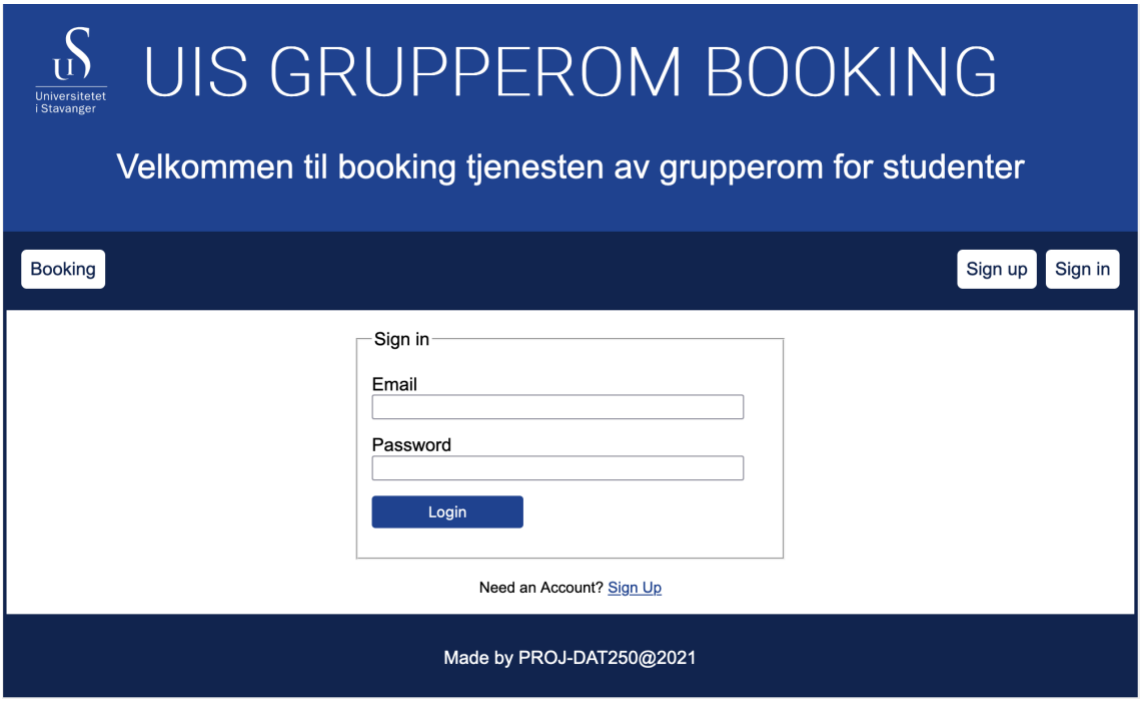
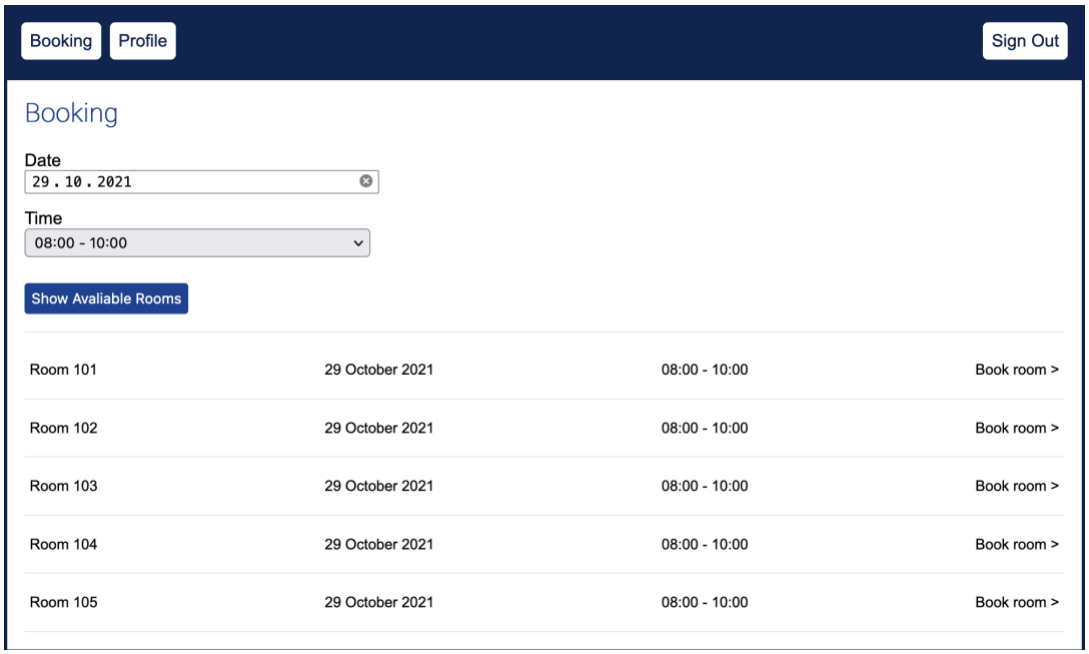
DAT250 project - Room booking

