

Encryption Design

Glenn Leyba

1/19/2011

**Confidential**

This document should only be distributed to those within CMSI on a need-to-know basis. This document should not be distributed outside the CMSI organization.

Contents

[Contents 1](#_Toc346618983)

[Overview and Design Approach 3](#_Toc346618984)

[Setting Up the Client Environment 3](#_Toc346618985)

[Creating Users and Keys in the Appliance 4](#_Toc346618986)

[Encrypting and Decrypting 5](#_Toc346618987)

[AES-128 vs AES-256 bit keys 5](#_Toc346618988)

[Key Versioning and Initialization Vectors (IVs) 6](#_Toc346618989)

[Tokenization (Masking) 7](#_Toc346618990)

[Searching using a HMAC Hash 7](#_Toc346618991)

[Protecting the User Name /Password for Appliance Access 8](#_Toc346618992)

[Database Changes for Storing Encrypted Values 8](#_Toc346618993)

[Determining what size to specify for an Encrypted column. 9](#_Toc346618994)

[Determining what size to specify for a Hash column 9](#_Toc346618995)

[Defining Encryption Columns on a Table by Table Basis 9](#_Toc346618996)

[Key and User Management 10](#_Toc346618997)

[Encrypting/Decrypting Files 10](#_Toc346618998)

[Managing Users and Keys in the Appliance 11](#_Toc346618999)

[Users 11](#_Toc346619000)

[Groups 12](#_Toc346619001)

[Keyowners 12](#_Toc346619002)

[Keys 13](#_Toc346619003)

[Defining Keys for Field Level Encryption 13](#_Toc346619004)

[Defining Keys for File Encryption 14](#_Toc346619005)

[Defining Keys for CLOB Encryption 14](#_Toc346619006)

[Defining Keys for Generating Hash Values 15](#_Toc346619007)

[Turning Encryption On and Off 15](#_Toc346619008)

[Origenate 15](#_Toc346619009)

[Note: The column names must match the generic keyname fields assigned in the section: Defining Keys for Field Level Encryption 16](#_Toc346619010)

[Gateway 16](#_Toc346619011)

[Evaluate 17](#_Toc346619012)

[Implementation 17](#_Toc346619013)

[Setting up the Client Side Environment 17](#_Toc346619014)

[Crypto.jar 17](#_Toc346619015)

[Classpath Changes 17](#_Toc346619016)

[Policy Files to Support AES-256 bit keys 17](#_Toc346619017)

[Client side config files, crypto.ini and IngrianNAE.properties files 18](#_Toc346619018)

[Securing Passwords on the Client Side 19](#_Toc346619019)

[Origenate.ini File Changes 20](#_Toc346619020)

[Client Side Coding and Usage 21](#_Toc346619021)

[Java Examples 21](#_Toc346619022)

[Encrypting a Clob Value 22](#_Toc346619023)

[Decrypting a Clob Value 23](#_Toc346619024)

[A Full List of Methods in crypto.jar 24](#_Toc346619025)

[ColdFusion Example 24](#_Toc346619026)

[Decrypting a clob value under CF 25](#_Toc346619027)

[Test Driver Program 26](#_Toc346619028)

[Encrypting Values in an Existing Table before Going Live 26](#_Toc346619029)

[Converting CLOB columns in an Existing Table before Going Live 26](#_Toc346619030)

[XMLDBT Changes 27](#_Toc346619031)

[Adding new xmldbt config rows to support encryption for a column 28](#_Toc346619032)

[Creating the ROWID column definition 28](#_Toc346619033)

[How these new column definitions are used 29](#_Toc346619034)

[Inbound XML 29](#_Toc346619035)

[Outbound XML 29](#_Toc346619036)

[Summary for XMLDBT field changes 29](#_Toc346619037)

[Calling GenX to generate XML 29](#_Toc346619038)

[Checklist for adding an encrypted field 30](#_Toc346619039)

[Gateway Changes to Support Encryption 31](#_Toc346619040)

[Java 1.6 Version of COLT 31](#_Toc346619041)

[Turning Encryption on in COLT 31](#_Toc346619042)

[Decrypting Encrypted Files 32](#_Toc346619043)

[COLT Environment Changes 32](#_Toc346619044)

[Encrypting Applications by Request ID 32](#_Toc346619045)

# Overview and Design Approach

The purpose of this document is to describe how database encryption can be accomplished with a Hardware Security Module (HSM or Appliance) and highlight some of the design decisions that we need to make to implement database encryption within Origenate. Actually it’s not database encryption but rather encryption of specific values and or files. This is a living document. As we make design decisions they will be reflected in subsequent versions of this document.

Our goal is to encrypt specific values in our database like ssn so that our database does not contain the actual ssn value (open to attack). To do this we are using SafeNet’s appliance to do the encryption for us. The appliance or Network-Attached-Encryption (NAE) server does not actually store encrypted values. Its purpose is to manage the keys that are used to perform crypto operations. Without the keys, an attacker can not decrypt our encrypted data.

## Setting Up the Client Environment

If we want to encrypt a value like ssn we communicate with the appliance from a client application via the JAVA JCE API’s. JCE stands for Java Cryptography Extension.

Before we can use the JCE from a client java app we need to install the Ingrian JCE Provider jar files. These are provided by SafeNet and consist of a set of jar files that need to be installed in the JRE/lib/ext dir in the client’s environment. See the JCE Users Guide, section: Obtaining Provider Software for instructions on how to get these jar files. The files that I installed in the ext dir to get the samples to compile were:

IngrianLog4j.jar

IngrianNAE-5.0.0.jar

Cryptix-jce-api.jar

IngrianNAE.properties

Note: The jar files do not have to reside in the ext dir if they are specified on the classpath of the program that is accessing them. Hence, they can live in our lib dirs.

Note2: I have checked these jars into VSS in the lib dir so you don’t need to download them.

In addition, an IngrianNAE.properties file also has to be installed in the ext dir. This file contains properties of how to communicate with the appliance. Ie) IP address, port, protocol, etc. In our properties file the IP of our test appliance and port is:

NAE\_IP.1=172.16.24.52

NAE\_Port=9000

Note: The location of the properties file can be specified at runtime by setting a java System property before accessing the server. The following is from the JCE Users Guide.

By default, the JCE Provider will look for the location of the IngrianNAE.properties file in the

same location as the IngrianNAE.jar. If you move the properties file, you must set the

com.ingrian.security.nae.IngrianNAE\_Properties\_Conf\_Filename property to the new location.

For example,

System.setProperty("com.ingrian.security.nae.IngrianNAE\_Properties\_Conf\_

Filename", "home/java/IngrianNAE.properties");

Note: You must include the filename when setting the property.

Note: This capability has now been encapsulated in the crypto.jar file, see later discussion on crypto.jar.

## Creating Users and Keys in the Appliance

Once our client environment is set up we are ready to call the NAE to encrypt/decrypt values. Before we can do this we need to do some one-time setup on the appliance. To setup the appliance you log into it via a browser as an admin user. <https://172.16.24.52:9443/> is our IP of our test appliance. User ID and pwd provided by Sys Eng. The Management Console in the appliance allows you to create users, groups, and manage keys.

