Security Protocol Project

Generic Feedback

Cristian Daniele & Erik Poll

Digital Security

Radboud University Nijmegen

Lifecycle changes

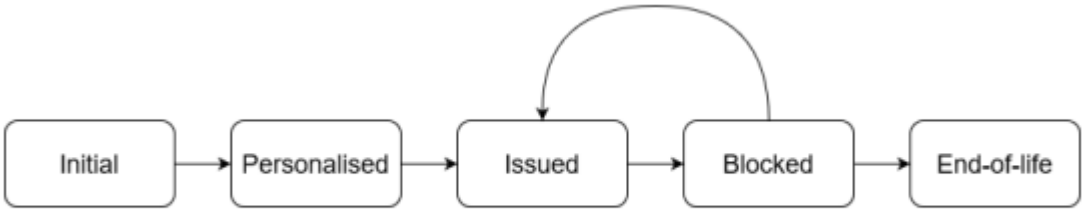

Figure 5: The lifecycle of a card

Lifecycle changes can happen by

1. an explicit action,

   eg. setting some boolean field to false or a byte field to `ISSUED`

2. something that happens implicitly,

   eg. a certificate expiring

- Mention any actions to change the lifecycle as explicit steps in your protocols
- If some lifecycle state (eg. end-of-life) or transition (e.g. unblocking) is not mentioned in any protocol, that is suspicious

