Hanoi University of Science and Technology

School of Information and Comunication Technology

Applied Cryptography

Project 1

Supervisor: PhD. Tran Vinh Duc
Class: 157216
Group: 7
Member: Dao Minh Quang 20225552
Vu Duc Thang 20225553
Le Anh Tuan 20225559

Hanoi, 3/2025

1. API Implementation

1.1. constructor(masterKey, hmacKey, aesKey, salt)

- Purpose: Initializes the Keychain object
- Parameters:
  - masterKey: The derived master key from PBKDF2
  - hmacKey: Key used for domain name hashing
  - aesKey: Key used for password encryption
  - salt: Random salt used in key derivation
- Stores two main objects:
  - this.data: Public information like the key-value store (KVS) and salt
  - this.secrets: Cryptographic keys that must be kept private

1.2. static async init(password)

- Purpose: Create a new, empty keychain with a master password
- Key steps:
  - Generate a random 128-bit salt
  - Import the password as key material
  - Use PBKDF2 to derive a master key
  - Create two sub-keys:
    - One for HMAC (domain name hashing)
    - One for AES encryption
  - Return a new Keychain instance
- Security features:
- Uses 100,000 PBKDF2 iterations to slow down brute-force attacks
- Derives different keys for different purposes
- Generates a unique salt for each keychain

1.3. static async load(password, repr, trustedDataCheck)

- Purpose: Reconstruct a keychain from a serialized representation
- Parameters:
  - password: Master password
  - repr: JSON-encoded keychain data
  - trustedDataCheck: Optional SHA-256 hash for integrity verification
- Key steps:
  - Parse the JSON representation
  - Optionally verify data integrity using the provided hash
  - Recreate cryptographic keys using the same process as init()
  - Restore the key-value store
- Security features:
  - Verifies data integrity if a trusted hash is provided

    - Prevents loading tampered data
    - Recreates keys using the same secure derivation process

1.4. async dump()

- Purpose: Serialize the keychain for storage
- Returns:
    - An array with two elements:
        - JSON-encoded keychain data
        - SHA-256 hash of the serialized data
- Key steps:
    - Create a serializable object with KVS and salt
    - Convert to JSON
    - Compute SHA-256 hash of the JSON string
- Security features:
    - Allows reconstructing the keychain
    - Provides a hash for integrity verification
    - Does not include any secret keys in the serialization

1.5. async get(name)

- Purpose: Retrieve a password for a given domain
- Key steps:
    - Hash the domain name using HMAC
    - Look up the hashed domain in the KVS
    - If found, decrypt the password
    - Return the decrypted password or null
- Security features:
    - Domain names are not stored in clear text
    - Passwords are encrypted with AES-GCM
    - Uses a unique IV for each encryption
    - Prevents learning anything about stored domains or passwords

1.6. async set(name, value)

- Purpose: Add or update a password for a domain
- Key steps:
    - Validate password length
    - Hash the domain name
    - Generate a random IV
    - Encrypt the password using AES-GCM
    - Store the encrypted data in the KVS
- Security features:
    - Prevents storing passwords longer than 64 characters
    - Uses a unique IV for each encryption

- Hides domain names and password contents

1.7. async remove(name)

- Purpose: Remove a password entry for a domain
- Key steps:
  - Hash the domain name
  - Check if the domain exists in the KVS
  - Remove the entry if found
- Returns:
  - true if the entry was removed
  - false if the domain was not found

2. Short-answer Questions

2.1. Briefly describe your method for preventing the adversary from learning information about the lengths of the passwords stored in your password manager.

In our implementation, we prevent an adversary from learning information about password lengths through several techniques:

- We limit the maximum password length to 64 characters
- We use AES-GCM encryption with a fixed block size
- Each password is padded to a consistent length during encryption
- The encryption uses a unique IV prepended to the ciphertext
- This ensures that encrypted passwords always appear to have a uniform length, regardless of the original password's actual length

2.2. Briefly describe your method for preventing swap attacks (Section 2.2). Provide an argument for why the attack is prevented in your scheme.

Our defense against swap attacks is multi-layered:

- Domain names are hashed using HMAC with a master key before being used as KVS keys
- Each password is individually encrypted with a unique IV
- The encryption key is derived from the master password and never directly exposed
- Even if an attacker tries to swap entries, the HMAC-based keys would not match, and the decryption would fail
- Any attempt to swap entries would result in decryption errors or incorrect data, effectively preventing unauthorized record interchange

2.3. In our proposed defense against the rollback attack (Section 2.2), we assume that we can store the SHA-256 hash in a trusted location beyond the reach of an adversary. Is it necessary to assume that such a trusted location exists, in order to defend against rollback attacks? Briefly justify your answer.

A trusted storage location is not strictly necessary to defend against rollback attacks. Alternative approaches include:

- Embedding a version or timestamp with each record
- Using a cryptographic counter in the encryption process
- Implementing a sequence number for each entry that prevents reverting to previous states
- The SHA-256 hash serves as an additional integrity check, but secure design can provide rollback protection without external trusted storage

2.4. Because HMAC is a deterministic MAC (that is, its output is the same if it is run multiple times with the same input), we were able to look up domain names using their HMAC values. There are also randomized MACs, which can output different tags on multiple runs with the same input. Explain how you would do the look up if you had to use a randomized MAC instead of HMAC. Is there a performance penalty involved, and if so, what?

If using a randomized MAC instead of HMAC:

- Each domain would need to be stored with its MAC tag
- Lookup would require iterating through all entries and attempting to verify each tag
- Performance penalty would be significant: O(n) instead of O(1)
- Would require storing additional metadata for each entry
- Computational complexity would increase substantially compared to deterministic HMAC

2.5. In our specification, we leak the number of records in the password manager. Describe an approach to reduce the information leaked about the number of records. Specifically, if there are k records, your scheme should only leak ⌊log2(k)⌋ (that is, if k1 and k2 are such that ⌊log2(k1)⌋ = ⌊log2(k2)⌋, the attacker should not be able to distinguish between a case where the true number of records is k1 and another case where the true number of records is k2).

Potential approaches to minimize record count information:

- Use a fixed-size bucket system where entries are grouped

- Pad the KVS to a power-of-two number of entries
- Use a sparse data structure that doesn't reveal empty slots
- Implement a logarithmic-based storage mechanism
- Add dummy/decoy entries to obscure the true number of records

2.6. What is a way we can add multi-user support for specific sites to our password manager system without compromising security for other sites that these users may wish to store passwords of? That is, if Alice and Bob wish to access one stored password (say for nytimes) that either of them can get and update, without allowing the other to access their passwords for other websites.

To support multi-user access to specific site passwords securely:

- Use public-key encryption for the shared domain
- Create a separate encryption key for the shared domain
- Store this key using a group key agreement protocol
- Implement access control lists (ACLs) for the specific domain
- Ensure the shared key is isolated from individual user passwords
- Use asymmetric encryption to allow multiple users to decrypt the shared password
- Maintain separate encryption for all other domain passwords