1. We first need to create a user name and pwd. This is the user that we will establish a session with the appliance from the client. *Since the user name and pwd have to be supplied by the client for access we will talk about securing these values at the client later in this document.* As an example I created a user called testuser. We then need to create a group. I created one called ‘origenate’. Once created you can then add users to this group. I added testuser to this group. Groups are associated with keys in the appliance. Users can not be assoc with a key in the appliance directly, only groups. Groups can have permissions assigned to them, explained later.
2. We now have to create a key that will allow us to encrypt/decrypt values. I created a key called testkey which was a versioned AES-128 bit key with a default IV. I will explain versioning, IVs and AES-128 vs AES-256 a little later.

## Encrypting and Decrypting

We are now ready to encrypt/decrypt values from our client. In our client java code we do the following:

Encrypting

1. Establish a session (connection) with the appliance using the user name and pwd.
2. Get a handle to the key we want to use to encrypt/decrypt using the keyname we created before. Ie. testkey (case sensitive)
3. Create a Cipher in encrypt mode using the key we obtained and an optional IV.
4. Encrypt the value to be encrypted using the cipher.
5. We can now store the encrypted value in our database. Since this is a binary value we need to base64 encode it before storing it.

Decrypting

1. Get the value to be decrypted from the database and decode it using base64 decoder.
2. Establish a session (connection) with the appliance using the user name and pwd.
3. Get a handle to the key we want to use to encrypt/decrypt using the keyname we created before. Ie. testkey (case sensitive)
4. Create a Cipher in decrypt mode using the key we obtained and an optional IV.
5. Decrypt the value using the cipher.

## AES-128 vs AES-256 bit keys

AES is more secure than DES which is why we will be using AES keys moving forward. When you create a key in the appliance you can specify the key strength, 128 or 256. 256 is considered more secure but it has some limitations. Using 256 bit keys is a little slower since it has to go thru more iterations during crypto operations. However, if we choose to use 256 bit keys we can not export the appliance out of the country because of export restrictions. There are certain countries on this list. SafeNet told me that Canada is not on this list but we need to make sure. This might affect RBC. Also SafeNet said that 128 bit keys are perfectly acceptable. Keep in mind that these keys never leave the appliance so it should not matter which we use.

Note: It was decided that we will use 256 bit keys going forward.

256 bit keys also have a client side config impact. If we use 256 bit keys then we need to download special JCE policy files that we need to store in the jre/lib/security directory. This is necessary to use unlimited strength keys. The files need to come from either Sun or IBM depending on the environment, ie Jrun vs Websphere.

Notes from JCE doc on where to get policy files:

To use unlimited strength ciphers (e.g. 192- and 256-bit AES keys) *you must download the*

*encryption policy files* for your Java implementation.

For a Sun Java implementation, download the Java Cryptography Extension (JCE) Unlimited

Strength Jurisdiction Policy File from

http://java.sun.com/javase/downloads/index.jsp.

For an IBM Java implementation, download the JCE Policy Files from

http://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=jcesdk

For more information on the IBM download, see

http://www-128.ibm.com/developerworks/java/jdk/security/50/

Backup jre/lib/security/local\_policy.jar and US\_export\_policy.jar and then copy the new ones in from the download.

## Key Versioning and Initialization Vectors (IVs)

When creating keys you have a choice to make them versioned or not. When creating a non versioned key the key is assigned a name and it is permanently assigned to that key. When accessing this key from a client you provide this name. If, for some reason, you need to replace the key (explained in a minute) you will need to create a new key and assign it a different name. This may cause config issues at the client. Ie) keeping the name of a key associated with a column in the database in a config table so you can get the correct key. Not to mention maintaing key names assoc with encrypted values for values that were created before the key changed. Versioned keys allows you to assign a single key name that can be used at the client even though there may be multiple key (versions) associated with that keyname in the appliance.

Key versions came about because of the PCI requirement that for certain data like credit card numbers a key must be changed on a regular basis (monthly, yearly, etc). This is known as key rotation. After talking with SafeNet, credit card numbers are the only type of data that falls under this category. However, I think it is good to use versioned keys to protect ourselves from a key change requirement for another field like ssn.

Versioned keys have implications. When a value is encrypted with a versioned keyname the current active key is used to encrypt the value. The first three bytes of the encrypted value indicates what version of the key was used. When decrypting this value the appliance looks at the header to determine which version of the key was used to encrypt and hence decrypt it using the correct version. If we want to search for ssn, using our search screen, we can not simply encrypt the user entered ssn and compare it to encrypted values in the database because we do not know which version of the key was originally used to encrypt the original ssn that was stored. See ***Searching using a HMAC Hash*** for how to solve this problem. Creating a new version of a key is a manual process in the management console and is not done automatically based on time, ie monthly. So, for a field like ssn we would define it as a versioned key and stay with the initial version until it might need to be changed.

**IVs**

When a key is created in the appliance an Initialization Vector (IV) value is generated. This is a 16 byte value which is used when encrypting and decrypting using the key. The IV acts as a kind of seed when encrypting the initial block of the value to be encrypted. In the client side examples provided by SafeNet it was shown that when creating the Cipher object you provide the key as well as the IV that goes with the key. This would cause immediate problems for versioned keys, such that, we would have to store the encypted value in our database as well as the IV that was used to create the encrypted value since we would not know during decryption which version of the key was used. After talking with SafeNet it turns out that this is not necessary. Even though the examples show having to pass the IV on the create cipher call, there is an overloaded constructor which does not require the IV parameter. In this case the value being decrypted and the key is all that’s needed. The appliance will look at the header of the encrypted value to determine which version of the key was used to encrypt it and then use the IV assoc with that key in the appliance to decrypt it.

## Tokenization (Masking)

When we encrypt a value like ssn we will store it in a new column along side the original ssn column. The original ssn column value will be changed to display an obfucated value for subsequent display purposes. This is known as a token (or mask). SafeNet has a Tokenization Manager but it only creates number tokens starting with an initial value and increases by one for each new value (kind of like an Oracle sequence). However, we want the ssn, for example, to contain all asterisks except for the last four digits of the ssn. This acts much like your credit card number on a receipt that you get at the store. To do this, when a value is encrypted we will return the encrypted value as well as a matching masked-value that can be stored in the original column value. We will have to write this and it may need to be parameterized to indicate how many remaining digits should be preserved.

Note: See the Implementation Section for how this was implemented.

## Searching using a HMAC Hash

When searching the database for a particular ssn we can not simply encrypt the value to search on and search the database for a matching encrypted value. This is because the original ssn stored may have been encrypted with a prior version of the keyname being used. To avoid this problem we need to generate a unique hash value that is consistent across all versions of a key. The appliance can do this for us but it requires a separate step from encrypting the string. After a value has been encrypted we will need to make an additional call to generate an HMAC value based on the clear-text value that was encrypted. An HMAC value is like a MAC address for your computer. It is expressed as a base64 encoded byte string. This value will be passed back so it can be stored in the database along side the encrypted value. When a user wants to search on ssn we will convert the clear-text ssn to its corresponding HMAC value and then compare that against the HMAC column in the database.

