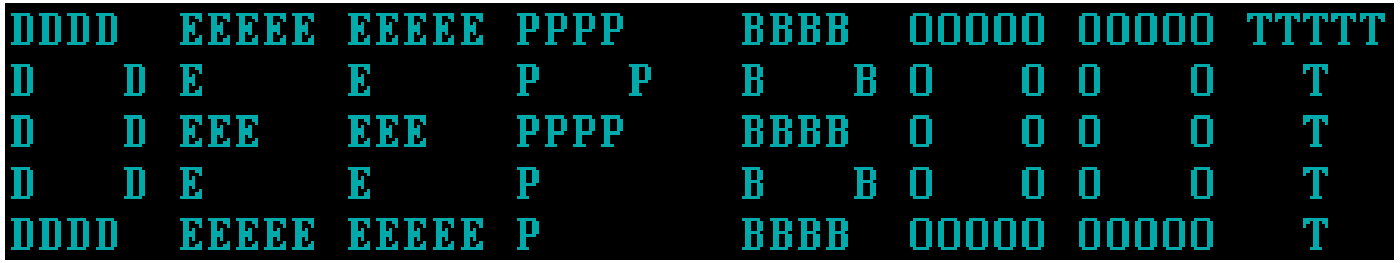
CanSecWest 2012



Nicolás A. Economou

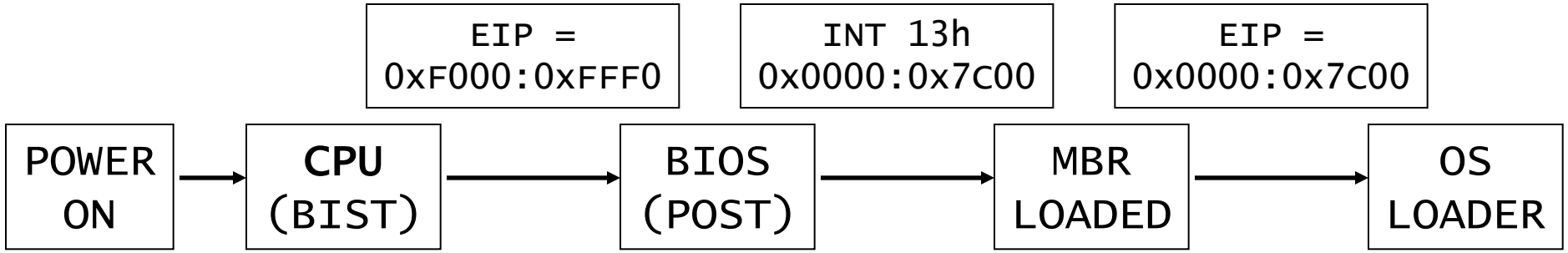
Andrés Lopez Luksenberg

INTRO

- ***"There have been as many new MBR threats found in the first seven months of 2011 as there were in previous three years ...".***

  Symantec Intelligence Report: August 2011

Boot Mechanism



Boot Mechanism

- Then, the code loading the next stage of the OS LOADER is executed.

- Ex. Windows 7
  - 1. MBR
  - 2. Bootmgr (file)
  - 3. Winload.exe (embedded in bootmgr)
  - 4. Winload.exe (file)
  - 5. Rest of the files (kernel, drivers, etc)

Executing from the beginning

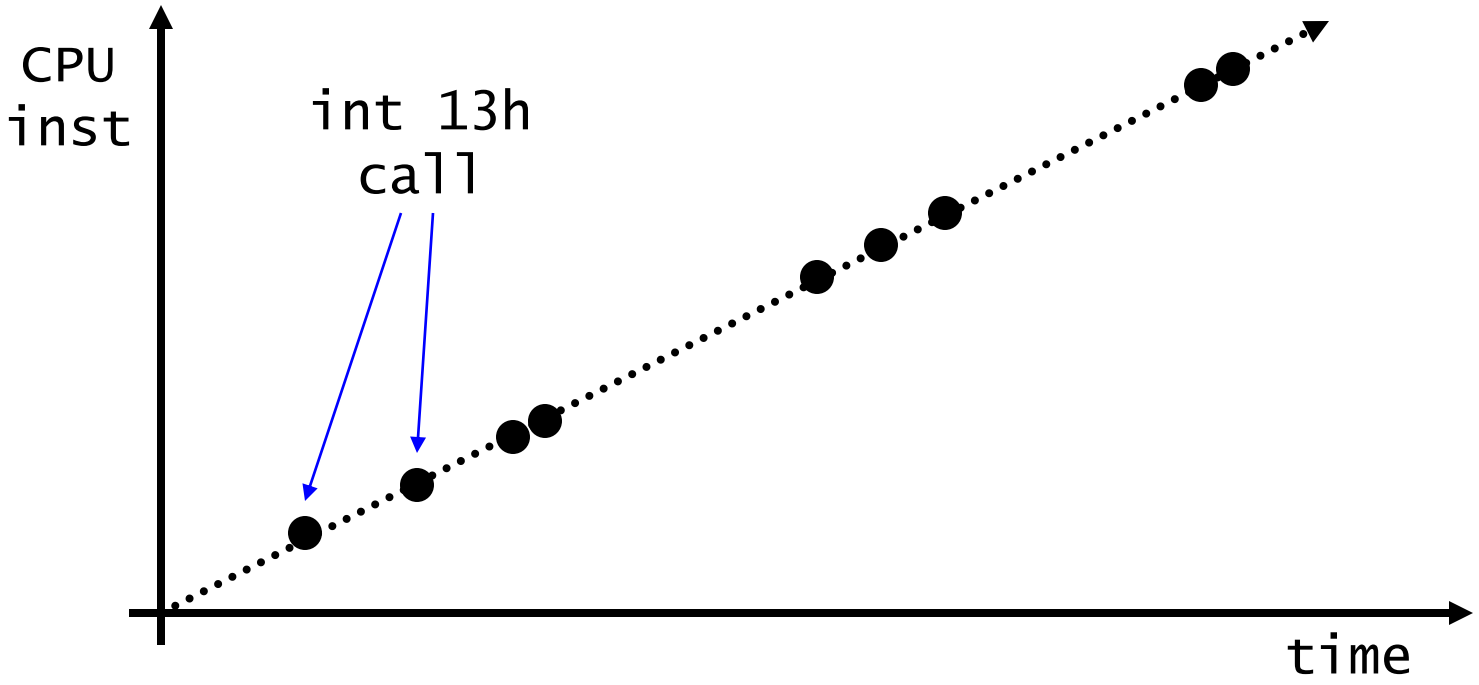
- Putting code in the MBR (Hard Disk)
- Booting from a removable device (CDs, PENDRIVEs, ZIPs, FLOPPY)
- PXE (Network)
- Modified BIOS (executing before 7C00h).
- WARM BOOT (the second beginning)

