SCdaemon — A New Framework for Smart Card Access

Werner Koch

g10 Code

<wk@gnupg.org>

February 9, 2007

For GnuPG 2.0 a new framework to access smart cards has been developed. It is based on a daemon process and designed as a replacement for the PC/SC architecture. The problems of PC/SC as and the new solution are described. The new API as well as an introduction on how supporting code for new smart cards needs to be written is given. Further we briefly describe a PKCS#11 connector based on this API.

1 Overview

Smart card support under Unix suffers from non-standard interfaces and non-matured supporting software. Only slowly the landscape is changing but there are still more than just a few problems left. In fact, applications on smart cards are often only barely supported and in most cases pretty buggy. While the GnuPG project features support for an openly specified smart cards for quite some years, the time has come to rethink the existing infrastructure.

With SCdaemon, one of the modules from GnuPG 2.0, a unified interface to smart cards has been developed. SCdaemon allows to support arbitrary smart card applications as long as their specifications are available. Instead of providing a low-level API like PC/SC does, a high-level API with easier use conditions is provided. For example, all PIN related user interaction may be made invisible to the caller.

SCdaemon is directly utilizing the modern CCID standard to access smart card readers. PC/SC will be used as a fallback mechanism with only little functional limitations. To the host application, SCdaemon looks like a smart hardware token with an integrated GUI. Its API is text based and thus easy to debug and audit. An internal interface within SCdaemon provides a standardized interface to any smart card application module to implement the host part of an actual smart card application. These modules allow implementing parts of smart card application on the host to utilize the host's larger resources. As of now, modules for several PKCS#15 applications, DINSIG, OpenPGP and the Telesec card are available.

We briefly describe the history of SCdaemon, the API, the general structure of modules and explain one concrete module. Scute, a PKCS#11 connector will be explained as an example on how the API is used by a real world client.

2 History

While working on the Ägypten project we had the task to support a smart card for use with S/MIME. It was decided that for security reasons it can't go into the gpg-agent (which in general takes care of private key operations), so a separate daemon was written to delegate the task of accessing a smart card on behalf of gpg-agent. This first version was pretty simple and supported only a self-made PKCS#15 formatted card.

With the development of the OpenPGP card, it was clear that the current structure of the SCdaemon was not sufficient and thus we implemented an alternative code path within SCdaemon to support other cards. At this point

the SCdaemon was only able to handle one request from one caller and had to be restarted for a new card. The reason for this was the impossibility of using a threaded design as we do in gpg-agent: The OpenSC library to access PKCS#15 card requires the use of Pthreads and can't be used by applications using a different threading implementation.

While implementing the secure shell agent protocol, we realized that there is no clean way of doing without Pth threads and thus we dropped PKCS#15 support and the use of OpenSC. This allowed us to fix the concurrency problems and implement decent support for the secure shell.

Going without PKCS#15 support for the future was of course not an option. What we did was to implement this support again, this time as a standard SCdaemon application module and without the requirement for the OpenSC library. It turned out that this was far easier than expected: OpenSC is not only a library to access PKCS#15 but it also features all the bits and bytes to actually create PKCS#15 applications on arbitrary cards. Given that we do not want to create smart cards, we could do fine without all of this support and cook the whole PKCS#15 support down to about 3400 SLoC.

SCdaemon also features its own implementaion of a CCID driver. This has mainly historic reasons: At the time of writing we used pretty new card readers and there was no support at all under Unix for these CCID (Chip Card Interface Devices) card readers. The specs for these USB devices are pretty much complete and writing a driver was a straightforward task. After libccid had been developed we had no incentive to move over to PC/SC; actually SCdaemon is designed as a replacement for PC/SC.

3 The New Framework

To see why a new smart card framework makes sense in a world where more and more IT systems are based on Free Software, we need to look at the existing systems.

Today almost every smart card system is based on PC/SC, a system once developed for Microsoft's Windows OS. The commonly used implementation for Unix like systems is based on the efforts of the MUSCLE project and known as *pcsclite*. Figure 1 is a sketch on how it works.

A central daemon (`pcscd`) is running and

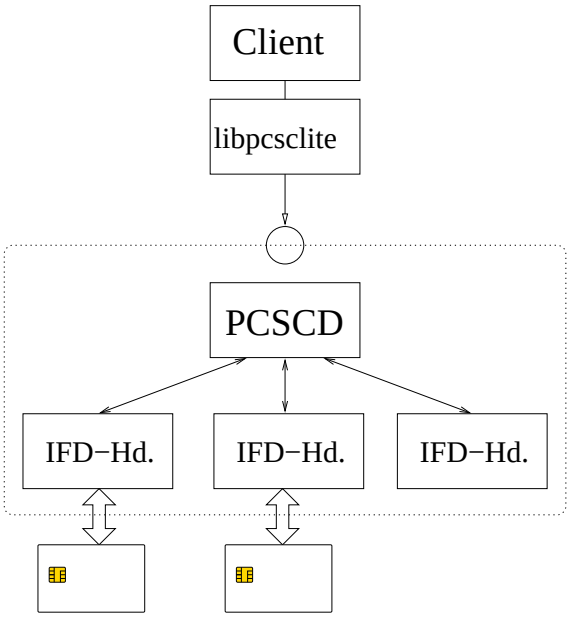

Figure 1: PC/SC architecture

has exclusive control over all smart card readers. The actual drivers for the card readers are called IFD-handlers and are dynamically loaded by `pcscd`. These handlers implement the reader specific parts of the card access including timing and the communication protocol selection. `pcscd` mainly manages the IFD-handler and provides the access point for applications. Applications usually don't access `pcscd` directly but make use of the convenience library `libpcsclite` which basically takes care of connecting to `pcscd` via a local socket.

There is actually nothing wrong with this architecture if we understand that it has been designed for the proprietary world: All the card reader vendors and sometimes also the smart card vendors provide their own code to access their readers. Despite that these drivers are not hardware drivers[1] but protocol stacks, they are kept close to hide details of the readers. Fortunately most card readers have meanwhile been reverse engineered and free IFD-handlers are available. With the availability of CCID based readers the situation has changed to the better as the vendors mostly comply with this standard and only try not to disclose some features. Another problem is that a lot of functionality is duplicated by each handler and thus problems

[1] the actual I/O is usually done via the standard RS-232 or USB drivers of the OS.

with smart card access very often depends on the card reader and the used handler.

The upshot is that the entire PC/SC architecture is only useful if there is a need to keep the specifications of the card readers secret. Because we want Free Software on our computers and in particular for security related applications, we can do much simpler than PC/SC. Figure 2 sketches the new framework.

