CHAPTER 15

Key Management

There are many considerations when designing a key-management system with a TPM. If keys are going to be used for critical operations, such as encryption or identification, it's vital that an architecture be used to provide a standard means of managing the key's lifetime and prepare for problems if hardware breaks. Such an architecture must be able to handle key generation, key distribution, key backup, and key destruction. The design of the TPM was architected with these things in mind. This chapter describes the various options possible for these steps in a key's life.

Key Generation

When generating a key, the most important thing the user has to consider is that the key be generated randomly. If a poor random number generator is chosen, the key that is picked won't be secure. The second most important thing to consider is keeping the key material confidential. TPMs are designed to be secure against software-based threats. Hardware threats can be protected against by the manufacturer, but that isn't part of the design per se. However, the design does allow for *key split* creation of keys, where entropy used to generate a key is stored in both in and outside the TPM, so that when the TPM isn't in use, keys remain secure even with physical access.

There are three ways that keys can come to reside in a TPM. They can be generated from a seed, generated using a random number generator in the TPM, or imported. Primary keys are generated using a seed that exists in the TPM. The seed used for generating the EK is associated with the Endorsement hierarchy and isn't likely to be one that the end user can change.

The seed associated with the storage hierarchy, on the other hand, changes whenever a `TPM_Clear` command is issued. This can be done either via the BIOS, which uses the platform hierarchy authorization, or by the end user using the dictionary-attack reset password.

As stated in Chapter 10, primary keys are generated using a FIPS-approved key derivation function (KDF), which hashes together the primary seed together with a key template. The template for key generation is in two parts. The first part is a description of the kind of key to generate—whether it's a signing key or an encryption key, asymmetric or symmetric, what type of signing scheme it uses if it's a signing key, the algorithm and key size, and so on. The other part is a place where entropy can be introduced to the command to be used in generating the key. In most cases, the second part is set to all

zeros (as in the TCG Infrastructure Work Group's published EK template). However, if the user doesn't trust the entropy generator in the TPM, they can use this facility to provide a key split.

A key split is a cryptographic construct where two sets of entropy—each with as much entropy as the final key—are used to produce a key. Neither one alone is able to provide even a single bit of the final key's entropy—both are necessary. Thus, one can be held separate from the TPM, and one held inside the TPM.

In case of a primary key, one split of the key is the hierarchy's seed, inside the TPM. The other, which can be stored securely when not in use (for example, in a smart card or safe) is held outside the TPM in the template.

Primary storage keys have an associated symmetric key which is generated when the primary key is generated and is associated with it. This is also derived from the primary seed and introduces entropy. As long as the seed associated with a hierarchy isn't changed, using the same template will generate the same primary key and associated symmetric key. Because both the primary key and the symmetric key use the template in generation, if entropy is introduced there, the entropy in the template also acts as a key split for them.

Why would anyone split a key? The main reason is usually that the user is worried that it might be possible for someone to get hold of one of the two key splits. Either they're worried that the TPM's seed was squirted into the TPM at manufacturing time and someone still has a copy, or they're worried that someone will de-layer the TPM, as was done with an Infineon 1.2 chip years ago.[1] These attacks are mostly worries for the truly paranoid—the Infineon attack was successful only after destroying a handful of TPMs, and at a cost of over $200,000. But people in the security space tend to be paranoid types.

Generating a primary key can either take a relatively long time (if the key is an RSA key) or be virtually instantaneous (if it's an ECC key.) If the key takes a long time to generate (or if the secret entropy introduced in its generation isn't generally available), then the user may decide to store the key in the persistent memory of the TPM using the `TPM2_EvictControl` command, which requires the associated hierarchy's authorization. In this case, the key is given a persistent handle, and a power cycle doesn't affect the presence of the key in the TPM. It can be evicted with the same command. Depending on what attacks a user is worried about, the user may or may not decide to make their key persistent.

If a user is worried that the TPM seed has been compromised, then they're worried that primary keys may be compromised. If the primary key is compromised, all keys stored using the primary key are also compromised. In this case, the user can use a key split to introduce their own entropy into primary keys via the template, make the key persistent, and then escrow the key's template somewhere where an attacker can't get it. This prevents an attacker who knows the TPM's seed (generated at manufacturing time) from being able to determine the secrets of the primary key.

