



Assignment of bachelor's thesis

Title:	Security Analysis of Data Stewardship Wizard Project
Student:	Konstantin Shadakh
Supervisor:	Ing. Marek Suchánek, Ph.D
Study program:	Informatics
Branch / specialization:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2024/2025

Instructions

The aim of this bachelor thesis is to conduct a comprehensive security analysis of the Data Stewardship Wizard (DSW), an open-source platform designed to support data stewards in the research data management planning process. The analysis will focus on identifying potential security risks and vulnerabilities within the system, as well as evaluating the effectiveness of the existing security measures in place. Additionally, the thesis will propose recommendations and strategies for enhancing the overall security posture of the DSW platform, in order to ensure the confidentiality, integrity, and availability of sensitive research data.

- Familiarize yourself with DSW, describe its architecture, components, features, and technical background.
- Analyze the security measures that are already in place such as static analysis, (automatic) dependency checks, or hashing/encryption algorithms used.
- Specify recommendations in terms of technologies and procedures to improve security of DSW.
- Determine which DSW features across its components could be vulnerable to relevant well-known issues and attacks (such as XSS, SQL injection, or STTI) and perform the penetration testing in a monitored environment.
- Evaluate the results of the tests performed and compose a list of implementation recommendations as a prioritized list of tasks.



**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

Bachelor's thesis

Security Analysis of Data Stewardship Wizard Project

Konstantin Shadakh

Department of Information Security
Supervisor: Ing. Marek Suchánek, Ph.D.

January 6, 2024

Acknowledgements

I want to start by expressing my deepest thanks to my supervisor, Ing. Marek Suchánek, Ph.D. His guidance and support have been invaluable throughout this thesis journey. His expertise and regular, insightful feedback have been crucial in shaping my research. I'm especially grateful for his dedication in guiding me through the intricacies of the application and the structure of the thesis, which played a significant role in the success of my work.

I'm also incredibly grateful to Bc. Aleksei Kravtsov for sharing his extensive knowledge and experience in security analysis and penetration testing. His guidance was key in helping me understand these complex areas and conduct a detailed examination of the Data Stewardship Wizard's functionalities, greatly enhancing the quality of my analysis.

A big thank you goes to my parents for their constant support and encouragement throughout my academic journey. Their unwavering belief in me and their sacrifices have been fundamental to my success.

I also want to acknowledge the hard work and dedication I put into this thesis. It's been a journey of both personal and professional growth, and I'm proud of what I've achieved.

This thesis is a culmination of not just my efforts but also the collaborative support and inspiration from those who have guided me along this path.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on January 6, 2024

Czech Technical University in Prague

Faculty of Information Technology

© 2024 Konstantin Shadakh. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Shadakh, Konstantin. *Security Analysis of Data Stewardship Wizard Project*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024. Also available from: <http://site.example/thesis>.

Abstrakt

Tato práce přináší hloubkovou bezpečnostní analýzu Data Stewardship Wizard (DSW), open-source nástroje určeného pro efektivní plánování správy dat. Hlavním cílem této studie bylo určit a posoudit potenciální bezpečnostní zranitelnosti v nástroji DSW a také posoudit robustnost jeho současných bezpečnostních protokolů. Prostřednictvím kombinace ručního zkoumání, penetračního testování a technik skenování zranitelností byla odhalena řada bezpečnostních problémů. Patří mezi ně zranitelnosti související se slabými zásadami přihlašovacích údajů, zranitelnost systému vůči útokům hrubou silou a rizika spojená s injekcí šablon na straně serveru.

Výsledky výzkumu zdůrazňují potřebu zlepšit bezpečnostní opatření v rámci systému DSW, zejména pokud jde o správu hesel, procesy ověřování uživatelů a důsledné ověřování vstupů. K řešení těchto problémů práce poskytuje soubor cílených doporučení zaměřených na zlepšení bezpečnostního rámce systému DSW. Tyto návrhy mají nejen posílit obranu společnosti DSW, ale také nabídnout cenné poznatky pro širší oblast kybernetické bezpečnosti v rámci platform pro správu dat. Tato práce tak slouží jako významný příspěvek k probíhajícímu úsilí o posílení bezpečnosti nástrojů pro správu dat, jako je DSW.

Klíčová slova Data Stewardship Wizard, kybernetická bezpečnost, penetrační testování, hodnocení zranitelnosti, správa dat.

Abstract

This thesis offers an in-depth security analysis of the Data Stewardship Wizard (DSW), an open-source tool designed for efficient data management planning. The main goal of this study was to pinpoint and assess potential security vulnerabilities within DSW, as well as to gauge the robustness of its current security protocols. Through a blend of manual examination, penetration testing, and vulnerability scanning techniques, a range of security issues were discovered. These include vulnerabilities related to weak credential policies, the system's vulnerability to brute-force attacks, and risks associated with server-side template injection.

The research findings underscore the need for improved security measures within DSW, especially concerning password management, user authentication processes, and rigorous input validation. To address these issues, the thesis provides a set of targeted recommendations aimed at enhancing DSW's security framework. These suggestions are intended not only to fortify DSW's defences but also to offer valuable insights into the wider field of cybersecurity within data management platforms. This thesis thus serves as a significant contribution to the ongoing efforts to strengthen the security of data management tools like DSW.

Keywords Data Stewardship Wizard, Cybersecurity, Penetration Testing, Vulnerability Assessment, Data Management.

Contents

Introduction	1
1 Local Deployment for Analysis and Penetration Testing	3
2 DSW Analysis	5
2.1 General Ideas and Aims	5
2.2 FAIR Principles	5
2.2.1 Findable	6
2.2.2 Accessible	6
2.2.3 Interoperable	6
2.2.4 Reusable	6
2.3 Overall Properties, Ideas, What It Does and How It Is Done . .	7
2.3.1 Core Objectives and Features	7
2.4 Main Features for Different Roles	8
2.4.1 Anonymous	8
2.4.2 Researchers	9
2.4.3 Data Stewards	9
2.4.4 Administrators	10
2.5 Architecture	10
3 Relevant Vulnerabilities and Attacks	13
3.1 Terminology	13
3.2 SQL Injection	14
3.3 Brute-Force Attack	15
3.4 Cross-Site Scripting	16
3.5 Cross-Site Request Forgery	16
3.6 Broken Authentication	17
3.7 Broken Access Control	18
3.8 Token Leakage	19
3.9 File Inclusion Vulnerabilities	19
3.10 Server-Side Template Injection	20
3.11 Vulnerabilities Introduced in Programming Languages and Li- braries	21
3.12 Conclusion	21

4	Current security measures	23
4.1	Authentication	23
4.1.1	Create a New User	23
4.1.2	Login	28
4.2	Authorization	30
4.3	Data Validation	33
4.3.1	File Upload Validation	34
4.3.2	Client-Side Validation	36
4.3.3	Server-Side Validation	37
4.4	Error Handling	38
4.5	Encryption	39
4.6	Hashing	40
4.7	Session Management	41
4.8	Logging and Monitoring	41
4.9	Security Checks	42
5	Technology recommendations	43
5.1	Login Process	43
5.2	Cryptography	43
5.3	Authorization and Logging	44
6	Vulnerabilities analysis	45
6.1	Penetration Testing Setup	45
6.2	Penetration Testing	46
6.2.1	Disclaimer for Penetration Testing Report	46
6.2.2	Brute-force Attack	47
6.2.3	SQL Injection	49
6.2.4	Cross-Site Scripting	52
6.2.5	Cross-Site Request Forgery	53
6.2.6	Broken Authentication	54
6.2.7	Broken Access Control	57
6.2.8	Token Leakage	60
6.2.9	Server-Side Template Injection	61
6.2.10	File Inclusion	62
7	Evaluate	65
7.1	The Common Vulnerability Scoring System (CVSS)	65
7.2	Summary	66
7.2.1	Weak Credential Policy - Password Strength	66
7.2.2	Password Brute-Force Attack	67
7.2.3	User Enumeration	67
7.2.4	Email Verification Denial of Service	68
7.2.5	Broken Access Control	68
7.2.6	Access Tokens in URL	68
7.2.7	Server-Side Template Injection	69
7.3	Prioritized List of Recommendations for DSW	69
7.3.1	High Severity Vulnerabilities	70
7.3.2	Medium Severity Vulnerabilities	70
7.3.3	Other Considerations	70

8 Conclusion	73
Bibliography	75
A Acronyms	77
B Contents of Electronic Attachment	79

List of Figures

2.1	Layout of the modules in Data Stewardship Wizard	12
4.1	Login form	23
4.2	Registration form	24
4.3	2FA authentication form	25
4.4	Filled registration form	25
4.5	Permissions for Admin	31
4.6	Permissions for Researcher	31
4.7	Permissions for Data Steward	31
6.1	Wordlist for the attack	49
6.2	The result of the attack	49
6.3	Setting up the Bulk Account Registration Attack using Burp Suite	55
6.4	Results of the Bulk Account Registration Attack	56

Introduction

In an era where data drives decisions, the sanctity and security of research data have never been more crucial. Researchers and data stewards pour countless hours into their work, ensuring that their findings can pave the way for new discoveries and innovations. The Data Stewardship Wizard (DSW) stands as a beacon in this landscape, offering a platform designed to streamline the research data management planning process. But with the digital realm's vast opportunities come challenges, especially when it concerns the safety of sensitive information.

Many tools and platforms promise efficient data management, but how many truly consider the myriad of threats lurking in the cyber shadows? The open-source nature of DSW is a double-edged sword. While it thrives on community contributions and the shared knowledge of many, it also stands exposed to those with less noble intentions.

This thesis is born out of a genuine concern and curiosity. How secure is DSW? Where might its defences falter, and how can I fortify them? Through a meticulous journey, I aim to dissect DSW's security measures, probe for vulnerabilities, and ultimately chart a path towards a more secure data stewardship experience.

This journey is structured into distinct chapters, each echoing a phase of my exploration. I begin by immersing in the world of DSW, understanding its heartbeat and mechanics in the analysis chapter. The design chapter sketches the blueprint of my investigative approach. As I delve deeper, the implementation chapter recounts my discoveries, the challenges faced, and the insights gained. In the testing chapter, I put DSW to the test, gauging its resilience and robustness against potential threats.

Local Deployment for Analysis and Penetration Testing

In this analysis, I have focused on the Data Stewardship Wizard version 3.28.0. This specific version was chosen as it represents the state of the application at a particular point in time, providing a stable reference for detailed examination and discussion. The features, functionalities, and security measures discussed throughout this document are based on the capabilities and characteristics of DSW as they existed in version 3.28.0. It's important to note that subsequent versions of DSW may include updates, enhancements, or changes that are not covered in this analysis.

This version of the application is deployed locally using Docker. This setup allows for a controlled environment to conduct thorough testing and analysis of the application, focusing specifically on the application itself rather than the deployment infrastructure. The docker image included several key services, each running in its own Docker container:

- `dsw-server`
- `dsw-client`
- `docworker`
- `mailer`
- `postgres` (the database used by DSW, running PostgreSQL version 15.5)
- `minio`(an object storage service used by DSW).

A more detailed description of the main services can be found in [Section 2.5](#).

DSW Analysis

“We present a tool, the “Data Stewardship Wizard”, that can bring together researchers, data stewards, and data experts pursuing better research through data management planning.” [25].

2.1 General Ideas and Aims

The description of the Data Stewardship Wizard application, encompassing its foundational principles, components, and various other aspects, is primarily sourced from the official DSW website at ds-wizard.org/. Additional insights and information have been gathered from its GitHub repository at github.com/ds-wizard, as well as through official videos available on YouTube. Furthermore, scholarly articles and publications by Robert Pergl, Marek Suchánek, and other contributors to the DSW project have significantly informed the content of this thesis.

The Data Stewardship Wizard is a tool for data management planning that focuses on maximizing the value of data management planning for the project itself rather than merely fulfilling obligations. It emphasizes the importance of data stewardship, which extends beyond the project’s duration and encompasses the long-term maintenance of the resulting research data. The term “Wizard” is used to denote the tool as an “expert system” that provides context-dependent guidance to its users.

The DSW aims to change the perception of data management planning from a burden to a benefit. It does this by highlighting the advantages of data management for the research project and the researcher, rather than focusing solely on obligations. For instance, the DSW points out appropriate tools that can assist in assembling and maintaining provenance metadata and relevant data standards. It also clearly indicates the impact of each answer on the adherence to the principles that data should be Findable, Accessible, Interoperable, and Reusable (FAIR).

2.2 FAIR Principles

The FAIR principles, an acronym for Findability, Accessibility, Interoperability, and Reusability, are essential guidelines for managing digital assets. These

principles are designed to ensure that data and metadata are easily manageable and usable by both humans and computational systems, with a focus on minimizing human intervention. This approach is increasingly vital due to the growing amount and complexity of data being generated. The description of the FAIR principles, along with their individual components, is derived from the article titled “Introducing the FAIR Principles for research software”, which was published on [nature.com](https://www.nature.com) [4] and the description of the FAIR principles on the official DSW web-page on ds-wizard.org/fair [8].

2.2.1 Findable

The essence of Findability is to ensure that both data and metadata are straightforward to locate for both human users and computer systems. This involves assigning unique and persistent identifiers to metadata and data, enriching data with comprehensive metadata, ensuring metadata accurately references the data it describes, and making sure that both data and metadata are indexed in a searchable resource.

2.2.2 Accessible

Accessibility requires that once data is located, the means to access it are clear, possibly encompassing authentication and authorization processes. This involves using standardized communication protocols that are open and universally implementable for retrieving metadata and data via their identifiers. These protocols may include measures for authentication and authorization if needed. Additionally, it’s important that metadata remains available even if the data itself is no longer accessible.

2.2.3 Interoperable

Interoperability focuses on the ability of data to integrate with other datasets and to function seamlessly within various applications or workflows. This is achieved by using a formal, accessible, shared, and broadly applicable language for knowledge representation in metadata and data. It also involves adhering to FAIR principles in the use of vocabularies and including qualified references to other metadata and data.

2.2.4 Reusable

The goal of Reusability is to maximize the potential for data to be reused in different contexts. This requires that both metadata and data are described in detail with multiple relevant attributes to facilitate replication or combination in various settings. Key aspects include releasing metadata and data with a clear data usage license, providing detailed provenance, and ensuring that they meet the standards of the relevant domain communities.

2.3 Overall Properties, Ideas, What It Does and How It Is Done

The Data Stewardship Wizard can be used as a pivotal tool in the realm of research data management, aiming to bridge the gap between data stewards, researchers, and data experts. Its inception and continual development are driven by a clear set of goals and a robust array of features, all meticulously designed to enhance the efficiency, accuracy, and overall quality of data management practices. This section delves into the core objectives that shape the DSW's functionality and the key features that empower users to navigate the complexities of data stewardship with ease and precision.

By exploring the main goals, I unravel the strategic vision behind DSW, understanding how it seeks to transform data management from a mandatory task into a value-adding activity. Concurrently, an examination of its features provides insight into the practical tools and resources that DSW offers, elucidating how it stands out as a comprehensive solution in the data stewardship landscape.

2.3.1 Core Objectives and Features

By gathering and summarizing information from the official Data Stewardship Wizard website, I can identify the four main objectives that DSW aims to achieve

- **Data Stewardship:** The discipline of data stewardship is intricate, demanding clear definition of data, establishment of processes and procedures, assurance of data quality, workflow optimization, and monitoring of data utilization to support teams, all while upholding data security and compliance standards. This responsibility falls on anyone engaged with digital data, spanning activities from collection and analysis to storage and usage.

“The ultimate goal is to provide high-quality data that is easily accessible in a consistent manner.” [28].

The DSW serves as a facilitator in the data stewardship process, simplifying the development of data management plans. To begin with, it provides a number of the most used templates for knowledge models (KM) and DMPs. Moreover, DSW allows users to create and manage their own DMP templates, which can be tailored to the specific needs of their research project or institution. It aids users by offering insights on effective data management practices, evaluating the FAIRness of their plans, and ensuring that the inquiries posed are pertinent to the specific project at hand.

- **Decision Support:** Creating a plan for managing data and ensuring the research data is of top-notch quality involves numerous factors that might seem overwhelming and challenging to initiate. In DSW, extensive writing is not necessary. You are prompted to respond to clear and straightforward questions in intelligent questionnaires, receive references to external materials, indications of FAIR metrics, and additional resources.

Depending on your responses, these questions may lead to further inquiries or direct you to external resources. This objective is implemented by the “Data Management Plans”.

A Data Management Plan (DMP) is a formal document that outlines how data will be handled both during and after a research project. It details the strategies for managing, storing, and securing data, ensuring that the data is well-organized and accessible. DMPs are increasingly becoming a required component of grant applications and are recognized as a best practice in conducting research.

DSW offers an extensive solution for perfect DMPs across various disciplines. The platform provides extensive guidance throughout the data stewardship journey, posing relevant questions, and offering tips, multimedia content, external resources, and community support. With permission from the Taylor & Francis Group, the DSW integrates insights and advice from “Data Stewardship for Open Science” by Barend Mons, directly into the platform.

