ICS 第 3 次小班课

2021-10-12

Roadmap

- **Review * 2**
  - Data
  - Advanced
  - Recap Chapter 3 ***(by TA)***
- **In-class Exercise**
- **Next class**

Datalab: *ANSI-C and its evolvement*

This is ANSI C.

```
unsigned int foo(unsigned int x)
{
    int y = 5;
    x = x * 2;

    if (x > 5) {
        int z = 4;
        x = x * 3;
        x = x * z;
    }

    return x * y;
}
```

This is *not* ANSI C.

```
unsigned int foo(unsigned int x)
{
    x = x * 2;
    int y = 5;

    if (x > 5) {
        x = x * 3;
        int z = 4;
        x = x * z;
    }

    return x * y;
}
```

Within two braces, all *declarations* must go before any *expressions*.

Recap Chapter 3 (quite rough)

- **基本/复合数据结构的表示**
  - **对齐**
    - 内存中 - 对齐要求 (还包括栈帧/数据块的要求)
    - 寄存器 - 低位格式 Exercise 1 & 2
  - **指针类型的表示, 解引用** Exercise
- **过程的实现**
  - 栈帧构造，参数传递和返回惯例
  - IA32 vs. x86-64 (Just for recreation)
- **GDB使用** Exercise
- **安全性与基本的攻击手段**
  - Attacklab & Exercise

Exercise 1

```
struct A {
        char CC1[6];
        int II1;
        long LL1;
        char CC2[10];
        long LL2;
        int II2;
};
```

(1) sizeof(A) = 56 字节。

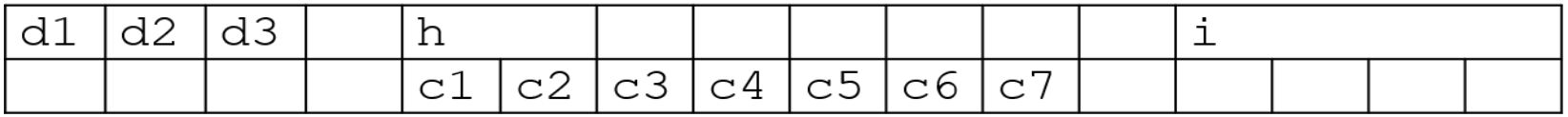
(2) 将A重排后，令结构体尽可能小，那么得到的新的结构体大小为 40 字节。

Exercise 2

```
typedef union {
    char c[7];
    short h;
} union_e;

typedef struct {
    char d[3];
    union_e u;
    int i;
} struct_e;

struct_e s;
```



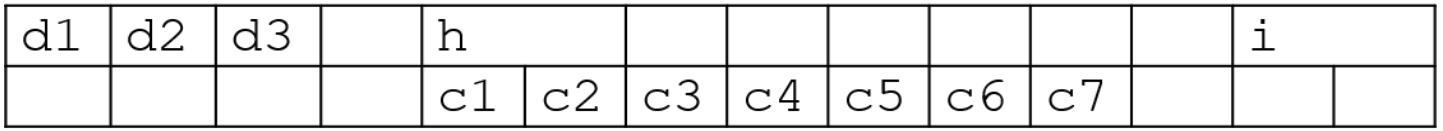
s.u.c的首地址相对于s的首地址的偏移量是__4__字节。
sizeof(union_e) = __8__字节。
s.i的首地址相对于s的首地址的偏移量是__12__字节。
sizeof(struct_e) = __16__字节。

Exercise 2

```
typedef union {
    char c[7];
    short h;
} union_e;

typedef struct {
    char d[3];
    union_e u;
    short i;
} struct_e;

struct_e s;
```

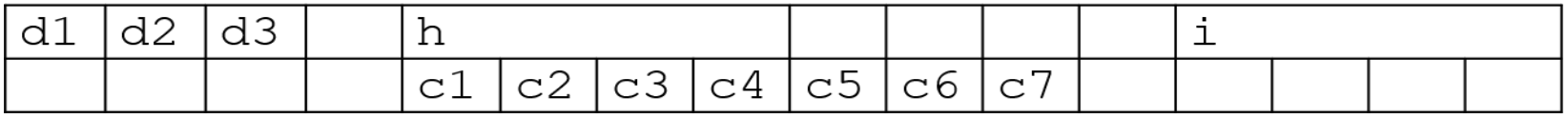


若只将i的类型改成short，那么sizeof(struct_e) = ___14___字节。

```
typedef union {
    char c[7];
    int h;
} union_e;

typedef struct {
    char d[3];
    union_e u;
    int i;
} struct_e;

struct_e s;
```

