Global Information Assurance Certification Paper

Analysis of the building blocks and attack vectors associated with the Unified Extensible Firmware Interface (UEFI)

GIAC (GREM) Gold Certification

Author: Jean-François Agneessens (jean.agneessens@ncirc.nato.int)
Advisor: Manuel Humberto Santander Pelaez

Accepted:

Abstract

While Operating Systems have seen tremendous and very visible developments, driven by the evolution of hardware components, there are still some remnants from the 8086-era, one of which is the BIOS. Led by a consortium of vendors, the industry is now implementing a new style of BIOS which, by design, appears to overcome all the issues introduced by the Intel 8086 engineering decisions back in 1978.

The Unified Extensible Firmware Interface (UEFI), replacement of the legacy BIOS, is a blank-sheet design based on modular pieces of code following the well-known Portable Executable/Common Object File Format (PE/COFF), found on all Microsoft OS-based executable code. The UEFI code can therefore be reverse-engineered using similar techniques learned during GREM. The concepts of UEFI, and some of its VMware implementation, are presented here, as well as an insight into the possible paths open for further exploitation of the extended capabilities offered by UEFI.

1. Introduction

The Basic Input/Output System (BIOS) is the code that is the closest you can get to the underlying hardware. Its role, since its inception in the early stage of the Intel 8086 era, has been to detect and initialize surrounding components, to prevent conflicts in those components, and to allow the Operating System (OS) to boot (Note: before the advent of Windows 95, the Disk Operating System (DOS) made constant use of the BIOS for all access to peripherals, such as keyboard, floppy disk drives, screens and printers). The DOS was running in 8086 mode (also called the real-address mode), which used a 20-bit addressing scheme to access up to 1MB of memory, sliced in 64KB blocks, called the segments. The DOS was a single user OS, running in ring-0, where the interface offered unrestricted access to the whole hardware (Figure 1).

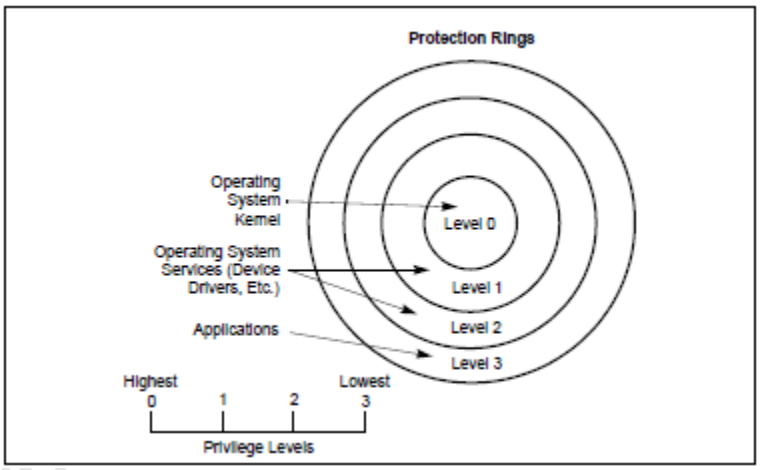


Figure 1: Privilege levels, or rings (Intel Press, 2011)

The evolution of computer hardware, their greater affordability, and the desire for GUI-based OS by the masses, were key in prompting OS vendors to make a complete abstraction of the BIOS after the Initial Program Load (IPL) (Compaq, Phoenix, Intel, 1996), known in UEFI terminology as the Boot Device Selection (BDS). The gap between the BIOS (made for 16-bit real-address mode OS), and the hardware/OS currently available, grew over time, to the moment were a decision had to be made with the introduction of Intel's new generation of 64 bit processors, the Itanium (Vincent Zimmer, 2010).

Intel decided to start building, from scratch, a new type of BIOS; the Extensible Firmware Interface (EFI), and this modular approach was adopted by other vendors, notably Apple, and eventually became known as the Unified EFI (UEFI). The UEFI is meant to be developed in C, with supporting libraries and a developer kit available for free, and while it will no longer be able to boot DOS, it does allow more freedom in what it offers, and is more tied with the OS (as in BIOS/DOS). Additionally, the presence of UEFI runtime services is also a topic of interest.

2. The BIOS, or the end of an era

2.1. Intel CPUs' mode of operations

Several modes of operation have been defined in the x86 architecture (Intel Press, 2011), and the default one at startup, called the real-address (or real) mode operates in 16bit mode, where only 1MB of memory can be addressed, and the memory is spliced into blocks, called segments. This is due to the inherent design of the 8086, where a 20bit addressing scheme is used. The 16 lower bits allow 64KB addressing (2^{16}) and the 4 extra bits provided by the Code Segment register (CS) shifted 4 bits to the left to allow inter-segment addressing (Figure 2).

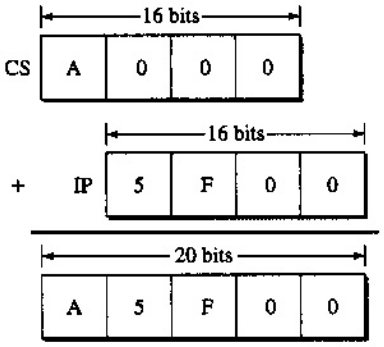


Figure 2: 20-bit addressing (Kholodov, 2007)

The next operating mode of interest is called the protected mode. This is a 32bit mode, for which memory addressing can be flat or segmented, and for which paging can optionally be used. In *the basic-flat model*, the Code Segment register (CS) and the Data segment registers (DS,ES,FS,GS,SS) all start at 0x0 and end at 0xFFFF_FFFF, which

means they overlap. Data can, therefore, be interpreted as code, depending which register is used to access the address. The *protected-flat model* follows the same principle, but will have an upper limit set to the real size of the DRAM. The multi-segment model allows segregation of memory area to protect applications from each other (non-overlapping address space) and typically makes sense for multitasking environments. For each case, paging can be used, which allows allocation of more memory than what is physically available and which, again, makes sense in multitasking environments.

The actual mode of the processor is stored on a special register called CR0. This register is composed of bits that are set high or low. At INIT# or RESET# (the 2 ways of bringing the computer to its initial stage), CR0 is set at 0x6000010, the real-address mode (Figure 3).

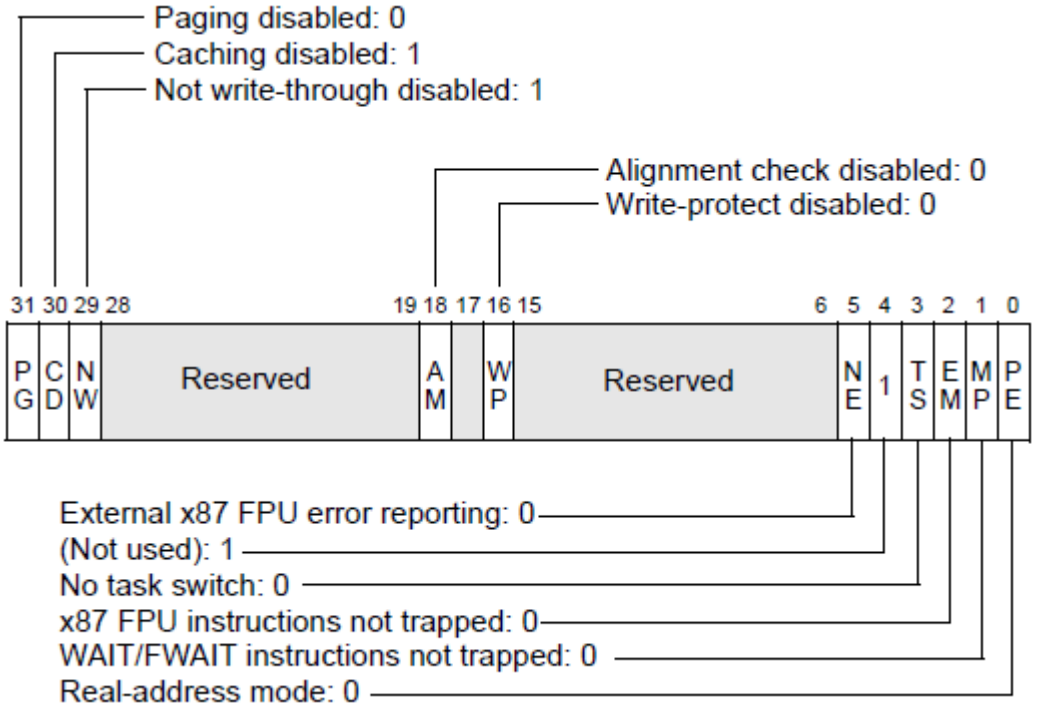


Figure 3: CR0 register (Intel Press, 2011)

The descriptor tables are the link between the linear memory addressing and the segmented memory addressing. They are composed of quad-words (8 bytes) elements, each of which will define a register, its start address, length, property bits and purpose. The mandatory descriptor table is called the Global Descriptor table, and is composed, at

the minimum, of a NULL descriptor (a sled of 0x0), a CS descriptor and a DS descriptor (Figure 4).

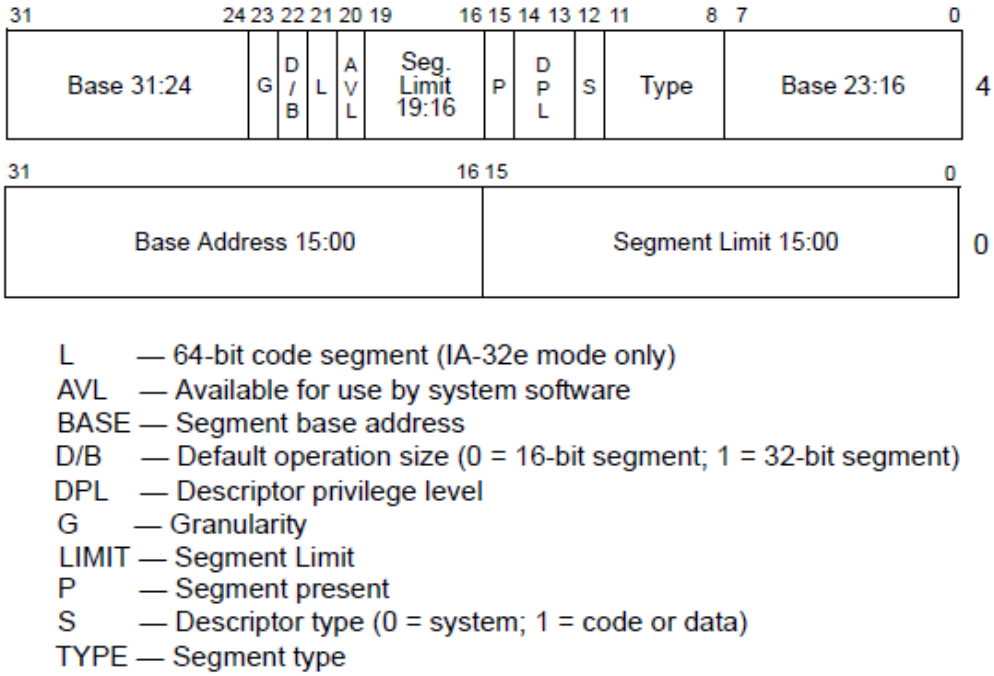


Figure 4: Descriptor Table (Intel Press, 2011)

To allow the processor to switch from real-address mode to protected-mode, several prerequisites have to be met, and (Intel Press, 2011) mentions a set of tasks to be executed in the right order: Disable interrupts, set the GDT address on the GDT Register, set CR0 to protected mode and execute a far jump.

The explanations above are only a small subset of what Intel's systems are providing, but this covers the prerequisites to go further. See Figure 5 for an overview in yellow of what has been covered.

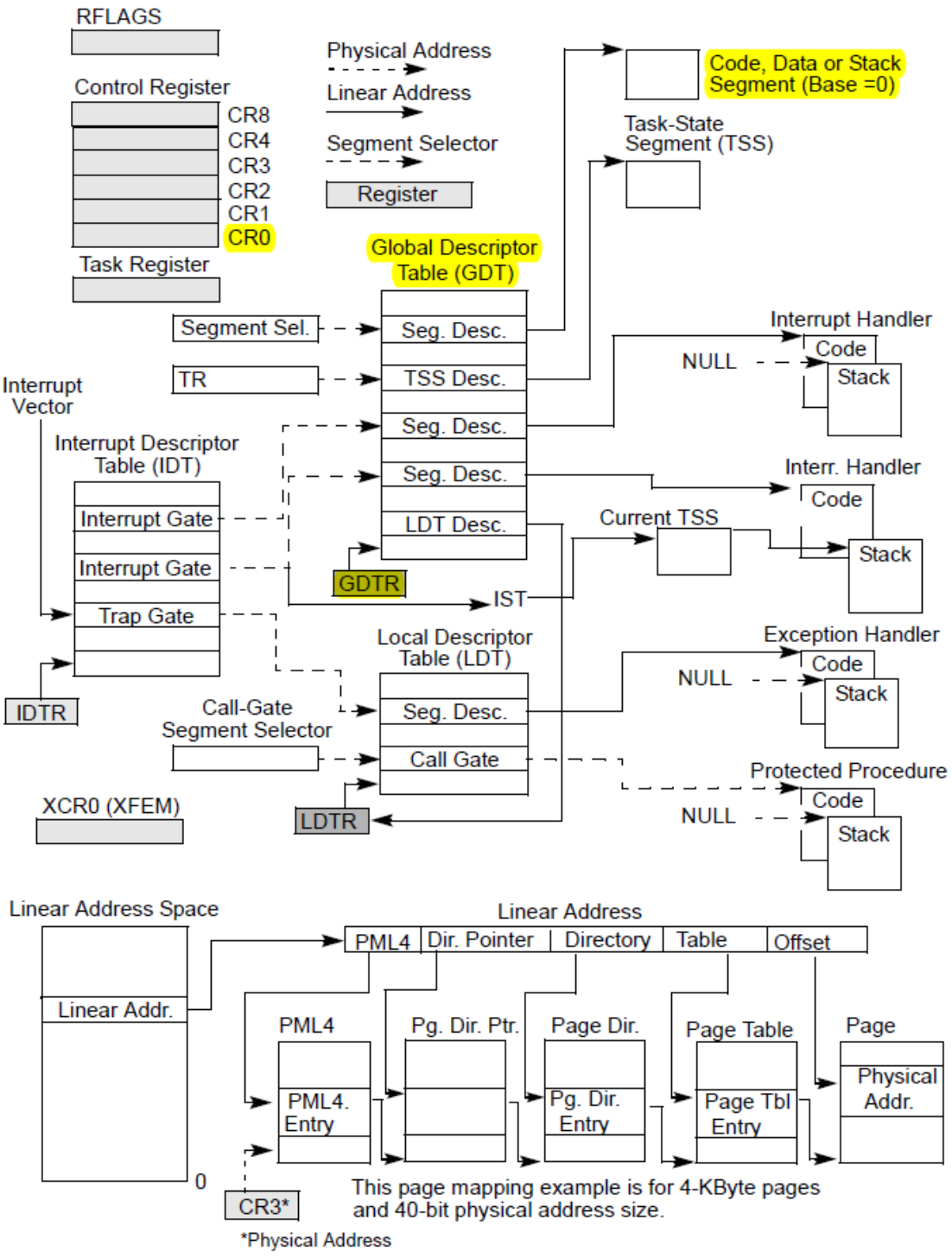


Figure 5: System-level Registers and Data Structure (Intel Press, 2011)

2.2. What happens at RESET# or INIT#

When the computer is powered on, it will, by default, look at the global address 0xF_FFF0, and known as the Host Reset Vector (HRV); at this time, the system is in real-address mode.

```
FFFF:FF8F                   db  20h
FFFF:FF90 aAllRightsReser   db 'All rights reserved.'
FFFF:FFA4                   db    0
FFFF:FFA5                   db 0Bh dup(0FFh)
FFFF:FFB0 aDellSystem760    db 'Dell System 760'
FFFF:FFBF                   db 21h dup(0)
FFFF:FFE0 aA02A02           db 'A02¦■',2,0,1,'A02¦■',2,0,1
FFFF:FFF0 ; ---------------------------------------------------------
FFFF:FFF0                   jmp     HRV_jump
FFFF:FFF0 ; ---------------------------------------------------------
FFFF:FFF3                   db 0Dh dup(0FFh)
FFFF:FFF3 FFFF              ends
FFFF:FFF3
FFFF:FFF3
```

Figure 6: HRV Jump on Dell Optiplex 760 A02 BIOS

The HRV points, by design, to the address 16 bytes before the 4GB upper limit and as this address is constructed by using the Code Segment (CS) register as high address (which by default will be at value 0xFFFF at INIT), and the Instruction Pointer (IP) as low address (0xFFF0 at INIT), there is little room for anything less than a jump to a smaller address to continue executing the early initialization code.

