Notes on NFC ticket design on MIFARE Ultralight C

(last updated 2018-11-08)

Tuomas Aura, Aalto University

Application is a data structure

The inexpensive smart cards used for ticketing applications, such as MIFARE Ultralight, Ultralight C, and DESFire, are basically memory cards with some special security features. The smart-card application development for these cards does not require writing any code that would run on the card. Instead, it is about defining the data structures and their layout in the card memory, and about key management. All the code and business logic is on the reader side. There exist programmable smart cards, such as JavaCard, but secure application development on them is much more complicated.

Stored value vs. identification

The smart cards have been designed to store value, such as tickets or the amount of money in a wallet. The architectural principle is that all the information needed for validating the tickets is stored on the card, which is carried by the user. This philosophy is followed especially by the developers of transport ticketing systems because the ticket readers may be mobile or without internet connection. An alternative solution would be to use the card only for identification and authentication, and to store all other information in the reader or in an online server. In this course, however, you are asked to design a very traditional stored-value application that stores the ticket data only on the card.

For security, it would be a good idea to keep track of the issued and used tickets also in an online server, even if that information is not updated or verified from the backend in real time. Having a shadow copy of the information online allows for security auditing, blacklisting of stolen, hacked or cloned tickets, and replacement of lost tickets. In older transport ticketing systems, the synchronization between the reader and backend happened daily at the vehicle depot, while the readers in new systems may be continuously online and synchronize event data within seconds. For online validation in real time, the delay should be at most 300 ms, which has not been achievable with mobile networks and cloud infrastructure. Ticket sales kiosks, on the other hand, tend to be online because the sales transaction admits longer delays than ticket validation in a vehicle or at an entry gate.

Cryptographic authentication between the card and reader

With Ultralight C, the simplest ticket design is to rely entirely on the built-in shared-key authentication: authenticate the card with a secret key and then read and write ticket information. The mutual secret-key authentication allows the reader to verify that the ticket is authentic, and it prevents modification of the ticket information such as the expiry date or number of rides without knowing the secret key. Cloning of the ticket to another blank Ultralight C card or to a special clone card is also prevented because the attacker would need to extract the secret key first.

Thus, you should replace the default key on each card with an application-specific secret key when initially formatting the card. The secret key needs to be stored on the reader or a backend cloud server. Also, you should remember to set the AUTH1 parameter on the card because, by default, all the memory pages can be written without authentication.

Many inexperienced ticketing application designers would stop here. However, there are further security and reliability issues that can and should be considered.

Systemic failure caused by a leaked key

Naturally, the secret key for the card authentication needs to be stored both on the reader and on the card. If all cards have the same secret key, it suffices for someone to physically break the security of one reader or one card and extract the key. After that, they can write to any of the cards and create tickets on blank cards. To prevent physical and hacking attacks that try to recover the secret key from a reader, the key in the reader can be stored in a physical security module (e.g. MIFARE SAM). Unfortunately, the SAMs have been too difficult or expensive to acquire for small deployments. In many systems, such as ticket vending machines, the reader key can be stored in an online server and not on the reader device. In vehicles, the reader device can be built to withstand physical hacking. The smart cards, however, are in the hands of the users. Criminals can easily get hold of several smart card tickets and expose them to analysis in a laboratory to discover the secret key on them.

Key diversification, event logging, and misuse detection

To prevent the large-scale security failure caused by breaking of one card, it is better to use a different secret key for each card. This is done by computing the diversified (i.e. per-card) key as *K = h(master secret | UID)* where *h* is a cryptographic hash function, UID is the unique 7-byte identifier of the card, and the *master secret* is stored only on the reader side (preferably in a physical security module or in an online server). Now, the attacker might still be able to extract the diversified key from the card and produce many copies of the card, but once the fraud is detected, the cloned or hacked cards can be blacklisted by the UID. For this to work, it is essential to monitor for situations where the same UID appears more often in event logs or moves faster in the geographic area than a real transport user would. System designers should include key diversification and event logging into the system design from the beginning, while misuse detection can be added later.

Sometimes it is better to write an application-specific ticket identifier on the card and use it for event logging and key diversification instead of the smart-card UID. In fact, this is the safe design option because most modern systems will eventually allow other media, such as mobile devices with a secure hardware module, to be used instead of the original type of smart card.

As a side note, any event logging on the backend needs to consider privacy regulation, such as GDPR. At minimum, there needs to be a way to delete old logs after some defined period, when they are no longer needed.

