Finding Vulnerabilities in Low-Level Protocols

Nordine Saadouni

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

2017

Abstract

To a large extent smartcards are used in industry due to the security they are expected to provide. The most common API used to instruct smartcards is PKCS #11. Each PKCS #11 function is broken down into multiple ISO/IEC 7816 APDU command response pairs. In this project we undertake the analysis of the **Athena IDprotect smartcard** that uses encrypted APDU level communication to prevent leaking sensitive key values. We present two new attacks at the APDU level for our smartcard. The first attack concerns reverse engineering the proprietary PIN authentication protocol that is used to log a user into the smartcard. The second attack concerns reverse engineering the proprietary secure messaging protocol used at the APDU level. The research of reverse engineering proprietary implementations of APDU encrypted communication is novel in that it has not been carried out before.

Acknowledgements

Acknowledgements go here.

Table of Contents

Chapter 1

Introduction

Smartcards are 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 used to preform operations where confidentiality, data integrity and authentication are of critical importance to the security of a system.

Smartcards are expected to offer more secure methods for using cryptographic operations. They should 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 smartcards have their own on-board micro-controller. This allows all of the operations to take place on the smartcard itself. This means the only actor that should be able to preform such operations would need to be in possession of the smartcard and the PIN/password. Smartcards are widely used by many industries for different application purposes such as, banking/ payment systems, telecommunications, healthcare and public sector transport. The popularity of their use is to a large extent driven by the security they are expected to provide.

The most common API (application programming interface) that is used to communicate with smartcards is PKCS#11 (Public Key Cryptography Standard). This is also known as 'Cryptoki' (cryptographic token interface). PKCS #11 has originated from RSA Laboratories, but has since been placed into the hands of OASIS PKCS#11 Technical Committee to continue its work (since 2013). [6].

PKCS #11 defines a platform-independent API and conditions that must be strictly followed to communicate and instruct smartcards /hardware security modules (HSM). The API only provides C header files (with the object tpyes, attributes and functions to be supported), and leaves the actual implementation to card manufacturers. Each function is broken down into multiple Application Data Protocol Unit (APDU) command-response pairs. These command-response pairs are binary instructions for the smartcard to execute and respond with data if requested. The International Standard Organization (ISO) and International Electrotechnical Commission (IEC) jointly manage the ISO/IEC 7816 standard that defines electronic identification cards. This standard provides details upon how these binary instructions should be formulated. They provide standardized inter-industry commands and responses, but also allow for proprietary commands and responses to be used, with the definitions behind the command-response pairs to be kept as proprietary information.

In the last 15 years or so, the literature [3, 5] has mainly focused the examination of the PKCS#11 API and the security it provides. However, little attention has been paid to the lower-level communication (APDU command-response pairs), in which the higher level API is broken down into. To the best of our knowledge, there are only two papers that have been published [4, 8]. The two papers concern the reverse engineering of proprietary ISO 7816 commands without access to the cards proprietary PKCS #11 middlware, and APDU-level attacks on hardware security modules that do not use encrypted APDU command-response pairs.

It is clear from the available research that APDU level attacks require further investigation. The research demonstrates the ability to grab sensitive key value and user PIN information at the APDU-level of the smartcards they examine. If the security at the low-level (APDU) command-response pairs is flawed it compromises the security that is provided by the PKCS #11 standard.

This motivates the research of our project. We undertake the analysis of the **Athena IDprotect smartcard** that uses encrypted APDU level communication to prevent leaking sensitive key values. This has made the work of this project more challenging. We present two new attacks at the APDU level for our smartcard. The first attack concerns reverse engineering the proprietary PIN authentication protocol that is used to log a user into the smartcard. The second attack concerns reverse engineering the proprietary secure messaging protocol used at the APDU level to protect sensitive keys from being revealed in plain text. This attack is novel in that it has not been carried out before. Furthermore, we extend the analysis of Lan (2016) [11] in two areas. The first one concerns (specifically for our smartcard) how PKCS #11 functions are broken down into ISO 7816 (APDU) command response pairs. The second one is regarding the key and attribute file locations in our smartcards memory. Finally, we test an attack that has been reported in the literature [4] to find out if our smartcard suffers from the same vulnerability.

Our study makes five important contributions, these are:

- We successfully complete the first attack which concerns the reverse engineering of the proprietary (challenge response) PIN authentication protocol used in our smartcard. The PIN authentication protocols differ depending on manufacturers and also the versions of smartcards they produce. This is a new vulnerability that applies to our project's smartcard. Now an attacker can brute force/ use a dictionary attack to calculate a user's PIN given exactly one successful login communication trace. This directly violates the PKCS #11 standard.

- The second attack concerns reverse engineering of the proprietary secure messaging protocol to encrypt APDU communication (specifically for our smartcard). We provide evidence to prove the cryptographic protocol used in the initialization process for secure messaging is Diffie Hellman. We also implement a man in the middle attack to set the value of the shared secret to zero. This gives an attacker the same knowledge as the API and smartcard. The final step in reverse engineering the secure messaging protocol is to derive two session keys given the shared secret value, a 32 byte challenge, and a key derivation function. Unfortunately, due to time constraints we are unable to complete this final step. We however include the methodology of the experiment that should be used to complete this attack. To the best of our knowledge, we are the first to research the reverse engineering of encrypted APDU communication (secure messaging).
- In our third contribution we extend Lan (2016) [11] analysis of how (specifically for our smartcard) the PKCS #11 functions are broken down into multiple (APDU) command-response pairs. In the cases of C_generateKey, C_generateKeyPair, C_destroyObject, C_wrap and C_unwrap we provide new insights into the APDU communication.
- Lan (2016) [11] finds the location of key and attribute files for block cipher keys only. We extend his work to include, RSA public and private, key and attribute file locations in the smartcards memory.
- In our final contribution we test work conducted in [4], whereby the controls dictated by the attributes of a PKCS #11 object can be overridden at the APDU level. We do this in order to test if our smartcard has the same vulnerability.

The remainder of the project is organized as follows. Chapter 2 provides a background of cryptographic standards. Chapter 3 explains the PKCS #11 and ISO/IEC 7816 standards. Chapter 4 provides a review of the relevant literature. Chapter 5 discusses details of the existing tools that are used for our project. The chapter also explain how we have developed further one of the tools. Chapter 6 provides an extended analysis of how the PKCS #11 functions are broken down into command response pairs (for our smartcard) and suggest possible vulnerabilities to investigate. Chapter 7 provides details regarding the motivations, implementation and results of the new attacks we investigate for our project smartcard. Chapter 8 concludes, suggests mitigations and finally provides details and reasoning for further research.

Chapter 2

Background - Cryptography

This chapter aims to provide an understanding of different cryptographic standards and how to use them (the chapter does not go into the details of the correctness of the standards). This will help our explanations regarding our smartcard's functionality and cryptographic operations it undertakes. In addition, this chapter also provides an understanding of terminology that is used throughout this project.

2.1 Cryptograhic - Hash Functions

FIPS PUB 180-4 define [14] hash functions are a one way functions. They take in as input a **variable sized** message (a string) and output a **fixed size** message digest (another string). Hash functions are expected to hold three properties:

1. **Simple Computation** - It should be easy to calculate the message digest of a given message.
2. **Difficult Reversing the Computation** - It should be difficult to find the message given its message digest.
3. **Collision Resistance** - It should be difficult to find two messages m_1 and m_2 that have identical message digests.

$$Message\_digest = Hash\_function(message)$$

2.2 Asymmetric Encryption

Asymmetric encryption relies on public and private key pairs. Public keys can be sent over insecure channels for an entity to use to encrypt a message. Private keys should always kept a secret. Only the private key can be used to decrypt messages encrypted via the public key. This allows communication to remain confidential despite the use insecure mediums of transport.

2.2.1 RSA

Rivest, Shamir, and Adleman in their 1978 paper [12] on RSA asymmetric encryption standard define a public key (N,e) and a private key (N,d), where N is the multiplication of two large primes p and q.

	Public Key	Private Key
Modulus	N = p x q	N = p x q
Exponent	e	d

Messages to be encrypted are converted into integers, and follow the formula's below for encryption and decryption.

RSA public key encryption

$$encrpyted\_message = message^{e} \quad mod \quad N$$

RSA private key decryption

$$message = encrypted\_message^{d} \quad mod \quad N$$

There are different RSA private and public key pairs. The size of the moudlus (N) in bits gives the names of the different forms of RSA. The different forms of RSA provide different levels of security, these are reported below.

RSA-type	Exponent size (bytes)
RSA-512	64
RSA-1024	128
RSA-2048	256
RSA-4096	512

Chinese Remainder Theorem (CRT) RSA cryptography

Shinde and Fadewar in their 2008 paper [13] explain how the use of the Chinese remainder theorem can improve the efficiency of RSA decryption by upto a factor of four. It requires three additional parameters to be calcualted upon key pair generation, these are given below.

1. $dP = (1/e) \quad mod(p-1)$
2. $dQ = (1/e) \quad mod(q-1)$

3. $qInv = (1/q) \quad mod(p)$

To compute the decryption ($m = c^d mod(N)$) using CRT apply the following:

$$m1 = c^{dP} mod p$$

$$m2 = c^{dQ} mod q$$

$$h = qInv.(m1 - m2) mod(p)$$

$$m = m2 + h.q == c^d mod(N)$$

2.2.2 Diffie Hellman

Diffie and Hellman in their 1976 paper [7] presented the Diffie Hellman protocol. The protocol has a public and private key pair that is used to derive a shared secret over an insecure channel. Two entities in communication with each other must agree on the global parameters.

	Global Parameter
Generator	G
Public Modulus	p

Each entity selects its own private key to be a random number between 1 and *p-1*. Using the global parameters each entity calculates its own public key as follows:

$$public\_key = G^{private\_key} \quad mod(p)$$

The two entities send each other their public keys and then calculate a shared secret. This cannot be replicated by other entities that may intercept the communication. The shared secret is computed as follows:

$$shared\_secret = public\_key_2^{private\_key_1} \quad mod(p)$$

2.2.3 Elliptic Curve Cryptography

[It is worth noting that in this section, we only report how Elliptic Curve Cofactor Diffie Hellman (ECCDH) operates as this is an area of research that is part of our second attack discussed in chapter 7]

Elliptic curve cryptography is based on the algebraic structure of elliptic curve over finite fields. The standard only started to attract wide spread since 2004. It is similar to RSA in that it provides a method for sharing an entity's public key over an insecure channel. The public key can be used to encrypt data, sign data or even use a version of the Diffie Hellman protocol to derive a shared secret (ECCDH). Hankerson et al. published a paper in 2006 [9] explaining how to use the ECC standard.

One major difference between ECC and standard cryptographic methods, is the addition and multiplication operations. In elliptic curve cryptograhpy addition and multiplication are redefined over the curve. The maths behind ECC means that addition and multiplication are no longer reversible operations, and are now assumed to be one way functions. This means to compute the reverse of one of the operations, one must solve the 'elliptic curve discrete logarithm problem'. In figure 2.1, we report examples of how addition and multiplication operation work in ECC.

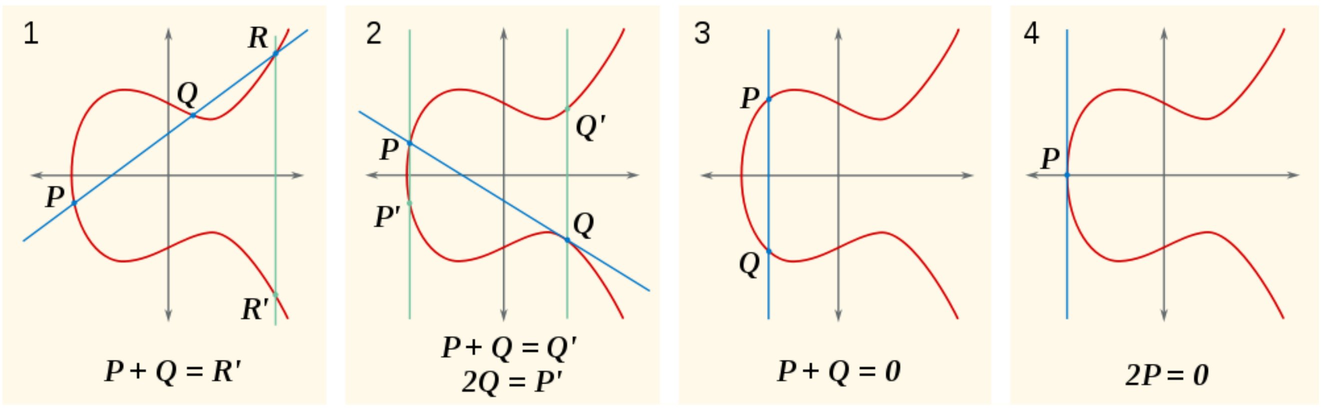


Figure 2.1: ECC Addition and Multiplication [2]

Global Paramters are shared (similar to Diffie Hellman) between two entities. These are given below.

Global Parameter	Description
p	The field the curve is defined over
(a,b)	Values defining the curve
G	Generator point (fixed point on the curve)
n	Prime order of G (n.G = elliptic identity)
h	Cofactor
Seed	Random number

Elliptic Curve Cofactor Diffie Hellman

Using the global parameters, two entities can now generate their own private keys (defined on the curve). The value of their respective private keys (d), are randomly selected numbers (seeded with *Seed*), in the range $1 <= d <= (n-1)$.

Using their own private key, each entity can now calculate its own public key. This is similar to Diffie Hellman, however the ECC multiplication operation is used instead.

public_key$_1$ = *(*X_1*,* Y_1*) - A point on the curve*

$public\_key_1 = (private\_key_1).G$

The two entities send each other their public keys and then calculate a shared secret. This cannot be replicated by other entities that may intercept the communication. The shared secret is computed as follows:

$$shared\_secret = public\_key_2.(private\_key_1.h)$$

2.3 Symmetric Encryption

Symmetric key algorithms rely on the fact that both entities in communication are both in the knowledge of a shared key. The shared key is to be held a secret by both entities and is used for both encryption and decryption. The symmetric key algorithms that are implemented on the our smartcard are block ciphers. They are named block ciphers because they encrypt and decrypt data in blocks (predefined number of bytes). Thus, if a message's length is not of a multiple of the block size it must be padded. The three block ciphers that the smartcard has implemented are:

1. Advanced Encryption Standard (AES)
2. Data Encryption Standard (DES)
3. Triple Data encryption Standard (3DES)

The characteristics of the three block ciphers in table 2.1. Block ciphers also have several different modes of operation. These modes dictate different methods of encrypting and decrypting data. We only explain the two that are present on our smartcard, these are ECB and CBC.

Block Cipher	Block Size (bytes)	Key Size (bytes)
AES-128	16	16
AES-192	16	24
AES-256	16	32
DES	8	8
3DES-64	8	8
3DES-128	8	16
3DES-192	8	24

Table 2.1: Block Cipher Characteristic's

2.3.1 Electronic CodeBook (ECB) Mode

Electronic CodeBook mode is the simplest form of encryption. Every block is treated separately, encrypted and concatenated together to form the encrypted message. This is shown in figure 2.2.

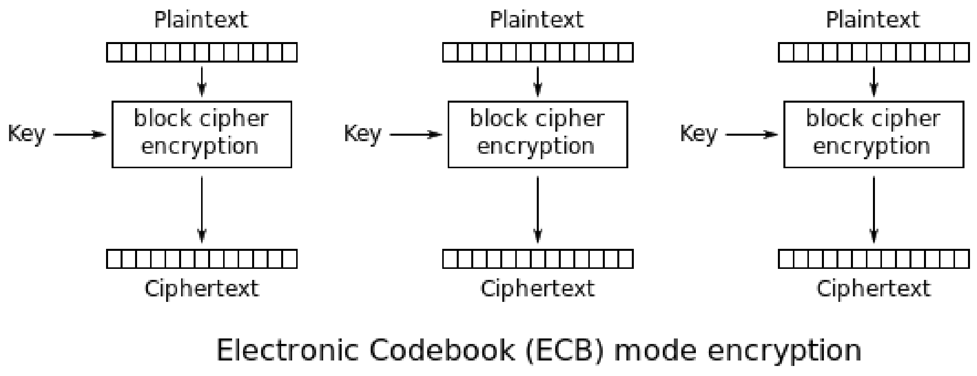


Figure 2.2: Block Cipher ECB Encryption [1]

2.3.2 Cipher Block Chaining (CBC) Mode

Cipher Block Chaining mode is more complex. It requires an *initialization vector* (IV), that is used to XOR with the first block of plaintext for encryption. The output of every encrypted block is then XOR'd with the next block of plaintext before it is encrypted. This can be seen in figure 2.3.

[The IV is always of the same length as the block size for a given block cipher]

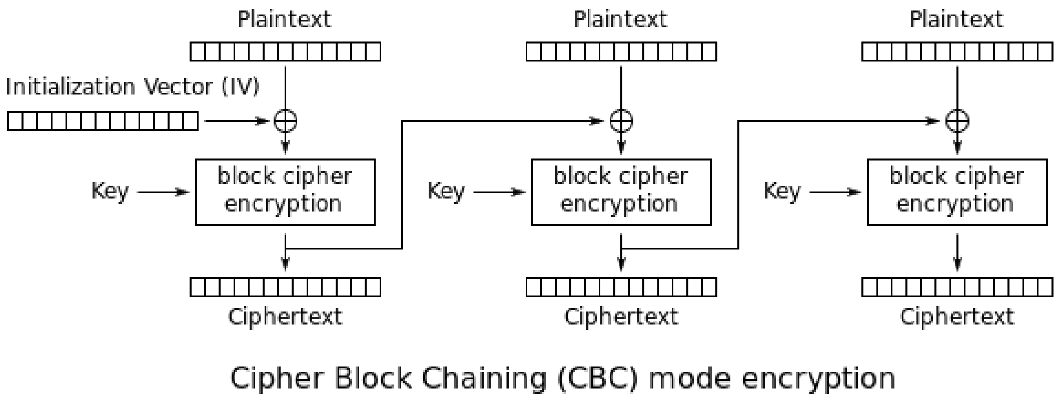

Plaintext
Initialization Vector (IV)
Key
block cipher encryption
Ciphertext
Cipher Block Chaining (CBC) mode encryption

Figure 2.3: Block Cipher CBC encryption [1]

2.4 Message Authentication Codes

A Message Authentication Code (MAC) is a checksum/ tag that is appended to a message. Its use is to provide data integrity to a message sent over an insecure channel. It does this by forming a tag that should be difficult to generate without the knowledge of the shared key and the message. Thus, the receiver of the message will use their shared key, the MAC algorithm and the message to compute (what should be the same tag) a tag. This tag is then cross-referenced with the tag appended to the message. If they match the entity is becomes aware that the message has not been tampered with.

2.4.1 Hash Based - Message Authentication Codes (HMAC)

RFC 2104 [10] defines the HMAC: keyed-hashing for message authentication. HMAC's use cryptographic hash functions to combine the message and the shared key to generate a tag that can only be computer with the knowledge the key and the message. The formula is:

Variable	Description
Hash	A cryptographic hash function (e.g. SHA-256)
Key	The shared secret key between the two entities
Msg	The message to compute the tag for
opad	The byte 0x36 repeated B times
ipad	The byte 0x5C repeated B times
B	Block size of the block cipher used in the cryptographic hash function

HMAC(Key, Msg) = Hash(Key XOR opad, Hash(Key XOR ipad, Msg))

2.4.2 Cryptographic Based - Message Authentication Codes (CMAC)

RFC 4493 [?] defines the AES Cryptograhic Message Authentication Code (the use of other block ciphers is also allowed for CMAC's). The standard defines that first, two additional keys K_1 and K_2 are generated using a derivation of the original shared key known by the two entities. The message is encrypted using the block cipher in CBC mode with an IV = 0, denoted Enc_1.

The two shared keys are used to alter last block of the message as defined in the standard, and then encrypted using the block cipher and original shared key. The output is XOR'd with Enc_1 to form the CMAC tag. This is reported in figure 2.4.

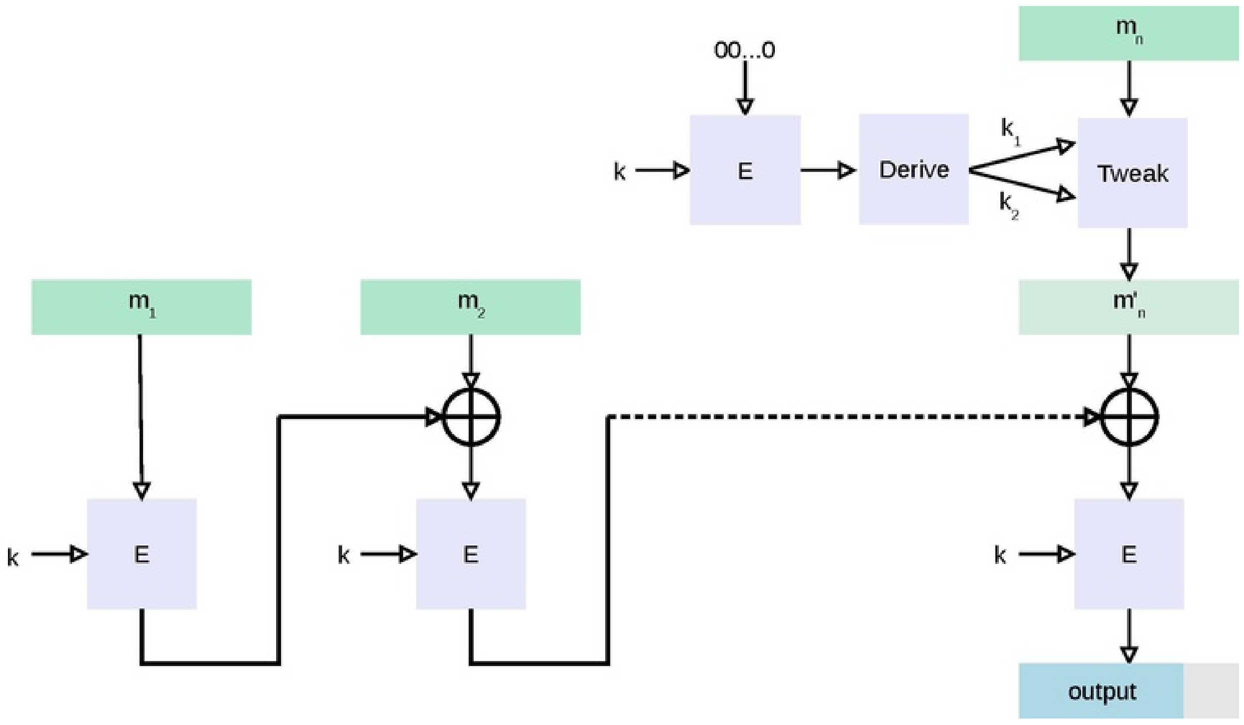


Figure 2.4: CMAC [?]

2.5 One Time Passwords

One Time passwords (OTP) is a password that is valid for one login. They overcome the short coming of original *static* based passwords that are vulnerable to replay attacks. Normally they are incorporated using two factor authentication methods (this doesn't have to be the case). This is done by proving knowledge of any two out of the following three pieces of information.

1. 'Something you know' - e.g. A PIN number or password
2. 'Something you have' - e.g. An RSA key fob (OTP calculator)
3. 'Something you are' - e.g. Your fingerprints

The entity requesting to login and the entity verifying the request need to be aware of the following:

- The shared secret key (password)
- The algorithm for generating OTP using the password
- Additional variables the algorithms require.

2.5.1 Hash Based - One Time Passwords (HOTP)

RFC 4226 [?] defines the hash based one time password (HOTP). The standard uses a cryptographic hash function (for HMAC), a counter, and a shared secret key (the password) to generate different passwords for authentication.

Variable	Description
Password (K)	The shared secret used to login
Counter (C)	The value that is used to generate different outputs given the same password
Hash function	The cryptographic hash function used in HOTP

HOTP uses the HMAC that we discussed in section 2.4.1 in order to calculate its output. The first byte of the output of logically AND'd with the value '7F' (hexidecimal) and truncated down to the number of bytes the user requests. The formula is reported below (4 byte output):

HOTP(K,C) = Truncate(HMAC(K,C)) & 0x7F FF FF FF

2.5.2 Time Based - One Time Passwords (TOTP)

RFC 6238 [?] defines the time based one time password (TOTP). The standard uses the hash based one time password (HOTP) as discussed above, but changes the counter value C to be a value determined by a time period (requiring synchronous clock between two devices). The variables required to use the TOPT are reported in the table below.

Variable	Description
Password (K)	The shared secret between the two entities
T0	Unix time to start counting steps (default 0)
X	Time steps/ Time period (in seconds)
Counter value (T)	(Current Unix time - T0) / X
Hash function	Used as the parameter to HOTP

The TOPT formula for calculating a one time password is derived by using a function of time that is synchronised between two devices is reported below:

TOTP(K, T) = HOTP(K, C)

Chapter 3

Background - Standards For Smartcards

This chapter aims to explain two important standards, PKCS #11 and ISO/IEC 7816. PKCS #11 is a standard that defines an application programming interface (API) for users to communicate with smartcards. The ISO/IEC 7816 defines electronic identification cards including the command response pairs that the PKCS #11 functions are broken down into. These two standards are required in order to operate our project's smartcard.

3.1 PKCS#11

Public Key Cryptography Standard #11 has been created by RSA Laboratories. It is a platform independent API, that defines rules and functions that should be implemented by the card manufacturer in the form of a 'middleware'. The API is written in the C programming language, and only header files are included. The implementation of the API's functions are left the discretion of each card's vendor. The card vendor is required to break down each function into multiple ISO 7816 command-response pairs to carry out the indented function.

Smartcards store cryptographic keys, certificates and data. The standard treats each of these data items as objects. With each object having attributes that define the controls that PKCS #11 API should implement. In sections 3.1.1 and 3.1.2 we report the definitions of the PKCS #11 objects, attributes and functions.

3.1.1 PKCS # 11 Object's and Attributes

Five different types of objects are supported by the PKCS #11 standard. These are presented as the CKA_class attribute and reported in table 3.1. The sensitive values stored in objects are reported in table 3.2. Finally, the attributes that each object has (depending on its type) are reported in table 3.3. The attributes in table 3.3, set controls that the API should implement.

CKA_class	Definition
Secret Key	A secret key object is an object that stores a block cipher key value and its attributes.
Public Key	A public key object is an object that stores the public key value and attributes of an asymmetric key pair
Private Key	A private key object is an object that stores the private key value and attributes of an asymmetric key pair
Data	A data object is a generic object that allows any value to be stored within its object's value.
Certificate	A certificate object stores certificates usually wrapped in an ASN.1 encoding (e.g X.509) for their use on-line.

Table 3.1: PKCS #11 Object Types

Object	Value	Definition
Secret Key	CKA_Value	Stores the value of the block cipher key
Public Key	CKA_public_exponent & CKA_public_modulus	Stores the public exponent and modulus of RSA asymmetric keys. (These are different for elliptic curve asymmetric public keys)
Private Key	CKA_private_exponent	Stores the private exponent for RSA asymmetric keys. (These are different for elliptic curve asymmetric keys)
Data & Certificate	CKA_Value	Stores the value of the data or certificate.

Table 3.2: PKCS #11 Senseitive Attribute Values

CKA_attribute	Value Type	Definition
Key Type	String	The key type attribute defines what type of key the object represents ***[This is not present for data and certificate objects]***
Token	Boolean	The token attribute defines whether or not the object is to be stored permanently on the card, or securely destroyed after a session. ***[If token = True, the object must have an ID]***
Modulus Bits	Integer	The modulus bits attribute sets the number of bits to use for an RSA asymmetric key pair (E.g. 1024)
Public Exponent	Integer	The public exponent attribute sets the public exponent for an RSA asymmetric key public key.
Value Length	Integer	The value length attribute sets the key size for a block cipher. (E.g. AES 16 byte or 32 byte key size)
Private	Boolean	The private attribute defines if a user has to be authenticated with the smartcard before the object can be utilised
Label	String	The label attribute gives a name to the object for reference by the user
ID	String	The ID attribute gives an ID string value for lookup searches within the API by the user, when searching for objects stored on the card.
Sensitive	Boolean	The sensitive object defines if an object can be extracted out of the smartcard in an unencrypted format
Extractable	Boolean	The extractable attribute defines if an object can be extracted out of the smartcard
Encrypt	Boolean	The encrypt attribute defines if the object can be used for encrypting data
Decrypt	Boolean	The decrypt attribute defines if an object can be used for decrypting data
Sign	Boolean	The sign attribute defines if an object can be used for signing data
Verify	Boolean	The verify attribute defines if an object can be used for verifying the signature of data
Wrap	Boolean	The wrap attribute defines if an object can be used for wrapping (encrypting) another key object for extraction out of the smartcard in an encrypted format
Unwrap	Boolean	The unwrap attribute defines if an object can be used for unwrapping (decrypting) another (encrypted) key object for storing within the smartcard

Table 3.3: PKCS #11 Attribute Definitions

[In table 3.3, we only report those attributes we have used in our study. They're additional attributes for different types of cryptographic keys]

3.1.2 PKCS #11 Functions

There are 13 functions (that are commonly used) that use the objects described in section 3.1.1. These functions preform different operations. We report in table 3.4 the function names and the description of the functionality each function provides. The implementation of these functions are all completed by the cards vendors middleware and therefore differ from one manufacturer to another. They also differ between different smartcards from the same manufacture. We report an analysis of how 'Athena smartcard solutions' implement their middleware for the Athena IDprotect smartcard we study. This is provided in chapter 6.

[Note the PKCS #11 standard states that the attributes CKA_sensitive and CKA_extractable cannot be changed using setAttribute function from false to true, and true to false respectively.]

Function	Parameters	Description
C_login	User PIN	The login function should send the user's PIN over to the smartcard for the smartcard to verify if the PIN is correct. In the case it is, the user should now be in an authenticated state with the smartcard.
C_findObject	-	The findObject function should search the smartcards file system for currently stored objects. This should return different object handles, dependant on the authenticated state of the user, and the *private* attribute of objects stored on the card.
C_generateKey	template of key	The generateKey function *(used for generating block cipher keys)* should use the template provided to store the attributes in the smartcards memory, and then set CKA_Value of the block cipher key to a computer generated value. (can be generated by either API on computer or the smartcard)
C_generateKeyPair	template of both keys	The generateKeyPair function *(used for generating asymmetric key pairs)* should use the template provided for both keys to store the attributes of each key individually to the smartcard's memory. The values of the sensitive attributes should be computer generated as well, just like generateKey function.

Table 3.4: PKCS #11 Functions

C_createObject	template of object	The createObject function should use the template provided to create the object in its entirity. Thus, the sensitive values should be included within the template *(unlike the generate functions)* as well. Which should then be saved to the smartcard's memory.
C_destroyObject	object handle	The destroyObject function uses the object handle to locate the object stored on the smartcard and delete it from its memory.
C_setAttribute	object handle & template for attribute to be changed	The setAttribute function uses the object handle provided to locate the object (in the smartcards memory) and change the attribute values to those provided in the template.
C_Encrypt	object handle & mechanism & data	The encrypt function uses the object handle to locate the cryptographic key. It then uses that key to encrypt the data provided using the cryptograhic mechanism (e.g. AES-ECB) and return the encrypted data.
C_Decrypt	object handle & mechanism & data	The decrypt function uses the object handle to locate the cryptographic key. It then uses that key to decrypt the data provided using the cryptographic mechanism (e.g. AES-ECB) and return the decrypted data.
C_Sign	object handle & mechanism & data	The sign function uses the object handle to locate the cryptographic key. It then uses that key to sign the data provided using the cryptographic mechanism (e.g. AES-CMAC) and return the signature of the data.
C_Verify	object handle & mechanism & data & signature	The verify function uses the object handle to locate the cryptographic key. It then uses that key to sign the data provided using the cryptographic mechanism (e.g. AES-CMAC) and return the comparison between the computed signature and the signature provided.
C_Wrap	2 x object handle's	The wrap function takes both object handles. The first one is to locate the key used for wrapping (encrypting) and the second one is used to locate the key to be wrapped. The function then encrpyts the second key using the first key, and returns the encryption.

Table 3.4: continued

C_Unwrap	object handle & template	The unwrap function takes the object handle to locate the key to be used to unwrap (decrypt) the sensitive value within the template. The template consists of the attributes for the wrapped key, and the encrypted key value. The key is then unwrapped and stored in the smartcards memory.

Table 3.4: continued

3.2 ISO/IEC 7816

The International Standards Organization (ISO) and the International Electrotechnical Commission (IEC) jointly manage the standard that defines electronic identification cards (mainly contact smartcards) [?]. The content table for ISO/IEC 7816 standard is reported in table 3.5.

The main sections of interest are 4, 7, and 8. We extract and report in sections 3.2.1 to 3.2.4, the information we use for our project from the ISO/IEC 7816 standard.

