

Reverse engineering smart cards

Christian M. Amsüss

`linuxwochen@christian.amsuess.com`

`http://christian.amsuess.com/`

2010-05-06

Overview

objective understand smart card communication based on sniffable communication

hardware standard card reader

software something that can talk to the smart card (typically in emulator), `cat /dev/usbmon0`, some own tools

Smart card basics

Practical examples

Smart card basics

Practical examples

Common cards and readers



Low level

- ▶ shape and contacts defined in ISO 7816-1 and -2
- ▶ contacts for ground, power, reset, clock, and I/O
- ▶ serial communication
- ▶ ATR: answer to reset (up to 33 byte)
- ▶ protocol T=1 for sending and receiving byte string messages

High level

```
1 > 00 a4 00 00 02 01 02
2 < 90 00
3 > 00 b0 00 00 00
4 < 00 00 02 14 90 00
```

- ▶ command/response dialogue
- ▶ command = APDU, consisting of
 - ▶ CLA (usually 00, other values indicate proprietary commands or RFU)
 - ▶ INS (instruction, eg. a4 = "Select File")
 - ▶ P1, P2 (arguments, eg 04 00 = "Select by DF")
 - ▶ length and data, depending on INS
- ▶ response, consisting of
 - ▶ data, depending on INS
 - ▶ SW1, SW2 (return code, eg 90 00 = "OK")

Interfaces and drivers

CCID standard for USB card readers

PC/SC Windows API for smart cards

PCSC-Lite the same interface on Linux and OS X

OpenSC library focused on crypto (PKCS#x), brings some own drivers

libchipcard library focused on not blocking unused devices

carddecoders my tools and example programs for smart card reverse engineering, based on Python PCSC bindings (<http://christian.amsuess.com/tools/carddecoders/>)

Smart card basics

Practical examples

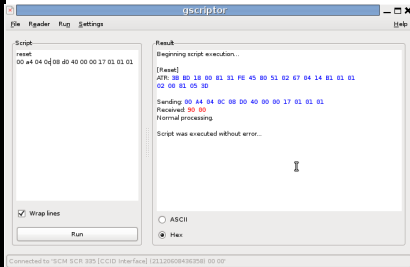
Trying it out: pcsc-tools

```
1 pcsc_scan
PC/SC device scanner
V 1.4.16 (c) 2001-2009, Ludovic Rousseau <ludovic.rousseau@free.fr>
Compiled with PC/SC lite version: 1.5.5
Scanning present readers...
0: SCM SCR 335 [CCID Interface] (21120608436358) 00 00

Wed May 5 17:04:10 2010
Reader 0: SCM SCR 335 [CCID Interface] (21120608436358) 00 00
Card state: Card inserted, Shared Node.
ATR: 3B 8D 18 00 01 31 FE 45 80 51 02 67 04 14 B1 01 01 02 00 81 05 3D

ATR: 3B 8D 18 00 01 31 FE 45 80 51 02 67 04 14 B1 01 01 02 00 81 05 3D
+ T0 = 3B --> Direct Convention
+ T0 = 8D, Y(i) = 1011, K: 13 (historical bytes)
  TR(1) = 18 --> F1=372, D1=12, 31 cycles/ETU
    12932 bits/s at 4 MHz, Max for F1 = 5 MHz => 161290 bits/s
  TB(1) = 00 --> VPP is not electrically connected
  TD(1) = 81 --> Y(i+1) = 1000, Protocol T = 1
-----
  TD(2) = 31 --> Y(i+1) = 0011, Protocol T = 1
-----
  TR(3) = FE --> IFSC: 254
  TB(3) = 45 --> Block Waiting Integer: 4 - Character Waiting Integer: 5
+ Historical bytes: 00 51 02 67 04 14 B1 01 01 02 00 81 05
Category indicator byte: 80 (compact TLV data object)
  Tag: 5, len: 1 (card issuer's data)
    Card issuer data: 02
  Tag: 6, len: 7 (pre-issuing data)
    data: 04 14 B1 01 01 02 00
  Tag: 8, len: 1 (status indicator)
    LCS (life card cycle): 05
+ TCK = 30 (correct checksum)

Possibly identified card (using /usr/share/pcsc/smartcard_list.txt):
3B 8D 18 00 01 31 FE 45 80 51 02 67 04 14 B1 01 01 02 00 81 05 3D
  Austrian "e-Card" (eHealth Card), BRUNNEN (since 06/2005)
  special version of Starcos 3.1
```



- ▶ pcsc_scan
- ▶ (g)scriptor

Sniffing on Linux

- ▶ Software that talks to the card can run in a VM (eg. ActiveX applet)
- ▶ Linux lets you sniff USB communication using `/dev/usbmon0`; output is CCID inside usbmon's binary logging format
- ▶ Workflow:
 - ▶ `sudo cat /dev/usbmon0 > sniffing_run_1.out`
 - ▶ Do something with the card
 - ▶ Stop cat with `^C`
 - ▶ `logdecoder -r sniffing_run_1.out` (from `carddecoders`)

```
1 > 00 a4 00 00 02 01 02
2 < 90 00
3 > 00 b0 00 00 00
4 < 00 00 02 14 90 00
```

