virus 2 / object formats

last time

x86-64 encoding

- built up from 16-bit
- (prefix) (opcode) (mod/reg/rm) (sib) ...
- REX prefix for extra registers, 64- v 32-bit
- absolute versus relative address encoding

Vienna as case study

- add jmp at beginning of .COM executable files
- make backup copy of what replaced jmp
- append code to end of executable
- push + ret for jump back to avoid changing relative jump

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?

Vienna relocation

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

Vienna design: need to access global variables, etc.
solution: base pointer for virus data
problem: location changes depending on where virus is

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: 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
```

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 containts 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

Y-1980	Mon	Day
15 9	8 5	4 0

H	Min	Sec/2
15 11	10 5	4 0

DOS last-written times

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

Y-1980	Mon	Day
15 9	8 5	4 0

H	Min	Sec/2
15 11	10 5	4 0

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

Vienna: on detection

special metadata mark could be looked for

distinctive pattern, in well known place

- future assignment: pattern matching to find known malware

Vienna: non-portability

relies on very simple executable formats

- modern (read: anything after DOS) executable formats more complex/featureful

relies on self-modifying code

- often requires extra steps on modern systems

uses metadata on filesystem

- quirk of DOS filesystem timestamp format

Vienna: non-portability

relies on very simple executable formats

- modern (read: anything after DOS) executable formats more complex/featureful

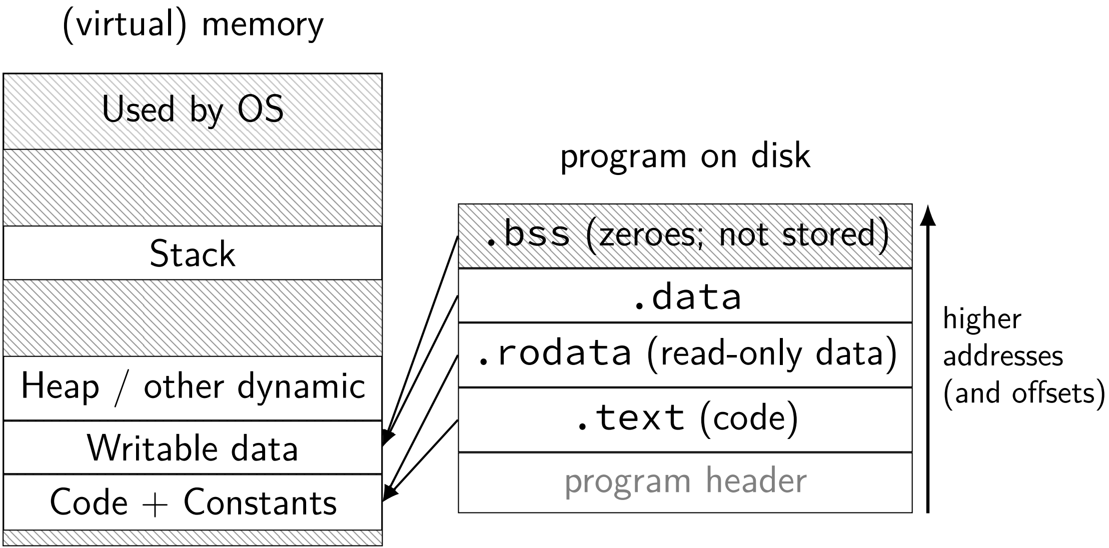
relies on self-modifying code

- often requires extra steps on modern systems

uses metadata on filesystem

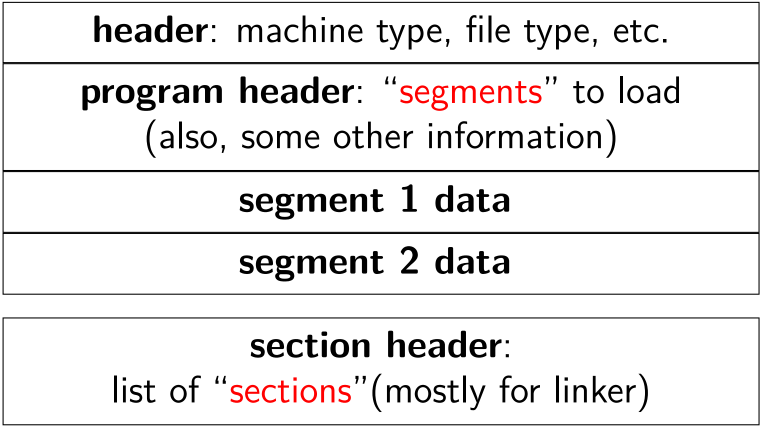
- quirk of DOS filesystem timestamp format

memory v. disk



ELF (executable and linking format)

Linux (and some others) executable/object file format



header: machine type, file type, etc.

program header: "segments" to load
(also, some other information)

segment 1 data

segment 2 data

section header:
list of "sections"(mostly for linker)

segments versus sections?

note: ELF terminology; may not be true elsewhere!

sections — object files (and usually executables), used by linker

- have information on intended purpose
- linkers combine these to create executables
- linkers might omit unneeded sections

segments — executables, used to actually load program

- program loader is dumb — doesn't know what segments are for

section headers