ISO 7816 Section	Description
1	Physical Characteristics
2	Cards with contacts - Dimensions and location of the contacts
3	Cards with contacts - Electrical interface and transmission protocols
4	**Organization, security and commands for interchange**
5	Registration of application providers
6	Interindustry data elements for interchange
7	**Interindustry commands for Structured Card Query Language (SCQL)**
8	**Commands for security operations**
9	Commands for card management
10	Electronic signals and answer to reset for synchronous cards
11	Personal verification through biometric methods
12	Cards with contacts - USB electrical interface and operating procedures
13	Commands for application management in multi-application environment
15	Cryptographic information application

Table 3.5: ISO/IEC 7816 - Content Table

3.2.1 APDU Command-Response Structure

ISO/IEC 7816 section 4 [?] provides details regarding the structure behind the **application data protocol units** (APDU) command-response pairs. Commands have a maximum of 7 fields. The definition of each field is reported in table 3.6 and the command structure is reported in figure 3.1. Due to the optional fields Lc, Data, and Le, there are four possible structures of commands depending on the optional fields that are present. These different possibilities are reported in figure 3.2. Finally the structure of the response is reported in figure 3.3, with the meaning behind the fields reported in table 3.7.

Field	Description
CLA	CLA is the 'class' byte. This dictates if the command is inter-industry or proprietary *(We provide both examples in section 4.5.2)*
INS	INS is the instruction byte. This dictates what the instruction/ command is.
P1	P1 is a byte which is the first parameter to the command
P2	P2 is a byte which is the second parameter to the command
Lc	Lc is a byte that dictates the length of the data field. *(Maximum = 256)*
Data	Data is a field that has maximum length 256 bytes. It provides that data the command sends to the smartcard.
Le	Le is a byte that dictates the number of bytes expected in the response datafield *(default = 00)*

Table 3.6: Command Field Explanations

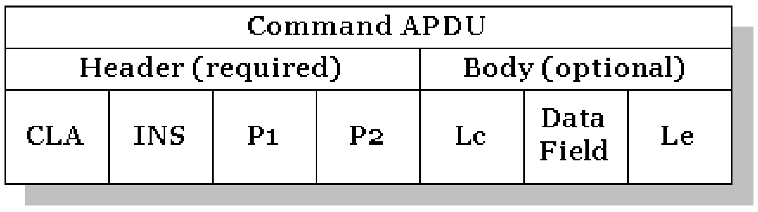


Figure 3.1: Command Structure [?]

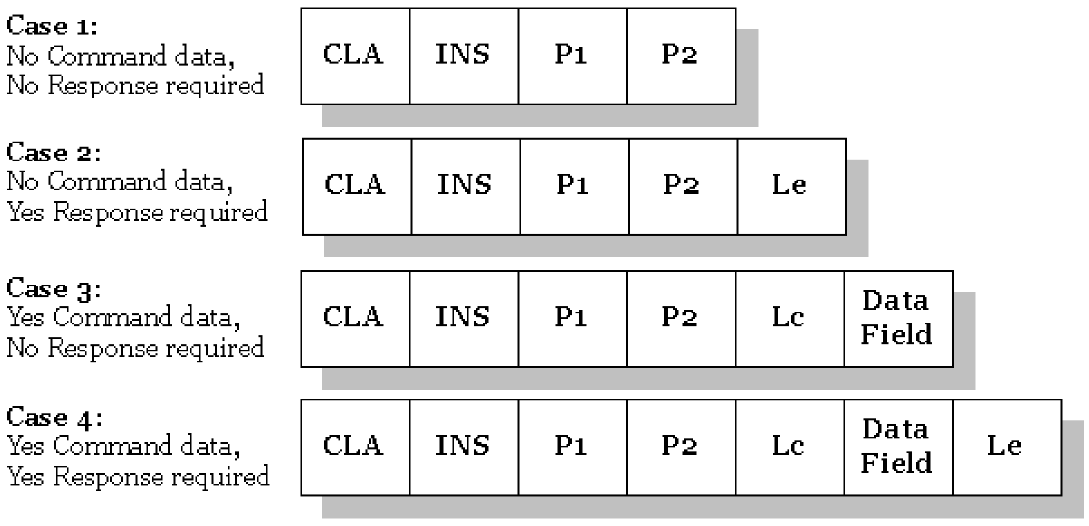
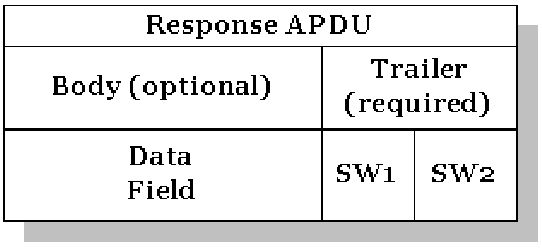


Figure 3.2: Different Command Possibilities [?]

Field	Description
Data	Data is a field that has maximum length 256 bytes. It provides that data the response sends back to the API.
SW1	SW1 is the first status bytes.
SW2	SW2 is the second status bytes.

Table 3.7: Response Field Explanations



Response APDU		
Body (optional)	Trailer (required)	
Data Field	SW1	SW2

Figure 3.3: Response Structure [?]

ISO/IEC 7816 - section 7 provides the definition of the response depending on status bytes SW1 and SW2. There are four main groupings for the types of response statuses. These are highlighted in figure 3.4 (XX dictates any hexadecimal value can be present). Due to the substantial volume of decodings for the status bytes, we only provide their exact decoding if and when required throughout this project.

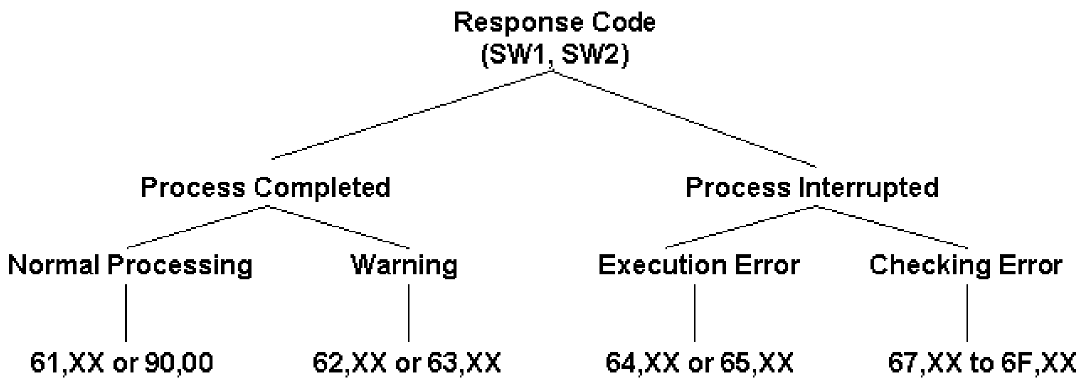


Figure 3.4: Response Definitions [?]

3.2.2 Inter-Industry and Proprietary Commands

The ISO 7816 standard defines two sets of different commands, inter-industry and proprietary. The CLA byte dictates if the command is either inter-industry or proprietary. Inter-Industry commands are standardized and reported in the ISO 7816 documentation. Proprietary commands are created by card vendors with the definition of each data field kept as proprietary information, including the commands response. Details regarding both inter-industry and proprietary commands for our project's smartcard are provided in table 4.4, chapter 4.

3.2.3 File System

As defined in section 4 of the ISO/IEC 7816 standard, contact smartcards have a standard for their file systems. A master file (MF) is the root of the file structure. This is analogous to the C drive for windows operating systems on computers. Under the MF, there are dedicated files (DF) and elementary files (EF). DF's are directories (similar to folders) and have EF's contained within them. EF's are files consisting of data. There are four types of files that EF's support. These are reported below and in figure 3.5.

1. Transparent Binary File
2. Linear structure with records of fixed size
3. Linear structure with records of variable size
4. Cyclic structure with records of fixed size

Elementary Files (EF) also take on two different formats. These have been discussed in chapter 6. *The explanations of the formats have been sourced from the ISO 7815 standard [?]*:

1. "An internal EF stores data interpreted by the card, i.e., data used by the card for management and control purposes."
2. "A working EF stores data not interpreted by the card, i.e., data used by the outside world exclusively."

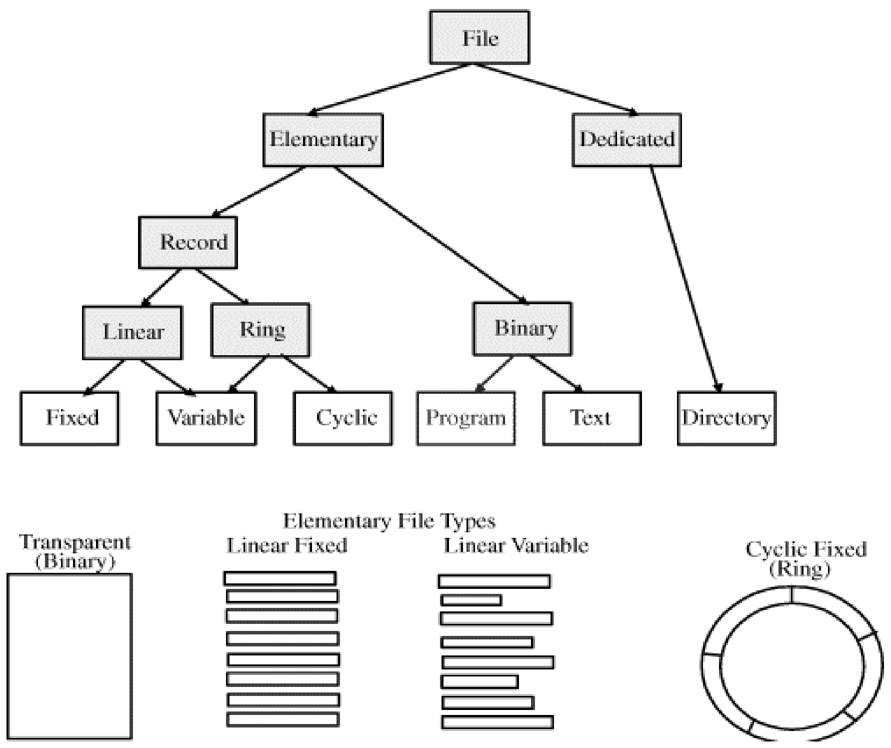


Figure 3.5: File Structure [?]

3.2.4 Secure Messaging

ISO/IEC 7816 section 4 reports details of the command structure of messages that occur within a 'secure messaging session'. We provide an example of the command structure in table 4.4, chapter 4 (the CLA byte is equal to 8C).

Secure messaging sessions derive two session keys S_{Enc} and S_{Mac} between the API and the smartcard. The S_{Enc} session key is used to encrypt the data-field of commands within a secure messaging session. S_{Mac} is used to compute a checksum (using C-MAC) of the entire command. This checksum is appended to the command. This provides confidentiality (only the systems in question should be able to read the actual messages) and integrity (data has not been altered in transit) to command-response pairs in a secure messaging session.

Chapter 4

Literature Review

This chapter provides a review of the literature relating to the area of computer security especially with regard to PKCS #11 and ISO 7816 for contact smartcards. We first summarise the findings of four papers and then in one case examine if our project's smartcard has the same vulnerability as that reported in one of these papers. We also extend some of the work of Lan (2015) [?] (who also studied the same smartcard), with regard to attribute and key file locations in the smartcard's memory. We also summarise all of the inter-industry and proprietary APDU command definitions for our smartcard.

4.1 APDU-level attacks in PKCS #11 devices

Bozzato, Focardi, Palmarini and Steel (2016) [?] examine new attacks at the APDU level with respect to a generated threat model of the PKCS #11 middleware. The authors investigate five commercially available smartcards (some were usb tokens), that do not use encrypted APDU's (secure messaging). Their key findings are summarised below.

- Two smartcards use plain PIN authentication and therefore sent the PIN to the smartcard at the APDU layer in plain text
- Three smartcards implement a challenge-response algorithm for PIN authentication. Two challenge-response algorithms are reported to have been reversed engineered, as a proof of concept that an attacker can then brute force the PIN with the knowledge of the algorithm and one successful login trace.
- All smartcards that are investigated do not implement any protection from man in the middle attacks.
- Three of the smartcards do not preform encryption and decryption on the smartcard, and instead delegate it to the middleware. To do this

they send the sensitive symmetric key value in plaintext at the APDU level to the API.

- One of the smartcards is usb token from the same company 'Athena smartcard solutions'. This device shows the following protocol has been implemented to generate block cipher keys. The middlware/API requests and receives 8 random bytes from the smartcard. The middlware/API requests and receives the smartcard's (not users) RSA public key. The middlware/API then encrypts (using the RSA public key) the entire Elementary File (EF) to store the symmetric block cipher key. The authors find that the key value is the requested random bytes that had been sent in plaintext. So despite the card manufacturer going to significant effort to protect the key value as its being stored, it had already been revealed at the APDU level.
- In all five smartcards the ability to bypass PKCS #11 attribute controls was achieved by sending the APDU commands for cryptographic operations and skipping the middleware (API) all together.

The authors conclude that to mitigate the attacks, all cryptographic operations should be implemented within the hardware security module (this is however very costly as a redesign is required for most of the devices analysed in the paper). They also recommend a second mitigation which is to run the middlware layer (PKCS #11 implementation) on the computer as a separate process at a different privilege to the user. They report that this prevents an attacker from attaching and observing/altering APDU commands. Finally they conclude that for devices with a display unit (not smartcards), the one time password functionality (OTP) should be used in the PIN authentication stage. They argued that this will raise significant difficulties in reverse engineering the authentication protocol and will also require access to the user's device, even if the protocol is reversed engineered.

4.2 REPROVE

Gkaniatsou, McNeill, Bundy, Steel, Focardi and Bozzato (2015) [?] paper reports a "proof of concept system for automatically analysing APDU low-level communication" named Reverse Engineering PROtocols for Verification (REPROVE). REPROVE uses first-order logic, information with regard to the PKCS #11 standard and the ISO 7816 standard to reduce the combinatorial search space. This enables the reverse engineering of the proprietary APDU commands with respect to the PKCS #11 functions that use them. It accomplishes this without the need to have physical access to the smartcard, nor their proprietary middleware (implementation of the PKCS #11 API)

Gkaniatsou et al. (2015) analyse five commercially available smartcards.

They find some smartcards revealing users PIN's and sensitive cryptographic keys in plaintext. They also find trivial (static) authentication methods for C_login. This consequently allowed replay/ man in the middle attacks to take place on the smartcards.

4.3 On the Security of PKCS #11

Clulow (2003) [?] finds the 'wrap/ decrypt attack'. The attack occurs at the API level and extracts sensitive key values of symmetric and asymmetric keys in plain text. The CKA_extractable attribute must be set to true, but other PKCS #11 attributes are not taken into consideration. Therefore in the case that CKA_sensitive is set to false, the attack directly contradicts the controls the API is expected to provide. The author proposes the following steps to conduct the attack.

1. Generate a new block cipher key K_1, with attributes C_wrap and C_decrypt set to True

2. Use K_1 to wrap the existing key K_2 within the token, and extract it in an encrypted format

3. Use K_1 to decrypt the wrapped key and expose its value in plain text

This can also be achieved with asymmetric keys instead of block cipher keys. Setting the public key to be able to wrap and the private key to be able to decrypt.

4.4 A Study On The Same Smartcard

Lan 2015 [?] carries out his work on the same smartcard as the one we use in our project (Athena IDprotect). We summarise the main contributions of his study in table 4.1. We also report in detail the findings of the paper (more so than other papers). The rationale for doing this is to clearly distinguish between his work and ours.

Finding	Description
1	Finding the meaning behind proprietary ISO 7816 commands for the smartcard analysed
2	Decoding of attribute files
3	Discovery of attribute and key file separation for the smartcard analysed
4	Discovery of *block cipher* key file locations
5	Discovery of a counter that is used upon file creation and deletion
6	Discovery of an implementation flaw regarding decryption using block cipher AES in ECB mode
7	Discovery that the functions *C_wrap* and *C_copyObject* are not supported by the manufactures middleware that implements the PKCS #11 functions
8	Attack$_1$ modifies the values of CKA_extractable and CKA_sensitive from false to true and true to false respectively. (This is not premitted by the PKCS #11 API)
9	Attack$_2$ modifies the key value of a block cipher key that is already stored within the smartcard.
10	Attack$_3$ finds the ability to use block cipher keys regardless of authentication status of the user, with CKA_private set to True. (This is not premitted by the PKCS#11 API)

Table 4.1: Summary of Findings

Lan (2015) [?] used the *pcsc-spy* tool. This tool is reported in section 5.1, chapter 5 of our project. The tool gave a good understanding of the proprietary commands implemented within the middlware. The inter-industry commands are standardized and can be found in the ISO 7816 specification. It is worth noting that we do not replicate this work as the details the proprietary command are listed in the appendix of the Lan's paper. We extract this information from this appendix and summarise all of the commands in section 4.5.2.

The second finding relates to the decoding of the attribute files. The attribute files in the Athena IDprotect smartcard are saved in a transparent binary file. The values in the file can be viewed in hexadecimal format at the APDU layer. Lan reports that each PKCS #11 attribute has a hexadecimal encoding. We extract the examples given in his paper and report them in table 4.2. The paper states that the representation of each attribute is stored in the attribute file in the following format:

Hexidecimal Encoding + Filler + Length + Value

Attribute	Hexadecimal Encoding
CKA_class	0x0000
CKA_private	0x0002
CKA_label	0x0003
CKA_value	0x0011
CKA_id	0x0102
CKA_sensitive	0x0103
CKA_modulus	0x0120
CKA_public_exponent	0x0122
CKA_private_exponent	0x0123
CKA_extractable	0x0162

Table 4.2: Attribute Encodings

The third finding is attribute and key file separation. Lan reports that despite there being fields in the attribute files to store sensitive values (see table 3.2, chapter 3 of our project), those fields were filled with 'FF' hexidecimal values and return 'NULL' at the API layer. We assume that this method (attribute and key file separation) is to patch the vulnerability that has been present in an earlier version of 'Athena smartcard solutions' hardware security module. This vulnerability is found in [?] and summarised in our literature review.

The forth finding concerns the memory locations on the smartcard for block cipher keys and attribute files. Lan reports that '00 CX' (X being any hexadecimal value) is the location of the EF's that store key values, for all type of keys. However, our analysis discussed in chapter 6 finds that this is not in fact the case. The location of block cipher keys is the same as what Lan reports, however we find different locations for RSA public and private keys. We report this in section 4.5.1, as part of our extended work.

The fifth finding is a counter file for symmetric and asymmetric keys. This value of the counter is incremented upon file deletion. On a new key creation/ generation, the counter is appended to the attribute file of the keys. In Lan noted that there is no indication of the value being used (e.g. to check if a file has been created not by the API) but the counter is present. We discuss this in chapter 6 of our project.

The sixth finding is an implementation flaw in the middleware that 'Athena Smartcard Solutions' created to implement the PKCS #11 API. The flaw is regarding the use of decryption with the block cipher AES in ECB mode. The wrong command is sent and thus returns an error '6A 86', meaning 'Incorrect P1 or P2 parameters'. Lan corrects the command and sends it to the smartcard which decrypts the ciphertext correctly. He points out that this is not a security vulnerability and rather an issue with functionality of the API. We also find a similar issue with the use of triple DES keys and discuss this in chapter 6 of our project.

The seventh finding concerns the removal of C_wrap and C_copyObject functions from the card vendors middleware that implements the PKCS #11 API. We believe that this could have been done in an attempt to prevent the 'wrap/decrypt attack', Clulow (2003) [?].

The eighth finding is attack$_1$. The attack modifies the attribute files at the APDU level by sending the 'UPDATE BINARY' command to alter the attributes that are stored on the card. The attributes altered are CKA_sensitive and CKA_extractable from true to false, and false to true respectively. This contradicts the PKCS #11 API as it is not permitted. Lan succeeds in altering the attribute values, but it yielded no security vulnerability. While a real change to the attributes are reported at the API level, the value of the sensitive key is reported to still be 'None'. We assume that when a key is created/ generated with CKA_sensitive set to true and CKA_extractable set to false, the key value is stored in an internal EF (discussed in section 3.2.3, chapter 3 of our project). If this assumption holds it explains why the key value cannot be extracted regardless of the change in the attribute values. We discuss this further in our chapter 6, but we do not replicate the attack as Lan reports no security issues.

The ninth finding is attack$_2$. The attack modifies the key value of already generated/stored block cipher keys. The attack consists of five elements:

1. Using the information regarding block cipher key file memory location on the card (see section 4.5.1) locate an already stored key$_1$ on the card, and its corresponding at attribute file.
2. Open the attribute file ('SELECT FILE' and 'READ BINARY') and save the hexadecimal values of the attributes
3. Delete both attribute and key files ('SELECT FILE' and 'DELETE FILE')
4. Create a new block cipher key$_2$ (of the same type) with C_createObject as to be able to set the sensitive key value (to a value known by the attacker).
5. Modify the attribute file of key$_2$ to be equal to the attribute of key$_1$ ('UPDATE BINARY')

It is difficult to understand how the above attack operates using the explanations provided by Lan. However, after studying the APDU traces given in the appendix of his paper, we are able to find how the attack is implemented. The above attack may not be valid unless it takes place as soon as a new block cipher key is generated by the user using C_generateKey (it cannot be created by the user using C_createObject, as they will be aware of the key value). This is because the attack would noticed by the user almost immediately. An example is whereby the user had already used the key to encrypt data for storage or communication purposes. The resulting decryption after the key file had been modified would not result in the decryption they would expect. Thus, alerting them to the fact that the key has been changed. It is for this reason we do not to replicate this attack. We rather focus on attacks that would find the key values stored on the smartcard without the users knowledge and without modifying it in a manner where the user would be alerted to it.

CKA_private with a value of true dictates that a user must be logged in to use the key for security operations. The final attack (attack$_3$) suggests that keys with the attribute CKA_private set to true can be used for security operations regardless of the user's logged in status. This is because the knowledge of the key file location in the smartcard's memory, and the commands required to be sent to the card for security operations are known. Thus, he suggests using 'SELECT FILE' command to select the key in question, and then send any of the security operation commands (e.g. 'ENCRYPT') to use the key while being not logged in.

We attempt to test if this attack is valid, but for an unknown reason received API segmentation faults when generating a block cipher key with the CKA_private and CKA_token attributes set to true. Therefore we could not create/ generate a persistant block cipher key on our smartcard with CKA_private set to true. This prevented us from verifying the validity of attack$_3$. We did however test a very similar attack that is reported in Bozzato et al. [?]. This attack also overrides API attribute controls by sending the commands directly at the APDU layer.

4.5 Additional Work

4.5.1 Directories of Key and Attribute Files (Extended Work)

Lan 2015 [?] finds that attributes and sensitive key values are stored within separate files in the smartcard. His results also show the location of these attribute files and key value files for **block cipher** keys. He assumes that asymmetric attribute and key value files are stored within the same directory that he reports for block cipher keys.

We find in chapter 6 (section 6.6) that Lan's above assumption is not valid. This is due to the fact that RSA public and private, attribute and key files have different directories for the EF's that store their respective values. We extend hid work to include the locations of RSA attribute and key files. We report results of the locations in table 4.3.

Key Type	File Type	Directory
Block cipher	Attribute	3F 00 30 00 30 **01 03 4X**
	Key	3F 00 30 00 30 **01 00 CX**
RSA public	Attribute	3F 00 30 00 30 **02 01 4X**
	Key	3F 00 30 00 30 **02 00 8X**
RSA private	Attribute	3F 00 30 00 30 **02 01 4X**
	Key	3F 00 30 00 30 **02 02 0X**

Table 4.3: Directories of Key and Attribute Files

4.5.2 APDU Command Definitions (Summary)

In chapter 6 we analyse the PKCS #11 functions and how they are broken down into multiple ISO 7816 command-response pairs (for our project's smartcard). We find as did Lan the use of the following inter-industry and proprietary commands. We provide a summary of all of the commands below.

[Lan 2015 [?] finds the 'inner-function' names which provide a good decoding of the proprietary commands listed in table 4.4. He has used a program known as 'pcsc-spy' to find the 'inner-function' names. We discuss this tool in section 5.1]

CLA	INS	P1	P2	Lc	Data Field	Le	Description
00	-	-	-	-	-	-	Inter-industry (II)
80	-	-	-	-	-	-	Proprietary (P)
XC	-	-	-	-	-	-	Secure Messaging (SM)
00	84	00	00	-	-	L	(II) Get Challenge [# bytes = L]
00	B0	X	X	-	-	L	(II) Read Binary [# bytes = L]
00	C0	00	00	-	-	L	(II) Get Remaining Bytes
00	D6	X	X	L	X	-	(II) Update Binary
00	E0	X	X	L	X	-	(II) Create Binary
00	47	00	00	L	Params	-	(II) Generate RSA KeyPair
00	E4	00	00	00	-	-	(II) Delete File
00	2A	82	05	L	X	00	(II) Encrypt Data
00	2A	80	05	L	X	00	(II) Decrypt Data
00	2A	9E	12	L	X	00	(II) Sign Data
00	2A	80	0A	L	X	00	(II) Unwrap Key (RSA_PKCS)
80	A4	08	00	L	FL	-	(P) Select File [FL = file location; L = len(fl)]
80	A4	00	00	-	FL	-	(P) Select File [append previous path]
80	A4	08	0C	L	FL	-	(P) Read File Control Parameters
80	20	00	00	10	R	-	(P) Verify PIN [R = Response]
80	28	00	00	04	00 00 00 20	-	(P) Clear Security Status
80	30	01	00	00	-	-	(P) List Files
80	48	00	80	00	-	-	(P) Get The Smartcards Public Key [SM]
80	48	00	00	00	-	-	(P) Get Generated Public Key
80	86	00	00	L	X	00	(P) Open Secure Messaging
80	86	FF	FF	-	-	-	(P) Close Secure Messaging
90	32	00	03	ff	X	-	Reallocate Binary 256 bytes
80	32	00	03	L	X	-	Reallocate Binary L bytes (changes file)

Table 4.4: APDU Commands for Athena IDprotect

4.5.3 Manually Overriding Attribute Controls (Replication Study)

One of the APDU level attacks reported by Bozzato et al. (2016) [?] involves sending APDU commands to the smartcard, bypassing the controls that attributes at the PKCS #11 API level is expected enforce. We test the same attack on our smartcard to find out if it has the same vulnerability as all five hardware security modules that are examined in the above paper.

To test this vulnerability, we generate a triple DES symmetric key on the smartcard and set CKA_encrypt attribute to false. First, we request the PKCS #11 API to encrypt the string 'TestString123456'. The API returns the following error 'CKR_KEY_FUNCTION_NOT_PERMITTED'. This states that the function is not supported due to the attribute settings. Second, we send the APDU command 'SELECT FILE' (reported in table 4.4) to select the triple DES key we have generated. Then we send the APDU command for the security operation 'ENCRYPT' to attempt to override the attribute controls that the API enforce.

Our result is consistent with the findings of Bozzato et al. (2016) [?]. The APDU command executed successfully and resulted in the encryption taking place on the card, and returning in the APDU response, the value of the encryption. The APDU communication trace of this test is reported in appendix A.4.1.

Chapter 5

Tools Developed/Used In This Project

This chapter explains the details regarding the use of pre-existing software packages. We also explain how we extend and develop the virtual smartcard reader for tasks in our project (specifically for our smartcard).

5.1 PCSC-Spy

One PKCS #11 function is broken down into multiple APDU command-response pairs. The APDU commands are sent from the middleware to the PC/SC layer on the computer, and then to the smartcard. This is illustrated in figure 5.1.

Rousseau [?] has created the open-source implementation of the PC/SC layer for computers using the linux operating system (known as PCSC-lite). As part of the implementation, he has created the *pcsc-spy* program. This effectively implements a man in the middle attack between the middleware and the PC/SC layer. Using this program it is possible to sniff the 'inner function' names of all the communication between the middleware and PC/SC layer (inner functions are functions within each PKCS #11 function that are called to send one command). By extracting this information, the program gives a good understanding of the proprietary ISO 7816 commands.

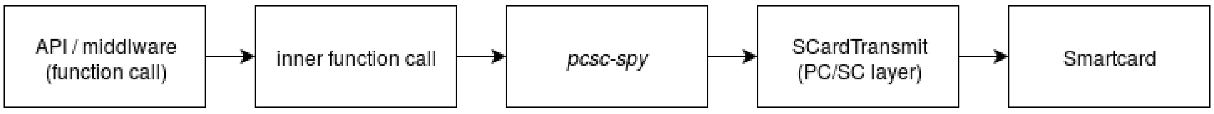


Figure 5.1: pcsc-spy implementation

5.2 Virtual Smartcard Reader

Morgner [?] has created the virtual smart card reader project. The project emulates a smartcard reader and creates a virtual one in the host machine. It also has the 'Relay OS' which has a basic implementation of a man in the middle attack provided in the code of the project (no APDU command response pairs are altered, the ability is just present). This can be seen in figure 5.2.

"The vpcd is a smart card reader driver for PCSC-Lite [2] and the windows smart card service. It allows smart card applications to access the vpicc through the PC/SC API. By default vpcd opens slots for communication with multiple vpicc's on localhost on port 35963 and port 35964. But the vpicc does not need to run on the same machine as the vpcd, they can connect over the internet for example.

Although the Virtual Smart Card is a software emulator, you can use PC/SC Relay to make it accessible to an external contact-less smart card reader." [?] - This is the explanation provided on his github page.

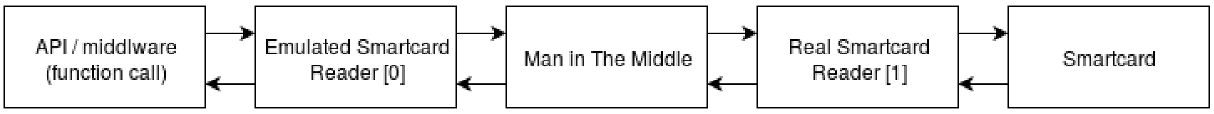

API / middlware (function call)
Emulated Smartcard Reader [0]
Man in The Middle
Real Smartcard Reader [1]
Smartcard

Figure 5.2: Man in The Middle implementation

5.3 Man in The Middle Tool (MiTM)

We use the Virtual Smartcard Reader project discussed above. We develop on top of the project's code to implement a software man in the middle attack for a variety of applications our project. The developed code includes both APDU commands and responses as printed output and uses the python package 'hexdump' for clarity in the printed APDU communication traces. We also added the functionality of altering the APDU communication from the terminal. Some of the applications for which the developed code is used for in our project are:

- Bypassing attribute controls by sending the APDU commands for secuirty operations (discussed in section 4.5.3, chapter 4)
- Altering the challenge sent by the smartcard in PIN authentication (discussed in section 7.2, chapter 7)
- Injecting our own response in PIN authentication (discussed in section section 7.2.3, chapter 7)
- Altering bytes of the smartcards public key to derive the protocol used in secure messaging (discussed in section 7.3, chapter 7)

Chapter 6

PKCS #11 Functions - APDU analysis

As discussed in chapter 2, PKCS #11 API functions are broken down into multiple ISO 7816 command-response pairs via the use of the card vendors middleware *(their implementation of the PKCS #11 API)*. This implementation is at the manufacturer's discretion, who may wish to use proprietary commands. The aim of this chapter is to analyse the APDU traces of 9 PKCS #11 functions, and how the card manufacturer has decided to split them into command-response pairs.

This is because previous literature has shown that some poor implemenations has revealed PIN's and encryption keys at the APDU layer which heavily violate the PKCS #11 standard. Our analysis will include a step by step guide as to how the functions are broken down into command response pairs at the APDU level. In addition, the analysis will provide explanations of how the implementation operates and therefore be able to suggest possible vulnerabilities, some of which will be investigated in chapter 7.

All the traces are reported in appendix B. These traces have been shortened to only include the command-response pairs which are deemed to be the most important to the evaluation of a function. However, the full traces for each function are given within the project directory. We will refer to appendix B including the corresponding command response pair which are given in brackets to allow ease of locating the steps throughout our analysis.

C_sign and *C_verify* are not included in this analysis as earlier work [?] on this card show that both of these functions operate in a similar manner to C_encrypt and C_decrypt. *C_createObject* is also not included in this analysis, because it operates in a similar fashion to *C_generateKey* and *C_generateKeyPair*. The only difference is that the key values are provided by the user rather than being computer generated.

To prevent repetition we list the dependencies (other functions that must be run before hand) at the start of each section when necessary.

It is worth stressing that chapter 6 will only provide explanations of the possible vulnerabilities that might be present. We leave all the investigations of them to chapter 7.