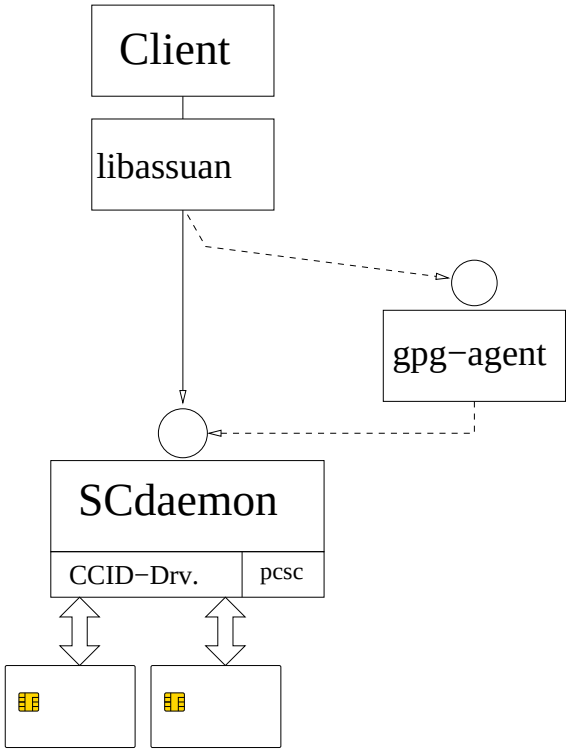

Figure 2: SCdaemon architecture

The basic difference is that SCdaemon does not anymore distinguish between IFD-handlers and a central daemon. SCdaemon itself takes directly care of call card readers and their applications. This is done by using application modules at the core of SCdaemon and a simple but sufficient layer to access the readers. SCdaemon takes advantage of the modern CCID reader and directly implements such a protocol stack (aka driver or handler). Optionally it allows to make use of an existing PC/SC framework; this is however only needed to use non-CCID drivers.

Another change in contrast to PC/SC is that SCdaemon is not a system service but usually run for each user. The rationale for this is that card readers don't make sense on a multi-user system because physical access to the reader is always required to take advantage of personalized smart cards.

SCdaemon may be used directly by a client application[2] in which case the client needs to provide a facility to enter a PIN. In almost all cases SCdaemon will be used via gpg-agent, the central part of the GnuPG 2.0 system for managing private keys. In this mode of operation, gpg-agent intercepts requests from SCdaemon for a PIN and uses its own mechanism (the Pinentry) to transparently ask for the PIN.

4 Public API

The API to SCdaemon is based on the Assuan protocol which provides an easy to audit interface. It is similar to a lot of well established network protocols like SMTP or FTP and uses clear text commands and replies. It is client/server based where SCdaemon implements an Assuan server.

The `gpg-connect-agent` tool may be used to directly communicate with SCdaemon. This tool provides all the features required to write scripts accessing SCdaemon. However, in almost all cases SCdaemon is not directly used but called on behalf of gpg-agent. gpg-agent even provides a mechanism to call arbitrary SCdaemon commands while at the same time provides the often required callback functionality for entering a PIN. To use this mode, simply run gpg-connect-agent and prefix all SCdaemon commands with "`SCD` ".

What follows is a description of the major commands implemented by SCdaemon.

[2] the convenience library libassuan makes this easier.

4.1 Command SERIALNO

The basic purpose of this command is to return the serial-number of the card. However it has a special role in SCdaemon as it is used to start talking to a card. To accomplish this it will check for the presence of a card and automatically initializes the connection to a new card. This is in contrast to most other commands which would simply return an error if the card is missing or has been replaced.

On success the serial number is returned using an Assuan status line; see the command LEARN for a description.

The serial number as used by SCdaemon is usually the hex encoded value identified by the 0x5A Tag in the GDO file (FID=0x2F02). Some cards may use different ways to store a serial number, however this is the default method which should work with most ISO7816 cards.

Optionally the command accepts an argument to select a specific application and not try to select the card application using a default strategy. As of GnuPG 2.0.2 the following applications are supported:

openpgp The OpenPGP card application. See http://g10code.com/docs/openpgp-card-1.1.pdf.

dinsig The DINSIG card application. This is an application as described in the German draft standard *DIN V 66291-1*. It is intended to be used by cards supporting the German signature law and its bylaws (SigG and SigV).

p15 A generic PKCS#15 card application. The current implementaion has tweaks for a few commonly used cards.

nks The Telesec NetKey card application. This is the main application of the Telesec cards. It is a superset of "dinsig"

4.2 Command LEARN

This is used to read in one shot all useful information from the card.

Unless the option `--force` is used with this command, SCdaemon will ask back to allow the client to cancel the operation in case he already knows enough about the particular card:

```
INQUIRE KNOWNCARDP HEXSTRING_WITH_SERIALNUMBER TIMESTAMP
```

The client should send an `END` if the processing should go on normally or a `CANCEL` to force the function to terminate with a cancel error message.

The response of this command is a list of status lines. Here is a description of common status lines. Future applications may add other status lines, some are only emitted for certain cards.

APPTYPE Returns the type of the application.

Synopsis:
```
S APPTYPE APPTYPE
```

The returned APPTYPE is one of the strings which may be passed as option to the SERIALNO command.

SERIALNO Returns the serial number of the card.

Synopsis:
```
S SERIALNO SERIAL_NUMBER RESERVED
```

The returned SERIAL_NUMBER is a hex encoded string. RESERVED is currently 0 but should be ignore. Also any other extra parameters should be ignored to achieve best future compatibility.

KEYPAIRINFO Returns information about a keypair.

Synopsis: `S KEYPAIRINFO HEXSTRING_WITH_KEYGRIP HEXSTRING_WITH_ID`

This describes information about the public key pairs available on the card. The first parameter is the so-called keygrip which depicts the actual key and is independent of any protocol. The second parameter is an internal ID to identify that key pair. This status line is send for each key pair.

CERTINFO Returns information about a certificate

Synopsis: `S CERTINFO CERTTYPE HEXSTRING_WITH_ID`

Returns information about all certificates stored on the card. There is one line for each certificate.

Where CERTTYPE is a number indicating the type of certificate:

0 Unknown type.

100 Regular X.509 certificate.

101 Trusted X.509 certificate.

102 Useful X.509 certificate.

110 Root CA certificate as used by DINSIG.

KEY-FPR Returns the fingerprint of the key.

