Defensive Design & Defensive Coding

Erik Poll

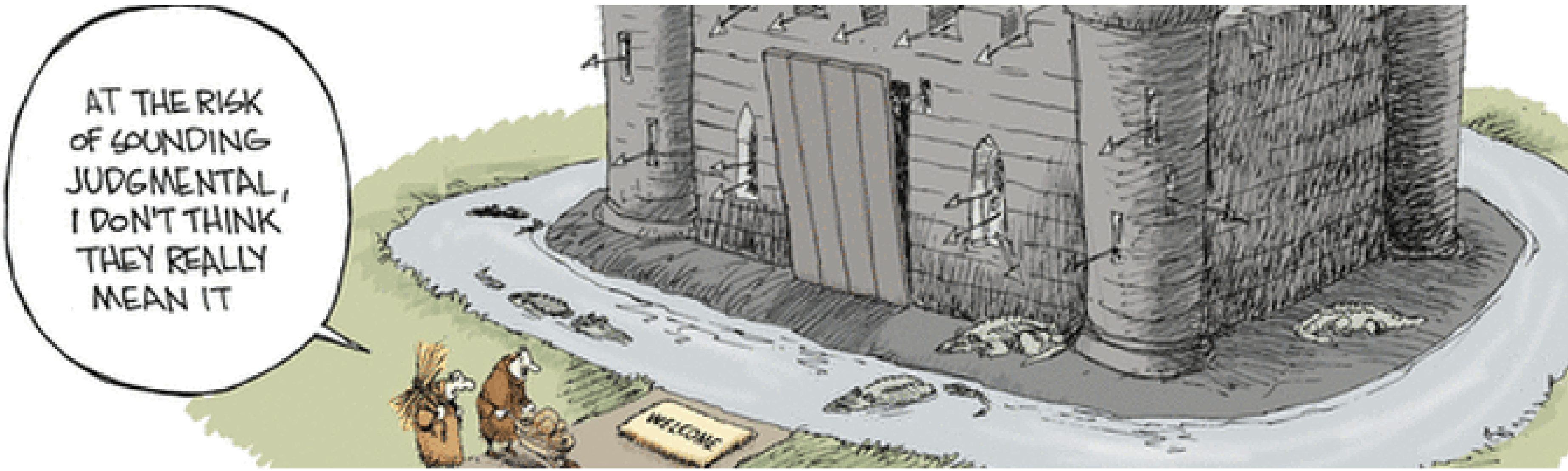
Digital Security

Radboud University Nijmegen

TRU/e Master in Cyber Security

Defensive *

General defensive design principle: **Defense in Depth**



Detection & Reaction

Important example of Defense in Depth:

Don't just try to prevent security problems, also try to detect them & react to them

Detection & Reaction

Important example of Defense in Depth:

Don't just try to prevent security problems, also try to detect them & react to them

There may be different goals with security cameras & other forms of monitoring

a) **detection & response *as it happens***

b) **investigation *after the fact***

What if … ?

Any design involves trust assumptions (e.g. a TCB)

Useful question to ask for defensive design:

What if these assumptions are broken?

Eg: What if

- ***key material of card, terminal, or back-end leaks?***
- ***card is cloned?***
- ***terminal is compromised?***
- ***issuance process is compromised?***
- ***there's an insider attacker with insider in role X?***

Example: Gemalto key leaks

Gemalto Says Alleged Hack Didn't Result in Massive Theft of SIM Keys

Company detects 'sophisticated intrusions' in 2010, 2011

SIM-card company Gemalto said it detected two "sophisticated intrusions" in 2010 and 2011 following a probe into alleged hacks by U.S. and U.K. intelligence agencies.

Note: Gemalto is not a telco but produces SIMs *for* telcos; why should they be hanging on key material anyway?

https://www.wsj.com/articles/dutch-firm-gemalto-investigates-hacking-claim-1424423264

Example: Estonian key generation problem

We report on our discovery of an algorithmic flaw in the construction of primes for RSA key generation in a widely-used library of a major manufacturer of cryptographic hardware. The primes generated by the library suffer from a significant loss of entropy. We propose a practical factorization method for various key lengths including 1024 and 2048 bits.

