CHAPTER 10

Keys

As a security device, the ability of an application to use keys while keeping them safe in a hardware device is the TPM's greatest strength. The TPM can both generate and import externally generated keys. It supports both asymmetric and symmetric keys. Chapter 2 covered the basic principles behind these two key types.

As a memory-constrained device, it acts as a key cache, with the application securely swapping keys in and out as needed. This key cache operation is discussed in the "Key Cache" section.

There are three key hierarchies under the control of different security roles, and each can form trees of keys in a parent-child relationship. Chapter 9 covered the hierarchies and their use cases.

Each key has individual security controls, which can include a password, an enhanced authorization policy, restrictions on duplication to another parent or another TPM, and limits on its use as a signing or decryption key. Keys can be both certified and used to certify other keys. Attributes specific to keys are discussed in the "Key Types and Attributes" section. The details of authorization common to all TPM entities, including password and policy, are deferred to Chapters 13 and 14.

Key Commands

Following is a summary of the TPM commands most often used with keys. It isn't a complete list. See the TPM 2.0 specification, Part 3, for the complete command set and API details. They're used in the descriptions and use cases that follow, as well as in subsequent chapters:

- `TPM2_Create` and `TPM2_CreatePrimary` create all key types from templates.
- `TPM2_Load` (for wrapped private keys) and `TPM2_LoadExternal` (for public keys and possibly plaintext private keys) load keys onto the TPM.
- `TPM2_ContextSave` and `TPM2_ContextLoad` are used to swap keys in and out of the TPM key cache. `TPM2_FlushContext` removes a key from the TPM. `TPM2_EvictControl` can make a loaded key persistent or remove a persistent ley from the TPM. These functions and their applications are explained in detail in Chapter 18.

- `TPM2_Unseal`, `TPM2_RSA_Encrypt`, and `TPM2_RSA_Decrypt` use encryption keys.
- `TPM2_HMAC`, `TPM2_HMAC_Start`, `TPM2_SequenceUpdate`, and `TPM2_SequenceCompete` use symmetric signing keys and the keyed-hash message authentication code (HMAC) algorithm.
- `TPM2_Sign` is a general-purpose signing command, and `TPM2_VerifySignature` verifies a digital signature.
- `TPM2_Certify`, `TPM2_Quote`, `TPM2_GetSessionAuditDigest`, and `TPM_GetTime` are specialized signing commands that sign attestation structures. In particular, `TPM2_Certify` can be used to have a TPM key sign another key (specifically its `Name`). Thus, the TPM can be used as a certificate authority, where the issuer key attests to the properties of the subject key.

Key Generator

Arguably, the TPM's greatest strength is its ability to generate a cryptographic key and protect its secret within a hardware boundary. The key generator is based on the TPM's own random number generator and doesn't rely on external sources of randomness. It thus eliminates weaknesses based on weak software random number generators or software with an insufficient source of entropy.

Primary Keys and Seeds

TPM keys can form a hierarchy, with parent keys wrapping their children. Primary keys are the root keys in the hierarchy. They have no parent. Chapter 9 discussed the general concept of hierarchies and their use cases. Their specific application to keys is discussed under "Key Hierarchy."

This section describes, in a linear flow, the creation and destruction of primary keys. In the narrative, the caller is some software that is provisioning the TPM, sending commands and receiving responses, whereas the TPM is the device that processes the commands. Provisioning software (see Chapter 19) typically performs these steps. Although end users may use primary keys, they would not typically be creating them.

Primary keys are created with the aptly named command `TPM2_CreatePrimary`. If you're familiar with TPM 1.2, you know that it has one key equivalent to the TPM 2.0 primary key: the *storage root key* (SRK), which is persistently stored in the TPM. TPM 2.0 permits an unlimited number of primary keys, which don't need to be persistent. Although you might think the number would be limited by the TPM persistent storage, it's not. Primary seeds, described shortly, permit the expansion.

There were two reasons TPM 1.2 could function with one SRK. First, it had only one algorithm and key size for wrapping keys, RSA-2048. The design of TPM 2.0, of course, permits multiple algorithms and key sizes. Second, TPM 1.2 has only one key hierarchy: the storage hierarchy. TPM 2.0 has three hierarchies, each with at least one root. Chapter 9 discussed the general concept of hierarchies and their use cases.

How can a TPM with limited persistent storage have an unlimited number of root keys? A root can't exist outside the TPM because it has no parent to wrap its secret parts. The answer is the primary seeds.

