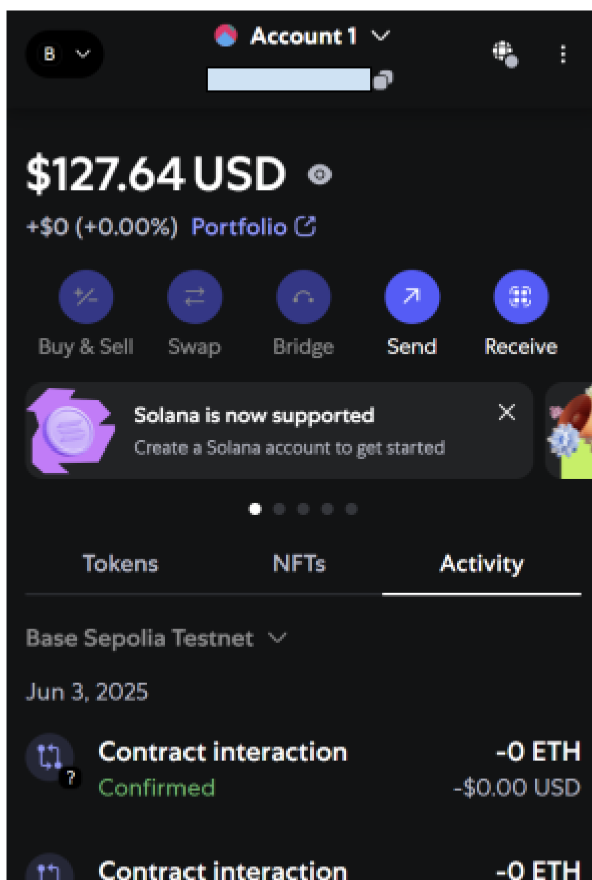
4. IMPLEMENTASI SISTEM

4.1. User Preparation

Pengguna diminta untuk meng*install* aplikasi *wallet Metamask* dan membuat akun di dalamnya. Sebagai contoh, pengguna dapat meng*install Metamask* melalui *extension browser* pada *Chrome*.

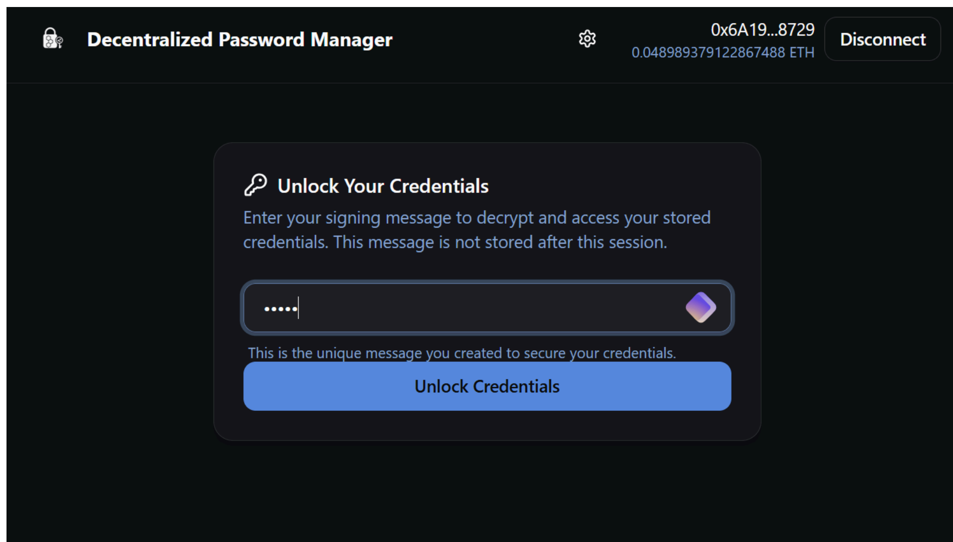


Gambar 4.1 Tampilan halaman utama akun *wallet Metamask*

Setelah pengguna berhasil membuat akun dan masuk ke dalam aplikasi *wallet Metamask* pada perangkat pengguna, *network Base Sepolia Testnet* dapat dipilih sebagai *network* yang akan digunakan sebagai *chain*. Pengguna diminta untuk mengisi *currency* dengan *claim faucet* pada *website* tertentu secara gratis dengan syarat harus memiliki *main currency* sebanyak 0.05 *ETH*. *Currency* inilah yang nanti akan digunakan untuk transaksi dengan aplikasi *password manager*.

4.2. Autentikasi *Wallet* dengan Aplikasi

Pengguna menghubungkan *wallet* ke aplikasi untuk inisialisasi koneksi antara aplikasi dengan *wallet* menggunakan *library thirdweb*.



