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 Design System	Various	Jan 90	Jerusalem (1813)
e.	Bureau of the Census, Elec. County & City Data Bk., 1988	Government Printing Office/US Census Bureau	Jan 90	Jerusalem-B
f.	Northern Computers (PC Manufacturer shipped infected systems.)	Iceland	Mar 90	Disk Killer

5. MACINTOSH INFECTIONS

	SOFTWARE	REPORTING LOCATION	DATE	VIRAL INFECTION
a.	NoteWriter	Colgate College	Sep 89	Scores and nVIR

https://groups.google.com/forum/#!original/comp.virus/XJCfYR9T6nI/azflHz5goooJ

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: 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="#includ
    char*x=%c%s
    %c%s%c";
int main(){
    printf(p,10,34,x,34,10,34,p,34,10,x,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 <s
    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);
}
```

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
```

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

...
// %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

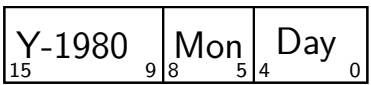
scans through active directories for executables

"marks" infected executables in *file metadata*

could have checked for virus code — but slow

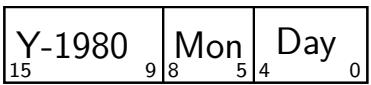
DOS last-written times

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



DOS last-written times

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



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

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

replace existing code

where to put code: options

one *or more* of:

replacing executable code

after executable code (Vienna)

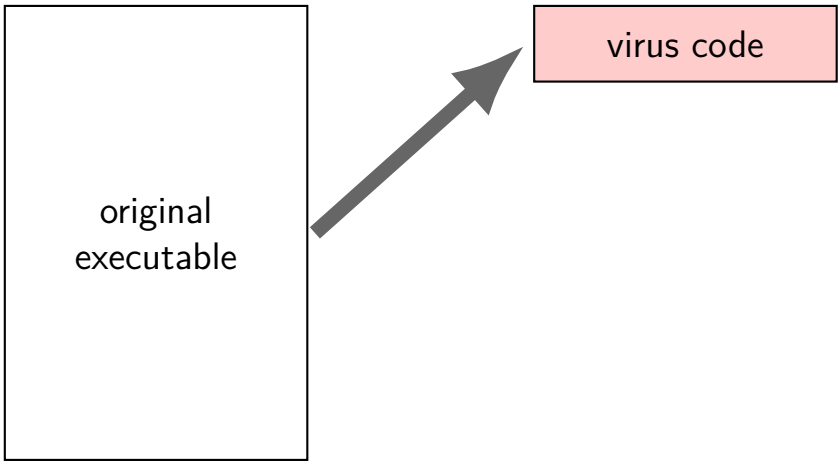
in unused executable code

inside OS code

in memory

replace existing code

replace executable



replace executable?

seems silly — not stealthy!

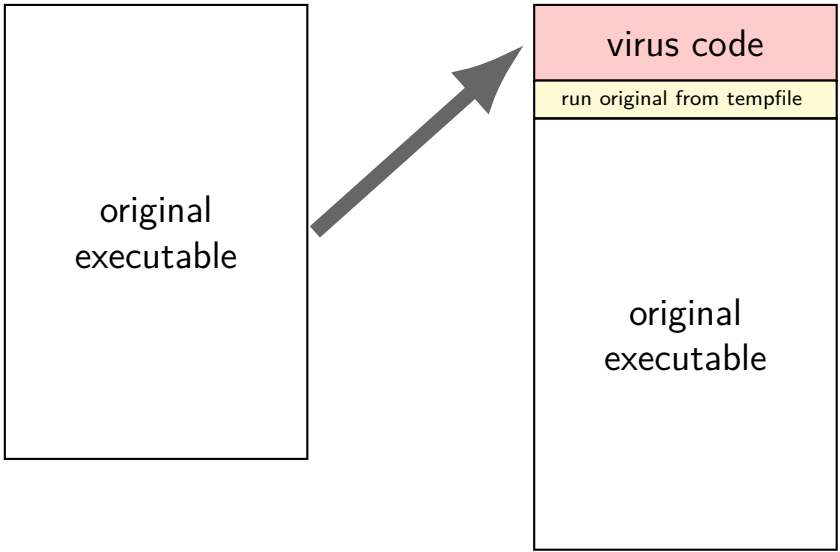
has appeared in the wild — ILOVEYOU

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)

