COMP 537: Cryptography **Fall 2025**

Programming Assignment 1: Private Notes

In this assignment, you will be implementing the back-end for a private note-taking 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*. The starting implementation we provide satisfies all of the functionality requirements, but none of the security requirements.

Acknowledgments. This assignment is adapted from a similar assignment from Stanford's CS255 course by Prof. Dan Boneh and UVA's CS6222 course by Prof. David Wu.

1 Overview

Note-taking applications are a very helpful tool for quickly jotting things down, keeping track of lists, and drafting messages or documents. Hosting the storage of notes in the cloud provides the added benefit that users can access their notes from multiples devices without losing their notes if a device is lost or broken. Unfortunately, this also means that the note-taking service provider has access to all of a user's notes. This is especially bad as note-taking tools are frequently used for short-term storage of sensitive information.

In this assignment, you will show that it is possible to enjoy the benefits of a note-taking application with cloud storage *without* compromising user privacy along the way. In particular, you will design and implement the back-end for a privacy-preserving note taking application. 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 Private Note Taking

You will implement the API for a back-end implementation of a note taking application. The note taking application will internally maintain a key-value store that maps note titles (keys) to notes (values). For our purposes, note titles will always be unique. The API will support serialization and deserialization methods for loading and writing the contents of the notes to disk (or cloud storage), as well as methods for adding, fetching, and removing notes. 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 notes and titles are ASCII strings.
- **Title encoding:** We want to hide the note titles while still enabling efficient look-ups. To support this, instead of using the title x itself as the key in the key-value store, you will use $\mathsf{HMAC}(k, x)$, where k is an HMAC key.

[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.

- **Note storage:** The notes in the key-value store should be encrypted using an authenticated encryption scheme. Since there can be a large number of notes stored, each note 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 all the notes for each lookup).

- **Hiding note length:** The application should not leak any information about the length of the notes or titles. To make this feasible, you may assume that the maximum length of any note is 2KB.

- **Key derivation:** The note-taking application itself is protected by a password. When the user initializes the application or loads the notes from disk, they must provide the password. The password should be used to derive a *single* 256-bit (32 byte) source key. If you need additional cryptographic keys, you should find a way to derive them from the source 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 source key:[2]

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

  The application is *not allowed* to include the password in its source code (or any value that leaks information about the password in its serialized representation). For instance, including a hash of the 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 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 source key output by PBKDF2.

- **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).

---

[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.

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

The note taking application 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 note taking application). 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 $\langle \text{title}, \text{note}_0, \text{note}_1 \rangle$. The challenger adds the title-note pair $\langle \text{title}, \text{note}_b \rangle$ to the database.

2. **Retrieve query:** The adversary specifies a title and the challenger replies to the adversary with the associated note.

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

4. **Serialize query:** The adversary requests the challenger to serialize the current contents of the note taking application. The adversary then provides the challenger a new string which the challenger immediately deserializes and uses the result as the new state of the note taking application[4].

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 title t, its *last* insertion query adding notes for title t, must have the property that $\text{note}_0 = \text{note}_1$.

Observe that without this restriction, the adversary could trivially win the game. Namely, the adversary can make an insertion query on a title t with distinct notes note_0 and note_1, and then make a retrieve query for title t to learn note_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 note database and can even read the contents of some of them (e.g., if a user voluntary shares something they had written down), it still is unable to learn anything about any other notes in the database (i.e., we have semantic security for all titles t 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 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 `Groceries` and `Secret Investigations`. Then, when the user tries to access or share a grocery list, they share sensitive secret information instead.

- **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 note for `Secret Investigations` to include a new important secret value, which would update the value for `Secret Investigations` in the

[4] This models a malicious cloud provider who may attempt to tamper with a user's notes

key-value store. In a rollback attack, the adversary replaces this updated record with the previous record for `Secret Investigations`.

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 notes. 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 application 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 (or performs other non-rollback attacks on your scheme).

