Mothrbox WASM SDK Guide

This package provides high-security, client-side encryption for the Mothrbox storage system. It runs entirely in the browser using WebAssembly (WASM).

Supported Algorithms

- **ECC (NIST P-256):** Authenticated Encryption (ECDH + AES-256-GCM).
- **AES-256:** Password-based encryption.
- **ChaCha20-Poly1305:** High-performance password-based encryption.

---

1. Installation

1. Download the **mothrbox-wasm-0.1.0.tgz** file.
2. Place it in your project's root directory.
3. Install it using pnpm:

Bash
```
pnpm add ./mothrbox-wasm-0.1.0.tgz
```

---

2. Next.js Configuration (Required)

You must enable Async WebAssembly in your Next.js config to load the encryption module.

File: next.config.mjs

JavaScript
```
/** @type {import('next').NextConfig} */
const nextConfig = {
  webpack(config) {
    config.experiments = {
      ...config.experiments,
      asyncWebAssembly: true,
      layers: true,
    };
    return config;
  },
};

export default nextConfig;
```

---

3. React Integration

Since WASM loads asynchronously, create a custom hook to ensure the library is ready before you try to encrypt/decrypt data.

File: src/hooks/useMothrbox.ts

TypeScript

```
import { useEffect, useState, useRef } from 'react';
// Import the init function and all exported methods
import initWasm, {
  ecc_generate_key,
  ecc_encrypt,
  ecc_decrypt,
  aes_encrypt,
  aes_decrypt,
  chacha_ecrypt, // Note: Matches Rust function name
  chacha_decrypt
} from 'mothrbox-wasm';

export const useMothrbox = () => {
  const [isReady, setIsReady] = useState(false);
  const initializing = useRef(false);

  useEffect(() => {
    // Prevent double-initialization in React Strict Mode
    if (initializing.current) return;
    initializing.current = true;

    initWasm()
      .then(() => setIsReady(true))
      .catch((err) => console.error("❌ Mothrbox WASM Failed:", err));
  }, []);

  return {
    isReady,
    ecc_generate_key,
    ecc_encrypt,
    ecc_decrypt,
    aes_encrypt,
    aes_decrypt,
    chacha_ecrypt,
    chacha_decrypt
  };
};
```

---

4. API Reference

Identity (ECC)

Function	Inputs	Returns	Description
ecc_generate_key()	None	{ private, public }	Generates a new P-256 Keypair.
ecc_encrypt(...)	data (Uint8Array) recipientPub (Uint8Array) myPriv (Uint8Array)	Uint8Array	Encrypts data for a user and signs it with your key.
ecc_decrypt(...)	encryptedData (Uint8Array) myPriv (Uint8Array)	Uint8Array	Decrypts message and verifies sender integrity.

Password Based (AES & ChaCha20)

Function	Inputs	Returns	Description

aes_encrypt(...)	data (Uint8Array) password (String)	Uint8Array	Encrypts using AES-256-GCM.
aes_decrypt(...)	data (Uint8Array) password (String)	Uint8Array	Decrypts AES data.
chacha_ecrypt(...)	data (Uint8Array) password (String)	Uint8Array	Encrypts using ChaCha20-Poly1305.
chacha_decrypt(...)	data (Uint8Array) password (String)	Uint8Array	Decrypts ChaCha20 data.

---

5. Usage Example

File: src/app/page.tsx

TypeScript
```
'use client';

import { useMothrbox } from '@/hooks/useMothrbox';
```

```
export default function EncryptionTest() {
  const { isReady, ecc_generate_key, ecc_encrypt, aes_encrypt } = useMothrbox();

  const handleTest = () => {
    if (!isReady) return;

    try {
      const msg = new TextEncoder().encode("Secret Data");

      // --- Scenario A: Sending to another user (ECC) ---
      const alice = ecc_generate_key();
      const bob = ecc_generate_key();

      const eccBlob = ecc_encrypt(msg, bob.public_key, alice.private_key);
      console.log("ECC Encrypted:", eccBlob);

      // --- Scenario B: Password Protecting a File (AES) ---
      const aesBlob = aes_encrypt(msg, "my-strong-password");
      console.log("AES Encrypted:", aesBlob);

    } catch (err) {
      console.error("Encryption Error:", err);
    }
  };

  if (!isReady) return <div>Loading Secure Module...</div>;

  return <button onClick={handleTest}>Run Encryption Test</button>;
}
```