

Integrating JCE with Azure Cloud HSM

Table of Contents

Summary	3
Prerequisites	4
System Requirements	4
Creating a User Generated KEK.....	6
Verify JCE Installation and Configuration	9
Appendix	11
Frequently Asked Questions	11
Verify JCE Example Source Code	13
JCE Specifications	14
JCE Sample Template.....	14
Generating Keys and Pairs	17
Supported Key Types	17
Supported Secure Random Algorithms	18
Supported Symmetric (Secret) Keys	20
Supported Asymmetric Keys	23
Making a Session Key Persistent and Deleting a Key in the HSM	26
Using Message Digests, MACs and HMACs.....	27
Supported Message Digests	27

Supported MACs and HMACs	29
Cipher Operations (Symmetric and Asymmetric Keys)	33
Methods to Define the Cipher Class	33
Supported Maximum Encryption and Decryption Lengths	33
Symmetric Key Cipher Operations	34
Asymmetric Key Cipher Operations	38
Transferring Keys	40
Wrapping and Unwrapping Keys	41
Importing Keys	50
Digital Signatures	55
Supported Digital Signatures	57
Key Exchange Using Diffie-Hellman	64
Managing Azure Cloud HSM Keystore	67
About the keytool Utility	68
Add Azure Cloud HSM JCE provider to Java runtime security configuration.	69
Update path and environment variables for keytool use.	70
Generating a Key Pair with keytool	70
Listing a Key with keytool	71
Listing all Keys with keytool	71
Replacing a Self-Signed Certificate with a Certificate from a CA	72
Signing and Verifying a Jar File in the LSKS	78

Key Factories	79
Supported Features and Requirements	79
Key Factory Examples	80
How to list all Algorithms in Azure Cloud HSM JCE provider	85
Notes on Azure Cloud HSM AES-GCM Implementation	89
Notes on Azure Cloud HSM AES-CCM Implementation	92

Summary

JCE (Java Cryptography Extension) and JCA (Java Cryptography Architecture) are components of the Java platform that provide a framework for implementing cryptographic operations in Java applications. In Java, JCE is now integrated with JCA. The classes and interfaces of JCE are now part of JCA. The Azure Cloud HSM JCE provider is compatible with JCA. Throughout this documentation, we will use the term "Azure Cloud HSM JCE provider" interchangeably with "JCE" to refer to JCE/JCA functionality. It is essential to have the Azure Cloud HSM Client SDK installed and configured on the host responsible for cryptographic operations in your Java applications.

JCE provides a comprehensive and extensible platform for handling cryptographic operations in Java applications, ensuring that developers can build secure and robust systems. They enable the use of standard cryptographic algorithms and services, making it easier for developers to implement secure communication, data integrity, and confidentiality in their Java applications.

Common use cases for JCE include secure communication over networks, digital signatures, encryption and decryption of sensitive data, secure key management, and more. Developers can leverage these APIs to build secure applications that adhere to industry-standard cryptographic practices.

Important Note: Azure Cloud HSM JCE does not support Windows. Azure Cloud HSM JCE supports Linux only.

Prerequisites

The following prerequisites are required to support the Azure Cloud HSM JCE provider. Please reference the Azure Cloud HSM Onboarding Guide for SDK Installation and Azure Cloud HSM configuration if you have not completed your HSM deployment.

System Requirements

- Supported Operating Systems: Ubuntu 20.04, Ubuntu 22.04, RHEL 7, RHEL 8, CBL Mariner 2
- Azure Cloud HSM resource has been deployed, initialized, and configured.
- [Azure Cloud HSM Client SDK](#)
- Copy of partition owner certificate “PO.crt” on host/signing server.
- Known address of your HSM “hsm1.chsm-<resourcename>-<uniquestring>.privatelink.cloudhsm.azure.net”.
- Creation of User-Generated KEK
- Knowledge of Crypto User credentials

Customers can download the Azure Cloud HSM SDK and Client Tools from GitHub:

@ [microsoft/MicrosoftAzureCloudHSM: Azure Cloud HSM SDK](#)

1. **Install OpenJDK:** To install OpenJDK on a Linux system, you'll need to execute the following commands. The exact command may vary depending on the version of Java you require.

- **Install OpenJDK 8 if using JCE for Java 8**

```
sudo apt update
sudo apt install -y openjdk-8-jdk
```

- **Install OpenJDK 11 if using JCE for Java 11**

```
sudo apt update
sudo apt install -y openjdk-11-jdk
```

2. **Validate OpenJDK is Installed.** You can check if OpenJDK is installed on a Linux system by opening a terminal and entering the following command. If OpenJDK is installed, this command will display the version number of OpenJDK that is currently installed on your system. If OpenJDK is not installed, the command will likely result in an error message.

```
java -version
```

3. **Set Environment Variables:** Add the following system environment variables to your Linux Server. You may also choose to set permanent environment variables in Linux. You can typically modify configuration files specific to your shell (bash, zsh, etc) or make system-wide changes in profile scripts. You will need to update your environment variables to reflect the correct SDK version running.

Recommended (Permanent Environment Variables): For production, make environment variables persistent across terminal sessions by adding them to your shell's configuration file. This example uses Bash.

- a. From the terminal edit the `.bashrc` file (for non-login shells) or `.bash_profile` (for login shells). You can use any text editor.

```
vim ~/.bashrc
```

or

```
vim ~/.bash_profile
```

- b. Add the following lines to the end of the file

Additional Environment Variables for OpenJDK 8

```
export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-amd64/
```

```
export CLASSPATH=/opt/azurecloudhsm/jce/jce-1.8.0 /
```

```
export LD_LIBRARY_PATH=/opt/azurecloudhsm/jce/jce-1.8.0/
```

Additional Environment Variables for OpenJDK 11

```
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64/
```

```
export CLASSPATH=/opt/azurecloudhsm/jce/jce-11/
```

```
export LD_LIBRARY_PATH=/opt/azurecloudhsm/jce/jce-11/
```

- c. Save the file and exit the editor

- d. To apply the changes immediately, run the following.

```
source ~/.bashrc
```

or

```
source ~/.bash_profile
```

Manual (Testing Purposes Only):

The following manual approach is temporary and will not persist after the terminal session ends. It should only be used for testing purposes.

Additional Environment Variables for OpenJDK 8

```
export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-amd64/  
export CLASSPATH=/opt/azurecloudhsm/jce/jce-1.8.0 /  
export LD_LIBRARY_PATH=/opt/azurecloudhsm/jce/jce-1.8.0/
```

Additional Environment Variables for OpenJDK 11

```
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64/  
export CLASSPATH=/opt/azurecloudhsm/jce/jce-11/  
export LD_LIBRARY_PATH=/opt/azurecloudhsm/jce/jce-11/
```

Creating a User Generated KEK

Creating a User-Generated KEK is necessary only for customers who require Key Import support or integration with JCE and OpenSSL. If you don't need support for those specific use cases, you can choose to skip the creation of a KEK.

1. Creating a user KEK handle involves initial steps where customers execute the `azcloudhsm_client`, connect to their Cloud HSM using the `azcloudhsm_util`, and subsequently log in to their HSM. The `azcloudhsm_client` must be running for `azcloudhsm_util` to execute. The provided example illustrates the process of generating a user KEK and obtaining its handle.

***Important Note:** To ensure the proper functioning of the Azure Cloud HSM SDK, customers need to create a user KEK. The functionality of OpenSSL, JCE, and other features within Azure Cloud HSM relies on the existence of a user-generated KEK. Failure to generate a user KEK will result in operational issues.*

***Important Note:** Please keep track of your Key Handle ID generated. You'll require this Key Handle ID to finalize the process. If you need to perform Key Import (key wrap/unwrap), ensure your user KEK is extractable and designated as trusted. Failure to designate your user KEK as extractable and trusted prior to attempting to perform key import will result in an exception (e.g. invalid attribute. attribute should be set for User KEK.)*

Option 1: User KEK with extractable and trusted. Required for customers that need key import to be operational in addition to JCE, OpenSSL and other utilities to be operational. The example below we're setting `-l` (key label) as `userkek`, `-t` (key type) and `-s` (key size) as AES, 32 bytes.

```
./azcloudhsm_util  
loginHSM -u CU -s cu1 -p user1234  
genSymKey -l userkek -t 31 -s 32 -wrap_with_trusted 1
```

```
Command: genSymKey -l userkek -t 31 -s 32 -wrap_with_trusted 1

Cfm3GenerateSymmetricKey returned: 0x00 : HSM Return: SUCCESS

Symmetric Key Created. Key Handle: 262259

Cluster Status:
Node id 1 status: 0x00000000 : HSM Return: SUCCESS
Node id 2 status: 0x00000000 : HSM Return: SUCCESS
Node id 3 status: 0x00000000 : HSM Return: SUCCESS
```

Option 2: User KEK with non-extractable but trusted. Required for customers that do not need key import but require JCE, OpenSSL and other utilities to be operational. The example below we're setting -l (key label) as userkek, -t (key type) and -s (key size) as AES, 32 bytes and setting the key as non-extractable.

```
./azcloudhsm_util
loginHSM -u CU -s cu1 -p user1234
genSymKey -l userkek -t 31 -s 32 -nex
```

```
Command: loginHSM -u CU -s cu1 -p user1234

Cfm3LoginHSM returned: 0x00 : HSM Return: SUCCESS

Cluster Status:
Node id 1 status: 0x00000000 : HSM Return: SUCCESS
Node id 2 status: 0x00000000 : HSM Return: SUCCESS
Node id 3 status: 0x00000000 : HSM Return: SUCCESS

Command: genSymKey -l userkek -t 31 -s 32 -nex

Cfm3GenerateSymmetricKey returned: 0x00 : HSM Return: SUCCESS

Symmetric Key Created. Key Handle: 262150

Cluster Status:
Node id 1 status: 0x00000000 : HSM Return: SUCCESS
Node id 2 status: 0x00000000 : HSM Return: SUCCESS
Node id 3 status: 0x00000000 : HSM Return: SUCCESS
```

2. After the key has been generated, customers will need to set the correct attributes on their key so that it can be used as a KEK. Firstly, you will need to end your azcloudhsm_util session. Customers will then run the management util and login as the Crypto Officer. You must be logged in as the Crypto Officer when setting the attributes on the Key.

```
sudo ./azcloudhsm_mgmt_util ./azcloudhsm_resource.cfg  
loginHSM CO admin adminpassword
```

3. Upon assuming the role of the Crypto Officer and logging in, proceed to configure the attributes of the previously generated key. Obtain the key handle from the previous step and execute the following command to establish its attributes, utilizing your KeyHandleID.

Usage: setAttribute <KeyHandle> <AttributeID> <AttributeValue>. AttributeID 134 sets OBJ_ATTR_TRUSTED. AttributeValue 1 sets OBJ_ATTR_AUTH_FACTOR which is 1FA. Customers must use 134 1. No other values are supported as we only support 1FA.

```
setAttribute <KeyHandleID> 134 1
```

```
cloudmgmt>setAttribute 262150 134 1  
*****CAUTION*****  
This is a CRITICAL operation, should be done on all nodes in the  
cluster. Cav server does NOT synchronize these changes with the  
nodes on which this operation is not executed or failed, please  
ensure this operation is executed on all nodes in the cluster.  
*****  
  
Do you want to continue(y/n)?y  
setAttribute success on server 0(10.0.2.4)  
setAttribute success on server 1(10.0.2.5)  
setAttribute success on server 2(10.0.2.6)  
cloudmgmt>
```

4. After configuring the attributes for the generated key, you can utilize it as a KEK (Key Encryption Key) by modifying the USER_KEK_HANDLE in your azcloudhsm_application.cfg file with the corresponding KeyHandleID.

```
DAEMON_ID=1  
SOCKET_TYPE=UNIXSOCKET  
PORT=1111  
USER_KEK_HANDLE=262150
```


Verify JCE Installation and Configuration

To verify JCE installation and configuration with Azure Cloud HSM we are going to perform an encrypt and decrypt operation using the `azcloudhsm_openssl` engine for Azure Cloud HSM.

***Important Note:** For your Java application to be successfully executed the `azcloudhsm_application.cfg` file and `*.dependencies.jar` file must exist in the same location as your Java application and the `azcloudhsm_client` must be running. If the `azcloudhsm_client` utility is not running, you will receive a failed socket connection.*

You may need to copy the `azcloudhsm_application.cfg` file from the default location below to the location of the Java application you are running. Follow the steps below to validate JCE is configured and operational with Azure Cloud HSM. In this example we are copying dependency files and executing from the home directory.

Verifying JCE Configuration:

1. Copy the `azcloudhsm_application.cfg` to your home directory.

```
cd /opt/azurecloudhsm/bin
cp ./azcloudhsm_application.cfg ~/azcloudhsm_application.cfg
```

2. Copy the JAR to your home directory.

- **OpenJDK 8**

```
cd /opt/azurecloudhsm/jce/jce-1.8.0
cp ./*.jar ~/azure-cloud-hsm-jca-1.8.0-jar-with-dependencies.jar
```

- **OpenJDK 11**

```
cd /opt/azurecloudhsm/jce/jce-11
cp ./*.jar ~/azure-cloud-hsm-jca-11-jar-with-dependencies.jar
```

3. Start the client daemon if it is not running.

It's recommended for production to run the client daemon as a service. If you installed the Azure Cloud HSM SDK using deb or rpm, the client is not configured automatically to run as a service. The SDK during installation includes a service unit file under `/etc/systemd/system/azure-cloud-hsm.service`. To enable `azcloudhsm_client` to run as a service you will need to use the predefined `azure-cloud-hsm.service` file. You will then need to reload the Systemd configuration and enable the service to ensure its continuously running. For details on how to configure the client daemon to run as a service please reference the Azure Cloud HSM onboarding guide.

The following steps below demonstrate two different ways to manually run the client daemon for testing OpenSSL integration.

```
cd /opt/azurecloudhsm/bin  
sudo ./azcloudhsm_client azcloudhsm_resource.cfg
```

You may also choose as an option to run the client daemon in the background using the following command or run the client daemon as a service to ensure it is always running.

```
sudo ./azcloudhsm_client azcloudhsm_resource.cfg > /dev/null 2>&1 &
```

4. **Check Number Existing Keys on HSM.** For a new Cloud HSM “total number of keys present” should be 1 as you created a user generated KEK prior to this step. If you have created other keys prior to this document, make note of the total number of keys so that you can validate the increment in the HSM upon next steps captured within this JCE integration guide.

```
sudo ./azcloudhsm_util singlecmd loginHSM -u CU -s cu1 -p user1234 findKey
```

3. **Create Sample Java Application** To verify JCE we will create a VerifyJCE.java file and copy the demo example Java source code from this document found under the appendix section into our java source file.

```
cd ~  
sudo vim VerifyJCE.java
```

5. **Build the Sample Java Application.** To build the JCE validation application we will execute the following command corresponding to the OpenJDK we require.

- **OpenJDK 8**

```
javac -cp azure-cloud-hsm-jca-1.8.0-jar-with-dependencies.jar: VerifyJCE.java
```

- **OpenJDK 11**

```
javac -cp azure-cloud-hsm-jca-11-jar-with-dependencies.jar: VerifyJCE.java
```

6. **Run the Sample Java Application.** To run the JCE validation application we will execute the following command corresponding to the OpenJDK we require.

- **OpenJDK 8**

```
java -cp .:azure-cloud-hsm-jca-1.8.0-jar-with-dependencies.jar: VerifyJCE
```

- **OpenJDK 11**

```
java -cp .:azure-cloud-hsm-jca-11-jar-with-dependencies.jar: VerifyJCE
```

```
chsmVMAdmin@myLinuxVM: ~$ java -cp .:azure-cloud-hsm-jca-11-jar-with-dependencies.jar: VerifyJCE
ERROR StatusLogger Log4j2 could not find a logging implementation. Please add log4j-core to the classpath. Using SimpleLogger to log to the console...
*****Generating AES Key *****
com.cavium.key.CaviumAESKey@4002a
```

7. **Validate New Key was created on HSM.** You should see “total number of keys present” has incremented by 1 and a new key handle id was created.

```
cd /opt/azurecloudhsm/bin
sudo ./azcloudhsm_util singlecmd loginHSM -u CU -s cu1 -p user1234 findKey
```

Appendix

Frequently Asked Questions

- **Does Azure Cloud HSM JCE support Windows?**

No. Azure Cloud HSM JCE supports Linux (Ubuntu 20.04, Ubuntu 22.04, RHEL 7, RHEL 8, CBL Mariner 2) only.

- **How do I not combine Azure Cloud HSM environment variables to support both OpenSSL and JCE?**

To combine the two LD_LIBRARY_PATH environment variables into one, you should concatenate them using a colon : as a separator.

- **OpenJDK 8**

```
export LD_LIBRARY_PATH=/opt/azurecloudhsm/lib64:/opt/azurecloudhsm/jce/jce-1.8.0:$LD_LIBRARY_PATH
```

- **OpenJDK 11**

```
export LD_LIBRARY_PATH=/opt/azurecloudhsm/lib64:/opt/azurecloudhsm/jce/jce-11:$LD_LIBRARY_PATH
```

- **Why am I getting keytool error message provider "com.cavium.provider.CaviumProvider" not found**

This error message suggests that the Azure Cloud HSM JCE provider is not referenced in your Java runtime's security configuration. To resolve this, you need to modify the java.security file located in the \$JAVA_HOME/conf/security directory. For example, the directory path might be /usr/lib/jvm/java-11-openjdk-amd64/conf/security. Add the following line to the file, replacing 'N' with the next available

provider number to include the Azure Cloud HSM JCE provider. Once you made changes to the java.security file, restart your Java process to apply the changes.

```
security.provider.N=com.cavium.provider.CaviumProvider
```

- **Why am I getting error message azcloudhsm_application.cfg is not present?**

To ensure the successful execution of your OpenSSL application, two conditions must be met. Firstly, the azcloudhsm_application.cfg file must be present in the same directory as your OpenSSL application. Secondly, the azcloudhsm_client should be running. If the azcloudhsm_client is not active, you will encounter a failed socket connection. Likewise, the absence of the azcloudhsm_application.cfg file will result in an error message. Depending on the execution location of your OpenSSL application, you may need to copy the azcloudhsm_application.cfg file from the Azure Cloud HSM SDK directory to the directory where you are running the OpenSSL application.

- **Why am I getting error messages that could not find credentials to login to the HSM?**

The error indicates a challenge in locating or authenticating the credentials required for logging into Cloud HSM. When utilizing the samples in this documentation or incorporating the Azure Cloud HSM JCE provider into your applications, it is essential to explicitly log in with credentials, as illustrated in the example below.

```
// Explicit Login with Credentials
LoginManager lm = LoginManager.getInstance();
lm.login("PARTITION_1", "cu1", "user1234");

// Logout of the HSM
lm.logout();
```

- **Why am I getting error message attribute with value 0 for this operation? Should attribute be set for User KEK?**

The error indicates that a user-generated KEK was not created and designated as extractable and wrap_with_trusted set to 1. Please see [creating a user-generated KEK](#) for appropriate settings when performing Key Import / Keywrap / Keyunwrap operations.

