Datawhale

Datawhale 开源社区

Datawhale open source community

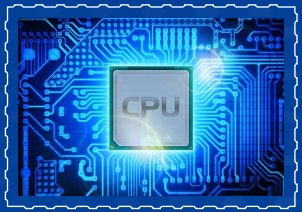
深入理解计算机系统（8）

Computer Systems A Programmer's Perspective CSAPP

李岳昆、易远哲

realyurk@gmail.com、yuanzhe.yi@outlook.com

2021 年 12 月 18 日



第 I 部分

程序的机器级表示-III

异质的数据结构

缓冲区溢出

变长数组

Datawhale

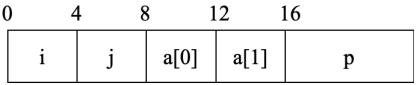
首先我们来看结构体的声明。

struct

```
struct rec{
    int i;
    int j;
    int a[2];
    int *p;
}
```

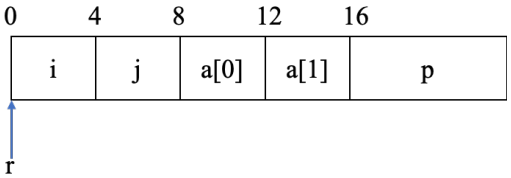
这个结构体包含四个字段：两个 int 类型的变量，个 int 类型的数组和一个 int 类型的指针。

我们可以画出各个字段相对于结构体起始地址处的字节偏移。



从这个图上可以看出数组 a 的元素是嵌入到结构体中的。接下来，我们看一下如何访问结构体中的字段。

例如，我们声明一个结构体类型指针变量 r，它指向结构体的起始地址。



变长数组

Datawhale

假设 r 存放在寄存器 rdi 中，可以使用下图的汇编指令将字段 i 的值复制到字段 j 中。

```
r in %rdi
  movl    (%rdi)，%eax   Get r->i
  movl    %eax，4(%rdi)  Store in r->j
```

- 首先读取字段 i 的值，由于字段 i 相对于结构体起始地址的偏移量为 0，所以字段 i 的地址就是 r 的值，而字段 j 的偏移量为 4，因此需要将 r 加上偏移量 4。
- 其中结构体指针 r 存放在寄存器 rdi 中，数组元素的索引值 i 存放在寄存器 rsi 中，最后地址的计算结果，存放在寄存器 rax 中。

综上所述，无论是单个变量还是数组元素，都是通过起始地址加偏移量的方式来访问。

数据对齐

Datawhale

对于图中的结构体，它包含两个 int 类型的变量和一个 char 类型的变量。

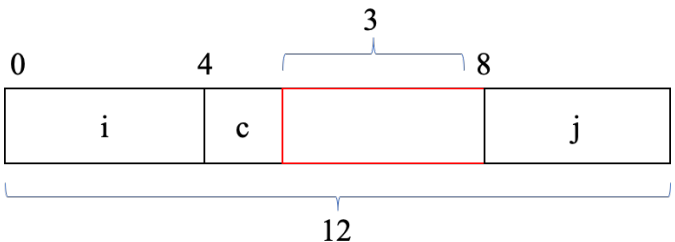
```
struct S1{
    int  i;
    char c;
    int  j;
}
```

根据前面的知识，我们会直观的认为该结构体占用 9 个字节的存储空间，但是当使用 sizeof 函数对该结构体的大小进行求值时，得到的结果却是 12 个字节。原因是为了提高内存系统的性能，系统对于数据存储的合法地址做出了一些限制。

数据对齐

Datawhale

例如变量 j 是 int 类型，占 4 个字节，它的起始地址必须是 4 的倍数，因此，编译器会在变量 c 和变量 j 之间插入一个 3 字节的间隙，这样变量 j 相对于起始地址的偏移量就为 8，整个结构体的大小就变成了 12 个字节。



数据对齐

Datawhale

对于不同的数据类型，地址对齐的原则是任何 K 字节的基本对象的地址必须是 K 的倍数。也就是说对于 short 类型，起始地址必须是 2 的倍数；对于占 8 个字节的数据类型，起始地址必须是 8 的倍数。

