Where We Stand (or Fall): An Analysis of CSRF Defenses in Web Frameworks

Xhelal Likaj
Saarland University
Saarbruecken, Germany
xhelallikaj20@gmail.com

Soheil Khodayari
CISPA Helmholtz Center for Information Security
Saarbruecken, Germany
soheil.khodayari@cispa.saarland

Giancarlo Pellegrino
CISPA Helmholtz Center for Information Security
Saarbruecken, Germany
gpellegrino@cispa.saarland

ABSTRACT

Cross-Site Request Forgery (CSRF) is among the oldest web vulnerabilities that, despite its popularity and severity, it is still an understudied security problem. In this paper, we undertake one of the first security evaluations of CSRF defense as implemented by popular web frameworks, with the overarching goal to identify additional explanations to the occurrences of such an old vulnerability. Starting from a review of existing literature, we identify 16 CSRF defenses and 18 potential threats agains them. Then, we evaluate the source code of the 44 most popular web frameworks across five languages (i.e., JavaScript, Python, Java, PHP, and C#) covering about 5.5 million LoCs, intending to determine the implemented defenses and their exposure to the identified threats. We also quantify the quality of web frameworks' documentation, looking for incomplete, misleading, or insufficient information required by developers to use the implemented CSRF defenses correctly.

Our study uncovers a rather complex landscape, suggesting that while implementations of CSRF defenses exist, their correct and secure use depends on developers' awareness and expertise about CSRF attacks. More than a third of the frameworks require developers to write code to use the defense, modify the configuration to enable CSRF defenses, or look for an external library as CSRF defenses are not built-in. Even when using defenses, developers need to be aware and address a diversity of additional security risks. In total, we identified 157 security risks in 37 frameworks, of which 17 are directly exploitable to mount a CSRF attack, leveraging implementation mistakes, cryptography-related flaws, cookie integrity, and leakage of CSRF tokens—including three critical vulnerabilities in CakePHP, Vert.x-Web, and Play. The developers' feedback indicate that, for a significant fraction of risks, frameworks have divergent expectations about who is responsible for addressing them. Finally, the documentation analysis reveals several inadequacies, including not mentioning the implemented defense, and not showing code examples for correct use.

CCS CONCEPTS

• **Security and privacy → Web application security**;

KEYWORDS

CSRF, Defenses, Web Frameworks

ACM Reference Format:
Xhelal Likaj, Soheil Khodayari, and Giancarlo Pellegrino. 2021. 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), October 6–8, 2021, San Sebastian, Spain.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

RAID '21, October 6–8, 2021, San Sebastian, Spain

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Cross-Site Request Forgery (CSRF) is among the oldest web vulnerabilities, consistently ranked as one of the top ten threats to web applications [88]. Successful CSRF exploitations could cause remote code execution [111], user accounts take-over [85, 87, 90, 122], or compromise of database integrity—to name only a few instances. Developers can protect web applications from CSRF attacks by implementing one of the many client-side and server-side defense mechanisms proposed by the research community (e.g., [99, 104, 112, 114, 124, 126, 133, 134]). Alternatively, developers can use off-the-shelf CSRF defenses as implemented by web frameworks, such as the ones provided by popular frameworks like Django for Python, Spring for Java, and ASP.NET for C#. Web frameworks are key components for developing web applications, providing convenient and powerful abstractions to separate low-level functionalities, e.g., databases and web page generation, from the application's logic. However, such a convenience comes at a great cost: a vulnerability in the framework will negatively affect the security of many web applications.

CSRF vulnerabilities are a major concern for web applications, with a steep increasing number of reported instances every year [58]. Yet, this class of vulnerabilities have received a marginal attention by the research community, where most of the previous effort focused largely on defense mechanisms (e.g., [99, 104, 112, 114, 124, 126, 133, 134]) and vulnerability detection (e.g., [101, 122, 129]. To date, we know none-to-little about the security of the CSRF defense implementations and their susceptibility against improper use.

In this paper, we undertake, to the best of our knowledge, the first security evaluation of CSRF defense as implemented by popular web frameworks, comprehensively and systematically covering the source code, defenses' design, documentation, and the operational aspects, with the overarching goal to identify additional explanations to the steady increase of such an old vulnerability. Starting with a thorough review of academic and non-academic literature, we enumerate existing CSRF defenses and threats against them, identifying 16 distinct defenses and 18 potential threats. Then, we

evaluate the source code of the 44 most popular web frameworks across the top five programming languages for web applications (i.e., JavaScript, Python, Java, PHP, and C#) covering about 5.5 million LoCs, intending to determine the implemented defenses and their exposure to the identified threats. Finally, we also quantify the quality of web frameworks' documentation, looking for incomplete, misleading, or insufficient information required by developers to implement and use the implemented CSRF defenses correctly.

Insights—Our study uncovers a complex landscape, suggesting that while implementations of CSRF defenses exist, their correct and secure implementation depends on developers' awareness about CSRF attacks, threats to CSRF defenses, and specific behaviors of the implementations.

Insight #1: Almost all frameworks offer a CSRF defense—either built-in or via external libraries, with the majority enforcing a token-based protection mechanism, one of the most robust CSRF defenses. For example, over 53% and 41% of the frameworks use double submit cookies and synchronizer tokens to mitigate CSRF attacks, respectively. Also, *all* frameworks (except Li3) use robust CPRNG for token generation. Even more promising, almost half of the frameworks (i.e., 19) enforce a defense-in-depth mechanism by applying two or more layers of defenses in sequence. For example, we observe that ten frameworks enable SameSite cookies by default—a promising defense-in-depth mechanism which can mitigate a number of CSRF attacks. Similarly, we observed that seven frameworks give state to the double submit cookie that is stateless in nature, making it immune to cookie tossing and jar overflow attacks. Finally, we noticed that frameworks offer defenses that can protect web applications from attackers with stronger capabilities than web attackers. For example, the cryptographic operations conducted in most frameworks use secure algorithms, sufficiently long tokens, or encryption that makes a CSRF attack even more difficult.

Insight #2: In total, 11 frameworks provide an enabled-by-default defense. In all other frameworks, developers need to write glue code to enable and use frameworks' defenses correctly. More specifically, more than a third of the frameworks (36%, i.e., 16 out of 44) do not provide any built-in defense—including popular ones such as Express, Flask, and Spring—requiring developers to search for external libraries or alternatively implement their defense. In most of these cases (i.e., 11 out of 16 frameworks), the frameworks' documentation suggests an external CSRF library, whereas, for the remaining ones (i.e., five out of 16 frameworks), it does not. Even if the vast majority of frameworks (67%, i.e., 28 out of 44) provide built-in defenses, all frameworks require developers to know the CSRF defense operations, the correct sequence of operations, and the security-sensitive server-side endpoint that need to be protected. Finally, for 17 of these 28 frameworks, developers need to enable CSRF defenses explicitly as they are disabled by default.

Insight #3: Implemented defenses—either built-in or via external libraries—can contain vulnerabilities or implement weak solutions. The deployment of robust defenses requires developers to be aware of additional threats that can weaken or even defeat CSRF defenses and their ability to develop or configure ad-hoc solutions. In total, we identified 157 distinct security risks in all but four frameworks (Falcon, Web2py, Apache Wicket, and Falcon), of which 17 are directly exploitable, affecting the building blocks of token-based CSRF defenses—the most widely implemented defense—such as token generation, transportation, and validation. In general, while token-based defenses use robust generation algorithms (i.e., cryptographically secure pseudorandom number generators, cryptographic algorithms, and implementations), the randomness of tokens is sub-par. Furthermore, if tokens leak, virtually all implemented defenses are exposed to token replay attacks as they reuse tokens across multiple requests. Only one framework implements a per-request unique token, rendering replay hard in practice. Also, we identified and already reported three critical vulnerabilities in CakePHP, Vert.x-Web, and Play, leading to a complete CSRF defense bypass. Finally, both our security analyses and developers' feedback reveal that, for a considerable fraction of security risks, frameworks' developers have divergent expectations from applications' developers about their responsibility for addressing the risks.

Insight #4: Lastly, the documentation of most of the frameworks with a CSRF defense is inadequate. For example, the documentation does not explain the implemented defense name or any kind of description (16 out of 39 frameworks with a defense), does not show the correct use via code examples (seven out of 39 frameworks), nor adequately describe the API calls and the configuration space of the CSRF defense (eight out of 39 frameworks).

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

- We conduct, to the best of our knowledge, the first security evaluation of CSRF defenses as implemented by popular web frameworks.
- We conduct an exhaustive review of the existing proposed defenses and potential threats, identifying 16 defenses and 18 security threats.
- We identify 44 popular web frameworks across the most popular programming languages for web applications (i.e., JavaScript, Python, Java, PHP, and C#). We evaluate 5.5 million LoCs, to identify implemented defenses and their exposure to the 18 security threats.
- We discovered 157 distinct security risks in 37 popular web frameworks, of which 17 are directly exploitable and three severe instances that can lead to a complete defense bypass in CakePHP, Vert.x-Web, and Play.
- We ranked the documentation of the web frameworks against six quality attributes, and show that documentation is largely inadequate and poorly supports developers in building robust defenses.

2 PROBLEM STATEMENT

2.1 Cross-Site Request Forgery (CSRF)

In a CSRF attack, an adversary tricks a victim's web browser into sending an authenticated HTTP request to a vulnerable web application in order to execute a state-changing operation, without the victim's consent or awareness. Targets of CSRF attacks can be, for example, changing user account privileges or credentials [85, 87, 90, 101], remote execution of arbitrary code [111, 122],

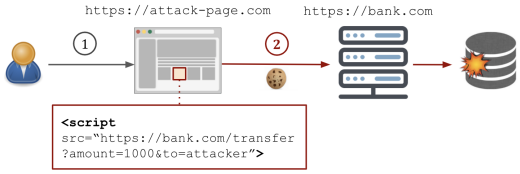

Figure 1: The workflow of a CSRF attack.

or illicit money transfer [122, 134]. A CSRF attack typically comprises of two phases: preparation and attack. During the preparation phase, the malicious code is added to the attacker-controlled website, whose aim is to submit a correct (i.e., with all expected parameters) cross-origin HTTP request to the target web application (e.g., `bank.com`). This can be achieved with a variety of methods, e.g., with a `script` tag, self-submitting HTML form, other HTML tags with the `src` attribute, or the JavaScript Fetch API [41]. In the attack phase, the attacker lures an authenticated victim into visiting the attack page (step one). Then, the malicious code in the attack page tricks the victim's browser into sending a cross-origin HTTP request to the target web application (step two). The browser complies with the Same-Origin Policy (SOP) for cookies and automatically includes the authentication header (e.g., HTTP session cookies) in the outgoing request [81, 104]. Finally, the cross-origin request triggers a state-changing operation (e.g., a money transfer), which the vulnerable server will execute due to the session cookies' presence. Figure 1 exemplifies the steps of this attack.

2.2 Research Questions

Despite the popularity of CSRF attacks, little has been done to understand how CSRF defenses are implemented and the hurdles when using them in practice. This paper takes the first step in this direction and explores the security of CSRF implementations of web frameworks to shed some light on possible causes and factors hampering web applications' security. More specifically, we aim to answer the following questions:

RQ1: Available CSRF Defenses—Over the past decades, we have seen a plethora of different ideas to protect from CSRF attacks. Still, we lack a comprehensive survey and categorization of proposed defenses. More importantly, we know little about the ones that are used in practice by web applications.

RQ2: Security of Existing Defenses—Prior work has mostly focused on proposing new defenses or devising new CSRF detection techniques (e.g., [101, 122, 129]). Yet, we do not know what are common mistakes introduced by developers when implementing CSRF defenses. The second question of this paper intends to answer that question by reviewing the source code of popular web frameworks and libraries against attacks targeting both CSRF defenses or other components whose compromisation might weaken them.

RQ3: Developers' Challenges—Even when frameworks implement robust CSRF defenses, their incorrect use can also severely impact web applications' security. Incorrect use may stem from under-documented APIs, missing or misleading documentation, wrong or missing code examples, and insecure defaults and configuration (see, e.g., [96, 107, 120, 121, 132]). Our third question intends to explore the extent to which the implementations of CSRF defenses might induce web developers in implementing insecure web applications.

3 METHODOLOGY

To answer our research questions presented in §2.2, we decompose our study into five main steps, organizing it into preparatory steps and analysis.

Preparatory Steps. The preparatory steps (i) identify relevant frameworks used by developers (§3.1), (ii) survey academic and non-academic literature to enumerate and classify proposed CSRF defenses (§3.2), and (iii) review existing literature to create a comprehensive list of threats against CSRF defenses (§3.3).

Analysis. Our analysis combines manual code review and dynamic testing (§3.4), and documentation review (§3.5). During code review and dynamic testing, we identify defenses used in practice and their robustness against our list of threats, answering to RQ1 and RQ2. Then, to review the documentation, we define measurable quality criteria of the web framework documentation, and rank frameworks accordingly.

