self-replicating malware
Changelog

Corrections made in this version not in first posting:
  1 Feb 2017: slide 12: `cmpq` corrected to `test`
  28 Feb 2017: slide 7: REX prefix’s first nibble is `0100`
RE assignment

assembly reading practice

due Friday
last time

executable formats
   using Linux as example, but concepts same elsewhere

started x86 encoding

why?
   manipulating machine code
      malware does it
      a little bit on assignments

want you to have option besides “use objdump blindly”
on assignments
# Overall Encoding

<table>
<thead>
<tr>
<th>Instruction Prefixes</th>
<th>Opcode</th>
<th>ModR/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefixes of 1 byte each (optional)¹,²</td>
<td>1-, 2-, or 3-byte opcode</td>
<td>1 byte (if required)</td>
<td>1 byte (if required)</td>
<td>Address displacement of 1, 2, or 4 bytes or none³</td>
<td>Immediate data of 1, 2, or 4 bytes or none³</td>
</tr>
</tbody>
</table>

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**
pushq %rax encoded as 50
   5-bit opcode 01010 plus 3-bit register number 000

pushq %r13 encoded as 41 55
   41: REX prefix 0100 (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 `0010+w:1, r:0, s:1, b:1`

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

84: ModRM: mod = 10; reg = 0000, 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
addq 0x12345678(%r10,%r11,2), %rax

4b: REX prefix 0010+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
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: relative and absolute

addresses in `mov/lea` are **absolute**

- address appears directly in machine code
- `mov foo, %eax:
  8b 04 25 (address of foo)
- except `mov foo(%rip), ...`, etc.

addresses in `jmp` are **relative**

- `jmp skip_nop; nop; skip_nop: ...:
  eb 01 (jmp skip_nop)
  90 (nop)
  (skip_nop:)
- value in machine code **added** to PC

addresses in `call` are relative
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 (signed) 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
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
string instructions (1)

memcpy: // copy %rdx bytes from (%rsi) to (%rdi)
test %rdx, %rdx
je done
movsb
subq $1, %rdx
jmp memcpy
done: ret

movsb (move data from string to string, byte)

mov one byte from (%rsi) to (%rdi)
increment %rsi and %rdi (*)
cannot specify other registers
string instructions (2)

memcpy: // copy %rdx bytes from (%rsi) to (%rdi)
    rep movsb
    ret

rep prefix byte

repeat instruction until %rdx is 0
decrement %rdx each time

cannot specify other registers

cannot use rep with all instructions
string instructions (3)

lodsb, stosb — load/store into string

movsw, movsd — word/dword versions

string comparison instructions

rep movsb is still recommended on modern Intel special-cased in processor?
exploring assembly

compiling little C programs looking at the assembly is nice:

```bash
gcc -S -O
extra stuff like .cfi directives (for try/catch)
```

or disassemble:

```bash
gcc -O -c file.c (or make an executable)
objdump -dr file.o (or on an executable)
  d: disassemble
  r: show (non-dynamic) relocations
```
exploring assembly

compiling little C programs looking at the assembly is nice:

```
gcc -S -O
```

extra stuff like .cfi directives (for try/catch)

or disassemble:

```
gcc -O -c file.c (or make an executable)
objdump -dr file.o (or on an executable)
```

d: disassemble  
r: show (non-dynamic) relocations
assembly without optimizations

compilers do **really silly things** without optimizations:

```c
int sum(int x, int y) { return x + y; }
```

```
sum:
pushq %rbp
movq %rsp, %rbp
movl %edi, −4(%rbp)
movl %esi, −8(%rbp)
movl −4(%rbp), %edx
movl −8(%rbp), %eax
addl %edx, %eax
popq %rbp
ret
```

instead of `gcc -O` version:

```
sum:
leal (%rdi,%rsi), %eax
ret
```
assembly reading advice

don’t know what an instruction does: look it up!

machine code: start with assembler/objdump
    might need to edit addresses, etc.

remember calling conventions

function/variable names (if present) help

try to name values in registers, on stack
    based on context
    “input size” not “rax”
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
## Virus Prevalence

