ASLR / NX / Bounds Checking

last time

stack canaries

- less-compatible alternative: shadow stacks

page-level protection

- RELRO — protect global offset table
- guard pages around memory allocations/etc.

start ASLR

- choose random addresses
- ideally attacker never learns addresses
  - except overflows can leak them

logistical note: FORMAT

deadline Saturday

- since executable was not linked correctly on time

format string exploit

- sufficient to overwrite `defaultLetterGrade` variable

Linux stack randomization (x86-64)

1. choose random number between `0` and `0x3F FFFF`
2. stack starts at `0x7FFF FFFF FFFF` - *random number* $\times$ `0x1000`

    randomization disabled? *random number* $= 0$

16 GB range!

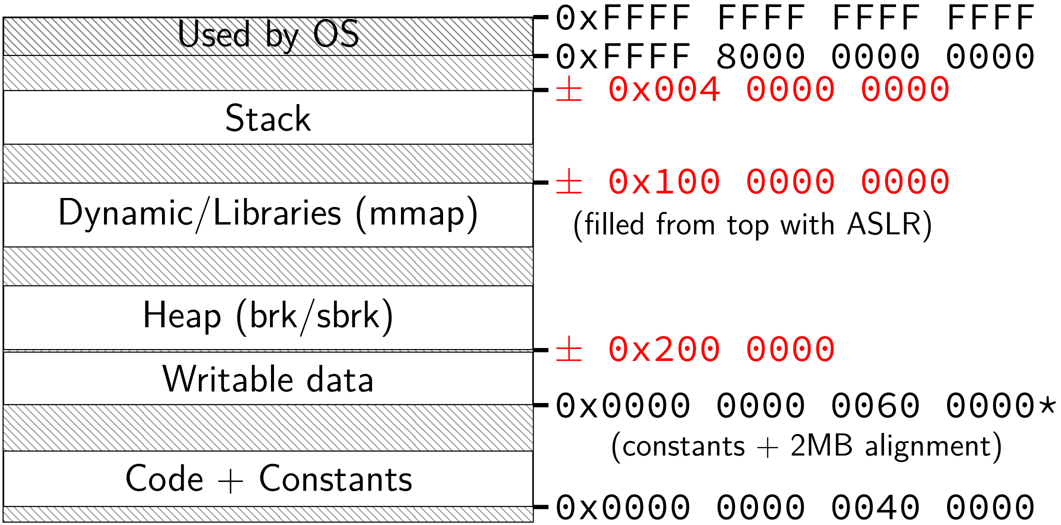
Linux stack randomization (x86-64)

1. choose random number between `0` and `0x3F FFFF`
2. stack starts at `0x7FFF FFFF FFFF` - *random number* × `0x1000`

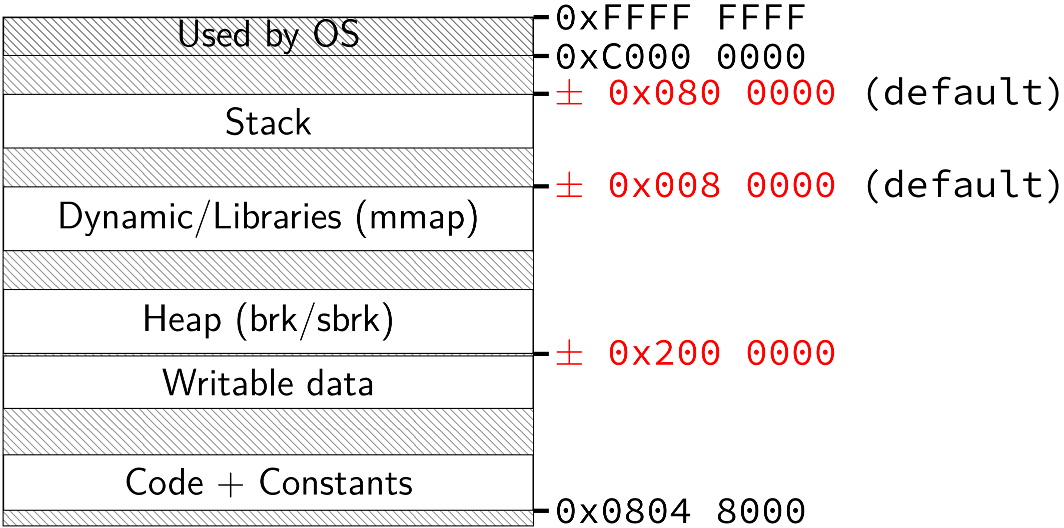
    randomization disabled? *random number* = 0

16 GB range!

program memory (x86-64 Linux; ASLR)



program memory (x86-32 Linux; ASLR)



how much guessing?

gaps change by multiples of page (4K)

- lower 12 bits are fixed

64-bit: huge ranges — need millions of guesses

- about 30 randomized bits in addresses

32-bit: smaller ranges — hundreds of guesses

- only about 8 randomized bits in addresses
- why? only 4 GB to work with!
- can be configured higher — but larger gaps