Application tag and version number

It is a good idea to reserve one or two pages of the card memory for an application tag and version number. The application tag is simply a constant string that is specific to your application and ticket design. It can be used to quickly identify cards that have been initialized for this specific application. If the correct application tag is found on the card, it is reasonable to assume that the authentication key has been changed to the application-specific (and diversified) key. On the other hand, if the application tag is not found on the card, it may be a blank card that can be formatted for your application. In that case, the authentication key should still have the manufacturer's default value. It could also be that the card has been initialized for an entirely different application. In commercial systems, you would want to check that the card is blank before formatting it and reject any previously used cards. During software development and in non-professional use, you may want to recycle as many cards as possible. To reuse a card, you need to know the card key, and you need to check that the values of the lock bits, OTP and counter are in acceptable initial state for your application.

The version number field will be helpful later when you update the data-structure specification. It enables multiple versions of the ticket to co-exist during a transition period.

Man-in-the-middle (MitM) attacks between the reader and card

Ultralight C does not have a secure session protocol. That is, after the initial cryptographic authentication, the communication between the reader and the card is insecure. Thus, it is possible mount a man-in-the-middle attack between the reader and the card and to send commands that modify the card contents after the real authentication. For example, the attacker could modify the expiry date or the maximum number of rides on the card. The MitM attack on NFC requires special hardware and is non-trivial to implement in practice. It probably will not be done by normal users, but someone could start a criminal business of selling modified cards e.g. with 10000 rides instead of 10. If the reader side consists of a secure CPU and an insecure USB card reader, it might also possible to spoof commands by becoming a MitM on the USB cable.

More advanced smart cards, such as the MIFARE DESFire ev.1 used for monthly bus passes in the Helsinki region, have session security that prevents the MitM attacks on the NFC interface, although early versions of DESFire have some weaknesses in this respect.

Message authentication code (MAC) on ticket data

To prevent the MitM attacks on the Ultralight family cards, you can compute a message authentication code (MAC) over the ticket data on the card, including the card UID, and write the MAC to the card. The reader should then verify the MAC when reading the ticket data. This prevents arbitrary modifications to the card data by the MitM attacker. Instead of including the UID into the MAC input, you can also compute the MAC with a diversified key.

The MAC does not need to be very long. If the attacker tries to create a correct MAC by brute-force guessing, it will need to tap the physical card reader after each guessed value to find out of the guess is correct. Depending on the application, just 4 bytes of MAC may be sufficient: the attacker will have to tap the reader 2^31 times, on the average, in order to make one malicious change to the card contents. Moreover, if the MAC is bound to the card UID, the produced false content cannot be reused on other cards, with makes such brute-force trials pointless for the attacker. In high-value applications (such as key cards to a treasure vault), the MAC should be slightly longer, but then it makes sense to use a more expensive smart card, instead.

It is usually not necessary to update the MAC when the card is validated. Instead, compute the MAC only on the issued ticket and exclude the ride counter and other continuously changing data from the MAC input. For example, the number of remaining rides can be calculated as the difference between the current counter value and an unchanging maximum value. The reasons for not updating the MAC are explained below in the Tearing section.

Rollback attack

Even with the MAC for integrity protection, the MitM attacker could write back old data to the same Ultralight C card, including the old MAC. The attacker simply reads the card content after the ticket is issued and stores the data. The attacker then uses the ticket in the normal way. Later, the attacker performs the MitM attack: after authentication between the card and a real reader, the MitM attacker stops forwarding commands between them and, instead, sends false commands that write the old data back into the card. This enables the attacker to restore the unused ticket to the card after using it. The attack is not practical for normal users because of the special equipment required for the MitM attack, but it is nevertheless a security vulnerability that may eventually cause problems. (Again, rollback is not a problem for the more advanced DESFire cards because the session security on those cards prevents the MitM step.)

Rollback prevention

To prevent roll-back on Ultralight C, it is necessary to use the special *OTP* or *counter* features on the card. Writing to these pages causes one-way changes to the card state that cannot be reverted. The simplest solution for rollback prevention in Ultralight C is to increment the 16-bit monotonic counter every time the ticket is used. For this reason, it is more secure to use the counter for counting used rides than to write the number of remaining rides to the normal memory pages. The MitM attacker cannot revert the counter value to an earlier state.

