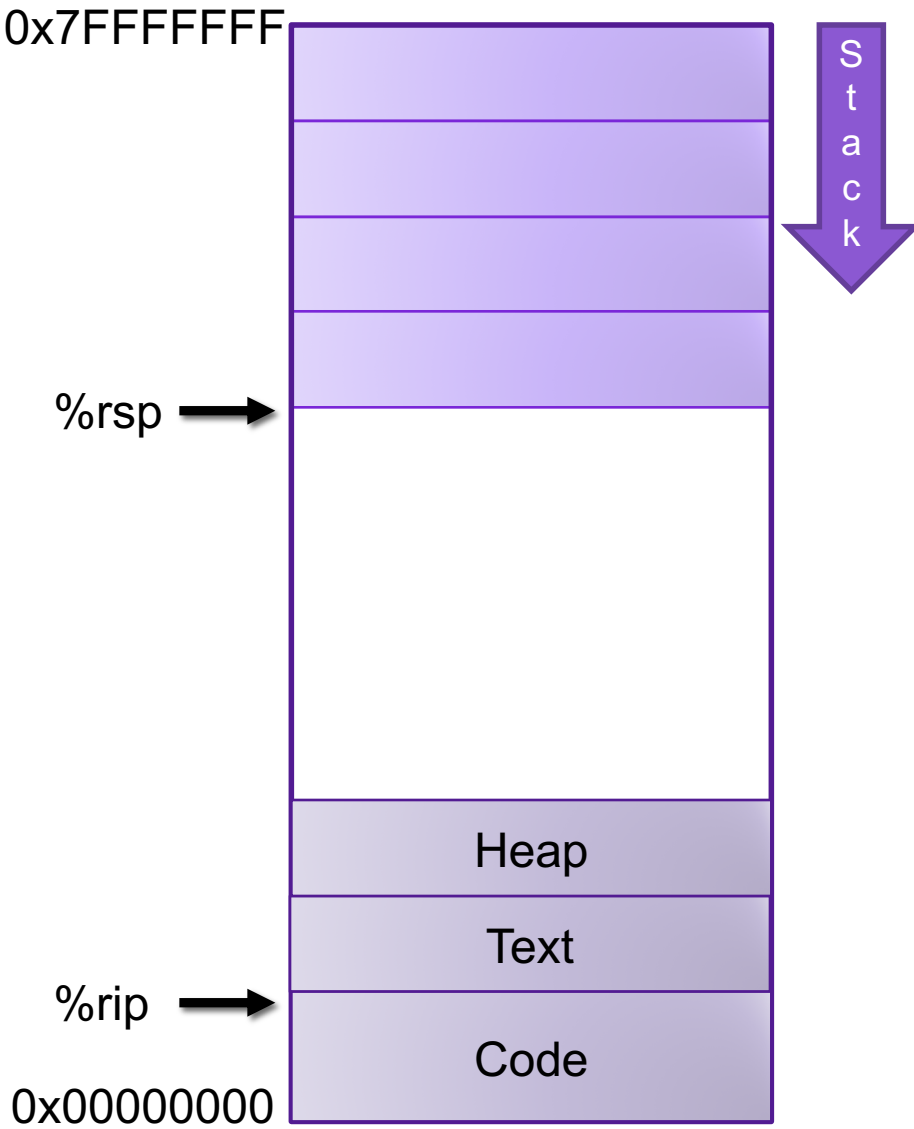
Lecture 8: Use and Abuse of the Stack

CS 105

September 24, 2019

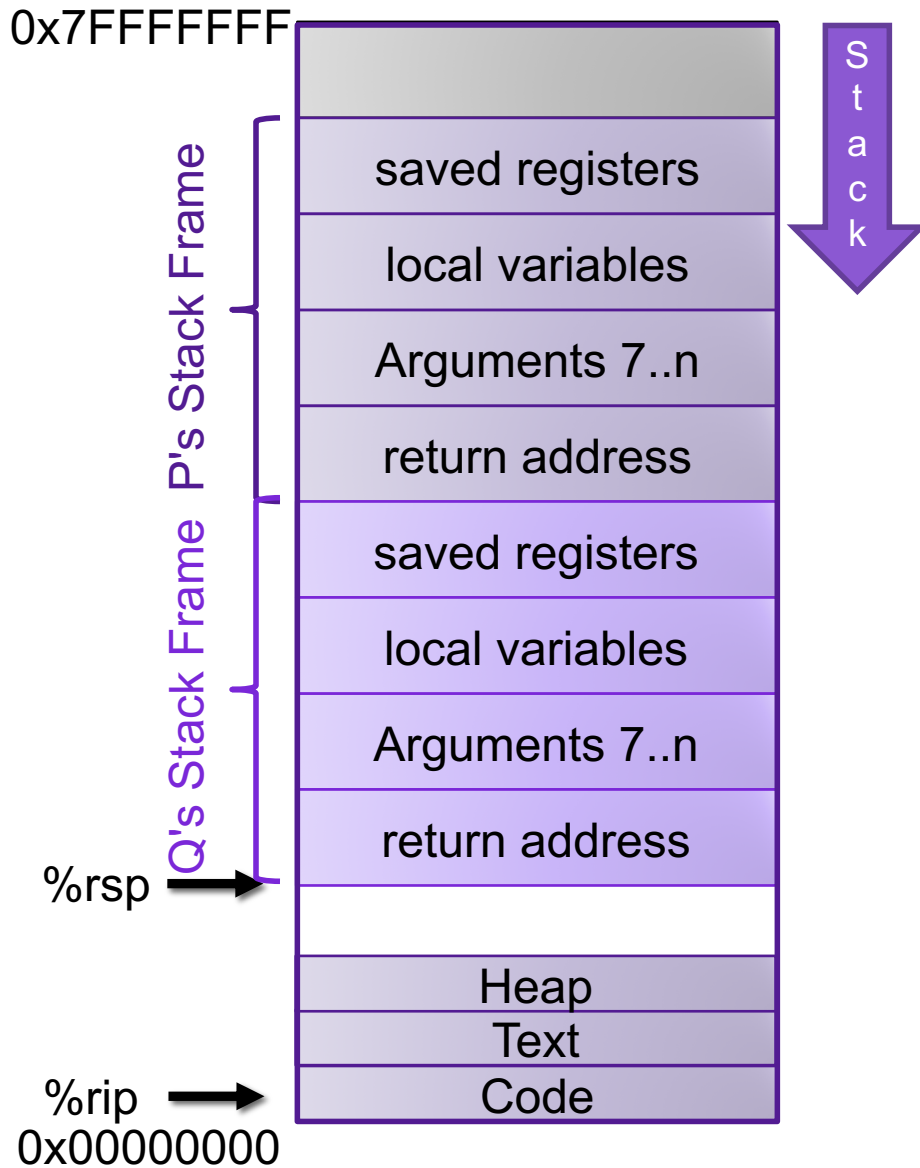
Review: x86-64 Linux Memory Layout

- Stack
  - Runtime stack (8MB limit)
  - E. g., local variables
- Heap
  - Dynamically allocated as needed
  - When calling malloc(), calloc(), new()
- Data
  - Statically allocated data
  - E.g., global vars, `static` vars, string constants
