

DAT250

PROSJEKT - HØSTEN 2021

Prosjekt-
oppgaven

Project 1, Website security

Gruppenavn

Gruppe 15, AlphaBank

Gruppens
medlemmer

Navn

Studentnummer

Matias Ramsland

259150

Lukasz Pietkiewicz

253469

Chiran Pokhrel

259205

Jakub Mroz

260703

Konrad Jarczyk

242615

?contentsname?

Innhold	i
Introduction	ii
1 Threat model and site map	1
2 TODO	6
2.1 Security and Design	6
2.2 How features are implemented	8
2.2.1 Getting user input	8
2.2.2 Creating user	8
2.2.3 Login process/Session management	10
2.2.4 ATM and Transaction	12
2.2.5 Database	12
Bibliografi	12

Introduction

The goal of the project was to create a secure banking application, resistant to OWASP TOP10 attacks.

Our application allows users to add money to their account, send it to a different user through a webpage. Visitors cannot use banking services. To become a user, a visitor must sign up for an account first.

The application is written in python with flask framework, and is using SQLAlchemy with Heroku addon resource Heroku-Postgresql as our database where we store our information.

Chapter 1

Threat model and site map

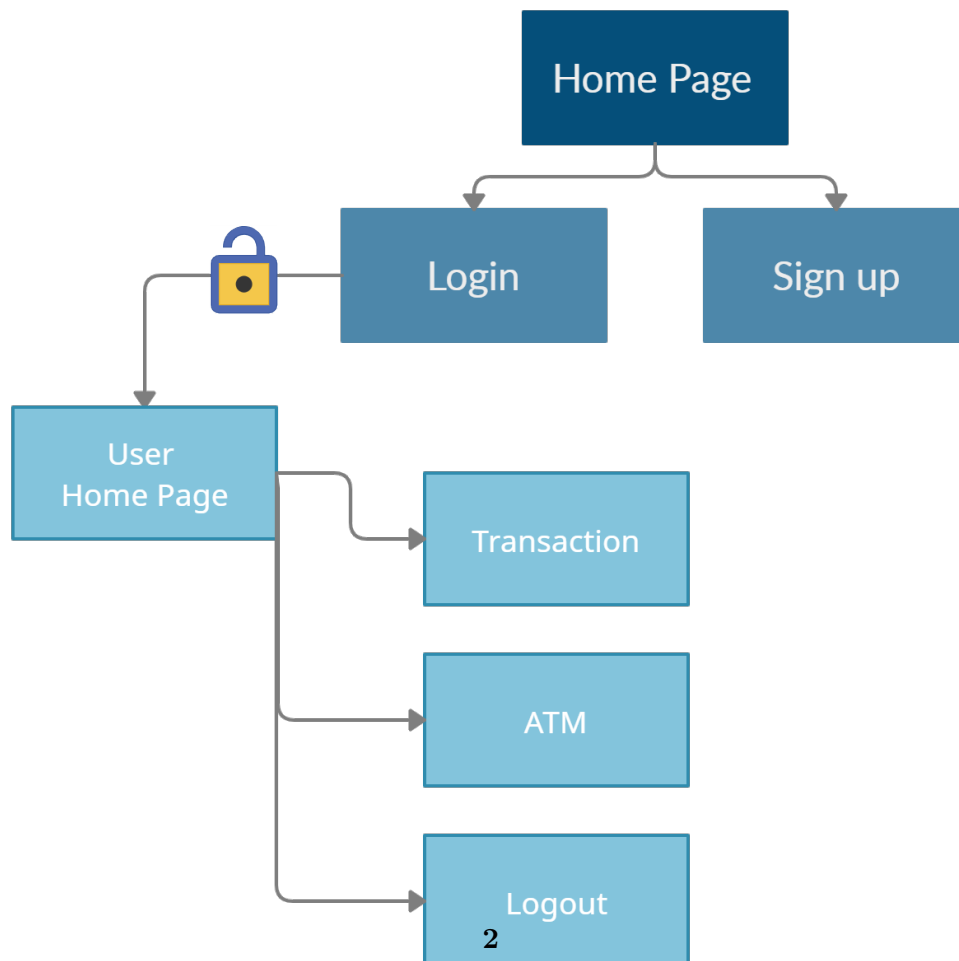


Fig. 1.1: Site map

Threat model and site map

A visitor can only see login and signup pages from the homepage. When an unlogged user tries to visit the URL of any other page, he gets redirected back to the homepage.