Each of the three persistent hierarchies has an associated primary seed: the *endorsement primary seed*, the *platform primary seed*, and the *storage primary seed*. These seeds never leave the TPM. They're the secret inputs to key-derivation functions. When the TPM creates a primary key, it uses a primary seed plus a public template. The template includes all the items you would normally expect when specifying a key: the algorithms and key size, its policy, and the type of key (signing, encryption, and so on). The caller can also provide unique data in the template. The unique data is input in the public key area of the template.

The key-derivation function is fixed and repeatable. For the same seed, the same template always produces the same key. By varying the unique data in the template, the caller can create an unlimited number of primary keys.

When the TPM creates a primary key, it remains on the TPM in volatile memory. The caller now has two choices. A limited number of primary keys can be moved to persistent memory using the `TPM2_EvictControl` command. Other keys can remain in volatile memory.

If more primary keys are needed than can fit in persistent storage or volatile memory, some can be flushed (from volatile storage) or moved from persistent storage and then flushed. Because the seed is persistent, the key isn't lost forever. If the caller knows the template, which may be completely public, the TPM can re-create the identical key on demand. If the key being regenerated is an RSA key, this process may take a lot of time. If the key is an elliptic curve cryptography (ECC), AES, or HMAC key, the process of creating a primary key is very fast. In most use cases, at least one storage primary key is made persistent in the TPM for the storage hierarchy, to act in a manner similar to the SRK.

How would this work in practice? In TPM 1.2, there was one endorsement key and an associated certificate signed by the TPM vendor. They resided in persistent storage, so that when the final user got a system with a TPM on it, the user also had a certificate stored in the TPM's NVRAM that matched the endorsement key stored in the TPM. In TPM 2.0, there can be many key/certificate pairs—at least one for each algorithm the TPM implements. However, the end user may not want to consume valuable persistent storage for keys and certificates that aren't being used, even if they could fit.

A possible solution, which TPM vendors are expected to implement, is to have the manufacturer use the endorsement seed to generate several endorsement primary keys and certificates using a standard set of algorithms, each with a well-known template. One popular key, say RSA-2048, and its certificate can be moved to persistent storage. The vendor flushes the other keys but retains the certificates.

The TCG Infrastructure work group has defined several such templates for endorsement primary keys. The RSA template uses RSA 2048, SHA-256, and AES-128. The ECC template uses ECC with the NIST P256 curve, SHA-256, and AES-128. Both use the same authorization policy, which requires knowledge of the endorsement hierarchy password. This delegates the key authorization to the endorsement hierarchy administrator. The unique data is empty, a trivial well-known value. The attributes (see "Key Types and Attributes") are `fixedTPM` and `fixedParent` true, as expected for an endorsement key that should never be duplicated. `userWithAuth` and `adminWithPolicy`

are specified so that a policy must always be used, not a password, which is appropriate because the TPM vendor has no way of passing a password to the end user. The key is a *restricted decrypt* key: that is, a storage key.

Suppose the end user desires a different primary key. That user can flush the one that was provisioned with the TPM and generate a new one with their algorithm of choice.

Magic happens now! Because the seed is unchanged and the user creates the primary key using the same template, they get the exact same key that the TPM vendor created. The user can treat the public part as an index into a TPM vendor certificate list. That list could even be on a public server. The user retrieves the certificate and is ready to go. This key-generation repeatability (the same seed and the same template always yield the same key) permits the TPM vendor to generate many keys and certificates during manufacturing, but not have to store them in the limited TPM nonvolatile storage. The end user can regenerate them as needed.

Note that the vendor must generate all needed primary keys and vendor certificates in advance. Because the seed is secret, the vendor would otherwise not be able to determine that a public key value came from the vendor's TPM.

Once a seed is changed, the primary keys can no longer be re-created, and any keys residing in the TPM based on the old seed are flushed. This means any certificates the vendor created also become worthless. Creating a new certificate for a TPM endorsement key (EK) signed by the vendor would be very difficult. Because of this, changing the seed to the endorsement hierarchy is controlled by the platform hierarchy, which in practice means the OEM. This makes it difficult for an end user to change this seed. On the other hand, by simply choosing a random input in the template, the end user can create their own set of endorsement keys that are totally independent of the EKs the vendor produced.

USE CASE: MULTIPLE PRIMARY KEYS

The user has several primary storage keys that serve as the root for a key hierarchy. They can't all fit in persistent storage. If the user creates the keys using well-known templates, they can be re-created as needed.

