Università
Ca'Foscari
Venezia

Master's Degree programme
in **Computer Science**

Final Thesis

A Whitebox Analysis of Session Management and Account Creation in Web Applications

Supervisor
Prof. Stefano Calzavara

Graduand
Simone Bozzolan
Matriculation number 878352

Academic Year
2023/2024

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. In this thesis, we go through the process of creating a representative dataset of web application code 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, such as password hashing and cryptographic key management. Our research analyzes more than 1,200 web applications built using the Django and Flask web development frameworks. Based on our dataset, we can see how different design choices of these frameworks affect the security features implemented by developers, such as CSRF protection, which is activated by default in 58% Django applications but only in 6% Flask applications. Our work unveils a number of new insights on web session security that prior work based on black-box testing was unable to cover.

Contents

1 Introduction **7**

2 Background **9**
2.1 Web Applications . . . 9
2.1.1 Web Application Structure and Architecture . . . 9
2.1.2 Web Application Frameworks . . . 10
2.1.3 Session Management . . . 11
2.2 Web Application Security . . . 12
2.2.1 Cross-Site Request Forgery (CSRF) . . . 12
2.2.2 Password Hashing . . . 13
2.3 Dynamic and Static Analysis . . . 14
2.3.1 Dynamic Analysis . . . 14
2.3.2 Static Analysis . . . 14
2.3.3 CodeQL . . . 15

3 Dataset Construction **17**
3.1 Initial Dataset . . . 17
3.2 Dataset Post-Processing . . . 18
3.3 Final Dataset . . . 20

4 Security Analysis **21**
4.1 Scope . . . 21
4.2 Methodology . . . 21

5 Session Management **23**
5.1 Cryptographic Keys . . . 23
5.1.1 CodeQL Queries. . . . 23
5.1.2 Analysis Results. . . . 23
5.2 Cross-Site Request Forgery (CSRF) . . . 24
5.2.1 CodeQL Queries. . . . 24
5.2.2 Analysis Results. . . . 24
5.3 Session Protection . . . 26
5.3.1 CodeQL Queries. . . . 26
5.3.2 Analysis Results. . . . 27

6 Account Creation **29**
6.1 Password Policies . . . 29
6.1.1 CodeQL Queries. . . . 29
6.1.2 Analysis Results. . . . 30
6.2 Password Hashing . . . 31
6.2.1 CodeQL Queries. . . . 31
6.2.2 Analysis Results. . . . 31

7 Discussion **33**
7.1 Methodological Take-Away Messages . . . 33
7.2 Security Take-Away Messages . . . 33
7.3 Limitations . . . 34
7.4 Research Ethics . . . 34

8 Related Work **37**
8.1 Dataset Construction 37
8.2 Web Session Security 37

9 Conclusion **39**

A CodeQL Queries **45**

B Manual Analysis **89**

Chapter 1

Introduction

Web applications routinely rely on client authentication to restrict access to personal data, like profile pages, and sensitive functionality, such as premium services gated by a paywall. Since web applications rely on the HTTP protocol which is stateless by design, they have to implement client authentication by leveraging the *session* abstractions available in popular web development frameworks and libraries. The key idea is simple: when users authenticate with their username and password, web applications set a cookie that identifies them in their web browser and use it to reconstruct state information when processing future HTTP requests. Despite their apparent simplicity, session implementations can suffer from a range of security threats [20]. Prior research investigated the prevalence of a number of vulnerabilities, like session hijacking [25], session fixation [32], and cross-site request forgery [47]. Detection of session vulnerabilities is normally performed using black-box testing techniques [22] and even the OWASP Testing Guide has a dedicated chapter on session management testing [12]. With black-box testing we refer to the method of assessing the security of a system or application from an external perspective, without detailed knowledge of its internal structure, architecture, or source code.

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 because important security checks are often performed by server-side logic. For example, password hashing is a vital component for the security of account creation but can only be assessed with a server-side view of the web application logic. As another example, sessions are often protected by cryptographic keys, e.g., used to sign session cookies, but the security of key management cannot be analyzed without having access to the web application backend. However, large-scale analysis of server-side logic is challenging for many reasons [31] as this part of an application lies behind the curtain.

On the other hand, white-box testing assesses the security of a system by looking at its internal structure and architecture. In our case, this entails having access to the source code that is being run on the servers. This enables the examination of vulnerabilities that remain undetectable through black-box testing. Moreover, having access to the source code obviates the necessity for reliance solely on dynamic analysis, as static analysis can be employed. This approach offers some clear advantages, the main one being that we do not have to install and execute the applications in order to analyze them, thus improving the scale of our study. Nevertheless, as we shall discuss subsequently, scaling up our analysis posed additional challenges.

In this thesis, we take on this challenge and 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 for the first time a number of security practices that cannot be assessed through black-box testing but that can be fruitfully checked by static analysis. 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 repositories 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, which we make publicly available to facilitate

future research [1].

After solving the main challenges associated with dataset construction, we use the CodeQL analyzer [29] from GitHub and write analysis queries to capture key aspects of session security measured on the server side. Using CodeQL, we analyze the security posture of more than 1,200 web applications developed using the Django [4] and Flask [5] frameworks for the Python programming language, uncovering previously unrecognized insights on web session security that previous black-box testing efforts overlooked.

To summarize, we here make the following contributions:

- We propose a methodology to assess server-side security issues by first constructing a large-scale dataset of relevant applications (Section 3) and then analyzing it with CodeQL (Section 4). For the research community to leverage this knowledge and enable more research on server-side issues, we make our code open-source [1].
- We are the first to report on session management security on the server side, investigating the management of cryptographic keys, CSRF protection, and custom practices for improved protection against session hijacking (Section 5).
- Additionally, we present insights into server-side implementations of the account creation process, with a specific focus on password policies and password hashing (Section 6).

Chapter 2

Background

To follow along with the thesis, we will give some background information on web applications, session management, web application security and GitHub's CodeQL static analysis engine in the following section.

2.1 Web Applications

A general overview of web applications follows.

2.1.1 Web Application Structure and Architecture

A web application is a software application that is accessed and operated through a web browser over a network, typically the internet. Unlike traditional desktop applications that are installed locally on a user's computer, web applications reside on remote servers and are accessed through URLs or hyperlinks.

Web applications encompass a wide range of functionalities and can serve various purposes, from simple websites with static content to complex platforms offering dynamic and interactive features. They can include e-commerce sites, social media platforms, online banking systems, email services, and productivity tools, among others.

At its core, a web application consists of two main components: the client-side (front-end) and the server-side (back-end). The client-side refers to the user interface and functionality that is executed on the user's web browser, typically using technologies such as HTML, CSS, and JavaScript. The server-side, on the other hand, handles the processing of requests, database operations, and other backend functionalities using server-side programming languages like Python and Java. The interaction between the client-side and server-side components enables the web application to deliver content and services to users seamlessly. The two components usually communicate over the internet using the HTTP protocol.

Typically, the structure of a web application can be abstracted in three layers, as shown in Figure 2.1. These layers are often referred to as the presentation layer, the business logic layer, and the data access layer:

- Presentation Layer (or User Interface Layer): This layer is responsible for presenting information to the user and handling user interactions. It typically consists of the user interface components, such as web pages, forms, buttons, and other elements that users interact with. Technologies like HTML, CSS, and JavaScript are commonly used to create the user interface and implement client-side interactivity. The presentation layer communicates with the business logic layer to retrieve and display data, as well as to handle user input and trigger actions.

- Business Logic Layer (or Application Layer): The business logic layer contains the core functionality and rules of the application. It encapsulates the logic that processes data, performs calculations and orchestrates interactions between different components of the application. This layer is independent of the user interface and the data storage mechanisms, making it reusable and easier to maintain. It typically implements functionalities such as user authentication, authorization, validation, and application-specific workflows. Server-side programming languages like Python and Java are commonly used to implement the business logic layer.

- Data Access Layer (or Persistence Layer): The data access layer is responsible for interacting with the data storage systems, such as databases or external APIs, to retrieve, store, and manipulate data. It abstracts the underlying data storage details and provides a unified interface for the business logic layer to access and manage data. This layer handles tasks such as querying databases, executing CRUD (Create, Read, Update, Delete) operations, and ensuring data integrity and security. Object-relational mapping (ORM) frameworks or libraries are often used to simplify database interactions and manage data models in an object-oriented manner.

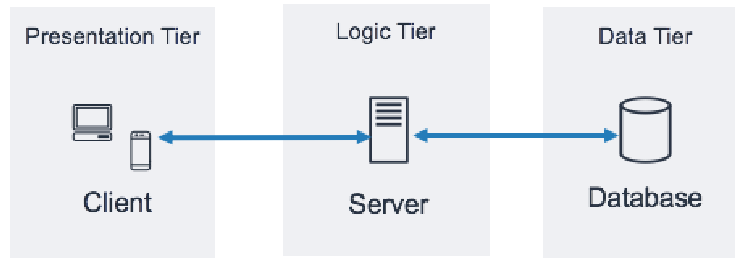


Figure 2.1: Typical layered structure of a web application. (url)

2.1.2 Web Application Frameworks

A web application framework is a software framework designed to aid developers in the development of web applications by providing a structure, set of libraries, and tools that simplify common tasks and promote best practices. These frameworks aim to streamline the process of building web applications by offering pre-built components and patterns for handling various aspects of web development. They handle common tasks such as routing URLs, rendering HTML templates, managing sessions, handling user authentication, and interacting with databases. By providing these features and tools, web application frameworks enable developers to focus on implementing business logic, rather than reinventing the wheel for common web development tasks. They promote code organization, maintainability, scalability, and security, making the development process more efficient and less error-prone. Two popular Python web frameworks are Flask [5] and Django [4], each with its own strengths and purposes.

Flask is a lightweight and flexible micro-framework. It is designed to be simple and easy to use, making it a great choice for small to medium-sized projects or when more control over the application's architecture is needed. It does not come with built-in features for database integration, authentication, or form validation, but it allows the developer to choose his own extensions or libraries for these functionalities. This gives developers more flexibility in customizing their application stack. Flask is often favored for rapid prototyping or projects where simplicity and minimalism are priorities.

On the other hand, Django is a high-level framework that comes with a wide range of built-in features and components, making it suitable for building complex, feature-rich web applications quickly. It follows the "batteries-included" philosophy, providing built-in support for database ORM (Object-Relational Mapping), user authentication, form handling, admin interface, caching, internationalization, and more. Django follows the "Convention over Configuration" principle, meaning it comes with a set of predefined conventions and best practices that help developers write clean, maintainable code. This can be beneficial for large teams or projects where consistency is crucial.

In summary, Flask and Django cater to different needs and preferences. Flask offers simplicity and flexibility making it ideal for small to medium-sized projects and developers who prefer a minimalist approach. Django, instead, provides a comprehensive set of built-in features, conventions, and tools, making it well-suited for large-scale projects and teams who do not want to delve too deeply into configuration settings and libraries.

2.1.3 Session Management

The presentation (client) and logic (server) layers usually communicate over the HTTP protocol. The HTTP protocol is based on a request-response paradigm involving a client (normally, a web browser) and a server. Since HTTP is stateless by design, the server relies on client-side state information, e.g., in the form of *cookies*, to keep track of previous interactions and build a stateful *session* abstraction, thus enabling the persistence of data across multiple HTTP requests. This is the most widespread technique for implementing client authentication on the Web.

Figure 2.2 shows the general workflow of cookie-based client authentication. 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 into the client, thus establishing a session. Later on, the 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 with the web application. Implementation details may vary, but sessions can be broadly classified into two categories:

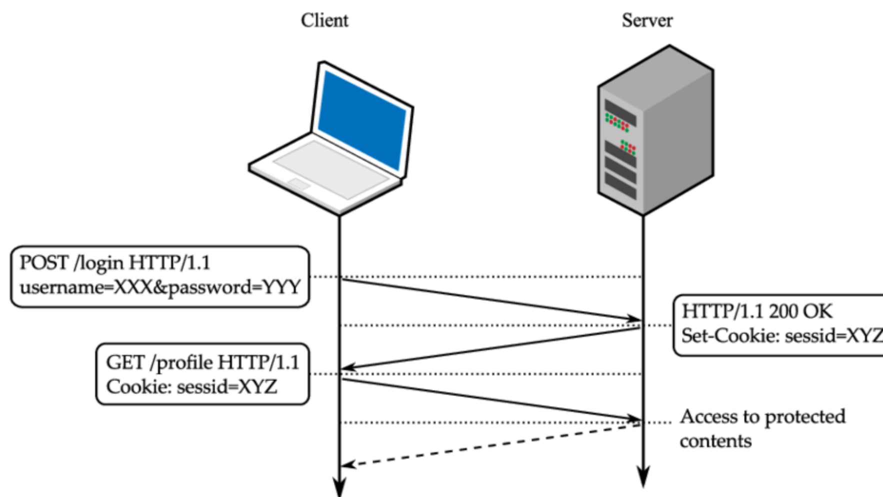


Figure 2.2: Cookie-based User Authentication. (url)

- Server-side Sessions: Server-side sessions involve storing session data on the server. When a user accesses a web application, the server generates a unique session identifier (usually a session ID) and associates it with a session data store, such as in-memory storage, a database, or a distributed cache. This session ID is then sent to the client, typically as a cookie. When the client makes subsequent requests to the server, it includes the session ID, allowing the server to retrieve the associated session data. The server can then use this data to maintain user-specific information, such as login credentials, shopping cart contents, or user preferences, throughout the user's interaction with the application.

- Client-side Sessions: Unlike server-side sessions, client-side sessions do not require the server to store session data or manage session IDs. Instead, client-side sessions involve storing session data on the client's device, such as in the browser's memory or inside cookies. Cookies are small pieces of data sent by the server and stored on the client's device and they can be used to store session data, such as user preferences.

Due to their storage on the client-side cookies are susceptible to client-side attacks, notably tampering. Consequently, cookies are commonly cryptographically signed as a preventive measure against such attacks. This cryptographic signing serves to safeguard the integrity of cookies, ensuring their authenticity and mitigating the risk of unauthorized alterations by malicious entities.

Figure 2.3 presents the code of a tiny Flask application using the popular Flask-Login library to implement client-side sessions based on cookies signed with the secret key at line 5. The application defines two *routes*, i.e., the HTTP endpoints `/login` (lines 8-16) and `/admin` (lines 18-21). Once an HTTP request reaches a route, it activates the corresponding *view*, i.e., the functions `login` and `panel` in our case. The first route extracts authentication credentials from incoming HTTP requests: if credentials for admin access are correct, it authenticates the user by calling the

`login_user()` function of Flask-Login at line 14, which will take care of generating the signed session cookie and sending it to the client, and then it will redirect the client to the admin panel at line 15. 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 at line 19. If the client does not send a signed cookie established by a previous invocation to `login_user()`, an error page is returned.

```python
from flask import Flask, render_template, redirect, url_for, request
from flask_login import LoginManager, login_user, login_required

app = Flask(__name__)
app.secret_key = 'supersecret'
login_manager = LoginManager(app)

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        if request.form['username'] == 'admin'
                and request.form['password'] == 'secure':
            user = User(user_id='admin')
            login_user(user)
            return redirect(url_for('private_area'))
    return render_template('login.html')

@app.route('/admin')
@login_required
def panel():
    return render_template('private.html')
```

Figure 2.3: Example Flask application

2.2 Web Application Security

We will now explain the main security vulnerabilities we will later analyze in this thesis.

2.2.1 Cross-Site Request Forgery (CSRF)

A Cross-Site Request Forgery (CSRF) is a type of web security vulnerability that occurs when a malicious website tricks a user's web browser into making unintended and unauthorized requests to a different website where the user is authenticated. For most websites, browser requests automatically include any credentials associated with the site, such as the user's session cookie, IP address, Windows domain credentials, and so forth. Therefore, if the user is currently authenticated to the site, the site will have no way to distinguish between the forged request sent by the victim and a legitimate request sent by the victim. In a CSRF attack, the attacker exploits the trust that a website has in a user's browser to perform actions on the user's behalf without their consent. Figure 2.4 shows how a typical CSRF attack works. Here is an explanation of what is shown in Figure 2.4:

1. Authentication: The victim user is authenticated and logged into a trusted website, such as a banking site or a social media platform, in one browser tab or window.

2. Malicious Website: The attacker creates a malicious website or sends a malicious link to the victim. When the victim visits the malicious website or clicks the malicious link, it executes code that triggers a request to the trusted website in the background.

3. Unintended Request: The malicious request sent by the victim's browser to the trusted website contains authentication credentials (e.g. session cookies) because the victim is logged in. The trusted website, unable to distinguish between a legitimate request and a CSRF attack, processes the request as if it came from the user themselves.

4. Unauthorized Action: The trusted website performs the action requested by the malicious request, such as transferring funds, changing account settings, or posting content on behalf of the user, without the user's knowledge or consent.

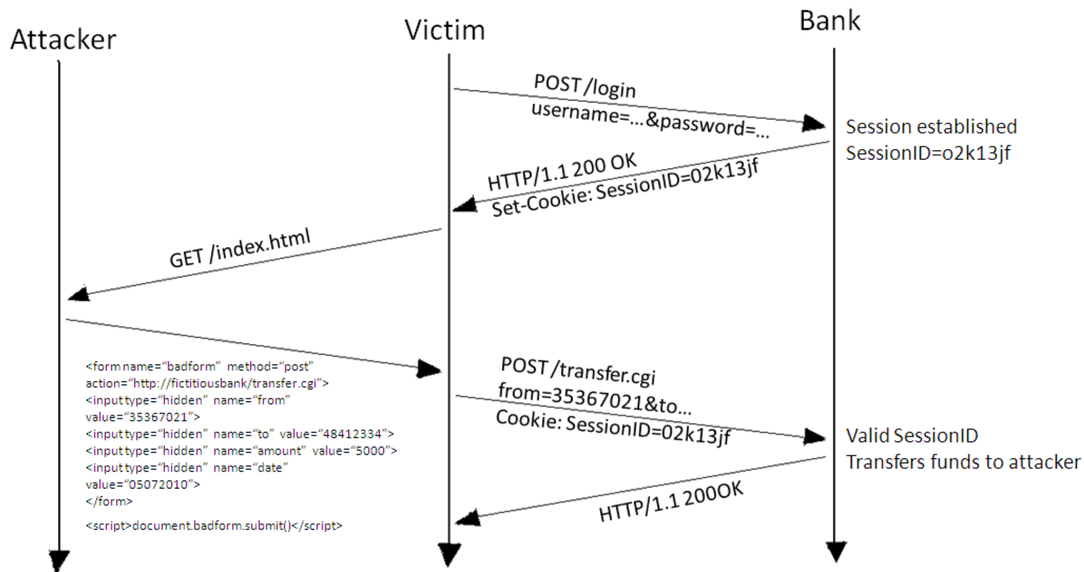


Figure 2.4: An example of a CSRF attack. (url)

CSRF attacks can have serious consequences, including financial loss, identity theft, and reputational damage. They are particularly dangerous because they exploit the inherent trust relationship between a user's browser and the websites they visit. To mitigate CSRF attacks, web developers can implement various defense mechanisms, including:

- Anti-CSRF Tokens: Including a unique token in each form or request that is tied to the user's session. This token must be submitted along with the request and validated by the server to ensure that the request originated from a legitimate source.
- SameSite Cookies: Setting the SameSite attribute on cookies to restrict their scope and prevent them from being sent in cross-site requests. Cookies marked as `SameSite=Strict` or `SameSite=Lax` are not sent in cross-origin requests, thereby mitigating CSRF attacks.
- Anti-CSRF Headers: Implementing server-side protections, such as checking the Origin or Referer headers of incoming requests to verify that they originated from the same domain as the website.

By employing these defense mechanisms and following secure coding practices, web developers can effectively protect their applications against CSRF attacks and safeguard the integrity and confidentiality of user data.

2.2.2 Password Hashing

Password hashing is a cryptographic technique used to securely store user passwords in a hashed form within a database. When a user creates an account or changes their password, the plaintext password provided by the user is transformed into a fixed-length string of characters, known as a hash value, using a hashing algorithm. This hash value is then stored in the database instead of the plaintext password.

The hashing algorithm makes use of a hashing function, that is a function which takes an input (or "message") and produces a fixed-size string of characters as output, known as the hash value. Here are some key characteristics and properties of hashing functions:

- Fixed Output Size: Hashing functions generate hash values of a fixed length, regardless of the length or complexity of the input data. For example, the SHA-256 hashing algorithm always produces a 256-bit (32-byte) hash value.

- One-Way Operation: Hashing functions are designed to be irreversible, meaning that it should be computationally infeasible to reverse-engineer the original input data from its hash value. This property ensures data integrity and confidentiality, as hashed data cannot be easily decrypted to obtain the original plaintext.
- Avalanche Effect: A small change in the input data should result in a significantly different hash value. This property ensures that even minor alterations to the input produce vastly different hash values, making it difficult for attackers to predict or manipulate hash values.

The primary reason for using password hashing is confidentiality: storing plaintext passwords in a database poses a significant security risk because if the database is compromised, all user passwords can be exposed. Hashing passwords ensures that even if the database is breached, attackers cannot retrieve the original passwords directly from the stored hashes. This is because hashes are irreversible, therefore they cannot be easily decrypted to obtain the original password.

It is important to note that not all hashing algorithms are created equal in terms of security. Strong cryptographic hash functions, such as scrypt, bcrypt, Argon2, and PBKDF2, are recommended for password hashing due to their resistance to brute-force attacks and other cryptographic vulnerabilities. An offline brute-force attack is when attackers try every possible combination of passwords or encryption keys offline, using a copy of encrypted data they have obtained. They automate this process until they find the correct password or key to gain unauthorized access. In order to protect against this kind of attack, slow cryptographic hash functions such as the ones mentioned above are recommended, since they make hashing computationally intensive. This means that it becomes computationally infeasible for an attacker to brute-force the hashes by trying all possible combinations.

In summary, password hashing is a fundamental security measure used to protect user passwords and ensure the confidentiality and integrity of user accounts in web applications and other systems. By storing only hashed passwords instead of plaintext passwords, organizations can significantly reduce the risk of data breaches and unauthorized access to sensitive information.

2.3 Dynamic and Static Analysis

Dynamic and static analysis are two common approaches used in software testing and security assessments, each with its own methodology and objectives.

