Implementasi Secure Local Storage dengan AES-256 dan PBKDF2 untuk Aplikasi Desktop

Andi Marihot Sitorus, 13522138
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13522138@mahasiswa.itb.ac.id, andisitorusman@gmail.com

***Abstract*—Aplikasi desktop sering menyimpan data sensitif seperti kredensial database, API keys, dan token autentikasi dalam file konfigurasi. Jika file ini tidak dienkripsi, data sensitif dapat dengan mudah dicuri oleh penyerang. Makalah ini membahas implementasi Secure Local Storage menggunakan algoritma AES-256-CBC yang dikombinasikan dengan PBKDF2 (Password-Based Key Derivation Function 2) untuk menurunkan kunci dari password pengguna. Untuk menjamin integritas data, digunakan skema Encrypt-then-MAC dengan HMAC-SHA256. Penulis membuat program Python yang mendemonstrasikan proses enkripsi dan dekripsi data sensitif, serta menguji ketahanan sistem terhadap serangan brute force dan manipulasi data. Hasil percobaan menunjukkan bahwa kombinasi AES-256-CBC, PBKDF2, dan HMAC-SHA256 dapat melindungi data sensitif dengan aman sambil tetap praktis untuk digunakan.**

***Keywords*— AES-256, enkripsi, HMAC, key derivation, PBKDF2, secure storage.**

I. PENDAHULUAN

Dalam pengembangan aplikasi desktop, seringkali diperlukan penyimpanan data sensitif seperti kredensial database, API keys, token autentikasi, dan informasi rahasia lainnya. Data ini biasanya disimpan dalam file konfigurasi agar aplikasi dapat mengaksesnya saat dijalankan. Namun, menyimpan data sensitif dalam bentuk plaintext adalah praktik yang sangat berbahaya karena siapa pun yang memiliki akses ke file tersebut dapat langsung membaca isinya.

Menurut materi Pengantar Kriptografi [1], salah satu tujuan utama kriptografi adalah menjaga kerahasiaan (confidentiality) data. Kerahasiaan berarti menjaga agar data tidak dapat dibaca oleh pihak yang tidak berhak [1]. Untuk mencapai tujuan ini, data sensitif harus dienkripsi sebelum disimpan.

Selain kerahasiaan, integritas data juga penting untuk dijaga [1]. Integritas menjamin bahwa data tidak diubah oleh pihak yang tidak berhak selama penyimpanan. Untuk menjamin integritas, dapat digunakan Message Authentication Code (MAC) yang menggabungkan fungsi hash dengan kunci rahasia [2].

Tantangan utama dalam mengenkripsi file konfigurasi adalah pengelolaan kunci. Jika kunci enkripsi disimpan bersama file yang dienkripsi, penyerang yang mendapatkan file tersebut juga akan mendapatkan kunci. Solusi yang lebih baik adalah menggunakan password dari pengguna sebagai dasar untuk menurunkan kunci enkripsi menggunakan Password-Based Key Derivation Function (PBKDF).

Makalah ini bertujuan untuk mengimplementasikan sebuah Secure Local Storage yang mampu melindungi data sensitif dengan menggunakan AES-256-CBC sebagai algoritma enkripsi, PBKDF2 untuk menurunkan kunci enkripsi dari password pengguna, serta HMAC-SHA256 guna menjamin integritas data. Selain implementasi, makalah ini juga melakukan pengujian ketahanan sistem terhadap serangan brute force pada mekanisme derivasi kunci dan upaya manipulasi data, sehingga dapat dievaluasi tingkat keamanan solusi yang diusulkan.

II. LANDASAN TEORI

A. Kriptografi dan Layanan Keamanan

Kriptografi berasal dari bahasa Yunani, cryptós (secret) dan gráphein (writing), artinya tulisan rahasia [1]. Kriptografi adalah ilmu dan seni untuk menjaga keamanan pesan. Tujuan utama kriptografi adalah untuk menjaga empat aspek keamanan [1]:

1. Kerahasiaan (Confidentiality): menjaga agar pesan tidak dapat dibaca oleh pihak yang tidak berhak
2. Integritas data (Data Integrity): menjamin pesan tidak diubah selama penyimpanan atau pengiriman
3. Autentikasi (Authentication): memastikan pesan berasal dari sumber yang sah
4. Nirpenyangkalan (Non-repudiation): pengirim tidak dapat menyangkal telah mengirim pesan

Dalam konteks Secure Local Storage, kerahasiaan dan integritas adalah dua aspek yang paling penting. Kerahasiaan dijamin dengan enkripsi, sedangkan integritas dijamin dengan MAC.