- **Collaboration:** In today’s collaborative work environment, being part of a team is essential. DSW offers a variety of ways to share your project with others. You have the ability to collaborate in real-time, discuss questions through comments, and instantly observe modifications made by your teammates. It’s easy to track who has responded to which questions, and every alteration is meticulously recorded in the version history. This allows you to mark particular versions or revert to any previous state in the project’s history.
- **Researcher Goals:** Individuals conducting research with the aid of DSW are advised to carefully consider the numerous obstacles related to managing data throughout every phase of their study. The primary goal is to develop an all-encompassing strategy for managing data from the beginning to the end, resulting in the formulation of a DMP.
- **Integration:** DSW offers integration with external resources, simplifying the task of responding to queries in the surveys. This approach not only accelerates the procedure but also guarantees that the responses are consistently in accordance with the FAIR principles. A tangible illustration of this functionality is DSW’s linkage with FAIRsharing, providing users with direct access to its meticulously curated content.

2.4 Main Features for Different Roles

The Data Stewardship Wizard distinguishes itself in the field of data management by offering specialized tools tailored to different user roles. These roles include anonymous users, researchers, data stewards, and administrators.

2.4.1 Anonymous

Previously, users of DSW who were not logged in, referred to as anonymous users, had limited capabilities. They could register, log in, or recover a forgotten password via a confirmed email address. Additionally, they had access to a

Questionnaire Demo, available next to the Log In and Sign Up buttons in the navigation bar. This demo allowed users to try out questionnaires without the ability to save or export answers, serving as a learning tool. This functionality was always accessible in the public instance of DSW.

However, recent updates have expanded the functionalities available to anonymous users. With the introduction of Public Knowledge Models, anonymous users can now create projects and fill out questionnaires, a feature previously unavailable to them. These anonymous projects function similarly to other projects, with a public link set to edit permissions. Notably, if a logged-in user accesses such a project, they can claim ownership by clicking the “Add to my projects” button. Despite these enhancements, anonymous users still cannot create new documents; for such actions, registration and login are required.

2.4.2 Researchers

Researchers are at the forefront of scientific exploration, and DSW provides them with tools to manage their data effectively:

- **Questionnaire Completion:** Researchers can complete questionnaires to generate data management plans (DMPs) and other essential data stewardship documentation.
- **Collaboration:** They have the ability to collaborate with data stewards on various data management tasks.
- **Guidance:** DSW offers guidance on best practices for data management, ensuring researchers follow established protocols.
- **Progress Tracking:** Researchers can monitor their progress on data management tasks, ensuring timely completion.
- **Export Options:** The platform allows researchers to export their data management documentation in multiple formats and templates, catering to diverse needs.

2.4.3 Data Stewards

Data stewards play a pivotal role in ensuring data quality and adherence to best practices. DSW equips them with:

- **Knowledge Models:** They can create and manage knowledge models, serving as templates for questionnaires.
- **Questionnaire Configuration:** Data stewards can tailor knowledge models to the unique requirements of their institution or research community.
- **DMP Review:** They are responsible for reviewing and approving DMPs submitted by researchers, ensuring compliance and quality.
- **Template Management:** Data stewards can also upload document templates and edit the existing ones.

2.4.4 Administrators

Administrators ensure the smooth operation of the DSW platform and its alignment with institutional goals:

- **User Management:** Administrators can manage user accounts, set roles, and define permissions, ensuring the right access for every user.
- **DSW Configuration:** They have the tools to configure the DSW instance, tailoring it to the needs of their institution or research community.
- **Monitoring:** Administrators can keep an eye on the usage of DSW, ensuring optimal performance and addressing issues proactively.

With these features, the Data Stewardship Wizard ensures that every stakeholder, from researchers to administrators, has the tools they need for effective and efficient data management.

2.5 Architecture

The Data Stewardship Wizard system is modular, with different components handling distinct functionalities (see Figure 2.1). The main three of them are:

- **engine-frontend:** The front-end application is the user interface of DSW. It's written in Elm, a functional language that compiles JavaScript. The front-end is responsible for presenting the user with an intuitive and responsive interface, allowing them to interact with the system and manage their data stewardship activities.
- **engine-backend:** The backend application is the server part of DSW. It's written in Haskell, a statically typed, purely functional programming language. The backend handles various functionalities including User Management, Organization Management, Knowledge Model Management, Knowledge Model Editor, Migration Tools for obsolete Knowledge Models and Questionnaires, and a Data Management Plan Generator. It communicates with the front-end application and the tools components to ensure smooth operation of the system.
- **engine-tools:** These are additional components that support the functionality of DSW. They include:

Libraries:

- **Command Queue (dsw-command-queue):** This is a part of the engine-tools that handles the queue of commands or tasks that need to be executed. It ensures that all tasks are executed in the correct order and allows for efficient management of resources.
- **Config (dsw-config):** This tool is responsible for managing the configuration settings of DSW. It allows to configure optional DSW features including registration or configurable questionnaire visibility by users, external authentication using OpenID standard, set up information texts and dashboard shown in the client (before login, after login, etc.).

- **Database (dsw-database)**: This tool handles all database-related operations for DSW. It ensures that all data is stored and retrieved efficiently. The database tool is crucial for operations like user management, organization management, and knowledge model management.
- **Storage (dsw-storage)**: This tool manages the storage needs of DSW. It's responsible for storing and retrieving files and other data that are not stored in the database. For example, it could be used to store files uploaded by users or generated by DSW.

Utilities:

- **Template Development Kit (dsw-tdk)**: You can use these utilities to manage and develop templates for the Data Stewardship Wizard. Those tools include new, list, get, put, verify and package. For example, you can create a new template project, make edits, and then update the template in DSW with `dsw-tdk put`. Alternatively, you can create a distribution ZIP package that is importable via DSW web interface with `dsw-tdk package`.

Workers: These are components of the system that perform specific tasks. Each worker is designed to handle a particular type of operation or process within the DSW system.

- **Data Seeder (dsw-data-seeder)**: a worker that seeds data into the database (seeding is the process of initializing a random number generator with an initial value, called a seed. The seed is used to generate a sequence of random numbers that are not truly random but are instead pseudorandom).
- **Document Worker (dsw-document-worker)**: a worker that generates documents based on templates.
- **Mailer (dsw-mailer)**: a worker that sends emails.

All additional components are currently kept compatible with Python 3.10 and higher.

In this analysis, my primary focus will be on the engine-frontend, engine-backend, mailer (dsw-mailer), and document worker (dsw-document-worker) components of the Data Stewardship Wizard system. These components are selected for detailed examination because they constitute the main line of defence in the system's architecture and play a crucial role in ensuring the overall security and functionality of the platform.

While my analysis primarily focuses on these components, it is important to acknowledge the significance of the other components within the engine-tools suite. Components such as the Command Queue, Config, Database, and Storage, though not the main focus of this investigation, support the overall functionality and stability of the DSW system. They contribute to efficient task management, configuration settings, data storage and retrieval, and file management, respectively. Their roles, while more background in nature, are essential for the smooth operation of the system and indirectly contribute to its security posture.

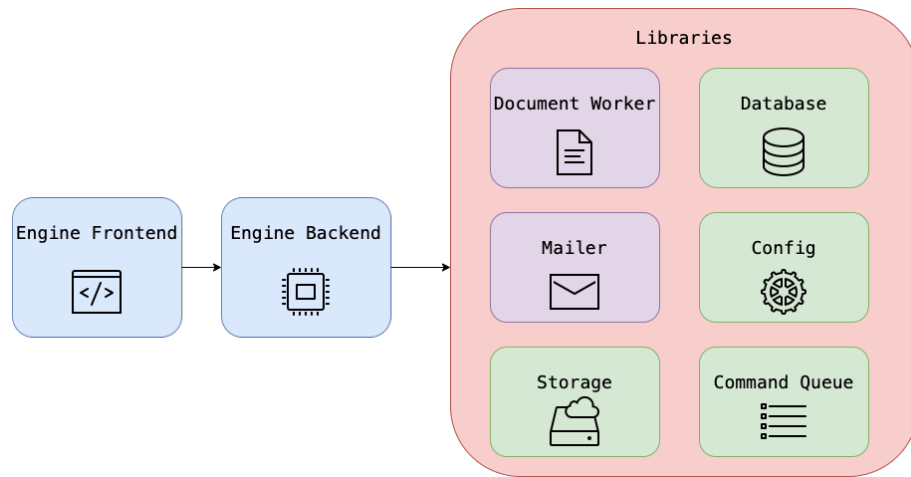


Figure 2.1: Layout of the modules in Data Stewardship Wizard

Relevant Vulnerabilities and Attacks

In today's fast-changing digital world, securing web applications and platforms is increasingly critical. As global interconnectivity rises, the variety and complexity of threats against these applications also escalate. Platforms like DSW are not exempt from these challenges. Recognizing potential vulnerabilities and the types of attacks that could exploit them is crucial in developing strong protective measures.

This chapter focuses on the typical vulnerabilities that web applications, including platforms like DSW, may encounter. It aims to shed light on the nature of these vulnerabilities, the ways in which attackers might exploit them, and the possible repercussions of such security breaches. Understanding these risks enables developers and administrators to better prepare and implement effective security strategies, thereby safeguarding the integrity, confidentiality, and availability of their platforms.

3.1 Terminology

Before delving into the specifics of various vulnerabilities and attacks, it's essential to familiarize yourself with some key terms and concepts. This will ensure a clear understanding of the subsequent sections and provide a foundation for grasping the intricacies of web application security.

- **Vulnerability:** A weakness in an IT system that can be exploited by an attacker to deliver a successful attack [5].
- **Exploit:** A piece of software, chunk of data, or sequence of commands that takes advantage of a vulnerability to cause unintended behavior in a system [14].
- **Attack:** An attempt to exploit a vulnerability to harm the system or its users.
- **Application Programming Interface (API):** A way used by applications to communicate with each other.

- **Token:** A piece of data that serves as a credential or identifier, often used for authentication and authorization purposes. There are three types of tokens: authentication tokens (to confirm a user's identity), API Token (authenticate requests to API), and CSRF tokens (unique tokens generated by the server and included in forms to prevent Cross-Site Request Forgery attack) [2].
- **Hypertext Transfer Protocol Secure (HTTPS):** An extension for the Hypertext Transfer Protocol that uses encryption for secure communication over a computer network [6].
- **Data Transfer Object (DTO):** These are objects that carry data between processes in order to reduce the number of methods calls [3].
- **Graphical User Interface (GUI):** A graphics-based operating system interface that uses icons, menus and a mouse (to click on the icon or pull down the menus) to manage interaction with the system [10].
- **The Open Worldwide Application Security Project (OWASP):** an online community that produces freely available articles, methodologies, documentation, tools, and technologies in the fields of IoT, system software and web application security [16].

Understanding these terms is crucial as they form the backbone of the discussion on web application vulnerabilities and attacks. As I proceed, I'll delve deeper into each vulnerability, exploring how they work, their potential impact, and the common attacks associated with them.

3.2 SQL Injection

Structured query language (SQL) is a programming language for storing and processing information in a relational database [1]. It is widely used in various applications due to its compatibility with multiple programming languages. It's a go-to tool for data analysts and developers, as it seamlessly integrates with programming languages, enabling the creation of efficient data processing applications that work with major SQL databases, including Oracle and PostgreSQL, which is used in the Data Stewardship Wizard.

Because SQL utilizes common English terms in its syntax, the queries, a message request to the database, are relatively straightforward to understand. For example, let's assume I have a table `user_data`, that contains information about users, including their identification number (ID), login names, passwords, and other data. If in the application I decided that I need to get a user's username based on the ID, I would use a query like this:

```
const auto userId = GetStringFromInput("Enter userId");
const auto userUsername = FetchSqlData(
    "SELECT user_name FROM user_data WHERE user_id = " + userId
);
```

Listing 1: Simplest SQL setup in C++

If `GetStringFromInput` returns a number, for example, 14, it will be used in the query like this:

```
SELECT user_name FROM user_data WHERE user_id = 14
```

Listing 2: SQL query when `user_id` is correct

As a result, I will get the `user_name` based on their ID. But what would happen if, instead of the correct number, the `GetStringFromInput` function returned 14 OR 1 = 1?

```
SELECT user_name FROM user_data WHERE user_id = 14 OR 1 = 1
```

Listing 3: SQL query when `user_id` is not correct (option 1)

In this case, I would get a list of all usernames stored in the table. And if I go even further and add `UNION` part to the query, I would be able to extract all the data from this and other tables:

```
SELECT user_name FROM user_data WHERE user_id = 14
UNION
SELECT password FROM user_data
UNION
SELECT tablename FROM pg_tables;
```

Listing 4: SQL query when `user_id` is not correct (option 2)

The result of this query is the list of all usernames, passwords, and the list of all tables in the database, if PostgreSQL is used.

SQL Injection attacks pose a significant threat, particularly when they lead to the compromise of sensitive data like personal details or financial records. To counter these attacks, web applications need to implement strong input validation and sanitization measures. This involves the use of prepared statements with parameterized queries, which effectively treat user input strictly as data, preventing it from being interpreted as part of the SQL command. Additionally, conducting regular security audits and tests, including penetration testing, is essential for detecting and addressing SQL Injection vulnerabilities.

3.3 Brute-Force Attack

A brute-force attack is an unsophisticated yet often effective method used by cybercriminals to gain unauthorized access to systems or data. This approach involves methodically trying every possible combination of passwords or phrases until the correct one is identified. The term “brute-force” reflects the reliance on relentless effort and power rather than subtlety or tactical approaches.

In such attacks, the perpetrator typically employs automated tools to generate a vast number of sequential guesses. These tools are used to crack user account passwords, decrypt data, or even discover hidden web pages. The key to this method’s simplicity is computational power – these tools can test thousands or millions of combinations in a short time.

Brute-force attacks are particularly effective against systems with lax password policies, like those allowing simple or commonly used passwords. The success rate of these attacks largely hinges on the password's complexity and length; more intricate and longer passwords significantly increase the number of required attempts, reducing the attack's feasibility.

To safeguard against brute-force attacks, implementing stringent password policies is crucial. These policies should encourage users to create complex passwords and update them regularly. Account lockout mechanisms, which block access after several failed login attempts, and two-factor authentication (2FA), which adds a second verification layer, are also effective. Monitoring login attempts and employing rate limiting to decelerate password attempts are additional strategies that can help mitigate brute-force attack risks.

3.4 Cross-Site Scripting

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites [18]. XSS attacks exploit the trust a user has for a particular site, allowing attackers to bypass access controls and potentially gain unauthorized access to sensitive data.

There are three main types of XSS attacks:

- **Reflected XSS:** This happens when an attacker tricks a user into clicking a seemingly legitimate link that contains a malicious script. Upon clicking, the script is sent to a vulnerable website, which then reflects the script back to the user's browser. The browser executes the script, mistaking it for a trusted source. For instance, an attacker could send a deceptive email with a link that, when clicked, executes a script to steal the user's browser cookies.
- **Stored XSS:** Here, the harmful script is permanently stored on the target server, such as in a comment section, forum, or database. When users access this stored content, the malicious script is activated. A typical example is a script embedded in a blog comment, which executes every time the comment is displayed.
- **DOM-based XSS:** This form of XSS attack arises from vulnerabilities in the client-side code rather than server-side code. It occurs when a web application's client-side script inappropriately writes user-supplied data to the Document Object Model (DOM) without proper sanitization. A common example is a URL that manipulates the page content using JavaScript based on URL parameters, thereby running the attacker's script.

3.5 Cross-Site Request Forgery

Cross-Site Request Forgery (CSRF) is a security vulnerability found in web applications. It enables attackers to trick users into executing unintended actions on a web application where they are authenticated. Unlike attacks aimed at data theft, CSRF targets requests that change the state of the application, as the attacker can't actually view the response to the forged request. The

definition and description of the CSRF are based on the “Cross site request forgery (CSRF) attack” on www.imperva.com [11].

To illustrate CSRF, consider this scenario:

- **Scenario:** A user is logged into their online banking account, a web application, and simultaneously visits a malicious website in another browser tab.
- **Attack Setup:** The malicious site contains a hidden form, automatically submitted via JavaScript upon page load. This form is designed to send a request to the banking application, such as transferring funds to the attacker’s account, with the form action pointing to the banking app’s URL for executing transfers.
- **Execution:** As the user visits the malicious page, the form gets submitted to the banking application. Since the user is already authenticated on the banking site, the application processes this request as if it were a legitimate action initiated by the user.
- **Outcome:** Without adequate CSRF protection, the banking application might carry out the action, unaware that the request originated from a malicious site. Consequently, the user’s account could be manipulated to perform actions they never intended.

To combat CSRF attacks, web applications employ anti-CSRF tokens. Unique to each user session, these tokens are embedded in forms and server requests. The server verifies the token with each request, ensuring it corresponds to the user’s session, thus validating the request’s authenticity and intent. If the token is absent or incorrect, the server rejects the request. This defence is effective because while a malicious site can generate requests to the target site, it cannot foresee or duplicate a user’s session-specific CSRF token.

3.6 Broken Authentication

Broken Authentication is a security vulnerability that occurs when a web application’s authentication and session management processes are implemented incorrectly, allowing attackers to compromise passwords, keys, session tokens, or exploit other implementation flaws to assume other users’ identities.