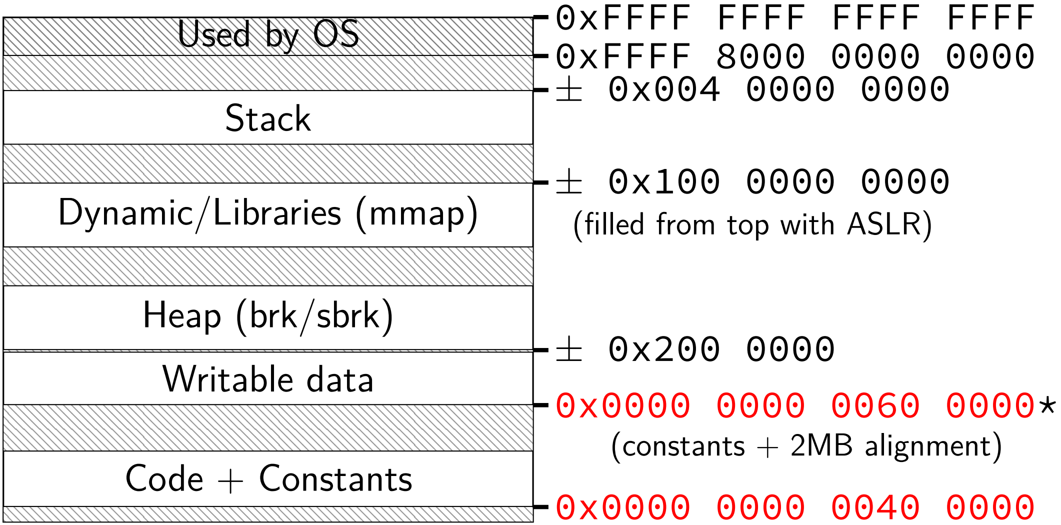
danger of leaking pointers

any stack pointer? know everything on the stack!

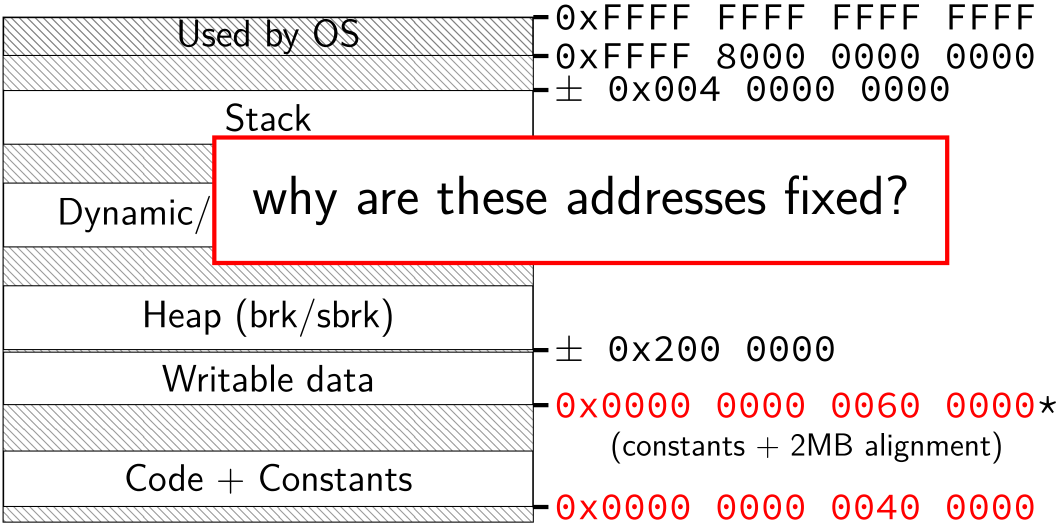
any pointer to a particular library? know everything in library!

- library loaded as one big chunk
- contains many offsets in instructions — can't split easily

program memory (x86-64 Linux; ASLR)



program memory (x86-64 Linux; ASLR)



fixed addresses

machine code contains hard-coded addresses

and is supposed to be loaded without changes

- only small global offset table, etc. changed

one possibility — not fixed

could just edit fixed addresses at load time

usual dynamic linking strategy avoids this

typical dynamic linkers support it, but unused by compilers, etc.

a lot slower than writing small table of addresses

a lot more “metadata” for linker

harder to share memory between programs

relocating: Windows

Windows will edit code to relocate

programs/libraries *usually* include relocation table

typically one fixed location per program/library **per boot**

- same address used across all instances of program/library
- still allows sharing memory

fixup once per program/library per boot

- before ASLR: code could be pre-relocated

Windows + Visual Studio had 'full' ASLR by default since 2010

recall: relocation

```
.data
string: .asciz "Hello,␣World!"
.text
main:
    movq $string, %rdi /* NOT PC/RIP-relative mov */
```

generates: (objdump --disassemble --reloc)

```
   0:   48 c7 c7 00 00 00 00    mov    $0x0,%rdi
                        3: R_X86_64_32S .data
```

relocation record says how to fix 0x0 in mov

- 3: location in machine code
- R_X86_64_32S: 32-bit signed integer
- .data: address to insert

relocating and stubs: Windows

What about the "stubs" and lookup tables?

