Deliverable D3.4
5G-PPP Security Enablers Documentation (v1.0)
Enabler Privacy Enhanced Identity Protection

Project name	5G Enablers for Network and System Security and Resilience	
Short name	5G-ENSURE	
Grant agreement	671562	
Call	H2020-ICT-2014-2	
Delivery date	30.09.2016	
Dissemination Level:	Public	
Lead beneficiary	NEC	Felix Klaedtke, felix.klaedtke@neclab.eu
Authors	TIIT: Madalina Baltatu, Luciana Costa, Dario Lombardo	

Contents

1 Introduction

The Privacy Enhanced Identity Protection enabler provides protection against subscriber's identity disclosure to unauthorized parties in 5G networks. The comprehensive goal of this enabler is to offer stronger user identity protection than in current 3G and 4G networks. The enabler can be summarized in several simple concepts: 5G user long term identity, also known as International Mobile Subscriber Identity (IMSI), shall not be transferred in clear text over the network in any situation. If a user long term identity (i.e., the IMSI) has to be sent from the UE to the network, it should be sent encrypted. For an encryption to be possible in situations where shared key material is not yet available, a public key-based mechanism is proposed (namely KPABE, [1], [2]).

The final version of the present enabler due in Release 2 will implement the following (full) set of features/components as enumerated in deliverable D3.1 [1]: Encryption of Long Term Identifiers and Pseudorandom dynamic pseudonyms. The present manual covers the version of the enabler provided in Release 1, as explained in D3.2 [2], and the specific feature planned for this version, namely: Encryption of long term or permanent identifiers.

2 Installation and Administration Guide

The present enabler consists in a cryptographic C library that implements the KP-ABE encryption scheme described in [3] allowing to encrypt and decrypt sensitive information like a given long term identifier (IMSI). For the rationales behind the use of KPABE the reader is asked to refer to previous study presented in D3.1 [1] and D3.2 [2]. The library provides the main encryption and decryption functions. It provides as well the setup and key generation functions, required in order to initialize the system and produce the key material.

In order to facilitate users' understanding of the functionalities and usage of the enabler, an example use case of the enabler is also described here and illustrated in Figure 1.

Figure 1 provides a simple proof of concept (PoC) deployment architecture, illustrating where the library should be installed. It also shows where each specific enabler's function is to be called in a 5G non 3GPP access scenario with EAP-AKA full authentication (based on IMSI).

This use case architecture is based on common hardware and readily available open source software, as indicated in the following sections.