Despite the general difficulty of obtaining relevant datasets with public keys from passports or eIDs that limited our analysis to only four countries, we detected two countries issuing documents with vulnerable keys. The public lookup service of *Estonia* allowed for a random sampling of the public keys of citizens and revealed that more than half of the eIDs of regular citizens are vulnerable and that all keys for e-residents are vulnerable.

[Nemec at al, The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli, CCS 2017, ACM, https://dx.doi.org/10.1145/3133956.3133969]

Example: Estonian key management problem

In this paper, we describe several security flaws found in the ID card manufacturing process. The flaws have been discovered by analyzing public-key certificates that have been collected from the public ID card certificate repository. In particular, we find that in some cases, contrary to the security requirements, the ID card manufacturer has generated private keys outside the chip. In several cases, copies of the same private key have been imported in the ID cards of different cardholders, allowing them to impersonate each other. In addition, as a result of a separate flaw in the manufacturing process, corrupted RSA public key moduli have been included in the certificates, which in one case led to the full recovery of the corresponding private key.

[Arnin Parsovs, Estonian Electronic Identity Card: Security Flaws in Key Management, USENIX Security 2020]

Some defensive design tricks

- Having *both* parties provide nonces always makes attack harder
  - even if one of these parties has to be 'trusted' for some security guarantee
- Including *more* info in (hash used to construct) MAC or signature always makes attacks harder
  - eg card or terminal IDs, sequence no, time stamp,…
- Advantages of *counters* over nonces:
  - avoids risk of crappy RNGs
  - makes checking for repeated of nonces in backend easier
  - may make other forms of compromise detectable
    - Eg, if card includes a counter in the receipts it signs, then existence of cloned cards is detectable in back-end

Moral of the story

it’s good to be paranoid!

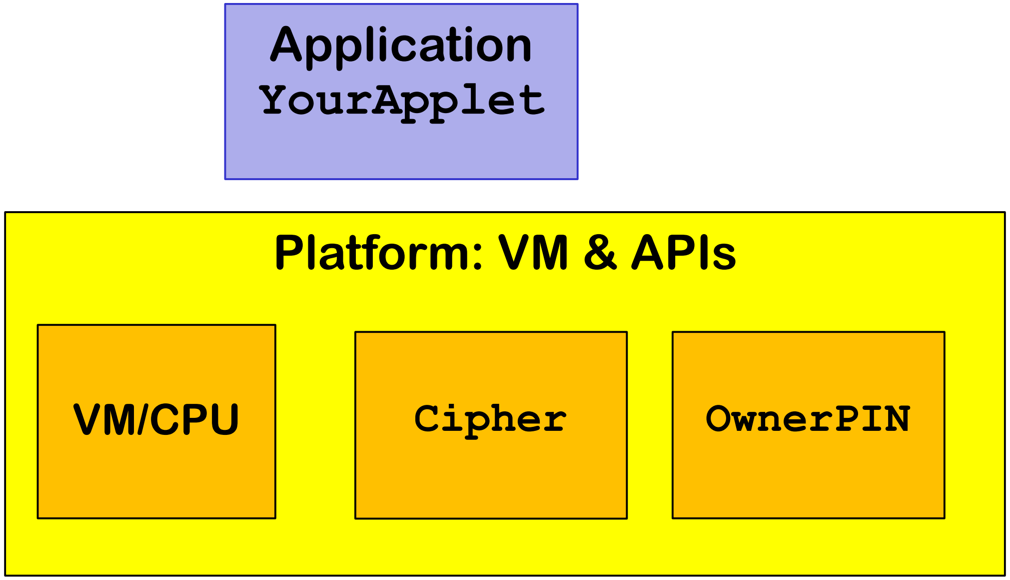
“Just because you're paranoid doesn't mean they aren't after you.”

