

**Cookbook
End-to-end Encryption
Unknown recipient
Version 1.3**

This document is provided to you free of charge by

The eHealth platform

**Willebroekkaai 38 Quai de
Willebroeck**

1000 BRUSSELS

All are free to circulate this document with reference to the URL source.

Table of contents

Table of contents	2
1 Document management	3
1.1 Document history.....	Error! Bookmark not defined.
1.2 Document references	3
1.3 Versioning.....	3
1.4 Goal of the document.....	3
2 Global overview of the ETEE service	4
2.1 Overview of all required components.....	4
2.2 High level schema of the Unknown Recipients functionality	4
2.3 Detailing the steps.....	5
3 Quick starting guide.....	9
3.1 Prerequisites	9
3.2 eHealth Authentication Certificate	9
3.3 ETK.....	9
3.4 The Crypto Library	10
3.5 Step-by-step instructions.....	11
4 General information.....	12
5 The Key Generation Storage Service (KGSS)	14
5.1 GetNewKey.....	15
5.2 GetKey	18
6 The Crypto Library	22
6.1 SealForUnknown	22
6.2 UnsealByUnknown.....	24
6.3 Error and failure messages.....	27
7 Implementation aspects	28
7.1 Development and test procedures	28
7.2 Request for release in production	28
7.3 Maintenance, support and monitoring of the service.....	28
8 Errors and solutions	29
8.1 EteeResponseType	29
8.2 Invalid Key Size.....	29
9 Risks	30
10 Contact.....	31
11 Licenses	32
11.1 Apache	32
11.2 Bouncy Castle.....	32
12 Annex.....	33



1 Document management

1.1 Document references

These documents can be found in the technical library of the eHealth portal.

ID	Title	Version	Date	Author
1	Requesting eHealth Certificates	v.2.0	May 2010	eHealth
2	Cookbook “Cookbook v.2.0: End-to-end Encryption for a known recipient (addressed messages)”	v.2.1	April 2010	eHealth
3	Cookbook STS			eHealth
4	Glossary	1.0		eHealth

1.2 Versioning

The versions of the related components and documentation of this document are:

Reference	Version	Date
Crypto Library	v.1.6.1	2010-08-20
Requesting eHealth Certificates	v.2.1	2010-04

1.3 Goal of the document

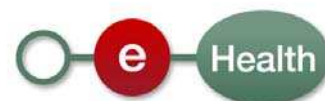
This document is intended as an integration reference for the eHealth ETEE¹ basic service *ETEE-Unknown Recipients*. The target audience is software integrators implementing the ETEE services in their own custom application. This document is not a software manual for end users containing instructions and screenshots of an application. Instead, it explains the concepts, principles and interface of the KGSS web service and the Crypto Library.

The *unknown recipient* functionality requires the use of all the *known recipient* components. You need the cookbooks “*ETEE known recipients*” and “*ETEE unknown recipients*” in order to integrate the “*ETEE Unknown recipients*” functionality.

This document will provide you with all the necessary elements to get you started developing. In this context it explains:

- the main concepts and principles;
- the use of KGSS web services;
- the use of the unknown recipients functionality offered by the Java Crypto Library.

¹ End-to-end Encryption



2 Global overview of the ETEE service

Please refer to the ETEE Known Recipients cookbook for a brief introduction and some references to important concepts and technologies. Please note that there is only one Crypto Library that is used for Known and Unknown Recipients.

The End-To-End Encryption basic services only offer building blocks that allow to integrate secure communications in applications.

It does not offer a prepackaged 'End-To-End' business solution. This means you have to create your own client application with an implementation of:

- an ETK Client;
- a KGSS Client;
- a software that integrates the Crypto Library;
- a way to pass on a message reference to a message receiver;
- a way to pass on a key reference to a message receiver (optional if a key reference is used in the MSS);
- a Message Storage Center (you could store the message reference in the MSS).

2.1 Overview of all required components

The ETEE unknown recipients service builds on the functionality of the known recipient's service:

- the ETK;
- the ETK Depot;
- the Crypto Library functionality for known recipients (Seal/Unseal/VerifyEtk).

Furthermore, it consists of two additional parts:

- The Key Generation Storage Service (KGSS) that creates, stores and delivers symmetrical keys. The KGSS is a service provided by eHealth that does not store messages.
- The Crypto Library functionality for unknown recipients (SealForUnknown/UnsealByUnknown) that uses symmetrical encryption.

eHealth does not provide a Message Storage Center, only a Key Generation and Storage Service. eHealth does never store medical information, not even if this information is encrypted.

2.2 High level schema of the Unknown Recipients functionality

These additional web services for Unknown Recipients and Crypto Library methods need a working solution. The example below describes how to use them in a business scenario. Note that this is an example and that these steps largely depend on the implementation requirements of the eHealth client.

The following high level schema represents the steps the sender and the receiver need to carry out in order to communicate in a secure way.

The lines in bold indicate that Unknown Recipients building blocks are used. These steps will be described in detail in the next section. The remaining steps are examples on how the solution could be implemented, but they are out of scope. eHealth only offers services for encrypting and decrypting information (the library) and key storage (the ETK and KGSS).



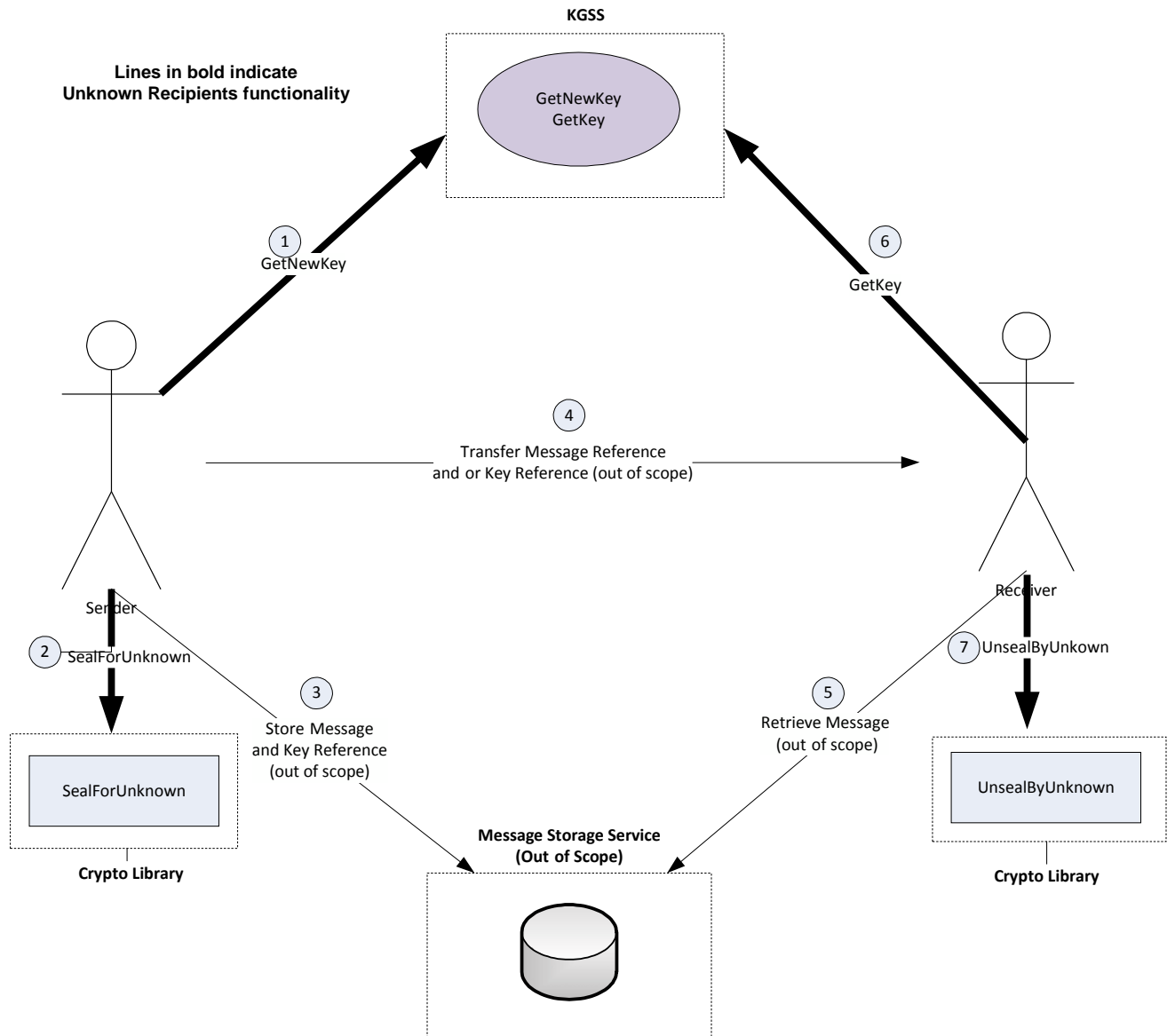
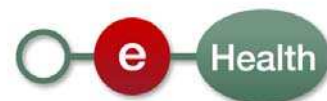


