CHAPTER 18

Context Management

In general, we don't prevent things unless there is a good reason for that. Put another way, we try to allow anything that doesn't cause a security problem.

David Wooten,
During an e-mail exchange about context management

TPMs, for all their tremendous capability, are very limited in their memory, largely to reduce cost. This means objects, sessions, and sequences must be swapped in and out of the TPM as needed, much like a virtual memory manager swaps memory pages to and from disk drives. In both cases, the calling application thinks it has access to many more objects and sessions (in the TPM case) or much more memory (in the virtual memory case) than can actually be present at any given time.

For a system where only a single application sends commands to the TPM, these swapping operations can be performed by the application itself.[1] However, when multiple applications and/or processes are accessing the TPM, two components of the TSS stack are required: the TPM Access Broker (TAB) and the resource manager (RM).

This chapter describes the high-level architecture of the TAB and RM. Then it describes the features and commands of the TPM that support swapping objects, sessions, and sequences in and out of TPM memory, the details of how different swappable entities are handled by the related commands, and some special cases.

TAB and the Resource Manager: A High-Level Description

The TAB and RM were first described in Chapter 7 as layers in the TSS. This section provides some insights into the internals of the TAB and RM.

The TAB and RM transparently isolate TPM applications and processes from the messiness of arbitrating multiprocess access to the TPM and swapping objects, sessions, and sequences in and out of TPM memory as needed. The TAB and RM are closely

[1]Even in this case, the use of an RM is advantageous because it relieves the application of the burden of performing the swapping.

related and typically integrated into the same software module. Depending on system design, they may reside in the top layer of a TPM device-driver stack, or they may be integrated as a daemon process sandwiched between the TSS system API layer above and the TPM device driver below.

TAB

The TAB's responsibility is fairly simple: arbitrate multiple processes accessing the TPM. At a minimum, all processes must be guaranteed that, between the time their command byte stream begins to be transmitted to the TPM and the time the response byte stream is fully received from the TPM, no other processes communicate with the TPM. Some examples of multiprocess collisions that could occur in the absence of a TAB are as follows:

- Process A's command byte stream begins to be transmitted to the TPM, and before it's done, process B's command byte stream starts being transmitted. The byte stream sent to the TPM will be a mix of command bytes from both processes, and the TPM will probably return an error code.
- Process A sends a command, and Process B reads the response.
- Process A's command byte stream is transmitted, and then, while its response is being read, Process B's byte stream starts being transmitted.
- Process A creates and loads a key exclusively for its own use. Process B context saves (using the `TPM2_ContextSave` command) the key and then, sometime later, context loads it (using the TPM2_ContextLoad command) and uses it for its own purposes.

A TAB can be implemented in a couple of different ways: either with a TPM lock or without one.

In the lock architecture, a software method of sending a "lock" down to the TAB could be designed. This would signal the TAB that no other process would be allowed access to the TPM until the process that locked it is completes. This would have the advantage of allowing a TAB to complete multiple commands to the TPM without interruption. And interestingly enough, this architecture would eliminate the need for an RM, assuming you could make it work. An application could claim a lock on the TPM and send commands to the TPM while managing all TPM contexts itself (this management includes cleaning up after itself by evicting all objects, sequences, and sessions before releasing the lock). The Achilles heel of this architecture is that the application might fail to release the lock, starving all other applications. There could be some sort of timeout mechanism, but that would place artificial limits on the time that an application could exclusively use the TPM. This in turn would force applications to do some fairly complex management of corner cases related to TPM contexts since they couldn't depend on the lock being active long enough to complete their TPM operations. This approach was initially considered by the TCG TSS working group and rejected after some very wise advice from Paul England of Microsoft.

The simpler lock-less architecture allows the transmission of a TPM command and reception of its response atomically without interruption from any other processes' interactions with the TPM. This reduces the time that the process exclusively accesses the TPM to the time it takes to send a command and receive a response. This architecture requires an underlying RM because competing applications cannot be in the business of managing each other's objects, sessions, and sequences. For example:

1. Process A happens to be the only application accessing the TPM for a while, and during this time it starts three sessions and loads three keys.
2. Then process B decides it's ready to talk to the TPM and wants to create a session and a key. If the TPM has only three session slots and three object slots, process B must first unload at least one of process A's sessions and one of its objects.
3. This forces process B to manage process A's TPM contexts—an untenable position in terms of inter-process isolation (security) and software complexity. Without a central RM, applications must manage all TPM contexts themselves. It's almost impossible to make this work, and it guarantees that processes can mess with each other's TPM contexts.

