icw
eHEALTH FRAMEWORK

eHealth Framework

How to configure a Module for Encryption

Imprint

InterComponentWare
Altrottstraße 31
69190 Walldorf
Tel.: +49 (0) 6227 385 0
Fax.: +49 (0) 6227 385 199

Document version: preliminary
Document Language: en (US)
Product Name: eHealth Framework
Product Version: 2.10.3
Last Change: 23.03.2010
Editorial Staff: BAS Technology

Notice

The wording in this document applies equally to women and men. The masculine form was selected to ease the comprehensibility and legibility of the text.

All company logos are a registered trademark of InterComponentWare AG.

The product names mentioned in this documentation are either trademarks or registered trademarks of the respective owners and are stated for identification purposes only.

Note:

The current version of this document has a draft status and various chapters are still in review.

The document is collaboratively built with the use of the Darwin-Information-Typing-Architecture (DITA) and has therefore a draft status concerning styles and layout. The necessary adaptations are currently also in a developmental stage.

Contents

1 Overview

The eHealth Framework (eHF) provides an infrastructure for pseudonymization and encryption of sensitive health data at application level. Sensitive data are encrypted in the application before going to the database. This document describes in detail the steps in your module to enable pseudonymization and encryption.

2 Tasks

The subsequent tasks guide you through the process for configuring a module for encryption and pseudonymization.

2.1 Activating Encryption

The activation of encryption requires additional dependencies and generator configuration. Once encryption is activated, unit testing can also be affected. The following steps detail the required modifications.

1. Include the following dependencies in your `project.xml` (versions of artifacts may differ), if you use maven 1:

```
<dependency>
    <groupId>ehf</groupId>
    <artifactId>ehf-commons-encryption-api</artifactId>
    <version>2.10</version>
    <type>jar</type>
</dependency>

<dependency>
    <groupId>ehf</groupId>
    <artifactId>ehf-commons-encryption-runtime</artifactId>
    <version>2.10</version>
    <type>jar</type>
</dependency>

<dependency>
    <groupId>ehf</groupId>
    <artifactId>ehf-commons-encryption-test</artifactId>
    <version>2.10</version>
    <type>jar</type>
</dependency>
```

   Include the following dependencies in your `pom.xml` (versions of artifacts may differ), if you use maven 2:

```
<dependency>
    <groupId>com.icw.ehf</groupId>
    <artifactId>ehf-commons-encryption</artifactId>
    <version>2.10</version>
    <scope>ehf[api,runtime,test]</scope>
</dependency>
```

2. Enable the eHF Generator property that controls the generation of encryption logics in domain objects and DAOs by including the following property into the `module.configuraton.properties` file:

```
ehf.generator.encryption.enabled=true
```

3. Provide a test Spring context `src/test/resources/META-INF/ehf-commons-encryption/ehf-cryptoservice-context.xml` with a `CryptoServiceMock` bean as shown in the following:

```
<bean id="cryptoService"
   class="com.icw.ehf.commons.encryption.mock.CryptoServiceMock"
   init-method="initializeContext" destroy-method="destroyContext">

  <property name="idBasedAttributes">
    <list>
    <!-- add id-based encrypted attributes here, for instance,
      <value>com.icw.ehf.medicine.cabinet.domain.IntakeEntry_createDate
```

```
      </value>
    -->
    </list>
  </property>

  <property name="scopeBasedAttributes">
    <list>
    <!-- add scope-based encrypted attributes here, for instance,
     <value>com.icw.ehf.medicine.cabinet.domain.IntakeEntry_comment
     </value>
    -->
    </list>
</property>
</bean>
```

A scope-based encryption takes the scope value of the associated domain object into account to select the cryptographic key and further parameters, while an id-based encryption takes the identifier of the particular instance of the domain object to select the key and parameters.