3.1 Identification of Popular Web Frameworks

In this paper, we focus on analyzing CSRF defenses in web frameworks. To find popular frameworks, we first identify the top five web programming languages from GitHub's 2019 Octoverse report [42], i.e. JS, Python, Java, PHP, and C# (ordered). Then, for each language, we identify the top 10 web frameworks. As a first step, we compiled a large list of frameworks for each language using numerous web resources. Then, we quantitatively measure the popularity of all identified frameworks based on these criteria (ordered by importance): the number of uses by other GitHub repositories (GitHub Used By), download statistics in package managers of each language (e.g., npm [61] for JS), number of relevant questions in Stack Overflow [77], and the number of GitHub watches, stars, and forks. Accordingly, we pick the top 10 frameworks for each language. For C# and JS, we only identified five and nine web frameworks, respectively. Thus, our testbed contains a total of 44 web frameworks (see Table 5 in Appendix A).

Finally, for frameworks that do not offer any built-in CSRF defense, we searched in their documentation for official indication on how to protect against CSRF, e.g., by importing external libraries. We label these libraries *official*. If we do not find any official indication, we search on developer communities, such as Stack Overflow, to find the library to use. We label these libraries *unofficial*. In total, we considered 13 libraries (11 official, and two unofficial). The complete list of libraries is provided in Table 6 of Appendix A.

3.2 Survey of CSRF Defenses

As the second step of our study, we compile a comprehensive list of defenses by reviewing the academic works (i.e., [3, 63, 74, 92, 95, 99–101, 104, 112–115, 119, 122–124, 126–128, 128, 129, 131, 133, 134]) and non-academic resources [3, 63, 74, 82]. Then, by dissecting CSRF attacks, we identify four distinct categories of vulnerable behaviors that when removed, a CSRF attack is no longer successful. In total, we identify 16 distinct defenses, each addressing one of the four vulnerable behaviors. We present each defense in §4.

3.3 Threat Analysis

We identified possible threats against CSRF defenses by systematically reviewing academic literature (i.e., [99, 100, 104–106, 109, 110, 114, 118, 126, 129, 130, 133, 134]), OWASP security best-practices [63], the National Vulnerability Database (NVD) [58], and web frameworks' GitHub issues. Of these, we consider in-scope those threats that can be exploited by a web attacker and a network attacker. We note that both attacker models are consistent with prior work in the area of CSRF defenses [99] and attacks [106]. However, we observe that the network attacker is, in general, stronger than a web attacker, and many attacks may be considered out of scope for CSRF defenses, i.e., network-level MITM. In this paper, we consider a weak form of the network attacker, e.g., an attacker that can leverage compromised DNS servers and unprotected wireless networks to control the victim users' network connections.

Once we identified relevant threats, we grouped them into four main categories: (i) threats affecting the generation of CSRF tokens, (ii) ways an attacker can obtain a valid CSRF token, (iii) as token-based defenses can rely on cookies, e.g., double submit cookie, we have threats against cookie integrity, and (iv) finally, we have threats affecting the CSRF validation of HTTP requests. In total, we identified 18 distinct threats which we present in §5.

3.4 Analysis of the Code

To the best of our knowledge, there is no single program analysis technique that satisfies three requirements: (1) analyze programs written in multiple programming languages; (2) analyze partial programs such as frameworks and libraries; (3) detect the threats we identified in §5. Accordingly, in this work, we defined a manual methodology that combines code review, dynamic testing, and a strict evaluation protocol.

Code Review. Manual code review is the first technique we used to analyze the security of each framework. First, we reviewed the documentation and API specifications. We observed that the documentation is rarely useful in practice. In §7.2, we present an in-depth evaluation of the documentation. Given the insufficient documentation, we searched for all CSRF-related GitHub issues in the framework's repository to gain insight into corner cases and design decisions.

Next, we reviewed the source code using two strategies. In the first review strategy, we examined the code to expand our understanding of frameworks, architectures, and modules. We loaded the source code in an IDE (IntelliJ for Java, VS Enterprise for C#, and VS Code for Python, PHP, and JavaScript), and then reviewed the CSRF-related source code files, following control-flow insensitive navigation. We noted possible entry points for the execution, configuration files, and parameters. In total, with the first strategy, we reviewed 5,585,275 LoC, containing 575,182 of JavaScript, 509,400 of Python, 1,915,669 of Java, 1,062,917 of PHP, and 1,522,104 of C#. Then, we evaluated the workflow of CSRF defenses. Specifically, we prepared a list of sensitive functions, e.g., token generation, leveraging the snippets and APIs from the documentation. Then, we located functions in the code and traced the program execution following a forward control and data-flow sensitive inspection. During the inspection, we collected sensitive functions that could be abused by an attacker, e.g., cryptographic functions. With the second strategy, we reviewed 14,696 of LoC, including 1,632 of JavaScript, 2,164 of Python, 3,661 of PHP, 3,773 of Java, and 3,466 of C#.

Manual Dynamic Testing. During our review, frameworks like Meteor, Vaadin, and Tornado were particularly challenging to analyze due to their complex, poorly-modularized source code. Therefore, we also performed dynamic testing as part of our evaluation for *all* frameworks. We built a basic web application for each framework, equipped it with a simple HTML form for a state-changing HTTP request, and monitored the execution using debuggers and breakpoints to ensure that our code review did not miss alternative executions.

Threats Prioritization. Our evaluation requires to test a large number of threats, which is demanding. Therefore, we prioritized our analysis and examined *all* threats against the top five frameworks for each language, for a total of 25 frameworks. For the remaining 19 frameworks, we only focused on a subset of threats that does not require a special testing environment but can be inferred directly from the source code review (e.g., implementation mistakes, cookie tossing, replay attacks, or BREACH). Furthermore, in this paper, we focus only on the default settings of each framework as it is infeasible to study every possible configuration setting.

3.5 Analysis of the Documentation

Documentation is pivotal for the correct use of CSRF defenses. After retrieving the documentations, we defined six quality criteria that the documentation of CSRF defenses should have, and ranked frameworks accordingly.

Defense Name or Description. The documentation should name and describe the implemented CSRF defense so that developers can properly apply and evaluate its enforcement.

API Specs. Using CSRF defenses may require the use of API functions, and the documentation should provide developers with an adequate description of their functionalities.

Configuration. The implemented CSRF defense may need configuration parameters, e.g., the secret key for encrypted tokens. Accordingly, the documentation should adequately describe the configuration parameters of the CSRF defense.

Code Example. Descriptive text may not be sufficient to convey the correct use of defenses. Often documentation includes snippets of code to illustrate the correct use.

Cryptographic Guarantees. Some CSRF defenses rely on cryptographic algorithms whose security depends on known weaknesses of the algorithms or weak secrets. Ideally, documentation should mention the names of used algorithms and default parameters, e.g., key length and entropy.

General Security Considerations. The documentation should describe known behaviors that can weaken the security guarantees, and provide general security considerations when not protecting state-changing operations against CSRF attacks.

4 SURVEY OF CSRF DEFENSES

In this section, we answer RQ1 by surveying existing literature on CSRF defenses following our methodology of §3.2. In total, we

identified 16 distinct defense mechanisms, each addressing one of the four vulnerable behaviors. Table 1 summarizes the identified defenses. The rest of this section details the defenses in four distinct categories.

4.1 Origin Checks

A distinctive feature of CSRF attacks is that the request's origin differs from the origin of the target. One of the first defenses consists of checking the origin of an HTTP request. For example, the server-side of the web application can check the HTTP request `Referer` and `Origin` headers [99, 114, 129]. Another defense relies on browsers complying with the Cross-Origin Resource Sharing (CORS) [99, 129]. When performing a cross-origin request (COR), browsers can send a so-called *pre-flight* request, whose goal is to check whether the COR complies with the CORS policy of the server-side before sending the actual COR request. However, pre-flight requests are issued only for non-simple requests. A web application can enforce these requests by requiring a custom HTTP request header [63] with each state-changing request and rejecting requests that lack this header.

4.2 Request Unguessability

Another distinctive feature of CSRF attacks is that an attacker can guess all parameters of a request, except for cookies. The second strategy of defense is the addition of unguessable parameters, often called CSRF tokens. For example, a popular CSRF defense is the Synchronizer Token Pattern (Plain Token) [126] which generates random tokens. This generated token is stored on the server-side and tied to the user session. For each incoming sensitive request, the server compares the token in the user session against the token in the request. Alternatively, the server-side can avoid storing tokens and check their validity with the aid of cryptographic primitives. For example, the Encryption-based Token Pattern (Encrypted Token) [63] and HMAC-based Token Pattern (HMAC Token) [63] use encryption and HMAC codes to protect a token containing a timestamp and a user session identifier. The server accepts requests if two conditions hold. First, the server decrypts the token correctly or checks whether the HMAC code is correct. Second, the server verifies that the timestamp is valid and that the session identifier is of the user submitting the request. Instead of using cryptographic primitives, servers can submit a pseudo-random value both as a request parameter and as a cookie. The server will accept the request only if the two values match. An example of such a defense is the Double Submit Cookie [134]. A variant of it is the Triple Submit Cookie [133], which suggests using a random cookie name to prevent attackers from writing cookies in specific scenarios such as XSS or cookie tossing attack.

Another approach is Cookie-less User Session management [55, 82] which relies on the `localStorage` APIs [83] to store user sessions. This defense could be effective since, unlike cookies, the browser does not automatically attach the values stored in web storage to HTTP requests.

4.3 Same-Origin Policy for Cookies

A behavior that is exploited by CSRF attacks is the compliance with SOP for cookies. When the user visits the malicious page, the malicious code will send a request to the target web application and the user's browser will include the cookies in this request. Over the past years, we have seen many defenses suggesting to modify this behavior of the browser, e.g., via proxies (e.g., [104, 114]), browser extensions (e.g., [112, 119, 123, 124]), or both [115]. Instead of relying on third-party components, `SameSite` attribute for cookies [128, 131] can be used. This attribute allows developers to choose among three policies (Strict, Lax, None) specifying when cookies should be included in cross-origin requests. An alternative approach that can be implemented by web applications is to limit the life span of session cookies, forcing frequent log out events [74].

4.4 User Intention

Finally, another crucial feature of CSRF attacks is that web applications cannot determine whether a request is the result of the users' intention. Inferring the user intention is not a trivial problem to solve and existing solutions require the user itself to perform additional steps before accepting a request. Examples of these defenses are user Re-authentication [63], One-time Token [63], solving (re)CAPTCHA challenges [63], and Multi-browser navigation [3, 63, 100].

5 THREAT MODELING

In our review, we focused on four categories of threats: (1) threats affecting the way CSRF defenses generate tokens; (2) threats that can leak the CSRF token; (3) threats against cookie integrity; (4) threats affecting the CSRF validation of requests. In total, we identified 18 distinct threats across all four categories, as presented in Table 1. The rest of this section presents the identified threats. For brevity, we discuss the threats in more coarse-grained groupings and mark each individual threat in *italic*. Also, for each threat, we provide the references to past vulnerabilities that affected CSRF protections or real-world web applications. We use reports from HackerOne (hereafter H1) [43], and CVEs from MITRE [57].

5.1 Token Generation

This section presents the first category of threats against the generation of CSRF tokens.

Weak Generators. An *insecure randomness* vulnerability occurs when developers generate predictable tokens, e.g. as in the recent CVE-2021-26296 [38]. Other example can be found in practice, such as hashing used identifiers [46] or the timestamp [69], or using the time to initialize a pseudo-random number generator (PRNG) [5, 53]. A more robust approach consists of using a cryptographically-secure PRNG (CSPRNG) [4] together with a secret key. Another common threat is caused when using *vulnerable cryptographic libraries*, as in CVEs 2013-2213 [7] and 2015-4056 [11]. We relied on the official documentation of each PRNG [54, 56, 60, 66, 71] to evaluate whether they were secure.

