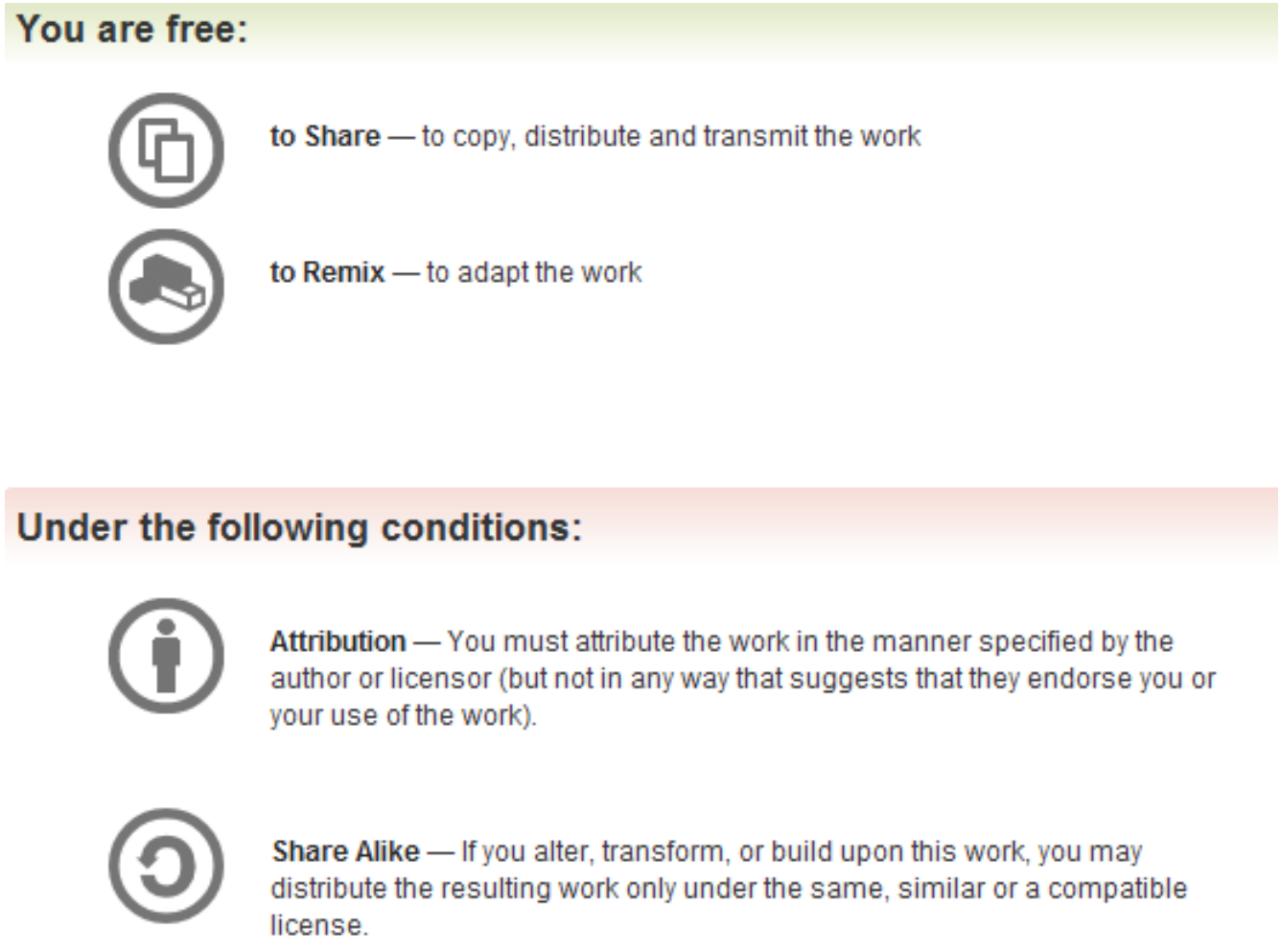
Intermediate x86
Part 3

Xeno Kovah – 2010

xkovah at gmail

All materials are licensed under a Creative Commons “Share Alike” license.

- http://creativecommons.org/licenses/by-sa/3.0/



Intermission
RDTSC

RDTSC—Read Time-Stamp Counter

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 31	RDTSC	Valid	Valid	Read time-stamp counter into EDX:EAX.

- What's the time? Time to ReaD (the) Time-Stamp Counter! (I know, it doesn't even sort of work)
- The TSC is a 64bit counter introduced in the Pentium
- Set to 0 when processor reset, incremented on each clock cycle thereafter.
- Can set a flag (in CR4) so that only Ring 0 can use the instruction (AFAIK no one does this)

RDTSC 2

- Processor families increment the time-stamp counter differently
- RDTSC in practice
  - Timing code for performance reasons
  - Code which checks for the presence of a debugger by timing its own execution
- Further reference - Vol. 3b chapter 18.11

Lab: Guestimate.c

- Simple code to use the timestamp counter to time how long it takes to run some code.

