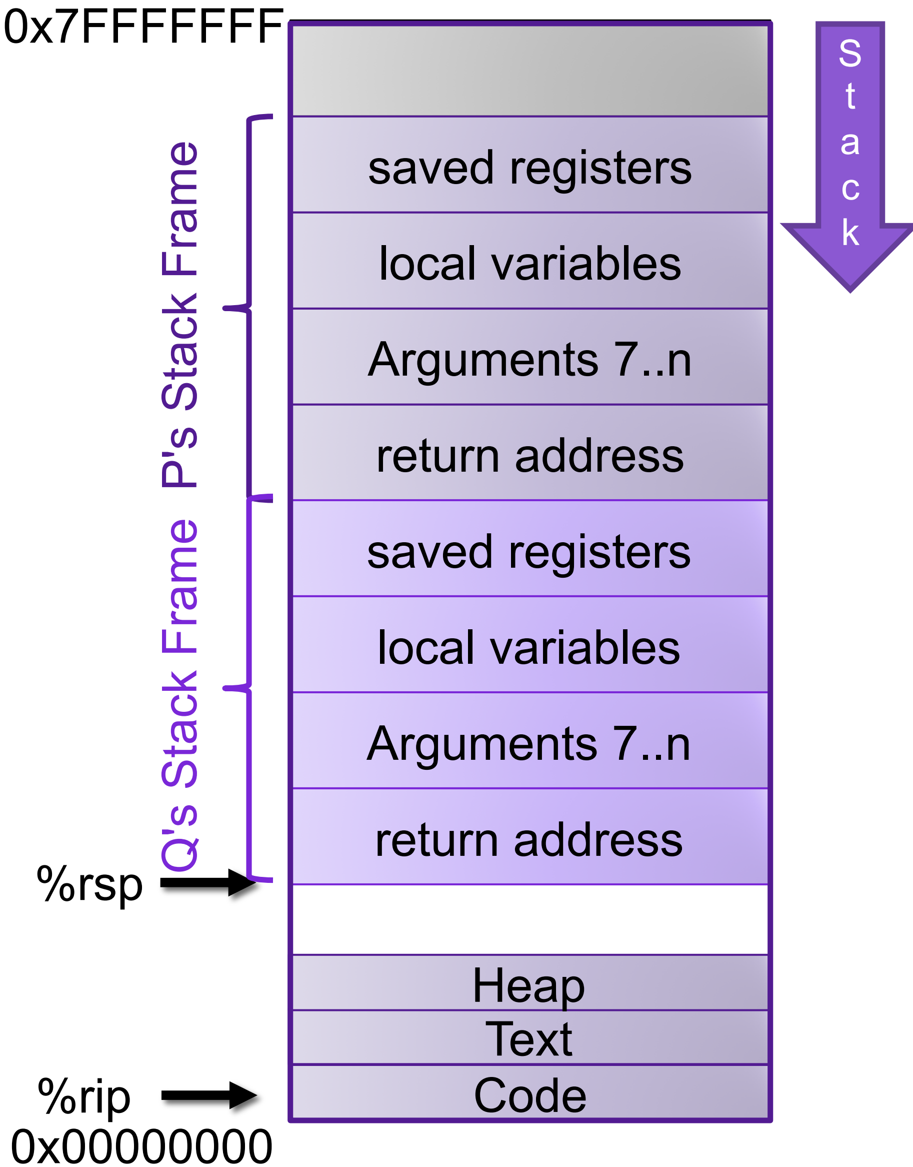
Lecture 10: Buffer Overflows (cont'd)