2.3.1 Dynamic Analysis

Dynamic analysis involves analyzing the behavior of the software during its execution. It focuses on observing how the program interacts with its environment, inputs, and other components while running. The primary goal of dynamic analysis is to assess the runtime behavior of the software, identify runtime errors, memory leaks, performance bottlenecks, and security vulnerabilities that may manifest during execution. Tools used for dynamic analysis interact with the software as it runs, collecting data on its behavior and performance.

Dynamic analysis provides insights into the actual behavior of the software in real-world scenarios, helping to uncover issues that may not be apparent from static analysis alone, however it does not have access to the inner workings of the software it is analyzing. Moreover, Dynamic analysis typically requires executing the software, which may not always be feasible or safe, especially in the case of potentially malicious or untrusted code. It may also overlook certain types of issues that can only be detected through static analysis, including those delineated within this thesis.

2.3.2 Static Analysis

Static analysis involves examining the code without actually executing the program. It focuses on understanding the structure, syntax and semantics of the code to identify potential issues, vulnerabilities or errors. Static analysis techniques include syntax checking, data flow analysis and control flow analysis. Automated tools, such as CodeQL [29], are often used to perform these analyses.

Static analysis can identify a wide range of potential issues without the need to execute the code, making it efficient for detecting certain types of errors such as coding mistakes, insecure coding practices and potential security vulnerabilities. However, it may produce false positives or miss certain types of issues that can only be detected during runtime. Additionally, it may not capture the behavior of the code in real-world execution environments.

2.3.3 CodeQL

In our analysis, we leverage GitHub's CodeQL [29]. CodeQL is a static code analysis engine designed to automate security checks and to identify vulnerabilities within codebases across various programming languages. To support various languages within the same tool, CodeQL translates abstract syntax trees (AST) into a graph database. The resulting database can be analyzed through CodeQL queries written in a SQL-like language. For example, researchers can leverage the underlying taint-tracking capabilities to follow the usage of specific variables. The result of a query shows a list of identified patterns that point to the belonging parts in the code so that the user can further analyze and evaluate the code snippets. Here we show the general CodeQL workflow for analyzing code-bases:

1. Database generation: using CodeQL's `database create` command a database is created using the specified code-base.

2. Query execution: using CodeQL's `query run` command the specified query, written in a `.ql` file, is executed on the previously created database.

3. Query decoding: once the query has finished execution the output will be in binary form and the user can decide how to decode it using CodeQL's `bqrs decode` command. Among the supported file formats for decoding are: text files (`.txt`) and JSON files (`.json`).

Figure 2.5 shows a simplified version of a query we used to find usages of Flask-Login's `login_user()` function. As demonstrated by the example, CodeQL is similar to SQL both syntactically and semantically. More specifically, the query selects, from all the possible data-flow nodes, those representing the `login_user` function imported from the `flask_login` module. We then filter the nodes by requiring them not to be import members and to have a control flow node associated with them. Finally, we print those nodes and their location in the source files. The result of the query is a list of Flask-Login's `login_user()` function calls and their location.

```
1 import python
2 import semmle.python.ApiGraphs
3
4 from DataFlow::Node node
5 where node = API::moduleImport("flask_login").getMember("login_user")
6             .getAValueReachableFromSource()
7     and not node.asExpr() instanceof ImportMember
8     and exists(node.asCfgNode())
9 select node, node.getLocation()
```

Figure 2.5: Example CodeQL query

Chapter 3

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 [42], widely accepted by the research community and considered a standard de facto. Unfortunately, there is no similar standard for open-source web applications. Prior work on server-side security [41, 34] evaluated applications from the Bitnami catalogue [3]. Unfortunately, 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 - a realistic evaluation would cover just a subset of an already small catalog.

As it turns out, coming up with a solid dataset for our purposes is a particularly challenging task. In this section, we discuss how we automatically create a dataset of popular open-source web applications and how we post-process it to ensure its representativeness. To support future research in the field, we release the final dataset and our dataset construction scripts to the community [1].

3.1 Initial Dataset

For our dataset and analysis, we focus on two well-known and relevant web frameworks for Python: Django [4] and Flask [5]. We focus on Python because it is one of the most popular programming languages along with JavaScript nowadays [27, 46]. We consider Django and Flask as web development frameworks because they are the most popular in the Python ecosystem according to prior work [37, 45]. 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 the server-side state by default, while Flask relies on the client-side state.

With the list of frameworks decided, we then go on to collected a set of GitHub repositories that we could analyze for session management issues. We leverage the GitHub REST API [28] for this task. The GitHub API allows us to request all the information that we can see on GitHub while enabling searches similar to the GitHub search on the web application. Users require an API key to request information and every request costs credits that are automatically refilled after an hour. The API returns a maximum of 1,000 results per request. We developed a crawler that leverages the GitHub REST API search to gather metadata about all GitHub repositories that import from the Django or Flask modules via the search terms *import <module>* and *from <module>*. To overcome the aforementioned cap of 1,000 search results that the API returns, we employed a method already used in previous work [49] which utilizes the file size filter. With this filter, the API only returns files that are exactly in the requested file size range. Hence, we use binary search to retrieve all repositories through multiple queries to the API.

In total, we identified 296,913 repositories containing uses of Django and 110,694 repositories containing uses of Flask (see Table 3.1). 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 [45]. After this step, we were left with

Table 3.1: The table lists the search terms we used to find framework uses and the number of findings.

Framework	**Filter**	**Repositories**
Django	Imports from Django	296,913
	Requires authentication	115,301
	Representative	7,898
Flask	Imports from Flask	110,694
	Requires authentication	20,007
	Representative	1,872

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:

- *Number of stars*: This metric is a standard indicator of the popularity of a repository.
- *Number of contributors*: This metric estimates both the popularity and the complexity of the application.
- *Number of commits*: This metric shows that the project has been actively developed, at least for a while.
- *Year of last commit*: This metric ensures that the project is not outdated.

Table 3.2: The quartiles for different filter options on the Django and Flask datasets.

Framework	**Filter**	**Q1**	**Q2**	**Q3**
Django	Stars	0	0	2
	Contributors	0	0	2
	Commits	5	25	98
Flask	Stars	0	0	1
	Contributors	0	0	2
	Commits	8	30	103

We decided only to 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, and we present their quartiles in Table 3.2. The table shows that the distributions of stars and contributors are highly skewed because a vast amount of repositories have no stars and no contributors besides their creator. Setting a meaningful threshold in this case is thus far from straightforward. We decided to consider as "representative" just those repositories such that all the three metrics fall in the fourth quartile (top 25%). After such filtering, we were left with 7,898 Django repositories and 1,872 Flask repositories.

3.2 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 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. Remarkably, some of these web applications are even *deliberately* vulnerable to teach web security concepts.

We then randomly sampled 100 repositories and manually labeled them all as web applications or not. In particular, we assume the following definition: *a web application is any piece of production*

software providing functionality to end users that anyone could host on their own server and reach over HTTP(S). This definition rules out applications that are not intended to be self-hosted, e.g., libraries and web frameworks, or that are not going to be deployed in production, e.g., tutorials, challenges, and academic projects. This manual process identified just 42 web applications out of 100 analyzed repositories, meaning that the quality of our initial dataset is rather low.

We then considered two approaches to reduce the number of repositories in our dataset and improve its representativeness:

- NLP with filtering: we translate to English the textual description of the repository and related metadata, we process it using the NLTK [10] library to translate it into a list of tokens and we apply manually-curated blocklists and allowlists to determine which repositories include web applications. In particular, the filtering is divided into two steps: (*i*) we apply the blocklist on the repositories' names, filtering out all repositories that include any term from our designated blocklist in their name; (*ii*) we download and translate to English the README file and Github's About section for the remaining repositories, which we then filter using our allowlist: we keep the repositories whose README or About section contains any term from our allowlist. The creation of such lists, which are available online [1] and in Table 3.3, was based on a manual investigation of the previously analyzed repositories.

- GPT-based labeling: we instruct ChatGPT about our definition of a web application, and we ask it to label the repositories in our dataset based on their README file (see Figure 3.1 for the prompt).

Table 3.3: Blocklists and Allowlists

	Blocklist	Allowlist
Single-term	["tutorial", "docs", "ctf", "test", "challenge", "demo", "example", "sample", "bootcamp", "assignment", "workshop", "homework", "course", "exercise", "hack", "vulnerable", "snippet", "internship", "programming", "flask", "book", "python", "django", "cybersecurity", "100daysofcode", "vulnerability", "vulnerabilities"]	["backend", "frontend", "fullstack", "selfhost", "ecommerce", "platform", "cms", "localhost", "bulletin", "127.0.0.1"]
Multi-term	-	[["web", "application"], ["self", "host"], ["content", "management", "system"]]

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 also contains "justification" with your justification.

Figure 3.1: Chat GPT Prompt

We compare the two approaches based on their accuracy on a manually curated ground truth of 100 new repositories, which were randomly sampled from those not considered during the design of the NLP-based approach. The confusion matrices of the two approaches are shown in Table 3.4 and Table 3.5 respectively. As we can see, the NLP-based approach misses many web applications (27), but it also suffers from a much smaller number of false positives than ChatGPT (10 vs. 30). ChatGPT indeed shows a clear bias towards the positive class, flagging most of the repositories as web applications. Interestingly, there were 2 cases where ChatGPT was unable to classify the repository due to the README file exceeding ChatGPT's input limits. While the ChatGPT-based approach would allow the detection of a larger number of web applications, it would also leave in

the dataset a number of repositories that we do not want to analyze. Moreover, ChatGPT sometimes fails to label the repositories, and labeling repositories at scale would be costly because it requires premium access to the ChatGPT APIs. Considering our goal of constructing a representative dataset of web applications, we preferred the use of the NLP-based approach, which, albeit imperfect, is very effective at removing false positives.

Table 3.4: Confusion Matrix of the NLP Approach

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

Table 3.5: Confusion Matrix of ChatGPT

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

3.3 Final Dataset

In the end, our final dataset contains a total of 4,472 repositories, including 3,514 Django repositories and 958 Flask repositories after post-processing. We make our dataset available to other researchers to support future work on server-side security analyses. We also release our GitHub crawler and all the filtering routines described in the thesis so that other researchers may reuse our code to further improve and extend the dataset, as well as to keep it up-to-date [1].

Post-processing respectively removed roughly 56% and 49% of the Django and Flask repositories initially identified on GitHub because there was insufficient evidence that they contain representative web applications to analyze. Of course, post-processing is not perfect: as Table 3.4 shows, it is certainly possible that some repositories in the dataset are still false positives. Yet, the precision of post-processing is 77%, meaning that the vast majority of the repositories in the dataset are expected to store web applications. Moreover, as Table 3.2 shows, the restriction to the fourth quartile of our metrics still leaves a lot of applications with a small number of stars and contributors. We do not perform more aggressive filtering to avoid the choice of arbitrary thresholds, and we prefer to err on the safe side by using a neutral approach based on quartiles. Additional filtering can be performed if one wants to bias their analysis towards more popular applications.

Chapter 4

Security Analysis

With the final dataset created, we now explain how we define and systematically look for web session vulnerabilities in our dataset.

4.1 Scope

Works on in-the-wild black-box testing include insufficient adoption of cookie security attributes [38, 19] and insecure configuration of HTTP headers [36, 48]. Given the readily-available nature of large numbers of live applications, such works can often focus on thousands or millions of sites. Similarly, session implementations might suffer from a number of different vulnerabilities [20]. Prior work on web session security used black-box testing to detect well-known vulnerabilities like session fixation and session hijacking [22], thus enabling large-scale measurements of web session security on live websites [25, 21]. However, all these works obstruct the researchers' view of the server code.

In contrast with such previous work, 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 new and distinctive vantage point, we primarily focus on vulnerabilities that are difficult or impossible to detect via black-box testing. For example, we analyze the security of password hashing practices and the correct management of cryptographic keys at the backend. Details of our analysis are presented in the following.

4.2 Methodology

We developed a number of CodeQL queries to detect insecure programming practices in web session implementations in our dataset. We designed queries based on an extensive analysis of the analyzed libraries and their recommended security practices, as well as existing literature on web session security. As well as presenting the queries in the thesis we also make them available online [1]. The key ideas of the queries and the main analysis results are discussed in the following sections. 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 actively restrict access to some 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 `UserCreation` form class, which may be used for account creation. For Flask applications, we leverage the observation that Flask-WTF [7] and WTForms [16] are the most popular libraries for handling forms in our dataset. 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".

For a more detailed view on the queries used to create the two subsets refer to Table A.1 of the Appendix.

Since our initial dataset contains a number of repositories with a low number of stars (see Table 3.2) and we are primarily interested in the security of popular web applications, we performed a preliminary filtering step, and we only considered repositories with at least 5 stars, i.e., the median value of stars observed in our dataset. Table 4.1 reports the number of repositories that we considered for our security analysis on session management and account creation.

Table 4.1: Number of repositories used in our security analysis

Purpose	**Django**	**Flask**	**Total**
Session management	920	353	1,273
Account creation	276	95	371

We performed our analysis on a virtualized environment running Ubuntu 20.04.6 LTS with 20 cores. Since both the graph database creation and the CodeQL queries may take a long time to run, we set a time limit for each command: 30 minutes for database creation and 20 minutes for query execution. With this configuration, we only experienced two timeouts during the execution of the queries and one timeout during database creation; hence, the time limits were appropriate for our analysis.

Chapter 5

Session Management

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

5.1 Cryptographic Keys

Secure session management often requires the use of cryptographic keys. When using client-side sessions in the style of Flask, one has to use cryptography to properly protect session cookies. Cookie content must at least be digitally signed to prevent forgeries enabling impersonation attacks. In the case of Django, cryptographic keys are used to create secure hashes and anti-CSRF tokens. This means that the disclosure of cryptographic keys might allow attackers to bypass CSRF protection and mount other attacks. The use of cryptography in web development frameworks is supported by the definition of a *secret key* in the configuration settings. Of course, this aspect is invisible to black-box testing strategies, but it can be fruitfully assessed by static analysis.

5.1.1 CodeQL Queries.

Our CodeQL queries check whether the secret key of the web application is hard-coded within its source code or not. This practice is dangerous because operators of the web application may just keep the default value of the secret key unchanged when setting it up, thus potentially making it available to attackers who access the source code over GitHub. The CodeQL query scans the following code parts:

- Django: the secret key is set in the `SECRET_KEY` variable of the `settings` module.
- Flask: the secret key is set in the `SECRET_KEY` field of the `app.config` object of Flask.

Once the CodeQL query detects that the secret key is hard-coded, i.e., it is set to a constant value, we extract its value to collect more insights on the security implications. However, many applications were setting the secret key at multiple different points in the code-base. This complexity arose due to the presence of testing and production configurations, the former usually setting the secret key to a hard-coded string, while the latter usually setting the secret key from an environment variable. Therefore, in order to avoid these kinds of false positives another query was developed that marks as false positives those applications that set the secret key to something different from a hard-coded string at least once. By then combining the results of our queries we were able to achieve notably accurate results, as is later proven by our manual analysis. For a more detailed view on the queries used to detect hard-coded secret keys refer to Table A.1 of the Appendix.

5.1.2 Analysis Results.

All the 1,273 applications in our dataset set a secret key, including 349 applications that hard-code the secret key in their source code (27%). Overall, Flask applications have fewer hard-coded secret keys in comparison to Django applications. We detected hard-coded secret keys in 58 out of 353 Flask applications (16%), while we identified hard-coded secret keys in 291 out of 920 Django applications (32%). 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 without passing any chosen secret key. Flask, in turn, leaves key creation entirely in charge of web developers.

Moreover, both Django and Flask recommend a minimum length of the secret key, with Django recommending stricter length requirements (50 characters vs. 24 characters). Looking at the hard-coded secret keys, we observed that 45 out of 353 Flask applications (13%) use a hard-coded secret key that is shorter than the recommended key length, while 73 out of 920 Django applications (8%) do not adhere to recommended practices. The lower number of uncompliant applications observed in Django can be explained by the fact that the automatic routine for secret key creation always generates keys of at least the minimum recommended length, but it is worth noticing that some developers are not leveraging this facility.

To corroborate our findings, we randomly sampled 10 Flask applications and 10 Django applications with a hard-coded secret key for manual analysis. As expected, all 20 applications were found to be true positives, meaning that they all hard-code the secret key. In particular, our analysis of Flask applications revealed that developers actually set the secret key to a short and predictable string in 9 applications with only a single application choosing a hard-coded yet long value. In 4 applications, the developers left a comment saying that the value of the secret key should be changed and remain hidden in production. As for Django applications, we identified secret keys set to long and random strings in 7 applications (with 3 choosing very short values). Notably, an automatically generated comment saying that the secret key should be changed and remain hidden was found in all the 10 applications.

Our analysis results provides more insights. The manual key creation approach embraced by Flask empirically drives developers to use hard-coded secret keys less frequently. However, when hard-coded keys are used, they usually are shorter than the recommended key length, and developers normally do not provide hints about the associated security risks. On the contrary, Django applications more frequently rely on hard-coded keys but tend to use longer ones in general and provide better advice on the importance of modifying the value of hard-coded keys. Detailed results of our manual evaluation process are shown in Table B.1 of the Appendix.

5.2 Cross-Site Request Forgery (CSRF)

CSRF is one of the most well-known web vulnerabilities. Although protection can be implemented in many different ways [18], a widespread solution is the use of secret tokens to authenticate security-sensitive requests in addition to cookies. As long as tokens are unpredictable, the attacker has no way to forge requests that get authenticated under the victim's identity.

5.2.1 CodeQL Queries.

We use CodeQL to detect insights about how developers are using tokens to protect their web applications against CSRF:

- Django: Django relies on an allowlist approach against CSRF. Each request is checked by the `CsrfViewMiddleware` component, and developers can opt out from protection on specific views with the `@csrf_exempt` decorator. Alternatively, developers can take the opposite approach and deactivate the `CsrfViewMiddleware` component, opting in for protection on specific views by means of the `@csrf_protect` decorator.

- Flask: Flask applications do not implement protection against CSRF by default. Protection can be activated on every request by creating a singleton of the `CSRFProtect` class of Flask-WTF, using decorators to denote specific views that do not require protection. Alternatively, CSRF protection can be enabled only at the level of individual forms, i.e., forms extending `FlaskForm` as is or extending the `Form` class overriding the default `Meta` subclass.

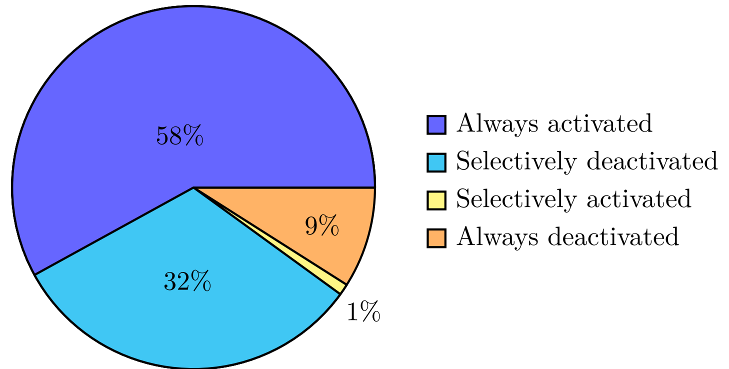
Based on the results of our queries, we are able to classify each application into one of four categories: (*i*) CSRF protection is activated by default and never deactivated; (*ii*) CSRF protection is activated by default, but deactivated on some views; (*iii*) CSRF protection is deactivated by default, but activated on some views; and (*iv*) CSRF protection is deactivated by default and never activated. For a more detailed view on the queries used to detect CSRF configurations refer to Table A.1 of the Appendix.

5.2.2 Analysis Results.

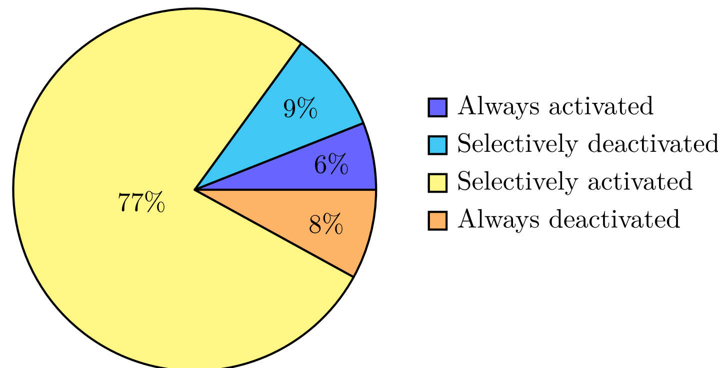
In total, we identified 1,108 applications (87%) that use the CSRF protection patterns supported by our analysis: 191 of them are in our set of Flask applications, while 917 fall in our set of Django

applications. The remaining applications (13%) do not use one of the CSRF protection patterns supported by our analysis, meaning we do not further analyze them.

Figure 5.1 shows how Django and Flask applications distribute over one of the following four security classes. Observe that the majority of Django applications (58%) mitigate CSRF by default, while the majority of Flask applications (77%) just selectively activate protection. This dichotomy could 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 by means of the `CSRFProtect` class, protection at the individual form level is more widespread in practice.



(a) Django



(b) Flask

Figure 5.1: Distribution of CSRF protection levels divided by framework

To collect additional insights and confirm results, we performed a manual investigation of a subset of our findings. First, we randomly sampled 15 applications where CSRF protection was selectively deactivated. Our findings reveal that among the predominant justifications for the deactivation of CSRF protection, there was the presence of publicly available APIs or webhooks and test views. These scenarios are typically deemed safe as CSRF protection is not necessary and, in some cases, even counterproductive. While the majority (11) of the analyzed applications deactivated CSRF protection for the above reasons, the remaining 4 were potentially vulnerable to CSRF. More specifically, the views encountered in these 4 cases were all POST requests that required authentication, such as placing an order on an e-commerce, conducting a transaction in a ledger, and submitting user data to a web application. Notably, in one of these vulnerable cases, the developer left a comment acknowledging the potential risks associated with deactivating CSRF protection for that specific view yet proceeded with the deactivation nonetheless. We are in the process of confirming and responsibly disclosing the other three detected vulnerabilities.