B. Advanced Encryption Standard (AES)

Advanced Encryption Standard (AES) adalah algoritma kriptografi simetris berbasis block cipher yang ditetapkan sebagai standar enkripsi [5]. AES dirancang untuk menggantikan DES yang sudah tidak aman [5]. AES bekerja dengan ukuran blok tetap 128 bit dan mendukung panjang kunci 128, 192, dan 256 bit [5].

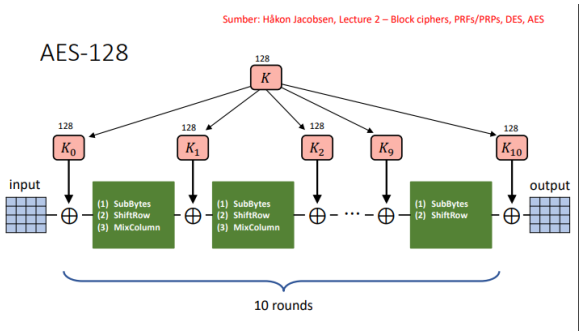
Panjang kunci menentukan jumlah putaran (round) dalam proses enkripsi [5]:

- AES-128: 10 putaran
- AES-192: 12 putaran

- AES-256: 14 putaran

Pada setiap putaran, AES melakukan empat transformasi [5]:

1. SubBytes: substitusi byte menggunakan S-box
2. ShiftRows: pergeseran baris secara wrapping
3. MixColumns: pengacakan kolom menggunakan perkalian matriks di Galois Field GF(2^8)
4. AddRoundKey: XOR state dengan round key



Gambar 1. AES-128[5]

Galois Field GF(2^m) Disebut juga medan berhingga biner. GF(2^m) atau F_2^m adalah ruang vektor berdimensi m pada GF(2). Setiap elemen di dalam GF(2^m) adalah integer dalam representasi biner sepanjang maksimal m bit[5].

C. Mode Operasi Cipher Block Chaining (CBC)

Karena AES hanya dapat memproses satu blok (128 bit) pada satu waktu, diperlukan mode operasi untuk mengenkripsi pesan yang lebih panjang dari satu blok [4]. Mode CBC (Cipher Block Chaining) adalah salah satu mode operasi yang umum digunakan.

Pada mode CBC, setiap blok plaintext di-XOR-kan dengan blok ciphertext sebelumnya sebelum dienkripsi [4]. Untuk blok pertama, digunakan Initialization Vector (IV) sebagai pengganti ciphertext sebelumnya. Rumus enkripsi CBC adalah:

C0 = IV

Ci = E(K, Pi ⊕ Ci-1) untuk i = 1, 2, ..., n

Proses dekripsi adalah kebalikannya:

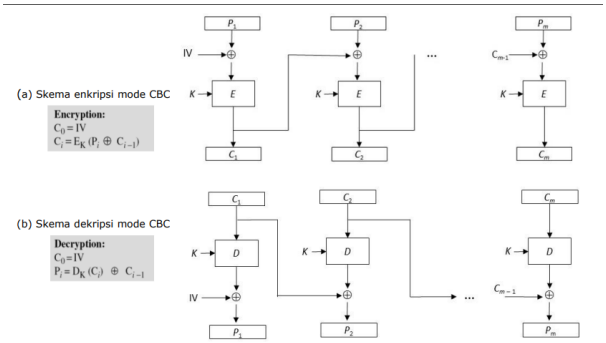
Pi = D(K, Ci) ⊕ Ci-1 untuk i = 1, 2, ..., n

Keunggulan mode CBC:

1. Blok plaintext yang sama menghasilkan ciphertext yang berbeda jika IV berbeda
2. Perubahan pada satu blok plaintext mempengaruhi semua blok ciphertext setelahnya

Kelemahan mode CBC:

1. Proses enkripsi harus dilakukan secara serial (tidak dapat diparalelkan)
2. Memerlukan IV yang unik untuk setiap enkripsi



Gambar 2. CBC

D. Fungsi Hash

Fungsi hash adalah fungsi yang mengkompresi pesan berukuran sembarang menjadi string (nilai hash) berukuran tetap [5]. Nilai hash disebut juga message digest. Fungsi hash bersifat satu arah (one-way), artinya nilai hash tidak dapat dikembalikan menjadi pesan semula [5].

h = H(M) (3)

di mana M adalah pesan dan h adalah nilai hash.