```
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .note.ABI-tag 00000020  0000000000400190  0000000000400190  00000190  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.gnu.build-id 00000024  00000000004001b0  00000000004001b0  000001b0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .rela.plt     00000210  00000000004001d8  00000000004001d8  000001d8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .init         0000001a  00000000004003e8  00000000004003e8  000003e8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  4 .plt          00000160  0000000000400410  0000000000400410  00000410  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  5 .text         0017ff1d  0000000000400570  0000000000400570  00000570  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  6 __libc_freeres_fn 00002032  0000000000580490  0000000000580490  00180490  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  7 __libc_thread_freeres_fn 0000021b  00000000005824d0  00000000005824d0  001824d0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  8 .fini         00000009  00000000005826ec  00000000005826ec  001826ec  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  9 .rodata       00044ac8  0000000000582700  0000000000582700  00182700  2**6
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 10 __libc_subfreeres 000000c0  00000000005c71c8  00000000005c71c8  001c71c8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 11 .stapsdt.base 00000001  00000000005c7288  00000000005c7288  001c7288  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 12 __libc_atexit 00000008  00000000005c7290  00000000005c7290  001c7290  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 13 __libc_thread_subfreeres 00000018  00000000005c7298  00000000005c7298  001c7298  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 14 .eh_frame     000141dc  00000000005c72b0  00000000005c72b0  001c72b0  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 15 .gcc_except_table 0000020b  00000000005db48c  00000000005db48c  001db48c  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 16 .tdata        00000030  00000000007dbea8  00000000007dbea8  001dbea8  2**3
                  CONTENTS, ALLOC, LOAD, DATA, THREAD_LOCAL
 17 .tbss         0000004a  00000000007dbed8  00000000007dbed8  001dbed8  2**3
```

sections

tons of “sections”

not actually needed/used to run program

size, file offset, flags (code/data/etc.)

- location in executable *and* in memory

some sections aren't stored (no “CONTENTS” flag)

- just all zeroes

selected sections

`.text`	program code
`.bss`	initially zero data (block started by symbol)
`.data`	other writeable data
`.rodata`	read-only data
`.init`/`.fini`	global constructors/destructors
`.got`/`.plt`	dynamic linking related
`.eh_frame`	try/catch related

based on `http://people.redhat.com/mpolacek/src/devconf2012.pdf`

ELF example

`objdump -x /bin/busybox` (on my laptop)

`-x`: output all headers

```
/bin/busybox:     file format elf64-x86-64
/bin/busybox
architecture: i386:x86-64, flags 0x00000102:
EXEC_P, D_PAGED
start address 0x0000000000402170

Program Header:
[...]

Sections:
[...]
```

ELF example

`objdump -x /bin/busybox` (on my laptop)

`-x`: output all headers

```
/bin/busybox:     file format elf64-x86-64
/bin/busybox
architecture: i386:x86-64, flags 0x00000102:
EXEC_P, D_PAGED
start address 0x0000000000402170

Program Header:
[...]

Sections:
[...]
```

ELF example

`objdump -x /bin/busybox` (on my laptop)

`-x`: output all headers

```
/bin/busybox:     file format elf64-x86-64
/bin/busybox
architecture: i386:x86-64, flags 0x00000102:
EXEC_P, D_PAGED
start address 0x0000000000402170

Program Header:
[...]

