Secure Software Design and Engineering
(CY-321)

API Security

Dr. Zubair Ahmad

A Quick Overview of API

waiter in a restaurant—

It takes your order (**request**),
delivers it to the kitchen (server),
and brings back your food
(response)

A Quick Overview of API

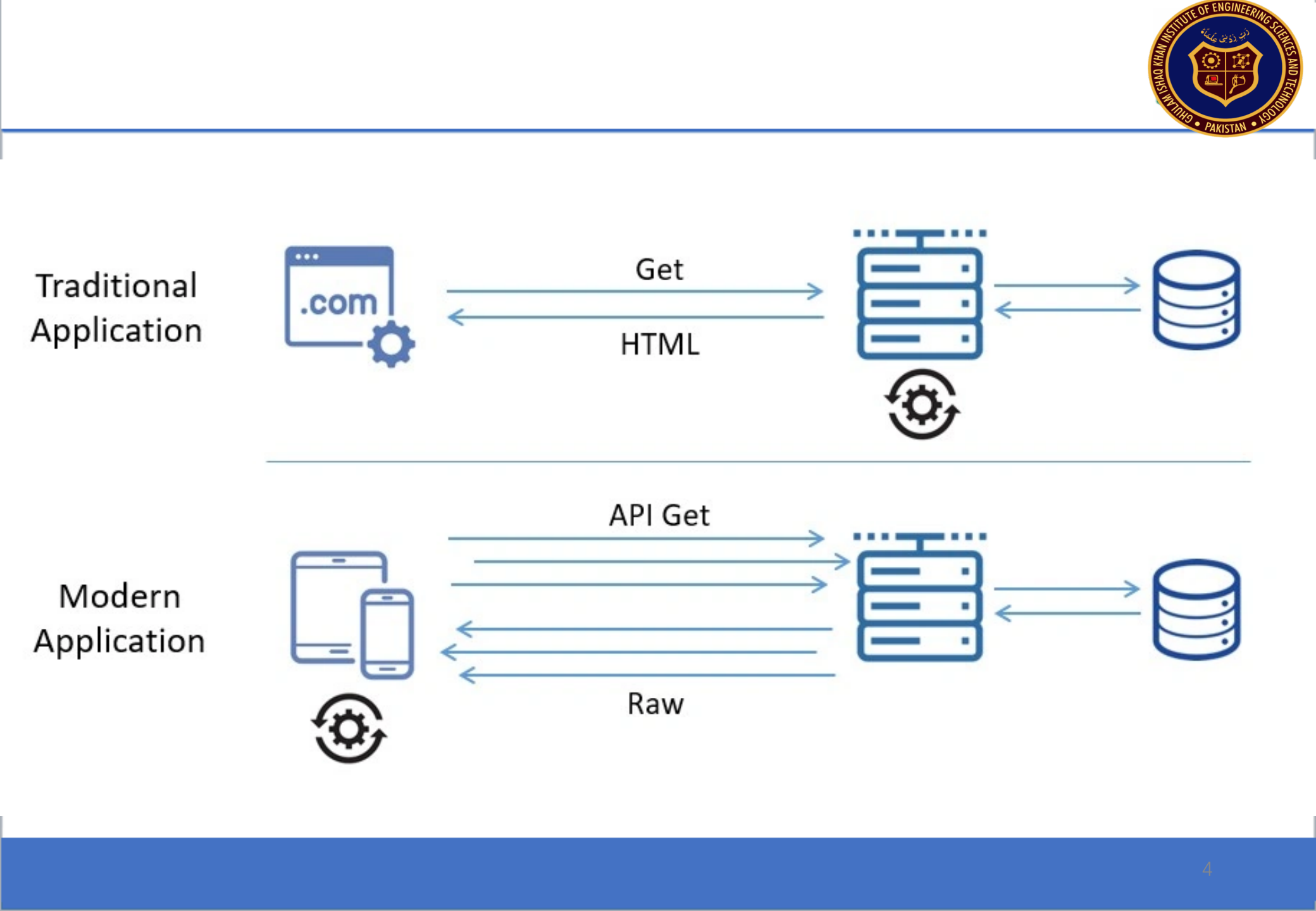
Imagine you are using a food delivery app

You choose a restaurant and place an order.