Based on OWASP Top Ten [22], this is one of the most common vulnerabilities out there. The definition of the attack along with its description is based on the related OWASP articles. There are three main scenarios of the attack:

- **Scenario #1 - Credential Stuffing Vulnerability:** Credential stuffing, involving the use of known password lists, is a prevalent attack method. If an application lacks automated defences against such threats or credential stuffing, it can inadvertently serve as a “password oracle”, helping attackers verify the validity of stolen credentials. This vulnerability arises when the application fails to detect and block repeated login attempts using different credentials, often sourced from data breaches.

- **Scenario #2 - Overreliance on Passwords and Outdated Practices:** Many authentication breaches stem from the reliance on passwords as the sole authentication factor. Practices once deemed secure, such as frequent password changes and complex requirements, are now seen as counterproductive. They often lead users to choose weaker, easily remembered (and often reused) passwords. Following NIST 800-63 guidelines, organizations are advised to move away from these outdated practices and adopt multi-factor authentication (MFA), which significantly enhances security by requiring additional verification methods beyond just a password.
- **Scenario #3 - Inadequate Session Timeout Management:** Proper session timeout settings are crucial for application security, especially when users access the application on public computers. Consider a user who accesses an app on a public computer but only closes the browser tab without logging out. If the application's session timeout isn't configured correctly, the session remains active. This oversight can allow an attacker to access the same browser later and find the user still authenticated, potentially leading to unauthorized access and data breaches. This scenario underscores the importance of setting appropriate session timeouts and educating users about the importance of actively logging out, especially in public settings.

3.7 Broken Access Control

Access control, also known as authorization, is a critical aspect of web application security. It determines which users can access specific content and functionalities within an application. This process occurs post-authentication and dictates the actions that authorized users can perform. While the concept of access control may seem straightforward, its proper implementation is often complex and challenging.

The access control model of a web application is intrinsically linked to the site's content and functionalities. Users typically fall into various groups or roles, each with distinct capabilities and privileges. However, developers often underestimate the complexity involved in creating a robust access control system. In many cases, these systems evolve organically alongside the website, leading to access control rules being scattered throughout the codebase. This ad-hoc approach can result in a convoluted set of rules that are difficult to manage, especially as the site approaches deployment.

Poorly designed access control systems are not only common but also relatively easy to exploit. According to OWASP Top Ten 2021 [20], this vulnerability is one of the easiest to exploit, and yet the 5th most popular attack on applications. Attackers can often gain unauthorized access simply by crafting requests for restricted content or functions. The repercussions of such breaches can be severe, ranging from unauthorized viewing or alteration of content to performing restricted functions or even hijacking site administration.

A particular concern in access control is the management of administrative interfaces. These interfaces are essential for site administrators to manage users, data, and content efficiently. Often, sites support various administrative roles to provide more detailed control over site management. However, due to

their extensive capabilities, these interfaces are prime targets for attacks, both from external sources and from within the organization. The security of these administrative portals is paramount, as they hold the keys to the entire site's functionality and data.

3.8 Token Leakage

Token leakage or theft is when an unauthorized party obtains or intercepts an OAuth token, either from the user, the client application, or the network [13]. There are many ways how this can happen:

- **Via URLs:** If tokens are included in URLs (for example, in the query string of a GET request), they can be leaked. URLs can be logged in server logs, browser history, or can be seen by third parties in unsecured network environments.
- **Referer Header:** When a user clicks a link on a webpage, the browser typically sends the URL of the current page as the “Referer” header to the destination server. If a token is in the URL, it can be leaked to the destination server.
- **Cross-Site Scripting:** If a web application is vulnerable to XSS, an attacker can inject malicious scripts to steal tokens stored in cookies or accessible via JavaScript.
- **Insecure Storage or Transmission:** Tokens stored insecurely on the client-side, or transmitted over unencrypted connections, can be intercepted and used by attackers.

To prevent token leakage, tokens should never be included in URLs, should be transmitted securely (e.g., over HTTPS), and should be stored securely on the client-side (e.g., in HTTPOnly cookies that are not accessible via JavaScript). Additionally, implementing robust security measures to protect against XSS and other injection attacks is very important.

3.9 File Inclusion Vulnerabilities

The File Inclusion vulnerability allows an attacker to include a file, usually exploiting a “dynamic file inclusion” mechanisms implemented in the target application. The vulnerability occurs due to the use of user-supplied input without proper validation [23]. Based on the related OWASP articles, there are two main types of this vulnerability:

- **Local File Inclusion (LFI):** This vulnerability arises when a web application permits the integration of files located on its own server into its web pages. Attackers can exploit LFI by altering input parameters to incorporate files stored on the server, such as configuration files, logs, or executable scripts. For instance, an LFI vulnerability might enable an attacker to include and run a PHP script located on the server, potentially leading to unauthorized access or control.

- **Remote File Inclusion (RFI):** RFI occurs when a web application allows the inclusion of files from external servers. In an RFI attack, the attacker manipulates the application to include and execute a script from a remote server. This can result in the execution of malicious code, controlled by the attacker, on the web server. RFI can be particularly dangerous as it gives attackers the ability to inject harmful scripts or programs into the web server, potentially compromising the server and affecting its operations.

As an example of this vulnerability, imagine a web application that dynamically includes content based on user input, such as:

```
include('pages/' . $_GET['page']);
```

Listing 5: PHP file inclusion

In this PHP code, the page to be included is determined by a user-supplied input (`$_GET['page']`). If this input is not properly sanitized, an attacker could supply a path to a file that should not be accessible. For example, something like this `http://page.com/index.php?page=../../../../etc/passwd` could print the contents of the `passwd` file.

To mitigate the risks, the developers must properly handle file path validation. This includes input validation, full path variables, correct application configuration, and proper error handling.

3.10 Server-Side Template Injection

Server-side template injection (SSTI) is a vulnerability found in web applications that utilize templating engines. These engines are tools that allow developers to create dynamic HTML pages. They work by incorporating special template tags or placeholders into HTML files, which are then substituted with real data when the page is loaded or rendered. SSTI occurs when user input is not adequately sanitized before being integrated into a template. This oversight can lead to situations where an attacker is able to inject harmful template code.

In such a scenario, if the user input is directly inserted into the template without proper checks, it can lead to the execution of unintended template commands or code. This could potentially allow attackers to manipulate the server-side processing of web pages, leading to a range of malicious activities. These activities might include data theft, website defacement, or even gaining unauthorized access to the server's backend systems. The risk with SSTI is particularly high because it directly affects the server-side execution environment, making it a critical vulnerability to address in web application security.

For example, consider a web application using a templating engine where a user's input is directly inserted into a template:

If the name parameter is not properly sanitized, an attacker could inject template code through the URL:

3.11. Vulnerabilities Introduced in Programming Languages and Libraries

```
@app.route('/')
def index():
    name = request.args.get('name', 'World')
    return render_template_string('Hello ' + name + '!')
```

Listing 6: Python code that writes input name to the template

```
http://example.com/?name={{ <malicious_code> }}
```

Listing 7: Example of exploiting SSTI

3.11 Vulnerabilities Introduced in Programming Languages and Libraries

Programming languages and their libraries are crucial in software development, but they can also be sources of vulnerabilities. Even with robust security measures in place for an application, vulnerabilities in the programming languages or libraries can still pose significant risks.

A notable example of such a vulnerability is the Heartbleed Bug, a severe flaw discovered in the widely-used OpenSSL cryptographic software library. Under normal circumstances, OpenSSL's SSL/TLS encryption is meant to provide secure communication over the Internet for various applications, including web browsing, email, instant messaging, and some VPNs. It's designed to ensure both security and privacy in these communications.

However, the Heartbleed bug exposed a critical weakness. It allowed anyone on the Internet to read the memory of systems protected by the affected versions of OpenSSL. This vulnerability compromised the secret keys used for service provider identification and traffic encryption, as well as user names, passwords, and actual content. The implications were severe: attackers could potentially eavesdrop on communications, directly steal data from services and users, and impersonate services and users. This bug highlighted the importance of not only securing the application but also ensuring the security of the underlying libraries and frameworks used in software development.

3.12 Conclusion

To test for these vulnerabilities, you can use automated tools known as vulnerability scanners. These tools scan web applications from the outside looking for security vulnerabilities such as XSS, SQL Injection, and others. They simulate known attack patterns and analyze the response of the website. However, it's important to note that these tools have their own strengths and weaknesses and should be used as part of a comprehensive security testing approach.

Current security measures

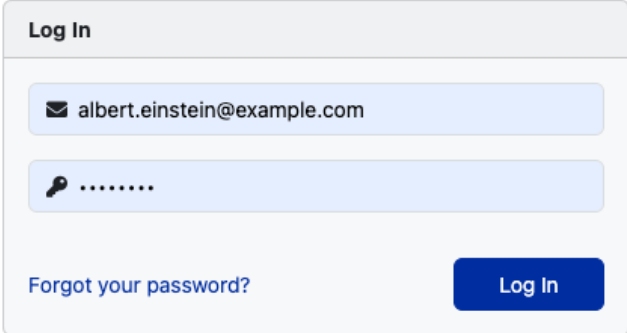
Ensuring the security and integrity of data is paramount, especially for platforms like the Data Stewardship Wizard that handle sensitive research data. Over the years, DSW has implemented a range of security measures to secure its platform and its users. This section provides an overview of the current security practices and tools in place.

4.1 Authentication

This is the process of verifying a user's identity. It includes checking whether a user is who they are claiming to be. Authentication is typically done by asking for a username and password, but it can also include other methods such as biometrics or Two-Factor Authentication.

4.1.1 Create a New User

Before delving into the code, it would be beneficial to illustrate what the authentication process looks like for an average DSW user. Figure 4.1 displays the main screen that users encounter when they wish to log in.



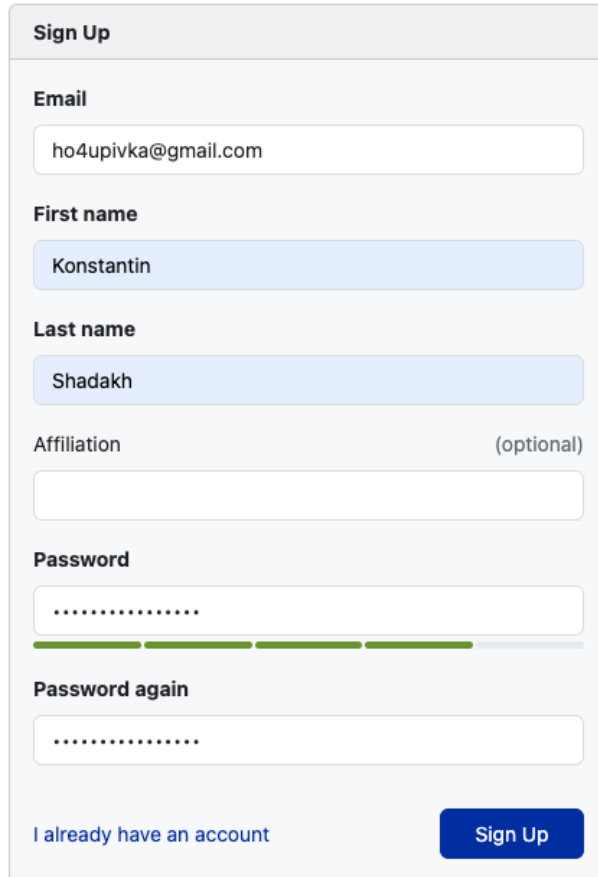
The image shows a login form with a light gray header containing the text "Log In". Below the header are two input fields. The first field has an envelope icon and contains the text "albert.einstein@example.com". The second field has a key icon and contains seven dots ".....". Below the input fields, there is a link "Forgot your password?" on the left and a blue button with the text "Log In" on the right.

Figure 4.1: Login form

If the user is not registered in the system, they are allowed to sign up in order to access anything except public knowledge models that allow anonymous

4. CURRENT SECURITY MEASURES

users. On the picture 4.2 you can find what the user sees on the registration page. However, the locally deployed version of the application does not support email verification, so the signup process could not be finished.



The image shows a registration form titled "Sign Up". It contains the following fields and elements:

- Email:** A text input field containing "ho4upivka@gmail.com".
- First name:** A text input field containing "Konstantin".
- Last name:** A text input field containing "Shadakh".
- Affiliation (optional):** An empty text input field.
- Password:** A password input field with a strength indicator below it showing approximately 75% completion (green bar).
- Password again:** A second password input field for confirmation.
- Navigation:** A link "I already have an account" and a blue "Sign Up" button.

Figure 4.2: Registration form

If desired, users have the option to enable two-factor authentication. However, this feature can only be activated by an administrator for all members of the project. It is found under Administration >Settings >Authorization >Two-Factor Authorization and is disabled by default. This setting allows the administrator to specify the code length and the expiration period for the one-time code, which is sent via email (as shown in Figure 4.3).

Two-Factor Authentication
If enabled, users first enter a username and password at login, and then they receive a one-time code to confirm the login on their email.

Code Length

Expiration

Expiration time of the authentication code in **seconds**.

Figure 4.3: 2FA authentication form

Also, there is another option for creating the user – an admin can create a user, assign their role, and set their password (it can be changed later by the user). In this case, the user will not need to verify the email, so the account will be activated immediately. The process of user creation by admin is shown in Figure 4.4.

Create user

Email

First name

Last name

Affiliation

Role

Password

Figure 4.4: Filled registration form

With the main processes outlined, let's delve into the code. As you are already aware, there are two methods to create a user, depending on the role of the current user (or the requestor). If the user is:

4. CURRENT SECURITY MEASURES

- an admin: `createUserByAdmin` is called.

```
createUserByAdmin :: UserCreateDTO ->
                  AppContextM UserDTO
createUserByAdmin reqDto =
  runInTransaction $ do
    checkPermission _UM_PERM
    uUuid <- liftIO generateUuid
    appUuid <- asks currentAppUuid
    clientUrl <- getAppClientUrl
    createUserByAdminWithUuid reqDto
                          uUuid
                          appUuid
                          clientUrl
                          False
```

Listing 8: `createUserByAdmin` function

Based on the code, the function starts by invoking `checkPermission _UM_PERM`, which checks if the current user has the necessary permissions to create a new user. If they do, the function generates Universally Unique Identifier (UUID) for the user being created. After retrieving the app UUID and Client URL, all the information and one boolean value is passed to `createUserByAdminWithUuid`, where the user gets created. The purpose of this boolean value is to indicate whether a registration email should be sent to the user.

```
createUserByAdminWithUuid :: UserCreateDTO -> U.UUID -> U.UUID
-> -> String -> Bool -> AppContextM UserDTO
createUserByAdminWithUuid reqDto uUuid appUuid clientUrl
-> shouldSendRegistrationEmail =
  runInTransaction $ do
    uPasswordHash <- generatePasswordHash reqDto.password
    serverConfig <- asks serverConfig
    appConfig <- getAppConfig
    let uRole = fromMaybe appConfig.authentication.defaultRole
        -> reqDto.uRole
    let uPermissions = getPermissionForRole serverConfig uRole
    userDto <- createUser reqDto uUuid uPasswordHash uRole
        -> uPermissions appUuid clientUrl
        -> shouldSendRegistrationEmail
    auditUserCreateByAdmin userDto
    return userDto
```

Listing 9: `createUserByAdminWithUuid` function

Here is what happens in `createUserByAdminWithUuid`:

- Password hashing: the user’s password from `reqDto` is hashed using `generatePasswordHash` function. That ensures that the actual password is not stored as a plain text in the database.

```

generatePasswordHash :: String -> AppContextM String
generatePasswordHash password = do
  hash <- liftIO $ BS.unpack <$> PasswordStore.makePasswordWith
    PasswordStore.pbkdf2 (BS.pack password) 17
  return $ "pbkdf2:" ++ hash

```

Listing 10: generatePasswordHash function

- Fetching configuration: the system fetches the server configuration using `serverConfig` and the app configuration using the `getAppConfig` function. These configurations contain settings and parameters required for user creation.
 - Determining user role: the user’s role can be defined in `reqDto.uRole`. But if it’s not, the default role from the app configuration is used `appConfig.authentication.defaultRole`
 - Determining user permissions: the user’s permissions are determined using the `getPermissionForRole` function, which maps roles to a set of permissions. It will be described in more detail in the next chapter.
 - User creation: create a new user in the system, check the user’s limits, validate the uniqueness of the email, prepare the user data, insert it into the database, create an action key for registration, send registration email and then return the `userDto`.
 - Auditing: `auditUserCreateByAdmin` function is called with the created user’s data `userDto`. This suggests that the system keeps an audit trail or log of user creation events, which can be useful for tracking and security purposes.
 - Returning: the created user’s data `userDto` is returned as the result of the function.
- **not** an admin: `registerUser` is called.

