Lecture 9: Buffer Overflows (cont'd)

CS 105

Spring 2026

Buffer Overflows – Checking In

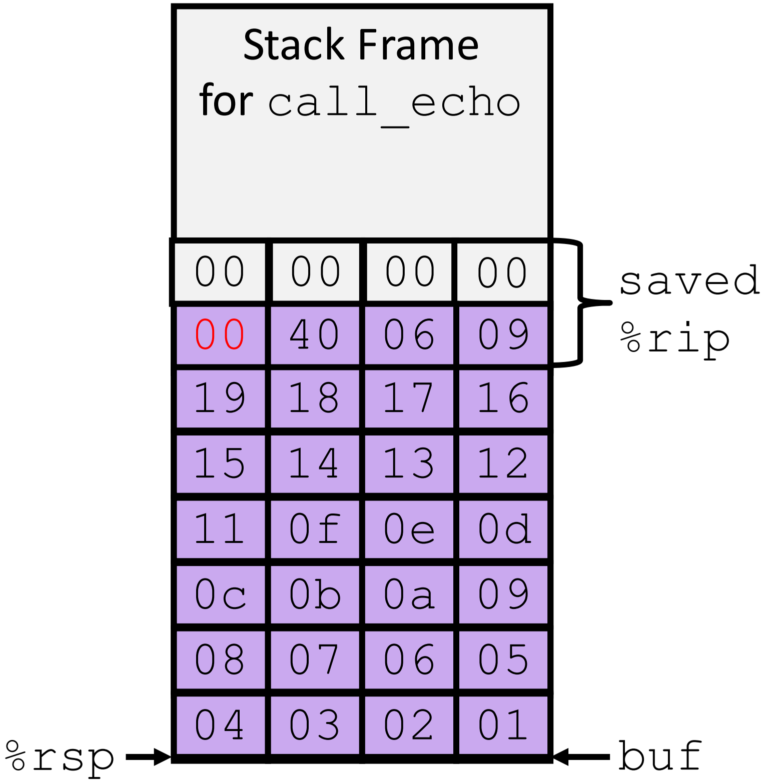
- What is an attacker’s goal when exploiting a buffer overflow?
- What is the vulnerability in this program?
- Which behavior is the attacker exploiting (e.g., which instructions)?
- What is the strategy for constructing a payload to exploit the vulnerability?

```
int authenticate(char *password){
  char buf[4];
  gets(buf);
  int correct = !strcmp(password, buf);
  return correct;
}

int main(int argc, char ** argv){
  char * pw = "123456";
  printf("Enter your password: ");
  while(!authenticate(pw)){
    printf("Incorrect. Try again: ");
  }
  printf("You are now logged in\n");
  return 0;
}
```

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

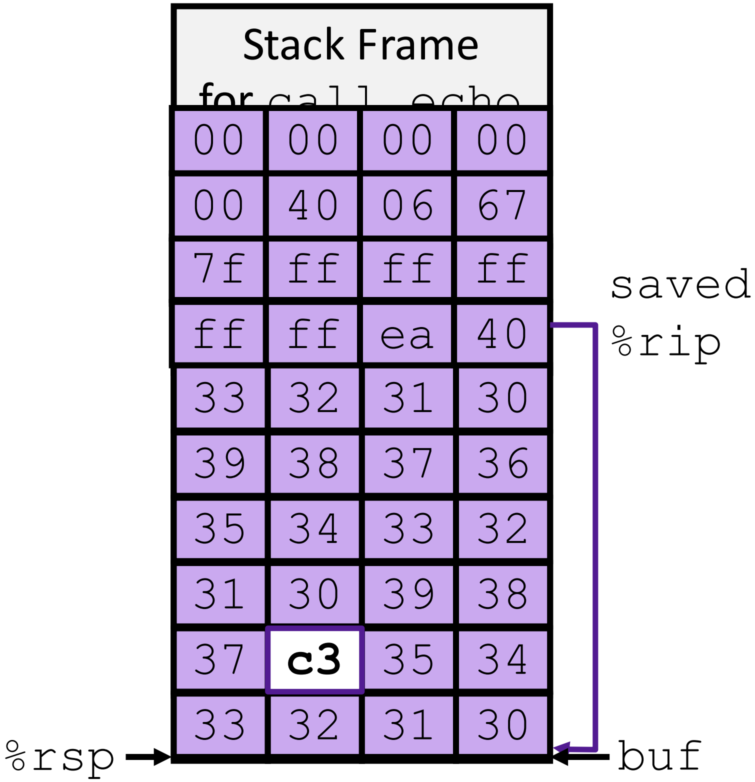


```
/* Echo Line */
void echo()
{
    char buf[4];
    gets(buf);
    puts(buf);
}
```

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

Code Injection

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



Stack Frame
for call echo

00	00	00	00
00	40	06	67
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	c3	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
```

Exercise: Code Injection

- Construct an input string that will cause this program to print 47.

