Cryptography
Project 1: Password Manager

April 18, 2024

Nguyen Duy An 20214943, Duong Hong Nam 20214967, Vu Hoang Ngoc 20210646

---

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

> **Ans:** We ensure that all passwords are the same length by padding them to 80 bytes. We add the sequence "100000..." at the end of each password. To extract the original password from a padded one, we look for the first occurrence of "1" from the end of the string. This method maintains uniform password lengths, safeguarding against potential attackers learning about the actual lengths of the passwords in your password manager.

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

> **Ans:** To add a record (key, value) in the KVS, begin by generating a random initialization vector (iv), then encrypt the value using AES-GCM, resulting in the ciphertext. Compute the tag using HMAC(key, ciphertext), and store key: [iv, ciphertext, tag] in the KVS. This approach protects against swap attacks. When retrieving a record for a key in the KVS, the associated value [iv, ciphertext, tag] is accessed. If the HMAC(key, ciphertext) doesn't match the tag (indicating the key has been tampered with), authentication fails, and no plaintext is returned, thereby preventing swap attacks.

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. ()

Ans: Yes, it is essential to assume the existence of a trusted location to protect against rollback attacks. Without this assumption, an attacker could know the prior SHA-256 hash value. They could then swap out updated records with older ones in the KVS and replace the current hash value with the previous SHA-256 hash value. This would result in a rollback attack. Therefore, having a trusted location is crucial.

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?. ()

Ans: For the Carter-Wegman MAC, you must select a seed s from the seed space and use it to verify with the stored HMAC keys, domain name, and seed s to obtain a tag t. It is necessary to check whether tag t is a key in the KVS. To make t a key in the KVS, you might need to repeatedly choose a seed from the seed space multiple times. Mathematically, after attempting this process $|S|$ times (where $|S|$ is the size of the seed space), you would expect to find a successful lookup at least once. This process incurs a performance penalty, as it takes $|S|$ tries to verify, similar to how HMAC works.

Generally, for a randomized MAC, the distribution $P(Y \mid m)$ of output Y depends on the message m used. If $|Y|$ represents the size of the output space $|Y|$, then to verify, you must calculate MAC(m) and check if the result is in the KVS. Given that MAC(m) is randomized (assuming a uniform distribution), you expect a successful lookup at least once after $|Y|$ trials. This also introduces a performance penalty, as it takes $|Y|$ attempts to verify, similar to HMAC verification.

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 $\lfloor \log_2(k) \rfloor$ (that is, if k_1 and k_2 are such that $\lfloor \log_2(k_1) \rfloor = \lfloor \log_2(k_2) \rfloor$, the attacker should not be able to distinguish between a case where the true number of records is k_1 and another case where the true number of records is k_2). ()

Ans: Combine key of HMAC (domain name) with its value. Create more records of the same length as the combined record that is $2^{\log_2(k)} - k$ records. Set all the bits in these additional records to 0. Then, merge each record with another record. If there are any unmerged records, pad them with 0s (the number of 0s equals the length of the record). Repeat this process until there are $\log_2(k)$ merged records with the same length. Since $\log_2(k_1) = \log_2(k_2)$, the number of final concatenated records is the same for both k_1 and k_2 records. Consequently, an attacker cannot distinguish between a scenario where the true number of records is k_1 and another scenario where the true number of records is k_2, provided that $\log_2(k_1) = \log_2(k_2)$.

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. ()

Ans: We can use the Menezes–Qu–Vanstone (MQV) protocol to establish a secure channel for communication. MQV, an authenticated key agreement protocol based on Diffie–Hellman, offers protection against active attackers, which the original Diffie–Hellman scheme does not. For more specific, refer to the MQV Wikipedia page.

Using this approach, both Alice and Bob incorporate a new feature: shared domains, which store the MAC (message authentication code) of the shared domain name. Suppose Bob has set up his keychain. When Alice wants to add a record to her keychain, she runs Project I and sends (domain name, tagA) to Bob via MQV. Bob decrypts the message to retrieve the domain name, calculates its MAC (tagB), and checks if it exists in his keys. If it does, he adds the pair tagB: tagA to the shared domains property and sends back "Yes" and tagB to Alice through MQV. Otherwise, Bob responds with "No."

If Alice receives "Yes," she adds tagA: tagB in her shared domains property. When Alice wants to change the password, she uses Project 1 to update her own KVS, looks up the corresponding tag of Bob in the shared domains property, and sends ("Change", tag, new value) to Bob under MQV. If this tag is in Bob's shared domains' keys, Bob then updates his keychain using this (tag, new value). The process is the same when Bob changes passwords for the shared domain name. Upon receiving a "Yes," Alice adds the tagA: tagB pair to her shared domains property. When Alice needs to change a password, she uses Project I to update her own KVS, checks Bob's corresponding tag in the shared domains property, and sends ("Change", tag, new value) to Bob using MQV. If the tag exists in Bob's shared domains, he updates his keychain with (tag, new value). The process is the same when Bob changes passwords for a shared domain name.

This approach ensures that Alice and Bob cannot access each other's exclusive domain names.