-- Joseph Heller, Catch 22

Defensive Coding against side channel attacks

Side-channels at application level?

As an application programmer
 (ie. if you only *use* crypto APIs, and do not *implement* them)
should you care about side channel attacks?



Confidentiality vs Integrity

Side channels can be a threat to

1. confidentiality
   - *eg leaking cryptographic keys or PIN codes*
   - *passive* attacks *by observing timing, power, EM, …*
2. integrity
   - *eg corrupting a cryptographic key*
   - *active* attacks to inject faults *by card tears, glitching, lasers, clock frequency, temperature, …*

Confidentiality vs Integrity

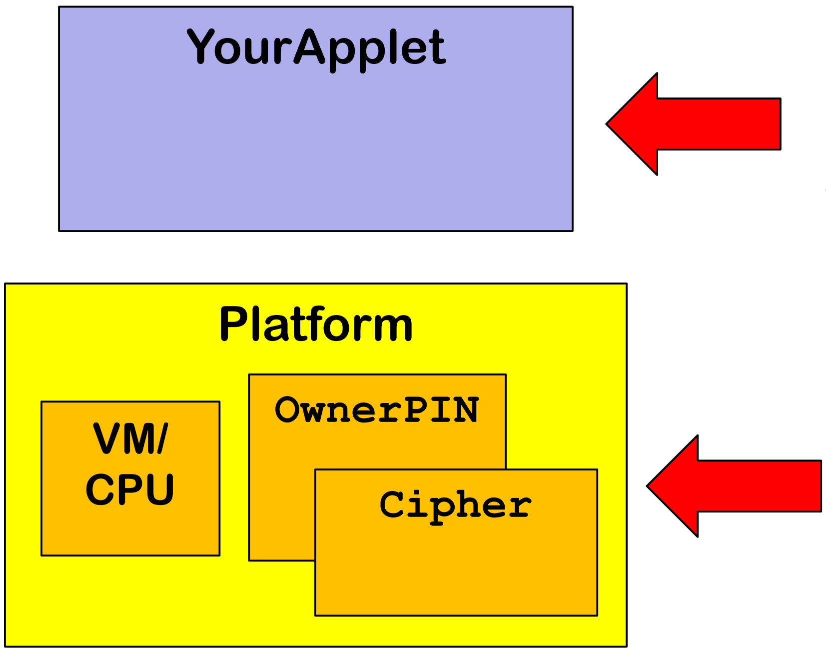
Side channels can be a threat to

1. confidentiality
   – *eg leaking cryptographic keys or PIN codes*
   – *passive* attacks *by observing timing, power, EM, …*
2. integrity
   – *eg corrupting a cryptographic key*
   – *active* attacks to inject faults *by card tears, glitching, lasers, clock frequency, temperature, …*

1 is a concern for confidential data, eg keys & PINs, so concern for implementation of crypto & handling PINs

2 is a concern for *any (security-critical) data and code*

What to attack & how ?

