NCC Group Whitepaper

TPM Genie: Interposer Attacks Against the Trusted Platform Module Serial Bus

March 9, 2018 – Version 1.0

Prepared by

Jeremy Boone – Principal Security Consultant

Abstract

TPM Genie is a serial bus interposer which has been designed to aid in the security research of Trusted Platform Module hardware. The tool demonstrates that a man-in-the-middle on the TPM serial bus can undermine many of the stated purposes of the TPM such as measured boot, remote attestation, sealed storage, and the hardware random number generator.

Through TPM Genie we also reveal that an interposer device can trigger a variety of parsing errors in the host-side driver software that is responsible for communicating with the TPM. Here, we show that many TPM driver implementations are extremely fragile and are affected by numerous memory corruption vulnerabilities.

Combined, these flaws allow an attacker to circumvent many of the security assurances provided by a TPM, thus compromising the trusted boot process for a large number of TPM-enabled computing platforms.

Table of Contents

1 Introduction . . . 3

2 The Trusted Platform Module . . . 4

3 Attack Surface and Threat Model . . . 6

3.1 Serial Bus Interposing . . . 6

4 Interposer Attacks . . . 8

4.1 Attacks on PCR Extension . . . 8

4.2 Attacks on the Random Number Generator . . . 9

4.3 Attacks on the Host-Side Response Parsers . . . 9

5 Code Audit Results . . . 10

5.1 Chipset-Specific Vulnerabilities . . . 10

5.2 Command-Specific Vulnerabilities . . . 11

5.3 TPM v2.0 Vulnerabilities . . . 12

6 Gaps in the TCG Specification . . . 14

6.1 Locality . . . 14

6.2 Authorization Sessions . . . 14

6.3 Alice and Bob . . . 15

6.4 Turtles All the Way Down . . . 18

7 TPM Genie . . . 19

8 Conclusion . . . 20

8.1 Patch Availability . . . 20

8.2 Mitigations . . . 20

9 Future Work . . . 22

10 Thanks . . . 23

1 Introduction

On many computer systems the Trusted Platform Module (TPM) exists on a daughter card which connects to the computer's main board via a simple header. Through this header a serial bus is exposed, over which the host communicates with the TPM via a simple command protocol. Typically, the communication interface is the Low Pin Count (LPC) bus; however I2C and SPI-based TPMs are also commonplace for Internet of Things devices that don't use the LPC bus architecture. This exposed serial bus was the main focus of our research.

This paper, along with the accompanying Git repository, describes the process of designing and building an interposer device which acts as a man-in-the-middle on the TPM I2C bus. The tool, which we have named TPM Genie, has the ability to intercept and modify all traffic which is sent over the bus, and is intended to assist vulnerability research of discrete TPMs and the host-side drivers that interact with them.

Leveraging TPM Genie, we show how an interposer device can trivially spoof measurements stored in the Platform Configuration Register (PCR) banks. This act of spoofing PCRs serves to realize a variety of attacks against the core functions of a TPM that rely on PCR integrity, namely: measured boot, remote attestation, and sealed storage.

Another attack made possible by an interposer device is the ability to weaken the platform's hardware random number generator (RNG). This attack is conducted by altering TPM response payloads that contain the output of the TPM's internal hardware RNG. Consequently, an interposer could significantly impair many types of cryptographic operations on the host operating system.

Finally, TPM Genie reveals that an adversary can attack the host machine by malforming certain TPM response packets in order to trigger memory corruption in the host-side response parsing software. Our research revealed approximately 30 memory corruption vulnerabilities which were discovered and disclosed during our research efforts. These vulnerabilities affect the majority of the TPM driver ecosystem. Many platforms are at risk, including the Linux kernel and a variety of bootloaders such as U-Boot, coreboot, tboot, and Tianocore EDKII.

Attacks against the TPM serial bus require that the adversary has temporary physical access to the affected machine in order to implant an interposer device. Such an attack could be conducted in an "Evil Maid" scenario, by a rogue employee in a data center that houses TPM-enabled enterprise servers, or by a supply chain interdiction attack similar to those found in the NSA ANT catalogue of hardware implants.

Regrettably, physical attacks on TPM communication buses are rarely considered by those who attempt to design secure computing systems. Many defensive security features possessed by the TPM, which are intended to guarantee either the integrity and confidentiality of the serial bus data, are ineffective when confronted with an interposer device.

2 The Trusted Platform Module

The Trusted Platform Module is a cryptoprocessor whose specification was first developed in 2009 by the consortium known as the Trusted Computing Group (TCG). The specification has slowly evolved over the years, and at the time of writing this paper the two revisions seen most frequently in TPM deployments are v1.2 and v2.0. There are material differences between the two specifications, but these differences have little impact on the research presented herein.

The TCG claims that over 2 billion[1] TPMs have been deployed within a variety of computing platforms. The TPM is most commonly observed in x86 system architectures, such as enterprise servers, laptops and desktop computers. Although it is less common, TPMs can also be found in other smart devices such as enterprise printers and Internet of Things products. The U.S. Army and Department of Defense[2] require that every new PC has a TPM.

The host machine communicates with the TPM chip via a simple command-response protocol. The TCG specification defines more than 100 command ordinals. This command set is leveraged by the host computer to implement the various features provided by the TPM, such as random number generation, on-chip creation of cryptographic keys, as well as a number of other cryptographic primitives.

A primary purpose of a TPM is to detect when any stage of the boot chain has been tampered with. This is accomplished through the use of the TPM's Platform Configuration Registers (PCR). The PCR bank is a memory region that exists inside every TPM chip and serves as a dedicated secure storage location for a series of cryptographic hash digests. The TCG specification states that a TPM device should contain a bank of 24 PCRs, where each PCR is 160 bits in size - the width of a SHA-1 digest. On the other hand, the TPM v2.0 specification allows multiple PCR banks to exist, where every PCR index in the same bank must be extended with the same hash algorithm.

