Microsoft Azure Cloud HSM Certificate Storage

Table of Contents

Summary 2

Prerequisite 2

System Requirements 2

Certificate Storage Prerequisites 3

Setting up an Azure Blob Storage Account 3

Setting up User Assigned Managed Identity to access storage 4

Configure the Azure Cloud HSM Client Tools 6

Create a Storage Signing Key 6

Update Configuration Files 7

Update Application Config 7

Validate PKCS#11 Configuration 8

PKCS#11 API for Certificate Storage 8

Using PKCS#11 API for X.509 Certificate Storage 8

C_CreateObject 8

C_DestroyObject 12

C_CopyObject 12

C_GetAttributeValue 13

C_SetAttributeValue 14

C_FindObjectsInit 15

C_FindObjects 16

C_FindObjectsFinal 16

Configure and run your PKCS#11 application with Azure Cloud HSM 17

Setup Prerequisites for Sample Certificate Storage Application Test 17

Update and Compile Sample Certification Storage Application 17

Run Sample Certification Storage Application 18

Certificate Structure in Storage 18
Verify Certificates in Storage 18
Appendix 21
Frequently Asked Questions 21
PKCS#11 Certificate Storage Sample Application 22

Summary

Azure Cloud HSM supports certificate storage via PKCS#11. The Azure Cloud HSM PKCS#11 library supports storing public key certificates as public objects, in accordance with the PKCS#11 v2.40. This capability is available starting with SDK version 2.0.2.0 and enables both public and private PKCS#11 sessions to create, retrieve, modify, and delete certificate objects.

To use this feature, you must enable certificate storage in your client configuration. Once enabled, PKCS#11 applications can manage certificate objects alongside keys. Operations that span object types such as C_FindObjects will return both key and certificate objects, depending on the query.

Important Note: *Certificate Storage is supported in Azure Cloud HSM SDK 2.0.2.0 or higher.*

An X.509 certificate contains signed metadata and a public key, neither of which are considered secret, so storing the certificate itself in an HSM is typically unnecessary. However, the associated private key is sensitive and should be protected within an HSM. Azure Cloud HSM supports secure storage of RSA and EC private keys in the HSM, while storing the X.509 certificates in the customer's storage account when using the Azure Cloud HSM PKCS#11 library.

Prerequisite

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

System Requirements

- Azure Cloud HSM resource has been deployed, initialized, and configured.
- Azure Cloud HSM Client SDK
- Copy of partition owner certificate "PO.crt" on application server.
- Known address of your HSM "hsm1.chsm-<resourcename>-<uniquestring>.privatelink.cloudhsm.azure.net".
- Knowledge of Crypto User credentials

Certificate Storage Prerequisites

- Azure Blob Storage Account
- Managed Identity to access storage

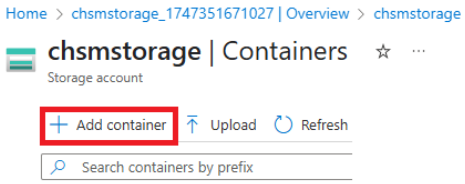
Important Note: Customers using any version of Windows Server should install the most recent version of Visual C++ Redistributable.

Setting up an Azure Blob Storage Account

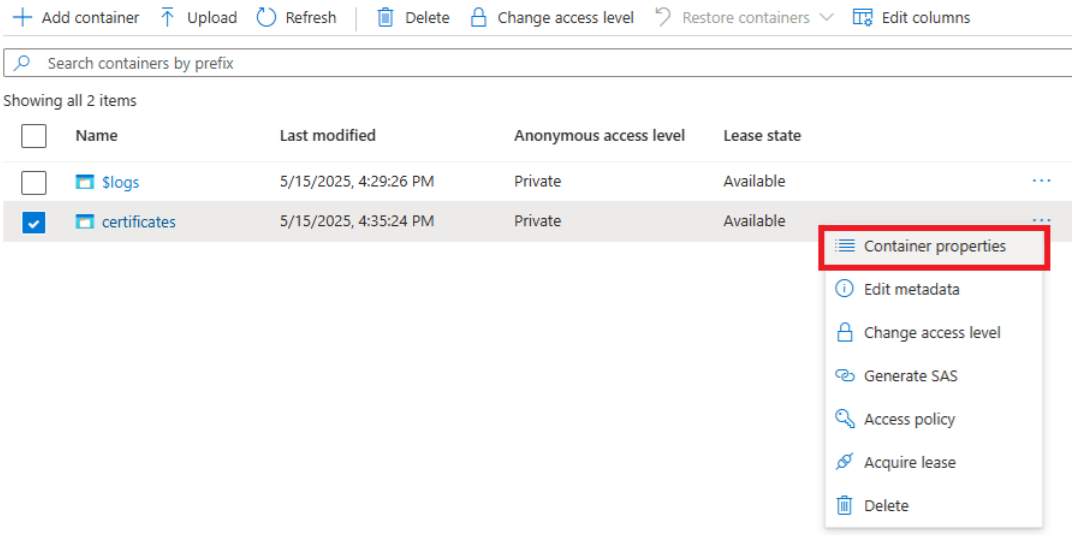
A prerequisite to running any PKCS#11 API for Certificate Storage is to create an Azure Blob Storage Account. This is where the PKCS#11 Certificate Objects will be stored (in JWS format) and read from.

1. To set up an Azure Blob Storage Account for PKCS#11 certificate storage, go to the Azure Portal and create a new **Storage Account**.

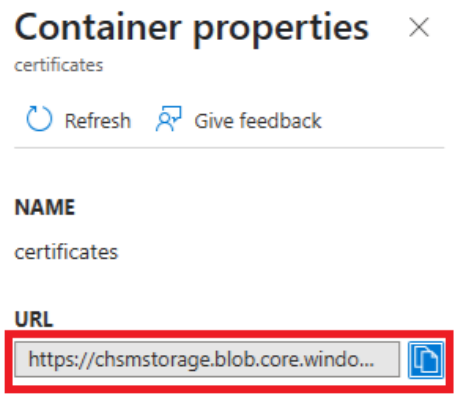
2. After successfully creating the Storage Account, navigate to it in the Azure Portal and select **Containers** under **Data storage**. Here, you will create a new container to store the blobs. Refer to the image below for guidance.



3. After creating the container, locate the container endpoint URL by navigating to **Container properties**. This URL will be needed later. See the image below for reference.



4. In **Container properties**, you will find the container URL listed. This URL is required later in the azcloudhsm_application.cfg file to enable PKCS#11 applications to locate the storage location for certificate objects. Refer to the image below for guidance.



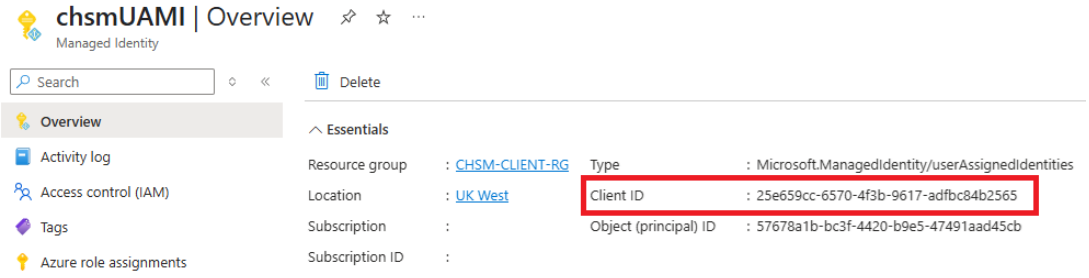
Setting up User Assigned Managed Identity to access storage

The next prerequisite for certificate storage is to create a **User Assigned Managed Identity**. This identity will be granted the necessary role to access the Azure Blob Storage Account and will be used to authenticate from your designated Admin VM.

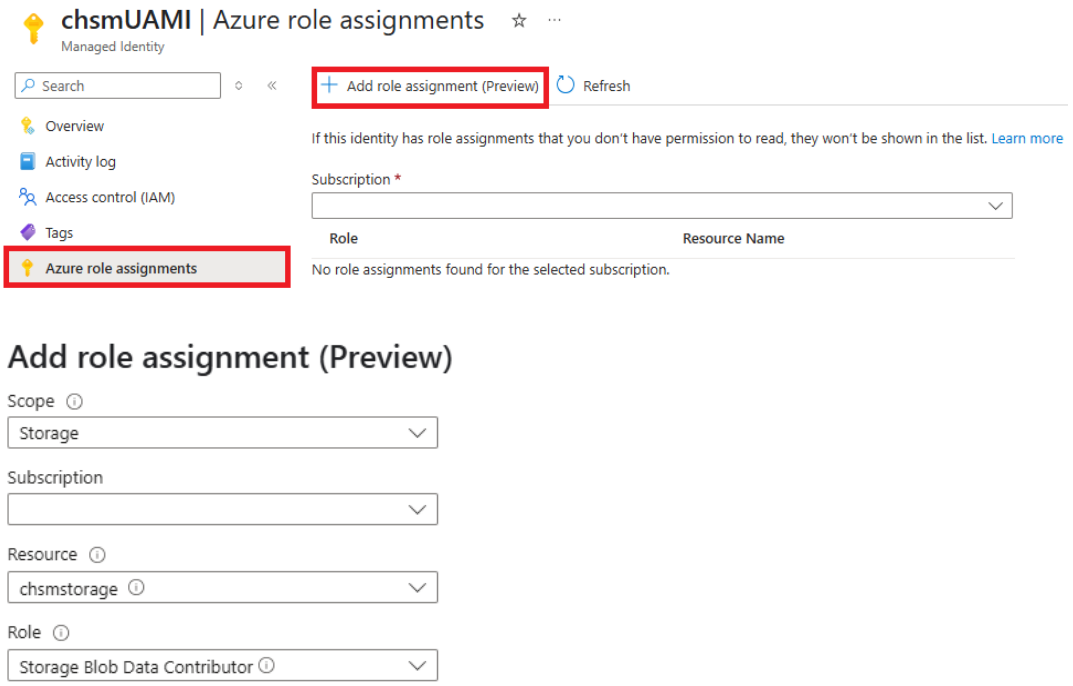
Important Note: *The following example will create and use a User Assigned Managed Identity. A System Assigned Managed Identity can also be created and used on the VM.*

1. To create a **User Assigned Managed Identity** for PKCS#11 certificate storage, navigate to the Azure Portal and create a new identity.