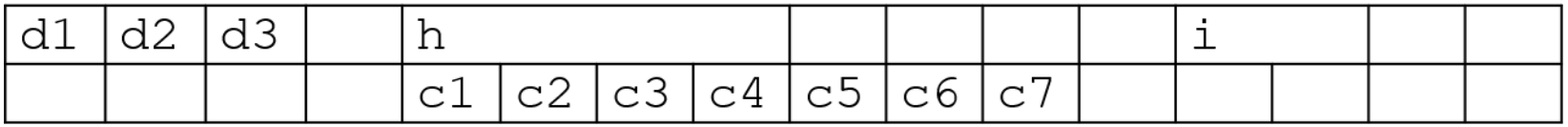


若只将h的类型改成int，那么sizeof(union_e) = ____8____字节。

```
typedef union {
    char c[7];
    int h;
} union_e;

typedef struct {
    char d[3];
    union_e u;
    short i;
} struct_e;

struct_e s;
```

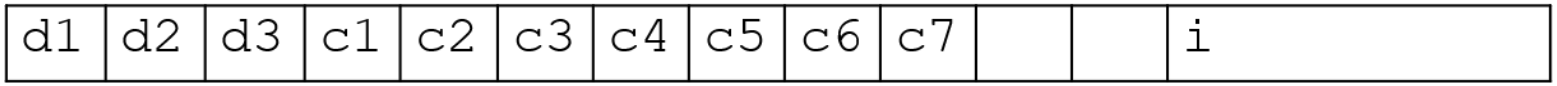


若将i的类型改成short、将h的类型改成int，那么sizeof(union_e) = ___8___字节，sizeof(struct_e) = ___16___字节。

```
typedef union {
    char c[7];
    short h;
} union_e;

typedef struct {
    char d[3];
    union_e u;
    int i;
} struct_e;

struct_e s;
```



若只将short h的定义删除，那么
s.u.c的首地址相对于s的首地址的偏移量是____3____字节。
sizeof(union_e) = ____7____字节。
s.i的首地址相对于s的首地址的偏移量是____12____字节。
sizeof(struct_e) = ____16____字节。

Go back

Additional Exercise 1 :

Pointers & Dereference

	sizeof(A)	What is A?
`int *A[3];`	24	Array of int *
`int *(A[3]);`	24	Same as above
`int (*A)[3];`	8	Pointer to the type int[3]
`int (*A[3]);`	24	Same as the first one
`int (*A[3])();`	24	Array of function pointers (function prototype: int (*func)(void);)

```
(int *) A[3]; // error: missing token before '*'
int *(A[3])(); // error: array of functions!
int *A()[3]; // error: function cannot return an array!
```

Go back

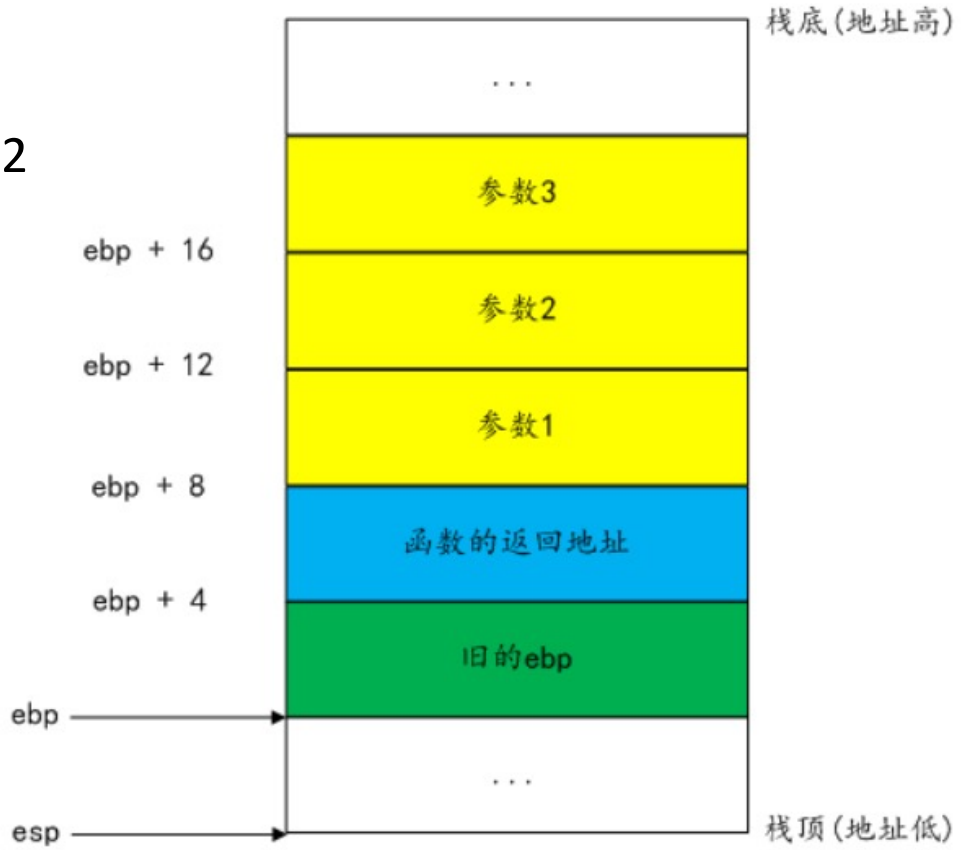
IA32 vs. x86-64 Asm. (Not in exams. Just have fun.)