```
int x = 13;

void echo(){
  char buf[12];
  gets(buf);
  puts(buf);
}

int main(int argc, char ** argv){
  printf("Enter a string: ");
  echo();
  printf("%d\n", x);
  return 0;
}
```

Exercise: Code In

- Construct an input string th print 47.

```
int x = 13;

void echo(){
  char buf[12];
  gets(buf);
  puts(buf);
}

int main(int argc, char ** argv){
  printf("Enter a string: ");
  echo();
  printf("%d\n", x);
  return 0;
}
```

Assume `%rsp == 0x7fffffffe168` at the beginning of `echo`

```
echo:
   0x400616 <+0>:   sub    $0x18,%rsp
   0x40061a <+4>:   lea    0x4(%rsp),%rax
   0x40061f <+9>:   mov    %rax,%rdi
   0x400622 <+12>:  mov    $0x0,%eax
   0x400627 <+17>:  callq  0x400520 <gets>
   0x40062c <+22>:  lea    0x4(%rsp),%rax
   0x400631 <+27>:  mov    %rax,%rdi
   0x400634 <+30>:  callq  0x400500 <puts@>
   0x400639 <+35>:  nop
   0x40063a <+36>:  add    $0x18,%rsp
   0x40063e <+40>:  retq
main:
   0x40063f <+0>:   sub    $0x18,%rsp
   0x400643 <+4>:   mov    %edi,0xc(%rsp)
   0x400647 <+8>:   mov    %rsi,(%rsp)
   0x40064b <+12>:  mov    $0x400728,%edi
   0x400650 <+17>:  mov    $0x0,%eax
   0x400655 <+22>:  callq  0x400510 <printf>
   0x40065a <+27>:  mov    $0x0,%eax
   0x40065f <+32>:  callq  0x400616 <echo>
   0x400664 <+37>:  mov    0x601034,%eax
   0x40066a <+43>:  mov    %eax,%esi
   0x40066c <+45>:  mov    $0x400739,%edi
   0x400671 <+50>:  mov    $0x0,%eax
   0x400676 <+55>:  callq  0x400510 <printf>
   0x40067b <+60>:  mov    $0x0,%eax
   0x400680 <+65>:  add    $0x18,%rsp
   0x400684 <+69>:  retq
```

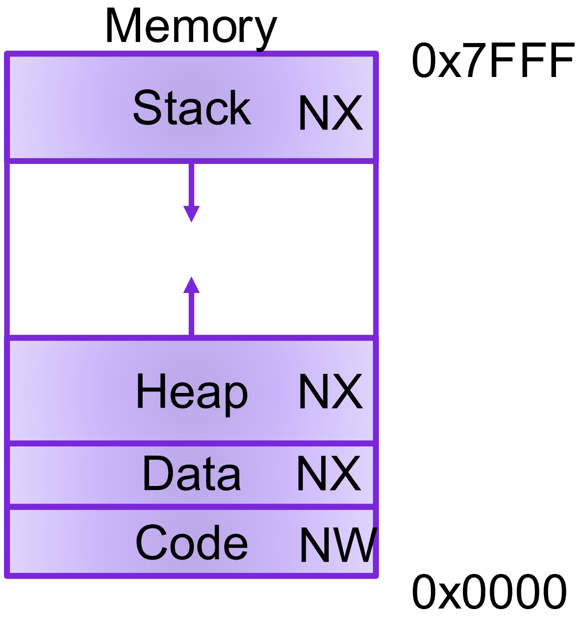
```
0000000000000000 <.text>:
   0:   48 c7 04 25 34 10 60 00 2f 00 00 00    movq   $0x2f,0x601034
   c:   c3                                     retq
```

Defense #1: Bounds Checks

```
/* Echo Line */
void echo(){
   char buf[12];
   fgets(buf, 12, 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

Defense #2: Memory Tagging



GCC Implementation

- Now the default
- Can disable with `-z execstack`

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)