Gambar 4.2 Tampilan halaman *Connect Wallet*

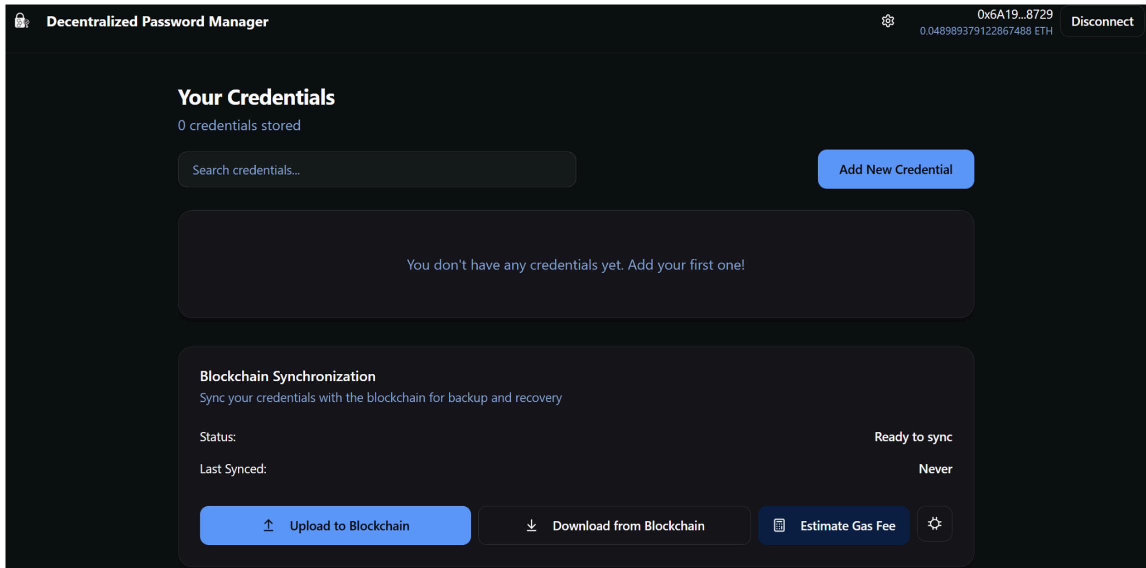
Saat user klik "*Connect Wallet*", maka aplikasi akan meminta akses *wallet* melalui ekstensi. *Password* akun *wallet* akan digunakan sebagai keamanan tambahan dalam kepemilikan *wallet*. Pada tahap selanjutnya, muncul permintaan kepada *user* untuk membuat *sign message* sebagai pengganti *master password* dalam aplikasi yang akan digunakan sebagai pembuktian bahwa *user* benar-benar merupakan pemilik *wallet* tersebut. *Signature key* dibuat menggunakan *private key wallet* pengguna dan hanya bisa dibuat oleh pemilik *wallet* itu sendiri.

4.2.1. *Signing Message* dan Enkripsi Data

Signing message adalah sebuah pesan khusus yang diminta oleh aplikasi untuk ditulis oleh pengguna ibarat adalah tanda tangan digital. Dalam proses penandatanganan ini akan menghasilkan tanda tangan digital yang unik untuk setiap akun *wallet*. *Signature* tidak hanya sebagai bukti autentikasi, tapi juga digunakan sebagai dasar untuk pembuatan kunci enkripsi. *Signature* yang dihasilkan akan diambil oleh aplikasi dan diproses melalui algoritma enkripsi untuk menghasilkan *encryption key*. *Encryption key* akan digunakan sebagai enkripsi dan dekripsi data *user* di sisi *frontend*. Hanya *user* itu sendiri yang bisa menghasilkan *signature* yang sama sehingga hanya *user* yang bisa mendekripsi datanya. *Encryption key* tidak pernah disimpan di manapun, baik di *server* ataupun dalam *chain*.

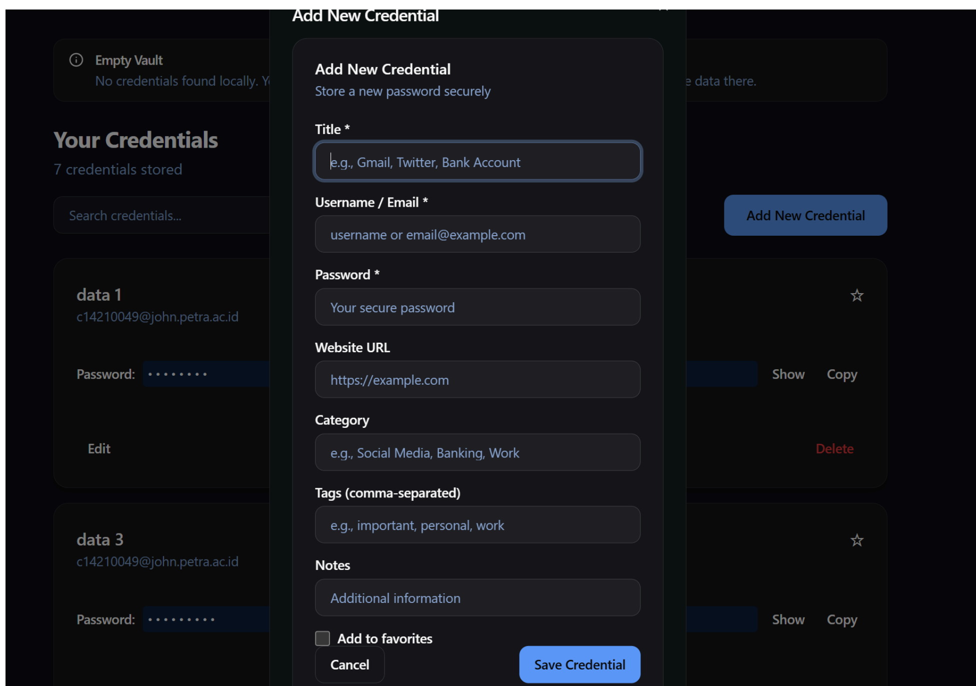
4.3. Fitur dan Kegunaan Aplikasi

Setelah pengguna berhasil masuk dengan menggunakan *signature key*, maka pengguna akan dialihkan ke halaman *home page*.



