DAT250 - Information security - Project report

Joachim Andreassen
Lars Sverre Levang
Ardijan Rexhaj
Vidar Andre Bø
Elias Nodland

2020-10-25

Contents

Chapter 1

Structure

To structure our website we make use of a number of various frameworks and tools to make everything work securely. Our website is built upon the popular python web-framework Flask. The main reason as to why we make use of an external framework such as Flask, is to maintain security by using a framework that has already been tried and tested by many industry professionals. It also has packages for all the functionality that is needed to make a functional website for many different types of applications, providing tools such as session-handling and easy database management.

1.1 Encryption and hashing

For hashing our users password, we elected to use `Scrypt` over alternatives such as `bcrypt` and `argon2`.

As for encryption we have implemented Fernet into a few easy to use functions. Using fernet we encrypt user information such as email, 2FA-secret and which accounts belong to the user. This user information is encrypted using a fernet key which is then also encrypted using the users password. As a consequence, the users information can only be accessed when the password is given, this also denies maintainers access to user information.

1.2 Site structure

Figure 1 shows our sitemap. The only websites available without any form of authentication is the front page and register. From the register page we also render another html page to verify the users 2FA-token, as they need to get their 2FA token set up before being able to log in.

As illustrated below, there are two authentication walls, one for login, and one for completing transactions. Both of these require at least 2 factors to complete, those being by password, and TOTP (timed one-time passwords).

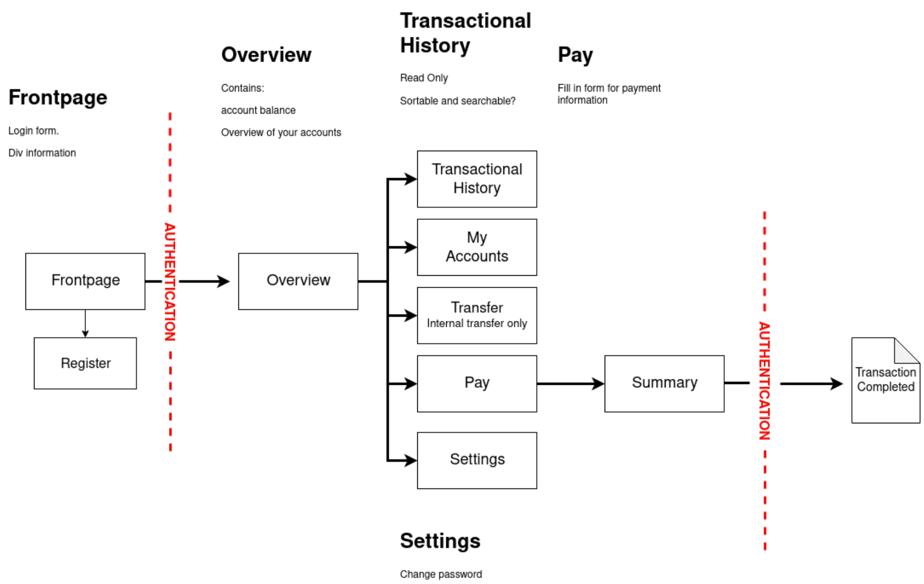


Figure 1.1: This is an illutstration of the site layout of Safecoin.tech

1.3 Tools & Packages

The tools and packages that are used to make Safecoin.tech:

1.3.1 Two-Factor Authentication

For 2FA we use TOTP codes generated for the user in their authentication app of choice. To generate these codes we make use of the packages `pyotp` and `flask-qrcode`. The QR-code is there as a user-friendly way to add the 2FA code to their app.

1.3.1.1 flask-qrcode

The main reason why we elected to use `flask-qrcode` over any other qr-code making packages is that by using `flask-qrcode` we can generate the QR-link without saving it as an image on our server. We want to manage unencrypted sensitive data as rarely as possible. By generating the QR-code when its needed we reduce the window of opportunity an attacker has to a minimum. Access to this information by anyone other than the user would be a breach of security. This is because it contains information about the email, issuer, and secret key, thus they would only need your password and defeat the purpose of two factor authentication.

Implementation:

Initialize QR-functionality in init.py

```
QRcode(app)
```

We then make the QR-link as we need it, pass it on to the relevant html, passing into it the variable qr_link which holds the OTP authentication information that gets encoded into the QR-code.

The QR-code is then generated by this function directly in the html:

```
<img src="{{ qrcode(qr_link) }}" alt="qr-kode">
```

1.3.1.2 pyotp - Python One-Time Password Library

To generate the link we use in the generated QR-code we use the `pyotp` package. Thus generating the QR-code with `flask-qrcode` is very simple, only a few lines of code are needed.

```
secret_key = pyotp.random_base32()

qr_link = pyotp.totp.TOTP(secret_key).provisioning_uri(name=form.email.data,
issuer_name="Safecoin.tech")
```

The secrey key is a randomly generated 16 character base32 secret that is compatible with Google authenticator and other OTP apps. These look like

`'POAATUEFZ5O4RFSN'` and `'LCUSQXJLUMKYIDVF'` as examples.

To generate the QR-code using the link, we return the **`qr_link`** to the new html page:

```
return render_template('TwoFactor.html', form2 = form2, qr_link = qr_link)
```

1.3.1.3 Authentication app

In order to authenticate a user needs to make use of an OTP authentication app. Without this they will not be able to sign in, as 2FA is required at log in, or to complete transactions. Users are required to add their secret key to their OTP authentication app during registration, the QR-code makes this simple.

1.3.2 Flask_Scrypt

`Flask_Scrypt` is a flask extension used to generate scrypt password hashes and random salts. The extension provides us with 3 functions that handle everything related to password management, generating salts and checking whether the provided password is the same as the one we have stored.

```
generate_password_hash(password, salt, N=2**14, r=8, p=1, buflen=64)
generate_random_salt(byte_size=64)
check_password_hash(password, password_hash, salt, N=2**14, r=8, p=1, buflen=64)
```

These three functions make hashing simple.

KDF	6 letters	8 letters	8 chars	10 chars	40-char text	80-char text
DES CRYPT	< \$1	< \$1	< \$1	< \$1	< \$1	< \$1
MD5	< \$1	< \$1	< \$1	\$1.1k	\$1	\$1.5T
MD5 CRYPT	< \$1	< \$1	\$130	\$1.1M	\$1.4k	$\$1.5 \times 10^{15}$
PBKDF2 (100 ms)	< \$1	< \$1	\$18k	\$160M	\$200k	$\$2.2 \times 10^{17}$
bcrypt (95 ms)	< \$1	\$4	\$130k	\$1.2B	\$1.5M	\$48B
scrypt (64 ms)	< \$1	\$150	\$4.8M	\$43B	\$52M	$\$6 \times 10^{19}$
PBKDF2 (5.0 s)	< \$1	\$29	\$920k	\$8.3B	\$10M	$\$11 \times 10^{18}$
bcrypt (3.0 s)	< \$1	\$130	\$4.3M	\$39B	\$47M	\$1.5T
scrypt (3.8 s)	\$900	\$610k	\$19B	\$175T	\$210B	$\$2.3 \times 10^{23}$

(2009)

The table above shows an approximate cost of brute-forcing passwords with the specified length in 2002. While technology has advanced far beyond 2002, the table still functions as an approximate estimation on how scrypt compares to many other hashing algorithms.

