CS 346: Cryptography **Fall 2024**

Homework 3: Password Manager

Due: October 15, 2024 at 11:59pm (Submit on Gradescope) **Instructor:** David Wu

Instructions. In this assignment, you will be implementing the back-end for a password manager system using the Python `cryptography` library. Please carefully read the *entire* specification and work out a design *before* starting your implementation. Your submission will be primarily evaluated for *security*. A functional, but insecure, implementation will receive very little to no credit. The starting implementation we provide satisfies all of the functionality requirements, but none of the security requirements.

Collaboration Policy. For this assignment, you may work in groups of up to two. If you work in a group of two, both of you will receive the same grade on the assignment. Only **one** person in the group should submit the assignment on Gradescope and the submitter is responsible for listing all group members in Gradescope. You may discuss your general *high-level* strategy with other students, but you may *not* share any written documents or code (this includes pseudocode and concrete implementation details) with students outside your group. You are not allowed to look at the code of students outside your group. Please refer to the course website for the full description of the collaboration policy.

Acknowledgments. This homework assignment is adapted from a similar assignment from Stanford's CS 255 course by Prof. Dan Boneh.

1 Overview

In many existing software systems, the primary weakness is the user's password. For security, it is important that users rely on strong passwords or passphrases to secure their accounts. However, strong passwords can be long and difficult to remember. This problem is further compounded by the fact that users should use different passwords for each service.

One way for users to manage their passwords is with a password manager. The password manager maintains a mapping between services and login credentials and is protected by a single "master password." With a password manager, users only have to remember the single master password. The burden, however, is now on the password manager to properly protect the user's credentials.

In this assignment, you will be writing the back-end for a secure password manager. As part of this project, you will be making use of a number of symmetric cryptographic primitives that we have discussed so far. Since it is rarely advisable to implement these low-level cryptographic primitives, you will be using the Python `cryptography` library in your implementation.

2 Secure Password Manager

The password manager will internally maintain a key-value store that maps domains (keys) to passwords (values). For example, the (plaintext) contents of the key-value store might look like the following:

Domain (Key)	Password (Value)
example.com	123456
facebook.com	gj3eXv6Qkf3ZHHQBNXrCSAqj
utexas.edu	ezTR34mehDz8G3yhk2yNG9rd

In this project, you will implement the API for a back-end implementation of a password manager. The API will support serialization and deserialization methods for loading and writing the contents of the password manager to disk, as well as methods for adding, fetching, and removing entries from the password manager. We impose the following security requirements on both the serialized *as well as* the in-memory representation of the key-value store:[1]

- **String encoding:** Throughout this project, you may assume that all domain names and passwords are ASCII strings.
- **Domain encoding:** We want to hide the domains in the password manager while still enabling efficient look-ups. To support this, instead of using the domain x itself as the key in the key-value store, you will use $\mathsf{HMAC}(k, x)$, where k is an HMAC key.
- **Password storage:** The passwords in the key-value store should be encrypted using an authenticated encryption scheme. Since there can be a large number of entries in the password manager, each password should be encrypted and stored *separately*. You should not encrypt the entire contents of the key-value store as a single ciphertext (otherwise, you would have to decrypt the entirety of the password manager for each lookup).
- **Hiding password length:** The password manager should not leak any information about the length of the passwords. To make this feasible, you may assume that the maximum length of any password is 64 bytes (i.e., 64 ASCII characters).
- **Key derivation:** The password manager itself is protected by a master password. When the user initializes a password manager or loads the contents of the password manager from disk, they must provide the master password. The master password should be used to derive a *single* 256-bit (32 bytes) master key. If you need additional cryptographic keys, you should find a way to derive them from the master key. In this assignment, you will use the password-based key-derivation function (PBKDF2) with 2,000,000 iterations of SHA-256 to derive your master key:[2]

  ```
  kdf = PBKDF2HMAC(algorithm = hashes.SHA256(), length = 32, salt = <YOUR SALT HERE>,
                   iterations = 2000000, backend = default_backend())
  key = kdf.derive(bytes(password, 'ascii'))
  ```