Sifat-sifat fungsi hash yang aman secara kriptografis [5]:

1. Collision resistance: sangat sukar menemukan dua input a dan b sedemikian sehingga H(a) = H(b)
2. Preimage resistance: untuk sembarang output y, sukar menemukan input a sedemikian sehingga H(a) = y
3. Second preimage resistance: untuk input a dan output y = H(a), sukar menemukan input kedua b sedemikian sehingga H(b) = y

SHA-256 adalah salah satu fungsi hash yang umum digunakan dengan panjang output 256 bit [5]. SHA-256 termasuk dalam keluarga SHA-2 dan tidak memiliki kelemahan kolisi yang diketahui.

Aplikasi fungsi hash antara lain [5]:

1. Menjaga integritas data
2. Menghemat waktu verifikasi
3. Menyimpan password dalam bentuk hash

E. PBKDF2 (Password-Based Key Derivation Function 2)

PBKDF2 adalah fungsi untuk menurunkan kunci kriptografi dari password [6]. Password biasanya memiliki entropi yang rendah (mudah ditebak), sehingga tidak aman digunakan langsung sebagai kunci enkripsi. PBKDF2 mengatasi masalah ini dengan dua mekanisme:

1. Salt: nilai acak yang ditambahkan ke password sebelum proses derivasi. Salt mencegah serangan rainbow table, di mana penyerang membuat tabel hash dari password yang umum.
2. Iterasi: password di-hash berulang kali (biasanya 100,000 kali atau lebih). Hal ini memperlambat proses derivasi kunci, sehingga serangan brute force menjadi sangat lambat.

Rumus PBKDF2:

DK = PBKDF2(PRF, Password, Salt, c, dkLen) (4)

di mana:

- PRF adalah pseudorandom function (biasanya HMAC-SHA256)
- Password adalah password dari pengguna
- Salt adalah nilai acak
- c adalah jumlah iterasi
- dkLen adalah panjang kunci yang diinginkan
- DK adalah derived key (kunci turunan)

Dengan 100,000 iterasi, setiap percobaan password memerlukan waktu sekitar 0.1-0.3 detik. Jika penyerang mencoba 1 juta password, diperlukan waktu 1-3 hari. Ini memberikan perlindungan yang signifikan terhadap serangan brute force.

F. Message Authentication Code (MAC)

MAC adalah kode yang dihasilkan dari pesan dan kunci untuk memeriksa integritas dan autentikasi pesan [2]. Berbeda dengan fungsi hash biasa yang tidak menggunakan kunci, MAC membutuhkan kunci rahasia [2].

Tag = MAC(K, M) (5)

Jika penerima menghitung MAC dari pesan yang diterima dan hasilnya sama dengan tag yang dikirim, maka pesan masih asli dan berasal dari pengirim yang sah [2].

HMAC (Hash-based MAC) adalah konstruksi MAC yang menggunakan fungsi hash [2]. HMAC-SHA256 menggunakan SHA-256 sebagai fungsi hash dasarnya:

HMAC(K, M) = H((K' ⊕ opad) || H((K' ⊕ ipad) || M)) (6)

di mana K' adalah kunci yang sudah di-pad, opad dan ipad adalah konstanta padding.

G. Skema Encrypt-then-MAC (EtM)

Ketika mengkombinasikan enkripsi dengan MAC, urutan operasi sangat penting. Ada tiga pendekatan:

1. Encrypt-then-MAC (EtM): Enkripsi terlebih dahulu, lalu hitung MAC dari ciphertext
2. MAC-then-Encrypt (MtE): Hitung MAC dari plaintext terlebih dahulu, lalu enkripsi keduanya
3. Encrypt-and-MAC (E&M): Enkripsi dan hitung MAC secara terpisah dari plaintext

Skema Encrypt-then-MAC (EtM) dianggap paling aman karena [2]:

1. MAC melindungi ciphertext dari modifikasi
2. Verifikasi MAC dilakukan sebelum dekripsi, sehingga ciphertext yang dimanipulasi langsung ditolak
3. Tidak membocorkan informasi tentang plaintext