Figure 1: High level schema

2.3 Detailing the steps

Step 1. The sender requests a new key and calls the KGSS. The KGSS returns a key and its corresponding key identifier. The secured part of the message request must be sealed with the ETK of the KGSS. The secured part of the response is sealed by the KGSS and must be unsealed by the sender.

The functionality for interaction with the KGSS is covered by the KGSS web service. Please consult section "5.1



GetNewKey” on page 15.

This request/response communication cycle with the KGSS requires the use of the “known recipients” functionalities to encrypt/decrypt the communication between the sender and the KGSS.

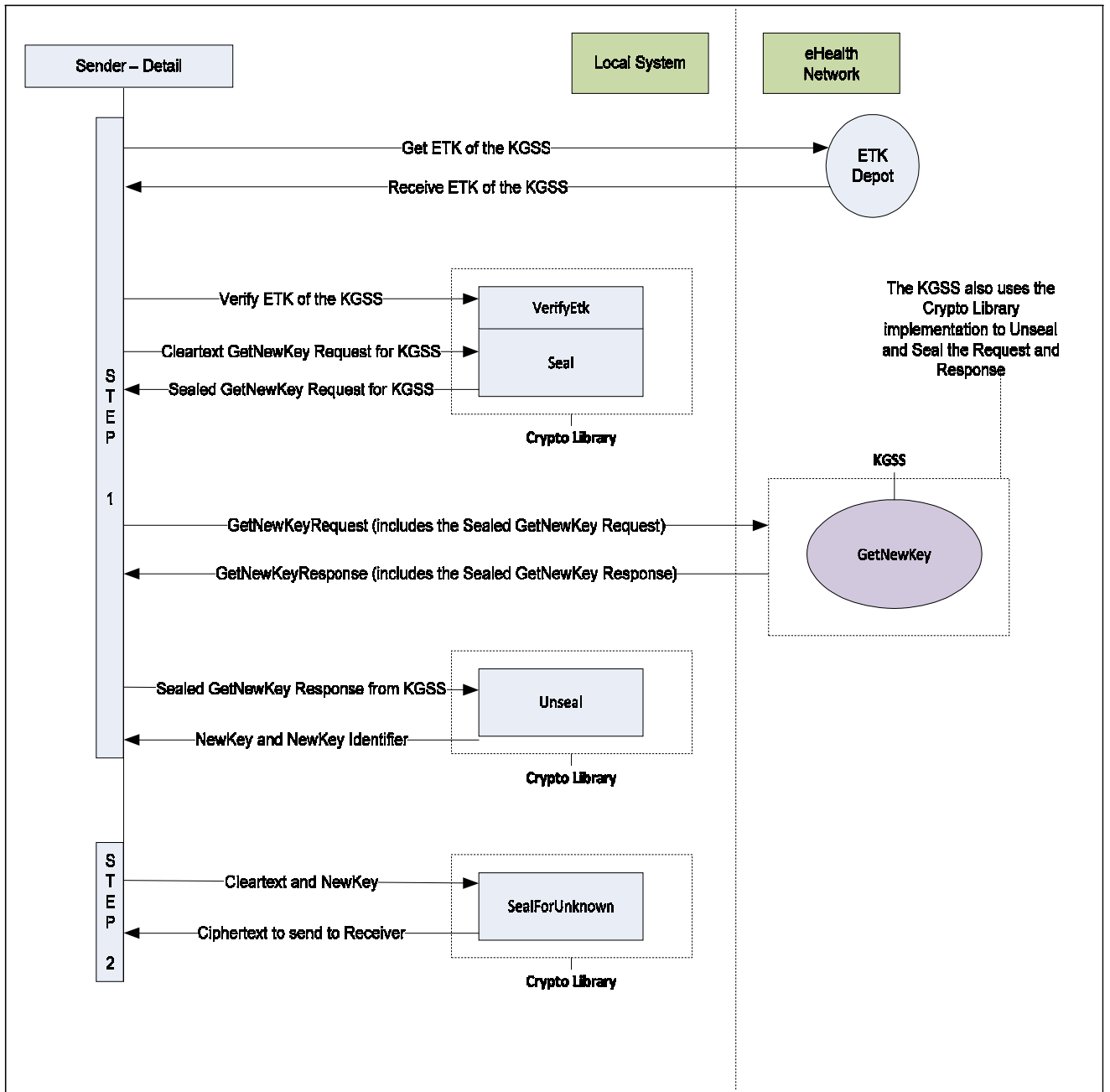


Figure 2: Detail GetNewKey

Step 2. The sender can use the Crypto Library (implemented in his own secured system) to SealForUnknown the message by using the new key.

This functionality is covered by the Crypto Library. Please consult section “6.1 SealForUnknown” on page 22”.

Step 3. The message must be stored in a Message Storage Service. This service is defined as the MSS. It is not a building block of the ETEE unknown recipient basic service (out of scope). Therefore eHealth does NOT provide



any kind of message storage system, nor does it provide an interface or authorization and authentication mechanisms. This system must be entirely implemented by the eHealth customer as a part of his project. eHealth does not store these confidential messages.

Step 4. The key and/or message reference of the message is passed onto the recipient of the message. This part does not lie within the scope of the ETEE unknown recipient basic service. The key identifier and/or message reference can be passed on by any possible means (SMS, paper, email, web service, file, ...).

Step 5. The receiver can retrieve the **message** with its message reference from the MSS.

Step 6. The receiver needs to obtain the **key** that corresponds to the key identifier from the KGSS, depending on which element has been provided.

The functionality that deals with the interaction with the KGSS is covered by the KGSS web service. Please consult section “5.2 GetKey” on page 18.

This request/response communication cycle with the KGSS requires the use of the “known recipients” functionalities to encrypt/decrypt the communication between the receiver and the KGSS.

If the receiver has both the key reference and the message reference available to him, steps 5 and 6 can be inverted.

Step 7. The retrieved message can be decrypted by using the ‘UnsealByUnknown’ method of the Crypto Library. This requires the symmetrical key and the encrypted message as input arguments.

This functionality is covered by the Crypto Library. Please consult section “6.2 UnsealByUnknown” on page 24.