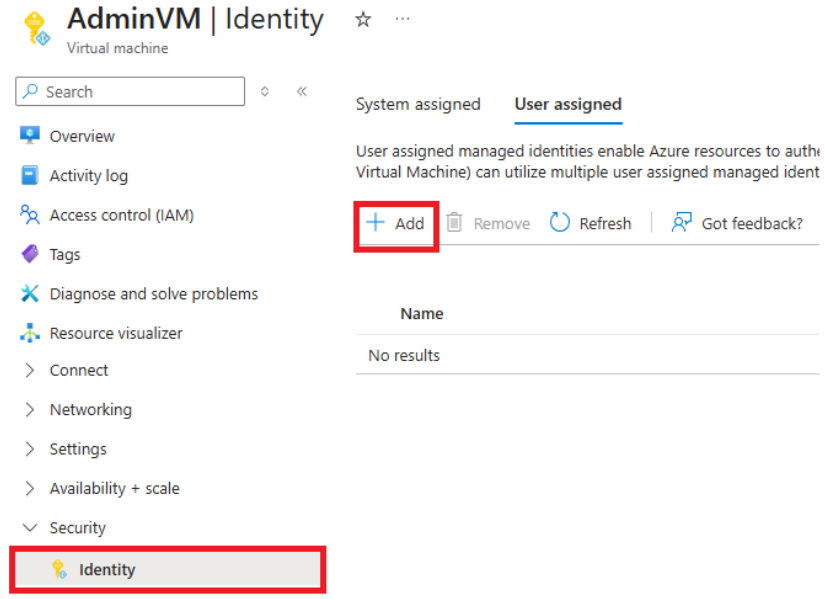
2. After successfully creating the Managed Identity, make note of the **Client ID**. This will be required later in the azcloudhsm_application.cfg file to enable authentication to the storage account from your VM. Refer to the image below for guidance.

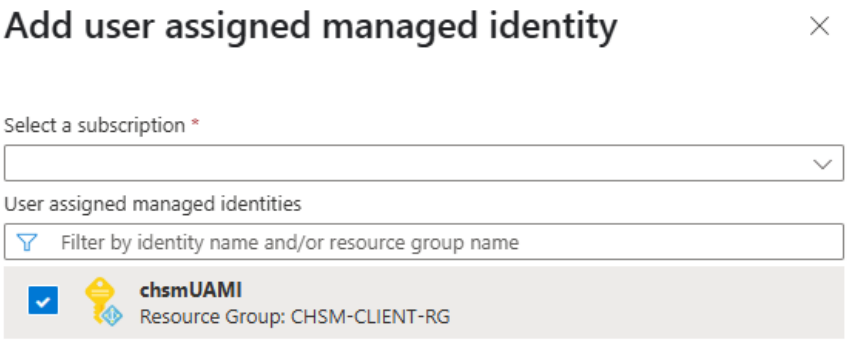


3. The next step is to assign the appropriate Azure role to grant the Managed Identity permission to read and write to the previously created Blob Storage Account. Assign the **Storage Blob Data Contributor** role to the Managed Identity, setting the **Scope** to **Storage** and selecting the specific Storage Account resource. Refer to the three images below for guidance.



4. The next step is to assign the **User Assigned Managed Identity** to the VM that will run your PKCS#11 certificate storage application. To do this, navigate to your VM resource in the Azure Portal, go to the **Security** section, select **Identity**, and add the User Assigned Identity. Refer to the image below for guidance.





Configure the Azure Cloud HSM Client Tools

Create a Storage Signing Key

The following azcloudhsm_util command can be used to create an RSA signing key pair for PKCS#11 certificate storage in a single step. By default, it generates a 2048-bit RSA key with a public exponent of 65537. You may modify the key size as needed.

Before running the command, ensure that the azcloudhsm_client is running as a service in the background.

Replace the placeholders as follows:

- PKCS11_S with your Crypto User username. (e.g. cu1)
- PKCS11_P with your Crypto User password. (e.g. user1234)
- SIGNING_KEY_ID with the desired key pair ID (this ID will also be used later in your azcloudhsm_application.cfg file)

Signing Key ID
For this example, we are going to set Signing Key ID to a random value.

```
SIGNING_KEY_ID=$(tr -dc 'a-z' </dev/urandom | head -c 10)
```

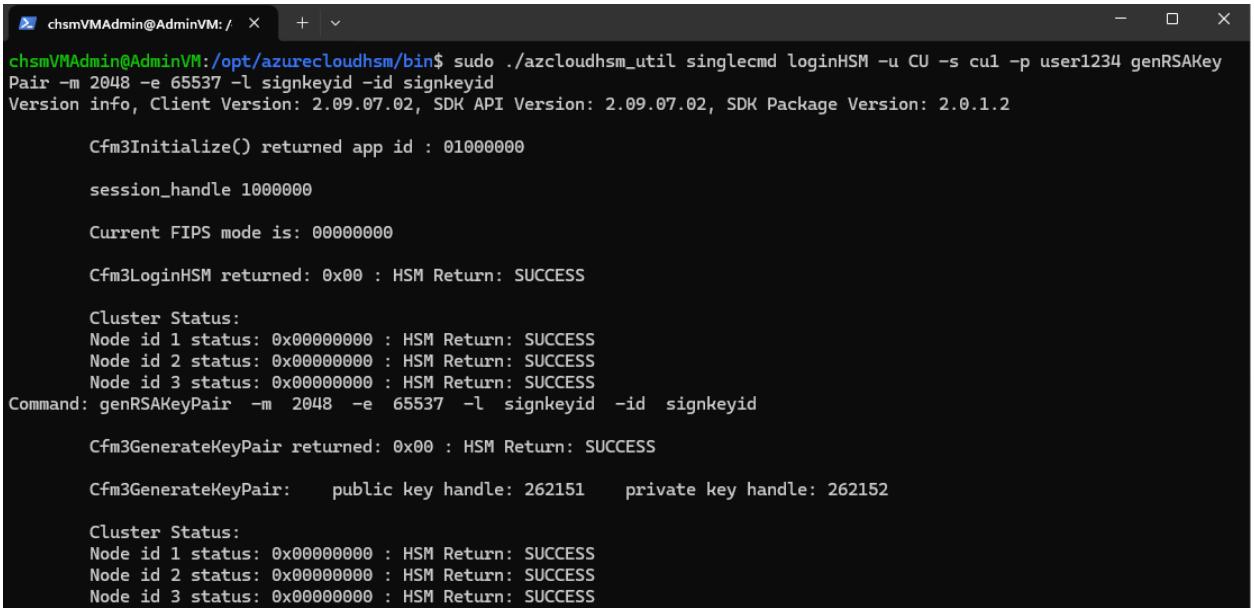
Linux:

```
sudo ./azcloudhsm_util singlecmd loginHSM -u CU -s $PKCS11_S -p $PKCS11_P genRSAKeyPair -m 2048 -e 65537 -l $SIGNING_KEY_ID -id $SIGNING_KEY_ID
```

Windows:

```
.\azcloudhsm_util.exe singlecmd loginHSM -u CU -s %PKCS11_S% -p %PKCS11_P% genRSAKeyPair -m 2048 -e 65537 -l %SIGNING_KEY_ID% -id %SIGNING_KEY_ID%
```

Important Note: *Please ensure that each of the HSM partitions returns to success.*



Update Configuration Files

Update Application Config

You will need to update the following parameters in the azcloudhsm_application.cfg file.

CERTSTORAGE_URL: This field refers to the URL of the container within the customer's Blob Storage Account and is used to store certificate information.
(e.g. https://chsmstorage.blob.core.windows.net/certificates)

CERTSTORAGE_SIGNING_KEYID: This field refers to the ID assigned to the key pair, which is used to perform integrity checks during read and write operations to storage (signing and verification).

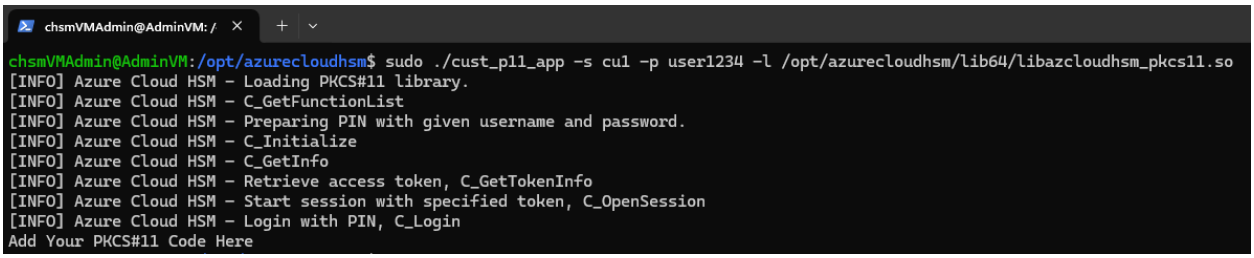
UAMI_CLIENT_ID: This field refers to the Client ID of the User Assigned Managed Identity, which is used to authenticate to the customer's Blob Storage Account. If left blank, authentication will default to using a System Assigned Managed Identity.