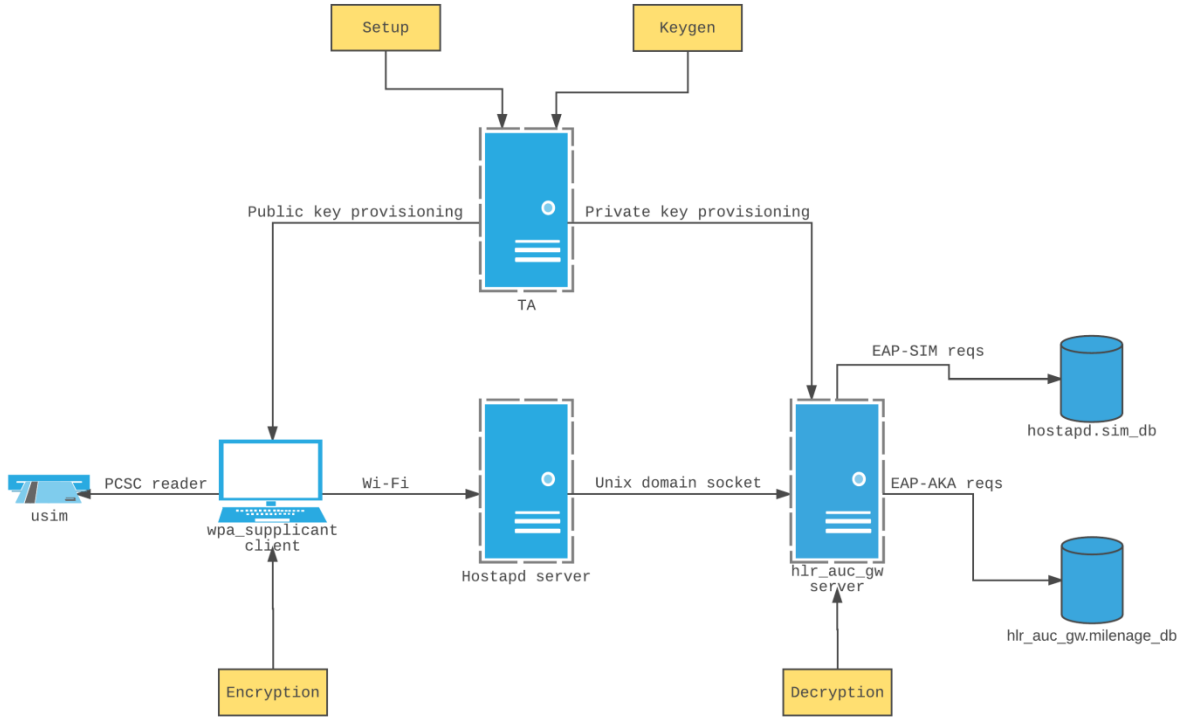

Figure 1 Example use case.

The algorithms implemented by the library are briefly described below with reference to the components illustrated in Figure 1.

- **Setup**: runs on a trusted authority server TA which is also responsible for the generation of private keys to trusted entities (e.g., network elements like authentication servers) based on the user access policy. At the cryptographic system initialization time, TA generates a public parameter (public key) and a master key. The latter is only known by the TA server.
- **Keygen:** runs on the TA server. An entity entitled to perform decryption (e.g., a network element like an authentication server) requests the provisioning of the decryption key (the private key) by presenting an access policy over an attribute or a set of attributes. The randomized key generation algorithm takes as input the public parameters (public key), the master key and the access policy. It outputs the authentication server's private key which is able to decrypt all IMSIs encrypted under the attributes that satisfy the access policy.
- **Encryption**: runs on the client components, e.g., on the UE devices (wpa_supplicant in Figure 1). The randomized encryption algorithm takes as input the user identity (i.e., the IMSI) to be encrypted, an attribute or set of attributes, and the public key. It outputs the ciphertext, i.e., the encrypted IMSI. In this way only the authentication servers which have the decryption key generated with the correct access policy (that matches the encryption attributes) will be able to decrypt the ciphertext.
- **Decryption:** the decryption algorithm runs on an authorized network element (i.e., the authentication server, i.e., the hlr_auc_gw in Figure 1). It takes as input the ciphertext, which was encrypted under the set of attributes, the public key parameter and the private key for access control. The output is the IMSI in clear text if the attributes included in the ciphertext satisfy the access policy.

The enabler delivers a C library (`libkpabe`) which implements these four functions required to perform all KPABE cryptographic operations. The library is developed by making use of some open source mathematics and cryptographic libraries that implement basic function blocks required by ABE and KPABE systems.

2.1 System Requirements

To work with the enabler, few requirements are necessary. They are listed below:

- A software for wireless communications, written in C language, that act as UE. An example of this software is wpa_supplicant.
- A software for wireless communications, that runs on the network side, written in C language. An example of this software is hlr_auc_gw (part of the hostapd project).
- 2 hardware equipment able to run the above softwares. An example of them are general purpose PCs with wifi cards.
- An operating system able to run on the above hardware and able to compile the enabler and the required libraries (see below). An example of it is Linux Ubuntu 14.04.

Additional software requirements on all systems are the prior installation of the following open source software libraries and packages:

- the PBC (Pairing Based Cryptography) C library [4],
- GMP, the GNU Multiple Precision Arithmetic C library [5],
- libcelia - a static linux C library implementing primitive kpabe operations [6],
- glib, the low-level core library used for GTK+ and GNOME and can be downloaded from [7].

2.2 Enabler Installation

Install GMP, PBC, glib and libcelia from the links provided in [5], [4], [7], and [7] by following the installation guides supplied with the corresponding packages. Note that glib and gmp can be installed on Ubuntu system from their packages by typing:

```
sudo apt-get install libglib2.0-dev
sudo apt-get install libgmp3-dev
```

Afterwards, libpkabe can be installed from its source archive. After tar gunzipping the archive, go to the libpkabe top directory and issue the following commands:

```
autoreconf -i
./configure
make
sudo make install
```

The working status of the local libkpabe installation can be verified by checking the output produced by the following command executed in the top directory of the library:

```
make check
```

The complete libkpabe documentation can be generated using:

```
make docs
```

from the top directory of the project. This command creates documentation subdirectories that contains the html and rtf versions of the manual.