- **Why am I getting error message daemon socket connection error, API login failed?**

The error indicates a challenge in locating or authenticating the credentials required for logging into Cloud HSM. When utilizing the samples in this documentation or incorporating the Azure Cloud HSM JCE provider into your applications, it is essential to explicitly log in with credentials, as illustrated in the example below.

Verify JCE Example Source Code

The following source code below is used to create the VerifyJCE.java source file mentioned in the Verify JCE Configuration with Azure Cloud HSM section of this document.

```
import java.security.Security;
import java.util.UUID;

import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;

import com.cavium.cfm2.*;
import com.cavium.key.parameter.CaviumAESKeyGenParameterSpec;
import com.cavium.provider.CaviumProvider;

public class VerifyJCE {

    public static SecretKey generateAESKeyPair() throws Exception
    {
        // Create instance of Key Generator
        KeyGenerator kg = KeyGenerator.getInstance("AES", "Cavium");

        // Initialize it with args of wanted Key
        kg.init(new CaviumAESKeyGenParameterSpec(256, UUID.randomUUID().toString(), true,true));

        // Generate and return Key
        return kg.generateKey();
    }

    public static void main(final String[] args) throws Exception {
        // Adding the Azure Cloud HSM JCE Provider Globally at Runtime
        try {
            Security.addProvider(new CaviumProvider());
        }
    }
}
```

```

    } catch (Exception ex) {
        System.out.println(ex);
        return;
    }

    // Create instance of login manager for logging into the HSM
    LoginManager lm = LoginManager.getInstance();
    try {
        // Explicit Login with credentials
        lm.login("PARTITION_1", "cu1", "user1234");
        SecretKey aeskp;

        // Create AES Key
        System.out.println("*****Generating AES Key *****");
        aeskp=generateAESKeyPair();

        //Print key object toString()
        System.out.println(aeskp.toString());
    } catch (Exception e) {
        System.out.println(e);
    } finally {
        // Logout of the HSM
        lm.logout();
    }
}
}
}

```

JCE Specifications

JCE Sample Template

Below is a boilerplate template for customers that want to run JAVA samples against Azure Cloud HSM.

```
import java.security.Provider;
import java.security.Provider.Service;
import com.cavium.provider.CaviumProvider;
import com.cavium.cfm2.*;
import com.cavium.cfm2.CFM2Exception;
import com.cavium.cfm2.LoginManager;
import com.cavium.cfm2.Util;
import com.cavium.cfm2.ImportKey;
import com.cavium.key.CaviumAESKey;
import com.cavium.key.CaviumKey;
import com.cavium.key.CaviumKeyAttributes;
import com.cavium.key.CaviumRSAPrivateKey;
import com.cavium.key.CaviumRSAPublicKey;
import com.cavium.key.CaviumKeyAttributes;
import com.cavium.key.CaviumECPrivateKey;
import com.cavium.key.CaviumECPublicKey;
import com.cavium.key.parameter.CaviumRSAKeyGenParameterSpec;
import com.cavium.key.parameter.CaviumAESKeyGenParameterSpec;
import com.cavium.key.parameter.CaviumKeyGenAlgorithmParameterSpec;
import com.cavium.key.parameter.CaviumECGenParameterSpec;
import com.cavium.key.parameter.CaviumDESKeyGenParameterSpec;
import com.cavium.key.parameter.CaviumGenericSecretKeyGenParameterSpec;
import java.security.SignatureException;
import java.util.*;
import java.util.Arrays;
import java.util.UUID;
import java.util.Random;
import java.util.Scanner;
import java.util.Enumeraation;
import java.security.*;
import java.security.Key;
import java.security.KeyPairGenerator; //Java 8
```

```

import java.security.KeyPair;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.Signature;
import java.security.Security;
import java.security.SecureRandom;
import java.security.interfaces.ECPrivateKey;
import java.security.interfaces.ECPublicKey;
import java.security.spec.ECGenParameterSpec;
import java.security.spec.ECPublicKeySpec;
import java.math.BigInteger;
import javax.crypto.*;
import javax.crypto.KeyAgreement;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.Cipher;
import javax.crypto.spec.IvParameterSpec;

public class CloudHSMJCETemplate {

    // Add Crypto Function Here

    public static void main(final String[] args) throws Exception {
        // Adding the Azure Cloud HSM JCE Provider Globally at Runtime
        try {
            Security.addProvider(new CaviumProvider());
        } catch (Exception ex) {
            System.out.println(ex);
            return;
        }

        // Create instance of login manager for logging into the HSM

```



```

LoginManager lm = LoginManager.getInstance();
try {
    // Explicit Login with credentials
    lm.login("PARTITION_1","cu1", "user1234");

    // Add Code Here
    // Call Crypto Function
    // End of Code Block

} catch (Exception e) {
    System.out.println(e);
} finally {
    // Logout of the HSM
    lm.logout();
}
}
}

```

Generating Keys and Pairs

This section details the different key types the Azure Cloud HSM JCE provider supports and provides code examples for generating each type of key. In addition, it describes and provides examples for generating secure random numbers with algorithms supported by the Azure Cloud HSM JCE provider.

Supported Key Types

The Azure Cloud HSM JCE provider includes support for the following key types and algorithms.

Key Type	Supported Size (bits)
AES	128, 192, 256 (default)
RSA key pairs	1024 (only in non-FIPS mode), 2048, 2304, 2560, 2816, 3072, 3328, 3584, 3840, 4096 (default)

EC key pairs	NIST P256 NIST P384 (No default value because the key must be constructed using a standard name.)
XDH key pairs	X25519
Triple DES (DESede)	192
ECC curve	secp256k1
HMAC	HmacSHA1, HmacSHA224, HmacSHA256, HmacSHA384, HmacSHA512

***Important Note:** For HMAC generation, the key size provided by the user will be rounded up to the nearest multiple of 8 bits, and the generated key algorithm name will be `GenericSecret`.*

Supported Secure Random Algorithms

The Azure Cloud HSM JCE provider includes support for the following two types of secure random algorithms.

Key Type	Supported Size (bits)
AES-CTR-DRBG (FIPS compliant)	The AES-CTR-DRBG secure random algorithm can generate up to 8000 bytes of random number within the HSM. <code>SecureRandom sr = SecureRandom.getInstance("AES-CTR-DRBG");</code>
X9-17-RNG (non-FIPS compliant)	The X9-17-RNG secure random algorithm can generate up to 16000 bytes of random number within the HSM <code>SecureRandom sr = SecureRandom.getInstance("X9-17-RNG");</code>

The Azure Cloud HSM JCE provider uses X9-17-RNG for ECDSA signatures and random IV generation in cipher classes.

***Important Note:** Secure Random does not support setting an initial seed for the generator itself. Calling the `setSeed()` method on the provider will throw an `UnsupportedOperationException` exception. The Secure Random implementation can be used to generate seed bytes for other random number generators using the `SecureRandom.generateSeed` method.*

Example code for generating a secure random number:

```
/**
 * Example code for generating a secure random number.
 */
```

```

public class RandomNumberGenerator {

    public static void main(final String[] args) throws Exception {
        try {
            Security.addProvider(new com.cavium.provider.CaviumProvider());
        } catch (Exception ex) {
            System.out.println(ex);
            return;
        }

        LoginManager lm = LoginManager.getInstance();
        lm.login("PARTITION_1", "cu1", "user1234");

        SecureRandom sr = SecureRandom.getInstance("AES-CTR-DRBG");
        byte[] bytes = new byte[16];
        sr.nextBytes(bytes);
        byte[] seed = sr.generateSeed(16);

        for (int element: seed) {
            System.out.println(element);
        }
        lm.logout();
    }
}

```

The examples continued below use a key specification approach for key generation. In addition to standard parameters, each Azure Cloud HSM JCE generated key can have additional parameters that specify:

- A key label that can be used to search for keys from the HSM-based KeyStore.
- An *isExtractable* flag indicating if the key can be exported from the hardware.
- An *isPersistent* flag indicating if the key can be stored in the KeyStore after the application has ended its session.

See the classes *CaviumKey*, *CaviumKeyGenAlgorithmParameterSpec*, *CaviumAESKeyGenParameterSpec*, and *CaviumRSAKeyGenParameterSpec* for details on how the parameters are specified.

Supported Symmetric (Secret) Keys

The Azure Cloud HSM JCE provider includes support for the following symmetric keys.

Key Type	Supported Size (bits)
AES	128, 192, 256 (default)
Triple DES (DESEde)	192
Generic Secret (HMACSHA1/224/256/384/512)	<= 6400 bits

Generating an AES Key

The following example uses the *CaviumAESKeyGenParameterSpec* to generate an AES key:

```
/**
 * Example code for generating an AES key.
 */
public class KeyGenerateAES {
    public static SecretKey generateAESKeyPair() throws Exception {
        KeyGenerator kg = KeyGenerator.getInstance("AES", "Cavium");
        kg.init(new CaviumAESKeyGenParameterSpec(256,
            UUID.randomUUID().toString(), true, true));
        return kg.generateKey();
    }

    public static void main(final String[] args) throws Exception {
        try {
            Security.addProvider(new com.cavium.provider.CaviumProvider());
        } catch (Exception ex) {
            System.out.println(ex);
        }
        return;
    }
}
```

```

LoginManager lm = LoginManager.getInstance();
lm.login("PARTITION_1","cu1", "user1234");
SecretKey aeskp;
System.out.println("*****Generating AES Key *****");
aeskp=generateAESKeyPair();
lm.logout();
}
}

```

The following spec must be sent to init:

```
new CaviumAESKeyGenParameterSpec(256, UUID.randomUUID().toString(), true,true));
```

This example initializes the key spec with the following parameters:

keySize	Size of the key in bits. In this example, 256.
label	Key label. In this example, a random string.
extractable	Boolean value, where true means the key will be extractable
Persistent	Boolean value, where true means the key will be persistent in the HSM

Generating a DES Key

The following sample code generates a DES key using the CaviumDESKeyGenParameterSpec.

```

/**
 * Example code for generating a DES key.
 */
public class KeyGenerateDES {
    public static void main(final String[] args) throws Exception {
        try {
            Security.addProvider(new com.cavium.provider.CaviumProvider());
        } catch (Exception ex) {
            System.out.println(ex);
        }
        return;
    }
}

```

```

}

LoginManager lm = LoginManager.getInstance();
lm.login("PARTITION_1","cu1", "user1234");
System.out.println("*****Generating DES Key *****");
KeyGenerator kg = KeyGenerator.getInstance("DESede", "Cavium");
kg.init(new CaviumDESKeyGenParameterSpec(192, "destest", true, true));
kg.generateKey();
lm.logout();
}
}

```

This example initializes the key spec with the following parameters:

keySize	Size of the key in bits. 168 and 192-bit keys are supported; here the key size is 192.
label	Key label. Here destest.
extractable	Boolean value, where true means the key will be extractable
Persistent	Boolean value, where true means the key will be persistent in the HSM

If any of the parameters are invalid, an InvalidAlgorithmParameterException is thrown.

Generating a Generic Secret Key

The following example illustrates how to create a generic secret key using HMACSHA1 and the CaviumGenericSecretKeyGenParameterSpec:

```

/**
 * Example code for generating a secret key.
 */
public class KeyGenerateSecret {
    public static void main(final String[] args) throws Exception {
        try {
            Security.addProvider(new com.cavium.provider.CaviumProvider());
        } catch (Exception ex) {
            System.out.println(ex);
        }
        return;
    }
}

```

```

}

LoginManager lm = LoginManager.getInstance();
lm.login("PARTITION_1","cu1", "user1234");
System.out.println("*****Generating Secret Key *****");
KeyGenerator kg = KeyGenerator.getInstance("hmacsha1", "Cavium");
kg.init(new CaviumGenericSecretKeyGenParameterSpec(192, "testGetSpec", true, true));
kg.generateKey();
lm.logout();
}
}

```

This example initializes CaviumGenericSecretKeyGenParameterSpec with the following parameters:

keySize	Size of the key in bits. 168 and 192-bit keys are supported; here the key size is 192.
label	Key label. Here testGetSpec.
extractable	Boolean value, where true means the key will be extractable.
Persistent	Boolean value, where true means the key will be persistent in the HSM.

If any of the parameters are invalid, an InvalidAlgorithmParameterException is thrown.

Supported Asymmetric Keys

The Azure Cloud HSM JCE provider includes support for the following asymmetric keys.

Key Type	Supported Size (bits)
RSA	1024 (only in non-FIPS mode) 2048, 2304, 2560, 2816, 3072, 3328, 3584, 3840, 4096 bits in size
ECDSA	NIST P256, NIST P384 secp256k1

Generating an RSA Key Pair

The following code provides an example of generating an RSA key pair:

```

/**
 * Example code for generating an RSA key pair.
 */
public class KeyPairGenerateRSA {
    public static KeyPair generateRSAKeyPair() throws Exception {
        KeyPairGenerator keyPair = KeyPairGenerator.getInstance("RSA","Cavium");
        keyPair.initialize(new CaviumRSAKeyGenParameterSpec(2048,new BigInteger("65537"),"publickey", "privatekey",true,true));
        return keyPair.generateKeyPair();
    }

    public static void main(final String[] args) throws Exception {
        try {
            Security.addProvider(new com.cavium.provider.CaviumProvider());
        } catch (Exception ex) {
            System.out.println(ex);
            return;
        }

        LoginManager lm = LoginManager.getInstance();
        lm.login("PARTITION_1","cu1", "user1234");
        KeyPair kp;
        System.out.println("*****Generating RSA Key Pair*****");
        kp=generateRSAKeyPair();
        PrivateKey pk = kp.getPrivate();
        PublicKey pubkey = kp.getPublic();
        lm.logout();
    }
}

```

Important Note: If you generate a key with modulus 1024 when the HSM is in FIPS mode, the following exception is returned:
HSM Error: This operation violates the current configured/FIPS policies.

The parameter values for generating the RSA key pair are in the highlighted section above; these include:

keySize	Size of the key in bits. Here 2048.
exponent	Public exponent; can be any odd number, typically ≥ 65537 to $2^{31}-1$.
pub_label	Public key label. Here publickey
priv_label	Private key label. Here privatekey
extractable	Boolean value, where true means the key will be extractable
persistent	Boolean value, where true means the key will be persistent in the HSM.

Generating an ECC Key Pair

The code snippet below provides an example of generating a key pair using ECC secp256k1. The example uses the constructor CaviumECGenParameterSpec.

```
/**
 * ECC Key Pair generation .
 */
public class KeyPairECC {
    public static KeyPair KeyPairECC() throws Exception {
        KeyPairGenerator keyPair = KeyPairGenerator.getInstance("EC", "Cavium");
        keyPair.initialize(new CaviumECGenParameterSpec("secp256k1", "ec_pub", "ec_priv", true, true));
        return keyPair.generateKeyPair();
    }

    public static void main(final String[] args) throws Exception {
        try {
            Security.addProvider(new com.cavium.provider.CaviumProvider());
        } catch (Exception ex) {
            System.out.println(ex);
            return;
        }

        LoginManager lm = LoginManager.getInstance();
        lm.login("PARTITION_1", "cu1", "user1234");
    }
}
```

```

    KeyPair ecc;
    ecc=KeyPairECC();
    lm.logout();
}
}

```

The parameter values for generating the ECC key pair are in the highlighted section above; these include:

keySize	Size of the key in bits. 168 and 192-bit keys are supported; here 192.
pub_label	Public key label. Here ec_pub.
priv_label	Private key label. Here ec_priv.
extractable	Boolean value, where true means the key will be extractable
persistent	Boolean value, where true means the key will be persistent in the HSM

Making a Session Key Persistent and Deleting a Key in the HSM

The method `com.cavium.cfm2.Util.persistKey()` can be used to make a session key persistent as a token key stored in the HSM.

The method `com.cavium.cfm2.Util.deleteKey()` deletes a key in the HSM.

The following code snippet illustrates how to make a session key persistent and delete a key in the HSM:

```

/**
 * Example of making session key persistent and deleting a key.
 */
public class KeyPersistDelete {
    public static KeyPair generateECKKeyPair() throws Exception {
        KeyPairGenerator keyPair = KeyPairGenerator.getInstance("EC","Cavium");
        keyPair.initialize(256, null);
        return keyPair.generateKeyPair();
    }
}

public static void main(final String[] args) throws Exception {

```

```

try {
    Security.addProvider(new com.cavium.provider.CaviumProvider());
} catch (Exception ex) {
    System.out.println(ex);
    return;
}

LoginManager lm = LoginManager.getInstance();
lm.login("PARTITION_1","cu1", "user1234");
KeyPair kp;
System.out.println("*****Generating ECC Key Pair*****");
kp=generateECCKeypair();
PrivateKey pk = kp.getPrivate();
PublicKey pubkey = kp.getPublic();

Util.persistKey((CaviumKey)pk);
Util.persistKey((CaviumKey)pubkey);
Util.deleteKey((CaviumKey)pk);
Util.deleteKey((CaviumKey)pubkey);
lm.logout();
}
}

```

Using Message Digests, MACs and HMACs

The Azure Cloud HSM JCE provider supports various methods to ensure data integrity. This section discusses the supported algorithms for the following types of authentications: Message Digests, MACs and HMACs

Supported Message Digests

A message digest algorithm calculates a cryptographic checksum, or hash, for a particular message and is used to guarantee the integrity of the message data. A hash function is a mathematical function that converts a numerical input value into another compressed numerical value. The input to the hash function is of arbitrary length, but output is always of fixed length.

Values returned by a hash function are called message digests, or simply hash values.

The Azure Cloud HSM JCE provider includes support for the following message digest types.

