CS255: Intro. to Cryptography **Winter 2023**

Project #1 Short-answer Questions

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

Answer: I set the length of the passwords after padding as 80 bytes. I append "100000.." at the end of the original passwords. To get the raw password from the padded password, I search the first "1" from the end. In this way, the padded passwords are all 80 bytes, which can prevent the adversary from learning information about the lengths of the passwords stored in your password manager.

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

Answer: To set a record (key, value) in the KVS, first get a random iv, use AESGCM to encrypt the value and get the ciphertext, and then use HMAC(key, ciphertext) to get the tag. Then set {key:[iv, ciphertext, tag]} in the KVS. In this way, thw swap attack is prevented. When retrieving the record of a key in the KVS, the value [iv, ciphertext, tag] corresponding to the key is available. If HMAC(key,ciphertext) is not equal to the tag(since this key is not equal to the key before swapping), the authentication fails and no plaintext would be returned. In this way, swap attacks are prevented.

Problem 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.

Answer: Yes, it is necessary to assume that such a trusted location exists, in order to defend against rollback attacks. Otherwise, an adversary may know the previous SHA-256 hash value. He can replace the updated record with the previous record of some entries in the KVS and at the same time replace the updated hash value with the previous SHA-256 hash value. This leads to a rollback attack. Thus, it is necessary that such a trusted location exists.

Problem 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?

Answer: For Carter-Wegman MAC, we need to choose a seed s from the seed space and run the verification with the stored HMACkeys, domain name, and seed s to get a tag t. We need to check whether the tag t is a key of the KVS or not. To make that t is a key in the KVS, we need to try to choose a seed from seed space multiple times. Mathematically, after we try this process $|S|$(which is the seed space) times, the expectation of a successful lookup is 1. Thus, there is a performance penalty involved. It takes $|S|$ times to verify as HMAC does, where $|S|$ is the seed space.
Generally, for a randomized MAC, the output Y distribution $P(Y|m)$ is determined by the message m used. Suppose $|Y|$ is the space of the output Y. To verify, we need to calculate MAC(m) and find if the result is in the KVS or not. Since MAC(m) is a randomized distribution(assume it is uniform), the expectation of a successful lookup is 1 after $|Y|$ times trials. Thus, there is a performance penalty involved. It takes $|Y|$ times to verify as HMAC does, where $|Y|$ is the output space of $P(Y|m)$.

Problem 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).

Answer: Concatenate the 256-bit key(HMAC(domain name)) and corresponding value. Add $2^{\lceil logk \rceil} - k$ records with the same length of the aforementioned concatenated record and set all bits as 0. Then, merge each record with another record. If there is an unmerged record. Pad the record with 0.(The number of 0 is the same of the length of the record) Do this process iteratively until there are $\lfloor logk \rfloor$ merged records with the same length. Since $\lfloor logk_1 \rfloor = \lfloor logk_2 \rfloor$, after the above operation, the number of final concatenated records is the same for k_1 and k_2 records. This means the attacker can 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 if $\lfloor logk_1 \rfloor = \lfloor logk_2 \rfloor$.

Problem 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.

Answer: Use Menezes–Qu–Vanstone(MQV) to establish a channel between Alice and Bob. MQV is an authenticated protocol for key agreement based on the Diffie–Hellman scheme. It provides protection against an active attacker, which Diffie–Hellman scheme cannot. (I would not like to copy the steps of this algorithm since they are in the Wikipedia link.) Using this algorithm, Alice and Bob both add a new property, shared domains, which is used to store the MAC of the shared domain name. For simplicity, suppose Bob has established his keychain. For each record that Alice wants to add in her keychain, she does the Project 1 program and sends the (domain name, tag_A) to Bob under MQV. Bob decrypts and get the domain name. Then he calculates MAC of this domain name as tag_B and checks whether is in his keys or not. If it is in his keys, he adds tag_B: tag_A in the shared domains property and sends ("Yes",tag_B) back to Alice under MQV. Otherwise, Bob sends ("No") back to Alice. If Alice receives "yes", she adds tag_A: tag_B in her shared domains property. When Alice wants to changes the password, she uses project 1 to update her own KVS, look 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. And it's obvious that they cannot access the domain names that are exclusive to themselves.