Important Note: These parameters only apply when running certificate operations in PKCS#11. It is not required for Key Operations.

```
DAEMON_ID=1
SOCKET_TYPE=UNIXSOCKET
PORT=1111
USER_KEK_HANDLE=262150
DEFAULT_WRAP_WITH_TRUSTED=1
CERTSTORAGE_URL=https://chsmstorage.blob.core.windows.net/certificates
CERTSTORAGE_SIGNING_KEYID=hjgrwvvofe
UAMI_CLIENT_ID=25e659cc-6570-4f3b-9617-adfbc84b2565
```

Validate PKCS#11 Configuration

Please refer to the PKCS#11 Integration Guide for sample-based validation of your PKCS#11 configuration.



PKCS#11 API for Certificate Storage

Using PKCS#11 API for X.509 Certificate Storage

The following existing APIs in PKCS#11 for Azure Cloud HSM have been expanded to add support for X.509 Public Key Certificates.

- **C_CreateObject:** Creates a new certificate object.
- **C_DestroyObject:** Deletes an existing certificate object.
- **C_CopyObject:** Copies an existing certificate object.
- **C_GetAttributeValue:** Gets the value of one or more attributes of a certificate object.
- **C_SetAttributeValue:** Updates the value of one or more attributes of a certificate object.
- **C_FindObjectsInit:** Starts a search for certificate objects.
- **C_FindObjects:** Continues a search for certificate objects.
- **C_FindObjectsFinal:** Ends a search for certificate objects.

C_CreateObject

The C_CreateObject API functions similarly for both keys and certificates. It expects an array of attributes, the number of attributes, and a pointer to an object handle where the generated handle will be stored.

Below is a sample of how to use the C_CreateObject.

```
int create_cert(CK_SESSION_HANDLE session_rw, CK_OBJECT_HANDLE_PTR cert_handle)
{
   // Dummy certificate data
   CK_BYTE certData[] = { 0x30, 0x82, 0x03, 0x08, 0x30, 0x82, 0x02, 0xD0 }; // Sample DER-encoded cert
   CK_ULONG certSize = sizeof(certData);

   CK_OBJECT_CLASS objClass = CKO_CERTIFICATE;
   CK_CERTIFICATE_TYPE certType = CKC_X_509;
   CK_BBOOL trueValue = CK_TRUE;
   CK_BYTE id[] = {123};
```

```
  // Dummy DER-encoded Subject Name (adjust as needed)
  CK_BYTE subjectData[] = { 0x30, 0x1D, 0x31, 0x1B, 0x30, 0x19, 0x06, 0x03,
                            0x55, 0x04, 0x03, 0x0C, 0x12, 'M', 'y', 'C', 'e',
                            'r', 't', 'i', 'f', 'i', 'c', 'a', 't', 'e', '-', 'B', 'b', 'j' };
  CK_ULONG subjectSize = sizeof(subjectData);

  CK_ATTRIBUTE certTemplate[] = {
     { CKA_CLASS, &objClass, sizeof(objClass) },
     { CKA_CERTIFICATE_TYPE, &certType, sizeof(certType) },
     { CKA_TOKEN, &trueValue, sizeof(trueValue) },
     { CKA_LABEL, "MyCertificate", 13 },
     { CKA_SUBJECT, subjectData, subjectSize },
     { CKA_ID, id, sizeof(id) },
     { CKA_VALUE, certData, certSize }
  };

  int n_attr = sizeof(certTemplate) / sizeof(CK_ATTRIBUTE);

  if ((func_list->C_CreateObject)(session_rw, certTemplate,
                          n_attr, cert_handle)) {
     return FAILED;
  }
#ifdef DEBUG
  printf("The cert handle created is : %lu \n", *cert_handle);
#endif

  return CKR_OK;
}
```

The following attributes represent the minimum required set to create an X.509 certificate in PKCS#11.

Layer	Attribute	Data Type	Description
Common Attributes	CKA_CLASS	CK_OBJECT_CLASS	Object class (type)
Certificate Objects	CKA_CERTIFICATE_TYPE	CK_CERTIFICATE_TYPE	Type of certificate, CKC_X_509 for X.509 public key certificates
X.509 Public Key Certificate Objects	CKA_SUBJECT	Byte array	DER-encoding of the certificate subject name

X.509 Public Key Certificate Objects	CKA_VALUE	Byte array	BER-encoding of the certificate

The following attributes are applicable to X.509 public key certificates.

Layer	Attribute	Data Type	Description
Common Attributes	CKA_CLASS	CK_OBJECT_CLASS	Object class (type)
Storage Objects	CKA_TOKEN	CK_BBOOL	CK_TRUE if object is a token object; CK_FALSE if object is a session object. Default is CK_FALSE.
Storage Objects	CKA_PRIVATE	CK_BBOOL	CK_TRUE if object is a private object; CK_FALSE if object is a public object. Default value is token-specific and may depend on the values of other attributes of the object.
Storage Objects	CKA_MODIFIABLE	CK_BBOOL	CK_TRUE if object can be modified, Default is CK_TRUE.
Storage Objects	CKA_LABEL	RFC2279 string	Description of the object (default empty).
Storage Objects	CKA_COPYABLE	CK_BBOOL	CK_TRUE if object can be copied using C_CopyObject. Defaults to CK_TRUE. Can't be set to TRUE once it is set to FALSE.
Storage Objects	CKA_DESTROYABLE	CK_BBOOL	CK_TRUE if the object can be destroyed using C_DestroyObject. Default is CK_TRUE.
Certificate Objects	CKA_CERTIFICATE_TYPE	CK_CERTIFICATE_TYPE	Type of certificate, CKC_X_509 for X.509 public key certificates
Certificate Objects	CKA_TRUSTED	CK_BBOOL	The certificate can be trusted for the application that it was created.
Certificate Objects	CKA_CERTIFICATE_CATEGORY	CKA_CERTIFICATE_CATEGORY	(default CK_CERTIFICATE_CATEGORY_UNSPECIFIED)
Certificate Objects	CKA_CHECK_VALUE	Byte array	Checksum

CKACertificate Objects	CKA_START_DATE	CK_DATE	Start date for the certificate (default empty)
Certificate Objects	CKA_END_DATE	CK_DATE	End date for the certificate (default empty)
Certificate Objects	CKA_PUBLIC_KEY_INFO	Byte Array	DER-encoding of the SubjectPublicKeyInfo for the public key contained in this certificate (default empty)
X.509 Public Key Certificate Objects	CKA_SUBJECT	Byte array	DER-encoding of the certificate subject name
X.509 Public Key Certificate Objects	CKA_ID	Byte array	Key identifier for public/private key pair (default empty)
X.509 Public Key Certificate Objects	CKA_ISSUER	Byte array	DER-encoding of the certificate issuer name (default empty)
X.509 Public Key Certificate Objects	CKA_SERIAL_NUMBER	Byte array	DER-encoding of the certificate serial number (default empty)
X.509 Public Key Certificate Objects	CKA_VALUE	Byte array	BER-encoding of the certificate
X.509 Public Key Certificate Objects	CKA_URL	RFC2279 string	If not empty this attribute gives the URL where the complete certificate can be obtained (default empty)
X.509 Public Key Certificate Objects	CKA_HASH_OF_SUBJECT_PUBLIC_KEY	Byte array	Hash of the subject public key (default empty). Hash algorithm is defined by CKA_NAME_HASH_ALGORITHM
X.509 Public Key Certificate Objects	CKA_HASH_OF_ISSUER_PUBLIC_KEY	Byte array	Hash of the issuer public key (default empty). Hash algorithm is defined by CKA_NAME_HASH_ALGORITHM
X.509 Public Key Certificate Objects	CKA_JAVA_MIDP_SECURITY_DOMAIN	CK_JAVA_MIDP_SECURITY_DOMAIN	Java MIDP security domain. (default CK_SECURITY_DOMAIN_UNSPECIFIED)

X.509 Public Key Certificate Objects	CKA_NAME_HASH_ALGORITHM	CK_MECHANISM_TYPE	Defines the mechanism used to calculate CKA_HASH_OF_SUBJECT_PUBLIC_KEY and CKA_HASH_OF_ISSUER_PUBLIC_KEY. If the attribute is not present, then the type defaults to SHA-1.

C_DestroyObject

The C_DestroyObject API takes a session handle and the object handle associated with the certificate you want to delete. Invoking this function removes the specified certificate from the Azure Blob Storage Account by deleting the corresponding JWS blob named pkcs11_certificate_<cert_handle>.

Below is a code snippet demonstrating how to call C_DestroyObject for certificates (the same approach applies to keys).

```
int delete_cert(CK_SESSION_HANDLE session_rw, CK_OBJECT_HANDLE cert_handle)
{
   CK_RV rv = 0;

   rv = (func_list->C_DestroyObject)(session_rw, cert_handle);

   if(rv != CKR_OK) {
      printf("Deleting Certificate failed \n");
      return rv;
   }

   return rv;
}
```

C_CopyObject

The C_CopyObject API takes a session handle, the handle of the object to be copied, and a pointer to receive the handle of the newly created object. To maintain parity with the C_CopyObject implementation for key objects in Azure Cloud HSM, the certificate implementation does not support modifying attributes during the copy operation.