### Viruses on Commercially Sold Software Media

From 1990 memo by Chris McDonald:

### 4. MS-DOS Infections

<table>
<thead>
<tr>
<th>SOFTWARE</th>
<th>REPORTING LOCATION</th>
<th>DATE</th>
<th>VIRAL INFECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. Unlock Masterkey</td>
<td>Kennedy Space Center</td>
<td>Oct 89</td>
<td>Vienna</td>
</tr>
<tr>
<td>b. SARGON III</td>
<td>Iceland</td>
<td>Sep 89</td>
<td>Cascade (1704)</td>
</tr>
<tr>
<td>c. ASYST RTDEMO02.EXE</td>
<td>Fort Belvoir</td>
<td>Aug 89</td>
<td>Jerusalem-B</td>
</tr>
<tr>
<td>d. Desktop Fractal</td>
<td>Various</td>
<td>Jan 90</td>
<td>Jerusalem (1813)</td>
</tr>
<tr>
<td>e. Bureau of the Government Printing</td>
<td>Jan 90</td>
<td>Jerusalem-B</td>
<td></td>
</tr>
<tr>
<td>(Census, Elec. County Office/US Census Bureau &amp; City Data Bk., 1988)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f. Northern Computers</td>
<td>Iceland</td>
<td>Mar 90</td>
<td>Disk Killer</td>
</tr>
<tr>
<td>(PC Manufacturer shipped infected systems.)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 5. MACINTOSH Infections

<table>
<thead>
<tr>
<th>SOFTWARE</th>
<th>REPORTING LOCATION</th>
<th>DATE</th>
<th>VIRAL INFECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. NoteWriter</td>
<td>Colgate College</td>
<td>Sep 89</td>
<td>Scores and nVIR</td>
</tr>
</tbody>
</table>

[https://groups.google.com/forum/#!original/comp.virus/XJcfYR9T6nI/azflHz5g00oJ](https://groups.google.com/forum/#!original/comp.virus/XJcfYR9T6nI/azflHz5g00oJ)
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


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
```

....

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:

```assembly
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”
# 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
#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;%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
dumb quine solution

```c
#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

```assembly
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

\texttt{mov $0x8f9, \%si}  // \%si = beginning of virus data

...  
\texttt{mov $0x288, \%cx}  // length of virus
\texttt{mov $0x40, \%ah}  // system call \# for write
\texttt{mov \%si, \%dx}
\texttt{sub $0x1f9, \%dx}  // \%dx = beginning of virus code
\texttt{int 0x21}  // make write system call
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 relocation

very little use of absolute addresses:
  jmps use relative addresses (value to add to PC)

virus uses %si as a “base register”
  points to beginning of virus data
  set very early in virus execution

set via mov $0x8fd, %si near beginning of virus
Vienna relocation

// set virus data address:
0x700: `mov $0x8f9, %si`
  // machine code: be f9 08
  // be: opcode
  // f9 08: immediate
...
// %ax contains file length (of file to infect)
`mov %ax, %cx`
...
`add $0x2f9, %cx`
`mov %si, %di`
`sub $0x1f7, %di` // %di ← 0x701
`mov %cx, (%di)` // update mov instruction
...
Vienna relocation

// set virus data address:
0x700:  mov $0x8f9, %si
       // machine code: be f9 08
       // be: opcode
       // f9 08: immediate

...%ax contains file length (of file to infect)

mov %ax, %cx

...

add $0x2f9, %cx

mov %si, %di

sub $0x1f7, %di  // %di ← 0x701

mov %cx, (%di)  // update mov instruction

...
Vienna relocation

// set virus data address:
0x700:  mov $0x8f9, %si
    // machine code: be f9 08
    // be: opcode
    // f9 08: immediate

...  
// %ax contains file length (of file to infect)
mov %ax, %cx  

...  
add $0x2f9, %cx
mov %si, %di
sub $0x1f7, %di // %di ← 0x701
mov %cx, (%di) // update mov instruction  
...
Vienna relocation

edit actual code for mov

why doesn’t this disrupt virus execution?
Vienna relocation

edit actual code for mov

why doesn’t this disrupt virus execution?
    already ran that instruction
Vienna relocation

