CHAPTER 7

TPM Software Stack

This book is primarily about TPM 2.0 devices. However, a TPM without software is like a car with a full tank of gas but no driver; it has great potential but isn't going anywhere. This chapter, in preparation for the rest of the book, introduces you to the TPM's "driver"[1], the TPM Software Stack (TSS). A good understanding of this topic will enable you to understand subsequent code examples in this book.

The TSS is a TCG software standard that allows applications to intercept the stack, that is, be written to APIs in the stack at various levels in a portable manner. Applications written to the TSS should work on any system that implements a compliant TSS. This chapter describes the layers of the TSS with a particular focus on the System API and Feature API layers. The other layers are described at a high level.

The Stack: a High-Level View

The TSS consists of the following layers from the highest level of abstraction to the lowest: Feature API (FAPI), Enhanced System API (ESAPI), System API (SAPI), TPM Command Transmission Interface (TCTI), TPM Access Broker (TAB), Resource Manager (RM), and Device Driver.[2]

Most user applications should be written to the FAPI, because it's designed to capture 80% of the common use cases. Writing to this layer is the TPM equivalent of writing in Java, C#, or some other higher-level language.

The next layer down is the ESAPI, which requires a lot of TPM knowledge but provides some session management and support for cryptographic capabilities. This is like writing in C++. At the time of this writing, the ESAPI specification is still a work in progress, so it isn't described in this chapter.

Applications can also be written to the SAPI layer, but this requires much more TPM 2.0 expertise. This is analogous to programming in C instead of a higher-level language. It provides you with access to all the functionality of the TPM but requires a high level of expertise to use.

[1]This is not to be confused with an OS device driver.
[2]The device driver isn't officially part of the TCG-defined TSS, but it makes sense to discuss it in this chapter because it's one of the layers in a TPM software stack.

TCTI is the layer used to transmit TPM commands and receive responses. Applications can be written to send binary streams of command data to the TCTI and receive binary data responses from it. This is like programming in assembly.

The TAB controls multiprocess synchronization to the TPM. Basically it allows multiple processes to access the TPM without stomping on each other.

The TPM has very limited on-board storage, so the Resource Manager is used in a manner similar to a PC's virtual memory manager to swap TPM objects and sessions in and out of TPM memory. Both the TAB and the RM are optional components. In highly embedded environments that don't have multiprocessing, these components are neither needed nor, in some cases, desired.

The last component, the device driver, handles the physical transmission of data to and from the TPM. Writing applications to this interface is possible as well and would be like programming in binary.

Figure 7-1 illustrates the TSS software stack. Some points to note:

- Although typically there is only one TPM available to applications, multiple TPMs could be available. Some of these could be software TPMs, such as the Microsoft simulator; others may be accessed remotely over the network—for instance, in the case of remote administration.
- Generally, components from the SAPI on up the stack are per-process components.
- Components below the SAPI are typically per-TPM components.
- Although Figure 7-1 doesn't show it, TCTI may be the interface between the RM and the device driver. In this case, the TCTI appears at multiple layers in the stack.
- At this time, we think the most common implementation will combine the TAB and the RM into a single module.