- Text / Shared Libraries
  - Executable machine instructions
  - Read-only



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



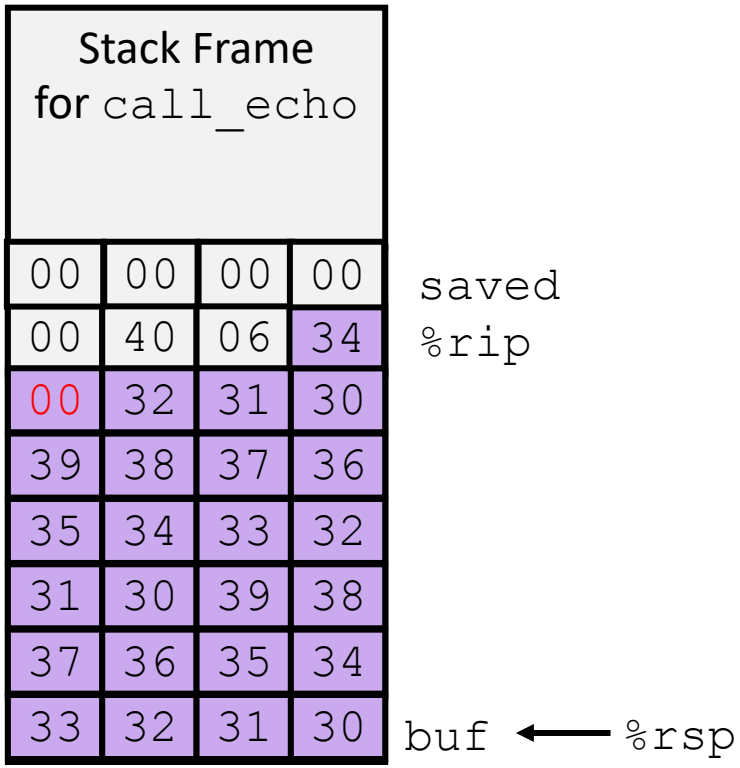
Memory Referencing Bug Example

```
int proc(int *p);

int example1(int x) {
  int a[4];
  a[3] = 10;
  return proc(a);
}
```

```
example1:
  subq  $16, %rsp
  movl  $10, 12(%rsp)
  movq  %rsp, %rdi
  call  proc
  addq  $16, %rsp
  ret
```