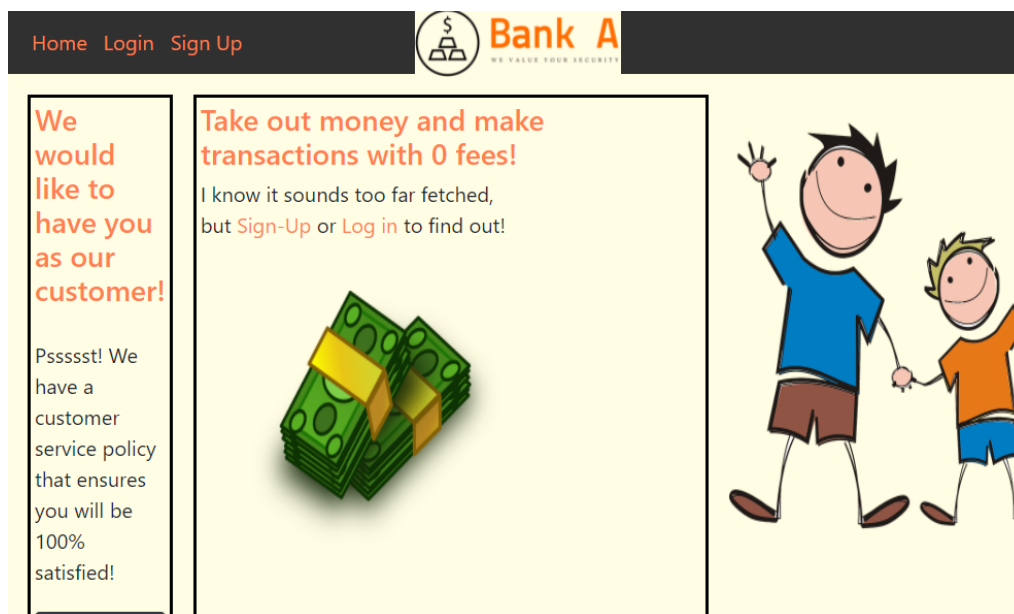


Fig. 1.2: Site landing page

After logging in, the user gets access to ATM and Transaction pages, also home-page changes to show the user's balance and transaction history.