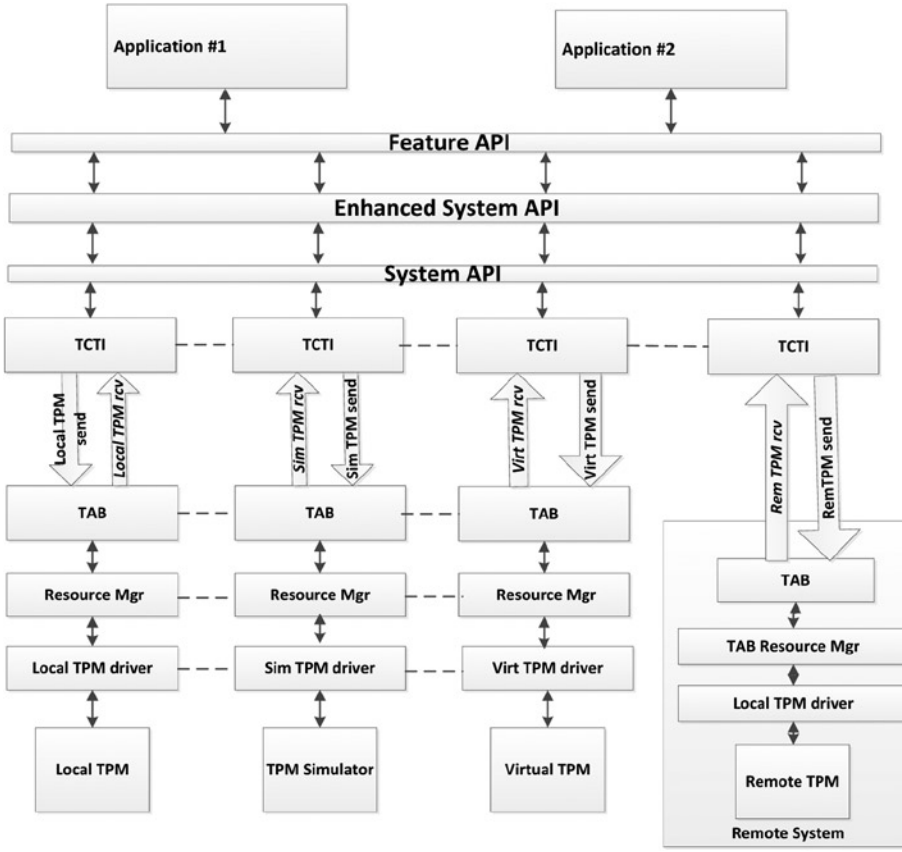

Figure 7-1. *TSS diagram*

The following sections describe each of the TSS layers.

Feature API

The TSS Feature API (FAPI) was created specifically to make the most-used facilities of the TPM 2.0 easily available to programmers. As such, it does not allow use of all the corner cases that a TPM is capable of doing.

It was designed with the hope that 80% of programs that would eventually use the TPM could be written by using the FAPI without having to resort to using other TSS APIs. It was also designed to minimize the number of calls you have to use and the number of parameters you have to define.

One way this was accomplished was by using a profile file to create default selections so you don't have to select algorithms, key sizes, crypto modes, and signing schemas explicitly when creating and using keys. It's assumed that *users* are normally the ones

who wish to select a matched set of algorithms, and you can default to user-selected configurations. In cases where you want to explicitly select a configuration file, you may do this as well, but default configurations are always selected by the user. FAPI implementations ship with pre-created configuration files for most common choices. For example:

- The `P_RSA2048SHA1` profile uses RSA 2048-bit asymmetric keys using PKCS1 version 1.5 for a signing scheme, SHA-1 for the hash algorithm, and AES128 with CFB mode for asymmetric encryption.
- The `P_RSA2048SHA256` profile uses RSA 2048-bit asymmetric keys using PKCS#1 version 1.5 for a signing scheme, SHA-256 for the hash algorithm, and AES-128 with CFB mode for asymmetric encryption.
- The `P_ECCP256` profile uses NIST ECC with prime field 256-bit asymmetric keys using ECDSA as a signing schema, SHA-1 for the hash algorithm, and AES-128 with CFB mode for asymmetric encryption.

Path descriptions are used to identify to the FAPI where to find keys, policies, NV, and other TPM objects and entities. Paths have a basic structure that looks like this:

```
<Profile name> / <Hierarchy> / <Object Ancestor> / key tree
```

If the profile name is omitted, the default profile chosen by the user is assumed. If the hierarchy is omitted, then the storage hierarchy is assumed. The storage hiearchy is H_S, the Endorsement hiearchy is H_E, and the Platform hierarchy is H_P. The object ancestor can be one of the following values:

- `SNK`: The system ancestor for non-duplicable keys
- `SDK`: The system ancestor for duplicable keys
- `UNK`: The user ancestor for non-duplicable keys
- `UDK`: The user ancestor for duplicable keys
- `NV`: For NV indexes
- `Policy`: For instances of policies

The key tree is simply a list of parent and children keys separated by / characters. The path is insensitive to capitalization.

Let's look at some examples. Assuming the user has chosen the configuration file `P_RSA2048SHA1`, all of the following paths are equivalent:

```
P_RSA2048SHA1/H_S/SNK/myVPNkey

H_S/SNK/myVPNkey

SNK/myVPNkey
```

```
P_RSA2048SHA1/H_S/SNK/MYVPNKEY
```

```
H_S/SNK/MYVPNKEY
```

```
SNK/MYVPNKEY
```

An ECC P-256 NIST signing key under a user's backup storage key might be:

```
P_ECCP256/UDK/backupStorageKey/mySigningKey
```

The FAPI also has some basic names for default types of entities.

Keys:

- `ASYM_STORAGE_KEY`: An asymmetric key used to store other keys/data.
- `EK`: An endorsement key that has a certificate used to prove that it (and, in the process, prove that other keys) belongs to a genuine TPM.
- `ASYM_RESTRICTED_SIGNING_KEY`: A key like the AIK of 1.2, but that can also sign any external data that doesn't claim to come from the TPM.
- `HMAC_KEY`: An unrestricted symmetric key. Its main use is as an HMAC key that can be used to sign (HMAC) data that isn't a hash produced by the TPM.

NV:

- `NV_MEMORY`: Normal NV memory.
- `NV_BITFIELD`: a 64-bit bitfield.
- `NV_COUNTER`: A 64-bit counter.
- `NV_PCR`: A `NV_PCR` that uses the template hash algorithm.
- `NV_TEMP_READ_DISABLE`: Can have its readability turned off for a boot cycle.