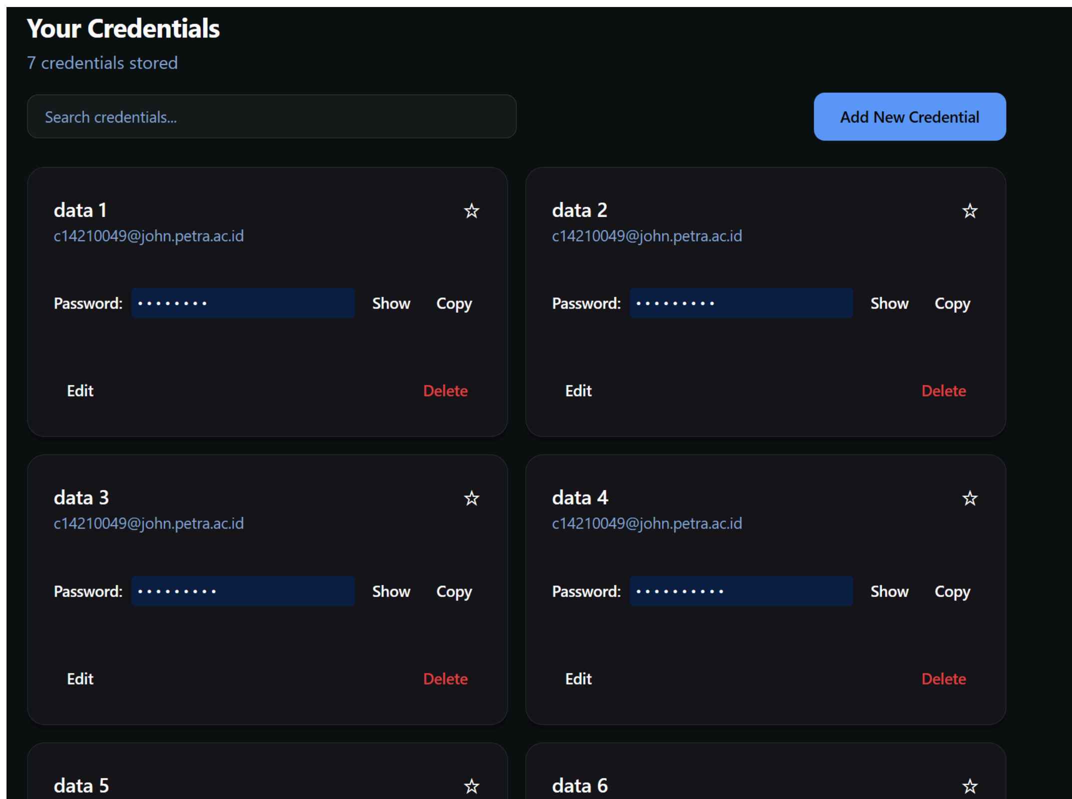
Gambar 4.3 Tampilan halaman utama aplikasi

Pada gambar 4.3, terdapat tampilan akun *wallet* di pojok kanan atas, kemudian fitur-fitur untuk *disconnect* akun *wallet*. Saat *user login*, *user* dapat menambah data baru secara lokal yang nantinya dapat di-*upload* ke dalam *chain*.



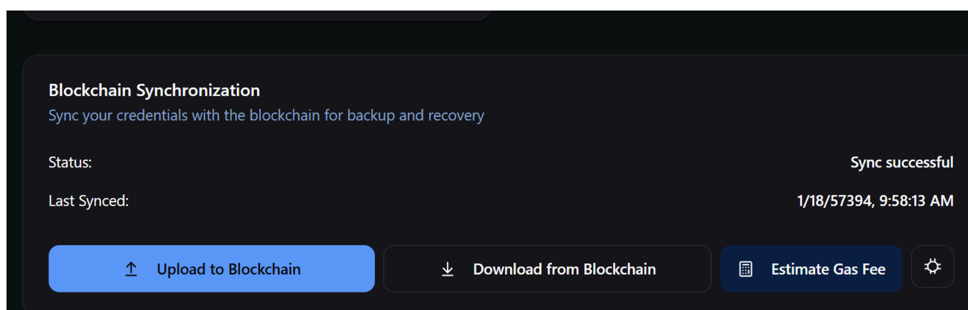
Gambar 4.4 Tampilan *pop-up add* data.

Pada gambar 4.4, *user* dapat memasukkan data berupa judul, *username/email*, dan *password* yang bersifat *mandatory*, serta *website, category, tags*, dan *notes* secara opsional. Setelah tertulis, maka data dapat disimpan dan ditampilkan dalam *home page.* Data yang disimpan secara lokal akan berujung hilang jika tidak ter-*back up*. Hal ini bertujuan agar jika *login* di perangkat lain, maka data tidak akan mudah terekspos oleh pihak luar. Oleh karena itu, untuk menghindari data hilang, maka harus ada *backup* data dan penyimpanan secara terdesentralisasi.



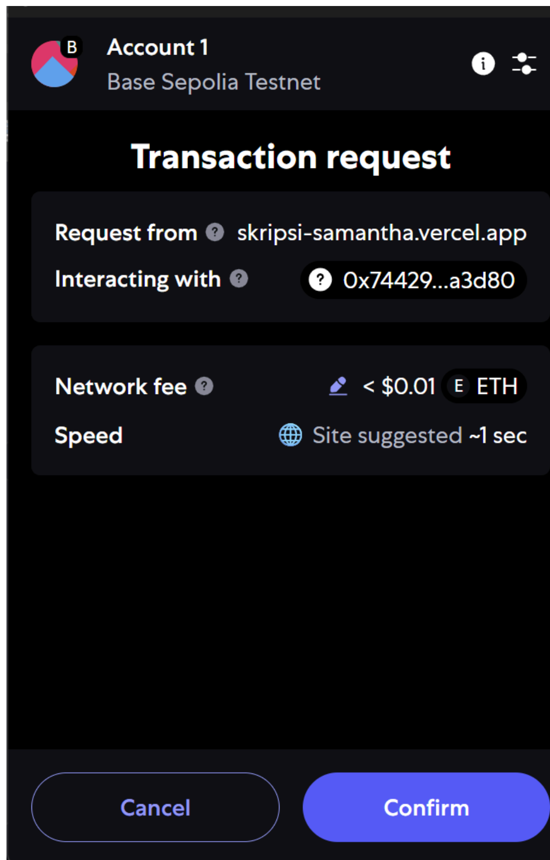
Gambar 4.5 Tampilan *credentials* pada *home page*