Digest Name	JCE Name
AES-CMAC	AESCMAC
SHA-1	SHA-1, SHA1
SHA-224	SHA-224, SHA224
SHA-256	SHA256
SHA-384	SHA384
SHA-512	SHA512

Creating a Message Digest

The following code sample illustrates how to create a message digest using the Azure Cloud HSM JCE provider and SHA-224.

```
/**
 * Example of creating a message digest.
 */
public class MessageDigestExample {
    public static void main(final String[] args) throws Exception {
        try {
            Security.addProvider(new
                com.cavium.provider.CaviumProvider());
        } catch (Exception ex) {
            System.out.println(ex);
            return;
        }

        byte[] message = new byte[] {
            0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
            0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
```

```
0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06 };
```

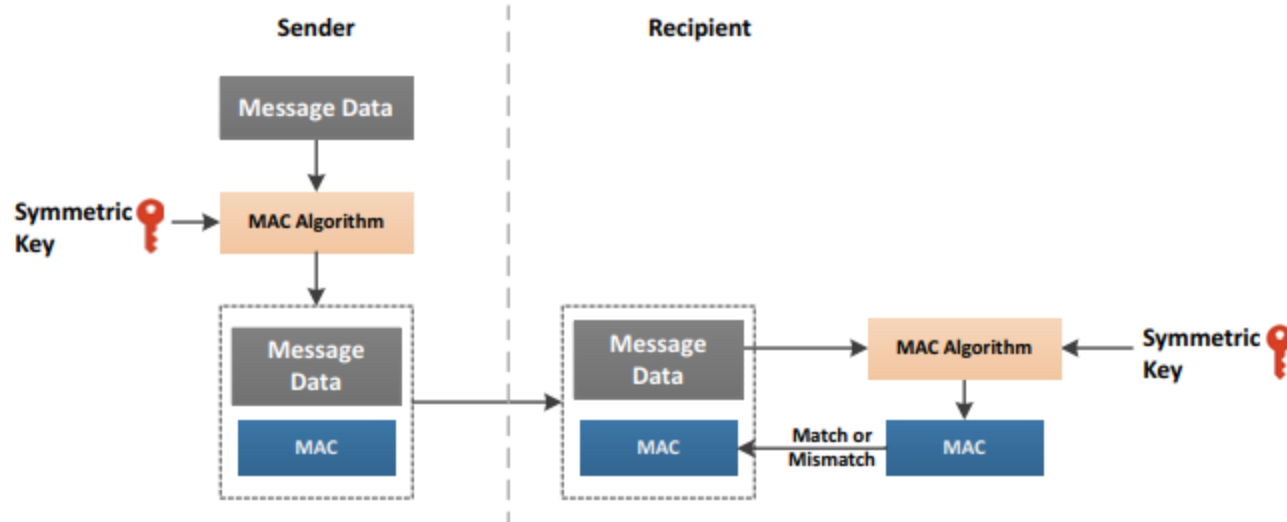
```
    LoginManager lm = LoginManager.getInstance();  
    lm.login("PARTITION_1", "cu1", "user1234");  
    System.out.println("*****Message Digest*****");  
    MessageDigest md = MessageDigest.getInstance("SHA-224", "Cavium");  
    byte[] digest = md.digest(message);  
    System.out.println(Arrays.toString(digest));  
    lm.logout();  
}  
}
```

Supported MACs and HMACs

A MAC algorithm is a symmetric key cryptographic technique to provide message authentication. To establish a MAC process, the sender and receiver must share a symmetric key.

In the following diagram, the sender generates a MAC with a symmetric key and then sends the message data with the MAC to the recipient. The recipient uses their symmetric key to calculate the MAC and verifies that it matches the MAC included with the message.

If a message digest rather than a symmetric key is used to generate the MAC, it is known as a Hash MAC, or HMAC.



The Azure Cloud HSM JCE provider supports the following algorithms for MAC/HMAC operations:

Algorithm	JCE Name
HmacSHA1	HmacSHA1, Hmac128SHA1
HmacSHA224	HmacSHA224, Hmac128SHA224
HmacSHA256	HmacSHA256, Hmac128SHA256
HmacSHA384	HmacSHA384, Hmac256SHA384
HmacSHA512	HmacSHA512, Hmac256SHA512
AESCMAC	AESCMAC

Important Note: In code, the algorithm name is not case.

Generating a MAC

The following example generates a 256-bit AES key to generate the MAC

```
/**
```

```

* Example of generating an AES key to generate a MAC.
*/
public class MacExample {
    public static void main(final String[] args) throws Exception {
        try {
            Security.addProvider(new
                com.cavium.provider.CaviumProvider());
        } catch (Exception ex) {
            System.out.println(ex);
            return;
        }

        byte[] message = new byte[] {
            0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
            0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
            0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06 };;

        LoginManager lm = LoginManager.getInstance();
        lm.login("PARTITION_1", "cu1", "user1234");
        KeyGenerator kg = KeyGenerator.getInstance("AES", "Cavium");
        kg.init(new CaviumAESKeyGenParameterSpec(256, "AESTest", true, true));
        SecretKey aesk = kg.generateKey();
        Mac mac = Mac.getInstance("AESCMAC", "Cavium");
        mac.init(aesk);
        mac.update(message);
        byte[] digest = mac.doFinal();
        System.out.println(Arrays.toString(digest));
        lm.logout();
    }
}

```

Note there are several different ways of calculating a MAC with a given cipher, such as CBC-MAC and GMAC.

Generating an HMAC

The following example generates an HMAC using the HMAC-SHA224 algorithm.

```
/**
 * Example of generating an HMAC with HMAC-SHA224.
 */
public class MacExample {
    public static void main(final String[] args) throws Exception {
        try {
            Security.addProvider(new
                com.cavium.provider.CaviumProvider());
        } catch (Exception ex) {
            System.out.println(ex);
            return;
        }

        byte[] message = new byte[] {
            0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
            0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
            0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06 };

        LoginManager lm = LoginManager.getInstance();
        lm.login("PARTITION_1", "cu1", "user1234");
        KeyGenerator kg = KeyGenerator.getInstance("DESede", "Cavium");
        kg.init(new CaviumDESKeyGenParameterSpec(192, "destest", true, true));
        SecretKey desk = kg.generateKey();
        Mac mac = Mac.getInstance("HmacSHA224", "Cavium");
        mac.init(desk);
        mac.update(message);
        byte[] digest = mac.doFinal();
        System.out.println(Arrays.toString(digest));
        lm.logout();
    }
}
```



```
}  
}
```

Cipher Operations (Symmetric and Asymmetric Keys)

In this segment, we explore the block and stream ciphers that the Azure Cloud HSM JCE Provider supports for executing encryption and decryption operations. It details the supported modes of operation for each type of cipher and provides examples for using the Azure Cloud HSM JCE Provider for cipher operations.

Methods to Define the Cipher Class

There are two methods for defining the cipher class to perform encryption and decryption:

- **Method 1** – This method is usually used for block ciphers.

```
definition := cipher_name "/" mode "/" padding
```

Example: Cipher c = cipher.getInstance("AES/CBC/PKCS5Padding");

- **Method 2** – This method usually used with stream ciphers.

```
Cipher c = cipher.getInstance("AES");
```

Supported Maximum Encryption and Decryption Lengths

The Azure Cloud HSM JCE Provider supports the following maximum encryption and decryption lengths:

Algorithm	Encryption Maximum Length	Decryption Maximum Length
AES-CBC	No max limit	No max limit
AES-CCM	16000 bytes	
AES-CTR	No max limit	No max limit
AES-ECB	No max limit	No max limit
AES-GCM	16000 bytes	
DESede-CBC	No max limit	No max limit
DESede-ECB	No max limit	No max limit

Symmetric Key Cipher Operations

The following symmetric algorithms/mode of operation/padding combinations have been defined for cipher operations with the Azure Cloud HSM JCE Provider. When the cipher name alone is used, the value in bold indicates the default mode of operation and padding.

Algorithm	Mode	Padding	Naming Notes	JCE Cipher
AES	CBC	AES/CBC/NoPadding AES/CBC/PKCS5Padding	Implements: Cipher.ENCRYPT_MODE Cipher.DECRYPT_MODE	AES
	CCM	AES/CCM/NoPadding	Implements: Cipher.ENCRYPT_MODE Cipher.DECRYPT_MODE	AES
	CTR	AES/CTR/NoPadding	Implements: Cipher.ENCRYPT_MODE Cipher.DECRYPT_MODE	AES
	ECB	AES/ECB/NoPadding	Implements: Cipher.ENCRYPT_MODE Cipher.DECRYPT_MODE	AES
	GCM	AES/GCM/NoPadding	Implements: Cipher.ENCRYPT_MODE Cipher.DECRYPT_MODE	AES
DESede	CBC	DESede/CBC/NoPadding DESede/CBC/PKCS5Padding	Implements: Cipher.ENCRYPT_MODE Cipher.DECRYPT_MODE	DESede
	ECB	DESede/CBC/NoPadding DESede/CBC/PKCS5Padding	Implements: Cipher.ENCRYPT_MODE Cipher.DECRYPT_MODE	DESede

Example Using AES/CBC with PKCS5 Padding

The below example illustrates how to perform encryption using AES in CBC mode with PKCS5 padding.

```
/**
 * Example of AES-CBC encryption.
 */
public class CipherAES {
    public static SecretKey generateAESKeyPair() throws Exception {
        KeyGenerator kg = KeyGenerator.getInstance("AES", "Cavium");
        kg.init(new CaviumAESKeyGenParameterSpec(256, UUID.randomUUID().toString(), true, true));
        return kg.generateKey();
    }
}
```

```

public static void main(final String[] args) throws Exception {
    try {
        Security.addProvider(new com.cavium.provider.CaviumProvider());
    } catch (Exception ex) {
        System.out.println(ex);
        return;
    }

    LoginManager lm = LoginManager.getInstance();
    lm.login("PARTITION_1", "cu1", "user1234");
    SecretKey aeskp;
    System.out.println("*****Generating AES Key *****");
    aeskp=generateAESKeyPair();

    byte[] plaintext = "SampleText".getBytes();
    System.out.println("*****Plain text *****");
    System.out.println(Arrays.toString(plaintext));
    Cipher c = Cipher.getInstance("aes/CBC/PKCS5Padding", "Cavium");
    c.init(Cipher.ENCRYPT_MODE, aeskp);
    byte[] ciphertext = c.doFinal(plaintext);

    System.out.println("*****Encrypted message *****");
    System.out.println(Arrays.toString(ciphertext));
    c.init(Cipher.DECRYPT_MODE, aeskp, new IvParameterSpec(c.getIV()));
    byte[] verifytext = c.doFinal(ciphertext);

    System.out.println("*****After Decryption Message *****");
    System.out.println(Arrays.toString(verifytext));
    lm.logout();
}
}

```

Sample output from the above code looks like the following:

```
chsmVMAdmin@myLinuxVM: ~$ java -cp .:azure-cloud-hsm-jca-11-jar-with-dependencies.jar: CipherAES
ERROR StatusLogger Log4j2 could not find a logging implementation. Please add log4j-core to the classpath. Using SimpleLogger to log to the console...
*****Generating AES Key *****
*****Plain text *****
[83, 97, 109, 112, 108, 101, 84, 101, 120, 116]
*****Encrypted message *****
[116, 66, -55, -44, -14, -34, -120, 93, 115, -6, -84, 13, -81, 63, 32, 42]
*****After Decryption Message *****
[83, 97, 109, 112, 108, 101, 84, 101, 120, 116]
```

Example Using AES/CTR with No Padding

The following example illustrates the use of AES-CTR with an external IV.

```
/**
 * Example of AES-CTR encryption.
 */
public class CipherAES {
    public static SecretKey generateAESKeyPair() throws Exception {
        KeyGenerator kg = KeyGenerator.getInstance("AES", "Cavium");
        kg.init(new CaviumAESKeyGenParameterSpec(256, UUID.randomUUID().toString(), true, true));
        return kg.generateKey();
    }

    public static void main(final String[] args) throws Exception {
        try {
            Security.addProvider(new com.cavium.provider.CaviumProvider());
        } catch (Exception ex) {
            System.out.println(ex);
            return;
        }

        LoginManager lm = LoginManager.getInstance();
```

```

lm.login("PARTITION_1","cu1", "user1234");
SecretKey aeskp;
System.out.println("*****Generating AES Key *****");
aeskp=generateAESKeyPair();
byte[] plaintext = "SampleText1".getBytes();
byte[] IV = new byte[] {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};

System.out.println("*****Plain text *****");
System.out.println(Arrays.toString(plaintext));
Cipher c = Cipher.getInstance("aes/CTR/NoPadding", "Cavium");
c.init(Cipher.ENCRYPT_MODE, aeskp, new IvParameterSpec(IV));
byte[] ciphertext = c.doFinal(plaintext);

System.out.println("*****Encrypted message *****");
System.out.println(Arrays.toString(ciphertext));
c.init(Cipher.DECRYPT_MODE, aeskp, new IvParameterSpec(IV));
byte[] verifytext = c.doFinal(ciphertext);

System.out.println("*****After Decryption Message *****");
System.out.println(Arrays.toString(verifytext));
lm.logout();
}
}

```

Sample output from the above code looks like the following:

```

chsmVMAdmin@myLinuxVM: X + v
chsmVMAdmin@myLinuxVM:~$ java -cp .:azure-cloud-hsm-jca-11-jar-with-dependencies.jar: CipherAES
ERROR StatusLogger Log4j2 could not find a logging implementation. Please add log4j-core to the classpath. Using SimpleLogger to log to the console...
*****Generating AES Key *****
*****Plain text *****
[83, 97, 109, 112, 108, 101, 84, 101, 120, 116, 49]
*****Encrypted message *****
[69, -72, -124, 58, -50, -8, 30, 1, -52, -121, -124]
*****After Decryption Message *****
[83, 97, 109, 112, 108, 101, 84, 101, 120, 116, 49]

```

Asymmetric Key Cipher Operations

The following public key algorithms/mode of operation/padding combinations have been defined for cipher operations with the Azure Cloud HSM JCE Provider. When the cipher name alone is used, the value in bold indicates the default mode of operation and padding.

Algorithm	Mode	Padding	Naming Notes	JCA Algorithm
RSA	ECB	RSA/ECB/NoPadding RSA/ECB/PKCS1Padding	Implements: Cipher.ENCRYPT_MODE Cipher.DECRYPT_MODE	RSA
		RSA/ECB/OAEPPadding RSA/ECB/OAEPWithSHA-1ANDMGF1Padding RSA/ECB/OAEPWithSHA-224ANDMGF1Padding RSA/ECB/OAEPWithSHA-256ANDMGF1Padding RSA/ECB/OAEPWithSHA-384ANDMGF1Padding RSA/ECB/OAEPWithSHA-512ANDMGF1Padding	Implements: Cipher.ENCRYPT_MODE Cipher.DECRYPT_MODE OAEPPadding is OAEPWithSHA1ANDMGF1Padding	RSA

Example Using RSA with ECB and PKCS1 Padding

The following example generates an RSA key pair and uses ECB with PKCS1 padding for encryption and decryption:

```

/*
 * Example of RSA-ECB encryption.
 */
public class CipherRSA {
    public static KeyPair generateRSAKeyPair() throws Exception {
        KeyPairGenerator keyPair = KeyPairGenerator.getInstance("RSA", "Cavium");
        keyPair.initialize(new CaviumRSAKeyGenParameterSpec(2048, new BigInteger("65537"), "publickey", "privatekey", true, true));
    }
}

```

```

    return keyPair.generateKeyPair();
}

public static void main(final String[] args) throws Exception {
    try {
        Security.addProvider(new com.cavium.provider.CaviumProvider());
    } catch (Exception ex) {
        System.out.println(ex);
    }
    return;
}

LoginManager lm = LoginManager.getInstance();
lm.login("PARTITION_1", "cu1", "user1234");
KeyPair kp;
System.out.println("*****Generating RSA Key Pair*****");
kp=generateRSAKeyPair();
PrivateKey pk = kp.getPrivate();
PublicKey pubkey = kp.getPublic();
byte[] plaintext = "SampleText1".getBytes();

System.out.println("*****Plain text *****");
System.out.println(Arrays.toString(plaintext));
Cipher c = Cipher.getInstance("RSA/ECB/PKCS1Padding", "Cavium");
c.init(Cipher.ENCRYPT_MODE, pubkey);
byte[] ciphertext = c.doFinal(plaintext);
System.out.println(Arrays.toString(ciphertext));
c.init(Cipher.DECRYPT_MODE, pk);
byte[] verifytext = c.doFinal(ciphertext);

System.out.println("*****After Decryption Message*****");
System.out.println(Arrays.toString(verifytext));
lm.logout();

```

```
}  
}
```

Sample output from the above code looks like the following:

```
chsmVMAdmin@myLinuxVM: X + v  
chsmVMAdmin@myLinuxVM:~$ java -cp .:azure-cloud-hsm-jca-11-jar-with-dependencies.jar: CipherRSA  
ERROR StatusLogger Log4j2 could not find a logging implementation. Please add log4j-core to the classpath. Using SimpleLogger to log to the console...  
*****Generating RSA Key Pair*****  
*****Plain text *****  
[83, 97, 109, 112, 108, 101, 84, 101, 120, 116, 49]  
[124, -27, 20, -87, -19, 100, -42, -85, 101, -58, 43, -103, -5, -56, -86, -25, 51, -16, -20, -1, -118, -120, 31, 37, -118, 38, -51, 92, 56, 47, 79, 52, 90, -6, -58, 36, 105, 0, 31, -61, 12, -65, -43, 86, -34, -26, 55, -5, 122, -42, -41, -7, -91, -106, 121, 2, -90, 91, 97, 13, -118, -92, -73, 105, -85, -44, -88, 92, 7, -45, -30, -1, 117, -65, -76, -55, 74, 118, -125, -94, -20, 14, -86, -94, 103, 5, 11, -72, 51, 15, 93, -30, 22, 99, 70, -99, 77, -36, -86, 51, -75, -91, -43, -112, 105, 112, -111, 33, 15, -30, -2, 60, 62, 85, -115, 27, 7, 0, -47, 57, 79, 119, 126, -58, -97, 51, -74, 18, 5, -50, 110, -65, 33, 122, 23, -20, -34, 115, 54, 12, -75, -127, -79, 46, 5, -105, -81, -117, 93, 20, -50, 28, -108, 42, -79, 32, 56, 37, -123, -88, -110, -78, 34, -122, -105, -74, -113, 5, -128, 75, 74, -3, -83, 75, -122, 9, -3, -96, 76, 121, -115, -87, -63, 53, -54, 4, -47, 76, 69, -15, 76, 2, -47, 120, 20, -37, 72, -46, 95, 127, 11, -83, 50, 95, -84, -64, 74, -35, 62, -9, 80, -2, -109, -67, 122, 75, -60, 84, 115, -11, -1, 127, 52, -58, -99, 4, 121, -97, 17, 13, -40, 91, 117, 25, -59, -91, 124, -56, 123, 50, 15, -87, 115, 67, 73, -71, -30, -106, -36, 8, -56, 27, 20, 125, 94, -22]  
*****After Decryption Message*****  
[83, 97, 109, 112, 108, 101, 84, 101, 120, 116, 49]
```

Transferring Keys

Below covers the supported methods for securely transferring keys using the Azure Cloud HSM JCE Provider. The supported features are detailed, and example code samples are provided for both wrapping/unwrapping and importing keys.

Important Note:

The JCE provider supports cryptographic operations exclusively for authorized crypto users. Azure Cloud HSM imposes restrictions on exporting (wrapping) a key to any key not designated as trusted by default. Designating a key as trusted necessitates permission from a cryptographic officer. Attempting to wrap and unwrap keys using the Azure Cloud HSM JCE provider without prior designation of the wrapping key as trusted will result in an exception.

For key wrap and unwrap operations, two options are available:

1. **Recommended:** Employing recommended approaches for WRAP_WITH_TRUSTED. Customers can designate the wrapping key as trusted using the `azcloudhsm_mgmt_util`, allowing it to be utilized as a wrapping key within the JCE provider. Additionally, the `azcloudhsm_util` offers a `wrap_with_trusted` parameter, enabling specification of the value on a per-key basis. This is most common for customers who need Key Import capabilities. Please see [creating a user generated KEK](#) section in this document for instructions.