Sections:
[...]
```

a program header (1)

```
Program Header:
[...]
LOAD off    0x0001000 vaddr 0x0401000 paddr 0x0401000 align 2**12
     filesz 0x01b04ed memsz 0x01b04ed flags r-x
[...]
LOAD off    0x0207950 vaddr 0x0608950 paddr 0x0608950 align 2**12
     filesz 0x0008f40 memsz 0x000c718 flags rw-
```

load `0x1bd04ed` bytes:

- from `0x1000` bytes into the file
- to memory at `0x401000`
- readable and executable

load `0x8f40` bytes:

- from `0x207950` bytes into the file
- to memory at `0x608950`
- plus (`0xc718`–`0x8f40`) bytes of zeroes
- readable and writable

a program header (1)

```
Program Header:
[...]
LOAD off    0x0001000 vaddr 0x0401000 paddr 0x0401000 align 2**12
     filesz 0x01b04ed memsz 0x01b04ed flags r-x
[...]
LOAD off    0x0207950 vaddr 0x0608950 paddr 0x0608950 align 2**12
     filesz 0x0008f40 memsz 0x000c718 flags rw-
```

load `0x1bd04ed` bytes:

from `0x1000` bytes into the file
to memory at `0x401000`
readable and executable

load `0x8f40` bytes:

from `0x207950` bytes into the file
to memory at `0x608950`
plus (`0xc718`–`0x8f40`) bytes of zeroes
readable and writable

a program header (1)

```
Program Header:
[...]
LOAD off    0x0001000 vaddr 0x0401000 paddr 0x0401000 align 2**12
     filesz 0x01b04ed memsz 0x01b04ed flags r-x
[...]
LOAD off    0x0207950 vaddr 0x0608950 paddr 0x0608950 align 2**12
     filesz 0x0008f40 memsz 0x000c718 flags rw-
```

load `0x1bd04ed` bytes:

- from `0x1000` bytes into the file
- to memory at `0x401000`
- readable and executable

load `0x8f40` bytes:

- from `0x207950` bytes into the file
- to memory at `0x608950`
- plus (`0xc718`–`0x8f40`) bytes of zeroes
- readable and writable

a program header (1)

```
Program Header:
[...]
LOAD off    0x0001000 vaddr 0x0401000 paddr 0x0401000 align 2**12
     filesz 0x01b04ed memsz 0x01b04ed flags r-x
[...]
LOAD off    0x0207950 vaddr 0x0608950 paddr 0x0608950 align 2**12
     filesz 0x0008f40 memsz 0x000c718 flags rw-
```

load `0x1bd04ed` bytes:

- from `0x1000` bytes into the file
- to memory at `0x401000`
- readable and executable

load `0x8f40` bytes:

- from `0x207950` bytes into the file
- to memory at `0x608950`
- plus (`0xc718`–`0x8f40`) bytes of zeroes
- readable and writable

a program header (2)

```
Program Header:
[...]
    NOTE off    0x0000290 vaddr 0x0400290 paddr 0x0400290 align 2**2
         filesz 0x0000044 memsz 0x0000044 flags r--
     TLS off    0x0207950 vaddr 0x0608950 paddr 0x0608950 align 2**3
         filesz 0x0000030 memsz 0x0000092 flags r--
0x6474e553 off  0x0000270 vaddr 0x0400270 paddr 0x0400270 align 2**3
         filesz 0x0000020 memsz 0x0000020 flags r--
   STACK off    0x0000000 vaddr 0x0000000 paddr 0x0000000 align 2**4
         filesz 0x0000000 memsz 0x0000000 flags rw-
   RELRO off    0x0207950 vaddr 0x0608950 paddr 0x0608950 align 2**0
         filesz 0x00066b0 memsz 0x00066b0 flags r--
[...]
```

NOTE — comment

TLS — thread-local storage region (used via `%fs`)

0x6474e553 — ‘GNU_PROPERTY’ — adtl linker/loader info

STACK — indicates stack is read/write

RELRO — make this read-only after runtime linking

exercise

