CHAPTER 9

Hierarchies

A *hierarchy* is a collection of entities that are related and managed as a group. Those entities include permanent objects (the hierarchy handles), primary objects at the root of a tree, and other objects such as keys in the tree. NV indexes belong to a hierarchy but aren't in a tree. Entities, other than permanent entities, can be erased as a group.

The cryptographic root of each hierarchy is a *seed*: a large random number that the TPM generates and never exposes outside its secure boundary. The TPM uses the seed to create primary objects such as storage root keys. Those keys form the parent at the top of a hierarchy and are used to encrypt its children. Chapter 15 goes into far more detail about keys.

Each hierarchy also has an associated *proof value*. The proof can be independently generated or derived from the hierarchy seed. The TPM uses the proof value to ensure that a value supplied to the TPM was originally generated by that TPM. Often, the TPM derives an HMAC key from the proof, and HMACs data that the TPM itself generates internally. When the data is later supplied back to the TPM, the HMAC is checked to verify the authenticity of the data.

A hierarchy can be persistent (retained through a reboot) or volatile (erased at reboot). Each hierarchy is targeted at specific use cases: for the platform manufacturer, for the user, for privacy-sensitive applications, and for ephemeral requirements.

Three Persistent Hierarchies

TPM 1.2 has one hierarchy, represented by the owner authorization and storage root key (SRK). There can be only one SRK, always a storage key, which is the lone parent at the base of this single hierarchy. The SRK is generated randomly and can't be reproduced once it's erased. It can't be swapped out of the TPM. Child keys can't be created and wrapped with (encrypted by) the SRK, and these child keys may in turn be storage keys with children of their own. However, the key hierarchy is under the control of the one owner authorization; so, ultimately, TPM 1.2 has only one administrator.

TPM 2.0, on the other hand, expands to three persistent hierarchies (platform, storage, and endorsement) to permit several use cases:

- Using the TPM as a cryptographic coprocessor
- Enabling or disabling parts of the TPM
- Separating privacy-sensitive and -nonsensitive applications

The three hierarchies have some common traits:

- Each has an authorization value and a policy.
- Each has an enable flag.
- Each has a seed from which keys and data objects can be derived. The seed is persistent.
- Each can have primary keys from which descendants can be created.

The primary keys are somewhat analogous to the TPM 1.2 SRK. You could create a single RSA 2048-bit with a SHA-1 primary storage key, which would then be equivalent to the SRK.

However, TPM 2.0 adds more flexibility. First, primary keys aren't limited to storage keys. They can also be asymmetric or symmetric signing keys. Second, there can be more than one (indeed, an unlimited number of) primary keys. This is useful because you might want keys of different types (storage, signing) and of different algorithms (RSA, ECC, SHA-1, SHA-256). Third, because there can be a large number of primary keys, it's impractical to store them all in TPM NV memory. Therefore, unlike the TPM 1.2 SRK, the primary keys are derived from the secret seeds. The process is repeatable: the same seed value and key properties always result in the same key value. Rather than store them all, you can regenerate the keys as needed. Essentially, the seeds are the actual cryptographic roots. A primary key can be swapped out of the TPM, context-saved, and loaded for the duration of a power cycle, to eliminate the time required to regenerate the keys.

Because the hierarchies have independent authorization controls (password and policy), they can naturally have separate administrators. The TCG chose the three hierarchies and their slightly different operations to accommodate different use cases, which are somewhat reflected in their names. They're next described in detail, along with the intended use cases.

Platform Hierarchy

The platform hierarchy is intended to be under the control of the platform manufacturer, represented by the early boot code shipped with the platform.[1] The platform hierarchy is new for TPM 2.0. In TPM 1.2, the platform firmware could not be assured that the TPM was enabled. Thus, platform firmware developers could not include tasks that relied on the TPM.

[1]In an x86 PC platform, this early boot code was called BIOS. More recently, it's called UEFI firmware.

USE CASE: UEFI

The platform firmware must verify an RSA digital signature to authenticate software as part of the Unified Extensible Firmware Interface (UEFI) secure boot process. The platform OEM stores a public key, or a digest of a list of trusted public keys, in a TPM NV index. The controls on the index permit only the platform OEM to update it. During boot, the platform firmware uses this trusted public key to verify a signature.

