Hotel Locks

Authors

- Alex Clark (ayc55)
- Reuben Rappaport (rbr76)

1.

We took inspiration for the design of our system from the discussion of reverse hash chains for one time passwords with tokens. Each lock in the system stores:

- `n_last`: the 16 bit index of the last seen password
- `p_last`: the 256 bit last seen password
- `id_room`: a 16 bit room id
- `programmed`: a 1 bit value which specifies whether the lock has ever been programmed
- a portable programmer tuple
  - `pp_salt`: the 128 bit salt
  - `hashed_pp_key`: the 256 bit salted and hashed portable programmer key
- `n_staff`: the 8 bit number of staff members stored
- a list of staff tuples
  - `staff_id`: the 16 bit staff id
  - `staff_salt`: the 128 bit salt
  - `hashed_staff_key`: the 256 bit salted and hashed staff key

Since there are about 100 staff members this takes up approximately 5 KB of space. The 16 KB of RAM available to the lock is more than sufficient to store all of this. Guest cards store

- `flag`: a 1 bit flag specifying that this is a guest card
- `n_next`: the 16 bit index of the next password
- `next`: the 256 bit next password
- `t_expiration`: the 32 bit expiration time (measured in milliseconds since 1970)
- `id_room`: the 16 bit room id

When a guest attempts to use a card on a lock the lock first checks that the `id_room` on the card matches its stored `id_room` and that `t_expiration` is less than its current system time. If `n_next` is greater than or equal to `n_last` it rejects the card immediately. If it is less than `n_last` then it computes

$$H^{n_{last}-n_{next}}(next)$$

and checks that it is equal to `last`. Since hashes are one way, knowing a password does not allow you to derive the next in the sequence. In order to bound the computation time we reject new indices that are more than 3 steps down the chain. Initially the encoder precomputes a database of all the one time keys for each room in the system. It stores the last assigned index for each room, starting at the end of a sequence. When it encodes a new card for that room it decrements the index and then encodes the card with the corresponding password.

The master card system takes its inspiration from the way password files are stored. Each master card stores

- `flag`: a 1 bit flag specifying that this is a master card
- `staff_id`: a 16 bit staff id
- `staff_key`: a 256 bit staff key

When a master card is used on the lock, the lock reads the card's stored `staff_id` and looks up the corresponding `staff_salt` and `hashed_staff_key` from its RAM. It then computes the `H(staff_salt,staff_key)` and checks that it is equal to `hashed_staff_key`.

The portable programmer uses more or less the same system without the id since there is only one authorized portable programmer. Once the portable programmer has been authenticated the lock supports full read-write access to its RAM. If the lock has never been programmed before then it will authenticate any portable programmer and store its salt and hashed key for the future. In order to support human usable dictionary encodings of passwords for programming, the portable programmer stores a small dictionary in its memory as does the encoder.

2.

When the encoder is installed at the hotel it will be given the number of rooms at the hotel. For each room it will generate a unique 256 bit cryptographically strong random number as the start of the hash chain and then iterate the hash chain $2^{14} = 16,384$ times, resulting in approximately sixteen thousand one time passwords. It will initalize each room's last issued index to 2^{14}. Then it will store each step in the hash chain along with the last issued index to a password database. Since each password is 256 bits this will require the following space.

$$600\ rooms * 2^{14}\ \frac{passwords}{room} * 256\ \frac{bits}{password} \approx 315\ MB$$

This is a completely reasonable amount of data to store on a general purpose computer. Since the life of the system is expected to be about 20 years (7300 days), 2^{14} passwords is enough that if there is a new guest in a room every single night and every single guest loses their keycard and has to have a new one issued once then the system will still not run out of passwords.

3.