Below is a sample snippet demonstrating how to use C_CopyObject for storing certificates.

```
int copy_cert(CK_SESSION_HANDLE session_rw, CK_OBJECT_HANDLE cert_handle,
              CK_OBJECT_HANDLE_PTR copied_cert_handle)
{
   CK_RV rv = 0;

   rv = (func_list->C_CopyObject)(session_rw, cert_handle, NULL, 0, copied_cert_handle);
```

```
    if(rv != CKR_OK) {
        printf("Copying Certificate failed \n");
        return rv;
    }

    return rv;
}
```

C_GetAttributeValue

The C_GetAttributeValue API allows retrieval of all attributes listed in the **C_CreateObject API** section. This API is typically invoked twice. The first call determines the size of attributes with unknown lengths such as CKA_SUBJECT, which contains the DER-encoded certificate subject.

Below is an example of how to call C_GetAttributeValue to obtain the sizes of the specified attributes shown in the image below:

```
int get_cert_attribute(CK_SESSION_HANDLE session_rw, CK_OBJECT_HANDLE_PTR cert_handle)
{
    CK_RV rv = 0;

    CK_ULONG cka_class = 0;
    CK_CERTIFICATE_TYPE cka_cert_type = 0;
    CK_BBOOL cka_token = 0;
    char* cka_label = NULL;
    char* cka_subject = NULL;
    CK_BYTE* cka_id = NULL;
    CK_BYTE* cka_value = NULL;

    // Determine size needed for each attribute by calling C_GetAttributeValue with NULL pointers
    // and zero as the length.

    CK_ATTRIBUTE cert_template[] = {
        { CKA_CLASS, NULL, 0 },
        { CKA_CERTIFICATE_TYPE, NULL, 0 },
        { CKA_TOKEN, NULL, 0 },
        { CKA_LABEL, NULL, 0 },
        { CKA_ID, NULL, 0 },
    };

    int n_attr = sizeof(cert_template) / sizeof(CK_ATTRIBUTE);

    rv = (func_list->C_GetAttributeValue)(session_rw, *cert_handle, cert_template, n_attr);

    if (rv != CKR_OK) {
        printf("C_GetAttributeValue failed with %ld\n", rv);
```

```
    return FAILED;
}
```

Once the attribute sizes are known, memory can be allocated accordingly. A second call to the C_GetAttributeValue API is then made to retrieve the attribute values and store them in the allocated memory.

The image below shows a code snippet demonstrating this process based on the previous example:

```
cka_label = (char*)malloc(cert_template[3].ulValueLen);
if (cka_label == NULL) {
    printf("Memory allocation failed for CKA_LABEL.\n");
    rv = FAILED;
    goto end_test_get_cert_attribute;
}

cert_template[3].pValue = cka_label;

if (cert_template[4].ulValueLen <= 0) {
    printf("CKA_ID size must be > 0.\n");
    rv = FAILED;
    goto end_test_get_cert_attribute;
}

cka_id = (CK_BYTE*)malloc(cert_template[4].ulValueLen);
if (cka_id == NULL) {
    printf("Memory allocation failed for CKA_ID.\n");
    rv = FAILED;
    goto end_test_get_cert_attribute;
}

cert_template[4].pValue = cka_id;

rv = (func_list->C_GetAttributeValue)(session_rw, *cert_handle, cert_template, n_attr);
if (rv != CKR_OK) {
    printf("C_GetAttributeValue failed with %ld\n", rv);
    rv = FAILED;
    goto end_test_get_cert_attribute;
}
```

C_SetAttributeValue

The C_SetAttributeValue API now supports updating certificate objects. It requires the session handle, the handle of the certificate to be updated, an array of attributes and their new values, and the number of attributes to update. Only attributes listed in the **C_CreateObject API Usage** table are supported for updates—attempting to modify unsupported attributes will result in a failed API call.

Below is a snippet showing how C_SetAttributeValue can be used with certificate objects:

```
int set_cert_attribute(CK_SESSION_HANDLE session_rw, CK_OBJECT_HANDLE_PTR cert_handle)
{
   CK_RV rv = CKR_OK;
   CK_BBOOL falseValue = CK_FALSE;
   CK_BYTE subjectData[] = { 0x40, 0x41, 0x42, 0x43, 0x44 };
   CK_BYTE id[] = {254};
   CK_BYTE certData[] = { 0x10, 0x20, 0x30, 0x40, 0x50, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF };

   CK_ATTRIBUTE certTemplateValid1[] = {
      { CKA_TOKEN, &falseValue, sizeof(falseValue) },
      { CKA_LABEL, "This is a new label", strlen("This is a new label") },
      { CKA_SUBJECT, subjectData, sizeof(subjectData) },
      { CKA_ID, id, sizeof(id) },
      { CKA_VALUE, certData, sizeof(certData) }
   };

   int n_attr = sizeof(certTemplateValid1) / sizeof(CK_ATTRIBUTE);
   rv = (func_list->C_SetAttributeValue)(session_rw, *cert_handle, certTemplateValid1, n_attr);
   if (rv != CKR_OK) {
      printf("test_set_cert_attribute failed when updating attribute values.\n");
      return FAILED;
   }

   return rv;
}
```

C_FindObjectsInit

The C_FindObjects* API now supports locating certificate objects in addition to key objects. A search operation can return both key and certificate handles if the search template includes attributes common to both object types. The C_FindObjectsInit API has been enhanced to support all certificate-related attributes listed in the **C_CreateObject API Usage** table.

Below is an example of a C_FindObjectsInit call that performs a certificate search using the CKA_CLASS, CKA_CERTIFICATE_TYPE, and CKA_LABEL attributes to find all matching certificate objects:

```
int find_cert(CK_SESSION_HANDLE session_rw, CK_OBJECT_HANDLE cert_handle)
{
   CK_RV rv;
   CK_OBJECT_CLASS objClass = CKO_CERTIFICATE;
   CK_CERTIFICATE_TYPE certType = CKC_X_509;

   CK_ATTRIBUTE certTemplate[] = {
      { CKA_CLASS, &objClass, sizeof(objClass) },
```

```
    { CKA_CERTIFICATE_TYPE, &certType, sizeof(certType) },
    { CKA_LABEL, "MyCertificate", 13 }
  };

  // Step 1: Initialize the search
  rv = (func_list->C_FindObjectsInit)(session_rw, certTemplate, sizeof(certTemplate) /
sizeof(CK_ATTRIBUTE));
  if (rv != CKR_OK) {
    printf("C_FindObjectsInit failed: 0x%lX\n", rv);
    return rv;
  }
```

C_FindObjects

After initializing the search parameters, the C_FindObjects API is used to retrieve the matching object handles. It also returns the number of objects found. This API takes the session handle, an array to store the resulting object handles, the maximum number of objects to retrieve, and an output parameter indicating how many objects were found.

The snippet below shows a call to C_FindObjects following the search template setup in the C_FindObjectsInit example above:

```
  // Step 2: Call C_FindObjects
  CK_OBJECT_HANDLE_PTR foundObjects = NULL;
  CK_ULONG maxObjects = 50;

  foundObjects = (CK_OBJECT_HANDLE_PTR)malloc(sizeof(CK_OBJECT_HANDLE) * maxObjects);
  if (!foundObjects) {
    printf("Memory allocation failed\n");
    return CKR_HOST_MEMORY;
  }
  CK_ULONG foundCount = 0;

  rv = (func_list->C_FindObjects)(session_rw, foundObjects, maxObjects, &foundCount);
  if (rv != CKR_OK) {
    printf("C_FindObjects failed: 0x%lX\n", rv);
    (func_list->C_FindObjectsFinal)(session_rw); // Ensure cleanup
    free(foundObjects);
    return rv;
  }
```

C_FindObjectsFinal

The C_FindObjectsFinal API behaves the same for both key and certificate objects. It takes the current session handle as an argument and performs cleanup of all search-related structures and memory allocated during the C_FindObjectsInit call.

Below is a snippet showing how to call C_FindObjectsFinal to complete and clean up the search process initiated by the C_FindObjectsInit and C_FindObjects APIs:

```
// Step 3: Finalize the search
rv = (func_list->C_FindObjectsFinal)(session_rw);
if (rv != CKR_OK) {
    printf("C_FindObjectsFinal failed: 0x%lX\n", rv);
    free(foundObjects);
    return rv;
}
```

Configure and run your PKCS#11 application with Azure Cloud HSM

Setup Prerequisites for Sample Certificate Storage Application Test

1. **Generate Private Key:** Use the following OpenSSL command to generate the private key on the HSM. Execute this command from your home directory, and do not run with sudo as it will run in a different session and fail to create the key.

   ```
   cd ~
   openssl genpkey -algorithm RSA -out private_key.pem -pkeyopt rsa_keygen_bits:2048 -engine azcloudhsm_openssl
   ```

2. **Generate Certificate:** Use the private key generated on the HSM to create a X509 certificate using the following OpenSSL command. Run command from the home directory.

   ```
   cd ~
   openssl req -new -x509 -key private_key.pem -out cert.pem -days 365 -engine azcloudhsm_openssl
   ```

Update and Compile Sample Certification Storage Application

Below we created a PKCS#11 certificate storage application example. Compile the program and generate the object file using gcc in the terminal. Then run the generated object file. While the PKCS#11 shared library exists under /opt/azurecloudhsm/**lib64** the .h files for Azure Cloud HSM PKCS#11 exist under /opt/azurecloudhsm/**pkcs11_headers/include**

You can install build essentials using your distribution package manager which is what we will use below to compile our sample PKCS#11 application.

```
sudo apt install build-essential
```

3. **Update pPin, pCert and priv_key_handle in sample application:** Copy the PKCS#11 sample application detailed further below in appendix and save as a .c file. In this example we saved the file as pkcs_certstore_test.c under /opt/azurecloudhsm.