Memory Referencing Bug Example

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




```
/* 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
```

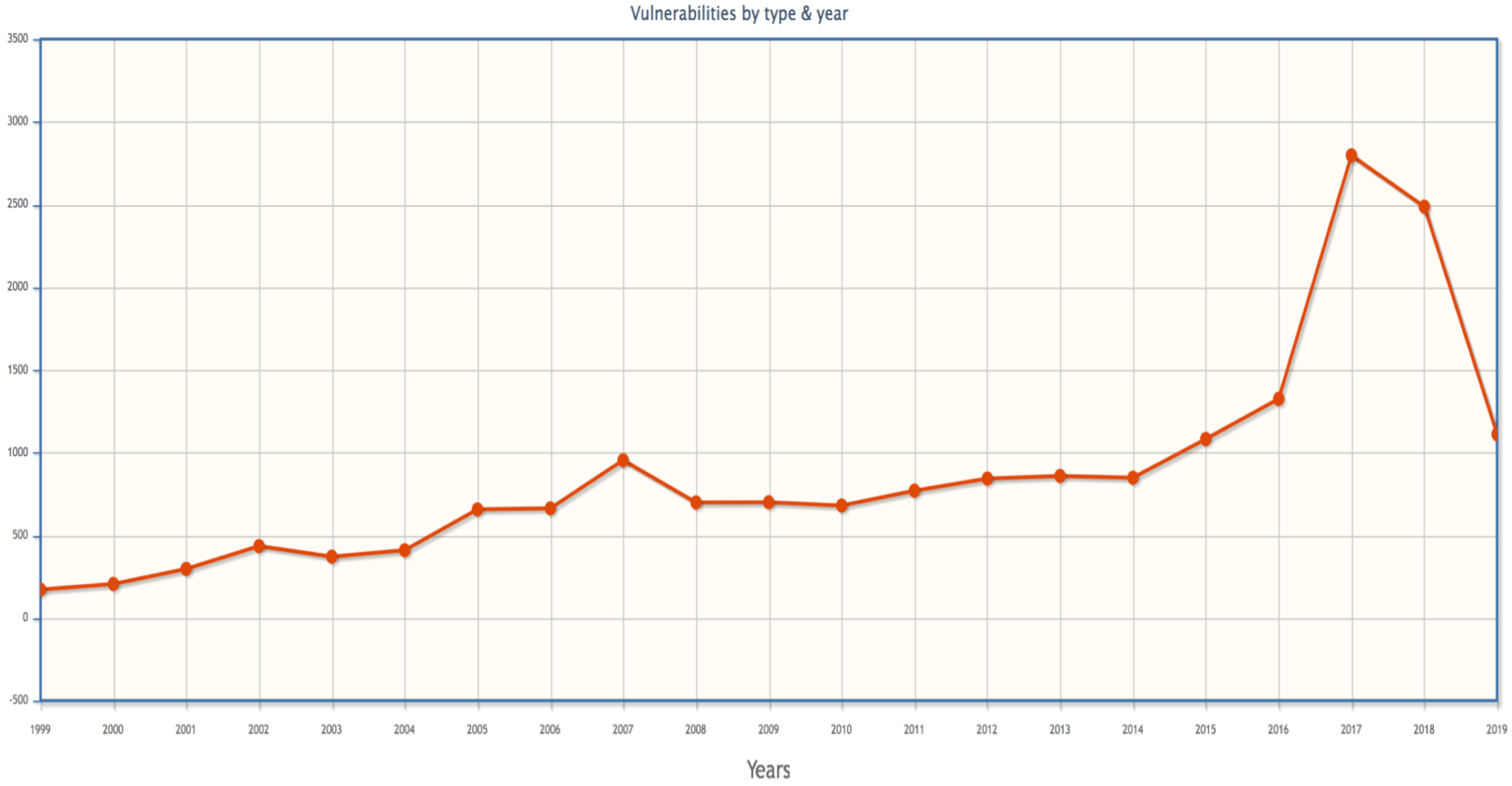
Example: server.c

Exercise

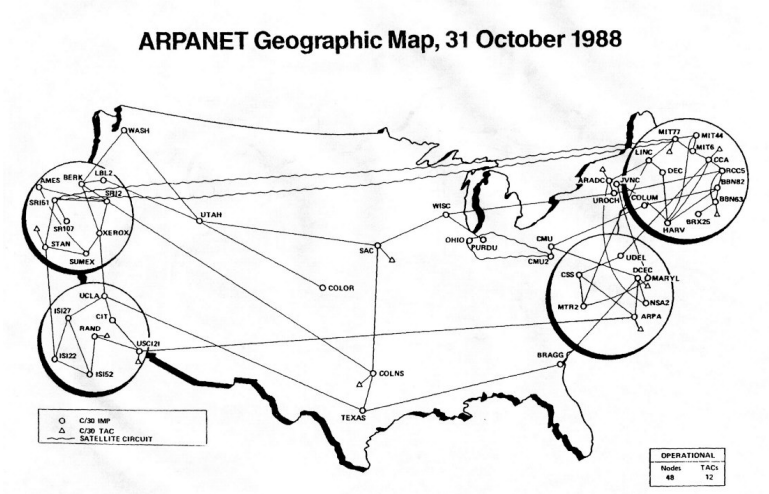
```
int authenticate(char *password){
  char buf[4];
  gets(buf);
  int correct
    = !strcmp(password, buf);
  return correct;
}

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;
}
```

```
authenticate:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    %rsp, %rdi
  movl    $0, %eax
  call    gets
  movq    %rsp, %rsi
