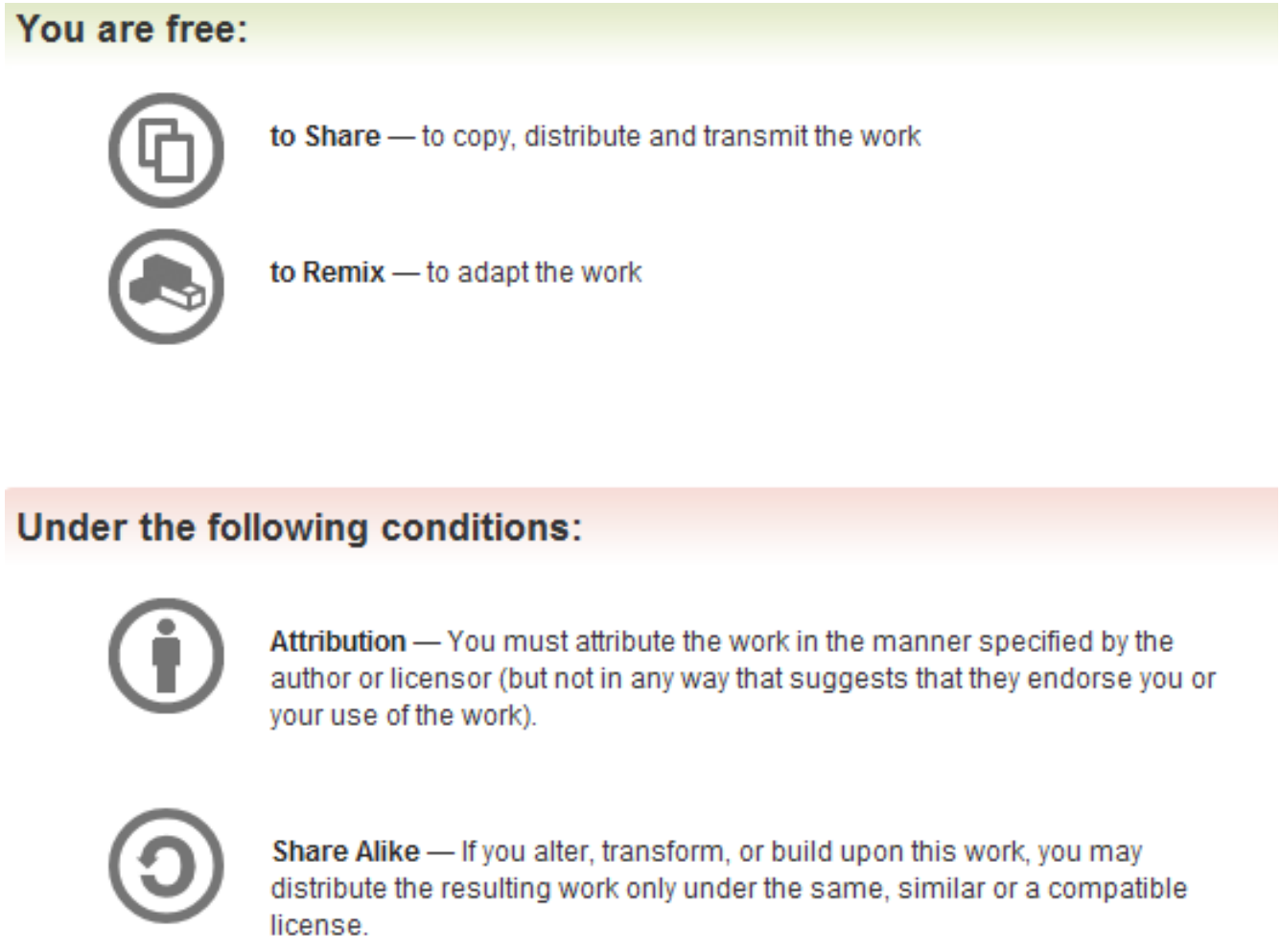
Intermediate x86
Part 4

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/



Interrupts & Debugging

- We know that IDT[3] is the Breakpoint Exception, and that it's important enough for INT 3 to have a separate one byte opcode form (0xCC).
- INT 3 is what debuggers are using when they say they are setting a "software breakpoint" (the default breakpoint in most cases)
- When a debugger uses a software breakpoint, what it does is overwrite the first byte of the instruction at the specified address. It keeps its own list of which bytes it overwrote and where. Then when breakpoint exception is received, it looks up the location, replaces the original byte and lets the instruction execute normally. Then typically it overwrites the first byte again (subject to configuration) so that the breakpoint will be hit if the address is executed again.

Lab: ProofPudding.c

- A program which reads its own memory in order to confirm that when a breakpoint is set, it overwrites a byte with the 0xCC form of the breakpoint interrupt, INT 3

Hardware Support for Debugging

Vol. 3b, Sect. 18

- Most debuggers also have support for something called a “hardware breakpoint”, and these breakpoints are more flexible than software breakpoints in that they can be set to trigger when memory is read or written, not just when it’s executed. However only 4 hardware breakpoints can be set.
- There are 8 debug registers DR0-DR7
  - DR0-3 = breakpoint linear address registers
  - DR4-5 = reserved (unused)
  - DR6 = Debug Status Register
  - DR7 = Debug Control Register
- Accessing the registers requires CPL == 0
  - MOV DR, r32
  - MOV r32, DR

