Journal of Information Security, **2012, 3, 251-258**
http://dx.doi.org/10.4236/jis.2012.34032 Published Online October 2012 (http://www.SciRP.org/journal/jis)

Secure Messaging Implementation in OpenSC

Maurizio Talamo[1,2], Maulahikmah Galinium[3], Christian H. Schunck[2], Franco Arcieri[2]
[1]Department of Engineering, University of Rome Tor Vergata, Rome, Italy
[2]Nestor Laboratory, University of Rome Tor Vergata, Rome, Italy
[3]Department of Information Science, University of Rome Tor Vergata, Rome, Italy
Email: talamo@nestor.uniroma2.it, galinium@nestor.uniroma2.it, schunck@nestor.uniroma2.it, arcieri@nestor.uniroma2.it

Received July 15, 2012; revised August 27, 2012; accepted September 8, 2012

ABSTRACT

Smartcards are used for a rapidly increasing number of applications including electronic identity, driving licenses, physical access, health care, digital signature, and electronic payments. The use of a specific smartcard in a "closed" environment generally provides a high level of security. In a closed environment no other smartcards are employed and the card use is restricted to the smartcard's own firmware, approved software applications, and approved card reader. However, the same level of security cannot be claimed for open environments where smartcards from different manufacturers might interact with various smartcard applications. The reason is that despite a number of existing standards and certification protocols like Common Criteria and CWA 14169, secure and convenient smartcard interoperability has remained a challenge. Ideally, just one middleware would handle the interactions between various software applications and different smartcards securely and seamlessly. In our ongoing research we investigate the underlying interoperability and security problems specifically for digital signature processes. An important part of such a middleware is a set of utilities and libraries that support cryptographic applications including authentication and digital signatures for a significant number of smartcards. The open-source project OpenSC provides such utilities and libraries. Here we identify some security lacks of OpenSC used as such a middleware. By implementing a secure messaging function in OpenSC 0.12.0 that protects the PIN and data exchange between the SC and the middleware, we address one important security weakness. This enables the integration of digital signature functionality into the OpenSC environment.

Keywords: Smart Card; Digital Signature; OpenSC; Secure Messaging

1. Introduction

The problem of secure Smartcard (SC) interoperability is one of the main issues that might limit the use of SCs in the future. The success of SC based online authentication and digital signature services critically depends on how this problem will be addressed: users expect SC based applications to work seamlessly in different environments (home, work, leisure) as well as in different countries (business travel, vacation). Existing ISO standards [1,2] and certification protocols like the Common Criteria (CC) [3] and CWA 14169 [4] do not yet facilitate such seamless use in a sufficiently secure setting.

A certified middleware that facilitates the usage of a wide range of SCs for diverse applications could provide a solution. Studying the security requirements for and creating such a middleware is at the focus of our ongoing research and led us to develop the Crypto Probing System (CPS) with an integrated Murphi model checker [5-7].

OpenSC [8] supports cryptographic operations which are used in SC security operations according to the PKCS#15 standard such as digital signature, the application on which we concentrate here. The PKCS#15 standard is based on the Digital Certificates on SCs and Secured Electronic Information in Society (SEIS) specifications for digital signature applications using SCs [9]. As OpenSC is easy to use and supports a wide range of SCs. OpenSC could be an ideal component of a universal middleware enabling SC interoperability. However, there are several security aspects that need to be addressed in order to ensure the security of such a middleware.

In an environment where several SCs are connected to various applications via a middleware, an evident security problem is that commands that are supposed to be executed on a certain SC are in fact executed on a different one. **Figure 1** illustrates this situation: SC applications give input to and receive an output from several SCs sharing a common middleware. The middleware translates the input into command sequences, *i.e*. into Straight Line Programmes (SLP), which are supposed to be executed on a corresponding SC. Dashed arrows indicate the possibility that commands can interleave between the straight line programs. As a result a command will be executed on a SC different from the intended one.

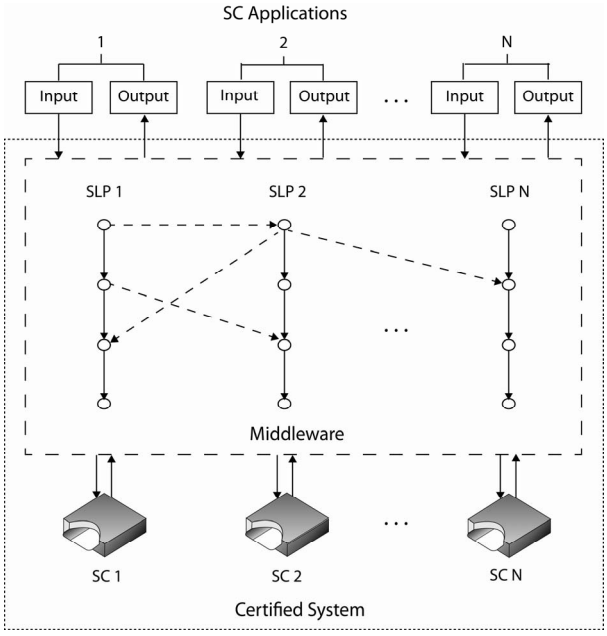

Figure 1. Middleware.