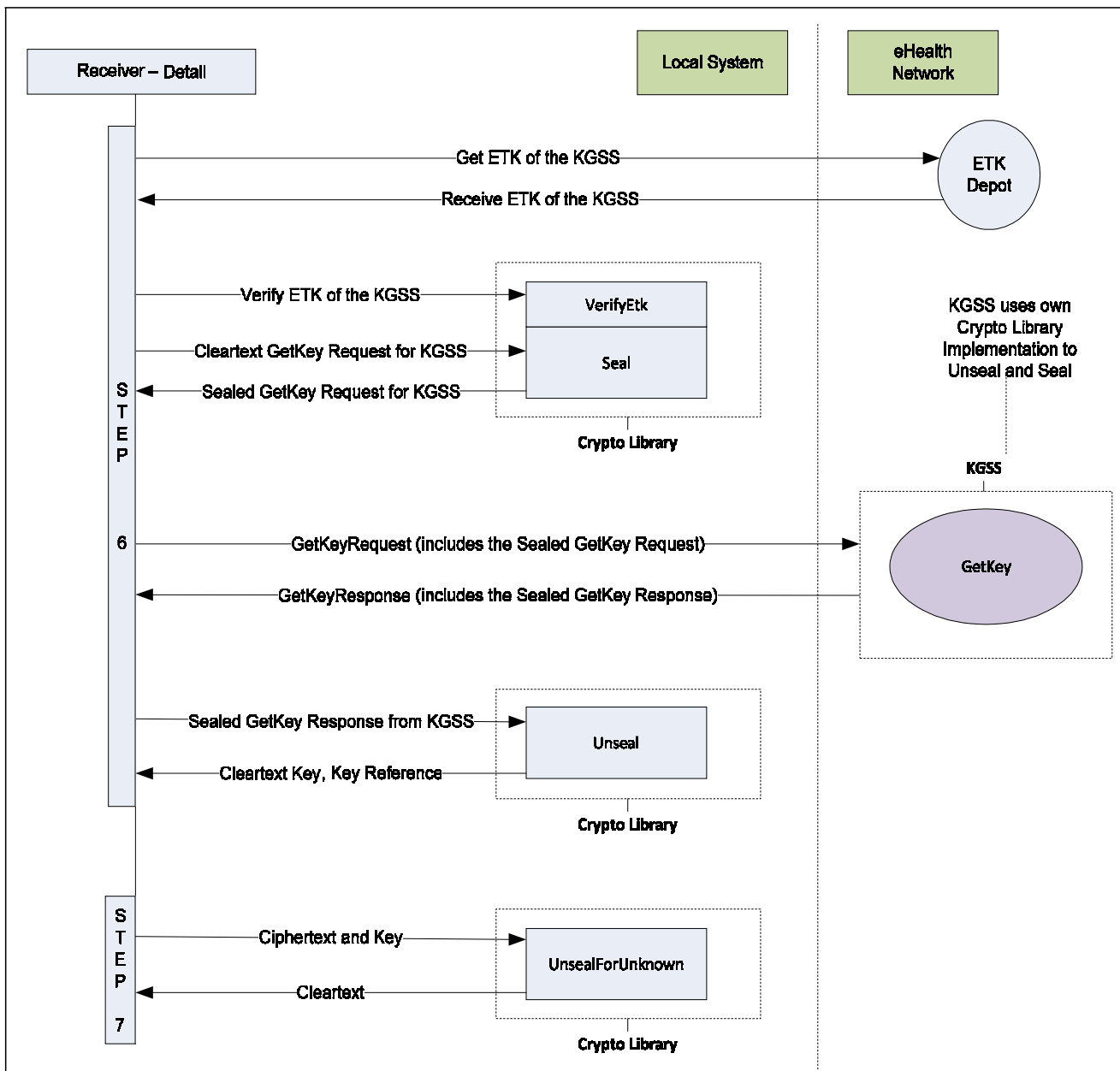


Figure 3: Detail GetKey



3 Quick starting guide

3.1 Prerequisites

3.1.1 Java

The ETEE Crypto Library can be used with any operating system that supports Java version 1.5.0 or above. Developers must use the JDK version. Clients that only use the ETEE service must have the JRE installed. Java can be downloaded from Sun Microsystems.

<http://java.sun.com>

3.1.2 Java Cryptography Extension (JCE)

You must download and install the Java(TM) Cryptography Extension Unlimited Strength Jurisdiction Policy Files 5.0.

<https://cde.sun.com>

3.1.3 Bouncy Castle

The library has been tested with Bouncy Castle 1.39 or higher for Java version 1.5.0. The required Bouncy Castle libraries are delivered with the ETEE package. The required JARs can also be downloaded from the Legion of the Bouncy Castle.

www.bouncycastle.org

3.1.4 Get Log 4j

This component is required to show exactly what the Crypto Library is doing when it is used, so that this log information can be shown or saved.

Apache log4j is a Java based logging utility. You must use version 1.2 or higher. The software can be downloaded from the Apache Software Foundation.

www.junit.org

logging.apache.org

3.2 eHealth Authentication Certificate

To use the ETEE service, you must have an eHealth authentication certificate. Please contact eHealth if you have any doubts about the compatibility of your current certificate.

3.3 ETK

In order to secure communications between a client and the Key Generation Storage Service (KGSS), an ETK is required. Please see the instructions in the "Requesting eHealth Certificates" document.



3.4 The Crypto Library

The Crypto Library can be found in the technical library of the eHealth portal. The following files are part of the Crypto Library package. The minimum version of the Crypto Library that supports unknown recipients is v1.5.

D <https://www.ehealth.fgov.be/nl/page/website/home/platform/technicallibrary.html>

FR <https://www.ehealth.fgov.be/fr/page/website/home/platform/technicallibrary.html>

The java documentation, examples, tests, test resources and additional libraries that are required and useful to test and integrate the Crypto Library are packaged as follows.

File: etee-crypto-[version].jar

3.4.1 Java Documentation

File: etee-crypto-[version]-javadoc.jar

The Java code has been thoroughly documented. A compilation of all this documentation can be found in this file.

3.4.2 Examples and tests

File: etee-crypto-[version]-tests.jar

The examples within the packages show error and exception handling. These examples will be discussed in detail below in this cookbook. The tests contain JUnit tests that can be used to check your configuration. All JUnit tests should pass successfully in order to confirm you have a good working environment. All JUnit files can be identified by their 'TEST' prefix.

3.4.3 Test sources

File: etee-crypto-[version]-test-sources.jar

The test sources contain all necessary files for Known and Unknown Recipients, including passwords to run the tests. These files are possibly key stores, test messages or ETks. Private Keystores contains Alice's and Bob's private authentication key and private encryption key including their corresponding certificates. As documented in the test resources, the password is always "test".

3.4.4 Additional libraries

These additional libraries can be identified as Bouncy Castle libraries, which are required at runtime. These are essential libraries that implement the basic cryptography functionalities. The additional libraries for Log4j Bouncy Castle must be copied to the Java CLASS PATH.

Files: bcmath-jdk15-1.44.jar
bcprov-jdk15-1.44.jar

The other libraries are testing dependencies that are useful if you want to run the JUnit tests or debug.

Files: log4j-1.2.13.jar
junit-4.7.jar



3.5 Step-by-step instructions

In order to use the building blocks that are specific to unknown recipients, the following actions are required:

1. Integrate and comply with the prerequisites (Java, JCE, bouncy castle).
2. Obtain an eHealth Authentication Certificate. The procedure 'Requesting eHealth Certificates' can be found in the eHealth Technical Library.
3. Obtain an ETK (using the ETEE Requestor application). The procedure 'Requesting eHealth Certificates' can be found in the eHealth Technical Library. In order to use the 'Unknown Recipients' functionality, you must also have an ETK for secure communications from and to the KGSS.
4. The Unknown Recipients functionality is based on the Known Recipients functionality. Therefore you need the Known Recipients components and functionality. This means:
 - ETK Depot (getEtk);
 - Crypto Library (Seal/Unseal/VerifyEtk).
5. Integrate the unknown recipients functionality:
 - KGSS;
 - Crypto Library version 1.5 or above (SealForUnknown/UnsealByUnknown).

