

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?

<b>Innhold</b>	<b>i</b>
<b>Introduction</b>	<b>iii</b>
<b>1 Threat model and site map</b>	<b>1</b>
<b>2 TODO</b>	<b>6</b>
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
<b>3 OWASP Top Ten</b>	<b>13</b>

## ?CONTENTSNAME?

---

3.1	Injection . . . . .	13
3.2	A07:2021 Identification and Authentication Failures . . . . .	14
3.3	A09 Security Logging and Monitoring Failures . . . . .	15
3.4	A05 Security Misconfiguration . . . . .	16
3.5	A04 Insecure Design . . . . .	17
3.6	A06 Vulnerable and Outdated Components . . . . .	18
3.7	A01 Broken Access Control . . . . .	18
3.8	A08 Software and Data Integrity Failures . . . . .	19

<b>Bibliografi</b>	<b>19</b>
--------------------	-----------

# 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

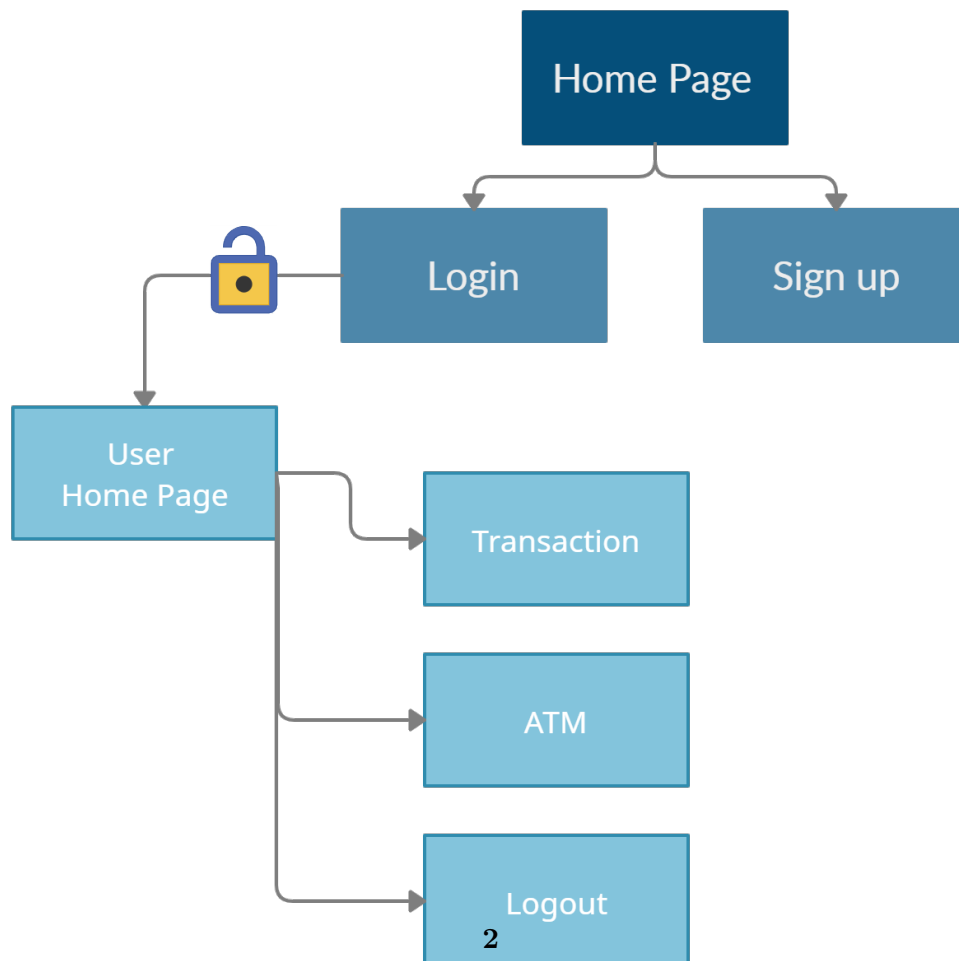
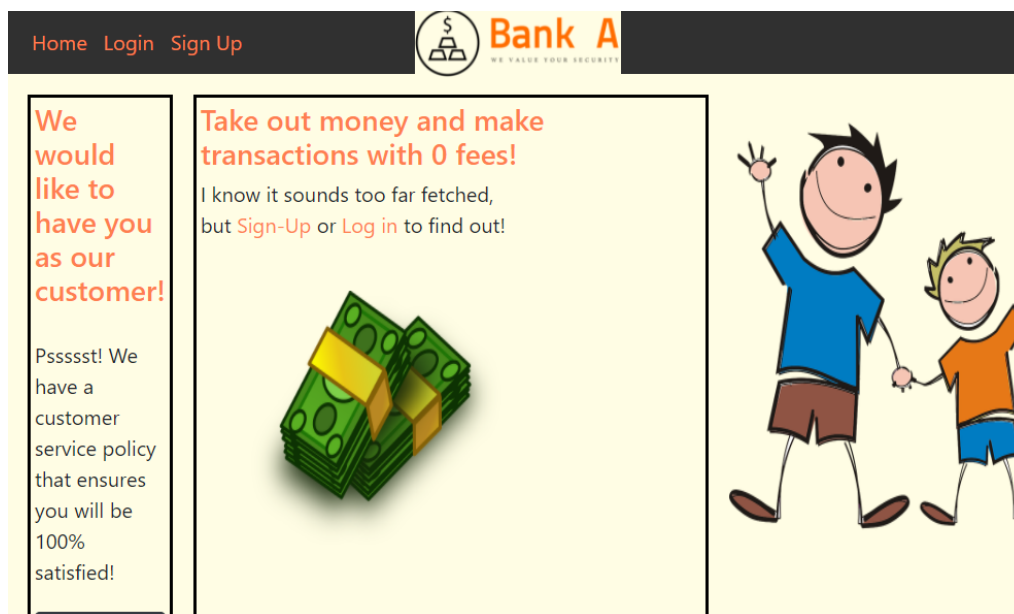