Insufficient Randomness. Even with robust cryptographic functions, developers can still introduce vulnerabilities by specifying short tokens that do not protect sufficiently against brute-forcing, i.e., *insufficient token randomness* [64]. Threats can also originate when generating insufficiently long secret keys for signature and

			Token Gen.				Token Leakage				Cookie Int.		Implementation Mistakes							
Category	CSRF Defense	Defense Source	Insecure Randomness	Insuff. Token Randomness	Insuff. Key Randomness	Vuln. Crypto Libraries	BREACH	Referrer Leakage	Over-perm. CORS Conf.	Timing Attacks on Cmp.	Cookie Tossing	Cookie Jar Overflow	Insec. Token Mapping	Safe HTTP Met. Check	Unsafe HTTP Met. Check	HTTP Method Override	Logical Errors	WS Hijacking	Replay Attack	Faulty Ref./Orig. Check
Origin Checks	`Referer/Origin` Header Check	[63, 99, 114, 129]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	●
	Custom Request Headers	[63, 99, 129]	-	-	-	-	-	-	●	-	-	-	-	-	-	-	●	-	-	-
Request Unguessability	Plain Token	[63, 126]	●	●	-	●	●	●	●	●	-	-	●	●	●	●	●	●	●	-
	Encrypted Token	[3, 63]	●	●	●	●	●	●	●	●	-	-	●	●	●	●	●	●	●	-
	HMAC Token	[3, 63, 99]	●	●	●	●	●	●	●	●	-	-	●	●	●	●	●	●	●	-
	Double Submit	[63, 134]	●	●	-	●	●	●	●	●	-	●	●	●	●	●	●	-		
	Triple Submit	[133]	●	●	-	●	●	●	●	●	●	●	●	●	●	●	-			
	Cookie-less User Sessions	[55]	●	●	-	●	●	●	●	●	-	-	●	●	●	●	●	●	●	-
SOP for Cookies	SameSite Cookies	[92, 95, 128, 131]	-	-	-	-	-	-	-	-	-	●	●	-	-	-	-	-	-	-
	Frequent Log Outs (server-enforced)	[74]	-	-	-	-	-	-	-	-	●	●	-	-	-	-	-	-	-	-
	Browser Extensions	[104, 114, 115]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-
	Server-side Proxies	[112, 115, 119, 123, 124]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-
User Intention	Re-authentication	[63]	-	-	-	●	-	-	-	-	-	-	-	-	-	-	●	-	-	-
	One-time Token	[63]	●	●	-	●	-	-	-	-	-	-	●	-	-	-	●	-	-	-
	(re)CAPTCHA	[63]	●	●	-	●	-	-	-	-	-	-	●	-	-	-	●	-	-	-
	Frequent Log Outs (user-enforced)	[74]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	Multi-browser Navigation	[3, 63, 100]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Legend: ● = potential security risk; - = not applicable;

Table 1: Overview of CSRF defenses and threats. The left part summarizes our survey of CSRF defenses. The right part shows the mapping between each defense and potential threats.

encryption, i.e., *insufficient key randomness*. Examples of insufficient randomness vulnerabilities are CVEs 2012-1598 [6], and 2021-23127 [37].

5.2 CSRF Token Leakage and Abuse

The second category of threats that we considered targets the exchange of the token between the client and server. This category includes attacks where the adversary can exploit weaknesses that allow leaking the token.

Side-channel Attacks. An attacker can leak the tokens through side-channels using attacks like *BREACH* [106], which affects applications that use data compression such as gzip or DEFLATE in HTTP responses. Here, an attacker could trick user's browser into submitting multiple requests and observe length variations due to compression and leak the token. Examples of BREACH vulnerabilities are CVE 2015-2206 [9] and 2014-9720 [10]. Another side-channel can originate from the way applications compare tokens. For example, default string comparison functions such as `strcmp` stop the comparison at the first mismatching character, allowing an external observer to monitor the time increase between comparisons, revealing the correct character. Examples of such a vulnerability are CVEs 2015-6728 [12], 2015-8125 [13], 2015-8623 [14], 2016-10535 [16], and 2018-1000119 [25].

Cross-Domain Referrer Leakage. When transmitting the token to the client, developers can use hidden HTML input forms, JavaScript variables, or custom request headers. However, developers should consider not including tokens as URL parameters as they can be leaked. For example, URLs may appear in the `Referer` HTTP header, disclosing tokens to external websites. Examples of these vulnerabilities are H1 report nos. 342693 [44] and 787160 [48], and CVEs 2016-5739 [18] and 2019-15515 [31].

CORS Misconfiguration. Attackers can also exploit server-side misconfigurations to leak the CSRF tokens. For example, an *over-permissive CORS policy* [2, 102] that sets `Access-Control-Allow-Origin` (ACAO) HTTP header to reflect the request's origin and `Access-Control-Allow-Credentials` (ACAC) to true, allow attackers to read responses' body of cross-origin requests, which is forbidden by default browsers' policies. Accordingly, the attacker could send a cross-origin request to fetch a page with the CSRF token and use it in a CSRF attack. Examples of these vulnerabilities are H1 report nos. 975983 [49], 577969 [47], and 426147 [45] or 2015-9243 [15] and CVEs 2016-10549 [17].

5.3 Cookie Integrity

Token-based CSRF defenses, such as Double and Triple Submit cookie, rely on cookies which an attacker may try to corrupt. For example, if the attacker controls a subdomain of the target web application, the attacker can set or overwrite cookies of the parent domain with attacker-specified values [116], known as *cookie tossing* [116]. A similar result can be achieved by *cookie jar overflow attack* [97], which floods the browser with HTTP requests to set cookies and exploit the limited capacity that browsers' cookie jar have [51]. When this limit is reached, older cookies get evicted [86] and can be replaced with attacker-specified values. Examples of these vulnerabilities are 2016-8615 [21] and CVEs 2019-14998 [30].

5.4 Implementation Mistakes

The final category of threats encompasses improper ways to validate incoming HTTP requests, ranging from incorrect user-token association to missing checks, as presented next.

Missing Checks on HTTP Methods. A robust CSRF verification should enforce CSRF checks on all incoming HTTP requests, irrespective of the request method. For example, GET is a *safe, idempotent* HTTP method that should not be used for state-changing

requests [84]. Yet, this does not prevent a developer from performing state-changing requests using GET which could lead to CSRF attacks. Similarly, developers may not perform CSRF checks on all *unsafe* HTTP methods, e.g., DELETE or PUT [72]. Finally, the problem is aggravated by the *HTTP Method Override* feature, which is used to change the request method. If CSRF checks are only applied to specific HTTP methods, CSRF validation could be bypassed by overriding the request method (see, e.g., 2017-16136 [23] and CVEs 2020-35239 [94]).

Logical Mistakes. Unlike syntactical code errors, logical errors do not trigger a compilation error and might go unnoticed. A simple *logic error* could be using the `OR` operator instead of `AND`, thus accepting a request if only one of the conditions holds (see, e.g., CVEs 2017-0894 [22], 2017-9339 [24], and 2019-12659 [28]). A similar mistake is the *incorrect user-token mapping*. If each user does not have a unique CSRF token for each session, an attacker can obtain the same token as the victim (see, e.g., CVE-2020-11825 [33]) by being a user of the application. Finally, CSRF validation of the request can be erroneous. For example, a *faulty comparison* between the request's origin and the application's origin can occur due to incorrect regular expressions [108]. Examples of this vulnerability are CVEs 2018-6651 [27], 2018-10899 [26], and 2016-6806 [20].

Replay Attacks. These attacks operate under the assumption that the attacker has leaked the CSRF token [105, 122]. In such cases, the attacker can *reuse* the same token to forge a state-changing request until the expiration of the session cookie (see, e.g., CVEs 2014-1808 [8], 2016-6582 [19], and 2020-5261 [36]).

Cross-Site WebSocket Hijacking. Another implementation mistake exploits the way WebSocket (WS) connections are authenticated. If the authentication solely relies on cookies (as opposed to TLS/HTTP authentication), the WS connection can be hijacked by a CSRF attack [125]. Instances of this vulnerability are CVEs 2019-13209 [29], 2019-17654 [32], 2020-14368 [34], and 2020-25095 [35].

6 SECURITY ANALYSIS OF THE IMPLEMENTATIONS

We now present the results of our security analysis on the 44 web frameworks identified in §3.1 against the 18 threats of §5. First, we present an overview of the usage of CSRF defenses (§6.1), and then we present the discovered security risks (§6.2).

6.1 Demographics of CSRF Defenses

Built-in Defenses and Defaults. Our analysis uncovered a ramified and rather complex landscape. First, not all frameworks provide off-the-shelve CSRF defenses. In total, 16 frameworks (about 36%) are shipped without built-in CSRF defenses, including Express, Flask, and Spring, the most popular frameworks in JavaScript, Python, and Java, respectively. For 13 of them, we identified external libraries providing CSRF protection. Then, the remaining 28 frameworks provide built-in CSRF defenses; however, in 17 of them (covering for more than 60% of the frameworks with built-in defenses), CSRF defenses are disabled by default. Both these cases can be problematic if developers are not security-aware and forget to install the necessary libraries or enable the defense correctly.

	Ref./Orig. Header	Plain Token	Encrypted Token	HMAC Token	Double Submit	Triple Submit	SameSite Cookies	Cust. Req. Hdr.	Cookie-less Usr Sess.	One-time Token	(re)CAPTCHA	Frequent Log outs	Re-authentication	Browser extensions	Server-side Proxies	Multi-browser Nav.
Ref./Orig. Header	4	0	0	2	3	0	2	0	0	0	0	0	0	0	0	0
Plain Token	0	18	0	0	0	0	3	0	0	0	0	0	0	0	0	0
Encrypted Token	0	0	4	4	4	0	1	0	0	0	0	0	0	0	0	0
HMAC Token	2	0	4	12	12	0	4	0	0	0	0	0	0	0	0	0
Double Submit	3	0	4	12	22	0	6	0	0	0	0	0	0	0	0	0
Triple Submit	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SameSite Cookies	2	3	1	4	6	0	10	0	0	0	0	0	0	0	0	0
Cust. Req. Hdr.	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Cookie-less Usr Sess.	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
One-time Token	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
(re)CAPTCHA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Frequent Log Outs	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Re-authentication	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Browser Extensions	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Server-side Proxies	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Multi-browser Nav.	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 2: Frequency of the combination of CSRF defenses. Each entry in this symmetric table shows the number of frameworks that use a certain combination.

Few but Popular Frameworks with No Defenses. Overall, five of the 44 frameworks do not have built-in CSRF defenses and the documentation does not suggest any as well. We point out that two of them, Bottle and Spark, are among the top five Python and Java frameworks, respectively.

Implemented Defenses. In total, 39 frameworks can have a CSRF defense as a built-in feature or via official external libraries. For Bottle and Spark, who have no defenses, we identified unofficial libraries via Stack Overflow [50, 75] and internet search [76]. Accordingly, we extended the testbed of frameworks to 41 frameworks by including Bottle and Spark. The vast majority of frameworks implement the Double Submit (i.e., 22) or Plain Token (i.e., 18) defense. The least frequent CSRF defense is Cookie-less user Session, used only by Meteor. This technique is an emerging pattern, where the web storage and custom client-side JS code replace cookies and cookie management policies, respectively. The number of frameworks that use the rest of the defenses are as follows: 12 HMAC Token, 10 SameSite Cookies, 4 Encrypted Token, and 4 *Referer/Origin* Check. We refer interested readers to Appendix A for a complete list of defenses implemented by each framework.

Defense in Depth. Web frameworks may implement multiple CSRF defenses at the same time. For example, almost half of the frameworks (i.e., 19) enforce two or more defenses in sequence. Table 2 shows the frequency of combinations of defenses across web frameworks of our testbed. We observe that Double Submit and HMAC Token are used together more than any other pair of defenses, i.e., in 12 frameworks.

6.2 Vulnerabilities and Security Risks

In total, we discovered 157 security risks affecting 37 frameworks, all of which can be mounted by a web attacker. However, we note that the exploitability of these risks may vary. For example, out of 157, 17 security risks are directly exploitable with one of two HTTP requests and without relying on developers' mistakes or misconfigurations of the targeted web application. We rank these security risks as *severe*. In contrast, 140 security risks can be exploited only under specific circumstances. For example, if a developer makes a

