Machine-Level Programming V: Advanced Topics

唐雯豪

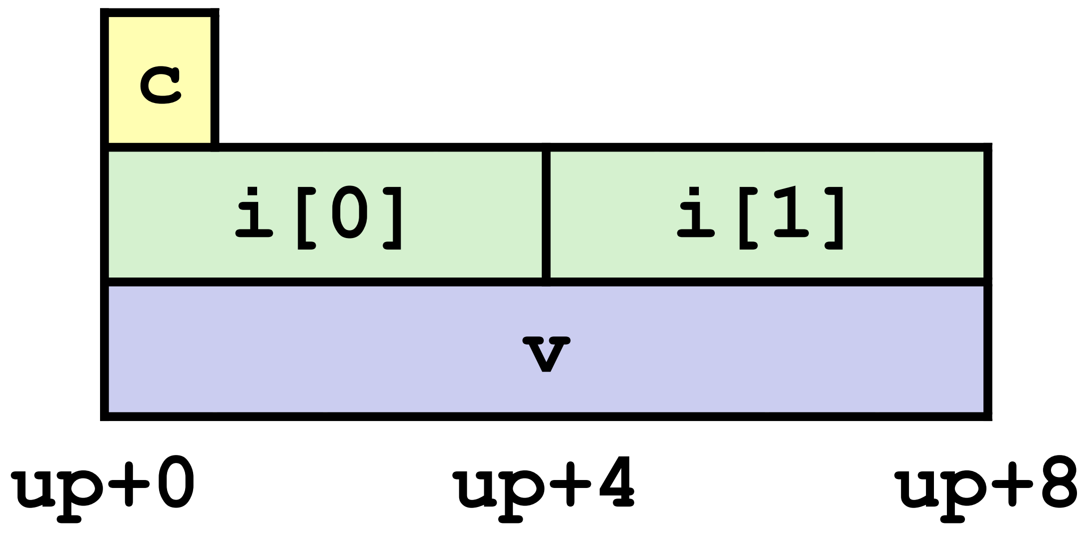
Outline

- Unions
- Memory Layout
- Buffer Overflow
- Variable-Size Stack Frames

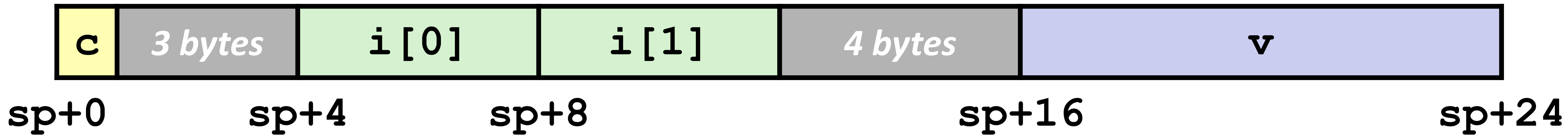
Unions

- Reference a single object according to multiple types.
- The overall size of a union equals the maximum size of any of its fields.
-

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```




```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



Byte Ordering

- **Big Endian**
  - Most significant byte has lowest address
  - Sparc
- **Little Endian**
  - Least significant byte has lowest address
  - Intel x86, ARM Android and IOS
- **Bi Endian**
  - Can be configured either way
  - ARM

Byte Ordering With Union

```
t.c                                                        buffers
#include <stdio.h>
union meow {
    unsigned char c[8];
    unsigned short s[4];
    unsigned int i[2];
    unsigned long long l;
} a;
int main() {
    a.l = 0x1020304050607080;
    for(int i=0; i<8; i++) printf("%x ", a.c[i]); puts("");
    for(int i=0; i<4; i++) printf("%x  ", a.s[i]); puts("");
    for(int i=0; i<2; i++) printf("%x    ", a.i[i]); puts("");
    printf("%llx ", a.l); puts("");
}
N…  t.c                                          c    7%  ☰   1:  1
```

```
~/shared/ics                                        ✔ 16:57:28
↳>>> gcc t.c -o t

