address space layout randomization (ASLR)

vary the location of things in memory

including the stack

designed to make exploiting memory errors harder

will talk more about later

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

recall: position independent executables

```
...
EXEC_P, D_PAGED
...
LOAD off    0x0000000 vaddr 0x400000 paddr 0x0400000 align 2**12
     filesz 0x00006c8 memsz 0x0006c8 flags r--
LOAD off    0x0001000 vaddr 0x401000 paddr 0x0401000 align 2**12
     filesz 0x01a7865 memsz 0x1a7865 flags r-x
```

some executables had LOADs at fixed addresses

- machine code might use hard-coded addresses

can't randomize program addresses

others did not (marked DYNAMIC)

```
...
HAS_SYMS, DYNAMIC, D_PAGED
...
LOAD off    0x000000 vaddr 0x000000 paddr 0x000000 align 2**12
     filesz 0x0036f8 memsz 0x0036f8 flags r--
LOAD off    0x004000 vaddr 0x004000 paddr 0x004000 align 2**12
...
```

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* $= 0$

   times `0x1000` because OS has to allocate whole pages (0x1000 bytes)

16 GB range!

Linux stack randomization (x86-64)

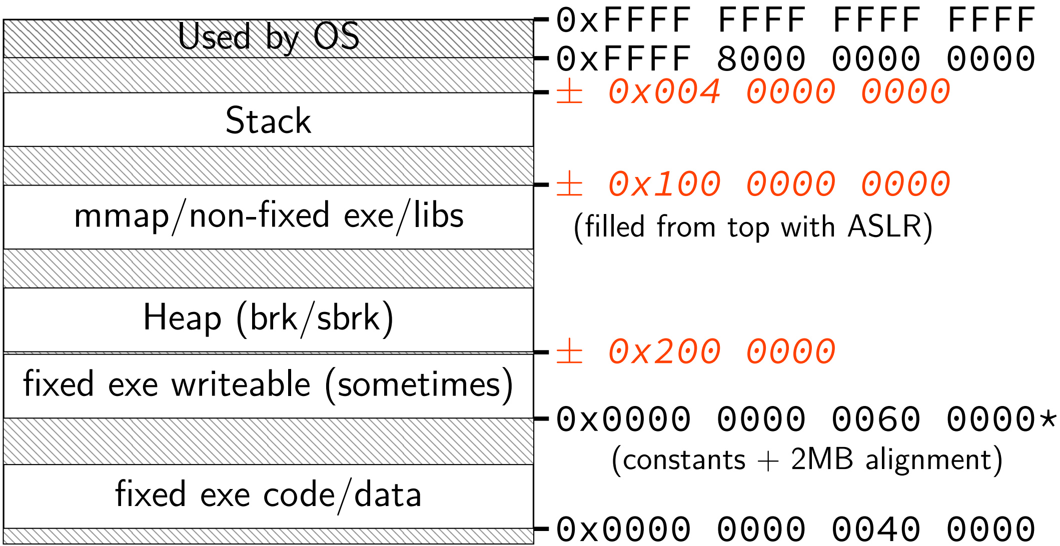
1. choose random number between `0` and *`0x3F FFFF`*
2. stack starts at `0x7FFF FFFF FFFF` - *random number* $\times$ `0x1000`