Web Framework	Token Gen.				Token Leakage					Cookie Int.		Implementation Mistakes									Total ○	Total ●	Total ◑
	Insecure Randomness	Insuff. Token Randomness	Insuff. Key Randomness	Vuln. Crypto Libraries	BREACH	Referrer Leakage	Over-perm. CORS Conf.	Timing Attacks on Cmp.		Cookie Tossing	Cookie Jar Overflow	Insec. Token Mapping	Safe HTTP Met. Check	Unsafe HTTP Met. Check	HTTP Method Override	Logical Errors	WS Hijacking	Replay Attack	Faulty Ref./Orig. Check				
JavaScript																							
#1 Express	○	○	◑	○	○	◑	○	○		◑	◑	○	◑	○	○	○	○	◑	-		11		6
#2 Meteor	○	○	-	○	◑	○	○	○		-	-	○	○	○	○	○	○	◑	-		12		2
#3 Koa	○	○	◑	○	○	◑	○	○		-	-	○	◑	○	○	○	●	◑	-		10	1	4
#4 Hapi	○	○	-	○	◑	○	○	◑		◑	◑	-	◑	●	○	○	○	◑	-		8	1	6
#5 Sails	○	○	-	○	○	◑	○	○		-	-	○	◑	○	○	○	○	◑	-		11		3
Python																							
#1 Flask	○	○	◑	○	○	○	○	○		○	○	-	◑	○	○	○	●	◑	●		12	2	3
#2 Django	○	○	-	○	○	○	○	○		◑	◑	-	◑	○	○	○	●	◑	●		10	2	4
#3 Tornado	○	○	◑	○	○	○	○	○		◑	◑	-	◑	○	○	○	○	◑	-		11		5
#4 Bottle	○	○	◑	○	○	○	○	○		◑	◑	-	○	○	○	○	●	○	-		12	1	3
#5 Pyramid	○	○	◑	○	◑	○	○	○		○	○	-	◑	○	○	○	○	◑	●		12	1	4
Java																							
#1 Spring	○	○	-	○	◑	◑	○	◑		-	-	○	◑	○	○	○	◑	◑	-		8		6
#2 Play	○	○	◑	○	○	◑	◑	○		○	○	-	◑	○	○	○	●	◑	-		10	1	5
#3 Spark	○	○	-	○	◑	○	○	◑		-	-	○	◑	○	○	○	●	◑	-		9	1	4
#4 Vert.x-Web	○	○	◑	○	◑	○	○	◑		◑	◑	-	◑	○	○	●	○	◑	-		8	1	7
#5 Vaadin	○	○	-	○	○	◑	○	◑		-	-	○	○	○	○	○	○	◑	-		11		3
PHP																							
#1 Laravel	○	○	◑	○	◑	○	○	○		-	-	○	◑	○	○	○	○	◑	-		11		4
#2 Symfony	○	○	-	○	◑	○	○	○		-	-	○	○	○	○	○	○	◑	-		12		2
#3 Slim	○	○	-	○	○	○	○	○		-	-	○	◑	○	○	○	○	◑	-		12		2
#4 CakePHP	○	○	◑	○	◑	○	○	○		◑	◑	-	◑	○	●	●	○	◑	-		8	2	6
#5 Zend/Laminas	○	○	-	○	○	○	○	◑		-	-	○	○	○	○	○	●	◑	-		11	1	2
C#																							
#1 ASP Web Forms	○	○	○	○	○	○	○	◑		○	○	-	○	○	○	○	○	◑	◑	-	13		3
#2 ASP MVC	○	○	○	○	○	○	○	○		○	○	-	○	○	○	○	○	◑	◑	-	14		2
#3 ASP Core	○	○	○	○	○	○	○	○		○	○	-	◑	○	○	○	○	◑	◑	-	13		3
#4 Service Stack	○	○	○	○	○	○	○	○		○	○	-	○	○	○	○	○	◑	◑	-	14		2
#5 Nancy	○	○	○	○	◑	○	○	◑		◑	◑	-	○	○	○	○	○	◑	◑	-	11		5
Total ○	25	25	5	25	15	19	24	17		7	7	11	9	24	24	23	13	1	3		**274**		
Total ●														1	1	2	7					**14**	
Total ◑			10		10	6	1	8		8	8		16				5	24					**96**

Legend: ○ = no security risk; ◑ = conditional risk; ● = severe risk; - = not applicable; empty cell = zero; blue = via library; red = no defense;

Table 3: Summary of results on top five frameworks of top five languages.

Web Framework	Token Leakage						Cookie Int.		Implementation Mistakes											Total ○	Total ●	Total ◑
	BREACH	Referrer Leakage	Timing-Based Attacks on Cmp.				Cookie Tossing	Cookie Jar Overflow	Insec. Token Mapping	Safe HTTP Met. Check	Unsafe HTTP Met. Check	Logical Errors	Replay Attack	Faulty Ref./Orig. Check								
JavaScript																						
#6 Fastify	○	◑	○				◑	◑	○	◑	○	○	○	◑	-		5		5			
#7 ThinkJS	-	-	-				-	-	-	-	-	-	-	-	-		-	-	-			
#8 Total.js	-	-	-				-	-	-	-	-	-	-	-	-		-	-	-			
#9 AdonisJs	○	○	○				-	-	○	◑	○	○	○	◑	-		6		2			
Python																						
#6 Falcon	-	-	-				-	-	-	-	-	-	-	-	-							
#7 Zope	◑	○	◑				◑	◑	-	◑	○	○	○	◑	-		3		6			
#8 Masonite	◑	○	◑				◑	◑	-	◑	●	○	○	◑	-		2	1	6			
#9 TurboGears	◑	○	◑				◑	◑	-	○	○	○	○	◑	-		4		5			
#10 Web2py	○	○	○				-	-	○	○	○	○	○	◑	-		8					
Java																						
#6 Dropwizard	-	-	-				-	-	-	-	-	-	-	-	-							
#7 Blade	◑	○	○				-	-	○	◑	●	○	○	◑	-		4	1	3			
#8 ZK	○	○	○				-	-	○	○	○	○	○	◑	-		7		1			
#9 Apache Struts	◑	○	◑				-	-	○	◑	○	○	○	◑	-		5		3			
#10 Apache Wicket	-	-	-				-	-	-	-	-	○	-	○			2					
PHP																						
#6 CodeIgniter	◑	○	○				◑	◑	-	◑	●	○	○	○	-		4	1	4			
#7 FuelPHP	◑	○	○				◑	◑	-	◑	○	○	○	○	-		5		4			
#8 Yii2	○	○	○				◑	◑	-	◑	○	○	○	◑	-		5		4			
#9 Phalcon	○	○	○				-	-	○	○	○	○	○	○	-		8					
#10 Li3	○	○	○				-	-	○	○	○	○	○	◑	-		7		1			
Total ○	7	13	10						8	5	11	15	5	1	**75**							
Total ●											3					**3**						
Total ◑	7	1	4	7	7				9			9					**44**						

Legend: ○ = no security risk; ◑ = conditional risk; ● = severe risk; - = not applicable; empty cell = zero; blue = via library; red = no defense;

Table 4: Summary of results on less popular frameworks of top four languages. Only five frameworks were identified for C#.

mistake, such as using a weak key or the insecure default configuration, and the framework facilitates making that mistake, e.g., no checks on key length. Other circumstances are whether attackers will be allowed to perform sufficiently many requests to exploit side channels without being detected by network monitoring tools. Table 3 summarizes the results of our security assessment for the top five frameworks of each language, and Table 4 presents the results for the remaining, less popular frameworks (see §3 for the methodology).

The 157 security risks comprise 80 implementation mistakes affecting 37 frameworks, 37 CSRF token leakage affecting 34 frameworks, 10 security risks in token generation of 17 frameworks, and finally, 30 security risks against cookie integrity affecting 15 frameworks. Accordingly, the most common category of threats against CSRF defenses are implementation mistakes, e.g., *replay attacks* and *missing checks on safe HTTP methods* which affect 33 and 25 frameworks, respectively. The least common category of threats are those that corrupt the cookie integrity, e.g., both *cookie tossing*, and *jar overflow attacks* affect 15 frameworks.

All frameworks with a CSRF defense, except four (Falcon, Web2py, Apache Wicket, Phalcon), are exposed to at least one threat for each of the four categories. ASP.Net Core is the least exposed framework, providing a robust token-based defense addressing all threats against token generation, leakage, and cookie integrity. The framework exposed to the highest relative number of security risks—excluding not applicable ones and including insecure defaults—is CakePHP, which is exposed to 8 out of 17 threats. On average, we observe that each framework is exposed to four security risks.

Looking at the affected programming languages, the most exposed language is Python, with 45 security risks mostly being implementation mistakes. In comparison, the least exposed language is C#, which is affected by less than half security risks when compared to Python, i.e., a total of 17 security risks.

Overall, an important insight of our study is that even if frameworks offer a CSRF defense, their implementation may require developers to be aware and then address additional security threats that could compromise or even lead to a complete bypass of the CSRF defense. The rest of this section details our findings for each threat of §5.

6.2.1 Token Generation. All frameworks implement token generation mechanisms that are robust against attackers stronger than the web attacker.

Weak Generators. No framework uses *insecure cryptographic libraries* for cryptographic operations (tested via snyk [79]). All frameworks use cryptographically-secure PRNG (e.g., crypto.randomBytes [60] in JavaScript, os.urandom [71] in Python, SecureRandom.nextBytes [54] in Java, random_bytes in PHP [66], and RandomNumberGenerator.Fill [56] in C#), secure cryptographic algorithms (e.g., AES-CBC for encrypting and HMAC-256 for signing across languages), and cryptographic libraries that are not known to be vulnerable.

Insufficient Randomness. In total, 13 frameworks use encrypted or HMACed tokens. Of these, 10 frameworks require the developer to provide a cryptographic key. Among these, however, Play is the only framework that checks the key length (but not randomness) and ensures that the default key is not used in production mode. All other frameworks do not perform any check on the length or randomness of the provided key, increasing the risk of using weak keys. Also, we discovered that *all* web applications developed via the command line interface of CakePHP share the same static, default key, allowing an attacker to set valid CSRF tokens, e.g., during cookie tossing attack, and forge HTTP requests.

6.2.2 Token Leakage. Overall, the vast majority of frameworks have strict CORS configuration (24 out of 25 frameworks) and offer ad-hoc APIs to manage web forms and HTTP requests in order to minimize cross-domain referrer leakage (32 out of 39 frameworks with token-based defenses). In contrast, the most common security risk is posed by timing-based side channels on the token comparison (12 out of 39) if a web application has no means to detect or block this brute-forcing technique.

Cross-Domain Referrer Leakage. Overall, seven out of the 39 frameworks that use token-based defenses provide APIs that place the token in the URL, increasing the risk of a token leakage if the web developer decides to place the CSRF token in the URL.

CORS Misconfiguration. Among tested frameworks, the only framework with a default vulnerable CORS configuration is Play which enables an attacker to leak the CSRF token and mount a successful attack as explained in details in §6.3.

Side-channel Attacks. Most frameworks operate on top or behind an HTTP server, and data compression at the HTTP/TLS level depends on these components. In total, we observed that 15 frameworks use the same CSRF token throughout the entire user session, exposing tokens to the BREACH attack (assuming that HTTP compression is enabled and the frequent brute-forcing requests are not detected and blocked). Also, two frameworks, CodeIgniter and Vert.x-Web, are generally exposed to BREACH attacks except certain cases, i.e., when using helper functions to build forms and when sessions are not used, respectively. Only in these cases, the CSRF token is unique per request. On the contrary, the remaining frameworks provide a better protection against BREACH attacks. Most frameworks (i.e., 22) use a fresh token which is updated frequently, reducing the time window validity in which the symbols of a token can be inferred because of data compression. For example, Vaadin, Phalcon, and Web2py generate a token for each request. C#-based frameworks follow a different approach. Instead of generating fresh tokens, they use AES-CBC encrypted CSRF tokens with a fresh initialization vector (IV) per request, which results in a different token for each request, thus preventing BREACH.

Another side-channel can originate from the token comparison. Out of the 39 frameworks that use token-based defenses, 27 frameworks use constant-time comparison functions to validate CSRF tokens. The rest (most notably Java-based frameworks) use directly or via wrappers the default string comparisons, which are vulnerable to timing-based side-channel attacks [98, 118] unless web application does not detect the frequent spike in request during this brute-forcing attack.

6.2.3 Cookie Integrity. This category of threats targets CSRF defenses that use cookies, such as Double Submit cookies, which is implemented by 22 frameworks. Overall, we detected a total of 30 security risks against cookie integrity in 15 frameworks, comprised of 15 cookie tossing and 15 jar overflow attacks each. Although seven of these frameworks sign and/or encrypt the CSRF cookie for additional security, this approach does not help to prevent cookie tossing [116] or jar overflow [97] attacks if an attacker controls or hijacks a subdomain of the target domain. Attackers can simply reuse an encrypted token that they receive as normal users of the web application. As opposed to these cases, seven frameworks provide a better defense. For example, Flask stores the CSRF token in a session cookie along with all other session information. An attacker cannot easily forge a valid session cookie without knowing the entire victim's session information, especially when the cookie is encrypted. While this approach could be effective, it is not an optimal solution since session information might grow in size and the cookie size is limited [51]. A better alternative is to store a unique user identifier within the CSRF token, as applied in C#-based frameworks (except Nancy).

6.2.4 Implementation Mistakes. When looking at the mistakes introduced during the validation of incoming requests, most frameworks miss checks on safe HTTP methods (i.e., 25 out of 39 frameworks with token-based defenses). Also, the majority of frameworks (i.e., 33) do not provide adequate defenses against replay attacks.

Missing Checks and Token Verification. Web frameworks verify the CSRF tokens in two distinct ways: ad-hoc, or systematic. In the systematic verification, the token is verified automatically for state-changing requests, unless explicitly specified. In contrast, if the framework offers ad-hoc verification, the developer has to manually call the verification function for each HTTP request handler that is expected to perform a state-changing operation. This has the obvious drawback that the developer has to manually invoke the CSRF verification module in all state-changing operations. In total, out of the 39 frameworks that provide a token-based defense, 29 frameworks provide a systematic verification, nine frameworks require ad-hoc verification, and one framework, Symfony, offers both, depending on whether the developer is using a built-in Symfony form (systematic verification) or a standard HTML form (ad-hoc verification).