The TPM commands are as follows:

- `TPM2_NV_Read`: Reads the well-known template from TPM NV space. The TPM vendor may provision several templates (for example, one for RSA and one for ECC) on the TPM, and these templates match the provisioned key certificates. The user may also have enterprise-wide templates.
- `TPM2_CreatePrimary`: Specifying the template.
- `TPM2_EvictControl`: Can optionally be used to make several keys persistent. Especially for RSA keys, this saves the time required to regenerate them. Keys can also remain in volatile memory and be re-created after each power cycle.

USE CASE - CUSTOM PRIMARY KEYS:

The user wishes to create a primary key using a user secret in the template rather than using a well-known template. Again, there are more primary keys than can fit in persistent storage. The user stores the secret in a TPM NV index, with suitable read access control, and retrieves it when needed to re-create the primary key.

The TPM commands are as follows:

- `TPM2_NV_Write`: Writes and protects a user secret.
- `TPM2_NV_Read`: Reads the secret using appropriate authorization. The secret is inserted into the key template.
- `TPM2_CreatePrimary`: Specify the template, which includes a user secret, to generate a custom primary key.

Persistence of Keys

A user calls the `TPM2_EvictControl` command to move a key from volatile to nonvolatile memory so it can remain loaded (persist) though power cycles. No key needs to be made persistent to be used. Typically, we expect that a small number of primary keys, perhaps one per hierarchy, will be made persistent to improve performance.

Keys in the endorsement, storage, and platform hierarchies, other than primary keys, can also be made persistent. A use case would be early in a boot cycle, when a key is needed before a disk is available. Another use case is a limited-resource platform such as an embedded controller, which may not have any external persistent storage.

No keys in the NULL hierarchy can be made persistent. All are voided at reboot.[1]

Only a limited number of keys can be persistent, but the TPM can handle an unlimited number of keys. The application does this by using the TPM as a key cache.

Key Cache

For keys other than primary keys, the TPM serves as a key cache. That is, the `TPM2_Create` command creates a key, wraps[2] (encrypts) it with the parent, and returns the wrapped key to the caller. The caller saves the key external to the TPM, perhaps on disk. To use the key, the user must first load it into the TPM under its parent using `TPM2_Load`. When finished, the caller can free memory using `TPM2_FlushContext`. This is different from a primary key, which has no parent and remains in the TPM after it's created.

[1]Chapter 9 discussed the unique properties of the NULL hierarchy.

[2]Wrapping is a common design pattern for hardware security modules. The wrapping key is an encryption key, sometimes called a *key encrypting key* or *master key*. The TCG calls it a *storage key*. The wrapping key and the wrapped key form a parent-child relationship.

A typical hardware TPM may have five to ten *key slots*: memory areas where a key can be loaded. TPM management middleware is responsible for swapping keys in and out of the cache.

If you read Chapter 13, you may notice that the key *handle* isn't included in the TPM parameters that are authorized. Rather, the key's `Name` is used. The reason is the key cache and swapping. A platform may have a large number of application keys on disk, perhaps identified by a user's handle. There are many more of these handles than key slots. When a user asks to use a key, the command includes the user's handle. However, when the middleware loads the key, it gets a different handle, related to the TPM key slot rather than the user's handle. The middleware must thus replace the user's handle with the TPM handle. If the authorization included the user's handle, the substitution would cause an authorization failure.

You may now ask, "If the handle can be replaced, then if I have two keys with the same authorization secret, how do I know that the middleware didn't use a different key than the one I wanted?" This was indeed a potential problem in TPM 1.2.

TPM solves this problem by using the key's `Name`, a digest of the key's public area, in the authorization. The middleware can replace the key handle (which was not authorized) but can't replace the `Name` (which was authorized).

The root keys (the parents) and the key cache (the children) form a tree of keys. The TPM provides for four of these trees, each with different controlling roles. The trees are called *hierarchies*.

Key Authorization

Although hardware protection of private or symmetric keys alone is a major improvement over software-generated keys, the TPM also offers strong access control. A software key often uses a password for access control, to protect the key. For example, the secret key may be encrypted with a password. This protection is only as strong as the password, and the secret key is vulnerable to an offline hammering attack. That is, once an attacker obtains the encrypted key, extracting the key is reduced to cracking the password. The key owner can't prevent a high-speed attack that tries an unlimited number of passwords. This attack can be parallelized, with many computers trying different passwords simultaneously. The cloud has made this kind of attack very feasible.

The TPM improves on software keys in two respects. First, when the key leaves the TPM (see the "Key Hierarchy" section), it's wrapped (encrypted) with a strong parent key encrypting key. The attacker now has to crack a strong key rather than a weak password. Second, when a key is loaded in the TPM, it's protected by what the specification calls *dictionary attack protection logic*. Each time an attacker fails to crack the key's authorization,[3] this logic logs the failure. After a configurable number of failures, the TPM blocks further attempts for a configurable amount of time. This limits, possibly severely, the speed at which an attacker can try passwords. The rate limiting can make even a weak TPM key password much more time consuming to crack than a strong software key password, where the attack isn't rate limited. Chapter 13 describes password and HMAC authorization in detail.

