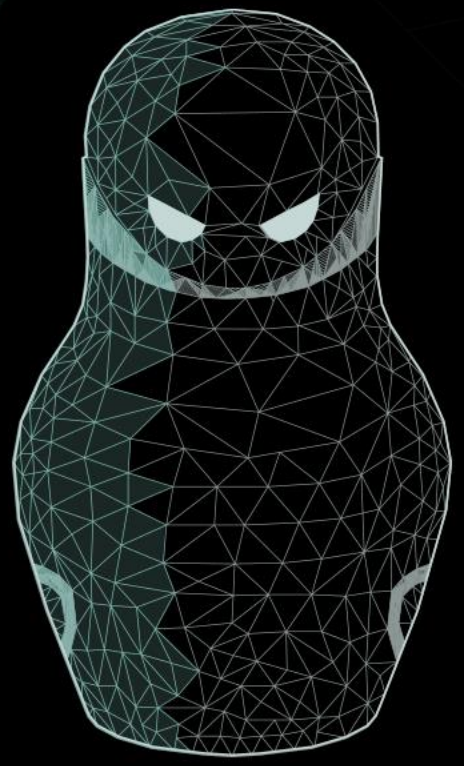


ZERO NIGHTS 2015

# Fix it yourself

## detecting and fixing UEFI firmware vulnerabilities without access to it's source code

Nikolaj Schlej
Software Engineer BIOS, congatec AG
schlej@live.de, @NikolajSchlej

26.11.2015

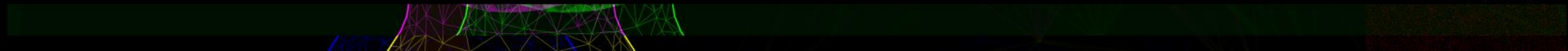
# About me



- a.k.a. CodeRush
- tinkering with UEFI since 2011
- came to InfoSec from BIOS modding community
- author of UEFITool
- wrote master thesis on CoMs UEFI security
- work for congatec AG as BIOS engineer

congatec





# Agenda



- Brief Intro to (U)EFI
- What an Attacker Can Do?
- Attack Vectors
- Protections
- An Average System
- Test Tools
- Test Results
- What Now?
- Prepare to Dig Deep
- Fix It Yourself
- Conclusion
- Q&A

# Brief Intro to (U)EFI



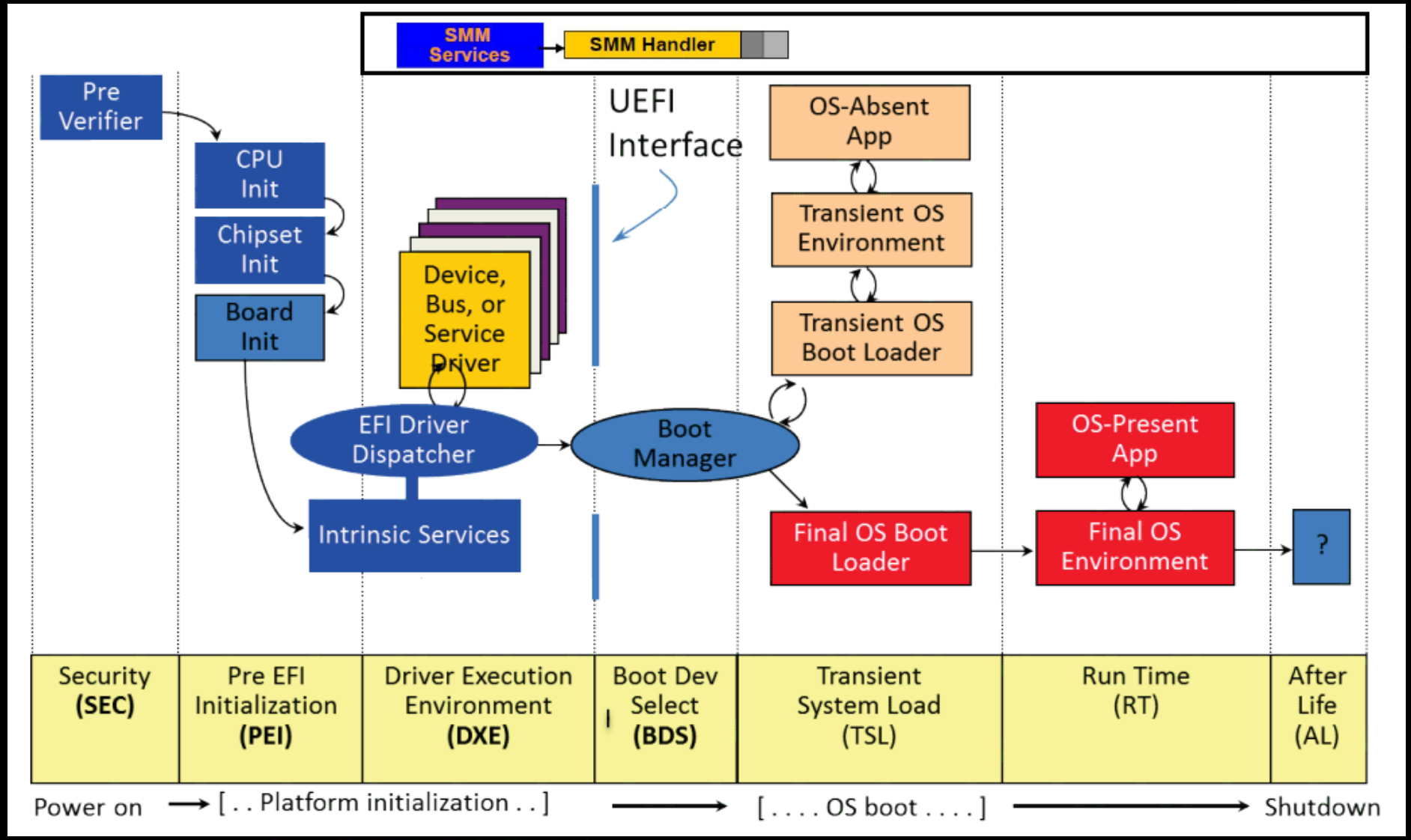
# Brief Intro to (U)EFI: What is it?



- **(Unified) Extensible Firmware Interface**
- modern industrial standard for x86 firmware
- initially developed by Intel as BIOS replacement for IA64
- used by Macs since 2007, PCs since 2009
- performs HW initialization required to start an OS
- modular and feature rich, uses well defined and known formats
- mostly written in C, much easier to develop as legacy BIOS

# Brief Intro to (U)EFI: Boot flow





# Brief Intro to (U)EFI: SEC concepts



- purpose: initialize enough HW to run code that uses stack
- wrote in assembler, microarchitecture dependent
- provided by CPU vendor
- despite of it's name, makes no security checks by default
- switches BSP to 32 bit mode with flat memory
- detects and initializes CPU caches
- sets L2 cache to no-eviction mode[1], so it can be used as preliminary RAM
- finds PEI Core and transfers control to it


- [1] a.k.a. Cache-as-RAM, more info here: coreboot.org/images/6/6c/LBCar.pdf

# Brief Intro to (U)EFI: PEI concepts



- purpose: initialize RAM and mission-critical hardware
- has two sub-phases: BeforeMem and AfterMem
- binaries stored in PE32 and TE[2] formats
- BeforeMem binaries must be executable in place
- PEI Core and Modules
- PEI dependency expressions
- PEI-to-PEI Interfaces and Hand-Off Blocks
- PeiServices
- on S3 resume, UEFI boot process ends here
- otherwise, control and HOBs are transferred to DXE Core


[2] Terse Executable, a PE32 with most of it's headers cut off to save precious space in L2 cache

# Brief Intro to (U)EFI: DXE concepts



- purpose: initialize all remaining hardware
- 64 bit code on most machines, exceptions are rare
- binaries stored in PE32+ format
- DXE Core, Dispatcher, Drivers, OROMs and Applications
- DXE Protocols
- SMM Core, Dispatcher, Drivers and SMI Handlers
- DXE dependency expressions
- BootServices and RuntimeServices
- dispatches all available drivers and starts BDS application

Driver Execution Environment **(DXE)**

# Brief Intro to (U)EFI: BDS concepts



- purpose: find a bootloader and transfer control to it
- POST screen, BIOS Setup, firmware update, recovery tools, etc.
- all UEFI stuff including BootServices, loaded drivers, published protocols and so on, is available to UEFI bootloaders
- final OS bootloader should generate ExitBS event
- after that, only runtime services, ACPI tables, SMI handlers and special purpose memory regions are accessible to OS, all other memory is marked free

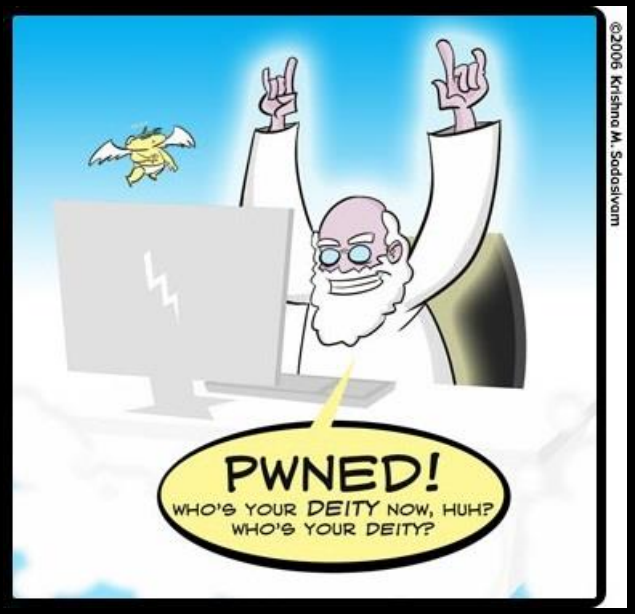
# What an Attacker Can Do?