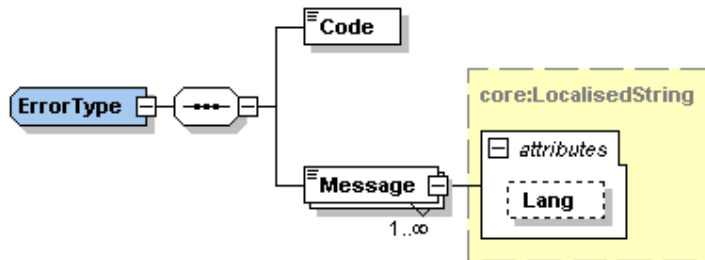


4 General information

The following section explains some complex elements that are defined as a type. These types are used as reusable definitions in the KGSS web services.

4.1.1 ErrorType

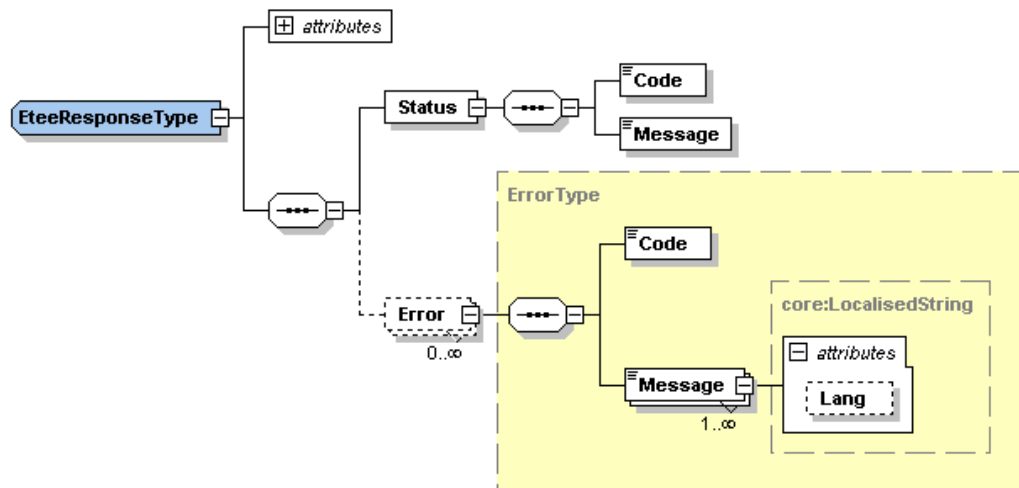
If an error occurs, the following information will be displayed.



Field name	Descriptions
Code	A custom code has to be defined. Three-letter error code.
Message	The error message. The 'Lang' attribute contains the language of the error message.

4.1.2 EteeResponseType

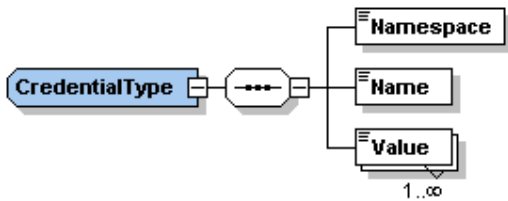
The EteeResponseType is a custom type that is used to provide generic information with every response. This standard way of providing feedback information enhances readability.



Field name	Descriptions	
Status	The <i>Status</i> block will contain a code and a message language reference.	
	Code	A three-letter error code that defines the status of the request.
	Message	Returns the message language of the message, if there is any.
<i>Error (optional)</i> ²	This custom type is also discussed in detail in the ‘Types’ section of this document. See section “4 General ” on page 12.	

4.1.3 CredentialType

A credentialType is used within the ETEE project to identify an AllowedReader or an ExcludedReader. These credentials are structured as follows:



Field name	Descriptions
Namespace	The namespace of the Credential attribute, e.g. - “urn:be:fgov:ehealth:certified-namespace”
Name	The name of the Credential attribute, e.g. - “urn:be:fgov:ehealth:doctor-nihii”
Value	The value of the Credential attribute, e.g. - “74042015445”

² This field is optional because a response does not always contain an error. Setting this field as a mandatory field would result in an invalid response for each request that is processed correctly.



5 The Key Generation Storage Service (KGSS)

The KGSS is only accessible through web services. The URL can be obtained by contacting the eHealth platform. You must have an eHealth authentication certificate in order to access the service. There are 2 basic functionalities, which are both described in detail and explained in relation to their position in the high level overview schema.

All KGSS communication is encrypted. This means that:

1. requests from the message sender or receiver to the KGSS are sealed by using the ETK of the KGSS;
2. responses from the KGSS to the message sender or receiver are sealed by using the sender's or receiver's ETK.

All of these steps must be completed for each message:

- GetNewKey – see step 1 on Figure 1: High level schema (p. 5) - the sender calls the KGSS with a request to return him a new key and an identifier. This key is used by the sender to encrypt information by using the SealForUnknown method of the Crypto Library.
E.g. the sender requests a new key. The KGSS creates and stores a new key. The sender receives a response with "AFAA18926FDF65C367BF9A838DAB4EF3" as the key identifier and "EE37154F94DBF8F8D42E218397B3EA24" as the key.
- GetKey – see step 6 on Figure 1: High level schema (p. 5) - the receiver of the sender's actual encrypted message can call the KGSS, asking for a specific existing key once he has obtained the key reference³. This can be done by using the key's 'key identifier' as a reference. The KGSS returns the corresponding key. This key can be used by the receiver of the message in order to decrypt the information by using the Crypto Library's UnsealByUnknown method.
- e.g. the sender requests the key for the previously created key identifier "AFAA18926FDF65C367BF9A838DAB4EF3". The KGSS returns the key "EE37154F94DBF8F8D42E218397B3EA24" in the response.
- Some examples of Key Identifiers and Keys that are stored in the KGSS:
Each 'Key Identifier' is unique and corresponds to exactly one 'Key'.

Key Identifier	Key
AFAA18926FDF65C367BF9A838DAB4EF3	EE37154F94DBF8F8D42E218397B3EA24
201E606A1EDA1B61414B6A746A1417D0	852C061A622857776F3CB3313318E2A2
883CDBA88E2B981A531556719838A769	6F838BF8D8F84C70B87B529E10AE4C2F

- please note that depending on the size of the secured information, the size of the 'SealedContent' elements shown in the different examples will be larger.

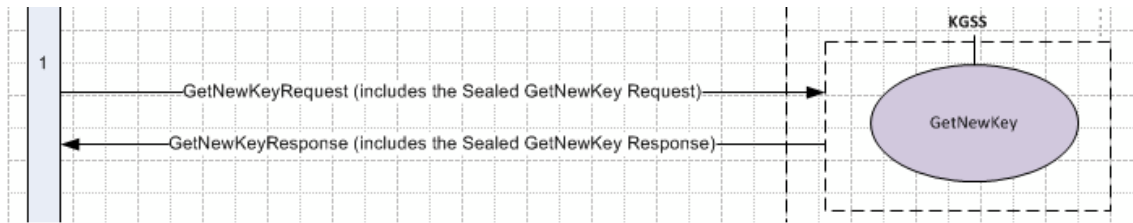
³ How and when reference information is passed, is out of scope. Theoretically a receiver can already have received the message from the sender, but he cannot have received the key reference to physically obtain it.



5.1 GetNewKey

Step 1 of the high level overview.

This method will create and store a new symmetrical encryption key. It is used by the sender. The key and its identifier are returned to the requestor.



Detail from functional step 1 in the high level overview: get a new key from the KGSS by the sender.

