

Information Security, Milestone 1

Lucia Eve Berger (luberger@unibz.it)
Giorgi Gabunia (ggabunia@unibz.it)
European Masters in Software Engineering

May 1, 2020

Part I

Design of the E-Commerce Website

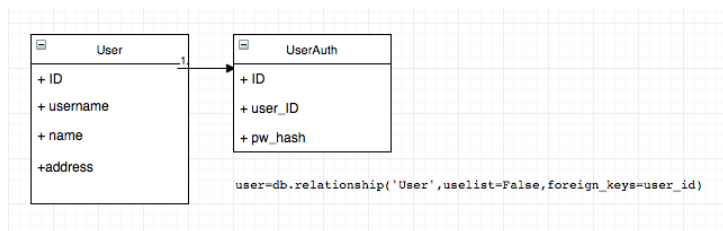
Solution. Our application relies on the design of a secure API. Our E-commerce website allows users to *securely* purchase government secrets. The below spec refers to the endpoints in the web application. The application will be written in python flask, a microframework. <http://flask.pocoo.org/> with javascript and html templating on the front-end. The database will be postgres.

1 User data management (name, address, login info, etc.)

1.1 Registration: */register*

From the client-side, form validation will be employed. The *username* field will be the email address of the user. Form validation will again be used to ensure the appropriate length of the password (rules on characters) The password should be inputted, validation to ensure they are the same. Once data is passed by front-end validator, it is posted to the server side.

From the server side, the hash of the password, (SHA256 Hash generator) is created and saved. Using the Flask convention, we will have a UserTable and an UserAuth Table. As shown below, they will be related on the userID foreign key, following the convention.



1.2 Login: */login*

After user enters credentials, they are validated and a session is created. Following login, the user is directed to the main page, all secrets. Upon login validated, the user can modify their profile. As of now, the details of the user profile are name and address but may expand as we develop the application.

CREATE A NEW ACCOUNT

Create an account to [sell](#) and [buy](#) government secrets.
Have an account? [login here](#)

First Name	Last Name
Email (Username)	
Password	
Confirm Password	

REGISTER

LOGIN

Email (Username)
Password

REGISTER

For session management, we will use the JSON Web Token (JWT) framework that creates a token for the user. The flask JWT library uses the public/private key pair using RSA. The tokens are created and compared.

1.3 Logout: */logout*

Clears the JWT token and closes the user session.

1.4 Profile edit: */editprofile*

The user can modify the details of their profile.

EDIT PROFILE

Edit your profile details

Name: FIRSTNAME

Last Name: LASTNAME

Email: EMAIL@email.com

[change password](#)

Address: XYZ Road, Street, City

Edit your preferred payment method

PAYMENT METHOD: ☒ Visa

1.5 View Purchased Secrets */mysecrets*

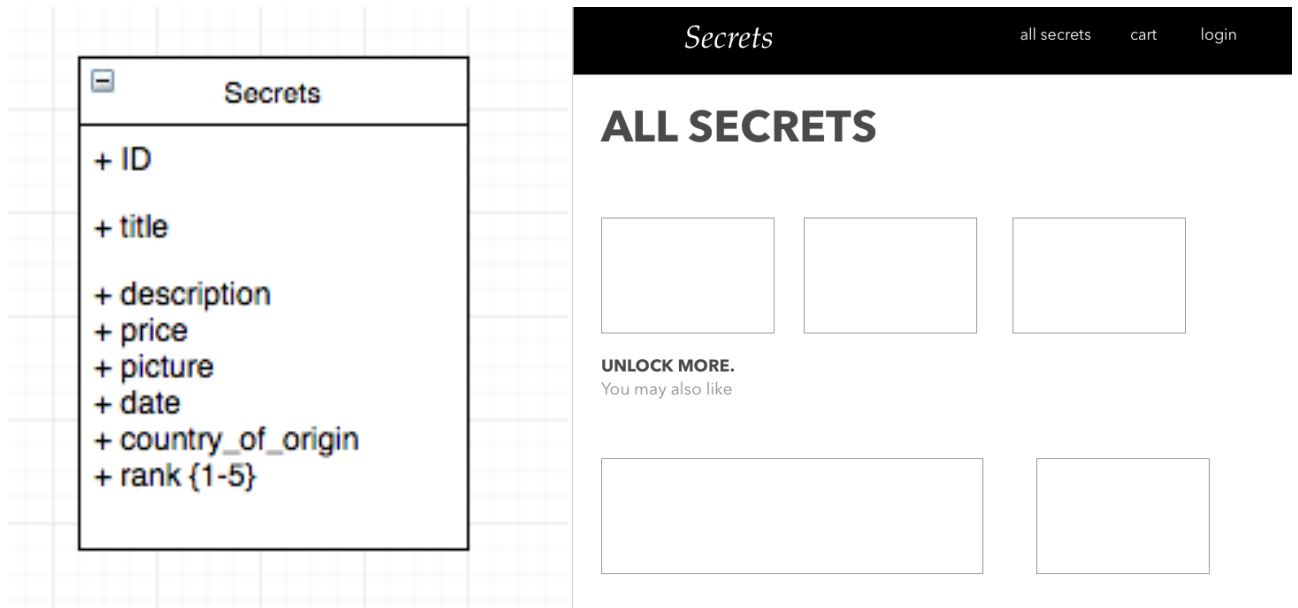
The mysecrets page shows recent purchases and recommends secrets to purchase in the future.

2 All Secrets

2.1 View All Secrets/*secrets*

Secrets is an open api and is available to all users. Users can query to the database to show all the available secrets. To get more details on a specific secrets, GET request with the ID of the secret can be made.

The secrets object will include



3 Shopping Cart

Login is required for the shopping cart. The shopping cart includes the below endpoints.