Lab: NavelGaze.c

- Code which looks at itself and changes its behavior if it takes too long to execute (e.g. if a breakpoint was set)
- Pop quiz hot shot: You gots malware that's altering its behavior in response to your breakpoints. What do you do? What DO you DO?!?!

Conclusion:

- Human stepping through code easy to detect
  - But analyst will hopefully grasp the nature of the timing check, and work around it
- IMHO RDTSC less fun than a barrel of monkeys but more fun than a tube sock of scorpions

Interrupts and Exceptions

- “Interrupts and exceptions are events that indicate that a condition exists somewhere in the system, the processor, or within the currently executing program or task that requires the attention of a processor.”
- “When an interrupt is received or an exception is detected, the currently running procedure or task is suspended while the processor executes an interrupt or exception handler. When execution of the handler is complete, the processor resumes execution of the interrupted procedure or task.”
- “The processor receives interrupts from two sources:
  - External (hardware generated) interrupts.
  - Software-generated interrupts.”

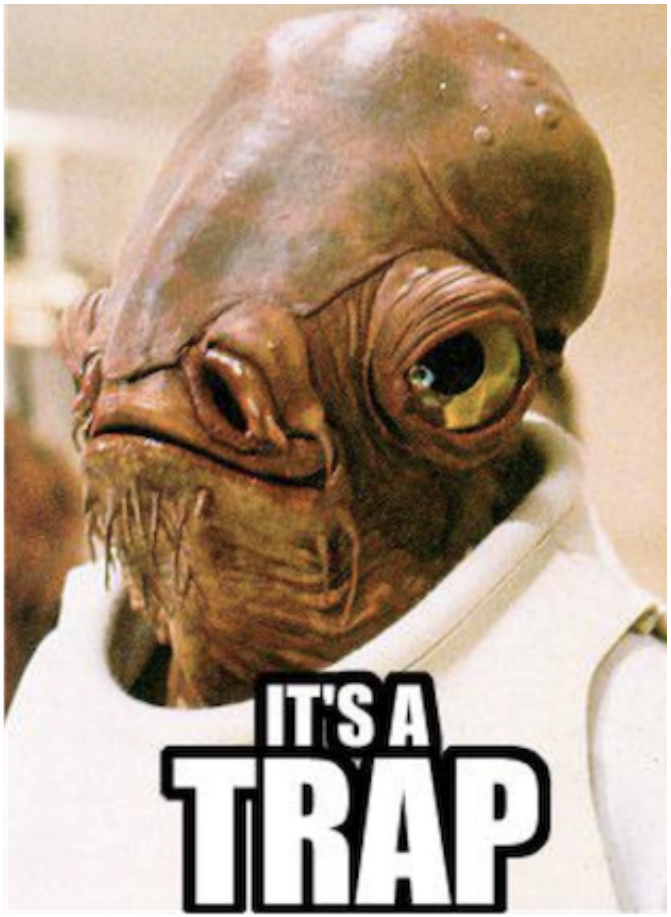
Difference between Interrupt and Exception?

- Exceptions typically indicate error conditions, whereas interrupts typically indicate events from external hardware.
  - "Exception - E is for error" ;)
- Interrupts clear the Interrupt Flag (IF - talked about later), Exceptions do not.
- 3 categories of exception:
  - Fault - recoverable - pushed EIP points to the faulting instruction
  - Trap - recoverable - pushed EIP points to the instruction following the trapping instruction
  - Abort - unrecoverable - may not be able to save EIP where abort occurred

Fault: EIP points at faulting instruction

“Zis is all YOUR fault!”

Trap: EIP points at instruction ***after*** the trapping instruction



Abort: unrecoverable, may not be able to save EIP…

Saving State

- When it says the current procedure is suspended, what does it actually mean?
- Need to save the relevant state for the current activity, so that the state can be restored after processing the interrupt
- The hardware itself saves relatively little state to the stack in the event of an interrupt. An interrupt handler on the other hand may choose to store all of the registers so that they can be replaced when the interrupt has been processed,