Intro

We chose to develop a website for booking rooms at UiS. When entering the website, you need to log in with your account. If you do not have an account, you need to sign up with an uis mail.



When you are logged in, you can access the booking and profile page. In the booking page you select the date and time slot you want to book a room, and the available rooms at that time are listed.



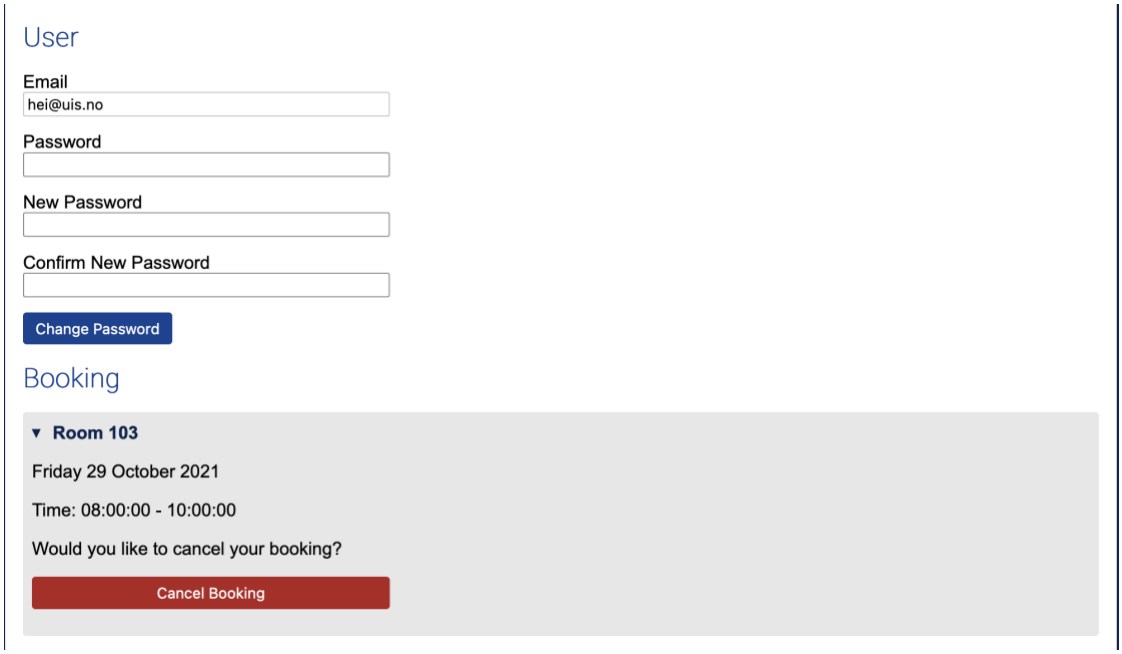
Then you click “Book room” and then “Confirm booking”. The room will now be booked if you do not already have a booking.

If you already have a booking:

You are also not allowed to book rooms back in time. In the picture below I tried to book a room from 8-10 when it was 1 o’clock.

You are allowed to book a new room when your booking expires. You do not need to cancel your expired booking to book a new room. The profile page shows your email and current

booking. You can cancel your booking and change your password if you'd like.



