# Homework 2 (100 pts)

**Rule: Finish all of the following on your own. Submit your solution in PDF format.**

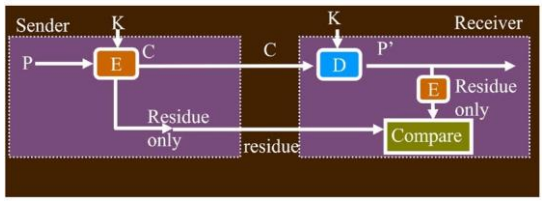
1. **What are the properties of hash functions?**
   Hash functions have four main properties:
   - Performance / Easy to compute: It is fast and efficient to compute the hash value H(m) for any message m.
   - One-way property: Given H(m), it is computationally infeasible to find the original message m.
   - Weak collision resistance: Given a message m and its hash H(m), it is computationally infeasible to find a different message m' such that H(m') = H(m).
   - Strong collision resistance: It is computationally infeasible to find any two distinct messages $m_1$ and $m_2$ such that $H(m_1) = H(m_2)$.

2. **Explain meet-in-the-middle attacks against double-DES.**
   Double-DES, as the name suggests, applies DES twice: $C = E_{K2}(E_{K1}(P))$, using two different 56-bit keys (total 112-bit key space). This equates to approximately $2^{112}$ security. However, meet-in-the-middle attacks reduces it to roughly $2^{57}$ time and $2^{56}$ space.

   Here's how it works, as per the plaintext scenario in the slides:
   - For a known plaintext-ciphertext pair (P, C), encrypt P with all possible $2^{56}$ values of K1 and store the intermediate results ($E_{K1}(P)$) in a table (sorted or hashed for fast lookup). The cost is $2^{56}$ encryptions and $2^{56}$ storage.
   - Then, you would decrypt C with all possible $2^{56}$ values of K2 to get intermediate values ($D_{K2}(C)$), and check if any match an entry in the table.
   - A match gives a candidate pair (K1, K2) where $E_{K2}(E_{K1}(P)) = C$. If there is a match, you'll want to verify with additional pairs if needed, but there are generally very few false positives.
   - Overall, the total effort required comes out to about $2 \times 2^{56}$ which equals $2^{57}$ operations plus table lookup, far less than brute-forcing $2^{112}$.

   Hence, this attack makes double-DES only marginally stronger than single DES (approx. $2^{56}$ security), which is why triple-DES (approx. $2^{112}$ security against MITM) is the stronger option.

3. **Explain why the following attempt cannot ensure integrity.**




   Here we see an attempt to add integrity protection by encrypting a message using a shared key K. This is known as an Encrypt-and-MAC or MAC-then-Encrypt construction. This scheme fails to ensure integrity because the residue/MAC is computed only on the plaintext (before encryption or after decryption), not on the transmitted ciphertext C. An attacker who intercepts C could possibly modify the ciphertext without knowing K. The receiver will still decrypt the tampered C to some P' (likely coming out as garbage, but

possibly meaningful text depending on the modifications), compute the residue on P', and if the attack crafts a valid-looking P' with matching residue, the comparison passes, even though the message was altered. Encryption alone provides no integrity as there is no way to detect changes, and protecting only the plaintext allows ciphertext tampering to go undetected until after decryption. Secure authenticated encryption requires computing the MAC on the ciphertext (Encrypt-then-MAC) to reject forged/modified ciphertexts immediately, before decryption, preventing these attacks. The pictured approach lacks that protection, violating integrity.

4. **Alice designs a new double-DES scheme. The scheme will first DES-encrypt a message using K1 to get an intermediate ciphertext, then DES-decrypt the intermediate ciphertext using K2 to get the final ciphertext. Is there any vulnerability in Alice's design?**

   Yes, Alice's design ($C = D_{K2}(E_{K1}(M))$) is vulnerable to the meet-in-the-middle attack, similar to standard double-DES ($E_{K2}(E_{K1}(M))$). An attacker with known plaintext-ciphertext pairs can encrypt M with all $2^{56}$ possible K1 values and store the intermediates. Then they can simply decrypt C with all $2^{56}$ possible K2 values to find matches, reducing the effective security to about $2^{57}$ operations instead of $2^{112}$. This makes it only marginally stronger than single DES.

5. **Suppose the sub-key generation function is to reverse all the bits of the key K (e.g., 0 1 1 1 ➔ 1 1 1 0), and the scrambling function is *f* = M XOR K', where M is the second half of input bits and K' is the sub-key (i.e., the reverse of the original key K). Now given the original K = 0011, and input bits 1111 0000, compute the output of the Feistel Cipher.**

   (K' = reverse = 1100).

   f = R0 XOR K' = 0000 XOR 1100 = 1100.