Pada gambar 4.5, data yang telah ditambahkan *user* dapat dilihat dengan sekaligus di*edit* jika *user* menginginkannya. Untuk mempermudah, *password* dapat dilihat secara langsung dengan menekan tombol *show*. Fitur *copy* digunakan untuk *copy password* secara cepat ke dalam *clipboard*.



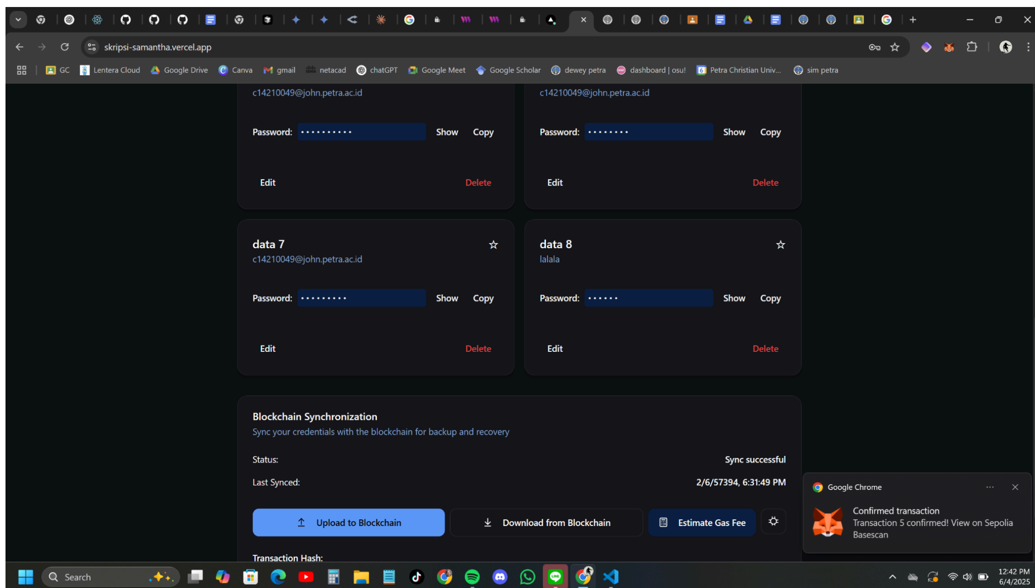
Gambar 4.6 Tampilan fitur *upload* dan *download* to *blockchain* pada *home page*

Pada gambar 4.6, *user* dapat *upload* data *credential* tersebut ke dalam *blockchain* sebagai tempat penyimpanan secara terdesentralisasi, yang memungkinkan untuk di *download* kembali suatu hari saat dibutuhkan.



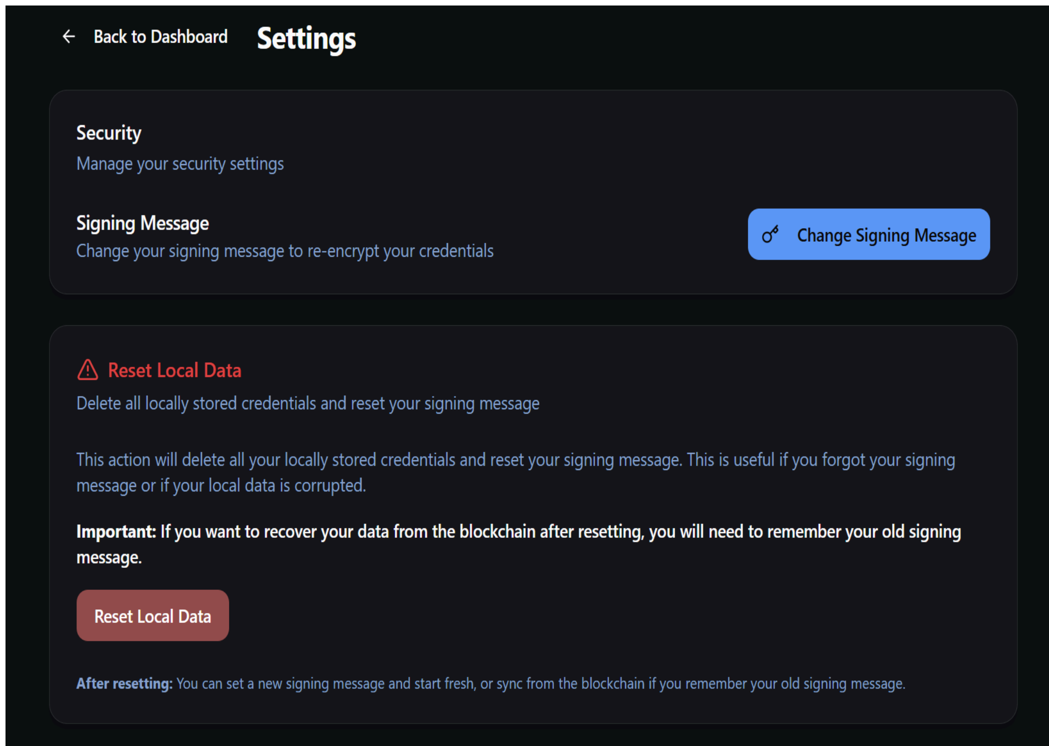
Gambar 4.7 Tampilan *pop-up request* ke *wallet* untuk konfirmasi transaksi

Pada gambar 4.7, saat *user* ingin *upload* data ke dalam *blockchain*, akan muncul konfirmasi transaksi beserta harga *gas fee* yang diperlukan. Pastikan *upload* data dilakukan dengan bijak untuk menghemat biaya.



