



CipherTrust Manager (CM) Platform

Tips & Techniques for Querying Databases with Column Level Encryption

Document Version 1

Content

PREFACE	3
DOCUMENTATION VERSION HISTORY	3
ASSUMPTIONS	3
GUIDE TO Thales DOCUMENTATION	3
SERVICES UPDATES AND SUPPORT INFORMATION	3
GETTING STARTED	4
Options Summary	4
Option Criteria	5
CipherTrust Manager Requirements	5
Options for Querying a Database Table with column level encryption.	6
Database Shadow Table Example.	6
Custom Database User Defined Function Example.	8
Appendix	12
Performance Considerations	12
CipherTrust Manager Detailed Architecture.	12
Enabling local mode for ProtectAPP	13

PREFACE

This document explores options on querying a database table that has encrypted columns.

DOCUMENTATION VERSION HISTORY

Product/Document Version	Date	Changes
V1.0	08/2020	M. Warner Initial document release

ASSUMPTIONS

This documentation assumes the reader is familiar with the following topics:

- Java
- Key management
- Data encryption
- Familiarity with databases

This document uses the Vertica database as an example but the process would be the same if another database were utilized instead.

Note: In Sept of 2020 Thales has rebranded the KeyManager named KeySecure or KeySecure Next Gen to CipherTrust Manager (CM). It combines capabilities from both the legacy Gemalto KeySecure and the Vormetric Data Security Manager products. Any reference in documentation to KeySecure , NextGen or Data Security Manager can be considered to now be the newly branded CipherTrust Manager (CM) product. See following link for more details:

<https://cpl.thalesgroup.com/encryption/ciphertrust-manager>

GUIDE TO Thales DOCUMENTATION

Related documents are available to registered users on the Thales Web site at

<https://supportportal.thalesgroup.com/>

SERVICES UPDATES AND SUPPORT INFORMATION

The license agreement that you have entered into to acquire the Thales products ("License Agreement") defines software updates and upgrades, support and services, and governs the terms under which they are provided. Any statements made in this guide or collateral documents that conflict with the definitions or terms in the License Agreement, shall be superseded by the definitions and terms of the License Agreement. Any references made to "upgrades" in this guide or collateral documentation can apply either to a software update or upgrade.

GETTING STARTED

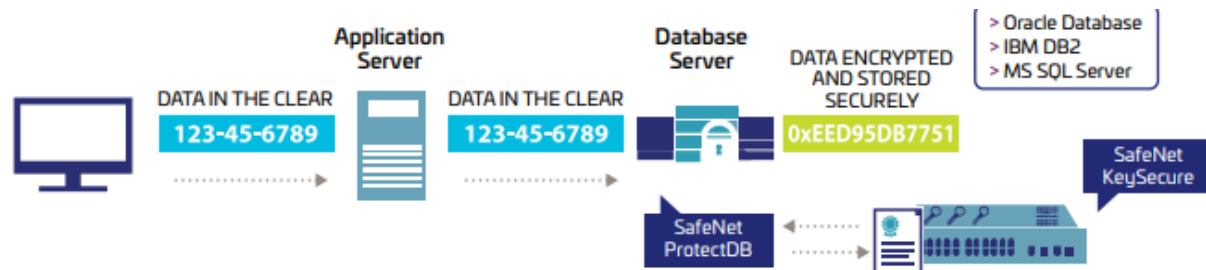
Options Summary

The goal of this document is to provide some simple examples showing how database users can still implement ad hoc queries or wild card searches against data in database that has implemented column level encryption. There are at least two approaches to solve this problem and this document will show working examples of both how a database function and a database shadow table would provide this capability.

Database Function.

The first option is a database function to encrypt/decrypt the column. Thales provides a capability called protectdb that provides UserDefinedFunctions for DB2, Oracle and Microsoft SQL Server.

https://cpl.thalesgroup.com/sites/default/files/content/solution_briefs/field_document/2020-04/protectdb-oraclesb-sb-a4-v4.pdf



If you have a database that is not one of the ones listed then a Custom User Defined Function can be written as well to implement this capability using the CM ProtectApp SDK.

