# Towards Secure Open Banking Architecture: An Evaluation with OWASP

Deina Kellezi[1], Christian Boegelund[1], and Weizhi Meng[1,2(✉)]

[1] Department of Applied Mathematics and Computer Science, Technical University of Denmark, Lyngby, Denmark
weme@dtu.dk

[2] Department of Computer Science, Guangzhou University, Guangzhou, China

**Abstract.** The European Union passed the PSD2 regulation in 2015, which gives ownership of bank accounts to the private person owning it. As a result, the term *Open Banking*, allowing third party providers and developers access to bank APIs, has emerged, welcoming a myriad of innovative solutions for the financial sector. However, multiple cyber security issues arise from exposing bank data to third party providers through an API. In this work, we propose an architectural model that ensures clear separation of concern and easy integration with Nordea's Open Banking APIs (sandbox version), and a technological stack, consisting of the micro-framework Flask, the cloud application platform Heroku and persistent data storage layer (using Postgres). We analyze the web application's security threats, and determine whether or not the technological frame provides adequate security protection, by leveraging the OWASP Top 10 list of the Ten Most Critical Web Application Security Risks. Our results can support future developers and industries working on web applications for Open Banking towards security improvement by choosing the right frameworks and considering the most important vulnerabilities, as well as contributing to the documentation and development of Nordea's APIs.

**Keywords:** Web security · Open Banking API · OWASP · Threat and risk · PSD2 regulation · Secure architecture

## 1 Introduction

The financial sector is transforming radically. Technological innovations arise due to new regulations, demanding banks to develop APIs that enables the following two features: (1) Access to bank account information; and (2) Triggering of transactions between different accounts.

*Nordea*, one of the largest banks in the Nordics, released the first version of their Open Banking API in January 2019. As one of few banks, they also released a sandbox version that allows possible third party providers to use the API in


J. K. Liu and X. Huang (Eds.): NSS 2019, LNCS 11928, pp. 185–198, 2019.
https://doi.org/10.1007/978-3-030-36938-5_11

a test environment. Online banking applications are one of the most lucrative targets for attacks. Many have been mitigated through Nordea's own protocols, such as the production APIs requiring a multiple step authentication through nemID 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 breaches.

Up to 3300 developers are currently registered as developers on Nordea Open Banking, and 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 [7]. Moreover, 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.

Due to the complicated process of obtaining a financial license to use actual production data, in this work, we delimit the problem by using only the sandbox version to develop the solution of triggering transactions based on habits, and to model the possible threats. In particular, we first identify the background of the technology stack needed to support development of a deployed web application, with a persistent data storage layer and a high level of security. We also define the system architecture as well as how it will communicate with the API. We then use the OWASP Top 10 list of the Ten Most Critical Web Application Security Risks methodology to study the possible threats and its risk levels. Our contributions in this paper can be summarized as follows.

- We propose a system architecture of the web application in practice, in collaborating with Nordea Open Banking.
- We identify relevant potential attacks on our system, and analyze the risk by considering the OWASP Top 10 list. We also provide insights on how to mitigate them accordingly.

The remainder of this article is structured as follows: Sect. 2 clarifies the important background information of the Flask framework and Nordea's Open Banking API. In Sect. 3, we present the proposed web application architecture of secure Open Banking. Section 4 evaluates our proposed architecture by leveraging the OWASP Top 10 list. Section 5 introduces the related work, and Sect. 6 summarizes our work with future directions.

## 2 Background

The micro-framework for web development, Flask (for Python), will be used to develop the application. Flask mitigates many security threats by default, supplemented by a number of renowned third-party extensions and packages from authenticated by the Flask community, and is customize-able to a great extend.

It is also provides out of the box abstraction layers for communicating with the popular object relational database Postgres and the cloud application platform Heroku 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, in this case web browsers, to this application instance. The logic is handled using the Web Server Gateway Interface (WSGI) as protocol, constantly awaiting requests. The framework is compliment with the WSGI server standard [8].

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

### 2.2 Cloud Application Platform

Heroku is one of the first and largest Platform as a Service (PaaS) provider 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, dynos, to measure usage of the service and perform different tasks on the server. It also provides a large number of plugins and addons for databases, email support, and many other services. Heroku supports Postgres [6] databases as an add-on, created and configured through the command line client.

### 2.3 Database Management

Flask puts no restriction on what database packages can be used and supports a number of different database abstraction layer packages. The web application will run on the Postgres database engine supported by the ORM, SQLAlchemy. This is based on the following evaluation on a number of different criteria:

- **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, compared to writing raw SQL statements [16].
- **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.

