Ada Keystore Guide

STEPHANE CARREZ

2020-12-26

Contents

1 Introduction

Ada Keystore is a library and tool to store information in secure wallets and protect the stored information by encrypting the content. It is necessary to know one of the wallet password to access its content. Ada Keystore can be used to safely store passwords, credentials, bank accounts and even documents.

Wallets are protected by a master key using AES-256 and the wallet master key is protected by a user password. The wallet defines up to 7 slots that identify a password key that is able to unlock the master key. To open a wallet, it is necessary to unlock one of these 7 slots by providing the correct password. Wallet key slots are protected by the user's password and the PBKDF2-HMAC-256 algorithm, a random salt, a random counter and they are encrypted using AES-256.

Values stored in the wallet are protected by their own encryption keys using AES-256. A wallet can contain another wallet which is then protected by its own encryption keys and passwords (with 7 independent slots). Because the child wallet has its own master key, it is necessary to known the primary password and the child password to unlock the parent wallet first and then the child wallet.

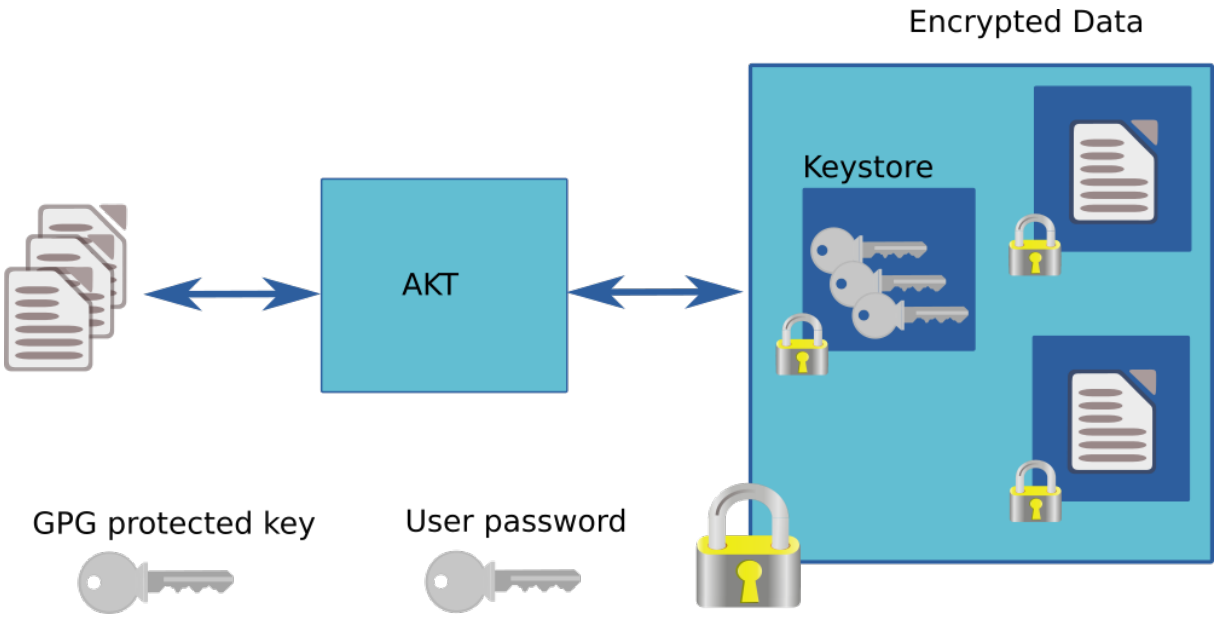

Figure 1: *AKT Overview*

The data is organized in 4K blocks whose primary content is encrypted either by the wallet master key or by the entry keys. The data block is signed by using HMAC-256. A data block can contain several values but each of them is protected by its own encryption key. Each value is also signed using HMAC-256.

The keystore uses several encryption keys at different levels to protect the content. A document stored in the keystore is split in data fragment and each data fragment is encrypted by using its own key. The data fragments are stored in specific data blocks so that they are physically separated from the

encryption keys.

The data fragment encryption keys are stored in the directory blocks and they are encrypted by using a specific key.