[3]Chapters 13 and 14 discuss the details of TPM authorization.

The TPM provides many access-control mechanisms beyond a simple password. However, it's the hardware protection of the dictionary-attack protection logic that makes a TPM key password resistant to attack.

Key Destruction

Sometimes a key should be destroyed. Perhaps the authorization has been exposed. Perhaps the machine is being repurposed. Keys that are stored in software can never be destroyed, because they may have been copied almost anywhere. But TPM keys have parents or are primary keys.

As described in Chapter 9, there are three persistent hierarchies (endorsement, storage, and platform) plus one volatile hierarchy (the null hierarchy). Each hierarchy has its unique primary seed. Erasing a primary seed prevents re-creation of primary keys in that hierarchy—obviously a drastic and rarely performed action. Erasing the primary keys then prevents their children from being loaded in the TPM. Any key with attributes that prove it can only exist in the TPM is then destroyed.

Key Hierarchy

A hierarchy can be thought of as having parent and child keys, or ancestors and descendants. All parent keys are storage keys, which are encryption keys that can wrap (encrypt) child keys. The storage key thus protects its children, offering secrecy and integrity when the child key is stored outside the secure hardware boundary of the TPM. These storage keys are restricted in their use. They can't be used for general decryption, which could then leak the child's secrets.

The ultimate parent at the top of the hierarchy is a primary key. Children can be storage keys, in which case they can also be parents. Children can also be non-storage keys, in which case they're *leaf keys*: children but never parents.

Key Types and Attributes

Each key has attributes, which are set at creation. They include the following:

- Use, such as signing or encryption
- Overall type, symmetric or asymmetric, and the algorithm
- Restrictions on duplication
- Restrictions on use

Symmetric and Asymmetric Keys Attributes

TPM 2.0 supports a variety of asymmetric algorithms, unlike TPM 1.2, which was fixed to RSA. TPM 2.0 also introduces some entirely new key types.

A symmetric signing key can be used in TPM HMAC commands. TPM 2.0 can do symmetric signing (a MAC) with a key that is never in the clear outside the TPM.

The TPM library specification includes symmetric encryption keys that can be used for general-purpose encryption such as AES. It's uncertain whether TPM vendors will include these functions, due to potential export restrictions. The commands are optional in the PC Client platform specification. Historically, TPM vendors haven't implemented optional TPM features.

Duplication Attributes

Duplication is the process of copying a key from one location in a hierarchy to another. The key can become the child of another parent key. The hierarchy or parent can be on the same or a different TPM. Primary keys can't be duplicated; they're fixed to one hierarchy on one TPM.

A primary use case for duplication is key backups. If a key were locked forever to one TPM, and the TPM or its motherboard failed, the key would be lost permanently. A second use case is the sharing of keys among several devices. For example, a user's signing key may be duplicated among a laptop, tablet, and mobile phone.

TPM 1.2 has a similar process called migration. The term *migration* implies that a key is moved: that is, that it would now exist at the destination location but no longer exist at the source. This implication was incorrect. After migration, the key could exist at both the destination and the source. For that reason, the TPM 2.0 term was changed to the more accurate *duplication*.

TPM 2.0 keys have two attributes that control duplication. At one extreme, a key may be locked to a single parent on a single TPM, and never duplicated. The opposite extreme is a key that may be freely duplicated to another parent on the same or another TPM.

The intermediate case is a key that is locked to a parent but that can be implicitly duplicated if the parent is moved. This case offers the possibility of duplicating an entire branch of a tree. If the parent is duplicated, all children wrapped to that parent are available at the destination, on down through all descendants.

The TPM specification talks of a *duplication root* and a *duplication group*. The root is a key that can be duplicated. The duplication process acts explicitly on that key. The group represents all descendants of that root. The entire duplication group duplicates implicitly when the root duplicates. The children, which aren't explicitly duplicated, remain with their parent. However, as the parent is copied, the children are implicitly copied with it.

The controlling key attributes are defined as follows:

- `fixedTPM`: A key with this attribute set to true can't be duplicated. Although the name seems to permit duplicating a key from one location in a hierarchy to another within a TPM, this isn't the case.
- `fixedParent`: A key with this attribute set to true can't be duplicated to (rewrapped to) a different parent. It's locked to always have the same parent.

These two boolean attributes define four combinations.