Transitioning over to instances where CSRF protection was selectively enabled, we proceeded to randomly sample 10 applications (5 for Flask and 5 for Django) from such set to manually inspect them. As it turns out, all of them selectively activated CSRF protection using either the `@csrf_protect` decorator of Django or extending the `FlaskForm` class of Flask-WTF. As expected, Flask

applications used the `FlaskForm` class to protect all of their forms from CSRF attacks, whereas the narrative becomes more interesting when examining Django applications. Given Django's inherent design, developers have to deactivate CSRF protection manually by removing the `CsrfViewMiddleware` in order to adopt a blocklist approach. Consequently, such an approach would only be logical if the number of views incompatible with CSRF protection, such as publicly available APIs, were predominant. Remarkably, this is not the case for 4 of the 5 analyzed Django applications. In three of these instances, the rationale behind the developer's decision to solely safeguard the most critical views, such as authentication views, remained unclear. Conversely, in the remaining case, we encountered an issue (on GitHub) raised by an operator expressing apprehension regarding the absence of adequate CSRF protection. The response given by the developer was that CSRF protection was deemed unnecessary since the application was meant to be hosted only locally. Subsequently, the developer decided to add CSRF protection exclusively to the most sensitive views, which included most authentication views. However, conspicuously absent from this implementation was the user registration view. This confirms the risks of defensive solutions based on a blocklist approach rather than on allowlists.

Finally, to examine the instances where CSRF protection was globally deactivated, we randomly sampled 10 applications. We observed that 2 of them were false positives due to the inherent shortcomings of static analysis, i.e., achieving perfect accuracy is extremely challenging due to the multitude of ways in which a specific function or class can be utilized. Conversely, the remaining 8 were true positives. Notably, none of them provided an explanation for the omission of CSRF protection, even in the case of more sensitive views.

In summary, although finding a solution devoid of trade-offs appears unattainable, Django's approach emerges as advantageous. Not only does it avoid the theoretical issues of blocklist approaches, but it also yields a higher proportion of secure applications in practice, as proven by empirical evidence. Detailed results of our manual evaluation process are shown in Table B.2 of the Appendix.

5.3 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 thesis) to determine whether a session cookie was stolen and is being reused on a different device. There are three levels of session protection available:

1. None: the client identity is never checked, i.e., no additional protection against session hijacking is in place;

2. Basic (default): the client identity is checked just for those functionalities requiring a fresh login, i.e., annotated with the `@fresh_login_required` decorator;

3. Strong: the client identity is checked for every request, i.e., requests coming from different clients than the one that established the session are rejected.

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.

5.3.1 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:

1. No protection: we check whether the application lacks enforcement of any additional protection against session hijacking. This is the case if either the `session_protection` field of the login manager is set to `None` or the `@fresh_login_required` decorator is never used;

2. Protection on some views: we check whether the application enforces additional protection against session hijacking on specific views. This happens when the `session_protection` field of the login manager is left to its default value (or set to `"basic"`) and the `@fresh_login_required` decorator is used on some views;

3. Complete protection: we check whether all the views of the application enjoy additional protection against session hijacking, meaning that the `session_protection` field of the login manager is set to `"strong"`.

As we did in section 5.1, we filtered out potential false positives. These were the cases where the `session_protection` field was set more once, likely because a testing configuration was present. As there were only a limited number of instances where this occurred, we manually analyzed all of them, confirming it was due to testing configurations. Therefore, we used the production configuration to classify them into their respective categories. For a more detailed view on the queries used to detect session protection configurations refer to Table A.1 of the Appendix.

5.3.2 Analysis Results.

In total, we have 353 Flask applications that distribute over the three security classes as follows: 325 applications (92%) do not benefit from session protection, 2 applications (1%) use session protection just on some views and 26 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.

Taking a closer look at the results, we found out that 4 applications, out of the 325 that do not use session protection, manually set the `session_protection` field to `None`. This is useless because it would suffice not to use the `@fresh_login_required` decorator. In order to try and understand why developers decided to do this, we looked at the comments in the source code, as well as GitHub's commits and issues. We found out that the developers of 2 applications (though it is very likely that one is a fork of the other) decided to manually switch off session protection because many users reported getting logged out while using the application, hence developers likely experimented with some higher level of session protection, but eventually decided to opt out. This suggests that the additional security layer provided by the session protection functionality renders some web applications unusable and is, therefore, not suitable for all scenarios, making it a security feature that is not broadly applicable.

Shifting our focus to the 2 applications that activated session protection on some views, we found out that they activated the security feature on the more sensitive parts of the application, such as the “change password” view and the views used by administrators to manage other users and permissions. This is the correct use case of session protection, i.e., enforcing additional security checks over sensitive views. Finally, we also investigated the applications that use the highest level of protection, by randomly sampling 10 applications making use of such features. After manual inspection, we observed that most of them set `session_protection` to `"strong"` without leaving any comment as to why or explaining potential issues related to it. There were, however, a couple of cases where session protection was relaxed either during testing or in certain production scenarios. Notably, some developers suggested disabling session protection when running the application behind a proxy or load balancer, since it may cause unintended issues. Detailed results of our manual evaluation process are shown in Table B.3 of the Appendix.

Chapter 6

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 available in our dataset and the password hashing techniques they adopt during the account creation phase.

6.1 Password Policies

It is well known that passwords should satisfy minimum password strength requirements to be secure against the threats of online and offline brute-force attacks [33]. 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 false negatives [17]. Unfortunately, their analysis also showed a 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* checks, which are mandatory for security and correctly enforced by definition.

6.1.1 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 4.2 and extract validators associated with their password fields. We discuss how we implemented this analysis on the different frameworks:

- Django: we infer the password policy by analyzing the content of the `AUTH_PASSWORD_VALIDATORS` variable from the `settings` module, which provides a declarative syntax to specify password requirements using a list of dictionaries. For example, it is possible to enforce a minimum length by creating a dictionary named `MinimumLengthValidator` and specifying a corresponding length. The validators are then enforced by default by the `UserCreation` form class.

- Flask: we use our heuristic approach to detect registration forms defined using either Flask-WTF or WTForms. Consequently, we detect the use of validators related to password strength, such as `Length` and `Regexp`, which are then passed to the `validators` argument of the registration form's `PasswordField`.

A relevant difference here is that Django automatically enforces a default password policy when the user registration form is submitted, while Flask-WTF does not implement any default policy and requires users to rely on explicit form validation methods. 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. For a more detailed view on the queries used to detect the password policies refer to Table A.1 of the Appendix

6.1.2 Analysis Results.

In total, we identified 371 applications, including a registration form, out of which just 251 perform some validation of its password fields (68%). We then observe that password validation is often overlooked by developers, or even deliberately deactivated when the library would have it enabled by default. Django's choice to enable password validation by default actually pays off. Indeed, we observe a greater proportion of Django applications than Flask applications performing some form of password validation (76% vs. 43%). Django applications without password validation turned off the default password policy by removing the default validators from the password field, likely for usability issues. Of course, our automated analysis of validators might suffer from imprecision because web applications are not required to use validators alone to enforce their password policies. For example, the user registration endpoint might implement custom logic to further process the password before account creation. To confirm the accuracy of our findings, we randomly sampled 10 applications among those that do not perform any password validation, and we manually inspected them to assess how common it is to perform additional checks over password fields besides validators. As it turns out, none of them conduct any additional checks, thus providing assurance of the trends observed in our analysis.

More in detail, we identified 240 applications (96%) using validators to enforce a minimum password length. Figure 6.1 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 [39]. 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. Hence, it appears that developers exhibit a propensity to relax established security recommendations, consequently resulting in less secure applications.

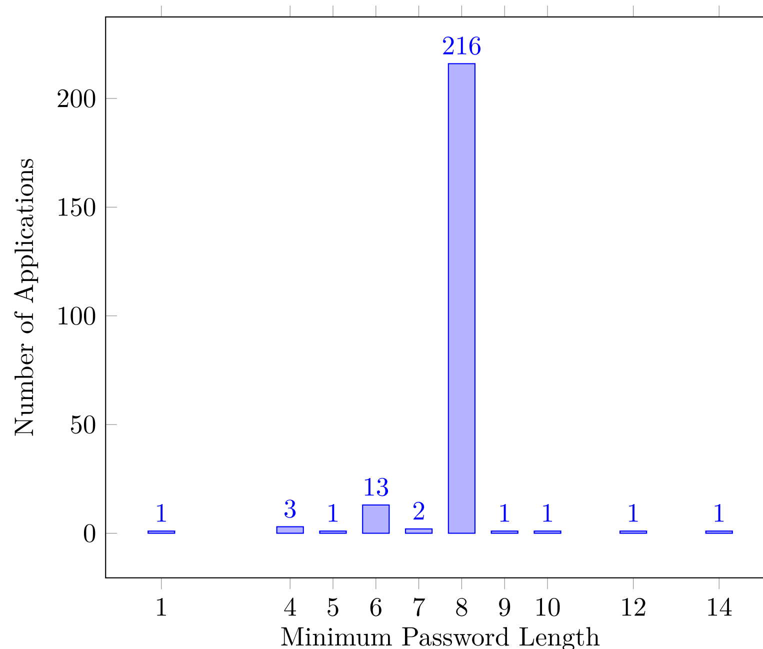


Figure 6.1: Distribution of minimum password lengths

In addition, we identified 15 applications using a custom validator. Our manual examination uncovered that all the custom validators integrated into Flask applications could have been implemented utilizing the standard built-in validators, such as the length and regular expression validators. Conversely, the custom validators adopted by Django applications could *not* have been replicated using the built-in validators, as the majority of them are either maximum length or regular expressions validations, which are not available in Django. This observation shows that, although Django introduces some applicable password validators, such as the similarity and common password validators, it needs a few validators provided by Flask-WTF, which are arguably indispensable according to real-world data.

Separating the discussion between the two frameworks, since they offer different types of validators, we found that 2 Flask applications (5%) check the password against a regular expression

and 18 (45%) enforce a maximum length. In both instances where applications incorporate password validation through regular expressions, they also integrate the minimum length validator. Upon manual examination of these two scenarios, we observed that one application employs the regular expression validator to enforce a specified degree of password complexity. Specifically, it verifies that the password includes at least one letter, one number, and one special character. Conversely, in the other case, the regular expression validator is utilized to prevent the inclusion of spaces and special characters in the password, consequently constraining the complexity of the passwords. Transitioning over to Django applications, we found that 203 applications (96%) enforce the similarity validator, 204 (97%) enforce the common password validator, 203 (96%) enforce the numeric validator and 201 (95%) combine all of the above with the minimum length validator. The widespread utilization of all validators can be attributed to Django's default behavior: all four password validators are active by default.

A relevant conclusion of our analysis is that the number of applications performing checks on *password complexity*, i.e., checking the use of specific sets of characters like numbers and symbols, is quite limited. These checks can usually be performed using regular expressions, but the number of applications using custom validators and regular expressions amounts to just 17 (7%). Note that not all applications using custom validators perform checks on password complexity. Detailed results of our manual evaluation process are shown in Table B.4 of the Appendix.

6.2 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.

6.2.1 CodeQL Queries.

We use CodeQL to verify compliance with the OWASP password hashing recommendations of April 2024 [40]. Recommended hashing algorithms include Argon2id, scrypt, PBKDF2, and bcrypt with specific configuration options. We discuss how we implemented these checks on the different analyzed frameworks:

- Django: passwords are hashed through the PBKDF2 algorithm with HMAC-SHA-256 by default. However, this behavior can be configured by setting the `PASSWORD_HASHERS` variable of the `settings` module.
- Flask: secure password storage is left to web developers in Flask-Login. Hence, we enumerated the most popular password-hashing libraries in our dataset (Table 6.1) and looked for invocations of the library's password-hashing function.

For a more detailed view on the queries used to detect the password hashing algorithms employed by the applications and their configuration refer to Table A.1 of the Appendix.

Table 6.1: Most popular hashing libraries in our Flask dataset

Library	Usages
Werkzeug [13]	65
Flask-Bcrypt [6]	17
bcrypt [2]	7
hashlib [9]	3
Passlib [11]	3

6.2.2 Analysis Results.

In total, we identified 371 applications providing a registration form, out of which 366 applications implement some form of password hashing (99%), meaning that our enumeration of popular

libraries yields almost complete coverage. The 5 applications (1%) that do not perform any password hashing according to our queries have been confirmed to follow this insecure practice after manual inspection.

Figure 6.2 shows how many applications are using a recommended hashing algorithm with a secure or an insecure configuration. We observe that the most popular hashing algorithm is PBKDF2, set in a secure configuration, followed by scrypt, set 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 [13], whose default configuration uses scrypt with an insecure configuration. Interestingly, Werkzeug recently modified its default hashing algorithm [14], transitioning from PBKDF2 to scrypt, which is widely regarded as more secure. However, even more curiously, they did not align with OWASP recommendations in the default configuration of scrypt, as by default the CPU/memory cost parameter is set to 2^{15} [15], while OWASP recommends at least 2^{17} when using Werkzeug's blocksize and degree of parallelism [40]. This weakens the recommended protection level against offline bruteforce attacks.

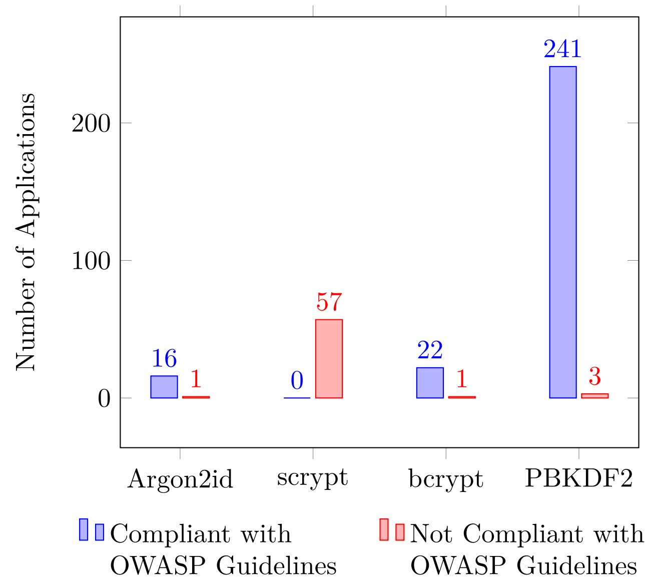


Figure 6.2: Distribution of hashing algorithms divided by compliance with the OWASP guidelines

Our analysis also found 25 applications (7%) that apparently do not use any of the four hashing algorithms covered by our queries. We manually inspected all of them, corresponding to 9 Flask applications and 16 Django applications. As it turns out, all the Flask applications were using a version of the SHA algorithm, which does not offer protection against offline bruteforce attacks. The Django applications, instead, were all false positives coming from the presence of test configurations using MD5 for performance reasons, while the production configurations were using the default PBKDF2 algorithm for password hashing. This shows again that the security-by-default approach of Django is useful in practice, because the adoption of potentially insecure hashing algorithms is confined to Flask applications. Detailed results of our manual evaluation process are shown in Table B.5 of the Appendix.

Chapter 7

Discussion

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

7.1 Methodological Take-Away Messages

From a methodological point of view, we observe that *constructing a meaningful dataset of web applications is a challenging task*. Blindly scraping GitHub based on the import of popular web development frameworks does not work, and a lot of care is needed to filter out false positives. While GPT-based filtering might offer helpful guidance in the near future, our experiments showed a significant bias towards under-filtering, proving the syntactic approach of using allow- and block-lists to be better suited for our purposes.

Luckily, once a dataset is available, *CodeQL is a reliable and effective analysis tool* for the security of session management. The number of analysis timeouts and failures was negligible in practice, allowing us to shed light on the security of session implementations in the wild. Additionally, the number of false positives reported by CodeQL turned out to be small upon manual inspection. This is enabled by our focus on syntactic code patterns, which are amenable to precise static analysis. We refer interested readers to our online repository to check details about our CodeQL queries [1].

As we have shown, with the right approach and methodology, analyzing the server-side logic of web applications at scale is indeed 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 and programming languages. Moreover, such a dataset, in combination with CodeQL, may allow one to explore not only session management but also other facets of server-side security, such as database flaws, authorization issues, and single sign-on and multi-factor authentication configurations.

7.2 Security Take-Away Messages

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 58% of the Django applications, while global protection against CSRF is enforced in just 6% of the Flask applications. Moreover, the distribution of the minimum password lengths enforced by the analyzed web applications is highly skewed towards 8, the default password length for Django applications. Also for password hashing we observe an important role of security-by-default: most of the invocations to password hashing functions are made with the default parameters set in the 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 this finding, 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.

On the negative side, though, *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 32% of the Django applications, while this practice affected just 16% 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 compared to Flask, where web developers are forced to implement password validation by themselves. In general, it seems that validation of password complexity is uncommon in our dataset, with less than 7% 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 insecure against offline bruteforce attacks by the OWASP guidelines, meaning that the Werkzeug developers failed at implementing security-by-default according to current best practices.

7.3 Limitations

The primary 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. Nevertheless, this limitation 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, carrying clear, practical implications. Extending our analysis to additional libraries and frameworks is certainly feasible with more engineering effort.

A second limitation of our analysis is its selective coverage of session security aspects. Notably, our examination does not encompass certain relevant factors, such as HTTPS adoption, proper configuration of security headers, and use of specific cookie security attributes. These exclusions are not inherent limitations; rather, they arise from practical considerations. For example, most web applications can be self-hosted either on HTTP or on HTTPS, hence they do not implement state-of-the-art countermeasures against network attackers by default. Additionally, aspects like cookie security attributes have already been extensively explored in the existing literature. While our code analysis approach could support similar investigations, we generally refrain from delving into security issues detectable through black-box testing. This choice aligns with the distinctive vantage point of our research, which grants access to the source code of open-source web applications.

Lastly, our dataset construction is limited by the capabilities of the GitHub REST API. Ideally, we would like to analyze every relevant repository that exists on GitHub, however, the vast scale of the corpus size of GitHub is too large to make every repository searchable. Knowing this limitation, GitHub recently introduced a new search engine [23], which indexes a significantly larger number of projects. Yet, as of this writing, this enhanced search engine is only available through the web search, not via the API. Consequently, the API returns fewer repositories in its search than what is displayed in a web search. Nonetheless, upon manual verification of the most relevant repositories in the web search, we confirmed their inclusion in our dataset generated via the API. This makes sense as even with their old search engine, GitHub is interested in prioritizing and representing the most important repositories to cater to users' interests. In conclusion, although our method cannot find every single project on GitHub, we are confident that our dataset represents relevant open-source projects.

7.4 Research Ethics

In this thesis, we decided to base the dataset construction on the GitHub API, adhering strictly to its best practice guidelines and rate limits. Afterward, we analyzed all projects independently on our own machines, interfering with no external entity. Regarding our findings, we evaluated the potential security impact of each case and internally discussed whether it should be disclosed or not. For example, some findings are primarily misconfigurations that, although divergent from best practices, may be considered by web developers as low-impact issues. Moreover, during our

manual investigation we found that some issues had already been acknowledged by comments in the source code or within a GitHub issue; in a few cases, the application was intentionally configured with reduced security to avoid breakage. Nevertheless, we identified a few arguably dangerous practices, such as CSRF vulnerabilities, that we are disclosing to developers. More generally, we are currently completing the design of our disclosure campaign to notify developers about the most relevant security issues. This process might also be useful to collect feedback from developers on their reasoning behind less secure coding choices, which might be insightful for the academic community.

Chapter 8

Related Work

This work is basically divided into two parts. In the first part, we created a dataset which serves as foundation for the queries we run in the second part. Consequently, we now first take a look at literature that inspired our approach by focusing on datasets, and then review related work in the web session security field.

8.1 Dataset Construction

Numerous researchers asked the question of how to create comprehensive datasets from software repositories, leading to the creation of a new research field. The importance of this topic is underscored by its own dedicated conference, the *International Conference on Mining Software Repositories*. Cosentino et al. [24] published a meta-paper at this conference 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 [30], 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 the GHArchive [8], 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 [26]. 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 reproducibility of analysis. However, for our study, we were not interested in a dataset of general applications but a sample set of very specific Flask and Django-based 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 [44], or how to find similar repositories for later analysis [50]. Koch et al. [43] analyzed various metrics of software repositories to understand the relationship between, for example, GitHub stars and download numbers, finding only a weak relationship 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 relationship, we decided to take a combination of various metrics. In terms of practical applications of the GitHub API, Wittern et al. [49] utilized the API to collect all available GraphQL schemes they could find via the API search. Similar to our approach, they used the file size as a parameter to search a larger space of repositories where they could conduct their security analysis.

8.2 Web Session Security

Prior work investigated different security threats against web session; see [20] for a survey of this important research area. The distinctive feature of this thesis compared to prior work in the field 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 password hashing and cryptographic key management, which are 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.

Here, we review relevant prior work on web session security based on black-box testing. Protection against session hijacking was primarily assessed by checking the appropriate adoption of cookie security attributes such as HttpOnly and Secure [38, 19]. CSRF protection was evaluated at scale in the wild by Sudhodanan et al., finding significant room for abuses [47]. More recently, Khodayari and Pellegrino measured the effectiveness of the SameSite cookie attribute for CSRF protection, quantifying its benefits and limitations [35]. Black-box testing strategies for secure session management were systematized and presented in [22]. Later work used similar testing strategies to perform large-scale measurements of web session security in the wild [25]. 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, as we do in this thesis.

Chapter 9

Conclusion

In this study, we focused on the server-side implementations of web session security on a large scale. To this end, we first developed a methodology scraping GitHub and filtering code repositories to create a dataset of relevant web applications and their source code. The developed tools and the dataset are open-source to aid further research and analyses [1]. Through our analyses of session management security features – like implemented secret keys, CSRF protection, and additional session protection – and account creation configurations – such as password policies and the usage of password hashing algorithms – we demonstrate the impact of framework design choices on session security. In fact, we underline that security-by-default pays off in practice, as evidenced by 58% Django applications in our dataset implementing CSRF protection across all endpoints as it is the default setting, compared to only 6% Flask applications, where developers must set their own protection. However, our analysis also highlights there remains a critical discussion on how to properly design such default security mechanisms. Among applications in our dataset that implement a secret key used to sign sessions or create secure CSRF tokens, 32% Django applications used a hard-coded secret key and pushed it to GitHub, contrasted with 16% for Flask. The difference Django sets a hard-coded key per default while for Flask the developers are required to set their own secret key.