The hashing algorithms `bcrypt` and `argon2` were also considered as alternatives to `scrypt`. After some research we relatively quickly decided that `scrypt` would be the best fit; It not only is secure, but also very easy to use. Bcrypt as can be seen in the table above, is far less secure. `Argon2` on the other hand was a real consideration. We landed on `Scrypt` because of its' better resistance to large scale brute force attacks. It manages this by not only scaling the cpu-usage, but the memory-usage as well. This approach makes it very expensive to execute a parallel brute force attacks.(2009)

1.3.3 flask_sqlalchemy

From SQLalchemy's website: "The main goal of SQLAlchemy is to change the way you think about databases and SQL!" SQLalchemy was designed to be for efficient and high-performing database access, adapted into a simple and pythonic language, in other words a tool designed to make database-management efficient and easy-to-use. The library automates redundant and time-consuming processes while the administrator retains the control of how the database is designed and how the SQL is constructed.("SQLAlchemy," n.d.)

We use `SQLalchemy` as it is such an easy and efficient tool, while at the same time eliminating the risk of SQL-injection attacks by sanitizing all queries by default. An alternative was to go for something like sqlite3, but this would potentially make it easy for us to make a mistake in how we structure our queries.

1.3.4 flask_login

We use `flask-login` alongside redis to handle sessions. `Flask-login` handles logging in, logging out, as well as database queries for each request.

`Flask-login` is primarily used for: - Handling active user ID's in the sessions, which allows us to easily log users in and out . - Restrict views to logged-in (or logged-out) users. - Help protect users' sessions from being stolen by cookie thieves.

By tapping into flask logins user classes we can easily control how it behaves by setting properties for:

```
user.is_authenticated
user.is_active
user.is_anonymous
user.get_id()
```

is_authenticated and is_active are set up as properties that query redis, to check if the user exists in the dictionary or otherwise logs the user out.

`flask-login` provides a structured way to handle user authentication, permissions and data.

1.3.5 WTForms & flask_wtf

`WTForms` along with `flask_wtf` handle all fields where we require user to input data, such as email, password, payment, KID/message-fields etc.

`flask_wtf` handles the `FlaskForm` class, and we create children of these classes in order to customize them as needed. These can then be used to create the fields we need such as `StringField, PasswordField, BooleanField` etc. The functionality of these can then be further defined by using built in functionality, example:

```python
class LoginForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Email()],
    render_kw={"placeholder": "email@example.com"})
    password = PasswordField('Password', validators=[DataRequired()],
    render_kw={"placeholder": "password"})
    otp = IntegerField('Two-factor Authentication', validators=[DataRequired()],
    render_kw={"placeholder": "Two-Factor Authentication"})
    remember = BooleanField('Remember me')
    submit = SubmitField('Login')
```

1.3.6 base64

base64 binary to text encoding or vice versa. Required by fernet.

1.3.7 cryptography

Contains the fernet encrypt/decrypt functions that are imported.

1.3.8 JSON

Redis requires dictionary entries to be in a string/byte format. Therefore we use this package to convert python dictionaries to a string and back.

1.3.9 Redis

Redis is a local key value server used to store decrypted user information in a non-permanent manner. The data is stored with an expiry date, usually set at maximum 15 minutes ahead. If the user cannot be found in the dictionary the user is effectively logged out.

Chapter 2

Databases

Our database is designed with user security and anonymity in mind. The database is split into four tables:

```
- Users         - Encrypted, holds personal information
- Accounts      - Plain text, holds all accounts and their balance
- Transactions  - Plain text, holds all transactions
- Logging       - Plain text, used to log pseudo-anonymous activity
```

This structure was made with one thing in mind; To give the user control of their information. This is achieved by encrypting the users table, which prevents updating or reading the information without the users password. This forces developers to always keep security in mind, since he/she cant alter or read usable information from database without having the users permission. The drawback on the other hand is that we need another way to temporarily store the users decrypted information while they are logged in.

2.1 Users

Users is a database of registered users, their 2fa `secret` key, and their accounts. This table does not contain any unencrypted or hashed information.

User table Fields

```
class User(db.Model, UserMixin):
    email = db.Column(db.String(80), primary_key=True, unique=True, nullable=False)
    enEmail = db.Column(db.String(80), nullable=False)
    password = db.Column(db.String(200), nullable=False)
    enKey = db.Column(db.String(128))
    accounts = db.Column(db.String(10000))
    secret = db.Column(db.String(32 + 128))
```

```python
    # If the user is not found in redis,
    # the user is effectively not logged in
    @property
    def is_authenticated(self):
        # Check if the current_user is logged in
        if redis.get(self.email):
            # If so set data to expire 10 minutes from now
            redis.expire(self.email, 3600)
            # return true to the flask login manager
            return True
        return False

    #essentially same as above property
    @property
    def is_active(self):
        if redis.get(self.email):
            redis.expire(self.email, 3600)
            return True
        return False
```

- `email` is the users identifier. It is stored in in a hashed state, without salt. The reason for hashing this is simply to have a fixed length for database lookup entries.
- `enEmail` is the users clear text email encrypted with the users Fernet key.
- `password` is the users hashed password concatenated with the salt used to hash it. The **`salt`** is the only thing stored in plain text. The resulting hash and salt is always the same length and thereby we can split this information into its respective parts.
- `enKey` is the users encryption-key that has been encrypted by the users hashed password without salt. This creates a new unique hash that is different from the salted password hash above. This hash is not done for security reasons, its purpose is to create a fixed length 32 byte key to encrypt the encryption key.
- `accounts` holds account ownership information that is encrypted with the encryption key.
- `secret` is the secret key used to generate One Time Passcodes (OTP's).

As shown all the data in this database table is either encrypted or hashed. While we certainly will always do everything we can to secure both physical and remote access, the worst case scenario should always be considered. The way this is structured, the user could potentially leave the website open on their computer and still be confident that no one could make changes to their account.

2.2 Accounts

Accounts is a table containing all banking accounts and their balance. Accounts are “use and dispose” meaning they are be easy to create and not limited in supply.

```
class Account(db.Model):
    number = db.Column(db.String(11), unique=True, nullable=False, primary_key=True)
    balanceField = db.Column(db.String(256))
    # tallet viser til maks lengde av et siffer

    # Henter verdien fra databasen og konverterer til streng
    @property
    def balance(self):
        return int(self.balanceField)

    # setter verdien i databasen, dersom den er en int, blir den til streng
    @balance.setter
    def balance(self, value):
        # Sjekker om jeg faar en int, ellers skal det ikke fungere.
        if type(value) == int:
            value = str(value)
            self.balanceField = value
        else:
            raise Exception("Can only set this to be an int")
```

- `account` is the account number. We use the standard `xxxx yy zzzzc` format.
  - `xxxx` is a bank registration number which identify the bank.
  - `yy` signifies account type, where we have only one.
  - `zzzz` is the customers account number.
  - `c` is the “control-digit”. Calculated by **xxxxyyzzzz**%10
- `balance` is a property of the account, it converts the balance into an int since its stored as a string. This is to prevent rounding errors.

Nothing in the accounts table is encrypted.

2.3 Transactions

Transactions is a database containing all transactions.

```
class Transactions(db.Model):
    transactionID = db.Column(db.Integer, primary_key=True)
    accountFrom = db.Column(db.String(80), nullable=False)
    accountTo = db.Column(db.String(80), nullable=False, )
    amountDB = db.Column(db.String(256))
    message = db.Column(db.String(90))
```

```
time = db.Column(DateTime, default=datetime.datetime.utcnow,nullable=False)
```

- `transactionID` is a unique transaction id associated with the transaction.
- `accountFrom` transaction origin.
- `accountTo` transaction destination.
- `amountDB` amount being sent.
- `message` a short message/number sent along with the transaction. This could plain text or a KID-message.
- `time` a time-stamp for the transaction.

The data in transactions table is not encrypted, but since the users table which holds ownership information is encrypted there is no direct way to find which user made the transaction. A user could potentially compromise their own or the receivers anonymity by writing personal information in the message field.