Windows GOT equivalent is IAT (Import Address Table)

Can't Windows avoid them by editing code?

Probably, but…it doesn't

Why?

Windows ASLR limitation

same address in all programs — not very useful against local exploits

PIC: Linux, OS X

position independent code

- instead of fixed-up hard-coded addresses

Linux, OS X don't fixup code at runtime

previously a challenge for ASLR

libraries on these systems previously had no fixed address

...but executables had a bunch

hard-coded addresses? (64-bit)

```c
int foo(long n) {
    switch (n) {
    case 0:
    case 2:
    case 4:
    case 5:
        return 1;
    case 1:
    case 3:
        return 2;
    default:
        return 3;
    }
}
```

```
foo:
        movl    $3, %eax
        cmpq    $5, %rdi
        ja      defaultCase
        jmp     *lookupTable(,%rdi,8)
        /* code for defaultCase, returnOne,
        ...
        .section        .rodata
lookupTable: /* read-only pointers: */
        .quad   returnOne
        .quad   returnTwo
        .quad   returnOne
        .quad   returnTwo
        .quad   returnOne
        .quad   returnOne
```

hard-coded addresses? (64-bit)

```c
int foo(long n) {
    switch (n) {
    case 0:
    case 2:
    case 4:
    case 5:
        return 1;
    case 1:
    case 3:
        return 2;
    default:
        return 3;
    }
}
```

```
400570 <foo> :
b8 03 00 00 00      mov     $0x3,%eax
48 83 ff 05         cmp     $0x5,%rdi
         /* jump to defaultCase: */
77 12               ja      0x40058d
         /* lookup table jump: */
ff 24 fd
18 06 40 00         jmpq    *0x400618(,%rdi,8)
...
 /* lookupTable @ 0x400618 */
@ 400618: 0x400588  /* returnOne */
@ 400620: 0x400582  /* returnTwo */
@ 400628: 0x400588
@ 400630: 0x400582
```

hard-coded addresses? (64-bit)

```
int foo(long n) {
    switch (n) {
    case 0:
    case 2:
    case 4:
    case 5:
        return 1;
    case 1:
    case 3:
        return 2;
    default:
        return 3;
    }
}
```

```
400570 <foo> :
b8 03 00 00 00      mov    $0x3,%eax
48 83 ff 05         cmp    $0x5,%rdi
        /* jump to defaultCase: */
77 12               ja     0x40058d
        /* lookup table jump: */
ff 24 fd
18 06 40 00         jmpq   *0x400618(,%rdi,8)
...
 /* lookupTable @ 0x400618 */
@ 400618: 0x400588  /* returnOne */
@ 400620: 0x400582  /* returnTwo */
@ 400628: 0x400588
@ 400630: 0x400582
```

hard-coded addresses? (64-bit)

```
int foo(long n) {
    switch (n) {
    case 0:
    case 2:
    case 4:
    case 5:
        return 1;
    case 1:
    case 3:
        return 2;
    default:
        return 3;
    }
}
```

```
400570 <foo> :
b8 03 00 00 00     mov    $0x3,%eax
48 83 ff 05        cmp    $0x5,%rdi
        /* jump to defaultCase: */
77 12              ja     0x40058d
        /* lookup table jump: */
ff 24 fd
18 06 40 00        jmpq   *0x400618(,%rdi,8)
...
 /* lookupTable @ 0x400618 */
@ 400618: 0x400588  /* returnOne */
@ 400620: 0x400582  /* returnTwo */
@ 400628: 0x400588
@ 400630: 0x400582
```

exercise: avoiding absolute addresses

```
foo:
        movl    $3, %eax
        cmpq    $5, %rdi
        ja      defaultCase
        jmp     *lookupTable(,%rdi,8)
returnOne:
        movl    $1, %eax
        ret
returnTwo:
        movl    $2, %eax
defaultCase:
        ret
```

```
lookupTable:
    .quad returnOne
    .quad returnTwo
    .quad returnOne
    .quad returnTwo
    .quad returnOne
    .quad returnOne
```

exercise: rewrite this without absolute addresses

but fast

hard-coded addresses

```
test-64-nopie.exe:     file format elf64-x86-64
test-64-nopie.exe
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0000000000400450

Program Header:
    PHDR off    0x0000000000000040 vaddr 0x0000000000400040  paddr
         filesz 0x00000000000001f8 memsz 0x00000000000001f8 flags
  INTERP off    0x0000000000000238 vaddr 0x0000000000400238  paddr
         filesz 0x000000000000001c memsz 0x000000000000001c flags
    LOAD off    0x0000000000000000 vaddr 0x0000000000400000  paddr
         filesz 0x000000000000078c memsz 0x000000000000078c flags
    LOAD off    0x0000000000000e10 vaddr 0x0000000000600e10  paddr
         filesz 0x0000000000000228 memsz 0x0000000000000230 flags
```

relocation?