5.1.1 Requesting the new key

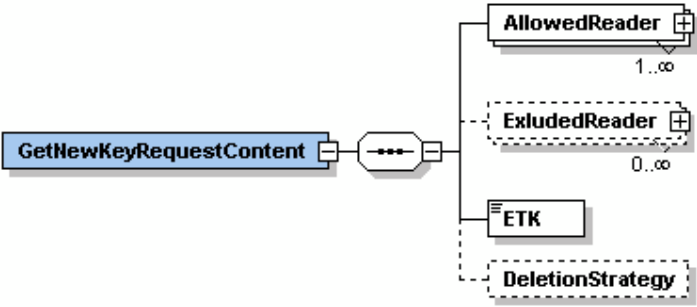
For each Unknown Recipient message sent, a new key needs must to be requested.

5.1.1.1 Structure of the GetNewKeyRequest



The GetNewKeyRequest contains the following information:



Field name	Descriptions					
SealedNewKeyRequest/ SealedContent	<p>----- The Following information is Encrypted for the KGSS by the sender -----</p> 					
	<table border="1"> <tr> <td data-bbox="581 558 797 831">AllowedReader</td> <td data-bbox="802 558 1466 831"> <p>A list of AllowedReaders. This list contains the readers that are allowed to obtain the newly created key. The allowed readers must be identified with the eHealth CredentialType. This custom type is discussed in detail in the 'Types' section of this document. See section "4 General information" on page 12.</p> <p>E.g. All of type doctor and one dentist with NIHII='1234567890'</p> </td> </tr> </table>	AllowedReader	<p>A list of AllowedReaders. This list contains the readers that are allowed to obtain the newly created key. The allowed readers must be identified with the eHealth CredentialType. This custom type is discussed in detail in the 'Types' section of this document. See section "4 General information" on page 12.</p> <p>E.g. All of type doctor and one dentist with NIHII='1234567890'</p>			
	AllowedReader	<p>A list of AllowedReaders. This list contains the readers that are allowed to obtain the newly created key. The allowed readers must be identified with the eHealth CredentialType. This custom type is discussed in detail in the 'Types' section of this document. See section "4 General information" on page 12.</p> <p>E.g. All of type doctor and one dentist with NIHII='1234567890'</p>				
	<table border="1"> <tr> <td data-bbox="581 837 797 1121"><i>ExcludedReader (optional)</i></td> <td data-bbox="802 837 1466 1121"> <p>A list of ExcludedReaders. This list contains the readers that are explicitly excluded from getting the newly created key. The excluded readers must be identified with the eHealth CredentialType. This custom type is discussed in detail in the 'Types' section of this document. See section "4 General information" on page 12.</p> <p>E.g. For all doctors except for an excluded family member that is a doctor with NIHII='5678910123'</p> </td> </tr> </table>	<i>ExcludedReader (optional)</i>	<p>A list of ExcludedReaders. This list contains the readers that are explicitly excluded from getting the newly created key. The excluded readers must be identified with the eHealth CredentialType. This custom type is discussed in detail in the 'Types' section of this document. See section "4 General information" on page 12.</p> <p>E.g. For all doctors except for an excluded family member that is a doctor with NIHII='5678910123'</p>	<table border="1"> <tr> <td data-bbox="581 1127 797 1211">ETK</td> <td data-bbox="802 1127 1466 1211"> <p>The ETK provided by the requestor that will be used by the KGSS to encrypt the response. This is usually the ETK of the message sender.</p> </td> </tr> </table>	ETK	<p>The ETK provided by the requestor that will be used by the KGSS to encrypt the response. This is usually the ETK of the message sender.</p>
	<i>ExcludedReader (optional)</i>	<p>A list of ExcludedReaders. This list contains the readers that are explicitly excluded from getting the newly created key. The excluded readers must be identified with the eHealth CredentialType. This custom type is discussed in detail in the 'Types' section of this document. See section "4 General information" on page 12.</p> <p>E.g. For all doctors except for an excluded family member that is a doctor with NIHII='5678910123'</p>				
ETK	<p>The ETK provided by the requestor that will be used by the KGSS to encrypt the response. This is usually the ETK of the message sender.</p>					
<table border="1"> <tr> <td data-bbox="581 1218 797 1276"><i>DeletionStrategy (optional)</i></td> <td data-bbox="802 1218 1466 1276"> <p>Reserved for later use to define in which circumstances a key can be deleted. No further details are available at this time.</p> </td> </tr> </table>	<i>DeletionStrategy (optional)</i>	<p>Reserved for later use to define in which circumstances a key can be deleted. No further details are available at this time.</p>	<p>----- END Encrypted information -----</p>			
<i>DeletionStrategy (optional)</i>	<p>Reserved for later use to define in which circumstances a key can be deleted. No further details are available at this time.</p>					

5.1.1.2 Example

```

<?xml version="1.0" encoding="UTF-8"?>
<!--Sample XML file generated by XMLSpy v2007 sp2 (http://www.altova.com)-->
<GetNewKeyRequest xsi:schemaLocation="urn:be:fgov:ehealth:etee:kgss:1_0:protocol
ehealth-etee-kgss-schema-protocol-3_1.xsd"
xmlns="urn:be:fgov:ehealth:etee:kgss:1_0:protocol"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SealedNewKeyRequest><SealedContent>UjBsR09EbGhjZ0dTQUxUjBsR09EbNQ
UFBUUNBRU1tQ1p0dU1GUXhEUzhi</SealedContent>
</SealedNewKeyRequest>
</GetNewKeyRequest>

```

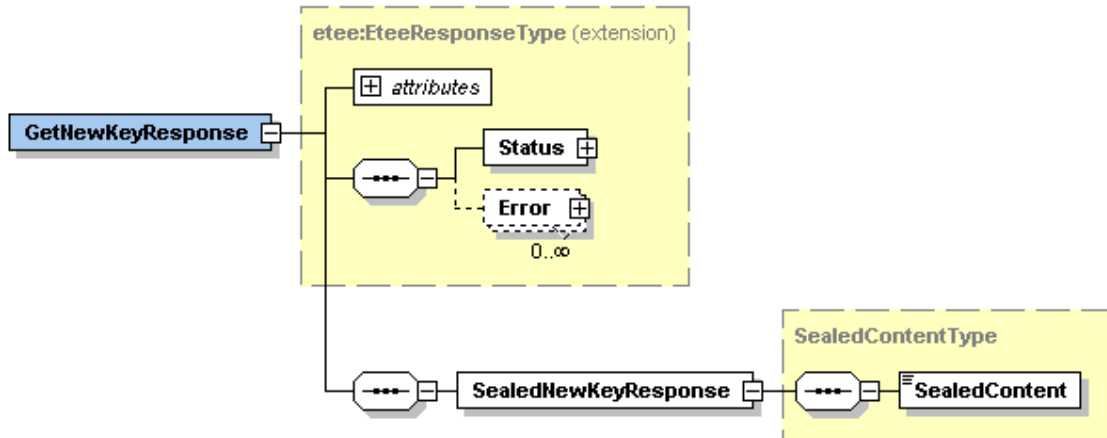


5.1.2 Receiving the newly generated key

This is the response to the message sender by the KGSS.

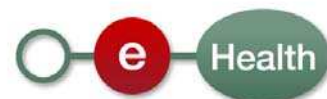
5.1.2.1 Structure of the response

This is a response to a GetNewKeyRequest. The client of the web service will receive the new symmetrical key and its identifier. The information is secured with the ETK that was sent in the request.



