KKI Short Report

SHA-256 & HMAC

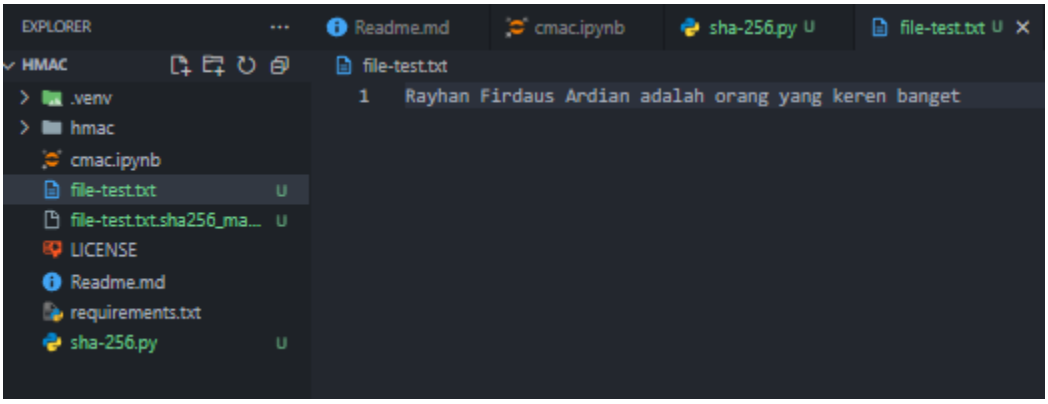
Kelompok 9
1. David Neilleen Irvinne (23/519095/PA/22219)
2. Rayhan Firdaus Ardian (23/519095/PA/22279)
3. Nugroho Adi Susanto (23/520312/PA/22367)
4. Muhammad Dhiwaul Akbar (23/523237/PA/22513)

Problem 1 : Hash Function Integrity Check

1. Mengapa SHA-256?
    a. Input yang sama, output hash akan sama, jadi ketika diubah maka hash digest akan berbeda (karakteristik hash)
    b. One way function and irreversible = cocok untuk hanya mengecek integritas saja
    c. SHA-256 lebih resist terhadap kolisi (Collision Resistance) → Sangat sulit secara komputasi untuk menemukan dua input data yang berbeda yang menghasilkan output hash SHA-256 yang sama. Dimana ini sangat penting karena apabila penyerang bisa dengan mudah membuat file berbahaya yang memiliki hash yang sama dengan file asli yang sah, maka pemeriksaan integritas akan gagal mendeteksi penggantian file tersebut
    d. Pre-image resistance → Walaupun memiliki output has SHA-265, sulit untuk menemukan input data asli yang menghasilkan hash tersebut

2. Apa yang terjadi jika menggunakan weaker hash func
    a. Collision Resistance yang buruk (Resiko lebih tinggi)
    b. Ukuran hash yang lebih kecil daripada SHA-256
    c. Pre-image resistance yang lebih rendah

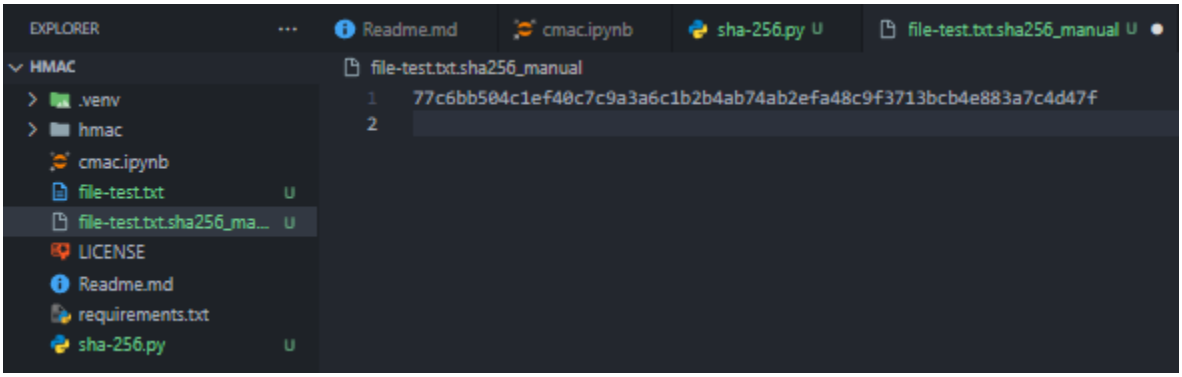
3. Integrity Verification Check :
    - File-test.txt :