one solution: be willing change addresses at load time

requires: table of relocations in executable code

Windows does this

Linux's dynamic linker is not willing to

PIE

position-independent executables (PIE)

- no hardcoded addresses

alternative: edit code (not global offset table) at load time

- Windows solution

GCC: `-pie -fPIE`

- `-pie` is linking option
- `-fPIE` is compilation option
- related option: `-fPIC` (position independent code)
  - used to compile runtime-loaded libraries

PIE jump-table

```
foo:
  movl   $3, %eax
  cmpq   $5, %rdi
  ja     retDefault
  leaq   jumpTable(%rip), %rax
  movslq (%rax,%rdi,4), %rdx
  addq   %rdx, %rax
  jmp    *%rax
returnTwo:
  movl  $2, %eax
  ret
returnOne:
  movl  $1, %eax
defaultCase:
  ret
```

```
  .section      .rodata
jumpTable:
  .long returnOne−jumpTable
  .long returnTwo−jumpTable
  .long returnOne−jumpTable
  .long returnTwo−jumpTable
  .long returnOne−jumpTable
  .long returnOne−jumpTable
```

PIE jump-table

```
foo:
  movl   $3, %eax
  cmpq   $5, %rdi
  ja     retDefault
  leaq   jumpTable(%rip), %rax
  movslq (%rax,%rdi,4), %rdx
  addq   %rdx, %rax
  jmp    *%rax
returnTwo:
  movl  $2, %eax
  ret
returnOne:
  movl  $1, %eax
defaultCase:
  ret
```

```
  .section      .rodata
jumpTable:
  .long returnOne−jumpTable
  .long returnTwo−jumpTable
  .long returnOne−jumpTable
  .long returnTwo−jumpTable
  .long returnOne−jumpTable
  .long returnOne−jumpTable
```

PIE jump-table

```
00000000000007ab  <foo>:
b8 03 00 00 00           mov     $0x3,%eax
48 83 ff 05              cmp     $0x5,%rdi
77 1b                    ja      7d0 <foo+0x25>
48 8d 05 ab 00 00 00     lea     0xab(%rip), %rax        # 868
48 63 14 b8              movslq  (%rax,%rdi,4), %rdx
48 01 d0                 add     %rdx,%rax
ff e0                    jmpq    *%rax
b8 02 00 00 00           mov     $0x2,%eax
c3                       retq
b8 01 00 00 00           mov     $0x1,%eax
c3                       retq
...
@ 868:  -156 /* offset */
@ 870:  -162
...
```

PIE jump-table

```
0000000000007ab  <foo>:
b8 03 00 00 00              mov    $0x3,%eax
48 83 ff 05                 cmp    $0x5,%rdi
77 1b                       ja     7d0 <foo+0x25>
48 8d 05 ab 00 00 00        lea    0xab(%rip), %rax        # 868
48 63 14 b8                 movslq (%rax,%rdi,4), %rdx
48 01 d0                    add    %rdx,%rax
ff e0                       jmpq   *%rax
b8 02 00 00 00              mov    $0x2,%eax
c3                          retq
b8 01 00 00 00              mov    $0x1,%eax
c3                          retq
...
@ 868:  −156 /* offset */
@ 870:  −162
...
```

PIE jump-table

```
00000000000007ab  <foo>:
b8 03 00 00 00              mov     $0x3,%eax
48 83 ff 05                 cmp     $0x5,%rdi
77 1b                       ja      7d0 <foo+0x25>
48 8d 05 ab 00 00 00        lea     0xab(%rip), %rax        # 868
48 63 14 b8                 movslq  (%rax,%rdi,4), %rdx
48 01 d0                    add     %rdx,%rax
ff e0                       jmpq    *%rax
b8 02 00 00 00              mov     $0x2,%eax
c3                          retq
b8 01 00 00 00              mov     $0x1,%eax
c3                          retq
...
@ 868:  −156 /* offset */
@ 870:  −162
...
```

added cost

replace `jmp *jumpTable(,%rdi,8)`

with:

`lea` (get table address — with relative offset)

`movslq` (do table lookup of offset)

`add` (add to base)

`jmp` (to computed base)

32-bit x86 is worse

no relative addressing for `mov`, `lea`, …

even changes “stubs” for printf:

```
// BEFORE: (fixed addresses)
08048310 <__printf_chk@plt>:
 8048310: ff 25 10 a0 04 08   jmp    *0x804a010
    /* 0x804a010 == global offset table entry */

// AFTER: (position-independent)
00000490 <__printf_chk@plt>:
 490:   ff a3 10 00 00 00    jmp    *0x10(%ebx)
    /* %ebx --- address of global offset table */
    /* needs to be set by caller */
```

32-bit x86 is worse

no relative addressing for `mov`, `lea`, ...

