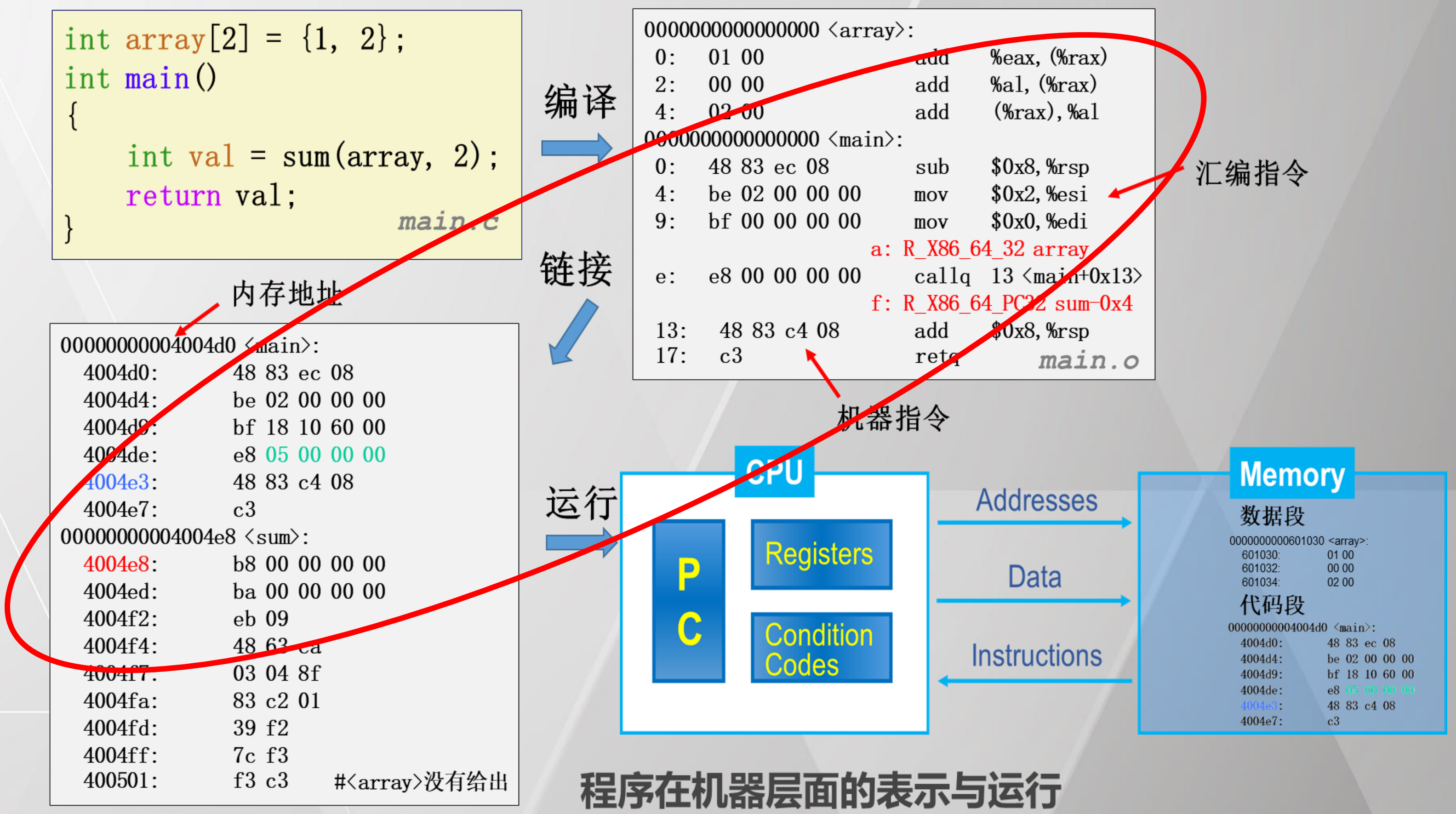
汇编语言程序设计

内存布局与缓冲区溢出

C程序在硬件层面的表示

- 数据/代码的内存地址定位
  - 链接（第九讲）
- 数据/代码的内存布局
  - 栈、堆等各类数据段以及代码段的layout（第十讲）
  - 缓冲区溢出等（第十讲）
- 讲解基本调试工具（GDB）的使用