CS 105

February 24, 2020

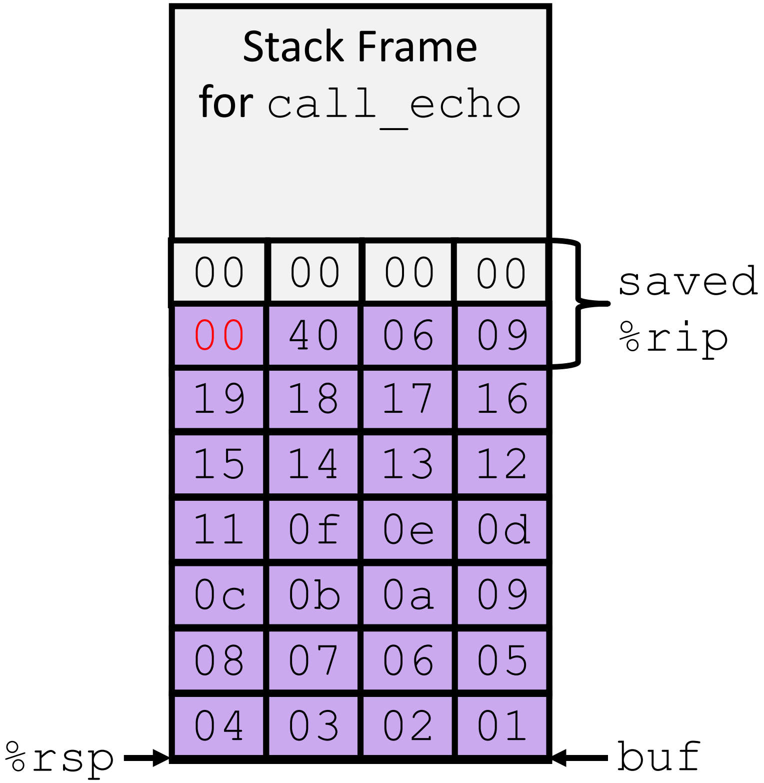
Review: Stack Frames

- Each function called gets a stack frame
- Passing data:
  - calling procedure P uses registers (and stack) to provide parameters to Q.
  - Q uses register %rax for return value
- Passing control:
  - **`call <proc>`**
    - Pushes return address (current **`%rip`**) onto stack
    - Sets **`%rip`** to first instruction of proc
  - **`ret`**
    - Pops return address from stack and places it in **`%rip`**
- Local storage:
  - allocate space on the stack by decrementing stack pointer, deallocate by incrementing



Review: Buffer Overflow Attack

- Most common form of memory reference bug
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack



Stack Frame for `call_echo`

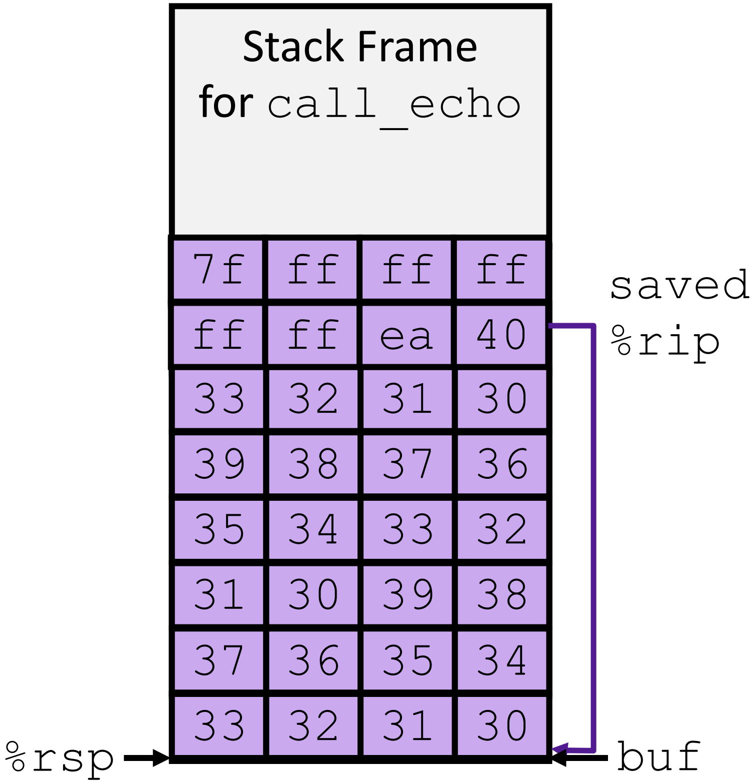
00	00	00	00
00	40	06	09
19	18	17	16
15	14	13	12
11	0f	0e	0d
0c	0b	0a	09
08	07	06	05
04	03	02	01

```
/* Echo Line */
void echo()
{
    char buf[4];
    gets(buf);
    puts(buf);
}
```

```
echo:
  subq  $0x18, %rsp
  movq  %rsp, %rdi
  call  gets
  call puts
  addq  $0x18, %rsp
  ret
```

Stack Smashing

- Idea: fill the buffer with bytes that will be interpreted as code
- Overwrite the return address with address of the beginning of the buffer



7f	ff	ff	ff
ff	ff	ea	40
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

```
/* Echo Line */
void echo()
{
    char buf[4];
    gets(buf);
    puts(buf);
}
```

```
echo:
  subq  $18, %rsp
  movq  %rsp, %rdi
  call  gets
  call puts
  addq  $18, %rsp
  ret
```