The Thales ProtectApp SDK can be used for many different use cases. Typically, it is used for scenarios when a company has sensitive data in a field of a particular file or a column in a database and they want to encrypt the sensitive data. The ProtectAPP SDK provides Format Preserved Encryption (FPE) which preserves the format of the original data. Use cases can include:

- Encrypt SSN or credit card number data at the point of entry of an application.
- Encrypt PII data that might be in a file.
- Encrypt sensitive data before inserted into a PAAS based offering.

Note: Thales CM also provides a REST API to encrypt/decrypt data but for performance reasons that option was not considered for a database scenario. The code examples provided are not production ready samples, but only to demonstrate capability.

Database shadow table.

The other method is using a shadow table that contains a subset of the original column unencrypted data. For example if you want to search on just the first initial of the employee first name then store that in a shadow table. If you are interested in searching on the first 3 characters of the employee last

name then store that in the same shadow table. This shadow table must be included to queries that need to implement fuzzy searches.

Option Criteria

Listed below is a matrix that describes some of the differences between a database user defined function (UDF) and a shadow table.

	Query options	Maintenance	Leverage Code	Time to implement	End User	Performance
Database User Defined Function	More. Endless.	On database upgrade. Retest	Yes, can use for other purposes. (Re-encrypt with new key)	Longer. Will take more time to write UDF.	Can build a database view to simplify implementation	
Shadow Table	Less. Have to decide ahead of time.	Have to create database triggers or other methods to keep shadow table up to date.	No	SQL Statements. Decide what columns you need to query and how. Might need to revisit.	Can also build a database view to simplify, but will be more work for end users. Have to remember what they can query by.	Faster. Join is faster than custom UDF.

The factors in the matrix above can be used as a starting point for project teams. Weights should be assigned to the various factors depending on the priorities of the customer so an appropriate technology can be chosen.

CipherTrust Manager Requirements

The following Product Versions were used for this testing.

- KeySecure/Next Gen 1.9.1 or greater
- ProtectApp_JCE_v8.12.1_Bundle

Documentation

- SafeNet ProtectApp JCE User Guide

Options for Querying a Database Table with column level encryption.

Most implementations will have at least two CipherTrust Managers handling requests as the diagrams show below. The platform operates as a cluster and it is easy to add more nodes to the cluster if needed. The examples below use an AES 256bit key created in the CipherTrust Manager.

As you can see below when using the ProtectApp SDK it includes its own load balancing and connection pooling. The diagrams below shows how the local mode of operation would work with the keys cached on the host. A request for a key would only be done if the time threshold has been exceeded.

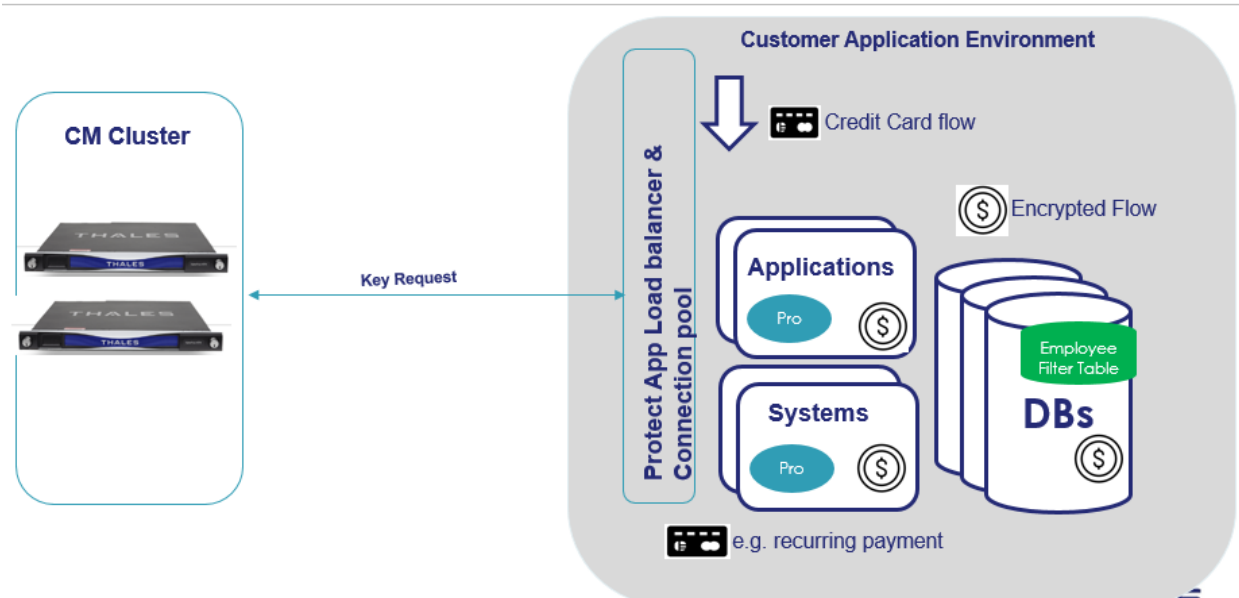
For both options the first step is to implement the initial encryption of the columns. This can be accomplished by either writing a JDBC/ODBC or ado.net application or via a database function. Both methods use the Thales ProtectApp API to implement the encryption.