4. Run a `maven clean dev:build` with maven 1 or `mvn -U clean install` with maven 2 in your project.

   By default, technical attributes `scope`, `creatorId`, `changerId`, `createDate` and `changeDate` in each domain object are pseudonymization-relevant and need to be encrypted. They are referred to as *pseudonymization defaults*. If the build process succeeded, you will find that the eHF Generator has generated public static `createAttributeTargetFor*()` methods for each of the above attributes in the associated domain objects. * stands for the name of the attribute with the first alphabet in capital letters.

   If the build process failed, it is likely that you have queries on encrypted fields. Please refer to Handling Custom Queries on Encrypted Data on page 4 for more information. Please also consult the troubleshooting section for further possible reasons.

2.2 Enhancing the Domain Model

Starting with version 1.3, eHF Profile supports encryption-relevant tags in the domain model. Please check the UML profile version you are using for your domain model. If an older version is used, please refer to appendix B.5 of the eHF Reference Documentation for upgrading to eHF Profile Version 1.3. With the new profile version enabled, you get access to the `classification` tag on attribute level and the `pseudonymized` tag on domain object level.

Note: We recommend that a first analysis of your model is conducted with the ICW TIS team. Depending on the model various modifications may be required to get started. Furthermore a classification strategy needs to be determined.

1. Decide for each domain object, whether its technical attributes which are pseudonymization defaults should be encrypted, and set the `pseudonymized` tag accordingly.

By default, the `pseudonymized` tag of each domain object is set to true. If you want to exclude the technical attributes of a domain object from encryption, please explicitly set the `pseudonymized` tag of the domain object to false.

2. Annotate the `classification` tag for each encryption-relevant attribute.

   eHF supports selective encryption, i.e. only classified attributes are encrypted. In general, applications are free to define their own classifications. However, a consistent convention or strategy should be applied.

2.3 Handling Custom Queries on Encrypted Data

Encryption affects queries. Some attributes are encrypted based on the scope of the associated domain object. Therefore, it is necessary to supply the scope value to the query. Queries on attributes which are id-based encrypted without knowing the particular identifiers are no more possible.

Encryption or decryption in create, update and load operations (CRUD operations) are transparent for application developers, since the eHF Generator generates application code to trigger the necessary encryption and decryption.

For custom queries on encrypted attributes, the application developer needs to add encryption functions to encrypt the query parameters and subsequently decryption functions to decrypt the query results. Custom queries are the only situation where encryption is not transparent for application developers.

For each classified attribute in the domain model, the eHF Generator generates a static `modulate*ForQuery()` and `revert*()` signature in the associated domain object class for encrypting and decrypting the attribute respectively. * stands for the name of the attribute with the first alphabet in capital letters.

With the following example, we illustrate the required steps to handle custom queries on encrypted data:

```
public Object doInHibernate(Session session)
    throws HibernateException, SQLException {

    Criteria criteria = session.createCriteria(IntakeEntry.class);
    criteria = criteria.add(Restrictions.eq("comment", comment));
    criteria = criteria.add(Restrictions.eq("scope", scope));

    return criteria.list();
}
```

1. Encrypt the query parameters with the `modulate*ForQuery()` methods provided in the domain object classes.
   Applying this modification to the above example the result would look as follows:

   ```
   public Object doInHibernate(Session session)
       throws HibernateException, SQLException {

       Criteria criteria = session.createCriteria(IntakeEntry.class);

       criteria = criteria.add(Restrictions.eq("comment",
                  IntakeEntry.modulateCommentForQuery(comment, scope)));

       criteria = criteria.add(Restrictions.eq("scope",
                  IntakeEntry.modulateScopeForQuery(scope)));

       return criteria.list();
   ```

```
}
```

2. Decrypt the query results with the `revert*()` methods provided in the domain object classes.

   This is just a simple example. Real life with object hierarchies, HQL, projections or plain JDBC access is more complex. However the basic principle of modulating (i.e. encrypting) the query parameters and reverting (i.e. decrypting) the query results is always the same. Apart from the `modulate*ForQuery()` and `revert*()` methods, the module-specific `ModuleAttributeModulation` class with its signatures for modulation and reversion can also be used.

