Machine Prog: Advanced

王振宇

- **联合 : Unions**
  - 联合数据类型
  - 大端法与小端法
  - 与枚举Enums的配合

- **内存引用越界和缓冲区溢出**
  - 内存布局与内存引用越界
  - 缓冲区溢出攻击
  - 对抗缓冲区溢出攻击

- **支持变长栈帧**

联合数据类型

联合：在同一个内存空间储存不同的数据类型，一次只能存储一种数据类型。规避C语言的类型系统，以多种类型引用同一个对象。

应用情况：当数据使用两种或多种格式，但不会同时使用时。

作用：用来访问不同数据类型的位模式；减少分配的内存空间。

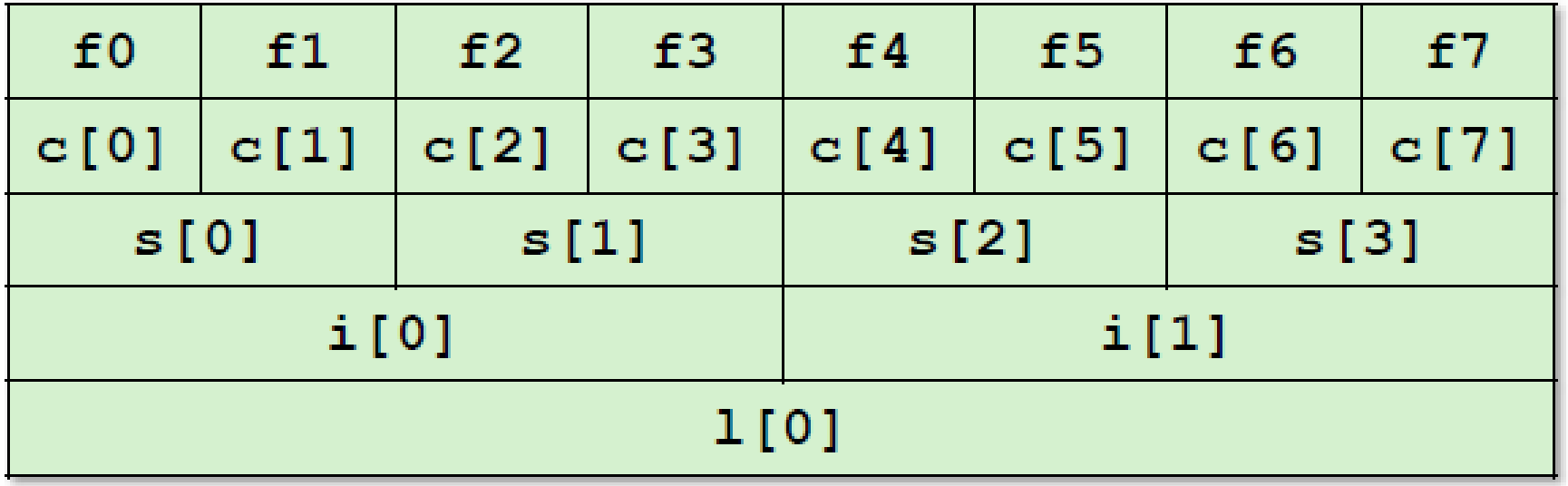
访问不同数据类型的位模式

```
unsigned long double2bits(double d) {
    union {
        double d;
        unsigned long u;
    } temp;
    temp.d = d;
    return temp.u;
};
```

将double的位表示通过联合转换为long

但凡涉及到字节与位的访问问题，都需要注意字节顺序，大小端机器对字节的读取方式不同。

访问不同数据类型的位模式



- x86-64

i[0] = [0xf3f2f1f0]; l[0] = [0xf7f6f5f4f3f2f1f0];

- Sun

i[0] = [0xf0f1f2f3]; l[0] = [0xf0f1f2f3];

减少分配的内存空间

```
union node_u {
    struct {
        union node_u *left;
        union node_u *right;
    } internal;
    double data[2];
};
```

