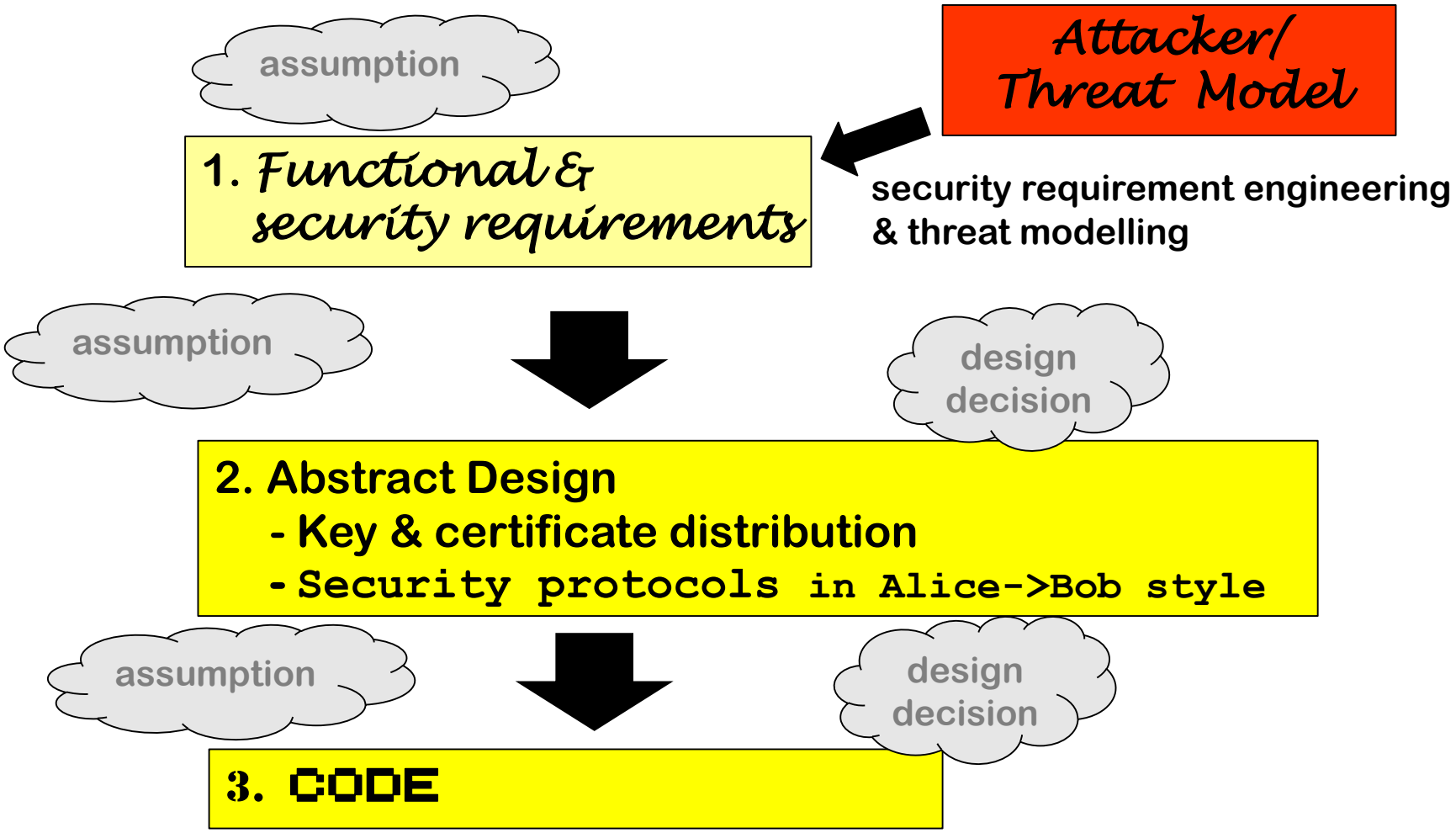
Design Process

Digital Security

Radboud University Nijmegen

The three levels of abstraction



Ingredients for your design document

Description of the first 2 levels,
ie requirements & abstract design,
with clear relations between them

This involves thinking about

- Use cases
- Security requirements, incl.
  - Attackers & threat model (maybe abuse cases ?)
  - Trust assumptions (incl. definition of the TCB and trusted insiders, etc.)
- Abstract protocols to provide the required functionality & security

Ideally making all design decisions explicit.

