Changelog

22 March 2021 (after lecture): ROP without stack overflow: adjust call gadget into jmp gadget so it is functional

22 March 2021 (after lecture): add "if exe addreses fixed" to ASLR diagrams showing typical locations of executable code loading

22 March 2021 (after lecture): add diagram illustrating executable staying together before explanation with objdump snippets

22 March 2021 (after lecture): dependencies between segments (2): correct formatting issue

22 March 2021 (after lecture): using leak exericses: add explanation slides

22 March 2021 (after lecture): using leak exericse (2): remove "in decimal" from options

last time

memory protection

- read-only function pointers
- write XOR execute

“gadgets” in existing code

return-oriented programming

some notes on OBFUSCATE

this was an assignment where I most unsure of difficulty callibration

one correct, but unintended solution:

- the password check used memcmp
- not much else used memcmp
- could replace memcmp call entirely
- probably should've reimplemented that

my TTT3 soln

```
case 104:
(PC[0]) ++; /* renamed via find-replace */
/* debug output: */
printf("branch @ PC = %ld\n", PC[0] - CODE[0]);
if (PC[0] - CODE[0] == 2063) { /* found by examining debug output */
    if (!(l___4915[0] + 0)->f___11) {
      PC[0] += *((int *)PC[0]);
    } else {
      PC[0] += 4;
    }
} else {
    if ((l___4915[0] + 0)->f___11) {
      PC[0] += *((int *)PC[0]);
    } else {
      PC[0] += 4;
    }
}
```

on OVER correction

ROP without a stack overflow (1)

we can use ROP ideas for non-stack exploits

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

...based on function argument registers/etc.

ROP without stack overflow (2)

example sequence:

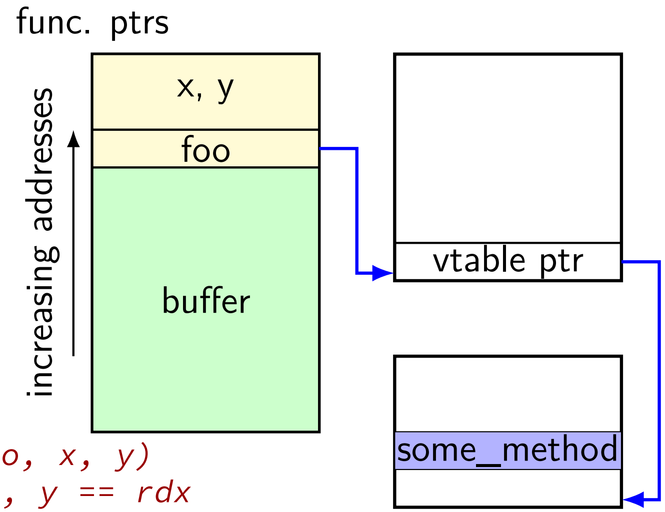
- gadget 1: `push %rdi; jmp *(%rdx)`
- gadget 2: `pop %rsp; ret`

set:

- overwritten function pointer = pointer to gadget 1
- arg 1: `%rdi` = desired stack pointer (pointer to next gadgets)
- arg 3: `%rdx` = pointer to gadget 2

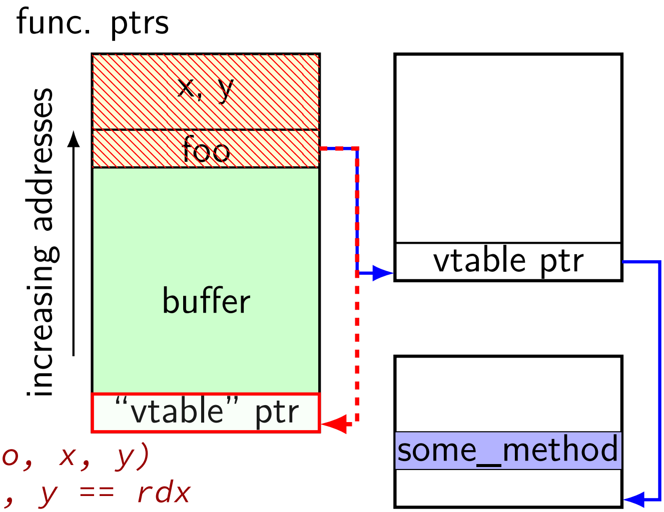
VTable overwrite with gadget

```
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
}
```