Such situations can arise inadvertently due to errors in the middleware but also be due to an attack. From a security perspective such events are particularly problematic if the SC executing a misdirected command does not immediately return an error message. This problem has been observed experimentally [10] and such a situation is called an "anomaly".

Any secure middleware must therefore be able to fully address the issue of misdirected or interleaving commands. There are several requirements that need to be fulfilled:

- The type of a SC must be securely identified.
- Sensitive information must be communicated using secure messaging (*i.e.* a protected channel between the SC application and the SC must be built and supported by the middleware).
- Anomalies must be efficiently detected and computational chains with two or more anomalies must be avoided [10].

The first problem of using OpenSC as part of a universal middleware is that the evaluation of a SC type is based on the Answer to Reset (ATR) and the file structure of the SC (**Figure 2**). As different SCs types may have the same ATR and file structure this method is not reliable. Apart from interoperability problems that may arise if a SC is actually of a different type than determined by OpenSC using the ATR, an attacker may engineer a SC so that it is recognized as being of a certain type (and serving a certain function) while it has been designed for malicious purposes. In the following we will not focus on this issue but we note that this problem needs to be addressed in future work.

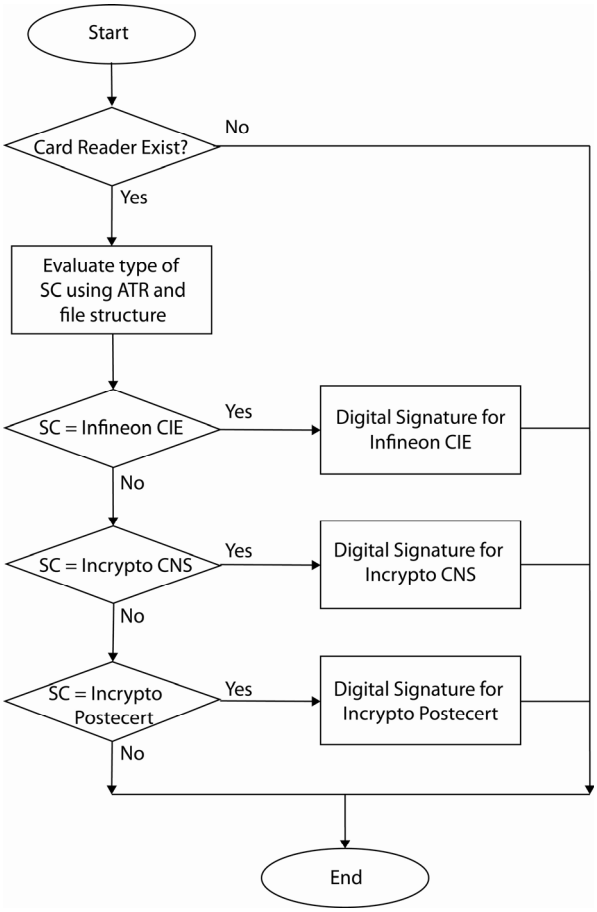

Figure 2. Evaluation of smartcard type in OpenSC.

The second problem of OpenSC is that the current OpenSC libraries do not support secure messaging operations which are required in most digital signature applications to protect the sensitive data exchange between software applications and the SC. In this work we extend the OpenSC libraries to include the secure messaging functionality (Sections 2 and 3). With this solution we facilitate the integration of commercially available digital signature SCs for example Postecert and Infocert into OpenSC 0.12.0 (Section 4). However, even after a "state of the art" integration of SM into OpenSC significant problems regarding the protection of the channel between SC applications and SCs persist as will be discussed in Section 5.

2. Digital Signature Process

The digital signature process as implemented in OpenSC 0.12.0 provides a basic functionality in the *iso*7816.*c* module for performing secure operation computing of a digital signature. However, this functionality does not support the secure messaging.