Explicit lifecycle state transitions

Eg. in

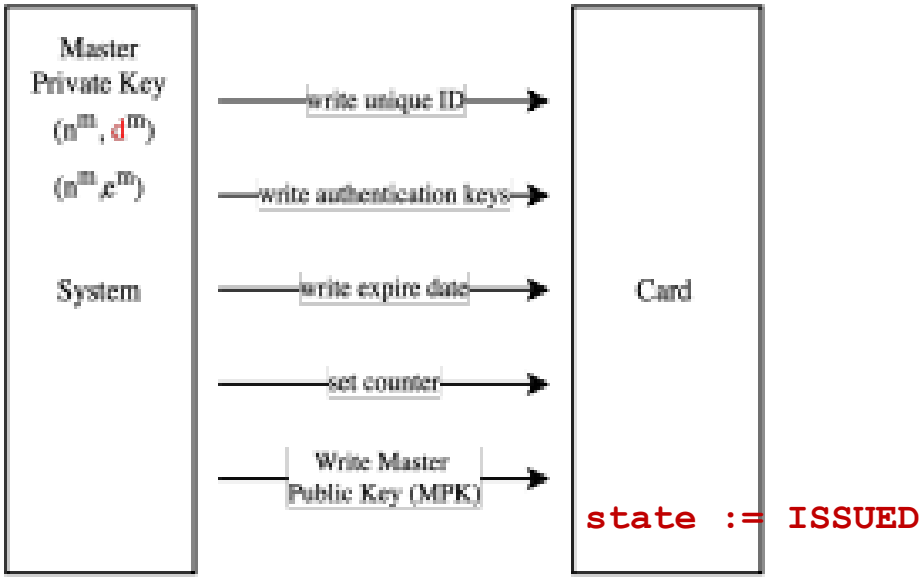

Figure 1: Initialisation protocol

good to add explicit state change

Good to also make other actions explicit

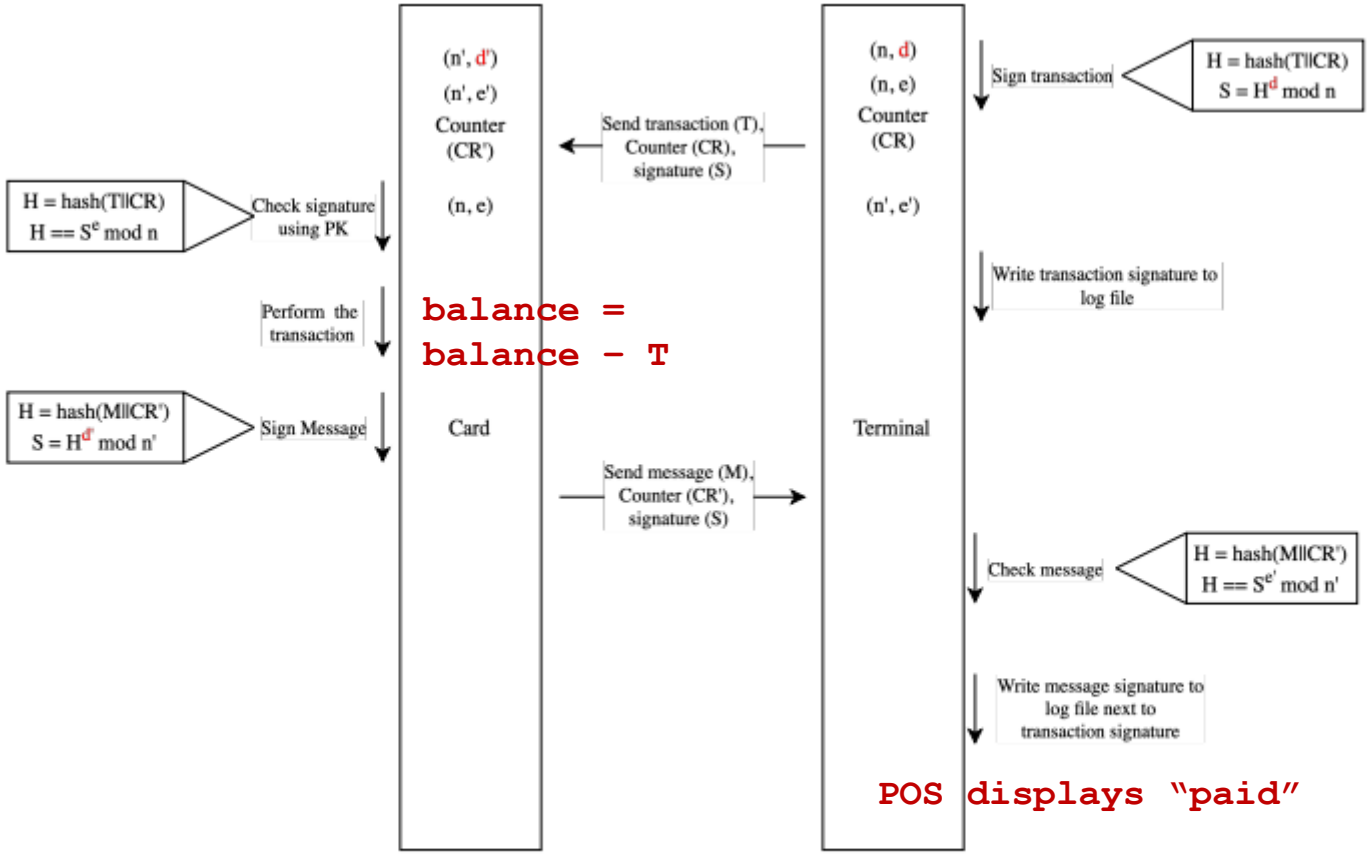

Figure 4: Signing protocol for POS terminals

Spot the security flaw

1. $Terminal \rightarrow Card : CHANGE\_BALANCE$
2. $Card \rightarrow Terminal : current\_balance, Sig_{SK_c}(current\_balance, n_t)$
3. Terminal checks the signature and determines by how much the balance can and should be changed.
4. $Terminal \rightarrow Card : balance\_change, Sig_{SK_t}(balance\_change, n_c)$

Message 4 includes a nonce n_c exchanged in mutual authentication phase.

This ensures 'freshness' of the message / ties it to the current session.

But.... it can be replayed within the same session!

Solutions?

1. **Card could set a Boolean flag and refuse a second CHANGE_BALANCE OPERATION or**
2. **Card could change (e.g. increment) the value of n_c**

Good to include such actions as explicit steps

Spot the potential security flaw with step 6

1. $Terminal \rightarrow Card : CHANGE\_BALANCE$
2. $Card \rightarrow Terminal : current\_balance, Sig_{SK_c}(current\_balance, n_t)$
3. Terminal checks the signature and determines by how much the balance can and should be changed.
4. $Terminal \rightarrow Card : balance\_change, Sig_{SK_t}(balance\_change, n_c)$
5. Card checks the signature and changes its balance if the signature is valid.
6. $Card \rightarrow Terminal : new\_balance, Sig_{SK_c}(new\_balance, n_t)$
7. Terminal checks the signature and if the new balance is the expected balance.

A MitM attacker could replay message 2 as message 6.

Including more information in the signature to distinguish message 2 and 6 would prevent this. Ideally, all signed messages have unambiguous meanings and can never be reused in the wrong place.

Also, what happens if we do a card tear before step 6?

Card tears messing with things are unavoidable, but check their security impact!

Generic improvement: sign *everything*

4. $Terminal \rightarrow Card : balance\_change, Sig_{SK_t}(balance\_change, n_c)$

Instead of just signing

balance_change, n_c

the protocol becomes more robust if we include signature over

BALANCE_CHANGE, current_balance, n_c

where *BALANCE_CHANGE* is some constant byte for this specific message/operation

WHAT vs HOW

Clearly separate

- **the security requirement (WHAT)**
- **the way in which that requirement is ensures (HOW)**

(SR1.4): Authentication of a terminal (reload or POS) to the E-Purse must be done using a challenge-response protocol where the E-Purse sends a challenge in the form of a random number and the terminal sends as response the encrypted random number.