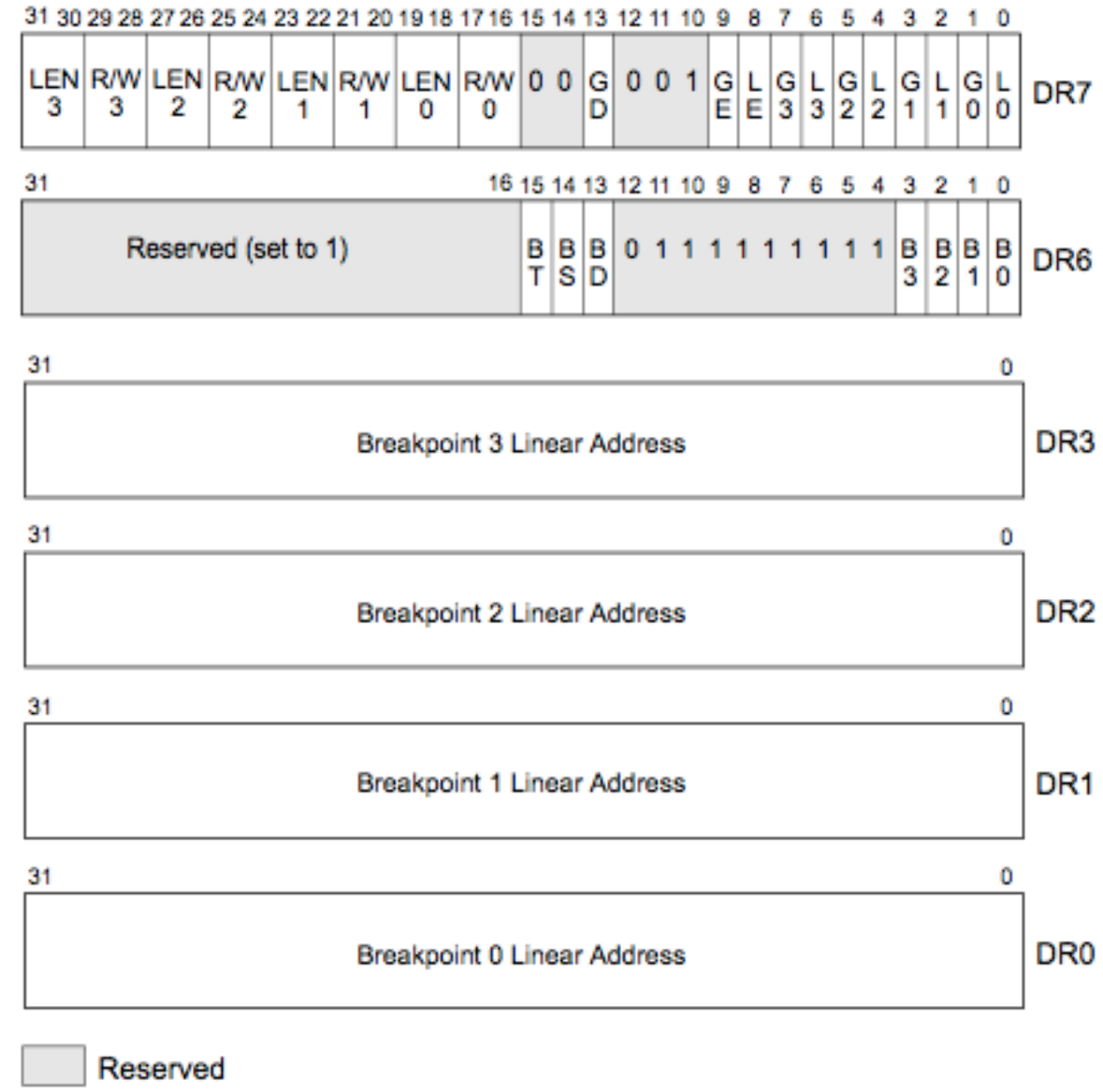
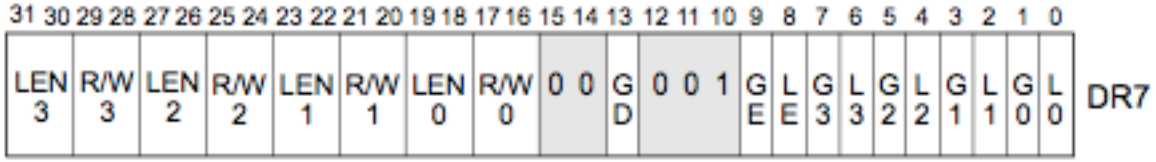


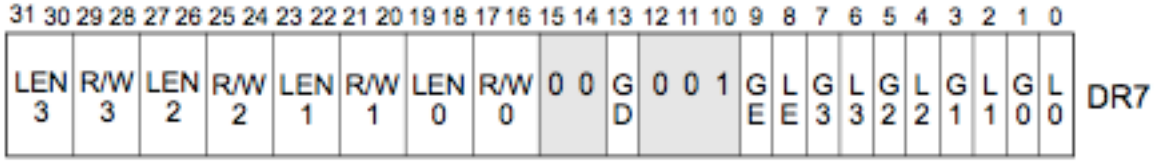
Figure 18-1. Debug Registers

DR7 - Debug Control Register



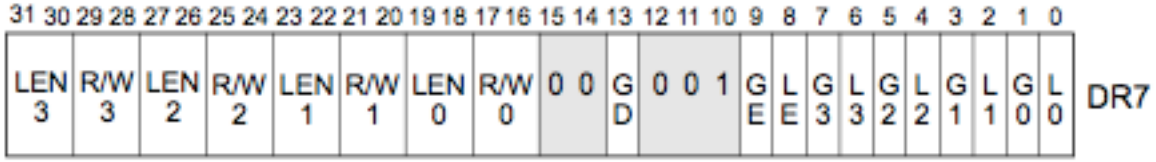
- L0-3 (local breakpoint enable) flags - Enables the DR0-3 breakpoint. But these flags are cleared on task switches to ensure that they do not fire when dealing with a different task.
- G0-3 (global breakpoint enable) flags - Enables the DR0-3 breakpoint. Does not get cleared on task switch, which is what makes it global obviously.
- LE & GE (local and global exact breakpoint enable) flags - If set to 1, enables detection of the exact **instruction that caused** a data breakpoint. Not supported on P6 microarchitecture and later, but if you need maximum compatibility for exact breakpoint detection you're recommended to set both to 1.

DR7 - Debug Control Register (2)



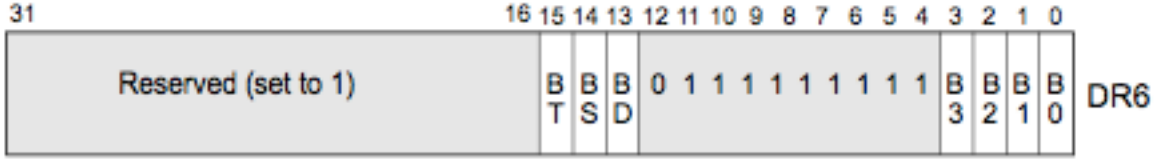
- GD (General Detect) flag - If set to 1, causes a debug exception prior to MOV instructions which access the debug registers. The flag is cleared when the actual exception occurs though, so that the handler can access the debug register as needed.
- The R/W0-3 are interpreted as follows:
  - 00 = Break on instruction execution only.
  - 01 = Break on data writes only.
  - If(CR4.DE == 1) then 10 = Break on I/O reads or writes.
  - If(CR4.DE == 0) then 10 = Undefined.
  - 11 = Break on data reads or writes but not instruction fetches.