Synopsis: `S KEY-FPR NO HEXSTRING`

For OpenPGP cards this returns the stored fingerprints of the keys. This may be used check whether a key is available on the card. NO may be 1, 2 or 3.

CA-FPR Returns the CA fingerprint field.

Synopsis: `S CA-FPR NO HEXSTRING`

Similar to above, these are the fingerprints of keys assumed to be ultimately trusted.

DISP-NAME Synopsis: `S DISP-NAME NAME_OF_CARD_HOLDER`

The name of the card holder as stored on the card; percent escaping takes place, spaces are encoded as '+'

All of these values (and more) may also be retrieved individually using the GETATTR command.

4.3 Command READCERT

This command is used to return a certificate.

Synopsis: `READCERT HEXIFIED_CERTID`

The certificate to read is identified by the HEXIFIED_CERTID as available through the CERTINFO status line.

4.4 Command READKEY

This command is used to return a public key.

Synopsis: `READKEY HEXIFIED_CERTID`

Return the public key for the given certificate or key ID as an S-expression.

4.5 Command SETDATA

This command is used to tell SCdaemon the actual data to be signed or decrypted.

Synopsis: `SETDATA HEXSTRING`

The data must be given as one string in hex notation.

4.6 Command PKSIGN

This command performs a signing operation on the smart card.

Synopsis: `PKSIGN [--hash=ALGONAME] KEYID`

If ALGONAME has been given it is one of `sha1`, `rmd160` or `md5`. The default is `sha1`. Setting this hash algorithm is required for most smart cards and it is strongly suggested to always set the correct has algorithm used to hash the to be signed data.

The KEYID is the hexified ID of the key to be used. It may have been retrieved using the LEARN command.

Before this command is used, the client must have issued the SETDATA command to tell what data is to be signed.

If the signing succeeds the result is returned using Assuan data (D) lines. Note that these data lines may contain binary characters.

4.7 Command PKDECRYPT

This command is used to decrypt data with a smart card.

Synopsis: `PKDECRYPT KEYID`

As with the PKSIGN command, KEYID is the hexified ID of the key to be used for the decryption. It may have been retrieved using the LEARN command.

Before this command is used, the client must have issued the SETDATA command to tell what data is to be decrypted.

If the decryption succeeds the result is returned using Assuan data (D) lines. Note that these data lines may contain binary characters.

4.8 Command GETATTR

Read an attribute's value.

Synopsis: `GETATTR NAME`

This command is used to retrieve data from a smart card. The allowed values for NAME depend on the currently selected smart card application. NAME must be percent and '+' escaped. The value is returned through a status message — see the LEARN command for details.

A few attributes are only available using this command. Here is a list of the names:

$AUTHKEYID Returns the key ID of the authentication key.

Synopsis: `S $AUTHKEYID KEYID`

This attribute is used by the Secure Shell agent code in gpg-agent to determine whether a card supports Secure Shell keys and what key ID to use for it. The returned KEYID may then be used with the PKAUTH command. For example, an OpenPGP card returns the string `OpenPGP.3`.

$DISPSERIALNO Return a human friendly serial number.

Synopsis: `S $DISPSERIALNO STRING`

If this command succeeds, the client should present STRING to the user as the serial number of the card. That value should however never be used for internal computations.

The rationale for this extra serial number is that the serial number is often printed in an abbreviated and special formatted form on the card. Using this command will make it easier for the user to understand the serial number.

4.9 Command SETATTR

Update an attribute's value.

Synopsis: `SETATTR NAME VALUE`

This command is used to store or update data on a smart card. The allowed names and values are depend on the currently selected smart card application. NAME and VALUE must be percent and '+' escaped. The total length of the value is limited by the maximum length of an Assuan line (about 1000 characters). This should not be a problem as attribute values tend to be quite short.

Note, that a PIN will be requested when updating most attributes. See the corresponding setattr function of the selected application `app-*.c` for details.

4.10 Command WRITEKEY

Write a key to a card.

Synopsis: `WRITEKEY [--force] KEYID`

This command is used to store a secret key on a smart card. The allowed values for KEYID depend on the currently selected smart card application. The actual key data is requested using the inquiry `KEYDATA` and needs to be provided without any protection (i.e. in the clear). The key data is expected to be the usual canonical encoded S-expression.

Using the `--force` will replace a key already existing under this KEYID.

Note, that a PIN will be requested in most cases.

4.11 Command GENKEY

Generate a new key on-card.

Synopsis: `GENKEY [--force] NO`

Generate a key on-card and store it under the card application specific number NO. Returned values are application specific. The public key may later be retrieved using the READKEY command.

Using the `--force` will replace a key already existing under the number NO.

4.12 Command RANDOM

Return random bytes generate on-card.

Synopsis: `RANDOM N`

Retrieves N bytes of random from the card and returns it as Assuan data lines. Not all cards support such a command. Warning: Excessive use of this command may reduce the life span of the card.

4.13 Command PASSWD

Change PINs.

Synopsis: `PASSWD [--reset] CHVNO`

Change the PIN or reset the retry counter of the card holder verification vector number CHVNO. The old and new PIN is queried using the usual PIN callback inquiry (and automagically handled if called via `gpg-agent`).

4.14 Command CHECKPIN

Perform an ISO 7816 VERIFY operation.

Synopsis: `CHECKPIN IDSTR`

Perform a VERIFY operation without doing anything else. This may be used to early initialize the PIN cache before starting a long lasting operation. The details are highly application dependent:

openpgp Does a simple verify operation for CHV1 and CHV2, so that further operations won't ask for CHV2. This allows for a cheap check on the PIN. For example to test whether the PIN entry system (GUI etc.) works as expected. In case of a failure only the regular CHV will get blocked and not the Admin PIN. IDSTR is the card's serial number in hex notation; an optional fingerprint part will get ignored.

If IDSTR is suffixed with the literal string `[CHV3]`, the Admin PIN (CHV3) is checked if and only if the retry counter is still at 3.

4.15 Command RESTART

Do a warm reset of the connection.

Synopsis: `RESTART`

Restarts the current connection; this is a kind of warm reset. It deletes and the context used by this connection between the client and SCdaemon but does not actually reset the card.

4.16 Command APDU

Send a verbatim APDU to the card.

Synopsis: `APDU [--atr] [--more] [HEXSTRING]`

Send an APDU to the current reader. This command bypasses the high level functions and sends the data directly to the card. HEXSTRING is expected to be a proper APDU. If HEXSTRING is not given, no commands are send to the card. However the command will implicitly check whether the card is ready for use.

