virus 2

last time

Vienna virus code fixups

- relocation: change 'data' address in appended code
- relocation: produce jump at beginning
- fixup: replace jump at beginning with normal program

real binary formats: example of ELF

- multiple segments loaded
- header information

virus code placement strategies

- replacing executables, temporary files
- appending

scheduling note

no lecture Wednesday (break day)

quiz due next week released by tomorrow

RE assignment still due Friday

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)

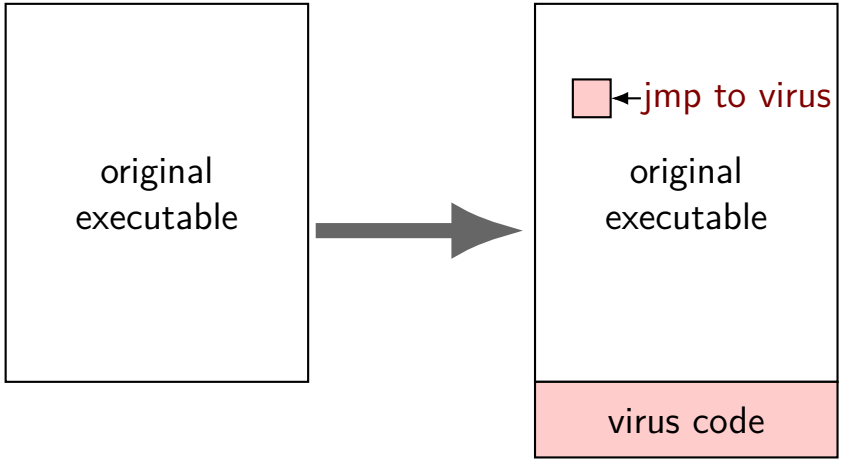
in unused executable code

inside OS code

in memory

replace existing code

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)

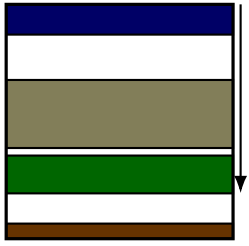
option 2: update last segment of program header

change its size

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

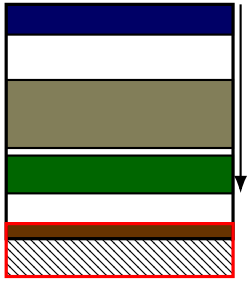
appending to ELF file (v1)

```
Program Header:
...
LOAD off    0x00000 vaddr 0x10000 paddr 0x10000 align 2**12
     filesz 0x005f8 memsz 0x005f8 flags r--
LOAD off    0x01000 vaddr 0x11000 paddr 0x11000 align 2**12
     filesz 0x00ef5 memsz 0x00ef5 flags r-x
LOAD off    0x02000 vaddr 0x12000 paddr 0x12000 align 2**12
     filesz 0x00858 memsz 0x00858 flags r--
LOAD off    0x02db8 vaddr 0x13db8 paddr 0x13db8 align 2**12
     filesz 0x00258 memsz 0x00360 flags rw-
```



appending to ELF file (v1)

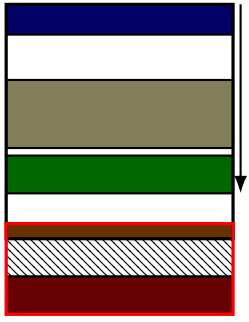
```
Program Header:
...
    LOAD off    0x00000 vaddr 0x10000 paddr 0x10000 align 2**12
         filesz 0x005f8 memsz 0x005f8 flags r--
    LOAD off    0x01000 vaddr 0x11000 paddr 0x11000 align 2**12
         filesz 0x00ef5 memsz 0x00ef5 flags r-x
    LOAD off    0x02000 vaddr 0x12000 paddr 0x12000 align 2**12
         filesz 0x00858 memsz 0x00858 flags r--
    LOAD off    0x02db8 vaddr 0x13db8 paddr 0x13db8 align 2**12
         filesz 0x00258 0x00860 memsz 0x00860 flags rw-
```



more zeroes

appending to ELF file (v1)

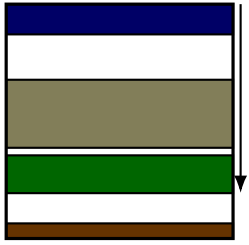
```
Program Header:
...
    LOAD off    0x00000 vaddr 0x10000 paddr 0x10000 align 2**12
         filesz 0x005f8 memsz 0x005f8 flags r--
    LOAD off    0x01000 vaddr 0x11000 paddr 0x11000 align 2**12
         filesz 0x00ef5 memsz 0x00ef5 flags r-x
    LOAD off    0x02000 vaddr 0x12000 paddr 0x12000 align 2**12
         filesz 0x00858 memsz 0x00858 flags r--
    LOAD off    0x02db8 vaddr 0x13db8 paddr 0x13db8 align 2**12
         filesz 0x008600x01058 memsz 0x008600x01058 flags rw-rwx
```



more zeroes
virus code

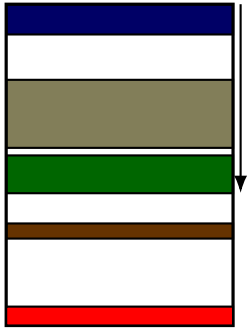
appending to ELF file (v2)

```
Program Header:
...
LOAD off    0x00000 vaddr 0x10000 paddr 0x10000 align 2**12
     filesz 0x005f8 memsz 0x005f8 flags r--
LOAD off    0x01000 vaddr 0x11000 paddr 0x11000 align 2**12
     filesz 0x00ef5 memsz 0x00ef5 flags r-x
LOAD off    0x02000 vaddr 0x12000 paddr 0x12000 align 2**12
     filesz 0x00858 memsz 0x00858 flags r--
LOAD off    0x02db8 vaddr 0x13db8 paddr 0x13db8 align 2**12
     filesz 0x00258 memsz 0x00360 flags rw-
.
.
```



appending to ELF file (v2)