   randomization disabled? *random number* $= 0$

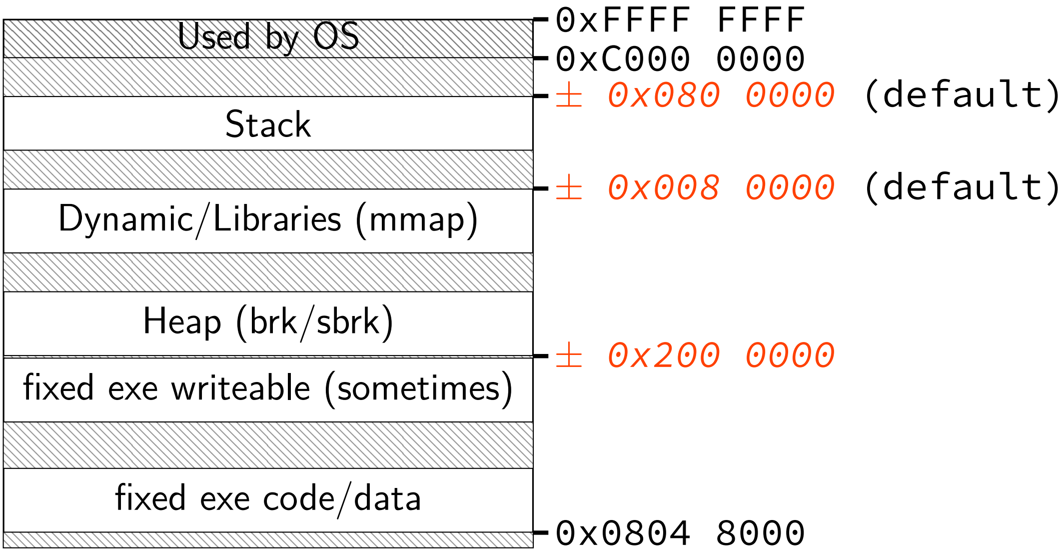
   times `0x1000` because OS has to allocate whole pages (0x1000 bytes)

16 GB range!

program memory (x86-64 Linux; ASLR)



program memory (x86-32 Linux; ASLR)



how much guessing?

gaps change by multiples of page size (4KB)

- 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

why do we get multiple guesses?

why do we get multiple guesses?

wrong guess might not crash

wrong guess might not crash whole application

- e.g. server that uses multiple processes

local programs we can repeatedly run

servers that are automatically restarted

entropy exercise

suppose we have 32-bit Linux server vulnerable to stack smashing

...but stack address randomized with 256 possible starting locations

- +/- 0x80 in increments of 0x1000

server is automatically restarted after unsuccessful attack

suppose stack layout is 8KB buffer + return address + 12KB other stuff

what should attacker do to maximize chance of success?

about how many tries needed for successful attack?

exercise

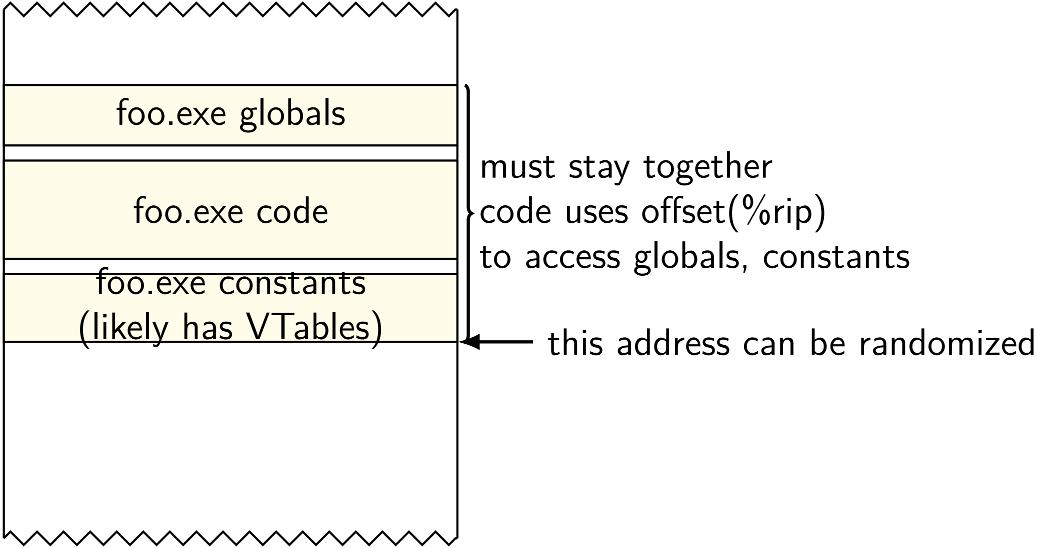
```
struct point {
    int x, y, z;;
};
```

```
struct point *p;
...
    if (command == "get") {
        /* 'p' could be uninitialized */
        printf("%d,%d,%d\n", p->x, p->y, p->z);
    } ...
...
```

Which initial value for p (“left over” from prior use of register, etc.) would be most useful for a later buffer overflow attack?