Hence the need for an RM.

Resource Manager

The RM is responsible for transparently handling all the details of swapping objects, sessions, and sequences in and out of the TPM. Very highly embedded, single-user applications may choose to handle these tasks themselves, but, as discussed previously, most systems are multiuser and require an RM. As an analogy, imagine if all PC applications had to manage memory swapping in and out of memory and the hard disk themselves instead of relying on the operating system to do this. The RM performs a similar function for processes that access TPMs.

Resource Manager Operations

At the risk of stating the obvious, if a transient entity is used in a command, it must be loaded into TPM memory. This means that an RM must parse the command byte stream before the command is sent to the TPM and take any actions required to ensure that all transient objects used by that command are loaded into the TPM. This includes all sessions referenced in the authorization area and all objects, sessions, and sequences whose handles are in the command's handle area.

The basic operations that an RM must perform are as follows[2]:

- Virtualize all object and sequence handles sent to and received from the TPM. Because more of these can be active than can be in the TPM's internal memory, they must be virtualized.[3]
- Maintain tables to keep track of contexts for objects and sequences.
- Maintain a virtual-to-TPM handle mapping for objects and sequences that are loaded in the TPM.
- For commands being sent to the TPM:
    - Capture all command byte streams before they're sent to the TPM.
    - Check for any handles in the authorization area or handle area. If these handles represent objects, sequences, or sessions, ensure that the saved contexts for these are loaded into the TPM so that the command can successfully execute. For objects and sequences, replace the virtual handles in the command byte stream with the real handles returned from the load commands before sending the command to the TPM.

■ **Note** Session handles do not need to be virtualized, because they are constant for the lifetime of the session: that is, when a session is reloaded, it keeps the same handle. The "why" of this is discussed later. For now, it's sufficient to understand the difference between objects and sequences, which get new handles every time they're context loaded, and sessions, which do not.

- For responses from the TPM:
    - Capture these responses before they are returned to higher layers of software.
    - Virtualize any object or sequence handles in the responses, and replace the handles returned by the TPM in the response byte stream with these virtualized handles.
- It must either proactively guarantee that commands will never fail due to being out of memory in the TPM or reactively fix the problem when the required contexts can't be loaded into the TPM.

[2]For a reference implementation of a resource manager see the Test/tpmclient/ResourceMgr directory in the SAPI library and test code. The location for this was described in Chapter 7.
[3]TPM handle virtualization is not to be confused with OS level virtualization of hardware or operating systems.

- There are two possible ways to implement the proactive approach:
    - The simplest proactive approach requires that, after completion of each TPM command from software layers above the RM, all objects, sessions, and sequences must be context saved and removed from TPM internal memory. This is inherently a simpler approach, but it does cause more commands to be sent to the TPM. For example, after a `load` command for an object, the object is unloaded even though it might be used in the next command.
    - The second proactive approach is to examine the command's handle and session area before executing it, to find all the objects, sequences, and sessions that must be loaded into the TPM. Next, enough of the objects, sequences, and sessions currently loaded into the TPM are evicted that the required ones can be context loaded. Then the required ones that aren't already loaded into the TPM are loaded.

■ **Note** A hardware-triggered command, `_TPM_Hash_Start`, is discussed later in this chapter. This command implicitly, and transparently to the RM, evicts an object or session. This imposes some special requirements on the second type of proactive approach.

- The reactive approach takes actions after a command fails when the response code indicates that the TPM was out of memory. When this type of error code is received, the RM must remove objects, sessions, or sequences until enough memory is freed to load the objects, sessions, and sequences required for the current command. Then the original command must be replayed.

■ **Note** From hard-earned experience, the reactive approach is extremely difficult to code and even harder to debug. I highly recommend one of the proactive approaches. I tried the reactive approach and it didn't end well, resulting in a recoding effort. Enough said.

- The RM must properly handle object, sequence, and session contexts across reset events, as described in the previous section.

The above covers the basic requirements. Others for handling corner cases and some more esoteric functionality are detailed in the "TSS TAB and Resource Manager Specification".

Now let's examine the TPM features that support RMs.

