Behind the Curtain: A Server-Side View of Web Session Security

Simone Bozzolan
Università Ca' Foscari Venezia
simone.bozzolan@unive.it

Stefano Calzavara
Università Ca' Foscari Venezia
stefano.calzavara@unive.it

Florian Hantke
CISPA Helmholtz Center for Information Security
florian.hantke@cispa.de

Ben Stock
CISPA Helmholtz Center for Information Security
stock@cispa.de

***Abstract*—Since the HTTP protocol is stateless by design, web applications have to implement client authentication by means of web sessions. Given the importance of client authentication, the web security community investigated session security at length. However, prior work in the field primarily focused on black-box testing, which has very limited access to the server-side logic of the web application. This curtain prevents the analysis of relevant session security aspects, such as cryptographic key management, and provides limited insights into why vulnerabilities arise due to insecure programming practices. In this paper, we go through the process of creating a representative dataset of open-source web applications and perform the first measurement of web session security based on static analysis of server-side code. From our distinctive vantage point, we are able to analyze a number of security practices that cannot be assessed through black-box testing alone. Our research analyzes around 1,200 web applications built using the Django and Flask web development frameworks for Python. Our study unveils a number of vulnerabilities and bad programming practices in real-world applications, while shedding light on how key design choices of Django and Flask impact the security posture of web applications.**

***Index Terms*—web security, web sessions, static analysis.**

I. Introduction

Web applications routinely rely on client authentication to restrict access to personal data, e.g., profile pages, and sensitive functionality, e.g., premium services gated by a paywall. Since the HTTP protocol is stateless by design, web applications have to implement client authentication by leveraging the *session* abstractions available in popular web development frameworks and libraries. Despite their apparent simplicity, session implementations can suffer from a range of security threats [1]. Prior research investigated the prevalence of a number of vulnerabilities, like session hijacking [2], session fixation [3], and cross-site request forgery [4]. Detection of session vulnerabilities is normally performed using black-box testing techniques [5] and even the OWASP Testing Guide has a dedicated chapter on session management testing [6].

Black-box testing is great because it is simple to use, amenable to automation, and applicable to any web application without requiring access to its source code. Yet, not all the relevant security aspects of session management can be meaningfully assessed by black-box testing alone, because important security checks are often performed by opaque server-side logic. For example, password hashing is an important component for the security of account creation, but can only be assessed through a server-side view of the web application logic. As another example, sessions are often protected by cryptographic keys, but the security of key management cannot be analyzed without having access to the web application backend. Another limitation of black-box testing is that it can detect vulnerabilities, but cannot explain why vulnerabilities arise, because it has limited visibility of what web developers are doing. White-box analysis of secure session management can investigate aspects that black-box testing normally overlooks or struggles to deal with. In addition, it can expose insights into secure and insecure coding practices, thus shedding light on useful design principles and relevant limitations of web development frameworks that developers face when writing their applications.

In this paper, we fill a significant gap in web security research by creating a dataset of relevant open-source web applications and performing the first measurement of web session security based on static analysis of server-side code. From our distinctive vantage point, we are able to analyze a number of security practices that cannot be assessed through black-box testing, but that can be fruitfully checked by static analysis. Our analysis does not just identify dangerous security practices in the wild, but it also investigates the impact of different design choices of existing web development frameworks in terms of the resultant security guarantees of web applications based on real-world data. Getting access to this novel vantage point is challenging, though, because we first have to come up with a relevant dataset of web applications to analyze. This is a difficult task for multiple reasons. Although online repository platforms like GitHub host millions of projects, scraping web applications to analyze requires a lot of care because repositories making use of web development frameworks are not necessarily web applications but could be, e.g., web development libraries. Moreover, not all web applications are worth analyzing: for example, deliberately vulnerable applications created for security challenges and insecure applications available as tutorial code should not be taken into account within a credible security assessment. We here propose and implement a methodology to build a solid dataset of open-source web applications to analyze.

After solving the main challenges associated with dataset construction, we use GitHub's CodeQL analyzer [7] and write queries to capture key aspects of session security measured on the server side. Using CodeQL, we analyze the security posture of around 1,200 web applications developed in Python using the Django [8] and Flask [9] frameworks. The choice of Django and Flask is motivated by the different design philosophies and implementation choices of the two frameworks, with Django being a complex framework with security built-in and Flask espousing a minimalist approach where security is largely delegated to web developers and external libraries.

Contributions: In essence:

- We propose a methodology to assess server-side security issues by first constructing a large-scale dataset of relevant applications (§III) and then analyzing it with CodeQL (§IV). For the research community to leverage this knowledge and enable more research on server-side security issues, we make our code and data available [10].
- We report on session management security on the server side, investigating cryptographic key usages, adoption of built-in CSRF protection, and practices to better protect against session hijacking (§V).
- We present insights into server-side implementations of the account creation process, with a focus on the analysis of password policies and password hashing (§VI).

We summarize the main take-away messages of our study and report limitations in §VII.

II. Background

The HTTP protocol is based on a request-response paradigm involving a client (e.g., a browser) and a server. Since HTTP is stateless, the server relies on client-side state information, such as *cookies*, to keep track of previous interactions and build a stateful *session* abstraction. This is the most widespread technique for implementing client authentication on the Web. When a user is prompted for their access credentials, e.g., username and password, the HTTP request triggered by form submission transmits them to the server. After checking their validity, the server uses the corresponding HTTP response to set a cookie on the client, thus establishing a session. This cookie is automatically attached to future HTTP requests to the server. By reading the information stored in the cookie, the server can restore the state of the session, e.g., to authenticate the user and restore their previous interactions.

Sessions can be broadly classified into two categories. In *server-side sessions*, the cookie stores just a random session identifier, which is used to pinpoint a specific session on the server; the session information is primarily stored at the server side, e.g., within a database indexed by session identifiers. In *client-side sessions*, the session information is set mainly within the cookie itself, e.g., using a digitally signed JSON.

Figure 1 presents the code of a Flask application using the popular Flask-Login library to implement client-side sessions based on cookies signed with the secret key at line 6. It defines two *routes*, i.e., the HTTP endpoints `/login` (lines 9-15) and `/private` (lines 17-20). Once an HTTP request reaches a route, it activates the corresponding *view*, i.e., the functions `auth` and `dashboard` in our case. The first route extracts authentication credentials from incoming HTTP requests: if access credentials are correct, it authenticates the user by calling Flask-Login's `login_user()` function and redirects the client to the private area. The second route renders some HTML document, e.g., granting access to security-sensitive functionality. Access to the second route is protected by means of the `@login_required` decorator. If the client does not send a signed cookie established by a previous invocation to `login_user()`, an error page is automatically returned.

```
1  from flask import Flask, render_template,
       redirect, url_for, request as r
2  from flask_login import LoginManager,
3      login_user, login_required
4
5  app = Flask(__name__)
6  app.secret_key = 'ubersecret'
7  login_manager = LoginManager(app)
8
9  @app.route('/login', methods=['POST'])
10 def auth():
11     user = r.form['username']
12     if valid_creds(user, r.form['password']):
13         login_user(User(user_id=user))
14         return redirect(url_for('private'))
15     return render_template('error.html')
16
17 @app.route('/private')
18 @login_required
19 def dashboard():
20     return render_template('private.html')
```

Fig. 1: Example Flask application.

III. Dataset Construction

Reliable web measurements require a solid dataset that reflects the real world as best as possible. Traditional web security measurements are performed over lists of popular websites, such as Tranco [11]. Unfortunately, there is no similar standard for open-source web applications. Prior work on server-side security [12], [13] evaluated applications from the Bitnami catalogue [14]. However, Bitnami is small in size and currently contains just 120 applications. Moreover, it includes applications developed with different programming languages, meaning that it is a challenging target for static code analysis. As it turns out, coming up with a solid dataset is particularly subtle. We here discuss how we automatically create a dataset of popular open-source web applications and how we post-process it to ensure its representativeness.