Fig. 1.1: Site map

## 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.



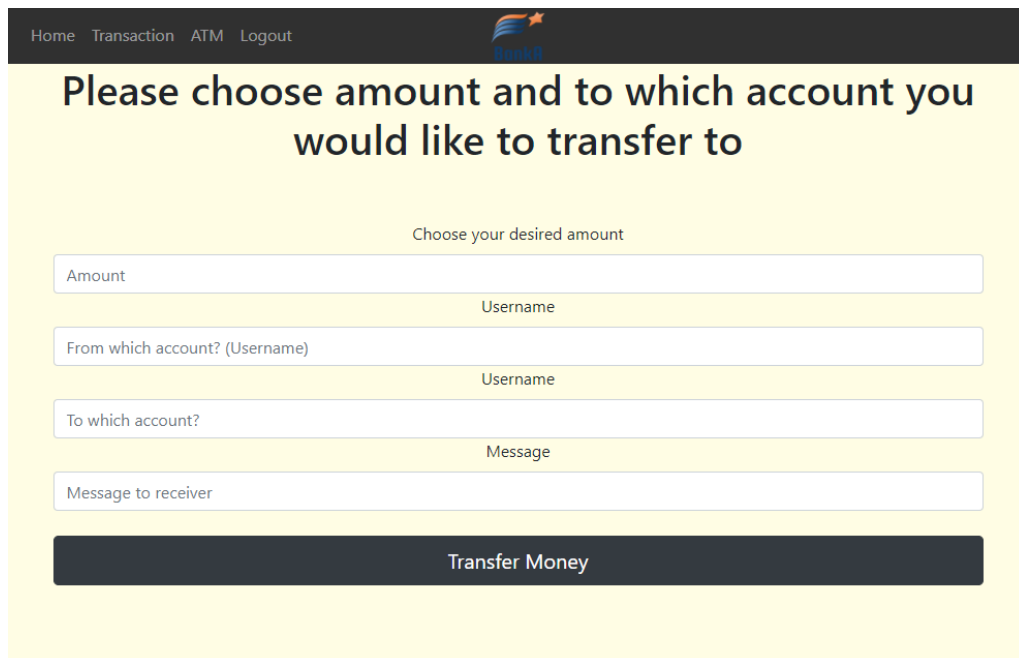
**Fig. 1.2:** Site landing page

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

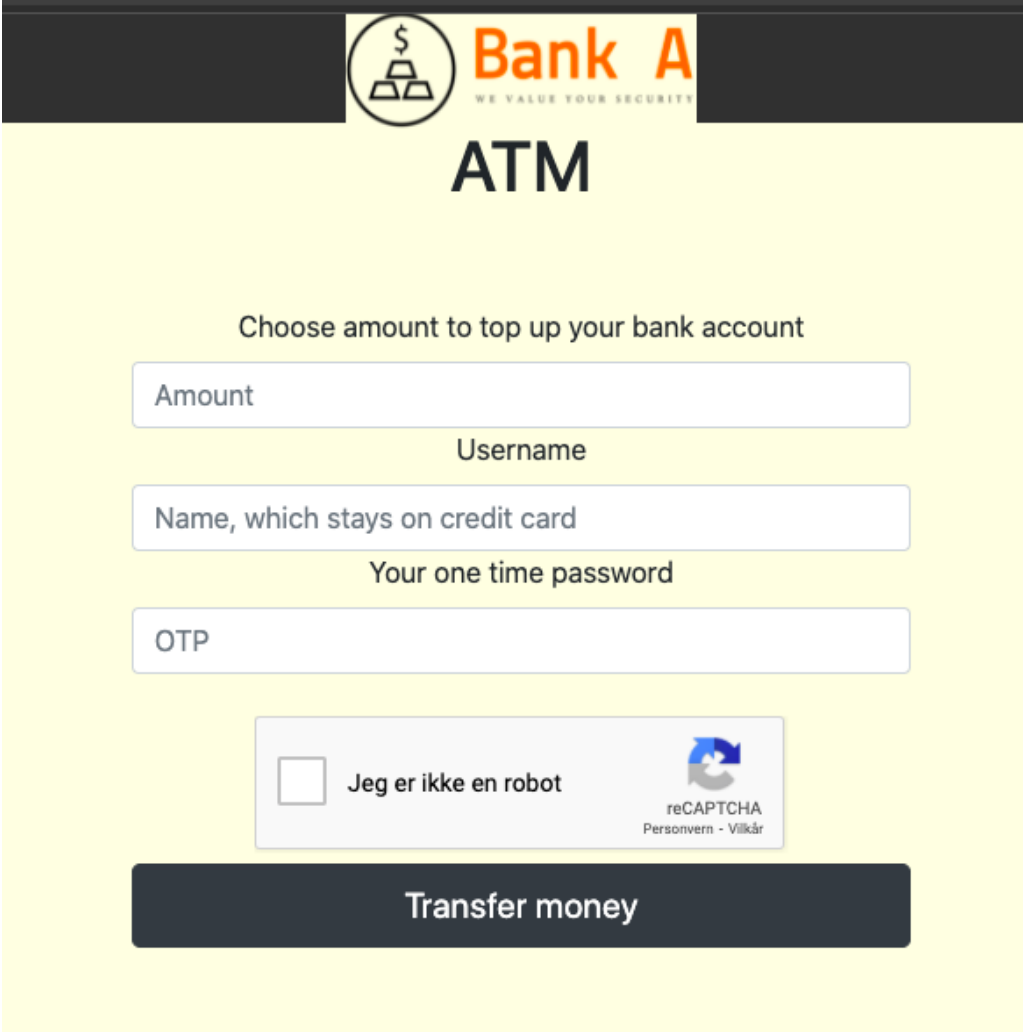


## Threat model and site map

---

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 containing the bank's logo and name. The main content area is yellow and titled 'ATM'. It contains a form with the following elements: a heading 'Choose amount to top up your bank account', a text input field labeled 'Amount', a text input field labeled 'Username', a text input field labeled 'Name, which stays on credit card', a text input field labeled 'Your one time password' with the label 'OTP' inside, a reCAPTCHA widget with the text 'Jeg er ikke en robot' and 'reCAPTCHA Personvern - Villkår', and a large dark button at the bottom labeled 'Transfer money'.