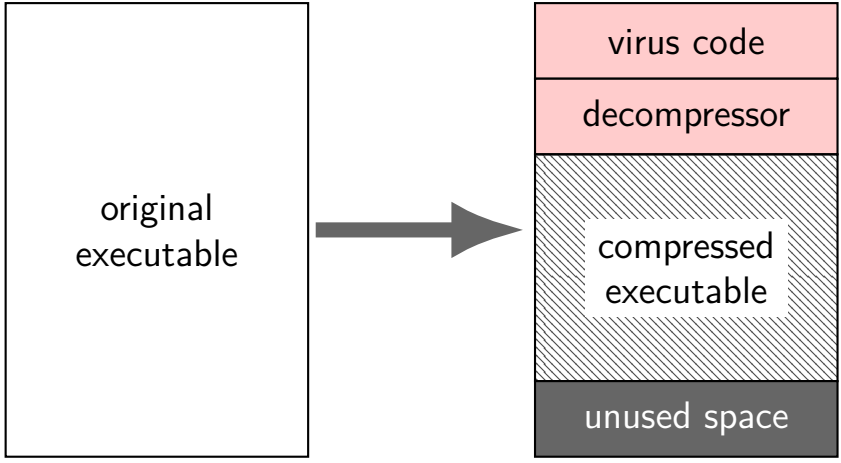
```
Program Header:
...
LOAD off    0x00000 vaddr 0x10000 paddr 0x10000 align 2**12
     filesz 0x005f8 memsz 0x005f8 flags r--
LOAD off    0x01000 vaddr 0x11000 paddr 0x11000 align 2**12
     filesz 0x00ef5 memsz 0x00ef5 flags r-x
LOAD off    0x02000 vaddr 0x12000 paddr 0x12000 align 2**12
     filesz 0x00858 memsz 0x00858 flags r--
LOAD off    0x02db8 vaddr 0x13db8 paddr 0x13db8 align 2**12
     filesz 0x00258 memsz 0x00360 flags rw-
LOAD off    0x04000 vaddr 0x20000 paddr 0x20000 align 2**12
     filesz 0x00400 memsz 0x00400 flags rwx
```



virus code

compressing viruses

file too big? how about compression



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

other empty space

unused space between segments/sections

unreachable bytes added for instruction alignment

unused header space

unused debugging information?

gaps between segments

```
LOAD off    0x0000000000000000 ...
     filesz 0x00000000000036a8 ...
LOAD off    0x0000000000004000 ...
     filesz 0x0000000000013581 ...
```

first LOAD loads `0x0-0x36a8`

second LOAD loads `0x4000-0x17358`

- starts at multiple of `0x1000` b/c that's what OS can handle

`0x4000-0x36a8` bytes of file not used

gaps between segments

```
LOAD off    0x0000000000000000 ...
     filesz 0x00000000000036a8 ...
LOAD off    0x0000000000004000 ...
     filesz 0x0000000000013581 ...
```

first LOAD loads `0x0-0x36a8`

second LOAD loads `0x4000-0x17358`

- starts at multiple of `0x1000` b/c that's what OS can handle

`0x4000-0x36a8` bytes of file not used

- (and loaded on Linux, because Linux rounds up filesz)

unused code case study: /bin/ls

unreachable no-ops!

```
...
  403788:        e9 59 0c 00 00              jmpq   4043e6 <__sprintf_chk@plt+0x1a
  40378d:        0f 1f 00                    nopl   (%rax)
  403790:        ba 05 00 00 00              mov    $0x5,%edx
...
  403ab9:        eb 4d                       jmp    403b08 <__sprintf_chk@plt+0x11
  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

nops allow compiler/assembler to align without checking reachability

nops better for disassembly

> Intel manual recommends form of nop for different lengths

possibly better for CPU

> "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..."

unused header space

often executable metadata has empty space

debugging info and info not strictly needed to run program

linker allocated space for N fields, used less

unused header space

often executable metadata has empty space

debugging info and info not strictly needed to run program

linker allocated space for N fields, used less

dynamic linking cavity

Linux example from my desktop

`.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 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  ................
```

exercise: cavity finding difficulty?

exercise: which is least, most difficult kind of cavity to *write code* to find

A. gaps between segments

B. space added for alignment between instructions

C. unused space in headers

exercise: scheme for finding alignment space

example unused space:

```
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
```

suppose a virus scanned the executable for this pattern of bytes (c3 0f 1f ...)
to find a cavity

for compiler generated code, which are likely causes of false positive?