Gambar 4.8 Tampilan *confirmed transaction*

Pada gambar 4.8, jika *user* berhasil melakukan transaksi dengan *chain*, maka akan muncul notifikasi dari *pop-up extension browser* berdasarkan banyak transaksi yang telah dilakukan pada sistem. Saat *user* ingin *download* kembali *credential* dari dalam *chain*, pastikan *sign message* sesuai dengan yang telah ditentukan dari awal oleh *user*. Jika tidak, maka *credential* yang ingin di-*download* tidak akan muncul dalam tampilan halaman. Pastikan *user* menyimpan data *credential* sesuai yang diinginkan ke dalam *chain*, karena *credential* lokal akan diperbarui dengan *credential* terakhir yang di-*upload* ke dalam *chain* setelah *user download* data dari *chain*.



Gambar 4.9 Tampilan fitur *setting*

Pada gambar 4.9, terdapat fitur *setting* yang di dalamnya terdapat 2 fitur utama untuk mengganti *sign message* dan *reset local data*. Penggantian *sign message* dilakukan jika *user* memiliki keinginan untuk merubah *signature*. Pastikan saat ingin mengganti *sign message*, data telah didekripsi terlebih dahulu. Setelah penggantian *sign message*, *upload* kembali data *credential* ke dalam *chain* untuk proses enkripsi yang baru dengan menggunakan *sign message* yang baru. Sedangkan fitur *reset local data* dapat digunakan bila *user* ingin menghapus data lokal secara cepat dan sekaligus dalam satu aplikasi.

4.4. Interaksi *Blockchain* dan *Smart Contract*

Dalam proses penyimpanan *credential* ke dalam *chain*, *credential* lokal masih belum terenkripsi dan rentan terhadap serangan ataupun kehilangan data. Maka dari itu, proses enkripsi dan dekripsi *credential* akan dilakukan saat *user* ingin melakukan interaksi penyimpanan desentralisasi dengan *chain* dan bantuan dari *Smart Contract*. Algoritma enkripsi *ChaCha20-Poly1305* akan digunakan dalam proses enkripsi dan dekripsi *credential*.

4.4.1. Proses Enkripsi Algoritma Enkripsi *ChaCha20-Poly1305*

```
export function deriveKeyFromSignature(signature: string): Uint8Array
{
  try {
    const cleanSignature = signature.startsWith('0x') ?
signature.slice(2) : signature;
    let signatureBytes: Uint8Array;

    try {
      signatureBytes = hexToBytes(cleanSignature);
    } catch {
      const paddedSignature = cleanSignature.length % 2 === 0 ?
cleanSignature : '0' + cleanSignature;
      try {
        signatureBytes = hexToBytes(paddedSignature);
      } catch {
        console.warn("Using signature string directly for key
derivation");
        signatureBytes = utf8ToBytes(signature);
      }
    }
    return sha256(signatureBytes);
```

Segmen Program 3.5 fungsi *deriveKeyFromSignature*

Signature hasil dari *signing message* yang telah ditulis oleh *user* akan diubah menjadi *encryption key* 32-*byte* dengan *hashing SHA-256* dengan menggunakan fungsi *deriveKeyFromSignature* pada segmen program 3.5.

4.4.1.1. Enkripsi pada Satu *Credential*

```
export async function encryptCredential(
  credential: Credential,
  key: Uint8Array
): Promise<EncryptedCredential> {
  try {
    const credentialJson = JSON.stringify(credential);
    const credentialBytes = utf8ToBytes(credentialJson);
    const nonce = randomBytes(24);
    const cipher = xchacha20poly1305(key, nonce);
    const encryptedBytes = cipher.encrypt(credentialBytes);
    const encryptedHex = bytesToHex(encryptedBytes);
    const nonceHex = bytesToHex(nonce);
    return {
      id: credential.id,
```

```
      encryptedData: encryptedHex,
      iv: nonceHex,
      createdAt: credential.createdAt,
      updatedAt: credential.updatedAt
    };
```
Segmen Program 3.6 fungsi *encryptCredential*

Pada segmen program 3.6, data *credential* diubah ke dalam format *JSON* dengan fungsi *JSON.stringify* pada variabel *credentialJson*. Data diubah ke dalam bentuk *JSON* agar data dapat diproses lebih lanjut dengan mudah. Setelah diubah menjadi bentuk *JSON*, data *credential JSON* diubah ke *bytes* dengan fungsi *utf8ToBytes* dan dienkripsi dengan *key* dan *nonce* acak 24-*byte* dengan fungsi *xchacha20poly1305*. Variabel *encryptedBytes* digunakan untuk mengacak data dengan enkripsi tersebut sehingga data tidak bisa dibaca orang lain tanpa *key* yang benar. Fungsi *bytesToHex* digunakan untuk mengubah hasil enkripsi dan *nonce* dari bentuk *byte* menjadi teks heksadesimal. Pengubahan ini memungkinkan data mudah disimpan ke dalam *chain*.