   New right half: R1 = L0 XOR f = 1111 XOR 1100 = 0011.

   New left half: L1 = R0 = 0000.

   Output: 00000011.

6. **A and B want to ensure the integrity and authenticity of the messages between them, but they do NOT care about the confidentiality. Assume A and B share a key K.**

   **Answering two questions**

   a. **How can they achieve their goal only with symmetric key cryptography?**
   A and B can achieve integrity and authenticity using a Message Authentication Code (MAC) with symmetric cryptography, such as CBC-MAC which computes the MAC by encrypting the message M in CBC mode with key K, taking the last block as the tag, and sends M || tag. B recomputes the MAC on received M and verifies if it matches the received tag. This ensures the message hasn't been tampered with and comes from someone knowing K.

   b. **How can they achieve their goal with hash function H?**
   A and B can use a keyed hash like HMAC: A computes tag = H(K XOR opad || H(K XOR ipad || M)) and sends M || tag. B recomputes the tag on received M and checks for a match. This provides integrity as collisions are infeasible and authenticity via shared K, without needing encryption.

7. **Bob is assigned a task to design a way to allow encryption of files stored in the system: all files are stored in an encrypted form. If a block of a file is requested, the system should retrieve the block, decrypt it and return the plaintext to the host. Similarly, if a host writes a block to the storage system, the system should retrieve the right keying material, encrypt the block, and only save the ciphertext on disk. Consider the modes of operations discussed in class (i.e., ECB, CBC, CFB, CTR). Which one should be used in terms of read/write efficiency? Why? (You do NOT need to consider the key storage problem.)**

   CTR mode should be used for read/write efficiency in file storage encryption. It allows independent encryption/decryption of any block by using a counter value per block (e.g., IV + block index), allowing for random access without the need to decrypt previous blocks. In contrast, CBC and CFB require chaining from prior blocks for decryption, making random reads inefficient, while ECB is independent but insecure due to pattern leakage.

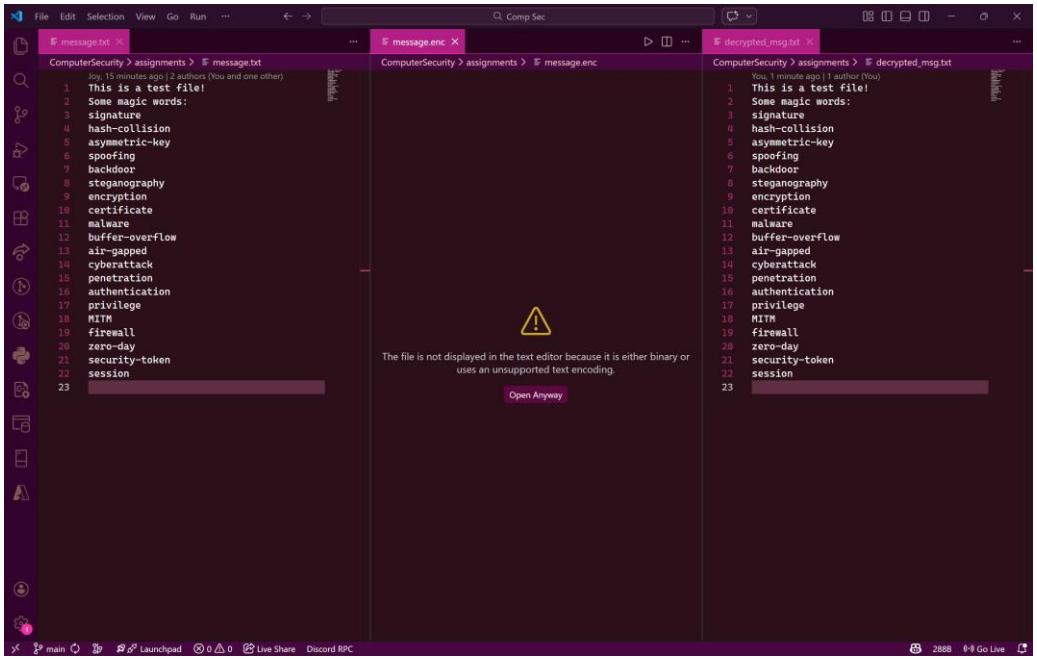
8. **<u>Lab Task: Exploring AES-CBC Encryption Using OpenSSL</u>**

