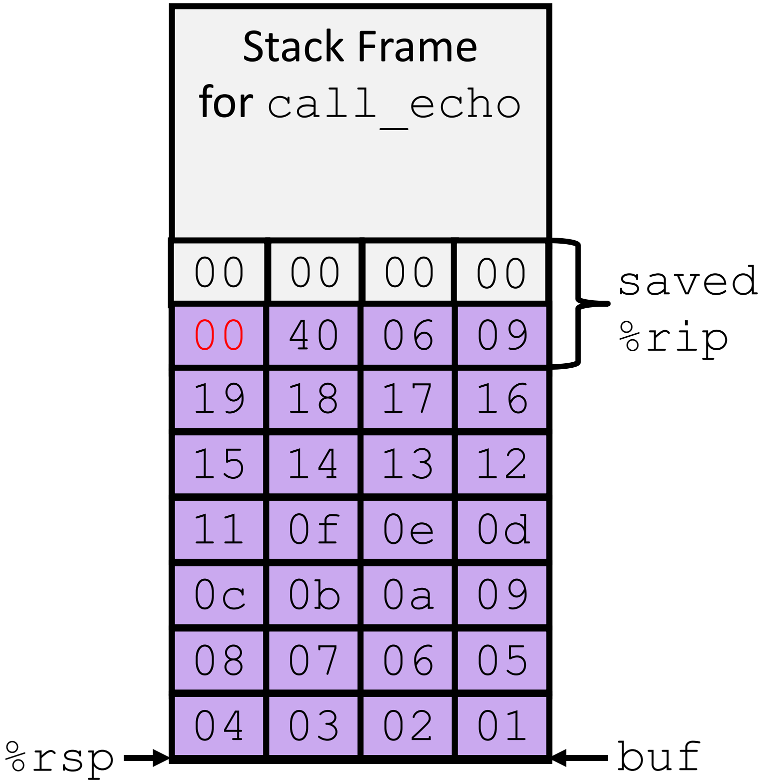
Lecture 10: Buffer Overflows (cont'd)

CS 105

Fall 2020

Review: Buffer Overflow Attack

- Idea: overwrite return address with address of instruction you want to execute next
  - If a string: use padding to fill up space between array and saved rip
  - Stack smashing: use padding to write program and jump there



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
```

saved %rip

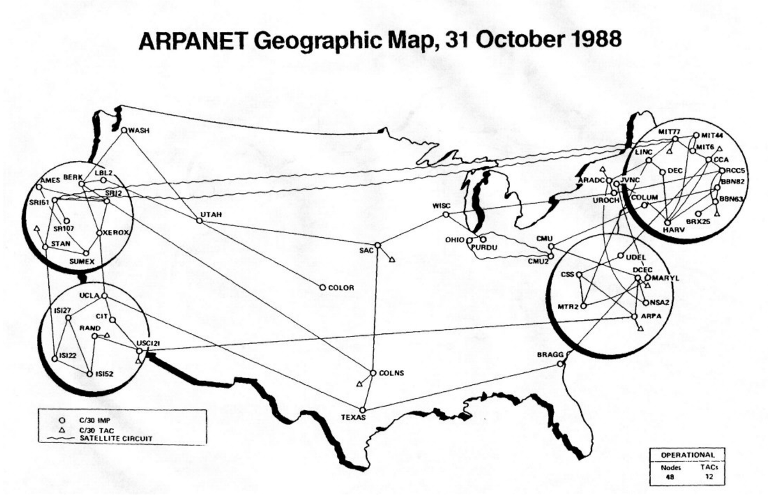
%rsp

buf