The TPM provides two benefits. First, it provides a secure location to store the public key. Second, it offers the RSA algorithm, so it need not be implemented in software. Here are the steps:

1. `TPM_NV_Read`
2. `TPM2_LoadExternal`
3. `TPM2_VerifySignature`

Unique among the hierarchies, at reboot, the platform hierarchy is enabled, the platform authorization value is set to a zero-length password, and the policy is set to one that can't be satisfied. The intent is that the platform firmware will generate a strong platform authorization value (and optionally install its policy). Unlike the other hierarchies, which may have a human enter an authorization value, the platform authorization is entered by the platform firmware. Therefore, there is no reason to have the authorization persist (and to find a secure place to store it) rather than regenerate it each time.

Because the platform hierarchy has its own enable flag, the platform firmware decides when to enable or disable the hierarchy. The intent is that it should always be enabled and available for use by the platform firmware and the operating system.

Storage Hierarchy

The storage hierarchy is intended to be used by the platform owner: either the enterprise IT department or the end user. The storage hierarchy is equivalent to the TPM 1.2 storage hierarchy. It has an owner policy and an authorization value, both of which persist through reboots. The intent is that they be set and rarely changed.

The hierarchy can be disabled by the owner without affecting the platform hierarchy. This permits the platform software to use the TPM even if the owner disables its hierarchy. In TPM 1.2, turning off the single storage hierarchy disabled the TPM. Similarly, this hierarchy can be cleared (by changing the primary seed and deleting persistent objects) independent of the other hierarchies.

The storage hierarchy is intended for non-privacy-sensitive operations, whereas the endorsement hierarchy, with separate controls, addresses privacy.

Endorsement Hierarchy

The endorsement hierarchy is the privacy-sensitive tree and is the hierarchy of choice when the user has privacy concerns. TPM and platform vendors certify that primary keys in this hierarchy are constrained to an authentic TPM attached to an authentic platform. As with TPM 1.2, a primary key can be an encryption key; and certificates can be created using `TPM2_ActivateCredential`, equivalent to the TPM 1.2 activate identity command. Unlike with TPM 1.2, a primary key can also be a signing key. Creating and certifying such a key is privacy sensitive because it permits correlation of keys back to a single TPM.

Because the endorsement hierarchy is intended for privacy-sensitive operations, its enable flag, policy, and authorization value are independent of the other hierarchies. They're under the control of a privacy administrator, who may be the end user. A user with privacy concerns can disable the endorsement hierarchy while still using the storage hierarchy for TPM applications and permitting the platform software to use the TPM.

Privacy

Privacy, as used here, means the inability of remote parties receiving TPM digital signatures to *correlate* them—to cryptographically prove that they came from the same TPM. A user can use different signing keys for different applications to make correlation difficult. The attacker's task is to trace these multiple keys back to a single user.

Privacy sensitivity is most applicable to home users who own and control their platform. In an enterprise, the IT department may control the platform completely and weaken the privacy features. This discussion is also concerned mostly with remote correlation—it doesn't consider an attacker who can confiscate a platform.

The requirement for correlation is ensuring that the signing keys came from a single, authentic TPM. If the key can be duplicated on another TPM or is from a software implementation, the signature can't be traced back to a single device.

The TPM vendor generates an endorsement primary seed, generates one or more primary keys from this seed, and then generates certificates for these keys. The certificates attest that the key is from an authentic TPM manufactured by the vendor. The platform manufacturer may create an analogous platform certificate. From primary keys, other keys are in some way certified.

If a primary key is a signing key and directly certifies other signing keys, correlation is simple, because all signatures converge at the same certificate. An attester seeing the certificate chain could prove that the attestation came from an authentic device. Further, the certificate chain can indicate that the key was fixed to that particular TPM. For this reason, primary keys in the endorsement hierarchy are typically encryption keys, not signing keys.

When the primary key is an encryption key, the process to create a descendent key certificate uses a more complicated flow, called *activating* a credential. The certificate authority is referred to as a *privacy CA*, because it's trusted not to leak any correlation between the keys it has certified.

Activating a Credential

The TPM doesn't mandate a credential format, but the intent is something like an X.509 certificate, where a credential provider such as a CA signs a public signing key and a statement about the key's attributes. The credential process in the TCG model has multiple goals:

- The credential provider can be assured of the key attributes it's certifying.
- Receivers of the TPM key signatures can't determine that the multiple keys are resident on the same TPM.

The certificate authority could provide this correlation, but you can assume that this privacy CA would not normally do so.