DR7 - Debug Control Register (3)



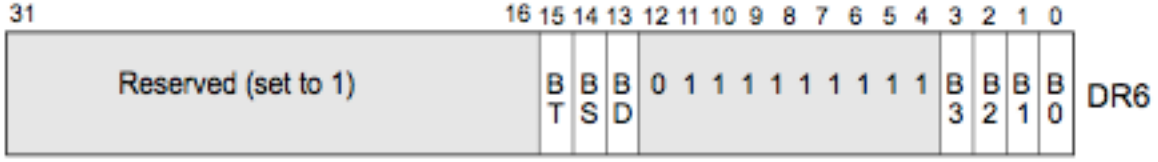
- LEN0-4 bits specify what size the address stored in the DR0-3 registers should be treated as.
  - 00 = 1-byte
  - 01 = 2-bytes
  - 10 = Undefined (or 8 bytes, see note below)
  - 11 = 4-bytes.
- While you might set a 1 byte size for an address pointing at the first byte of an instruction, on a break-on-execute, you might want to set a 4 byte breakpoint on writes to a memory location you know to be a DWORD.
- "For Pentium 4 and Intel Xeon processors with a CPUID signature corresponding to family 15 (model 3, 4, and 6), break point conditions permit specifying 8-byte length on data read/write with an of encoding 10B in the LENx field."

DR6 - Debug Status Register



- **B0-B3** (breakpoint condition detected) flags - When the B{0,1,2,3} bit is set, it means that the {0th,1st, 2nd,3rd} condition specified in DR7 has been satisfied. The bits are set even if the DR7 says that condition is currently disabled. I.e. software needs to crosscheck these bits against whether it currently cares.
- **BD** (debug register access detected) flag - Indicates that the next instruction will try to access the debug registers. This flag only enabled if GD (general detect) flag in DR7 is set. Thus this signals if someone else was trying to access the debug registers. NO! MINE!

DR6 - Debug Status Register (2)



- BS (single step) flag - If set, the debug exception was triggered by single-step execution mode (talked about later).
- BT (task switch) flag - Related to TSS so we don't care
- "Certain debug exceptions may clear bits 0-3. The remaining contents of the DR6 register are never cleared by the processor. To avoid confusion in identifying debug exceptions, debug handlers should clear the register before returning to the interrupted task." Seems like an important point if you're making a debugger :)

So what actually happens when a hardware breakpoint fires?

- It fires IDT[1], a Debug Exception
- When it is an execute breakpoint or general detect (someone trying to access debug regs) it's a fault.
- For other cases It's A Trap!

  - That means if it was a break on write, the data is overwritten before the exception is generated. A handler which wants to show the before and after is responsible for keeping a copy of the before value.
- Instruction breakpoints are actually detected before the instruction executes. Therefore if the handler doesn't remove the breakpoint, and it just returned, the same exception would be raised over and over.
- This is where the Resume Flag (RF) comes into play

Resume Flag (RF in EFLAGS)

- When the RF is set, the processor ignores instruction breakpoints.
- To set the flag, a debug interrupt handler would manipulate the EFLAGS stored on the stack and then use IRETD (POPF, POPFD, and IRET do not transfer RF from the stack into EFLAGS)
- “The processor then ignores instruction breakpoints for the duration of the next instruction.” “The processor then automatically clears this flag after the instruction returned to has been successfully executed.”

Trap Flag (TF in EFLAGS)

- Only being able to invoke the debug exception handler on 4 addresses is somewhat limiting.
- When TF is 1, it causes a debug exception after every instruction. This is called “single-step” mode.
  - Useful for capabilities such as “step out” which just steps until it steps through a RET
- Remember that we said that if the debug exception is in response to single stepping, it sets the BS flag in DR6.
- The processor clears the TF flag before calling the exception handler, so if it wants to keep single-stepping it needs to set it again before returning.
- Also, the INT and INTO instructions clear TF. So a single stepping debugger handler should compensate accordingly.

WinDbg Hardware Breakpoints

- Hardware Breakpoint = ba rather than bp. Stands for break on access, where access can be read/write/ execute or port IO. Below is a simplified form of the command (see help page for full form)
- **ba** *Access Size Address*
  - *Access*
    - r = read/write
    - w = write
    - e = execute
    - i = I/O port (talked about later)
  - *Size.* Width of data over which you want the breakpoint to have effect. Must be 1 for access = e, but can be 1, 2, 4, or 8 for other types
  - *Address.* Where you want the breakpoint to be targeted.

HW Breakpoint Examples

- ba e 1 0x80541ac0
  - Break on execute of address 0x80541ac0 (windbg specifies size must be 1 on break on execute)
- ba w 4 0x80541ac0
  - Break if anyone reads from the 4 bytes specified by 0x80541ac0 to 0x80541ac4
- ba r 2 0x80541ac0
  - Break if anyone reads or writes from byte 0x80541ac0 or 0x80541ac1(remember, the debug registers don't have a way to specify only read)

Lab: Watching the debugger debug

- TryToRunTryToHide.c take 2
- A discourse on the efficacy of using a hardware breakpoint on the INT 3 handler, rather than a software breakpoint ;)

Malware Use of Debug Regs

- “A packer such as tElock makes use of the debug registers to prevent reverse-engineers from using them. “
  - http://www.securityfocus.com/infocus/1893
- The Art of Unpacking (lists various anti-debug tricks including reading the debug registers)
  - https://www.blackhat.com/presentations/bh-usa-07/Yason/Whitepaper/bh-usa-07-yason-WP.pdf
- But I thought you had to be running in ring 0 to manipulate debug registers? Well, yes, but windows saves thread information (including the contents of the debug register) into a CONTEXT structure, and provides userspace a way to get and set thread context, thus letting malware detect .

The New Stuff

(Which I don't yet know enough about to teach. Read 3b, Sect. 18.4 on your own)

- "P6 family processors introduced the ability to set breakpoints on taken branches, interrupts, and exceptions, and to single-step from one branch to the next. This capability has been modified and extended in the Pentium 4, Intel Xeon, Pentium M, Intel CoreTM Solo, Intel CoreTM Duo, Intel CoreTM2 Duo, Intel CoreTM i7 and Intel AtomTM processors to allow logging of branch trace messages in a branch trace store (BTS) buffer in memory."
- The branch trace is basically a stack which holds pairs of addresses which represent the source of the branch and the destination.
- Pedram Amini talked about the performance improvement vs. single step mode here:
  - http://www.openrce.org/blog/view/535/Branch_Tracing_with_Intel_MSR_Registers

Port I/O

- “In addition to transferring data to and from external memory, IA-32 processors can also transfer data to and from input/output ports (I/O ports).”
- “I/O ports are created in system hardware by circuity that decodes the control, data, and address pins on the processor. These I/O ports are then configured to communicate with peripheral devices.”
- “An I/O port can be an input port, an output port, or a bidirectional port.”

Ports

- There are 2^16 8bit IO ports, numbered 0-0xFFFF.
- Can combine 2 or 4 consecutive ports to achieve a 16 or 32 bit port.
- “32-bit ports should be aligned to addresses that are multiples of four (0, 4, 8, ...).”

Accessing the Ports

- You cannot use the IN/OUT instructions to access the ports unless you have sufficient privileges.
- There is a 2 bit IOPL (I/O Privilege Level) field in EFLAGS. You can only perform IO if CPL <= IOPL.
- Also FYI, the STI/CLI instructions we saw before also are only allowed if CPL <= IOPL.
- Most OSes set IOPL to 0
- But think back to privilege rings. An appropriately modified OS (say, a paravirtualized one), could allow IOPL to be something like 2.