A. p is an invalid pointer and accessing it will crash the program
B. p points to global variable
C. p points to space on the stack that is currently unallocated, but last contained an input buffer
D. p points to space on the stack that currently holds a return address
E. p points to space on the stack that is currently unallocated, but last contained a pointer to the last used byte of an array on the stack

exes, libraries stay together



dependencies between segments (1)

```
$ objdump -x foo.exe
...
LOAD off    0x0000000000000000 vaddr 0x0000000000000000 paddr 0x000
     filesz 0x0000000000000620 memsz 0x0000000000000620 flags r--
LOAD off    0x0000000000001000 vaddr 0x0000000000001000 paddr 0x000
     filesz 0x0000000000000205 memsz 0x0000000000000205 flags r-x
LOAD off    0x0000000000002000 vaddr 0x0000000000002000 paddr 0x000
     filesz 0x0000000000000150 memsz 0x0000000000000150 flags r--
LOAD off    0x0000000000002db8 vaddr 0x0000000000003db8 paddr 0x000
     filesz 0x000000000000025c memsz 0x0000000000000260 flags rw-
```

4 seperately loaded segments: can we choose random addresses for each?

dependencies between segments (2)

```
0000000000001050 <__printf_chk@plt>:
    1050:       f3 0f 1e fa             endbr64
    1054:       f2 ff 25 75 2f 00 00    bnd jmpq *0x2f75(%rip)
    105b:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```

dependency from 2nd LOAD (0x1000-0x1205) to 4th LOAD (0x3db8-0x4018)

uses relative addressing rather than linker filling in address

dependencies between segments (3)

```
0000000000001060 <main>:
    1060:       f3 0f 1e fa             endbr64
    1064:       50                      push   %rax
    1065:       8b 15 a5 2f 00 00       mov    0x2fa5(%rip),%edx
# 4010 <global>
    106b:       48 8d 35 92 0f 00 00    lea    0xf92(%rip),%rsi
# 2004 <_IO_stdin_used+0x4>
    1072:       31 c0                   xor    %eax,%eax
    1074:       bf 01 00 00 00          mov    $0x1,%edi
    1079:       e8 d2 ff ff ff          callq  1050 <__printf_chk@p
```

dependency from 2nd LOAD (0x1000-0x1205) to 3rd LOAD (0x2000-0x2150)

uses relative addressing rather than linker filling in address

why is this done?

Linux made a choice:
no editing code when loading programs, libraries

allows same code to be loaded in multiple processes

danger of leaking pointers

any stack pointer? know everything on the stack!

any pointer within executable? know everything in the executable!

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

exericse: using a leak (1)

```
class Foo {
    virtual const char *bar() { ... }
};
...
Foo *f = new Foo;
printf("%s\n", f);
```

Part 1: What address is most likely leaked by the above?

A. the location of the Foo object allocated on the heap
B. the location of the first entry in Foo's VTable
C. the location of the first instruction of Foo::Foo() (Foo's compiler-generated constructor)
D. the location of the stack pointer

using a leak (1) answer

printing out beginning of C++ object = VTable pointer

exercise: using a leak (2)

```
class Foo { virtual const char *bar() { ... } };
...
Foo *f = new Foo;
char *p = new char[1024];
printf("%s\n", f);
```

if leaked value was 0x822003 and in a debugger (with **different randomization**):

- stack pointer was 0x7ffff000
- Foo::bar's address was 0x400000
- f's address was 0x900000
- f's Vtable's address was 0x403000
- a "gadget" address from the main executable was 0x401034
- a "gadget" address from the C library was 0x2aaaa40034
- p's address was 0x901000

which of the above can I compute based on the leak?

using a leak (2) answer

VTable pointer part of same object/library containing class Foo definition

so can use its location to find code/data from same executable

- gadget in main executable
- Foo::bar definition
- global variables (not listed)

can't use it to find things on heap, stack, in C library

- those are separately randomized

ex: using information leak (2)