Management of Objects, Sessions, and Sequences

Because the TPM has limited internal memory, objects, sessions, and sequences need to be dynamically swapped in and out of memory. As an example, the reference implementation of a TPM 2.0 implemented by the Microsoft simulator only allows room for three object slots. An object slot is internal TPM memory for an object or sequence. There are also three session slots. Hence the need for virtualization of transient entities (here, the term *transient entities* describes transient objects, sessions, and sequences). This section describes the TPM capabilities and commands that are used to accomplish this virtualization.

TPM Context-Management Features

The TPM capabilities used to manage transient entities are capability properties that can be queried, special error codes, three TPM commands, and some special handling for `TPM2_Startup` and `TPM2_Shutdown`. These capabilities are used by the caller or, preferably, a dedicated RM to virtualize and manage transient entities.

TPM Internal Slots

TPM internal memory for transient entities consists of slots. A maximum number of slots are available for loaded objects and sequences (`MAX_LOADED_OBJECTS`), and a similar maximum for loaded sessions (`MAX_LOADED_SESSIONS`). Both of these maximums can be queried using the `TPM2_GetCapability` command. The RM can query these maximums and use them to manage the loaded contexts. Or it can rely on error codes returned by the TPM. Because of a special case related to `_TPM_Hash_Start`, some RMs require a combination of these.

Special Error Codes

Special error codes are used to tell the RM that the TPM is out of memory, meaning no slots are available, and the RM must do something: `TPM_RC_OBJECT_MEMORY` (out of memory for objects and sequences), `TPM_RC_SESSION_MEMORY` (out of session memory), or `TPM_RC_MEMORY` (out of memory in general).

These error codes are returned by commands that need to use object, sequence, or session slots in the TPM. For example the `TPM2_Load` command tries to load an object, and if there is no memory, `TPM_RC_OBJECT_MEMORY` or `TPM_RC_MEMORY` may be returned. The commands that explicitly use object or sequence slots are `TPM2_CreatePrimary`, `TPM2_Load`, `TPM2_LoadExternal`, `TPM2_HashSequenceStart`, `TPM2_HMAC_Start`, and `TPM2_ContextLoad` (when the context being loaded is an object or sequence context).

Additionally, three commands implicitly use an object slot. `TPM2_Import` uses one slot for scratchpad memory; the slot is freed after the command completes. Likewise, any command that operates on a persistent handle uses one slot for scratchpad operations and frees the slot after completion. Both types of commands return one of the above error codes if no slots are available. In response, the RM must evict a transient entity and retries the command.

The third command that implicitly uses an object slot is kind of strange: `_TPM_Hash_Start`. This command is typically triggered by hardware events, and it doesn't return an error code if no slots are available. Instead, it kicks an object out and provides no indication of which object was evicted. This means the RM and/or calling applications had better make sure one of the following is true during any time period when this command could be triggered by hardware:[4]

- One slot is available. The RM can use the `TPM2_GetCapability` command and query the `MAX_LOADED_OBJECTS` property. Based on the response, the RM can offload an object to free a slot. [5]
- The contexts for all objects or sequences that currently occupy the TPM's slots are saved. Otherwise, an object will be evicted with no ability to be reloaded.
- The contexts for all objects or sequences that currently occupy the TPM's slots are unneeded. In this case, it doesn't matter if one is evicted with no chance of being reloaded.[6]

The commands that explicitly use session slots are `TPM2_StartAuthSession`, `TPM2_ContextLoad` (when the context being loaded is a session context), and all commands that use sessions (except for those that use sessions for password authorization).

TPM Context-Management Commands

The TPM commands that enable transient entity management are `TPM2_ContextSave`, `TPM2_ContextLoad`, and `TPM2_FlushContext`. These commands have different effects depending on the type of transient entity being operated on.

`TPM2_ContextSave` saves the context of a transient entity and returns the entity's context. The returned context is encrypted and integrity protected. This is done in a manner that only allows the context to be loaded on the exact same TPM that saved it; a saved context can't be loaded into a different TPM. It is important to note that saving the context saves it into system memory, which may be volatile. Some other mechanism is required if the saved context needs to preserved across events that might erase memory contents, such as hibernation or sleep. For the PC, in the case of hibernation, the system saves all memory contents to some form of nonvolatile storage, such as a disk drive; for a sleep event, the memory remains powered on, which preserves the memory contents.