# What an Attacker Can Do: PWNAGE



- persist across OS reinstall and hard drive change
- obtain full access to hardware, physical memory and CPU context
- steal secrets from OS and sent them out by network
- resist detection to a point of being virtually undetectable without additional hardware
- in other words: **PWN EVERYTHING**



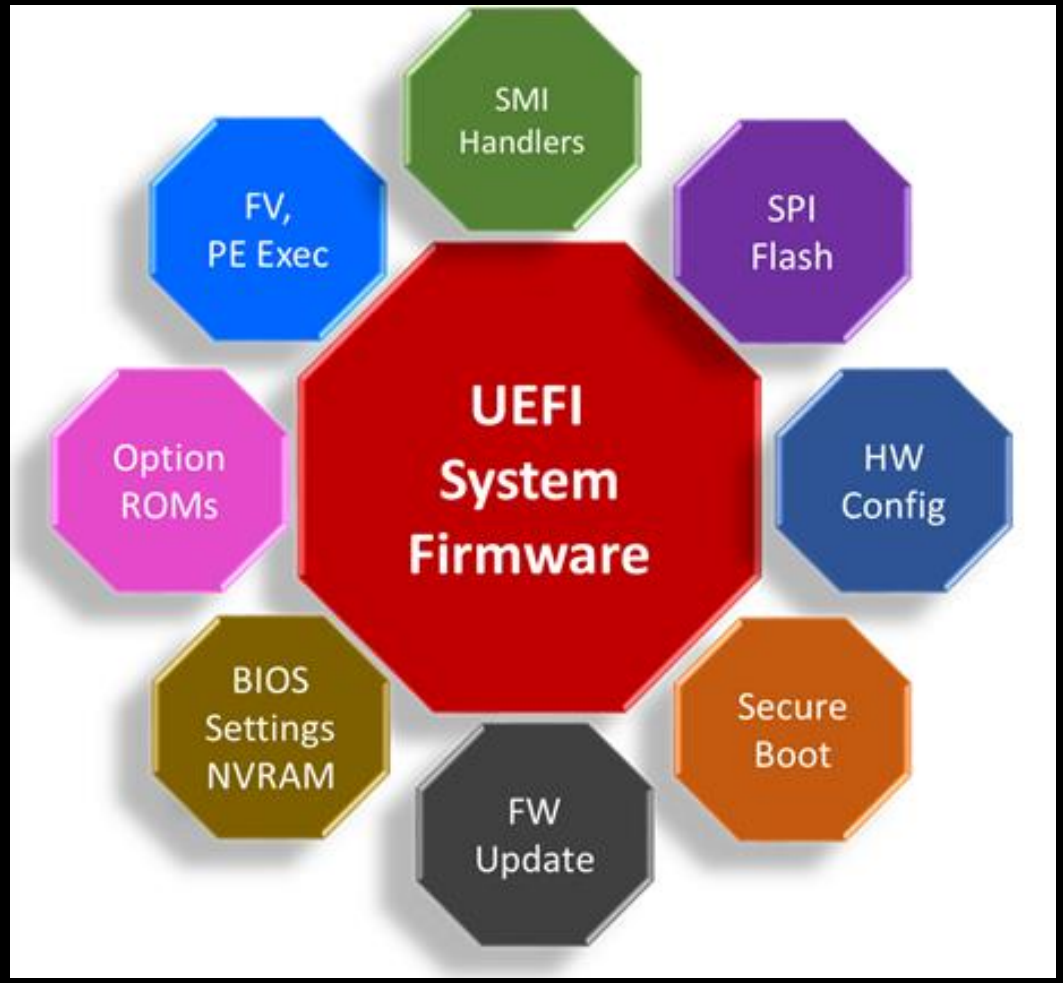
# What an Attacker Can Do: A quote




[3] “How Many Million BIOSes Would You Like to Infect” by Corey Kallenberg and Xeno Kovah

# Attack Vectors






[4] “UEFI, Open Platforms, and the Defender’s Dilemma” by Vincent Zimmer

# Attack Vectors: SPI flash chip



- the most desired and dangerous attack vector
- successful attack leads to full system control if verified boot isn't used, DoS otherwise
- provides persistence, renders all other protections futile
- there are millions of systems without any SPI flash protection, waiting to be pwned

```
Intel (R) Flash Programming Tool. Version:  8.1.10.1286
Copyright (c) 2007 - 2012, Intel Corporation. All rights reserved.

Platform: Intel(R) Patsburg Chipset - Reserved DID 0x1D41
Reading HSFSTS register... Flash Descriptor: Valid

    --- Flash Devices Found ---
    W25Q64BV    ID:0xEF4017    Size: 8192KB (65536Kb)

PDR Region does not exist.

- Reading Flash [0x003000]    8KB of    8KB - 100% complete.
- Erasing Flash Block [0x003000] - 100% complete.
- Programming Flash [0x003000]    8KB of    8KB - 100% complete.
- Verifying Flash [0x003000]    8KB of    8KB - 100% complete.
RESULT: The data is identical.

FPT Operation Passed
```

# Attack Vectors: SMI handlers



- one of the most privileged execution modes available
- successful attack very often provides SPI flash access
- tons of systems are (still) vulnerable
- hard to fix even with source code, almost impossible without one
- can’t be disabled without major rework of the whole firmware


[3] “How Many Million BIOSes Would You Like to Infect” by Corey Kallenberg and Xeno Kovah

# Attack Vectors: S3 BootScript



- set of instructions to restore system configuration on ACPI S3 resume, gathered in DXE phase
- enables fast resume, when the whole DXE phase is skipped
- was stored in ACPI NVS memory available for OS-level attacker
- now copied to SMRAM, a successful attack on SMM renders SMM-based BootScript protection useless
- SPI flash protections can be disabled after a vulnerable S3 cycle
- most systems are vulnerable even after a year from disclosure

# Attack Vectors: Option ROMs



- stored on internal PCI(e) devices
- and any external devices capable of PCI bus mastering: Firewire, Thunderbolt, ExpressCard, etc.
- loaded during DXE phase as normal DXE drivers
- no verification performed if SecureBoot is disabled
- some systems will launch an Option ROM even if verification fails
- attacker can modify an Option ROM of an existing device

```
Intel(R) Rapid Storage Technology - Option ROM - 10.5.0.1034
Copyright(C) 2003-11 Intel Corporation.  All Rights Reserved.

 RAID Volumes:                                  * = Accelerated Disk/Volume
 ID   Name             Level           Strip      Size Status      Bootable
 1    Volume_0000       RAID0(Cache)    128KB    18.6GB Normal        No

 Physical Devices:
 Port Device Model     Serial #                      Size Type/Status(Vol ID)
 *0   WDC WD20EADS-00R WD-WCAVY0006618             1.8TB Non-RAID Disk
 1    INTEL SSDSA2SP02 LC101000CJ020AGN           18.6GB Cache Disk(1)
Press <CTRL-I> to enter Configuration Utility...
```

# Attack Vectors: BIOS Setup



- can control SPI flash access
- controls SecureBoot
- settings are stored in NVRAM variable called “Setup”, which can be changed from UEFI Shell (always) or any UEFI-bootable OS (if the variable has RT flag)
- protected by password, which can be stored using a weak algorithm, removed with “factory password” or reset by PC vendor

ACERFTW.rom

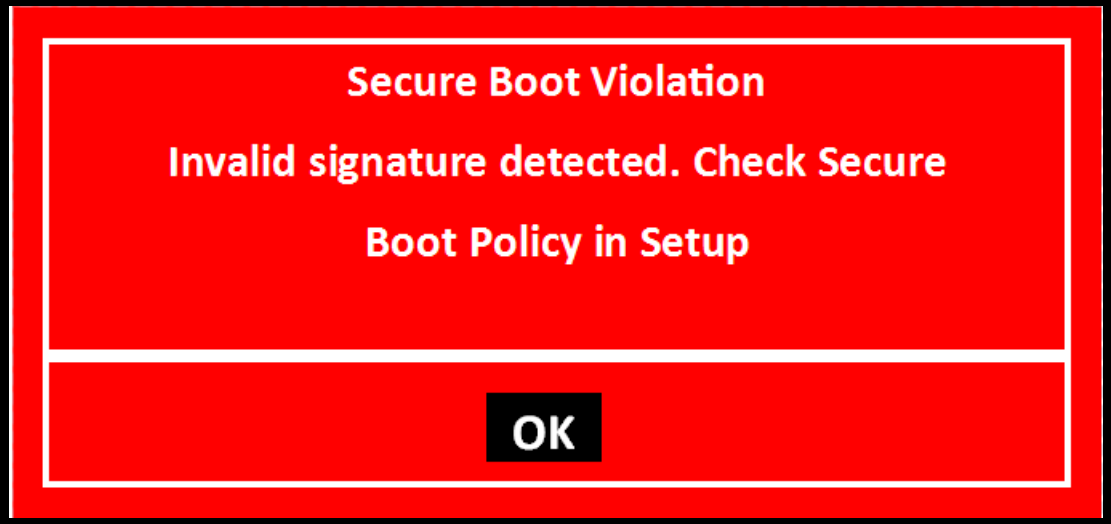
```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0025BFF0   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0025C000   5F 50 53 57 5F 00 00 00 00 00 00 00 00 00 00 00  _PSW_...........
0025C010   07 41 43 45 52 46 54 57 ED 00 00 00 00 00 00 00  .ACERFTWí.......
0025C020   07 55 53 45 52 50 57 44 CF 00 00 00 00 00 00 00  .USERPWDÏ.......
0025C030   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
```