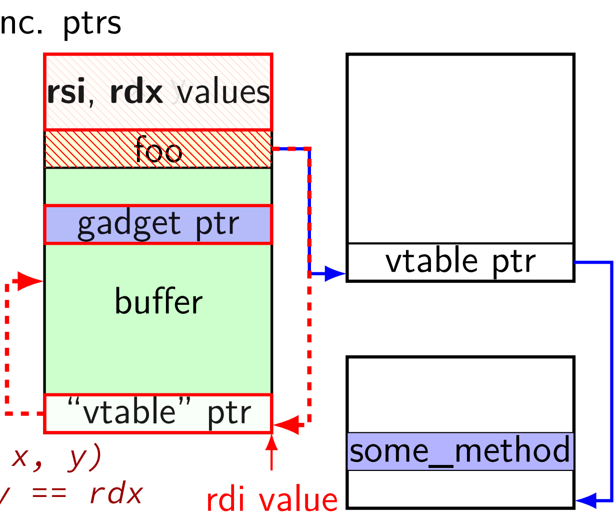
VTable overwrite with gadget

```
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
}
```

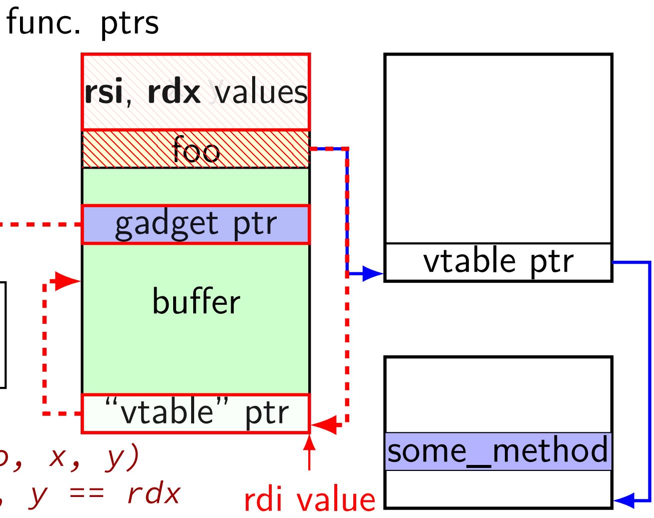


VTable overwrite with gadget

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

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

```
   gets(buffer);
   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

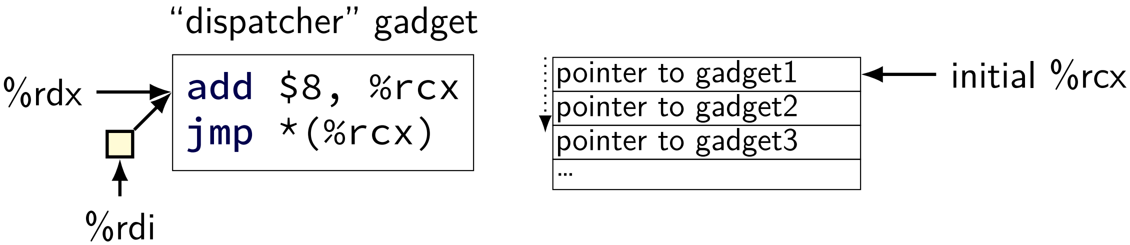
- but almost always as powerful as ret-based gadgets

programming JOP

"dispatcher" gadget

```
add $8, %rcx
jmp *(%rcx)
```

programming JOP



programming JOP

“dispatcher” gadget

%rdx

%rdi

```
add $8, %rcx
jmp *(%rcx)
```

pointer to gadget1
pointer to gadget2
pointer to gadget3
…

initial %rcx

template for other gadgets

```
...
jmp *%rdx
```

— *OR* —

```
...
jmp *(%rdi)
```

programming JOP

“dispatcher” gadget

%rdx → / %rdi →

```
add $8, %rcx
jmp *(%rcx)
```

pointer to gadget1
pointer to gadget2
pointer to gadget3
…

← initial %rcx

template for other gadgets