6.1 Initialization

Initialization is process whereby the smart-card sends 2 serial numbers to the API, in order for the API to have the knowledge of which model of smart-card it is to communicate with. This occurs as soon as a session is opened with the card. The process has the following steps:

1. Open the laser PKI file and select the applet (B.1 - command 1)
2. Send to the API the card serial number (B.1 - command 4,5)
3. Send to the API the 2nd serial number (B.1 - command 8,9)

The trace doesn't reveal sensitive information. These steps are required in order for the card to operate and communicate with the API. The API is a generic library that works with many different hardware security modules created by the 'Athena smartcard solutions'. Hence the serial numbers are required by the library in order for it to operate correctly for the specific type of card.

6.2 C_login

Function dependencies → [Initialization]

The login function has 5 main components (these are listed below).

1. Open file control parameters for file X_1 (B.2 - command 10)
2. Select file X_2 (B.2 command 11)
3. Request challenge from the card (B.2 - command 12)
4. Use the proprietary VERIFY command for verification of PIN value (B.2 - command 13)
5. Clear security Status (B.2 - command 20)

The API requests to view the file control parameters for file X_1. The file location as stated within the trace is at '3F 00 00 20 00'. This holds the retry counter value, which dictates how many attempts are left before the card is blocked *(A value of 'A0' states the card is currently in a blocked* *status)*. To make it easier to locate within the trace we have highlighted the current value in bold which is of ’AA’, meaning there is 10 attempts left.

Following this check, file X_2 is selected. We assume that this file holds a pointer to EEPROM memory where the PIN is securely stored by the smartcard. This file cannot be accessed with the inter-industry command ’READ BINARY’ nor the proprietary command ’READ FILE CONTROL PARAMETERS’. However if this file is not selected PIN verification fails despite having the correct PIN value. This finding was reported by (enter name) last year [?].

After file X_2 has been selected, the API requests the smartcard to send an 8 byte challenge. Upon receiving said challenge, the API then responds with a 16 bytes of data for PIN verification using the proprietary ’VERIFY’ command. Assuming the correct PIN has been supplied the user is now authenticated with the card, and the last step clears the security status.

The evaluation of multiple traces of the login function shows that the response calculated appears to have an element of randomness. From studying the traces alone, the method of calculating the value of the response, given the PIN and the smartcards challenge, cannot be intuitively determined.

No immediate sensitive information is revealed in this trace, due to the use of the challenge-response algorithm used for verifying a user’s PIN. However, if an attacker can successfully reverse engineer the challenge-response protocol, the PIN can be calculated given 1 successful login trace. Furthermore, if an attacker can find a method for changing the file control parameters, they might be able to reset the retry counter at the APDU level to allow an unlimited number of attempts of logging in. These possible vulnerabilities will be discussed in detail in chapter 7.

6.3 C_findObject

[For this trace to show meaningful information we have generated and stored a triple DES key onto the card, before running the findObject function. The label of the key was ’des3’ with id ’01’]

Function dependencies $\rightarrow$ *[Initialization, Login]*

The findObjects function is split into 3 main components (these are listed below).

1. Select and open ’cmapfile’ [List files command]. This stores all file locations for attributes of the keys (B.3 - command 32, 33, 34)

2. Using the data from 'cmapfile', Open the first attribute key file (B.3 - command 36)
3. Open the next attribute key file. (Which is empty) (B.3 - command 38, 39)

First, a cmapfile is accessed. This file stores the location of key attribute files. Following this, attribute files are accessed. The attribute files store all the information regarding a specific keys attributes (listed in section 3.1.1). Finally the next (and last) file is opened. In the case more than one key was stored on the card at the time of running this function the next file would contain the attributes of the next key. However, since there is only 1 key, the next file is empty (containing all zero's) thus determining there are no more key objects to be found.

Due to key and attribute files being stored separately, this function call reveals no sensitive information. This finding was reported by (name) 2015 [?] and is stated within our literature review. Only the attributes are opened and listed at the APDU layer. These attributes can be printed at the API layer as well.

A possibility for an attack is present here. The ISO 7816 proprietary 'REALLOCATE BINARY' and the inter-industry 'UPDATE BINARY' commands can be used to modify a file. This will allow modification of attributes.

As reported in the literature review, the work by (name) 2015 [?] on the same card shows that they were able to modify any attributes. They tested the modification of CKA_sensitive & CKA_extractable from false to true. The change in this direction is not permitted by the PKCS #11 standard. However the ability to change these at the APDU level was achieved. This approach still yielded no significant results, as keys and attribute are stored in separate files, thus the keys could still not be loaded. This is discussed in chapter 7.

6.4 C_generateKey

[For this trace we generate a triple DES key, key length of 24 bytes.]

Function dependencies $\rightarrow$ *[Initialization, Login]*

The generateKey function has 9 main components (these are listed below).

1. Open secure messaging (B.4 - commands 26, 27, 28)
2. Generate 24 random bytes and send them to the API via secure messaging (B.4 - command 29)

3. Close secure messaging (B.4 - command 30)
4. commands skipped include finding spare file for attributes and opening the file.
5. Update file with key attributes (B.4 - commands 40, 41)
6. Open secure messaging [again] (B.4 - commands 42, 43, 44)
7. Open key file directory via secure messaging (B.4 - command 45)
8. Create key file for triple DES key via secure messaging (B.4 - command 46)
9. Close secure messaging (B.4 - command 47)

Secure messaging session have an initialization process whereby the API and smartcard generate 2 session keys S_{Enc} and S_{Mac}. Once the initialization is completed the following communications data fields are encrypted with S_{Enc} and a checksum is computed via C-MAC (see section 2.4.2) using S_{Mac}. They provide confidentiality and integrity of the data being sent within the data fields of the commands during a secure messaging session.

Two different secure messaging sessions are used. The first is to request the card to generate 24 bytes (same as the key length) and send it to the API. The second is to place the key value into the key file. The attribute file is created without the use of secure messaging as it does not contain sensitive information.

The analysis of the communication trace suggests that the card is requested to generate the key value from a 'GET CHALLENGE' request (within secure messaging). We assume that the generated bytes are used to be stored in the key file as the key value. At this stage it is not possible to verify this. The reason being that two different secure messaging sessions are used, and therefore have different session keys. This causes the encryption of the generated bytes and the storing of them in a file, to be different. Despite the possibility they might be identical.

As discussed in the literature review chapter, previous hardware security modules created by 'Athena smartcard solutions' have in the past requested the card to generate random bytes and use them as the key value (when generateKey function is called). This was completed without the use of secure messaging. Thus, it appears that they might be using a similar approach but with the use of secure messaging (to prevent revealing the encryption keys in plain text at the APDU layer) on this smartcard.

If the assumption of the generated bytes being used as the key value holds, it may result in a significant vulnerability that could be exploited. This will only be valid if the secure messaging protocol (that is a proprietary implementation by the card vendor) can be reversed engineered. This should allow an attacker to inject a new key value or decrypt the key value to be stored. This would be in breach of the PKCS #11 standard, depending on attribute values set.

A second vulnerability is also present. Unlike previous attacks whereby attributes are modified after a key has been created. An attacker could modify the attributes during the key generation process (this would take place at step 5, in the component list above). Changing attribute values would allow for the key to be extracted as soon as it is created (this has been checked). It is worth noting that the user will be aware of it, as it is printed at the API layer. In addition they will also be able to see the modified attributes which they had not set. Thus we determine this to be a vulnerability, but not as serious as the previous one mentioned. As if it were utilised, the user would most likely notice and therefore generate a new key to be used.

Both of these possible vulnerabilities are discussed in chapter 7.

6.5 C_generateKeyPair

[For this trace we generate an RSA-1024 public/private key pair]

Function dependencies $\rightarrow$ [Initialization, Login]

We find 15 main components in this generation of the RSA public and private key pair (these are listed below).

1. Create public key attribute file (B.5 - command 54)
2. Add public key attributes to file [inc. public exponent] (B.5 - commands 56, 57)
3. Create private key attribute file (B.5 - command 59)
4. Add private key attributes to file (B.5 - commands 61, 62)
5. Create Private CTR RSA Key file (B.5 - command 64) [ref crypto section] [?]
6. Select temporary file (B.5 - command 65)
7. Generate RSA Key Pair (B.5 - command 66)
8. Select temporary file (B.5 - command 67)
9. Get RSA public key (B.5 - command 68)
10. Select parent folder of private key attribute file (B.5 - command 69)
11. Create new file, with public modulus and additional info (B.5 - command 70

12. Select temporary file (B.5 - command 71)
13. Get RSA public key (B.5 - command 72)
14. Select public attribute file (B.5 - command 73)
15. Add public modulus to attribute file (B.5 - command 74)

The rest of the communication we believe to be resetting the temporary file, and adding file location information to file control parameters of parent directories. (These commands are not included in the traces within the appendix)

Components 1, 3, 5, 7, 9, 11, and 13 transfer data that are wrapped within an ASN.1 BER encoding [?]. We have used an online tool [?] to decode the data fields. The following are the ASN.1 BER decodings of the commands that have these wrapping.

1. Create public key attribute file (B.5 - command 54)
Application 2(4 elem)
[10] (1 byte) 04
[03] (2 byte) **01 40**
[00] (2 byte) 01 A7
[06] (8 byte) 00 20 00 20 00 20 00 20

3. Create private key attribute file (B.5 - command 59)
Application 2(5 elem)
[10] (1 byte) 04
[03] (2 byte) **02 00**
[00] (2 byte) 01 23
[04] kx s0
[06] (8 byte) 00 00 00 20 00 20 00 20

5. Create private CTR RSA key file (B.5 - command 64)
Application 2(6 elem)
[10] (1 byte) 04
[03] (2 byte) **00 41**
[00] (2 byte) 00 80
[05] (5 byte) 05 0C 20 00 A3
[06] (14 byte) 00 00 00 FF 00 FF 00 20 00 20 00 00 00 20
Application 17(0 elem)

7. Generate RSA key pair (B.5 - command 66)
[12] (2 elem)
[00] (1 byte) 06

[01] (3 byte) 01 00 01

9,13. Get RSA public key (B.5 - commands 68, 72)
Application 73(2 elem)
[01] (128 byte) D1 EF 7C A5 06 A1 87 FD 5F 13 5B 25 B7 16..
[02] (3 byte) 01 00 01

11. Create new file with public modulus and additional info (B.5 - command 70)
Application 2(6 elem)
[10] (1 byte) 04
[03] (2 byte) **00 81**
[00] (2 byte) 00 80
[05] (5 byte) 05 08 20 00 A3
[06] (14 byte) 00 00 00 FF 00 FF 00 20 00 20 00 00 00 20
Application 17(2 elem)
[16] (3 byte) 01 00 01
[17] (128 byte) D1 EF 7C A5 06 A1 87 FD 5F 13 5B 25 B7...

RSA key pair generation occurs on the card. A dedicated processor is used for this, hence the need to change to temporary files to access the generated key and then store the public information. We notice no sensitive information being revealed within the traces in plain text. We are of the opinion that the information provided that is wrapped within the ASN.1 BER encoding, especially for **5**, might have exported the private key and the additional parameters required for CRT RSA keys [?]. To test this theory we have deleted the generated key and generated another RSA key. This resulted in the public modulus of the RSA key to change, but the remaining parameters did not. A change in the public modulus would cause a change in both the private exponent and CRT parameters. This leads us to conclude that no part of the private key is exported in plain text within the communication traces.

As we will explain in the destroyObject function analysis, when a key is deleted from the card, the location in memory where the keys are securely stored (i.e. cannot be read) is revealed. We have used the destroyObject function to delete both RSA public and private keys. This enabled us to find the locations of the key and attribute files for both keys. These results are reported below:

	Key File	Attribute File
RSA public key	3F0030003002 **00 81** (11. [03])	3F0030003002 **01 40** (1. [03])
RSA private key	3F0030003002 **02 00** (3. [03])	3F0030003002 **00 41** (5. [03])

Table 6.1: File Location Table

Our work shows that components 1, 3, 5, and 11 (listed above) are used to save the details of the location in the smartcard's memory of the RSA public & private attribute and key files. The corresponding componenets with the file location parameters are reported in brackets in the table 6.1. Components 9 and 13 reveal the RSA public exponent and modulus used to be saved into attribute and key files. Finally, component 7 is used to give the size of the RSA key and its public exponent for key pair generation on the card.

Therefore, the only vulnerabilities we notice are, the ability to modify attributes as the key is being generated (components 2 and 4). This was discussed within generateKey function analysis and did not prove to be a vulnerability that would not be noticed by the user. The second vulnerability is the disclosure of the file location of the private key. The opening and possible reading of the key value is discussed as part of the destroyObject function analysis and also in chapter 7.

6.6 C_destroyObject

[For this trace we delete/destroy a triple DES (key option 2) 16 byte key and its attribute file]

Function dependencies $\rightarrow$ [Initialization, Login, FindObjects]

Once the key and attribute files have been located and the user is authenticated with the card, there are 6 main components to the destruction of the object. These are listed below:

1. Select counter file for key attributes (B.6 - command 49)
2. Update counter (B.6 - command 50)
3. Select key file (B.6 - command 53)
4. Delete key file (B.6 - command 54)
5. Select key attribute file (B.6 - command 55)
6. Delete key attribute file (B.6 - command 56)

Work conducted by (name) 2015 [?] did show that the counter file maintains track of how many keys have been created and deleted in a certain path within the smartcard's memory. We are unsure about its specific use (except for maintaining track of the number of keys) as we have only seen it being updated upon deletion of an object, and the new counter value being appended to the new keys attributes (the one created after deleting an object). The original and updated counter value is reported in appendix B.6 - command 37 and 50 respectively.

Once the counter file has been updated, the key attribute file and the key file are selected and deleted using the inter-industry command 'DELETE FILE'. This is the first finding where the location of a key file is accessed without the use of secure messaging. The finding is consistent with the earlier work [?]. Thus we think that this may present a vulnerability which would allow us to open and read key values at the APDU level.

We attempt to use the 'SELECT FILE' command and then 'READ BINARY' command to try and read the files data. This resulted in an error message = '69 81', meaning the command is incompatible with the file structure. In out next step, we try the other command which is 'OPEN FILE CONTROL PARAMETERS'. This did successfully work. We find that the output is wrapped in an ASN.1 BER encoding, thus we decoded it this gave the following result:

Application 2(6 elem)
[07] (1 byte) 08
[03] (2 byte) **00 C1**
[00] (2 byte) 00 18
[10] (1 byte) 04
[06] (14 byte) 00 00 00 FF 00 FF 00 00 00 00 00 00 00 00
[05] (4 byte) 01 0C 10 00

The above decoding is similar to what we have reported for the RSA private key analysis in section 6.5. The 16 byte key that would be used for triple DES is not present. There is however pointers to the file that we just read the parameters of *(00 C1)*, and other additional parameters that we are unsure of their use.

All security policies [?] that we have reviewed concerning 'Athena smartcard solutions' suggest that keys are stored in internal EF's (cannot be accessed by outside actors, see section 3.2.3, chapter 3), and only when required are encrypted internally (by the smartcard's OS) and stored in the EEPROM (electrically erasable programmable read only memory) for them to be used for the security operation required of them.

Thus we did not find any security vulnerabilities from this function. And also eliminates one of the vulnerabilities we suggested within the generateKeyPair function analysis. We discuss further in chapter 7.

6.7 C_encrypt

[For this function we use a triple DES key to encrypt a string 'TestString123456'] using ECB mode]

Function dependencies → [Initialization, Login, FindObjects]

The commands given for this trace are actually repeated and therefore occur twice for one encrpytion of a given string. This is due to PKCS #11 library being programmed in C by the card's manufacturer. The resulting length of the encryption is assumed to be of an undetermined size (for block ciphers and even RSA this can be pre-calculated given the input length). However the implementation runs the encryption twice, the first run is to calculate the resulting length of the encryption, and the second run saves the result in a buffer that has been pre-allocated the correct number of bytes in a char array.

This function only has only 2 main components (these are listed below).

1. Select key file (B.7 - command 52)
2. Encrypt data (B.7 - command 53)

The key file is selected and then the encryption APDU command is called with the string in the data field. As explained in destroyObject function analysis, despite the key file location being revealed here, it does not present a vulnerability. The key file can neither be opened nor reveal the value of the key. Instead it provides pointers to other locations in the smartcard's memory that hold the key value securely. It is reasonable to assume (though unable to test or verify) that only the smartcard's operating system has access to these locations.

Therefore we find no evidence of other vulnerabilities that can be exploited.

6.8 C_decrypt

[For this trace we use the same triple DES key to decrypt the message we previously encrypted]

Function dependencies → [Initialization, Login, FindObjects]

The decrypt function operates nearly in an identical manner to the encrypt function. The only difference is that it simply changes one byte (P1 in the APDU command header) in command 65. The key is loaded and then the decrypt APDU command is sent, with the encrypted string placed in the data field. For the same reasons stated in the encrypt function, the APDU command is sent twice.

It also has 2 main components (these are listed below).

1. Select key file (B.8 - command 64)
2. Decrypt data (B.8 - command 65)

As the decrypt function operates in a similar fashion to the encrypt function, we find no vulnerabilities here either. However we did find a similar implementation flaw that has been reported in earlier work [?] regarding decryption using an AES block cipher key in ECB mode.

We use a triple DES key instead of AES, and find for an unknown reason the first 8 bytes are removed from the encrypted data. This only decrypts and returns the second 8 bytes. As this does not present a security vulnerability, but it is rather an issue with the functionality of the decrypt function we do not analyse it any further.

6.9 C_setAttribute

[For this trace we modify the label of the triple DES key from 'des3' to 'changed']

Function dependencies → [Initialization, Login, FindObjects]

Similar to encrypt and decrypt, this function has only 2 main components. The first one selects the attribute file, and the following 2 commands update the whole file to include the new name of the label to the key. The components are listed below:

1. Select key attribute file (B.9 - command 51)
2. Modify attribute file to change the label name from 'des3' to 'changed' (B.9 - commands 52,53)

Command chaining was used as the number of bytes within the attribute file exceeded the limit of 256. Command chaining is indicated by the use of '90' instead of '80' as the CLA byte. This was seen in the 'REALLOCATE BINARY' command that was used to modify the attribute file.

We notice that at the APDU layer attributes can be changed via the 'REALLOCATE BINARY' command. This is a vulnerability that was discovered and tested in previous work [?]. With the main focus on changing the attribute values of CKA_sensitive and CKA_extractable from true to false and false to true respectively. The change in this direction is not permitted by the PKCS #11 standard. However, this was conducted at the APDU layer. The tests carried out in the literature found that due to the split between attribute files and key value files, the key value could still not be extracted despite the modification to the attributes.

Thus, despite the vulnerability being present it does not appear to cause a major security issue, in the sense that the key value is still stored securely. We discuss this further in chapter 7.

6.10 C_unwrap

[For this trace we used an RSA public key to encrypt a triple DES key value = 12345678. With its encryption saved, we then create a template for the key attributes and used the RSA private key to unwrap the key value and template, to save the key on the card]

Function dependencies → [Initialization, Login, FindObjects, Encrypt]

There are 10 main components involved in the unwrapping of a key. 2 different secure messaging sessions are used (just like in the generateKey function). The first secure messaging session is used to unwrap the encrypted key into a temporary file. The second secure messaging session is used to save the triple DES key into a key file. This follows after the attribute file has been created in between these 2 secure messaging sessions. The order of the components are listed below:

1. Open secure messaging (B.10 - command 92, 93, 94)
2. Select a file [the file location is encrypted and therefore unknown] (B.10 - command 95)
3. Unwrap key and attributes (B.10 - command 96)
4. Close secure messaging (B.10 - command 97)
5. Select key attribute file (B.10 - command 119)
6. Add key attributes to file (B.10 - command 120, 121)
7. Open secure messaging (B.10 - command 122, 123, 124)
8. Select the key file directory (B.10 - command 125)
9. Create key file (B.10 - command 126)
10. Close secure messaging (B.10 - command 127)

In a scenario where we are not already in the knowledge of the key value, the secure messaging sessions are the main vulnerability that we will aim to try and exploit. While the key is not exported at the APDU layer in plain text. Successfully reverse engineering the secure messaging protocol, would allow an attacker to exploit it. This can be achieved by injecting our own session keys and decrypt the key value. This was discussed in analysis of the generateKey function.

Furthermore, an attacker can also modify the attributes as the key is saved within the card for the first time. But as discussed in the generatKey function analysis, this attack would make the user aware of the fact that the key has been exported and the attributes have been changed. This cannot be deemed as serious a vulnerability compared to one that would expose the key in plain text at the APDU layer without the user's knowledge. These possible vulnerabilities are discussed further in chapter 7.

6.11 C_wrap

As discussed in section 4.3, literature showed the wrap/decrypt attack is carried out at the API level whereby one key can be given wrapping and decryption functionalities within its attributes. This allows for any key (regardless of the attributes) to be wrapped, exported out of the card, and then decrypted by the same key. This would reveal its key value in plain text. This vulnerability bypassed all controls that the attributes of a key are expected to provide. Due to this vulnerability discovery, it is reasonable to assume that 'Athena smartcard solutions' decided to remove the ability to wrap keys completely in an attempt to prevent this type of attack. This reasoning is based on the fact that requesting any key to wrap another key at the API level causes an error: *CKR_Function_Not_Supported.*

From the analysis of table 4.4 (see section 4.5.2), we find a correlation between the P1 and P2 parameters for the APDU commands for encryption, decryption and unwrapping. These values are tabulated below:

P1	P2	Command
82	05	encrypt
80	05	decrypt
80	0A	unwrap
?	?	wrap

The CLA and INS bytes are identical for all of the above commands.

Given the fact that for encrypt and decrypt, P2 remains the same and only P1 alters from '82' to '80', we infer that if the wrap command still exists at the APDU layer, then the values for P1 and P2 would be **82** and **0A** respectively. Therefore, an example full command would be: **00 2a 82 0a L Data 00**. Were L is the length of the data, and the data is the key to be wrapped.

We ran a simple test to see if this command operates on the card:

1. Wrap key (string '12345678') (B.11 - command 1)

The result of the test shows an error at the APDU level. The error value was '6A 80', meaning the parameters within the data field are incorrect. This suggests that the command does exist and most likely corresponds to the wrap function.

The data field of unwrap is encrypted, as it is used within secure messaging. However if this was reversed engineered and we are able to understand the data field of the unwrap command, it will be highly likely that we would be able to create the correct data field for the wrap command. This would allow us to bypass all attribute controls of keys at the API level, and implement the wrap/decrypt attack at the APDU level. This in turn would allow all keys on the card to be exported out of the card in plain text. This would be a serious violation of the PKCS #11 standard.

Chapter 7

New Attack’s At the APDU Level

This chapter explains the motivations, implementation and discusses the results of the attacks we conduct as the APDU level. For clarity we extract the vulnerabilities we suggest for investigation in our analysis of each PKCS #11 function in chapter 6. We report the attacks we decide to investigate and the reasoning for selecting them. The two attacks we do investigate are the reverse engineering of the PIN authentication protocol and the secure messaging protocol.

7.1 Motivations

Vulnerability	Functions	To be Investigated Further
Modifying attributes (keys already generated)	C_findObjects	χ
Modifying attributes (as keys are being generated)	C_generateKey, C_generateKeyPair	χ
Opening key file memory locations	C_encrypt, C_decrypt, C_generateKeyPair, C_destroyObject	χ
Reverse engineering PIN authentication protocol	C_Login	✓
Reverse engineering secure messaging protocol	C_generateKey, C_unwrap	✓
Wrap/Decrypt attack at the APDU layer	-	future work
Modifying the retry counter	C_login	future work

7.1.1 Vulnerabilities not investigated

The *initialization* function does not reveal any vulnerabilities that cane be exploited and is there for the API to correctly operate.

We have highlighted in the *Login* function analysis that it might be possible to use either 'REALLOCATE BINARY' or 'DELETE FILE' and 'CREATE FILE' in order to modify the retry counter. The aim of this would be to allow an unlimited number of attempts at guessing the user's PIN. We argue that is it not advisable to focus on this attack as it would not be particularly useful without the knowledge of the challenge-response algorithm that is used to verify the user's PIN. Thus, we decided to focus on the reverse engineering of the PIN authentication protocol first, and left this attack for future work.

The vulnerability discussed in the *FindObject* function analysis presents the possibility to locate the attribute files at the APDU level, and modify them using 'REALLOCATE BINARY'. The two attributes that have the most significance would be CKA_extractable and CKA_sensitive. Previous work [?] has analysed this vulnerability and the findings did not reveal any significant security concerns. This was due to the fact that once the attributes had been set as part of the key generation process, the attribute file and the key file are separated and thus prevent the key from being exposed in plain text. Since this attack had already been investigated we did not feel that it would be appropriate to investigate it any further at this stage.

GenerateKey and *GenerateKeyPair* function analysis both suggest a vulnerability whereby attributes can be modified as the attribute file is being created for the first time. This leaves the possibility of the key value being exposed at the API layer. This effectively could result in the card being instructed to generate the key with the attributes the attacker sets. Setting the key attributes to have CKA_sensitive set to false and CKA_extractable set to true, will allow for key value exposure at the API level. This vulnerability would always have to be exploited upon key generation and would be visible to the user of the smartcard. Although this might be a viable attack to investigate, we have decided that it would be better to look into methods that would allow all keys to be exported in plain text regardless of whether the key was already generated and saved on the card without informing the user.

Encrypt, Decrypt, GenerateKeyPair and *DestroyObject* functions export in plain text the directory of the key file. Following our discussion in chapter 6, (destroyObject analysis, section 6.6) we hypothesise that this might have provided the ability to open the file and read the key value in plain text. This hypothesis turned out not to be valid, and instead the we could only view the file control parameters, which stored data wrapped in an ASN.1 BER encoding, that had pointers to locations within the smartcard's memory where the key values are stored securely within internal EF's (see section 3.2.3, chapter 3).

We find that *Unwrap* function does not reveal any vulnerabilities. Instead we would like to learn exactly how the data field is built, but this cannot be inferred from traces as the command is only ever used within secure messaging. Therefore, the data field is encrypted. The rational for focusing on understanding of how the command works is related to possibility of implementing the wrap/decrypt attack at the APDU layer. This is explained in the next section.

7.1.2 Vulnerabilities to investigate

From studying the traces for the ***login*** function it was apparent that a challenge-response algorithm has been implemented. This helps to provide a secure method for transporting the PIN for verification by the smartcard without ever revealing its value in plain text (which would be in breach of the PKCS #11 standard). This is an improvement upon implementations from other card manufacturers whereby they did just send the PIN in plain text over the APDU layer for it to be verified. This leaves the possibility of an an attacker simply intercept the communication and extract the user's PIN number. However, the challenge-response algorithm that 'Athena smart-card solutions' has implemented on this card still leaves the potential for an attacker to reverse engineer the protocol, and then brute force the PIN until the same response is calculated to be sent to the card. Doing this would give the attacker the knowledge of the users PIN and also remove the first dependency on the use of the API by an attacker. This is therefore the first attack that we investigate.

Furthermore the use of secure messaging to encrypt and also provide an integrity check (that the commands have not been altered in transit) in the form a message authentication code being appended to the command was found within two functions. These are ***generateKey*** and ***unwrap***. The ability to successfully reverse engineering the secure messaging protocol would allow us to achieve two objectives. First, within the generateKey function an attacker would be able to implement a man in the middle attack to either alter the key being generated with a value known to the attacker, or simply decrypt the key to find out its value. This would be possible regardless of the attributes set for the key, and therefore would violate the PKCS #11 standard, especially if the attributes for CKA_sensitive and CKA_extractable were set to true and false respectively. As an attacker would be able to know the block cipher keys generated when in fact the only system that should be in possession of said knowledge is the smartcard.

Secondly is the understanding behind how the ***Unwrap*** APDU command operates. As stated in section 6.11, despite the API removing the ability to use the ***Wrap*** functionality (most likely due the wrap/decrypt attack discovery [?]), with a high probability the APDU command still exists as we managed to send what we believe to be the correct command and received back an error stating the data field parameters were incorrect. Therefore reverse engineering the secure messaging protocol would allow us to fully understand the ***Unwrap*** command data field, and possibly then be able to generate the correct command for wrapping a key. If this is possible, then the wrap/decrypt attack could be implemented at the APDU level, bypassing all attribute controls and exporting any/ all keys that are stored within the card. This would be considered a serious vulnerability.

All of the vulnerabilities discussed within this section are to be implemented at the APDU layer and would not provide the user the knowledge that an attack is taking place. This is the main reasoning behind selecting these vulnerabilities for further investigation, and the fact they are far more serious compared to the vulnerabilities discussed earlier, in terms of violations of the PKCS #11 API. Furthermore the attacks suggested for further investigation also remove the dependencies on the use of the API, which will allow an attacker to send commands at the APDU layer independently. This gives an attacker a higher capability.

7.2 Reverse Engineering PIN Authentication Protocol

The first attack we investigate is the reverse engineering of the PIN authentication protocol. If this is successful, an attacker will be able to brute force and/ or use a dictionary attack to calculate the users PIN. The attack shall also remove one dependency on the use of the API. At this stage the API is required to login into the smartcard as we are unaware of how the challenge-response algorithm is implemented. Figure 7.1 shows the procedure for logging into the smartcard at the APDU level.

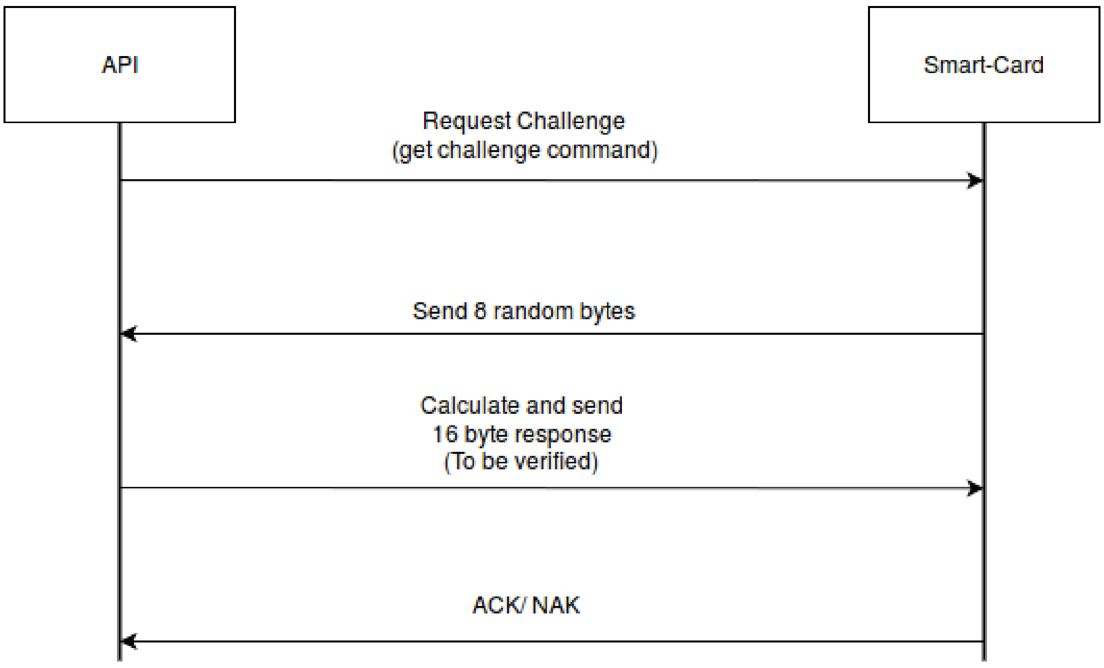


Figure 7.1: Challenge Response Procedures

The API requests (via a ’GET CHALLENGE’ command) the smartcard to generate and send back 8 random bytes. Based upon the knowledge of the challenge (8 random bytes) and the user’s PIN, the API then calculates the 16 byte response. The 16 byte response is sent to the smartcard (via a ’VERIFY’ command), which returns one of two possible outcomes:

1. ’90 00’ $\rightarrow$ Verification of user’s PIN succeeded
2. ’63 CX’ $\rightarrow$ Verification of user’s PIN failed. (X = number of attempts left before the card is blocked)

The following sections provide explanations of the searches that we have conducted in trying to reverse-engineer the protocol shown above. To give a good understanding of how challenging this part of the project is, we will explain the combinations of different possibilities that we have tested, and the reasoning behind each of them. These are divided into different ’searches’, with the findings of each ’search’ being incorporated into the next one.

To facilitate these explanations, we first introduce here 3 sub-functionalities that have been used throughout the majority of our searches. Table 7.1 lists all of the hash functions (see section 2.1 of chapter 2) that are used. The table also provides the output length in bits & bytes. These hash functions are all supported by openSSL and the smartcard. Table 7.2 provides the names of the bitwise logical operations that are used to ’join’ two bytes together. Table 7.3 provides the description of truncation methods that are used to reduce the output size of a search down to 16 bytes to match the response provided by the API.