Later on, at the code level, there will be with further design decisions.

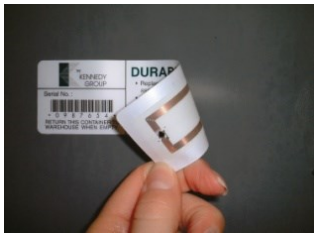

Security requirements
Threats
Trust assumptions

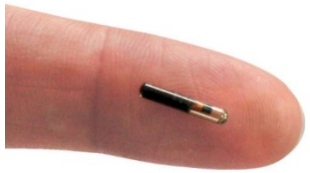
Tip to articulate security requirements: Remember the old-fashioned alternatives

and the security guarantees they make

Differences & commonalities?

1) W.r.t. functionality?





2) W.r.t. security?

Threats vs attacks vs risks

Often used interchangeably, but there are different meanings

- Threat

  Something bad that may happen, an attacker's goal
- Attack (or attack vector)

  A particular way to realise that a threat
- Risk

  $\sum_{\text{attacks}}$ probability * impact

To distinguish these, note that

- There may be *different* attacks to achieve the *same* threat
- Threats never really go away, no matter how good the defences, but *risk* of threats can be reduced.
  Some attack vectors do go away with a certain design.
- Even if we cannot prevent some attack,
  we can detect & react to them, which reduces their risk

Attacker/threat modeling

Threat / attacker model describes of

1. the attacker's capabilities
    - knowledge, skills, expertise
    - (physical or logical) access to places, systems, info
        - insiders? malicious clients? …
    - time & money to buy equipment, expertise, or bribe people
2. the attacker's motivation/goals
    - ie. the bad things you do not want to happen

Complementary notion: trust assumptions describe which systems or agents we trust for some property

- because we *want* to (eg because the risk is small, or because it simplifies the design), or because we *have* to

Use cases: incl. *personalization, issuance, and end-of-life?*

- Cards need to be personalised
  - installing software, initialising keys, PIN codes, IDs, names, …

  before issuance to the user (aka card holder)

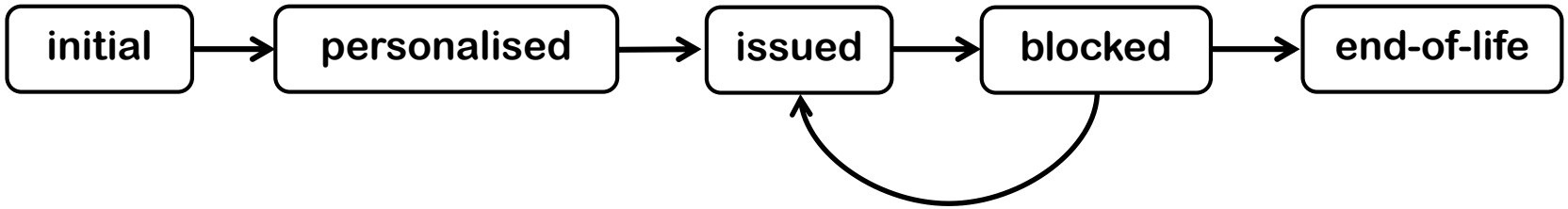
  This will typically require a separate terminal
  - In addition to say point-of-sale terminal,…
  - This may happen in several stages

- Cards may also need to be disabled, eg. at the end-of-life?
  - Or still be able to report data for fraud investigations?