1. The easiest case to understand is `fixedTPM` true and `fixedParent` false, because it isn't permitted. A key with fixedTPM true can't be duplicated, whereas fixedParent false says it can be moved to a different parent The TPM checks for and doesn't allow this inconsistency.
2. `fixedTPM` true and `fixedParent` true defines an object that can't be duplicated, either explicitly or implicitly.
3. `fixedTPM` false and `fixedParent` true indicates a key that can't be directly duplicated. It's fixed to a parent. However, if an ancestor is duplicated, this key naturally moves with it. That is, it may be in a duplication group, but it isn't the root of a group.
4. `fixedTPM` false and `fixedParent` false indicates a key that can be duplicated. If it's a parent, a duplication root, its children move with it.

The fourth case is perhaps the most interesting, because the key may be a duplication root. For example, it permits backup of a group of keys, called a *duplication group* in the specification. That is, once this parent is duplicated, all descendants are immediately duplicated to the new location without the need to duplicate each child individually. It also simplifies the task of tracking the location of a key. You need only track the parent, not children with `fixedParent` true, which remain with their parent.

Observe also that these children are still wrapped by their original parent. The key being duplicated must have `fixedParent` false. The children can be loaded into the TPM where their parent is loaded, regardless of where their parent was originally loaded. `fixedParent` determines whether a key can be directly duplicated, not whether it can or can't be duplicated by implication when its parent is duplicated. In other words, the child wasn't duplicated through any operation involving the TPM. Once its parent is duplicated, the child can be simply moved to the new location (for example, with a file copy of the wrapped child key) and loaded.

A child can have more than one parent. The duplication process establishes a new parent-child relationship but doesn't destroy the old one.[4] The key is now a child of both the original parent and the new parent. A key can be part of more than one duplication group if more than one of its ancestors has `fixedParent` false. That is, a child key in a tree can have more than one ancestor that is a duplication root. If any root is duplicated, the child is duplicated.

[4] It's for this reason that the TPM 1.2 term *migration* was changed to *duplication*. Migration implied that the old parent-child relationship was severed, which isn't true even in TPM 1.2.

The TPM puts a restriction on the relationship between parent and child. A child can only be created with `fixedTPM` true if:

1. Its parent also has `fixedTPM` true (the parent can't be explicitly duplicated).
2. Its parent has `fixedParent` true (the parent can't be implicitly duplicated).

The TPM enforces this restriction back to the primary keys, which are by nature fixed to their TPM.

Restricted Signing Key

A variation on the key attribute `sign` (a signing key) is the `restricted` attribute. The use case for a restricted key is signing TPM attestation structures. These structures include Platform Configuration Register (PCR) quotes, a TPM object being certified, a signature over the TPM's time, or a signature over an audit digest. The signature is, of course, over a digest, but the verifier wants assurance that the digest was not simply created externally over bogus values and delivered to the TPM for signing. For example, a quote is a signature over a set of PCR values, but the actual signing process signs a digest. A user could generate a digest of any PCR values and use a nonrestricted key to sign it. The user could then claim that the signature was a quote. However, the relying party would observe that the key was not restricted and thus not trust the claim. A restricted key provides assurance that the signature was over a TPM generated digest.

A restricted signing key can only sign a digest produced by the TPM. This is a generalization of the TPM 1.2 Info keys and attestation identity key (AIKs), which could only sign a TPM internally created structure. For internal TPM data, this assurance is easy, because the TPM created the digest from its internal data at signing time.

However, a restricted key can also sign data supplied to the TPM, as long as the TPM performed the digest using either `TPM2_SequenceComplete` or `TPM2_Hash`. Because the digest is later supplied to the TPM for signing, how does the TPM know that it calculated the digest?

The answer is a *ticket*. When the TPM calculates the digest, it produces a ticket that declares that the TPM itself calculated that digest. When the digest is presented to `TPM2_Sign`, the ticket must accompany it. If not, the restricted key doesn't sign.

So what? How does this restrict what can be signed? If you can digest any external data and obtain a ticket, why would it matter where the digest was calculated?

The answer is a 4-byte magic value called `TPM_GENERATED`. Each of the attestation structures—the structures the TPM constructs from internal data—begins with this magic number. If the TPM is digesting externally supplied data, it produces a ticket only if the data did *not* begin with the magic number.

The net result is that you can sign almost any externally supplied data with a restricted key. The only data that you can't sign is data beginning with `TPM_GENERATED`. This prevents you from spoofing TPM attestation structures, which all start with that value.

Restricted Decryption Key

A restricted decryption key is in fact a storage key. This key only decrypts data that has a specific format, including an integrity value over the rest of the structure.