0x700: \texttt{mov} $0x8f9, \%si  
...  
// \%ax contains file length  
// (of file to infect)  
\texttt{mov} \%ax, \%cx  
\texttt{sub} $3, \%ax  
// update template jmp instruction  
\texttt{mov} \%ax, 0xe(\%si) // 0xe + \%si = 0x907  
...  
\texttt{mov} $40, \%ah  
\texttt{mov} $3, \%cx  
\texttt{mov} \%si, \%dx  
\texttt{add} $0xD, \%dx // dx ← 0x906  
int 0x21 // system call: write 3 bytes from 0x906  
...  
0x906: e9 fd 05 // jmp PC+FD 05
Vienna relocation

0x700: mov $0x8f9, %si
...
// %ax contains file length
// (of file to infect)
mov %ax, %cx
sub $3, %ax
// update template jmp instruction
mov %ax, 0xe(%si) // 0xe + %si = 0x907
...
mov $40, %ah
mov $3, %cx
mov %si, %dx
add $0xD, %dx // dx ← 0x906
int 0x21 // system call: write 3 bytes from 0x906
...
0x906: e9 fd 05 // jmp PC+FD 05
Vienna relocation

0x700: mov $0x8f9, %si
...
// %ax contains file length
// (of file to infect)
mov %ax, %cx
sub $3, %ax
// update template jmp instruction
mov %ax, 0xe(%si) // 0xe + %si = 0x907
...
mov $40, %ah
mov $3, %cx
mov %si, %dx
add $0xD, %dx // dx ← 0x906
int 0x21 // system call: write 3 bytes from 0x906
...
0x906: e9 fd 05 // jmp PC+FD 05
alternative relocation

could avoid having pointer to update:

0000000000000000 <next-0x3>:
  0:   e8 00 00              call  3 <next>
      target addresses encoded relatively
      pushes return address (next) onto stack

0000000000000003 <next>:
  3:   59                    pop   %cx
      cx contains address of the pop instruction

why didn’t Vienna do this?
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: avoiding reinfection

scans through active directories for executables

“marks” infected executables in file metadata

could have checked for virus code — but slow
DOS last-written times

16-bit number for date; 16-bit number for time

<table>
<thead>
<tr>
<th>Y-1980</th>
<th>Mon</th>
<th>Day</th>
<th>H</th>
<th>Min</th>
<th>Sec/2</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>9</td>
<td>8</td>
<td>15</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
DOS last-written times

16-bit number for date; 16-bit number for time

<table>
<thead>
<tr>
<th>Y-1980</th>
<th>Mon</th>
<th>Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>H</th>
<th>Min</th>
<th>Sec/2</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Sec/2: 5 bits: range from 0–31 corresponds to 0 to 62 seconds

Vienna trick: set infected file times to 62 seconds need to update times anyways — hide tracks
virus choices

where to put code

how to get code ran
virus choices

where to put code

how to get code ran
where to put code

considerations:

spreading — files that will be copied/reused
spreading — files that will be ran
stealth — user shouldn’t know until too late
where to put code: options

one or more of:

replacing executable code

after executable code (Vienna)

in unused executable code

inside OS code

in memory
where to put code: options

one or more of:

replacing executable code

after executable code (Vienna)

in unused executable code

inside OS code

in memory
replace executable

original executable

virus code
replace executable?

seems silly — not stealthy!

has appeared in the wild — ILOVEYOU

2000 ILOVEYOU Worm
  written in Visual Basic (!)
  spread via email
  replaced lots of files with copies of itself

huge impact
replace executable — subtle

original executable

virus code
run original from tempfile

original executable
where to put code: options

one or more of:
replacing executable code
after executable code (Vienna)
in unused executable code
inside OS code
in memory
appending

original executable

original executable

virus code

jmp to virus
note about appending

COM files are very simple — no metadata
modern executable formats have length information to update
  add segment to program header
  update last segment of program header (size + make it executable)
compressing viruses

file too big? how about compression

original executable

---

virus code

decompressor

compressed executable

unused space
where to put code: options

one or more of:
replacing executable code
after executable code (Vienna)
in unused executable code
inside OS code
in memory
unused code???

why would a program have unused code???
unused code case study: /bin/ls

unreachable no-ops!