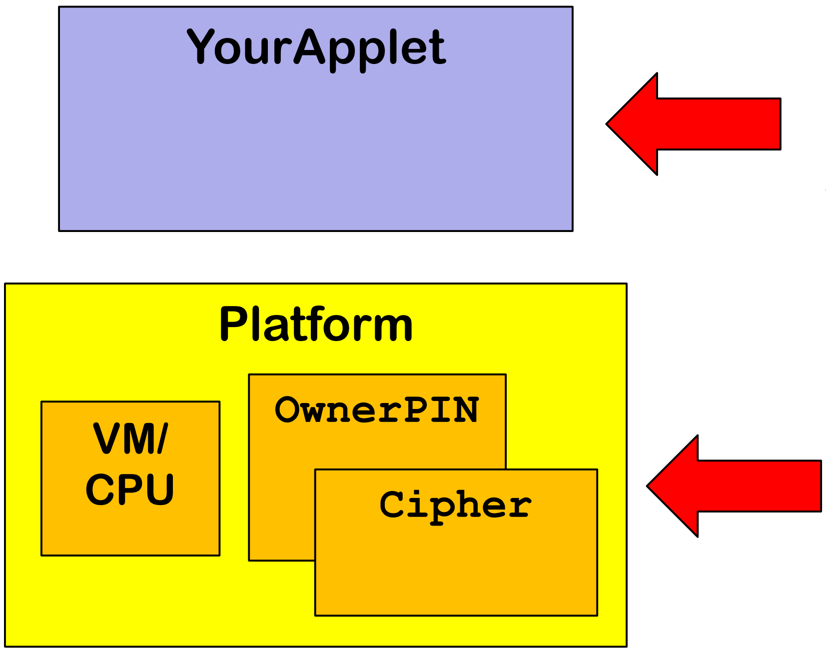


Fault injection to corrupt integrity of data or of execution

Attacks to extract confidential info less of a concern

Any attack (passive of active) to extract keys, PINs, ...

What to attack & how ?



Fault injection to corrupt integrity of data or of execution

Attacks to extract confidential info less of a concern

Any attack (passive of active) to extract keys, PINs, …

If platform APIs are well-protected, applet developers still have to worry about esp. active side channels

This requires knowledge or assumptions about which faults are possible & what their effect is on VM/CPU and APIs

- This goes against the whole idea of the JavaCard platform hiding low level details…

Fault injections

- Card Tears
- Physical
  - putting a 0 or 1 on a databus line
- Glitching (late 1990s)
  - briefly dipping voltage of power supply
  - affects memory but also functionality
  - difficult to do nowadays to corrupt data; but skipping instructions on a Java Card VM may be possible!
- Light manipulations (early 2000s)
  - light flash on chip surface affects its behaviour

Fault injections: practical complications

Many **parameters for the attacker to play with**

- *when* to do a card tear
- *when* to glitch*; how far to dip the voltage*; for *how long*
- *when* & *where (x and y dimension)* to shoot a laser;
  for *how long; how strong;* and *which colour?*
- *Multiple faults?*

  Multiple glitches are possible, multiple laser attacks harder

This can make fault attacks a hit-and-miss process for the attacker (and security evaluator)

Attack targets of fault injections

- Attacks can be on data or on code
  - including data and functionality of the CPU,
    eg the program counter (PC)

- Code manipulation may
  - turn instruction into NOP
  - skip instructions
  - skip (conditional) jumps

    Default behaviour of CPU is to increment program counter

- Data manipulation may result in
  - special values: 0x00 or 0xFF
  - just random values

Attack targets of fault injections

Fault attacks can target

- crypto

  some crypto-algorithms are sensitive to bit flips;

  the classic example is RSA

- any other security-critical functionality

  any security-sensitive part of the code or data can be targetted

Physical vs Logical Countermeasures

Physical countermeasures

- Prevention – make it hard to attack a card
- Detection: include a detector that can notice an attack
  - eg a detector for light or dips in power supply

Logical countermeasures

- Program defensively to not leak info or resist faults
  - For JavaCard, this can be at platform level or applet level

Physical vs Logical Countermeasures

Physical countermeasures

- Prevention – make it hard to attack a card
- Detection: include a detector that can notice an attack
  - eg a detector for light or dips in power supply

This starts another arms race: attackers use another fault attack on such detectors. Popular example: glitch a card and simultaneously use a laser to disable the glitch detector!