   Ensure that the correct PIN is set using your Azure Cloud HSM user credentials in the format: username:password. The sample application uses this PIN with the C_Login function. It also expects an X509 certificate file named cert.pem. To run the sample, you must generate a private key and corresponding digital certificate, then update the application with the correct PEM file name and the handle of the associated private key.

```
sudo vim pkcs_certstore_test.c
```

Example:

```
char pPin[256] = "cu1:user1234";
char pCert[256] = "cert.pem";
char priv_key_handle[256] = "262150";
```

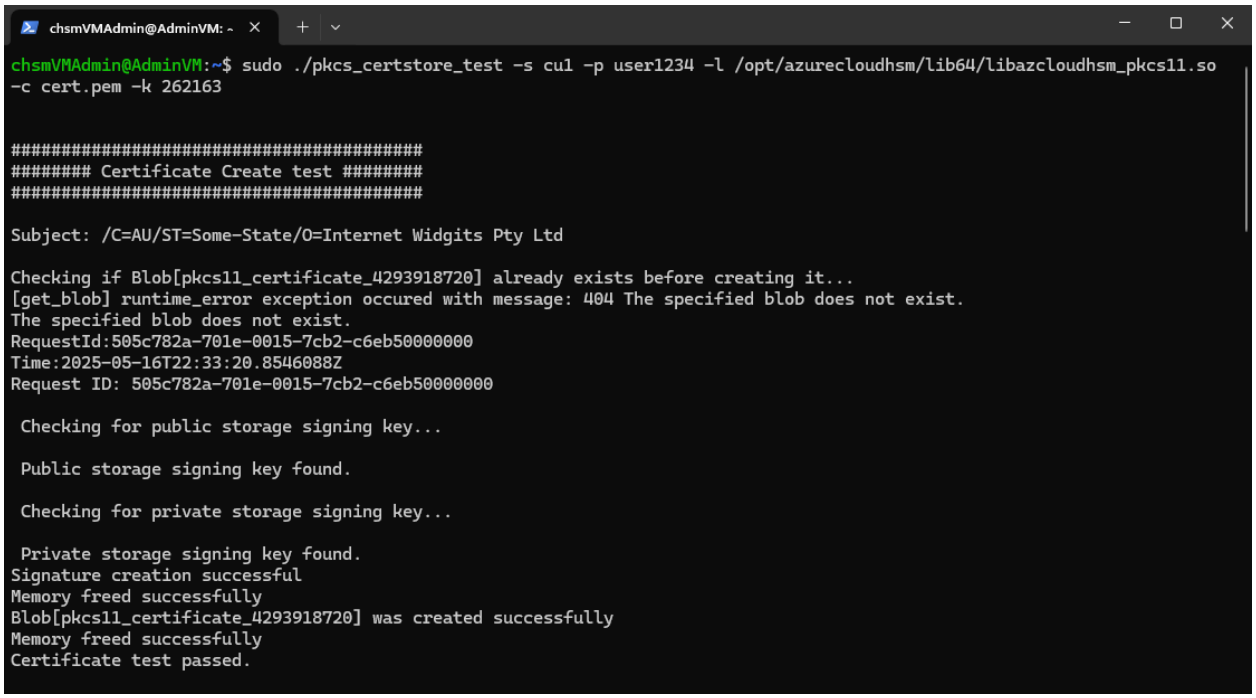
4. **Generate the object file by compiling the program using 'gcc'.** The example below is using the PKCS#11 sample application template captured below.

```
sudo gcc -o pkcs_certstore_test pkcs_certstore_test.c -
I/opt/azurecloudhsm/pkcs11_headers/include -ldl -lssl -lcrypto
```

Run Sample Certification Storage Application

5. **Run Sample Application:** Execute the custom PKCS#11 application from the example application template below.

```
sudo ./pkcs_certstore_test -s cu1 -p user1234 -l
/opt/azurecloudhsm/lib64/libazcloudhsm_pkcs11.so -c cert.pem -k {priv_key_handle}
```



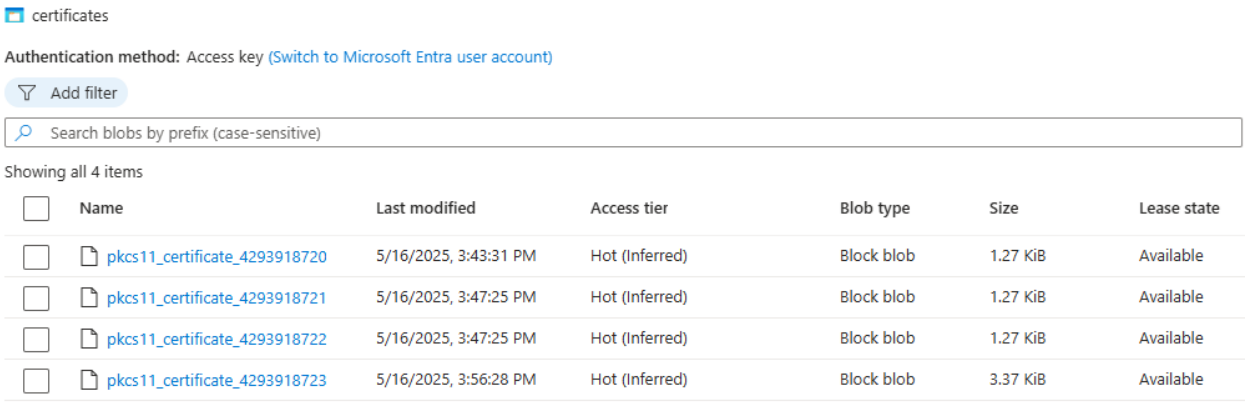
Certificate Structure in Storage

Verify Certificates in Storage

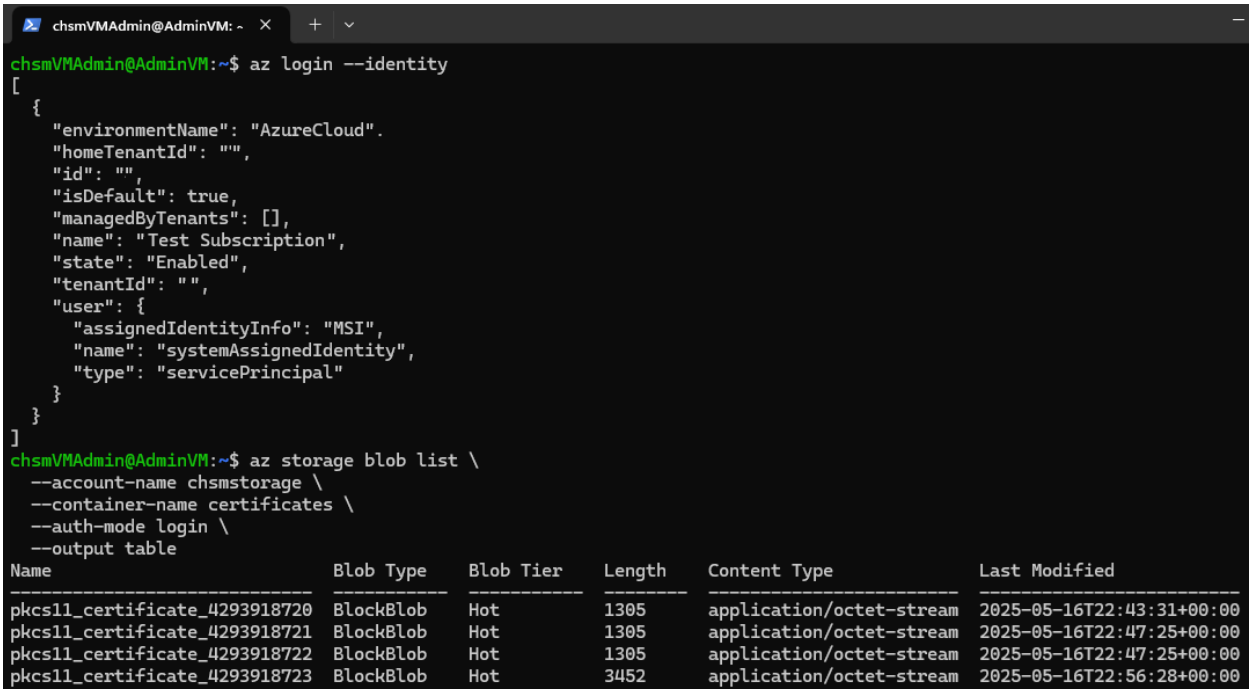
After a successful call to the C_CreateObject() API, the newly created certificate object will appear in your Azure Blob Storage account, as specified in the azcloudhsm_application.cfg file. The blob will be named using the format pkcs11_certificate_<ObjectHandle>, as shown below. Certificate objects are assigned object handles ranging from 0xFFF00000 to 0xFFFFFFFF (decimal range: 4,293,918,720 to 4,294,967,295), allowing support for up to 1,048,575 certificates.

From both Azure Portal as well as from your Azure VM you can see the certificates stored.