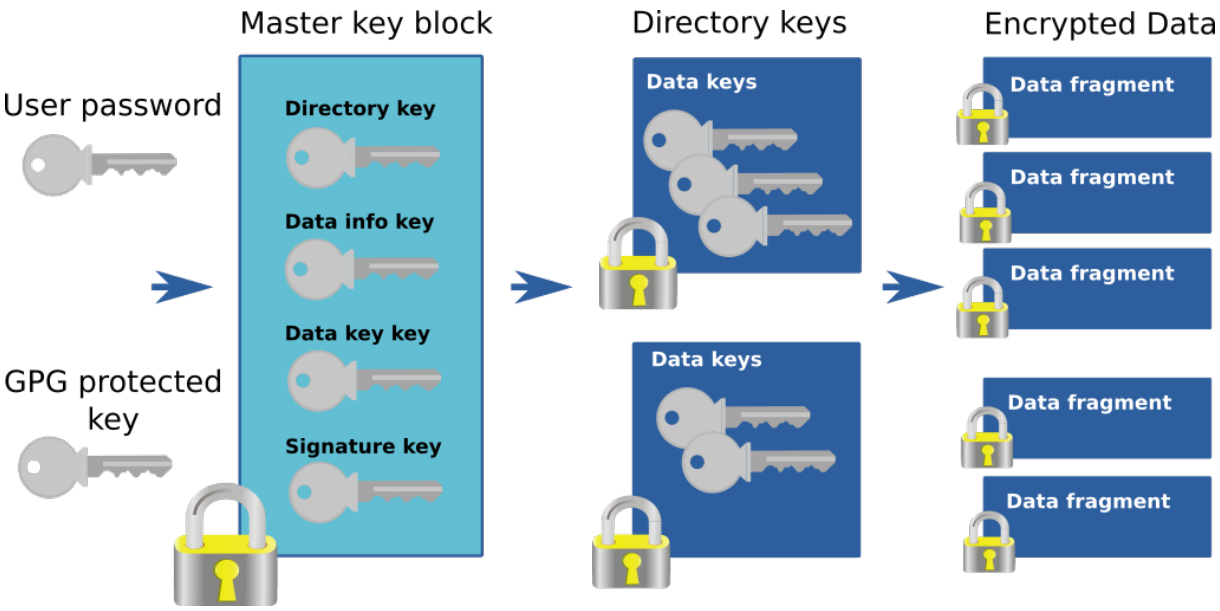

Figure 2: *AKT Keys*

This document describes how to build the tool and library and how you can use the different features to protect your sensitive data.

2 Installation

This chapter explains how to build and install the library.

2.1 Before Building

To build the Ada Keystore you will need the GNAT Ada compiler, either the FSF version available in Debian, FreeBSD systems NetBSD or the AdaCore GNAT Community 2019 edition.

2.1.1 Ubuntu

Install the following packages:

```
sudo apt-get install -y make gnat-7 gprbuild git gnupg2
```

2.1.2 FreeBSD 12

Install the following packages:

```
pkg install gmake gcc6-aux-20180516_1,1 gprbuild-20160609_1 git gnupg
    -2.2.17_2
```

2.1.3 Windows

Get the Ada compiler from AdaCore Download site and install.

Install the following packages:

```
pacman -S git
pacman -S make
pacman -S base-devel --needed
```

2.2 Getting the sources

The project uses a sub-module to help you in the integration and build process. You should checkout the project with the following commands:

```
    git clone --recursive https://github.com/stcarrez/ada-keystore.git
    cd ada-keystore
```

2.3 Configuration

The library uses the `configure` script to detect the build environment, check which Ada Utility Library to use. If some component is missing, the `configure` script will report an error or it will disable the feature. The `configure` script provides several standard options and you may use:

- `--prefix=DIR` to control the installation directory,
- `--enable-fuse` to enable building the `mount` command with FUSE,
- `--enable-gtk` to enable building the Gtk tool,
- `--enable-shared` to enable the build of shared libraries,
- `--disable-static` to disable the build of static libraries,
- `--disable-nls` to disable NLS support,
- `--with-ada-util=PATH` to control the installation path of Ada Utility Library,
- `--with-gtkada=PATH` to control the installation path of Gtk Ada Library,
- `--help` to get a detailed list of supported options.

In most cases you will configure with the following command:

```
1 ./configure
```

The GTK application is not compiled by default unless you configure with the `--enable-gtk` option. Be sure to install the GtkAda library before configuring and building the project.

```
1 ./configure  --enable-gtk
```

On Windows, FreeBSD and NetBSD you have to disable the NLS support:

```
1 ./configure --disable-nls
```

2.4 Build

After configuration is successful, you can build the library by running:

```
1 make
```

After building, it is good practice to run the unit tests before installing the library. The unit tests are built and executed using:

```
1 make test
```

And unit tests are executed by running the `bin/keystore_harness` test program.

2.5 Installation

The installation is done by running the `install` target:

```
make install
```

If you want to install on a specific place, you can change the `prefix` and indicate the installation direction as follows:

```
make install prefix=/opt
```

2.6 Using

To use the library in an Ada project, add the following line at the beginning of your GNAT project file:

```
with "keystoreada";
```

3 Using Ada Keystore Tool

The `akt` tool is the command line tool that manages the wallet. It provides the following commands:

- `create`: create the keystore
- `edit`: edit the value with an external editor
- `get`: get a value from the keystore
- `help`: print some help
- `list`: list values of the keystore
- `remove`: remove values from the keystore
- `set`: insert or update a value in the keystore

To create the secure file, use the following command and enter your secure password (it is recommended to use a long and complex password):

```
1    akt create secure.akt
```

At this step, the secure file is created and it can only be opened by providing the password you entered. To add something, use:

```
1    akt set secure.akt bank.password 012345
```

To store a file, use the following command:

```
1    akt store secure.akt contract.doc
```

If you want to retrieve a value, you can use one of:

```
1    akt get secure.akt bank.password
2    akt extract secure.akt contract.doc
```

You can also use the `akt` command together with the `tar` command to create secure backups. You can create the compressed tar file, pipe the result to the `akt` command to store the content in the wallet.

```
1    tar czf - dir-to-backup | akt store secure.akt -- backup.tar.gz
```

To extract the backup you can use the `extract` command and feed the result to the `tar` command as follows:

```
1    akt extract secure.akt -- backup.tar.gz | tar xzf -
```

4 Programmer's Guide

4.1 Keystore

The `Keystore` package provides operations to store information in secure wallets and protect the stored information by encrypting the content. It is necessary to know one of the wallet password to access its content. Wallets are protected by a master key using AES-256 and the wallet master key is protected by a user password. The wallet defines up to 7 slots that identify a password key that is able to unlock the master key. To open a wallet, it is necessary to unlock one of the 7 slots by providing the correct password. Wallet key slots are protected by the user's password and the PBKDF2-HMAC-256 algorithm, a random salt, a random counter and they are encrypted using AES-256.

