations

Token fixation attacks are enabled by cookie integrity violations from network and same-site attackers. Hence, preventing cookie tossing attacks 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 Web 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. Here we discuss the mitigation to the pre-login attack and refer to Appendix A for the post-login variant.

Token Secret Refresh (Pre-Login). A robust mitigation for token fixation attacks for websites that implement the synchronizer 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 session, so that precondition (iii) of the pre-login attack is not 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.

6 Systematic Evaluation of Web Frameworks

In this section, 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. Albeit we agree that developers are ultimately responsible for securing their Web applications, we also believe that security abstractions should provide defaults that ensure safe composition. 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. 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. We discuss representative case studies of vulnerable frameworks in this section and refer the reader to the full list of framework issues in Appendix D.

Selection Criteria. The selection criteria of Web development frameworks to analyze followed the approach adopted

by Likaj et al.in their comprehensive study of Web framework security [54]. First, we considered the top 5 languages used for Web development in 2022, i.e., JS, Python, Java, C#, and PHP, according to [47], and then selected the most used frameworks from this pool. For this purpose, we used the GitHub metrics *watch*, *fork*, and *stars*, and picked the top 10 of each category. This selection resulted in a total of 12 frameworks as illustrated in Appendix C. We disregarded the *Meteor* framework as it implements cookie-less user sessions, which are out of the scope of this study.

6.1 Methodology

We performed a manual analysis of the selected frameworks, aimed at exposing Web session integrity vulnerabilities that derive from the composition of session management and CSRF protection mechanisms. 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 native libraries were not available, we used external libraries that are widely used by the community, thus being considered the *de facto* standards. In 2 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. As a result, we have reproducible PoCs that we intend to release publicly.

We performed a coordinated disclosure of the identified vulnerabilities between May and July 2022, and assisted framework developers to understand the threat model and 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 concerning 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 libraries used to implement the login and CSRF functionalities, as well as the adopted CSRF protection mechanisms. The table also shows the outcome of our disclosure process, reporting the delta before and after our disclosure (denoted with an arrow symbol). Out of the 12 analyzed frameworks, we identified 11 frameworks supporting the synchronizer token pattern, among which 6 were found vulnerable to our novel CORF token fixation attack (pre-login). Furthermore, 5 frameworks

implemented the double submit pattern, resulting vulnerable to the post-login attack variant. We also discovered 2 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 a selection of case studies of our CORF token fixation attack (pre-login) (see Sec. 5) and corresponding bypasses of 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 reuse the CSRF token issued for the attacker's session following the steps described in Fig. 5.

6.2.1 Express, Koa, and Passport

Several frameworks based on Node.js integrate with the Passport authentication middleware to support authenticated user sessions. Express natively integrates with Passport, while Koa requires an additional Passport middleware (koa-passport). CSRF protection is implemented by the expressjs/csurf CSRF token middleware in Express. Koa uses a different middleware called koa-csrf. Both 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. 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` 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 with the website.

Disclosure. We reported this vulnerability to the Passport developer, who promptly fixed it in version 0.6.0 by clearing all attributes from the session object after login. 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. Unfortunately, the new version of koa-passport middleware (6.0.0, published on February 2023) did not adopt the best practices implemented in the updated Passport release and remains vulnerable. We are currently in touch with the developers to identify an effective mitigation.

6.2.2 Symfony

Symfony provides native user management and relies on the official library symfony/security-csrf to support CSRF pro-

