Hindawi
Wireless Communications and Mobile Computing
Volume 2021, Article ID 8028073, 13 pages
https://doi.org/10.1155/2021/8028073

Research Article

Securing Open Banking with Model-View-Controller Architecture and OWASP

Deina Kellezi, Christian Boegelund, and Weizhi Meng

Department of Applied Mathematics and Computer Science, Technical University of Denmark, Denmark

Correspondence should be addressed to Weizhi Meng; yuxin.meng@my.cityu.edu.hk

Received 3 August 2021; Accepted 3 September 2021; Published 21 September 2021

Academic Editor: Ximeng Liu

In 2015, the European Union passed the PSD2 regulation, with the aim of transferring ownership of bank accounts to the private person. As a result, Open Banking has become an emerging concept, which provides third-party financial service providers open access to bank APIs, including consumer banking, transaction, and other financial data. However, such openness may also incur many security issues, especially when the data can be exposed by an API to a third party. Focused on this challenge, the primary goal of this work is to develop one innovative web solution to the market. We advocate that the solution should be able to trigger transactions based on goals and actions, allowing users to save up money while encouraging positive habits. In particular, we propose a solution with an architectural model that ensures clear separation of concern and easy integration with Nordea's (the largest bank in the Nordics) Open Banking APIs (sandbox version), and a technological stack with the microframework Flask, the cloud application platform Heroku, and persistent data storage layer using Postgres. We analyze and map the web application's security threats and determine whether or not the technological frame can provide suitable security level, based on the OWASP Top 10 threats and threat modelling methodology. The results indicate that many of these security measures are either handled automatically by the components offered by the technical stack or are easily preventable through included packages of the Flask Framework. Our findings can support future developers and industries working with web applications for Open Banking towards improving security by choosing the right frameworks and considering the most important vulnerabilities.

1. Introduction

Traditional banks often run their services independently and maintain their own users, while it is hard to obtain the data from other customers. Such data obstacle restricts many services such as product and service innovations and business operation across different banks (e.g., money transfer) [1]. From October 2005, a revised Payment Services Directive (PSD2) has been adopted in Europe aiming to enhance the development and use of innovative online and mobile payments through giving consumers more choice and higher security for online payments in the EU. Open Banking refers to the practice of securely sharing financial data, based on the customer consent. The data exchange between the bank and authorized third parties is enabled via Application Programming Interfaces (APIs). With the radical transformation of financial sector and new regulations, banks are demanded to develop Open Banking APIs that enable the following two properties: (1) securing access to bank account data and information and (2) allowing transactions to be completed among different accounts. As a result, Open Banking is an important sharing data solution with the aim of eliminating barriers to data access while increasing the customers' control over their data.

1.1. Motivation. One of the largest banks in the Nordics, Nordea, released the first version of their Open Banking API in January 2019. We notice that they also released a sandbox version that allows potential third-party providers to access the APIs in a test environment. Online banking applications are one of the most lucrative targets for attacks, although many have been mitigated through Nordea's own protocols, i.e., the production APIs require a multistep authentication through nemID (a common log-in solution for Danish Internet banks) when signing up. However, breaking into an application, by gaining access to a user's password, can give intruders direct access to triggering transactions. The application security itself, on a range of different areas such as data storage, injections, and communication, should therefore be considered very important to mitigate during development, as this can easily result in data breaches.

After more investigation, we find that 3300 developers are currently registered as developers for Nordea Open Banking, but only one product has been realized so far. The adoption of Open Banking exposes data to more actors than ever before, especially new companies and startups, and therefore, also an enlargement of the security risks that the financial industry is facing, with existing risks being increased and new risks being introduced [2]. Further, the threat becomes higher when leveraging applications on a web platform, with possibly insecure protocols that might not be possible on a desktop or phone application.

1.2. Contribution. Due to the complicated process of obtaining a financial license to use actual production data, in this work, we collaborate with Nordea Bank (Denmark) and delimit the problem by using only their sandbox version to develop the solution of triggering transactions based on users' habits and model the potential risks and threats. In this work, we first identify the background of a technology stack that can be used for development support. Then, we develop a web application that can enable persistent data storage and a high level of security and explain the system architecture and the API communication. The OWASP Top 10 list of the Ten Most Critical Web Application Security Risks methodology is used to investigate the potential threats and risks. Based on the identified threats, we also suggest the integration of Bcrypt algorithm [3] for storage security, which uses a 128-bit salt and encrypts a 192-bit magic value. Our contributions can be summarized as below.

(i) We investigate the Nordea Open Banking APIs by collaborating with Nordea Bank in Denmark, regarding access authorization, account information services, and payment initialization services

(ii) We then design a web application and introduce the system architecture based on the Model-View-Controller architecture (MVC), including three parts such as model, controller, and view. Our approach can handle the API integration through an abstraction layer with the MVC

(iii) To identify potential risks and threats, we use the methodology of OWASP Top 10 with a threat modelling method for categorizing the threats in six different areas, such as threat agents, exploitability, weakness prevalence, weakness detectability, technical impacts, and business impacts

(iv) Our results found that many security threats like Broken Authentication can be handled automatically by the components offered by the technical stack or can be easily preventable through included packages of the Flask Framework. However, TLS Layer in Nordea's Open Banking API may cause some crashes with HTTPS

In comparison to the previous study [4], this work provides more information on Nordea's Open Banking API, such as sequence diagram, access authorization flow, account information, and payment initialization, and introduces the OWASP Risk Rating Methodology in more detail.

The rest of this work is organized as follows. Section 2 introduces the basic background of the Flask Framework, cloud application platform, OWASP Top 10, database management, hashing and salting, and the Nordea's Open Banking API. Section 3 reviews the related work, and Section 4 details our proposed web application, including the architecture and object relational database. Section 5 identifies and discusses the potential risks and threats of Nordea's Open Banking API and our proposed application by leveraging the OWASP Top 10 list. We conclude this work in Section 6.

2. Background

In this work, we adopt the microframework for web development, Flask (for Python) to develop our web application. The Flask can mitigate many security threats by default, supplemented by a number of renowned third-party extensions and packages authenticated by the Flask community, which can be customizable according to the demands. It also provides out-of-the-box abstraction layers for communicating with the popular object relational database-PostgreSQL [5] and the cloud application platform-Heroku [6] for deployment.

2.1. The Flask Framework. A Flask application is initialized by creating an application instance through the Flask class with the application package as argument. The web server then passes all received requests from clients, such as web browsers, to this application instance. The logic is handled by using the Web Server Gateway Interface (WSGI) as protocol, through constantly awaiting requests. The framework is compliment with the WSGI server standard [7].

The application instance also needs to know which part of the logic has to run for each requested URL. This is done through a mapping of URLs to the Python functions, which handle the logic associated with a URL. This association is called route between the URL and the handling function, which can be defined by the @package.route decorator. The return value of the function is the response that the client received in the form of a template or a redirect.

2.2. Cloud Application Platform. Heroku [6] is one of the first and largest PaaS (Platform as a Service) providers with their Cloud Application Platform. The developer can deploy an application to Heroku using Git to first clone the source code from the developer branch and then push the application to the Heroku Git server. The command automatically triggers the installation, configuration, and deployment of the application. The platform uses units of computing and dynos to measure the usage of service and perform for different tasks on the server. It also provides a large number of plugins and add-ons for databases, email support, and many other services. Heroku supports PostgreSQL [5] databases as an add-on, created and configured through the command line client.