[1]William Jackson, "Engineer Shows How to Crack a 'Secure' TPM Chip," GCN, February 2, 2010, http://gcn.com/articles/2010/02/02/black-hat-chip-crack-020210.aspx.

Alternatively, a primary key can be generated and used only to create another storage key child of the primary key. The storage key is then loaded into the TPM under the primary key of the TPM. The new child storage key is then made persistent. This key behaves similarly to a TPM 1.2 SRK. It's generated by the TPM's random number generator, not from the seed. However, it exists in encrypted form for a short period of time outside the TPM after it's created, but before it's reloaded—which results in a slight risk of attack if the primary key were compromised.

If a user is worried about physical attacks against the TPM, they may wish to use entropy encoded into the key's template in a second factor and present that entropy each time the primary seed is to be generated, but *not* store the primary key in the TPM. (If the primary key is stored persistently, then a physical attack may be able to recover it.) In this case, each time the TPM is power cycled, all traces of the primary key disappear from the TPM. This is of course hard to manage, because the template must be kept secret (possibly in a USB key), separate from the TPM, and then introduced each time the key is to be loaded into the TPM.

Similarly, for the truly paranoid, who not only are worried about the TPM seed but also don't trust the TPM's random number generator, an external key can be generated by a trusted entropy source and then wrapped so that it can be imported into the TPM by a primary (or any storage) key (generated with entropy that is later discarded) and made persistent; and then the primary key is evicted. If additionally this person is worried about their system being stolen and the TPM de-layered to reveal its secrets, they should not make keys persistent in the TPM, but rather should redo this complicated loading of a key every time they power on the TPM.

USE CASE: CREATION OF DIFFERENT SRKS FOR DIFFERENT USERS

If a system has several users, they may want to have completely different sets of keys. If this is the case, they may all generate their own SRKs (individual primary restricted storage keys). This is easily possible if they each use different entropy in their template when creating their primary seed to use as their SRK. In order to make sure the same key isn't generated for each user, the templates used to generate the keys must be distinct. For example, they could use the hash of a user secret as entropy in the key's template. However, different users might pick the same user secret. It's probably better to have the TPM use its hardware random number generator to create a key under the SRK for each user.

THE RULE OF THUMB

There are only three reasons to make a key persistent. The key may be an RSA key and hence may take an unreasonably long time to re-generate, the key may be one created using a secret entropy source in the template that isn't always available, or there may not be enough (or any) persistent memory outside the TPM to store a key template. The last may be the case if the TPM is being used in a constrained environment, such as during a boot cycle. In any other case, a key should be generated as necessary. This is different from the 1.2 design, because in a 2.0 design, key loading is done with symmetric decryption and hence is very quick.

Templates

There are standard templates for creating keys, and generally it makes sense to use those rather than create your own. Templates typically use matched algorithm strengths. The one time you might *not* use matched algorithm strengths is when choosing the symmetric key. Because the symmetric key is used for loading other keys into the TPM rather than the asymmetric key, it's possible to design a system where a symmetric key with a higher strength than the asymmetric key is used for the primary key. Once this is done, no keys generated on the TPM are exposed to the weakness of the asymmetric key or algorithm.

Key Trees: Keeping Keys in a Tree with the Same Algorithm Set

Although it's technically possible to mix algorithms—make a key with one set of algorithms and then store it under a key with a different set of algorithms—it's a bad practice (and one, as you have seen, that the TSS Feature Application Programming Interface [FAPI] won't allow.) The problem is that the strength of a set of keys is dictated by the strength of the weakest key in a chain. This means not only should algorithm sets not be mixed, but chains of keys (with one key wrapping another one) should generally be kept fairly short. If any key in a chain is broken, then all keys below it are broken. So a key chain of four keys is four times weaker than a chain with one link when exposed to a brute force attack. (Of course, given a reasonable key size, a factor of 4 is unimportant.)

The reason you might decide to have a longer chain is manageability. A user may want to migrate their entire set of keys or a subset of those keys from one system to another system, or duplicate their set of keys among two or more computers. In order to make this easy, it's likely that the user will wish to rewrap only one key—at the top of a tree of keys—with the public key on a different system and then copy the encrypted blobs that represent their other keys to the appropriate location in the other system.

You might want to keep enterprise keys separate from personal keys, and different department keys separate in an enterprise, as shown in Figure 15-1. Nevertheless, it's best to keep key trees as short as possible.