Common Ways (File Patching)

- Using the BIOS “INT 13H”
  - Pattern Matching:
    - » Go through the disk and patch it
    - » Hook while the OS is loading and patch it (**TDL4**)
  - Understand the file system and add/patch a file (e.g **Computrace v.1**)

Execution/Time graphic

- E.g Hooking the “INT 13H” (REAL MODE)



Non - Common Ways

- Virtualize the OS from the beginning
  - VMBR (Virtual Machine Based Rootkit)
    - E.g "**SubVirt**" (Michigan Univ. + Microsoft Research)
- Deep Boot ;-)

What is Deep Boot ?

- It's a project (technique + research + implementation)
- **Independent from the OS**
- Continuos Execution: NEVER loses the execution control (during the boot)
- Use **only** Intel 80386 features !
- Implementation in C (PIC) and inlined ASM (16 and 32 bits)

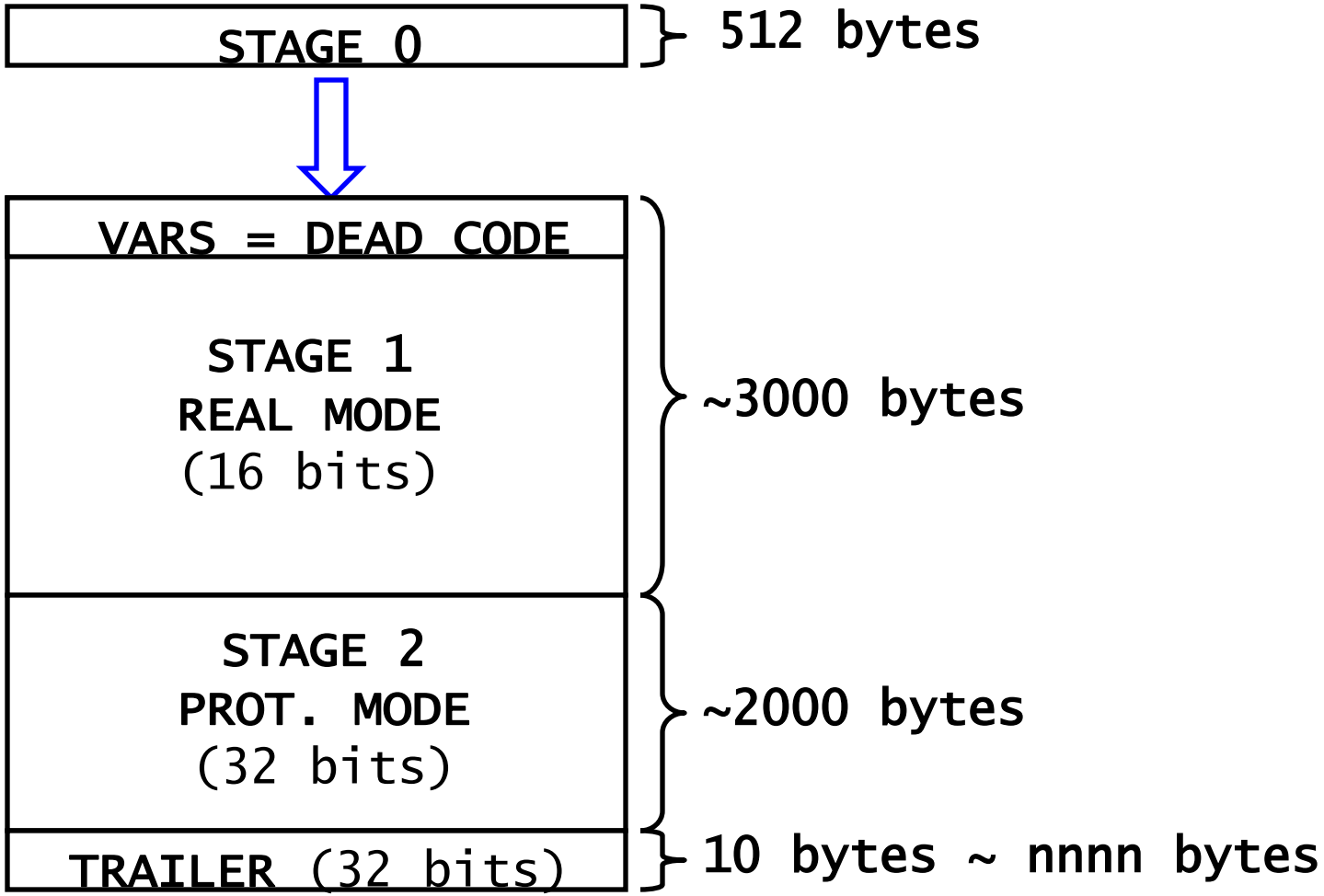
What is Deep Boot ?