even changes "stubs" for printf:

```
// BEFORE: (fixed addresses)
08048310 <__printf_chk@plt>:
 8048310: ff 25 10 a0 04 08    jmp    *0x804a010
    /* 0x804a010 == global offset table entry */

// AFTER: (position-independent)
00000490 <__printf_chk@plt>:
 490:   ff a3 10 00 00 00    jmp    *0x10(%ebx)
    /* %ebx --- address of global offset table */
    /* needs to be set by caller */
```

32-bit x86 is worse

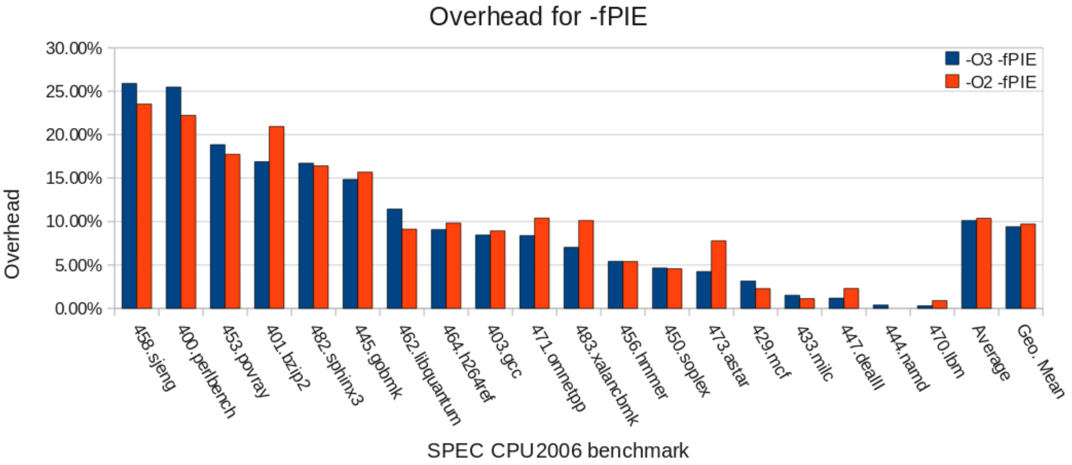
no relative addressing for `mov`, `lea`, ...

even changes "stubs" for printf:

```
// BEFORE: (fixed addresses)
08048310 <__printf_chk@plt>:
 8048310: ff 25 10 a0 04 08   jmp    *0x804a010
    /* 0x804a010 == global offset table entry */

// AFTER: (position-independent)
00000490 <__printf_chk@plt>:
 490:   ff a3 10 00 00 00    jmp    *0x10(%ebx)
    /* %ebx --- address of global offset table */
    /* needs to be set by caller */
```

position independence cost (32-bit)



Payer, "Too much PIE is bad for performance", ETH Zurich Tech Report

position independence cost: Linux

geometric mean of SPECcpu2006 benchmarks on x86 Linux

  with particular version of GCC, etc., etc.

64-bit: 2-3% (???)

  "preliminary result"; couldn't find reliable published data

32-bit: 9-10%

depends on compiler, ...

position independence: deployment

my laptop (64-bit Ubuntu 16.04): ~14% of executables are PIE

Ubuntu 16.10 (released 2016 Oct): enables PIE by default

also Debian Stretch (late 2016), Fedora 23 (late 2015), ...

OS X enables PIE by default since 10.7 (despite perf. cost)

the mapping (set by OS)

program address range	read?	write?	exec?	real address
0x0000 --- 0x0FFF	no	no	no	---
0x1000 --- 0x1FFF	no	no	no	---
…				
0x40 0000 --- 0x40 0FFF	yes	no	yes	0x...
0x40 1000 --- 0x40 1FFF	yes	no	yes	0x...
0x40 2000 --- 0x40 2FFF	yes	no	yes	0x...
…				
0x60 0000 --- 0x60 0FFF	yes	yes	no	0x...
0x60 1000 --- 0x60 1FFF	yes	yes	no	0x...
…				
0x7FFF FF00 0000 — 0x7FFF FF00 0FFF	yes	yes	no	0x...
0x7FFF FF00 1000 — 0x7FFF FF00 1FFF	yes	yes	no	0x...
…				

write XOR execute

many names:

- `W^X` (write XOR execute)
- DEP (Data Execution Prevention)
- NX bit (No-eXecute) (hardware support)
- XD bit (eXecute Disable) (hardware support)

mark writeable memory as executable

how will users insert their machine code?

- can only code in application + libraries
- a problem, right?

hardware support for write XOR execute

everywhere today

not historically common

early x86: execute implied by read

NX support added with x86-64 and around 2000 for x86-32

deliberate use of writeable code

"just-in-time" (JIT) compilers

- fast virtual machine/language implementations

some weird GCC features

older "signals" on Linux

- OS wrote machine code on stack for program to run

couldn't even disable executable stacks without breaking applications

why doesn't W xor X solve the problem?