```
LOAD off    0x0000000 vaddr 0x00400000 paddr 0x00400000 align
     filesz 0x0000518 memsz 0x00000518 flags r--
LOAD off    0x0001000 vaddr 0x00401000 paddr 0x00401000 align
     filesz 0x009352d memsz 0x0009352d flags r-x
LOAD off    0x0095000 vaddr 0x00495000 paddr 0x00495000 align
     filesz 0x00265e5 memsz 0x000265e5 flags r--
LOAD off    0x00bc0c0 vaddr 0x004bd0c0 paddr 0x004bd0c0 align
     filesz 0x0006170 memsz 0x000078c0 flags rw-
```

Q1: about how large is this executable on disk?

Q2: this executable contains a global array declared like
`int array[SIZE] = ...;`
what is the largest plausible value for SIZE based on the header?

ELF loading: pages

Linux, most other OSes manage memory/files in *pages*

- hardware feature: virtual memory
- on x86-64: typically 4096 bytes

changes how LOADs work:

- offset must be rounded to multiple of page size
- size loaded rounded up to whole number of pages

ELF loading: pages example

program header:

```
LOAD off    0x0000000 vaddr 0x00400000 paddr 0x00400000 align
     filesz 0x0000518 memsz 0x00000518 flags r--
LOAD off    0x0001000 vaddr 0x00401000 paddr 0x00401000 align
     filesz 0x009352d memsz 0x0009352d flags r-x
LOAD off    0x0095000 vaddr 0x00495000 paddr 0x00495000 align
     filesz 0x00265e5 memsz 0x000265e5 flags r--
LOAD off    0x00bc0c0 vaddr 0x004bd0c0 paddr 0x004bd0c0 align
     filesz 0x0006170 memsz 0x000078c0 flags rw-
```

---

actually loaded (via Linux `/proc/PID/maps`):

```
memory address       size        r/w? file offset
~~~~~~~~~~~~~~~~~ (~~~~~~~~)     ~~~~ ~~~~~~~~
00400000-00401000 (  0x1000)     r--p 00000000
00401000-00495000 ( 0x94000)     r-xp 00001000
00495000-004bc000 ( 0x27000)     r--p 00095000
004bd000-004c0000 (  0x3000)     r--p 000bc000
004c0000-004c4000 (  0x4000)     rw-p 000bf000
```

ELF loading: pages example

program header:

```
LOAD off    0x0000000 vaddr 0x00400000 paddr 0x00400000 align
     filesz 0x0000518 memsz 0x00000518 flags r--
LOAD off    0x0001000 vaddr 0x00401000 paddr 0x00401000 align
     filesz 0x009352d memsz 0x0009352d flags r-x
LOAD off    0x0095000 vaddr 0x00495000 paddr 0x00495000 align
     filesz 0x00265e5 memsz 0x000265e5 flags r--
LOAD off    0x00bc0c0 vaddr 0x004bd0c0 paddr 0x004bd0c0 align
     filesz 0x0006170 memsz 0x000078c0 flags rw-
```

actually loaded (via Linux /proc/PID/maps):

```
memory address       size        r/w? file offset
~~~~~~~~~~~~~~~~~ (~~~~~~~~)     ~~~~ ~~~~~~~~
00400000-00401000 (   0x1000)    r--p 00000000
00401000-00495000 (  0x94000)    r-xp 00001000
00495000-004bc000 (  0x27000)    r--p 00095000
004bd000-004c0000 (   0x3000)    r--p 000bc000
004c0000-004c4000 (   0x4000)    rw-p 000bf000
```

ELF loading: pages example

program header:

```
LOAD off    0x0000000 vaddr 0x00400000 paddr 0x00400000 align
     filesz 0x0000518 memsz 0x00000518 flags r--
LOAD off    0x0001000 vaddr 0x00401000 paddr 0x00401000 align
     filesz 0x009352d memsz 0x0009352d flags r-x
LOAD off    0x0095000 vaddr 0x00495000 paddr 0x00495000 align
     filesz 0x00265e5 memsz 0x000265e5 flags r--
LOAD off    0x00bc0c0 vaddr 0x004bd0c0 paddr 0x004bd0c0 align
     filesz 0x0006170 memsz 0x000078c0 flags rw-
```

actually loaded (via Linux /proc/PID/maps):

