
# 《计算机系统结构》课程直播

## 2020. April .2

请将ZOOM名称改为“姓名”；

**听不到声音请及时调试声音设备；签到将在课结束后继续**

# Machine-Level Programming : Advanced Topics

From CMU: Introduction to Computer Systems
Randal E. Bryant and David R. O'Hallaron

# Today

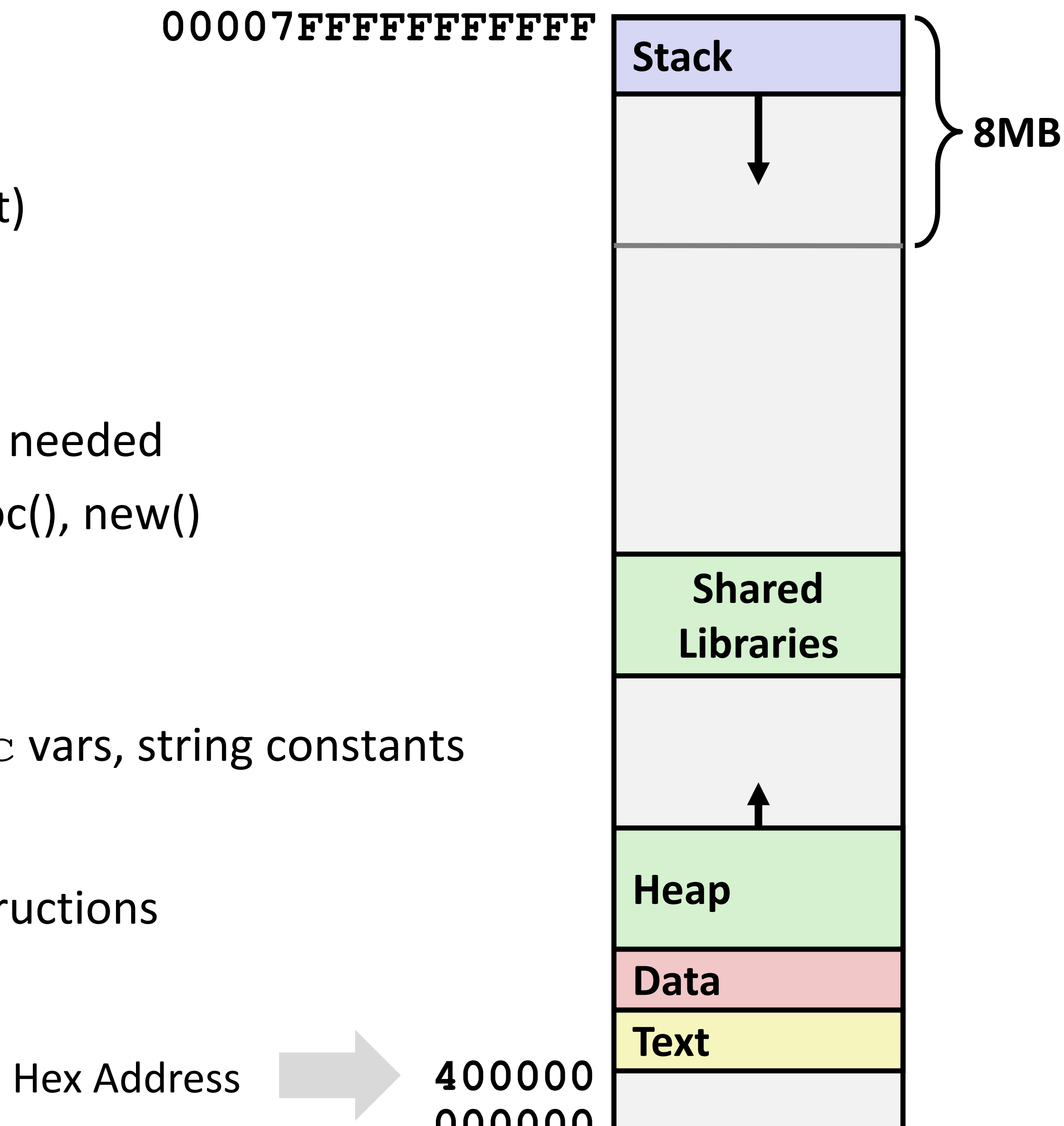
- X86-64 Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data

- Buffer Overflow
  - Vulnerability
  - Protection

# x86-64 Linux Memory Layout

*not drawn to scale*

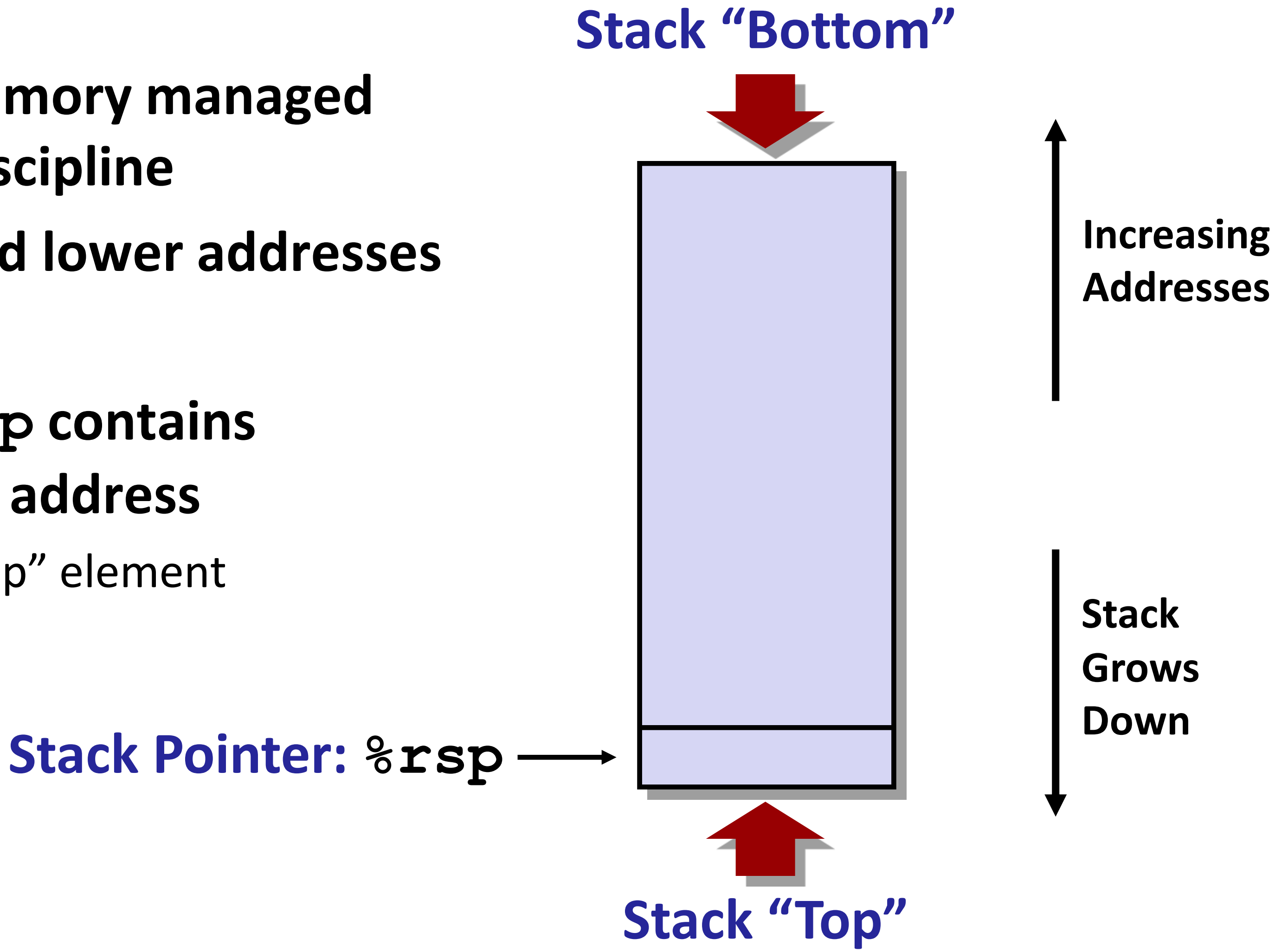
- **Stack**
  - Runtime stack (8MB limit)
  - E. g., local variables
- **Heap**
  - Dynamically allocated as needed
  - When call malloc(), calloc(), new()
- **Data**
  - Statically allocated data
  - E.g., global vars, `static` vars, string constants
- **Text / Shared Libraries**
  - Executable machine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