Framework	Version	Language	Loginlib@version	CSRFlib@version	CSRF Protections		Vulnerabilities		
					SP	DS	PreLogin	PostLogin	Session Fixation
Express [3]	4.18.1	JS	passport@0.5.3 [49]	expressjs/csurf@1.11.0 [4]	●	◐	🐛⇒✔⁻	🐛⇒🐛¹	🐛⇒✔⁻
Koa [9]	2.13.4	JS	koa-passport@4.1.4 [12]	koajs/csrf@5.0.1 [10]	●	▬	🐛	▬	✔
Sails [14]	1.4.4	JS	in cookies as in docs	native (expressjs/csurf@1.10.0) [4]	●	▬	🐛⇒🐛⁺	▬	🐛⇒🐛⁺
Django	3.2.13	Python	native	native	◐	●	✔	🐛	✔
Flask [6]	2.1.3	Python	flask-login@0.6.1 [7]	flask-wtf@1.0.1 [8]	●	▬	🐛⇒W	▬	✔
Tornado [17]	6.2.0	Python	in cookies as in docs	native	▬	●	▬	🐛⇒W	✔
Laravel	8.57.0	PHP	native	native	●	▬	✔	▬	✔
Symfony [16]	5.3.1	PHP	native	symfony/security-csrf@5.3.0 [15]	●	▬	🐛⁺⇒✔	▬	✔
CodeIgniter4 [1]	4.2.1	PHP	codeigniter4/shield@v1.0.0-beta [2]	native	◐⇒●	●⇒▬	🐛⇒✔	🐛⇒▬	✔
Yii2 [20]	2.0.45	PHP	native	native	◐	●	✔	🐛⇒W	✔
ASP.NET Core		C#	native	native	●	▬	✔	▬	✔
Spring	2.5.2	Java	spring-security-core	spring-security-core	●	▬	✔	▬	✔

Table 5: Analyzed Web frameworks, and their respective login and CSRF libraries.
● = default, ◐ = available, ✔ = unaffected, 🐛 = vulnerable, ▬ = not implemented.
✔⁻ = safe but there are insecure options, 🐛⁺ = vulnerable but there are secure options, W = still discussing with developers
🐛¹ = csurf was deprecated as a result of a simultaneous disclosure

tection. Symfony supports three different *strategies* to handle session identifiers and session content while authenticating users. The default strategy (MIGRATE) regenerates the session identifier upon login, but preserves the remaining session attributes. Consequently, since the CSRF secret is not pruned upon login 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, our 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.

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 [31] library for the ProVerif [35] protocol verifier, which enables automated security proofs for Web applications.

Our formalization focuses on the 6 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, using the passport library, and, for instance, express and sails also share the CSRF protection, implemented by 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 defined in Sec. 3, essentially considering cookies with no integrity w.r.t. same-site attackers, i.e., a more powerful attacker model than **SS-HOST-S**. This over-approximation results in 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* [76] 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.

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

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 and Express (see session fixation attacks, Appendix D.3). With these modifications, we obtain four fixed models for which ProVerif proves that our correspondence property is valid. Notice that in 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. For space reasons, we refer the reader to Appendix B for the details on the formalization of Web frameworks.

8 Related Work

Cookie Integrity. Several studies have focused on the issue of cookie integrity, with a particular emphasis on session integrity [37, 38, 40, 42, 62, 78]. In their seminal work, Bortz et al. [37] introduce the related-domain attacker model, emphasizing the lack of strong integrity properties for cookies and proposing a mechanism, named *origin cookies* to bind cookies to specific origins. The __Host- prefix mechanisms builds on this proposal and has been integrated into modern browsers [43]. Other studies proposed the use of browser extension to, e.g., transparently strip session (cookie) identifiers from network requests to avoid session hijacking [38, 62]; others, instead, focused on the server-side. Calzavara et al. [40], for instance, propose a type system for verifying session integrity of PHP code against a variety of attackers, including the Web, network and related domain attackers. These works, except for [78], do not assess the implications of the lack of cookie integrity for real world application. Zheng et al. [78] 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. More recently, Squarcina et al. [68] measured and quantified the threats posed by related domain attackers on web application security. In their study of cookies, they discovered the majority of the cookies on sites vulnerable to subdomain takeover do not have integrity against related domain attackers. The authors highlight that, although representing the only solution to improve cookie integrity, 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 in the adoption of __Host- cookies in the last 2 years, especially in the lower-ranked websites.

