THALES

CipherTrust Manager (CM) Platform

Batch alternatives using application encryption.

Document Version 1.2

Content

PREFACE

This document provides options on how to use CM application encryption to encrypt data in batch mode. Thales CipherTrust Manager does provide a file based encryption capability called CipherTrust Transparent Encryption (CTE) which is used for most use cases when encrypting files. It is important to note that CTE provides both encryption and granular access controls. There are some use cases where it is not economically feasible or due to customer infrastructure limitations, using CTE may not be possible.

DOCUMENTATION VERSION HISTORY

Product/Document Version	Date	Changes
V1.0	09/2021	M. Warner Initial document release
V1.2	9/7/21	Add –a for the Transformation Utility.

ASSUMPTIONS

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

- Java
- Key management
- Data encryption
- Familiarity with REST.

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in this guide or collateral documents that conflict with the definitions or terms in the License Agreement, shall be superseded by the definitions and terms of the License Agreement.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

Use Cases

The goal of this document is to provide some simple examples on how to encrypt or decrypt the sensitive data contained in files. Typically, the methods provided in this document would be used for scenarios where customers have these conditions:

- Small number of files
- Size of files is not large
- Frequency of encryption is not high.

Determining various thresholds for the options above depends on what the customer’s batch window is and the performance of the deployment infrastructure. Testing should be done to determine if batch window times could be achieved. As mentioned above if customers have lots of files, lots of data then the CipherTrust Transparent Encryption (CTE) method would be a better fit. Some of the options listed below may not be appropriate, for example if the data to be encrypted is binary files then doing field level encryption using the CM REST API would not apply.

CipherTrust Manager REST API Requirements

The following requirements are necessary for the REST API.

- KeySecure/Next Gen 1.9.1 or greater for the REST API
- CM 2.6 or greater for the new batch_request json payload enhancement.
- Administrator access to the CM:
    - Security administrators: “Access to the Key Users Group.”

Documentation

- https://ciphertrustmanagerip/playground_v2/api/Alarms

Example Applications.

Architecture

Most implementations will have at least two CipherTrust Managers handling requests. The platform operates as a cluster and it is easy to add more nodes to the cluster if needed. The examples in this case utilized a single CM for testing.

Encrypt an entire file in chunks using the CM REST API Example.