In TPM 1.2, a key that can be activated is restricted to be an identity key (AIK), which isn't migratable (can't be backed up), is restricted to signing only TPM-generated data, and is always a child of the SRK. In TPM 2.0, all these restrictions have been removed while still achieving both of the previously stated goals.

In this description, recall that a TPM 2.0 key's `Name` is a digest of its public data. It completely identifies the key. The digest includes the public key and its attributes.

The simplified concept is that the primary key is a decryption key, not a signing key. The CA constructs a certificate and encrypts it with the primary key public key. Only the TPM with the corresponding private key can recover the certificate. See Figure 9-1.

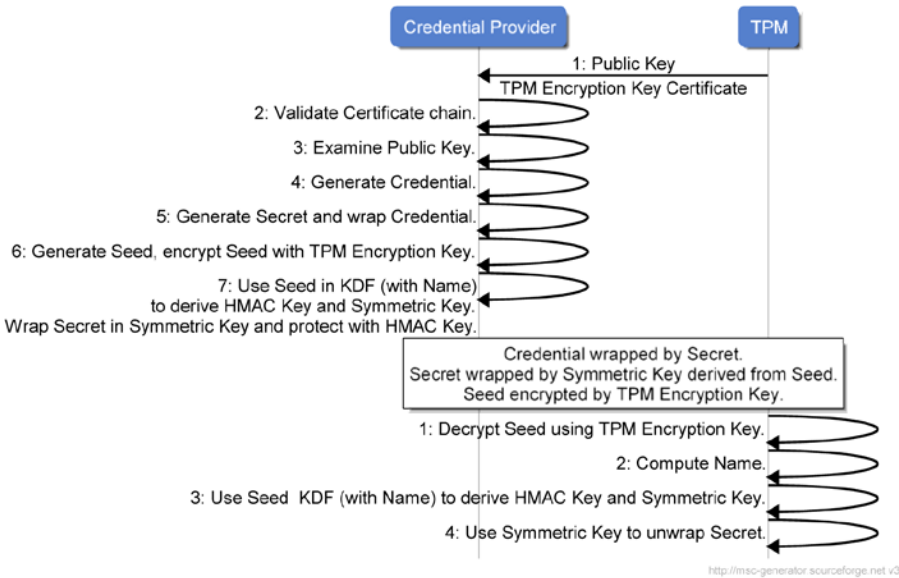

Figure 9-1. *Activating a Credential*

The following happens at the credential provider:

1. The credential provider receives the Key's public area and a certificate for an Encryption Key. The Encryption Key is typically a primary key in the endorsement hierarchy, and its certificate is issued by the TPM and/or platform manufacturer.
2. The credential provider walks the Encryption Key certificate chain back to the issuer's root. Typically, the provider verifies that the Encryption Key is fixed to a known compliant hardware TPM.
3. The provider examines the Key's public area and decides whether to issue a certificate, and what the certificate should say. In a typical case, the provider issues a certificate for a restricted Key that is fixed to the TPM.
4. The requester may have tried to alter the Key's public area attributes. This attack won't be successful. See step 5 in the process that occurs at the TPM.
5. The provider generates a credential for the Key
6. The provider generates a Secret that is used to protect the credential. Typically, this is a symmetric encryption key, but it can be a secret used to generate encryption and integrity keys. The format and use of this secret aren't mandated by the TCG.
7. The provider generates a 'Seed' to a key derivation function (KDF). If the Encryption Key is an RSA key, the Seed is simply a random number, because an RSA key can directly encrypt and decrypt. If the Decryption Key is an elliptic curve cryptography (ECC) key, a more complex procedure using a Diffie-Hellman protocol is required.
8. This Seed is encrypted by the Encryption Key public key. It can later only be decrypted by the TPM.
9. The Seed is used in a TCG-specified KDF to generate a symmetric encryption key and an HMAC key. The symmetric key is used to encrypt the Secret, and the HMAC key provides integrity. Subtle but important is that the KDF also uses the key's `Name`. You'll see why later.
10. The encrypted Secret and its integrity value are sent to the TPM in a credential blob. The encrypted Seed is sent as well.