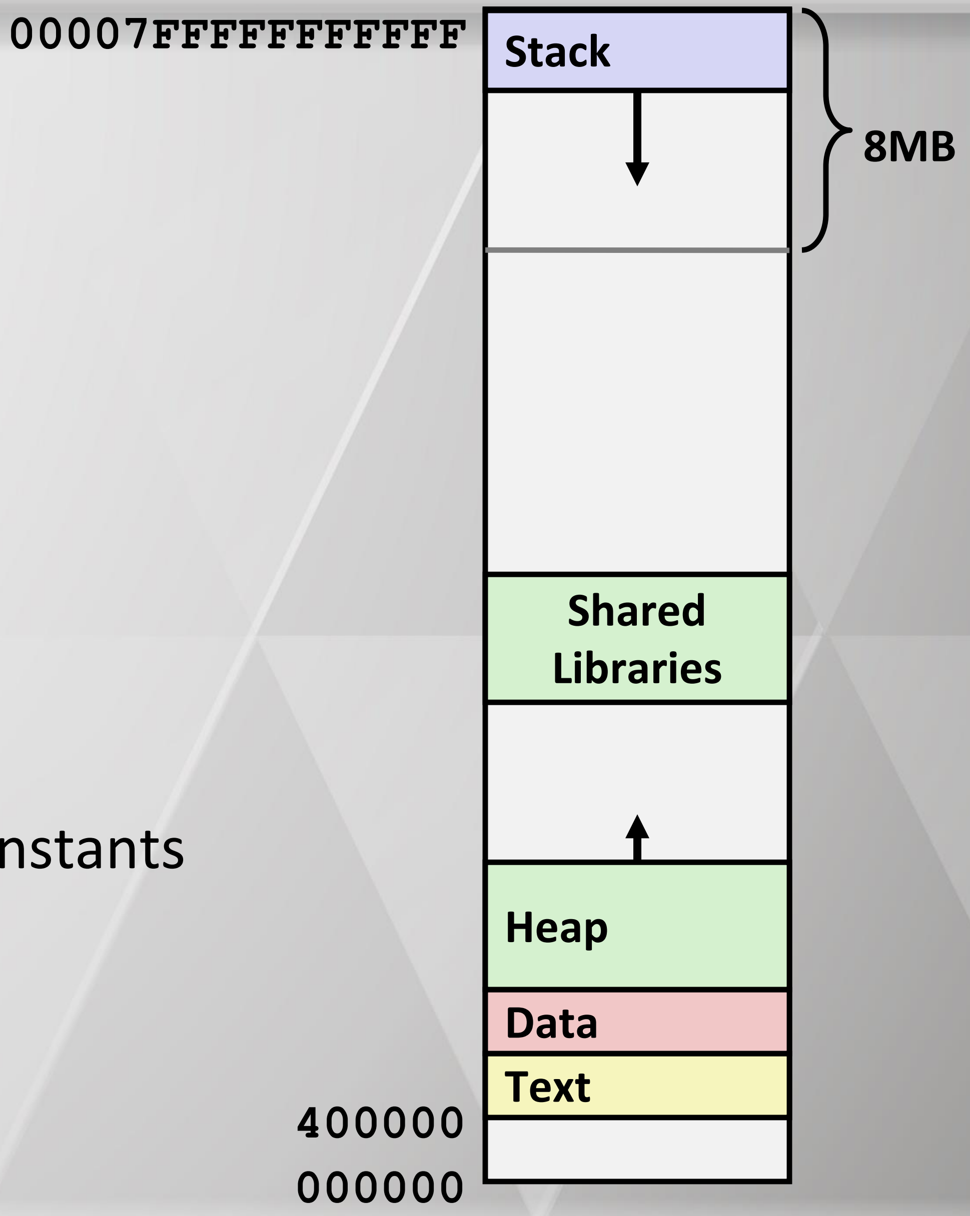


程序在机器层面的表示与运行

- **内存布局（memory layout）**
- 缓冲区溢出（buffer overflow）

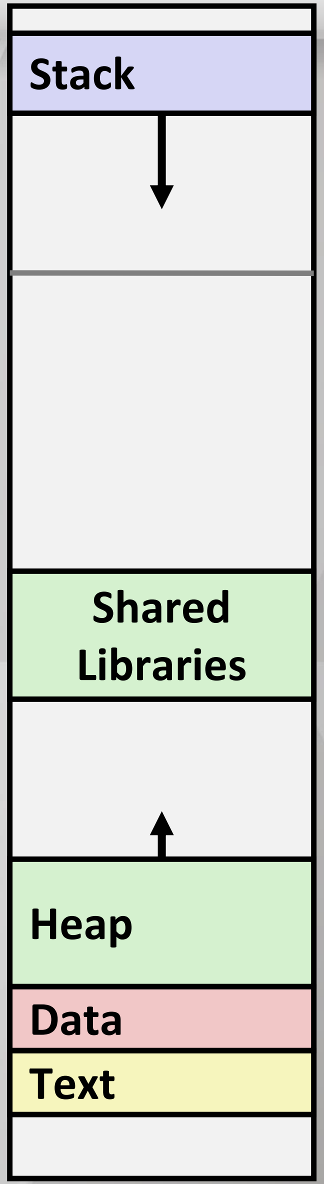
Linux进程的内存布局（x86-64）

- **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