A. Initial Dataset

For our dataset and analysis, we focus on web applications written in Python, which is one of the most popular programming languages nowadays according to recent statistics [15], [16]. Besides being widely popular, Python is an interesting playground because it allows us to focus on two well-known and relevant web development frameworks: Django [8] and Flask [9]. These are the most popular frameworks in the Python ecosystem according to prior work [17], [18] and a preliminary analysis based on GitHub search (Django and Flask are both used twice as often as the third most popular framework, which is FastAPI). Moreover, Django and Flask are interesting case studies in their own right, because they embrace different philosophies and implement different approaches to session management. Django is monolithic and provides native facilities for secure session management, while Flask is minimalist and requires developers to rely on external libraries for most tasks. In addition, Django implements sessions using server-side state by default, while Flask relies on client-side state, meaning that they cover the two traditional ways to implement web sessions.

With the list of frameworks decided, we then go on to collect a set of GitHub repositories that we could analyze for session management issues. We leverage the GitHub REST API [19] for this task. In total, we identified 296,913 repositories containing uses of Django and 110,694 repositories containing uses of Flask. Given the large number of repositories we found for each framework, analyzing all of them would not be feasible. Moreover, this is not even desirable for multiple reasons. One of the reasons is that not all the identified repositories require some form of session management, e.g., some web applications do not offer a private area. To restrict our focus to those applications that may implement session management, we apply an additional filter: for Django applications, we require the inclusion of the `django.contrib.auth` authentication module; for Flask applications, we require the inclusion of the Flask-Login library, which is the most popular authentication library according to prior work [18]. After this step, we were left with 115,301 Django repositories and 20,007 Flask repositories where authentication might be required.

These repositories do not necessarily include representative applications to analyze yet, e.g., they might include toy examples with no built-in security, solutions to academic homework, prototype software that was never released, etc. To improve the representativeness of our dataset, we considered a number of criteria to decide which applications to keep: (i) *Number of stars*: this metric is a standard indicator of the popularity of a repository. (ii) *Number of contributors*: this metric estimates both the popularity and the complexity of the application. (iii) *Number of commits*: this metric shows that the project has been actively developed, at least for a while. (iv) *Year of last commit*: this metric ensures that the project is not outdated (we only consider commits from users, filtering out commits from automated bots such as Dependabot). (v) *Source code*: we use the size of the Python code as a proxy for the complexity of the application and we just keep applications including some HTML and CSS, because we are interested just in web apps.

We decided to only keep repositories whose last commit was performed in 2020 or later. For the other metrics, we performed a preliminary data analysis step to understand their distribution. As it turns out, the distributions of stars and contributors are highly skewed because a vast amount of repositories have no stars and no contributors besides their creator. We decided to consider as "most representative" just those repositories for which all the four metrics fall in the fourth quartile (top 25%). After such filtering, we were left with 2,997 Django repositories and 692 Flask repositories.

B. Dataset Post-Processing

We performed a preliminary manual investigation of the remaining repositories, and we observed that, despite our efforts to improve representativeness, a significant number of them were not web applications or were not relevant for security analyses. For example, we identified many popular libraries used within existing web applications that are not web applications themselves. Moreover, we identified many web applications that are not intended for production use, such as tutorial code or capture-the-flag challenges. These applications are popular and actively maintained, but their inclusion would downgrade the representativeness of our dataset. Notably, some of these web applications are even *deliberately* vulnerable to teach web security concepts.

To mitigate this problem, we considered two different post-processing approaches, one NLP-based and one GPT-based. Details of the different approaches and the final post-processing process can be found in Appendix A. In the end, we opted for NLP-based filtering as our post-processing module. Our final dataset comprises 1,999 repositories: 1,620 Django and 379 Flask. We make our dataset, GitHub crawler and filtering routines available to other researchers to support future work on server-side security analyses [10].

IV. White-Box Analysis Methodology

We now explain how we define and systematically look for web session vulnerabilities in our dataset.

A. Analysis Scope

Prior work on web measurements investigated relevant aspects of web session security, such as the insufficient adoption of cookie security attributes [20], [21] and the insecure configuration of HTTP headers [22], [23]. Moreover, prior work on web session security also proposed black-box testing to detect classic vulnerabilities like session fixation and session hijacking [5], [2], [24]. These works exclusively focus on live websites, hence have no visibility of the server-side code.

In contrast, our dataset provides access to the source code of the web applications, allowing us to use static analysis to detect unsafe programming practices. Given this distinctive vantage point, we focus on vulnerabilities that are difficult or impossible to detect via black-box testing to clarify the value of our approach. For example, we analyze the management of cryptographic keys at the backend and the security of password hashing practices based on source code analysis.

B. Queries and Dataset

We developed a number of CodeQL [7] queries to detect insecure programming practices in web session implementations. We use CodeQL because it has an expressive query language that we can use to tailor our security analysis around relevant issues. We may have chosen other static analysis tools, e.g., SonarQube, but for our purposes any analysis tool with a configurable query language would have worked. Additionally, CodeQL provides some publicly available queries [25], but they do not address the specific aspects considered in our

TABLE I: Number of analyzed GitHub repositories.

Purpose	**Django**	**Flask**	**Total**
Session management	882	307	1,189
Account creation	294	84	378

analysis. Indeed, such queries are usually quite general and limited to checking whether a particular security feature is enabled. Therefore, while they represent a useful starting point, their number is limited, and more specialized queries are required to support our analysis. All queries are available online [10].

Queries fall into two broad categories, i.e., session management and account creation, and are correspondingly performed over two subsets of applications:

- *Session management:* to analyze the security of session management practices, we restrict our focus to web applications performing invocations to the login functions of Django or Flask-Login, and checking at least once whether the user is logged in or not. This way, we are sure that these applications authenticate users and restrict access to specific functionality.
- *Account creation:* to analyze the security of account creation, we start from the previous set of applications implementing session management. We further restrict our focus to web applications enabling the creation of new accounts. For Django applications, we only keep repositories where we find instances of the `UserCreationForm` class, which may be used for account creation. For Flask applications, we leverage the observation that Flask-WTF and WTForms are the most popular libraries for handling forms in our dataset. In total we found 202 applications using either Flask-WTF or WTForms, and we did not find any usage of other widely adopted form-handling libraries, such as Deform. We look for registration forms by (i) searching for instances of the main form classes of Flask-WTF and WTForms including at least one password field, and (ii) filtering them only to keep those whose name includes at least one keyword related to account creation, such as “register” or “signup”.

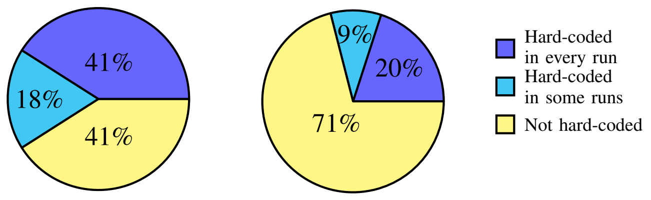
Table I reports the number of repositories that we eventually considered for our security analysis on session management and account creation based on this methodology. For space reasons, examples of vulnerable applications (confirmed through manual investigation) are reported in Appendix B. We responsibly disclosed the confirmed vulnerabilities and we report on the received feedback in Appendix C.

V. Session Management

We analyze different aspects of session management which are reported in the following.

A. Cryptographic Keys

Secure session management often requires the use of cryptographic keys. Indeed, both Django and Flask require operators to set a *secret key* within their web application. We use CodeQL to identify hard-coded secret keys within web applications. Using hard-coded secret keys is a dangerous practice, because the secret key may be available to attackers who inspect the web application code on GitHub. When using client-side sessions in the style of Flask, the secret key is used to digitally sign session cookies: if the secret key is known to the attacker, such cookies may be forged and enable impersonation attacks. In the case of Django, the secret key is used for multiple purposes, such as protecting the password reset functionality and any usage of cryptographic signing [26]. Correct configuration and management of the secret key is difficult to assess at scale for black-box testing strategies that do not have access to the source code, but it can be fruitfully checked by static analysis.