The enabler is now installed on the host system. The software used to implement the cryptographic functions can now be linked against it and modified according to the desired workflow.

2.3 Enabler Configuration

The enabler is a dynamic linked library, hence it doesn't rely on configuration files, but, instead, allows dynamic configuration during the library calls. For better understanding on how to configure the options (like the attributes used for encryption or the location of the encryption keys), please see 3.1.

2.4 Troubleshooting

The library functions that compose the enabler have a very easy interface with few control on the error messages. Every function return a Boolean type that notify the caller whether the call was success or not.

Within the library the g_log() logging subsystem from glib has been used. According to glib's manual (see [10]), to raise or lower the logging some environment variables can be set. This will give the user the ability to better understand what's going wrong inside the library stack.

3 User and Programmer Guide

This enabler is a software library, hence the user guide actually can also be considered to some extent the programmer guide, since the end user of a C library is indeed a programmer. The user guide provides specific instructions for the user to deploy the example use case illustrated in Figure 1.

3.1 User Guide

There are four main steps to be undertaken in order to work with libkpabe. Herein each of them is described in details. Note that the regression test in tests/test.c shows a practical example of using the functions corresponding to each step. Wherever a variable is not listed, we consider its declaration/assignment as done in the previous steps.

3.1.1 Setup

This step is performed by a trusted party that is called Trust Authority (TA). During the setup the TA initializes the KP-ABE crypto system by exhaustively enumerating the attributes that are part of the authorized attributes universe and passing them to the setup function. Subsequently, by adding some randomness, this function creates a public key and a master key. The former is widely distributed to all the participants in the scheme, as in traditional public key encryption systems, while the latter is securely stored by the TA and it is never shared, not even with the authorized parties that are part of the ABE crypto system.

Example of use:

```
gchar universe[] = "t1 t2 t3";
GByteArray* pub;
GByteArray* msk;
if (kpabe_sys_setup(universe, &pub, &msk)) {
    print_some_error();
}
```

Attributes can be any string of letters, digits and underscores, and they must begin with a letter. The keywords “and”, “or” and “of” are reserved for the policy language and may not be used for an attribute.

An example of attributes are space separated strings, where each string identify the network authorized to request a private key from the TA. Such strings may be any identifiers such as SSID, PLMN-ID, authentication server identifier, etc.

The generated public and master keys can be accessed through pointers to their respective GByteArray variables. The GByteArray data type is provided by GLib.

The setup function returns FALSE if an uninitiated or an empty attributes string is passed as an argument.

This step is performed once when the system is booted up, and it’s made by a central trusted authority (called TA).

3.1.2 Key generation

Every authorized participant in the system asks the TA for the release of a private key. This key, paired with the common public key is used for decryption. The private key is therefore secret and must be securely stored by the authorized participant.

Example:

```
gchar policy[] = "t2";
GByteArray* prv;
if (kpabe_entity_keygen(policy, pub, msk, &prv)) {
    print_some_error();
}
```

The generation of a new private key requires the master key and the public key that were generated for the current cryptographic system. The policy parameter, which is a string passed to the keygen function, specifies what set of attributes the private key is able to decrypt from. While attributes can be any string of letters, digits and underscores, and they must begin with a letter, the keywords “and”, “or” and “of” are reserved for the policy language of the underlying library (libcelia) and may be used to logically combine authorized attributes in order to form a policy (hence they are forbidden as attributes). A policy is also known as an access structure which indicates the actual private keys that can decrypt the message encrypted under a set of authorized attributes.

This function provides the user with a pointer to the GByteArray variable that is used to store the generated private key.

The keygen function will return FALSE if the attribute(s) declared in the policy variable is/are not included in the set of the attributes universe specified in the setup function.

3.1.3 Encryption

Any UE that wants to send the network with information that need to be kept secret, performs the encryption of the secret by using the public key of the network and the related encryption attributes. Herein an example for the IMSI encryption.

Example of use:

```
char* imsi = "222011111111111";
guchar policy[] = "t2";
guchar* beimsi;

if (kpabe_imsi_encryption(pub, imsi, strlen(imsi), policy, &beimsi)) {
      print_some_error();
}
```

The input parameters of the encryption function are the public key data, a non-empty string containing the data to encrypt and the policy attributes. The list of attributes included in the encryption policy string directly determines which private keys are able to decrypt the message on the receiving system (network element).