To meet the requirements of a complete digital signature process as implemented for example on the Incrypto

chip based PosteCert and CNS (Carta Nazionale Servizi) the digital signature process [11] involves the following steps:

step 0 Reset the SC.

step 1 Change directory to the subdirectory containing the digital signature certificate which will be used (SELECT FILE command).

step 2 Activate the security environment for the digital signature (MSE RESTORE command).

step 3 At file system level, choose the private key to be used in the activated security environment (MSE SET command).

step 4 Ask the SC for a random number to be used as a challenge; the first step of activating SM (GET CHALLENGE command).

step 5 Transmit a random number to the SC as a challenge; the second step of activating SM (GIVE CHALLENGE command).

step 6 By using the two random numbers previously exchanged and ciphering 3DES with the shared 3DES key, transmit the PIN that is connected to the private key used for the digital signature operation (VERIFY PIN command). This closes the first SM operation.

step 7 Ask the SC for the random number to be used as (new) challenge; the first step of activating SM (GET CHALLENGE command).

step 8 Transmit a random number to the SC as a challenge; the second step of activating SM (GIVE CHALLENGE command).

step 9 By using the two random numbers previously exchanged and ciphering 3DES with the shared 3DES key, compute and send the input data buffer which is ciphered using the selected private key. Furthermore, receiving the result of the digital signature operation (PSO CDS—Perform Security Operation Compute Digital Signature command).

Extending the OpenSC capabilities to include the additional steps for SM in the digital signature process required modifications of the *pkcs*15-*tool.c* module of OpenSC 0.12.0 which will be detailed below. Integrating the digital signature functions of the digital signature SC into OpenSC also required to take the different file structures (e.g. different locations of PIN and PUK) into account.

3. Secure Messaging in OpenSC

Secure messaging (SM) is used to protect the exchange of sensitive data (e.g. the user's PIN) between the middleware and the SC. In digital signature processes SM is therefore used to protect the data exchanged in connection with the VERIFY PIN and PSO CDS commands (steps 6 and 9 of the digital signature process detailed).

[12] presents a first but incomplete step towards integrating SM into OpenSC. While some of this work was useful as a starting point, achieving the SM functionality required substantial modifications and extensions. As seen in the steps of the digital signature process detailed in the previous section SM requires both the "Get Challenge" and "Give Challenge" functions. However, OpenSC provides only the Get Challenge function so the Give Challenge command was added to the *iso*7816.*c* module and registered in the *sc_card_operations* structure that consists of all the required SC operation commands. Any newly created SC command must be registered in this structure (**Figure 3**).

Furthermore the SM main module *sm.c* and a corresponding header file *sm.h* have been developed from scratch. The header file *sm.h* consists of interface functions used in the *sm.c* module. The *sm.c* module contains all essential functions related to SM operations.

As in [12], we created a SM hook in order to link the SM code into OpenSC. A *sc_sm_context* structure was set up in the header file *opensc.h*. **Figure 4** shows how the *sc_sm_context* structure is used in the *sc_card* structure in order to be recognized by the SC. The *sc_sm_context*

iso7816.c

```
static struct sc_card_operations ops2 = {
        ....
        iso7816_set_security_env,
        is07816_restore_security_env,
        iso7816_give_challenge,
        iso7816_get_challenge (),
        ....
};
```

Figure 3. Structure of the SC operations.

opensc.h

```
...

typedef struct sc_sm_context {
        int use_sm;
} sc_sm_context_t;

...

typedef struct sc_card {
        ...
        ...
        struct sc_card_operations *ops;
        struct sc_sm_context sm_ctx;
        ...
} sc_card_t;

...
```

Figure 4. Secure messaging hook into OpenSC.

structure contains the variable *use_sm* that will be used in the *apdu.c* module as a flag for SM. If the SC requires SM, this variable is flagged when the *pkcs*15-*tool.c* module calls the digital signature process.

For the SM operation, the original APDU serves as an input. The output of the SM operation is the encrypted APDU which is sent to the SC. This involves the following steps [13]:

step 1 Initialize the 24 bytes of SM key.

step 2 Set up and divide the SM key into three encryption and signature keys (8 bytes for each key) according to the 3DES algorithm.

step 3 Obtain two random numbers as challenges (get and give challenge functions—8 bytes for each random number).

step 4 Set up the header block for the APDU Command using the 8 byte random number generated by the get challenge function (**Figure 5(a)**). The header block also contains the mandatory header of the APDU command: the Class Byte (CLA), the Instruction Byte (INS) and two parameter bytes (P1,P2). Since the length of an object must be an integer multiple of 8 bytes, 4 byte padding is required.