  The password manager is *not allowed* to include the master password (or any value that leaks information about the master password in its serialized representation). For instance, including

[1] Ensuring that the in-memory representation of the key-value store is encrypted is a good precaution (as would storing cryptographic keys in a protected segment or in a hardware security module). However, we emphasize that this is still *not* sufficient for security against an adversary that has access to memory. For instance, standard features and library functions in Python may leave data in memory that leak secrets—both on the call stack and in garbage-collected structures on the heap. While this assignment provides a valuable proof-of-concept illustration of many important aspects of cryptographic systems, it is still far from a complete picture, and you should *not* rely on the code you produce to be fully secure in all practical settings.

[2] At a high-level, PBKDF2 derives the key by applying SHA-256 to the (salted) password 2,000,000 times. Since we are using SHA-256 as our underlying hash function and the block-size of SHA-256 is 256-bits, the output of PBKDF2 is also 256-bits.

a hash of the master password is *not* secure. Because PBKDF2 is designed to be a "slow" hash function, you can call it *at most once* in your implementation.

- **Password salting:** When using PBKDF2 to derive your keys, you should always use a randomly-generated salt.[3] In this assignment, you should use a randomly-generated 128-bit salt (e.g., can be obtained by calling `os.urandom(16)`). The salt does *not* have to be secret, and can be stored in the clear in your serialized representation.

- **No external sources of randomness:** Since good sources of randomness are expensive and not always available, you cannot use any external sources of randomness other than for generating the salt for PBKDF2. This means that you *cannot* call methods like `AESGCM.generate_key` or `secrets.choice` anywhere in your implementation. All cryptographic keys and sources of randomness that you rely on should be (securely) derived from the master key output by PBKDF2.

- **Generating new passwords:** Your password manager must provide an interface for generating random alphanumeric passwords. The generated passwords should be computationally indistinguishable from a random alphanumeric password of the prescribed length. The same restrictions apply here: namely, you cannot use any external source of randomness to implement this feature.

- **No secrets in code:** You should not rely on any hard-coded secrets in your source code. You should assume that the adversary has complete knowledge of your source code.

3 Threat Model

When designing any system with security goals, it is important to specify a threat model. Specifically, we must define the power of the adversary, as well as the condition the adversary must satisfy in order to be considered to have "broken" the system. We define security via the following game-based definition (similar in flavor to notions like CPA security of PRF security):

The password manager plays the role of challenger and interacts with an adversary that is able to make a sequence of adaptive queries (as governed by the API of the password manager). The challenger's response to these queries depends on a bit $b \in \{0,1\}$ (as in the CPA security and PRF security games). In this case, we allow the adversary to make the following queries:

1. **Insertion query:** The adversary specifies a triple $\langle \text{domain}, \text{password}_0, \text{password}_1 \rangle$. The challenger adds the domain-password pair $\langle \text{domain}, \text{password}_b \rangle$ to the database.

2. **Retrieve query:** The adversary specifies a domain and the challenger replies to the adversary with the associated password in the password manager.[4]

3. **Remove query:** The adversary specifies a key (domain) that the challenger must remove from the password database.

[3] Using a random salt of sufficient length protects against common preprocessing attacks such as an offline dictionary attacks or rainbow tables.

[4] This models the setting where the adversary is able to cause the client to attempt to authenticate to a malicious server (e.g., through phishing, control hijacking, etc.), and is thus able to learn the password associated with a particular (malicious) domain in the client's password manager.

4. **Serialize query:** The adversary requests the challenger to serialize the current contents of the password manager. The adversary then provides the challenger a new string which the challenger immediately deserializes and uses the result as the new state of the password manager. Note that if the password manager ever ends up in an inconsistent state, then the challenger will respond to all subsequent queries with a special symbol $\perp$.