Standard polies and authentications:

- `TSS2_POLICY_NULL`: A `NULL` policy (empty buffer) that can never be satisfied.
- `TSS2_AUTH_NULL`: A zero-length password, trivially satisfied.
- `TSS2_POLICY_AUTHVALUE`: Points to the object's authorization data.
- `TSS2_POLICY_SECRET_EH`: Points to the endorsement hierarchy's authorization data.
- `TSS2_POLICY_SECRET_SH`: Points to the storage hierarchy's authorization data.

- TSS2_POLICY_SECRET_PH: Points to the platform hierarchy's authorization data.
- TSS2_POLICY_SECRET_DA: Points to the dictionary attack handle's authorization data.
- TSS2_POLICY_TRIVIAL: Points to a policy of all zeroes. This is easy to satisfy because every policy session starts with its policy buffer equal to this policy. This can be used to create an entity that can be trivially satisfied with the FAPI.

All objects created and used by FAPI commands are authorized by a policy. This doesn't mean the authorization value can't be used: it can be used if the policy is TSS2_POLICY_AUTHVALUE. However, under the covers, a password session is never used. And if an authorization value is used, it's always done with a salted HMAC session.

One structure used constantly in the FAPI is TSS2_SIZED_BUFFER. This structure consists of two things: a size and a pointer to a buffer. The size represents the size of the buffer:

```
typedef struct { size_t    size;
               uint8_t     *buffer;
               } TSS2_SIZED_BUFFER;
```

You need to know one more thing before writing a program: at the beginning of your program, you must create a context, which you must destroy when you're done with it.

Let's write an example program that creates a key, uses it to sign "Hello World," and verifies the signature. Follow these steps:

1. Create a context. Tell it to use the local TPM by setting the second parameter to NULL:

   ```
   TSS2_CONTEXT *context;
   Tss2_Context_Intialize(&context, NULL);
   ```

2. Create a signing key using the user's default configuration. Here you explicitly tell it to use the P_RSA2048SHA1 profile instead of the default. By using the UNK, you tell it that it's a user key that is non-duplicable. Name it mySigningKey.

   Using ASYM_RESTRICTED_SIGNING_KEY makes the key a signing key. You also give it a trivially satisfied policy and a password of NULL:

   ```
   Tss2_Key_Create(context, // pass in the context I just created
           "P_RSA2048SHA1/UNK/mySigningKey",  // non-duplicable
   RSA2048
           ASYM_RESTRICTED_SIGNING_KEY,       // signing key
           TSS2_POLICY_TRIVIAL,               // trivially policy
           TSS2_AUTH_NULL);                   // the password is NULL
   ```

3. Use the key to sign "Hello world." First you have to hash "Hello World" with an OpenSSL library call:

```
TSS2_SIZED_BUFFER myHash;
myHash.size=20
myHash.buffer=calloc(20,1);
SHA1("Hello World",sizeof("Hello World"),myHash.buffer);
```

4. The Sign command returns everything necessary to verify the signature. Because you just created this key, the certificate comes back with a certificate that is empty:

```
TSS2_SIZED_BUFFER signature, publicKey,certificate;

Tss2_Key_Sign(context,    // pass in the context
        "P_RSA2048SHA1/UNK/mySigningKey", // the signing key
        &myHash,
        &signature,
        &publicKey,
        &certificate);
```

5. At this point you could save the outputs, but instead let's check them:

```
if (TSS_SUCCESS!=Tss2_Key_Verify(context ,&signature,
   &publicKey,&myHash) )
{
     printf("The command failed signature verification\n");
}
else printf("The command succeeded\n");
```

6. Destroy the buffers that have been allocated, now that you're done with them:

```
free(myHash.buffer);
free(signature.buffer);
free(publicKey.buffer);
/* I don't have to free the certificate buffer, because
it was empty */
Tss2_Context_Finalize(context);
```

It's easy to see that this example cheats a little. In particular, the key doesn't require any type of authorization. Next you will learn what to do if authentication is required.

All FAPI functions assume that keys are authenticated only through policy. If a key is to be authenticated with a password, then the password is assigned to the key, and a policy is created using TPM2_PolicyAuthValue. The predefined TSS2_POLICY_AUTHVALUE does this. However, this leaves you with the bigger question of how to satisfy the policy.