3. System-level Protection: Memory Tagging

Code Reuse Attacks

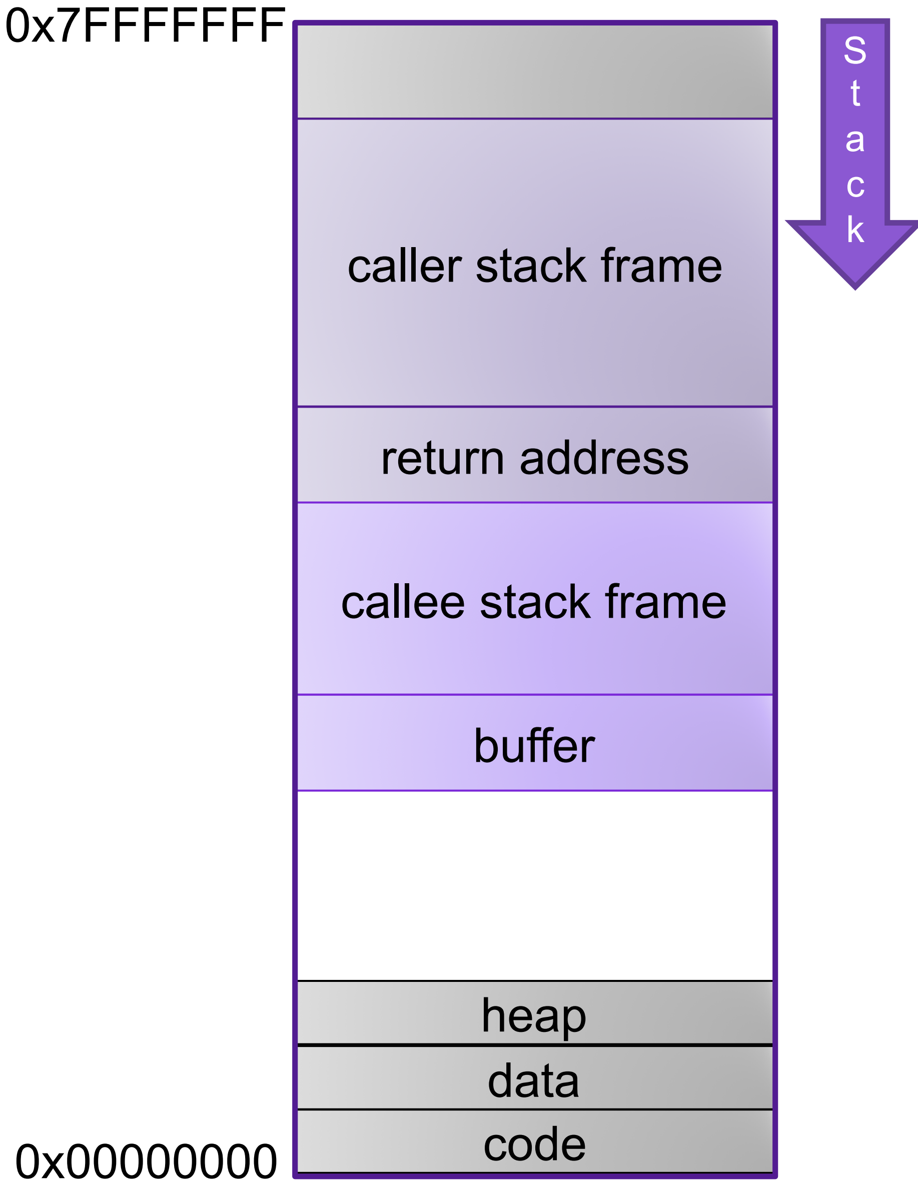
- Key idea: execute instructions that already exist

- Defeats memory tagging defenses

- Examples:
  1. return to a function in the current program
  2. return to a library function (e.g., return-into-`libc`)
  3. return to some other instruction (return-oriented programming)