4.1.1 Creation

To create a keystore you will first declare a `Wallet_File` instance. You will also need a password that will be used to protect the wallet master key.

```
1 with Keystore.Files;
2 ...
3   WS   : Keystore.Files.Wallet_File;
4   Pass : Keystore.Secret_Key := Keystore.Create ("There was no choice
       but to be pioneers");
```

You can then create the keystore file by using the `Create` operation:

```
1   WS.Create ("secure.akt", Pass);
```

4.1.2 Storing

Values stored in the wallet are protected by their own encryption keys using AES-256. The encryption key is generated when the value is added to the wallet by using the `Add` operation.

```
1   WS.Add ("Grace Hopper", "If it's a good idea, go ahead and do it.");
```

The `Get` function allows to retrieve the value. The value is decrypted only when the `Get` operation is called.

```
1   Citation : constant String := WS.Get ("Grace Hopper");
```

The `Delete` procedure can be used to remove the value. When the value is removed, the encryption key and the data are erased.

```
1    WS.Delete ("Grace Hopper");
```

5 AKT Tool

5.1 NAME

akt - Tool to protect your sensitive data with secure storage

5.2 SYNOPSIS

akt [-v] [-vv] [-vv] [-V] [-c *config-file*] [-t *count*] [-z] *command* [-k *file*] [-d *dir*] [-p *password*] [–password *password*] [–passfile *file*] [–passenv *name*] [–passfd *fd*] [–passask] [–passcmd *cmd*] [–wallet-key-file *file*]

5.3 DESCRIPTION

akt is a tool to store information in secure wallets and protect the stored information by encrypting the content. It is necessary to know one of the wallet password to access its content. *akt* can be used to safely store passwords, credentials, bank accounts and even documents.

Wallets are protected by a master key using AES-256 and the wallet master key is protected by a user password. The wallet defines up to 7 slots that identify a password key that is able to unlock the master key. To open a wallet, it is necessary to unlock one of these 7 slots by providing the correct password. Wallet key slots are protected by the user's password and the PBKDF2-HMAC-256 algorithm, a random salt, a random counter and they are encrypted using AES-256.

Values stored in the wallet are protected by their own encryption keys using AES-256. A wallet can contain another wallet which is then protected by its own encryption keys and passwords (with 7 independent slots). Because the child wallet has its own master key, it is necessary to known the primary password and the child password to unlock the parent wallet first and then the child wallet.

The data is organized in blocks of 4K whose primary content is encrypted either by the wallet master key or by the entry keys. The data block is signed by using HMAC-256. A data block can contain several values but each of them is protected by its own encryption key. Each value is also signed using HMAC-256. Large values can be written to several data blocks and in that case each fragment is encrypted by using its own encryption key.

The tool provides several commands that allow to create a keystore, insert values, retrieve values or delete them. You can use it to store your passwords, your secret keys and even your documents.

Passwords are retrieved using one of the following options:

- by reading a file that contains the password,
- by looking at an environment variable,
- by using a command line argument,

- by getting the password through the *ssh-askpass*(1) external command,
- by running an external command,
- by asking interactively the user for the password,
- by asking through a network socket for the password.

5.4 OPTIONS

The following options are recognized by *akt*:

-V Prints the *akt* version.

-v Enable the verbose mode.

-vv Enable debugging output.

-c *config-file* Defines the path of the global *akt* configuration file.

-t *count* Defines the number of threads for the encryption and decryption process. By default, it uses the number of system CPU cores.

-k file

Specifies the path of the keystore file to open.

-d directory

Specifies the directory path of the keystore data files. When this option is used, the data blocks are written in separate files. The data blocks do not contain the encryption keys and each of them is encrypted with its own secure key.

-p password

The keystore password is passed within the command line. Using this method is convenient but is not safe.

–passenv envname

The keystore password is passed within an environment variable with the given name. Using this method is considered safer but still provides some security leaks.

–passfile path

The keystore password is passed within a file that is read. The file must not be readable or writable by other users or group: its mode must be r??——. The directory that contains the file must also satisfy the not readable by other users or group members, This method is safer.

–passfd fd

The keystore password is passed within a pipe whose file descriptor number is given. The file descriptor is read to obtain the password. This method is safer.

–passask

The keystore password is retrieved by the running the external tool *ssh-askpass*(1) which will ask the password through either KDE, Gnome or another desktop interactive application. The password is retrieved through a pipe that *akt* sets while launching the command.

–passcmd cmd

The keystore password is retrieved by the running the external command defined in *cmd*. The command should print the password on its standard output without end of line. The password is retrieved through a pipe that *akt* sets while launching the command.

–wallet-key-file file Defines the path of a file which contains the wallet master key file.

-z Erase and fill with zeros instead of random values.

5.5 COMMANDS

5.5.1 The create command

```
akt create keystore.akt [--force] [--counter-range min:max] [--split
   count] [--gpg user ...]
```

Create a new keystore and protect it with the password. When the keystore file already exist, the create operation will fail unless the *–force* option is passed.

The password to protect the wallet is passed using one of the following options: *–passfile* , *–passenv* , *–password* , *–passsocket* or *–gpg*. When none of these options are passed, the password is asked interactively.

The *–counter-range* option allows to control the range for the random counter used by PBKDF2 to generate the encryption key derived from the specified password. High values provide a strongest derived key at the expense of speed. This option is ignored when the *–gpg* option is used.