ASLR, stack canaries, etc. had performance penalty
- and failed if there was an information leak

W xor X is "almost free", keeps attacker from writing code?

problem: useful machine code is in program already
- just need to find writable function pointer

next topic: ROP

return-oriented programming

AKA arc injection

AKA return-to-libc

find “chain” of machine code that does what you want

ROP case study

simple stack buffer overflow with write XOR execute

stack canaries disabled

ASLR disabled

- in practice — rely on information disclosure bug

vulnerable application

```
#include <stdio.h>

int vulnerable() {
    char buffer[100];
    gets(buffer);
}

int main(void) {
    vulnerable();
}
```

vulnerable function

```
0000000000400536 <vulnerable>:
  400536:       48 83 ec 78             sub    $0x78,%rsp
  40053a:       31 c0                   xor    %eax,%eax
  40053c:       48 8d 7c 24 0c          lea    0xc(%rsp),%rdi
  400541:       e8 ca fe ff ff          callq  400410 <gets@plt>
  400546:       48 83 c4 78             add    $0x78,%rsp
  40054a:       c3                      retq
```

vulnerable function

```
0000000000400536 <vulnerable>:
  400536:       48 83 ec 78             sub    $0x78,%rsp
  40053a:       31 c0                   xor    %eax,%eax
  40053c:       48 8d 7c 24 0c          lea    0xc(%rsp),%rdi
  400541:       e8 ca fe ff ff          callq  400410 <gets@plt>
  400546:       48 83 c4 78             add    $0x78,%rsp
  40054a:       c3                      retq
```

buffer at `0xC` + stack pointer

return address at `0x78` + stack pointer

= `0x6c` + buffer

memory layout

going to look for interesting code to run in libc.so

- implements gets, printf, etc.

loaded at address `0x2aaaaacd3000`

our task

print out the message “You have been exploited.”

ultimately calling `puts`

which will be at address `0x2aaaaad42690`

shellcode

```
        lea  string(%rip), %rdi
        mov  $0x2aaaaad42690, %rax /* puts */
        jmpq *(%rax)
string: .ascii "You␣have␣been␣exploited.\0"
```

but — can't insert code

surely this code doesn't exist in libc already

solution: find code for pieces

loading string into RDI

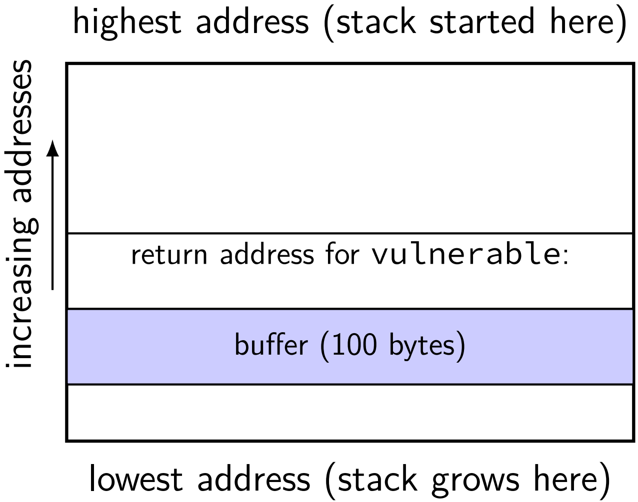
can we even load a pointer to the string into `%rdi`?

let's look carefully at code in `libc.so`

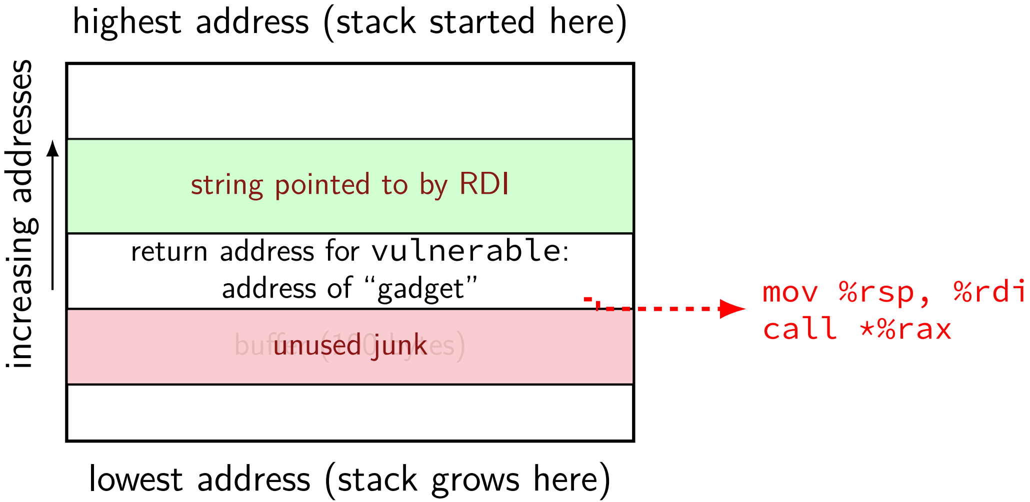
```
2aaaaadfdc95:       48 89 e7                    mov    %rsp,%rdi
2aaaaadfdc98:       ff d0                       callq  *%rax
```

just need to get address of `puts` into `%rax` before this

load RDI


highest address (stack started here)
increasing addresses
return address for vulnerable:
buffer (100 bytes)
lowest address (stack grows here)