In addition to the given analysis, we point out that both analyses require knowledge of server-side code – an area that we have not seen explored on a large scale in existing literature. We therefore propose a clear method to acquire this knowledge encouraging researchers to conduct more large-scale measurement studies on the server side in the future, for example, investigating the implementation of database security or authorization mechanisms.

There are many possible follow-ups to this work, as we only set the groundwork for research in this topic, by providing guidance on the methodology. We propose GitHub as a data source for server-side code and use filter lists to exclude deliberately vulnerable or tutorial applications, encouraging further research to build up on our decisions as there is a clear need for more sophisticated heuristics. Moreover, we only covered some of the most prominent aspects of server-side security, as there are many more left to uncover and analyze such as: the implementation of database connections and queries, authorization mechanisms, multi-factor authentication, account recovery, password change and reset. In addition, an analysis of other programming languages and frameworks could be undertaken, not only to elucidate their respective strengths and weaknesses but also to compare between them.

We have proven not only the feasibility but also the significance and compelling nature of conducting this kind of research. By deepening our understanding of such security flaws and development decisions that lead to them, we can better design secure frameworks, thereby enhancing the security of web applications going forward.

Bibliography

[1] Artifacts. https://anonymous.4open.science/r/CCS-paper-artifacts-CodeQL-DD14. Repository containing all of the artifacts related to this thesis.

[2] Bcrypt. https://github.com/pyca/bcrypt/. [Accessed 07-05-2024].

[3] Bitnami. https://bitnami.com/. [Accesses 24-04-2024].

[4] Django. https://www.djangoproject.com/. [Accessed 22-04-2024].

[5] Flask. https://flask.palletsprojects.com/. [Accessed 22-04-2024].

[6] Flask-Bcrypt. https://flask-bcrypt.readthedocs.io/en/1.0.1/. [Accessed 07-05-2024].

[7] Flask-WTF. https://flask-wtf.readthedocs.io/en/1.2.x/. [Accessed 07-05-2024].

[8] GH Archive. https://www.gharchive.org/. [Accessed 22-04-2024].

[9] hashlib. https://docs.python.org/3/library/hashlib.html. [Accessed 07-05-2024].

[10] Natural Language Toolkit. https://www.nltk.org/. [Accessed 22-04-2024].

[11] PassLib. https://passlib.readthedocs.io/en/stable/. [Accessed 07-05-2024].

[12] Session management testing. https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/06-Session_Management_Testing/README. [Accessed 23-04-2024].

[13] Werkzeug. https://werkzeug.palletsprojects.com/en/3.0.x/. [Accessed 22-04-2024].

[14] Werkzeug changed default password hasher. https://werkzeug.palletsprojects.com/en/3.0.x/changes/#version-3-0-0. [Accessed 25-04-2024].

[15] Werkzeug default password hasher settings. https://werkzeug.palletsprojects.com/en/3.0.x/utils/#werkzeug.security.generate_password_hash. [Accessed 25-04-2024].

[16] WTForms. https://wtforms.readthedocs.io/en/3.1.x/. [Accessed 07-05-2024].

[17] Alroomi, S., and Li, F. 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* (2023), W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, Eds., ACM, pp. 3108–3122.

[18] Barth, A., Jackson, C., and Mitchell, J. C. 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* (2008), P. Ning, P. F. Syverson, and S. Jha, Eds., ACM, pp. 75–88.

[19] Bugliesi, M., Calzavara, S., Focardi, R., and Khan, W. Cookiext: Patching the browser against session hijacking attacks. *J. Comput. Secur. 23*, 4 (2015), 509–537.

[20] Calzavara, S., Focardi, R., Squarcina, M., and Tempesta, M. Surviving the web: A journey into web session security. *ACM Comput. Surv. 50*, 1 (2017), 13:1–13:34.

[21] Calzavara, S., Jonker, H., Krumnow, B., and Rabitti, A. Measuring web session security at scale. *Comput. Secur. 111* (2021), 102472.

[22] Calzavara, S., Rabitti, A., Ragazzo, A., and Bugliesi, M. 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* (2019), K. Sako, S. A. Schneider, and P. Y. A. Ryan, Eds., vol. 11736 of *Lecture Notes in Computer Science*, Springer, pp. 606–624.

[23] Clem, T. The Technology behind GitHub's New Code Search. `https://github.blog/2023-02-06-the-technology-behind-githubs-new-code-search/`, Feb. 2023.

[24] Cosentino, V., Luis, J., and Cabot, J. Findings from GitHub: Methods, Datasets and Limitations. In *Proceedings of the 13th International Conference on Mining Software Repositories* (2016), ACM.

[25] Drakonakis, K., Ioannidis, S., and Polakis, J. 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* (2020), J. Ligatti, X. Ou, J. Katz, and G. Vigna, Eds., ACM, pp. 1953–1970.

[26] Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)* (2013), IEEE, pp. 422–431.

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

[28] GitHub. GitHub REST API documentation. `https://docs.github.com/en/rest`, 2023.

[29] GitHub. CodeQL. `https://codeql.github.com/`, 2024.

[30] Gousios, G., and Spinellis, D. GHTorrent: GitHub's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)* (2012), IEEE, pp. 12–21.

[31] Hantke, F., Roth, S., Mrowczynski, R., Utz, C., and Stock, B. Where are the red lines? towards ethical server-side scans in security and privacy research. In *2024 IEEE Symposium on Security and Privacy (SP)* (2024), IEEE Computer Society, pp. 103–103.

[32] Johns, M., Braun, B., Schrank, M., and Posegga, J. Reliable protection against session fixation attacks. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan, March 21 - 24, 2011* (2011), W. C. Chu, W. E. Wong, M. J. Palakal, and C. Hung, Eds., ACM, pp. 1531–1537.

[33] Kelley, P. G., Komanduri, S., Mazurek, M. L., Shay, R., Vidas, T., Bauer, L., Christin, N., Cranor, L. F., and López, J. 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* (2012), IEEE Computer Society, pp. 523–537.

[34] Khodayari, S., and Pellegrino, G. JAW: studying client-side CSRF with hybrid property graphs and declarative traversals. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021* (2021), M. D. Bailey and R. Greenstadt, Eds., USENIX Association, pp. 2525–2542.

[35] Khodayari, S., and Pellegrino, G. 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* (2022), IEEE, pp. 1590–1607.

[36] Kranch, M. J., and Bonneau, J. 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* (2015), The Internet Society.

[37] Likaj, X., Khodayari, S., and Pellegrino, G. 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* (New York, NY, USA, Oct. 2021), RAID '21, Association for Computing Machinery, pp. 370–385.

[38] Nikiforakis, N., Meert, W., Younan, Y., Johns, M., and Joosen, W. Sessionshield: Lightweight protection against session hijacking. In *Engineering Secure Software and Systems - Third International Symposium, ESSoS 2011, Madrid, Spain, February 9-10, 2011. Proceedings* (2011), Ú. Erlingsson, R. J. Wieringa, and N. Zannone, Eds., vol. 6542 of *Lecture Notes in Computer Science*, Springer, pp. 87–100.

[39] NIST. Digital Identity Guidelines. `https://pages.nist.gov/800-63-3/sp800-63b.html`. [Accessed 22-04-2024].

[40] OWASP. Password Storage - OWASP Cheat Sheet Series. `https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html`. [Accessed 16-04-2024].

[41] Pellegrino, G., Johns, M., Koch, S., Backes, M., and Rossow, C. 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* (2017), B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds., ACM, pp. 1757–1771.

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

[43] 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).

[44] Spadini, D., Aniche, M., and Bacchelli, A. Pydriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering* (2018), pp. 908–911.

[45] Squarcina, M., Adão, P., Veronese, L., and Maffei, M. Cookie Crumbles: Breaking and Fixing Web Session Integrity. In *32nd USENIX Security Symposium (USENIX Security 23)* (2023).

[46] 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.

[47] Sudhodanan, A., Carbone, R., Compagna, L., Dolgin, N., Armando, A., and Morelli, U. 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* (2017), IEEE, pp. 350–365.

[48] Weichselbaum, L., Spagnuolo, M., Lekies, S., and Janc, A. 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* (2016), E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds., ACM, pp. 1376–1387.

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

[50] Zhang, Y., Lo, D., Kochhar, P. S., Xia, X., Li, Q., and Sun, J. Detecting similar repositories on github. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2017), IEEE, pp. 13–23.

Appendix A

CodeQL Queries

Here we report the core CodeQL queries that were developed for this project (Table A.1). A library (named `CodeQL_Library`) was also developed for this project and some queries make use of said library. We decided not to include the library, as the functionality it provides is readily discernible from its function names. For those interested, the implementation details are available online [1].

Table A.1: List of CodeQL Queries for each Section of the thesis

Section	Queries
Section 4.2 (Methodology)	Figures A.1, A.2, A.3, A.4, A.5, A.6
Section 5.1 (Cryptographic Keys)	Figures A.7, A.8, A.9, A.10
Section 5.2 (CSRF)	Figures A.11, A.12, A.13, A.14, A.15, A.16, A.17
Section 5.3 (Session Protection)	Figures A.18, A.19, A.20, A.9
Section 6.1 (Password Policies)	Figures A.21, A.22, A.23, A.24, A.25, A.26, A.27, A.28, A.29
Section 6.2 (Password Hashing)	Figures A.31, A.30, A.33, A.32, A.35, A.34, A.37, A.36, A.39, A.38, A.41, A.40, A.43, A.42, A.44, A.46, A.45, A.48, A.47, A.49, A.51, A.50, A.53, A.52, A.55, A.54, A.57, A.56, A.58, A.62, A.61, A.60, A.59

```
1  import python
2  import semmle.python.ApiGraphs
3
4  from DataFlow::Node node
5  where (node = API::moduleImport("flask_login").getMember("login_user").
       getAValueReachableFromSource()
6          or node = API::moduleImport("flask_login").getMember("utils").
               getMember("login_user").getAValueReachableFromSource())
7      and not node.asExpr() instanceof ImportMember
8      and exists(node.asCfgNode())
9      and exists(node.getLocation().getFile().getRelativePath())
10 select "The flask-login library is actually used", node.getLocation()
```

Figure A.1: CodeQL query that looks for invocations of Flask-Login's `login_user()` function.

```
1 import python
2 import semmle.python.ApiGraphs
3
4 where exists(DataFlow::Node auth |
5         auth = API::moduleImport("django").getMember("contrib").getMember("
            auth").getMember("authenticate").getAValueReachableFromSource()
6         and not auth.asExpr() instanceof ImportMember
7         and exists(auth.asCfgNode())
8         and exists(auth.getLocation().getFile().getRelativePath()))
9     or exists(DataFlow::Node login |
10        login = API::moduleImport("django").getMember("contrib").getMember(
            "auth").getMember("login").getAValueReachableFromSource()
11        and not login.asExpr() instanceof ImportMember
12        and exists(login.asCfgNode())
13        and exists(login.getLocation().getFile().getRelativePath()))
14    or exists(StrConst str | str.getText() = "django.contrib.auth.urls")
15    or exists(API::moduleImport("django").getMember("contrib").getMember("
        auth").getMember("urls"))
16    or exists(DataFlow::Node views |
17        views = API::moduleImport("django").getMember("contrib").getMember(
            "auth").getMember("views").getMember("LoginView").
            getAValueReachableFromSource()
18        and not views.asExpr() instanceof ImportMember
19        and exists(views.asCfgNode())
20        and exists(views.getLocation().getFile().getRelativePath()))
21    or exists(DataFlow::Node form |
22        form = API::moduleImport("django").getMember("contrib").getMember("
            auth").getMember("forms").getMember("AuthenticationForm").
            getAValueReachableFromSource()
23        and not form.asExpr() instanceof ImportMember
24        and exists(form.asCfgNode())
25        and exists(form.getLocation().getFile().getRelativePath()))
26 select "Django authentication is actually used"
```

Figure A.2: CodeQL query that looks for usages of Django's built-in user authentication system, therefore it checks the following: whether the login or authenticate functions are called, whether Django's login view is used or whether Django's authenticate form is used.

```
1 import python
2 import semmle.python.ApiGraphs
3
4 where not exists(ControlFlowNode node |
5     (node = API::moduleImport("flask_login").getMember("login_required").
        getAValueReachableFromSource().asCfgNode()
6         or node = API::moduleImport("flask_login").getMember("current_user"
            ).getAValueReachableFromSource().asCfgNode()
7         or node = API::moduleImport("flask_login").getMember("utils").
            getMember("current_user").getAValueReachableFromSource().
            asCfgNode()
8         or node = API::moduleImport("flask_login").getMember("utils").
            getMember("login_required").getAValueReachableFromSource().
            asCfgNode())
9     and not node.isImportMember())
10 select "The application never accesses the current_user object and never
    uses the @login_required decorator"
```

Figure A.3: CodeQL query that checks whether the Flask application ever requires the user to be logged in, by using the `@login_required` decorator, or needs the user to be logged in because it accesses the `current_user` object.

```
1  import python
2  import semmle.python.ApiGraphs
3  import CodeQL_Library.DjangoSession
4
5  where not exists(ControlFlowNode node |
6      (node = API::moduleImport("django").getMember("contrib").getMember("
           auth").getMember("decorators").getMember("login_required").
           getAValueReachableFromSource().asCfgNode()
7      or node = API::moduleImport("django").getMember("contrib").getMember("
           auth").getMember("mixins").getMember("LoginRequiredMixin").
           getAValueReachableFromSource().asCfgNode()
8      or node = DjangoSession::getAUserObject())
9      and not node.isImportMember())
10 select "The application never accesses the user object and never uses the
       @login_required decorator and never uses the LoginRequiredMixin (for
       class based views)"
```

Figure A.4: CodeQL query that checks whether the Django application ever requires the user to be logged in, by using the `@login_required` decorator or the `LoginRequiredMixin` class, or needs the user to be logged in because it accesses the user object.

```
1 import python
2 import semmle.python.ApiGraphs
3
4 predicate formClass(Class cls) {
5     exists(cls.getLocation().getFile().getRelativePath())
6     and (cls.getABase().toString() = "Form"
7         or cls.getABase().toString() = "BaseForm"
8         or cls.getABase().toString() = "FlaskForm")
9 }
10
11 predicate classWithPasswordField(Class cls) {
12     exists(API::Node node |
13         (node = API::moduleImport("wtforms").getMember("PasswordField")
14             or node = API::moduleImport("flask_wtf").getMember("
                PasswordField"))
15         and cls.getAStmt().(AssignStmt).getValue().(Call).getFunc() = node.
            getAValueReachableFromSource().asExpr())
16 }
17
18 Class getSignUpFormClass() {
19     exists(Class cls, Class supercls |
20         if exists(Class superclss | superclss.getName() = cls.getABase().(
            Name).getId())
21         then supercls.getName() = cls.getABase().(Name).getId()
22             and (formClass(cls)
23                 or formClass(supercls))
24             and (classWithPasswordField(cls)
25                 or classWithPasswordField(supercls))
26             and (cls.getName().toLowerCase().matches("%registration%")
27                 or cls.getName().toLowerCase().matches("%register%")
28                 or cls.getName().toLowerCase().matches("%createaccount%")
29                 or cls.getName().toLowerCase().matches("%signup%")
30                 or cls.getName().toLowerCase().matches("%adduser%")
31                 or cls.getName().toLowerCase().matches("%useradd%")
32                 or cls.getName().toLowerCase().matches("%regform%")
33                 or cls.getName().toLowerCase().matches("%newuser%")
34                 or cls.getName().toLowerCase().matches("%userform%")
35                 or cls.getName().toLowerCase().matches("%usersform%")
36                 or cls.getName().toLowerCase().matches("%registform%"))
37             and result = cls
38         else formClass(cls)
39             and classWithPasswordField(cls)
40             and (cls.getName().toLowerCase().matches("%registration%")
41                 or cls.getName().toLowerCase().matches("%register%")
42                 or cls.getName().toLowerCase().matches("%createaccount%")
43                 or cls.getName().toLowerCase().matches("%signup%")
44                 or cls.getName().toLowerCase().matches("%adduser%")
45                 or cls.getName().toLowerCase().matches("%useradd%")
46                 or cls.getName().toLowerCase().matches("%regform%")
47                 or cls.getName().toLowerCase().matches("%newuser%")
48                 or cls.getName().toLowerCase().matches("%userform%")
49                 or cls.getName().toLowerCase().matches("%usersform%")
50                 or cls.getName().toLowerCase().matches("%registform%"))
51             and result = cls)
52 }
53
54 from Class cls
55 where cls = getSignUpFormClass()
56 select cls, cls.getLocation(), "This form has a password field and is
    probably a signup form"
```

Figure A.5: CodeQL query that looks for sign-up forms created using either Flask-WTF or WT-Forms in Flask applications. So forms that have at least a password field and whose name contains one of the specified keywords.

```
import python
import semmle.python.ApiGraphs

from DataFlow::Node form
where (form = API::moduleImport("django").getMember("contrib").getMember("
    auth").getMember("forms").getMember("UserCreationForm").
    getAValueReachableFromSource()
        or form = API::moduleImport("django").getMember("contrib").
            getMember("auth").getMember("forms").getMember("
            BaseUserCreationForm").getAValueReachableFromSource())
    and not form.asExpr() instanceof ImportMember
    and exists(form.asCfgNode())
    and exists(form.getLocation().getFile().getRelativePath())
select form, form.getLocation(), "Django's built in user creation form is
    used"
```

Figure A.6: CodeQL query that looks for sign-up forms created using Django's built-in `UserCreationForm` class.

```
import python
import CodeQL_Library.FlaskLogin

string output(Expr seckey) {
  if seckey.(StrConst).getS().length() < 24
    then result = "The secret key is a hardcoded string and it's too short"
    else result = "The secret key is a hardcoded string"
}

from Expr expr
where expr = FlaskLogin::getConfigValue("SECRET_KEY", "secret_key")
  and expr instanceof StrConst
select expr, expr.getLocation(), output(expr), expr.(StrConst).getS()
```

Figure A.7: CodeQL query that detects hard-coded secret keys in Flask and calculates their length.

```
import python
import semmle.python.dataflow.new.DataFlow
import semmle.python.dataflow.new.DataFlow2

class SecretKeyConfiguration extends DataFlow2::Configuration {
    SecretKeyConfiguration() { this = "SecretKeyConfiguration" }

    override predicate isSource(DataFlow2::Node source) {
        source.asExpr() instanceof StrConst
    }

    override predicate isSink(DataFlow2::Node sink) {
        exists(AssignStmt asgn, Name name |
            name.getId() = "SECRET_KEY"
            and asgn.getATarget() = name
            and exists(asgn.getLocation().getFile().getRelativePath())
            and asgn.getValue().getAFlowNode() = sink.asCfgNode()
        )
    }
}

string output(StrConst key) {
    // minimum length recommended by the docs is 50
    if key.getS().length() < 50
    then result = "The secret key is a hardcoded string and it's too short"
    else result = "The secret key is a hardcoded string"
}

string output2(StrConst key) {
    if key.getS().prefix(16) = "django-insecure-"
    then result = "The secret key was not freshly generated and is insecure
        (starts with django-insecure-)"
    else result = "The secret key was freshly generated (doesn't starts
        with django-insecure-)"
}

from DataFlow2::Node secsource, DataFlow2::Node key, SecretKeyConfiguration
    sconfig
where sconfig.hasFlow(secsource, key)
select key.asExpr().(StrConst).getS(), key.getLocation(), output(key.asExpr
    ()), output2(key.asExpr())
```

Figure A.8: CodeQL query that detects hard-coded secret keys in Django and calculates their length. The class `SecretKeyConfiguration` is just used to add data-flow analysis from any string constant to Django's `SECRET_KEY` configuration variable.

```
import python
import CodeQL_Library.FlaskLogin

DataFlow::Node getSessionProtectionSource() {
    exists(DataFlow::Node n |
        (n = API::moduleImport("flask_login").getMember("LoginManager").
            getReturn().getMember("session_protection").
            getAValueReachingSink()
            or n = API::moduleImport("flask_login").getMember("
                login_manager").getMember("LoginManager").getReturn().
                getMember("session_protection").getAValueReachingSink())
        and result = n)
}

string auxsk() {
    exists(Expr expr1, Expr expr2 |
        expr1 = FlaskLogin::getConfigValue("SECRET_KEY", "secret_key")
        and expr2 = FlaskLogin::getConfigValue("SECRET_KEY", "secret_key")
        and expr1 != expr2
        and exists(expr1.getLocation().getFile().getRelativePath())
        and exists(expr2.getLocation().getFile().getRelativePath())
        and expr1.getLocation().toString() != expr2.getLocation().toString
            ()
        and (not expr1 instanceof Str
            or not expr2 instanceof Str)
        and result = "un_secret_key " + expr1 + " " + expr1.getLocation() +
             " " + expr2 + " " + expr2.getLocation())
}

string auxsp() {
    exists(Expr expr1, Expr expr2 |
        expr1 = getSessionProtectionSource().asExpr()
        and expr2 = getSessionProtectionSource().asExpr()
        and expr1 != expr2
        and exists(expr1.getLocation().getFile().getRelativePath())
        and exists(expr2.getLocation().getFile().getRelativePath())
        and expr1.getLocation().toString() != expr2.getLocation().toString
            ()
        and result = "sf_session_protection sf_session_protection_strong
            uf_session_protection_basic un_session_protection_basic_is_used
            " + expr1 + " " + expr1.getLocation() + " " + expr2 + " " +
            expr2.getLocation())
}

string aux() {
    result = auxsk()
    or result = auxsp()
}

select aux()
```