2.3. Database Management. The Flask puts no restriction on what database packages can be used and supports a number of different database abstraction layer packages. The web application can run on the PostgreSQL database engine supported by the Object Relational Mapping (ORM) and SQLAlchemy. The selection is based on the following different criteria:

(i) *Easy Usage*. Using a database abstraction layer (object-relational mappers ORMs) such as SQLAlchemy provides transparent conversion of high-level object-oriented operations into low-level database instructions, in comparison to writing raw SQL statements [8]

(ii) *Performance*. ORM conversions can result in a small performance penalty, yet the productivity gain far outweighs the performance degradation. The few outlying queries that degrade the performance can be subsidized by raw SQL statements

(iii) *Portability*. The application platform-Heroku can support a number of different database engine choices, where the most popular and extensible ones are Postgres and MySQL [6]

(iv) *Integration*. The Flask includes several packages designed to handling ORMs, such as Flask-SQLAlchemy [9], which includes engine-specific commands to handle connection

2.4. OWASP Top 10. The Open Web Application Security Project (OWASP) [10] is a worldwide organization focused on improving the security of software. The OWASP has identified a list of the Ten Most Critical Web Application Security Risks that can be used for vulnerabilities mapping, which include:

(1) *Injection*. Injection flaws, such as SQL and ORMs, occur when untrusted data is sent to a field as part of a command or query. The attacker's hostile statements can trick the backend into executing unintended commands

(2) *Broken Authentication*. Application functions related to authentication and session management are often missed or implemented incorrectly, allowing attackers to compromise passwords or session tokens, or to exploit other implementation flaws to infer the user's identity

(3) *Sensitive Data Exposure*. Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and personally identifiable information (PII). Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes without encryption

(4) *XML External Entities (XXE)*. Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files

(5) *Broken Access Control*. Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality or data, such as other user's accounts or access rights

(6) *Security Misconfigurations*. Security misconfiguration is the most commonly posed issue. This is commonly a result of insecure default or manual configurations, open cloud storage, misconfigured HTTP headers, and error messages or stack traces containing sensitive data

(7) *Cross-Site Scripting (XSS)*. XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping. XSS allows attackers to execute scripts in the victim's browser, which can hijack user sessions, deface websites, or redirect the user to malicious sites

(8) *Insecure Deserialization*. Insecure deserialization can lead to remote code execution. Even if deserialization flaws do not result in this, it can be used to perform a different number of attacks suck as replay attacks, injection attacks, and privilege escalation attacks

(9) *Using Components with Known Vulnerabilities*. Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If one of these is vulnerable and exploited, it can facilitate data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts

(10) *Insufficient Logging and Monitoring*. Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allow attackers to further intrude systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data

2.5. Hashing and Salting of Sensitive Data. When signing up for the application, the user will need to provide a password and load account data, inevitably sensitive information such as account number. Generally, hash algorithm can be used to securely saving passwords and account numbers on the server side.

2.5.1. Hash Algorithm. Hash algorithm refers to a one-way mathematical function that takes data with an arbitrary length and maps it to a fixed length bit string. The purpose of a hash algorithm is to store the sensitive data securely in the database, simultaneously confirming that the provided password or account number is correct. A good hash algorithm should hold the following properties [11]:

(i) *Preimage Resistance*. For a given h in the output space of the hash function, it is hard to find any message x with $H(x) = h$

(ii) *Second Preimage Resistance*. For a given message x_1, it is hard to find a second message $x_2 \neq x_1$ with $H(x_1) = H(x_2)$

(iii) *Collision Resistance*. It is hard to find a pair of messages $x_1 \neq x_2$ with $H(x_1) = H(x_2)$

There are a number of different hash algorithms, all with different properties. A selected number of hashing algorithms, for instance, the MD5 algorithm or the SHA1 algorithm, are designed to be fast and efficient. This is preferable when messages should be hashed quickly to check for equality.

However, in terms of passwords, account numbers, or other sensitive information, fast hash algorithms are not always optimal. If there is a security breach that allows attackers gaining access to the data in the database, they can quickly be breached using fast hash algorithms. In particular, MD5 is not recommended, as it is unsalted. This can be done with a precomputed lookup table, also known as a rainbow table. Further, these algorithms can be accelerated significantly by using a GPU [3].

On the other hand, slower hash algorithms such as bcrypt initially create a slower run time. However, when it comes to precomputing hashing values, it is much more difficult, as the algorithm is designed to be slower by an order of magnitude. Brute-forcing the data is therefore way more difficult.

2.5.2. Salting a Hash. There is an observation that users may often choose weak passwords due to the long-term memory limitation [12, 13]. For example, top 10,000 passwords are used by 30% of all users [14]. Even if one were to use bcrypt, a slow hash algorithm, passwords can be quickly be compromised via a parallel set of GPUs. Because of this, we need to enhance the password strength. Salting is an effective technique for this purpose, which entails adding a random string to the beginning or end of the password before hashing. In practice, we have to make the password almost impossible to crack with current technology. A short calculation shows that it is infeasible to guess a salt of 12 characters. Even if we constrict passwords to letters only (a-z and A-Z), of which there are 52 characters, we are able to create: $12^{52} = 1.3 \cdot 10^{56}$ different salts. As a result, even with strong computer power, it is infeasible to guess the salt even with the password. If we were able to check 100 billion salts per second, it needs to cost us: $3.17 \cdot 10^{37}$ years to guess it. In comparison, the age of the universe is around $1.38 \cdot 10^{10}$.

2.6. The Nordea Open Banking API. The Nordea API provides access to a number of different endpoints in order to facilitate the connection to the accounts of the user. Some API endpoints must be used in order to authenticate the user before changing the data, while other endpoints involve a number of side effects, i.e., changing the balance on the accounts [15]. We check a list of the relevant endpoints with Nordea Bank (Denmark), and the following are the most crucial ones:

(i) Access authorization

(ii) Account information services

(iii) Payment initialization services

2.6.1. Access Authorization. To leverage the functionality of the API, the Client ID and Client Secret must be obtained. The values can be retrieved by creating a project on the Nordea Open Banking website. The Client ID and Client Secret are parameters that are configured to the client, and they are never exposed to the actual application user. Once the account has been approved, we must obtain an access token in order to gain access to the API.

Figure 1 describes access authorization flow required to obtain the access token. The faded lines describe the OAuth flow, handling the multifactor authentication [15]. This is not part of the sandbox version of the API and will therefore not be handled in our approach. It describes the authentication flow that would be present in a production environment, i.e., users with actual accounts using applications that require a NemID authentication (NemID users are assigned a unique ID number that can be used as a username in addition to their CPR-Number or a user-defined username).

2.6.2. Account Information. The account information API includes the possibility to check the contents of the different sample accounts in the sandbox version. We can create new accounts, delete current accounts, and add funds to relevant accounts. This can be done by sending a request to the account endpoint [15]. Based on the URL and the type of request, the function will be different as shown in Table 1.

The flow of account information API depends mainly on which type of request is made. Figure 2 describes two scenarios of requests made to the API. Both of them return a response code with the requested information.

2.6.3. Payment Initialization. The payment initialization API provides functionality to create payments directly in the API, moving funds from one account to another [15].

Figure 3 shows the protocol for payments between the two accounts provided in the request. The final response will confirm whether or not the transaction has been made.

