Intel Cloud Integrity Technology 3.0

**Key Storage**

# Background

The key broker provides APIs for managing keys but it does not provide a secure key storage mechanism.

# Architecture

## Key Broker

The key broker provides APIs for creating, storing, transferring, revoking, or deleting keys and their associated metadata.

Keys and their metadata (key attributes) are both first-class concepts for the key broker. The key attributes must be protected because secure use of the key may depend on its attributes. For example, a private key may be used for decryption or signature, but not both. The key itself is capable of doing both but a secure system depends on the key attributes and its policy mechanism to prevent it from using the same private key for both types of operations.

Keys are protected for confidentiality.

Key attributes are protected for integrity. Key attributes are stored non-encrypted because they must be searchable.

The key broker enforces policies on the use and storage of keys. To ensure that attackers cannot circumvent the key broker’s policy enforcement mechanisms, the key broker encrypts keys stored in the key vault so the managed keys must be accessed through the key broker.

This means there is one key, called the master key or master password, which must be entered into the key broker directly.

## Key Vault

The key vault provides a secure key storage mechanism. The key vault must protect the key material from being exposed to unauthorized users. However, the key broker already encrypts the keys prior to storage.

The key vault may be a hardware security module, or a trusted software key vault.

The default key vault is a local directory-based storage which not intended for use in production, but allows a user to install and start the key broker to become familiar with its functions.

If a key vault does not implement an adequate key attribute storage and integrity protection mechanism, the key broker may supplement the key vault with separate key attribute storage.

## Key Attribute Store

The key attribute store provides a secure key attribute storage mechanism. The key attributes must be searchable. The key attribute store must protect the integrity of the key attributes.

The key attribute store should accommodate a fast response for a high volume of search operations, for example by indexing the attributes.

## Key Manager

The KeyManager interface is a Java application programming interface defined by the key broker. It encapsulates all key management operations.

One implementation of this interface, called RemoteKeyManager, encapsulates the policy enforcement and encryption of keys for storage.

Other implementations of this interface are used for integrating with various key vaults. For example, there are KMIP4J, Barbican, and Directory (local storage) implementations of KeyManager.

## Key Transfer Policy

The key transfer policy is a per-key policy that indicates who may perform operations on the key, who may retrieve the key or its metadata, and under what conditions.

## Crypto SDK

The cryptography software development kit integrates with applications to provide block or streaming encryption, decryption, as well as signature and verification functions with built-in key management.

The crypto SDK includes support for selecting an appropriate key based on the application context, for rotating keys automatically, and for handling a variety of crypto document formats such as OpenPGP, S/MIME, and XML DSIG.

# KeyManager API

## Create Key

Causes a new key to be generated and stored. If a key vault is used that also has secure key generation capabilities, the key generation is delegated to the key vault.

However, because generating a key on the key vault means that it is outside the key broker’s protection, the key broker uses output from the external key vault as a source of randomness and derives a new key based on this random source. The new derived key is then encrypted and stored in the key vault, and the original key created by the key vault is deleted.

This procedure applies to symmetric and asymmetric keys.

For asymmetric keys, see java.security.KeyPairGenerator which can accept a SecureRandom provider as input. The specified SecureRandom instance can be initialized using the random key material received from the key vault and salted with random material from the local system so the key is not predictable by the key vault.

## Register Key

Stores an existing key provided by the client. The key is encrypted before storing in the key vault.

The client is expected to properly discard its original key when it no longer needs it, and to obtain it in the future from the key broker.

## Delete Key

Deletes an existing key identified by the client. This operation is delegated to the key vault to delete the stored key.

## Transfer Key

Retrieves a key from the key vault, wraps it with a transport encryption key, and returns the wrapped key to the client. This operation involves policy enforcement regarding who may retrieve the key and under what conditions.

Every key may have its own key transfer policy.

The policy is declared in XML format. Alternative format may be JSON-LD.

Typically the key to retrieve is specified with a unique identifier. To support high-volume applications, the key to retrieve may also be specified as the result of a key attribute search. This allows a high-volume application to issue one request (transfer key based on search results) instead of two requests (search for key attributes, then transfer of the resulting key).

## Get Key Attributes

Retrieves the metadata associated with a key.

## Search Key Attributes

Retrieves the metadata associated with any key matching specified search criteria.

## Update Key Attributes

Modifies the metadata associated with an existing key. This can be used to revoke, activate, or deactivate an existing key.

Keys can be revoked by publishing key revocation lists, which are digitally signed lists identifying keys and their revocation dates. The key metadata can include a link to the revocation list which is stored separately.

The key manager and key attribute store must support extended attributes - that is, arbitrary attributes specified by the user. These extended attributes must be searchable and also have integrity protection like built-in attributes.

# Crypto SDK

The crypto SDK provides a high-level programming interface for applications to abstract the key management required for various cryptographic operations.

The functions in the crypto SDK are context-sensitive. An application using the crypto SDK must start by defining the cryptosystem and the key broker URL and credentials. The cryptosystem is defined by the combination of allowed algorithms, key lengths, cipher modes, and padding schemes.

The crypto SDK can be integrated with an application via a language binding (Java, C, Python, etc. library) or via a command-line tool that can obtain input from files or pipes including standard input and write output to files or pipes including standard output.

The Crypto SDK is further described in a separate blueprint.

# Permissions

The key broker restricts access to its APIs with user permissions.