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
```

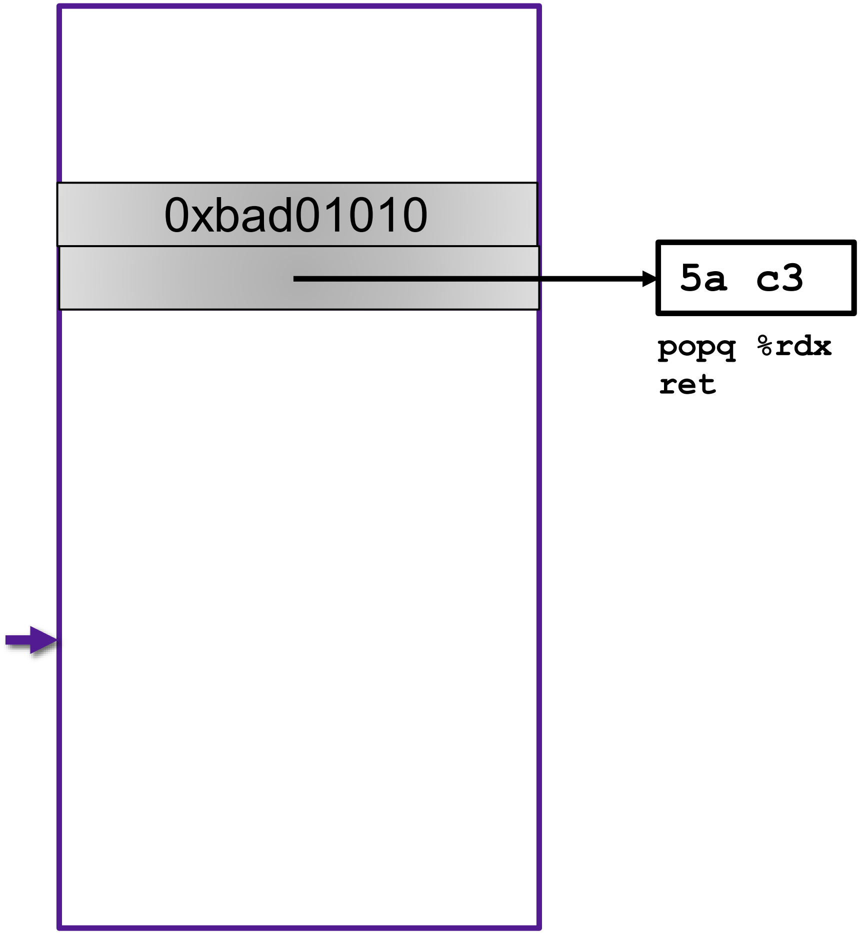
gadget address:	`0x4004dc`

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

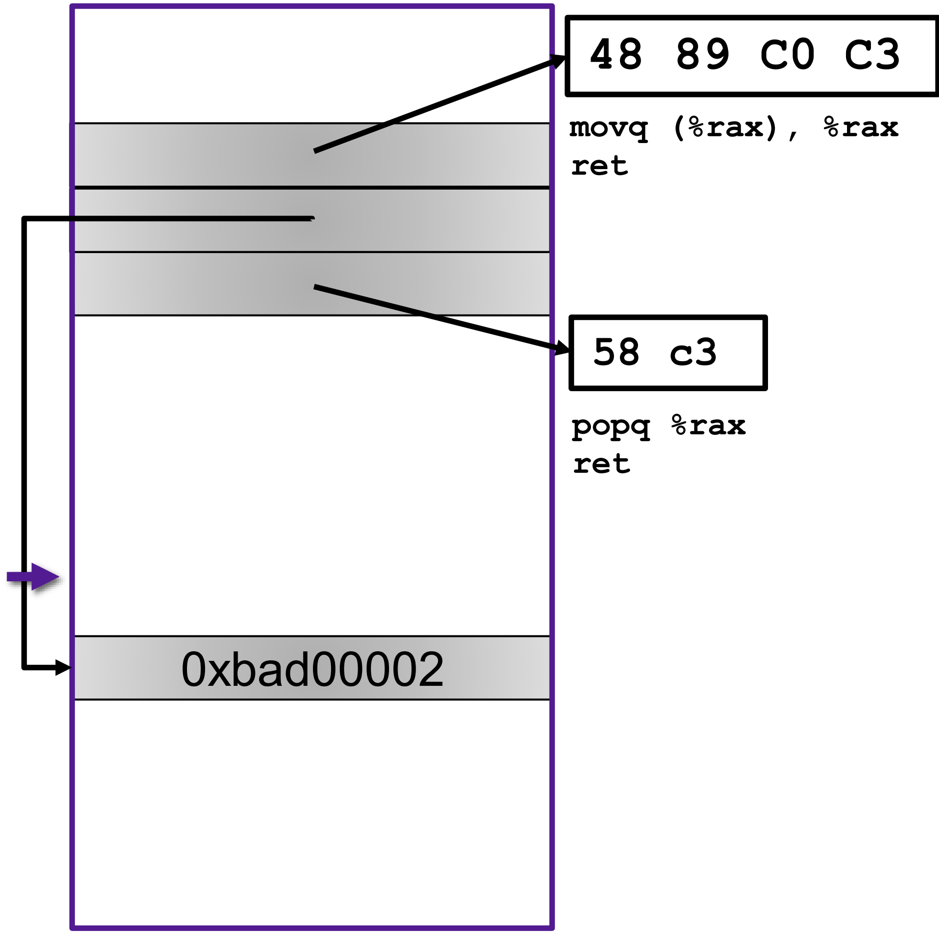