Here is what happens in `registerUser`:

- Registration check: check if the “registration” feature is enabled in `c.authentication.internal.registration.enabled`.
- UUID Generation: generates a new UUID for the user being registered. The `liftIO` function is used to lift an IO action (in this case, `generateUuid`) into the current monadic context `AppContextM`.
- Password hashing: the user’s password from `reqDto` is hashed using `generatePasswordHash` function.
- Fetching configuration: the system fetches the server configuration using `serverConfig` and the app configuration using the `getAppConfig` function. These configurations contain settings and parameters required for user creation.
- Determining user role: the user’s role can be defined in `reqDto.uRole`. But if it’s not, the default role from the app configuration is used `appConfig.authentication.defaultRole`

```
registerUser :: UserCreatedDTO -> AppContextM UserDTO
registerUser reqDto =
  runInTransaction $ do
    checkIfRegistrationIsEnabled
    uUuid <- liftIO generateUuid
    uPasswordHash <- generatePasswordHash reqDto.password
    serverConfig <- asks serverConfig
    appConfig <- getAppConfig
    let uRole = appConfig.authentication.defaultRole
        uPermissions = getPermissionForRole serverConfig
                                uRole

    clientUrl <- getClientUrl
    appUuid <- asks currentAppUuid
    createUser reqDto
              uUuid
              uPasswordHash
              uRole
              uPermissions
              appUuid
              clientUrl
              True
```

Listing 11: registerUser function

- Determining user permissions: the user’s permissions are determined using the `getPermissionForRole` function, which maps roles to a set of permissions.
- Fetching client URL: get the client’s URL.
- Fetching app UUID: using `getCurrentAppUuid`.
- User Creation: create a new user in the system, check the user’s limits, validate the uniqueness of the email, prepare the user data, insert it into the database, create an action key for registration, send a registration email and then return the `userDto`.

4.1.2 Login

To verify that the user’s credentials correspond to a specific person, a new `LoginDTO` (DTO means Data Transfer Object) with those credentials is created.

```
data LoginDTO = LoginDTO
  { email :: String
  , password :: String
  , code :: Maybe Int
  }
  deriving (Generic)
```

Listing 12: Definition of LoginDTO

Then this object is transferred to `LoginService.hs`, where DSW matches login credentials to a particular user using `createLoginTokenFromCredentials` function. To do that, the following steps are made:

- Find user by email: another function `findUserByEmail` is called. If the user is not found, it throws an error indicating an incorrect email or password. Otherwise, continues.
- Validation: if the user is found, the system validates the provided credentials against the stored ones using the `validate` function:

```
validate :: LoginDTO -> User -> AppContextM ()
validate reqDto user = do
  validateIsUserActive user
  validateUserPassword reqDto user
```

Listing 13: Definition of `validate` function

Based on the code, the application subsequently calls two functions – `validateIsUserActive` and `validateUserPassword`.

- `validateIsUserActive`: This function checks if the `active` attribute of the `User` is `True`. If the user is active (`active` is `True`), it does nothing and returns. If the user is not active (`active` is `False`), it throws an error indicating that the account is not activated.
- `validateUserPassword`: This function checks if the provided password (from `LoginDTO`) matches the hashed password of the `User` using `verifyPassword`. If no user is found, it throws an error indicating incorrect email or password. Otherwise, it uses the password hash, which is stored in `passwordHashFromDB` for further processing.

```
verifyPassword :: String -> String -> Bool
verifyPassword incomingPassword
  passwordHashFromDB =
case splitOn ":" passwordHashFromDB of
  ["pbkdf1", hashFromDB] ->
    PasswordStore.verifyPassword
      (BS.pack incomingPassword)
      (BS.pack hashFromDB)
  ["pbkdf2", hashFromDB] ->
    PasswordStore.verifyPasswordWith
      PasswordStore.pbkdf2
      (2 ^)
      (BS.pack incomingPassword)
      (BS.pack hashFromDB)
  _ -> False
```

Listing 14: Definition of `verifyPassword` function

Based on the code, the system stores not only the hash in the database but also a prefix indication of the hashing algorithm used,

followed by the actual hash. Based on the split, the system decides how the password matching is handled:

- * If the split results in two parts, and the first part is `pbkdf1`, it uses the `PasswordStore.verifyPassword` (a function from `Crypto.PasswordStore` library) function to check if the `incomingPassword` matches the `hashFromDB`. The `BS.pack` function is used to convert the `String` to a `ByteString`.
 - * If the split results in two parts, and the first part is `pbkdf2`, it uses a different verification function, `PasswordStore.verifyPasswordWith`, along with the `PasswordStore.pbkdf2` hashing algorithm. Again, the password and hash are converted to `ByteString` before verification.
 - * If the split doesn't match any of the first two formats, the function returns `False`, indicating, that the password verification failed.
- Two-Factor Authentication: the system checks if two-factor authentication (2FA) is enabled for the user.
 - if two-factor authentication is **not** enabled: it updates the user's last visited timestamp and creates a login token.
 - if two-factor authentication is enabled, but no code provided in `LoginDTO`: any existing action keys for the user are deleted. A new 2FA code is generated and stored as an action key in the database. The 2FA code is then sent to the user via email. The system returns a `CodeRequiredDTO`, indicating that the user needs to provide the 2FA code to complete the authentication.
 - if two-factor authentication is enabled and the code is provided: the system validates the provided code against the stored action key using the `validateCode` function. It checks the database to see if there's a matching `ActionKey` for the user's `UUID` and the provided code. If a match is found, the function checks if the code has expired. If the code is valid and hasn't expired, the function completes successfully. If the code is incorrect or has expired, appropriate errors are thrown.
 - JSON Web Token generation: once the user is authenticated (with or without 2FA), a JSON Web Token (JWT) is generated for the user. It contains claims such as `user.uuid`, `user.appUuid`, `serverConfig.jwt.expiration`. Then the token is signed, stored in the database as a user token and returned to the user, which they can use for subsequent authenticated requests.

4.2 Authorization

Authorization refers to the procedure of allowing or restricting access to a resource, depending on the authenticated user's rights. It entails verifying if a user possesses the required permissions to access a certain resource or

carry out a particular task. Typically, authorization goes hand in hand with authentication, ensuring that the server has an understanding of the identity of the client seeking access.

The system employs a function named `getPermissionForRole` to map each user to their respective permissions based on their role. This function acts as a gatekeeper, determining what actions a user is authorized to perform within the application. The roles are configured in `config.roles.*`, and each role comes pre-equipped with a specific set of permissions.

It is important to note that these permissions are set by default and are intrinsic to the roles themselves. This means that they are not directly modifiable on a per-user basis. In other words, a user's permissions are inherently tied to their role, and any change in permissions would necessitate a change in the user's role.

The application maintains a comprehensive list of permissions, each of which can be assigned to different roles. However, it is crucial to carefully consider which permissions are assigned to which roles to maintain the integrity and security of the system. Here is the list of all permissions introduced in the DSW:

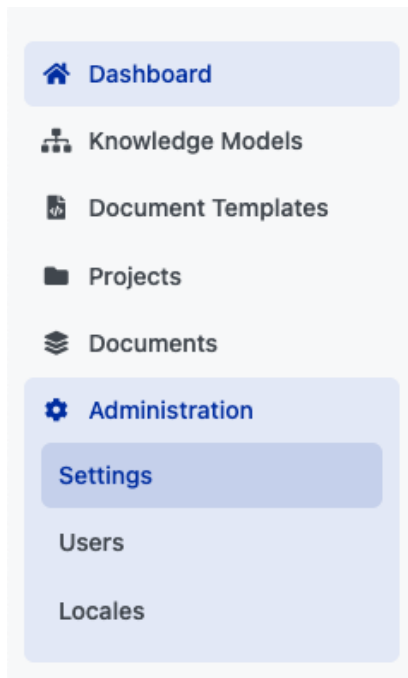


Figure 4.5: Permissions for Admin

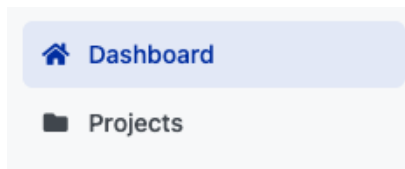


Figure 4.6: Permissions for Researcher

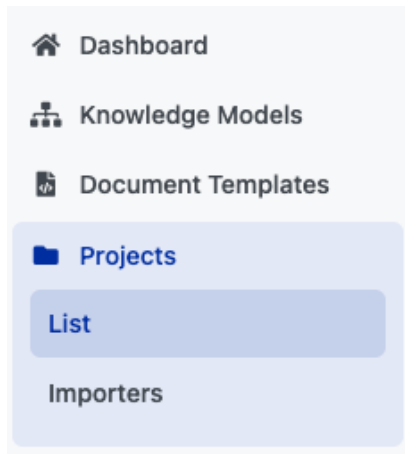


Figure 4.7: Permissions for Data Steward

- **_UM_PERM** - User Management Permission - used for various operations on users - create, read, update, delete (CRUD). Available for:

4. CURRENT SECURITY MEASURES

- `_USER_ROLE_ADMIN`
- **`_KM_PERM`** - allows CRUD operations on branches. Available for:
 - `_USER_ROLE_ADMIN`
 - `_USER_ROLE_DATA_STEWARD`
- **`_KM_PUBLISH_PERM`** - permission to publish packages. Available for:
 - `_USER_ROLE_ADMIN`
 - `_USER_ROLE_DATA_STEWARD`
- **`_KM_UPGRADE_PERM`** - allows CRUD operations on migrations. Available for:
 - `_USER_ROLE_ADMIN`
 - `_USER_ROLE_DATA_STEWARD`
- **`_PM_READ_PERM`** - permission to read or access package information. Available for:
 - `_USER_ROLE_ADMIN`
 - `_USER_ROLE_DATA_STEWARD`
 - `_USER_ROLE_RESEARCHER`
- **`_PM_WRITE_PERM`** - permission related to writing, modifying, importing, and deleting package information. Users must have this permission to import and convert package bundles, pull package bundles from a registry, modify package details, delete packages based on query parameters, and delete specific package. Available for:
 - `_USER_ROLE_ADMIN`
 - `_USER_ROLE_DATA_STEWARD`
- **`_QTN_PERM`** - allows CRUD operations on questionnaires. Available for:
 - `_USER_ROLE_ADMIN`
 - `_USER_ROLE_DATA_STEWARD`
 - `_USER_ROLE_RESEARCHER`
- **`_QTN_TML_PERM`** - permission to create a questionnaire with given UUID. Available for:
 - `_USER_ROLE_ADMIN`
 - `_USER_ROLE_DATA_STEWARD`
- **`_QTN_IMPORTER_PERM`** - permission to modify questionnaire importer. Available for:
 - `_USER_ROLE_ADMIN`

- `_USER_ROLE_DATA_STEWARD`
- **`_SUBM_PERM`** - permission to submit a document. Available for:
 - `_USER_ROLE_ADMIN`
 - `_USER_ROLE_DATA_STEWARD`
 - `_USER_ROLE_RESEARCHER`
- **`_DOC_TML_READ_PERM`** - permission to get document template and template suggestions. Available for:
 - `_USER_ROLE_ADMIN`
 - `_USER_ROLE_DATA_STEWARD`
 - `_USER_ROLE_RESEARCHER`
- **`_DOC_TML_WRITE_PERM`** - allows CRUD operation on templates, template assets, and drafts. Available for:
 - `_USER_ROLE_ADMIN`
 - `_USER_ROLE_DATA_STEWARD`
- **`_DOC_PERM`** - permission to read information about document. Available for:
 - `_USER_ROLE_ADMIN`
- **`_LOC_PERM`** - allows CRUD operations on locales. Available for:
 - `_USER_ROLE_ADMIN`
- **`_CFG_PERM`** - permission to modify client customization, get and modify application configuration, and delete and update application logo. Available for:
 - `_USER_ROLE_ADMIN`

The application creates records that chronologically catalogue system activities. By recording who performed an action, what action was performed, and when it was performed. They are essential for maintaining security, providing a way to track user actions, helping to detect and investigate unauthorised access or anomalies, and ensuring accountability within the application.

4.3 Data Validation

Input validation is performed to ensure only properly formed data is entering the workflow in an information system, preventing malformed data from persisting in the database and triggering malfunction of various downstream components. Input validation should happen as early as possible in the data flow, preferably as soon as the data is received from the external party. [19]

It's essential to apply input validation to data from all sources that might not be completely secure. Any of these sources could be compromised and might begin to transmit improperly formatted data. For a complete immersion

in this topic, familiarization with a very important resource of information – OWASP is necessary.

The Open Web Application Security Project (OWASP) is a worldwide non-profit dedicated to enhancing software security. It's a community-driven organization, with security professionals around the world contributing to its wealth of freely accessible resources, which include articles, methodologies, documentation, tools, and technologies focused on web application security.

Among the key resources provided by OWASP is the “Input Validation Cheat Sheet”. This resource is a detailed guide designed to help developers grasp the best practices for input validation. It details methods for effectively checking, filtering and cleaning data before it's processed by the system. Here are all the steps of input validation from the cheat sheet and their implementation in the Data Stewardship Wizard:

4.3.1 File Upload Validation

Many web applications, including the Data Stewardship Wizard, offer users the ability to upload various types of files, such as project templates, knowledge models, and locales. However, it's crucial for these files to undergo thorough verification and validation on the backend. This process includes checking the file format, ensuring uniqueness, and implementing other security measures. To accomplish this, DSW developers have implemented multiple functions. In this section, I will go through the most critical file validation function – template validation, implemented in `importAndConvertBundle`.

This is a Haskell function that takes two arguments: a `ByteString`, representing the content of a document template bundle, and a `Bool`, indicating whether the bundle originates from a registry or not. Initially, by using the `fromDocumentTemplateArchive` function, the file content is converted into a document template archive. This function returns an `Either` type, which can be either `Right` (indicating success) or `Left` (indicating an error). In the case of success, the system checks whether the number of document templates or the storage used exceeds a predefined limit using `findCurrentAppLimit` function. If no error occurs, the bundle is converted into a document template using the `fromBundle` function, which is then validated by utilizing the `validateNewDocumentTemplate` function.

The `validateNewDocumentTemplate` function is designed to perform a series of validation checks:

- `validateCoordinateFormat`: The function starts with splitting the `coordinate` part using “.” as a separator. This must result in three parts: identification number, organization ID, and template ID. The identification number and organization ID must not be empty. If either of these conditions is not met, the function throws an error indicating an invalid coordinate format `_ERROR_VALIDATION_INVALID_COORDINATE_FORMAT`. Otherwise, the function returns the result of matching the third part to the regular expression

```
validationRegex = mkRegex "[0-9]+\\. [0-9]+\\. [0-9]+$"
```

- `validateDocumentTemplateIdUniqueness`: Using function `findDocumentTemplateById` it checks whether the given document ID

```

importAndConvertBundle :: BSL.ByteString ->
                        Bool ->
                        AppContextM DocumentTemplateBundleDTO
importAndConvertBundle contentS fromRegistry =
  case fromDocumentTemplateArchive contentS of
    Right (bundle, assetContents) -> do
      checkDocumentTemplateLimit
      let assetSize =
          foldl (\acc (_, content) -> acc +
              (fromIntegral . BS.length $ content)) 0 assetContents
          checkStorageSize assetSize
          appUuid <- asks currentAppUuid
          let tml = fromBundle bundle appUuid
              validateNewDocumentTemplate tml True
              deleteOldDocumentTemplateIfPresent bundle
          traverse_ (\(a, content) ->
              putAsset tml.tId a.uuid a.contentType content)
              assetContents
          insertDocumentTemplate tml
          traverse_
            (insertFile . fromFileDTO tml.tId appUuid tml.createdAt)
              bundle.files
          traverse_
            ( \(assetDto, content) ->
              insertAsset $ fromAssetDTO tml.tId
                (fromIntegral . BS.length $ content)
                appUuid tml.createdAt assetDto
            )
              assetContents
      if fromRegistry
      then auditBundlePullFromRegistry tml.tId
      else auditBundleImportFromFile tml.tId
      return bundle
    Left error -> throwError error

```

Listing 15: Definition of importAndConvertBundle function

already exists in the database. If it is, it raises an error `_ERROR_VALIDATION_DOC_TML_ID_UNIQUENESS`.

- `validateCoordinateWithParams`: It extracts the identification number, organization ID, template ID, and version from the template, and tries to create a new template using those values using the template constructor by simple concatenation using “:” as a delimiter. The created string is then compared to the coordinate of the file. If they don’t match, an error is raised `_ERROR_VALIDATION_COORDINATE_MISMATCH`.
- `shouldValidateMetamodelVersion`: This part is optional, but in my case, it gets executed. It ensures that the template metamodel version is equal to a constant value stored in `documentTemplateMetamodelVersion`.

```
validateNewDocumentTemplate :: DocumentTemplate ->
                             Bool ->
                             AppContextM ()

validateNewDocumentTemplate tml
    shouldValidateMetamodelVersion = do
    validateCoordinateFormat False tml.tId
    validateDocumentTemplateIdUniqueness tml.tId
    validateCoordinateWithParams tml.tId
                                tml.organizationId
                                tml.templateId
                                tml.version
    when shouldValidateMetamodelVersion
        (validateMetamodelVersion tml)
```

Listing 16: Definition of validateNewDocumentTemplate function

If those values are different, the function throws an error
`ERROR_VALIDATION_TEMPLATE_UNSUPPORTED_METAMODEL_VERSION`.

After making sure that the template is valid, the `importAndConvertBundle` function checks whether there's an existing document template with the same ID, this function removes it along with its associated assets. This step is crucial for ensuring that the new template does not conflict with existing ones. After this step, the template, including its assets and files, records an audit log entry indicating the source of the bundle is stored.

4.3.2 Client-Side Validation

Client-side data validation is the first line of defence against incorrect or malicious user input. It takes place directly on the user's device, usually within a web browser, prior to the data being transmitted to the server. This form of validation offers quick feedback to the user, improving the overall experience by promptly identifying mistakes and minimizing the need for extensive server-side validations.

