Programming Assignment 1 Winter 2015

CS 255: Intro to Cryptography

Prof. Dan Boneh Due **Feb. 13, 11:59pm**

1 Introduction

In many software systems today, the primary weakness often lies in the user's password. This is especially apparent in light of recent security breaches that have highlighted some of the weak passwords people commonly use (e.g., `123456` or `password`). It is very important, then, that users choose strong passwords (or "passphrases") to secure their accounts, but strong passwords can be long and unwieldy. Even more problematic, the user generally has many different services that use password authentication, and as a result, the user has to recall many different passwords.

One way for users to address this problem is to use a password manager, such as LastPass and KeePass. Password managers make it very convenient for users to use a unique, strong password for each service that requires password authentication. However, given the sensitivity of the data contained in the password manager, it takes considerable care to store the information securely.

In this assignment, you will be writing a secure and efficient password manager. In your implementation, you will make use of various cryptographic primitives we have discussed in class—notably, authenticated encryption and collision-resistant hash functions. Because it is ill-advised to implement your own primitives in cryptography, you should use an established library: in this case, the Stanford Javascript Crypto Library (SJCL). We will provide starter code that contains a basic template, which you will be able to fill in to satisfy the functionality and security properties described below.

2 Secure Password Manager

2.1 Implementation details

In general, a password manager application will store its password database on disk, protected by a strong master password; in addition, while it is in use, it may store an "unlocked" representation of the database in memory, from which it can provide the password for each desired domain. Instead of implementing a full standalone password manager application, for this project you will only be responsible for the core library. Thus, you will not need to implement the interactive front-end for interacting with the password manager, nor will you need to actually write the contents to disk. Instead, you will simulate these functionalities by providing features to serialize and deserialize your data structures to string representations, so that it would be easy to complete a full password manager application by writing these representations to disk.

Your password manager will keep its in-memory password data in a key-value store (KVS), represented by a Javascript object whose keys correspond to domain names, and whose values correspond

to passwords for each domain. For example, a sample password manager instance might store the following information:

Key	Value
`www.google.com`	`password`
`www.example.com`	`123456`
`www.amazon.com`	`6U)qAlOBy%3SZX$o`
`www.ebay.com`	`guest`

Naturally, writing this information to disk in the clear is not secure. In this assignment, you will need to preserve both the confidentiality and the integrity of the values in your KVS. In order to accommodate a potentially large number of entries in the password manager, you will encrypt and store each record individually in-memory. In other words, it is not appropriate to encrypt the entire KVS as a single blob. This way, you do not have to decrypt every entry in the password manager when fetching a single record.

We also do not want to leak any information about the domains the user has stored in the password manager. At the same time, we want to maintain the ability to search for the data corresponding to a specific domain. In this assignment, the KVS (Javascript object) storing the password data should have as its keys the HMAC of each domain name, rather than the actual domain name in the clear.[1] Then, to look up the data corresponding to domain name x, you first compute $\mathsf{HMAC}(k, x)$, where k is the secret key for HMAC, and check whether the result exists as a key in the key-value store.

If you simply encrypt each domain/password pair in the KVS directly, your implementation will probably leak some information about the lengths of the passwords. We will not consider such implementations secure; rather, your implementation must prevent an adversary from learning any information about the password lengths. (To make this feasible, you may assume that the maximum length of any password is 64 bytes.)

The password manager itself will be protected by a key derived from a master password. Thus, when the user creates a new password manager (also called a *keychain*), or loads the contents of the password manager from disk, s/he must provide this master password. The master password will be used to derive keys for the HMAC (for the KVS keys, i.e., the domain names) as well as for authenticated encryption (for the KVS values, i.e., the passwords). In general, you should use the password-based key derivation function, PBKDF2, to derive keys from passwords (see Section 2.3 for a more detailed discussion on PBKDF2). Note that a secure password manager is *not allowed* to include the master password (or any values that would leak information about it) in the serialized password database on disk. (Recall from Homework 1 that leaking even one bit of the key can compromise the security of a given scheme!)

Important note: In practice, there are many other considerations to take into account when designing an application like a secure password manager. For instance, as we will see below, we make no effort to thwart timing attacks or other side-channel attacks, and indeed this is extremely difficult using a language like Javascript that is so far removed from the underlying machine model. Along similar lines, it is always a good practice to erase secret keys in memory when the application

[1] Technically, you will need to use some serialized string representation of the HMAC value—for instance, hex or base64.

is finished using them, in case the memory is subsequently exposed to the attacker. However, this is likewise infeasible in Javascript, as many standard features and library functions (including SJCL) leave data in memory that may leak secrets—both on the call stack and in garbage-collected structures on the heap. Thus, while this project provides a valuable proof-of-concept illustration of many important aspects of crypto implementation, it is very far from a complete picture, and you should *not* rely on the code you produce to be secure in practical settings.[2]

2.2 Threat model

When designing any system with security goals, it is important to specify a threat model: i.e., we must precisely define the power of the adversary, as well as the condition the adversary must satisfy in order to be considered to have "broken" the system. Thus, we will now specify the threat model for our secure password manager, in the form of a security game (of the same flavor as the PRF or CPA games). In particular, just as the CPA game allows the adversary to specify messages of its choice, our definition will seem to give the adversary a great deal of power over the contents of the password database. It is important to remember that we must make such strong assumptions when attempting to show that a system is secure for general use, because we have no idea under what circumstances it may end up being deployed.

