

《深入理解计算机系统》| 程序的机器级表示



唐鱼的学习探索

关注



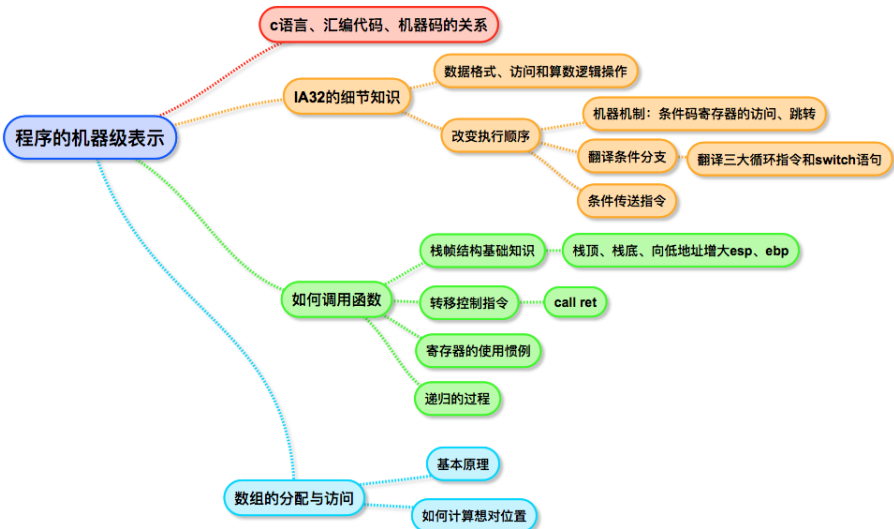
0.403

2016.10.30

23:23:14

字数 3,502

阅读 2,683



目 录

精通细节是理解更深和更基本概念的先决条件，这一章节首先讲解了C代码、汇编代码与机器代码的关系，再次重申了汇编的承上启下的重要作用。接着从IA32的细节一步步讲起，如何存储数据、如何访问数据、如何完成运算、如何进行跳转，在了解了这些细节以后告诉你我们常用的分支语句、循环语句是怎么完成了。在如何调用函数的部分，花费的篇幅较大，详细的讲解了栈帧结构，也让我们更好的了解了递归的过程。（其他方面还对数组、结构、联合有所讲解，难度不大）通过对编译器产生的汇编代码表示，我们了解了编译器和它的优化能力，知道了编译器为我们完成了哪些工作。

[本章内容]

- ※c语言、汇编代码以及机器代码之间的关系；
- ※介绍IA32的细节；
- ※讲解过程的实现，包括如何维护运行栈来支持过程间数据和控制传递；
- ※理解存储器访问越界问题，以及缓冲区溢出攻击问题；
- ※IA32扩展到64位（x86-64）

[笔记]

一、c语言代码、汇编代码、机器代码之间的关系

在第一章开始的部分我们就已经讲解过这三者的关系大概顺序是：1]C预处理器扩展源代码，展



唐鱼的学习探索

关注

总资产29 (约2.85元)

如何高效的准备一次考试

阅读 1,737

都9102年了，你还不知道anki是什么

阅读 102

开所以的#include命名的指定文件；2]编译器产生汇编代码（.s）；3]汇编器将汇编代码转化成二进制目标文件（.o）。二进制目标文件是很难阅读懂的，我们使用imac下的otool工具翻译如下：