2.4 Contributing Target-Mapping to the Assembly

With annotations on encryption-relevant attributes in the domain model you have specified *what* to encrypt, with the target mapping defined in the following you specify *how* to encrypt.

The target mapping file is a bridge between an annotated domain model and the encryption infrastructure. By defining the mapping between a classification to cryptographic parameters (e.g. key scope, key length, cryptographic algorithm and engine), it specifies how attributes of this classification are to be encrypted.

1. Define a target mapping in a `target-mapping.fragment` fragment located in the `src/main/config/merge/assembly` folder of your project. Fragments from modules will be merged to a single target mapping file on assembly level.

   The following shows an example for the module `com.icw.ehf.medicine.cabinet`.

```
<module module-id="com.icw.ehf.medicine.cabinet">
    <config-element classification="scope">
        <parameters key-pool="pool_scope"/>
    </config-element>
    <config-element classification="c-amount">
        <parameters key-type="scope" iv-type="scope"
            iv-generator="native">
        </parameters>
    </config-element>
    <config-element classification="freetext">
        <parameters key-pool="pool_freetext" key-type="scope"
            iv-type="scope"/>
    </config-element>
    <config-element classification="c-date">
        <parameters key-pool="pool_c-date"/>
    </config-element>
    <config-element classification="medical">
        <parameters key-type="id" key-pool="pool_medical"/>
    </config-element>
    <config-element attribute="timeCanonicDate">
        <parameters key-pool="pool_c-date_OPE"/>
    </config-element>
    <key-pool id="pool_freetext" minKeys="2" engine-id="JCE"
        provider="SunJCE" key-length="192"/>
    <key-pool id="pool_c-date" engine-id="JCE"   key-length="128"/>
    <key-pool id="pool_c-date_OPE" engine-id="LocalOPE"/>
    <key-pool id="pool_medical" minKeys="10"/>
    <key-pool id="pool_scope" engine-id="JCE"   key-length="256"/>
</module>
```

   The main elements of a target mapping are `config-element`, `parameters` and `key-pool`.

   config-element

   A `config-element` element has the attributes `classification`, `owner-class` and `attribute`. The attributes can be combined, for example,

```
<config-element
  owner-class="com.icw.ehf.medicine.cabinet.Medicine"
  classification="scope">.
```

At runtime, attributes of a `config-element` are compared with properties of a domain attribute to determine, whether this `config-element` applies to it. If no `config-element` applies to a particular domain attribute, this attribute is not encryption-relevant. A `config-element` consists of a `parameters` element.

parameters

A `parameters` element refers to a key pool and specifies the key scope, iv scope and iv generator in the `key-type`, `iv-type` and `iv-generator` attributes respectively.

- The `key-type` attribute specifies the scope of a cryptographic key. If key-type is `scope`, a scope-based key is used. If key-type is `id`, an instance-based key is used, i.e. the identifier of the instance is taken into account to select the cryptographic key. If no key-type is explicitly specified, the same key is used to encrypt all domain attributes defined by the enclosing `config-element`.
- The `iv-type` attribute specifies the scope of an initialization vector. Like the `key-type` attribute, it is optional and the allowed values are `scope` and `id`.
- The `iv-generator` attribute specifies the generator to use for creating an initialization vector. If unspecified or specified as `iv-generator="native"`, the default iv generator of the system is used. In case of using a user-defined iv generator, the fully qualified class name of the generator must be provided.

key-pool

A `key-pool` element specifies a set of keys sharing the same characteristics such as the cryptographic algorithm, the mode, the padding scheme, and key size. Keys in different key pools are disjunct. Hence, using different key pools ensures using different keys. The attributes of a `key-pool` element are explained in the following:

- The `id` attribute specifies the identifier of the key pool. It is referenced by one or multiple `parameters` elements.
- The `algorithm`, `mode` and `padding` attributes specify the cryptographic algorithm with specific mode and padding. For performance considerations, only symmetric block ciphers are supported in eHF. If you want to use a Null Cipher (which returns the same plaintext for encryption and ciphertext for decryption) for testing purposes, specify `algorithm="null"`(case insensitive).
- The `engine-id` attribute specifies the cryptographic engine to use. Currently, allowed values are `JCE` and `LocalOPE`. `JCE` stands for Java Cryptography Extension. `LocalOPE` is our own engine which supports order-preserving encryption of numeric values.
- The provider architecture of JCE allows for using different providers which are specified using the `provider` attribute. If unspecified, the default provider `SunJCE` is used.
- The `key-length` specifies the key size. If unspecified, the default key size offered by the provider (for example, 128 bit from SunJCE) is used. To use key size larger than 128 bit, please consult Compliance with Meaningful Use on page 12.
- The `minKeys` attribute specifies the minimum number of keys in this particular key pool.

Note: In contrast to the JCE engine, the order preserving encryption engine (LocalOPE) is not configurable concerning its provider and key size. If specified, these properties are ignored by the encryption infrastructure.

As mentioned above, module-specific target mapping fragments are assembled to a single target mapping file on assembly level. The following shows the target mapping file in assembly with placeholders for the fragments.

```
<?xml version="1.0" encoding="UTF-8"?>
<target-mapping xmlns="http://www.intercomponentware.com/schema/ehf-
encryption">

    @@@target-mapping.fragment@@@

    <template>
        <config-element classification="scope">
            <parameters key-pool="global_pool_scope"/>
        </config-element>
        <config-element classification="identifier">
            <parameters iv-type="scope" key-pool="global_pool_identifier"/>
        </config-element>
        <config-element classification="date">
            <parameters key-pool="global_pool_date" />
        </config-element>
        <config-element classification="confidential">
            <parameters key-pool="global_pool_confidential"/>
        </config-element>
        <config-element classification="strictly-confidential">
            <parameters key-pool="global_pool_strictly_confidential" />
        </config-element>
        <config-element classification="freetext">
            <parameters key-pool="global_pool_freetext"/>
        </config-element>
        <key-pool id="global_pool_date" engine-id="LocalOPE" />
        <key-pool id="global_pool_identifier" minKeys="5"/>
        <key-pool id="global_pool_scope" minKeys="5"/>
        <key-pool id="global_pool_freetext" minKeys="5"/>
        <key-pool id="global_pool_confidential" minKeys="5"/>
        <key-pool id="global_pool_strictly_confidential" minKeys="10"/>
    </template>

    <properties>
        <property name="algorithm" value="AES"/>
        <property name="padding" value="PKCS5Padding"/>
        <property name="mode" value="CBC"/>
        <property name="engine-id" value="JCE"/>
        <property name="minKeys" value="1"/>
    </properties>

</target-mapping>
```

Apart from module-specific parts, a target mapping file optionally consists of a `properities` and a `template` element.

template

The `template` element provides module-unspecific, global configurations. It defines a set of global key pools for the preserved classifications `scope`, `identifier`, `date`, `freetext`, `confidential` and `strictly-`

`confidential`. It allows for using the same cryptographic parameters for attributes from different modules.

properties

The `properties` element specifies default properties for all key pools. Each key pool can override these property values. Currently following properties are supported: `algorithm`, `mode`, `padding`, `engine-id`, `provider`, `key-length` and `minKeys`.

Note: If not explicitly specified, the encryption infrastructure uses `AES/CBC/PKCS5Padding` offered by the `JCE` from the `SunJCE` provider by default.

3 Conventions for Using Classifications

Encryption-relevant classifications in the domain model are mapped to cryptographic parameters in the target mapping file. A mapping strategy should be defined in order to decide which classification to assign to a particular attribute in a consistent way.

For eHF application modules, we have defined a set of classifications with the following recommended usage. You are free to introduce new classifications for your application.

