Finding Vulnerabilities in Low-Level Protocols

Nordine Saadouni

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2017

Abstract

Basic example of an abstract (this will be changed)

Smart cards are used commercially and within industry for authentication, encryption, decryption, signing and verifying data. This paper aims to look into how the smart card interacts with an application at the lower level. PKCS#11 (public key cryptography system?) is the standard that is implemented at the higher level and then broken down into command/response pairs sent as APDU traffic to and from the smart card. It is the APDU low-level protocol that will be analysed to see if any vulnerabilities are present with regard to the smart cards tested.

Acknowledgements

Acknowledgements go here.

Table of Contents

Chapter 1

Introduction

Smart-cards are formally known as integrated circuit cards (ICC), and are universally thought to be secure, tamper-resistant devices. They store and process, cryptographic keys, authentication and user sensitive data. They are utilised to preform operations where confidentiality, data integrity and authentication are key to the security of a system.

Smart-cards offer what seems to be more secure methods for using cryptographic operations. (And should still provide the same level of security that would be offered to un-compromised systems, compared to those that are compromised by an attacker). This is partly due to the fact that the majority of modern smart-cards have their own on-board micro-controller, to allow all of these operations to take place on the card itself, with keys that are unknown to the outside world and stored securely on the device. Meaning the only actor that should be able to preform such operations would need to be in possession of the smart-card and the PIN/password. In many industries, for applications such as, banking/ payment systems, telecommunications, healthcare and public sector transport, smart-cards are used due to the security they are believed to provide.

The most common API (application programming interface) that is used to communicate with smart-cards is the RSA defined PKCS#11 (Public Key Cryptography Standard). Also known as 'Cryptoki' (cryptographic token interface, pronounced as 'crypto-key'). The standard defines a platform-independent API to smart-cards and hardware security modules (HSM). PKCS#11 originated from RSA security, but has since been placed into the hands of OASIS PKCS#11 Technical Committee to continue its work (since 2013). [reference wikipedia PKCS#11].

In the previous 10-15 years, literature has shown a great deal of research has examined the PKCS#11 API. But not much attention has been paid to that of the lower-level communication, in which the higher level API is broken down into. This is analogous to C code being complied down to binary data to be operated on by the CPU. In the same context, a PKCS#11 function cannot be considered ’secure’ without its corresponding APDU command/response pairs also being considered ’secure’. Much like the addition of two integers cannot be considered to be correct in a higher level lanugage such as C, unless the corresponding binary instructions sent to the CPU are correct as well.

Chapter 2

Background

2.1 PKCS#11

2.1.1 Key Object

2.1.2 Attributes

2.1.3 Most Common Functions

2.2 ISO 7816

2.2.1 Command Structure

2.2.2 Response Structure

2.2.3 Inter-Industry/ Proprietary

2.2.4 Most Common Commands

2.2.5 File Structure

Chapter 3

Cryptographic Operations

3.1 Hash Function

explain this in an overview term

3.2 Asymmetric Encryption

explain this in an overview term

3.2.1 RSA

3.3 Symmetric Encryption

explain this in an overview term

3.3.1 DES

3.3.2 Triple DES

3.3.3 AES

3.3.4 ECB Mode

3.3.5 CBC Mode

3.4 Message Authentication Codes

explain what a message authentication code is, what its used for
(wikipedia have good diagrams and explanations of this!!)

3.4.1 Hash Based - Message Authentication Codes (HMAC)

3.4.2 Cryptographic Based - Message Authentication Codes (CMAC)

3.5 One Time Passwords

What is a one time password → what forms are there?