Verify from Azure Portal

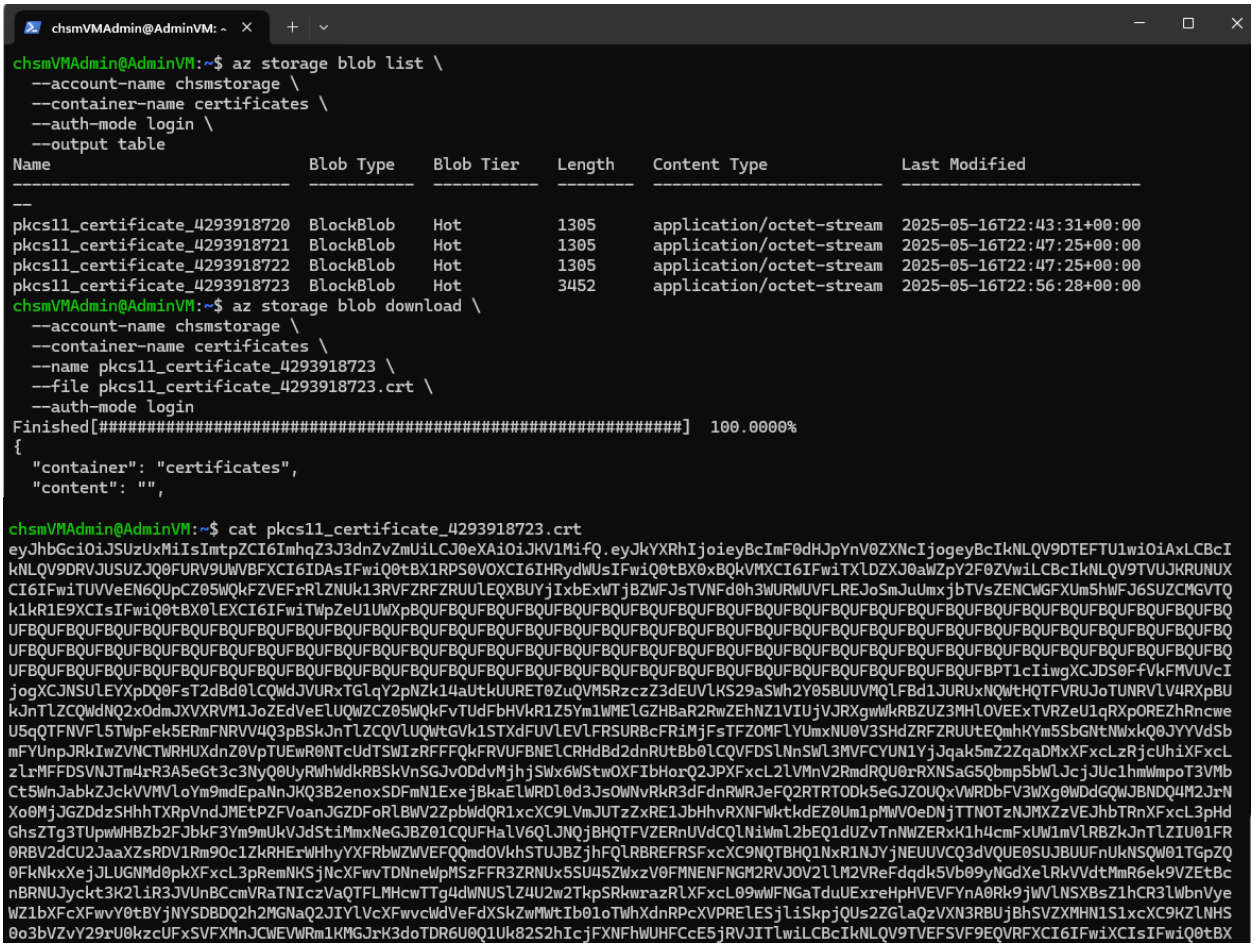


Verify from Azure VM with AZ CLI installed



Downloading the blob or viewing it in the Azure Portal and inspecting its contents will reveal that the certificate is stored as a JWS (JSON Web Signature) token. The token follows the standard JWS structure, which is divided into the following format:

Header.Payload.Signature Example



You can examine the structure of the JWS token by pasting it into JSON Web Tokens – jwt.io. The site will display the token's **Encoded** sections highlighting the header, payload, and signature as well as the decoded **Header** and **Payload**.

Below is an example of a certificate JWS token visualized on the site:

Appendix

Frequently Asked Questions

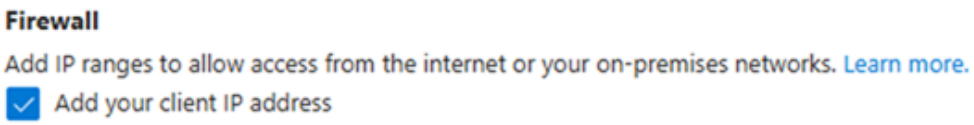
- **C_Login failed with error code: 0xa3.**
    - Encountering (0xa3) indicates a login failure that the issue might be related to the format of the PIN parameter you're passing or incorrect PIN (password).
        - The PIN should be provided in the following format: char pPin[256] = "crypto_user:user123"; The pPin value should follow the structure of "<username>:<password>". Please verify that you are providing the pPin in this format, as any deviation might result in a login failure.

- **C_Initialize() failed with 00000005 (Failed to connect socket).**
    - Encountering a failed socket connection during C_Initialize might be related to the azcloudhsm_client not running on your system. For your PKCS#11 applications to execute successfully the azcloudhsm_client must be running. You can start the azcloudhsm_client by ensuring its running as a service or by manually running the client.

- **azcloudhsm_application.cfg is not present (Cannot find file).**
    - The file location of the azcloudhsm_application.cfg is different for Linux and Windows. You may need to copy the azcloudhsm_application.cfg file from the default location below to the location of the PKCS#11 application you are running. They should both exist in the same directory.
        - **Linux File Location:** /opt/azurecloudhsm/bin
        - **Windows File Location:** C:\Program Files\Microsoft Azure Cloud HSM Client SDK\utils\azcloudhsm_util\azcloudhsm_application.cfg

- **C_Login failed with error code: 0xa3.**
    - Encountering (0xa3) indicates a login failure that the issue might be related to the format of the PIN parameter you're passing or incorrect PIN (password).

- **Accessing Storage Account in portal returns This request is not authorized to perform this operation.**
    - Go to Azure Portal and navigate to storage accounts > chsmstorage > networking. Under firewalls and virtual networks ensure the option selected networks is chosen. Click and add your client IP address and save the configuration.



- **PKCS#11 sample application failed with error: cryptoki.h no such file or directory.**
    - You need to tell gcc where to find the cryptoki.h file using the -I (include path) flag. This is located under /opt/azurecloudhsm/pkcs11_headers/include.
        - Example: sudo gcc -o pkcs_certstore_test pkcs_certstore_test.c -I/opt/azurecloudhsm/include -ldl

- **PKCS#11 sample application failed with error: openssl/*.h no such file or directory.**
    - The error means the compiler can't find the OpenSSL development headers. You will need to install libssl. In the sample application we're linking against OpenSSL in addition to the /pkcs11_headers/include files for PKCS#11.
        - **For Red Hat based (using yum):**

```
sudo yum install openssl-devel
```

        - **Red Hat based systems (using dnf):**

```
sudo dnf install openssl-devel
```

        - **For Ubuntu based systems (using apt):**

```
sudo apt update
sudo apt install libssl-dev
```

PKCS#11 Certificate Storage Sample Application

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#ifdef _WIN32
#include <WinSock2.h>
#include <winpthreads.h>
#else
#include <pthread.h>
#endif // #ifdef _WIN32
#include <ctype.h>
```

```
#include "cryptoki.h"
#ifndef _WIN32
#include "pkcs11t.h"
#else
#pragma pack(push, cryptoki, 1)
#include "pkcs11t.h"
#pragma pack(pop, cryptoki)
#endif
#include "dlfcn.h"
#ifndef _WIN32
#include "openssl/rand.h"
#else
#include <openssl/rand.h>
#endif
#include "openssl/crypto.h"
#include "helper.h"
#define MAX_DATA_LENGTH 8192
#ifdef DEBUG
#define print_dbg(...)  printf(__VA_ARGS__)
#else
#define print_dbg(...)   do { } while(0)
#endif

// Following includes are for reading the certificate
#include <openssl/x509.h>
#include <openssl/pem.h>
#include <openssl/bio.h>

#define CAVIUM_SLOT 1

#define FAILED -1

char pPin[256] = "cu1:user1234";
char pCert[256] = "cert.pem";
char priv_key_handle[256] = "-1";

X509 *load_cert_from_file(const char *filename)
{
   FILE *fp = fopen(filename, "rb");
   if (!fp) {
      perror("Error opening cert file");
      return NULL;
   }

   X509 *cert = NULL;
   if (strstr(filename, ".pem")) {
      cert = PEM_read_X509(fp, NULL, NULL, NULL);
   } else {
```

```
        cert = d2i_X509_fp(fp, NULL);
    }

    fclose(fp);
    return cert;
}

int test_create_cert(CK_SESSION_HANDLE session_rw, CK_OBJECT_HANDLE_PTR cert_handle)
{
    X509 *cert = load_cert_from_file(pCert);
    if (!cert) {
        printf("Failed to load certificate from file: %s\n", pCert);
        return FAILED;
    }

    // Get DER-encoded certificate value
    int len = i2d_X509(cert, NULL);
    if (len <= 0) {
        X509_free(cert);
        return FAILED;
    }

    CK_BYTE *cert_der = (CK_BYTE *)malloc(len);
    unsigned char *tmp = cert_der;
    i2d_X509(cert, &tmp);

    // Get subject
    X509_NAME *subject = X509_get_subject_name(cert);

    // Print subject string
    char subject_str[256];
    if (X509_NAME_oneline(subject, subject_str, sizeof(subject_str)) != NULL) {
        printf("Subject: %s\n", subject_str);
    } else {
        printf("Failed to get subject string\n");
    }

    unsigned char *subject_der = NULL;
    int subject_len = i2d_X509_NAME(subject, &subject_der);

    // Store the priv_key handle in the ID field
    CK_BYTE id[256];
    CK_ULONG id_len = sizeof(priv_key_handle);
    memcpy(id, priv_key_handle, id_len);