Analysis of Web Frameworks CSRF Protections. The most widely used approach for defending against CSRF attacks consists in attaching additional information to the request, e.g., a random token, that servers can use to discriminate legitimate usage from forged requests [34, 54, 64]. The main variants of this approach (STP, DSP) are affected by implementation pitfalls that may impact the efficacy of the protection. This is particularly relevant if such oversights in the implementation of the defense are included, in the form of libraries or web framework, in a large number of applications. Likaj et al. [54] evaluate the mechanisms implemented by major Web frameworks, quantifying their exposure to CSRF attacks that results from 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 [54]), originating from the composition of different libraries, may lead to a bypass of the protection in presence of same-site attackers. For instance, the new attack we discover (CORF token fixation) bypasses the Flask framework protections, which were 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 [43].

References

[1] CodeIgniter 4 web framework. https://codeigniter.com/user_guide/index.html.

[2] codeigniter4/shield: Authentication and authorization for CodeIgniter 4. https://codeigniter4.github.io/shield/.

[3] Express – Node.js web application framework. https://expressjs.com/.

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

[5] expressjs/session: Simple session middleware for express. https://github.com/expressjs/session.

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

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

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

[9] Koa – next generation web framework for node.js. https://koajs.com.

[10] koajs/csrf: CSRF tokens for koa. https://github.com/koajs/csrf.

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

[12] rkusa/koa-passport: Passport middleware for koa. https://github.com/rkusa/koa-passport.

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

[14] Sails.js – realtime MVC framework for node.js. https://sailsjs.com/.

[15] symfony-csrf. https://github.com/symfony/security-csrf.

[16] Symfony, high performance PHP framework for web. https://symfony.com/.

[17] Tornado web server. https://www.tornadoweb.org/.

[18] The Web Platform Tests project. https://web-platform-tests.org/.

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

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

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

[22] 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.

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

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

[25] 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.

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

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

[28] Martın Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*. ACM, January 2001.

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

[30] HTTP Archive. The http archive. https://httparchive.org/.

[31] C. Bansal, K. Bhargavan, and S. Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *CSF 2012*. IEEE.

[32] Adam Barth. HTTP State Management Mechanism. RFC 6265, Internet Engineering Task Force, 04 2011.

[33] Adam Barth. The Web Origin Concept. RFC 6454, Internet Engineering Task Force, 12 2011.

[34] Adam Barth, Collin Jackson, and John C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS 2008*, pages 75–88, 2008.

[35] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *CSFW 2001*. IEEE.

[36] Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre. Proverif 2.00: Automatic cryptographic protocol verifier, user manual and tutorial.

[37] Andrew Bortz, Adam Barth, and Alexei Czeskis. Origin Cookies: Session Integrity for Web Applications. In *W2SP*, 2011.

[38] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Wilayat Khan. Cookiext: Patching the browser against session hijacking attacks. *J. Comput. Secur.*, 23:509–537, 2015.

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

[40] Stefano Calzavara, Riccardo Focardi, Niklas Grimm, Matteo Maffei, and Mauro Tempesta. Language-based web session integrity. *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, pages 107–122, 2020.

[41] Stefano Calzavara, Riccardo Focardi, Marco Squarcina, and Mauro Tempesta. Surviving the Web: A Journey into Web Session Security. *ACM Computing Surveys (CSUR)*, 50(1):13:1–13:34, 2017.

[42] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. Sub-session hijacking on the web: Root causes and prevention. *J. Comput. Secur.*, 27(2):233–257, jan 2019.

[43] L. Chen, S. Englehardt, M. West, and J. Wilander. Cookies: HTTP State Management Mechanism (IETF Draft). RFC 6265bis, Internet Engineering Task Force, 11 2022.

[44] 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.

[45] 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.

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

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

[48] Google. Chrome ux report. https://developer.chrome.com/docs/crux/.

[49] Jared Hanson. Passport – simple, unobtrusive authentication for node.js. https://www.passportjs.org/.

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

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

[52] Johannes Henkel and Barry Pollard. Adding rank magnitude to the crux report in bigquery. https://developer.chrome.com/blog/crux-rank-magnitude/, 2021.