The app communicates with the restaurant’ s system to confirm your order.

The restaurant prepares your food and updates the app.

Finally, the app tells you that your food is on its way


Traditional Application
.com
Get
HTML
Modern Application
API Get
Raw

Traditional

Architecture: Mono.

Deployment: Runs on a **single server or data center**.**APIs:**

Mostly **internal APIs** (used within the same system).

Scalability: Hard to scale; if one part fails, the whole system is affected.

Example: A legacy banking system where all features (login, transactions, reporting) are in one big software.

Modern

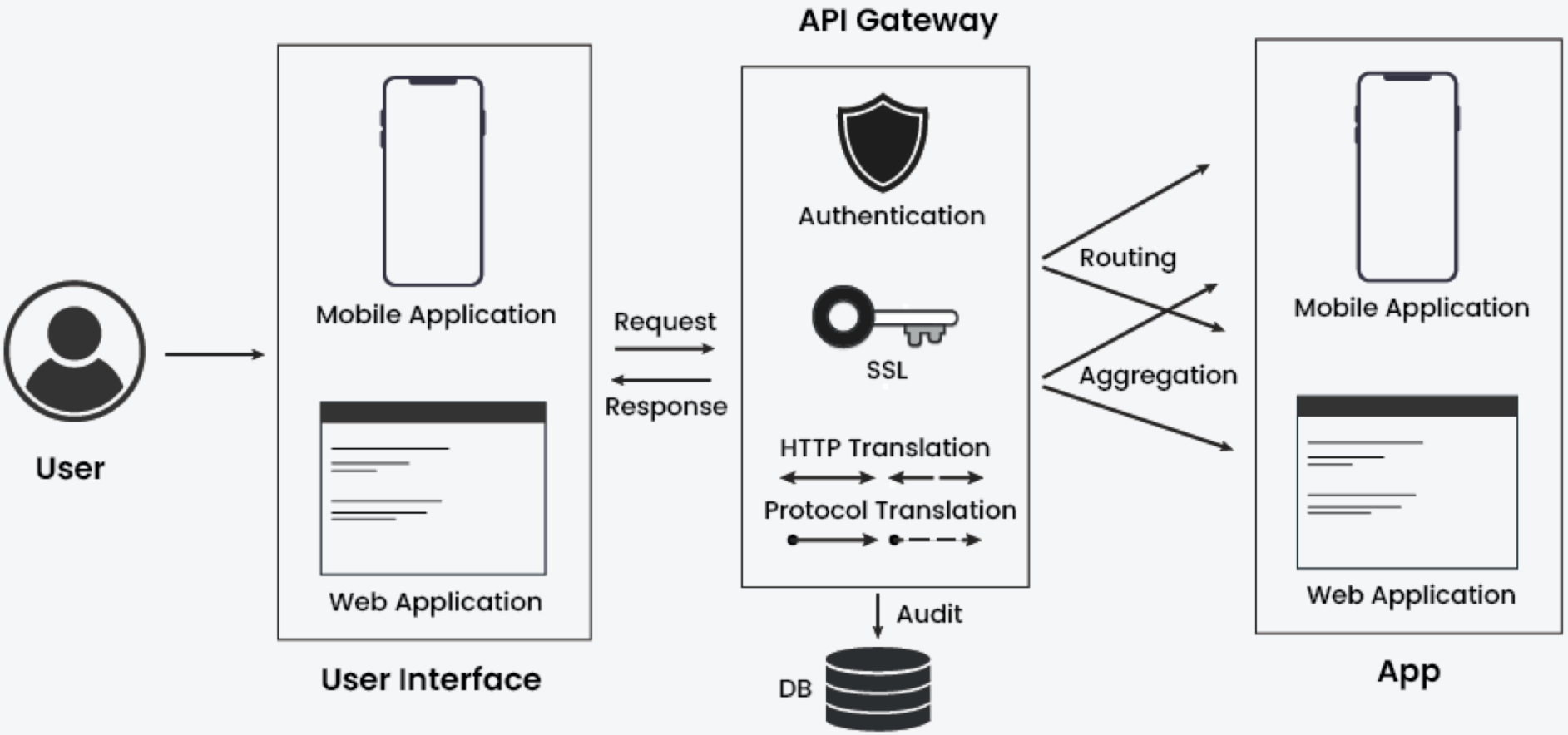
Architecture: Microservices (small, independent services).