As in the PRF and CPA games, we say that the adversary wins the game if its probability of outputting 1 (for its guess of the bit b) differs by a non-negligible amount when $b = 0$ and when $b = 1$. Unlike the PRF and CPA games, however, we need an additional restriction for our security definition here. In particular, we will only allow adversaries whose queries are "admissible" in the following sense:

- Whenever the adversary makes a retrieve query on a domain d, its *last* insertion query adding passwords for a domain d, must have the property that $\text{password}_0 = \text{password}_1$.

Observe that without this restriction, the adversary could trivially win the game. Namely, the adversary can make an insertion query on a domain d with distinct passwords password_0 and password_1, and then make a retrieve query for domain d to learn password_b (and correspondingly, the challenger's bit b).

This security definition captures the fact that even if the adversary is able to exert substantial control over the contents of the password database—and even if it controls some malicious remote servers—it still is unable to learn anything about the passwords in the database for any *other* servers (i.e., we have semantic security for all domains d for which the adversary did not make a retrieve query).

For this project, you will not be required to give a formal proof that your system fulfills the strong security definition we have just stated, but such a proof should indeed be possible. We note here that this definition immediately precludes a number of attacks such as *swap attacks* and *rollback attacks*:

- **Swap attack:** In a swap attack, the adversary interchanges the values corresponding to different keys. For instance, the adversary might switch the entries for `www.google.com` and `www.evil.com`. Then, when the user (for whatever reason) tries to authenticate to `www.evil.com`, the user inadvertently provides its credentials for `www.google.com`.

- **Rollback attack:** In a rollback attack, the adversary can replace a record with a previous version of the record. For example, suppose the adversary was able to retrieve the key-value store shown above. At some later time, the user changes their password for `www.google.com` to `google_pwd`, which would update the value for `www.google.com` in the key-value store. In a rollback attack, the adversary replaces this updated record with the previous record for `www.google.com`.

  Observe first that authenticated encryption by itself does not protect against this attack. In your implementation, you should compute a SHA-256 hash of the *serialized* contents of the password manager. You can assume this hash value can be saved to a trusted storage medium (inaccessible to the adversary) such as an external flash drive. Whenever you load the password manager from disk, you should verify that the hash is valid. This way, you can be assured that the contents of the key-value store have not been tampered with.

In this assignment, you must implement some mechanism to defend against swap attacks together with the above method to defend against rollback attacks. Depending on your design, your defense against rollback attacks might also protect against the swap attacks described earlier. However, you *must still implement* an explicit defense against swap attacks. In other words, the defenses you develop must work *independently* of one another. Even if a SHA-256 hash is *not* provided from trusted storage, your scheme must be secure against an adversary that swaps two records.

4 API description

Here are descriptions of the functions you will need to implement. For each function, we also prescribe the run-time your solution must achieve (as a function of the number of entries n in the password database). We will assume that the input values (domain names and passwords) are of length $O(1)$, and regard each operation on a dictionary as a constant-time operation. Of course, if your solution is asymptotically more efficient than what we prescribe, that is acceptable. The starter code we provide you contains a basic *insecure* implementation that satisfies all of the functionality requirements.

4.1 `PasswordManager.__init__(password, data = None, checksum = None)`

- **Inputs:**
  - `password` (`string`): master password (an ASCII string) for the password manager
  - `data` (`string`): hex-encoded serialization of the password manager; defaults to `None`
  - `checksum` (`string`): hex-encoded SHA-256 checksum of the password manager for rollback protection; defaults to `None`
- **Raises**: `ValueError` if the provided data could not be deserialized properly (due to tampering, incorrect password, or if provided, an incorrect checksum)
- **Running time:** $O(n)$

This is the constructor for the password manager. If `data` is not provided, then this method should initialize an empty password manager with the provided `password` as the master password. Otherwise, it should load the password manager from `data`. In addition, if the `checksum` is provided, the password manager should additionally validate the contents of the password manager against the checksum. If the provided `data` is malformed, the password is incorrect, or the checksum (if provided) is invalid, this method *must* raise a `ValueError`.