If you follow all this, you have the following:

- A credential protected by a Secret
- A Secret encrypted by a key derived from a Seed and the key's `Name`
- A Seed encrypted by a TPM Encryption Key

These things happen at the TPM:

1. The encrypted Seed is applied against the TPM Encryption Key, and the Seed is recovered. The Seed remains inside the TPM.
2. The TPM computes the loaded key's `Name`.
3. The `Name` and the Seed are combined using the same TCG KDF to produce a symmetric encryption key and an HMAC key.
4. The two keys are applied to the protected Secret, checking its integrity and decrypting it.
5. This is where an attack on the key's public area attributes is detected. If the attacker presents a key to the credential provider that is different from the key loaded in the TPM, the `Name` will differ, and thus the symmetric and HMAC keys will differ, and this step will fail.
6. The TPM returns the Secret.

Outside the TPM, the Secret is applied to the credential in some agreed upon way. This can be as simple as using the Secret as a symmetric decryption key to decrypt the credential.

This protocol assures the credential provider that the credential can only be recovered if:

- The TPM has the private key associated with the Encryption Key certificate.
- The TPM has a key identical to the one presented to the credential provider.

The privacy administrator should control the use of the endorsement key, both as a signing key and in the activate-credential protocol, and thus control its correlation to another TPM key.

Other Privacy Considerations

The TPM owner can clear the storage hierarchy, changing the storage primary seed and effectively erasing all storage hierarchy keys.

The platform owner controls the endorsement hierarchy. The platform owner typically doesn't allow the endorsement primary seed to be changed, because this would render the existing TPM certificates useless, with no way to recover.

The user can create other primary keys in the endorsement hierarchy using a random number in the template. The user can erase these keys by flushing the key from the TPM, deleting external copies, and forgetting the random number. However, these keys do not have a manufacturer certificate.

When keys are used to sign (attest to) certain data, the attestation response structure contains what are possibly privacy-sensitive fields: `resetCount` (the number of times the TPM has been reset), `restartCount` (the number of times the TPM has been restarted or resumed), and the firmware version. Although these values don't map directly to a TPM, they can aid in correlation.

To avoid this issue, the values are obfuscated when the signing key isn't in the endorsement or platform hierarchy. The obfuscation is consistent when using the same key so the receiver can detect a change in the values while not seeing the actual values.

USE CASE: DETECTING A REBOOT BETWEEN ATTESTATIONS

An attestation server polls a platform at set intervals, verifying either that the PCRs haven't changed or that the new PCR values are trusted. In TPM 1.2, the platform may have transitioned to an untrusted state and then rebooted back to a trusted state. The server can't detect the reboot.

In TPM 2.0, the attestation data includes boot-count information. Although attestations in the storage hierarchy have the information obfuscated, the server can still tell that a value changed and thus that a reboot occurred.

Here are the steps:

1. `Execute the TPM2_Quote` command periodically.
2. Each quote returns a `TPM2B_ATTEST` structure.
3. The quote includes the `TPM2B_ATTEST->TPMS_CLOCK_INFO->resetCount` value.
4. `resetCount` is obfuscated with a symmetric key based on the quote key `Name`.
5. For the same key, the obfuscated `resetCount` has the same value if `resetCount` doesn't change.
6. For a different key, the obfuscated `resetCount` has a different value, preventing correlation.

Separate from the three persistent hierarchies is the one volatile hierarchy, called the NULL hierarchy.

NULL Hierarchy

The NULL hierarchy is analogous to the three persistent hierarchies. It can have primary keys from which descendants can be created. Several properties are different:

- The authorization value is a zero-length password, and the policy is empty (can't be satisfied). These can't be changed.
- It can't be disabled.
- It has a seed from which keys and data objects can be derived. The seed isn't persistent. It and the proof are regenerated with different values on each reboot.

A subtle use case, which the normal end user doesn't see, is that sessions, saved context objects, and sequence objects (digest and HMAC state) are in the NULL hierarchy. This permits them to be voided on reboot, because the seed and proof change. A user typically doesn't change the endorsement hierarchy seed (because it would invalidate certificates), the storage hierarchy seed (because it would invalidate keys with a long lifetime), or the platform hierarchy seed (because the user may not have that capability).

Ephemeral keys are keys that are erased at reboot. An entire hierarchy, primary keys, storage keys, and leaf keys can be constructed in the NULL hierarchy. On reboot, as the seed changes, the entire key hierarchy is cryptographically erased. That is, the wrapped keys may exist on disk, but they can't be loaded.

The TPM can be used as a cryptographic coprocessor, performing cryptographic algorithms on externally generated keys. Keys that have both a public and a private part are loaded in the NULL hierarchy, because they may not become part of a persistent hierarchy.