IN - Input from Port

IN—Input from Port

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
E4 *ib*	IN AL, *imm8*	Valid	Valid	Input byte from *imm8* I/O port address into AL.
E5 *ib*	IN AX, *imm8*	Valid	Valid	Input word from *imm8* I/O port address into AX.
E5 *ib*	IN EAX, *imm8*	Valid	Valid	Input dword from *imm8* I/O port address into EAX.
EC	IN AL,DX	Valid	Valid	Input byte from I/O port in DX into AL.
ED	IN AX,DX	Valid	Valid	Input word from I/O port in DX into AX.
ED	IN EAX,DX	Valid	Valid	Input doubleword from I/O port in DX into EAX.

- Note it's DX, not DL. That means the DX form can specify all 2^16 ports, but the IMM8 form can only specify 2^8 ports.
- "When accessing a 16- and 32-bit I/O port, the operand-size attribute determines the port size." (Because as usual there's an overloaded opcode for 16/32 bit form)
  - Remember if you're in a 16 bit segment it's 16 bit, if you're in a 32 bit segment it's 32 bit. But you can override it with an operand size instruction prefix which is talked about later.

OUT - Output to Port

OUT—Output to Port

Opcode*	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
E6 *ib*	OUT *imm8*, AL	Valid	Valid	Output byte in AL to I/O port address *imm8*.
E7 *ib*	OUT *imm8*, AX	Valid	Valid	Output word in AX to I/O port address *imm8*.
E7 *ib*	OUT *imm8*, EAX	Valid	Valid	Output doubleword in EAX to I/O port address *imm8*.
EE	OUT DX, AL	Valid	Valid	Output byte in AL to I/O port address in DX.
EF	OUT DX, AX	Valid	Valid	Output word in AX to I/O port address in DX.
EF	OUT DX, EAX	Valid	Valid	Output doubleword in EAX to I/O port address in DX.

- Basically the same caveat as IN

Lab: ParlorTrick.c

- Accessing the vmware “backdoor” IO port.
- For more tricks, see VMBack - http://chitchat.at.infoseek.co.jp/vmware/backdoor.html

The 8042 keyboard controller

- Lots of good info here: http://www.computer-engineering.org/ps2keyboard/
- The original IBM PC used a chip by the name of 8042 as the keyboard controller. PS/2 keyboards use this chip, and luckily for us, so do VMWare virtual machines by default (since PS/2 is way simpler than USB)
- The 8042 has a status/command register mapped to IO port 0x60 and a data register mapped to IO port 0x64

Lab: basic_hardware.c Spooky action at a distance

- Code taken as-is from http://www.rootkit.com/vault/hoglund/basic_hardware.zip
- Starts a timer in the kernel which calls a function every 300ms. That function talks to the keyboard controller using port IO, and sets a new value for the LED indicator lights for Num/Caps/Scroll lock

Lab: bhwin_keysniff.c
Low level keystroke logging

- Code taken from http://www.rootkit.com/vault/hoglund/basic_keystroke.zip with changes and hackery to get it to work as noted inline
- Every time a key is pressed/released, it triggers an interrupt. In our case it is INT[0x93] which is usually handled by i8042prt.sys's I8042KeyboardInterruptService() function.
- Keys are represented as "scancodes" indicating position on the keyboard, not ASCII values.
- This code hooks the keyboard interrupt(i.e. puts itself into the IDT descriptor) and then reads and stores the incoming scancode, puts it back into the buffer, and then calls the original handler.

HW Breakpoint Examples

- WinDbg commands to set a breakpoint on port IO
- ba i 1 0x60
  - Break on 1 byte access to port address 0x60
  - The value read in will be in al
- ba i 4 0x60
  - Break on 1-4 byte access to port address 0x60
  - The value read in will be in al, ax, or eax depending on the size used in the IN instruction

Debug Reg Keyboard Sniffer

- We just saw from the bhwin_keysniff.c source, that you speak to port 0x60 to talk to the 8042 keyboard controller.
- We saw earlier that the debug registers have an option to break on port IO access.
- Combine, and you get a keyboard sniffer which hooks the debug breakpoint handler rather than keyboard handler (but still has to hook the IDT to catch the debug register interrupts, INT 1).
- It is admittedly fragile. If someone overwrites the debug reg entry, purposely or accidentally, it is blinded. But it could keep polling to check if it has been removed, and replace itself. Though sooner or later an analyst is going to wonder why his hardware breakpoints keep getting missed and overwritten.
- PoC: http://www.rootkit.com/vault/chpie/0x60_hook.zip

Tap Into Your Hidden Potential and Disassemble Binary Using Only THE POWER OF YOUR MIND!!!

- Foretell the future (values of registers)!
- Impress friends and cow orkers!
- Be the life of the party!
- Burn excess calories!
- Live in infamy!
- Call Now! Just 9 easy payments of 9.99!