4.2 `PasswordManager.dump()`

- **Inputs:** None
- **Return:** `data` (`string`) and `checksum` (`string`)
  - `data` (`string`): hex-encoded representation of the password manager that can subsequently be loaded to initialize the password manager (via `PasswordManager.__init__(...)`)
  - `checksum` (`string`): hex-encoded hash of the contents of the serialized representation
- **Running time:** $O(n)$

This method should create a hex-encoded serialization of the contents of the password manager, such that it may be loaded back into memory via a subsequent call to `PasswordManager.__init__(...)`. It should additionally output a SHA-256 hash of the serialized contents (for rollback protection).

4.3 `PasswordManager.get(domain)`

- **Inputs:**
  - `domain` (`string`): domain (an ASCII string) to fetch
- **Return:** `password` (`string`): password associated with `domain` and `None` if not present
- **Running time:** $O(1)$

If the requested domain is in the password manager, then this method should return the password associated with the domain. If the requested domain is not in the password manager, then this method should return `None`.

4.4 `PasswordManager.set(domain, password)`

- **Inputs:**
  - `domain` (`string`): domain (an ASCII string) to add
  - `password`(`string`): password (an ASCII string) associated with the domain to store in the password manager
- **Return:** None
- **Raises**: `ValueError` if the provided password exceeds the maximum length (64 bytes)
- **Running time:** $O(1)$

This method should insert the domain together with its associated password into the password manager. If the domain is already in the password manager, this method will update its value. Otherwise, it will create a new entry. If `password` is more than 64 bytes, this method should abort with a `ValueError`.

4.5 `PasswordManager.remove(domain)`

- **Inputs:**
  - `domain` (`string`): domain (an ASCII string) to remove
- **Return:** `success` (`bool`): whether the domain was found in the password manager or not
- **Running time:** $O(1)$

Removes the target domain from the password manager. If the requested domain is found, then this method should remove it and return `True`. If the domain is not found, return `False`.

4.6 `PasswordManager.generate_new(domain, desired_len)`

- **Inputs:**
  - `domain` (`string`): domain (an ASCII string) for which to generate a new password
  - `desired_len` (`string`): length of desired password
- **Return:** `password` (`string`): the newly-generated password
- **Raises:** `ValueError`: if the desired length exceeds the maximum length (64 bytes) or if the domain already exists in the password manager
- **Running time:** $O(\ell)$**, where** $\ell =$ `desired_len`

Generates a random alphanumeric password (i.e., every character is either a lowercase letter, an uppercase letter, or a digit) of the desired length. Each invocation of this method should yield a password that is (computationally) indistinguishable from a uniformly random password of the given length (i.e., running `generate_new` for the *same* domain and target length should return two different passwords). This requirement does *not* have to hold if the state of the password manager is reset or rolled back. The newly-generated password is then added to the password manager (if there does not currently exist a password associated with the requested domain). This method should raise a `ValueError` if the requested domain already exists in the password manager or if the desired length exceeds the maximum password length (64 bytes). **Hint:** If you sample $x \xleftarrow{R} \{0,\ldots,2^{128}-1\}$ and compute x mod 62, the result is sufficiently close to the distribution of $x \xleftarrow{R} \{0,\ldots,61\}$.

5 Additional Hints, Notes, and Requirements

- The starter code (`password_manager.py`) contains a fully-functional password manager that illustrates the basic functionality requirements. We include a basic testing script (`main.py`) that exercises some of the basic functionalities of the password manager.[5] Please note that this script does *not* cover all of the properties we will test during grading, and in particular, does not capture the security requirements. You are encouraged to design additional test cases to evaluate the correctness and security of your implementation.
- You cannot change the signatures of the methods we provide. If your implementation does not work with our provided `main.py` script, then you will not receive any credit for the assignment. That said, you are welcome to (and encouraged) to add additional helper methods in your implementation.
- Your design and implementation must satisfy the requirements from Section 2 as well as be provably secure under our threat model in Section 3. While we do not require a formal security proof in your submission, you should be prepared to provide one if requested.
- For serialization and deserialization of basic Python data structures (including dictionaries, lists, strings, byte-arrays, etc.), you can use the `pickle` library:
  - `ser_data = pickle.dumps(data).hex()` will output a hex-encoded serialization of `data`.

