Authors: Floran NARENJI-SHESHKALANI (643166) & Jean-Marcellin TRUONG (643357)

1 Featureset

This ski ticket can hold both value and time subscription. The former is stored as a decrementing number of rides. For the later, a dormant subscription is stored as a number of hours, while an active subscription is stored as an expiry date/time. Time subscriptions will be used prioritarily over value.

This design is simplistic, and does not support advanced features such as cash on the ticket, zones or pay-per-hour subscriptions.

With the goal here being NFC-related technical specification rather than ski ticket functional specification, it was felt that the above features were sufficient for a proof of concept. Furthermore, the current design uses less than half of the storage space, and already provides a tear-proof, mostly tamper-proof way, eventually fraud-proof (as in 'eventually consistent') way of storing offline information on the card. It should therefore be quite straightforward to extend the features for more complex use cases.

The ticket also provides pass back protection with a one minute cooldown time between ticket validations, no matter the type of subscription. Pass back protection here is intended against malicious users trying use a single card for two persons (i.e. the card is personnal and not to be shared).

The length of the timer is easily adjustable in the code (for the prototype). It should not be too short for safety reasons (e.g. someone could loiter until the timer resets) and not too long in case someone mistakenly validates his card and is then locked out of the ride for an unreasonable amount of time.

As the card stores the date of last validation, the sister feature (not billing twice for the same ride) could be implemented by a rather simple software update deployed to the readers, without updating the card's internal structure.

2 Implementation

2.1 Platform choice

The prototype runs on the PC platform instead of the Android default. The main motivation for that choice is that the PC provides a faster edit/build/debug cycle and also had good support for unit testing. With the provided NFC library being in Java, it was elected to write the prototype in Scala (both run on the JVM and therefore have good interoperability). Scala was chosen because it is generally a more modern, elegant and fun (in the writer's opinion) language compared to Java.

2.2 Serialization library

For manipulating binary data, we use the `scodec` library. This serialization library defines an algebra for mapping Scala data structures ('case classes') to binary format by usage of codecs.

A lot of useful codecs are provided by default, however the use cases at hand required some custom codecs (e.g. for MAC, offset date time storage…).

`scodec` is designed such that the programmer does not explicitly select the individual position of each field of the ticket. Instead, the (simplified) idea is to define a codec for each field and to let `scodec` handle bit packing, manipulation, alignment…. Note that `scodec` preserves the order of the fields as written in the codec & case class.

For a interacting with space constrained environments such as an NFC card or legacy systems (with different endianesses…), which are clearly not Java's forte, `scodec` is quite handy.

The custom codecs can be found in the `src/main/scala/skiticket/utils/codecs/package.scala` file. The codec for the Ticket class itself is at the bottom of the `src/main/scala/skiticket/data/Ticket.scala` file.

2.3 Unit testing

For unit testing, `ScalaTest`, the standard unit testing library for Scala, is used. Unit tests enable painless writing of fully autonomous and unsupervised test cases for all the use cases of the library. Compared to manual testing, this is even more advantageous than for traditional software: this removes the need for unlocking a smartphone,

deploying the app, tapping the card one or more times depending on the complexity of the use case, and also avoids all form of run-time human errors.

Note that those are not unit tests in the strictest sense as they rely on external resources (NFC card, NFC card reader & in-memory database) and necessarily test more than a single feature (e.g. NFC test cases begin with at least authentication and formatting).

Unit tests can be found the `src/test/scala/skiticket/nfc/NfcTicketTest.scala` file.

2.4 Runnable

The final form of the prototype is a library with a demonstration application. Having the ticket functionalities as library (in the sense of a separate set of classes) potentially allows for porting on a different platform, and for usage with other JVM language, such as Java (e.g. porting to an Android app is realistic).

The demonstration client is located in the `src/main/scala/skiticket/Main.scala` file. Its features are described in the section below.

The library itself consists of the subfolders in the `src/main/scala/skiticket/` folder. All files are commented where useful and have JavaDoc.

2.5 Sanity checks

The number of rides loaded on a card can not exceed 100. The maximum duration of a subscription is 365 days (e.g. season pass including glacier skiing).

3 Running instructions

3.1 Prebuilt binaries

For convenience's sake, a JAR file embedding all dependencies is provided (including Scala-related). This does not include card drivers, as those need to be set up separately.

Usage is then: `java -jar SkiTicket-assembly-0.1.0-SNAPSHOT.jar`.

Shortly, the demonstration application allows issuing both kinds of subscription with the option of forcing formatting and use (validation) with the options of disabling pass back checking, blacklisting a specific card, and tearing test mode.

The full usage guide is printed by the application when ran without arguments (as above).