Threat model and site map



Fig. 1.3: User homepage

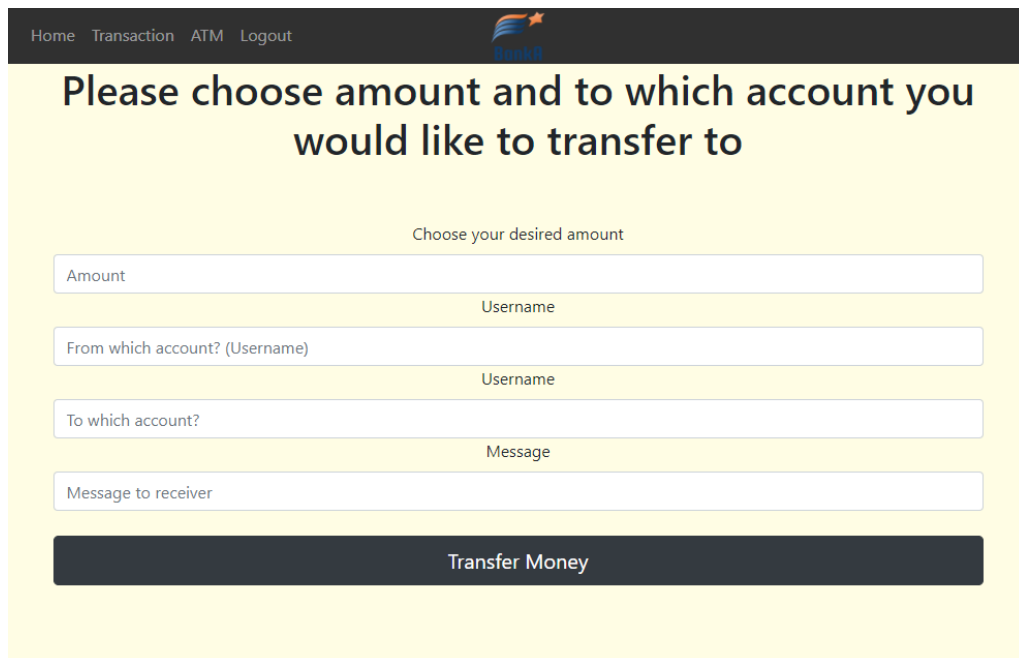


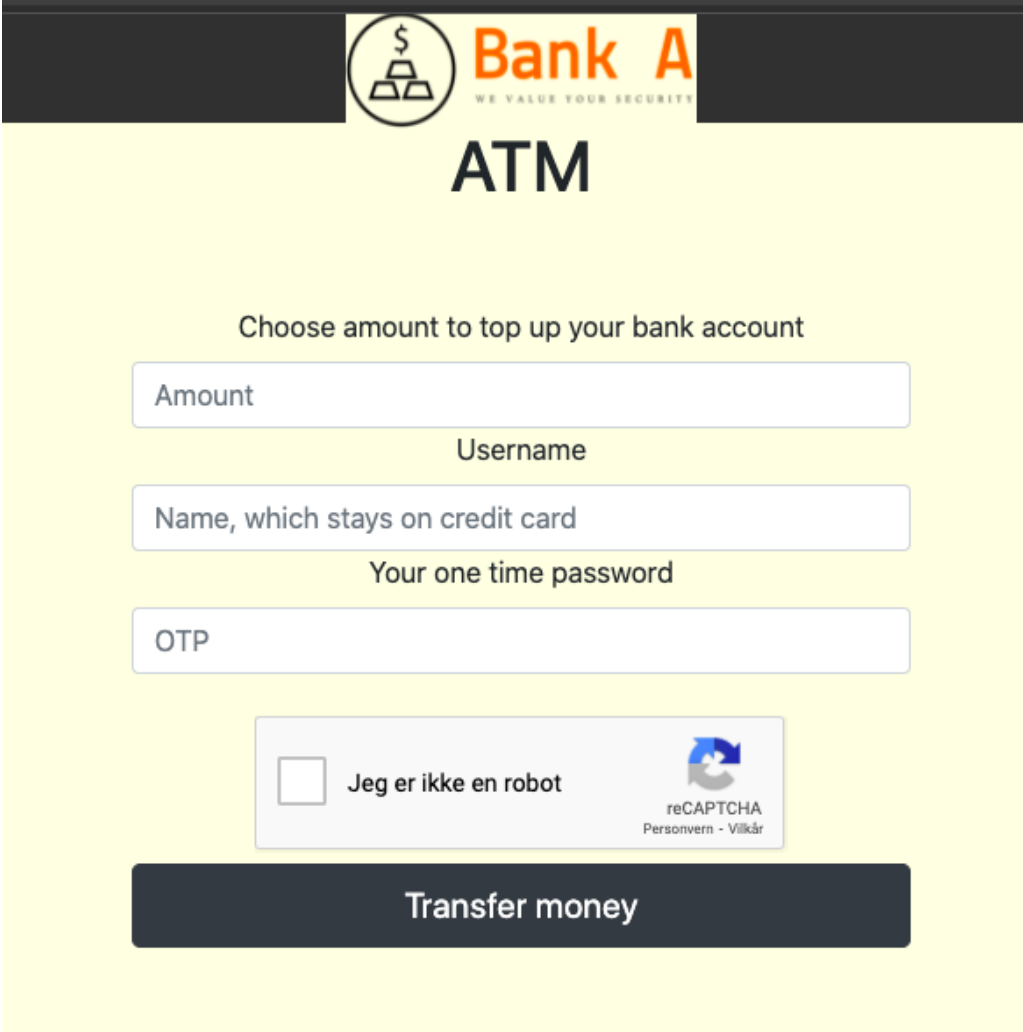
Fig. 1.4: Transaction page

To transfer cash from your own bank account in the Transaction page, the user

Threat model and site map

must know the username of the receiver, and in addition confirm his own. The user can also choose to send it with a message. This action must go through reCAPTCHA and 2FA authentication.

ATM service simulates depositing cash in a local ATM to fill your account. This page looks like this, and must also confirm/verify the name of the user, reCAPTCHA and 2FA authentication. The limit is set to 10 000kr to have realistic values. The ATM page looks like this:



The image shows a web page for 'Bank A' with a dark header bar. In the center of the header is a logo consisting of a circle with a dollar sign and three gold bars below it. To the right of the logo, the text 'Bank A' is written in orange, with the tagline 'WE VALUE YOUR SECURITY' in smaller black text