3.5.1 Hash Based - One Time Passwords (HOTP)

3.5.2 Time Based - One Time Passwords (TOTP)

Chapter 4

Tools

4.1 PCSC-lite

4.2 Virtual Smart-Card

This is the man in the middle attack
Need to explain how this works thoroughly

4.3 Parsing → Scripts created for ease of reading

Chapter 5

Related Work / Literature Review

This will be a brief chapter and will discuss all of the research I have conducted. Mainly regarding PKCS#11 API attacks due to the small amount of literature that is available for APDU level attacks I shall also explain why some of the attacks are not able to be conducted on the particular card I am reviewing

Chapter 6

PKCS#11 Functions - APDU analysis

Here I shall simply give a trace of each PKCS#11, and give an analysis of each trace

6.1 Initialization?

Might be worth separating these!

6.2 C_login

Place an image here of the trace

6.3 C_findObjectInit

Place an image here of the trace

6.4 C_generateKey

Place an image here of the trace

6.5 C_generateKeyPair

Place an image here of the trace

6.6 C_createObject

Place an image here of the trace

6.7 C_destroyObject

Place an image here of the trace

6.8 C_encrypt

Place an image here of the trace

6.9 C_decrypt

Place an image here of the trace

6.10 C_sign

Place an image here of the trace

6.11 C_verify

Place an image here of the trace

6.12 C_setAttribute

Place an image here of the trace

6.13 C_wrap / C_unwrap

Place an image here of the trace

Chapter 7

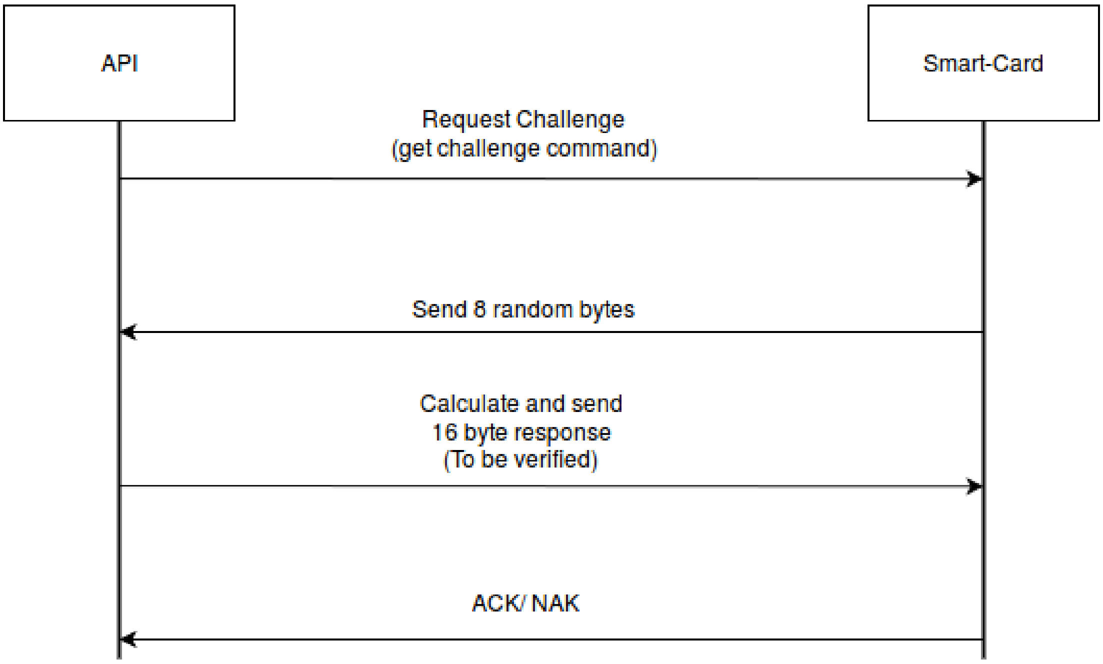
Attempts To Attack At the APDU Level

7.1 Reverse Engineering PIN/password Authentication

The first attack that I decided to attempt is to reverse-engineer the PIN authentication method. The reasoning behind this is because if this can be successfully done, the PIN can then be inferred from one communication trace sniffed between smart-card and the API (on the computer). The inference comes from the fact that once the method is deduced, an attacker can simply brute force the possible combinations of a PIN, to test if a match is found. (This becomes clearer in the latter sections)