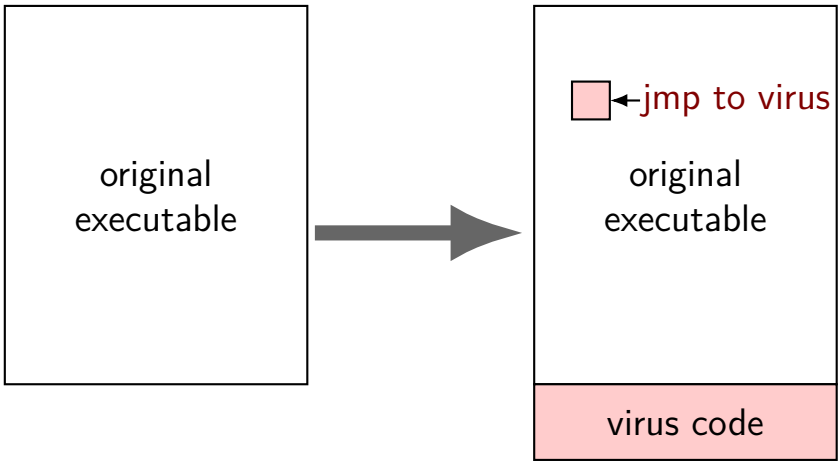
in unused executable code

inside OS code

in memory

replace existing code

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)

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 existing code

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+0x1
  40378d:        0f 1f 00                    nopl   (%rax)
  403790:        ba 05 00 00 00              mov    $0x5,%edx
...
  403ab9:        eb 4d                       jmp    403b08 <__sprintf_chk@plt+0x1
  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,1)
  404a0d:        00 00 00
  404a10:        be 00 e6 61 00              mov    $0x61e600,%esi
...
```

why empty space?

Intel Optimization Reference Manual:
"**Assembly/Compiler Coding Rule 12. (M impact, H generality)**
All branch targets should be 16-byte aligned."

better for instruction cache (and TLB and related caches)

better for instruction decode logic

function calls, jumps count as branches for this purpose

why weird nops

could fill with *anything* — unreachable

some platforms: filled with crashing instructions

why not in example? assembler just told to align instruction

- not told previous instruction was jump/ret/etc. ...
- and assembler doesn't bother checking

probably better for CPU to fill with some instruction; Intel manual:

> "Placing data immediately following an indirect branch can cause performance problems. If the data consists of all zeros, it looks like a long stream of ADDs to memory destinations, and this can cause resource conflicts..."

other empty space

unused dynamic linking structure

unused space between segments

unused debugging/symbol table information?

unused header space

- file offsets of segments can be in middle of header
- loader doesn't care what segments "mean"

other empty space

unused dynamic linking structure

unused space between segments

unused debugging/symbol table information?

unused header space

> file offsets of segments can be in middle of header
> loader doesn't care what segments "mean"

dynamic linking cavity

`.dynamic` 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 f0fffff6f 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  ................
```

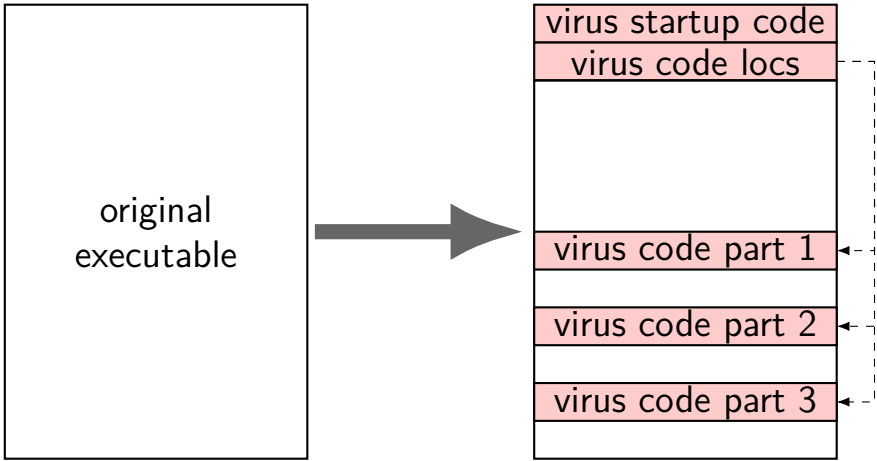
is there enough empty space?

cavities look awfully small

really small viruses?

