Lecture 9: Buffer Overflows

CS 105

February 19, 2020

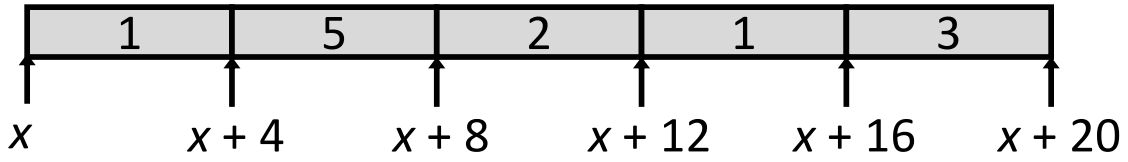
From Last time…

- Basic Principle

  T **`A[`***L***`];`**

  - Array of data type *T* and length *L*
  - Identifier **`A`** can be used as a pointer to array element 0: Type *T**

`int val[5];`



- Reference

Reference	Type	Value
`val[4]`	`int`	3
`val`	`int *`	*x*
`val+1`	`int *`	*x* + 4
`&val[2]`	`int *`	*x* + 8
`val[5]`	`int`	??
`*(val+1)`	`int`	5
`val + i`	`int *`	*x* + 4 *i*

Memory Referencing Bug Example

```
void f1(){
  double a2[2] = {1.0,2.0};
  int a1[4] = {1,2,3,4};
  a1[4] = 1413754136;
  a1[5] = 1074340347;
```

```
f1:
    sub     $0x38,%rsp
    movsd   0x216(%rip),%xmm0
    movsd   %xmm0,0x20(%rsp)
    movsd   0x210(%rip),%xmm0
    movsd   %xmm0,0x28(%rsp)
    movl    $0x1,0x10(%rsp)
    movl    $0x2,0x14(%rsp)
    movl    $0x3,0x18(%rsp)
    movl    $0x4,0x1c(%rsp)
    movl    $0x54442d18,0x20(%rsp)
    movl    $0x400921fb,0x24(%rsp)
    add     $0x38,%rsp
    retq
```

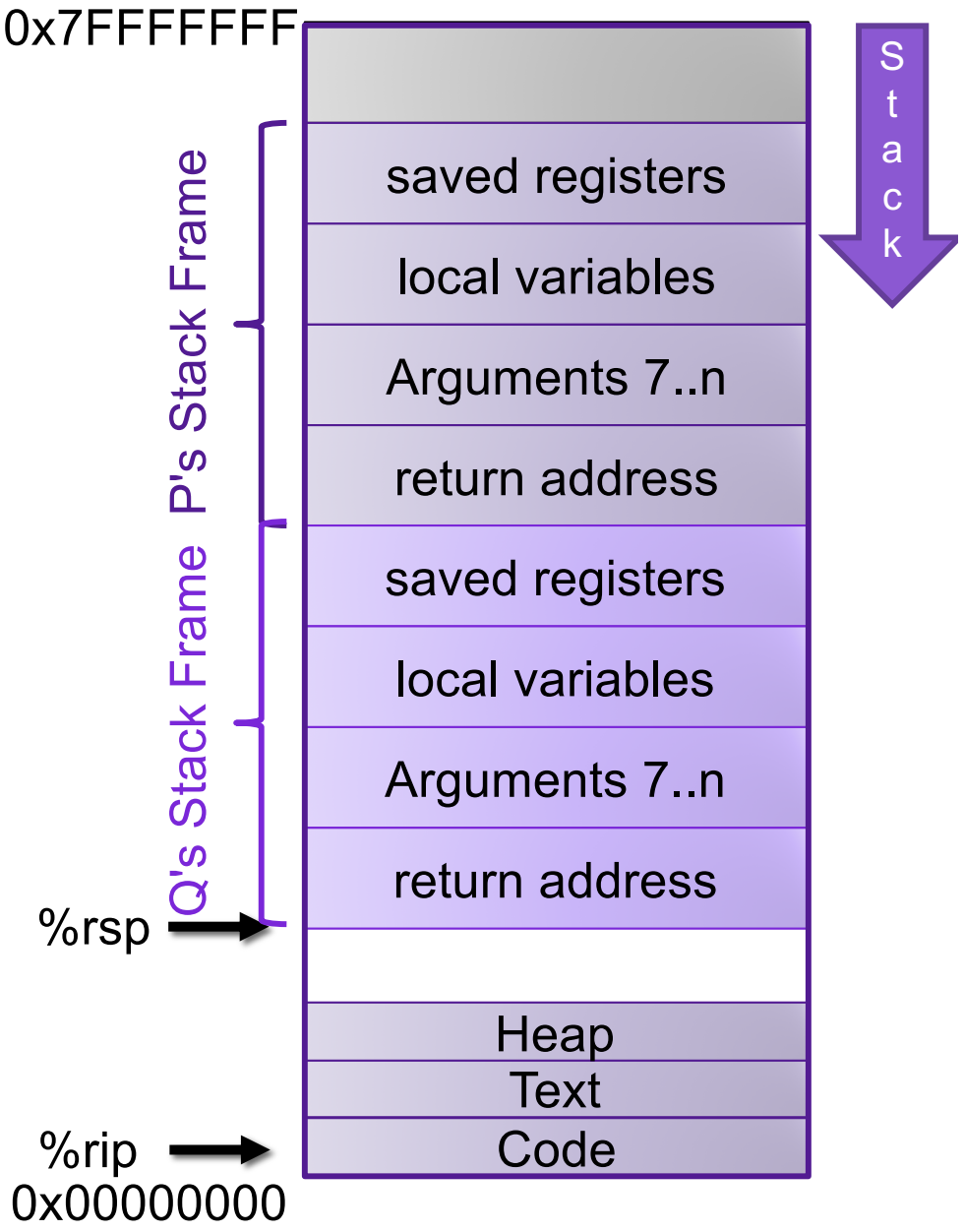
Another Memory Bug Example

```
int f2(){
  int a1[4] = {1,2,3,4};
  a1[6] = 47;
}
```

```
f2:
   sub     $0x18,%rsp
   movl    $0x1,(%rsp)
   movl    $0x2,0x4(%rsp)
   movl    $0x3,0x8(%rsp)
   movl    $0x4,0xc(%rsp)
   movl    $0x2f,0x18(%rsp)
   add     $0x18,%rsp
   retq
```