  movq    %rbx, %rdi
  call    strcmp
  testl   %eax, %eax
  sete    %al
  movzbl  %al, %eax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Buffer Overflow Vulnerabilities



Buffer Overflow Examples



Defending against buffer overflow attacks

- Avoid overflow vulnerabilities
- Randomize the stack
- Insert compiler checks
- Employ system-level protections

1. Avoid Overflow Vulnerabilities in Code (!)

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

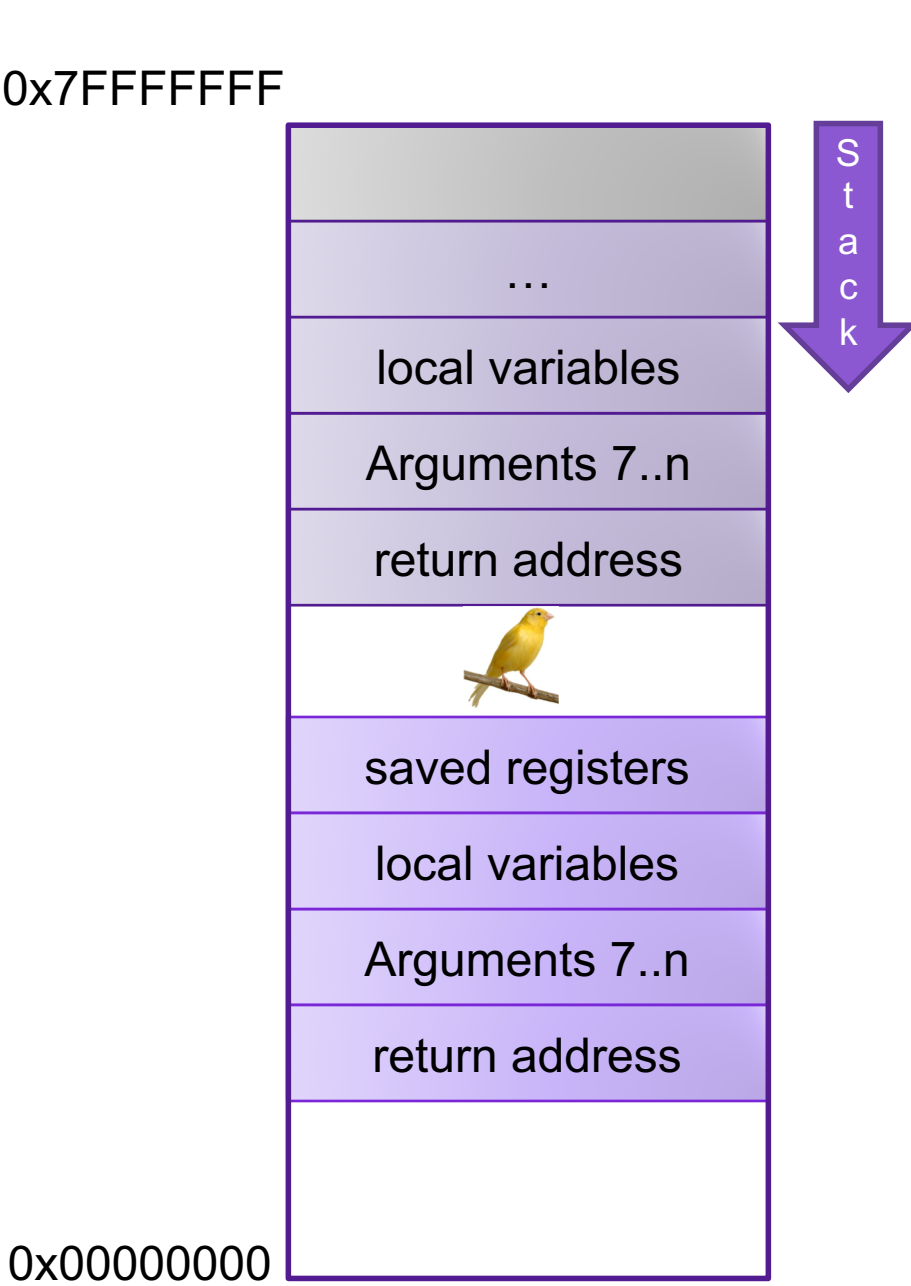
- For example, use library routines that limit string lengths
