**Network Security Project 1 Design Documentation**

Group 10

**Note**

1. In terms of the card and HMAC master key stored in the cloud, we set 2 minutes to expire but in practice, it is usually expired after one day.
2. The remaining validity period of the ticket is 2 minutes for testing, however, it takes normally a few days to expire in the real world.
3. The maximum number of ride is 20 for testing while more than 100 rides are common in the production environment.

**Cloud**

1. We implemented the backend with Deno and PostgreSQL. The backend renders the login page, ticket-blocking and logs viewing/clearing management pages.
2. The server authentication is realized with the help of HTTPS and TLS protocols. The client authentication is realized with the help of the API secret with card UID as salt value. (API master secret is hidden in the source code using the [hidden-secrets-gradle-plugin](https://github.com/klaxit/hidden-secrets-gradle-plugin) by storing an obfuscated secret into an NDK binary as a hexadecimal array).
3. We have implemented blacklisting tickets in the cloud so stolen, hacked, or cloned tickets can get blocked manually. Serial numbers are stored as base64 strings. Detected forgeries can get added to the blacklist, which is downloaded to the ticket reader asynchronously whenever the serial number is requested from the ticket. The reader device can work without the Internet and update the list with the cloud connection when available.
4. We store the card and HMAC master key in the cloud. We have a cache for the keys by encrypting using the Android Keystore and storing them in the shared preference. It is available locally for two minutes (testing) before expiry. Every time the cache expires, we fetch the keys from the cloud and compare them with the local ones. If any difference is detected, we send a log notifying the malicious issue to the cloud and stop the reader (distributed verification). It will work as usual if the key expires and there is no Internet connection.
5. The actual API key is calculated by PBKDF2WithHmacSHA512(master key | form data). Form data includes the card serial number and the timestamp when the request is made, so that blocked cards cannot get the card and HMAC master key. The requested timestamp should be younger than 5 seconds when the request gets handled.
6. Logs are cached in the card reader. The reader submits cached logs and clears locally when the Internet is OK.
7. The logs viewing page supports clearing logs in the cloud that are older than the specified minute (GDPR needs).
8. Logs record the identifier (serial number for the ticket events logging and IP address for other requests), timestamp, remaining uses, and log type. The logging type includes the following:
9. Tickets get issued.
10. Tickets are used.
11. User logs into the management system.
12. The card reader fetches the master key.
13. The card reader fetches the HMAC key.
14. A ticket gets blocked.
15. A ticket gets unblocked.
16. Malicious events are happening.
17. We consider the following events as malicious:
18. The ticket stored expected counter differs from the actual card counter value (MitM replay attack).
19. The remaining rides are greater than 20.
20. The last ticket check-in time is greater than the current time.
21. The initial counter value does not match the current counter value, while the expiry time is 0.
22. The remaining validity period of the ticket is greater than 2 minutes (max expiry time).
23. Locally stored keys decrypted using Android Keystore are different from the cloud.

**Ticket**

1. We store the ticket data only on the card (for working offline without the cloud).
2. We use the card-featured counter for counting used rides, and write the initial counter value when issuing the ticket, as well as the expected counter value when the ticket is used to the ticket to prevent the MitM attacker.
3. We write a timestamp about the recent validation (last check-in time) into the card.
4. The tickets are valid for 2 minutes (testing) from when they were first used.
5. Start the validity period only when the ticket is used for the first time.
6. We set 4 bytes of message authentication code (MAC) calculated with UID and ticket data.
7. The ticket HMAC is calculated by combining the card serial number, counter limit, initial counter value, expected counter value, last check-in time, and ticket expiry time together.
8. The card key or HMAC key is calculated by PBKDF2WithHmacSHA512(master key | card serial number)
9. New data is written to different pages than the old data, and a counter update commits the write operation. We use odd and even counter values to indicate different memory pages for the ticket data block (even for block 2 and odd for block 1), so the previous ticket is always retained. The counter is incremented as the final step.
10. Aside from the malicious events check mentioned in the cloud section, we also check HMAC and whether the counter has reached the maximum value during both the issue and use process.
11. We log basic information for the latest five events, like the timestamp, event type, and remaining uses into tickets.

**Memory data structure:**

**Tag block**

**4**  -> application tag (CSE4)

**5**  -> version (v0.1)

**2 ticket blocks**

**6** -> max ride number (counter limit)

**7**  -> initial counter, expected counter

**8** -> last check-in time

**9** -> expiration time

**10** -> hmac

**11** -> max ride number (counter limit)

**12** -> initial counter, expected counter

**13** -> last check-in time

**14** -> expiration time

**15** -> hmac

**Logs block**

**30,31** -> timestamp, remaining ride, type

**32,33** -> timestamp, remaining ride, type

**34,35** -> timestamp, remaining ride, type

**36,37** -> timestamp, remaining ride, type

**38,39** -> timestamp, remaining ride, type

*If any of the following flows get interrupted, we will show "Communication error!" to the user.*

**Issue flow**

1. We first check the serial number, determine whether the card is blocked, and then calculate the card key.
2. Authenticate the card with the default key, assuming the card is blank. If we succeed, we set the header and update the password. If it fails, then the card is not blank. We authenticate with our key and check the header, then read the ticket data from the block according to the counter parity, and then check if the ticket data is valid. If ticket data is valid, but the remaining rides exceed 15 (to keep maximum rides under 20), reject top-up more.
3. We issue tickets with a constant number of rides (5).
4. Issue additional rides (+5) to a card without erasing any still-valid ticket.
5. We reset the counter limit when tickets expire or have been used up.
6. Calculate the counter limit according to the above policy, set the check-in time to the current time, and the expiry time to 0 (to be determined during use).
7. We set AUTH0 to 3 and AUTH1 to 0 to protect the whole card memory against reading (remaining replay) and writing without authentication. (We should still protect the whole memory instead starting from the ticket block. Otherwise, the attacker can modify the header to play a downgrade attack or just a DoS attack by destroying the ticket. Of course, locking the tag block may be good, but we shall make the ticket recyclable and upgradeable)
8. Write the ticket data to another new block, and log the event.

**Use flow**

1. We first check the serial number, determine whether the card is blocked, and then calculate the card key.
2. Authenticate with the key and check the header, then read the ticket data from the block according to the counter value, and check if the ticket is expired, used up, or the last check-in time is within 5 seconds (testing)(pass-back).
3. If the initial counter value matches the current counter value, set the expiry time to be 2 minutes (testing) later.
4. Write the ticket data to another new block, increase the counter, and log the event.