load RDI


highest address (stack started here)
increasing addresses
string pointed to by RDI
return address for vulnerable:
address of "gadget"
buffer (100 bytes)
unused junk
lowest address (stack grows here)
mov %rsp, %rdi
call *%rax

loading puts addr. into RAX

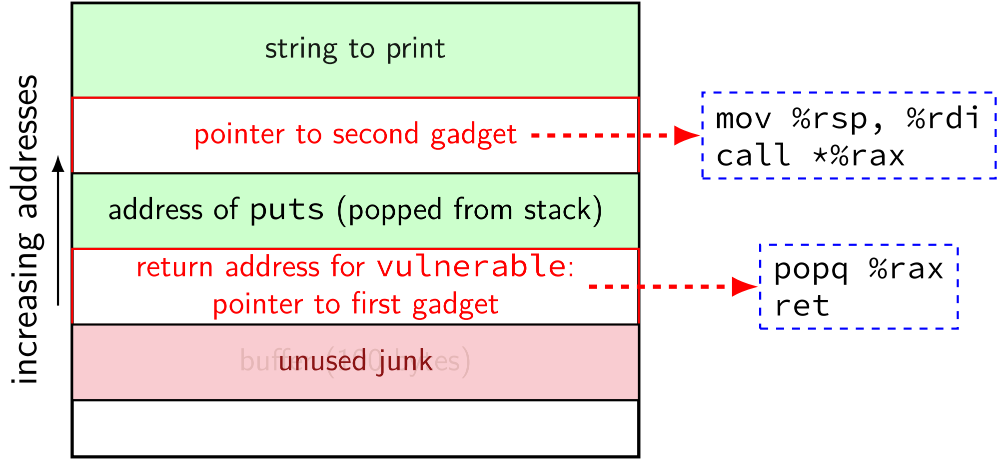
```
2aaaaad06543:       e8 58 c3 fe ff          callq  2aaaaaaf48a0
```

`58 c3` can be interpreted another way:

```
2aaaaad06544:       58           popq %rax
2aaaaad06545:       c3           retq
```

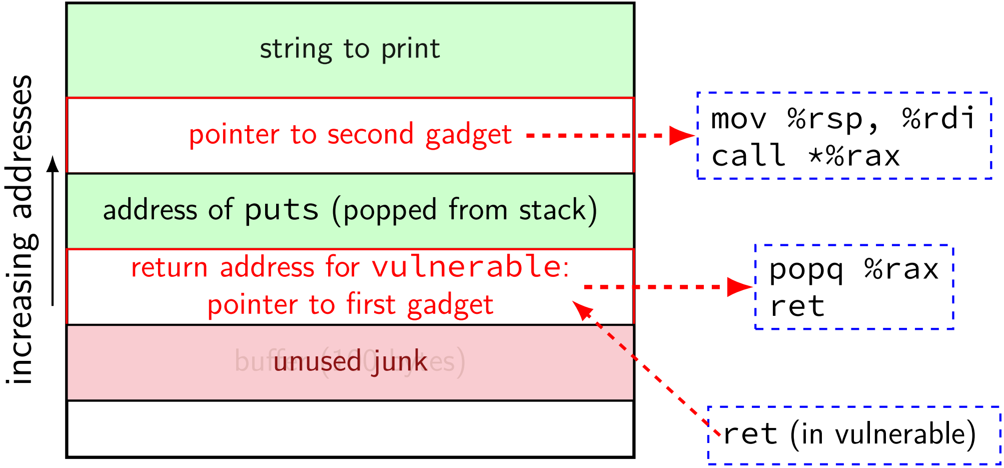
"ret" lets us **chain** this to execute `call` snippet next

ROP chain



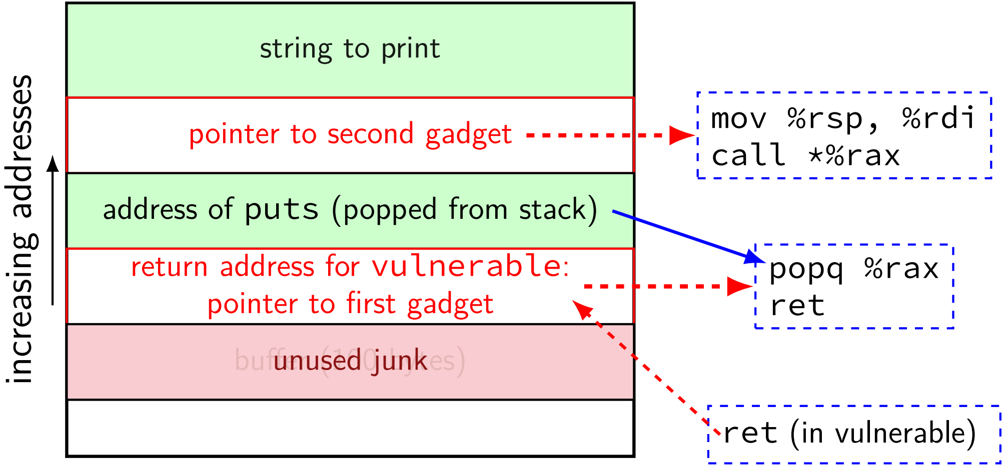
increasing addresses
string to print
pointer to second gadget
address of puts (popped from stack)
return address for vulnerable:
pointer to first gadget
unused junk
mov %rsp, %rdi
call *%rax
popq %rax
ret