... 403788: e9 59 0c 00 00 jmpq 4043e6 <__sprintf_chk@plt+0x1a06> 
40378d: 0f 1f 00 nopl (%rax)  
403790: ba 05 00 00 00 mov $0x5,%edx  
...
403ab9: eb 4d jmp 403b08 <__sprintf_chk@plt+0x1128>  
403abb: 0f 1f 44 00 00 nopl 0x0(%rax,%rax,1)  
403ac0: 4d 8b 7f 08 mov 0x8(%r15),%r15  
...
404a01: c3 retq  
404a02: 0f 1f 40 00 nopl 0x0(%rax)  
404a06: 66 2e 0f 1f 84 00 00 nopw %cs:0x0(%rax,%rax)  
404a0d: 00 00 00 mov $0x61e600,%esi  
404a10: be 00 e6 61 00 mov $0x61e600,%esi  
...
why empty space?


better for instruction cache (and TLB and related caches)
better for instruction decode logic
function calls count as branches for this purpose
other empty space

unused dynamic linking structure

unused debugging/symbol table information?

unused header space

recall — header loaded into memory!
other empty space

unused dynamic linking structure

unused debugging/symbol table information?

unused header space

recall — header loaded into memory!
dynamic linking cavity

.dynam ic section — data structure used by dynamic linker:

format: list of 8-byte type, 8-byte value terminated by type == 0 entry

Contents of section .dynamic:

600e28 01000000 00000000 01000000 00000000 ..................
... several non-empty entries ...
600f88 f0ffff6f 00000000 56034000 00000000 ...o....V.@.....
VERSYM (required library version info at) 0x400356
600f98 00000000 00000000 00000000 00000000 ..................
NULL --- end of linker info
600fa8 00000000 00000000 00000000 00000000 ..................
unused! (and below)
600fb8 00000000 00000000 00000000 00000000 ..................
600fc8 00000000 00000000 00000000 00000000 ..................
600fd8 00000000 00000000 00000000 00000000 ..................
600fe8 00000000 00000000 00000000 00000000 ..................
is there enough empty space?

cavities look awfully small

really small viruses?

solution: chain cavities together
case study: CIH (1)

original executable

- virus startup code
- virus code locs
  - virus code part 1
  - virus code part 2
  - virus code part 3
case study: CIH (2)

in memory:

<table>
<thead>
<tr>
<th>virus startup code</th>
</tr>
</thead>
<tbody>
<tr>
<td>virus code locs (table)</td>
</tr>
<tr>
<td>virus code part 1</td>
</tr>
<tr>
<td>virus code part 2</td>
</tr>
<tr>
<td>virus code part 3</td>
</tr>
</tbody>
</table>

| virus code part 1           |
| virus code part 2           |
| virus code part 3           |
CIH cavities

gaps between sections
  common Windows linker aligned sections
  (align = start on address multiple of \( N \), e.g. 4096)
  (normal Linux linker doesn’t do this...)

reassembling code avoids worrying about splitting instructions
where to put code: options

one or more of:

replacing executable code

after executable code (Vienna)

in unused executable code

inside OS code

in memory
boot process

processor reset

BIOS/EFI
(chip on motherboard)

very CPU/motherboard-specific code

fixed location on disk
code that understands files

bootloader

files in a filesystem

operating system
boot process

very CPU/motherboard-specific code

fixed location on disk
code that understands files

files in a filesystem

processor reset

BIOS/EFI
(chip on motherboard)

bootloader

operating system
bootloaders in the DOS era

used to be common to boot from floppies

default to booting from floppy if present
even if hard drive to boot from

applications distributed as bootable floppies

so bootloaders on all devices were a target for viruses
historic bootloader layout

bootloader in first sector (512 bytes) of device
(along with partition information)

code in BIOS to copy bootloader into RAM, start running

bootloader responsible for disk I/O etc.

some library-like functionality in BIOS for I/O
bootloader viruses

example: Stoned

- partition table
- bootloader

- partition table
- virus code
- partition table (unused)
- saved bootloader
bootloader viruses

example: Stoned

- partition table
- bootloader
- data here???

