Hacking Toshiba Laptops
Or how to mess up your firmware security

REcon Brussels 2018
whois

Serge Bazanski
Freelancer in devops & (hardware) security.
Twitter: @q3k
IRC: q3k @ freenode.net

Michał Kowalczyk
Vice-captain @ Dragon Sector
Researcher @ Invisible Things Lab
Reverse engineer, amateur cryptanalyst
Twitter: @dsredford
IRC: Redford @ freenode.net
Toshiba Portégé R100

Intel Pentium M 1 GHz
256MB RAM
But there’s a catch...
Quite the catch, actually.

CMOS clear jumper? None to be found.

Yank out the battery? Password still there.

Take a door key and pass it over the pins of things that look like flash chips hopefully causing a checksum failure and resetting the password?

Nice try. No luck, though.
A-ha!

PC Serial No. = 0000000000
Challenge Code = 2HPV3-6EEED-UCWBK-VJ6LC-QUPGY
Response Code =
BIOS analysis
How to get the BIOS code?

Physical memory? Not with a locked-down laptop.

Dump of the flash chip? Ugh.

Unpack some updates? Let’s see.
Unpacking the updates

https://support.toshiba.com/
Decompression

Unknown format

Default unpacker is a 16-bit EXE

There's an alternative one, 32-bit!
Decompression

BuIsFileCompressed
BuGetFileSize
BuDecodeFile
Decompression

Just ~50 lines of C!

... 

BuIsFileCompressed(compressed, &is_compressed);
if (is_compressed) {
    BuDecodeFile(compressed, fsize, decompressed);
}

...
Dumping the BIOS flash
Where to start looking
Chip Safari

RAM

Flash

Google it
Interfacing to flash chips

**In-circuit:** test pads or protocol that permits multi-master access

**Out-of-circuit (??):** desolder, attach to breakout/clip, use main communication interface
Custom breakout board

**KiCAD** (or $whatever, really) PCB design.

**Thermal transfer** for DIY PCB manufacturing.

Hot air gun to desolder, soldering station to re-solder.
Hackerspace

25eur p/m + BYOB
When your etching rig breaks but you really need that PCB made today.
Setup

- PC
- UART (2 wires) to FPGA Devboard
- A/A Mux (~30 wires) to Flash
- Breakout Board
- Gimme block X
- X * 1024
- Data word
- X * 1024 + 4
- Data word
- X * 1024 + ...
- Data word

1kB of data
Setup issues

- Breakout Board
- PC
- UART (2 wires)
- FPGA Devboard
- A/A Mux (~30 wires)
- Flash
- Breakout Board

Gimme block X

X * 1024

Data word

X * 1024 + 4

Data word

X * 1024 + ...

Data word

1kB of data

Forgot checksums

Mixed them up.
But why the FPGA?

Using an FPGA was unnecessary - just needed a bunch of I/O.

Comparatively difficult to develop for. And to debug.

Should’ve gone for a uC with a bunch of I/O or with a multiplexer.

But at least now we know \_(ツ)_/\.
q3k8amnesia $ strings herpderp | grep -i respo

Response Code =
Response Code =
Response Code =
Response Code =
Response Code =

q3k8amnesia $ stat herpderp

File: 'herpderp' Blocks: 6328 Links: 1
Size: 4266597 Inode: 4198912
Device: fd03h/64771d Uid: ( 1000/ q3k8amnesia)
Access: 2014-01-05 17:37:03.512176339 +0100
Access: 2014-01-05 17:44:00.657188279 +0100
Modify: 2014-01-05 17:44:00.657188279 +0100
Change: 2014-01-05 17:44:00.657188279 +0100
Birth: -

q3k8amnesia $
BIOS code analysis
How to start?

CPU mode?

Entry point?

Memory map?
“A hardware reset sets each processor’s registers to a known state and places the processor in real-address mode.”

Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 3
<table>
<thead>
<tr>
<th>Register</th>
<th>Power up</th>
<th>Reset</th>
<th>INIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFLAGS¹</td>
<td>000000002H</td>
<td>000000002H</td>
<td>000000002H</td>
</tr>
<tr>
<td>EIP</td>
<td><strong>0000FFFF0H</strong></td>
<td>0000FFFF0H</td>
<td>0000FFFF0H</td>
</tr>
<tr>
<td>CRO</td>
<td>600000010H²</td>
<td>600000010H²</td>
<td>600000010H²</td>
</tr>
<tr>
<td>CR2, CR3, CR4</td>
<td>000000000H</td>
<td>000000000H</td>
<td>000000000H</td>
</tr>
<tr>
<td>CS</td>
<td>Selector = F000H</td>
<td>Selector = F000H</td>
<td>Selector = F000H</td>
</tr>
<tr>
<td></td>
<td>Base = <strong>FFFF0000H</strong></td>
<td>Base = FFFF0000H</td>
<td>Base = FFFF0000H</td>
</tr>
<tr>
<td></td>
<td>Limit = FFFFH</td>
<td>Limit = FFFFH</td>
<td>Limit = FFFFH</td>
</tr>
<tr>
<td>SS, DS, ES, FS, GS</td>
<td>Selector = 0000H</td>
<td>Selector = 0000H</td>
<td>Selector = 0000H</td>
</tr>
<tr>
<td></td>
<td>Base = 00000000H</td>
<td>Base = 00000000H</td>
<td>Base = 00000000H</td>
</tr>
<tr>
<td></td>
<td>Limit = FFFFH</td>
<td>Limit = FFFFH</td>
<td>Limit = FFFFH</td>
</tr>
<tr>
<td>EDX</td>
<td>000n06xxH³</td>
<td>000n06xxH³</td>
<td>000n06xxH³</td>
</tr>
<tr>
<td>EAX</td>
<td>0⁴</td>
<td>0⁴</td>
<td>0⁴</td>
</tr>
<tr>
<td>EBX, ECX, ESI, EDI, EBP, ESP</td>
<td>000000000H</td>
<td>000000000H</td>
<td>000000000H</td>
</tr>
<tr>
<td>STO through ST7⁵</td>
<td>+0.0</td>
<td>+0.0</td>
<td>FINIT/FNINIT: Unchanged</td>
</tr>
</tbody>
</table>

¹ EFLAGS is initialized to 000000002H, reflecting the default settings.

² CRO is initialized to 600000010H, which is the default for the 68020 processor.

³ EDX is initialized to 000n06xxH, representing a default value.

⁴ EAX is initialized to 0⁴, indicating a default setting.

⁵ STO through ST7 are initialized to +0.0, indicating the default mathematical state.
CPU start

We start at the address:

$$\text{CS:EIP} = \text{CS.Base} + \text{EIP} = 0xFFFFFFFF0$$

Real Mode $\Rightarrow$ physical address. A20 enabled.

So, what’s there?
Memory mapping

Northbridge: Intel Odem MCH-M

No info about that region ⇒ let’s check the southbridge
### Memory mapping

**Southbridge: Intel ICH4-M**

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFF8 0000–FFFF FFFFh</td>
<td>FWH</td>
</tr>
<tr>
<td>FFB8 0000–FFBF FFFFh</td>
<td>Always enabled. The top two 64 KB blocks of this range can be swapped, as described in Section 7.4.1.</td>
</tr>
</tbody>
</table>

FWH = Firmware Hub = BIOS flash

Out dump has exactly 0x80000 bytes!
### FWH_F8_EN — R/W. Enables decoding two 512 KB FWH memory ranges, and one 128KB memory range.

- **0** = Disable
- **1** = Enable the following ranges for the FWH:
  - FFF80000h–FFFFFFFh
  - FFB80000h–FFBFFFFFh
  - 000E0000h–000FFFFFh

### FWH_F0_EN — R/W. Enables decoding two 512 KB FWH memory ranges.

- **0** = Disable.
- **1** = Enable the following ranges for the FWH:
  - FFF00000h–FFFF7FFFFFh
  - FFB00000h–FFB7FFFFFh

### FWH_E8_EN — R/W. Enables decoding two 512 KB FWH memory ranges.

- **0** = Disable.
- **1** = Enable the following ranges for the FWH:
  - FFE80000h–FFEFFFFFh
  - FFA80000h–FFAFFFFFFFh
Entry point

FFFFFFFFFF0:    jmp far FC00:3FA0
000FFFFFFA0:    jmp far FC00:00A2
000FC0A2:        cli
000FC0A3:        cld
000FC0A4:        mov al, 2
000FC0A6:        out 92h, al ; Enable A20

...
BIOS RE: Initialization

No stack! (and also no RAM)

16-bit Protected Mode + Unreal Mode

Checksums

RAM initialization

Self-copying into RAM
BIOS RE: Initialization

16-bit Protected Mode → segments!