3. Related Work

3.1. Web-Based Solution. In the state-of-the-art, there is few work regarding how web applications or the technical stack can integrate with Open Banking APIs. This is due to two main reasons as below:

(i) The novelty of most of the interfaces, including Nordea's Open Banking APIs

(ii) The requirements of developers need to be approved by national financial authorities when using the APIs in production

These factors have delimited the pool of possible researchers to only a few authorized third parties or those using the sandbox version. No official paper has dived into integrating with Nordea's Open Banking API as a third-party provider, nor proposed a model for an architectural model or stack that secures bank account information and transaction functionality in a web application. Nevertheless, a lot of work has generally been done in the field of web application security overall, including several models to identify, analyze, and mitigate possible security breaches under a cyber attack [16]. One example is a study in the field of web application security vulnerabilities detection that conducts a security analysis and threat modelling based on the OWASP Top 10 list and threat modelling [17].

The sandbox version of the Nordea Open Banking API was officially released at the beginning of the project in January 2019. During the attempt to generate the access_token for establishing connection before beginning the development of the application, the error codes were limited to

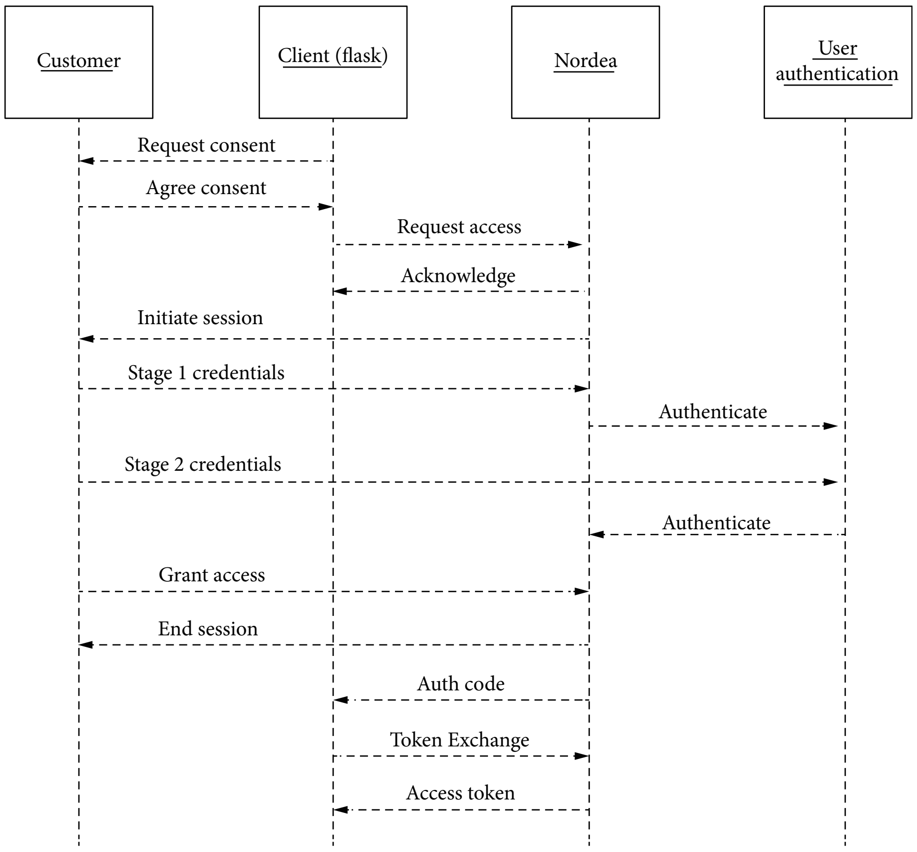


Figure 1: Sequence diagram, access authorization flow.

Table 1: The request type and the URL with relevant functions.

Type	URL	Function
GET	/v3/accounts	Information about accounts
POST	/v3/accounts	Create new account
POST	/v3/accounts/{ID}	Create transaction

generic server errors. The limited sample codes and lacking documentation on possible error codes ("The error messages are not descriptive at the moment, and this issue is noted. The error messages will be improved over time.") made it difficult to correct. In order to find a solution, we conducted a simulation with the API simulation tool named Postman [18]. The connection was successful, and the code in Postman worked and did not return any error codes. This led to the conclusion that something was wrong with our implementation of the API calls. To understand the difference between the HTTP packages, the difference between them was negligible. We contacted the senior software architect of Nordea Open Banking. The support team from Nordea Bank tried to assist us in making the API work and assess the possible errors made through logging of their own servers. Ultimately, they did not succeed in resolving the issue. The origin of the error was later found: the redirect URI, a crucial part of the OAuth (OAuth is one of the leading protocols within authentication) 2.0-process was set to an incorrect value. We thus contributed to the community of developers by using the Nordea Open Banking API and creating a pull request (The PR can refer to: https://github.com/NordeaOB/examples/pull/7). At the moment, the sample code only works with version 2 of the API, while the API has been updated to version 3 since then.

3.2. Blockchain-Based Solution. With the advent of blockchain technology, it becomes a popular solution for securing Open Banking. For example, Xu et al. [1] first identified some potential issues of Open Banking, i.e., mutual authentication is hard to be transparently managed, and Access Control List (ACL) controlled by users may pose privacy issues. Then, they introduced PPM, a Provenance-Provided Data Sharing Model for open banking system via blockchain technology, which could employ the programmable smart contracts as the middle witness between users and third-party services to guarantee the reliability and trust communication. Meanwhile, Dong et al. [19] argued that Open Banking may cause a risk of privacy leakage and personal information misconduct. They then introduced BBM, a blockchain-based self-sovereign identity system model, which allowed users to provide their digital identities in the off-line world as same as they use physical identities. Wang et al. [20] also introduced a data privacy management framework based on the blockchain technology, which could be used for Open Banking and the financial sector.

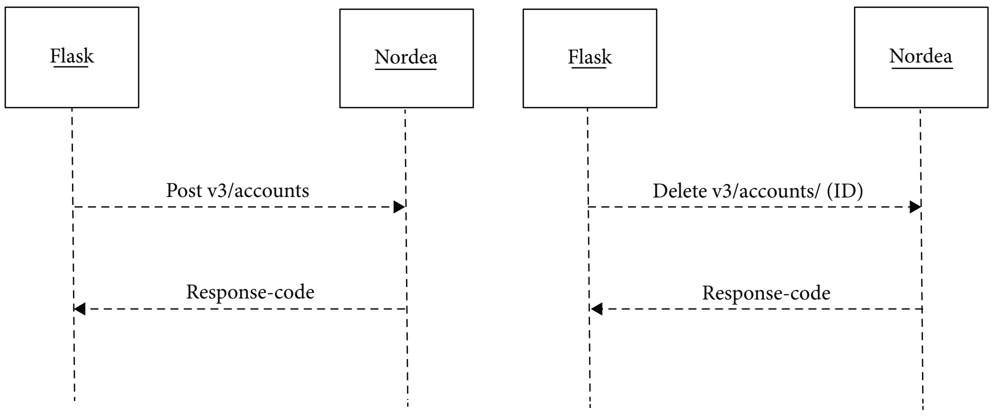


Figure 2: Sequence diagram of AIS-API.