From previous work completed on this card by an MSC student last year [?], and from the analysis conducted in section 6.2, it was quite clear that the card has the following characteristics in terms of PIN authentication. The PKCS#11 API requests a challenge, the smart-card responds with an 8 byte challenge, the API then calculates a 16 byte response (using the 8 byte challenge, and the PIN), the smart-card verifies whether or not the response is correct. There are two response formats to that APDU verification command:

- ’90 00’ → verification succeeded, correct PIN was entered
- ’63 CX’ → verification failed (where X is the number of attempts left before the card is blocked)



The following sections are explanations of the searches that we conducted to try and reverse-engineer the protocol showed above. To give a full understanding of how challenging this part of the project was, we will explain the combinations of possibilities we checked, and the reasoning behind each of them. These will be split up into different 'searches', and increment in terms of new findings and understanding of how the protocol may be implemented.

To aid these explanations, we introduce here 3 key sub-functionalities that most of the conducted searches use. Table 7.1 lists all the hash functions (see section 3.1) that were used, and provides the output length in bits & bytes. Those hash functions were all supported by openSSL and the python package 'hashlib'. Table 7.2 provides the names of the bitwise logical operations that were used to 'join' two bytes together. And table 7.3 provides the description of truncation methods used to reduce the output size of a search down to 16 bytes to match the response provided by the API.

From here on, in the explanation of the searches I will just refer to HASH, JOIN & TRUNCATE which will suggest that all of the elements in the tables have been iterated over and preformed on. For example TUNRCATE(HASH('this is a string')), means the string, 'this is a string', is to be hashed with all the functions in table 7.1, and then truncated to 16 bytes using all the methods listed in table 7.3.

Hash Name	Output Length (bits)	Output Length (Bytes)
MD4	128	16
MD5	128	16
MDC2	128	16
RIPEMD160	160	20
SHA	160	20
SHA1	160	20
SHA224	224	28
SHA256	256	32
SHA384	384	48
SHA512	512	64
WHIRLPOOL	512	64

Table 7.1: Hash Functions

Logical Operations
AND
OR
XOR
NOT AND
NOT OR
NOT XOR

Table 7.2: Bitwise Logical Operations (Joins)

Truncation method	Description?
first_16	Truncates the output by taking the first 16 bytes
last_16	Truncates the output by taking the last 16 bytes
mod_16	Truncates the output by taking modulus 2^{128} [We use 128 because that's the number of bits in 16 bytes]

Table 7.3: Truncation Methods

Before any searches could be conducted, the first task was to extract the values of the 8 byte challenge (denoted X), and the 16 byte response (denoted Y), from a communication trace of C_login. Table 7.4 provides the values for the PIN, X and Y in hexadecimal format.

Data	ASCII	HEX
PIN	'0000000000000000'	30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
Y	N/A	53 17 55 20 F4 30 18 56 80 E6 75 55 E1 91 A7 EC
X	N/A	68 F1 E4 92 85 36 39 A3

Table 7.4

In the very first original experiments, we made assumptions as did previous work on this card [ref Msc] that the PIN number was only numerical characters, only had 4-8 digits and if the PIN was less than 8 characters, it was padded using a special character to form 8 bytes. These assumptions turned out to be false, and thus the elementary experiments were all flawed from the beginning. I have not included a description of those experiments in the following sections, as many of them were in fact very similar to those explained below, just with different assumptions of the PIN. They also used a PIN for the card that was '12345', and hence there was an exponential explosion in the number of experiments for a particular search, due to needing to test for different padding schemes and characters. Hence why in these following experiments the PIN had been set to '0000000000000000' (16 zeros, no need for padding).