A1: Injection

SQL injection can happen when someone tries to manipulate SQL queries by sending in user input. User input is often used to fetch or write to the database. Here the attacker can write something that can for example fetch usernames and passwords from the database, or even delete the whole database. You are vulnerable to SQL injection if you concatenate the user input into your query without validating it, shown below.

To prevent this attack, we escape characters that can be used to do this. This is done by the db.execute command from the database library, shown below.

- **`db.execute`** takes a SQL query with **?** placeholders for any user input, and a tuple of values to replace the placeholders with. The database library will take care of escaping the values so you are not vulnerable to a *SQL injection attack*.

Code example:

```
booking = db.execute(
    'SELECT user_id FROM room_time where room_id = ?'
    ' and from_time = ? and to_time = ?',
    (room_id, date_time_from, date_time_to)).fetchone()
```

A2: Broken Authentication

Authentication is broken when hackers bypass the login security to gain access to all the privileges owned by the user. For our sign up function, we make sure that the user doesn't already exist. Before adding the new user information we make sure to hash the password. Hashing passwords makes it impossible to decrypt.

For our login function, we check if the entered email address is in our database. If so, the user exists. The entered password will be hashed and compared to the hashed password in the database for that user. Only when both hashed passwords match is the user logged in. We added a session to that user, which gives the user access to the secured pages.

```
50          if error is None:
51              session.clear()
52              session['user_id'] = user['id']
53              current_app.logger.info(user['username'] + ' logged in. IP: ' + str(request.environ['REMOTE_ADDR']))
54              return redirect(url_for('index'))
```

(auth.py)

The session will be unvalid after 60 minutes and the user has to reauthenticate.

```
PERMANENT_SESSION_LIFETIME = timedelta(minutes=60),
```

(__init__.py)

In addition to session timeout, Flask tailsman sets the session cookie to secure and httponly.

Implementation of password strength control:

```
password = PasswordField('Password', [
    validators.DataRequired(message="Password is required"),
    validators.EqualTo('confirm_password', message="Passwords must match"),
    validators.Regexp(
        "^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)[a-zA-Z\d\w\W]{8,}$",
        message="Password must have minimum eight characters, at least one up
    )
])
```

(forms.py)

Other features like multi-factor authentication and recaptcha would make the application even more secure against automated credential stuffing, brute force, and stolen credential reuse attacks.

A3: Sensitive Data Exposure

Sensitive data exposure is when an attacker is able to steal keys, execute man-in-the-middle attacks, or steal clear text data off the server, while in transit, or from the user's client.

To secure from transit attacks we employ a strict TLS (HTTPS) connection, with "Waitress" as the WSGI server.

```
app.run(host='0.0.0.0', port=5000, ssl_context=(cwd+'\cert.pem', cwd+'\key.pem'))
```

To securely store passwords we use a modern password hashing algorithm. The generate_password_hash function (werkzeug.security) utilizes PBKDF2 to generate a salted password hash.

```
db.execute(
    "INSERT INTO user (username, password) VALUES (?, ?)",
    (email, generate_password_hash(password)),
)
db.commit()
return redir
```

```
(function) generate_password_hash: (password: str, method: str =
"pbkdf2:sha256", salt_length: int = 16) -> str
```

(auth.py)

A4: XML External Entities XXE

OWASP defines an XXE attack as:

> An *XML External Entity* attack is a type of attack against an application that parses XML input. This attack occurs when **XML input containing a reference to an external entity is processed by a weakly configured XML parser**. This attack may lead to the disclosure of confidential data, denial of service, server side request forgery, port scanning from the perspective of the machine where the parser is located, and other system impacts.

We do not use XML or XML uploads in our application, so we should be protected from XXE attacks.

A5: Broken Access Control

Access control is broken when users are able to act outside their intentional permissions. It determines restrictions to access the data and resources. It is through authentication and authorization that access control policies are set with limits. Authorization is the process of specifying and enforcing access rights of users to resources.

