CHAPTER 20

Debugging

"I still remember my early days of Z80 assembly language programming: a couple of times, just to get a reaction, I bragged that I could write bug free code the first time. Just write, assemble, link and voila, a perfectly working program! I was scoffed at and justifiably so—truthfully, in over 30 years of experience, I've never achieved that and probably never will."

Will Arthur, reflecting while writing this chapter

It is possible that somebody somewhere has written a significant program that ran without bugs the first time, but if so, it's in the noise, statistically speaking. Because the process of programming seems to also be the process of inserting bugs, the need for debugging is an inexorable law of the programming world.

This chapter aims to educate you about some specific tools and methods for debugging TPM 2.0 programs as well as debugging common bug areas. We will discuss two major areas of debugging: lower-level applications that communicate with the TPM and higher-level applications that use the Feature API to communicate with the TPM. Lower-level applications include applications that use the ESAPI and SAPI, as well as implementations of specialized TSS layers such as the SAPI library, TPM Access Broker (TAB) and Resource Manager (RM). Most of the text for the lower-level applications flows directly from the experience of that section's author in developing the System API, TAB, RM, and TPM 2.0 device driver, all very low-level pieces of software. Because of the current lack of implementations for the Feature API and Enhanced System API layers of the TSS stack, no TSS 2.0–specific tribal knowledge is available for debugging those areas. For this reason, the chapter relies on debugging experiences from TSS 1.2, because we expect that many of the issues are similar.

Low-Level Application Debugging

Because low-level applications are the only ones we have actual experience debugging for TPM 2.0, we discuss them first.

The Problem

When a TPM command has an error, an error code is returned. The TPM 2.0 specification writers worked very hard to design error codes that facilitate debugging. Many times, the error code tells you exactly what's wrong, but due to lack of familiarity with the specification, you're left scratching your head. And in a few cases, the error code only indicates an area to look for the error, without the granularity needed to identify the exact cause. Either way, whether due to knowledge or specification deficiencies, you, the poor programmer, are left wondering what to do.

This section describes a few different error conditions and illustrates a hierarchy of techniques used to debug these types of problems, from simplest to more complex:

- *Error code analysis:* Many simple errors can be completely debugged this way.
- *Debug trace analysis:* This requires instrumenting the low-level driver to spit out the command byte stream being sent to the TPM and the response byte stream received from the TPM. Quite often, comparing known good debug trace data to bad data quickly highlights the problem.
- *More complicated errors:* These take more work to debug. An HMAC authorization error is quickly spotted by the error code, but debugging it requires detailed examination of all the steps that were used to generate the HMAC.
- *The hardest errors to debug:* These require stepping through the TPM 2.0 simulator with a debugger in order to understand what the TPM is unhappy about. Often, after debugging through the simulator code, the answer becomes obvious. A better understanding of the specification would have uncovered the problem. Human nature being what it is, we often get distracted by the forest and fail to see the particular tree that needs to be eradicated. Hence, this technique is often needed. The author of this section has debugged many errors this way both for himself and for others within Intel developing TPM 2.0 code.

Analyze the Error Code

The first step when you get an error is decoding the error code. This decoding is described in the "Response Code Details" section of Part 1 of the TPM 2.0 specification. The "Response Code Evaluation" flowchart in that section is particularly helpful.

The first example error involves the TPM2_Startup command. The following code snippet generates an error (this is a slight modification of the TPM2_Startup test described in Chapter 7:

```
rval = Tss2_Sys_Startup( sysContext, 03 )[1];
CheckPassed(rval);
```

The call to CheckPassed fails because an error code of 0x000001c4 was returned instead of 0 (TPM2_RC_SUCCESS). Following the flowchart mentioned previously, you can decode this error as follows:

```
Bit 8: 1
Bit 7: 1
Bit 6: 1
```

These bits indicate that the error code is in bits 5:0 and the parameter number is in bits 11:8.[2] So, parameter #1 is bad, and the specific error is TPM_RC_VALUE from the "TPM_RC Values" section of Part 1 of the TPM 2.0 specification. The text description for this error is, "value is out of range or isn't correct for the context." This means parameter #1 was bad for this command. In looking at the description of the TPM2_Startup command, it's easy to see that the only values allowed for this parameter are TPM_SU_CLEAR(0x0000) and TPM_SU_STATE (0x0001). Obviously, using 0x3 for the parameter is the source of our error.