To create an HMAC hash you need to create a separate key used just for creating HMAC values. To do this I created a new key in the appliance called ‘hmac\_key’. The chosen algrothrim should be HmacSHA1, not AES. HmacSHA1 is used for generating MACs. When calling the API to generate a HMAC value, we go through the same steps as with encryption, that is, provide user/pwd to establish a session, get an instance of the hmackey and then generate the hmac value using the clear-text value passed in (ssn).

Note: HMAC generation is a one-way operation. You can not provide a hmac value and get back its clear-text value.

Note2: There will be a single hmack key defined to be used across all environments and evaluators.

## Protecting the User Name /Password for Appliance Access

There are weak links in any security solution. For me, I think the weak link is on the client side.

1. The Ingrian properties file has the IP address of the appliance. This file can not be encrypted.
2. A clear-text user name and password have to be supplied to the SafeNet APIs to establish a session with the appliance.

I don’t think we can secure the properties file for the reasons mentioned above. But I do think we need to talk about how to manage the values in this file, ie rollout, etc

As for protecting the session user name/pwd this is the most critical issue. Since it has to be avail from the client we should probly store this as an encrypted value in an ini file or database config table. Note, we may have multiple users defined based on roles, see ***Key and User Management*** below.

To encrypt the pwd value SafeNet suggests the following options:

1. Obfuscate the username/password using a combination of bit wise operators (e.g., XOR) and encoding (base16/base64)

2. Encrypt the username/password using a local key in software

3. Encrypt the username/password with an DataSecure global key

I think option one fits well for us. I can create a simple java app that will encrypt an entered pwd and diplay its encrypted value in base64 so we can store it in the database. An attacker will not be able to XOR the value back unless they have the key byte value used to do the initial XOR operation. This will be embedded in the java code directly. I will also supply helper classes to decrypt an encrypted pwd.

Note: It has been decided that we will go with option one. See the implementation section for details.

## Database Changes for Storing Encrypted Values

Using SSN as an example:

1. Store masked version of ssn in original ssn column.
2. Create a new column, that has the same name with a different ending, to store the encrypted value
3. Create a hash column to store the hash value to search on. Note: This is not required if we will never be searching on this column.
4. When creating a hash column also create an index on that column.

The naming convention for the encrypted column and the hash column will be as follows, using the soc\_sec\_num\_txt column as an example.

Original column: SOC\_SEC\_NUM\_TXT

Encrypted column name: SOC\_SEC\_NUM\_TXT\_ENC

Hash column name: SOC\_SEC\_NUM\_TXT\_HASH

Hash column index name: SOC\_SEC\_NUM\_TXT\_HNDX

### Determining what size to specify for an Encrypted column.

To determine what size to make an encrypted column run the Test Driver program, described near the end of this document. Run the Driver program against a sample value to see what the length of the encrypted value is. Using ssn as an example:

Java Driver e origenate ssn 123456789

For nine digit ssn’s the resulting returned length is 28 characters. I would increase the size by a few just to make sure we can store all generated values. For SSN I am suggesting we use 32.

### Determining what size to specify for a Hash column

Using the same technique as above you would run the Driver to generate a hash for a sample SSN.

Java Driver gh origenate 123456789

Turns out it also returns a 28 byte string. So I am suggesting we use 32 as well.

### Defining Encryption Columns on a Table by Table Basis

Using SSN as an example, most tables in Origenate use a column name of soc\_sec\_num\_txt. Unfortunately this convention has not been used for all tables. For each table in Origenate that contains ssn we need to add the encryption columns mentioned above. I have created a Stored Procedure called **add\_encryption\_columns** that will automatically create these columns using the naming convention above as well as the index on the hash column, if the hash column is created. To use the procedure from an Origenate sql script use the following sql example:

Exec add\_encryption\_columns (‘REQUESTOR’,’SOC\_SEC\_NUM\_TXT’,32,32);

Where:

Parm1: The table name containing the column to be encrypted.

Parm2: The original column name of the column in the table that will be encrypted.

Parm3: The length of the new encryption column, using the technique described above.

Parm4: The length of the hash column if defined. **Note: If zero is passed then a hash column will not be created.**

Using the above example the following columns and indexes will be created in the Requestor table:

Original column: SOC\_SEC\_NUM\_TXT

Encrypted column name: SOC\_SEC\_NUM\_TXT\_ENC

Hash column name: SOC\_SEC\_NUM\_TXT\_HASH

Hash column index name: SOC\_SEC\_NUM\_TXT\_HNDX

The procedure has been written in such a way that it can be stored in the evaluate schema and used the same way.

For Origenate, when we want to convert SSN to use encryption we will need to define a script that calls this procedure for each table that contains a SSN column.

## Key and User Management

We will need to discuss how we want to set up users, groups and keys for various operations. For example, do we want to have a single hmac key used for generating hashes for any column in the database or do we want separate keys for each column. Do we need to define different groups of users for diferent operations or just a single ‘origenate’ user.

Note: A lot of these decisions have already been made. See *Managing Users and Keys in the Appliance*.

There is a good document provided by SafeNet that helps answer these questions based on our business needs. It is ***Best Practices – Keys\_Users\_Groups.pdf***. If anyone needs a copy I can send it to them.

## Encrypting/Decrypting Files

I have successfully created methods to encrypt and decrypt any file. It goes thru the same process as encrypting a text value but uses File Input streams to read and write files. This means that where ever this code is installed it will need access to the local file system. Ie, Encrypting XML files on disk at the gateway.

# Managing Users and Keys in the Appliance

How we set up crypto config in Origenate and elsewhere is dependent on how we manage users and keys in the appliance. A good place to start for understanding how we should organize keys can be found in SafeNet’s document: ***Best Practices – Keys\_Users\_Groups.pdf***. I can provide a copy if you do not have it. It outlines suggestions on how to define keys and users in light of a company’s requirements. Our requirements have been outlined in a document provided by Dave called: ***Database Encryption Planning.docx***. I will simplify these requirements so I can suggest a way of managing users and keys in the appliance.

Since a single appliance can be used to manage keys for all evaluators we need to define a naming convention for users that will allow us to log at the appliance what operations were done by which evaluator. Our goal is to make the calls to encrypt/decrypt values as simple as possible from the client’s perspective.

**Note:** user names, key names, and group names are case sensitive. I suggest that all names use lower case.

## Users

Our encryption needs will be spread across many different environments, eg. Origenate, Evaluate, Gateway, etc. Within these environments we will need to encrypt individual fields like SSN or entire files that may contain sensitive data. The SafeNet appliance logs info about crypto operations based on user name. They suggest creating different users for different classes of action to facilitate debugging when receiving crypto commands from different apps or locations. I suggest creating a separate user for each different environment or action. Here are the major areas I think we should create a separate user for:

Origenate

Evaluate

Gateway

XMLDBT

Portal

Note: A ‘file’ user does not need to be defined because we will be using a single file key to do all file operations. So, if a file were to be encrypted on the gateway, the gateway user would be used. On the appliance it will show that a cmsi\_file\_key was used from the gateway.