- Script python dijalankan :

```
dofala@PC MINGW64 /d/Project/hmac (main)
$ python sha-256.py store file-test.txt
Hash SHA-256 (manual) untuk 'file-test.txt' telah dihitung dan disimpan ke 'file-test.txt.sha256_manual'
Hash: 77c6bb504c1ef40c7c9a3a6c1b2b4ab74ab2efa48c9f3713bcb4e883a7c4d47f
```

- Muncul file : file-test.txt.sha256_manual yang berisi hash :



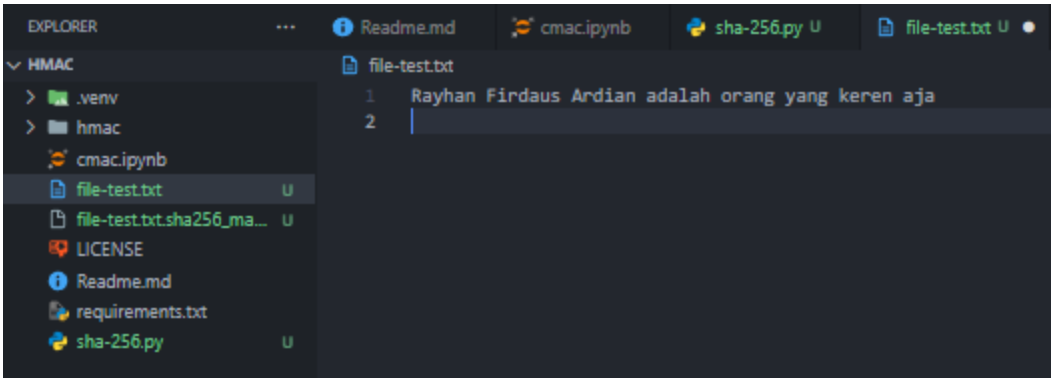
- Verify :

```
dofala@PC MINGW64 /d/Project/hmac (main)
$ python sha-256.py verify file-test.txt

Memverifikasi integritas file: 'file-test.txt'
Hash Tersimpan : 77c6bb504c1ef40c7c9a3a6c1b2b4ab74ab2efa48c9f3713bcb4e883a7c4d47f
Hash Saat Ini  : 77c6bb504c1ef40c7c9a3a6c1b2b4ab74ab2efa48c9f3713bcb4e883a7c4d47f
hasil: Integritas file aman. Tidak ada perubahan terdeteksi.
```

Disini terdapat pengecekan apakah file-test.txt masih sama atau tidak (ada perubahan atau tidak)

- Terjadi Perubahan, misal file-test.txt diganti (banget → aja) :



- Jalankan script verify :

```
dofala@PC MINGW64 /d/Project/hmac (main)
$ python sha-256.py verify file-test.txt

Memverifikasi integritas file: 'file-test.txt'
Hash Tersimpan : 77c6bb504c1ef40c7c9a3a6c1b2b4ab74ab2efa48c9f3713bcb4e883a7c4d47f
Hash Saat Ini   : 9cf6a88ef8e936c3bb0951b7c9cabfaef362738b28a4ad997d2ef4876cffd663
hasil: Integritas file rusak. File telah dimodifikasi!
```

Problem 2 : Message Authentication using HMAC

1. Penerimaan Pesan dan Secret Key di Awal Program (Sisi Pengirim)

Pada awal program, di dalam fungsi `main` pada file `hmac.c`, pesan asli dan kunci rahasia (secret key) didefinisikan secara langsung dalam kode:

```
uint8_t secret_key[] = "kunciRahasiaSuperAman123";
uint8_t original_message[] = "Ini adalah pesan rahasia.";
```

`secret_key`: Ini adalah kunci simetris yang diasumsikan telah dibagikan secara aman sebelumnya antara pengirim dan penerima yang sah. Kunci ini krusial untuk keamanan HMAC. Dalam contoh ini, kunci berupa string literal.
`original_message`: Ini adalah data atau pesan yang ingin dikirim oleh pengirim dan perlu dilindungi integritas serta otentisitasnya.
Panjang dari `secret_key` dan `original_message` kemudian dihitung menggunakan `strlen` untuk diteruskan ke fungsi HMAC.