Chapter 3

Registration

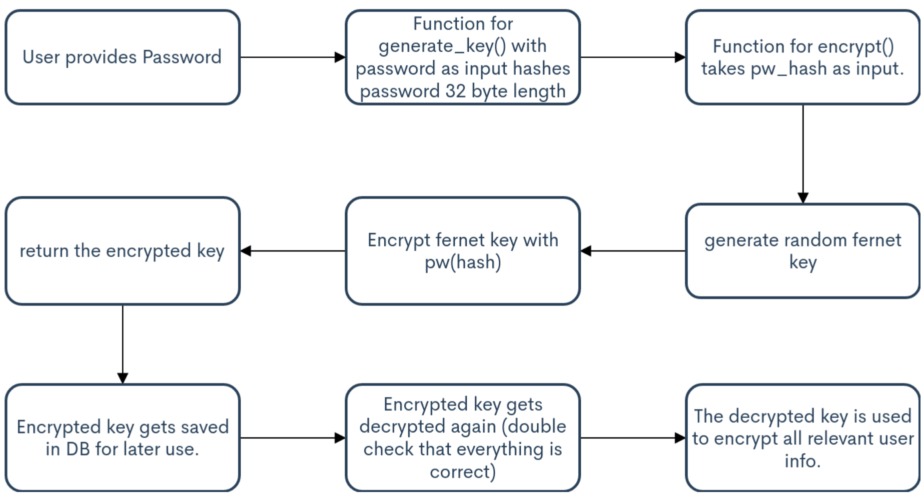


Figure 3.1: Illustrating the flow when a user creates a password, and how the encryption key for the users info gets made.

On the registration page, the user will have to input the following.

3.1 Email

At Safecoin, the users email will be used as the user name. During the registration process, the email will be hashed with scrypt's hash-function without a salt, and compared with the other hashed emails in the database. The page will return a generic error if it is already in use.

3.2 Password

Password requirements: - Minimum 12 characters in length - Not an integer - Cannot match top 10000 most common passwords - Cannot contain the users email address

The users password is cryptographically hashed with scrypt and a random salt. These are as mentioned in the previous chapter concatenated and stored together.

Scrypt allows the password to be of an arbitrary length, within reason.

3.3 TwoFactor Authentication

If the user passes both the email and password-checks, the user will be sent over to a page to complete two-factor-authentication setup. The user will be presented a QR-code that can be scanned in a authenticator app. During this registration process, we temporarily store the user information in its encrypted final form until the user finishes 2FA setup and is only written to the database upon completion.

Chapter 4

Authentication

Authentication of the registered user is required for the following actions: - login - transfer of money - creation of bank account - deletion of bank account - deletion of user

In order to make verification easy to maintain we use the same verification function across the site. The only exception is during registration, where the information is being generated. This function takes user input and verifies it. It also syncs the users information stored in redis with the database.

```
def verifyUser(email, password, addToActive=False):
  # hash the email
  if email is None:
      hashed_email = current_user.email
  else:
      hashed_email = flask_scrypt.generate_password_hash(email, "")

  # create user class with information from database
  userDB = User.query.filter_by(email=hashed_email).first()

  # if the user does not exist in database
  if userDB is None:
      return False, None, None

  # format password from database
  DBpw = userDB.password.encode('utf-8')

  # check if the hashed email is the same as the one in the database,
  #just a double check.
  # Strictly not necessary, but just seems logical to do.
  emailOK = hashed_email.decode('utf-8') == userDB.email.decode('utf-8')
```

```python
# Verify that the password is correct
pwOK = flask_scrypt.check_password_hash(password.encode('utf-8'), DBpw[:88],
       DBpw[88:176])

# If the password is correct and email exists in the database
if emailOK and pwOK:

    # Decrypt the users encryption key
    decryptKey = decrypt(password, userDB.enKey.encode('utf-8'), True)

    # Decrypt the secret key
    secret_key = decrypt(decryptKey, userDB.secret.encode('utf-8'))

    if addToActive:
        # sync redis with database! This is a function.
        redis_sync(decryptKey, hashed_email)

    return True, userDB, secret_key

return False, None, None
```

Chapter 5

Accounts and transactions

The ownership of an account is determined by which user holds the account number in their accounts list. In order for a user to release ownership if an account the account balance must be 0. An account can loose its owner but it can never be deleted from the accounts database, this ensures that we cant generate the same account-number twice.

Transactions was written with security in mind, it was written as if redis was compromised and not trustworthy. Therefore for every transaction the user must provide their respective password and only then be able to transfer the given amount.

It is quite extensive but summarized these are the steps.

- User is logged in.
- Verify the user with `verifyUser` function (same as mentioned in authentication)
- Check ownership
- Both accounts exist
- Check transaction validity eg. check sum of both account balances is the same before and after, sender has the required balance etc.
- Do the actual transaction of balance
- Update both balances
- Write new transaction information to transactions database.

```
def submitTransaction(password, accountFrom, accountTo, amount, message):
    accountFrom=str(accountFrom)
    accountTo=str(accountTo)
    #Check user password
    verified, userDB, ubrukt = verifyUser(None, password)

    if verified is False:
```

```
        return False
        #Decrypt and check user account with user database
    decryptKey = decrypt(password, userDB.enKey.encode('utf-8'), True)
    accountsDB = decrypt(decryptKey, userDB.accounts.encode('utf-8'))


    accountsDict = DBparseAccounts(accountsDB)

    #is the account one of the users accounts
    if str(accountFrom) in accountsDict:
        #if above checks out
        #do transfer
        accountDBFrom = Account.query.filter_by(number=accountFrom).first()
        accountDBTo = Account.query.filter_by(number=accountTo).first()

        if TransactionChecks(accountDBFrom, amount, accountDBTo, accountsDict,message)==Fals
            return False

        accountDBFrom.balance -= amount
        accountDBTo.balance += amount

        # LOGGING
        trans=Transactions()
        trans.accountTo = accountTo
        trans.accountFrom = accountFrom
        trans.amount = amount
        trans.message = message
        trans.eventID = 'transaction'


        db.session.add(accountDBFrom)
        db.session.add(accountDBTo)
        db.session.add(trans)
        db.session.commit()

        redis_sync(decryptKey,current_user.email)
        return True
        #add the transaction to the transaction history
    else:
        # user does not own this account, return
        return False


def TransactionChecks(accountFrom, amount, accountTo, accountsDict, message):
    #internal transfer check that user balance remains unchanged
```

```
    #check for stuff
    if accountFrom==None or accountTo==None or accountFrom==accountTo:
        return False
    if amount<1 or type(amount)!=int or amount>accountFrom.balance:
        return False

    sumBefore=accountFrom.balance+accountTo.balance
    fromBalance = accountFrom.balance
    toBalance   = accountTo.balance
    fromAfter   = fromBalance - amount
    toAfter     = toBalance + amount

    sumAfter = fromAfter + toAfter

    if sumBefore!=sumAfter:
        return False

    if illegalChar(message,90):
        return False

    return True

def illegalChar(text, maxlength,alphabet="abcdefghijklmnopqrstuvwxyzæøå0123456789 "):
    if text==None:
        return False

    try:
        text=str(text)
    except:
        return True

    if len(text)>maxlength:
        return True

    #Transform name to lowercase and check if its not in the alphabet
    for letter in text.lower():
        if letter not in alphabet:
            return True
    return False
```

Chapter 6

OWASP Top Ten

In this chapter we will go through the each prevention we have implemented for the the OWASP Top 10 vulnerabilities. Some of the OWASP 10 prevention measures are not implemented because they are not applicable or too complex to implement for this project.