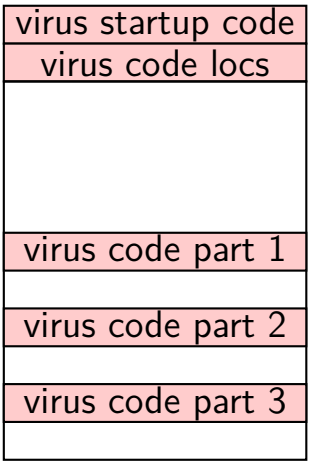
solution: chain cavities tgoether

case study: CIH (1)

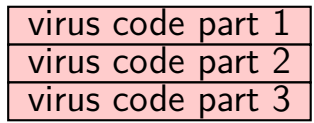


case study: CIH (2)

on disk:



in memory:



CIH cavities

gaps between sections

- common Windows linker aligned sections
  (align = start on address multiple of N, e.g. 4096)

reassembling code avoids worrying about splitting instructions

segment rounding

objdump -x /bin/ls:

```
   LOAD off    0x0000000000004000 vaddr 0x0000000000004000 paddr 0x0000000000004000
        filesz 0x0000000000013091 memsz 0x0000000000013091 flags r-x
   LOAD off    0x0000000000018000 vaddr 0x0000000000018000 paddr 0x0000000000018000
        filesz 0x0000000000007458 memsz 0x0000000000007458 flags r--
```

running /bin/ls in gdb:

```
(gdb) info proc map
process 1178818
Mapped address spaces:
          Start Addr           End Addr       Size     Offset  Perms  objfile
      0x555555554000     0x555555558000     0x4000        0x0  r--p   /usr/bin/ls
      0x555555558000     0x55555556c000    0x14000     0x4000  r-xp   /usr/bin/ls
      0x55555556c000     0x555555574000     0x8000    0x18000  r--p   /usr/bin/ls
....
```

requested 0x13091 bytes, loaded 0x14000

x86-64 Linux: OS allocates only in one page = 4096-byte chunks

segment rounding

objdump -x /bin/ls:

```
   LOAD off    0x0000000000004000 vaddr 0x0000000000004000 paddr 0x0000000000004000
        filesz 0x0000000000013091 memsz 0x0000000000013091 flags r-x
   LOAD off    0x0000000000018000 vaddr 0x0000000000018000 paddr 0x0000000000018000
        filesz 0x0000000000007458 memsz 0x0000000000007458 flags r--
```

running /bin/ls in gdb:

```
(gdb) info proc map
process 1178818
Mapped address spaces:
          Start Addr           End Addr       Size     Offset  Perms  objfile
      0x555555554000     0x555555558000     0x4000        0x0  r--p   /usr/bin/ls
      0x555555558000     0x55555556c000    0x14000     0x4000  r-xp   /usr/bin/ls
      0x55555556c000     0x555555574000     0x8000    0x18000  r--p   /usr/bin/ls
....
```

requested 0x13091 bytes, loaded 0x14000

x86-64 Linux: OS allocates only in one page = 4096-byte chunks

segment rounding

objdump -x /bin/ls:

```
   LOAD off    0x0000000000004000 vaddr 0x0000000000004000 paddr 0x0000000000004000
        filesz 0x0000000000013091 memsz 0x0000000000013091 flags r-x
   LOAD off    0x0000000000018000 vaddr 0x0000000000018000 paddr 0x0000000000018000
        filesz 0x0000000000007458 memsz 0x0000000000007458 flags r--
```

running /bin/ls in gdb:

```
(gdb) info proc map
process 1178818
Mapped address spaces:
          Start Addr           End Addr       Size     Offset  Perms  objfile
      0x555555554000     0x555555558000     0x4000        0x0  r--p   /usr/bin/ls
      0x555555558000     0x55555556c000    0x14000     0x4000  r-xp   /usr/bin/ls
      0x55555556c000     0x555555574000     0x8000    0x18000  r--p   /usr/bin/ls
....
```

requested 0x13091 bytes, loaded 0x14000

x86-64 Linux: OS allocates only in one page = 4096-byte chunks

where to put code: options

one *or more* of:

replacing executable code

after executable code (Vienna)