Logical countermeasures

- Program defensively to not leak info or resist faults
  - For JavaCard, this can be at platform level or applet level

Spot the bugs/potential weaknesses

```
class OwnerPIN{
 boolean validated = false;
    short tryCounter = 3;
   byte[] pin;

boolean check (byte[] guess) {
     validated = false;
     if (tryCounter != 0) {
          if arrayCompare(pin, 0, guess, 0, 4)
                { validated = true;
                  tryCounter = 3;}
          else {tryCounter--;
                ISOException.throwIt(WRONG_PIN); }
      else ISOException.throwIt(PIN_BLOCKED); }
```

Basic defensive coding for OwnerPIN

- Checking & resetting PIN try counter in a safe order
  - to defeat card tear attacks
- `validated` should be in transient memory
  - ensuring automatic reset to false
  - only way to do this in JavaCard: make it a transient arry of length one
- Does timing of `arraycompare` leak how many digits of the PIN code we got right?
  - read the JavaDocs for `arraycompare` !

Getting more paranoid: data integrity

What if attacker can corrupt data?

- Checking for illegal values of `tryCounter`
  - eg negative values or greater than 3
- Redundancy in data type representation
  - eg record `tryCounter*13`

    or use an error-detecting/correcting code
- Keeping two copies of `tryCounter`
- Even better: keep one of these copies in RAM

  where RAM copy is initialised on applet selection

  Why is this better?

  Attacker must attack both RAM & EEPROM, and synchronise these attacks

Getting more paranoid: data integrity

- Suppose the VM represents Boolans as follows
  - 00 is `false`
  - all other values 01..FF represent `true`

  – *Why is that potentially a dangerous choice?*

    If attacker can corrupt data,
    Booleans are likely to turn `true`

- Better choice: representing `true` as 85, `false` as -86 and throw a SecurityException for any other values
  – *Why are 85 and -86 good choices ?*

    Binary representations 01010101 and 10101010
    have equal Hamming weight

Getting more paranoid: data integrity

- Avoiding use of special values such as 00 and FF
- Use restricted domains and check against them
- Introduce redundancy

  when storing data or when performing computations

  Eg

  - doing the same computation twice & compare results
  - for asymmetric crypto: use the cheap operation to check validity of the expensive one

Getting more paranoid: control flow integrity

What if attacker can corrupt control-flow?

Eg with glitching, causing the card to skip instructions

- **Doing security-sensitive checks twice**
- **Changing order of tests**
  - **thinking of how this looks in bytecode**

```
if (pinOK)  { // allow access
                    ...
            }
      else { // error handling
                    ...
            }
```

Better

```
if (!pinOK) { // error handling
                   ...
            }
       else { // allow access
                   ...
            }
```

Better to branch (conditionally jump) to the "good" (ie. "dangerous") case

if faults can get the card to skip instructions

Even more paranoid

```
if (!pinOK) { // error handling
                      ...
            }
       else { if (pinOK) {
                    ...
              }
              else {
                // We are under attack!
                // Start erasing keys
                   ....
            }
```

Getting more paranoid: control flow integrity

- Add control flow integrity checks

  eg by setting flags in the code to confirm integrity of the execution run

```
       byte b = 0x01;
S1;    b = b || 0x02;
S2;    b = b || 0x04;
S3;    b = b || 0x08;
S4;    if (b!=0x0F) {// under attack!
                      ...
                     }
```

Beware: a clever compiler might optimise way a this code!

JavaCard API could be extended with `beginSensitive()` and `endSensitive()` to turn on such countermeasures in the VM

SensitiveResult (new in Java Card 3.0.5)

VM maintains a special variable records the result of the last sensitive method call, to easily double-check it without invoking the method twice

```
boolean res = signature.verify(...); // sets this variable
if (res) {
        SensitiveResult.assertTrue(); // double-checks it
        // Grant service
} else {
        SensitiveResult.assertFalse();
        // Deny service
}
```

https://docs.oracle.com/javacard/3.0.5/api/javacardx/security/SensitiveResult.html

SensitiveArrays (new in Java Card 3.0.5)

This class provides methods for creating special arrays with built-in (unspecified) integrity checks

- The integrity check could be a check-sum

  For example

```
makeIntegritySensitiveArray(ARRAY_TYPE_BOOLEAN,
                            MEMORY_TYPE_TRANSIENT_RESET,
                            56)
```

  creates transient sensitive array of booleans with length 56.