For example, calls from Origenate may use a user called ‘origenate’, while calls from evaluate might use ‘evaluate’. To complicate things, we may also be using a single appliance to manage users for multiple evaluators (ASP mode). Hence, we will append the user name with a schema identifier. Here are the users we will be defining:

Origenate – origenate\_xx

Evaluate – evaluate\_xx

Portal – portal\_xx

Gateway – gateway

XMLDBT – xmldbt\_xx

Where ‘xx’ is the schema name

Note: It was decided that the ‘xx’ value will be derived from the database schema name. In Origenate this can be found in the origenate.ini file under **database.user**. Evaluate maintains its own schema name but it will be consistent with Origenate’s value. For the Gateway it will not matter because we are only doing file conversions under a common user name of ‘gateway’.

Note: Since gateway encryptions apply across all evaluators a single user name will be used: gateway

This user only needs to be defined once in the appliance.

## Groups

User names are associcated with groups and groups are associated with keys in the appliance. While group names do not need to be specified by the client we will create one group to be used by all evaluators:

cmsi

**Note:** Groups can be assigned different permissions. For instance Group1 users can only do encrypt operations while Group2 can do both encryption/decryption. I do not see a need for this level of control to satisfy our requirements. That is why I am proposing a single group for all evaluators that can encrypt/decrypt.

Every time a new user is created it should be added to the cmsi group.

## Keyowners

When creating keys, each key has a keyowner. The keyowner must be a defined user. For security reasons, SafeNet suggests that the owner of the key is not the same as users that have access to the key. We will define one user called ‘keyowner’ that can generate keys for any purpose.

## Keys

SafeNet suggests that a separate key be used for different fields in an application. This is because if the requirements for a particular field change such that we need to create a different key or version then it will not affect other fields in the application. If a field is used across environments, eg Origenate and Evaluate then we should use the same key in both environments. This helps support portability because the same key will be used on both ends to generate the same value to store and or retrieve values.

**Note:** All keys defined below will use versioned AES-256 bit keys except for Hashing keys; they need to use hmacSHA1 keys.

### Defining Keys for Field Level Encryption

We have decided to use one set of field level keys for all evaluators. This will simplify key management in the appliance. For instance, if we need to create a key to encrypt ssn across all environments then all evaluators will use this key. The naming convention used when defining a key will be as follows:

cmsi\_xxx\_key

xxx – will be a generic name of the field we are trying to encrypt, like ‘ssn’. Ie. cmsi\_ssn\_key

Each touchpoint that needs to encrypt a value will know what field they are trying to encrypt (ssn). When making a client side call they will simply provide ‘ssn’ as the field name. The crypto rtns will automatically create a key name of cmsi\_ssn\_key when it references the key in the appliance.

In our first release it has been decided that the following key categories will be created to satisfy all Federal and State requirements. Others may be added later as needed.

1. Social Security Number/Social Insurance Number –cmsi\_ssn\_key
2. Driver’s License number – cmsi\_drivers\_key
3. Trade account numbers – cmsi\_trade\_key
4. Bank account numbers - cmsi\_bank\_key
5. DB passwords stored in .ini files – cmsi\_dbpwd\_key

Therefore, the keynames that you pass into the encryption rtns will be as follows:

ssn

drivers

trade

bank

dbpwd

Note: always use lower case.

### Defining Keys for File Encryption

When ever we store sensitive data in a file on disk we will need to encrypt the file. Files are generated at various touchpoints in all environments. For example:

Gateway history files.

Oriegante, Evaluate logs.

Nightly reports.

Servicing file dumps, etc.

It has been decided that we will use a single key for all file encryption across all evaluators. The name of the file key will be:

cmsi\_file\_key

This key will be used across all evaluators.

We also need to discuss at what point do these files need to be decrypted and by whom. Will we need to provide a decrypt file utility? If so, how will we secure it, etc.

**Note: If you want to use the cmsi\_file\_key using the encryptString function then pass in a key name of ‘file’ (not cmsi\_file\_key).**

### Defining Keys for CLOB Encryption

There are many tables in Origenate that contain clob columns that can potentially store sensitive data. They are:

TABLE\_NAME

------------------------------

CREDIT\_REQUEST\_XML

CREDIT\_REQ\_CIS\_APP\_ENTRY\_CLOB

CREDIT\_REQ\_CONTR\_MPE

CREDIT\_REQ\_TITLE\_DEED

CREDIT\_REQ\_TITLE\_PHRASE

CREDIT\_REQ\_TITLE\_RESP

CREDIT\_REQ\_TITLE\_RESP

POSTING\_QUEUE

CREDIT\_REQUEST\_ADDITIONAL\_DATA

CREDIT\_REQUEST\_PAYOFF\_MPE

REPORT\_GENERATION\_LOG

Since, we can not determine the location within a clob where a field like SSN may reside we have no choice but to encrypt the entire clob if it could contain ssn.

It has been decided that we will use a single versioned key for all clob encryption across all evaluators. The name of the key will be:

cmsi\_clob\_key

Also, a clob like CREDIT\_REQ\_CONTR\_MPE.MPE\_DATA\_CLOB can contain multiple sensitive data like ssn and account numbers, which will be sent to servicing systems. For this reason if an evaluator starts using encryption for *any* column then all clob columns in *all* tables will need to be encrypted with the cmsi\_clob\_key.

Note: Additional columns for encryption are not needed for clob columns. The data in the clob column will be encrypted and stored back in the same column.

**Important:** When encrypting values to be stored in a clob use the encryptString() method in crypto.jar. You MUST use a field name of ‘clob’ (lower case) to encrypt the string. When the method sees a field name of clob it will use the cmsi\_clob\_key to encrypt the value. The same goes for decryptString().

### Defining Keys for Generating Hash Values

The last type of key we need to define is a hmac key for generating hash values from clear-text to generate a searchable hash value for storage in our database. This key needs to be defined as a hmacSHA1 key. This key will also be defined as non-versioned key. Since the values generated by this key are searchable, unique, and can not be decrypted we do not need to define a separate hmac key for each evaluator. Hense, we will define a single hmac key in the appliance called:

hmac\_key

# Turning Encryption On and Off

## Origenate

We want to be able to control at the evaluator level which fields they want to encrypt and weather encryption should take place. To do this we will create a new table called: config\_encryption. It will contain the following columns.

evaluator\_id

evaluate\_client\_id\_txt

ssn\_flg

trade\_flg

bank\_flg

drivers\_flg

The table will be keyed by evaluator ID. There will be a flag column for each possible field that we support to encrypt. If a row in this table does not exist then the evaluator is not using encryption. If an evaluator IS using encryption then a row will be added and each flag column will need to be set to determine if it should be encrypted or not.

When we add a new field level key in the appliance it will take the naming convention of:

cmsi\_xxx\_key

xxx – will be a generic name that identifies the field to be encrypted across all environments (ssn). The corresponding flag column will then be named xxx\_flg, ie) ssn\_flg.