- **Portability:** The application platform of choice, Heroku, supports a number of different database engine choices, the most popular and extensible being Postgres and MySQL [2].
- **Integration:** Flask includes several package designed to handle ORMs, such as Flask-SQLAlchemy [17], which includes engine-specific commands to handle connection.

### 2.4 The Nordea Open Banking APIs

The Nordea APIs provide 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, e.g. changing the balance on the accounts [3]. In the following, we will go through a list of the relevant endpoints, the most crucial being:

- **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 which 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.
- **Account Information Services.** The API of Account Information includes the possibility to check the contents of the different sample accounts in the sandbox version. We can create new accounts, delete accounts and add funds to said accounts. This can be done by sending a request to the `Accounts` endpoint [3].
- **Payment Initialization Services.** The Payment Initialization API provides functionality to create payments directly in the API, moving funds from one account to another [3].

## 3 Our Proposed Web Application Architecture

### 3.1 The Architecture

In order to define the architecture, we present a model based on the Model-View-Controller architecture (MVC) specifically adjusted for web development as proposed by Dragos-Paul Pop and Adam Altar [11]. They found that developers often combine HTML code with server side programming languages during web development to create dynamic web pages and applications, and this leads to highly entangled and unmaintainable code. With an MVC pattern, it is possible to prevent cluttering by separating the three overall parts of a web application. The model also proposes how to handle the API integration through an abstraction layer and include it in the MVC.

- **Model:** A persistent data storage layer through a data centre or database.
- **Controller:** The HTTP requests triggered by user actions and general routing of different sub-pages.
- **View:** The HTML code and mark-up languages in the templates rendered to the user as a result of a request.

These three main components will be built through a modular approach, using blueprints as recommended by Flask.

Figure 1 presents the proposed diagram for the adjusted MVC, 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.

**The Model.** Presented as blue in the figure and shows the modelling of the data objects and relationships. This is the direct representation of the schema in the database. Whenever the SQLAlchemy methods, either querying, updating or deleting data, are called on the defined data objects in the model, the database is updated accordingly. This also provides simpler commands for establishing connections to Postgres through the URL of the database as handled by the controller.

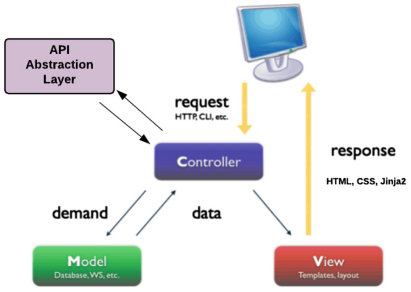


**Fig. 1.** MVC architecture for web application. (Color figure online)

**The Controller.** Presented as green in the figure and shows the controller separated into three blueprints:

- `auth_controller`: Rendering the pages responsible for signing up and authenticating users logging in.
- `main_controller`: Rendering the pages of the specific user session, containing URLs for creating habits, checking off habits that are completed, overview over habits, overview over accounts and settings. This is restricted to authenticated users only.
- `admin_controller`: Rendering the pages of the administration page included for demonstration purposes that allows to test the different API functionality. This is restricted to users with admin rights only.

The blueprints provide a clearer separation of the different states in the application. This separation could done through application dispatching, ie. 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 [1]. 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.

**The View.** Presented as orange in the figure and 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 Flask, enabling all templates to inherit from a base design, as well as register onto their specific controller through the aforementioned blueprints. This also allows dynamic rendering of values provided as argument to the templates when rendered [1].

### 3.2 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 [18].

**API Abstraction Layer.** We present two supplementary API 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 methods and handling responses accordingly.

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 is 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 were also able to keep the access token saved across web pages without having to re-instantiate it. Both classes are created as instances for the controller.

**Platform Architecture.** In order for the application to be deployed in production mode, we propose a platform hosted on the cloud application platform, Heroku [2], with the database connected through Heroku's Postgres add-on.

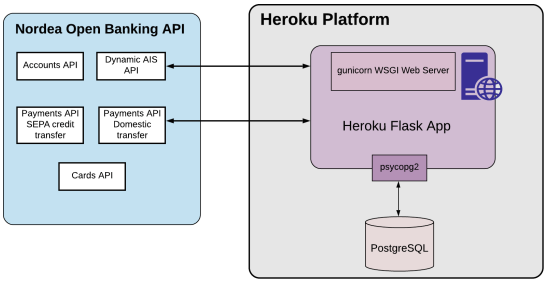


**Fig. 2.** Platform architecture.

Figure 2 shows the architecture of the platform the application is deployed onto. The Flask application itself is as described run through a WSGI server during development. The application will therefore need to be configured to run through HTTP/HTTPS Server instead to run outside of local host. We propose Gunicorn, a WSGI HTTP server, as recommended by Heroku [5]. The application will send the ORM statements to the database through the database driver psycopg2, the most popular for the Python language.

