SHA256_HMAC Encrypt Project

Adrian Konwel

1.) making function to compute SHA-256 Hash to derive a key from user's password

→ obv void() to return nothing

↳ arguments: char *input, unsigned char output[SHA_DIGEST_LENGTH];

↳ pointer to a char to make a string, 'input'

→ unsigned char represents a byte (8 bits)

→ Each element in this array stores a single byte (0-255)

→ output[SHA_DIGEST_LENGTH] declares an array that can store 32 bytes which is the output of SHA-256 (256 bits)

2.) Making a function that computes the HMAC (Hash-based Message Authentication Code) using SHA-256. This will verify the integrity and authenticity of a message using a secret key.

→ obv void(), return nothing

↳ arguments: const char *key, const char *message, unsigned char output[SHA256_DIGEST_LENGTH]

↳ Pointer to a char, make a string to store secret key used for HMAC

↳ Pointer to a char, make a string so the message to be authenticated

↳ output array (32 bytes) to store the HMAC-SHA-256-DIGEST

Unsigned int len = SHA256_DIGEST_LENGTH;

Unsigned int -> represents a non-negative whole number

SHA256_DIGEST_LENGTH -> its a constant defined in OPENSSL as a 32 byte (256-bit) output of SHA_256

```
HMAC(EVP_sha256(), Key, strlen(key), (unsigned char*)message,
        strlen(message), output, &len);
```

-> So by default the HMAC() function always requires 7 arguments

... = means for example

- 1st => Hash function (e.g, SHA-256) ... const EVP_MD or *evp_md
- 2nd => Secret key ... const void *key
- 3rd => length of the key ... int key_len
- 4th => message data ... const unsigned char *data
- 5th => length of the message ... int data_len
- 6th => Output Buffer for the HMAC Digest ... unsigned char *md
- 7th => Output length of the digest ... unsigned int *md_len

-> HMAC(), requires an EVP_MD struct., which EVP_sha256() provides. SHA256() computes raw sha-256 hash!!! NOT HMAC

-> Key, User-provided input from void function arguments used for HMAC generation

-> strlen(key), Computes length of key in bytes, HMAC needs to know exact key size to perform hashing properly

→ Remember if your key is longer than SHA-256's block size (64 bytes), HMAC() automatically hashes to 32 bytes

→ (unsigned char *) message, message which is data whose integrity we are verifying. HMAC() expects binary data, so message is orginally a char * in void function arguments so we cast to (unsigned char *)

→ Strlen(message), computes length of the message in bytes. Tells HMAC(), exactly how many bytes to process

→ output, stores the HMAC-SHA-256 digest. Preallocated buffer (32 bytes) to store final HMAC digest. Result is a binary value, later converted to Hexa for display.

→ &len, stores the length of the output digest. Pointer to unsigned int, where OpenSSL stores the actual output length.

New Encryption Method, Storing PBKDF2-Hashed Passwords securely with AES-256 Encryption

PBKDF2 (Password-Based Key Derivation Function 2)

↳ Crypto algorithm that derives a key from a password. It's used to make passwords secure by make brute-force attacks more diffucit

→ So we hash the password with PBKDF2-HMAC-SHA256

↳ Converts password into secure 256-bit (32-byte) hash using 100,000 iterations

↳ Random salt ensures that no 2 users with same password can have same hash.

→ Now encrypt hashed password using AES-256-GCM

↳ Provides authentication, ensuring when the AES-256-GCM encrypts the PBKDF2-hashed password to prevent unauthorized access.

Process: 1.) User is prompted to enter password

2.) A random 16 or 32 byte salt is generated
3.) Now we set an iteration count for PBKDF2 example is 100,000 iterations.
4.) Runs HMAC-SHA256 hashing password 100,000 times
5.) PBKDF2 uses 16/32 byte random generated salt and hashes it 100,000 times
6.) Now after hashing password and key, PBKDF2 derives a secure 256-bit (32-byte) key as an output.

IV is needed to decrypt AES-GCM

Authentication tag is 16 bytes

7.) Now AES-256-GCM generates a 12-byte IV (initialization vector) to ensure encryption is unique
8.) Now encrypts PBKDF2-derived key with AES-256-GCM
9.) Then AES-256-GCM produces an authentication tag to ensure when we use the IV when decrypting the authentication tag will confirm Data integrity and authenticity

→ Even if the attacker modifies 1 byte of encrypted password, authentication tag will not match. When decryption func. runs, it will fail.