step 5 Set up the Cipher Text Block (CTB) and the Cipher Text Object (CTO) by encrypting the data field of the original APDU using the 3DES algorithm. In the VERIFY PIN command, the data field contains the 8 byte PIN of the SC. In the PSO CDS command, the data field contains the input data (117 bytes) to be digitally signed. The CTO structure is a TLV (Tag-Length-Value) object containing a 1 byte Tag, a 1 byte Length variable and a Value variable (encrypted data and padding, either 16 or 120 bytes). **Figure 5(b)** shows the CTO structure using the VERIFY PIN (PSO CDS) commands as examples: the Value contains 16 (120) bytes of encrypted data from the 8 byte PIN (117 byte input data) and 8 (3) byte padding. Furthermore the CTB requires 5 byte padding to concatenate with the 19 (123) byte CTO (**Figure 5(c)**).

step 6 Set up the Net Le Object (if required—**Figure 5(d)**). The Net Le object is used in MAC (Message Authentication Code) Computation when the original APDU does not have a data field. In our experimentation, we do not use this object.

step 7 Set up MAC Computation (using 3DES algorithm), which requires a header block, the CTB and optionally the Net Le Object and set up the MAC object. **Figure 5(e)** shows the MAC Object structure: it is a TLV object containing a 1 byte Tag (0 × 8E), 1 byte Length (0 × 08) and an 8 byte Value (as a result of MAC Computation).

step 8 Create the SM APDU Command including CTO, MAC Object and the optional Net Le Object in the data field (**Figure 5(f)**).

step 9 Send the SM APDU Command and wait for SM APDU Response. SM APDU Response to the Verify PIN command is only a Status Word (SW) without a data field while the response to the PSO CDS command includes a data field as well as the SW (**Figure 5(g)**). This SW is a response code from the SC indicating whether or not the APDU command has generated an error.

These steps are coded in the *do_single_transmit* function of the *apdu.c* module. If SM is required, the *do_single_transmit* function calls the *applySM* function which is coded in the *sm.c* module. As shown in **Figure 6**, the *applySM* function supports the SM process from setting up the header block until composing the APDU