3.2 Building & Testing

In order to build the fat JAR file, the `sbt` tool is required. `sbt` is the standard Scala build tool, and is found in the homonymous package in all major Linux distributions.

In the root of the project, run `sbt assembly` to package the fat JAR containing all dependencies. The output file will be `target/scala-2.12/SkiTicket-assembly-0.1.0-SNAPSHOT.jar`.

To run the unit tests, use `sbt test`. The card reader should already be connected and an NFC card should be in place. No further human interaction is required for the duration of the tests. At the end of the tests, a summary will be shown.

Both commands above will automatically take care of all necessary steps such as dependency download (which can take some time for first launch) and compilation.

4 NFC card internals

4.1 Card-based security

As soon as the card is formatted, the 3DES authentication key is changed from its default to a key calculated as follows, with H being SHA256 and || being concatenation:

$$K = H(H(MasterKey)||H(UID))_{0..15}$$

The key is therefore a function of both hash and UID.
The authentication bits are set such that reading and writing on any page requires authentication.

4.2 Memory structure

#	Byte 0	Byte 1	Byte 2	Byte 3
4	S	K	I	0
5 6 7 8 9 10 11 12 13 14	Data #1			
14 15 16 17 18 19 20 21 22 23	Data #2			
...				
41	Mono. counter			

Table 1: NFC card memory organization

#	Byte 0	Byte 1	Byte 2	Byte 3
i+0	No. rides		Sub #1	
i+1	Sub #1		Sub #2	
i+2	Sub #2		Sub #3	
i+3	Sub #3		Last val.	
i+4	Last val.		MAC	
i+5	MAC			
i+6	MAC			
i+7	MAC			
i+8	MAC			

Table 2: Data block structure

4.3 Features

In Table 2, we can see all the feature fields. The number of rides is stored as a decrementing short integer. The subscriptions are each stored as a 4-byte integer. A dormant one has its first bit set to 1 and the remaining bits contains the number of hours for the subscription. An active one has its first bit set to 0 and is stored as an offset number of seconds representing the expiry date. The SKI epoch is set to 2017-12-03 00:00:00, to avoid overflow concerns. The expiry date is calculated as time of first use plus duration in hours.

The last validation field provides information for the pass back detection feature, and is also stored as an offset number of seconds since SKI epoch.

Finally, a MAC is appended as a way to validate the authenticty of the information contained in the card. This prevents man-in-the-middle attacks. The MAC is computed as, with H being SHA-256, $HMAC$ being HMAC-MD5 and || being concatenation:

$$K = H(H(MasterKey)||H(UID))$$
$$MAC = HMAC(Data||Counter, K)$$

The counter present in the MAC is the monotonic counter. This counter is increased every time the card is successfully written, e.g after issuing or validation. Note that usage of the monotonic counter is not necessary for that, it would have been sufficient to just have the counter as part of the data block. This choice is justified by the fact that the monotonic has a second use in tearing protection.

4.4 Tearing protection

One of the major concerns when using this card is tearing: if the card tears, it becomes entirely unusable, which implies bad customer experience.

The card used here has no support for transactions. Therefore, they had to be manually implemented.

An atomic operation is a requirement for implementing transactions, and it appears that increasing the monotonic counter is one such operation.

The final implementation of transactions is inspired by the left-right concurrency pattern. As seen in Table 1, the card contains two data blocks. At any time, only one data block is considered to be holding the correct information and that is the data block that is pointed at by the last bit of the monotonic counter.

This leaves the application free to write however many pages it wants to the other block, because the contents of that block have no relevance to the card's functional ability. Finally, when done writing, the application atomically increases the monotonic counter, marking the newly written data block as the working one.

5 Final thoughts

Because the card use 3DES as an authentication measure, storage of the master key is the main concern in order to ensure the safety of the system. In the prototype, the master key is locally stored in the reader device. Even if it was stored inside a secure hardware module, theft of it would still be possible: because the reader device is outside, it has, from a security standpoint, to be considered as if it was in the enemy's hands.

We've thought long and hard, discussed with the course's members multiple times and came to the following conclusion.

Many mitigation techniques could be employed. They could contain elements of data heuristics, asymmetric cryptography (as discussed during the presentation) or require permanent online access (which is generally far from guaranteed in a skiing resort).

In the end, the system's main limitation is really the card (lack of session security and asymmetric cryptography on the card's side). Development & maintenance costs those heuristics or online systems would probably be higher than simply having a better, higher-end card. From a business standpoint, better cards would have a deposit on them, making that a higher initial investiment, but a zero-sum operation in the mid term (or one could even make a profit on the deposits).

From another point of view, one might also argue that the system does not need to be mathematically foolproof, only strong enough to deter most attackers. Whether that is true or not can be argued both ways.