Signature | PW Length | PW ASCII Clear Text | Checksum8(Length + Text)

# Attack Vectors: SecureBoot



- controlled from BIOS Setup
- default mode trusts too many parties: PC vendor, Microsoft, anyone who get an image signed by Microsoft ($99 only), Canonical, may be more.
- erasing PK variable disables SecureBoot completely
- various known vulnerabilities can still be present



# Attack Vectors: Vendor-specific



- engineering backdoors may exist in different parts of the system:
  0. factory password for BIOS Setup
  1. vendor’s SMM-based BIOS flasher used for SMBIOS information editing
  2. special signature or hash in DB to bypass SecureBoot
  3. anything that can help restore a locked or unbootable configuration without RMA
- vulnerabilities may (sure) exist in vendor flash tools, SMI handlers, SecureBoot implementations, etc.
- there are almost no public researches for vendor-specific vulnerabilities, mostly because of NDAs

# Protections



# Protections: SPI flash



- read-only flash is the best, but almost impossible because of ME and NVRAM
- Intel Flash Descriptor restricts access to regions other than BIOS
- hardware-validated boot (Intel BootGuard, AMD PSP) converts evil SPI reflash from total PWNAGE to an ordinary DoS
- BiosGuard (PFAT) disables SPI reflash from SMM code
- Protected Range Registers can be used to restrict read/write access to certain flash regions, even for SMM code
- SMM_BWP bit disables SPI reflash from normal OS code
- BLE/BIOS_WE considered broken[5] and shouldn't be used anymore

[5] "Attacks on UEFI security, inspired by Darth Venamis's misery and Speed Racer" by Korey Kallenberg and Rafal Wojtczuk

# Protections: SMI handlers



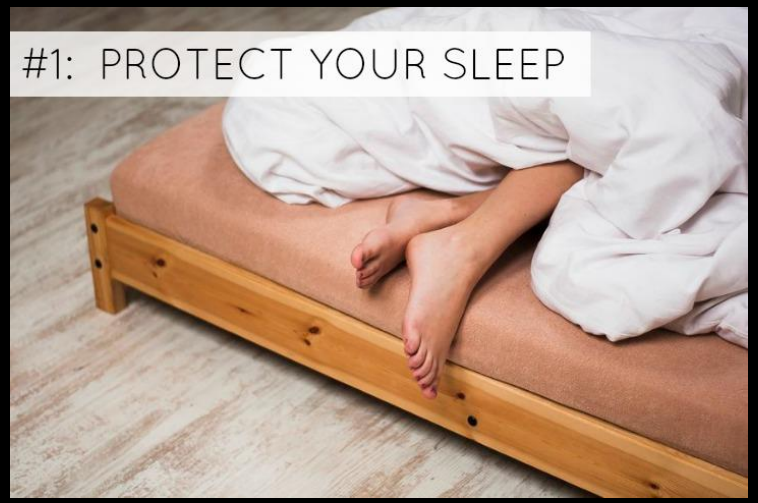
- the best protection is to disable SMM for good, if possible
- TSEGMB register protects SMM code and data from DMA attacks
- Intel SMI Transaction Monitor can be used to protect the system from evil SMM code
- Intel Software Guard Extension can be used to protect an application from all evil code, including SMM
- MSR_SMM_FEATURE_CONTROL can be used to protect from SMM call-outs
- Phoenix NX trick[6] can also be used to protect from SMM call-outs even on AMD and older-than-Haswell Intel systems
- Defensive coding and multistage code review are very helpful

[6] “UEFI Firmware – Securing SMM” by Dick Wilkins

# Protections: S3 BootScript



- the best protection is to disable S3 for good
- BootScript can be stored inside security coprocessor (AMD PSP)
- SmmLockBox can be used to store a copy of the BootScript in SMRAM to verify that the original is not changed
- hardware-reduced platforms and embedded systems can use a static BootScript-less S3 resume boot path



# Protections: Option ROMs



- disable any external interfaces with PCI bus mastering
- enable, setup and use SecureBoot, set it’s policy to prevent launch of unsigned Option ROMs
- prevent unnoticed hardware changes


(Tamper evident screws, as suggested by Eric Michaud at 30c3)

# Protections: BIOS settings

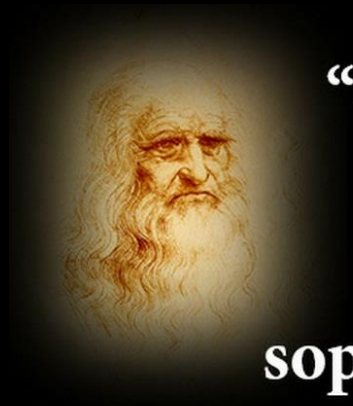


- the best protection is to remove NV from NVRAM, a.k.a. NVRAM emulation
- use BIOS password to prevent easy access to BIOS Setup
- use SecureBoot to prevent loading of UEFI Shell, which can access all non-auth variables including “Setup” and “Defaults”
- review the code of your bootloader, as it have the same access level as UEFI Shell until ExitBS event is generated

# Protections: Vendor-specific and 0-days



- the best protection from that stuff is a good old paranoia
- second best are KISS principle and common sense
- if the system can boot your favorite OS without any given component, remove it to reduce possible attack surface
- remove all vendor-specific flash tools, recovery tools, password restoration tools, etc.
- use a minimal trusted bootloader, write one if you don’t trust shim



“Simplicity is the ultimate sophistication.”

— Leonardo da Vinci

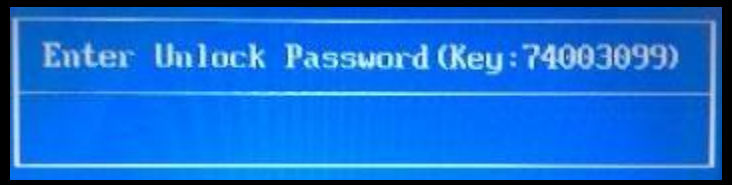
# An Average System



# An Average System: Acer R3-471T



- entry-level laptop produced by Acer
- based on Haswell ULT SoC, Core i3-4030U in my case
- uses Quanta ZQX motherboard with 4Gb DDR3L memory and optional dedicated graphic card onboard
- firmware based on Insyde $H_2O$ platform, UEFI 2.3.1C compatible
- latest firmware version is v1.09, published in July 2015
- supports SecureBoot, BIOS Setup password protection, BIOS Setup unlock via Acer Support



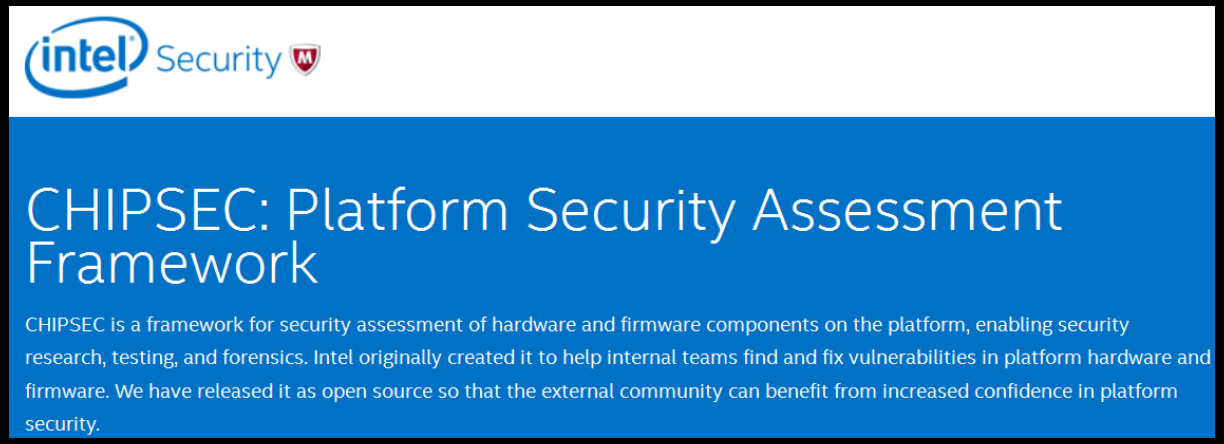
# Test Tools



# Test Tools: CHIPSEC



- platform security assessment framework by Intel ATR
- has two versions, open source and proprietary
- open source version is available on github.com/chipsec/chipsec
- proprietary version is available for systems vendors and has diagnostics for yet undisclosed vulnerabilities
- can be started from Linux and UEFI Shell
- detects various misconfigurations and vulnerabilities specific to the Intel platform it runs on