We strongly encourage the use of a tool for decoding errors, because hand-decoding errors can become burdensome for repeated debug sessions. Sample output of such a tool looks like this[3]:

```
>tpm2decoderring /e 1c4

ERROR:  PARAM #1, TPM_RC_VALUE: value is out of range or is not correct for
the context
```

Debug Trace Analysis

Quite often, due to lack of TPM knowledge or, in some cases, obscurity of the error, error-code analysis is insufficient. Additionally, if a program was previously working, a comparison to the output from the previously working program can highlight the error much more quickly. For this reason, we highly recommend instrumenting

[1]Experienced programmers will immediately notice the use of a "magic" number in this line of code. There is no constant defined in the TPM specification for a bad parameter to the startup command. Although not generally a good programming practice, in this case hardcoding the number seems better than defining a special value for use here.

[2]In some cases, the error code may indicate a parameter, session, or handle number of 0. This indicates an error with a parameter, session, or handle, but the TPM isn't giving any indication of which one.

[3]This is the output from an Intel internal tool. This tool hasn't been released publicly, but development of such a tool for public use is strongly encouraged.

the TPM device driver with code that displays the command bytes sent to the TPM and the response bytes received from the TPM. This feature has saved many weeks of debugging time over the past two years of TPM 2.0 development work.

In the case of the previous program, the trace dump from the command from a good program looks like this:

```
Cmd sent: TPM2_Startup
Locality = 3
80 01 00 00 00 0c 00 00 01 44 00 00

Response Received:
80 01 00 00 00 0a 00 00 00 00
    passing case:      PASSED!
```

The trace dump from the bad program, with the differences highlighted, looks like this:

```
cmd sent: TPM2_Startup
Locality = 3
80 01 00 00 00 0c 00 00 01 44 00 03

Response Received:
80 01 00 00 00 0a 00 00 01 c4
    passing case:      FAILED!  TPM Error -- TPM Error: 0x1c4
```

Use of a good compare utility quickly highlights the bad value, `00 03`, in the bad command.

One caveat in this method is that much of the TPM output is randomized, and often these randomized outputs are fed back as inputs to other commands. This means these randomized parts of your trace data need to be ignored when you visually compare output. Experience will help you quickly learn which areas to ignore and which differences to focus on. It's not nearly as hard as it sounds.

Another good way to use trace dumps is when comparing multiple traces coming from different layers in the stack. For instance, you might have a trace dump from the ESAPI layer, one from the driver layer, and maybe even one from the TPM simulator. It can be challenging to synchronize these trace dumps. Session nonces, because they are random, unique, and the same no matter where in the stack they appear, can be used to synchronize these trace dumps. Find a nonce being returned in a session in one trace dump, and then search for where that same nonce is returned in the other trace dumps.

More Complex Errors

An example of a more complex error is an HMAC authorization error. This error is indicated by the error code, `TPM_RC_AUTH_FAIL`. The description of this error is, "the authorization HMAC check failed and DA counter incremented," and high-order bits in the error code tell which session is seeing the HMAC error.

Unfortunately, it isn't nearly so easy to debug this error. Many steps go into calculating the HMAC: key generation, hashing of input and/or output parameters, and the HMAC calculation. There are also many input parameters that feed into these steps: nonces, keys, and the type of authorization session. Any error in the data or steps used to generate the HMAC will result in a bad HMAC.

The only effective way we've found to debug these errors is to enhance the debug trace capabilities to display all the inputs and outputs for the key generation, hashing, and HMAC calculation steps. A very careful analysis of these inputs while carefully comparing to the TPM specification usually pinpoints the failure. This type of debugging requires very detailed knowledge of the TPM 2.0 specification—in particular, all the nuances of how HMACs are calculated.