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

[54] Xhelal Likaj, Soheil Khodayari, and Giancarlo Pellegrino. Where we stand (or fall): An analysis of csrf defenses in web frameworks. In *24th International Symposium on Research in Attacks, Intrusions and Defenses, RAID '21*, 2021.

[55] Daiping Liu, Shuai Hao, and Haining Wang. All Your DNS Records Point to Us: Understanding the Security Threats of Dangling DNS Records. In *CCS*, 2016.

[56] Rich Lundeen. The deputies are still confused. https://media.blackhat.com/eu-13/briefings/Lundeen/bh-eu-13-deputies-still-confused-lundeen-wp.pdf, 2013.

[57] 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.

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

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

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

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

[62] Nick Nikiforakis, Wannes Meert, Yves Younan, Martin Johns, and Wouter Joosen. Sessionshield: Lightweight protection against session hijacking. In Úlfar Erlingsson, Roel Wieringa, and Nicola Zannone, editors, *Engineering Secure Software and Systems*, pages 87–100, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[63] nVisium. OWASP Top 10 2007-2017: The Fall Of CSRF. https://blog.nvisium.com/p139, 2018.

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

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

[66] 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.

[67] Kapil Singh, Alexander Moshchuk, Helen J. Wang, and Wenke Lee. On the incoherencies in web browser access control policies. In *2010 IEEE Symposium on Security and Privacy*, pages 463–478, 2010.

[68] Marco Squarcina, Mauro Tempesta, Lorenzo Veronese, Stefano Calzavara, and Matteo Maffei. Can I Take Your Subdomain? Exploring Same-Site Attacks in the Modern Web. In *30th USENIX Security Symposium*, 2021.

[69] The Chromium Projects. SameSite Updates. https://www.chromium.org/updates/same-site, 2020.

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

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

[72] web.dev. Schemeful same-site. https://web.dev/schemeful-samesite/.

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

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

[75] John Wilander. Advanced csrf and stateless anti-csrf. https://owasp.org/www-pdf-archive//AppSecEU2012_Wilander.pdf, 2012.

[76] Thomas Y. C. Woo and Simon S. Lam. A semantic model for authentication protocols. *Proceedings 1993 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 178–194, 1993.

[77] Michael Zalewski. The tangled web: A guide to securing modern web applications. 2011.

[78] X. Zheng, J. Jiang, J. Liang, Haixin Duan, S. Chen, T. Wan, and N. Weaver. Cookies lack integrity: Real-world implications. *Proceedings of the 24th USENIX Conference on Security Symposium, SEC'15*, pages 707–721, 01 2015.

A Mitigating Post-Login Token Fixations

In 2012, J. Wilander [75] 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 otherwsie (stateless).

The stateful variant is equivalent to a synchronizer 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 [56]. 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 pattern and refreshing the secret upon login.

B Formal Model of Web Frameworks

In the following, we describe our modeling of Web framework using the applied pi calculus [28], introducing only the parts of the formalism relevant for the understanding of the model. In particular, Table 6 reports the syntax for processes, which, in addition to the standard (simple) processes, includes tables (or databases) operations for storing (*insert*) and retrieving (*get*) persistent data [36]. Additionally, we use an extended term language that supports functions and records, to ease the definition of the framework API.

$P, Q :=$ Processes
  0 null process
  $P \mid Q$ parallel composition
  $!P$ replication
  $\nu n.P$ name restriction
  $c(x).P$ message input from channel c
  $\overline{c}\langle M \rangle.P$ message output to channel c
  if M **then** P **else** Q conditional (else is optional when Q is 0)
  let $x = M$ **in** P term evaluation, local variable definition
  insert$(d(M_1,\ldots,M_n)).P$ insert record into table d
  get$(d(M_1,\ldots,M_n)).P$ read record from table d
  $R(M_1,\ldots,M_2)$ macro usage

Table 6: Syntax of processes.