[4]This argues in favor of the simplest proactive approach to RM design, described earlier in this chapter. In that case, for the most part, none of these mitigations is required. There is still a small vulnerability: the time window from the time the RM completes the command to the time it is done evicting objects and sessions. If the `_TPM_Hash_Start` is triggered in this time window, an object or sequence context will be lost.
[5]The best way to do this would be for the process that's going to trigger the `_TPM_Hash_Start` command to send a request to the RM to free up an object slot.
[6]Since the RM really has no way to do this, the OS would have to have some way of ensuring this.

After the context is saved, if the entity is an object or a sequence, the entity still resides in the TPM and has the same handle. The saved context is a new copy of the object or sequence context.

A session, however, is handled differently. When a session's context is saved by the `TPM2_ContextSave` command, it is evicted from TPM memory. A session's context handling is unique: the context can either be evicted and in system memory, or it can be loaded on the TPM, but not both. Regardless of where the session's context resides, it always has the same handle and it's always "active" until its context is flushed by the `TPM2_FlushContext` command or a command is sent using the session with the `continueSession` flag cleared.

The reason for this special handling of sessions' handles is to prevent multiple copies of sessions and, hence, session replay attacks. A small piece of session context is retained inside the TPM after the context is saved.

For objects, sequences, and sessions, `TPM2_FlushContext` removes all of the transient entity's context from the TPM. In the case of an object or a sequence, the object—which still resides in the TPM after the `TPM2_ContextSave` command—is completely removed, but the saved context can still be reloaded. In the case of a session, the remaining session context is removed, which means the session is no longer active, the session context cannot be reloaded, and the session handle is freed for use by another session.

`TPM2_ContextLoad` is used to reload a saved context into the TPM.[7]

For an object or sequence, the context is loaded and a new handle is created for the object. An object or sequence context can be reloaded multiple times, returning a new handle each time. This means multiple copies of an object or sequence can reside in the TPM at any given time.[8]

For a session, the `TPM2_ContextLoad` command reloads the session and returns the same session handle. Also, a session's context can only be reloaded once after its context was saved in order to prevent session replay attacks.

Special Rules Related to Power and Shutdown Events

TPM Restart, TPM Reset, and TPM Resume are described in detail in Chapter 19. There are some special context-handling rules related to these events. This section describes the high-level "why" of these rules and then the details of the rules themselves.

A TPM Reset is like a cold power reboot, so session, object, and sequence contexts saved before a TPM Reset can't be reloaded afterward. Because `TPM2_Shutdown(TPM_SU_CLEAR)` was performed or no `TPM2_Shutdown` at all was executed, none of the information required to reload saved contexts was saved.

---

[7]It should be noted that although the names are similar, the `TPM2_Load` and `TPM2_ContextLoad` commands are quite different. `TPM2_Load` performs the initial load of an object after creation by `TPM2_Create`. This load command can't be used for sessions, nor can it be used to load an object's or sequence's saved context. `TPM2_ContextLoad` loads a transient entity after its context has been saved by the `TPM2_ContextSave` command. The similar names and overlap in functionality (in the case of objects) has tripped up many an unwary developer, including me at times.

[8]Having multiple copies of an object or a sequence loaded in the TPM serves no useful purpose and uses up more of the limited slots available in the TPM. The quote by David Wooten at the beginning of this chapter resulted from a discussion about this. Even though it serves no useful purpose, it doesn't pose a security risk, so the TPM allows it in the interest of internal TPM firmware simplicity.

A TPM Restart is used to boot after the system hibernated, and a TPM Resume is used to turn on your computer after a sleep state has been entered. For both of these cases, because `TPM2_Shutdown(TPM_SU_STATE)` was executed, saved session, object, and sequence contexts can be reloaded; the one exception is that objects with the `stClear` bit set cannot be reloaded after a TPM Restart.

The detailed rules are as follows:

- Any type of TPM reset removes transient entities from the TPM. If the transient entity's context wasn't saved, there is no way to reload the entity.
- As for the case of the context being previously saved, if:
    - *TPM Resume occurs:* Saved contexts can be context loaded.
    - *TPM Restart occurs and the object has the* `stClear` *bit cleared:* The object's saved context can be context loaded.
    - *TPM Reset or a TPM Restart occurs with the object's* `stClear` *bit set:* The saved object's context can't be context loaded.
- For a session, if the session's context was saved:
    - The context can be context loaded after a TPM Resume or TPM Restart.
    - The context can't be context loaded after a TPM Reset.