ROP chain



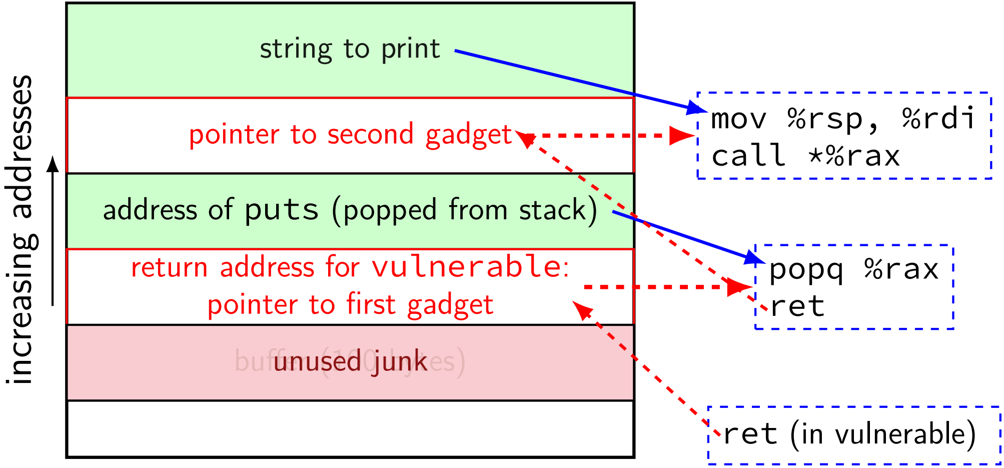
increasing addresses
string to print
pointer to second gadget
address of puts (popped from stack)
return address for vulnerable:
pointer to first gadget
unused junk
mov %rsp, %rdi
call *%rax
popq %rax
ret
ret (in vulnerable)

ROP chain



increasing addresses
string to print
pointer to second gadget
address of puts (popped from stack)
return address for vulnerable:
pointer to first gadget
unused junk
mov %rsp, %rdi
call *%rax
popq %rax
ret
ret (in vulnerable)

ROP chain


increasing addresses
string to print
pointer to second gadget
address of puts (popped from stack)
return address for vulnerable:
pointer to first gadget
buffer (100 bytes)
unused junk
mov %rsp, %rdi
call *%rax
popq %rax
ret
ret (in vulnerable)

demo

how did I find that?

no, I am not really good at looking at `objdump` output

tools scan binaries for *gadgets*

one you'll use in upcoming homework

gadgets generally

bits of machine code that do work, then return or jump

“chain” together, by having them jump to each other

most common: find gadget ending with `ret`

- pops address of next gadget offs tack

can do pretty much anything

ROP and ASLR

find a pointer to known thing in libc (or other source of gadgets)

e.g. information leak from use-after-free

use that to compute address of all gadgets

then address randomization doesn't matter

ROP and write XOR execute

all the code we're running is supposed to be executed

completely defeats write XOR execute

ROP and stack canaries

information disclosure reveals canary value if needed, still

full stack canaries should reduce number of gadgets

- no real returns without canary checks

...but typically only canaries if stack-allocated buffer

and return opcodes within other instructions

ROP without a stack overflow (1)

e.g. VTable overwrite

look for gadget(s) that set `%rsp`

…based on function argument registers/etc.

ROP without stack overflow (2)

example sequence from my libc:

```
push %rdi; call *(%rdx)
pop %rsp; ret
```

set:

overwritten vtable entry = first gadget
arg 1: `%rdi` = desired stack pointer
arg 3: `%rdx` = pointer to second gadget

jump-oriented programming

just look for gadgets that end in `call` or `jmp`

don't even need to set stack

harder to find than `ret`-based gadgets

- but almost always as powerful as ret-based gadgets

finding gadgets

find code segments of exectuable/library

look for opcodes of arbitrary jumps:

```
ret
jmp *register
jmp *(register)
call *register
call *(register)
```

disassemble starting a few bytes before

- invalid instruction? jump before ret? etc. — discard

sort list

finding gadgets: demo

programming with gadgets

can usually find gadgets to:

- pop from stack into argument register
- write register to memory location in another register
- clear a register

along with gadget for `syscall` (make OS call) — can do anything

common, reusable ROP sequences

open a command-line — what `ROPgadget` tool defaults to

make memory executable + jump

generally: just do enough to ignore write XOR execute

often only depend on memory locations in shared library

ROP ideas

incidental *existing* snippets of code

chain together with non-constant jumps

- returns, function pointer calls, computed jumps

snippets form “language”

- usually Turing-complete