A. `c3` might be part of non-ret instruction
B. `ret` might return to byte immediately after (0x404a02 in example)
C. some other instruction might jump to one of the bytes after
D. the entire sequence could be part of the program's data
(if the virus didn't check that it was in the text section)

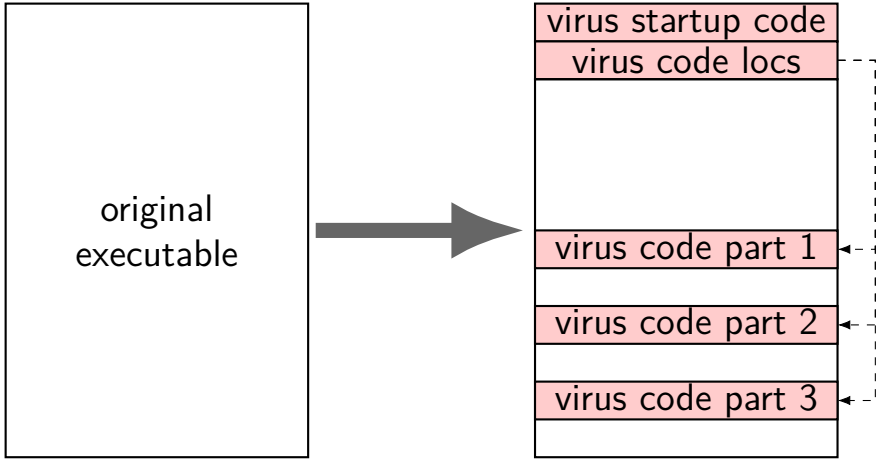
is there enough empty space?

cavities look awfully small

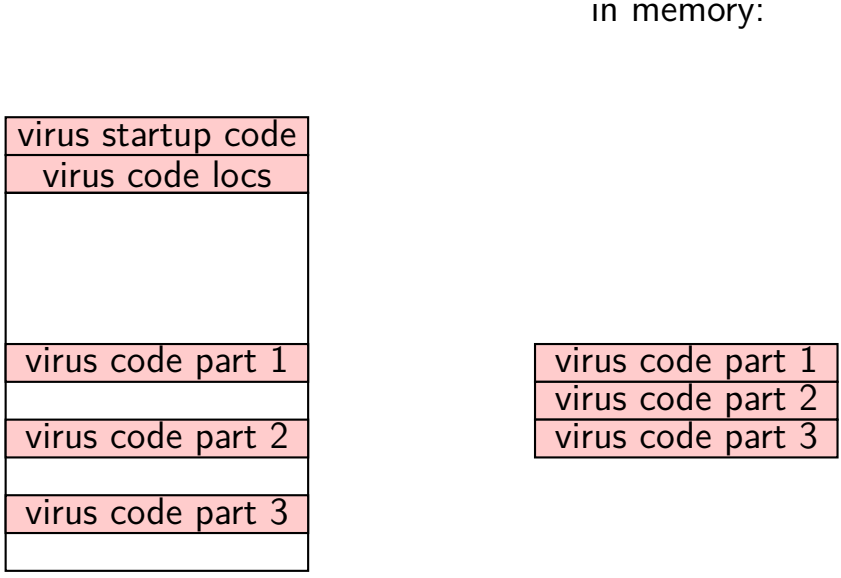
really small viruses?

solution: chain cavities tgoether

case study: CIH (1)



case study: CIH (2)



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

detecting cavity hiding?

thought for future: detecting cavity hiding?

example: filling empty space in header — what should we look for?

detecting cavity hiding?

thought for future: detecting cavity hiding?

example: filling empty space in header — what should we look for?

- stuff that looks like machine code?
- validate matches executable data structure?

detecting cavity hiding?

thought for future: detecting cavity hiding?

example: filling empty space in header — what should we look for?

- stuff that looks like machine code?
- validate matches executable data structure?

follow-up: possibility for false positives?

detecting cavity hiding?

thought for future: detecting cavity hiding?

example: filling empty space in header — what should we look for?

- stuff that looks like machine code?
- validate matches executable data structure?

follow-up: possibility for false positives?

- additional executable format features?
- values that happen to be same as machine code?

where to put code: options

one *or more* of:

replacing executable code

after executable code (Vienna)