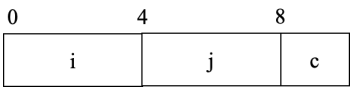
K	Types
1	char
2	short
4	int, float
8	long, double, char*

基于上表的规则，编译器可能需要在字段的地址空间分配时插入间隙，以此保证每个结构体的元素都满足对齐的要求。

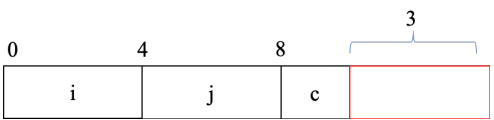
除此之外，结构体的末尾可能需要填充间隙，还是刚才的这个结构体，可以通过调整字段 j 和字段 c 的排列顺序，使得所有的字段都满足了数据对齐的要求。

```
struct S1{
    int  i;
    char c;
    int  j;
};
```

```
struct S2{
    int  i;
    int  j;
    char c;
};
```



但是当我们声明一个结构体数组时，分配 9 个字节的存储空间，是无法满足所有数组元素的对齐要求，因此，编译器会在结构体的末端增加 3 个字节的填充，这样一来，所有的对齐限制都满足了。



数据对齐

Datawhale

根据上述对齐原则，我们看一个复杂的示例。

```
struct {
    char    *a;
    short    b;
    double   c;
    char     d;
    float    e;
    char     f;
    long     e;
    int      h;
}rec;
```

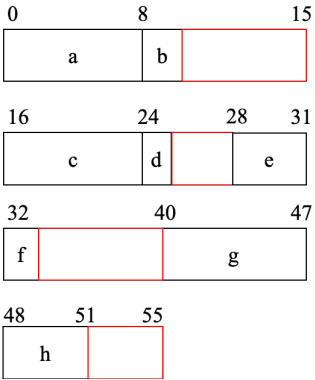
对于图中的这个结构体，可以画出所有字段起始地址的偏移量。

数据对齐

Datawhale

1. 变量 a 是一个指针变量，占 8 个字节。
2. 变量 b 是 short 类型，占两个字节，它起始地址的字节偏移量是 8，满足对齐规则的 2 的倍数。
3. 由于变量 c 是 double 类型，占 8 个字节，因此，该变量起始地址的偏移量需要是 8 的倍数，所以需要在变量 b 之后插入 6 个字节的间隙。
4. 对于变量 d 只占一个字节，顺序排列即可。
5. 由于变量 e 占 4 个字节，它的偏移量需要是 4 的倍数，因此，需要在变量 d 之后插入 3 个字节的间隙。
6. 同样变量 f 是 char 类型，顺序排列即可。
7. 由于变量 g 占 8 个字节，因此需要在变量 f 之后插入 7 个字节的间隙。
8. 最后一个变量 h 占 4 个字节，此时结构体的大小为 52 个字节，为了保证每个元素都满足对齐要求，还需要在结构体的尾端填充 4 个字节的间隙。

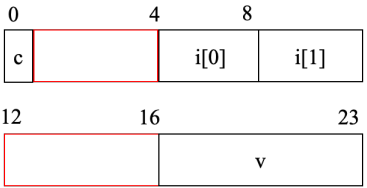
最终结构体的大小为 56 个字节，具体排列如图所示。



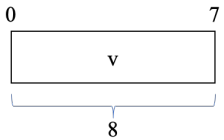
此外，关于更多情况的数据对齐情况，还需要针对不同型号的处理器以及编译系统进行具体分析。

与结构体不同，联合体中的所有字段共享同一存储区域，因此联合体的大小取决于它最大字段的大小。

```
struct S3{
    char    c;
    int     i[2];
    double  v;
};
```




```
union S3{
    char    c;
    int     i[2];
    double  v;
};
```