The *–split* option indicates to use several separate files for the data blocks and it controls the number of separate files to use. When used, a directory with the name of the keystore file is created and will contain the data files.

The *–gpg* option allows to protect the keystore by using a user's GPG encryption key. The option argument defines the GPG user's name or GPG key. When the keystore password is protected by the user's GPG key, a random password is generated to protect the keystore. The *gpg2*(1) command is used to encrypt that password using the user's public key and save it in the keystore header. The *gpg2*(1)

command is then used to decrypt that and be able to unlock the keystore provided that the user's private key is known. When using the *–gpg* option, it is possible to protect the keystore for several users, thus being able to share the secure file with each of them.

5.5.2 The extract command

```
akt extract keystore.akt -- name
```

```
akt extract keystore.akt {name...}
```

This command allows to extract files or directories recursively from the keystore. It is possible to extract several files and directories at the same time.

When the – option is passed, the command accepts only one argument. It extracts the specified name and writes the result on the standard output. It can be used as a target for a pipe command.

5.5.3 The mount command

```
akt mount keystore.akt [-f] [--enable-cache] mount-point
```

This command is available when the *fuse*(8) support is enabled. It allows to mount the keystore content on the *mount-point* directory and access the encrypted content through the filesystem. The *akt* tool works as a daemon to serve *fuse*(8) requests that come from the kernel. The *-f* option allows to run this daemon as a foreground process. By default, the kernel cache are disabled because the keystore content is decrypted and given as clear content to the kernel. This could be a security issue for some system and users. The kernel cache can be enabled by using the *–enable-cache* option.

To unmount the file system, one must use the *mount*(8) command.

```
umount mount-point
```

5.5.4 The set command

```
akt set keystore.akt name value
```

The *set* command is used to store a content passed as command line argument in the wallet. If the wallet already contains the name, the value is updated.

5.5.5 The store command

```
akt store keystore.akt -- name
```

```
akt store keystore.akt {file...|directory...}
```

This command can store files or directories recursively in the keystore. It is possible to store several files and directories at the same time.

When the – option is passed, the command accepts only one argument. It reads the standard input and stores it under the specified name. It can be used as a target for a pipe command.

5.5.6 The remove command

```
akt remove keystore.akt name ...
```

The *remove* command is used to erase a content from the wallet. The data block that contained the content to protect is erased and replaced by zeros. The secure key that protected the wallet entry is also cleared. It is possible to remove several contents.

5.5.7 The edit command

```
akt edit keystore.akt [-e editor] name
```

The *edit* command can be used to edit the protected wallet entry by calling the user's prefered editor with the content. The content is saved in a temporary directory and in a temporary file. The editor is launched with the path and when editing is finished the temporary file is read. The temporary directory and files are erased when the editor terminates successfully or not. The editor can be specified by using the *-e* option, by setting up the *EDITOR* environment variable or by updating the *editor*(1) alternative with *update-alternative*(1).

5.5.8 The list command

```
akt list keystore.akt
```

The *list* command describes the entries stored in the keystore with their name, size, type, creation date and number of keys which protect the entry.

5.5.9 The get command

```
akt get keystore.akt [-n] name...
```

The *get* command allows to retrieve the value associated with a wallet entry. It retrieves the value for each name passed to the command. The value is printed on the standard output. By default a newline is emitted after each value. The *-n* option prevents the output of the trailing newline.

5.5.10 The password-add command

```
akt password-add keystore.akt [--new-passfile file] [--new-password
   password] [--new-passenv name]
```

The *password-add* command allows to add a new password in one of the wallet key slot. Up to seven passwords can be defined to protect the wallet. The overall security of the wallet is that of the weakest password. To add a new password, one must know an existing password.

5.5.11 The password-remove command

```
akt password-remove keystore.akt [--force]
```

The *password-remove* command can be used to erase a password from the wallet master key slots. Removing the last password makes the keystore unusable and it is necessary to pass the *–force* option for that.

5.5.12 The password-set command

```
akt password-set [--new-passfile file] [--new-password password] [--new
   -passenv name]
```

The *password-set* command allows to change the current wallet password.

5.6 SECURITY

Wallet master keys are protected by a derived key that is created from the user's password using *PBKDF2* and *HMAC-256* as hashing operation. When the wallet is first created, a random salt and counter are allocated which are then used by the *PBKDF2* generation. The wallet can be protected by up to 7 different passwords. Despite this, the security of the wallet master key still depends on the strength of the user's password. For this matter, it is still critical for the security to use long passphrases.

The passphrase can be passed within an environment variable or within a command line argument. These two methods are considered unsafe because it could be possible for other processes to see these values. It is best to use another method such as using the interactive form, passing the password through a file or passing using a socket based communication.

When the wallet master key is protected using *gpg2*(1) a 32-bytes random binary key and a 16-bytes random binary IV is created to protect the wallet master key. Another set of 80 bytes of random binary data is used to encrypt and sign the whole wallet master key block. The 128 bytes that form these random binary keys are encrypted using the user's GPG public key and the result saved in the keystore header block. The *–gpg* option is specified only for the creation of the keystore and allows to encrypt a master key slot for several GPG keys. To unlock the keystore file, the *gpg2*(1) command will be used to decrypt the keystore header content automatically. When the user's GPG private key is not found, it is not possible to unlock the keystore with this method.

When several GPG keys are used to protect the wallet, they share the same 80 bytes to decrypt the wallet master key block but they have their own key and IV to unlock the key slot.