Cryptographic Primitives

TPM 2.0 can function purely as a cryptographic coprocessor. Although the following applications can be performed using any hierarchy, they're best suited for the NULL hierarchy because it's always enabled and the authorization is always a zero-length password. It's thus always available.

I hesitate to call the TPM a crypto accelerator, because it's likely to be slower than a pure software implementation. However, there are a few niche applications where these features are useful:

- A resource-constrained environment, such as early boot software, may not want to implement complex crypto math.
- In a low-performance application, it may be easier for a developer to use the TPM than to procure commercial software or vet an open source license.
- Applications may deem hardware superior to software.
- Applications could require a certified implementation, assuming the TPM is certified.

The TPM primitives, random numbers, digests, HMAC, and symmetric and asymmetric key operations are described next.

Random Number Generator

The TPM offers a simple interface to a hardware random number generator. It's particularly useful when another source of entropy may not be available. Examples are embedded systems or early in a platform boot cycle.

The TPM can be considered a more trusted source of random numbers than the software generator. See the paper "Ron was wrong, Whit is right"[2] for a discussion of issues resulting from poor software random number generators.

Digest Primitives

TPM 2.0 offers two cryptographic digest primitive APIs. Both are hash agile, permitting the hash algorithm to be specified in the call.

The simpler but less flexible option is `TPM2_Hash`. The caller inputs the message, and the TPM returns the digest. The message length is limited by the TPM input buffer size, typically 1 or 2 KB. The other API implements the usual start/update/complete pattern using `TPM2_HashSequenceStart`, `TPM2_SequenceUpdate`, and `TPM2_SequenceComplete`.

USE CASE: HASHING A LARGE FILE

In this use case, assume that the TPM input buffer is 2 KB. The user desires to SHA-256 hash a 4 KB file. The user uses the TPM because the SHA-256 algorithm isn't available in software. The user uses the sequence commands because the file is larger than 2 KB.

Here are the steps:

1. `TPM2_HashSequenceStart`, specifying the SHA-256 algorithm.
2. `TPM2_SequenceUpdate` two times, with a sequence of 2 KB buffers.
3. `TPM2_SequenceComplete` to return the result.

This API is similar to that of TPM 1.2, but it has several enhancements:

- It supports multiple hash algorithms.
- The `start` function returns a handle. More than one digest operation can be in progress at a time.

[2]Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung, and Christophe Wachter, "Ron was wrong, Whit is right," *International Association for Cryptologic Research*, 2012, https://eprint.iacr.org/2012/064.pdf.

- The `update` function isn't restricted to a multiple of 64 bytes.
- The `complete` function can be more than 64 bytes.
- The `complete` function can return a ticket, which is used when signing with a restricted key. See the discussion of `TPM_GENERATED` for details.

USE CASE: TRUSTED BOOT

CRTM software would like to verify a signed software update. Because it's resource constrained, it uses the TPM to digest the update, avoiding the need to implement the digest calculation in the CRTM.

TPM 2.0 offers `TPM2_EventSequenceComplete` as an alternative to `TPM2_SequenceComplete`. This command can only terminate a digest process where no algorithm was specified. This null algorithm causes the TPM to calculate digests over the message for all supported algorithms.

The command, an extension of TPM 1.2's `TPM_SHA1CompleteExtend`, has two enhancements:

- It permits the result of the digest operation to be extended into a PCR. One PCR index is specified, but all PCRs at that index (that is, all banks) are extended with a digest corresponding to that PCR bank's algorithm. Multiple digests are returned, one for each supported algorithm.
- The SHA-1 algorithm is being deprecated in favor of stronger algorithms such as SHA-256. This command, which can do both algorithms simultaneously, permits a staged phase-out of SHA-1, because it can return multiple results and extend multiple PCR banks.

USE CASE: TRUSTED BOOT

`TPM2_EventSequenceComplete` allows software to measure (digest and extend) software using the TPM. It avoids the need for the measuring software to implement a digest algorithm.

USE CASE: MULTIPLE SIMULTANEOUS TPM DIGEST ALGORITHMS

In one pass, the software can measure into PCR banks for several algorithms. A TPM may be unlikely to support multiple simultaneous PCR banks (multiple sets of PCRs with different algorithms). The current PC Client TPM doesn't require this. But if one ever does, the TPM API supports it.