内存分配示例

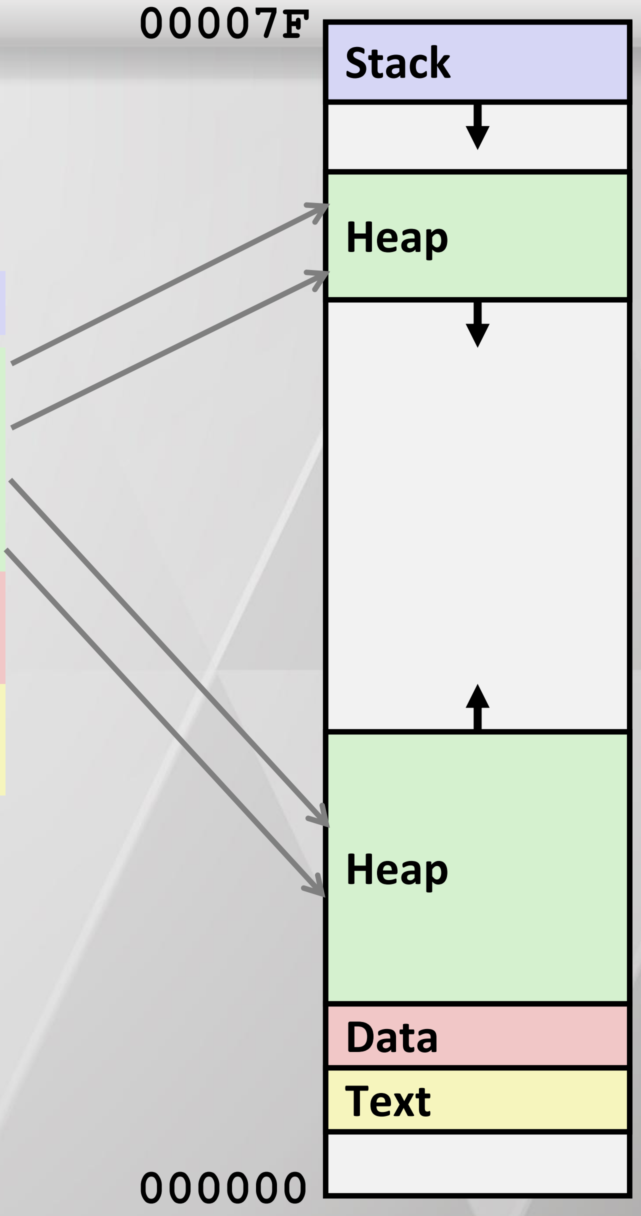
```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256  B */
    p3 = malloc(1L << 32); /*   4 GB */
    p4 = malloc(1L << 8);  /* 256  B */
 /* Some print statements ... */
}
```