Figure A.9: CodeQL query used to detect false positives in Flask, specifically in regards to hard-coded secret keys and session protection. The query returns `true` if it finds more than one point in the code-base where the specific configuration variable gets set. In case of secret keys it also requires that at least one of the detected points in the code-base sets the secret key to something different from an hard-coded string.

```
import python
import CodeQL_Library.DjangoSession

bindingset[configsetting]
AssignStmt aux(string configsetting) {
    exists(Name name, AssignStmt asgn |
        name.getId() = configsetting
        and asgn.getATarget() = name
        and result = asgn)
}

bindingset[configsetting, queryname]
string auxx(string configsetting, string queryname) {
    exists(AssignStmt asgn1, AssignStmt asgn2 |
        asgn1 = aux(configsetting)
        and asgn2 = aux(configsetting)
        and asgn1 != asgn2
        and exists(asgn1.getLocation().getFile().getRelativePath())
        and exists(asgn2.getLocation().getFile().getRelativePath())
        and asgn1.getLocation().toString() != asgn2.getLocation().toString
            ()
        and result = queryname + " " + asgn1 + " " + asgn1.getLocation() +
            " " + asgn2 + " " + asgn2.getLocation())
}

bindingset[configsetting, queryname]
string auxsk(string configsetting, string queryname) {
    exists(AssignStmt asgn1, AssignStmt asgn2 |
        asgn1 = aux(configsetting)
        and asgn2 = aux(configsetting)
        and asgn1 != asgn2
        and exists(asgn1.getLocation().getFile().getRelativePath())
        and exists(asgn2.getLocation().getFile().getRelativePath())
        and asgn1.getLocation().toString() != asgn2.getLocation().toString
            ()
        and (not asgn1.getValue() instanceof Str
            or not asgn2.getValue() instanceof Str)
        and result = queryname + " " + asgn1 + " " + asgn1.getLocation() +
            " " + asgn2 + " " + asgn2.getLocation())
}

string output() {
    result = auxsk("SECRET_KEY", "un_secret_key")
}

select output()
```

Figure A.10: CodeQL query used to detect false positives in Django, specifically in regards to hard-coded secret keys. The query returns `true` if it detects more than one point in the code-base where the `SECRET_KEY` variable gets set and it also requires at least one of these points to set the variable to something different from a hard-coded string.

```
1 import python
2 import semmle.python.ApiGraphs
3
4 from API::Node node, DataFlow::Node n
5 where (node = API::moduleImport("flask_wtf").getMember("csrf").getMember("
      CSRFProtect")
6          or node = API::moduleImport("flask_wtf").getMember("csrf").
              getMember("CSRFProtect")
7          or node = API::moduleImport("flask_wtf").getMember("csrf").
              getMember("CSRFProtect").getReturn().getMember("init_app")
8          or node = API::moduleImport("flask_wtf").getMember("csrf").
              getMember("CSRFProtect").getReturn().getMember("init_app"))
9      and (exists(node.getParameter(0).getAValueReachingSink())
10         or exists(node.getKeywordParameter("app").getAValueReachingSink()))
11     and (n = node.getAValueReachableFromSource()
12         or n = node.getAValueReachingSink())
13     and exists(n.asCfgNode())
14     and not n.asExpr() instanceof ImportMember
15 select n, n.getLocation(), "Flask-WTF csrf protection is enabled globally"
```

Figure A.11: CodeQL query that detects whether Flask-WTF CSRF protection is enabled globally, by looking for usages of Flask-WTF's `CSRFProtect` class.

```
1 import python
2 import semmle.python.ApiGraphs
3
4 from API::Node node, DataFlow::Node n
5 where (node = API::moduleImport("flask_wtf").getMember("csrf").getMember("
      CSRFProtect")
6          or node = API::moduleImport("flask_wtf").getMember("csrf").
              getMember("CSRFProtect"))
7      and (exists(node.getParameter(0).getAValueReachingSink())
8          or exists(node.getKeywordParameter("app").getAValueReachingSink())
9          or exists(node.getReturn().getMember("init_app").getParameter(0).
              getAValueReachingSink())
10         or exists(node.getReturn().getMember("init_app").
              getKeywordParameter("app").getAValueReachingSink()))
11     and n = node.getReturn().getMember("exempt").
          getAValueReachableFromSource()
12     and exists(n.asCfgNode())
13     and not n.asExpr() instanceof ImportMember
14 select n, n.getLocation(), "Flask-WTF csrf protection is disabled
      selectively using csrf exempt"
```

Figure A.12: CodeQL query that detects whether the `@csrf.exempt` decorator is used.

```
1 import python
2 import semmle.python.ApiGraphs
3
4 from DataFlow::Node node
5 where node = API::moduleImport("flask_wtf").getMember("FlaskForm").
      getAValueReachableFromSource()
6      and not node.asExpr() instanceof ImportMember
7      and exists(node.asCfgNode())
8 select node, node.getLocation(), "FlaskForm is being used, which already
      has csrf protection enabled"
```

Figure A.13: CodeQL query that checks whether the application is extending `FlaskForm` when creating forms.

```
1  import python
2  import semmle.python.ApiGraphs
3
4  predicate csrfEnabledInMetaSubclass(Class cls) {
5      exists(AssignStmt asgn, Class meta |
6          meta = cls.getAStmt().(ClassDef).getDefinedClass()
7          and meta.getName() = "Meta"
8          and asgn = meta.getAStmt().(AssignStmt)
9          and asgn.getATarget().toString() = "csrf"
10         and asgn.getValue().(ImmutableLiteral).booleanValue() = true)
11 }
12
13 predicate csrfEnabledOnTheFly(Class cls) {
14     exists(Keyword item, Call call, KeyValuePair meta |
15         call.getAFlowNode() = cls.getClassObject().getACall()
16         and ((call.getNamedArgs().getAnItem().(Keyword) = item
17                 and item.getArg() = "meta"
18                 and item.getValue().(Dict).getAnItem().(KeyValuePair) =
                        meta)
19             or call.getPositionalArg(4).(Dict).getAnItem().(KeyValuePair) =
                    meta)
20         and meta.getKey().(StrConst).getS() = "csrf"
21         and meta.getValue().(ImmutableLiteral).booleanValue() = true)
22 }
23
24 from Class cls
25 where exists(cls.getLocation().getFile().getRelativePath())
26     and (cls.getABase().toString() = "Form"
27         or cls.getABase().toString() = "BaseForm")
28     and (csrfEnabledInMetaSubclass(cls)
29         or csrfEnabledOnTheFly(cls))
30 select cls, cls.getLocation(), "This form has enabled wtforms csrf
       protection at least once"
```

Figure A.14: CodeQL query that checks whether the application is overriding the `Meta` subclass when creating forms using WTForms in order to enable CSRF protection.

```
import python
import semmle.python.ApiGraphs

class MiddlewareConfiguration extends DataFlow::Configuration {
    MiddlewareConfiguration() { this = "MiddlewareConfiguration" }

    override predicate isSource(DataFlow::Node source) {
        exists(source.getLocation().getFile().getRelativePath())
        and (source.asExpr() instanceof List
            or source.asExpr() instanceof Tuple)
    }

    override predicate isSink(DataFlow::Node sink) {
        exists(AssignStmt asgn, AugAssign augasgn, Name name |
            (name.getId() = "MIDDLEWARE"
                or name.getId() = "MIDDLEWARE_CLASSES")
            and ((asgn.getATarget() = name
                and exists(asgn.getLocation().getFile().getRelativePath())
                and asgn.getValue().getAFlowNode() = sink.asCfgNode())
            or (augasgn.getTarget() = name
                and exists(augasgn.getLocation().getFile().getRelativePath
                    ())
                and augasgn.getValue().getAFlowNode() = sink.asCfgNode()))
        )
    }
}

where not exists(DataFlow::Node source, DataFlow::Node sink,
    MiddlewareConfiguration config |
    config.hasFlow(source, sink)
    and (source.asExpr().(List).getAnElt().(StrConst).getS() = "django.
        middleware.csrf.CsrfViewMiddleware"
        or source.asExpr().(Tuple).getAnElt().(StrConst).getS() = "django.
            middleware.csrf.CsrfViewMiddleware"))
select "Global CSRF protection is disabled"
```

Figure A.15: CodeQL query that checks whether CSRF protection is disabled in a Django application. It works by verifying the absence of Django's CSRF middleware from the list of active middlewares.

```
import python
import semmle.python.ApiGraphs

from ControlFlowNode node
where node = API::moduleImport("django").getMember("views").getMember("
    decorators").getMember("csrf").getMember("csrf_exempt").
    getAValueReachableFromSource().asCfgNode()
    and not node.isImportMember()
select node, node.getLocation(), "The application is disabling csrf
    protection for certain views"
```

Figure A.16: CodeQL query that checks whether the `@csrf_exempt` decorator is used within a Django application.

```
import python
import semmle.python.ApiGraphs

from ControlFlowNode node
where node = API::moduleImport("django").getMember("views").getMember("
    decorators").getMember("csrf").getMember("csrf_protect").
    getAValueReachableFromSource().asCfgNode()
    and not node.isImportMember()
select node, node.getLocation(), "The application is enabling csrf
    protection for certain views"
```

Figure A.17: CodeQL query that checks whether the `@csrf_protect` decorator is used within a Django application.

```
import python
import semmle.python.ApiGraphs

from DataFlow::Node n
where (n = API::moduleImport("flask_login").getMember("LoginManager").
    getReturn().getMember("session_protection").getAValueReachingSink()
    or n = API::moduleImport("flask_login").getMember("login_manager").
        getMember("LoginManager").getReturn().getMember("session_protection"
        ).getAValueReachingSink())
  and n.asExpr().toString() = "None"
select n.getLocation(), "Session protection is manually disabled, there is
    no way to know if the cookies are stolen or not"
```

Figure A.18: CodeQL query that checks whether the developer has manually disabled Flask-Login's session protection functionality by setting it to `None`.

```
import python
import semmle.python.ApiGraphs

from DataFlow::Node n
where (((n = API::moduleImport("flask_login").getMember("LoginManager").
    getReturn().getMember("session_protection").getAValueReachingSink()
    or n = API::moduleImport("flask_login").getMember("login_manager").
        getMember("LoginManager").getReturn().getMember("session_protection"
        ).getAValueReachingSink())
  and n.asExpr().(StrConst).getText() = "basic")
  or (not exists(API::moduleImport("flask_login").getMember("LoginManager")
      .getReturn().getMember("session_protection").getAValueReachingSink())
    and not exists(API::moduleImport("flask_login").getMember("
        login_manager").getMember("LoginManager").getReturn().getMember("
        session_protection").getAValueReachingSink())))
  and exists(ControlFlowNode cfn |
    (cfn = API::moduleImport("flask_login").getMember("fresh_login_required
        ").getAValueReachableFromSource().asCfgNode()
      or cfn = API::moduleImport("flask_login").getMember("utils").
          getMember("fresh_login_required").getAValueReachableFromSource().
          asCfgNode())
    and not cfn.isImportMember())
select "Session protection is enabled (in basic mode)"
```

Figure A.19: CodeQL query that checks whether the basic level of session protection is enforced. It checks whether the `@fresh_login_required` decorator is used and that `session_protection` is left as default or set to basic.

```
1 import python
2 import semmle.python.ApiGraphs
3
4 from DataFlow::Node n
5 where (n = API::moduleImport("flask_login").getMember("LoginManager").
      getReturn().getMember("session_protection").getAValueReachingSink()
6     or n = API::moduleImport("flask_login").getMember("login_manager").
          getMember("LoginManager").getReturn().getMember("session_protection"
          ).getAValueReachingSink())
7   and n.asExpr().(StrConst).getText() = "strong"
8 select n.getLocation(), "Session protection is set to strong"
```

Figure A.20: CodeQL query that checks whether the highest level (strong) of session protection is enforced.

```
import python
import semmle.python.ApiGraphs
import CodeQL_Library.FlaskLogin

class FormConfiguration extends DataFlow::Configuration {
    FormConfiguration() { this = "FormConfiguration" }

    override predicate isSource(DataFlow::Node source) {
        exists(Class cls |
            source.asCfgNode() = cls.getClassObject().getACall())
    }

    override predicate isSink(DataFlow::Node sink) {
        exists(Attribute atr, AssignStmt asgn |
            exists(atr.getLocation().getFile().getRelativePath())
            and (atr.getName() = "validate"
                or atr.getName() = "validate_on_submit")
            and asgn.getATarget().(Name).getVariable() = atr.getObject().(
                Name).getVariable()
            and exists(asgn.getLocation().getFile().getRelativePath())
            and asgn.getValue().getAFlowNode() = sink.asCfgNode())
    }
}

Class getFormClasses() {
    exists(Class cls, API::Node node, AssignStmt asgn, Call call |
        exists(cls.getLocation().getFile().getRelativePath())
        and (cls.getABase().toString() = "Form"
            or cls.getABase().toString() = "BaseForm"
            or cls.getABase().toString() = "FlaskForm")
        and (node = API::moduleImport("wtforms").getMember("PasswordField")
            or node = API::moduleImport("flask_wtf").getMember("
                PasswordField"))
        and asgn = cls.getAStmt().(AssignStmt)
        and asgn.getValue().(Call).getFunc() = node.
            getAValueReachableFromSource().asExpr()
        and call = asgn.getValue().(Call)
        and (exists(call.getPositionalArg(1))
            or call.getANamedArgumentName() = "validators"
            or cls.getAMethod().getName().prefix(9 + asgn.getATarget().(
                Name).getId().length()) = "validate_" + asgn.getATarget().(
                Name).getId())
        and result = cls)
}

predicate formIsValidated(Class c) {
    exists(DataFlow::Node source, DataFlow::Node sink, FormConfiguration
        config |
        config.hasFlow(source, sink)
        and source.asCfgNode() = c.getClassObject().getACall())
}

from Class cls
where cls = getFormClasses()
    and exists(cls.getClassObject().getACall())
    and not formIsValidated(cls)
    and cls = FlaskLogin::getSignUpFormClass()
select cls, cls.getLocation(), "This form with a password field (that has
    some validators) is never validated"
```

Figure A.21: CodeQL query that checks whether the developer forgot to validate the sign-up form's password fields. The query also applies a filter in order to include only the password fields that have some validators associated with them.

```
import python
import semmle.python.ApiGraphs
import CodeQL_Library.FlaskLogin

DataFlow::Node inlineCustomValidators() {
    exists(Class cls, DataFlow::Node node, AssignStmt asgn |
        exists(cls.getLocation().getFile().getRelativePath())
        and (cls.getABase().toString() = "Form"
            or cls.getABase().toString() = "BaseForm"
            or cls.getABase().toString() = "FlaskForm")
        and (node = API::moduleImport("wtforms").getMember("PasswordField")
            .getAValueReachableFromSource()
            or node = API::moduleImport("flask_wtf").getMember("
                PasswordField").getAValueReachableFromSource())
        and exists(cls.getLocation().getFile().getRelativePath())
        and asgn = cls.getAStmt().(AssignStmt)
        and asgn.getValue().(Call).getFunc() = node.asExpr()
        and cls.getAMethod().getName().prefix(9 + asgn.getATarget().(Name).
            getId().length()) = "validate_" + asgn.getATarget().(Name).getId
            ()
        and result = node)
}

DataFlow::Node customValidators() {
    exists(DataFlow::Node node, ControlFlowNode element |
        (node = API::moduleImport("wtforms").getMember("PasswordField").
            getParameter(1).getAValueReachingSink()
            or node = API::moduleImport("flask_wtf").getMember("
                PasswordField").getParameter(1).getAValueReachingSink()
            or node = API::moduleImport("wtforms").getMember("PasswordField
                ").getKeywordParameter("validators").getAValueReachingSink()
            or node = API::moduleImport("flask_wtf").getMember("
                PasswordField").getKeywordParameter("validators").
                getAValueReachingSink())
        and (element = node.asExpr().(List).getAnElt().getAFlowNode()
            or element = node.asExpr().(Tuple).getAnElt().getAFlowNode())
        and not element = API::moduleImport("wtforms").getMember("
            validators").getAMember().getReturn().
            getAValueReachableFromSource().asCfgNode()
        and result = node)
}

from DataFlow::Node passfield, Class cls, Class supercls
where (passfield = inlineCustomValidators()
        or passfield = customValidators())
    and cls = FlaskLogin::getSignUpFormClass()
    and if exists(Class superclss | superclss.getName() = cls.getABase().(
        Name).getId())
        then supercls.getName() = cls.getABase().(Name).getId()
            and (passfield.getScope() = cls
                or passfield.getScope() = supercls)
        else passfield.getScope() = cls
select passfield, passfield.getLocation(), "Using a custom validator to
    check password strength"
```

Figure A.22: CodeQL query that checks whether custom password validators are being used in the sign-up form created using either Flask-WTF or WTForms.

```
1  import python
2  import semmle.python.ApiGraphs
3  import CodeQL_Library.FlaskLogin
4
5  bindingset[val, pos]
6  string getValue(ControlFlowNode cfg, string val, int pos) {
7      if exists(IntegerLiteral value | value.getAFlowNode() = cfg.(CallNode).
           getArgByName(val)
8          or value.getAFlowNode() = cfg.(CallNode).getArg(pos))
9      then exists(IntegerLiteral value |
10         (value.getAFlowNode() = cfg.(CallNode).getArgByName(val)
11             or value.getAFlowNode() = cfg.(CallNode).getArg(pos))
12         and result = val + " value: " + value.getValue())
13     else result = val + " value not set or it is not an integer literal"
14 }
15
16 predicate isInsideSignUpForm(DataFlow::Node passfield) {
17     exists(Class cls, Class supercls |
18         cls = FlaskLogin::getSignUpFormClass()
19         and if exists(Class superclss | superclss.getName() = cls.getABase
               ().(Name).getId())
20             then supercls.getName() = cls.getABase().(Name).getId()
21                 and (passfield.getScope() = cls
22                     or passfield.getScope() = supercls)
23             else passfield.getScope() = cls)
24 }
25
26 from DataFlow::Node node, ControlFlowNode validator
27 where (node = API::moduleImport("wtforms").getMember("PasswordField").
       getParameter(1).getAValueReachingSink()
28         or node = API::moduleImport("flask_wtf").getMember("PasswordField")
               .getParameter(1).getAValueReachingSink()
29         or node = API::moduleImport("wtforms").getMember("PasswordField").
               getKeywordParameter("validators").getAValueReachingSink()
30         or node = API::moduleImport("flask_wtf").getMember("PasswordField")
               .getKeywordParameter("validators").getAValueReachingSink())
31     and (validator = API::moduleImport("wtforms").getMember("validators").
           getMember("Length").getReturn().getAValueReachableFromSource().
           asCfgNode()
32         or validator = API::moduleImport("wtforms").getMember("validators")
               .getMember("length").getReturn().getAValueReachableFromSource().
               asCfgNode())
33     and (node.asExpr().(List).getAnElt().getAFlowNode() = validator
34         or node.asExpr().(Tuple).getAnElt().getAFlowNode() = validator)
35     and isInsideSignUpForm(node)
36 select node, node.getLocation(), "Length checks are being performed on the
       password field", getValue(validator, "max", 1), getValue(validator, "min
       ", 0)
```

Figure A.23: CodeQL query that checks whether the length validator is being used to check password strength in sign-up forms created using either Flask-WTF or WTForms. If so it also extracts the minimum and maximum length values that are being enforced.

```
import python
import semmle.python.ApiGraphs
import CodeQL_Library.FlaskLogin

string getRegexp(ControlFlowNode validator) {
    if exists(StrConst regexp | regexp.getAFlowNode() = validator.(CallNode
        ).getArgByName("regex")
        or regexp.getAFlowNode() = validator.(CallNode).getArg(0))
    then exists(StrConst regexp |
        (regexp.getAFlowNode() = validator.(CallNode).getArgByName("regex")
            or regexp.getAFlowNode() = validator.(CallNode).getArg(0))
        and result = "The regex being used is: " + regexp.getText())
    else result = "Either the regexp is not set or it is not a string"
}

predicate isInsideSignUpForm(DataFlow::Node passfield) {
    exists(Class cls, Class supercls |
        cls = FlaskLogin::getSignUpFormClass()
        and if exists(Class superclss | superclss.getName() = cls.getABase
            ().(Name).getId())
            then supercls.getName() = cls.getABase().(Name).getId()
                and (passfield.getScope() = cls
                    or passfield.getScope() = supercls)
            else passfield.getScope() = cls)
}

from DataFlow::Node node, ControlFlowNode validator
where (node = API::moduleImport("wtforms").getMember("PasswordField").
    getParameter(1).getAValueReachingSink()
        or node = API::moduleImport("flask_wtf").getMember("PasswordField")
            .getParameter(1).getAValueReachingSink()
        or node = API::moduleImport("wtforms").getMember("PasswordField").
            getKeywordParameter("validators").getAValueReachingSink()
        or node = API::moduleImport("flask_wtf").getMember("PasswordField")
            .getKeywordParameter("validators").getAValueReachingSink())
    and (validator = API::moduleImport("wtforms").getMember("validators").
        getMember("Regexp").getReturn().getAValueReachableFromSource().
        asCfgNode()
        or validator = API::moduleImport("wtforms").getMember("validators")
            .getMember("regexp").getReturn().getAValueReachableFromSource().
            asCfgNode())
    and (node.asExpr().(List).getAnElt().getAFlowNode() = validator
        or node.asExpr().(Tuple).getAnElt().getAFlowNode() = validator)
    and isInsideSignUpForm(node)
select node, node.getLocation(), "The password is being checked using a
    regexp", getRegexp(validator)
```

Figure A.24: CodeQL query that checks whether regular expressions are being used to check password strength in sign-up forms created using either Flask-WTF or WTForms.

```
1 import python
2 import semmle.python.dataflow.new.DataFlow
3
4 class PasswordValidatorsConfiguration extends DataFlow::Configuration {
5     PasswordValidatorsConfiguration() { this = "
        PasswordValidatorsConfiguration" }
6
7     override predicate isSource(DataFlow::Node source) {
8         exists(source.getLocation().getFile().getRelativePath())
9         and (source.asExpr() instanceof List
10            or source.asExpr() instanceof Tuple)
11    }
12
13    override predicate isSink(DataFlow::Node sink) {
14        exists(AssignStmt asgn, AugAssign augasgn, Name name |
15            name.getId() = "AUTH_PASSWORD_VALIDATORS"
16            and ((asgn.getATarget() = name
17                and exists(asgn.getLocation().getFile().getRelativePath())
18                and asgn.getValue().getAFlowNode() = sink.asCfgNode())
19            or (augasgn.getTarget() = name
20                and exists(augasgn.getLocation().getFile().getRelativePath
                    ())
21                and augasgn.getValue().getAFlowNode() = sink.asCfgNode()))
22        )
23    }
24 }
25
26 string output(Dict pr) {
27     if pr.getAnItem().(KeyValuePair).getKey().(StrConst).getS() = "OPTIONS"
28     then exists(KeyValuePair pair, KeyValuePair prnt |
29         prnt = pr.getAnItem()
30         and prnt.getKey().(StrConst).getS() = "OPTIONS"
31         and pair = prnt.getValue().(Dict).getAnItem()
32         and pair.getKey().(StrConst).getS() = "min_length"
33         and result = "Min value manually set: " + pair.getValue().(
            IntegerLiteral).getValue().toString())
34     else result = ""
35 }
36
37 from DataFlow::Node source, DataFlow::Node sink,
    PasswordValidatorsConfiguration config, KeyValuePair pair, Dict dct
38 where config.hasFlow(source, sink)
39     and (dct = source.asExpr().(List).getAnElt().(Dict)
40         or dct = source.asExpr().(Tuple).getAnElt().(Dict))
41     and pair = dct.getAnItem()
42     and pair.getKey().(StrConst).getS() = "NAME"
43     and (pair.getValue().(StrConst).getS() = "django.contrib.auth.
        password_validation.MinimumLengthValidator"
44         or pair.getValue().(BinaryExpr).getLeft().(StrConst).getS() + pair.
            getValue().(BinaryExpr).getRight().(StrConst).getS() = "django.
            contrib.auth.password_validation.MinimumLengthValidator")
45 select pair.getLocation(), source, sink, source.getLocation(), sink.
    getLocation(), output(dct), "Using a length password validator"
```