Policy commands come in two flavors. Some policy commands require interaction with the outside world:

- `PolicyPassword`: Asks for a password
- `PolicyAuthValue`: Asks for a password
- `PolicySecret`: Asks for a password
- `PolicyNV`: Asks for a password
- `PolicyOR`: Asks for a selection among choices
- `PolicyAuthorize`: Asks for a selection among authorized choices
- `PolicySigned`: Asks for a signature from a specific device

Other policy commands don't require outside interaction:

- `PolicyPCR`: Checks the values of the TPM's PCRs
- `PolicyLocality`: Checks the locality of the command
- `PolicyCounterTimer`: Checks the counter internal to the TPM
- `PolicyCommandCode`: Checks what command was sent to the TPM
- `PolicyCpHash`: Checks the command and parameters sent to the TPM
- `PolicyNameHash`: Checks the name of the object sent to the TPM
- `PolicyDuplicationSelect`: Checks the target of duplication of a key
- `PolicyNVWritten`: Checks if an NV index has ever been written

Many policies require a mix of the two. If a policy requires one of the authorizations of the second type, it's the responsibility of the FAPI to handle it. If it's an authorization of the first type, then you're responsible for providing to the FAPI the parameters it doesn't have access to.

This is done via a callback mechanism. You must register these callbacks in your program so that FAPI knows what do to if it requires a password, selection, or signature. The three callbacks are defined as follows:

- `TSS2_PolicyAuthCallback`: Used when a password is required
- `TSS2_PolicyBranchSelectionCallback`: Used when the user needs to select from among more than one policy in a `TPolicyOR` or `TPM2_PolicyAuthorize`
- `TSS2_PolicySignatureCallback`: Used when a signature is required to satisfy the policy

The first is easiest. After a context is registered, you have to create a callback function that is used when the FAPI is asked to execute a function that requires interaction with the user asking for a password. In this case, the FAPI sends back to the program the description of the object that needs to be authorized and requests the authorization

data. The FAPI takes care of salting and HMACing this authorization data. The user must do two things: create the function that asks the user for their password, and register this function so that the FAPI can call it.

Here is a simple password-handler function:

```
myPasswordHandler (TSS2_CONTEXT           context,
                   void                  *userData,
                   char const            *description,
                   TSS2_SIZED_BUFFER     *auth)
{
/* Here the program asks for the password in some application specific
way. It then puts the result into the auth variable. */
return;
}
```

Here is how you register it with the FAPI so it knows to call the function:

```
Tss2_SetPolicyAuthCallback(context, TSS2_PolicyAuthCallback, NULL);
```

Creating and registering the other callbacks is very similar.

At the time of writing this book, the specification for using XML to write a policy for a command has not yet been written, although it's likely to come out in 2014. However, one thing is known: it will be possible for hardware OEMs (for example, a smartcard provider) to provide a library that contains these callback functions. In this case, the callback function will be registered in the policy rather than in the program, so you won't need to provide it. Similarly, software libraries can be used to provide these callback functions in policies. If this is done, you won't have to register any callbacks.

System API

As mentioned earlier, the SAPI layer is the TPM 2.0 equivalent of programming in the C language. SAPI provides access to all the capabilities of TPM 2.0; as is often said in this business when describing low-level interfaces, we give application writers all the rope they need to hang themselves. It's a powerful and sharp tool, and expertise is required to use it properly.

The SAPI specification can be found at www.trustedcomputinggroup.org/developers/software_stack. The design goals of the SAPI specification were the following:

- Provide access to all TPM functionality.
- Be usable across the breadth of possible platforms, from highly embedded, memory-constrained environments to multiprocessor servers. To support small applications, much consideration was given to minimizing, or at least allowing minimization, of the memory footprint of the SAPI library code.

- Within the constraint of providing access to all functionality, make programmers' jobs as easy as possible.
- Support both synchronous and asynchronous calls to the TPM.
- SAPI implementations aren't required to allocate any memory. In most implementations, the caller is responsible to allocate all memory used by the SAPI.

There are four groups of SAPI commands: command context allocation, command preparation, command execution, and command completion. Each of these groups is described in this section. Within the command preparation, execution, and completion groups, there are some utility functions that are used regardless of which TPM 2.0 command in Part 3 of the TPM specification is being called; others are specific to each Part 3 command.

First we will describe each of the four groups of commands at a high level. As these commands are described, we will show code fragments for a very simple code example, a TPM2_GetTestResult command. At the end, we will combine these fragments into a single program to do a `TPM2_GetTestResult` command using three different methods: one call, asynchronous, and synchronous multi-call. Code examples for SAPI functions that require knowledge of sessions and authorizations and encryption and decryption are deferred until Chapters 13 and 17; the SAPI functions that support these features will only make sense after you understand the features. This chapter ends with a brief description of the test code that is distributed with the System API code.[3]

Command Context Allocation Functions

These functions are used to allocate a SAPI command context data structure, an opaque structure that is used by the implementation to maintain any state data required to execute the TPM 2.0 command.

The `Tss2_Sys_GetContextSize` function is used to determine how much memory is needed for the SAPI context data structure. The command can return the amount of memory required to support any TPM 2.0 Part 3 command, or the caller can provide a maximum command or response size and the function calculates the context size required to support that.

`Tss2_Sys_Initialize` is used to initialize a SAPI context. It takes as inputs a pointer to a memory block of sufficient size for the context, the context size returned by `Tss2_Sys_GetContextSize`, a pointer to a TCTI context (described in the later "TCTI" section) used to define the methods for transmitting commands and receiving responses, and the calling application's required SAPI version information.