State Diagrams

Because of all these complicated rules, some diagrams may help to illustrate both the normal handling and the special rules related to TPM Reset, TPM Restart, and TPM Resume (see Figure 18-1).

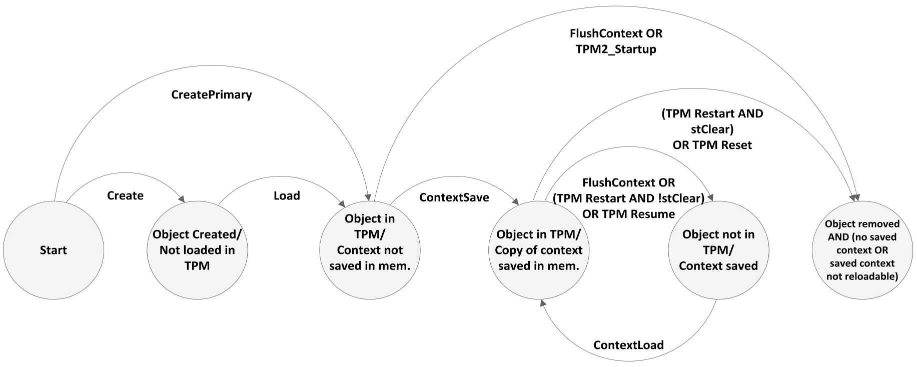

Figure 18-1. *TPM state diagram for objects and sequences*

Some notes about this diagram:

- Even though the word *objects* is used, this refers to both objects and sequences.
- The `Load` and `ContextLoad` arcs can be performed multiple times. Each instance results in a new copy of the object in the TPM with a new handle. Other than the handle, this object is identical to the other copies. Having multiple copies loaded in the TPM serves no useful purpose, as noted earlier.[8]
- The `ContextSave` arc can occur multiple times. Each instance results in a new copy of the object's context.
- For sequences, the diagram is the same except for the following: a sequence's context must be saved after each `SequenceUpdate`. Otherwise a `ContextSave` followed by a `ContextLoad` would result in a bad hash or HMAC computation.

The state diagram for sessions is relatively simple compared to objects and sequences (see Figure 18-2). The important differences to note are as follows:

- Objects and sequences can exist both on and off the TPM simultaneously, whereas sessions can't.
- Objects can be flushed and then reloaded. Sessions, when flushed, are terminated, and their saved contexts can't be reloaded.
- Unlike objects and sequences, active sessions always keep the same handle.
- Sessions can be "active" whether loaded in the TPM or not. They only become inactive when they are terminated (Session Ended state).

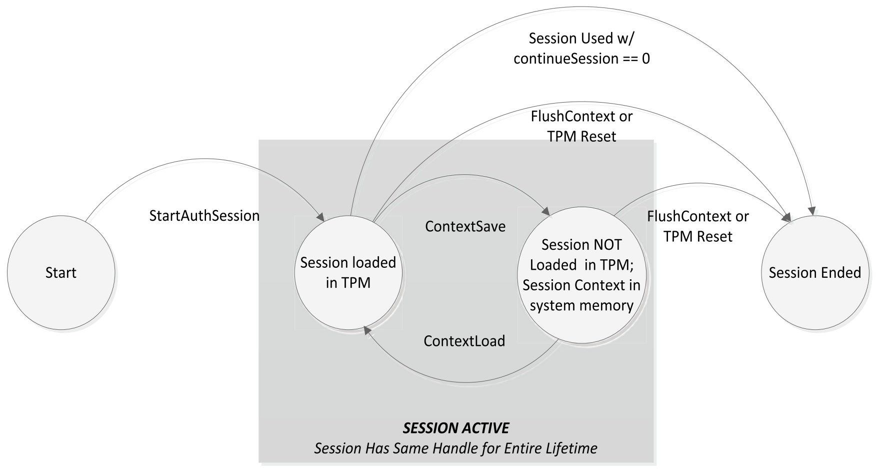

Figure 18-2. *TPM state diagram for sessions*

Summary

This concludes the discussion of context management. The TPM provides all the functionality needed to implement a resource manager. Although there are probably many ways to design a resource manager, at a high level, the simplest proactive approach is recommended.