Secure messaging of Gemalto IDClassic 340

Julijonas Kikutis

July 14, 2017

1 Smartcard

Gemalto IDClassic 340 is a smartcard storing private and public keys and performing signing and decryption. Its datasheet [1] says it is based on the IDCore JavaCard platform and the Classic v3 applet and works with its minidriver and the IDGo 300 (Classic Client) software. The Classic v3 is a JavaCard applet installed on the card that provides cryptographic functions, file management, and security. The cryptographic functions are: signing, key pair generation, and session key decryption. Supported algorithms on the card are: 3DES (ECB, CBC), SHA-1, SHA-256, RSA 1024, RSA 2048.

There are 12 private key slots on the card for the following purposes:

- 1-2 are RSA 1024 signing keys,
- 3-6 are RSA 1024 signing and decryption keys,
- 7-8 are RSA 2048 signing keys, and
- 9-12 are RSA 2048 signing and decryption keys.

There is a Classic Client 32-bit Windows application for administrative tasks, such as card initialisation and PIN reset. The Windows software also includes PKCS#11 (Cryptoki) library. A version of Classic Client for Linux containing only libraries can be found on the internet. There are two libraries that provide the same PKCS#11 API and are called `libgclib.so` and `libgck2015x.so` in `/usr/lib/ClassicClient`.

2 File structure

The dedicated file that is selected as application ID as the first command is A0 00 00 00 18 0C 00 00 01 63 42 00 and enables most commands. Then the following files are accessed by the PKCS#11 library using SELECT FILE.

2F 00	EF_DIR containing AID and textual label of application (constant)
50 00 50 31	Repeating pattern 30 08 04 06 3F 00 50 00 50 06 A0 0A (constant)
50 00 50 33	Remaining free key slots, erased and written again after key generation, delete key
50 00 50 32	8 byte card identifier and constant “Gemalto S.A. GemP15-1”
00 01	8 byte card identifier, used to derive challenge–response keys
00 02	Numeric value in format 333Z.33333333 that library updates on every write access
50 00	DF, to read its content need to satisfy security condition
50 00 50 01	Labels and IDs of private keys
50 00 50 02	Labels and IDs of public keys
50 00 50 06	PIN and PUK code information
50 00 50 34	Information about private key slots

Data object TLV tags accessed using GET DATA:

9F 7F	Card production life cycle (CPLC) data, 2 bytes are different for different cards
DF 30	“v3.03” (constant)

The following GET DATA command is used to retrieve public keys:

00 CB 00 FF 0A B6 03 83 01 **0A** 7F 49 02 **81** 00 where first byte in bold is slot ranging from 03 to 06 and from 09 to 0C and second byte in bold is either 81 modulus or 82 exponent.

3 Cryptographic functions

Before any operation, the PIN is sent to card using VERIFY message in plaintext. For encryption, the public key is retrieved from the card in plaintext. For decryption, the ciphertext is sent and plaintext is returned unencrypted and not in secure messaging. During key pair generation, private and public key import, secure messaging is established using challenge–response authentication as seen in subsection 4.3. Then the appropriate metadata files are modified, GENERATE PUBLIC KEY PAIR issued or PUT DATA with encrypted public or private key sent as part of secure messaging. The encryption of public and private keys during import needs to be investigated.

4 Secure messaging

Before doing any smartcard state changing operations, such as import or generation of keys, the smartcard and library performs challenge–response authentication, which establishes new keys used for MAC calculation in secure messaging. Secure messaging is indicated by APDU command class byte 0C. Messages sent in secure messaging are sent in plaintext but with their MAC appended. This is known as the authentic mode procedure, which should guarantee authentic transmission of APDUs, meaning that the APDUs are protected against manipulation during transmission [2]. However, the PKCS#11 library derives the keys used for challenge–response encryption from static key and card identifier, which allows reverse engineering of all subsequent keys by an attacker and does not guarantee authentic transmission.