Be explicit about the life-cycle of the card, eg with a state diagram

Persistent life cycle state

Card always has to record some life cycle state



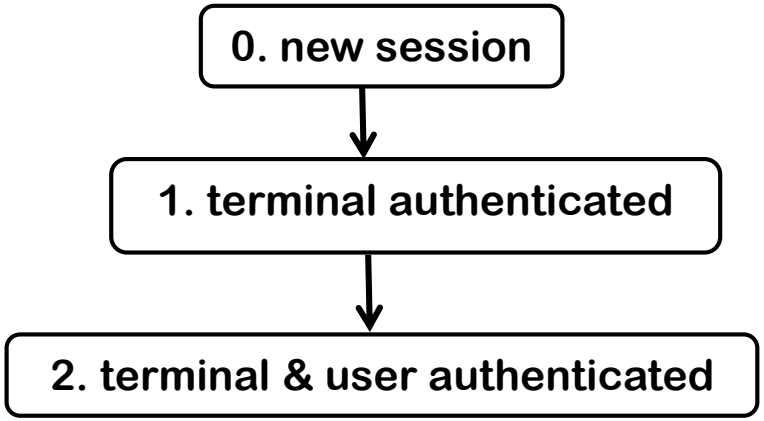
This state has to be recorded & maintained in persistent memory (ie EEPROM)

Your report MUST include a state machine like this!

Transient protocol state?

Cards and terminals may need to maintain transient protocol state for each 'session'

Eg



with some actions only allowed in state 1 or 2

Such constraints may be enforced by cryptographic relations between messages. If not, they may need to be enforced by recording the protocol state in transient memory (ie RAM)

Example security requirements

Eg

- **authentication of the card holder**
- **authentication of smart card**
- **authentication of all communication by party A**
- **confidentiality of PIN code**
- **non-repudiation**

What is wrong with these?

Example security requirements

Eg

- **authentication of the card holder**
- **authentication of smart card**
- **authentication of all communication by party A**
- **confidentiality of PIN code**
- **non-repudiation**

What is wrong with these?

Example security requirements

OF WHAT?

BY WHOM?

TO WHOM?

Example security requirements

- Authentication of the card holder *by the card*
  - Or *by the terminal?*
  - Or *by the back-end?*

- Authentication of all communication by party the smartcard
  - *up to the terminal?*
  - *up to the backend?*

- Confidentiality of data Z so that only parties A and B can read it

Authentication (of entities or of data)

Beware of the subtle differences, eg between

1. authentication of smart card by the terminal
2. authentication of an individual message sent by the smartcard
3. authentication of the entire communication session from smartcard

2 does not prevent replay of an individual message, but 3 does.

Authenticity and *freshness* needed to prevent replays.

Extra tricky: (non-)repudiation

- In Dutch: (on)weerlegbaarheid
- Tricky to express & potentially confusing!

  Non-repudiation of some action X by some party B to another party A

  is more clearly be expressed as

  B can prove to A that action X took place

  or *B can prove to A that C agreed to action X*

  or *B and C cannot deny to A action X took place*

Non-repudiation to reduce trust

Non-repudiation can be crucial to reduce the TCB

for some *specific* security property

Eg

- **Bank may not want to trust POS terminals in shops**
  - **for which properties?**
  - **for which properties is this unavoidable?**

- **Government may not want to trust equipment at petrol stations**
  - **for which properties?**
  - **for which properties is this unavoidable?**

Don't forget detection & response

It's natural to focus on preventing security problems,
and forget about detecting or reacting to problems

- Logging can be useful - or crucial - here !
- Note that logs can serve different aims, eg
  - detecting that things go wrong
  - doing forensics or rolling back transactions in case things went wrong

The ability to detect problems or to determine where things went wrong can be important security requirements!

- Eg. even if you cannot prevent some forms of insider attack, you want to be able find out who or what was responsible if & when these come to light

Example design decisions

It should be crystal clear that your implementation is 'secure'

- ie. that all security requirements are ensured in the design and in the code, by some design decisions

Eg

- authentication of the card holder by party B using a PIN code
- authentication of party A by party B using private/public keys & certificate that have been distributed as follows and the following challenge-response protocol: …
- non-repudiation of action X by having MAC or digital signature over data Z using key K

Pitfall: HOW vs WHAT

It is easy to mix up

- *WHAT* security requirement you want to meet

  eg

  - authentication of party A by B
  - authentication of message M
  - …

- *HOW* you meet that security requirement

  eg

  - some challenge-response protocol between A & B
  - some digital signature or MAC on message M
  - …

Keep these separate, so that difference between security requirements (WHAT) and design decisions (HOW) is clear

Include overview of key & certificate distribution!