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 7.1, 7,2 and 7.3 have been iterated over and preformed on. For example, ***truncate(hash('example'))***, means the string, 'example', 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)
MD5	128	16
SHA1	160	20
SHA256	256	32
SHA384	384	48
SHA512	512	64

Table 7.1: Hash Functions *(supported by the card)*

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 conducting any searches, we need to carry out the first task which is 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 reports 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 original experiments, we have assumed that the PIN number was only numerical characters, had 4-8 digits and if the PIN was less than 8 characters, it was padded using a special character to form 8 bytes. Our tests show that none of these assumptions hold. Thus our preliminary experiments are flawed and do not provide any meaningful results that are worth reporting. It is for this reason that we have not included a description of these experiments. In addition many of these experiments are in fact similar to those ones that are to be explained below, but with different assumptions of the PIN. These experiments also use 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 the need to testing for different padding schemes and characters. This provides justifications as to why in the following experiments the PIN had been set to ’0000000000000000’ (16 zeros, no need for padding). We found that 16 characters was the upper limit to what a user can set the PIN to be by changing the PIN and incrementing its length until the API returned an error.

7.2.1 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 a given challenge and PIN there is only one corresponding response that API will calculate

Search 1.1 - Hash functions

In the early stage of this project we were of the opinion that there is a high chance that the 16 byte response is generated by hashing a combination of the PIN and the 8 byte challenge. This is partly due to common practices that are used by the industry whereby users passwords are often hashed (see section 2.1), and in most cases salted (A salt is a random string appended to a message before hashing), before storing them in databases.

This practice is more secure than storing plain text passwords. If an attacker were to gain access to the back end databases where passwords are stored, these passwords would not be available to in plain text. For authenticating a user on a website, the password is just hashed (and salted, if a salt is used), and then compared against to the stored value (correct password hashed) within the database.

The fact that from multiple traces the 16 byte response seemed to be uniformly random, this provides support to our hypothesis that the PIN is hashed to provide the 16 byte response. Thus, our first completed search has focused on the use of different hash functions. Below we list the methods we have tested in our experiments to generate 16 bytes, given X and the PIN.

[We denote || as the concatenation 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 table 7.1. The output 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.]

This experiment did not result in finding a match to the response generated by the API. Thus, we move onto our search 1.2.

Search 1.2 - PBKDF2

Following the failure of search 1, but still assuming there is a high chance of a hash function being used, due to the common practices discussed under search 1, and the characteristics known so far of the 16 bytes as calculated by the API, we have decided to test the password-based key derivation function.

PBKDF2 has been created as part of PKCS #5 by RSA laboratories [?] and more recently is being 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. The evidence reported in the literature [?] in 2008 (check date) has shown that the standard for the number of iterative rounds used to be 10,000. The literature also suggests the use of as many rounds as is computationally feasible by any device. Due to the processing power of a smartcard, we assume that the number of iterative rounds does not exceed 100,000 rounds.

Search 2 generates experiments that ran through $1 \rightarrow 100{,}000$ rounds of PBKDF2. As the default of PBKDF2 is to truncate the output by taking the first *i* number of bytes, therefore the only truncation method used is 'first_16' (see table 7.3).

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

- 16 byte response = PBKDF2(hash_function, PIN, X, number_of_rounds)

The above experiment has generated 100,000 different tests per hash function. With 5 hash functions, this hash resulted in half a million tests being run. Due to the substantial computational power required for this search, we have decided to parallelize the search based on the hash function, and run them on separate cores of a server. Despite the significant improvement of the efficiency of this search, the search still undertook 2 weeks to conduct.

Unfortunately, this did not result in a match between the 16 byte responses calculated and Y (the API's response). Therefore, we move onto the search 2.1.

7.2.2 Authentication Protocol Search 2.0 (One Time Passwords)

Search 2.1 - OCRA: OATH Challenge-Response Algorithm

With no success in deducing the challenge-response algorithm, we have decided to examine more complex standards that already exist and are used in different sectors of the computing industry for challenge response protocols, rather than password storage techniques. The international engineering task force (IETF) released a paper in 2011 [?]. This paper provides details of a one-way challenge response algorithm which fitted the characteristics of the authentication that takes place between the API and the smartcard. We have extracted the protocol and provided it below:

```
 CLIENT                                   SERVER
(PROVER)                                 (VERIFIER)
   |                                        |
   |   Verifier sends challenge to prover   |
   |   Challenge = Q                        |
   |<---------------------------------------|
   |                                        |
   |   Prover Computes Response             |
   |   R = OCRA(K, [C] | Q | [P | S | T])   |
   |   Prover sends Response = R            |
   |--------------------------------------->|
   |                                        |
   |  Verifier Validates Response           |
   |  If Response is valid, Server sends OK |
   |  If Response is not,  Server sends NOK |
   |<---------------------------------------|
   |                                        |
```

```
C - Counter, optional.

Q - Challenge question, mandatory, supplied by the verifier.

P - Hashed version of PIN/password, optional.

S - Session information, optional.

T - Timestamp, optional.

(source) [?]
```

The paper also explains that 'OCRA' (in the instance of this protcol) stands for a generic variable to state which form of HOTP is to be used. HOTP (see section 2.5.1, chapter 2) can take on different formula's due to using different hash functions. The generic formula is:

$$HOTP(K,C) = Truncate(HMAC(K,C)) \quad \&\& \quad 0x7FFFFFFF$$

where K and C stands for the password (secret key) and is a counter (the challenge) respectively.

We ran experiments using the 5 hash functions listed in table 7.1, and even inter-changed K and C using each others values. This did not result in finding a matching response to the one the API has calculated.

Search 2.2 - TOTP

Following search 2.1, we thought it would be appropriate to rule out the possible use of *time based one time passwords* (see section 2.5.2). TOTP algorithms are rarely used within smartcard's due to the complexity of syncing the clocks between the computer and the smartcards processors, and then also accounting for any lag. Despite the low chance of TOTP being used it was worth testing.

This was completed by halting communication between the smartcard and API using the man-in-the-middle tool (see section 5.3). We halted the communication for upto 2 hours. Our primary objective is to look for a failed verification, despite having the correct PIN. The failure should have been caused by the delay in the response if TOTP is used. This was not found to be the case and therefore we ruled out as a possibility that TOTP is used.

7.2.3 Authentication Protocol Search 3.0 (Triple DES Encryption)

With none of the above searches yielding a match to the API calculated response, we decided to focus on verifying the 4th assumption (given the PIN and challenge the API only calculates 1 response) made at the beginning of section 7.2.1. Our preference is to have carried out this test at an earlier stage. This was not possible as it took longer to make the code for the man in the middle attack (which is required for this task) operational.

To verify this assumption we have used the man in the middle tool to inject our own challenge (that the card normally sends) and set it to '00 00 00 00 00 00 00 00'. Completing this several times, we have found that this assumption does not hold. Following these multiple attempts, we have found different responses being calculated by the API, despite the fact that the PIN and the challenge remain the same. We have also found that logins with quick successions of one an other had the same value for the response when the same PIN and challenge were present, and in some cases only the last 8 bytes have changed with the first 8 bytes remaining the same.

Because of these findings we have decided to run multiple experiments to find characteristics of the 16 bytes response *(which we now believe to be split into two 8 byte parts)*. We have done this by logging into the card twice, and altering the three parameters *(one at a time)* that we have found to cause the response to alter. The parameters we alter are ***time, PIN*** and the ***challenge***. The results of the experiments are in table 7.5.

[For further clarification, we have placed the actual traces of these experiments within ***appendix A****. We use variable names for the two 8 byte sections of the 16 byte response. These are denoted below as A and B]*

16 byte response	1st 8 bytes	2nd 8 bytes
Y =	A	B

Time	PIN	Challenge	A	B	Appendix
Different	-	-	χ	χ	A.1.1
Same	Same	Same	✓	✓	A.1.2
Same	**Same**	**Different**	✓	χ	**A.1.3**
Same	Different	Same	χ	χ	A.1.4
Same	Different	Different	χ	χ	A.1.5

Table 7.5: Multiple logins with different parameters

Table 7.5 shows that changing the time by 1 second between the two seperate logins, or changing the PIN causes the whole 16 bytes to alter. If both login attempts are within the same second, and the same PIN and challenge are given to the API then the exact same 16 byte response is calculated. The most interesting characteristic we have found within these experiments arises when the logins occur within the same second, with the same PIN but with different challenges. This causes the first 8 bytes of the responses of the two logins to be identical, but the second 8 bytes to differ.

These characteristics provide evidence to support the fact that the 16 byte response is in-fact split into two 8 byte sections. The first 8 bytes (denoted A) appears to be a nonce (random number) generated by the API. This nonce changes each second, and thus we assume a random number generator is used within the API and is seeded with time. This also shows that the second 8 bytes (denoted B) is most likely to be a function that utilises the challenge. As a different challenge causes it to change. Based on these facts we hypothesise the following:

1. The first 8 bytes is a random number N_A = Nonce API, which is seeded with time (in seconds)
2. The second 8 bytes is a another number that verifies the knowledge of both the challenge and N_A. We denote this as V_{AC}
3. A block cipher (see section 2.3) is used to encrypt both of these numbers, to send over to the card for PIN verification
4. A function of the PIN, is the password to the block cipher
5. The PIN is still of maximum length 16 bytes
6. The PIN is still in ASCII format

To sum up we assume at this stage the challenge-response algorithm uses a function of the PIN to be the password to a block cipher which encrypts a message in this format:

$$block\_cipher\_encrypt(N_A||V_{AC})$$

The block ciphers that are supported by the card are listed below:

Block Cipher	Key Size	Block Size
AES	16	16
DES	8	8
Triple DES *(key option 2)*	**16**	**8**
Triple DES *(key option 3)*	24	8

The use of the block cipher AES is ruled out as a possibility. As if it were used the characteristic whereby the second 8 bytes (denoted B) alters between two separate logins within the same second, using the same PIN but with different challenges, would not be present. In this scenario the use of AES would alter the entire 16 bytes due to the block cipher operating on a block size of 16 bytes. This leaves only DES and triple DES as a viable option.

We have been able to eliminate the possibility of the standard DES being used due to its key size. It would not be logical to use a function of the password and then truncate it down to 8 bytes from 16 bytes. This would result in the security of the login to drop from 256 bits to 128 bits for no apparent reason, as other options are available.

Therefore, this leaves us with triple DES key options 2 and 3. Given the fact that the PIN for the user is of maximum length 16 bytes, we have decided to focus on key option 2 which has 16 byte password. It is a possible that the password is hashed for example and 24 bytes are used with key option 3, but we have decided to first test key option 2 before testing 3 in the case that 2 fails.

Based on the assumption that the second 8 bytes is a verification (possible join, or hashing) of both the cards challenge and N_A, then we cannot simply just decrypt the 16 byte response and analyse it. As both numbers (A & B) could be completely unknown. Therefore, we would have no method of verifying whether we have the correct protocol. To mitigate this we decided to obtain 2 different responses, within the same second, with the same challenge, but using different PINs. In the case the hypothesised protocol is used and the password to the block cipher is in-fact a function of the PIN, then the correct decryption of both responses with different passwords should be identical. This is the approach we have followed to verify our hypothesis.

Search 3.1 - Triple DES *(Key option 2)*-ECB

Table 7.6 reports the data for two logins that have occurred within the same second, with the same challenge (that we injected), but using two different PINs. We then generate 6 different possible passwords for the encryption using the PIN. The first is simply the PIN in ascii format at 16 bytes (all zeros). The following 5 are the PIN hashed with the hash functions that are reported in table 7.1, and truncated using only 'first_16' truncation method in table 7.3. Using the generated passwords we decrypt the 16 byte responses using triple DES key option 2, and the results are reported in table 7.7.

Response_1	93 57 F7 53 42 A3 27 3D E6 B8 7E A6 81 98 5B 1C
PIN_1	0000000000000000
Response_2	90 6A 74 D1 BD 2C 75 2E 52 bA 17 87 E3 70 51 EF
PIN_2	1111111111111111
$\text{Challenge}_{1,2}$	00 00 00 00 00 00 00 01

Table 7.6: Experimental setup

Pin Method	Response	Decryption using Triple DES-ECB
PIN_1	1	A4E8604FFFEABBC7 A85FDCA6E1D8187F
PIN_2	2	23479D5F8CA2EAB7 AE0EB9D09C987484
MD5(PIN_1)	1	C4B103E761F3688A 45681FD6B9F7137D
MD5(PIN_2)	2	404F5882AFE4600D E8E8C4F918E0C55B
SHA1(PIN_1)	1	**FE5DBEA8ECAB2F86** 9357F75342A3273C
SHA1(PIN_2)	2	**FE5DBEA8ECAB2F86** 906A74D1BD2C752F
SHA256(PIN_1)	1	5F6D92B899EF4557 A1B797A4F65D5B30
SHA256(PIN_2)	2	74CED585EABBD62E B847856F6A3AEBC7
SHA384(PIN_1)	1	DFF3A6A88508D7CF 802CF22A15E4051C
SHA384(PIN_2)	2	A75AB9FEE2A2B445 54618384E33CA723
SHA512(PIN_1)	1	473EF6CC4BA45EF7 EA740F801A595B4A
SHA512(PIN_2)	2	1C202A831098546D 8EF674B4A0B0D7AD

Table 7.7: *Decryption Results*

The results are interesting. While we do not find an exact match of the entire responses when decrypting both of them, we do find a match for the first 8 bytes. This suggests that we have found the correct protocol and formatting of the PIN, but the wrong decryption methodology. A change in the second 8 bytes suggests that cipher block chaining is used. Next we move onto search 3.2.

Search 3.2 - Triple DES *(Key option 2)*-CBC

Triple DES block cipher using cipher block chaining (CBC) requires an initialization vector (IV) to be used (see section 2.3, chapter 2). The IV is 8 bytes in length. It is Common practice is to send this in plain text along with the encryption of a message. It is evident from our results that this is not the case. The only other possibility is that the card and the API already have agreed upon a value to use, and always use this same value. The most common practice is to use an IV = 00 00 00 00 00 00 00 00. Despite it being a an insecure practice, this still does occur. Hence, in the following experiment we only test the pin method for SHA1(PIN$_{1,2}$) and decrypt the corresponding responses using triple DES-CBC with an IV of zeros.

Pin Method	Response	Decryption using Triple DES-ECB
SHA1(PIN)	1	FE5DBEA8ECAB2F86 0000000000000001
SHA1(PIN)	2	FE5DBEA8ECAB2F86 0000000000000001

Table 7.8: Decryption Results

Table 7.8 reports the results of both decryptions. The results show the second 8 bytes, which we thought would have been a verification number (join or hash) (showing knowledge of both nonces), is in fact just the smartcard's challenge. The is better than what we had expected in our hypothesis. This almost completes our attack. As now we can produce a 16 byte response, given the cards challenge, the PIN and a nonce to place at the start of the message to encrypt.

Now our main issue is that we are not sure if we can just place any 8 byte nonce at the start of the message to be encrypted. This is because the API might be expecting a counter value, and might refuse invalid values. Hence we conduct the final stage of this attack to test if we can place any value for N$_A$, and hence will have fully reversed engineered the PIN authentication protocol.

To sum up so far this is the protocol we believe to be in place:

- Request and get an 8 byte challenge from the card, denoted $\mathbf{N}_c$
- Password for encryption = Truncate_First_16(SHA1 (PIN)))
- Block cipher used = Triple-DES-CBC
- Initialization Vector = 00 00 00 00 00 00 00 00
- Calculate the response as the following:

$$Encrypt(N_a||N_c)$$

- were $\mathbf{N}_a$ is a nonce determined by the API.

We set N$_a$ to equal 00 00 00 00 00 00 00 00 00 00, calculate what we believe to be a valid response and send it the the API. This experiment is reported in appendix A.2. The result of the experiment is an APDU response of '90 00', successful login, with our injected response.

Thus concluding this attack.

7.3 Reverse Engineering Secure Messaging

The first objective of our second attack is to investigate the reverse engineering of the secure messaging protocol used by the card. As discussed in section 3.2.4 (chapter 3), and reported in 'Athena smartcard solutions' security policies [?] for several different cards the vendor produces, the secure messaging initialization is used to derive 2 session keys. S_{Enc}, S_{Mac} are used to encrypt and compute a message authentication code (MAC) of the messages within a secure messaging session respectively. This provides confidentiality and data integrity of the messages. The decision concerning how to derive these keys is left to the discretion the card's vendor. This makes this attack inherently difficult as there are several methods that could be used.

As discussed in chapter 6, the secure messaging sessions are used in the ***GenerateKey*** and ***Unwrap*** functions. This is also reported in ***CreateObject*** function by (name) 2015 [?]. In all cases, it is used to provide confidentiality of the CKA_value of block cipher keys. The ability to successfully reverse engineer the secure messaging protocol would allow an attacker to use a man in the middle attack to derive two sets of session keys (one with the API and one with the smartcard). This would allow the attacker to decrypt the command data field and find value of the generated block cipher keys. This would violate the PKCS #11 standard if the keys were set to be sensitive and un-extractable. This attack is far more serious than the previously analysed attacks, due to the high chance the user would be completely unaware of the attack.

Our second objective is to learn how the ***unwrap*** command data field is built and then build the correct command for ***wrap*** at the APDU level. This might be completed by decrypting the data field of the unwrap command when used via the API. With this knowledge it should then be possible to implement the wrap/decrypt attack [?] at the APDU level, and will therefore allow the exporting of all keys including RSA private keys which are generated on the card which are assumed to be securely stored.

We use the API to call the function ***GenerateKey***, as we know that secure messaging is used within this function. This permits us to have a good overview about how the secure messaging protocol works. Below is a diagram that visually demonstrates what exactly occurs at the APDU level. The actual APDU commands are reported in appendix A.3.1

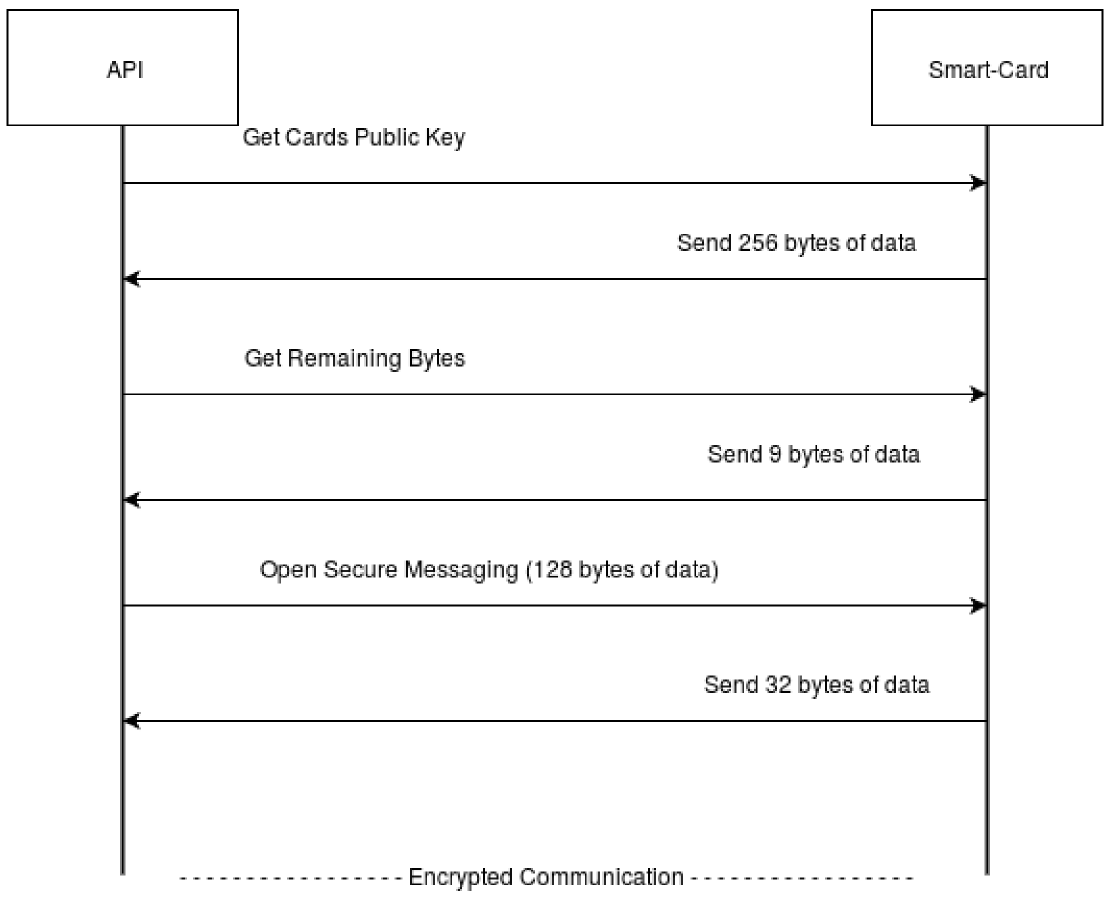


Figure 7.2: Secure Messaging Initialization

As shown in figure 7.2 the API first requests the public key of the card. In total (split between two messages due to the size exceeding 256 bytes), the smartcard sends to the API 265 bytes worth of data (its public key). The API then proceeds to send 131 bytes to the smartcard. Finally, the smartcard sends 32 bytes of data to the API, which completes the initialization of secure messaging. At this stage both the smartcard and the API are aware of two sets of block cipher keys S_{Enc} and S_{Mac} which are used to encrypt all of the following communication. From studying multiple traces we have found that these 131 bytes and 32 bytes of data, sent by the API and smartcard respectively, change every time a new secure messaging session is opened. The 265 bytes of data sent first are static (remain the same).

7.3.1 Raw byte analysis of the data fields

The first step we have decided to take is to analyse the raw bytes within the data fields. From studying the traces of secure messaging sessions, we found that the 265 bytes that where sent from the smartcard to the API (smartcards public key) had an encoding. From reviewing ISO/IEC 7816 section 13 we have found that the 265 bytes are in fact 3 distinctive pieces of data of the following values.

Send 265 bytes of data (cards public key) (A.3.1 - commands 24, 25)
[00] (1 byte) 05
[01] (128 byte) F7 B5 15 72 07 22 94 6F C4 08 ...
[02] (128 byte) 3C 52 D2 06 89 28 92 2C AB E6 ...

[These control bytes separating the encoding of the data have been highlighted in bold within the commands in the appendix A.3.1 to show how the data is split]

We did not find any of the other data fields wrapped in an encoding. Thus now we know the following:

Data	Explanation
Cards public key (static key) **257 bytes in total**	(1 byte) = 05 (128 bytes) = F7 B5 15 72 07 22 94 6F C4 08 ... (128 bytes) = 3C 52 D2 06 89 28 92 2C AB E6 ...
API data *public key or encrypted data* (Not static)	(128 bytes) = 08 9F EA A1 DC 8F C3 43 ...
Card random challenge (Not static)	(32 bytes) = 66 56 36 31 16 42 8D 8A ...

Table 7.9: Data organization within secure messaging initialization process

Using the information in table 7.9 we now test protocols that would allow session keys to be derived, and match the amount of data being sent across from the API to the card and vice versa.

7.3.2 Protocol search

There are three main protocols that would support session keys being derived that would remain a secret between the API and the smartcard. These are: RSA encryption, Elliptic Curve Cofactor Diffie Hellman (ECCDH) and Diffie Hellman (DH) (see section 2.2.2, chapter 2).

The security policies for 'Athena smartcard solutions' [?] states that ECCDH is used for some of their smartcards. They also report that the 32 bytes of random data sent by the card is used to XOR with the shared secret derived from ECCDH to create the 2 session keys.

*[The same documentation stated that the shared secret derived from ECCDH was 32 bytes in length, and the session keys were 2 x **Triple DES (key option 2)** 16 byte keys]*

Our project's smartcard supports the key generation of elliptic curve public and private keys for this exact purpose, and for signing and verifying data. Thus, the first protocol we look into is ECCDH.

Elliptic Curve Cofactor Diffie Hellman

The smartcard we analyse only supports curves P-192, and P-521 for EC-CDH (see section 2.2.3, chapter 2). The security policies state that curve P-256 is used. This is the first indication that this smartcard does is not use ECCDH for secure messaging. Having said this, we believe it is still worth investigating in case one of the above curves is used.

First, we calculate the number of bytes that are required in order to setup the global parameters (for ECCDH) that must be sent from the card to the API, and also the number of bytes of each public key. A match between the number of bytes calculated and the number of bytes that are sent across could suggest this protocol is being used for secure messaging initialization. *We use OpenSSL to generate the following different public and private keys, along with the global parameters and outputted them in DER format to find the length of the corresponding data.*

Elliptic Curve Field	P-192	P-256	P521
Global Parameter Length	233	233	233
Public Key Length	48	64	131
Total	**281**	**297**	**364**

Table 7.10: Number of bytes required for ECCDH [Card to API]

As reported in table 7.10 none of the curves for ECCDH have a value of 257 bytes in total that would be sent from the card to the API to begin the protocol. None of the public key sizes match 128 bytes which is what the API returns to the card as well. Thus, we conclude that ECCDH is not used and move on to test the next protocol, RSA encryption.

RSA Public Key Encryption of Block Cipher Keys

If RSA is used then the only method to derive the keys between the API and the smartcard, is for the API to use the public key of the smartcard to encrypt session keys created by the API. The smartcard will then use its private RSA key to decrypt the session keys. Considering that there are 32 bytes sent by the smartcard at the end of the secure messaging initialization process, it is likely that they are XOR'd with the bytes encrypted via RSA, to generate the final session keys to be used.

Next we complete the same analysis as the one for ECCDH and calculate the number of bytes required to find out if they match the corresponding bytes sent via the API and the smartcard. These results are tabulated below:

RSA	1024-bit	2048-bit
Public Modulus length	128 bytes	256 bytes
Public Exponent length	1 byte = 5	1 byte = 5
Encryption length of 32 bytes	128 bytes	256 bytes

Table 7.11: Number of bytes required for RSA

The results reported in table 7.11 reveal that the key option of 2048-bit fits the size of the smartcards public key perfectly *(This is based on the public exponent having a value of 5, see table 7.9).* In contrast, the encryption of the 32 bytes by the API would result in 256 bytes, not 128 bytes which is sent from the API to the smartcard. This result rules out the possibility that RSA-2048 key is used, leaving the other possibility which is two RSA-1024 keys.

The literature suggests that some manufacturers have implemented smartcards with multiple asymmetric keys for secure messaging. Whereby different versions of their middleware (implementation of the PKCS #11 API) will only use one of the keys. Thus the first test we ran was editing the first byte of each of the 128 bytes to find out if one or both of the suspected RSA-1024 keys are used. For each test we run, the secure messaging initialization completes, but then upon the first instance of encrypted communication an APDU error is raised, 'error with secure messaging'. Thus, the results confirm that both 128 byte keys are used.

Assuming two RSA-1024 bit keys are used, this would suggest that the session keys are generated by the API are encrypted twice with the two RSA-1024 bit keys. The output of the first encryption being the input to the second encryption. Both RSA-1024 keys have a public exponent equal to 5.

Despite matching the bytes sent across in secure messaging initialization and the number bytes that would be required when using two RSA-1024 bit keys. The use of two RSA-1024 bit keys is unreasonable, as in any case two RSA-1024 bit keys are no more secure than just using one. It would be more logical to implement RSA-2048.

We run our next test by changing what we believe to be the public exponent in order to find out if RSA keys are used. Due to how messages are encrypted with RSA, setting the public exponent to 1 should reveal the message itself, and setting the public exponent to 0 should return a value of 1. This is verified by the following formula:

$$ciphertext = message^{e} \quad mod \quad n$$

Where e is the public exponent, and n is the public modulus (128 bytes).

	public exponent = 0	public exponent = 1
cipher text *(128 bytes sent from API to card)*	**0**	**1**
Appendix	A.3.3	A.3.2

Table 7.12: Results from altering public exponent

The results reported in table 7.12 are not consistent with our expectations relating to RSA encryption. The value of the cipher text when the public exponent was 0 was 0, and 1 when it was 1. The value of the cipher text should have been 1 and the value of the message respectively. The results do however fit the mathematics behind the Diffie Hellman protocol. This leads onto our final search.

Diffie Hellman - Key Agreement Protocol

As discussed within 2.2.2 (chapter 2) the Diffie Hellman protocol relies on the fact that the global parameters are known by the 2 entities in communication with one and other (API and smartcard). The 2 entities then share their own respective public keys which allows them to derive a shared secret without ever revealing it in plain text. Again, the following is the Diffie Hellman protocol.

Public Parameters
[G must be a primitive root modulo p]
Generator = G
Public modulus = p

Private keys (a number between 1 and p-1) are selected by each individual entity and generate the public keys as follows:

$$public\_key_1 = G^{private\_key_1} \quad mod \quad p$$

$$public\_key_2 = G^{private\_key_2} \quad mod \quad p$$

Then a shared secret **S** can be calculated by each entity with the following formula:

$$S_1 = public\_key_1^{private\_key_2} \quad mod \quad p$$

$$S_2 = public\_key_2^{private\_key_1} \quad mod \quad p$$

$$S = S_1 = S_2$$

Therefore, if the generator were set to be **0**, then the public key that is calculated by the API would be 0 as well. In the case the generator is set to **1**, the API would calculate its public key as 1. Therefore due to the mathematical properties of the Diffie Hellman protocol and the results we report in table 7.12, we conclude that the secure messaging initialization process is the Diffie Hellman protocol. Whereby the data sent during the initialization of secure messaging is as reported in table 7.13.

Data	Explanation	DH Parameters
Cards public key (static key)	(1 byte) = 05 (128 bytes) = (128 bytes) =	Generator public key or modulo public key or modulo
API data *public key or encrypted data* (Not static)	(128 bytes) =	API's public key
Card random challenge (Not static)	(32 bytes) =	Used for session key derivation (not part of DH)

Table 7.13: Data organization within secure messaging initialization process

We assume that the 32 bytes sent at the end of the secure messaging initialization is to be similar to the ones reported under 'Athena Security Policies' that used ECCDH for secure messaging initialization [?]. They will be used to generate the final session keys S_{Enc} and S_{Mac} (this is discussed in more detail in section 7.3.4). The next process is to implement a man in the middle attack for the Diffie Hellman protocol so that an attacker is also in the knowledge of the shared secret.

7.3.3 Man in the middle attack - Diffie Hellman Key Agreement Protocol

A normal man in the middle attack for Diffie Hellman would require two Diffie Hellman key agreements to occur. One between the API and the attacker, and one between the attack and the smartcard. From the results reported in 7.12 we have found that we can force the API's public key to be 0, by setting the generator to be 0. We can also force the shared secret to be zero, if we alter the cards public key to be 0 as well. This leads to the card, the API and the attacker to be in the knowledge of the shared secret which is zero.

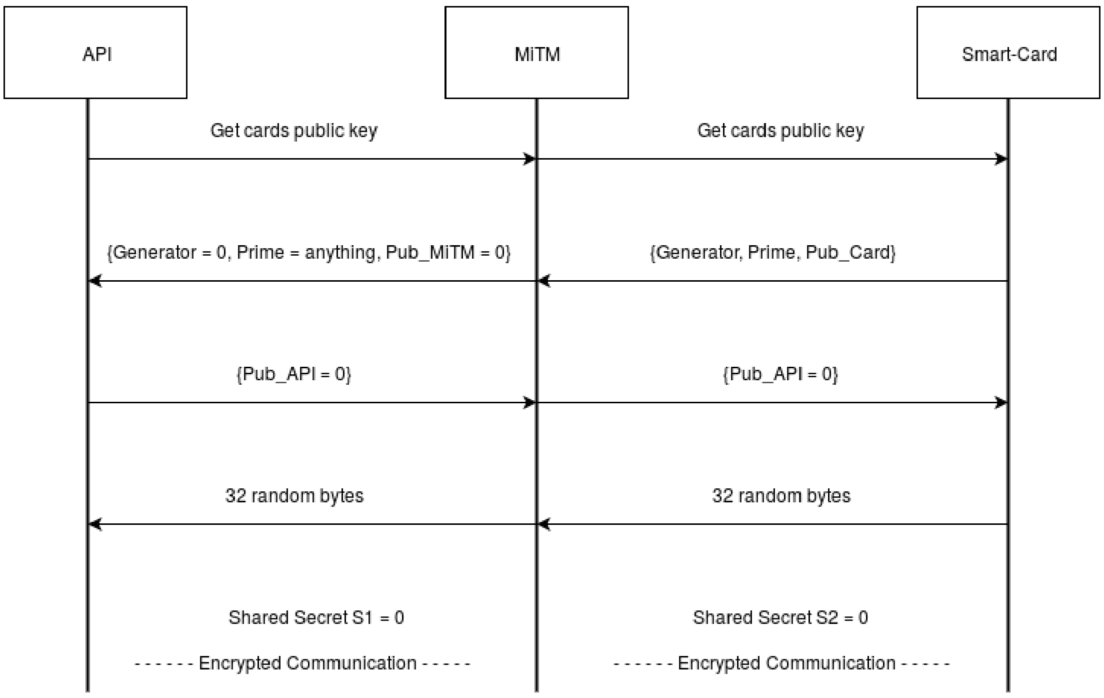

API
MiTM
Smart-Card
Get cards public key
Get cards public key
{Generator = 0, Prime = anything, Pub_MiTM = 0}
{Generator, Prime, Pub_Card}
{Pub_API = 0}
{Pub_API = 0}
32 random bytes
32 random bytes
Shared Secret S1 = 0
Shared Secret S2 = 0
- - - - - - Encrypted Communication - - - - -
- - - - - - Encrypted Communication - - - - -

Figure 7.3: Diffie Hellman - Man in The Middle Attack

This is verified by the mathematics behind the Diffie Hellman protocol.