- Uses the TRAP FLAG (SINGLE STEP) mechanism to survive and control the execution.
- “Emulates” some CRITICAL instructions
- **Survives to OS CONTEXT SWICHes !**
- Doesn’t affect the “target” OS behavior

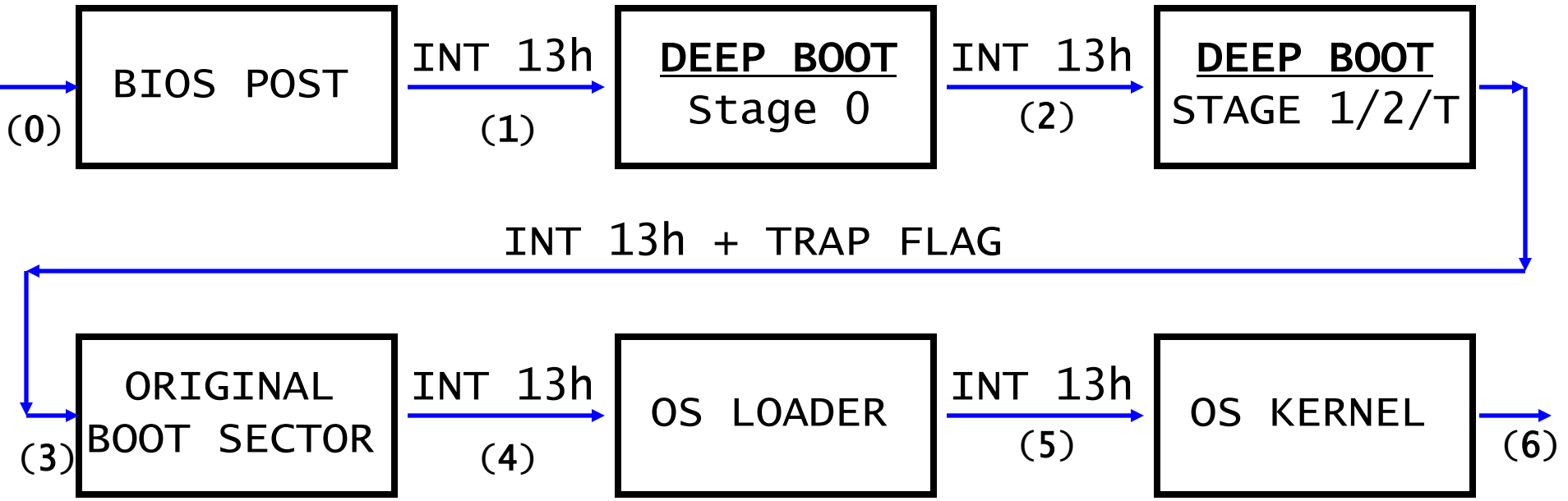
Deep Boot Body

- Formed by:
  - **Stage 0** (stage 1 + stage 2 + trailer LOADER)
  - **Stage 1** (16 bits - REAL MODE handler)
  - **Stage 2** (32 bits - PROT. MODE handler)
  - **Trailer** (32 bits - CALLBACK called by each executed instruction in PROT. MODE)