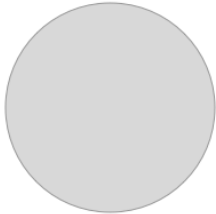
- *add /cart/ID*
- *delete/cart/ID*
- *charge /cart/ID*

Secrets

all secrets cart logout

CHECKOUT

Title: Your secret
Total: \$26
Description: xyea afasfs
afdafd bfdaskfsa fdsal.



Edit your preferred payment method

PAYMENT METHOD: ☒ Visa

The web-app will use the payment API of VISA, Paypal and Sofort for processing. To processes a payment the user must be logged in with an open session. The processing will only be open for 5 minutes to make a payment. The item can stay in the cart without being processed. The user has to be logged in to make a cart.

Part II

Address vulnerabilities:

4 Code injection (SQL or other)

There are only 3 cases where the User creates a query that accesses database to write something: when he/she registers, when he/she edits account profile and during the authorization. In each of those cases, several layer of security will be implemented to avoid SQL injection:

- Client-Side form validation in html (input types and regex validators)
- Flasks Server-Side form validation
- Using parameterized queries, which ensure that input data is insulated and encapsulated as variables. Flask can use SQLAlchemy for the parameterization and ensures all the dangerous characters are escaped. E.g., it automatically quotes all special characters (Semicolons, apostrophes etc.).
- All the other cases the database is accessed is when changing page of displayed products (i.e., we show 30 products on first page, user clicks next or 2 to go to second page, where next 30 products are displayed) or calling the detailed information on a product. In all those cases, only argument for the query is an int: either the ID of a product or the number of the page. In this case, if the user tries to inject sql instead of an ID number, symple type-check will be enough to find the injection. If the user alters the url `http://site.com/product?id=2` and puts a string value instead of 2, e.g. `?id=2273?DROP0TABLE?20Products?3B`, we will check if id parameter is int, and since it is not, we will not execute it.

5 Cross-site scripting (javascript or other)

Flask uses Jinja2 Syntax for generating HTML pages. By default, it is configured in such a way that it automatically escapes all the strings loaded through it. Thus, we will always load everything (user input if it is necessary anywhere, as well as the data from our database) using jinja2, thus avoiding Cross-Site Scripting.

```
<a href="{{ productLink }}">Product Title</a>
```

6 Misconfiguration issues

To address misconfiguration

- we will use environmental variables for configuration. For example, we will be using

```
os.get(VARIABLE, '')
```

for the database string. The environmental variable approach will also help us maintain our application in testing, QA and production, with each having specific configurations derived from the system environment.

- Development and production configuration to be deployed in different configurations, including the removal of all test accounts.
- Users will be required to change their passwords after 3 months.
- There will be no backdoor (default admin) account.

7 Exposure of sensitive data

The only sensitive data we have will be passwords of the users. To avoid its exposure, the password will not be stored as a clear text anywhere. Instead, during the registration process, after the user posts his registration details on the website, the password will be hashed using SHA256 encryption algorithm.

- We will store the hash as the user password and, when the user tries to authorize, the password he/she inputs will be hashed and compared to the one in database. If the hashes match, authorization will be successful, otherwise- not. Thus, no instances of passwords as cleartext will be stored.

```
@staticmethod
def generate_hash(password):
    return sha256.hash(password)
@staticmethod
def verify_hash(password, hash):
    return sha256.verify(password, hash)
```

- Another sensitive data would be the bought items list for a particular user. To avoid this, the request for that data will check if the user is authorized. If not, request will be denied. If yes, it will automatically only retrieve the products bought by that particular, currently authorized user. No user input will be checked for this, so they can not try to access products of other users.

8 Session management

To manage sessions, we will use the *JSON Web (JWT) token* approach, *flask Sessions* and the *SqlAlchemy-SessionInterface*.

- For the JWT token, we will manage the authorization with a secure token approach. The JWT secret key constant is handled with the base64url encoding of the header and payload. After, it is processed with the HS256 algorithm to create the secret. This key is only sent over the HTTPS connection, between the trusted client and trusted server.
- Information that is stored in the Flask session cookie is strictly divided by type, i.e. user, cart, etc. On the client-side web sessions, after the JWT token has been validated, details on the cart entry will be addressed in the encrypted session payload.
- By using the Flask session library, the server encrypts the session data and also sets a timer on it. For example, with the *PERMANENTSESSIONLIFETIME*, the time-line can be set. We intend to use a timed 5 minutes while in the cart-process. We can use the Sessions to track where in the cart process the user is.

9 Insecure Direct Object References

To avoid this problem, jinja2 syntax is once again very useful. All the data that will be displayed will be displayed through jinja2, thus no information about directories, files, database etc. will be displayed for any user.

10 Brute Force Authorization Attempts

To avoid users simply brute forcing their way into some account, we will give 3 attempts to authorize. If all 3 fail, some kind of captcha (probably google recaptcha) will appear, making automated attempt to bruteforce impossible.

11 Cross-site request forgery

To limit the threat of CSRF or a trusted user exploiting a GET or POST request, we propose adding a random string to the session. Since we are using a JSON based API, the random string CSRF token will be defined on a separate GET request.

An additional measure we will add is the *HttpOnlycookies*. As an add-on to the above session management, we propose allowing cookies only on HTTP requests.

```
app.jinja_env.globals['csrf_token'] = generate_csrf_token
```

Part III

Sources:

<http://flask.pocoo.org/snippets/3/>

<https://blog.miguelgrinberg.com/post/how-secure-is-the-flask-user-session>

<https://medium.com/vandium-software/5-easy-steps-to-understanding-json-web-tokens-jwt-1164c0adfcec>

□