At this stage of the power-on process, the CPU is not aware of its surrounding environment (what amount of memory can be used, what are the expansion buses, the peripherals, the ports, the expansion cards, the other CPUs), so the role of the NorthBridge chipset is to redirect the HRV address to the BIOS code (through the Southbridge) stored on the NVRAM. How this happens is hardware specific, but on fairly recent chipsets it is done by means of the Programmable Attribute Maps Registers (PAM), which control if shadowing occurs for certain memory ranges or not. In actual fact, the global addressing is not only used to access the memory, but the whole set of peripherals that are surrounding the CPUs. The NorthBridge is the chipset that is redirecting to the different hardware, and in some cases, one global address can be redirected differently depending on the state of some Northbridge registers. On INIT#, it

will shadow the range 0xF_0000-0xF_FFFF to the NVRAM, whereas later on in the boot process, it will most likely be pointing to the RAM, as executing from RAM is faster than doing it from NVRAM (Dice, 2011).

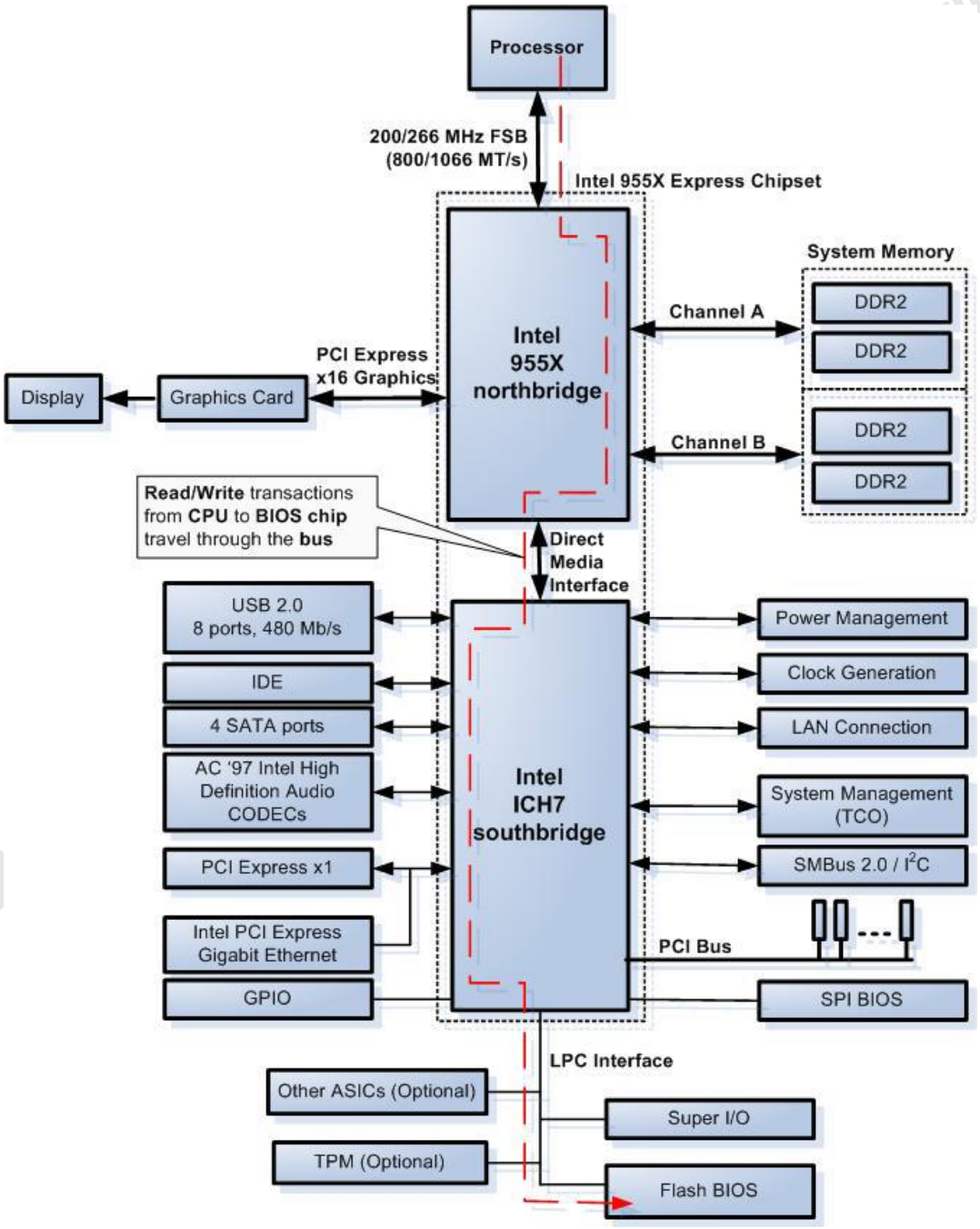


Figure 7: Intel 955X-ICH7 chipsets (Salihun, 2007)

Because the HRV is pointing at 0XFFFF_FFF0, there must be an alias between 0xF_FFF0 (1MB-16 bytes) and the HRV.

The actual jump address will depend on the physical size of the BIOS chip. Depending on the version, these chips can have size up to 16Mb (2MB), which would then place the start of the NVRAM below 0xE_0000 (in fact, so big that the NVRAM could not be copied to the 1MB addressable memory), but the problem is, by design, only 128K has been allocated for the BIOS. The memory below is reserved for Option ROM (expansion card ROM), the VGA ROM and the DOS (Figure 8), therefore only a part of the NVRAM will be accessible from the INIT#/RESET#. This means the jump will most likely go anywhere within the last 64K of the NVRAM area and depends on how the BIOS vendor implemented its code.

(Dice, 2011) covers the boot flow of a modern Intel System and indicates that the boot code's goal is to switch as early as possible in protected basic flat mode with no paging. (Salihun, 2007) extensive legacy BIOS reverse engineering explains that BIOS ROM are made up of components that are compressed and added next to each other because the limited size of NVRAM chipset made compression mandatory to hold all the code (chipsets used to be 64KB or 128KB). Therefore, after having switched to protected mode, the system will need to initialize the memory (controlled by the NorthBridge), initialize the SouthBridge (to access the Low Pin Count (LPC) interface), and then execute the code that will allow decompressing the different components of the Flash. It can now copy itself into RAM (for faster processing of the code), initialize the Root Complex Register Block (RCRB) (which allows addressing the PCI(-e) bus and peripherals), which in turn gives the ability to look for expansion cards and their firmware (called the Option ROM). This part of the boot process is particularly tricky, as all the cards firmware need to be held in 128K of memory (and the results of heavily populated expansion buses usually resulted in the BIOS hanging frequently).

According to (Compaq, Phoenix, Intel, 1996), the BIOS boot order is linked to two lists; the Initial Program Load (IPL), the famous A:,C: in a user-defined order, and the Boot Connection Vector (BCV), which points to a specific piece of code of an Option ROM. The BIOS proceeds, in the given priority, to try to boot an OS in a sequential

manner. This also implies that any failing boot device should return the control to the BIOS, and this handover back to the BIOS is made with INT 18h. The way the BIOS gives control to a given IPL boot device is by calling INT 19h. but in the case of BCV (aka Option ROM) boot devices it will be done by calling INT 13h. The Option ROM, when executed, will hook INT 13h, and this interrupt is called before trying to start the OS boot and gives the opportunity to the card to provide the first hard drive (C:), known as 80h. The complexity arises when several Option ROMs are available and a chaining of INT 13h devices occurs, so the first Option ROM that can provide the drive 80h become, therefore, the "C:" in the IPL discussed above. This process is not trivial and lead to a lot of frustrations in the past.

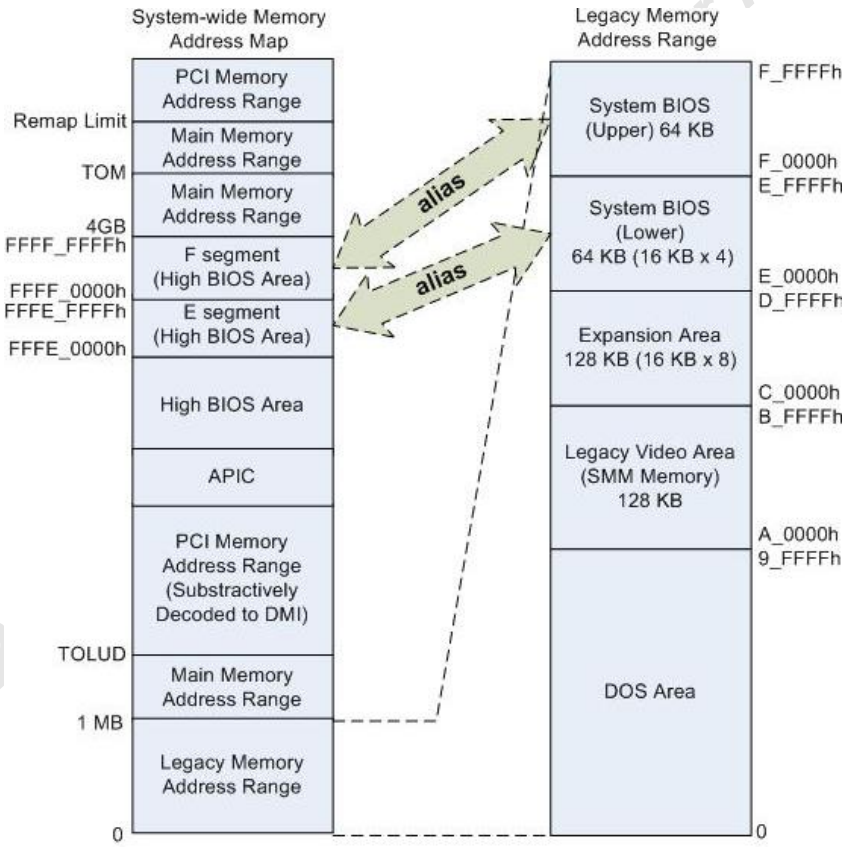


Figure 8: Memory mapping and legacy (DOS) memory area (Salihun, 2007)

3. UEFI, future-proof by design

3.1. Concepts and overall structure

It must be pointed out that using UEFI as the term to describe the replacement of the legacy BIOS is not entirely correct, as another set of reference documents (PI Working Group, 2012) exists to describe the Platform Initialization (PI). PI is a child of UEFI, and the intent for UEFI is to focus on the interfaces with the boot devices and the Operating System, while PI focuses on the initialization of system components (chipsets, memory, buses) and component drivers (Option ROM in legacy BIOS terminology). This means that both specifications are important, but PI is to be referred to first until the hardware is brought into a UEFI compliant mode. These frameworks are complementary (Figure 9), where PI is covering the yellow area and UEFI is covering the pink one. Most of the information provided here is based on the book (Vincent Zimmer, 2010) and the PI/UEFI specifications: (PI Working Group, 2012) (UEFI Spec Working Group, 2012).

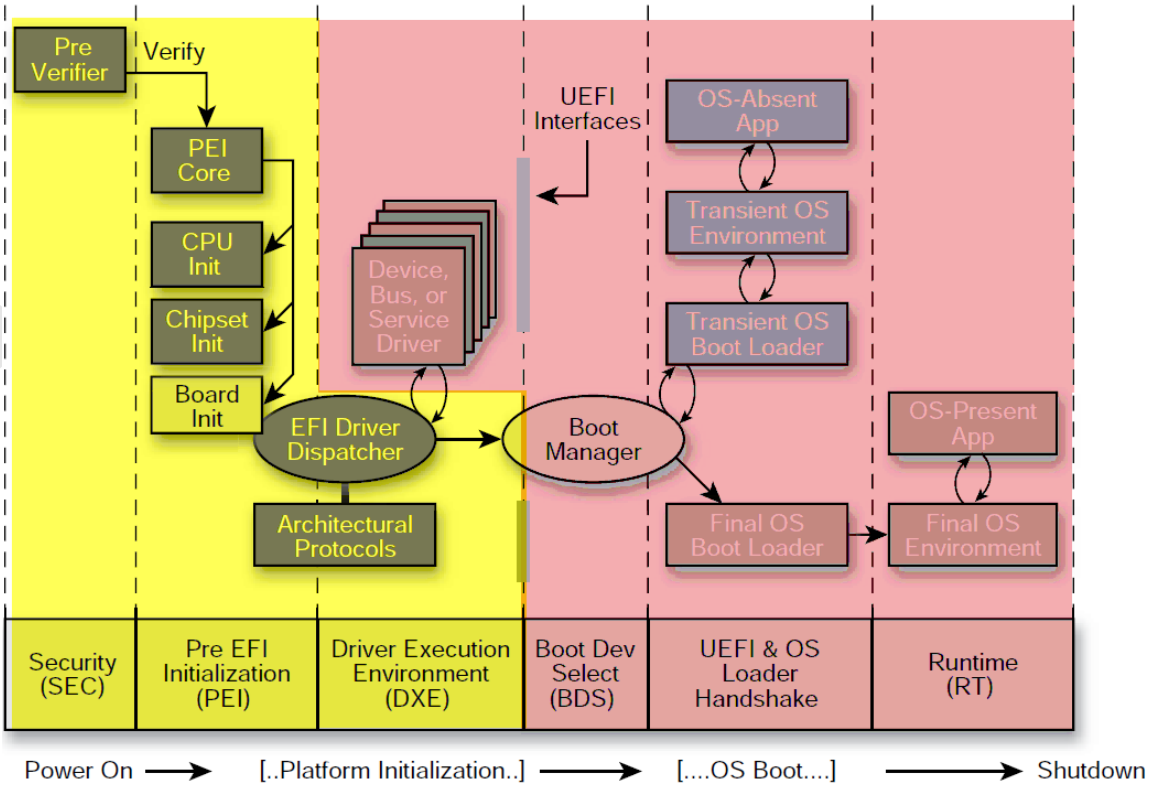


Figure 9: PI and UEFI coverage (Vincent Zimmer, 2010)

PI is made up of Security (SEC), Pre-EFI Initialization (PEI) and Driver Execution Environment (DXE). UEFI is composed of the Boot Device Selection (BDS), the Transient System Load (TSL) and the Runtime (RT).