2. **Alternative:** Wrapping a key with a non-trusted key requires WRAP_WITH_TRUSTED=0. Customers then alter the default behavior by adjusting the configuration setting DEFAULT_WRAP_WITH_TRUSTED=1 to DEFAULT_WRAP_WITH_TRUSTED=0 in the azcloudhsm_application.cfg. This setting allows keys generated within the Azure Cloud HSM SDK with the EXTRACTABLE=1 attribute (enabled) to be exported as plaintext without involvement from the partition's cryptographic officer. However, it's crucial to comprehend the security ramifications of such changes and explore more secure alternatives based on your specific use case and security requirements. Note that DEFAULT_WRAP_WITH_TRUSTED is set to 1 (enabled) by default for security reasons.

Wrapping and Unwrapping Keys

Supported Wrapping Operations

The following algorithm/mode of operation/padding combinations have been defined for key wrapping operations with the Azure Cloud HSM JCE provider. When the cipher name alone is used, the value in **bold** indicates the default mode of operation and padding.

The following key wrapping operations are available:

Symmetric Cipher Key Wrapping Operations

Algorithm	Mode	Padding	Naming Notes	JCE Name
AES	CBC	AES/CBC/NoPadding AES/CBC/PKCS5Padding	Implements Cipher.ENCRYPT_MODE & Cipher.DECRYPT_MODE Additionally, it supports Cipher.WRAP_MODE and Cipher.UNWRAP_MODE if Util.ENABLE_14_X_BEHAVIOUR is set to true	AES
	ECB	AES/ECB/NoPadding AES/ECB/PKCS5Padding	Implements Cipher.ENCRYPT_MODE & Cipher.DECRYPT_MODE	
	CTR	AES/CTR/NoPadding	Implements Cipher.ENCRYPT_MODE & Cipher.DECRYPT_MODE	
	GCM	AES/GCM/NoPadding	Implements Cipher.ENCRYPT_MODE, Cipher.DECRYPT_MODE Cipher.WRAP_MODE & Cipher.UNWRAP_MODE	

	CCM	AES/CCM/NoPadding	Implements Cipher.ENCRYPT_MODE, Cipher.DECRYPT_MODE	
AESWrap	ECB	AESWrap/ECB/NoPadding AESWrap/ECB/ZeroPadding AESWrap/ECB/PKCS5Padding	Implements Cipher.WRAP_MODE & Cipher.UNWRAP_MODE	AES
DESede	CBC	DESede/CBC/NoPadding DESede/CBC/PKCS5Padding	Implements Cipher.ENCRYPT_MODE, Cipher.DECRYPT_MODE	DeSede
	ECB	DESede/ECB/NoPadding DESede/ECB/PKCS5Padding	Implements Cipher.ENCRYPT_MODE, Cipher.DECRYPT_MODE	
DESedeWrap	ECB	DESedeWrap/ECB/NoPadding	Implements Cipher.WRAP_MODE & Cipher.UNWRAP_MODE	DeSede
RSA	ECB	RSA/ECB/NoPadding	Implements Cipher.ENCRYPT_MODE & Cipher.DECRYPT_MODE	RSA
		RSA/ECB/PKCS1Padding	Implements Cipher.ENCRYPT_MODE, Cipher.DECRYPT_MODE, Cipher.WRAP_MODE & Cipher.UNWRAP_MODE	
		RSA/ECB/OAEPPadding RSA/ECB/OAEPWithSHA-1ANDMGF1Padding RSA/ECB/OAEPWithSHA-224ANDMGF1Padding RSA/ECB/OAEPWithSHA-256ANDMGF1Padding RSA/ECB/OAEPWithSHA-384ANDMGF1Padding RSA/ECB/OAEPWithSHA-512ANDMGF1Padding	Implements Cipher.ENCRYPT_MODE, Cipher.DECRYPT_MODE, Cipher.WRAP_MODE & Cipher.UNWRAP_MODE OAEPPadding is OAEPWithSHA1ANDMGF1Padding	
RSAAESWrap	ECB	RSAAESWrap/ECB/OAEPPadding RSAAESWrap/ECB/OAEPWithSHA1ANDMGF1Padding RSAAESWrap/ECB/OAEPWithSHA-224ANDMGF1Padding RSAAESWrap/ECB/OAEPWithSHA-256ANDMGF1Padding RSAAESWrap/ECB/OAEPWithSHA-384ANDMGF1Padding RSAAESWrap/ECB/OAEPWithSHA-512ANDMGF1Padding	Implements Cipher.WRAP_MODE & Cipher.UNWRAP_MODE OAEPPadding is OAEPWithSHA1ANDMGF1Padding	RSA

Asymmetric Cipher Key Wrapping Operations

Algorithm	Mode	Padding	Naming Notes	JCE Name
RSA	ECB	RSA/ECB/NoPadding	Implements Cipher.ENCRYPT_MODE & Cipher.DECRYPT_MODE	RSA
		RSA/ECB/PKCS1Padding	Implements Cipher.ENCRYPT_MODE, Cipher.DECRYPT_MODE, Cipher.WRAP_MODE & Cipher.UNWRAP_MODE	
		RSA/ECB/OAEP Padding RSA/ECB/OAEPWithSHA-1ANDMGF1Padding RSA/ECB/OAEPWithSHA-224ANDMGF1Padding RSA/ECB/OAEPWithSHA-256ANDMGF1Padding RSA/ECB/OAEPWithSHA-384ANDMGF1Padding RSA/ECB/OAEPWithSHA-512ANDMGF1Padding	Implements Cipher.ENCRYPT_MODE, Cipher.DECRYPT_MODE, Cipher.WRAP_MODE & Cipher.UNWRAP_MODE OAEP Padding is OAEPWithSHA1ANDMGF1Padding	
RSAAESWrap	ECB	RSAAESWrap/ECB/OAEP Padding RSAAESWrap/ECB/OAEPWithSHA1ANDMGF1Padding RSAAESWrap/ECB/OAEPWithSHA-224ANDMGF1Padding RSAAESWrap/ECB/OAEPWithSHA-256ANDMGF1Padding RSAAESWrap/ECB/OAEPWithSHA-384ANDMGF1Padding RSAAESWrap/ECB/OAEPWithSHA-512ANDMGF1Padding	Implements Cipher.WRAP_MODE & Cipher.UNWRAP_MODE OAEP Padding is OAEPWithSHA1ANDMGF1Padding	RSA

Wrapping and Unwrapping Keys with AES and AESWrap

The following example illustrates wrapping and unwrapping a key using AESWrap with an external IV.

Important Note: For this operation to be successful WRAP_WITH_TRUSTED=0 but be set in the azcloudhsm_application.cfg.

```

/*
 * Example of wrapping/unwrapping using AESWrap and external IV.
 */
public class WrapUnwrapAESNoPad {
    public static SecretKey generateAESKey() throws Exception {

```

```
KeyGenerator kg = KeyGenerator.getInstance("AES", "Cavium");
kg.init(new CaviumAESKeyGenParameterSpec(256, UUID.randomUUID().toString(), true,true));
return kg.generateKey();
}
```

```
public static void main(final String[] args) throws Exception { try {
    Security.addProvider(new com.cavium.provider.CaviumProvider());
} catch (Exception ex) {
    System.out.println(ex);
    return;
}
```

```
LoginManager lm = LoginManager.getInstance(); KeyPair kp;
lm.login("PARTITION_1","cu1", "user1234");
SecretKey wrapkp,aeskp;
aeskp=generateAESKey();
wrapkp=generateAESKey();
```

```
Cipher caviumAes = Cipher.getInstance("AESWrap/ECB/NoPadding", "Cavium");
caviumAes.init(Cipher.WRAP_MODE,aeskp);
byte[] cText=caviumAes.wrap(wrapkp);
System.out.print("Wrapped Content");
for(int i=0; i< cText.length ; i++)
{
    System.out.print(cText[i] +" ");
}
System.out.print("\n");
```

```
Scanner input = new Scanner(System.in);
int num = input.nextInt();
caviumAes.init(Cipher.UNWRAP_MODE, aeskp);
Key uk = caviumAes.unwrap(cText, "AES", Cipher.SECRET_KEY);
```

```

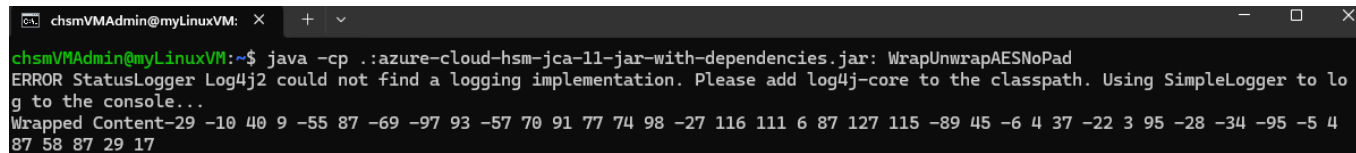
input = new Scanner(System.in);
num = input.nextInt();

if( Arrays.equals(wrapkp.getEncoded(),uk.getEncoded()) == false)
    System.out.println("FAIL - Wrapped / UnWrapped Cavium Key bits mismatch");

lm.logout();
}
}

```

Sample output from the above code looks like the following:



```

chsmVMAdmin@myLinuxVM: ~$ java -cp .:azure-cloud-hsm-jca-11-jar-with-dependencies.jar: WrapUnwrapAESNoPad
ERROR StatusLogger Log4j2 could not find a logging implementation. Please add log4j-core to the classpath. Using SimpleLogger to log to the console...
Wrapped Content-29 -10 40 9 -55 87 -69 -97 93 -57 70 91 77 74 98 -27 116 111 6 87 127 115 -89 45 -6 4 37 -22 3 95 -28 -34 -95 -5 4
87 58 87 29 17

```

Wrapping and Unwrapping Keys with OAEP

Performing OAEP Key Wrapping

```

Cipher caviumRsa = Cipher.getInstance("RSA/ECB/OAEPWithSHA-256ANDMGF1Padding", "Cavium");
caviumRsa.init(Cipher.WRAP_MODE, pubkey);
byte[] cText=caviumRsa.wrap(aeskp);

```

Performing OAEP Key Unwrapping

The example below is the corresponding unwrap method for the wrap method above.

```

Cipher caviumRsa = Cipher.getInstance("RSA/ECB/OAEPWithSHA-256ANDMGF1Padding", "Cavium");
caviumRsa.init(Cipher.UNWRAP_MODE, pk);
Key uk=caviumRsa.unwrap(cText, "AES", Cipher.SECRET_KEY);

```

Example 1: Wrapping an AES Key

In this example, the AES key is wrapped using an RSA public key, and then unwrapped using the RSA private key.

Important Note: For this operation to be successful `WRAP_WITH_TRUSTED=0` but be set in the `azcloudhsm_application.cfg`.

```

/**
 * Example of wrapping AES key with RSA public key and unwrapping with RSA private key.
 */
public class WrapUnwrapRSAAES {
    public static KeyPair generateRSAKeyPair() throws Exception {
        KeyPairGenerator keyPair = KeyPairGenerator.getInstance("RSA","Cavium");
        keyPair.initialize(new CaviumRSAKeyGenParameterSpec(2048,new BigInteger("65537"),"publickey", "privatekey",true,true));
        return keyPair.generateKeyPair();
    }

    public static SecretKey generateAESKey() throws Exception {
        KeyGenerator kg = KeyGenerator.getInstance("AES", "Cavium");
        kg.init(new CaviumAESKeyGenParameterSpec(256, UUID.randomUUID().toString(), true,true));
        return kg.generateKey();
    }

    public static void main(final String[] args) throws Exception { try {
        Security.addProvider(new com.cavium.provider.CaviumProvider());
    } catch (Exception ex) {
        System.out.println(ex);
        return;
    }

    LoginManager lm = LoginManager.getInstance(); KeyPair kp;
    lm.login("PARTITION_1","cu1", "user1234");
    SecretKey aeskp,wrapkey;
    kp=generateRSAKeyPair();
    PrivateKey pk = kp.getPrivate();
    PublicKey pubkey = kp.getPublic();
    aeskp=generateAESKey();
    Cipher caviumRsa = Cipher.getInstance("RSAESWrap/ECB/OAEPADDING", "Cavium");
    caviumRsa.init(Cipher.WRAP_MODE,pubkey);

```

```

byte[] cText=caviumRsa.wrap(aeskp);
System.out.print("Wrapped Content");
for(int i=0; i< cText.length ; i++)
{
System.out.print(cText[i] +" ");
}
System.out.print("\n");

Scanner input = new Scanner(System.in);
int num = input.nextInt();
caviumRsa.init(Cipher.UNWRAP_MODE, pk);
Key uk = caviumRsa.unwrap(cText, "AES", Cipher.SECRET_KEY);
input = new Scanner(System.in);
num = input.nextInt();

if( Arrays.equals(aeskp.getEncoded(),uk.getEncoded()) == false)
    System.out.println("FAIL - Wrapped / UnWrapped Cavium Key bits mismatch");

lm.logout();
}
}

```

Example 2: Using RSA-AES Wrapping/Unwrapping Methods for RSA

The following code illustrates the wrapping/unwrapping methods defined for RSA.

Important Note: For this operation to be successful `WRAP_WITH_TRUSTED=0` but be set in the `azcloudhsm_application.cfg`.

```

/**
 * Test Wrap and unwrap function using RSAAESWrap mechanism.
 */
public class WrapUnwrapRSAES {
    /**
     * This function generates the RSA Key Pair

```

```

/**/
public static KeyPair generateRSAKeyPair() throws Exception {
    KeyPairGenerator keyPair = KeyPairGenerator.getInstance("RSA","Cavium");
    keyPair.initialize(new CaviumRSAKeyGenParameterSpec(2048,new BigInteger("65537"),"publickey", "privatekey",true,true));
    return keyPair.generateKeyPair();
}

/**
 * This function generates AES Key
 */
public static SecretKey generateAESKey() throws Exception {
    KeyGenerator kg = KeyGenerator.getInstance("AES", "Cavium"); kg.init(new CaviumAESKeyGenParameterSpec(256, UUID.randomUUID().toString(),
true,true));
    return kg.generateKey();
}

public static void main(final String[] args) throws Exception { try {
    Security.addProvider(new com.cavium.provider.CaviumProvider());
} catch (Exception ex) {
    System.out.println(ex);
    return;
}

    LoginManager lm = LoginManager.getInstance(); KeyPair kp;
    lm.login("PARTITION_1","cu1", "user1234");
    SecretKey aeskp,wrapkey; kp=generateRSAKeyPair(); PrivateKey pk = kp.getPrivate(); PublicKey pubkey =
    kp.getPublic(); aeskp=generateAESKey();

    // wrap the key
    Cipher caviumRsa = Cipher.getInstance("RSAESWrap/ECB/OAEPADDING", "Cavium");
    caviumRsa.init(Cipher.WRAP_MODE,pubkey);
    byte[] cText=caviumRsa.wrap(aeskp);

```



```

for(int i=0; i< cText.length; i++)
{
    System.out.print(cText[i] + " ");
}
System.out.print("\n");

// unwrap the Key
caviumRsa.init(Cipher.UNWRAP_MODE, pk);
Key uk=caviumRsa.unwrap(cText, "AES", Cipher.SECRET_KEY); Scanner input = new Scanner(System.in);
int num = input.nextInt();
if( Arrays.equals(aeskp.getEncoded(),uk.getEncoded()) == false) System.out.println("FAIL - Wrapped/UnWrapped Cavium Key bits mismatch\n");
lm.logout();
}
}

```

Sample outputs from the above code look like the following:

```
Cipher cp = Cipher.getInstance("AES/GCM/NoPadding", "Cavium");
```

The output from running findKey before and after the program is illustrated below.

Before running the program:

Command: findKey

Total number of keys present: 9 Number of matching keys from start index 0::8

Handles of matching keys: 6, 7, 8, 9, 10, 11, 12, 13, 14

Cluster Status: Node id 0 status: 0x00000000 : HSM Return: SUCCESS Cfm3FindKey returned: 0x00 :
HSM Return: SUCCESS

Run the program:

```
chsmVMAdmin@myLinuxVM:~$ java -cp .:azure-cloud-hsm-jca-11-jar-with-dependencies.jar: WrapUnwrapRSAAES
ERROR StatusLogger Log4j2 could not find a logging implementation. Please add log4j-core to the classpath. Using SimpleLogger to log to the console...
18 15 -38 102 44 -50 28 26 -49 98 96 43 -119 -90 -5 96 81 30 -122 -100 59 -4 45 76 91 -4 106 91 -127 50 -46 -104 -45 54 -67 14 115
-58 -46 42 -53 -52 -22 -64 -46 45 75 40 -30 119 -36 -105 -85 -41 92 -92 33 -106 91 -128 24 -6 10 -86 -34 58 -25 21 70 -10 -124 -20
96 93 97 -59 -101 -71 48 61 -103 -122 106 82 -43 -63 -13 -12 79 -93 61 -37 -70 -103 -103 -53 -80 21 89 -56 -22 -97 -78 73 -62 -8 34
-19 -45 92 26 63 12 -50 99 -88 95 58 94 83 29 -15 -77 -51 -65 -87 -31 -109 21 22 10 89 -88 -30 73 -43 77 -5 80 -45 0 112 -99 96 -7
0 117 -48 35 107 25 -82 -110 111 -6 103 124 56 71 41 100 -94 64 0 -70 124 -18 -25 -110 36 65 29 126 -27 -53 6 -120 112 -124 -50 -10
1 -40 27 41 -69 67 108 -122 -51 69 -104 -119 -45 -38 9 122 114 106 -4 95 57 -101 118 45 -103 3 -47 -99 -3 -113 -122 3 24 81 74 96 4
1 -64 -103 -23 -58 51 -104 -126 -27 48 70 101 -9 58 -14 114 -82 79 -65 63 -22 -57 77 92 69 22 -34 -121 -126 -23 47 -27 -84 -114 -11
4 -14 28 112 -120 -93 4 -105 90 -51 113 -80 6 -108 67 -56 -90 -25 -64 -87 59 32 -100 -85 89 103 -8 -104 -20 -28 76 57 121 -50 -19 -
23 85 -86 -70 -24 113 -75 -94 -40 126 -111 -14
```

After running the program (2 keys for RSA, 1 key for AES, 1 key that is wrapped):

Command: findKey

Total number of keys present: 13 Number of matching keys from start index 0::12

Handles of matching keys: 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18

Cluster Status: Node id 0 status: 0x00000000 : HSM Return: SUCCESS Cfm3FindKey returned: 0x00 :

HSM Return: SUCCESS

Importing Keys

Azure Cloud HSM supports importing keys created outside the HSM FIPS boundary and exporting keys in the HSM out of the HSM FIPS boundary. The Azure Cloud HSM JCE Provider can work with keys generated using its key and key pair generators or can potentially work with keys generated by other providers.

Important Note: The Azure Cloud HSM JCE Provider has currently been tested for keys generated by the following SUN providers.