~/shared/ics                                        ✔ 16:57:58
↳>>> ./t
80 70 60 50 40 30 20 10
7080  5060  3040  1020
50607080    10203040
1020304050607080
```

Little Endian

<table>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="8">l[0]</td></tr>
<tr><td>80</td><td>70</td><td>60</td><td>50</td><td>40</td><td>30</td><td>20</td><td>10</td></tr>
</table>

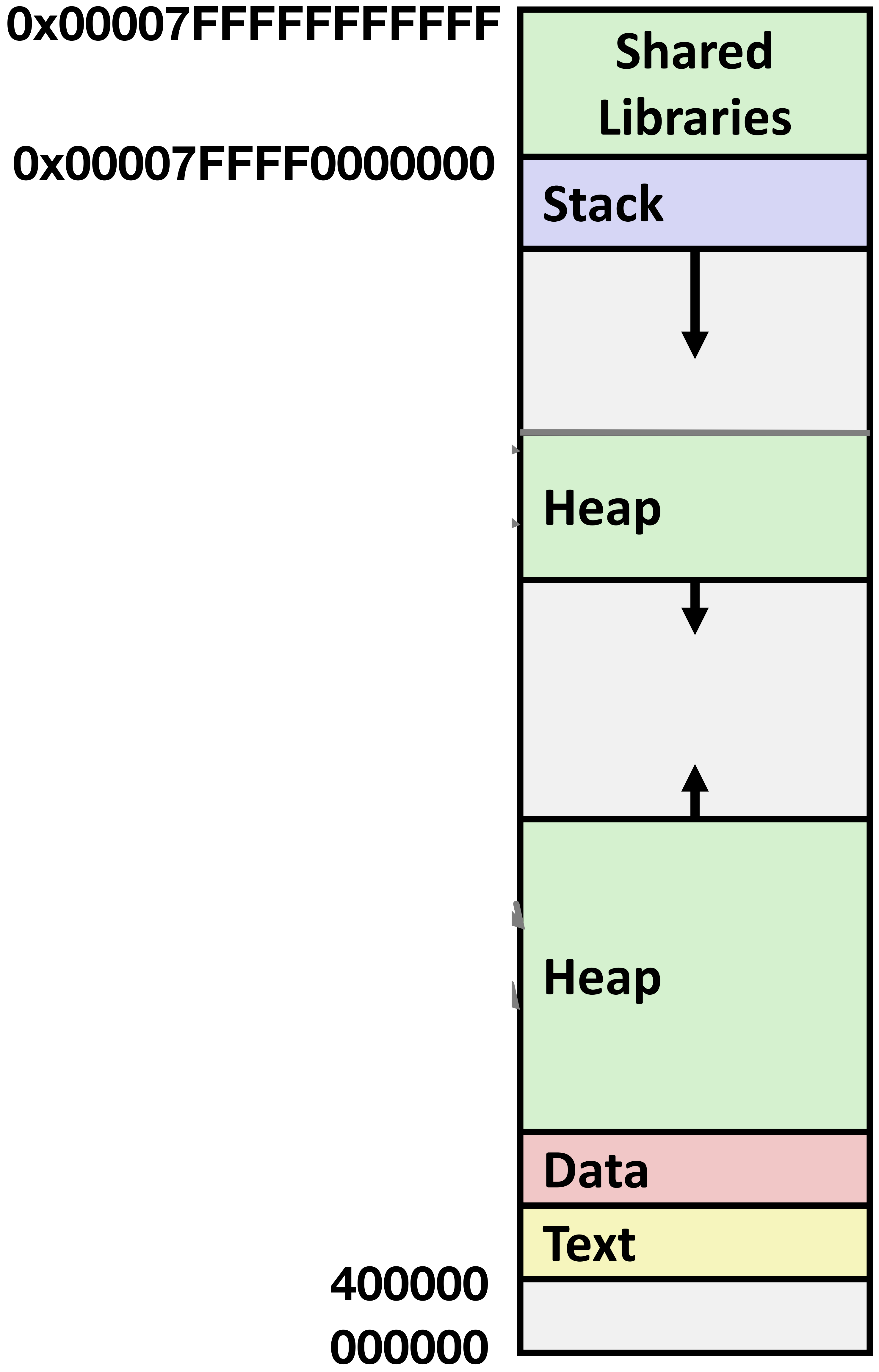
Big Endian

<table>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="8">l[0]</td></tr>
<tr><td>10</td><td>20</td><td>30</td><td>40</td><td>50</td><td>60</td><td>70</td><td>80</td></tr>
</table>

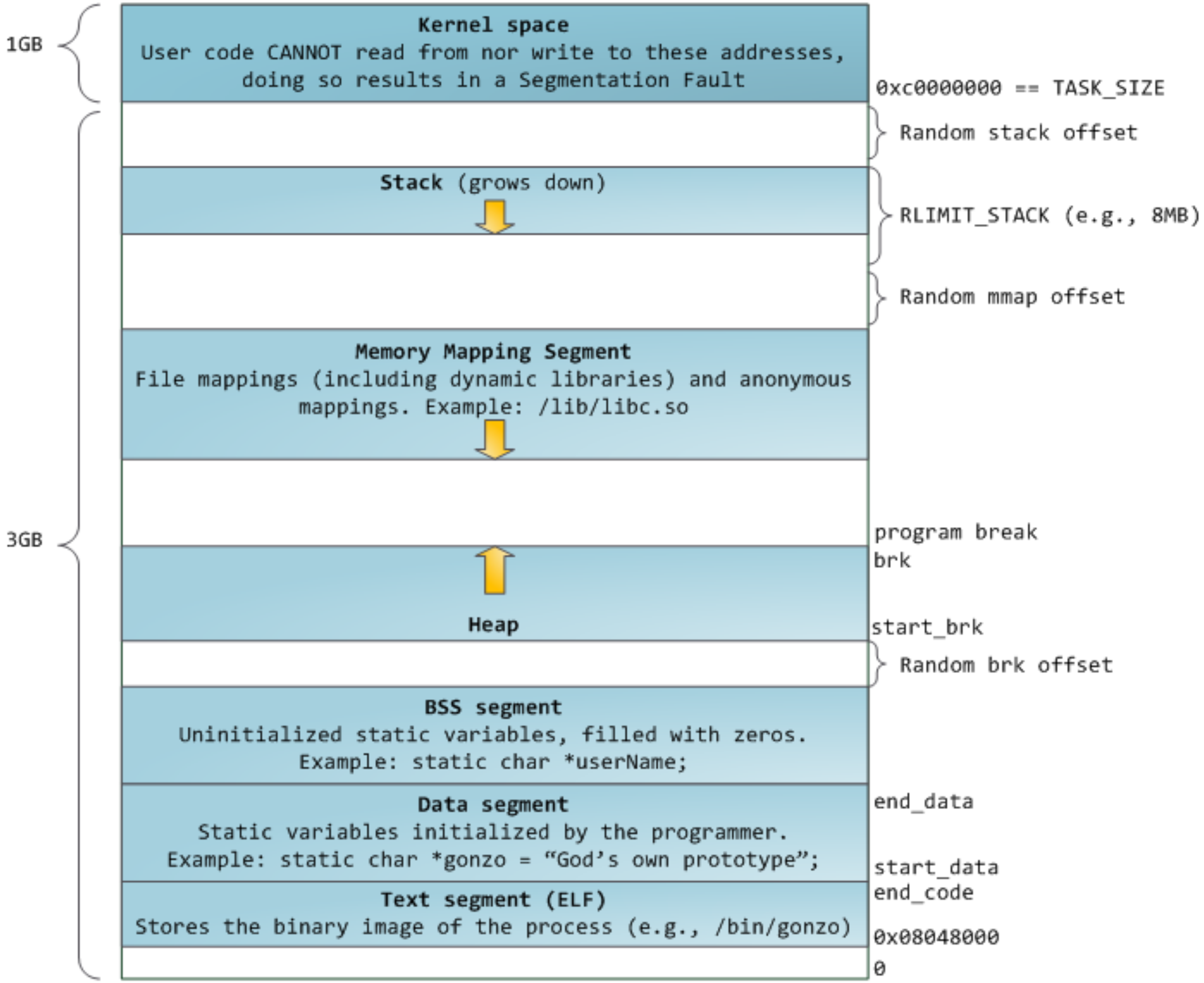
x86-64 Linux Memory Layout

使用虚拟地址，最高位置是$2^{48}-1$
(64位系统中只使用48位作为地址)

- Shared Libraries(Kernel Space)
- Stack
  - Runtime stack (8MB limit)
  - E.g., local variables
- Heap