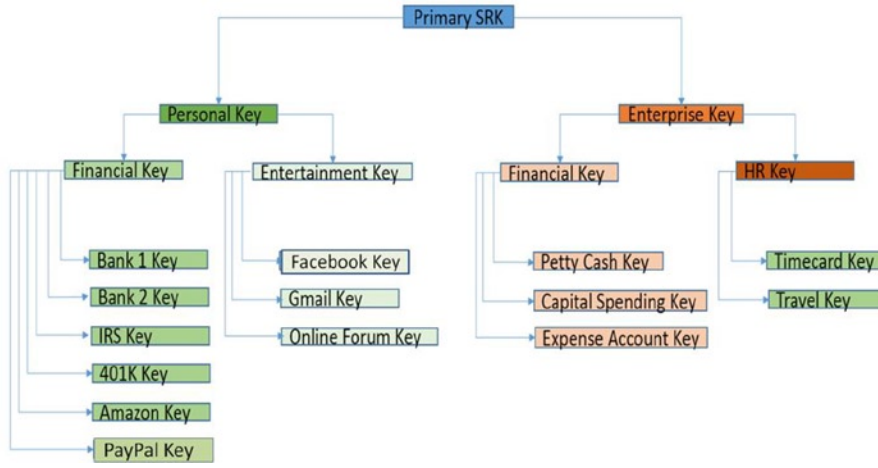

Figure 15-1. *Example key tree*

Duplication

In the key tree in Figure 15-1, the keys that might be duplicated are the Personal Key, the Enterprise Key, the Financial Keys (both Personal and Enterprise), the Entertainment Key, or the HR Key. In order to do this, all of these keys must be created to be duplicable, and they must have a policy created for them that has `TPM2_Policy_CommandCode` with `TPM2_Duplicate` selected (along with whatever restrictions are associated with duplicating a key). In most cases, a user creates two different duplication policies—one for personal keys and one for business keys—and associates those policies with a parent personal duplicable key (PDK) and a business duplicable key (BDK).

If a key isn't going to be duplicated, it can be made `fixedParent`. If a key under the SDK or UDK is going to be duplicated apart from the SDK or UDK, then it also must have a policy that allows for duplication.

With TPM 1.2, it wasn't possible to create a key that could be duplicated only to a few specified new parents and no others. With TPM 2.0, this is now possible using a command called `TPM2_DuplicationSelect`. This command allows you to specify exactly which parent (or parents) a key is targeted to be duplicated to. The main reason for using this command is in conjunction with `PolicyAuthorize`. By using `PolicyAuthorize`, an IT organization can change the target backup key for duplication. So if the organization normally backs up keys to a specified server, and that server dies, then by signing a `TPM2_DuplicationSelect` command that selects a new server, the organization can mail out a new signed policy to employees, knowing that they now are allowed to duplicate their keys to the new server. This allows the new duplication target without allowing employees to back up their keys to their home computers (which may not be trusted).

Because TPM2_Duplicate and TPM2_DuplicateSelect can't be authorized with a password or an HMAC session, in order to duplicate a key, you must first start a policy session and then satisfy the branch of the policy that has the TPM2_PolicyCommandCode linked with TPM2_Duplicate or TPM2_DuplicationSelect. Then you can execute the appropriate command to duplicate the key to a new parent.

USE CASE: A SET OF SERVERS ACTS AS ONE

In this use case, a set of SSL servers acts as a failover for one another or for load balancing. The company doesn't want users to need to know which server they're connected to—it isn't something users care about. So the company needs the same key to exist on all the servers that are being used to service its web page. (This also means the company has to get only one certificate for this key, instead of a separate key for each server.)

The company creates a duplicable key with a PolicyAuthorize command as the policy and then uses the private key associated with the PolicyAuthorize command to sign several TPM2_DuplicationSelect commands, each of which points to a different server. A user gets this key certified and puts a copy of the certificate on each server. The user then duplicates the key from the original server to all the other servers, and finally imports the key using TPM2_Import into each of the other servers. At this point, all the servers look identical to an outside user.

