x86-64 encoding / viruses

last time

x86-64 registers

- overlapping registers
- write 32-bit → overwrite top of 64-bit
- pseudo-register `%rip`: address of end of instruction
- %xmm/%ymm registers; x87 floating point stack

AT&T syntax:

- destination last, $ for constants (otherwise memory)
- $B, I, S \rightarrow B + I \cdot S$

Linux x86-64 calling convention

lea (load effective address) — copy address to result

segmentation: `%fs:0x10`: access 0x10 + base address for FS

- set by OS, used for thread-local storage

quiz reliability aside

appears the quiz site sometimes didn't give feedback when submitting logged-out of NetBadge

> I don't understand how (200 GET response when there should be NetBadge prompt?)

problem should not recur (quiz submission w/o NetBadge)

let me know if this affected your quiz score

x86 instruction encoding

in 2110, 3330 you learned a "teaching" machine code

Y86 (3330) is very like what x86 should be

...but it isn't

why? history!

the 8086

1979 Intel processor

4 general purpose 16-bit registers: AX, BX, CX, DX

4 special 16-bit registers: SI, DI, BP, SP

8086 instruction encoding: simple

special cases: 1-byte instructions:

- anything with no arguments
- push ax, push bx, push cx, ...(dedicated opcodes)
- pop ax, ...

8086 instruction encoding: two-arg

1-byte opcode

sometimes ModRM byte:

- 2-bit "mod" and
- 3-bit register number (source or dest, depends on opcode) and
- 3-bit "r/m" (register or memory)

"mod" + "r/m" specify one of:

- `%reg` (mod = `11`)
- `(%bx/%bp, %si/%di)`
- `(%bx/%si/%di)`
- `offset(%bx/%bp/,%si/%di)` (8- or 16-byte offset)

non-intuitive table

16-bit ModRM table

Effective Address	Mod	R/M
[BX+SI]	00	000
[BX+DI]		001
[BP+SI]		010
[BP+DI]		011
[SI]		100
[DI]		101
disp16[2]		110
[BX]		111
[BX+SI]+disp8[3]	01	000
[BX+DI]+disp8		001
[BP+SI]+disp8		010
[BP+DI]+disp8		011
[SI]+disp8		100
[DI]+disp8		101
[BP]+disp8		110
[BX]+disp8		111
[BX+SI]+disp16	10	000
[BX+DI]+disp16		001
[BP+SI]+disp16		010
[BP+DI]+disp16		011
[SI]+disp16		100
[DI]+disp16		101
[BP]+disp16		110
[BX]+disp16		111
EAX/AX/AL/MM0/XMM0	11	000
ECX/CX/CL/MM1/XMM1		001
EDX/DX/DL/MM2/XMM2		010
EBX/BX/BL/MM3/XMM3		011
ESP/SP/AHMM4/XMM4		100
EBP/BP/CH/MM5/XMM5		101
ESI/SI/DH/MM6/XMM6		110
EDI/DI/BH/MM7/XMM7		111

16-bit ModRM table

Effective Address	Mod	R/M
[BX+SI]	00	000
[BX+DI]		001
[BP+SI]		010
[BP+DI]		011
[SI]		100
[DI]		101
disp16[2]		110
[BX]		111
[BX+SI]+disp8[3]	01	000
[BX+DI]+disp8		001
[BP+SI]+disp8		010
[BP+DI]+disp8		011
[SI]+disp8		100
[DI]+disp8		101
[BP]+disp8		110
[BX]+disp8		111
[BX+SI]+disp16	10	000
[BX+DI]+disp16		001
[BP+SI]+disp16		010
[BP+DI]+disp16		011
[SI]+disp16		100
[DI]+disp16		101
[BP]+disp16		110
[BX]+disp16		111
EAX/AX/AL/MM0/XMM0	11	000
ECX/CX/CL/MM1/XMM1		001
EDX/DX/DL/MM2/XMM2		010
EBX/BX/BL/MM3/XMM3		011
ESP/SP/AHMM4/XMM4		100
EBP/BP/CH/MM5/XMM5		101
ESI/SI/DH/MM6/XMM6		110
EDI/DI/BH/MM7/XMM7		111

e.g. `add %bl, %cl`
(Intel syntax: `add CL, BL`)
Intel manual:
`02 /r`: `ADD r8 (dest), r/m8`
`/r` means ModRm byte with reg set to reg#

opcode = `0x02` ModRM byte =
`11` (mod) / `001` (reg: %cl) / `011` (r/m: %bl)
or `1100 1011`
final encoding: `02 cb`