Last Resort

The last category of errors consists of those that resist all other attempts at debugging. Typically these occur when implementing new TPM commands or features. There's no debug trace data from a previously working program to compare to, and error-code analysis doesn't pinpoint the problem. Fortunately, the situation isn't at all desperate; with the simulator, these errors can be easily debugged.

A common error in this category is a *scheme error*, `TPM_RC_SCHEME`. This error indicates something wrong with the scheme for a key, either when creating it or when using it. Schemes are typically unions of structures, each of which contains multiple fields. Much of the understanding of how to set up schemes is non-intuitive, especially to newcomers to TPM 2.0.

Often, the best way to debug these errors or any other errors that have resisted easier debugging techniques is to run the code against the TPM 2.0 simulator and single-step through the simulator. This provides an inside view of what the TPM is expecting to receive and why it's returning an error. Of course, this assumes that you have access to the source code of the simulator.[4] With the TPM source code, you can step into the TPM 2.0 simulator and figure out exactly why the TPM is complaining.

[4]Currently all TCG members have access to this source code.

The steps to debug this way are as follows:

1. Build and start the TPM 2.0 simulator on a Windows system[5] in Visual Studio. Review the instructions in Chapter 6 that describe how to do this. Select the "Debug" pull-down tab, and select "Start Debugging" to start the simulator running in debug mode.

2. Port your failing program to run against the simulator. The easiest way to do this is to create a subroutine using the System API functions and then add that subroutine to the list of test routines called by the System API test code. This way, because the System API test code, by default, communicates with the simulator, you don't have to develop a TPM 2.0 driver to talk to the simulator or deal with the simulator-specific platform commands to turn on the simulator, set the locality, and so on. You also get a TAB and resource manager for free. If you don't go this route, you must do all this work yourself.

3. Start your failing program in the debugger of your choice,[6] step to the place where it sends the failing command, and stop. Use a combination of single-stepping and breakpoints to get to this place.

4. Pause the simulator in its instance of Visual Studio by selecting the "Debug" pull-down and selecting "Break All".

5. Set a breakpoint in the simulator at an entry point that you know you'll hit. If you're new to the simulator, set a breakpoint at the `_rpc__Send_Command` function in the `TPMCmdp.c` source file.

6. Start the simulator running again, by selecting the Debug pull-down and selecting Continue.

7. In the test program's debugger, select the proper command to cause the test program to continue running from the breakpoint where it was stopped.

8. The simulator stops at the breakpoint you selected in the simulator code. From here you can debug into various subroutines and eventually figure out why the TPM is generating the error.

[5]Currently, only one TPM 2.0 simulator is available, and it only runs under Microsoft Visual Studio. If and when this changes, all steps related to debugging through the simulator will need to be altered accordingly.

[6]Because communication with the TPM 2.0 simulator is via sockets, the test program can be built and debugged on a remote system running any operating system. This means any debugger can be used to debug the test program. Chapter 6 describes how to run the test program on a remote system.

Common Bugs

Now that we've discussed debugging techniques for TPM 2.0 programs, we'll briefly describe some of the common bug areas. Again, keep in mind that these are all based on our experience with fairly low-level programming. These bugs are the types of issues that low-level TPM 2.0 programmers are likely to encounter. These bugs fall into the following categories: endianness, marshalling/unmarshalling errors, bad parameters (including the scheme errors mentioned earlier), and authorization errors.

When programming on a little-endian system such as an x86 system, endianness has to be properly altered during marshalling and marshalling of data. This is a very common source of errors, and it can typically be spotted by a careful analysis of the TPM trace data.

Marshalling and unmarshalling errors are closely related to endianness errors and in a similar manner can be easily debugged by looking at the trace data. This requires understanding the details of the TPM 2.0 specification, specifically Parts 2 and 3.