```
...
jmp *%rdx
```

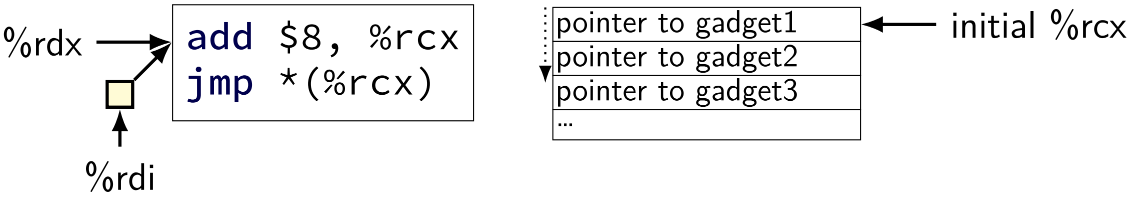
— *OR* —

```
...
jmp *(%rdi)
```

setup: find a way to set %rdx, %rdi, %rcx appropriately

programming JOP

"dispatcher" gadget



template for other gadgets

```
...
jmp *%rdx
```

— *OR* —

```
...
jmp *(%rdi)
```

setup: find a way to set %rdx, %rdi, %rcx appropriately

note: can choose different registers, dispatcher design

dispatcher gadgets?

```
/* from libc on my desktop: */
adc esi, edi ; jmp qword ptr [rsi + 0xf]
add al, ch ; jmp qword ptr [rax - 0xe]

/* from firefox on my desktop: */
add eax, ebp ; jmp qword ptr [rax]
add edi, -8 ; mov rax, qword ptr [rdi] ; jmp qword ptr [rax + 0x68]
sub esi, dword ptr [rsi] ; jmp qword ptr [rsi - 0x7d]
```

adc (add with carry) — Intel syntax: destination first

using function pointer overwrite (1)

```
struct Example {
    char input[1000];
    void (*process_function)(Example *, long, char *);
};
void vulnerable(struct Example *e) {
    long index;
    char name[1000];
    gets(e->input); /* can overwrite process_function */
    scanf("%ld,%s", &index, &name[0]); /* expects <decimal number>,<string> */
    (e->process_function)(e /* rdi */, index /* rsi */, name /* rdx */);
}
```

if we overwrite process_function's address with the address of the gadget `mov %rsi, %rsp; ret`, then the beginning of the input should contain...

A. the shellcode to run

B. an ROP chain to run

C. the address of shellcode (or existing function) in decimal

D. the address of the ROP chain to run written out in decimal

E. the address of a RET instruction written out in decimal

explanation

```
gets(e->input); /* can overwrite process_function */
scanf("%ld,%s", &index, &name[0]); /* expects <decimal number>,<string> */
(e->process_function)(e /* rdi */, index /* rsi */, name /* rdx */);
```

`"1234,FOO........"` + addr of `mov %rsi, %rsp, ret`

arguments setup registers for gadget:

- %rdi (irrelevant) is "1234,FOO..." (copy in e)
- %rsi is 1234 (from scanf)
- %rdx (irrelevant) is "FOO..." (pointer to name)

mov in gadget: %rsi (1234) becomes %rsp

ret in gadget: read pointer at 1234, set %rsp to 1234 + 8

- jump to next gadget (whose address should be stored at 1234)
- if that gadget returns, will read new return address from 1238

using function pointer overwrite (2)

```
struct Example {
    char input[1000];
    void (*process_function)(Example *, long, char *);
};
void vulnerable(struct Example *e) {
    long index;
    char name[1000];
    gets(e->input); /* can overwrite process_function */
    scanf("%ld,%s", &index, &name[0]); /* expects <decimal number>,<string> */
    (e->process_function)(e /* rdi */, index /* rsi */, name /* rdx */);
}
```

if we overwrite process_function's address with the address of the gadget `push %rdx; jmp *(%rdi)`, then the beginning of the input should contain...

A. the shellcode to run
B. an ROP chain to run
C. the address of shellcode (or existing function)
D. the address of the ROP chain
E. the address of a RET instruction

explanation (one option)

```
gets(e->input); /* can overwrite process_function */
scanf("%ld,%s", &index, &name[0]); /* expects <decimal number>,<string> */
(e->process_function)(e /* rdi */, index /* rsi */, name /* rdx */);
```

`"FOOBARBAZ........"` + addr of `push %rdx; jmp *(%rdi)`

arguments setup registers for gadget:

- %rdi is "FOOBARBAZ...." (copy in e)
- %rsi (irrelevant) is uninitialized? (scanf failed)
- %rdx (irrelevant) is uninitialized? (scanf failed)

push in gadget: top of stack becomes copy of uninit. value

jmp in gadget

- interpret "FOOBARBA" as 8-byte address
- jump to that address

explanation (unlikely alternative?)