# x86-64 Stack

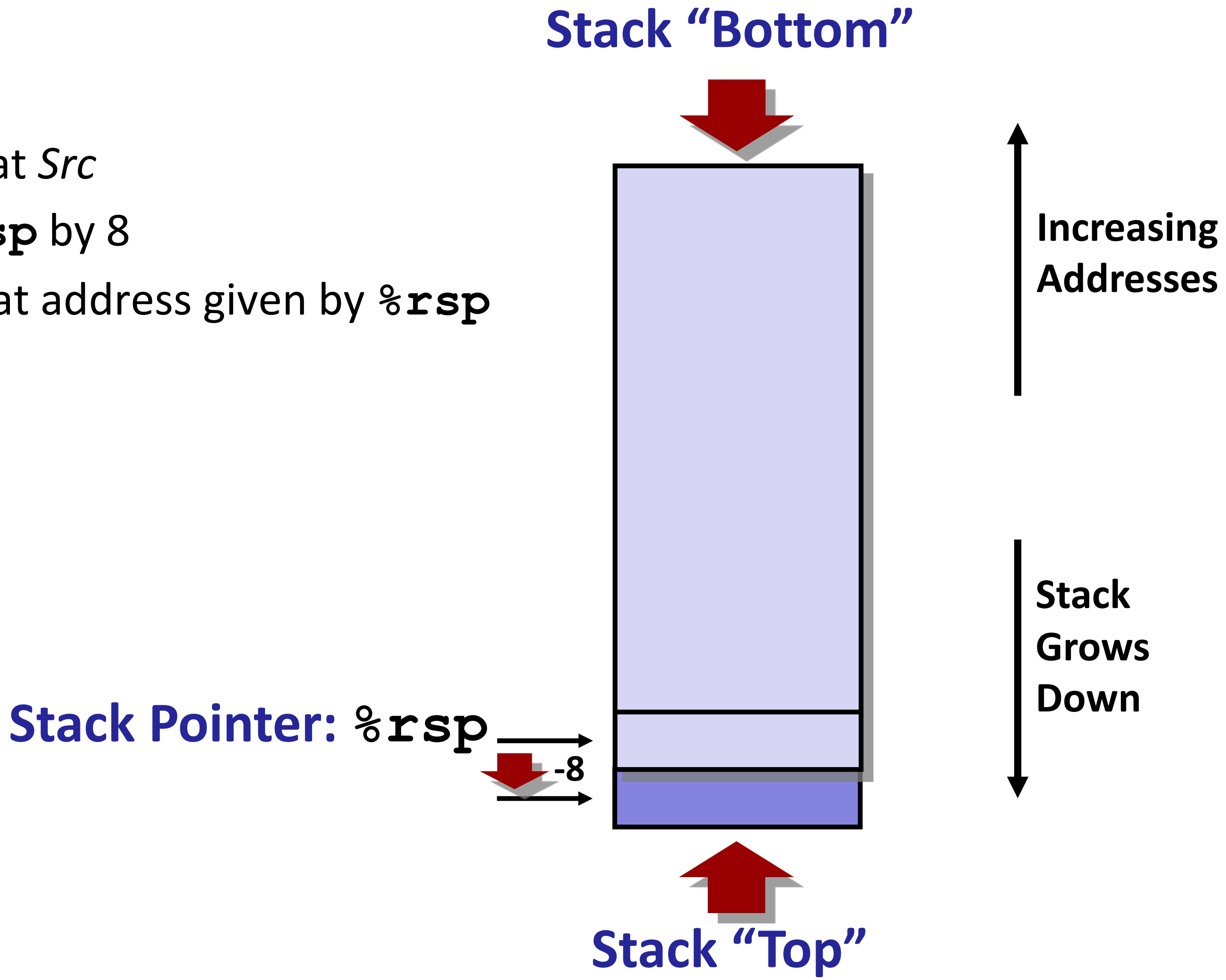
- Region of memory managed with stack discipline
- Grows toward lower addresses