Authentication:

1. The code snippet specifies that the user who wants to register must use their UiS email.

```
class RegistrationForm(FlaskForm):
    email = StringField('Email', [
        validators.DataRequired(message="Email is required"),
        validators.Email(message="Invalid email, or alredy in use."),
        validators.Regexp('.*(@uis\.no)$', message="Invalid email, or alredy in use.")
    ])
```

(forms.py)

Authorization:

1. The code example shows how a login is required before someone can book a room.

```
17   @bp.route('/', methods=('GET', 'POST'))
18   @login_required
19   def booking():
```
(booking.py)

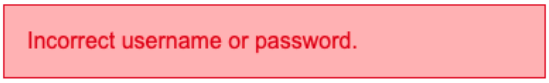
2. All forms include a random CSRF token to prevent an attacker from sending requests on behalf of other users.

```
50        CSRFProtect(app)
```
(__init__.py)

A6: Security Misconfiguration

Security misconfigurations are security controls that are inaccurately configured or left insecure, putting the system and data at risk. In other words what was thought of as a safe environment in the system actually has dangerous gaps or mistakes leaving the data open to risk. Misconfigurations probably happen, because it's a widespread problem and can happen at any level of the application stack. Any poorly documented configuration changes, default settings, or a technical issue across any component in the endpoint could lead to misconfiguration.

Error messages should contain minimal details to avoid sharing useful information to an attacker. For example using the same messages for all outcomes after failed login attempts:



A7: Cross-Site Scripting

Cross-Site Scripting attacks occur when malicious scripts are entered into an application, and are executed on the computer of another user. To prevent XXS attacks, the HTML templates are rendered on the server before being displayed. Jinja will automatically escape all arbitrary HTML tags.

```
return render_template('booking/booking.html', **context)
```

Using quoted variables in HTML attributes to prevent possible javascript handlers.

```
<input readonly id="email" name="email" type="text" value="{{g.user['username']}}">
```

Url building handels escaping of special characters. The generated url paths are absolute, which avoids unexpected behaviour of relative paths in browsers.

```
@bp.route("/booking/confirm/<int:room_id>", methods = ['GET','POST'])
@login_required
def confirm(room_id):
    room_number = get_room(room_id)
```

Input validation with Flask WTF can avoid storing malicious scripts permanently.
Flask Tailsman enables content security policy, which is intended to prevent XXS attacks.

A8: Insecure Deserialization

Deserialization of untrusted data can possibly result in executing unauthorized actions. Flask uses signed cookies to assure that the deserialized session object has not been modified. However, we never store sensitive data in the session, because the data can easily be decoded.

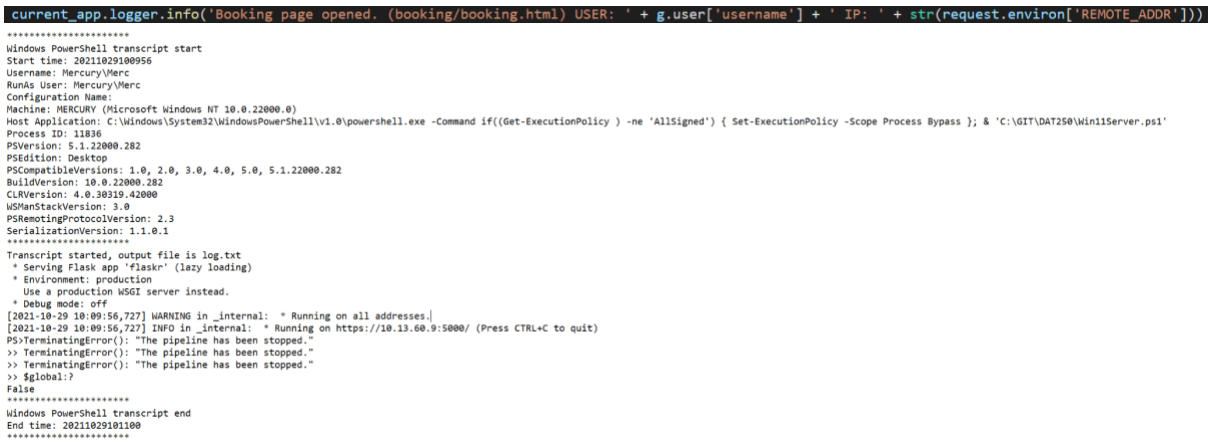
A9: Using Components with Known Vulnerabilities

Components such as libraries and frameworks used within the system almost always execute with full privileges. If a vulnerable component is exploited, it makes the hacker's job easier to cause a serious data loss or server takeover. Virtually every application has these issues because most development teams don't focus on ensuring their components/libraries are up to date. Sometimes, the developers don't even know all the components they are using, never mind their versions. So we should be aware of the components we are using and update or remove flawed ones.