(a) Django (b) Flask

Fig. 2: Distribution of secret key management practices divided by framework.

1) CodeQL Queries: The queries perform data flow analysis to associate each application to one of the following security classes: (i) the secret key is hard-coded to a constant value within the source code; (ii) the secret key may be set to a hard-coded default value in some program runs, e.g., when a configuration file is missing; or (iii) the secret key is never set to a hard-coded constant value, e.g., it is read from an environment variable or set to a new random string every time the application is launched. Details on how the queries work can be found in our online repository [10].

2) Analysis Results: Figure 2 shows how Django and Flask applications are distributed over the three security classes. Note that the amount of applications using hard-coded secret keys, either in every run or under specific circumstances, is quite significant in practice. In total, we identified 424 applications, out of the analyzed 1,189 applications, that hard-code the secret key to some constant value in every program run in their source code (36%). Overall, Flask applications have fewer hard-coded secret keys in comparison to Django applications. We detected hard-coded secret keys in 62 out of 307 Flask applications (20%), while we identified hard-coded secret keys in 362 out of 882 Django applications (41%). This phenomenon might be explained by the fact that Django automatically generates a random secret key and hard-codes it in the `settings` module when starting a new project. Flask, in turn, leaves key creation entirely in charge of web developers. Empirically, it thus seems that Django’s approach to secret key creation is potentially dangerous in practice, because almost half of the repositories rely on hard-coded secret keys that the operators of their web applications are not forced to update.

Moreover, both Django and Flask recommend a minimum length of the secret key, with Django recommending stricter length requirements (50 characters vs. 24 characters). These requirements are recommended to mitigate brute-force attacks where the attacker tries to reconstruct the correct secret key by exhaustive enumeration. Looking at the hard-coded secret keys, we observed that 47 out of 62 Flask applications using hard-coded secret keys set a key that is shorter than the recommended key length (76%), while this is the case just for 125 out of 362 Django applications (35%). This highlights that, when developers are required to generate secret keys themselves, they may not comply with recommended security practices. In the case of Flask applications, the use of short and predictable session keys is particularly concerning because brute-force attacks can be performed offline and may not involve any interaction with the server. Indeed, the attacker can acquire a valid session cookie from the web application, inspect its content (the cookie is not encrypted) and try to enumerate keys until a correct signature matching the cookie is generated. Once the correct key has been found, the attacker can forge new session cookies.

In the end, our analysis shows an intriguing duality. Django applications are more likely to expose hard-coded secret keys than Flask applications, while Flask applications are more likely to use secret keys that are vulnerable to brute-forcing than Django applications. Based on this, we conclude that Django's choice of automating secret key creation is useful in practice, but the framework should be made more secure by avoiding the inclusion of hard-coded secret keys in the `settings` module, because developers are not required to update them. We recommend modifying Django's approach so that the `settings` module does not contain any secret key by default, thus triggering an exception (with the current implementation); secret key creation, in turn, should be deferred to a configuration script to be run before deployment, which would populate the `settings` module with an automatically generated secure secret key. Django's exception message may be updated to report the need to run the configuration script.

B. Cross-Site Request Forgery (CSRF)

CSRF is one of the most well-known web vulnerabilities [27]. It abuses the automated attachment of session cookies to HTTP requests to forge authenticated requests from the victim's browser and trigger security-sensitive actions on their behalf. Protection against CSRF can be implemented in different ways, such as the use of secret tokens to authenticate security-sensitive requests in addition to cookies. Web development frameworks normally offer simple declarative techniques to mitigate CSRF built on top of established protection mechanisms. Developers can then annotate specific views/forms requiring CSRF protection or conversely mark escape hatches where CSRF protection is undesired.

1) CodeQL Queries: We use CodeQL to investigate how developers are taking advantage of built-in protection mechanisms against CSRF. Based on the results of our queries, we are able to classify each application into one of four categories: (i) CSRF protection is activated by default and never deactivated; (ii) CSRF protection is activated by default, but deactivated on some views; (iii) CSRF protection is deactivated by default, but activated on some views; or (iv) CSRF protection is deactivated by default and never activated.

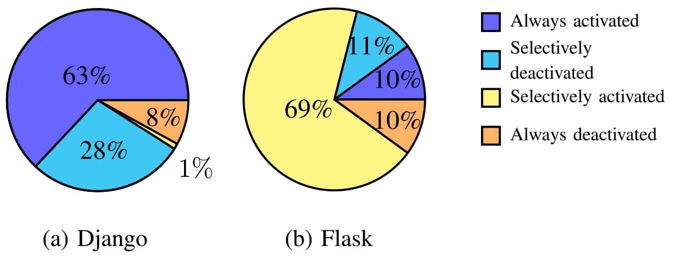


Fig. 3: Distribution of CSRF protection levels per framework.

We then write CodeQL queries to better understand the security implications of the views that are not protected against CSRF. In particular, we identify *sensitive* views as those requiring authentication and writing into the underlying database. Intuitively, sensitive views require CSRF protection because the attacker may abuse an authenticated session by forging HTTP requests from the victim's browser to trigger a side-effect on the web application. Details on how the queries work can be found in our online repository [10].

2) Analysis Results: In total, we identified 1,048 applications (88%) that use the CSRF protection patterns supported by our analysis: 165 of them are in our set of Flask applications, while 883 fall in our set of Django applications. This shows that built-in protection mechanisms against CSRF are popular among developers and their adoption is worth additional attention. The remaining 141 applications (12%) do not use one of the CSRF protection patterns supported by our analysis. As a result, no conclusions are drawn, and they are excluded from further analysis.

Figure 3 shows how Django and Flask applications are distributed over one of the four mentioned security classes. Observe that the majority of Django applications (63%) mitigate CSRF on all views, while the majority of Flask applications (69%) just selectively activate protection. This dichotomy can be attributed to the default settings of their respective CSRF protection mechanisms. By default, Flask-WTF enables CSRF protection only for forms that are created extending `FlaskForm`, while Django enables CSRF protection globally by default. Hence, there exists a compelling argument in favor of secure-by-design frameworks, as the architectural choices embedded within a framework significantly impact the security posture of resultant applications. Although Flask-WTF also supports global protection at the application level by means of the `CSRFProtect` class, we observe that protection at the individual form level is more widespread in practice.

To better understand the importance of CSRF protection (or the lack thereof) in practice, we complement our analysis with an investigation of the sensitivity of relevant views. We focus on applications where CSRF protection is either activated by default but deactivated in some views, or deactivated by default but activated in some views. In both cases, we look for the presence of sensitive views among those that are unprotected against CSRF, thus estimating the potential security risks of incomplete CSRF protection. In total, we identified 27 applications with unprotected sensitive views out of the 266 applications where CSRF protection was selectively deactivated (10%) and 48 applications with unprotected sensitive views out of the 122 applications where CSRF protection was selectively activated (39%). This gives clear evidence of the practical advantages of secure-by-default solutions in the style of Django, leading to a significantly reduced risk of leaving sensitive views unprotected. Besides the sheer quantitative finding of our experiment, we also observe that developers making use of decorators like `@csrf_exempt` are explicitly acknowledging the lack of protection for specific views, while approaches based on selective protection like the extension of `FlaskForm` may lead to overlooking security-sensitive functionality. To confirm this, we tokenized the name of sensitive views missing CSRF protection in Flask applications, observing the presence of several concerning terms, such as "delete" (122 occ.), "add" (64 occ.) and "edit" (53 occ.).