- Register `%rsp` contains lowest stack address
  - address of “top” element



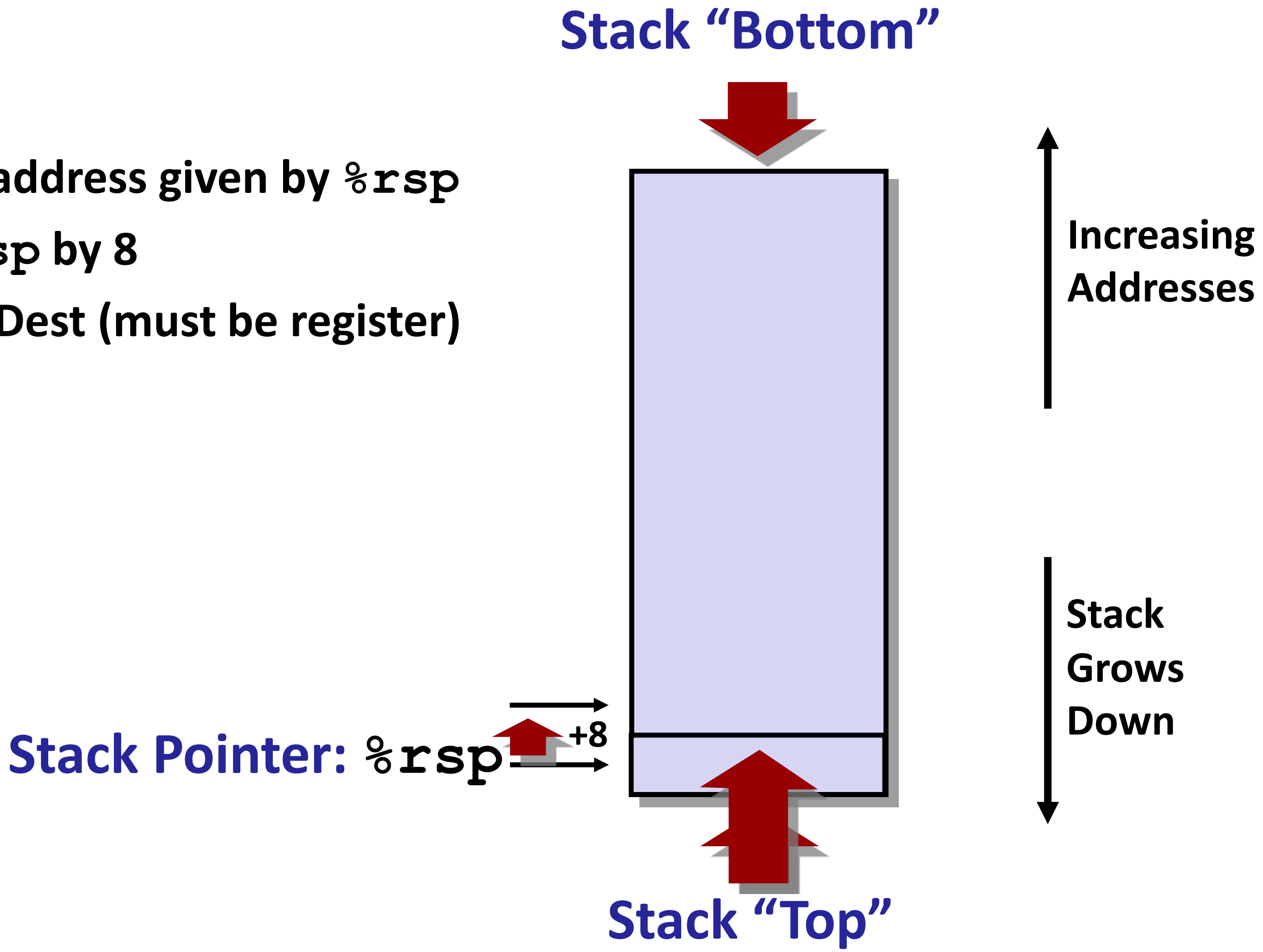
# x86-64 Stack: Push

- **`pushq`** ***Src***
  - Fetch operand at *Src*
  - Decrement **`%rsp`** by 8
  - Write operand at address given by **`%rsp`**



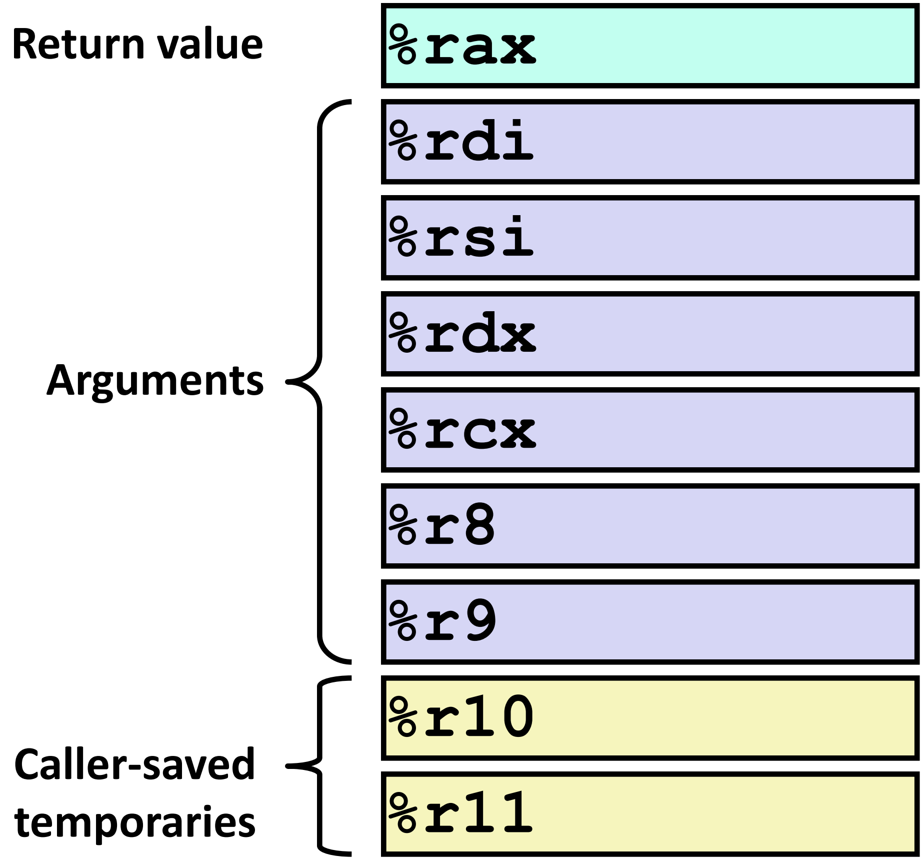
# x86-64 Stack: Pop

- `popq` ***Dest***
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at Dest (must be register)



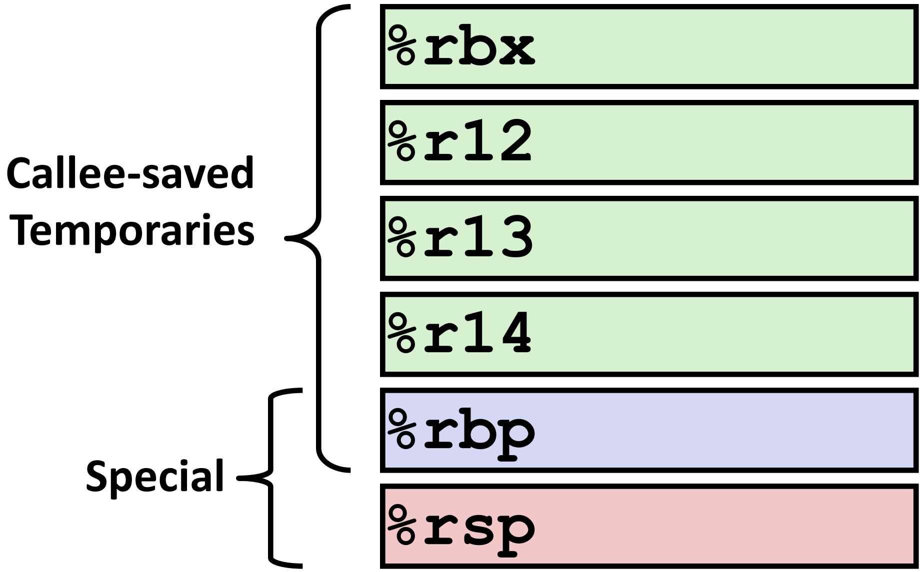
# x86-64 Linux Register Usage #1

- `%rax`
  - Return value
  - Also caller-saved
  - Can be modified by procedure
- `%rdi`, …, `%r9`
  - Arguments
  - Also caller-saved
  - Can be modified by procedure
- `%r10`, `%r11`
  - Caller-saved
  - Can be modified by procedure