```
/* 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
```

Review: Buffer Overflow Examples

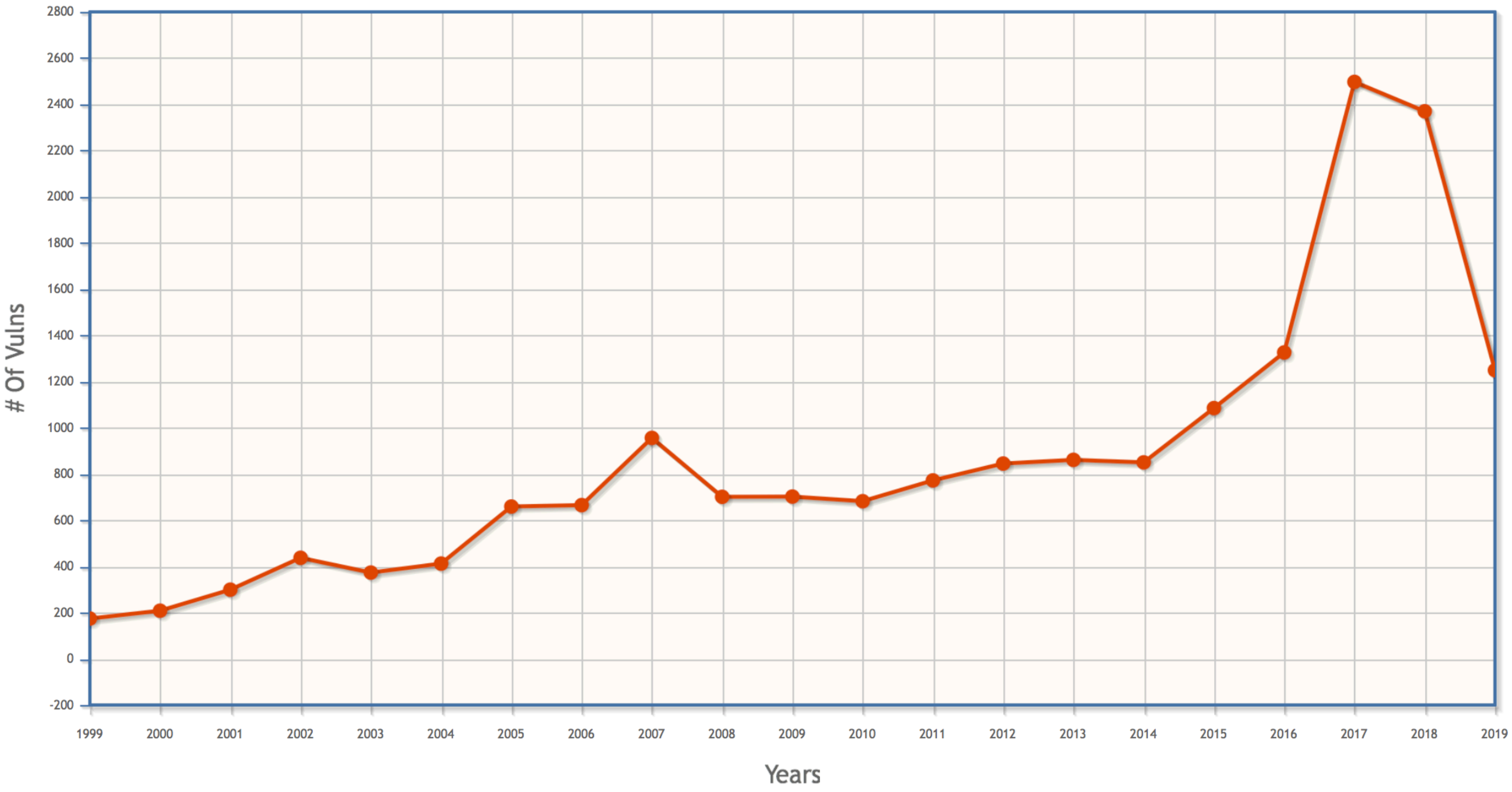


Defense #1: Avoid Overflow Vulnerabilities

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- For example, use library routines that limit string lengths
  - `fgets` instead of `gets`
  - `strncpy` instead of `strcpy`
  - Don’t use `scanf` with `%s` conversion specification (use `fgets` to read the string or use `%ns` where `n` is a suitable integer)
- Or use a high-level language

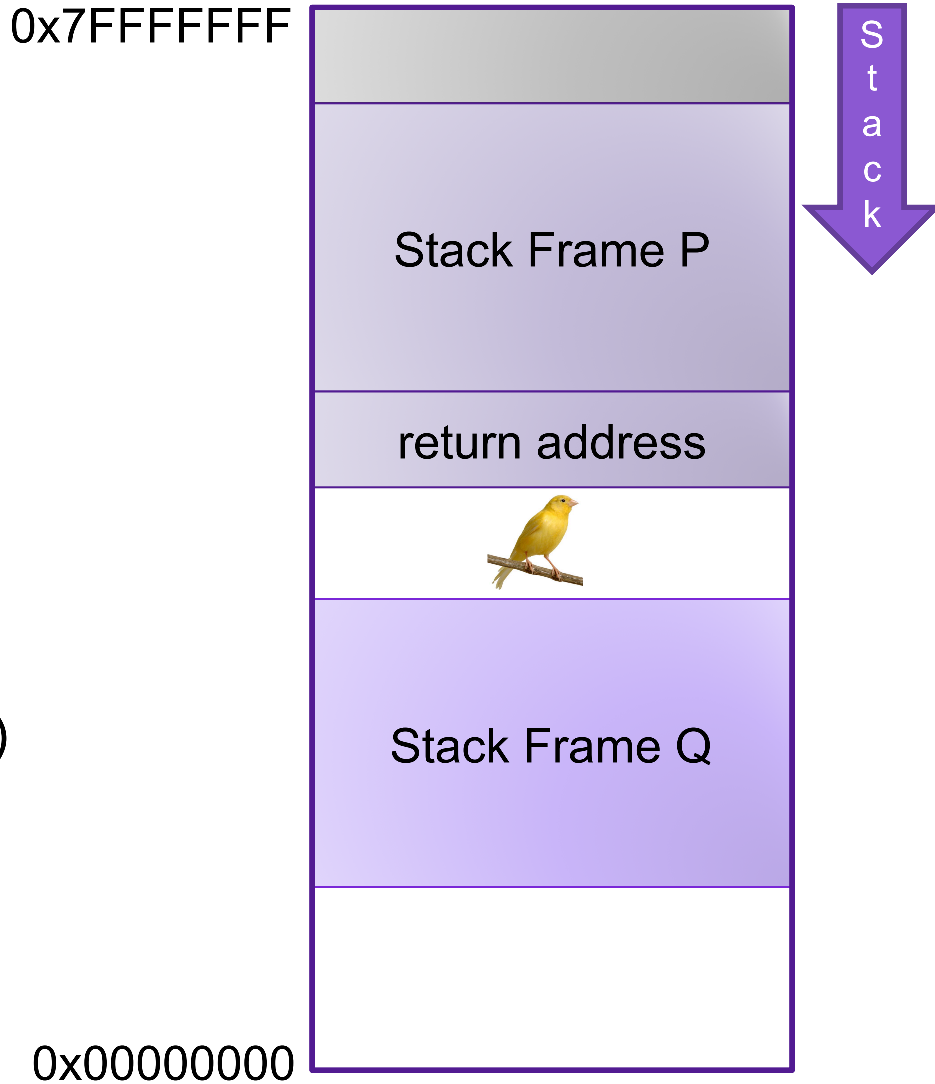
Buffer Overflow Vulnerabilities



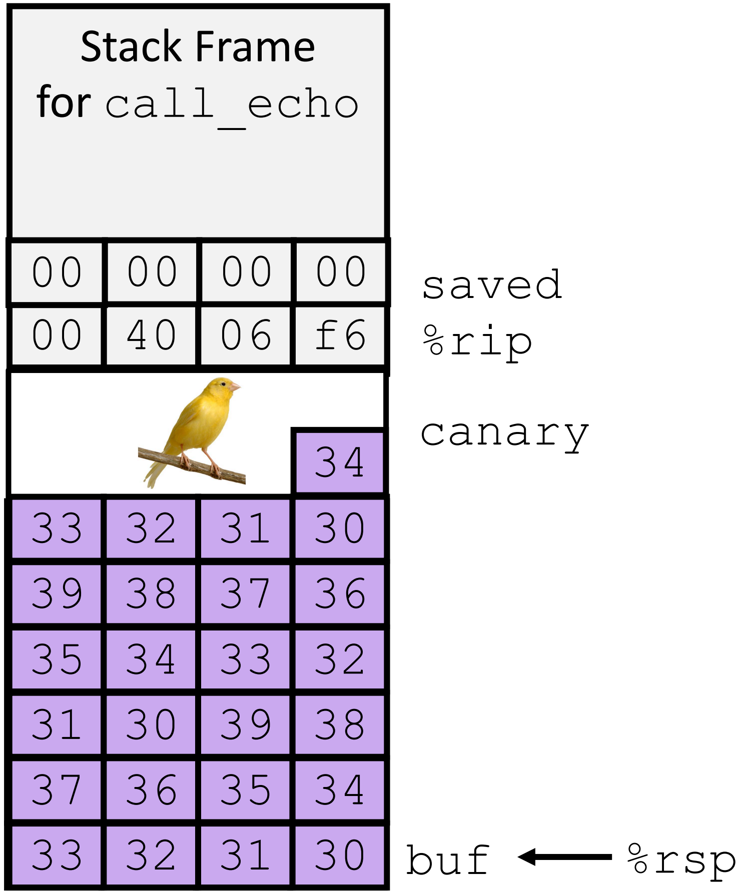
Defense #2: Compiler checks

- Idea
  - Place special value ("canary") on stack just beyond buffer
  - Check for corruption before exiting function

- GCC Implementation
  - `-fstack-protector`
  - Now the default (disabled earlier)



Stack Canaries



00	00	00	00
00	40	06	f6
			34
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