Deployment: Runs in the **cloud (AWS, Google Cloud)**.

APIs: Uses **RESTful** to communicate.

Scalability: Easily scalable; one microservice can scale without affecting others

Example: Netflix, Uber, or any cloud-based SaaS applications.


API Gateway
Mobile Application
Web Application
User Interface
User
Request
Response
Authentication
SSL
HTTP Translation
Protocol Translation
Audit
DB
Routing
Aggregation
Mobile Application
Web Application
App

Traditional (Example)

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route("/")
def home():
    return "<h1>Welcome to Traditional Web App</h1>"

if __name__ == "__main__":
    app.run(debug=True)
```

Modern (Example)

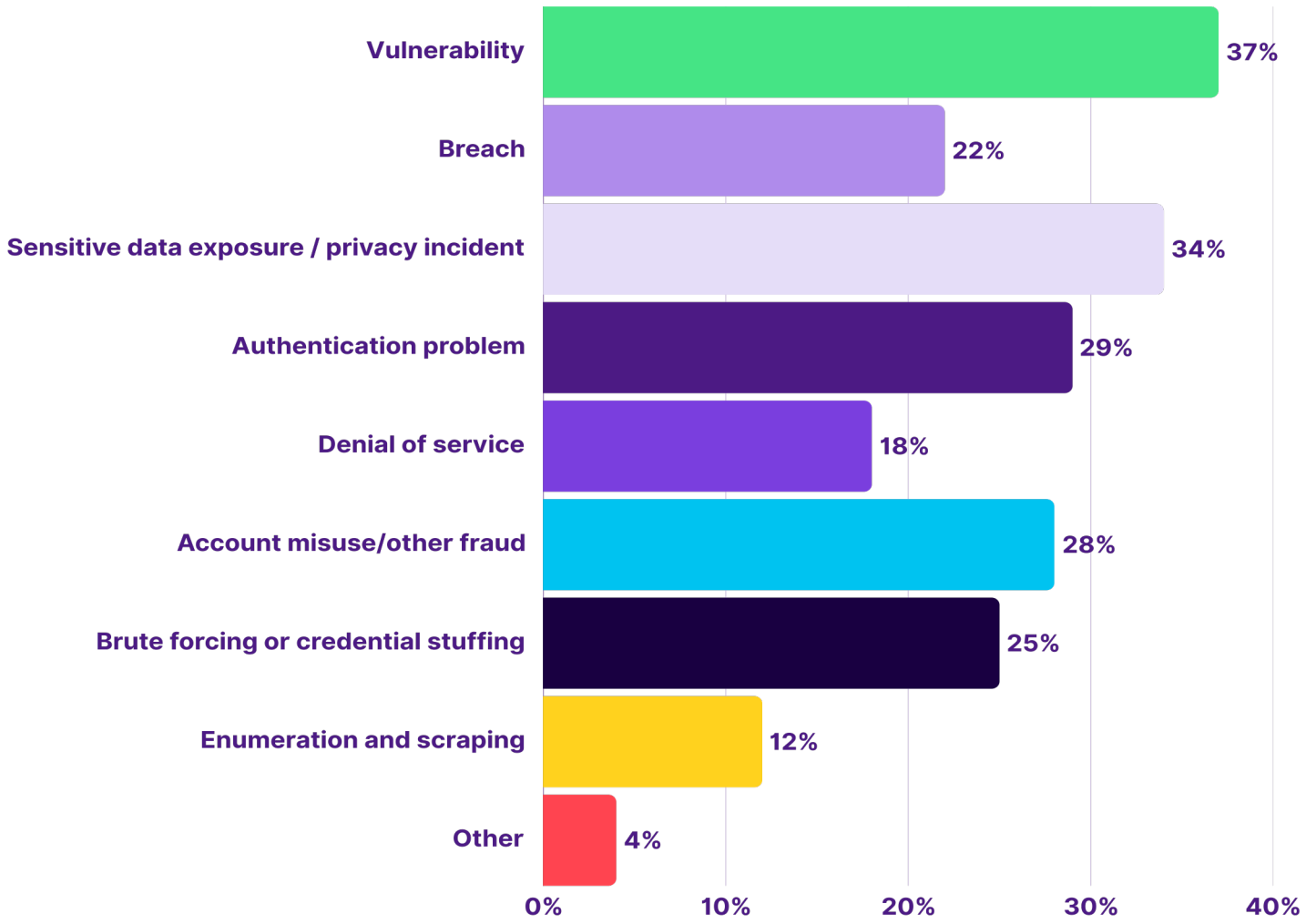
```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route("/api/data", methods=["GET"])
def get_data():
    data = {"message": "Hello from Modern API",
"status": "success"}
    return jsonify(data)