[3]The code in this SAPI section is working code that is included in the SAPI and test code package. This package is currently shared among TCG members via a GitHub site. TCG members can contact TSS Workgroup members to gain access to it. It is expected that this code will be open sourced before or shortly after this book is published.

■ **Note** One note about the following code: `rval` is shorthand for *return value* and is a 32-bit unsigned integer. This is used repeatedly in upcoming code examples.

Here's a code example for a function that creates and initializes a system context structure.

■ **Note** The function that follows is declared to return a pointer to a `TSS2_SYS_CONTEXT` structure. This structure is defined as follows:

```
typedef struct _TSS2_SYS_OPAQUE_CONTEXT_BLOB TSS2_SYS_CONTEXT;
```

But the opaque structure is never defined anywhere. This works because `TSS2_SYS_CONTEXT` structures are always referenced by a pointer. Basically, this is a compiler trick that provides an advantage over using `void` pointers: it performs some compile time type checking.

```
//
// Allocates space for and initializes system
// context structure.
//
// Returns:
//   ptr to system context, if successful
//   NULL pointer, if not successful.
//
TSS2_SYS_CONTEXT *InitSysContext(
    UINT16 maxCommandSize,
    TSS2_TCTI_CONTEXT *tctiContext,
    TSS2_ABI_VERSION *abiVersion
)
    UINT32 contextSize;
    TSS2_RC rval;
    TSS2_SYS_CONTEXT *sysContext;

    // Get the size needed for system context structure.
    contextSize = Tss2_Sys_GetContextSize( maxCommandSize );

    // Allocate the space for the system context structure.
    sysContext = malloc( contextSize );
    if( sysContext != 0 )
    {
        // Initialize the system context structure.
        rval = Tss2_Sys_Initialize( sysContext,
                contextSize, tctiContext, abiVersion );
```

```
        if( rval == TSS2_RC_SUCCESS )
            return sysContext;
        else
            return 0;
    }
    else
    {
        return 0;
    }
}
```

The last function in this group is `Tss2_Sys_Finalize`, which is a placeholder for any functionality that may be required to retire a SAPI context data structure before its allocated memory is freed. Here's an example of how this might be used:

```
void TeardownSysContext( TSS2_SYS_CONTEXT *sysContext )
{
    if( sysContext != 0 )
    {
        Tss2_Sys_Finalize(sysContext);

        free(sysContext);
    }
}
```

■ **Note** In this case, `Tss2_Sys_Finalize` is a dummy function that does nothing, because the SAPI library code doesn't need it to do anything. Note that the system context memory is freed after the `Finalize` call.

Command Preparation Functions

As explained in Chapters 13 and 17, HMAC calculation, command parameter encryption, and response parameter decryption often require pre- and post-command processing. The command preparation functions provide the pre-command execution functions that are needed before actually sending the command to the TPM.

In order to calculate the command HMAC and encrypt command parameters, the command parameters must be marshalled. This could be done with special application code, but because the SAPI already contains this functionality, the API designers decided to make this functionality available to the application. This is the purpose of the `Tss2_Sys_XXXX_Prepare` functions. Because the command parameters are unique for each Part 3 command, there is one of these functions for each TPM command that needs it. The "XXXX" is replaced by the command name; for instance, the `Tss2_Sys_XXXX_Prepare`

function for TPM2_StartAuthSession is Tss2_Sys_StartAuthSession_Prepare. Following is a call to the prepare code for TPM2_GetTestResult:

```
rval = Tss2_Sys_GetTestResult_Prepare( sysContext );
```

■ **Note** The only parameter to this function is a pointer to the system context, because TPM2_GetTestResult has no input parameters.

After the Tss2_Sys_XXXX_Prepare call, the data has been marshalled. To get the marshalled command parameter byte stream, the Tss2_Sys_GetCpParam function is called. This returns the start of the cpBuffer, the marshalled command parameter byte stream, and the length of the cpBuffer. How this is used is described further in Chapters 13 and 17.

Another function that is needed to calculate the command HMAC is Tss2_Sys_GetCommandCode. This function returns the command code bytes in CPU endian order. This function is also used in command post-processing.

The Tss2_Sys_GetDecryptParam and Tss2_Sys_SetDecryptParam functions are used for decrypt sessions, which you learn about in Chapter 17. For now, the Tss2_Sys_GetDecryptParam function returns a pointer to the start of the parameter to be encrypted and the size of the parameter. These two returned values are used by the application when it calls Tss2_Sys_SetDecryptParam to set the encrypted value into the command byte stream.

The Tss2_Sys_SetCmdAuths function is used to set the command authorization areas (also called *sessions*) in the command byte stream. This is explained in detail in Chapter 13, when sessions and authorizations are discussed.