Depending on the size, a data stored in the wallet is split in one or several data entry. Each wallet data entry is then protected by their own secret key and IV vector. Wallet data entry are encrypted using AES-256-CBC. The wallet data entry key and IV vectors are protected by the wallet master key.

When the *–split* option is used, the data storage files only contain the data blocks. They do not contain any encryption key. The data storage files use the *.dkt* file extension.

5.7 CONFIGURATION

The *akt* global configuration file contains several configuration properties which are used to customize several commands. These properties can be modified with the *config* command.

5.7.1 gpg-encrypt

This property defines the *gpg2*(1) command to be used to encrypt a content. The content to encrypt is passed in the standard input and the encrypted content is read from the standard output. The GPG key parameter can be retrieved by using the *$USER* pattern.

5.7.2 gpg-decrypt

This property defines the *gpg2*(1) command to be used to decrypt a content. The content to decrypt is passed in the standard input and the decrypted content is read from the standard output.

5.7.3 gpg-list-keys

This property defines the *gpg2*(1) command to be used to retrieve the list of available secret keys. This command is executed when the keystore file is protected by a GPG key to identify the possible GPG Key ids that are capable of decrypting it.

5.7.4 fill-zero

This property controls whether *akt* must fill unused data areas with zeros or with random bytes.

5.8 SEE ALSO

editor(1), *update-alternative(1)*, *ssh-askpass(1)*, *gpg2(1)*, *mount(8)*, *fuse(8)*

5.9 AUTHOR

Written by Stephane Carrez.

6 Implementation

This chapter explains how the wallets are organised and protected.

6.1 File layouts

The data is organized in 4K blocks. The first block is a header block used to store various information to identify the storage files. Other blocks have a clear 16-byte header and an HMAC-256 signature at the end. Blocks are encrypted either by using the master key, the directory key, the data key or a per-data fragment key.

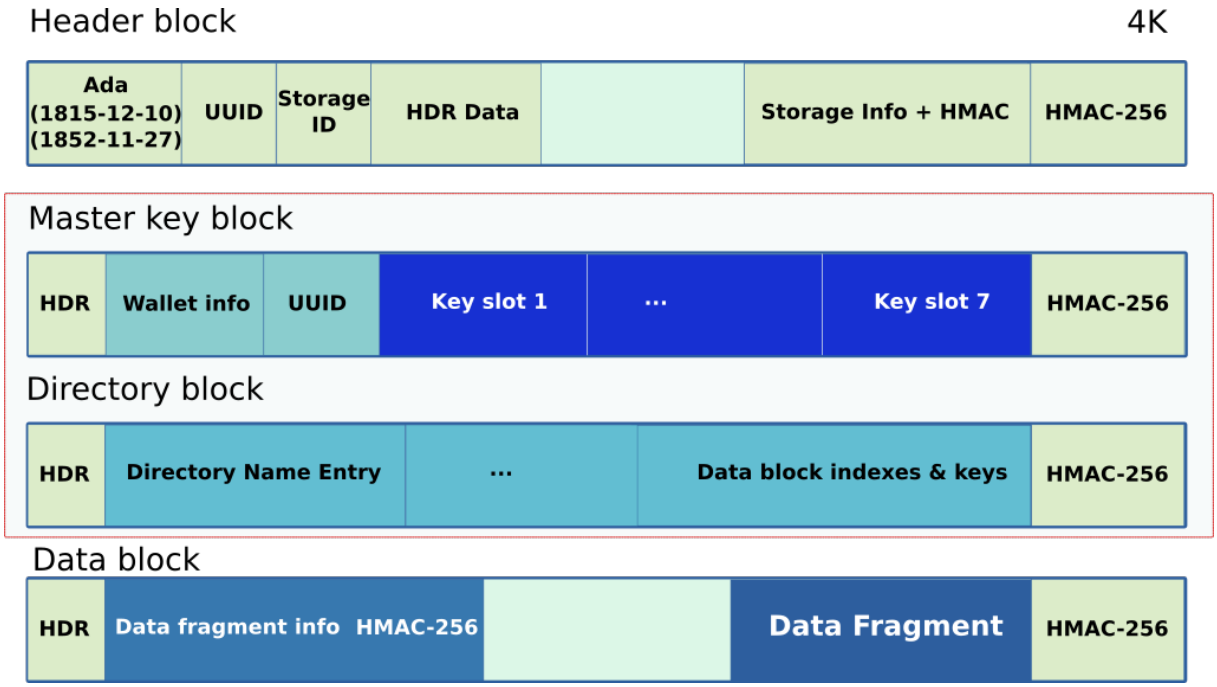

Figure 3: *Keystore blocks overview*

The master key block and directory block are the two blocks that contain encryption keys.