During the platform's boot sequence, each boot stage should compute a hash of the subsequent stage and the resultant hash digest should be stored in a PCR. The TCG EFI Platform Specification suggests that the PCR set should be organized as follows:

- PCR 0 – Core Root of Trust Measurement (CRTM), UEFI Firmware, SMI Handlers, …
- PCR 1 – SMBIOS, ACPI tables
- PCR 2 – Option ROM Code
- PCR 3 – Option ROM Configuration and Data
- PCR 4 – Initial Program Loader (IPL), Master Boot Record (MBR)
- PCR 5 – MBR Partition Table, …
- PCR 6 – Legacy BIOS, …
- PCR 7 – UEFI Secure Boot Policy, …
- PCR 8-24 – Platform or application specific

These hashes are often referred to as measurements. Using TCG terminology, a measurement is never "written" to a PCR, but is rather "extended" into a PCR. In fact, a PCR can be extended multiple times in order to track the integrity measurements associated with multiple critical boot parameters. Essentially, the previous digest is combined with the new digest, as summarized by the following notation:

```
digest_new := hashAlg( digest_old || data_new )
```

This "extend" terminology is carefully chosen by the TCG to imply that a PCR value can never be cleared.

[1] https://trustedcomputinggroup.org/protect-data-enhance-security/
[2] https://www.securityfocus.com/brief/265

In fact, the only way to reset a PCR to its original state of 20 null bytes (`0x00`) is to reset the TPM itself. This means that only the first stage bootloader should observe empty PCR banks.

PCR extension is an essential property of the TPM as it implies that a PCR value cannot be arbitrarily set by malicious code on the host machine. Malicious software can certainly attempt to extend new data into a PCR index, but this will only serve to compute a new (and incorrect) measurement for the PCR.

Of course, this suggests that the measurements should at some point be verified and appropriate action should be taken when the system observes that the PCR set does not match the expected values. The TCG doesn't define what "appropriate" means in these circumstances, as this decision is left up to the system implementers. However, one would expect that the boot sequence should be terminated.

As expected, in order for the Measured Boot process to work as intended, the first stage bootloader must be executed from write-protected memory. Otherwise an attacker is able to subvert the foundational code that anchors this chain of trust. This subversion would enable an attacker to effectively control the entire boot sequence including the ability to spoof all subsequent PCR extend operations.

In addition to the Measured Boot behavior described above, the TPM has two other important features related to the enablement of Trusted Computing: Remote Attestation and Sealed Storage. Once again, the properties of PCR extension are extremely useful here.

Remote Attestation enables an authorized remote party to detect software modifications that have been made to a computer. Attestation is typically used to prevent a user (or an attacker) from tampering with their computer in order to bypass protection measures. At a high level, an authorized remote party will request a Quote from the TPM, which is simply a digitally signed report of platform's PCR set.[3] The remote party is then expected to verify the quote using public key cryptography. Nonces should also be employed to prevent an adversary from replaying a previously validated quote.

A valid quote will demonstrate to the remote party that the correct measurements were made during the boot sequence and that the computer can be trusted because it has shown that it is running the anticipated software. At this point, the next steps are application specific. For example, the remote party could provide the computer with a DRM key, or a disk-decryption key, or simply provide the computer with a session token allowing it to interact with an exposed network service.

Sealed Storage is a mechanism by which a secret can be bound to a specific PCR set. The bound secret is stored in the TPM's non-volatile memory, and will be released only when the platform can demonstrate to the TPM that it has booted into the correct state. In other words, the secret can only be unsealed when the current PCR values are identical to those that were used to initially seal the secret. Typically, sealed storage is used to protect disk encryption keys or in DRM applications.

[3] Principles of Remote Attestation

3 Attack Surface and Threat Model

There are many types of Trusted Platform Modules. The most common is the *discrete* TPM, which exists on a separate semiconductor package. A TPM can also be *integrated* or embedded within another chip. *Firmware* TPMs are usually implemented within a secure execution environment,[4] such as ARM TrustZone. *Software* TPMs are effectively a TPM emulator that runs as a process or kernel module, such as ibmswtpm.[5] Finally, a *virtual* TPM is implemented in a hypervisor, perhaps the most common example of which is in Xen.[6]

The discrete Trusted Platform Module is often touted[7] as the most secure type of TPM, at least according to the Trusted Computing Group. Chipset manufacturers such as Infineon, ST Micro, Nuvoton, and Atmel have implemented a variety of tamper resistant features that defend their TPMs against fault injection, side channel, and invasive silicon attacks.

These defenses are not always perfect, as demonstrated by Christopher Tarnovsky at the BlackHat DC conference in 2010.[8] Tarnovsky executed an invasive silicon attack against an Infineon SLE66 chip and successfully extracted secrets from within the silicon package. The barrier for entry here is rather high, requiring a focussed ion beam (FIB), acid to decap the chip and expose the inner circuitry, and conductive needles to intercept the chip's internal data bus. Although Tarnovsky's research was extremely important to demonstrate that tamper resistance is not a panacea, it also makes clear that these types of attacks are extremely time consuming, prohibitively expensive, and are simply not portable.

It is indeed true that an attacker with physical access and unlimited resources will always win. Thus, the defender's task (in this case, the TPM chip vendors) is to deter the largest number of attackers by increasing the attack cost as much as possible.

Although TPM manufacturers have expended a considerable amount of engineering effort to harden their products against expensive invasive and semi-invasive silicon attacks, these manufacturers and the Trusted Computing Group have both entirely ignored the path of least resistance: inexpensive circuit-level attacks against the serial bus that the TPM uses to interact with the host machine.

Johannes Winter and Kurt Dietrich published two important papers on the topic of the TPM serial bus. The "Hijacker's Guide to the LPC Bus"[9] and "Hijacker's Guide to Communication Interfaces of the Trusted Platform Module"[10] papers were both essential reference materials for the research conducted by NCC Group. These papers make the important observation that although the TPM itself is tamper resistant, the serial bus that the TPM uses to communicate with the host operating system is comparatively insecure.

3.1 Serial Bus Interposing

So then, what circuit-level attacks can be leveraged against the Trusted Platform Module?

Although discrete TPMs are sometimes soldered directly onto the computer's main board, they are frequently present on a daughter card that connects to the main board via a header. This header exposes the serial bus to a rather straightforward means of tampering.

