1. An adversary can win the security game by exploiting a swap attack in the following manner: the adversary inserts two indistinguishable notes (note0 and note1) for different titles, such as "TitleA" and "TitleB", which satisfies the game's admissibility condition. After the notes are added, the adversary serializes the note data, swaps the encrypted entries for "TitleA" and "TitleB" in the serialized form, and then provides the manipulated data for deserialization. When the adversary later retrieves the note for "TitleA" but receives the note meant for "TitleB", they can distinguish the bit b chosen by the challenger. The adversary is admissible in this scenario because they issue only valid queries and operate on titles where the inserted notes were identical (note0 = note1).

2. An adversary can win the security game by performing a rollback attack as follows: After inserting a note and serializing the data, the adversary rolls back the state of the notes database to an earlier version where a different note was stored for the same title. If the application accepts the older data and does not detect the rollback, the adversary can retrieve the previous note and distinguish the bit b used by the challenger based on the returned content. This adversary is admissible because they are only issuing legitimate queries to the application while exploiting the vulnerability in the rollback detection mechanism.

3. My method for checking passwords uses the PBKDF2 key derivation function to generate a source key from the provided password and a stored salt. If loading existing data, the derived key is used to decrypt a sample note from the database. If decryption fails, it indicates that the password is incorrect, and a 'ValueError' is raised. This ensures that incorrect passwords are caught early before any further operations are allowed.

4. In my code, the method for preventing an adversary from learning the lengths of notes is implemented by padding each note to a fixed maximum length (2048 bytes) before encryption. This is done in the 'set' method, where the actual length of the note is calculated, and if the note is shorter than the maximum length, it is padded with null bytes ('00'). This padded note is then encrypted. The original length of the note is stored in the key-value store ('self.kvs') along with the ciphertext, but the ciphertext itself always corresponds to the fixed-size padded note. When retrieving the note via the 'get' method, the original length is used to correctly remove the padding after decryption. This ensures that an adversary cannot infer the true length of any note by simply analyzing the size of the ciphertext.

5. In my implementation, swap attacks are prevented by binding each note's content to its title through the use of HMAC and incorporating a nonce counter. When a note is set or retrieved, an HMAC is computed on the title (`_compute_title_hmac`), which serves as a unique key for the note in the key-value store. Additionally, each encryption uses a derived nonce that is based on both the title's HMAC and a counter (`_derive_nonce`). This counter increments with each new note stored, ensuring that the encryption is unique even if the same note is stored under the same title. Moreover, the title HMAC is included as associated data in the encryption process. If an adversary attempts to swap encrypted notes between titles, the decryption will fail due to a mismatch in the associated data, triggering an 'InvalidTag' error. The combination of the title binding and unique nonce ensures that swap attacks are effectively prevented.