变量 v 和数组 i 的大小都是 8 个字节，因此，这个联合体的占 8 个字节的存储空间。

数据对齐

联合体的一种应用情况是：我们事先知道**两个不同字段的使用是互斥的**，那么我们可以将这两个字段声明为一个联合体。原理就是不会让不可能有数据的字段白白浪费内存。

例如，我们定义一个二叉树的数据结构，这个二叉树分为内部节点和叶子节点，其中每个内部节点不含数据，都有指向两个孩子节点的指针，每个叶子节点都有两个 double 类型的数据值。

我们可以用结构体来定义该二叉树的节点。

node

```
struct node_s{
    struct node_s *left;
    struct node_s *right;
    double data[2];
};
```

那么每个节点需要 32 个字节，由于该二叉树的特殊性，我们事先知道该二叉树的任意一个节点不是内部节点就是叶子节点，因此，我们可以用联合体来定义节点.

数据对齐

Datawhale

node

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

这样，每个节点只需要 16 个字节的存储空间，相对于结构体的定义方式，可以节省一半的空间。不过，这种编码方式存在一个问题，就是没有办法来确定一个节点到底是叶子节点还是内部节点，通常的解决方法是引入一个枚举类型，然后创建一个结构体，它包含一个标签和一个联合体

数据对齐

Datawhale

node

```
typedef enum{N_LEAF, N_INTERNAL} nodetype_t;
struct node_t{
    nodetype_t type;
    union{
      struct{
        union node_s *left;
        union node_s *right;
      }internal;
      double data[2];
    }info;
};
```

其中 type 占 4 个字节，联合体占 16 个字节，type 和联合体之间需要加入 4 个间隙，因此，整个结构体的大小为 24 个字节。

在这种例子中，虽然使用联合体可以节省存储空间，但是相对于给代码编写造成的麻烦，这样的节省意义不大。因此，对于有较多字段的情况，使用联合体带来的**空间节省**才会更吸引人。

除此之外，联合体还可以用来访问不同数据类型的位模式，当我们使用简单的强制类型转换，将 double 类型的数据转换成 unsigned long 类型时，除了 d 等于 0 的情况，二者的二进制位表示差别很大，这时我们可以将这两种类型的变量声明为一个联合体，这样就是可以以一种类型来存储，以另外一种类型来访问，变量 u 和 d 就具有相同的位表示。

union

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

我们通过一个代码示例看一下什么是缓冲区溢出。

echo

```
1 void echo(){
2     char buf[8];
3     gets(buf);
4     puts(buf);
5 }
```

数据对齐

Datawhale

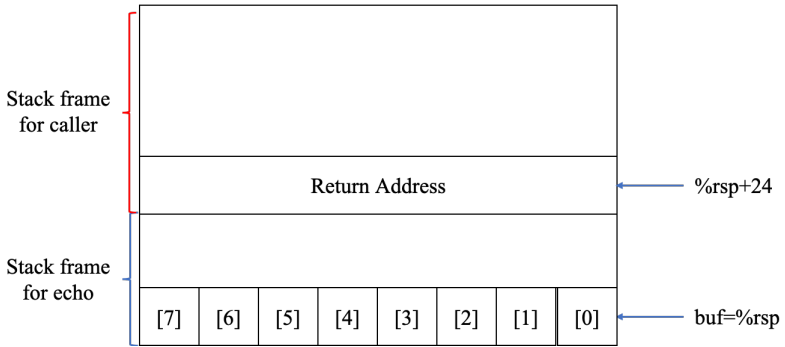
echo 函数声明了一个长度为 8 的字符数组。gets 函数是 C 语言标准库中定义的函数，它的功能是从标准输入读入一行字符串，在遇到回车或者某个错误的情况时停止，gets 函数将这个字符串复制到参数 buf 指明的位置，并在字符串结束的位置加上 null 字符。注意 gets 函数会有一个问题，就是它无法确定是否有足够大的空间来保存整个字符串，长一些字符串可能会导致栈上的其他信息被覆盖，通过汇编代码，我们可以发现实际上栈上分配了 24 个字节的存储空间。

```
void echo()
{
    char    buf[8];
    gets    (buf);
    puts    (buf);
};
```

```
echo:
    subq    $24,    %rsp
    movq    %rsp ,  %rdi
    call    gets
    movq    %rsp ,  %rdi
    call    puts
    addq    $24 ,   %rsp
    ret
```