Using the option `--atr` returns the ATR of the card as a status message before any data.

Example: `S CARD-ATR 3BFA1300FF813180450031C173C00100009000B1`

Using the option `--more` handles the card status word MORE_DATA (61xx) and concatenate all responses to one block.

4.17 Communication Example

To show how all these commands work together we will give an example by looking at the debug output of SCdaeman (debug 1024) while we are running the command

```
fortune | gpg2 -sbau dd9jn
```

to sign a random text:

```
1   DBG: <- RESTART
2   DBG: -> OK
3   DBG: <- SETDATA B09C4147497F5F55FB8550E7161B615B16514622
4   DBG: -> OK
5   DBG: <- PKSIGN D2760001240101010001000003470000/7B96D39[...]CE0C630
6   DBG: -> INQUIRE POPUPKEYPADPROMPT ||Please [...] keypad%0A[sigs done: xxx]
7   DBG: <- END
8   DBG: -> INQUIRE DISMISSKEYPADPROMPT
9   DBG: <- END
10  DBG: -> [ 44 20 04 bd 1e ed 7e 80 70 9f 8f 56 ...(122 bytes skipped) ]
11  DBG: -> OK
```

The first 2 lines are used to reset the connection. Line 3 tells SCdaemon what data to sign. As usual this is a SHA-1 hash of the text we are signing. At line 5 SCdaemon received the actual signing command along with the card's serial number (which is stored in gpg's keyring) and the fingerprint of the key to use.

While processing this command SCdaemon realizes that the PIN of the card is required. Because the attached card reader features a PIN-pad, there is no need to ask back for the PIN, however the user needs to be notified that he should move his fingers over to the readers keypad and thus the caller is asked at line 6 to show a popup window. After that has been acknowledged by the caller (line 7), the actual operation takes place. After that has been finished the caller is asked to dismiss the popup window. Finally at line 10, SCdaemon sends back the actual signature so that gpg can construct a valid OpenPGP message out of it.

5 Application Modules

Given that there are lot of smart card based applications in use and that even PKCS#15 is not a suitable standard to address all features of cards, it is required to write custom code for each smart card application.

"Applications" as used in this context are either the applications implemented on the card or the code in SCdaemon used to interface with these applications. In general it is not relevant to the user to distinguish between the part of the application on-card and the part on-host (i.e. as implemented by SCdaemon). Smart cards have very limited computing resources and thus they should implement only the part of the application which needs to be on-card for security or practical purposes.

In the proprietary world the on-host code is often implemented in the so-called "driver" and makes up a lot of the card application's functionality. In the Free World we won't have access to that part of the application and thus we need to implement it in our software.[3]

5.1 General Structure of an Application

Application modules in SCdaemon are implemented in C and directly linked into it. There is no plugin system because it is not necessary as we have access to the source code and new smart card applications are created rarely and thus updating SCdaemon to a newer version is easier than to maintain a wealth of application plugins. It also allows changes to the internal API without introducing to big hassles.

The application modules may use any public function from SCdaemon but it is suggested that only those functions are used which have been written with application module use in mind. 4 Groups of functions are to be mentioned:

- ISO-7816 function calls in `iso7816.c` as high-level calls to the standard commands of smart cards. These functions provide an abstract interface so that there is no need to have a deeper knowledge of how APDUs (the messages exchanges between a host and a card) are to be constructed or parsed. In addition to the standard functions a few variants are available to support card readers with an integrated keypad. A function call to test whether a PIN pad can be used is also available.
- Direct card reader access by using the APDU interface (`apdu.c`). These are the low level function as used by the ISO-7816 functions but are available to application modules too. They are useful to send special crafted APDUs or ISO commands without a counterpart in `iso7816.c`. Unless really needed they should be avoided as they break the usual layering.
- Application helper functions (`app-help.c`) implement commonly used patterns.
- General utility functions to extend libc and as wrapper around malloc. These are implemented by the GnuPG convenience library `libcommon-pth`.

In addition a module may make use of the two libraries Libgcrypt and Libksba for general cryptographic operations. It should however take care that SCdaemon is build on the GNU Pth user land thread library and thus long running operations and file access needs to be coordinated with Pth (e.g. by using the Pth variants of the standard functions or to use `pth_yield`).

It is further important to realize that for concurrent access to a card, a single card operation should not take too long. An internal lock is used to prevent simultaneous access of several clients to the same card. For the purpose of this it is useful to imagine that the code in the application module would actually run on the card and other operations to the card are queued up.

[3] Using proprietary drivers is not an option for a Free Software based system. Although the smart card is in most cases proprietary, it provides a clear interface and has no direct access to host resources.

The interface to the application module consists of one public function and a couple of function pointers. That public function should be named `app_select_FOO` for an application named `foo`. Its prototype needs to be put into `app-common.h`. To actually make use of the module you would modify `app.c`:

```
    if (err && is_app_allowed ("openpgp")
            && (!name || !strcmp (name, "openpgp")))
      err = app_select_openpgp (app);
    if (err && is_app_allowed ("nks") && (!name || !strcmp (name, "nks")))
      err = app_select_nks (app);
    if (err && is_app_allowed ("p15") && (!name || !strcmp (name, "p15")))
      err = app_select_p15 (app);
    if (err && is_app_allowed ("dinsig") && (!name || !strcmp (name, "dinsig")))
      err = app_select_dinsig (app);
    if (err && name)
      err = gpg_error (GPG_ERR_NOT_SUPPORTED);
```

Add your select function in a similar way to that list. It is suggested to put it at the end of the list so that it extends SCdaemon only and does not modify its behavior: If no specific application has been requested, the order of these tests defines which application will be used in case several are implemented on the card.

The select function is responsible to setup the function pointer to its actual code. setting this function pointers to `NULL` is explicitly allowed to declare that the command is not supported.

The functions a module may implement are:

deinit is used to deallocate resources allocated at initialization time of the module (i.e. by the select function). It is only called after all clients have given up access to the card and the card has been removed or another application has been selected.

learn_status is called while processing the LEARN command.

readcert is called while processing the READCERT command.

readkey is called while processing the READKEY command.

getattr is called while processing the GETATTR command.

setattr is called while processing the SETATTR command.

sign is called while processing the PKSIGN command.

auth is called while processing the PKAUTH command.

decipher is called while processing the PKDECRYPT command.

writekey is called while processing the WRITEKEY command.

genkey is called while processing the GENKEY command.

changepin is called while processing the PASSWD command.

check_pin is called while processing the CHECKPIN command.