- SunRsaSign provider for RSA keys
- SunJCE provider for AES keys
- SunEC provider for EC keys
- SunEC provider of JDK 13 for XDH keys

Supported Methods for Importing Keys

The Azure Cloud HSM JCE provider can work with keys generated using its key and key pair generators. If an application wants to work with keys generated by other providers, however, the keys must be imported into the HSM using the *KeyFactory* class or the *ImportKey* utility class.

Importing Keys with the *ImportKey* Utility Class

Azure Cloud HSM keys have attributes that define their behavior. To import a key and set these attributes, use the *ImportKey* utility class. This class accepts the *CaviumKeyGenAlgorithmParameterSpec* instance to:

1. Set the key label.
2. Specify if the key is persistent or a session key.
3. Specify if the key is extractable. Note that only symmetric and private keys can be set to be non-extractable. Public keys are always extractable.

Important Note: If an application tries to import a Cavium key, Azure Cloud HSM JCE will throw an exception with the error "The key is an instance of CaviumKey and cannot be imported".

Importing a Key Generated Outside the FIPS Boundary

In this example, the RSA key pair, ECC key pair, and AES key pair are generated outside the FIPS boundary. The application then imports the keys into Azure Cloud HSM. Because the key pair is generated outside the FIPS boundary its treated as untrusted.

Important Note: You can execute this operation by adjusting the "WRAP_WITH_TRUSTED" parameter within the "azcloudhsm_application.cfg" file, setting it either to 1 or 0. However, it's advisable not to alter the default setting of "WRAP_WITH_TRUSTED=1" (enabled) specifically for Key Import within the "azcloudhsm_application.cfg" file, as this would result in exporting the key in plaintext. To ensure a successful Key Import, it's necessary to have previously [created a user-generated KEK](#) following option 1, where "wrap_with_trusted" is set to 1 and the key is designated as extractable.

```
/**
 * Test Program to generate RSA and AES key and import into the HSM.
 **/
public class KeyImportHSM {
    public static SecretKey generateAESKey( String provider ) throws Exception {
        KeyGenerator kg = KeyGenerator.getInstance("AES", provider);
        Provider kpgenProv = kg.getProvider();
```

```

    kg.init(256);
    System.out.printf("Provider : %s%nInfo: %s%n", kpgenProv.getName(),
        kpgenProv.toString());
    return kg.generateKey();
}

public static KeyPair generateRSAKeyPair( String provider ) throws Exception {
    KeyPairGenerator keyPair = KeyPairGenerator.getInstance("RSA", provider);
    Provider kpgenProv = keyPair.getProvider();
    System.out.printf("Provider : %s%nInfo: %s%n", kpgenProv.getName(), kpgenProv.toString());
    keyPair.initialize(new CaviumRSAKeyGenParameterSpec(2048,new BigInteger("65537"),"publickey", "privatekey",true,true));
    return keyPair.generateKeyPair();
}

public static KeyPair generateECCKeyPair( String provider ) throws Exception {
    KeyPairGenerator kpg1 = KeyPairGenerator.getInstance("EC", provider);
    ECGenParameterSpec ecSpec = new ECGenParameterSpec("secp256r1");
    Provider kpgenProv = kpg1.getProvider();
    System.out.printf("Provider : %s%nInfo: %s%n", kpgenProv.getName(), kpgenProv.toString());
    kpg1.initialize(ecSpec);
    return kpg1.generateKeyPair();
}

public static void main(final String[] args) throws Exception {
    try {
        Security.addProvider(new com.cavium.provider.CaviumProvider());
    } catch (Exception ex) {
        System.out.println(ex);
        return;
    }

    LoginManager lm = LoginManager.getInstance();

```

```

lm.login("PARTITION_1","cu1", "user1234");

CaviumKeyGenAlgorithmParameterSpec spec;
SecretKey k,k1;
Key unwrapped_key;
PrivateKey bc_pk,ec_pk;
PublicKey bc_pbk,ec_pbk;
CaviumKey cavium_pk;
boolean extractable = true;
boolean persistent = true;
KeyPair bckp = generateRSAKeyPair("SunRsaSign");
bc_pk = bckp.getPrivate();
bc_pbk = bckp.getPublic();
k = generateAESKey("SunJCE");
KeyPair eckp = generateECKKeyPair("SunEC");
ec_pk=eckp.getPrivate();
ec_pbk=eckp.getPublic();

/** Important Note **
//This requires an existing userKEK marked wrap_with_trusted 1 and attribute OBJ_ATTR_TRUSTED 1

//Import the RSA Private Key
cavium_pk = (CaviumKey) ImportKey.importKey(bc_pk, new CaviumKeyGenAlgorithmParameterSpec("rsaeswraptestprivatekeyimported", true, true));

//Import the AES key (User KEK needs to be created as extractable and trusted for import)
spec = new CaviumKeyGenAlgorithmParameterSpec("myAesKey1", extractable, persistent);
k1 = (SecretKey) ImportKey.importKey(k, spec);

//Import ECC Key Pair
CaviumKeyGenAlgorithmParameterSpec pspec = new
CaviumKeyGenAlgorithmParameterSpec("ECPrivate1", true, true);
CaviumECPrivateKey cecprivKey = (CaviumECPrivateKey)

```

```
    ImportKey.importKey(ec_pk, pspec);

    lm.logout();
}
}
```

Compiling the program

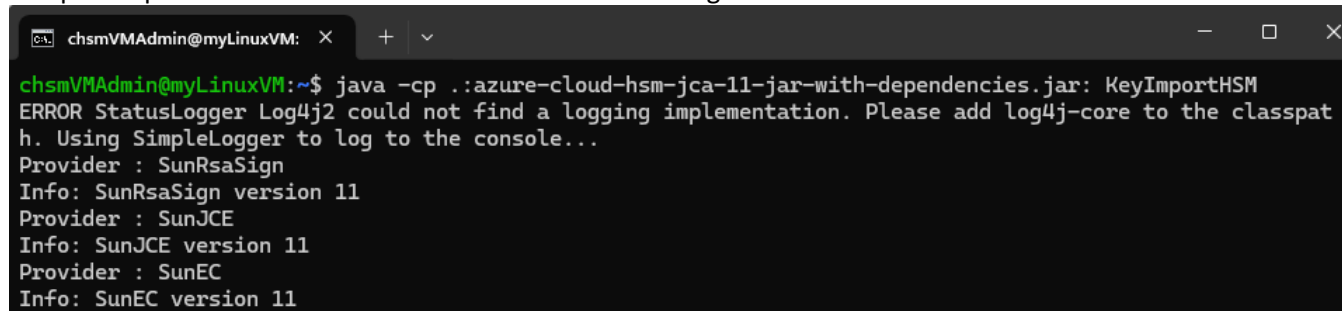
To compile the program, run the following command:

```
javac -cp azure-cloud-hsm-jca-11-jar-with-dependencies.jar: KeyImportHSM.java
```

Running the executable

```
java -cp .:azure-cloud-hsm-jca-11-jar-with-dependencies.jar: KeyImportHSM
```

Sample output from the above code looks like the following:



```
chsmVMAdmin@myLinuxVM: x + v
chsmVMAdmin@myLinuxVM:~$ java -cp .:azure-cloud-hsm-jca-11-jar-with-dependencies.jar: KeyImportHSM
ERROR StatusLogger Log4j2 could not find a logging implementation. Please add log4j-core to the classpath.
Using SimpleLogger to log to the console...
Provider : SunRsaSign
Info: SunRsaSign version 11
Provider : SunJCE
Info: SunJCE version 11
Provider : SunEC
Info: SunEC version 11
```

To verify the application is performing correctly, run findKey from azcloudhsm_util. The output should show all three keys.

Command: findKey

Total number of keys present: 3

Number of matching keys from start index 0::2

Handles of matching keys: 6, 7, 8

Cluster Status:

Node id 0 status: 0x00000000 : HSM Return: SUCCESS

Cfm3FindKey returned: 0x00 : HSM Return: SUCCESS

In this output:

6 = Represents the RSA private key handle

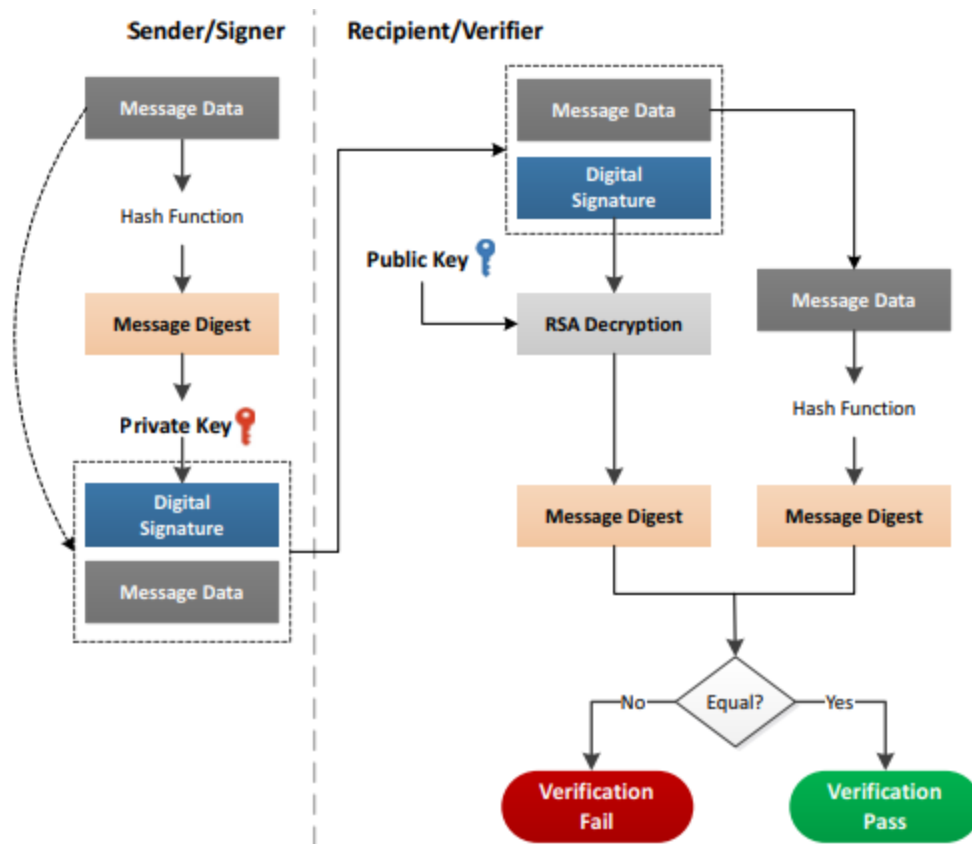
7 = Represents the AES private key handle

8 = Represents the ECC private key handle

Digital Signatures

A digital signature is a method for verifying the authenticity of a message using hash functions and asymmetric key cryptography. To create the signature, a message hash is generated and encrypted with the sender's private key. This signature is sent along with the unencrypted message data. The recipient decrypts the signature with the sender's public key to get the original hash, then generates their own hash from the message and compares the two. If they match, the identity of the sender is confirmed.

The following diagram illustrates this digital signature process.



Signature Class

Objects of the signature type are not created directly, but are created through `getInstance()` methods. As signature algorithms usually involve a message digest, a standard form for signature algorithm names has evolved. The standard format has the form of:

`<message digest>with<public key algorithm>`

For example, `SHA256withRSA` is the algorithm name for RSA using the SHA-256 message digest for processing the message.

Function Used to Create the Digital Signatures

In the case of signature creation, which is a private key operation, the `initSign()` method is used. In the case of signature verification, which is a public key operation, the `initVerify()` method is called with either a public key or a certificate object that contains the public key

Supported Digital Signatures

The following types of digital signatures are supported by the Azure Cloud HSM JCE provider.

Algorithm	Signatures	JCE Name
EC	NONE with ECDSA	NONEwithECDSA
	SHA1 with ECDSA	SHA1withECDSA
	SHA224 with ECDSA	SHA224withECDSA
	SHA256 with ECDSA	SHA256withECDSA
	SHA384 with ECDSA	SHA384withECDSA
	SHA512 with ECDSA	SHA512withECDSA
RSA / PKCS1.5	NONE with RSA	NONEwithRSA
	SHA1 with RSA	SHA1withRSA
	SHA224 with RSA	SHA224withRSA
	SHA256 with RSA	SHA256withRSA
	SHA384 with RSA	SHA384withRSA
	SHA512 with RSA	SHA512withRSA
PSS	PSS SHA1 with RSA	SHA1withRSA/PSS SHA1withRSAandMGF1
	PSS SHA224 with RSA	SHA224withRSA/PSS SHA224withRSAandMGF1
	PSS SHA256 with RSA	SHA256withRSA/PSS SHA256withRSAandMGF1
	PSS SHA384 with RSA	SHA384withRSA/PSS SHA384withRSAandMGF1
	PSS SHA512 with RSA	SHA512withRSA/PSS SHA512withRSAandMGF1

Digital Signing Using ECDSA

The following example shows how to perform digital signing and verification using ECDSA. This requires first generating the EC key pair, and then performing sign and verify.

```
/**
 * Example of digital signing and verification using ECDSA.
 */
public class BasicECDSAExample {
    public static KeyPair ecc_key_pair() throws Exception {
        KeyPairGenerator keyPair = KeyPairGenerator.getInstance("EC", "Cavium");
        keyPair.initialize(new CaviumECGenParameterSpec("secp256k1", "ec_pub", "ec_priv", true, true));
        return keyPair.generateKeyPair();
    }

    public static void main(final String[] args) throws Exception {
        try {
            Security.addProvider(new com.cavium.provider.CaviumProvider());
        } catch (Exception ex) {
            System.out.println(ex);
            return;
        }

        LoginManager lm = LoginManager.getInstance();
        lm.login("PARTITION_1", "cu1", "user1234");
        KeyPair eccPair = ecc_key_pair();

        // generate a signature
        Signature s = Signature.getInstance("SHA256withECDSA", "Cavium");
        s.initSign(eccPair.getPrivate());
        byte[] message = new byte[] { (byte)'a', (byte)'b', (byte)'c' };
        s.update(message);
        byte[] sigBytes = s.sign();
    }
}
```

```

    // verify a signature
    s.initVerify(eccPair.getPublic());
    s.update(message);
    if (s.verify(sigBytes))
    {
        System.out.println("signature verification succeeded.");
    }
    else
    {
        System.out.println("signature verification failed.");
    }
    lm.logout();
}
}

```

RSA Signature with PKCS#1.5 Padding

The following example uses an RSA signature with PKCS#1.5 padding.

```

/**
 * Example of RSA signature with PKCS#1.5 padding.
 */
public class BasicRSASignExample {
    public static KeyPair rsa_key_pair() throws Exception {
        KeyPairGenerator keyPair = KeyPairGenerator.getInstance("RSA", "Cavium");
        keyPair.initialize(new CaviumRSAKeyGenParameterSpec(2048, new BigInteger("65537"), "publickey", "privatekey", true, true));
        return keyPair.generateKeyPair();
    }

    public static void main(final String[] args) throws Exception {
        try {
            Security.addProvider(new com.cavium.provider.CaviumProvider());
        } catch (Exception ex) {

```

```

        System.out.println(ex);
        return;
    }

    LoginManager lm = LoginManager.getInstance();
    lm.login("PARTITION_1","cu1", "user1234");
    KeyPair rsaPair = rsa_key_pair();

    // generate a signature
    Signature s = Signature.getInstance("SHA224withRSA", "Cavium");
    s.initSign(rsaPair.getPrivate());
    byte[] message = new byte[] { (byte)'a', (byte)'b', (byte)'c' };
    s.update(message);
    byte[] sigBytes = s.sign();

    // verify a signature
    s.initVerify(rsaPair.getPublic());
    s.update(message);
    if (s.verify(sigBytes))
    {
        System.out.println("signature verification succeeded.");
    }
    else
    {
        System.out.println("signature verification failed.");
    }
    lm.logout();
}
}

```

RSA Signature with PSS Padding

This example illustrates using an RSA signature with PSS padding.

```

/**
 * Example of RSA signing with PSS padding.
 */
public class BasicRSASignExample {
    public static KeyPair rsa_key_pair() throws Exception {
        KeyPairGenerator keyPair = KeyPairGenerator.getInstance("RSA", "Cavium");
        keyPair.initialize(new CaviumRSAKeyGenParameterSpec(2048, new BigInteger("65537"), "publickey", "privatekey", true, true));
        return keyPair.generateKeyPair();
    }

    public static void main(final String[] args) throws Exception {
        try {
            Security.addProvider(new com.cavium.provider.CaviumProvider());
        } catch (Exception ex) {
            System.out.println(ex);
            return;
        }

        LoginManager lm = LoginManager.getInstance();
        lm.login("PARTITION_1", "cu1", "user1234");
        KeyPair rsaPair = rsa_key_pair();

        // generate a signature
        Signature s = Signature.getInstance("SHA256withRSAandMGF1", "Cavium");
        s.initSign(rsaPair.getPrivate());
        byte[] message = new byte[] { (byte)'a', (byte)'b', (byte)'c' };
        s.update(message);
        byte[] sigBytes = s.sign();

        // verify a signature
        s.initVerify(rsaPair.getPublic());
    }
}

```

```

    s.update(message);
    if (s.verify(sigBytes))
    {
        System.out.println("signature verification succeeded.");
    }
    else
    {
        System.out.println("signature verification failed.");
    }
    lm.logout();
}
}

```

Sign and Verify Using PSS Parameter Spec

This example uses the PSS parameter spec and performs sign and verify.

```

/**
 * Example of signing and verification using PSS parameter spec.
 */
public class c {
    public static KeyPair rsa_key_pair() throws Exception {
        KeyPairGenerator keyPair = KeyPairGenerator.getInstance("RSA","Cavium");
        keyPair.initialize(new CaviumRSAKeyGenParameterSpec(2048,new BigInteger("65537"),"publickey", "privatekey",true,true));
        return keyPair.generateKeyPair();
    }

    public static void main(final String[] args) throws Exception {
        try {
            Security.addProvider(new com.cavium.provider.CaviumProvider());
        } catch (Exception ex) {
            System.out.println(ex);
            return;
        }
    }
}

```

```
LoginManager lm = LoginManager.getInstance();
lm.login("PARTITION_1", "cu1", "user1234");
PSSParameterSpec spec= new PSSParameterSpec("SHA-256", "MGF1", new MGF1ParameterSpec("SHA-256"),0, 1);
KeyPair rsaPair = rsa_key_pair();

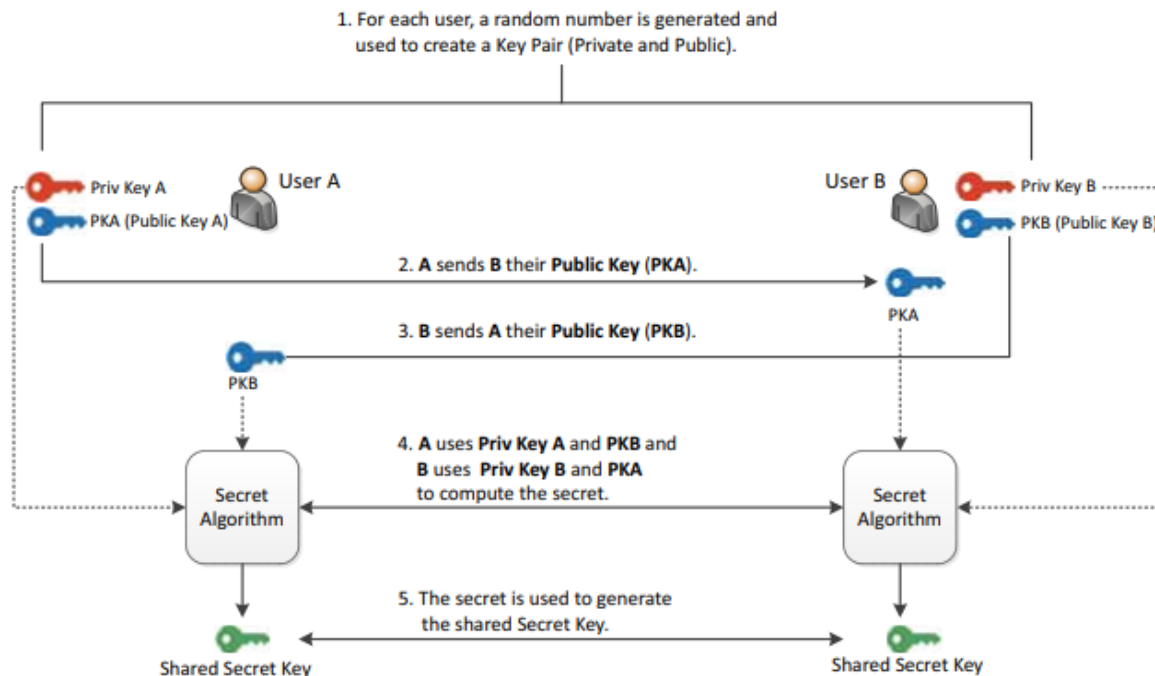
// generate a signature
Signature s = Signature.getInstance("SHA256WithRSA/PSS", "Cavium");
s.initSign(rsaPair.getPrivate());
byte[] message = new byte[] { (byte)'a', (byte)'b', (byte)'c' };
s.setParameter(spec);
s.update(message);
byte[] sigBytes = s.sign();

// verify a signature
s.initVerify(rsaPair.getPublic());
s.setParameter(spec);
s.update(message);
if (s.verify(sigBytes))
{
    System.out.println("signature verification succeeded.");
}
else
{
    System.out.println("signature verification failed.");
}
lm.logout();
}
```

Key Exchange Using Diffie-Hellman

The Azure Cloud HSM JCE provider supports the Diffie-Hellman (DH) method of securely exchanging cryptographic keys over an insecure channel to generate a shared, secret key. The secret key can then be used to encrypt future communication.