Our security game proceeds as follows. As usual, the password manager will play the role of the challenger—interacting with another implicit party, the disk storage—while the adversary will make a series of adaptive queries that determine the behavior of the system. Some of the adversary's queries may include a contingency for each of two "worlds"—as in worlds 0 and 1 in the CPA game—and, as in the CPA game, the "world bit" parameter, b, will determine which series of queries the challenger actually executes. Each query will take one of the following forms:

1. Specify values $\langle \text{domain}, \text{password}_0, \text{password}_1 \rangle$ to be added to the database. In world 0, the challenger must run the password manager algorithm to add the domain-password pair $\langle \text{domain}, \text{password}_0 \rangle$ to the database, while in world 1, the challenger must run the same algorithm, but on the pair $\langle \text{domain}, \text{password}_1 \rangle$.

2. Specify a key (domain) that the challenger must remove from the password database.

3. Specify that the challenger must serialize the database state to "disk", whereupon the adversary will receive the entire result of the serialization, and will be able to replace it with an alternative of its choice (which must then immediately be deserialized by the challenger, running the password manager algorithm).

4. Specify a key (domain) for authentication, at which point the challenger (again running the password manager algorithms) must send the adversary the password corresponding to that domain.[3]

As in the PRF and CPA games, we say that the adversary wins the game if its probability of outputting 1 (for its guess of the world bit b) differs by a nonnegligible amount between world 0

[2] For an additional perspective on the pitfalls of implementing crypto in languages like Javascript, see http://www.matasano.com/articles/javascript-cryptography/.

[3] This ability models the fact that in real life, the adversary may control the server running at the other endpoint of the client's query, and therefore may see the result of a client password manager's attempted authentication.

and world 1. Unlike the PRF and CPA games, however,[4] we will need an additional restriction for our security definition here. In particular, we will only allow adversaries whose queries are "admissible" in the following sense:

- Whenever the adversary makes a query (4), requiring the challenger to send its password for a given domain d, on its *last* query (1) adding passwords for a domain d, it must be the case that $\text{password}_0 = \text{password}_1$.

(Indeed, it is fairly easy to see that without this restriction, the adversary could trivially win the game—by query (1), it could cause the challenger to add a password for some domain d, under the adversary's control, so that the password value differed between worlds 0 and 1; and then require, by query (4), that the challenger authenticate to it using domain d's password.)

Intuitively, this definition captures the fact that even if the adversary is able to exert substantial control over the contents of the password database—and even if it controls some malicious remote servers—it still cannot learn anything about the passwords in the database for any *other* servers.

For this project, you will not be required to give a formal proof that your system fulfills the strong security definition we have just stated. However, you should note that this definition immediately precludes a number of interesting attacks, notably **swap attacks** and **rollback attacks**. In a swap attack, the adversary interchanges the values corresponding to different keys. For instance, the adversary might switch the entries for `www.google.com` and `www.evil.com`. Then, when the user (for whatever reason) tries to authenticate to `www.evil.com`, the user inadvertently provides its credentials for `www.google.com`. It should be easy to see that an adversary able to perform a swap attack can easily win the security game we outlined above. In your implementation, you must provide a defense against a swap attack.

In a rollback attack, the adversary can replace a record with a previous version of the record. For example, suppose the adversary was able to retrieve the KVS in the example above. At some later time, the user changes her/his password for `www.google.com` to `google_pwd`, which would update the value for `www.google.com` in the KVS. However, the adversary can replace this updated record with the previous record for `www.google.com`. Note that, as in the previous section, merely using authenticated encryption does not protect against this attack. Rather, in your implementation, you should compute a SHA-256 hash of the contents of the password manager. You can assume this hash value can be saved to a trusted storage medium (inaccessible to the adversary)—such as a flash drive on the user's person. Whenever you load the password manager from disk, you should verify that the hash is valid. This way, you can be assured that the contents of the KVS have not been tampered with.

Depending on your design, your defense against rollback attacks might also turn out to protect against the swap attacks described earlier. However, you **must still implement** an explicit defense against swap attacks. In other words, the defenses you develop must work **independently** of one another. Even if a SHA-256 hash is not provided from trusted storage, your scheme must be secure against an adversary that swaps two records.