示例地址

address range ~2^{47}

local	0x00007ffe4d3be87c
p1	0x00007f7262a1e010
p3	0x00007f7162a1d010
p4	0x000000008359d120
p2	0x000000008359d010
big_array	0x0000000080601060
huge_array	0x0000000000601060
main()	0x000000000040060c
useless()	0x0000000000400590



- 内存布局（memory layout）
- **缓冲区溢出（buffer overflow）**

缓冲区溢出

- **Implementation of Unix function `gets()`**
  - No way to specify limit on number of characters to read
- **Similar problems with other string library functions**
  - **`strcpy`**, **`strcat`**: Copy strings of arbitrary length
  - **`scanf`**, **`fscanf`**, **`sscanf`**, when given **`%s`** conversion specification

```
/* 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;
}
```

易受攻击的缓冲区相关代码

```
/* 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
```

echo:

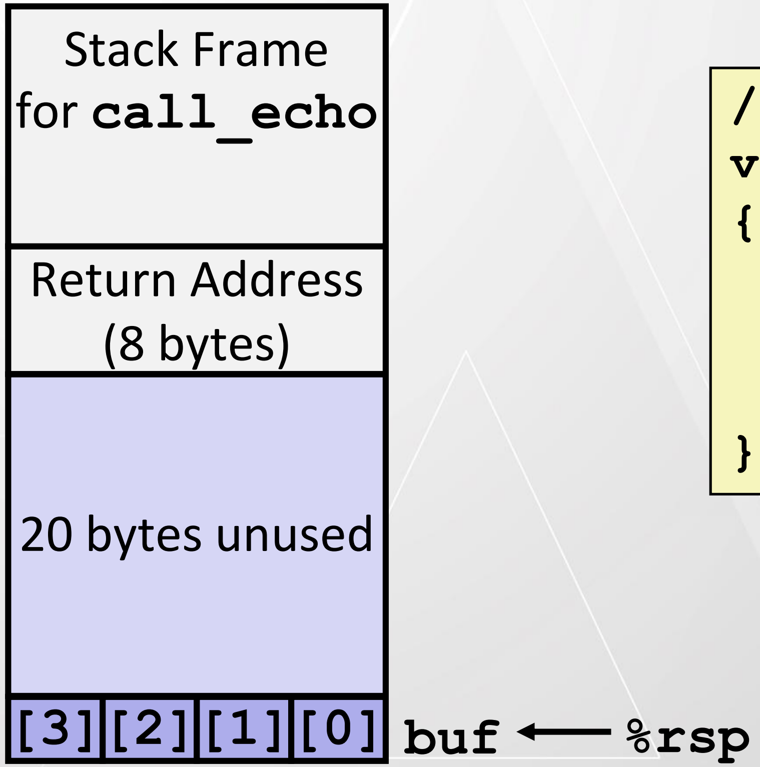
```
00000000004006cf <echo>:
 4006cf: 48 83 ec 18              sub    $0x18,%rsp
 4006d3: 48 89 e7                 mov    %rsp,%rdi
 4006d6: e8 a5 ff ff ff           callq  400680 <gets>
 4006db: 48 89 e7                 mov    %rsp,%rdi
 4006de: e8 3d fe ff ff           callq  400520 <puts@plt>
 4006e3: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

缓冲区溢出时的栈

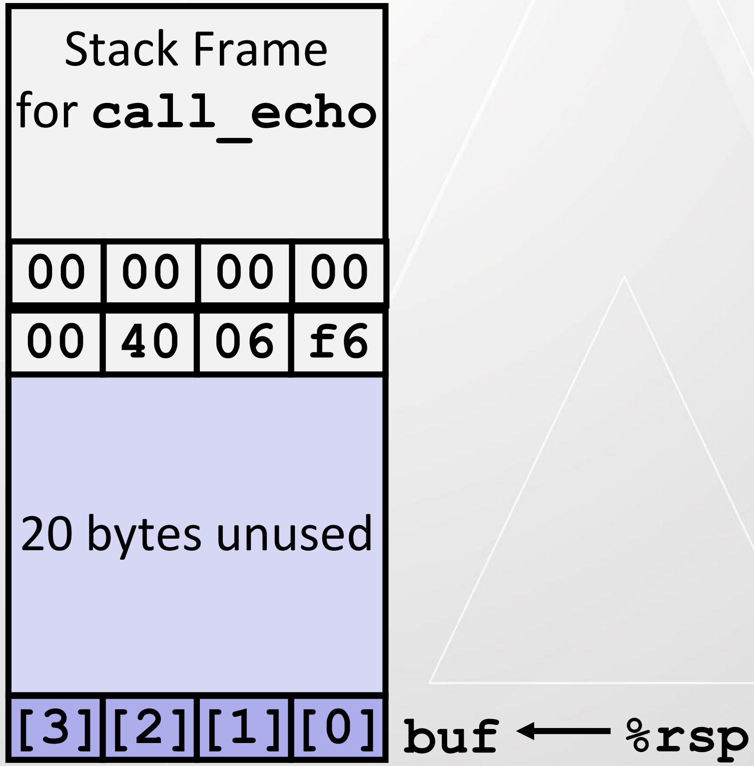
Before call to gets




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

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

Before call to gets



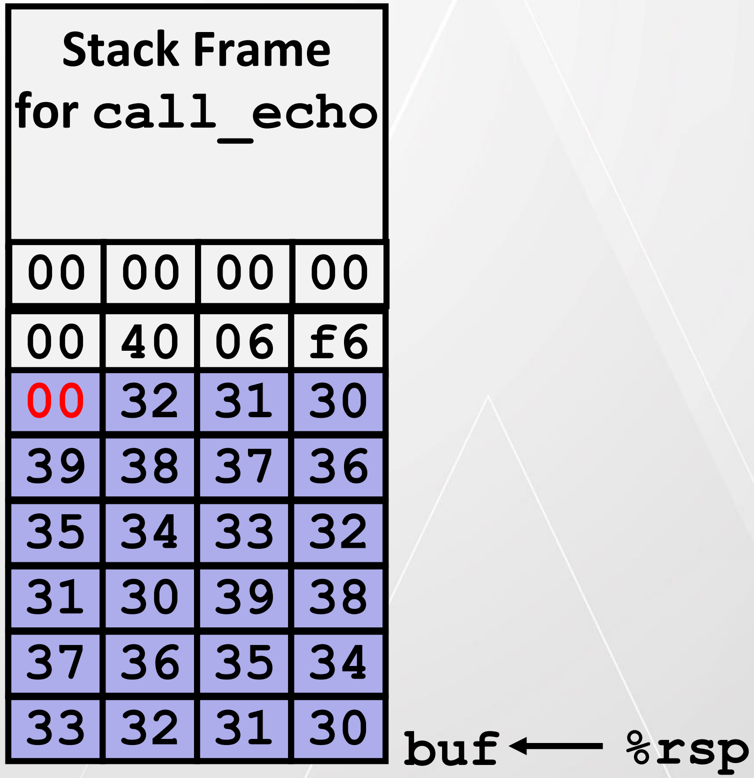