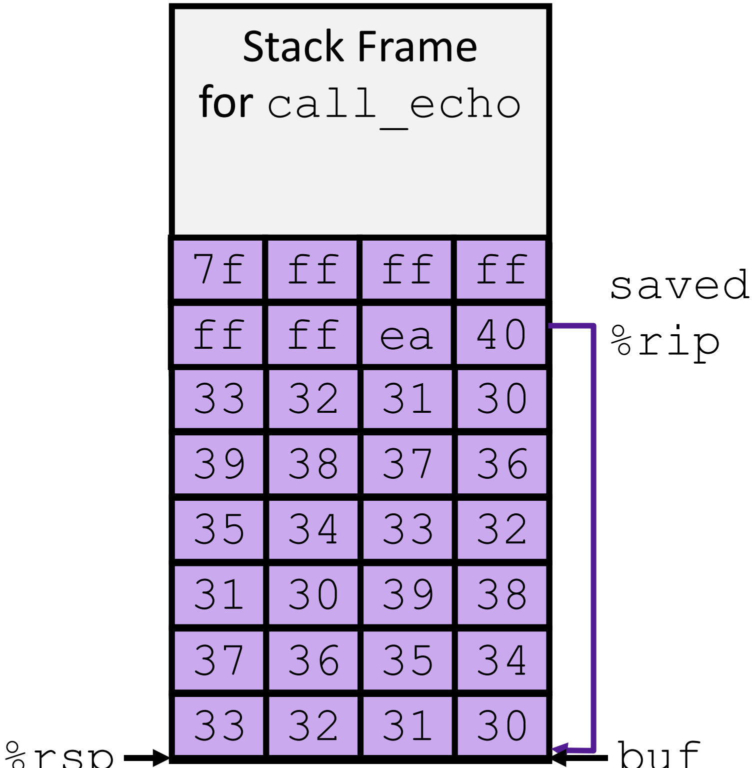
```
authenticate:
  pushq    %rbx
  subq     $16, %rsp
  movq     %rdi, %rbx
  movq     %fs:40, %rax
  movq     %rax, 8(%rsp)
  xorl     %eax, %eax
  movq     %rsp, %rdi
  call     gets
  movq     %rsp, %rsi
  movq     %rbx, %rdi
  call     strcmp
  testl    %eax, %eax
  sete     %al
  movq     8(%rsp), %rdx
  xorq     %fs:40, %rdx
  je       .L2
  call     __stack_chk_fail
.L2:
  movzbl   %al, %eax
  addq     $16, %rsp
  popq     %rbx
  ret
```

Exercise 1: Stack Canaries

- Which of the following would make a good stack canary?
  1. A secret, constant value
  2. A fixed sequence of common terminators (\0, EOF, etc.)
  3. A random number chosen each time the program is run

Review: Stack Smashing

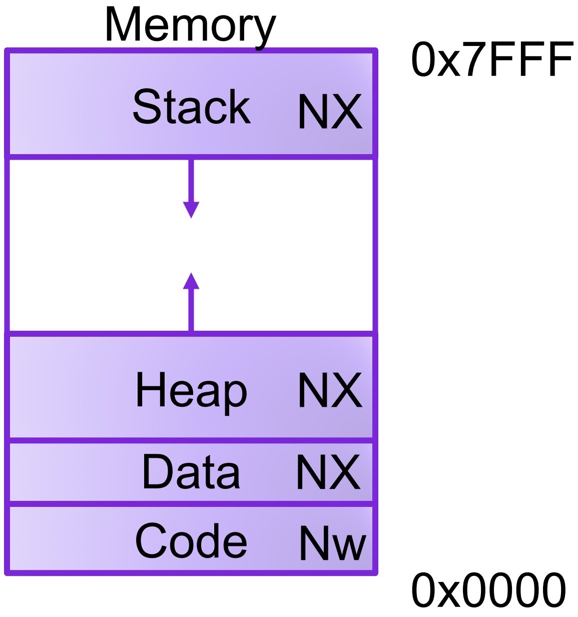
- Idea: fill the buffer with bytes that will be interpreted as code
- Overwrite the return address with address of the beginning of the buffer