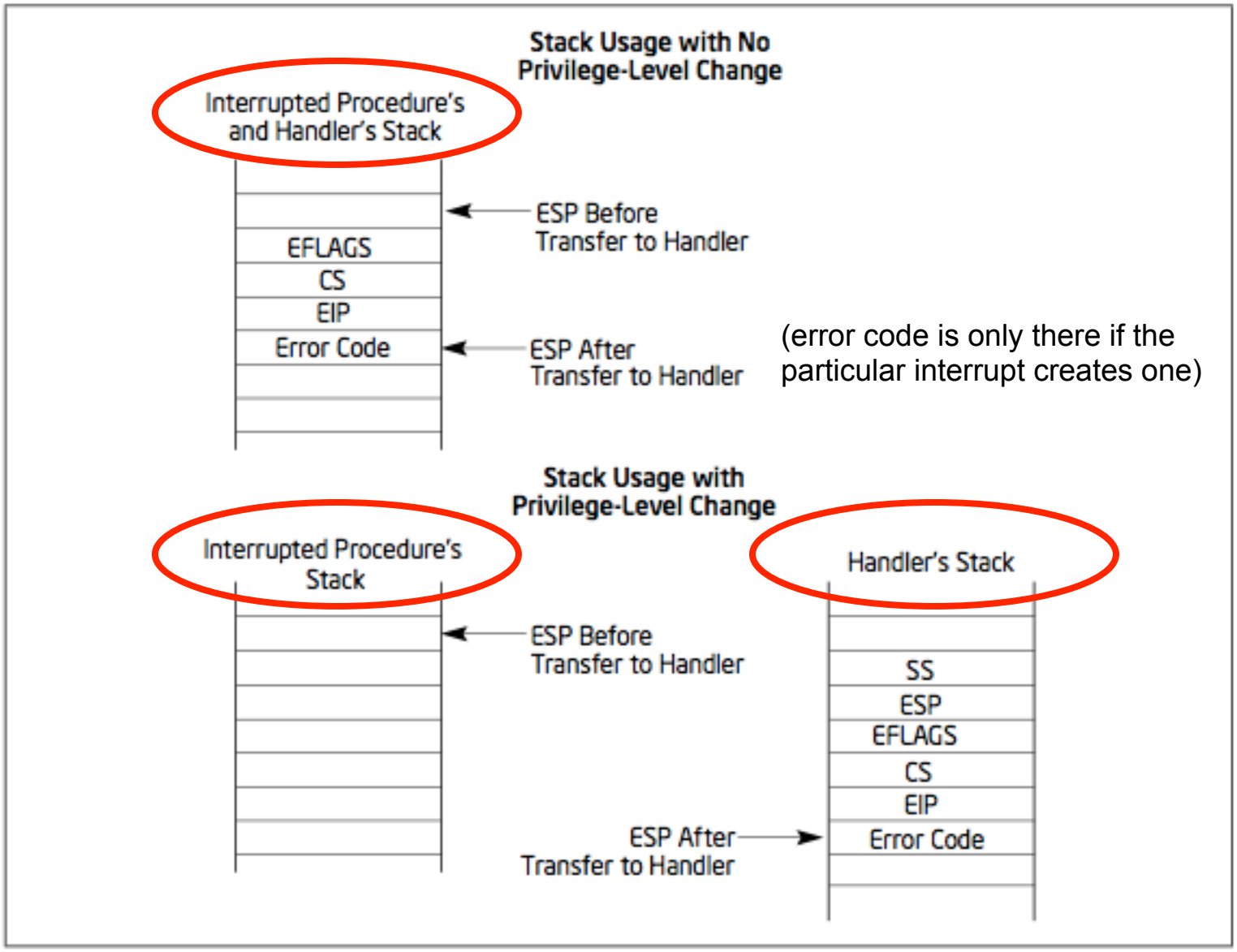


Figure 6-5. Stack Usage on Transfers to Interrupt and Exception Handling Routines

From Vol. 1

You couldn't find your handler's stack with both hands

- A good question (asked by my wife) is: when there is a privilege change how does the hardware find the stack for the interrupt handler in order to push the saved state onto *that* stack rather than that of the interrupted procedure?
- It consults a structure called the Task State Segment (TSS) to find the new SS and ESP it will use.

A little bit about “tasks”

- The Intel documentation often references the notion of a “task”.
- Eventually I decided to remove most of the description of how tasks work, because it is tangential to the more important stuff, and I anticipate a lack of time.
- If you’re a conspiracy theorist, you are invited to read Vol 2a, Section 6 to find out “what Xeno doesn’t want you to know!”….But then…that’s just what I’d be expecting, isn’t it?
- Anyway, the Task State Segment (TSS) is something which **must** be used by the OS by virtue of being consulted on privilege changing interrupts, so we’ll at least talk about that.
- There’s a dedicated 16 bit Task Register (TR) which holds a segment selector selecting an entry in the GDT of Task Segment type. (Load/store task register with LTR, STR instructions respectivly
- The task segment points to the Task State Segment, which is just a large data structure of mostly saved register values.

31	15	0	
I/O Map Base Address	Reserved	T	100
Reserved	LDT Segment Selector		96
Reserved	GS		92
Reserved	FS		88
Reserved	DS		84
Reserved	SS		80
Reserved	CS		76
Reserved	ES		72
EDI			68
ESI			64
EBP			60
ESP			56
EBX			52
EDX			48
ECX			44
EAX			40
EFLAGS			36
EIP			32
CR3 (PDBR)			28
Reserved	SS2		24
ESP2			20
Reserved	SS1		16
ESP1			12
Reserved	SS0		8
ESP0			4
Reserved	Previous Task Link		0

Reserved bits. Set to 0.