# x86-64 Linux Register Usage #2

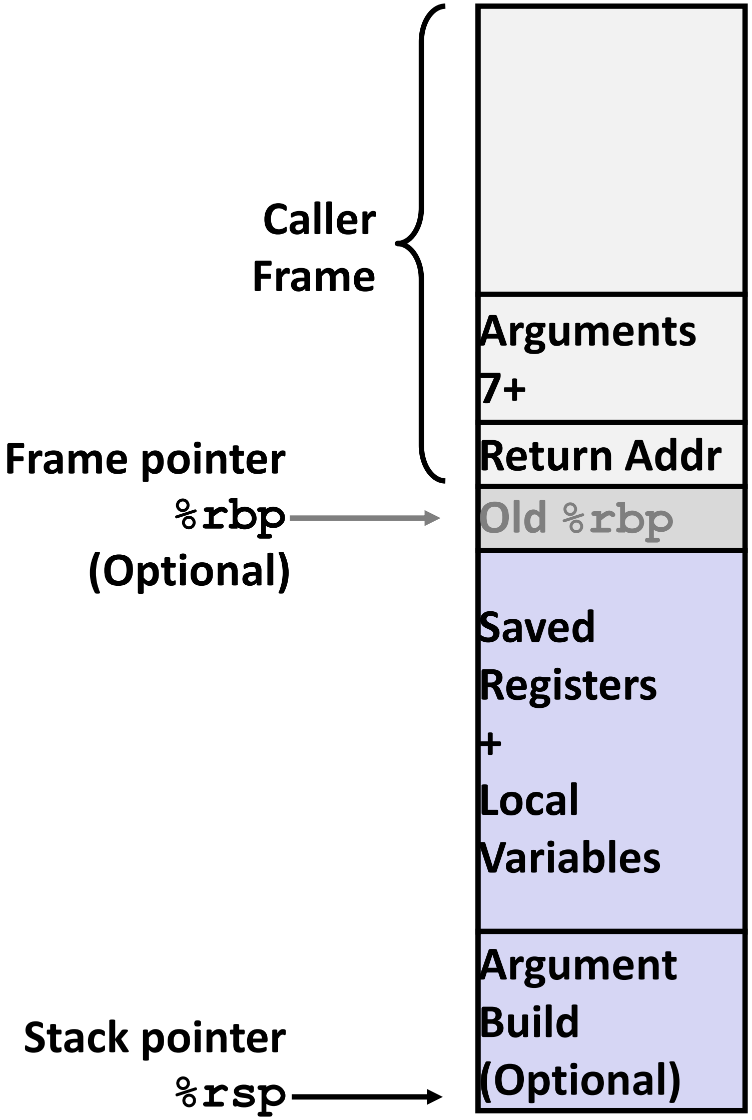
- `%rbx`, `%r12`, `%r13`, `%r14`
  - Callee-saved
  - Callee must save & restore
- `%rbp`
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer
  - Can mix & match
- `%rsp`
  - Special form of callee save
  - Restored to original value upon exit from procedure



# x86-64/Linux Stack Frame

- **Current Stack Frame ("Top" to Bottom)**
  - "Argument build:"
    Parameters for function about to call
  - Local variables
    If can't keep in registers
  - Saved register context
  - Old frame pointer (optional)

- **Caller Stack Frame**
  - Return address
    - Pushed by **`call`** instruction
  - Arguments for this call



# Today

- X86-64 Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
- Buffer Overflow
  - Vulnerability
  - Protection

# Recall: Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

```
fun(0)  ➙  3.14
fun(1)  ➙  3.14
fun(2)  ➙  3.1399998664856
fun(3)  ➙  2.00000061035156
fun(4)  ➙  3.14
fun(6)  ➙  Segmentation fault
```

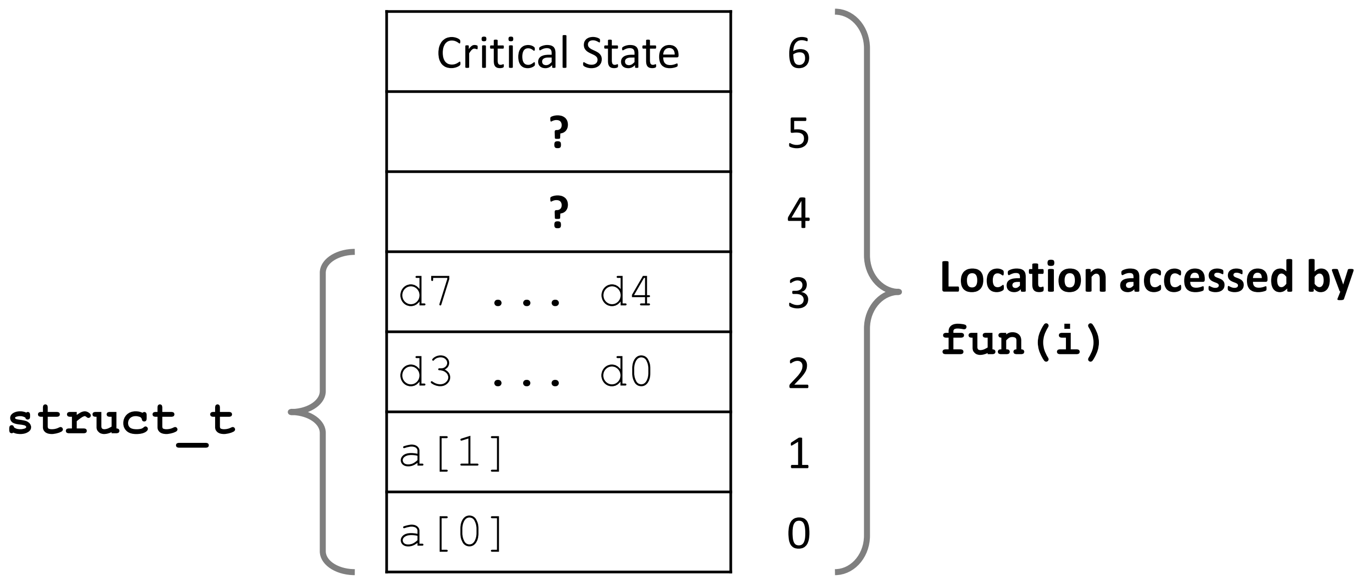
- Result is system specific

# Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;
```

`fun(0)` → 3.14
`fun(1)` → 3.14
`fun(2)` → 3.1399998664856
`fun(3)` → 2.00000061035156
`fun(4)` → 3.14
`fun(6)` → Segmentation fault

## Explanation:



# Such problems are a BIG deal

- **Generally called a "buffer overflow"**
  - when exceeding the memory size allocated for an array
- **Why a big deal?**
  - It's the #1 technical cause of security vulnerabilities
    - #1 overall cause is social engineering / user ignorance
- **Most common form**
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack
    - sometimes referred to as stack smashing

# String Library Code

- **Implementation of Unix function `gets()`**

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

  - **No way to specify limit on number of characters to read**

- **Similar problems with other library functions**
  - **`strcpy`, `strcat`**: Copy strings of arbitrary length

# Vulnerable Buffer Code

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

←btw, how big is big enough?

```
void call_echo() {
    echo();
}
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
012345678901234567890123
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901234
Segmentation Fault
```

# Buffer Overflow Disassembly

## echo:

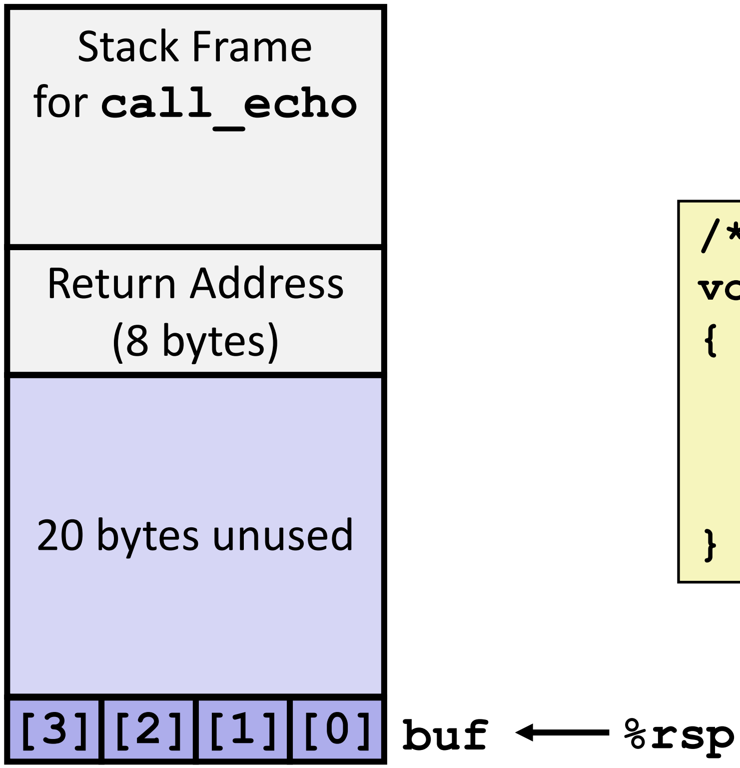
```
00000000004006cf <echo>:
  4006cf:  48 83 ec 18              sub    $0x18,%rsp
  4006d3:  48 89 e7                 mov    %rsp,%rdi
  4006d6:  e8 a5 ff ff ff           callq  400680 <gets>
  4006db:  48 89 e7                 mov    %rsp,%rdi
  4006de:  e8 3d fe ff ff           callq  400520 <puts>
  4006e3:  48 83 c4 18              add    $0x18,%rsp
  4006e7:  c3                       retq
```

## call_echo:

```
 4006e8:   48 83 ec 08              sub    $0x8,%rsp
  4006ec:  b8 00 00 00 00           mov    $0x0,%eax
  4006f1:  e8 d9 ff ff ff           callq  4006cf <echo>
  4006f6:  48 83 c4 08              add    $0x8,%rsp
  4006fa:  c3                       retq
```

# Buffer Overflow Stack

*Before call to gets*

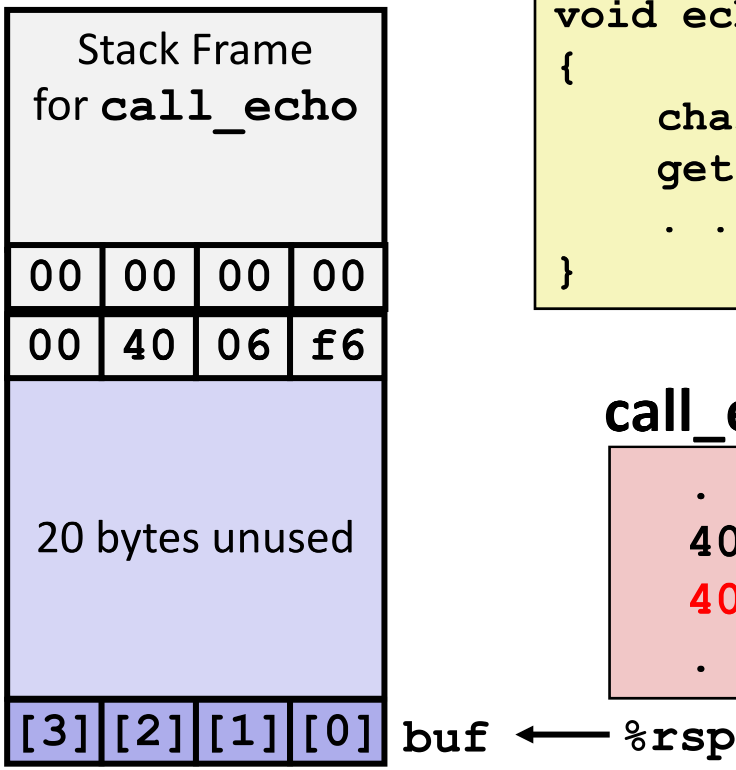



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

# Buffer Overflow Stack Example

*Before call to gets*




```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

**call_echo:**

```
  . . .
  4006f1:  callq   4006cf <echo>
  4006f6:  add     $0x8,%rsp
  . . .
```

# Buffer Overflow Stack Example #1

*After call to gets*

| Stack Frame for `call_echo` | | | |
|---|---|---|---|
| 00 | 00 | 00 | 00 |
| 00 | 40 | 06 | f6 |
| 00 | 32 | 31 | 30 |
| 39 | 38 | 37 | 36 |
| 35 | 34 | 33 | 32 |
| 31 | 30 | 39 | 38 |
| 37 | 36 | 35 | 34 |
| 33 | 32 | 31 | 30 |

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

**call_echo:**

```
  . . .
  4006f1:  callq  4006cf <echo>
  4006f6:  add    $0x8,%rsp
  . . .
```

```
unix>./bufdemo-nsp
Type a string:01234567890123456789012
01234567890123456789012
```

**Overflowed buffer, but did not corrupt state**

# Buffer Overflow Stack Example #2

*After call to gets*

| Stack Frame for `call_echo` | | | |
|---|---|---|---|
| 00 | 00 | 00 | 00 |
| 00 | 40 | 00 | 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 |

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

**call_echo:**

```
  . . .
  4006f1:  callq  4006cf <echo>
  4006f6:  add    $0x8,%rsp
  . . .
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123 4
Segmentation Fault
```

**Overflowed buffer and corrupted return pointer**

# Buffer Overflow Stack Example #3

*After call to gets*

| Stack Frame for `call_echo` | | | |
|---|---|---|---|
| 00 | 00 | 00 | 00 |
| 00 | 40 | 06 | 00 |
| 33 | 32 | 31 | 30 |
| 39 | 38 | 37 | 36 |
| 35 | 34 | 33 | 32 |
| 31 | 30 | 39 | 38 |
| 37 | 36 | 35 | 34 |
| 33 | 32 | 31 | 30 |

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
   subq   $24, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

**call_echo:**

```
   . . .
   4006f1:   callq   4006cf <echo>
   4006f6:   add     $0x8,%rsp
   . . .
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
012345678901234567890123
```

**Overflowed buffer, corrupted return pointer, but program seems to work!**

# Buffer Overflow Stack Example #3 Explained

*After call to gets*

| Stack Frame for `call_echo` | | | |
|---|---|---|---|
| 00 | 00 | 00 | 00 |
| 00 | 40 | 06 | 00 |
| 33 | 32 | 31 | 30 |
| 39 | 38 | 37 | 36 |
| 35 | 34 | 33 | 32 |
| 31 | 30 | 39 | 38 |
| 37 | 36 | 35 | 34 |
| 33 | 32 | 31 | 30 |

`buf` ← `%rsp`

**register_tm_clones:**

```
   . . .
400600:  mov    %rsp,%rbp
400603:  mov    %rax,%rdx
400606:  shr    $0x3f,%rdx
40060a:  add    %rdx,%rax
40060d:  sar    %rax
400610:  jne    400614
400612:  pop    %rbp
400613:  retq
```