[4] But similar to the CCA game.

2.3 PBKDF2 Security Assumption

In the previous section, we did not give a precise formulation of the security properties we are assuming of PBKDF2. To give the full picture, we would need to work in a framework called the "random oracle model," which would take us too far afield. Instead, we will specify here the properties we need for the password manager. Though you will not be required to submit a formal proof of security of your system, your system should still be secure in the threat model from Section 2.2 (and a proof should exist, even though you do not have to produce it).

We will assume that the master password is drawn from a distribution $\mathcal{D}$ of "common passwords", whose support is of exponential (but known) size – say, 2^{64} in practice. The adversary has oracle access to some unknown gadget $\mathcal{T}$ such that, for any "common password" x drawn from $\mathcal{D}$, the attacker can send $\mathcal{T}$ some auxiliary information $\text{aux}(x)$, and obtain x. In practice, we can think of $\mathcal{T}$ as a "rainbow table", generated by expensive precomputation. However, we assume that even in the presence of $\mathcal{T}$:

- Let $\mathcal{D}'$ be a distribution such that if y is drawn from $\mathcal{D}'$, then the probability that y is also in the support of $\mathcal{D}$ is negligible. Then, if an efficient adversary can distinguish $\mathsf{PBKDF2}(y)$ from a uniformly random 256-bit string, then there is another efficient adversary that can guess y with nonnegligible success rate (given no other information).
- If x is drawn from the password distribution $\mathcal{D}$, then no adversary can guess x (given no other information).
- If x (drawn from $\mathcal{D}$) is the user's true password, then, letting $m(x)$ be the set of passwords the user might ever enter (say, by mistake, due to a typo), no adversary can guess any element of $m(x)$ with nonnegligible probability (given no other information).

The main takeaway from this disccusion is that you should always salt your passwords. Of course, the length of the salt is also important. For instance, a 1-bit salt won't do much. The above discussion should give you an idea of how to set the salt length, but to summarize: the size of the space $\mathcal{D}$ of common passwords (say, 2^{64}) should be a negligible fraction of the size of the space $\mathcal{D}'$ of salted passwords.

3 API description

Here are descriptions of the functions you will need to implement. For each function, we also prescribe the run-time your solution must achieve (as a function of the number of entries n in the password database). We will assume that the input values (domain names and passwords) are of length $O(1)$, and regard each operation on an efficient dictionary/object/in-memory-key-value-store as a single step. Of course, if your solution is asymptotically more efficient than what we prescribe, that is acceptable.

3.1 `keychain.init(password)`

- `password`: password used to protect the keychain (`string`)

- No return value
- Run-time: $O(1)$

This method should create a new KVS. This function is also responsible for generating the necessary keys you need to provide for the various functionality of the password manager. Once initialized, the password manager should be in ready to support the other functionality described in the API.

3.2 `keychain.load(password, representation, trusted_data_check)`

- `password`: password used to authenticate keychain (`string`)
- `representation`: JSON encoded serialization of the keychain (`string`)
- `trusted_data_check`: SHA-256 hash of the keychain; note that this is an **optional** parameter that is used to check integrity of the password manager (`string`)
- Returns: `boolean`
- Run-time: $O(n)$

This method loads the keychain state from a serialized representation. You can assume that `representation` is a valid serialization generated by a call to `keychain.dump()`. This function should verify that the given `password` is valid for the keychain. If the parameter `trusted_data_check` is provided,[5] this function should also affirm the integrity of the KVS. If tampering is detected, this function should throw an exception. If everything passes, the function should return `true` and the keychain object should be ready to support the other functionality described in the API. If the provided `password` is invalid, the function should return `false`.

If this method is called with the wrong master password, your code must return `false`, and no other queries `keychain.get`, `keychain.set`, or `keychain.remove` can be performed on the password manager unless the client calls `keychain.init` or successfully calls `keychain.load`. It is incorrect for your password manager to pretend like nothing is wrong when the wrong password is provided to `keychain.load`, and only later, fail to answer queries.

3.3 `keychain.dump()`

- Returns: an array consisting of two components, where the first is a JSON encoded serialization of the keychain and the second is a SHA-256 hash of the contents of the keychain (`array`)
- Run-time: $O(n)$

If the keychain has not been initialized or successfully loaded into memory, this method should return `null`. Otherwise, this method should create a JSON encoded serialization of the keychain, such that it may be loaded back into memory via a subsequent call to `keychain.load`.

[5] In Javascript, if an argument to a function is not provided, its value will be the special sentinel value `undefined`. You can test that a value is not `undefined` using the expression: `x !== undefined`.