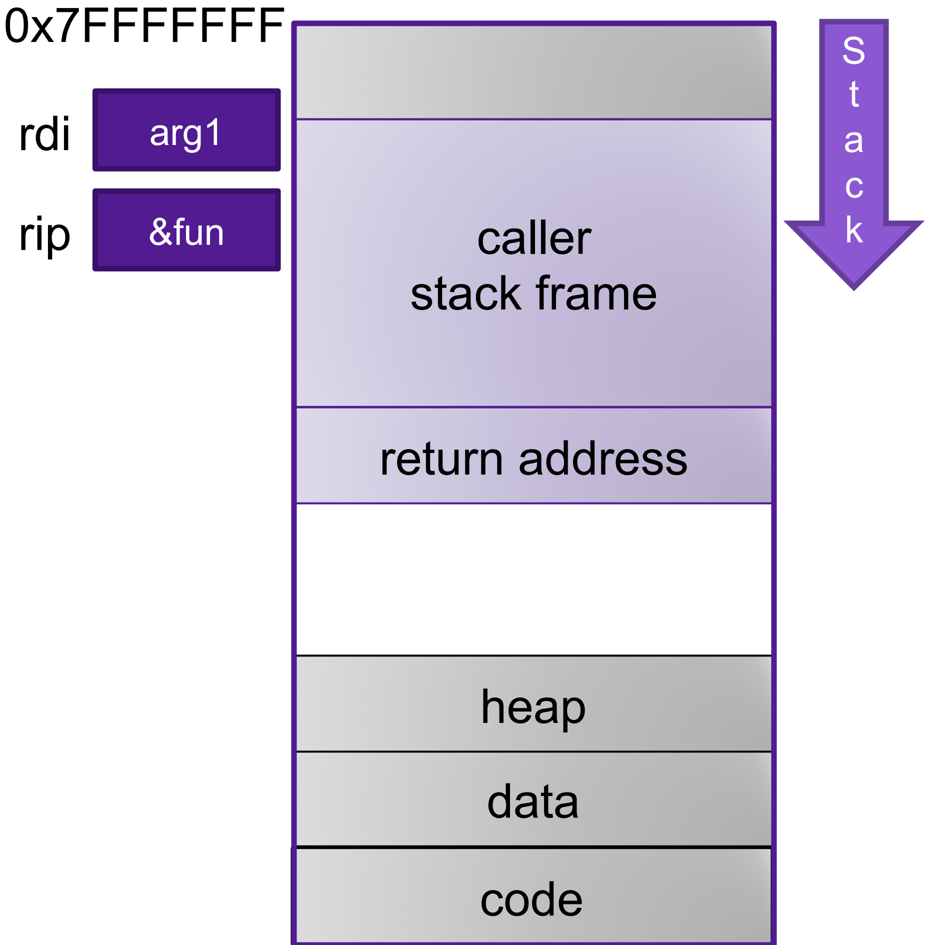
Returning to a function

- Overwrite the saved return address with the location of a function in the current program