Database Shadow Table Example.

Architecture


As you can see below the **application** will be the **only** means to decrypt data. The diagram shows how the ProtectApp is installed on any machine that needs to decrypt or encrypt data in the database. This example shows how the extra database filter table was created to support ad-hoc queries for our employee_dimension table. The data in the filter table will **NOT** be encrypted only the data in the employee_dimension table for our example.

Database Filter Table Option



Sample SQL Statements

This is a snapshot of the employee_dimension table **before** encryption was applied.

* 	employee_key	employee_gender	courtesy_title	employee_first_name	employee_middle_initial	employee_last_name	employee_age	hire_date	employee_street_address	employee_city	employee_state
1	1	Male	Sir	Craig	F	Robinson	22	2003-02-16	5 Bakers St	Thousand Oaks	CA
2	2	Male	Dr.	Thom	Q	Brown	52	1973-06-13	123 Hereford Rd	Fontana	CA
3	3	Female	Mrs.	Julie	E	Jones	17	2004-10-27	19 Lake St	Pueblo	CO
4	4	Male	Mr.	John	T	Smith	48	2004-09-17	225 Davis Rd	Springfield	IL
5	5	Male	Dr.	Sam	R	Jackson	57	1989-10-05	108 Market St	Beaumont	TX
6	6	Female	Mrs.	Linda	Z	Stein	15	2006-12-09	211 Brook St	Houston	TX
7	7	Female	Ms.	Tiffany	D	Vu	59	1970-07-12	391 Mission St	Gilbert	AZ
8	8	Female	Mrs.	Meghan	J	Stein	20	2003-06-11	45 Humphrey St	Sioux Falls	SD
9	9	Female	Ms.	Joanna	C	Young	53	2005-03-13	299 Market St	Clarksville	TN
10	10	Female	Ms.	Samantha	T	Wilson	36	1992-01-04	267 Mission St	Evansville	IN

In our example, we will be encrypting both the “employee_first_name” and the “employee_last_name” columns. For demonstration purposes, the original first and last name columns were kept just to compare, but in production, this would **not** be done.

employee_first_name	employee_first_name_enc	employee_last_name	employee_last_name_enc
Craig	5BA07D416CACF54A70244B17BD21FB1827B3492B34	Robinson	4ABD7E41651F2A82A47DAE737E157ABCC386B99975EE1A17
Thom	4CBA7345BD408CE581EDE46A8FCD64B76E510EBE	Brown	5AA0735F659A3505AE3B276B9311CFE1EF80148A9C
Julie	52A770416E01BAC82660808665DD824692966C99A6	Jones	52BD724D785F5DBE4C23D8CE4AB3C2D51EA6993A7A
John	52BD7446A6F4C7E1C9C969E65894673E5482377A	Smith	4B8F755C63CD763BE75F7D28B6FB75E5D2BFD39E11
Sam	4BB3712CAC36775A7888B0A3E693211D8DDF85	Jackson	52B37F4378032B3CF545599B7FEB6A33A9E766458F4AD4
Linda	54BB724C6ABE12D7584A789632A35C42D0EA2F41D0	Stein	4BA6794165FBB463C99B462D7049E7AD853DE3BF29
Tiffany	4CBB7A4E6A023C059B373BD9256C2F8C19C78301675182	Vu	4EA7199517AA8B142D85FA99BD095042B714
Meghan	55B37B406A03ADCE2308B6CF5063A5E037A755A40C12	Stein	4B6394165FBB463C99B462D7049E7AD853DE3BF29

Here are the sql statements to test using filter table to filter data based on an employee_dimension table.