Figure 6-2. 32-Bit Task-State Segment (TSS)

15	0
Task LDT Selector	42
DS Selector	40
SS Selector	38
CS Selector	36
ES Selector	34
DI	32
SI	30
BP	28
SP	26
BX	24
DX	22
CX	20
AX	18
FLAG Word	16
IP (Entry Point)	14
SS2	12
SP2	10
SS1	8
SP1	6
SS0	4
SP0	2
Previous Task Link	0

Figure 6-10. 16-Bit TSS Format

!descriptor TSS printing

- I added the following commands to the ! descriptor windbg plugin
  - !descriptor TSS32 <address>
  - !descriptor TSS16 <address>
- Just reads memory at the given address and prints out value according to the structure definition
- Use with !descriptor GDT <index> to find the base address of a GDT descriptor of 16/32 bit TSS type

Interrupt Descriptor Table (IDT)

- And array of <= 256 8-byte descriptor entries. 0 through 31 are reserved for architecture-specific exceptions and interrupts. 32-255 are user defined.
- While it interacts with segments, you can think of it as being an array of function pointers, and when interrupt *n* is invoked by software or hardware, execution transfers to the address pointed to by the *nth*descriptor in the table.

Table 5-1. Protected-Mode Exceptions and Interrupts

Vector No.	Mne-monic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	RESERVED	Fault/ Trap	No	For Intel use only.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT 3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction.[2]
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.

Table 5-1. Protected-Mode Exceptions and Interrupts

Vector No.	Mne-monic	Description	Type	Error Code	Source
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.
15	—	(Intel reserved. Do not use.)		No	
16	#MF	x87 FPU Floating-Point Error (Math Fault)	Fault	No	x87 FPU floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Fault	Yes (Zero)	Any data reference in memory.[3]
18	#MC	Machine Check	Abort	No	Error codes (if any) and source are model dependent.[4]
19	#XM	SIMD Floating-Point Exception	Fault	No	SSE/SSE2/SSE3 floating-point instructions[5]
20-31	—	Intel reserved. Do not use.			
32-255	—	User Defined (Non-reserved) Interrupts	Interrupt		External interrupt or INT *n* instruction.

For lots of details on 0-19, see Vol. 3a Sect. 5.15

The SegFault is a Lie!

- On *nix systems when you access somewhere you're not supposed to, the program will be shut down with the given reason being "Segmentation Fault"
- Looking back at the table we see there is no listed Segmentation Fault.
- In fact, for most segmentation errors you can get, they will generate a general protection fault
- And we know that segmentation isn't used to protect memory, so usually if you access invalid memory you're getting a page fault.
- The reality of the SegFault is that it's a unix Signal, SIGSEGV (signal segment violation). But signals are an OS level abstraction, and this "fault" isn't a x86 fault.
- (I would welcome anyone digging into a *nix and confirming that things like page faults and general protection faults get turned into things like SIGSEGV signals)

Software-Generated Interrupts

- INT n - Invoke IDT[n]
  - Important: While you can invoke anything in the IDT, some interrupts expect an error code. INT does not push any error code, and therefore a handler may not behave correctly.
  - Interrupts generated in software with the INT n instruction cannot be masked by the IF flag in the EFLAGS register. (Talked about later)

- IRET - returns from an interrupt, popping all the saved state back into the correct registers

- INT 3 - The special 0xCC opcode form of this has an extra caveat that when hardware virtualization is used, it doesn't intercept this interrupt like it would do with others, and just passes it to the OS debug interrupt handler

- INTO - Invoke overflow interrupt if the overflow flag (OF) in EFLAGS is set to 1 (like a conditional INT 4)

- UD2 - Invoke invalid opcode interrupt (same as INT 6)

Lab: TryToRunTryToHide.c BreakOnThruToTheOtherSide.c

- Taking a look at what changes and what doesn't when jumping from userspace to kernel via a software interrupt
- Showing the kernel stack values being derived from the TSS
- Using rootkit hooking technique you're not expected to understand yet to intercept the interrupt ;)

Changed Register Results

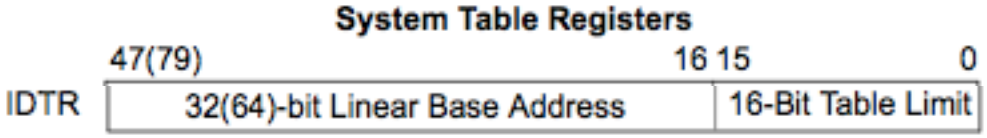
(everything else that we were watching was the same except eip obviously)

Userspace	Kernel
CS = 0x1B	CS = 0x08
SS = 0x23	SS = 0x10
FS = 0x3B	FS = 0x30
ESP = 0x12FDDC	ESP = 0xF7A2ADCC
EFLAGS:IF = 1	EFLAGS:IF = 0

How is the IDT found?