5.2 Example Application

To show how an application is build we will now look at a real example. This is the application used for the Telesec cards. It is actually a pretty simple one, which allows only encryption and signing and has no advanced features. The name of this application is NKS and thus the file name to look at is `app-nks.c`. Here is it with boilerplate, white space and some comments removed:

```
static struct
{
  int fid;       /* File ID. */
  int certtype;  /* Type of certificate or 0 if it is not a certificate. */
  int iskeypair; /* If true has the FID of the correspoding certificate. */
  int issignkey; /* True if file is a key usable for signing. */
  int isenckey;  /* True if file is a key usable for decryption. */
} filelist[] = {
  { 0x4531, 0,  0xC000, 1, 0 },
  { 0xC000, 101 },
  { 0x4331, 100 },
  { 0x4332, 100 },
  { 0xB000, 110 },
  { 0x45B1, 0,  0xC200, 0, 1 },
  { 0xC200, 101 },
  { 0x43B1, 100 },
  { 0x43B2, 100 },
  { 0, 0 }
};
```

Obviously the module needs to know some bits of the card's application. In particular information on the use of the files is required. Conveniently this is hold in an array.

```
/* Read the file with FID, assume it contains a public key and return
   its keygrip in the caller provided 41 byte buffer R_GRIPSTR. */
static gpg_error_t
keygripstr_from_pk_file (int slot, int fid, char *r_gripstr)
{
  gpg_error_t err;
  unsigned char grip[20];
  unsigned char *buffer[2];
  size_t buflen[2];
  gcry_sexp_t sexp;
  int i;

  err = iso7816_select_file (slot, fid, 0, NULL, NULL);
  if (err)
    return err;
  err = iso7816_read_record (slot, 1, 1, 0, &buffer[0], &buflen[0]);
  if (err)
    return err;
  err = iso7816_read_record (slot, 2, 1, 0, &buffer[1], &buflen[1]);
  if (err)
    {
      xfree (buffer[0]);
      return err;
    }
```

This is a helper function to compute the keygrip from a file containing the public key. SLOT is used to identify the card reader used, FID is the File ID of the file with the public key and R_GRIPSTR is a buffer to receive the binary 20 byte keygrip.

What follows is selecting the file and reading the first two records of that file (cards commonly store data as structured files consisting of variable length records).

```
  for (i=0; i < 2; i++)
    {
      /* Check that the value appears like an integer encoded as
         Simple-TLV.  We don't check the tag because the tests cards I
         have use 1 for both, the modulus and the exponent - the
         example in the documentation gives 2 for the exponent. */
      if (buflen[i] < 3)
        err = gpg_error (GPG_ERR_TOO_SHORT);
      else if (buffer[i][1] != buflen[i]-2 )
        err = gpg_error (GPG_ERR_INV_OBJ);
    }

```

```
95      if (!err)
96        err = gcry_sexp_build (&sexp, NULL,
97                               "(public-key (rsa (n %b) (e %b)))",
98                               (int)buflen[0]-2, buffer[0]+2,
99                               (int)buflen[1]-2, buffer[1]+2);
100     xfree (buffer[0]);
101     xfree (buffer[1]);
102     if (err)
103       return err;
```

Here we convert the public key to a format suitable to be passed to Libgcrypt. Libgcrypt uses an S-expression based format to represent data used with public key cryptography.

```
106     if (!gcry_pk_get_keygrip (sexp, grip))
107       {
108         err = gpg_error (GPG_ERR_INTERNAL); /* i.e. RSA not supported by
109                                                libgcrypt. */
110       }
111     else
112       {
113         for (i=0; i < 20; i++)
114           sprintf (r_gripstr+i*2, "%02X", grip[i]);
115       }
116     gcry_sexp_release (sexp);
117     return err;
118   }
```

Finally a Libgcrypt function is called to compute the keygrip.

Now for some real stuff. We hack up the code to implement the LEARN command:

```
122   static gpg_error_t
123   do_learn_status (app_t app, ctrl_t ctrl)
124   {
125     gpg_error_t err;
126     char ct_buf[100], id_buf[100];
127     int i;
128
129     /* Output information about all useful objects. */
130     for (i=0; filelist[i].fid; i++)
131       {
```

As most functions in an application module, this one takes the APP argument which represents the current context of the module. Because we want to return some information it also takes a CTRL argument to have access to state information of the connection. The code works by looping over the file information list (cf. line 36).

```
132         if (filelist[i].certtype)
133           {
134             size_t len;
135
136             len = app_help_read_length_of_cert (app->slot,
137                                                 filelist[i].fid, NULL);
138             if (len)
139               {
140                 /* FIXME: We should store the length in the application's
141                    context so that a following readcert does only need to
142                    read that many bytes. */
143                 sprintf (ct_buf, "%d", filelist[i].certtype);
144                 sprintf (id_buf, "NKS-DF01.%04X", filelist[i].fid);
145                 send_status_info (ctrl, "CERTINFO",
146                                   ct_buf, strlen (ct_buf),
147                                   id_buf, strlen (id_buf),
148                                   NULL, (size_t)0);
149               }
150           }
```

Now, if the file is flagged as one holding a certificate we call a standard helper function to select a file, read it and return the length of the certificate. This is used to check whether a certificate is actually stored there. If that is the case a CERTINFO status line is prepared and send back to the client.

```
151         else if (filelist[i].iskeypair)
152           {
153             char gripstr[40+1];
154
155             err = keygripstr_from_pk_file (app->slot, filelist[i].fid, gripstr);
156             if (err)
157               log_error ("can't get keygrip from FID 0x%04X: %s\n",
158                          filelist[i].fid, gpg_strerror (err));
159             else
160               {
161                 sprintf (id_buf, "NKS-DF01.%04X", filelist[i].fid);
162                 send_status_info (ctrl, "KEYPAIRINFO",
163                                   gripstr, 40,
164                                   id_buf, strlen (id_buf),
165                                   NULL, (size_t)0);
166               }
167           }
168       }
169
170     return 0;
171   }
```

If the file is flagged in our list as one to hold a public key, we use the helper function from line 60 to get the key's keygrip. An error may occur if no key is stored (we print an error diagnostic then). If the keygrip could be computed, we construct a KEYPAIRINFO status line and send that back to the client. Well, that was all pretty easy.