### Note: The column names must match the generic keyname fields assigned in the section: Defining Keys for Field Level Encryption

## Gateway

For the Gateway it’s a different story. We do not have access to the database and the gateway is used to handle traffic for multiple evaluators. The Gateway is transient in nature. For instance when we receive an app from DealerTrack it will contain a clear text value of ssn. The gateway converts DT’s XML to LoanML (containing the ssn) and posts it to the Post Queue Processor. After the data is posted to the PQP it is deleted from its input directories. Hence, we will not need to do field level encryption within the gateway. However, the gateway also stores a copy of the input and output files for each transaction in the history dir. In this case I suggest that we use the file encryption key to do this for all evaluators. We can control this option by defining a new colt.ini file parameter flag. But again, once turned on, this encryption will apply to all evaluators.

Note: It was decided that we will not encrypt dead files since they are transient in nature. This will also allow support to repush dead files without having to decrypt a dead file.

Note: It was also decided that COLServlet will not encrypt files that get deposited in Colt’s input directories. This because they are short lived and we do not want to introduce a performance bottleneck within COLServlet.

## Evaluate

Need to discuss how to control turning on and off at the Evaluator level.

# Implementation

## Setting up the Client Side Environment

### Crypto.jar

I have developed a new jar file which contains all the functions for doing encryption/decryption with the SafeNet appliance. It is called crypto.jar. The source for the classes in this jar can be found in VSS under common/src/…/origenate/crypto. I put it under common so it can be built each time our build is done. The classes in this dir are not included in common.jar because we do not want to have to deliver common.jar to external environments like the Gateway or Evaluate. I adjusted build.xml to create the crypto.jar file and deploy it just like common.jar. This jar file can be used by Origenate and environments outside of Origenate, like, Evaluate and the gateway.

Note: Sys Eng will need to include crypto.jar in the classpath for things like the RoutingQueueProcessor if that process will be using crypto operations. The classpath will also be adjusted to include the crypto supporting jar files mentioned below.

### Classpath Changes

The following new jar files have been added to the lib dir in Origenate. They will need to be included in the classpath of any environment that wishes to use the new crypto operations.

crypto.jar

Commons-codec-1.3.jar

IngrianLog4j.jar

IngrianNAE-5.0.0.jar

Cryptix-jce-api.jar

### Policy Files to Support AES-256 bit keys

256 bit keys have a client side config impact. If we use 256 bit keys (which we probably will) then we need to download special JCE policy files that we store in the jre/lib/security directory. This is necessary to use unlimited strength keys. The files need to come from either Sun or IBM depending on the environment, ie Jrun vs Websphere.

Notes from JCE doc on where to get policy files:

To use unlimited strength ciphers (e.g. 192- and 256-bit AES keys) *you must download the*

*encryption policy files* for your Java implementation.

For a Sun Java implementation, download the Java Cryptography Extension (JCE) Unlimited

Strength Jurisdiction Policy File from

http://java.sun.com/javase/downloads/index.jsp.

For an IBM Java implementation, download the JCE Policy Files from

http://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=jcesdk

For more information on the IBM download, see

http://www-128.ibm.com/developerworks/java/jdk/security/50/

**Installation**: Backup jre/lib/security/local\_policy.jar and US\_export\_policy.jar and then copy the new ones in from the download.

### Client side config files, crypto.ini and IngrianNAE.properties files

#### crypto.ini

Crypto.jar will access an INI file called crypto.ini. The ini file will specify all user names and encrypted passwords needed to access the appliance. Here is an example of its contents:

[users]

origenate\_xx = GF5CT0hPS0lC

Using the example below, this is how the crypto.ini file will be used:

crypto.encryptString(“origenate\_xx”,”ssn”,value);

When connecting to the appliance the underlying code will use the user name passed ‘origenate\_xx’ and access the crypto.ini file to get the password needed to connect to the appliance.

It will then construct a key name called cmsi\_ssn\_key to be used to encrypt the value passed.

This approach simplifies client encryption since it only needs to know the name of user making the call (for logging and connection purposes) and the name of the field it is trying to encrypt.

Crypto.jar needs to know the location of the crypto.ini file. It has been decided that a common location will be used across all environments. The location will be:

/safenet – for windows

/.safenet – for unix

When setting up your environment create the above directory and install the crypto.ini file in it. When a user is added to the appliance the crypto.ini file will need to be updated to include it.

Note: Crypto.jar has been enhanced to cache the ini file so it does not have to be read on each crypto operation. It has also been enhanced to detect when the file has been changed so it can be reloaded into memory to avoid bouncing.

#### IngrianNAE.properties File Location

The IngrianNAE.properties file contains ini file settings that dictate how a client communicates with the Appliance. Ie) The IP address and port of the appliance as well as SSL parameters, etc. Once this file is configured it should be used across all environments that need client access.

This file should be located in the /safenet directory as well.

Note: The name of the properties file MUST be IngrianNAE.properties

### Securing Passwords on the Client Side

After we define a user and pwd in the appliance we need to connect to the appliance with the same user name and password from the client software. Since the password is stored locally at the client in the crypto.ini file we do not want to store it as a clear-text value. If we did and a hacker got a hold of it then they would gain access to the appliance and decrypt our encrypted values. To avoid this I wrote a separate Encryption and Decryption tool that can be used to encrypt a password before we define it in crypto.ini. These utils are in VSS under origenate/tools/encryption and consist of two java applications (Encrypt and Decrypt). These are not deployed as part of our standard rollout.

Note: these utils access methods in crypto.jar to encrypt/decrypt passwords.

After Sys Eng sets up a new user name and password in the appliance they would use the Encrypt util to encrypt the password into a XOR base64 encoded character string. This value would then be stored in our crypto.ini file.

To use these utils you will need the following jar files in your classpath. They can be obtained from the lib directory in Origenate.

Crypto.jar

Commons-codec-1.3.jar

IngrianLog4j.jar

IngrianNAE-5.0.0.jar

Cryptix-jce-api.jar

Usage:

Java Encrypt passwordToEncrypt

Ie) java Encrypt glenn

The output will be shown as a string that can be stored in the database, ie: TUZPREQ=

This is the value that will be stored in crypto.ini.

To decrypt this value do the following:

Java Decrypt TUZPREQ=

The output will be ‘glenn’.

### Origenate.ini File Changes

A new origenate.ini file parameter will be added that controls global encryption that is not evaluator based. It is under a new section called ‘encryption’.

[encryption]

encryption\_flg = 0 # valid values: 1 – on 0 – off

When the flag is on any non-evaluator based encryption will take place in all places where needed.

Examples:

1. Storing clob data in the posting\_queue.
2. Saving xml data in credit\_request\_xml.
3. Etc

Note: For Coldfusion, configGlobalconstants.cfm will need to be adjusted to save this value in the application scope.

These values will be called

Application.encryption\_flg

Application.encryption\_user\_name (equates to database.user in origenate.ini)

# Client Side Coding and Usage

Now that we have the client environment set up we can now make calls to crypto .jar to perform crypto opertions. This section provides examples of calling crypto operations from Java and ColdFusion.