```
printf("buffer = %p", buffer)
```

```
buffer = 0x646d06d15040
```

```
$ objdump -tR a.out
...
0000000000004040 g     O .bss   0000000000000400              buffer
...
0000000000003fb0 R_X86_64_JUMP_SLOT  strlen@GLIBC_2.2.5
$ objdump -d a.out
...
0000000000001090 <strlen@plt>:
    1090:       f3 0f 1e fa             endbr64
    1094:       ff 25 16 2f 00 00       jmp    *0x2f16(%rip)        # 3fb0 <strlen
    109a:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
...
```

exercise: address to overwrite to make strlen(X) run other code?

ex: using information leak (2) soln

buffer address = 0x646d06d15040 - offset = 0x4040

- printed out actual value

offset = 0x646d06d11000

GOT entry address = 0x3fb0 + offset = 0x646d06d14fb0

- 0x3fb0 = jump slot location

why not always ASLR?

ASLR seems like no-brainer

- have to choose address anyway
- why not choose at random?

big problem: performance/code size impacts

(smaller problem: inconsistent behavior when bugs)

position-independent code

position-independent code = code that can be loaded anywhere

- no hard-coded addresses

necessary prerequisite for most of ASLR

Unix did this for libraries for non-security reasons

- share memory between multiple programs loading same library
- allow programs to load libraries at any location

but not other programs, probably because of overheads

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 (lower runtime cost)

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

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* (lower runtime cost)

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

Windows ASLR limitation

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

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

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

```
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
...
```

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 (1)

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 (1)

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 (1)

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 (1)

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

even changes "stubs" for printf:

```
// BEFORE: (fixed addresses)
08049040 <puts@plt>:
 8049040:       ff 25 04 c0 04 08       jmp    *0x804c004

// AFTER: (position-independent)
00000490 <puts@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 (1)

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

even changes "stubs" for printf:

```
// BEFORE: (fixed addresses)
08049040 <puts@plt>:
 8049040:       ff 25 04 c0 04 08       jmp    *0x804c004

// AFTER: (position-independent)
00000490 <puts@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 (1)

no relative addressing for mov, lea, ...

even changes "stubs" for printf:

```
// BEFORE: (fixed addresses)
08049040 <puts@plt>:
 8049040:       ff 25 04 c0 04 08       jmp    *0x804c004

// AFTER: (position-independent)
00000490 <puts@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 (2)

changes to call

```
// BEFORE: (fixed addresses)
 8049061:   68 08 a0 04 08          push   $0x804a008
 8049066:   e8 d5 ff ff ff          call   8049040 <puts@plt>

// AFTER: (position-independent)
000010d0 <__x86.get_pc_thunk.bx>:
    10d0:   8b 1c 24                mov    (%esp),%ebx
    10d3:   c3                      ret
...

    106e:   e8 5d 00 00 00          call   10d0 <__x86.get_pc_thunk.bx>
    1073:   81 c3 65 2f 00 00       add    $0x2f65,%ebx
...
    107d:   8d 83 30 e0 ff ff       lea    -0x1fd0(%ebx),%eax
    1083:   50                      push   %eax
    1084:   e8 b7 ff ff ff          call   1040 <puts@plt>
```

extra relocations

```
struct Foo {
    virtual const char *bar() { return "Foo::bar"; }
};

int main() {
    Foo *f = new Foo;
    f->bar();
}
```