Some concepts and terminology need to be described before jumping into the sub-components. A *handle* can be compared to a class in Object Oriented Programming, and is a collection of one or several *protocols*; it is managed in the *Handle Database*. A *protocol* is defined by a *GUID*, and can contain function pointers and data structures. It can be used as an interface for hardware, as well as an interface with other protocols. Examples of protocols are: *UsbAtapi*, *VgaMiniPort, TCP/IP*. Examples of functions could be *ClearScreen()* or *OpenVolume()*. The functions mentioned above are provided by *UEFI drivers* which are a subset of the different types of *UEFI Images*, and Images can also be *UEFI applications* or *OS loaders* and will be covered later.

The core of UEFI is called the *System Table* (Figure 10). It contains references to the available protocols as well as two other tables, the *Boot* and *Runtime (RT) Service Tables*. *Boot services* are available until RT is achieved, and RT is called at the end of TSL, when the OS Loader UEFI Image calls the boot service function *ExitBootServices()*. *Runtime services* are also available when the OS is running, meaning there are interfaces available between UEFI and the OS. The list of boot and runtimes services is fixed by the UEFI specifications revision, but can be extended above that minimum list, and the remaining protocols to be found in the System Table are modular elements added by the hardware vendor or the firmware manufacturer, hence the meaning of "extensible" in UEFI.

A last table referred by the System Table is called the *System Configuration Table*. It points to other tables, notably the *Advanced Configuration and Power Interface (ACPI), the SMBIOS Table*, the *Hands-Off Block Table (HOB)* and *DXE Services Tables*.

The phases are explained more in details below, and the extensive terminology will likely be better understood after reading these paragraphs.

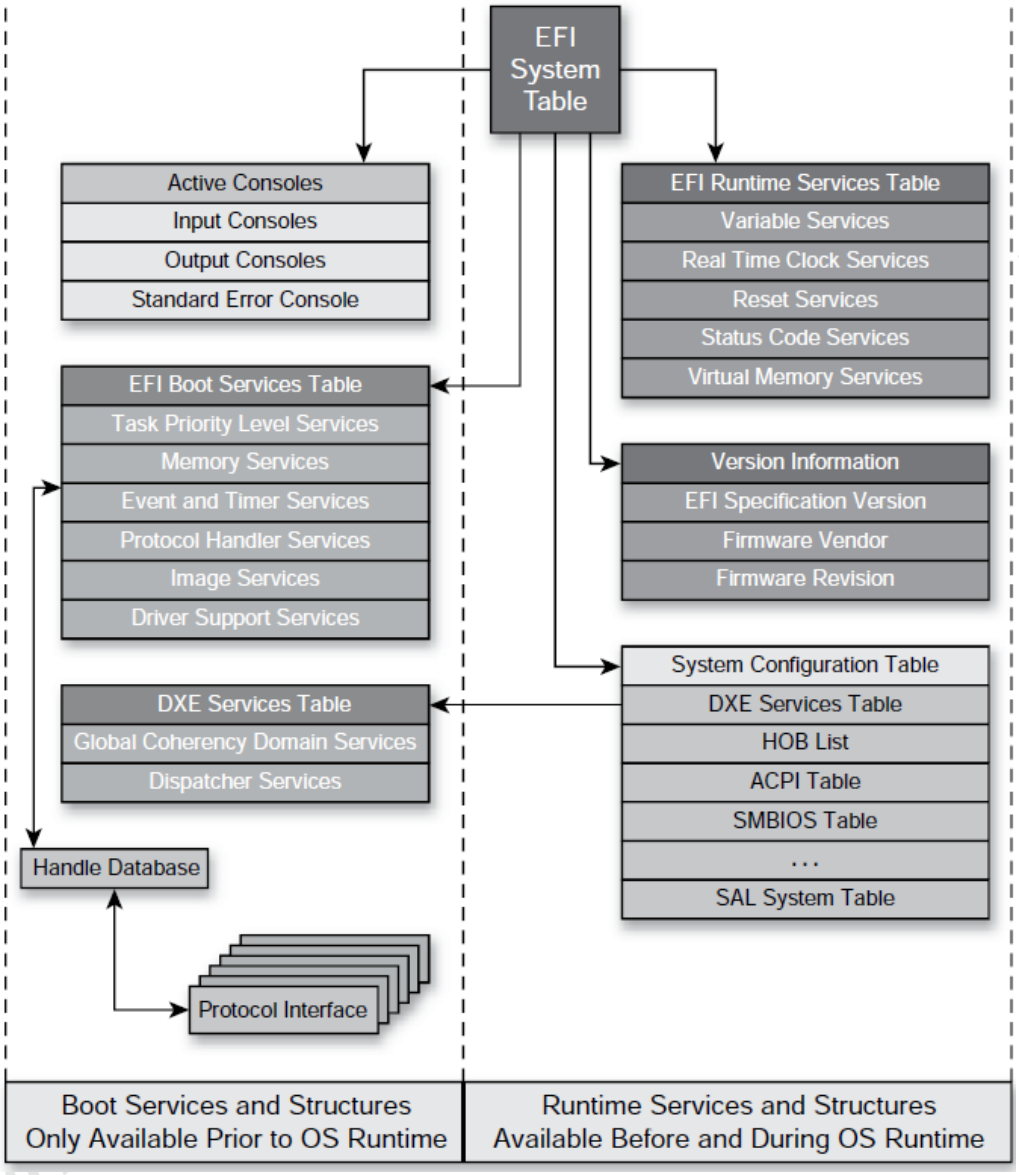


Figure 10: UEFI System Table structure (Vincent Zimmer, 2010)

3.2. SEC and PEI

The SEC phase has four responsibilities, as described in (PI Working Group, 2012). It handles the platform restart events (there are 11 "boot modes" described in the PI specifications, notably "BOOT_WITH_DEFAULT_SETTINGS", "BOOT_ON_FLASH_UPDATE", "BOOT_WITH_FULL_CONFIGURATION", "BOOT_ON_S3_RESUME", etc). Next, SEC takes care of creating a temporary memory store by using the processor cache as a flat memory. This is known as Cache-as-RAM (Salihun, 2007) in Legacy BIOS terminology. SEC can serve as the Root of Trust (hence

its name), but this phase is optional. This is typically how the link with the Trusted Platform Module (TPM) will happen and is the base of the Secure Boot. Lastly, SEC will hand over the pointers to the temporary memory, the temporary stack and the *Boot Firmware Volume* (BFV) to PEI.

The role of PEI role is to initialize the memory for the DXE phase to start, and as such its role is very limited, but due to the overall stage of initialization of the system it will rely on code that should be very similar to what can be found on legacy BIOS, with the main difference being the format of the code itself, explained below. PEI makes use of specialized drivers called the *PEI Modules* (PEIM) which are contained in the BFV provided by SEC and the format in which they are encoded follows the Flash File System (FFS) described in PI. Each PIEM is, therefore, a file encoded in PE/COFF (MZ then PE headers) or in the *Terse Executable* format (TE) which will have a VZ (Vincent Zimmer) header instead of the MZ (Mark Zbikowski) header. TE is a subset of PE as an answer to the limited amount of resources available at such a stage of the boot process.

Hands-Off Blocks are data structures containing the state of the system. It is received as the only input for the DXE phase, explained below.

3.3. DXE

The role of the Driver Execution Environment is to initialize the systems components (chipsets, add-on cards) and to hand over to a specific *Architectural Protocol* (see below) called the Boot Device Selection. To meet these requirements, the DXE is composed of three elements, with the first one being the *DXE Core*, and its goal is to produce the Boot, Runtime, DXE Services, to populate the EFI System Table and to create the Handle Database. The DXE Core is the receiver of the HOBs List, from PEI, and consumes Architectural Protocols. The difference between a *Protocol* and an *Architectural Protocol* is the dependency of hardware for the later one.

The question is, how is it expected to work? In the HOBs lists, one HOB (or more) contains the description of the firmware volume (The PI FFS for instance), and this HOB has a specific GUID, which is linked to a DXE driver, the second of the three DXE elements.

The last element, called the DXE Dispatcher, is the one which links the HOB GUID to a specific DXE driver, so there must be a HOB related to a FFS DXE driver that, when executed, will read the volume looking for a specific file called the *apriori*, which is an authoritative list of DXE drivers to execute in order.

The *apriori* file is not mandatory, and if not present, the DXE Dispatcher will search for DXE drivers in the volume, look at their dependencies, and then decide the execution order. The DXE Dispatcher is also responsible for looking for DXE drivers related to the other HOBs, which themselves can relate to other volumes, containing more DXE drivers, and it is the versatility of this system that permits common file systems like "FAT" to be recognized so that in the GUID Partition Table (GPT) the 200MB hidden system partition is exactly that; a partition recognized at the DXE (or BDS) phase that can potentially contain DXE drivers.

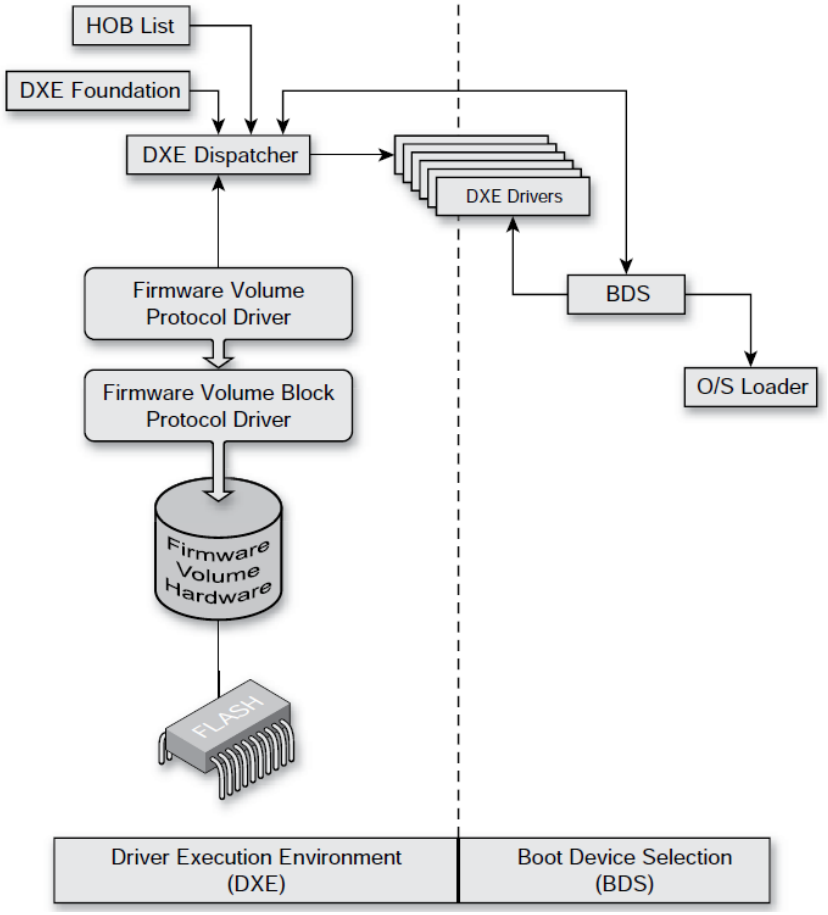


Figure 11: Driver Execution Environment (Vincent Zimmer, 2010)

There are two categories of DXE drivers; the initial ones loaded at the early stage (which should not be very different from legacy BIOS style code), and the true UEFI drivers (which are compliant with the specifications). All of them are in the PE/COFF format (or PE32+, if they include digital signature) and the true UEFI drivers can also be coded in *EFI Byte Code* (EBC), which is platform independent but runs on top of an EBC Virtual Machine, as described in (UEFI Spec Working Group, 2012). The DXE phase can, therefore, be compared to the Option ROM loading with legacy BIOSs, but in a much more controlled way.

3.4. BDS, TSL and RT

When all of the mandatory Architectural Protocols defined in (UEFI Forum, 2012) are available (that is, the Boot and Runtime services tables are filled), the DBS Architectural Protocol is executed and has to follow a strict policy: Initialize the elements required for human interaction (to allow entering Setup, a boot menu,…), load all drivers stored in a specific variable, and attempt to boot every item contained in the *BootOrder* variable. If any of these steps fails, the BDS gives control back to the DXE dispatcher (Figure 11), and the dispatcher will try to find alternative DXE drivers in the newly discovered firmware volumes resulting from the driver load of BDS.

For an OS to boot, the BDS will look for a platform specific file (Figure 12), this file will always be in the **/EFI/BOOT/BOOT/** folder, and once found the OS loader will call ExitBootServices(), which only allows the Runtime Services to stay resident.

	File Name Convention	PE Executable Machine Type *
32-bit	BOOTIA32.EFI	0x14c
x64	BOOTx64.EFI	0x8664
Itanium architecture	BOOTIA64.EFI	0x200
ARM architechture	BOOTARM.EFI	0x01c2
Note: * The PE Executable machine type is contained in the machine field of the COFF file header as defined in the Microsoft Portable Executable and Common Object File Format Specification, Revision 6.0		

Figure 12: GPT Boot file name (UEFI Spec Working Group, 2012)

In addition to the capability of booting an OS, the BDS allows UEFI applications to be run, one of which is the UEFI shell, which mixes DOS and UNIX notations. In

VMware, it can be accessed by selecting EFI BIOS for a new virtual machine and not creating any hard drive. The system will fail boot in the UEFI shell.