  - Dynamically allocated as needed
  - Big data grows down, small data grows up.
- Data
  - Statically allocated data
- Text
  - The binary image of the process



Another Example



Buffer Overflow

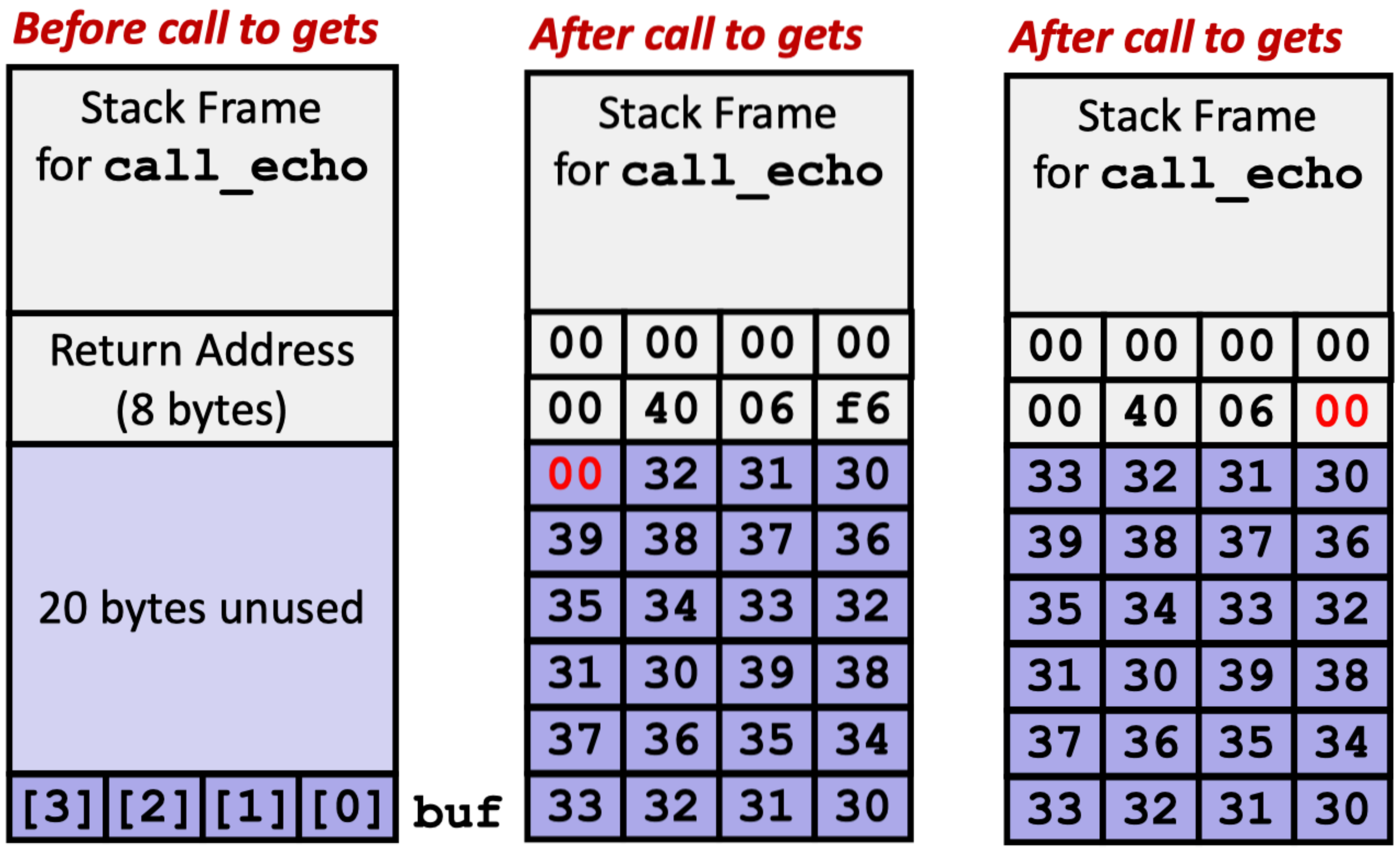
- Overruns the buffer's boundary and overwrites adjacent memory locations while writing data to a buffer(array).
- Most common reason:
  - Write to a character arrays on the stack without checking the boundary, for example many library functions like gets, strcpy, scanf, etc.
- Also referred to as **stack smashing**.

Vulnerable Buffer Code Example

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

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

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



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
```