8086 evolution

Intel 8086 — 1979, 16-bit registers

Intel (80)386 — 1986, 32-bit registers

AMD K8 — 2003, 64-bit registers

x86 modes

x86 has multiple modes

maintains compatiblity

e.g.: modern x86 processor can work like 8086
called "real mode"

different mode for 32-bit/64-bit

same basic encoding; some sizes change

32-bit ModRM table

r8(/r)
r16(/r)
r32(/r)
mm(/r)
xmm(/r)
(In decimal) /digit (Opcode)
(In binary) REG =

Effective Address	Mod	R/M
[EAX]	00	000
[ECX]		001
[EDX]		010
[EBX]		011
[--][--][1]		100
disp32[2]		101
[ESI]		110
[EDI]		111
[EAX]+disp8[3]	01	000
[ECX]+disp8		001
[EDX]+disp8		010
[EBX]+disp8		011
[--][--]+disp8		100
[EBP]+disp8		101
[ESI]+disp8		110
[EDI]+disp8		111
[EAX]+disp32	10	000
[ECX]+disp32		001
[EDX]+disp32		010
[EBX]+disp32		011
[--][--]+disp32		100
[EBP]+disp32		101
[ESI]+disp32		110
[EDI]+disp32		111
EAX/AX/AL/MM0/XMM0	11	000
ECX/CX/CL/MM/XMM1		001
EDX/DX/DL/MM2/XMM2		010
EBX/BX/BL/MM3/XMM3		011
ESP/SP/AH/MM4/XMM4		100
EBP/BP/CH/MM5/XMM5		101
ESI/SI/DH/MM6/XMM6		110
EDI/DI/BH/MM7/XMM7		111

32-bit ModRM table

r8(/r)
r16(/r)
r32(/r)
mm(/r)
xmm(/r)
(In decimal) /digit (Opcode)
(In binary) REG =

Effective Address	Mod	R/M
[EAX]	00	000
[ECX]		001
[EDX]		010
[EBX]		011
[--][--][1]		100
disp32[2]		101
[ESI]		110
[EDI]		111
[EAX]+disp8[3]	01	000
[ECX]+disp8		001
[EDX]+disp8		010
[EBX]+disp8		011
[--][--]+disp8		100
[EBP]+disp8		101
[ESI]+disp8		110
[EDI]+disp8		111
[EAX]+disp32	10	000
[ECX]+disp32		001
[EDX]+disp32		010
[EBX]+disp32		011
[--][--]+disp32		100
[EBP]+disp32		101
[ESI]+disp32		110
[EDI]+disp32		111
EAX/AX/AL/MM0/XMM0	11	000
ECX/CX/CL/MM/XMM1		001
EDX/DX/DL/MM2/XMM2		010
EBX/BX/BL/MM3/XMM3		011
ESP/SP/AH/MM4/XMM4		100
EBP/BP/CH/MM5/XMM5		101
ESI/SI/DH/MM6/XMM6		110
EDI/DI/BH/MM7/XMM7		111

general pattern for 32-bit x86 register numbering:
AX = 0, CX, DX, BX, SP, BP, SI, DI = 7

not all registers treated equally to make space for special types of addressing:
(base + index * scale, constant address)

32-bit addition: SIB bytes

8086 addressing modes made registers different

32-bit mode got rid of this (mostly)

problem: not enough spare bits in `ModRM` byte

solution: if "r/m" bits = `100` (4, normally ESP), extra "SIB" byte:

- 2 bit scale: `00` is 1, `01` is 2, `10` is 4, `11` is 8
- 3 bit index: index register number
- 3 bit base: base register number

```
(%baseReg,%indexReg,scale)
```

32-bit addition: SIB bytes

8086 addressing modes made registers different

32-bit mode got rid of this (mostly)

problem: not enough spare bits in `ModRM` byte

solution: if "r/m" bits = `100` (4, normally ESP), extra "SIB" byte:

- 2 bit scale: `00` is 1, `01` is 2, `10` is 4, `11` is 8
- 3 bit index: index register number
- 3 bit base: base register number

```
(%baseReg,%indexReg,scale)
```