```
gets(e->input); /* can overwrite process_function */
scanf("%ld,%s", &index, &name[0]); /* expects <decimal number>,<string> */
(e->process_function)(e /* rdi */, index /* rsi */, name /* rdx */);
```

`"1234567890,FOO........"` + addr of `push %rdx; jmp *(%rdi)`

arguments setup registers for gadget:

- %rdi is address of string "12345678,FOO..." (copy in e)
- %rsi is 12345678
- %rdx is address of string "FOO..." (copy in name)

push in gadget: top of stack becomes address of "FOO..."

jmp in gadget

- interpret *ASCII encoding of "12345678"* (???) as 8-byte address
- jump to that address

can we get rid of gadgets? (1)

Onarlioglu et al, "G-Free: Defeating Return-Oriented Programming through Gadget-Less Binaries" (2010)

two parts:

- get rid of unintended jmp, ret instructions
- add stack canary-like checks to jmp, ret instructions

hope: no *useful* gadgets b/c of canary-like checks

- all gadgets should be useless without a secret value?
- still vulnerable to information leaks

overhead is not low:

- 20–30% (!) space overhead
- 0–6% time overhead

no unintended jmp/ret (1)

```
addl $0xc2, %eax   ⇒   addl $0xc1, %eax
                       inc %eax
```

`addl $0xc2, %eax`: `05 c2 00 00 00`

problem: `c2 00 00`: variant of ret instruction

paper's proposed fix: change the constant

no unintended jmp/ret (1)