```
void echo()
{
      char buf[4];
      gets(buf);
      . . .
}
```

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

call_echo:

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

After 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
  . . .
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
0123456789012345678901
```

Overflowed buffer, but did not corrupt state

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
Segmentation Fault
```

Overflowed buffer and corrupted return pointer

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
  . . .
```

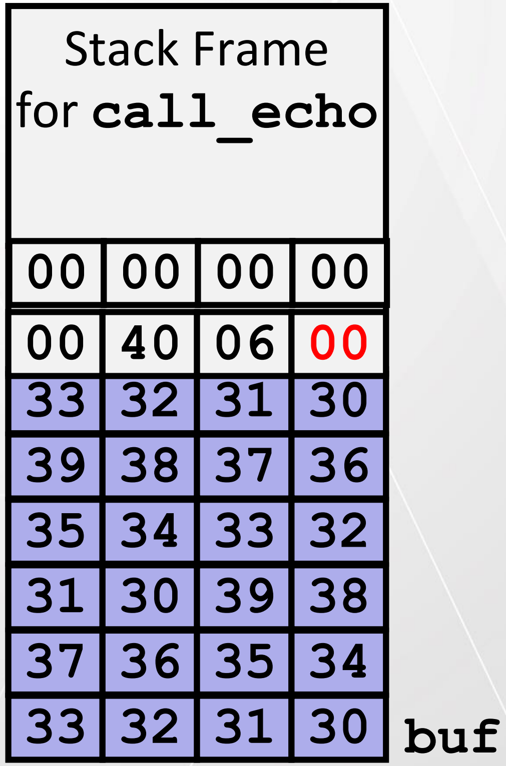
call_echo:

```
  . . .
  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!

After call to gets



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
```