# Test Tools: UEFITool



- open source tool to decompose, view and modify UEFI firmware images
- parses firmware image file into a tree structure, detects some misconfigurations
- reconstructs an image with all modifications done

| Name | Action | Type | Subtype | Text |
|---|---|---|---|---|
| BIOS region | | Region | BIOS | |
| Padding | | Padding | Non-empty | |
| 7A9354D9-0468-444A-81CE-0BF617D890DF | | Volume | FFSv2 | |
| 4A538818-5AE0-4EB2-B2EB-488B23657022 | | File | DXE core | |
| Compressed section | | Section | Compressed | |
| Raw section | | Section | Raw | |
| Padding | | Padding | Non-empty | |
| 7A9354D9-0468-444A-81CE-0BF617D890DF | | Volume | FFSv2 | |
| ABB74F50-FD2D-4072-A321-CAFC72977EFA | | File | PEI module | SmmRelocPeim |
| 35B898CA-B6A9-49CE-8C72-904735CC49B7 | | File | DXE core | DxeMain |
| 4D37DA42-3A0C-4EDA-B9EB-BC0E1DB4713B | | File | PEI module | PpisNeededByDxeCore |
| F276BDEC-6C41-21E5-9E71-00A13807B45E | | File | DXE driver | RestoreMtrr |
| 987EA6EA-FBFD-4273-B819-A7210ADF6760 | | File | DXE driver | StatusCodeReport |
| E8F8CCFB-E880-0361-BCD1-FE248B2A307E | | File | DXE driver | SaveMemoryConfig |
| C8F9202B-8983-4470-940A-E65E7A552270 | | File | DXE driver | SavePegConfig |
| 51C9F40C-5243-4473-B265-B3C8FFAFF9FA | | File | DXE driver | Crc32SectionExtract |
| FC1BCDB0-7D31-49AA-936A-A4600D9DD083 | | Section | GUID defined | |
| DXE dependency section | | Section | DXE dependency | |
| PE32 image section | | Section | PE32 image | |

# Test Tools: UEFI Shell

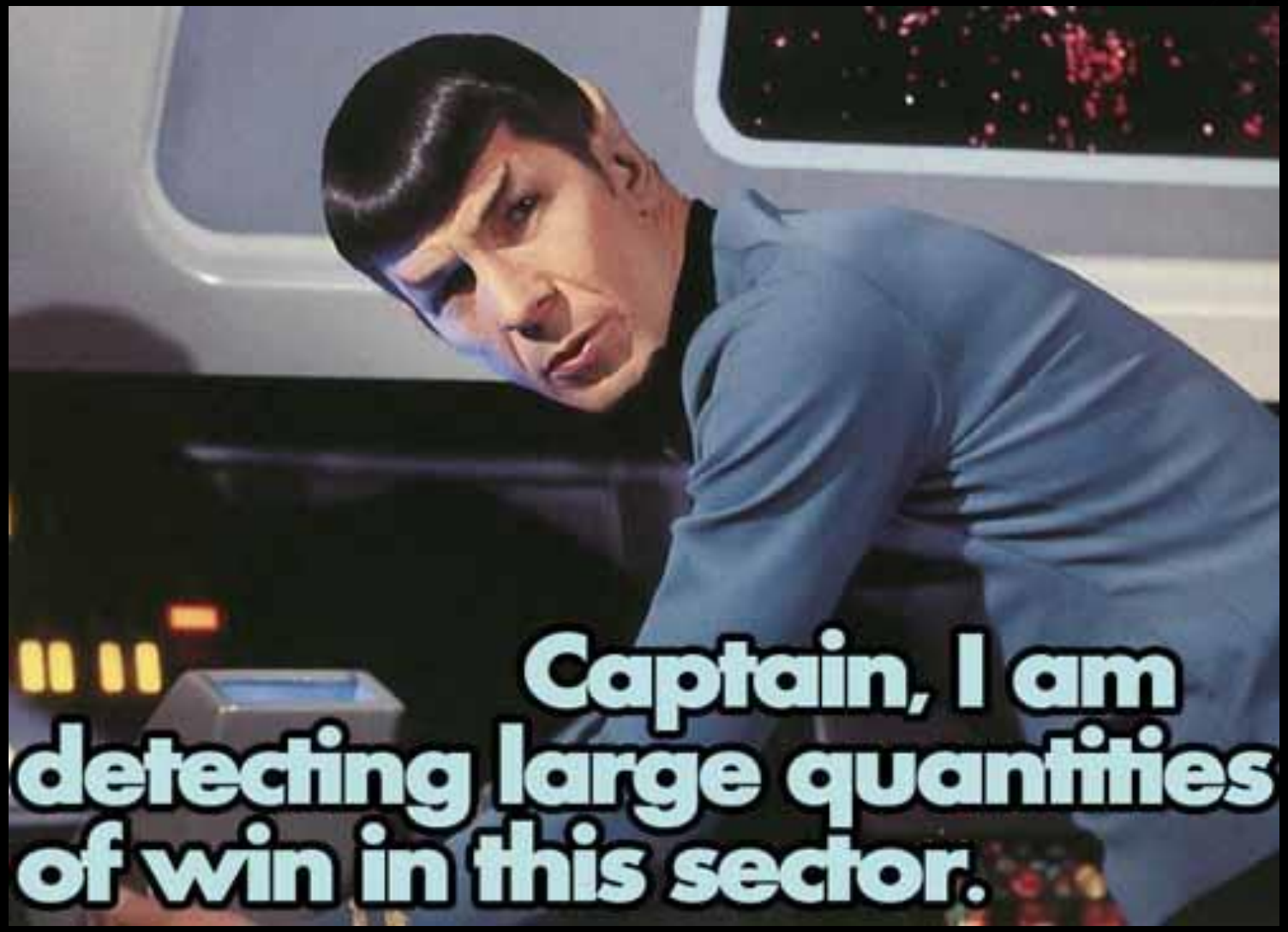


- loaded as transient UEFI booloader
- provides console with access to UEFI state after BDS phase end
- has various useful command to view and change physical memory, load and unload DXE drivers, change NVRAM variables and so on
- can run UEFI applications under itself
- acts as modern DOS replacement, single user CUI OS with direct memory and hardware access
- useful for UEFI debugging, extremely dangerous in wrong hands

```
EFI Shell version 2.00 [4096.1]
Current running mode 1.1.2
Device mapping table
  fs0     :Removable HardDisk - Alias hd52g0b blk0
           Acpi(PNP0A03,0)/Pci(1D|7)/Usb(6,0)/HD(Part1,Sig90909090)
  blk0    :Removable HardDisk - Alias hd52g0b fs0
           Acpi(PNP0A03,0)/Pci(1D|7)/Usb(6,0)/HD(Part1,Sig90909090)

Press ESC in 1 seconds to skip startup.nsh, any other key to continue.
Shell> _
```

# Test Results





# Test Results: CHIPSEC



- PR registers set to WP the whole BIOS except NVRAM — OK
- PR registers are reset to zero after S3 — FAIL
- S3 BootScript is located outside SMRAM, not protected from any changes, has numerous DISPATCH opcodes — FAIL
- SMM_BWP is not set, NVRAM region is protected only by BLE/BIOS_WE — FAIL
- “Setup” variable has RT flag — FAIL

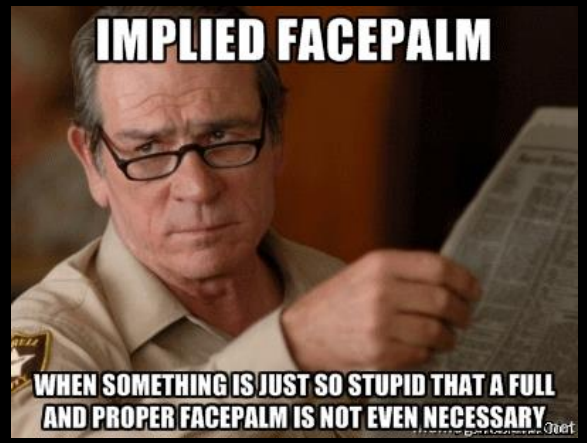
QUADRUPLE FACEPALM

When you fail really, really bad.