- partition table
- virus code
- partition table (unused)
- saved bootloader
data here???

might be data there — risk

some unused space after partition table/boot loader common

(allegedly)

also be filesystem metadata not used on smaller floppies/disks

but could be wrong — oops
modern bootloaders — UEFI

BIOS-based boot is going away (slowly)

new thing: UEFI (Universal Extensible Firmware Interface)

like BIOS:
  library functionality for bootloaders
  loads initial code from disk/DVD/etc.

unlike BIOS:
  much more understanding of file systems
  much more modern set of library calls
modern bootloaders — secure boot

“Secure Boot” is a common feature of modern bootloaders

idea: UEFI/BIOS code checks bootloader code, fails if not okay
    requires user intervention to use not-okay code
Secure Boot and keys

Secure Boot relies on cryptographic signatures
  idea: accept only “legitimate” bootloaders
  legitimate: known authority vouched for them

user control of their own systems?
  in theory: can add own keys

what about changing OS instead of bootloader?
  need smart bootloader
boot process

processor reset

very CPU/motherboard-specific code →

BIOS/EFI (chip on motherboard)

fixed location on disk

code that understands files

bootloader

files in a filesystem →

operating system
BIOS/UEFI implants

infrequent

BIOS/UEFI code is very non-portable

BIOS/UEFI update often requires physical access

BIOS/UEFI code sometimes requires cryptographic signatures

...but very hard to remove — can reinstall other malware

reports that Hacking Team (Milan-based malware company) had UEFI-infecting “rootkit”
The boot process begins with the processor reset, which triggers the BIOS/EFI (chip on motherboard). BIOS/EFI is very CPU/motherboard-specific code located at a fixed location on disk. It contains a bootloader, which is code that understands files and loads the operating system. The operating system then manages the files in a filesystem.
system files

simpliest strategy: stuff that runs when you start your computer

add a new startup program, run in the background easy to blend in

alternatively, infect one of many system programs automatically run
malware wants to keep doing stuff

one option — background process (easy on modern OSs)

also stealthy options:
  - insert self into OS code
  - insert self into other running programs

more commonly, OS code used for hiding malware
  - topic for later
virus choices

where to put code

how to get code ran
invoking virus code: options

boot loader

change starting location

alternative approaches: “entry point obscuring”

edit code that’s going to run anyways

replace a function pointer (or similar)

...
invoking virus code: options

boot loader

change starting location

alternative approaches: “entry point obscuring”

edit code that’s going to run anyways

replace a function pointer (or similar)

...
starting locations

/bin/ls: file format elf64-x86-64
/bin/ls
architecture: i386:x86-64, flags 0x00000112: EXEC_P, HAS_SYMS, D_PAGED
start address 0x00000000004049a0

modern executable formats have ‘starting address’ field

just change it, insert jump to old address after virus code
invoking virus code: options

boot loader
change starting location
alternative approaches: “entry point obscuring”
edit code that’s going to run anyways
replace a function pointer (or similar)
...
...
run anyways?

add code at start of program (Vienna)

return with padding after it:

```
  404a01:  c3
  404a02:  0f 1f 40 00
replace with
  404a01:  e9 XX XX XX XX
```

replace with
```
  404a01:  e9 XX XX XX XX
```

any random place in program?