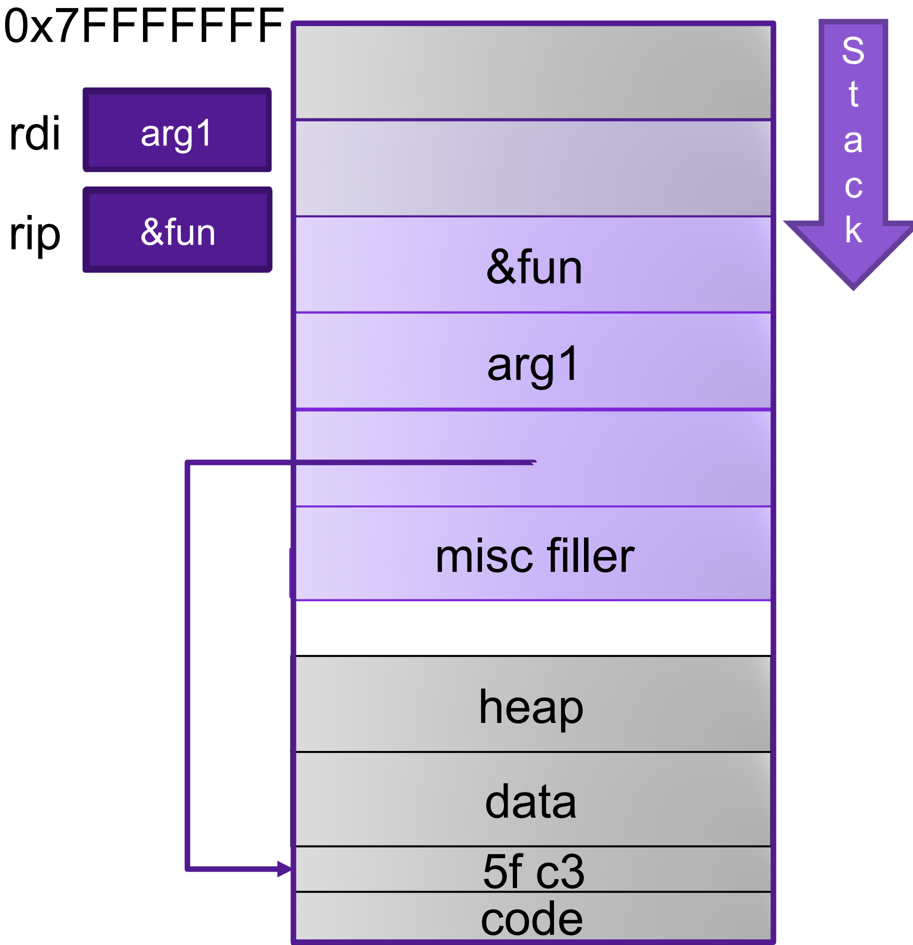


Handling Arguments

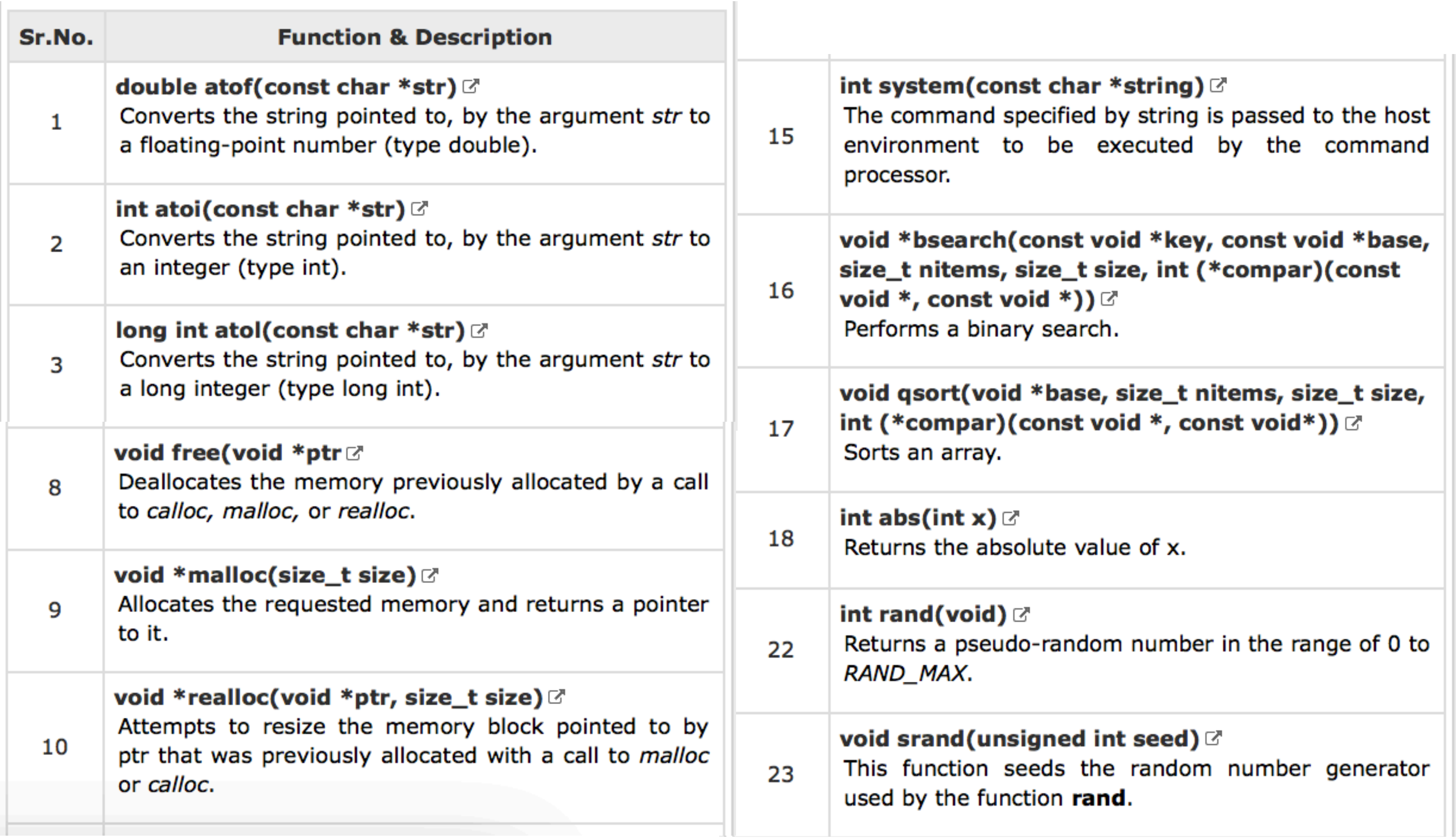
what function expects
when it is called…



overflow with argument



Return-into-libc



Sr.No.	Function & Description
1	**double atof(const char *str)** Converts the string pointed to, by the argument *str* to a floating-point number (type double).
2	**int atoi(const char *str)** Converts the string pointed to, by the argument *str* to an integer (type int).
3	**long int atol(const char *str)** Converts the string pointed to, by the argument *str* to a long integer (type long int).
8	**void free(void *ptr** Deallocates the memory previously allocated by a call to *calloc, malloc,* or *realloc*.
9	**void *malloc(size_t size)** Allocates the requested memory and returns a pointer to it.
10	**void *realloc(void *ptr, size_t size)** Attempts to resize the memory block pointed to by ptr that was previously allocated with a call to *malloc* or *calloc*.
15	**int system(const char *string)** The command specified by string is passed to the host environment to be executed by the command processor.
16	**void *bsearch(const void *key, const void *base, size_t nitems, size_t size, int (*compar)(const void *, const void *))** Performs a binary search.
17	**void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*))** Sorts an array.
18	**int abs(int x)** Returns the absolute value of x.
22	**int rand(void)** Returns a pseudo-random number in the range of 0 to *RAND_MAX*.
23	**void srand(unsigned int seed)** This function seeds the random number generator used by the function **rand**.

ASCII Armoring

- Make sure all system library addresses contain a null byte (0x00).
- Can be done by placing this code in the first 0x01010101 bytes of memory

Properties of x86 Assembly

- variable length instructions
- not word aligned
- dense instruction set

Gadgets

```
void setval(unsigned *p) {
  *p = 3347663060u;
}
```

```
<setval>:
4004d9: c7 07 d4 48 89 c7  movl $0xc78948d4,(%rdi)
4004df: c3                 ret
```