```
addl $0xc2, %eax   ⇒   addl $0xc1, %eax
                       inc %eax
```

`addl $0xc2, %eax`: `05 c2 00 00 00`

problem: `c2 00 00`: variant of ret instruction

paper's proposed fix: change the constant

no unintended jmp/ret (2)

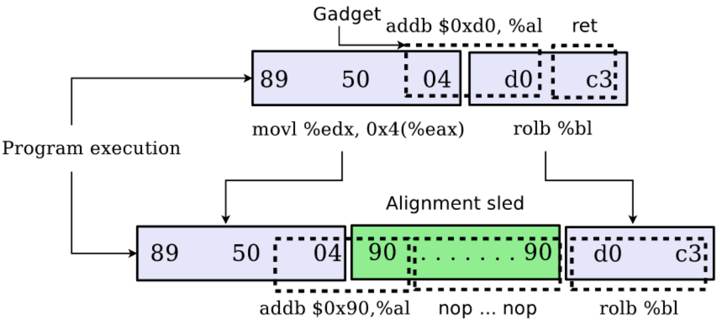


Figure 2: Application of an alignment sled to prevent executing an unaligned `ret` (`0xc3`) instruction

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* $\times$ `0x1000`

   randomization disabled? *random number* $= 0$

16 GB range!

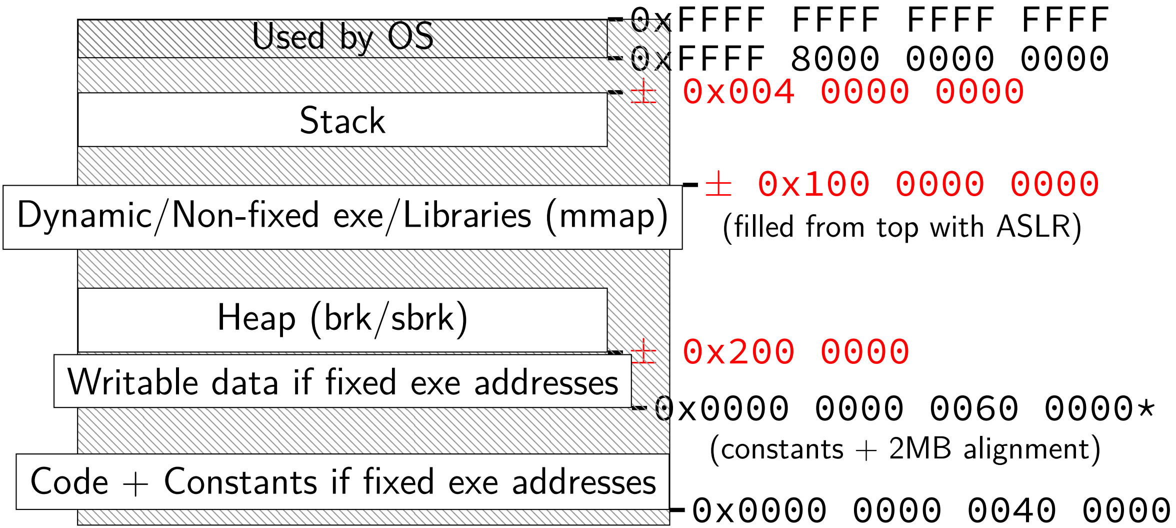
Linux stack randomization (x86-64)

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

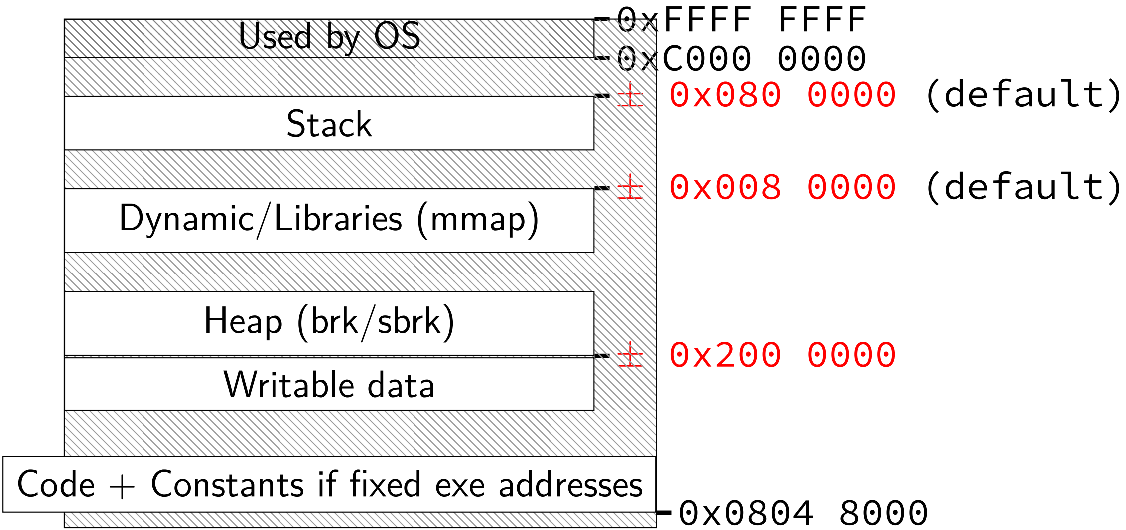
   randomization disabled? *random number* $= 0$

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

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 0x0000
     filesz 0x0000000000000620 memsz 0x0000000000000620 flags r--
LOAD off    0x0000000000001000 vaddr 0x0000000000001000 paddr 0x0000
     filesz 0x0000000000000205 memsz 0x0000000000000205 flags r-x
LOAD off    0x0000000000002000 vaddr 0x0000000000002000 paddr 0x0000
     filesz 0x0000000000000150 memsz 0x0000000000000150 flags r--
LOAD off    0x0000000000002db8 vaddr 0x0000000000003db8 paddr 0x0000
     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@pl
```

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)