**"Returns" to unrelated code**

**Lots of things happen, without modifying critical state**

**Eventually executes `retq` back to `main`**

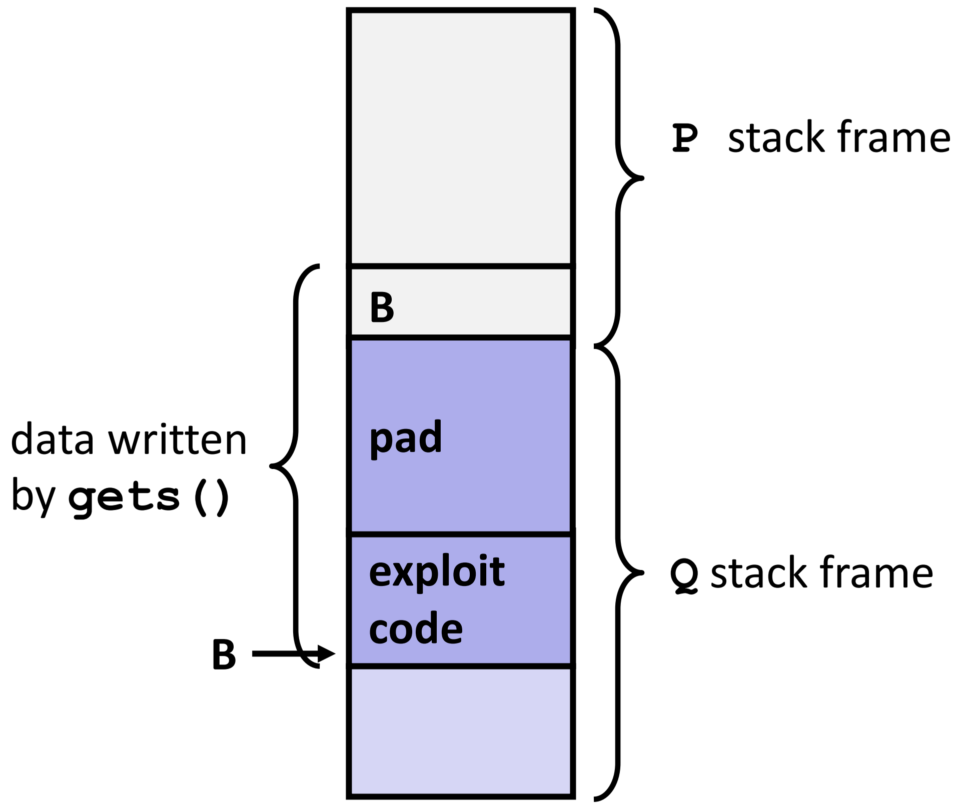
# Code Injection Attacks

```
void P(){
  Q();
  ...
}
```

return address A

```
int Q() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```

Stack after call to **gets()**




- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When Q executes `ret`, will jump to exploit code

# Exploits Based on Buffer Overflows

- ***Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines***
- **Distressingly common in real progams**
  - Programmers keep making the same mistakes ☹
  - Recent measures make these attacks much more difficult
- **Examples across the decades**
  - Original “Internet worm” (1988)
  - “IM wars” (1999)
  - Twilight hack on Wii (2000s)
  - … and many, many more

# Example: the original Internet worm (1988)

- **Exploited a few vulnerabilities to spread**
  - Early versions of the finger server (fingerd) used **`gets()`** to read the argument sent by the client:
    - **`finger droh@cs.cmu.edu`**
  - Worm attacked fingerd server by sending phony argument:
    - **`finger "`*`exploit-code padding new-return-address`*`"`**
    - exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.
- **Once on a machine, scanned for other machines to attack**
  - invaded ~6000 computers in hours (10% of the Internet ☺ )
    - see June 1989 article in *Comm. of the ACM*
  - the young author of the worm was prosecuted…
  - and CERT was formed… still homed at CMU

# What to do about buffer overflow attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use “stack canaries”
- Lets talk about each…

# 1. Avoid Overflow Vulnerabilities in Code (!)

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

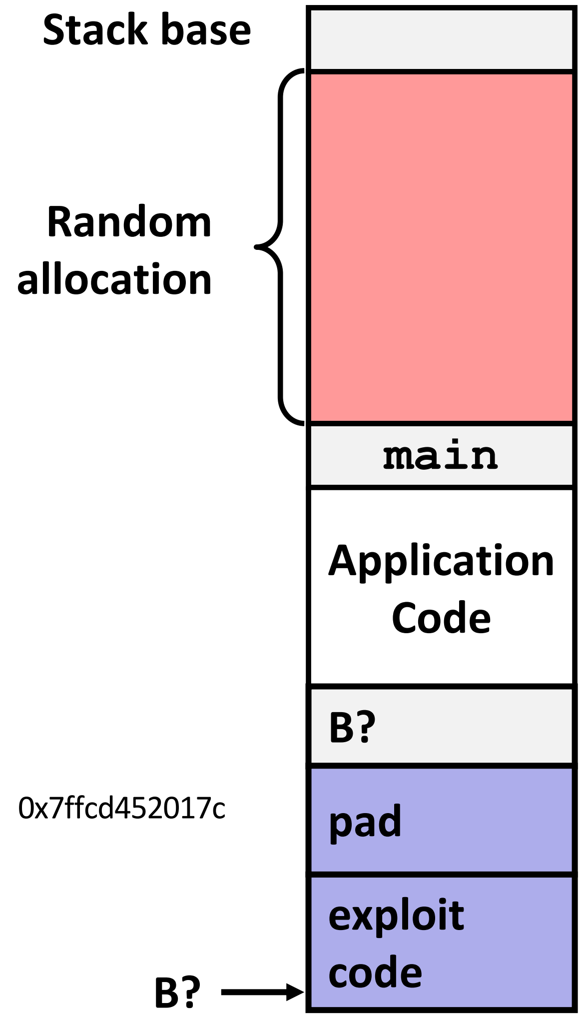
- **For example, use library routines that limit string lengths**
  - **`fgets`** instead of **`gets`**
  - **`strncpy`** instead of **`strcpy`**
  - Don't use **`scanf`** with **`%s`** conversion specification
    - Use **`fgets`** to read the string
    - Or use **`%ns`** where **`n`** is a suitable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

# 2. System-Level Protections can help

- **Randomized stack offsets**
- **栈随机化**
  - At start of program, allocate random amount of space on stack
  - Shifts stack addresses for entire program
  - Makes it difficult for hacker to predict beginning of inserted code
  - E.g.: 5 executions of memory allocation code

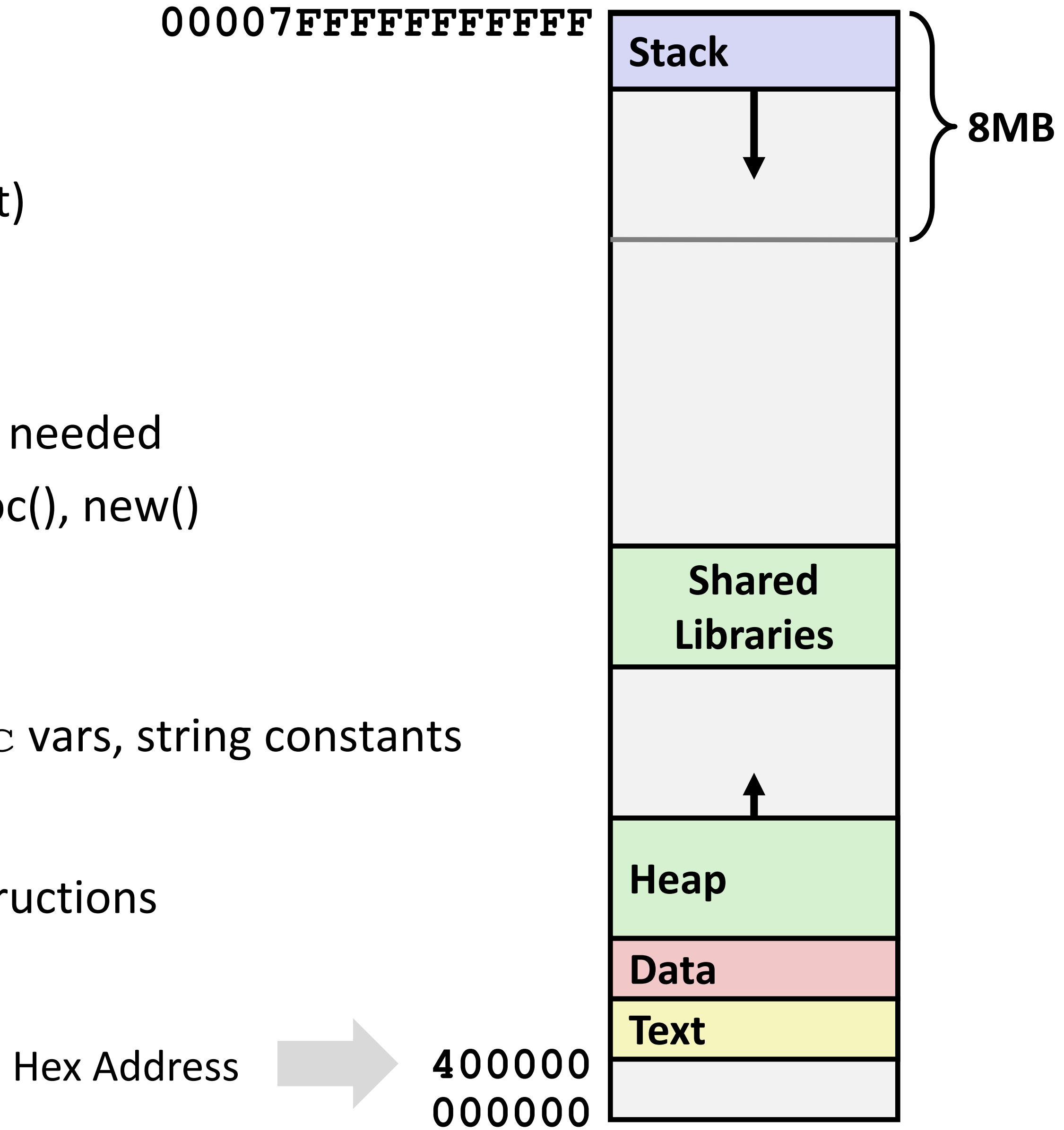
| | | | | | |
|---|---|---|---|---|---|
| local | 0x7ffe4d3be87c | 0x7fff75a4f9fc | 0x7ffeadb7c80c | 0x7ffeaea2fdac | 0x7ffcd452017c |

  - Stack repositioned each time program executes



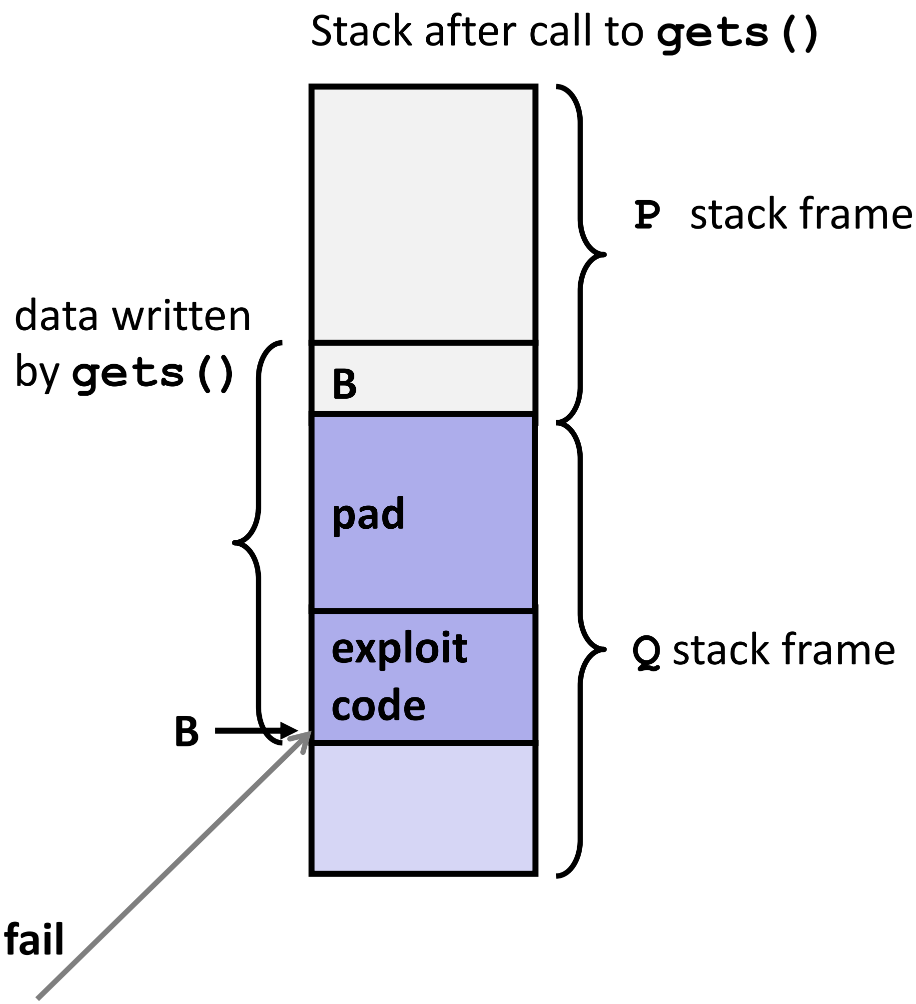
# x86-64 Linux Memory Layout

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



# 2. System-Level Protections can help

- **Non-executable code segments**
- **限制可执行代码区域**
- **In traditional x86, can mark region of memory as either "read-only" or "writeable"**
  - Can execute anything readable
- **X86-64 added explicit "execute" permission**
- **Stack marked as non-executable**




**Any attempt to execute this code will fail**

# 3. Stack Canaries can help

- **Idea：栈破坏检测**
  - Place special value ("canary") on stack just beyond buffer
  - Check for corruption before exiting function
- **GCC Implementation**
  - `-fstack-protector`
  - Now the default (disabled earlier)