(Dramatic) Intel Instruction Format

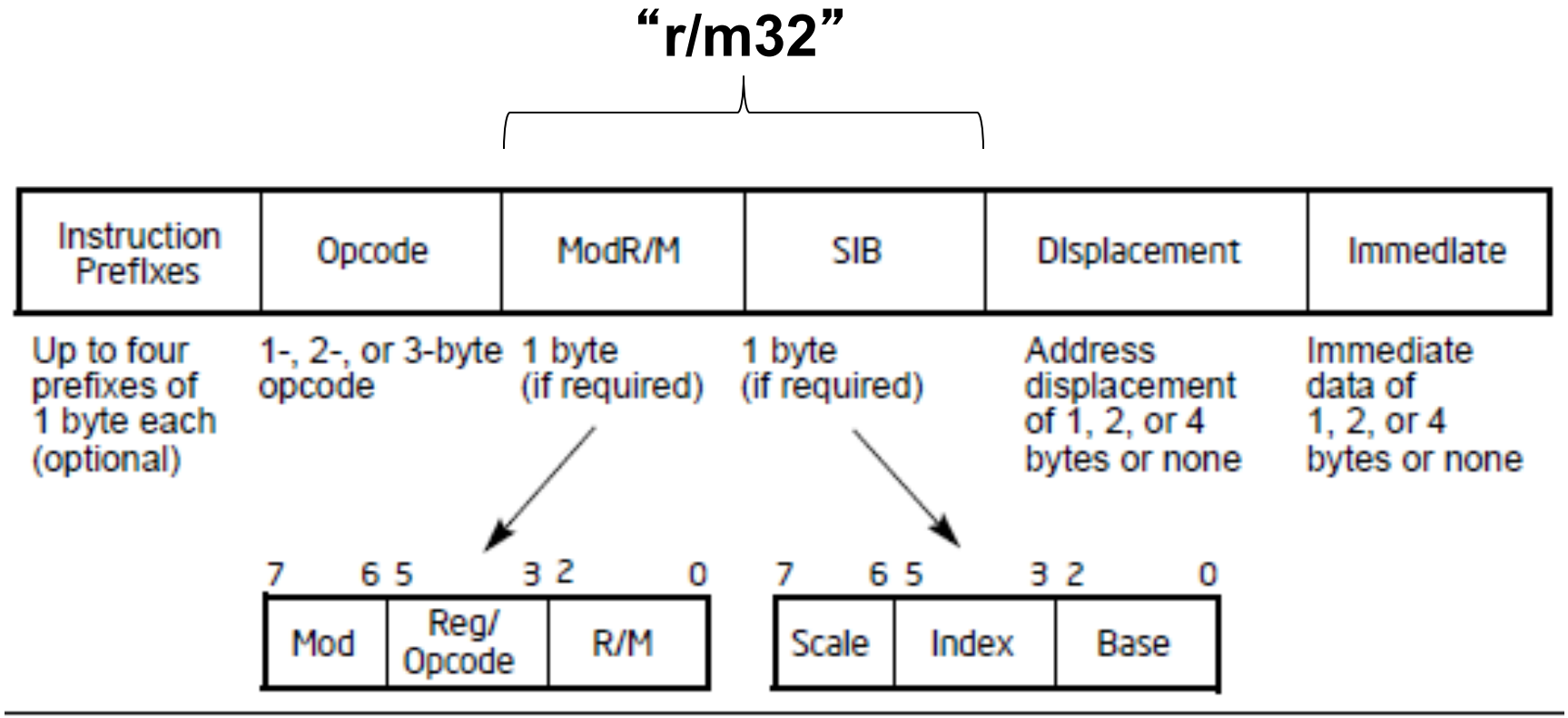


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

Explaination of “r/m32s”

- What I called an “r/m32” is actually the combination of the “ModR/M” and “SIB” bytes from the previous slide
- I had previously promised to get into why AND has a /4 in its opcode column.
- Vol. 2a page 3-2 section 3.1.1.1’s explanation of the opcode column says:
- “**/digit** – A digit between 0 and 7 indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.”
- “**/r — Indicates that the ModR/M byte of the instruction contains a register**”
- operand and an r/m operand.
- And here’s what 3.1.1.2 says about the instruction column:
- OK, what’s a “ModR/M” byte?

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

The src/dst register part

The memory part

NOTES:

1. The [--][--] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is sign-extended and added to the index.

The base register

Table 2-3. 32-Bit Addressing Forms with the SIB Byte

r32 (In decimal) Base = (In binary) Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	**SS**	**Index**	**Value of SIB Byte (in Hexadecimal)**							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	89	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

This is misleading. Looking at the chart you would think you can't access [ESP], but you can (if your instruction either uses /r (or /5? Any such instruction?)

Examples of testing

- (You have to put each _emit on its own line, I just wrote them this way for conciseness)
- __asm{_emit 0x80; _emit 0x20; _emit 0xFF;};
- 80 20 FF and byte ptr [eax],0FFh
- __asm{_emit 0x80; _emit 0x60; _emit 0x20; _emit 0xFF;};
- 80 60 20 FF and byte ptr [eax+20h],0FFh
- __asm{_emit 0x80; _emit **0x00**; _emit 0xFF;};
- 80 00 FF add byte ptr [eax],0FFh
- Wait, add? But I'm still using 0x80 at the front?!

Overloaded Opcodes

- We've already seen cases where opcodes are overloaded for different data size, but some like 0x80 are also overloaded for things like arithmetic/ logical operations

80 /4 *ib*	AND *r/m8, imm8*	Valid	Valid	*r/m8* AND *imm8*.
80 /0 *ib*	ADD *r/m8, imm8*	Valid	Valid	Add *imm8* to *r/m8*.

- Back to the tables!

More than one way to skin an instruction

- If you read “skin” to mean “remove the skin of” rather than “put a skin onto”, then you are sick and should seek help :P
- ModRM = 0x0 = [EAX]
- ModRM = 0x8 = [EAX] - To the tables!
- Note that you only have the option of using either if your opcode is specified with a /r (like MOV) rather than a /digit (like AND/ADD)
- ModRM = 0x4 = Add SIB, SIB = 0x0 = [EAX] therefore 0x4 0x0 = [EAX]
- Yet another reason trying to use binary signatures sucks in x86

Instruction Prefixes

- There are 4 groups of prefixes specified
  - Group 1: LOCK/REP/REPNE
  - Group 2: Segment Override, Branch Hints (Seemingly unrelated, but put together because the prefix bytes are overloaded)
  - Group 3: Operand-size override
  - Group 4: Address-size override

Vol 2a, Sect 2.1.1

Lock prefix

- Lock prefix = 0xF0
- Locks the memory bus, preventing anyone else from changing memory until the instruction is done
- lock xchg eax, [ebx]
- Used for creating the simplest mutual exclusion primitive – mutex
- Cannot be applied to all instructions (see the specific instruction's details)

REP prefixes

Table 4-3. Repeat Prefixes

Repeat Prefix	Termination Condition 1*	Termination Condition 2
REP	RCX or (E)CX = 0	None
REPE/REPZ	RCX or (E)CX = 0	ZF = 0
REPNE/REPNZ	RCX or (E)CX = 0	ZF = 1

- REP/REPE/REPZ = 0xF3
- REPNE/REPNZ = 0xF2
- Both apply only to string (movs, stos, etc-ending-in-s) and in/out instructions
- REP
- 0xA5 = movsd es:[edi], ds:[esi]
- 0xF3 0xA5 = rep movsd es:[edi], ds:[esi]

Segment Override Prefixes

- “0x2E—CS segment override (use with any branch instruction is reserved)
- 0x36—SS segment override prefix (use with any branch instruction is reserved)
- 0x3E—DS segment override prefix (use with any branch instruction is reserved)
- 0x26—ES segment override prefix (use with any branch instruction is reserved)
- 0x64—FS segment override prefix (use with any branch instruction is reserved)
- 0x65—GS segment override prefix (use with any branch instruction is reserved)”
- Note, they’re only reserved with conditional branches
- FF 24 24 jmp dword ptr [esp]
- 65 FF 24 24 jmp dword ptr gs:[esp]
- 8B 04 24 mov eax,dword ptr [esp]
- 26 8B 04 24 mov eax,dword ptr es:[esp]