Every subsection title is the prevention measure as written in the OWASP Top 10 document ("OWASP Top 10" 2017)

6.1 Injection

6.1.1 The preferred option is to use a safe API, which avoids the use of the interpreter entirely or provides a parameterized interface, or migrate to use Object Relational Mapping Tools (ORMs).

We use the flask-API `flask_sqlalchemy` and strictly only query the database through it. This is not inherently safe on it's own, but used correctly it mitigates or completely eliminates the possibility of an SQL-injection attack.

Example:

```
def getAccount(account_number):
    account: Account = Account.query.filter_by(number=account_number).first()
    return account
```

The above "query.filter_by" sanitizes input by default and is the only method used to query the database. The only exception being:

```
#Sanitize input illegalChar(text,maxlength,"string Allowed chars")
illegal=illegalChar(transForm.accountSelect.data,11,"0123456789")
```

```
if (str(transForm.accountSelect.data) in str(accountList)) and illegal==False:
    query=Transactions.query.filter((Transactions.accountFrom ==
            transForm.accountSelect.data) | (Transactions.accountTo ==
              transForm.accountSelect.data))

    TransList=QueryToList(query, accountList, transForm.accountSelect.data)
else:
    pass
    #TODO LOG THIS!
```

Here we get input from the user about which account history the user wants to view. The method "query.filter" is not considered safe. Since the account number can be converted to an integer and back, any dangerous value would prevent this conversion, thus raise and exception and skip the query. The account number is also looked up in the user's account list. These safety measures make this query safe.

6.1.2 Use positive or "whitelist" server-side input validation. This is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications.

As seen in the example above, input validation is used to ensure database queries are safe from sql-injection. We specifically require there to be no special characters and for general input fields, we use an illegal character checking function:

It checks the input text character by character and only allows the characters that exists it the alphabet to be used. In the above code you can see how its implemented and that it allows nothing but numbers, not even spaces.

```
def illegalChar(text, maxlength,alphabet="abcdefghijklmnopqrstuvwxyzæøå0123456789 "):
    if text==None:
        return False

    try:
        text=str(text)
    except:
        return True

    if len(text)>maxlength:
        return True

    #Transform name to lowercase and check if its not in the alphabet
    for letter in text.lower():
        if letter not in alphabet:
```

```
        return True
    return False
```

6.2 Broken Authentication

6.2.1 Where possible, implement multi-factor authentication to prevent automated, credential stuffing, brute force, and stolen credential re-use attacks.

We implement multi-factor authentication and Google's reCAPTHCA on the login page to prevent automated attacks. Wherever authentication is required after the user is logged in, two factor authentication is also required.

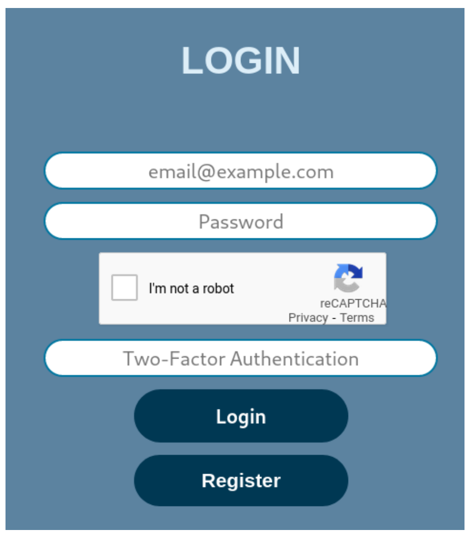


Figure 6.1: This is a screenshot of the login page demonstrating 2FA and reCAPTHCA at Safecoin.tech

6.2.2 Implement weak-password checks, such as testing new or changed passwords against a list of the top 10000 worst passwords.

Password requirements:

- Minimum 12 characters in length
- Not an integer
- Cannot match top 10000 most common passwords that are 12 characters or longer

- Cannot contain the users email address

The users password is cryptographically hashed with scrypt and a random salt. The password and its salt are then concatenated and stored together.

Scrypt allows the password to be of an arbitrary length.

Code for the password requirements:

```python
def getPasswordViolations(errList, password, email):
    if type(password) != str:
        errList.append("An error occurred!")
        return

    if isCommonPassword(password):
        errList.append("Password is too common")
        return

    if "safecoin" in password.lower() or email.lower() in password.lower():
        errList.append("Please choose a better password")
        return

    # Password params
    cfg = ConfigParser()
    cfg.read("safecoin/config.ini")
    policy = cfg["passwordPolicy"]
    try:
        want_length = int(policy["length"])
    except (KeyError, TypeError):
        want_length = 12

    if len(password) < want_length:
        errList.append(f"Password should be at least {want_length} characters")

def isCommonPassword(password):
    with open("safecoin/commonPasswords.txt", "r") as f:
        for weakpwd in f:
            if password == weakpwd[:-1]:
                return True
    return False
```

6.2.3 Ensure registration, credential recovery, and API pathways are hardened against account enumeration attacks by using the same messages for all outcomes

While developing the website we had generic errors in mind as a measure to prevent account enumeration. The code-snippet below contains a short example:

```
    else:
        # Generisk feilmelding dersom noe går galt
        flash('Something went wrong. Please try again.')
        log_loginattempt(False, userDB.email)
else:
    # Generisk feilmelding dersom noe går galt
    flash('Something went wrong. Please try again.')
    log_loginattempt(False, userDB.email)


#example from register page where this is the only error message
flash("Couldn't continue, due to an error", "error")
```

An attacker will in this case not be able to determine whether an account exists in the database or not.

6.2.4 Use a server-side, secure,built-in session manager that generates a new random session ID with high entropy after login. Session IDs should not be in the URL, be securely stored and invalidated after logout, idle, and absolute timeouts.

We use flask_login to manage users sessions, this is built in and this generates a new session ID with every browser instance when session_protection is set to strong.

```
app.config['SECURITY_TOKEN_MAX_AGE'] = 3600
app.config['PERMANENT_SESSION_LIFETIME'] = 3600
app.config['REMEMBER_COOKIE_DURATION'] = timedelta(minutes=0)

login_manager = LoginManager()
login_manager.init_app(app)
login_manager.login_view = '/login'
#Ensures a new session ID is created for every browser instance
login_manager.session_protection = "strong"


#Logout route
@app.route("/logout")
@login_required
def logout():
    #slett ifra redis først ellers er current_user ikke definert.
    redis.delete(current_user.email)
    log_logout(current_user.email)
```

```
    #Logg så ut fra login manager
    logout_user()
    return redirect(url_for('home')), disable_caching
```

6.3 Sensitive Data Exposure

To limit the risk of data exposure, we attempt to handle as little personal information as possible.

6.3.1 Make sure to encrypt all sensitive data at rest.

6.3.2 Don't store sensitive data unnecessarily. Discard it as soon as possible or use PCI DSS compliant tokenization or even truncation. Data that is not retained cannot be stolen.

These measures solve both 6.3.3 and 6.3.4

All sensitive data is only temporarily stored in redis with an expiry date, otherwise it is at rest and stored in an encrypted form which is inaccessible without user input. This way ensure that we do not store or have access to decrypted data longer than necessary.