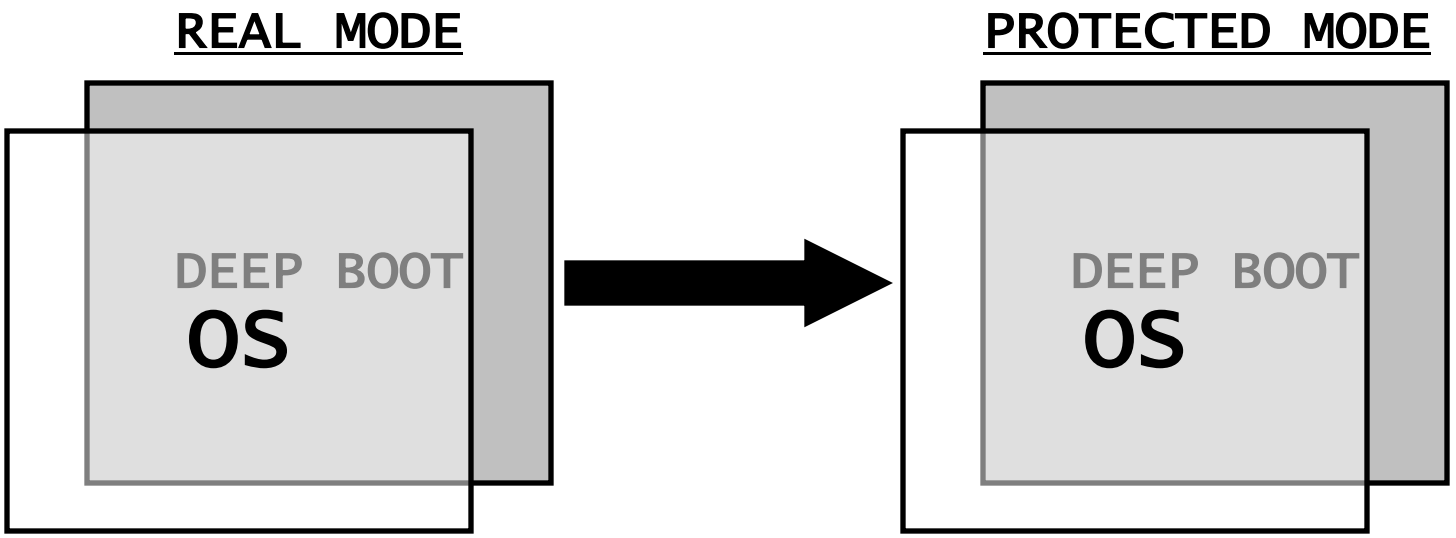
Deep Boot Body today



Taking over from the beginning



Deep Boot Concept



A little of theory ...

- GDT
- IVT
- IDT
- SINGLE STEP

GDT

- GDT: “General Descriptor Table”
- Used by the CPU in **protected mode**
- Designed to “separate/protect” memory areas
- Each entry is called **DESCRIPTOR**
- The first descriptor is not used (NULL)
- Size can vary between 3 and 8192 entries.
- Loaded by the “lgdt” instruction

GDT DESCRIPTOR

- Defines a memory segment (area)
- Referenced with an offset in the GDT (**SELECTOR** – E.g CS, SS, DS, ES, FS,GS)
- 8 byte structure
  - » Base Address: DWORD (32 bits)
  - » Limit: WORD+NIBBLE (20 bits)
  - » Privileges: Code, Data, Gates

IVT

- IVT: “Interrupt Vector Table”
- Used by the CPU in **real mode**
- Used to link an interrupt with a handler
- Interruptions can be generated by IRQs
- The size of each entry is 4 bytes (SEGM:OFFSET)
- Length of 256 entries.

IDT

- IDT: “Interrupt Descriptor Table”
- Used by the CPU in **protected mode**
- Used to link an interrupt with a handler
- Interruptions can be generated by IRQs
- Each entry is called **DESCRIPTOR**
- Size can vary between 0 and 256 entries.
- Loaded by the “lidt” instruction

IDT DESCRIPTOR

- Points to a handler (“callback”).
- Used by the CPU when a interruption is generated
  - E.g: → ZERO DIVISION → exception → the CPU reads the IDT descriptor number 0 → handler
- 8 byte structure
  - » Base Address: DWORD (32 bits)
  - » Selector: WORD (16 bits)
  - » Privileges: Ring from the interrupt can be called

SINGLE STEP

- Produces an exception for each instruction to be executed
- The exception is processed by a “handler” (callback)
- The handler is in the same physical memory as the instruction to be executed (THE TRICK !)
- Used by all debuggers
- Implemented in x86-x64 in the EFLAGS register (Trap Flag)