Figure A.25: CodeQL query that checks whether the length validator is being used to check password strength in Django applications. To use the length validator one has to include it in the list of validators specified by the `AUTH_PASSWORD_VALIDATORS` variable. The class `PasswordValidatorsConfiguration` is just needed to add data-flow analysis to the query.

```
import python
import semmle.python.dataflow.new.DataFlow

class PasswordValidatorsConfiguration extends DataFlow::Configuration {
    PasswordValidatorsConfiguration() { this = "
        PasswordValidatorsConfiguration" }

    override predicate isSource(DataFlow::Node source) {
        exists(source.getLocation().getFile().getRelativePath())
        and (source.asExpr() instanceof List
            or source.asExpr() instanceof Tuple)
    }

    override predicate isSink(DataFlow::Node sink) {
        exists(AssignStmt asgn, AugAssign augasgn, Name name |
            name.getId() = "AUTH_PASSWORD_VALIDATORS"
            and ((asgn.getATarget() = name
                and exists(asgn.getLocation().getFile().getRelativePath())
                and asgn.getValue().getAFlowNode() = sink.asCfgNode())
            or (augasgn.getTarget() = name
                and exists(augasgn.getLocation().getFile().getRelativePath
                    ())
                and augasgn.getValue().getAFlowNode() = sink.asCfgNode()))
        )
    }
}

from DataFlow::Node source, DataFlow::Node sink,
    PasswordValidatorsConfiguration config, KeyValuePair pair
where config.hasFlow(source, sink)
    and (pair = source.asExpr().(List).getAnElt().(Dict).getAnItem()
        or pair = source.asExpr().(Tuple).getAnElt().(Dict).getAnItem())
    and pair.getKey().(StrConst).getS() = "NAME"
    and (pair.getValue().(StrConst).getS() = "django.contrib.auth.
        password_validation.CommonPasswordValidator"
        or pair.getValue().(BinaryExpr).getLeft().(StrConst).getS() + pair.
            getValue().(BinaryExpr).getRight().(StrConst).getS() = "django.
            contrib.auth.password_validation.CommonPasswordValidator")
select pair.getLocation(), source, sink, source.getLocation(), sink.
    getLocation(), "Using a common password validator"
```

Figure A.26: CodeQL query that checks whether the common password validator is being used to check password strength in Django applications. To use the validator one has to include it in the list of validators specified by the `AUTH_PASSWORD_VALIDATORS` variable. The class `PasswordValidatorsConfiguration` is just needed to add data-flow analysis to the query.

```
import python
import semmle.python.dataflow.new.DataFlow

class PasswordValidatorsConfiguration extends DataFlow::Configuration {
    PasswordValidatorsConfiguration() { this = "
        PasswordValidatorsConfiguration" }

    override predicate isSource(DataFlow::Node source) {
        exists(source.getLocation().getFile().getRelativePath())
        and (source.asExpr() instanceof List
            or source.asExpr() instanceof Tuple)
    }

    override predicate isSink(DataFlow::Node sink) {
        exists(AssignStmt asgn, AugAssign augasgn, Name name |
            name.getId() = "AUTH_PASSWORD_VALIDATORS"
            and ((asgn.getATarget() = name
                and exists(asgn.getLocation().getFile().getRelativePath())
                and asgn.getValue().getAFlowNode() = sink.asCfgNode())
            or (augasgn.getTarget() = name
                and exists(augasgn.getLocation().getFile().getRelativePath
                    ())
                and augasgn.getValue().getAFlowNode() = sink.asCfgNode()))
        )
    }
}

from DataFlow::Node source, DataFlow::Node sink,
    PasswordValidatorsConfiguration config, KeyValuePair pair
where config.hasFlow(source, sink)
    and (pair = source.asExpr().(List).getAnElt().(Dict).getAnItem()
        or pair = source.asExpr().(Tuple).getAnElt().(Dict).getAnItem())
    and pair.getKey().(StrConst).getS() = "NAME"
    and (pair.getValue().(StrConst).getS() = "django.contrib.auth.
        password_validation.NumericPasswordValidator"
        or pair.getValue().(BinaryExpr).getLeft().(StrConst).getS() + pair.
            getValue().(BinaryExpr).getRight().(StrConst).getS() = "django.
            contrib.auth.password_validation.NumericPasswordValidator")
select pair.getLocation(), source, sink, source.getLocation(), sink.
    getLocation(), "Using a numeric password validator (checking that the
    password is not entirely numeric)"
```

Figure A.27: CodeQL query that checks whether the numeric password validator is being used to check password strength in Django applications. To use the validator one has to include it in the list of validators specified by the `AUTH_PASSWORD_VALIDATORS` variable. The class `PasswordValidatorsConfiguration` is just needed to add data-flow analysis to the query.

```
import python
import semmle.python.dataflow.new.DataFlow

class PasswordValidatorsConfiguration extends DataFlow::Configuration {
    PasswordValidatorsConfiguration() { this = "
        PasswordValidatorsConfiguration" }

    override predicate isSource(DataFlow::Node source) {
        exists(source.getLocation().getFile().getRelativePath())
        and (source.asExpr() instanceof List
            or source.asExpr() instanceof Tuple)
    }

    override predicate isSink(DataFlow::Node sink) {
        exists(AssignStmt asgn, AugAssign augasgn, Name name |
            name.getId() = "AUTH_PASSWORD_VALIDATORS"
            and ((asgn.getATarget() = name
                and exists(asgn.getLocation().getFile().getRelativePath())
                and asgn.getValue().getAFlowNode() = sink.asCfgNode())
            or (augasgn.getTarget() = name
                and exists(augasgn.getLocation().getFile().getRelativePath
                    ())
                and augasgn.getValue().getAFlowNode() = sink.asCfgNode()))
        )
    }
}

from DataFlow::Node source, DataFlow::Node sink,
    PasswordValidatorsConfiguration config, KeyValuePair pair
where config.hasFlow(source, sink)
    and (pair = source.asExpr().(List).getAnElt().(Dict).getAnItem()
        or pair = source.asExpr().(Tuple).getAnElt().(Dict).getAnItem())
    and pair.getKey().(StrConst).getS() = "NAME"
    and (pair.getValue().(StrConst).getS() = "django.contrib.auth.
        password_validation.UserAttributeSimilarityValidator"
        or pair.getValue().(BinaryExpr).getLeft().(StrConst).getS() + pair.
            getValue().(BinaryExpr).getRight().(StrConst).getS() = "django.
            contrib.auth.password_validation.
            UserAttributeSimilarityValidator")
select pair.getLocation(), source, sink, source.getLocation(), sink.
    getLocation(), "Using a password similarity (with username and other
    fields) validator"
```

Figure A.28: CodeQL query that checks whether the similarity password validator is being used to check password strength in Django applications. To use the validator one has to include it in the list of validators specified by the `AUTH_PASSWORD_VALIDATORS` variable. The class `PasswordValidatorsConfiguration` is just needed to add data-flow analysis to the query.

```
import python
import semmle.python.dataflow.new.DataFlow

class PasswordValidatorsConfiguration extends DataFlow::Configuration {
    PasswordValidatorsConfiguration() { this = "
        PasswordValidatorsConfiguration" }

    override predicate isSource(DataFlow::Node source) {
        exists(source.getLocation().getFile().getRelativePath())
        and (source.asExpr() instanceof List
            or source.asExpr() instanceof Tuple)
    }

    override predicate isSink(DataFlow::Node sink) {
        exists(AssignStmt asgn, AugAssign augasgn, Name name |
            name.getId() = "AUTH_PASSWORD_VALIDATORS"
            and ((asgn.getATarget() = name
                and exists(asgn.getLocation().getFile().getRelativePath())
                and asgn.getValue().getAFlowNode() = sink.asCfgNode())
            or (augasgn.getTarget() = name
                and exists(augasgn.getLocation().getFile().getRelativePath
                    ())
                and augasgn.getValue().getAFlowNode() = sink.asCfgNode()))
        )
    }
}

from DataFlow::Node source, DataFlow::Node sink,
    PasswordValidatorsConfiguration config, KeyValuePair pair
where config.hasFlow(source, sink)
    and (pair = source.asExpr().(List).getAnElt().(Dict).getAnItem()
        or pair = source.asExpr().(Tuple).getAnElt().(Dict).getAnItem())
    and pair.getKey().(StrConst).getS() = "NAME"
    and (if exists(pair.getValue().(StrConst).getS().prefix(40))
        then pair.getValue().(StrConst).getS().prefix(40) != "django.
            contrib.auth.password_validation."
        else pair.getValue() instanceof Str)
select pair.getLocation(), source, sink, source.getLocation(), sink.
    getLocation(), "Using a custom password validator"
```

Figure A.29: CodeQL query that checks whether custom password validators are being used to check password strength in Django applications. This is done by including custom classes in the list of validators specified by the `AUTH_PASSWORD_VALIDATORS` variable. The class `PasswordValidatorsConfiguration` is just needed to add data-flow analysis to the query.

```
import python
import CodeQL_Library.DjangoSession

bindingset[attrName, value]
predicate attrCheck(Class cls, string attrName, int value) {
    exists(Expr expr |
        expr = DjangoSession::getAttrValue(cls, attrName)
        and if expr instanceof IntegerLiteral
            then expr.(IntegerLiteral).getValue() < value
            else none())
}

string output(Class cls) {
    if exists(DjangoSession::overridenImplOfHashingAlgIsUsed("
        Argon2PasswordHasher"))
    then if not DjangoSession::getAttrValue(cls, "time_cost") instanceof
        IntegerLiteral or not DjangoSession::getAttrValue(cls, "memory_cost"
        ) instanceof IntegerLiteral or not DjangoSession::getAttrValue(cls,
        "parallelism") instanceof IntegerLiteral
        then result = "Argon2 is being used as the password hashing
            algorithm but binary expressions are being used so don't know if
             it's owasp compliant"
        else result = "Argon2 is being used as the password hashing
            algorithm and it's owasp compliant"
    else if exists(DjangoSession::defaultImplOfHashingAlgIsUsed("django.
        contrib.auth.hashers.Argon2PasswordHasher"))
        then result = "Argon2 is being used as the password hashing
            algorithm and it's owasp compliant"
        else none()
}

from Class cls
where (cls = DjangoSession::overridenImplOfHashingAlgIsUsed("
    Argon2PasswordHasher")
        and not attrCheck(cls, "time_cost", 2)
        and not attrCheck(cls, "memory_cost", 19456)
        and not attrCheck(cls, "parallelism", 1))
    or exists(DjangoSession::defaultImplOfHashingAlgIsUsed("django.contrib.
        auth.hashers.Argon2PasswordHasher"))
select output(cls)
```

Figure A.30: CodeQL query that checks whether Argon2id is being used in an OWASP compliant configuration by the Django application. In order to use it one has to include it as a first item in the list of hashers specified by the `PASSWORD_HASHERS` variable, and this is checked using the `DjangoSession` library.

```
import python
import CodeQL_Library.DjangoSession

from ControlFlowNode cfn
where cfn = DjangoSession::defaultImplOfHashingAlgIsUsed("django.contrib.
    auth.hashers.Argon2PasswordHasher").getAFlowNode()
    or cfn = DjangoSession::overridenImplOfHashingAlgIsUsed("
        Argon2PasswordHasher").getClassObject()
select cfn, cfn.getLocation(), "Argon2 is being used as the password
    hashing algorithm"
```

Figure A.31: CodeQL query that checks whether Argon2id is being used by the Django application. In order to use it one has to include it as a first item in the list of hashers specified by the `PASSWORD_HASHERS` variable, and this is checked using the `DjangoSession` library.

```
1 import python
2 import CodeQL_Library.DjangoSession
3
4 bindingset[attrName, value]
5 predicate attrCheck(Class cls, string attrName, int value) {
6     exists(Expr expr |
7         expr = DjangoSession::getAttrValue(cls, attrName)
8         and if expr instanceof IntegerLiteral
9             then expr.(IntegerLiteral).getValue() < value
10            else none())
11 }
12
13 string output(Class cls) {
14     if exists(DjangoSession::overridenImplOfHashingAlgIsUsed("
           BCryptPasswordHasher"))
15     then if not DjangoSession::getAttrValue(cls, "rounds") instanceof
           IntegerLiteral
16         then result = "Bcrypt is being used as the password hashing
               algorithm but binary expressions are being used so don't know if
                it's owasp compliant"
17         else result = "Bcrypt is being used as the password hashing
               algorithm and it's owasp compliant"
18     else if exists(DjangoSession::defaultImplOfHashingAlgIsUsed("django.
           contrib.auth.hashers.BCryptPasswordHasher"))
19         then result = "Bcrypt is being used as the password hashing
               algorithm and it's owasp compliant"
20         else none()
21 }
22
23 from Class cls
24 where (cls = DjangoSession::overridenImplOfHashingAlgIsUsed("
       BCryptPasswordHasher")
25         and not attrCheck(cls, "rounds", 10))
26     or exists(DjangoSession::defaultImplOfHashingAlgIsUsed("django.contrib.
           auth.hashers.BCryptPasswordHasher"))
27 select output(cls)
```

Figure A.32: CodeQL query that checks whether bcrypt is being used in an OWASP compliant configuration by the Django application. In order to use it one has to include it as a first item in the list of hashers specified by the `PASSWORD_HASHERS` variable, and this is checked using the `DjangoSession` library.

```
1 import python
2 import CodeQL_Library.DjangoSession
3
4 from ControlFlowNode cfn
5 where cfn = DjangoSession::defaultImplOfHashingAlgIsUsed("django.contrib.
       auth.hashers.BCryptPasswordHasher").getAFlowNode()
6     or cfn = DjangoSession::overridenImplOfHashingAlgIsUsed("
           BCryptPasswordHasher").getClassObject()
7     or cfn = DjangoSession::defaultImplOfHashingAlgIsUsed("django.contrib.
           auth.hashers.BCryptSHA256PasswordHasher").getAFlowNode()
8     or cfn = DjangoSession::overridenImplOfHashingAlgIsUsed("
           BCryptSHA256PasswordHasher").getClassObject()
9 select cfn, cfn.getLocation(), "Bcrypt is being used as the password
       hashing algorithm"
```

Figure A.33: CodeQL query that checks whether bcrypt is being used by the Django application. In order to use it one has to include it as a first item in the list of hashers specified by the `PASSWORD_HASHERS` variable, and this is checked using the `DjangoSession` library.

```
import python
import CodeQL_Library.DjangoSession

bindingset[attrName, value]
predicate attrCheck(Class cls, string attrName, int value) {
    exists(Expr expr |
        expr = DjangoSession::getAttrValue(cls, attrName)
        and if expr instanceof IntegerLiteral
            then expr.(IntegerLiteral).getValue() < value
            else none())
}

bindingset[attrName, value]
predicate attrCheckNonCompliant(Class cls, string attrName, int value) {
    exists(Expr expr |
        expr = DjangoSession::getAttrValue(cls, attrName)
        and if expr instanceof IntegerLiteral
            then expr.(IntegerLiteral).getValue() < value
            else none())
    or not exists(DjangoSession::getAttrValue(cls, attrName))
}

string output(Class cls) {
    if exists(DjangoSession::overridenImplOfHashingAlgIsUsed("PBKDF2PasswordHasher")) or exists(DjangoSession::overridenImplOfHashingAlgIsUsed("PBKDF2SHA1PasswordHasher"))
    then if not DjangoSession::getAttrValue(cls, "iterations") instanceof IntegerLiteral
        then result = "PBKDF2 is being used as the password hashing algorithm but binary expressions are being used so don't know if it's owasp compliant"
        else result = "PBKDF2 is being used as the password hashing algorithm and it's owasp compliant"
    else if exists(DjangoSession::defaultImplOfHashingAlgIsUsed("django.contrib.auth.hashers.PBKDF2PasswordHasher")) or not exists(DataFlow3::Node source, DataFlow3::Node sink, DjangoSession::PasswordHashersConfiguration config | config.hasFlow(source, sink))
        then result = "PBKDF2 is being used as the password hashing algorithm and it's owasp compliant"
        else none()
}

from Class cls
where (cls = DjangoSession::overridenImplOfHashingAlgIsUsed("PBKDF2PasswordHasher")
        and not attrCheck(cls, "iterations", 600000))
    or (cls = DjangoSession::overridenImplOfHashingAlgIsUsed("PBKDF2SHA1PasswordHasher")
        and not attrCheckNonCompliant(cls, "iterations", 1300000))
    or exists(DjangoSession::defaultImplOfHashingAlgIsUsed("django.contrib.auth.hashers.PBKDF2PasswordHasher"))
    or not exists(DataFlow3::Node source, DataFlow3::Node sink, DjangoSession::PasswordHashersConfiguration config |
        config.hasFlow(source, sink))
select output(cls)
```

Figure A.34: CodeQL query that checks whether PBKDF2 is being used in an OWASP compliant configuration by the Django application. In order to use it one has to include it as a first item in the list of hashers specified by the `PASSWORD_HASHERS` variable, and this is checked using the `DjangoSession` library.

```
1  import python
2  import CodeQL_Library.DjangoSession
3
4  where exists(ControlFlowNode cfn |
5          cfn = DjangoSession::defaultImplOfHashingAlgIsUsed("django.contrib.
               auth.hashers.PBKDF2PasswordHasher").getAFlowNode()
6          or cfn = DjangoSession::overridenImplOfHashingAlgIsUsed("
               PBKDF2PasswordHasher").getClassObject()
7          or cfn = DjangoSession::defaultImplOfHashingAlgIsUsed("django.
               contrib.auth.hashers.PBKDF2SHA1PasswordHasher").getAFlowNode()
8          or cfn = DjangoSession::overridenImplOfHashingAlgIsUsed("
               PBKDF2SHA1PasswordHasher").getClassObject())
9      or not exists(DataFlow3::Node source, DataFlow3::Node sink,
          DjangoSession::PasswordHashersConfiguration config |
10         config.hasFlow(source, sink))
11 select "PBKDF2 is being used as the password hashing algorithm"
```

Figure A.35: CodeQL query that checks whether PBKDF2 is being used by the Django application. In order to use it one has to include it as a first item in the list of hashers specified by the `PASSWORD_HASHERS` variable, and this is checked using the `DjangoSession` library.

```
1 import python
2 import CodeQL_Library.DjangoSession
3
4 bindingset[attrName, value]
5 predicate attrCheck(Class cls, string attrName, int value) {
6     exists(Expr expr |
7         expr = DjangoSession::getAttrValue(cls, attrName)
8         and if expr instanceof IntegerLiteral
9             then expr.(IntegerLiteral).getValue() < value
10            else none())
11 }
12
13 bindingset[attrName, value]
14 predicate attrCheckNonCompliant(Class cls, string attrName, int value) {
15     exists(Expr expr |
16         expr = DjangoSession::getAttrValue(cls, attrName)
17         and if expr instanceof IntegerLiteral
18             then expr.(IntegerLiteral).getValue() < value
19             else none())
20     or not exists(DjangoSession::getAttrValue(cls, attrName))
21 }
22
23 string output(Class cls) {
24     if not DjangoSession::getAttrValue(cls, "work_factor") instanceof
        IntegerLiteral or not DjangoSession::getAttrValue(cls, "block_size")
         instanceof IntegerLiteral or not DjangoSession::getAttrValue(cls, "
        parallelism") instanceof IntegerLiteral
25     then result = "Scrypt is being used as the password hashing algorithm
        but binary expressions are being used so don't know if it's owasp
        compliant"
26     else result = "Scrypt is being used as the password hashing algorithm
        and it's owasp compliant"
27 }
28
29 from Class cls
30 where (cls = DjangoSession::overridenImplOfHashingAlgIsUsed("
    ScryptPasswordHasher")
31         and not attrCheckNonCompliant(cls, "work_factor", 131072)
32         and not attrCheck(cls, "block_size", 8)
33         and not attrCheck(cls, "parallelism", 1))
34 select output(cls)
```

Figure A.36: CodeQL query that checks whether scrypt is being used in an OWASP compliant configuration by the Django application. In order to use it one has to include it as a first item in the list of hashers specified by the `PASSWORD_HASHERS` variable, and this is checked using the `DjangoSession` library.

```
1 import python
2 import CodeQL_Library.DjangoSession
3
4 from ControlFlowNode cfn
5 where cfn = DjangoSession::defaultImplOfHashingAlgIsUsed("django.contrib.
    auth.hashers.ScryptPasswordHasher").getAFlowNode()
6     or cfn = DjangoSession::overridenImplOfHashingAlgIsUsed("
        ScryptPasswordHasher").getClassObject()
7 select cfn, cfn.getLocation(), "Scrypt is being used as the password
    hashing algorithm"
```

Figure A.37: CodeQL query that checks whether scrypt is being used by the Django application. In order to use it one has to include it as a first item in the list of hashers specified by the `PASSWORD_HASHERS` variable, and this is checked using the `DjangoSession` library.

```
1 import python
2 import CodeQL_Library.DjangoSession
3
4 from Class cls, StrConst str
5 where str = DjangoSession::getDefaultHashingAlg()
6     and cls.getName() = str.getS().splitAt(".")
7     and not cls.getABase() = API::moduleImport("django").getMember("contrib
          ").getMember("auth").getMember("hashers").getAMember().
          getAValueReachableFromSource().asExpr()
8     and (if exists(str.getS().prefix(28))
9         then str.getS().prefix(28) != "django.contrib.auth.hashers."
10        else any())
11    and exists(cls.getLocation().getFile().getRelativePath())
12 select cls, cls.getLocation(), "Using a completely custom password hasher"
```