- **Alignment**
  - IA32 1/2/4字节对象对齐到1/2/4字节。4字节以上对齐到4字节。 Illustration on Ubuntu 32 & 64
- **Procedure Conventions**
  - 参数传递 / 栈指针

  Illustration on Ubuntu 32

```
leave

movq   %rbp, %rsp
popq   %rbp
```



IA32 vs. x86-64 Asm. (Not in exams. Just have fun.)

```
youwin2152@ubuntu:~/Desktop$ cat test.c
#include<stdio.h>
int test(int a,int b
{
        int c=a+b;
        printf("c=%d
        return c;
}
int main()
{
        int t=test(3
        printf("test
        return 0;
}
```

```
0804840b <test>:
 804840b:       55                      push   %ebp
 804840c:       89 e5                   mov    %esp,%ebp
 804840e:       83 ec 18                sub    $0x18,%esp
 8048411:       8b 55 08                mov    0x8(%ebp),%edx
 8048414:       8b 45 0c                mov    0xc(%ebp),%eax
 8048417:       01 d0                   add    %edx,%eax
 8048419:       89 45 f4                mov    %eax,-0xc(%ebp)
 804841c:       83 ec 08                sub    $0x8,%esp
 804841f:       ff 75 f4                pushl  -0xc(%ebp)
 8048422:       68 00 85 04 08          push   $0x8048500
 8048427:       e8 b4 fe ff ff          call   80482e0 <printf@plt>
 804842c:       83 c4 10                add    $0x10,%esp
 804842f:       8b 45 f4                mov    -0xc(%ebp),%eax
 8048432:       c9                      leave
 8048433:       c3                      ret
```

```
8048448:       6a 07                   push   $0x7
804844a:       6a 03                   push   $0x3
804844c:       e8 ba ff ff ff          call   804840b <test>
```

Go back

GDB调试器基本操作指令

- 断点调试
  - Q：如何在函数fun入口处设置断点
  - A：`break fun`
  - Q: `nexti` 和`stepi` 的区别
  - A: 后者仅执行一条指令，而前者以函数调用为单位
- 检查数据
  - Q：检查函数fun的前10个字节的指令？
  - A：`x/10b fun`
  - Q：输出位于地址`%rsp+8`处的长整数
  - A：`print *(long *) (%rsp+8)`

Exercise 3

17. 写出使用gcc编译源代码lab.c、生成可执行文件lab、采用二级编译优化的命令。

【答】gcc lab.c -o lab -O2 ~~gcc -O2 -o lab.c~~ -> All were gone!