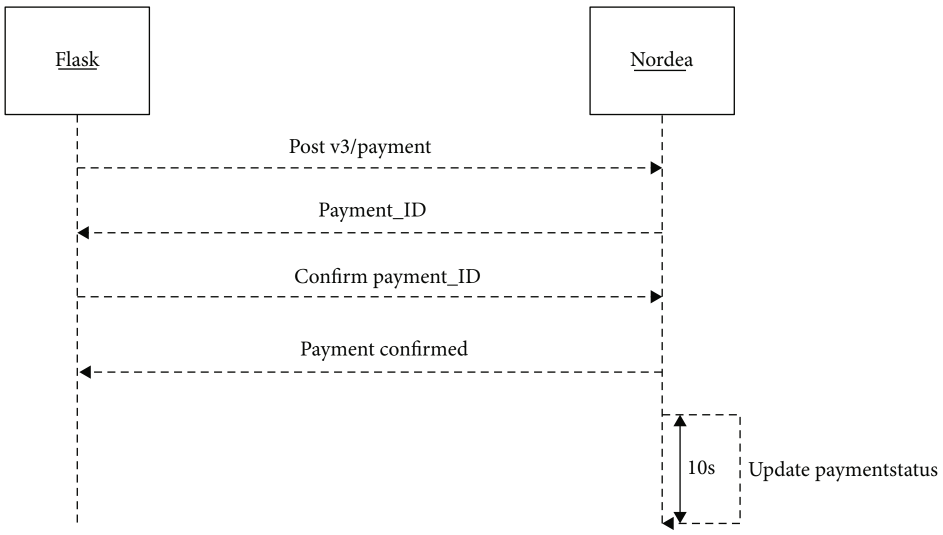


Figure 3: Sequence diagram of payment initialization service.

Due to the benefits provided by Open Banking, its data sharing model has been studied in other areas such as Electronic Health Records [21]. Hence, there is a great need to further enhance its security.

3.3. Intrusion Detection Solution. To protect the web application and open banking security, intrusion detection system (IDS) is a basic and necessary mechanism. Based on the detection approaches, it can be identified as either signature-based or anomaly-based detection. For example, an enhanced filter mechanism (EFM) [22] could be used to provide a comprehensive protection, including a context-aware blacklist-based packet filter, an exclusive signature matching component, and a KNN-based false alarm filter. Ma et al. [23] introduced a Distributed Consensus-based Trust Model to evaluate the trustworthiness of IoT nodes, against three typical attacks—tamper attack, drop attack, and replay attack, by sharing certain information. Sohi et al. [24] introduced RNNIDS that could enhance the detection performance by using Recurrent Neural Networks (RNNs) to find complex patterns in attacks and generate mutants of attacks as well as synthetic signatures. Further, an IDS can work with other security mechanisms towards an enhanced security level.

4. Our Proposed Approach

4.1. The Application Architecture. For defining the architecture, we present a model based on the Model-View-Controller architecture (MVC) specifically adjusted for web development as proposed by Pop and Altar [25]. It was found that developers often combine the HTML code with server side programming languages during the web development and create dynamic web pages and applications. This may lead to highly entangled and unmaintainable code. With an MVC pattern, it is possible to prevent cluttering by separating three overall parts of a web application, including model, controller, and view. The model will also introduce how to handle the API integration through an abstraction layer and how to include it in the MVC.

(i) *Model.* A persistent data storage layer through a data centre or database

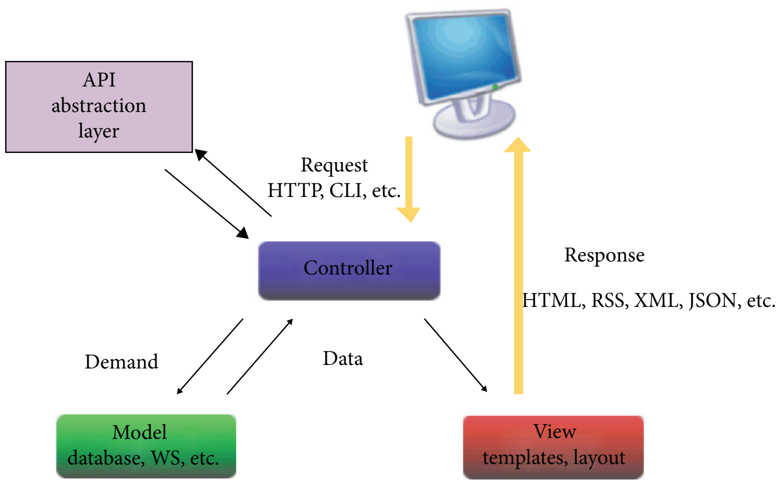


Figure 4: MVC architecture for web application.

(ii) *Controller*. The HTTP requests triggered by user actions and general routing of different subpages

(iii) *View*. The HTML code and mark-up languages in the templates rendered to the user as a result of a request

These main components will be built through a modular approach, using blueprints as recommended by the Flask. Figure 4 presents the proposed diagram for the adjusted MVC, which can be further adjusted to include supplementary components for interacting with the API. This model allows us to further propose how this fits into the Flask Framework and an effective abstraction layer integration with the API.

Figure 5 presents the schematic of the application architecture and how the MVC components are implemented.

4.1.1. The Model. As shown in Figure 5, the blue box shows the modelling of the data objects and relationship. This is the direct representation of the schema in the database. Whenever the SQLAlchemy methods start either querying, updating, or deleting data, they are called on the defined data objects in the model, and the database is updated accordingly. This also provides simpler commands for establishing connections to PostgreSQL through the URL of the database as handled by the controller.

4.1.2. The Controller. As shown in Figure 5, the green box shows the controller that is classified into three blueprints:

(i) *auth_controller*. Rendering the pages responsible for signing up and authenticating users logging in

(ii) *main_controller*. Rendering the pages of specific user session, containing URLs for creating habits, checking off habits that are completed, overview over habits, and overview over accounts and settings. This is restricted to authenticated users only

(iii) *admin_controller*. Rendering the pages of administration content included for demonstration purposes that allow to test the different API functionality. This is restricted to users with admin rights only

The blueprints provide a clearer separation of different states in the application, which could be done through application dispatching, i.e., creating multiple application objects, however, this would require separate configurations for each of the objects and management at the server level (WSGI). Blueprints instead support the possibility of separation at the Flask application level, ensuring the same application and object configurations across all controllers, and most importantly, the same API access. This means that a Blueprint object works similarly to a Flask application object, but is not an actual application as it is a blueprint of how to construct or extend the application at runtime [26]. When binding a function with the decorator @auth.route, the blueprint will record the intention of registering the associated function from the auth package blueprint on the application object. It will also prefix the name of the blueprint (given to the Blueprint constructor) auth to the function.

Each blueprint handles initialization, routing, and execution in the application. The initialization entails creating a Flask object instance by taking a specific set of configurations for either development, testing, or production environment, establishing connection to the API by obtaining the Client ID and connecting to the database. For the routing, Flask requires us to define routing functions for each of the URL routes for the web application. This allows the Flask to map the incoming request from the user to a specific response, triggering change in the state of the application in the model and rendering the template with the changed data. We have limited the requests to GET and POST methods, following a POST/REDIRECT/GET pattern. Moreover, the controllers are responsible for running the application instance through the main method provided by the Flask.

4.1.3. The View. As shown in Figure 5, the orange box shows the inheritance hierarchy of the templates that primarily consist of HTML and CSS, built upon a number of frameworks. The inheritance is supported by the Jinja2 Template Engine, offered by the Flask, enabling all templates to inherit from a base design, as well as register into their specific controller through the aforementioned blueprints. This also allows dynamic rendering of values provided as argument to the templates when rendered [26].