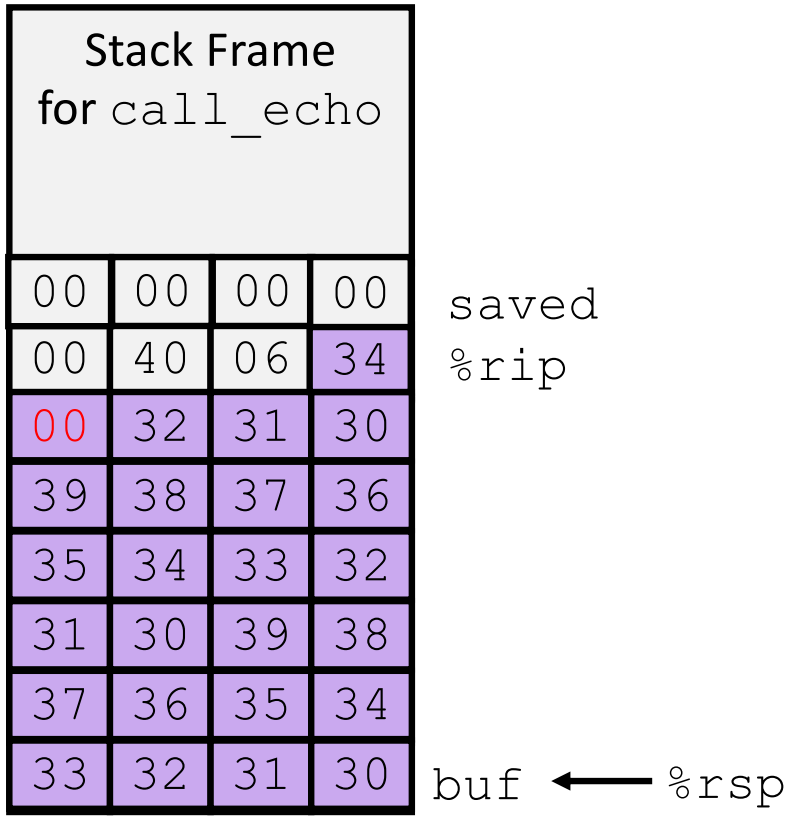
Review: Stack Frames

- 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



String Memory Referencing Bug

- 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);
}
```

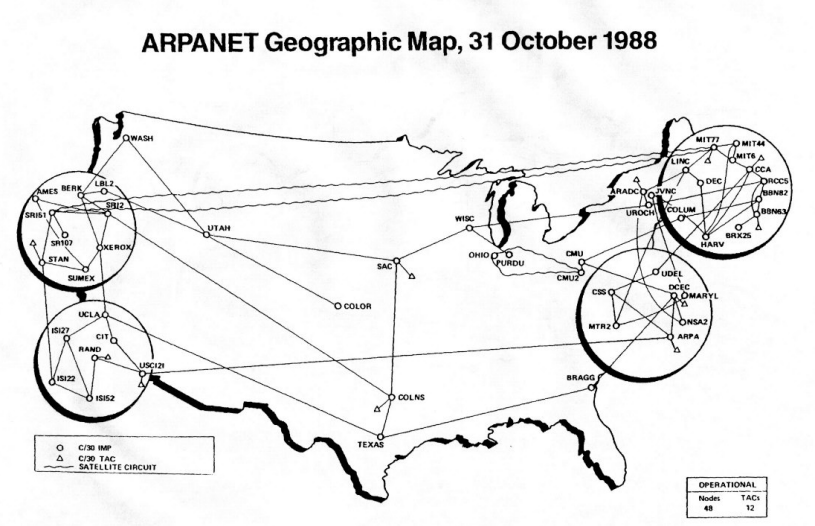
```
echo:
   subq   $0x18, %rsp
   movq   %rsp, %rdi
   call   gets
   call puts
   addq   $0x18, %rsp
   ret
```

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;
}
```