executes:	`%rdi <- %rax`

Example Gadgets

Load Constant



Load from memory

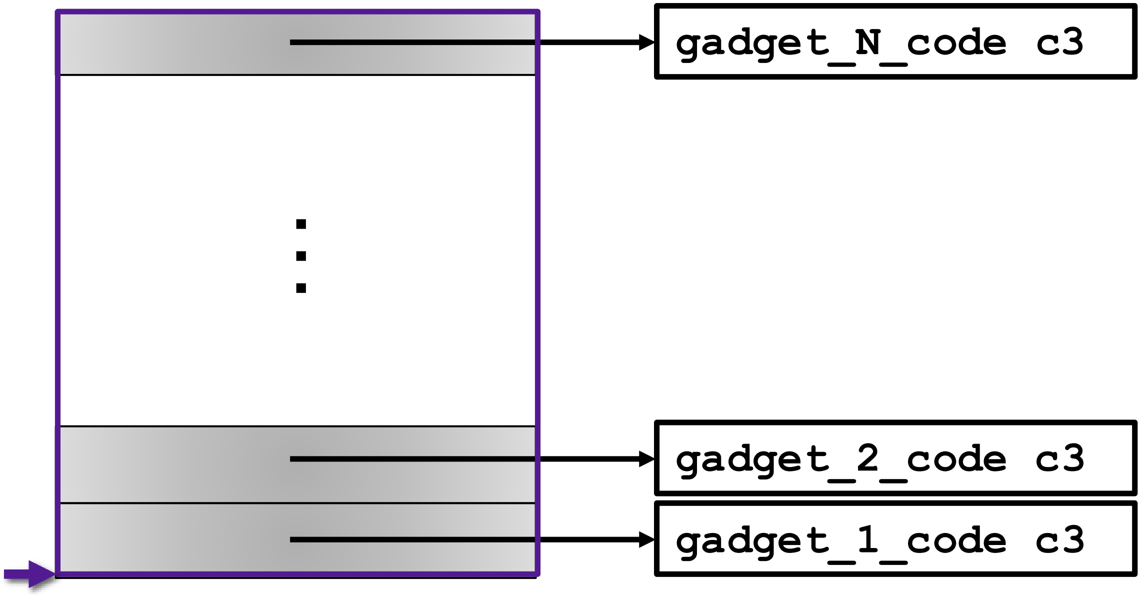


Return-oriented Programming

Return-oriented Programming

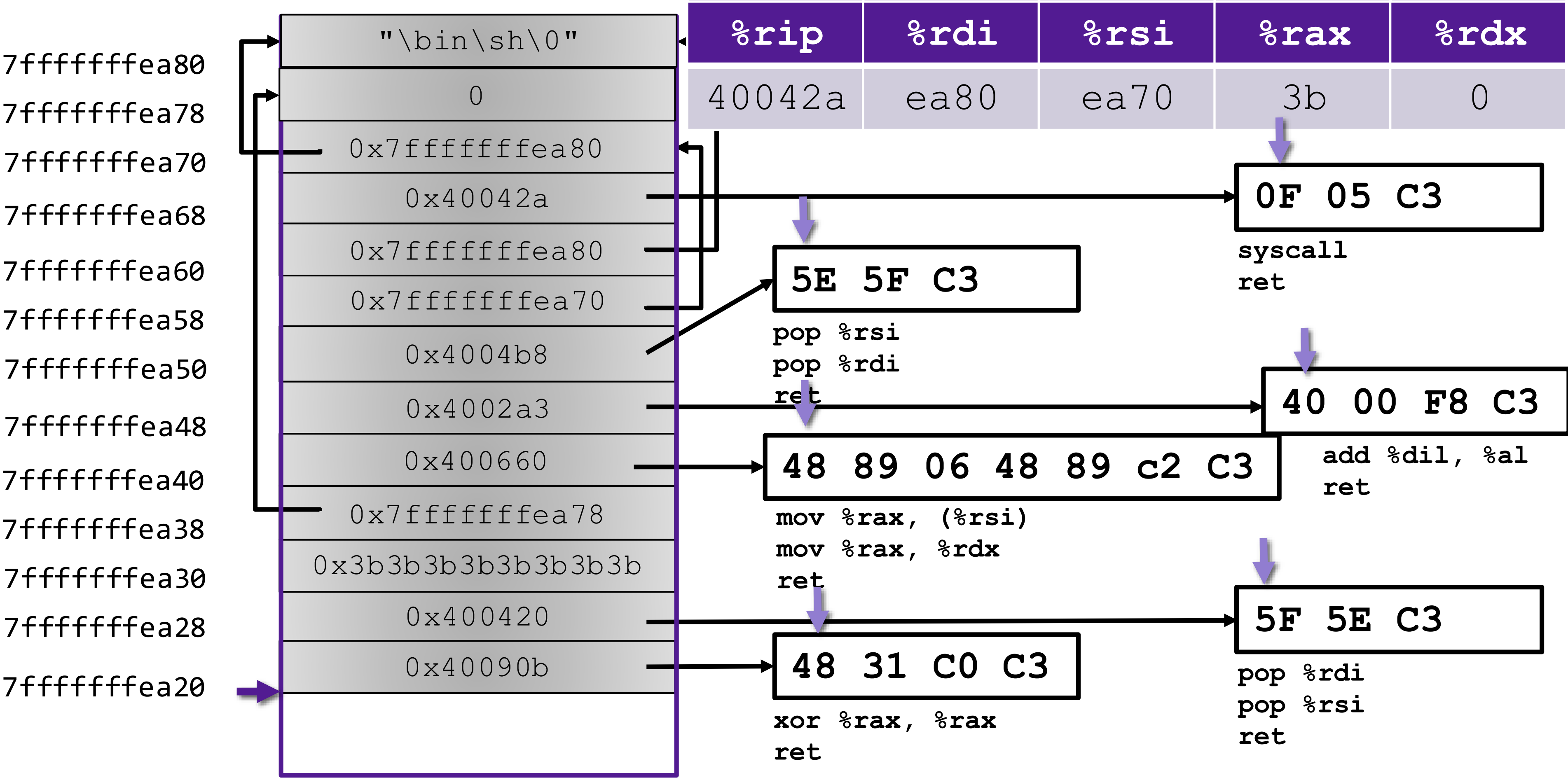
is a lot like a ransom note, but instead of cutting cut letters from magazines, you are cutting out instructions from text segments

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
```