We define a framework as a set of functions and type definition for handling user management and CSRF protection.

$$Framework := \{ \quad \textbf{type } Session$$
$$empty\text{-}session : unit \to Session$$
$$valid\text{-}session : Session \to bool$$
$$session\text{-}from\text{-}cookie : Cookie \to Session$$
$$session\text{-}to\text{-}cookie : Session \to Cookie$$
$$is\text{-}loggedin : Session \to bool$$
$$login\text{-}user : (Session, User) \to Session$$
$$logout\text{-}user : Session \to Session$$
$$generate\text{-}token : Session \to Session$$
$$serialize\text{-}token : Session \to CSRFToken$$
$$validate\text{-}token : (CSRFToken, Session) \to Bool \quad \}$$

In particular, each framework model provides definitions for:

Session. The implementation for the *Session* abstract data type and the functions to construct, validate and serialize such session to a cookie: *empty-session*, *valid-session*, *session-from-cookie*, *cookie-of-session*. The session could be stored in the server or client side: in the first case, *session-from-cookie* fetches from the server storage the session corresponding to the session id in the cookie, in the second case, the function decodes the cookie value into a session that can be later validated with *valid-session*. Similar considerations apply for the inverse operation.

User management. The implementation of the user management functions such as *is-loggedin*, *login-user*, *logout-user*. Such function modify the current session with the status of the user if, e.g., the credentials are valid.

CSRF protection. The implementation of the generation and validation of CSRF tokens: *generate-token*, *serialize-token*, *validate-token*. These function may rely on implementation specific-types for the CSRF secret that can be stored in the session. The *generate-token* function generates the CSRF secret, possibly updating the session; similarly, the validation and serialization of the CSRF token may depend on the secret stored in the session (see synchronizer pattern, Sec. 2.3)

The functions that are part of the *Framework* API are used in a generic model of a web application composed of 3 endpoints, /login, /logout, /action, supporting multiple HTTP methods. The /login and /action endpoints return a CSRF token protected form for GET requests, and execute the corresponding action, e.g., checking user credentials after validating the token, for POST requests. The /logout endpoint only supports POST requests. We model the application as an applied pi calculus process, consisting of the parallel composition of 5 *handler processes*, one for each combination of endpoint and HTTP method. We discuss in the following the definition of the application, focusing on the process handling POST requests to the /action endpoint.

$$App(Host, \{session\text{-}from\text{-}cookie, is\text{-}loggedin, validate\text{-}token, \ldots\}) := \ldots \mid$$
$$!(httpServerRequest(((\texttt{https://}, Host, \texttt{/action}), headers, POST(token), c)).$$
$$\textbf{let } s = session\text{-}from\text{-}cookie(cookie(headers)) \textbf{ in}$$
$$\textbf{if } vaid\text{-}session(s) \wedge is\text{-}loggedin(s) \wedge validate\text{-}token(s, token) \textbf{ then}$$
$$\textbf{event}(app\text{-}action\text{-}successful(cookie(headers), token)).$$
$$\overline{httpServerResponse}\langle((\texttt{https://}, Host, \texttt{/action}), httpOk, c)\rangle)$$

The *App* process takes as parameter the host in which the application is running and an instance of a framework. The */action* handler reads from the *httpServerRequest* channel, provided by the WebSpi [31] library, a POST requests to the URL composed of the https scheme, the application host and the /action path. The obtained request includes the HTTP *headers* and the submitted CSRF *token*. The handler then builds a session from the received cookie header using the framework function *session-from-cookie*. When the session is valid, and belongs to a logged-in user, the CSRF *token* is validated with the *validate-token* function, executing the event *app-action-successful* upon successful validation. The process then returns to the client the *httpOk* response by writing it to the *httpServerResponse* channel.