Enhancing Security with a Hardware Security Module (HSM)

→ HSM's are hardened tamper-resistant hardware devices that secure cryptographic processes by generating, protecting, and managing keys used for encrypting and decrypting data and creating digital signatures and certificates.

→ Performs AES encryption inside a tamper-proof hardware.

→ Ensures only authorized users can encrypt/decrypt passwords.

Derive Key is generated in main() function

↳ unsigned char derived_key[DERIVED_KEY_SIZE]; // 256-Bit AES key

↳ derive_key_PBKDF2(password, salt, derived_key);

→ But the Security Risk is that the derived_key is stored in memory

↳ An attacker would exploit this by a buffer overflow attack / memory dumps

SECURE HSM APPROACH

1.) PBKDF2 generates derived_key, instead of keep it in RAM

↳ Encrypt derived_key inside an HSM (using a KEK (key encryption key) stored in the HSM

↳ Store only encrypted derived_key inside database (HSM)

2.) When decryption is needed:

↳ Retrieve encrypted derived_key from HSM (database)

↳ Unwrap (decrypt) inside HSM (won't ever expose key)

↳ Use unwrapped derived_key for AES encryption.

→ Now remember if using a HSM-AES256 based encryption/decryption program, no password will be inputted from user. This is due to weak user-generated passwords and instead of deriving a AES key from password, we generate inside HSM so it's never in memory.

Yes, technically the derived_key is the DEK(Data encryption key)

↳ Data encryption key(DEK) is the key used for encrypting/decrypting data with AES-256-GCM

Decryption Steps DEK/KEK

Steps	Action	where is key?
1. Retrieve wrapped DEK (encrypted)	Load encrypted DEK from Storage	Still encrypted in Database(HSM)
2. Unwrap DEK inside HSM	Decrypt DEK using the KEK inside HSM	now in memory only temporarily
3. Retrieve encrypted data and IV	Load the AES-256 ciphertext	Data is still encrypted
4. Decrypt using AES-256-GCM	use unwrapped DEK to decrypt data	DEK in memory only temporarily
5. Print the decrypted message	output everything from original plaintext	DEK cleared from memory

→ Now for the program

↳ RAND_bytes() and RAND_bytes(unsigned char *kek)

↳ generates num random bytes using a cryptographically secure pseudo random generator and stores in buffer

↳ RAND_bytes (unsigned char *buf, int num);

↳ Takes two arguments by default, buf -> pointer to buffer wher random bytes will be stored. num -> number of random bytes to generate

↳ RETURN VALUE: returns 1 if succesful, this means random bytes generated sucessfully
↳ Returns 0: if it fails (not secure or error occurred)

HSM_AES256

↳ generate_KEK(unsigned char *kek)

↳ generates 256-bit KEK using RAND_bytes openssl.
↳ The generate_DEK is for data encryption key generation

↳ Remember all stored in HSM (simulated)

→ wrap_DEK_with_KEK()
↳ uses AES256-ECM mode to encrypt DEK using KEK
↳ uses openSSL's EVP API

↳ Initializes encryption context (EVP_CIPHER_CTX_new())
↳ uses EVP_EncryptInit(), EncryptUpdate(), EncryptFINAL() to encrypt
↳ Then cleans context using EVP_CIPHER_CTX_free()

↳ A security concern that I thought about but haven't fixed is AES-ECB mode isn't totally recommended for key encryption due to lack of randomization (the IV is NULL)

A Possible Solution can be: AES-256-KW

Which is a key wrap, 5 round Feistel Network, One half of AES block is used to encrypt the key, second half of last permutation is used to compute 64-bit MAC

↳ Haven't tested it yet!!!

→ Also can use AES key wrap Algo. (RFC 3394) instead of ECB mode because ECB mode doesn't use IV, making it possibly predictable if keys are encrypted multiple times

→ encrypt_AES_GCM()

↳ Initializes encryption context

↳ sets IV (rand generated) using RAND_bytes

↳ encrypts data using EVP_EncryptUpdate()

↳ Finalizes encryption using EVP_EncryptFinal()

↳ Extracts Auth Tag using EVP_CIPHER_CTX_ctrl()

↳ allows various cipher specific parameters to be determined and set.

↳ ctx, encryption context; out, buffer to store final encrypted data; outlen, pointer to store number of bytes written