----- Vertica shadow table.

```
CREATE TABLE employee_dim_filters(  
    ID IDENTITY(1,1),  
    employee_key INTEGER,  
    filter_label VARCHAR(255),  
    filter_value VARCHAR(255)  
);
```

Sample data in the employee_dim_filters table.

ID	employee_key	filter_label	filter_value
60001	1	employee_first_name	c
50001	1	employee_last_name	rob
60002	2	employee_first_name	t
50002	2	employee_last_name	bro
60003	3	employee_first_name	j
50003	3	employee_last_name	jon
60004	4	employee_first_name	j
50004	4	employee_last_name	smi
60005	5	employee_first_name	s
50005	5	employee_last_name	jac

```

//Populate the filter table to search by first 3 characters of last name
insert into employee_dim_filters (employee_key, filter_label,filter_value) (select
employee_key, 'employee_last_name',lower(substr(employee_last_name,1,3)) from
employee_dimension)
//Populate the filter table to search by first character of first name
insert into employee_dim_filters (employee_key, filter_label,filter_value) (select
employee_key, 'employee_first_name',lower(substr(employee_first_name,1,1)) from
employee_dimension)
//Search by first 3 characters of last name
select * from employee_dim_filters edf ,employee_dimension ed where edf.employee_key
= ed.employee_key and filter_value = 'rob' and filter_label = 'employee_last_name'
//Search by first 1 characters of first name
select * from employee_dim_filters edf ,employee_dimension ed where edf.employee_key
= ed.employee_key and filter_value = 'r' and filter_label = 'employee_first_name'

```

Custom Database User Defined Function Example.

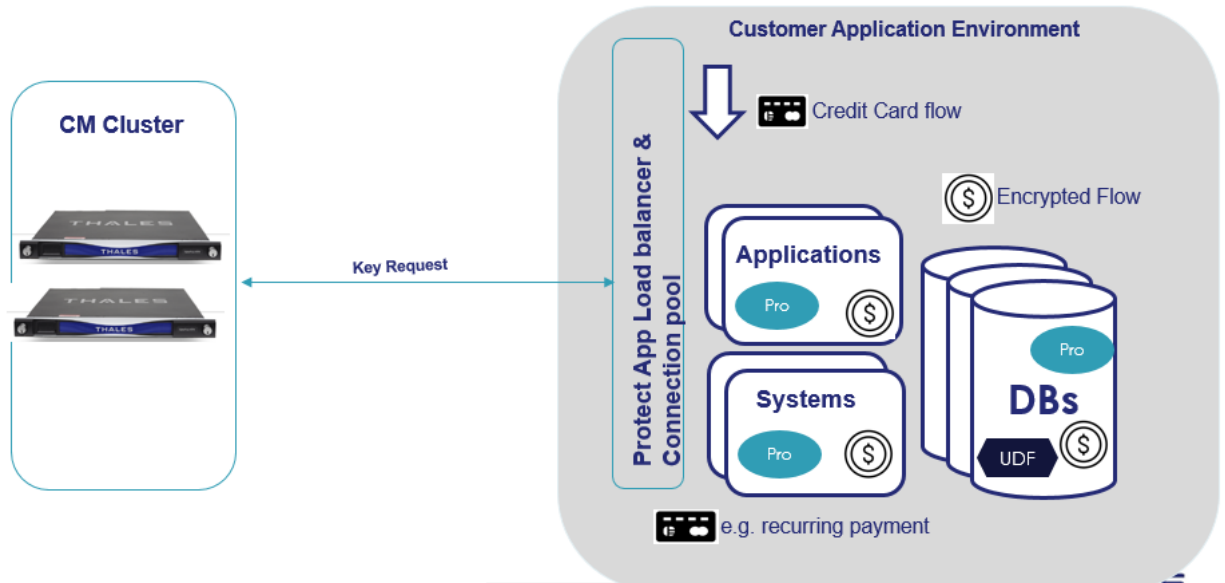
This example uses the CM ProtectApp JCE which is a Java Cryptography Extension provider that enables you to integrate your Java applications with the cryptographic and key-management abilities of the Thales CM. All applications, servlets, or scripts see a conventional JCE interface and issue simple Java-based (JCE) commands to the CM to perform cryptographic operations.