### Java Examples

In any java code that needs to access crypto rtns the following import needs to be defined:

import com.cmsinc.origenate.crypto.\*;

To encrypt a value the following calls would be made: (this is just an example using origenate as the user and ssn as the field we want to encrypt)

Crypto crypto = CryptoFactory.getCrypto();

String encryptedValue = crypto.encryptString(“origenate”,”ssn”,”123684567”);

If we wanted to encrypt ssn, generate a hash, and generate a mask then we would do the following:

Crypto crypto = CryptoFactory.getCrypto();

String encryptedValue = crypto.encryptString(“origenate”,”ssn”,”123684567”);

String hashValue = crypto.generateMAC(“origenate”,”123684567”);

String maskValue = crypto.generateMask(”123684567”);

These values would then be stored in their respective columns.

#### Masking clear-text values

When we encrypt a value like SSN and store it in our database in a separate column we will want to store a ‘masked’ representation of the SSN in the original ssn column. To do this you will make a call to the crypto.jar file. For instance, to generate a mask for ssn 212441234 you would make the following call:

Crypto crypto = CryptoFactory.getCrypto();

String maskedValue = crypto.generateMask(“212441234”);

The returned value will be: \*\*\*\*\*1234

The method assumes that the last four characters of the input value will be preserved, prefaced with asterisks. Another (overloaded) version of the method is also available so that you can specify how many trailing digits will be preserved. Example:

Crypto crypto = CryptoFactory.getCrypto();

String maskedValue = crypto.generateMask(“212441234”,3);

The returned value will be: \*\*\*\*\*\*234

Yet another overloaded version allows you to specify the trailing chars as well as the mask.

Crypto crypto = CryptoFactory.getCrypto();

String maskedValue = crypto.generateMask(“212441234”,3,”a”);

The returned value will be: aaaaaa234

### Encrypting a Clob Value

Crypto crypto = CryptoFactory.getCrypto();

// get the clob value in this row

ps = con.prepareStatement("select "+columnName+" from "+tableName+" where rowid = '"+rowID+"'");

rs = ps.executeQuery();

rs.next();

clob = rs.getClob(1);

origValue = clob.getSubString(1, (int)clob.length());

rs.close();

ps.close();

String encValue = crypto.encryptString(userName,keyName,origValue);

// store encrypted value back in original column

ps = con.prepareStatement("UPDATE "+tableName+" set "+columnName+" = ? where rowid = ?");

oracle.sql.CLOB newClob = oracle.sql.CLOB.createTemporary(con, false, oracle.sql.CLOB.DURATION\_SESSION);

newClob.putString(1,encValue);

ps.setClob(1, newClob);

ps.setString(2,rowID);

ps.executeUpdate();

ps.close();

### Decrypting a Clob Value

Crypto crypto = CryptoFactory.getCrypto();

// get the clob value in this row

ps = con.prepareStatement("select "+columnName+" from "+tableName+" where rowid = '"+rowID+"'");

rs = ps.executeQuery();

rs.next();

clob = rs.getClob(1);

encValue = clob.getSubString(1, (int)clob.length());

rs.close();

ps.close();

origValue = crypto.decryptString(userName,keyName,encValue);

// store clear-text back in original column

ps = con.prepareStatement("UPDATE "+tableName+" set "+columnName+" = ? where rowid = ?");

oracle.sql.CLOB newClob = oracle.sql.CLOB.createTemporary(con, false, oracle.sql.CLOB.DURATION\_SESSION);

newClob.putString(1,origValue);

ps.setClob(1, newClob);

ps.setString(2,rowID);

ps.executeUpdate();

ps.close();

### A Full List of Methods in crypto.jar

The following is a complete list of crypto methods available:

// encrypted string is returned as a base64 encoded value

public String encryptString(String callerID,String fieldName,String fieldValue) throws Exception;

// value to decrypt is supplied as a base64 encoded encrypted value, returned value is clear-text

public String decryptString(String callerID,String fieldName,String stringToDecrypt) throws Exception;

// these methods are used to encrypt/decrypt files

// note, srcFile and destFile can not be the same

public void encryptFile(String callerID,String srcFile,String destFile) throws Exception;

// note, srcFile and destFile can not be the same

public void decryptFile(String callerID,String srcFile, String destFile) throws Exception;

// method to generate a searchable MAC value for a given string, common across all versions of a key and searchable

public String generateMAC(String callerID,String value) throws Exception;

// encrypt a user password using XOR / base64 encoding

public String encryptPassword(String pwd) throws Exception;

// assumes in value was generated with encryptPassword

public String decryptPassword(String encryptedPwd) throws Exception;

// returns the in value as all asterisks with the last n chars preserved

public String generateMask(String valueToMask, int preserveCnt) throws Exception;

// returns the in value as all asterisks with the last 4 chars preserved

public String generateMask(String valueToMask) throws Exception;

// returns the in value as all 'maskingChar' with the last n chars preserved

public String generateMask(String valueToMask, int preserveCnt,String maskingChar) throws Exception;

### ColdFusion Example

The following shows how to call a crypto method directly from ColdFusion. This is just an example of encrypting ssn but once you get this you can call any method in crypto.jar.

<CFOBJECT ACTION="CREATE" TYPE="java" CLASS="com.cmsinc.origenate.crypto.CryptoFactory" NAME="cryptoFactory">

<CFSET cryptoInst = cryptoFactory.getCrypto()>

<CFSET test = cryptoInst.encryptString(“origenate”,”ssn”,"999999999")>

<cfoutput>#test#</cfoutput>

### Decrypting a clob value under CF

Decrypting a clob value is no different then decrypting a string value, just retrieve the clob value and then call the decrypt method. For example:

<CFQUERY NAME="qryXML" DATASOURCE="#REQUEST.APPLICATION.CreditOnlineDSN#">

SELECT value\_txt AS xml\_txt

FROM posting\_queue

WHERE transaction\_id = <CFQUERYPARAM value = "#ATTRIBUTES.inpTransID#" CFSQLType = "CF\_SQL\_FLOAT" NULL="#YesNoFormat(ATTRIBUTES.inpTransID EQ '')#">

AND (posted\_field\_name\_txt = 'XML' or posted\_field\_name\_txt like '%WDDX')

</CFQUERY>

<CFOBJECT ACTION="CREATE" TYPE="java" CLASS="com.cmsinc.origenate.crypto.CryptoFactory" NAME="cryptoFactory">

<CFSET cryptoInst = cryptoFactory.getCrypto()>

<CFSET test = cryptoInst.decryptString(“origenate”,”clob”,#qryXML.xml\_txt#)>

<cfoutput>#test#</cfoutput>

Note: remember to use the key name of ‘clob’ when decrypting a clob. This should have been the same key name used when encrypting a clob.

# Test Driver Program

I have developed a command line Java test driver program that can be used to exercise all crypto functions provided by crypto.jar. The driver program is called Driver.class and it can be found in VSS under tools/encryption. A setup and usage guide for it can also be found in the encryption directory. It is called: ***TestDriverUsage.docx***

# Encrypting Values in an Existing Table before Going Live

Once an evaluator decides to encrypt a field like ssn we need to provide a way to convert any existing ssn values in their database before they turn on the switch to use encryption. To do so we need to take the following steps:

1. Run the add\_encryption\_columns **stored procedure** to create the columns needed for encryption in **each** table that contains SSN. Note: Some tables may not need to create a hash column for the encrypted value if it will not be searched on. This will require an Origenate script that calls the stored procedure for each table that contains ssn.
2. Run the Table Converter java program against each table to encrypt existing ssn values.
3. Turn on the encryption flag for ssn in the config\_encryption table for this evaluator.