Classification	Recommended Usage
identifier	Use scope-based cryptographic parameters, either a scope-based key (i.e. `key-type="scope", minKeys="10"`) or a scope-based initialization vector (i.e. `iv-type="scope"`). The minimum available number of keys specified by the `minKeys` attribute are shared by domain objects. Thus, take the real number of scopes into account when setting this property. Using scope-based initialization vectors (IVs) is preferred due to its lower space demand. In case the owning domain object does not have a `scope` attribute, use a unique key for each module.
scope	Use a unique key for each module.
date	The configuration depends on whether you want to let the database sort dates. If yes, use the order preserving encryption algorithm (i.e. `engine-id="LocalOPE"`). Make sure, exactly one key and one IV is used (i.e. no specification for `key-type` and `iv-type`).
freetext	Should be used for information which is sensitive. Use a unique key for each module.
confidential	Should be used for information which is highly sensitive. Use at lease scope-based IVs (i.e. `iv-type="scope"`). In case the owning domain object does not have a `scope` attribute, use a unique key for each module.
strictly-confidential	Should be used for information which is highly sensitive and could be stigmatizing for the affected person or his/her relatives (e.g. diagnosis, cause of death). Use scope-based keys (i.e. `key-type="scope", minKeys="10"` (the IVs are then automatically scope-based). Alternatively, add instance-based IVs (`key-type = "scope", iv-type="id"`). In case the owning domain object does not have a `scope` attribute, either use a unique key for each module, or use instance-based keys (i.e. `key-type="id", minKeys="100"`).

Table1. Recommended mapping between classifications and cryptographic parameters

The eHF Record Medical Module applies the above convention. Its target mapping file shown in the following serves as an example.

```
<module module-id="com.icw.ehf.record.medical">
    <config-element classification="identifier">
        <parameters key-pool="pool_identifier" iv-type="scope"/>
    </config-element>
    <config-element classification="scope">
        <parameters key-pool="pool_scope"/>
    </config-element>
    <config-element classification="freetext">
        <parameters key-pool="pool_freetext"/>
```

```
    </config-element>
     <config-element classification="date">
        <parameters key-pool="pool_date"/>
    </config-element>
    <config-element classification="confidential">
        <parameters key-pool="pool_confidential" iv-type="scope"/>
    </config-element>
    <config-element classification="strictly-confidential">
        <parameters key-pool="pool_strictly_confidential" key-type="scope"/>
    </config-element>

    <key-pool id="pool_identifier"/>
    <key-pool id="pool_scope"/>
    <key-pool id="pool_freetext"/>
    <key-pool id="pool_confidential"/>
    <key-pool id="pool_strictly_confidential" minKeys="10"
         engine-id="SunJCE" algorithm="AES"/>
    <key-pool id="pool_date" engine-id="LocalOPE"/>
</module>
```

4 Troubleshooting

This section provides useful information for solving possible issues.

No createAttributeTargetFor*() method generated

The eHF Generator generates static createAttributeTargetFor*() methods for each encrypted attribute in the associated domain object class. If you cannot find these methods, please make sure that the property `ehf.generator.encrypted.enabled` is set to true in your `module.configuration.properties` file. Delete the `src/main/oaw/generated.jar` file, since the generator skips the generation if a `generated.jar` is already available. Run `maven dev:build` or `mvn -U clean install` again.

Encrypt and decrypt BLOBs and CLOBs as streams

eHF supports the encryption and decryption of BLOBs (Binary Large Objects) and CLOBs (Character Large Objects). However, due to their size, BLOBs and CLOBs may not fit into the memory. Hence, they are encrypted or decrypted as streams. Please contact the ICW TIS team to get support for handling BLOBs and CLOBs.

Order-preserving encryption for dates

Using an order-preserving encryption (OPE) scheme, the order in the plaintexts is preserved in the ciphertexts after an encryption. An OPE allows sorting and range queries on encrypted data without having to decrypt them at first. In eHF, OPE is supported only for numeric data.