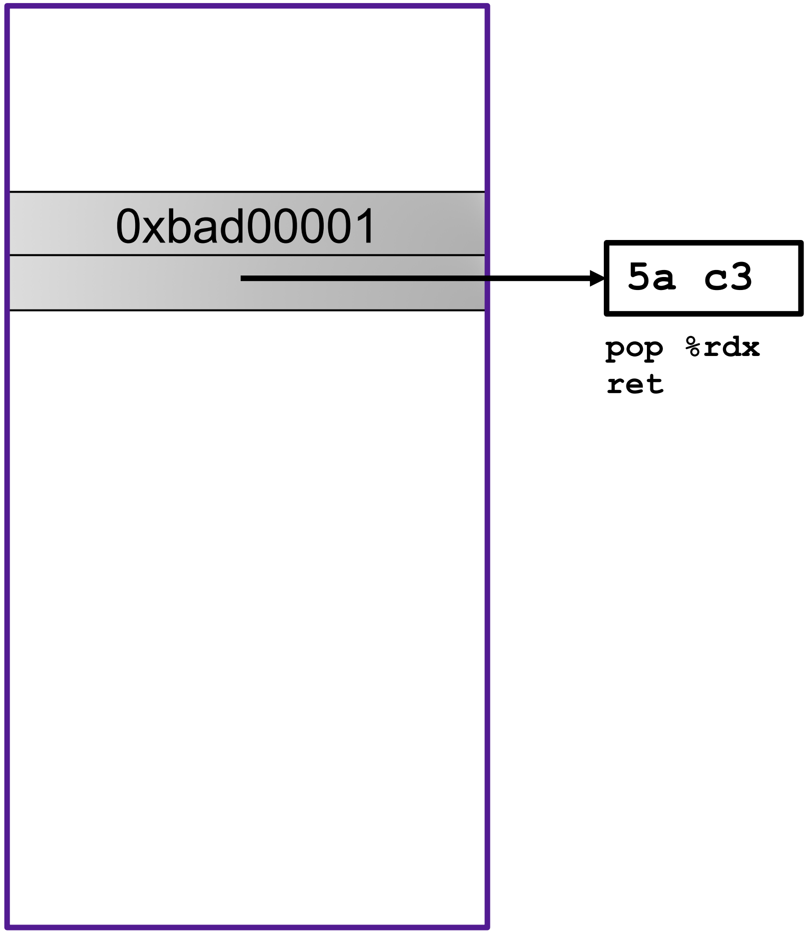
gadget address: `0x4004dc`

encodes:
```
movq %rax, %rdi
ret
```

executes: `%rdi <- %rax`

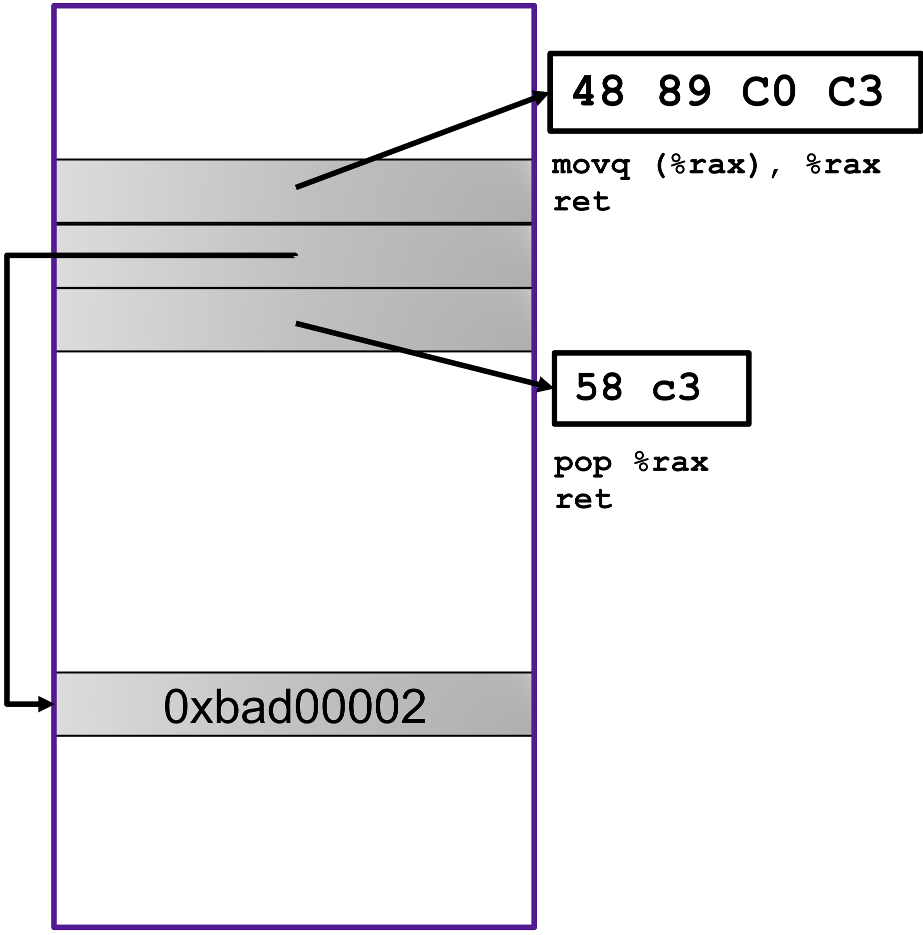
Example Gadgets

Load Constant



```
pop %rdx
ret
```

Load from memory



```
movq (%rax), %rax
ret
```