Redis sync, syncs redis with database, requires users to be verified beforehand.

```
def redis_sync(deKey, hashed_mail):
    if type(deKey) == str:
        deKey = deKey.encode('utf-8')
    # Get user from database
    userDB = User.query.filter_by(email=hashed_mail).first()

    # create user dict for json dump
    userInfo = {}

    # Add plaintext email as a key
    # Check if its a string
    if type(userDB.enEmail) == str:
        userInfo['email'] = decrypt(deKey, userDB.enEmail.encode('utf-8')).decode('utf-8')
    else:
        userInfo['email'] = decrypt(deKey, userDB.email).decode('utf-8')

    # If the user has any accounts
    if userDB.accounts is not None:
        # decrypt them
        if type(userDB.accounts) == str:
            accounts = decrypt(deKey, userDB.accounts.encode('utf-8'))
```

```
        else:
            accounts = decrypt(deKey, userDB.accounts)

        # add them to the dictionairy of the user
        userInfo['accounts'] = DBparseAccounts(accounts)

    userInfo = json.dumps(userInfo)
    # add it to the redis database

    redis.set(userDB.email, userInfo)
    # set the expiration time of the data added
    # 900 seconds= 15 minutes
    redis.expire(userDB.email, 900)
```

6.3.3 Store passwords using strong adaptive and salted hashing functions with a work factor (delay factor), such as Argon2, scrypt, bcrypt, or PBKDF2

The password stored in the database is generated with `scrypt`. We use the default settings here, as can be seen in the code-snippet below. These settings make sure that memory and CPU-usage is very high, and therefore very expensive in a brute-force attack.

```
# ---- SALT AND HASH PASSWORD --------------------------------------------------
salt = flask_scrypt.generate_random_salt()

# add hashed password to user dictionairy
# This can be directly saved to database later
userDict['password'] = flask_scrypt.generate_password_hash(form.password.data,
                          salt) + salt
```

This function is the flask_scrypt default settings which we use to generate the password hash.

```
def generate_password_hash(password, salt, N=2**14, r=8, p=1, buflen=64):


    if PYTHON2:
        password = password.encode('utf-8')
        salt = salt.encode('utf-8')
    pw_hash = scrypt_hash(password, salt, N, r, p, buflen)
    return enbase64(pw_hash)
```

6.3.4 Disable caching for responses that contain sensitive data

We have disabled browser caching on the website, this is so that the browser does not store potentially personal information and so that users/bad-actors can go back and look at previously browser cached pages after logout.

```
disable_caching = {'Cache-Control': 'no-cache, no-store, must-revalidate',
                   'Pragma': 'no-cache',
                   'Express': '0'}
```

6.4 Broken Access Control

6.4.1 With the exception of public resources, deny by default.

For all of the pages except for the login and register page, the user is required to be logged in. This is accomplished with the built in login-manager, see example below:

```
@app.route('/profile/', methods=["GET", "POST"])
@login_required
def profilePage():
    form = DeleteUserForm()
```

6.4.2 Implement access control mechanisms once and re-use them throughout the application, including minimizing CORS usage.

From the start of the project we deliberately implemented a single function to verify users and adapted this function to suit our needs. The function is the only verification function used throughout the entire backend.

```
def verifyUser(email, password, addToActive=False):
   # hash the email
   if email is None:
       hashed_email = current_user.email
   else:
       hashed_email = flask_scrypt.generate_password_hash(email, "")

   # create user class with information from database
   userDB = User.query.filter_by(email=hashed_email).first()

   # if the user does not exist in database
   if userDB is None:
```

```
        return False, None, None

    # format password from database
    DBpw = userDB.password.encode('utf-8')

    # check if the hashed email is the same ass the one in the database,
    #just a double check.
    # Strictly not necessary, but just seems logical to do.
    emailOK = hashed_email.decode('utf-8') == userDB.email.decode('utf-8')
    # boolean to compare with

    # Verify that the password is correct
    pwOK = flask_scrypt.check_password_hash(password.encode('utf-8'),
        DBpw[:88], DBpw[88:176])

    # Check if the password is correct and email exists in the database
    if emailOK and pwOK:

        # decrypt the users encryption key
        decryptKey = decrypt(password, userDB.enKey.encode('utf-8'), True)

        # Decrypt the secret key
        secret_key = decrypt(decryptKey, userDB.secret.encode('utf-8'))

        if addToActive:
            # sync with redis!
            redis_sync(decryptKey, hashed_email)

        return True, userDB, secret_key

    return False, userDB, None
```

6.5 Cross-Site Scripting

6.5.1 Preventing XSS requires separation of untrusted data from active browser content

We implement a very strict policy on user input, XSS would apply for the transaction message. This message will show up on both the sender and receiver's side. We prevent XSS by enforcing strict input sanitization. The only allowed characters can be seen in the code below:

```
illegalChar(text, maxlength,alphabet="abcdefghijklmnopqrstuvwxyzæøå0123456789 "):
```

6.6 Using Components with known vulnerabilities

6.6.1 Remove unused dependencies, unnecessary features, components, files, and documentation.

We went through the entire source code of the project and checked for unused dependencies. While developing the website we also ensured that code and documentation are kept completely separate. Upon completion we went through the entire directory and removed unused or unnecessary files and folders.

6.6.2 Monitor for libraries and components that are unmaintained or do not create security patches for older versions

To combat this issue we use github's built in dependabot. Dependabot automatically scans and creates pull-requests for outdated dependencies. Dependabot will also alert us if any libraries get deprecated. From there we would have to take action accordingly. ("Dependabot" 2020)

6.7 Insufficient Logging & Monitoring

6.7.1 Ensure that logs are generated in a format that can be easily consumed by a centralized log management solutions.

For logging we've made functions for each `eventType`, which log the related information for each of them. Some of these types are login, failed login, register, transaction and many others. These functions look like this:

```
def log_register(is_validated: bool, hashedEmail: str):
    msg = "Created:"
    if is_validated:
        msg += "YES"
    else:
        msg += "NO"
    log(msg, "register", hashedEmail)
```

We pass the `message`, `eventType` and `hashedEmail` into a log functions which takes the information, builds the database request and commits it to the database:

```
def log(message: str = "NA", eventType: str = "NA", hashedEmail: str = "NA"):
    req = requestLogs(message=message, eventType=eventType, email=hashedEmail)
    db.session.add(req)
    db.session.commit()
```

Everything on our website that is user-interactable is logged. Every log contains the following info:

- eventID

  Incremental ID for easy storing and sorting.

- email

  Hashedemail of the user.

- eventType

  What kind of event. E.g. transaction, login, failed login, register, delete user etc.

- message

  Event message. Eventually KID message for transfers.

- time

  Timestamp of the event.

With this we could even backtrack if we needed. It is stored in a database that would be easy for admins to access and is in a format that could easily be consumed later by some centralized log management solution.

Refrences

"Dependabot." 2020. https://dependabot.com/.

"OWASP Top 10." 2017. https://owasp.org/www-project-top-ten/2017/; The OWASP Foundation.

Percival, C. 2009. "Stronger Key Derivation via Seqential Memory-Hard Functions." http://www.tarsnap.com/scrypt/scrypt.pdf.

"SQLAlchemy." n.d. https://www.sqlalchemy.org/.