just not in the middle of instruction
challenge: valid locations

x86: probably don’t want a full instruction parser

x86: might be non-instruction stuff mixed in with code:

do_some_floating_point_stuff:
    movss float_one(%rip), %xmm0
    ...
    retq

float_one: .float 1

floating point value one (00 00 80 3f) is not valid machine code
disassembler might lose track of instruction boundaries
finding function calls

one idea: replace calls

normal x86 call FOO: E8 (32-bit value: PC – address of foo)

could look for E8 in code — lots of false positives
probably even if one excludes out-of-range addresses
really finding function calls

e.g. some popular compilers started x86-32 functions with

foo:

```
push %ebp       // push old frame pointer
  // 0x55
mov %ebp, %esp  // set frame pointer to stack
  // 0x89 0xec
```

use to identify when e8 refers to real function

(full version: also have some other function start patterns)
remember stubs?

```
0000000000400400 <puts@plt>:
  400400:  ff 25 12 0c 20 00          jmpq  *0x200c12(%rip)
        /* 0x200c12+RIP = _GLOBAL_OFFSET_TABLE_+0x18 */
  400406:  68 00 00 00 00          pushq  $0x0
  40040b:  e9 e0 ff ff ff          jmpq  4003f0 <_init+0x28>

replace with:
  400400:  e8 XX XX XX XX          jmpq  virus_code
  400405:  90                      nop
  400406:  68 00 00 00 00          pushq  $0x0
  40040b:  e9 e0 ff ff ff          jmpq  4003f0 <_init+0x28>
```

in known location (particular section of executable)
invoking virus code: options

boot loader

change starting location

alternative approaches: “entry point obscuring”

edit code that’s going to run anyways

replace a function pointer (or similar)

...
stubs again

000000000000400400 <puts@plt>:
400400:    ff 25 12 0c 20 00 jmpq *0x200c12(%rip)
   /* 0x200c12+RIP = _GLOBAL_OFFSET_TABLE_+0x18 */
400406:    68 00 00 00 00 pushq $0x0
40040b:    e9 e0 ff ff ff jmpq 4003f0 <_init+0x28>

don't edit stub — edit initial value of _GLOBAL_OFFSET_TABLE_
 stored in data section of executable

originally: pointer 0x400406; new — virus code
relocations?

hello.exe: file format elf64-x86-64

DYNAMIC RELOCATION RECORDS
OFFSET TYPE VALUE
0000000000060ff8 R_X86_64_GLOB_DAT __gmon_start__
00000000000601018 R_X86_64_JUMP_SLOT puts@GLIBC_2.2.5
replace with:
00000000000601018 R_X86_64_JUMP_SLOT _start + offset_of_virus
00000000000601020 R_X86_64_JUMP_SLOT __libc_start_main@GLIBC_2.2.5

tricky — usually no symbols from executable in dynamic symbol table
(symbols from debugger/disassembler are a different table)
Linux — need to link with -rdynamic
relocations?

hello.exe: file format elf64-x86-64

<table>
<thead>
<tr>
<th>OFFSET</th>
<th>TYPE</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000600ff8</td>
<td>R_X86_64_GLOB_DAT</td>
<td><strong>gmon_start</strong></td>
</tr>
<tr>
<td>000000000601018</td>
<td>R_X86_64_JUMP_SLOT</td>
<td>puts@GLIBC_2.2.5</td>
</tr>
</tbody>
</table>

replace with:

<table>
<thead>
<tr>
<th>OFFSET</th>
<th>TYPE</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000601018</td>
<td>R_X86_64_JUMP_SLOT</td>
<td>_start + offset_of_virus</td>
</tr>
<tr>
<td>000000000601020</td>
<td>R_X86_64_JUMP_SLOT</td>
<td>__libc_start_main@GLIBC_2.2.5</td>
</tr>
</tbody>
</table>

but...same idea works on shared library itself
infecting shared libraries

```
infecting shared libraries

kernel32.dll

header
symbol table
GetFileAttributesA
...

kernel32.dll

header
symbol table
GetFileAttributesA
...

virus code

93
```
summary

how to hide:
  separate executable
  append
  existing “unused” space
  compression

how to run:
  change entry point
  or “entry point obscuring”:
  change some code (requires care!)
  change library
### 32-bit ModRM table