Only these keys can be used as parents to create or load child objects or to activate a credential. These operations place restrictions on the result of the decryption. For example, loading doesn't return the result of the decryption.

An unrestricted key can perform a general-purpose decryption on any supplied data and return the result. If it were permitted to be used as a storage key, it could decrypt and return the private key of a child. If it could be used on sealed data, it would return the data without checking the unseal authorization.

Context Management vs. Loading

Loading a key involves supplying the wrapped (encrypted) key and specifying a loaded parent. The TPM parent key unwraps (decrypts) the child key and holds it in a volatile key slot.

Context management involves *context-saving* a loaded key off the TPM and then *context-loading* it onto the TPM at a later time. When the key is saved, it's wrapped with a symmetric key derived from a hierarchy secret, called a *hierarchy proof*. Upon load, it's unwrapped with the same key. A context-saved key has no parent, but it's connected to a hierarchy.

Why use one or the other? In TPM 1.2, context management was important, because child keys were always wrapped with a parent RSA key. The load operation required a time-consuming RSA decryption. Context-saved keys were wrapped with a symmetric key and thus were much faster. In TPM 2.0, child keys are wrapped with the symmetric key of the parent, even if the parent is itself an asymmetric key. All storage keys have a symmetric secret. Thus, reloading a key using its parent should be as fast as a context load and of course eliminates the context save.

So why ever use context management to load a key? The use case for context-loading keys is when the parent isn't loaded. The key could be a descendent deep down a hierarchy. Loading it could require loading a long chain of ancestors. A parent authorization may require an inconvenient password prompt. A parent authorization may be impossible if, for example, its policy requires a PCR state that has passed.

Specifically, suppose a key is four layers of parent down from a primary key. The first child is loaded under its parent. That parent is no longer needed and can be flushed from the TPM's key cache. Now the next child is loaded, and the process repeats four times until the final leaf key is reached. Once the leaf key is loaded, all its ancestors can be flushed. However, if the leaf key is flushed, the entire process must repeat. The alternative is to context-save the leaf key. Then it can be context-loaded independent of its ancestors. Chapter 18 explains this process in detail.

NULL Hierarchy

In addition to the three persistent hierarchies, the TPM has a NULL hierarchy.[5] This hierarchy has its own unique seed, and both primary and descendent keys can exist in this hierarchy. However, neither the seed nor primary keys can be persistent. A new seed is created on each TPM reset. Thus, keys in this hierarchy are *ephemeral*: they're erased on a reset.

Certification

The TPM can of course act as a certificate authority. In fact, even before you consider unique TPM features such as PCR, authorization policies, audit, and hierarchies, it's valuable simply as a hardware key store. The private signing key is protected by the hardware and a wide range of authorization options, but it can be easily backed up. This widely available and very inexpensive part offers far better protection than a software key.

A third-party certificate authority can also sign a X.509 certificate for a TPM key. For decryption keys, there is a complication due to a typical CA requirement for proof of possession. The certificate requestor must provide evidence to a CA that it possesses the private key. This is typically done by self signing the certificate signing request (CSR).[6]

For decryption keys, the TPM can't simply sign the CSR, because these keys are restricted to decryption and can't sign. The TPM has a workaround (see "Activating a Credential" in Chapter 9), but this requires a nonstandard CA.

Less obvious is that the TPM can certify data located on the device. The TPM offers several commands to support this feature.

`TPM2_Certify` asserts that an object with a `Name` is loaded on the TPM. Because the name cryptographically represents the object's public area, a relying party can be assured that the object has an associated private part. The `Name` also incorporates the key's attributes, including whether it's restricted, fixed to a parent or fixed to a TPM, and the authorization policy.

USE CASE: CERTIFYING A TPM QUOTE KEY

A signing key is used for attestation: for example, to quote (sign) a set of PCR values. The quote is far more useful if the relying party verifying the quote is assured that the signing key is restricted to the TPM, and therefore that the PCR values were actually on the TPM. The party first uses `TPM2_Certify` to get a certificate over the quote key's public area.

Naturally, the certifying key itself requires a certificate. Eventually, a useful certificate chain leads back to a root. Chapter 19 explains how TPM key certificates are provisioned and how these chains can be validated back to a trusted root key.

[5]Chapter 9 discusses the NULL hierarchy.
[6]See, for example the PKCS #10 standard in IETF RFC 2986.