7.1.1 Authentication Protocol Search 1.0 (Password Storage)

Assumptions:

- PIN consists of alpha - numeric characters
- PIN is a maximum of 16 bytes
- PIN is encoded in ASCII characters
- For any PIN that is less than 16 bytes long, there is padding character used to pad the PIN to 16 bytes

Search 1 - Hash functions

In this initial stages we thought that there is a large possibility that the 16 byte response was generated by hashing a combination of the PIN and the 8 byte challenge. This was partly due to common practices used in industry whereby users passwords are often hashed, and in most cases salted (see section 3.1), before storing them in databases. This practice is more secure than storing plain text passwords, as if an attacker were to gain access to the back end databases storing said passwords, the password itself would not be available to see. For authentication the password is just hashed (and salted, if a salt is used), and then compared against to the stored value. The fact that from multiple traces the 16 byte response seemed to be uniformly random, it supported this hypothesis.

Thus the first search that we completed focused on the fact that a hash function was used to produce the 16 byte response. Below we have listed the methods tested in expirments to generate a 16 bytes, given X and the PIN.

We donote || as the concatination function. Thus for two strings 'string1 ' || 'string2' = 'string1 string2'.

Methods tested that produced a 16 byte output using X and the PIN

- join(truncate(hash(X)), pin))
- truncate(hash(join(pin, X || X)))
- truncate(hash(pin||X))
- truncate(hash(X||pin))
- truncate(hash(pin+X))
- truncate(hash(join(pin, square(X))))

[The methods should be read from the most inner brackets, outward. Therefore this means that the first method dictates that X is first hashed using one of the hash functions listed in the table 7.1. The output of that is then truncated to 16 bytes using one of the functions from table 7.3. All iterations of the functions in the tables were tested.]

The following experiment resulted in 2592 individual tests, but did not find a match to the response generated by the API [Y in table 7.4]. Thus we moved onto search 2.

Search 2 - PBKDF2

Following the failure of search 1, but still assuming there is a large possibility of a hash function being used due to the common practices mentioned in search 1, and the characteristics known so far of the 16 byte response Y. Then we decided to look at the password-based key derivation function (PBKDF2), which was created as part of PKCS #5 by RSA laboratories [?]. It has been mentioned in literature [?], and started to be used for more secure password storage as well as for key derivation. Essentially PBKDF2 takes as input, a password (the PIN), a salt (the 8 byte challenge X), a hash function, and the number of iterative rounds. If the number of iterative round is set to 10, then the salted password would be hashed once, and the output of that would be the input for the next round of hashing. This would be completed 10 times. Literature [?] has shown that the standard for the number of iterative rounds used to be 10,000, back in 2008 (check this date). Now it is suggested to use as many rounds as is computationally feasible by the device. Due to the processing power of a smart-card I assumed that this would not exceed 100,000 rounds, in the case that PBKDF2 was used.

Hence this search generated experiments that ran through 1 → 100,000 rounds of PBKDF2. As the default of PBKDF2 is to truncate the output by taking the first X (X here is a variable) bytes only 'first_16' truncation method was used.

Methods tested that produced a 16 byte output using X and the PIN

- PBKDF2(hash_function, PIN, X, number_of_rounds)

This generated 100,000 experiments per hash function. With 12 hash functions, this resulted in 1.2 million tests being run. Due to the sheer computational power required for this search I decided to parallelize the search based on the hash function, and run them on separate cores of a server. Even by improving the efficiency of this search, it still took 2 weeks to conduct.

Again this unfortunately did not result in a match between the 16 byte responses calculated and Y (the API's response). Hence we move onto search 3.

7.1.2 Authentication Protocol Search 2.0 (One Time Passwords)

Search 3 - OCRA: OATH Challenge-Response Algorithm

With no luck of deducing the method the authentication protocol uses, we decided to look into more complex standards that exist and used in different parts of the computing industry for challenge response protocols, rather than password storage techniques. The international engineering task force (IETF) released a paper in 2011 [?]. Section 7.1 of the paper gives a one-way challenge response algorithm which fitted the characteristics of the authentication that takes between the API and the smart-card.