If you don't, it is easy for two to get mixed and for design decisions to become implicit.

Eg why not use a counter instead of random number?
Or maybe the terminal sends more data, including a nonce, that is signed (rather than encrypted)?

Use case: lost or stolen cards

What happens if cards get stolen or lost?

Reporting a card as stolen or lost would be a separate use case

Decision not to have procedure for this deserves to be explicitly stated & motivated.

Blocking cards

'Blocking a card' can be an overloaded term, as it may mean

a) blocking a card itself

e.g. setting a EEPROM `state` flag on the card to `BLOCKED`

b) blocking a card in the back-end

e.g. setting some flag in the back-end database

You may have either or both forms of blocking.

Spotting protocol flaws

- Authentication MUST use some form of challenge-response
  - Just exchanging & certificates is not enough!
  - The challenge has to be a nonce,
    which can be a random number OR a counter

- Double-check that message that triggers the actual transaction cannot be replayed within a transaction

- Beware of unauthenticated responses
  eg a card or terminal saying `OK`

Spotting protocol flaws/improvements

- If you have a session key, it is dangerous to let only one party decide the session key
  - Better - or necessary – to let both parties contribute randomness
- MACing or signing data with *long-term* (private) key provides a stronger guarantee than MACing with a *session* key
- If you use encryption in your protocol, double-check if there is a corresponding security requirement about confidentiality
  - Unless it is encryption of a nonce for authentication, of course

Certificates

A certificate is not a just a signed public key,
it is a signed blob of information that *includes* a public key

A typical certificate will be

$(id \,||\, PubKey_{id} \,||\, expiry\text{-}date \,||\, type\text{-}info \,||\, \ldots)$

signed by a public master key

Or, more formally,

$Cert_{id} = Signed_{PubKeyM}(id \,||\, PbK_{id} \,||\, expiry\text{-}date \,||\, \ldots)$

where

$Signed_{PK}(m) = m \,||\, Enc_{PK}(hash(m))$

Potential improvement?

A certificate consists of its content, and the signature over its content, generated with the private key of the issuer. For example, the smart card certificate $\text{Cert}_{SC} = (\text{ID}_{SC}, \text{expiry}_{SC}, \text{Pub}_{SC}, \text{Sign}_{\text{Priv}_{PT}}(\text{ID}_{SC}, \text{endoflife}_{SC}, \text{Pub}_{SC}))$

Presumably `expiry` is the same as `endoflife`

Is there a way to distinguish the certificate of a smartcard and the certificate of terminal?

Could an attacker steal a certificate and private key from a terminal and use that info make a fake card, or vice versa?

You could use different ranges of IDs for cards & terminals, but clearer to include an explicit byte field in the certificate to indicate a type

IDs

- If a card (or terminal) has its own keypair, then you can use public keys to identify that card.

- But it is much cleaner to give cards and terminals unique identifier *cid* and *tid* as well as own keypairs
  - You might want to have customer-id's *and* card-id's

Defense in Depth

What if…? one of your security assumptions is broken

Would you be able to detect it if

- *a malicious insider issues loads of cards?*
- *a malicious POS operator gives away free points or redeems non-existent points?*
- *a malicious shop owner claims too much money, eg by duplicating transactions?*
- *a card is cloned?*
- *key material from a terminal leaks?*
- *a terminal is hacked to compromise its behaviour?*
- *…*

Logging & procedures to inspect logs can help

Stylistic advice

Avoid duplication

Duplication is bad in code, but also in text,

so avoid it in your report

when describing protocols, giving definitions, discussing attacker models, listing security requirements, ...

- **It is better to have fewer SRs than many SRs**
  - **so avoid duplicating or overlapping security requirements**

Notation

- Be aware of the difference between
  - constants, e.g. `BLOCKED` , `OK OR`, `CHANGE_BALANCE`
  - program variables, e.g. `state`
  - meta-variables for *values* used in protocols

    e.g. *amount* , *card_id* , *terminal_id* , or *PIN_guess*

  Different fonts & capitalisation can help to distinguish them

- Be aware of different meanings of = which include
  - mathematical definitions

    $EncSign_{K1,K2}(m) =_{def} Enc_{K1}(m) \,||\, Enc_{K2}(hash(m))$
  - assignments in code

    `state := PERSONALISED`

Notation

- **Introduce convenient mathematical functions & notation**

  Eg

  $m = Encrypt_K(amount \,||\, card\_id \,||\, nonce)$

  $(amount, cid, time) = Decrypt_K(payload)$

  $m_2 = DecryptAndCheckSignature_{K1,K2}(m_1)$; abort if signature incorrect

- **Numbering steps in protocols can be useful**
  - also when you start coding