Initially the encoder configures the portable programmer by generating a random salt and key and setting the portable programmer up to use those as its id.

```
1. E: pp_salt = Gen(128)
      pp_key = Gen(256)
2. E -> P: pp_salt, pp_key
3. P: stored_pp_salt = pp_salt
      stored_pp_key = pp_key
```

Next to configure each lock the encoder looks up the appropriate starting password for the room id and uses a dictionary encoding to print out the room id, starting number, and starting password. A human takes the print out and the portable programmer to the lock and enters in the dictionary encoded values. If the lock has been programmed before then it uses authentication to check that this is the same portable programmer before allowing it read write access, otherwise it saves the portable programmer's information so that it can authenticate it in the future. Then the portable programmer installs the lock's starting values

```
1. E: id_room = room_id
      n_last = 16384
      p_last = password_database[id_room][n_last]
2. E: id_room_dict = DictEncode(id_room)
      n_last_dict = DictEncode(n_last)
      p_last_dict = DictEncode(p_last)
3. E -> Hu: id_room_dict, n_last_dict, p_last_dict
4. Hu -> P: id_room_dict, n_last_dict, p_last_dict
5. P: id_room = DictDecode(id_room_dict)
      n_last = DictDecode(n_last_dict)
      p_last = DictDecode(p_last_dict)
6. P -> L: pp_salt, pp_key
7. L: if programmed = 0 then
          programmed = 1
          stored_pp_salt = pp_salt
          stored_hashed_pp_key = H(pp_salt,pp_key)
      else
          check that H(stored_pp_salt,pp_key) = stored_hashed_pp_key
8. P -> L: id_room, n_last, p_last
9. L: stored_id_room = id_room
      stored_n_last = n_last
      stored_p_last = p_last
```

To add or remove a staff member from the authorized list the encoder uses the same dictionary encoding to communicate with the portable programmer via human. The portable programmer and the lock follow the same authentication protocol but then either add or remove the staff member from the stored list

```
1. E: staff_id = unused staff id
      staff_salt = Gen(128)
      staff_key = Gen(256)
2. E: staff_id_dict = DictEncode(staff_id)
      staff_salt_dict = DictEncode(staff_salt_dict)
      staff_key_dict = DictEncode(staff_key)
3. E -> Hu: staff_id_dict, staff_salt_dict, staff_key_dict
4. Hu -> P: staff_id_dict, staff_salt_dict, staff_key_dict
5. P: staff_id = DictDecode(staff_id_dict)
      staff_salt = DictDecode(staff_salt_dict)
      staff_key = DictDecode(staff_key_dict)
6. P -> L: pp_salt, pp_key
7. L: if programmed = 0 then
          programmed = 1
          stored_pp_salt = pp_salt
          stored_hashed_pp_key = H(pp_salt,pp_key)
      else
          check that H(stored_pp_salt,pp_key) = stored_hashed_pp_key
8. P -> L: staff_id, staff_salt, staff_key, ADD or REMOVE
9. L: if ADD then
          add (staff_id, staff_salt, H(staff_salt,staff_key)) to the list
      else
          look up staff_id in the list and remove it if it is present
```

4.

When the encoder creates a guest keycard for a specific room it first looks up the last issued index for that room. It then decrements that index and looks up the one time password corresponding to the new index. It encodes the guest card as follows

- `flag = 0`
- `n_next = the new index`
- `next = the corresponding password`
- `t_expiration = the unix standard time when the guest's reservation will expire`
- `id_room = the room's id`

This works both for checkin and for when a guest loses their card and needs a new one issued. As soon as the new card is used to authenticate the room the lock will no longer accept the old card, neatly revoking access to the possibly stolen original token.

5.

When the encoder creates a new staff keycard it first assigns the staff member a unique 16 bit id. Then it randomly generates a cryptographically strong 128 bit salt and a 256 bit key. It creates a master keycard as follows

- `flag = 1`
- `staff_id = the unique id`
- `staff_key = the generated key`

In order to add the keycard to the system, a staff member will need to go around to each lock with the system and use the portable programmer to add the tuple `(staff_id, staff_salt, hashed_staff_key)` to its list of authorized staff members. In order to make it reasonable for a human to enter this data into the portable programmer, the encoder will use the dictionary word encoding to print out a list of short words for the staff member responsible to carry around and enter at each lock.

Similarly, when a staff member loses their card or is fired, the card will need to be revoked by sending someone around to each lock in the system and using the portable programmer to remove that staff member from the list of authorized workers. Although this is a burden we may assume that both events - the hiring of a new staff member and the firing of an old one are fairly rare so this will not have to be done very often.

6.