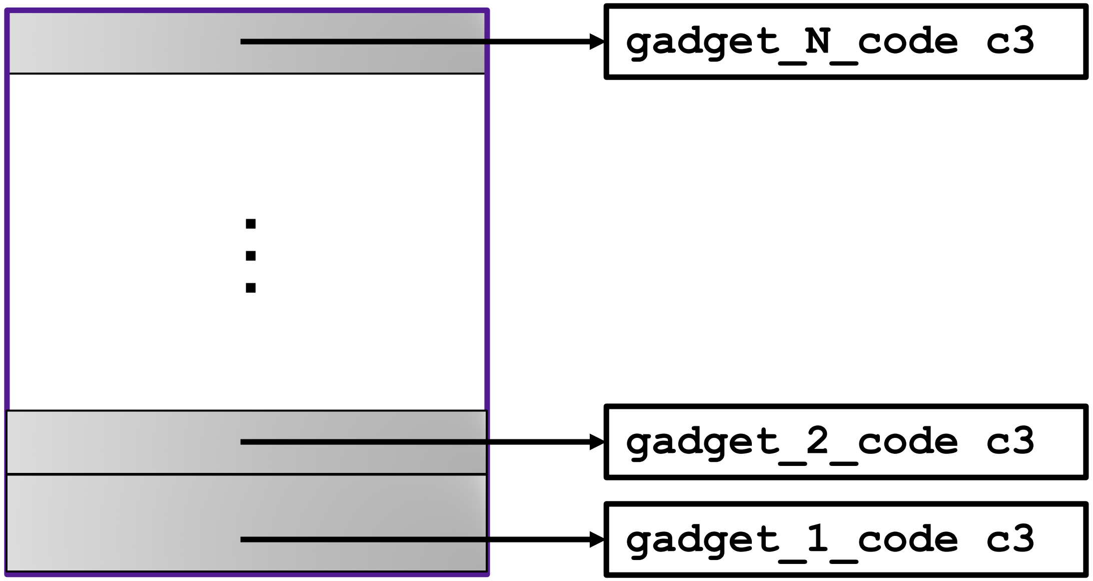
```
pop %rax
ret
```

Return-oriented Programming

Return-oriented Programming

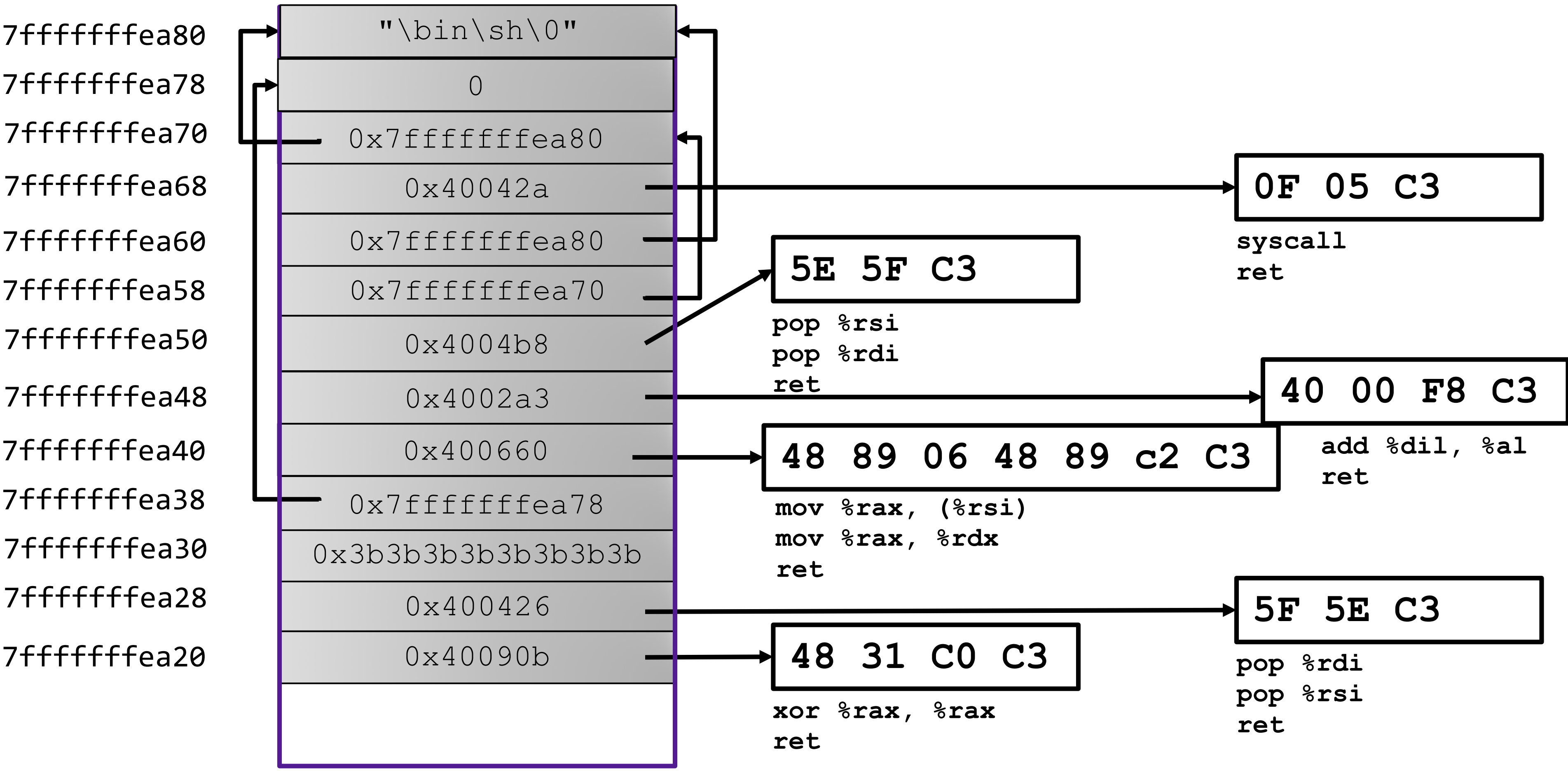
is a lot like a ransom note, but instead of cutting cut letters from magazines, you are cutting out instructions from .text segments