```
memory address        size          r/w? file offset
~~~~~~~~~~~~~~~~~ (~~~~~~~~)        ~~~~ ~~~~~~~~
00400000-00401000 (  0x1000)        r--p 00000000
00401000-00495000 ( 0x94000)        r-xp 00001000
00495000-004bc000 ( 0x27000)        r--p 00095000
004bd000-004c0000 (  0x3000)        r--p 000bc000
004c0000-004c4000 (  0x4000)        rw-p 000bf000
```

ELF loading: pages example

program header:

```
LOAD off    0x0000000 vaddr 0x00400000 paddr 0x00400000 align
     filesz 0x0000518 memsz 0x00000518 flags r--
LOAD off    0x0001000 vaddr 0x00401000 paddr 0x00401000 align
     filesz 0x009352d memsz 0x0009352d flags r-x
LOAD off    0x0095000 vaddr 0x00495000 paddr 0x00495000 align
     filesz 0x00265e5 memsz 0x000265e5 flags r--
LOAD off    0x00bc0c0 vaddr 0x004bd0c0 paddr 0x004bd0c0 align
     filesz 0x0006170 memsz 0x000078c0 flags rw-
```

actually loaded (via Linux `/proc/PID/maps`):

```
memory address       size        r/w? file offset
~~~~~~~~~~~~~~~~~ (~~~~~~~~~)     ~~~~ ~~~~~~~~
00400000-00401000 (   0x1000)     r--p 00000000
00401000-00495000 (  0x94000)     r-xp 00001000
00495000-004bc000 (  0x27000)     r--p 00095000
004bd000-004c0000 (   0x3000)     r--p 000bc000
004c0000-004c4000 (   0x4000)     rw-p 000bf000
```

preview: dynamic linking

shown so far:

statically linked executables

- include all library code (instead of loading it from other files)

whose code is loaded at fixed address

- instead of that address being changeable

most common today:
dynamically-linked, position-independent executables

where to put code

viruses insert code in other programs

Vienna's choice: end of executables