```
/* 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
```

Defense #3: Memory Tagging



Memory

0x7FFF

Stack NX

Heap NX

Data NX

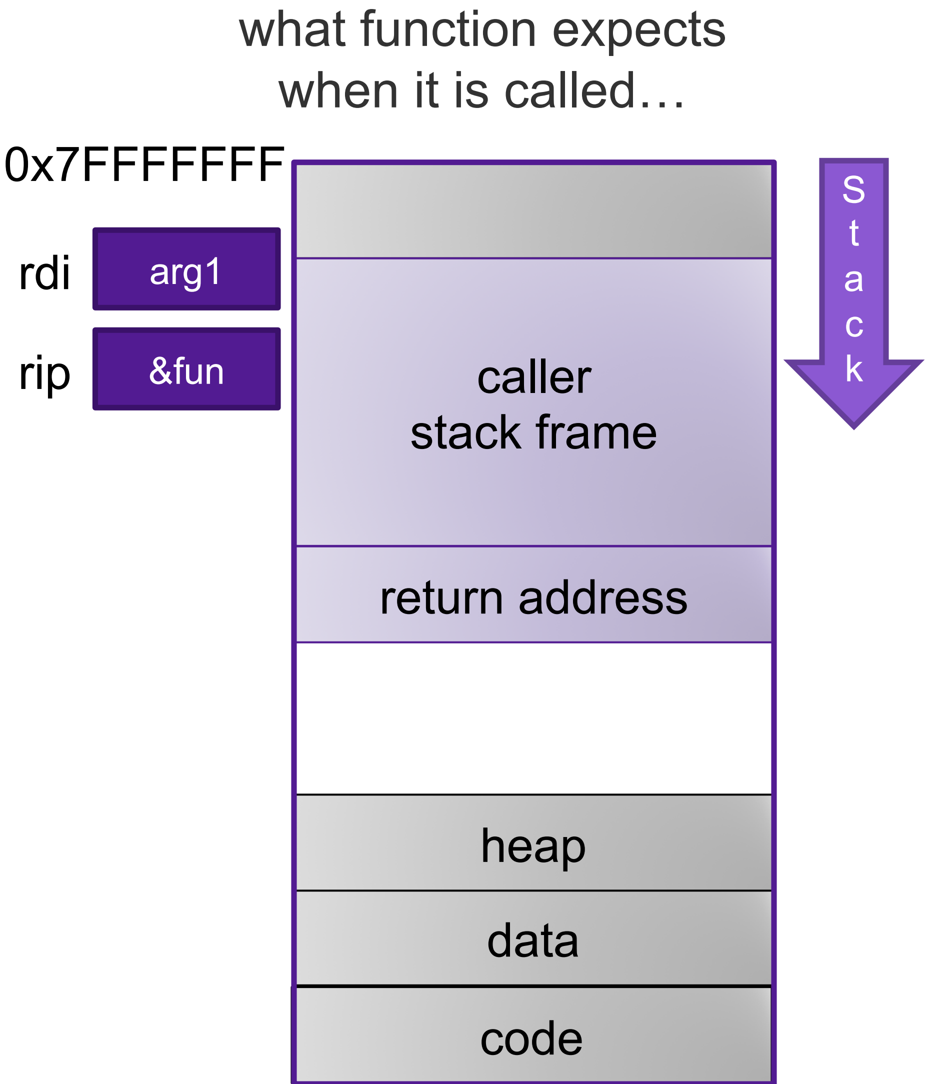
Code Nw

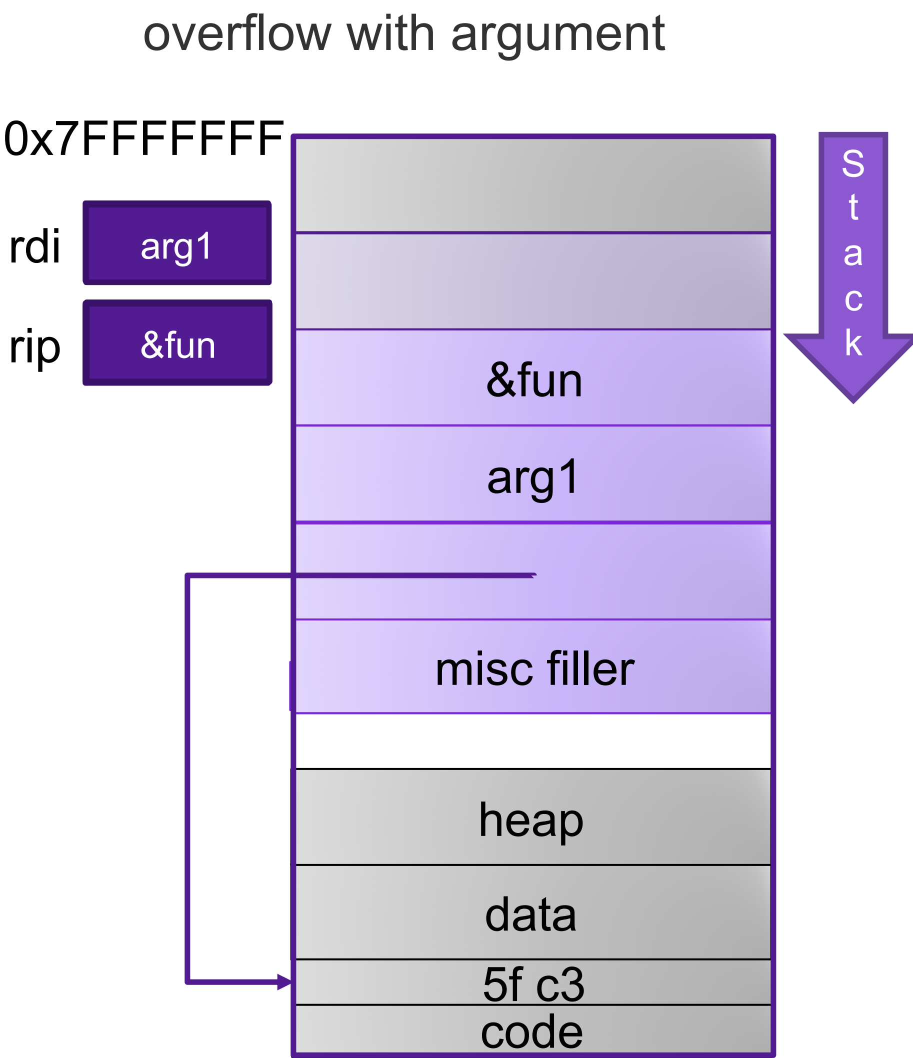
0x0000

Code Reuse Attacks

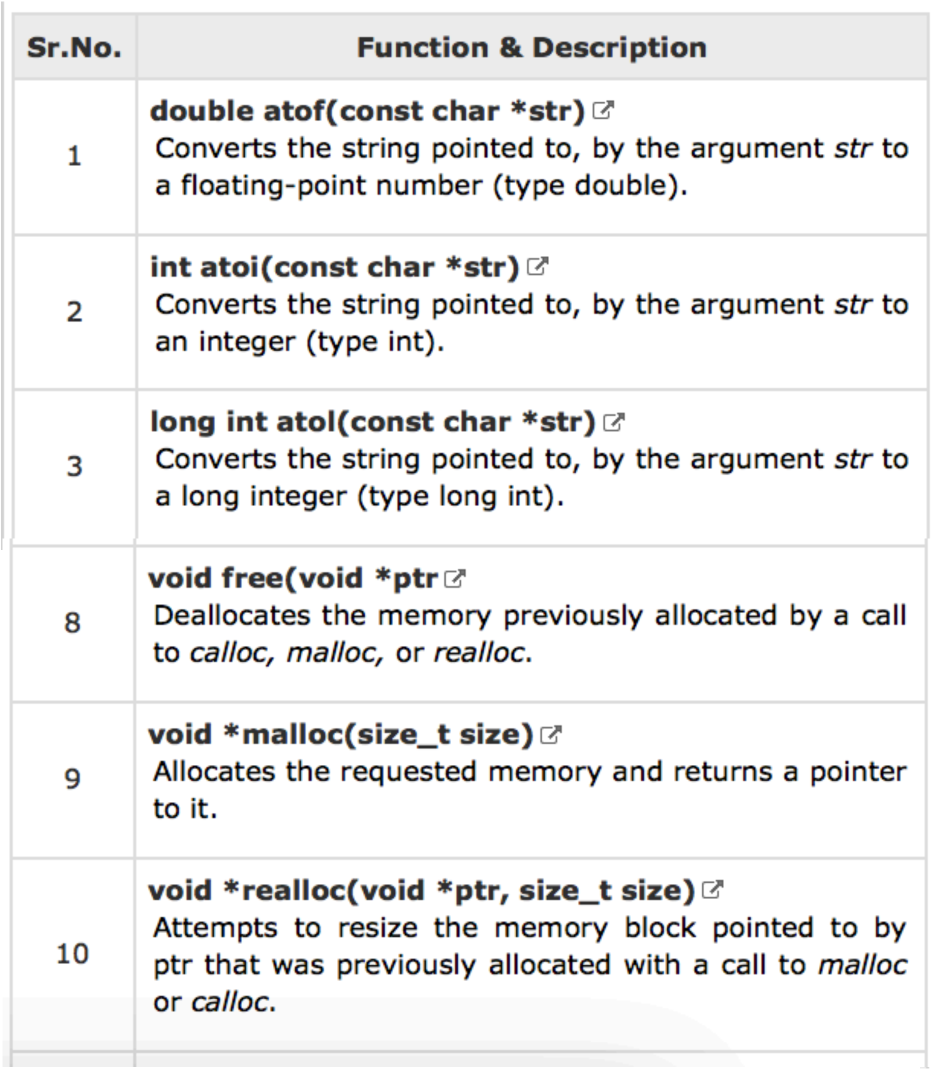
- Key idea: execute instructions that already exist
- Defeats memory tagging defenses
- Examples:
    1. return to a function or line in the current program
    2. return to a library function (e.g., return-into-`libc`)
    3. return to some other instruction (return-oriented programming)

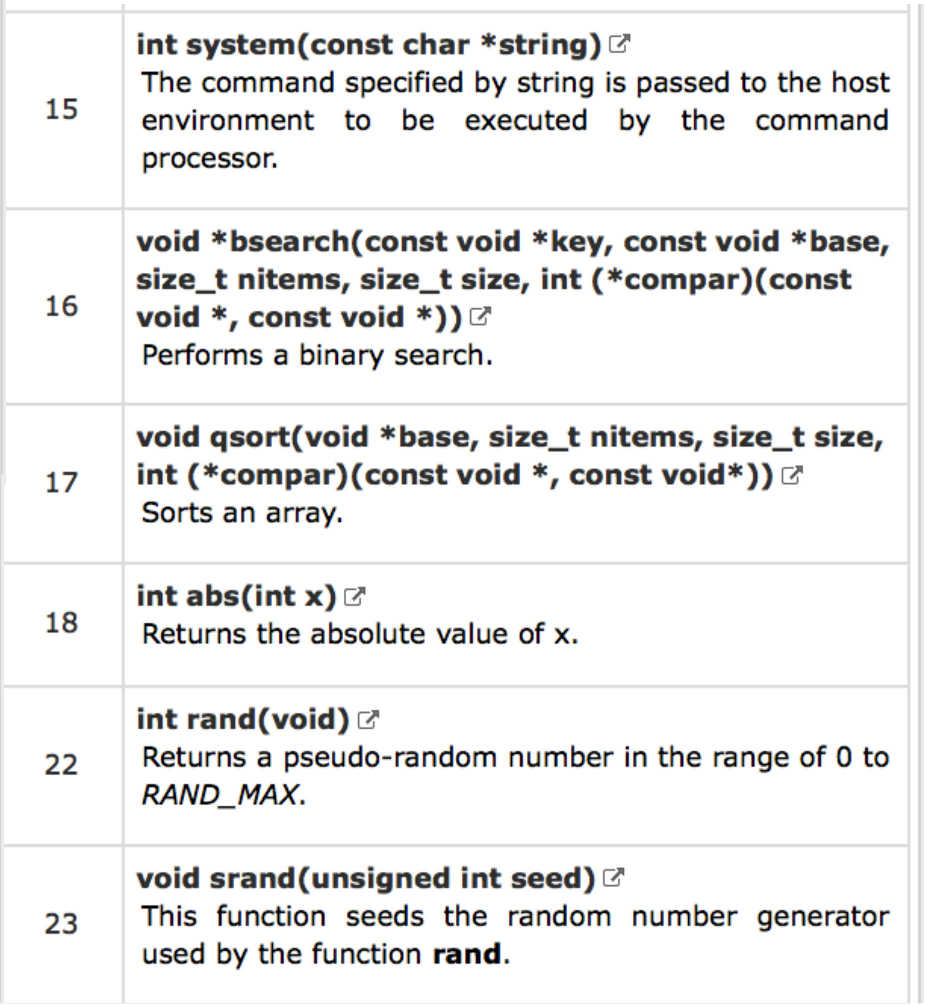
Handling Arguments





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

- lots of instructions
- 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
```