4.1 Static key

A static key K_S is calculated in the library using AES-CBC decryption:

$$K_S = \mathrm{D}_{\text{AES-CBC}}(K_{AES}, C, IV)[0:16]$$

The ciphertext C and initialisation vector IV is read by the library from file `keys.conf` in the library directory. There are multiple values in the same TLV format `31 10` $\|$ IV $\|$ `32 01 10 33 20` $\|$ C, the relevant one is under heading `[v3_3des_1]`. The alphanumeric AES key K_{AES} cannot be found in the libraries using simple byte search. Because the card does not support AES algorithm to calculate this key itself, it is likely that the decrypted static key is constant among cards of the same model, and does not depend on some other values. Note that 32 byte K_{AES} is a concatenation of 16 byte alphanumeric string and its reverse. The three parameters are:

K_{AES}	`Yy32echR8gWImxqKKqxmIWg8Rhce23yY`
C	`58 dc e2 03 c6 63 d1 ac  42 a0 e9 8e 70 32 a9 18`
	`71 47 79 06 c5 6f 8b 76  41 f6 b8 be d1 20 f4 6a`
IV	`c2 fd fa 6b 6f b4 87 38  07 89 10 40 6e d7 fa 2a`

4.2 Card identifier

The card identifier I consists of 8 bytes and is kept in files 50 00 50 32 and 00 01. The identifier in file 00 01 is used to derive challenge–response encryption and MAC keys. It is retrieved from the card using SELECT FILE and READ BINARY.

Note that the card identifier and contents of other files are cached by the library for subsequent usage of the same smartcard in multiple files in `/dev/shm` directory. To be able to calculate the challenge–response keys solely from an APDU trace, the files in this directory have to be deleted so that the library would read the files again.

The identifiers for two cards are as follows:

Card 1	`30 40 00 1A 66 83 29 71`
Card 2	`30 40 00 19 67 C3 29 71`

Two 8 byte keys in K_S are swapped to derive another key:

$$K'_S = K_S[8:16] \,\|\, K_S[0:8]$$

Then challenge–response encryption and MAC keys are calculated as follows:

$$K_{CR} = \mathrm{E}_{\text{3DES-ECB}}(K_S, I) \parallel \mathrm{E}_{\text{3DES-ECB}}(K'_S, I)$$
$$K_{MAC} = \mathrm{E}_{\text{3DES-ECB}}(K_S, \hat{I}) \parallel \mathrm{E}_{\text{3DES-ECB}}(K'_S, \hat{I})$$

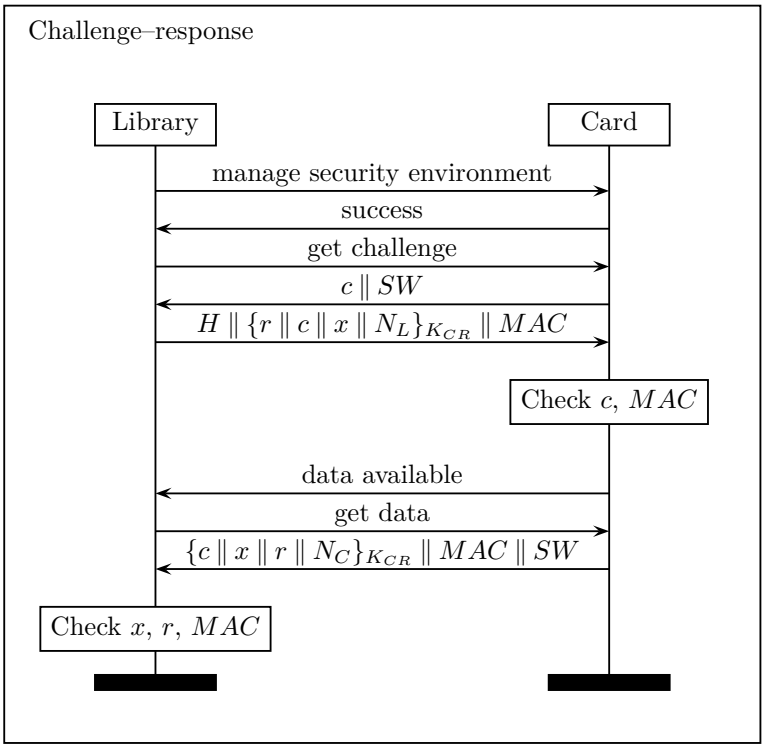
where $\hat{I}$ is reversed byte string I.