III. IMPLEMENTASI

A. Lingkungan dan Alat

Spesifikasi teknis implementasi:

- Bahasa Pemrograman: Python 3.9+
- Pustaka Kriptografi: cryptography (untuk AES-CBC), hashlib (untuk PBKDF2 dan HMAC)
- Algoritma Enkripsi: AES-256-CBC
- Algoritma Key Derivation: PBKDF2-HMAC-SHA256
- Algoritma MAC: HMAC-SHA256

B. Arsitektur Sistem

Sistem Secure Local Storage terdiri dari empat modul utama:

1. Modul PBKDF2: Menurunkan kunci enkripsi dan kunci HMAC dari password pengguna.

```python
class PBKDF2:
    @staticmethod
    def derive_key(password: str, salt: bytes, iterations: int,
                   key_length: int, hash_algo: str = 'sha256') -> bytes:
        return hashlib.pbkdf2_hmac(
            hash_algo,
            password.encode('utf-8'),
            salt,
            iterations,
            dklen=key_length
        )

    @staticmethod
    def derive_keys(password: str, salt: bytes, iterations: int) -> Tuple[bytes, bytes]:
        # Derive master key
        master_key = PBKDF2.derive_key(
            password, salt, iterations,
            KEY_LENGTH + HMAC_KEY_LENGTH
        )

        # Split menjadi encryption key dan HMAC key
        encryption_key = master_key[:KEY_LENGTH]
        hmac_key = master_key[KEY_LENGTH:]

        return encryption_key, hmac_key
```

Gambar 3. Modul PBKDF2

2. Modul AES256CBC: Mengenkripsi dan mendekripsi data menggunakan AES-256 dengan mode CBC.

```python
class AES256CBC:
    def __init__(self, key: bytes):
        if len(key) != KEY_LENGTH:
            raise ValueError(f"Kunci harus {KEY_LENGTH} bytes (256 bit)")
        self.key = key

    def encrypt(self, plaintext: bytes) -> Tuple[bytes, bytes]:
        # Generate IV acak untuk setiap enkripsi
        iv = os.urandom(IV_LENGTH)

        # Padding PKCS7 agar panjang plaintext kelipatan 16 bytes
        padder = padding.PKCS7(128).padder()
        padded_data = padder.update(plaintext) + padder.finalize()

        # Enkripsi dengan AES-256-CBC
        cipher = Cipher(
            algorithms.AES(self.key),
            modes.CBC(iv),
            backend=default_backend()
        )
        encryptor = cipher.encryptor()
        ciphertext = encryptor.update(padded_data) + encryptor.finalize()

        return iv, ciphertext
```

Gambar 4. Potongan kode modul AES256CBC

3. Modul HMACAuth: Menghitung dan memverifikasi tag HMAC untuk menjamin integritas.

```python
class HMACAuth:
    def __init__(self, key: bytes):
        self.key = key

    def compute_tag(self, data: bytes) -> bytes:
        return hmac.new(self.key, data, hashlib.sha256).digest()

    def verify_tag(self, data: bytes, tag: bytes) -> bool:
        expected_tag = self.compute_tag(data)
        return hmac.compare_digest(expected_tag, tag)
```

Gambar 5. Modul HMACAuth

4. Modul SecureStorage: Menggabungkan semua modul untuk menyediakan API enkripsi dan dekripsi yang mudah digunakan.

```python
class SecureStorage:
    def __init__(self, iterations: int = PBKDF2_ITERATIONS):
        self.iterations = iterations

    def encrypt(self, password: str, data: Dict[str, Any]) -> EncryptedData:
        # Generate salt acak
        salt = os.urandom(SALT_LENGTH)

        # Derive keys dari password
        encryption_key, hmac_key = PBKDF2.derive_keys(
            password, salt, self.iterations
        )

        # Konversi data ke JSON bytes
        plaintext = json.dumps(data, ensure_ascii=False).encode('utf-8')

        # Enkripsi dengan AES-256-CBC
        aes = AES256CBC(encryption_key)
        iv, ciphertext = aes.encrypt(plaintext)

        # Hitung HMAC tag (Encrypt-then-MAC)
        # Tag mencakup: salt || iv || ciphertext
        auth_data = salt + iv + ciphertext
        hmac_auth = HMACAuth(hmac_key)
        hmac_tag = hmac_auth.compute_tag(auth_data)

        return EncryptedData(
            salt=salt,
            iv=iv,
            ciphertext=ciphertext,
            hmac_tag=hmac_tag,
            iterations=self.iterations
        )
```

