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. __init__(self, keychain_password, salt, kvs)

- Purpose: Initializes the Keychain object
- Parameters:
  - ***keychain_password***: The master password for deriving encryption keys.
  - ***salt***: Random salt used for key derivation
  - ***kvs***: The stored key-value dictionary
- Security features:
  - Uses **PBKDF2** to derive a master key from the password and salt
  - Generates two sub-keys:
    - ***hmac_key*** for hashing domain names
    - ***aes_key*** for encrypting stored passwords
  - Keeps cryptographic keys in a private ***secrets*** dictionary

1.2. @staticmethod new(keychain_password)

- Purpose: Create a new, empty keychain with a master password
- Key steps:
  - Generate a random 128-bit salt
  - Use **PBKDF2** to derive a master key
  - Return a new Keychain instance with an empty password database
- Security features:
  - Uses 100,000 **PBKDF2** iterations to slow down brute-force attacks
  - Unique salt ensures different keychains have distinct encryption keys

1.3. @staticmethod load(keychain_password, repr, trusted_data_check=None)

- Purpose: Loads an existing keychain from a serialized representation
- Key steps:
  - Parses the **JSON**-encoded keychain data
  - Derives encryption keys using **PBKDF2** with the provided password and salt
  - Validates password correctness by attempting decryption on an entry
  - Optionally verify data integrity using **SHA-256** checksum
- Security features:
  - Ensures that only the correct password can decrypt stored credentials
  - Protects against rollback attacks by verifying the checksum

1.4. dump(self)

- Purpose: Serialize the keychain for storage
- Returns:
  - A tuple containing:

        - **JSON**-encoded keychain data
        - **SHA-256** hash of the serialized data
- Security features:
    - Stores only encrypted passwords, ensuring confidentiality
    - Prevents rollback attacks by including an integrity hash

1.5. get(self, domain)

- Purpose: Retrieve a password for a given domain
- Key steps:
    - Hash the domain name using **HMAC-SHA256**
    - Look up the hashed domain in the **KVS**
    - If found, decrypt the password using **AES-GCM**
- Security features:
    - Domain names are not stored in clear text
    - Passwords are encrypted with **AES-GCM**

1.6. set(self, domain, password)

- Purpose: Add or update a password for a domain
- Key steps:
    - Validate password length using **HMAC-SHA256**
    - 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

1.7. remove(self, domain)

- Purpose: Remove a password entry for a domain
- Key steps:
    - Hash the domain name using **HMAC-SHA256**
    - 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
- Security features:
    - Prevents attackers from accessing domain names in plaintext
    - Ensures deleted entries are completely removed from memory

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:

- Each password is encrypted using **AES-GCM**, ensuring that the encrypted data does not directly reveal the original password length.
- A unique IV (nonce) is generated for every encryption operation, making ciphertexts appear uniformly random.
- We enforce a maximum password length of 64 characters, preventing adversaries from distinguishing password lengths by analyzing ciphertext size.

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-SHA256** before being stored, making it impossible to swap entries based on domain name visibility.
- Each password is individually encrypted with **AES-GCM** using a unique IV, ensuring that a swapped ciphertext cannot be decrypted correctly.
- When calling ***get()***, if an entry has been swapped, the decryption will either fail due to an authentication error or return incorrect data, preventing unauthorized record swapping.

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:

- A **SHA-256** checksum of the entire key-value store is generated when calling ***dump()*** and verified when calling ***load()***.
- If an attacker attempts to restore an old version of the password manager, the checksum comparison will fail, triggering a ***ValueError("Checksum failed! Possible rollback attack.").***
- This ensures that tampered or outdated data cannot be loaded without detection.

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:

- HMAC is deterministic, meaning the same input always produces the same output, allowing for efficient lookups.
- If a randomized MAC was used, the MAC tag would change each time, preventing direct lookups by domain.
- Possible solutions:
    - Store both the original domain and its MAC tag in the key-value store
    - Perform linear search across all stored entries, verifying each MAC tag individually
- Downside: This would result in an O(n) lookup time instead of O(1), significantly reducing performance

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 solutions:
    - Padding the database with dummy entries, making it unclear which records are real
    - Using bucket-based storage, where records are grouped into fixed-size buckets (e.g., groups of 16, 32, or 64 entries).
    - Obfuscating record count using a logarithmic approach, so the actual number of entries is hidden within a larger estimated range.

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:

- Each user has a public/private key pair.
- Shared passwords are encrypted using **AES-GCM**, and the **AES** key is encrypted with the public keys of users who have access.
- Each record includes a list of users who are authorized to access the password.
- Only users in the access control list can decrypt and retrieve the password.
- If multiple users need access to a password, a secure key exchange protocol (e.g., **ECDH**) can be used to derive a shared encryption key without exposing individual user credentials.