4.3 Challenge–response authentication

The challenge–response authenticates and establishes nonces N_L and N_C for secure messaging. The encryption used is 3DES-CBC under secret key $K_{CR} = k_1 \parallel k_2$ and initialization vector of zeros. The key is used the following way in each DES encryption during 3DES: $\mathrm{E}_{\text{3DES}}(K_{CR}, m) = \mathrm{E}_{\text{DES}}(k_1, \mathrm{D}_{\text{DES}}(k_2, \mathrm{E}_{\text{DES}}(k_1, m)))$.

Encrypt-then-MAC approach is used. MAC of the two encrypted messages $m = \{\,\cdot\,\}_{K_{CR}}$ is calculated as shown in subsection 4.5 using a seed of zeros and another secret key K_{MAC}, e.g. $\mathrm{MAC}(0, K_{MAC}, m)$.

K_{CR}	encryption key for challenge–response	16 bytes
K_{MAC}	MAC key for challenge–response	16 bytes
c	random smartcard challenge	8 bytes
r	random terminal value	16 bytes
x	constant value	8 bytes
N_L	library nonce	32 bytes
N_C	card nonce	32 bytes



x	constant value from library	`22 34 00 00 AF 04 E3 A9`
H	message header of library challenge	`80 82 00 00 48`
SW	response status for success	`90 00`

4.4 Secure messaging algorithm

For each new message sent in secure messaging mode, we increment the counter i, first message has $i = 1$. We calculate the secure messaging MAC key and seed:

$$K_{SM} = \mathrm{SHA1}((N_L \oplus N_C) \parallel \texttt{00 00 00 02})$$
$$S = (c[4:8] \parallel r[4:8]) + i$$

Data m and header $h = CLA \parallel INS \parallel P1 \parallel P2$ parts are used to calculate $\mathrm{MAC}(K_{SM}, S, m, h)$ as seen in subsection 4.5. The resultant message is $h \parallel LC \parallel m \parallel \texttt{8E 08} \parallel MAC$. Similarly, a response is $m \parallel \texttt{8E 08} \parallel MAC \parallel SW1 \parallel SW2$.

4.5 CBC-MAC algorithm

The following MAC algorithm $\mathrm{MAC}(s, k, m, h)$ is used to calculate MAC for challenge–response and secure messaging messages. The inputs are 8 byte seed s, 16 byte key $k = k_1 \parallel k_2$, message data m, and optional 4 byte header h. The output is 8 byte MAC.

A padding function $\mathrm{P}(m)$ is used that appends `80` byte and an appropriate number of zero bytes to m so that length of $\mathrm{P}(m)$ in bytes n would be at next closest multiple of 8. For secure messaging message MAC calculation, header h is provided and it is padded such that $m' = \mathrm{P}(h) \parallel \mathrm{P}(m)$. For challenge–response MAC and for secure messaging response MAC, $m' = \mathrm{P}(m)$.

Then we initialize with seed as $v_0 = s$. Then $v_i = \mathrm{E}_{\mathrm{DES}}(k_1, v_{i-1}) \oplus m_i$ where m_i is the ith block of padded message m'. Lastly, $MAC = \mathrm{E}_{\text{3DES-ECB}}(k, m_n)$.

References

[1] Gemalto N.V. IDClassic 340 product datasheet, 2008. URL http://webcache.googleusercontent.com/search?q=cache:7iMqALikLXwJ:www.gemalto.com/products/classic_tpc/download/IDClassic340_Product_Datasheet_Dec12.pdf+&cd=1&hl=en&ct=clnk&gl=uk.

[2] W. Rankl and W. Effing. *Smart card handbook*. John Wiley & Sons, 2004.