4. Analyzing VMware UEFI 64 bit implementation

4.1. Extracting the ROM file

VMware has been emulating the chipset Intel 440BX and has, therefore, been using an implementation of a legacy BIOS based on that chipset, but since VMware Workstation 8/ESXi 5 the option exists to allow you to boot with a UEFI firmware.

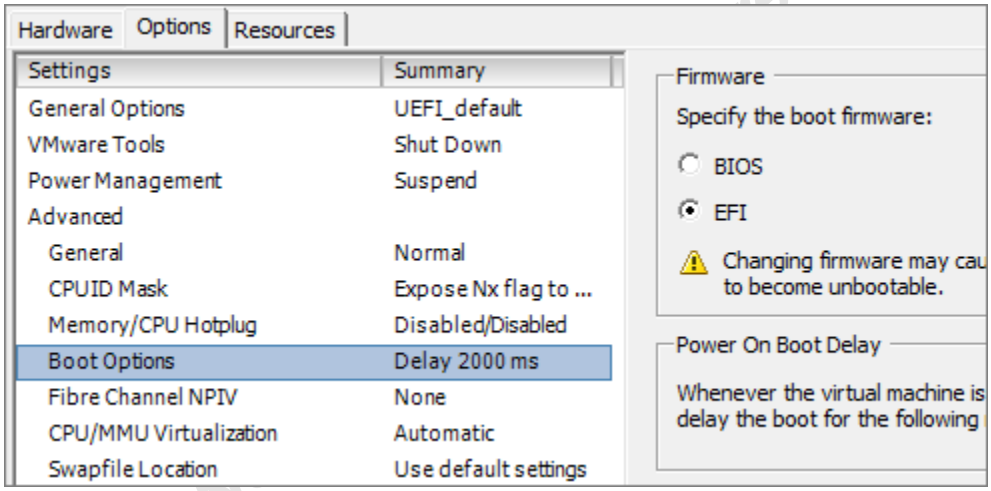


Figure 13: VMware UEFI boot option

The firmware itself is embedded in the file **/bin/vmx** of the hypervisor, and by using **objdump** and **objcopy**, the UEFI firmware can be extracted.

```
citau@backup:~$ objdump -h vmx

vmx:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .interp       0000001c  0000000000000270  0000000000000270  00000270  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag 00000020  000000000000028c  000000000000028c  0000028c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .hash         00002b5c  00000000000002b0  00000000000002b0  000002b0  2**3
                  CONTENTS, ALLOC, LOAD, READONLY
 31 efi32         000eaa70  0000000000000000  0000000000000000  00989922  2**0
                  CONTENTS, READONLY
 32 efi64         000e5408  0000000000000000  0000000000000000  00a74392  2**0
                  CONTENTS, READONLY
 33 nvram         00000324  0000000000000000  0000000000000000  00b5979a  2**0
```

The object 32 can be extracted and uncompressed by using zlib (939016 is the size of the compressed ROM file).

```
objcopy vmx -O binary -j efi64 --set-section-flags efi64=a efi64.rom

perl -e 'use Compress::Zlib; my $v; read STDIN, $v, 939016;
                $v = uncompress($v); print $v;' < efi64.rom > efi64.rom.2
```

The ROM studied here has a MD5 hash of **dbc1a58988c150cb1eb95c04dcf06a1b**

4.2. Preliminary Analysis

The uncompressed file is 2MB, and running strings on it shows some interesting information. DLL names are self-explanatory for UEFI-aware people and PE structures are to be found.

```
citau@backup:~$ strings efi64.rom.2 |grep dll
SecMain.dll
PeiCore.dll
PcdPeim.dll
StatusCodePei.dll
PlatformPeim.dll
DxeIpl.dll
DxeCore.dll
```

Another topic worth mentioning is the very limited size of the last 64KB segment. All the code and supporting data is contained between 0xFFDD0 and 0xFFFFF, which corresponds to 559 bytes. The segment corresponding to 0xE_0000 is empty as well, and analysing the legacy VMWare BIOS 440 shows that most of the last 128KB is not null.

4.3. From Power Up to PEI

As stated earlier, the system has to be in real-address mode, and the last 64KB of the ROM are matching the addresses 0xF_0000 and 0xFFFF_0000 (the latter one is meaningless before jumping in protected mode).

The options to be set when loading the file in IDA Pro are to set the CPU to 80x86:metapc, and to force the decoding in 16bit. When the file is loaded, segments must be created and properly allocated. Therefore, the following IDC script has been used.

```
auto ea,ea_src,ea_dst;
for(ea = 0x0; ea < 0x200000; ea = ea + 0x10000)
{
        SegCreate(ea, ea + 0x10000, ea>>4, 0, 0, 0);
}
ea_src=0x001F0000;
ea_dst=0xFFFE0000;
SegCreate(ea_dst, ea_dst + 0x10000, ea_dst>>4, 0, 0, 0);
for(ea_dst; ea_dst <= 0xFFFFFFFF; ea_dst = ea_dst + 4)
{
        PatchDword(ea_dst, Dword(ea_src));
        ea_src = ea_src + 4;
}
```

The entry point starts at 0xFFFF_FFF0 (Figure 14). The code running in 16bit is exactly following what (Dice, 2011) presented and mentioned earlier in 2.2. Enabling the

A20 Gate is required to switch to protected mode, and while out of scope here, the history of the A20 spare pin is worth reading to understand how legacy decision are impacting current architecture (Brouwer). Note that the value of DI, set in real-address mode, will later be used after the switch to protected mode for a comparison. The instruction to load the Global Descriptor Tables complies with 2.1. With CS and DS starting at 0 and covering the whole memory range, this is according to the requirements of the basic flat protected mode (see Figure 15). Note that the Code Segment is defined twice: one in 32bit, once in 64bit mode. The 32bit version of the VMWare UEFI ROM presents exactly the same GDT.

```
F000:FFF0 ; ---------------------------------------------------------------------------
F000:FFF0
F000:FFF0 HRV_jump:                               ; flush cache
F000:FFF0                 wbinvd
F000:FFF2                 jmp     A20Enable
F000:FFF5 ; ---------------------------------------------------------------------------
F000:FF80 ; ---------------------------------------------------------------------------
F000:FF80
F000:FF80 A20Enable:                              ; CODE XREF: F000:FFF2↓j
F000:FF80                 mov     di, ax          ; save AX in DI
F000:FF82                 mov     al, 2           ; AL=2
F000:FF84                 out     92h, al         ; Enable A20 (port 92h, command 02h)
F000:FF86                 mov     ax, di          ; restore AX
F000:FF88                 mov     di, 5042h       ; set 5042h to DI, likely for a later
F000:FF8B                 jmp     short SwitchToProtectedMode_1
F000:FF19 ; ---------------------------------------------------------------------------
F000:FF19
F000:FF19 SwitchToProtectedMode:                  ; CODE XREF: F000:SwitchToProtectedMode_3↓j
F000:FF19                 cli                     ; clear interrupt flag.
F000:FF1A                 db      66h
F000:FF1A                 lgdt    fword ptr cs:GDT ; Load Global Descriptor Table (GDT)
F000:FF21                 mov     eax, 40000023h  ; set bit 0,1,5,30 to 1 --> PE,MP,NE,CD
F000:FF21                                         ; Protected Enable
F000:FF21                                         ; Numeric Error (x87 FPU error handling)
F000:FF21                                         ; Monitor Coprocessor
F000:FF21                                         ;           (set x87 execution instructions)
F000:FF21                                         ; Cache Disable
F000:FF27                 mov     cr0, eax        ; switch to protected mode!
F000:FF2A                 jmp     large far ptr 10h:0FFFFFF32h ; switching to protected mode
F000:FF2A                                         ;  MUST be followed by a far jmp.
F000:FF2A ; ---------------------------------------------------------------------------
```

Figure 14: from real-address mode to protected mode

```
F000:FF60                db 0, 0, 0, 0, 0, 0, 0, 0 ; NULL descriptor
F000:FF68                db 0FFh, 0FFh, 0, 0, 0, 93h, 0CFh, 0 ; Data Segment (DS)
F000:FF68                                          ; base: 0x00000000
F000:FF68                                          ; limit: 0xFFFFFFFF
F000:FF68                                          ; Flags:
F000:FF68                                          ; - P (present) ,
F000:FF68                                          ; - S (Code/data),
F000:FF68                                          ; - Type 0011 (Data),
F000:FF68                                          ; - Granularity (4KB),
F000:FF68                                          ; - D/B (32bit)
F000:FF70                db 0FFh, 0FFh, 0, 0, 0, 9Bh, 0CFh, 0 ; Code segment (CS)
F000:FF70                                          ; Flags
F000:FF70                                          ; - P (present) ,
F000:FF70                                          ; - S (Code/data),
F000:FF70                                          ; - Type 1011 (Code),
F000:FF70                                          ; - Granularity (4KB),
F000:FF70                                          ; - D/B (32bit)
F000:FF78                db 0FFh, 0FFh, 0, 0, 0, 9Bh, 0AFh, 0 ; Code segment (CS)
F000:FF78                                          ; Flags
F000:FF78                                          ; - P (present) ,
F000:FF78                                          ; - S (Code/data),
F000:FF78                                          ; - Type 1011 (Code),
F000:FF78                                          ; - Granularity (4KB),
F000:FF78                                          ; - D/B (64bit) !
```

Figure 15: Global Descriptor Table

Because the firmware now needs to be studied in 32 bit mode, the file is reopened accordingly with IDA Pro. The entry point is 0xFFFF_FF32. After a few instructions, (setting some register at initial value and comparison of DI from real-address mode), the code is defining a value for a GUID.

```
seg000:001FFDF2 LookForGUID_boot:                       ; CODE XREF: seg000:001FFE05↓j
seg000:001FFDF2                                         ; seg000:001FFE0E↓j ...
seg000:001FFDF2                 sub     eax, 1000h      ; remove 1000h from eax
seg000:001FFDF7                 cmp     eax, 0FF000000h ; verify eax < 4GB
seg000:001FFDFC                 jb      short GUIDFindFailure ; if bigger, then we mis
seg000:001FFDFE                 cmp     dword ptr [eax+10h], 8C8CE578h
seg000:001FFE05                 jnz     short LookForGUID_boot
seg000:001FFE07                 cmp     dword ptr [eax+14h], 4F1C8A3Dh
seg000:001FFE0E                 jnz     short LookForGUID_boot
seg000:001FFE10                 cmp     dword ptr [eax+18h], 61893599h
seg000:001FFE17                 jnz     short LookForGUID_boot
seg000:001FFE19                 cmp     dword ptr [eax+1Ch], 0D32DC385h
seg000:001FFE20                 jnz     short LookForGUID_boot
seg000:001FFE22                 cmp     dword ptr [eax+24h], 0
seg000:001FFE29                 jnz     short LookForGUID_boot
seg000:001FFE2B                 mov     ebx, eax        ; GUID found
seg000:001FFE2D                 add     ebx, [eax+20h]
seg000:001FFE30                 jnz     short LookForGUID_boot
seg000:001FFE32                 jmp     GUIDfound
```

The GUID is: **8C8CE578-8A3D-4F1C-9935-896185C32DD3**. Using a Hex Editor, it can be found at address 0x10 of the binary ROM file. The next bit of interesting code is

the check for the MZ/VZ and PE headers. The figures below show a hexdump of the first bytes of the ROM file, and the corresponding code for header checks.

```
seg000:001FFED1 MZheaderCheck:                          ; CODE XREF: seg000:001FFEB8↑j
seg000:001FFED1                 add     eax, 4          ; add 4 to current eax address
seg000:001FFED6                 mov     ebx, eax        ; save eax on ebx
seg000:001FFED8                 cmp     word ptr [eax], 5A4Dh ; look for MZ header
seg000:001FFEDD                 jnz     short VZHeadercheck ; not a MZ header, see if VZ header
seg000:001FFEDF                 movzx   ebx, word ptr [eax+3Ch]
seg000:001FFEE3                 add     ebx, eax
seg000:001FFEE5
seg000:001FFEE5 VZHeadercheck:                          ; CODE XREF: seg000:001FFEDD↑j
seg000:001FFEE5                 cmp     word ptr [ebx], 5A56h ; look for VZ header
seg000:001FFEEA                 jnz     short PEHeadercheck ; not a VZ header, look for PE header
seg000:001FFEEC                 add     eax, [ebx+8]
seg000:001FFEEF                 add     eax, 28h ; '('
seg000:001FFEF4                 movzx   ebx, word ptr [ebx+6]
seg000:001FFEF8                 sub     eax, ebx
seg000:001FFEFA                 jmp     HeaderFound
seg000:001FFEFF ; ---------------------------------------------------------------------------
seg000:001FFEFF
seg000:001FFEFF PEHeadercheck:                          ; CODE XREF: seg000:001FFEEA↑j
seg000:001FFEFF                 cmp     dword ptr [ebx], 4550h ; look for PE header
seg000:001FFF05                 jnz     short BytesNotMatching
seg000:001FFF07                 add     eax, [ebx+28h]
seg000:001FFF0A                 jmp     HeaderFound
```

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000010   78 E5 8C 8C 3D 8A 1C 4F 99 35 89 61 85 C3 2D D3   xåŒŒ=Š.O™5‰a…Ã-Ó
00000020   00 00 20 00 00 00 00 00 5F 46 56 48 FF FE 07 00   .. ....._FVHÿþ..
00000030   48 00 8B F6 00 00 00 02 20 00 00 00 00 00 01 00   H.‹ö.... .......
00000040   00 00 00 00 00 00 00 00 0A CC 45 1B 6A 15 8A 42   ........ÌE.j.ŠB
00000050   AF 62 49 86 4D A0 E6 E6 B8 AA 02 00 2C 00 00 F8   ¯bI†M ææ¸ª..,..ø
00000060   14 00 00 19 4F DA 3A 9B 56 AE 24 4C 8D EA F0 3B   ....OÚ:›V®$L.êð;
00000070   75 58 AE 50 FF FF FF FF F6 CE 1C DF 01 F3 63 4A   uX®PÿÿÿÿöÎ.ß.ócJ
00000080   96 61 FC 60 30 DC C8 80 D8 AA 03 00 DC 42 00 F8   –aü`0ÜÈ€Øª..ÜB.ø
00000090   A4 42 00 10 4D 5A 00 00 00 00 00 00 00 00 00 00   ¤B..MZ..........
000000A0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000000B0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000000C0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000000D0   80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   €...............
000000E0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000000F0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000100   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000110   00 00 00 00 50 45 00 00 4C 01 07 00 00 00 00 00   ....PE..L.......
00000120   00 00 00 00 00 00 00 00 E0 00 0E 03 0B 01 02 14   ........à.......
00000130   00 3E 00 00 00 48 00 00 20 00 00 00 A0 02 00 00   .>...H.. ... ...
00000140   A0 02 00 00 60 3F 00 00 94 00 E0 FF 20 00 00 00    ...`?..”.àÿ ...
00000150   20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ...............
00000160   00 00 00 00 00 41 00 00 90 02 00 00 00 00 00 00   .....A..........
00000170   0B 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

GUID: Flash FileSystem v2
FFS header (look for _FVH)
GUID: *apriori* file
Cheksums
Filetype: EFI_FV_FILETYPE_FREEFORM
Attributes: None
Length: 44 bytes (size is 3 consecutive UINT8, so technically written in big endian)
File Status: ?
Section Size: 20 bytes
Section Type: EFI_SECTION_RAW
GUID: *SEC_MAIN*
note padding before GUID. Files must be 8 bytes aligned.
Cheksums
Filetype: EFI_FV_FILETYPE_SECURITY_CORE
Attributes: None
Length: 17116 bytes
File Status: ?
Section Size: 17060 bytes
Section Format: EFI_SECTION_PE32

The GUID (highlighted in yellow) mentioned earlier is referring to the EFI_FIRMWARE_VOLUME_FILE_SYSTEM2_GUID, according to (PI Working Group, 2012). The GUID is a constant and can therefore easily be found within a ROM file. Another element to look for is the signature “_FVH”, typical for a FFS header. The

characteristics of the FFS is that it is composed of 32 blocks of 64K each which translates to 2MB.