Return-oriented Programming



Final ret in each gadget sets pc (%rip) to beginning of next gadget code

Return-Oriented Shellcode



7fffffffea80	"\bin\sh\0"
7fffffffea78	0
7fffffffea70	0x7fffffffea80
7fffffffea68	0x40042a
7fffffffea60	0x7fffffffea80
7fffffffea58	0x7fffffffea70
7fffffffea50	0x4004b8
7fffffffea48	0x4002a3
7fffffffea40	0x400660
7fffffffea38	0x7fffffffea78
7fffffffea30	0x3b3b3b3b3b3b3b3b
7fffffffea28	0x400426
7fffffffea20	0x40090b

```
0F 05 C3
syscall
ret
```

```
5E 5F C3
pop %rsi
pop %rdi
ret
```

```
40 00 F8 C3
add %dil, %al
ret
```

```
48 89 06 48 89 c2 C3
mov %rax, (%rsi)
mov %rax, %rdx
ret
```

```
5F 5E C3
pop %rdi
pop %rsi
ret
```

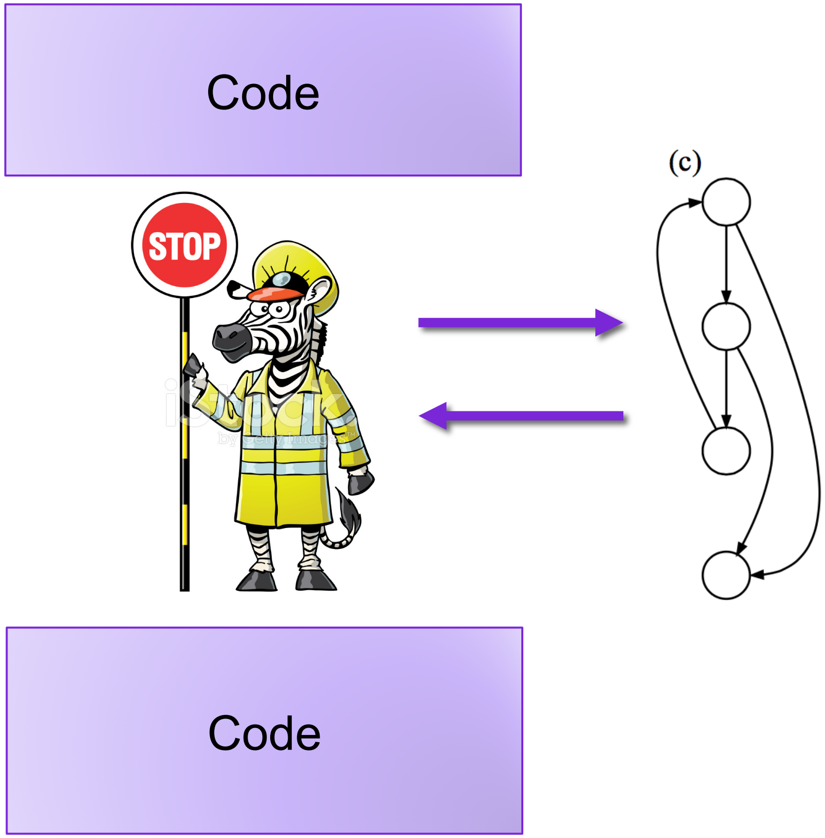
```
48 31 C0 C3
xor %rax, %rax
ret
```

Address Space Layout Randomization

Other defenses

Gadget Elimination

Control Flow Integrity


Code

Code

The state of the world

Defenses:

- high-level languages
- Stack Canaries
- Memory tagging
- ASLR
- continuing research and development…

But all they aren't perfect!