18. 写出使用gcc编译源代码foo.c、生成汇编语言文件foo.s的命令

【答】gcc foo.c -S

19. 将可执行文件bar逆向工程为汇编代码的工具是(　　)
A. gcc　　B. gdb　　C. objdump　　D. hexedit　　E. gedit

【答】C

20. gdb中，单步指令执行的命令是
A. r　　B. b　　C. p　　D. finish　　E. si
F. disas

【答】E

Go back

```
typedef struct{
    long a;
    long b;
} pair_type;
long func(pair_type *p) {
    if (p -> a < p -> b) {
        long temp = p -> a;
        p -> a = p -> b;
        p -> b = temp;
    }
    if ((7)                ) {
        return p -> a;
    }
    pair_type np;
    np.a = (8)                ;
    np.b = (9)                ;
    return func(&np);
}
int main(int argc,
char* argv[]) {
    pair_type np;
    np.a = (10)                ;
    np.b = (11)                ;
    printf("%ld", func(&np));
    return 0;
}
```

```
0000000004005fc <main>:
  4005fc: sub    $0x28,%rsp
  400600: mov    %fs:0x28,%rax
  400609: mov    %rax,0x18(%rsp)
  40060e: xor    %eax,%eax
  400610: movq   0x69,(%rsp)
  400618: movq   0xfc,0x8(%rsp)
  400621: mov    %rsp,%rdi
  400624: callq  400596 <func>
  400629: mov    %rax,%rsi
  40062c: mov    $0x4006e4,%edi
  400631: mov    $0x0,%eax
  400636: callq 400470 <printf@plt>
  40063b: mov    0x18(%rsp),%rdx
  400640: xor    (4)               ,%rdx
  400649: (5)    400650 <main+0x54>
  40064b: callq  400460
<__stack_chk_fail@plt>
  400650: mov    $0x0,%eax
  400655: add    (6)               ,%rsp
  400659: retq
```

```
0000000000400596 <func>:
  400596: sub    $0x28,%rsp
  40059a: mov    %fs:0x28,%rax
  4005a3: mov    %rax,0x18(%rsp)
  4005a8: xor    %eax,%eax
  4005aa: mov    (%rdi),%rax
  4005ad: mov    0x8(%rdi),%rdx
  4005b1: cmp    %rdx,%rax
  4005b4: jge    (1)
  4005b6: mov    %rdx,(%rdi)
  4005b9: mov    %rax,0x8(%rdi)
  4005bd: mov    0x8(%rdi),%rax
  4005c1: test   %rax,%rax
  4005c4: jne    4005cb <func+0x35>
  4005c6: mov    (%rdi),%rax
  4005c9: jmp    (2)
  4005cb: mov    (%rdi),%rdx
  4005ce: sub    %rax,%rdx
  4005d1: mov    %rdx,(%rsp)
  4005d5: mov    %rax,0x8(%rsp)
  4005da: mov    (3)              ,%rdi
  4005dd: callq  400596 <func>
```

```
  4005e2: mov 0x18(%rsp)
,%rcx
  4005e7: xor
(4)      ,%rcx
  4005f0: (5)        4005f7
<func+0x61>
  4005f2: callq      400460
<__stack_chk_fail@plt>
  4005f7: add
(6)      ,%rsp
  4005fb: retq
```

Go back

Any questions?

Fiddle with those handy tools!
Like `gdb` and `objdump`…

变长栈帧

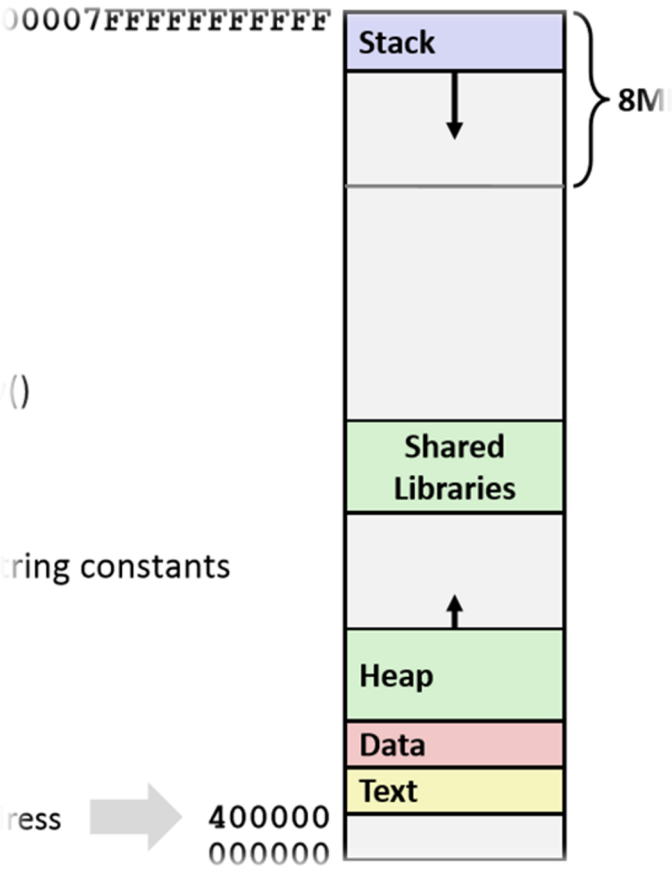
- %rbp作为帧指针的作用
  - 在整个函数的执行过程中，%rbp始终指向函数栈的顶端（在返回地址和保存被调用者保存寄存器的值的下方）
  - 利用固定长度的局部变量相对于%rbp的偏移量来引用它们
  - leave指令释放整个栈帧
- 完成练习题3.49

浮点代码

- 基础知识
  - %xmm0存放浮点返回值
  - %xmm0~%xmm7在过程中传递浮点参数
  - 所有的XMM寄存器都是调用者保存的
  - 浮点运算操作指令中第一个源操作数可以是一个XMM寄存器或者内存地址，第二个源操作数和目的操作数必须是XMM寄存器
- 浮点常数
  - 浮点操作不能以立即数作为操作数，浮点常数必须存储在内存中
  - 练习题3.55