Stack Smashing Attacks

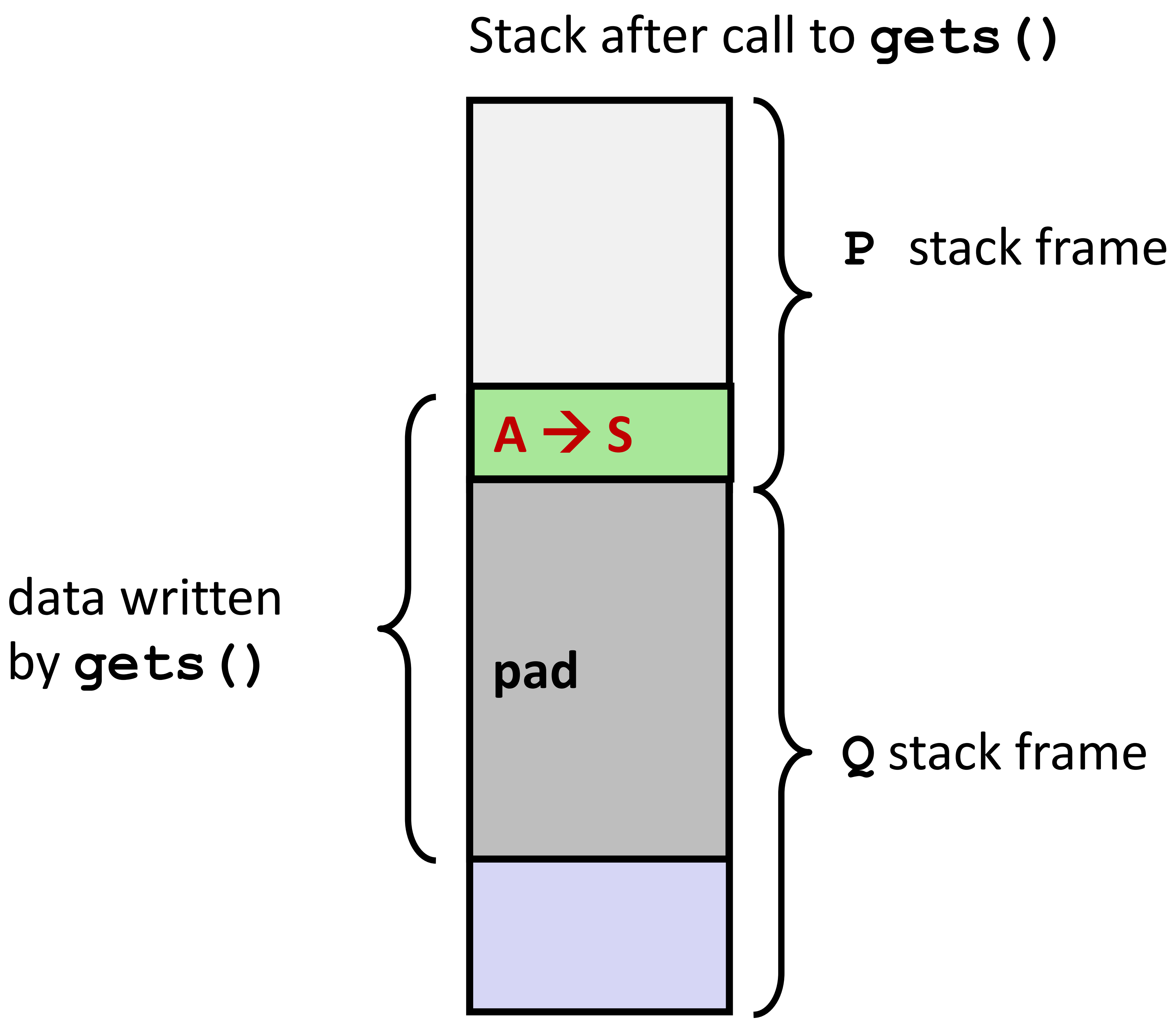
- **The core: get control of the instruction pointer**
- Overwrite normal return address A with address of some other code S (called exploit code).
- When Q executes ret, will jump to other code
-

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

return address A

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

```
void S(){
/* Something
   unexpected */
  ...
}
```