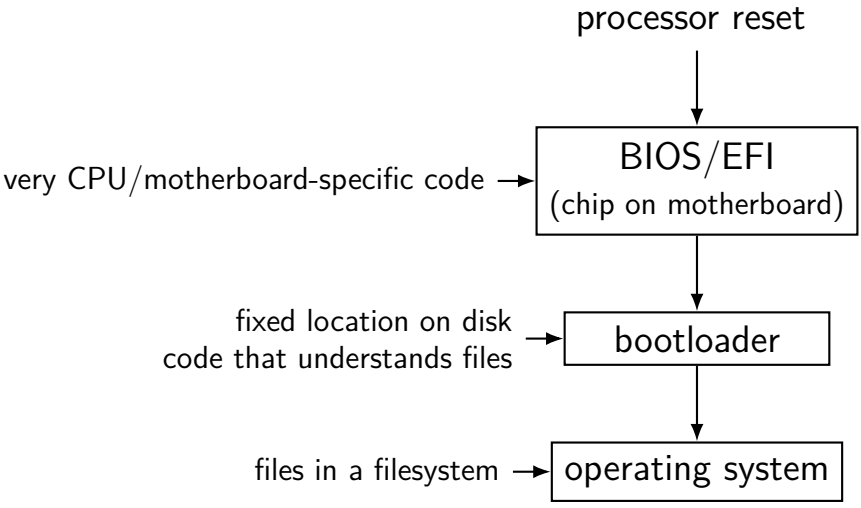
in unused executable code

inside OS code

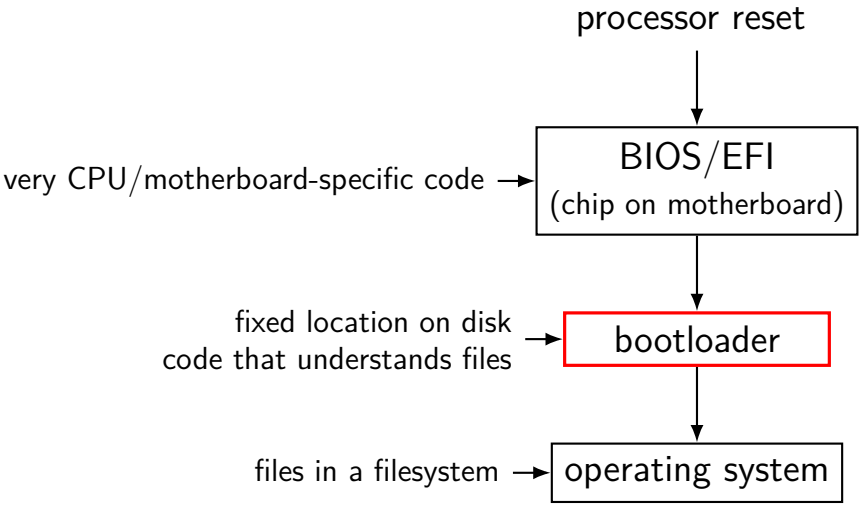
in memory

replace existing code

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)

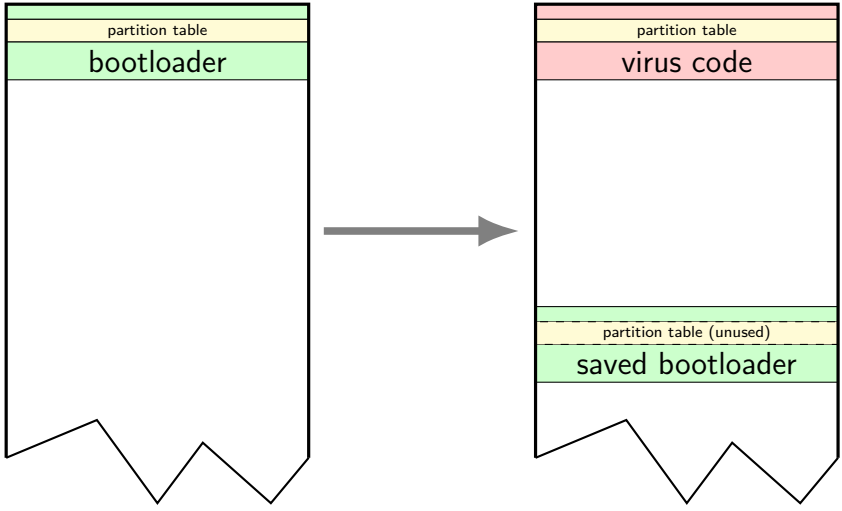
code in BIOS to copy bootloader into RAM, start running

bootloader responsible for disk I/O etc.

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

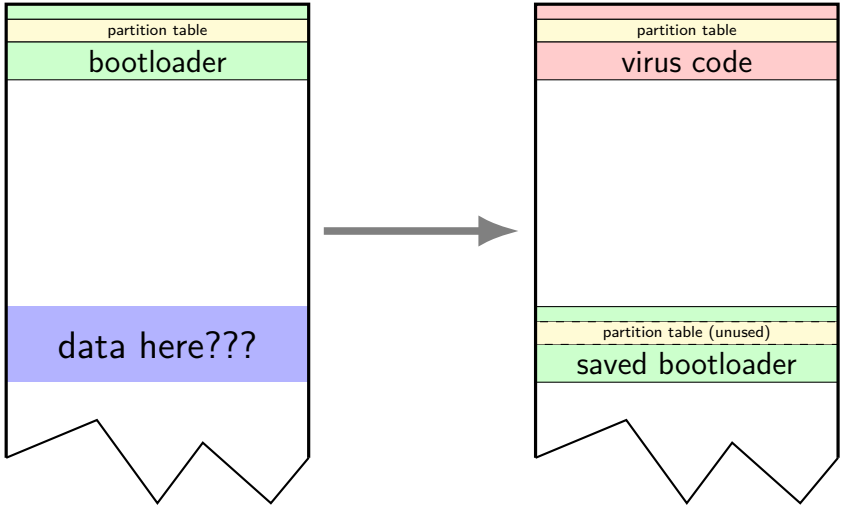
bootloader viruses

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)

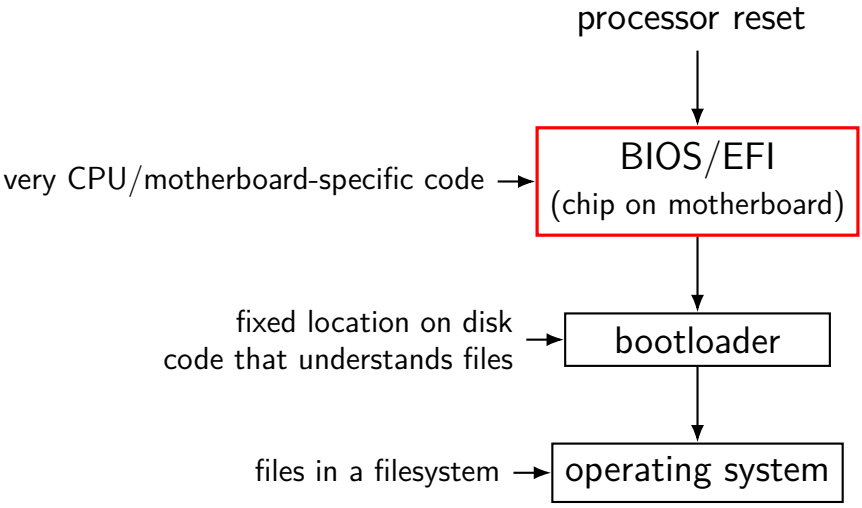
like BIOS:

- 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

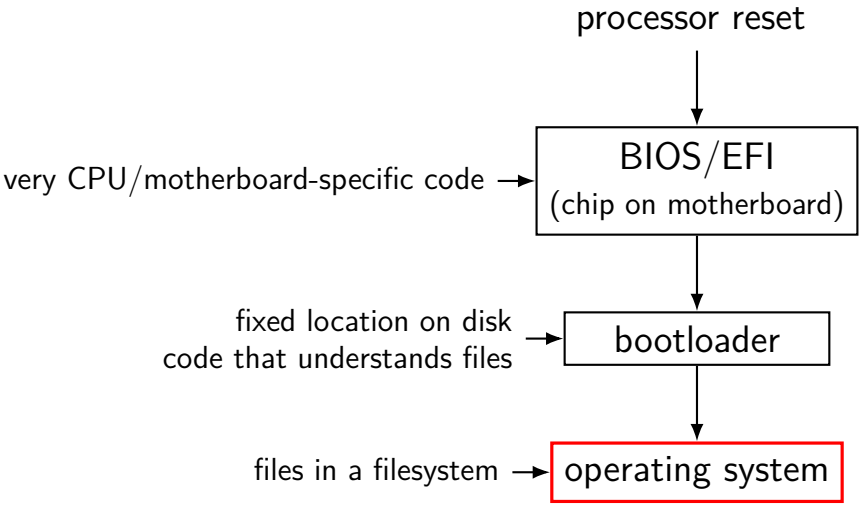
BIOS/UEFI code may require cryptographic signatures

...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

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

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

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

virus: easiest code to find?

what should be easiest/hardest to identify without many false positives?

A. replaced executable

B. appended to executable w/o compression

C. replaced cavities

D. replaced bootloader

E. new automatically started system program

virus: easiest code to find pro/con

A. replaced executable
- \+ executable size size change
- \+ could detect write+launch executable
- \- programs running programs is common (e.g. start browser)
- \- confused with software updater

B. appended to executable w/o compression
- \+ executable size change
- \+ unusual segment/section layout
- \- confused with unusual compilers

C. replaced cavities
- \+ code outside of segments/sections if virus not careful
- \+ machine code patterns not seen by compilers
- \- need to search entire exe file
- \- confused with unusual compilers?

D. replaced bootloader

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

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)

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

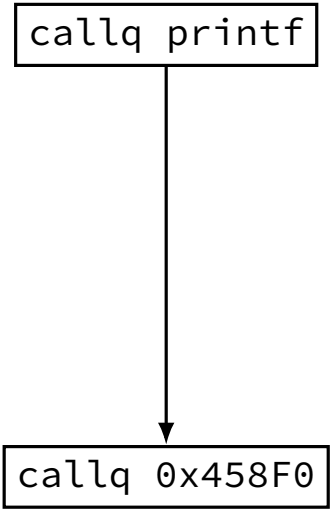
modern OS: segfault/general protection fault

- 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

linking



static v. dynamic linking

static linking — linking to create executable

dynamic linking — linking when executable is run

static v. dynamic linking

static linking — linking to create executable

dynamic linking — linking when executable is run

conceptually: no difference in how they work

reality — very different mechanisms

- extra indirection to avoid modifying much code at runtime
- change *lookup table* instead of code for library locations

interlude: strace

`strace` — system call tracer

- on Linux, some other Unices
- OS X approx. equivalent: `dtruss`
- Windows approx. equivalent: Process Monitor

indicates what system calls (operating system services) used by a program

statically linked hello.exe

`gcc -static -o hello-static.exe hello.s`

`strace ./hello-static.exe`:

```
execve("./hello-static.exe", ["./hello-static.exe"], [/* 46 vars */]) = 0
uname(sysname="Linux", nodename="reiss-lenovo", ...) = 0
brk(NULL)                               = 0x20a5000
brk(0x20a61c0)                          = 0x20a61c0
arch_prctl(ARCH_SET_FS, 0x20a5880)      = 0
readlink("/proc/self/exe", "/home/cr4bd/spring2017/cs4630/sl"..., 4096) = 62
brk(0x20c71c0)                          = 0x20c71c0
brk(0x20c8000)                          = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
fstat(1, st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...) = 0
write(1, "Hello, World!\n", 14)         = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

statically linked hello.exe

`gcc -static -o hello-static.exe hello.s`

`strace ./hello-static.exe`:

```
execve("./hello-static.exe", ["./hello-static.exe"], [/* 46 vars */]) = 0
uname(sysname="Linux", nodename="reiss-lenovo", ...) = 0
brk(NULL)                               = 0x20a5000
brk(0x20a61c0)                          = 0x20a61c0
arch_prctl(ARCH_SET_FS, 0x20a5880)      = 0
readlink("/proc/self/exe", "/home/cr4bd/spring2017/cs4630/sl"..., 4096) = 62
brk(0x20c71c0)                          = 0x20c71c0
brk(0x20c8000)                          = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
fstat(1, st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...) = 0
write(1, "Hello, World!\n", 14)         = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

standard library startup

statically linked hello.exe

`gcc -static -o hello-static.exe hello.s`

`strace ./hello-static.exe`:

```
execve("./hello-static.exe", ["./hello-static.exe"], [/* 46 vars */]) = 0
uname(sysname="Linux", nodename="reiss-lenovo", ...) = 0
brk(NULL)                                = 0x20a5000
brk(0x20a61c0)                           = 0x20a61c0
arch_prctl(ARCH_SET_FS, 0x20a5880)       = 0
readlink("/proc/self/exe", "/home/cr4bd/spring2017/cs4630/sl"..., 4096) = 62
brk(0x20c71c0)                           = 0x20c71c0
brk(0x20c8000)                           = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK)       = -1 ENOENT (No such file or directory)
fstat(1, st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...) = 0
write(1, "Hello, World!\n", 14)          = 14
exit_group(14)                           = ?
+++ exited with 14 +++
```

memory allocation

statically linked hello.exe

`gcc -static -o hello-static.exe hello.s`

`strace ./hello-static.exe`:

```
execve("./hello-static.exe", ["./hello-static.exe"], [/* 46 vars */]) = 0
uname(sysname="Linux", nodename="reiss-lenovo", ...) = 0
brk(NULL)                               = 0x20a5000
brk(0x20a61c0)                          = 0x20a61c0
arch_prctl(ARCH_SET_FS, 0x20a5880)      = 0
readlink("/proc/self/exe", "/home/cr4bd/spring2017/cs4630/sl"..., 4096) = 62
brk(0x20c71c0)                          = 0x20c71c0
brk(0x20c8000)                          = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
fstat(1, st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...) = 0
write(1, "Hello, World!\n", 14)         = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

implementation of puts

statically linked hello.exe

`gcc -static -o hello-static.exe hello.s`

`strace ./hello-static.exe`:

```
execve("./hello-static.exe", ["./hello-static.exe"], [/* 46 vars */]) = 0
uname(sysname="Linux", nodename="reiss-lenovo", ...) = 0
brk(NULL)                               = 0x20a5000
brk(0x20a61c0)                          = 0x20a61c0
arch_prctl(ARCH_SET_FS, 0x20a5880)      = 0
readlink("/proc/self/exe", "/home/cr4bd/spring2017/cs4630/sl"..., 4096) = 62
brk(0x20c71c0)                          = 0x20c71c0
brk(0x20c8000)                          = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
fstat(1, st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...) = 0
write(1, "Hello, World!\n", 14)         = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

standard library shutdown

dynamically linked hello.exe

`gcc -o hello.exe hello.s`

`strace ./hello.exe`:

```
execve("./hello.exe", ["./hello.exe"], [/* 46 vars */]) = 0
...
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fdfeeb39000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, st_mode=S_IFREG|0644, st_size=137808, ...) = 0
...
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"..., 832) = 832
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
mmap(0x7fdfee90c000, 24576, PROT_READ|PROT_WRITE, ..., 3, 0x1bf000) = 0x7fdfee90c000
mmap(0x7fdfee912000, 14752, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x7fdfee912000
close(3)                                = 0
...
write(1, "Hello, World!\n", 14)         = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

dynamically linked hello.exe

`gcc -o hello.exe hello.s`

`strace ./hello.exe`:

```
execve("./hello.exe", ["./hello.exe"], [/* 46 vars */]) = 0
...
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fdfeeb39000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, st_mode=S_IFREG|0644, st_size=137808, ...) = 0
...
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"..., 832) = 832
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
mmap(0x7fdfee90c000,                                    x7fdfee90c000
mmap(0x7fdfee912000,                                    e912000
close(3)                                = 0
...
write(1, "Hello, World!\n", 14)         = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

the standard C library (includes puts)

dynamically linked hello.exe

`gcc -o hello.exe hello.s`

`strace ./hello.exe`:

```
execve("./hello.exe", ["./hello.exe"], [/* 46 vars */]) = 0
...
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fdfeeb39000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, st_mode=S_IFREG|0644, st_size=137808, ...) = 0
...
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"..., 832) = 832
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
mmap(0x7fdfee90c000,                                          0x7fdfee90c000
mmap(0x7fdfee912000,                                          ee912000
close(3)                                = 0
...
write(1, "Hello, World!\n", 14)         = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

memory allocation (different method)

dynamically linked hello.exe

`gcc -o hello.exe hello.s`

`strace ./hello.exe`:

```
execve("./hello.exe", ["./hello.exe"], [/* 46 vars */]) = 0
...
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fdfeeb39000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, st_mode=S_IFREG|0644, st_size=137808, ...) = 0
...
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"..., 832) = 832
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
mmap(0x7fdfee90c000, 2457                                    0) = 0x7fdfee90c000
mmap(0x7fdfee912000, 1475                                    x7fdfee912000
close(3)                                = 0
...
write(1, "Hello, World!\n", 14)          = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

read standard C library header

dynamically linked hello.exe

`gcc -o hello.exe hello.s`

`strace ./hello.exe`:

```
execve("./hello.exe", ["./hello.exe"], [/* 46 vars */]) = 0
...
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fdfeeb39000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, st_mode=S_IFREG|0644, st_size=137808, ...) = 0
...
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"..., 832) = 832
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
mmap(0x7fdfee90c000                                                  7fdfee90c000
mmap(0x7fdfee912000                                                  912000
close(3)                                = 0
...
write(1, "Hello, World!\n", 14)          = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

load standard C library (3 = opened file)

dynamically linked hello.exe

`gcc -o hello.exe hello.s`

`strace ./hello.exe`:

```
execve("./hello.exe", ["./hello.exe"], [/* 46 vars */]) = 0
...
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fdfeeb39000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, st_mode=S_IFREG|0644, st_size=137808, ...) = 0
...
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"..., 832) = 832
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
mmap(0x7fdfee                                                    e90c000
mmap(0x7fdfee                                                    0
close(3)                                = 0
...
write(1, "Hello, World!\n", 14)          = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

allocate zero-initialized data segment for C library

where's the linker

Where's the code that calls `open("...libc.so.6")`?