二叉树节点储存两个子节点指针
或两个double类型的值

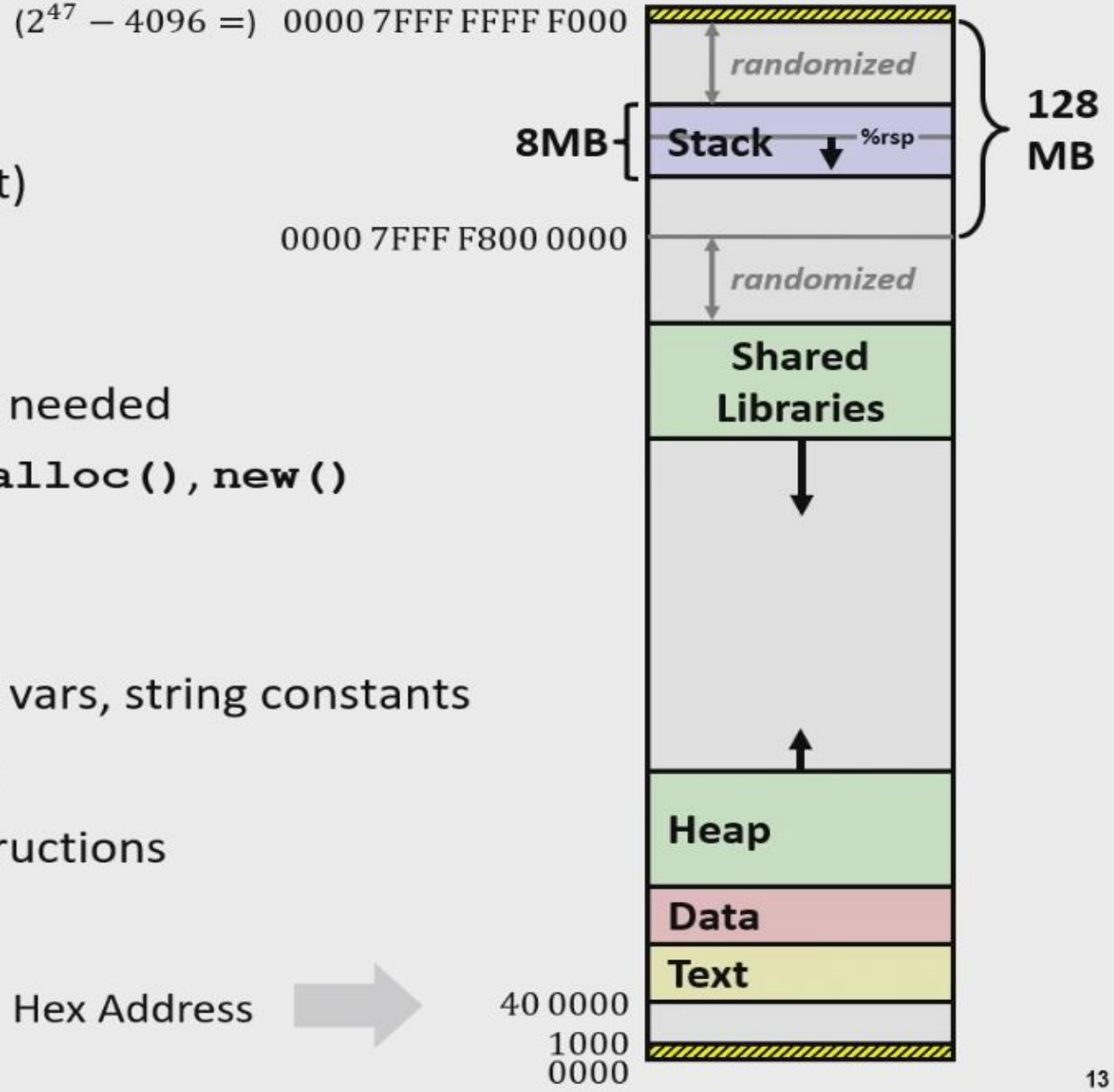
```
typedef enum { N_LEAF, N_INTERNAL } nodetype_t;

struct node_t {
    nodetype_t type;
    union {
        struct {
            struct node_t *left;
            struct node_t *right;
        } internal;
        double data[2];
    } info;
};
```

用一个枚举变量确定该二叉树节点是内部节点还是叶子节点，确定联合中存放的是指针还是double类型

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

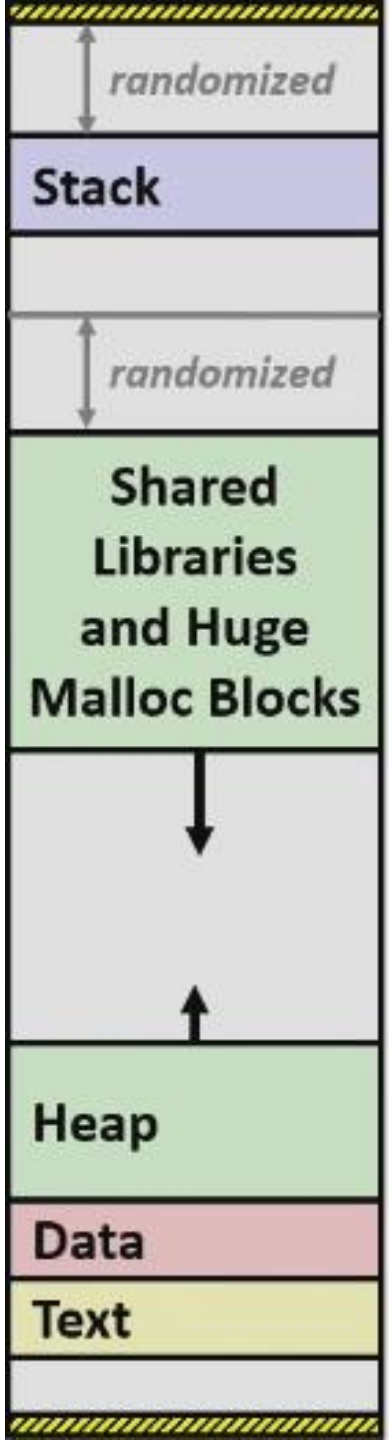


```
char big_array[1L<<24]; /* 16 MB */
char huge_array[1L<<31]; /* 2 GB */

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 ... */
}
```