	API	Smartcard
Generator	0	5
Modulo	p	p
Public Key	$0 = 0^{private\_api} \quad mod \quad p$	$x_{actual} = 5^{private\_card} \quad mod \quad p$ $x_{injected} = 0$
Shared Secret $pub_1^{pub_2} \quad mod \quad p$	$x_{injected}^{private\_api} \quad mod \quad p$ $\mathbf{0} = 0^{private\_api} \quad mod \quad p$	$public\_key_{API}^{private\_card} \quad mod \quad p$ $\mathbf{0} = 0^{private\_card} \quad mod \quad p$

Since we did not know which of the 128 bytes is the modulus, and which is the public key of the smartcard, we complete two further tests setting each of the 128 bytes and the generator to be zero. This allows us to test our our hypothesis described in figure 7.3. The results of these two tests are reported in appendix **A.3.4** and **A.3.5**.

The first test, sets the first 128 bytes and the generator to be 0 (see table 7.13). This caused the API to halt with no further communication between the API and smartcard possible. We can assume that the first 128 bytes is the public modulus. This assumption comes from the fact that setting the public modulus to be 0 would cause OpenSSL to try and create a public key with what is an invalid modulus. This should cause OpenSSL to crash or behave in an undefined manner.

The second test, sets the second 128 bytes and the generator to be 0. Following the results of test 1 we assume that the second 128 bytes is the public key of the smartcard. We therefore expect the shared secret to be set to 0 for both entities (smartcard and API) as described in figure 7.3.

If the encrypted communication after secure messaging initialization does not fail with an APDU error, we will know for certain that the Diffie Hellman protocol is used and the meaning of the data sent from the smartcard and API. The results of our test are as expected. Thus, we find that the Diffie Hellman protocol is used and we also find how the data in the secure messaging initialization is organized. The latter finding is reported in table 7.14.

Data	Explanation	DH Parameters
Cards public key (static key)	(1 byte) = 05 (128 bytes) = (128 bytes) =	Generator **Public Modulo** **Public key**
API data *public key or encrypted data* (Not static)	(128 bytes) =	API's public key
Card random challenge (Not static)	(32 bytes) =	Used for session key derivation (not part of DH)

Table 7.14: Data organization within secure messaging initialization process

7.3.4 Final steps for reverse engineering the secure messaging protocol

With the shared secret from the Diffie Hellman protocol now set to 0 (because the attack in figure 7.3 is carried out), the final steps to reverse engineering the secure messaging protocol are:

1. Find out which parts of the 32 bytes sent from the smartcard to the API are for the S_{Enc} and S_{Mac} session keys.

2. Find the key derivation function used for producing both keys with the knowledge of the shared secret and the 32 byte challenge from the smartcard.

3. Find the block cipher used within the secure messaging sessions

4. Decrypt the secure messaging communication

Experiment 1

The first experiment we run concerns the 32 bytes that are sent from the smartcard to the API at the end of the secure messaging initialization (first step listed above). This should enable us to find out sections of the 32 bytes that are used in generating S_{Enc} and S_{Mac}. We select the 'GET CHALLENGE' command from the first secure messaging session used within the generateKey function, reported in section 6.4. The 'GET CHALLENGE' command that is used in secure messaging does not have a data field (APDU command case 2, figure 3.2, chapter 3). This means that the S_{Enc} key derived from secure messaging initialization is not used. Thus, this allows us to modify parts of the 32 byte challenge to eliminate the bytes that are used for S_{Enc} key derivation. This leaves us with the knowledge of which bytes out of the 32 bytes are used for deriving S_{Enc} and S_{Mac}.

The experiment consists of two tests. Both tests implement the man in the middle attack described in figure 7.3 to set the shared secret value to 0. Test 1 modifies the first 16 bytes of the 32 bytes (see table 7.14) and sets them to a value of 0. Test 2 does the same for the second set of 16 bytes. If the 'GET CHALLENGE' command under secure messaging is successful (i.e. no APDU error), when we modify one of the sets of 16 bytes we are able to find the bytes used to derive S_{Mac}.

	'GET CHALLENGE' Response	Secure Messageing Succeeded	Appendix
Test 1	(24 bytes) + '90 00'	✓	A.3.6
Test 2	'69 88'	χ	A.3.7

Table 7.15: Experiment to find out how the 32 bytes from the card are utilised

The results reported within table 7.15 provide evidence that the second 16 bytes of the 32 bytes sent from the smartcard are used in the derivation of S_{Mac}. We therefore make the logical assumption that the first 16 bytes are used for the derivation of S_{Enc}.

Experiment 2 (Final Experiment)

The second experiment concerns finding the key derivation function and the block cipher. The key derivation function is used for producing both S_{Enc} and S_{Mac}. The block cipher is used for encrypting commands in secure messaging (steps 2 and 3 listed above). This experiment is more challenging as in order to be able to test whether a certain key derivation function is the one used for generating the S_{Enc} and S_{Mac} secure messaging session keys, we need to do the following:

1. Obtain the knowledge of the encrpyted and unencrypted command
2. Obtain the knowledge of how to build the encrypted command

3. Generate session keys S_{Enc}, S_{Mac} using the shared secret (0), the 32 bytes from the card and a key derviation function **KDF**.

4. Build the encrypted command with keys we generate, and different block ciphers.

5. Compare the encrypted command generate by the API to the command we generate. If they match we have found the KDF, and block cipher.

The first step is simple, since we already know the ISO 7816 inter-industry 'GET CHALLENGE' command. This is discussed in the reverse engineering of the pin authentication protocol (section 7.2) and is also reported in table 4.4 (chapter 4). We also have the encrypted version of the 'GET CHALLENGE' command, which is reported in appendix A.3.6 (command/response pair 26) and table 7.16 (The commands reported below are for generating 24 bytes (24 = '18' in hexadecimal) using 'GET CHALLENGE' ISO 7816 command).

'GET CHALLENGE' unencrypted	00 84 00 00 18
'GET CHALLENGE' encrypted	0C 84 00 00 0D **97 01** 18 **8E 08** 1C 0B 23 9F 34 B7 29 FD **00**

Table 7.16: Secure Messaging Command Comparison

Following our review of multiple traces of different secure messaging sessions, we have found that the highlighted bytes are constant for commands with empty data fields. For commands with data fields we have found another 2 bytes that are also constant, these are '87' and '01'. It appears that control bytes are placed structurally throughout the encrypted command in order to separate the encrypted data field and the CMAC checksum.

Given the above finding, we review the GlobalPlatform 2.2 [?] standard which provides explanations for secure channel protocols. Our object here is to find how to build encrypted commands using secure messaging. Unfortunately this review does not provide any useful explanations with regard to how to build the encrypted commands.

Thus we carry out a further thorough research of the 'Athena smartcard solutions' security polices. We find one security policy [?] that discusses the possible use of Diffie Hellman for secure messaging. This security policy cites a paper by the *International Civil Aviation Organization* [?]. This latter paper is aims at explaining machine readable travel documents and their implementation. It shows a proprietary implementation of how to build the commands and responses for secure messaging sessions. The constant bytes present in the communication traces of secure messaging that we review for our smartcard match the constant bytes that are reported to be used in the methods explained in the paper. The diagrams reported below are extracted from the paper.

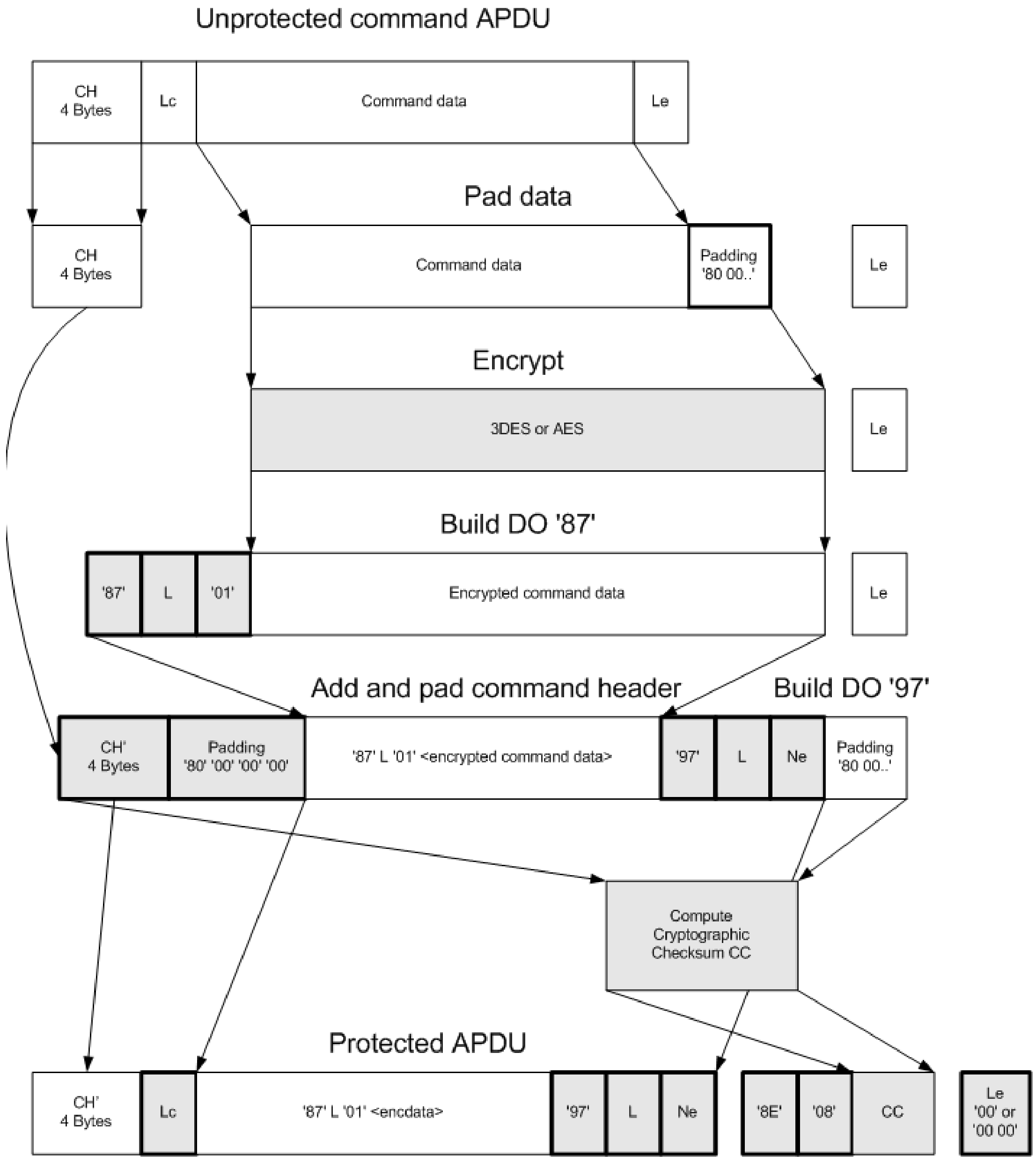


Figure 7.4: How to build APDU commands within secure messaging [?]

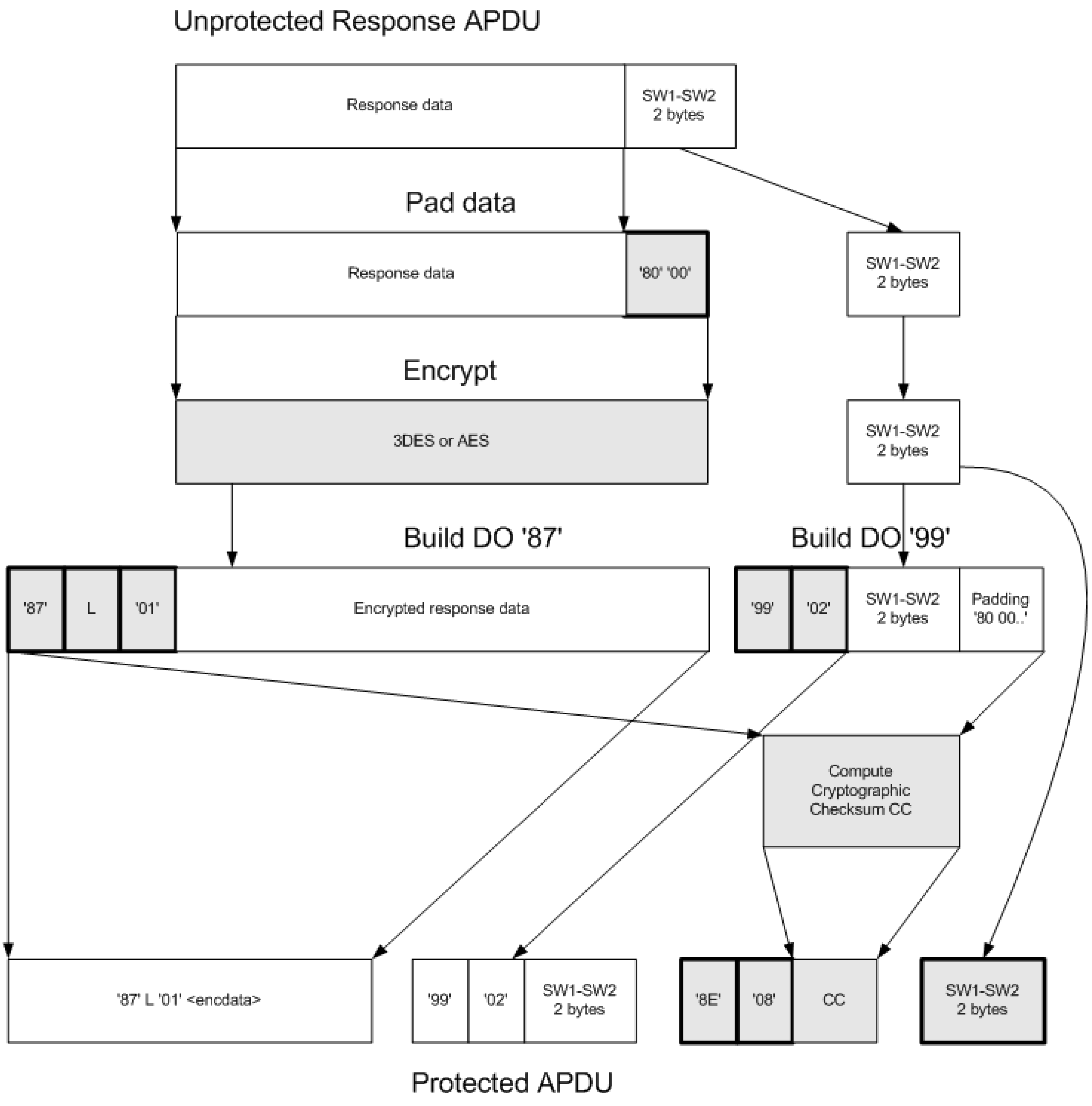


Figure 7.5: How to build APDU response's within secure messaging [?]

The paper suggests that only triple DES or AES block ciphers (see section 2.3, chapter 2) are used within secure messaging. With the knowledge of how to build the commands under secure messaging, we can deduce the length of the checksum (CMAC). For our smartcard the checksum length is 8 bytes. This therefore rules out the possibility of the use of AES, and only leaves triple DES. Triple DES has 3 key options. In experiment 1 we report that the first 16 bytes of the cards 32 byte challenge is used for deriving S_{Enc}, and the second 16 bytes are for S_{Mac}. Considering it is 16 bytes, and not 8 nor 24, we assume that key option 2 for triple DES is used for secure messaging.

Block Cipher	Key size	Block Size	Used in Secure Messaging
AES	16	16	χ
DES *(key option 1)*	8	8	χ
DES *(key option 2)*	**16**	**8**	✓
DES *(key option 3)*	24	8	χ

Based on the knowledge of how to build the encrypted commands, and the block cipher that is used, we can build the encrypted command for 'GET CHALLENGE' with different generated keys, using DES3 (key option 2) separately. The only remaining step is to generate session keys using different key derivation functions, and the 32 byte card challenge. Unfortunately, due to time constraints we are unable to complete this part of the experiment and therefore the reverse engineering of the secure messaging protocol. We will however explain the key derivation functions (KDF) we believe should be tested, and how to conduct the tests. This is for future work.

The first step in the experiment is to generate session keys using the 32 byte challenge from the card and the shared secret from the Diffie Hellman protocol. We report these two values in table 7.17. We also need to reintroduce the hash functions that the card supports as these will be used for the KDF's, in order to generate the session keys. These are reported in table 7.18.

Variable	**Value**
First 16 bytes of 32 byte card challenge (denoted **cc_1**)	C8 04 C7 25 A9 14 2D 58 E8 01 64 6D 72 DA C4 9C
Second 16 bytes of 32 byte card challenge (denoted **cc_2**)	A5 F0 D1 FA AF 53 93 A7 49 8D 69 8B 7A 1A D0 A4
Shared secret (denote **ss**)	00

Table 7.17: Key generation parameters

Hash Name	Output Length (bits)	Output Length (Bytes)
MD5	128	16
SHA1	160	20
SHA256	256	32
SHA384	384	48
SHA512	512	64

Table 7.18: Hash Functions *(supported by the card)*

Using the data in tables 7.17 and 7.18, we can now generate session keys S_{Enc} and S_{Mac} using key derivation techniques described in table 7.19.

KDF	S_{Enc}	S_{Mac}
1	Truncate(ss XOR cc_1)	Truncate(ss XOR cc_2)
2	Truncate(Hash(ss || cc_1))	Truncate(Hash(ss || cc_2))
3	Truncate(Hash(ss) XOR cc_1)	Truncate(Hash(ss) XOR cc_2)
4	Truncate(Hash(ss||cc_1||cc_2))[0:16]	Truncate(Hash(ss||cc_1||cc_2))[16:32]
5	Truncate(Hash(s))[0:16] XOR cc_1	Truncate(Hash(ss))[16:32] XOR cc_2

Table 7.19: Methods for generation session keys

[Note that for KDF 4 and 5, the hash functions used must have an output size greater than or equal to 32 bytes. Thus, MD5 and sha1 cannot be used.]

With the generated session keys S_{Enc} and S_{Mac} it is now possible to build the encrypted command for 'GET CHALLENGE' for comparison against the command that will be generated by the API (with the correct session keys). To build a command (given generated session keys), follow the steps that are given in figure 7.4. Since the data field of the 'GET CHALLENGE' command is empty, the only computation that is required is the checksum (CMAC) of the entire message. This can be calculated by following these steps:

1. Block cipher = Triple DES - key option 2
2. Mode = C-MAC
3. IV = 00 00 00 00 00 00 00 00
4. Password = S_{Mac}
5. Message = 0C 84 00 00 80 00 00 00 97 01 08 80 00 00 00 00

$$Checksum = encrypt(Message)$$

Encrypted command = 0C 84 00 00 0D 97 01 18 8E 08 $Checksum$ 00

If a match between the encrypted command that is generated and the command the API encrypts is found, the method for deriving S_{Enc} and S_{Mac} will be known. This completes the reverse engineering of the proprietary secure messaging protocol implemented on our smartcard.

In the case that none of the methods suggested in table 7.19 do not reverse engineer the session keys S_{Enc} and S_{Mac} there is also another possibility that within the key derivation function a counter value is used. This is reported as an option in the machine readable travel document paper [?]. It states 2 possibilities, either an initial value of 0 is used, or the counter value can be generated from the first 4 bytes of the nonce sent by the API and the first 4 bytes of the smartcards challenge concatenated together. These bytes are from the login authentication that we reversed engineered in the section 7.2. Thus, the next option to test afterwards (in the case of failing to find a match) is:

$$S_{Enc} = Truncate(\ Hash(\ ss \,||\, counter\ ))\ XOR\ cc\_1$$
$$S_{Mac} = Truncate(\ Hash(\ ss \,||\, counter\ ))\ XOR\ cc\_2$$

Chapter 8

Conclusion

We examine in this project the Athena IDprotect smartcard that uses encrypted APDU communication in an attempt to prevent sensitive information from being revealed at the APDU level. To the best of our, knowledge this is the first study that examines reverse engineering a proprietary implementation of encrypting APDU communication between the smartcard and the middleware. We use knowledge of cryptographic protocols, the PKCS #11 standard, the ISO/IEC 7816 standard and literature to analyse the communication between the smartcard, the card vendors middlware and successfully complete the attacks. In summary we complete the following in our project.

We successfully reverse engineer the proprietary implementation for PIN authentication. Using the protocol we deduce in section 7.2, an attacker can brute force/ use a dictionary attack to calculate a users PIN given one successful login (APDU) communication trace. This seriously violates the PKCS #11 standard.

We also successfully reverse engineer the cryptographic protocol that is implemented at the secure messaging (SM) initialization process. We find that this process uses the Diffie Hellman protocol. We implement a man in the middle attack to set the shared secret to a value of zero. Following our thorough analysis of a number of security policies of 'Athena Smartcard Solutions', we find a paper by the International Civil Aviation Organization [?] that describes how to build encrypted APDU commands and responses that matches that of our smartcard. The final step involves using the 32 byte card challenge, the shared secret (set to 0), a key derivation function and the encrypted command 'GET CHALLENGE' to calculate the SM session keys S_{Enc} and S_{Mac}. We leave this for future work as time constraints does not permit its completion.

Our contribution to this future work is that we provide detailed explanations (in section 7.3, chapter 7) of the steps that need to be taken to complete the reverse engineering of the secure messaging protocol. We are confident that using our suggestions and experimental setup, this attack can be completed. If completed this will reveal sensitive key values of block cipher keys created and/or generated. It will also lead onto the investigation of whether or not the Clulow (2003) [?] 'wrap/decrpyt attack' can be achieved at the APDU level on our smartcard. The attack is not achievable at this stage due to the removal of the wrap functionality at the PKCS #11 API level of our smartcard. We believe from the investigation we undertook in section 6.11, the security operation for wrapping is still supported at the APDU level. If this is achieved the attacker can extract all keys including RSA private keys from our smartcard. This will be in direct violation of the PKCS #11 standard. We provide the steps necessary for this second investigation within section 8.1.

We test the vulnerability presented in Bozzato et al. (2016) [?] whereby attribute controls at the PKCS #11 API level can be overridden at the APDU level. The results of the test show that our smartcard suffers from the same vulnerability, and that security operations (e.g. encrypt, sign) can be conducted on the smartcard regardless of the values of the keys attributes, that dictate if they key should be able to do so.

We provide an extended analysis of how PKCS #11 functions are broken down into ISO 7816 APDU command response pairs (for our projects smartcard). In the cases of C_generateKey, C_generateKeyPair, C_destroyObject, C_wrap and C_unwrap we provide new insights.

Finally we extend the work of Lan (2016) [?] to include the key and attribute file locations of not only block cipher keys, but also RSA public and private keys.

8.1 Future work

The first suggestion we have for future work is the completion of the reverse engineering of the secure messaging protocol for our smartcard. The explanation for how to do this can be found in chapter 7, section 7.3. We are confident that this can be achieved. If this is achieved, the first attack that will then be possible is the decryption of block cipher key values as they are being created/ generated with C_createObject and C_generateKey respectively.

More importantly is our second attack suggestion. Based on the reverse engineering of the secure messaging protocol, our second suggestion for future work is the implementation of the 'wrap/decrypt attack' at the APDU level. This should be able to be achieved by completing the following:

- Generate a new key with C_generateKey function and decrypt the data field for create file (when creating the key file)

- Unwrap a key using C_unwrap and decrypt the data field of the 'unwrap' security operation APDU command

We suggest decrypting the data field for the creation of a key file to find the understanding of how to create and store in the smartcards memory, a block cipher key without the need for the middlware. This is the first part to be completed before we can complete 'wrap/decrypt' attack at the APDU level.

The second part concerns our discussion in chapter 6, section 6.11. We find the APDU command we assume to correspond to the 'wrap' security operation. We tested an example command and find an APDU level error, stating the parameters in the data field are incorrect. We think that studying the data field of the un-encrypted 'unwrap' APDU command might give insight into how the 'wrap' data field is defined. This is the reason for decrypting the data field of the 'unwrap' APDU command. If this does not achieve the understanding required for the APDU 'wrap' command, a study into previous hardware security modules and/or middlware versions (of 'Athena Smartcard Solutions') could find the correct correct APDU 'wrap' command.

Once these two additional understandings are found, using the knowledge we provide in this project the following attack should be able to take place on the smartcard. The attack is at the APDU level and does not require the use of the middlware (just send APDU commands directly to the smartcard).

- Using 'CREATE FILE' APDU command, create a block cipher key$_1$ (see table 4.4, chapter 4) at a file location that is empty. (See table 4.3 for file locations)
- Incrementally select key file directories with 'SELECT FILE' APDU command. This will return '90 00' in the case that the file exists. Or an APDU error in the case that the key file does not exist. This will provide the information regarding the locations of saved keys on the smartcard.
- For each key$_i$ found, use the 'wrap' APDU command to wrap the key$_i$ with key$_1$. This will return in the APDU response the encrypted key, key$_e$
- Using the value we set for key$_1$, decrypt key$_e$ outside of the smartcard.

All key values stored in the smartcard regardless of any CKA_attribute values should now appear in an un-encrypted format outside of the smartcard. This is in direct violation of the PKCS #11 standard. And if successful is a very serious attack. Due to sending the commands directly to the smartcard, the middlware is no longer required. Therefore, the mitigation suggested by Bozzato et al. (2016) [?] of running the middlware on a separate process and different privilege to user would not mitigate this attack.

As we only study one smartcard in our project, a study into more PKCS #11 hardware security modules that use encrypted APDU command response pairs should take place. On the basis of the above attack being successful, one of the aims of the study should be to find an understanding of whether the above attack is possible on other (more secure) smartcards.

If this is the case, this could provide evidence of whether a change in the standards for contact smartcards is necessary. In the following chapter we hypothesise that it might only be possible to provide aspects of the security of PKCS #11, by implementing the PKCS #11 standard on the smartcard. Thus, it would be the hardware to reject requests for security operations to take place, when the keys attributes do not permit it. Not the middleware on the computer. This will prevent APDU level attacks that override the controls the PKCS #11 API dictates. It will also allow for the key separation principle to be enforced and never overridden at the APDU level, preventing the 'wrap/decrpyt attack'.

8.2 Mitigation And Recommendations

Smartcards should provide isolated security especially for the sensitive key values that are stored in their memory. This security should be provided regardless of the threat model. Therefore the assumption that a physical man in the middle attack is in process (on the users system), should remain true throughout the development process of smartcards, to provide the security against this happening.

It has been proven that our projects smartcard does not provide this protection, by the attacks we successfully complete. Bozzato et al. (2016) [?] also report the smartcards they examine providing no detection nor protection against man in the middle attacks.

The methods for detecting man in the middle attacks require synchronous clocks between the smartcard and the users computer. Synchronizing clocks will allow for relative time measurements between the smartcard and the API to be calculated. This task is far from simple, and so far has not been reported as a methodology used by card vendors (that create PKCS #11 hardware security modules) in literature. A trivial example of a detection method would be to calculate the averaged time between each specific APDU command and its corresponding response. If for a specific command response pair, the time exceeds X milliseconds assume a man in the middle attack is occurring.

The implementation of detection and prevention of man in the middle attacks for smartcards that do not already support it, will be of high cost. A re-design to the smartcards hardware will most likely be required. This in most cases should prevent similar analysis's that we undertook in this project from being completed. This would aid the prevention of attackers gaining knowledge that is required to carry out the attacks we present and suggest as part of future work.

Mitigations suggested in literature [?] are more reasonable for smartcard manufacturers to be able to implement at a much lower cost. These include:

1. The key separation principle at the API layer. One key/ key pair should not hold the ability for wrapping/unwrapping and encryption/decryption. They should hold only one of those sets.

2. Using the OTP functionality of some hardware security modules (usb tokens) for PIN authentication protocols. This should make it far more difficult for an attacker to reverse engineer the PIN authentication protocol. In the case they do, the attacker will still require physical access to the device.

3. Running the middleware on a different process and at a higher privilege than the user of the system. This should prevent an attacker from attaching to the process and observing/ altering APDU communication.

(1) as we discuss as part of our second suggestion for future work might be of no use if this can be overridden at the APDU level. (2) would only apply to that of hardware secuirty modules with the OTP functionality implemented and have a display unit. (3) does not prevent an attacker from gaining the understanding of the smartcards middleware implementation on his own computer (as he can give himself the same privilege level as the middleware). With that understanding an attacker could then send APDU commands directly to the smartcard on a users infected machine without the use of the middleware. Thus rendering the approach in-applicable to those forms of attacks.

These forms of attacks rely on the fact that an attacker is able to gain a detailed understanding of how the card manufacturers middleware is implemented. We therefore suggest that the middleware that is released to customers be stripped of proper idiomatic function names, replacing them with names that only have meaning to the manufacturer. This will only help reduce the ability for an attacker to gain this understanding. It will however not prevent it, as reported in Gkaniatsou et al. (2015) [?] achieving this understanding is possible without the use of the manufacturers middleware.

Furthermore detection and prevention of man in the middle attacks should be implemented. This will seriously halt the ability to gain the understanding required for the APDU level attacks discussed, if implemented correctly. However there is no guarantee that detection and prevention is possible for man in the middle attacks that are automated by software. Considering clock synchronization between two devices always incurs a lag, it is possible that a software man in the middle attack could go undetected. This is worth investigating for card manufactures but as stated before would not come at a cheap cost.

Finally, in the instance of our projects smartcard, good defense programming techniques would have made it far more difficult to reverse engineer the protocols we studied. For example not seeding the random number generator with only the current time (in PIN authentication). Or not allowing invalid values such as zero for the generator (for secure messaging). Defensive programming techniques would however only of made the job of reverse engineering the protocols more difficult, not prevent them. Thus we argue that the methods suggested above will only provide obscurity, not security. The hardware security modules discussed in this project, Gkaniatsou et al. (2015) and Bozzato et al. (2016) were once thought to be secure devices. A study into hardware security modules that are thought to be more secure could provide evidence that supports the mitigations that we suggest below (which are of high cost):

- The ability for a user to set the key values using C_createObject be removed
- All key values are to be generated by the smartcard's hardware.
- PKCS #11 attribute controls should be enforced by the smartcard's hardware
- The key separation principle should be enforced by the smartcard's hardware
- All key values are to be stored in internal EF's and never be extracted out of smartcard 'unwrapped' / un-encrypted.
- "All cryptographic operations take place in the smartcard" [?]
- The status of a user being authenticated (correct PIN) should be controlled by smartcard's hardware. And prevent invalid function requests if their authentication status is false.5

The above suggestions would require a re-design to the smartcards internal operation, however would prevent all the attacks we discuss regarding the disclosure of sensitive key values. It would also provide the security the PKCS #11 API is expected to provide, which recent literature is showing can be overridden at the APDU level. This is due to the fact that the generation of keys will all take place within the smartcard with a command specifying the key type. This will remove the need for generating the key value and storing it in a file with 2 commands (requiring secure messaging to try and prevent revealing the key value). Furthermore the removal for the ability for a user to set the key value using C_createObject will remove the need for secure messaging entirely. Which should now prevent the sensitive key value of any type of key from being leaked at the APDU level during key generation.

Since the responsibility of the controls set out by the PKCS #11 attribute values are to be delegated to the smartcards hardware, it will not be possible to override them at the APDU level. With the key separation principle also enforced by the smartcards hardware the ’wrap/decrypt attack’ will not be possible at the API or the APDU level. Finally separating file location of sensitive key values and placing them in internal EF’s should prevent the opening and disclosure of the sensitive key values at the APDU level (as see n in our smartcards project). [**?**]

Bibliography

[1] *Block Cipher Modes Image Source*, (accessed April 6, 2017). https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation.

[2] *ECC Image Source*, (accessed April 6, 2017). http://bitcoin.stackexchange.com/questions/37289/in-elliptic-curve-addition-where-does-the-second-point-come-from.