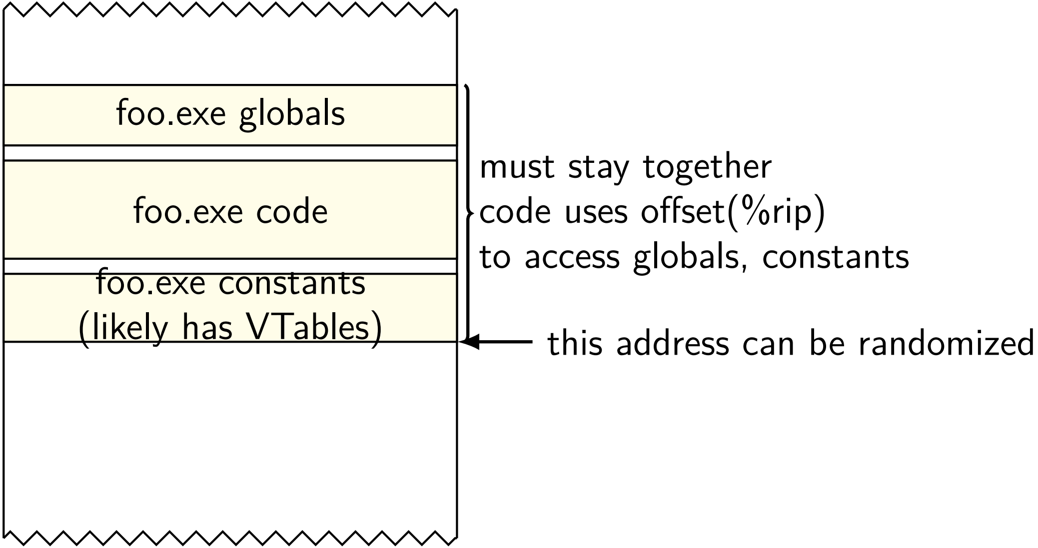
```
class Foo {
    virtual const char *bar() { ... }
};
...
Foo *f = new Foo;
char *p = new char[1024];
printf("%s\n", f);
```

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

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

exes, libraries stay together



backup slides

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

programs as weird machines

ROP, format strings: mini machine language

set of instructions including:

- reading/writing values from memory
- in ROP: flow control
- ...and we'll see more

can be viewed as virtual machine with unusual instruction set

can be analyzed using CS 3102 techniques

- what can it compute?

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

ROPgadget

ROPgadget: tool that does this

```
$ ROPgadget.py --binary /bin/ls
....
0x000000000000f09d : xor r8d, r8d ; cmp rcx, rsi ; jb 0xf0b9 ;
0x0000000000012a22 : xor r8d, r8d ; jmp 0x11fee
0x0000000000013d86 : xor r8d, r8d ; jmp 0x137a8
0x000000000001421a : xor r8d, r8d ; jmp 0x141b0
0x0000000000006aa1 : xor r8d, r8d ; jmp 0x69d5
0x00000000000099f0 : xor r8d, r8d ; jmp 0x931d
0x000000000000e6d0 : xor r8d, r8d ; mov rax, r8 ; ret
0x00000000000127a7 : xor r8d, r8d ; xor esi, esi ; jmp 0x11fee
0x000000000000e640 : xor r8d, r8d ; xor esi, esi ; jmp 0xe66a
0x000000000001435d : xor r9d, r9d ; jmp 0x141b0
0x0000000000008a03 : xor r9d, r9d ; xor r12d, r12d ; jmp 0x873
0x0000000000014217 : xor r9d, r9d ; xor r8d, r8d ; jmp 0x141b0

Unique gadgets found: 6472
```

common, reusable ROP sequences

open a command-line

- `ROPchain.py --binary example --ropchain` tries to do this

make memory executable + jump

- generally: just do enough to ignore write XOR execute

often only depend on memory locations in shared library

ROPgadget.py –ropchain (works)

```
ROPgadget.py --binary /lib/x86_64-linux-gnu/libc.so.6 \
             --offset 0x10000000 --ropchain
...
        #!/usr/bin/env python2
        # execve generated by ROPgadget

        from struct import pack

        # Padding goes here
        p = ''
        p += pack('<Q', 0x00000000101056fd) # pop rdx ; pop rcx ; pop rbx ; ret
        p += pack('<Q', 0x00000000101eb1a0) # @ .data
        p += pack('<Q', 0x4141414141414141) # padding
        p += pack('<Q', 0x4141414141414141) # padding
        p += pack('<Q', 0x000000001004a550) # pop rax ; ret
        p += '/bin//sh'
        p += pack('<Q', 0x00000000100374b0) # mov qword ptr [rdx], rax ; ret
...
```

ROPgadget.py –ropchain (does not work?)

```
ROPgadget.py --binary /bin/ls --ropchain
...
ROP chain generation
===========================================================

- Step 1 -- Write-what-where gadgets

        [+] Gadget found: 0x7694 mov byte ptr [rax], 0xa ; pop rbx ; pop rbp ; pop r12 ; ret
        [-] Can't find the 'pop rax' gadget. Try with another 'mov [reg], reg'

        [-] Can't find the 'mov qword ptr [r64], r64' gadget
...
```