2. Pembuatan HMAC dari Pesan Menggunakan Secret Key (Sisi Pengirim)

Pengirim menggunakan fungsi `hmac_sha256` untuk menghasilkan Message Authentication Code (MAC) dari `original_message` menggunakan `secret_key`. Proses di dalam fungsi `hmac_sha256` (di `hmac.c`) adalah sebagai berikut, sesuai dengan standar HMAC:

A. **Pemrosesan Kunci (`K_plus`)**:

a. Jika panjang `secret_key` lebih besar dari ukuran blok SHA-256 (`SHA256_BLOCK_SIZE_BYTES`, yaitu 64 byte), maka `secret_key` akan di-hash terlebih dahulu menggunakan SHA-256. Hasil hash (32 byte untuk SHA-256) ini kemudian akan di-padding dengan bit nol di sebelah kanan hingga mencapai 64 byte. Kunci yang diproses ini disebut `K_plus`.
b. Jika panjang `secret_key` kurang dari atau sama dengan 64 byte, maka `secret_key` akan di-padding dengan bit nol di sebelah kanan hingga panjangnya menjadi 64 byte, membentuk `K_plus`.

B. **Pembentukan `i_key_pad` dan `o_key_pad`**:
   a. `i_key_pad` (inner pad key) dibuat dengan melakukan operasi XOR antara `K_plus` dengan `ipad` (konstanta `0x36` yang diulang sebanyak ukuran blok).
   b. `o_key_pad` (outer pad key) dibuat dengan melakukan operasi XOR antara `K_plus` dengan `opad` (konstanta `0x5C` yang diulang sebanyak ukuran blok).
C. **Perhitungan Hash Dalam (Inner Hash)**:
   a. Fungsi hash SHA-256 (`sha256_init`, `sha256_update`, `sha256_final` dari modul `sha256.c`) diterapkan pada gabungan (concatenation) dari `i_key_pad` dan `original_message`.
   b. Hasil dari langkah ini adalah `inner_hash_result` (sebuah digest SHA-256 sepanjang 32 byte).
D. **Perhitungan Hash Luar (Outer Hash)**:
   a. Fungsi hash SHA-256 diterapkan pada gabungan dari `o_key_pad` dan `inner_hash_result`.
   b. Hasil dari langkah ini adalah **HMAC digest akhir** (`mac_digest`).

Kode di `hmac.c` yang melakukan ini:

```
7  void print_hex(char* label, uint8_t *data, int len) {
8      printf("%s: ", label);
9      for (int i = 0; i < len; ++i) {
10         printf("%02x", data[i]);
11     }
12     printf("\n");
13 }
14
15 void hmac_sha256(uint8_t *key, size_t key_len, uint8_t *message, size_t message_len, uint8_t *mac_digest) {
16
```

Pada titik ini, `hmac_digest` berisi MAC dari pesan asli. Pengirim kemudian akan mengirimkan `original_message` bersama dengan `hmac_digest` ini.

3. Pengiriman Pesan ke Receiver dan Intersepsi oleh MITM

Dalam simulasi:

- Pesan (`original_message`) dan HMAC-nya (`hmac_digest`) diasumsikan dikirim melalui saluran komunikasi.
- Seorang **Man-in-the-Middle (MITM)** berhasil mencegat transmisi ini.