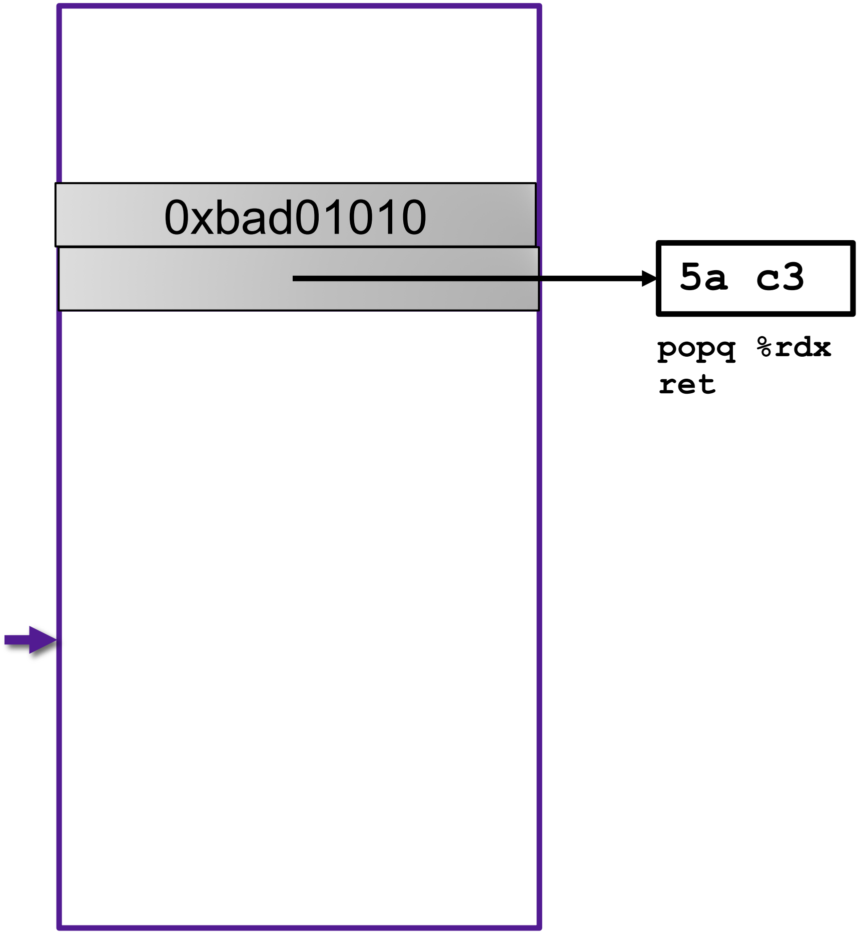
gadget address:	`0x4004dc`

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

executes:	`%rdi <- %rax`

Example Gadgets

Load Constant

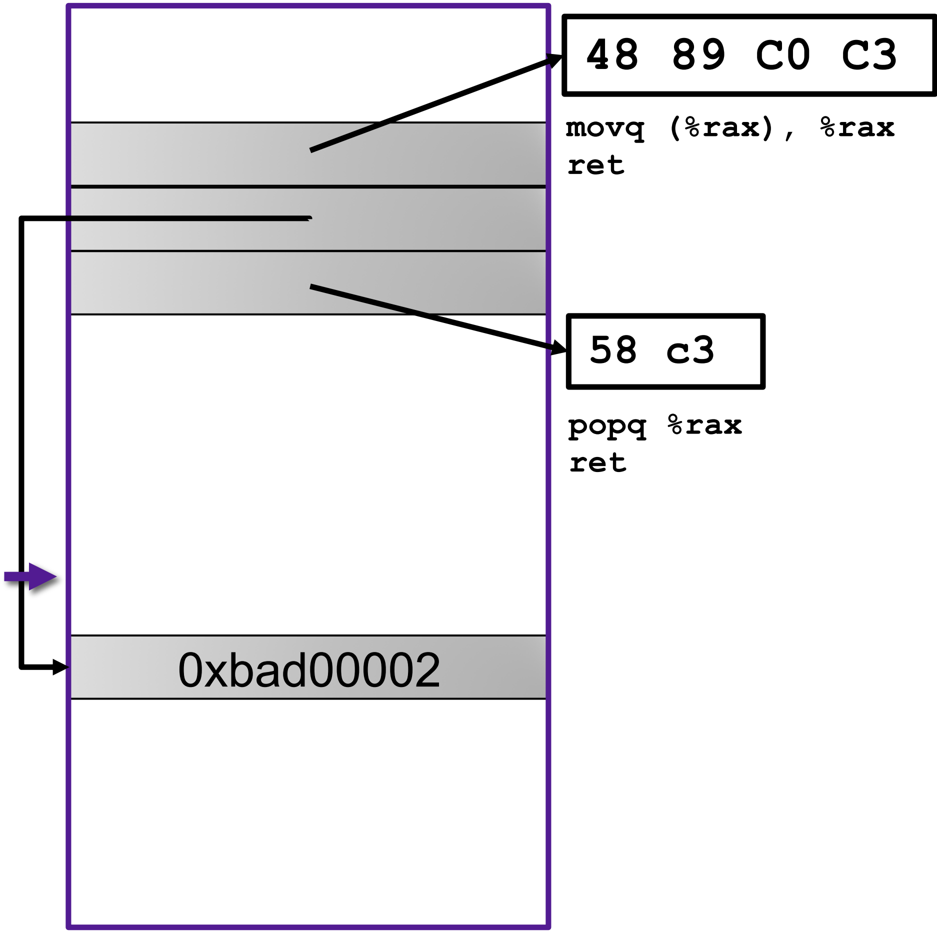


0xbad01010

5a c3

```
popq %rdx
ret
```

Load from memory



48 89 C0 C3

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

58 c3

```
popq %rax
ret
```

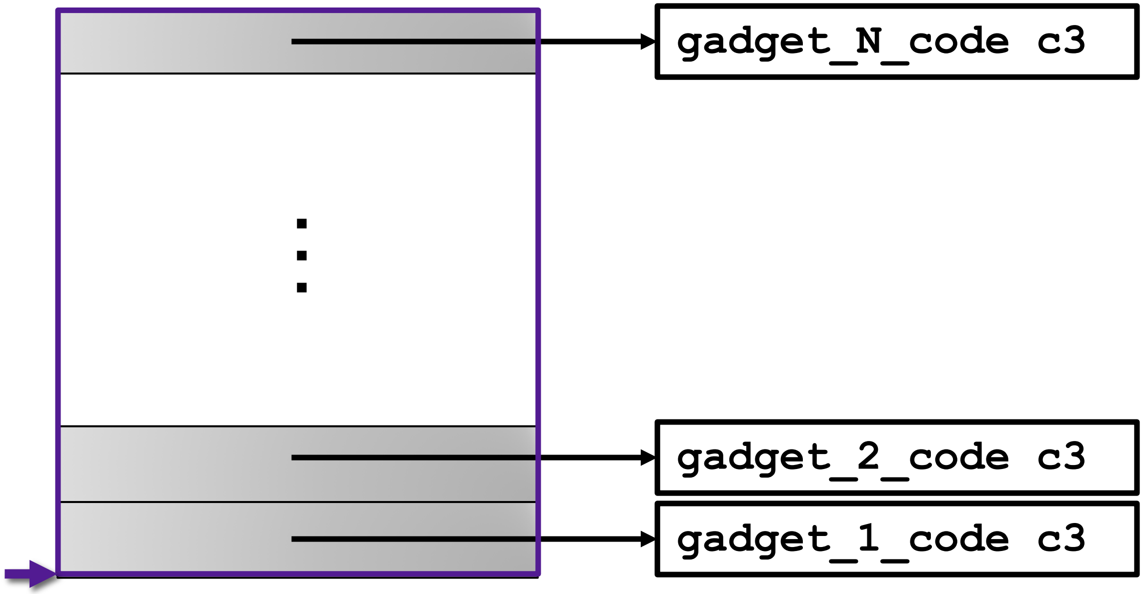
0xbad00002

Return-oriented Programming

Return-oriented Programming

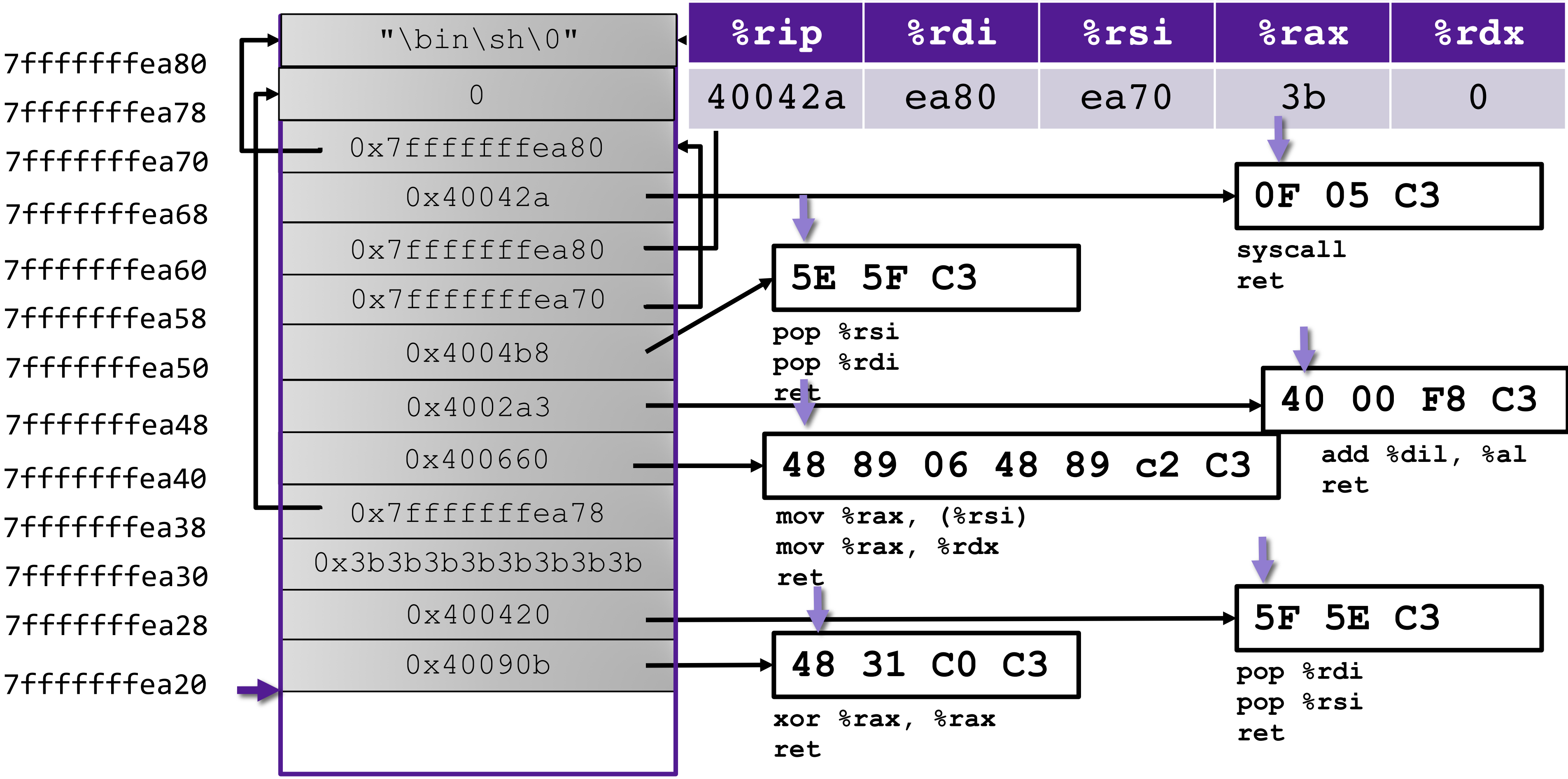
is a lot like a ransom note, but instead of cutting cut letters from magazines, you are cutting out instructions from .text segments

Return-oriented Programming