The next GUID to be found is the *apriori* file (in pink). The file is enclosed into a file header of 24 bytes, followed by a 4 bytes section header. The content of the file contains only one GUID, corresponding to the PCD PEIM, which is a PEI Module that holds the Platform Configuration Database. Although not covered earlier, the PCD is a means to pass parameters to Modules (being PEI or DXE drivers) or to store platform specific information. The end goal of the PCD is to limit the modification of source code for developers by allowing parameters to fine tune use of a specific module (Intel, 2010).

The next GUID (in light blue) is of some interest, as it happens to be SEC_MAIN, the core module of the SEC phase. Shown as encoded in PE32, based on the file and section headers, the MZ and PE header are present but meaningless because before the DXE phase it is not defined how to interpret a proper PE/COFF header. When looking back to IDA Pro, we can find the check for 0x10. The SEC phase continues at 0x330.

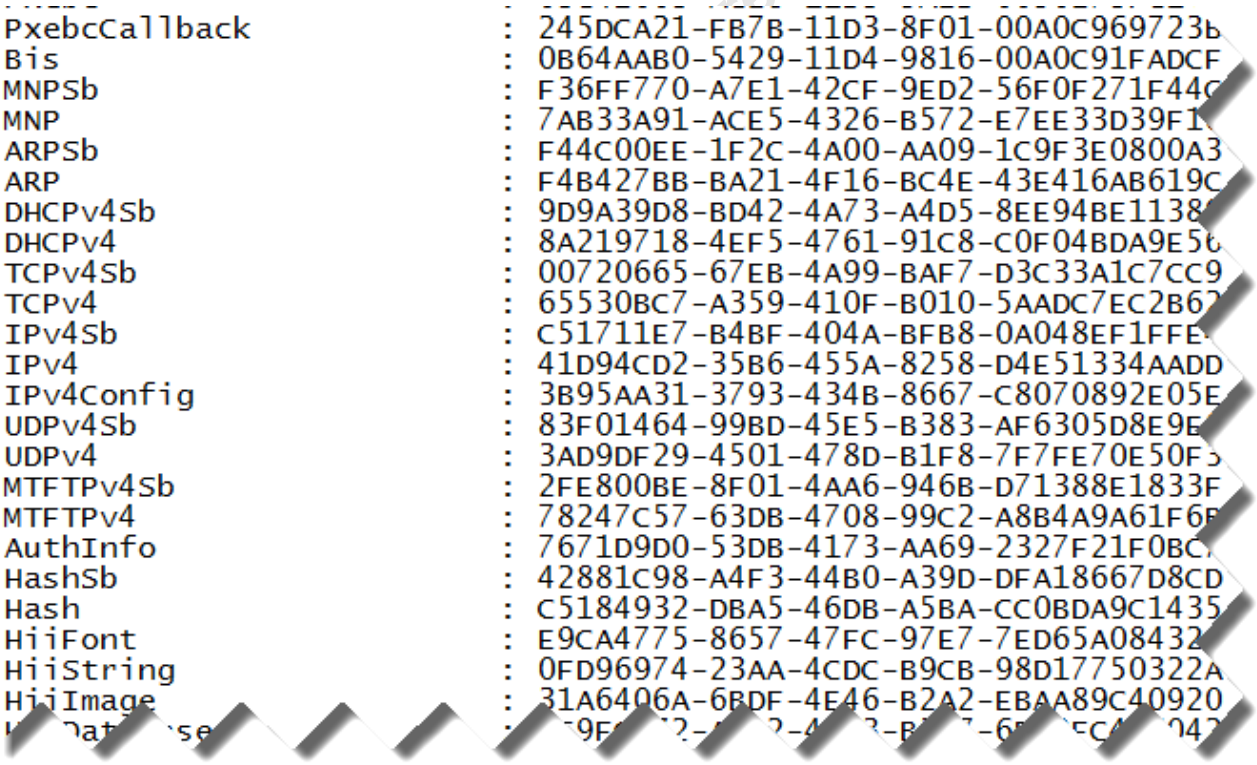
```
seg000:001FFEB0                 cmp     eax, ecx        ; eax>ecx
seg000:001FFEB2                 jnb     short BytesNotMatching
seg000:001FFEB4                 cmp     byte ptr [eax+3], 10h ; look for "EFI_SECTION_PE32"
seg000:001FFEB8                 jz      short MZheaderCheck
```

Having the files properly structured in a FFS allow us to find the different files by looking at their header, just like the BIOS will do when executed, and working this way it is much easier to find files boundaries, as well as much easier to add, delete, modify or replace modules within a ROM file (in Para 5.3 such a method is used to bypass windows validation by stealing OEM strings and certificates and replacing them in other FFS-structured files).

At this stage, it is too early to consider opening the PEI or SEC files with IDA Pro, as we would for windows applications. Similar Reverse Engineering techniques, as used for legacy BIOSs, could be used to analyze the files (unless the files are coming straight from Intel's development kit), but even if they are not, having the source code of similar implementations could really help, and the next phase is what would be more interesting.

4.4. The Driver Execution Environment

Knowing that each element of the FFS is identified by its GUID, the path taken from here will be relatively easy. The idea is to extract a driver at the DXE level by means of analyzing which GUIDs are available, find them in the File System, and then extract them. Accessing the shell in the VMware Environment can be done by not having any bootable media for the guest VM and to have the UEFI BIOS selected (as explained in 4.1). While this is not the intent to look at the details provided by the CLI ("help" is your friend), the command **guid** display the drivers and the corresponding GUID.



None of the DXE drivers could, in fact, be found directly in the FFS, because a file, called DXE File Volume (DXEFV) based on its GUID, contains all the DXE drivers in a single RAW section. According to comments within the Virtual Box flash drivers (Oracle Corporation), the section is compressed with LZMA. The file starts at 0x49150 and ends at 0x10EC52 and note the extension of the file; **unlzma** (**xv**) wants to see it to decompress the file.

```
dd if=efi64.rom of=DXEFV.lzma skip=299344 count=809730 bs=1
unlzma DXEFV.lzma
```

The uncompressed file has a MD5 hash of **9735970d40d9c626fffa23efa9e77d11**

It contains another FFS, as clearly identified by _FVH at 0x38, and using **strings** against the file now reveals what is expected:

```
br0l@toshiba /cygdrive/z/BIOS/UEFI_ROMS
$ strings DXEFV_LZMA |grep dll
StatusCodeRuntimeDxe.dll
PcdDxe.dll
RuntimeDxe.dll
SecurityStubDxe.dll
DataHubDxe.dll
EbcDxe.dll
Legacy8259.dll
CpuIo2Dxe.dll
CpuDxe.dll
Timer.dll
PciExpressHostBridge.dll
PciBusDxe.dll
PmRuntimeDxe.dll
Metronome.dll
PcRtc.dll
WatchdogTimer.dll
MonotonicCounterRuntimeDxe.dll
CapsuleRuntimeDxe.dll
ConPlatformDxe.dll
ConSplitterDxe.dll
GraphicsConsoleDxe.dll
TerminalDxe.dll
BdsDxe.dll
DevicePathDxe.dll
PrintDxe.dll
Di  ToDxe.  ll
```

Most of these files will follow the UEFI DXE drivers interface. The DiskIO driver, with a GUID of **CE345171-BA0B-11D2-8E4F-00A0C969723B** (as reported by the command **guid** within the UEFI shell) cannot be found in the image, although "DiskIODxe.dll" can be found by using **strings** on the FFS image. The corresponding GUID, based on a reverse search of the MZ header from that location, reveals a GUID value of **6B38F7B4-AD98-40E9-9093-ACA2B5A253C4,** which corresponds to a UEFI 1.0 DiskIO driver! (it is unclear why a 2.0+ compliant GUID is reported when a 1.0 version is loaded). Any occurrence of the GUID seems to be linked to a list of dependencies (GUIDs are next to each other's, as an array), except at address 0xC3017C, where a lot of GUIDs are presents but with associated values. This address is part of the UEFI shell application, and it may be a way to create aliases to GUIDs, but this cannot be demonstrated.

However, the goal was to access the files that can, if needed, be loaded in IDA Pro as 64bit PE structured files, and to be complete, the exercise will:

1. Use the EDK II development interface to code a subverted driver and to give it the associated GUID; the sky is the limit at this stage.
2. Replace the real driver with a new driver, but because it is in a File System, there are some checksums at file level and File System level to recalculate. (see 0)
3. Compress the resulting File System with LZMA so it is now the new DXEFV to be added in the Binary ROM image.
4. Replace the DXEFV file with the new one, recalculating the checksum for the file and File System of the ROM file, and paying particular attention to ensure the position of other files are not changed (as it may break the early stage of booting where the BIOS is not yet able to read a file system).
5. In VMWare, copy the modified BIOS into the home folder of the virtual machine and edit the corresponding vmx configuration file of the guest VM and by adding the following entry it will now boot from the modified UEFI firmware.

```
efi64.filename = "myModdedUEFI.rom"
```

6. Note that if you are creating your own driver, you can assign another GUID to it, and it would be interesting to see if the GUID can just be added to the list in the file to be loaded and executed, instead of modifying an existing capability.

5. How UEFI can be subverted

There are several paths worth considering for exploitation, and while they are presented in dedicated subtopics, some of them are linked where exploitations paths are likely to make use of several for effectiveness or wider coverage. Using a program like **RW-Everything** (see 6.5), some of the specific elements mentioned below (ACPI Table, Option ROMS, SMBIOS Table,…) can be easily accessed and are worth investigating.

5.1. On signed code and the Trusted Platform Module

The code signing requirement of UEFI is not a mandatory one, but the principle behind it is to verify that all code executed by the processor has been signed by a trusted vendor. For this to be effective, it requires access to a safe place which contains the list of all trusted certificates for every single code being executed which is called for verification first. However, because this requirement was not planned in the earlier releases of UEFI, code-signing is basically an add-on to the existing interface. The requirement for a safe place to hold the certificates is, in most cases, the Trusted Platform Module (TPM), connected to the Low Pin Count (LPC) interface of the Southbridge (see Figure 7: Intel 955X-ICH7 chipsets (Salihun, 2007)). The TPM is not available on all motherboards, and as such does not really make sense for the consumer products due to extra cost, little or no added value for user experience, and extra burden for the end-user. Additionally, the list of trusted certificates needs an update mechanism, so implementation errors could lead to bogus certificates being added to the trusted list, and another issue is the availability of 2 protocols to load UEFI binaries, as stated in (Michael Rothman, 2009), which could possibly be another means of avoiding any signature verification.:

> The simpler method is known as the Load File Protocol and its expanded version, the Load File2 Protocol. These load files off of disks and the like. These protocols are allowed to assume that the format of the drive is FAT. Unlike older operating systems, UEFI uses a new highly extensible partition scheme, known as the GUIDed Partition Table.
>
> The second method for loading images (programs) allows digitally signed images to be loaded but is hidden in the security part of the specification.

In the VMWare implementation, the following 2 methods have been found by using the command **guid** in the CLI:

```
Load        : 56EC3091-954C-11D2-8E3F-00A0C969723B
Load2       : 4006C0C1-FCB3-403E-996D-4A6C8724E06D
```

Exploitation against the TPM has been demonstrated several times, although they are fairly complex and were not targeting exploitation at the BIOS level.

5.2. On the System Management Mode

System Management Mode (SMM) is a 16 bit mode which has existed since the 386 (Collins, 1997), mainly for debugging purposes, and could be utilized by use of a System Management Interrupt (SMI), where these interrupts are true hardware-based interrupts on a physical pin of the CPU. Later on, use of SMM became more widespread as it was very convenient to use it for Power Management (APM, ACPI) and hardware control (for instance, OS-based BIOS firmware update or the associated actions when you close the lid of a laptop). When SMM is called, all the CPU registers are saved on a place in memory called the SMRAM and the CPU is switched back to 16 bit mode but with access to the complete memory range. The power of SMM relies on its privileged access to all the system resources and on its non-detectability by the OS (BSDaemon, 2008).

While several protection mechanisms have been set up at chipset level, with specific registers controlling the accessibility of the SMRAM and its read/write status, security experts have so far managed to circumvent these defences (Loïc Duflot, 2006), (BSDaemon, 2008),(Wecherowski, 2009), (Rafal Wojtczuk). The SMM is a CPU feature and therefore needs to be supported by UEFI (PI Working Group, 2012), for which a whole chapter is devoted. The presence of properly documented libraries provided by UEFI and its associated development kit will likely ease the task to prevent the SMM protection from being circumvented, as nothing new on that subject has been brought forward in recent chipset development.

5.3. On the Advanced Configuration and Power Interface (ACPI) and the System Management BIOS (SMBIOS)

The ACPI, as briefly explained in 5.2, is used to control the Power efficiency of the running system, as well as controlling embedded hardware like fans or physical buttons connected to the motherboard. In contrary to its predecessor the Advanced Power Management (APM), ACPI is meant to be controlled by the OS, and as such, offers ACPI registers for control and ACPI tables to describe what capability the specific system has to offer to the OS (Loïc Duflot O. L., 2009).

Without going too much into the details, the OS accesses the ACPI tables to extract the Differentiated System Descriptor Table (DSDT), which defines the ACPI registers and the methods available to use them. In the same paper mentioned above, the DSDT can point to any memory address, even if it is totally unrelated to the ACPI registers. (Heasman, 2006) flashed a BIOS with a corrupted DSDT table to later use crafted OS driver accessing it. Again, (Loïc Duflot O. L., 2009) pushed the concept further by allowing the hidden code to be triggered by external physical events, in that case the connection of the laptop's power supply. Once again, the ACPI is a functionnality that has to be supported by UEFI, and as such, the same kind of flaws could, potentially, be developped on a UEFI based BIOS.