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 notes database). We will assume that the input values (titles and notes) 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 `Notes.__init__(password, data = None, checksum = None)`

- **Inputs:**
  - `password` (`string`): password (an ASCII string) for the note taking application.
  - `data` (`string`): hex-encoded serialization of the note database; defaults to `None`
  - `checksum` (`string`): hex-encoded SHA-256 checksum of the notes database 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 note database. If `data` is not provided, then this method should initialize an empty note database with the provided `password` as the password. Otherwise, it should load the notes from `data`. In addition, if the `checksum` is provided, the application should additionally validate the contents of the notes database 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`.

If this method is called with the wrong password, i.e., not the password used to initialize the provided `data`, your code must return a `ValueError`, and no other queries can be performed unless the client calls init successfully. It is incorrect for your application to pretend like nothing is wrong when the wrong password is provided and only fail to answer queries later.

4.2 `Notes.dump()`

- **Inputs:** None
- **Return:** `data` (`string`) and `checksum` (`string`)
  - `data` (`string`): hex-encoded representation of the notes that can subsequently be loaded to initialize the application (via `Notes.__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 notes database, such that it may be loaded back into memory via a subsequent call to `Notes.__init__(...)`. It should additionally output a SHA-256 hash of the serialized contents (for rollback protection).

4.3 `Notes.get(title)`

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

If the requested title is in the notes database, then this method should return the note associated with the title. If the requested title is not in the database, then this method should return `None`.

4.4 `Notes.set(title, note)`

- **Inputs:**
  - `title` (`string`): title (an ASCII string) to add
  - `note(string)`: note (an ASCII string) associated with the title to store in the database
- **Return:** None
- **Raises**: `ValueError` if the provided note exceeds the maximum length (2KB)
- **Running time:** $O(1)$

This method should insert the title together with its associated note into the database. If the title is already in the database, this method will update its value. Otherwise, it will create a new entry. If `note` is more than 2KB, this method should abort with a `ValueError`.

4.5 `Notes.remove(title)`

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

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

4.6 API Usage

The API is designed to be called by a front-end application that provides a user interface for note retrieval. Whenever Alice requests a note by name, creates/edits a new note, or deletes a note, the API will make calls to `get`, `set`, or `remove`, respectively.

When Alice closes the application, or when there is some application-dependent "save" event, the `dump` function will be called. The resulting `data` will be saved to disk by the front-end application, and the `checksum`, if used by the calling application, will be stored separately in a protected place.

When Alice opens the front-end application or when there is some application-dependent "load" event, `init` will be called. The `data` used will be loaded from disk and Alice will provide the password. If the application is using the checksum, it will load `checksum` from the protected storage. We assume that neither Alice nor any attacker can modfiy the contents of the protected storage at any time – only the honest application can update this storage. If Alice is creating a new account in the application, `init` will be called with no `data` and the password provided by Alice will become the password required to access the resulting data again in the future.

5 Additional Hints, Notes, and Requirements

- The starter code (`private_notes.py`) contains a fully-functional note taking application that illustrates the basic functionality requirements. We include a basic testing script (`main.py`) that exercises some of the basic functionalities. [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.

[5] The starter implementation was tested with Python 3.10.12.

- 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`.
  - `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 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, you will be responsible for figuring out how to use them in a way that achieves the required security properties. 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) source key. If you need additional cryptographic keys, you must securely derive them from the source key.
- Your implementation can only make use of standard Python modules not related to cryptography and the `cryptography` library. All cryptographic operations must use this library. You will be using the library's "hazardous materials" layer.
- 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 file is under 200 lines of Python code, and the diff with the starter code is even smaller:

```
$diff -y --suppress-common-lines base/private_notes.py private_notes.py  | wc -l
  126
```

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 note taking security game. Note that you must say why the adversary you construct is admissible for the 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 security game. As before, you should say why the adversary you construct is admissible.

3. Briefly describe your method for checking passwords.

4. Briefly describe your method for preventing the adversary from learning information about the lengths of the notes stored in your note-taking application.

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

6. **Optional feedback.** How much time did you spend on this assignment? Did you find it too easy/hard or just right?

7. **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:

- `private_notes.py`: this file contains your implementation of the private notes system.
- `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`). If submitting with a partner, please have one partner submit for both people, and please include your names in a comment at the top of the python file.

Grading: The short answer questions are each worth 4 points. There are 16 points of basic automated tests and 14 points of code review for security.