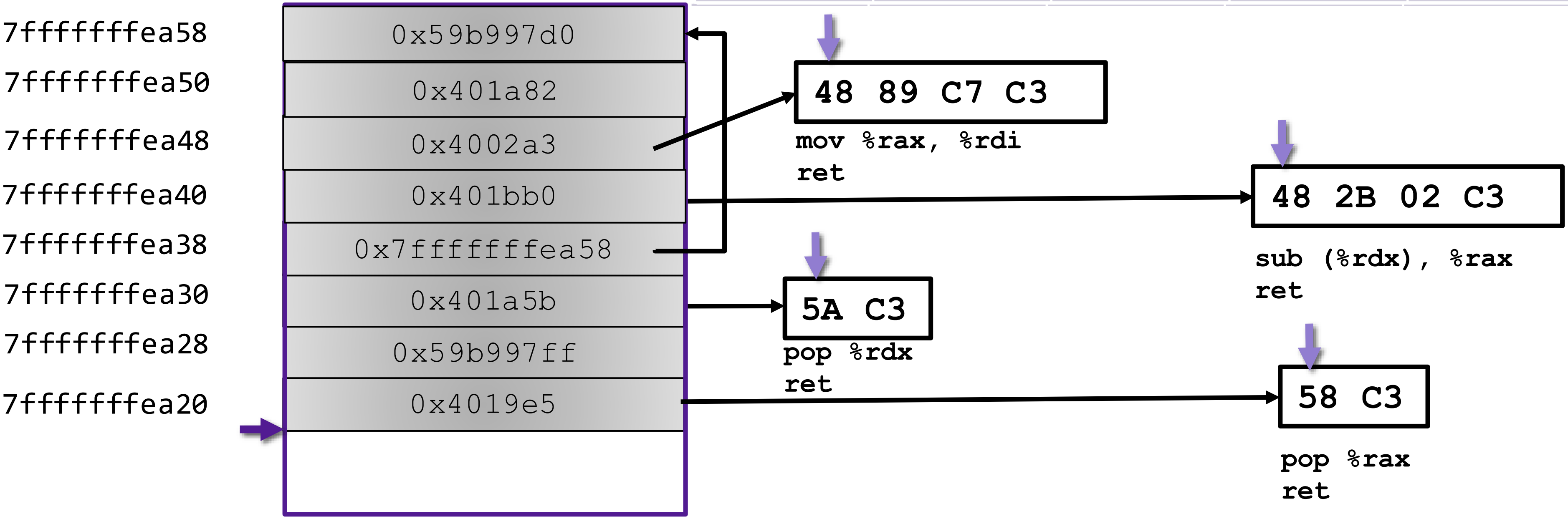
```
5F 5E C3
pop %rdi
pop %rsi
ret
```

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

Exercise: ROP

- What are the values in the registers when the function at address 0x401a82 starts executing?

%rip	%rdi	%rsi	%rax	%rdx



7fffffffea58	0x59b997d0
7fffffffea50	0x401a82
7fffffffea48	0x4002a3
7fffffffea40	0x401bb0
7fffffffea38	0x7fffffffea58
7fffffffea30	0x401a5b
7fffffffea28	0x59b997ff
7fffffffea20	0x4019e5

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

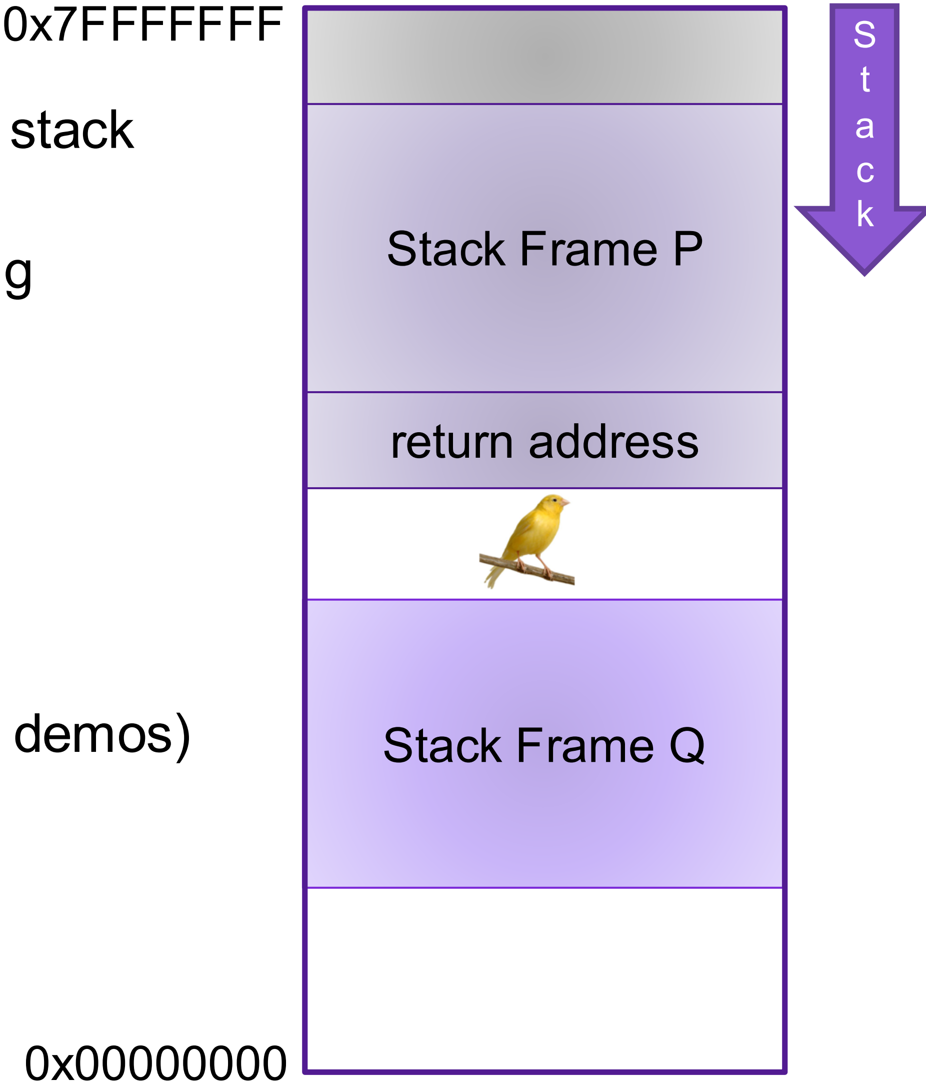
```
48 2B 02 C3
sub (%rdx), %rax
ret
```