Branch Hints

- Tell the processor's branch prediction unit to assume that a branch will be taken one way or the other.
- Implicitly you would only do this when you knew that your code was structured such that it ran contrary to the default rules for branch prediction.
- Since good compilers try to generate code with the branch prediction defaults in mind, one would only expect to see this with handcoded asm.
- 0x2E – Branch not taken
- 0x3E – Branch taken
- These can only be used with conditional jumps (Jcc)
- Note that they are also valid segment override prefixes (and since there are more segment override prefixes than 0x2E/0x3E, I'm not sure why it says the other things should not be used with branch instructions)

Operand and Address Size Override Prefixes

- 0x66 – Operand Size Override
- 0x67 – Address Size Override
- In both cases, if something was in a 32 bit segment, it would be using 32 bit size, but adding these before instructions forces operands or addresses to be treated as 16 bits. (And if things were in 16 bit segments these prefixes would force things to 32 bits)

Lab: InstructionPrefixes.c

- More looking at and playing with the various prefixes

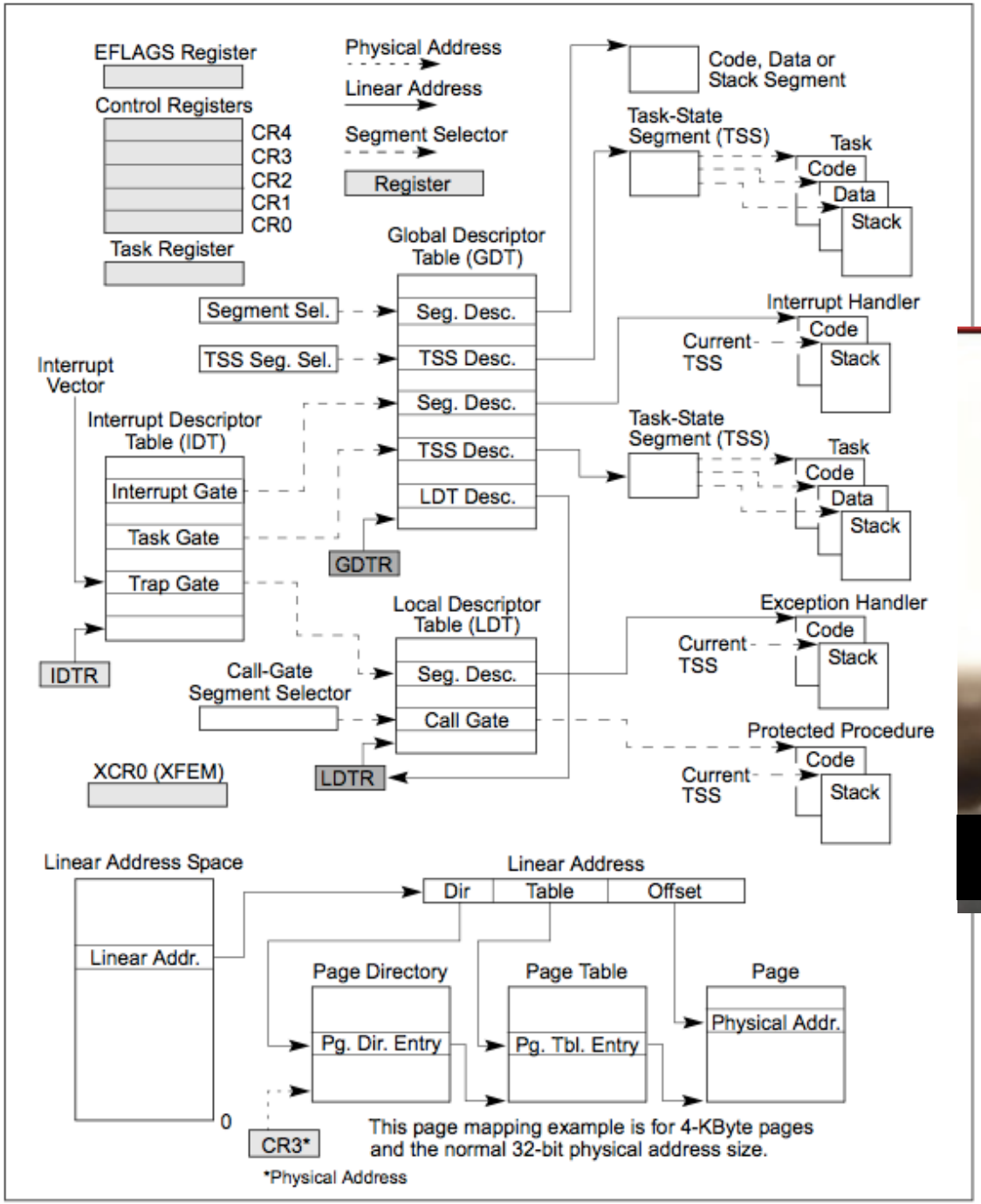
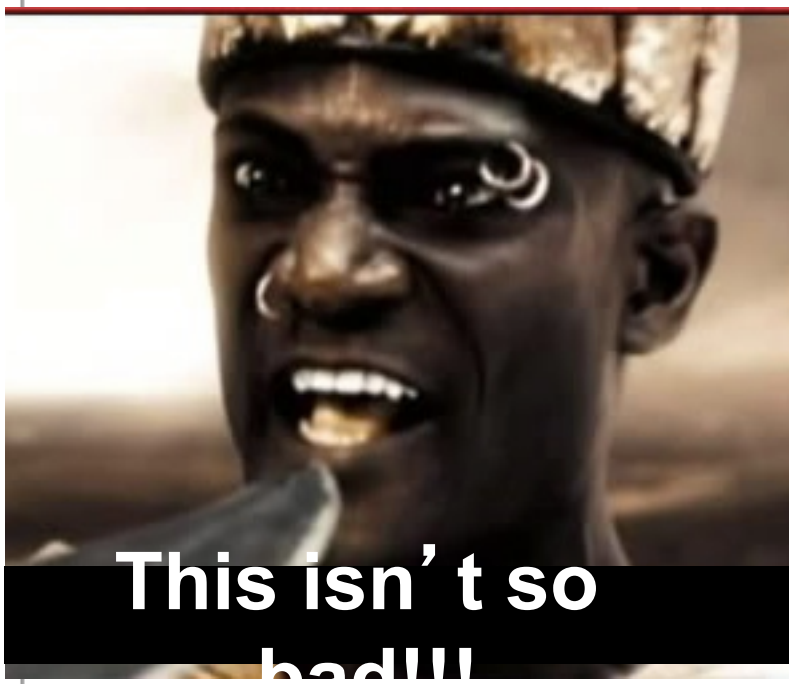