## 4 Evaluation of Attacks Against Application Integrating with Open Banking API

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

Four of these areas are pre-determined in the model and have been 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 [13,14]. However, observing the two areas, Threat Agents and Business Impact, these areas impact how critical a given threat is. If the Threat Agent and/or Business Impact has a low threat level, then the threat can quickly become irrelevant. OWASP provides a comprehensive model for calculating the risk factor of Threat Agents and Business Impacts [15].

However, the limitations imposed by using the sandbox version means that we have a non-existing user base, lacking business context and problems that arise as a result of using the sandbox that 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 financial industry is generally a critical target due to the possibility of financial rewards. The Business Impact estimation needs to include factors like financial damage, reputation damage, non-compliance and privacy violation, data that requires an actual business context. We therefore conduct simple estimates of Business Impact, based on factors that are critical for the end user and their bank accounts:

1. **Low:** Security is compromised in areas not containing sensitive data, areas that do not trigger unintentional transactions, or attempt attacks that does 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.

### 4.1 Applying OWASP to the Application Ensuring Secure Integration with the API

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

- Input should be filtered
- 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 access to view the bank information of the 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.

**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:

- A set of criteria for the user credentials at sign up
- Preventing that passwords are saved in plain text
- Using multi-factor authentication during either sign-up and/or login
- A user should only be allowed to enter URLs that they are authenticated to enter

The user is required to provide a user name and password at sign-up. Most application nowadays provide the possibility of signing up through email. This is so the company is able to authenticate and send information through a mail integration. The user name should therefore be a valid email, so we are able to perform multi-factor 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 to be at least 10 characters long, include both lowercase and uppercase letters, numbers as well as a special sign. Most password breaches happen as a result of weak password criteria, and setting up a number of requirements for the password is therefore an easy and very effective ways 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 version of the password is stored in the database. The bcrypt hashing 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 hashing algorithms. However, we are willing to make this 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.

**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.

**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.

**Broken Access Control - Threat Agents: 3, Business Impact: 3.** We propose ensuring that the functionality of the application is only exposed to the specific user logged in. The user is able to check off a number of habits and actions, resulting in automatically transfer funds. It is therefore necessary to ensure that it is not possible to gain access to this POST request from other sources. For instance, including the current user ID in the POST request to the URL, would enable an attacker access from the outside, since the request could easily be faked. Thus, the only to ensure that the it is in fact the logged in user performing the check off, is 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 log the attempt in our logging system. This 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.

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

- Revealed stack traces or overly informative error messages
- Improperly configured permissions
- 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 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, making it possible to record a custom message at the end of a request and access it in the next request and only 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, Flask eliminated error and stack traces from application running in production mode[1], so it is no longer necessary to create custom error messages.

For mitigating improperly configured permissions, the cloud service provider of choice does not allow open default sharing permissions to the Internet or other users. This ensures that sensitive data stored within cloud storage is not accessed wrongfully. Heroku PaaS is a large service provider and regular audits are performed to ensure that permission breaches does not occur. Lastly, the included

[1] https://github.com/pallets/flask/issues/1082.

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.

**Components with Known Vulnerabilities - Threat Agents: 3, Business Impact: 3.** The components we use have no major known vulnerabilities. The Flask framework is one of the most popular Python micro-frameworks and therefore has a number of requirements to ensure adequate security. Moreover, the wide community of developers and contributors ensure that measures are taken to maintain this security level by frequently updating the most popular and renowned packages. The Postgres database [6] is also addressed at several levels:

- Database file protection. All files stored within the database are protected from reading by any account other than the Postgres superuser account.
- Connections from a client to the database server are, by default, allowed only via a local Unix socket, not via TCP/IP sockets.
- Client connections can be restricted by IP address.
- Client connections may be authenticated vi other external packages.
- Each user in Postgres is assigned a username and password.
- Users may be assigned to groups, and table access may be restricted, for instance through admin privileges.

Furthermore, as mentioned previously, there are currently 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.

**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 alarm whenever a user is logged in from a different country.

**Discussion.** Applying the OWASP Top 10 Threats and Risk Modelling Framework to the application 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 result in the user losing 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. Specifically 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 occurring 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 that 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.