[4] TPM Mobile with Trusted Execution Environment for Comprehensive Mobile Device Security

[5] http://ibmswtpm.sourceforge.net/

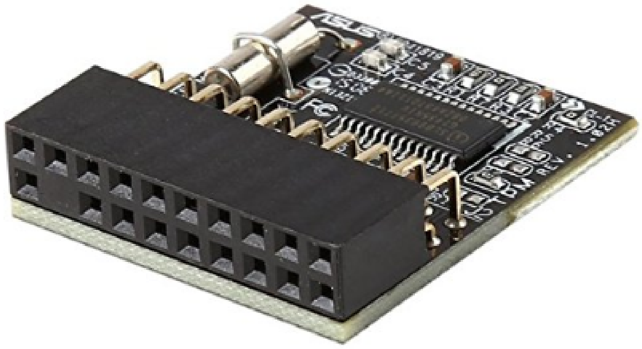
[6] https://wiki.xenproject.org/wiki/Virtual_Trusted_Platform_Module_(vTPM)

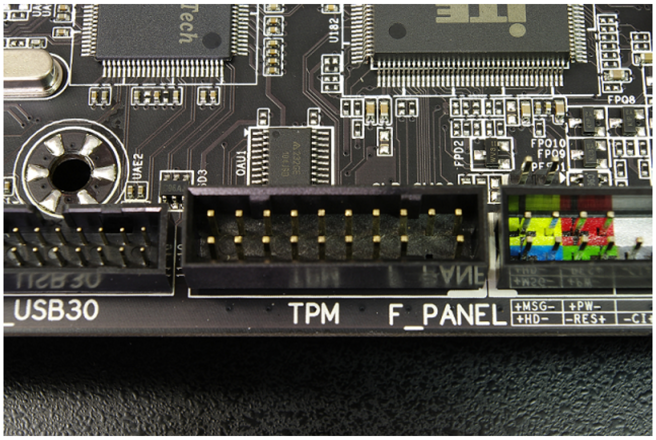
[7] Trusted Platform Module: A Brief Introduction

[8] http://www.smartcard.co.uk/articles/Whatthesiliconmanufacturerhasputtogetherletnomanputasunder/

[9] A Hijacker's Guide to the LPC Bus

[10] Hijacker's Guide to Communication Interfaces of the Trusted Platform Module





This act of tampering could be conducted by a simple shim-like hardware implant ("interposer"), possibly implemented using a low-cost microcontroller. The interposer would sit in the header between the host and the TPM daughter card, acting as a man-in-the-middle on the serial bus. The interposer would have the ability to capture all TPM command and response traffic that is transmitted on the bus.

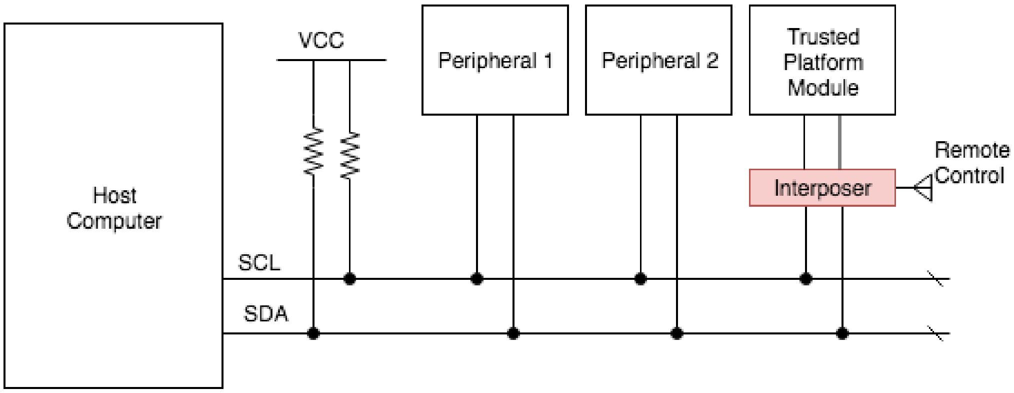


Figure 1: High level block diagram of Interposer on an I2C bus

Due to the convenient header, this type of attack is non-invasive and, unlike other types of circuit-level attacks, the adversary would not need to cut traces or desolder the TPM from the main board. One simply needs to unplug the TPM, insert the interposer into the header, and then connect the TPM to the interposer device. A practiced adversary could pull off such an attack in mere seconds.

In addition to passively sniffing traffic, an interposer could have the facility to modify the data that is sent over the serial bus. The following sections describe a variety of attacks made possible by a TPM interposer that is capable of modifying both command packets (host-to-TPM) and response packets (TPM-to-host).

4.1 Attacks on PCR Extension

As previously described, there are a number of TPM features that rely on PCR integrity, such as measured boot, remote attestation, and sealed storage. If an adversary wishes to defeat these protection measures, they would need to target the PCR Extend command ordinal with an interposer device.

All commands are prefixed with a 10-byte header that contains three fields: a 2-byte tag (whether the operation uses a session), a 4-byte length (the overall size of the packet) and a 4-byte ordinal (the command being executed). Shown below is the layout of an example `TPM_ORD_PcrExtend` command packet. For this particular command, the packet body contains two arguments: the PCR index that is being written (in this case index #1), and a 20-byte digest (a SHA-1 hash of the string "hello world").

```
+-----------------------------------+-----------------------------------------------------------------+
|              header               |                               body                              |
|-----------------------------------+-----------------------------------------------------------------|
|  tag  |   length    |   ordinal   |   index     |                 digest (20 bytes)                 |
+-------+-------------+-------------+-----------------------------------------------------------------+
  00 C1   00 00 00 22   00 00 00 14   00 00 00 01   2A AE 6C 35 C9 4F CF B4 15 DB E9 5F 40 8B 9C E9 ..
```

After the TPM has finished processing the command, it returns the following response packet. Note that the returned digest differs from the measurement that was submitted in the request packet. This is an intentional property of the Trusted Platform Module. Remember that new measurements are always extended, or mixed, with the previous measurement.

```
+-----------------------------------+-----------------------------------------------------------------+
|              header               |                                body                             |
|-----------------------------------+-----------------------------------------------------------------|
|  tag  |   length    |   status    |                          digest (20 bytes)                      |
+-------+-------------+-------------+-----------------------------------------------------------------+
  00 C4   00 00 00 1E   00 00 00 00     54 C5 28 F7 74 CE B1 F2 70 BA 53 49 FC AB C2 A1 BD 1F 10 D4
```

In order to attack PCR Extension, the interposer device would simply wait until the host transmits the PCR Extend command and then replace the 20-byte digest as it is transmitted over the serial bus. In this sense, the interposer could cause the TPM chip would receive a different digest than what was transmitted.