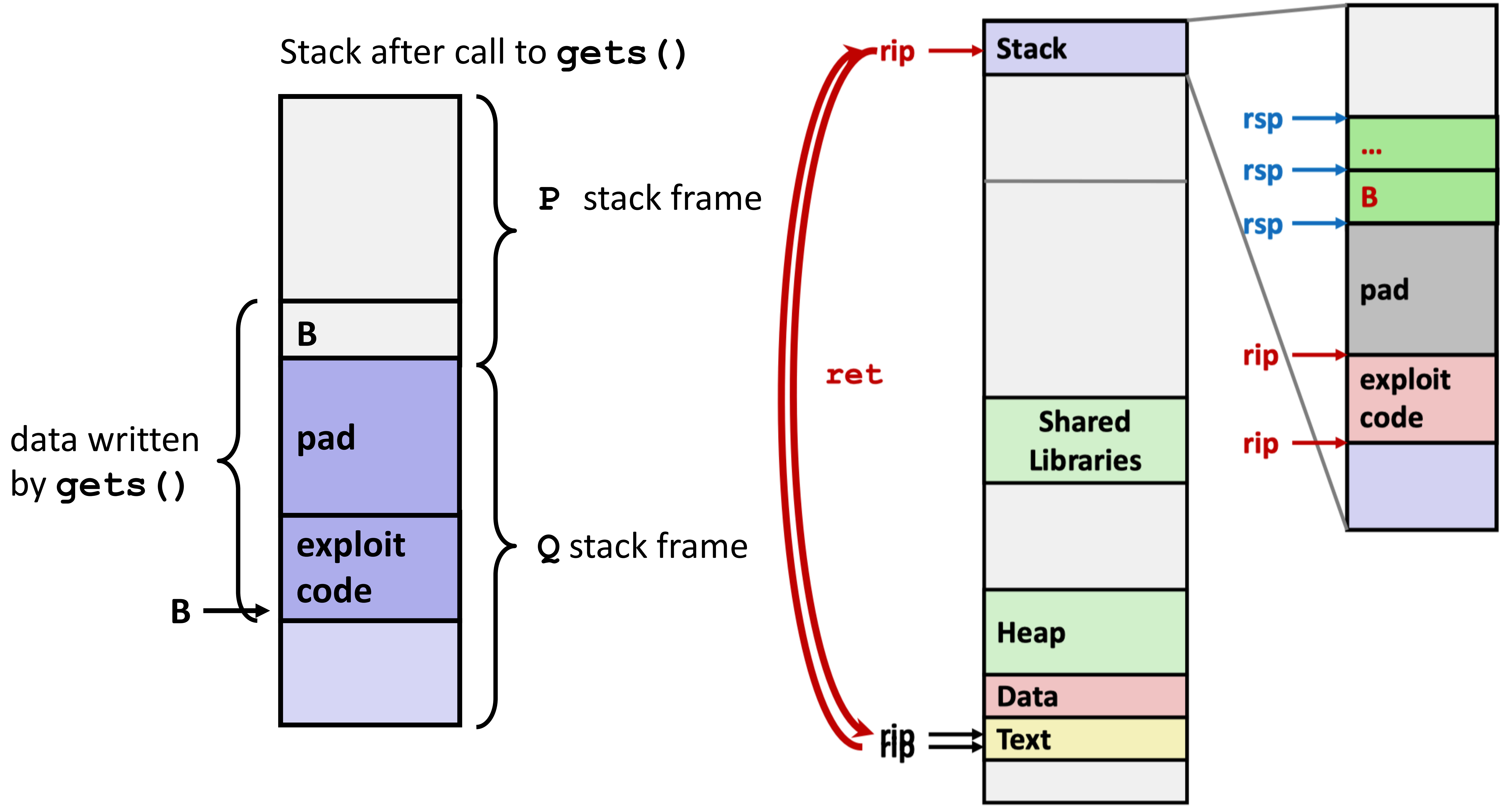
Code Injection Attacks

- Input string contains byte representation of executable code
-

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

return address A

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



Stack Smashing Attacks Protections

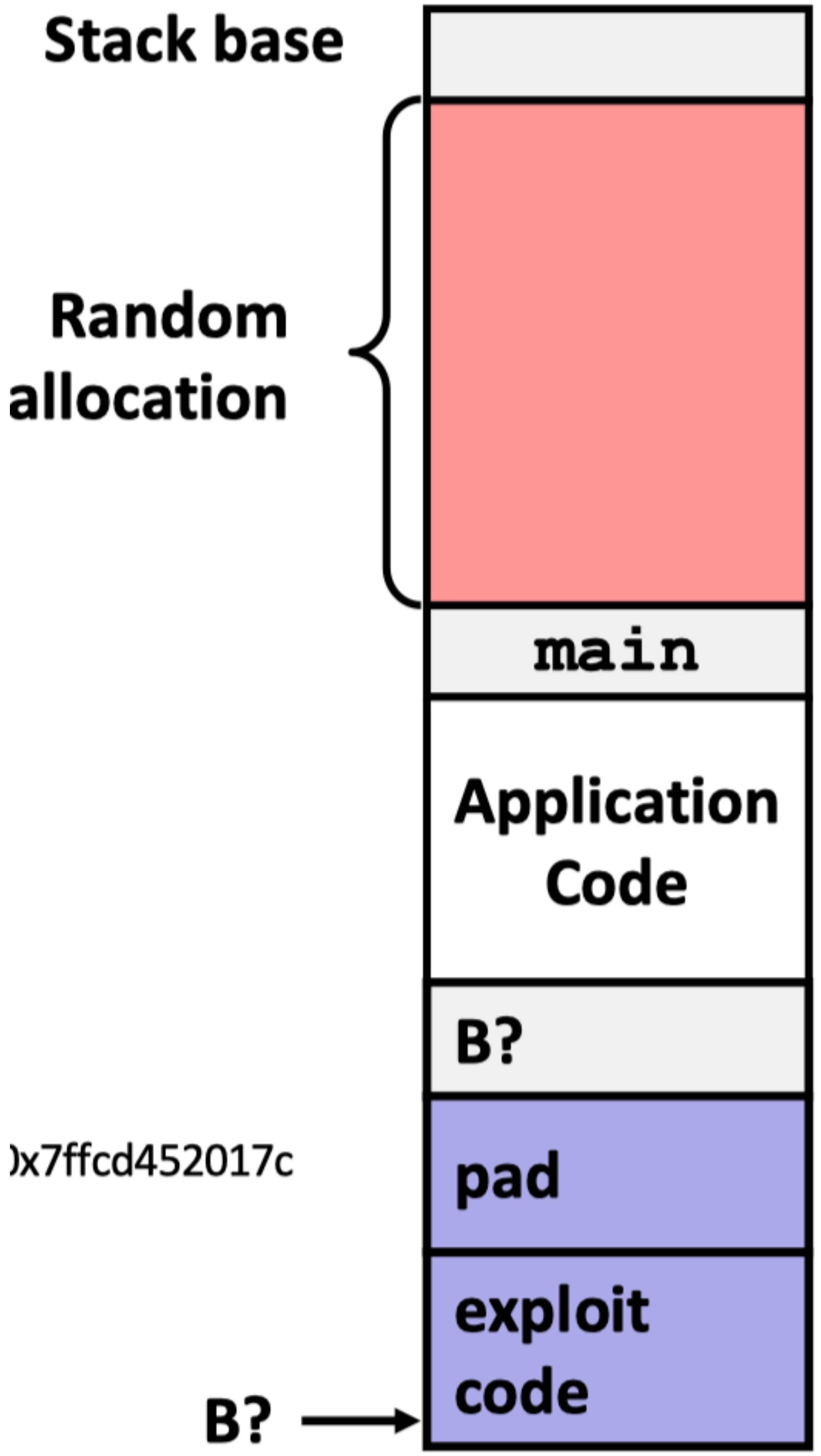
- Avoid overflow vulnerabilities
- Employ system-level protections
- StackGuard protection

Avoid Overflow Vulnerabilities in Code

- 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

System-level protection 1: Stack Randomization

- 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
- In several mainstream operating systems, a technique called **Address space layout randomization (ASLR)** is implemented.