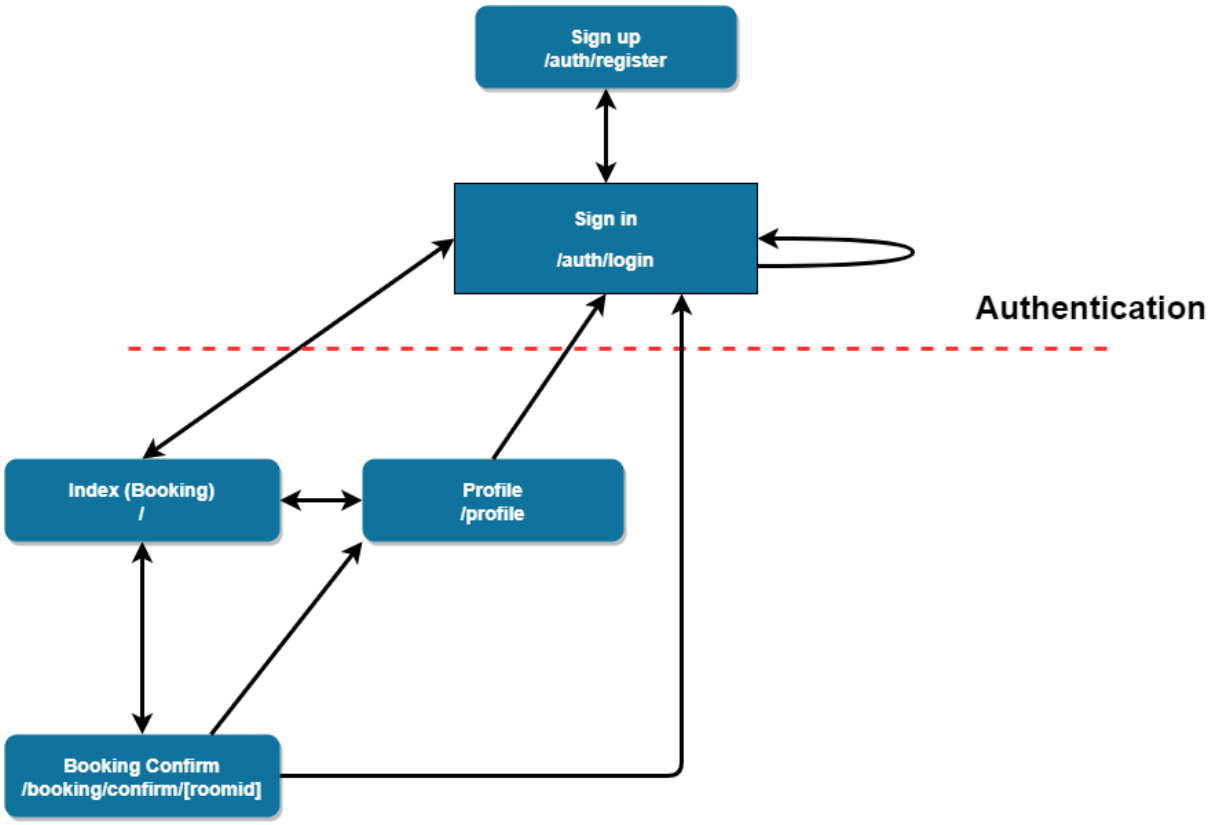
A10: Insufficient Logging & Monitoring

Insufficient logging and monitoring is the most common reason why companies fail to deal with a security breach and problems/downtime effectively.

We are logging important events using the "current_app.logger.info/warning" function, where we write all console output to a file. We include machine info, other environment info, current user and ip.



Site map


Sign up
/auth/register
Sign in
/auth/login
Authentication
Index (Booking)
/
Profile
/profile
Booking Confirm
/booking/confirm/[roomid]

Threat model