In the end, our analysis provides insights into how developers are making use of built-in protection mechanisms available in popular web frameworks to reduce the potential attack surface. Moreover, it can complement existing dynamic approaches to CSRF detection [12], [28] to improve their coverage and thus mitigate one of the most significant limitations of dynamic analysis. Indeed, we were able to identify new confirmed CSRF vulnerabilities in existing applications just by inspecting the unprotected sensitive views identified by our queries (see Appendix B). Feeding such endpoints to existing black-box testing tools would be useful to improve their coverage and automatically confirm the detected vulnerabilities, similar to what we did in our own scrutiny.

C. Session Protection

Flask-Login implements additional protection against session hijacking by defining different levels of *session protection*, which determine whether sessions should be tied to the requesting client. In particular, Flask-Login computes a secure hash of the IP address and user agent of the requesting client (for short, *client identity* in this paper) to determine whether a session cookie was stolen and is being reused on a different device. There are three levels of session protection available: (i) None: the client identity is never checked, i.e., no additional protection against session hijacking is in place; (ii) Basic (default): the client identity is checked just for those functionalities requiring a fresh login, i.e., annotated with the `@fresh_login_required` decorator; (iii) Strong: the client identity is checked for every request.

Of course, session protection improves security against session hijacking at the expense of usability, since users might be required to login more frequently due to session invalidation, e.g., when the same device is assigned a different IP address.

1) CodeQL Queries: We run multiple CodeQL queries to understand the use of session protection in Flask applications and categorize the results in the following way: (i) No protection: we check whether the application lacks enforcement of any additional protection against session hijacking; (ii) Protection on some views: we check whether the application enforces additional protection against session hijacking on specific views; (iii) Complete protection: we check whether all the views of the application enjoy additional protection against session hijacking. Details on how the queries work can be found in our online repository [10].

2) Analysis Results: In total, we have 307 Flask applications that distribute over the three security classes as follows: 281 applications (92%) do not benefit from session protection, 3 applications (1%) activate session protection just on some views by using the `@fresh_login_required` decorator, and 23 applications (7%) enforce session protection on all views. This shows that developers exhibit a highly polarized behavior with respect to session protection: most applications do not use it at all, while a few applications activate it everywhere; the number of applications using session protection just on some specific views is negligible.

To gain deeper insights into how developers deploy session protection, we additionally searched through the commit histories of the 307 Flask applications, tracking the usage and changes of the default protection behavior via the `session_protection` field. We utilized *git grep* for this experiment, as searching through tens of thousands of commits would be extremely resource-intensive with CodeQL. Despite the limitations of this simple syntactic match, our grep search identified several matching repositories. We found a total of 31 applications that overwrote the default session protection setting at some point (10%). 11 applications moved to a more secure session protection configuration: in all cases, this meant activating strong session protection. On the contrary, 5 applications moved from an initially stronger protection to a less secure configuration. All of these cases deactivated protection due to documented login problems or code refactoring. The remaining 15 applications explicitly overwrote the `session_protection` field in the initial commit, or the first time user management was added, and never changed it again. 12 of these applications implemented strong session protection from the beginning, while only two applications explicitly set a basic protection and one configured an insecure protection from the initial commit.

In summary, our findings indicate that the majority of developers do not change their initial configuration of session protection and, when modifications are made, they are primarily to enhance security rather than to reduce it. Since several applications successfully activated strong session protection from the very beginning and we only observed a tiny number of regressions to less secure configurations, our findings suggest that more developers may likely be able to strengthen session protection in their applications.

VI. Account Creation

Account creation is a delicate process because web applications should force their users to choose strong passwords and implement appropriate protection mechanisms for them. Here, we analyze the key features of the password policies enforced in the web applications of our dataset and the password hashing techniques they adopt during account creation.

A. Password Policies

It is well known that passwords should satisfy minimum strength requirements to be secure against online and offline brute-force attacks [29]. Password strength is difficult to estimate using black-box techniques: for example, Alroomi and Li proposed a sophisticated inference algorithm for password policies, which is computationally heavy and suffers from possible inaccuracies [30]. Unfortunately, their analysis revealed limited deployment of client-side password strength checks, which would be a natural avenue to analyze password security without having access to the web application code. Since our methodology is based on source code analysis, we are in a privileged position to analyze password strength based on server-side security checks.

1) CodeQL Queries: We use CodeQL to collect insights into the password policies enforced by the web applications in our dataset. In particular, we implement queries to detect registration forms using the heuristics in Section IV-B and extract *validators* associated with their password fields. Details on how the queries work can be found in our online repository [10].

A relevant point to note here is that Django automatically enforces a default password policy when the registration form is submitted, while Flask-WTF does not implement any default policy. Moreover, Django offers a broader variety of password validators than Flask-WTF, though it lacks the regular expression and maximum length validators present in Flask-WTF. While Flask-WTF encompasses validators for minimum length, maximum length, and regular expression matches, Django offers validators for minimum length, password-username similarity, detection of common passwords, and identification of passwords consisting entirely of numbers.

2) Analysis Results: In total, we identified 378 applications (294 for Django and 84 for Flask) including a registration form, out of which 272 perform some validation of their password fields (72%). This shows that validators are a popular tool, yet password validation may still be overlooked by developers in practice, or even deliberately deactivated when the library would have it enabled by default. Django's choice to enable password validation by default pays off. Indeed, we observe a greater proportion of Django applications than Flask applications performing some form of password validation (78% vs. 50%). Django applications without any password validation turned off the default password policy by removing the default validators from the password field.

The most common use case of validators is to enforce a minimum password length: we identified 264 applications

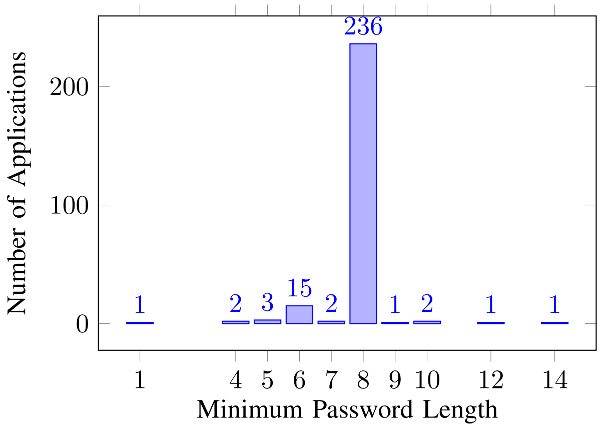


Fig. 4: Distribution of minimum password lengths.

doing that, i.e., 97% of the applications making use of validators check the enforcement of a minimum password length. Figure 4 shows the distribution of the minimum password length enforced by the applications using validators for this purpose. The distribution is skewed on length 8 because this is the default length required by Django, which aligns with the recommendation outlined by NIST [31]. The next most prevalent choice was a minimum length of 6 characters and there is a higher number of applications that relax the NIST-recommended minimum length requirement, rather than implementing a stricter requirement.

Perhaps surprisingly, most of the analyzed web applications do not check any structural property of their passwords at registration time besides their length. Django's default validators prevent the use of passwords that are too similar to the username, that are common according to a preloaded list, or that are entirely numeric, but they do not enforce any check on *password strength*. A relevant observation of our analysis is that the number of applications checking the use of specific sets of characters like lowercase letters, uppercase letters and symbols is quite limited. These checks may be normally performed using regular expressions, but the number of applications using custom validators and regular expressions amounts to just 13 (5%). Upon manual inspection, we observed that only 8 out of the 13 applications were using these validators to test password strength. The other 5 applications were either checking the absence of certain characters such as spaces, or implementing length checks. As this is a very low number, we would expect developers to use other methods to check password strength. However, this does not seem to be the case based on additional experiments that we carried out. First, we looked for the presence of popular password strength libraries like django-password-strength and zxcvbn-python, finding just 12 imports of the libraries across all of our dataset. Then, using CodeQL, we looked for instances where Python's `re` module was used to check password strength, by looking at whether the registration form's password field was checked using one of the `re` module's functions. We found that only a negligible amount of applications were performing such checks. Finally, we wrote CodeQL queries to identify all the applications accessing the password field of Django's built-in `UserCreationForm` class. Since this class automatically saves registered users in the database, developers only need to access its fields to implement custom functionalities, such as verifying password strength. After manual analysis of the matching applications, we found that only a negligible amount of them were accessing the password field to evaluate password strength. Hence, while it is conceivable that we may have overlooked some instances, all of our experiments suggest that developers largely neglect password strength checks. This can possibly be attributed to the fact that neither Django nor Flask provides built-in password strength validators.