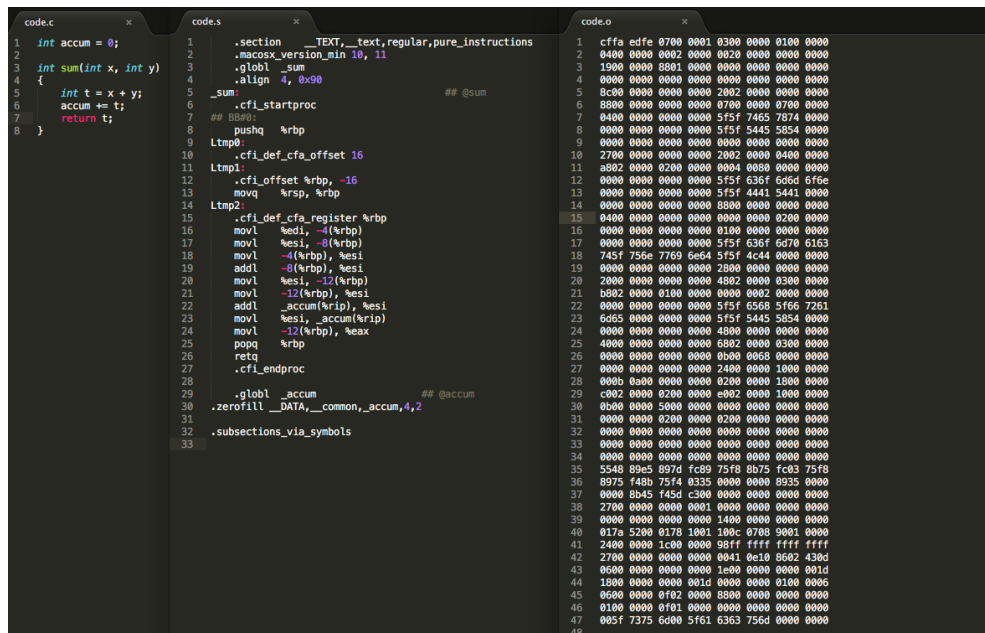


图1：从左到右分别是：c语言源码、汇编源码和机器目标码

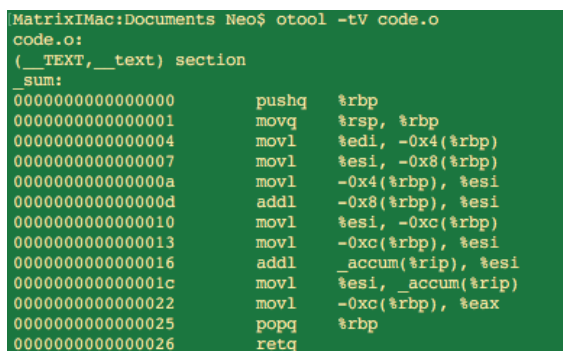


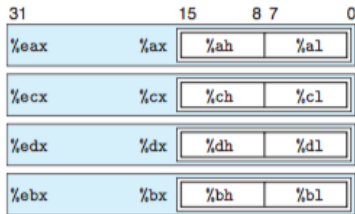
图2：使用命令：tool -tv

我们从图1中可以看出，汇编代码起到了承上启下的作用，目前已经不再要求程序员手写汇编代码，但是理解汇编代码以及与它的源C代码之间的联系，是理解程序如何执行的关键一步，因为编译器隐藏了太多的细节如：程序计数器、寄存器（整数、条件码、浮点）等。

二、IA32指令的细节

1) 区分字节与字：Intel使用术语“字”表示16位数据类型而“字节”代表的是8个位的数据。如果不习惯理解的话，可以做一个比喻“字节”相当于“字截”所以少了被截断了的嘛