Such an attack would most effectively be conducted in combination with firmware tampering. For the sake of example, let's assume that an attacker wishes to alter the contents of flash memory to force the operating system to spawn the SSH daemon after bootup. Of course, this act of tampering would result in the computation of a different measurement, which should be recognized by a remote attestation service. To circumvent this fact, the interposer device would merely have to modify the PCR Extend packet payload, replacing the new measurement with the original pre-tamper value. This would fool the TPM into believing that the anticipated software was running on the platform.

4.2 Attacks on the Random Number Generator

The TPM's hardware random number generator is often relied upon when the host processor does not have its own strong RNG. The lack of an on-chip entropy source is a common deficiency in many low-cost embedded systems, as well as x86 platforms prior to the introduction of the RdRand instruction in 2013.

Modern Linux systems that use of the `tpm` kernel module will by default also use the `tpm_rng` module. This module ties into another kernel module named `hw_rng`, which is exposed to userspace via `/dev/hwrng`, such that all reads on this character device will result in the TPM chip providing the requested quantity of random bytes. Likewise, the OpenSSL TPM Engine leverages the TSS API to establish the TPM hardware RNG as engine's default source of randomness.

Shown below is the format of the simple `TPM_ORD_GetRandom` command packet. In this example, the host is requesting 16 random bytes from the TPM's hardware random number generator:

```
+-----------------------------------+-------------+
|              header               |    body     |
|-----------------------------------+-------------|
|  tag  |   length    |   ordinal   | rng_length  |
+-------+-------------+-------------+-------------+
  00 C1   00 00 00 0E   00 00 00 46   00 00 00 10
```

After the TPM has finished processing the command, it will reply with the following response packet. The response contains two fields: a length (`rng_length`) followed by a variable-size data payload (`rng_data`).

```
+-----------------------------------+-----------------------------------------------------------------+
|              header               |                              body                               |
|-----------------------------------+-------------+---------------------------------------------------|
|  tag  |   length    |   status    | rng_length  |                     rng_data                      |
+-------+-------------+-------------+-------------+---------------------------------------------------+
  00 C4   00 00 00 1E   00 00 00 00   00 00 00 10    DB 7E A1 71 BC A6 F2 C1 7F 26 CE A9 3E A1 75 78
```

In order to attack the hardware RNG, the interposer simply needs to replace the `rng_data` field in the packet body before forwarding the response payload to the host. A TPM interposer might therefore have full control over the platform's entropy source, which could allow it to undermine almost all cryptographic operations performed by the host.

4.3 Attacks on the Host-Side Response Parsers

The TPM specification is fairly complex and many of the 100+ commands contain variable-length payloads in both the request and response packet bodies. After spending some time analyzing the TCG specification, NCC Group investigated whether an interposer could induce other kinds of faults in the host software by malforming these response packets.

If the response parsers happened to be fragile, then a malformed response packet could potentially induce memory corruption. Such a vulnerability would be extremely valuable to an attacker, who might leverage the bug to compromise a host machine after it has booted into a fully measured and attested state.

The following sections describe the process and results of hunting for this class of TPM driver vulnerability.

5 Code Audit Results

After a short time-bounded code review effort, NCC Group discovered more than 30 memory corruption vulnerabilities in the host-side response parsing software. These issues affect a large cross section of the TPM driver ecosystem. The issues are summarized in the following table:

Software	Quantity	Patch Availability
Linux kernel	6	Patches available in 4.16 kernel
U-Boot	3	Patches available in master branch
Coreboot	1	Patches not yet available, but present in the public bug tracker
tboot	13	Patches not yet available
Tianocore EDKII	10	Patches not yet available

The subsequent sections perform an in-depth analysis of a handful of these vulnerabilities.

5.1 Chipset-Specific Vulnerabilities

Some of the discovered vulnerabilities happened to be chipset specific. These bugs affect only the lowest layer of the TPM driver and exist primarily within the vendor-specific chip interfaces.

For example, a common flaw among many kernel drivers was to blindly trust the length value contained within the TPM response header. The kernel drivers were architected such that, when reading a TPM response packet, they first read the 10-byte header, extract the 4-byte length field, and use that length to determine the size of the following read operation.

The TPM response header is structured as follows:

```c
struct tpm_output_header {
    __be16  tag;
    __be32  length;
    __be32  return_code;
} __packed;
```

Listing 1: `drivers/char/tpm/tpm.h`

Although many vendor drivers were affected by this flaw, the ST Micro driver implementation is suitable to illustrate this bug. The vulnerable code is shown below:

```c
static int st33zp24_recv(struct tpm_chip *chip, unsigned char *buf, size_t count)
{
    int size = 0;
    int expected;
    ...
    size = recv_data(chip, buf, TPM_HEADER_SIZE);
    ...
    expected = be32_to_cpu(*(__be32 *)(buf + 2));
    if (expected > count) {
        size = -EIO;
        goto out;
    }

    size += recv_data(chip, &buf[TPM_HEADER_SIZE], expected - TPM_HEADER_SIZE);
    ...
}
```

Listing 2: `drivers/char/tpm/st33zp24/st33zp24.c`

Above we see that recv_data is first called to retrieve the 10-byte header (TPM_HEADER_SIZE) from the TPM's data FIFO. Next, 4 bytes are extracted from the header and are then assigned to the expected variable. This value represents the overall length of the TPM response packet (sizeof(header)+sizeof(body)). Finally, expected is checked to ensure that it is not too large (>count), however, no checks are performed to ensure that it is not smaller than TPM_HEADER_SIZE. In this case, the operation expected-TPM_HEADER_SIZE would underflow, which would result in the driver reading an excessive quantity of bytes from the TPM, overflowing the destination buffer.

The patch is, of course, very straight forward:

```
-  if (expected > count) {
+  if (expected > count || expected < TPM_HEADER_SIZE) {
```

A serial bus interposer could freely manipulate the contents of the TPM response header, and this code is executed long before the response packet is unmarshalled and verified by the upper layers of the driver.

Curiously, this flawed coding pattern was observed to be nearly identical across all TPM chipset vendors, indicating that these manufacturers simply copy-and-pasted each other's driver code.

