# 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 'cryptokey'). 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

## 3.2 Asymmetric Encryption

## 3.3 Symmetric Encryption

## 3.4 MAC

## 3.5 OTP

# Chapter 4

# Tools

## 4.1 PCSC

## 4.2 Virtual Smart-Card

## 4.3 Parsing?

# 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

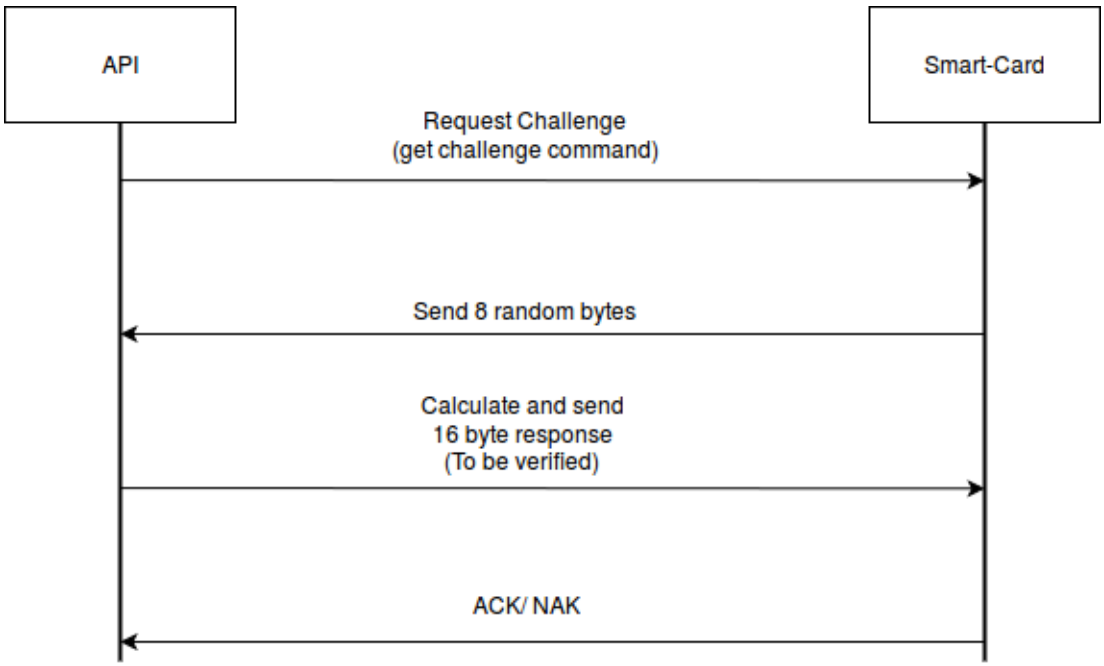
This will be the description of the attacks I have attempted, and most likely mentioned during the analysis from the previous chapter

## 7.1 Reverse Engineering PIN/password Authentication

The first attack that I decided to attempt is to reverse-engineer the PIN/password authentication method. The reasoning behind this is because if this can be successfully done, the PIN/password can then be inferred from one communication trace sniffed between the computer using the smart-card and the smart-card itself. The inference comes from the fact that once the method is deduced, an attacker can simply brute force the possible combinations of a PIN/password, and run them through the 'method' that has been reversed engineered to see if a matching response is found. If so, then the PIN is that of the matching response.

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/password authentication. The 'middleware'/API requests a challenge, the smart-card responds with an 8 byte challenge, the API then calculates a 16 byte response, the smart-card verifies whether or not the response is correct. There are two response formats to that verification:

- '90 00' $\rightarrow$ verification succeeded, correct PIN/password was given
- '63 CX' $\rightarrow$ verification failed (where X is the number of attempts left before the card is blocked)




Initially it was assumed that the PIN number consisted only of numbers, and was between 4-8 digits long. This assumption was found to be incorrect, however the assumption is key to the explanation of the following section (authentication protocol search 1.0), as it supports the reasoning behind the initial attempts made to reverse-engineer the authentication protocol.

DO NOT FORGOT!
Need to mention somewhere in this section that I have tried:
delaying the response to the smart-card to test for TOTP (upto 2 hours)
Modified initialization data to test if there is a master key, which could be used to create a derived key to encrypt the PIN/Password. (there are references to papers which suggest some smart-card manufactures do this!)

### 7.1.1 Authentication protocol search 1.0

Assumptions:

- PIN consists of only numerical digits
- PIN is a maximum of 8 bytes
- PIN is encoded in ASCII characters
- For any PIN that is less than 8 bytes long, there is padding character used to pad the PIN to 8 bytes

The following subsections are explanations of the searches conducted under the assumptions listed above, in order to try and find the method used in calculating the response to be verified using the password and the 8 byte challenge. To give a full understanding of how challenging this part of the project was I will give a brief explanation of the code that runs through different combinations of possibilities for the challenge-response protocol.

### 7.1.2 Search 1.1

Please ignore this for now, its just placed here while I write up this actual section
X is the random 8 byte challenge

method 1:
HASH(X) -¿ 16 bytes, join(HASH(X), pin)

method 2:
HASH(join(pin , X——X)), reduce to 16 bytes

method 3:
salt pin with X (pin comes first), hash salted pin

method 4:
salt pin with X (X comes first), hash salted pin

method 5:
format pin to integer, pin += random number (from get challenge), hash(all)

method 6:
square(X) (creates 16 bytes), join(pin, square(x)), hash(all)

iterate through:
methods $1 \rightarrow 6$
padding methods (pad at end, pad at start)
padding character used ($0 \rightarrow 255$)
hash used
join method (xor, nxor, and, nand, or, nor)
truncate output to 16 bytes using one of the following functions (start 16, last 16, mod

output)

### 7.1.3 Search 1.2

### 7.1.4 Search 1.3

### 7.1.5 Authentication protocol search 2.0

# Chapter 8

# Conclusion / Results

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