We observed that 25 frameworks that perform systematic verification exempt requests with *safe HTTP methods* (i.e., GET, OPTIONS, HEAD) [73] from the token verification step. This implies that if developers use safe HTTP methods for state-changing operations, they will not be CSRF-protected by the framework. When we reported these vulnerabilities, some frameworks, such as Sails,

Fastify and AdonisJS, decided to patch the issue by adding a pseudo-random synchronizer token in a custom HTTP header. However, other frameworks, such as Apache Struts or Express, were concerned if adding such protection could urge application developers to mis-use requests with safe HTTP methods for state-changing operations, which is against the RFC 7231 specification [52]. Also, this distinction of HTTP methods can lead to a bypass of the CSRF defense. For example, we discovered that the CakePHP framework allows the developer to override (via *HTTP Method Override* [59, 62]) the request method to an *arbitrary* string that is not an unsafe method (or even an HTTP method), thus not triggering the CSRF verification. We discuss the vulnerability in detail in §6.3. In addition, four of these 25 frameworks have *missing checks on unsafe HTTP Methods* since they also exclude DELETE, PUT, and PATCH requests from the token verification. These frameworks perform the CSRF verification only for POST HTTP requests.

Replay Attacks. If the CSRF token is leaked, replay attacks are possible until the expiration of the token. In total, six out of the 39 frameworks mitigate replay attacks by applying per-request tokens and invalidating them after they are consumed. However, the majority of the frameworks, i.e., 30/39, do not offer such protection. In addition, in three frameworks, the mitigation against replay attacks can be bypassed. Specifically, Vert.x-Web and Vaadin do provide per-request tokens, but they generate a new token only if the web page is refreshed. In Slim, an old CSRF token will remain in the session storage until the storage capacity is reached. Accordingly, in these cases, the same token can be abused multiple times.

Also, we noticed that many frameworks (e.g., Flask, Tornado, ASP.NET) attach a timestamp to HTTP requests. OWASP [63] suggests that timestamps can be used to prevent replay attacks. However, no framework currently does that. For example, the timestamp in Tornado is an incomplete feature [80]. Yet, frameworks such as ASP.NET MVC and ASP.NET Core allow adding extra information to the CSRF cookie (e.g., nonces or timestamps [117]), which can then be verified during the CSRF validation.

Cross-Site WebSocket Hijacking. We observed that WS connections are not treated the same in all frameworks. Overall, we found seven frameworks that allow CSRF attacks in WS connections. Then, five frameworks provide a weak defense against hijacking WS connections by a CSRF attack. For example, Spring only performs a lenient `Origin` header check, which can be bypassed if the `Origin` header is missing or null. Also, all C#-based frameworks except Nancy use an `Origin` header check (if SignalR library is not used) which allow *all origins* by default.

Finally, we observed that the rest of the frameworks offer protections against attacks hijacking WS connections, including those that not support WS (e.g., Symfony [78]). These frameworks use a strict `Origin` header check, a CSRF token verification (e.g., Laravel), or a combination of the two (e.g., Sails). Also, we noticed that one framework, Hapi, uses the `SameSite=Strict` attribute on cookies to prevent CSRF attacks on WS connections.

Logical Mistakes None of the frameworks we analyzed suffers from *insecure user-token mapping*, whereas three frameworks implement a *faulty Referer/Origin header check*, as explained in §6.1. Finally, we identified two *logical errors* in CakePHP and Vert.x-Web which result in a complete bypass of the CSRF defense. In CakePHP, for example, the route handler does not check the HTTP verb when the `connect` API [1] is used since this API is designed to process any kind of HTTP request. An attacker can exploit this by sending an HTTP request with an unknown HTTP verb, bypassing the CSRF protection. In the following subsection, we provide more details about these vulnerabilities.

6.3 Examples of Critical Vulnerabilities.

In this section, we present three critical vulnerabilities affecting Vert.x-Web, CakePHP, and Play, which are among the top five frameworks of Java and PHP, respectively, and lead to a complete bypass of the CSRF defense.

Vert.x-Web. The vulnerability [93] in Vert.x-Web framework resides in the way CSRF verification is performed. Whenever a new CSRF token is generated and signed, it is stored in a CSRF cookie which is sent to the client-side. Additionally, this generated CSRF token is stored in session storage. During the CSRF verification, Vert.x-Web retrieves the CSRF token of the user from the session storage. Then, the CSRF verification module compares the token in the CSRF cookie with the token from the session storage. However, this leads to a logical error because the CSRF token in the HTML form (i.e., in the request body) is not considered at all. Due to SOP for cookies, the CSRF cookie will always be sent by the victim's browser, thus the verification will always succeed, even if the attacker does not supply any CSRF token in the HTTP request.

CakePHP. The vulnerability [94] in CakePHP exploits the HTTP Method Override feature [59, 62] and a missing check in the routing middleware of the framework. The problem arises from the fact that the framework performs the CSRF token validation only if the request method is an unsafe HTTP method [73]. Thus, if an attacker abuses HTTP Method Override to change the request method to anything that is not an unsafe HTTP method (even an arbitrary string), the CSRF verification function is not invoked, and hence, even without a CSRF token, the request reaches the targeted endpoint, bypassing the CSRF verification.

Play. A critical vulnerability arises when the developer enables the CORS module. The default configuration of this module sets the ACAO HTTP header to reflect back the request's origin via dynamic generation [103]. Additionally, the configuration sets the ACAC header to true [40, 68]. This combination allows an attacker to send an authenticated GET request on behalf of the victim, read the CSRF token from the HTTP response across origins, and mount a CSRF attack. We also noticed a dangerous feature in Play, `bypassCorsTrustedOrigins`, which is by default set to true. This feature allows the CSRF check to be bypassed if the request's origin is trusted by the CORS module [68, 70].

7 DOCUMENTATION AND API ANALYSIS

In this section, we address RQ3 (§2.2) by presenting the results of our documentation analysis.

7.1 Quality of the Documentation

As presented in §3.5, we quantify the quality of documentation using six distinct criteria, i.e., presence of CSRF defense name or

description, presence of API specifications, description of the configuration space, presence of code examples, description of the cryptographic guarantees, and presence of a general security considerations section.

Ideally, frameworks' documentation should fulfill all six criteria, but only two frameworks achieve that. The next best documentation that meets at least five of the six criteria is observed in nine frameworks (20.5%). On the other end of the spectrum, we have six frameworks that do not mention anything about CSRF and CSRF defenses. Additionally, seven frameworks fulfill only one out of the six quality criteria, i.e., a total of 13 frameworks (29.6%) provide minimal or no CSRF documentation at all.

For the remaining 22 frameworks, the documentation contains between two to four of the quality criteria (seven, seven, and eight frameworks, respectively). We also noticed that out of all the frameworks that provide CSRF documentation, 12 frameworks do not explain the implemented defense. These frameworks provide only minimal information which is mostly focused on how to include the CSRF token in an HTML form or activate the defense. In comparison, the rest of the frameworks explain the defense and available options. Overall, our results suggest that 61.7% of the frameworks do not meet at least half of the quality criteria. Such a result is alarming, and it indicates that even if the frameworks support CSRF defenses, misconceptions and usability issues may arise as developers try to use these defenses. The complete mapping between web frameworks and documentation content is shown in Table 7 in Appendix A.

7.2 API Abstraction Analysis

Although our evaluation identified different CSRF defenses, the vast majority of frameworks (i.e., 84%) implement a defense from *Request Unguessability* category. The APIs analysis reveals a variety in the semantics and operations, diverging in the integration, configuration, generation, and validation of CSRF tokens. Overall, there is no established consensus in the way unguessable request defenses are exposed to developers.

Defense Configuration. In total, 11 frameworks have the CSRF defense enabled by default. The developers need to install an external library or enable the defense in a configuration file for the rest of the frameworks. We observed that 20 frameworks allow the developers to configure the CSRF defense. However, the configuration mainly relates to the type of defense to implement (Plain Token or Double Submit) and the cryptographic key. Other features such as the token length, token generation method, signing/encryption of the tokens, and HTTP methods to validate are neither explained nor configurable. As such, developers may be forced to customize the CSRF defense code via "monkey patching" to increase the security guarantees of the defense.

Token Generation. Our review of the APIs show that web frameworks provide three distinct ways to add tokens to requests. The first option is by calling the token generation function and arbitrarily placing the token in the request. The second option uses helpers or pseudo-variables that are interpreted by a template engine when generating the HTML code for the browser. The third option is using framework-provided special form objects.

Looking at frameworks, first, a total of three frameworks allow calling the token generation function directly. Then, 34 frameworks rely on template engines to render CSRF tokens in a form. Among these frameworks, the semantics and operations to use tokens in templates are not uniform. For example, some frameworks require using special keywords to generate the input field that stores the token (e.g., `@csrf` in Laravel). Other frameworks require the developer to create the HTML form and input fields manually and only replace the `value` attribute with a specific pseudo-variable. Unfortunately, these pseudo-variables are not always directly interpretable by the template engine. In Vert.x-Web, for example, the developer needs to pass the pseudo-variable to the template engine for every HTTP response so that the template engine can recognize it. Other frameworks, e.g., Django or Flask, offer template engines that handle this process automatically.

Finally, a total of nine frameworks offer special form objects to add the CSRF token, out of which six add the CSRF token automatically in the HTML, while for the rest, the developer needs to add the token manually.

Token Validation. Web frameworks provide three different techniques to validate the CSRF token: (1) calling the CSRF verification function; (2) using method decorators; (3) automatic. Regardless of the technique, the state-changing request is verified before reaching the targeted endpoint.

8 DEVELOPERS FEEDBACK

We notified the affected frameworks about the discovered security risks (by sharing a proof-of-concept exploit), and our findings of the status of the documentation. The summary of our notification campaign is in Table 9.

Overview. Out of the total 157 notifications, 55 security risks were confirmed by the frameworks' developers who replied to our report. Out of 55, 27 security risks are already patched in eight frameworks (see, e.g, [93], or [94]), and the rest of the confirmed security risks (i.e., 28) are still in the process of being patched. Then, 24 out of 157 vulnerability reports are still in the process of being reviewed by the eight affected frameworks (e.g., Pyramid, Hapi, or Laravel). For 17 vulnerability reports that affect four frameworks (i.e., CodeIgniter, FuelPHP, Blade, and ServiceStack) and two external libraries (i.e., `koa-socket` for Koa and `swool` for Zend), the developers have not replied to our notifications yet. Similarly, for 18 security risks that affect four frameworks (i.e., Bottle, Zope, Nancy, and ASP.NET Web Forms), the developers said the code is no longer maintained or replaced by a newer option. Finally, for 43 security risks of 19 different frameworks, the frameworks' developers decided not to take any further action because they either did not confirm the vulnerability, or determined that the potential impact of the issue is low.

Inconsistent Threat Model. When looking at the developers' feedback over the reported 14 security risks, we observe inconsistent responses about the threat's validity. For 12 out of the 14 reported, we have at least one developers team who did not consider the reported threat a valid concern. Instead, they indicated web frameworks' users as the ones who should address them. On the contrary, for 10 out of the 14 reported vulnerabilities, at least one team acknowledged the threat and patched their code. Interestingly,

for six out of the 14 reported vulnerabilities, we have one team that addressed the threat and another team that did not.

9 DISCUSSION

The overarching goal of the study of this paper is to identify possible new explanations to a two-decades-old web vulnerability, by looking at the code and documentation of existing CSRF defenses as implemented by web frameworks. In this section, we distill our main findings, showing that while implementations of CSRF defenses exist, much of their correct and secure implementation depends on developers' awareness about CSRF attacks, threats to defenses, and specific behaviors of the implementations.

Who is Responsible. Both our security analysis and developers' feedback indicate that a large fraction of threats is adequately addressed by web frameworks, showing a rather consistent threat model across the various web frameworks. However, at the same time, our results show that for a significant fraction of threats, frameworks' developers have divergent expectations about who is responsible for addressing them. For example, when we reported the insufficient key randomness security risk to the affected frameworks, CakePHP and Vert.x-Web patched the issue, but other frameworks, such as Flask and Express, questioned their responsibility for checking the developer-provided secret keys' randomness. Another example is the CSRF token verification in Hapi and Masonite, which is disabled by default for unsafe HTTP methods. On the one hand, Masonite decided to patch the issue. On the other hand, Hapi argued that they expect the developer to change the default configuration. While both frameworks acknowledge that insecure default configurations lead to a vulnerability, they do not agree on the responsible party to address it.

Correct Use Require Awareness. All frameworks, except for Meteor, requires developers to at least write additional code to protect against CSRF attacks. Meteor does not require that, and it offers a by-default, cookie-less CSRF defense. Even worse, for more than 60% of the CSRF defenses, developers need to enable them explicitly. Finally, not all frameworks offer built-in defenses, and for more than a third of the cases, developers must look for an external library by themselves. In most of these cases, frameworks' documentation provides pointers to libraries implementing a CSRF defense, whereas the remaining ones do not offer such help.

Secure Defenses Require Diverse Expertise. Even when developers can correctly enable and use CSRF defenses, they need to be well informed about the plethora of threats that might weaken or bypass the defense building blocks. Our evaluation identified 14 treats and, while token generation is, in general, the most robust building block, the remaining building blocks are equally exposed to a variety of risks. We mention replay attacks, cookie tossing and jar overflow among the most concerning ones.