- There is a specific register which points at the base (0th entry) of the IDT. The IDT Register is named IDTR ;)
- When interrupt/exception occurs, the hardware automatically
  - consults the IDTR
  - finds the appropriate offset in the IDT
  - pushes the saved state onto the stack
  - changes EIP to the address of the interrupt handler, as read from the IDT entry (interrupt descriptor).

IDTR Format



From Vol 3a. Figure 2-5

- The upper 32 bits of the register specify the *linear* address where the IDT is stored. The lower 16 bits specify the size of the table in bytes.
- Special instructions used to load a value into the register or store the value out to memory
  - LIDT - Load 6 bytes from memory into IDTR
  - SIDT - Store 6 bytes of IDTR to memory
- Structured the same way as the GDT
- Also, WinDbg displays upper 32 bits as idtr, and lower parts as idtl.

IDTR Usage

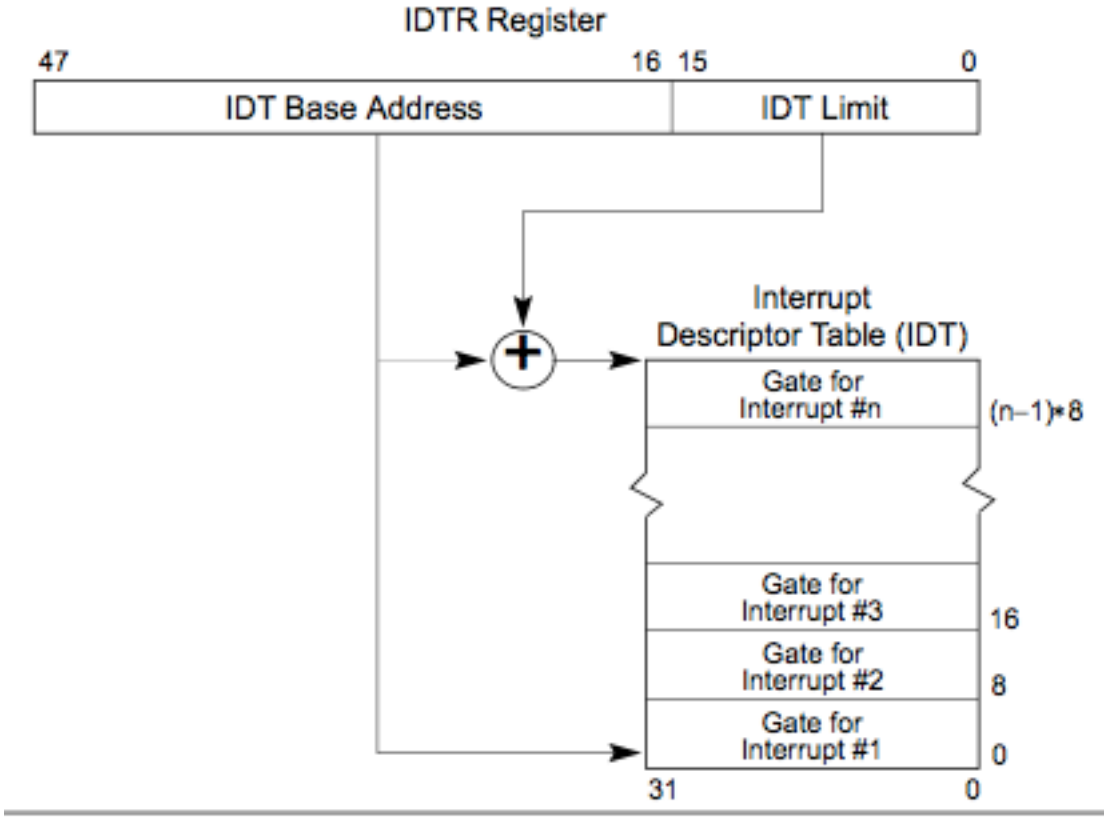
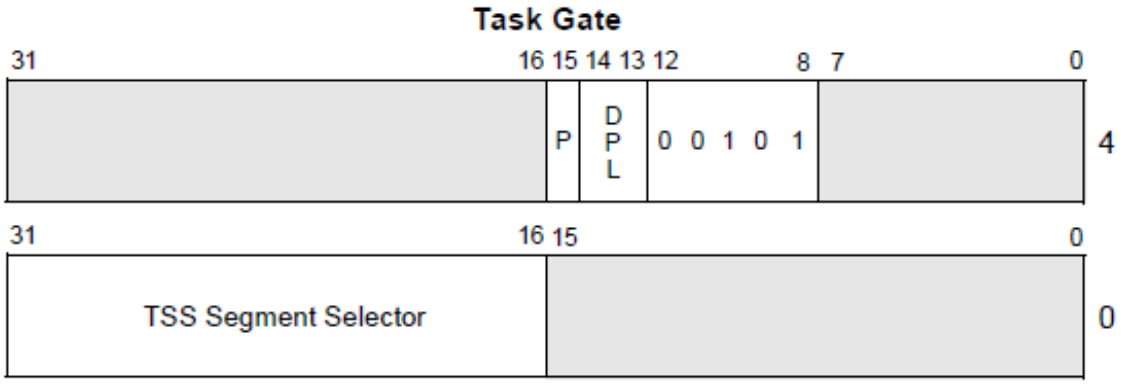