The OTP page on Ultralight and Ultralight C can be used similarly as a unary counter with maximum value of 32. Since Ultralight C has the 16-bit monotonic counter, the OTP bits are more commonly used for rollback prevention in other state changes that should be irreversible, such as to invalidate the entire card.

Remaining replay vulnerability

While the counter can prevent rollback of the card state, it is still possible to relay the earlier card state from the equipment used in the MitM attack. That is, the attacker reads the card memory after the ticket was issued. It them emulates card behavior on an FPGA board. Only the authentication is forwarded to the real card, and all other behavior is emulated. The attack will be made more difficult if you protect the card memory against reading (see the AUTH1 parameter), but ultimately the attacker will be able to learn the necessary card memory contents by monitoring the communication between the card and the real reader. (Again, the more advanced DESFire card prevent such attacks on the NFC interface by protecting session integrity between the card and the reader.)

Tearing

If the reader performs multiple write commands on the card, there is the possibility that the user will pull the card away from the reader too early, in the middle of writing. This easily happens when “tapping” the reader in a hurry. It can cause the card to go to an inconsistent state. For example, if the card is removed from the reader in the middle of writing the MAC, the result may be a card with a partly written MAC. After that, the reader will reject the ticket as invalid. Even if this happens with low probability, it will cause support costs and make some customers unhappy.

The word “tearing” comes from paper ticket printers where customers may pull the paper and cause it to be torn before the ticket has been completely printed. If that happens, should the customer be left without a valid ticket or should the printer output a second copy? It is difficult question, and therefore most ticket printers have been designed to prevent tearing by the customer. Pay attention to this when you next time buy a paper ticket form a vending machine.

Tearing protection

The DESFire cards used for the Helsinki region monthly bus passes have built-in tearing protection: a commit command similar to the one used in transactional databases. In these cards, if one taps the card on the reader too quickly, the card will return to the state that it had before the tapping. Cheap smart cards like Ultralight C, however, have no such fancy transaction features. To prevent tearing on the cheap cards, tapping operations like ticket *validation* should only do one write operation on the card, such as writing only one page, or setting only one bit in the OTP, or only incrementing the counter. This means that tapping operations should not update a MAC or other multi-page content on the card. More complex operations like ticket *issuing* are typically done by trained staff or by placing the card into a writer device for some seconds, which reduces the risk of tearing. This is the reason why ticket vendors place the black card into a holder in the reader instead of simply tapping the reader.

How can you implement a ticket with tearing protection on Ultralight C? Use card authentication and a MAC to protect the static content of the original, issued ticket, such as expiry date and value loaded to a wallet. Then, use the 16-bit counter or OTP to keep track of how much of the issued ticket has been used. Writing the dynamically changing data, such as the used or remaining ticket value, to just one memory page also solves the tearing problem, but then you don't get the rollback protection provided by the counter or OTP.

Sometimes, ticket validation may be a complex operation that cannot be implemented as a simple increment of the counter or by setting a bit in the OTP. The dynamically changing MAC or signature on the card may be an application requirement. In that case, you can design the multi-page updates to the card data structures in such a way that new data is written to different pages than the old data, and a counter update commits the write operation. For example, odd and even counter values could indicate different memory pages for the ticket, so that the previous ticket is always retained. The expected new counter value should be included into the MAC input, and the counter is incremented as the final step. Should tearing happen, the counter will keep the old value and the (partially) written new data will be ignored. If the number of expected transactions is small, you could also keep writing to new memory pages and locking the pages as you go.

Before the more advanced cards like DESFire became available for transport tickets, tearing protection was an important part of the ticket design. Nowadays, few designers would implement multi-page writes with tearing protection on cheap cards like Ultralight or Ultralight C. Your best option is to make sure that when the user taps the card in normal use, only one memory page or counter is written. If that is not feasible, switch to a more advanced smart card.

Transaction log on the card

It is a good idea to keep a log of the latest events, such as ticket purchase or use, in the card memory. It may be sufficient to store basic information such as the date and time, event type, and transaction amount for the 1-5 latest events. This will help in debugging and resolving problem cases. If the log file is kept only for informational and debugging purposes, it is not even necessary to worry about tearing of the log data because its integrity is not critical. In many ticketing systems, however, there is a way for service personnel to cancel (i.e. refund) the latest purchase, in which case the integrity of the log data may be considered critical.