Incomplete and Inconsistent Documentation. Our qualitative evaluation of the documentation shows that most of the frameworks and libraries have incomplete and inconsistent documentation. Issues range from trivial details such as the name of the implemented defense (e.g., double submit token) or advanced ones such as no code examples showing correct use. Even more concerning is the lack of details about the threats considered in this study that can affect the implemented defense's building blocks. In general, we find that the documentation accompanying web frameworks and libraries does not adequately help developers in raising awareness nor build-up cognition about the many threats that can weaken or bypass CSRF defenses.

Same-Site Helpful but May not Completely Cut it. SameSite cookies are relatively new mechanisms that could help protecting from CSRF attacks. Our evaluation shows that only ten frameworks and libraries offer SameSite protection. However, we point out that SameSite cookies *alone* may not be sufficient to protect from CSRF attacks. For example, they are not sufficient to protect against a newer variant of CSRF attacks that leverage on the client-side JavaScript code [89]. Similarly, when applications mis-use GET-based HTTP requests for sensitive state-changing operations, the new default SameSite cookie policy (i.e., Lax) cannot mitigate CSRF attacks [39]. Additionally, as SameSite can change the behavior of existing web services, developers may relax it on purpose to avoid adapting their code. Finally, SameSite cookies are not fully supported by all programming languages [91]. For example, SameSite cookies are not supported in PHP until PHP 7.3 [65, 67].

Language-specific Results. A closer look at our results show appreciable language-based differences across CSRF implementations. For example, C#-based frameworks come with a built-in CSRF defense, whereas, for the other languages, a fraction between 20% to 70% of the frameworks do not offer a built-in defense. We also point out that web frameworks and libraries within a language tend to implement the same defense. For example, JavaScript and Java frameworks mostly implement Plain Token, while Python and C# frameworks implement Double Submit. For JavaScript, the reason is that most of the frameworks rely only on two external libraries (i.e., *csurf* and *csrf* libraries). We also noticed that, in contrast to the vast majority of the web frameworks, most Java frameworks are subject to timing-based side-channel attacks since they use the Java's built-in `equals` function for token comparison. The default behavior of this function does not perform a constant-time comparison (unless overridden). These examples show that the implementation of CSRF defenses may differ among frameworks of different languages.

10 CONCLUSION

In this work, we identified and analyzed all existing CSRF defenses for potential security threats. We performed the first systematic study of the implementation and security guarantees of CSRF defenses in 44 top web frameworks of five popular programming languages. Our results are alarming. We identified 157 security risks affecting 37 frameworks that can be exploited to perform a CSRF attack. We discovered three critical vulnerabilities in CakePHP, Vert.x-Web, and Play that allow the attacker to bypass the CSRF defense. Also, a closer look at the developers' feedback reveals that, for a considerable fraction of security risks, frameworks have divergent expectations about who is responsible for addressing them. Also, we systematically reviewed the CSRF-related documentation and API surface of CSRF defenses which showed that at least 61.4% of the frameworks do not provide sufficient documentation regarding the CSRF defense. Overall, our research shows that although web frameworks provide developers with the tools to defend against CSRF, their correct and secure implementation too often depends on developers' awareness and diverse expertise about CSRF attacks,

who need to overcome missing and misleading documentation as well as insecure defaults.

REFERENCES

[1] [n.d.]. *CakePHP Documentation: Routing.* https://book.cakephp.org/3/en/development/routing.html#routes-configuration.

[2] [n.d.]. *Common CSRF prevention misconceptions.* https://www.nccgroup.com/uk/about-us/newsroom-and-events/blogs/2017/september/common-csrf-prevention-misconceptions/.

[3] [n.d.]. *Common Weakness Enumeration: A Community-Developed List of Software & Hardware Weakness Types.* https://cwe.mitre.org/data/definitions/352.html.

[4] [n.d.]. *Cryptographic Storage Cheat Sheet.* https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html.

[5] [n.d.]. *CVE-2010-5084.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-5084.

[6] [n.d.]. *CVE-2012-1598.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-1598.

[7] [n.d.]. *CVE-2013-2213.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2213.

[8] [n.d.]. *CVE-2014-1808.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1808.

[9] [n.d.]. *CVE-2014-9720.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-9720.

[10] [n.d.]. *CVE-2015-2206.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2206.

[11] [n.d.]. *CVE-2015-4056.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-4056.

[12] [n.d.]. *CVE-2015-6728.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6728.

[13] [n.d.]. *CVE-2015-8125.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8125.

[14] [n.d.]. *CVE-2015-8623.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8623.

[15] [n.d.]. *CVE-2015-9243.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-9243.

[16] [n.d.]. *CVE-2016-10535.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10535.

[17] [n.d.]. *CVE-2016-10549.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10549.

[18] [n.d.]. *CVE-2016-5739.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5739.

[19] [n.d.]. *CVE-2016-6582.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-6582.

[20] [n.d.]. *CVE-2016-6806.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-6806.

[21] [n.d.]. *CVE-2016-8615.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8615.

[22] [n.d.]. *CVE-2017-0894.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-0894.

[23] [n.d.]. *CVE-2017-16136.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-16136.

[24] [n.d.]. *CVE-2017-9339.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9339.

[25] [n.d.]. *CVE-2018-1000119.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1000119.

[26] [n.d.]. *CVE-2018-10899.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-10899.

[27] [n.d.]. *CVE-2018-6651.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6651.

[28] [n.d.]. *CVE-2019-12659.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-12659.

[29] [n.d.]. *CVE-2019-13209.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-13209.

[30] [n.d.]. *CVE-2019-14998.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-14998.

[31] [n.d.]. *CVE-2019-15515.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-15515.

[32] [n.d.]. *CVE-2019-17654.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-17654.

[33] [n.d.]. *CVE-2020-11825.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-11825.

[34] [n.d.]. *CVE-2020-14368.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-14368.

[35] [n.d.]. *CVE-2020-25095.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-25095.

[36] [n.d.]. *CVE-2020-5261.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-5261.

[37] [n.d.]. *CVE-2021-23127.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-23127.

[38] [n.d.]. *CVE-2021-26296.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-26296.

[39] [n.d.]. *Defending against CSRF with SameSite cookies.* https://portswigger.net/web-security/csrf/samesite-cookies.

[40] [n.d.]. *Documentation: Cross-Origin Resource Sharing.* https://www.playframework.com/documentation/2.8.x/CorsFilter.

[41] [n.d.]. *Fetch API.* https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.

[42] [n.d.]. *GitHub's Annual Report: The State of the Octoverse.* https://octoverse.github.com/.

[43] [n.d.]. *Hackerone.* https://hackerone.com.

[44] [n.d.]. *HackerOne, Report 342693: CSRF and password reset token leakage via referer.* https://hackerone.com/reports/342693.

[45] [n.d.]. *HackerOne, Report 426147: CORS misconfiguration lead to CSRF and account takeover.* https://hackerone.com/reports/426147.

[46] [n.d.]. *HackerOne, Report 576504: Authentication bypass by abusing insecure crypto tokens in Revive Adserver.* https://hackerone.com/reports/576504.

[47] [n.d.]. *HackerOne, Report 577969: CORS misconfiguration allows to steal customers data and CSRF tokens.* https://hackerone.com/reports/577969.

[48] [n.d.]. *HackerOne, Report 787160: Referer leakage vulnerability in rockstargames leads to Facebook's OAuth token theft.* https://hackerone.com/reports/787160.

[49] [n.d.]. *HackerOne, Report 975983: Site-wide CSRF on Safari due to CORS misconfiguration.* https://hackerone.com/reports/975983.

[50] [n.d.]. *How do I get the parameters of a post request when using a pac4j security filter in Spark Java?* https://stackoverflow.com/questions/43240829/how-do-i-get-the-parameters-of-a-post-request-when-using-a-pac4j-security-filter.

[51] [n.d.]. *HTTP State Management.* https://tools.ietf.org/html/rfc6265.

[52] [n.d.]. *Hypertext Transfer Protocol (HTTP/1.1).* https://tools.ietf.org/html/rfc7231.

[53] [n.d.]. *Insecure token generation in Kayako.* https://www.sjoerdlangkemper.nl/2016/06/23/insecure-tokens-in-kayako/.

[54] [n.d.]. *Java Documenation: Class SecureRandom.* https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html.

[55] [n.d.]. *Meteor.js and CSRF/XSS Attacks.* https://stackoverflow.com/questions/21807229/meteor-js-and-csrf-xss-attacks.

[56] [n.d.]. *Microsoft Documentation: RandomNumberGenerator.Fill(Span<Byte>) Method.* https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.randomnumbergenerator.fill?view=net-5.0.

[57] [n.d.]. *MITRE CVE database.* https://cve.mitre.org/.

[58] [n.d.]. *National Vulnerability Database: CSRF statistics.* https://nvd.nist.gov/vuln/search/statistics?form_type=Advanced&results_type=statistics&query=CSRF&search_type=all.

[59] [n.d.]. *NODE.JS CONNECT CSRF BYPASS ABUSING METHODOVERRIDE MIDDLEWARE.* http://blog.nibblesec.org/2014/05/nodejs-connect-csrf-bypass-abusing.html.

[60] [n.d.]. *Node.js Documentation: crypto.randomBytes.* https://nodejs.org/api/crypto.html#crypto_crypto_randombytes_size_callback.

[61] [n.d.]. *NPM package manger.* https://www.npmjs.com/.

[62] [n.d.]. *Often Misused: HTTP Method Override.* https://vulncat.fortify.com/en/detail?id=desc.dynamic.xtended_preview.often_misused_http_method_override.

[63] [n.d.]. *OWASP: Cross-Site Request Forgery Prevention Cheat Sheet.* https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html.

[64] [n.d.]. *OWASP: Insufficient Session-ID Length.* https://owasp.org/www-community/vulnerabilities/Insufficient_Session-ID_Length.

[65] [n.d.]. *PHP 7.3: SameSite cookie support.* https://php.watch/versions/7.3/same-site-cookies.

[66] [n.d.]. *PHP Documentation: random_bytes.* https://www.php.net/manual/en/function.random-bytes.php.

[67] [n.d.]. *PHP setcookie SameSite=Strict.* https://php.watch/versions/7.3/same-site-cookies.

[68] [n.d.]. *Play filter configurations.* https://www.playframework.com/documentation/2.8.x/resources/confs/filters-helpers/reference.conf.

[69] [n.d.]. *Predictable token in Froxlor that uses timestamps and the rand() method.* https://github.com/Froxlor/Froxlor/commit/da4ec3e1b591de96675817a009e26e05e848a6ba.

[70] [n.d.]. *Protecting against Cross Site Request Forgery.* https://www.playframework.com/documentation/2.8.x/JavaCsrf.

[71] [n.d.]. *Python: os — Miscellaneous operating system interfaces.* https://docs.python.org/3/library/os.html#os.urandom.

[72] [n.d.]. *Question: Was it intentional to validate crumb key for POST only?* https://github.com/hapijs/crumb/issues/4.

[73] [n.d.]. *Safe HTTP Methods.* https://developer.mozilla.org/en-US/docs/Glossary/safe.

[74] [n.d.]. *SameSite Cookies & CSRF Attacks.* https://symfonycasts.com/screencast/api-platform-security/samesite-csrf.

[75] [n.d.]. *Spark Framework CSRF Protection.* https://stackoverflow.com/questions/43317938/spark-framework-csrf-protection.

[76] [n.d.]. *Spark Framework CSRF Protection.* https://bottle-utils.readthedocs.io/en/latest/csrf.html.

[77] [n.d.]. *Stackoverflow Tags.* https://stackoverflow.com/help/tagging.

[78] [n.d.]. *Symfony: [RFC] Add support for Websockets and real-time applications.* https://github.com/symfony/symfony/issues/17051.

[79] [n.d.]. *Test your code.* https://snyk.io/test/.

[80] [n.d.]. *Tornado Github Issue 2722: Misleading CSRF Docs / Bug in Setting CSRF Cookie.* https://github.com/tornadoweb/tornado/issues/2722.

[81] [n.d.]. *The Web Origin Concept.* https://www.ietf.org/rfc/rfc6454.txt.

[82] [n.d.]. *Why Meteor doesn't use session cookies.* https://blog.meteor.com/why-meteor-doesnt-use-session-cookies-e988544f52c9.

[83] [n.d.]. *Window.localStorage APIs.* https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage.

[84] 1999. *Hypertext Transfer Protocol – HTTP/1.1.* https://tools.ietf.org/html/rfc2616#page-53.

[85] 2009. *Netflix CSRF Revisited.* https://appsecnotes.blogspot.com/2009/01/netflix-csrf-revisited.html.

[86] 2010. *Patching auto-complete vulnerabilities not enough, Cookie Eviction to the rescue.* https://blog.jeremiahgrossman.com/2010/07/patching-auto-complete-vulnerabilities.html.