The next thing we want to implement is the code to actually read a certificate given the ID as retrieved using the LEARN command.

```
176   /* Read the certificate with id CERTID (as returned by learn_status in
177      the CERTINFO status lines) and return it in the freshly allocated
178      buffer put into CERT and the length of the certificate put into
179      CERTLEN. */
180   static gpg_error_t
181   do_readcert (app_t app, const char *certid,
182                unsigned char **cert, size_t *certlen)
183   {
184     int i, fid;
185     gpg_error_t err;
186     unsigned char *buffer;
187     const unsigned char *p;
188     size_t buflen, n;
189     int class, tag, constructed, ndef;
190     size_t totobjlen, objlen, hdrlen;
191     int rootca = 0;
192
193     *cert = NULL;
194     *certlen = 0;
195     if (strncmp (certid, "NKS-DF01.", 9) )
196       return gpg_error (GPG_ERR_INV_ID);
197     certid += 9;
198     if (!hexdigitp (certid) || !hexdigitp (certid+1)
199         || !hexdigitp (certid+2) || !hexdigitp (certid+3)
200         || certid[4])
201       return gpg_error (GPG_ERR_INV_ID);
202     fid = xtoi_4 (certid);
203     for (i=0; filelist[i].fid; i++)
204       if ((filelist[i].certtype || filelist[i].iskeypair)
205           && filelist[i].fid == fid)
206         break;
```

```
  if (!filelist[i].fid)
    return gpg_error (GPG_ERR_NOT_FOUND);
```

The function is called with the CERTID and two parameters to return the certificate (CERT and CERTLEN). Because this is more or less directly by the client, some simple checks are first done, so that an appropriate error code can be returned. We check that CERTID starts off with the common prefix used by this module for certificates (`NKS-DF01`), it is followed by four hex digits (the file ID) and actually listed in our file list.

Let's continue:

```
  /* If the requested objects is a plain public key, redirect it to
     the corresponding certificate.  The whole system is a bit messy
     because we sometime use the key directly or let the caller
     retrieve the key from the certificate.  The rationale for
     that is to support not-yet stored certificates. */
  if (filelist[i].iskeypair)
    fid = filelist[i].iskeypair;


  /* Read the entire file.  fixme: This could be optimized by first
     reading the header to figure out how long the certificate
     actually is. */
  err = iso7816_select_file (app->slot, fid, 0, NULL, NULL);
  if (err)
    {
      log_error ("error selecting FID 0x%04X: %s\n", fid, gpg_strerror (err));
      return err;
    }

  err = iso7816_read_binary (app->slot, 0, 0, &buffer, &buflen);
  if (err)
    {
      log_error ("error reading certificate from FID 0x%04X: %s\n",
                 fid, gpg_strerror (err));
      return err;
    }

  if (!buflen || *buffer == 0xff)
    {
      log_info ("no certificate contained in FID 0x%04X\n", fid);
      err = gpg_error (GPG_ERR_NOT_FOUND);
      goto leave;
    }
```

This is some code to actually read the certificate from the file and to check that there is a certificate. The card stores FF bytes for unused or deleted data. As described in the comment is is a bit more complicate than a straightforward implementation. What follows is a lot of ugly code to parse the ASN.1 structure to figure out what kind of certificate we have.

```
  /* Now figure something out about the object. */
  p = buffer;
  n = buflen;
  err = parse_ber_header (&p, &n, &class, &tag, &constructed,
                          &ndef, &objlen, &hdrlen);
  if (err)
    goto leave;
  if ( class == CLASS_UNIVERSAL && tag == TAG_SEQUENCE && constructed )
    ;
  else if ( class == CLASS_UNIVERSAL && tag == TAG_SET && constructed )
    rootca = 1;
  else
    return gpg_error (GPG_ERR_INV_OBJ);
  totobjlen = objlen + hdrlen;
  assert (totobjlen <= buflen);

```

```
260     err = parse_ber_header (&p, &n, &class, &tag, &constructed,
261                             &ndef, &objlen, &hdrlen);
262     if (err)
263       goto leave;
264
265     if (rootca)
266       ;
267     else if (class == CLASS_UNIVERSAL && tag == TAG_OBJECT_ID && !constructed)
268       {
269         const unsigned char *save_p;
270
271         /* The certificate seems to be contained in a userCertificate
272            container.  Skip this and assume the following sequence is
273            the certificate. */
274         if (n < objlen)
275           {
276             err = gpg_error (GPG_ERR_INV_OBJ);
277             goto leave;
278           }
279         p += objlen;
280         n -= objlen;
281         save_p = p;
282         err = parse_ber_header (&p, &n, &class, &tag, &constructed,
283                                 &ndef, &objlen, &hdrlen);
284         if (err)
285           goto leave;
286         if ( !(class == CLASS_UNIVERSAL && tag == TAG_SEQUENCE && constructed) )
287           return gpg_error (GPG_ERR_INV_OBJ);
288         totobjlen = objlen + hdrlen;
289         assert (save_p + totobjlen <= buffer + buflen);
290         memmove (buffer, save_p, totobjlen);
291       }
292
293     *cert = buffer;
294     buffer = NULL;
295     *certlen = totobjlen;
296
297    leave:
298     xfree (buffer);
299     return err;
300   }
```

If no errors occurred, we were finally able to store the certificate at the caller provided address CERT and its length at CERTLEN. Note that the buffer has been allocated while reading the data at line 229.

The next in turn is a helper function to do a verify operation. This helper is called later right before a decrypt or signing operation. Do make things easier for the user, it caches the fact that a PIN has successfully been verified. This is of course an application specific semantic as there is no standard on when a verification is actually needed. Some card applications require a verify operation ("presenting the CHV" in card parlance) right before every signing operation to get a consent from the user. However, our card keeps the state of the verification until the next card reset.

```
303   /* Verify the PIN if required.  */
304   static gpg_error_t
305   verify_pin (app_t app,
306               gpg_error_t (*pincb)(void*, const char *, char **),
307               void *pincb_arg)
308   {
309     /* Note that force_chv1 is never set but we do it here anyway so
310        that other applications may reuse this function.  For example it
311        makes sense to set force_chv1 for German signature law cards.
312        NKS is very similar to the DINSIG draft standard. */
313     if (!app->did_chv1 || app->force_chv1 )
314       {
```

Note that we use standard variables available in the application context APP to mark that we have verified the CHV.

```
        char *pinvalue;
        int rc;

        rc = pincb (pincb_arg, "PIN", &pinvalue);
        if (rc)
          {
            log_info ("PIN callback returned error: %s\n", gpg_strerror (rc));
            return rc;
          }
```

To query the PIN we use the callback function PINCB which has been passed as an argument to us. PINCB_ARG is a common pattern to convey a context to the callback. If the callback function failed, we immediately leave this function with an error.