  - **`fgets`** instead of **`gets`**
  - **`strncpy`** instead of **`strcpy`**
  - Don't use **`scanf`** with **`%s`** conversion specification
    - Use **`fgets`** to read the string
    - Or use **`%ns`** where **`n`** is a suitable integer

2. Randomize the Stack (ASLR)

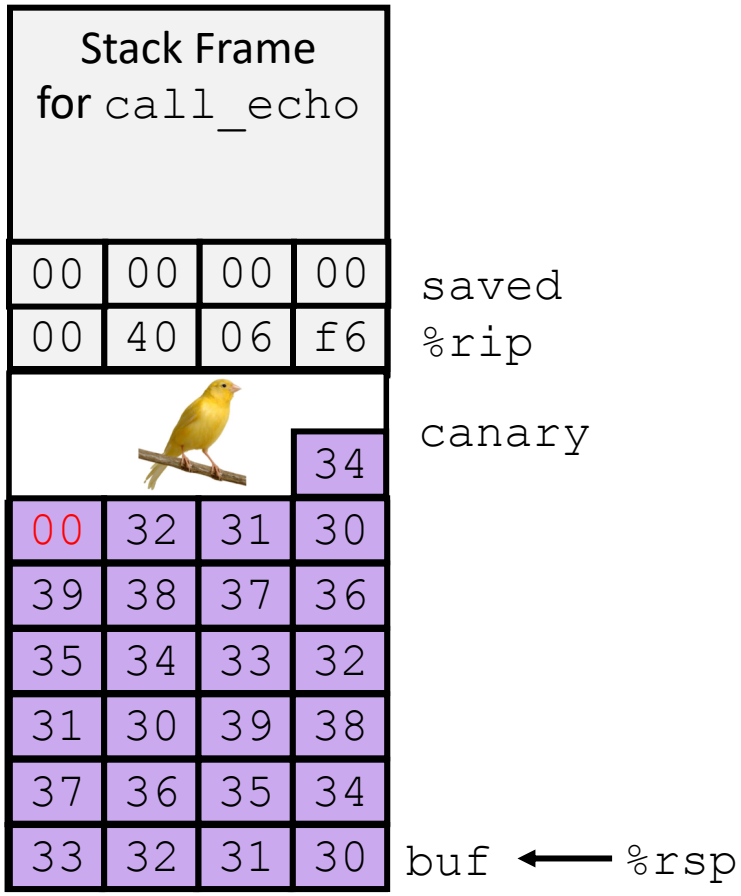
3. Compiler checks

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

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



Stack Canaries




```
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
```

4. System-level Protection: Memory Tagging

Code-Reuse Attacks

- Next time!