5.2 Command-Specific Vulnerabilities

Other vulnerabilities were observed to affect the upper layers of the TPM driver stack. This code is responsible for sending specific TPM commands and parsing the response payloads. In multiple instances, this parsing logic was fragile, and once again appeared to blindly trust size fields supplied in the variable-length response packet. Between the Linux kernel, tboot and Tianocore EDK2, a large number of commands were affected, such as:

- TPM_ORD_PcrRead
- TPM_ORD_GetRandom
- TPM_ORD_GetCapability
- TPM_ORD_Seal
- TPM_ORD_Unseal
- TPM_ORD_NV_ReadValue

The Linux kernel's TPM_ORD_GetRandom command handler will be used for the purposes of illustration. The tpm_get_random function is called by the Hardware RNG kernel module whenever the character device /dev/hwrng is accessed from userspace. For example, if the user requests 64 random bytes, the caller will pre-allocate a 64-byte buffer pointed at by out, and the argument max will be equal to 64.

```
int tpm_get_random(u32 chip_num, u8 *out, size_t max) {
    struct tpm_cmd_t tpm_cmd;
    u32 recd, num_bytes = min_t(u32, max, TPM_MAX_RNG_DATA);
    u8 *dest = out;
    ...
    tpm_cmd.header.in = tpm_getrandom_header;
    tpm_cmd.params.getrandom_in.num_bytes = cpu_to_be32(num_bytes);
    err = tpm_transmit_cmd( chip, &tpm_cmd, TPM_GETRANDOM_RESULT_SIZE + num_bytes );
    ...
    recd = be32_to_cpu(tpm_cmd.params.getrandom_out.rng_data_len);
    memcpy(dest, tpm_cmd.params.getrandom_out.rng_data, recd);
    ...
}
```

Listing 3: drivers/char/tpm/tpm-interface.c

The `tpm_get_random` function calls the `tpm_transmit_cmd`, which will send the `TPM_ORD_GetRandom` command to the TPM chip, and read the resulting response. The `tpm_cmd` structure is a union that contains both the request and the response data structures. The response body is contained within the field `tpm_cmd.params.getrandom_out`, and its declaration is shown below:

```
struct tpm_getrandom_out {
    __be32 rng_data_len;
    u8     rng_data[TPM_MAX_RNG_DATA];
} __packed;
```

Listing 4: `drivers/char/tpm/tpm.h`

In the event that the TPM's entropy pool has been exhausted, the TCG specification allows for a situation where the TPM may return fewer random bytes than were requested.[11]

> *Set randomBytesSize to the number of bytes available from the RNG. This number MAY be less than bytesRequested.*

This helps explain why the response structure for the `TPM_ORD_GetRandom` command is variable length, where `rng_data_len` indicates the number of bytes that follow. However, the specification does not explain whether a TPM could ever return more bytes than were requested by the caller. A serial bus interposer could easily manipulate the response packet, setting a very large value in the `rng_data_len` field.

Therefore, the unsafe code in `tpm_get_random` can be summarized in only two lines:

```
recd = be32_to_cpu(tpm_cmd.params.getrandom_out.rng_data_len);
memcpy(dest, tpm_cmd.params.getrandom_out.rng_data, recd);
```

This code trusts the `rng_data_len` field that was received from the TPM chip. If an interposer replaces this length with a value that is longer than `max`, then a trivial kernel overflow of the `dest` buffer is triggered. And once again, the patch is very simple:

```
   recd = be32_to_cpu(tpm_cmd.params.getrandom_out.rng_data_len);
+  recd = min_t(u32, recd, num_bytes);
   memcpy(dest, tpm_cmd.params.getrandom_out.rng_data, recd);
```

5.3 TPM v2.0 Vulnerabilities

Of particular interest was the `TPM2B` structure which is used widely throughout drivers that implement the TPM v2.0 specification. This structure is used to describe a generic outline of a variable-length packet.

Although other platforms made similar mistakes, for the purposes of illustration, we will highlight tboot's use of the `TPM2B` structure. The following code shows the packet format for responses to `TPM2_ORD_NV_READ` commands:

[11] 13.6 TPM_GetRandom, TPM Main Part 3 Commands, Specification Version 1.2

```
typedef struct {
    uint16 size;
    u8         buffer[1];
} TPM_2B;

typedef struct {
    u16    size;
    u8     buffer[MAX_NV_INDEX_SIZE];
} MAX_NV_BUFFER_2B;

typedef union {
    MAX_NV_BUFFER_2B    t;
    TPM2B               b;
} TPM2B_MAX_NV_BUFFER;
```

Listing 5: `tboot/include/tpm_20.h`

The intention of the TCG specification is that the `size` field will always match the length of the buffer. However, this assumption could be trivially broken by an interposer that carefully malforms the response payload. In many cases, if the size was greater than expected, the host-side response parser code would copy too many bytes into a fixed-length destination buffer, resulting in memory corruption. The following code block shows how tboot issues the NV Read command, and how it parses the variable-length response structure.

```
static bool tpm20_nv_read(struct tpm_if *ti, uint32_t locality,
                          uint32_t index, uint32_t offset,
                          uint8_t *data, uint32_t *data_size)
{
    tpm_nv_read_in read_in;
    tpm_nv_read_out read_out;
    ...
    if ( *data_size > MAX_NV_INDEX_SIZE )
        *data_size = MAX_NV_INDEX_SIZE;

    read_in.handle = index;
    /* NCC: initialize rest of read_in struct */
    ...
    ret = _tpm20_nv_read(locality, &read_in, &read_out);
    ...
    *data_size = read_out.data.t.size;
    if( *data_size > 0 )
        memcpy(data, &read_out.data.t.buffer[0], *data_size);
    ...
}
```

Listing 6: `tboot/common/tpm_20.c`

Notice that although the `data_size` is checked to ensure that it is not negative, there is no validation to ensure that it is not greater than the destination buffer pointed at by `data`. This results in a rather straightforward memory overrun which could be triggered by a man-in-the-middle on the serial bus.

These same types of mistakes were repeated throughout all of the audited code. It appears the driver developers are trusting the TCG protocol specification that has been implemented by the TPM hardware manufacturers. In general it can be concluded that there were significant oversights in the threat modelling exercises performed by these developers.

6 Gaps in the TCG Specification

The question arises: Did the Trusted Computing Group anticipate serial bus attacks and are there any provisions in the specification that can help mitigate the aforementioned vulnerabilities? The following sections survey the effectiveness of the various defensive features of the Trusted Platform Module.

6.1 Locality

Locality is a concept that was introduced in v1.2 of the TPM specification. Locality is essentially the TPM nomenclature for where a TPM command was issued from, designating which privilege level originated the command.

The main purpose of Locality is to define a simple access control policy, dictating which TPM commands can be performed when the host is executing in a particular security context. For example, the TPM Interface Specification states that certain PCRs can only be reset and extended when executing in specific Localities.