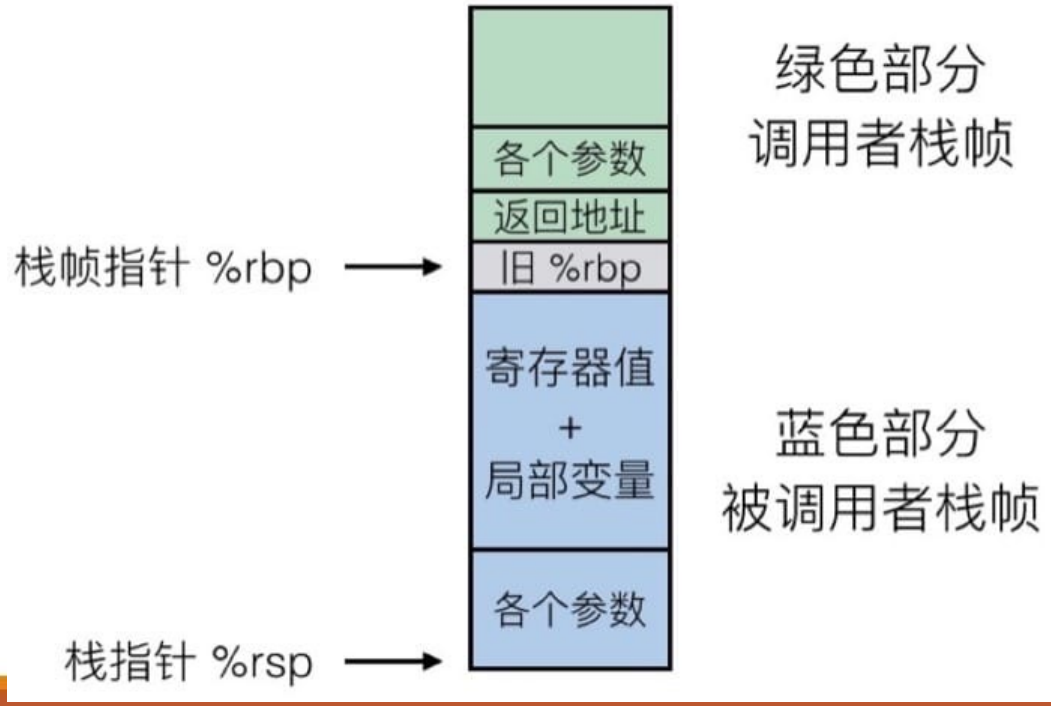
Memory layout

（1）代码区：存放函数二进制代码，只读
（2）数据区：系统运行时申请内存并初始化，系统退出时系统释放。全局变量、静态变量、常量。
①bss区：未初始化的
②文字常量区：在程序中使用的常量存储在此区域。程序结束后，由系统释放。
（3）堆区：通过malloc等函数或new等操作符动态申请得到，需手动申请和释放。（内存泄漏）
（4）栈区：函数模块内申请，函数结束时由系统自动释放。存放局部变量、函数参数 。大小为8MB