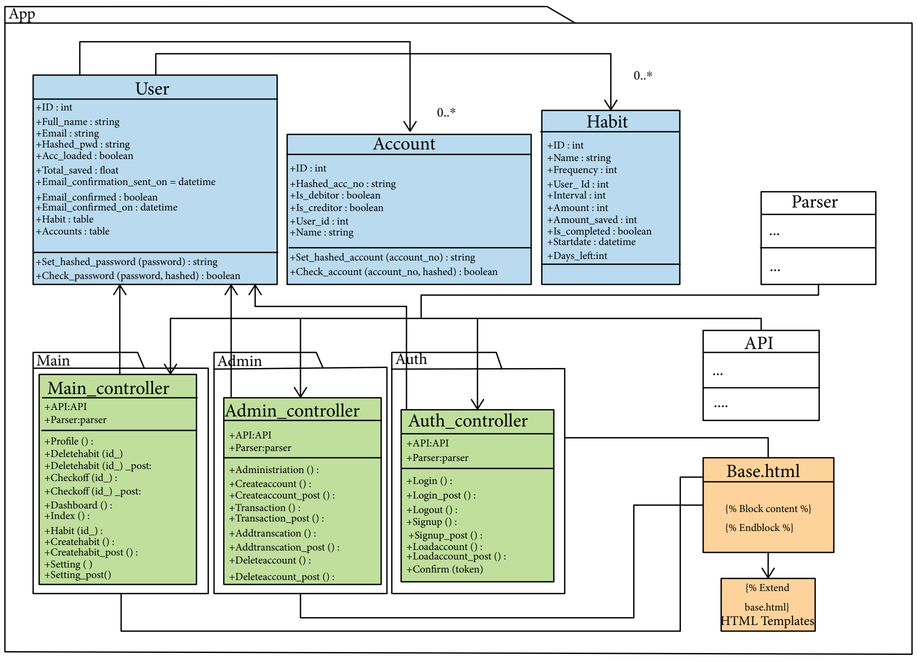


Figure 5: Class diagram for MVC.

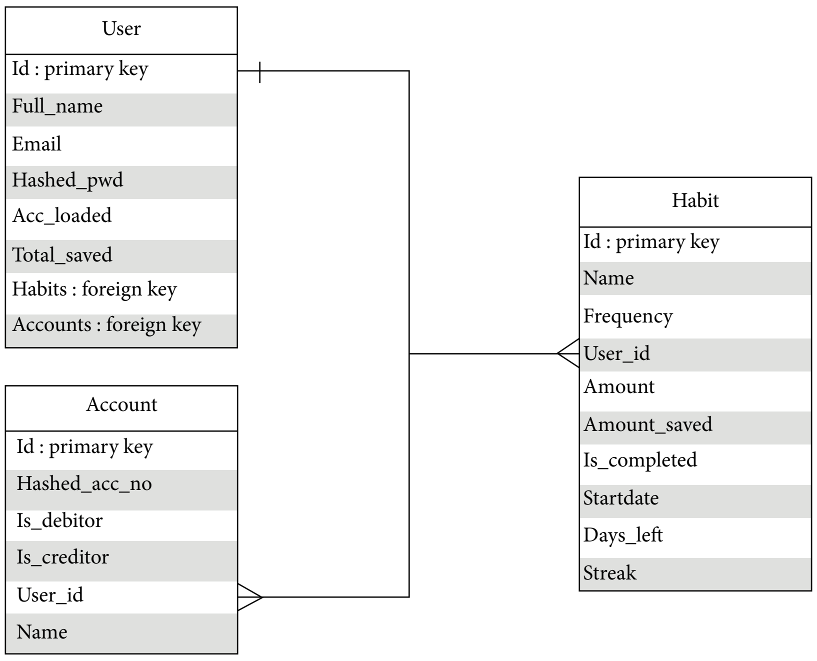


Figure 6: Entity-relationship diagram for database.

4.2. Object-Relational Database. Ensuring that the application data is stored in an organized and secure way requires a database model. Databases can be modelled in different ways, and we need a model that can effectively represent the following information: users, the individual user's habits, and the individual user's accounts. This constitutes an object-relational database [27].

Figure 6 illustrates the entity-relationship diagram for database. It is easy to change the schema for future feature implementations, which will be relevant for further development of the application. Moreover, it models the entity relations in the application domain in a simple way, i.e., as shown in Figure 6, users that each own a number of habits and a number of accounts, each represented as rows in a table. The tables have a fixed number of columns with the variable names related to the object and a variable number of rows with values. Each table also has a column with a primary key, holding unique identifier for all rows stored in

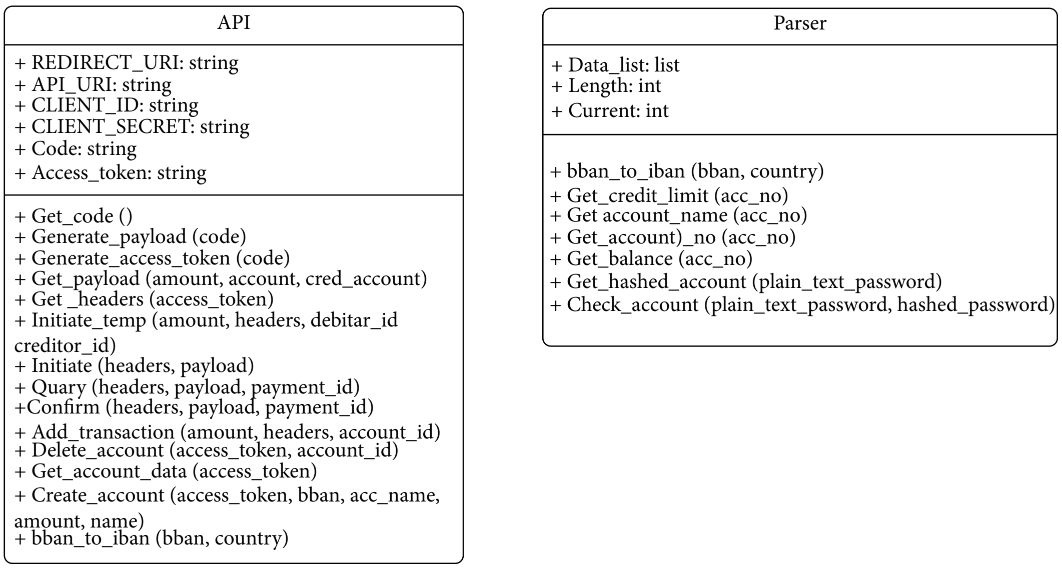

API
+ REDIRECT_URI: string
+ API_URI: string
+ CLIENT_ID: string
+ CLIENT_SECRET: string
+ Code: string
+ Access_token: string
+ Get_code ()
+ Generate_payload (code)
+ Generate_access_token (code)
+ Get_payload (amount, account, cred_account)
+ Get _headers (access_token)
+ Initiate_temp (amount, headers, debitar_id creditor_id)
+ Initiate (headers, payload)
+ Quary (headers, payload, payment_id)
+Confirm (headers, payload, payment_id)
+ Add_transaction (amount, headers, account_id)
+ Delete_account (access_token, account_id)
+ Get_account_data (access_token)
+ Create_account (access_token, bban, acc_name, amount, name)
+ bban_to_iban (bban, country)
Parser
+ Data_list: list
+ Length: int
+ Current: int
+ bban_to_iban (bban, country)
+ Get_credit_limit (acc_no)
+ Get account_name (acc_no)
+ Get_account)_no (acc_no)
+ Get_balance (acc_no)
+ Get_hashed_account (plain_text_password)
+ Check_account (plain_text_password, hashed_password)

Figure 7: Class diagram of API abstraction layer.

that table. The foundation of the relational database model is the foreign keys in the tables that reference the relationship between users and their habits and accounts through lists.

