CS 255
Short Answers Project Two

Atticus Christensen, Andrew Moreland

Febuary 22, 2014

1. Because we pad the password before encrypting it, it is impossible to find the length of the password from the result, as information about what the padded password (notably the length) cannot be recovered.

2. We defend against swap attacks by storing with the password for the domain the HMAC of the concatenation of the domain, password, and length hashed against our master key k.

   For each domain, we concatenate the hmac of the domain and the encrypted password and then HMAC with our derived HMAC key k_{HMAC}.

   If the adversary swapped the entries for www.google.com and www.evil.com, then when we go to look up the password for www.google.com, we concatenate the HMAC of the domain and encrypted password for the entry and compare it with the saved HMAC. If the domain name were tampered with, we would notice that the HMACs do not agree and throw an exception. The attacker cannot efficiently construct a false entry for www.google.com because HMAC is existentially unforgeable if it uses a secret key.

3. There are two cases we need to consider if we do not have a trusted storage location. In one case, we just blindly trust what we are given. Obviously this is vulnerable to a rollback attack. In the other case, the password manager must perform some computation on the given state in order to verify that it is valid. However, since the adversary controls all observable state, the adversary can always construct an environment for the password manager which causes it to believe an "old" valid state is still valid. In particular, if the password manager were able to reject a previously-valid state, then some state of the machine must

have changed. Since the attacker controls everything, the attacker can replicate the old state identically and fool the password manager.

Therefore, we need to store some information in order to prevent rollback attacks.

4. It is possible to construct a secure MAC which yields an insecure implementation of our password manager.

   We define a MAC: take the key k and first byte of the message m, m_1, to $m_1 \| \text{HMAC}(k, m)$. This is still a secure MAC. Note that if we were able to construct a forgery of some message m', then necessarily we would have to be able to construct a forgery of $\text{HMAC}(k, m')$. By assumption, this is impossible.

   However, this MAC leaks the first byte of the domain names in the keychain, so it's not secure.

5. One way to avoid leaking information about the size is to "pad" the password managers database and then encrypt the output. Effectively, during a dump operation we would add dummy passwords in order to round the total number of passwords up to a constant number significantly larger than the number of stored passwords, say 1000, or the next largest power of 2. During the load operation we would remove these passwords.

   Barring timing attacks, this would only leak vague information about the number of passwords stored. Depending on the storage requirements that the user is willing to accept, the padding could be made large enough that the total size of the dump would be almost always constant so that no information would leak.

Extra credit: We implement a solution to the requirements posed by the extra credit portion of the assignment. Instead of using a SHA256 hash as a checksum, we instead use a counter which is incremented on each dump operation. Note that this has no effect on the swap-attack defense.

In order to use this counter to prevent rollback attacks, we store some extra information in our serialized output. In particular, we compute the HMAC (using a key derived from the password that is reserved for this purpose) of a deterministic serialization of the *priv.data* datastructure concatenated with the counter, and then store this value v as part of the JSON output of dump. When we attempt to load from a serialized representation provided by the adversary, we de-serialize the data storage that they provide, and them recompute the

deterministic serialization, concatenate the counter they give us, and then re-HMAC using the same key derived from the password. If this output does not match v, then we throw an exception.

This is secure because HMAC is existentially unforgeable. For the adversary to execute a rollback attack that defeated our verification scheme, they would have to take an old version of the database and construct a message which, when concatenated with the correct counter, appears to be signed by the challenger. This is precisely what the existential unforgeability property of HMAC prevents, so our scheme is secure.er