Command Execution Functions

This group of functions actually sends commands to and, receives responses from the TPM. The commands can be sent synchronously or asynchronously. There are two ways to send commands synchronously: via a sequence of three to five function calls; and via a single "does everything" call, the *one-call*. Support for asynchronous vs. asynchronous and one-call vs. a finer-grained multi-call approach arose from the desire to support as many application architectures as possible.

Tss2_Sys_ExecuteAsync is the most basic method of sending a command. It sends the command using the TCTI transmit function and returns as quickly as possible. Here's an example of a call to this function:

```
rval = Tss2_Sys_ExecuteAsync( sysContext );
```

`Tss2_Sys_ExecuteFinish` is the companion function to `ExecuteAsync`. It calls the TCTI function to receive the response. It takes a command parameter, `timeout`, that tells it how long to wait for a response. Here's an example that waits 20 msec for a response from the TPM:

```
rval = Tss2_Sys_ExecuteFinish( sysContext, 20 );
```

`Tss2_Sys_Execute` is the synchronous method and is the equivalent of calling `Tss2_Sys_ExecuteAsync` followed by `Tss2_Sys_ExecuteFinish` with an infinite timeout. Here's an example:

```
rval = Tss2_Sys_Execute( sysContext );
```

The last function in the execution group, `Tss2_Sys_XXXX`, is the *one-call* or "do everything" function. This function assumes that authorizations aren't needed, a simple password authorization is being used, or that authorizations such as HMAC and policy have already been calculated. There is one of these commands for each Part 3 command.[4] As an example, the one-call function for the `Tpm2_StartAuthSession` command is `Tss2_Sys_StartAuthSession`. When used with the associated `Tss2_Sys_XXXX_Prepare` call, the one-call interface can do any type of authorization. An interesting side effect of this is that the command parameters are marshalled twice: once during the `Tss2_Sys_XXXX_Prepare` call and once during the one-call function call. This was a design compromise because the one-call needed to be capable of being used as a standalone call and paired with the `Tss2_Sys_XXXX_Prepare` call. Here's an example of the one-call with no command or response authorizations:

```
rval = Tss2_Sys_GetTestResult( sysContext, 0, &outData, &testResult, 0 );
```

■ **Note** The function takes a pointer to a system context structure; a pointer to a command authorization's array structure; two output parameters, `outData` and `testResult`; and a pointer to a response authorization structure. The parameters that are 0 are the command and response authorization array structures. For this very simple example, these aren't necessary, so `NULL` pointers are used. Use of these is explained in Chapter 13.

Command Completion Functions

This group of functions enables the command post-processing that is required. This includes response HMAC calculation and response parameter decryption if the session was configured as an encrypt session.

[4]Part 3 does describe some hardware-triggered commands. These start with an underscore character and aren't included in the SAPI.

`Tss2_Sys_GetRpBuffer` gets a pointer to and the size of the response parameter byte stream. Knowing these two values enables the caller to calculate the response HMAC and compare it to the HMAC in the response authorization areas.

`Tss2_Sys_GetRspAuths` gets the response authorization areas. These are used to check the response HMACs in order to validate that the response data hasn't been tampered with.

After validating the response data, if the response was sent using an encrypt session, `Tss2_Sys_GetEncryptParam` and `Tss2_Sys_SetEncryptParam` can be used to decrypt the encrypted response parameter and insert the decrypted response parameter into the byte stream prior to unmarshalling the response parameters. These two functions are described in greater detail in Chapter 17 in the discussion of decrypt and encrypt sessions.

After the response parameter has been decrypted, the response byte stream can be unmarshalled. This is done by a call to `Tss2_Sys_XXXX_Complete`. Because each command has different response parameters, there is one of these per Part 3 command.[5] An example of this call is as follows:

```
rval = Tss2_Sys_GetTestResult_Complete( sysContext, &outData, &testResult );
```

You've now seen all the SAPI calls. Some of these are specific to Part 3 commands, and some apply regardless of which Part 3 command is being executed.

Simple Code Example

The next code example, from the SAPI library test code, performs a `TPM2_GetTestResult` command three different ways: one-call, synchronous calls, and asynchronous calls. Comments help delineate the tests of the three different ways:

■ **Note** `CheckPassed()` is a routine that compares the passed-in return value to 0. If they aren't equal, an error has occurred, and the routine prints an error message, cleans up, and exits the test program.

```
void TestGetTestResult()
{
        UINT32 rval;
        TPM2B_MAX_BUFFER    outData;
        TPM_RC              testResult;
        TSS2_SYS_CONTEXT    *systemContext;

        printf( "\nGET TEST RESULT TESTS:\n" );
```

[5]Commands that have no response parameters don't have a corresponding `Complete` call.

```
// Initialize the system context structure.
systemContext = InitSysContext( 2000, resMgrTctiContext, &abiVersion );
if( systemContext == 0 )
{
        Handle failure, cleanup, and exit.
        InitSysContextFailure();
}
```