32-bit addition: SIB bytes

8086 addressing modes made registers different

32-bit mode got rid of this (mostly)

problem: not enough spare bits in `ModRM` byte

solution: if "r/m" bits = `100` (4, normally ESP), extra "SIB" byte:

- 2 bit scale: `00` is 1, `01` is 2, `10` is 4, `11` is 8
- 3 bit index: index register number
- 3 bit base: base register number

```
(%baseReg,%indexReg,scale)
```

32-bit addition: SIB bytes

8086 addressing modes made registers different

32-bit mode got rid of this (mostly)

problem: not enough spare bits in `ModRM` byte

solution: if "r/m" bits = `100` (4, normally ESP), extra "SIB" byte:

- 2 bit scale: `00` is 1, `01` is 2, `10` is 4, `11` is 8
- 3 bit index: index register number
- 3 bit base: base register number

```
(%baseReg,%indexReg,scale)
```

intel manual: SIB table

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	99	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

NOTES:

1. The [*] nomenclature means a disp32 with no base if the MOD is 00B. Otherwise, [*] means disp8 or disp32 + [EBP]. This provides the

basic 32-bit encoding

dashed: not always present

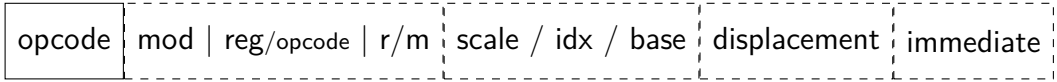
opcodes: 1-3 bytes

- some 5-bit opcodes, with 3-bit register field
- (alternate view: 8-bit opcode with fixed register)
- sometimes Reg part of ModRM used as add'tl part of opcode

displacement, immediate: 1, 2, or 4 bytes

- or, rarely, 8 bytes

exercise 1



r8(/r)
r16(/r)
r32(/r)
mm(/r)
xmm(/r)
(In decimal) /digit (Opcode)
(In binary) REG =

Effective Address	Mod	R/M
[EAX]	00	000
[ECX]		001
[EDX]		010
[EBX]		011
[--][--][1]		100
disp32[2]		101
[ESI]		110
[EDI]		111
[EAX]+disp8[3]	01	000
[ECX]+disp8		001
[EDX]+disp8		010
[EBX]+disp8		011
[--][--]+disp8		100
[EBP]+disp8		101
[ESI]+disp8		110
[EDI]+disp8		111
[EAX]+disp32	10	000
[ECX]+disp32		001
[EDX]+disp32		010
[EBX]+disp32		011
[--][--]+disp32		100
[EBP]+disp32		101
[ESI]+disp32		110
[EDI]+disp32		111
EAX/AX/AL/MM0/XMM0	11	000
ECX/CX/CL/MM/XMM1		001
EDX/DX/DL/MM2/XMM2		010
EBX/BX/BL/MM3/XMM3		011
ESP/SP/AH/MM4/XMM4		100
EBP/BP/CH/MM5/XMM5		101

BTS—Bit Test and Set

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F AB /r	BTS *r/m16, r16*	MR	Valid	Valid	Store selected bit in CF flag and set.
0F AB /r	BTS *r/m32, r32*	MR	Valid	Valid	Store selected bit in CF flag and set.
REX.W + 0F AB /r	BTS *r/m64, r64*	MR	Valid	N.E.	Store selected bit in CF flag and set.
0F BA /5 *ib*	BTS *r/m16, imm8*	MI	Valid	Valid	Store selected bit in CF flag and set.
0F BA /5 *ib*	BTS *r/m32, imm8*	MI	Valid	Valid	Store selected bit in CF flag and set.
REX.W + 0F BA /5 *ib*	BTS *r/m64, imm8*	MI	Valid	N.E.	Store selected bit in CF flag and set.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA

exercise: encode `btsl $7, 4(%rax)`
(Intel syntax: `BTS DWORD PTR[RAX+4], 7`)

what about 64-bit?

adds 8 more registers — more bits for reg #?