   **In this lab, you will explore encryption using OpenSSL. First, ensure you have OpenSSL installed. If you are using Linux or macOS, OpenSSL is usually pre-installed; you can check by running openssl version. If it is not installed, use sudo apt install openssl (Ubuntu) or brew install openssl (macOS). On Windows, download OpenSSL from https://slproweb.com/products/Win32OpenSSL.html (Or using WSL on Windows) and follow the installation instructions. Once installed, verify by running openssl version in the terminal or command prompt.**

   **<u>Tasks:</u>**

   1. **Basic Encryption and Decryption**
      - **Use OpenSSL to encrypt a text file (message.txt) with AES-256-CBC encryption using a password.**
      - **Decrypt the file and verify the content.**
      - **Provide a screenshot of the commands and output.**

```
code-to-joy@JM-SurfacePro11:~/CompSec/ComputerSecurity/assignments$ open
ssl enc -aes-256-cbc -pbkdf2 -in message.txt -out message.enc
enter AES-256-CBC encryption password:
Verifying - enter AES-256-CBC encryption password:
code-to-joy@JM-SurfacePro11:~/CompSec/ComputerSecurity/assignments$ ls
Assignment1.pdf  message.enc  message.txt
code-to-joy@JM-SurfacePro11:~/CompSec/ComputerSecurity/assignments$ open
ssl enc -aes-256-cbc -pbkdf2 -d -in message.enc -out decrypted_msg.txt
enter AES-256-CBC decryption password:
code-to-joy@JM-SurfacePro11:~/CompSec/ComputerSecurity/assignments$ ls
Assignment1.pdf  decrypted_msg.txt  message.enc  message.txt
code-to-joy@JM-SurfacePro11:~/CompSec/ComputerSecurity/assignments$
```




2. **Encrypt and Decrypt Using a Key File**
   - **Generate a random encryption key using OpenSSL and save it to a file.**
   - **Use this key to encrypt message.txt and decrypt it back.**
   - **Show the encryption key, encrypted file, and decrypted file in a screenshot.**

```
code-to-joy@JM-SurfacePro11:~/CompSec/ComputerSecurity/assignments/Assignment2$ xxd -p -c 32 enckey.key
0b7cc37f597faf984e32ab419efd65e19ac05d9e50fb0532da403ac27f375e55
code-to-joy@JM-SurfacePro11:~/CompSec/ComputerSecurity/assignments/Assignment2$ xxd -l 64 message2.enc
00000000: b437 75b9 447b eb25 3716 46d0 8046 d2f0  .7u.D{.%7.F..F..
00000010: 2ffe 747d eaa5 92a6 5232 19e8 c063 0e19  /.t}....R2...c..
00000020: e72c 1a8e 2931 1aa7 9727 6446 4238 f552  .,..)1...'dFB8.R
00000030: e289 1c71 640b 9527 001b 2bdd d90d 5ea3  ...qd..'..+...^.
code-to-joy@JM-SurfacePro11:~/CompSec/ComputerSecurity/assignments/Assignment2$ cat decrypted_message2.txt
This is a test file!
Some magic words:
signature
hash-collision
asymmetric-key
spoofing
backdoor
steganography
encryption
certificate
malware
buffer-overflow
air-gapped
cyberattack
penetration
authentication
privilege
MITM
firewall
zero-day
security-token
code-to-joy@JM-SurfacePro11:~/CompSec/ComputerSecurity/assignments/Assignment2$ ls
decrypted_message2.txt  decrypted_msg.txt  enckey.key  iv.bin  message2.enc  message_decrypted.txt  message.enc  message.txt
```

3. **Observing Encrypted Data**
   - **Encrypt message.txt using AES-256-CBC and then open the encrypted file in a text editor.**
   - **Compare the original and encrypted content, then describe your observations.**

The original message.txt file contains clear, readable plaintext with different words or phrases on 22 rows. This structure is human-readable with reasonable formatting, line breaks, and recognizable English words and grammar, making the text immediately interpretable without any tools.

On the other hand, the encrypted message.enc file, when opened in a text editor, appears as complete binary nonsense with random bytes, starting with the "Salted__" header (which indicates password-based key derivation with salt) followed by uncomprehensible combinations of letters, numbers, symbols and accented characters. No words, patterns, or structure are visible to me and the original plaintext is fully obscured and indistinguishable as is.

This contrast highlights AES-256-CBC's strength in providing computational security through diffusion and confusion in CBC (cipher block chaining) mode. With it, no frequency patterns or leaks appear, unlike earlier, weaker, ciphers. The encrypted output file visually confirms that modern encryption such as this hides information completely from unauthorized viewers while allowing perfect recovery upon correct decryption.