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

Return-Oriented Shellcode



%rip	%rdi	%rsi	%rax	%rdx
40042a	ea80	ea70	3b	0

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	0x400420
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
```

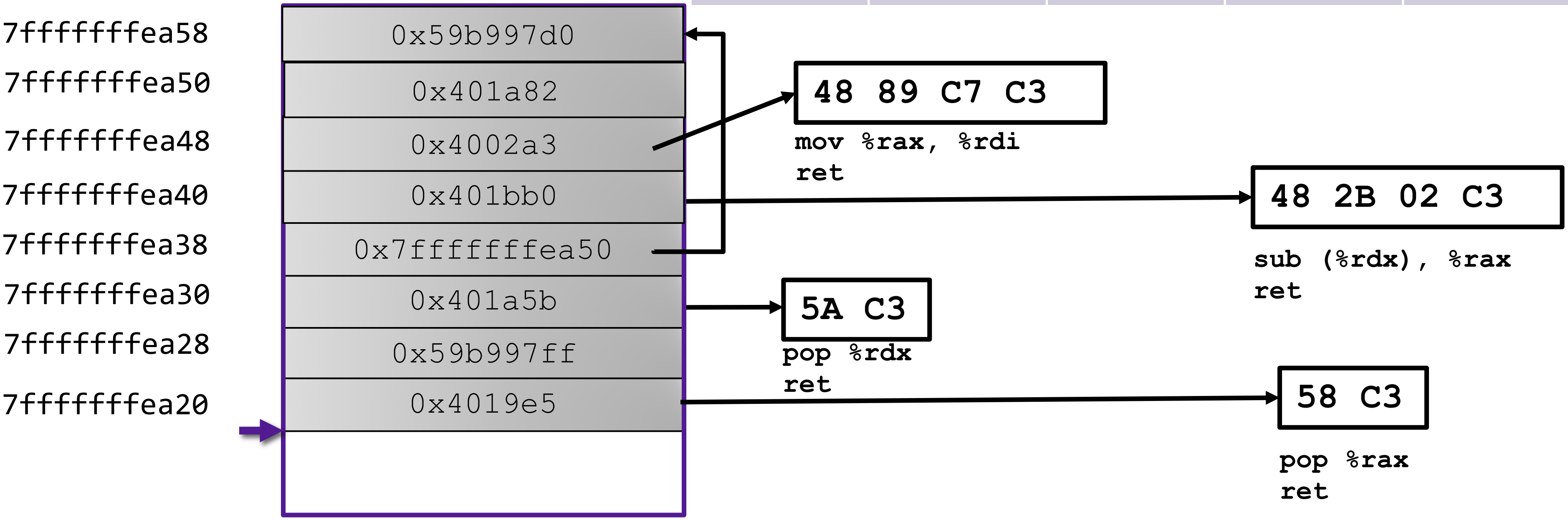
```
5F 5E C3
pop %rdi
pop %rsi
ret
```

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

Exercise 2: ROP

- What are the values in the registers when the function at address 0x401a82 gets called?

%rip	%rdi	%rsi	%rax	%rdx



7fffffffea58	0x59b997d0
7fffffffea50	0x401a82
7fffffffea48	0x4002a3
7fffffffea40	0x401bb0
7fffffffea38	0x7fffffffea50
7fffffffea30	0x401a5b
7fffffffea28	0x59b997ff
7fffffffea20	0x4019e5