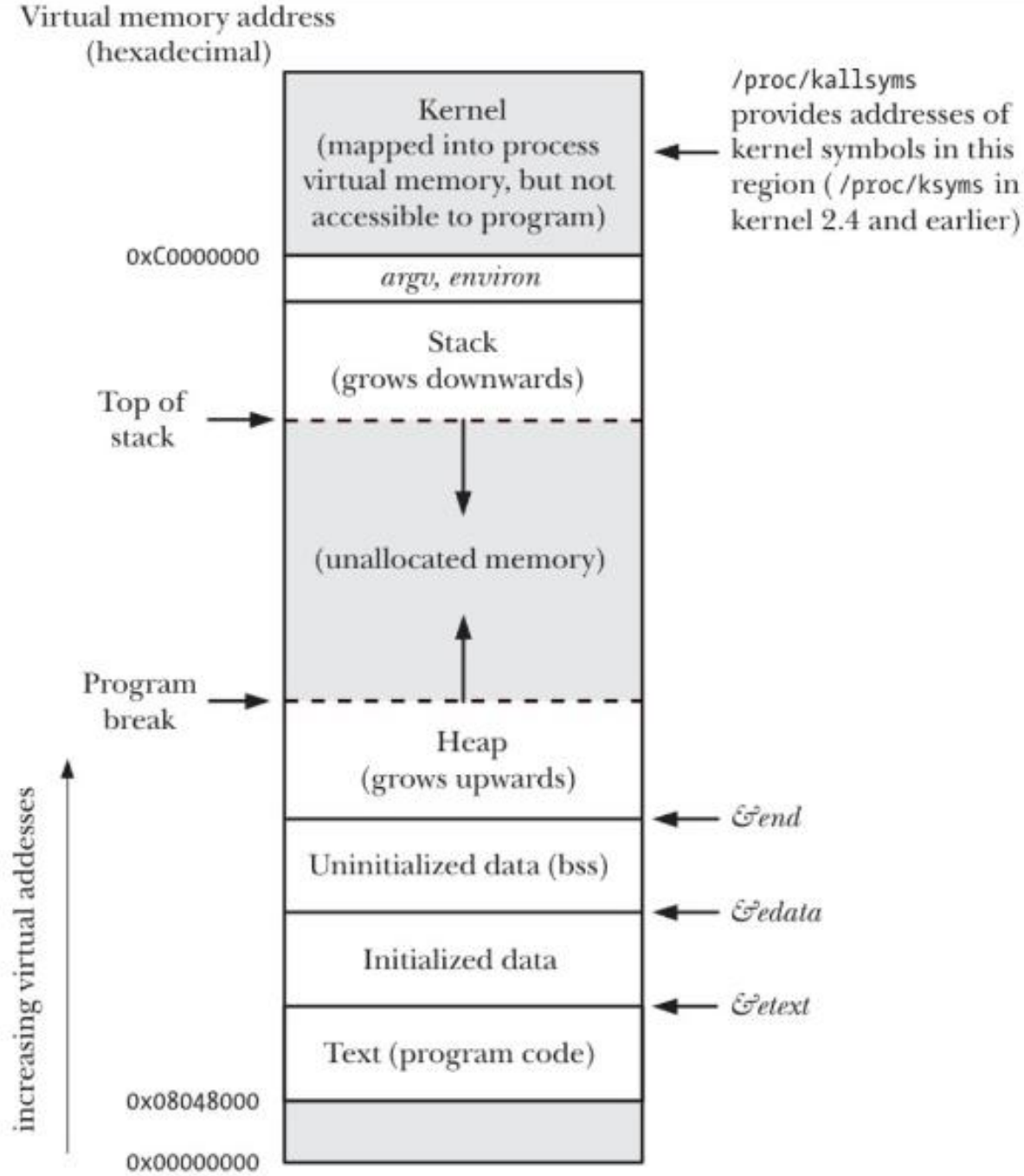


Figure 6-1: Typical memory layout of a process on Linux/x86-32

内存布局与内存引用越界

C对数组引用不进行任何边界检查，而且局部变量和状态信息都存放在栈中。对越界的数组元素的写操作会破坏储存在栈中的状态信息，当程序使用被破坏的状态信息，试图重新加载寄存器或执行ret指令时，就会出现很严重的错误。