Figure 5-1. Relationship of the IDTR and IDT

Interrupt Descriptors

- The descriptors in the IDT describe one of three *gate* types
  - Trap Gate
  - Task Gate
  - Interrupt Gate
- "The only difference between an interrupt gate and a trap gate is the way the processor handles the IF flag in the EFLAGS register." Discussed later, but from this you can infer that a Trap *Exception* isn't related to a Trap *Gate*. Since there's the difference between where EIP points for trap vs interrupt exceptions.
- Gates are used in the IDT to facilitate control flow transfers between privilege levels

Task Gate Descriptor



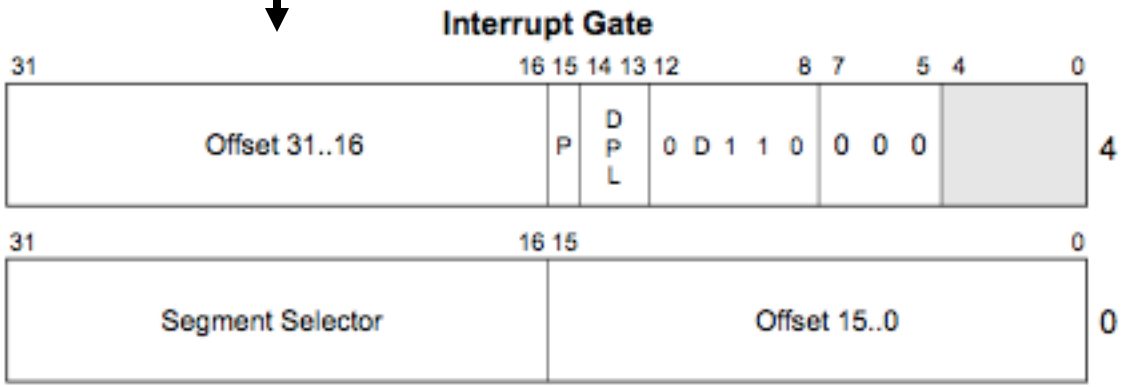
DPL	Descriptor Privilege Level
Offset	Offset to procedure entry point
P	Segment Present flag
Selector	Segment Selector for destination code segment
D	Size of gate: 1 = 32 bits; 0 = 16 bits

Reserved

Descriptors not in use should have P = 0

As I said before, I'm not really going to talk about tasks. But as you can see their gate descriptors are booooorrrrriiiinnnggggg!

Interrupt Gate Descriptor



Note that the two halves of the offset form a 32 bit address.

DPL	Descriptor Privilege Level
Offset	Offset to procedure entry point
P	Segment Present flag
Selector	Segment Selector for destination code segment
D	Size of gate: 1 = 32 bits; 0 = 16 bits