4.4.1.2. Dekripsi pada Satu *Credential*

```
export async function decryptCredential(
  encryptedCredential: EncryptedCredential,
  key: Uint8Array
): Promise<Credential> {
  try {
    if (!encryptedCredential.encryptedData || !encryptedCredential.iv)
{
      throw new Error('Invalid encrypted credential: missing data or
IV');
    }
    let encryptedBytes: Uint8Array;
    let nonce: Uint8Array;

    try {
      encryptedBytes = hexToBytes(encryptedCredential.encryptedData);
    } catch (error) {
      console.error('Failed to convert encrypted data from hex:',
error);
      throw new Error('Invalid encrypted data format');
    }

    try {
      nonce = hexToBytes(encryptedCredential.iv);
    } catch (error) {
```

```
      console.error('Failed to convert nonce from hex:', error);
      throw new Error('Invalid nonce format');
    }
    if (nonce.length !== 24) {
      throw new Error(`Invalid nonce length: ${nonce.length} (expected 
24)`);
    }
    const cipher = xchacha20poly1305(key, nonce);
    const decryptedBytes = cipher.decrypt(encryptedBytes);
    const decryptedJson = bytesToUtf8(decryptedBytes);
    const credential = JSON.parse(decryptedJson) as Credential;
    if (!credential.id || !credential.title || !credential.username || 
!credential.password) {
      throw new Error('Decrypted credential is missing required 
fields');
    }

    return credential;
  }
```

Segmen Program 3.7 fungsi *decryptCredential*

Pada segmen program 3.7, hal pertama yang akan dieksekusi adalah memastikan data terenkripsi dan *nonce* ada. Berkebalikan dengan enkripsi, proses dekripsi dilakukan dengan mengubah teks *hex* menjadi angka (*byte*). Validasi panjang *nonce* dilakukan untuk memastikan *nonce* yang digunakan benar-benar sepanjang 24 *byte* agar aman. Dengan fungsi yang berkebalikan urutan dengan proses enkripsi, maka proses dekripsi pun dapat dilakukan sehingga menjadi objek *credential* dapat dibaca oleh *user* dalam aplikasi. Validasi hasil dekripsi dilakukan untuk memastikan semua data penting ada pada hasil dekripsi dan menghindari data rusak atau tidak lengkap.

4.4.1.3. Enkripsi pada Lebih dari Satu *Credential*

```
export async function encryptCredentials(
  credentials: Credential[],
  key: Uint8Array
): Promise<EncryptedCredential[]> {
  if (!credentials || !Array.isArray(credentials) || 
credentials.length === 0) {
    return [];
  }

  if (!key || key.length !== 32) {
    console.error('Invalid encryption key:', key);
```

```
    throw new Error('Invalid encryption key: must be 32 bytes');
  }

  const encryptedCredentials: EncryptedCredential[] = [];
  const errors: Error[] = [];

  for (const credential of credentials) {
    try {
      if (!credential || !credential.id) {
        console.warn('Skipping invalid credential:', credential);
        continue;
      }

      const encryptedCredential = await encryptCredential(credential,
key);
      encryptedCredentials.push(encryptedCredential);
    } catch (error) {
      console.error(`Failed to encrypt credential ${credential.id}:`,
error);
      errors.push(error instanceof Error ? error : new
Error(String(error)));
    }
  }

  if (errors.length > 0) {
    console.warn(`Failed to encrypt ${errors.length} out of
${credentials.length} credentials`);
  }

  return encryptedCredentials;
}
```

Segmen Program 3.8 fungsi *encryptCredentials*

Pada segmen program 3.8, enkripsi data dapat dilakukan dengan melibatkan banyak data. Validasi input akan dilakukan dengan melakukan pengecekan pada *array*. Validasi kunci dilakukan dan memastikan jumlahnya berjumlah 32-*byte*. Setiap *credential* akan dilakukan pengecekan dan diproses dengan fungsi *encryptCredential*. Jika berhasil maka akan dimasukkan ke dalam *array* asli, dan jika *error* maka error akan dicatat dan proses tetap lanjut hingga pada *credential* terakhir.

4.4.1.4. Dekripsi pada Lebih dari Satu *Credential*

```
export async function decryptCredentials(
  encryptedCredentials: EncryptedCredential[],
```

```
  key: Uint8Array
): Promise<Credential[]> {
  if (!encryptedCredentials || !Array.isArray(encryptedCredentials) ||
encryptedCredentials.length === 0) {
    return [];
  }

  if (!key || key.length !== 32) {
    console.error('Invalid encryption key:', key);
    throw new Error('Invalid encryption key');
  }

  const credentials: Credential[] = [];
  const errors: Error[] = [];

  for (const encryptedCredential of encryptedCredentials) {
    try {
      if (!encryptedCredential || !encryptedCredential.id) {
        console.warn('Skipping invalid encrypted credential:',
encryptedCredential);
        continue;
      }

      const credential = await decryptCredential(encryptedCredential,
key);
      credentials.push(credential);
    } catch (error) {
      console.error(`Failed to decrypt credential
${encryptedCredential.id}:`, error);
      errors.push(error instanceof Error ? error : new
Error(String(error)));
    }
  }

  if (errors.length > 0) {
    console.warn(`Failed to decrypt ${errors.length} out of
${encryptedCredentials.length} credentials`);
  }

  return credentials;
}
```

Segmen Program 3.9 fungsi *decryptCredentials*

Pada langkah-langkah juga sama dilakukan pengecekan input *credential* yang dari terenkripsi, kemudian validasi kunci dan proses dekripsi dilakukan satu per satu. Jika ada *credential* yang tidak valid, maka akan dilewati. *Credential* yang berhasil didekripsi kemudian

akan diletakkan ke dalam *array* dan dikembalikan ke dalam aplikasi sehingga dapat digunakan oleh *user*.