Who has which keys & certificates for what purpose?

For example

- The smart card has keys SK_C, PK_C, PK_{master} and certificate C_{cardID}
  The terminal has keys K1, K2 and certificate C_{termID}

- Key AK_C is used to authenticate messages sent by the card
- Certificate C_{ID} signed by X proves that …
- Key EK is used to encrypt …

NB it is bad practice to use the *same* key for *different* security goals (eg integrity *and* confidentiality)

Logical chain of motivation

Ideally assets & attacker model

lead to

security requirements

lead to

design decisions - use of keys, protocols, PINs, ...

In practice, and chronologically, things often happen in the opposite order!

eg you decide 'let's encrypt this', then consider why, and only then discover some security requirement about confidentiality that was implicit so far

That's fine, as long as in the end the motivations and rationale are clear and explicit!

Design process

Typically a combination of

- structured & methodological approach

  using standard lists of security objectives, attacks, etc.

- creative chaos

  brainstorming about attacks, solutions, etc

  - Brainstorming may work best if everyone first tries it on their own, to avoid tunnel vision
  - As usual, thinking like an attacker is the only way to see if a design is secure

Either is fine, as long as the end result is clearly documented, and rationale are clear

Pitfalls

- *Implicit* assumptions
  - Could be invalid, or become invalid over time due to changing circumstances or new functionality (function creep)
  - Better an unrealistic assumption than an implicit one!

- *Implicit* or *unmotivated* design decisions
  - These may hide implicit security requirements or implicit threats, or be totally pointless

- It is fine to ignore some threats because you think the risk is negligible, or because they are to hard to defend against, but say so explicitly

Design choice: symmetric vs asymmetric crypto?

Pros symmetric

- Cost & Speed?
  - Old cheap smartcards could only do symmetric crypto
  - But modern cheap cards can do asymm. crypto, and fast enough, so not really an issue?
- Quantum resistance

Pros asymmetric

- Symmetric crypto cannot achieve proper non-repudiation, as verifying MAC requires access to the same key that created it.
- Key management is much more flexible with asymm. crypto, as you can use PKI & certificates.
- Security benefit: some parties only need to know a public key instead of a shared symmetric key
  - eg some master public key to verify certificates

Key diversification for symmetric crypto

If each card has its own symmetric key, the back-end (and terminals?) has to record all of them

- If we use asymm. crypto, the use of certificates avoids this.

A standard trick to reduce this key management hassle:

- Give terminals and/or central back-end some master key M
- Give card with card number ID a diversified key derived from the master key M and ID, eg.

$$M_{ID} = AES_M(ID)$$

This avoids the need of a database recording the key for every card. Still, the master key ending up in lots of places remains a security weakness.

Pitfalls

Encrypting data does not ensure integrity!!

- Attacker can flip bits in $encrypt_K(M)$ with unpredictable – and undetectable – result

- The only robust way to ensure the authenticity of a message M is to append a Message Authentication Code (MAC) or a digital signature, ie. an encrypted hash of the message M

Practical considerations

- Model & implement as little of the back-office system as possible
- Don't forget about personalisation & issuing of smartcards
  - This will require another terminal application
- Steal as many standard solutions as possible
  - eg. crypto protocols, key management, key diversification
- It is OK to have security flaws, or cut corners because you accept certain risks, as long as these are documented!
  - esp. in the rush to meet the final deadline, you may have to cut corners

DEADLINE

- **Feb 18: high-level design document**

- **You can read back what I told you in the document linked in Brightspace**

Checklist before handing it in

Your report MUST include

- **List of security requirements Description of the life-cycle stages**
- **A clear overview of the keys & certificates used, incl. description of who has which keys & what the purpose of a key is**
- **Security protocols as messages sequence charts or in Alice->Bob style**
  - **Explain notations for signing and encrypting data used here!**
- **Think about non-repudiation requirements, if you haven't mentioned any**
- **Check that you use consistent notation for the keys and parties throughout the document, that is explained**
- **Number sections & pages. Numbering or labelling other things can be useful too: eg. use cases, the steps in a protocol, security requirements, ...**