Test the one-call API.

```
//
// First test the one-call interface.
//
rval = Tss2_Sys_GetTestResult( systemContext, 0, &outData, &testResult,
      0 );
CheckPassed(rval);
```

Test the synchronous, multi-call APIs.

```
//
// Now test the synchronous, non-one-call APIs.
//
rval = Tss2_Sys_GetTestResult_Prepare( systemContext );
CheckPassed(rval);
// Execute the command synchronously.
rval = Tss2_Sys_Execute( systemContext );
CheckPassed(rval);

// Get the command results
rval = Tss2_Sys_GetTestResult_Complete( systemContext, &outData,
        &testResult );
CheckPassed(rval);
```

Test the asynchronous, multi-call APIs.

```
//
// Now test the asynchronous, non-one-call interface.
//
rval = Tss2_Sys_GetTestResult_Prepare( systemContext );
CheckPassed(rval);
```

```
        // Execute the command asynchronously.
        rval = Tss2_Sys_ExecuteAsync( systemContext );
        CheckPassed(rval);

        // Get the command response. Wait a maximum of 20ms
        // for response.
        rval = Tss2_Sys_ExecuteFinish( systemContext, 20 );
        CheckPassed(rval);

        // Get the command results
        rval = Tss2_Sys_GetTestResult_Complete( systemContext, &outData,
        &testResult );
        CheckPassed(rval);

        // Tear down the system context.
        TeardownSysContext( systemContext );
}
```

System API Test Code

As mentioned, the previous `GetTestResult` test is included as one of the tests in the SAPI test code. This section briefly describes the structure of the test code and some design features.

Many other tests in this code test various SAPI capabilities. But you should beware that this test suite is by no means comprehensive; there are too many permutations and not enough time for a single developer to write all the tests. These tests were written to provide sanity checks and, in some cases, more detailed tests of targeted functionality.

The test code resides in the `Test\tpmclient` subdirectory. In this directory, the `tpmclient.cpp` file contains the test application's initialization and control code as well as all the main test routines. Subdirectories of `tpmclient` provide support code needed for the tests. The `simDriver` subdirectory contains a device driver for communicating with the TPM simulator. The `resourceMgr` subdirectory contains code for a sample RM. And the `sample` subdirectory contains application-level code that performs the following tasks: maintaining session state information, calculating HMACs, and performing cryptographic functions.

A major design principle of the SAPI test code was to use the TPM itself for all cryptographic functions. No outside libraries such as OpenSSL are used. The reason for this was twofold. First, it increased the test coverage of the SAPI test code by calling TPM cryptographic commands. Second, it allowed the test application to be a stand-alone application with no dependency on outside libraries. And there was a third reason: the developer thought it was kind of a cool thing to do! The SAPI test code can be used as a starting point for developers: find a command you want to use that's called in the test code, and it will give you a significant boost in your code development.

The SAPI test code uses other elements of the TSS stack to perform its tests: the TCTI, TAB, and RM. Because SAPI uses the TCTI to send commands to the TAB, TCTI is described next.

TCTI

You've seen the system API functions, but the question that hasn't been answered yet is how command byte streams are transmitted to the TPM and how the application receives response byte streams from the TPM. The answer is the TPM Command Transmission Interface (TCTI). You saw this briefly in the description of the `Tss2_Sys_Initialize` call. This call takes a TCTI context structure as one of its inputs. Now we will describe this layer of the stack in detail.

The TCTI context structure tells the SAPI functions how to communicate with the TPM. This structure contains function pointers for the two most important TCTI functions, `transmit` and `receive`, as well as less frequently used functions such as `cancel`, `setLocality`, and some others described shortly. If an application needs to talk to more than one TPM, it creates multiple TCTI contexts and sets each with the proper function pointers for communicating with each TPM.

The TCTI context structure is a per-process, per-TPM structure that is set up by initialization code. It can be set up at compile time or dynamically when the OS is booted. Some process has to either discover the presence of TPMs (typically a local TPM) or have a priori knowledge of remote TPMs and initialize a TCTI context structure with the proper function pointers for communication. This initialization and discovery process is out of scope of the SAPI and TCTI specification.

The most frequently used and required function pointers, `transmit` and `receive`, do what you'd expect them to. Both of them get a pointer to a buffer and a size parameter. The SAPI functions call them when they're ready to send and receive data, and the functions do the right thing.

The `cancel` function pointer supports a new capability in TPM 2.0: the ability to cancel a TPM command after it's been transmitted to the TPM. This allows a long-running TPM command to be cancelled. For example, key generation can take up to 90 seconds on some TPMs. If a sleep operation is initiated by the OS, this command allows early cancellation of long-running commands so that the system can be quiesced.[6]

The `getPollHandles` function pointer comes into play when SAPI is using the asynchronous method of sending and receiving responses—that is, the `Tss2_Sys_ExecuteAsync` and `Tss2_Sys_ExecuteFinish` functions. This is an OS-specific function that returns the handles that can be used to poll for response-ready conditions.