Before we send the verify command to the card, we do some basic checks on the PIN. This is no security issues as the length constraints of the PIN are specified for the card and no attack would use too short or too long PINs. In fact it helps to isolate faults from ill running code in the callback.

```
        /* The following limits are due to TCOS but also defined in the
           NKS specs. */
        if (strlen (pinvalue) < 6)
          {
            log_error ("PIN is too short; minimum length is 6\n");
            xfree (pinvalue);
            return gpg_error (GPG_ERR_BAD_PIN);
          }
        else if (strlen (pinvalue) > 16)
          {
            log_error ("PIN is too large; maximum length is 16\n");
            xfree (pinvalue);
            return gpg_error (GPG_ERR_BAD_PIN);
          }

        /* Also it is possible to use a local PIN, we use the gloabl
           PIN for this application.  */
        rc = iso7816_verify (app->slot, 0, pinvalue, strlen (pinvalue));
        if (rc)
          {
            if ( gpg_error (rc) == GPG_ERR_USE_CONDITIONS )
              log_error (_("the NullPIN has not yet been changed\n"));
            else
              log_error ("verify PIN failed\n");
            xfree (pinvalue);
            return rc;
          }
        app->did_chv1 = 1;
        xfree (pinvalue);
      }

    return 0;
  }
```

Finally the verify command is send to the card. We do check for the error and distinguish between the special TCOS[4] error code for a so-called NullPIN and a wrong password (which might also be due to a blocked card). If the verify succeeded we mark that in the application context.

And now we can actually do something with the card: Let's create a digital signature:

```
  static gpg_error_t
```

[4]This is the card OS used with the NKS card.

```
do_sign (app_t app, const char *keyidstr, int hashalgo,
         gpg_error_t (*pincb)(void*, const char *, char **),
         void *pincb_arg,
         const void *indata, size_t indatalen,
         unsigned char **outdata, size_t *outdatalen )
{
```

This function is the main worker of the PKSIGN command. Aside of the application context APP it is called with the string KEYIDSTR to identify the key to be used, a callback to allow asking for the PIN (PINCB and PINCB_ARG), the data to be signed (INDATA with length given at INDATALEN). On success the function stores the address of a new buffer with the resulting signature at OURDATA and the length of the result at OUTDATALEN.

We continue with a couple of argument checks:

```
  static unsigned char sha1_prefix[15] = /* Object ID is 1.3.14.3.2.26 */
    { 0x30, 0x21, 0x30, 0x09, 0x06, 0x05, 0x2b, 0x0e, 0x03,
      0x02, 0x1a, 0x05, 0x00, 0x04, 0x14 };
  static unsigned char rmd160_prefix[15] = /* Object ID is 1.3.36.3.2.1 */
    { 0x30, 0x21, 0x30, 0x09, 0x06, 0x05, 0x2b, 0x24, 0x03,
      0x02, 0x01, 0x05, 0x00, 0x04, 0x14 };
  int rc, i;
  int fid;
  unsigned char data[35];   /* Must be large enough for a SHA-1 digest
                               + the largest OID _prefix above. */

  if (!keyidstr || !*keyidstr)
    return gpg_error (GPG_ERR_INV_VALUE);
  if (indatalen != 20 && indatalen != 16 && indatalen != 35)
    return gpg_error (GPG_ERR_INV_VALUE);

  /* Check that the provided ID is vaid.  This is not really needed
     but we do it to enforce correct usage by the caller. */
  if (strncmp (keyidstr, "NKS-DF01.", 9) )
    return gpg_error (GPG_ERR_INV_ID);
  keyidstr += 9;
  if (!hexdigitp (keyidstr) || !hexdigitp (keyidstr+1)
      || !hexdigitp (keyidstr+2) || !hexdigitp (keyidstr+3)
      || keyidstr[4])
    return gpg_error (GPG_ERR_INV_ID);
  fid = xtoi_4 (keyidstr);
  for (i=0; filelist[i].fid; i++)
    if (filelist[i].iskeypair && filelist[i].fid == fid)
      break;
  if (!filelist[i].fid)
    return gpg_error (GPG_ERR_NOT_FOUND);
  if (!filelist[i].issignkey)
    return gpg_error (GPG_ERR_INV_ID);
```

and then prepare the data for consumption by the card:

```
  /* Prepare the DER object from INDATA. */
  if (indatalen == 35)
    {
      /* Alright, the caller was so kind to send us an already
         prepared DER object.  Check that it is waht we want and that
         it matches the hash algorithm. */
      if (hashalgo == GCRY_MD_SHA1 && !memcmp (indata, sha1_prefix, 15))
        ;
      else if (hashalgo == GCRY_MD_RMD160 && !memcmp (indata, rmd160_prefix,15))
        ;
      else
        return gpg_error (GPG_ERR_UNSUPPORTED_ALGORITHM);
      memcpy (data, indata, indatalen);
    }
  else
    {
```

```
      if (hashalgo == GCRY_MD_SHA1)
        memcpy (data, sha1_prefix, 15);
      else if (hashalgo == GCRY_MD_RMD160)
        memcpy (data, rmd160_prefix, 15);
      else
        return gpg_error (GPG_ERR_UNSUPPORTED_ALGORITHM);
      memcpy (data+15, indata, indatalen);
    }
```

As the last step we run the actual signing command:

```
  rc = verify_pin (app, pincb, pincb_arg);
  if (!rc)
    rc = iso7816_compute_ds (app->slot, data, 35, outdata, outdatalen);
  return rc;
}
```

Note, that before the signing command is called we call `verify_pin` to make sure that we have access to the signing command. The signature is then directly created by the ISO 7816 function call.