The function stores the encrypted IMSI in a Base64-encoded string. It returns an FALSE if the policy attributes string has not been properly initiated, is empty or includes attributes that are not part of the attributes universe stored in the public key.

3.1.4 Decryption

The decryption is performed by the authorized participant that owns the right private key. Anyone participating in the KP-ABE scheme can encrypt a message with the attribute(s) bound to an authorized private key, but only that specific private key can decrypt it and obtain the original message in clear text.

Example of use:

```
guchar *decimsi;
if (kpabe_imsi_decryption(pub, prv, beimsi, &decimsi)) {
      print_some_error();
}
```

The decryption function requires the public key, a private key and the Base64-encoded encrypted data. The result of the decryption operation is saved in a string.

FALSE is returned if the selected private key is not authorized to decrypt the encrypted data because of the lack of attributes satisfying the encryption condition.

3.1.5 Deployment example

The enabler is not an end-user software, hence it has not a deployment procedure to follow. The extended software that links the enabler should have them. In this chapter we will provide an example of an extended software (hostapd) that make use of the enabler and its deployment. We don't provide the source code or patch for hostapd since it is not the enabler itself, but it's an example of the applied strategy.

All components are freely available on the Internet, while libkpabe is provided by the 5G ENSURE project under the GPL open source license.

The high level steps that a user/programmer has to perform in order to set up a functioning system for the use case illustrated in Figure 1, follow.

1. Get USIMs with known secrets (Ki). If they are not available, acquire a programmable USIM and a USB card reader. If the USIM does not come with preprogramed parameters or they have not been disclosed, it can be programmed with the desired authentication values using tools such as pySim [9]. At this step the user should know the IMSI, the Ki and the OPc (calculated from the Ki and the OP) values of his programmable USIM card. The latter approach is used for the proof of concept implemented to demonstrate the usage of the libkpabe enabler, since it's easier and more flexible.
2. Download the hostap project [8]. It ships wpa_supplicant (the component for the client system) and hostap+hlr_auc_gw (the authentication system component for the network side). Some changes are needed on those software in order to encrypt/decrypt the IMSI (just an example for any of the secrets we would like to protect).
3. Recompile the above components on 2 systems (client system and authentication system) with the libkpabe installed (see 2 for details).The aforementioned changes should allow the user to specify the details for kpabe (e.g. the attributes and the cryptographic keys paths).
4. Edit the hostapd runtime configuration file on the server. Configure the EAP-AKA milenage file that is used by the hlr_auc_gw component to authenticate the users. Using the parameters obtained in the first step, add a line for each allowed user matching the following format:

   ```
   <IMSI> <Ki> <OPc> <AMF> <SQN>
   ```

   The `AMF` and `SQN` parameters can be set respectively to `0000` and `000000000000`.

By following these instructions the user/programmer should be able to setup a running system as illustrated in Figure 1.

3.2 Programmer Guide

This section is not applicable as the enabler is not programmable.

4 Unit Tests

The libkpabe library is shipped with regression tests that run the aforementioned steps (i.e., setup, key generation, encryption, decryption). They are based on autotools test suite. To perform the tests just type:

```
make check
```

in the top directory of the libkpabe project. A short summary will illustrate the tests result. A report like the one illustrated below shows a healthy situation:

```
==================================================================================
Testsuite summary for libkpabe 0.1
==================================================================================
# TOTAL: 1
# PASS:  1
# SKIP:  0
# XFAIL: 0
# FAIL:  0
# XPASS: 0
# ERROR: 0
==================================================================================
```

Exclusively for tests purposes, in order to obtain deterministic verifiable results, the libkpabe randomness has been disabled while running the regression tests.

The input of the regression test consists of known strings for the attributes universe and for the setup/encryption attributes policies. Every function of the library is invoked sequentially following the `setup→keygen→encryption→decryption` order. Since the test is running in a deterministic environment, the intermediate results of each function call are known a priori and can be compared to the expected data output. A regression test returns with a failure code if there is a mismatch between the produced data and the predefined one at any stage of the program.

4.1 Information about Tests

There are four unit tests available to be performed on libkpabe in order to ascertain its correct functioning, one unit for each of its main functions. The tests are run within the same executable (that's why the test report from autotools shows just 1 test).