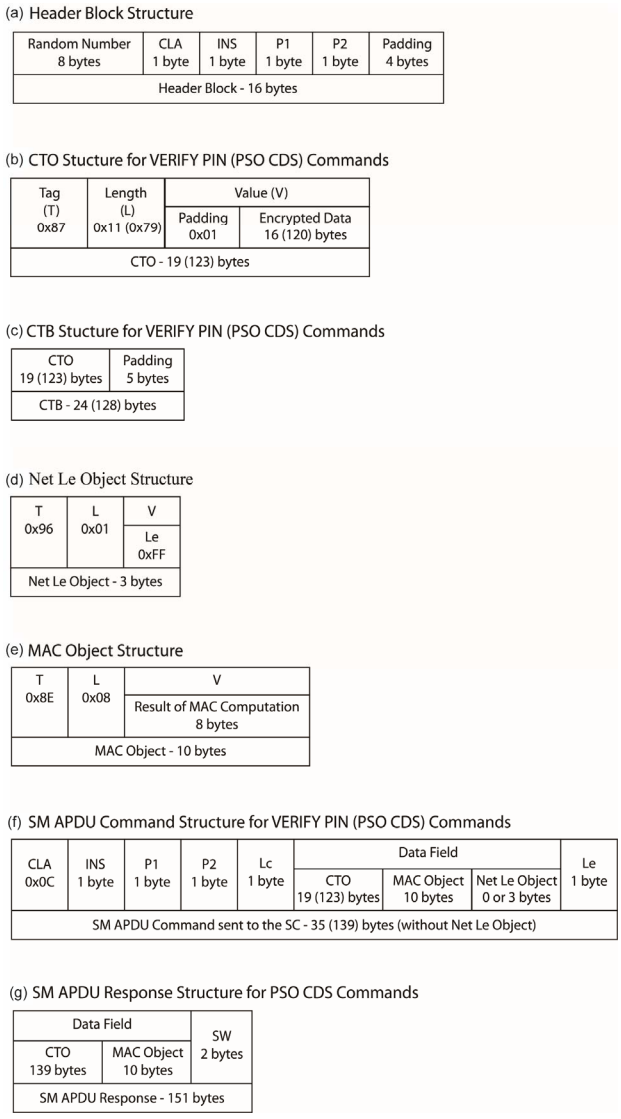

Figure 5. Structure of the objects in the SM operation.


```
sm.c

…

void applySM(sc_apdu_t *apdu, sc_apdu_t *apduSM)
{
    …
    Step I: Set up Header Block
    makeHeaderBlock ( … );

    Step 2: Set up CTB and CTO
    makeCipherTextObjectBlock ( … );

    Step 3: Set up Net Le Object
    makeNetLEObject( … );

    Step 4: Set up MAC Object
    makeMACObject ( … );

    Step 5: Set up APDU SM
    apduSM->cla = apdu->cla | 0x0c;
    apduSM->ins = apdu->ins;
    apduSM->p1  = apdu->p1;
    apduSM->p2  = 0x80 | apdu->p2;
    #ifdef USE_NET_LE
        apduSM->lc = dwCipherTextObjLen + 0x03 + dwMacBlockLen;
    #else
        apduSM->lc = dwCipherTextObjLen + dwMacBlockLen;
    #endif

    apduSM->data = malloc((apduSM->lc) * sizeof(u8));
    apduSM->datalen = apduSM->lc;
    apduSM->resp = apdu->resp;
    apduSM->resplen = apdu->resplen;
    if(apduSM->ins == 0x2A)
        apduSM->le = apdu->le + 1;
    else
        apduSM->le = apdu->le;

    Step 6: Clear temporary variables

    …
}
…
```

Figure 6. Secure messaging implementation in OpenSC.

which is sent to the SC (steps 4 - 8). In particular the *applySM* function implements padding, the 3DES encryption algorithm, MAC signing and the challenge functions.

4. Integrating Digital Signature Cards into OpenSC

OpenSC 0.12.0 recognizes three SCs with digital signature functionality which are approved in Italy [14]:

- The government issued official Italian electronic identity card (Carta Identita Elettronica—CIE).
- The digital signature card issued by the Italian Chambers of Commerce (InfoCert) but without supporting its digital signature functionality.
- The Carta Nazionale Servizi (CNS), which is available to citizens in some Italian regions.

In the previous section we have shown how the digital signature functionality of these cards can be supported via the introduction of SM. Here we show how other digital signature SC can be integrated into OpenSC (including their digital signature functionality) using the example of the PosteCert card issued by the Italian Postal Service (Posteitaliane).

First, a file called *card-itaposte.c* was created which is used to initialize the Postecert card and to call the Postecert card driver (matching the ATR of the card). To recognize the driver, we add the Postecert card by defining *sc_card_type_itaposte_generic* variable and declaring a driver function *sc_get_itaposte_driver* in the file *cards.h*. Finally the driver *sc_get_itaposte_driver* is registered in file *ctx.c* which is used for context related functions.

In order to enable the digital signature functionality of the PosteCert card, the required PKCS#15 functions are provided in the *pkcs*15-*itaposte.c* module. All certificates, private keys and the PIN are created and their locations in the card's file structure are coded in function *sc_pkcs*15*emu_itaposte_init*. The *pkcs*15-*itaposte.c* module is a modified version of the *pkcs*15-*postecert.c* [15] and of the *pkcs*15-*itacns.c* [16] modules to match the current file structure of the Postecert card. We also register the function *sc_pkcs*15*emu_itaposte_init_ex* as built-in emulators in the *pkcs*15-*syn.c* module.

Furthermore an *itaposte.h* module is created in order to support the *card-itaposte.c* and *pkcs*15-*itaposte.c* modules. The *itaposte.h* module is derived from the *itacns.h* module of the CNS card. The *itaposte.h* module contains a structure called *itaposte_drv_data_t*, which consists of the IC manufacturer code, mask manufaturer code, operating system version, card type and SM key. The process of adding Postecert SC into OpenSC is shown in **Figures 7** and **8**.

Finally, the newly added and modified files are compiled together with the unchanged OpenSC files. To do that, the *itaposte.h*, *card-itaposte.c*, *pkcs*15-*itaposte.c*, *sm.h* and *sm.c* modules are registered in *Makefile.am*, and *card-itaposte.obj* and *pkcs*15-*itaposte.obj* are registered in *Makefile.mak*.

Other digital signature SCs [17] can be integrated into OpenSC analogously with minimal changes such as adjusting for different ATR and file structures.