6.1.1 Header block

The first block of the file is the keystore header block which contains clear information signed by an HMAC header. The header block contains the keystore UUID as well as a short description of each storage data file. It also contains some optional header data.

```
+------------------+
| 41 64 61 00      | 4b = Ada
```

```
| 00 9A 72 57       | 4b = 10/12/1815
| 01 9D B1 AC       | 4b = 27/11/1852
| 00 01             | 2b = Version 1
| 00 01             | 2b = File header length in blocks
+-------------------+
| Keystore UUID     | 16b
| Storage ID        | 4b
| Block size        | 4b
| Storage count     | 4b
| Header Data count | 2b
+-------------------+-----
| Header Data size  | 2b
| Header Data type  | 2b = 0 (NONE), 1 (GPG1) 2, (GPG2)
+-------------------+
| Header Data       | Nb
+-------------------+-----
| ...               |
+-------------------+-----
| 0                 |
+-------------------+-----
| ...               |
+-------------------+-----
| Storage ID        | 4b
| Storage type      | 2b
| Storage status    | 2b  00 = open, Ada = sealed
| Storage max bloc  | 4b
| Storage HMAC      | 32b = 44b
+-------------------+----
| Header HMAC-256   | 32b
+-------------------+----
```

6.1.2 GPG Header data

The GPG encrypted data contains the following information:

```
+-------------------+-----
| TAG               | 4b
+-------------------+-----
| Lock Key          | 32b
| Lock IV           | 16b
| Wallet Key        | 32b
| Wallet IV         | 16b
```

```
| Wallet Sign       | 32b
+-------------------+-----
```

6.1.3 Master keys

Wallet header encrypted with the parent wallet id

```
+-------------------+
| 01 01             | 2b
| Encrypt size      | 2b
| Parent Wallet id  | 4b
| PAD 0             | 4b
| PAD 0             | 4b
+-------------------+
| Wallet magic      | 4b
| Wallet version    | 4b
| Wallet lid        | 4b
| Wallet block ID   | 4b
+-------------------+
| Wallet gid        | 16b
+-------------------+
| Wallet key count  | 4b
| PAD 0             | 4b
+-------------------+
| Key type          | 4b
| Key size          | 4b
| Counter for key   | 4b
| Counter for iv    | 4b
| Salt for key      | 32b
| Salt for iv       | 32b
| Key slot sign     | 32b
| Dir key # 1       | 32b ---
| Dir iv # 1        | 16b  ^
| Dir sign # 1      | 32b  |
| Data key # 1      | 32b  |
| Data iv # 1       | 16b  | Encrypted by user's password
| Data sign #1      | 32b  |
| Key key # 1       | 32b  |
| Key iv # 1        | 16b  v
| Key sign #1       | 32b ---
| Slot HMAC-256     | 32b
| PAD 0 / Random    | 80b
```

```
+------------------+
| Key slot #2      | 512b
+------------------+
| Key slot #3      | 512b
+------------------+
| Key slot #4      | 512b
+------------------+
| Key slot #5      | 512b
+------------------+
| Key slot #6      | 512b
+------------------+
| Key slot #7      | 512b
+------------------+
| PAD 0 / Random   |
+------------------+
| Block HMAC-256   | 32b
+------------------+
```

6.1.4 Directory Entries

The wallet repository block is encrypted with the wallet directory key.

```
+------------------+
| 02 02            | 2b
| Encrypt size     | 2b = BT_DATA_LENGTH
| Wallet id        | 4b
| PAD 0            | 4b
| PAD 0            | 4b
+------------------+
| Next block ID    | 4b  Block number for next repository block with
   same storage
| Data key offset  | 2b  Starts at IO.Block_Index'Last, decreasing
+------------------+
| Entry ID         | 4b   ^
| Entry type       | 2b   | = T_STRING, T_BINARY
| Name size        | 2b   |
| Name             | Nb   | DATA_NAME_ENTRY_SIZE + Name'Length
| Create date      | 8b   |
| Update date      | 8b   |
| Entry size       | 8b   v
+------------------+
| Entry ID         | 4b   ^
```

```
| Entry type        | 2b    | = T_WALLET
| Name size         | 2b    |
| Name              | Nb    | DATA_NAME_ENTRY_SIZE + Name'Length
| Create date       | 8b    |
| Update date       | 8b    |
| Wallet lid        | 4b    |
| Wallet master ID  | 4b    v
+-------------------+
| ...               |
+-------------------+--
| 0 0 0 0           | 4b (End of name entry list = DATA_KEY_SEPARATOR)
+-------------------+--
| ...               |     (random or zero)
+-------------------+--  <- = Data key offset
| ...               |
+-------------------+
| Storage ID        | 4b    ^ Repeats "Data key count" times
| Data block ID     | 4b    |
| Data size         | 2b    | DATA_KEY_ENTRY_SIZE = 58b
| Content IV        | 16b   |
| Content key       | 32b   v
+-------------------+
| Entry ID          | 4b    ^
| Data key count    | 2b    | DATA_KEY_HEADER_SIZE = 10b
| Data offset       | 4b    v
+-------------------+
| Block HMAC-256    | 32b
+-------------------+--
```

6.1.5 Data Block

Data block start is encrypted with wallet data key, data fragments are encrypted with their own key. Loading and saving data blocks occurs exclusively from the workers package. The data block can be stored in a separate file so that the wallet repository and its keys are separate from the data blocks.

```
+-------------------+
| 03 03             | 2b
| Encrypt size      | 2b = DATA_ENTRY_SIZE * Nb data fragment
| Wallet id         | 4b
| PAD 0             | 4b
| PAD 0             | 4b
+-------------------+-----
```

```
| Entry ID          | 4b  Encrypted with wallet id
| Slot size         | 2b
| 0 0               | 2b
| Data offset       | 8b
| Content HMAC-256  | 32b => 48b = DATA_ENTRY_SIZE
+-------------------+
| ...               |
+-------------------+-----
| ...               |
+-------------------+
| Data content      |     Encrypted with data entry key
+-------------------+-----
| Block HMAC-256    | 32b
+-------------------+
```

6.2 Keystore Protections

The master key block contains the primary keys that are used to encrypt other blocks. The master key block contains 7 key slots that are capable to unlock the master keys. Each slot is independent and can be associated with a specific authentication method. Two authentication methods are supported:

- password based authentication,
- GPG based authentication.