Threat Modeling Report

Created on 10/25/2020 1:32:45 PM

Threat Model Name:
Owner:
Reviewer:
Contributors:
Description:
Assumptions:
External Dependencies:

Threat Model Summary:

Not Started	0
Not Applicable	0
Needs Investigation	0
Mitigation Implemented	23
Total	23
Total Migrated	0

Diagram: Diagram 1

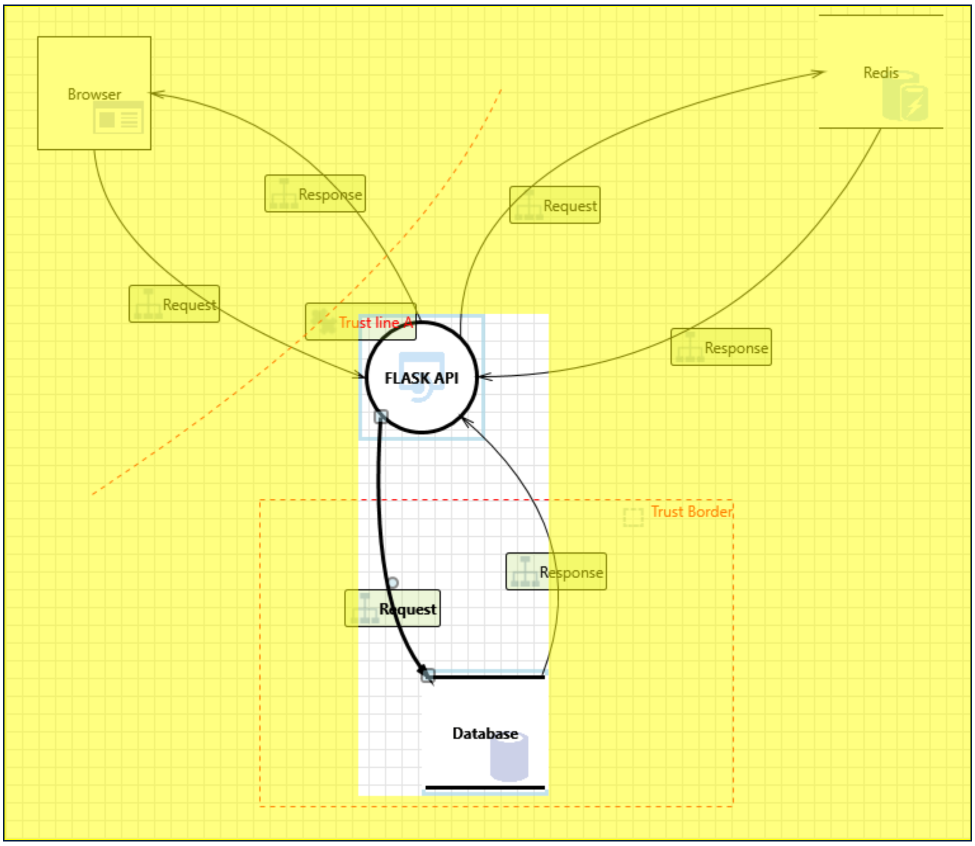
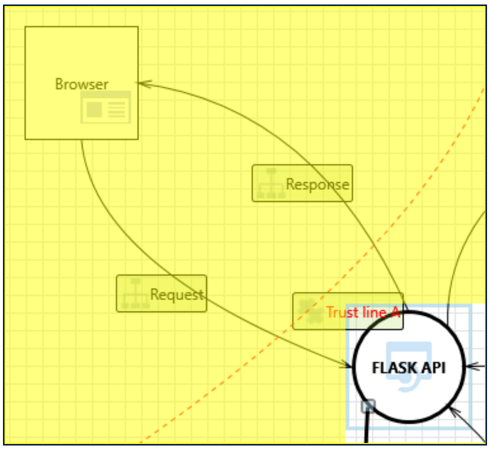


Diagram 1 Diagram Summary:

Not Started	0
Not Applicable	0
Needs Investigation	0
Mitigation Implemented	23
Total	23
Total Migrated	0

Interaction: Request



1. An adversary may gain unauthorized access to Web API due to poor access control checks [State: Mitigation Implemented] [Priority: High]

Category: Elevation of Privileges
Description: An adversary may gain unauthorized access to Web API due to poor access control checks
Justification: Strict access control, and multiple checks inplace. (2FA, existance in redis, strong passwords) Integrated login manager
Short Description: A user subject gains increased capability or privilege by taking advantage of an implementation bug
Possible Mitigation(s): Implement proper authorization mechanism in ASP.NET Web API. Refer: https://aka.ms/tmtauthz#authz-aspnet
SDL Phase: Implementation

2. An adversary can gain access to sensitive information from an API through error messages [State: Mitigation Implemented] [Priority: High]

Category: Information Disclosure
Description: An adversary can gain access to sensitive data such as the following, through verbose error messages - Server names - Connection strings - Usernames - Passwords - SQL procedures - Details of dynamic SQL failures - Stack trace and lines of code - Variables stored in memory - Drive and folder locations - Application install points - Host configuration settings - Other internal application details
Justification: Implemeted generic error messages where this could occur.
Short Description: Information disclosure happens when the information can be read by an unauthorized party
Possible Mitigation(s): Ensure that proper exception handling is done in ASP.NET Web API. Refer: https://aka.ms/tmtxmgmt#exception
SDL Phase: Implementation

3. An adversary may retrieve sensitive data (e.g, auth tokens) persisted in browser storage [State: Mitigation Implemented] [Priority: High]

Category: Information Disclosure
Description: An adversary may retrieve sensitive data (e.g, auth tokens) persisted in browser storage
Justification: This is mitigated, after logout its no longer possible to view cache.
Short Description: Information disclosure happens when the information can be read by an unauthorized party
Possible Mitigation(s): Ensure that sensitive data relevant to Web API is not stored in browser's storage. Refer: https://aka.ms/tmtdata#api-browser
SDL Phase: Implementation

4. An adversary can gain access to sensitive data by sniffing traffic to Web API [State: Mitigation Implemented] [Priority: High]

Category: Information Disclosure
Description: An adversary can gain access to sensitive data by sniffing traffic to Web API
Justification: HTTPS
Short Description: Information disclosure happens when the information can be read by an unauthorized party
Possible Mitigation(s): Force all traffic to Web APIs over HTTPS connection. Refer: https://aka.ms/tmtcommsec#webapi-https
SDL Phase: Implementation

5. An adversary can gain access to sensitive data stored in Web API's config files [State: Mitigation Implemented] [Priority: Medium]

Category: Information Disclosure
Description: An adversary can gain access to the config files. and if sensitive data is stored in it, it would be compromised.
Justification: Mitigated through use of well known and tested API.
Short Description: Information disclosure happens when the information can be read by an unauthorized party
Possible Mitigation(s): Encrypt sections of Web API's configuration files that contain sensitive data. Refer: https://aka.ms/tmtconfigmgmt#config-sensitive

SDL Phase: Implementation

6. An adversary may inject malicious inputs into an API and affect downstream processes [State: Mitigation Implemented] [Priority: High]

Category: Tampering

Description: An adversary may inject malicious inputs into an API and affect downstream processes

Justification: User input sanitization, checking each char in inpt string. Only allowing highly restricted set of characters.

Short Description: Tampering is the act of altering the bits. Tampering with a process involves changing bits in the running process. Similarly, Tampering with a data flow involves changing bits on the wire or between two running processes

Possible Mitigation(s): Ensure that model validation is done on Web API methods. Refer: https://aka.ms/tmtinputval#validation-api Implement input validation on all string type parameters accepted by Web API methods. Refer: https://aka.ms/tmtinputval#string-api

SDL Phase: Implementation

7. An adversary can gain access to sensitive data by performing SQL injection through Web API [State: Mitigation Implemented] [Priority: High]

Category: Tampering

Description: SQL injection is an attack in which malicious code is inserted into strings that are later passed to an instance of SQL Server for parsing and execution. The primary form of SQL injection consists of direct insertion of code into user-input variables that are concatenated with SQL commands and executed. A less direct attack injects malicious code into strings that are destined for storage in a table or as metadata. When the stored strings are subsequently concatenated into a dynamic SQL command, the malicious code is executed.