CM ProtectApp JCE enables your Java client to perform cryptographic operations either by requesting that operations be performed on the Thales CM (remote mode) or by caching keys on the client and performing crypto locally (local mode).

Architecture

As you can see below the **application or the database user defined functions** can decrypt data. The diagram shows how the ProtectApp is installed on any machine that needs to decrypt or encrypt data in the database including the database. This example shows how the database UDF was created to support ad-hoc queries for our employee_dimension table. There is **no** need to create any extra tables for this option.

Custom Database User Defined Functions (UDF) Option



The example uses the CipherTrust Manager ProtectAPP API to encrypt the data. To use the ProtectApp JCE SDK it is necessary to:

1. Install the ProtectAPP SDK
2. Set the necessary property file settings as indicated in the documentation and
3. Copy the appropriate protect app jar files to your project directory

At this point, your application can start to make encryption calls. There are many modes or algorithms that can be used such as GCM and Format Preserved Encryption (FPE). The GCM example listed below used the AES/GCM/NoPadding cipher.

Sample UDF Code

This is the code for the encrypt UDF. The decrypt is very similar to what is listed below. This is the syntax for Vertica database.

```
public class ThalesProtectAppEncryptGCM extends ScalarFunctionFactory {

    String username = "admin";
    String password = "yourpwd";
    String keyName = "MyAESEncryptionKey26";
    int authTagLength = Integer.parseInt("128");
    String iv = "6162636465666768696a6b6c";
    String aad = "6162636465666768696a6b6c";
    NAESession session = null;
    NAEKey key = null;
    byte[] ivBytes = IngrianProvider.hex2ByteArray(iv);
    byte[] aadBytes = IngrianProvider.hex2ByteArray(aad);
    String tweakData = null;
    String tweakAlgo = null;
    GCMPParameterSpec param = null;

    public static final String plainTextInp = "Plain text message to be encrypted.";
```

```

@Override
    public void getPrototype(ServerInterface srvInterface, ColumnTypes argTypes,
ColumnTypes returnType) {
        // field name is column to encrypt
        argTypes.addVarchar();
        returnType.addVarchar();
    }

    public class ThalesEncryptGCMData extends ScalarFunction {

        public void setup(ServerInterface srvInterface, SizedColumnTypes
argTypes) {

            session = NAESession.getSession(username, password.toCharArray(),
"hello".toCharArray());
            key = NAEKey.getSecretKey(keyName, session);
            param = new GCMParameterSpec(authTagLength, ivBytes, aadBytes);

            //          srvInterface.log("After  setup");
        }

        public void destroy(ServerInterface srvInterface, SizedColumnTypes
argTypes) {

            //          srvInterface.log("End EncryptDecryptMessage.");
        }

        public void processBlock(ServerInterface srvInterface, BlockReader
arg_reader, BlockWriter res_writer)
            throws UdfException, DestroyInvocation {

            do {
                String results = null;
                String sensitive  = arg_reader.getString(0);

                Cipher encryptCipher;
                try {
                    encryptCipher =
NAECipher.getNAECipherInstance("AES/GCM/NoPadding", "IngrianProvider");
                    encryptCipher.init(Cipher.ENCRYPT_MODE, key, param);

                    byte[] outbuf =
encryptCipher.doFinal(sensitive.getBytes());
                    results = IngrianProvider.byteArray2Hex(outbuf);

                } catch (NoSuchAlgorithmException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                } catch (NoSuchProviderException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                } catch (NoSuchPaddingException e) {
                    // TODO Auto-generated catch block

```

```

        e.printStackTrace();
    } catch (InvalidKeyException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (InvalidAlgorithmParameterException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (IllegalBlockSizeException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (BadPaddingException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    res_writer.setString(results);

    res_writer.next();
} while (arg_reader.next());
}

@Override
public void getReturnType(ServerInterface srvInterface, SizedColumnTypes
argTypes, SizedColumnTypes returnType) {
    returnType.addVarchar((argTypes.getColumnType(0).getStringLength() +
200) * 2, argTypes.getColumnName(0));
}

@Override
public ScalarFunction createScalarFunction(ServerInterface srvInterface) {
    return new ThalesEncryptGCMDData();
}
}