Gambar 6. Potongan kode modul SecureStorage

C. Format File Vault

Data terenkripsi disimpan dalam file JSON dengan format:

```
{
  "version": 1,
  "iterations": 100000,
  "salt": "U88KAH4HnFtQNd2m7tCpsg==",
  "iv": "aeR06dF4n77xUxupNmCgWg==",
  "ciphertext": "XLyUdBqs85Tl0I6KRCt033Yd8uH+ZzRhE0/M724Mym8M2DPtL7e0D4PWox9l5GIE5zJ1zlk
  "hmac_tag": "GwspK0wb5Gf4ijNjXTrPOiEGXGpCTg+TPG9xhdR9h1Y="
}
```

Gambar 7. Format file vault

D. Alur Enkripsi

Proses enkripsi data sensitif:

1. Fase Persiapan:
   a. Pengguna memasukkan password
   b. Generate salt acak 128 bit
   c. Derive encryption key dan HMAC key dari password menggunakan PBKDF2
2. Fase Enkripsi:
   a. Konversi data ke format JSON
   b. Generate IV acak 128 bit
   c. Enkripsi data dengan AES-256-CBC
3. Fase Autentikasi:
   a. Gabungkan salt, IV, dan ciphertext
   b. Hitung HMAC tag dari gabungan tersebut
4. Fase Penyimpanan:
   Menyimpan semua komponen ke file dalam format JSON.

E. Alur Dekripsi

Proses dekripsi data sensitif:

1. Fase Pembacaan:
   a. Baca file vault dan parse JSON
   b. Decode semua komponen dari base64
2. Fase Derivasi Kunci:
   Derive encryption key dan HMAC key dari password menggunakan salt yang tersimpan
3. Fase Verifikasi:
   a. Gabungkan salt, IV, dan ciphertext
   b. Hitung HMAC tag dan bandingkan dengan tag yang tersimpan
   c. Jika tidak cocok, tolak dekripsi (password salah atau data dimanipulasi)
4. Fase Dekripsi:
   a. Dekripsi ciphertext dengan AES-256-CBC
   b. Parse JSON untuk mendapatkan data asli

IV. HASIL DAN ANALISIS

A. Hasil Percobaan

Percobaan dijalankan dengan data sensitif berupa kredensial database dan API keys. Berikut adalah output dari eksekusi program:

```
Secure Storage diinisialisasi
    Algoritma Enkripsi: AES-256-CBC
    Key Derivation:     PBKDF2-HMAC-SHA256
    Autentikasi:        HMAC-SHA256 (Encrypt-then-MAC)
    Iterasi PBKDF2:     100,000

  Password: MySecretPassword123!
```

Gambar 8. Output Fase Setup

```
Data sensitif (plaintext):
{
    "database": {
        "host": "db.example.com",
        "username": "admin",
        "password": "db_secret_123"
    },
    "api_keys": {
        "stripe": "sk_live_abc123xyz",
        "sendgrid": "SG.abcdefghijk"
    },
    "credentials": {
        "email": "admin@example.com",
        "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
    }
}

Ukuran plaintext: 264 bytes
```

Gambar 9. Data sensitif

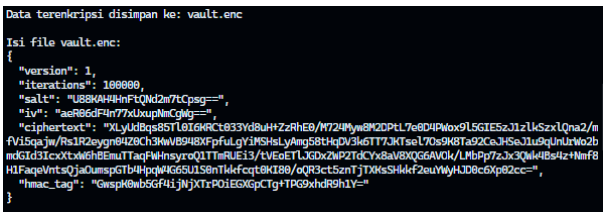
```
Langkah 1: Generate salt acak (128 bit)
    Salt: 1fbdd4677eead201c41d605dc1d2f89b

Langkah 2: Derive keys dengan PBKDF2
    Encryption Key (256 bit): 667db0353b274f7cc43d5fcf997ea51f7160982b87816612c297dc1fd2f37e1f
    HMAC Key (256 bit): 883fa8c8484d5a21fbdd055b9491630f55ce522cc852281ee321dbe83e64dd65
    Waktu derivasi: 0.302 detik

Langkah 3: Enkripsi dengan AES-256-CBC
    IV (128 bit): 69e4744e9d1789fbef1531ba93660a05a
    Ciphertext: 5cbc947411aacf394e5d08e8a442b74df761df2e1fe673461134fccef6e0ccb0f...
    Ukuran ciphertext: 272 bytes

Langkah 4: Hitung HMAC tag (Encrypt-then-MAC)
    HMAC Tag (256 bit): 1b0b292b4c1be467f88a33635d3acf3a21865c6a424e0f933c6f7185d47d8756
    Waktu total enkripsi: 0.371 detik
```

