Grant Ho grantho@stanford
Veni Johanna veni@stanford

CS 255 Project 2 Short Answer

1. How does our method prevent the adversary from learning information about the password lengths?

We pad all encrypted value entries to be the same length (320 bytes); from here on, we'll value the value entries "password blobs". Each password blob contains the password, HMAC of domain, and padding. Since passwords are a max of 64 bytes and HMAC(domain) = 256 bytes, the maximum "real" information in a password blob is 64 + 256 = 320 bytes. Thus, if we pad all password blobs to 320 bytes before authenticated encryption, we know that the final encrypted password blobs will all have the same length.

2. How does our method prevent swap attacks?

We prevent swap attacks by binding each password to its domain in the password blobs. In our KVS, the password blob contains both the password and the HMAC of the domain name; these entries are then encrypted under authenticated encryption. When getting a password from a keychain.get call, we first make sure the password blob successfully decrypts; if this authenticated decryption succeeds, we then make sure the HMAC(domain) in the password blob matches HMAC(domain_from_get) before returning the password.

It is impossible for an attacker to succeed in a swap attack without breaking the security of authenticated encryption and HMAC. Suppose an attacker wants to swap attack against google.com; thus he replaces google.com's encrypted password blob with evil_domain's encrypted password blob. Because we verify that HMAC(keychain.get's domain) equals the HMAC(domain) in the KVS (i.e. HMAC(google.com)), the attacker must (1) somehow modify the encrypted password blob of evil_domain so that HMAC(evil_domain) is replaced with HMAC(google.com) or (2) find an evil_domain name such that HMAC(evil_domain) = HMAC(google.com). However, approach (1) means that the attacker needs to break the security of authenticated encryption (modifying an A.E. cipher-text such that the integrity check doesn't fail) and approach (2) means that the attacker needs to break the security of HMAC (finding two distinct messages that generate the same MAC). Thus, by the security of authenticated encryption and HMAC, our scheme is secure against swap attacks.

3. Why is it necessary to assume a trusted location exists for the SHA-256 hash to defend against rollback attacks?

Yes, in the proposed defense and our implementation, a trusted location must exist. Suppose we stored the trusted_data_check in an insecure/untrusted location. Then an attacker can simply modify the contents of our password manager dump (e.g. by replacing one password record with an old version), compute the new SHA-256 hash of this modified dump, and then replace the trusted_data_check with this new SHA hash. Since anyone can compute the SHA-256 hash of anything, this attack is completely within the attacker's power. Now, when we compute the SHA hash of our password manager during load, it will match the trusted_data_check that the attacker computed and we will not be able to detect that a rollback attack occurred.

4. Would the scheme still satisfy the desired security properties if we used a different MAC on the domain name?

It depends on the replacement MAC and in general, we are not guaranteed that the security properties are still satisfied. Consider the following secure MAC:

$$\textbf{\textit{GMAC}}(k, \text{domain\_name}) = \text{domain\_name} \,||\, \text{HMAC}(k, \text{domain\_name}).$$

GMAC is secure MAC because finding two messages that output the same GMAC tag requires finding two messages that output the same HMAC tag (since all GMAC tags use an HMAC output as its tag suffix). However,

GMAC clearly reveals the domain name for each password entry - which violates the security property that the domain names should be hidden.

5. **Describe an approach to reduce or completely eliminate the information leaked about the number of records in the password manager?**

Every time we set (add) a password entry, we can randomly decide to add a dummy record to the password manager (i.e. securely generate a random number from [0,1] and check if it's greater than 0.5 in keychain.set(). If it's greater than 0.5, then we generate a random string and a random password and add them to the password manager just like any other valid password record). Now, at best, the attacker knows the maximum number of records our password manager stores under this implementation, but the exact number of records is hidden because the current number of records in the password manager includes a random number of dummy records.