if __name__ == "__main__":
    app.run(debug=True)
```

API Breach Analysis



API Breach Analysis

Rate Limiting

Controls the number of API requests a user or system can make in a given time.

If not implemented correctly, attackers can **flood** the API with excessive requests, causing service disruption or account takeover attempts (brute force attacks)

Tesla API allowed unlimited login attempts without restriction.

API Breach Analysis

Rate Limiting (Insecure)

```
from flask import Flask, request

app = Flask(__name__)

@app.route("/login", methods=["POST"])
def login():
    username = request.form["username"]
    password = request.form["password"]

    # Dummy check (Insecure: Allows unlimited login
attempts)
    if username == "admin" and password ==
"password123":
        return {"message": "Login successful"}
    return {"message": "Invalid credentials"}, 401
```

API Breach Analysis

Rate Limiting (Secure)

```
from flask import Flask, request
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address

app = Flask(__name__)
limiter = Limiter(get_remote_address, app=app,
default_limits=["5 per minute"])

@app.route("/login", methods=["POST"])
@limiter.limit("5 per minute")  # Limit login attempts per
IP
def login():
    username = request.form["username"]
    password = request.form["password"]

    if username == "admin" and password == "password123":
        return {"message": "Login successful"}
    return {"message": "Invalid credentials"}, 401
```

API Breach Analysis

JWT (JSON Web Token) is a compact, URL-safe token format used for authentication and data exchange. It consists of three parts:

1. **Header** – Contains metadata, such as the signing algorithm (e.g., HS256, RS256).
2. **Payload** – Contains claims (user data, roles, expiration, etc.).
3. **Signature** – Ensures integrity by signing the header and payload with a secret key.

API Breach Analysis

Instead of requiring users to share their passwords with third-party apps, OAuth 2.0 lets them grant access via **tokens**. Here's a simplified process:

User Initiates Authorization → A user wants to log in to a third-party app (e.g., a fitness tracker) using their Google account.

App Requests Authorization → The fitness app redirects the user to Google's OAuth authorization server.

User Grants Permission → The user logs in and grants the fitness app access to specific data (e.g., Google Fit).

Authorization Server Issues a Token → Google generates an **access token** and sends it back to the fitness app.

App Uses the Token → The fitness app can now access Google Fit's API securely without needing the user password.

API Breach Analysis

Broken Authorization

APIs should **restrict access** based on user roles and permissions.

If broken authorization exists, unauthorized users can access **data or functions they shouldnt be allowed to.**

Uber’ s API had a vulnerability where a normal user could **change their account type to an admin** by modifying the API request.

API Breach Analysis

Broken Authorization (Insecure)

```
@app.route("/account/<account_id>", methods=["GET"])
def get_account(account_id):
    user = get_logged_in_user()  # Fetch the authenticated user
    account = Account.query.filter_by(id=account_id).first()

    if not account:
        return jsonify({"error": "Account not found"}), 404


    return jsonify({"account_id": account.id, "balance":
account.balance})
```

API Breach Analysis

Broken Authorization (Insecure)

The API assumes the user requesting the account data has access to it but does not check if account_id belongs to user.
An attacker could simply change the account_id in the request to access someone else's bank details.