HMAC Primitives

TPM 2.0 supports HMAC as a primitive, whereas TPM 1.2 offered only the underlying digest API. The HMAC key is a loaded, keyed, hash TPM object. For a restricted key, the key's algorithm must be used. For an unrestricted key, the caller can override the key's algorithm. As with any key, the full complement of authorization methods is available.

As with digests, there are both simple and fully flexible APIs. `TPM2_HMAC` is the simpler API. You input a key handle, a digest algorithm, and a message, and the TPM returns the HMAC. The other API again implements the usual start/update/complete pattern, using `TPM2_HMAC_Start`, `TPM2_SequenceUpdate`, and `TPM2_SequenceComplete`.

USE CASE: STORING LOGIN PASSWORDS

A typical password file stores salted hashes of passwords. Verification consists of salting and hashing a supplied password and comparing it to the stored value. Because the calculation doesn't include a secret, it's subject to an offline attack on the password file.

This use case uses a TPM-generated HMAC key. The password file stores an HMAC of the salted password. Verification consists of salting and HMACing the supplied password and comparing it to the stored value. Because an offline attacker doesn't have the HMAC key, the attacker can't mount an attack by performing the calculation.

Here are the steps:

1. `TPM2_Create`, specifying an HMAC key (called a `keyedHash` object) that can be used with a zero-length, well-known password.
2. `TPM2_Load` to load the HMAC key, or optionally load and then `TPM2_EvictControl` to make the key persistent.
3. `TPM2_HMAC` to calculate an HMAC of the salted password.

RSA Primitives

Two commands offer raw RSA operations: `TPM2_RSA_Encrypt` and `TPM2_RSA_Decrypt`. Both operate on a loaded RSA key. Both permit several padding schemes: PKCS#1, OAEP, and no padding. The loaded key's padding can't be overridden. The caller can, however, specify a padding scheme if the key's scheme is null.

`TPM2_RSA_Decrypt` is the private key operation. The decryption key must be authorized, and padding is validated and removed before the TPM returns the plaintext.

`TPM2_RSA_Encrypt` is the public key operation. A key and a message must be specified, but no authorization is required for this public key operation. Padding is added before the encryption.

USE CASE: CRTM SIGNATURE VERIFICATION

A platform implements CRTM updates. The design requires that updates be signed, because compromising the CRTM subverts the entire platform. However, the CRTM is constrained in both code and data space. The CRTM would like to use the TPM to verify the signature.

The CRTM uses a hard-coded public key blob in a format ready to be loaded on the TPM. The key has a null padding scheme. The CRTM then uses the `TPM2_RSA_Encrypt` command to apply the public key to the signature, specifying no padding. Finally, the CRTM does a simply byte compare on the result against a padded digest of the update.

Symmetric Key Primitives

`TPM2_EncryptDecrypt` permits the TPM to perform symmetric key encryption and decryption. The function operates on a small number of blocks due to the TPM input buffer size. However, the API includes an initialization vector (IV) on input and a chaining value on output, so a larger number of blocks can be operated on in parts.

As with an HMAC key, a restricted key has a fixed mode. The caller can specify the mode when using an unrestricted key.

The key must be a symmetric cipher object. It must be authorized, and the full set of authorization options are available.

Symmetric key encryption is a sensitive subject. Although the TPM isn't very fast, its hardware-protected keys are far more secure than software keys. It thus may be subject to import controls and may draw the attention of government agencies. For this reason, the PC Client platform specifies this command as optional.

Summary

The TPM has three persistent hierarchies. The platform hierarchy is generally used by the platform OEM, as represented by early boot code. The platform OEM can depend on this hierarchy being enabled, even if the end user turns off the other hierarchies. The storage hierarchy is under the control of the user and is used for non-privacy-sensitive operations. The endorsement hierarchy, with its TPM vendor and OEM certificates, is under the control of a privacy administrator and is used for privacy-sensitive operations. The privacy-sensitive credential activation is typically performed in the endorsement hierarchy.

The NULL hierarchy is volatile. Sessions, contexts, and sequence objects are in this hierarchy, but an entire tree of volatile keys and objects can also be created here, ensuring that they're deleted on a power cycle.

Besides its secure storage features, the TPM can be used as a cryptographic coprocessor, performing cryptographic algorithms on externally generated secrets or algorithms for which no secrets are needed. Its capabilities include a random number generator, digest and HMAC algorithms, and symmetric and asymmetric key operations.