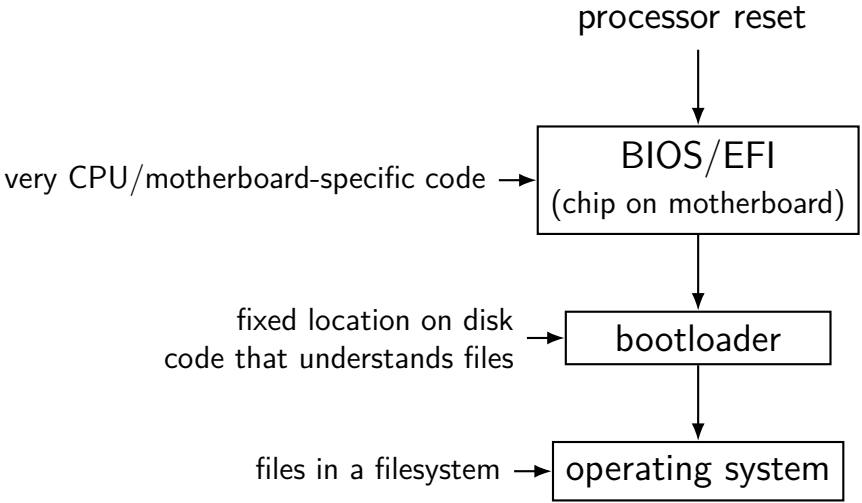
in unused executable code

inside OS code

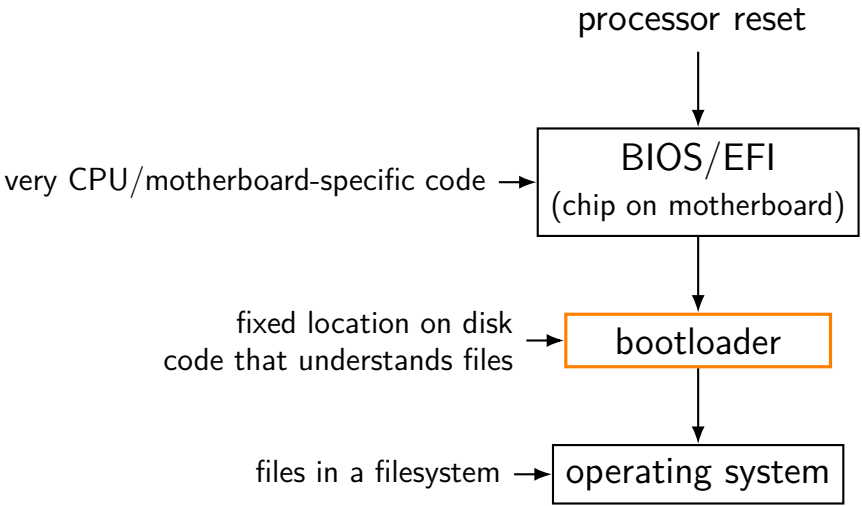
in memory

replace existing code

boot process



boot process



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)

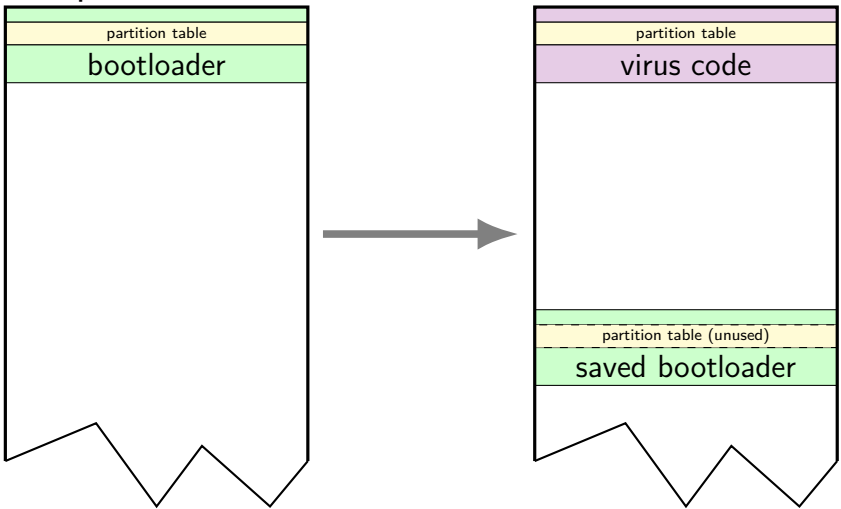
code in BIOS to copy bootloader into RAM, start running

bootloader responsible for disk I/O etc.