- Violations of the integrity check result in a `SecurityException`

https://docs.oracle.com/javacard/3.0.5/api/javacard/framework/SensitiveArrays.html

When & What to code defensively?

- First step: decide
  - which data
  - which parts of the execution

  are sensitive and should be protected?

- Program annotations can help with this, to mark sensitive data or operations
  - eg using Java annotation mechanism, introducing a tag like `@Sensitive`

Defensive coding tricks: information leakage

- Make sure code executes in constant time
- Make sure error messages do not leak interesting info
  - Do the two different error codes in OwnerPIN, `WRONG_PIN` and `PIN_BLOCKED`, leak interesting information ?

    Probably not, but error messages are notorious for leaking info

This is not really a *side*-channel attack,

but a (normal I/O) *channel* attack

Coding trick to remove timing sensitivity

Replace

```
if (b) then { x = e} else {x = e';}
```

with

```
a[0]= e;
a[1]= e';
x = b ? a[0] : a[1];
```

to remove timing sensitivity

- at expense of efficiency, obviously..

Example information leaks: e-passports

- **Modern passports contain an ISO 14433 contactless smartcard**
- **Specs defined by ICAO**

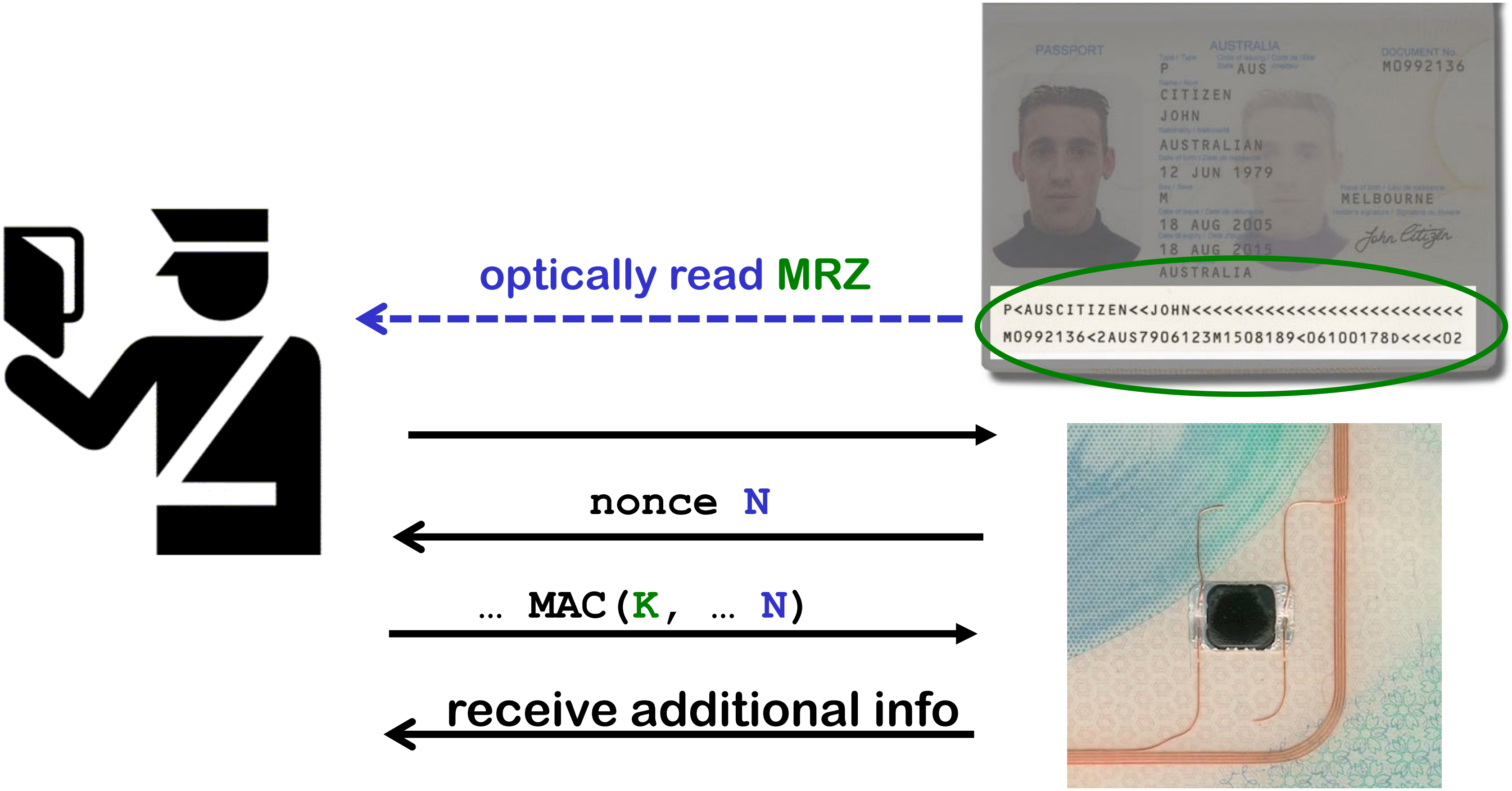
- **Our open source JavaCard implementation of the spec is at http://jmrtd.org**
- **If you have an NFC Android phone, you can read out the chip with the ReadID - NFC Passport Reader app**