5. Experimentation and Results

In this section we provide an overview of the digital signature process as we have implemented it in OpenSC. **Figure 9** shows the modules which are used in the digital signature process for a certain SC. OpenSC requires the following modules:

- *pkcs*15-*tool.c* module. This module is used to initialize the digital signature process. After calling this module, the digital signature process function is chosen according to the SC used. The digital signature

card-itaposte.c

```
…
static const struct sc_card_operations *default_ops = NULL;
static struct sc_card_operations itaposte_ops;
static struct sc_card_driver itaposte_drv = {
     "Italian Poste", "itaposte", &itaposte_ops, NULL, 0, NULL
};
/* List of ATR's for "hard" matching. */
static struct sc_atr_table itaposte_atrs[] = {
     { "3b:ff:18:00:ff:81:31:fe:55:00:6b:02:09:03:03:01:01:01:44:53:44:10:31:80:90",
          NULL, NULL, SC_CARD_TYPE_ITAPOSTE_GENERIC, 0, NULL},
     { NULL, NULL, NULL, 0, 0, NULL}
};
…
static int itaposte_match_card(sc_card_t *card)
{     …
      itaposte_atr_match ( );
      …
}
static int itaposte_init(sc_card_t *card)
{     …
      itaposte_match_card (card);
      …
}
…
```

cards.h

```
…
enum {
      …
            /* Italian CNS cards */
      SC_CARD_TYPE_ITACNS_BASE = 23000,
      SC_CARD_TYPE_ITACNS_GENERIC,
      …
            /*Postecert Italian card */
      SC_CARD_TYPE_ITAPOSTE_GENERIC,
}
…
extern sc_card_driver_t *sc_get_itacns_driver(void);
extern sc_card_driver_t *sc_get_itaposte_driver(void);
…
```

ctx.c

```
…
static const struct _sc_driver_entry internal_card_drivers[] =
{
      …
      { "itacns",  (void *(*)(void)) sc_get_itacns_driver },
      { "itaposte",  (void *(*)(void)) sc_get_itaposte_driver },
      …
}
…
```

itaposte.h

```
typedef struct {
     u8 ic_manufacturer_code;
     u8 mask_manufacturer_code;
     u8 os_version_h;
     u8 os_version_l;
     u8 poste_version;
          u8 *sm_key;
} itaposte_drv_data_t;
```

Figure 7. Adding postecert SC into OpenSC (1).

process function consists of all sequential required steps for the digital signature.

- *iso*7816.*c* module. This module contains ISO 7816 standard functions which are used in the digital signature process (Section 2).

pkcs15-itaposte.c

```
…
static int sc_pkcs15emu_itaposte_init(sc_pkcs15_card_t * p15card)
{
      …
      const char *itaposte_auth_cert_path = "81108010";
      …
          // add authentication PIN
     sc_format_path("3F008110", &path);
      …
}
int sc_pkcs15emu_itaposte_init_ex(sc_pkcs15_card_t *p15card,
                sc_pkcs15emu_opt_t *opts)
{     …
      sc_pkcs15emu_itaposte_init(p15card);
}
…
```

pkcs15-syn.c

```
…
extern int sc_pkcs15emu_itaposte_init_ex(sc_pkcs15_card_t *,
                        sc_pkcs15emu_opt_t *);

static struct {
      const char *          name;
      int                   (*handler)(sc_pkcs15_card_t *, sc_pkcs15emu_opt_t *);
} builtin_emulators[] = {
      …
      { "itacns",  sc_pkcs15emu_itacns_init_ex},
      { "itaposte",  sc_pkcs15emu_itaposte_init_ex  },
}
…
```

Figure 8. Adding postecert SC into OpenSC (2).

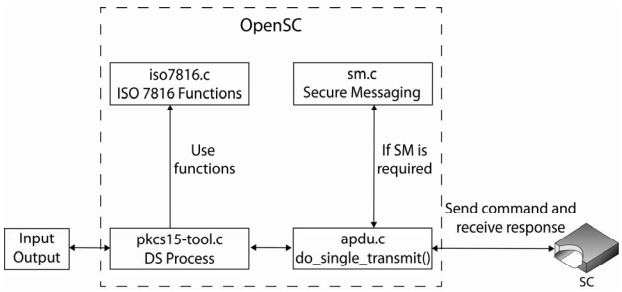

Figure 9. Digital signature process in OpenSC.

- *sm.c* module. This module is used to perform the SM operation.
- *apdu.c* module. This module is used to send the APDU Commands to and receive the APDU Responses from the SCs.

In our experimentation the data to be signed (string of 117 bytes) are provided to the middleware as a file (*input.c*).

While we have implemented "state of the art" secure messaging we note that the security provided by the SM functionality is limited. First, the SM key used to encrypt the input plaintext is generally the same for all SCs of a given type and therefore this key is essentially publicly known and therefore offers questionable protection. Second, the random numbers exchanged in connection with the Get and Give challenge commands are sent in clear between the middleware and SCs. An attacker who is

able to sniff the exchanged random numbers and who has knowledge of the SC specific SM key can then easily decrypt the exchanged messages and/or inject his own commands into the "secured process".