```
unix>./bufdemo-sp
Type a string:0123456
0123456
```

```
unix>./bufdemo-sp
Type a string:01234567
*** stack smashing detected ***
```

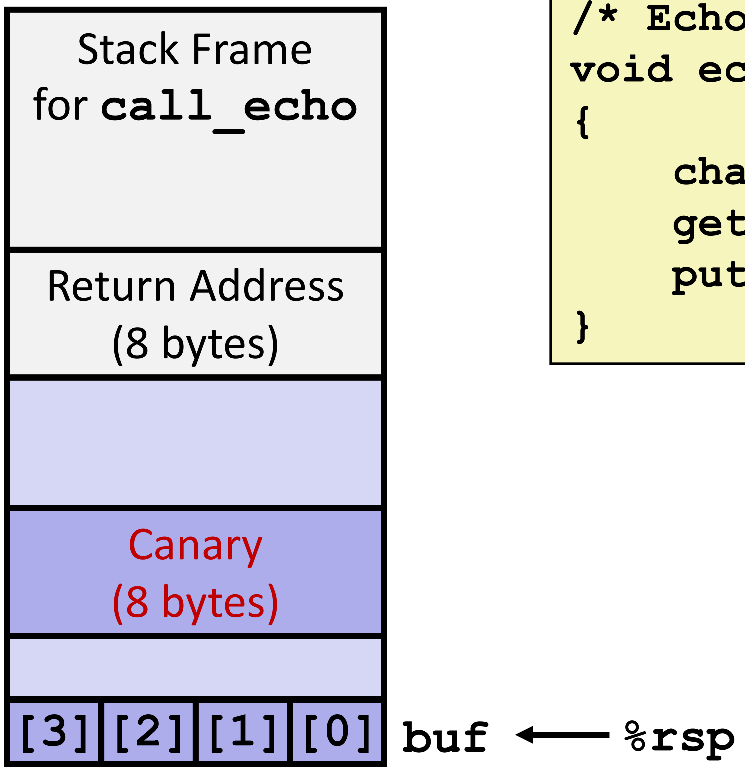
# Protected Buffer Disassembly

**echo:**