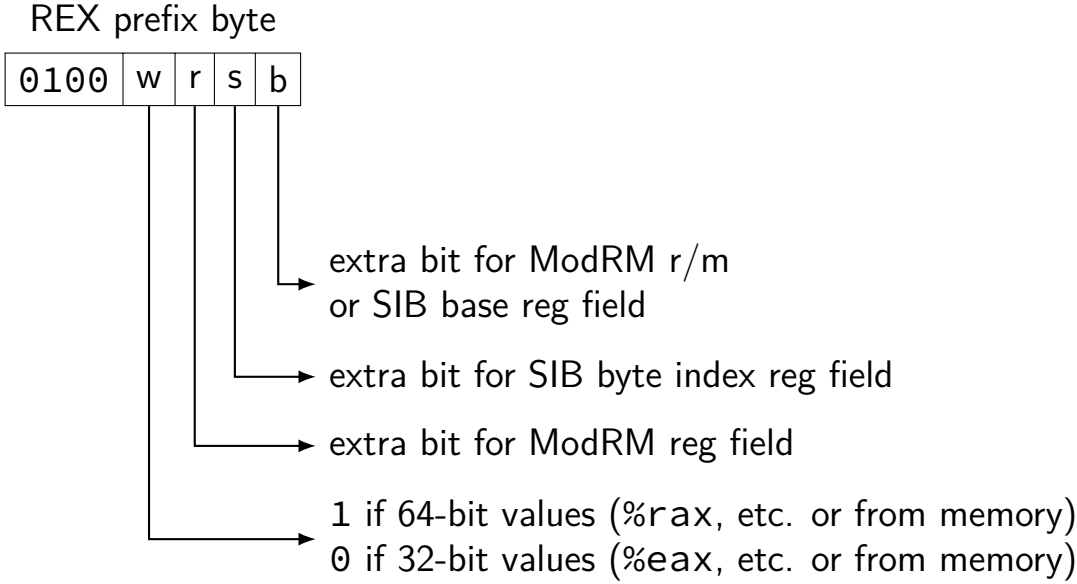
didn't change encoding for existing instructions, so...

instruction prefix "REX"

- 32-bit x86 already had many prefixes

also selects 64-bit version of instruction

REX prefix


REX prefix byte
0100
w
r
s
b
extra bit for ModRM r/m
or SIB base reg field
extra bit for SIB byte index reg field
extra bit for ModRM reg field
1 if 64-bit values (%rax, etc. or from memory)
0 if 32-bit values (%eax, etc. or from memory)

64-bit REX exercise (1)

`add %eax, %ecx` (Intel: `ADD ecx, eax`)

`01` (opcode) `c1` (MOD: 11 / REG: 000 (eax) / RM: 001 (ecx))

exercise 2a: `add %eax, %r10d` (Intel: `ADD r9d, eax`) = ???

REX prefix + `01` + MOD-REG-R/M byte

REX prefix:

`0100`
w (is 64-bit values?)
r (extra bit for Reg field)
s (extra bit for SIB index reg)
b (extra bit for R/M or SIB base field)

64-bit REX exercise (2)

`add %eax, %ecx` (Intel: `ADD ecx, eax`)

- `01` (opcode) `c1` (MOD: 11 / REG: 000 (eax) / RM: 001 (ecx))

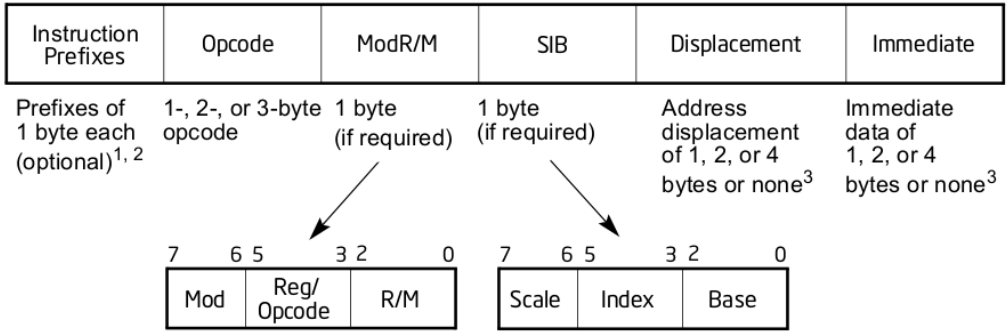
exercise 2b: `add %rax, %rcx` (Intel: `ADD rcx, rax`) = ???