In **Table 1** we summarize the complete digital signature process in terms of the exchanged APDU commands and responses for a digital signature SC using the Incrypto microprocessor. In the Verify (PSO CDS) command the PIN (Input data) are encoded in the byte sequences [09 0F...91 AE] ([5B 5E...3E F0]) and the last 8 bytes of the input data are the result of the MAC computation. Bytes [D6 E4...C1 DA] of the output data contain the digitally signed input data. For each APDU command, the microprocessor replies with an APDU response that consists of optional response data and a 2 byte of Status Word (SW). The SW [90 00] indicates that the APDU command has been processed by the microprocessor without error.

Table 1. APDU command and APDU response for Incrypto CNS and postecert card (hexadecimal representation).

Command	APDU command							APDU response	
	CLA	INS	P1	P2	Lc	Data in CNS (postecert)	Le	Response data	SW
Select file	00	A4	08	00	04 (02)	14 00 81 10 (81 10) This indicates the certificate location of the SCs.	FF	6F 34 81 02 00 00 83 02 81 10 86 0A FF FF FF FF FF FF FF FF FF FF 8A 01 05 8B 18 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF 82 01 38 This indicates file control parameters that consist of a Tag [6F], a Length variable [34] and a Value variable [81 02... 01 38]	90 00
MSE restore	00	22	F3	03	00	-	-	-	90 00
MSE set	00	22	F1	B6	03	83 01 10	-	-	90 00
Get challenge	00	84	00	00	00	-	08	8 bytes random number	90 00
Give challenge	80	86	00	00	08	8 bytes random number	-	-	90 00
Verify	0C	20	00	9A	1D	87 11 01 09 0F 0C EC CC 81 FB 91 F2 3A 45 96 7E 46 91 AE 8E 08 67 1E 69 C0 FA 33 BD 26 [87 11... 91 AE] is a TLV object of CTO structure which consists of a Tag [87], a Length variable [11] and a Value variable [01 09... 91 AE] that contains padding [01] and encoded PIN [09 0F... 91 AE] [8E 08... BD 26] is a TLV object of MAC object structure which consists of a Tag [87], a Length variable [08] and a Value variable [67 1E... BD 26] that indicates result of MAC Computation (see **Figures 5(b)**, **(e)**, **(f)**)	00	-	90 00
Get challenge	00	84	00	00	00	-	08	8 bytes random number	90 00
Give challenge	80	86	00	00	08	8 bytes Random Number	-	-	90 00
PSO_CDS	0C	2A	9E	9A	85	87 79 01 5B 5E 97 05 BD 43 96 B1 FA 8A 5B E5 C1 BC 2A 24 23 ED 5D 51 D4 DA D4 D5 AC F8 70 96 83 75 F7 38 41 00 0F 88 D1 A6 B3 7C F0 6C 1C 41 86 2A 05 1D 52 6B 2B 15 B9 FD AC ED 25 12 AC C0 2A C3 1C 7F 92 65 10 1D 89 52 5D A6 F1 F5 81 CE 6A 0C AE 3A F4 62 C4 ED BC 0E 89 27 1D 25 01 D7 18 5C E1 06 B1 E9 5A 7B 91 E6 D5 5F 47 72 B8 68 C0 B3 B1 50 17 58 BD A6 F5 A7 3E F0 8E 08 EC 51 AA 8D 9F AA 1A 3A [87 79... 3E F0] is a TLV object of CTO structure which consists of a Tag [87], a Length variable [79] and a Value variable [01 5B... 3E F0] that contains padding [01] and encoded Input [5B 5E... 3E F0] [8E 08... 1A 3A] is a TLV object of MAC object structure which consists of a Tag [8E], a Length variable [08] and a Value variable [EC 51... 1A 3A] that indicates result of MAC Computation (see **Figures 5(b)**, **(e)**, **(f)**)	FF	87 89 01 D6 E4 88 36 FE A7 AC F6 D7 46 4D E4 61 F2 E6 E2 4E 3D 04 F8 8B 00 DD B9 90 DD A0 0A D2 93 E5 91 46 C1 26 D1 32 BE 1E EC 03 FB FC 3C 12 FC 9F 16 6F 1A E6 E9 CD CA 42 72 CF 88 9A A5 7E 2D 4F F0 6D EC 11 AC 63 9B 2A 47 70 70 A6 81 59 8C 87 62 5B 45 8F 0A B8 35 23 BC 67 F4 AD 60 AC 73 19 7E C7 94 A2 29 78 45 E8 4A E7 D5 F2 68 68 32 52 BD BB 1C 14 EB 0F E5 9F E3 4C 63 0E E9 D9 7A EC 1D A1 C4 11 1F 13 34 C1 DA 8E 08 42 A9 5C 97 FE B9 07 FD [87 89... C1 DA] is a TLV object of CTO structure which consists of a Tag [87], a Length variable [89] and a Value variable [01 D6... C1 DA] that contains padding [01] and encoded Output [D6 E4... C1 DA] [8E 08... 07 FD] is a TLV object of MAC object structure which consists of a Tag [8E], a Length variable [08] and a Value variable [42 A9... 07 FD] that indicates result of MAC Computation (see **Figure 5(g)**)	90 00