Address space layout randomization (ASLR)

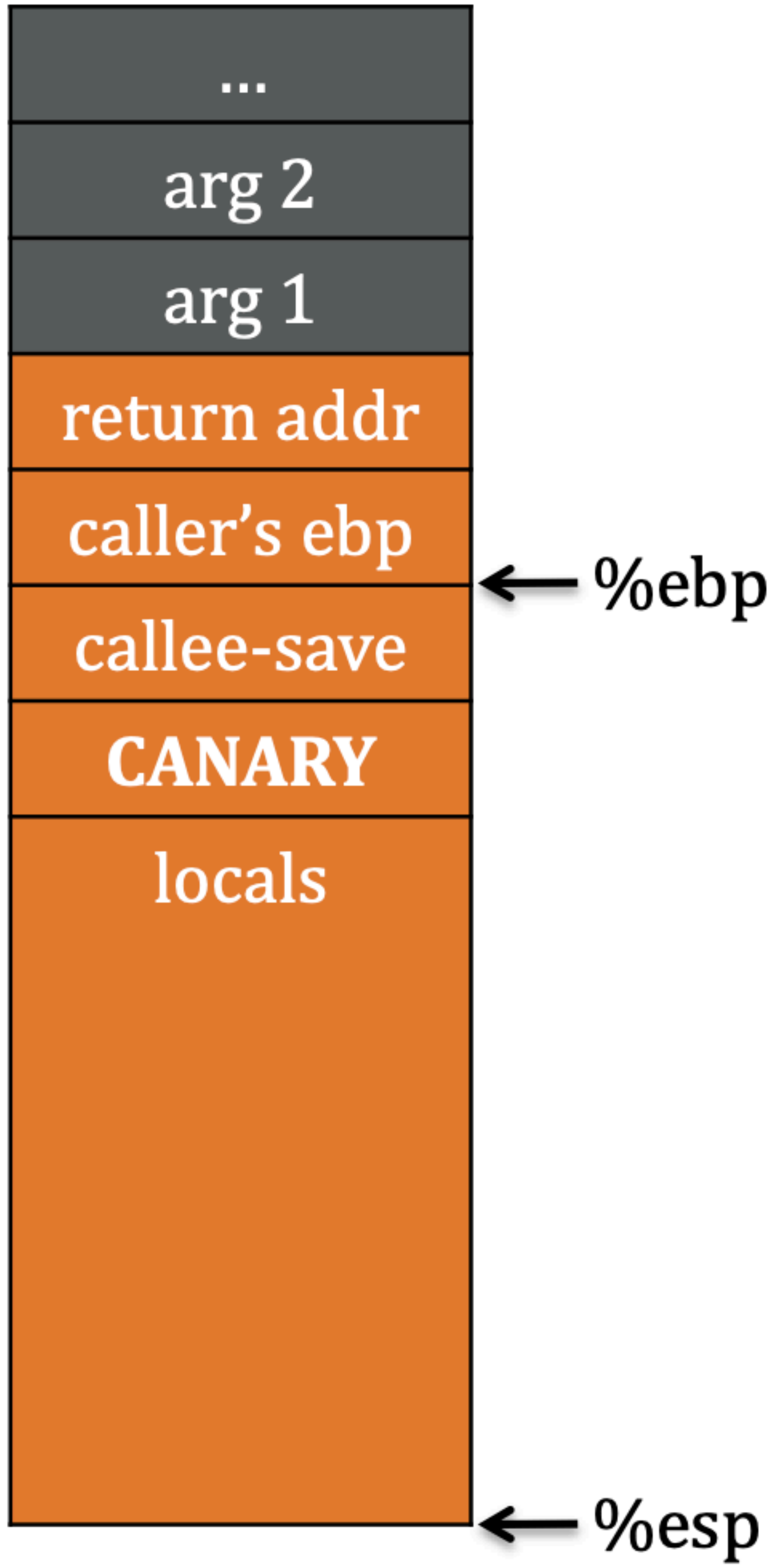
- Traditional exploits need precise addresses
  - buffer overflow: location of shell code
  - return-to-libc: library addresses
- **Problem**: program's memory layout is fixed
- **Solution**: randomize addresses of each region

System-level protection 2: Limiting Executable Code Regions

- Also referred to as **Data Execution Prevention(DEP)**
- In traditional x86, can mark region of memory as either “read-only” or “writeable”. Everything readable is executable.
- X86-64 added explicit “execute” permission.
- Stack marked as non-executable

System-level protection 3: Stack Corruption Detection

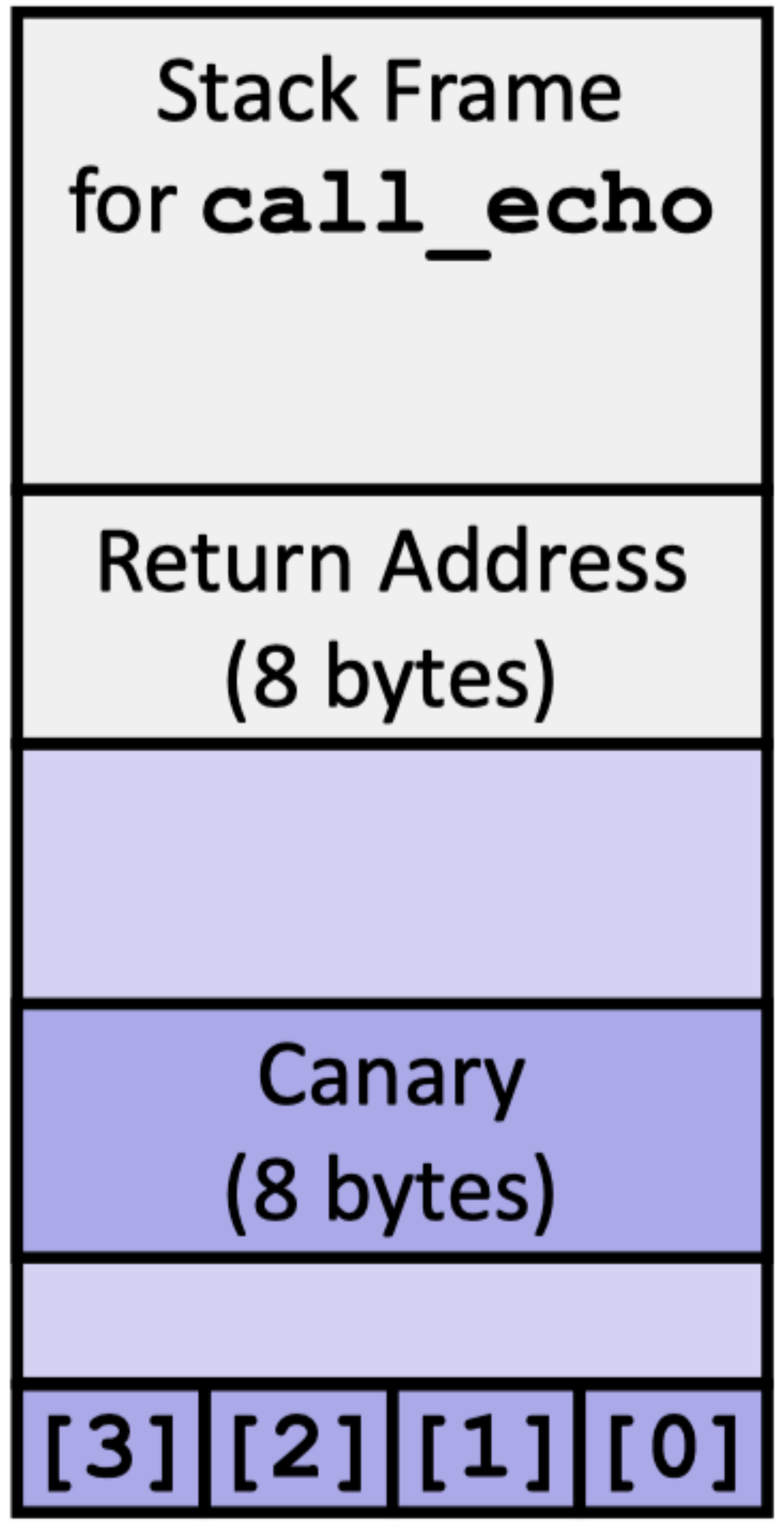
- The idea is to store a special **canary** value 4 in the stack frame between any local buffer and the rest of the stack state.