The generic application model *App* is run on the example.com host in parallel with a same-site attacker *Attacker*, hosted on the sibling domain attacker.example.com. Additionally, the application is visited by an unbounded number of distinct registered users, modeled by the *UA* process. We use the *Users* table to store and validate registered users for the *App* application.

$$Browser \mid HttpServer \mid$$
$$Attacker(\texttt{attacker.example.com}) \mid$$
$$App(\texttt{example.com}, Framework) \mid$$
$$!(\nu id).(\nu pwd).\textbf{insert}(Users(id, pwd)).UA(id, pwd)$$

The *Browser* and *HttpServer* processes are part of the WebSpi library and provide browser and server functionality, for example encrypting/encoding and decrypting/decoding requests from the network and sending them to the correct application on the *httpServerRequest* and *httpServerResponse*, or attaching cookies to browser requests.

The process modeling users *UA* follows a similar structure to the *App* process, consisting in the parallel composition of all possible user actions. We focus on the sub-process modeling the submission of the token-protected form.

17

$$UA(id, pwd) := (\nu b : Browser).(\dots \mid$$
$$newPage_b((p, \texttt{example.com/action}, form(token))).$$
$$\mathbf{event}(app\text{-}action\text{-}begin(b, token)).$$
$$\overline{pageClick_b}\langle(p, \texttt{example.com/action}, POST(token))\rangle)$$

The process reads from the *newPage_b* channel, which is defined in WebSpi and models the loading of a page in the user (private) browser. The obtained page from example.com/action includes a form containing a CSRF *token*. The process then executes the event *app-action-begin* before submitting the form (*pageClick_b*), performing a POST request which includes the *token*. This event is only performed by the user modeled by the *UA* process before the (explicit) form submission, so it will not be present in traces in which the attacker performs a CSRF.

C   Web Framework Analysis

Table 7 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 12 frameworks.

D   Web Framework Vulnerabilities

In the following, we report on the full list of vulnerabilities we discovered in the analyzed Web frameworks.

D.1   Synchronizer Token Pattern Issues

D.1.1   Sails

Sails does not implement a login handler function, however it ships with a generator [13] that bootstraps a template application providing a user-management service based on express-session [5]. 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, and for this reason, we consider this unsafe code pattern to be inherited by real-world websites.

Disclosure. This unsafe code pattern was disclosed 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. 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.

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 7: Web development frameworks from [54] ranked according to GitHub metrics as of April 8, 2022.

D.1.2   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 [19]. 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.

D.1.3 CodeIgniter4

CodeIgniter4 provides user management via the (official) library codeigniter4/shield, while CSRF protection is included natively and can be easily enabled. CodeIgniter4 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. Even when the synchronizer token pattern is enabled, applications based on this framework are vulnerable to the CORF token fixation (pre-login) attack when the CSRF secret is not refreshed at login. However, we discovered that even regenerating the CSRF secret at login via the `security.regenerate = true` option does not prevent the attack.

CodeIgniter4 sessions objects are stored on the server and contain CSRF secrets as attributes called `csrf_test_name`. When a user accesses https://example.com, 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 https://example.com/login to 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 codeigniter4/shield that 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, (iii) disables the double submit pattern. CVE-2022-35943 was issued for this vulnerability.

D.2 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.

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 vulnerabilities associated with the double submit pattern (Django).

The remaining vulnerabilities were communicated to the developers of the 3 frameworks. CodeIgniter4/shield discontinued the double submit pattern, relying now only on the synchronizer token pattern as a more robust CSRF protection. As for Tornado and Yii2, we are still actively discussing with the respective developers an acceptable tradeoff between compatibility and security, while minimizing the required framework changes.

D.3 Session Fixation Vulnerabilities

In our study, we also found 2 frameworks that were 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. 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 is enough to prevent session fixation, but it is not enough to mitigate the CORF token fixation attacks if CSRF secret values still propagate unchanged to the authenticated session.

D.3.1 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 express-session module to generate a new `sessionId` after a successful login. CVE-2022-25896 was issued for this vulnerability.

D.3.2 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 login interface/module, it provides an application template that bootstraps a project, and consequently unsafe code patterns 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 app, although the addition of the optional `__Host-sails.sid` in production mode described before mitigates the impact of this attack.