6. Conclusion and Outlook

We have extended OpenSC 0.12.0 to include secure messaging so that the digital signature functionality of SCs can be supported in OpenSC. This will enable us to run extensive test on the interoperability of a wide class of digital signature SCs which are connected with their software applications via a single middleware [7]. We have identified several important security issues that must be addressed in future work. Part of this effort will include combining the OpenSC middleware with a model checker as a "watch-dog" to identify and prevent anomalies. The ultimate goal is to certify the secure interoperability of all SCs integrated into such an environment.

7. Acknowledgements

This project has been supported in part by MIUR under contract PRIN 2008ZE493H.

REFERENCES

[1] International Organization for Standardization (ISO) "Identification Cards—Integrated Circuit Cards Part 4: Organization, Security and Commands for Interchange," International Organization for Standardization Std., Geneva, 2005.

[2] International Organization for Standardization (ISO) "Identification Cards—Integrated Circuit Cards Programming Interfaces—Part 3: Application Programming Interface," International Organization for Standardization Std., Geneva, 2008.

[3] The Common Criteria, "Common Criteria for Information Technology Security Evaluation," Common Criteria Std., 2009. http://www.commoncriteriaportal.org/cc/

[4] The European Committee for Standardization (CEN), "Secure Signature-Creation Devices 'EAL 4+'," European Committee for Standardization (CEN) Std., Brussels, 2004.

[5] M. Talamo, *et al.*, "Robustness and Interoperability Problems in Security Devices," *Proceedings of 4th International Conferences on Information Security and Cryptology*, Beijing, 14-17 December 2008.

[6] M. Talamo, *et al.*, "Verifying Extended Criteria for Interoperability of Security Devices," *Proceedings of 3rd International Symposium on Information Security*, Monterrey, 10-11 November 2008, pp. 1131-1139.

[7] M. Talamo, M. Galinium, C. H. Schunck and F. Arcieri, "Interleaving Command Sequences: A Thread to Secure Smartcard Interoperability," *Proceedings of the 10th International Conference on Information Security and Privacy*, Jakarta, 1-3 December 2011, pp. 102-107.

[8] OpenSC, "OpenSC Tools and Libraries for Smartcard," 2001. http://www.opensc-project.org/opensc

[9] W. Rankl and W. Effing, "Smart Card Handbook," 4th Edition, Wiley, West Sussex, 2010. doi:10.1002/9780470660911

[10] M. Talamo, M. Galinium, C. H. Schunck and F. Arcieri, "Interleaving Commands: A Threat to the Interoperability of Smartcard Based Security Applications," *International Journal of Computer and Communication*, Vol. 6, No. 1, 2012, pp. 76-83.

[11] M. Talamo, M. Galinium, C. H. Schunck and F. Arcieri, "Integrating Secure Messaging into OpenSC," *Proceedings of the 2nd International Conference on Computer and Management*, Wuhan, 9-11 March 2012, pp. 1222-1227.

[12] E. Pucciarelli, "Implementation of Secure Messaging," 2008. http://www.mail-archive.com/opensc-devel@lists.opensc-project.org/msg03034.html

[13] A. Villani, "Incrypto34v2 User and Administrator Guidance," ST. Incard, Marcianese, 2004.

[14] OpenSC. "Supported Hardware (Smart Cards and Usb tokens)," 2011. http://www.opensc-project.org/opensc/wiki/SupportedHardware

[15] O. Kirch and A. Iacono, "pkcs15-Postecert.c," 2004. http://www.opensc-project.org/opensc/browser/OpenSC/src/libopensc/pkcs15-postecert.c

[16] E. Pucciarelli, "pkcs15-Itacns.c," 2008. http://www.opensc-project.org/opensc/browser/OpenSC/src/libopensc/pkcs15-itacns.c

[17] Agencia per L'Italia Digitale, "Certificatori Firma Digitale. Ente Nazionale per la Digitalizzazione della Pubblica Amministrazione," 2011. http://www.digitpa.gov.it/certificatori_firma_digitale