← %rsp

"Returns" to unrelated code

Lots of things happen, without modifying critical state

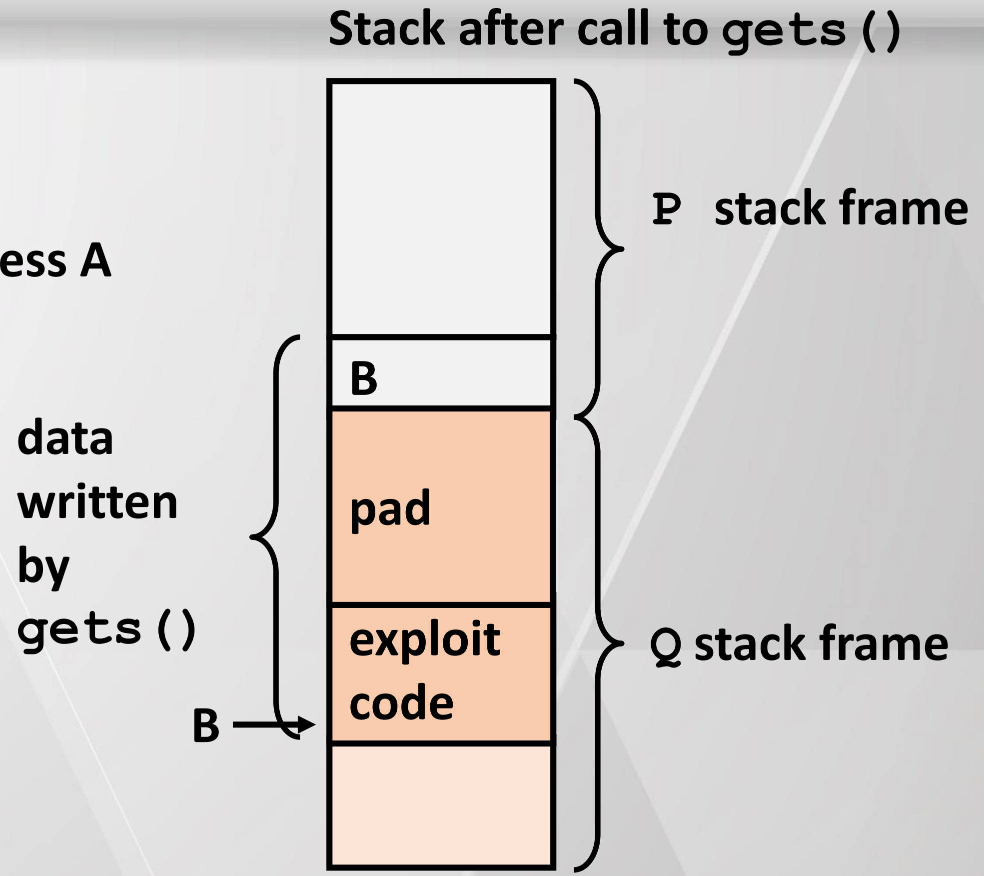
Eventually executes `retq` back to `main`

代码注入攻击

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

return address A

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



- 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

预防手段

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use “stack canaries”