Section 3.5.1 explains hash based one time passwords. But in essence HTOP is:
$HOTP(K,C) = Truncate(HMAC(K,C))\&0x7FFFFFFF$ [?-wiki]

Still to complete.

Search 4 - TOTP

With the above in mind, I also wanted to rule out TOTP. This was completed by halting communication between the smart-card and API using the man-in-the-middle tool (see section 4.2). I did this for upto 2 hours. We were looking for a failed verifaction, despite having the correct PIN. The failure should have been caused by the delay in the response if TOTP was used. This was not found and therefore ruled out the possibility of TOTP.

This section will be explained better, and will also include reasoning behind why TOTP is not often used (time sync problems).

Note there are 4 more searches conducted that included the use of block ciphers.
After that I will move onto explain how I thought there was a possibility of the use of a derived key from a globally known master key (this is due to a paper I read, on

zero knowledge protocols). And hence used MiTM tool to change the serial number of the smart-card to see if an incorrect response would be produced by the API, despite having the correct PIN.

Finally I will explain the 4 charactersistics I found by verifying with the card one after the other very quickly
let Y = Y1 || Y2
same PIN and X: same Y1 and Y2
same PIN, different X: same Y1 different Y2
different PIN, same X & different PIN and different X: different Y1 and different Y2

These characteristics are only present if the 'logins' happen within the same second. If the second changes then we have a different Y for the same PIN and same X. This suggests the use of a nonce (calculated by API on the computer), as well as X. My assumption is: encrpyt(nonce||X) [using a block cipher the PIN is the password]

Search 5
Search 6
Search 7
Search 8
Search 9

```
template = (\\
(CKA\_CLASS, LowLevel.CKO\_SECRET\_KEY), [private, public, data, cert (X.509)]
(CKA\_KEY_TYPE, LowLevel.CKK_DES), [AES,DES,RSA,ECC]
(CKA\_LABEL, label), [name]
(CKA\_ID, chr(id)), [id]
(CKA\_PRIVATE, True), [requires authentication]
(CKA\_SENSITIVE, True), [cannot be extracted unencrypted]
(CKA\_ENCRYPT, True), [can be used for encrypting data]
(CKA\_DECRYPT, True), [can be used for decrypting data]
(CKA\_SIGN, True), [[can be used for signing data]
(CKA\_VERIFY, True), [can be used for verifiying data]
(CKA\_TOKEN, True), [can be stored permanently on the device]
(CKA\_WRAP, True), [can encrypt a key to be extracted]
(CKA\_UNWRAP, True), [can decrypt an encrypted key]
(CKA\_EXTRACTABLE, False)) [is allowed to be extracted from the device]
```

Problem 1:
I inject my own RSA-2048 public key (only the public modulus part, the injected key has the same public exponent 9 bytes).
When secure messaging is starting to open straight after the communiacation of the injected key SEG_FAULT! Back to square 1, do not know what the key is.

7.2 Altering File Control Parameters for Login Retry Couner

7.3 Block Cipher Injection (MiTM)

7.4 Altering Key Object Attributes

- MSC's work

7.5 Using Key's without Authentication

- MSC's work - there is an attribute that should prevent this!

Chapter 8

Conclusion / Results

This shall summarise the whole report and my findings in regard to low-level vulnerabilities on the card.

Bibliography

[1] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1993.

[2] Albert Einstein. *Zur Elektrodynamik bewegter Körper*. (German) [*On the electrodynamics of moving bodies*]. Annalen der Physik, 322(10):891921, 1905.

[3] Knuth: Computers and Typesetting,
`http://www-cs-faculty.stanford.edu/~uno/abcde.html`

[4] Frank Morgner : Creating a Virtual Smart Card
`https://frankmorgner.github.io/vsmartcard/virtualsmartcard/README.html`

[5] Internet Engineering Task Force (IETF).
OCRA: OATH Challenge-Response Algorithm, June 2011
`https://tools.ietf.org/html/rfc6287`