USE CASE: CREATING A CERTIFICATE CHAIN

A signing key is located deep in a key hierarchy. A relying party wants to be assured that all keys in the chain back to a primary key are suitably protected, that all encryption algorithms and key sizes are of sufficient strength. The party uses TPM2_Certify to get a certificate chain that cryptographically signs the public areas of all keys in the chain.

`TPM2_Certify` signs the entire public area, including a key's policy. This leads to other use cases.

USE CASE: ASSURING THAT A KEY'S AUTHORIZATION REQUIRES A DIGITAL SIGNATURE

A relying party wants assurance that only a restricted role can use a signing key, indicated by a signature with a particular authorizing key. It uses `TPM2_Certify` to certify a key. It then validates that the policy includes a `TPM2_PolicySigned` with the public key corresponding to that role.

In this case, the policy need not have a `policyRef` parameter. The digital signature is over the challenge but not over any additional information specific to the signer.

USE CASE: ASSURING THAT A KEY'S AUTHORIZATION REQUIRES A BIOMETRIC

A relying party can validate that a signing key's policy includes a fingerprint authorization, indicated by a `TPM2_PolicySigned` with the fingerprint reader's public key and a `policyRef` parameter referring to a particular user identity.

This case is a variation of the previous case. The fingerprint reader signs not only the challenge but also a `policyRef`. The digital signature proves both possession of the private key and that the correct user's finger was supplied.[7]

`TPM2_NV_Certify` serves a similar purpose for an NV defined index. It certifies that the data at an NV index is indeed on the TPM. See Chapter 11 for details on the NV index options.

[7]Chapter 14 discusses the details of policies—in particular, the variations of the `TPM2_PolicySigned` command.

USE CASE: ASSURANCE OF NV DATA

An application is using an NV index as a counter or bit map together with a policy for a signing key. The index is used to revoke key usage: for example, when a count is reached or when a bit is set in a bit map. The application wants certainty that the NV index has been updated and uses `TPM_NV_Certify` to get a signature over the NV data.

USE CASE: QUOTE EQUIVALENT FOR AN NV EXTEND INDEX

An application is using a hybrid index as an extend index to effectively create a new PCR that is authorized, under control of the application. (Using a hybrid extend index as a PCR is explained in Chapter 11.) The explicit `quote` command only reports the standard PCR values. The application can use `TPM_NV_Certify` to sign the equivalent of a quote.

As with `TPM2_Certify`, `TPM2_NV_Certify` signs the NV index policy. The relying party can validate the NV index access policy before entrusting the NV index value in another policy.

Keys Unraveled

TPM keys have many layers of nested structures. For reference, here are several structures unrolled down to primitive types.

The following is a typical RSA key:

```
TPM2B_PUBLIC

size                    UINT16
publicArea              TPMT_PUBLIC
 type                   TPMI_ALG_PUBLIC = TPM_ALG_RSA
 nameAlg                TPMI_ALG_HASH   = TPM_ALG_SHA256
 objectAttributes       TPMA_OBJECT
 authPolicy             TPM2B_DIGEST
  size                  UINT16
  buffer                BYTE
 parameters             TPMU_PUBLIC_PARMS
  rsaDetail             TPMS_RSA_PARMS  = TPM_ALG_RSA
   symmetric            TPMT_SYM_DEF_OBJECT
                        For AES example
    Algorithm           TPMI_ALG_SYM_OBJECT
    keyBits             TPMU_SYM_KEY_BITS->TPMI_AES_KEY_BITS
```

```
    mode                TPMU_SYM_MODE->TPMI_ALG_SYM_MODE
    details             TPMU_SYM_DETAILS
   scheme               TPMT_RSA_SCHEME
    scheme              TPMI_ALG_RSA_SCHEME     = e.g., TPM_ALG_OAEP
    details             TPMU_ASYM_SCHEME        = e.g., TPMS_SCHEME_OAEP
   keyBits              TPMI_RSA_KEY_BITS       = e.g. 2048
   exponent             UINT32                  = default 2^16 + 1
 unique                 TPMU_PUBLIC_ID->TPM2B_PUBLIC_KEY_RSA
  size                  UINT16
  buffer                BYTE

TPMT_SENSITIVE

sensitiveType           TPMI_ALG_PUBLIC         = TPM_ALG_RSA
authValue               TPM2B_AUTH (TPM2B_DIGEST)
seedValue               TPM2B_DIGEST
sensitive               TPMU_SENSITIVE_COMPOSITE,TPM2B_PRIVATE_KEY_RSA
  size                  UINT16
  buffer                BYTE
```