```
48 89 C7 C3
mov %rax, %rdi
ret
```

```
48 2B 02 C3
sub (%rdx), %rax
ret
```

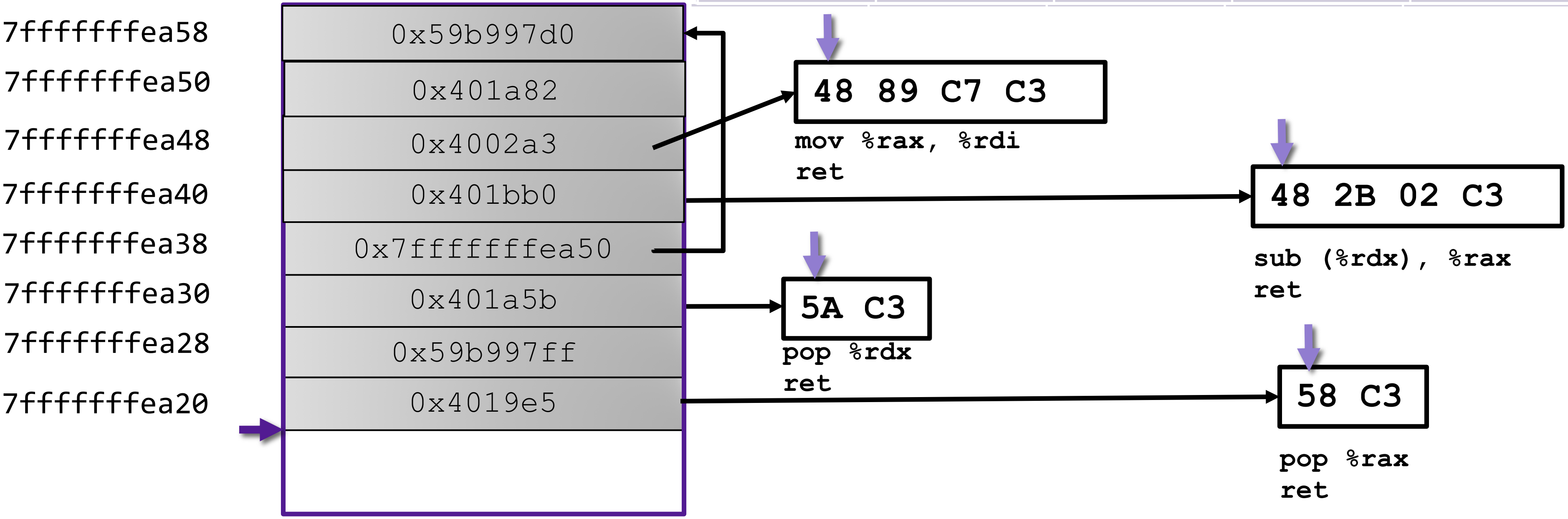
```
5A C3
pop %rdx
ret
```

```
58 C3
pop %rax
ret
```

Exercise 2: ROP

- What are the values in the registers when the function at address 0x401a82 gets called?

%rip	%rdi	%rsi	%rax	%rdx
401a82	2f		2f	..ea58



7fffffffea58	0x59b997d0
7fffffffea50	0x401a82
7fffffffea48	0x4002a3
7fffffffea40	0x401bb0
7fffffffea38	0x7fffffffea50
7fffffffea30	0x401a5b
7fffffffea28	0x59b997ff
7fffffffea20	0x4019e5

48 89 C7 C3

```
mov %rax, %rdi
ret
```

48 2B 02 C3

```
sub (%rdx), %rax
ret
```

5A C3

```
pop %rdx
ret
```

58 C3

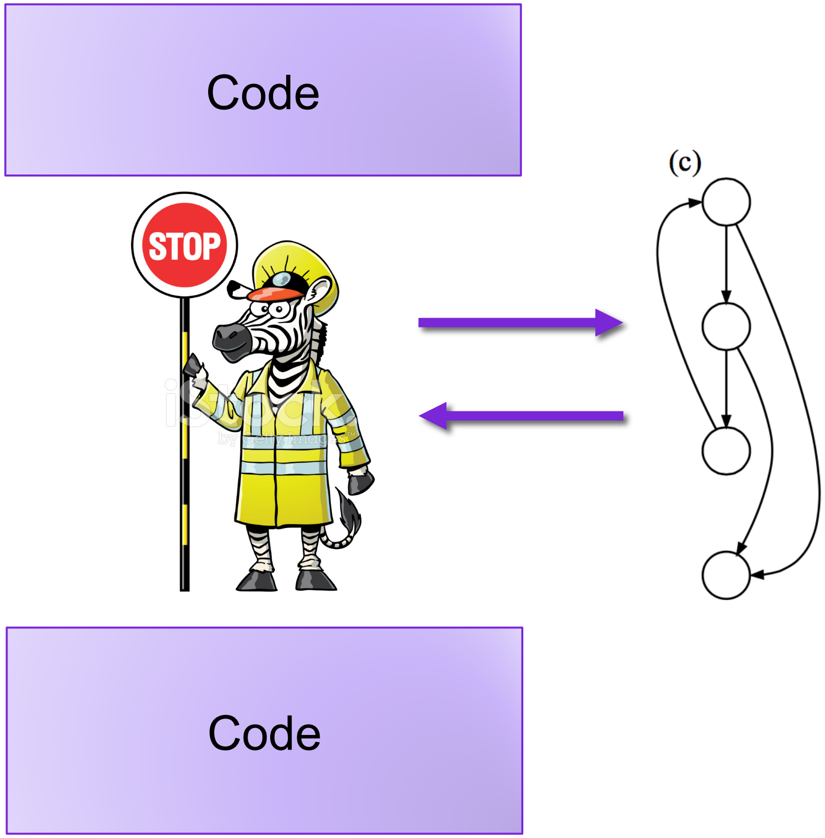
```
pop %rax
ret
```

Address Space Layout Randomization

Other defenses

Gadget Elimination

Control Flow Integrity



The state of the world

Defenses:

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

But all they aren't perfect!

Exercise 3: Feedback

1. Rate how well you think this recorded lecture worked
   1. Better than an in-person class
   2. About as well as an in-person class
   3. Less well than an in-person class, but you still learned something
   4. Total waste of time, you didn't learn anything

2. How much time did you spend on this video lecture (including time spent on exercises)?

3. Do you have any questions that you would like me to address in this week's problem session?

4. Do you have any other comments or feedback?