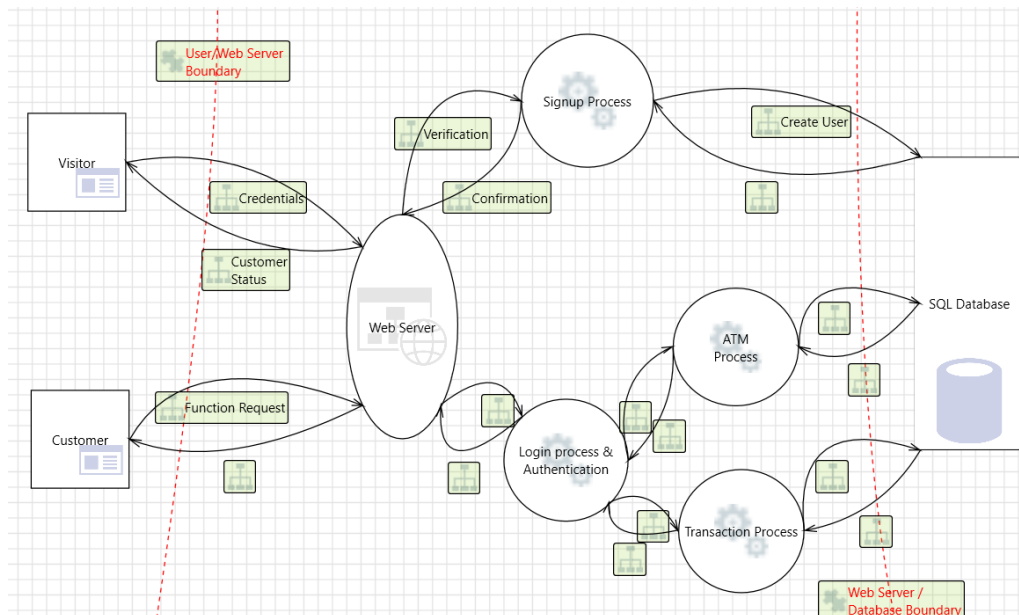
Fig. 1.5: ATM page

## 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

---

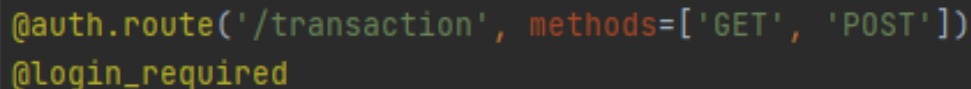
```
        db.session.add(Logs(log=message))
        db.session.commit()
        session['logged_in'] = True
        message = "Log-in: User: " + user.username + "Status: Sucess. T"
        db.session.add(Logs(log=message))
        db.session.commit()
        return redirect(url_for('auth.home_login'))
    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)
```

---



## Chapter 3

# OWASP Top Ten

This section describes how prepared are we against common threats listed in OWASP Top 10 //TODO

### 3.1 Injection

Injection attack happens when web application receives input which can be interpreted a command or query. One of the most notable injection attacks is SQL injection, which can lead to reading, modifying or deleting database by an attacker. Typically injection attacks happens through form input fields, like those in login form.

To avoid this, user input must be validated before it is sent as a query. In addition we also use SQLAlchemy which automatically quotes special characters like apostrophes in data.

(ref. <http://www.rmunnn.com/sqlalchemy-tutorial/tutorial.html>)

### 3.2 A07:2021 Identification and Authentication Failures

---

## 3.2 A07:2021 Identification and Authentication Failures

This security fault is known as Broken Authentication, which is about user's identity is breached with weaknesses with the authentication in our application, for example using automated attacks. We have implemented a few things to mitigate this, as reCAPTCHA, ratelimit, two factor authentication, strong password etc.

Firstly to avoid brute forcing and automated attacks, reCAPTCHA and ratelimit has a huge impact on this. The way we implanted this is making a reCAPTCHA user on google and verifying this in our flask WTforms. `<code>reCAPTCHA = recaptchaField() <code>`. It is possible to bypass wtforms validators/reCAPTCHA so we also made a request limiter, which makes a response and blocks out the user, which will reset in a couple of minutes. We have sat the limit to 60 POST/GET request since that should be more than enough for a normal user. Anything more than that we will block it. The code we used to implement this is:

---

```
@auth.app_errorhandler(429)
def ratelimit_handler(e):
    try:
        message = "Request Limit: User: " + current_user.username\
            + ". Time: " + str(datetime.datetime.now())
    except:
        message = "Request Limit: User: None . Time: "\
            + str(datetime.datetime.now())
    db.session.add(Logs(log=message))
    db.session.commit()
    logout_user()
    session['logged_in'] = False
    return make_response(
        jsonify(
            error="Ratelimit exceeded %s" % e.description +
                ". Our BOT killer detected unusual many request."+
                "Please slow down or turn of your BOT!")
        , 429
    )
```

---

If an attacker manages to get ahold of an users password we uses two factor

### 3.3 A09 Security Logging and Monitoring Failures

---

authentication which is compatible with google authenticator. We uses this verification after each transaction, deposit and when we log-in. This makes the attacker not being able to access that users account without also having access to their google authenticator app. This acts like a second layer to the users security. The code we used to implement this is `<code>` We have also implemented many more features to prevent this type of security breach like same message if failed log in attempt but we won't cover this since it is not the most important.

### 3.3 A09 Security Logging and Monitoring Failures

A09 has been recently more important over the years. This security breach includes not being able to detect or log activity on the website. This includes also securing the log functions so that no malicious attacks can go through this feature.

In our website we log every major request. This includes log in, sign-up, transactions, ATM deposits and if the user goes over the request limiter and the ratelimit function is called. We log both successes and failures. The way we store the data is in a separate table in the database. We simple store a string of text that is set by the request the user makes. So a typical string will contain the request (log-in/sign-up etc.), username (if exists), failed or passed and then lastly what time it happened. The way we implemented this is: `<code>`

We also verifies that the input doesn't contain any illegal character, so the way we log things doesn't become a security breach, for example injection in to the database. This is because so that it is harder to breach the log and loads of information about users becomes available to the attacker. The code for this check is:

---

```
def validate_username(username):
    """Checks for valid username (only letters and numbers)"""
    if len(username) < 2 or len(username) > 50:
        flash("Username must be longer than one character,"+
              "and shorter than fifty", category='error')
        return False

    # If only contains small and big letters, and numbers
    if re.search("[a-zA-Z0-9s]+$", username):
        return True
    flash("Username can only contain letters and numbers",
```

### 3.4 A05 Security Misconfiguration

---

```
category='error')  
return False
```

---

Another logging information we have is ReCAPTCHA. As you can see from picture below is that we can see the statistic of the uses of it and if it have detected any unusual activity. It will also send us an email if detected anything unusual. The negativity with this is that reCAPTCHA uses a long time to update. It uses a couple of days usually, therefore we cannot solely trust on this.