```

Sample SQL Statements

In my example, I was able to use the UDF's to implement the initial encryption with this query. In this case we are using the **thalesencryptgcmdata()** UDF to encrypt data.

```

create table employee_dimension_protectapp as select *,
thalesencryptgcmdata(employee_first_name) employee_first_name_enc,
thalesencryptgcmdata(employee_last_name) employee_last_name_enc from
employee_dimension ed

```

Now to verify the accuracy another query can be run that uses the **thalesdecryptgcmdata()** UDF. In this case I wanted to find everyone that had a last name of 'Robinson' with a first initial of 'C'.

```

select employee_first_name, employee_first_name_enc,
employee_last_name, employee_last_name_enc from employee_dimension_protectapp where
thalesdecryptgcmdata(employee_last_name_enc) = 'Robinson' and
thalesdecryptgcmdata(employee_first_name_enc) like 'C%'

```

employee_first_name	employee_first_name_enc	employee_last_name	employee_last_name_enc
Craig	5BA07D416CACF54A70244B17BD21FB1827B3492B34	Robinson	4ABD7E41651F2A82A47DAE737E157ABCC386B99975EE1A17
Carla	5BB36E446A363B99A6F45DB5ED4FE7073974519A1B	Robinson	4ABD7E41651F2A82A47DAE737E157ABCC386B99975EE1A17

If I want to verify decrypt is working I can decrypt the entire column by running :

```
select employee_first_name, thalesdecryptgcndata(employee_first_name_enc) ,
employee_last_name, thalesdecryptgcndata(employee_last_name_enc) from
employee_dimension_protectapp where thalesdecryptgcndata(employee_last_name_enc) =
'Robinson' and thalesdecryptgcndata(employee_first_name_enc) like 'C%'
```

employee_first_name	thalesdecryptgcndata	employee_last_name	thalesdecryptgcndata
Craig	Craig	Robinson	Robinson
Carla	Carla	Robinson	Robinson

As you can see the decrypt function did provide the original data.