Pass-back protection

One special case of event logging is the so-called *pass-back protection*. This refers to situations where one ticket is used by multiple people and passed back to outside the gate or inspection point. One special case is when the passenger in the bus shows her ticket to the ticket inspector and then secretly passes it to a friend towards the back of the vehicle who displays the same ticket. Fraudulent use of the ticket by more than one person at a time can be detected by writing a time stamp or other log information about the recent validation or inspection onto the card. That way, the entry gate or ticket inspector can notice if another passenger has already presented the same ticket.

Multiple use vs. idempotent operations

One important design decision is what happens when the user taps the ticket twice. Should the second tap cause the validation of a second ticket, or should it just reconfirm the previous event? Idempotent operations are good in the sense that, if the first validation attempt fails, or if the user suspect its failure, there is no harm in repeating the action. For example, if you validate a swimming hall ticket and then remember that you forgot your towel in the car, it would be nice to get in without having to use a second ticket. The disadvantage of such policy is that pass-back becomes a problem, and also that two people cannot pay with the same multi-ride smart-card ticket. There could, of course, be a user interface with different options on the ticket reader device, but most application designers prefer simple tapping without any other user interface. The Helsinki regional public transport is pretty unusual in having a user interface where passengers can choose the ticket type and the number of persons.

When to signal success

It is important to complete the entire transaction before signaling ticket-validation success. For example, if the ticket is validated at a turnstile, the turnstile should not be released immediately after checking ticket validity but, instead, only after incrementing the counter or making other updates to the card. Even though the difference may be only milliseconds and not visible to the user, this will prevent a simple MitM-like attacks where the ticket holder blocks the writing of updates to the card. In fact, it would be even more prudent to read back the written data and verify that it is correct on the card – but that should only be done if it does not cause an observable delay in the ticket validation process.

Checking for safe limits

Ticker designers should define some maximum values for security-critical parameters such as the length of the remaining validity period and the number of rides stored on the card. These limits should be checked both when the ticket is used and when the customer buys more rides to the same card. This can help to catch security failures. For example, if someone finds a way to make counterfeit tickets, e.g. with the MitM attack mentioned above, they probably want to produce as valuable fake tickets as possible, such as ones that are valid for 100 years and have thousands of rides pre-loaded. Checking the limits can prevent such attacks and alert the system administrators to any attempts.

Transaction speed

Card validation should be faster than 300 ms for the users to feel that it is instantaneous and for the flow of people past the reader to be smooth. With modern reader hardware, quite complex operations including cryptographic computation can be performed within this time. It is nevertheless a good idea to look at the timing data in the smart-card datasheet and to calculate the expected ticket validation times.

As mentioned earlier, the real performance issues in modern systems arise from connections to a remote online server. In transport applications where the user validates the ticket, online connections to remote servers are generally considered too slow and too unreliable.

One compromise solution between offline and online readers is to use the card only for identification and to push or cache the other information to the reader devices. The readers should then communicate with the cloud server with a few seconds of delay. This is the case, for example, in many smart-card-operated door locks, which may feel slower on the first access of the day.

On-the-fly modifications to data structures

What if your deployed card application does not meet future requirements? In theory, it would be possible to update the card application, i.e. the data structures on the card, after its deployment. The reader could perform the update operation when the user taps it with the card. But that is not done in practice. In low-end cards like Ultralight C, the dynamic multi-page update could fail because of tearing, making the update unreliable. Even on cards with built-in tearing protection, such as DESFire, programming the one-time update process to the reader devices is not an attractive solution. Thus, you should take care to define the card application in such a way that it covers the foreseeable future needs. Replacing physical cards that are held by the customers is an expensive process – and you have to be particularly careful about its implementation if there is value, such as tickets, stored on the card. (For example, all transport passes in the Helsinki region will be replaced in 2018, even though the new cards are physically the same DESFire ev.1 as most of the old cards, and a soft update would be possible in theory.)

Standard data structures

There are standard data structures for smart-card applications in transport ticketing. In particular, ISO standard 1545 defines many ready-to-use data elements and values in the ASN.1 notation. These are usually encoded as bits and bytes with the ASN.1 BER encoding rules. In non-transport applications, however, it is easier to define your own data structures. Moreover, the standards have been defined for the early smart cards that had very limited memory capacity (e.g. MIFARE Ultralight had 48 bytes of user memory compared to the 144 bytes in Ultralight C). With the newer cards, it is not necessary to optimize the length of every data field to the absolute minimum, and you can use full bytes or 32-bit pages for the data values. In any case, a professional smart card application designer or security analyst should be familiar with the standards (which sadly are available online only for a fee).