The SMBIOS (System Management BIOS) is another legacy feature that was carried over in UEFI. It is currently under revision 2.7.1, and its role is to *address how motherboard and system vendors present management information about their products in a standard format by extending the BIOS interface on Intel architecture systems*. (DMTF, 2011). As such, the SMBIOS offers a Table, similar to the ACPI, with the intent of it being accessed by interfaces like WMI for Windows systems. The support for SMBIOS is offered in (PI Working Group, 2012). One of the uses of SMBIOS is the integration of OEM strings that can automate the activation of Windows without going online (Techie, 2010). It is called SLP (System Locked Preinstallation), and is a code stored in the OEM Strings of SMBIOS. The other piece of the BIOS needed is called the SLIC (Software License Internal Code) which is a certicate stored in an ACPI Table called the Software Licensing Descriptor Table. One of the tricks used by crackers is to extract the SLP code and SLIC from a big vendor BIOS, modify a flashable version of their own motherboard BIOS, and apply the flash. This shows that the UEFI does not solve the piracy issue, at least not without the use of the TPM to protect its content.

5.4. On Option ROM support and the EFI Byte Code (EBC)

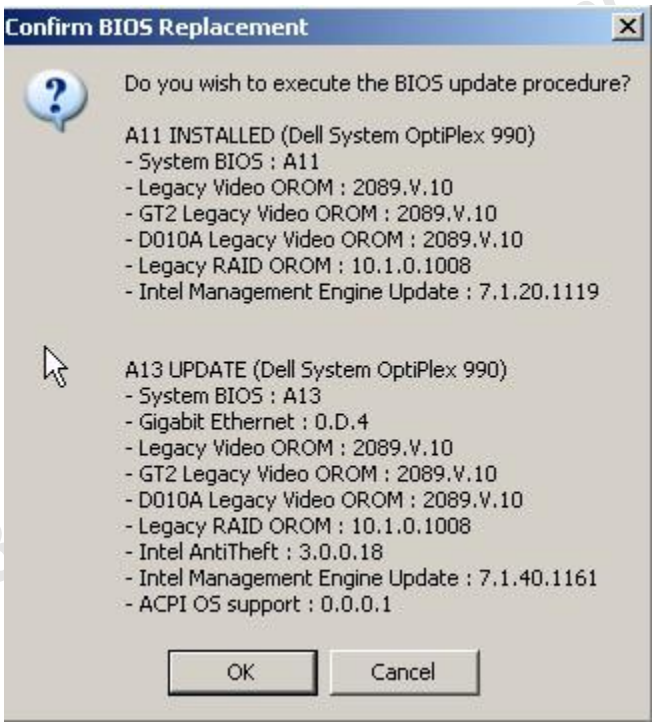
Option ROM (also called PCI expansion ROM) is the piece of code executed at BIOS to allow the support of the PCI card (or embedded PCI chipset) by the system before boot stage and is typically a network card with PXE capability, a RAID card and nowadays USB chipsets. Option ROM is true legacy BIOS technology, dating back to the ISA bus time, and is the answer to Personal Computer hardware flexibility.

In the legacy BIOS way of working, there is a specific memory region (OxC_0000 to 0xD_FFFF) of 128KB size, used for Option ROM to be executed. After POST, each PCI device is scanned, and whenever an expansion ROM was available (identified by AA55h in its Base Address), it was copied over in memory starting from 0xB_0000. When all the Option ROMs are copied over, the BIOS will then sequentially execute the ROMs (by looking for the 55AAh header in 2KB block) found in the order they have been copied (Salihun, 2007). This led to a multitude of problems, such as when Option ROMs were not compatible with each other (remember the times when you had to swap expansion cards in the slot to get the computer working), or when there was not enough space available for the code to be copied in the memory for execution. (UEFI Spec Working Group, 2012)

ROM code is written for the same execution mode as the BIOS, explaining why some physically compatible cards cannot work on a PC if they are coming from a SUN computer for instance. If the execution code is RISC based and not x86, there is no chance it will be interpreted correctly by the CPU. By design, the option ROM, when executed, has a full access to the system and resources. (Heasman, Implementing and detecting a PCI rootkit, 2007)

UEFI provides a completely different approach, where drivers have to be UEFI compliant, where memory is assigned dynamically for it (see DXE), where drivers can be linked to several PCI chipsets (no need to have 2 copies of the same option ROM in the memory to get 2 cards running). The question comes when you want to know how your PCI or PCIe card will be recognized on your UEFI computer, and several options exist.

First option, you can have the UEFI driver of your card already available in the original BIOS firmware; unlikely unless you go for a brand-name PC with well-defined pre-order options. Second option, you can have a UEFI driver on the EEPROM of your expansion card; more likely to happen if it is brand new. Third option would be the lack of support at BIOS level of your hardware, which can be problematic if it goes about a network or block device controller. Last option, you can have the UEFI BIOS reading and executing the legacy Option ROM as shown below in an example from a Dell Optiplex 990. While the BIOS is UEFI, it is still embedded with legacy option ROM that will be executed by the UEFI BIOS:



The support for legacy Option ROM in UEFI is planned in the UEFI specifications 13.4.2. (UEFI Spec Working Group, 2012), and described for developers in (Phoenix, 2009). What is not clear is how a proper legacy driver will be granted access to the system. The kind of attack (Heasman, Implementing and detecting a PCI rootkit, 2007) proved in using Option ROM to load his rootkit code can, potentially, have the same impact on a UEFI BIOS.

MANUEL HUMBERTO SANTANDER PELAEZ MANUEL HUMBERTO SANTANDER PELAEZ

Another topic worthy of study is the EFI Byte Code, or EBC, which is pseudo assembly code that is meant to be architecture agnostic. Such an UEFI Option ROM could then be run on any architecture which offers booting on UEFI aware architecture. It prevents the hardware vendors from writing several versions of their UEFI drivers based on the underlying CPU, and it also means that the EBC could be the source of code to write multi-architecture rootkit at pre-OS level.

A last point, which can be linked further to 5.5 and 5.6, is the ability to have applications stored on an expansion ROM and have them executed, as cited in (UEFI Spec Working Group, 2012) below:

> *It is also possible to place an application written to this specification in a PCI Option ROM. However, the PCI Bus Driver will ignore these images. The exact mechanism by which applications can be loaded and executed from a PCI Option ROM is outside the scope of this document.*

5.5. On the runtime services and the capsule

A brief mention of the boot and runtime services has been made in 3.4. The boot services can be consumed during pre-OS level, while the runtime services are present during the whole uptime of the system. There is a minimum set of runtime services that have to be made available to be compliant, but nothing to prevent extra services made available to the OS.

One such service is the capsule service (Vincent Zimmer, 2010). The capsule is a placeholder that survives a reboot or reset. The most obvious use for such a service is the update of the firmware. First, the capsule service is called to hold a copy of the piece of code that needs to be loaded after reboot and then the reset service is called, in this case "BOOT_ON_FLASH_UPDATE". This method is convenient, because for the first time, there is a generic way of upgrading a BIOS; however, it is also a convenient storage for any piece of malware that needs to survive a reboot. When updating the BIOS of a Dell Optiplex 990, a specific file called DBUtil_2_3.sys is created in the default temporary folder, and then rapidly deleted. To get hold of this file you can explicitly deny anyone the right to delete subfolders and files on the NTFS partition. The file has the following MD5 hash: **084bd27e151fef55b5d80025c3114d35**

Using strings on the file the following item can be found:

c:\data\work\tools_efitools\trunk\ringzeroaccesslibrary\win\kernelmodedriver\objfre_wlh_x86\i386\DBUtilDrv2_32.pdb

The driver has not been studied further, and unfortunately no clear evidence of the use of the capsule could be found. In reality, and quoting Vincent Zimmer himself in a very recent blog post (Zimmer, 2012), the Windows OS offers very limited access to UEFI so far, so that the UEFI runtime services are at the day of today pretty much not in use.

Because a runtime service can technically link a ring-3 user to anything the service offers, a modified UEFI driver could technically link a runtime service to a System Management Interrupt, which from there would launch the same kind of undetectable exploits as described in 5.2. A runtime service could also, for instance, be used to get access to a block device, or to the whole memory range, to make use of the network card without using any of the OS drivers.

5.6. On the scripting capability and binary shell execution

The UEFI BIOS offer the ability to access the command line, which is an interesting mix between DOS and UNIX commands. The shell happens before the *ExitBootServices(),* meaning that the boot services are available. The UEFI shell is a 21st century DOS environment, in the sense it is a single user mode with access to all resources. The environment was originally created for hardware vendors, integrators and BIOS developers to test the capability of their systems. Because of the scripting capabilities, and the ease of development by mean of the freely available EDK II, applications could be created at pre-OS level. This full access to the system means any application badly written, or written for a bad purpose, could be harmful for the system itself. There are obviously some interesting applications that should ease for instance forensics examiners (like dd for UEFI), but it is out of the scope of this document.

Recently, a bootkit attack against Windows 8 has been demonstrated, where the fact that the early stage of windows boot happens before the *ExitBootServices()* could be exploited in conjunction with the fact that UEFI applications or drivers can reside on any block device, in this case the GPT hidden partition (Allievi, 2012). This is a good example of the danger of a powerful pre-boot environment.

6. Online Resources and Tools

Two main categories of tools exist to assist in extracting or modifying the ROM binaries. First of all, and for obvious reasons, tools are being developed by BIOS vendors to help hardware integrators to tweak them to their likings and to allow end-users to flash the upgrade on their motherboard. The tools are coming from Intel, Phoenix, AMI, Award and Insyde. There is also the UEFI Forums' development toolkit which allows new drivers and applications to be coded.

The second category is the one filled by non-official tools. Interestingly enough, the tools are mostly developed for hardware modders. It permits them to change some settings unavailable from the BIOS interface and to replace an Option ROM with a modified one (typical for video cards). As a general rule, these tools are also BIOS vendor specific.

To start digging into BIOS modification resources, the following forums are worth visiting.

My Digital Life : http://forums.mydigitallife.info/

BIOS Mods : http://www.bios-mods.com/

The Rebels Haven : http://www.rebelshavenforum.com/sis-bin/ultimatebb.cgi

Wim's BIOS : http://www.wimsbios.com/forum/

6.1. Intel BIOSs

The **Intel Integrator Toolkit v5.0**[1] is an official tool to modify UEFI BIOS. It allows you to select from a list of motherboards and will download whichever version of the firmware you want to modify. It allows the change of logos, the default values of the settings, and specific details about the SMBIOS (Some motherboards require the v4.0 toolkit).

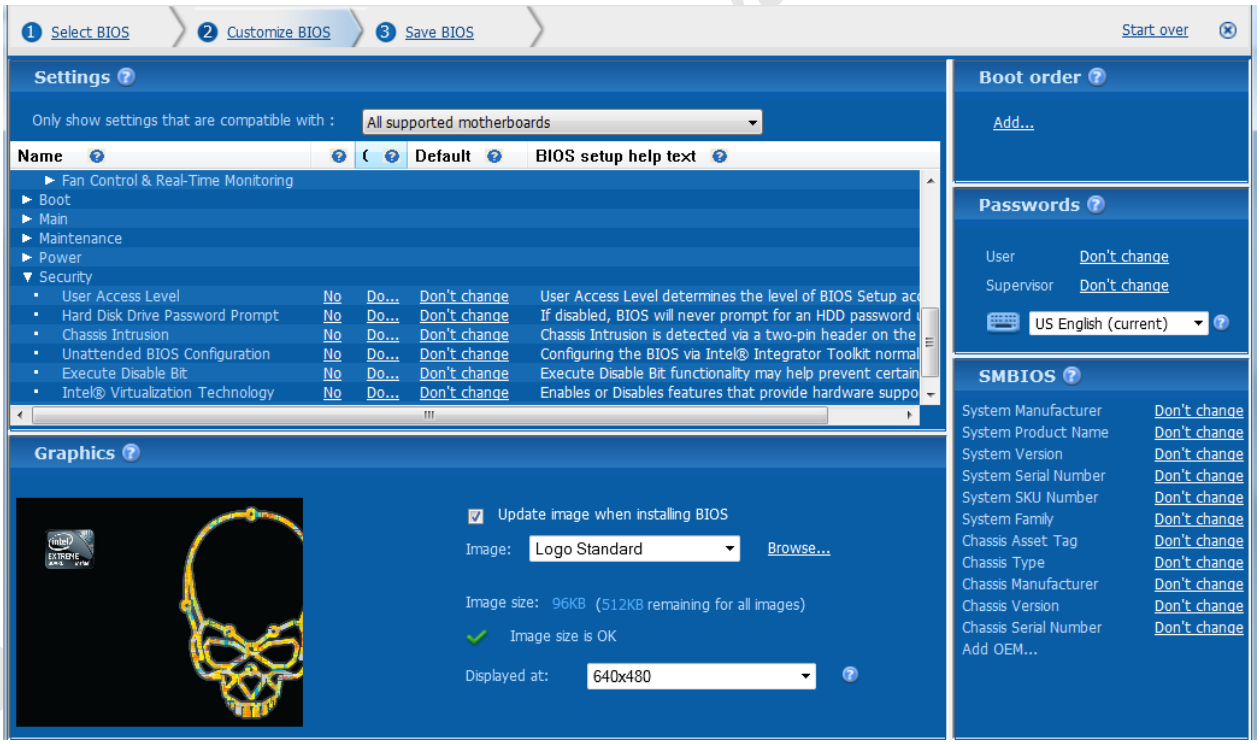


Figure 16: Intel Integrator Toolkit

While this was not meant to be a hacker tool, the benefit of this tool is that you are able to access all BIOS revisions for all available Intel motherboards. This tool will allow you to create a perfectly valid ROM with some interesting options to consider, such

[1] http://downloadcenter.intel.com/Detail_Desc.aspx?DwnldID=20829

as ***Unattended BIOS Configuration*** and the ***Execute Disable Bit;*** creating a valid BIOS, with a downgraded security posture, is a first step towards exploitation!

6.2. Phoenix BIOSs

Phoenix WinPhlash 1.7.16.0[2] is a tool to backup, and load, firmware on Phoenix based motherboards (such as some DELL computers).