    CK_OBJECT_CLASS objClass = CKO_CERTIFICATE;
    CK_CERTIFICATE_TYPE certType = CKC_X_509;
    CK_BBOOL trueValue = CK_TRUE;
```

```
   CK_ATTRIBUTE certTemplate[] = {
      { CKA_CLASS, &objClass, sizeof(objClass) },
      { CKA_CERTIFICATE_TYPE, &certType, sizeof(certType) },
      { CKA_TOKEN, &trueValue, sizeof(trueValue) },
      { CKA_LABEL, "MyCertificate", 13 },
      { CKA_SUBJECT, subject_der, subject_len },
      { CKA_ID, id, sizeof(id) },
      { CKA_VALUE, cert_der, len }
   };

   int n_attr = sizeof(certTemplate) / sizeof(CK_ATTRIBUTE);
   CK_RV rv = (func_list->C_CreateObject)(session_rw, certTemplate, n_attr, cert_handle);

   OPENSSL_free(subject_der);
   free(cert_der);
   X509_free(cert);

   if (rv != CKR_OK) {
      return FAILED;
   }

#ifdef DEBUG
   printf("Certificate object created successfully. Handle: %lu\n", *cert_handle);
#endif
   return CKR_OK;
}

int test_delete_cert(CK_SESSION_HANDLE session_rw, CK_OBJECT_HANDLE cert_handle)
{
   CK_RV rv = 0;

   rv = (func_list->C_DestroyObject)(session_rw, cert_handle);

   if(rv != CKR_OK) {
      printf("Deleting Certificate failed \n");
      return rv;
   }

   return rv;
}

int test_copy_cert(CK_SESSION_HANDLE session_rw, CK_OBJECT_HANDLE cert_handle,
                CK_OBJECT_HANDLE_PTR copied_cert_handle)
{
   CK_RV rv = 0;

   rv = (func_list->C_CopyObject)(session_rw, cert_handle, NULL, 0, copied_cert_handle);
```

```
   if(rv != CKR_OK) {
      printf("Copying Certificate failed \n");
      return rv;
   }

   return rv;
}

int test_get_cert_attribute(CK_SESSION_HANDLE session_rw, CK_OBJECT_HANDLE_PTR cert_handle)
{
   CK_RV rv = 0;

   CK_ULONG cka_class = 0;
   CK_CERTIFICATE_TYPE cka_cert_type = 0;
   CK_BBOOL cka_token = 0;
   char* cka_label = NULL;
   char* cka_subject = NULL;
   CK_BYTE* cka_id = NULL;
   CK_BYTE* cka_value = NULL;

   // Determine size needed for each attribute by calling C_GetAttributeValue with NULL pointers
   // and zero as the length.

   CK_ATTRIBUTE cert_template[] = {
      { CKA_CLASS, NULL, 0 },
      { CKA_CERTIFICATE_TYPE, NULL, 0 },
      { CKA_TOKEN, NULL, 0 },
      { CKA_LABEL, NULL, 0 },
      { CKA_ID, NULL, 0 },
   };

   int n_attr = sizeof(cert_template) / sizeof(CK_ATTRIBUTE);

   rv = (func_list->C_GetAttributeValue)(session_rw, *cert_handle, cert_template, n_attr);

   if (rv != CKR_OK) {
      printf("C_GetAttributeValue failed with %ld\n", rv);
      return FAILED;
   }

   // Check for valid length and allocate memory buffer for each attribute.

   if (cert_template[0].ulValueLen != sizeof(cka_class)) {
      printf("CKA_CLASS size mismatch.\n");
      rv = FAILED;
      goto end_test_get_cert_attribute;
   }
```

```
cert_template[0].pValue = &cka_class;

if (cert_template[1].ulValueLen != sizeof(cka_cert_type)) {
   printf("CKA_CERTIFICATE_TYPE size mismatch.\n");
   rv = FAILED;
   goto end_test_get_cert_attribute;
}

cert_template[1].pValue = &cka_cert_type;

if (cert_template[2].ulValueLen != sizeof(cka_token)) {
   printf("CKA_TOKEN size mismatch.\n");
   rv = FAILED;
   goto end_test_get_cert_attribute;
}

cert_template[2].pValue = &cka_token;

if (cert_template[3].ulValueLen <= 0) {
   printf("CKA_LABEL size must be > 0.\n");
   rv = FAILED;
   goto end_test_get_cert_attribute;
}

cka_label = (char*)malloc(cert_template[3].ulValueLen);
if (cka_label == NULL) {
   printf("Memory allocation failed for CKA_LABEL.\n");
   rv = FAILED;
   goto end_test_get_cert_attribute;
}

cert_template[3].pValue = cka_label;

if (cert_template[4].ulValueLen <= 0) {
   printf("CKA_ID size must be > 0.\n");
   rv = FAILED;
   goto end_test_get_cert_attribute;
}

cka_id = (CK_BYTE*)malloc(cert_template[4].ulValueLen);
if (cka_id == NULL) {
   printf("Memory allocation failed for CKA_ID.\n");
   rv = FAILED;
   goto end_test_get_cert_attribute;
}

cert_template[4].pValue = cka_id;
```

```
rv = (func_list->C_GetAttributeValue)(session_rw, *cert_handle, cert_template, n_attr);
if (rv != CKR_OK) {
   printf("C_GetAttributeValue failed with %ld\n", rv);
   rv = FAILED;
   goto end_test_get_cert_attribute;
}

size_t cka_id_size = 0;
size_t cka_label_size = 0;

for (int i = 0; i < n_attr; i++)
{
   if (cert_template[i].type == CKA_ID) {
      cka_id_size = cert_template[i].ulValueLen;
   }
   else if (cert_template[i].type == CKA_LABEL) {
      cka_label_size = cert_template[i].ulValueLen;
   }
}

printf("CKA_CLASS: %ld\n", cka_class);
printf("CKA_CERTIFICATE_TYPE: %ld\n", cka_cert_type);
printf("CKA_TOKEN: %d\n", cka_token);

printf("CKA_LABEL: ");
for (size_t i = 0; i < cka_label_size; i++)
{
   printf("%c ", cka_label[i]);
}
printf("\n");

printf("CKA_ID: ");
for (size_t i = 0; i < cka_id_size; i++)
{
   printf("%d ", cka_id[i]);
}
printf("\n");

if (cka_class != CKO_CERTIFICATE) {
   printf("For CKA_CLASS, expected %ld but got %ld\n", CKO_CERTIFICATE, cka_class);
   rv = FAILED;
   goto end_test_get_cert_attribute;
}
if (cka_cert_type != CKC_X_509) {
   printf("For CKA_CERTIFICATE_TYPE, expected %ld but got %ld\n", CKC_X_509, cka_cert_type);
   rv = FAILED;
   goto end_test_get_cert_attribute;
```

```
    }
    if (cka_token != TRUE) {
        printf("For CKA_TOKEN, expected %d but got %d\n", TRUE, cka_token);
        rv = FAILED;
        goto end_test_get_cert_attribute;
    }
    if (strncmp(cka_label, "MyCertificate", 13) != 0) {
        printf("For CKA_LABEL, expected %s but got %s\n", "MyCertificate", cka_label);
        rv = FAILED;
        goto end_test_get_cert_attribute;
    }

    end_test_get_cert_attribute:
        // Free allocated memory
        if (cka_label)
            free(cka_label);
        if (cka_subject)
            free(cka_subject);
        if (cka_id)
            free(cka_id);
        if (cka_value)
            free(cka_value);

    return rv;
}

int test_set_cert_attribute(CK_SESSION_HANDLE session_rw, CK_OBJECT_HANDLE_PTR cert_handle)
{
    CK_RV rv = CKR_OK;
    CK_BBOOL falseValue = CK_FALSE;
    CK_BYTE subjectData[] = { 0x40, 0x41, 0x42, 0x43, 0x44 };
    CK_BYTE id[] = {254};
    CK_BYTE certData[] = { 0x10, 0x20, 0x30, 0x40, 0x50, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF };

    CK_ATTRIBUTE certTemplateValid1[] = {
        { CKA_TOKEN, &falseValue, sizeof(falseValue) },
        { CKA_LABEL, "This is a new label", strlen("This is a new label") },
        { CKA_SUBJECT, subjectData, sizeof(subjectData) },
        { CKA_ID, id, sizeof(id) },
        { CKA_VALUE, certData, sizeof(certData) }
    };

    int n_attr = sizeof(certTemplateValid1) / sizeof(CK_ATTRIBUTE);
    rv = (func_list->C_SetAttributeValue)(session_rw, *cert_handle, certTemplateValid1, n_attr);
    if (rv != CKR_OK) {
        printf("test_set_cert_attribute failed when updating attribute values.\n");
        return FAILED;
    }
```

```
    return rv;
}

int test_find_cert(CK_SESSION_HANDLE session_rw, CK_OBJECT_HANDLE cert_handle)
{
    CK_RV rv;
    CK_OBJECT_CLASS objClass = CKO_CERTIFICATE;
    CK_CERTIFICATE_TYPE certType = CKC_X_509;

    CK_ATTRIBUTE certTemplate[] = {
        { CKA_CLASS, &objClass, sizeof(objClass) },
        { CKA_CERTIFICATE_TYPE, &certType, sizeof(certType) },
        { CKA_LABEL, "MyCertificate", 13 }
    };

    // Step 1: Initialize the search
    rv = (func_list->C_FindObjectsInit)(session_rw, certTemplate, sizeof(certTemplate) /
sizeof(CK_ATTRIBUTE));
    if (rv != CKR_OK) {
        printf("C_FindObjectsInit failed: 0x%lX\n", rv);
        return rv;
    }