The last function pointer, `finalize`, is used to clean up before a TCTI connection is terminated. Actions that are required upon connection termination, if any, are performed by this function.

TCTI can be used at any level in the TPM stack where marshalled byte streams are being transmitted and received. Currently, the thinking is that this occurs at two places: between the SAPI and the TAB, and between the RM and the driver.

[6]The `cancel` capability is specified in the TCG PC Client Platform TPM Profile (PTP) Specification. TPMs that support other platforms may not include the `cancel` command.

TPM Access Broker (TAB)

The TAB is used to control and synchronize multiprocess access to a single shared TPM. When one process is in the middle of sending a command and receiving a response, no other process is allowed to send commands to or request responses from the TPM. This is the first responsibility of the TAB. Another feature of the TAB is that it prevents processes from accessing TPM sessions, objects, and sequences (hash or event sequences) that they don't own. Ownership is determined by which TCTI connection was used to load the objects, start the sessions, or start the sequences.

The TAB is integrated with the RM into a single module in most implementations. This makes sense because a typical TAB implementation can consist of some simple modifications to the RM.

Resource Manager

The RM acts in a manner similar to the virtual memory manager in an OS. Because TPMs generally have very limited on-board memory, objects, sessions, and sequences need to be swapped from the TPM to and from memory to allow TPM commands to execute. A TPM command can use at most three entity handles and three session handles. All of these need to be in TPM memory for the TPM to execute the command. The job of the RM is to intercept the command byte stream, determine what resources need to be loaded into the TPM, swap out enough room to be able to load the required resources, and load the resources needed. In the case of objects and sequences, because they can have different handles after being reloaded into the TPM, the RM needs to virtualize the handles before returning them to the caller.[7] This is covered in more detail in Chapter 18; for now, this ends the brief introduction to this component.

The RM and TAB are usually combined into one component, the TAB/RM, and as a rule there is one of these per TPM; that's an implementation design decision, but this is typically the way it's done. If, on the other hand, a single TAB/RM is used to provide access to all the TPMs present, then the TAB/RM needs a way to keep track of which handles belong to which TPMs and keep them separated; the means of doing this is outside the scope of the TSS specifications. So, whether the boundary is enforced by different executable code or different tables in the same code module, clear differentiation must be maintained in this layer between entities that belong to different TPMs.

Both the TAB and RM operate in a way that is mostly transparent to the upper layers of the stack, and both layers are optional. Upper layers operate the same with respect to sending and receiving commands and responses, whether they're talking directly to a TPM or through a TAB/RM layer. However, if no TAB/RM is implemented, upper layers of the stack must perform the TAB/RM responsibilities before sending TPM commands, so that those commands can execute properly. Generally, an application executing in a multithreaded or multiprocessing environment implements a TAB/RM to isolate application writers from these low-level details. Single-threaded and highly embedded applications usually don't require the overhead of a TAB/RM layer.

---

[7]For this reason, handles aren't included in authorization calculations. Otherwise, authorizations would fail because the application only sees virtual handles. Names are used instead, and these names aren't affected by virtualization of the handles.

Device Driver

After the FAPI, ESAPI, SAPI, TCTI, TAB, and RM have done their jobs, the last link, the device driver, steps up to the plate. The device driver receives a buffer of command bytes and a buffer length and performs the operations necessary to send those bytes to the TPM. When requested by higher layers in the stack, the driver waits until the TPM is ready with response data and reads that response data and returns it up the stack.

The physical and logical interfaces the driver uses to communicate with the TPM are out of scope of the TPM 2.0 library specification and are defined in the platform-specific specifications. At this time, the choice for TPMs on PCs is either the FIFO[8] or Command Response Buffer (CRB) interface. FIFO is first-in, first-out byte-transmission interface that uses a single hardcoded address for data transmission and reception plus some other addresses for handshaking and status operations. The FIFO interface remained mostly the same for TPM 2.0, with a few small changes. FIFO can operate over serial peripheral interface (SPI) or low pin count (LPC) interface busses.

The CRB interface is new for TPM 2.0. It was designed for TPM implementations that use shared memory buffers to communicate commands and responses.

Summary

This completes the discussion of the TSS layers, which provide a standard API stack for "driving" the TPM. You can intercept this stack at different levels depending on your requirements. These layers, especially FAPI and SAPI, are used in the following chapters, so please refer to this this chapter while studying the code examples.

[8]The FIFO interface is mostly identical to the interface used by the TPM Interface Specification (TIS) for TPM 1.2 devices. The TIS specification included much more than the interface, such as the number of PCRs, a minimum set of commands, and so on, so the use of "TIS" has been deprecated for TPM 2.0.