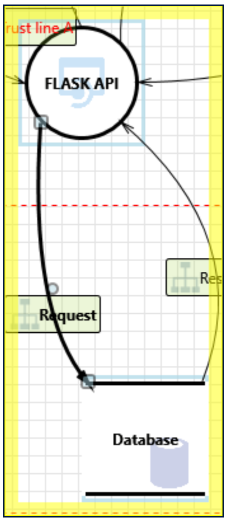
Justification: all queries are done through flask, with fucntion query.filter_by() that sanitizes by default. We do use raw sql commands in trasaction history, however its mitigated with illegalChars() function

Short Description: Tampering is the act of altering the bits. Tampering with a process involves changing bits in the running process. Similarly, Tampering with a data flow involves changing bits on the wire or between two running processes

Possible Mitigation(s): Ensure that type-safe parameters are used in Web API for data access. Refer: https://aka.ms/tmtinputval#typesafe-api

SDL Phase: Implementation

Interaction: Request



8. An adversary can gain unauthorized access to database due to lack of network access protection [State: Mitigation Implemented] [Priority: High]

Category: Elevation of Privileges

Description: If there is no restriction at network or host firewall level, to access the database then anyone can attempt to connect to the database from an unauthorized location

Justification: Integrated database into flask API. Technically possible to retrive database by connecting to server and downloading it. Mitigated with server access control and firewalls.

Short Description: A user subject gains increased capability or privilege by taking advantage of an implementation bug

Possible Mitigation(s): Configure a Windows Firewall for Database Engine Access. Refer: https://aka.ms/tmtconfigmgmt#firewall-db

SDL Phase: Implementation

9. An adversary can gain unauthorized access to database due to loose authorization rules [State: Mitigation Implemented] [Priority: High]

Category: Elevation of Privileges

Description: Database access should be configured with roles and privilege based on least privilege and need to know principle.

Justification: Integrated database. Random secret key.

Short Description: A user subject gains increased capability or privilege by taking advantage of an implementation bug

Possible Mitigation(s): Ensure that least-privileged accounts are used to connect to Database server. Refer: https://aka.ms/tmtauthz#privileged-server Implement Row Level Security RLS to prevent tenants from accessing each others data. Refer: https://aka.ms/tmtauthz#rls-tenants Sysadmin role should only have valid necessary users . Refer: https://aka.ms/tmtauthz#sysadmin-users

SDL Phase: Implementation

10. An adversary can gain access to sensitive data by performing SQL injection [State: Mitigation Implemented] [Priority: High]

Category: Information Disclosure

Description: SQL injection is an attack in which malicious code is inserted into strings that are later passed to an instance of SQL Server for parsing and execution. The primary form of SQL injection consists of direct insertion of code into user-input variables that are concatenated with SQL commands and executed. A less direct attack injects malicious code into strings that are destined for storage in a table or as metadata. When the stored strings are subsequently concatenated into a dynamic SQL command, the malicious code is executed.

Justification: If the flask API is compromised then this is possible, mitigtaed through using a well known API

Short Description: Information disclosure happens when the information can be read by an unauthorized party

Possible Mitigation(s): Ensure that login auditing is enabled on SQL Server. Refer: https://aka.ms/tmtauditlog#identify-sensitive-entities Ensure that least-privileged accounts are used to connect to Database server. Refer: https://aka.ms/tmtauthz#privileged-server Enable Threat detection on Azure SQL database. Refer: https://aka.ms/tmtauditlog#threat-detection Do not use dynamic queries in stored procedures. Refer: https://aka.ms/tmtinputval#stored-proc

SDL Phase: Implementation

11. An adversary can gain access to sensitive PII or HBI data in database [State: Mitigation Implemented] [Priority: High]

Category: Information Disclosure

Description: Additional controls like Transparent Data Encryption, Column Level Encryption, EKM etc. provide additional protection mechanism to high value PII or HBI data.

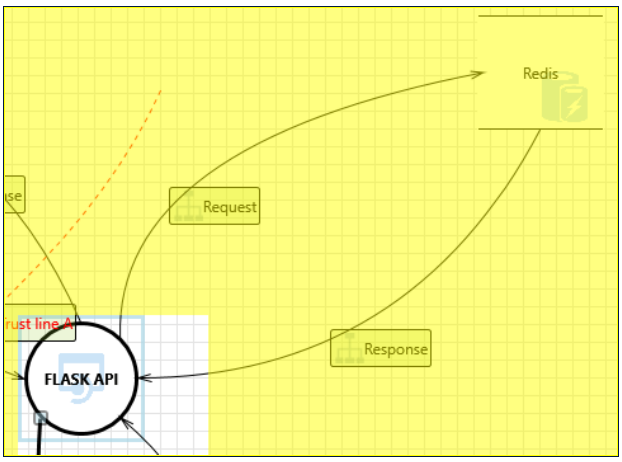
Justification: Personal data is encrypted and secured with user password. Ownership lookup requires password and is not possible to do without the password either for developers or users.

Short Description: Information disclosure happens when the information can be read by an unauthorized party

Possible Mitigation(s): Use strong encryption algorithms to encrypt data in the database. Refer: https://aka.ms/tmtcrypto#strong-db Ensure that sensitive data in database columns is encrypted. Refer: https://aka.ms/tmtdata#db-encrypted Ensure that database-level encryption (TDE) is enabled. Refer: https://aka.ms/tmtdata#tde-enabled Ensure that database backups are encrypted. Refer: https://aka.ms/tmtdata#backup Use SQL server EKM to protect encryption keys. Refer: https://aka.ms/tmtcrypto#ekm-keys Use AlwaysEncrypted feature if encryption keys should not be revealed to Database engine. Refer: https://aka.ms/tmtcrypto#keys-engine

SDL Phase: Implementation

Interaction: Request



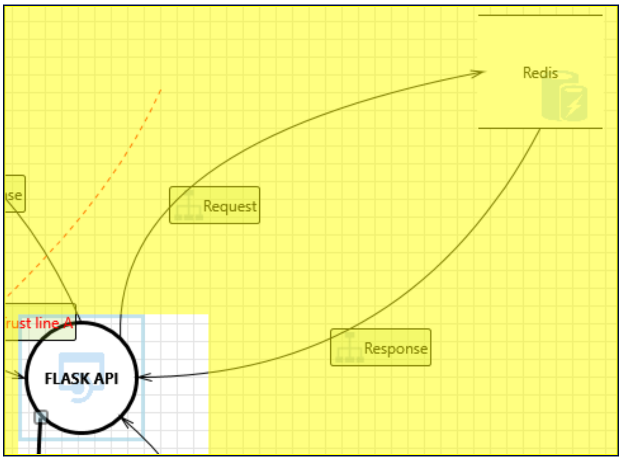
12. An adversary can read sensitive data by sniffing traffic to Redis [State: Mitigation Implemented] [Priority: High]

Category: Information Disclosure

Description: An adversary can read sensitive data by sniffing traffic to Redis

Justification:	Redis runs locally, but possible. Again there are more serious issues if this is the case.
Short Description:	Information disclosure happens when the information can be read by an unauthorized party
Possible Mitigation(s):	Ensure that communication to Redis is over SSL/TLS. Configure Redis such that only connections over SSL/TLS are permitted. Also ensure that connection string(s) used by clients have the ssl flag set to true (I.e. ssl=true). Refer: https://aka.ms/tmt-th14.
SDL Phase:	Implementation

Interaction: Response



13. An adversary can gain access to sensitive data by performing SQL injection through Web API [State: Mitigation Implemented] [Priority: High]

Category:	Tampering
Description:	SQL injection is an attack in which malicious code is inserted into strings that are later passed to an instance of SQL Server for parsing and execution. The primary form of SQL injection consists of direct insertion of code into user-input variables that are concatenated with SQL commands and executed. A less direct attack injects malicious code into strings that are destined for storage in a table or as metadata. When the stored strings are subsequently concatenated into a dynamic SQL command, the malicious code is executed.
Justification:	No raw user input is used to generate queries to redis.
Short Description:	Tampering is the act of altering the bits. Tampering with a process involves changing bits in the running process. Similarly, Tampering with a data flow involves changing bits on the wire or between two running processes
Possible Mitigation(s):	Ensure that type-safe parameters are used in Web API for data access. Refer: https://aka.ms/tmtinputval#typesafe-api
SDL Phase:	Implementation