B. Password Hashing

Passwords should not be saved in plaintext on the server to prevent their disclosure and reuse on different services upon data breaches; rather, a secure hash of the password should be stored. Techniques for secure password hashing aimed at mitigating offline brute-force attacks are well known, however, they are implemented by means of server-side logic, which cannot be assessed by black-box testing.

1) CodeQL Queries: We use CodeQL to verify compliance with the OWASP password hashing recommendations of April 2024 [32]. Recommended hashing algorithms include Argon2id, scrypt, PBKDF2, and bcrypt with specific configuration options. Details on how the queries work can be found in our online repository [10].

2) Analysis Results: In total, we identified 378 applications providing a registration form, out of which 372 applications implement some form of password hashing (98%), meaning that our enumeration of popular hashing libraries yields almost complete coverage. The 6 applications that do not perform any password hashing according to our queries have been discontinued, are not interesting from a security perspective or are false negatives, e.g., one application uses a hashing library not supported by our queries.

Figure 5 shows how many applications are using a recommended hashing algorithm with a secure or an insecure configuration. In total, 85% of the applications use a recommended hashing algorithm set in a secure configuration, showing that the importance of performing secure password hashing has been widely understood by developers over the years. We observe that the most popular hashing algorithm is PBKDF2 in a secure configuration, followed by scrypt in an insecure configuration. This trend can be explained by examining the popularity of the hashing libraries in our dataset: the most popular library is Django's built-in hashing library, whose default configuration uses PBKDF2 with a secure configuration, while the second most popular library is Werkzeug [33], whose default configuration uses scrypt with an insecure configuration. Interestingly, Werkzeug recently modified its default hashing algorithm [34], transitioning from PBKDF2 to scrypt, which is widely regarded as more secure. However, they did not align with OWASP recommendations in the default configuration of scrypt, as by default the

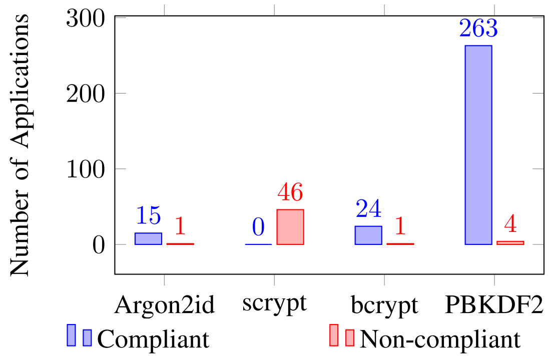


Fig. 5: Distribution of hashing algorithms divided by compliance with the OWASP guidelines.

CPU/memory cost parameter is set to 2^{15} [35], while OWASP recommends at least 2^{17} when using Werkzeug's default values for the other parameters [32]. This weakens the recommended protection level against offline brute-force attacks.

Our analysis also found 21 applications (6%) that apparently do not use any of the four hashing algorithms covered by our queries. We manually inspected all of them, corresponding to 7 Flask applications and 14 Django applications. As it turns out, all the Flask applications were using some version of the SHA algorithm, which does not offer robust protection against offline brute-force attacks. Luckily, most of these Flask applications make use of invocations to Werkzeug that would lead to runtime errors in modern versions of the library, e.g., SHA256 is now unsupported for password hashing. This means that these applications would be vulnerable only if they were run with an old version of Werkzeug. The Django applications, instead, were all false positives due to the presence of test configurations using MD5 for performance reasons, while the production configurations used stronger algorithms for password hashing, e.g., the default PBKDF2 algorithm. This shows again that the secure-by-default approach of Django is useful in practice, because potentially insecure hashing algorithms are confined to Flask applications alone.

VII. Discussion

With the results in mind, we now summarize the main findings of our study and acknowledge limitations.

A. Main Take-Aways

From a methodological point of view, we observe that *constructing a meaningful dataset of web applications is a challenging task* that requires careful filtering, as naive approaches like scraping GitHub for framework imports yield many irrelevant results. However, once the dataset is constructed, *CodeQL proves to be an effective and reliable tool* for analyzing session security, with very few timeouts or failures in practice. A careful manual analysis of the findings revealed that the number of false positives was generally low, thanks to the focus on syntactic code patterns that are well-suited to static analysis. Most of the false positives stemmed from applications with multiple configurations (e.g., separate settings for development and testing) or those using custom security mechanisms. In such cases, the false positives were due to conflicting configurations, e.g., enabling password strength validators on some password fields but not others, or replacing built-in CSRF protection with custom mechanisms. Although assessing whether these applications are actually secure or not would require careful scrutiny, such practices are generally considered dangerous, therefore these cases were manually reviewed or conservatively classified as insecure. Nonetheless, they were rare and did not impact the overall trends. In general, false positives were very limited and manageable through manual inspection, as most applications used a single configuration and adhered to the default settings of their respective libraries.

Our work shows that, with the right approach and methodology, analyzing the server-side logic of web applications at scale is possible. Hence, we are confident that future research can build up on our methodology and dataset to extend the scope of the security analysis to additional frameworks, programming languages and other facets of server-side security, such as database flaws, authorization issues, single sign-on and multi-factor authentication configurations.

From the point of view of session security, our analysis reveals several interesting insights. First, *security-by-default pays off in practice*. There are a number of areas where Django applications are more protected than Flask applications. For example, CSRF protection is always activated in 63% of the Django applications, while global protection against CSRF is enforced in just 10% of the Flask applications. Our empirical analysis also shows that automated CSRF protection is positively correlated with a reduced risk of leaving sensitive views unprotected against attack attempts. Moreover, the distribution of the minimum password lengths enforced by the analyzed web applications is highly skewed towards 8, which is the default password length for Django applications. Also for password hashing we observe a very important role of security-by-default: most of the invocations to password hashing functions are made with the default parameters set in the chosen cryptographic library, meaning that most of the practical uses can be classified as secure or not just based on the default choices of library developers. To further corroborate the importance of security-by-default, we observed that opt-in defensive measures like the session protection feature of Flask-Login have limited practical adoption: just 8% of the analyzed Flask applications take advantage of session protection, although our analysis of commit histories suggests that a larger adoption might actually be feasible.

On the negative side, our analysis also shows that *security-by-default is not always properly designed*. Django's choice of automatically generating secret keys is great for enforcing a reasonable key length, but the choice of hard-coding the generated keys within a Python file might unduly expose web applications to the risk of being deployed on the Internet with a known cryptographic key. We observed hard-coded secret keys in 41% of the Django applications, while this practice affected just 20% of the Flask applications. To improve the security of Django applications, we suggest demanding the creation of secret keys to an installation script, thus mirroring the automated facilities offered to Django developers to end users as well. In general, we recommend the inclusion of secret keys within specific configuration files rather than in source files that can easily enter version control systems. Also, the use of default validators for password strength implemented in Django is undoubtedly useful, but the toolbox of password validators offered to web developers is lackluster. In general, it seems that validation of password complexity is uncommon in our dataset, with less than 5% of the applications performing such checks according to our analysis. Password strength can certainly be improved by enforcing stricter password complexity guidelines by default. We observed similar issues with security-by-default when analyzing password hashing practices: the scrypt implementation of Werkzeug runs by default with a configuration that is deemed insufficient against offline brute-force attacks by the OWASP guidelines, meaning that the Werkzeug developers failed at implementing security-by-default according to current best practices.