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

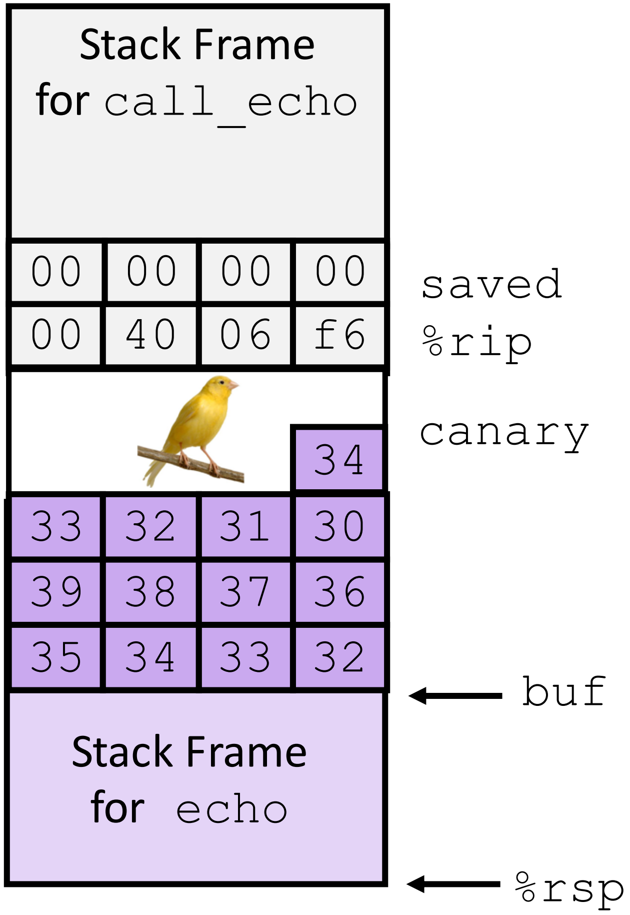
```
58 C3
pop %rax
ret
```

Defense #3: 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 in prior demos)



Stack Canaries



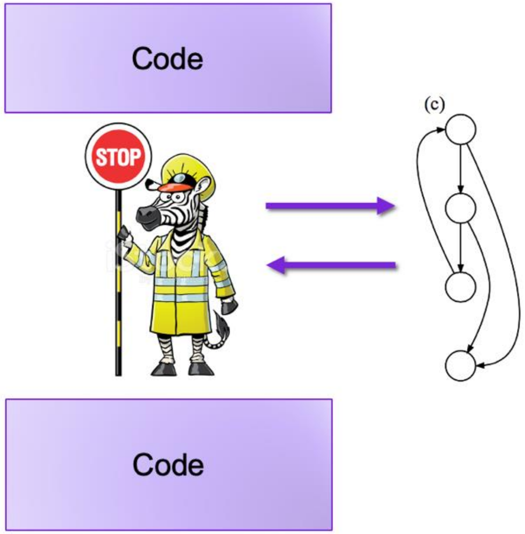
```
echo:
    sub     $0x28,%rsp
    mov     %fs:0x28,%rax
    mov     %rax,0x18(%rsp)
    xor     %eax,%eax
    lea     0xc(%rsp),%rax
    mov     %rax,%rdi
    mov     $0x0,%eax
    callq   0x400590 <gets>
    lea     0xc(%rsp),%rax
    mov     %rax,%rdi
    callq   0x400560 <puts>
    nop
    mov     0x18(%rsp),%rax
    xor     %fs:0x28,%rax
    je      0x4006cf <echo+73>
    callq   0x400570 <__stack_chk_fail>
    add     $0x28,%rsp
    retq
```