    // Step 2: Call C_FindObjects
    CK_OBJECT_HANDLE_PTR foundObjects = NULL;
    CK_ULONG maxObjects = 50;

    foundObjects = (CK_OBJECT_HANDLE_PTR)malloc(sizeof(CK_OBJECT_HANDLE) * maxObjects);
    if (!foundObjects) {
        printf("Memory allocation failed\n");
        return CKR_HOST_MEMORY;
    }
    CK_ULONG foundCount = 0;

    rv = (func_list->C_FindObjects)(session_rw, foundObjects, maxObjects, &foundCount);
    if (rv != CKR_OK) {
        printf("C_FindObjects failed: 0x%lX\n", rv);
        (func_list->C_FindObjectsFinal)(session_rw); // Ensure cleanup
        free(foundObjects);
        return rv;
    }

    // Step 3: Finalize the search
    rv = (func_list->C_FindObjectsFinal)(session_rw);
    if (rv != CKR_OK) {
        printf("C_FindObjectsFinal failed: 0x%lX\n", rv);
        free(foundObjects);
```

```
    return rv;
  }

  // Step 4: Check if cert_handle is in foundObjects
  for (CK_ULONG i = 0; i < foundCount; i++) {
    if (foundObjects[i] == cert_handle) {
      printf("Certificate handle %lu found successfully.\n", cert_handle);
      rv = CKR_OK;
      free(foundObjects);
      return rv; // Success
    }
  }

  printf("Certificate handle %lu was not found.\n", cert_handle);
  free(foundObjects);
  return FAILED; // Not found
}

/*
 * Main function
 */
int main_func()
{
  CK_SESSION_HANDLE session_rw;
  CK_SLOT_ID slot = CAVIUM_SLOT;
  CK_FUNCTION_LIST_PTR list;
  CK_INFO info;
  CK_OBJECT_HANDLE cert_handle = -1;
  CK_OBJECT_HANDLE copied_cert_handle = -1;
#ifndef _WIN32
  CK_TOKEN_INFO token_info = { };
#else
  CK_TOKEN_INFO token_info = { 0 };
#endif
  CK_RV rv;
  int n_pin = strlen(pPin);
  int retval = 0;
  int s_done = 0;

  rv = (func_list->C_GetFunctionList) (&list);
  if (CKR_OK != rv) {
    printf("\nCall through function list %08lx\n", rv);
    retval = FAILED;
    goto clean_up;
  }

  if (memcmp(list, func_list, sizeof(*list)) != 0)
    printf
```

```
        ("\nCall doesn't return same data as library C_GetFunctionList entry point\n");

print_dbg("\nC_Initialize\n");

rv = (func_list->C_Initialize) (NULL);
if (CKR_OK != rv) {
   printf("\nC_Initialize() failed with %08lx\n", rv);
   retval = FAILED;
   goto clean_up;
}

rv = (func_list->C_GetInfo) (&info);
if (CKR_OK != rv) {
   printf("\nCall through function list %08lx\n", rv);
   retval = FAILED;
   goto clean_up;
}

rv = (func_list->C_GetTokenInfo) (slot, &token_info);
if (CKR_OK != rv) {
   printf("\nCall through function list %08lx\n", rv);
   retval = FAILED;
   goto clean_up;
}

rv = (func_list->C_OpenSession) (slot,
       CKF_SERIAL_SESSION | CKF_RW_SESSION,
       NULL, NULL, &session_rw);
if (CKR_OK != rv) {
   printf("\nC_OpenSession() failed with %08lx\n", rv);
   retval = FAILED;
   goto clean_up;
}
s_done = 1;

rv = (func_list->C_Login) (session_rw, CKU_USER,
       (CK_UTF8CHAR_PTR) pPin, n_pin);
if (CKR_OK != rv) {
   print_dbg("\nlogin failed %08lx\n", rv);
   retval = FAILED;
   goto clean_up;
}
print_dbg("\nNormal login %08lx\n", rv);

printf("\n\n##########################################");
printf("\n######## Certificate Create test ########\n");
printf("##########################################\n\n");
rv = test_create_cert(session_rw, &cert_handle);
```

```
if (CKR_OK == rv) {
   printf("Certificate test passed.\n");
} else {
   printf("Error: Certificate test failed.\n");
   retval = FAILED;
   goto clean_up;
}

printf("\n\n##################################################");
printf("\n######## Certificate Get Attribute test ########\n");
printf("##################################################\n\n");
rv = test_get_cert_attribute(session_rw, &cert_handle);
if (CKR_OK == rv) {
   printf("Certificate Get Attribute test passed.\n");
} else {
   printf("Error: Certificate Get Attribute test failed.\n");
   retval = FAILED;
   goto clean_up;
}

printf("\n\n##########################################");
printf("\n######## Certificate Find Object ########\n");
printf("##########################################\n\n");
rv = test_find_cert(session_rw, cert_handle);
if (CKR_OK == rv) {
   printf("Certificate Find test passed.\n");
} else {
   printf("Error: Certificate Find test failed.\n");
   retval = FAILED;
   goto clean_up;
}

printf("\n\n##################################################");
printf("\n######## Certificate Set Attribute test ########\n");
printf("##################################################\n\n");
rv = test_set_cert_attribute(session_rw, &cert_handle);
if (CKR_OK == rv) {
   printf("Certificate Set Attribute test passed.\n");
} else {
   printf("Error: Certificate Set Attribute test failed.\n");
   retval = FAILED;
   goto clean_up;
}

printf("\n\n########################################");
printf("\n######## Certificate Copy test ########\n");
printf("########################################\n\n");
rv = test_copy_cert(session_rw, cert_handle, &copied_cert_handle);
```

```
   if (CKR_OK == rv) {
      printf("Copy Certificate test passed.\n");
      test_delete_cert(session_rw, copied_cert_handle);
   } else {
      printf("Error: Copy Certificate test failed.\n");
      retval = FAILED;
      goto clean_up;
   }

   printf("\n\n##########################################");
   printf("\n######## Certificate Delete test ########\n");
   printf("##########################################\n\n");
   rv = test_delete_cert(session_rw, cert_handle);
   if (CKR_OK == rv) {
      printf("Certificate delete test passed.\n");
   } else {
      printf("Error: Certificate delete test failed.\n");
      retval = FAILED;
      goto clean_up;
   }

 clean_up:
   if (s_done) {
      /* Do logout */
      rv = func_list->C_Logout(session_rw);
      if (CKR_OK != rv) {
         printf("\nC_Logout() failed with %08lx\n", rv);
         retval = FAILED;
      }
      /* Do close session */
      rv = func_list->C_CloseSession(session_rw);
      if (CKR_OK != rv) {
         printf("\nC_CloseSession() failed with %08lx\n", rv);
         retval = FAILED;
      }
   }
   /* Do app shutdown */
   rv = func_list->C_Finalize(NULL);
   if (CKR_OK != rv) {
      printf("\nC_Finalize() failed with %08lx\n", rv);
      retval = FAILED;
   }
   return retval;
}

int main(int argc, char *argv[])
{
   int err = 0;
```

```
   char *username = NULL;
   char *password = NULL;
   char *libname = NULL;
   char buName = FALSE;
   char bPassword = FALSE;
   char bLibray = FALSE;
   char *certName = NULL;
   char bCertName = FALSE;
   char *private_key_handle = NULL;
   char bPrivateKey = FALSE;
   int  cmd_options = 0;

   if (argc < 11) {
      printf
        ("\nMissing arguments: The format is %s -s <username> -p <password> -l <pkcs-lib> -c <cert-
name> -k <private-key-handle>\n\n", argv[0]);
      return FAILED;
   }

   if (argc > 11) {
      printf
        ("\nToo many arguments: The format is %s -s <username> -p <password> -l  <pkcs-lib> -c <cert-
name> -k <private-key-handle>\n\n", argv[0]);
      return FAILED;
   }

   while ((cmd_options = getopt(argc, argv,"s:p:l:c:k:")) != FAILED) {
      switch (cmd_options) {
          case 's' : username = optarg;
                buName = TRUE;
             break;
          case 'p' : password = optarg;
                bPassword = TRUE;
             break;
          case 'l' : libname = optarg;
                bLibray = TRUE;
             break;
          case 'c' : certName = optarg;
                bCertName = TRUE;
             break;
          case 'k' : private_key_handle = optarg;
                bPrivateKey = TRUE;
             break;
          default: printf("\nWrong arguments 1: The format is %s -s <username> -p <password> -l <pkcs-
lib> -c <cert-name> -k <private-key-handle>\n\n",argv[0]);
             return FAILED;
      }
   }
```

```
  if (!buName || !bPassword || !bLibray || !bCertName || !bPrivateKey) {
     printf
        ("\nWrong arguments 2: The format is %s -s <username> -p <password> -l <pkcs-lib> -c <cert-
name> -k <private-key-handle>\n\n", argv[0]);
     return FAILED;
  }

  if (load_lib(libname)) {
     printf("\n Error: loading library %s\n ", libname);
     err = FAILED;
     goto end;
  }

  if (!module) {
     printf("\n Error: module loading failure %s\n ", libname);
     err = FAILED;
     goto end;
  }

  /* prepare the pPin with given username and password */
  snprintf(pPin, sizeof(pPin), "%s:%s", username, password);

  /* prepare the pCert with given cert name */
  snprintf(pCert, sizeof(pCert), "%s", certName);

  /* prepare the private_key_handle with given private key handle */
  snprintf(priv_key_handle, sizeof(priv_key_handle), "%s", private_key_handle);

  err = main_func();
  if (err) {
     printf("Error while running the tests\n");
  }
 end:
  lib_close();
  return err;
}
```