System-level protection 3: Using canaries

- **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
```



Method to defeat Stack Randomization

- 然而，一个执著的攻击者总是能够用蛮力克服随机化，他可以反复地用不同的地址进行攻击。一种常见的把戏就是在实际的攻击代码前插入很长一段的 nop(读作“no op”，no operatioin 的缩写)指令。执行这种指令除了对程序计数器加一，使之指向下一条指令之外，没有任何的效果。只要攻击者能够猜中这段序列中的某个地址，程序就会经过这个序列，到达攻击代码。这个序列常用的术语是“空操作雪橇(nop sled)”[97]，意思是程序会“滑过”这个序列。如果我们建立一个 256 个字节的 nop sled，那么枚举 $2^{15}=32\,768$ 个起始地址，就能破解 $n=2^{23}$ 的随机化，这对于一个顽固的攻击者来说，是完全可行的。对于 64 位的情况，要尝试枚举 $2^{24}=16\,777\,216$ 就有点儿令人畏惧了。我们可以看到栈随机化和其他一些 ASLR 技术能够增加成功攻击一个系统的难度，因而大大降低了病毒或者蠕虫的传播速度，但是也不能提供完全的安全保障。

Method to defeat Data Execution Prevention

- **Return-Oriented Programming Attacks(ROP)**
- Use machine instruction sequences that are already present in the machine's memory, called "**gadgets**".
- Each gadget typically ends in a return instruction and is located in a subroutine within the existing program or shared library code.

Gadgets Example