Steps

1. Create policy P using TPM2_PolicyAuthorize, an enterprise public signing key, and a policyRef of SSL.
2. Create a duplicable key using policy P on the initial SSL server.
3. Create restricted storage keys on all the other SSL servers using TPM2_Create, and call them SRK_i.
4. Use the enterprise private key to sign TPM2_PolicyDuplicateSelect, selecting the $SRK_{i\ public}$ as the target of duplication. Do this once for each SRK_i.
5. Use TPM2_VerifySignature on the initial SSL server to create a ticket for each signed policy. This allows that signed policy to be used by the initial SSL server for duplication.

6. For each policy, create a duplicated key that can be loaded by SRK_i by doing the following on the initial SSL server:
    a. Load the enterprise public signing key using `TPM2_Load`.
    b. Load the SRK_i public key into the initial SSL server using `TPM2_LoadExternal`.
    c. Use `TPM2_StartAuthSession` to start a policy session.
    d. Execute `TPM2_PolicyDuplicationSelect`, selecting the SRK_i of one of the target servers.
    e. Execute `TPM2_PolicyAuthorize`, using the policy, a `policyRef` of SSL, and the `Ticket` corresponding to SRK_i.
    f. Execute `TPM2_Duplicate`, passing it the handle of the loaded SRK_i public and the handle of the enterprise signing key.
    g. The result of the `TPM2_Duplicate` command is an encrypted version of the enterprise signing key. Send it to the server with SRKi.
    h. Import the duplication blob into the server with SRKi.
7. Copy the certificate of the enterprise signing key to that server.

At this point, the key is the same on all the servers and can be used for SSL identification and communication.

Key Distribution

In some cases, keys need to be distributed long after a system is initially set up. Being able to distribute keys securely is very important in these cases. The TPM design makes this easy. When each system is set up, a non-duplicable storage key is generated on the system, and a central system keeps a record associating this key with the system name (or perhaps the system serial number). This can be done in an Active Directory or LDAP database. Additionally, at provisioning time, the local platform gets a public key of the central system that corresponds to a signing key. At some later point, if the central system wants to distribute an HMAC key to the system, the following takes place:

1. The central IT system creates an HMAC key using `TPM2_GetRandom`.
2. The central IT system encrypts the HMAC key with the public portion of the target client's storage key.

3. The central IT system signs the encrypted HMAC key with its private signing key. This is done so the local platform knows that what is being sent is authorized by IT.
4. The encrypted HMAC key is sent to the client along with a signature that proves it came from the central IT system.
5. The client verifies the signature on the encrypted key by loading the central server's public key. (This can be done with the TPM using `TPM2_Load` and then using `TPM2_VerifySignature`, if you like.)
6. The client imports the verified, encrypted HMAC key into its system using `TPM2_Import`, getting out a loadable, encrypted blob containing the HMAC key.
7. The client loads the HMAC key when the user wishes to use it, using `TPM2_Load`, and uses it as normal. At this point, the local platform has received an HMAC key from the IT central system that has never been decrypted in the local system's memory.

Key Activation

Because of the ability to create and re-create keys from the seed in the TPM, it's possible to use multiple key templates at provisioning time of a system and have a central IT system record the key template and corresponding public portion of the keys associated with the system. Central IT can then power cycle the TPM, destroying the system's copy of the key. Thus when the system is distributed to an end user, it doesn't have any of these keys available.

Later, when IT wants to activate those keys, it need only send the key template used to create the key to the end user and allow the system to re-generate the key from the template using `TPM2_CreatePrimary`. Note that the key template includes the policy of the key so generated, but not the password associated with it, which is chosen whenever the key is re-generated. If the central system wishes to avoid the use of that password when controlling the key, two bits in the template can be selected: `userWithAuth` and `adminWithPolicy`. These can be set in such a way as to make the password unable to control the key. If `userWithAuth` is set `FALSE`, and `adminWithPolicy` is set `TRUE`, then the password can't cause the key to perform any functions.[2]

In using this technique, the templates should be chosen in such a way as to include random entropy. Without the template, the key can't be re-created, so the central system can be sure the key isn't used until the template is received by the client.

[2]Because these flags are part of the template, if a user tries to change them, the user gets a different key.