The developers have primarily utilized Elm to construct this segment of the application. For the purposes of validating and confirming data, they have used the `Validate` library. For example, let's take a look at the signup validation process:

```
signupFormValidation : Validation e SignupForm
signupFormValidation =
    Validate.map5 SignupForm
        (Validate.field "organizationId"
            (validateRegex "^(?![.])?!.*[.]$[a-zA-Z0-9.]+$"))
        (Validate.field "name" Validate.string)
        (Validate.field "email" Validate.email)
        (Validate.field "description" Validate.string)
        (Validate.field "accept" validateAcceptField)
```

Listing 17: Definition of signupFormValidation function

The `organizationId` field is validated using a regular expression, which is used in `validateRegex` function. Important to mention, that `validateRegex` is a wrapper over the `Validate` function that checks if a string matches a given regular expression pattern. Also, the regular expression does not seem to be correct, as it cannot start with `^^`.

Name, email, and description is validated using `Validate.string` function. To validate the `accept` field, a special function has been created – `validateAcceptField`, which basically checks if the user has checked a required checkbox (such as one to accept terms and conditions), before they can successfully submit the form.

4.3.3 Server-Side Validation

Server-side validation is an important aspect of creating secure and stable web applications. It differs from client-side validation, which is performed in the user's browser and is vulnerable to being overridden by the user. Server-side validation occurs on the web server once the data has been sent. This validation step is very important as it serves as the ultimate checkpoint, confirming that all incoming data matches the required format, type, and content before any processing or storage takes place.

The server-side validation offers a more secure and trustworthy method for data verification. It's especially vital for thwarting various attack vectors, such as SQL injection, cross-site scripting, and other forms of data manipulation that could endanger the application's integrity and the data it manages. Choosing the right programming language for the client-side is a critical decision. That's why Haskell was chosen.

Haskell, as a high-level, statically typed functional programming language, removes the need for developers to handle memory manipulation directly. It automatically oversees memory management and lacks the features for direct memory address access or manual memory management tasks that are typically found in lower-level languages. This design significantly reduces the risk of buffer overflows and other possible attacks:

- Memory Safety – the language handles memory allocation and garbage collection without exposing the low-level details to a programmer.
- Immutable Data – once the object is created, it cannot be modified.
- Strong Typing – the language ensures that functions receive inputs of the correct type and typically of a known size.
- Bounds Checking – when working with arrays, vectors or other similar data structures, Haskell checks at the run-time for the bounds.
- Avoidance of Null – Haskell does not have null values. Instead, it uses the “Maybe” type to explicitly handle the presence or absence of the value, which avoids null pointer exceptions.

In addition to the type checking inherent to Haskell, it's important to ensure that the data being saved to the database is of the correct type. For this purpose, the project utilizes the `Database.PostgreSQL.Simple` library, which offers type-safe methods to engage with a PostgreSQL database from Haskell.

However, it's important to note that this library is labelled as experimental in terms of stability. This designation implies that certain features might be unreliable and could potentially be exploited.

To verify that the data conforms to a specific format. This type of check is done mostly by using regular expressions, but basic formats such as phone numbers or emails are verified on the client side. For other things, such as files and projects, the `matchRegex` function from `Text.Regex` library is used.

4.4 Error Handling

Error handling is a critical aspect of software development, particularly in web applications where a variety of issues can occur due to network problems, user input errors, or internal server faults. Proper error handling ensures that the application can gracefully manage unexpected situations, providing informative feedback to the user, and preventing the exposure of sensitive system information that could be exploited by malicious actors.

Error handling in web applications is a vital mechanism designed to address unexpected conditions during run-time. It serves multiple purposes: it enhances user experience by providing meaningful error messages, it helps developers diagnose issues through logging, and it secures the application by preventing the leakage of sensitive information and system details that could be leveraged in cyber attacks.

Within the Data Stewardship Wizard, effective error handling is essential. Considering the delicate task of data management planning and the intricate interactions that can occur in the system, DSW is designed to adeptly manage a broad spectrum of possible errors.

```
data AppError
= AcceptedError
| MovedPermanentlyError String
| FoundError String
| ValidationError [LocaleRecord] (M.Map String [LocaleRecord])
| UserError LocaleRecord
| SystemLogError LocaleRecord
| UnauthorizedError LocaleRecord
| ForbiddenError LocaleRecord
| NotExistsError LocaleRecord
| LockedError
| GeneralServerError String
| HttpClientError Status String
deriving (Show, Eq)
```

Listing 18: Definition of `AppError`

- `AcceptedError` – represents a successful operation that is still processing (HTTP 202).
- `MovedPermanentlyError` – represents a resource that has been moved to a new URL (HTTP 301). The new URL provided as a string.

- **FoundError** – represents a resource that has been found but is temporarily located at a different URL (HTTP 302).
- **ValidationError** – represents an error due to validation failure, with details about the error provided in a list of `LocaleRecord` and a map from field names to lists of `LocaleRecord`.
- **UserError** – represents a generic error that can be shown to the user, with a localized message i.e `ERROR_SERVICE_COMMON_FEATURE_IS_DISABLED`.
- **SystemLogError** – represents an error that should be logged by the system.
- **UnauthorizedError** – Represents an authentication error where the user is not authorized (HTTP 401).
- **ForbiddenError** – represents an authorization error where the user is forbidden from accessing a resource (HTTP 403).
- **NotExistsError** – represents an error where a resource does not exist (HTTP 404).
- **LockedError** – represents a resource that is locked and cannot be accessed (HTTP 423).
- **GeneralServerError** – represents a generic server error, with a message as a `String`.
- **HttpClientError** – represents an error related to an HTTP client operation, with a status indicating the HTTP status code and a `String` for an additional message.

The complete list of all errors can be found in `Public.hs` files.

As previously noted, the client-side component of the application is developed in Elm, a language known for its absence of exceptions. Elm is renowned for its promise of eliminating run-time errors in practice. This is achieved in part because Elm handles errors as data. Instead of causing a crash, Elm explicitly models the potential for failure using custom types, allowing for more graceful error handling. This feature has been extensively utilized in the client-side development.

4.5 Encryption

Encryption is a crucial security strategy that safeguards the confidentiality and integrity of data, whether it's stored or being transmitted. This technique converts accessible data, or plain text, into a scrambled form, referred to as cypher text, that can only be interpreted or utilized once it's been decrypted. Encryption is vital for protecting sensitive information from unauthorized access, a concern especially relevant for applications like DSW that manage sensitive research data.

Based on the code, DSW uses a cryptographic utility module that includes functions for generating random strings, encrypting and decrypting sensitive

data and configurations with AES-256, hashing with MD5, and reading RSA keys from memory `Crypto.hs`. This module includes a wrapper for the encryption and decryption of data.

For example, here is a function that encrypts plain text into a cypher text using AES-256:

```
encryptAES256WithB64 :: String -> String -> String
encryptAES256WithB64 key plainData =
  if B64.isBase64 (BS.pack plainData)
  then BS.unpack $ encryptAES256Raw (BS.pack key)
    → (B64.decodeBase64Lenient . BS.pack $ plainData)
  else BS.unpack . B64.encodeBase64' \$ encryptAES256Raw
    → (BS.pack key) (BS.pack plainData)
```

Listing 19: Definition of `encryptAES256WithB64` function

This function is designed to handle encryption of data that may or may not already be Base64-encoded. If `plainData` is already Base64 encoded, it gets decoded from Base64, but it will happen leniently, meaning that all non-Base64 characters will be ignored. The decoded data is then encrypted using the `encryptAES256Raw` function with the provided key. The encrypted data, which is a `ByteString`, is then unpacked back into a string. If `plainData` is not Base64 encoded, the function encrypts the data and then encodes it in Base64: encrypt the message with the key, encode the cypher into Base64, pad it, and, finally, unpack it into a string.

4.6 Hashing

Unlike encryption, which is designed to be reversible, hashing is a one-way process that converts data into a fixed-size string of characters, which is typically a digest that represents the data. Hashing is widely used for securing passwords as it allows systems to verify data integrity without needing to know the actual data.

Hashing methods are located in the same file as the encryption ones. Here is an example:

```
hashMD5 :: String -> String
hashMD5 text =
  let digest :: Digest MD5
      digest = hash . TE.encodeUtf8 . T.pack \$ text
  in show digest
```

Listing 20: Definition of `hashMD5` function

This function receives a plain text, which is a string. Inside the function, a digest is computed by first packing the input into a `Text` type using `T.pack`. Then it is encoded into a UTF-8 `ByteString`. After all, using `Crypto.Hash.hash` function, the hash is generated. `Digest MD5` is a type-safe representation of the MD5 hash. Finally, the `show` function converts `Digest`

MD5 into a string. The result is the MD5 hash of the input as a hexadecimal string.

4.7 Session Management

A web session represents is a series of HTTP requests and responses linked to a single user. Contemporary and sophisticated web applications need to maintain user-specific information or status across numerous requests. Sessions enable the setting of certain variables, like access permissions and language preferences, that persist throughout all of a user's interactions with the web application for the time the session is active.

Session management in DSW is meticulously designed to handle user states across multiple requests. When a user logs in, DSW creates a unique session identifier (session ID) that is securely transmitted between the client and server for the duration of the session. This session ID is a critical component, as it is the key to accessing the user's session data stored on the server.

Based on the code, the whole session management is based on the session key, a variable set to `session/wizard`, which is consistently used across different parts of the code to access the session information using the function `localStorage(sessionKey)`. This data includes a JSON Web Token, which is used to authenticate API requests. JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties [12]. This token is included in the `Authorization` header of HTTP requests made to the server.

When a user enters their credentials, if the server-side authentication process verifies them, the session token is generated, which includes its session expiration time. Once the expiration time is reached, the session is closed, all the data from `localStorage` is wiped out, and the user is notified that the session has expired.

4.8 Logging and Monitoring

The concept of logging is a very important aspect of the security of the system. Logging is used for debugging applications and their diagnostic. Monitoring is the live review of application and security logs using various forms of automation[17]. Application logging might be useful for identifying security incidents, monitoring them, and identifying system problems, and actors who allowed this. However, by being too specific, the same logs can be used as a source of sensitive information.

In the server part of the DSW, logs can go to two places – standard output (`runStdoutLoggingT`) and/or to the Sentry (`sendToSentry`) if the type of the log is `Error`. Sentry is a software monitoring tool that helps developers identify and fix code-related issues. From error tracking to performance monitoring, Sentry provides code-level observability that makes it easy to diagnose issues and learn continuously about your application code health [26].

There are four types of logging functions:

- `logDebugI` – for logging debug-level messages.
- `logInfoI` – for logging informational messages.

- `logWarnI` – for logging warnings.
- `logErrorI` – for logging errors.

Each function corresponds to its log level `LogLevel`, which is a part of the `Control.Monad.Logger` module introduced in Haskell. The application also keeps audit trails – records that chronologically catalogue system activities by recording who performed an action, what action was performed, and when it was performed.

4.9 Security Checks

The Data Stewardship Wizard demonstrates a commitment to maintaining a secure and reliable application through its implementation of automated security checks. A key component of this security strategy is the utilization of Grype, a service integrated into the application’s GitHub repository for local deployment. Grype specializes in scanning and identifying vulnerabilities within the application’s dependencies.

One of the most notable aspects of DSW’s security approach is the daily execution of security audits. These audits are automated and rigorously scan the application to identify potential vulnerabilities. The results of these audits are transparently stored in the Security Audit section, accessible publicly. This level of transparency not only fosters trust but also allows for continuous monitoring and assessment of the application’s security posture.

The audit results reveal various vulnerabilities within the application. However, it is important to note that none of these vulnerabilities have been classified as “critical”. This classification indicates that while there are areas of concern, they do not pose an immediate and severe threat to the application’s overall security.

DSW’s official documentation highlights a responsive approach to handling serious bugs and vulnerabilities. In the event that a critical vulnerability is identified, the development team is immediately notified. This prompt response is crucial for mitigating potential risks and is followed by the swift release of a hotfix. Such a proactive stance on addressing security issues underscores the development team’s dedication to ensuring the application’s integrity and the safety of its users.

In addition to its existing robust security measures, the Data Stewardship Wizard also utilizes Dependabot, a crucial tool for enhancing its security framework. Dependabot, an automated service integrated within GitHub, routinely checks the project’s dependencies for any known vulnerabilities or outdated components. Upon detecting such issues, Dependabot automatically creates pull requests to upgrade these dependencies to their latest, more secure versions. This ongoing process of monitoring and updating plays a significant role in mitigating the risk of security breaches linked to vulnerable dependencies. By integrating Dependabot into its operational routine, DSW ensures that it consistently applies the latest security updates, thus bolstering the platform’s defence against new and evolving threats.

Technology recommendations

Based on the analysis of the code and documentation, several modifications have been identified that could greatly enhance the overall security of the application.

5.1 Login Process

The login procedure appears to be well-implemented. Nonetheless, the absence of rate limiting on login attempts could leave the door open for brute-force attacks. Implementing a rate-limiting mechanism could be an effective strategy to prevent such security breaches. Rate limiting is a strategy for limiting network traffic by putting a “cap” on how often someone can repeat an action within a certain timeframe [7]. Additionally, introducing rate limiting can protect the system against DoS, DDoS, and web scraping attacks.

Concerns were also raised regarding password strength. Currently, the system allows weak passwords such as `password1234`, `passwordpassword`, or `aaaaaaaaaa`. The strength of these passwords can be checked online using tools like Password Monster (<https://www.passwordmonster.com/>). Furthermore, using the Rumkin library’s validator (<https://rumkin.com/tools/password>), it’s evident that passwords accepted by the application can range from weak to strong. This indicates a need for more advanced security measures in the login process, such as CAPTCHA implementation, timeouts after a series of incorrect login attempts, and a stricter password policy to enhance overall security.

5.2 Cryptography

Based on the code, the implementation of encryption and decryption is good, because AES256 is currently considered a secure algorithm. But because it uses CTR (counter) mode, and the initial vector (IV) is null `'nullIV'`, which is a static IV of all zeroes, this can be a significant vulnerability, because the same plain text will always result in the same cypher text, when encrypted with the same key, making the encryption deterministic and susceptible to various attacks. The IV must be generated randomly for each encryption operation.

Additionally, the hashing algorithm MD5, which is used as the main hashing algorithm, is outdated. The MD5 hash function’s security is considered to be severely compromised. Collisions can be found within seconds, and they

can be used for malicious purposes [15]. It is advisable to switch from MD5 to a more secure hashing algorithm like ARGON2, which, despite its computational intensity, offers enhanced security. If the data isn't highly sensitive, a less complex yet still secure option like SHA-256 or SHA-3 could be considered. These algorithms are generally secure and are suitable for tasks that require data integrity verification or authenticity checks, although they are less resource-intensive than ARGON2.

5.3 Authorization and Logging

The application strikes a balance with logs that are informative without being overly detailed. Nevertheless, it would be beneficial to implement an automated audit analysis system to swiftly detect anomalies and potential security threats. Given the critical nature of these logs, it's also wise to consider regular backups. In the event of a disaster, these backups could serve as a crucial component of the recovery strategy.

Vulnerabilities analysis

As I delve deeper into the security analysis of the Data Stewardship Wizard, it's crucial to build upon the insights and observations from the previous chapters. The application, crafted with a focus on robust and secure programming practices, stands as a formidable barrier against a range of common cybersecurity threats. However, the inherent complexities of web applications, combined with the dynamic and ever-evolving nature of cyber threats, call for a more comprehensive and nuanced examination that extends beyond a standard observation based on the code review.

While DSW has been diligent in implementing a variety of security measures, a fundamental principle in cybersecurity is the acknowledgement that no system is entirely invulnerable to attacks. Potential security weaknesses in DSW might not always be directly linked to the application's core coding. Instead, they could be the result of external factors, such as dependencies on third-party libraries, the integration of external APIs, or even the methodologies employed in certain feature implementations.

For instance, the application uses a third-party Jinja2 template format. Based on the code in `dsw-document-worker`, after successful validation, which is described in [Section 4.3.1](#), the template is built based on the `ByteString`. However, it is later executed, which could be exploited by injecting malicious code into the template body.

Moreover, the application's interaction with external services and between components can also introduce vulnerabilities. Issues such as data leakage, improper error handling, or insufficient validation of external data can pose significant risks. Additionally, the evolving landscape of user behavior and expectations can lead to new security challenges. Features that were once considered secure may become liabilities as new attack vectors are discovered and exploited by malicious actors.

In this chapter, I will summarize the issues identified in the code from previous chapters, and then I will use this knowledge to perform penetration testing on the application.

6.1 Penetration Testing Setup

For the security testing of the Data Stewardship Wizard application, I utilized a virtual machine running Kali Linux, a popular choice for penetration testing

due to its comprehensive suite of tools. The following is a detailed list of the main tools employed in the testing process, along with their respective versions:

- **Kali Linux Version:** 2023.4 - This version of Kali Linux provided a stable and up-to-date environment for conducting the tests.
- **Sqlmap Version:** 1.7.12#stable - Sqlmap was used for automating the detection and exploitation of SQL injection vulnerabilities.
- **Burp Suite Community Edition Version:** 2023.10.3.6-24994 - The Community Edition of Burp Suite offered essential features for web application security testing.
- **Burp Suite Professional Edition Version:** 2023.11.1.2-25469 - The Professional Edition of Burp Suite, used for its advanced capabilities in security testing, was a key component in the testing process.