- REX prefix + `01` + MOD-REG-R/M byte

REX prefix:

- `0100`
- w (is 64-bit values?)
- r (extra bit for Reg field)
- s (extra bit for SIB index reg)
- b (extra bit for R/M or SIB base field)

overall encoding



1. The REX prefix is optional, but if used must be immediately before the opcode; see Section 2.2.1, “REX Prefixes” for additional information.
2. For VEX encoding information, see Section 2.3, “Intel® Advanced Vector Extensions (Intel® AVX)”.
3. Some rare instructions can take an 8B immediate or 8B displacement.

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

instruction prefixes

REX (64-bit and/or extra register bits)

VEX (SSE/AVX instructions; other new instrs.)

operand/address-size change (64/32 to 16 or vice-versa)

LOCK — synchronization between processors

REPNE/REPNZ/REP/REPE/REPZ — turns instruction into loop

segment overrides

x86 encoding example (1)

`pushq %rax` encoded as `50`

5-bit opcode `01010` plus 3-bit register number `000`

`pushq %r13` encoded as `41 55`

`41`: REX prefix `0010` (constant), w:`0`, r:`0`, s:`0`, b:`1`

w = `0` because push is never 32-bit in 64-bit mode

`55`: 5-bit opcode `01010`; 3-bit reg # `101`

4-bit reg # `1101` = 13

x86 encoding example (2)

`addq 0x12345678(%rax,%rbx,2), %ecx`

`03`: opcode — add r/m32 to r/m32

`8c`: ModRM: mod = `10`; reg = `001`, r/m: `100`
reg = `001` = `%ecx` (table)
SIB byte + 32-bit displacement (table)

`58`: SIB: scale = `01`, index = `011`, base = `000`
index `011` = `%rbx`; base `000` = `%rax`;

`78 56 32 12`: 32-bit constant `0x12345678`

x86 encoding example (3)

`addq 0x12345678(%r10,%r11,2), %rax`

4b: REX prefix 0100+w:1, r:0, s:1, b:1

03: opcode — add r/m64 to r64 (with REX.w)

84: ModRM: mod = 10; reg = 000, r/m: 100
reg = 0000 = %rax
SIB byte + 32-bit displacement (table)

5a: SIB: scale = 01, index = 011, base = 010
with REX: index = 1011 (11), base = 1010 (10)

78 56 32 12: 32-bit constant 0x12345678

x86 encoding example (3)

`addq 0x12345678(%r10,%r11,2), %rax`

4b: REX prefix 0100+w:1, r:0, s:1, b:1

03: opcode — add r/m64 to r64 (with REX.w)

84: ModRM: mod = 10; reg = 000, r/m: 100
reg = 0000 = %rax
SIB byte + 32-bit displacement (table)

5a: SIB: scale = 01, index = 011, base = 010
with REX: index = 1011 (11), base = 1010 (10)

78 56 32 12: 32-bit constant 0x12345678

x86 encoding example (4)

`movq %fs:0x10,%r13`

64: FS segment override

48: REX: w: 1 (64-bit), r: 1, s: 0, b: 0

8b: opcode for MOV memory to register

2c: ModRM: mod = 00, reg = 101, r/m: 100
with REX: reg = 1101 [%r12]; r/m = 100 (SIB follows)

25: SIB: scale = 00; index = 0100; base = 0101
no register/no register in table

10 00 00 00: 4-byte constant 0x10

x86-64 impossibilities

illegal: `movq 0x12345678ab(%rax), %rax`

maximum 32-bit displacement

`movq 0x12345678ab, %rax` okay

extra `mov` opcode for `%rax` only

illegal: `movq $0x12345678ab, %rbx`

maximum 32-bit constant

`movq $0x12345678ab, %rax` okay

illegal: `pushl %eax`

no 32-bit push/pop in 64-bit mode

but 16-bit allowed (operand size prefix byte 66)

illegal: `movq (%rax, %rsp), %rax`

cannot use `%rsp` as index register

`movq (%rsp, %rax), %rax` okay

position dependence

two ways of encoding addresses in x86-64 assembly:

- address in little endian (typically 32-bits — limit on executable size)
- difference between address and %rip (next instruction address)

`movb label, %al`	`8a 04 25` *label's addr (32 bit)*
`jmp *label`	`ff 24 25` *label's addr (32 bit)*
`movb label(%rip), %al`	`8a 05` *%rip - label's addr (32 bit)*
`jmp label`	`e9` *%rip - label's addr (32 bit)*

how to know which? — check manual

position-independence: which to use?