Here is the code to test using CipherTrust Manager REST API to encrypt a file in chunks. You can specify the chunk size a one of the settings in the script. Please note this code is for testing only as only limited testing was performed. The 3rd party libraries jq was to parse json.

```
#!/bin/bash
CT="Content-Type:application/json"
#This script will encrypt and decrypt a file in data chunks using the Thales
CipherTrust Manager REST API.  Chunk size can be altered below.
#Note this example has NOT been extensively tested and should only be used
for proof of concepts.

URL='ipaddressofcm'

#create cm token

PWD="yourpwd"
CT="Content-Type:application/json"
Credentials='{"username":"youruseraccount","password":"'$PWD'"}'
echo $Credentials
AUTH="curl -k -X POST -H $CT -d $Credentials
https://$URL/api/v1/auth/tokens/"
RESPONSE=`$AUTH`
TOKEN=$(echo "$RESPONSE" | jq -r '.jwt')
echo $TOKEN


URL=https://ipaddressofcm/api/v1/crypto
KEY=YourCMKey
IV="VCC3VwxWu6Z6jfQw"
AAD="YXV0aGVudGljYXRl"
CHUNKSIZE=2048
FILE="/root/enc01file/yob2019-100.txt"
ALG="gcm"
BASE=`basename $FILE`
OIV=$IV
if [ ! -e ./$BASE ]
then
cp $FILE .
fi
rm -f ${BASE}-* *-${BASE}-*
split -b $CHUNKSIZE ./$BASE ${BASE}-
NOMORE=0
echo "========================== Encryption =========================="
for i in ${BASE}-*
do
echo "I VALUE IN LOOP ENCYT"
echo $i
```

```
if [ $NOMORE -eq 1 ]
then
break
fi
PTEXT=`cat $i | base64 -w 0`
echo "BASE64 of i"
echo $PTEXT
PAYLOAD=$(jq -n -r \
--arg id $KEY \
--arg iv $IV \
--arg pt "$PTEXT" \
--arg mode $ALG \
--arg aad $AAD \
'{ plaintext: $pt, mode: $mode, id: $id, iv: $iv, aad: $aad  }'
)
RESULT=`echo $PAYLOAD | curl -Ss -X POST -k  -H 'Authorization: Bearer
'"$TOKEN"''  -d @- -H "Content-Type: application/json" -H "Accept:
application/json" $URL/encrypt`
TAG=`echo $RESULT | jq -r '.tag'`
echo "TAG"
echo $TAG
echo "RESULT"
echo $RESULT
CTEXT=`echo $RESULT | jq -r '.ciphertext'`
if [ -z "$NEWIV" ] || [ "$NEWIV" == "null" ]
then
NOMORE=1
fi
IV=$IV
echo -n $CTEXT | base64 -d -w0 > ${i}.enc
done
cat ${BASE}-*.enc > ./${BASE}.enc
split -b $CHUNKSIZE ./${BASE}.enc enc-${BASE}-
IV=$IV
NOMORE=0
echo "========================== Decryption =========================="
for i in enc-${BASE}-*
do
if [ $NOMORE -eq 1 ]
then
break
fi
CTEXT=`cat $i | base64 -w0`
PAYLOAD=$(jq -n -r \
--arg id $KEY \
--arg iv $IV \
--arg ct "$CTEXT" \
--arg mode $ALG \
--arg aad $AAD \
--arg tag $TAG \
'{ ciphertext: $ct, mode: $mode, id: $id, iv: $iv , aad: $aad, tag: $tag}'
)
```

```
RESULT=`echo $PAYLOAD | curl -Ss -X POST -k  -H 'Authorization: Bearer
'"$TOKEN"'' -d @-  -H "Content-Type: application/json"  -H "Accept:
application/json" $URL/decrypt`
NEWIV=`echo $RESULT | jq -r '.iv'`
echo "RESULT DECRYPT"
echo $RESULT
PTEXT=`echo $RESULT | jq -r '.plaintext'`
if [ -z "$NEWIV" ] || [ "$NEWIV" == "null" ]
then
NOMORE=1
fi
IV=$NEWIV
echo -n $PTEXT | base64 -d -w0 > ${i}.dec
done
cat enc-${BASE}-*.dec > ${BASE}.dec
diff ${BASE}.dec ${BASE}
echo diff status = $?
```

Encrypting multiple fields in a single call Example.

This example uses the CM REST API to encrypt particular fields in a file. The capability to encrypt multiple fields in a single call is provided starting with version CM 2.6.

This example shows how to use the same mode and iv for both pieces of data.

https://{{ip}}/api/v1/crypto/encrypt

```
{
  "id": "MyAESEncryptionKey26",
  "iv": "VCC3VwxWu6Z6jfQw",
    "mode": "gcm",
  "aad": "YXV0aGVudGljYXRl",
      "batch_request": [
        {
  "plaintext": "VGhpcyBpcyBhIHRlc3QgdGhpcyBpcyBvbmx5IGEgdGVzdA=="
        },
                {
  "plaintext": "VGhpcyBpcyB0ZXN0aW5nIHRoZSBuZXcgYmF0Y2hfcmVxdWVzdCB3aXRoIENNIDIuNg=="
        }
    ]
}
```

Results

```
{
    "id": "14fc7b3bbc7b4ed482cbbe58e46917dc3159a1987c234c4fb17e583dffbb996d",
    "type": "id",
    "version": 0,
    "mode": "gcm",
    "iv": "VCC3VwxWu6Z6jfQw",
    "aad": "YXV0aGVudGljYXRl",
    "batch_response": [
        {
            "ciphertext": "zEFeZehaQX5EnB9H877PlVFEzQcT2U/NOb8MjZXdBVXbjg==",
            "tag": "iw/8U1WjaNpmg9l/a404/w=="
        },
        {
            "ciphertext": "zEFeZehaQX5R2RhW6aSIwU1F2wcUzxiCNbIBzpyiA1XZjyWo8M4uIfd8LVrA6CAxUQ=
=",
            "tag": "bkgatsYgF7gdL1oeb/zyRw=="
        }
    ]
}
```