# Test Results: UEFITool



- BIOS has Capsule flash services, Insyde IHISI flash interface, Intel PFAT flash driver, SecureFlash driver — MISTRUST
- has WPBT[7] driver, the firmware can automatically install and run Windows executables — MISTRUST
- has UnlockPwd driver, the attacker can override BIOS Setup password with Acer's tech support help — MISTRUST
- BIOS Setup password is stored as cleat text — FAIL

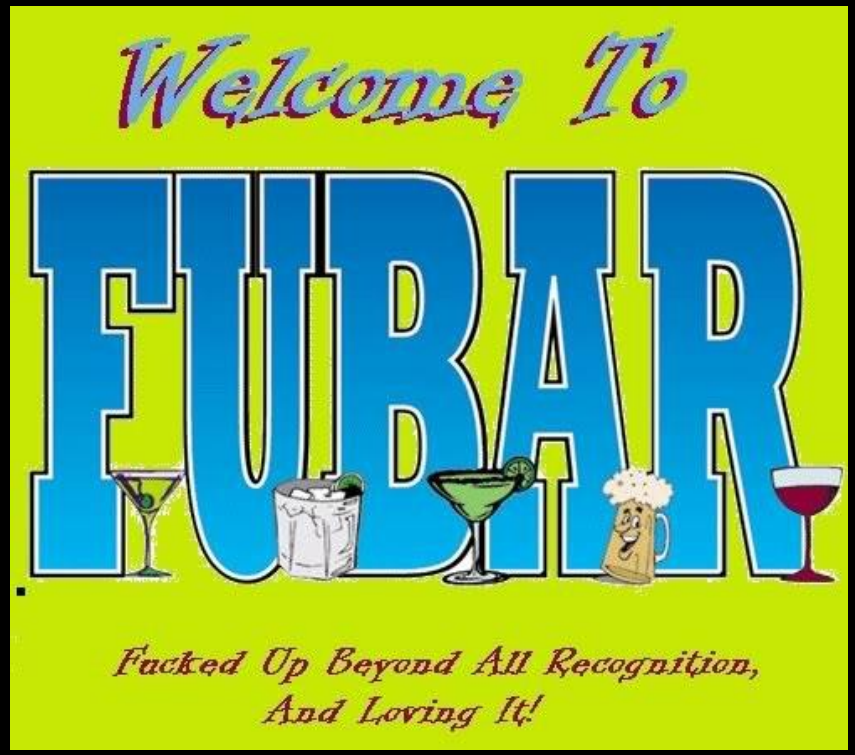



[7] http://download.microsoft.com/download/8/a/2/8a2fb72d-9b96-4e2d-a559-4a27cf905a80/windows-platform-binary-table.docx

# Test Results: UEFI Shell



- UEFI network stack is loaded and operational regardless of PXE and/or boot settings MISTRUST
- SPI flash protection and SecureBoot can both be disabled by changing “Setup” variable FAIL



# What Now?



# What Now: Answers



- emulate NVRAM to get rid of any flash writes after POST
- make the whole BIOS region read-only
- protect BIOS password storage from reading
- disable S3 and remove it’s support for good
- setup, test and use SecureBoot
- remove WPBT driver, UnlockPwd driver, BIOS protection drivers, flash drivers, recovery drivers, UEFI network stack and other unneeded and potentially vulnerable components
- prepare a golden BIOS image and periodically check against it

# Prepare to Dig Deep



# Prepare to Dig Deep: Hardware



- field surgery on UEFI images will be painful
- it will crash, it will brick, it will go nuts, but you can try again until ether it is done or you are done
- some things can be done to ease the pain, such as:
  0. disassemble the laptop to the bare board and remove all you can remove from it
  1. find or buy board schematics
  2. buy (and modify, if needed) mPCIe LPC POST card
  3. get a decent SPI programmer and replace onboard SPI chip with ZIF socket, or use a SPI emulator instead

# Prepare to Dig Deep: POST card

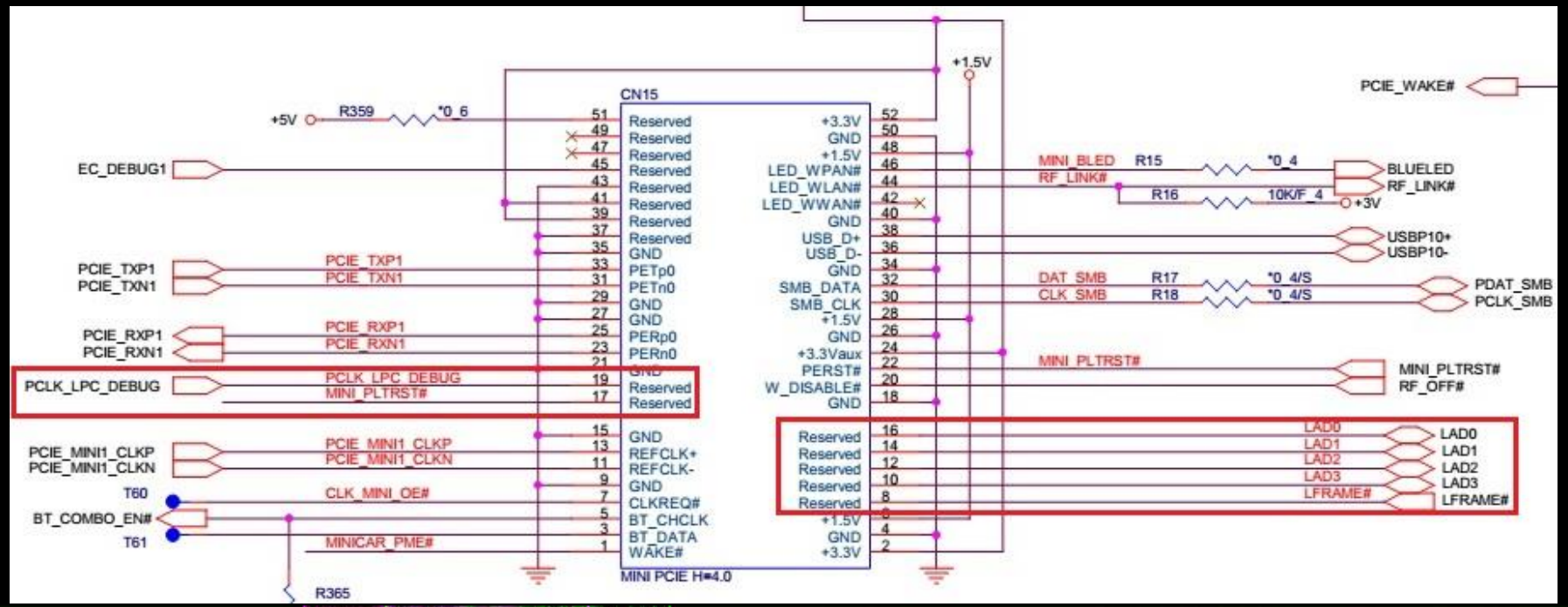


- most laptops still use SuperIO controllers on LPC bus
- debug port 80h is decoded to LPC bus by default
- Sintech makes cheap LPC debug cards, ideal for our purposes
- there is no standard LPC debug port pinout, so your system may use a different one
- two possible solutions: either find a compatible card or make one compatible

# Prepare to Dig Deep: ST8672 card pinout



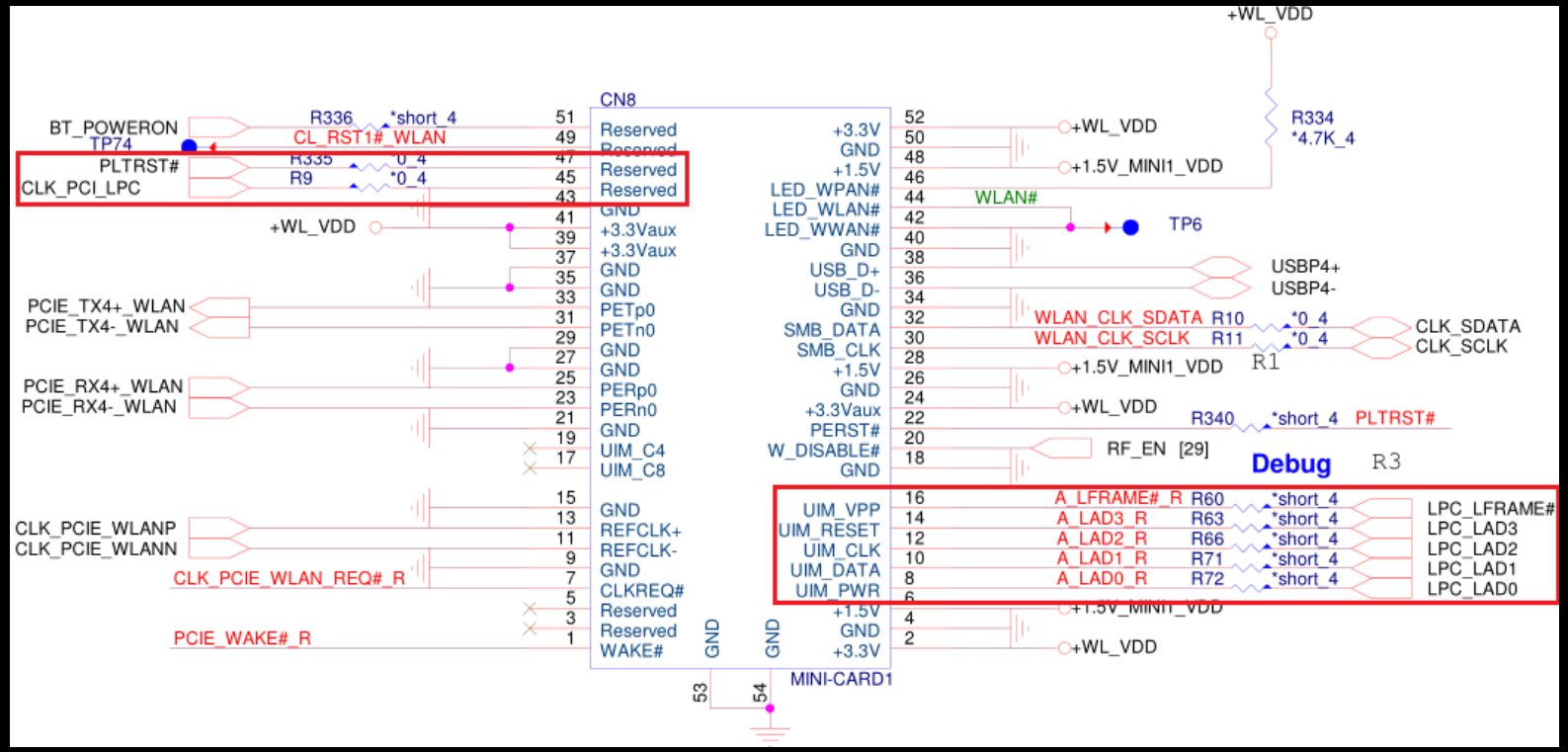
- Sintech ST8672 is an LPC debug card with mPCIe interface
- costs about $5, easy to modify because of single-layer PCB
- LPC signals are on pins 8 (LAD0), 10, 12, 14 and 16 (LFRAME#)
- LPC clock is on pin 19, platform reset signal is on pin 17
- this is the most popular pinout with the cheapest cards available