e-passport security measures

- Protocols have been carefully designed to prevent information leakage
- The e-passport only provides information after reader proves knowledge of the Machine Readable Zone (MRZ)
  - using BAC or PACE protocol

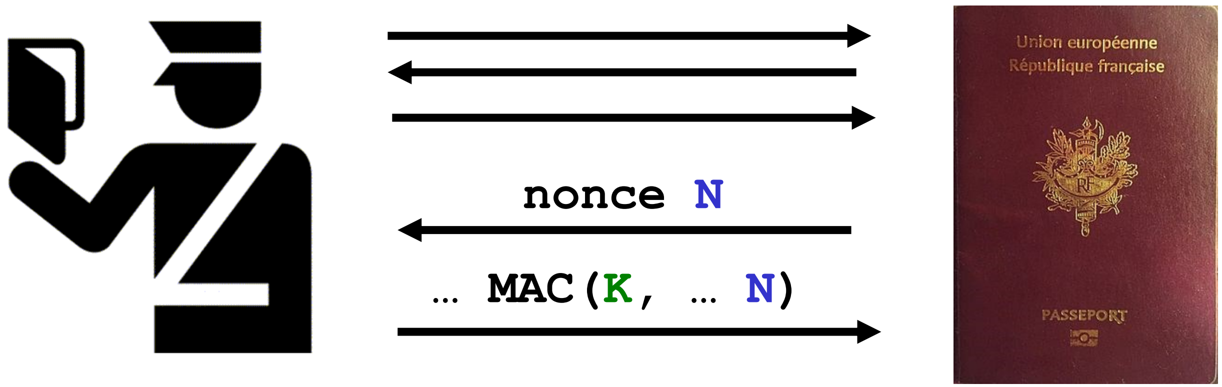
MRZ

e-passport access control (using BAC or PACE)



Information leak through error messages

e-passport protocol:



French passports report different error messages when the MAC is wrong and when the nonce is wrong.

The key K is unique for a specific passport, so replaying an old message now identifies that specific passport

Passports from some (all?) other countries expose this through a timing leak

[Tom Chothia & Vitaliy Smirnov, A Traceability Attack against e-Passports, FC 2010]

Earlier information leak found

Response to B0 "read binary", and is only allowed after BAC

	status word	meaning
Belgian	6986	not allowed
Dutch	6982	security status not satisfied
French	6F00	no precise diagnosis
Italian	6D00	not supported
German	6700	wrong length

All passports we had, from different countries, could be distinguished from the error responses to 3 APDUs

[BSc thesis of Henning Richter, 2008]

deVerdieping
Trouw
dinsdag 8 april 2008
66ste jaargang nr 19447
www.trouw.nl

• Legio criminele toepassingen

Na ov-chip nu ook lek in paspoort

De chip in het nieuwe Nederlandse paspoort en andere passen is 'lek'. Dieven kunnen snel zien of iemand een paspoort bij zich heeft en uit welk land hij komt.