There is another way to do key activation, similar to what was possible with TPM 1.2: using migratable keys. When a key is duplicated, you can doubly encrypt it: once using the parent key of the system to which it's being duplicated, and once using a symmetric key that is inserted when the duplication is done. The produces a key blob that is encrypted twice. The outside encryption is gated by the new parent's private key. The inner encryption is done with a symmetric key. In this case, when a `TPM2_Import` command is executed, the TPM must have the private asymmetric key already loaded; its handle is given to `TPM2_Import`, and a secret is passed into the `TPM2_Import` command as a parameter. The secret is used in calculating the symmetric key, which in turn is used to decrypt the inner encryption. The command flow is as follows:

1. A duplicable key is generated on a central system.
2. The key is duplicated to the client system using the symmetric key option. This parameter is called `encryptionKeyIn` in the `TPM2_Duplicate` function.
3. The key blob is signed by the central system and sent to the client, but the `encryptionKeyIn` parameter is kept safe by the IT administrator.
4. When the IT administrator wishes to allow the key be used, `encryptionKeyIn` is provided to the client system, allowing the client system to import it using the `TPM2_Import` command.

Key Destruction

Once a key has been created, it's sometimes important to be able to destroy it as well. One example is if a user is going to sell, surplus, or recycle a computer and wants to make sure data that was encrypted on that system with that key is no longer available. TPMs provide this facility in a number of easy ways.

If the key used is a primary key, the easiest way to destroy it is to ask the TPM to change its copy of the seed of the hierarchy on which it was created (usually the storage hierarchy). `TPM2_Clear` does this for the storage hierarchy. Clearing the TPM destroys all non-duplicable keys that are associated with the hierarchy, evicts all keys in the hierarchy from the TPM, and changes the seed, preventing any primary keys previously associated with that hierarchy from being re-generated. It also flushes the endorsement hierarchy, but it doesn't change that seed.[3] Duplicable keys can no longer be loaded into the system, although if they have been duplicated to a different system, they may not be destroyed.

[3]This way, the EK and TPM vendor certificates are still valid.

If such a drastic step isn't necessary (perhaps the machine is only going to be loaned for a time to a different employee, or multiple employees are using the machine), other things can be done, if preparations are made ahead of time. For example, if a primary key is generated with secret entropy in the template and then made into a persistent key, then the only thing that needs to be done to destroy the key is to destroy the copies of the template and evict the primary key from the persistent storage. Once this is done, the key is gone and can't be re-generated. Because there can be multiple trees underneath different primary keys, this provides a way to destroy a particular tree of keys without destroying all the trees of keys in a TPM. This may be important if multiple users are using the same TPM.

It's even possible to destroy keys that are generated outside the TPM, imported into the TPM, and then made persistent. If the copies outside the TPM are destroyed (which may be possible if the import was done in a controlled facility), then merely evicting the key from persistent memory also destroys the key.

Putting It All Together

This section provides two examples of how different types of businesses might decide to manage TPM entities. We start with a simple case, which might apply to a small business, and then consider a large enterprise.

Example 1: Simple Key Management

An end user is handling all of their own keys. The user has two systems: a primary system and a backup system for backing up keys. Here are the steps the user follows to manage the keys:

1. Create an SRK on each system using a standard non-duplicable key template. Set `userWithAuth` to `TRUE`, `adminWithPolicy` to `FALSE`, and the policy to a NULL policy. This means the policy is disabled and the password can be used to authorize use of the SRK. The user sets the password to a well-known password when using the `TPM2_CreatePrimary` command to create the SRKs.

2. Create a duplicable storage key (DSK) under the SRK on the primary system. Use `TPM2_Create` to create this key. It has `userWithAuth` set to `TRUE` and `adminWithPolicy` set to `TRUE`. This allows the password to authorize using the key and the policy for duplicating the key. (Remember that keys can only be duplicated using a policy.) It has a policy that specifically has a branch with `TPM2_PolicyCommandCode` with `TPM2_Duplicate` selected, together with `TPM2_PolicyAuthValue`. This policy requires the user to prove knowledge of the key's password in order to duplicate it.

3. Load the public key of the new SRK to which the key is to be duplicated.

4. Duplicate this storage key to the backup system by creating a policy session, executing `TPM2_PolicyCommandCode` with `TPM2_PolicyDuplicate`, and then executing `TPM2_PolicyAuthValue`. Then an HMAC session is started (using the DSK password). The two sessions are referenced when executing the `TPM2_Duplicate` command, passing it the handle of the DSK and the public key of the SRK of the backup system. This produces a blob that contains the duplicated key and is encrypted in a way that can be imported into the TPM, which knows the SRK private portion.

