Lab3

Subjects	FSCT8561 - Security Applications
Date	@January 27, 2026

Reflection Question

1. Why is hashing required for password storage?
   So the server does not keep your actual password anywhere. It only stores a one way scrambled version (usually with a unique salt per user). When you log in, it scrambles what you typed and compares the results. If the database leaks, attackers do not instantly get real passwords, and turning a good hash back into the original password is extremely hard.
2. How does OTP mitigate replay attacks?
   Because it is time bound and validated against time window, a captured OTP has a short lifespan. The value will not validate once the time window has passed. This prevents an adversarial from reusing a previously obtained code making it much harder
3. What happens if the client and server clocks are not synchronized?
   Because OTP is time based if it is out of sync, they will be computed differenetr time step and it will fail. A lot of websites will add tolerances to handle this by checking adjacent time windows (https://datatracker.ietf.org/doc/html/rfc6238#section-6) however a large shift in time will cause authentication errors
4. What are the limitations of OTP-based authentication?
   It helps a lot with reused or stolen passwords, but it is not a magic shield. Phishing and real-time MITM can still work because an attacker can use the code right away. Also, if you lose the phone/app (or it resets), you can get locked out unless you have backup codes or recovery set up

Security Analysis

Looking at this authentication system, I can see it handles some security concerns but has several gaps that need attention.

The OTP implementation in `auth_server.py` using pyotp does protect against replay attacks. Since TOTP codes are time-based and only valid for a short window, a captured code becomes useless quickly. This also helps when passwords are compromised or reused elsewhere, then attackers would need both the password & the current OTP code to get in.

But there are still several attack vectors. The biggest issue is that everything is transmitted over plaintext `auth_client.py`. Anyone on the network can see your username, password, and OTP in clear JSON. A MITM attacker could intercept and replay the OTP code immediately since they don't need to decrypt it first. Also, the password hashing in `auth_server.py` uses SHA256 with salt, which is better than plaintext but outdated when looking at modern algorithms like bcrypt or argon2.

The thing is, authentication alone doesn't give you full security. This system just checks who you are but doesn't establish a secure session or handle authorization. After logging in, there's no session token, so clients would need to re-authenticate for every request or the server wouldn't know who's who. Also, once authenticated, the data stays vulnerable to the same network sniffing since nothing encrypts it.

For real-world deployment, adding some authentication to data in transit like TLS to encrypt everything in transit, also upgrading to bcrypt for password hashing. We can also generate JWT or session tokens after successful auth. For the account lockout in `auth_server.py` maybe IP-based rate limiting too.