below it. Below the header, the word 'ATM' is displayed in large, bold, black letters. The main content area has a light yellow background. It contains the instruction 'Choose amount to top up your bank account' in black text. Below this are four white input fields with rounded corners, each with a label above it: 'Amount', 'Username', 'Name, which stays on credit card', and 'Your one time password'. Below the 'OTP' field is a reCAPTCHA widget. It includes a small square checkbox, the text 'Jeg er ikke en robot', and the reCAPTCHA logo with the text 'reCAPTCHA' and 'Personvern - Villkår' below it. At the bottom of the page is a dark grey button with the text 'Transfer money' in white.

Fig. 1.5: ATM page

Chapter 2

TODO

2.1 Security and Design

Here is a data flow diagram to help identify a flow of data and interactions between components in the application:

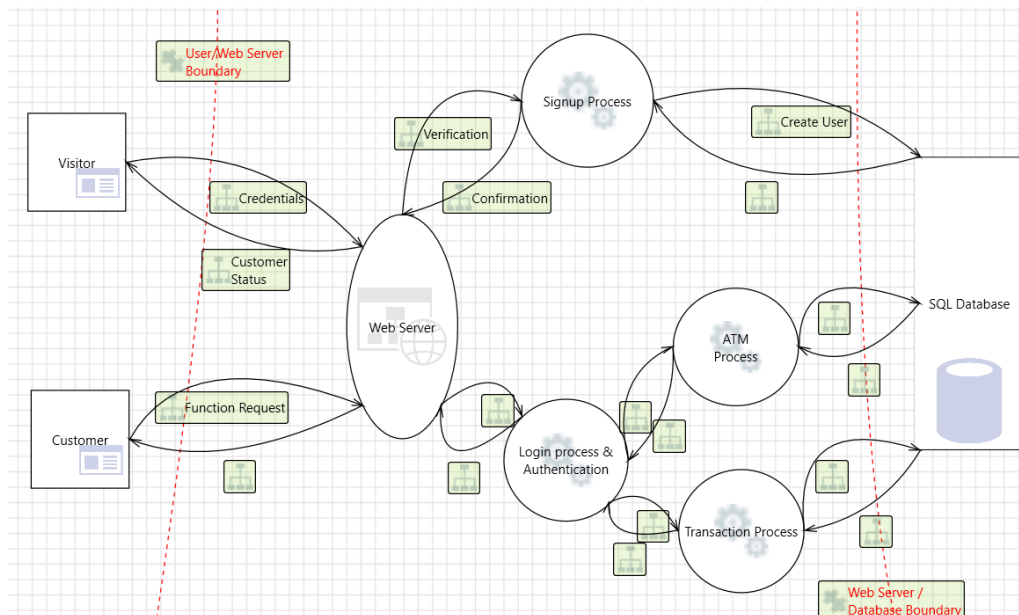


Fig. 2.1: Data flow

2.1 Security and Design

Some possible threats that we could identify by looking at the system model are:

- Spoofing
 - Credential stuffing (attacker having list of valid usernames and passwords)
 - Brute force authentication
 - Evading the authentication system and exploiting session management
 - Cross-Site Request Forgery
 - Missing session timeouts
- Tampering
 - Improper input validation
 - SQL injection
 - XML injection
 - Forcefully browsing to authenticated pages
 - tampering with URL to avoid authentication checks
- Repudiation
 - Lack of monitoring (performing unauthorized operation without the ability to be traced or detected)
- Information disclosure
 - Unauthorized access to database
 - Capturing non-encrypted data
 - Access to weakly encrypted content

In next sections we will show the basics of how our website is structured in the code and how our features are implemented within our website. Lastly we will look at how to mitigate some of these threats mentioned above both in chapter 1 but also in chapter 2.

2.2 How features are implemented

2.2 How features are implemented

2.2.1 Getting user input

WTForms extension is used to gather and validate input from users. Depending on the need, built-in or custom validators are used, unfortunately it can be easily bypassed so this is our front-end checks. We implemented it by creating a Flask form class which is shown in picture X and this method is used for every input we get from a user. For example, sign-up transactions etc.

```
class LoginForm(FlaskForm):
    email = StringField(label='Email', validators=[Email()])
    password = PasswordField(label='Password',
                             validators=[DataRequired(),
                             Length(min=1, max=100,
                             message="Password must be between"
                             +"7 and 100 characters!")])

    OTP = IntegerField(label="Your one time password",
                       validators=[DataRequired()])

    recaptcha = RecaptchaField()

    submit = SubmitField(label='Log in')
```

We validate the user input on our back-end as well using various function to check if it contains any illegal characters. More on this in chapter 4.

2.2.2 Creating user

When the signup form is sent, the input is validated both within WTforms and on backend, then a check is made to see if a user with a given name or email already exists. If not - then the given password is hashed and user data is sent to the database and saved. The application then redirects you to the homepage. We also logs this info in our log database model. If something goes wrong during the process, a red error message will appear.

```
@auth.route('/sign-up', methods=['GET', 'POST'])
```

2.2 How features are implemented

```
def sign_up():
    if current_user.is_authenticated:
        flash("You are already logged in")
        return redirect(url_for('auth.home_login'))
    form = RegisterForm()
    if form.validate_on_submit():
        init_db()

        if validate_password(form.password1.data) and validate_username(form.username.data) and validate_email(form.email.data) and validate_password(form.password1.data == form.password2.data):
            # Correct input, now check database
            success = True
            user_by_username = User.query.filter_by(username=form.username.data).first()
            user_by_email = User.query.filter_by(email=form.email.data).first()
            if user_by_username:
                flash("Username taken!", category='error')
                success = False
            if user_by_email:
                flash("Email taken!", category='error')
                success = False
            if success:
                userName = form.username.data
                # encUsername = EncryptMsg(userName)
                email = form.email.data
                # hashedEmail = FinnHash(email)
                password1 = form.password1.data
                hashedPassword = argon2.hash(password1)
                secret = pyotp.random_base32()
                user = User(username=userName, email=email, password=hashedPassword, secret=secret)
                db.session.add(user)
                db.session.commit()

                login_user(user)
                session['logged_in'] = True
                session['user'] = email
                session.permanent = True
                message = "Sign-up: User: " + str(userName) + ". Status success. T"
                db.session.add(Logs(log=message))
                db.session.commit()
```

2.2 How features are implemented

```
        return redirect(url_for('auth.two_factor_view'))
    else:
        message = "Sign-up: User: " + form.username.data + ". Status fail
            datetime.datetime.now())
        db.session.add(Logs(log=message))
        db.session.commit()
        return render_template('signup.html', form=form)
    else:
        flash("Check your input", category='error')
    return render_template('signup.html', form=form)
```

Argon2 function is being used to hash passwords.

