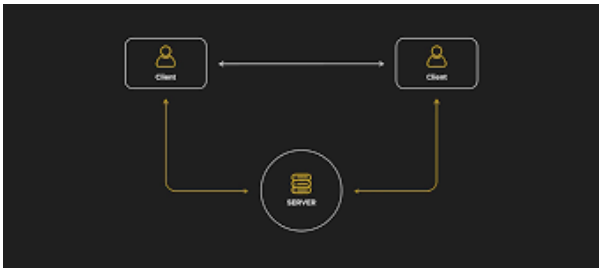
# CODE REVIEW




### What is our GOAL for this CLASS?

In this class, students went through the code of a secure ecommerce app and explored how vulnerabilities were avoided.

### What did we ACHIEVE in the class TODAY?

- Review and explore the code of a secure web app
- Understand how to secure the application in various ways

### Which CONCEPTS/ CODING BLOCKS did we cover today?

- Flask
- Python
- SQLAlchemy
- Cyber Security
- Code Review

## How did we DO the activities?

**Activity:**

1. Refer to the following repository to see the code of the secure ecommerce website.
2. To avoid SQL Injection attack, one should use the SQLALchemy syntax

**Views.py in non-secure version of the code**

```
@views.route('/order')
@cross_origin()
def order():
    try:
        product_id = request.args.get("id")
        if not product_id:
            return jsonify({
                "message": "No product for purchase!",
                "status": "error"
            }), 400
        query = f"select * from products where id={product_id};"
        product = db.engine.execute(query).first()
        address_query = f"select * from address where user_id='{session.get('user_id')}'"
        addresses = db.engine.execute(address_query).all() or []
        return render_template("/order/order.html", product=product, addresses=addresses, user_id=session.get('user_id'))
    except Exception as e:
        return jsonify({
            "message": str(e),
            "status": "error"
        }), 400
```

**Views.py in secure version of the code**

```python
@views.route('/order')
@cross_origin()
def order():
    try:
        product_id = request.args.get("id")
        if not product_id:
            return jsonify({
                "message": "No product for purchase!",
                "status": "error"
            }), 400
        product = Products.query.filter_by(id=product_id).first()
        addresses = Address.query.filter_by(user_id=user.id).all()
        return render_template("/order/order.html", product=product, addresses=addresses, user_id=session.get('user_id'))
    except Exception as e:
        return jsonify({
            "message": str(e),
            "status": "error"
        }), 400
```

3. To understand better, look at **user.py** in **models** folder -

```python
class Users(db.Model):
    __tablename__ = "users"
    id = db.Column(db.Integer, primary_key=True)
    guid = db.Column(db.String, nullable=False, unique=True)
    name = db.Column(db.String(64))
    email = db.Column(db.String(64))
    password = db.Column(db.String(64))
    contact = db.Column(db.String(64))
    addresses = db.relationship(Address, lazy=True, backref="user")
    orders = db.relationship(Orders, lazy=True, backref="user")
    tickets = db.relationship(Tickets, lazy=True, backref="user")
```

4. The schema is defined with -
    a. A class called **Users** that takes **db.Model** as an argument. This means that class **Users** is a SQL model for a table whose tablename would be **users** and the columns with their data types are listed.
    b. Here, **backref="users"** means if someone is querying **address, order or tickets**, they can know which user it belongs to.
5. Since it is a class, it has some of the functions of its own. It also takes on the functions of SQLAlchemy since it's using ***db.Model*** and **query()** is a function of it, used like following -

```python
product = Products.query.filter_by(id=product_id).first()
```

6. We are actually using SQLAlchemy's function ***query*** to query the table ***Products*** with

a **where** condition defined in the ***filter_by()*** function and the ***first()*** means that we want to get just the first value, since we are expecting and needing only one result here.

a. SQLAlchemy itself provides a secure layer where SQL Injection cannot be performed on the database.

7. When we fetch something this way, an object is fetched which means that we can query it's attribute like -
   a. **product.id** instead of **product['id']**
8. Next vulnerability is the IDOR attack that occurs when the URL can be manipulated to fetch unauthorized data.
9. Take a look at the ***login()*** function in ***api.py***

```
if user:
    session["email"] = email
    session["user_id"] = user[0]
    return jsonify({
        "status": "success",
        "id": user[0]
    }), 200
```

10. Here, ***session*** is used to save the ***email*** and ***user_id*** of the user if their credentials are correct.
    a. ***Sessions*** are browser sessions for when a user has logged in. It works like a dictionary and can hold values for each and every user specific to their browser.
    b. This means that these email and user_id values can be retrieved for the user who is accessing the website from the same browser again and again, without them having to log back in again.
11. The **user_id** can hence be accessed directly from the session instead of getting it from the URL in the **profile** page where the IDOR attack was performed.
12. The ***recreate_db()*** and ***seeder()*** are then used in a function called ***rsd()*** which also has a decorator ***@cli.command*** which means that this function can be used from the command line / terminal directly with the following command -
13. Take a look at the ***profile()*** function in ***views.py*** -

***Non secure version -***

```
@views.route('/profile')
@cross_origin()
def profile():
    try:
        user_id = request.args.get("id")
```

***Secure version -***

```
@views.route('/profile')
@cross_origin()
def profile():
    try:
        user_id = session.get('user_id')
```

14. Now, there is Phishing attack where the attacker could upload any extension file like .html or .exe
    a. For it, take a look at the ***submit_help()*** function in ***api.py*** in both ***secure and non-secure*** version -

***Non - Secure version -***

```python
@api.route("/submit-help", methods=["POST"])
def submit_help():
    title = request.form.get("title")
    description = request.form.get("description")
    attachment = request.files.get("attachment")
    if attachment:
        filename = secure_filename(attachment.filename)
        attachment.save(os.path.join(UPLOAD_FOLDER, filename))
    user_email = session.get("email")
    user_query = f"select * from users where email='{user_email}';"
    user = db.engine.execute(user_query).first()
    Tickets.create(user["id"], title, description, filename)
    return jsonify(
            {
                "status": "success",
            }, 201
        )
```



***Secure version -***

```python
@api.route("/submit-help", methods=["POST"])
def submit_help():
    title = request.form.get("title")
    description = request.form.get("description")
    attachment = request.files.get("attachment")
    if attachment:
        filename = secure_filename(attachment.filename)
        extension = filename.split(".")[1]
        if extension.lower() not in [".png", ".jpg", ".jpeg", ".gif"]:
            return jsonify({
                "status": "error",
                "message": "Invalid file!"
            }), 400
        attachment.save(os.path.join(UPLOAD_FOLDER, filename))
    user_email = session.get("email")
    user_query = f"select * from users where email='{user_email}';"
    user = db.engine.execute(user_query).first()
    Tickets.create(user["id"], title, description, filename)
    return jsonify(
            {
                "status": "success",
            }, 201
        )
```

15. Here, a simple check for the extension of the file is added.
16. To conclude, those hackers who find out ways to hack a website and report it to the concerned company or team are known as ***whitehat*** while those who misuse it are known as ***blackhat***. Finding out vulnerabilities and reporting it to the company is known as bug-bounty hunting and it often results in high rewards and even jobs from the companies as their applications are already very secure.

### What's NEXT?

In the next class, we will be reviewing what we have learnt in the networking module and practice some of the important concepts.

### Expand Your Knowledge:

Explore more about cyber security here