B. Limitations

The primary limitation of our study is its focus on the Python programming language, which, although popular, is just one specific language for web application development. This limitation is motivated by the fact that static analysis is inherently language-based, hence generalization to multiple languages requires significant engineering effort. Thanks to CodeQL's multi-language support, this effort mainly involves adapting the existing queries to account for framework-specific and language-specific features, and constructing suitable datasets of web applications. Although labor-intensive, this work does not demand novel design. That said, our coverage of two different web development frameworks with different design choices already pays off in terms of collected insights about security-by-default and support for secure web development. Another limitation of our study revolves around its focus on specific and widely used session management libraries. This targeted approach is motivated by the inherent challenges in analyzing custom session management implementations at scale. In custom scenarios, developers may integrate authentication atop their unique session cookies, making it challenging to distinguish them from other cookies serving different purposes. Consequently, our analysis excludes a detailed examination of custom session management practices, potentially overlooking pertinent security vulnerabilities. While dynamic analysis could overcome this limitation, this gap also presents an opportunity. By concentrating on well-established libraries widely adopted in numerous web applications, our findings shed light on best programming practices of broad significance with clear, practical implications. Compared to dynamic analysis, static analysis is prone to false positives, as previously discussed, and requires access to the source code, which is not always available. Nevertheless, we have shown that the number of false positives was limited and therefore had a negligible impact on the observed trends. Moreover, static analysis allowed us to assess security aspects, like password hashing, that dynamic analysis cannot deal with.

VIII. Related Work

We categorize existing work we compare with in two major research lines: dataset construction and web session security.

A. Dataset Construction

Numerous researchers have asked the question of how to create comprehensive datasets from software repositories. Cosentino et al. [36] published a meta-paper analyzing 93 academic papers to understand the empirical methods and datasets researchers use to conduct their studies. They identified various databases used in these studies, including GHTorrent [37], which served as an offline mirror of data from GitHub tailored towards academic studies, but stopped publishing new datasets in 2021. A similar project is GHArchive [38], yet it only provides status updates and activities of repositories and not the actual content itself. Another contribution with which researchers tried to find a solution for software datasets is the Boa paper [39]. In their paper, the authors propose a new domain-specific language and infrastructure designed to query large datasets of software repositories collected from platforms like GitHub or SourceForge. With their platform, they provide a service that enables fast prototyping of tests and analysis reproducibility. However, for our study, we were not interested in a dataset of general applications, but a sample set of Flask and Django web applications.

Besides the mentioned papers that try to provide general datasets, researchers also analyzed how to best investigate GitHub repositories, e.g., the PyDrill project [40], or how to find similar repositories for later analysis [41]. Koch et al. [42] analyzed various metrics of software repositories to understand the relationship between, for example, GitHub stars and download numbers, finding only a weak correlation, but noting the limitation of their study to client-side projects, suggesting different dynamics for server-side projects. While this field is still very unexplored, metrics like GitHub's stars are the only indicators we can use to find relevant projects. Due to the weak correlation, we decided to take a combination of various metrics. In terms of practical applications of the GitHub API, Wittern et al. [43] utilized the API to collect all available GraphQL schemes they could find.

B. Web Session Security

Prior work investigated different security threats against web sessions, see [1], [44] for a survey of this important research area. Protection against session hijacking was primarily assessed by checking the appropriate adoption of cookie security attributes, such as HttpOnly and Secure [20], [21]. CSRF protection was first analyzed in the classic paper by Barth et al. [27] and multiple defenses have been proposed [45], [46]. The dangers of CSRF have also been evaluated at scale in the wild by Sudhodanan et al., finding significant room for abuses [4]. More recently, Khodayari and Pellegrino measured the effectiveness of the SameSite cookie attribute for CSRF protection, quantifying its benefits and limitations [47]. Black-box testing strategies for secure session management were also systematized and presented by Calzavara et al. [5]. Later work used similar testing strategies to perform large-scale measurements of web session security in the wild [2], [24].

All these works clarified the practical relevance of different session security vulnerabilities, but did not investigate web session security from the eyes of web developers, i.e., in terms of programming practices detectable through source code analysis. The distinctive feature of this paper compared to prior work is its unique vantage point on server-side code, which is assessed by means of static analysis. This point of view allows us to investigate aspects that prior work based on black-box testing was unable to cover, such as cryptographic key management, which is only visible within the web application backend. Moreover, our diverse dataset of Django and Flask applications allows us to understand how different design choices of web development frameworks affect the security features eventually implemented by web developers.

IX. Conclusion

This study examined server-side web session security at scale by developing a methodology to build and analyze a dataset of open-source web applications, which we have publicly released together with the developed tools [10]. Our analysis of session management features and account creation configurations revealed how key design choices in popular frameworks like Django and Flask affect security. While security-by-default pays off in practice, it is not always optimally implemented. Based on our findings, we identified areas for improvement and provided specific recommendations for frameworks, libraries, and development practices.

In future work, we plan to expand our analysis to more programming languages and frameworks, broadening the studied security aspects to include database vulnerabilities, authorization, single sign-on, and multi-factor authentication. We also aim to advance static analysis techniques for web security by developing new methods better suited to the complexity of server-side code.

Acknowledgments

We thank the reviewers for their feedback. This research was supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - Next-GenerationEU CUP N.H73C22000890001 and by Agenzia per la cybersicurezza nazionale under the 2024-2025 funding programme for promotion of XL cycle PhD research in cybersecurity – CUP N.H71J24001710005. The views expressed are those of the authors and do not necessarily represent the funding institutions.

Use of AI-based tools: During the preparation of this work, the authors used ChatGPT in order to improve the readability and language of the manuscript. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the published article.

REFERENCES

[1] S. Calzavara, R. Focardi, M. Squarcina, and M. Tempesta, "Surviving the web: A journey into web session security," *ACM Comput. Surv.*, vol. 50, no. 1, pp. 13:1–13:34, 2017. [Online]. Available: https://doi.org/10.1145/3038923

[2] K. Drakonakis, S. Ioannidis, and J. Polakis, "The cookie hunter: Automated black-box auditing for web authentication and authorization flaws," in *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, J. Ligatti, X. Ou, J. Katz, and G. Vigna, Eds. ACM, 2020, pp. 1953–1970. [Online]. Available: https://doi.org/10.1145/3372297.3417869

[3] M. Johns, B. Braun, M. Schrank, and J. Posegga, "Reliable protection against session fixation attacks," in *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan, March 21 - 24, 2011*, W. C. Chu, W. E. Wong, M. J. Palakal, and C. Hung, Eds. ACM, 2011, pp. 1531–1537. [Online]. Available: https://doi.org/10.1145/1982185.1982511

[4] A. Sudhodanan, R. Carbone, L. Compagna, N. Dolgin, A. Armando, and U. Morelli, "Large-scale analysis & detection of authentication cross-site request forgeries," in *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*. IEEE, 2017, pp. 350–365. [Online]. Available: https://doi.org/10.1109/EuroSP.2017.45

[5] S. Calzavara, A. Rabitti, A. Ragazzo, and M. Bugliesi, "Testing for integrity flaws in web sessions," in *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part II*, ser. Lecture Notes in Computer Science, K. Sako, S. A. Schneider, and P. Y. A. Ryan, Eds., vol. 11736. Springer, 2019, pp. 606–624. [Online]. Available: https://doi.org/10.1007/978-3-030-29962-0_29

[6] OWASP, "Session management testing," https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/06-Session_Management_Testing/README, 2025.

[7] GitHub, "CodeQL," https://codeql.github.com/, 2025.

[8] Django Software Foundation, "Django," https://www.djangoproject.com/, 2005.

[9] Armin Ronacher, "Flask," https://flask.palletsprojects.com/, 2010.

[10] S. Bozzolan, S. Calzavara, F. Hantke, and B. Stock, "Artifacts," https://github.com/Asterius27/BTC-paper-artifacts, 2025, repository containing all of the artifacts related to this paper.