Reserved

Descriptors not in use should have P = 0

Figure 5-2. IDT Gate Descriptors

Trap Gate Descriptor

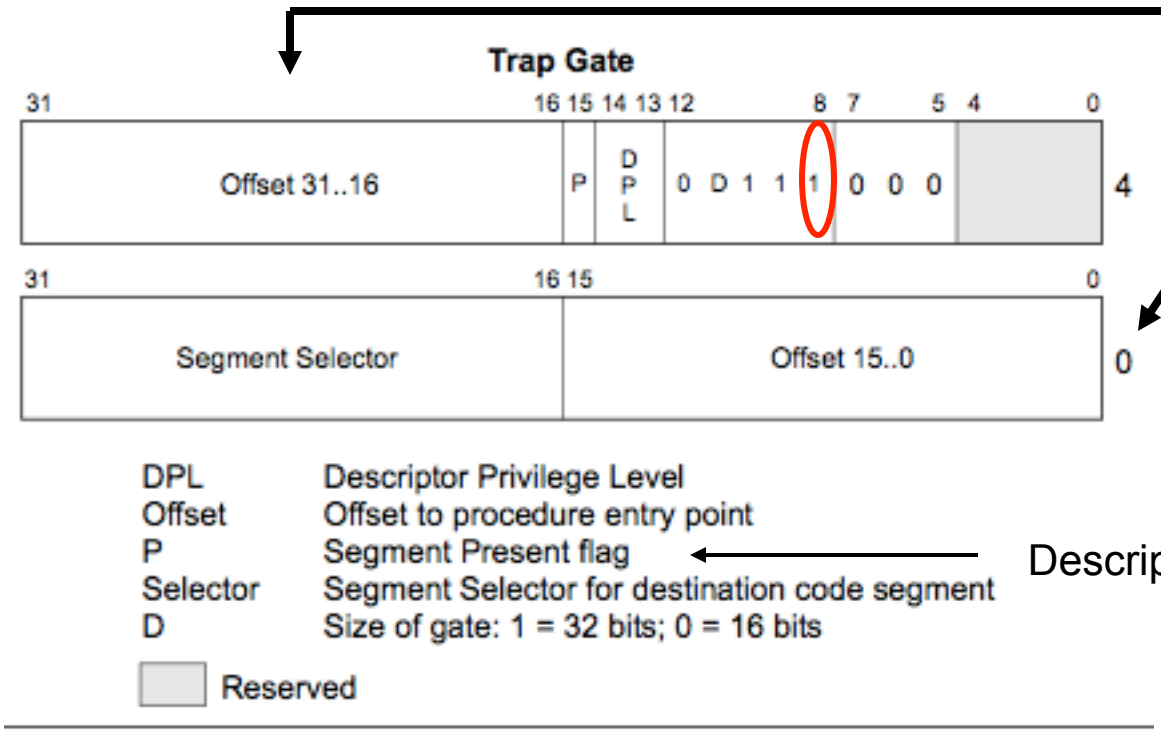


Figure 5-2. IDT Gate Descriptors

Note that the two halves of the offset form a 32 bit address

Descriptors not in use should have P = 0

Blink and you'll miss it…if you're unable to see red ovals after having blinked. The format is the same except for one bit.

Descriptor Descriptions

- The DPL is again the Descriptor Privilege Level. And it is only checked when a descriptor is accessed by a software interrupt, in which case it is only allowed if CPL <= DPL (ignored on hardware interrupts)
- Note that the descriptor specifies a segment selector and 32 bit address. Why that looks like a “logical address” aka “far pointer” to me!
- D flag specifies whether you’re jumping into a 16 or 32 bit segment.
- P (Present) flag