Intel TRAP FLAG

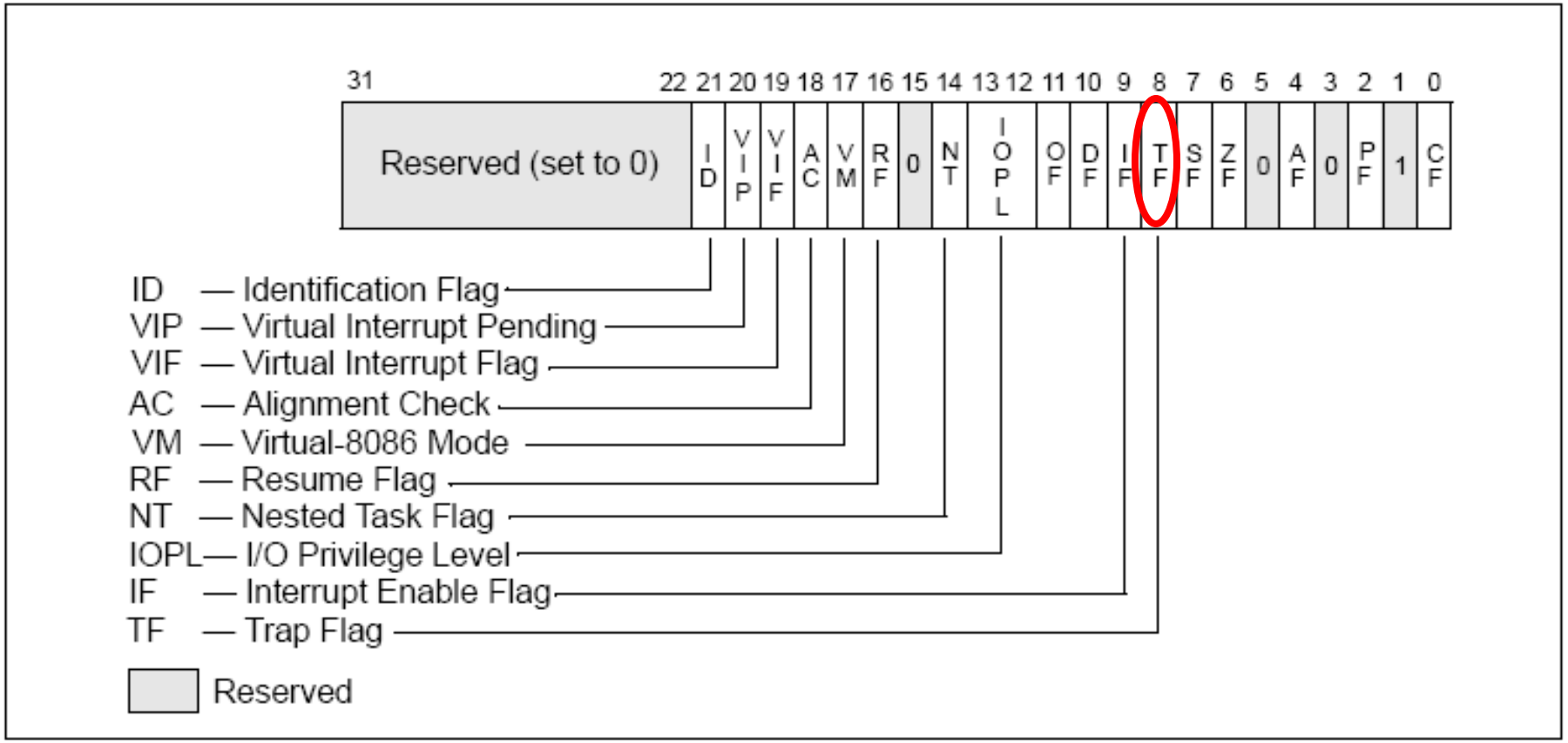
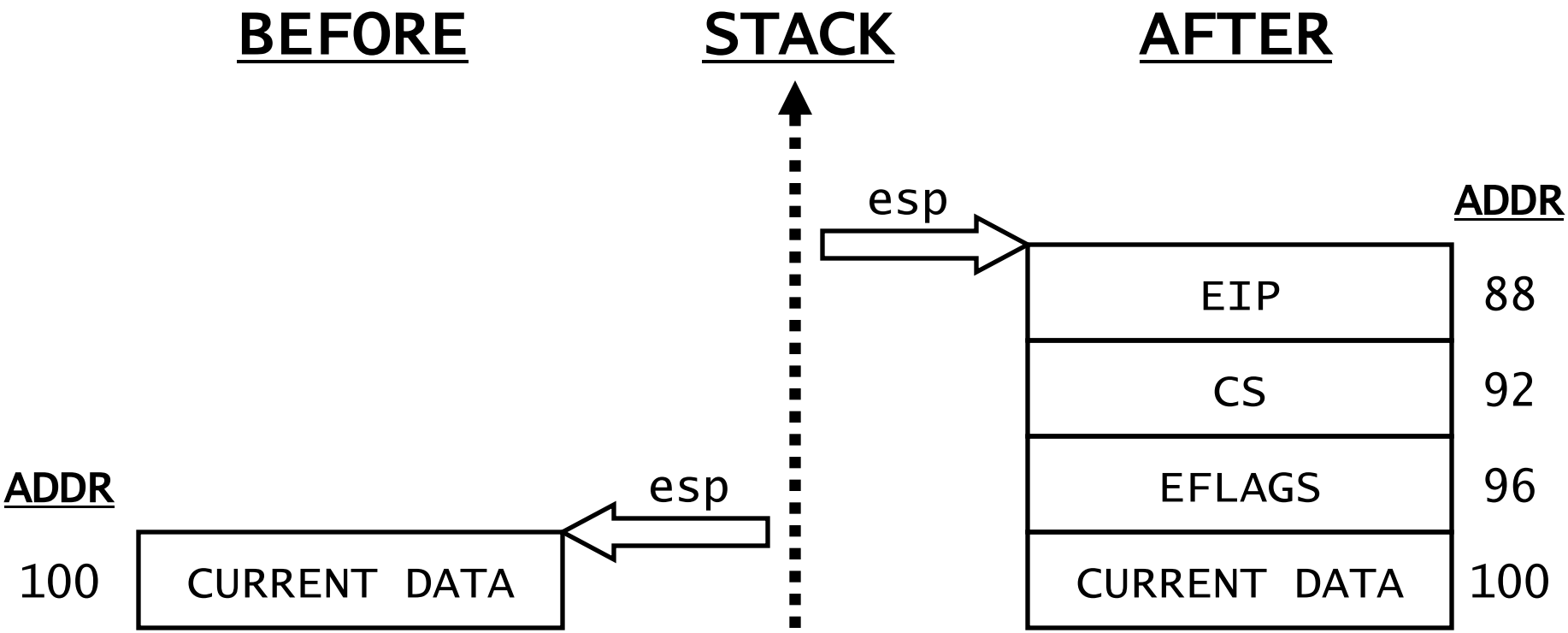


Figure 2-3. System Flags in the EFLAGS Register

Intel TRAP FLAG

- When it's enabled:
  - Invokes the interruption number **1**
  - In REAL MODE uses the IVT
  - In PROTECTED MODE uses the IDT

Handling SINGLE STEPs



Deep Boot Development

A hard work ...

Main Problems

- **Survive:**
  - **OS CONTEXT SWITCHes**:
    - 1. REAL MODE → PROTECTED MODE
    - 2. PROTECTED MODE → PROTECTED MODE
    - 3. PROTECTED MODE → REAL MODE
  - **OS STACK SWITCHes**:
  - Keep **Deep Boot** in a reliable memory area
  - Keep the OS integrity (MEM and REGS)
  - Keep the TRAP FLAG ENABLED ...