Gambar 10. Proses Enkripsi



Gambar 11. File vault

```python
print("Memuat data dari file...")
loaded_encrypted = storage.load_from_file(vault_file)

print(f"Mendekripsi dengan password: {password}")
start_time = time.time()
decrypted_data = storage.decrypt(password, loaded_encrypted)
decrypt_time = time.time() - start_time

print(f"Dekripsi berhasil! (Waktu: {decrypt_time:.3f} detik)")
print("\nData yang dipulihkan:")
print(json.dumps(decrypted_data, indent=4))

# Verifikasi
if decrypted_data == sensitive_data:
    print("\nVERIFIKASI: Data yang dipulihkan IDENTIK dengan data asli!")
```

```
Memuat data dari file...
Mendekripsi dengan password: MySecretPassword123!
Dekripsi berhasil! (Waktu: 0.213 detik)

Data yang dipulihkan:
{
    "database": {
        "host": "db.example.com",
        "username": "admin",
        "password": "db_secret_123"
    },
    "api_keys": {
        "stripe": "sk_live_abc123xyz",
        "sendgrid": "SG.abcdefghijk"
    },
    "credentials": {
        "email": "admin@example.com",
        "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
    }
}

VERIFIKASI: Data yang dipulihkan IDENTIK dengan data asli!
```

Gambar 12. Kode dan output dekripsi dengan password yang benar

```python
wrong_password = "WrongPassword456!"
print(f"Mencoba dekripsi dengan password: {wrong_password}")

try:
    storage.decrypt(wrong_password, loaded_encrypted)
    print("GAGAL: Dekripsi seharusnya ditolak!")
except ValueError as e:
    print(f"BERHASIL DITOLAK: {e}")
```

```
Mencoba dekripsi dengan password salah: WrongPassword456!
BERHASIL DITOLAK: Autentikasi gagal: Password salah atau data dimanipulasi!
```

Gambar 13. Kode dan output dekripsi dengan password yang salah

```python
print("Simulasi: Penyerang mengubah 1 byte ciphertext...")

# Buat salinan dan ubah 1 byte
tampered = EncryptedData(
    salt=loaded_encrypted.salt,
    iv=loaded_encrypted.iv,
    ciphertext=bytes([loaded_encrypted.ciphertext[0] ^ 0xFF]) +
               loaded_encrypted.ciphertext[1:],  # Flip bit pertama
    hmac_tag=loaded_encrypted.hmac_tag,
    iterations=loaded_encrypted.iterations
)

print("Mencoba dekripsi data yang dimanipulasi...")

try:
    storage.decrypt(password, tampered)
    print("GAGAL: Data yang dimanipulasi seharusnya ditolak!")
except ValueError as e:
    print(f"BERHASIL DITOLAK: {e}")
```

```
Simulasi: Penyerang mengubah 1 byte ciphertext...
Mencoba dekripsi data yang dimanipulasi...
BERHASIL DITOLAK: Autentikasi gagal: Password salah atau data dimanipulasi!
```

Gambar 14. Kode dan output dekripsi dengan byte yang diubah

B. Benchmark Iterasi PBKDF2

Pengujian dilakukan untuk mengukur waktu derivasi kunci dengan berbagai jumlah iterasi:

Tabel I Hasil Benchmark PBKDF2

Iterasi	Waktu (Detik)	Keamanan
10000	0.019	Kurang aman
50000	0.094	Minimum
100000	0.187	Standar
200000	0.374	Sangat aman
500000	0.935	Sangat aman

C. Analisis Keamanan

1. Perlindungan Kerahasiaan
   Data sensitif berhasil dienkripsi dengan AES-256-CBC. Kunci enkripsi 256 bit memberikan tingkat keamanan yang sangat tinggi. Dengan serangan brute force, diperlukan 2^256 percobaan untuk menemukan kunci, yang secara praktis tidak mungkin dilakukan.
2. Perlindungan terhadap Brute Force Password:
   Dengan 100,000 iterasi PBKDF2, setiap percobaan password memerlukan waktu ~0.19 detik. Jika penyerang mencoba 1 juta password:
   - Waktu yang diperlukan: 1,000,000 × 0.19 = 190,000 detik = ~2.2 hari
   - Dengan GPU yang lebih cepat, waktu bisa berkurang, tetapi tetap signifikan