<table>
<thead>
<tr>
<th>Effective Address</th>
<th>Mod</th>
<th>R/M</th>
</tr>
</thead>
<tbody>
<tr>
<td>[EAX]</td>
<td>00</td>
<td>000</td>
</tr>
<tr>
<td>[ECX]</td>
<td></td>
<td>001</td>
</tr>
<tr>
<td>[EDX]</td>
<td></td>
<td>010</td>
</tr>
<tr>
<td>[EBX]</td>
<td></td>
<td>011</td>
</tr>
<tr>
<td>[EBP]</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>[ESI]</td>
<td></td>
<td>101</td>
</tr>
<tr>
<td>[EDI]</td>
<td></td>
<td>110</td>
</tr>
<tr>
<td>[ESP]</td>
<td></td>
<td>111</td>
</tr>
<tr>
<td>[EBP]</td>
<td>01</td>
<td>000</td>
</tr>
<tr>
<td>[ESP]</td>
<td></td>
<td>001</td>
</tr>
<tr>
<td>[ESI]</td>
<td></td>
<td>010</td>
</tr>
<tr>
<td>[EDI]</td>
<td></td>
<td>011</td>
</tr>
<tr>
<td>[ESP]</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>[ESP]</td>
<td></td>
<td>101</td>
</tr>
<tr>
<td>[ESP]</td>
<td></td>
<td>110</td>
</tr>
<tr>
<td>[ESP]</td>
<td></td>
<td>111</td>
</tr>
<tr>
<td>[EBP]</td>
<td>10</td>
<td>000</td>
</tr>
<tr>
<td>[ESP]</td>
<td></td>
<td>001</td>
</tr>
<tr>
<td>[ESI]</td>
<td></td>
<td>010</td>
</tr>
<tr>
<td>[EDI]</td>
<td></td>
<td>011</td>
</tr>
<tr>
<td>[ESP]</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>[ESP]</td>
<td></td>
<td>101</td>
</tr>
<tr>
<td>[ESP]</td>
<td></td>
<td>110</td>
</tr>
<tr>
<td>[ESP]</td>
<td></td>
<td>111</td>
</tr>
<tr>
<td>[EBP]</td>
<td>11</td>
<td>000</td>
</tr>
<tr>
<td>[ESP]</td>
<td></td>
<td>001</td>
</tr>
<tr>
<td>[ESI]</td>
<td></td>
<td>010</td>
</tr>
<tr>
<td>[EDI]</td>
<td></td>
<td>011</td>
</tr>
<tr>
<td>[ESP]</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>[ESP]</td>
<td></td>
<td>101</td>
</tr>
<tr>
<td>[ESP]</td>
<td></td>
<td>110</td>
</tr>
<tr>
<td>[ESP]</td>
<td></td>
<td>111</td>
</tr>
</tbody>
</table>
Table 2-3. 32-Bit Addressing Forms with the SIB Byte