数据对齐

为了方便表述，我们将栈的数据分布画了出来，其中字符数组位于栈顶的位置。



数据对齐

实际上当输入字符串的长度不超过 23 时，不会发生严重的后果，超过以后，返回地址以及更多的状态信息会被破坏，那么返回指令会导致程序跳转到一个完全意想不到的地方。

历史上许多计算机病毒就是利用缓冲区溢出的方式对计算机系统进行攻击的，针对缓冲区溢出的攻击，现在编译器和操作系统实现了很多机制，来限制入侵者通过这种攻击方式来获得系统控制权。

例如栈随机化、栈破坏检测以及限制可执行代码区域等。

栈随机化

Datawhale

main

```
1 int main(){
2     long local;
3     printf("local at %p\n", &local);
4     return 0;
5 }
```

在过去，程序的栈地址非常容易预测，如果一个攻击者可以确定一个 web 服务器所使用的栈空间，那就可以设计一个病毒程序来攻击多台机器，栈随机化的思想是栈的位置在程序每次运行时都有变化，上面这段代码只是简单的打印 main 函数中局部变量 local 的地址，每次运行打印结果都可能不同。

栈随机化

Datawhale

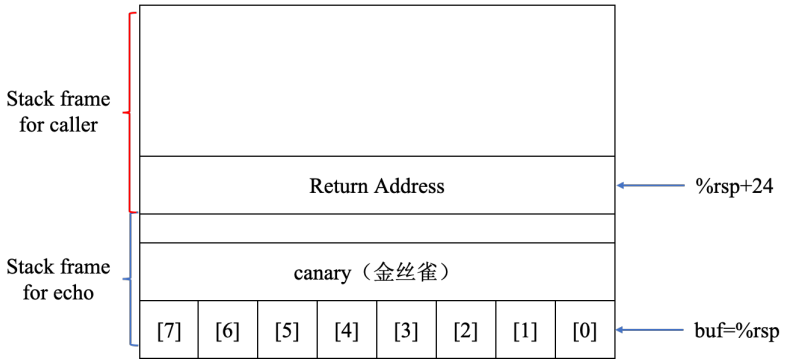
在 64 位 linux 系统上，地址的范围：0x7fff0001b698 0x7ffffffaa4a8。因此，采用了栈随机化的机制，即使许多机器都运行相同的代码，它们的栈地址也是不同的。

在 linux 系统中，栈随机化已经成为标准行为，它属于地址空间布局随机化的一种，简称 ASLR，采用 ASLR，每次运行时程序的不同部分都会被加载到内存的不同区域，这类技术的应用增加了系统的安全性，降低了病毒的传播速度。

栈破坏检测

Datawhale

编译器会在产生的汇编代码中加入一种栈保护者的机制来检测缓冲区越界，就是在缓冲区与栈保存的状态值之间存储一个特殊值，这个特殊值被称作金丝雀值，之所以叫这个名字，是因为从前煤矿工人会根据金丝雀的叫声来判断煤矿中有毒气体的含量。



栈破坏检测

Datawhale

金丝雀值是每次程序运行时随机产生的，因此攻击者想要知道这个金丝雀值具体是什么并不容易，在函数返回之前，检测金丝雀值是否被修改来判断是否遭受攻击。

接下来，我们通过汇编代码看一下编译器是如何避免栈溢出攻击的。

```
echo:
    subq    $24,    %rsp
    movq    %fs:40 , %rax
    movq    %rax ,  8(%rsp)
    xorl    %eax,   %eax
    movq    %rsp,   %rdi
    call    gets
    movq    %rsp ,  %rdi
    call    puts
    movq    8(%rsp), %rax
    xorq    %fs:40 , %rax
    je .L9
    call    __stack_chk_fail
.L9:
    addq    $24 ,   %rsp
    ret
```