Each of these tools played a critical role in the comprehensive security assessment of DSW, ensuring a thorough evaluation of potential vulnerabilities and security weaknesses within the application.

6.2 Penetration Testing

In this section, I will methodically explore and attempt some of the most critical and potentially devastating attacks on the locally deployed version of the application. My focus will be on systematically applying each attack, one by one, to assess the application's resilience and identify any vulnerabilities.

6.2.1 Disclaimer for Penetration Testing Report

This penetration testing report has been compiled with the utmost diligence and attention to detail, aiming to provide a comprehensive evaluation of the security posture of the application under test. Despite the thoroughness of the testing methodology and the extensive range of tests conducted, it is important to acknowledge certain limitations inherent in any penetration testing process.

Firstly, while every effort has been made to cover a broad spectrum of potential vulnerabilities, it is impossible for any test to encompass all possible combinations and scenarios. The field of cybersecurity is vast and ever-evolving, and new vulnerabilities or attack vectors may emerge that were not known or considered at the time of testing. Consequently, there may be security weaknesses within the application that this test has not detected.

Secondly, it is also possible that some of the findings reported may include features or behaviors that were intentionally designed and implemented as part of the application. These may have been identified as potential vulnerabilities in the context of this test, but they could be intentional aspects of the application, designed to meet specific requirements or functionalities.

Therefore, while this report provides valuable insights and identifies potential areas of concern, it should not be considered an exhaustive or definitive assessment of the application's security. It is recommended that security testing be an ongoing process, complementing this report with regular updates, reviews, and tests to adapt to new threats and changes in the application.

The findings and recommendations in this report are intended to enhance the security of the application and should be used as a guide for further investigation and remediation efforts. It is the responsibility of the application's developers and security team to assess the relevance and impact of each finding in the context of the application's specific environment and requirements.

6.2.2 Brute-force Attack

Recalling previous chapters, a brute-force attack is a trial-and-error method used to decode encrypted data such as passwords. This attack method involves systematically checking all possible combinations until the correct one is found. It is a straightforward approach where the attacker tries numerous possibilities, hoping to eventually guess correctly. The attack's success largely depends on the complexity and length of the password.

In my penetration testing scenario, I focused on executing a brute-force attack against the locally deployed Data Stewardship Wizard application. The setup involved a Kali Linux environment and the DSW application running on 127.0.0.1:8080. My tool of choice for this attack was Burp Suite, a powerful tool for web application security testing.

The environment setup required configuring a proxy server, so I decided to go with 127.0.0.1:8081. The best way to do that as fast as possible was using FoxyProxy. This application helps to easily switch between proxy setups, which is very helpful with this and following attacks. However, the new version of the application didn't work well. User reviews were good proof that something wasn't right there. Having this figured out, I installed an older version of the application. After the setup, the application was accessed through this proxy, allowing Burp Suite to intercept the HTTP requests.

Upon loading the DSW login page, a test email address and an incorrect password were entered, resulting in an "Incorrect email or password" message. Using Burp Suite connected to the proxy, the login attempt was captured (Listing 21).

```
OPTIONS /wizard-api/tokens HTTP/1.1
Host: localhost:3000
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0)
→ Gecko/20100101 Firefox/115.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Access-Control-Request-Method: POST
Access-Control-Request-Headers: content-type
Referer: http://localhost:8080/
Origin: http://localhost:8080
Connection: close
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-site
```

Listing 21: Captured first login request

The initial request intercepted did not contain credential information, so it was released. The subsequent request captured (Listing 22) contained the relevant fields for the brute-force.

```
POST /wizard-api/tokens HTTP/1.1
Host: localhost:3000
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0)
↳ Gecko/20100101 Firefox/115.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/json
Content-Length: 74
Origin: http://localhost:8080
Connection: close
Referer: http://localhost:8080/
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-site

{"email":"albert.einstein@example.com","password":"password1","}
↳ code:null}
```

Listing 22: Captured second login request

The captured request was sent to Burp Suite’s Intruder module. Intruder is a tool within Burp Suite designed for automating customized attacks against web applications. In this case, it was configured to perform a brute-force attack.

Positions: As I was trying to crack the password, I had to modify the password field in the following way:

```
{
  "email":"albert.einstein@example.com",
  "password":"$Spasword$",
  "code":null
}
```

Listing 23: Modified JSON part of the request

Attack Type: Sniper mode was selected, which is effective for testing individual parameters. Payload: A custom wordlist (Figure 6.1) was used as the payload. This list included several incorrect passwords and one correct password. The absence of a timeout feature in the application’s login process allowed for rapid testing without significant delays. Grep - Match: The error message “Incorrect email or password” was set up in the Grep - Match section to identify failed login attempts.

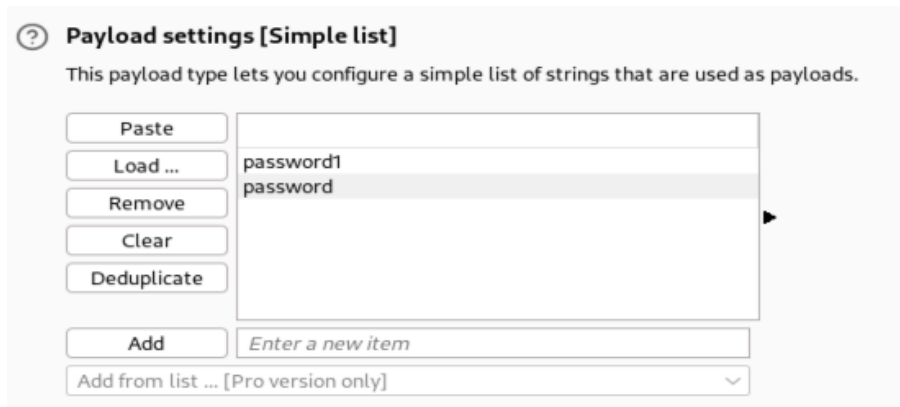


Figure 6.1: Wordlist for the attack

The attack was initiated, and Burp Suite’s Intruder sent two requests. The first request, containing `password1`, failed to log in, as indicated by the presence of the error message. However, the second request, containing the correct password `password`, succeeded, evidenced by the absence of the error message in the response (see Figure 6.2).

Request	Payload	Status code	Error	Timeout	Length	Incorrect e...	Com...
0		400	<input type="checkbox"/>	<input type="checkbox"/>	537	1	
1	password1	400	<input type="checkbox"/>	<input type="checkbox"/>	537	1	
2	password	201	<input type="checkbox"/>	<input type="checkbox"/>	1446		

Figure 6.2: The result of the attack

Conclusion: The brute-force attack successfully identified the correct password, highlighting a potential vulnerability in the DSW application. The absence of account lockout mechanisms or timeout features after consecutive failed login attempts made the application susceptible to this type of attack. This test underscores the importance of implementing robust authentication controls to safeguard against brute-force attempts.

6.2.3 SQL Injection

In the previous sections, I’ve established that the Data Stewardship Wizard exhibits strong capabilities in managing SQL operations. This section will further explore the application’s defence against SQL Injection vulnerabilities, with a specific focus on three key components: `engine-frontend`, `engine-backend`, and `engine-tools`.

- **engine-frontend:** based on the code analysis, this component of the application does not directly interact with SQL. Instead, it serves as an intermediary, passing values and data to the backend for SQL query processing. This architecture inherently reduces the front-end's exposure to SQL Injection risks.
- **engine-tools:** while the tools repository does engage with SQL, it's important to note that the inputs for these SQL queries are statically defined within the code. This means that they are not influenced by external user input, thereby mitigating the risk of SQL Injection through this component.
- **engine-backend:** given these observations, the primary focus of my SQL Injection vulnerability test will be on the engine-backend component. This part of the application is responsible for handling and executing SQL queries, making it a critical area for assessing the application's vulnerability to SQL Injection attacks.

The testing will employ two highly effective tools: Sqlmap and Burp Professional. Sqlmap is an open-source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws and taking over of database servers [27]. Burp, on the other hand, is a half-automated tool which allows running an active search for every field of the HTTP request. By combining these tools, I will be able to simulate hundreds of requests which might be able to identify vulnerable places.

Burp Suite, conversely, is a semi-automated tool that facilitates active searches across all fields of an HTTP request. By utilizing both tools in tandem, I aim to simulate numerous requests that could potentially uncover vulnerabilities.

Through code analysis, it was determined that the Data Stewardship Wizard employs the `Database.PostgreSQL.Simple` library for executing queries. As discussed in previous chapters, this approach, composing a separate SQL query string, preparing a list of parameters for the string, and then invoking the `Database.PostgreSQL.Simple::query` function – is deemed a secure method for executing SQL statements.

To ensure thorough testing, I will select specific areas based on the code for conducting tests. The first area is the login page. Initially, I intercepted the login request the same way I did for the Brute-Force attack (see Listing 22). This request was then altered by inserting “§” symbols before and after the input fields – email, password, and code. Subsequently, the modified request was sent for an active scan. As anticipated, no issues were found. Following this, the same request, without any modifications, was saved to a file for later use with Sqlmap. To execute the attack, I used the following command:

```
sudo sqlmap --dbms=PostgreSQL --level=5 --risk=3
↪ --proxy=https://127.0.0.1:8081 -r req4.txt -p parameter
↪ --os-shell --os-pwn
```

-dbms=PostgreSQL This option specifies that the target database is PostgreSQL.

-level=5 This option specifies how deep the test has to be performed - how

many test and how many entry points will be used, where 5 is the maximum.
-risk=3 This option specifies the risk level, where 3 is the maximum.
-proxy=https://127.0.0.1:8081 This options specifies the proxy settings.
-r request This option specifies the file with the request.
-p parameter This option specifies what parameter should be tested i.e. "email".
-os-shell If the vulnerability is found, this option will prompt for an interactive operating system shell.
-os-pwn If the vulnerability is found, this option will ask to Prompt for an Out-of-Band (OOB) shell, Meterpreter or VNC.

The first step involved specifying the request file and input parameters and then running the command through Sqlmap. This **initial test did not reveal any vulnerabilities**.

The next request I examined was one related to "creating a template editor".

```
POST /document-template-drafts HTTP/1.1
Host: localhost:3000
Content-Length: 101
sec-ch-ua: "Chromium";v="119", "Not?A_Brand";v="24"
Content-Type: application/json
sec-ch-ua-mobile: ?0
Authorization: Bearer DELETED AUTH TOKEN
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
↳ AppleWebKit/537.36 (KHTML, like Gecko)
↳ Chrome/119.0.6045.199 Safari/537.36
sec-ch-ua-platform: "Linux"
Accept: */*
Origin: http://localhost:8080
Sec-Fetch-Site: same-site
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://localhost:8080/
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Connection: close

{"name":"dfasfasd","templateId":"dfasfasdfaf","version":"1.0.0"}
↳ , "basedOn":"myorg1:dfasfasdfaf:1.0.0"}
```

During this phase, both the active analysis and Sqlmap were employed to scrutinize various aspects of the request, including the `name`, `templateId`, and `basedOn` columns. However, **these tests also did not indicate any vulnerabilities**.

The final request I chose to investigate was related to changing system settings. Due to the extensive nature of this request, which includes a large JSON segment encompassing various settings and their values, I have not included the full request here.

```
PUT /configs/app HTTP/1.1
```

In this instance, I again utilized Sqlmap and Burp's active analysis, testing different settings within the request. Similar to the previous tests, this comprehensive examination did not uncover any vulnerabilities. This thorough testing process, encompassing a range of critical requests and employing advanced tools, reinforces the robustness of the application's security measures against SQL injection and other potential vulnerabilities.

6.2.4 Cross-Site Scripting

Despite Cross-Site Scripting (XSS) no longer being featured in the OWASP Top 10, it continues to be a common and relatively easy attack to execute. Bounty hunters are often well-compensated for discovering XSS vulnerabilities, given their significant impact. In the Data Stewardship Wizard application, various requests involve user input, presenting potential opportunities for XSS exploits. The goal of this test was to confirm that none of these requests were susceptible to over 100 different XSS attack combinations, as identified in Burp Suite's wordlists. Below is a comprehensive list of all HTTP requests that were thoroughly analyzed, with each possible input parameter being tested:

```
GET /packages?page=0&q=test&sort=name,asc&size=20 HTTP/1.1

GET /branches?page=0&q=test&sort=updatedAt,desc&size=20 HTTP/1.1

GET /document-templates?page=0&q=test&sort=name,asc&size=20
↳ HTTP/1.1

GET /document-template-drafts?page=0&q=test&sort=updatedAt,desc
↳ &size=20
↳ HTTP/1.1

GET /questionnaires?page=0&q=test&sort=updatedAt,desc&size=20
↳ HTTP/1.1

GET /questionnaire-importers?page=0&q=test&sort=name,asc&size=20
↳ 0
↳ HTTP/1.1

GET /documents?page=0&q=test&sort=createdAt,desc&size=20
↳ HTTP/1.1

PUT /configs/app HTTP/1.1

GET /questionnaires/{questionnaireID}/websocket?Authorization=Bearer%REST-OF-AUTH-TOKEN
↳ HTTP/1.1

GET /questionnaire-importers/suggestions?page=0&sort=name,asc&size=20&questionnaireUuid={questionnaireID}&enabled=true
↳ HTTP/1.1

GET /questionnaires/{questionnaireID}/report HTTP/1.1
```

PUT /questionnaires/{questionnaireID} HTTP/1.1

GET /users/suggestions?page=0&q=test&size=20 HTTP/1.1

Throughout this testing process, over 2000 requests were made, and impressively, no vulnerabilities were found. However, it's important to note that while semi-automated tools like these are powerful, they cannot provide a 100% guarantee of security. XSS attacks, which often hinge on the subtleties of user input and application responses, can sometimes slip through even the most advanced detection methods. As such, while the findings from this test are promising, they should be considered as part of a continuous security evaluation process, rather than a final verdict on the application's immunity to XSS attacks.

6.2.5 Cross-Site Request Forgery

In this security analysis of the Data Stewardship Wizard, the testing methodology was specifically designed to suit the project's unique goals and limitations. Although Cross-Site Request Forgery (CSRF), Denial of Service (DoS), and Distributed Denial of Service (DDoS) attacks are crucial in web application security, they were not the primary focus of this project for several reasons.

Cross-Site Request Forgery (CSRF): CSRF attacks leverage the trust a web application places in an authenticated user, deceiving the user into making an unintended request. These attacks are particularly pertinent in multi-user environments, often involving a victim unknowingly performing actions on the attacker's behalf.

The chosen test environment, a locally deployed instance with a single user, lacks the multi-user dynamics needed to effectively simulate CSRF attacks. The controlled nature of this environment, without real-world user interactions, limits the relevance of CSRF testing in this scenario.

While not a focus of my project, CSRF is a significant security concern for applications with numerous active users. Recognized in the OWASP Top 7 API Security Risks 2023 [21], CSRF underscores the importance of strong protective measures in live, production settings.

A more appropriate setting for CSRF vulnerability testing would involve a deployed application with multiple active users and simulated interactions that reflect real-world usage, enabling a more accurate evaluation of CSRF attack susceptibility.

Denial of Service (DoS) and Distributed Denial of Service (DDoS): DoS and DDoS attacks aim to render a web service unavailable by inundating it with traffic. These attacks target the availability aspect of security, focusing on the application's infrastructure and network resilience.

This project concentrates on DSW's application-level security, not its network or infrastructure resilience. DoS and DDoS attacks pertain more to these layers, which are outside my analysis scope.

Conducting DoS or DDoS attacks also requires a setup that often involves simulating network traffic and stress-testing infrastructure. Such tests also carry ethical and legal considerations, particularly if not performed in a controlled, authorized environment.

Given the nature of my testing environment and my project's focus on application-level vulnerabilities, my approach is more targeted. The efforts are thus dedicated to identifying and addressing vulnerabilities that can be effectively tested and analyzed within my local, single-user deployment of DSW.

6.2.6 Broken Authentication

In the Section 6.2.2, I discussed how the Data Stewardship Wizard shows considerable susceptibility to brute-force attacks. This vulnerability is primarily due to the lack of protective measures like timeouts, CAPTCHAs, or account lockout policies. Such shortcomings make the application an easy target for these types of attacks, raising serious concerns about its security framework.

A specific issue arises during the DSW signup process. The application indicates if an email address is already in use by displaying a message stating "Email is already in use". This response enables user enumeration, a technique where attackers can determine which email addresses are registered on the application. By using a tool like Burp Suite to sort responses by length, attackers can efficiently compile a list of valid user emails. This information becomes a valuable asset for launching targeted brute-force attacks to guess passwords.

The implications of having a list of known user emails are significant. Attackers can systematically attempt brute-force attacks on each account, dramatically increasing the chances of unauthorized access. Moreover, these valid email addresses can be exploited in phishing campaigns, making them more effective. The absence of email validation timeouts is another concern, as it allows attackers to create multiple accounts, potentially using real users' email addresses.

```
{  
  "email": "test@gmail.com",  
  "firstName": "Anything",  
  "lastName": "Anything",  
  "affiliation": null,  
  "password": "Anything1",  
  "role": null  
}
```

Listing 24: JSON part from the signup request.