The TransferKey method has a standard permission-based mechanism and also it can authorize a client to retrieve a key based on its trust report from Mt Wilson. The key transfer policy must authorize trust-based retrieval. When a client provides a SAML document to authorize trust-based retrieval, and the client is determined to be trusted by verifying the content and signature of the SAML document, the key broker does not need to authenticate the client for checking the normal permission system.

The standard permission-based mechanism is used to let the owner of the key retrieve it as needed, for example the Trust Director retrieves a key to encrypt a VM image. The trust-based mechanism is used to let anonymous or non-registered clients retrieve a key, for example a trusted host retrieves a key to decrypt a VM image.

# Integrations

## Cloud Integrity Technology 3.0

The key broker’s trust-based key transfer API provides the cornerstone for enforcing a tenant’s VM trust policy by authorizing key transfers only to hosts that are compliant with the tenant’s trust policy.

The Trust Director uses the CreateKey and TransferKey APIs to create and obtain VM encryption keys. The Trust Director discards the VM encryption key after encrypting the VM. The Trust Director includes the Key Transfer URL, provided by the key broker, in its VM trust policy.

A trusted host that needs to decrypt the VM image follows the URL to obtain a key. The key request must include the trust host’s AIK public key. The key request is intercepted by the Key Proxy, which contacts Mt Wilson attestation server to obtain a trust report for the requesting host, and then forwards the key request with the SAML trust report to the key broker. The key broker evaluates the trust report for the host, and if it matches the key transfer policy for the requested key the key broker extracts the public binding key for the trust host from the SAML trust report and uses it to wrap the VM encryption key. If the key transfer policy also indicates the key must be sealed to specific PCR values included in the SAML trust report, then to decrypt the VM encryption key the trusted host must maintain the same trusted state as shown in the trust report.

The key transfer policy should include the whitelist of operating system software and drivers that will restrict the use and storage of the data encryption keys so they are not leaked to systems outside of the boundary.

## Barbican

The OpenStack Barbican API can be used as the key vault.

A KeyManager implementation incorporating a Barbican API client has already been developed.

## KMIP4J

Any KMIP-compliant server can be used as the key vault.

A KeyManager implementation incorporating the open source KMIP4J client has already been developed.

## Cryptsoft

Any KMIP-compliant server can be used as the key vault.

This requires a KeyManager implementation incorporating the Cryptosoft KMIP client.

## Data Security Appliance

The DSA has the following key management requirements:

### Protection of SSL private key used in DSA web interface

This need is shared across our other projects including attestation server, trust agent, key broker, and trust director.

Non-interactive options:

* No protection
* Disk-based password protection; stored in /root for automatic start after reboot
* TPM-based password protection; stored in /root and sealed to TPM PCRs

Interactive options:

* Administrator enters password and starts service after reboot (no auto start)
* Trust-based key retrieval; DSA requests password from KMS and must be trusted by CIT before retrieving password (auto start possible but may fail due to separate services downtime even if DSA is trusted)

### SSL public key certificate renewal

This is a standard key management issue related to use of SSL keys, and is out of scope for the current key broker requirements. Products such as Venafi TrustAuthority™ SSL already exist to monitor and replace SSL certificates.

### Protection of data encryption keys

DSA uses one key per data chunk or object, and caches keys locally for quick look-up. The locally cached keys require protection. Even if they are stored only in memory, they might be written to disk during memory swapping.

Options:

* Store cached keys in memory only; setup dm-crypt for swap partition
* Eventually migrate the local key cache to be part of key management SDK
* When SGX becomes available, maintain cached keys in enclave

Currently the DSA uses a KMIP server directly to retrieve keys that are not in the cache. Options:

* Keep direct KMIP integration
* Insert key broker into architecture between DSA and KMIP with new APIs to improve performance with bulk operations; write new key broker API to search for and retrieve one or more keys specified as the result of a metadata search for a combination of key attributes and new key broker APIs to register multiple keys at a time and to create and retrieve multiple keys at a time

### Data encryption

The DSA encrypts and decrypts data in chunks. The identity of the chunk determines which key should be used. A new key is created for new chunks to encrypt.

Options:

* Eventually migrate the encryption and decryption to be part of key management SDK to implement: search for a key associated with an encrypted object, retrieve a specific key using an identifier, create a new key, associate a key with an encrypted object
* When SGX becomes available, encrypt and decrypt data within enclave

## Cryptomathic

The Cryptomathic key server has a rekeying feature and also allows for use of multiple key vault vendors.

The key broker can be integrated into Cryptomathic as a key vault using its JSON/HTTP interface by writing a plugin for Cryptomathic.

The Cryptomathic key server can be used as a drop-in replacement for the key broker by implementing the key transfer API in Cryptomathic key server for use by Intel Cloud Integrity Technology 3.0.

## Data Crypto Boundary

The data crypto boundary is comprised of the set of computing systems that request and use encryption keys and enforce that no data may be exported to other systems in plaintext form and that no keys may be exported to other systems.

Any system inside the data crypto boundary may request a new key from the key broker via the CreateKey API in order to encrypt new data.

Any system inside the data crypto boundary may request an existing key from the key broker via the TransferKey SAML API (trust-based key retrieval) in order to decrypt existing data.

The key transfer policy should include the whitelist of operating system software and drivers that will restrict the use and storage of the data encryption keys so they are not leaked to systems outside of the boundary.