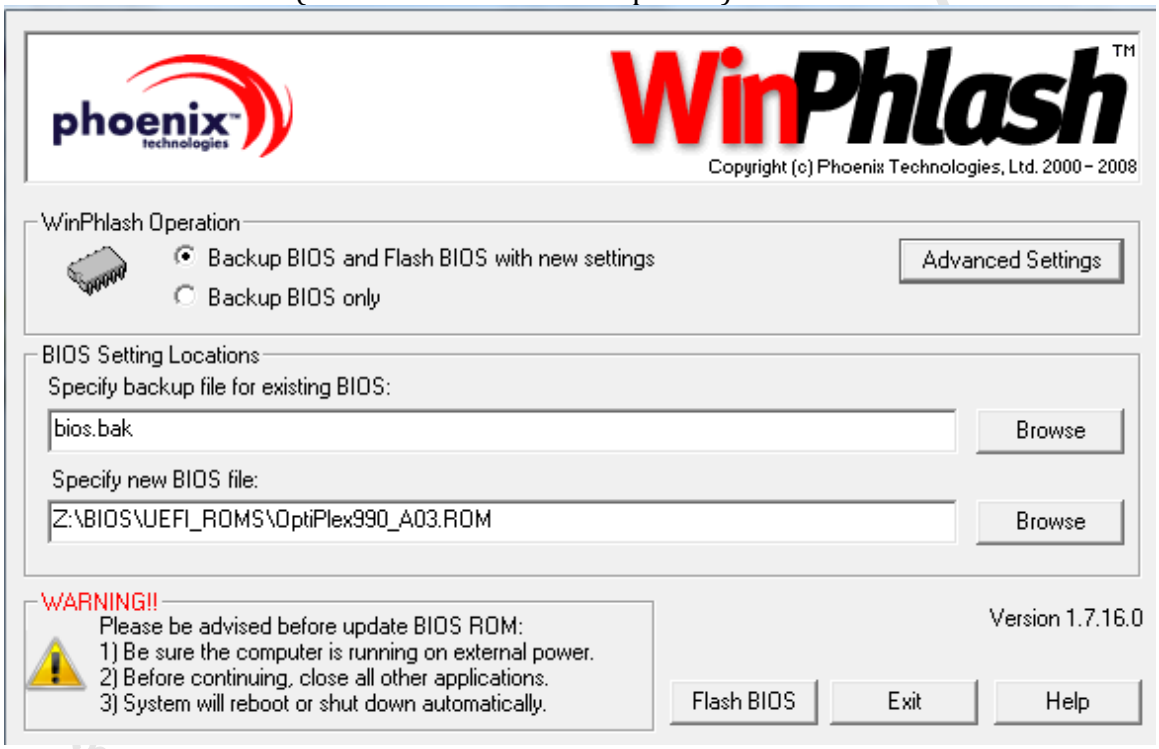


Figure 17 : WinPhlash

The tool only works on a Phoenix-based motherboard and also exists as a 64 bit version (**WinPhlash64**). It is used to extract the original BIOS and flash a modified one, and the tool can be rebranded by hardware integrators. While it has limited capabilities for modifying its configuration, an alternate tool, discussed in 6.5 will allow the proper insertion, removal or change of specific GUID and to recreate a valid UEFI FFS.

- [2] http://www.bios-mods.com/tools/index.php?dir=Phoenix+WinPhlash+v1.7.16.0%2F&download=Phoenix+WinPhlash+v1.7.16.0.zip

6.3. American Megatrends (AMI) BIOSs

AMI version of UEFI BIOS is called Aptio, and. **MMTool Aptio v 4.50.0.23** allows to modify a BIOS file by extracting, inserting, deleting or replacing modules (identified by their GUID) in a ROM FFS.

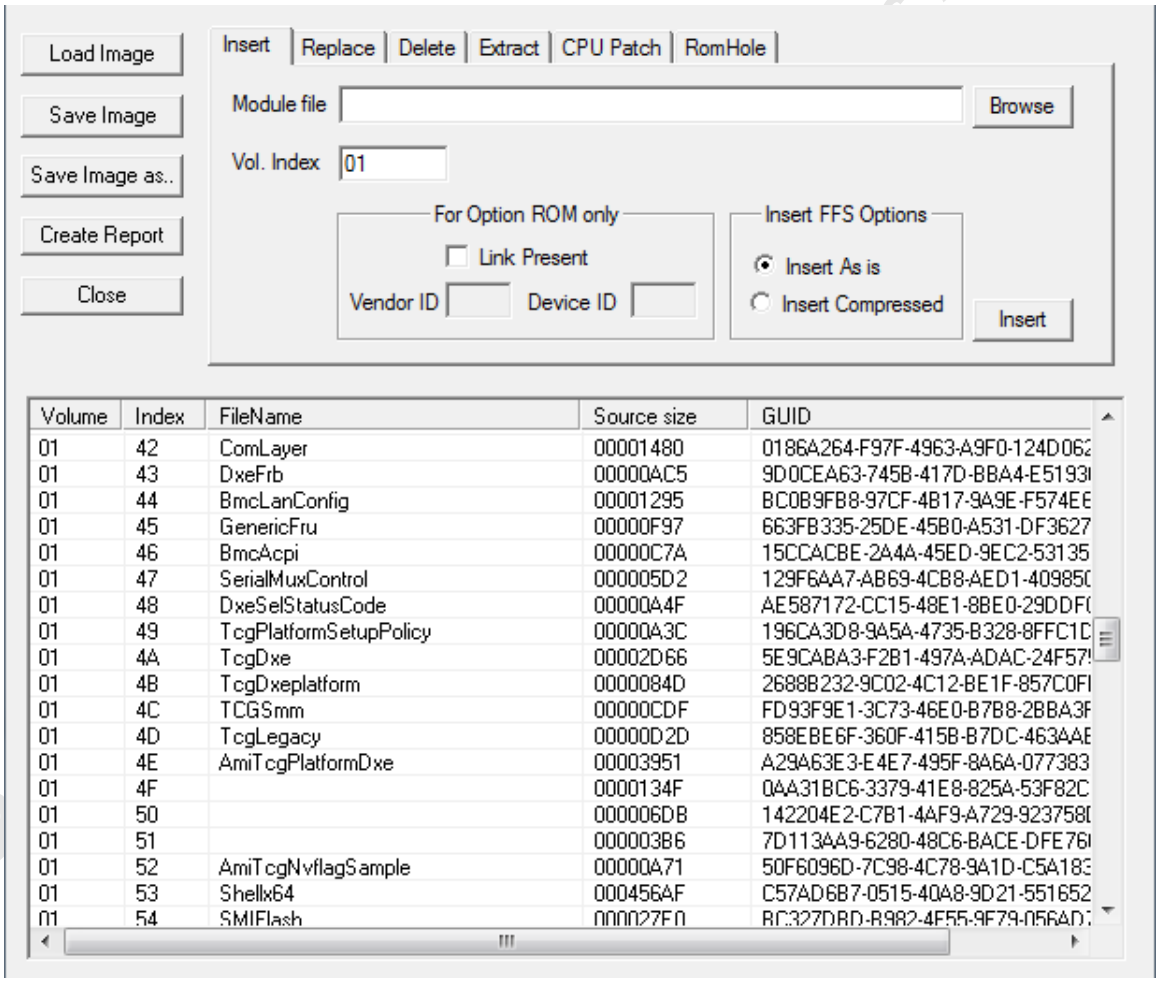


Figure 18: MMTool Aptio

The ROM file above is from a Lenovo ThinkServer RD630. At index 53, the presence of the UEFI shell can be identified. This tool, combined with the **AMIFlash**[3] or

[3] http://www.ami.com/support/downloads/amiflash.zip

its Linux equivalent, **AFULNX**[4] (here embedded with the Lenovo BIOS) should be enough to manipulate Aptio-based ROM binaries. These two applications allow you to back-up the current version or flash another binary.

6.4. Insyde BIOSs

Insyde BIOS is what some HP products, and VMWare, are using. **InsydeFlash**[5] version 4.0.7.3 is the application which allows the EEPROM to be backed-up or flashed (the DOS based version is called **FlashIT**[6]**)**. There is another tool from Insyde, called **InsydeFDPacker**[7]**,** but only an older version (2.0.6) could be found embedded with InsydeFlash version 3.53. This tool allows the creation of an executable with everything embedded for flashing the firmware. Nevertheless, as for Phoenix UEFI images, a third-party application, covered in the next section, will allow to properly modify the Insyde BIOSs.

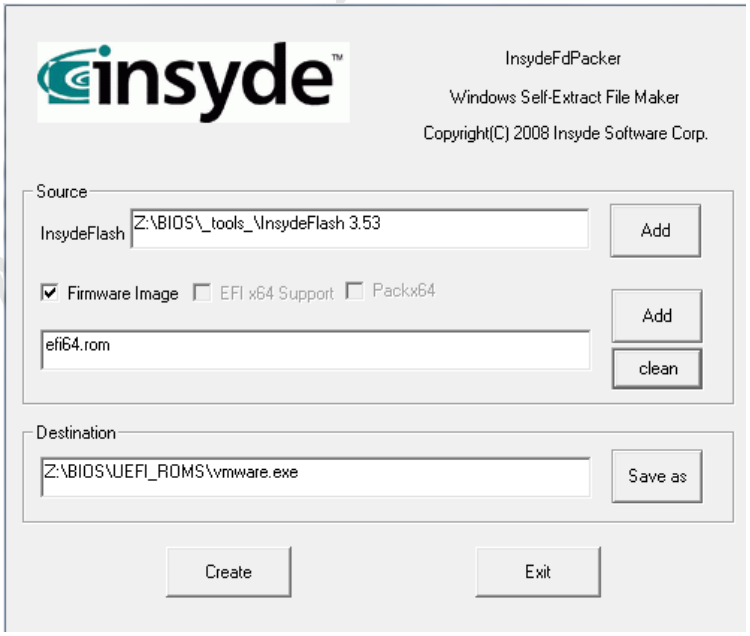


Figure 19: InsydeFDPacker

[4] http://download.lenovo.com/ibmdl/pub/pc/pccbbs/thinkservers/bios_rd530_v205.tgz

[5] http://www.mediafire.com/?d6fgcgqq08cwc3n

[6] http://www.mediafire.com/?tlsbq634hinzavn

[7] http://www.biosrepair.com/biosfiles/InsydeFlash%203.53.rar

6.5. Generic

There is one tool which is very handy when dealing with UEFI images and it has several names. It used to be called the **PhoenixTools**, more recently called **Phoenix/Dell/EFI SLIC MOD**[8] v 2.12. The tool works with some legacy BIOS (hence the reference to Dell, Phoenix and Insyde) but can also decode UEFI FFS. While primarily meant to modify SLIC and SLP, as explained in 5.3, the tool can also allow viewing, inserting, replacing and deleting elements of a UEFI firmware, which means that the features lacking with some BIOS vendor-specific tools are actually provided by this underground tool. The tool can read the VMware EFI64 image, as seen below:

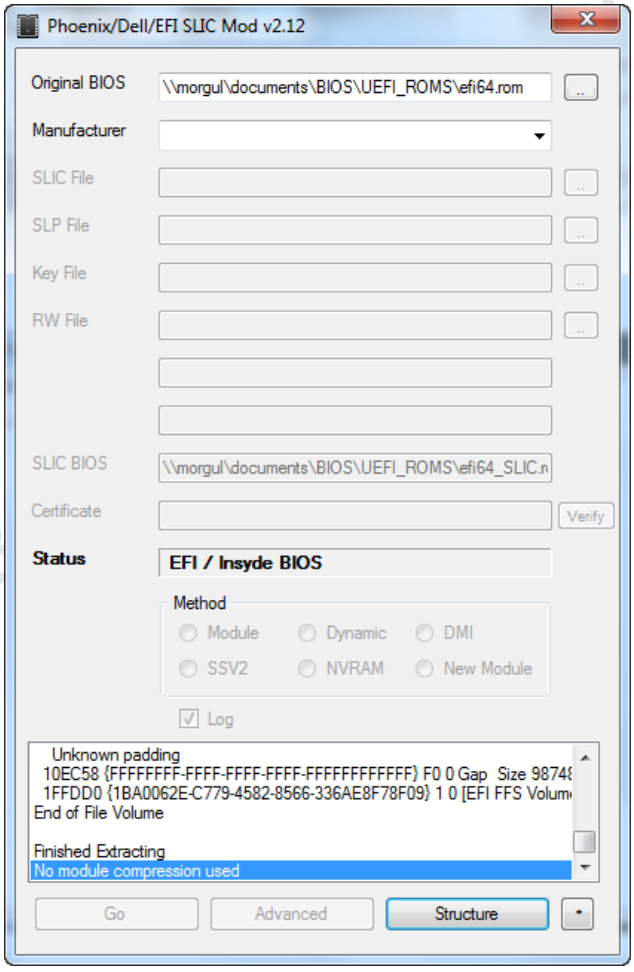


Figure 20 : VMWare EFI64 FFS in PhoenixTools

[8] http://www.sendspace.com/file/cztx25

As well as the Dell Optiplex 990 EFI ROM extracted using WinPlash. The tool can be used to extract EFI applications or drivers to be reused in another firmware, perhaps from another vendor as well.