needed: VTable for Foo

contains function pointers — but function addresses change

how is that setup? extra work on program loading

position-independent versus not

```
$ objdump -R example2

example2:     file format elf64-x86-64

DYNAMIC RELOCATION RECORDS
OFFSET           TYPE              VALUE
0000000000003da8 R_X86_64_RELATIVE  *ABS*+0x0000000000001160
0000000000003db0 R_X86_64_RELATIVE  *ABS*+0x0000000000001120
0000000000004008 R_X86_64_RELATIVE  *ABS*+0x0000000000004008
0000000000003fd8 R_X86_64_GLOB_DAT  __cxa_finalize@GLIBC_2.2.5
0000000000003fe0 R_X86_64_GLOB_DAT  _ITM_deregisterTMCloneTable
0000000000003fe8 R_X86_64_GLOB_DAT  __libc_start_main@GLIBC_2.2.5
0000000000003ff0 R_X86_64_GLOB_DAT  __gmon_start__
0000000000003ff8 R_X86_64_GLOB_DAT  _ITM_registerTMCloneTable
0000000000003fd0 R_X86_64_JUMP_SLOT  _Znwm@GLIBCXX_3.4
```

```
$ objdump -R example2-nopie

example2-nopie:     file format elf64-x86-64

DYNAMIC RELOCATION RECORDS
OFFSET           TYPE              VALUE
0000000000403ff0 R_X86_64_GLOB_DAT  __libc_start_main@GLIBC_2.2.5
0000000000403ff8 R_X86_64_GLOB_DAT  __gmon_start__
0000000000404018 R_X86_64_JUMP_SLOT  _Znwm@GLIBCXX_3.4
```

GCC/Clang options

-fPIC: generate position-independent code for library

-fpic — possibly less flexible/faster version on some platforms

-fPIE, -fpie: generate position-independent code for executable

-pie: link position-independent executable

-no-pie: don't (where -pie is default)

-shared: link shared library

-fPIC/-fPIE differences

```
extern int foo;
int example() {return foo;}
```

with -fPIC:

```
0000000000000000 <example>:
   0:   48 8b 05 00 00 00 00     mov     0x0(%rip),%rax
                3: R_X86_64_REX_GOTPCRELX        foo-0x4
   7:   8b 00                    mov     (%rax),%eax
   9:   c3                       ret
```

with -fPIE:

```
0000000000000000 <example>:
   0:   8b 05 00 00 00 00        mov     0x0(%rip),%eax
                2: R_X86_64_PC32        foo-0x4
   6:   c3                       ret
```

GOTPCREL

saw two different relocations for global `int foo`:

R_X86_64_PC32 relocation = 32-bit offset to variable

- okay in executable: we'll figure out where `foo` is
- will redirect libraries to use exectuable version

R_X86_64_REX_GOTPCRELX relocation = 32-bit offset to global offset table entry containing address

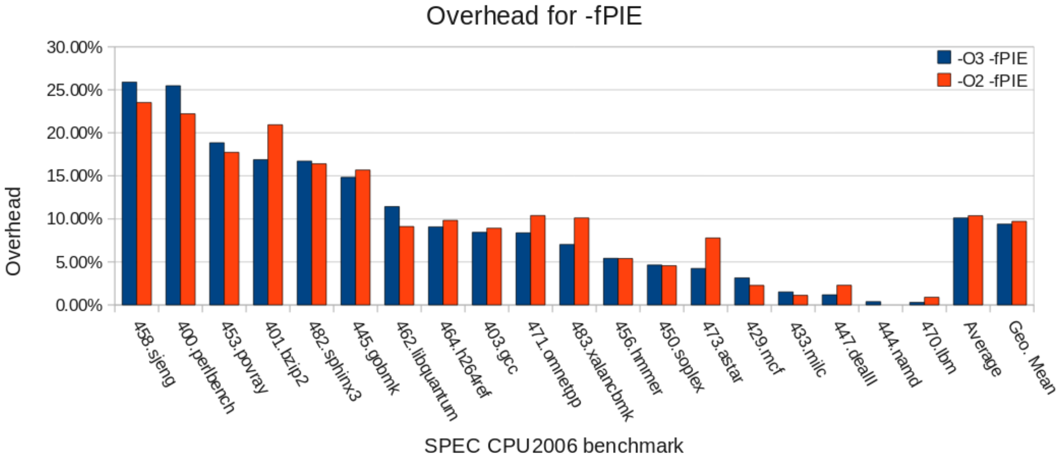
- `foo`'s location decided at runtime by linker
- runtime linker writes pointer to library's global offset table
- ('REX' part says where instruction starts relative to constant, for fancy linkers)

global offset tableS?

executable and library loaded at different addresses

each has own global offset table loaded next to it

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

common for a very long time in dynamic libraries

default for all executables 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

backup slides