Can also have different modes and iv by including them in the batch_request json.

https://{{ip}}/api/v1/crypto/encrypt

```
{
  "id": "MyAESEncryptionKey26",

  "aad": "YXV0aGVudGljYXRl",
      "batch_request": [
        {
  "plaintext": "VGhpcyBpcyBhIHRlc3QgdGhpcyBpcyBvbmx5IGEgdGVzdA==",
    "iv": "VCC3VwxWu6Z6jfQw",
    "mode": "gcm"
        },
                {
  "plaintext": "MTIzNDU2Nzg5MDEyMzQ1Ng==",
  "iv": "MTIzNDU2Nzg5MDEyMzQ1Ng==",
    "mode": "cbc"
        }
    ]
}
```

Results

```
{
    "id": "14fc7b3bbc7b4ed482cbbe58e46917dc3159a1987c234c4fb17e583dffbb996d",
    "type": "id",
    "version": 0,
    "mode": "gcm",
    "aad": "YXV0aGVudGljYXRl",
    "batch_response": [
        {
            "ciphertext": "zEFeZehaQX5EnB9H877PlVFEzQcT2U/NOb8MjZXdBVXbjg==",
            "tag": "iw/8U1WjaNpmg9l/a404/w==",
            "iv": "VCC3VwxWu6Z6jfQw"
        },
        {
            "ciphertext": "h/rCQ9tv34+A1QeHYn53og==",
            "tag": "tWa9thjiTUD+Hoke5oXpRQ==",
            "iv": "MTIzNDU2Nzg5MDEyMzQ1Ng=="
        }
    ]
}
```

Protect app special purposed classes for batch.

The ProtectApp or newly branded Thales CipherTrust Data Protection (CADP) has special classes that can more efficiently encrypt and decrypt files. Listed below is a snippet of the java code example. For working example visit.

https://github.com/thalescpl-io/CipherTrust_Application_Protection/blob/0d1f07a009a4352404fdab216d9e3b032a1a6c0f/crypto/java/FileEncryptionSample.java

```
// create CipherInputStream that will read in data from file and encrypt it
CipherInputStream cis = new CipherInputStream(new FileInputStream(srcName), encryptCipher);
                        FileOutputStream fos = new FileOutputStream(dstName);

// Read the file as blocks of data
byte[] inbuf = new byte[BUFSIZE];
for (int inlen = 0; (inlen = cis.read(inbuf)) != -1;) {
        fos.write(inbuf, 0, inlen);
}
System.out.println("Done encrypting file.  Closing files");

//Decrypt

// create CipherInputStream that will read in data from file and  decrypt it
// This line reads the data from the encrypted file, applies the cipher (now in decrypt mode),
and stores the data in the CipherInputStream.

cis = new CipherInputStream(new FileInputStream(dstName), decryptCipher);
fos = new FileOutputStream(decrName);

for (int inlen = 0; (inlen = cis.read(inbuf)) != -1;) {
        fos.write(inbuf, 0, inlen);
}
System.out.println("Done decrypting file.  Closing files");
```

Protect app transformation utility.

CipherTrust Manager protect app provides a special purposed utility to efficiently process data in file for encryption or decryption. The obvious advantage of this method is that it does not require any code to be written.

To encrypt an entire file the –a parameter can be used. Here is the description of the parameter.