REAL MODE → PROT. MODE

- Normal steps:
  - 1. The OS loads the GDT (“lgdt”)
  - 2. The OS loads the IDT (“lidt”)
  - 3. The OS sets the PDE (register CR3)
  - 4. The OS enables the bit 0-31 of the CR0 reg.
  - 5. Finally, the OS executes a JUMP FAR

REAL MODE → PROT. MODE

- With Deep Boot running:
  - The OS loads the GDT (“lgdt”)
  - The OS **ATTEMPTS** to load the IDT (“lidt”)
  - The OS sets the PDE (register CR3)
  - The OS **ATTEMPTS** to enable the bit 0-31 of the CR0 reg.
  - Finally, the OS executes a JUMP FAR (but this time in **REAL MODE** ;-))

REAL MODE → PROT. MODE

- Deep Boot's CONTEXT SWITCH conditions:
  - 1. If the bit 0 of the CR0 register attempted to be enabled
  - 2. If the OS executed a JUMP FAR, RETF, CALL FAR, IRET
    - » If (LAST_CS != CURRENT_CS)

Crossing to PROT. MODE

- Deep Boot steps:
  - 1. Adds/reuses two 32 bit GDT descriptors (code and data)
  - 2. Sets two handlers in the **original IDT** (SINGLE STEP and BREAKPOINT)
  - 3. Loads the **original IDT** (The one OS couldn't !)
  - 4. Enables the bit 0-31 of the CR0 register
  - 5. Executes an **IRETD** to **CURRENT_CS:CURRENT EIP** (Effective cross to PROTECTED MODE)

PROT. MODE → PROT. MODE

- Normal steps:
  - The OS loads a new GDT
  - The OS loads a new IDT
  - The OS enables pagination or changes the PDE

- With Deep Boot running:
  - IDEM

PROT. MODE → PROT. MODE

- Deep Boot steps:
  - If the OS is going to load a new GDT (“lgdt”)
    - **Deep Boot adds/reuses 2 descriptors**
    - **Deep Boot updates the IDT descriptors**
  - If the OS is going to load a new IDT (“lidt”)
    - **Deep Boot sets two handlers in the new IDT (SINGLE STEP and BREAKPOINT)**
  - If the OS changes the PAGE DIRECTORY ENTRY base (CR3 register)
    - **Deep Boot doesn’t do anything ... ???**

PROT. MODE → REAL MODE

- Normal steps:
  - 1. The OS jumps to 16 bit code in PM
  - 2. The OS disables the bit 0 of the CR0 reg.
  - 3. The OS executes a JUMP FAR to RM
  - 4. The OS loads a new IDT pointing to the address 0 (IVT)

PROT. MODE → REAL MODE

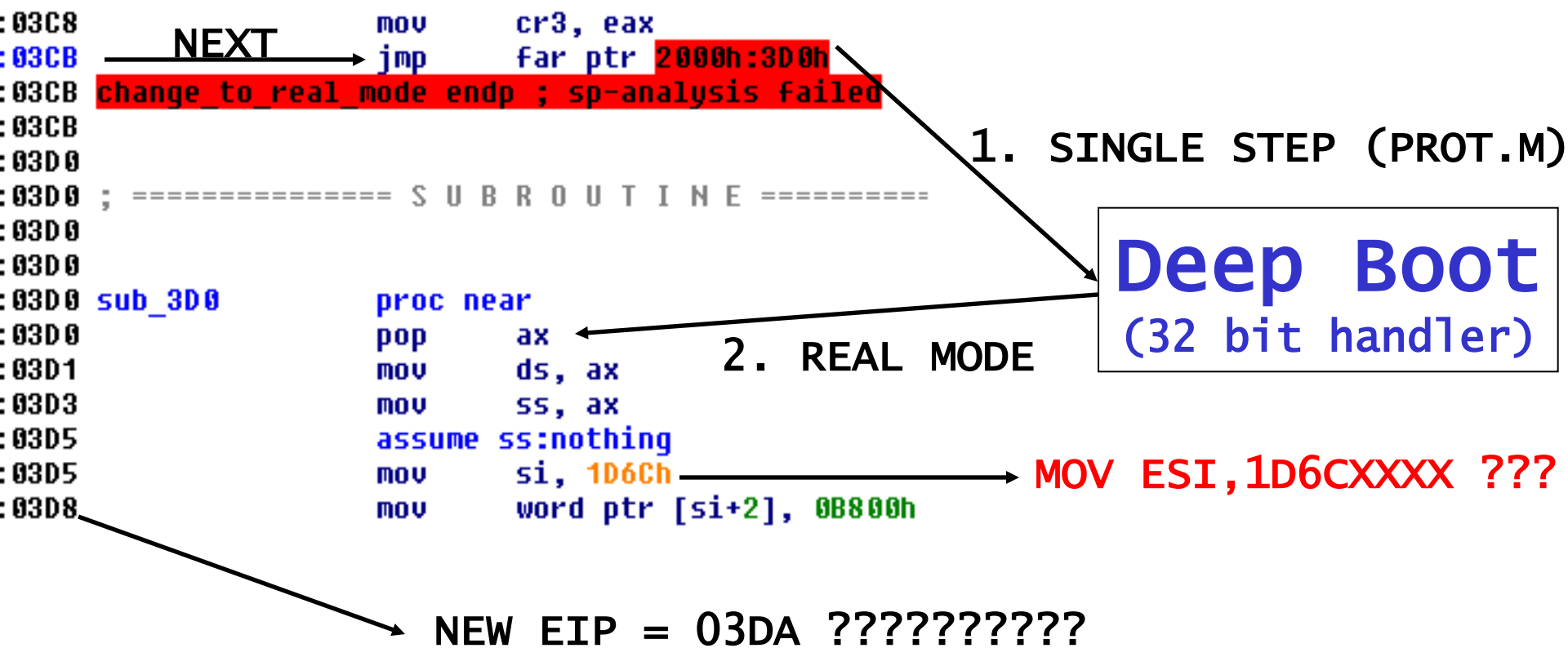
- With Deep Boot running:
  - 1. The OS jumps to 16 bit code (IDEM)
  - 2. If the OS is going to disable the bit 0 of the CR0 reg.
    - **Deep Boot sets a new IDT at 0 address (IVT)**
  - 3. The OS disables the bit 0 of the CR0 register
  - 4. A “JUMP FAR” is produced by the same SINGLE STEP mechanism (“int 1”)
  - 5. The interruption is handled by the REAL MODE handler

PROT. MODE → REAL MODE

- CS = 16 bit code selector
  - JUMP FAR to REAL_MODE → **OK**

- CS = 32 bit code selector
  - JUMP FAR to REAL_MODE → **HALT STATE ???**

Intel Undocumented Mode ?