Bad parameters, including bad fields in schemes, are sometimes harder to spot. They require a very detailed understanding of all three parts of the TPM 2.0 specification in order to diagnose from the trace data. For this reason, debugging these often requires stepping into the simulator.

The last category of errors—authorization errors, whether HMAC or policy—requires a detailed analysis of the whole software stack that was used to generate the authorization. As mentioned earlier, this can be accelerated by enhanced trace messages that display the inputs and outputs to all the operations leading up to the command being authorized.

Debugging High-level Applications

Debugging applications, specifically those using the Feature API of the TSS, requires a different approach than debugging low-level software such as the TSS itself. This is because the expected errors are different. An application developer using a TSS shouldn't have to deal with bugs caused by parameter marshalling or unmarshalling, command and response packet parsing, and malformed byte stream errors. The reason is simple: the TSS libraries already perform those steps. Thus there should hopefully be no need to trace or decompose the command and response byte streams.

Our experience with TPM 1.2 applications—which we expect to carry forward to TPM 2.0—suggests that you should begin with the simulator. And we don't mean, "begin with the simulator after you hit a bug," but rather, start your developing using the simulator instead of a hardware TPM device. This approach offers several advantages:

- At least initially, hardware TPM 2.0 platforms may be scarce. The simulator is always available.
- A simulator should be faster than a hardware TPM, which is important when you start running regression tests. This becomes apparent when a test loop may generate a large number of RSA keys, or when NV space is rapidly written and a hardware TPM would throttle the writes to prevent wear-out.

- The simulator and TSS connect through a TCP/IP socket interface. This permits you to develop an application on one operating system (that might not yet have a TPM driver) while running the simulator on its supported platform.
- It's easy to restore a simulated TPM to its factory state by simply deleting its state file. A hardware TPM is harder to de-provision: you would have to write (and debug) a de-provisioning application.
- The normal TPM security protections (such as limited or no access to the platform hierarchy) don't get in the way.
- It's easy to "reboot" a simulated TPM without rebooting the platform. This eases tests for persistence issues and power management (suspend, hibernate) problems. It also speeds debugging.
- Finally, when it's time to debug, you already have the simulator environment set up.

Our experience with TPM 1.2 is that, once an application works with the simulator, it works unmodified with the hardware TPM.

Debug Process

Unlike the IBM TPM 1.2 simulator, the current Microsoft TPM 2.0 simulator, available to TCG members, has no tracing facility. You can't simply run the application and read the simulator's output. It's also unclear whether the TSS implementation will have a tracing capability. Trousers, the TPM 1.2 TSS, had little beyond command and response packet dumps.

However, the simulator source is available. The process is thus the same for nearly any application bug:

1. Run the application to failure, and determine which TPM command failed.
2. Run the simulator in a debugger, and set a breakpoint at the command. Each TPM 2.0 command has a C function call that has exactly the same name as the Part 3 command.
3. Step through the command until the error is detected.
4. It may be necessary to run again, stepping into the Part 4 subroutines, but our experience is that this is often unnecessary.

Typical Bugs

This section presents some TPM 1.2 application bugs in the hope that they may carry over to 2.0. We also list a few new anticipated error possibilities for TPM 2.0.

Authorization

TPM 2.0 plain text password authorization should be straightforward. However, HMAC authorization failures are common. The approach is the usual "divide and conquer." Trace the command (or response) parameter hash, the HMAC key, and the HMAC value. If the hash differs, the parameters that were used in the calculation were different from those sent to the TPM. If the HMAC keys differ, most likely the wrong password was used or the wrong entity was specified. If the HMAC value differs, either it was passed in differently or the salt or bind value was wrong.

Disabled Function

Perhaps the most common TPM 1.2 error we've seen is trying to use a disabled function. TPM 1.2 had disabled and deactivated flags, and TPM 2.0 has the corresponding hierarchy enabled.

TPM 2.0 has an additional HMAC error case: the entity may have been created in a way that disallows HMAC authorization. See the attributes `userWithAuth` and `adminWithPolicy` in Part 1 of the TPM 2.0 specification.