4.2.1. API Abstraction Layer. We present two supplementary APIs and Parser classes to the MVC model. These classes work as abstraction layers for easing communicating with the APIs and filtering out unnecessary data for the application. The purpose is to avoid interacting directly with the API and therefore avoiding unnecessary complexities and errors by encapsulating complex requests in our methods and handling responses accordingly. Figure 7 describes the class diagram of API abstraction layer.

The user's bank and account information can be retrieved directly through the account information services API as a JSON response. The calls to retrieve this response are separated into several methods in the Parser class. The response is first separated into a list object as a field in the Parser and then indexed to extract the needed information. It also consists of different conversion methods to convert different account representations, as well as methods to hash and check account numbers. The API class contains the methods handling the payment initialization service API, hence, triggering transactions between the bank accounts, called whenever habits have been checked off. Through the fields of the API class, we are also able to keep the access token saved across web pages without having to reinstantiate it. Both classes are created as instances for the controller.

4.2.2. Platform Architecture. In order for the application to be deployed in production mode, we propose a platform hosted on the cloud application platform-Heroku [6], with the database connected through Heroku's Postgres add-on.

Figure 8 shows the architecture of the platform where the application is deployed onto. The Flask application itself is as described run through a WSGI server during the development. The application has therefore to be configured to run through HTTP/HTTPS Server instead of running outside of the local host. We propose Gunicorn, a WSGI HTTP server, as recommended by Heroku. The application will send the ORM statements to the database through the database driver psycopg2, which is the most popular for the Python language.

5. Evaluation

In this section, in order to investigate the security and effectiveness of our approach with Open Banking, we introduce our evaluation methodology with the OWASP Top 10 and discuss the identified attacks against application integrating with the Open Banking API.

5.1. Methodology. Figure 9 shows the methodology for applying the OWASP Top 10 to the described application and its architecture, which entails systematically going through the list from the most critical to the least critical threat. The OWASP methodology provides a threat modelling method for categorizing the threats in six different areas, which might result in the weighing of threats to change.

Four of these areas are predetermined in the model and should be the basis of the top 10 ranking in the first place. The categorizations for each element in the list can be viewed from the OWASP documentation. However, by observing the two areas of Threat Agents and Business Impact, they can impact how critical a given threat is. If the Threat Agent and/or Business Impact have a low threat level, then, the threat can quickly become irrelevant.

The OWASP provides a comprehensive model for calculating the risk factor of Threat Agents and Business Impacts [28]. However, the limitations imposed by using the sandbox version indicate that we have a nonexisting user base, lacking business context, and problems that arise as a result of using the sandbox to prevent testing some of the factors. It can therefore be difficult to reach feasible estimates of both Threat Agents and Business Impact. The Threat Agents will therefore simply be assumed high across all areas, since the

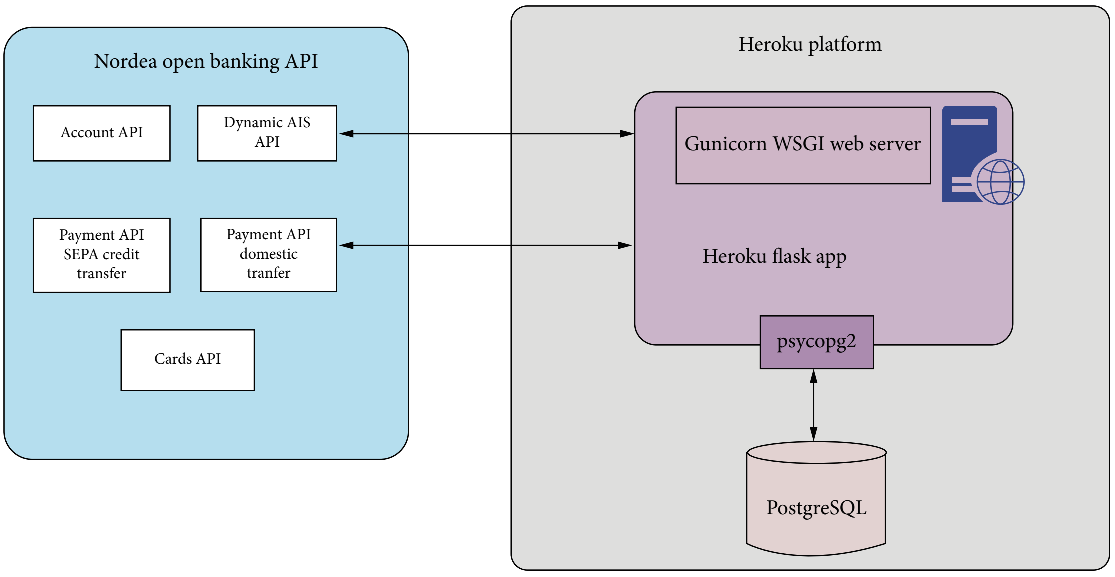
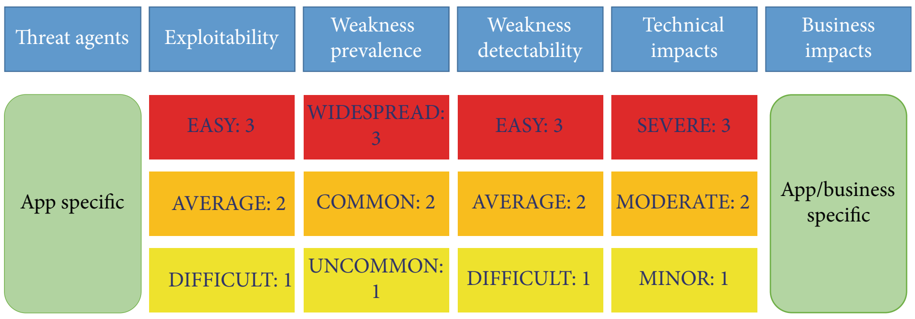


Figure 8: Platform architecture.



Threat agents	Exploitability	Weakness prevalence	Weakness detectability	Technical impacts	Business impacts
App specific	EASY: 3	WIDESPREAD: 3	EASY: 3	SEVERE: 3	App/business specific
	AVERAGE: 2	COMMON: 2	AVERAGE: 2	MODERATE: 2	
	DIFFICULT: 1	UNCOMMON: 1	DIFFICULT: 1	MINOR: 1	

Figure 9: The OWASP Risk Rating Methodology.

financial industry is generally a critical target due to the possibility of financial rewards. The Business Impact estimation needs to include factors such as financial damage, reputation damage, noncompliance, and privacy violation, data that requires an actual business context. We, therefore, conduct a simple estimate of Business Impact, based on the factors that are critical for the end users and their bank accounts:

(1) *Low*. Security is compromised in areas not containing sensitive data, areas that do not trigger unintentional transactions, or attempted attacks that do not affect the application in any way

(2) *Medium*. Security is compromised such that the attacker gains access to sensitive data in the form of bank data or habits stored in the database

(3) *High*. Security is compromised such that the attacker gains access to functionality using the payment initialization service and can trigger unintentional transactions, leading to either small, substantial, or large financial consequences

Each threat area in the top 10 list will be addressed, with an emphasis on the areas that are estimated as highly for the Business Impact.

5.2. Evaluation on Nordea's Open Banking API with the OWASP

5.2.1. Injection and XSS, Threat Agents: 3, Business Impact: 1. We propose a critical approach regarding user input to prevent injection. A number of tests should be made:

(i) Input should be filtered

(ii) Output should be escaped by filtering input

All input fields from the user should be filtered from code-like plain text or injecting raw SQL statements into the database. Submitting unfiltered input into the database can result in a large exposure to SQL injections. This can be detrimental to the privacy of the data; potentially allowing an attacker to access to view the bank information of a user. No further measures need to be proactively taken to prevent injections. ORM SQLAlchemy automatically filters the input of the user, and the Flask Framework automatically escapes output when inserting values into templates, mitigating threats such as JavaScript injection or similar.

