COSC 3371 Cybersecurity - Assignment 3

Authentication and Confidentiality
March 31, 2026

Course	Points	Language	Due Date
COSC 3371 Cybersecurity	100 pts	Python 3.10+ | cryptography library required	April 16, 2026

1. Overview

In this assignment, you will develop a client-server application that employs the MAC-then-Encrypt method (See Slide Page 30 of Chapter 8) to combine authentication and confidentiality using a shared-secret approach. You will complete the provided skeleton files by filling in the TODO placeholders, running the programs, and explaining the observed behavior.

2. Learning Objectives

- Explain how a MAC supports message authentication and integrity.
- Describe how encryption ensures confidentiality in the MAC-then-Encrypt workflow.
- Develop sender and receiver logic based on partially completed skeleton code.
- Demonstrate one successful case and one failure case.
- Interpret the verification results and explain why a message is accepted or rejected.

3. Prerequisites

Before starting, ensure the following are in place:

Software	Knowledge
Python 3.10+ (install package with: pip install cryptography)	Basic Python programming, strings/bytes, functions, and exception handling
PyCharm or VS Code IDE	Basic understanding of hashing, symmetric encryption, MAC, and public/private keys
The provided skeleton files: client.py and server.py	How authentication and confidentiality are achieved in a shared-secret design

4. Background

MAC-then-Encrypt

For a plaintext message M, the sender first computes a MAC tag using key K_1, then appends the tag to the plaintext, and finally encrypts the combined data with key K_2. The receiver first decrypts the message, separates the plaintext from the received tag, recomputes the MAC, and compares the two tags. If the tags match, the message is authenticated; otherwise, it is rejected.

Formula: tag = HMAC(K_1, M); payload = M || tag; C = E(K_2, payload)

5. Files Provided

File	Purpose
Assignment3.docx	Assignment description
client.py	Client skeleton (sender): MAC-then-Encrypt with TODO blanks for shared keys, crypto helpers, sender logic, and two test cases
server.py	Server skeleton (receiver): TODO blanks for shared keys, AES-CBC decryption, MAC verification, and TCP receive loop.

6. Tasks

MAC-then-Encrypt (100 pts)

1. Open client.py and server.py and complete every TODO blank in both files.
2. Set the keys and IV precisely as specified by the skeleton format: a 32-byte HMAC key, a 16-byte AES key, and a 16-byte IV.
3. In `client.py`, complete the helper functions for MAC calculation, PKCS7 padding, and AES-CBC encryption. In server.py, finish the helper functions for MAC verification, PKCS7 unpadding, and AES-CBC decryption.
4. In `client.py`, complete the `sender()` function so that it converts the message to bytes, computes the MAC, appends the tag, encrypts the payload, prints the required output, and transmits the ciphertext to the server over TCP.
5. In `server.py`, complete the `receiver()` function, so it receives the ciphertext over TCP, decrypts the payload, splits the plaintext and tag, recomputes the MAC, and compares the tags securely using `hmac.compare_digest()`.
6. Start `server.py`, then run client.py with Case 1 (valid message: “Transfer $100 to Bob”). Show that the server output reports: Tag matched: message accepted.
7. Start `server.py`, then run client.py with Case 2 (tampered message: “Transfer $1000 to Bob” with a stale MAC tag). Show that the server output reports "Tag not matched: message rejected."
8. Provide a brief explanation of how confidentiality is maintained and why authentication fails when tampering occurs.

Program behavior to demonstrate

Case / Scenario	What you do	Expected result
Case 1	Run the code with the original message	Tag matched: message accepted
Case 2	Run the code with the changed plaintext	Tag not matched: message rejected

7. Deliverables (100 pts total)

Deliverable	Points
Completed client.py and server.py	50
Screenshot/output for Case 1 and Case 2	30
Short explanation of authentication and confidentiality	20
Total	100

8. Hints & Quick Reference

- Use HMAC-SHA256 exactly as indicated in the skeleton files.
- Do not implement AES, HMAC, or SHA-256 from scratch. Use the cryptography library.
- Uses different keys: K1 for MAC and K2 for encryption. Do not reuse a key for both functions..

- The HMAC-SHA256 tag length is 32 bytes, which is why the receiver extracts the last 32 bytes from the decrypted payload.
- When testing Case 2, altering even one byte should trigger verification failure or a decryption/padding error, both of which must result in rejection..
- Always start the server before the client. Keep your terminal output clear: label the client side, server side, and each case or scenario distinctly so the TA can easily follow your results.

9. Submission

Submit via the Canvas course portal one ZIP file named:

assignment3_<YourLastName_YourID>.zip

Your ZIP file must include:

- the completed client.py and server.py files
- one PDF report containing your screenshots, output, and short answers

Academic Integrity: All code must be your own. You may discuss concepts, but you must not share or copy code. Violations will result in a grade of zero.

Good luck - and remember, every secure connection you make on the web uses concepts you are implementing here!