<table>
<thead>
<tr>
<th>r32</th>
<th>EAX</th>
<th>ECX</th>
<th>EDX</th>
<th>EBX</th>
<th>ESP</th>
<th>[*]</th>
<th>ESI</th>
<th>EDI</th>
</tr>
</thead>
<tbody>
<tr>
<td>(In decimal)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>(In binary)</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scaled Index</th>
<th>SS</th>
<th>Index</th>
<th>Value of SIB Byte (in Hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[EAX]</td>
<td>00</td>
<td>000</td>
<td>00 01 02 03 04 05 06 07</td>
</tr>
<tr>
<td>[ECX]</td>
<td>00</td>
<td>001</td>
<td>01 09 0A 0B 0C 0D 0E 0F</td>
</tr>
<tr>
<td>[EDX]</td>
<td>00</td>
<td>010</td>
<td>02 0A 0B 0C 0D 0E 0F 10</td>
</tr>
<tr>
<td>[EBX]</td>
<td>00</td>
<td>011</td>
<td>03 0B 0C 0D 0E 0F 10 11</td>
</tr>
<tr>
<td>none</td>
<td>00</td>
<td>100</td>
<td>10 19 1A 1B 1C 1D 1E 1F</td>
</tr>
<tr>
<td>[EBP]</td>
<td>00</td>
<td>101</td>
<td>11 18 19 1A 1B 1C 1D 1E</td>
</tr>
<tr>
<td>[ESI]</td>
<td>00</td>
<td>110</td>
<td>12 17 18 19 1A 1B 1C 1D</td>
</tr>
<tr>
<td>[EDI]</td>
<td>00</td>
<td>111</td>
<td>13 16 17 18 19 1A 1B 1C</td>
</tr>
<tr>
<td>[EAX*2]</td>
<td>01</td>
<td>000</td>
<td>40 41 42 43 44 45 46 47</td>
</tr>
<tr>
<td>[ECX*2]</td>
<td>01</td>
<td>001</td>
<td>42 43 44 45 46 47 48 49</td>
</tr>
<tr>
<td>[EDX*2]</td>
<td>01</td>
<td>010</td>
<td>44 45 46 47 48 49 50 51</td>
</tr>
<tr>
<td>[EBX*2]</td>
<td>01</td>
<td>011</td>
<td>51 52 53 54 55 56 57 58</td>
</tr>
<tr>
<td>none</td>
<td>01</td>
<td>100</td>
<td>59 60 61 62 63 64 65 66</td>
</tr>
<tr>
<td>[EBP*2]</td>
<td>01</td>
<td>101</td>
<td>67 68 69 70 71 72 73 74</td>
</tr>
<tr>
<td>[ESI*2]</td>
<td>01</td>
<td>110</td>
<td>75 76 77 78 79 80 81 82</td>
</tr>
<tr>
<td>[EDI*2]</td>
<td>01</td>
<td>111</td>
<td>83 84 85 86 87 88 89 90</td>
</tr>
<tr>
<td>[EAX*4]</td>
<td>10</td>
<td>000</td>
<td>8A 8B 8C 8D 8E 8F</td>
</tr>
<tr>
<td>[ECX*4]</td>
<td>10</td>
<td>001</td>
<td>9A 9B 9C 9D 9E 9F</td>
</tr>
<tr>
<td>[EDX*4]</td>
<td>10</td>
<td>010</td>
<td>AA 9A 9B 9C 9D 9E 9F 10</td>
</tr>
<tr>
<td>[EBX*4]</td>
<td>10</td>
<td>011</td>
<td>AB 9B 9C 9D 9E 9F 10 11</td>
</tr>
<tr>
<td>none</td>
<td>10</td>
<td>100</td>
<td>AC 9B 9C 9D 9E 9F 10 11</td>
</tr>
<tr>
<td>[EBP*4]</td>
<td>10</td>
<td>101</td>
<td>AD 9B 9C 9D 9E 9F 10 11</td>
</tr>
<tr>
<td>[ESI*4]</td>
<td>10</td>
<td>110</td>
<td>AE 9B 9C 9D 9E 9F 10 11</td>
</tr>
<tr>
<td>[EDI*4]</td>
<td>10</td>
<td>111</td>
<td>AF 9B 9C 9D 9E 9F 10 11</td>
</tr>
<tr>
<td>[EAX*8]</td>
<td>11</td>
<td>000</td>
<td>0B 0C 0D 0E 0F 10</td>
</tr>
<tr>
<td>[ECX*8]</td>
<td>11</td>
<td>001</td>
<td>0C 0D 0E 0F 10 11</td>
</tr>
<tr>
<td>[EDX*8]</td>
<td>11</td>
<td>010</td>
<td>0D 0E 0F 10 11 12</td>
</tr>
<tr>
<td>[EBX*8]</td>
<td>11</td>
<td>011</td>
<td>0E 0F 10 11 12 13</td>
</tr>
<tr>
<td>none</td>
<td>11</td>
<td>100</td>
<td>0F 10 11 12 13 14</td>
</tr>
<tr>
<td>[EBP*8]</td>
<td>11</td>
<td>101</td>
<td>10 11 12 13 14 15</td>
</tr>
<tr>
<td>[ESI*8]</td>
<td>11</td>
<td>110</td>
<td>11 12 13 14 15 16</td>
</tr>
<tr>
<td>[EDI*8]</td>
<td>11</td>
<td>111</td>
<td>12 13 14 15 16 17</td>
</tr>
</tbody>
</table>

**NOTES:**

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