Encrypts or decrypts an entire file. This command treats the entire file as one column. When transformations are performed on an entire file, the input configuration file should consist of a single transformation object. The <IVType> must be set to Column.
For example:

```
<Transformation>
<Type>DECRYPT</Type>
<Key>AES-128</Key>
<Algorithm>AES/CBC/PKCS5Padding</Algorithm>
<IVType>Column</IVType>
<IV>01234567012345670123456701234567</IV>
</Transformation>
```

The output is an encrypted bytes stream - there is no output configuration file. To specify base 16 or 64 encoding, use the -B flag from the command line when running the session.

It is also possible to encrypt particular fields in the file as well. Listed below is an example of the configuration file required and sample command.

```
<ETLIN>
<FormatType>Delimited</FormatType>
<Delimiter>,</Delimiter>
<ColumnList>
<Column>
<Name>LASTNAME</Name>
<MaximumLength>20</MaximumLength>
</Column>
<Column>
<Name>FIRSTNAME</Name>
<MaximumLength>20</MaximumLength>
</Column>
<Column>
<Name>PHONE</Name>
<MaximumLength>10</MaximumLength>
</Column>
<Column>
<Name>ID</Name>
<MaximumLength>8</MaximumLength>
<Transformation>
<Type>Encrypt</Type>
<Key>encKey</Key>
<Algorithm>AES/CBC/PKCS5Padding</Algorithm>
<IVType>Column</IVType>
<IV>E88874641058DF13136D0F653B473900</IV>
<NewName>ID_NEW</NewName>
</Transformation>
</Column>
<Column>
<Name>SSN</Name>
<MaximumLength>11</MaximumLength>
<Transformation>
<Type>Encrypt</Type>
<Key>encKey</Key>
<Algorithm>AES/CBC/PKCS5Padding</Algorithm>
<IVType>Column</IVType>
```

```
<IV>DB1BE0FEC759E2FFE828D8053D955FF9</IV>
<NewName>SSN_NEW</NewName>
</Transformation>
</Column>
<Column>
<Name>CREDITCARD</Name>
<MaximumLength>16</MaximumLength>
<Transformation>
<Type>Encrypt</Type>
<Key>encKey</Key>
<Algorithm>AES/CBC/PKCS5Padding</Algorithm>
<IVType>Column</IVType>
<IV>2FF0A9AE1B019495A998296593974140</IV>
<NewName>CREDITCARD_NEW</NewName>
</Transformation>
</Column>
</ColumnList>
</ETLIN>
```

```
E:\>ingtu.exe -f IngrianNAE.properties -I TUINenc.txt -g -B 16 -t -O
TUOUTenc.txt -u NAE_User1 -p qwerty12345 -i INPUTenc.txt -o OUTPUTenc.txt -e
errorlog.txt
```

Details on parameters.

- The -f option specifies the location of the IngrianNAE.properties file. The file must be in the same
  directory as SafeNet Transform Utility unless a path is included. The properties file must be version 2.0.
- The -I option specifies the input configuration file. The file must be in the same directory as SafeNet Transform Utility unless a path is specified. If the input configuration file is named something other than TUINenc.txt, the command should be adjusted appropriately.
- The -g option instructs SafeNet Transform Utility to create an output configuration file based on the data in the input configuration file. This file is named according to the value passed with -O.
- The -B option instructs SafeNet Transform Utility to use <Base16/> elements when creating the output configuration file. These elements indicate columns to contain encrypted data.
- The -t option instructs SafeNet Transform Utility to execute the command using plain text credentials.
- The -O option specifies the name of the output configuration file. The file is created in the same directory as SafeNet Transform Utility.
- The -u option specifies the NAE user who performs the encrypt operation. The command example above specifies NAE_User1. The specified user must have permission to use the key specified in the <Key> element of the input configuration file.
- The -p option specifies the password for NAE_User1.
- The -i option specifies the input file that contains the data to encrypt. The command example above specifies INPUTenc.txt. The file must be in the same directory as SafeNet Transform Utility unless a path is included.
- The -o option specifies the output file that contains the data encrypted by SafeNet Transform Utility. The command example above specifies OUTPUTenc.txt.
- The -e option specifies the name and location of a log file that SafeNet Transform Utility writes to, in the event that it encounters any bad lines.