4.4.2. Peran *Smart Contract* dalam Membantu Menjaga Keamanan *Credential*

Smart Contract berjalan di dalam *blockchain*. Aturan-aturan dalam *Smart Contract* yang telah di *deploy* tidak dapat diubah sehingga dapat terhindar dari manipulasi keamanan.

```
modifier onlySelf(address _userAddress) {
        if (msg.sender != _userAddress) {
            revert AddressMismatch(msg.sender, _userAddress);
        }
        _;
    }
```

Segmen Program 3.10 cek kepemilikan *address*

Meskipun data yang terenkripsi dalam *chain* dapat dilihat oleh pihak lain, hanya pemilik *wallet* yang dapat dipastikan dapat menulis, mengubah, atau menghapus data milik pemilik *wallet* itu sendiri. Dengan aturan *Smart Contract* diatas, pihak lain tidak dapat mengubah data milik pengguna asli.

4.4.2.1. Mapping

```
mapping(address => UserStorageEntry) private userEntries;
```

Segmen Program 3.11 *mapping*

Setiap *address* milik *wallet user* memiliki data tersendiri dalam penyimpanan tersendiri. *Mapping* digunakan untuk memudahkan *Smart Contract* dalam segi efisiensi untuk menemukan data milik *user* tertentu hanya dengan menggunakan *address wallet user*.

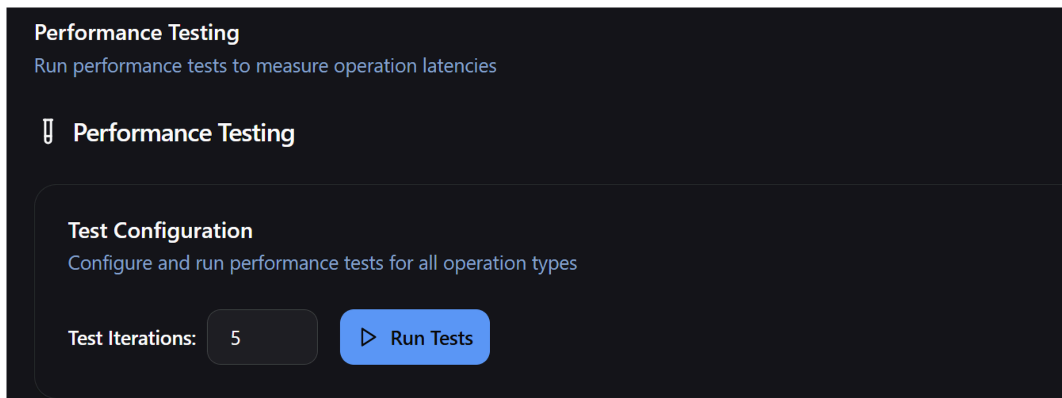
4.4.2.2. Manajemen Data *Credential*

```
function writeData(address _userAddress, bytes memory _encryptedData)
external onlySelf(_userAddress) {
        uint256 dataLength = _encryptedData.length;

        if (dataLength == 0) {
            revert CannotStoreEmptyData(); //cek data ga kosong
        }
        if (dataLength > MAX_DATA_SIZE) {
            revert DataTooLarge(dataLength, MAX_DATA_SIZE);//cek data
```

```
ga kebesaran
        }

        userEntries[_userAddress] = UserStorageEntry({
            data: _encryptedData,
            lastUpdated: block.timestamp //simpen data dan timestamp
        });

        emit DataUpdated(_userAddress, block.timestamp, dataLength);
    }
function readData(address _userAddress)
        external
        view
        returns (bytes memory currentData, uint256 timestamp)
    {
        UserStorageEntry storage entry = userEntries[_userAddress];
        if (entry.lastUpdated == 0) {
            revert NoDataFound();
        }
        return (entry.data, entry.lastUpdated);
    }
function hasData(address _userAddress) external view returns (bool) {
        return userEntries[_userAddress].lastUpdated != 0;
    }

 function deleteData(address _userAddress) external
onlySelf(_userAddress) {
        if (userEntries[_userAddress].lastUpdated == 0) {
            revert NoDataFound();
        }
        delete userEntries[_userAddress]; // Sets lastUpdated to 0
and data to empty
        // Optionally, emit an event for data deletion
        // emit DataDeleted(_userAddress, block.timestamp);
    }
```
Segmen Program 3.12 fungsi *writeData*

Setiap proses manajemen data berupa *read*, *write*, pengecekan data, serta penghapusan data disesuaikan dengan otoritas masing-masing pemilik *wallet*.