A typical candidate to use OPE is the `com.icw.ehf.core.domain.Date` domain class. However, a `Date` object consists of several attributes (mapped to different columns in the database), for example, `isoDate` and `canonicDate`. The `isoDate` attribute is a String representation of Date which is compliant to ISO 8601. The `canonicDate` attribute is of type long. Hence, OPE is supported for `canonicDate`, but not for `isoDate`. If you annotate an attribute of type `Date` in your domain model with "date", and you want to use OPE for this attribute, you must explicitly specify that only `canonicDate` shall be encrypted with OPE, while `isoDate` shall be encrypted with an ordinary block cipher such as AES. The following shows the required configuration in a target mapping file.

```
<config-element classification="date">
        <parameters key-pool="pool_date_OPE"/>
    </config-element>

    <config-element classification="date" attribute="isoDate">
        <parameters key-pool="pool_date"/>
    </config-element>

    <key-pool id="pool_date" engine-id="JCE" algorithm="AES"/>
    <key-pool id="pool_date_OPE" engine-id="LocalOPE" />
```

5 Compliance with Meaningful Use

The Department of Health and Human Services (HSS) has issued an interim final rule to adopt standards, implementation specifications, and certification criteria to enhance the interoperability, functionality, utility, and security of health information technology and to support its meaningful use.

Relation to the interim final rule

With respect to the objective "protect electronic health information created or maintained by the certified EHR technology through the implementation of appropriate technical capabilities", the certification criteria requires

"4. Encrypt and decrypt electronic health information according to user-defined preferences (e.g., backups, removable media, at log-on/off) in accordance with the standard specified in Table 2B row 1".

Table 2B row 1 defines for the purpose "General Encryption and Decryption of Electronic Health Information" the following adopted standard:

"A symmetric 128 bit fixed-block cipher algorithm capable of using a 128, 192, or 256 bit encryption key must be used (e.g., FIPS 197 Advanced Encryption Standard, (AES), Nov 2001)."

Compliance of eHF

The eHF encryption infrastructure supports application-level encryption. Sensitive electronic health information is already encrypted in the application, before being transferred to and stored in the database. Hence, sensitive health information is also protected by encryption in backups or archives.

Using the eHF encryption infrastructure, electronic health information can be encrypted and decrypted in compliance with the adopted standard specified in Table 2B row 1. The infrastructure is configurable with respect to the symmetric block cipher as well as the key size used. By default, the 128 bit block cipher AES using a 128 bit encryption key provided by SunJCE is utilized.

Cryptographic algorithms, key sizes and the cryptographic JCE providers are configured in a target mapping file. The following describes how to configure JCE providers as well as key sizes.

Configuring JCE providers

1. Add the Jar file of your preferred JCE provider to the classpath.
2. Register the provider in the Java master security properties file `java.security` which is located in your JRE in the `lib/security` folder with the following line:

   ```
   security.provider.<n>=<className>
   ```

   This declares a provider, and specifies its preference order n. The preference order is the order in which providers are searched for requested algorithms (when no specific provider is requested for that algorithm). The order 1 is the most preferred, followed by 2, and so on.

   `className` specifies your provider class whose constructor sets the values of various properties that are required for the Java Security API to look up the algorithms or other facilities implemented by the provider.

   For instance, the following line registers the bouncycastle provider with preference order 10:

```
security.provider.10=org.bouncycastle.jce.provider.BouncyCastleProvider
```

3. Specify the provider with the `provider` attribute of a `key-pool` element in your target mapping file. The following line specifies the bouncycastle provider with its name "BC" for keys in the key pool `pool_scope`.

```
<key-pool id="pool_scope" engine-id="JCE" provider="BC"
    key-length="192"/>
```

Configuring key size

1. Specify the key size with the `key-length` attribute of a `key-pool` element in the target mapping file. The following example specifies a key of length 256.

```
<key-pool id="pool_scope" engine-id="JCE" provider="SunJCE"
    key-length="256"/>
```

2. Java Cryptography Extension Unlimited Strength Jurisdiction Policy is required to use key sizes larger than 128 (e.g. 192 or 256).

   Download JCE Unlimited Strength Jurisdiction Policy (`US_export_policy.jar`) from Sun Developer Network (http://java.sun.com/javase/downloads/index.jsp). Unpack the content of this Jar file (i.e. `local_policy.jar` and `US_export_policy.jar`) to the `lib/security` folder of your JRE to override the existing policy files. It is recommended to make a copy of the original policy files for later usage.