Could check `hello.exe` — it's not there!

where's the linker

Where's the code that calls `open("...libc.so.6")`?

Could check `hello.exe` — it's not there!

instead: "interpreter" `/lib64/ld-linux-x86-64.so.2`

on Linux: contains loading code instead of core OS

- OS loads it instead of program

objdump — the interpreter

excerpt from `objdump -sx hello.exe`:

```
Program Header:
...
  INTERP off    0x0000238 vaddr 0x0400318 paddr 0x0400238 align 2**0
         filesz 0x000001c memsz 0x000001c flags r--
...
Contents of section .interp:
 400318 2f6c6962 36342f6c 642d6c69 6e75782d  /lib64/ld-linux-
 400328 7838362d 36342e73 6f2e3200           x86-64.so.2.
```

dynamic linking: what to load? (1)

excerpt from `objdump -sx hello.exe`:

```
Program Header:
...
 DYNAMIC off    0x0000000000002e20 vaddr 0x0000000000403e20 paddr 0x0000000000403e2
         filesz 0x00000000000001d0 memsz 0x00000000000001d0 flags rw-
...
Dynamic Section:
  NEEDED               libc.so.6
  INIT                 0x0000000000401000
  ...
  STRTAB               0x0000000000400420
  ...
```

program header: identifies where dynamic linking info is

dynamic linking info: array of key-value pairs

- needed libraries
- constructor locations ('INIT')
- string table location
- ...

backup slides

dynamic linking: what to load? (2)

excerpt from `objdump -sx hello.exe`:

```
Program Header:
...
 DYNAMIC off    0x0000000000002e20 vaddr 0x0000000000403e20 paddr 0x0000000000403e2
         filesz 0x00000000000001d0 memsz 0x00000000000001d0 flags rw-
...
Dynamic Section:
  NEEDED               libc.so.6
  INIT                 0x0000000000401000
  ...
  STRTAB               0x0000000000400420
  ...
...
 403e20 01000000 00000000 01000000 00000000  ................
 403e30 0c000000 00000000 00104000 00000000  ..............@.
```

type 0x1 = "DT_NEEDED" (from ELF manual)

value 0x1 = string table entry 1

type 0xC = "DT_INIT"

value 0x401000

dynamic linking: what to load? (2)

excerpt from `objdump -sx hello.exe`:

```
Program Header:
...
 DYNAMIC off    0x0000000000002e20 vaddr 0x0000000000403e20 paddr 0x0000000000403e2
         filesz 0x00000000000001d0 memsz 0x00000000000001d0 flags rw-
...
Dynamic Section:
  NEEDED               libc.so.6
  INIT                 0x0000000000401000
  ...
  STRTAB               0x0000000000400420
  ...
...
 403e20 01000000 00000000 01000000 00000000  ................
 403e30 0c000000 00000000 00104000 00000000  ..............@.
```

type 0x1 = "DT_NEEDED" (from ELF manual)

value 0x1 = string table entry 1

type 0xC = "DT_INIT"

value 0x401000

dynamic linking: what to load? (2)

excerpt from `objdump -sx hello.exe`:

```
Program Header:
...
 DYNAMIC off    0x0000000000002e20 vaddr 0x0000000000403e20 paddr 0x0000000000403e2
         filesz 0x00000000000001d0 memsz 0x00000000000001d0 flags rw-
...
Dynamic Section:
  NEEDED               libc.so.6
  INIT                 0x0000000000401000
  ...
  STRTAB               0x0000000000400420
  ...
...
 403e20 01000000 00000000 01000000 00000000  ................
 403e30 0c000000 00000000 00104000 00000000  ..............@.
```