[3] Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. Attacking and fixing pkcs# 11 security tokens. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 260–269. ACM, 2010.

[4] Claudio Bozzato, Riccardo Focardi, Francesco Palmarini, and Graham Steel. Apdu-level attacks in pkcs# 11 devices. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 97–117. Springer, 2016.

[5] Jolyon Clulow. On the security of pkcs# 11. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 411–425. Springer, 2003.

[6] Oasis Technical Committee. *OASIS PKCS 11 TC*, (accessed October 15, 2016). https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=pkcs11.

[7] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[8] Andriana Gkaniatsou, Fiona McNeill, Alan Bundy, Graham Steel, Riccardo Focardi, and Claudio Bozzato. Getting to know your card: reverse-engineering the smart-card application protocol data unit. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 441–450. ACM, 2015.

[9] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[10] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. Rfc 2104: Hmac: Keyed-hashing for message authentication. 1997.

[11] Xiao Lan. *Finding Vulnerabilities In Low Level Protocols*, 2016.

[12] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[13] GN Shinde and HS Fadewar. Faster rsa algorithm for decryption using chinese remainder theorem. In *International Conference on Computational and Experimental Engineering and Sciences (ICCES)*, volume 5, pages 255–262, 2008.

[14] Secure Hash Standard. Fips pub 180-2. *National Institute of Standards and Technology*, 2002.

Appendices

Appendix A

Attack Traces

A.1 Multiple C_login Traces

A.1.1 Different Second

```
----- APDU command/response pair 0 -----
(Inter-Industry) Get Challenge
00000000: 00 84 00 00 08                                      .....

00000000: 00 00 00 00 00 00 00 00  90 00                      ..........

----- APDU command/response pair 1 -----
(Proprietary) Verify
00000000: 80 20 00 00 10 61 50 65  E1 AF 05 7B C3 35 98 0D  . ...aPe...{.5..
00000010: DC 9D C5 42 96                                      ...B.

00000000: 63 C9                                               c.

----- APDU command/response pair 1 -----
(Inter-Industry) Get Challenge
00000000: 00 84 00 00 08                                      .....

00000000: 00 00 00 00 00 00 00 00  90 00                      ..........

----- APDU command/response pair 1 -----
(Proprietary) Verify
00000000: 80 20 00 00 10 BF 73 83  F9 30 B1 74 D2 E4 98 83  . ....s..0.t....
00000010: 3A 9F 1F 37 BA                                      :..7.

00000000: 63 C8                                               c.
```

A.1.2 Same Pin, Same Challenge

```
----- APDU command/response pair 0 -----
(Inter-Industry) Get Challenge
00000000: 00 84 00 00 08                                    .....

00000000: 00 00 00 00 00 00 00 00  90 00                    ..........

----- APDU command/response pair 1 -----
(Proprietary) Verify
00000000: 80 20 00 00 10 EE D2 F1  54 07 18 8A 8F AB A3 F7  . ......T.......
00000010: 3E 64 17 2D 6E                                    >d.-n

00000000: 63 C9                                             c.

----- APDU command/response pair 1 -----
(Inter-Industry) Get Challenge
00000000: 00 84 00 00 08                                    .....

00000000: 00 00 00 00 00 00 00 00  90 00                    ..........

----- APDU command/response pair 1 -----
(Proprietary) Verify
00000000: 80 20 00 00 10 EE D2 F1  54 07 18 8A 8F AB A3 F7  . ......T.......
00000010: 3E 64 17 2D 6E                                    >d.-n

00000000: 63 C8                                             c.
```

A.1.3 Same Pin, Different Challenge

```
----- APDU command/response pair 0 -----
(Inter-Industry) Get Challenge
00000000: 00 84 00 00 08                                    .....

00000000: 00 00 00 00 00 00 00 00  90 00                    ..........

----- APDU command/response pair 1 -----
(Proprietary) Verify
00000000: 80 20 00 00 10 6E 78 D4  D5 61 AD 3C 26 D3 89 E8  . ...nx..a.<&...
00000010: 96 B9 92 0D 40                                    ....@

00000000: 63 C9                                             c.

----- APDU command/response pair 1 -----
(Inter-Industry) Get Challenge
00000000: 00 84 00 00 08                                    .....
```

```
00000000: 00 00 00 00 00 00 00 01  90 00                    ..........

----- APDU command/response pair 1 -----
(Proprietary) Verify
00000000: 80 20 00 00 10 6E 78 D4  D5 61 AD 3C 26 BC C6 AA  . ...nx..a.<&...
00000010: 72 D3 95 2B 94                                    r..+.

00000000: 63 C8                                             c.
```

A.1.4 Different Pin, Same Challenge

```
----- APDU command/response pair 0 -----
(Inter-Industry) Get Challenge
00000000: 00 84 00 00 08                                    .....

00000000: 00 00 00 00 00 00 00 00  90 00                    ..........

----- APDU command/response pair 1 -----
(Proprietary) Verify
00000000: 80 20 00 00 10 8F 58 1B  91 BC 78 D3 37 A9 D9 FB  . ....X...x.7...
00000010: 9C 20 58 F6 0A                                    . X..

00000000: 63 C9                                             c.

----- APDU command/response pair 1 -----
(Inter-Industry) Get Challenge
00000000: 00 84 00 00 08                                    .....

00000000: 00 00 00 00 00 00 00 00  90 00                    ..........

----- APDU command/response pair 1 -----
(Proprietary) Verify
00000000: 80 20 00 00 10 E0 30 33  BB 03 0F 6E 11 08 C0 8D  . ....03...n....
00000010: 1D 9D 85 C4 A6                                    .....

00000000: 63 C8                                             c.
```

A.1.5 Different Pin, Different Challenge

```
----- APDU command/response pair 0 -----
(Inter-Industry) Get Challenge
00000000: 00 84 00 00 08                                    .....

00000000: 00 00 00 00 00 00 00 00  90 00                    ..........
```

```
----- APDU command/response pair 1 -----
(Proprietary) Verify
00000000: 80 20 00 00 10 C2 29 5F  78 D1 68 29 13 78 BE 7B  . ....)_x.h).x.{
00000010: 8E 61 9B 32 2E                                    .a.2.

00000000: 63 C9                                             c.

----- APDU command/response pair 1 -----
(Inter-Industry) Get Challenge
00000000: 00 84 00 00 08                                    .....

00000000: 00 00 00 00 00 00 00 01  90 00                    ..........

----- APDU command/response pair 1 -----
(Proprietary) Verify
00000000: 80 20 00 00 10 C4 6D 44  03 92 F3 6B EF 13 18 07  . ....mD...k....
00000010: CE 5A A4 B9 27                                    .Z..'

00000000: 63 C8                                             c.
```

A.2 Successful Login Injection

```
----- APDU command/response pair 12 -----

(Inter-Industry) Get Challenge
COMMAND from API
00000000: 00 84 00 00 08                                    .....

Do you want to automate the injection your own login response? (Y/n)

RESPONSE
00000000: E7 69 60 B5 C8 FC D2 02  90 00                    .i‘.......

----- APDU command/response pair 13 -----

(Proprietary) Verify
COMMAND from API
00000000: 80 20 00 00 10 4A D1 3D  AB 98 7F C5 18 A9 B3 1F  . ...J.=........
00000010: 2F 96 B4 3C AF                                    /..<.

(Proprietary) Verify
COMMAND injected
00000000: 80 20 00 00 10 32 5A 9F  38 CA 4F BE 44 3A CD E1  . ...2Z.8.O.D:..
00000010: C5 03 84 35 DF                                    ...5.

RESPONSE
00000000: 90 00                                             ..
```

A.3 Open Secure Messaging Traces

A.3.1 Generator = 5, [Not modified]

```
----- APDU command/response pair 24 -----

(Proprietary) Get Card Public Key
COMMAND from API
00000000: 80 48 00 80 00                                      .H...

Do you want to to alter command? (y/N)

RESPONSE
00000000: 80 01 05 81 81 80 F7 B5  15 72 07 22 94 6F C4 08  .........r.".o..
00000010: 64 CB BD AF EA 55 7D BD  8F 55 36 B0 01 C2 8B 2E  d....U}..U6.....
00000020: 32 B6 5D 45 F1 74 5D 38  12 0B AD 9D 2C 03 9C 22  2.]E.t]8....,.."
00000030: 46 68 EB 2E A2 8C 20 95  A8 2E 6C A8 E0 6D 47 F2  Fh.... ...l..mG.
00000040: D3 1E D7 01 F8 15 5C AD  DC 05 70 C0 93 B2 6D 74  ......:.p...mt
00000050: B0 9B 95 E6 4D 8C D2 FC  73 3E CD 0F 30 68 79 A5  ....M...s>..0hy.
00000060: B9 35 F2 41 3F 52 AD AD  32 A0 99 1A 18 3D CC 57  .5.A?R..2....=.W
00000070: 7E 39 DA 47 53 1E 67 15  AB 01 70 7F F2 47 96 71  ~9.GS.g...p..G.q
00000080: 44 23 CE 7B 60 67 82 81  80 3C 52 D2 06 89 28 92  D#.{‘g...<R...(.
00000090: 2C AB E6 3C 4E E6 DF 0E  D2 29 F1 01 BE 36 C4 F8  ,..<N....)...6..
000000A0: 54 40 56 F3 4A FA 8D 2E  9B 60 F5 07 BC ED B4 44  T@V.J....‘.....D
000000B0: 56 68 5D 82 4C C4 EA D7  96 20 F8 C5 46 A6 E0 16  Vh].L.... ..F...
000000C0: B8 AB A5 D8 43 29 58 53  77 17 09 97 AA 70 68 33  ....C)XSw....ph3
000000D0: 9E F1 41 0A 5F 39 D9 75  24 7F 3A 53 63 61 47 87  ..A._9.u$.:ScaG.
000000E0: 87 7F 88 96 BC BB 83 A1  CB D1 42 E0 EB 99 CF 34  ..........B....4
000000F0: 0E CA 56 4F 2C 57 50 6E  7B 1A FC 1F 90 7A E0 C2  ..VO,WPn{....z..
00000100: 61 09                                               a.

Do you want to alter the response? (y/N)

----- APDU command/response pair 25 -----

(Inter-Industry) Get Remaining Bytes
COMMAND from API
00000000: 00 C0 00 00 09                                      .....

Do you want to to alter command? (y/N)

RESPONSE
00000000: A8 5D D3 30 E3 5C A9 00  39 90 00                   .].0....9..

Do you want to alter the response? (y/N)

----- APDU command/response pair 26 -----
```

(Proprietary) Open Secure Messaging

```
COMMAND from API
00000000: 80 86 00 00 80 08 9F EA  A1 DC 8F C3 43 FD FD 4A  ...........C..J
00000010: E6 95 7E C0 D3 C6 FE 81  61 59 4B CE 45 21 96 63  ..~.....aYK.E!.c
00000020: 0F AB 19 D8 61 1A B2 6B  00 E2 44 0F 06 A3 5B 60  ....a..k..D...[‘
00000030: 87 76 C0 B7 E9 15 D5 50  DB 17 D6 C1 3C 26 54 47  .v.....P....<&TG
00000040: AA A3 4B DC 2C 14 81 08  84 0D F0 CA FB 49 8B C1  ..K.,........I..
00000050: B1 0B A1 2B 86 20 02 F2  0F 69 F0 56 2C 83 0C 6E  ...+. ...i.V,..n
00000060: A6 6A E9 86 56 47 71 24  0C B7 91 7F 37 85 0A D4  .j..VGq$....7...
00000070: 12 35 1F CE 17 6C D2 52  FB 04 24 CF DD E9 53 BE  .5...l.R..$...S.
00000080: DA 26 EA 54 FB 00                                .&.T..
Do you want to to alter command? (y/N)

RESPONSE
00000000: 66 56 36 31 16 42 8D 8A  BC 06 BA AC 5D 35 26 F5  fV61.B......]5&.
00000010: BF 58 15 7F 00 4F EF 2F  54 FB C4 F2 10 8F CB D6  .X...O./T.......
00000020: 90 00                                            ..

Do you want to alter the response? (y/N)

----- APDU command/response pair 27 -----
```

(Proprietary) Get Challenge [SM]

```
COMMAND from API
00000000: 0C 84 00 00 0D 97 01 20  8E 08 05 E4 4A 19 32 DE  ....... ....J.2.
00000010: 51 CB 00                                         Q..

Do you want to to alter command? (y/N)

RESPONSE
00000000: 87 29 01 BD 69 F3 85 A7  98 2E 08 07 21 88 30 2F  .)..i.......!.0/
00000010: 06 FF 93 E4 2F 31 C5 4A  40 FB 45 3A 45 C1 4A 84  ..../1.J@.E:E.J.
00000020: 7F BA 59 BC 44 8A 70 A0  BC DA FB 99 02 90 00 8E  ..Y.D.p.........
00000030: 08 44 26 95 74 6A 51 A3  72 90 00                .D&.tjQ.r..

Do you want to alter the response? (y/N)

----- APDU command/response pair 28 -----
```

(Proprietary) Close Secure Messaging

```
COMMAND from API
00000000: 80 86 FF FF                                      ....
```

A.3.2 Generator = 1

```
----- APDU command/response pair 24 -----
```

(Proprietary) Get Card Public Key

```
COMMAND from API
00000000: 80 48 00 80 00                                      .H...

Do you want to to alter command? (y/N)

RESPONSE
00000000: 80 01 05 81 81 80 F7 B5  15 72 07 22 94 6F C4 08  .........r.".o..
00000010: 64 CB BD AF EA 55 7D BD  8F 55 36 B0 01 C2 8B 2E  d....U}..U6.....
00000020: 32 B6 5D 45 F1 74 5D 38  12 0B AD 9D 2C 03 9C 22  2.]E.t]8....,.."
00000030: 46 68 EB 2E A2 8C 20 95  A8 2E 6C A8 E0 6D 47 F2  Fh.... ...l..mG.
00000040: D3 1E D7 01 F8 15 5C AD  DC 05 70 C0 93 B2 6D 74  ......:.p...mt
00000050: B0 9B 95 E6 4D 8C D2 FC  73 3E CD 0F 30 68 79 A5  ....M...s>..0hy.
00000060: B9 35 F2 41 3F 52 AD AD  32 A0 99 1A 18 3D CC 57  .5.A?R..2....=.W
00000070: 7E 39 DA 47 53 1E 67 15  AB 01 70 7F F2 47 96 71  ~9.GS.g...p..G.q
00000080: 44 23 CE 7B 60 67 82 81  80 3C 52 D2 06 89 28 92  D#.{‘g...<R...(.
00000090: 2C AB E6 3C 4E E6 DF 0E  D2 29 F1 01 BE 36 C4 F8  ,..<N....)...6..
000000A0: 54 40 56 F3 4A FA 8D 2E  9B 60 F5 07 BC ED B4 44  T@V.J....‘.....D
000000B0: 56 68 5D 82 4C C4 EA D7  96 20 F8 C5 46 A6 E0 16  Vh].L.... ..F...
000000C0: B8 AB A5 D8 43 29 58 53  77 17 09 97 AA 70 68 33  ....C)XSw....ph3
000000D0: 9E F1 41 0A 5F 39 D9 75  24 7F 3A 53 63 61 47 87  ..A._9.u$.:ScaG.
000000E0: 87 7F 88 96 BC BB 83 A1  CB D1 42 E0 EB 99 CF 34  ..........B....4
000000F0: 0E CA 56 4F 2C 57 50 6E  7B 1A FC 1F 90 7A E0 C2  ..VO,WPn{....z..
00000100: 61 09                                               a.

Do you want to alter the response? (y/N)
y

00000000: 80 01 01 81 81 80 F7 B5  15 72 07 22 94 6F C4 08  .........r.".o..
00000010: 64 CB BD AF EA 55 7D BD  8F 55 36 B0 01 C2 8B 2E  d....U}..U6.....
00000020: 32 B6 5D 45 F1 74 5D 38  12 0B AD 9D 2C 03 9C 22  2.]E.t]8....,.."
00000030: 46 68 EB 2E A2 8C 20 95  A8 2E 6C A8 E0 6D 47 F2  Fh.... ...l..mG.
00000040: D3 1E D7 01 F8 15 5C AD  DC 05 70 C0 93 B2 6D 74  ......:.p...mt
00000050: B0 9B 95 E6 4D 8C D2 FC  73 3E CD 0F 30 68 79 A5  ....M...s>..0hy.
00000060: B9 35 F2 41 3F 52 AD AD  32 A0 99 1A 18 3D CC 57  .5.A?R..2....=.W
00000070: 7E 39 DA 47 53 1E 67 15  AB 01 70 7F F2 47 96 71  ~9.GS.g...p..G.q
00000080: 44 23 CE 7B 60 67 82 81  80 3C 52 D2 06 89 28 92  D#.{‘g...<R...(.
00000090: 2C AB E6 3C 4E E6 DF 0E  D2 29 F1 01 BE 36 C4 F8  ,..<N....)...6..
000000A0: 54 40 56 F3 4A FA 8D 2E  9B 60 F5 07 BC ED B4 44  T@V.J....‘.....D
000000B0: 56 68 5D 82 4C C4 EA D7  96 20 F8 C5 46 A6 E0 16  Vh].L.... ..F...
000000C0: B8 AB A5 D8 43 29 58 53  77 17 09 97 AA 70 68 33  ....C)XSw....ph3
000000D0: 9E F1 41 0A 5F 39 D9 75  24 7F 3A 53 63 61 47 87  ..A._9.u$.:ScaG.
000000E0: 87 7F 88 96 BC BB 83 A1  CB D1 42 E0 EB 99 CF 34  ..........B....4
000000F0: 0E CA 56 4F 2C 57 50 6E  7B 1A FC 1F 90 7A E0 C2  ..VO,WPn{....z..
00000100: 61 09                                               a.
response changed!

----- APDU command/response pair 25 -----
```

(Inter-Industry) Get Remaining Bytes

```
COMMAND from API
00000000: 00 C0 00 00 09                                    .....

Do you want to to alter command? (y/N)

RESPONSE
00000000: A8 5D D3 30 E3 5C A9 00  39 90 00                 .].0....9..

Do you want to alter the response? (y/N)

----- APDU command/response pair 26 -----
```

(Proprietary) Open Secure Messaging

```
COMMAND from API
```

`00000000: 80 86 00 00 80 00 00 00  00 00 00 00 00 00 00 00  ................`
`00000010: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................`
`00000020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................`
`00000030: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................`
`00000040: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................`
`00000050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................`
`00000060: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................`
`00000070: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................`
`00000080: 00 00 00 00 01 00                                ......`

```
Do you want to to alter command? (y/N)

RESPONSE
00000000: 7D 2C 25 47 1C 16 34 51  E9 C3 49 38 C8 79 1E ED  },%G..4Q..I8.y..
00000010: A2 6B 20 D4 54 BD 67 0A  D3 85 3E B9 E0 6E D5 5E  .k .T.g...>..n.^
00000020: 90 00                                             ..

Do you want to alter the response? (y/N)

----- APDU command/response pair 27 -----
```

(Proprietary) Get Challenge [SM]

```
COMMAND from API
00000000: 0C 84 00 00 0D 97 01 20  8E 08 08 C6 59 9B 57 E6  ....... ....Y.W.
00000010: B4 4E 00                                          .N.

Do you want to to alter command? (y/N)

RESPONSE
00000000: 69 88                                             i.

Do you want to alter the response? (y/N)

----- APDU command/response pair 28 -----
```

```
(Proprietary) Close Secure Messaging
COMMAND from API
00000000: 80 86 FF FF                                       ....
```

A.3.3 Generator = 0

```
----- APDU command/response pair 24 -----

(Proprietary) Get Card Public Key
COMMAND from API
00000000: 80 48 00 80 00                                    .H...

Do you want to to alter command? (y/N)

RESPONSE
00000000: 80 01 05 81 81 80 F7 B5  15 72 07 22 94 6F C4 08  .........r.".o..
00000010: 64 CB BD AF EA 55 7D BD  8F 55 36 B0 01 C2 8B 2E  d....U}..U6.....
00000020: 32 B6 5D 45 F1 74 5D 38  12 0B AD 9D 2C 03 9C 22  2.]E.t]8....,.."
00000030: 46 68 EB 2E A2 8C 20 95  A8 2E 6C A8 E0 6D 47 F2  Fh.... ...l..mG.
00000040: D3 1E D7 01 F8 15 5C AD  DC 05 70 C0 93 B2 6D 74  ......:.p...mt
00000050: B0 9B 95 E6 4D 8C D2 FC  73 3E CD 0F 30 68 79 A5  ....M...s>..0hy.
00000060: B9 35 F2 41 3F 52 AD AD  32 A0 99 1A 18 3D CC 57  .5.A?R..2....=.W
00000070: 7E 39 DA 47 53 1E 67 15  AB 01 70 7F F2 47 96 71  ~9.GS.g...p..G.q
00000080: 44 23 CE 7B 60 67 82 81  80 3C 52 D2 06 89 28 92  D#.{‘g...<R...(.
00000090: 2C AB E6 3C 4E E6 DF 0E  D2 29 F1 01 BE 36 C4 F8  ,..<N....)...6..
000000A0: 54 40 56 F3 4A FA 8D 2E  9B 60 F5 07 BC ED B4 44  T@V.J....‘.....D
000000B0: 56 68 5D 82 4C C4 EA D7  96 20 F8 C5 46 A6 E0 16  Vh].L.... ..F...
000000C0: B8 AB A5 D8 43 29 58 53  77 17 09 97 AA 70 68 33  ....C)XSw....ph3
000000D0: 9E F1 41 0A 5F 39 D9 75  24 7F 3A 53 63 61 47 87  ..A._9.u$.:ScaG.
000000E0: 87 7F 88 96 BC BB 83 A1  CB D1 42 E0 EB 99 CF 34  ..........B....4
000000F0: 0E CA 56 4F 2C 57 50 6E  7B 1A FC 1F 90 7A E0 C2  ..VO,WPn{....z..
00000100: 61 09                                             a.

Do you want to alter the response? (y/N)
y

00000000: 80 01 00 81 81 80 F7 B5  15 72 07 22 94 6F C4 08  .........r.".o..
00000010: 64 CB BD AF EA 55 7D BD  8F 55 36 B0 01 C2 8B 2E  d....U}..U6.....
00000020: 32 B6 5D 45 F1 74 5D 38  12 0B AD 9D 2C 03 9C 22  2.]E.t]8....,.."
00000030: 46 68 EB 2E A2 8C 20 95  A8 2E 6C A8 E0 6D 47 F2  Fh.... ...l..mG.
00000040: D3 1E D7 01 F8 15 5C AD  DC 05 70 C0 93 B2 6D 74  ......:.p...mt
00000050: B0 9B 95 E6 4D 8C D2 FC  73 3E CD 0F 30 68 79 A5  ....M...s>..0hy.
00000060: B9 35 F2 41 3F 52 AD AD  32 A0 99 1A 18 3D CC 57  .5.A?R..2....=.W
00000070: 7E 39 DA 47 53 1E 67 15  AB 01 70 7F F2 47 96 71  ~9.GS.g...p..G.q
00000080: 44 23 CE 7B 60 67 82 81  80 3C 52 D2 06 89 28 92  D#.{‘g...<R...(.
00000090: 2C AB E6 3C 4E E6 DF 0E  D2 29 F1 01 BE 36 C4 F8  ,..<N....)...6..
000000A0: 54 40 56 F3 4A FA 8D 2E  9B 60 F5 07 BC ED B4 44  T@V.J....‘.....D
000000B0: 56 68 5D 82 4C C4 EA D7  96 20 F8 C5 46 A6 E0 16  Vh].L.... ..F...
```

```
000000C0: B8 AB A5 D8 43 29 58 53  77 17 09 97 AA 70 68 33  ....C)XSw....ph3
000000D0: 9E F1 41 0A 5F 39 D9 75  24 7F 3A 53 63 61 47 87  ..A._9.u$.:ScaG.
000000E0: 87 7F 88 96 BC BB 83 A1  CB D1 42 E0 EB 99 CF 34  ..........B....4
000000F0: 0E CA 56 4F 2C 57 50 6E  7B 1A FC 1F 90 7A E0 C2  ..VO,WPn{....z..
00000100: 61 09                                             a.
response changed!

----- APDU command/response pair 25 -----

(Inter-Industry) Get Remaining Bytes
COMMAND from API
00000000: 00 C0 00 00 09                                    .....

Do you want to to alter command? (y/N)

RESPONSE
00000000: A8 5D D3 30 E3 5C A9 00  39 90 00                 .].0....9..

Do you want to alter the response? (y/N)

----- APDU command/response pair 26 -----

(Proprietary) Open Secure Messaging
COMMAND from API
00000000: 80 86 00 00 80 00 00 00  00 00 00 00 00 00 00 00  ................
00000010: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000030: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000040: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000060: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000070: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000080: 00 00 00 00 00 00                                 ......

Do you want to to alter command? (y/N)

RESPONSE
00000000: 7E 4B 40 E6 E3 B1 5D 25  2B 02 48 50 B3 63 CC 9E  ~K@...]%+.HP.c..
00000010: 79 41 34 FC 04 B3 57 1C  06 E3 D1 36 3C 24 45 8D  yA4...W....6<$E.
00000020: 90 00                                             ..

Do you want to alter the response? (y/N)

----- APDU command/response pair 27 -----

(Proprietary) Get Challenge [SM]
COMMAND from API
00000000: 0C 84 00 00 0D 97 01 20  8E 08 99 BD 52 69 31 DD  ....... ....Ri1.
00000010: DB FD 00                                          ...
```

```
Do you want to to alter command? (y/N)

RESPONSE
00000000: 69 88                                             i.

Do you want to alter the response? (y/N)

----- APDU command/response pair 28 -----
```

(Proprietary) Close Secure Messaging

```
COMMAND from API
00000000: 80 86 FF FF                                       ....
```

A.3.4 First 128 bytes set to zero and generator 0

```
----- APDU command/response pair 26 -----
```

(Proprietary) Get Card Public Key

```
COMMAND from API
00000000: 80 48 00 80 00                                    .H...

Do you want to to alter command? (y/N)

RESPONSE
00000000: 80 01 05 81 81 80 F7 B5  15 72 07 22 94 6F C4 08  .........r.".o..
00000010: 64 CB BD AF EA 55 7D BD  8F 55 36 B0 01 C2 8B 2E  d....U}..U6.....
00000020: 32 B6 5D 45 F1 74 5D 38  12 0B AD 9D 2C 03 9C 22  2.]E.t]8....,.."
00000030: 46 68 EB 2E A2 8C 20 95  A8 2E 6C A8 E0 6D 47 F2  Fh.... ...l..mG.
00000040: D3 1E D7 01 F8 15 5C AD  DC 05 70 C0 93 B2 6D 74  ......:.p...mt
00000050: B0 9B 95 E6 4D 8C D2 FC  73 3E CD 0F 30 68 79 A5  ....M...s>..0hy.
00000060: B9 35 F2 41 3F 52 AD AD  32 A0 99 1A 18 3D CC 57  .5.A?R..2....=.W
00000070: 7E 39 DA 47 53 1E 67 15  AB 01 70 7F F2 47 96 71  ~9.GS.g...p..G.q
00000080: 44 23 CE 7B 60 67 82 81  80 3C 52 D2 06 89 28 92  D#.{‘g...<R...(.
00000090: 2C AB E6 3C 4E E6 DF 0E  D2 29 F1 01 BE 36 C4 F8  ,..<N....)...6..
000000A0: 54 40 56 F3 4A FA 8D 2E  9B 60 F5 07 BC ED B4 44  T@V.J....‘.....D
000000B0: 56 68 5D 82 4C C4 EA D7  96 20 F8 C5 46 A6 E0 16  Vh].L.... ..F...
000000C0: B8 AB A5 D8 43 29 58 53  77 17 09 97 AA 70 68 33  ....C)XSw....ph3
000000D0: 9E F1 41 0A 5F 39 D9 75  24 7F 3A 53 63 61 47 87  ..A._9.u$.:ScaG.
000000E0: 87 7F 88 96 BC BB 83 A1  CB D1 42 E0 EB 99 CF 34  ..........B....4
000000F0: 0E CA 56 4F 2C 57 50 6E  7B 1A FC 1F 90 7A E0 C2  ..VO,WPn{....z..
00000100: 61 09                                             a.

Do you want to alter the response? (y/N)
y

00000000: 80 01 00 81 81 80 00 00  00 00 00 00 00 00 00 00  ................
00000010: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

```
00000020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000030: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000040: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000060: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000070: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000080: 00 00 00 00 00 00 82 81  80 3C 52 D2 06 89 28 92  .........<R...(.
00000090: 2C AB E6 3C 4E E6 DF 0E  D2 29 F1 01 BE 36 C4 F8  ,..<N....)...6..
000000A0: 54 40 56 F3 4A FA 8D 2E  9B 60 F5 07 BC ED B4 44  T@V.J....‘.....D
000000B0: 56 68 5D 82 4C C4 EA D7  96 20 F8 C5 46 A6 E0 16  Vh].L.... ..F...
000000C0: B8 AB A5 D8 43 29 58 53  77 17 09 97 AA 70 68 33  ....C)XSw....ph3
000000D0: 9E F1 41 0A 5F 39 D9 75  24 7F 3A 53 63 61 47 87  ..A._9.u$.:ScaG.
000000E0: 87 7F 88 96 BC BB 83 A1  CB D1 42 E0 EB 99 CF 34  ..........B....4
000000F0: 0E CA 56 4F 2C 57 50 6E  7B 1A FC 1F 90 7A E0 C2  ..VO,WPn{....z..
00000100: 61 09                                            a.
response changed!

----- APDU command/response pair 27 -----
```

(Inter-Industry) Get Remaining Bytes

```
COMMAND from API
00000000: 00 C0 00 00 09                                   .....

Do you want to to alter command? (y/N)

RESPONSE
00000000: A8 5D D3 30 E3 5C A9 00  39 90 00                .].0....9..

Do you want to alter the response? (y/N)

(Halts here)
```

A.3.5 Second 128 bytes set to zero and generator 0

```
----- APDU command/response pair 35 -----
```

(Proprietary) Get Card Public Key

```
COMMAND from API
00000000: 80 48 00 80 00                                   .H...

Do you want to to alter command? (y/N)

RESPONSE
00000000: 80 01 05 81 81 80 F7 B5  15 72 07 22 94 6F C4 08  .........r.".o..
00000010: 64 CB BD AF EA 55 7D BD  8F 55 36 B0 01 C2 8B 2E  d....U}..U6.....
00000020: 32 B6 5D 45 F1 74 5D 38  12 0B AD 9D 2C 03 9C 22  2.]E.t]8....,.."
00000030: 46 68 EB 2E A2 8C 20 95  A8 2E 6C A8 E0 6D 47 F2  Fh.... ...l..mG.
00000040: D3 1E D7 01 F8 15 5C AD  DC 05 70 C0 93 B2 6D 74  ......:.p...mt
```

```
00000050: B0 9B 95 E6 4D 8C D2 FC  73 3E CD 0F 30 68 79 A5  ....M...s>..0hy.
00000060: B9 35 F2 41 3F 52 AD AD  32 A0 99 1A 18 3D CC 57  .5.A?R..2....=.W
00000070: 7E 39 DA 47 53 1E 67 15  AB 01 70 7F F2 47 96 71  ~9.GS.g...p..G.q
00000080: 44 23 CE 7B 60 67 82 81  80 3C 52 D2 06 89 28 92  D#.{‘g...<R...(.
00000090: 2C AB E6 3C 4E E6 DF 0E  D2 29 F1 01 BE 36 C4 F8  ,..<N....)...6..
000000A0: 54 40 56 F3 4A FA 8D 2E  9B 60 F5 07 BC ED B4 44  T@V.J....‘.....D
000000B0: 56 68 5D 82 4C C4 EA D7  96 20 F8 C5 46 A6 E0 16  Vh].L.... ..F...
000000C0: B8 AB A5 D8 43 29 58 53  77 17 09 97 AA 70 68 33  ....C)XSw....ph3
000000D0: 9E F1 41 0A 5F 39 D9 75  24 7F 3A 53 63 61 47 87  ..A._9.u$.:ScaG.
000000E0: 87 7F 88 96 BC BB 83 A1  CB D1 42 E0 EB 99 CF 34  ..........B....4
000000F0: 0E CA 56 4F 2C 57 50 6E  7B 1A FC 1F 90 7A E0 C2  ..VO,WPn{....z..
00000100: 61 09                                            a.