递归时栈

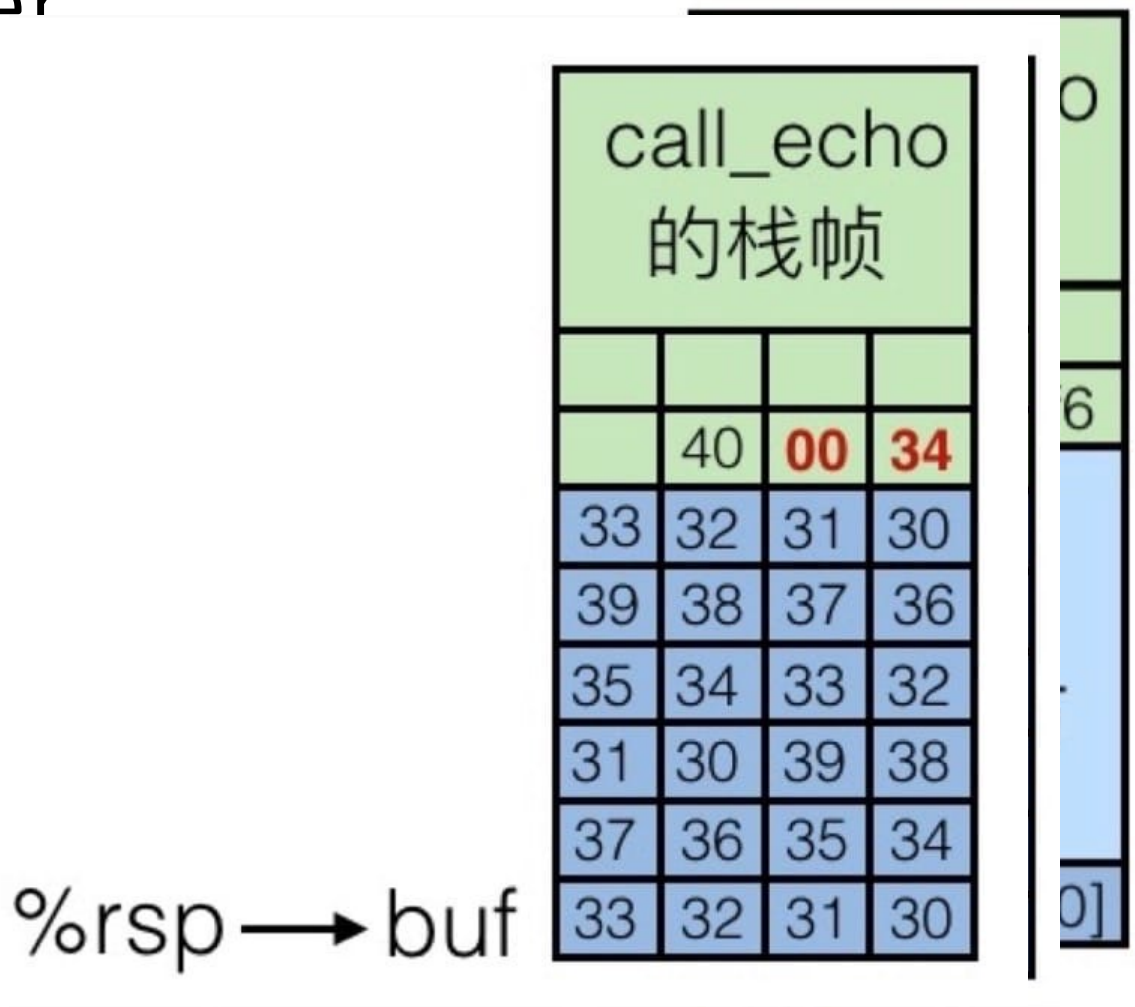
- 函数递归调用时回用%rax寄存器保存返回值，这是递归能实现的基础。
- 对于每一次调用，栈将会开辟一个空间（Frame）来保存必要信息
- 返回信息
- 本地变量
- 临时空间



Buffer

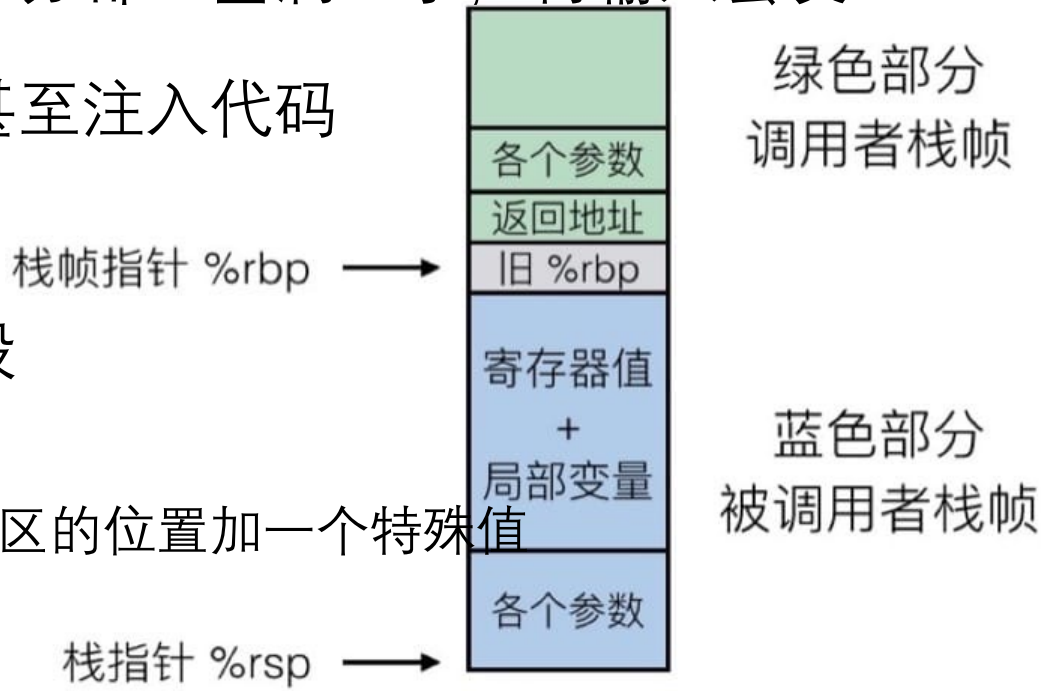
- 当输
  入会
- 可能
- 由此

，再输



Buffer overflow

- 当输入内容过多把蓝色部分都“塞满”了，再输入会发生什么？
- 可能会改变返回地址，甚至注入代码
- 由此可以进行攻击
- Attacklab
- 相应地，也就有防护手段
  - 使用更安全的输入函数
  - 随机栈
  - “金丝雀”，在超出缓冲区的位置加一个特殊值
  - 禁止栈内代码执行
  - ROP attack