[11] V. L. Pochat, T. V. Goethem, S. Tajalizadehkhoob, M. Korczynski, and W. Joosen, "Tranco: A Research-Oriented Top Sites Ranking Hardened against Manipulation," in *Network and Distributed Systems Security (NDSS) Symposium 2019*. Internet Society, 2019.

[12] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow, "Deemon: Detecting CSRF with dynamic analysis and property graphs," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 1757–1771. [Online]. Available: https://doi.org/10.1145/3133956.3133959

[13] S. Khodayari and G. Pellegrino, "JAW: studying client-side CSRF with hybrid property graphs and declarative traversals," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. D. Bailey and R. Greenstadt, Eds. USENIX Association, 2021, pp. 2525–2542. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/khodayari

[14] Broadcom Inc., "Bitnami," https://bitnami.com/, 2025.

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

[16] Stack Overflow, "Stack Overflow Developer Survey 2023," https://survey.stackoverflow.co/2023/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2023, 2023.

[17] X. Likaj, S. Khodayari, and G. Pellegrino, "Where We Stand (or Fall): An Analysis of CSRF Defenses in Web Frameworks," in *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID '21. New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 370–385.

[18] M. Squarcina, P. Adão, L. Veronese, and M. Maffei, "Cookie crumbles: Breaking and fixing web session integrity," in *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, J. A. Calandrino and C. Troncoso, Eds. USENIX Association, 2023, pp. 5539–5556. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/squarcina

[19] GitHub, "GitHub REST API documentation," https://docs.github.com/en/rest, 2025.

[20] N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen, "Sessionshield: Lightweight protection against session hijacking," in *Engineering Secure Software and Systems - Third International Symposium, ESSoS 2011, Madrid, Spain, February 9-10, 2011. Proceedings*, ser. Lecture Notes in Computer Science, Ú. Erlingsson, R. J. Wieringa, and N. Zannone, Eds., vol. 6542. Springer, 2011, pp. 87–100. [Online]. Available: https://doi.org/10.1007/978-3-642-19125-1_7

[21] M. Bugliesi, S. Calzavara, R. Focardi, and W. Khan, "Cookiext: Patching the browser against session hijacking attacks," *J. Comput. Secur.*, vol. 23, no. 4, pp. 509–537, 2015. [Online]. Available: https://doi.org/10.3233/JCS-150529

[22] M. J. Kranch and J. Bonneau, "Upgrading HTTPS in mid-air: An empirical study of strict transport security and key pinning," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015. [Online]. Available: https://www.ndss-symposium.org/ndss2015/upgrading-https-mid-air-empirical-study-strict-transport-security-and-key-pinning

[23] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, "CSP is dead, long live csp! on the insecurity of whitelists and the future of content security policy," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 1376–1387. [Online]. Available: https://doi.org/10.1145/2976749.2978363

[24] S. Calzavara, H. Jonker, B. Krumnow, and A. Rabitti, "Measuring web session security at scale," *Comput. Secur.*, vol. 111, p. 102472, 2021. [Online]. Available: https://doi.org/10.1016/j.cose.2021.102472

[25] CodeQL, "CodeQL built-in query for CSRF protection weakened or disabled," https://codeql.github.com/codeql-query-help/python/py-csrf-protection-disabled/, 2025.

[26] Django Software Foundation, "Django Secret Key," https://docs.djangoproject.com/en/5.0/ref/settings/#secret-key, 2005.

[27] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, P. Ning, P. F. Syverson, and S. Jha, Eds. ACM, 2008, pp. 75–88. [Online]. Available: https://doi.org/10.1145/1455770.1455782

[28] S. Calzavara, M. Conti, R. Focardi, A. Rabitti, and G. Tolomei, "Mitch: A machine learning approach to the black-box detection of CSRF vulnerabilities," in *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*. IEEE, 2019, pp. 528–543. [Online]. Available: https://doi.org/10.1109/EuroSP.2019.00045

[29] P. G. Kelley, S. Komanduri, M. L. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, and J. C. López, "Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms," in *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*. IEEE Computer Society, 2012, pp. 523–537. [Online]. Available: https://doi.org/10.1109/SP.2012.38

[30] S. Alroomi and F. Li, "Measuring website password creation policies at scale," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, Eds. ACM, 2023, pp. 3108–3122. [Online]. Available: https://doi.org/10.1145/3576915.3623156

[31] NIST, "Digital Identity Guidelines," https://pages.nist.gov/800-63-3/sp800-63b.html, 2023.

[32] OWASP, "Password Storage - OWASP Cheat Sheet Series," cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html, 2025.

[33] Werkzeug, "Werkzeug," https://werkzeug.palletsprojects.com/en/3.0.x/, 2007.

[34] ——, "Werkzeug changed default password hasher," https://werkzeug.palletsprojects.com/en/3.0.x/changes/#version-3-0-0, 2023.

[35] ——, "Werkzeug default password hasher settings," https://werkzeug.palletsprojects.com/en/3.0.x/utils/#werkzeug.security.generate_password_hash, 2023.

[36] V. Cosentino, J. L. C. Izquierdo, and J. Cabot, "Findings from github: methods, datasets and limitations," in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, M. Kim, R. Robbes, and C. Bird, Eds. ACM, 2016, pp. 137–141. [Online]. Available: https://doi.org/10.1145/2901739.2901776

[37] G. Gousios and D. Spinellis, "Ghtorrent: Github's data from a firehose," in *9th IEEE Working Conference of Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland*, M. Lanza, M. D. Penta, and T. Xie, Eds. IEEE Computer Society, 2012, pp. 12–21. [Online]. Available: https://doi.org/10.1109/MSR.2012.6224294

[38] Ilya Grigorik, "GH Archive," https://www.gharchive.org/, 2022.

[39] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: a language and infrastructure for analyzing ultra-large-scale software repositories," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. IEEE Computer Society, 2013, pp. 422–431. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606588

[40] D. Spadini, M. F. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, G. T. Leavens, A. Garcia, and C. S. Pasareanu, Eds. ACM, 2018, pp. 908–911. [Online]. Available: https://doi.org/10.1145/3236024.3264598

[41] Y. Zhang, D. Lo, P. S. Kochhar, X. Xia, Q. Li, and J. Sun, "Detecting similar repositories on github," in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, M. Pinzger, G. Bavota, and A. Marcus, Eds. IEEE Computer Society, 2017, pp. 13–23. [Online]. Available: https://doi.org/10.1109/SANER.2017.7884605

[42] Simon Koch, David Klein, and Martin Johns, "The Fault in Our Stars: An Analysis of GitHub Stars as an Importance Metric for Web Source Code," in *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb) 2024*, 2024.

[43] E. Wittern, A. Cha, J. C. Davis, G. Baudart, and L. Mandel, "An Empirical Study of GraphQL Schemas," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science, S. Yangui, I. Bouassida Rodriguez, K. Drira, and Z. Tari, Eds. Springer International Publishing, 2019, pp. 3–19.

[44] X. Li and Y. Xue, "A survey on server-side approaches to securing web applications," *ACM Comput. Surv.*, vol. 46, no. 4, pp. 54:1–54:29, 2014. [Online]. Available: https://doi.org/10.1145/2541315

[45] A. Czeskis, A. Moshchuk, T. Kohno, and H. J. Wang, "Lightweight server support for browser-based CSRF protection," in *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013*, D. Schwabe, V. A. F. Almeida, H. Glaser, R. Baeza-Yates, and S. B. Moon, Eds. International World Wide Web Conferences Steering Committee / ACM, 2013, pp. 273–284. [Online]. Available: https://doi.org/10.1145/2488388.2488413

[46] P. D. Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen, "Csfire: Transparent client-side mitigation of malicious cross-domain requests," in *Engineering Secure Software and Systems, Second International Symposium, ESSoS 2010, Pisa, Italy, February 3-4, 2010. Proceedings*, ser. Lecture Notes in Computer Science, F. Massacci, D. S. Wallach, and N. Zannone, Eds., vol. 5965. Springer, 2010, pp. 18–34. [Online]. Available: https://doi.org/10.1007/978-3-642-11747-3_2

[47] Soheil Khodayari and Giancarlo Pellegrino, "The state of the samesite: Studying the usage, effectiveness, and adequacy of samesite cookies," in *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, pp. 1590–1607. [Online]. Available: https://doi.org/10.1109/SP46214.2022.9833637

[48] NLTK Team, "Natural Language Toolkit," https://www.nltk.org/, 2024.