This section explains the procedure and provides an example program for performing an exchange using the Azure Cloud HSM JCE Provider.



In this example, there are two users (A and B). The key exchange includes the following steps:

1. A and B each select a random secret number and Public/Private key pairs are generated.
2. A sends its public key (PKA) to B
3. B sends its public key (PKB) to A.
4. A computes the secret, and B computes the secret.

5. The secret key is generated on both A and B sides.

From a JCE perspective, this procedure involves calling the three functions. The Key Agreement class divides the protocol up into three classes of methods:

init()	Takes the caller's private key and (optionally) any other parameter.
doPhase()	Takes another party's public key and a boolean. When the boolean is true, it indicates the protocol has completed successfully.
generateSecret()	Returns the shared secret.

The sample application below demonstrates a Diffie-Hellman key exchange.

```
/**
 * Test Program to illustrate Diffie-Hellman exchange.
 * */
public class BasicECDH {
    public static KeyPair generateECKeypair( String provider ) throws Exception {
        KeyPairGenerator kpg1 = KeyPairGenerator.getInstance("EC", provider);
        ECGenParameterSpec ecSpec = new ECGenParameterSpec(CaviumECGenParameterSpec.PRIME256);
        Provider kpgenProv = kpg1.getProvider();
        System.out.printf("Provider : %s%nInfo: %s%n", kpgenProv.getName(),
            kpgenProv.toString());
        kpg1.initialize(new CaviumECGenParameterSpec("prime256v1", "ec_pub", "ec_priv", true, true));
        return kpg1.generateKeyPair();
    }

    public static void main(final String[] args) throws Exception {
        try {
            Security.addProvider(new com.cavium.provider.CaviumProvider());
        } catch (Exception ex) {
            System.out.println(ex);
        }
        return;
    }
}
```

```

LoginManager lm = LoginManager.getInstance();
lm.login("PARTITION_1", "cu1", "user1234");

PrivateKey ec_pk;
PublicKey ec_pbk;
KeyPair kp1 = generateECCKeypair("Cavium");

KeyAgreement keyAgree1 = KeyAgreement.getInstance("ECDH", "Cavium");
keyAgree1.init(kp1.getPrivate());
KeyPair kp2 = generateECCKeypair("Cavium");

KeyAgreement keyAgree2 = KeyAgreement.getInstance("ECDH", "Cavium");
keyAgree2.init(kp2.getPrivate());
keyAgree1.doPhase(kp2.getPublic(), true);
keyAgree2.doPhase(kp1.getPublic(), true);

byte[] secret1 = keyAgree1.generateSecret();
byte[] secret2 = keyAgree2.generateSecret();
System.out.println(Arrays.equals(secret1, secret2));

lm.logout();
}
}

```

Compilation Instructions

To compile the program, run the following command:

```
javac -cp azure-cloud-hsm-jca-11-jar-with-dependencies.jar: BasicECDH.java
```

Running the Executable

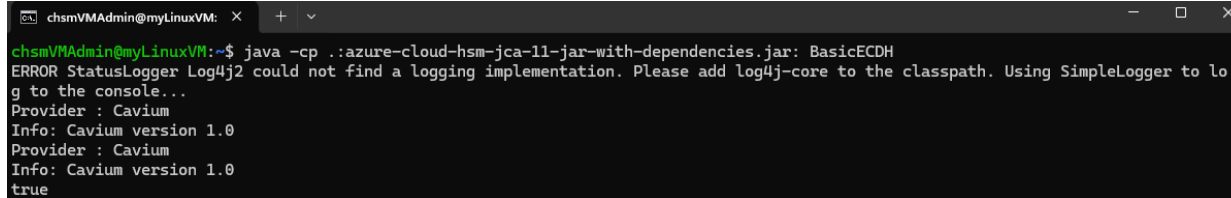
To execute the program, run the following command:

```
java -cp .:azure-cloud-hsm-jca-11-jar-with-dependencies.jar: BasicECDH
```

Sample Output

Important Note: It is important that the output returns true; if it does not, the handshake has failed.

Output will look like the following:

A terminal window titled 'chsmVMAdmin@myLinuxVM' showing the execution of a Java command. The command is 'java -cp .:azure-cloud-hsm-jca-11-jar-with-dependencies.jar: BasicECDH'. The output shows an error message from StatusLogger Log4j2, followed by two lines of information: 'Provider : Cavium' and 'Info: Cavium version 1.0', and finally 'true' on the last line.

```
chsmVMAdmin@myLinuxVM:~$ java -cp .:azure-cloud-hsm-jca-11-jar-with-dependencies.jar: BasicECDH
ERROR StatusLogger Log4j2 could not find a logging implementation. Please add log4j-core to the classpath. Using SimpleLogger to log to the console...
Provider : Cavium
Info: Cavium version 1.0
Provider : Cavium
Info: Cavium version 1.0
true
```

Managing Azure Cloud HSM Keystore

Important Note: The Azure Cloud HSM provider supports two key stores: CAVIUM and LSKS. The CAVIUM key store does not implement most of the key store methods, making it incompatible with KeyTool. Therefore, it is recommended to use the LSKS key store instead of the CAVIUM key store.

Important Note: The Java Keytool is part of the Java Development Kit (JDK) and is owned by Oracle Corporation. **Microsoft does not recommend using Key Tool**, because it extracts keys from keystore to perform sign operations using a standard provider (i.e Sun, SunJCE) pre-installed with JDK. While instructions below detail configuration for Keystore please note that **Keytool does not support or allow customers to use custom JCE providers (Cavium Marvell Provider, Cloud HSM JCE)**. Which is why if keys are set to non-extractable, it fails because export key fails. Since the keys are taken out of HSM boundary, we do not recommend using Key Tool. Even if you explicitly give higher priority, Key Tool will not use the Cloud HSM JCE provider for generating signatures.

The LSKS key store does not work with existing keys in Azure Cloud HSM if they were generated using other utilities, such as PKCS11 or azcloudhsm_util. For existing keys to work with the LSKS key store, key objects should be explicitly created using key handles and stored in the key store file using the setEntry or setKeyEntry methods.

- **LiquidSecurity Keystore (LSKS)** – The LiquidSecurity keystore is a read-write keystore that supports all interface operations. It stores keys to a stream provided by the caller to the store method. When used with standard tools like keytool, keys are stored in a file.

Only keys stored explicitly using a setEntry method followed by a keystore store operation will be persisted, and only these will be presented when getting a list of aliases or a count of the aliases. Individual operations, like getting a single entry or checking for the presence of a key, will still fall back to the HSM. Notably, ALIAS (KeyTool) and LABEL (PKCS#11) are the same.

- **Cavium Keystore** – The Cavium (Marvell) keystore is a read-only keystore that will always go to the HSM to get information about keys. The keystore will display a self-signed certificate for all supported private keys, and keys are searchable even if they have never been explicitly saved to the keystore using setEntry, etc.

Distinctive behavior:

- The Keystore.size method is not accurate and returns a count of all key entries on the HSM.
- The getKey/getEntry methods will return the first entry found if there are duplicate labels.

This section provides instructions for working with the Azure Cloud HSM keystore.

About the keytool Utility

Important Note: Keytool has limitations that make it unsuitable for high-security or production environments. Its private keys, stored in Java Keystores (JKS), can be extracted in plaintext. It supports fewer algorithms and key sizes compared to tools like OpenSSL and relies on the outdated SHA-1-based JKS format, which weakens security. Keytool lacks secure audit logging, a critical feature for regulated environments. Microsoft advises against using Keytool in high-security and production environments.

The keytool is a Java utility that provides for command-line processing of keystores. Syntax: # keytool

The utility includes the following commands to manage keys and certificates:

Command	Description
-certreq	Generates a certificate request
-changealias	Changes an entry alias
-delete	Deletes an entry
-exportcert	Exports certificate
-genkeypair	Generates a key pair
-genseckey	Generates a secret key
-gencert	Generates certificate from a certificate request
-importcert	Imports a certificate or a certificate chain
-importpass	Imports a password
-importkeystore	Imports one or all entries from another keystore
-keypasswd	Changes the key password of an entry
-list	Lists entries in a keystore

-printcert	Prints the content of a certificate
-printcertreq	Prints the content of a certificate request
-printcrl	Prints the content of a CRL file
-storepasswd	Changes the store password of a keystore

Use keytool - -help for usage of a specific command.

Add Azure Cloud HSM JCE provider to Java runtime security configuration.

Add the Azure Cloud HSM JCE provider to your Java runtime's security configuration. Edit the java.security file located in the \$JAVA_HOME/conf/security directory. In the example below the directory is /usr/lib/jvm/java-11-openjdk-amd64/conf/security

Add the following line to include the Azure Cloud HSM JCE provider, replacing **N** with the next available provider number.

```
security.provider.N=com.cavium.provider.CaviumProvider
```

Important Note: Once you made changes to the java.security file, restart your Java process to apply the changes.

Example

```
#
# List of providers and their preference orders (see above):
#
security.provider.1=SUN
security.provider.2=SunRsaSign
security.provider.3=SunEC
security.provider.4=SunJSSE
security.provider.5=SunJCE
security.provider.6=SunJGSS
security.provider.7=SunSASL
security.provider.8=XMLDSig
security.provider.9=SunPCSC
security.provider.10=JdkLDAP
security.provider.11=JdkSASL
security.provider.12=SunPKCS11
security.provider.13=com.cavium.provider.CaviumProvider
```

Update path and environment variables for keytool use.

The following environment variables must be set up for Keytool to function properly. Examples of how to edit the .bashrc file (for non-login shells) or .bash_profile (for login shells) using a text editor can be found in the system requirements section of this document to ensure environment variables are persistent across terminal sessions for production.

```
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64/  
export PATH=$JAVA_HOME/bin:$PATH  
export HSM_USER=cu1  
export HSM_PASSWORD=user1234  
export HSM_PARTITION=PARTITION_1
```

Generating a Key Pair with keytool

Important Note: Keytool generated private keys are plaintext extractable. Microsoft advises against using Keytool in high-security and production environments.

Use the following command to generate an RSA key pair with keytool:

```
keytool -genkeypair -alias rsaKey1 -keyalg RSA -keysize 2048 -keystore myKeystore -storepass cavium -keypass cavium -storetype LSKS -  
providerpath azure-cloud-hsm-jca-11-jar-with-dependencies.jar -providerclass com.cavium.provider.CaviumProvider
```

Use the following command to generate an EC key pair with keytool:

```
keytool -genkeypair -alias eckey1 -keyalg EC -groupname secp256r1 -keystore myKeystore -storepass cavium -keypass cavium -storetype LSKS -  
providerpath azure-cloud-hsm-jca-11-jar-with-dependencies.jar -providerclass com.cavium.provider.CaviumProvider
```

Output:

```
chsmVMAdmin@AdminLinuxVM:~$ keytool -genkeypair -alias key1 -keyalg RSA -keysize 2048 -keystore rsa.jks -storepass cavium -keypass cavium -storetype LSKS -providerpath azure-cloud-hsm-jca-11-jar-with-dependencies.jar -providerclass com.cavium.provider.CaviumProvider

What is your first and last name?
[Unknown]: John Smith
What is the name of your organizational unit?
[Unknown]: Org Unit
What is the name of your organization?
[Unknown]: My Org
What is the name of your City or Locality?
[Unknown]: My City
What is the name of your State or Province?
[Unknown]: My State
What is the two-letter country code for this unit?
[Unknown]: XX
Is CN=John Smith, OU=Org Unit, O=My Org, L=My City, ST=My State, C=XX correct?
[no]: yes
```

Listing a Key with keytool

Use the following command to list a key. In this example we are going to specify alias `rsaKey1` which we created above in keystore `myKeystore`.

```
keytool -list -alias rsaKey1 -keystore myKeystore -storepass cavium -keypass cavium -storetype LSKS -providerpath azure-cloud-hsm-jca-11-jar-with-dependencies.jar -providerclass com.cavium.provider.CaviumProvider
```

If you created an EC key pair you will use the following command to list a key.

```
keytool -list -alias eckey1 -keystore myKeystore -storepass cavium -keypass cavium -storetype LSKS -providerpath azure-cloud-hsm-jca-11-jar-with-dependencies.jar -providerclass com.cavium.provider.CaviumProvider
```

Output:

```
chsmVMAdmin@AdminLinuxVM:~$ keytool -list -alias key1 -keystore rsa.jks -storepass cavium -keypass cavium -storetype LSKS -providerpath azure-cloud-hsm-jca-11-jar-with-dependencies.jar -providerclass com.cavium.provider.CaviumProvider

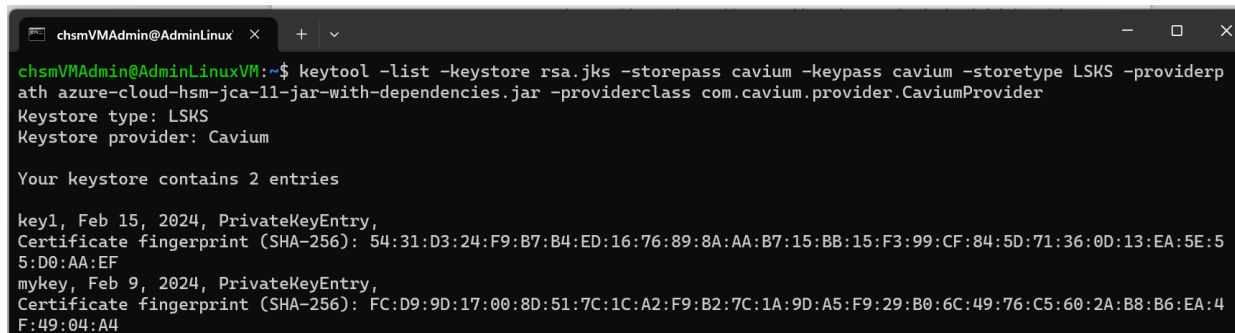
key1, Feb 15, 2024, PrivateKeyEntry,
Certificate fingerprint (SHA-256): 54:31:D3:24:F9:B7:B4:ED:16:76:89:8A:AA:B7:15:BB:15:F3:99:CF:84:5D:71:36:0D:13:EA:5E:55:D0:AA:EF
```

Listing all Keys with keytool

Use the following command to list all keys. In this example we are going to not use alias and just specify list against keystore `myKeystore`

```
keytool -list -keystore myKeystore -storepass cavium -keypass cavium -storetype LSKS -providerpath azure-cloud-hsm-jca-11-jar-with-dependencies.jar -providerclass com.cavium.provider.CaviumProvider
```

Output:

A terminal window titled 'chsmVMAdmin@AdminLinux' showing the output of the 'keytool -list' command. The command is: 'keytool -list -keystore rsa.jks -storepass cavium -keypass cavium -storetype LSKS -providerpath azure-cloud-hsm-jca-11-jar-with-dependencies.jar -providerclass com.cavium.provider.CaviumProvider'. The output shows the keystore type as LSKS and the provider as Cavium. It then lists two entries: 'key1' and 'mykey', both created on Feb 15, 2024, and Feb 9, 2024 respectively, both of type PrivateKeyEntry. The SHA-256 fingerprints are displayed for each entry.

```
chsmVMAdmin@AdminLinux:~$ keytool -list -keystore rsa.jks -storepass cavium -keypass cavium -storetype LSKS -providerpath azure-cloud-hsm-jca-11-jar-with-dependencies.jar -providerclass com.cavium.provider.CaviumProvider
Keystore type: LSKS
Keystore provider: Cavium

Your keystore contains 2 entries

key1, Feb 15, 2024, PrivateKeyEntry,
Certificate fingerprint (SHA-256): 54:31:D3:24:F9:B7:B4:ED:16:76:89:8A:AA:B7:15:BB:15:F3:99:CF:84:5D:71:36:0D:13:EA:5E:55:D0:AA:EF
mykey, Feb 9, 2024, PrivateKeyEntry,
Certificate fingerprint (SHA-256): FC:D9:9D:17:00:8D:51:7C:1C:A2:F9:B2:7C:1A:9D:A5:F9:29:B0:6C:49:76:C5:60:2A:B8:B6:EA:4F:49:04:A4
```

Replacing a Self-Signed Certificate with a Certificate from a CA

After generating a key with a self-signed certificate, users often want to get a certificate from a Certification Authority to replace the self-signed certificate.