This is a typical HMAC key:

```
TPM2B_PUBLIC

size                    UINT16
publicArea              TPMT_PUBLIC
 type                   TPMI_ALG_PUBLIC   = TPM_ALG_KEYEDHASH
 nameAlg                TPMI_ALG_HASH     = TPM_ALG_SHA256
 objectAttributes       TPMA_OBJECT -> UINT32
 authPolicy             TPM2B_DIGEST
  size                  UINT16
  buffer                BYTE
 parameters             TPMU_PUBLIC_PARMS
  keyedHashDetail       TPMS_KEYEDHASH_PARMS
   scheme               TPMT_KEYEDHASH_SCHEME
    scheme              TPM_ALG_HMAC
     details            TPMU_SCHEME_KEYEDHASH
     hmac               TPMS_SCHEME_HMAC
      hashAlg           TPMI_ALG_HASH     = TPM_ALG_SHA256
 unique                 TPMU_PUBLIC_ID
  keyedHash             TPM2B_DIGEST
   size                 UINT16
   buffer               BYTE
```

```
TPMT_SENSITIVE

 sensitiveType          TPMI_ALG_PUBLIC = TPM_ALG_KEYEDHASH
 authValue              TPM2B_AUTH
   size                 UINT16
   buffer               BYTE
 seedValue              TPM2B_DIGEST
   size                 UINT16
   buffer               BYTE
 sensitive              TPMU_SENSITIVE_COMPOSITE
  bits                  TPM2B_SENSITIVE_DATA
   size                 UINT16
  buffer                BYTE
```

And this is a typical ECC key:

```
TPM2B_PUBLIC

size                    UINT16
publicArea              TPMT_PUBLIC
 type                   TPMI_ALG_PUBLIC = TPM_ALG_ECC
 nameAlg                TPMI_ALG_HASH   = TPM_ALG_SHA256
 objectAttributes       TPMA_OBJECT
 authPolicy             TPM2B_DIGEST
  size                  UINT16
  buffer                BYTE
 parameters             TPMU_PUBLIC_PARMS
  eccDetail             TPMS_ECC_PARMS
   symmetric            TPMT_SYM_DEF_OBJECT
                For AES example
    Algorithm           TPMI_ALG_SYM_OBJECT     = TPM_ALG_AES
    keyBits             TPMU_SYM_KEY_BITS->TPMI_AES_KEY_BITS
    mode                TPMU_SYM_MODE->TPMI_ALG_SYM_MODE = TPM_ALG_CBC
    details             TPMU_SYM_DETAILS
   scheme               TPMT_ECC_SCHEME
    scheme              TPMI_ALG_ECC_SCHEME    = TPM_ALG_ECDSA
    details             TPMU_SIG_SCHEME
     ecdsa              TPMS_SCHEME_ECDSA
                        TPMS_SCHEME_SIGHASH
      hashAlg           TPMI_ALG_HASH        = TPM_ALG_SHA256
   curveID              TPMI_ECC_CURVE       = TPM_ECC_NIST_P256
  kdf                   TPMT_KDF_SCHEME
    scheme              TPMI_ALG_KDF         = TPM_ALG_NULL
    details             TPMU_KDF_SCHEME
 unique                 TPMU_PUBLIC_ID
  ecc                   TPMS_ECC_POINT
   x                    TPM2B_ECC_PARAMETER
```

```
    size            UINT16
    buffer              BYTE
   y                    TPM2B_ECC_PARAMETER
    size                UINT16
    buffer              BYTE

TPMT_SENSITIVE

sensitiveType           TPMI_ALG_PUBLIC = TPM_ALG_ECC
authValue               TPM2B_AUTH
                        TPM2B_DIGEST
 Size                   UINT16
 Buffer                 BYTE
seedValue               TPM2B_DIGEST
 size                   UINT16
 buffer                 BYTE
sensitive               TPMU_SENSITIVE_COMPOSITE
 ecc                    TPM2B_ECC_PARAMETER
  size                  UINT16
  buffer                BYTE
```

Summary

A primary use of a TPM is as a hardware security module to safely store keys. The TPM stores keys on one of four hierarchies. Each hierarchy has primary (root) parent keys and trees of child keys. A parent is an encryption key, and a parent key wraps (encrypts) child keys before they leave the TPM secure boundary.

Keys can be duplicated (wrapped with a different parent), and all children are duplicated when the parent is duplicated. Duplication is subject to restrictions. Some keys are fixed to the TPM; they can't be duplicated. Some are fixed to their parent and so can only be duplicated when the parent is duplicated.

Keys can have use restrictions as well. They can be specified as only signing or only decryption keys, and they can be restricted to only signing or decrypting certain data. Finally, keys can be certified by other TPM keys, and a relying party can validate the public key, the key's attributes, and even its policy.