# Prepare to Dig Deep: Quanta ZQX pinout



- Quanta uses a different pinout for it's newer boards
- LPC signals are still on pins 8, 10, 12, 14 and 16, but reversed
- LPC clock is on pin 45 (instead of 19), platform reset signal is on pin 47 (instead of 17), some resistors must be added



# Prepare to Dig Deep: SPI programmer



- cheapest SPI programmers (down to $3) are based on WCH CH341A USB-serial converter, but they totally worth the price
- FTDI FT2232H-based programmers are relatively cheap (around $20) and able to flash all current SPI chips using flashrom[8]
- even better options are Autoelectric TL866A and Dediprog SF600, but beware the price
- use voltage level shifter to interface with 1.8V SPI flash chips, if your programmer doesn't support them natively




[8] http://flashrom.org/Flashrom

# Prepare to Dig Deep: SPI emulator



- replaces physical SPI flash chip with virtual one
- 100x speed-up of flash update, image editing on-the-fly
- the only downside compared to SPI programmer is price, which starts on $300 and goes up very fast
- tried Samedisk ZC25128 and Dediprog EM100-Pro, 10/10 will use again


# Prepare to Dig Deep: Software



- Linux with python2 for using Chipsec, openssl and efitools for setting up SecureBoot and signing your bootloader
- TianoCore EDK2 build environment to build your PEI/DXE drivers
- UEFI Shell binary that works on your platform
- UEFITool or any other similar tool to modify UEFI images
- good disassembler like radare2 or IDA
- Intel ASL compiler for possible DSDT reverse and modification
- your favorite hex editor
- anything else you will find useful

# Fix It Yourself



# Fix It Yourself: NVRAM emulation



- NVRAM is accessible using four UEFI runtime services: GetVariable, SetVariable, GetNextVariableName, QueryVariableInfo
- we need to hook this services after they are published and after all reads and writes initiated by the firmware are done
- possible hook points are on EndOfDxe, ReadyToBoot and ExitBS
- EndOfDxe hook will render BIOS Setup useless – too early
- ReadyToBoot hook will require auth. variable emulation for SecureBoot – too hard to implement properly
- ExitBS hook is the latest point possible, and a bootloader will have full access to the firmware – SecureBoot and trusted bootloader are mission-critical

# Fix It Yourself: EmuVariable driver



- TianoCore has a reference NVRAM emulation driver in MdePkg
- Clover Bootloader has an expanded one[9] for starting OSX on PC
- changed the driver from Clover to start emulation on ExitBS

```
/**

  EmuVariable Driver main entry point.
  The Variable driver places the 4 EFI runtime services in the EFI System Table
  and installs arch protocols for variable read and write services being available.
  It also registers notification functions for EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE
  and EVT_SIGNAL_EXIT_BOOT_SERVICES events.

  @param[in] ImageHandle    The firmware allocated handle for the EFI image.
  @param[in] SystemTable    A pointer to the EFI System Table.

  @retval EFI_SUCCESS       Variable service successfully initialized.

**/
EFI_STATUS
EFIAPI
VariableServiceInitialize (
  IN EFI_HANDLE         ImageHandle,
  IN EFI_SYSTEM_TABLE   *SystemTable
  )
```

[9] http://sourceforge.net/p/cloverefiboot/code/HEAD/tree/EmuVariableUefi/

# Fix It Yourself: Problems with FLOCKDN



- PR registers should be set to write-protect the real NVRAM region
- which means it must also be done on ExitBS
- but we can’t change their values after FLOCKDN bit is set
- it’s done by PchInitDxe driver on this platform
- the driver must be patched to prevent locking the flash configuration in DXE phase, but we absolutely must lock it in our driver afterwards

```
0000000180001446: BA 00 80 00 00      mov         edx,8000h BIT15 - FLOCKDN
000000018000144B: 41 8D 8D 04 38 00   lea         ecx,[r13+00003804h]  offset of SPIBAR (3800h) + HSFS register (04h)
                  00
0000000180001452: E8 A5 B5 00 00      call        000000018000C9FC MmioOr16(AddressInEcx, ValueInEdx)
```

# Fix It Yourself: Set PR registers

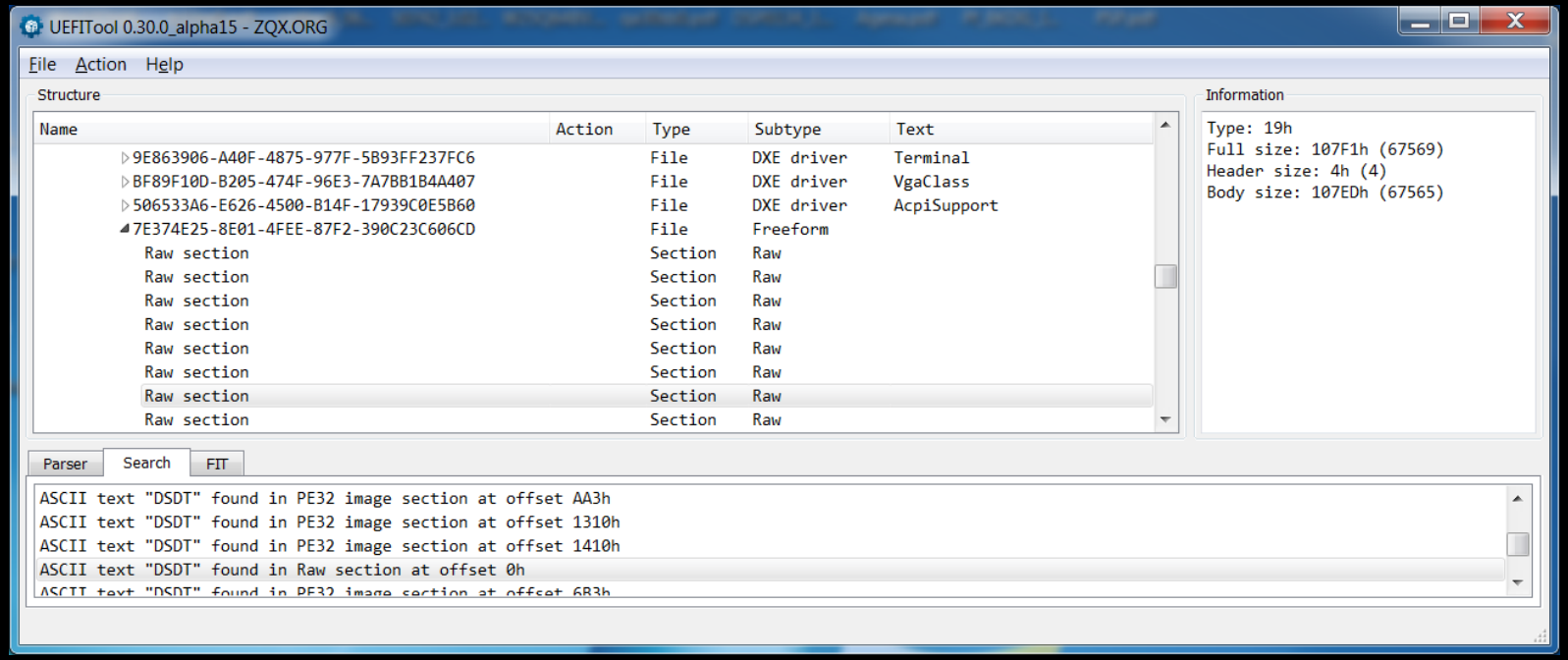


```
// Set PR0 to protect the whole BIOS region and set FLOCKDN bit to prevent it's removal
// All magic values below are platfrom-specific, use chipset datasheet to obtain them
//
{
  #pragma warning( disable : 4306) // Disable the warning about converting 32-bit values to 64-bit pointers and back
  UINT32 RcbaLocationAddress = 0xE00F80F0; // Memory-mapped PCI device B0:D31:F0, register offset 0xF0;
  UINT32 RootComplexBaseAddress = (*(UINT32*)RcbaLocationAddress) & 0xFFFFFC00;
  UINT32 SpiBaseAddress = RootComplexBaseAddress + 0x3800; // SPIBAR for all registers below
  UINT32 HsfsRegisterAddress = SpiBaseAddress + 0x04;      // HSFS register for FLOCKDN
  UINT32 BfprRegisterAddress = SpiBaseAddress + 0x00;      // BFPR register to get BIOS region base and limit
  UINT32 Pr0RegisterAddress = SpiBaseAddress + 0x74;       // PR0 register to enable BIOS region write protection
  UINT32 Pr1RegisterAddress = SpiBaseAddress + 0x78;       // PR1 register to enable BIOS password read protection

  UINT16 HsfsRegisterValue = *(UINT16*)HsfsRegisterAddress;
  UINT32 BfprRegisterValue = *(UINT32*)BfprRegisterAddress;
  UINT32 Pr0RegisterValue = BfprRegisterValue | BIT31;     // Set PR0 to cover the whole BIOS region, enable WP
  UINT32 Pr1RegisterValue = 0x825D825C;                    // Set PR1 to cover 25C000 - 25D000 flash region,
                                                           // where the BIOS password is stored as clear text,
                                                           // enable both RP and WP
  //
  // Check FLOCKDN bit to be 0
  //
  if ((HsfsRegisterValue & BIT15) == 0) {
    //
    // Set new PR0 and PR1 values
    //
    *(UINT32*)Pr0RegisterAddress = Pr0RegisterValue;
    *(UINT32*)Pr1RegisterAddress = Pr1RegisterValue;

    //
    // Set FLOCKDN bit
    //
    HsfsRegisterValue |= BIT15;
    *(UINT16*)HsfsRegisterAddress = HsfsRegisterValue;
  }
}
```