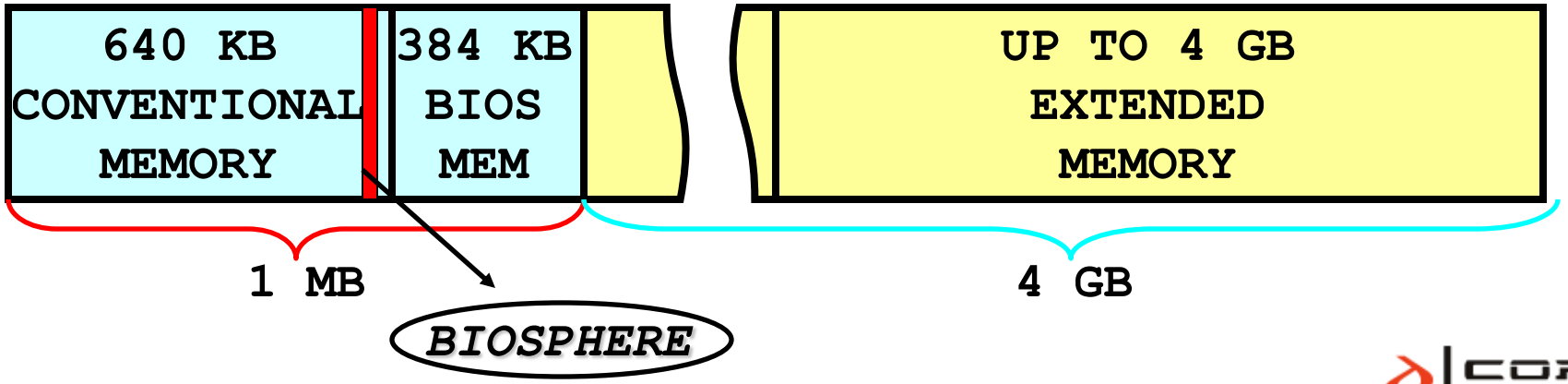


Intel Undocumented Mode ?

- **REAL MODE** → 16 bit native code, 64 kb code and data segments.
- **UNREAL MODE** → 16 bit native code, up to 4 GB data segments (using instructions with prefixes).
- **THIS MODE** (UNREAL MODE 32 ???)→ **32 bit native code** (without prefixes) + UNREAL MODE

BIOSPHERE

- Memory area where Deep Boot is executed.
- Base Address = 9E00h:0000h = 0x9E000
- Located in the first MEGABYTE (R.MODE and P.MODE)
- Code and vars in the same memory area.
- Shared memory with the OS (share the same context).



OS Integrity

- Save and Restore all register status by each SINGLE STEP handled
- Don’t write memory outside the BIOSPHERE
- Don’t change the **normal course** of the execution

Misc

- **Prevent the TRAP FLAG from being disabled ?**
  - No OS attempts to disable it ...
- **OS Stack Switches**
  - Create a STACK trampoline to survive because the stack switch atomicity is not respected by Windows
    - "**MOV SS,reg16**"
    - **… DON'T PUT INSTRUCTIONS PLEASE !!! …**
    - "**MOV ESP,reg32**"
- **Temporary IDT**
  - When the OS doesn't need it (E.g GRUB)

"Emulated/Intercepted" Instructions

- LGDT
- LIDT
- MOV CR0, REG32
- MOV SS, REG16
- MOV ESP, REG32
- JUMP LARGE FAR (RM → PM 32)

Performance

- Each **traced instruction** has a cost
- Average 700.000 inst/seg (in this notebook)
- The idea is to supress most of the handling **without losing control**

Optimization Chances

- Emulate some basic instructions
  - » Conditional jumps
  - » Unconditional jumps
  - » mov reg32,reg32
  - » nops
- Using Breakpoints
  - – Check the source and Skip some repetition instructions (“rep movsX”, “rep stosX”, “rep outsX”)
  - – Skip function calls (DANGEROUS !)