[49] Luxembourg House of Cybersecurity, "MOSP," https://github.com/NC3-LU/MOSP, 2024.

[50] Obico Team, "Obico Server," https://github.com/TheSpaghettiDetective/obico-server, 2025.

[51] Samuel Clay, "NewsBlur," https://github.com/samuelclay/NewsBlur, 2025.

[52] Cédric Bonhomme, "Freshermeat," https://github.com/cedricbonhomme/freshermeat, 2025.

[53] princenyeche, "BOP," https://github.com/princenyeche/BOP, 2024.

[54] S. Bozzolan, S. Calzavara, F. Hantke, and B. Stock, "Obico Server GitHub issue," https://github.com/TheSpaghettiDetective/obico-server/issues/990, 2024.

[55] ——, "Freshermeat GitHub Issue," https://github.com/cedricbonhomme/freshermeat/issues/48, 2024.

[56] ——, "MOSP GitHub Issue," https://github.com/NC3-LU/MOSP/issues/79, 2024.

Appendix

A. Post-Processing Details

We randomly sampled 100 repositories and manually labeled them as web applications or not. Two independent researchers performed the labeling and resolved disagreements by consensus, identifying only 42 web applications, highlighting the initial dataset's low quality. To reduce false positives and improve representativeness, we evaluated two approaches:

- NLP with filtering: we translate repository descriptions and metadata into English, tokenize them with NLTK [48], and apply manually curated allowlists and blocklists to identify web applications. These lists, available online [10], were built through manual analysis of previously studied repositories.
- GPT-based labeling: we instruct ChatGPT about our definition of a web application, and ask it to classify repositories based on their README files, using the following prompt: "You are my web application checker. A web application is something one could host on their server. It is not the application framework itself, not a library, not a CTF challenge, not a tutorial code and not a cheatsheet. You are given a README file. Return a JSON containing the answer yes if it belongs to a web application or no if not. The JSON must also contain a textual justification of your answer".

In both cases, if the README file and the repository's textual description are not present, then we assign that repository to the negative class.

We evaluated both approaches on a manually curated ground truth of 100 repositories, randomly sampled outside the NLP model's training set. The confusion matrices are shown in Table II and Table III. The NLP-based method misses more web applications (27) but achieves higher precision (77% vs. 66%) with fewer false positives (10 vs. 30). ChatGPT, instead, tends to overpredict the positive class, sometimes failing to classify repositories due to input limits, hence entries in the table do not add up to 100. In addition, it requires costly API access for large-scale labeling. Given our goal of building a reliable dataset with minimal false positives, we opted for the NLP-based approach.

B. Case Studies

1) Cryptographic Keys: MOSP [49] is a Flask-based platform for creating, editing and sharing validated JSON objects of any type. The application has 74 stars on GitHub, four contributors and is actively maintained. MOSP can be self-hosted or deployed on Heroku, includes documentation on how to install it and runs with a hard-coded secret key. We verified that this hard-coded secret key allows attackers to

TABLE II: Confusion matrix of the NLP approach.

		Predicted Class	
		Positive	Negative
Actual Class	Positive	33	27
	Negative	10	30

TABLE III: Confusion matrix of ChatGPT.

		Predicted Class	
		Positive	Negative
Actual Class	Positive	57	3
	Negative	30	8

forge valid user sessions in our local instance. Notably, the MOSP documentation does not contain any information on how to update the secret key, nor warn the users of potential security risks, irrespective of the chosen deployment option. Hence, users of MOSP will not be aware of the severe security issues associated with secret key reuse, unless they are Flask developers themselves and check the source code of the web application.

Obico Server [50] is a community-driven 3D printing platform developed with Django. It has 1.4k stars on GitHub, more than 70 contributors, and is actively maintained and hosted online. If no `DJANGO_SECRET_KEY` environment variable is set, the server falls back to the default secret key, which is hard-coded in the `settings.py` file. The documentation mentions changing the secret key via the environment variable, yet the step is optional and only hinted at on the configurations page, not the installation guide itself. Based on a manual analysis of Obico, we identified several points in the source code where the secret key is used, including the generation of hashes to protect uploaded media files. This hash, calculated from the file's URL path and the secret key, is used as an access control mechanism for the uploaded media item by validating the hash for every item. However, if an attacker were able to obtain the secret key, they could generate hashes granting access to media files that were supposed to be private.

2) Cross-Site Request Forgery (CSRF): NewsBlur [51] is a news reader with social elements implemented in Django. It has 6.8k stars on GitHub, 90 collaborators and is actively maintained. NewsBlur deactivates Django's `CsrfViewMiddleware` component and resorts to CSRF protection on individual forms. Our analysis identified a few sensitive views requiring authentication and writing into the database without being protected against CSRF. For example, the `add_site_authed` view subscribes the user to a given URL. By mounting a CSRF attack against this view, the attacker may force a target user of NewsBlur to automatically subscribe to a feed chosen by the attacker. The attack works despite the adoption of lax same-site cookies in Django, because it is implemented on top of a GET request, hence protection is bypassed upon form submission.

Freshermeat [52] is an open-source software directory and release tracker written in Flask. The application uses the `FlaskForm` class to protect specific functionality against CSRF, but does not implement protection on all the sensitive views. For example, the `delete_user` view is extremely dangerous, because it allows users with admin privileges to delete any user of the application, yet it is not protected against CSRF. The reason for this lack of protection is that the HTTP request asking for account deletion is not triggered by a form submission, but it is a simple GET request where the user to be deleted is passed in the query string.

As another example, BOP [53] is a bulk operation app for Jira written in Flask and with more than 1,700 installs according to the Atlassian marketplace. It suffers from a similar security issue as the previous application. Indeed, we verified in our own Jira instance that the sensitive view `account_delete` is not protected against CSRF. Triggering this functionality requires a POST request, but BOP does not activate same-site cookies, meaning that the CSRF attack would work on any browser without automated CSRF mitigation such as Firefox.

C. Responsible Disclosure

Our analysis mostly identifies potentially insecure programming practices that might enable concrete and dangerous attacks, but require additional investigation to be confirmed as vulnerabilities. For example, a hard-coded secret key may be updated by site operators before installation and sensitive views lacking CSRF protection might actually be intended due to specific use cases of the web applications, such as the implementation of REST APIs. The examples reported in Appendix B show that exploitation is certainly possible, but understanding the actual security implications of our findings would require a careful case-by-case analysis, which would be difficult to carry out at scale.

In the end, we decided to notify developers of our findings when we explicitly mentioned their applications in the paper, and for all the cases of hard-coded secret keys in Flask applications, as these can easily lead to impersonation attacks. Below is a summary of the responses we have received from the developers at the time of writing, along with references to the corresponding GitHub issues, where available:

- Obico Server [50], [54]: Upon reporting this issue, developers admitted the problem but argued their tool is a local-only server, which means the user is responsible for ensuring security on their own.
- Freshermeat [52], [55]: The developers of this application admitted the issue stating that they relied on the default Flask-WTF security against CSRF (which did not help in this case). They plan to adjust the code according to our recommendations and extend the patch to other projects they develop.
- BOP [53]: For this application as well, the developers acknowledged the issue upon reporting. However, they stated this issue only occurs in the free version of their application, and they have decided to accept the risk.
- For the reported hard-coded secret keys, we also saw a number of positive comments and fixes in the code [56].