5.2.2. Broken Authentication-Threat Agents: 3, Business Impact: 3. We propose a number of actions to mitigate broken authentication, as it is one of the most critical threats against the application and the API:

(i) A set of criteria for the user credentials at sign up

(ii) Preventing that passwords are saved in plain text

(iii) Using multifactor authentication during either signup and/or login

(iv) A user should only be allowed to enter URLs that they are authenticated to enter

The user is required to provide a username and password at signup, and most applications nowadays provide the possibility of signing up through email. That is, the company is able to authenticate and send information through a mail integration. The username should therefore be a valid email, so we are able to perform multifactor authentication by sending a confirmation email to the address. The Flask-Mail extension provides a simple interface to set up SMTP with your Flask application and to send messages directly from the controller. We also require the password to be at least 10 characters long and include both lowercase and uppercase letters, numbers, and a special sign. Most password breaches occurred as a result of weak password criteria, and setting up a number of requirements for the password is therefore an easy and very effective way of preventing broken authentication.

The authentication can also be broken by gaining access to the database and extracting the plain text version of the password. Therefore, only the hashed password will be stored in the database. The bcrypt hash algorithm, combined with salting, is one of the most effective ways to permit brute force attacks. A salt with a length of 12 characters will result in millions of different combinations, making it almost impossible for an attacker to decode. It does have a larger penalty on the time complexity compared to other hash functions. However, there is still a need to make a trade-off.

The Flask-Login extension provides user session management for Flask and allows us to restrict views through a simple decorator to only authenticated users. The Flask Framework therefore provides an easy way of restricting specific URLs.

5.2.3. Sensitive Data Exposure-Threat Agents: 3, Business Impact: 3. We propose only storing the most important data in the database for the application to run. The remaining data will be exposed during run time from the API response, retrieved by the API abstraction layer. The information stored in the database includes a hashed version of the account number and the name of the account. The rest of the information of that specific account can be retrieved at run time by checking the hashed account number against all the user's accounts in the API. The idea is to keep as much information as possible from an attacker that gains access to the database without compromising functionality.

5.2.4. XML External Entities-Threat Agents: 3, Business Impact: 1. The application accepts no uploads or XML, and therefore, an attack of this nature has no Business Impact. It is therefore not relevant to address.

5.2.5. Broken Access Control-Threat Agents: 3, Business Impact: 3. We propose ensuring that the functionality of the application is only exposed to the specific legitimate user, who is able to check off a number of habits and actions, resulting in automatically transferring funds. It is therefore necessary to ensure that it is not possible to gain access to this POST request from other sources. For instance, the current user ID in the POST request to the URL would enable an attacker accessing from the outside, since the request could easily be faked. Thus, we need to ensure that it is in fact the legitimate user who performs the check off, and to check the user owning the habit up against the user that is currently in the session. If an attacker is not allowed to check off a habit, but attempts to do it anyway, they are redirected to an error page. We also need to record this attempt in our logging system, which allows us to have an overview of potential security issues and discover possible threat agents.

In order to further strengthen the application, we have implemented protection against Cross-Site Request Forgery (CSRF) with the Flask package CSRFProtect. This is done by adding a hidden field to all forms. This results in the user having to fill out the form on the website in order to have their request accepted, thus, creating a defence against a myriad of automatic scripts. As an additional security measure, CSRF also requires a secret key to sign the token.

5.2.6. Security Misconfiguration-Threat Agents: 3, Business Impact: 2. Misconfiguration can have a number of different sources that can bring disruption to the application, some of which include:

(i) Revealed stack traces or overly informative error messages

(ii) Improperly configured permissions

(iii) Incorrect values for security settings across servers, frameworks, libraries, or databases

We propose using large parts of the security packages and settings offered by the different parts of the technical stack. Flask provides a number of ways to handle custom error messages to the user in order to prevent showing stack traces or overly informative error messages to users. We propose a combination of the following. Message Flashing, that can be included in the templates, is making it possible to record a custom message at the end of a request and access it in the next request and only the next request. The Python logging package also provides the possibility of printing custom messages and stack traces to the console, limiting the information from showing specific request methods and URLs. However, in 2014, as Flask eliminated error and stack traces from application started running in production mode (https://github.com/pallets/flask/issues/1082), it is no longer necessary to create custom error messages.

To mitigate improperly configured permissions, the selected cloud service provider will not allow open default sharing permissions to the Internet or other users. This ensures that sensitive data stored within cloud storage is not accessed by illegal users. Heroku PaaS is a large service provider and regular audits with the aim to ensure that permission breaches does not occur.

Lastly, the included Flask packages provide a number of security settings. One example is the Flask LoginManager package, from which it is possible to choose from different levels (none, basic, or strong) of security against user session tampering. The latter ensures that Flask-Login keeps track of the client IP address as well as browser agent during browsing. If a change is detected, the user will automatically be logged out.

5.2.7. Components with Known Vulnerabilities-Threat Agents: 3, Business Impact: 3. The components we used have no major known vulnerabilities. The Flask Framework is one of the most popular Python microframeworks and therefore has a number of requirements to ensure adequate security. Moreover, the wide community of developers and contributors can ensure that measures are taken to maintain this security level by frequently updating the most popular and renowned packages. The PostgreSQL database [5] is also addressed at several levels:

(i) *Database File Protection*. All files stored within the database are protected from reading by any account other than the PostgreSQL superuser account

(ii) Connections from a client to the database server are, by default, allowed only via a local Unix socket, not via TCP/IP sockets

(iii) Client connections can be restricted by IP address

(iv) Client connections may be authenticated via other external packages

(v) Each user in PostgreSQL is assigned with a username and a password

(vi) Users may be assigned to groups, and table access may be restricted, for instance, through admin privileges

Furthermore, as mentioned previously, there are many problems with the deployment of the application to Heroku PaaS. Heroku is not known to have any known vulnerabilities itself. However, the server routinely crashes in production mode with no useful error messages when enforcing HTTPS on Heroku. We suspect that this is caused by problems with the TLS Layer, with error messages that stem from Nordea's Open Banking API. Hence, we suspect that the errors stem from how the API handles the TLS Layer in the sandbox version. This imposes a high risk for the packages sent between the application and the API to be intersected. However, no sufficient documentation explains how to mitigate this issue in Nordea's documentation. This will be an interesting topic for future enhancement.

5.2.8. Insufficient Logging and Monitoring-Threat Agents: 3, Business Impact: 2. As mentioned previously, whenever a user attempts to check off the habit, or perform any other actions in the application, of another user, it is added to the log. The log is handled through a logging package offered by the Python library. We propose also including logging for IP addresses and alarms whenever a user is logged in from a different country.

6. Discussion

Applying the OWASP Top 10 Threats and Risk Modelling Framework to our web application and Nordea's Open Banking API shows that it can mitigate a large part of the most critical threats to the application. The threats posed by Broken Authentication, the most critical in terms of Business Impact, is now largely protected from breaches that could cause the user to lose account funds. The same applies for Sensitive Data Exposure and Broken Access Control that were also categorized as very critical threats.

However, the OWASP framework also exploited that the components with known vulnerabilities posed a high threat to the application, especially Nordea's APIs. The problems with the TLS Layer in Nordea's Open Banking API force us to use HTTP in production mode to avoid the routinely crashes occurred with HTTPS. This means that the packages sent from the API to the application are encrypted. Packages that can contain access tokens, client IDs, or secret keys might give access to Nordea's infrastructure. This vulnerability is impossible to handle without more documentation of the API, since it does not stem from the application itself. Below are some main challenges on open banking security.