# Fix It Yourself: Find DSDT



- ACPI S3 state availability is reported to OS via DSDT table
- DSDT must patched to always report S3 as disabled
- but it can also be controlled by BIOS Setup
- on this platform, DSDT is stored in a RAW section together with other ACPI tables



# Fix It Yourself: Decompile DSDT



- ACPI tables are stored in ACPI Machine Language format, but they can be decompiled into human-readable ACPI Source Language using Intel iASL Compiler[10]

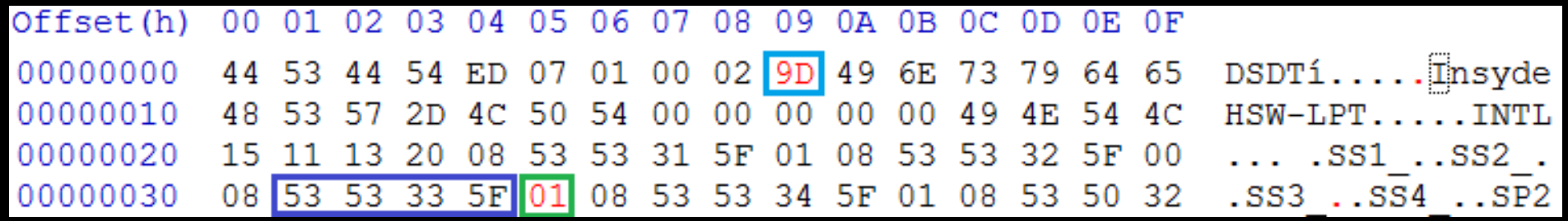
```
    Name (SS1, One)
    Name (SS2, Zero)
    Name (SS3, One)
    Name (SS4, One)

Scope (_SB)
{
...
    If (SS3)
    {
        Name (_S3, Package (0x04)  // _S3_: S3 System State
        {
            0x05,
            Zero,
            Zero,
            Zero
        })
    }
...
```

[10] https://www.acpica.org/downloads

# Fix It Yourself: Patch DSDT



- you can try to edit and recompile DSDT, but it can be tricky to get rid of errors
- it's much easier to patch the default value of SS3 variable from 1 to 0 and fix DSDT checksum
- decompile the patched file again, iASL tells what the valid checksum should be



| Offset(h) | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00000000 | 44 | 53 | 44 | 54 | ED | 07 | 01 | 00 | 02 | 9D | 49 | 6E | 73 | 79 | 64 | 65 | DSDTí.....Insyde |
| 00000010 | 48 | 53 | 57 | 2D | 4C | 50 | 54 | 00 | 00 | 00 | 00 | 00 | 49 | 4E | 54 | 4C | HSW-LPT.....INTL |
| 00000020 | 15 | 11 | 13 | 20 | 08 | 53 | 53 | 31 | 5F | 01 | 08 | 53 | 53 | 32 | 5F | 00 | ... .SS1_..SS2_. |
| 00000030 | 08 | 53 | 53 | 33 | 5F | 01 | 08 | 53 | 53 | 34 | 5F | 01 | 08 | 53 | 50 | 32 | .SS3_..SS4_..SP2 |

# Fix It Yourself: Patch S3 setup handler



- DSDT patch alone may not be enough, because SS3 variable can be changed from it's default during DXE phase
- find and patch the handler that changes this variable…
- …or just replace dangerous SS3 with harmless SS1 in it
- you can patch it differently, but the goal is to disable S3 for OS

```
00000001800005D6: BA 53 53 33 5F      mov     edx,5F335353h  SS3_ ASCII text
00000001800005DB: 48 8B 4C 24 50      mov     rcx,qword ptr [rsp+50h] new value
00000001800005E0: E8 9B FC FF FF      call    0000000180000280 SetValue
```

# Fix It Yourself: Final stone on S3 grave



- now we need to make sure that S3 boot path will never work, so even if the attacker manages to prepare the whole new DSDT with enabled S3, it will only result in hard reset or crash
- it can be done by patching PeiGetBootMode to never return BOOT_ON_S3_RESUME
- or by destroying S3 BootScript on ExitBS
- or by writing a simple PEI driver that will call PeiResetService if PeiGetBootMode tells we are in S3 resume
- either way is fine

# Fix It Yourself: PreventS3Pei driver

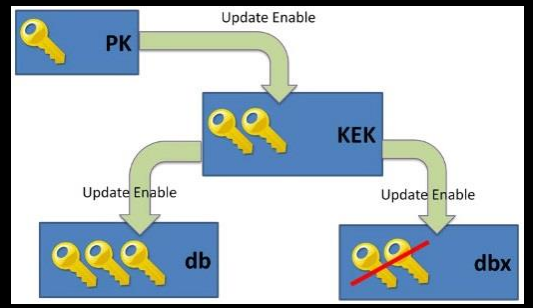


```
EFI_STATUS
EFIAPI
PreventS3PeiEntry (
  IN EFI_PEI_FILE_HANDLE       FileHandle,
  IN CONST EFI_PEI_SERVICES    **PeiServices
  )
{
    EFI_STATUS  Status;
    EFI_BOOT_MODE BootMode = BOOT_WITH_FULL_CONFIGURATION;

    // Get current boot mode
    (*PeiServices)->GetBootMode(PeiServices, &BootMode);

    // Continue normal boot if it's not S3
    if (BootMode != BOOT_ON_S3_RESUME)
        return EFI_SUCCESS;

    // Try to reset the system
    (*PeiServices)->ResetSystem(PeiServices);

    // Hang if reset attempt failed
    while(1);

    // Should never end here
    return EFI_INVALID_PARAMETER;
}
```

# Fix It Yourself: SecureBoot key pairs



- now we have our SPI flash read-only after ExitBS
- but still accessible from UEFI Shell and bootloaders
- SecureBoot is to the rescue in this case
- to use SecureBoot properly, we need three different key pairs:
  1. Platform Key to enable SecureBoot and sign KEK
  2. Key Exchange Key to sign db and dbx
  3. Image Signing Key to sign bootloaders and executables
- all this keys can be generated by using OpenSSL




# Fix It Yourself: Generate PK



- PK key pair suitable for SecureBoot can be generated by the following openssl call:

```
openssl req -new -x509 -newkey rsa:2048 -subj
"/CN=Platform Key/" -keyout PK.key -out PK.crt -days
3650 -nodes -sha256
```

- cert-to-efi-sig-list utility from efitools package converts generated certificate to EFI Signature List format:

```
cert-to-efi-sig-list -g "$(uuidgen)" PK.crt PK.esl
```

- PK.esl must be self signed using sign-efi-sig-list utility:

```
sign-efi-sig-list -k PK.key -c PK.crt PK PK.esl
PK.auth
```

# Fix It Yourself: Generate KEK



- KEK key pair can be generated by a similar openssl call:

```
openssl req -new -x509 -newkey rsa:2048 -subj
"/CN=Key Exchange Key/" -keyout KEK.key -out KEK.crt
-days 3650 -nodes -sha256
```