```
@app.route("/account/<account_id>", methods=["GET"])
def get_account(account_id):
    user = get_logged_in_user()  # Fetch the authenticated
user
    account = Account.query.filter_by(id=account_id).first(

    if not account:
        return jsonify({"error": "Account not found"}), 404


    return jsonify({"account_id": account.id, "balance":
account.balance})
```

API Breach Analysis

Broken Authorization (Secure)

```
@app.route("/account/<account_id>",
methods=["GET"])
def get_account(account_id):
    user = get_logged_in_user()
    account =
Account.query.filter_by(id=account_id,
user_id=user.id).first()  # Authorization check!

    if not account:
        return jsonify({"error": "Unauthorized
access"}), 403

    return jsonify({"account_id": account.id,
"balance": account.balance})
```

API Breach Analysis

Broken Authentication

Broken authentication happens when an API **fails to properly verify user identity**, allowing attackers to **impersonate legitimate users**

Facebook exposed **access tokens** of **50 million users** due to a flaw in its API authentication.

API Breach Analysis

Broken Authentication

```
@app.route("/dashboard", methods=["GET"])
def dashboard():
    token = request.headers.get("Authorization")

    if not token:
        return jsonify({"error": "Unauthorized"}), 401

    try:
        decoded = jwt.decode(token, SECRET_KEY,
algorithms=["HS256"])  # Decoding the JWT
    except jwt.ExpiredSignatureError:
        return jsonify({"error": "Token expired"}), 401
    except jwt.InvalidTokenError:
        return jsonify({"error": "Invalid token"}), 401

    return jsonify({"message": "Welcome to your dashboard"})
```

API Breach Analysis

Broken Authentication

Some developers make a critical mistake by allowing `algorithms=["none"],` which means an attacker can create an unsigned `token (alg: none)` and bypass authentication.

```
decoded = jwt.decode(token, SECRET_KEY, algorithms=["none"])  # Vulnerable code
```

API Breach Analysis

Excess Data Exposure

APIs should only return **necessary data**. If they return **too much information**, attackers can extract sensitive details from responses.

API Breach Analysis

Excess Data Exposure (Insecure)

```
@app.route("/user/<user_id>", methods=["GET"])
def get_user(user_id):
    user =
User.query.filter_by(id=user_id).first()

    if not user:
        return jsonify({"error": "User not
found"}), 404

    # Exposing too much information!
    return jsonify(user.__dict__)
```

API Breach Analysis

Excess Data Exposure (Insecure)

If the User model contains sensitive fields (e.g., password, security questions), this API will expose them in the response.

```
{
  "id": 1,
  "username": "john_doe",
  "email": "john@example.com",
  "password": "hashed_password_here",
  "security_question": "Your first pet?"
}
```

API Breach Analysis

Excess Data Exposure (Secure)

```
@app.route("/user/<user_id>", methods=["GET"])
def get_user(user_id):
    user = User.query.filter_by(id=user_id).first()

    if not user:
        return jsonify({"error": "User not found"}), 404

    return jsonify({
        "id": user.id,
        "username": user.username,
        "email": user.email  # No sensitive data!
    })
```

Avoiding API Abuse

Use Role-Based or Attribute-Based Access Control (RBAC/ABAC)

•Define user roles (admin, user, editor) and limit access to API endpoints based on role.

Enforce User-Specific Data Access

•Ensure that users can only access their own data

Avoiding API Abuse

Use OAuth2 with Scopes

- Implement OAuth2 and define fine-grained permissions.

Use API Gateway or Data Filtering Middleware

- Centralize response filtering to **remove unnecessary fields**.

Questions??

zubair.ahmad@giki.edu.pk

Office: G14 FCSE lobby