## 5 Related Work

The previous work carried out on web applications integrating with Open Banking APIs is limited, and practically non-existing using the technical stack described in this paper. This is due to two reasons such as:

- The novelty of most of the interfaces, including Nordea's APIs
- The requirements of developers need to be approved by national financial authorities for using the APIs in production

These factors has 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 a 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. One example is a study on 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 [12].

The sandbox version of the Nordea Open Banking API was officially released during 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 generic server errors. The limited sample codes and lacking documentation on possible error codes made it difficult to correct. In order to find a solution, we conducted a simulation with the API simulation tool named Postman [4]. The connection was successful, 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 were negligible. We contacted the senior software architect of Nordea Open Banking. The support team 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`[2] 2.0-process was set to an incorrect value. We decided to contribute to the community of developers using the Nordea Open Banking API by creating a pull request[3]. At the moment, the samle code only works with version 2 of the API, while the API has been updated to version 3 since then. to contribute to the wider community of developers using the Nordea Open Banking API.

## 6 Conclusion and Future Work

In this work, we systematically proposed and described a technical stack and architectural model to ensure a web application that could integrate easily with Nordea's Open Banking API in a secure manner. This was used as input to the OWASP Top 10 Threats and threat modelling methodology to identify the most prevalent threats to the application data and, indirectly, the functionality of the APIs. The OWASP recommendations were used to prevent these attacks by taking adequate security methods to the most critical areas. 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 compromising 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 records 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. The results also show that creating an API Abstraction Layer eases communication with the API during development, and that it can be implemented as a modification to the MVC for web applications.

For future work, we intend to contact the support team of the API to gain more information on the handling of the TLS Layer, information that is currently lacking in the sandbox documentation. This may lead to one more contribution to the documentation, other than the pull-request made to the Open Banking Team code samples [9,10]. We could also consider applying other popular threat models to the application in attempt to find other vulnerabilities not detected by the use of OWASP.

**Acknowledgments.** Weizhi Meng was partially supported by H2020-SU-ICT-03-2018: CyberSec4Europe with No. 830929, and National Natural Science Foundation of China (No. 61802077).

[2] OAuth is one of the leading protocols within authentication.

[3] The PR can be seen here: https://github.com/NordeaOB/examples/pull/7.

## References


1. Grinberg, M.: Flask Web Development: Developing Web Applications with Python. OReilly, California (2014)
2. Heroku Dev Center (2018). https://devcenter.heroku.com/categories/reference
3. Nordea Open Banking Team (2019). https://developer.nordeaopenbanking.com/app/documentation?api=Accounts%20API
4. Post Learning Center. https://learning.getpostman.com/docs/postman/api_documentation/intro_to_api_documentation/s
5. Heroku Dev Center (2019). https://devcenter.heroku.com/articles/heroku-postgresql. Documentation
6. The PostgreSQL Global Development Group (1996–2019). https://www.postgresql.org/docs/12/index.html
7. Kiljan, S., Simoens, K., Cock, D.D., van Eekelen, M.C.J.D., Vranken, H.P.E.: A survey of authentication and communications security in online banking. ACM Comput. Surv. **49**(4), 61:1–61:35 (2017)
8. Pallets Team: Flask's Documentation. http://flask.pocoo.org/docs/1.0/
9. Rogaway, P., Shrimpton, T.: Cryptographic hash-function basics: definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In: Roy, B., Meier, W. (eds.) FSE 2004. LNCS, vol. 3017, pp. 371–388. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25937-4_24
10. Niels, P., David, M.: A future adaptable password scheme. The OpenBSD Project (1999)
11. Dragos-Paul, P., Adam, A.: Designing an MVC model for rapid web application development. Procedia Eng. **69**, 1172–1179 (2014)
12. Sajjad, R., Mamoona, H., Bushra, H., Ansar A., Muhammad, A., Kamil, I.: Web application security vulnerabilities detection approaches: a systematic mapping study. IEEE (2015)
13. The OWASP Foundation: OWASP top 10 - the ten most critical web application security risks. Release notes (2013)
14. The OWASP Foundation: Top 10 List (2017). https://www.owasp.org/index.php/Category:OWASP_Top_Ten_2017_Project. Documentation
15. The OWASP Foundation: Risk Rating Methodology (2017). https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology. Documentation
16. The SQLAlchemy authors and contributors (2019). https://docs.sqlalchemy.org/en/13/. Documentation
17. Pallets Team (2010). https://flask-sqlalchemy.palletsprojects.com/en/2.x/. Flask-SQLAlchemy Documentation
18. IBM Informix (2011). https://www.ibm.com/support/knowledgecenter/hu/SSGU8G_11.50.0/com.ibm.gsg.doc/ids_gsg_416.htm