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

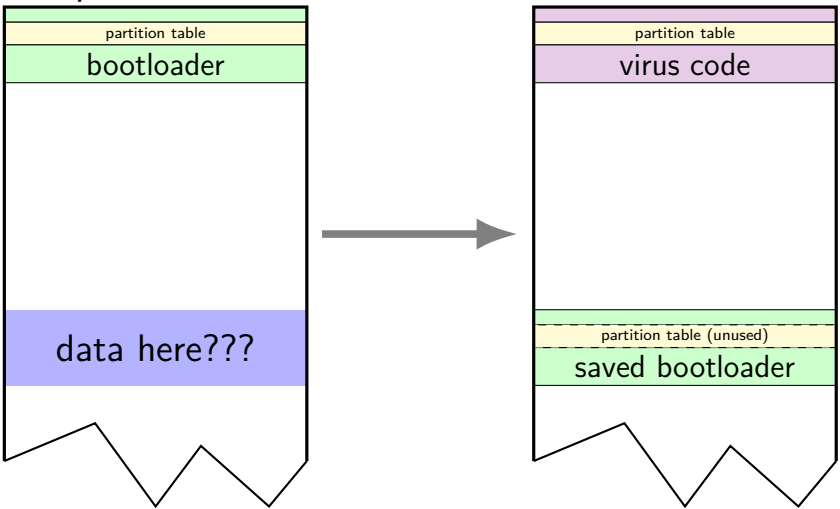
bootloader viruses

example: Stoned



bootloader viruses

example: Stoned



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)

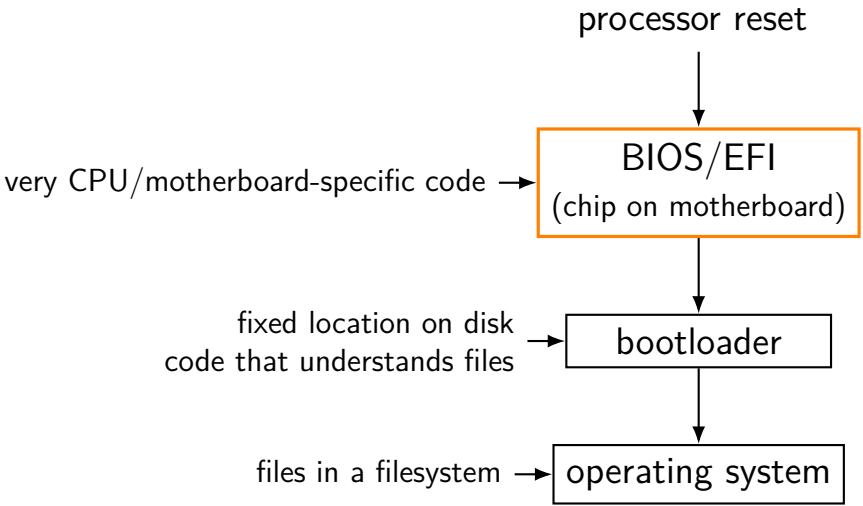
like BIOS:

- 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

boot process



BIOS/UEFI implants

infrequent

BIOS/UEFI code is *very non-portable*

BIOS/UEFI update may require physical access

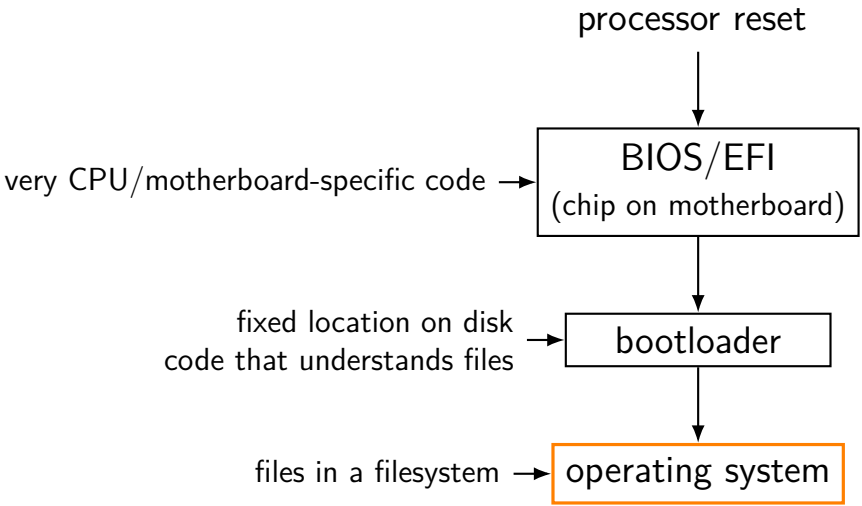
BIOS/UEFI code may require cryptographic signatures