[87] 2013. *Twitter CSRF account control exploit.* https://www.itproportal.com/2013/11/07/twitter-rapidly-fixes-csrf-account-control-exploit/.

[88] 2016. *OWASP Top Ten.* https://owasp.org/www-project-top-ten/.

[89] 2018. *Client-Side CSRF.* https://www.facebook.com/notes/facebook-bug-bounty/client-side-csrf/2056804174333798/.

[90] 2019. *Critical CSRF Vulnerability on Facebook.* https://www.acunetix.com/blog/web-security-zone/critical-csrf-vulnerability-facebook/.

[91] 2019. *Developers: Get Ready for New SameSite=None; Secure Cookie Settings.* https://blog.chromium.org/2019/10/developers-get-ready-for-new.html.

[92] 2019. *Intent to implement: Cookie SameSite=lax by default and SameSite=none only if secure.* https://groups.google.com/forum/#!msg/mozilla.dev.platform/nx2uP0CzA9k/BNVPWDHsAQAJ.

[93] 2020. *CVE-2020-35217.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-35217.

[94] 2020. *CVE-2020-35239.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-35239.

[95] 2020. *SameSite cookie attribute, Chromium, Blink.* https://www.chromestatus.com/feature/4672634709082112.

[96] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky. 2017. Comparing the Usability of Cryptographic APIs. In *IEEE Symposium on Security and Privacy (SP)*.

[97] Wade Alcorn, Christian Frichot, and Michele Orru. 2014. *The Browser Hacker's Handbook.* John Wiley & Sons. 268–270 pages.

[98] Scott Arciszewski. [n.d.]. *Preventing Timing Attacks on String Comparison with a Double HMAC Strategy.* https://paragonie.com/blog/2015/11/preventing-timing-attacks-on-string-comparison-with-double-hmac-strategy.

[99] Adam Barth, Collin Jackson, and John C. Mitchell. 2008. Robust Defenses for Cross-Site Request Forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '08)*. Association for Computing Machinery, New York, NY, USA, 75–88. https://doi.org/10.1145/1455770.1455782

[100] Jeremiah Blatz. [n.d.]. CSRF: Attack and Defense. ([n. d.]).

[101] Stefano Calzavara, Mauro Conti, Riccardo Focardi, Alvise Rabitti, and Gabriele Tolomei. 2019. Mitch: A Machine Learning Approach to the Black-Box Detection of CSRF Vulnerabilities. In *Proceedings of the IEEE European Symposium on Security and Privacy*.

[102] Jianjun Chen, Jian Jiang, Haixin Duan, Tao Wan, Shuo Chen, Vern Paxson, and Min Yang. 2018. We Still Don't Have Secure Cross-Domain Requests: An Empirical Study of CORS. In *Proceedings of the 27th USENIX Conference on Security Symposium*.

[103] Jianjun Chen, Jian Jiang, Haixin Duan, Tao Wan, Shuo Chen, Vern Paxson, and Min Yang. 2018. We Still Don't Have Secure Cross-Domain Requests: an Empirical Study of CORS. In *27th USENIX Security Symposium*.

[104] Alexei Czeskis, Alexander Moshchuk, Tadayoshi Kohno, and Helen J. Wang. 2013. Lightweight server support for browser-based CSRF protection. In *Proceedings of the International Conference on World Wide Web*.

[105] Dorothy E Denning and Giovanni Maria Sacco. 1981. Timestamps in key distribution protocols. *Commun. ACM* 24, 8 (1981), 533–536.

[106] Yoel Gluck, Neal Harris, and Angelo Prado. 2013. BREACH: reviving the CRIME attack. *Unpublished manuscript* (2013).

[107] Peter Leo Gorski, Yasemin Acar, Luigi Lo Iacono, and Sascha Fahl. 2020. Listen to Developers! A Participatory Design Study on Security Warnings for Cryptographic APIs. In *Proceedings of the 2020 Conference on Human Factors in Computing Systems (CHI)*.

[108] Michael Howard and David LeBlanc. 2003. *Writing secure code.* Pearson Education. 350–361 pages.

[109] Lin-Shung Huang, Alex Moshchuk, Helen J. Wang, Stuart Schechter, and Collin Jackson. 2012. Clickjacking: Attacks and Defenses. In *Proceedings of the 21st USENIX Conference on Security Symposium*.

[110] David Johansson. 2017. A Double Defeat of the Double-Submit Cookie Pattern. (2017).

[111] Martin Johns. 2007. The three faces of CSRF. Talk at the DeepSec2007 conference. (2007). https://deepsec.net/archive/2007.deepsec.net/speakers/index.html#martin-johns.

[112] Martin Johns and Justus Winter. 2006. RequestRodeo: Client side protection against session riding. https://www.owasp.org/images/4/42/RequestRodeo-MartinJohns.pdf.

[113] Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. 2006. Preventing Cross Site Request Forgery Attacks. In *Second International Conference on Security and Privacy in Communication Networks and the Workshops (SecureComm)*.

[114] Florian Kerschbaum. 2007. Simple cross-site attack prevention. In *2007 Third International Conference on Security and Privacy in Communications Networks and the Workshops-SecureComm 2007*. IEEE, 464–472.

[115] Sebastian Lekies, Walter Tighzert, and Martin Johns. 2012. Towards stateless, client-side driven Cross-site request forgery protection for Web applications. *SAP Research* (2012).

[116] Rich Lundeen, Jesse Ou, and Travis Rhodes. 2011. New ways im going to hack your web app. *Blackhat AD* (2011), 1–11.

[117] Sreekanth Malladi, Jim Alves-Foss, and Robert B Heckendorn. 2002. *On preventing replay attacks on security protocols.* Technical Report. IDAHO UNIV MOSCOW DEPT OF COMPUTER SCIENCE.

[118] Jian Mao, Yue Chen, Futian Shi, Yaoqi Jia, and Zhenkai Liang. 2016. Toward Exposing Timing-Based Probing Attacks in Web Applications. In *Proceedings of the International Conference on Wireless Algorithms, Systems, and Applications*.

[119] Ziqing Mao, Ninghui Li, and Ian Molloy. 2009. Defeating Cross-Site Request Forgery Attacks with Browser-Enforced Authenticity Protection. In *13th International Conference on Financial Cryptography and Data Security*.

[120] Kai Mindermann and Stefan Wagner. 2018. Usability and Security Effects of Code Examples on Crypto APIs. In *16th Annual Conference on Privacy, Security and Trust (PST)*.

[121] Nikhil Patnaik, Joseph Hallett, and Awais Rashid. 2019. Usability Smells: An Analysis of Developers' Struggle With Crypto Libraries. In *Fifteenth Symposium on Usable Privacy and Security(SOUPS)*.

[122] Giancarlo Pellegrino, Martin Johns, Simon Koch, Michael Backes, and Christian Rossow. 2017. Deemon: Detecting CSRF with dynamic analysis and property graphs. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*.

[123] Philippe De Ryck, Lieven Desmet, Thomas Heyman, Frank Piessens, and Wouter Joosen. 2010. CsFire: Transparent Client-Side Mitigation of Malicious Cross-Domain Requests. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*.

[124] Philippe De Ryck, Lieven Desmet, Wouter Joosen, and Frank Piessens. 2011. Automatic and Precise Client-Side Protection against CSRF Attacks. In *European Symposium on Research in Computer Security (ESORICS)*.

[125] Christian Schneider. [n.d.]. *Cross-Site WebSocket Hijacking.* https://christian-schneider.net/CrossSiteWebSocketHijacking.html.

[126] Thomas Schreiber. 2004. Session Riding-A Widespread Vulnerability in Today's Web Applications.(2004).

[127] Hossain Shahriar and Mohammad Zulkernine. 2010. Client-Side Detection of Cross-Site Request Forgery Attacks. In *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering*.

[128] Robin Sharma. 2017. Preventing cross-site attacks using same-site cookies. https://blogs.dropbox.com/tech/2017/03/preventing-cross-site-attacks-using-same-site-cookies/.

[129] Avinash Sudhodanan, Roberto Carbone, Luca Compagna, and Nicolas Dolgin. 2017. Large-scale analysis & detection of authentication cross-site request forgeries. In *2017 IEEE European Symposium on Security and Privacy*.

[130] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. 2015. The Clock is Still Ticking: Timing Attacks in the Modern Web. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*.

[131] Mike West. 2019. Incrementally Better Cookies. (2019). https://tools.ietf.org/html/draft-west-cookie-incrementalism-00.

[132] Chamila Wijayarathna and Nalin A. G. Arachchilage. 2018. A methodology to Evaluate the Usability of Security APIs. In *IEEE International Conference on Information and Automation for Sustainability (ICIAfS)*.

[133] John Wilander. 2012. Advanced CSRF and Stateless Anti-CSRF. (2012).

[134] William Zeller and Edward W. Felten. 2008. Cross-Site Request Forgeries: Exploitation and Prevention. In *Princeton University*.

A ADDITIONAL EVALUATION DETAILS

Nr.	Web Framework	GitHub Used By	Monthy Downloads	Stack Overflow Questions	GitHub Watch	GitHub Star	GitHub Fork
	JavaScript						
1	Express	6.4m	42.8m	65.9k	1.8k	49.1k	8.1k
2	Meteor	-	-	28.6k	1.7k	41.7k	5.1k
3	Koa	124k	1.7m	1.1k	882	29.5k	2.8k
4	Hapi	-	828k	523	439	12.5k	1.3k
5	Sails	24.7k	112k	6.5k	709	21.4k	1.9k
6	Fastify	7.2k	360k	142	265	14.7k	1k
7	ThinkJS	1.9k	2.2k	-	277	5.1k	632
8	Total.js	512	2.3k	69	235	4.1k	440
9	AdonisJs	112	620	440	243	8.1k	451
	Python						
1	Flask	462k	11.6m	38.3k	2.3k	51k	13.6k
2	Django	398k	4.8m	236k	2.3k	50.4k	21.8k
3	Tornado	98.2k	10.8m	3.5k	1.1k	19.2k	5.4k
4	Bottle	15.1k	1.3m	1.4k	322	6.9k	1.3k
5	Pyramid	9.7k	172k	2.2k	174	3.7k	865
6	Falcon	6.7k	488k	183	296	7.9k	791
7	Zope	-	18k	717	85	224	84
8	Masonite	-	2.9k	18	61	1.4k	88
9	TurboGears	587	636	150	32	749	67
10	Web2py	54	340	2.1k	231	1.9k	836
	PHP						
1	Laravel	488k	243k	150k	4.7k	59.9k	18.8k
2	Symfony	59.3k	24.9k	66.2k	1.3k	26k	7.6k
3	Slim	25.1k	10.2k	2.6k	564	10.7k	1.9k
4	CakePHP	10k	4.2k	30.9k	617	8.2k	3.4k
5	Zend/ Laminas	6.3k	4.5k	20.4k	542	5.7k	2.8k
6	CodeIgniter	858	551	66.8k	1.7k	18k	7.9k
7	FuelPHP	510	297	525	110	1.4k	294
8	Yii2	30	7k	14.2k	1.2k	13.3k	6.9k
9	Phalcon	9	58	1.9k	701	10.2k	1.9k
10	Li3	-	55	300	93	1.1k	243
	Java						
1	Spring	162k	-	171k	3.5k	38k	25.7k
2	Play	16.8k	-	-	712	11.6k	3.9k
3	Spark	19k	-	534	436	8.8k	1.5k
4	Vert.x-Web	16.2k	-	1.9k	87	767	361
5	Vaadin	10.3k	-	5k	151	1.6k	730
6	Dropwizard	8.7k	-	1.8k	439	7.7k	3.2k
7	Blade	1.9k	-	2.5k	315	5.4k	1.1k
8	ZK	1.4k	-	1.1k	49	304	166
9	Apache Struts	482	-	3.6k	129	1k	671
10	Apache Wicket	459	-	3.5k	60	513	339
	C#						
1	ASP.NET Web Forms	322k	-	357k	77	606	290
2	ASP.NET MVC	322k	-	357k	77	606	290
3	ASP.NET Core	10k	-	48k	1.5k	18.1k	5.1k
4	Service Stack	2.1k	-	5k	542	4.8k	1.6k
5	Nancy	-	-	1.1k	452	7.1k	1.5k

Table 5: Selected Web Frameworks.

Language	Web Framework	Library
JavaScript	Express v4.17.1	csurf v1.11
	Koa v2.12	koa-csrf v3.0.8
	Hapi v19.1.1	crumb v8.0
	Fastify v2.14.1	fastify-csrf v1.0.3
	AdonisJs v0.9	shield v3.0
Python	Flask v1.1.2	flask-wtf v0.14.3
	Bottle v0.12	bottle-utils v2.0
	Zope v4.2	zope.formlib v4.7.1
	TurboGears v2.4.1	tgext.utils v0.0.3
Java	Spring v5.2.6	Spring Security v5.2.1
	Spark v3.0	pac4j v4.0.2
PHP	Symfony v5.0.8	security-csrf v5.1
	Slim v4.0	Slim-Csrf v1.0

Legend: red = unofficial external library;

Table 6: Official and unofficial external libraries implementing CSRF defenses.