14. An adversary may inject malicious inputs into an API and affect downstream processes [State: Mitigation Implemented] [Priority: High]

Category:	Tampering
Description:	An adversary may inject malicious inputs into an API and affect downstream processes
Justification:	requests are generated by Web API and user input is sanitized.
Short Description:	Tampering is the act of altering the bits. Tampering with a process involves changing bits in the running process. Similarly, Tampering with a data flow involves changing bits on the wire or between two running processes
Possible Mitigation(s):	Ensure that model validation is done on Web API methods. Refer: https://aka.ms/tmtinputval#validation-api Implement input validation on all string type parameters accepted by Web API methods. Refer: https://aka.ms/tmtinputval#string-api
SDL Phase:	Implementation

15. An adversary can gain access to sensitive data stored in Web API's config files [State: Mitigation Implemented] [Priority: Medium]

Category:	Information Disclosure
Description:	An adversary can gain access to the config files. and if sensitive data is stored in it, it would be compromised.
Justification:	Technically true, but redis config editing requries root privileges.
Short Description:	Information disclosure happens when the information can be read by an unauthorized party
Possible Mitigation(s):	Encrypt sections of Web API's configuration files that contain sensitive data. Refer: https://aka.ms/tmtconfigmgmt#config-sensitive
SDL Phase:	Implementation

16. An adversary can gain access to sensitive information from an API through error messages [State: Mitigation Implemented] [Priority: High]

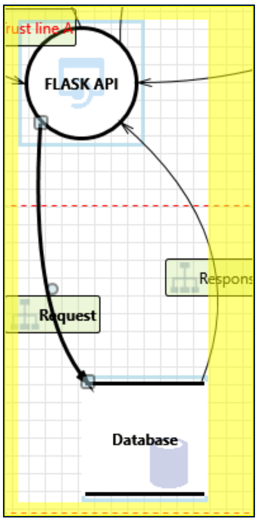
Category:	Information Disclosure
Description:	An adversary can gain access to sensitive data such as the following, through verbose error messages - Server names - Connection strings - Usernames - Passwords - SQL procedures - Details of dynamic SQL failures - Stack trace and lines of code - Variables stored in memory - Drive and folder locations - Application install points - Host configuration settings - Other internal application details

Justification: redis errors are never displayed to user.
Short Description: Information disclosure happens when the information can be read by an unauthorized party
Possible Mitigation(s): Ensure that proper exception handling is done in ASP.NET Web API. Refer: https://aka.ms/tmtxmgmt#exception
SDL Phase: Implementation

17. An adversary may gain unauthorized access to Web API due to poor access control checks [State: Mitigation Implemented] [Priority: High]

Category: Elevation of Privileges
Description: An adversary may gain unauthorized access to Web API due to poor access control checks
Justification: Mitigated with server password, redis server only runs locally.
Short Description: A user subject gains increased capability or privilege by taking advantage of an implementation bug
Possible Mitigation(s): Implement proper authorization mechanism in ASP.NET Web API. Refer: https://aka.ms/tmtauthz#authz-aspnet
SDL Phase: Implementation

Interaction: Response



18. An adversary can gain access to sensitive data by performing SQL injection through Web API [State: Mitigation Implemented] [Priority: High]

Category: Tampering
Description: SQL injection is an attack in which malicious code is inserted into strings that are later passed to an instance of SQL Server for parsing and execution. The primary form of SQL injection consists of direct insertion of code into user-input variables that are concatenated with SQL commands and executed. A less direct attack injects malicious code into strings that are destined for storage in a table or as metadata. When the stored strings are subsequently concatenated into a dynamic SQL command, the malicious code is executed.
Justification: given that the web API is compromised this is possible, but mitigated by using a well tested API.
Short Description: Tampering is the act of altering the bits. Tampering with a process involves changing bits in the running process. Similarly, Tampering with a data flow involves changing bits on the wire or between two running processes
Possible Mitigation(s): Ensure that type-safe parameters are used in Web API for data access. Refer: https://aka.ms/tmtinputval#typesafe-api
SDL Phase: Implementation

19. An adversary may inject malicious inputs into an API and affect downstream processes [State: Mitigation Implemented] [Priority: High]

Category: Tampering
Description: An adversary may inject malicious inputs into an API and affect downstream processes
Justification: Possible, but we strictly enforce input sanitization which mitigates this threat.
Short Description: Tampering is the act of altering the bits. Tampering with a process involves changing bits in the running process. Similarly, Tampering with a data flow involves changing bits on the wire or between two running processes
Possible Mitigation(s): Ensure that model validation is done on Web API methods. Refer: https://aka.ms/tmtinputval#validation-api Implement input validation on all string type parameters accepted by Web API methods. Refer: https://aka.ms/tmtinputval#string-api
SDL Phase: Implementation

20. An adversary can gain access to sensitive data stored in Web API's config files [State: Mitigation Implemented] [Priority: Medium]

Category: Information Disclosure
Description: An adversary can gain access to the config files. and if sensitive data is stored in it, it would be compromised.
Justification: Yes, this does require server login, and thus mitigated thorugh server security implementations.

Short Description: Information disclosure happens when the information can be read by an unauthorized party

Possible Mitigation(s): Encrypt sections of Web API's configuration files that contain sensitive data. Refer: https://aka.ms/tmtconfigmgmt#config-sensitive

SDL Phase: Implementation

21. An adversary can gain access to sensitive data by sniffing traffic to Web API [State: Mitigation Implemented] [Priority: High]

Category: Information Disclosure

Description: An adversary can gain access to sensitive data by sniffing traffic to Web API

Justification: Yes, but this is running locally on server so you would need to actually be logged in for this to occur. By then we have larger issues.

Short Description: Information disclosure happens when the information can be read by an unauthorized party

Possible Mitigation(s): Force all traffic to Web APIs over HTTPS connection. Refer: https://aka.ms/tmtcommsec#webapi-https

SDL Phase: Implementation

22. An adversary can gain access to sensitive information from an API through error messages [State: Mitigation Implemented] [Priority: High]

Category: Information Disclosure

Description: An adversary can gain access to sensitive data such as the following, through verbose error messages - Server names - Connection strings - Usernames - Passwords - SQL procedures - Details of dynamic SQL failures - Stack trace and lines of code - Variables stored in memory - Drive and folder locations - Application install points - Host configuration settings - Other internal application details

Justification: Mitigated, either no error is given or a generic one is given by the actual function that preforms the action. ciritcal functions like verify user are encapsulated in try: except: to catch all errors.

Short Description: Information disclosure happens when the information can be read by an unauthorized party

Possible Mitigation(s): Ensure that proper exception handling is done in ASP.NET Web API. Refer: https://aka.ms/tmtxmgmt#exception

SDL Phase: Implementation

23. An adversary may gain unauthorized access to Web API due to poor access control checks [State: Mitigation Implemented] [Priority: High]

Category: Elevation of Privileges

Description: An adversary may gain unauthorized access to Web API due to poor access control checks

Justification: Web API can only be access through server, wich is secured with strong password and firewall.

Short Description: A user subject gains increased capability or privilege by taking advantage of an implementation bug

Possible Mitigation(s): Implement proper authorization mechanism in ASP.NET Web API. Refer: https://aka.ms/tmtauthz#authz-aspnet

SDL Phase: Implementation