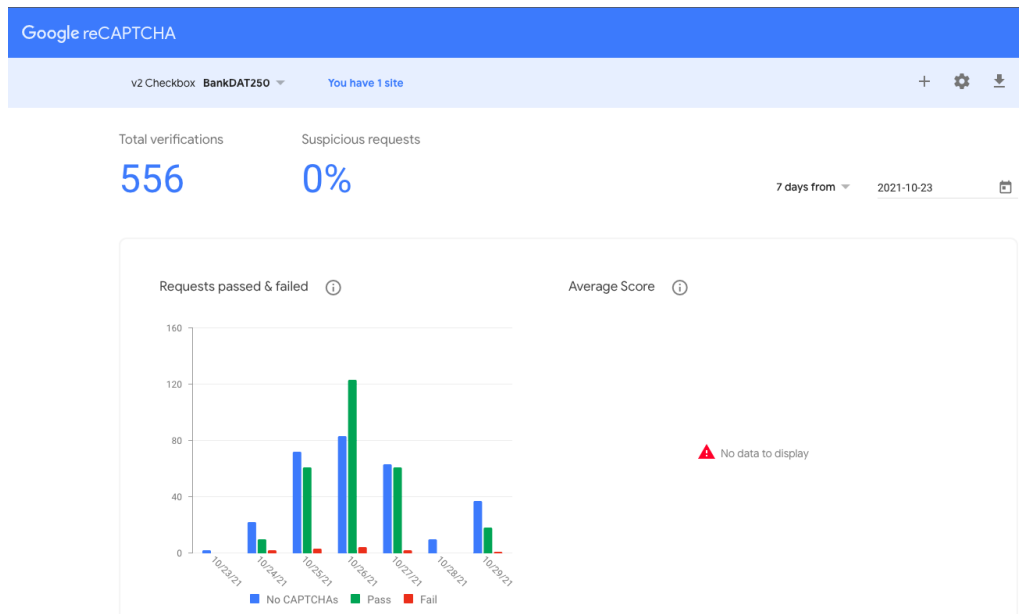


Fig. 3.1: reCAPTCHA Log

### 3.4 A05 Security Misconfiguration

This security flaw is about misconfiguration of the website, for example available administrative interfaces, not good error handling and still being in debug mode etc. For example when we are in development face we often use debug mode, which makes it easier to debug error in the website but also it automatically update your code changes (at least in flask). This can be harmful if not reverted back on deployment. This is because an attacker loves to get error messages and can easily understand how your website is flawed and in which way. This is simply

### 3.5 A04 Insecure Design

---

to remember to disable debug mode which is this line of code: `<code>` Another thing we need to is have an error handler for flask, so that the attacker won't really know what error happened if it manages to create one. In our application we simply check for any errors and if detected we flash a red error message and redirects you to the home page. This also acts as a protective layer against crashing the website. The code used is:

---

```
@auth.errorhandler(Exception)
def basic_error(e):
    """Error handler so flashes message
    and redirects user if error occur"""
    flash("Something went wrong", category='error')
    return redirect(url_for('auth.home_login'))
```

---

Since we don't use any administrative access because of the potential of being a security risk, all the admin access must have access to the deployment website and can make changes there, not directly on the website. Therefore all users have the same access when using the website.

### 3.5 A04 Insecure Design

This new category in OWASP Top 10 is about flaws in design and system architecture. Some weaknesses and technical requirements need to be thought about before starting the implementation. An example of that could be risk profiling and resource management. "A house is as strong as its foundation". While creating this application we have researched secure way to design systems and created threat model to help us recognize and evaluate possible vulnerabilities. We have also prepared for some known attack methods and evaluated which external packages will be used based on security. Furthermore everything that has been implemented we focus on security, for example by setting bounds to user input so that it don't cause buffer overflow, sanitize all inputs of a user and a robust error handler so it doesn't crash our website.

### 3.6 A06 Vulnerable and Outdated Components

---

## 3.6 A06 Vulnerable and Outdated Components

This Owasp category is about use of unmaintained or out-of-date components with known vulnerabilities. It may become hard to track all dependencies together with nested dependencies. To work against this we remove all unused dependencies and we check those which are being used, so we would know when one becomes unmaintained. All dependencies are kept on a list together with version used so the process becomes semi-automated.

## 3.7 A01 Broken Access Control

The number one of security breaches is Broken Access Control and therefore very important for our development team. This breach is about being able to acts outside of their intended permissions. For example if a user is not logged in it can't view pages on the website that they are not authorized to see or make requests.

A way to mitigate this flaw is by using flask-login. When a user login we use the login(user) code. This logs the user inn and can access routes in our website that requires a user to log-in. We use the "@login required" parameter for the implemented route. The code makes it so that the user that tries to access the page/route must be logged in. When a user logs out we call the code logout(user). This removes the flag that the user is logged in and has now not access to the pages it had when logged in. The code snippets below demonstrate this.

Lastly to mitigate this security fault is checking if the user has been inactive for more than 5 minutes. This is important if a user forgets to exit the website or logging out, because then the session cookie will still be available. The implementation looks like this:

---

```
@auth.before_request
def before_request():
    """Timeout user when inactive in 5 min"""
    flask.session.permanent = True
    current_app.permanent_session_lifetime\
        = datetime.timedelta(minutes=5)
    flask.session.modified = True
    flask.g.user = flask_login.current_user
```

---

### 3.8 A08 Software and Data Integrity Failures

---

## 3.8 A08 Software and Data Integrity Failures

This new category in 2021 is about making sure application uses trusted packages from external sources. This is because using modules from insecure sources can lead to injection of malicious code into the system.

External packages used in this application are received via python pip to which only maintainers have access to. It shouldn't be a problem as long as we don't make a typo in package name – which can be used in “typosquatting” attack, where attacker uploads a malicious code to python pip with a name similar to another package.

We can also manually verify that it was produced by the publisher by downloading package and checking signature file.