We have to find and parse GDT

Only then we can analyze the code
BIOS RE: The password check

```
prompt: ; {align:79}[goto col:0]
mov    si, offset line_with_spaces
call   print_string_at
lea    bx, [di+1Fh]
call   zero_at_bx     ; count = [di+0Eh]
push   cs
push   0ABDCh
push   2Ch ; ','
call   50h:5321h     ; seg3:5321
test   [di+screen_struct.ask_flags], 2
jz     short ask_for_pwd

push   cs     ; ask for response
call   near ptr print_pc_serial
push   cs
call   near ptr print_challenge
mov    si, offset a_response_code ; " Response Code = "
mov    word ptr [di+screen_struct.max_read], 25
jmp    short loc_ABFC
```

```
ask_for_pwd: ; Password =
mov    si, offset a_pwd_prompt
mov    word ptr [di+screen_struct.max_read], 50
```
Everything eventually lands up in one function
\[ f(\text{in\_buf}) \rightarrow \text{out\_buf} \]

After long analysis: all bytes are sent to I/O ports \textbf{62h} and \textbf{66h}
BIOS RE: The password check

From the southbridge manual:

<table>
<thead>
<tr>
<th>Hex</th>
<th>Description</th>
<th>Description</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>60h</td>
<td>Microcontroller</td>
<td>Microcontroller</td>
<td>Forwarded to LPC</td>
</tr>
<tr>
<td>61h</td>
<td>NMI Controller</td>
<td>NMI Controller</td>
<td>CPU I/F</td>
</tr>
<tr>
<td>62h</td>
<td>Microcontroller</td>
<td>Microcontroller</td>
<td>Forwarded to LPC</td>
</tr>
<tr>
<td>63h</td>
<td>NMI Controller</td>
<td>NMI Controller</td>
<td>CPU I/F</td>
</tr>
<tr>
<td>64h</td>
<td>Microcontroller</td>
<td>Microcontroller</td>
<td>Forwarded to LPC</td>
</tr>
<tr>
<td>65h</td>
<td>NMI Controller</td>
<td>NMI Controller</td>
<td>CPU I/F</td>
</tr>
<tr>
<td>66h</td>
<td>Microcontroller</td>
<td>Microcontroller</td>
<td>Forwarded to LPC</td>
</tr>
</tbody>
</table>

Table 6-2. Fixed I/O Ranges Decoded by Intel ICH4
“Microcontroller”???
EC/KBC

CPU
Intel Pentium M

Northbridge
Intel Odem
MCH-M

GPU
Trident XP4

Southbridge
Intel ICH4-M

RAM

Audio

BIOS

HDD

Ethernet

Battery

Keyboard

Touchpad

EC/KBC
Renesas
M306K9FC
LRP

PSC

LPC
How to obtain the code?

Updates!
EC: Dump

No updates available

BIOS changelog: nothing about the EC

Maybe a similar laptop model?

Portégé S100!
EC: Updates

Inside: 3 update blobs (different versions)
EC: Update installer

Uses ports 62h & 66h

Sends the 1st part (~2,5KB)

Sends the 2nd part (~100KB)
It’s decoded inside EC - no code available :(

Let’s try some analysis!
EC: Update blob - analysis

High entropy $\Rightarrow$ encryption or compression

No regularities in trigrams $\Rightarrow$ encryption

Size always divisible by 8 $\Rightarrow$ encryption

Longest repeated substring is short $\Rightarrow$ if encryption, then not ECB
EC: Update blob - analysis

Looks like a dead-end...
Serge, could you please desolder something again...?
One last breakout later...
Let’s dump this thing.
EC: Programming Protocol

Programmer

SCLK
RXD
TXD
Busy

M16C
EC: Programming Protocol
EC: Programming Protocol

Programmer

M16C

Flash Page X?

Flash Page X
ID code check function

The function is used in standard serial I/O mode. If the flash memory is not blank, the ID code sent from serial burner is compared with that inside flash memory to check the agreement. If the ID codes do not match, the commands from serial burner are not accepted. Each ID code consists of 8-bit data, the areas of which, beginning from the 1st byte, are 0FFDF16, 0FFFE316, 0FFFEB16, 0FFFFF16, 0FFFF316, 0FFFF716, 0FFFFFF16. Write a program with the ID code at these addresses to the flash memory.
EC: Programming Protocol

ID Check (K0...K6)

Status?

Status (Unlocked/locked)

Flash Page X?

Flash Page X
Side channel attacks?

Fault injection?
Not so fast.
Software level ‘side’ channels

An PIN unlock request does not result in any immediate success/failure transmission, but...