Canary Protection

示例：

```
00000000000014e8 <phase_4>:
   14e8:    48 83 ec 18             sub    $0x18,%rsp
   14ec:    64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
   14f3:    00 00
   14f5:    48 89 44 24 08          mov    %rax,0x8(%rsp)
   14fa:    31 c0                   xor    %eax,%eax
……
   1549:    48 8b 44 24 08          mov    0x8(%rsp),%rax
   154e:    64 48 33 04 25 28 00    xor    %fs:0x28,%rax
   1555:    00 00
   1557:    75 05                   jne    155e <phase_4+0x76>
   1559:    48 83 c4 18             add    $0x18,%rsp
   155d:    c3                      retq
   155e:    e8 6d f9 ff ff          callq  ed0 <__stack_chk_fail@plt>
```

Canary:
提供了一种保护机制，可以在返回前检测程序是否被破坏，其值具有不可预测性；

仍然可以被利用，由于canary值来自%fs:0x28,只要找到这个值就可以破解这种机制，可以通过格式化字符串漏洞来办到

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
```

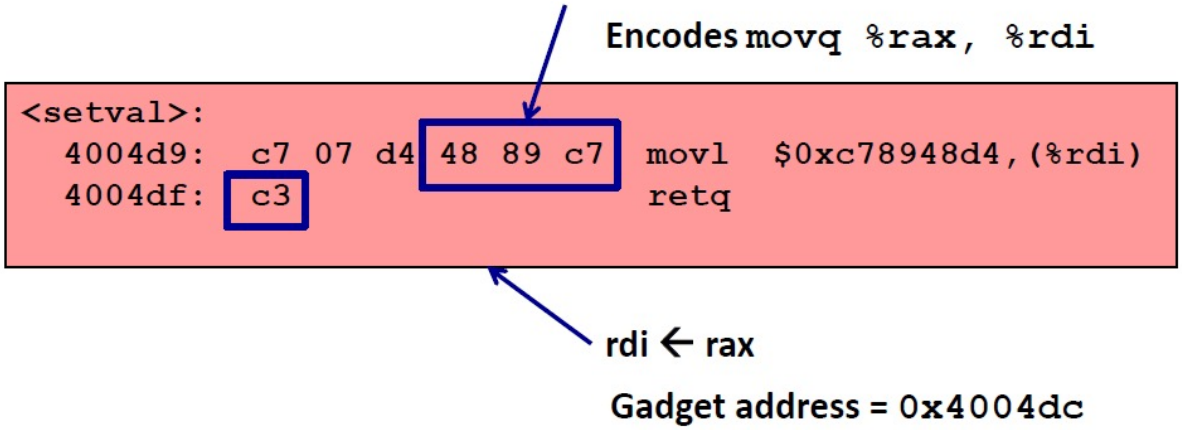
rax ← rdi + rdx

Gadget address = 0x4004d4

- Use tail end of existing functions

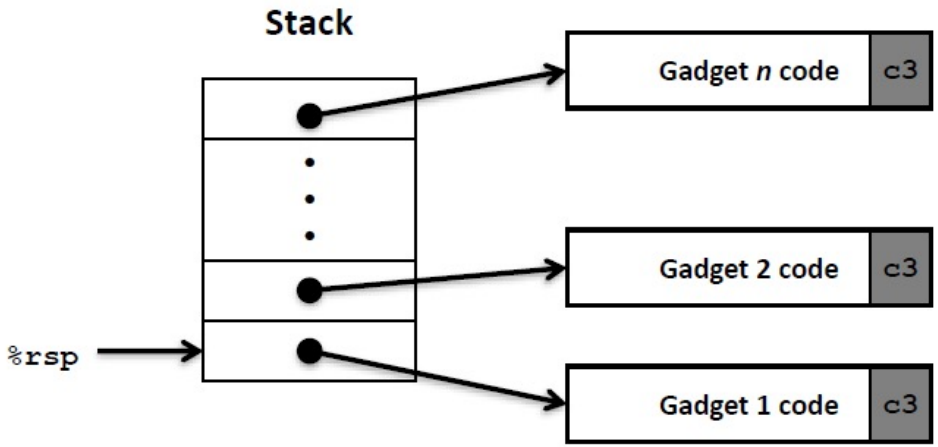
Gadget Example #2