[5] The starter implementation was tested with Python 3.

  - `data = pickle.loads(bytes.fromhex(ser_data))` can be used to deserialize the data.

- To obtain a byte-array (needed for cryptographic methods) from an ASCII-encoded string, use `bytes(message, 'ascii')`. To convert back from a byte-array to an ASCII-encoded string, use `message.decode('ascii')`.

- One way to convert integers into byte arrays is to use the `to_bytes` method. For example, if you want an 8-byte array that represents the number n, you can write `(n).to_bytes(8, 'little')`.

- For this assignment, you may only make the following assumptions (and *nothing* more):

  - AES is a secure PRP.
  - AES-GCM is an authenticated encryption with associated data (AEAD) scheme.
  - HMAC (with SHA-256) is both a secure MAC and a secure PRF (on a variable-size domain).
  - SHA-256 is a collision-resistant hash function.
  - PBKDF2 (with SHA-256) is an ideal hash function (i.e., a "random oracle"); you can only invoke PBKDF2 **once** on the master password and only to generate a single 256-bit key (see Section 2).

  You should only need to rely on these primitives in your implementation. While you are free to use other algorithms that build upon these basic primitives (e.g., HKDF), you will be responsible for figuring out how to use them in a way that achieves the required security properties. We will not answer any questions about proper usage of any primitive that is not in the list above. Our reference implementation only uses the primitives listed above.

- As stated in Section 2, the only source of external randomness you are allowed to use is for the salt in PBKDF2. You can only call PBKDF2 *once* in your implementation, and you can only use it to generate a 256-bit (32-byte) master key. If you need additional cryptographic keys, you must securely derive them from the master key.

- Your implementation can only make use of standard Python modules and the `cryptography` library.

- Carefully think through your design *before* starting on your implementation. The number of lines of code you need to write should be modest. As a point of reference, our reference solution is under 250 lines of Python code (including the 100 lines in the starter file):

```
diff -y --suppress-common-lines base/password_manager.py password_manager.py  | wc -l
  142
```

6 Short-Answer Questions

In addition to your implementation, please include *short* responses (e.g., 1-5 sentences) to the following questions regarding your design and implementation. You do not need to give formal proofs, but you should be precise and include important details in your responses.

1. Suppose an adversary is able to perform a swap attack (as described in Section 3). Show how such an adversary can win the password manager security game. Note that you must say why the adversary you construct is admissible for the password manager security game.

2. Suppose an adversary is able to perform a rollback attack (as described in Section 3). Show how such an adversary can win the password manager security game. As before, you should say why the adversary you construct is admissible.

3. There are at least two methods in the provided API (other than the constructor) that (should) rely on a source of randomness. Identify two of these methods and briefly state how your implementation generates the randomness needed for these two methods.

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

5. Briefly describe your method for preventing swap attacks (Section 3). Provide an argument for why the attack is prevented in your scheme.

6. Suppose instead that we only had 16 bits of trusted storage. How would you modify the defense against rollback attacks to remain secure even in this setting? You may assume that the user will make at most 2^{16} updates to the password manager.

7. How much time did you spend on this assignment? This is for calibration purposes, and the response you provide will not affect your score.

8. **Optional feedback.** Please let us know if you have any feedback on the design of this assignment or on the course in general.

Please submit your responses as a PDF file `answers.pdf` with your submission.

7 Submission Instructions

To submit your assignment, upload the following two files to Gradescope:

- `password_manager.py`: this file contains your implementation of the password manager.
- `answers.pdf`: this file contains your answers to the short answer questions.

Do *not* submit any other files with your submission (e.g., do not submit your copy of `main.py`). You should make only **one** submission per group. Indicate all group members on Gradescope.