suppose we're inserting "evil" code
at changing addresses in executable's memory

which of the following do we want absolute encoding for?
(i.e. which would absolute encoding be easier than relative)

A. address of a jump from evil code to function at fixed loc in executable
B. address of a jump in a loop in the "evil" code
C. address of a string in the "evil" code
D. address of a string in the executable

self-replicating malware

attacker's problem:
getting malware to run where they want

some options:

connect to machine and install it there

send to someone

convince someone else to send it to someone

self-replicating malware

attacker's problem:
getting malware to run where they want

some options:

connect to machine and install it there

send to someone

convince someone else to send it to someone

all automatable!

recall: kinds of malware

viruses — infects other programs

worms — own malicious programs

trojans — useful (looking) program that also is malicious

rootkit — silent control of system

viruses: hiding in files

get someone run your malware?

program they already want to run

to spread your malware?

program they already want to copy

trojan approach: create/modify new program

simpler: modify already used/shared program

viruses: hiding in files

get someone run your malware?

program they already want to run

to spread your malware?

program they already want to copy

trojan approach: create/modify new program

simpler: modify already used/shared program

viruses: infecting programs?

viruses infecting other programs seems less common

(but hard to get good statistics...)

but producing infected versions of legitimate software is common

e.g. fake download site

techniques for automated infection similar to manual infection

virus prevalence

viruses on commerically sold software media

from 1990 memo by Chris McDonald:

```
4.  MS-DOS INFECTIONS

SOFTWARE                REPORTING LOCATION      DATE     VIRAL INFECTION

a.  Unlock Masterkey    Kennedy Space Center    Oct 89   Vienna
b.  SARGON III          Iceland                 Sep 89   Cascade (1704)
c.  ASYST RTDEMO02.EXE  Fort Belvoir            Aug 89   Jerusalem-B
d.  Desktop Fractal     Various                 Jan 90   Jerusalem (1813)
        Design System
e.  Bureau of the       Government Printing     Jan 90   Jerusalem-B
    Census, Elec. County   Office/US Census Bureau
    & City Data Bk., 1988
f.  Northern Computers  Iceland                 Mar 90   Disk Killer
    (PC Manufacturer shipped infected systems.)
5.  MACINTOSH INFECTIONS

    SOFTWARE            REPORTING LOCATION      DATE     VIRAL INFECTION

a.  NoteWriter          Colgate College         Sep 89   Scores and nVIR
.......
```

early virus motivations

lots of (but not all) early virus software was “for fun”

not trying to monetize malware

(like is common today)

hard: Internet connections uncommon

Case Study: Vienna Virus

Vienna: virus from the 1980s

This version: published in Ralf Burger, "Computer Viruses: a high-tech disease" (1988)

targetted COM-format executables on DOS

Diversion: .COM files

.COM is a very simple executable format

no header, no segments, no sections

file contents loaded at fixed address `0x0100`

execution starts at `0x0100`

everything is read/write/execute (no virtual memory)

Vienna: infection

uninfected

```
0x0100:
    mov $0x4f28, %cx
    /* b9 28 4f */
0x0103:
    mov $0x9e4e, %si
    /* be 4e 9e */
    mov %si, %di
    push %ds
    /* more normal
       program
       code */
....
0x0700: /* end */
```

infected

```
0x0100: jmp 0x0700
0x0103: mov $0x9e4e, %si
...
0x0700:
    push %cx
    ... // %si ← 0x903
    mov $0x100, %di
    mov $3, %cx
    rep movsb
    ...
    mov $0x0100, %di
    push %di
    xor %di, %di
    ret
...
0x0903:
    .bytes 0xb9 0x28 0x4f
...
```

Vienna: “fixup”

```
0x0700:
    push %cx // initial value of %cx matters??
    mov $0x8fd, %si // %si ← beginning of data
    mov %si, %dx // save %si
        // movsb uses %si, so
        // can't use another register
    add $0xa, %si // offset of saved code in data
    mov $0x100, %di // target address
    mov $3, %cx // bytes changed
    /* copy %cx bytes from (%si) to (%di) */
    rep movsb
    ...
...
// saved copy of original application code
0x903: .byte 0xb9 .byte 0x28 .byte 0x4f
```