The GetNewKeyResponse contains the following information:

Field name	Descriptions				
EteeResponseType (extension)	This custom type is described in detail in the 'Types' section of this document. See section "4 General information" on page 12.				
SealedNewKeyResponse/ SealedContent	<p>-- The Following information is Encrypted with the ETK given in the Request --</p> <pre> graph LR GNRContent[GetNewKeyResponseContent] --- NKI[NewKeyIdentifier] GNRContent --- NK[NewKey] </pre> <table border="1"> <tr> <td>NewKeyIdentifier</td> <td>Contains the identifier (=reference) for the new key to be passed onto the recipient.</td> </tr> <tr> <td>NewKey</td> <td>The requested new key requires an encryption of your message.</td> </tr> </table> <p>..... END Encrypted information</p>	NewKeyIdentifier	Contains the identifier (=reference) for the new key to be passed onto the recipient.	NewKey	The requested new key requires an encryption of your message.
NewKeyIdentifier	Contains the identifier (=reference) for the new key to be passed onto the recipient.				
NewKey	The requested new key requires an encryption of your message.				



5.1.2.2 Example

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Sample XML file generated by XMLSpy v2007 sp2 (http://www.altova.com)-->
<n1:GetNewKeyResponse Id="String"
xsi:schemaLocation="urn:be:fgov:ehhealth:etee:kgss:1_0:protocol ehhealth-etee-kgss-schema-
protocol-3_1.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:n1="urn:be:fgov:ehhealth:etee:kgss:1_0:protocol">
  <Status><Code>200</Code><Message> The KGSSRequest was correctly
processed.</Message></Status>
  <n1:SealedNewKeyResponse>
  <n1:SealedContent>RU1tQ1p0dU1GUXhEUzhiUjBsR09EbGhjZ0dTQUxNQUFBU
UNB</n1:SealedContent>
  </n1:SealedNewKeyResponse>
</n1:GetNewKeyResponse>
```

5.2 GetKey

This method will retrieve an existing symmetrical encryption key. It is used by the receiver. The key identifier must be provided in the request and the symmetrical key itself is returned in the response. For this request you must have a valid SAML token.

The SAML token have been explained entirely in the cookbook of STS.

It is a part of step 6 that was previously described in the example scenario: the part in which the communication with the KGSS is turned into a request/response cycle.



(Detail from functional step 6 in the high level overview: get an existing key from the KGSS by the receiver).

5.2.1 Retrieving an existing key

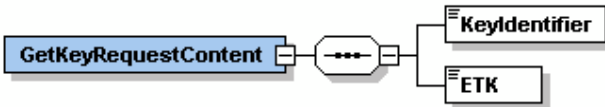
For each Unknown Recipient message, a key must be fetched from the KGSS.



5.2.1.1 Structure of the request

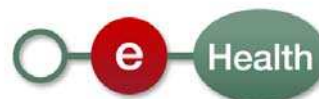


The GetKeyRequest contains the following information:

Field name	Descriptions			
SealedKeyRequest/ SealedContent	----- The Following information is Encrypted for the KGSS by the sender ----- 			
	<table border="1"> <tr> <td>KeyIdentifier</td> <td>The identifier of the key that must be retrieved.</td> </tr> <tr> <td>ETK</td> <td>The ETK of the requestor that will be used by the KGSS for encrypting the response. This is usually the ETK of the message receiver.</td> </tr> </table>	KeyIdentifier	The identifier of the key that must be retrieved.	ETK
KeyIdentifier	The identifier of the key that must be retrieved.			
ETK	The ETK of the requestor that will be used by the KGSS for encrypting the response. This is usually the ETK of the message receiver.			
	----- END Encrypted information -----			

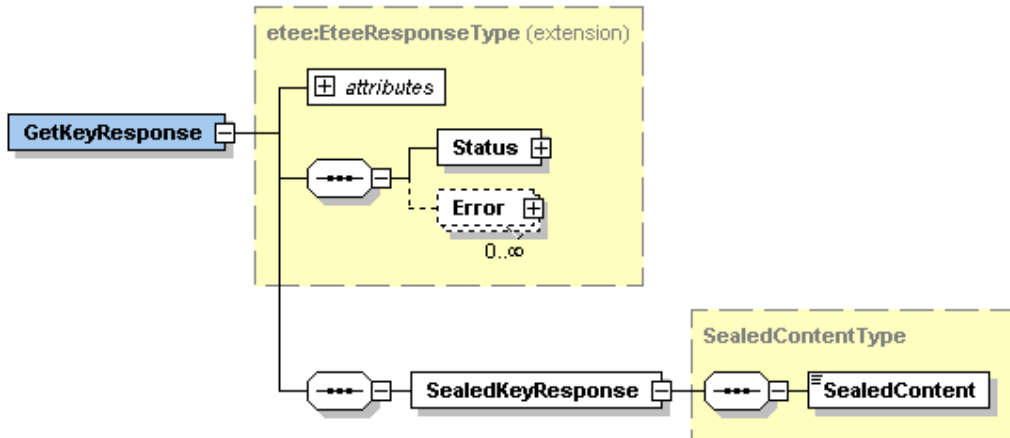
5.2.1.2 Example

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Sample XML file generated by XMLSpy v2007 sp2 (http://www.altova.com)-->
<GetKeyRequest xsi:schemaLocation="urn:be:fgov:ehhealth:etee:kgss:1_0:protocol ehhealth-
etee-kgss-schema-protocol-3_1.xsd" xmlns="urn:be:fgov:ehhealth:etee:kgss:1_0:protocol"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SealedKeyRequest><SealedContent>0dU1GUXhEUzhiUjBsR09EbGhjZ UjBsR
0dTQxNQFBUUNBRU1tQ1p </SealedContent>
  </SealedKeyRequest>
</GetKeyRequest>
```



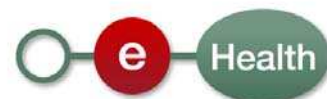
5.2.2 The GetKeyResponse

5.2.2.1 Structure of the response



The GetKeyResponse contains the following information:

Field name	Descriptions		
EteeResponseType (extension)	This custom type is discussed in detail in the 'Types' section of this document. See section "4 General information" on page 12.		
SealedKeyResponse/ SealedContent	<p>---- The Following information is Encrypted with the ETK given in the Request ----</p> <table border="1"> <tr> <td>Key</td> <td>In order to obtain the required symmetrical key, you must decrypt your message by using the Crypto Library UnsealByUnknown method.</td> </tr> </table> <p>----- END Encrypted information -----</p>	Key	In order to obtain the required symmetrical key, you must decrypt your message by using the Crypto Library UnsealByUnknown method.
Key	In order to obtain the required symmetrical key, you must decrypt your message by using the Crypto Library UnsealByUnknown method.		



5.2.2.2 Example of GetKeyResponse message

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Sample XML file generated by XMLSpy v2007 sp2 (http://www.altova.com)-->
<n1:GetKeyResponse Id="String"
xsi:schemaLocation="urn:be:fgov:ehhealth:etee:kgss:1_0:protocol ehealth-etee-kgss-schema-
protocol-3_1.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:n1="urn:be:fgov:ehhealth:etee:kgss:1_0:protocol">
<Status><Code>200</Code><Message> The KGSSRequest was correctly
processed.</Message></Status>
  <n1:SealedKeyResponse>
    <n1:SealedContent>GhjZ0dTQUxUjBsR09EbNQUBUUNBRU1tQ1p0dU1GUXhE
Uzhi</n1:SealedContent>
  </n1:SealedKeyResponse>
</n1:GetKeyResponse>
```



6 The Crypto Library

The Crypto Library version 1.5 and above offers an Unknown Recipients functionality. This library is fully compatible with the interfaces and functionalities that are already provided for the known recipients.

Please note that there are no separate releases for the Crypto Library containing the 'known recipients' and the 'unknown recipients' functionality. Therefore, it is in your own interest to keep the Crypto Library you are using up-to-date by checking the eHealth technical library at least every month.

Encrypting and decrypting are the main functionalities offered by the Crypto Library. For known recipients, the process is carried out by using an asymmetric encryption schema, in which anyone can encrypt by using the recipient's public key.

Symmetric encryption is used for the functionalities of the group of unknown recipients. This means that the same key is used for encrypting and decrypting. A symmetric encryption schema shows that this key is kept secret between the sender and the recipient(s) of a message.

This symmetric encryption key that is used for the eHealth ETEE 'Unknown Recipients' is created and exchanged by using the KGSS. Since we deal with unknown recipients, these recipients can obtain the required decryption key from the KGSS (once identified).

6.1 SealForUnknown

6.1.1 General

You will need:

- the data to Seal;
- an Encryption Key - the key provided by the KGSS GetNewKey method.

6.1.2 SealForUnknown code sample