...but *very hard to remove* — "persist" other malware

reports of BIOS/UEFI-infecting "implants"

- sold by Hacking Team (Milan-based malware company)
- listed in leaked NSA Tailored Access Group catalog

boot process



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

memory residence

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

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

run anyways?

add code at start of program (Vienna)

plus restore replaced code after running malware code

return with padding after it:

```
404a01:       c3                      retq
404a02:       0f 1f 40 00             nopl   0x0(%rax)
              replace with
404a01:       e9 XX XX XX XX          jmpq    YYYYYYY
```

plus return after running malware code

any random place in program?

just not in the *middle of instruction*

and replace orignal code after running malware code

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 (1)

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

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

use to identify when e8 refers to real function

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

really finding function calls (2)

x86-64 assembly seen a lot of ENDBR64 (hex `f3 0f 1e fa`)

marker for valid locations to jump to

> intention: part of possible defense against return-oriented-programming-style attacks (we'll talk about what this means later)

likely only seen at beginning of functions, switch statement cases, etc.

run anyways?

add code at start of program (Vienna)

plus restore replaced code after running malware code

return with padding after it:

```
404a01:       c3                      retq
404a02:       0f 1f 40 00             nopl   0x0(%rax)
              replace with
404a01:       e9 XX XX XX XX          jmpq    YYYYYYY
```

plus return after running malware code

any random place in program?

just not in the *middle of instruction*

and *replace orignal code after running malware code*

restoring replaced code?

Vienna: just write to memory addres

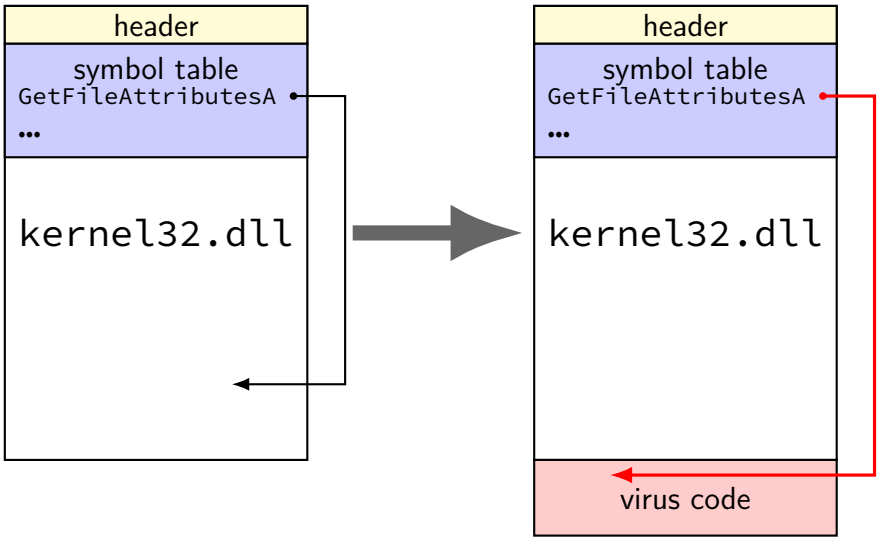
modern OS: segfault/general protection fault

- code loaded read-only

easy solution: make library call to make it writable

- Linux: `mprotect`
- functionality exists to, e.g., allow compiling code at runtime

infecting shared libraries via relocations



other dynamic-linking-based infections

could also modify

relocations on executable

- this isn't the global offset table entry for `puts`,
  it's the one for `evilvirus`

list of needed libraries?

- the C standard library and virus.so
- 'init' code run when shared libraries loaded

stubs and calls to stub

- very regular and easy to locate

summary

how to hide:

- separate executable
- append
- existing “unused” space
- append + compression

how to run:

- change entry point (start address)
- change calls
- change beginning of function
- change dynamic-linking-related pointers
- arrange to run as part of OS