Missing Objects

A typical TPM 1.2 misunderstanding was that creating a key simply returned the key wrapped with the parent—that is, encrypted with the parent's key. It doesn't actually load the key into the TPM; a separate command is required to do that.

TPM 2.0 has an additional case. The TPM 1.2 SRK was inherently persistent. TPM 2.0 primary keys are transient and must be made persistent. Thus, primary keys may be missing after a reboot.

Finally, an object may have been loaded but is no longer there. You can break at the flush call (or add a `printf` to the flush call) and the failing command to determine when the object was flushed.

Similarly, a common error for TSS 1.2 was a *resource leak*—objects (or sessions) were loaded and not flushed, so the TPM eventually filled all its slots. Tracking the load and flush pairs should expose the missing flush, and this will also be true for TSS 2.0.

Wrong Type

In TPM 1.2, keys are basically signing keys or decryption/storage keys. Stepping through the function that performs the type check should uncover the error.

TPM 2.0 adds the concept of restricted keys, which introduce two new error cases. First a restricted key might may be used where only a nonrestricted key is permitted. Second, the user may try to change an algorithm, but restricted keys are created with an algorithm set that can't be changed at time of use.

There is also far more variability with respect to algorithms than TPM 1.2 has, where there were just a few padding schemes. Even PCRs have variable algorithms, which may lead to failures during an extend operation.

In addition, TPM 2.0 NV space has four types (ordinary, bit field, extend, and counter). This will undoubtedly lead to errors such as trying to write ordinary data into a bit-field index.

Bad Size

Asymmetric key operations are limited in the data size they can operate on. A common bug is trying to sign or decrypt (unseal) data that exceeds the capacity of the key and algorithm. For example, an RSA 2,048-bit key can operate on somewhat less than 256 bytes. The "somewhat" accounts for prepended data that includes padding and perhaps an object identifier (OID).

Policy

TPM 2.0 introduces policy authorization, which is very flexible but may prove hard to debug. Fortunately, the TPM itself has a debug aid, `TPM2_PolicyGetDigest`. Although you can't normally look inside the TPM or dump the contents of internal structures, this command is an exception and does exactly that.

Recall that an entity requiring authorization has a policy digest, which was precalculated and specified when the key was created. The value is computed by extending each policy statement. At policy-evaluation time, a policy session starts with a zero session digest. As policy commands are executed, the session digest is extended. If all goes well, the session digest eventually matches the policy digest, and the key is authorized for use.

However, in this chapter, the presupposition is that all isn't well. The digests don't match, and the authorization fails. We anticipate that the debug process will again consist of "divide and conquer." First determine which policy command failed, and then determine why.

Some policy commands, such as `TPM2_PolicySecret`, are straightforward, because they return an error immediately if authorization fails. Others—deferred authorizations like `TPM2_PolicyCommandCode`—are harder to debug because failure is only detected at time of use.

To determine which policy command failed, we suggest that you save the calculations used to calculate the policy hash. That is, the first hash value is all zeroes, there is an intermediate hash value for each policy statement, and there is a final value (the policy hash). Then, at policy-evaluation time, after each policy command, use `TPM2_PolicyGetDigest` to get the TPM's intermediate result. Compare the expected value (from the policy precalculation) to the actual value (from the TPM). The first miscompare isolates the bug to that policy statement.

One author's dream is that a premium TSS Feature API (FAPI) implementation will perform these steps. It has the policy, an XML document, so it can recalculate the intermediate hashes (or even cache them in the XML policy). It could implicitly send a `TPM2_PolicyGetDigest` after each policy evaluation step. This way, the evaluation could abort with an error message at the first failure rather than waiting until time of use, where it can only return a generic, "it failed, but I'm not sure where" message.

Determining why it failed strongly depends on the policy statement. Debugging the "why" is left as an exercise for you, the reader.

Summary

This chapter has described many of the best-known methods we've found for debugging TPM applications. Hopefully these will give you a good start in debugging, and you'll go on to discover even better techniques.