Figure 2-1. IA-32 System-Level Registers and Data Structures

That's what you learned! :D



The Big Picture

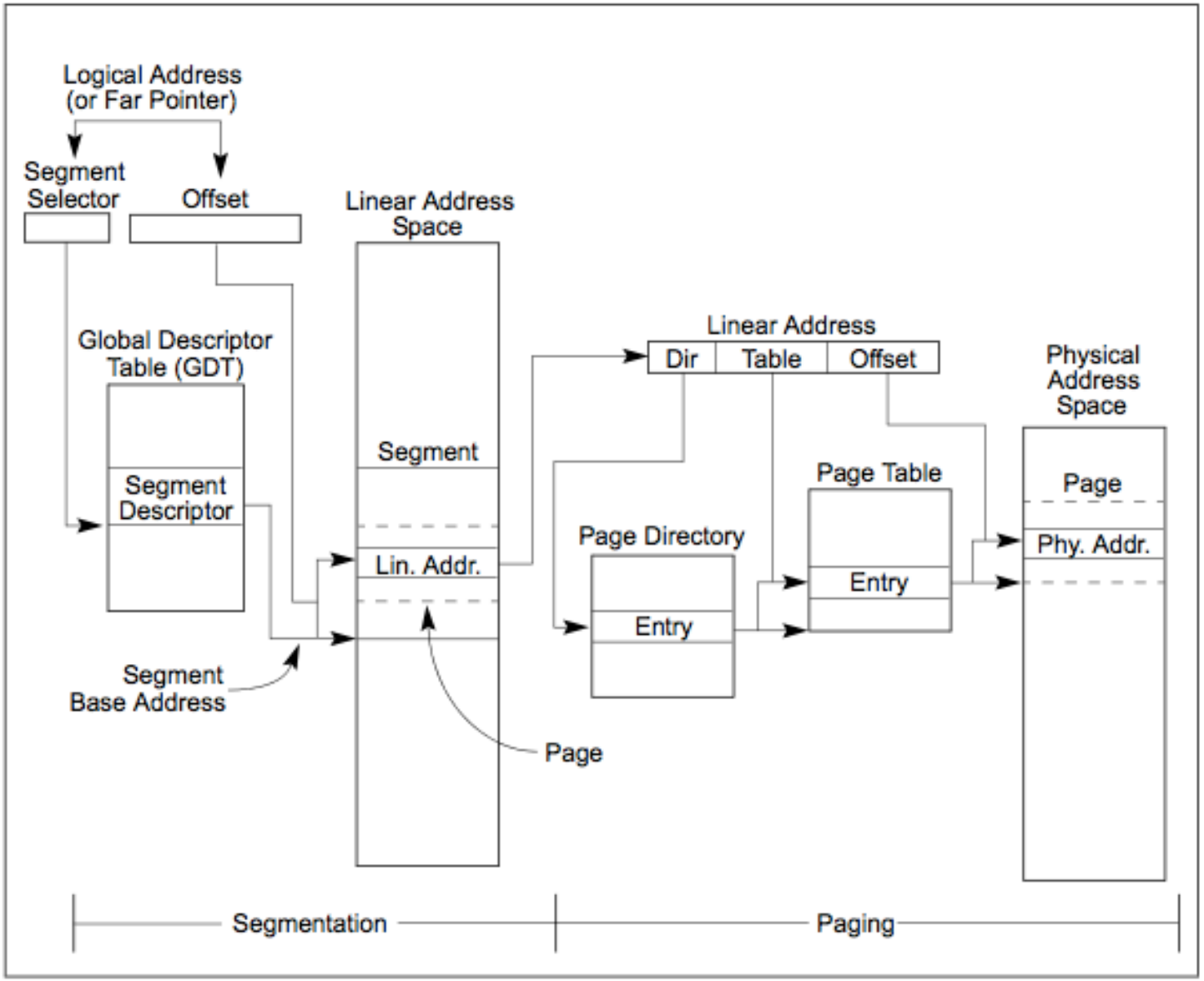


Figure 3-1. Segmentation and Paging

Virtual Memory in Four Fruity Flavors

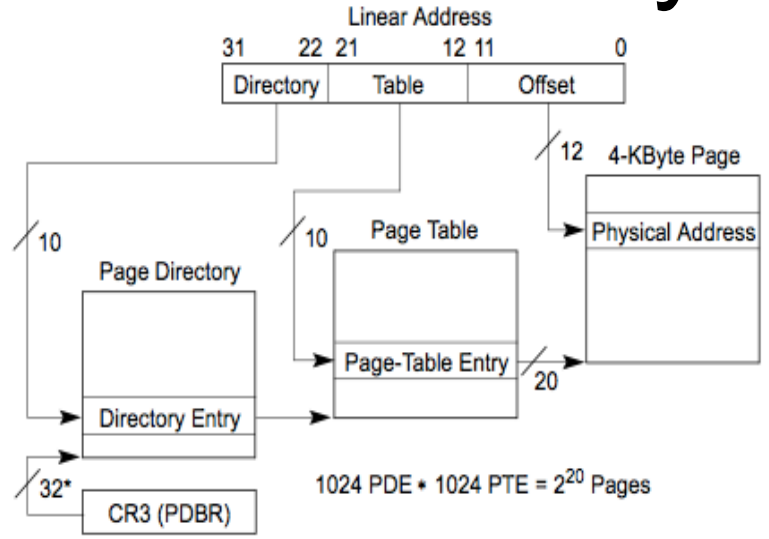


Figure 3-12. Linear Address Translation (4-KByte Pages)

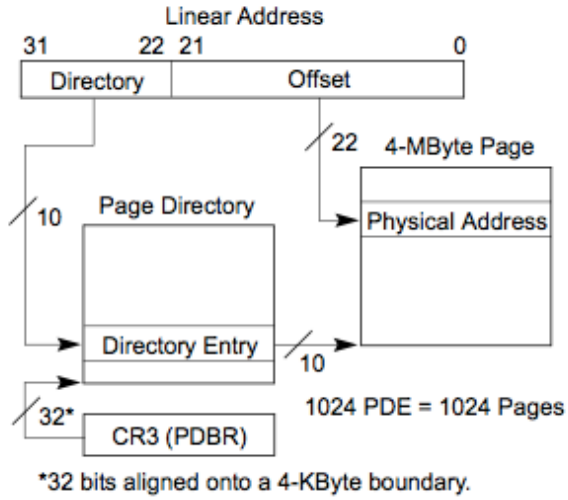


Figure 3-13. Linear Address Translation (4-MByte Pages)