2.2.3 Login process/Session management

When the login form is sent, the input is also validated both front-end and back-end. Then the database is searched for a user with a given email and given password is compared with hashed database password. It also checks for reCAPTCHA and the OTP (one-time-password) is correct. If everything is okay, the app redirects the user to his homepage. If something goes wrong during the process, a red error message will appear.

```
@auth.route('/login', methods=['GET', 'POST'])
def login():
    if current_user.is_authenticated:
        flash("You are already logged in")
        return redirect(url_for('auth.home_login'))
    form = LoginForm()
    if form.validate_on_submit():
        if validate_password(form.password.data) and validate_email(form.email.data):
            user = User.query.filter_by(email=form.email.data).first()
            otp = form.OTP.data
            if user is not None:
                if argon2.verify(form.password.data, user.password) and pyotp.TOTP(
                    login_user(user)
                    user.FA = True
                    message = "Log-in: User: " + user.username + "Status: Success
                        datetime.datetime.now())
```

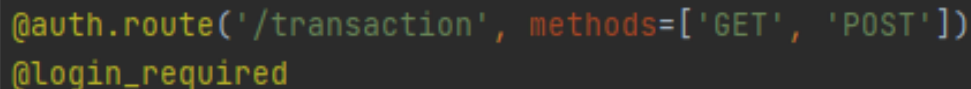
2.2 How features are implemented

```
        db.session.add(Logs(log=message))
        db.session.commit()
        session['logged_in'] = True
        message = "Log-in: User: " + user.username + "Status: Sucess. T"
        db.session.add(Logs(log=message))
        db.session.commit()
        return redirect(url_for('auth.home_login'))
    else:
        flash("Email, Password or OTP does not match!", category="error")
        message = "Log-in: User: " + user.username + "Status: Fail. T"
        db.session.add(Logs(log=message))
        db.session.commit()

        flash("Something went wrong. Please try again", category="error")
    else:
        flash("Invalid request", category='error')
        message = "Log-in: User: Invalid Input. Status: Fail. Time: " + str(d
        db.session.add(Logs(log=message))
        db.session.commit()
    return render_template('login.html', form=form)
```

Furthermore a user can log out simply by clicking on the logout button in the navigation bar, and gets redirected to the log-in page.

The application uses the Flask-Login extension for session management. It allows us to restrict some pages to be visible only for logged-in users, but we will go deeper into the security of this in chapter 4.



```
@auth.route('/transaction', methods=['GET', 'POST'])
@login_required
```

Fig. 2.2: Login required



You need to log in to access this page!

Fig. 2.3: No access

2.2 How features are implemented

2.2.4 ATM and Transaction

These services first validate data both front-end and back-end. Then authenticate given data to check if said username exists, valid amount etc. It also check for reCAPTCHA and OTP. If everything checks out, a transaction is made. If something goes wrong during the process, a red error message will appear.

2.2.5 Database

Flask-sqlalchemy extension provides support for SQLAlchemy. We use three database models – user, transaction and Logs. The user stores the user information, like username, email, password, id, token(used for 2FA) and FA check (whether they have already accesses QR-code page). The password is stores in the database using argon2 hash. We will discuss our approach on why we chose this algorithm in chapter 4.

In our transaction model we save the id of the transaction, from_user, to_user, in and out money and a message. As you have seen we don't store the account balance in the database as a security measures and to get as close to the real world as possible. We simply use a for loop to go through each transaction for that given user. The implementation looks like this:

```
class Transaction(UserMixin, db.Model):
    transaction_id = db.Column(db.Integer, primary_key=True)
    from_user_id = db.Column(db.Text, nullable=True)
    out_money = db.Column(db.Text, nullable=True)
    to_user_id = db.Column(db.Text)
    in_money = db.Column(db.Text)
    message = db.Column(db.String(120))
```

Lastly we have another database model called logs. This simply stores the id of the log and the log message, which contains the action made, user (if any), success or fail and what time it was logged/happened. We store both success made by a user and failures.

```
class Logs(db.Model):
    log_id = db.Column(db.Integer, primary_key=True)
    log = db.Column(db.Text)
```