6.2.1 Password Protection

In this mode, three secret information must be provided:

- the wallet header key and IV,
- the wallet signature key,
- the user password.

First, the wallet master key block is decrypted with AES-256-CBC by using the wallet header key and IV. The HMAC-256 signature is then computed with the wallet signature key on the decrypted content and the clear 16-byte header at beginning of the block. The HMAC signature must match the signature found at end of the block.

Once the wallet master key block is decrypted, the user password is checked against the available key slots. For a given password protected key slot, a derived key is generated by using the PBKDF2-HMAC256 algorithm. First, a 16-byte IV is generated and then a 32-byte key is generated. For each PBKDF2 execution a specific 32-byte salt and counter is used. The key slot is then decrypted by using the derived keys with AES-256-CBC. An HMAC-256 signature is built to verify the decrypted content.

When the HMAC signature matches the signature found in the key slot, the provided user's password is valid.

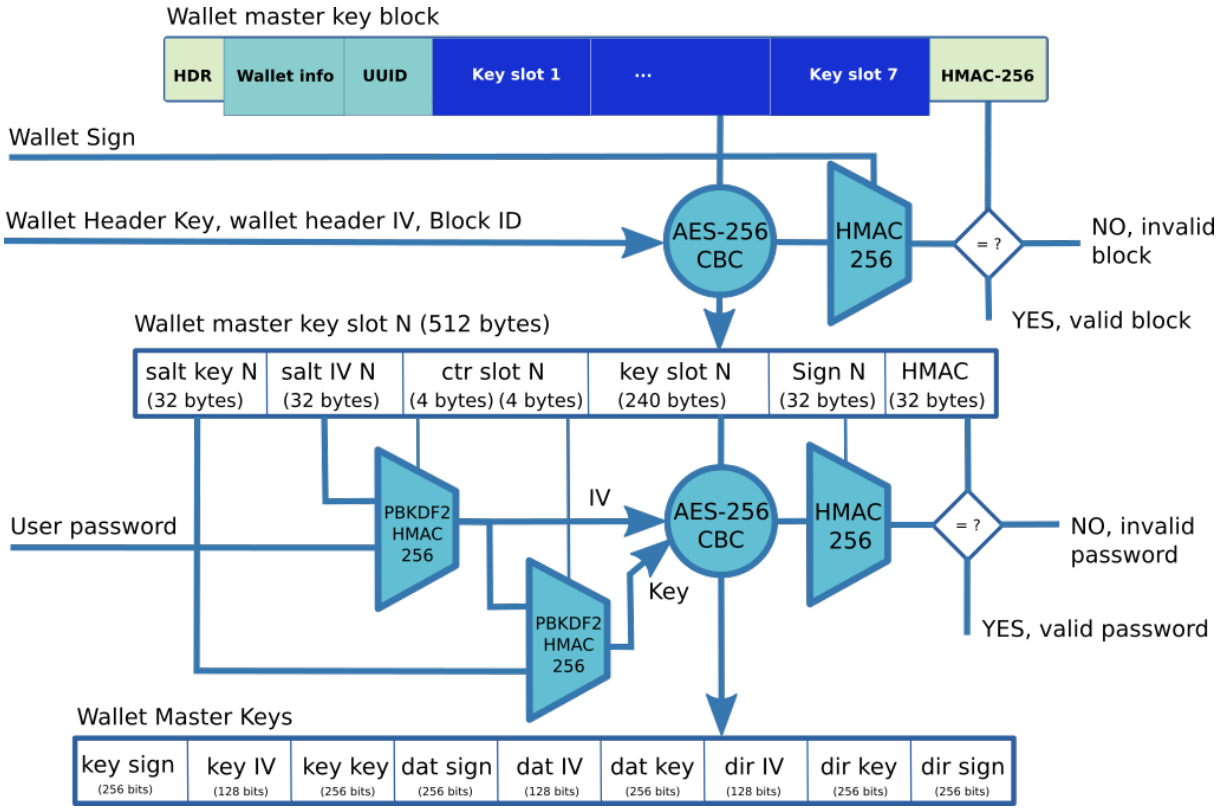

Figure 4: *Password based protection*

6.2.2 GPG Protection

With the GPG protection, the header block contains additional information that is decrypted with the user's GPG private key. When such additional data is successfully decrypted, it contains several parts:

- the wallet header key and IV,
- the wallet signature key,
- the key slot encryption key and IV.

The wallet master key block is decrypted and validated using the same process as the password protection.

The key slot that matches the GPG key is identified by a header tag that is found in the key slot and in the GPG header data. The key slot is decrypted by using the key slot encryption key and IV that was decrypted by GPG. It is validated using HMAC-256.