Generate an RSA Key Pair.

```
keytool -genkeypair -alias key2 -keyalg RSA -keysize 2048 -dname "cn=Cloud.HSM, ou=Engineering, o=Microsoft, c=US" -keystore myKeystore -storepass cavium -keypass cavium -storetype LSKS -providerclass com.cavium.provider.CaviumProvider -providerpath azure-cloud-hsm-jca-11-jar-with-dependencies.jar
```

Generate an EC Key Pair.

```
keytool -genkeypair -alias key3 -keyalg EC -groupname secp256r1 -keystore myKeystore -storepass cavium -keypass cavium -storetype LSKS -providerclass com.cavium.provider.CaviumProvider -providerpath azure-cloud-hsm-jca-11-jar-with-dependencies.jar
```

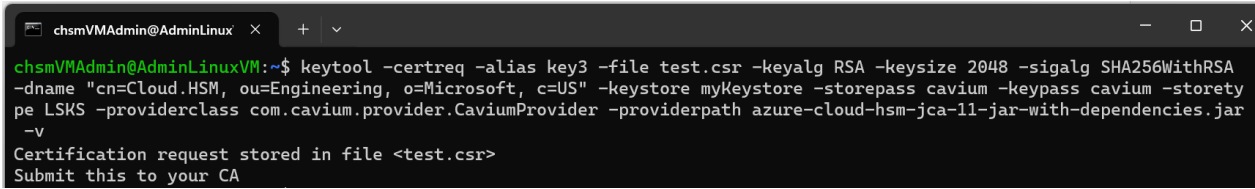
Run the following command to generate the RSA CSR.

```
keytool -certreq -alias key2 -file test.csr -keyalg RSA -keysize 2048 -sigalg SHA256WithRSA -dname "cn=Cloud.HSM, ou=Engineering, o=Microsoft, c=US" -keystore myKeystore -storepass cavium -keypass cavium -storetype LSKS -providerclass com.cavium.provider.CaviumProvider -providerpath azure-cloud-hsm-jca-11-jar-with-dependencies.jar -v
```


Run the following command to generate the EC CSR.

```
keytool -certreq -alias key3 -file test.csr -keyalg EC -sigalg SHA256withECDSA -keystore myKeystore -storepass cavium -keypass cavium -storetype LSKS -providerclass com.cavium.provider.CaviumProvider -providerpath azure-cloud-hsm-jca-11-jar-with-dependencies.jar -v
```

Output:



```
chsmVMAdmin@AdminLinuxVM:~$ keytool -certreq -alias key3 -file test.csr -keyalg RSA -keysize 2048 -sigalg SHA256WithRSA -dnname "cn=Cloud.HSM, ou=Engineering, o=Microsoft, c=US" -keystore myKeystore -storepass cavium -keypass cavium -storetype LSKS -providerclass com.cavium.provider.CaviumProvider -providerpath azure-cloud-hsm-jca-11-jar-with-dependencies.jar -v
Certification request stored in file <test.csr>
Submit this to your CA
```

Next, sign the CSR and send it to the trust anchor. (For testing purposes, create and sign the CSR locally).

1. **Generate the trustanchor.crt and trustanchor.key**

```
openssl req -newkey rsa:2048 -nodes -keyout trustanchorRSA.key -x509 -days 365 -out trustanchorRSA.crt
openssl x509 -days 365 -req -in test.csr -CA trustanchorRSA.crt -CAkey trustanchorRSA.key -set_serial 01 -out signed.crt
```

For an EC key generate the trustanchor.crt and trustanchor.key using the following command.

```
openssl ecparam -name prime256v1 -genkey -noout -out trustanchorEC.key
openssl req -new -x509 -key trustanchorEC.key -out trustanchorEC.crt -days 365
openssl x509 -days 365 -req -in test.csr -CA trustanchorEC.crt -CAkey trustanchorEC.key -set_serial 01 -out signed.crt
```

Output:

```
chsmVMAdmin@AdminLinuxVM:~$ openssl req -newkey rsa:2048 -nodes -keyout trustanchor.key -x509 -days 365 -out trustanchor.crt
Generating a RSA private key
.....+++++
.....+++++
writing new private key to 'trustanchor.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Microsoft
Organizational Unit Name (eg, section) []:Engineering
Common Name (e.g. server FQDN or YOUR name) []:Cloud.HSM
Email Address []:
chsmVMAdmin@AdminLinuxVM:~$ openssl x509 -days 365 -req -in test.csr -CA trustanchor.crt -CAkey trustanchor.key -set_serial 01 -out signed.crt
Signature ok
subject=C = US, O = Microsoft, OU = Engineering, CN = Cloud.HSM
Getting CA Private Key
```

2. Import the trust anchor as a trusted certificate into the keystore as follows:

```
keytool -importcert -alias caname -file trustanchorRSA.crt -keystore myKeystore -storepass cavium -keypass cavium -storetype LSKS -
providerclass com.cavium.provider.CaviumProvider -providerpath azure-cloud-hsm-jca-11-jar-with-dependencies.jar -v
```

Output:

```
chsmVMAAdmin@AdminLinuxVM:~$ keytool -importcert -alias caname -file trustanchor.crt -keystore myKeystore -storepass cavi
um -keypass cavium -storetype LSKS -providerclass com.cavium.provider.CaviumProvider -providerpath azure-cloud-hsm-jca-1
1-jar-with-dependencies.jar -v
Owner: CN=Cloud.HSM, OU=Engineering, O=Microsoft, ST=Some-State, C=US
Issuer: CN=Cloud.HSM, OU=Engineering, O=Microsoft, ST=Some-State, C=US
Serial number: 1fe9d7061005cd0cac9d97b7ef1e25b49dcad48b
Valid from: Thu Feb 15 20:39:22 UTC 2024 until: Fri Feb 14 20:39:22 UTC 2025
Certificate fingerprints:
    SHA1: 3A:D2:37:0B:F2:DE:74:02:39:68:94:28:CD:BA:D8:6A:F3:A2:68:EB
    SHA256: 91:07:B7:E6:EC:33:FA:DE:79:06:3A:11:2A:B2:FE:A5:C0:EB:E3:63:82:62:25:CD:A8:8F:A3:D0:83:20:8A:1E
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 3

Extensions:

#1: ObjectId: 2.5.29.35 Criticality=false
AuthorityKeyIdentifier [
KeyIdentifier [
0000: F8 1E 2A 09 4A DE E4 C6 7D EE B7 5B 72 22 7F 92 ..*.J.....[r"..
0010: 9B B8 63 2F ..c/
]
]

#2: ObjectId: 2.5.29.19 Criticality=true
BasicConstraints:[
CA:true
PathLen:2147483647
]

#3: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: F8 1E 2A 09 4A DE E4 C6 7D EE B7 5B 72 22 7F 92 ..*.J.....[r"..
0010: 9B B8 63 2F ..c/
]
]

Trust this certificate? [no]: yes
Certificate was added to keystore
[Storing myKeystore]
```

3. Update the response.

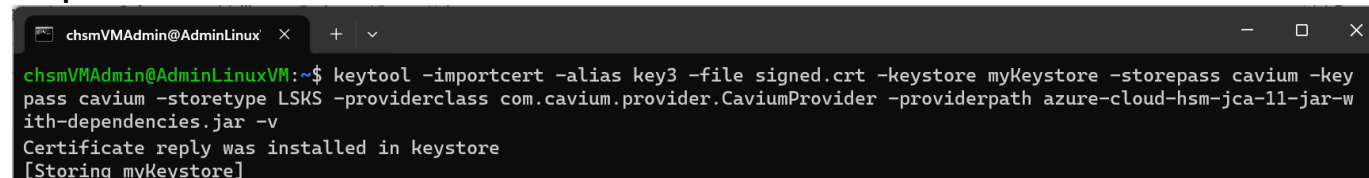
This time, the response is imported using the same alias used to generate the original RSA key.

```
keytool -importcert -alias key2 -file signed.crt -keystore myKeystore -storepass cavium -keypass cavium -storetype LSKS -providerclass
com.cavium.provider.CaviumProvider -providerpath azure-cloud-hsm-jca-11-jar-with-dependencies.jar -v
```

If you used the example to create an EC key, run the following command below. The response is imported using the same alias used to generate the original EC key.

```
keytool -importcert -alias key3 -file signed.crt -keystore myKeystore -storepass cavium -keypass cavium -storetype LSKS -providerclass com.cavium.provider.CaviumProvider -providerpath azure-cloud-hsm-jca-11-jar-with-dependencies.jar -v
```

Output:

A terminal window titled 'chsmVMAdmin@AdminLinux' with a dark background. The prompt is 'chsmVMAdmin@AdminLinuxVM:~\$'. The command entered is 'keytool -importcert -alias key3 -file signed.crt -keystore myKeystore -storepass cavium -keypass cavium -storetype LSKS -providerclass com.cavium.provider.CaviumProvider -providerpath azure-cloud-hsm-jca-11-jar-with-dependencies.jar -v'. The output is 'Certificate reply was installed in keystore' followed by '[Storing myKeystore]' on the next line.

```
chsmVMAdmin@AdminLinuxVM:~$ keytool -importcert -alias key3 -file signed.crt -keystore myKeystore -storepass cavium -keypass cavium -storetype LSKS -providerclass com.cavium.provider.CaviumProvider -providerpath azure-cloud-hsm-jca-11-jar-with-dependencies.jar -v
Certificate reply was installed in keystore
[Storing myKeystore]
```

To verify the key was stored, list the key:

```
keytool -list -keystore myKeystore -storepass cavium -keypass cavium -storetype LSKS -providerclass com.cavium.provider.CaviumProvider -providerpath azure-cloud-hsm-jca-11-jar-with-dependencies.jar -v
```

Output: Note that the Owner and the Issuer are the same.

```
chsmVMAAdmin@AdminLinux x + v
Alias name: key3
Creation date: Feb 15, 2024
Entry type: PrivateKeyEntry
Certificate chain length: 2
Certificate[1]:
Owner: CN=Cloud.HSM, OU=Engineering, O=Microsoft, C=US
Issuer: CN=Cloud.HSM, OU=Engineering, O=Microsoft, ST=Some-State, C=US
Serial number: 1
Valid from: Thu Feb 15 20:39:33 UTC 2024 until: Fri Feb 14 20:39:33 UTC 2025
Certificate fingerprints:
    SHA1: 5F:F9:60:2B:5D:7D:65:9E:5F:44:65:FF:38:49:1A:0B:41:EE:89:46
    SHA256: 14:A1:30:32:20:60:9D:CC:E9:A2:FB:3A:81:4D:CF:7C:2C:91:50:3E:D1:AE:B0:69:97:30:ED:8B:26:83:8F:36
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 1
Certificate[2]:
Owner: CN=Cloud.HSM, OU=Engineering, O=Microsoft, ST=Some-State, C=US
Issuer: CN=Cloud.HSM, OU=Engineering, O=Microsoft, ST=Some-State, C=US
Serial number: 1fe9d7061005cd0cac9d97b7ef1e25b49dcad48b
Valid from: Thu Feb 15 20:39:22 UTC 2024 until: Fri Feb 14 20:39:22 UTC 2025
Certificate fingerprints:
    SHA1: 3A:D2:37:0B:F2:DE:74:02:39:68:94:28:CD:BA:D8:6A:F3:A2:68:EB
    SHA256: 91:07:B7:E6:EC:33:FA:DE:79:06:3A:11:2A:B2:FE:A5:C0:EB:E3:63:82:62:25:CD:A8:8F:A3:D0:83:20:8A:1E
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 3

Extensions:

#1: ObjectId: 2.5.29.35 Criticality=false
AuthorityKeyIdentifier [
KeyIdentifier [
0000: F8 1E 2A 09 4A DE E4 C6 7D EE B7 5B 72 22 7F 92  ..*.J.....[r"..
0010: 9B B8 63 2F                                     ..c/
]
]

#2: ObjectId: 2.5.29.19 Criticality=true
BasicConstraints:[
CA:true
PathLen:2147483647
]

#3: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: F8 1E 2A 09 4A DE E4 C6 7D EE B7 5B 72 22 7F 92  ..*.J.....[r"..
0010: 9B B8 63 2F                                     ..c/
]
]
```

Signing and Verifying a Jar File in the LSKS

jarsigner is a command-line tool that comes with the Java Development Kit (JDK), and it is used to sign Java Archive (JAR) files and verify the signatures of signed JAR files.

Important Note: The jarsigner tool does not allow for the specification of a path where the provider can be found. Consequently, to use the jarsigner you must copy the Azure Cloud HSM JCE jar file to a standard location; for example, \$JAVA_HOME/lib.

In addition, jarsigner only uses the specified provider to read the keystore. It does not use it to instantiate the signature implementation. To do this, you must configure the JRE security configuration to prioritize the Azure Cloud HSM provider above other providers (in this case, the SunRSASign provider)

Signing a JAR File:

1. Generate a Keystore:

If you don't have a keystore containing your private key, you can create one using the keytool command:

```
keytool -genkeypair -alias key4 -keyalg RSA -keysize 2048 -keystore myKeystore -storepass cavium -keypass cavium -storetype LSKS -  
providerpath azure-cloud-hsm-jca-11-jar-with-dependencies.jar -providerclass com.cavium.provider.CaviumProvider
```

2. Sign the JAR file:

Use jarsigner to sign your JAR file with the private key from the keystore. In this example we created a test.jar file to sign against.

```
jarsigner -J-classpath '-J/opt/azurecloudhsm/jce/jce-11/azcloudhsm_jca-11-jar-with-dependencies.jar' -providerclass  
com.cavium.provider.CaviumProvider -keystore myKeystore -storepass cavium -keypass cavium -storetype LSKS -sigalg SHA256withRSA -  
digestalg SHA-256 test.jar key4
```

3. Verifying a Signed JAR file:

To verify the signature of a signed JAR file, use the following command:

```
jarsigner -J-classpath '-J/opt/azurecloudhsm/jce/jce-11/azcloudhsm_jca-11-jar-with-dependencies.jar' -verify -keystore myKeystore -  
keypass cavium -storepass cavium -certs test.jar -providerclass com.cavium.provider.CaviumProvider
```

Key Factories

The Java Key Factory (`java.security.KeyFactory`) converts keys back and forth between keys and key specifications. It provides a method to take a key created by the JavaJCE and turn it into an Azure Cloud HSM key specification, with the result that the key can be imported into the Azure Cloud HSM. Below details the requirements for using the Java key factory with the Azure Cloud HSM JCE Provider and provides examples for translating different types of keys.

Supported Features and Requirements

The Azure Cloud HSM implementation features and requirements for different types of keys are listed in the following table:

Key Factory Supported Features/Requirements

Key Type	Requirements
RSA	<ol style="list-style-type: none">1. When calling <i>generatePublic</i>, only <i>RSAPublicKeySpec</i> or <i>X509EncodedKeySpec</i> instances are supported.2. When calling <i>generatePrivate</i>, only <i>RSAPrivateCrtKeySpec</i> or <i>PKCS8EncodedKeySpec</i> instances are supported.3. When calling <i>translateKey</i>, only RSA key are supported.4. This class is only known to work with the RSA keys generated by the SunRSASign provider.5. The return value of <i>translateKey</i>, <i>generatePublicKey</i>, and <i>generatePrivateKey</i> methods will be an instance of <i>CaviumKey</i>.6. <i>translateKey</i>, <i>generatePublicKey</i>, and <i>generatePrivateKey</i> internally generate a key with attributes as <i>extractable = true</i>, <i>sessionkey = true</i>, and a random label.
EC	<ol style="list-style-type: none">1. When calling <i>generatePublic</i>, only <i>ECPublicKeySpec</i> or <i>X509EncodedKeySpec</i> instances are supported.2. When calling <i>generatePrivate</i>, only <i>ECPrivateKeySpec</i> or <i>PKCS8EncodedKeySpec</i> instances are supported.3. When calling <i>translateKey</i>, only EC keys are supported.4. This class is only known to work with the EC keys generated by the SunEC provider.5. The return value of <i>translateKey</i>, <i>generatePublicKey</i>, and <i>generatePrivateKey</i> methods will be an instance of <i>CaviumKey</i>.6. <i>translateKey</i>, <i>generatePublicKey</i>, and <i>generatePrivateKey</i> internally generate a key with attributes as <i>extractable = true</i>, <i>sessionkey = true</i>, and a random label

AES	<ol style="list-style-type: none"> 1. When calling <i>generateSecret</i>, only <i>SecretKeySpec</i> with algorithm set to "AES" is supported. 2. When calling <i>translateKey</i> only AES keys are supported. 3. This class is only known to work with the AES keys generated by the SunJCE provider. 4. The return value of <i>translateKey</i> and <i>generateKey</i> is an instance of <i>CaviumKey</i>. 5. <i>translateKey</i> and <i>generateKey</i> internally generate a key with attributes as <i>extractable = true</i>, <i>sessionkey = true</i>, and a random label.
DESede	<ol style="list-style-type: none"> 1. When calling <i>generateSecret</i>, only <i>DESedeKeySpec</i> instances can be used. 2. When calling <i>translateKey</i>, only DESede keys are supported. 3. This class is only known to work with the DESede keys generated by the SunJCE provider. 4. The return value of <i>translateKey</i> and <i>generateKey</i> is an instance of <i>CaviumKey</i>. 5. <i>translateKey</i> and <i>generateKey</i> internally generate a key with attributes as <i>extractable = true</i>, <i>sessionkey = true</i>, and a random label
Generic	<ol style="list-style-type: none"> 1. When calling <i>generateSecret</i>, <i>SecretKeySpec</i> with algorithm set to "TlsRsaPremasterSecret" , "TlsPremasterSecret", "GenericSecret", "hmacSha1", "hmacSha224", "hmacSha256", "hmacSha384", and "hmacSha512" is supported. 2. When calling <i>translateKey</i>, <i>TlsRsaPremasterSecret</i>, <i>TlsPremasterSecret</i>, <i>GenericSecret</i>, <i>hmacSha1</i>, <i>hmacSha224</i>, <i>hmacSha256</i>, <i>hmacSha384</i>, and <i>hmacSha512</i> keys is supported. 3. Supported key sizes are 1 to 800 bytes only. 4. This class is only known to work with the above keys generated by the SunJCE provider. 5. The return values of <i>translateKey</i> and <i>generateKey</i> are instances of <i>CaviumKey</i>. 6. <i>translateKey</i> and <i>generateKey</i> internally generate a key with attributes as <i>extractable = true</i>, <i>sessionkey = false</i>, <i>key algorithm = "GenericSecret"</i>, and a random label.
XDH	<ol style="list-style-type: none"> 1. When calling <i>generateSecret</i>, only <i>CaviumXDHKeyGenParameterSpec</i> with algorithm set to "XDH" is supported.