6.1.1 Weaknesses

Unfortunately, TPM Locality provides very little defense against a man-in-the-middle on the serial bus. In fact, an interposer device would never need to attempt to violate the Locality policy, because the interposer is ever-present on the bus, and can manipulate existing commands or issue entirely new commands whenever it pleases. In this sense, an interposer has no regard for the security context that the host-side software is executing in.

6.2 Authorization Sessions

The classic defense against packet tampering is the use of a hash-based message authentication code (HMAC). The computation of an HMAC involves the use of a cryptographic hash function and a secret key.

The TPM specification has a concept of authorization sessions wherein the host will append an HMAC to the command packet, and the TPM will append an HMAC to the response. Each side is then responsible for verifying the other's HMAC, which not only verifies the data integrity (with the hash digest), but also authenticates the payload (with the secret key). The HMAC key is based on the shared secret known as "AuthData" that was established when the TPM was initially configured with the `TPM_ORD_TakeOwnership` command.

In TPM v1.2 there are two types of authorization sessions: the object-independent authentication protocol (OIAP) and the object-specific authentication protocol (OSAP). In order to initiate a session, the host must issue either the `TPM_ORD_OIAP` or `TPM_ORD_OSAP` command to the TPM. The TPM will then respond with the session data (an AuthHandle and a nonce) which the host uses with the AuthData to compute a HMAC that is appended to each subsequent command.

The TPM v2.0 specification eliminates the notion of OIAP and OSAP session types. Instead, they are replaced with new session types allowing for the use of an HMAC as well as encryption of both command parameters and responses. Although the notation has changed, the behavior is more or less the same as the v1.2 specification. TPM 2.0 allows these authorization sessions to be stacked which provides the user with the flexibility needed to ensure the necessary integrity and confidentiality protection level for each transaction over the serial bus.

In order to prevent replay attacks, sessions use rolling nonces, wherein a new 20-byte nonce is generated by each party upon issuing or receiving a command. Nonces generated by the TPM are named "even" nonces, and those generated by the host are named "odd" nonces.

6.2.1 Weaknesses

Our research uncovered a number of specification flaws and host-side driver implementation errors that serve to weaken authorization sessions.

The primary problem with authorization sessions is that their use is optional for many commands (TPM Main Part 2 - Structures, section 5.9 "Auth Data Usage"). The TPM Command Ordinal table reveals that sessions are not mandatory for many delicate commands such as `TPM_ORD_GetRandom` and `TPM_ORD_PcrExtend`. Throughout all TPM interface implementations surveyed NCC Group, a general pattern was observed: When the TCG specification described an authorization session as optional, the developer opted not use it. In these situations, an interposer could freely manipulate these command and response payloads without detection by the host.

Other implementation weaknesses related to the use of nonces. When sealing and unsealing data, tboot was observed to initiate an authorization session and apply an HMAC to the command packet. However, it would never verify the HMAC in the response packet. Therefore, an interposer could manipulate the response payload, and tboot would never recognize that the modification had occurred.

Additionally, tboot intentionally skips the generation of the odd nonces, opting instead to use uninitialized stack memory for the nonce, and sending that in the command payload. There are multiple problems here with this approach. First, uninitialized stack memory is insufficiently random to serve as a nonce. If this stack memory happened to possess the same value on multiple iterations, then the nonce is rendered useless and does not defend against replay attacks. Second, this represents a serious memory disclosure vulnerability - tboot is essentially revealing 20-bytes of its memory space to an interposer device.

Similarly problematic was the Linux kernel, which uses the `TPM_ORD_GetRandom` command to generate a 20 byte nonce for the authorization session associated with the seal and unseal operations. We've shown how an interposer can take full control the hardware RNG output stream, so the conclusion is obvious: The interposer also controls the nonce value.

6.3 Alice and Bob

Through a significant investment in engineering effort, a highly motivated adversary could develop a TPM emulator to replace the comparatively naive interposer. An emulator would behave just like a legitimate TPM, except that it would be entirely attacker controlled and could exfiltrate all secrets that have been stored there by the user. Once this emulator has been implanted in the victim device, the host-side software may never realize that it is not communicating with a genuine TPM.

In general, it is difficult for the host-side driver software to verify whether it is communicating with a legitimate TPM chip. If the legitimacy of the TPM cannot be assured, then the integrity guarantees provided by the authorization sessions may be undermined. Two features in the TCG specification come close to solving this problem, but both fall short of the mark.

The first method is the TPM's vendor/device identifier (VIDID) register. The host can read this register to get a 4-byte value that identifies the TPM manufacturer and part number. But the VIDDID is no better than a serial number, and besides, an interposer can trivially spoof this value because it controls all peripheral register reads that are performed over the serial bus.

The second method relates to the public/private key pair known as the Endorsement Key (EK) which is embedded in a shielded location within the TPM chip. The host can request the public portion of this key by executing the command `TPM_ORD_Read_PubEK`. Alongisde the PubEK, a checksum is returned in the response payload. The host-side software is meant to validate this checksum to ensure the PubEK was not corrupted due to, say, an accidental glitch on the bus. But a simple checksum is not a suitable replacement for proper key verification that would detect malicious data manipulation by an interposer.

It is extremely important that the host verify the Endorsement Key because it is used by critical TPM operations, such as the important act of taking ownership of a TPM (`TPM_ORD_TakeOwnership`). Here, the owner is expected to use the PubEK to encrypt the owner's AuthData, which is the shared secret that is used to calculate the authorization session HMAC. Within the TPM chip, the paired PrivEK is used to decrypt the AuthData after the command has been received. This mechanism guarantees the confidentiality of the AuthData as it is transmitted over the serial bus.

6.3.1 Weaknesses

However, it is still possible to undermine this design. It is possible for a TPM interposer to provide its own public key to the host in response to a `TPM_ORD_Read_PubEK` command. If the host driver does not verify the attacker-controlled PubEK, then it would never realize it was communicating with a man-in-the-middle.

In this scenario, the host may unknowingly pass its AuthData to the interposer, and because the AuthData was encrypted using the attacker's PubEK, the attacker could also decrypt the secret. Consequently, the interposer could still establish itself as a man-in-the-middle on the serial bus, even when the host encrypts all of its command payloads and protect them with HMACs through the use of authorization sessions.