Do you want to alter the response? (y/N)
y

00000000: 80 01 00 81 81 80 F7 B5  15 72 07 22 94 6F C4 08  .........r.".o..
00000010: 64 CB BD AF EA 55 7D BD  8F 55 36 B0 01 C2 8B 2E  d....U}..U6.....
00000020: 32 B6 5D 45 F1 74 5D 38  12 0B AD 9D 2C 03 9C 22  2.]E.t]8....,.."
00000030: 46 68 EB 2E A2 8C 20 95  A8 2E 6C A8 E0 6D 47 F2  Fh.... ...l..mG.
00000040: D3 1E D7 01 F8 15 5C AD  DC 05 70 C0 93 B2 6D 74  ......:.p...mt
00000050: B0 9B 95 E6 4D 8C D2 FC  73 3E CD 0F 30 68 79 A5  ....M...s>..0hy.
00000060: B9 35 F2 41 3F 52 AD AD  32 A0 99 1A 18 3D CC 57  .5.A?R..2....=.W
00000070: 7E 39 DA 47 53 1E 67 15  AB 01 70 7F F2 47 96 71  ~9.GS.g...p..G.q
00000080: 44 23 CE 7B 60 67 82 81  80 00 00 00 00 00 00 00  D#.{‘g..........
00000090: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000100: 61 09                                            a.
response changed!

----- APDU command/response pair 36 -----

(Inter-Industry) Get Remaining Bytes
COMMAND from API
00000000: 00 C0 00 00 09                                   .....

Do you want to to alter command? (y/N)

RESPONSE
00000000: A8 5D D3 30 E3 5C A9 00  39 90 00                .].0....9..

Do you want to alter the response? (y/N)
y
```

```
00000000: 00 00 00 00 00 00 00 00  00 90 00                 ...........
response changed!

----- APDU command/response pair 37 -----
```

(Proprietary) Open Secure Messaging

```
COMMAND from API
00000000: 80 86 00 00 80 00 00 00  00 00 00 00 00 00 00 00  ................
00000010: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000030: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000040: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000060: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000070: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000080: 00 00 00 00 00 00                                 ......

Do you want to to alter command? (y/N)

RESPONSE
00000000: 22 2C 15 03 96 1F CA 12  4B CE CA 67 FE 92 5D A1  ",......K..g..].
00000010: 71 53 96 34 CD F3 10 8E  F9 2F 6F 28 CB E0 CE 78  qS.4...../o(...x
00000020: 90 00                                             ..

Do you want to alter the response? (y/N)

----- APDU command/response pair 38 -----

COMMAND from API
00000000: 0C 84 00 00 0D 97 01 20  8E 08 DF C1 4D 9E 7B 0A  ....... ....M.{.
00000010: 24 E7 00                                          $..

Do you want to to alter command? (y/N)

RESPONSE
00000000: 87 29 01 C7 40 87 51 07  6E 7A 62 89 6C 53 46 BC  .)..@.Q.nzb.lSF.
00000010: 4E 7A 2C E0 3B A7 A8 5B  44 90 4D 62 2C FB 1C 33  Nz,.;..[D.Mb,..3
00000020: 7D 18 CF 56 F8 76 8F 4C  E9 A2 F3 99 02 90 00 8E  }..V.v.L........
00000030: 08 D1 DC BB 76 9A A7 6F  25 90 00                 ....v..o%..

Do you want to alter the response? (y/N)

----- APDU command/response pair 39 -----

COMMAND from API
00000000: 80 86 FF FF                                       ....

Do you want to to alter command? (y/N)
```

```
RESPONSE
00000000: 90 00                                            ..
```

A.3.6 Set 1st 16 bytes of card challenge to zero

```
----- APDU command/response pair 24 -----

COMMAND from API
00000000: 80 48 00 80 00                                   .H...

Do you want to to alter command? (y/N)

RESPONSE
00000000: 80 01 05 81 81 80 F7 B5  15 72 07 22 94 6F C4 08  .........r.".o..
00000010: 64 CB BD AF EA 55 7D BD  8F 55 36 B0 01 C2 8B 2E  d....U}..U6.....
00000020: 32 B6 5D 45 F1 74 5D 38  12 0B AD 9D 2C 03 9C 22  2.]E.t]8....,.."
00000030: 46 68 EB 2E A2 8C 20 95  A8 2E 6C A8 E0 6D 47 F2  Fh.... ...l..mG.
00000040: D3 1E D7 01 F8 15 5C AD  DC 05 70 C0 93 B2 6D 74  ......:.p...mt
00000050: B0 9B 95 E6 4D 8C D2 FC  73 3E CD 0F 30 68 79 A5  ....M...s>..0hy.
00000060: B9 35 F2 41 3F 52 AD AD  32 A0 99 1A 18 3D CC 57  .5.A?R..2....=.W
00000070: 7E 39 DA 47 53 1E 67 15  AB 01 70 7F F2 47 96 71  ~9.GS.g...p..G.q
00000080: 44 23 CE 7B 60 67 82 81  80 3C 52 D2 06 89 28 92  D#.{‘g...<R...(.
00000090: 2C AB E6 3C 4E E6 DF 0E  D2 29 F1 01 BE 36 C4 F8  ,..<N....)...6..
000000A0: 54 40 56 F3 4A FA 8D 2E  9B 60 F5 07 BC ED B4 44  T@V.J....‘.....D
000000B0: 56 68 5D 82 4C C4 EA D7  96 20 F8 C5 46 A6 E0 16  Vh].L.... ..F...
000000C0: B8 AB A5 D8 43 29 58 53  77 17 09 97 AA 70 68 33  ....C)XSw....ph3
000000D0: 9E F1 41 0A 5F 39 D9 75  24 7F 3A 53 63 61 47 87  ..A._9.u$.:ScaG.
000000E0: 87 7F 88 96 BC BB 83 A1  CB D1 42 E0 EB 99 CF 34  ..........B....4
000000F0: 0E CA 56 4F 2C 57 50 6E  7B 1A FC 1F 90 7A E0 C2  ..VO,WPn{....z..
00000100: 61 09                                            a.

Do you want to alter the response? (y/N)
y

00000000: 80 01 00 81 81 80 F7 B5  15 72 07 22 94 6F C4 08  .........r.".o..
00000010: 64 CB BD AF EA 55 7D BD  8F 55 36 B0 01 C2 8B 2E  d....U}..U6.....
00000020: 32 B6 5D 45 F1 74 5D 38  12 0B AD 9D 2C 03 9C 22  2.]E.t]8....,.."
00000030: 46 68 EB 2E A2 8C 20 95  A8 2E 6C A8 E0 6D 47 F2  Fh.... ...l..mG.
00000040: D3 1E D7 01 F8 15 5C AD  DC 05 70 C0 93 B2 6D 74  ......:.p...mt
00000050: B0 9B 95 E6 4D 8C D2 FC  73 3E CD 0F 30 68 79 A5  ....M...s>..0hy.
00000060: B9 35 F2 41 3F 52 AD AD  32 A0 99 1A 18 3D CC 57  .5.A?R..2....=.W
00000070: 7E 39 DA 47 53 1E 67 15  AB 01 70 7F F2 47 96 71  ~9.GS.g...p..G.q
00000080: 44 23 CE 7B 60 67 82 81  80 00 00 00 00 00 00 00  D#.{‘g.........
00000090: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

```
000000E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000100: 61 09                                             a.
response changed!

----- APDU command/response pair 25 -----

COMMAND from API
00000000: 00 C0 00 00 09                                    .....

Do you want to to alter command? (y/N)

RESPONSE
00000000: A8 5D D3 30 E3 5C A9 00  39 90 00                 .].0.:9..

Do you want to alter the response? (y/N)
y

00000000: 00 00 00 00 00 00 00 00  00 90 00                 ...........
response changed!

----- APDU command/response pair 26 -----

COMMAND from API
00000000: 80 86 00 00 80 00 00 00  00 00 00 00 00 00 00 00  ................
00000010: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000030: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000040: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000060: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000070: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000080: 00 00 00 00 00 00                                 ......

Do you want to to alter command? (y/N)

RESPONSE
00000000: C8 04 C7 25 A9 14 2D 58  E8 01 64 6D 72 DA C4 9C  ...%..-X..dmr...
00000010: A5 F0 D1 FA AF 53 93 A7  49 8D 69 8B 7A 1A D0 A4  .....S..I.i.z...
00000020: 90 00                                             ..

Do you want to alter the response? (y/N)
y

00000000: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000010: A5 F0 D1 FA AF 53 93 A7  49 8D 69 8B 7A 1A D0 A4  .....S..I.i.z...
00000020: 90 00                                             ..
response changed!
```

```
----- APDU command/response pair 27 -----

COMMAND from API
00000000: 0C 84 00 00 0D 97 01 18  8E 08 1C 0B 23 9F 34 B7  ............#.4.
00000010: 29 FD 00                                          )..

Do you want to to alter command? (y/N)

RESPONSE
00000000: 87 21 01 5D 5B F0 E6 13  1F 81 42 0D 4E B8 39 A9  .!.][.....B.N.9.
00000010: 66 C4 D8 80 E2 BB D8 1F  32 08 35 CF 59 EF 5A 38  f.......2.5.Y.Z8
00000020: 42 65 51 99 02 90 00 8E  08 BE 24 CF FA 69 84 EE  BeQ.......$..i..
00000030: BB 90 00                                          ...

Do you want to alter the response? (y/N)

----- APDU command/response pair 28 -----

COMMAND from API
00000000: 80 86 FF FF                                       ....
```

A.3.7 Set 2nd 16 bytes of card challenge to zero

```
----- APDU command/response pair 24 -----

COMMAND from API
00000000: 80 48 00 80 00                                    .H...

Do you want to to alter command? (y/N)

RESPONSE
00000000: 80 01 05 81 81 80 F7 B5  15 72 07 22 94 6F C4 08  .........r.".o..
00000010: 64 CB BD AF EA 55 7D BD  8F 55 36 B0 01 C2 8B 2E  d....U}..U6.....
00000020: 32 B6 5D 45 F1 74 5D 38  12 0B AD 9D 2C 03 9C 22  2.]E.t]8....,.."
00000030: 46 68 EB 2E A2 8C 20 95  A8 2E 6C A8 E0 6D 47 F2  Fh.... ...l..mG.
00000040: D3 1E D7 01 F8 15 5C AD  DC 05 70 C0 93 B2 6D 74  ......:.p...mt
00000050: B0 9B 95 E6 4D 8C D2 FC  73 3E CD 0F 30 68 79 A5  ....M...s>..0hy.
00000060: B9 35 F2 41 3F 52 AD AD  32 A0 99 1A 18 3D CC 57  .5.A?R..2....=.W
00000070: 7E 39 DA 47 53 1E 67 15  AB 01 70 7F F2 47 96 71  ~9.GS.g...p..G.q
00000080: 44 23 CE 7B 60 67 82 81  80 3C 52 D2 06 89 28 92  D#.{‘g...<R...(.
00000090: 2C AB E6 3C 4E E6 DF 0E  D2 29 F1 01 BE 36 C4 F8  ,..<N....)...6..
000000A0: 54 40 56 F3 4A FA 8D 2E  9B 60 F5 07 BC ED B4 44  T@V.J....‘.....D
000000B0: 56 68 5D 82 4C C4 EA D7  96 20 F8 C5 46 A6 E0 16  Vh].L.... ..F...
000000C0: B8 AB A5 D8 43 29 58 53  77 17 09 97 AA 70 68 33  ....C)XSw....ph3
000000D0: 9E F1 41 0A 5F 39 D9 75  24 7F 3A 53 63 61 47 87  ..A._9.u$.:ScaG.
000000E0: 87 7F 88 96 BC BB 83 A1  CB D1 42 E0 EB 99 CF 34  ..........B....4
000000F0: 0E CA 56 4F 2C 57 50 6E  7B 1A FC 1F 90 7A E0 C2  ..VO,WPn{....z..
00000100: 61 09                                             a.
```

```
Do you want to alter the response? (y/N)
y

00000000: 80 01 00 81 81 80 F7 B5  15 72 07 22 94 6F C4 08  .........r.".o..
00000010: 64 CB BD AF EA 55 7D BD  8F 55 36 B0 01 C2 8B 2E  d....U}..U6.....
00000020: 32 B6 5D 45 F1 74 5D 38  12 0B AD 9D 2C 03 9C 22  2.]E.t]8....,.."
00000030: 46 68 EB 2E A2 8C 20 95  A8 2E 6C A8 E0 6D 47 F2  Fh.... ...l..mG.
00000040: D3 1E D7 01 F8 15 5C AD  DC 05 70 C0 93 B2 6D 74  ......:.p...mt
00000050: B0 9B 95 E6 4D 8C D2 FC  73 3E CD 0F 30 68 79 A5  ....M...s>..0hy.
00000060: B9 35 F2 41 3F 52 AD AD  32 A0 99 1A 18 3D CC 57  .5.A?R..2....=.W
00000070: 7E 39 DA 47 53 1E 67 15  AB 01 70 7F F2 47 96 71  ~9.GS.g...p..G.q
00000080: 44 23 CE 7B 60 67 82 81  80 00 00 00 00 00 00 00  D#.{‘g.........
00000090: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000100: 61 09                                             a.
response changed!

----- APDU command/response pair 25 -----

COMMAND from API
00000000: 00 C0 00 00 09                                    .....

Do you want to to alter command? (y/N)

RESPONSE
00000000: A8 5D D3 30 E3 5C A9 00  39 90 00                 .].0.:9..

Do you want to alter the response? (y/N)
y

00000000: 00 00 00 00 00 00 00 00  00 90 00                 ...........
response changed!

----- APDU command/response pair 26 -----

COMMAND from API
00000000: 80 86 00 00 80 00 00 00  00 00 00 00 00 00 00 00  ................
00000010: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000030: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000040: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000060: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

```
00000070: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000080: 00 00 00 00 00 00                                 ......

Do you want to to alter command? (y/N)

RESPONSE
00000000: FA AD 82 BB C2 95 69 6E  2C 69 DF B2 75 90 DF BD  ......in,i..u...
00000010: F7 FA 17 55 24 4A 1B CD  7B 1A 1D 92 A3 74 9F 98  ...U$J..{....t..
00000020: 90 00                                             ..

Do you want to alter the response? (y/N)
y

00000000: FA AD 82 BB C2 95 69 6E  2C 69 DF B2 75 90 DF BD  ......in,i..u...
00000010: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000020: 90 00                                             ..
response changed!

----- APDU command/response pair 27 -----

COMMAND from API
00000000: 0C 84 00 00 0D 97 01 18  8E 08 E4 8B 06 AD EF A3  ................
00000010: DF E6 00                                          ...

Do you want to to alter command? (y/N)

RESPONSE
00000000: 69 88                                             i.

Do you want to alter the response? (y/N)

----- APDU command/response pair 28 -----

COMMAND from API
00000000: 80 86 FF FF                                       ....
```

A.4 Overriding Attribute Controls

A.4.1 Encrypt_False

```
Do you want to to alter command? (y/N)
y
Enter command (spaced integers)
128 164 8 0 8 63 0 48 0 48 1 0 193                 (integers)
80 A4 08 00 08 3F 00 30 00 30 01 00 C1        (hexadecimal)
command changed!
```

```
RESPONSE
00000000: 90 00                                            ..

Do you want to alter the response? (y/N)

----- APDU command/response pair 49 -----

COMMAND from API
00000000: 80 A4 08 00 08 3F 00 30  00 30 01 D0 7E           .....?.0.0..~

Do you want to to alter command? (y/N)
y
Enter command (spaced integers)
0 42 130 5 19 128 129 16 84 101 115 116 83 116 114 105 110 103 49 50 51 52 53 54 0 (ints)
00 2A 82 05 13 80 81 10 54 65 73 74 53 74 72 69 6E 67 31 32 33 34 35 36 00 (hexadecimals)
command changed!

RESPONSE
00000000: 82 10 8B 4C 51 82 70 4A  1F 9D 79 A8 68 3D 23 8C  ...LQ.pJ..y.h=#.
00000010: E8 BE 90 00                                      ....
```

Appendix B

API Function Traces

B.1 Initialization

```
----- APDU command/response pair 1 -----
00000000: 00 A4 04 00 0C A0 00 00  01 64 4C 41 53 45 52 00  .........dLASER.
00000010: 01 00                                             ..

00000000: 90 00                                             ..

----- APDU command/response pair 4 -----
00000000: 80 A4 08 00 06 3F 00 30  00 C0 00                 .....?.0...

00000000: 90 00                                             ..


----- APDU command/response pair 5 -----
00000000: 00 B0 00 00 00                                    .....

00000000: 49 44 50 72 6F 74 65 63  74 20 20 20 20 20 20 20  IDProtect
00000010: 20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
00000020: 41 74 68 65 6E 61 20 53  6D 61 72 74 63 61 72 64  Athena Smartcard
00000030: 20 53 6F 6C 75 74 69 6F  6E 73 20 20 20 20 20 20   Solutions
00000040: 49 44 50 72 6F 74 65 63  74 20 20 20 20 20 20 20  IDProtect
00000050: 30 44 35 30 30 30 30 39  32 31 32 32 38 37 39 36  0D50000921228796
00000060: 0D 04 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000070: 00 00 00 00 10 00 00 00  04 00 00 00 FF FF FF FF  ................
00000080: 00 00 00 00 FF FF FF FF  00 00 00 00 01 00 01 00  ................
00000090: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000A0: 00 90 00                                          ...


----- APDU command/response pair 8 -----
00000000: 80 A4 08 00 08 3F 00 30  00 30 03 40 00           .....?.0.0.@.
```

```
00000000: 90 00                                             ..


----- APDU command/response pair 9 -----
00000000: 00 B0 00 02 64                                    ....d

00000000: 41 54 48 45 4E 41 53 4E  C0 AD AA 78 FC 88 42 0D  ATHENASN...x..B.
00000010: 90 00                                             ..
```

B.2 C_login

```
----- APDU command/response pair 10 -----
00000000: 80 A4 08 0C 04 3F 00 00  20 00                    .....?.. .

00000000: 62 2F 87 01 08 83 02 00  20 80 02 00 10 8A 01 04  b/...... .......
00000010: 86 0E 00 FF C0 30 00 FF  00 10 00 FF 00 10 00 00  .....0..........
00000020: 85 0F 00 01 00 00 AA 00  04 10 00 00 00 00 00 FF  ................
00000030: FF 90 00                                          ...

----- APDU command/response pair 11 -----
00000000: 80 A4 08 00 04 3F 00 00  20                       .....?..

00000000: 90 00                                             ..

----- APDU command/response pair 12 -----
00000000: 00 84 00 00 08                                    .....

00000000: 11 B7 B2 80 4B 17 0D A4  90 00                    ....K.....


----- APDU command/response pair 13 -----
00000000: 80 20 00 00 10 1D ED 9E  47 A8 C9 EA CE 37 82 2C  . ......G....7.,
00000010: 92 CF 07 20 2D                                    ... -

00000000: 90 00                                             ..


----- APDU command/response pair 20 -----
00000000: 80 28 00 00 04 00 00 00  20                       .(......

00000000: 90 00                                             ..
```

B.3 C_findObject

```
----- APDU command/response pair 32 -----
00000000: 80 30 01 00 00                                    .0...

00000000: D1 02 00 03 D2 02 03 40  D2 02 03 46 D2 0A 86 7F  .......@...F....
00000010: 63 6D 61 70 66 69 6C 65  90 00                    cmapfile..


----- APDU command/response pair 33 -----
00000000: 80 A4 08 00 06 3F 00 30  00 30 02                 .....?.0.0.

00000000: 90 00                                             ..


----- APDU command/response pair 34 -----
00000000: 80 30 01 00 00                                    .0...

00000000: D1 02 00 00 90 00                                 ......


----- APDU command/response pair 35 -----
00000000: 80 A4 08 00 08 3F 00 30  00 30 01 03 40           .....?.0.0..@

00000000: 90 00                                             ..


----- APDU command/response pair 36 -----
00000000: 00 B0 00 00 00                                    .....

00000000: 00 03 03 40 01 23 18 00  00 00 00 04 04 00 00 00  ...@.#..........
00000010: 00 01 00 00 01 01 00 02  00 00 01 00 00 03 10 00  ................
00000020: 04 64 65 73 33 FF FF FF  FF FF FF FF FF FF FF FF  .des3...........
00000030: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000040: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000050: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000060: FF 00 11 01 00 18 FF FF  FF FF FF FF FF FF FF FF  ................
00000070: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF 01 00  ................
00000080: 00 00 04 15 00 00 00 01  02 10 00 01 01 FF FF FF  ................
00000090: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000000A0: FF FF FF FF FF FF FF FF  FF FF FF FF 01 03 30 00  ..............0.
000000B0: 01 01 01 04 00 00 01 01  01 05 50 00 01 01 01 06  ..........P.....
000000C0: 00 00 01 00 01 07 50 00  01 01 01 08 50 00 01 01  ......P.....P...
000000D0: 01 0A 00 00 01 01 01 0C  10 00 01 00 01 10 10 00  ................
000000E0: 00 FF FF FF FF FF FF FF  FF 01 11 10 00 00 FF FF  ................
000000F0: FF FF FF FF FF FF 01 62  50 00 01 00 01 63 00 00  .......bP....c..
00000100: 61 27                                             a’
```

```
----- APDU command/response pair 37 -----
00000000: 00 B0 01 00 00                                     .....

00000000: 01 01 01 64 00 00 01 01  01 65 00 00 01 01 01 66  ...d.....e.....f
00000010: 00 00 04 31 01 00 00 01  70 00 00 01 01 80 10 00  ...1....p.......
00000020: 00 01 00 99 03 99 03 90  00                        .........


----- APDU command/response pair 38 -----
00000000: 80 A4 08 00 08 3F 00 30  00 30 01 03 46           .....?.0.0..F

00000000: 90 00

----- APDU command/response pair 39 -----
00000000: 00 B0 00 00 00                                     .....

00000000: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000010: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000030: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000040: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000060: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000070: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000080: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000090: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000100: 61 2F                                              a/


----- APDU command/response pair 40 -----
00000000: 00 B0 01 00 00                                     .....

00000000: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000010: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 90  ................
00000030: 00                                                 .
```

B.4 C_generateKey

```
----- APDU command/response pair 26 -----
00000000: 80 48 00 80 00                                     .H...
```

```
00000000: 80 01 05 81 81 80 F7 B5  15 72 07 22 94 6F C4 08  .........r.".o..
00000010: 64 CB BD AF EA 55 7D BD  8F 55 36 B0 01 C2 8B 2E  d....U}..U6.....
00000020: 32 B6 5D 45 F1 74 5D 38  12 0B AD 9D 2C 03 9C 22  2.]E.t]8....,.."
00000030: 46 68 EB 2E A2 8C 20 95  A8 2E 6C A8 E0 6D 47 F2  Fh.... ...l..mG.
00000040: D3 1E D7 01 F8 15 5C AD  DC 05 70 C0 93 B2 6D 74  ..........p...mt
00000050: B0 9B 95 E6 4D 8C D2 FC  73 3E CD 0F 30 68 79 A5  ....M...s>..0hy.
00000060: B9 35 F2 41 3F 52 AD AD  32 A0 99 1A 18 3D CC 57  .5.A?R..2....=.W
00000070: 7E 39 DA 47 53 1E 67 15  AB 01 70 7F F2 47 96 71  ~9.GS.g...p..G.q
00000080: 44 23 CE 7B 60 67 82 81  80 3C 52 D2 06 89 28 92  D#.{‘g...<R...(.
00000090: 2C AB E6 3C 4E E6 DF 0E  D2 29 F1 01 BE 36 C4 F8  ,..<N....)...6..
000000A0: 54 40 56 F3 4A FA 8D 2E  9B 60 F5 07 BC ED B4 44  T@V.J....‘.....D
000000B0: 56 68 5D 82 4C C4 EA D7  96 20 F8 C5 46 A6 E0 16  Vh].L.... ..F...
000000C0: B8 AB A5 D8 43 29 58 53  77 17 09 97 AA 70 68 33  ....C)XSw....ph3
000000D0: 9E F1 41 0A 5F 39 D9 75  24 7F 3A 53 63 61 47 87  ..A._9.u$.:ScaG.
000000E0: 87 7F 88 96 BC BB 83 A1  CB D1 42 E0 EB 99 CF 34  ..........B....4
000000F0: 0E CA 56 4F 2C 57 50 6E  7B 1A FC 1F 90 7A E0 C2  ..VO,WPn{....z..
00000100: 61 09                                             a.


----- APDU command/response pair 27 -----
00000000: 00 C0 00 00 09                                    .....

00000000: A8 5D D3 30 E3 5C A9 00  39 90 00                 .].0....9..


----- APDU command/response pair 28 -----
00000000: 80 86 00 00 80 84 7F A0  E7 6C 8F AA 50 9C C3 6E  .........l..P..n
00000010: 82 5E 84 B6 E4 F6 77 1C  45 FA AB 06 1B 24 C4 A8  .^....w.E....$..
00000020: 92 03 A9 9C A8 2B BE 1B  28 C4 57 83 A5 5E BB 8D  .....+..(.W..^..
00000030: D2 BF 3F D5 02 8A 7C 13  10 9C 75 06 91 1A 0F 05  ..?...|...u.....
00000040: 55 B4 C9 12 8A 69 59 B6  07 1D 67 F2 8A C9 FA BC  U....iY...g.....
00000050: F3 BE 16 73 51 C0 76 0C  11 E5 0C D3 8C FE 09 E5  ...sQ.v.........
00000060: 1E 52 DE 38 D9 AC 2D EB  C6 A1 C4 8E ED 03 7D 07  .R.8..-.......}.
00000070: 85 B7 FE 66 82 2F 03 65  94 DC 27 77 2B 3A 28 71  ...f./.e..’w+:(q
00000080: 97 08 5D 03 80 00                                 ..]...

00000000: F9 D0 66 F7 48 CB BB E8  CE 93 60 05 99 1B 81 2E  ..f.H.....‘.....
00000010: 73 0B B7 B8 DC 10 A7 84  B3 99 D8 C8 60 D6 48 5A  s...........‘.HZ
00000020: 90 00                                             ..


----- APDU command/response pair 29 -----
00000000: 0C 84 00 00 0D 97 01 18  8E 08 2B 88 7C 0C 8C 24  ..........+.|..$
00000010: 00 1F 00                                          ...

00000000: 87 21 01 69 AB B7 01 F5  F5 8E EA B8 F3 09 D7 5E  .!.i...........^
00000010: F5 26 3C 7F 1D 15 90 B8  40 D4 A1 85 9C 57 3F 27  .&<.....@....W?’
00000020: 87 84 C6 99 02 90 00 8E  08 42 84 88 19 99 3B C2  .........B....;.
```

```
00000030: 10 90 00                                         ...


----- APDU command/response pair 30 -----
00000000: 80 86 FF FF                                      ....

00000000: 90 00                                            ..


----- APDU command/response pair 39 -----
00000000: 80 A4 08 00 08 3F 00 30  00 30 01 03 40          .....?.0.0..@

00000000: 90 00                                            ..


----- APDU command/response pair 40 -----
00000000: 00 D6 00 00 FA 01 03 03  40 01 23 18 00 00 00 00  ........@.#.....
00000010: 04 04 00 00 00 00 01 00  00 01 01 00 02 00 00 01  ................
00000020: 00 00 03 10 00 04 64 65  73 33 FF FF FF FF FF FF  ......des3......
00000030: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000040: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000050: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000060: FF FF FF FF FF FF 00 11  01 00 18 FF FF FF FF FF  ................
00000070: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000080: FF FF FF 01 00 00 00 04  15 00 00 00 01 02 10 00  ................
00000090: 01 01 FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000000A0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000000B0: FF 01 03 30 00 01 01 01  04 00 00 01 01 01 05 50  ...0...........P
000000C0: 00 01 01 01 06 00 00 01  00 01 07 50 00 01 01 01  ...........P....
000000D0: 08 50 00 01 01 01 0A 00  00 01 01 01 0C 10 00 01  .P..............
000000E0: 00 01 10 10 00 00 FF FF  FF FF FF FF FF FF 01 11  ................
000000F0: 10 00 00 FF FF FF FF FF  FF FF FF 01 62 50 00     ............bP.

00000000: 90 00                                            ..


----- APDU command/response pair 41 -----
00000000: 00 D6 00 FA 2D 01 00 01  63 00 00 01 01 01 64 00  ....-...c.....d.
00000010: 00 01 01 01 65 00 00 01  01 01 66 00 00 04 31 01  ....e.....f...1.
00000020: 00 00 01 70 00 00 01 01  80 10 00 00 01 00 88 03  ...p............
00000030: 88 03                                            ..

00000000: 90 00                                            ..


----- APDU command/response pair 42 -----
00000000: 80 48 00 80 00                                   .H...

00000000: 80 01 05 81 81 80 F7 B5  15 72 07 22 94 6F C4 08  .........r.".o..
```

```
00000010: 64 CB BD AF EA 55 7D BD  8F 55 36 B0 01 C2 8B 2E  d....U}..U6.....
00000020: 32 B6 5D 45 F1 74 5D 38  12 0B AD 9D 2C 03 9C 22  2.]E.t]8....,.."
00000030: 46 68 EB 2E A2 8C 20 95  A8 2E 6C A8 E0 6D 47 F2  Fh.... ...l..mG.
00000040: D3 1E D7 01 F8 15 5C AD  DC 05 70 C0 93 B2 6D 74  ......:.p...mt
00000050: B0 9B 95 E6 4D 8C D2 FC  73 3E CD 0F 30 68 79 A5  ....M...s>..0hy.
00000060: B9 35 F2 41 3F 52 AD AD  32 A0 99 1A 18 3D CC 57  .5.A?R..2....=.W
00000070: 7E 39 DA 47 53 1E 67 15  AB 01 70 7F F2 47 96 71  ~9.GS.g...p..G.q
00000080: 44 23 CE 7B 60 67 82 81  80 3C 52 D2 06 89 28 92  D#.{‘g...<R...(.
00000090: 2C AB E6 3C 4E E6 DF 0E  D2 29 F1 01 BE 36 C4 F8  ,..<N....)...6..
000000A0: 54 40 56 F3 4A FA 8D 2E  9B 60 F5 07 BC ED B4 44  T@V.J....‘.....D
000000B0: 56 68 5D 82 4C C4 EA D7  96 20 F8 C5 46 A6 E0 16  Vh].L.... ..F...
000000C0: B8 AB A5 D8 43 29 58 53  77 17 09 97 AA 70 68 33  ....C)XSw....ph3
000000D0: 9E F1 41 0A 5F 39 D9 75  24 7F 3A 53 63 61 47 87  ..A._9.u$.:ScaG.
000000E0: 87 7F 88 96 BC BB 83 A1  CB D1 42 E0 EB 99 CF 34  ..........B....4
000000F0: 0E CA 56 4F 2C 57 50 6E  7B 1A FC 1F 90 7A E0 C2  ..VO,WPn{....z..
00000100: 61 09                                            a.


----- APDU command/response pair 43 -----
00000000: 00 C0 00 00 09                                   .....

00000000: A8 5D D3 30 E3 5C A9 00  39 90 00                .].0.:9..


----- APDU command/response pair 44 -----
00000000: 80 86 00 00 80 C3 88 FD  AF 64 0D 35 77 85 D4 20  .........d.5w..
00000010: 57 10 02 F4 1E 38 51 37  40 31 7F 7F 11 E8 4B 8D  W....8Q7@1....K.
00000020: A5 CE C0 50 EB 6B CE E6  E0 DE E8 34 7C FE 0B 6C  ...P.k.....4|..l
00000030: F0 70 9F E3 5D F7 AA 50  BB 1C F6 8C 00 1B 18 EA  .p..]..P........
00000040: BF 73 E4 BE 75 B6 AE 29  B1 A2 A3 B8 1D 52 FD 19  .s..u..).....R..
00000050: C9 CA 20 FB 80 C2 20 A9  E3 A6 15 6C 11 B3 E9 18  .. ... ....l....
00000060: 13 3F 65 02 28 21 74 72  29 EA E2 27 8B DA 3E 45  .?e.(!tr)..’..>E
00000070: 82 A1 B0 D9 A7 1A 3D F3  5D 4D 27 F4 D2 73 ED 0F  ......=.]M’..s..
00000080: A8 88 41 F2 4F 00                                ..A.O.

00000000: 14 8C 30 9E D5 10 25 B1  F7 AF 07 E7 25 8B 22 3C  ..0...%.....%."<
00000010: 62 61 8F 24 FB 59 E1 63  D7 B1 08 6D 07 7A DD 93  ba.$.Y.c...m.z..
00000020: 90 00                                            ..


----- APDU command/response pair 45 -----
00000000: 8C A4 08 00 15 87 09 01  E5 61 A8 BF 89 AD D7 FF  .........a......
00000010: 8E 08 C2 B3 32 7B D7 83  C9 D1                    ....2{....

00000000: 99 02 90 00 8E 08 E6 37  E6 BE 12 F8 73 6F 90 00  .......7....so..