1. Avoid Overflow Vulnerabilities in Code (!)

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

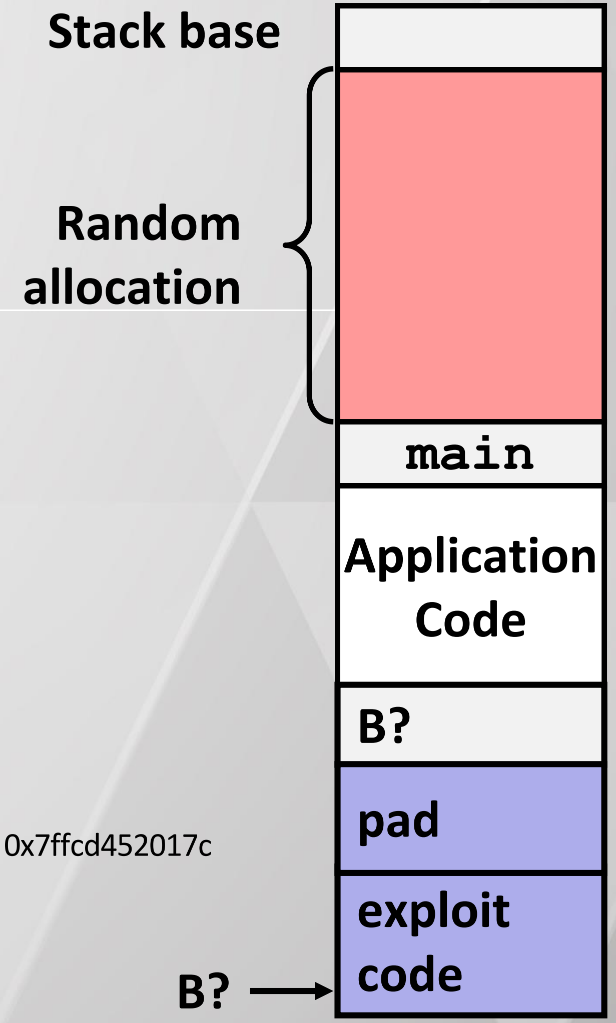
- **For example, use library routines that limit string lengths**
  - **`fgets`** instead of **`gets`**
  - **`strncpy`** instead of **`strcpy`**
  - Don't use **`scanf`** with **`%s`** conversion specification
    - Use **`fgets`** to read the string
    - Or use **`%ns`** where **`n`** is a suitable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

2. System-Level Protections can help

- **Randomized stack offsets**
  - 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

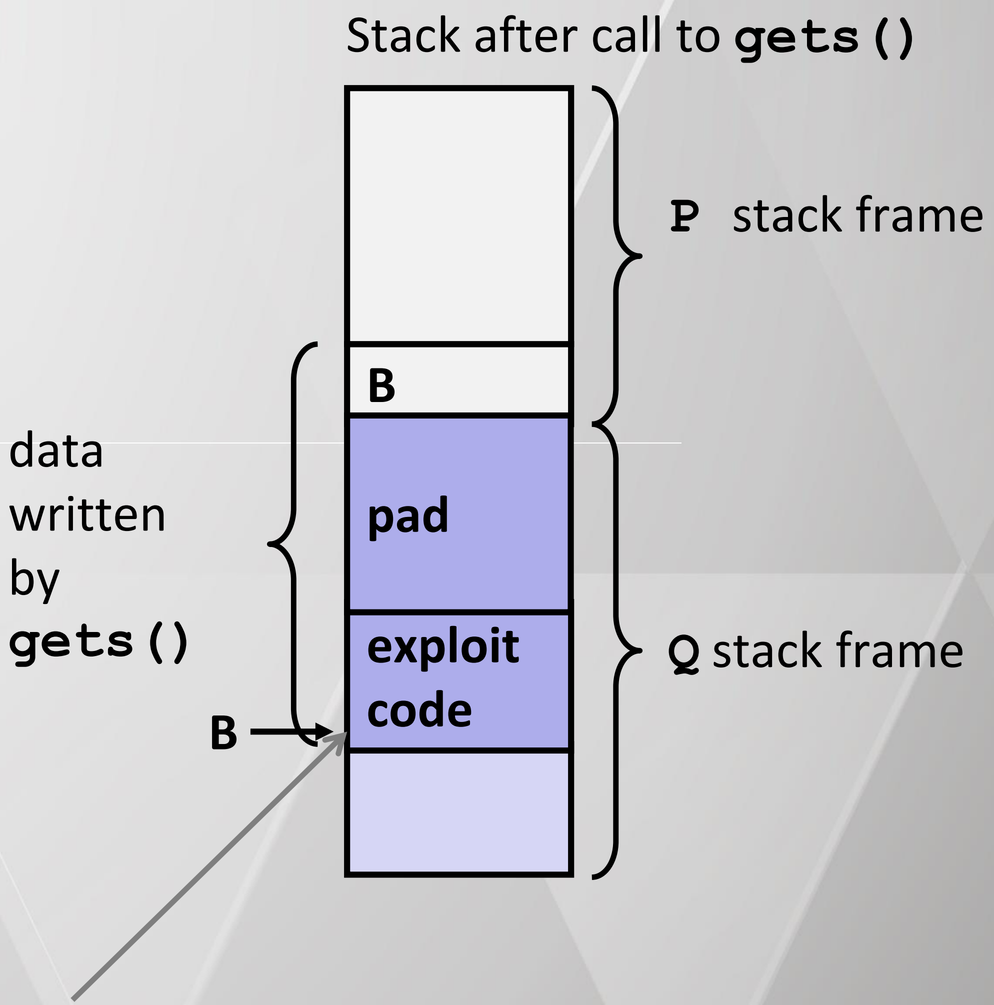
local	0x7ffe4d3be87c	0x7fff75a4f9fc	0x7ffeadb7c80c	0x7ffeaea2fdac	0x7ffcd452017c