- MITM **tidak mengetahui `secret_key`** yang digunakan untuk membuat HMAC.
- MITM kemudian **mengubah isi pesan**. Dalam kode `hmac.c`:

```
71      char modification[] = "{{Pesan ini diubah oleh MITM}}";
72      uint8_t intercepted_message[256];
73      strcpy((char*)intercepted_message, (const char*)original_message);
74      strcat((char*)intercepted_message, modification);
75      printf("Pesan diubah menjadi: %s\n", intercepted_message);
76
```

Karena MITM tidak memiliki `secret_key`, ia **tidak dapat menghasilkan HMAC yang valid** untuk pesan yang telah diubah (`intercepted_message`). Sebagai gantinya, untuk mencoba mengelabui penerima, MITM mungkin melakukan salah satu hal berikut:

- Mengirim pesan yang diubah dengan HMAC asli dari pesan original (ini akan dideteksi).
- Mengirim pesan yang diubah dengan *tidak ada* HMAC (ini akan langsung mencurigakan).
- Mencoba membuat "digest" baru dari pesan yang diubah tanpa kunci. Dalam simulasi kode, MITM membuat hash SHA-256 biasa (bukan HMAC) dari `intercepted_message`:

```
77      uint8_t mitm_digest[SHA256_DIGEST_LENGTH];
78      SHA256_CTX mitm_sha_ctx;
79      sha256_init(&mitm_sha_ctx);
80      sha256_update(&mitm_sha_ctx, intercepted_message, strlen((char*)intercepted_message));
81      sha256_final(&mitm_sha_ctx, mitm_digest);
82      print_hex("Digest dari MITM", mitm_digest, SHA256_DIGEST_LENGTH);
```

MITM kemudian mengirimkan `intercepted_message` (pesan yang telah diubah) beserta `mitm_digest` (digest palsu/tidak otentik) kepada penerima.

4. Bagaimana Receiver Memverifikasi Keaslian Pesan yang Diterima

Penerima menerima pesan (yang dalam kasus ini adalah `intercepted_message`) dan digest yang menyertainya (`mitm_digest`). Untuk memverifikasi keaslian dan integritas pesan, penerima melakukan langkah-langkah berikut:

1. Penerima **sudah memiliki `secret_key` yang sama** dengan pengirim (karena ini adalah kunci simetris yang telah dibagikan sebelumnya).
2. Penerima mengambil pesan yang diterima (`intercepted_message`).
3. Penerima **menghitung ulang HMAC** dari `intercepted_message` menggunakan `secret_key` yang ia miliki dan fungsi `hmac_sha256` yang sama persis seperti yang digunakan oleh pengirim.

```c
    uint8_t calculated_hmac_on_received[SHA256_DIGEST_LENGTH];
    hmac_sha256(secret_key, strlen((char*)secret_key), intercepted_message,
                            strlen((char*)intercepted_message), calculated_hmac_on_received);
    print_hex("HMAC yang dihitung penerima", calculated_hmac_on_received, SHA256_DIGEST_LENGTH);
```

4. Penerima kemudian **membandingkan** HMAC yang baru dihitung (`calculated_hmac_on_received`) dengan digest yang diterima bersama pesan (`mitm_digest`).
    i. **Jika kedua nilai HMAC ini identik**, maka pesan dianggap otentik (berasal dari pengirim yang benar) dan tidak diubah selama transmisi.
    ii. **Jika kedua nilai HMAC ini berbeda**, maka pesan dianggap telah diubah atau bukan berasal dari pengirim yang sah (atau keduanya).
    iii. Contoh Simulasi:

```
david@swift:~/Documents/matkul/sem4/kripto/hmac/CMAC-Cryptography/hmac$ gcc hmac.c sha256.c -o hmac
david@swift:~/Documents/matkul/sem4/kripto/hmac/CMAC-Cryptography/hmac$ ./hmac

----- SISI PENGIRIM -----
Pesan Original: Ini adalah pesan rahasia.
HMAC yang dihasilkan: 708ee96094abd7a18593d49a1dc00a670e9ced2351cb50c2c295349c285b3f77


----- INTERSEPSI & SERANGAN MITM (tanpa kunci) -----
Pesan diubah menjadi: Ini adalah pesan rahasia.{{Pesan ini diubah oleh MITM}}
Digest dari MITM: 3da71acfa97607d19c6683f221948c53cfcd6107dfe14e5ed3b63a34e7ad5ced


----- SISI PENERIMA -----
HMAC yang dihitung penerima: ac26d8e010efb39828b501287c741d7aa889f9af239f3a4996643db1ce2463b4

VERIFIKASI: Pesan terindikasi telah dipalsukan karena digest yang berbeda.
```