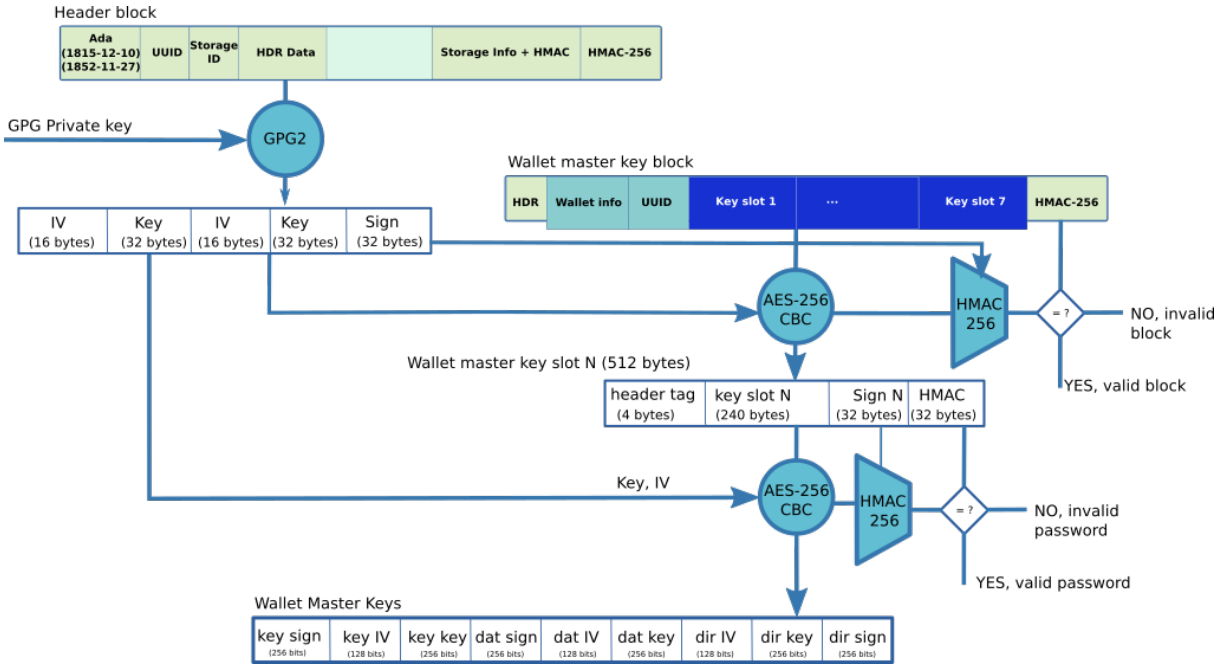

Figure 5: *GPG based protection*

6.2.3 Directory Protection

A directory block contains the name of contents found in the keystore as well as the keys used to encrypt data fragments. The directory block is decrypted with AES-256-CBC by using the directory key and IV. The directory block number is xored on the directory IV to obtain the IV used for the decryption. An HMAC-256 signature is computed with the clear 16-byte header and the decrypted directory content. It is then verified against the block HMAC.

Once decrypted, the directory block contains two areas. At beginning of the block, it contains the entry names that are stored in the keystore. For each entry, a unique entry ID is assigned and is used as a unique reference.

At end of the block, it contains the encryption keys and the block numbers where the data fragments are stored. Each data fragment has its own encryption key and IV.

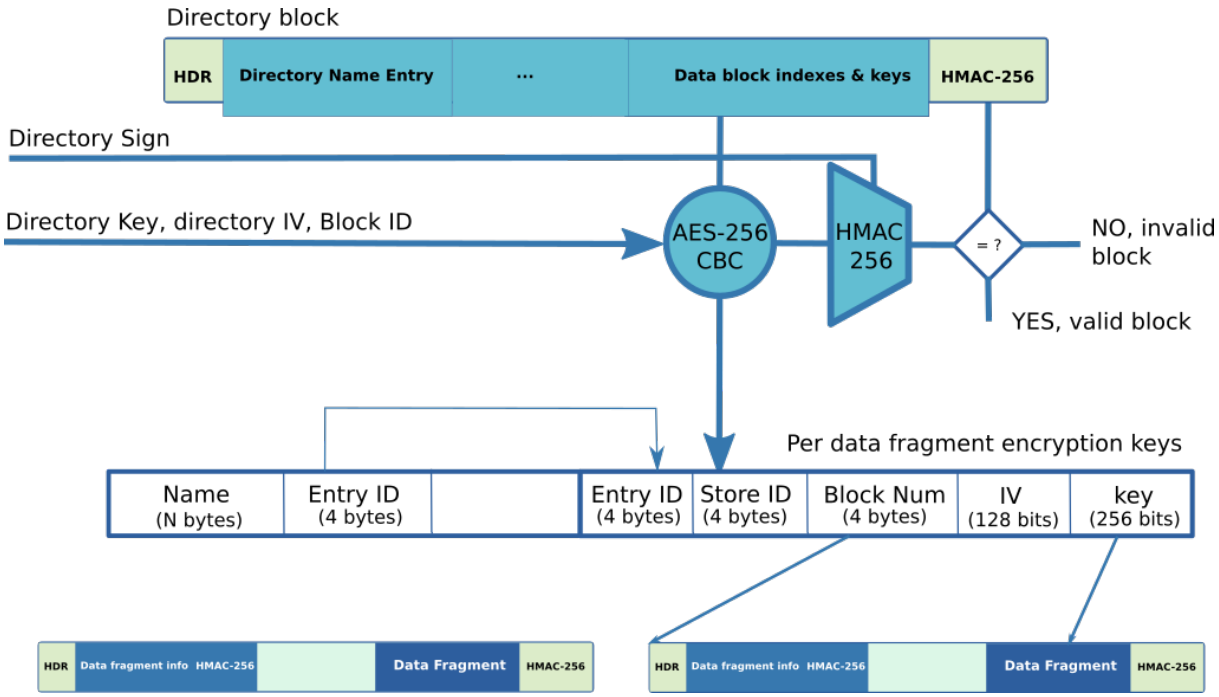

Figure 6: *Directory protection*