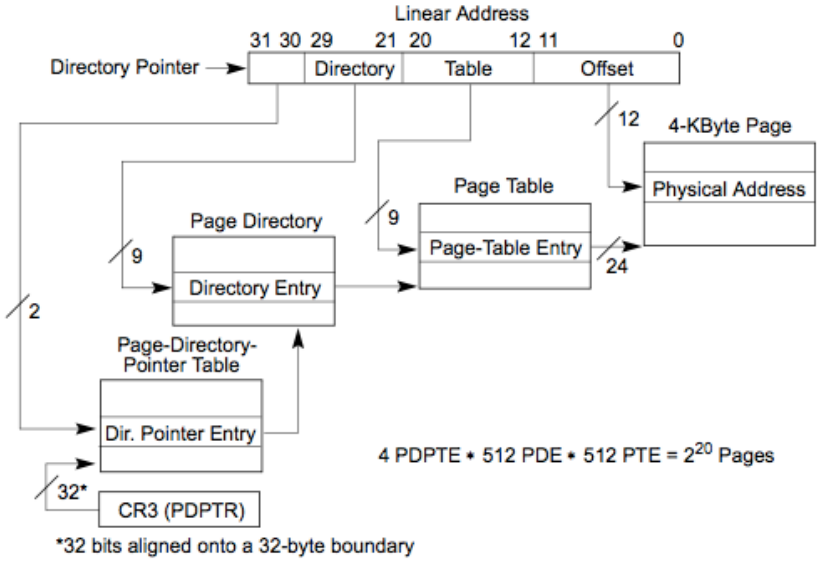


Figure 3-18. Linear Address Translation With PAE Enabled (4-KByte Pages)

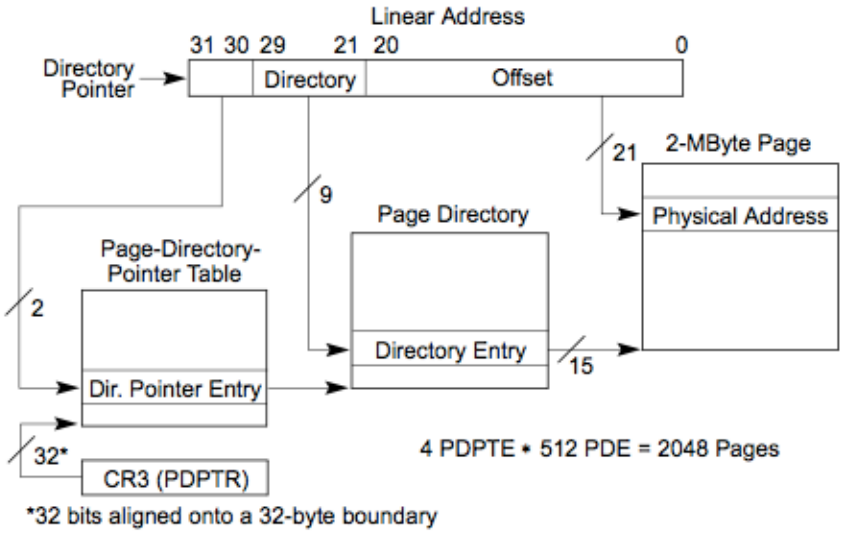


Figure 3-19. Linear Address Translation With PAE Enabled (2-MByte Pages)

Friends we've met along the way

- CPUID – Identify CPU features
- PUSHFD/POPFD – Push/Pop EFLAGS
- SGDT/LGDT – Store/Load GDTR
- SLDT/LLDT – Store/Load LDTR
- RDTSC – Read TimeStamp Counter
- MOV CR, r32 – (CR = Control Register)
- MOV r32, CR – (CR = Control Register)
- INVLPG – Invalidate page entry in TLB
- SIDT/LIDT – Load/Store IDTR
- STI/CLI – Set/Clear Interrupt Flag(IF)

More Friendly Friends!

- STR/LTR – Load/Store Task Register (TR)
- UD2 – Undefined Instruction Interrupt
- INTO – Overflow Interrupt
- INT n – Software Interrupt, invoke handler n
- 0xCC form of INT 3 – Breakpoint Interrupt
- IRET – Return from interrupt
- MOV DB, r32 (DB = Debug Register)
- MOV r32, DB (DB = Debug Register)
- IN/OUT – Input/Output Port IO

- LOCK prefix (if we had time for prefixes)

Closing thoughts…

- I tried to present most things in their Intel, mostly-OS-agnostic form, just using Windows as the example system due to robustness of the tools (and cause I've been forced to use it lately.)
- You're now more qualified to start digging into how *any* OS running on the x86 platform works. Just look for the relevant "* Internals" book, pdfs, presentations, etc.
- You're actually quite well suited to start exploring other architectures as well, having been exposed to the deep internals of x86 you can see the similarities and differences in other architectures and reason about where a given component is more or less trying to achieve the same functionality as what you already know, etc.

Closing thoughts…

- You're also more qualified to start digging into how virtualization works
- OS is to Process as Hypervisor is to ?
- OS plays games among processes to share CPU time and memory. Hypervisor plays games among OSes to share CPU time and memory (and limited number of system control registers and structures).
- Ask yourself the question "If I wanted to trick multiple OSes into thinking they were accessing the hardware as normal, when in fact only I do, what would I do with control registers? IO? Page tables?" Then you start to see why virtualization is eminently detectable on x86.
- Then when you think about it in terms of tricking the OS, think about it in terms of *maliciously* tricking the OS, and you've got yourself a hypervisor-based rootkit :)

Closing thoughts…

- There are tool-users and tool-builders/ understanders. When you learn reverse engineering by necessity, you are a tool user. When you look at the tool builders or the rock stars of RE, they are the understanders.
- You now understand a lot more about how things work than many reverse engineers. (Note to self, see note)
- “How does a debugger work” is a great interview question to ask to and drill down on with someone with lots of experience reverse engineering, to help understand if they’re a user or an understander :)

Closing thoughts…

- In general, kernel-mode rootkit researchers come from a background of knowing a lot of the OS and architecture level details
- Very difficult to combat adversaries which know more than you about the system you're defending.
- You're all now more qualified to work on my research! :D Hidden agenda #427 revealed!

To be continued?

- Once you successfully digest this material you will know nearly as much stuff about x86 as I do.
- If there's an Advanced x86 class, a big reason for it will be for me to nail down stuff I've been meaning to learn better…getting paid to share the knowledge with you would just be a side benefit :P
- System Management Mode (hot topic in rootkits…you really want me to teach this, because if I do, I will try to pull together the existing SMM attacks into sweet sweet VM escape PoCs)
- Hardware Virtualization (I'm less interested in virtualization at the vendor level and more interested in how it works)
- Intel 64 bit (OS X >= 10.6 now native 64 bit)
- Advanced Programmable Interrupt Controller (APIC) – Even deeper into how interrupts work. Useful for sending inter-processor interrupts on multi-CPU/core systems
- Intel SpeedStep – manually controlling CPU throttling. A MS guy on a Windows developer list said if you're not the OS you shouldn't be messing with it, which just makes me want to do it more ;)
- BIOS maybe, but probably not

Other stuff

- The Life of Binaries class
- RE class(es?)
- Vulnerabilities & Exploits class
- Understanding Malware: Rootkits class?
- Understanding Malware: Botnets class?
- Advocacy!
- Fill out the feedback!