   Salt acak 128 bit mencegah penggunaan rainbow table. Setiap vault memiliki salt yang berbeda, sehingga penyerang harus melakukan brute force dari awal untuk setiap vault.
3. Perlindungan Integritas:
   HMAC-SHA256 berhasil mendeteksi perubahan pada ciphertext. Percobaan manipulasi data langsung ditolak karena tag HMAC tidak cocok. Skema Encrypt-then-MAC memastikan verifikasi dilakukan sebelum dekripsi.
4. Perlindungan terhadap Password Salah:
   Password yang salah menghasilkan kunci yang

berbeda, yang menghasilkan tag HMAC yang berbeda. Sistem langsung menolak dekripsi tanpa memberikan informasi apakah password salah atau data dimanipulasi.

D. Perbandingan dengan Penyimpanan Plaintext

Tabel II Perbandingan Keamanan

Aspek	Plaintext	Secure Storage
Kerahasiaan	Tidak ada	AES-256
Integrits	Tidak ada	HMAC-SHA256
Perlindungan Brute Force	Tidak ada	PBKDF2 100K iterasi
Overhead Performa	0 detik	~0.2 detik

V. KESIMPULAN

Makalah ini telah mengimplementasikan Secure Local Storage untuk melindungi data sensitif pada aplikasi desktop. Kombinasi AES-256-CBC untuk enkripsi, PBKDF2 untuk derivasi kunci, dan HMAC-SHA256 untuk integritas memberikan perlindungan yang komprehensif.

Kesimpulan dari makalah ini menunjukkan bahwa kerahasiaan data dapat terjamin melalui penggunaan AES-256-CBC dengan kunci yang diturunkan dari password menggunakan PBKDF2, sehingga tidak diperlukan penyimpanan kunci secara terpisah. Integritas data berhasil dijaga menggunakan HMAC-SHA256 dengan skema Encrypt-then-MAC, yang mampu mendeteksi setiap perubahan pada data terenkripsi. Perlindungan terhadap serangan brute force diperkuat melalui penggunaan salt acak untuk mencegah rainbow table attack serta jumlah iterasi PBKDF2 yang tinggi untuk memperlambat percobaan password. Selain itu, overhead performa yang rendah sekitar 0,2 detik per operasi menunjukkan bahwa sistem ini tetap praktis untuk diterapkan pada aplikasi nyata.

VII. ACKNOWLEDGMENT

Penulis mengucapkan terima kasih kepada Bapak Rinaldi Munir selaku dosen mata kuliah IF4020 Kriptografi atas bimbingan dan materi yang telah diberikan.

REFERENCES

[1] R. Munir, "Pengantar Kriptografi," Materi Kuliah IF4020 Kriptografi, Institut Teknologi Bandung, 2025. https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2025-2026/01-Pengantar-Kriptografi-(2025).pdf

[2] R. Munir, "Message Authentication Code (MAC)," Materi Kuliah IF4020 Kriptografi, Institut Teknologi Bandung, 2025. https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2025-2026/28-MAC-2025.pdf

[3] R. Munir, "Beberapa Block Cipher (Bagian 2)," Materi Kuliah IF4020 Kriptografi, Institut Teknologi Bandung, 2025. https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2025-2026/17-Beberapa-block-cipher-bagian2-2025.pdf

[4] R. Munir, "Block Cipher (Bagian 1)," Materi Kuliah IF4020 Kriptografi, Institut Teknologi Bandung, 2025. https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2025-2026/14-Block-Cipher-Bagian1-2025.pdf

[5] R. Munir, "Fungsi Hash," Materi Kuliah IF4020 Kriptografi, Institut Teknologi Bandung, 2025. https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2025-2026/27-Fungsi-Hash-2025.pdf

[6] IETF, "PKCS #5: Password-Based Cryptography Specification Version 2.1," RFC 8018, 2017. https://datatracker.ietf.org/doc/html/rfc8018

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 31 Desember 2025

Andi Marihot Sitorus, 13522138