```
  40072f:  sub    $0x18,%rsp
  400733:  mov    %fs:0x28,%rax
  40073c:  mov    %rax,0x8(%rsp)
  400741:  xor    %eax,%eax
  400743:  mov    %rsp,%rdi
  400746:  callq  4006e0 <gets>
  40074b:  mov    %rsp,%rdi
  40074e:  callq  400570 <puts@plt>
  400753:  mov    0x8(%rsp),%rax
  400758:  xor    %fs:0x28,%rax
  400761:  je     400768 <echo+0x39>
  400763:  callq  400580 <__stack_chk_fail@plt>
  400768:  add    $0x18,%rsp
  40076c:  retq
```

# Setting Up Canary

*Before call to gets*

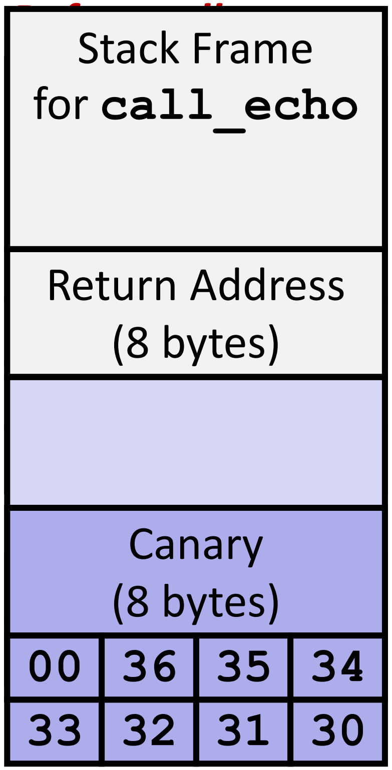



```
/* Echo Line */
void echo()
{
    char buf[4];   /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    movq    %fs: 0x28, %rax  # Get canary
    movq    %rax, 8(%rsp) # Place on stack
    xorl    %eax, %eax    # Erase canary
    . . .
```

# Checking Canary

*After call to gets*




buf ⟵ %rsp

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

**Input:** ***0123456***

```
echo:
    . . .
    movq     8(%rsp), %rax      # Retrieve from stack
    xorq     %fs:40, %rax       # Compare to canary
    je       .L6                # If same, OK
    call     __stack_chk_fail   # FAIL
.L6:         . . .
```

# Return-Oriented Programming Attacks

- **Challenge (for hackers)**
  - Stack randomization makes it hard to predict buffer location
  - Marking stack nonexecutable makes it hard to insert binary code
- **Alternative Strategy**
  - Use existing code
  - String together fragments（指令片段） to achieve overall desired outcome
  - *Does not overcome stack canaries*
- **Construct program from *gadgets***
  - Sequence of instructions ending in `ret`
    - Encoded by single byte `0xc3`
  - Code positions fixed from run to run
  - Code is executable

# Gadget Example #1

```
long ab_plus_c
   (long a, long b, long c)
{
    return a*b + c;
}
```

```
00000000004004d0 <ab_plus_c>:
  4004d0:  48 0f af fe  imul %rsi,%rdi
  4004d4:  48 8d 04 17  lea (%rdi,%rdx,1),%rax
  4004d8:  c3           retq
```

rax ← rdi + rdx

Gadget address = `0x4004d4`

- Use tail end of existing functions

# Gadget Example #2

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

Encodes `movq %rax, %rdi`

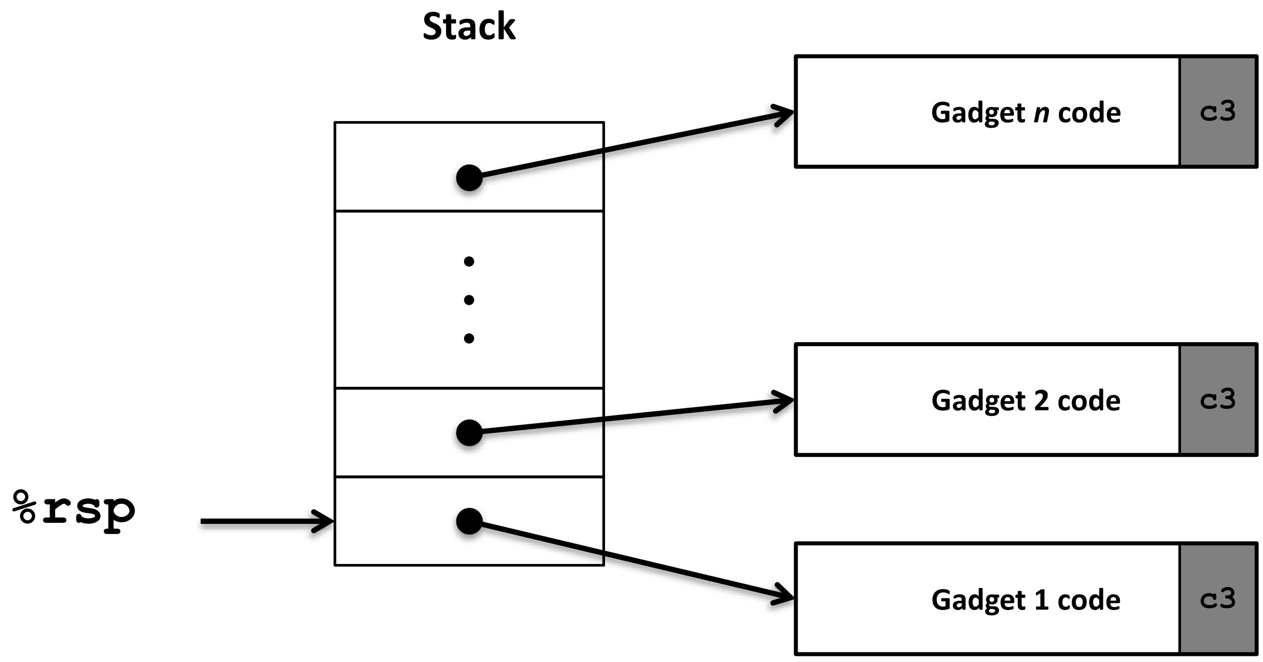
```
<setval>:
  4004d9:  c7 07 d4 48 89 c7   movl   $0xc78948d4,(%rdi)
  4004df:  c3                  retq
```

rdi ← rax

Gadget address = `0x4004dc`

- **Repurpose byte codes**

# ROP Execution




- **Trigger with `ret` instruction**
  - Will start executing Gadget 1
- **Final `ret` in each gadget will start next one**

# Gadget地址表格

| Gadget | addresses |
|---|---|
| pop ecx ; pop eax ;; | 0xB7F3BF30 |
| mov [eax] ecx ;; | 0xB7E8CECF |
| pop edx ;; | 0xB7E64A9E |
| mov [edx+0x18] eax ;; | 0xB7E8D722 |
| pop ebx ;; | 0xB7EDBAF4 |
| pop ecx ; pop edx ;; | 0xB7E8D6EB |
| xor eax eax ;; | 0xB7EDC6BF |
| add eax 0xb ;; | 0xB7EE2AA8 |
| call gs:[0x10] ;; | 0xB7F040F5 |

# Summary

- X86-64 Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
- Buffer Overflow
  - Vulnerability
  - Protection

# Next week

- CPU Design