On the Listing 24 you can see how the entered data is transferred in the HTTP request. By inserting "test" within "\$" signs (\$test@gmail.com), I can execute the attack using Burp (see Figure 6.3), which will sequentially create an increasing number of accounts (refer to Figure 6.4). However, since these accounts will not be deleted if they are not activated after a certain period, this approach could lead to a situation where legitimate users are unable to register with their own email addresses, or the application could run out of resources.

6.2. Penetration Testing

The screenshot shows the 'Payloads' tab in Burp Suite. At the top, there are tabs for 'Positions', 'Payloads', 'Resource pool', and 'Settings'. Below the tabs, there is a 'Start attack' button. The main content is divided into two sections: 'Payload sets' and 'Payload settings [Brute forcer]'. The 'Payload sets' section includes a description, a 'Payload set' dropdown menu set to '1', a 'Payload count' of 1,679,616, a 'Payload type' dropdown menu set to 'Brute forcer', and a 'Request count' of 1,679,616. The 'Payload settings [Brute forcer]' section includes a description, a 'Character set' text input field containing 'abcdefghijklmnopqrstuvwxyz0123456789', a 'Min length' input field set to '4', and a 'Max length' input field set to '4'.

Positions Payloads Resource pool Settings

? Payload sets Start attack

You can define one or more payload sets. The number of payload sets depends on the attack type defined in the Positions tab. Various payload types are available for each payload set, and each payload type can be customized in different ways.

Payload set: 1 Payload count: 1,679,616

Payload type: Brute forcer Request count: 1,679,616

? Payload settings [Brute forcer]

This payload type generates payloads of specified lengths that contain all permutations of a specified character set.

Character set: abcdefghijklmnopqrstuvwxyz0123456789

Min length: 4

Max length: 4

Figure 6.3: Setting up the Bulk Account Registration Attack using Burp Suite

















	Anything Anything oaaa@gmail.com · internal	Data Steward	inactive
	Anything Anything paaa@gmail.com · internal	Data Steward	inactive
	Anything Anything raaa@gmail.com · internal	Data Steward	inactive
	Anything Anything qaaa@gmail.com · internal	Data Steward	inactive
	Anything Anything saaa@gmail.com · internal	Data Steward	inactive
	Anything Anything taaa@gmail.com · internal	Data Steward	inactive
	Anything Anything uaaa@gmail.com · internal	Data Steward	inactive
	Anything Anything vaaa@gmail.com · internal	Data Steward	inactive
	Anything Anything jaaa@gmail.com · internal	Data Steward	inactive
	Anything Anything kaaa@gmail.com · internal	Data Steward	inactive
	Anything Anything faaa@gmail.com · internal	Data Steward	inactive
	Anything Anything gaaa@gmail.com · internal	Data Steward	inactive
	Anything Anything eaaa@gmail.com · internal	Data Steward	inactive
	Anything Anything caaa@gmail.com · internal	Data Steward	inactive
	Anything Anything haaa@gmail.com · internal	Data Steward	inactive
	Anything Anything iaaa@gmail.com · internal	Data Steward	inactive

Figure 6.4: Results of the Bulk Account Registration Attack

What's also worth mentioning is the fact that by updating a user password by using the `PUT /users/userID/password` HTTP/1.1 request, which is responsible for updating a user's password by an admin, the changed password can be even simpler or even be empty:

```
{
```

```

    "password": "",
}

```

On a more positive note, DSW's session management seems to be well-implemented. Each login session is linked to a unique token, enhancing security by maintaining session integrity. These tokens are designed to expire after a certain period, requiring users to re-authenticate, which is a commendable security measure. However, despite the robustness of the session management system, it does not fully compensate for the vulnerabilities present in the authentication process. These issues, if not addressed, could undermine the overall security of the application.

6.2.7 Broken Access Control

In the previous sections, I established that the Data Stewardship Wizard has implemented a sophisticated system to control application functionalities among different user roles. As mentioned earlier, the primary roles in the application are admin, data steward, researcher, and anonymous. However, within a project, the access control system is more nuanced, with roles determining varying levels of privileges:

- **Owner:** Project owners have comprehensive control over their projects. They can fill out questionnaires, access metrics, preview the project, create documents, leave comments, and change settings. It's important to note that users with the basic application role of `admin` have all the privileges of a project owner, regardless of their assigned project role.
- **Editor:** Project editors can perform all actions available to project owners, except for altering project settings.
- **Commenter:** Project commenters are allowed to do everything editors can, except for filling out questionnaires and creating documents.
- **Viewer:** Project viewers are limited to viewing filled-in questionnaire information, metrics, previews, and created project documents.

These roles are manually assigned using the `Share` option, where each user is designated a specific role. A project can also be made public to those who have the link. Users accessing the project via the link also have roles, but this role is common to all link-holders and is set in the same menu.

For the description of access based on roles, I need to consider two sets of roles, each arranged in descending order of permissions:

- **Application Roles:** Admin, Data Steward, Researcher.
- **Project Roles:** Admin, Owner, Editor, Commenter, Viewer, Not Logged In User.

The aim of this test, conducted from a non-developer perspective and without access to published documentation on user permissions, is to simulate a variety of API requests across different application and project roles. This approach is based on observing the capabilities and restrictions within the GUI -

essentially, what different users can or cannot do. The objective is to validate that users who are not authorized to perform certain actions, as indicated by their GUI access, are indeed unable to execute these actions through the API.

A detailed enumeration of all the requests simulated during the testing phase is comprehensively documented in the file [http_requests.pdf](#). In this section, however, I have selectively highlighted only those requests that elicited particular interest or raised questions. This focus is primarily due to discrepancies observed between their access rules and the functionalities as presented from the graphical user interface perspective. For a more extensive review, readers are encouraged to refer to the mentioned file, where each request is meticulously catalogued.

POST /document-template-drafts HTTP/1.1

Description: Create document template editor

Access Requirements: Accessible for **Data Steward** and above.

Result: **Failed**. This request not only can be executed by a researcher, but it also will create a document template editor, which is not allowed in the GUI.

GET /document-templates/{templateID} HTTP/1.1

Description: Get information about the document template.

Access Requirements: Accessible for **Data Steward** and above.

Result: **Failed**. This request can be executed by a researcher, and the template information will be sent in the response. This is not allowed in the GUI.

POST /document-templates/bundle HTTP/1.1

Description: Import a new template from a file.

Access Requirements: Accessible for **Data Steward** and above.

Result: **Failed**. This request can be executed by a researcher, and the template will be created using the information in the request. This is not allowed in the GUI.

POST /knowledge-models/preview HTTP/1.1

Description: Preview a knowledge model.

Access Requirements: Accessible for **Data Steward** and above.

Result: **Failed**. This request can be executed by a researcher, and the knowledge model preview will be displayed in the response. This is not allowed in the GUI.

GET /packages HTTP/1.1

Description: Get the list of all knowledge models.

Access Requirements: Accessible for **Data Steward** and above.

Result: **Failed**. This request can be executed by a researcher. This is not allowed in the GUI.

GET /packages/{knowledgeModelID} HTTP/1.1

Description: Get information about the knowledge model.

Access Requirements: Accessible for **Data Steward** and above.

Result: **Failed**. This request can be executed by a researcher. This is not allowed in the GUI.

`GET /packages/{knowledgeModelID}/bundle HTTP/1.1`

Description: Export the knowledge model.

Access Requirements: Accessible for **Data Steward** and above.

Result: **Failed**. This request can be executed by a researcher. This is not allowed in the GUI.

`GET /questionnaire-importers HTTP/1.1`

Description: Get questionnaire importers.

Access Requirements: Accessible for **Data Steward** and above.

Result: **Failed**. This request can be executed by a researcher. This is not allowed in the GUI.

`POST /questionnaires/{questionnaireID}/revert/preview HTTP/1.1`

Description: Preview the questionnaire of the project document

Access Requirements: Accessible for **Viewer** and above.

Result: **Failed**. This request can be executed by anyone with the questionnaire ID, even if the project is not accessible via link. Using this request, I managed to intercept all the information which was filled in the questionnaire.

`GET /questionnaires/{questionnaireID}/events HTTP/1.1`

Description: View the history of the project.

Access Requirements: Accessible for **Owner** and **Admin**.

Result: **Failed**. This request can be executed by anyone with the questionnaire ID, even if the project is not accessible via link.

`GET /configs/bootstrap?clientId=http%3A%2F%2Flocalhost%3A8080
↪ HTTP/1.1`

Description: Get the application's configuration.

Access Requirements: Accessible for **Admin** only.

Result: **Failed**. All users can run this request and receive the application's configuration.

`GET /usage HTTP/1.1`

Description: Get the application usage - number of users, active users, projects etc.

Access Requirements: Accessible for **Admin** only.

Result: **Failed**. All authenticated users can run this request and receive the application's usage information.

`POST /users HTTP/1.1`

Description: Create a user.

Access Requirements: Accessible for Admin only.

Result: **Failed.** All authenticated users can run this request and create inactivated user accounts. But if in the application's settings the default role is not researcher, users with less privileged accounts might be able to create accounts with higher roles.

It's crucial to note that the requests covered here are all that I could identify, but there may be additional ones. This is particularly relevant for features like email validation, document submission, and creating locales, as these either cannot be tested on a locally deployed application or require essential files that are missing. Based on the requests described earlier, the application appears to have some issues with access controls. These issues were promptly reported to the application developers for them to issue a fix.

Moreover, certain features of the application, while not accessible to unauthorized users, are still visible to them. This includes options like "Start migration" and "Delete project" in a project. Should an unauthorized user attempt to use these options in the GUI, they would encounter an error message stating, "You do not have permission to view this page". This message is somewhat misleading, as the issue is not about viewing a page but rather about the authorization to perform certain actions.

6.2.8 Token Leakage

As it was described above, token leakage is a situation when an unauthorized party obtains an authentication token. This situation can also be achieved when the OAuth token is shared in the URL.

In the Data Stewardship Wizard, a specific instance of token leakage is observed during the process of retrieving information about a knowledge model or a knowledge model's fork. This issue becomes apparent in the second consecutive request to the system. The request format, as observed, is as follows:

```
GET /questionnaires/{questionnaireID}/websocket?Authorization=Bearer%REST-OF-AUTH-TOKEN
↪ HTTP/1.1
Host: localhost:3000
Connection: Upgrade
Pragma: no-cache
Cache-Control: no-cache
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
↪ AppleWebKit/537.36 (KHTML, like Gecko)
↪ Chrome/119.0.6045.199 Safari/537.36
Upgrade: websocket
Origin: http://localhost:8080
Sec-WebSocket-Version: 13
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Sec-WebSocket-Key: GUSF1jW7XSZd6b4XS22PkA==
```

In this scenario, the most significant issue is that the authentication token (a Bearer token) is included in the URL of the GET request. URLs are generally

logged in various places such as web server logs, proxy server logs, browser history, and potentially other monitoring tools. This exposure means that the token, which should be confidential, could be accessed by unauthorized parties. This practice is problematic for several reasons:

- **URL Logging:** URLs are often logged by web servers, browsers, and intermediary devices or software. This means that the token, embedded in the URL, could be inadvertently logged and stored in an unsecured manner.
- **Browser History:** URLs with tokens can also be saved in the browser history, making them accessible to anyone with access to the user's device.
- **Referer Headers:** When a web request is made, browsers typically send the URL of the current page as a Referer header to the destination server. If the URL contains the token, it could be inadvertently shared with third parties.
- **Cache Storage:** URLs may also be cached by the browser or by intermediary proxies, further increasing the risk of exposure.
- **Stateless Authentication** The token appears to be a JSON Web Token, which is often used for stateless authentication. If this token is leaked, it could allow an attacker to impersonate the user it represents, potentially gaining unauthorized access to sensitive resources or data.

6.2.9 Server-Side Template Injection

Server-side template injection is a critical security vulnerability that targets template engines, such as Jinja2, which is employed by the Data Stewardship Wizard. To assess the susceptibility of DSW to SSTI, I conducted a series of tests using a template provided by one of the developers and my supervisor, Marek Suchanek. The template's content, designed to test the application's template processing capabilities, is as follows:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <title>Testing SSTI</title>
</head>

<body>
  <h1>Testing SSTI</h1>

  {#- possibly something unsecured/SSTI -#}
  Questionnaire name: "{{ ctx.questionnaireName }}"

  {#
  <pre>{{ ctx|tojson(2 )}}</pre>
  #}
```

```
</body>
</html>
```

The initial step in identifying the potential for an SSTI attack was the proof-of-concept, which involved modifying the `ctx.questionnaireName` in the template to `7*7`. Upon rendering this modified template within the application's editor and previewing it, the output displayed `49`. This result indicated that the template engine was executing the embedded Python code, confirming the possibility of exploiting this behavior.

To further demonstrate the exploitability of this vulnerability, I introduced an additional line into the template, which was created introduced by Pwn-Function in the “Server-Side Template Injections Explained” video [24]:

```
<p>Script : "{ { '.__class__.__base__.__subclasses__()[141]().__i_
↪ nit__.__globals__['sys'].modules['os'].popen("id").read()
↪ } }"␣\p>
```

This complex line of code exploits Python's introspection capabilities. It starts with an empty string (a Python object), accesses its class, then its base class, and navigates through the subclasses to find a specific class that allows importing the `sys` module. From there, it directly accesses the `os` module to execute the `popen` function with the command `id`. The execution of this command reveals sensitive information about the server's environment:

```
Script : "uid=1000(user) gid=100(users) groups=100(users) "
```

The successful execution of arbitrary Python code via the template engine reveals a substantial Server-Side Template Injection (SSTI) vulnerability in DSW. This vulnerability enables an attacker to carry out system-level commands, which could result in unauthorized access to sensitive data, alteration of server operations, or further exploitation of the system.

6.2.10 File Inclusion

As previously discussed, there are two main types of file inclusion attacks: Local File Inclusion (LFI) and Remote File Inclusion (RFI). In the Data Stewardship Wizard's architecture, file uploads are limited to a user's local computer, with no provision for specifying external URLs for file imports. This design significantly mitigates the risk of Remote File Inclusion attacks. Therefore, my testing focused on the security of local file uploads, especially examining the validation process for knowledge model uploads, a critical functionality of DSW.

To thoroughly evaluate the security of knowledge model file uploads, I utilized the upload scanner feature in Burp Suite Professional. This tool is adept at scrutinizing file upload handling and pinpointing potential vulnerabilities. The initial step was to intercept the HTTP POST request responsible for knowledge model uploads `POST /package/bundle HTTP/1.1`, which contains the data of the knowledge model being uploaded. After capturing this request, I rerouted it to Burp Suite's upload scanner and dropped the original

request, enabling an in-depth analysis of DSW's processing and validation of the uploaded knowledge model files.

For a comprehensive assessment, I chose to activate all options in the upload scanner's configuration. While this method was resource-intensive, it was essential for a detailed examination.

The testing proved to be quite demanding in terms of resource usage, consuming considerable CPU power for over an hour. Nonetheless, this extensive analysis was vital for a complete evaluation of the file upload feature. At the end of the test, the upload scanner did not detect any major issues or vulnerabilities in DSW's handling of knowledge model file uploads. This result indicates that the application's mechanisms for processing and validating uploaded files are robust, effectively countering the threats tested.

The lack of vulnerabilities found in the knowledge model upload process is an encouraging sign of DSW's defence against file inclusion attacks, particularly regarding local file uploads. This insight adds to the overall understanding of the application's security stance, especially in terms of managing user-supplied content.

Evaluate

In this chapter, I summed up the results of the test that were performed on the Data Stewardship Wizard. Those evaluations will include a description of found vulnerabilities, their severity according to the base Common Vulnerability Scoring System (CVSS), and the prioritized list of recommendations for DSW.

7.1 The Common Vulnerability Scoring System (CVSS)

The Common Vulnerability Scoring System (CVSS) is a universally recognized and open standard for evaluating the severity of security vulnerabilities in computer systems. It plays a crucial role in determining the urgency and priority of responses to these vulnerabilities.

CVSS offers a systematic approach to capture the key characteristics of a vulnerability and compute a numerical score reflecting its severity. This score is then converted into a qualitative representation (like low, medium, high, and critical), aiding organizations in effectively assessing and prioritizing their vulnerability management efforts.

CVSS has evolved through several versions, with CVSS v3.1 being the standard nowadays. Each iteration enhances the previous one, providing more precise and comprehensive methods for vulnerability assessment.

CVSS scores are derived from various metrics, and categorized into three groups:

Base Metrics: These metrics reflect the inherent qualities of a vulnerability that remain constant over time and across different user environments. They include:

- **Attack Vector (AV):** The means by which the vulnerability can be exploited (e.g., locally, adjacent network, network).
- **Attack Complexity (AC):** The complexity involved in exploiting the vulnerability.
- **Privileges Required (PR):** The level of privileges needed for an attacker to exploit the vulnerability.
- **User Interaction (UI):** The necessity of user interaction for exploiting the vulnerability.

- **Scope (S):** The potential of a vulnerability in one component to affect other components beyond its security scope.
- **Confidentiality (C), Integrity (I), and Availability (A) Impact:** These assess the extent to which the vulnerability affects these three fundamental aspects of information security.

Temporal Metrics: These metrics vary over time but not among different user environments. They include metrics like Exploit Code Maturity, Remediation Level, and Report Confidence.

Environmental Metrics: These are tailored to the significance of the affected IT asset within an organization, factoring in existing mitigations. They encompass metrics such as Collateral Damage Potential, Target Distribution, and the requirements for Confidentiality, Integrity, and Availability.

However, as I used a locally deployed version of the application, I will use only the base metrics to define the severity. The CVSS score is computed using a formula that incorporates the above metrics. The final score ranges from 0 to 10, with higher scores indicating more severe vulnerabilities.

The score is represented both numerically and qualitatively:

- **0.0:** None
- **0.1-3.9:** Low
- **4.0-6.9:** Medium
- **7.0-8.9:** High
- **9.0-10.0:** Critical

7.2 Summary

7.2.1 Weak Credential Policy - Password Strength

Description: Allow users to assign weak passwords to their accounts, which can later be easily found by an attacker through brute-force or dictionary attacks [9].

Vector String :

CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:L/A:N

Base Score: 5.3

Base Severity: Medium

The assigned base score of 5.3 and a severity rating of Medium reflect the potential impact of this vulnerability. While it does not directly compromise system confidentiality or availability, it can undermine the integrity of the system by allowing unauthorized access. This vulnerability notably enhances the risk of successful brute-force attacks, as discussed in Section 6.2.2. It is crucial for systems to enforce strong password policies to mitigate such risks effectively.

7.2.2 Password Brute-Force Attack

Description: This vulnerability was identified and exploited through a successful brute-force attack on a password using HTTP requests. The attacker was able to crack the password, demonstrating a significant security weakness in the system's authentication mechanism. The lack of adequate protection against brute-force attacks, such as account lockout policies or rate limiting, allowed for repeated and unchecked password-guessing attempts, leading to unauthorized access.

Vector String :

CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N

Base Score: 7.5

Base Severity: High

The calculated base score of 7.5 with a severity rating of High underscores the critical nature of this vulnerability. The high impact on confidentiality (C:H) indicates that successful exploitation of this vulnerability could lead to significant unauthorized disclosure of information. However, the integrity (I:N) and availability (A:N) of the system remain unaffected. This vulnerability highlights the importance of implementing robust security measures against brute-force attacks, such as strong password policies, rate limiting, and account lockout mechanisms, to prevent unauthorized access to sensitive information.

7.2.3 User Enumeration

Description: This vulnerability is identified in the signup process of an application, where it explicitly indicates if an email address is already in use. This behavior can be exploited to compile a list of valid email addresses registered with the application. When combined with the previously noted ease of performing a brute-force password attack, this issue poses a significant risk. Attackers can use this information to target specific accounts for further attacks, including brute-force attempts to crack passwords.

Vector String :

CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:N/

Base Score: 5.3

Base Severity: Medium

The base score of 5.3 and a severity rating of Medium reflect the potential risk posed by this vulnerability. The primary impact is on confidentiality (C:L), as it allows attackers to confirm the existence of specific user accounts. However, the integrity (I:N) and availability (A:N) of the system are not directly compromised by this vulnerability. This vulnerability underscores the need for careful consideration of information disclosure in application design, particularly during processes like user registration, to prevent aiding potential attackers in their efforts.

7.2.4 Email Verification Denial of Service

This vulnerability occurs in the application's signup process. An attacker can exploit this by creating a large number of unverified accounts using legitimate email addresses. Since there is no deadline for email verification, these accounts remain unverified indefinitely, leading to the exhaustion of system resources. Additionally, this can prevent real users from registering with their own email addresses, as their emails are already tied to these unverified accounts. This attack can effectively deny service to legitimate users trying to sign up.

Vector String :

CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H

Base Score: 7.5

Base Severity: High

With a base score of 7.5 and a severity rating of High, this vulnerability is significant due to its impact on the availability of the service. The high availability impact (A:H) reflects the potential for the system to be rendered unusable for legitimate users attempting to register. This underscores the importance of implementing effective controls in the signup process, such as setting a verification deadline or limiting the number of accounts that can be created from a single IP address, to prevent such denial-of-service scenarios.

7.2.5 Broken Access Control

In Chapter 6.2.7, I identified 13 discrepancies between API access and the graphical user interface (GUI) in the Data Stewardship Wizard. However, due to my limited knowledge of the intended access requirements, it's challenging to definitively classify these discrepancies as vulnerabilities or as intentional design choices. Consequently, assessing the severity of these potential issues is not feasible within the scope of my analysis.

Recommendation for the DSW Team:

It is imperative for the DSW team to carefully review the list of identified discrepancies. This review is crucial to ascertain whether these discrepancies align with the intended functionality of the application or if they represent significant security concerns. Given the other vulnerabilities discovered during the testing process, any unintended discrepancies could pose serious risks to the application's security. If these issues are indeed vulnerabilities, they could be exploited in conjunction with other identified weaknesses, potentially leading to more severe security breaches. Therefore, a thorough examination and validation of these discrepancies are essential to ensure the overall security and integrity of the DSW application.

7.2.6 Access Tokens in URL

This vulnerability involves the exposure of a user's JWT (JSON Web Token) in the URL of a web request. The token was found to be transmitted in plain text, without any form of protection or encryption. This insecure practice can lead to several security risks, such as unauthorized access and data exposure.

7.3. Prioritized List of Recommendations for DSW

Tokens in URLs can be easily intercepted, logged in server logs, or stored in browser histories, making them vulnerable to being captured by malicious actors. This vulnerability is particularly concerning as JWTs are often used for authentication and can grant access to sensitive user data and functionalities.

Vector String :

CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:L

Base Score: 5.3

Base Severity: Medium

The calculated base score of 5.3 and a severity rating of Medium reflect the potential risks associated with this vulnerability. The impacts on confidentiality (C:L) and availability (A:L) are rated as low, indicating that while there is a risk of unauthorized data access and potential data manipulation, it may not lead to severe consequences. However, the integrity (I:N) is not directly impacted by this vulnerability. This issue highlights the importance of secure transmission practices for sensitive tokens, such as using HTTP headers for JWTs instead of URL parameters and ensuring encryption through HTTPS to protect data in transit.

7.2.7 Server-Side Template Injection

This vulnerability is a case of Server-Side Template Injection (SSTI) where an attacker successfully executed arbitrary Python code on the server using the Jinja2 template engine. By exploiting this vulnerability, the attacker was able to print out system information, demonstrating the ability to execute unauthorized commands on the server. SSTI vulnerabilities like this can lead to various malicious activities, including data theft, website defacement, and in more severe cases, complete server takeover. The ability to execute code on the server poses a significant security risk, as it could potentially allow access to sensitive data and system resources.

Vector String :

CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:L

Base Score: 6.3

Base Severity: Medium

The base score of 7.1 with a severity rating of High reflects the serious impact of this vulnerability. The high impact on confidentiality (C:H) indicates that successful exploitation could lead to significant unauthorized disclosure of sensitive data. However, to perform the attack, the user has to have permissions to be able to import the template.

7.3 Prioritized List of Recommendations for DSW

Based on the vulnerabilities identified in the Data Stewardship Wizard, I have compiled a prioritized list of recommendations. These suggestions are organized according to the severity ratings determined using the Common Vulnerability Scoring System.

7.3.1 High Severity Vulnerabilities

- **Password Brute-Force Attack (CVSS Score: 7.5)**
 - Implement account lockout policies or rate limiting.
 - Introduce CAPTCHA mechanisms.
- **Email Verification Denial of Service (CVSS Score: 7.5)**
 - Set a deadline for email verification.
 - Limit account creation from a single IP address.

7.3.2 Medium Severity Vulnerabilities

- **Server-Side Template Injection (CVSS Score: 6.3)**
 - Sanitize and validate inputs to the Jinja2 template engine.
 - Restrict template engine permissions.
- **Weak Credential Policy - Password Strength (CVSS Score: 5.3)**
 - Enforce strong password policies.
- **User Enumeration (CVSS Score: 5.3)**
 - Use generic error messages for account processes.
- **Access Tokens in URL (CVSS Score: 5.3)**
 - Transmit JWTs in HTTP headers.
 - Ensure encryption of sensitive data in transit.

7.3.3 Other Considerations

- **Broken Access Control**
 - Review and strengthen access control policies.
- **Potential False Positives**
 - Verify findings with the development team.
- **Comprehensive Security Review**
 - Address all identified issues.
- **Continuous Monitoring and Testing**
 - Regularly update security measures.
 - Conduct periodic vulnerability assessments.

7.3. Prioritized List of Recommendations for DSW

Additional Considerations: Some flagged requests might be intentionally designed features of the application. It is essential to confirm these findings with the development team to discern if they represent actual vulnerabilities or deliberate functionalities.

While CVSS scores are useful for prioritizing vulnerabilities, it is vital to address all identified issues to improve DSW's overall security. Also, it is very important to keep in mind, that regular updates to security protocols and periodic vulnerability assessments are crucial to keep pace with emerging threats and maintain a strong security stance.

Conclusion

As this thesis comes to a close, reflecting on the in-depth security analysis of the Data Stewardship Wizard brings a sense of accomplishment and a deeper understanding of the nuances of cybersecurity. This journey has shed light on the intricate and vital role of cybersecurity in data stewardship and management.

My analysis revealed a spectrum of vulnerabilities in DSW, from weaknesses in password policies to more complex risks like server-side template injection. Evaluating these vulnerabilities through the Common Vulnerability Scoring System has not only identified key security gaps but also helped prioritize them for effective remediation. This process has set the stage for enhancing DSW's defence mechanisms.

The recommendations proposed in this thesis, while addressing specific vulnerabilities in DSW, also serve as guidelines for broader security enhancements. These include strengthening password protocols, improving user input security, and enforcing rigorous access controls. As the field of cybersecurity is dynamic, ongoing vigilance and adaptation to new threats are essential.

This project has reignited my passion for cybersecurity. Far from being a dry, technical exercise, it has been an engaging and thought-provoking experience. Navigating through the complexities of digital security has renewed my interest in the field, highlighting its ever-changing and challenging nature. It's been a reminder that cybersecurity isn't just about technical skills but also involves creativity and continuous learning.

In sum, this thesis underscores the critical importance of cybersecurity in today's digital landscape. It emphasizes that security is not a one-off task but an ongoing process of learning, adapting, and enhancing. Through this work, I aim not only to bolster the security of DSW but also to encourage a broader appreciation of cybersecurity as an exciting and essential field, full of challenges and opportunities for innovation. This exploration into DSW's security has been more than an academic endeavour; it's been a journey of rediscovery and a reaffirmation of the excitement inherent in the world of cybersecurity.

Bibliography

- [1] Amazon. *What is SQL?* <https://aws.amazon.com/what-is/sql/>. Accessed: 2023-12-02.
- [2] Anshita Bhasin. *Most commonly used API tokens: Detailed Guide*. <https://medium.com/@anshita.bhasin/most-commonly-used-tokens-detailed-guide-51133cec1e49>. Accessed: 2024-01-01.
- [3] Baeldung. *The DTO Pattern*. <https://www.baeldung.com/java-dto-pattern>. Accessed: 2024-01-01.
- [4] Barker, M., Chue Hong, N.P., Katz, D.S. *Introducing the FAIR Principles for research software*. <https://doi.org/10.1038/s41597-022-01710-x>. Accessed: 2024-01-01.
- [5] National Cyber Security Centre. *Understanding vulnerabilities*. <https://www.ncsc.gov.uk/information/understanding-vulnerabilities>. Accessed: 2024-01-01.
- [6] CLOUDFLARE. *What is HTTPS*. <https://www.cloudflare.com/learning/ssl/what-is-https>. Accessed: 2024-01-01.
- [7] CLOUDFLARE. *What is rate limiting*. <https://www.cloudflare.com/learning/bots/what-is-rate-limiting>. Accessed: 2023-11-05.
- [8] DSW. *FAIR*. <https://ds-wizard.org/fair>. Accessed: 2024-01-01.
- [9] fluidattacks. *Weak credential policy - Password strength*. <https://docs.fluidattacks.com/criteria/vulnerabilities/363/>. Accessed: 2023-12-29.
- [10] Gartner. *GUI (Graphical User Interface)*. <https://www.gartner.com/en/information-technology/glossary/gui-graphical-user-interface>. Accessed: 2023-12-26.
- [11] imperva. *Cross site request forgery (CSRF) attack*. <https://www.imperva.com/learn/application-security/csrf-cross-site-request-forgery/>. Accessed: 2024-01-01.
- [12] Michael Jones, John Bradley, and Nat Sakimura. *JSON Web Token (JWT)*. <https://datatracker.ietf.org/doc/html/rfc7519>. Accessed: 2023-11-05. 2015.
- [13] LinkedIn. *How do you deal with OAuth token leakage or theft on your resource server?* <https://www.linkedin.com/advice/0/how-do-you-deal-oauth-token-leakage-theft-your-resource-server>. Accessed: 2023-12-02.

BIBLIOGRAPHY

- [14] malwarebytes. *What is Exploit Protection*. <https://support.malwarebytes.com/hc/en-us/articles/360038523394-What-is-Exploit-Protection>. Accessed: 2024-01-01.
- [15] okta. *What is MD5? Understanding Message-Digest Algorithms*. <https://www.okta.com/identity-101/md5/>. Accessed: 2023-11-05. 2022.
- [16] OWASP. <https://en.wikipedia.org/wiki/OWASP>. Accessed: 2024-01-01.
- [17] OWASP. *C9: Implement Security Logging and Monitoring*. <https://owasp.org/www-project-proactive-controls/v3/en/c9-security-logging>. Accessed: 2023-11-05. 2018.
- [18] OWASP. *Cross Site Scripting (XSS)*. <https://owasp.org/www-community/attacks/xss/>. Accessed: 2023-12-02. 2022.
- [19] OWASP. *Input Validation Cheat Sheet*. https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html. Accessed: 2023-11-04.
- [20] OWASP. *OA01:2021 – Broken Access Control*. https://owasp.org/Top10/A01_2021-Broken_Access_Control/. Accessed: 2023-12-02. 2021.
- [21] OWASP. *OWASP Top 10 API Security Risks – 2023*. <https://owasp.org/API-Security/editions/2023/en/0x11-t10/>. Accessed: 2023-12-27.
- [22] OWASP. *OWASP Top Ten 2017*. https://owasp.org/www-project-top-ten/2017/A2_2017-Broken_Authentication. Accessed: 2023-12-02. 2017.
- [23] OWASP. *Testing for Local File Inclusion*. https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/07-Input_Validation_Testing/11.1-Testing_for_Local_File_Inclusion. Accessed: 2023-12-02.
- [24] PwnFunction. *Server-Side Template Injections Explained*. <https://youtu.be/SN6EVIG4c-0?si=pANYhLT0Sub9O4dh>.
- [25] Robert Pergl, Robert Pergl, Rob Hooft, Marek Suchánek, Vojtěch Knaisl, Jan Slifka. “*Data Stewardship Wizard: A Tool Bringing Together Researchers, Data Stewards, and Data Experts around Data Management Planning*”. <https://datascience.codata.org/articles/10.5334/dsj-2019-059>. Accessed: 2023-09-29.
- [26] sentry. *Get Started with Sentry*. <https://docs.sentry.io/product/sentry-basics>. Accessed: 2023-11-05.
- [27] Sqlmap. *Introduction*. <https://sqlmap.org/>. Accessed: 2023-12-27.
- [28] Data Stewardship Wizard. *Data Stewardship Wizard*. <https://ds-wizard.org/data-stewardship>. Accessed: 2023-10-29.

Acronyms

2FA Two-Factor Authentication.

AES Advanced Encryption Standard.

API Application Programming Interface.

BAC Broken Access Control.

CAPTCHA Completely Automated Public Turing test to tell Computers and Humans Apart.

CRUD Create, Read, Update, Delete.

CSRF Cross-Site Request Forgery.

CTR Counter.

CVSS Common Vulnerability Scoring System.

DB Database.

DDoS Distributed Denial of Service.

DM Document Model.

DMP Data Management Plan.

DoS Denial of Service.

DOM Document Object Model.

DTO Data Transfer Object.

DSW Data Stewardship Wizard.

FAIR Findable, Accessible, Interoperable, and Reusable.

GUI Graphical User Interface.

HTTPS Hypertext Transfer Protocol Secure.

HTTP Hypertext Transfer Protocol.

A. ACRONYMS

ID Identifier.

IO Input/Output.

IT Information Technology.

JWT JSON Web Token.

KM Knowledge Model.

LFI Local File Inclusion.

MD5 Message-Digest Algorithm 5.

MFA Multi-Factor Authentication.

NIST National Institute of Standards and Technology.

OOB Out-of-Band.

OWASP Open Web Application Security Project.

RFI Remote File Inclusion.

SQL Structured Query Language.

SSTI Server-Side Template Injection.

SSL Secure Sockets Layer.

TSL Transport Layer Security.

TDK Template Development Kit.

URL Uniform Resource Locator.

UUID Universally Unique Identifier.

VNC Virtual Network Computing.

VPN Virtual Private Network.

XSS Cross-Site Scripting.

Contents of Electronic Attachment

- | README.md file with contents description
- | thesis.pdf thesis text in PDF format
- | src/ directory of LaTeX source codes of the thesis
- | dsw-v3.28/ used DSW Docker deployment and configuration
- | results/ directory with other results related to security analysis
 - | results.pdf file with found vulnerabilities and their evaluation
 - | http_requests.pdf file with all executed HTTP requests for BAC