5. Move the blob to the backup system, and use `TPM2_Import` to import the key into the backup system. This produces another blob, which can be loaded into the backup system on demand.

6. As new keys are created under the DSK on the primary system, send copies of those key blobs to the backup system, where they can also be loaded using the copy of the DSK, and used.

7. To decommission the primary system, use `TPM2_Clear`, using the lockout password to clear the TPM's storage hierarchy.

8. To migrate all keys to a new system, create an SRK on the new primary system.

9. Repeat the process of duplication from step 4. This time, the new parent is the SRK of the new primary system.

10. Copy all other keys blobs onto the new primary system.

Example 2: An Enterprise IT Organization with Windows TPM 2.0 Enabled Systems

In this case, the enterprise doesn't want to use EKs that are potentially known outside the organization. The enterprise wants to use its own EKs after the machines are provisioned, but use the OEM EK to prove to itself that the system is genuine. The organization provisions each system as it comes in, as follows:

1. Generate the OEM EK using `TPM2_CreatePrimary` and the TCG Infrastructure Workgroup's standard EK template. Compare it to the vendor EK certificate that came with the system. Check the certificate as well, using the vendor's public key.

2. Run `TPM2_Clear` to wipe the TPM's storage and EK hierarchies.

3. In a trusted location, evict the OEM EK, ask the TPM for a random number, repopulate the EK template with this entropy, and read out the EK public portion, making the enterprise's own certificate for this key.

4. Change the storage hierarchy, endorsement hierarchy, and dictionary-attack authorization values to random values, storing them in an LDAP server.

5. Create a restricted encryption key and make it persistent, for use as an SRK. This key has an authorization value of `NULL` and a policy of `NULL`. This allows anyone to use it who wishes to.

6. Create a restricted signing key and make it persistent, for use as an AIK. This key also has an authorization value of `NULL` and a policy of `NULL`, so that TNC software can use it.

7. Store a copy of the SRK's public key in the enterprise's LDAP associated with this machine.

8. Create a certificate of the AIK that associates it with this machine. This certificate is stored both on the LDAP and on the system itself.

9. Uses the AIK to quote the current PCR values, and check them against golden measurements that came with the system.

10. Change the software load and configuration of the system to match the enterprise's own policies.

11. Use the AIK to quote the current PCR values, and store them in the LDAP associated with this system. These are used as a new set of golden measurements.

12. Create virtual smartcards on the machine using the TPM, and set up the Windows VPN server to accept the certificate of this key.

13. Set up another virtual smartcard on the machine using the TPM, and set up a Radius server to accept it for connections to the enterprise's wireless network, using WPA2 in Enterprise mode.

14. Create a 32-byte NV index, and store a hash of the enterprise's IT organization public key there. The policy of the key only allows this same key to be used to write to the index. This key will be used later to check software updates before they're installed, to see if they're approved by IT.

15. Install Wave software to report the PCR measurements on each boot, sending an alert to the IT organization if they aren't correct. (Alternatively, a StrongSwan VPN can be used, which doesn't grant access to the network unless the PCRs pass muster.)

16. Create a policy for allowing duplication of a key to the IT backup server's TPM, and store it on the system. This policy is signed with the IT private key.

17. The user's boss provides a policy that allows the boss to use the keys in their employee's absence.

18. When the user gets the system, the user creates a duplicable restricted decryption (storage) key, under which the user stores all their enterprise keys. The policy the user gives it is the OR of the policies in steps 16 and step 17. Before doing this, the user checks the policy's signature using the hash of the public key stored in step 14.

19. The user duplicates their duplicable storage key to the IT organization's backup server and then its wrapped keys in their normal backup.

20. If the user quits or the machine is to be recycled, use the stored owner authorization to send the system a notice that it should execute `TPM2_Clear`, thus wiping all the keys stored on the system. The OEM EK is used to restart the process.

21. If the user is moved to a new system or their motherboard dies, re-duplicate their backed-up key (stored in step 18) to their new system, and copy their other key blobs from backup to the new system. The user can then continue working.

Summary

You've seen that the facilities of the TPM allow for very sophisticated or very simple key management, depending on the needs of the end user. These needs can range from those of a paranoid enterprise worried about industrial espionage, including theft of machines, to those of a non-paranoid home user, who merely wishes to keep their keys safe on their home system. By crafting the key hierarchies and setting up authorizations and policies correctly, you can keep keys safe and usable.