Key Factory Examples

Example 1: SunJCE AES and DES Keys

The following sample code generates the key with the SunJCE, translates the key and imports it key into the Azure Cloud HSM. It includes examples for AES and DESede keys.

```
/**
 *Test Program to illustrate using SunJCE and importing key into HSM.
```



```

**/
public class KeyFactory {
    public static void main(final String[] args) throws Exception { try {
        Security.addProvider(new com.cavium.provider.CaviumProvider());
    } catch (Exception ex) {
        System.out.println(ex);
        return;
    }

    LoginManager lm = LoginManager.getInstance();
    lm.login("PARTITION_1", "cu1", "user1234");
    SecretKeyFactory factory = SecretKeyFactory.getInstance("AES", "Cavium");
    KeyGenerator kg = KeyGenerator.getInstance("AES", "SunJCE");
    SecretKey sk = kg.generateKey();
    SecretKey csk = (SecretKey) factory.translateKey(sk);
    Scanner input = new Scanner(System.in);
    int num = input.nextInt();

    factory = SecretKeyFactory.getInstance("DESede", "Cavium");
    kg = KeyGenerator.getInstance("DESede", "SunJCE");
    sk = kg.generateKey();
    csk = (SecretKey) factory.translateKey(sk);
    CaviumKey cavkey = (CaviumKey) csk;
    input = new Scanner(System.in);
    num = input.nextInt();
    lm.logout();
    }
}

```

Example 2: Getting the Spec for AES and DESede Keys

The following example illustrates how to get the key spec for AES and DESede keys.

```

/**
 * Test Program to get the key spec for AES and DESede keys.
 **/
public class KeyFactorySpec {
    public static void main(final String[] args) throws Exception { try {
        Security.addProvider(new com.cavium.provider.CaviumProvider());
    } catch (Exception ex) {
        System.out.println(ex);
        return;
    }

    LoginManager lm = LoginManager.getInstance();
    lm.login("PARTITION_1", "cu1", "user1234");
    SecretKeyFactory factory = SecretKeyFactory.getInstance("DESede", "Cavium");
    KeyGenerator kg = KeyGenerator.getInstance("DESede", "Cavium");
    kg.init(new CaviumDESKeyGenParameterSpec(192, "testGetSpec", true, false));
    Key csk = kg.generateKey();
    DESedeKeySpec spck = (DESedeKeySpec) factory.getKeySpec((SecretKey) csk, DESedeKeySpec.class);

    //assertArrayEquals(spck.getKey(), csk.getEncoded());
    if (Arrays.equals(spck.getKey(), csk.getEncoded()) == false)
        System.out.println("FAIL - SPEC doesn't match\n");
    factory = SecretKeyFactory.getInstance("AES", "Cavium");
    kg = KeyGenerator.getInstance("AES", "Cavium");
    kg.init(new CaviumAESKeyGenParameterSpec(192, "testGetSpec", true, false));
    csk = kg.generateKey();
    SecretKeySpec sks = (SecretKeySpec) factory.getKeySpec((SecretKey) csk, SecretKeySpec.class);

    if (Arrays.equals(sks.getEncoded(), csk.getEncoded()) == false)
        System.out.println("FAIL - SPEC doesn't match\n");
    lm.logout();
}

```

```
}
```

Example 3: Translating a SunJCE Key to RSA Key

In this example, the RSA key is generated in the SunJCE and then translated using a key factory.

```
/**
 * Test Program to generate RSA key with SunJCE and translate using key factory.
 **/
public class KeyFactorySpec {
    public static void main(final String[] args) throws Exception { try {
        Security.addProvider(new com.cavium.provider.CaviumProvider());
    } catch (Exception ex) {
        System.out.println(ex);
        return;
    }

    LoginManager lm = LoginManager.getInstance();
    KeyPairGenerator sunKey = KeyPairGenerator.getInstance("RSA", "SunRsaSign");
    sunKey.initialize(2048);
    KeyPair sun = sunKey.generateKeyPair();
    KeyFactory factory = KeyFactory.getInstance("RSA", "Cavium");
    Key k1 = factory.translateKey(sun.getPrivate());

    if ( Arrays.equals(k1.getEncoded(), sun.getPrivate().getEncoded()) == false)
        System.out.println(" Key doesn't match\n");
    Key k2 = factory.translateKey(sun.getPublic());

    if ( Arrays.equals(k2.getEncoded(), sun.getPublic().getEncoded()) == false)
        System.out.println(" Key doesn't match\n");
    Scanner input = new Scanner(System.in);
    int num = input.nextInt();
}
```

```
}
```

Example 4: Generating a Key Spec from an RSA Key

The following code sample shows how to generate a private RSA key spec from an RSA key.

```
/**
 * Test Program to generate RSA key with SunJCE and translate using key factory.
 **/
public class KeyFactorySpec {
    public static void main(final String[] args) throws Exception { try {
        Security.addProvider(new com.cavium.provider.CaviumProvider());
    } catch (Exception ex) {
        System.out.println(ex);
        return;
    }

    LoginManager lm = LoginManager.getInstance();
    lm.login("PARTITION_1", "cu1", "user1234");
    KeyPairGenerator sunKey = KeyPairGenerator.getInstance("RSA", "SunRsaSign");
    sunKey.initialize(2048);
    KeyPair sun = sunKey.generateKeyPair();
    KeyFactory factory = KeyFactory.getInstance("RSA", "Cavium");
    Key k1 = factory.translateKey(sun.getPrivate());

    if ( Arrays.equals(k1.getEncoded(), sun.getPrivate().getEncoded()) == false)
        System.out.println(" Key doesn't match\n");
    Key k2 = factory.translateKey(sun.getPublic());

    if ( Arrays.equals(k2.getEncoded(), sun.getPublic().getEncoded()) == false)
        System.out.println(" Key doesn't match\n");
    Scanner input = new Scanner(System.in);
    int num = input.nextInt();
```

```
}  
}
```

How to list all Algorithms in Azure Cloud HSM JCE provider

The code provided below demonstrates how to enumerate all supported algorithm names for ciphers, key agreements, MACs, message digests, and signatures for Azure Cloud HSM JCE.

```
import java.security.Provider;  
import java.security.Provider.Service;  
import com.cavium.provider.CaviumProvider;  
import com.cavium.cfm2.*;  
import com.cavium.cfm2.CFM2Exception;  
import com.cavium.cfm2.LoginManager;  
import com.cavium.cfm2.Util;  
import com.cavium.cfm2.ImportKey;  
import com.cavium.key.CaviumAESKey;  
import com.cavium.key.CaviumKey;  
import com.cavium.key.CaviumKeyAttributes;  
import com.cavium.key.CaviumRSAPrivateKey;  
import com.cavium.key.CaviumRSAPublicKey;  
import com.cavium.key.CaviumKeyAttributes;  
import com.cavium.key.CaviumECPrivateKey;  
import com.cavium.key.CaviumECPublicKey;  
import com.cavium.key.parameter.CaviumRSAKeyGenParameterSpec;  
import com.cavium.key.parameter.CaviumAESKeyGenParameterSpec;  
import com.cavium.key.parameter.CaviumKeyGenAlgorithmParameterSpec;  
import com.cavium.key.parameter.CaviumECGenParameterSpec;  
import java.security.SignatureException;  
import java.util.*;  
import java.util.Arrays;  
import java.util.UUID;  
import java.util.Random;  
import java.util.Scanner;
```

```
import java.util.Enumeration;
import java.security.Key;
import java.security.KeyPairGenerator; //Java 8
import java.security.KeyPair;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.Signature;
import java.security.Security;
import java.security.interfaces.ECPrivateKey;
import java.security.interfaces.ECPublicKey;
import java.security.spec.ECGenParameterSpec;
import java.security.spec.ECPublicKeySpec;
import java.math.BigInteger;
import javax.crypto.KeyAgreement;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.Cipher;
import java.security.MessageDigest;

public class ListAllAlgorithms {
    /**
     * Print out the set entries, indented, one per line, with the name of the set
     * unindented appearing on the first line.
     *
     * @param setName the name of the set being printed
     * @param algorithms the set of algorithms associated with the given name
     */
    public static void printSet(String setName, Set algorithms)
    {
        System.out.println(setName + ":");
        if (algorithms.isEmpty())
        {
```

```

        System.out.println(" None available.");
    }
    else
    {
        Iterator it = algorithms.iterator();
        while (it.hasNext())
        {
            String name = (String)it.next();
            System.out.println(" " + name);
        }
    }
}

/**
 * List the available algorithm names for ciphers, key agreement, macs,
 * message digests and signatures.
 */
public static void main(final String[] args) throws Exception {
    try {
        Security.addProvider(new com.cavium.provider.CaviumProvider());
    } catch (Exception ex) {
        System.out.println(ex);
        return;
    }

    LoginManager lm = LoginManager.getInstance();

    int count=0;
    Provider provider = Security.getProvider("Cavium");
    Set ciphers = new HashSet();
    Set keyAgreements = new HashSet();
    Set macs = new HashSet();

```

```
Set messageDigests = new HashSet();
Set signatures = new HashSet();

if(provider != null)
{
    for (Iterator it = provider.keySet().iterator(); it.hasNext();)
    {
        String entry = (String)it.next();
        if (entry.startsWith("Alg.Alias."))
        {
            entry = entry.substring("Alg.Alias.".length());
        }
        if (entry.startsWith("Cipher."))
        {
            ciphers.add(entry.substring("Cipher.".length()));
        }
        else if (entry.startsWith("KeyAgreement."))
        {
            keyAgreements.add(entry.substring("KeyAgreement.".length()));
        }
        else if (entry.startsWith("Mac."))
        {
            macs.add(entry.substring("Mac.".length()));
        }
        else if (entry.startsWith("MessageDigest."))
        {
            messageDigests.add(entry.substring("MessageDigest.".length()));
        }
        else if (entry.startsWith("Signature."))
        {
            signatures.add(entry.substring("Signature.".length()));
        }
    }
}
```



```

    }
    printSet("Ciphers", ciphers);
    printSet("KeyAgreements", keyAgreements);
    printSet("Macs", macs);
    printSet("MessageDigests", messageDigests);
    printSet("Signatures", signatures);
    System.out.println("Name: " + provider.getName());
}
else
{
    System.err.println("No Algorithm present in this provider");
}
}
}

```

Notes on Azure Cloud HSM AES-GCM Implementation

The Azure Cloud HSM GCM implementation is distinctive due to HSM FIPS requirements and other implementation details, specifically:

1. In encrypt and decrypt mode, if a GCMPParameterSpec is supplied, it must specify a 64/96/128-bit tag length and a 12-byte IV.
2. In wrap and unwrap mode, if a GCMPParameterSpec is supplied, it must specify a 64/96/128-bit tag length and a 12-byte IV.
3. In encrypt and wrap mode, the IV passed by the application is not be used. Instead, the HSM generates a 12-byte random IV and performs the crypto operation.
4. After the encryption and wrap operation, the application must retrieve the IV from the cipher object and secure it so that it has access to this IV when the decryption/unwrap operation is to be performed. Warning! Failure to do so will result in undecryptable data.
5. If a ciphertext provided by the Azure Cloud HSM JCE Provider needs to be decrypted using other providers, then 12 bytes of the post-encryption IV that was retrieved from the cipher should be used.
6. If the Azure Cloud HSM JCE Provider is used to decrypt ciphertext provided by another provider, then the IV used for the encryption should be used. Again, only data encrypted for a 64/96/128-bit tag length and using a 12-byte IV can be decrypted by the Azure Cloud HSM GCM implementation.
7. If wrapped data provided by the Azure Cloud HSM JCE Provider needs to be unwrapped using other providers, then 12 bytes of the post-wrap IV that was retrieved from the cipher should be used.

8. If the Azure Cloud HSM JCE Provider is used to unwrap wrapped data provided by another provider, then the IV used for the wrap should be used. Again, only data wrapped for a 64/96/128-bit tag length and using a 12-byte IV can be unwrapped by the Azure Cloud HSM GCM implementation.
9. The Azure Cloud HSM JCE Provider can encrypt a maximum of 16000 bytes of data.
10. The Azure Cloud HSM JCE Provider can decrypt a maximum of 16000 bytes of data + tag length.

Examples of interoperation between the SunJCE and Azure Cloud HSM JCE Provider are shown below.

```
public void testGCM() throws Exception {
    Cipher cp = Cipher.getInstance("AES/GCM/NoPadding", "Cavium");
    Cipher sp = Cipher.getInstance("AES/GCM/NoPadding", "SunJCE");
    SecretKey k = getCaviumSecretKey(256);
    byte[] iv = new byte[12];
    Random r = new Random();
    r.nextBytes(iv);

    byte[] p = new byte[] {1};
    byte[] aad = new byte[] {2};
    sp.init(Cipher.ENCRYPT_MODE, k, new GCMParameterSpec(128, iv));
    sp.updateAAD(aad);
    byte[] sc = sp.doFinal(p);
    cp.init(Cipher.ENCRYPT_MODE, k, new GCMParameterSpec(128, iv));
    cp.updateAAD(aad);
    byte[] c = cp.doFinal(p);
    byte[] newiv = cp.getIV();
    cp.init(Cipher.DECRYPT_MODE, k, new GCMParameterSpec(128, newiv));
    cp.updateAAD(aad);
    byte[] v = cp.doFinal(c);
    assertTrue("Cavium/Cavium Data compare failed", Arrays.equals(p, v));

    sp.init(Cipher.DECRYPT_MODE, k, new GCMParameterSpec(128, newiv));
    sp.updateAAD(aad);
    v = sp.doFinal(c);
    assertTrue("Cavium/Sun Data compare failed", Arrays.equals(p, v));
}
```

```

    cp.init(Cipher.DECRYPT_MODE, k, new GCMParameterSpec(128, iv));
    cp.updateAAD(aad);
    v = cp.doFinal(sc);
    assertTrue("Sun/Cavium Data compare failed", Arrays.equals(p, v));
}

public void testGCMWrapUnwrap() throws Exception {
    Cipher cp = Cipher.getInstance("AES/GCM/NoPadding", "Cavium");
    Cipher sp = Cipher.getInstance("AES/GCM/NoPadding", "SunJCE");
    KeyGenerator kg = KeyGenerator.getInstance("AES", "Cavium");
    kg.init(new CaviumAESKeyGenParameterSpec(256, "aetest-wrap-unwrap", true, true));
    SecretKey wk = kg.generateKey();
    SecretKey k = CaviumTestData.getExtractableCaviumAESKey(256);
    byte[] iv = new byte[12];
    Random r = new Random();
    r.nextBytes(iv);
    byte[] p = new byte[] { 1 };

    //WRAP with Cavium and UNWRAP with Sun
    cp.init(Cipher.WRAP_MODE, wk, new GCMParameterSpec(128, iv));
    byte[] wrapped = cp.wrap(k);
    sp.init(Cipher.UNWRAP_MODE, wk, new GCMParameterSpec(128, cp.getIV()));
    Key uk = sp.unwrap(wrapped, "AES", Cipher.SECRET_KEY);
    assertEquals("Wrapped / UnWrapped Key bits mismatch", k.getEncoded(),
        uk.getEncoded());

    //WRAP with Cavium and UNWRAP with Cavium
    cp.init(Cipher.WRAP_MODE, wk, new GCMParameterSpec(128, iv));
    wrapped = cp.wrap(k);

    byte[] newiv = cp.getIV();

```

```

cp.init(Cipher.UNWRAP_MODE, wk, new GCMPParameterSpec(128, newiv));
uk = cp.unwrap(wrapped, "AES", Cipher.SECRET_KEY);
assertArrayEquals("Wrapped / UnWrapped Key bits mismatch", k.getEncoded(),
uk.getEncoded());

//WRAP with Sun and UNWRAP with Cavium
sp.init(Cipher.WRAP_MODE, wk, new GCMPParameterSpec(128, iv));
wrapped = sp.wrap(k);

cp.init(Cipher.UNWRAP_MODE, wk, new GCMPParameterSpec(128, iv));
uk = cp.unwrap(wrapped, "AES", Cipher.SECRET_KEY);
assertArrayEquals("Wrapped / UnWrapped Key bits mismatch", k.getEncoded(),
uk.getEncoded());
}

```

Notes on Azure Cloud HSM AES-CCM Implementation

The following details apply to the Azure Cloud HSM CCM implementation:

1. This cipher supports only encrypt and decrypt modes.
2. A maximum of 16000 bytes can be processed for operations. These bytes may be part of update calls and the doFinal call.
3. Objects of this class cannot be cloned.
4. This cipher does not support CipherSpi.engineInit(int opmode, Key key, AlgorithmParameters params, SecureRandom rand).
5. The CCM cipher currently supports CaviumAEADParameterSpec and IvParameterSpec for AlgorithmParameterSpec.
6. CaviumAEADParameterSpec takes AAD, tagLengthInBits, and Nonce.
7. CaviumAEADParameterSpec passed to engineInit just meet the following restrictions:
 - a. nonce buffer length must be less than 13 and greater than 7. In encrypt mode, nonce buffer can be null; then a buffer of max length is created. In decrypt mode, nonce cannot be null.
 - b. AAD buffer must be less than 1023 bytes. It can be null in encrypt/decrypt modes.
 - c. tagLengthInBits must be between 32 and 128 with 16 step increase.
 - d. If IvParameterSpec is used in init method, then above checks for nonce are performed. AAD will be null and default tag would be considered.
 - e. AAD buffer passed in engineUpdateAAD(), will be appended to the AAD passed in the above spec.
8. In encrypt mode, nonce passed by the application is used for encryption. If passed nonce is null, then HSM generates nonce.
9. If the nonce passed in encrypt mode is null, then getIV() on cipher object would return a zero filled buffer of MAX_NONCE_length(13).

10. If the IV is generated in HSM for encryption, then the application should retrieve the IV from the Cipher object and secure it so it has access to this when the decryption is performed. `engineGetIV()` can be used to get the nonce generated in the encrypt operation.
WARNING: Failure to do so can lead to data mismatch.
11. If cipher is not initialized with algorithm parameters in Encrypt mode then Cavium CCM implementation generates the algorithm parameters. Note: default tag length is 128 bits and default nonce length is 13 bytes.
12. The Azure Cloud HSM JCE Provider can encrypt/decrypt maximum of 16000 bytes of data.

Examples of interoperation between the SunJCE and Azure Cloud HSM JCE Provider are shown below.

```
public void testCCM() throws Exception {
    Cipher enc = Cipher.getInstance("AES/CCM/NoPadding", "Cavium");
    Cipher dec = Cipher.getInstance("AES/CCM/NoPadding", "Cavium");
    SecretKey k = getCaviumSecretKey(256);
    int tagLen = 16*8;
    byte[] aad = new byte[10];
    byte[] nonce = new byte[10];
    Random r = new Random();
    r.nextBytes(nonce);
    r.nextBytes(aad);

    CaviumAEADParameterSpec spec = new CaviumAEADParameterSpec(nonce, aad, tagLen);
    byte[] p = new byte[] {1};
    enc.init(Cipher.ENCRYPT_MODE, k, spec);
    byte[] sc = enc.doFinal(p);

    CaviumAEADParameterSpec spec1 = new CaviumAEADParameterSpec(nonce, aad, tagLen);
    dec.init(Cipher.DECRYPT_MODE, k, spec1);
    byte[] v = dec.doFinal(sc);
    assertTrue("Cavium encrypt/decrypt Data compare failed", Arrays.equals(p, v));
}
```