IDT Relation to Segments

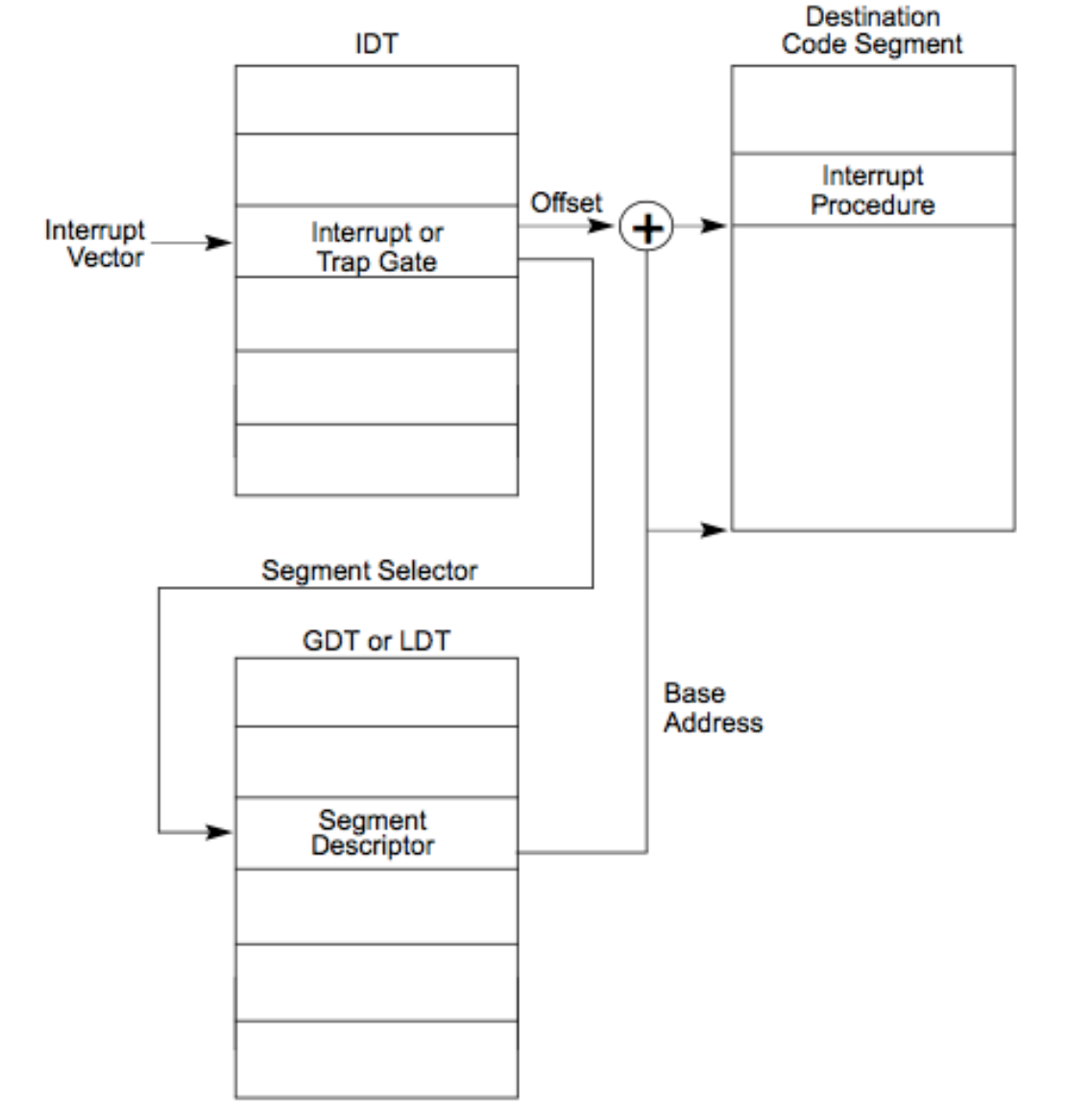


Figure 5-3. Interrupt Procedure Call

Lab: Pearly Gates

- We can take a look at what types of gates are used in the IDT by using this new command I added to the descriptor plugin:
- !descriptor IDT
- !descriptor IDT_FULL
- !descriptor DUMP_IDT_TYPES
- Can also use the built in !idt and !idt –a commands

Interrupt Masking

- It is sometimes useful to disable some interrupts. This is called “masking” the interrupt.
- The Interrupt Flag (IF) in the EFLAGS register is cleared automatically whenever an interrupt occurs through an interrupt gate (but not a trap gate).
- Maskable interrupts can be manually masked by clearing IF. IF can be cleared with the **CLI** instruction, or set with **STI** (both ring-0-only instructions). But only if you have sufficient privileges (talked about later)

Interrupt Masking Exceptions

- The IF does not mask the explicit invocation of an interrupt with the INT instruction
- The IF does not mask a Non Maskable Interrupt - IDT[2] (duh ;))

Red Pill

- Joanna Rutkowska, 2004 - “Red Pill... or how to detect VMM using (almost) one CPU instruction”
  http://www.invisiblethings.org/papers/redpill.html
- Using SIDT (Store Interrupt Descriptor Table Register) instruction to profile the current value in the IDTR
- She had found that the most significant byte of the IDTR had a predictable value in VMWare 4 and VirtualPC, which was different from what it was in an unvirtualized system

Lab: VerboseRedPill.c

- As opposed to:

```
int swallow_redpill () {
  unsigned char m[2+4], rpill[] = "\x0f\x01\x0d\x00\x00\x00\x00\xc3";
  *((unsigned*)&rpill[3]) = (unsigned)m;
  ((void(*)())&rpill)();
  return (m[5]>0xd0) ? 1 : 0;
}
```

Red Pill 2

- The fundamental problem is that while LIDT is a privileged instruction (requires Ring 0), SIDT is not (and probably should be).
- For more examples of why x86 as a platform is not meant to be transparently virtualized see these:
  - Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor http://www.cs.nps.navy.mil/people/faculty/irvine/publications/2000/VMM-usenix00-0611.pdf
  - Compatibility is Not Transparency: VMM Detection Myths and Realities http://www.stanford.edu/~talg/papers/HOTOS07/vmm-detection-hotos07.pdf
- Implication is that malware can detect if it's being virtualized…but…as more hosts are virtualized, there's disincentive for malware to not run in virtualized environments.

Ironic Table

2.7 SYSTEM INSTRUCTION SUMMARY

System instructions handle system-level functions such as loading system registers, managing the cache, managing interrupts, or setting up the debug registers. Many of these instructions can be executed only by operating-system or executive procedures (that is, procedures running at privilege level 0). Others can be executed at any privilege level and are thus available to application programs.

Table 2-2 lists the system instructions and indicates whether they are available and useful for application programs. These instructions are described in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A & 2B*.

Table 2-2. Summary of System Instructions

Instruction	Description	Useful to Application?	Protected from Application?
LLDT	Load LDT Register	No	Yes
Instruction	**Description**	**Useful to Application?**	**Protected from Application?**
SLDT	Store LDT Register	No	No
LGDT	Load GDT Register	No	Yes
SGDT	Store GDT Register	No	No
LTR	Load Task Register	No	Yes
STR	Store Task Register	No	No
LIDT	Load IDT Register	No	Yes
SIDT	Store IDT Register	No	No
MOV CR*n*	Load and store control registers	No	Yes

Stray Instructions Adopted Along The Way

- SIDT/LIDT –Store/Load IDTR
- STI/CLI – Set/Clear Interrupt Flag(IF)
- STR/LTR – Load/Store Task Register (TR)
- INT n – Software Interrupt, invoke handler n
- UD2 – Undefined Instruction Interrupt
- INTO – Overflow Interrupt
- 0xCC form of INT 3 – Breakpoint Interrupt
- IRET – Return from interrupt