Other Defenses

Address Space Layout Randomization

Gadget Elimination

Control Flow Integrity



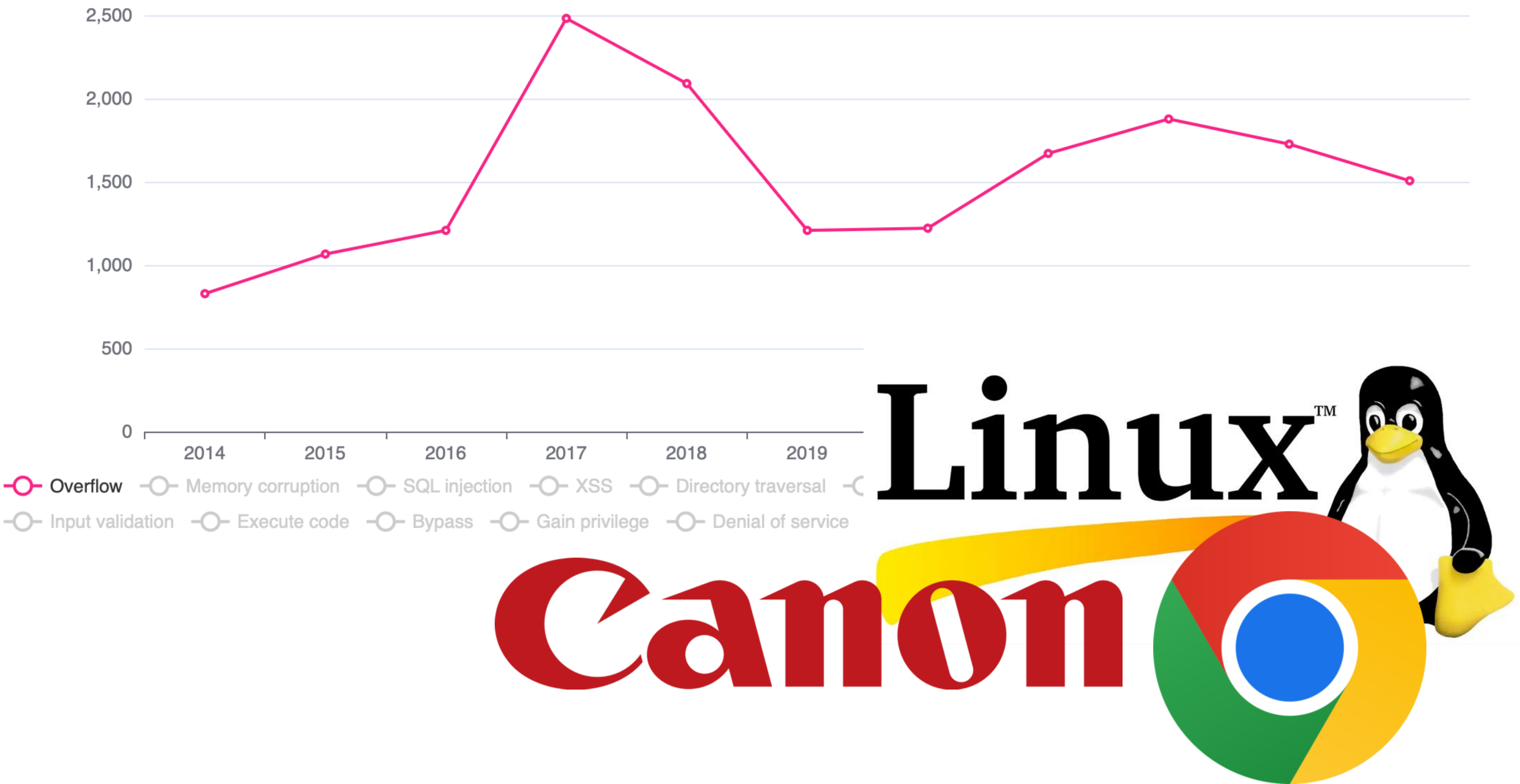
The state of the world

Defenses:

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

But all they aren't perfect!

The state of the world


Overflow
Memory corruption
SQL injection
XSS
Directory traversal
Input validation
Execute code
Bypass
Gain privilege
Denial of service