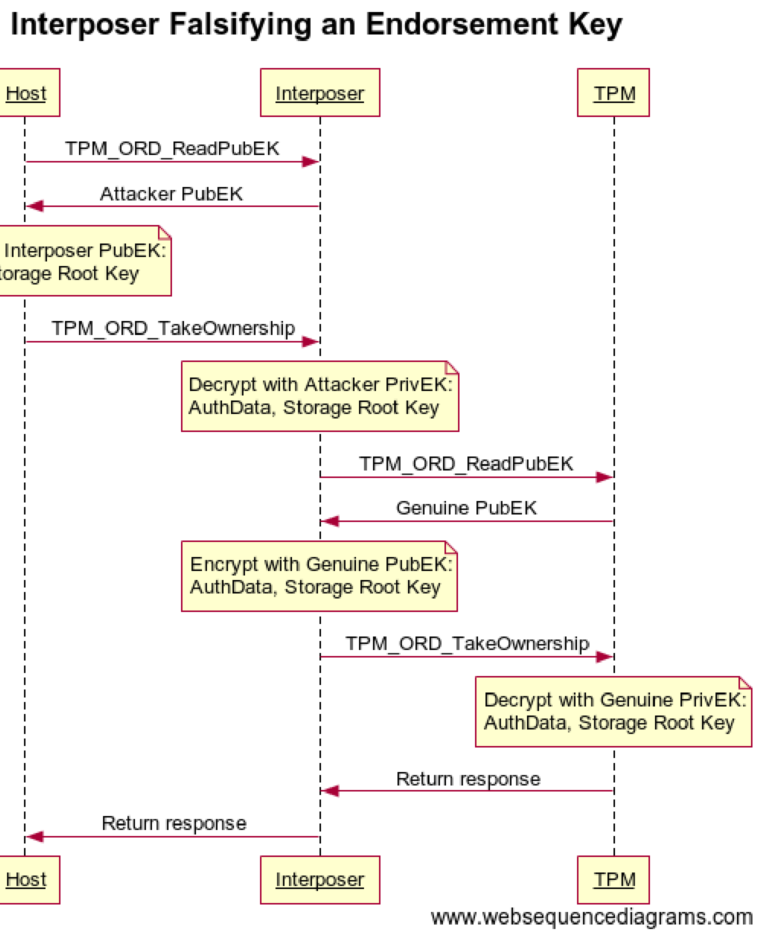

Interposer Falsifying an Endorsement Key
Host
Interposer
TPM
TPM_ORD_ReadPubEK
Attacker PubEK
Encrypt with Interposer PubEK: AuthData, Storage Root Key
TPM_ORD_TakeOwnership
Decrypt with Attacker PrivEK: AuthData, Storage Root Key
TPM_ORD_ReadPubEK
Genuine PubEK
Encrypt with Genuine PubEK: AuthData, Storage Root Key
TPM_ORD_TakeOwnership
Decrypt with Genuine PrivEK: AuthData, Storage Root Key
Return response
Return response
Host
Interposer
TPM
www.websequencediagrams.com

Figure 2: Interposer sequence diagram

The TCG specification only partially solves this problem. A robust solution for authenticating TPM chips should build upon robust Public Key Infrastructure (PKI) services. In fact, the TCG "Credential Profile" working group appears to have worked on this problem by defining a unified credential format that wraps the TPM's Endorsement Key. This Endorsement Certificate (EC) can be retrieved from TPM's NVRAM area (TPM Main Structures, Section 19.1.2) and can be used to verify the EK.

However, only some TPM chips are provisioned with an Endorsement Certification. For example, ST[12] and Infineon[13] both publish the EK on their website. Unfortunately, NCC Group was not able to identify any host-side driver implementations that use the EC to properly verify the EK prior to interacting with the TPM.

[12] ST Trusted Platform Module Endorsement Key certificates

[13] Infineon Optiga TPM certificates

6.4 Turtles All the Way Down

The following passage from the TPM specification (Design Principles, Section 8) is very revealing, and highlights the gaps in the threat modeling performed by the Trusted Computing Group.

> Outside the TPM, a reference monitor of some kind is responsible for protecting AuthData. AuthData should be regarded as a controlled data item (CDI) in the context of the security model governing the reference monitor. TCG expects this entity to preserve the interests of the platform Owner.

The TCG is encouraging the use of a *reference monitor* in order to protect the shared secret known as AuthData. This AuthData is presented by the owner in order to prove ownership of, or request authorization to use, an object within the TPM.

> A wide-range of objects use AuthData. It is used to establish platform ownership, key use restrictions, object migration and to apply access control to opaque objects protected by the TPM.

The AuthData is therefore a highly sensitive secret in the context of a TPM. Merely knowing the 20-byte AuthData value is sufficient to be granted access to many sensitive areas of the TPM:

> Neither the TPM, nor its objects (such as keys), contain access controls for its objects [..]. If a subject presents the AuthData, that subject is granted full use of the object based on the object's capabilities [..].

Given the sensitivity of the AuthData, it makes sense that it should be carefully protected. The properties of a reference monitor can be summarized by the rather simple NEAT acronym:

- **N**on by-passable
- **E**valuable (verifiable)
- **A**lways invoked (always on)
- **T**amper-proof (self-protecting)

Curiously, a *reference monitor* sounds a lot like a Trusted Platform Module. This begs the question: If a second TPM is used to protect the AuthData associated with the first TPM, then should a third TPM be used to protect the second TPM's AuthData? However, this doesn't scale.

7 TPM Genie

On its own, code review is perhaps insufficient evidence as to the validity of the vulnerabilities described in this paper. Although the response parsing memory corruption bugs may be apparent through code review, vulnerabilities that affect the hardware RNG and PCR extension are more subtle and require concrete evidence. We therefore decided to design and build an actual interposer device to demonstrate the legitimacy of these attacks. This tool has been named TPM Genie.

TPM Genie is able to manipulate TPM command and response packets as they are submitted over the I2C serial bus. Although the LPC bus is the most common bus architecture used by the TPM, the I2C protocol was selected in our design because it simplified the interposer hardware and firmware and thus shortened the development effort. Support for the LPC and SPI protocols is planned as a future undertaking.

As a man-in-the-middle, TPM Genie is able to actively attack PCR Extend operations executed by the host, yielding the ability to undermine most of the stated purposes of a TPM: measured boot, remote attestation, and sealed storage. Our interposer makes it is possible to spoof PCR measurements by replacing the digest contained within the PCR Extend command as it is transmitted over the serial bus.