The Table Converter program is called TableConverter.class and can be found in VSS under the db/tools directory. A setup and usage guide for it can also be found in the tools directory. It is called: ***TableConverterUsage.docx***

# Converting CLOB columns in an Existing Table before Going Live

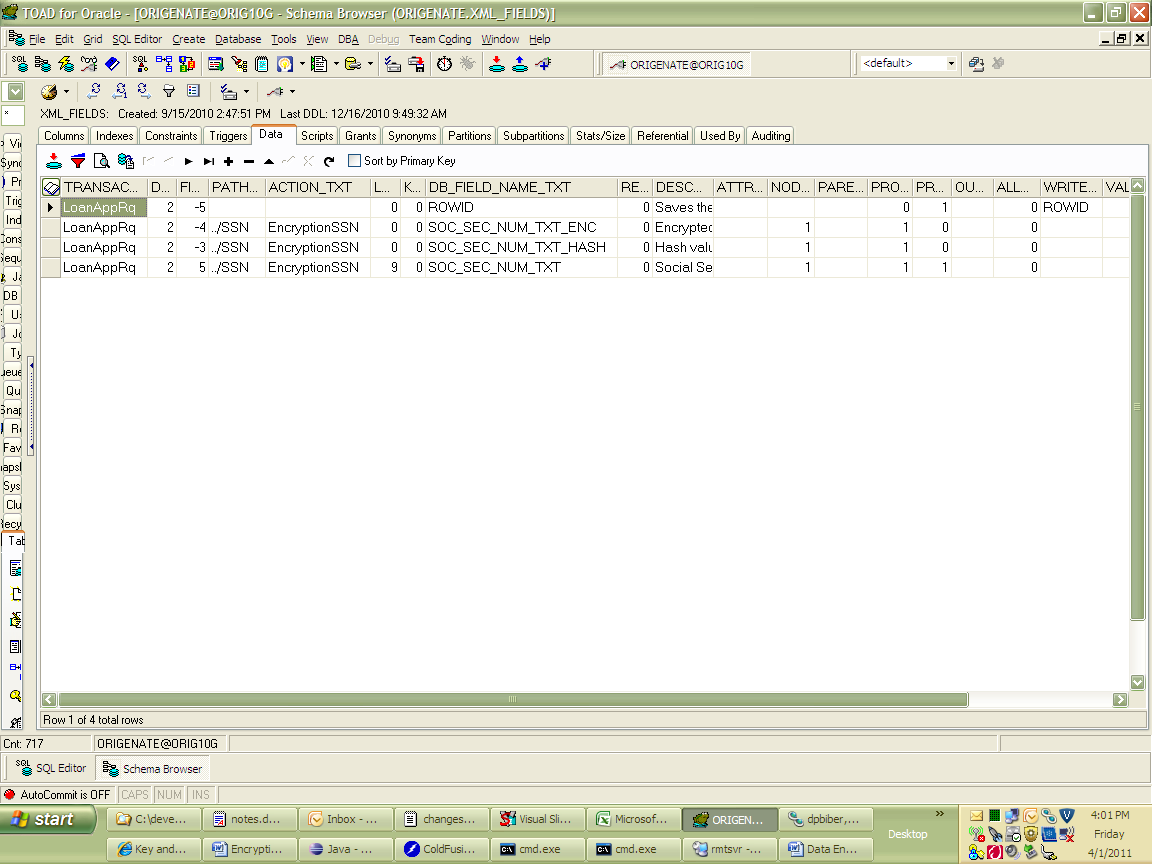
As mentioned earlier in the document if any column is to be encrypted, like ssn, then all clob columns will need to be encrypted that might contain ssn. A separate program has been created to encrypt clob columns because they need to be handled differently. It is called ClobConverter.class and can be found in VSS under the db/tools directory. A setup and usage guide for it can also be found in the tools directory. It is called: ***ClobConverterUsage.docx***

Note: After running this program it will replace the original clob value with the encrypted clob value. No new columns need to be added.

# XMLDBT Changes

XMLDBT needed to be changed to support encrypting values on inbound apps and optionally generating clear-text of encrypted values when generating xml. For example, when generating xml for servicing we will want to expell clear text for all fields. But for view app a user may be able to see clear text for some fields but not others. This will be controlled by the user’s security fuction list.

When a field in a table is to be encrypted (like requestor.ssn) then we will run the stored procedure to create the additional encryption fields. If this field is used in any transaction\_type of xmldbt then we need to tell xmldbt that the field might need to be encrypted on inbound and possibly decrypted on outbound. The following config chgs need to be made for any table containing said field. These changes are confined to the xml\_fields table. I will use requestor.ssn as an example. The following is a screenshot of the TOAD rows in xml\_fields needed to implement encryption for this column. This is just an example and will need to be replicated for any field that is being encrypted.



When we added encryption fields for ssn to the requestor table we called the stored procedure to add the soc\_sec\_num\_txt\_enc and soc\_sec\_num\_txt\_hash columns. Since ssn is captured on incoming apps we need to update the LoanAppRq config in xml\_fields to accommodate these columns.

## Adding new xmldbt config rows to support encryption for a column

The original ssn column defined in xml\_fields (field\_seq\_num 5) does not change except for specifying a new conversion object for it under action\_txt (EncryptionSSN). The conversion object will do all of the work to handle masking, encrypting, and defining hash values. A new conversion object has already been created for each data type that we will initially support. They are:

EncryptionSSN

EncryptionDrivers

EncryptionTrade

EncryptionBank

We now need to create a row for the new hash and enc columns.

**IMPORTANT!!!**

**The field\_seq\_num value for these rows MUST be lower than the field seq num value of the original column (in this case 5). It is OK to assign negative values as a field seq num value since in this example all possible positive numbers below ssn’s field seq num were already taken.**

Each row that you add will reflect the same values as the original ssn row except for the following:

1. Set the action\_txt to EncryptionSSN or the name of the conversion object that relates to your data type. Valid values are: EncryptionSSN, EncryptionBank, EncryptionTrade, and EncryptionDrivers.
2. Set the length\_num to 0.
3. Set the db\_field\_name\_txt column to either the hash or enc column you are defining.
4. Set thedescription\_txt appropriately. I have provided a generic description that should apply to any data type.
5. Set the process\_in flg to 1 and the process\_out flg to 0.
6. Set the validate in flg to 0
7. Set the data type to text

Note: If the field to be encrypted does not have a \_HASH column then do not define a row for it.

## Creating the ROWID column definition

We need to add one more row to xml\_fields to optionally support outbound decryption of the ssn field. Using the exmaple above define a row and make sure the following values are set.