在栈中分配某个字符数组来保存字符串，但字符串的长度超出为数组分配的空间，将引发缓冲区溢出。

缓冲区溢出

```
/* Implementation of library function gets() */
char *gets(char *s)
{
    int c;
    char *dest = s;
    while ((c = getchar()) != '\n' && c != EOF)
        *dest++ = c;
    if (c == EOF && dest == s)
        /* No characters read */
        return NULL;
    *dest++ = '\0'; /* Terminate string */
    return s;
}

/* Read input line and write it back */
void echo()
{
    char buf[8];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
    void echo()
1   echo:
2     subq    $24, %rsp          Allocate 24 bytes on stack
3     movq    %rsp, %rdi         Compute buf as %rsp
4     call    gets               Call gets
5     movq    %rsp, %rdi         Compute buf as %rsp
6     call    puts               Call puts
7     addq    $24, %rsp          Deallocate stack space
8     ret                        Return
```

输入的字符数量	附加的被破坏的状态
0～7	无
9～23	未被使用的栈空间
24～31	返回地址
32+	caller 中保存的状态

缓冲区溢出攻击

通过缓冲区溢出，可以让程序执行本不应被执行的函数，甚至是通过缓冲区溢出插入的可执行代码。

输入给程序一个字符串，该字符串包含可执行的字节编码（一般插入到栈帧），同时用另外一些字节用指向攻击代码的指针覆盖返回指针，从而使得执行ret指令的效果为跳转到攻击代码。

对抗缓冲区溢出攻击

- 避免缓冲区溢出，使用更安全的方法，如：fgets, strncpy 等等。
- 栈随机化
- 栈破坏检测
- 限制可执行代码区域

对抗缓冲区溢出攻击

- **栈随机化**

  程序开始时，在栈上分配一段0~n字节之间的随机大小的空间，程序不使用这段空间，但是它会导致程序每次执行时后续的栈位置发生了变化，使得难产生指向攻击代码的指针。

- **地址空间布局随机化 ASLR**

  每次运行时程序代码、库代码、栈、全局变量和堆数据都加载到内存的不同位置。

- **空操作雪橇**

  在实际的攻击代码前插入很长一段的nop指令，枚举起始地址从而破解随机化。（nop指令增长了攻击代码的长度，更容易被枚举到）

对抗缓冲区溢出攻击

- **栈破坏检测**

  程序在缓冲区与栈状态间插入金丝雀值，在恢复寄存器状态和从 函数返回之前，程序检查这个金丝雀值是否被该函数的某个操作或者该函数调用的某个函数的某个操作改变。

```
3     movq    %fs:40, %rax        Retrieve canary
4     movq    %rax, 8(%rsp)       Store on stack
5     xorl    %eax, %eax          Zero out register
6     movq    %rsp, %rdi          Compute buf as %rsp
7     call    gets                Call gets
8     movq    %rsp, %rdi          Compute buf as %rsp
9     call    puts                Call puts
10    movq    8(%rsp), %rax       Retrieve canary
11    xorq    %fs:40, %rax        Compare to stored value
12    je      .L9                 If =, goto ok
13    call    __stack_chk_fail    Stack corrupted!
14  .L9:                        ok:
15    addq    $24, %rsp           Deallocate stack space
16    ret
```

通过段寻址 %fs:40 读入储存在特殊的“只读”段中，返回前通过xorq指令比较栈上的金丝雀值是否被修改。

基本绕过方式：由于canary保护仅仅是检查canary是否被改写，而不会检查其他栈内容，因此如果攻击者能够泄露出canary的值（一般利用格式化字符串漏洞），便可以在构造攻击负载时填充正确的canary，从而绕过canary检查，达到实施攻击的目的。

对抗缓冲区溢出攻击

- **限制可执行代码区域**

  三种访问形式：读（从内存读数据）、写（存储数据到内存）和执行（将内存的内容看作机器级代码。 限制哪些内存区域能够存放可执行代码，消除攻击者向系统中插入可执行代码的能力。

- **返回导向编程**

  可以利用修改已有的代码，来绕过系统和编译器的保护机制，攻击者控制堆栈调用以劫持程序控制流并执行针对性的机器语言指令序列（称为Gadgets）。每一段 gadget 通常结束于 return 指令，并位于共享库代码中的子程序。系列调用这些代码，攻击者可以在拥有更简单攻击防范的程序内执行任意操作。

支持变长栈帧

%rbp作为帧指针、基指针

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

leave将释放整个栈帧，恢复%rbp的值。等价于

movq %rbp，%rsp

popq %rbp