	Defense Name and Description	Cryptographic Guarantees	Code Example	API Specifications	Defense Configuration	General Security Considerations
JavaScript						
#1 Express v4.17.1	●	-	●	●	●	-
#2 Meteor v1.10.2	-	●	●	●	-	●
#3 Koa v2.12	-	-	●	●	-	-
#4 Hapi v19.1.1	●	●	●	●	●	●
#5 Sails v1.2.4	●	-	●	●	-	●
#6 Fastify v2.14.1	●	-	●	●	●	●
#7 ThinkJS v3.2.11	-	-	-	-	-	-
#8 Total.js v2.9	-	-	-	-	-	-
#9 AdonisJs v0.9	-	-	●	-	●	-
Python						
#1 Flask v1.1.2	-	-	●	●	●	-
#2 Django v3.0.6	●	-	●	●	●	●
#3 Tornado v6.0.4	●	-	●	●	●	-
#4 Bottle v0.12	-	-	-	-	-	-
#5 Pyramid v1.10.4	●	-	●	●	●	●
#6 Falcon v2.0	●	-	●	●	●	-
#7 Zope v4.2	-	-	●	-	-	-
#8 Masonite v2.3.8	●	-	●	-	-	-
#9 TurboGears v2.4.1	-	-	●	-	●	-
#10 Web2py v2.20.4	-	-	-	●	-	-
Java						
#1 Spring v5.2.6	●	-	●	●	●	●
#2 Play v2.8.1	●	-	●	●	●	●
#3 Spark v3.0	-	-	-	-	-	-
#4 Vert.x-Web v4.0 milestone1-4	●	-	●	●	●	-
#5 Vaadin v8.11	-	-	-	●	-	-
#6 Dropwizard v2.0	-	-	-	-	-	-
#7 Blade v2.0.15	-	-	-	●	-	-
#8 ZK v9.0	●	-	-	●	●	-
#9 Apache Struts v2.3	-	-	-	●	-	-
#10 Apache Wicket v8.8	●	●	●	●	●	●
PHP						
#1 Laravel v7.0	-	-	●	●	-	-
#2 Symfony v5.0.8	●	-	●	●	●	-
#3 Slim v4.0	-	-	●	-	-	-
#4 CakePHP v4.0	●	-	●	●	●	-
#5 Zend/Laminas v3.1.1	●	-	●	●	-	-
#6 CodeIgniter v3.1.11	●	-	●	-	-	-
#7 FuelPHP v1.9	-	-	●	●	-	-
#8 Yii2 v2.0.35	-	-	●	●	-	●
#9 Phalcon v4.0	●	-	●	●	-	-
#10 Li3 v1.2	●	●	-	●	-	-
C#						
#1 ASP.NET Web Forms v4.8	●	-	●	●	-	-
#2 ASP.NET MVC v4.8	●	-	●	●	●	●
#3 ASP.NET Core v3.1	●	-	●	●	●	●
#4 Service Stack v5.9.0	-	-	●	-	-	-
#5 Nancy v2.0	-	-	-	-	-	-

Legend: ● = contained in the documentation;
- = not contained;
red = no official CSRF defense;

Table 7: Mapping between web frameworks and quality of the documentation.

	Plain Token	Encrypted Token	HMAC Token	Double Submit	Triple Submit	Cookie-less User Sessions	Referer/Origin Header Check	Custom Request Header	Re-authentication	One-Time Token	(re)CAPTCHA	Frequent log outs	SameSite cookies
JavaScript													
#1 Express v4.17.1	●	-	-	●	-	-	-	-	-	-	-	-	-
#2 Meteor v1.10.2	-	-	-	-	-	●	-	-	-	-	-	-	-
#3 Koa v2.12	●	-	-	-	-	-	-	-	-	-	-	-	-
#4 Hapi v19.1.1	-	-	-	●	-	-	-	-	-	-	-	-	-
#5 Sails v1.2.4	●	-	-	-	-	-	-	-	-	-	-	-	-
#6 Fastify v2.14.1	●	-	-	●	-	-	-	-	-	-	-	-	-
#7 ThinkJS v3.2.11	-	-	-	-	-	-	-	-	-	-	-	-	-
#8 Total.js v2.9	-	-	-	-	-	-	-	-	-	-	-	-	-
#9 AdonisJs v0.9	●	-	-	-	-	-	-	-	-	-	-	-	-
Python													
#1 Flask v1.1.2	-	-	●	●	-	-	●	-	-	-	-	-	-
#2 Django v3.0.6	-	-	-	●	-	-	●	-	-	-	-	-	●
#3 Tornado v6.0.4	-	-	●	●	-	-	-	-	-	-	-	-	-
#4 Bottle v0.12	-	-	●	●	-	-	-	-	-	-	-	-	-
#5 Pyramid v1.10.4	-	-	●	●	-	-	●	-	-	-	-	-	●
#6 Falcon v2.0	-	-	-	-	-	-	-	-	-	-	-	-	●
#7 Zope v4.2	-	-	-	●	-	-	-	-	-	-	-	-	-
#8 Masonite v2.3.8	-	-	-	●	-	-	-	-	-	-	-	-	-
#9 TurboGears v2.4.1	-	-	-	●	-	-	-	-	-	-	-	-	-
#10 Web2py v2.20.4	●	-	-	-	-	-	-	-	-	-	-	-	-
Java													
#1 Spring v5.2.6	●	-	-	-	-	-	-	-	-	-	-	-	-
#2 Play v2.8.1	-	-	●	●	-	-	-	-	-	-	-	-	●
#3 Spark v3.0	●	-	-	-	-	-	-	-	-	-	-	-	-
#4 Vert.x-Web v4.0 milestone1-4	-	-	●	●	-	-	-	-	-	-	-	-	-
#5 Vaadin v8.11	●	-	-	-	-	-	-	-	-	-	-	-	-
#6 Dropwizard v2.0	-	-	-	-	-	-	-	-	-	-	-	-	-
#7 Blade v2.0.15	●	-	-	-	-	-	-	-	-	-	-	-	-
#8 ZK v9.0	●	-	-	-	-	-	-	-	-	-	-	-	-
#9 Apache Struts v2.3	●	-	-	-	-	-	-	-	-	-	-	-	-
#10 Apache Wicket v8.8	-	-	-	-	-	-	●	-	-	-	-	-	-
PHP													
#1 Laravel v7.0	●	-	-	-	-	-	-	-	-	-	-	-	●
#2 Symfony v5.0.8	●	-	-	-	-	-	-	-	-	-	-	-	●
#3 Slim v4.0	●	-	-	-	-	-	-	-	-	-	-	-	●
#4 CakePHP v4.0	-	-	●	●	-	-	-	-	-	-	-	-	●
#5 Zend/Laminas v3.1.1	●	-	-	-	-	-	-	-	-	-	-	-	-
#6 CodeIgniter v3.1.11	-	-	-	●	-	-	-	-	-	-	-	-	-
#7 FuelPHP v1.9	-	-	-	●	-	-	-	-	-	-	-	-	-
#8 Yii2 v2.0.35	-	-	-	●	-	-	-	-	-	-	-	-	●
#9 Phalcon v4.0	●	-	-	-	-	-	-	-	-	-	-	-	-
#10 Li3 v1.2	●	-	-	-	-	-	-	-	-	-	-	-	-
C#													
#1 ASP.NET Web Forms v4.8	-	●	●	●	-	-	-	-	-	-	-	-	-
#2 ASP.NET MVC v4.8	-	●	●	●	-	-	-	-	-	-	-	-	-
#3 ASP.NET Core v3.1	-	●	●	●	-	-	-	-	-	-	-	-	●
#4 Service Stack v5.9.0	-	●	●	●	-	-	-	-	-	-	-	-	-
#5 Nancy v2.0	-	-	●	●	-	-	-	-	-	-	-	-	-
Total	18	4	12	22		1	4						10

Legend: ● = CSRF defense; - = not offered;
red = no official CSRF defense;

Table 8: Mapping between web frameworks and CSRF defenses they implement.

	Token Gen.				Token Leakage				Cookie Integrity		Implementation Mistakes								
Web Framework	Insecure Randomness	Insuff. Token Randomness	Insuff. Key Randomness	Vuln. Crypto Libraries	BREACH	Referrer Leakage	Over-perm. CORS Conf.	Timing Attacks on Cmp.	Cookie Tossing	Cookie Jar Overflow	Insecure Token Mapping	Safe HTTP Met. check	Unsafe HTTP Met. check	HTTP Method Override	Logical Errors	XS WS Hijacking	Replay Attack	Faulty Ref./Orig. Check
JavaScript																		
#1 Express	-	-	N	-	-	P	-	-	P	P	-	N	-	-	-	-	P	-
#2 Meteor	-	-	-	-	P	-	-	-	-	-	-	-	-	-	-	-	P	-
#3 Koa	-	-	N	-	-	P	-	-	-	-	-	R	-	-	-	U	R	-
#4 Hapi	-	-	-	-	R	-	-	N	R	R	-	R	N	-	-	-	R	-
#5 Sails	-	-	-	-	-	P	-	-	-	-	-	P	-	-	-	-	P	-
#6 Fastify					-	F		-	P	P	-	P	-		-	-	P	-
#7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
#8 Total.js	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
#9 AdonisJs					-	-		-	-	-	-	P	-		-	-	P	-
Python																		
#1 Flask	-	-	N	-	-	-	-	-	-	-	-	N	-	-	-	N	N	N
#2 Django	-	-	-	-	-	-	-	-	N	N	-	N	-	-	-	P	N	P
#3 Tornado	-	-	N	-	-	-	-	-	N	N	-	R	-	-	-	-	N	-
#4 Bottle	-	-	M	-	-	-	-	-	M	M	-	-	-	-	-	M	-	-
#5 Pyramid	-	-	R	-	R	-	-	-	-	-	-	R	-	-	-	-	R	R
#6 Falcon					-	-		-	-	-	-	-	-	-	-	-	-	-
#7 Zope					M	-		M	M	M	-	M	-		-		M	-
#8 Masonite					F	-		F	F	F	-	N	F		-		F	-
#9 TurboGears					F	-		F	P	P	-	-	-		-		F	-
#10 Web2py					-	-		-	-	-	-	-	-		-		-	-
Java																		
#1 Spring	-	-	-	-	N	N	-	F	-	-	-	N	-	-	-	N	N	-
#2 Play	-	-	R	-	-	R	N	-	-	-	-	R	-	-	-	R	R	-
#3 Spark	-	-	-	-	F	-	-	F	-	-	-	N	-	-	-	R	F	-
#4 Vert.-Web	-	-	F	-	F	-	-	F	F	F	-	N	-	-	F	-	F	-
#5 Vaadin	-	-	-	-	-	N	-	F	-	-	-	-	-	-	-	-	N	-
#6 Dropwizard					-	-		-	-	-	-	-	-		-		-	-
#7 Blade					U	-		-	-	-	-	U	U		-		U	-
#8 ZK					-	-		-	-	-	-	-	-		-		P	-
#9 Apache Struts					N	-		N	-	-	-	N	-		-		-	-
#10 Apache Wicket					-	-		-	-	-	-	-	-		-		-	-
PHP																		
#1 Laravel	-	-	R	-	R	-	-	-	-	-	-	R	-	-	-	-	R	-
#2 Symfony	-	-	-	-	N	-	-	-	-	-	-	-	-	-	-	-	N	-
#3 Slim	-	-	-	-	-	-	-	-	-	-	-	N	-	-	-	-	F	-
#4 CakePHP	-	-	F	-	F	-	-	-	P	P	-	N	-	F	F	-	N	-
#5 Zend/Laminas	-	-	-	-	-	-	-	P	-	-	-	-	-	-	-	U	P	-
#6 CodeIgniter					U	-		-	U	U	-	U	U		-		-	-
#7 FuelPHP					U	-		-	U	U	-	U	-		-		-	-
#8 Yii2					-	-		-	P	P	-	R	-		-		P	-
#9 Phalcon					-	-		-	-	-	-	-	-		-		-	-
#10 Li3					-	-		-	-	-	-	-	-		-		N	-
C#																		
#1 ASP.NET Web Forms	-	-	-	-	-	-	-	M	-	-	-	-	-	-	-	M	M	-
#2 ASP.NET MVC	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	N	N	-
#3 ASP.NET Core	-	-	-	-	-	-	-	-	-	-	-	N	-	-	-	N	N	-
#4 Service Stack	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	U	U	-
#5 Nancy	-	-	-	-	M	-	-	M	M	M	-	-	-	-	-	-	M	-

Legend: blue = CSRF defense via official external library; red = no official CSRF defense;
- = already secure; F = fixed; P = fix in progress; N = not confirmed (risk acceptance);
R = under review; U = unanswered; M = not maintained anymore; empty cell = not evaluated;

Table 9: Summary of vulnerability disclosure