1. **Make sure the field\_seq\_num is LOWER than all the previous rows defined for ssn.**
2. Set its path\_txt to null
3. Set its action\_txt to null
4. Set its length num to 0
5. Set its db\_field\_name\_txt to ROWID
6. Use the generic description provided
7. Set its node type to null
8. Set its process\_in flg to 0
9. Set its process\_out flg to 1
10. Set its write\_var\_txt to ROWID
11. Set its validate\_in\_flg to 0
12. Set it data types to text

## How these new column definitions are used

### Inbound XML

For inbound xml transactions xmldbt will process the incoming ssn value for the enc and hash columns first. It will encrypt the clear-text ssn value in the xml and store it in the \_enc column in the database. It will then create a hash value and store it in the hash column. Finally, it will process the originall ssn field (seq 5) and generate a mask and store it in the original ssn column.

Note: These new column defs will apply to all installations going forward. If a particular evaluator is not encrypting ssn then the EncryptionSSN conversion object will automatically adjust and process ssn the old way. It does this by checking the ssn\_flg in the config\_encryption table.

### Outbound XML

When generating outbound xml only the ROWID and original ssn column are processed. For the original field if encryption is on, the conversion object will query the ENC column value using the rowid of the table retrieved earlier. If the field is to returned as clear text the object will decrypt the enc column and return its value. If clear text is not to be returned it will return the original fields masked value.

### Summary for XMLDBT field changes

In summary, we updated the original ssn column def to point to the EncryptionSSN conversion object and then added three new rows (all with lower field seq num values) to populate the encryption and hash columns.

## Calling GenX to generate XML

GenX is used to generate xml at various points throughout Origenate. For instance it is called by MPE to generate XML for servicing. It can also be called by View App to display application data to an end user. Depending on the context we may want to generate clear text values when generating the xml. For instance, in MPE we will always want to generate clear text for the servicing system. For view app, depending on the end users security functions they may be able to view account numbers but not the ssn.

To accomplish this when the calling code gets ready to call GenX it needs to first gather this information and set parameters in GenX before instructing it to generate xml. A CF example of this follows:

<CFTRY>

<CFOBJECT ACTION="CREATE" TYPE="java" CLASS="com.cmsinc.origenate.xmldbt.GenX" NAME="GenXInst">

<CFSET inst = GenXInst.init("#REQUEST.APPLICATION.cfx\_genx\_log\_file#",

"#REQUEST.APPLICATION.client\_name#",

JavaCast("int","#REQUEST.APPLICATION.genx\_debug\_level#"),

"#param\_names#",

"#param\_values#",

"#ATTRIBUTES.additional\_xml\_xpaths#",

"#ATTRIBUTES.additional\_xml\_values#"

)>

<!--- Add code here to check the user function list to determine what the user is allowed to see.

Only specify keynames that they are allowed to see,separated by commas. In this test I hardcoded the

list to ssn and drivers (for drivers license number). Also notice that we need to pass in the db\_user

from the origenate.ini file set in the application scope by configglobalconts.cfm. You may want to

pass in your new application scope var that has the same value.

--->

<CFSET allowedToLookAt = "ssn,drivers">

<CFSET s\_xml = GenXInst.setEncryptionParms("#REQUEST.APPLICATION.db\_user#","#allowedToLookAt#")>

<CFSET s\_xml = GenXInst.sGetXMLorThrow(JavaCast("String","#attributes.trans\_id#"),

JavaCast("String","#attributes.apply\_xsl#"),

JavaCast("boolean","#sechaspriv#")

)>

<CFSET sVoid = GenXInst.cleanup()> <!--- GL. Must return the connection to the pool --->

<CFCATCH>

<CFTHROW MESSAGE="Error GenX - #cfcatch.message# - #cfcatch.detail#">

</CFCATCH>

</CFTRY>

Notice that we now have a new GenX method called setEncryptionParms that allows us to pass in the db\_user and list of keynames that we should generate clear text for. In the case of MPE where we want to generate all fields as clear text we will have to pass in a list of all possible keynames. For example:

ssn,drivers,bank,trade

# Checklist for adding an encrypted field

Whenever we want to encrypt a column in a particular table we need to perform the following steps.

Note: each of these steps are defined in detail above.

1. Create a script to generate the enc and hash columns using the stored procedure defined previously. Remember to calculate the length of the enc and hash columns.
2. Run the Conversion utils to encrypt all existing values in the table. Again described above.
3. If this column is used in xmldbt then make the appropriate changes to xml\_fields.
4. Scan the code for all places that access this column (CF or Java) and make approprite crypto calls. Remember, to check the config\_encryption table by evaluator to see if you should use encryption or not.
5. If the column value could possibly appear in any clob column perform clob conversions described above. Remember, clob conversions are based on the encryption\_flg in the origenate.ini file so it might be worth just updating all code that handles clobs once.
6. As always, test, test, test.

# Gateway Changes to Support Encryption

## Java 1.6 Version of COLT

COLT has been upgraded to run under Java 1.6. A new colt.jar has been made which will only run under 1.6. When started it will write a line to the log indicating the new build:

COLT v3.0.0Java1.6 build 10192011.1055 Initialization complete, running...

## Turning Encryption on in COLT

The new colt has also been enhanced to support encryption. It was determined that the only data-at-rest processed by COLT is the files written to the history dir. These files can potentially contain sensitive data and therefore must be encrypted. Dead files will not be encrypted because they are transient in nature and are usually handled by support shortly after they are created.

A new ini file parm has been added to the colt.ini file called:

encryption-flg = 1

If set to 0 or not specified then colt will not encrypt history files.

When set to 1 then colt will encrypt all files written to the history dir. For each transaction colt processes, colt writes the incoming file to the history dir with an ‘.in’ extension. It also writes the outgoing converted files as an ‘.out’ file. When encryption is turned on the both the .in and .out files will be encrypted with a .enc extension appended to the file name. For example, if the .in file is named:

20111024.1100.xml.in then the resulting encrypted version of the file will be named 20111024.1100.xml.in.enc and the original file (20111024.1100.xml.in) is removed.

Note: COLT will use a user name of ‘gateway’ when encrypting files with the ‘file’ key for logging purposes.

## Decrypting Encrypted Files

A new utility has been created to allow support to decrypt these files if necessary. The utility is called DecryptFile.class. It can be found in VSS under Origenate/tools/encryption. Using the above file name as an example you can decrypt the encrypted file using the following command:

java DecryptFile 20111024.1100.xml.in.enc

The resulting clear text file will be written as 20111024.1100.xml.in.

Note: This utility can only be used to decrypt history files that were encrypted with COLT. Also, the file name provided must end with a .enc extension. The utility will decrypt the .enc file into a file name without the .enc extension and leave the .enc file inplace.

## COLT Environment Changes

Whether encryption is turned or not the new COLT requires the supporting encryption jars mentioned previously in this document. They will need to be added to the classpath.

If encryption is turned on or you are using the decryption utility then you will additionally need to install the unlimited strength key policy files mentioned in this document as well as create the /.safenet directory and setup the crypto.ini file and the Ingrian properties file.

# Encrypting Applications by Request ID

A feature was added to pass starting and/or ending REQUEST\_ID to select only those applications to perform encryption on. For usage examples please see Tools/encryption/TestDriverUsage.docx.

Usage:

encrypt\_all.sh ENVNAME EVALID STARTREQUESTID ENDREQUESTID

* End of document -