```
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

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

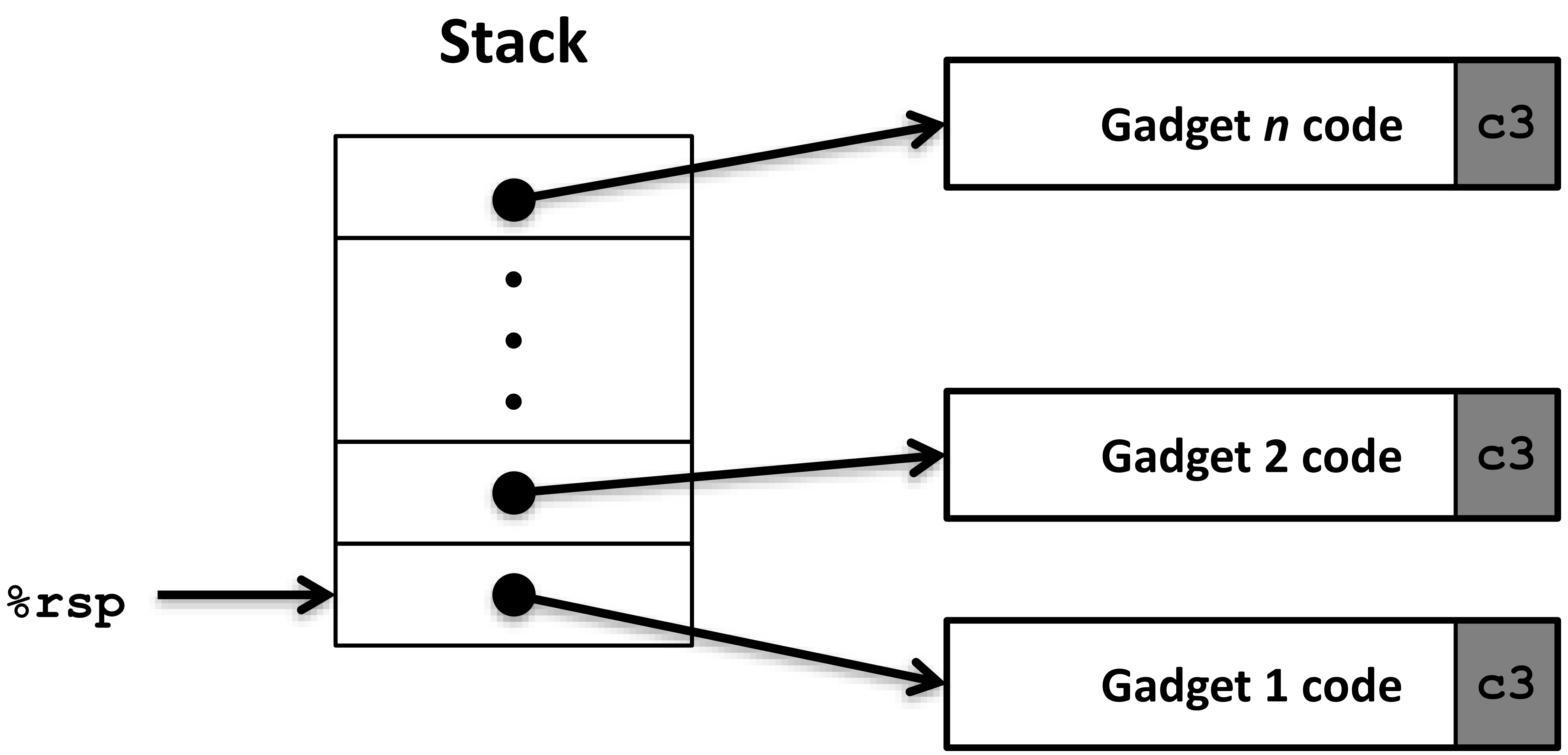
Encodes movq %rax, %rdi

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

rdi ← rax

Gadget address = 0x4004dc

ROP Execution



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

Method to defeat Canaries

- *Canary Weakness: Check does **not** happen until epilogue.*
- Function pointer subterfuge
- Data-pointer modification
- C++ virtual table hijack
- Exception handler hijack
- Guess the canary value
- …

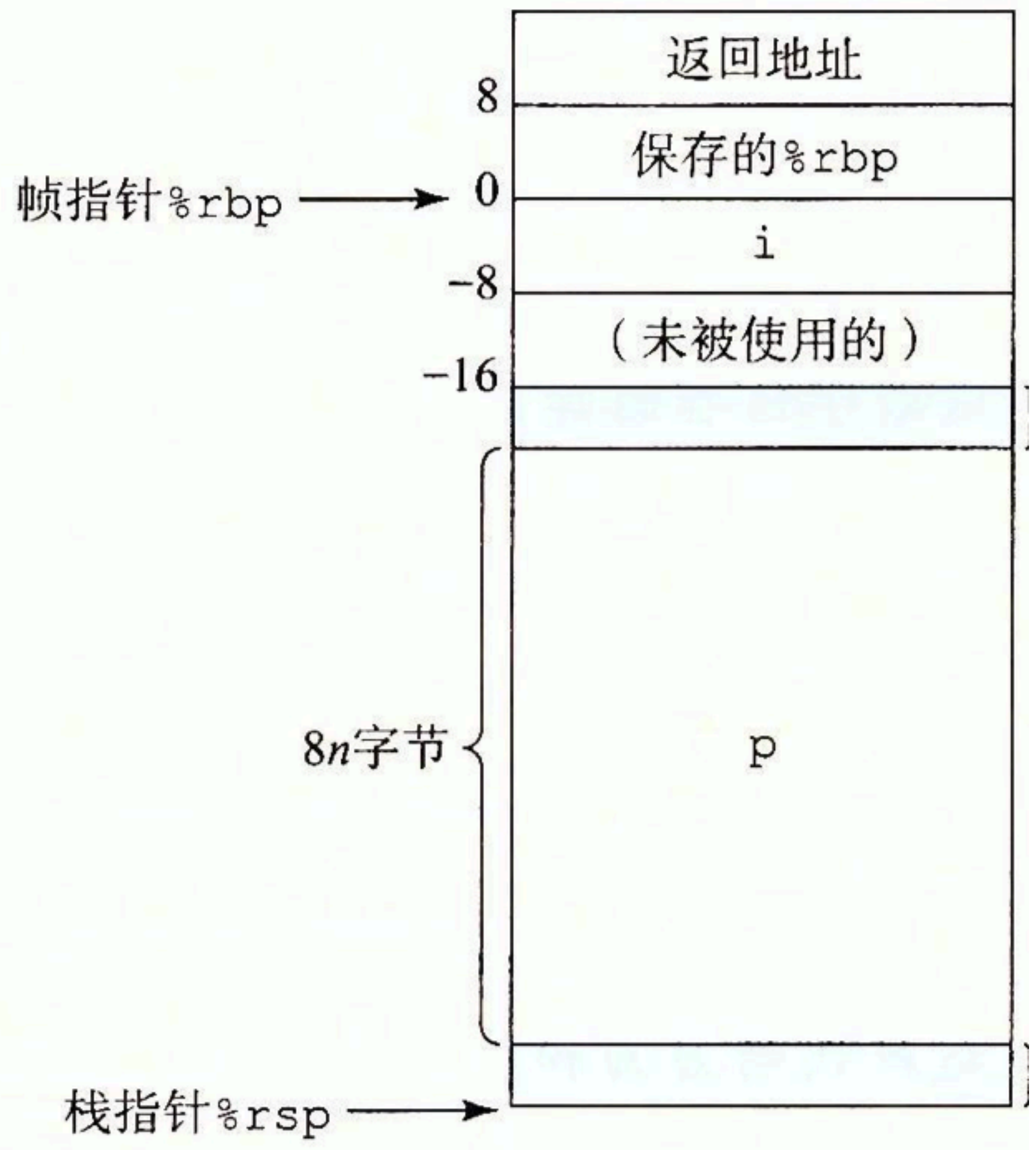
Variable-Size Stack Frames

- When dynamically allocated a local array
- x86-64 code uses **%rbp** as a **frame pointer**(or **base pointer**).

```
vframe:
  pushq   %rbp                      Save old %rbp
  movq    %rsp, %rbp                Set frame pointer
  subq    $16, %rsp                 Allocate space for i (%rsp = s1)
```

```
  leave                             Restore %rbp and %rsp
```

```
movq %rbp, %rsp    Set stack pointer to beginning of frame
popq %rbp          Restore saved %rbp and set stack ptr
                     to end of caller's frame
```



Thanks.