----- APDU command/response pair 46 -----
00000000: 0C E0 08 00 4D 87 41 01  41 03 69 5A A4 EE 5F 44  ....M.A.A.iZ.._D
```

```
00000010: 2C 4C A9 FE 46 8D 1F 5B  79 D6 89 68 EB 94 CF FB  ,L..F..[y..h....
00000020: 6B A2 55 F6 65 B7 19 66  B3 67 E0 DF 46 F2 27 22  k.U.e..f.g..F.'"
00000030: AC D8 C1 57 C5 54 5B DF  B9 10 87 58 81 2E 9E 65  ...W.T[....X...e
00000040: 07 B1 6E 14 F8 DE 09 AF  8E 08 8C 79 AD C4 3B E2  ..n........y..;.
00000050: D2 84                                             ..


00000000: 99 02 90 00 8E 08 A5 D0  49 2A C0 91 47 68 90 00  ........I*..Gh..

----- APDU command/response pair 47 -----
00000000: 80 86 FF FF                                       ....

00000000: 90 00                                             ..
```

B.5 C_generateKeyPair

```
----- APDU command/response pair 54 -----
00000000: 00 E0 01 00 18 62 81 15  8A 01 04 83 02 01 40 80  .....b........@.
00000010: 02 01 A7 86 08 00 20 00  20 00 20 00 20           ...... . . .

00000000: 90 00                                             ..


----- APDU command/response pair 55 -----
00000000: 80 A4 08 00 08 3F 00 30  00 30 02 01 40           .....?.0.0..@

00000000: 90 00                                             ..


----- APDU command/response pair 56 -----
00000000: 00 D6 00 00 FA 01 01 01  40 01 A3 16 00 00 00 00  ........@.......
00000010: 04 02 00 00 00 00 01 00  00 01 01 00 02 00 00 01  ................
00000020: 01 00 03 10 00 03 70 75  62 FF FF FF FF FF FF FF  ......pub.......
00000030: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000040: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000050: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000060: FF FF FF FF FF FF 00 86  32 00 01 00 01 00 00 00  ........2.......
00000070: 04 00 00 00 00 01 01 10  00 00 FF FF FF FF FF FF  ................
00000080: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000090: FF FF FF FF FF FF FF FF  FF FF 01 02 10 00 01 03  ................
000000A0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000000B0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF 01  ................
000000C0: 04 00 00 01 01 01 06 00  00 01 01 01 0A 00 00 01  ................
000000D0: 01 01 0B 00 00 01 00 01  0C 10 00 01 00 01 10 10  ................
000000E0: 00 00 FF FF FF FF FF FF  FF FF 01 11 10 00 00 FF  ................
000000F0: FF FF FF FF FF FF FF 01  20 00 00 80 A8 FD 0C     ........ ......
```

```
00000000: 90 00                                             ..


----- APDU command/response pair 57 -----
00000000: 00 D6 00 FA AD 53 6B 7F  00 00 A8 FD 0C 53 6B 7F  .....Sk......Sk.
00000010: 00 00 B0 AA 47 51 6B 7F  00 00 00 30 00 00 00 00  ....GQk....0....
00000020: 00 00 B0 AA 47 51 6B 7F  00 00 02 30 00 00 00 00  ....GQk....0....
00000030: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000040: 00 00 00 00 00 00 00 00  00 00 11 01 00 00 00 00  ................
00000050: 00 00 58 FD 0C 53 6B 7F  00 00 58 FD 0C 53 6B 7F  ..X..Sk...X..Sk.
00000060: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000070: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000080: 00 00 01 21 00 00 04 00  04 00 00 01 22 00 00 03  ...!........"...
00000090: 01 00 01 01 63 00 00 01  01 01 66 00 00 04 00 00  ....c.....f.....
000000A0: 00 00 01 70 00 00 01 01  80 10 00 00 01 00 93 03  ...p............
000000B0: 93 03                                             ..

00000000: 90 00                                             ..


----- APDU command/response pair 58 -----
00000000: 80 A4 08 00 06 3F 00 30  00 30 02                 .....?.0.0.

00000000: 90 00                                             ..


----- APDU command/response pair 59 -----
00000000: 00 E0 01 00 1E 62 81 1B  8A 01 04 83 02 02 00 80  .....b..........
00000010: 02 01 23 84 04 6B 78 73  30 86 08 00 00 00 20 00  ..#..kxs0..... .
00000020: 20 00 20                                           . 

00000000: 90 00                                             ..


----- APDU command/response pair 60 -----
00000000: 80 A4 08 00 08 3F 00 30  00 30 02 02 00           .....?.0.0...

00000000: 90 00                                             ..


----- APDU command/response pair 61 -----
00000000: 00 D6 00 00 FA 01 02 02  00 01 1F 16 00 00 00 00  ................
00000010: 04 03 00 00 00 00 01 00  00 01 01 00 02 00 00 01  ................
00000020: 01 00 03 10 00 04 70 72  69 76 FF FF FF FF FF FF  ......priv......
00000030: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000040: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000050: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000060: FF FF FF FF FF FF 01 00  00 00 04 00 00 00 00 01  ................
00000070: 01 10 00 00 FF FF FF FF  FF FF FF FF FF FF FF FF  ................
```

```
00000080: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000090: FF FF FF FF 01 02 10 00  01 03 FF FF FF FF FF FF  ................
000000A0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000000B0: FF FF FF FF FF FF FF FF  FF 01 03 30 00 01 01 01  ...........0....
000000C0: 05 50 00 01 01 01 07 50  00 01 01 01 08 50 00 01  .P.....P.....P..
000000D0: 01 01 09 50 00 01 00 01  0C 10 00 01 00 01 10 10  ...P............
000000E0: 00 00 FF FF FF FF FF FF  FF FF 01 11 10 00 00 FF  ................
000000F0: FF FF FF FF FF FF FF 01  62 50 00 01 01 01 63     ........bP....c

00000000: 90 00                                             ..


----- APDU command/response pair 62 -----
00000000: 00 D6 00 FA 29 00 00 01  01 01 64 00 00 01 00 01  ....).....d.....
00000010: 65 00 00 01 01 01 66 00  00 04 00 00 00 00 01 70  e.....f........p
00000020: 00 00 01 01 80 10 00 00  01 00 93 03 93 03        ..............

00000000: 90 00                                             ..


----- APDU command/response pair 63 -----
00000000: 80 A4 08 00 06 3F 00 30  00 30 02                 .....?.0.0.

00000000: 90 00                                             ..


----- APDU command/response pair 64 -----
00000000: 00 E0 08 00 27 62 81 24  8A 01 04 83 02 00 41 80  ....'b.$......A.
00000010: 02 00 80 85 05 05 0C 20  00 A3 86 0E 00 00 00 FF  ....... ........
00000020: 00 FF 00 20 00 20 00 00  00 20 71 00              ... . ... q.

00000000: 90 00                                             ..


----- APDU command/response pair 65 -----
00000000: 80 A4 00 00 02 00 41                              ......A

00000000: 90 00                                             ..


----- APDU command/response pair 66 -----
00000000: 00 47 00 00 0C AC 81 09  80 01 06 81 81 03 01 00  .G..............
00000010: 01                                                .

00000000: 90 00                                             ..


----- APDU command/response pair 67 -----
00000000: 80 A4 08 00 08 3F 00 30  00 30 02 00 41           .....?.0.0..A
```

```
00000000: 90 00                                             ..


----- APDU command/response pair 68 -----
00000000: 80 48 00 00 00                                    .H...

00000000: 7F 49 81 88 81 81 80 D1  EF 7C A5 06 A1 87 FD 5F  .I.......|....._
00000010: 13 5B 25 B7 16 B9 BA A7  21 43 3D DB 51 9D C1 D1  .[%.....!C=.Q...
00000020: 5A 3C 95 7C B6 F0 37 57  83 CF 2D 0B 53 66 C7 11  Z<.|..7W..-.Sf..
00000030: D5 6B FD 28 FA A0 EA 50  1E 2B FD B5 09 49 E2 E7  .k.(...P.+...I..
00000040: 51 67 1B 00 B0 9D 52 CD  22 D8 69 8C 36 74 54 41  Qg....R.".i.6tTA
00000050: 6E 40 58 4F 79 52 E4 D9  00 43 9C 2C 79 FE A6 48  n@XOyR...C.,y..H
00000060: B7 31 8A B2 05 04 C4 DD  B3 86 E6 4F 38 A6 5D 2A  .1.........O8.]*
00000070: CD A8 3F 95 E4 FF 7B 05  1E ED 4A B5 99 69 36 F0  ..?...{...J..i6.
00000080: B9 5B 29 C6 EC B3 25 82  03 01 00 01 90 00        .[)...%.......


----- APDU command/response pair 69 -----
00000000: 80 A4 08 00 06 3F 00 30  00 30 02                 .....?.0.0.

00000000: 90 00                                             ..


----- APDU command/response pair 70 -----
00000000: 00 E0 08 00 B0 62 81 AD  8A 01 04 83 02 00 81 80  .....b..........
00000010: 02 00 80 85 05 05 08 20  00 A3 86 0E 00 00 00 FF  ....... ........
00000020: 00 FF 00 20 00 20 00 00  00 20 71 81 88 90 03 01  ... . ... q.....
00000030: 00 01 91 81 80 D1 EF 7C  A5 06 A1 87 FD 5F 13 5B  .......|....._.[
00000040: 25 B7 16 B9 BA A7 21 43  3D DB 51 9D C1 D1 5A 3C  %.....!C=.Q...Z<
00000050: 95 7C B6 F0 37 57 83 CF  2D 0B 53 66 C7 11 D5 6B  .|..7W..-.Sf...k
00000060: FD 28 FA A0 EA 50 1E 2B  FD B5 09 49 E2 E7 51 67  .(...P.+...I..Qg
00000070: 1B 00 B0 9D 52 CD 22 D8  69 8C 36 74 54 41 6E 40  ....R.".i.6tTAn@
00000080: 58 4F 79 52 E4 D9 00 43  9C 2C 79 FE A6 48 B7 31  XOyR...C.,y..H.1
00000090: 8A B2 05 04 C4 DD B3 86  E6 4F 38 A6 5D 2A CD A8  .........O8.]*..
000000A0: 3F 95 E4 FF 7B 05 1E ED  4A B5 99 69 36 F0 B9 5B  ?...{...J..i6..[
000000B0: 29 C6 EC B3 25                                    )...%

00000000: 90 00                                             ..


----- APDU command/response pair 71 -----
00000000: 80 A4 08 00 08 3F 00 30  00 30 02 00 41           .....?.0.0..A

00000000: 90 00                                             ..


----- APDU command/response pair 72 -----
00000000: 80 48 00 00 00                                    .H...
```

```
00000000: 7F 49 81 88 81 81 80 D1  EF 7C A5 06 A1 87 FD 5F  .I.......|....._
00000010: 13 5B 25 B7 16 B9 BA A7  21 43 3D DB 51 9D C1 D1  .[%.....!C=.Q...
00000020: 5A 3C 95 7C B6 F0 37 57  83 CF 2D 0B 53 66 C7 11  Z<.|..7W..-.Sf..
00000030: D5 6B FD 28 FA A0 EA 50  1E 2B FD B5 09 49 E2 E7  .k.(...P.+...I..
00000040: 51 67 1B 00 B0 9D 52 CD  22 D8 69 8C 36 74 54 41  Qg....R.".i.6tTA
00000050: 6E 40 58 4F 79 52 E4 D9  00 43 9C 2C 79 FE A6 48  n@XOyR...C.,y..H
00000060: B7 31 8A B2 05 04 C4 DD  B3 86 E6 4F 38 A6 5D 2A  .1.........O8.]*
00000070: CD A8 3F 95 E4 FF 7B 05  1E ED 4A B5 99 69 36 F0  ..?...{...J..i6.
00000080: B9 5B 29 C6 EC B3 25 82  03 01 00 01 90 00        .[)...%.......


----- APDU command/response pair 73 -----
00000000: 80 A4 08 00 08 3F 00 30  00 30 02 01 40           .....?.0.0..@

00000000: 90 00                                             ..


----- APDU command/response pair 74 -----
00000000: 00 D6 00 F5 82 00 80 D1  EF 7C A5 06 A1 87 FD 5F  .........|....._
00000010: 13 5B 25 B7 16 B9 BA A7  21 43 3D DB 51 9D C1 D1  .[%.....!C=.Q...
00000020: 5A 3C 95 7C B6 F0 37 57  83 CF 2D 0B 53 66 C7 11  Z<.|..7W..-.Sf..
00000030: D5 6B FD 28 FA A0 EA 50  1E 2B FD B5 09 49 E2 E7  .k.(...P.+...I..
00000040: 51 67 1B 00 B0 9D 52 CD  22 D8 69 8C 36 74 54 41  Qg....R.".i.6tTA
00000050: 6E 40 58 4F 79 52 E4 D9  00 43 9C 2C 79 FE A6 48  n@XOyR...C.,y..H
00000060: B7 31 8A B2 05 04 C4 DD  B3 86 E6 4F 38 A6 5D 2A  .1.........O8.]*
00000070: CD A8 3F 95 E4 FF 7B 05  1E ED 4A B5 99 69 36 F0  ..?...{...J..i6.
00000080: B9 5B 29 C6 EC B3 25                              .[)...%

00000000: 90 00                                             ..
```

B.6 C_destroyObject

```
----- APDU command/response pair 35 -----
00000000: 80 A4 08 00 08 3F 00 30  00 30 01 03 40           .....?.0.0..@

00000000: 90 00                                             ..

----- APDU command/response pair 36 -----
00000000: 00 B0 00 00 00                                    .....

00000000: 00 03 03 40 01 23 18 00  00 00 00 04 04 00 00 00  ...@.#..........
00000010: 00 01 00 00 01 01 00 02  00 00 01 00 00 03 10 00  ................
00000020: 04 64 65 73 33 FF FF FF  FF FF FF FF FF FF FF FF  .des3...........
00000030: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000040: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
```

```
00000050: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000060: FF 00 11 01 00 18 FF FF  FF FF FF FF FF FF FF FF  ................
00000070: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF 01 00  ................
00000080: 00 00 04 15 00 00 00 01  02 10 00 01 01 FF FF FF  ................
00000090: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000000A0: FF FF FF FF FF FF FF FF  FF FF FF FF 01 03 30 00  ..............0.
000000B0: 01 01 01 04 00 00 01 01  01 05 50 00 01 01 01 06  ..........P.....
000000C0: 00 00 01 00 01 07 50 00  01 01 01 08 50 00 01 01  ......P.....P...
000000D0: 01 0A 00 00 01 01 01 0C  10 00 01 00 01 10 10 00  ................
000000E0: 00 FF FF FF FF FF FF FF  FF 01 11 10 00 00 FF FF  ................
000000F0: FF FF FF FF FF FF 01 62  50 00 01 00 01 63 00 00  .......bP....c..
00000100: 61 27                                             a’


----- APDU command/response pair 37 -----
00000000: 00 B0 01 00 00                                    .....

00000000: 01 01 01 64 00 00 01 01  01 65 00 00 01 01 01 66  ...d.....e.....f
00000010: 00 00 04 31 01 00 00 01  70 00 00 01 01 80 10 00  ...1....p.......
00000020: 00 01 00 97 03 97 03 90  00                       .........

----- APDU command/response pair 49 -----
00000000: 80 A4 08 00 08 3F 00 30  00 30 03 40 01           .....?.0.0.@.

00000000: 90 00

----- APDU command/response pair 50 -----
00000000: 00 D6 00 04 04 98 03 98  03                       .........

00000000: 90 00                                             ..

----- APDU command/response pair 53 -----
00000000: 80 A4 08 00 08 3F 00 30  00 30 01 00 C1           .....?.0.0...

00000000: 90 00                                             ..


----- APDU command/response pair 54 -----
00000000: 00 E4 00 00                                       ....

00000000: 90 00                                             ..


----- APDU command/response pair 55 -----
00000000: 80 A4 08 00 08 3F 00 30  00 30 01 03 40           .....?.0.0..@

00000000: 90 00                                             ..
```

```
----- APDU command/response pair 56 -----
00000000: 00 E4 00 00                                       ....

00000000: 90 00                                             ..
```

B.7 C_encrypt

```
----- APDU command/response pair 52 -----
00000000: 80 A4 08 00 08 3F 00 30  00 30 01 00 C1           .....?.0.0...

00000000: 90 00                                             ..


----- APDU command/response pair 53 -----
00000000: 00 2A 82 05 13 80 81 10  54 65 73 74 53 74 72 69  .*......TestStri
00000010: 6E 67 31 32 33 34 35 36  00                       ng123456.

00000000: 82 10 B4 F0 97 B6 63 E4  68 7A 8B 00 4F DF 3A C1  ......c.hz..O.:.
00000010: 49 9F 90 00                                       I...
```

B.8 C_decrypt

```
----- APDU command/response pair 64 -----
00000000: 80 A4 08 00 08 3F 00 30  00 30 01 00 C1           .....?.0.0...

00000000: 90 00                                             ..


----- APDU command/response pair 65 -----
00000000: 00 2A 80 05 0B 82 81 08  8B 00 4F DF 3A C1 49 9F  .*........O.:.I.
00000010: 00                                                .

00000000: 80 08 6E 67 31 32 33 34  35 36 90 00              ..ng123456..
```

B.9 C_setAttribute

```
----- APDU command/response pair 51 -----
00000000: 80 A4 08 00 08 3F 00 30  00 30 01 03 40           .....?.0.0..@

00000000: 90 00                                             ..


----- APDU command/response pair 52 -----
```

```
00000000: 90 32 00 03 FF 01 27 00  00 62 82 01 27 00 03 03  .2....’..b..’...
00000010: 40 01 23 18 00 00 00 00  04 04 00 00 00 00 01 00  @.#.............
00000020: 00 01 01 00 02 00 00 01  00 00 03 10 00 07 63 68  ..............ch
00000030: 61 6E 67 65 64 FF FF FF  FF FF FF FF FF FF FF FF  anged...........
00000040: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000050: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000060: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF 00 11  ................
00000070: 01 00 18 FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000080: FF FF FF FF FF FF FF FF  FF FF FF 01 00 00 00 04  ................
00000090: 15 00 00 00 01 02 10 00  01 01 FF FF FF FF FF FF  ................
000000A0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000000B0: FF FF FF FF FF FF FF FF  FF 01 03 30 00 01 01 01  ...........0....
000000C0: 04 00 00 01 01 01 05 50  00 01 01 01 06 00 00 01  .......P........
000000D0: 00 01 07 50 00 01 01 01  08 50 00 01 01 01 0A 00  ...P.....P......
000000E0: 00 01 01 01 0C 10 00 01  00 01 10 10 00 00 FF FF  ................
000000F0: FF FF FF FF FF FF 01 11  10 00 00 FF FF FF FF FF  ................
00000100: FF FF FF 01                                       ....

00000000: 90 00                                             ..


----- APDU command/response pair 53 -----
00000000: 80 32 00 03 30 62 50 00  01 00 01 63 00 00 01 01  .2..0bP....c....
00000010: 01 64 00 00 01 01 01 65  00 00 01 01 01 66 00 00  .d.....e.....f..
00000020: 04 31 01 00 00 01 70 00  00 01 01 80 10 00 00 01  .1....p.........
00000030: 00 99 03 99 03                                    .....

00000000: 90 00                                             ..
```

B.10 C_unwrap

```
----- APDU command/response pair 92 -----
00000000: 80 48 00 80 00                                    .H...

00000000: 80 01 05 81 81 80 F7 B5  15 72 07 22 94 6F C4 08  .........r.".o..
00000010: 64 CB BD AF EA 55 7D BD  8F 55 36 B0 01 C2 8B 2E  d....U}..U6.....
00000020: 32 B6 5D 45 F1 74 5D 38  12 0B AD 9D 2C 03 9C 22  2.]E.t]8....,.."
00000030: 46 68 EB 2E A2 8C 20 95  A8 2E 6C A8 E0 6D 47 F2  Fh.... ...l..mG.
00000040: D3 1E D7 01 F8 15 5C AD  DC 05 70 C0 93 B2 6D 74  ......:.p...mt
00000050: B0 9B 95 E6 4D 8C D2 FC  73 3E CD 0F 30 68 79 A5  ....M...s>..0hy.
00000060: B9 35 F2 41 3F 52 AD AD  32 A0 99 1A 18 3D CC 57  .5.A?R..2....=.W
00000070: 7E 39 DA 47 53 1E 67 15  AB 01 70 7F F2 47 96 71  ~9.GS.g...p..G.q
00000080: 44 23 CE 7B 60 67 82 81  80 3C 52 D2 06 89 28 92  D#.{‘g...<R...(.
00000090: 2C AB E6 3C 4E E6 DF 0E  D2 29 F1 01 BE 36 C4 F8  ,..<N....)...6..
000000A0: 54 40 56 F3 4A FA 8D 2E  9B 60 F5 07 BC ED B4 44  T@V.J....‘.....D
000000B0: 56 68 5D 82 4C C4 EA D7  96 20 F8 C5 46 A6 E0 16  Vh].L.... ..F...
```

```
000000C0: B8 AB A5 D8 43 29 58 53  77 17 09 97 AA 70 68 33  ....C)XSw....ph3
000000D0: 9E F1 41 0A 5F 39 D9 75  24 7F 3A 53 63 61 47 87  ..A._9.u$.:ScaG.
000000E0: 87 7F 88 96 BC BB 83 A1  CB D1 42 E0 EB 99 CF 34  ..........B....4
000000F0: 0E CA 56 4F 2C 57 50 6E  7B 1A FC 1F 90 7A E0 C2  ..VO,WPn{....z..
00000100: 61 09                                             a.


----- APDU command/response pair 93 -----
00000000: 00 C0 00 00 09                                    .....

00000000: A8 5D D3 30 E3 5C A9 00  39 90 00                 .].0....9..


----- APDU command/response pair 94 -----
00000000: 80 86 00 00 80 D0 7E EE  17 C7 31 DD 53 FB 1F D4  ......~...1.S...
00000010: 36 65 EB 7F 2C B0 A2 34  44 80 D7 F4 31 96 12 DF  6e..,..4D...1...
00000020: C8 AD 3C 41 EE 8F 13 C2  8A 3B 8D 6B 73 18 A6 1B  ..<A.....;.ks...
00000030: 46 3E 10 93 5C 2F 35 1C  A3 FC 48 09 DB E4 BB EA  F>..5...H.....
00000040: 3F 1A 11 7D 85 57 2F 85  75 1D 8B F4 E6 39 2B FA  ?..}.W/.u....9+.
00000050: 19 3D 7A BB E3 75 B2 A2  A9 E4 EE 79 4F A6 3F EE  .=z..u.....yO.?.
00000060: FD BF 4A F8 43 8F DA A9  D1 8D 58 63 12 5D C8 E8  ..J.C.....Xc.]..
00000070: 2C 77 8F 5F 96 C0 51 CA  19 B1 80 D5 80 4E 50 8B  ,w._..Q......NP.
00000080: 88 6B 64 43 0D 00                                 .kdC..

00000000: FD 74 3B 38 48 7E 0E D9  4D B0 BF E7 66 3D E4 63  .t;8H~..M...f=.c
00000010: 15 24 EC 7B F3 93 C7 90  85 43 E8 DF D9 E0 60 88  .$.{.....C....‘.
00000020: 90 00                                             ..


----- APDU command/response pair 95 -----
00000000: 8C A4 08 00 1D 87 11 01  65 FD 9A B5 09 70 96 93  ........e....p..
00000010: FB 5D 39 FF B3 24 6B 8E  8E 08 04 D7 B0 58 E0 96  .]9..$k......X..
00000020: E6 01                                             ..

00000000: 99 02 90 00 8E 08 67 C9  1F 50 18 5F 6D 6A 90 00  ......g..P._mj..


----- APDU command/response pair 96 -----
00000000: 0C 2A 80 0A 99 87 81 89  01 1D 7A 97 D8 25 8F 60  .*........z..%.‘
00000010: 52 07 AE DC A9 AC 33 7C  6E 12 A9 79 71 B8 36 1B  R.....3|n..yq.6.
00000020: 29 C3 54 C1 A8 29 A4 4F  75 72 4E C6 C5 71 22 88  ).T..).OurN..q".
00000030: 50 0C 29 9F 75 C7 99 39  E9 B6 5B AF A1 65 51 DE  P.).u..9..[..eQ.
00000040: 56 84 6D 30 B6 2F F3 19  6B 83 82 C4 6B AB 59 E3  V.m0./..k...k.Y.
00000050: 2B FD B1 4B FC 3D BE CD  16 C8 C0 69 80 5C 0E 72  +..K.=.....i...r
00000060: C0 0F 24 0A 3E 8A 88 4A  CA 68 02 5C FA B5 36 33  ..$.>..J.h..63
00000070: CB 5A F7 BE 86 21 2F 68  DB 5F 46 1D 67 FA C2 8B  .Z...!/h._F.g...
00000080: A9 58 37 5C F0 34 7E FE  FC 1A 78 46 C7 51 0B 13  .X74~...xF.Q..
00000090: B2 97 01 00 8E 08 F9 2D  53 7C AD 46 EB 79 00      .......-S|.F.y.
```

```
00000000: 87 11 01 FC E7 96 A5 B1  96 E9 E3 1D 2D 3A 49 46  ............-:IF
00000010: 8C 97 A7 99 02 90 00 8E  08 41 16 94 D4 58 27 D0  .........A...X’.
00000020: 3F 90 00                                          ?..


----- APDU command/response pair 97 -----
00000000: 80 86 FF FF                                       ....

00000000: 90 00                                             ..


----- APDU command/response pair 119 -----
00000000: 80 A4 08 00 08 3F 00 30  00 30 01 03 41           .....?.0.0..A

00000000: 90 00                                             ..


----- APDU command/response pair 120 -----
00000000: 00 D6 00 00 FA 01 03 03  41 01 23 18 00 00 00 00  ........A.#.....
00000010: 04 04 00 00 00 00 01 00  00 01 01 00 02 00 00 01  ................
00000020: 00 00 03 10 00 04 74 65  73 74 FF FF FF FF FF FF  ......test......
00000030: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000040: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000050: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000060: FF FF FF FF FF FF 00 11  01 00 08 FF FF FF FF FF  ................
00000070: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00000080: FF FF FF 01 00 00 00 04  13 00 00 00 01 02 10 00  ................
00000090: 01 10 FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000000A0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000000B0: FF 01 03 30 00 01 01 01  04 00 00 01 01 01 05 50  ...0...........P
000000C0: 00 01 01 01 06 00 00 01  00 01 07 50 00 01 01 01  ...........P....
000000D0: 08 50 00 01 01 01 0A 00  00 01 01 01 0C 10 00 01  .P..............
000000E0: 00 01 10 10 00 00 FF FF  FF FF FF FF FF FF 01 11  ................
000000F0: 10 00 00 FF FF FF FF FF  FF FF FF 01 62 50 00     ............bP.

00000000: 90 00                                             ..


----- APDU command/response pair 121 -----
00000000: 00 D6 00 FA 2D 01 00 01  63 00 00 01 00 01 64 00  ....-...c.....d.
00000010: 00 01 00 01 65 00 00 01  00 01 66 00 00 04 FF FF  ....e.....f.....
00000020: FF FF 01 70 00 00 01 01  80 10 00 00 01 00 B0 03  ...p............
00000030: B0 03                                             ..

00000000: 90 00                                             ..


----- APDU command/response pair 122 -----
00000000: 80 48 00 80 00                                    .H...
```

```
00000000: 80 01 05 81 81 80 F7 B5  15 72 07 22 94 6F C4 08  .........r.".o..
00000010: 64 CB BD AF EA 55 7D BD  8F 55 36 B0 01 C2 8B 2E  d....U}..U6.....
00000020: 32 B6 5D 45 F1 74 5D 38  12 0B AD 9D 2C 03 9C 22  2.]E.t]8....,.."
00000030: 46 68 EB 2E A2 8C 20 95  A8 2E 6C A8 E0 6D 47 F2  Fh.... ...l..mG.
00000040: D3 1E D7 01 F8 15 5C AD  DC 05 70 C0 93 B2 6D 74  ......:.p...mt
00000050: B0 9B 95 E6 4D 8C D2 FC  73 3E CD 0F 30 68 79 A5  ....M...s>..0hy.
00000060: B9 35 F2 41 3F 52 AD AD  32 A0 99 1A 18 3D CC 57  .5.A?R..2....=.W
00000070: 7E 39 DA 47 53 1E 67 15  AB 01 70 7F F2 47 96 71  ~9.GS.g...p..G.q
00000080: 44 23 CE 7B 60 67 82 81  80 3C 52 D2 06 89 28 92  D#.{‘g...<R...(.
00000090: 2C AB E6 3C 4E E6 DF 0E  D2 29 F1 01 BE 36 C4 F8  ,..<N....)...6..
000000A0: 54 40 56 F3 4A FA 8D 2E  9B 60 F5 07 BC ED B4 44  T@V.J....‘.....D
000000B0: 56 68 5D 82 4C C4 EA D7  96 20 F8 C5 46 A6 E0 16  Vh].L.... ..F...
000000C0: B8 AB A5 D8 43 29 58 53  77 17 09 97 AA 70 68 33  ....C)XSw....ph3
000000D0: 9E F1 41 0A 5F 39 D9 75  24 7F 3A 53 63 61 47 87  ..A._9.u$.:ScaG.
000000E0: 87 7F 88 96 BC BB 83 A1  CB D1 42 E0 EB 99 CF 34  ..........B....4
000000F0: 0E CA 56 4F 2C 57 50 6E  7B 1A FC 1F 90 7A E0 C2  ..VO,WPn{....z..
00000100: 61 09                                             a.


----- APDU command/response pair 123 -----
00000000: 00 C0 00 00 09                                    .....

00000000: A8 5D D3 30 E3 5C A9 00  39 90 00                 .].0....9..


----- APDU command/response pair 124 -----
00000000: 80 86 00 00 80 95 3B CF  46 B8 4E 67 E4 6B 97 4B  ......;.F.Ng.k.K
00000010: 70 AD B3 44 22 6A 1B 42  18 4B A9 44 FF 28 FA C0  p..D"j.B.K.D.(..
00000020: 0A EF 44 CD DA C1 28 2B  CF FD 5D 20 48 50 33 59  ..D...(+..] HP3Y
00000030: 7D B7 CB 73 4A EF 28 0A  C7 E4 02 2A 91 A9 F6 55  }..sJ.(....*...U
00000040: 97 D3 A8 DE 21 90 0E 23  0B 9C ED 4B 52 39 46 ED  ....!..#...KR9F.
00000050: 13 1F 7F 9D CB EF 7A DD  7C D7 39 EC 1F BD 2A 3A  ......z.|.9...*:
00000060: 45 48 8F 6C 7E 82 71 E5  14 8F C1 9D F8 E8 53 2B  EH.l~.q.......S+
00000070: D3 AF 3D 7C 11 59 E3 81  F4 0B 08 17 A9 0F 37 69  ..=|.Y........7i
00000080: 90 C1 11 E2 1B 00                                 ......

00000000: B3 0F 6C 66 E6 56 8F 44  55 B2 A6 02 0E 0B 80 01  ..lf.V.DU.......
00000010: FF 89 7A 65 FC 68 25 82  22 C9 97 74 D1 6B 00 AB  ..ze.h%."..t.k..
00000020: 90 00                                             ..


----- APDU command/response pair 125 -----
00000000: 8C A4 08 00 15 87 09 01  82 46 BD FD 60 2D E4 C6  .........F..‘-..
00000010: 8E 08 25 35 C0 28 0E E1  20 93                    ..%5.(.. .

00000000: 99 02 90 00 8E 08 8C D6  A9 A8 99 7F 14 12 90 00  ................
```

```
----- APDU command/response pair 126 -----
00000000: 0C E0 08 00 3D 87 31 01  4D 4F 3D AB 31 72 FC F7  ....=.1.MO=.1r..
00000010: B4 84 D1 41 19 1C 22 DF  3F 60 BE 6B 0A 1E 49 5F  ...A..".?‘.k..I_
00000020: AD 3D 6D 61 5E DA E3 F7  A8 0A 82 EA 65 16 8A 01  .=ma^.......e...
00000030: C5 4F BF 3F 44 73 9C 61  8E 08 A8 A9 A5 4D 55 BB  .O.?Ds.a.....MU.
00000040: E7 B3                                             ..

00000000: 99 02 90 00 8E 08 23 E5  DF 34 11 21 87 1C 90 00  ......#..4.!....


----- APDU command/response pair 127 -----
00000000: 80 86 FF FF                                       ....

00000000: 90 00                                             ..
```

B.11 C_wrap

```
Enter command (spaced integers)
00 42 130 10 08 49 50 51 52 53 54 55 56 00 (integers)
00 2A 82  0A 08 31 32 33 34 35 36 37 38 00 (hexadecimals)
command changed!

RESPONSE
00000000: 6A 80                                             j.
```