search for `.COM` executables on system

considerations for other options:

spreading: identifying useful files to infect

- will be copied elsewhere?
- will be run?

stealth: avoiding detection

- Vienna: file size changes — easy to find?
- Vienna: weird modification time — easy to find?

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

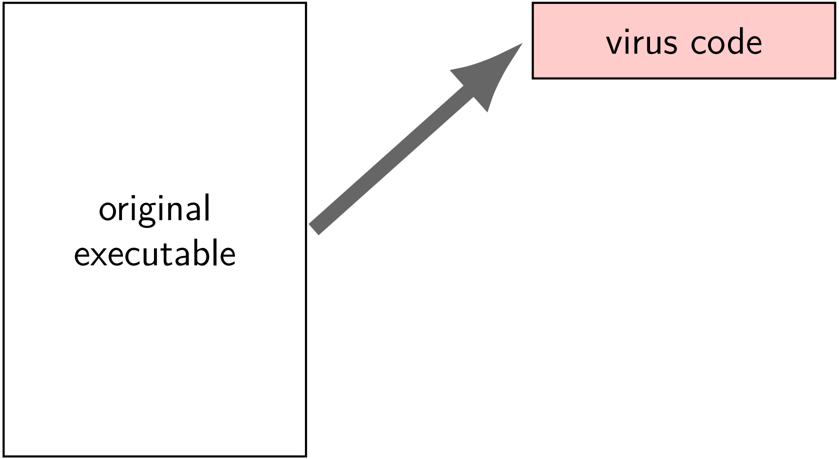
after executable code (Vienna)

in unused executable code

inside OS code

in memory

replace executable



replace executable?

seems silly — not stealthy!

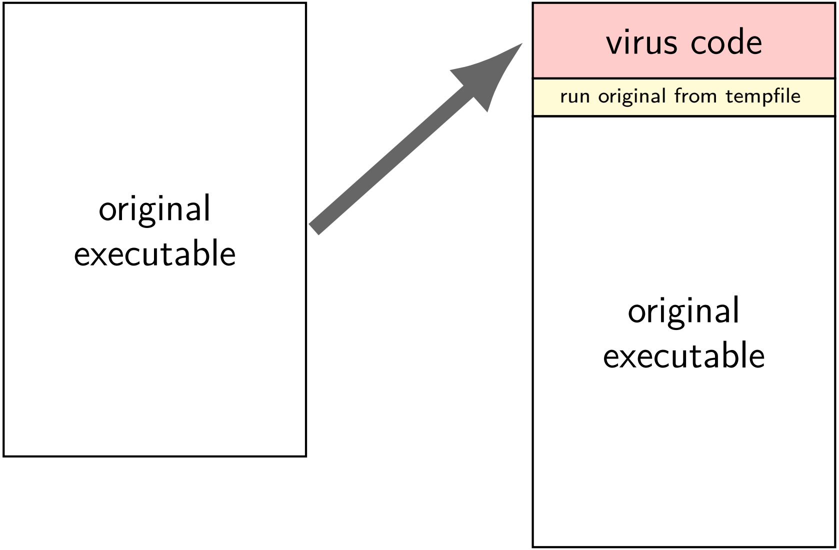
has appeared in the wild — ILOVEYOU

2000 ILOVEYOU Worm

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

huge impact — because destroying data to copy itself

replace executable — subtle



where to put code: options

one *or more* of:

replacing executable code

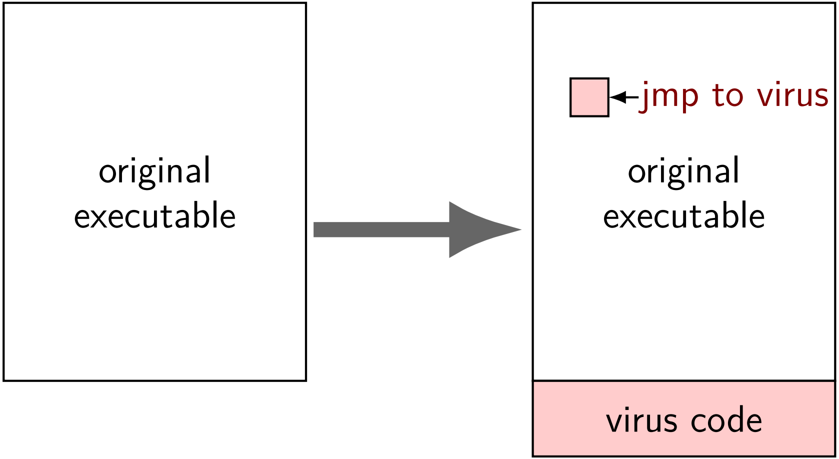
after executable code (Vienna)

in unused executable code

inside OS code

in memory

appending


original executable
jmp to virus
original executable
virus code

appending and executable formats

COM files are very simple — no metadata

modern executable formats have length information to update:

option 1: add segment (ELF LOAD) to program header

(often a little extra space after program header, due to page-alignment)

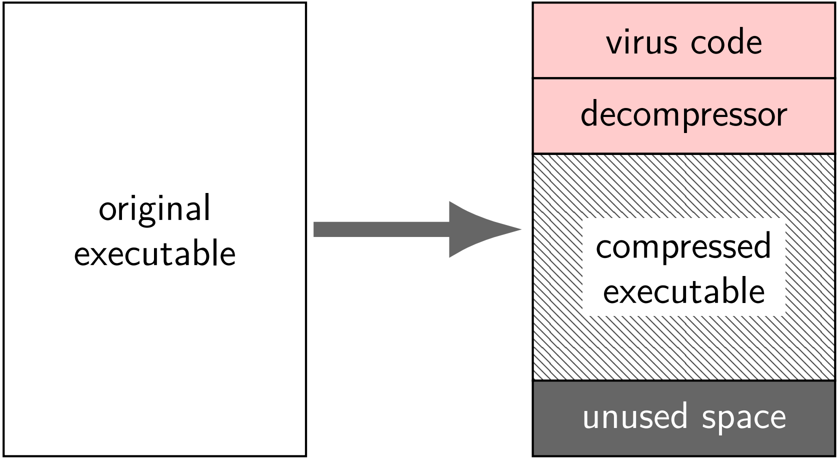
option 2: update last segment of program header

change its size

make it executable if it isn't (and often not — often data)

compressing viruses

file too big? how about compression



backup slides

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?

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
```