The NKS card also allows to decrypt data. This is done using a function very similar to the signing function. It is somewhat simpler as there is no need to prepare the data. Here we go:

```
/* Decrypt the data in INDATA and return the allocated result in OUTDATA.
   If a PIN is required the PINCB will be used to ask for the PIN; it
   should return the PIN in an allocated buffer and put it into PIN.  */
static gpg_error_t
do_decipher (app_t app, const char *keyidstr,
             gpg_error_t (*pincb)(void*, const char *, char **),
             void *pincb_arg,
             const void *indata, size_t indatalen,
             unsigned char **outdata, size_t *outdatalen )
{
  static const unsigned char mse_parm[] = {
    0x80, 1, 0x10, /* Select algorithm RSA. */
    0x84, 1, 0x81  /* Select local secret key 1 for decryption. */
  };
  int rc, i;
  int fid;

  if (!keyidstr || !*keyidstr || !indatalen)
    return gpg_error (GPG_ERR_INV_VALUE);

  /* Check that the provided ID is vaid.  This is not really needed
     but we do it to to enforce correct usage by the caller. */
  if (strncmp (keyidstr, "NKS-DF01.", 9) )
    return gpg_error (GPG_ERR_INV_ID);
  keyidstr += 9;
  if (!hexdigitp (keyidstr) || !hexdigitp (keyidstr+1)
      || !hexdigitp (keyidstr+2) || !hexdigitp (keyidstr+3)
      || keyidstr[4])
    return gpg_error (GPG_ERR_INV_ID);
  fid = xtoi_4 (keyidstr);
  for (i=0; filelist[i].fid; i++)
    if (filelist[i].iskeypair && filelist[i].fid == fid)
      break;
  if (!filelist[i].fid)
    return gpg_error (GPG_ERR_NOT_FOUND);
  if (!filelist[i].isenckey)
    return gpg_error (GPG_ERR_INV_ID);

  /* Do the TCOS specific MSE. */
  rc = iso7816_manage_security_env (app->slot,
                                    0xC1, 0xB8,
                                    mse_parm, sizeof mse_parm);
  if (!rc)
```

```
      rc = verify_pin (app, pincb, pincb_arg);
    if (!rc)
      rc = iso7816_decipher (app->slot, indata, indatalen, 0x81,
                             outdata, outdatalen);
    return rc;
  }
```

Note that many cards need to setup some parameters before a crypto operation can be done. This is accomplished with the Manage Security Environment (MSE) ISO command. This is often very card specific.

Now that we have all the code assembled, we only need to link this module to the SCdaemon core. This is done by the select function which will be called from SCdaemon's upper layer:

```
  /* Select the NKS 2.0 application.  */
  gpg_error_t
  app_select_nks (app_t app)
  {
    static char const aid[] = { 0xD2, 0x76, 0x00, 0x00, 0x03, 0x01, 0x02 };
    int slot = app->slot;
    int rc;

    rc = iso7816_select_application (slot, aid, sizeof aid, 0);
    if (!rc)
      {
        app->apptype = "NKS";

        app->fnc.learn_status = do_learn_status;
        app->fnc.readcert = do_readcert;
        app->fnc.getattr = NULL;
        app->fnc.setattr = NULL;
        app->fnc.genkey = NULL;
        app->fnc.sign = do_sign;
        app->fnc.auth = NULL;
        app->fnc.decipher = do_decipher;
        app->fnc.change_pin = NULL;
        app->fnc.check_pin = NULL;
     }

    return rc;
  }
```

All the select functions may be called by SCdaemon in turn to select the application to be used. If this modules feels that it is responsible for the card and can support the application it should set the application type and return success. If it is not responsible it should return an error. To implement this function, it is often sufficient to just do a SELECT FILE command with the application identifier. This is what we do for NKS. Other cards might need some more checking or preparation, though.

6 SCdaemon and PKCS#11

In the proprietary world another interface is used to let an application from one vendor communicate with the smart card of another vendor. This interface is called PKCS#11 and consist of an API description to be implemented by both sides. This API as evolved a lot over time and only in recent times most proprietary extensions to this standard[5] have been replaced by specified ones. As with PC/SC the major reason for this interface is to allow proprietary applications talk to each other and claim interoperability where there is actually none. Between Free Software components such a messy and non-matured interface would have not been useful as it is easier to do it in the Right Way.

However, all Mozilla based software (e.g. Firefox and Thunderbird) makes use of PKCS#11 and thus there has always been public demand to support them by GnuPG. We eventually heard the voices and developed *Scute*[6], our PKCS#11 to SCdaemon connector. It is currently in the beginnings but already very useful to use the OpenPGP card with Mozilla to access client certificate authenticated websites. Encryption support for use by Thunderbird will follow soon.

Scute is an example on how the SCdaemon based system may be used to enable applications with smart card support. To an application SCdaemon appears as an independent hardware token with its own display and keyboard. That is in fact also the view of PKCS#11 and thus it was not very hard to implement it. Well, kind of. The major hassles are bugs in Mozilla's implementaion of PKCS#11 and the public header file with the API declaration not being compatible with GPL or LGPL software. We had to rewrite this header file to provide a really free implementation.

Of particular interest might be the way the support for the OpenPGP card is implemented: The OpenPGP card does not store a certificate or a keyblock (OpenPGP's variant of a certificate). This is due to size reasons and well, because it is basically targeted to OpenPGP. To overcome this, Scute combines access to SCdaemon with calls to `gpgsm` to send corresponding certificates to the caller. This allows to use the very same smart card for use with OpenPGP and X.509 (S/MIME).

[5] which used to made it pointless as a "standard"

[6] http://www.scute.org

7 The Road Ahead

As usual in the Free Software community, development never stops and new ideas are integrated as the system evolves. We have a couple of plans for the near future which includes enhancements to Scute and an easy way to generate certificates from existing OpenPGP keys.

The CCID driver will be enhanced to support more card readers with PIN pads (PIN pad access is not very well standardized). Although support for several readers is basically implemented, it needs some cleanups to make proper use of more than one reader. There is also the idea of implementing something like virtual readers, so that we are able to make use of RFID cards.

Porting the whole architecture to Windows is in the works; this will allow to use this system in heterogeneous environments.

Poldi, the existing PAM for using the OpenPGP card for login authentication will be migrated to use SCdaemon instead of accessing the smart card directly (based on source copied code from SCdaemon).

The most important task however is to support more smart cards; i.e. writing new application modules. The problem with this is not the actual work but the tendency of almost all smart card vendors not to disclose the specification or requiring to sign strict non-disclosure agreements. Only a very few vendors have meanwhile understood that support for Unix based systems is of their own interest and are helpful in providing documentation and samples.

So while we are waiting for new smart cards, you may want to look at the public API and check whether it is useful to enhance your application with smart card support. If there is something missing, feel free to discuss this on the GnuPG developer's mailing list.

Copyright 2007 Werner Koch. This work is licensed under the Creative Commons Attribution-Share Alike 2.5 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/2.5/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA. The quoted source code is part of GnuPG and used herein only for documentation. To use or distribute this source code as computer software, you must adhere to the terms of the GNU General Public License as included with the GnuPG software (see http://www.gnupg.org).