Buffer Overflow Examples

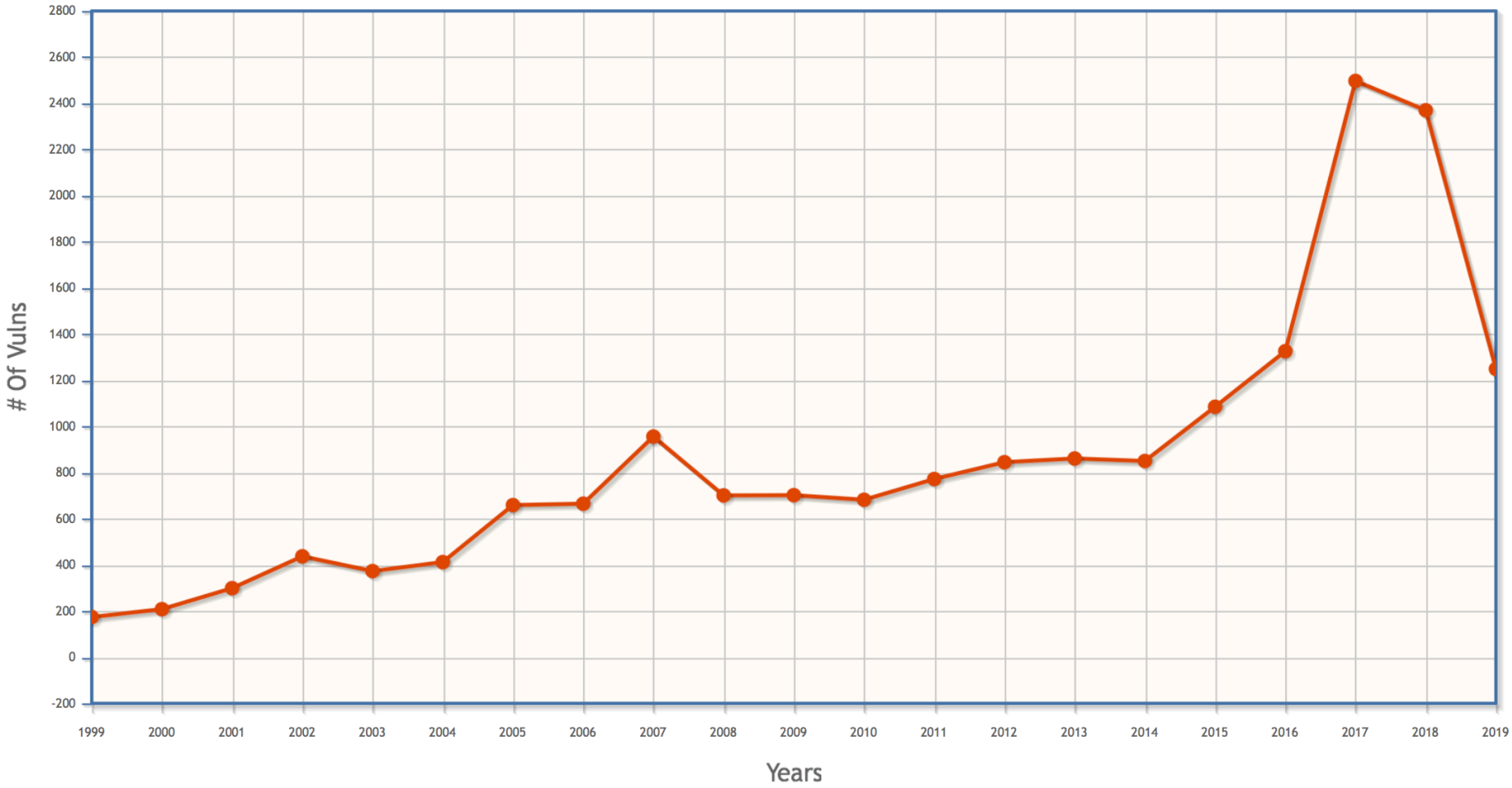


Defense #1: Avoid Overflow Vulnerabilities

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

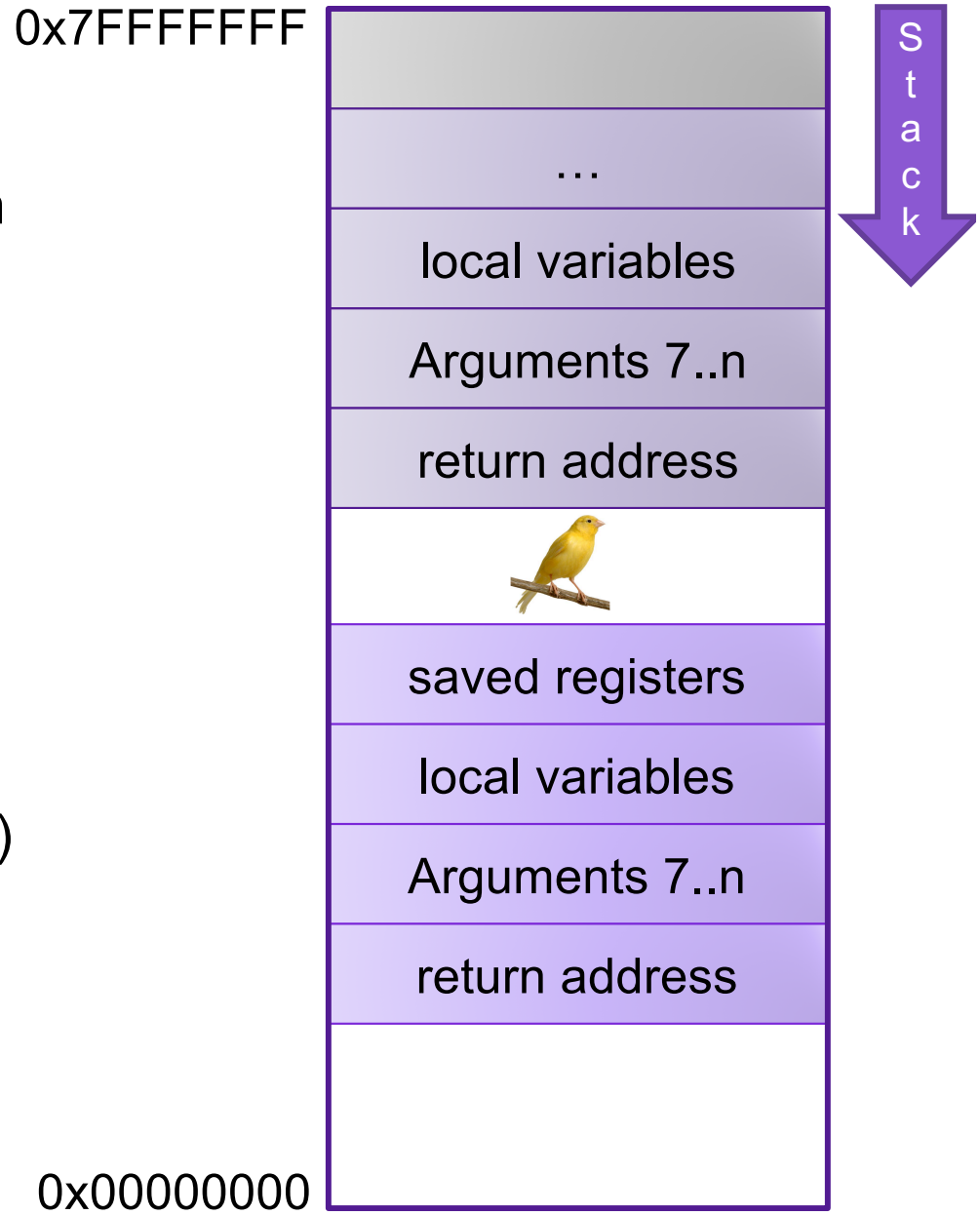
- For example, use library routines that limit string lengths
  - **`fgets`** instead of **`gets`**
  - **`strncpy`** instead of **`strcpy`**
  - Don't use **`scanf`** with **`%s`** conversion specification
    - Use **`fgets`** to read the string
    - Or use **`%ns`** where **`n`** is a suitable integer