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



- Repurpose byte codes

ROP Execution

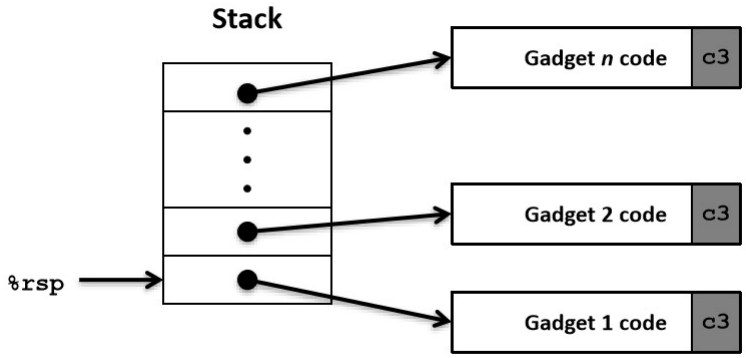


- **Trigger with `ret` instruction**
  - Will start executing Gadget 1
- **Final `ret` in each gadget will start next one**

Attack: ROP

示例：

攻击目标为实现**system(“echo success”)**
这个函数调用



素材：

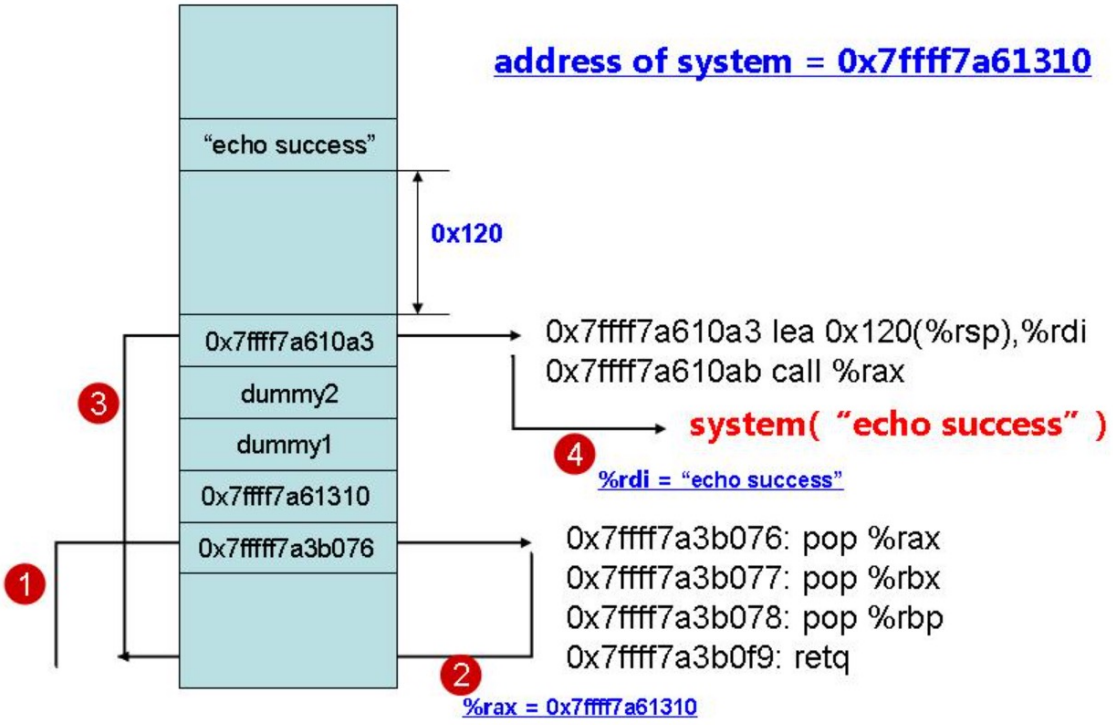
```
1  0x7ffff7a610a3      lea 0x120(%rsp),%rdi
2  0x7ffff7a610ab      call    %rax
```

```
1  0x7ffff7a3b076:    pop %rax
2  0x7ffff7a3b077:    pop %rbx
3  0x7ffff7a3b078:    pop %rbp
4  0x7ffff7a3b0f9:    retq
```

通过gdb查看system函数的地址：

```
1  (gdb) p system
2  $4 = {<text variable, no debug info>} 0x7ffff7a61310 <system>
```

Attack: ROP


address of system = 0x7ffff7a61310
"echo success"
0x120
0x7ffff7a610a3
dummy2
dummy1
0x7ffff7a61310
0x7ffff7a3b076
0x7ffff7a610a3 lea 0x120(%rsp),%rdi
0x7ffff7a610ab call %rax
system("echo success")
%rdi = "echo success"
0x7ffff7a3b076: pop %rax
0x7ffff7a3b077: pop %rbx
0x7ffff7a3b078: pop %rbp
0x7ffff7a3b0f9: retq
%rax = 0x7ffff7a61310
1
2
3
4