```
/*
 * CVS file status:
 *
 * $Id: SealForUnknown.java,v 1.1 2010/03/04 13:20:01 jeh Exp $
 *
 * Copyright (c) Smals
 */
package be.smals.ehealth.eteecrypto.examples;

import java.io.File;
import java.io.IOException;
import java.security.KeyStoreException;

import javax.crypto.SecretKey;

import org.bouncycastle.cms.CMSException;
import org.bouncycastle.util.encoders.Base64;

import be.smals.ehealth.eteecrypto.encrypt.DataSealer;

/**
 * An example that illustrates the actions at Alice's (sender) side in
 * order to
```



```

* protect a message for an unknown addressee.
*
* @author jeh
*
* @since 1.5.0
*
*/
public class SealForUnknown extends AbstractExample {

    /**
     * @param args
     */
    public static void main(String[] args) {
        try {
            // 0. During initialisation, Alice creates her DataSealer
            that she can use to seal data.
            DataSealer alicesDataSealer = initSealing();

            // 1. Here Alice has a message for an unknown addressee that
            must be
            // sealed to secure its confidentiality, integrity and
            authenticity.
            byte[] messageToProtect = "This is a secret message from Alice
            for an unknown addressee.".getBytes();

            // 2. Get a new key and it's ID from the KGSS web service.
            SecretKey kek = getSecretKeyFromKgss();
            String kekId = getKekIdFromKgss();

            // 3. Seal the dataToBeSealed
            byte[] sealedData = alicesDataSealer.seal(messageToProtect,
            kek, kekId);

            // 4. Write the sealed data to your transportation medium
            // 4.1. As binary in a file...
            File cmsFile = writeToTransportMedium(sealedData,
            ExampleProperties.MSG_FROM_ALICE_TO_UNKNOWN);
            System.out.println("The sealed data is written in file: " +
            cmsFile.getAbsolutePath());

            // 4.2 Or as Base64 encoded binary text...
            byte[] encodedSealedData = Base64.encode(sealedData);
            System.out.println("base64-encoded sealedData : " + new
            String(encodedSealedData));
        } catch (IOException e) {
            System.err.println("The I/O operations during sealing the
            data failed or was interrupted.");
            e.printStackTrace();
        } catch (CMSEException e) {
            System.err.println("The data sealing failed. Check the log
            and root cause of the exception for details.");
            e.printStackTrace();
        } catch (RuntimeException e) {
            // RuntimeException must not be caught, this is just to
            document that
            // RuntimeExceptions are due to :
            // - improper setup of the runtime environment (consult the FAQ
            section in the Cookbook)
            // - not related to the Crypto Library of eHealth End-To-End
            Encrypton (check out the root cause).
        }
    }
}

```



```

        System.err.println("Your runtime environment is not
properly set up regarding dependencies " + "(Consult the FAQ), or there
are other issues not related to " + "the Crypto Library of eHealth End-
To-End Encryption " + "(check out the root cause of the exception)");
        e.printStackTrace();
    } catch (KeyStoreException e) {
        System.err.println("Alice's private authentication key
could not be retrieved from her Keystore.");
        e.printStackTrace();
    }
}

/**
 * @return
 */
private static String getKekIdFromKgss() {
    return ExampleProperties.getBase64EncodedKekId();
}
}

```

6.1.3 SealForUnknown console output

The expected output for this sealing example is:

"The sealed data are written in file: c:\javadev\prj\etee\crypto\message_from_alice_for_unknown.msg"

Note that depending on your Log4j settings, you can generate additional debugging output.

6.2 UnsealByUnknown

This method of the Crypto Library is used by the receiver of an encrypted message.

6.2.1 General

To unseal the data, you need:

- the data To Unseal;
- the Decryption key - the key provided by the KGSS GetKey method.

6.2.2 UnsealByUnknown code sample

```

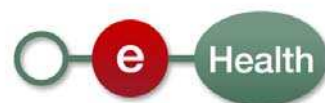
/*
 * CVS file status:
 *
 * $Id: UnsealByUnknown.java ,v 1.1 2010/03/04 15:06:36 jeh Exp $
 *
 * Copyright (c) Smals
 */
package be.smals.ehealth.etee.crypto.examples;

import java.io.IOException;
import java.io.InputStream;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.cert.CertificateException;

import javax.crypto.SecretKey;

import be.smals.ehealth.etee.crypto.decrypt.DataAuthenticationError;
import be.smals.ehealth.etee.crypto.decrypt.DataAuthenticationFailure;
import be.smals.ehealth.etee.crypto.decrypt.DataUnsealer;

```




```

import be.smals.ehealth.eteecrypto.decrypt.UnsealResult;

/**
 * An example that illustrates the actions at Bob's (recipient) side in
 * order to read and verify a sealed data message from Alice (sender).
 *
 * @author jeh
 *
 * @since 0.8.5
 */
public class UnsealByUnknown extends AbstractExample {

    /**
     * @param args
     */
    public static void main(String[] args) {
        try {
            // 0. During initialisation, Bob creates his DataUnsealer
            that he can use to
            // decrypt and verify incoming sealed data.
            DataUnsealer bobsDataUnsealer =
                initUnsealing();

            // 1. Now Bob receives some 'sealed data'
            byte[] sealedData =
                getSealedData(ExampleProperties.MSG_FROM_ALICE_TO_UNKNOWN);

            SecretKey kek = getSecretKeyFromKgss();

            // 2. Unseal the received message
            UnsealResult rslt = bobsDataUnsealer.unseal(sealedData, kek);

            // 3. Process the result of the unseal operation
            if (rslt.hasUnsealedData()) { // 3.A. The decryption operation
                succeeded
                if (rslt.isValid()) { // 3.A.A. There are no errors or
                    failures
                        // 3.A.A.1. Get the author
                        System.out.println("from author: " +
                            rslt.getAuthenticationCertificate().getSubjectDN());
                        // 3.A.A.2. Get the unsealed data
                        InputStream unsealedDataStream =
                            rslt.getUnsealedData();
                        byte[] unsealedData = getBytes(unsealedDataStream);
                        System.out.println("unsealed data: " + new
                            String(unsealedData));

                        } else { // 3.A.B. The data authenticity is not OK
                            // 3.A.B.1. Get the DataAuthenticationErrors or
                            DataAuthenticationFailures
                            // and do your specific security failure or error
                            processing
                            // BEFORE reading the unsealed data (otherwise you will
                            have an RuntimeException
                            for (DataAuthenticationError error :
                                rslt.getDataAuthenticationErrors()) {
                                    // e.g.
                                    System.err.println("error: " + error);
                                }
                            }
                }
            }
        }
    }
}

```



```

    }
    for (DataAuthenticationFailure failure :
rslt.getDataAuthenticationFailures()) {
        // e.g.
        System.err.println("failure: " + failure);
    }
    // 3.A.B.2. After checking the errors and failures you
can get the unsealed data
    InputStream unsealedDataStream =
rslt.getUnsealedData();
    byte[] unsealedData = getBytes(unsealedDataStream);
    System.out.println("unsealed data: " + new
String(unsealedData));

    // 3.A.B.3. and in most cases also the author
    if
(!rslt.getDataAuthenticationFailures().contains(DataAuthenticationFailure.A
UTHENTICATION_CERTIFICATE_EXPECTED_BUT_NOT_PRESENT)) {
        System.out.println("author certificate: " +
rslt.getAuthenticationCertificate());
    }
} else { // 3.B the decryption failed, there is no decrypted
data
    System.out.println("the msg could not be unsealed,
because:" + rslt.getDecryptionFailure());
}

} catch (CertificateException e) {
    System.err.println("creation of Bobs DataUnsealer failed:" +
e.getStackTrace());
} catch (KeyStoreException e) {
    System.err.println("creation of Bobs DataUnsealer failed:" +
e.getStackTrace());
} catch (NoSuchAlgorithmException e) {
    System.err.println("creation of Bobs DataUnsealer failed:" +
e.getStackTrace());
} catch (IOException e) {
    System.err.println("creation of Bobs DataUnsealer failed:" +
e.getStackTrace());
}
}
}

```

6.2.3 UnsealByUnknown console output

Note: depending on your Log4j settings, you can generate additional debugging output.