Buffer Overflow Vulnerabilities

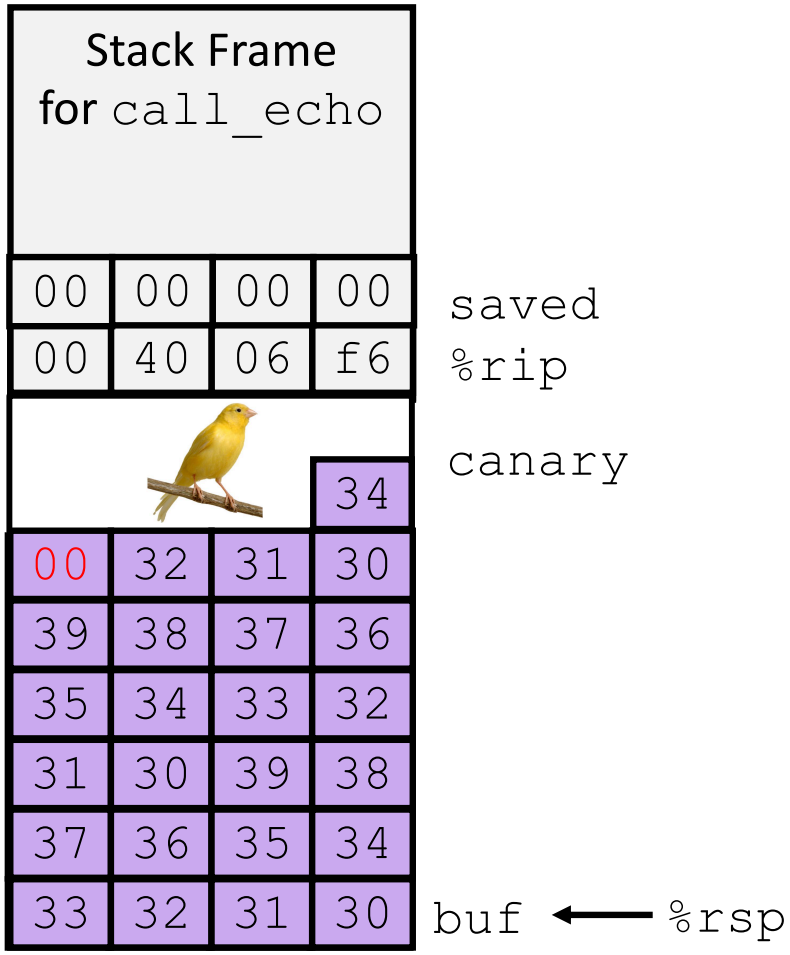


Defense #2: Compiler checks

- 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




```
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
```