栈破坏检测

Datawhale

图中这两行代码是从内存中读取一个数值，然后将该数值放到栈上，其中这个数值就是刚才提到的金丝雀值，存放的位置与程序中定义的缓冲区是相邻的。其中指令源操作数%fs:40 可以简单的理解为一个内存地址，这个内存地址属于特殊的段，被操作系统标记为“只读”，因此，攻击者是无法修改金丝雀值的。

```
echo:
    subq    $24,    %rsp
    movq    %fs:40 , %rax
    movq    %rax ,  8(%rsp)
    xorl    %eax,   %eax
    movq    %rsp,   %rdi
    call    gets
    movq    %rsp ,  %rdi
    call    puts
    movq    8(%rsp), %rax
    xorq    %fs:40 , %rax
    je .L9
    call    __stack_chk_fail
.L9:
    addq    $24 ,   %rsp
    ret
```

函数返回之前，我们通过指令 xor 来检查金丝雀值是否被更改。如果金丝雀值被更改，那么程序就会调用一个错误处理例程，如果没有被更改，程序就正常执行。

限制可执行代码区域

Datawhale

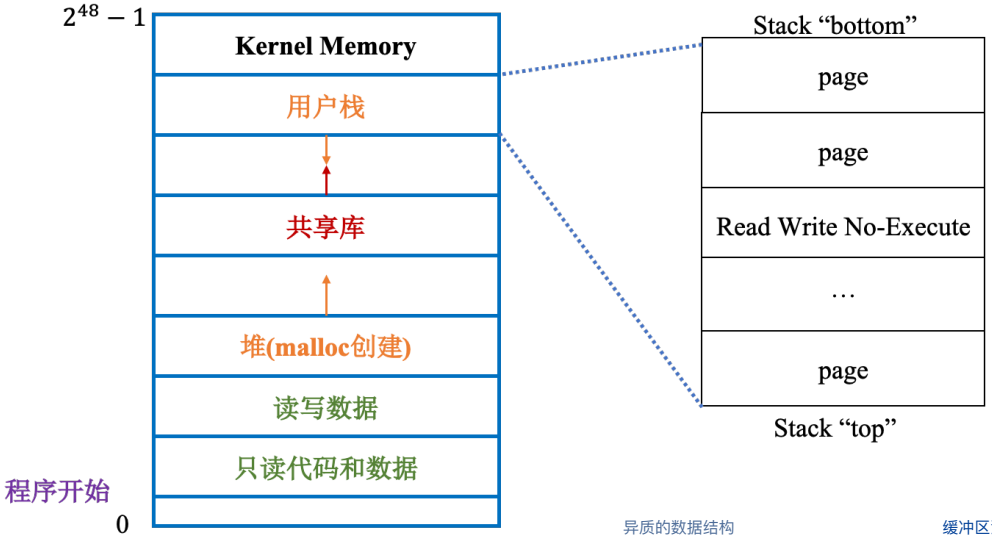
最后一种机制是消除攻击者向系统中插入可执行代码的能力，其中一种方法是限制哪些内存区域能够存放可执行代码。

以前，x86 的处理器将可读和可执行的访问控制合并成一位标志，所以可读的内存页也都是可执行的，由于栈上的数据需要被读写，因此栈上的数据也是可执行的。

虽然实现了一些机制能够限制一些页可读且不可执行，但是这些机制通常会带来严重的性能损失，后来，处理器的内存保护引入了不可执行位，将读和可执行访问模式分开了。有了这个特性，栈可以被标记为可读和可写，但是不可执行。检查页是否可执行由硬件来完成，效率上没有损失。

限制可执行代码区域

Datawhale



以上这三种机制，都不需要程序员做任何额外的工作，都是通过编译器和操作系统来实现的，单独每一种机制都能降低漏洞的等级，组合起来使用会更加有效。

不幸的是，仍然有方法能够对计算机进行攻击。