address space layout randomization (ASLR)

assume: addresses don't leak

choose *random* addresses each time

- for *everything*, not just the stack

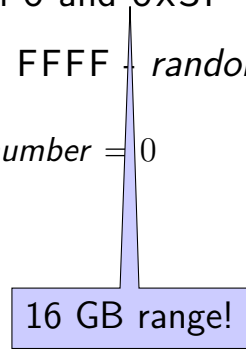
enough possibilities that attacker won't "get lucky"

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`

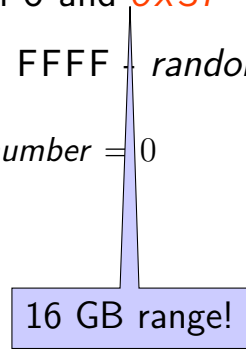
   randomization disabled? *random number* = 0



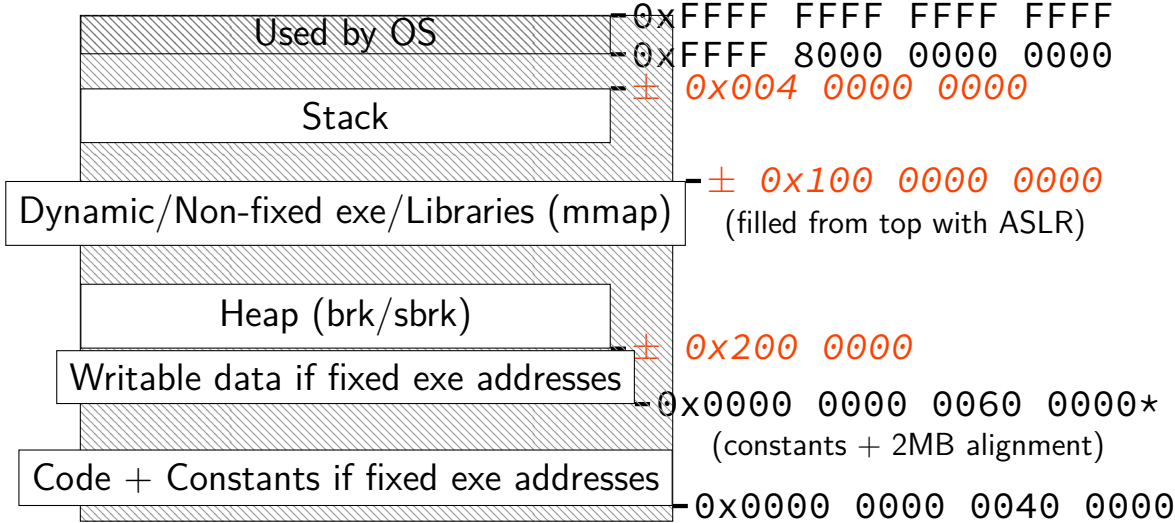
Linux stack randomization (x86-64)

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

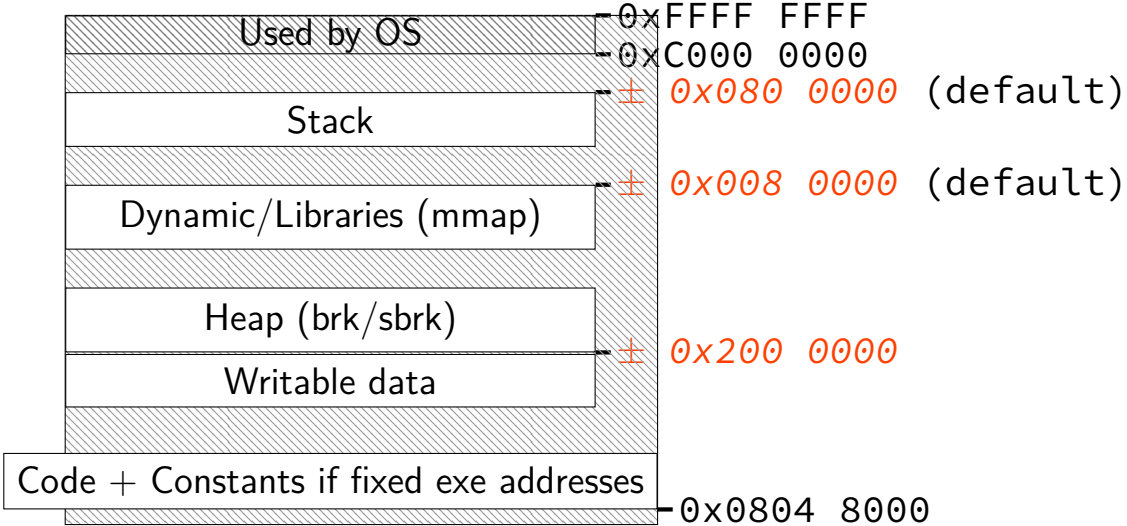
   randomization disabled? *random number* $= 0$