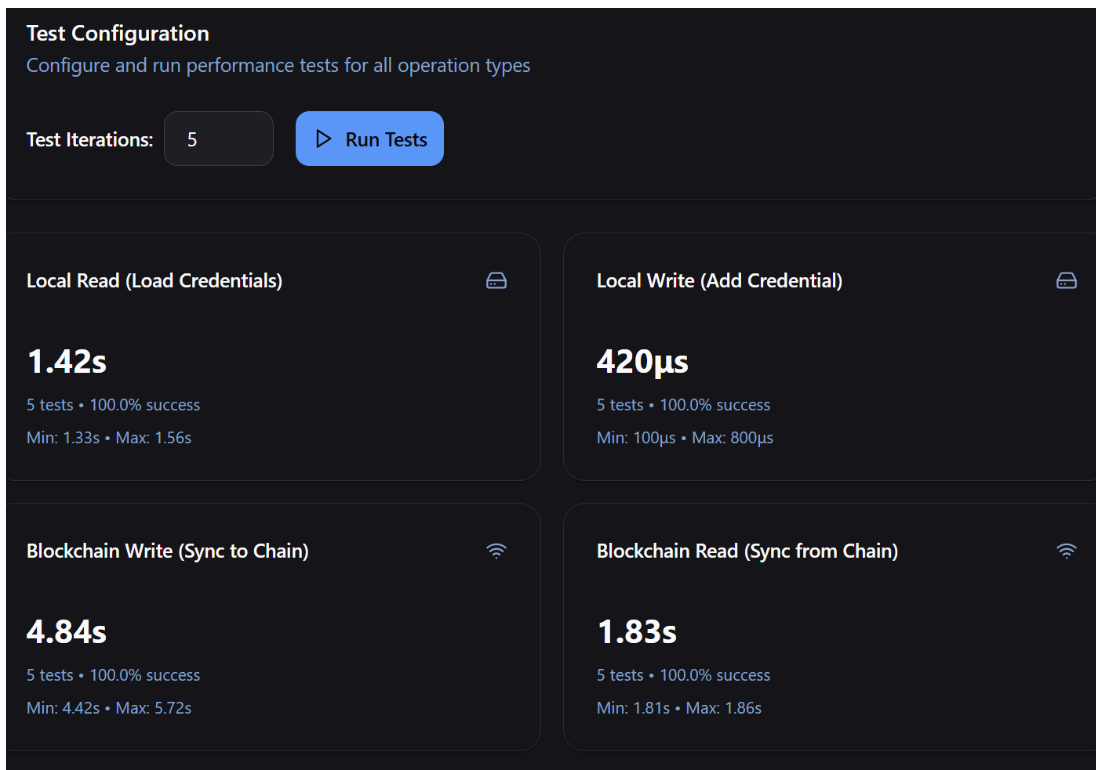
4.5. Fitur *Performance Testing*



Gambar 4.10 Tampilan fitur *Performance Testing*

Pada gambar 4.10, fitur *performance testing* digunakan untuk mengukur waktu yang dibutuhkan dalam melakukan operasi utama baik dalam *local storage* maupun interaksi dengan *blockchain*. Fitur ini memberikan layanan pengujian secara otomatis melalui fitur yang tersedia dalam aplikasi.

User mengatur jumlah iterasi pengujian yang dikehendaki melalui UI (*User Interface*) aplikasi. Aplikasi akan menjalankan empat jenis operasi utama secara berulang kali sesuai dengan jumlah iterasi, yaitu *Local Read* yang berfungsi untuk memuat data *credential* dari *local storage*, *Local Write* untuk menambah *credential* ke *local storage*, *Blockchain Write* untuk mengukur waktu sinkronisasi/*upload* data ke *blockchain*, dan *Blockchain Read* untuk mengukur waktu sinkronisasi/*download* data dari *blockchain*.



Gambar 4.11 Tampilan fitur *Performance Testing* (2)

Pada gambar 4.11, pengukuran waktu dilakukan pada keempat operasi sehingga menghasilkan hasil waktu eksekusi yang berbeda. Hasil akan ditampilkan dalam UI (*User Interface*) aplikasi yang terdiri dari waktu rata-rata tiap operasi pada tiap iterasi, waktu minimum sebagai waktu tercepat dalam satu sampel eksekusi, waktu maksimum sebagai waktu terlama dalam satu sampel eksekusi, dan tingkat keberhasilan sebagai konfirmasi bahwa eksekusi dapat berjalan dengan lancar.

4.5.1. Perhitungan Waktu Operasi

Waktu operasi merupakan selisih antara waktu selesai dengan waktu mulainya sebuah eksekusi dan dapat dihitung dengan rumus:

Waktu Operasi (ms) = Waktu Selesai - Waktu Mulai

Di mana waktu operasi dihitung dalam milisekon. Waktu mulai dicatat sebelum operasi dilakukan dan waktu selesai dicatat setelah operasi selesai. Pencatatan dilakukan dengan fungsi *performance.now()*.

```
const startTime = performance.now();
const endTime = performance.now();
```
Segmen Program 3.13 fungsi *performance.now()*

Contoh pengaplikasian rumus dapat dilihat pada fungsi perhitungan *Local Read* sebagai berikut:

```
const runLocalReadTest = async (): Promise<TestResult> => {
    try {
      const startTime = performance.now();
      await triggerManualLoadCredentials();
      const endTime = performance.now();

      return {
        operation: "Local Read (Load Credentials)",
        duration: endTime - startTime,
        success: true,
      };
    } catch (error) {
      return {
        operation: "Local Read (Load Credentials)",
        duration: 0,
        success: false,
        error: error instanceof Error ? error.message :
String(error),
      };
    }
  };
```
Segmen Program 3.14 fungsi perhitungan *Local Read*

Perhitungan waktu rata-rata, waktu minimum dan waktu maksimum dari seluruh iterasi diimplementasikan pada fungsi *getOperationStats* sebagai berikut:

```
const getOperationStats = (operation: string) => {
    const operationResults = testResults.filter(
      (r) => r.operation === operation
    );
    if (operationResults.length === 0) return null;

    const durations = operationResults
      .filter((r) => r.success)
      .map((r) => r.duration);
    const successRate =
```

```
      (operationResults.filter((r) => r.success).length /
        operationResults.length) *
      100;

    return {
      count: operationResults.length,
      average:
        durations.length > 0
          ? durations.reduce((a, b) => a + b, 0) / durations.length
          : 0,
      min: durations.length > 0 ? Math.min(...durations) : 0,
      max: durations.length > 0 ? Math.max(...durations) : 0,
      successRate,
    };
  };
```
Segmen Program 3.15 fungsi *getOperationStats*