Hmm.
EC: M16C bootloader bug

Let’s run some quick tests.
EC: M16C bootloader bug
EC: M16C bootloader bug
Well that’s not good.
EC: M16C Bootloader bug

Programmer

Response time measurement

M16C

ID Check - 00 FF FF FF FF FF FF FF
ID Check - 01 FF FF FF FF FF FF FF
ID Check - 02 FF FF FF FF FF FF FF
ID Check - .. FF FF FF FF FF FF FF
ID Check - FE FF FF FF FF FF FF
ID Check - FF FF FF FF FF FF FF

Average time + 3μs

Average time + 3μs

Response time measurement

M16C

ID Check - 00 FF FF FF FF FF FF FF
ID Check - 01 FF FF FF FF FF FF FF
ID Check - 02 FF FF FF FF FF FF FF
ID Check - .. FF FF FF FF FF FF FF
ID Check - FE FF FF FF FF FF FF
ID Check - FF FF FF FF FF FF FF

Average time + 3μs
EC: M16C Bootloader bug

Programmer

Response time measurement

M16C

ID Check - 00 FF FF FF FF FF FF

ID Check - 01 FF FF FF FF FF FF

ID Check - 02 FF FF FF FF FF FF

ID Check - .. FF FF FF FF FF FF

ID Check - FE FF FF FF FF FF

ID Check - FF FF FF FF FF FF

Average time + 3μs

Ergo, the first byte of the key is 02.
Thus, we can enumerate all bytes of the key one by one, using the timing difference for each correct byte to reduce our search to just 0x100*7 checks.

And we get the key.
EC: M16C Bootloader bug
EC: M16C Bootloader bug
EC: M16C Bootloader bug

PoC || GTFO

https://github.com/q3k/m16c-interface/

(note: doesn’t work for all M16Cs... yet)
EC: RE

Code (~700 functions)

R/O data

Crypto

Bootloader
EC: RE

Much simpler code than in the BIOS
No strings
We’re looking for LPC communication and BIOS-call table
Finding the table is easy

~100 different BIOS<->EC calls

We know the numbers of the interesting calls ⇒ let’s analyze the handlers!

Sounds easy...?
EC: RE of the handlers

Manual context-switching

No common call convention

Handlers aren’t split into functions

Jumps to the middle of other functions
Password check: BIOS

\[
\text{out\_buf} = \text{call\_EC}(\text{func}=0x24, \\
\quad \text{in\_buf}=\text{MD5(input)}[:8] + \text{pwd\_type})
\]

\[
\text{out\_buf}[0] == 0 \Rightarrow \text{success}
\]
Password check: EC

Let’s look at the handler on the EC side...

...6 levels down the call hierarchy:

BMGEU/C  p6_4,  p6
BSET     pd6_4,  pd6
JSR.W    set_p6_5
JSR.W    clear_p6_5

I/O on pins 40 & 41
Password check: EC

Oh, come on... :(
Password check: EC

This time it’s only an EEPROM :) 

EC reads one block, decrypts it and compares with the received MD5
Challenge/Response

Screw it, we’re looking for a universal attack

Let’s look at the challenge/response!
Challenge: BIOS

\[
\text{out\_buf} = \text{call\_EC}(\text{func}=0x1A, \\
\quad \text{in\_buf}=\text{rdtsc}() + \text{MD5}(\text{pc\_serial})[:8])
\]

\[
\text{challenge} = \text{bytes\_to\_string}(\text{out\_buf})
\]
Challenge: EC

1. RDTSC
2. Entropy pool
3. 7 random bytes
4. Checksum
5. ENC
6. PC_SERIAL_MD5
7. DEC
8. CHALL 1
9. CHALL 2
Response: BIOS

```python
out_buf = call_EC(func=0x1B, in_buf=string_to_bytes(user_input))
```

`out_buf[0]` ⇒ success/fail
Verify checksum

\[ = 0? \]
EC: Encryption

ENC? DEC?
EC: Encryption

A custom 64-bit block cipher

INPUT (8B) → ENC → OUTPUT (8B)

KEY A (256B) → ENC → KEY B (128B)
Challenge/Response

We just need to rewrite it in Python and ...
EC: Update system

Let’s decrypt the updates!
EC: Update system

Uh, symmetric signatures?

We can generate our own!
So, how's it like on their newer laptops?
If it ain’t broke, don’t fix it!
(that applies to keys, too)
Unlocking any (business) laptop.

Permanent rootkit in the EC.

We can attack the host from the EC.
Rootkit in EC?

DMA to the host via LPC (not supported by this particular EC).

Keylogging & storage.

USB-Rubber-Ducky-like (key/mouse injection).

BIOS exploitation via the internal API.
Toshiba is working on a temporary BIOS update that can be used to prevent the security issue that has been raised and expects to release this update on its website within the next 2 weeks.

Toshiba plans to start the release of a permanent fix for some models from January, 2018 and will complete the releases of permanent fix for all applicable models by the end of March 2018.
Questions?