```

"from author:
CN=NIHII\=00000000101,OU=NIHII\=00000000101,OU=Alice,OU=eHealth-
platform Belgium,O=Federal Government,C=BE
unsealed data for unknown recipient: This is a secret message from Alice
for an unknown addressee."

```



6.3 Error and failure messages

There are additional error messages that may occur when trying to encrypt or decrypt information for unknown recipients. A symmetrical key is referenced as a 'secret key' in the error messages (the key for unknown recipients is secret, whereas the terminology 'public' and 'private' keys is used for known recipients).

The given decryption key can not decrypt the KEKRecipientInformations in the EnvelopedData

- *SECRET_KEY_CAN_NOT_DECRYPT_KEKRECIPIENTINFOS,*

The EnvelopedData do not contain a KEKRecipientInformation

- *ENVELOPEDDATA_CONTAINS_NO_KEKRECIPIENTINFOS*



7 Implementation aspects

7.1 Development and test procedures

Once a KGSS client has been developed that can connect to our web service in the acceptance environment, integration and acceptance tests can begin. We ask to perform tests for at least one month. The reason behind this is to protect the production environment to the fullest against any incidents.

When developing an application that will use the ETEE infrastructure, you must notify eHealth by a written document at least 3 months before the scheduled production date. This is required for efficient capacity planning and will allow the eHealth platform to assure the eHealth SLAs. Please contact the contact center for more information.

If everything is correct, eHealth and the partner agree on a release date. eHealth should prepare the connection to the production environment and provide the production environment URL.

During the release day in acceptance, the partner in the health care sector provides feedback on the release test results to the designated eHealth contact(s). You will be provided with a list of designated contact(s) for your project.

A contract must be signed by which you confirm as a partner that the integrator will also observe the eHealth rules in case of new releases of his software.

7.2 Request for release in production

If the acceptance tests are successful, the partner in the health care sector sends the test results and test performance results, as well as samples of 'request' and 'eHealth answer' to their eHealth designated contact(s) by email.

The following items must be checked before you can access the production environment:

1. request New keys from the KGSS (GetNewKey);
2. get existing Keys from the KGSS (GetKey).

7.3 Maintenance, support and monitoring of the service

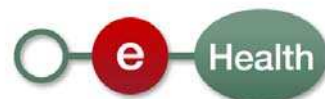
Before deploying an application or using the web service in production, the partner in the health care sector who is using the web service for one of its applications should always first run tests in the acceptance environment and then release any adaptation to his own application. In addition, he should inform eHealth on the changes and their test period.

When developing an additional use case, based on an existing integration, eHealth must be informed at least one month in advance and be provided with a detailed estimate of the expected load. This will ensure an effective capacity management.

In case of technical issues on the web service, the technician of the partner in the health care sector may get support from eHealth.

If eHealth finds a bug or vulnerability in its software, the partner has to update the application with the latest version of the software within 10 business days after a notification has been sent by the eHealth newsletter.

If the partner finds a bug or vulnerability in the software delivered by eHealth, he must immediately contact and inform eHealth and he is not allowed to publish this bug or vulnerability by any means.



8 Errors and solutions

8.1 EteeResponseType

The EteeResponseType has status codes. These status codes can be:

- 200 The KGSSRequest was correctly processed.
- 400 The KGSSRequest SOAP message is incorrect.
- 500 The KGSSRequest could not be completed due to an internal server error.

8.2 Invalid Key Size

Error messages referring to an 'illegal key size' commonly indicate that there is a problem with this prerequisite: it means that the maximum allowed key size is set too small to allow the eHealth Crypto Library to work. Please consult the prerequisites section "3.1.2 Java Cryptography Extension (JCE)" on page 9.



9 Risks

Your PC needs to be properly secured. The computer has a quality and up-to-date antivirus software and a network firewall.



10 Contact

The eHealth platform help desk is available for any questions or problems you may have. You can use a contact form that is available on the eHealth platform portal or contact the eHealth platform by phone (+32 2 788 51 55).

Contact form D <https://www.ehealth.fgov.be/nl/contactform/website/home/contactform.html>

Contact form FR <https://www.ehealth.fgov.be/fr/contactform/website/home/contactform.html>

- For users in production, please contact:
support@ehealth.fgov.be
- For users in acceptance, please contact info@ehealth.fgov.be.



11 Licenses

In order to respect the licence agreement of third party software providers, the eHealth platform is requested to publish the following information:

11.1 Apache

Copyright © 2009 - 2010 eHealth-platform

Licensed under the Apache License, Version 2.0 (the "License");
You may not use this file except in compliance with the License.
You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

11.2 Bouncy Castle

Copyright © 2000 - 2009 The Legion Of The Bouncy Castle (<http://www.Bouncycastle.org>)

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.



12 Annex

Communication regarding the Java End-To-End Encryption Library and the .Net Library for End-To-End Encryption

The eHealth platform offers its users access to the Java End-To-End Encryption Library (Java ETEE Library). This library, which is distributed under a free license, is available on the eHealth website at the following address: <http://www.ehealth.fgov.be>.

Certain elements of the Java ETEE library of the eHealth platform use external components that are distributed under the Apache license and under the license distributed with the software « The Legion of The Bouncy Castle ».

In addition to the rules specified in the licenses mentioned above, the user must also take into account the following independent and additional stipulations regarding the guarantee and liability of the managers, administrators, employees and staff members of the eHealth platform.

When adapting a free software package, the eHealth platform makes every effort in order for the software to function correctly, nevertheless without assuming any obligation of result with respect to this matter.

The user commits himself to use the Java ETEE library that is available to him in the most correct and adequate way possible and to provide the eHealth platform, if necessary, with all the necessary information in order to solve problems concerning the use of the library.

Since the use of the Java ETEE library is free, the eHealth platform can on no account be held responsible for any kind of damage, direct or indirect, secondary or accessory, material or moral, caused to the user or to any third party, as a result of the use or the impossibility to use the library.

The Java ETEE Library must not be confused with the .Net Library for End-To-End Encryption which was developed by Siemens on behalf of Microsoft.

The .NET ETEE Library, an adaptation of the eHealth .Net Java ETEE Library, is available on the website <http://etee.codeplex.com/>. The library is distributed in compliance with the terms of the GNU Lesser General Public License. It is free and available to anyone who wishes to use it. The documentation available in the library was written and published by Siemens. Users who want more guarantees can conclude a contract with Siemens or with any other service provider. In accordance with the conditions contained in this contract, the users of the .NET ETEE Library will only have access to the technical support offered by the concerned service provider.

The eHealth platform does not offer any technical support with regard to the .NET Libraries.

The eHealth platform can therefore in no event be held responsible for any damage, direct or indirect, secondary or accessory, material or moral, caused to the user or to any third party, as a result of the use or the impossibility to use the library.

Questions or remarks about the .NET ETEE Library can be posted on this website <http://etee.codeplex.com/>.