- use cert-to-efi-sig-list to convert KEK.crt into KEK.esl:

```
cert-to-efi-sig-list -g "$(uuidgen)" KEK.crt KEK.esl
```

- to add multiple certificates into KEK, concatenate their ESL files:

```
cat KEK.esl MsKek.esl > KEK.esl
```

- combined KEK.esl must be signed by PK using sign-efi-sig-list:

```
sign-efi-sig-list -k PK.key -c PK.crt KEK KEK.esl
KEK.auth
```

# Fix It Yourself: Generate db



- db key pair can also be generated by a similar openssl call:

```
openssl req -new -x509 -newkey rsa:2048 -subj
"/CN=Image Signing Key/" -keyout db.key -out db.crt
-days 3650 -nodes -sha256
```

- use cert-to-efi-sig-list to convert db.crt into db.esl:

```
cert-to-efi-sig-list -g "$(uuidgen)" db.crt db.esl
```

- to add multiple certificates into db, concatenate their ESL files:

```
cat db.esl MsWin.esl UefiCa.esl > db.esl
```

- combined db.esl must be signed by KEK using sign-efi-sig-list:

```
sign-efi-sig-list -k KEK.key -c KEK.crt db db.esl
db.auth
```

# Fix It Yourself: Sign your bootloader



- sign an UEFI Shell binary by db key using sbsign utility:

```
sbsign --key db.key --cert db.crt --output
bootx64.efi shellx64.efi
```

- prepare a FAT32-formatted USB flash drive with this signed UEFI Shell in /EFI/Boot, copy UpdateVars.efi from /usr/share/efitools/efi onto it:

```
cp bootx64.efi /media/UsbFlash/EFI/Boot/
cp /usr/share/efitools/efi/UpdateVars.efi
/media/UsbFlash/
```

- also copy all *.auth files you've prepared on previous steps:

```
cp *.auth /media/UsbFlash/
```

# Fix It Yourself: Enable SecureBoot



- go to BIOS Setup and set a good supervisor password
- insert the prepared USB flash drive and try to boot from it with SecureBoot disabled, it should boot to UEFI Shell
- go to Security, select Clear All SecureBoot Variables option, save and exit BIOS Setup
- boot from prepared USB flash drive and execute:

```
UpdateVars db db.auth
UpdateVars KEK KEK.auth
UpdateVars PK PK.auth
```

- try to execute the last command again, it should fail with “security violation” message
- if so, we have SecureBoot armed and ready
- reboot and test if the signed UEFI shell is bootable, it should be

# Fix It Yourself: Remove UnlockPwd



- now we are in control of the boot process, but only if BIOS password is not known to the attacker and can’t be reset
- Acer uses UplockPwd driver for resetting forgotten BIOS passwords
- if the driver is removed, BIOS will crash after 3 failed password attempts

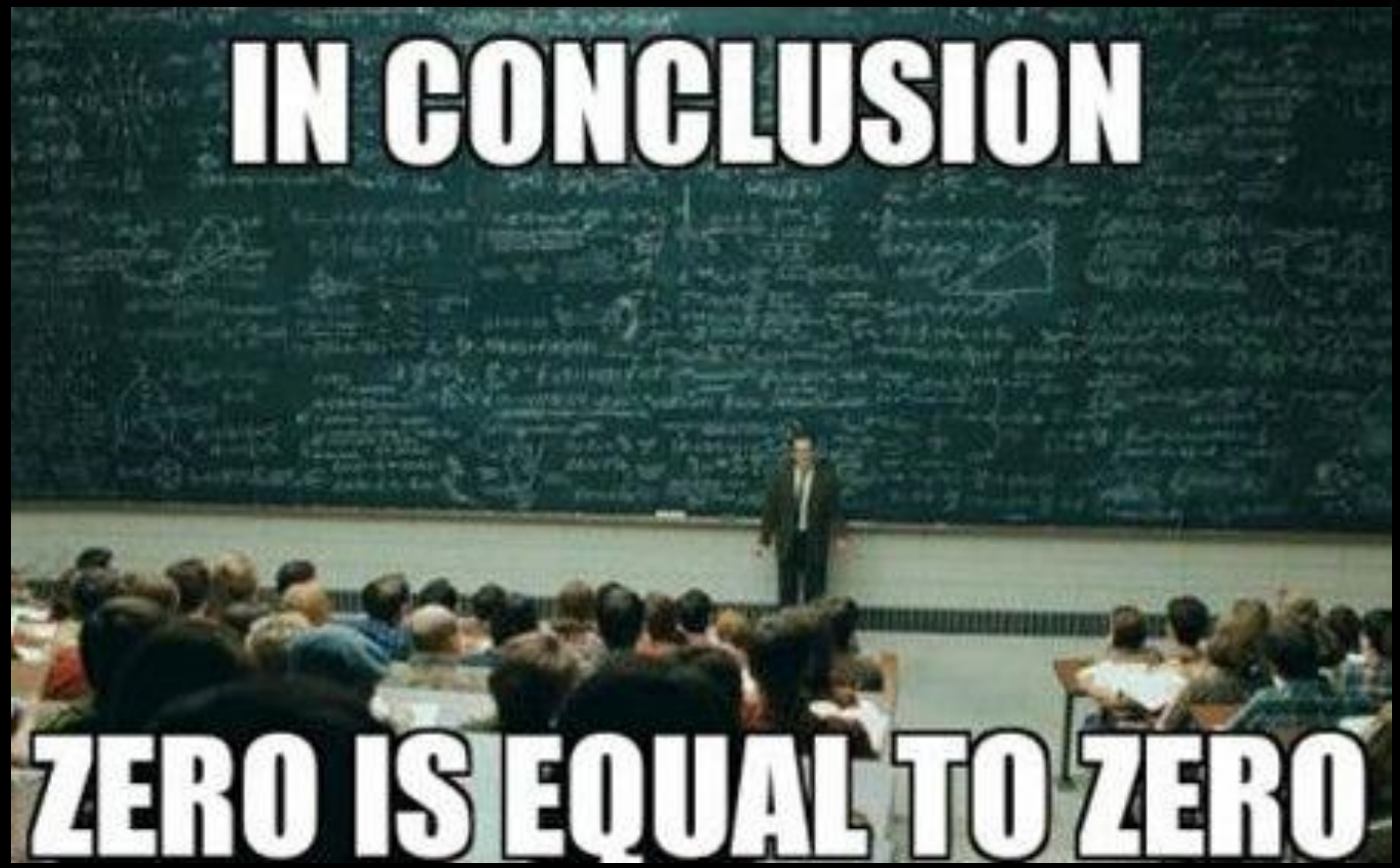
# Fix It Yourself: Remove other stuff



- remove WPBT driver
- remove flash drivers: PfatServices, Ihisi, SecureFlashDxe,
- remove CapsulePei, replace CapsuleDxe with a dummy driver that overrides UpdateCapsule and QueryCapsuleCapabilities to return EFI_UNSUPPORTED
- remove unused protections: BiosProtect, PchBiosWriteProtect
- remove network stack: NetworkLocker, DhcpDummy, Dpc, Mnp, Arp, Snp, Ip4, Ip4Config, Udp4, Dhcp4, Mtftp4, RtkUndiDxe
- remove ME-related drivers: MeFwDowngrade, BiosExtensionLoader
- remove all other stuff you don’t want or need, if the firmware can still boot your OS - it’s fine to have that components removed

# Conclusion





# Conclusion: Summary



- PC is still working and can boot an UEFI-compatible OS
- after ExitBS:
  0. BIOS is read-only for all execution modes including SMM
  1. BIOS password storage is inaccessible
  2. NVRAM is emulated, no changes can persist across reset
- SecureBoot works and allows executing only the images we sign
- ACPI S3 is disabled for good
- Intel FPT utility can be used for further BIOS updates, but it can only be performed if BIOS password is known
- UEFI network stack is no more

# Conclusion: Issues remaining and new



- SMM is still vulnerable, but breaching into it grants no persistence
- BootXXXX variables can’t be created by OS, we have to add new bootloaders manually
- ACPI S3 doesn’t work, but it can be replaced by S4 on fast SSD
- platform security relies upon PR registers, BIOS password, SecureBoot implementation and bootloader, a breach in any of them renders it useless
- network boot is not working anymore
- signed UEFI Shell we’ve prepared for SecureBoot testing is a skeleton key for the platform and must either be kept secret or securely removed

# Conclusion: What else?



- try to contact Acer once again and push them to fix the mess
- or throw the system out and don’t buy from them anymore
- fixing security vulnerabilities without sources is hard, but possible
- the vendor who care can fix them much faster and much better
- BIOS region can be made read-only by hardware SPI protection, but it requires two SPI chips to be designed in during board manufacturing
- the idea of having NVRAM in the same flash chip as BIOS was bad from the beginning and gets worse every day
- AMD-based platforms may be vulnerable too (harder to test without CHIPSEC)
- publish the sources, patched binaries and slides on GitHub

## Q&A



# THANK YOU FOR your ATTENTION! ANY QUESTIONS?