    - Stack repositioned each time program executes



2. System-Level Protections can help

- **Nonexecutable code segments**
  - 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 (金丝雀) can help

- **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)

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

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

FS:0x28 on Linux is storing a special sentinel stack-guard value. This address is defined as *stack_chk_guard* in glibc, and the related code might look like this:

```
unsigned long __stack_chk_guard;
void __stack_chk_guard_setup(void)
{
     __stack_chk_guard = 0xBAAAAAAD;//provide some magic numbers
}

void __stack_chk_fail(void)
{
 /* Error message */
}// will be called when guard variable is corrupted
```

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

Checking Canary

Before call to gets

After call to gets

Return-Oriented Programming Attacks

- **Challenge (for hackers)**
  - Stack randomization makes it hard to predict buffer location
  - **Marking stack non-executable makes it hard to insert binary code**
- **Alternative Strategy**
  - Use existing code
    - E.g., library code from stdlib
  - String together fragments 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`

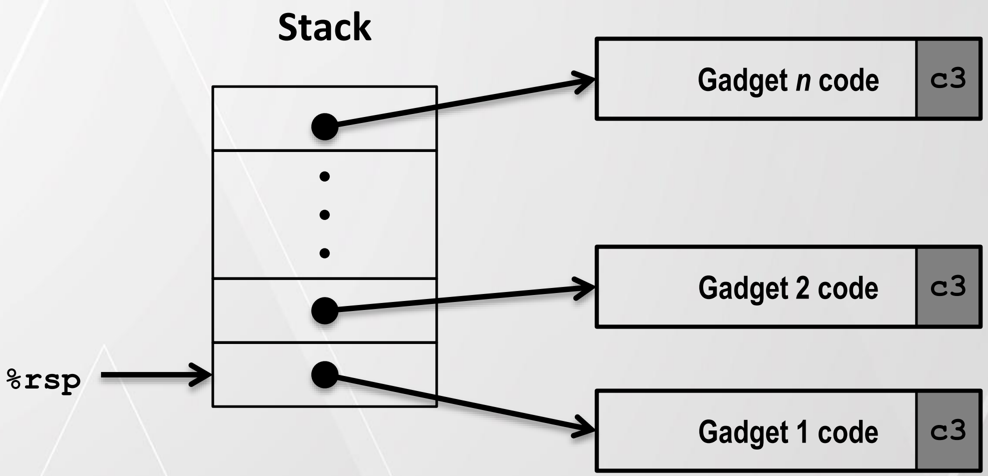
```
<setval>:
  4004d9:  c7 07 d4 48 89 c7  movl   $0xc78948d4,(%rdi)
  4004df:  c3                 retq
```

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**