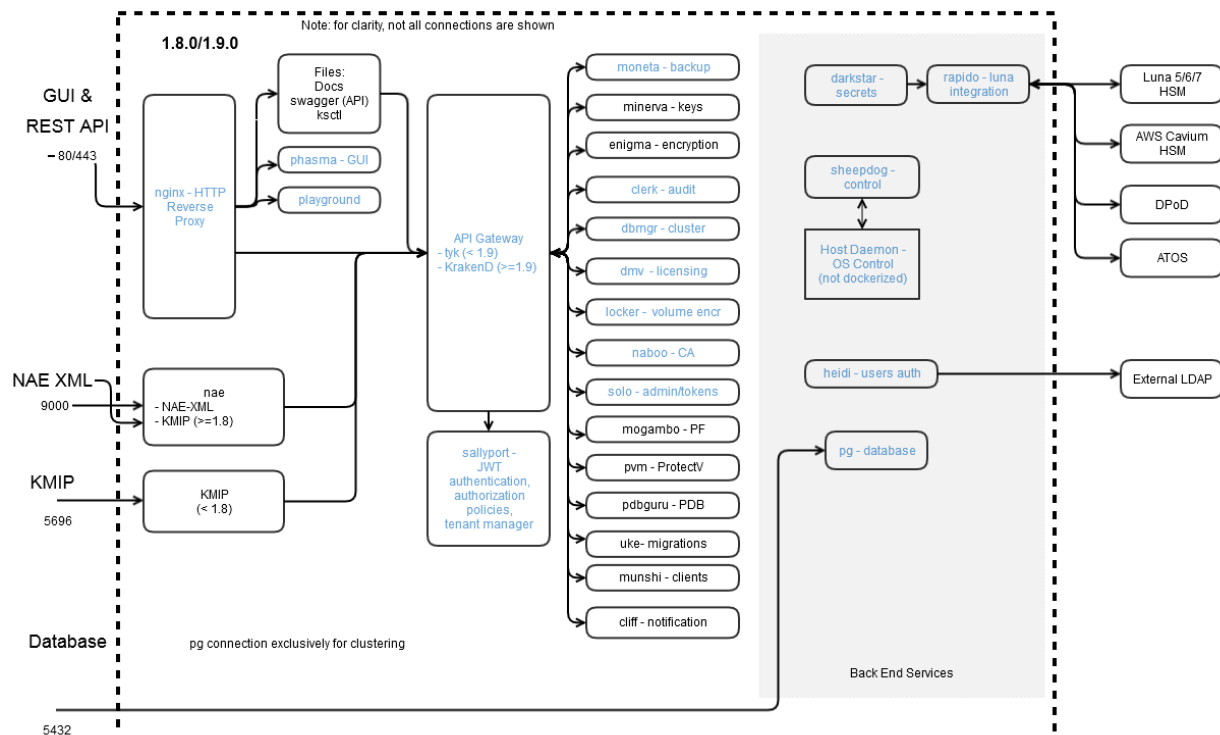
Appendix

Performance Considerations

When writing the Custom User Defined Functions the fastest programming language should be used and in most cases it would be c or c++. This example used Java as a demonstration which works as an out of process application from the database perspective which is safer but also slower. The java pool size of the database should also be optimized depending on your memory requirements of the Custom User Defined Functions and how many concurrent invocations would be expected.

CipherTrust Manager Detailed Architecture.

The diagram below shows CipherTrust Manager is based on micro services using Docker containers.



Enabling local mode for ProtectAPP

There are a few requirements to enable local mode processing for ProtectApp. Having the appropriate key settings and IngrianNAE.properties file settings.

Key Properties

<input checked="" type="checkbox"/> Sign	<input type="checkbox"/> Verify MAC	<input checked="" type="checkbox"/> Wrap Key	<input type="checkbox"/> Key Agreement
<input checked="" type="checkbox"/> Verify	<input checked="" type="checkbox"/> FPE Encrypt	<input checked="" type="checkbox"/> Unwrap Key	<input type="checkbox"/> Certificate Sign
<input checked="" type="checkbox"/> Encrypt	<input checked="" type="checkbox"/> FPE Decrypt	<input checked="" type="checkbox"/> Export Key	<input type="checkbox"/> CRL Sign
<input checked="" type="checkbox"/> Decrypt		<input type="checkbox"/> Derive Key	<input type="checkbox"/> Generate Cryptogram
<input type="checkbox"/> Generate MAC		<input type="checkbox"/> Content Commitment	<input type="checkbox"/> Validate Cryptogram

Key Behaviors

☐ Prevent this key from being deleted
 ☒ Prevent this key from being exported

When running in local mode it is necessary to have the allow Export Key and **Uncheck** the “*Prevent this key from being exported*” or you will encounter this error.

```
Exception in thread "main" java.security.InvalidKeyException:
java.security.InvalidKeyException: 1440: Key is not exportable
    at com.ingrian.security.nae.CipherValidator.a(CipherValidator.java:227)
    at com.ingrian.security.nae.CipherValidator.a(CipherValidator.java:188)
    at
com.ingrian.security.nae.AdvAbstractNAECipher.engineInit(AdvAbstractNAECipher.java:77
7)
    at javax.crypto.Cipher.init(Cipher.java:1394)
    at javax.crypto.Cipher.init(Cipher.java:1327)
    at
AWSMySQLRDSProtectAppExample3.encrypt(AWSMySQLRDSProtectAppExample3.java:300)
    at AWSMySQLRDSProtectAppExample3.main(AWSMySQLRDSProtectAppExample3.java:88)
```

It is also necessary to have the following settings in your IngrianNAE.properties file.

Valid values: yes, no, tcp_ok

Default: no

Recommended: no

Symmetric_Key_Cache_Enabled=tcp_ok

Asymmetric_Key_Cache_Enabled=tcp_ok

[Client Key Caching]

[Symmetric_Key_Cache_Expiry]

Time period since key was cached after which a symmetric key

may be removed from cache. Symmetric_Key_Cache_Expiry can be specified

in any time units (default - seconds)

Setting this value to 0 is equivalent to an infinite timeout.

Note: This field is also applicable to Asymmetric key cache expiry

Default: 43200 (12 hours)

Symmetric_Key_Cache_Expiry=43200