(i) How to securely share data when transforming the relationship between customers and banks

(ii) How to transparently manage mutual authentication

(iii) How to secure data privacy when enabling users to control and share personal data by customizing the Access Control List (ACL) [1]

7. Conclusion and Future Work

Currently, Open Banking has received much attention, which refers to the process of using APIs to open up consumers' financial data to third parties. This concept is believed to be secure by enforcing that only the customer and data owner can authorise any connection between the bank and a regulated third party. However, such openness may also incur some kind of security issue. In this work, we proposed a technical stack and an architectural model that can easily integrate and secure Nordea's Open Banking API. In the evaluation, we applied the OWASP Top 10 threats and threat modelling methodology to identify the most prevalent threats regarding the application data and the functionality of the APIs.

The results showed that many of these security measures were either handled automatically by the components offered by the technical stack or were easily preventable through included packages of the Flask Framework. However, it also shows that the application faces a high risk due to the compromised handling of the TLS Layer in the API, causing the production server to routinely crash when using HTTPS. These risks may propagate upwards in the architecture, resulting in high risks for the user's account data and funds. Since the server loggings show that the errors stem from the API itself, it is most likely not due to the choices of any of the cloud application platform, packages, libraries, database, or frameworks. It is also found that adding an API abstraction layer can facilitate the communication when developing the API, and that it can be implemented as a modification to the MVC for web applications.

For future work, we plan to keep gaining more data and information on the TLS Layer handling by cooperating with the support team from Nordea Bank in Denmark, especially the sandbox documentation. With more Open Banking Team code samples, we believe it can help make more practical contributions to the documentation. In addition, it is another interesting and important direction to apply other threat models to examine the threats and risk compared with the current results with OWASP.

Data Availability

No data were used to support this study.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This is a full version of our paper published in Proc. 13th International Conference on Network and System Security (NSS), pp. 185-198, 2019 [4]. To differentiate our current version from the previous one, we have improved the article in the following aspects. (1) In the conference version, there are mainly two diagrams, while in this version, we have 9 diagrams to describe our approach in more detail. (2) In addition, in this full version, we provide more information on Nordea's Open Banking API, such as sequence diagram, access authorization flow, and account information. (3) In this full version, we also provide more information and details on the OWASP Risk Rating Methodology, which can facilitate readers to understand how this methodology works. The authors would like to thank the help from Nordea Bank in Denmark, and Weizhi Meng was partially supported by H2020 CyberSec4Europe under project no. 830929.

References

[1] Z. Xu, Q. Wang, Z. Wang, D. Liu, Y. Xiang, and S. Wen, "PPM: a provenance-provided data sharing model for open banking via blockchain," in *Proceeding of ACSW*, pp. 1–8, Melbourne, Australia, 2020.

[2] S. Kiljan, K. Simoens, D. D. Cock, M. C. J. D. van Eekelen, and H. P. E. Vranken, "A survey of authentication and communications security in online banking," *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, pp. 1–35, 2017.

[3] N. Provos and D. Mazieres, "A future-adaptable password scheme," *Proceedings of USENIX Annual Technical Conference*, , pp. 81–91, FREENIX Track, 1999.

[4] D. Kellezi, C. Boegelund, and W. Meng, "Towards secure open banking architecture: an evaluation with OWASP," in *Proceedings of the 13th International Conference on Network and System Security (NSS)*, pp. 185–198, Sapporo, Japan, 2019.

[5] PostgreSQL, *The world's most advanced open source database*https://www.postgresql.org/.

[6] *Heroku: Cloud Application Platform*https://www.heroku.com/.

[7] Pallets Team, *Flask's Documentation*http://flask.pocoo.org/docs/1.0/.

[8] The SQLAlchemy authors and contributors2019 https://docs.sqlalchemy.org/en/13/.

[9] Pallets Team2010 https://flask-sqlalchemy.palletsprojects.com/en/2.x/.

[10] OWASP, *Top Ten Web Application Security Risks*https://owasp.org/www-project-top-ten/.

[11] P. Rogaway and T. Shrimpton, "Cryptographic hash-function basics: definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance," *Proceedings of FSE*, , pp. 371–388, Springer, 2004.

[12] W. Meng and Z. Liu, "TMGMap: designing touch movement-based geographical password authentication on smartphones," *Proceedings of the 14th International Conference on Information Security Practice and Experience (ISPEC)*, , pp. 373–390, Springer, Cham, 2018.

[13] W. Meng, L. Zhu, W. Li, J. Han, and Y. Li, "Enhancing the security of FinTech applications with map-based graphical password authentication," *Future Generation Computer Systems*, vol. 101, pp. 1018–1027, 2019.

[14] M. Burnett, *10,000 Top Passwords*https://xato.net/10-000-top-passwords-6d6380716fe0.

[15] Nordea Open Banking Team, 2019, https://developer.nordeaopenbanking.com/app/documentation?api=Accounts%20API.

[16] A. Sapan, B. Oztekin, E. Unsal, and A. Sen, "Testing OpenAPI banking payment system with model based test approach," in *2020 Turkish National Software Engineering Symposium (UYMS)*, Istanbul, Turkey, 2020.

[17] S. Rafique, M. Humayun, B. Hamid, A. Abbas, M. Akhtar, and K. Iqbal, "Web application security vulnerabilities detection approaches: a systematic mapping study," in *2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pp. 469–474, Takamatsu, Japan, 2015.

[18] Post Learning Centerhttps://learning.getpostman.com/docs/postman/api_documentation/intro_to_api_documentation/s.

[19] C. Dong, Z. Wang, S. Chen, and Y. Xiang, "BBM: a blockchain-based model for open banking via self-sovereign identity," in *International Conference on Blockchain*, pp. 61–75, Springer, Cham, 2020.

[20] H. Wang, S. Ma, H. N. Dai, M. Imran, and T. Wang, "Blockchain-based data privacy management with nudge theory in open banking," *Future Generation Computer Systems*, vol. 110, pp. 812–823, 2020.

[21] A. Stranieri, A. N. McInnes, M. Hashmi, and T. Sahama, "Open banking and electronic health records," in *2021 Australasian Computer Science Week Multiconference*, Dunedin, New Zealand, 2021.

[22] W. Meng, W. Li, and L. Kwok, "EFM: enhancing the performance of signature-based network intrusion detection systems using enhanced filter mechanism," *Computers & Security*, vol. 43, pp. 189–204, 2014.

[23] Z. Ma, L. Liu, and W. Meng, "Towards multiple-mix-attack detection via consensus-based trust management in IoT networks," *Computers & Security*, vol. 96, article 101898, 2020.

[24] S. M. Sohi, J. P. Seifert, and F. Ganji, "RNNIDS: enhancing network intrusion detection systems through deep learning," *Computers & Security*, vol. 102, p. 102151, 2021.

[25] D. P. Pop and A. Altar, "Designing an MVC model for rapid web application development," *Procedia Engineering*, vol. 69, pp. 1172–1179, 2014.

[26] M. Grinberg, *Flask Web Development: Developing Web Applications with Python*, OReilly, California, USA, 2014.

[27] IBM Informix, 2011, https://www.ibm.com/support/knowledgecenter/hu/SSGU8G_11.50.0/com.ibm.gsg.doc/ids_gsg_416.htm.

[28] The OWASP Foundation, 2017, https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.