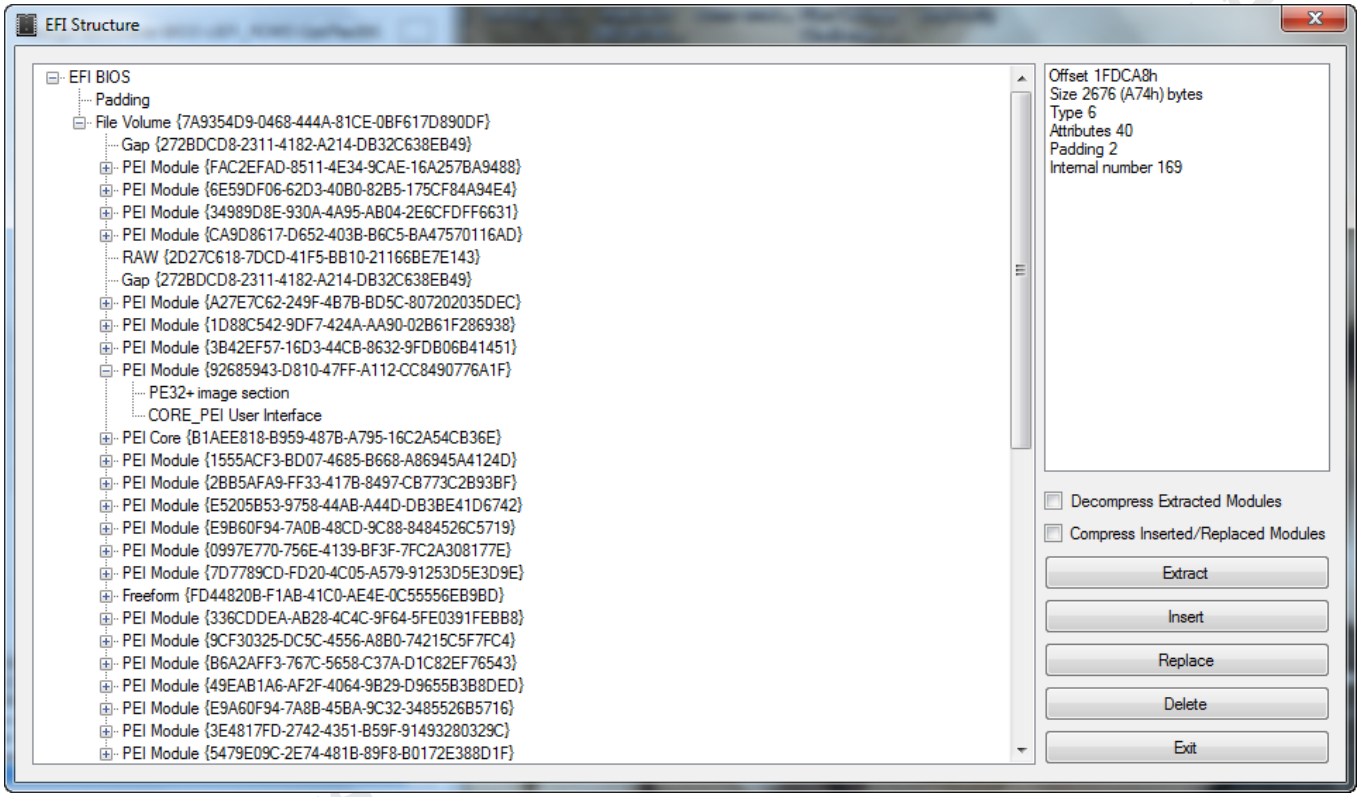


Figure 21 : Dell Optiplex 990-A02 structure with PhoenixTools

Another useful tool is **RW-Everything**[9], and version 1.4.9 has been tested. The tool can help viewing and extracting the tables available in the system: SMBIOS, ACPI-related ones like the DSDT or RSDT. It can also extract the content of Option ROMs.

[9] http://jacky5488.myweb.hinet.net/

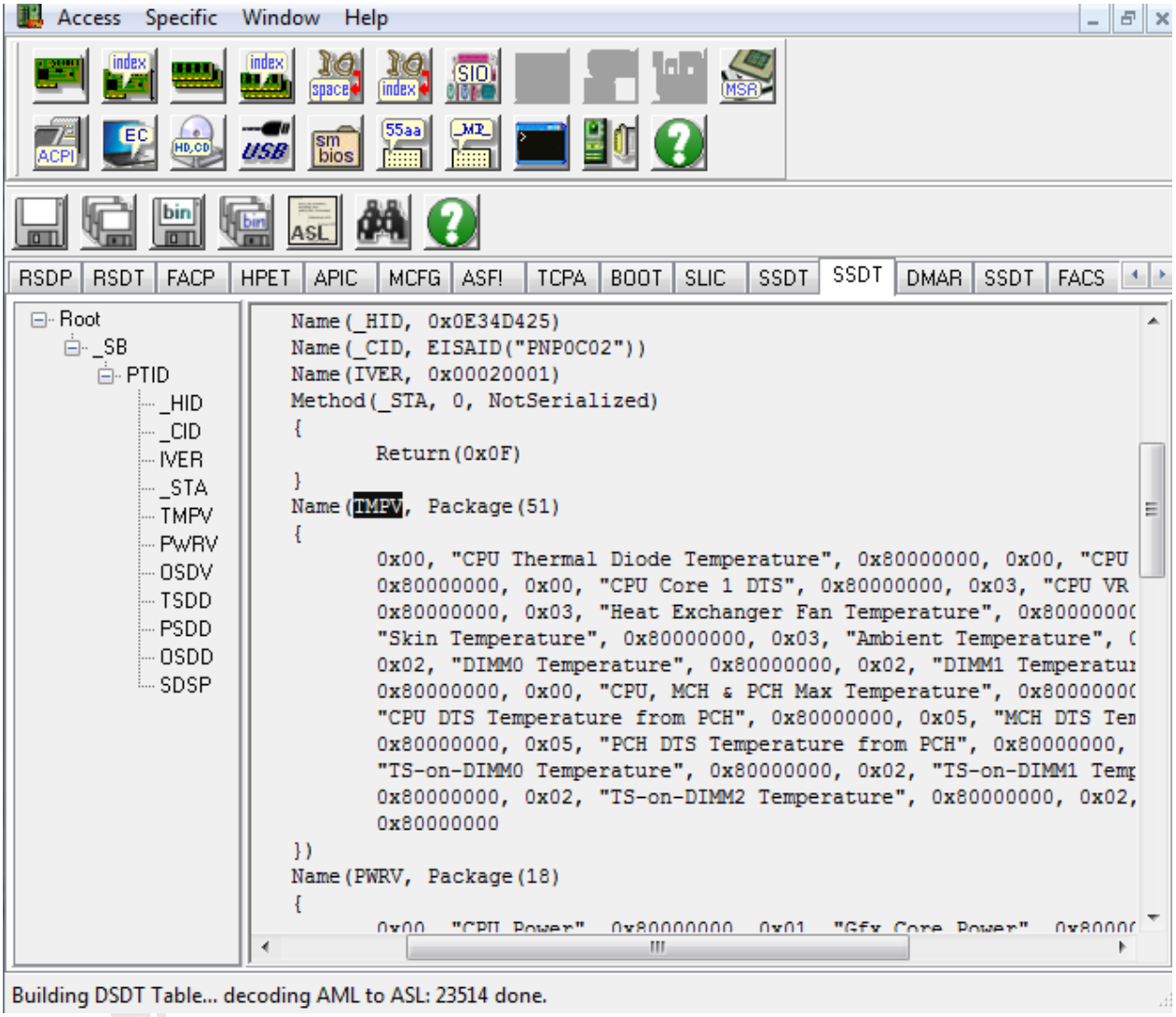


Figure 22: RW-Everything on a Toshiba Tecra Laptop

6.6. The development environment

Initially called the EFI Development Kit (EDK), at that time mostly an Intel initiative, it became EDK II when UEFI became v2.0 and came with a UEFI implementation called TianoCore, for which all the source code and libraries were available. The UEFI Development Kit (UDK 2010)[10] is based on stable and validated code from EDK II. Additionally, starting from EDK II the code can be compiled with

[10] http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=UDK2010

GCC, whereas before the development environment had to run on Microsoft Windows only.

The UDK2010 comes with some base tools for developers. Two tools are definitely needed to handle firmware image properly: **GenFfs** and **GenSec**. GenSec's role is to take, as input, an EFI binary (being most likely an application or a driver) and make a FFS-compatible section of it. (Files on Flash File Systems can be composed of sections, as explained in (PI Working Group, 2012)). GenFfs is used to take one or more sections to structure them into a File (or module) that can be integrated into a Flash File System. The tricky bit with such file systems is that specific details regarding block size and alignment needs to be extracted from the File Volume when the File is created with GenFfs. The other point of contention is related to the different status that can be given to files and sections within a File volume. Some options seem to be missing from the EDK tools, but having wrong parameters in the file or section headers can potentially result in unbootable systems.

In the case of VMWare, the example given in 4.4 has an extra layer of complexity, and the GenSec and GenFfs need to be used first on the module to be inserted or replaced on the compressed file Volume, then when done, it needs to be compressed with lzma. The resulting file has to be included in a section using these commands:

```
lzma embeddedFileVolume.ffs

GenSec -o FileVolume.SEC -s EFI_SECTION_GUID_DEFINED -c PI_NONE -g EE4E5898-3914-4259-9D6E-DC7BD79403CF -l 0 -r PROCESSING_REQUIRED embeddedFileVolume.ffs.lzma

GenFfs -o FileVolume.MOD -t EFI_FV_FILETYPE_FIRMWARE_VOLUME_IMAGE -g 20BC8AC9-94D1-4208-AB28-5D673FD73486 -s -i fileVolume.SEC
```

The insertion of the generated module can be done with MMTool or the PhoenixTools. The former does not handle the GUID section within a GUID File properly and creates an empty section in the firmware, whereas the PhoenixTools is able to handle the replacement file properly. However, byte to byte comparison between the original firmware, and the modified one, shows unsuitable file status in the FFS header.

While the section status can be set with GenFv, the file status is missing from the options with GenFfs, and as any hexeditor modification is also breaking the checksum of the file content, this is also not a solution. This means that the base tools of the EDK II might need to be modified to allow more flexibility for UEFI hackers. While the theory is still valid, VMware refused to start with the modified firmware; this problem has also been found on physical hardware[11].

One point to consider when wanting to add applications to UEFI is the unavailability of the FFS when in the shell. The binaries need to be loaded from elsewhere, like a FAT volume (USB stick), or maybe the network. In contrary to drivers that can reside on the FFS, and that will be loaded by the DXE dispatcher, for an UEFI application residing on the firmware volume to be executable from the shell, it would require the assistance of a third party DXE driver, called the **EDK II FileSystemPkg**[12]. This is definitely a driver to include in hacked firmware if there is a need to have access to other applications stored on that firmware.

Finally, the **EFI Toolkit**[13] v2.0.0.1 is worth mentioning. It contains a pack of applications, with the corresponding source code, that can be used as basis for further development, or as nice add-ons to a UEFI shell. The Toolkit provides **ifconfig**, **route**, **ping** but also a **python** interpreter and other useful examples.

[11] http://forums.mydigitallife.info/threads/13194-Tool-to-Insert-Replace-SLIC-in-Phoenix-Insyde-Dell-EFI-BIOSes/page280

[12] https://github.com/cfdrake/FileSystemPkg

[13] http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=EFI_Toolkit

7. Conclusions

UEFI has three main goals: to make it easier for developers, integrators and hardware manufacturers; to fix the legacy issues of hardware support and boot device selection; and to be future-proof by means of a modular design. There is little doubt these goals are being achieved and that the building blocks are very well thought out, but based on its potential power (basically an OS before the OS) and the legacy elements that have to be carried over, UEFI could very much become a Pandora's Box that has just been cracked open. Never before was the pre-boot environment so convenient for the power user, so well documented, so properly supported by means of a development environment and libraries. Never before was the BIOS code so portable up to cross-platform level, and never before was it so easy to add extra code to be executed at pre-boot level that could reside on the local hard drive or even on the network.

Sadly, the security considerations for its design came too late and are not enforced, allowing manufacturers and developers to make it happen with only usability and cost in mind. The security community should be worried on the exploitation paths UEFI could be victim of, or could facilitate. They should, at the enterprise level, ensure that all security measures proposed by the UEFI specifications are set. Another mitigation measure is the enforcement of the Intel Trusted Execution Technology (TxT) or the AMD Presidio, which allow safe execute of code, even on untrusted environments.

A lot of power can be gained, with much less effort than with the legacy BIOS, and as such, we might be looking at the dawn of another wave of BIOS-based malware.

8. Bibliography

Allievi, A. (2012, September 9). *UEFI technology: say hello to the Windows 8 bootkit!* Retrieved from http://www.itsec.it/2012/09/18/uefi-technology-say-hello-to-the-windows-8-bootkit/

Brouwer, A. E. (n.d.). *A20 - a pain from the past*. Retrieved from Technische Universiteit Eindhoven: http://www.win.tue.nl/~aeb/linux/kbd/A20.html

BSDaemon, c. (2008). System Management Mode Hack - Using SMM for "Other Purposes. *Phrack 65*.

Collins, R. R. (1997, January). *Intel's System Management Mode.* Retrieved from Dr Dobbs' undocumented corner: http://www.rcollins.org/ddj/Jan97/Jan97.html

Compaq, Phoenix, Intel. (1996, January 11). *BIOS boot specification 1.01.* Retrieved from http://www.scs.stanford.edu/nyu/04fa/lab/specsbbs101.pdf

Dice, P. (2011). The flow of booting an Intel Architecture system.

DMTF. (2011). *SMBIOS Reference Specifications.*

Heasman, J. (2006). Implementing and Detecting a ACPI BIOS rootkit.

Heasman, J. (2007). Implementing and detecting a PCI rootkit.

Hoffman, A. (1989). *Assembleur sur PC.* Paris: Micro Applications.

Intel. (2008, March). Intel® X48 Express Chipset datasheet. Retrieved from http://www.intel.com/Assets/PDF/datasheet/319122.pdf

Intel. (2010). *UEFI Development environment update*. Retrieved from intel.com: http://software.intel.com/sites/default/files/m/1/f/0/3/4/31686-11_UEFI_Development_Environment_Update.pdf

Intel Press. (2011). *Intel 64 and IA-32 Architecture Software Developer's manuals.* Intel Press.

Kholodov, I. (2007). *CIS-77 Introduction to Computer Systems.* Bristol Community College.

Loïc Duflot, D. E. (2006). Using CPU System Management Mode to Circumvent Operating System Security Functions.

Loïc Duflot, O. L. (2009). ACPI et routines de traitement de la SMI: des limites à línformatique de confiance?

Michael Rothman, T. L. (2009). *Harnessing the UEFI shell: moving the platform beyond DOS.* Santa Clara: Intel Press.

Oracle Corporation. (n.d.). *VBoxPkg.fdf - VirtualBox Flash Device.* Retrieved from http://www.virtualbox.org/svn/vbox/trunk/src/VBox/Devices/EFI/Firmware2/VBoxPkg/VBoxPkg.fdf

Phoenix. (2009, January 9). *BIOS Undercover: Launching A Legacy Option ROM In SecureCore-Tiano*. Retrieved from http://blogs.phoenix.com/phoenix_technologies_bios/2009/01/bios-undercover-launching-a-legacy-option-rom-in-securecoretiano.html

PI Working Group. (2012). *Platform Initialization Framework 1.2.1.* UEFI Forum.

Rafal Wojtczuk, J. R. (n.d.). Attacking SMM Memory via Intel® CPU Cache Poisoning.

Salihun, D. M. (2007). *BIOS Disassembly Ninjutsu Uncovered.* Wayne: A-List, LLC.

Techie, G. (2010, February 25). *How SLP and SLIC Works*. Retrieved from http://www.guytechie.com/articles/2010/2/25/how-slp-and-slic-works.html

UEFI Spec Working Group. (2012). *Unified Extensible Firmware Interface Specifications 2.3.1 Errata B.* UEFI Forum.

Vincent Zimmer, M. R. (2010). *Beyond BIOS: Developing with the Unified Extensible Firmware Interface* (2nd ed.). Santa Clara: Intel Press.

Wecherowski, F. (2009). A Real SMM Rootkit: Reversing and Hooking BIOS SMI Handlers. *Phrack 66*.

Zimmer, V. (2012, 12 18). *Accessing UEFI from Operating System*. Retrieved from http://vzimmer.blogspot.be/2012/12/accessing-uefi-form-operating-system.html