Additionally, TPM Genie can weaken the Linux hardware random number generator. We demonstrate that TPM Genie has full control over the random bytes returned by the TPM, and thus is able to impair many types of cryptographic operations on the host.

And finally, by subtly altering the contents of a certain TPM response packet, we were able to trigger one of the previously discussed parsing vulnerabilities. As a proof of concept, a single vulnerability which affects the `TPM_ORD_GetRandom` command handler was selected, and TPM Genie is able to crash the Linux kernel by manipulating a length field in the response packet.

We have decided to make TPM Genie an open source project. The full hardware design plan, firmware implementation, usage instructions and proof-of-concept exploits are available in the `nccgroup/TPMGenie`[14] repository on GitHub.

[14] https://github.com/nccgroup/TPMGenie

8 Conclusion

Due to time constraints, we were unable to exhaustively audit every single TPM driver in every single operating system or bootloader. In fact, our research was limited to only open source TPM drivers. Based on the high density of parsing vulnerabilities discovered by our brief code audit, it is our belief that additional bugs are still undiscovered and could pose further risk for TPM users.

Regrettably, the Trusted Computing Group specification does not account for attacks from local or physical vectors. This is unfortunate because in many ways, the TPM is meant to defend a platform from physical attacks (ex: by an adversary that tampers with firmware that resides flash memory). And yet, the TPM's data path itself is not resilient to trivial physical attacks, such as those against the serial bus.

TPM Genie neatly demonstrates that a TPM cannot be relied upon to make the host system robust against cheap physical attacks. In other words, simply adding one secure component (a TPM) to an insecure system does not make the overall system secure. We hope that by releasing these practical tools, the myth of TPM security will be dispelled.

Ultimately, if physical vectors are part of your threat model, then using a discrete TPM can actually increase the attack surface of your device, and lower an adversaries cost of conducting a successful attack. In order to help mitigate circuit level attacks of the TPM's serial bus, certain design principles should be followed. These best practices are described in the following sections.

8.1 Patch Availability

TPM users who are concerned with physical attack vectors should apply patches for the host software vulnerabilities discussed in this paper. Unfortunately, at the time of writing, none of these patches are readily available.

The Linux kernel memory corruption patches will be released in version 4.16 and also will be back ported to a number of stable branches. However, the patches that improve the kernel's use of authorization sessions are still a work in progress.

The U-Boot vulnerabilities have been patched, and at the time of writing, can be found in the master branch.

The Coreboot vulnerability report is still untouched and presently sits idle the public bug tracker.

Intel has not yet patched the vulnerabilities that affect tboot and Tianocore EDK2. In fact, Intel's Product Security Incident Response Team have decided that the vulnerabilities do not pose sufficient risk to require an embargo on their public disclosure. Intel PSIRT has given NCC Group permission to disclose the vulnerabilities.

8.2 Mitigations

In order for TPMs to defend against interposer attacks, the Trusted Computing Group should revise the Trusted Platform Module specification. The use of authorization sessions should be made mandatory for all TPM command ordinals. Likewise, the TCG specification needs to make Endorsement Key Certificates a mandatory feature for all TPM hardware, and manufacturers should publish the certificates that are embedded in the chip. And finally, the host-side drivers should be revised to validate the EKCert before communicating with the TPM.

In the meantime, systems engineers should account for serial bus interposers when designing TPM-enabled devices. First and foremost, socketed discrete TPMs should be avoided entirely because they enable an adversary to conduct an interposer attack in mere minutes once physical access is obtained.

When designing circuit board layouts, hardware engineers should ensure that the TPM is soldered onto the main board, covered with a RF shield, and underfilled with epoxy resin. These steps will raise the cost of the attack, forcing the adversary to employ more invasive circuit modification techniques to remove the TPM and install the interposer.

Ideally, hardware engineers should also consider burying the serial bus in an inner layer of the printed circuit board. It can be difficult to hide the traces completely if there is a need for pull-up or termination resistors, such as is required for I2C and LPC. In such cases, system designers should seek host processors or TPM components that have internal resistors for this purpose.

Alternatively, the most resilient solution would be to never expose the serial bus in the first place. The TPM could be embedded within the semiconductor package of the main processor. This raises the cost for an adversary, by requiring more expensive invasive silicon attacks.

9 Future Work

Although TPM Genie is a useful tool that serves as a proof-of-concept for demonstrating the legitimacy of TPM interposer attacks, much more work is needed if we wish to realize mature attacks on other platforms.

As a priority, NCC Group would like to expand TPM Genie's support for the TPM hardware ecosystem. Namely, we would like to:

- Test TPM Genie against additional TPM chips and vendors
- Add explicit support for version 2.0 of the TPM specification
- Add support for both SPI and LPC buses

Due to the higher data transmission rates of SPI and LPC, and the more complex v2.0 feature set, it is possible that the current Arduino-based design will be insufficient to interpose traffic in real time. In this case, it may be necessary to re-implement TPM Genie using an FPGA or faster microcontroller.

Although breadboards are excellent for prototype development, an actual interposer would need to masquerade as a legitimate computer component in order to make for a convincing hardware implant. We would like to miniaturize TPM Genie and have it professionally silk-screened to mimic the brand markings of a common TPM chipset vendor.

Additionally, a real hardware implant would benefit from some sort of remote access capability so that the adversary could control the interposer remotely. We would like to investigate the possibility of adding a low-cost, long-range radio module for remote control purposes.

We would also like to add better emulation of the `TPM_ORD_Read_PubEK` and `TPM_ORD_TakeOwnership` commands. This would allow TPM Genie to steal the host's AuthData, and then perform active man-in-the-middle attacks against commands that use an authorization session.

Finally, we feel that it is necessary to audit the remainder of the TPM driver ecosystem. Our initial audit was time boxed, and as such, not all TPM-enabled platforms could be surveyed, and in particular, many closed-source implementations were not investigated. We believe that we've only scratched the surface with respect to the sorts of vulnerabilities that could be triggered by TPM Genie.

10 Thanks

I would like to thank my colleagues at NCC Group who helped me navigate the process of designing and building TPM Genie, as well as disclosing the vulnerabilities. They are Rob Wood, Ian Robertson, Sultan Qasim Khan, Jason Meltzer and Jon Szymaniak.

I would also like to thank James Bottomley and Jarkko Sakkinen for their assistance with patching the Linux kernel's TPM drivers.