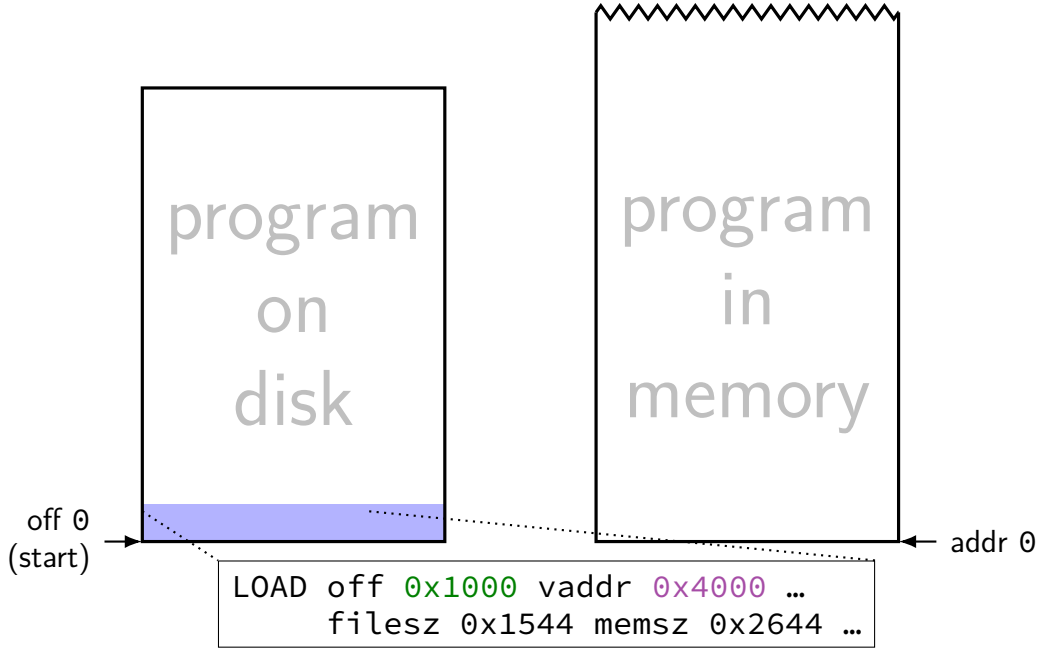
type 0x1 = "DT_NEEDED" (from ELF manual)

value 0x1 = string table entry 1

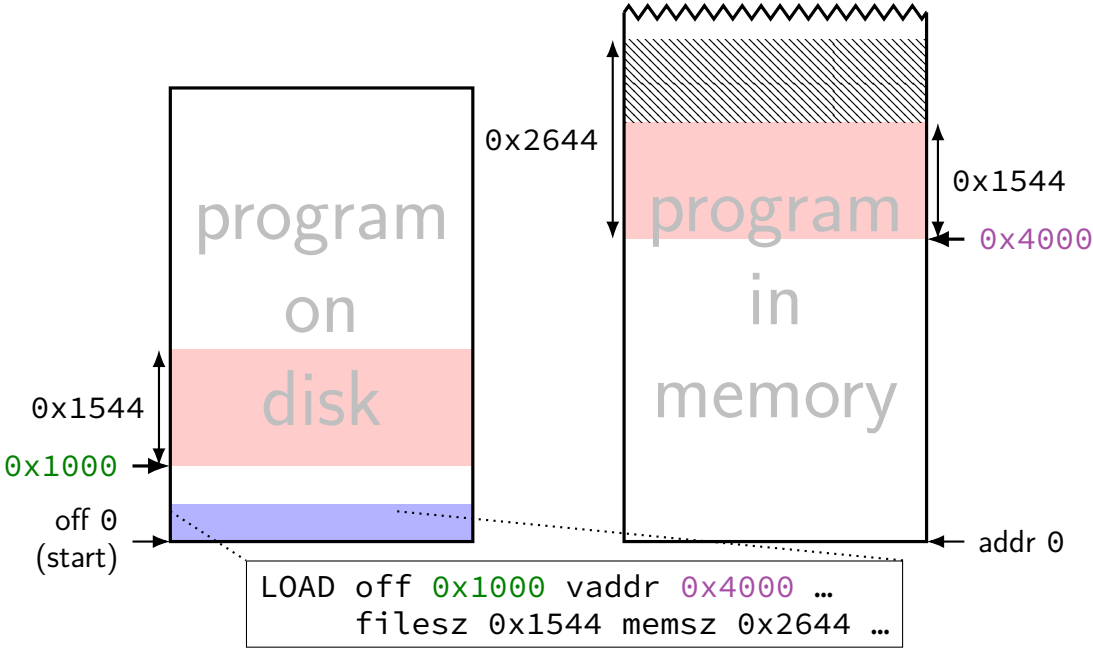
type 0xC = "DT_INIT"

value 0x401000

ELF LOAD


program
on
disk
program
in
memory
off 0
(start)
addr 0
LOAD off 0x1000 vaddr 0x4000 …
filesz 0x1544 memsz 0x2644 …

ELF LOAD


program
on
disk
0x1544
0x1000
off 0
(start)
0x2644
program
in
memory
0x1544
0x4000
addr 0
LOAD off 0x1000 vaddr 0x4000 …
filesz 0x1544 memsz 0x2644 …

hello.s

```
.data
string: .asciz "Hello, World!"
.text
.globl main
main:
    movq $string, %rdi
    call puts
    ret
```

hello.o (pre-static or dynamic linking)

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

hello.o (pre-static or dynamic linking)

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

undefined symbol: look for `puts` elsewhere

hello.o (pre-static or dynamic linking)

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

insert address of puts, format for `call`

hello.o (pre-static or dynamic linking)

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

insert address of string, format for `movq`

hello.o (pre-static or dynamic linking)

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

different ways to represent address
32S — signed 32-bit value
PC32 — 32-bit difference from current address

hello.o (pre-static or dynamic linking)

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

g: global — used by other files
l: local

hello.o (pre-static or dynamic linking)

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

`.text` segment beginning plus 0 bytes

hello.exe

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

hello.exe

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

undefined symbol: look for `puts` elsewhere

hello.exe

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

insert address of puts, format for `call`

hello.exe

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

insert address of string, format for `movq`

hello.exe

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

different ways to represent address
32S — signed 32-bit value
PC32 — 32-bit difference from current address

hello.exe

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

g: global — used by other files
l: local

hello.exe

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

`.text` segment beginning plus 0 bytes