Vienna: “fixup”

```
0x0700:
    push %cx // initial value of %cx matters??
    mov $0x8fd, %si // %si ← beginning of data
    mov %si, %dx // save %si
        // movsb uses %si, so
        // can't use another register
    add $0xa, %si // offset of saved code in data
    mov $0x100, %di // target address
    mov $3, %cx // bytes changed
    /* copy %cx bytes from (%si) to (%di) */
    rep movsb
    ...
...
// saved copy of original application code
0x903: .byte 0xb9 .byte 0x28 .byte 0x4f
```

Vienna: “fixup”

```
0x0700:
    push %cx // initial value of %cx matters??
    mov $0x8fd, %si // %si ← beginning of data
    mov %si, %dx // save %si
        // movsb uses %si, so
        // can't use another register
    add $0xa, %si // offset of saved code in data
    mov $0x100, %di // target address
    mov $3, %cx // bytes changed
    /* copy %cx bytes from (%si) to (%di) */
    rep movsb
    ...
...
// saved copy of original application code
0x903: .byte 0xb9 .byte 0x28 .byte 0x4f
```

Vienna: return

```
0x08e7:
    pop %cx // restore initial value of %cx, %sp
    xor %ax, %ax // %ax ← 0
    xor %bx, %bx
    xor %dx, %dx
    xor %si, %si
    // push 0x0100
    mov $0x0100, %di
    push %di
    xor %di, %di // %di ← 0
    // pop 0x0100 from stack
    // jmp to 0x0100
    ret
```

question: why not just jmp 0x0100 ?

Vienna: infection outline

Vienna appends code to infected application

where does it read the code come from?

how is code adjusted for new location in the binary?

- what linker would do

how does it keep files from getting infinitely long?

Vienna: infection outline

Vienna appends code to infected application

where does it read the code come from?

how is code adjusted for new location in the binary?

- what linker would do

how does it keep files from getting infinitely long?

quines

exercise: write a C program that outputs its source code
(pseudo-code only okay)

possible in any (Turing-complete) programming language

called a “quine”

clever quine solution

```
#include <stdio.h>
char*x="int main(){
       printf(p,10,34,x,34,10,34,p,34,10,x,10);
       }";
char*p="#include <stdio.h>%c
    char*x=%c%s%c;%cchar*p=%c%s%c;
    %c%s%c";
int main(){
    printf(p,10,34,x,34,10,34,p,34,10,x,10);
}
```

some line wrapping for readability — shouldn't be in actual quine

clever quine solution

```
#include <stdio.h>
char*x="int main(){
        printf(p,10,34,x,34,10,34,p,34,10,x,10);
        }";
char*p="#include <stdio.h>%c
    char*x=%c%s
    %c%s%c";
int main(){
    printf(p,10
}
```

`printf` to fill template:
10 = newline; 34 = double-quote;
x, p = template/constant strings

some line wrapping for readability — shouldn't be in actual quine

clever quine solution

```
#include <stdio.h>
char*x="int main(){
       printf(p,10,34,x,34,10,34,p,34,10,x,10);
       }";
char*p="#include <st
    char*x=%c%s%c;%c
    %c%s%c";
int main(){
    printf(p,10,34,x,34,10,34,p,34,10,x,10);
}
```

template filled by printf

some line wrapping for readability — shouldn't be in actual quine

dumb quine solution

```
#include <stdio.h>
int main(void) {
    char buffer[1024];
    FILE *f = fopen("quine.c", "r");
    size_t bytes = fread(buffer, 1,
                         sizeof(buffer), f);
    fwrite(buffer, 1, bytes, stdout);
    return 0;
}
```

a lot more straightforward!

but “cheating”

Vienna copying

```
mov $0x8f9, %si // %si = beginning of virus data
...
mov $0x288, %cx // length of virus
mov $0x40, %ah  // system call # for write
mov %si, %dx
sub $0x1f9, %dx // %dx = beginning of virus code
int 0x21 // make write system call
```

Vienna copying

```
mov $0x8f9, %si // %si = beginning of virus data
...
mov $0x288, %cx // length of virus
mov $0x40, %ah  // system call # for write
mov %si, %dx
sub $0x1f9, %dx // %dx = beginning of virus code
int 0x21 // make write system call
```