Vincent Dekker

Moderne paspoorten in tassen of binnenzakken verraden draadloos hun aanwezigheid én uit welk land ze ko-

antwoorden op elke correcte vraag van een officieel leesapparaat, zoals bij de douane. Maar men is vergeten dat ook te regelen voor antwoorden op verkeerde vragen. In de praktijk blijkt dat elk land een eigen manier heeft bedacht om met foute codes om te gaan. Analyseer de foutmelding die je terugkrijgt na het bewust versturen van een verkeerde code en je weet uit welk land het paspoort komt."

Foutmeldingen verraden veel over de werking van computers en zijn al

Olympische fakkeltocht wacht in San Francisco volgend protest

Na Londen ontaardde ook in Parijs de olympische fakkeltocht door Tibetprotesten in chaos. De volgende steden maken hun borst al nat.

Van onze redactie buitenland

De olympische vlam verliet gisteravond Parijs, op weg naar de volgende bestemming: San Francisco. Maar sommige officials beginnen zich vanwege alle Tibetprotesten af te vragen of de estafette wel door moet gaan.

De route van de vlam door Parijs werd gisteren ingekort. De protesten tegen het Chinese ingrijpen in Tibet veroorzaakten dermate veel chaos dat de fakkel liefst vijfmaal gedoofd moest worden – volgens de organisatoren één keer vanwege een defect en vier keer uit voorzorg. De olympische vlam bleef volgens hen wel permanent branden in een busje. Maar

Passport bombs?

http://www.youtube.com/watch?v=-XXaqraF7pI

Your project code

- For your projects, you do not have to do program defensively to withstand faults
  - except that you have to resist card tears

- So you do ***not*** have to add your own integrity checks on data stored on the card; for this you may simply trust the card

Security by Obscurity rules!

Knowing the code of an implementation,
or the layout of data in memory,
really helps an attacker with fault attacks!

Obscurity makes the life of the attacker harder!

E.g. open source code will be harder to glitch than closed source code ...

Security by Obscurity rules!

Knowing the code of an implementation,
or the layout of data in memory,
really helps an attacker with fault attacks!

Obscurity makes the life of the attacker harder!

E.g. open source code will be harder to glitch than closed source code …

SPA has been used to leak the code of a Java Card

- ie. the power signal betrays which bytecode is instructed.
  (A single bytecode instruction takes many machine instructions.)
- The code is not confidential, per se, but this can help an attacker

Side channels going mainstream

Side channel attacks used to be the concern for embedded security, esp. smartcards

- **where attacker has physical access to do side-channel attacks on confidentiality (eg DPA) or integrity (eg glitching & light attacks)**

Side channels going mainstream

Side channel attacks used to be the concern for embedded security, esp. smartcards

- where attacker has physical access to do side-channel attacks on confidentiality (eg DPA) or integrity (eg glitching & light attacks)

However, in the last decade this went mainstream due to

- micro-architectural attacks
  Spectre & Meltdown on confidentiality
- RowHammering as fault injection attack to comprise integrity

Side channels going mainstream

Side channel attacks used to be the concern for embedded security, esp. smartcards

- where attacker has physical access to do side-channel attacks on confidentiality (eg DPA) or integrity (eg glitching & light attacks)

However, in the last decade this went mainstream due to

- micro-architectural attacks
  Spectre & Meltdown on confidentiality
- RowHammering as fault injection attack to comprise integrity

The attacker model is different here: not a physical attacker, but a malicious execution thread