Figure A.38: CodeQL query that checks whether custom functions are being used to hash the password by the Django application. In order to use a custom function one has to include it as a first item in the list of hashers specified by the `PASSWORD_HASHERS` variable, and this is checked using the `DjangoSession` library.

```
1 import python
2 import CodeQL_Library.DjangoSession
3
4 from ControlFlowNode cfn
5 where cfn = DjangoSession::defaultImplOfHashingAlgIsUsed("django.contrib.
      auth.hashers.MD5PasswordHasher").getAFlowNode()
6     or cfn = DjangoSession::overridenImplOfHashingAlgIsUsed("
          MD5PasswordHasher").getClassObject()
7 select cfn, cfn.getLocation(), "MD5 is being used as the password hashing
      algorithm"
```

Figure A.39: CodeQL query that checks whether built-in unsafe (according to OWASP) password hashing algorithms are being used by the Django application. In order to use them, one has to include them as a first item in the list of hashers specified by the `PASSWORD_HASHERS` variable, and this is checked using the `DjangoSession` library.

```
import python
import semmle.python.ApiGraphs

DataFlow::Node libraryIsUsed() {
    exists(DataFlow::Node node |
        node = API::moduleImport("bcrypt").getMember("hashpw").
            getAValueReachableFromSource()
        and exists(node.asCfgNode())
        and not node.asExpr() instanceof ImportMember
        and result = node)
}

predicate workFactor() {
    exists(DataFlow::Node node |
        (node = API::moduleImport("bcrypt").getMember("gensalt").
            getParameter(0).getAValueReachingSink()
            or node = API::moduleImport("bcrypt").getMember("gensalt").
                getKeywordParameter("rounds").getAValueReachingSink())
        and node.asExpr().(IntegerLiteral).getValue() < 10) // owasp
            recommendation minimum
}

from DataFlow::Node node
where node = libraryIsUsed()
    and not workFactor()
select node, node.getLocation(), "Bcrypt is being used, it's compliant with
     owasp guidelines, but it doesn't handle passwords that are longer than
    72 bytes, so should also check that there is a limit on the password
    length (by looking at the password strength length checks queries)"
```

Figure A.40: CodeQL query that checks whether the Bcrypt library is being used to hash the passwords in an OWASP compliant configuration. This is done by using the `hashpw` function.

```
import python
import semmle.python.ApiGraphs

from DataFlow::Node node
where node = API::moduleImport("bcrypt").getMember("hashpw").
    getAValueReachableFromSource()
    and exists(node.asCfgNode())
    and not node.asExpr() instanceof ImportMember
select node, node.getLocation(), "Bcrypt is being used"
```

Figure A.41: CodeQL query that checks whether the Bcrypt library is being used to hash the passwords. This is done by using the `hashpw` function.

```
1 import python
2 import semmle.python.ApiGraphs
3 import CodeQL_Library.FlaskLogin
4
5 DataFlow::Node libraryIsUsed() {
6     exists(DataFlow::Node node |
7         (node = API::moduleImport("flask_bcrypt").getMember("Bcrypt").
              getReturn().getMember("generate_password_hash").
              getAValueReachableFromSource()
8             or node = API::moduleImport("flask_bcrypt").getMember("
                  generate_password_hash").getAValueReachableFromSource())
9         and exists(node.asCfgNode())
10        and not node.asExpr() instanceof ImportMember
11        and result = node)
12 }
13
14 predicate workFactor() {
15     exists(DataFlow::Node node |
16         (node = API::moduleImport("flask_bcrypt").getMember("Bcrypt").
              getReturn().getMember("generate_password_hash").
              getKeywordParameter("rounds").getAValueReachingSink()
17             or node = API::moduleImport("flask_bcrypt").getMember("Bcrypt")
                  .getReturn().getMember("generate_password_hash").
                  getParameter(1).getAValueReachingSink()
18             or node = API::moduleImport("flask_bcrypt").getMember("
                  generate_password_hash").getKeywordParameter("rounds").
                  getAValueReachingSink()
19             or node = API::moduleImport("flask_bcrypt").getMember("
                  generate_password_hash").getParameter(1).
                  getAValueReachingSink())
20         and node.asExpr().(IntegerLiteral).getValue() < 10) // owasp
              recommendation minimum
21     or exists(Expr expr |
22         expr = FlaskLogin::getConfigValue("BCRYPT_LOG_ROUNDS")
23         and expr.(IntegerLiteral).getValue() < 10) // owasp recommendation
              minimum
24 }
25
26 predicate length() {
27     exists(Expr expr |
28         expr = FlaskLogin::getConfigValue("BCRYPT_HANDLE_LONG_PASSWORDS")
29         and expr.(ImmutableLiteral).booleanValue() = true)
30 }
31
32 string output() {
33     if length()
34     then result = "Flask-Bcrypt is being used, it's compliant with owasp
          guidelines and it's set to handle passwords that are longer than 72
          bytes"
35     else result = "Flask-Bcrypt is being used and it's compliant with owasp
           guidelines, however it doesn't handle passwords that are longer
          than 72 bytes, so should also check that there is a limit on the
          password length (by looking at the password strength length checks
          queries)"
36 }
37
38 from DataFlow::Node node
39 where node = libraryIsUsed()
40     and not workFactor()
41 select node, node.getLocation(), output()
```

Figure A.42: CodeQL query that checks whether the Flask-Bcrypt library is being used to hash the passwords in an OWASP compliant configuration. This is done by calling the `generate_password_hash` function.

```
1 import python
2 import semmle.python.ApiGraphs
3
4 from DataFlow::Node node
5 where (node = API::moduleImport("flask_bcrypt").getMember("Bcrypt").
      getReturn().getMember("generate_password_hash").
      getAValueReachableFromSource()
6         or node = API::moduleImport("flask_bcrypt").getMember("
              generate_password_hash").getAValueReachableFromSource())
7     and exists(node.asCfgNode())
8     and not node.asExpr() instanceof ImportMember
9 select node, node.getLocation(), "Flask-Bcrypt is being used"
```

Figure A.43: CodeQL query that checks whether the Flask-Bcrypt library is being used to hash passwords. This is done by calling the `generate_password_hash` function.

```
1 import python
2 import semmle.python.ApiGraphs
3
4 from ControlFlowNode node
5 where (node = API::moduleImport("hashlib").getMember("pbkdf2_hmac").
      getReturn().getAValueReachableFromSource().asCfgNode()
6         or node = API::moduleImport("hashlib").getMember("scrypt").
              getReturn().getAValueReachableFromSource().asCfgNode())
7     and (exists(node.(CallNode).getArg(0))
8         or exists(node.(CallNode).getArg(1))
9         or exists(node.(CallNode).getArgByName("password")))
10    and not node.isImportMember()
11 select node, node.getLocation(), "Hashlib is being used to hash passwords"
```

Figure A.44: CodeQL query that checks whether the hashlib library is being used to hash passwords. This is done by using either the `pbkdf2_hmac` or the `scrypt` functions, since these are the two hashing functions designated for password hashing in hashlib.

```
1 import python
2 import semmle.python.ApiGraphs
3
4 bindingset[method, iterations]
5 predicate isCompliant(string method, int iterations) {
6     (method = "sha256"
7         and iterations >= 600000)
8     or (method = "sha512"
9         and iterations >= 210000)
10    or (method = "sha1"
11        and iterations >= 1300000)
12 }
13
14 from ControlFlowNode node, StrConst method, IntegerLiteral iterations
15 where node = API::moduleImport("hashlib").getMember("pbkdf2_hmac").
      getReturn().getAValueReachableFromSource().asCfgNode()
16    and (exists(node.(CallNode).getArg(1))
17        or exists(node.(CallNode).getArgByName("password")))
18    and not node.isImportMember()
19    and (method.getAFlowNode() = node.(CallNode).getArg(0)
20        or method.getAFlowNode() = node.(CallNode).getArgByName("hash_name"
              ))
21    and (iterations.getAFlowNode() = node.(CallNode).getArg(3)
22        or iterations.getAFlowNode() = node.(CallNode).getArgByName("
              iterations"))
23    and isCompliant(method.getText(), iterations.getValue())
24 select node, node.getLocation(), "Hashlib PBKDF2 is being used to hash
      passwords and it's owasp compliant"
```

Figure A.45: CodeQL query that checks whether hahslib's PBKDF2 function is being used in an OWASP compliant configuration to hash passwords.

```
import python
import semmle.python.ApiGraphs

from ControlFlowNode node
where node = API::moduleImport("hashlib").getMember("pbkdf2_hmac").
    getReturn().getAValueReachableFromSource().asCfgNode()
    and (exists(node.(CallNode).getArg(1))
        or exists(node.(CallNode).getArgByName("password")))
    and not node.isImportMember()
select node, node.getLocation(), "Hashlib PBKDF2 is being used to hash
    passwords"
```

Figure A.46: CodeQL query that checks whether hahslib's PBKDF2 function is being used to hash passwords.

```
import python
import semmle.python.ApiGraphs

bindingset[n, r, p]
predicate isCompliant(int n, int r, int p) {
    n >= 131072 and r >= 8 and p >= 1
}

from ControlFlowNode node, IntegerLiteral n, IntegerLiteral r,
    IntegerLiteral p
where node = API::moduleImport("hashlib").getMember("scrypt").getReturn().
    getAValueReachableFromSource().asCfgNode()
    and (exists(node.(CallNode).getArg(0))
        or exists(node.(CallNode).getArgByName("password")))
    and not node.isImportMember()
    and n.getAFlowNode() = node.(CallNode).getArgByName("n")
    and r.getAFlowNode() = node.(CallNode).getArgByName("r")
    and p.getAFlowNode() = node.(CallNode).getArgByName("p")
    and isCompliant(n.getValue(), r.getValue(), p.getValue())
select node, node.getLocation(), "Hashlib scrypt is being used to hash
    passwords and it's owasp compliant"
```

Figure A.47: CodeQL query that checks whether hahslib's scrypt function is being used in an OWASP compliant configuration to hash passwords.

```
import python
import semmle.python.ApiGraphs

from ControlFlowNode node
where node = API::moduleImport("hashlib").getMember("scrypt").getReturn().
    getAValueReachableFromSource().asCfgNode()
    and (exists(node.(CallNode).getArg(0))
        or exists(node.(CallNode).getArgByName("password")))
    and not node.isImportMember()
select node, node.getLocation(), "Hashlib scrypt is being used to hash
    passwords"
```

Figure A.48: CodeQL query that checks whether hahslib's scrypt function is being used to hash passwords.

```
1  import python
2  import semmle.python.ApiGraphs
3
4  from DataFlow::Node node
5  where (node = API::moduleImport("passlib").getMember("hash").getAMember().
       getMember("hash").getAValueReachableFromSource()
6          or node = API::moduleImport("passlib").getMember("hash").getAMember
              ().getMember("using").getReturn().getMember("hash").
              getAValueReachableFromSource()
7          or node = API::moduleImport("passlib").getMember("hash").getAMember
              ().getMember("encrypt").getAValueReachableFromSource()
8          or node = API::moduleImport("passlib").getMember("hash").getAMember
              ().getMember("using").getReturn().getMember("encrypt").
              getAValueReachableFromSource()
9          or node = API::moduleImport("passlib").getMember("handlers").
              getAMember().getAMember().getMember("hash").
              getAValueReachableFromSource()
10         or node = API::moduleImport("passlib").getMember("handlers").
              getAMember().getAMember().getMember("using").getReturn().
              getMember("hash").getAValueReachableFromSource()
11         or node = API::moduleImport("passlib").getMember("handlers").
              getAMember().getAMember().getMember("encrypt").
              getAValueReachableFromSource()
12         or node = API::moduleImport("passlib").getMember("handlers").
              getAMember().getAMember().getMember("using").getReturn().
              getMember("encrypt").getAValueReachableFromSource()
13         or node = API::moduleImport("passlib").getMember("context").
              getMember("CryptContext").getReturn().getMember("hash").
              getAValueReachableFromSource()
14         or node = API::moduleImport("passlib").getMember("context").
              getMember("CryptContext").getReturn().getMember("encrypt").
              getAValueReachableFromSource())
15     and exists(node.asCfgNode())
16     and not node.asExpr() instanceof ImportMember
17 select node, node.getLocation(), "PassLib is being used"
```

Figure A.49: CodeQL query that checks whether the PassLib library is being used to hash passwords. Therefore, the query checks whether one of Passlib's functions designated for password hashing are being used.

```
1 import python
2 import semmle.python.ApiGraphs
3 import CodeQL_Library.Passlib
4
5 predicate memoryConfiguration(API::Node node) {
6     exists(DataFlow::Node param |
7         (param = node.getParameter(1).getAValueReachingSink()
8             or param = node.getKeywordParameter("memory_cost").
                getAValueReachingSink())
9         and param.asExpr().(IntegerLiteral).getValue() < 19456) // 19 MiB (
            owasp recommendation minimum)
10 }
11
12 predicate iterationCount(API::Node node) {
13     exists(DataFlow::Node param |
14         (param = node.getParameter(3).getAValueReachingSink()
15             or param = node.getKeywordParameter("time_cost").
                getAValueReachingSink()
16             or param = node.getKeywordParameter("rounds").
                getAValueReachingSink())
17         and param.asExpr().(IntegerLiteral).getValue() < 2) // owasp
            recommendation minimum
18 }
19
20 predicate degreeOfParallelism(API::Node node) {
21     exists(DataFlow::Node param |
22         param = node.getKeywordParameter("parallelism").
            getAValueReachingSink()
23         and param.asExpr().(IntegerLiteral).getValue() < 1) // owasp
            recommendation minimum
24 }
25
26 predicate argonType(API::Node node) {
27     exists(DataFlow::Node param |
28         (param = node.getParameter(0).getAValueReachingSink()
29             or param = node.getKeywordParameter("type").
                getAValueReachingSink())
30         and param.asExpr().(StrConst).getS() != "ID") // owasp
            recommendation
31 }
32
33 from DataFlow::Node hash
34 where exists(API::Node node |
35         node = PassLib::getCustomUsingNode("argon2")
36         and not memoryConfiguration(node)
37         and not iterationCount(node)
38         and not degreeOfParallelism(node)
39         and not argonType(node)
40         and hash = node.getReturn().getMember("hash").
            getAValueReachableFromSource())
41     or hash = PassLib::getDefaultUsageNode("argon2")
42 select hash, hash.getLocation(), "PassLib is being used with argon2 and it'
    s compliant with owasp guidelines"
```

Figure A.50: CodeQL query that checks whether PassLib's Argon2id function is being used in an OWASP compliant configuration to hash passwords.

```
1  import python
2  import semmle.python.ApiGraphs
3
4  from DataFlow::Node node
5  where (node = API::moduleImport("passlib").getMember("hash").getMember("
       argon2").getMember("hash").getAValueReachableFromSource()
6          or node = API::moduleImport("passlib").getMember("hash").getMember(
               "argon2").getMember("using").getReturn().getMember("hash").
               getAValueReachableFromSource()
7          or node = API::moduleImport("passlib").getMember("hash").getMember(
               "argon2").getMember("encrypt").getAValueReachableFromSource()
8          or node = API::moduleImport("passlib").getMember("hash").getMember(
               "argon2").getMember("using").getReturn().getMember("encrypt").
               getAValueReachableFromSource()
9          or node = API::moduleImport("passlib").getMember("handlers").
               getMember("argon2").getMember("argon2").getMember("hash").
               getAValueReachableFromSource()
10         or node = API::moduleImport("passlib").getMember("handlers").
               getMember("argon2").getMember("argon2").getMember("using").
               getReturn().getMember("hash").getAValueReachableFromSource()
11         or node = API::moduleImport("passlib").getMember("handlers").
               getMember("argon2").getMember("argon2").getMember("encrypt").
               getAValueReachableFromSource()
12         or node = API::moduleImport("passlib").getMember("handlers").
               getMember("argon2").getMember("argon2").getMember("using").
               getReturn().getMember("encrypt").getAValueReachableFromSource())
13     and exists(node.asCfgNode())
14     and not node.asExpr() instanceof ImportMember
15 select node, node.getLocation(), "PassLib's argon2 hasher is being used"
```

Figure A.51: CodeQL query that checks whether PassLib's Argon2id function is being used to hash passwords.

```
import python
import semmle.python.ApiGraphs
import CodeQL_Library.Passlib

predicate workFactor(API::Node node) {
    exists(DataFlow::Node param |
        param = node.getKeywordParameter("rounds").getAValueReachingSink()
        and param.asExpr().(IntegerLiteral).getValue() < 10) // owasp recommendation minimum
}

string outputs() {
    exists(DataFlow::Node hash, API::Node node |
        ((node = PassLib::getCustomUsingNode("bcrypt")
                and not workFactor(node)
                and hash = node.getReturn().getMember("hash").getAValueReachableFromSource())
            or hash = PassLib::getDefaultUsageNode("bcrypt"))
        and result = hash.toString() + ", " + hash.getLocation().toString() + ", PassLib is being used with bcrypt and it's compliant with owasp guidelines, however it doesn't handle passwords that are longer than 72 bytes, so should also check that there is a limit on the password length (by looking at the password strength length checks queries)")
}

string outputl() {
    exists(DataFlow::Node hash, API::Node node |
        ((node = PassLib::getCustomUsingNode("bcrypt_sha256")
                and not workFactor(node)
                and hash = node.getReturn().getMember("hash").getAValueReachableFromSource())
            or hash = PassLib::getDefaultUsageNode("bcrypt_sha256"))
        and result = hash.toString() + ", " + hash.getLocation().toString() + ", PassLib is being used with bcrypt, it's compliant with owasp guidelines and it's set to handle passwords that are longer than 72 bytes")
}

string output() {
    if exists(outputs())
    then if exists(outputl())
        then result = outputs() + "; " + outputl()
        else result = outputs()
    else if exists(outputl())
        then result = outputl()
        else none()
}

select output()
```

Figure A.52: CodeQL query that checks whether PassLib's bcrypt function is being used in an OWASP compliant configuration to hash passwords.

```
1 import python
2 import semmle.python.ApiGraphs
3
4 from DataFlow::Node node
5 where (node = API::moduleImport("passlib").getMember("hash").getMember("
      bcrypt").getMember("hash").getAValueReachableFromSource()
6          or node = API::moduleImport("passlib").getMember("hash").getMember(
               "bcrypt").getMember("using").getReturn().getMember("hash").
               getAValueReachableFromSource()
7          or node = API::moduleImport("passlib").getMember("hash").getMember(
               "bcrypt_sha256").getMember("hash").getAValueReachableFromSource
               ()
8          or node = API::moduleImport("passlib").getMember("hash").getMember(
               "bcrypt_sha256").getMember("using").getReturn().getMember("hash"
               ).getAValueReachableFromSource()
9          or node = API::moduleImport("passlib").getMember("hash").getMember(
               "bcrypt").getMember("encrypt").getAValueReachableFromSource()
10         or node = API::moduleImport("passlib").getMember("hash").getMember(
               "bcrypt").getMember("using").getReturn().getMember("encrypt").
               getAValueReachableFromSource()
11         or node = API::moduleImport("passlib").getMember("hash").getMember(
               "bcrypt_sha256").getMember("encrypt").
               getAValueReachableFromSource()
12         or node = API::moduleImport("passlib").getMember("hash").getMember(
               "bcrypt_sha256").getMember("using").getReturn().getMember("
               encrypt").getAValueReachableFromSource()
13         or node = API::moduleImport("passlib").getMember("handlers").
               getMember("bcrypt").getMember("bcrypt").getMember("hash").
               getAValueReachableFromSource()
14         or node = API::moduleImport("passlib").getMember("handlers").
               getMember("bcrypt").getMember("bcrypt").getMember("using").
               getReturn().getMember("hash").getAValueReachableFromSource()
15         or node = API::moduleImport("passlib").getMember("handlers").
               getMember("bcrypt").getMember("bcrypt_sha256").getMember("hash")
               .getAValueReachableFromSource()
16         or node = API::moduleImport("passlib").getMember("handlers").
               getMember("bcrypt").getMember("bcrypt_sha256").getMember("using"
               ).getReturn().getMember("hash").getAValueReachableFromSource()
17         or node = API::moduleImport("passlib").getMember("handlers").
               getMember("bcrypt").getMember("bcrypt").getMember("encrypt").
               getAValueReachableFromSource()
18         or node = API::moduleImport("passlib").getMember("handlers").
               getMember("bcrypt").getMember("bcrypt").getMember("using").
               getReturn().getMember("encrypt").getAValueReachableFromSource()
19         or node = API::moduleImport("passlib").getMember("handlers").
               getMember("bcrypt").getMember("bcrypt_sha256").getMember("
               encrypt").getAValueReachableFromSource()
20         or node = API::moduleImport("passlib").getMember("handlers").
               getMember("bcrypt").getMember("bcrypt_sha256").getMember("using"
               ).getReturn().getMember("encrypt").getAValueReachableFromSource
               ())
21     and exists(node.asCfgNode())
22     and not node.asExpr() instanceof ImportMember
23 select node, node.getLocation(), "PassLib's bcrypt hasher is being used"
```