Interpreting returned data: Encodings

- ▶ Look for numbers known to be read
- ▶ Big Endian: 02 00 = 512

```
1 > 00 a4 00 00 02 01 02
2 < 90 00
3 > 00 b0 00 00 00
4 < 00 00 02 14 90 00
```

€ 5.32

Interpreting returned data: Encodings

- ▶ Look for numbers known to be read
- ▶ Big Endian: 02 00 = 512
- ▶ Binary Coded Decimal: 12 34 = 1 234

```
1 > 00 a4 00 00 02 3f 00
2 < 90 00
3 > 00 a4 00 00 02 00 02
4 < 90 00
5 > 00 b0 00 00 08
6 < 09 6f 06 70 00 21 20 00 90 00
```

BLZ 12000

Interpreting returned data: Encodings

- ▶ Look for numbers known to be read
- ▶ Big Endian: 02 00 = 512
- ▶ Binary Coded Decimal: 12 34 = 1234
- ▶ ASCII: 31 32 33 34 = 1234

Interpreting returned data: Encodings

- ▶ Look for numbers known to be read
- ▶ Big Endian: 02 00 = 512
- ▶ Binary Coded Decimal: 12 34 = 1 234
- ▶ ASCII: 31 32 33 34 = 1234
- ▶ Other creative encodings for dates etc.

```
1 > 00 b2 01 04 00
2 < [...] 90 00 01 00 05 10 46 01 00 [...]
3 > 00 b2 02 04 00
4 < [...] 90 00 00 93 44 13 31 00 00 [...]
5 > 00 b2 03 04 00
6 < [...] 90 00 00 93 44 13 31 00 00 [...]
```

2010-01-05, 10:46 local time (day 5 of the year '010)

2009-12-10, 13:31 local time (day 344 of the year '009)

Exploring commands

- ▶ Some commands can be bent.

```
1 > 00 b0 00 00 08
2 < 09 6f 06 70 00 21 20 00 90 00
```

According to ISO 7816, the last byte gives the number of bytes to read. Let's assume it works like POSIX's read:

```
1 > 00 b0 00 00 00
2 < 09 6f [...] 95 01 23 66 02 00 [...] 01 90 00
```

Exploring commands

- ▶ Some commands can be bent.
- ▶ Others can be bruteforced.

```
1 > 00 a4 00 00 02 df 01
2 < 90 00
```

This was known to work... Let's try this:

```
1 > 00 a4 00 00 02 df 08
2 < 6a 00
```

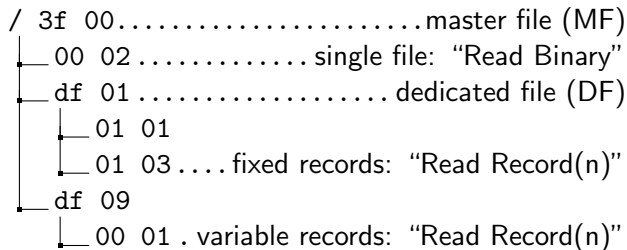
No ... One more?

```
1 > 00 a4 00 00 02 df 09
2 < 6f 14 84 07 a0 00 [...] 54 52 4f 90 00
```

This works, and even sends data immediately.

Card state

- ▶ Smart card directory structure:



- ▶ File selection seems rather safe for experimenting
- ▶ More card state: authentication, challenge/response (limited tries!)

Tools provided by carddecoders

- ▶ logdecoder

Decodes usbmon output to

```
1 > 00 a4 00 00 02 00 02
2 < 90 00
3 > 00 b0 00 00 08
4 < 09 6f 06 70 00 21 20 00 90 00
```

... And generates Python code from it:

```
1 card.transmit(SelectFile([0x00, 0x02]))
2 # OK
3 card.transmit(ReadBinary(length=8))
4 # 09 6f 06 70 00 21 20 00, OK
```

Tools provided by carddecoders

- ▶ logdecoder
- ▶ carddecoders.reverse_helpers

Find numbers in various encodings:

```
1 >>> contains_number(ByteString(  
2     "09 6f 06 70 00 21 20 00"), 12000)  
3 number found in BCD at offset 5.5 bytes  
4 >>> contains_number(ByteString(  
5     "09 6f 06 70 00 21 20 00"), 1648)  
6 number found in big endian encoding ending  
7 at 4.0 bytes
```

Find length indicators:

```
1 >>> backward_length(ByteString(  
2     "70 3c 5f [...] 5f 28 02 00 40"))  
3 index 1: 60 remaining  
4 index 59: 2 remaining
```

Further reading

- ▶ Introduction to Smart Cards

<http://www.smartcard.co.uk/tutorials/sct-itsc.pdf>

- ▶ Overview over ISO 7816

http://www.cardwerk.com/smartcards/smartcard_standard_ISO7816.aspx

- ▶ Smartcard protocol sniffing (hardware side)

<http://events.ccc.de/congress/2007/Fahrplan/events/2364.en.html>