4.1.1 Unit Test 1

This unit test checks the correct functionality of the setup function and is implemented as follows :

```
if (kpabe_sys_setup(universe, &pub, &msk)) {
        print_some_error();
}
if (compare_keys(PUB_KEY, pub->data, pub->len)) {
        print_some_error();
}
if (compare_keys(MASTER_KEY, msk->data, msk->len)) {
      print_some_error();
```

```
}
```

A known string containing the attributes universe is supplied to the setup function which produces a public key and a master key. The key generation process is deterministic since the PBC library randomness has been disabled in the regression test. This implies that the results can be matched against the expected output of the setup function, as shown by the `compare_key()` being called two times on each generated key. This unit test fails if any of the produced key is different than the expected

4.1.2 Unit Test 2

This unit test checks the correct functionality of the key generation function and is implemented as follows:

```
if (kpabe_entity_keygen(policy, pub, msk, &prv)) {
      print_some_error();
}
if (compare_keys(PRIVATE_KEY, prv->data, prv->len)) {
      print_some_error();
}
```

The `kpabe_entity_keygen()` function accepts a known string containing the chosen policy, the public key and the master key generated in Section 4.1.1. The output of the function is a private key with the input policy. Since the results are deterministic, the private key can be compared with its expected output using the `compare_key()` function. If the payloads differ, this unit test fails.

4.1.3 Unit Test 3

This unit test checks the correct functionality of the encryption function and is as follows:

```
if (kpabe_imsi_encryption(pub, message_clear, strlen((char*)message_clear),
            policy, &curr_message_enc)) {
      print_some_error();
}
if (compare_payloads(message_enc, strlen((char*)message_enc), curr_message_enc,
            strlen((char*)curr_message_enc))) {
      print_some_error();
}
```

The encryption function receives the public key generated in Section 4.1.2, a known string containing the IMSI to encrypt, the IMSI length and a known encryption policy string as input. It returns the Base64-encoded encrypted IMSI whose payload can be compared to the expected output since the encryption process is deterministic for the regression test. This unit test fails if there is a mismatch in the `compare_payloads()` function.

4.1.4 Unit Test 4

This unit test checks the correct functionality of the decryption function and is implemented as follows:

```
if (kpabe_imsi_decryption(pub, prv, curr_message_enc, &curr_message_dec)) {
```

```
        print_some_error();
}
if(compare_payloads(message_clear, strlen((char*)message_clear),
            curr_message_dec, strlen((char*)curr_message_dec))) {
        print_some_error();
}
```

The input parameters of the decryption function are the public key generated in Section 4.1.1, the private key generated in Section 4.1.2 and the Base64-encoded encrypted IMSI calculated at section 4.1.3. The decryption function returns the decrypted IMSI which is compared to the known string containing the plain IMSI passed to the encryption function as input parameter in Section 4.1.4. This unit test fails if the two strings do not match.

5 Acknowledgements

We want to acknowledge the developers and contributors of the open source projects used in the present work, especially libcelia, PBC, hostapd, and the Linux Ubuntu community.

6 Abbreviations

5G-PPP	5G Infrastructure Public Private Partnership
ABE	Attribute Base Encryption
KPABE	Key Policy Attribute Base Encryption
AKA	Authentication and Key-agreement
EAP	Extensible Authentication Protocol
EAP-AKA	Extensible Authentication Protocol Authentication Key Agreement (RFC 4187)
IMSI	International Mobile Subscriber Identity
PoC	Proof of Concept
GNOME	GNU Object Model Environment

7 References

[1] 5G ENSURE Deliverable D3.1, “5G-PPP security enablers technical roadmap (early vision)”

[2] 5G ENSURE Deliverable D3.2, “5G-PPP security enablers open specifications (v1.0))”

[3] Goyal, Pandey, Sahai, Waters, Attribute Based Encryption for Fine Grained Control of Encrypted Data, https://eprint.iacr.org/2006/309.pdf

[4] PBC, The Pairing-Based Cryptography Library, https://crypto.stanford.edu/pbc/

[5] The GMP library, the GNU Multiple Precision Arithmetic library, https://gmplib.org/

[6] Libcelia, a static Linux library implementing primitive kpabe operations,

https://github.com/gustybear/libcelia

[7] GLib, https://developer.gnome.org/glib/.

[8] hostapd, IEEE 802.11 AP, IEEE 802.1X/WPA/WPA2/EAP/RADIUS Authenticator,

https://w1.fi/hostapd/

[9] pySim, A python tool to program magic SIMs, http://cgit.osmocom.org/pysim/

[10] https://developer.gnome.org/glib/stable/glib-Message-Logging.html