Figure A.53: CodeQL query that checks whether PassLib's bcrypt function is being used to hash passwords.

```
import python
import semmle.python.ApiGraphs
import CodeQL_Library.Passlib

bindingset[rounds, keyword, pos]
predicate workFactor(API::Node node, int rounds, string keyword, int pos) {
    exists(DataFlow::Node param |
        (param = node.getKeywordParameter(keyword).getAValueReachingSink()
            or param = node.getParameter(pos).getAValueReachingSink())
        and param.asExpr().(IntegerLiteral).getValue() < rounds)
    or not exists(DataFlow::Node param |
        (param = node.getKeywordParameter(keyword).getAValueReachingSink()
            or param = node.getParameter(pos).getAValueReachingSink()))
}

from API::Node node
where (node = PassLib::getCustomUsingNode("pbkdf2_sha256")
        and (not workFactor(node, 600000, "rounds", 6) // owasp
            recommendation minimum
            or not workFactor(node, 600000, "min_desired_rounds", 0) //
                owasp recommendation minimum
            or not workFactor(node, 600000, "min_rounds", 4) // owasp
                recommendation minimum
            or not workFactor(node, 600000, "default_rounds", 2)) // owasp
                recommendation minimum
        and not exists(PassLib::getDefaultUsageNode("pbkdf2_sha256")))
    or (node = PassLib::getCustomUsingNode("pbkdf2_sha512")
        and (not workFactor(node, 210000, "rounds", 6) // owasp
            recommendation minimum
            or not workFactor(node, 210000, "min_desired_rounds", 0) //
                owasp recommendation minimum
            or not workFactor(node, 210000, "min_rounds", 4) // owasp
                recommendation minimum
            or not workFactor(node, 210000, "default_rounds", 2)) // owasp
                recommendation minimum
        and not exists(PassLib::getDefaultUsageNode("pbkdf2_sha512")))
select node.getReturn().getMember("hash").getAValueReachableFromSource(),
    node.getReturn().getMember("hash").getAValueReachableFromSource().
    getLocation(), "PassLib is being used with pbkdf2 and it's compliant
    with owasp guidelines"
```

Figure A.54: CodeQL query that checks whether PassLib's PBKDF2 function is being used in an OWASP compliant configuration to hash passwords.

```
1 import python
2 import semmle.python.ApiGraphs
3
4 from DataFlow::Node node
5 where (node = API::moduleImport("passlib").getMember("hash").getMember("
      pbkdf2_sha1").getMember("hash").getAValueReachableFromSource()
6         or node = API::moduleImport("passlib").getMember("hash").getMember(
              "pbkdf2_sha1").getMember("using").getReturn().getMember("hash").
              getAValueReachableFromSource()
7         or node = API::moduleImport("passlib").getMember("hash").getMember(
              "pbkdf2_sha1").getMember("encrypt").getAValueReachableFromSource
              ()
8         or node = API::moduleImport("passlib").getMember("hash").getMember(
              "pbkdf2_sha1").getMember("using").getReturn().getMember("encrypt
              ").getAValueReachableFromSource()
9         or node = API::moduleImport("passlib").getMember("handlers").
              getMember("pbkdf2").getMember("pbkdf2_sha1").getMember("hash").
              getAValueReachableFromSource()
10        or node = API::moduleImport("passlib").getMember("handlers").
              getMember("pbkdf2").getMember("pbkdf2_sha1").getMember("using").
              getReturn().getMember("hash").getAValueReachableFromSource()
11        or node = API::moduleImport("passlib").getMember("handlers").
              getMember("pbkdf2").getMember("pbkdf2_sha1").getMember("encrypt"
              ).getAValueReachableFromSource()
12        or node = API::moduleImport("passlib").getMember("handlers").
              getMember("pbkdf2").getMember("pbkdf2_sha1").getMember("using").
              getReturn().getMember("encrypt").getAValueReachableFromSource())
13    and exists(node.asCfgNode())
14    and not node.asExpr() instanceof ImportMember
15 select node, node.getLocation(), "PassLib's pbkdf2 hasher is being used"
```

Figure A.55: CodeQL query that checks whether PassLib's PBKDF2 function is being used to hash passwords. Only the PBKDF2 function that uses SHA1 as an HMAC is shown here, as it is the same for SHA256 and SHA512.

```
1  import python
2  import semmle.python.ApiGraphs
3  import CodeQL_Library.Passlib
4
5  predicate memoryConfiguration(API::Node node) {
6      exists(DataFlow::Node param |
7          param = node.getKeywordParameter("rounds").getAValueReachingSink()
8          and param.asExpr().(IntegerLiteral).getValue() < 17) // owasp
               recommendation minimum
9      or not exists(DataFlow::Node param |
10         param = node.getKeywordParameter("rounds").getAValueReachingSink())
11 }
12
13 predicate blockSize(API::Node node) {
14     exists(DataFlow::Node param |
15         (param = node.getParameter(0).getAValueReachingSink()
16             or param = node.getKeywordParameter("block_size").
                   getAValueReachingSink())
17         and param.asExpr().(IntegerLiteral).getValue() < 8) // owasp
               recommendation minimum
18 }
19
20 predicate degreeOfParallelism(API::Node node) {
21     exists(DataFlow::Node param |
22         param = node.getKeywordParameter("parallelism").
               getAValueReachingSink()
23         and param.asExpr().(IntegerLiteral).getValue() < 1) // owasp
               recommendation minimum
24 }
25
26 from API::Node node
27 where node = PassLib::getCustomUsingNode("scrypt")
28     and not memoryConfiguration(node)
29     and not blockSize(node)
30     and not degreeOfParallelism(node)
31     and not exists(PassLib::getDefaultUsageNode("scrypt"))
32 select node.getReturn().getMember("hash").getAValueReachableFromSource(),
       node.getReturn().getMember("hash").getAValueReachableFromSource().
       getLocation(), "PassLib is being used with scrypt and it's compliant
       with owasp guidelines"
```

Figure A.56: CodeQL query that checks whether PassLib's scrypt function is being used in an OWASP compliant configuration to hash passwords.

```
1 import python
2 import semmle.python.ApiGraphs
3
4 from DataFlow::Node node
5 where (node = API::moduleImport("passlib").getMember("hash").getMember("
     scrypt").getMember("hash").getAValueReachableFromSource()
6          or node = API::moduleImport("passlib").getMember("hash").getMember(
             "scrypt").getMember("using").getReturn().getMember("hash").
             getAValueReachableFromSource()
7          or node = API::moduleImport("passlib").getMember("hash").getMember(
             "scrypt").getMember("encrypt").getAValueReachableFromSource()
8          or node = API::moduleImport("passlib").getMember("hash").getMember(
             "scrypt").getMember("using").getReturn().getMember("encrypt").
             getAValueReachableFromSource()
9          or node = API::moduleImport("passlib").getMember("handlers").
             getMember("scrypt").getMember("scrypt").getMember("hash").
             getAValueReachableFromSource()
10         or node = API::moduleImport("passlib").getMember("handlers").
             getMember("scrypt").getMember("scrypt").getMember("using").
             getReturn().getMember("hash").getAValueReachableFromSource()
11         or node = API::moduleImport("passlib").getMember("handlers").
             getMember("scrypt").getMember("scrypt").getMember("encrypt").
             getAValueReachableFromSource()
12         or node = API::moduleImport("passlib").getMember("handlers").
             getMember("scrypt").getMember("scrypt").getMember("using").
             getReturn().getMember("encrypt").getAValueReachableFromSource())
13     and exists(node.asCfgNode())
14     and not node.asExpr() instanceof ImportMember
15 select node, node.getLocation(), "PassLib's scrypt hasher is being used"
```

Figure A.57: CodeQL query that checks whether PassLib's scrypt function is being used to hash passwords.

```
1 import python
2 import semmle.python.ApiGraphs
3
4 from DataFlow::Node node
5 where node = API::moduleImport("werkzeug").getMember("security").getMember(
     "generate_password_hash").getAValueReachableFromSource()
6     and exists(node.asCfgNode())
7     and not node.asExpr() instanceof ImportMember
8 select node, node.getLocation(), "Werkzeug is being used"
```

Figure A.58: CodeQL query that checks whether the Werkzeug library is being used to hash passwords. This is done by looking for invocations of Werkzeug's `generate_password_hash` function.

```
1 import python
2 import semmle.python.ApiGraphs
3
4 DataFlow::Node libraryIsUsed() {
5     exists(DataFlow::Node node |
6         (node = API::moduleImport("werkzeug").getMember("security").
              getMember("generate_password_hash").getKeywordParameter("method"
              ).getAValueReachingSink()
7               or node =API::moduleImport("werkzeug").getMember("security").
                  getMember("generate_password_hash").getParameter(1).
                  getAValueReachingSink())
8         and node.asExpr().(StrConst).getS().prefix(6) = "scrypt"
9         and result = node)
10 }
11
12 bindingset[n, r, p]
13 predicate isCompliant(int n, int r, int p) {
14     n >= 131072 and r >= 8 and p >= 1
15 }
16
17 predicate aux(DataFlow::Node node) {
18     if exists(node.asExpr().(StrConst).getS().splitAt(":", 0)) and exists(
           node.asExpr().(StrConst).getS().splitAt(":", 1).toInt()) and exists(
           node.asExpr().(StrConst).getS().splitAt(":", 2).toInt()) and exists(
           node.asExpr().(StrConst).getS().splitAt(":", 3).toInt())
19     then isCompliant(node.asExpr().(StrConst).getS().splitAt(":", 1).toInt
           (), node.asExpr().(StrConst).getS().splitAt(":", 2).toInt(), node.
           asExpr().(StrConst).getS().splitAt(":", 3).toInt())
20     else if exists(node.asExpr().(StrConst).getS().splitAt(":", 0)) and
           exists(node.asExpr().(StrConst).getS().splitAt(":", 1).toInt()) and
           not exists(node.asExpr().(StrConst).getS().splitAt(":", 2).toInt())
           and not exists(node.asExpr().(StrConst).getS().splitAt(":", 3).toInt
           ())
21         then isCompliant(node.asExpr().(StrConst).getS().splitAt(":", 1).
              toInt(), 8, 1)
22         else if exists(node.asExpr().(StrConst).getS().splitAt(":", 0)) and
               exists(node.asExpr().(StrConst).getS().splitAt(":", 1).toInt())
               and exists(node.asExpr().(StrConst).getS().splitAt(":", 2).
              toInt()) and not exists(node.asExpr().(StrConst).getS().splitAt(
              ":", 3).toInt())
23             then isCompliant(node.asExpr().(StrConst).getS().splitAt(":",
                  1).toInt(), node.asExpr().(StrConst).getS().splitAt(":", 2).
                  toInt(), 1)
24             else none()
25 }
26
27 from DataFlow::Node node
28 where node = libraryIsUsed()
29     and aux(node)
30     and not node.asCfgNode().isImportMember()
31     and exists(node.asCfgNode())
32 select node, node.getLocation(), "Werkzeug's scrypt hasher is being used
       and it's compliant with owasp guidelines"
```

Figure A.59: CodeQL query that checks whether Werkzeug's scrypt function is being used in an OWASP compliant configuration to hash passwords.

```
import python
import semmle.python.ApiGraphs

from ControlFlowNode node
where node = API::moduleImport("werkzeug").getMember("security").getMember(
    "generate_password_hash").getReturn().getAValueReachableFromSource().
    asCfgNode()
    and (exists(StrConst method |
            (method.getAFlowNode() = node.(CallNode).getArgByName("method")
                or method.getAFlowNode() = node.(CallNode).getArg(1))
            and method.getText().prefix(6) = "scrypt")
        or not exists(ControlFlowNode method |
            (method = node.(CallNode).getArgByName("method")
                or method = node.(CallNode).getArg(1))))
    and not node.isImportMember()
    and exists(CallNode cn | cn = node.(CallNode))
select node, node.getLocation(), "Werkzeug's scrypt hasher is being used"
```

Figure A.60: CodeQL query that checks whether Werkzeug's scrypt function is being used to hash passwords.

```
import python
import semmle.python.ApiGraphs

DataFlow::Node libraryIsUsed() {
    exists(DataFlow::Node node |
        (node = API::moduleImport("werkzeug").getMember("security").
            getMember("generate_password_hash").getKeywordParameter("method"
            ).getAValueReachingSink()
            or node = API::moduleImport("werkzeug").getMember("security").
                getMember("generate_password_hash").getParameter(1).
                getAValueReachingSink())
        and exists(node.asCfgNode())
        and node.asExpr().(StrConst).getS().prefix(6) = "pbkdf2"
        and result = node)
}

bindingset[method, iterations]
predicate isCompliant(string method, int iterations) {
    (method = "sha256"
        and iterations >= 600000)
    or (method = "sha512"
        and iterations >= 210000)
}

predicate aux(DataFlow::Node node) {
    if exists(node.asExpr().(StrConst).getS().splitAt(":", 1)) and exists(
        node.asExpr().(StrConst).getS().splitAt(":", 2).toInt())
    then isCompliant(node.asExpr().(StrConst).getS().splitAt(":", 1), node.
        asExpr().(StrConst).getS().splitAt(":", 2).toInt())
    else if exists(node.asExpr().(StrConst).getS().splitAt(":", 1))
        then node.asExpr().(StrConst).getS().splitAt(":", 1) = "sha256" or
            node.asExpr().(StrConst).getS().splitAt(":", 1) = "sha512"
        else any()
}

from DataFlow::Node node
where node = libraryIsUsed()
    and aux(node)
select node, node.getLocation(), "Werkzeug's pbkdf2 hasher is being used
    and it's compliant with owasp guidelines"
```

Figure A.61: CodeQL query that checks whether Werkzeug's PBKDF2 function is being used in an OWASP compliant configuration to hash passwords.

```
import python
import semmle.python.ApiGraphs

from DataFlow::Node node
where (node = API::moduleImport("werkzeug").getMember("security").getMember
    ("generate_password_hash").getKeywordParameter("method").
    getAValueReachingSink()
        or node = API::moduleImport("werkzeug").getMember("security").
            getMember("generate_password_hash").getParameter(1).
            getAValueReachingSink())
    and exists(node.asCfgNode())
    and node.asExpr().(StrConst).getS().prefix(6) = "pbkdf2"
select node, node.getLocation(), "Werkzeug's pbkdf2 hasher is being used"
```

Figure A.62: CodeQL query that checks whether Werkzeug's PBKDF2 function is being used to hash passwords.

Appendix B

Manual Analysis

We present here the more detailed results of the manual inspections done in Section 5.1 (Table B.1), Section 5.2 (Table B.2), Section 5.3 (Table B.3), Section 6.1 (Table B.4) and Section 6.2 (Table B.5).

Table B.1: Manual Analysis Secret Keys

Cryptographic Keys	**URLs**	**Result**
Flask hard-coded secret keys	poopak	Secret key hard-coded and it is too short
	cait	Secret key hard-coded and it is too short
	CollegeEventPortal	Secret key hard-coded and it is too short
	OpenAtlas	Secret key hard-coded, too short and called ”change me” or the likes
	atlanticwave-proto	Secret key hard-coded and left a comment saying it should be changed
	ActiveDriverDB	Secret key hard-coded, too short and called ”change me” or the likes
	MCGJ	Secret key hard-coded, too short and left a comment saying it should be changed
	vs-tabletop	Secret key hard-coded and it is too short
	Bmaps-Backend	Secret key hard-coded and it is too short
	ProjetoGSW	Secret key hard-coded and it is too short
Django hard-coded secret keys	IERT-Webapp	Secret key hard-coded and it is a long random string
	bounswe2022group5	Secret key hard-coded and it is a long random string
	loonflow	Secret key hard-coded and it is a long random string
	quantum-management	Secret key hard-coded and it is a long random string
	EasyTP	Secret key hard-coded and it is a long random string
	mit-tab	Secret key hard-coded and it is a long random string
	roo.me	Secret key hard-coded and it is a long random string
	graphite-web	Secret key hard-coded and it is too short
	TEQST_Backend	Secret key hard-coded and it is too short
	OtterBot	Secret key hard-coded and it is too short

Table B.2: Manual Analysis of CSRF

CSRF	URLs	Result
CSRF selectively deactivated	Lurnby	Disabled CSRF protection for API requests
	archivy	Disabled CSRF protection for API requests
	natlas	Disabled CSRF protection for API requests
	pybossa	Disabled CSRF protection for API requests
	wlnupdates	Disabled CSRF protection for API POST request
	ledger-web link 2	Ledger web application that disables CSRF protection for POST requests that handle sensitive operations such as transaction (this is a potentially vulnerable application)
	commerce-coordinator	Disabled CSRF protection for Webhook requests
	GreaterWMS	Disabled CSRF protection for some POST requests such as the register view (this is a potentially vulnerable application)
	Stocksera	Disabled CSRF protection for API GET request
	project_mono	Disabled CSRF protection for Webhook requests
	Archery	Disabled CSRF protection for error views (this is a common practice in Django because of the way CSRF protection is implemented using the middleware)
	bearblog	Disabled CSRF protection for some POST requests such as the forum post up-vote request (this is a potentially vulnerable application)
	exhibition-inference	Disabled CSRF protection for some POST requests and it has also been marked as potentially unsafe in a comment (this is a potentially vulnerable application)
	InvenTree	Disabled CSRF protection for Webhook requests
	FuzzManager	Disabled CSRF protection for API requests
CSRF selectively activated	NewsBlur link 2 Issues	Enabled CSRF protection only for certain views, though not for the register view, after an operator expressed concern in the github issues
	djblets	Enabling CSRF protection only for certain views
	dds_web	Extending the FlaskForm class when creating certain forms (e.g. register form)
	CollegeEventPortal	Extending the FlaskForm class when creating certain forms (e.g. register form)
	Vessel-app	Extending the FlaskForm class when creating certain forms (e.g. register form)
	fstack-forum	Extending the FlaskForm class when creating certain forms (e.g. register form)
	Fashion-Store	Extending the FlaskForm class when creating certain forms (e.g. register form)
	Misago	Enabling CSRF protection only for certain views
	reviewboard	Enabling CSRF protection only for certain views
	ESSArch	Enabling CSRF protection only for certain views
CSRF always deactivated	gitlab-tools	CSRF protection is not activated, despite using a library supported in our analysis
	mediatum	False positive, seems to be implementing custom CSRF protection
	pyweekorg	CSRF protection is not activated, despite using a library supported in our analysis
	workbench	False positive, this is a very challenging and uncommon scenario to catch using CodeQL
	OnlineMooc	CSRF protection is not activated, despite using a library supported in our analysis

Table B.2: Manual Analysis of CSRF

CSRF	URLs	Result
	uno-cpi	CSRF protection is not activated, despite using a library supported in our analysis
	geopuzzle	CSRF protection is not activated, despite using a library supported in our analysis
	roundware-server	CSRF protection is not activated, despite using a library supported in our analysis
	lotus	CSRF protection is not activated, despite using a library supported in our analysis.
	camomilla	CSRF protection is not activated, despite using a library supported in our analysis

Table B.3: Manual Analysis Session Protection

Session Protection	**URLs**	**Result**
Session Protection disabled	notify	Session protection set to None
	notifications-admin commit	Session protection set to None to fix an issue where the users were being constantly logged out of the application
	zwift-offline	Session protection set to None
	notification-admin commit	Session protection set to None to fix an issue where the users were being constantly logged out of the application
Session Protection basic	evesrp	Session protection set to basic and the decorator `@fresh_login_required` is used
	site-secomp	Session protection set to basic and the decorator `@fresh_login_required` is used
Session Protection strong	backend	Session protection set to strong
	walle-web	Session protection set to strong
	Things-Organizer	Session protection set to strong
	Ignite	Session protection set to strong
	misp-dashboard	Session protection set to strong
	next-gen-scholars	Session protection set to strong
	decider	Session protection set to strong
	tipvote_webapp	Session protection set to strong
	scoringengine	Session protection set to strong
	OpenOversight	Session protection set to strong

Table B.4: Manual Analysis of Password Policies

Password Policies	URLs	Result
No password validation	forme-app	Not performing password strength validation
	OpenAtlas	Not performing password strength validation
	access-control-web	Not performing password strength validation
	Things-Organizer	Not performing password strength validation
	Portfolio-CareerLink	Not performing password strength validation
	waveform-annotation	Not performing password strength validation
	ISS	Not performing password strength validation
	Sefaria-Project	Not performing password strength validation
	anytask	Not performing password strength validation
	aoe2map	Not performing password strength validation
Using custom validators	jorvik	Using custom validators to implement maximum password length check
	TeamGroove	Using custom validators to implement maximum password length check
	concrete-datastore	Using custom validators to implement a password regexp check: the password must have at least a certain amount of lowercase characters, uppercase characters, special characters and digits
	seed	Using custom validators to implement a password regexp check: the password must have at least a certain amount of lowercase characters, uppercase characters and digits
	shared-music	Using custom validators to implement a password regexp check: the password must only contain A-Z, a-z, 0-9 or special symbols
	tamato	Using custom validators to implement a password regexp check: the password contains at least 1 uppercase character, 1 lowercase character, 1 digit and a special character.
	money-to-prisoners-api	Overrides the default validators just to change the default validation error message
	bluebutton-web-server	Password reuse validator: it checks that the password has not expired (so password have an expiration date) and that new passwords are different from passwords that have been used in the past (by that user)
	arches	Using custom validators to implement a password regexp check: the password must have at least one lowercase characters, one uppercase characters, a special characters and a digit
	110-1_Database-System_Project	Using custom validators to implement a password regexp check: the password must have at least one character and a digit
	ThisIGet	Using custom validators to implement a minimum length check: the password must be at least 8 characters long
	walle-web	Using custom validators to implement a regexp check
	MatBoy	Using custom validators to implement length checks: the password must be at least 8 characters long and a maximum of 50 characters
	meetings-registration-tool	Using custom validators to check that password and confirm password match

Table B.4: Manual Analysis of Password Policies

Password Policies	**URLs**	**Result**
	pybossa	Using custom validators to implement password regexp checks and length checks: the password must have at least one lowercase characters, one uppercase characters, a special characters, a digit and be in between 8 and 15 characters in length
Using regexp validator	tipvote_webapp	Using the regexp validator to check that the password does not have any special characters or spaces
	Parfumier	Using the regexp validator to check that the password has at least a letter, a number and a special character

Table B.5: Manual Analysis of Password Hashing (only 10 results are shown for ”unsafe algorithm” as it is a representative subset of the 25 that were found, since they were all the same)

Password Hashing	**URLs**	**Result**
No password hashing	portfolio	Not performing password hashing
	geonode	Using a Django hashing algorithm which does not exist
	Compiler-2020	Not performing password hashing
	destiny_focus	Not performing password hashing
	alarmdecoder-webapp	False positive
Unsafe algorithm	hue	Using MD5 for testing (false positive)
	matorral	Using MD5 for testing (false positive)
	platform	Using MD5 for testing (false positive)
	ezeeai	Using sha256
	BikenWeb	Using sha256
	bio-gen-calc	Using sha256
	Fashion-Store	Using sha256
	Team-44_Chunk-file	Using sha256
	InteractiveQA	Using sha256
	glider-dac	Using sha512