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

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

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

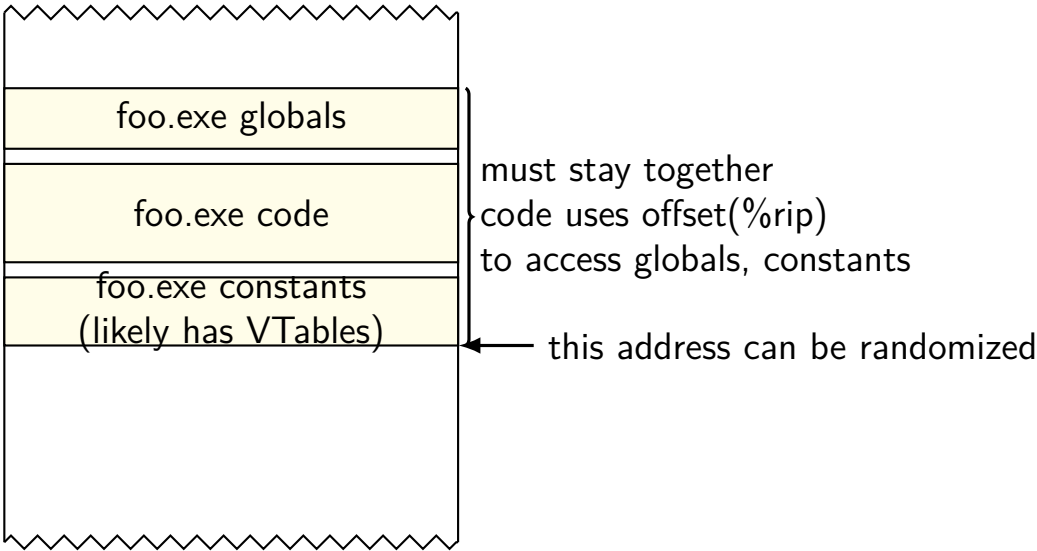
exercise: using a leak (2)

```
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

exes, libraries stay together



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

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 */
```

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

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;
    }
}
```

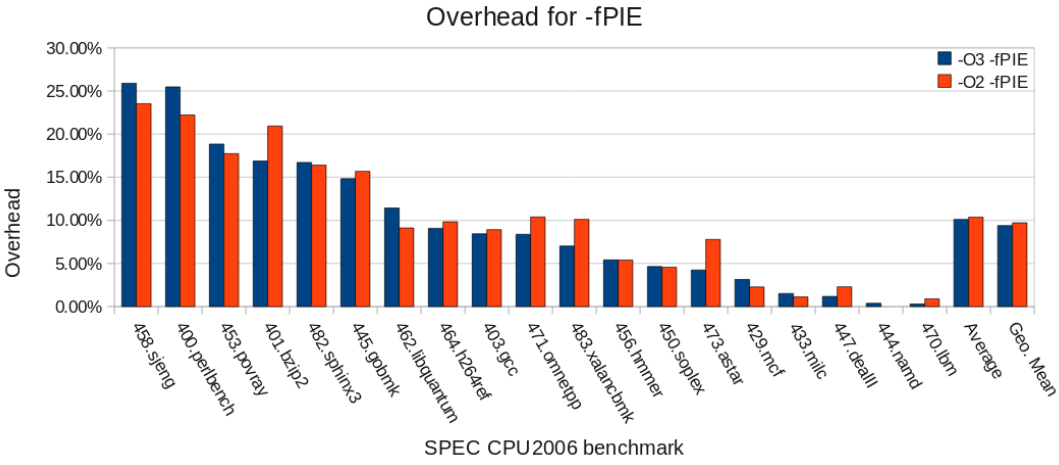
```
400570 <foo>:
b8 03 00 00 00     mov    $0x3,%eax
48 83 ff 05        cmp    $0x5,%rdi
          /* jump to defaultCase: */
77 12              ja     0x40058d
          /* lookup table jump: */
ff 24 fd
18 06 40 00        jmpq   *0x400618(,%rdi,8)
...
 /* lookupTable @ 0x400618 */
@ 400618: 0x400588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

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

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