3.4 `keychain.set(name, value)`

- `name`: domain name of entry to add to the password manager (`string`)
- `value`: value associated with the given domain to store in the password manager (`string`)
- No return value
- Run-time: $O(1)$

If the keychain has not been initialized or successfully loaded into memory, this method should throw an exception: `throw "Keychain not initialized."` Otherwise, the method should insert the domain and associated data into the KVS. If the domain is already in the password manager, this method will update its value. Otherwise, it will create a new entry.

3.5 `keychain.get(name)`

- `name`: domain name of entry to fetch (`string`)
- Return: `string` (the value associated with the requested domain, `null` if not found)
- Run-time: $O(1)$

If the keychain has not been initialized or successfully loaded into memory, this method should throw an exception: `throw "Keychain not initialized."` If the requested domain is in the KVS, then this method should return the the saved data associated with the domain. If the requested domain is not in the KVS, then this method should return `null`.

3.6 `keychain.remove(name)`

- `name`: domain name of entry to fetch (`string`)
- Return: `boolean` (`true` if record with the specified name is found, `false` otherwise)
- Run-time: $O(1)$

If the keychain has not been initialized or successfully loaded into memory, this method should throw an exception: `throw "Keychain not initialized."` If the requested domain is in the KVS, then this method should remove the record from the KVS. The method returns `true` in this case. Otherwise, if the specified domain is not present, return `false`.

4 Hints and Summary

4.1 Implementation hints

- You should run your code using node.js; we have provided a simple test suite, which you can run using the command:

```
node test-password-manager.js
```

Note that this test suite does not cover all of the properties we will test during grading, and in particular, does not capture many of the security requirements.

- Your password manager will depend on the Stanford Javascript Crypto Library (SJCL) for its underlying crypto implementation. However, you **should not need** to call the SJCL functions directly (and our starter code does not include it directly). Instead, you should consult our support code library, `lib.js`, which provides wrappers for any SJCL functions that you should need, as well as some additional utility functions.
- Serialization and deserialization of your password database (for the `load` and `dump`) functions should be done using JSON, via the standard Javascript APIs: `JSON.stringify` and `JSON.parse`.
- If your application detects tampering with any of its values at any point, it should throw an exception (thereby terminating the execution). We will not test what exception is thrown; it is fine to throw a string with an English description of the potential tampering.
- The functions `keychain.init`, `keychain.load`, `keychain.dump`, should make *at most one call* to PBKDF2. The functions `keychain.set`, `keychain.get`, `keychain.remove` should not make any calls to PBKDF2. PBKDF2 is designed to be slow, so you should make use of it as little as possible.
- The *only* thing you should assume about SHA-256 is that it is collision resistant.
- In the unlikely event that you should need it, documentation for SJCL is provided here: http://bitwiseshiftleft.github.io/sjcl/doc/.

4.2 Summary of requirements

To summarize, you must implement a secure password manager that satisfies the following properties:

- The underlying in-memory data structure for the password manager should be a key-value store (Javascript object), where the keys correspond to domain names and the values correspond to the passwords for the given domain names.
- The password manager should be protected by a master password. Your implementation cannot include the master password (or any values that would leak information about it) in the serialized password database.
- When you need to derive a key from a password, you should use PBKDF2.
- Your system should satisfy the security properties described in the threat model (Section 2.2). In particular, you should defend against swap attacks and rollback attacks.
- You should implement all of the API functions (Section 3) with the parameters described there. Notably, your defenses for swap attacks and rollback attacks should be *independent*—your system must continue to be secure against swap attacks even if a hash value from trusted storage is not provided.

- You should include answers to the short-answer questions (Section 5) in a file called `answers.txt` or `answers.pdf`.
- Extra credit (5 %): instead of keeping a SHA-256 hash in the trusted storage, modify your implementation so that the *only* item i trusted storage is a counter, which stores the number of times the database has ever been updated. (Note that your system must still satisfy all of the other properties above, and, in particular, must still satisfy the security definition of Section 2.2; if it does not, points may be deducted from the main score as well as the extra credit.)

5 Short-answer questions

In addition to your implementation, please include answers to the following questions regarding your implementation. Your answers need not be long, but should include important details.

1. Briefly describe your method for preventing the adversary from learning information about the lengths of the passwords stored 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.
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.
4. What if we had used a different MAC (other than HMAC) on the domain names to produce the keys for the key-value store? Would the scheme still satisfy the desired security properties? Either show this, or give an example of a secure MAC for which the resulting password manager implementation would be insecure.
5. In our specification, we leak the number of records in the password manager. Describe an approach to reduce or completely eliminate the information leaked about the number of records.

6 Questions?

If you have general questions, please post on them on Piazza at `https://piazza.com/class/i4jbd95to3hgp` so that all students can benefit from the answer. If you have a problem with your specific implementation, come to office hours or ask a private Piazza question.