To authenticate a keycard to a lock, the first check after swiping is whether the card's flag is set to 1 (to signify a staff card) or 0 (to signify a guest card).

```
1. K -> L: flag
2. L: if (flag == 1) execute staff protocol, otherwise execute guest protocol
```

For a staff card, the lock looks up the salt and hashed staff key, stored in RAM, corresponding to the card's staff ID. The lock then computes the SHA-256 hash on its own salt and the card's staff key, checking that the result is equal to the lock's stored hashed staff key value. Once this is done, the staff card has been authenticated, and the lock opens.

```
1. K -> L: staff_id, staff_key
2. L: if staff_id is not stored reject card
3. L: staff_salt, hashed_staff_key = lookup(staff_id)
4. L: if H(staff_salt,staff_key) = hashed_staff_key then
          open door
      else
          reject card
```

For a guest card, the lock checks its own room ID against the card's room ID to ensure the card is meant for this door, and verifies that the card has not expired (i.e. that the guest's allotted time has passed) by comparing the card's expiration time to the lock's internal clock time. Then it checks whether `n_next`, the number of times the card's password has been hashed, exceeds `n_last` (the number of times that the last accepted card had been hashed). If `n_next >= n_last`, it means the card's value is older than the last accepted keycard, and thus rejects it.

If `n_next < n_last`, and `n_last - n_next < 3` (ie it is not too far down the chain) then it hashes `next` `n_last - n_next` times and checks that this is equal to `last`. If it is then it opens the door and if`n_next` is more than 1 step down the chain updates the stored values for `n_last` and `p_last`. When a new card is issued `n_last - n_next = 2` and the stored values are updated. The same occurs if a single guest never enters the room. However if two guests in a row do not enter the room then the system will reject the new card and a staff member will have to come by with a portable programmer and manually fix the stored values.

```
1. K -> L: n_next, next, t_expiration, id_room
2. L: if t_expiration >= current_time reject card
3. L: if id_room != stored_id_room reject card
4. L: if n_next >= n_last reject card
5. L: if n_last - n_next > 3 reject card
6. L: p = H^(n_last - n_next) (next)
7. L: if p = last
          if n_last - n_next > 1
              n_last = n_next + 1
              p_last = H^(n_last - n_next - 1) (next)
          open door
      else
          reject card
```

7.

This solution satisfies revocation of access from former occupants because whenever a keycard is authenticated for a lock, if its passcode represents a "newer" passcode in the reverse hash-chain, the lock's `n_last` variable will be updated, and older passcodes will no longer be checked for the purpose of being accepted. Thus, whenever a newer occupant uses their keycard to unlock a room, no former occupants will be able to. This will currently be satisfied if up to 2 immediately previous guests did not actually enter the room.

8.

This solution satisfies the condition for timely response from the lock. When the lock authenticates a guest card it will perform at most three SHA-256 hashes on a value of size 256 bits. This takes 6 milliseconds. When the lock authenticates a staff card or a portable programer it performs one SHA-256 hash on a value of size 384 bits. This takes 2 milliseconds.

9.

This solution satisfies the condition for defense against malicious guests. A guest keycard will be unable to open any lock other than its assigned one, due to the lock's room ID being checked against the card's room ID. Additionally, it is highly unlikely that any card will be able to open a lock in a different hotel, since we are randomly generating 256-bit passwords. We expect to obtain the first collision after drawing $O(\sqrt{2^{256}}) = O(2^{128})$ values. This is far more rooms than could possibly be in the hotel system.

We also defend against malicious guests who have their own means of reading a keycard, since the reverse-hash chains used for room passwords give no way of predicting the next password in a chain due to hashes being one way, and there will also be no relation between the password chains for different rooms. Even with read access to the lock's memory, a guest would be unable to recover the next password. Similarly, since staff passwords are stored salted and hashed, a guest will not be able to pull it directly out of memory.

Additionally, we defend against malicious guests who have their own portable programmers, due to requiring that a portable programmer be authenticated with its own password, which in a non-hashed form is only saved on the programmer itself and in the encoder, and therefore would only be obtainable by a guest stealing the hotel's portable programmer or accessing the encoder (out of scope in terms of threats we are expected to defend against). The only time a lock is potentially vulnerable is after it has first been installed and not yet programmed, at which point it will retain the information of the first portable programmer used to connect to it, but this is a relatively straightforward vulnerability to protect against by ensuring locks are programmed quickly after installation.