Deep Boot Applications

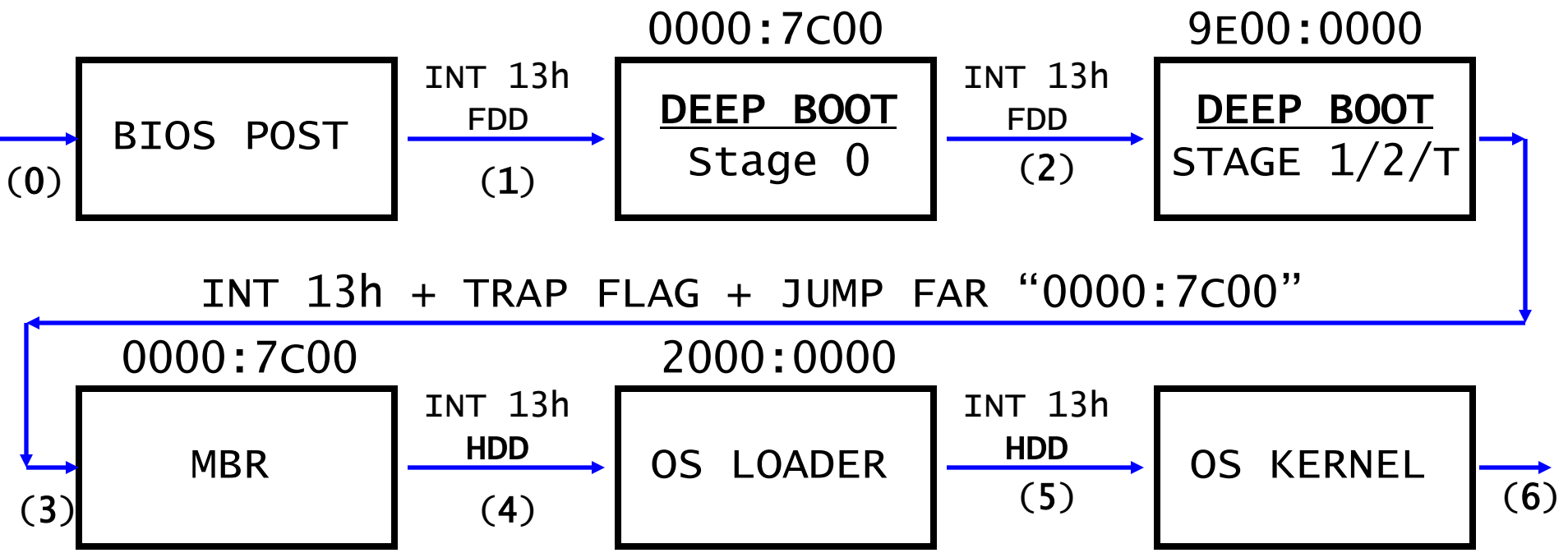
- Rootkits (?)
- Hot Patching (kernel bugs)
- Code Bypass (e.g overwrite the kernel “setuid” function)
- Add new features to the OS
- Interaction with the OS (functions use)
- Encapsulating the OS (Hypervisor)
- Generic kernel debugger in boot time

DEMO TIME

Deep Boot

Trailer

Booting from a Removable Device



DEMO 1

- Target:
  - OpenBSD v5.0

- Boot Device:
  - Virtual FDD
- Cutting Condition:
  - When 30.000.000 instructions are executed
- Mission:
  - Demostrate the Deep Boot operation

DEMO 2

- Target:
  - Arch Linux - kernel v2.6.35

- Boot Device:
  - Virtual FDD
- Cutting Condition:
  - When the **getuid()** function is patched
- Mission:
  - Set **ID = 0** for all users

DEMO 2 – patching getuid()

```
mov eax, fs:[0xc144048c]
mov eax, [eax+1FCh]
    push ebx
    xor ebx,ebx
    mov [eax+4],ebx
    pop ebx
mov eax, [eax+4]
ret
```

PATCH

DEMO 3

- Target:
  - Windows XP SP3 + **Norton 360 AV**

+ Norton from symantec

- Boot Device:
  - HDD (Master Boot Record)
- Cutting Condition:
  - When the “ntkrlnpa.KiSystemStartup()” function is executed
- Mission:
  - Install a **BOOTKIT** in REAL TIME

Supported OSes (until now)

- Windows
  - XP
  - 2003
  - Windows Vista
  - Windows 7
  - Windows 2008 (32 and 64 bits)
- Linux
  - Debian 6.0
  - Arch
- OpenBSD

What is the MBR ?

- MBR (Master Boot Record)
  - First Hard Disk sector
  - Size is 512 bytes
  - Contains the stage 0 (first loader) of any OS
  - Contains the partition table

```
0118  13 61 61 73 0E 4F 74 0B  32 E4 8A 56 00 CD 13 EB  .aas.Ot.2S.V.-.d
0128  D6 61 F9 C3 49 6E 76 61  6C 69 64 20 70 61 72 74  +a-+Invalid.part
0138  69 74 69 6F 6E 20 74 61  62 6C 65 00 45 72 72 6F  ition.table.Erro
0148  72 20 6C 6F 61 64 69 6E  67 20 6F 70 65 72 61 74  r.loading.operat
0158  69 6E 67 20 73 79 73 74  65 6D 00 4D 69 73 73 69  ing.system.Missi
0168  6E 67 20 6F 70 65 72 61  74 69 6E 67 20 73 79 73  ng.operating.sys
0178  74 65 6D 00 00 00 00 00  00 00 00 00 00 00 00 00  tem............
0188  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0198  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
01A8  00 00 00 00 00 00 00 00  00 00 00 00 00 2C 44 63  .............,Dc
01B8  C2 B4 C2 B4 00 00 80 01  01 00 07 FE BF 08 3F 00  -¦-¦.......¦+.?.
01C8  00 00 8A B6 7F 00 00 00  00 00 00 00 00 00 00 00  ...¦............
01D8  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
01E8  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
01F8  00 00 00 00 00 00 55 AA                           ......U¬........
```

Protections ?

- BIOS + TPM (Chip)
  - BitLocker (Microsoft)
    - » Windows Vista Ultimate/Enterprise
    - » Windows 7 Ultimate/Enterprise
    - » Windows 2008
  - TrustedGRUB
    - » Linux

HLT

- Questions ?