2) 访问信息

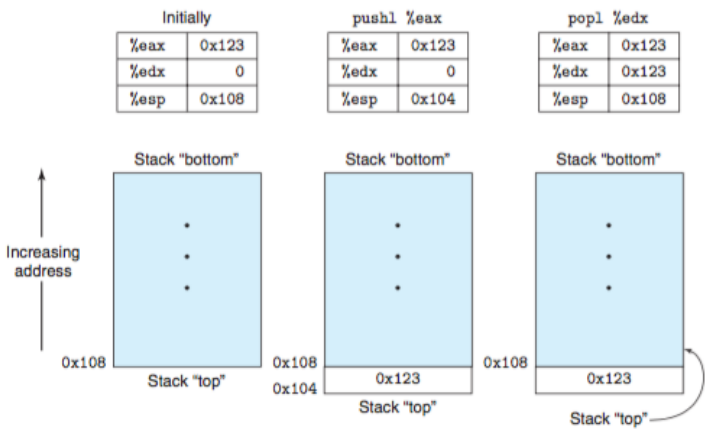


前4个寄存器可以独立访问低位字节

传送指令：move 源 --> 目的地（两个操作数不能同时指向存储器，需要寄存器周转）

指针：就是地址，间接引用指针就是将指针放入一个寄存器中，然后在存储器中使用这个寄存器

栈数据的基本理解：



地址向上增大，push向下压栈

push指令相当于：sub \$4, %esp 然后move %ebp, (%esp)

pop指令相当于：move (%esp), %eax 然后 add \$4, %esp

栈的数据结构是向低地址方向增长的，无论如何esp都是指向栈顶顶

3) 算数和逻辑操作

其实讲的就是加减乘除、与或非一系列的指令：

指令	效果	描述
<code>leal S, D</code>	$D \leftarrow \&S$	加载有效地址
<code>INC D</code>	$D \leftarrow D + 1$	加1
<code>DEC D</code>	$D \leftarrow D - 1$	减1
<code>NEG D</code>	$D \leftarrow -D$	取负
<code>NOT D</code>	$D \leftarrow \sim D$	取补
<code>ADD S, D</code>	$D \leftarrow D + S$	加
<code>SUB S, D</code>	$D \leftarrow D - S$	减
<code>IMUL S, D</code>	$D \leftarrow D * S$	乘
<code>XOR S, D</code>	$D \leftarrow D \oplus S$	异或
<code>OR S, D</code>	$D \leftarrow D \vee S$	或
<code>AND S, D</code>	$D \leftarrow D \& S$	与
<code>SAL k, D</code>	$D \leftarrow D \ll k$	左移
<code>SHL k, D</code>	$D \leftarrow D \ll k$	左移（等同于SAL）
<code>SAR k, D</code>	$D \leftarrow D \gg_A k$	算术右移
<code>SHR k, D</code>	$D \leftarrow D \gg_L k$	逻辑右移

加载有效地址指令: `leal S, D ==> (&S-->D)` 将有效地址写入到目的操作数中去

汇编代码与C语言源码中的顺序可能不同:

```
1  int arith(int x,
2      int y,
3      int z)
4  {
5      int t1 = x+y;
6      int t2 = z*48;
7      int t3 = t1 & 0xFFFF;
8      int t4 = t2 * t3;
9      return t4;
10 }
```

a) C 语言代码

lopSage.com

```
      x at %ebp+8, y at %ebp+12, z at %ebp+16
1  movl 16(%ebp), %eax      z
2  leal (%eax,%eax,2), %eax  z*3
3  sall $4, %eax           t2 = z*48
4  movl 12(%ebp), %edx      y
5  addl 8(%ebp), %edx       t1 = x+y
6  andl $65535, %edx       t3 = t1&0xFFFF
7  imull %edx, %eax        Return t4 = t2*t3
```

b) 汇编代码

leal和sall组合实现了z*48

4) 改变执行顺序

a. 机器机制

我们这里用到的是条件码寄存器，常见的有：

CF:	(unsigned) t < (unsigned) a	无符号溢出
ZF:	(t == 0)	零
SF:	(t < 0)	负数
OF:	(a < 0 == b < 0) && (t < 0 != a < 0)	有符号溢出

4个常用的条件码寄存器

可以通过cmp和test设置条件码寄存器：

指令	基于	描述
CMP S_2, S_1	$S_1 - S_2$	比较
cmpb cmpw cmpl	Compare byte Compare word Compare double word	
TEST S_2, S_1	$S_1 \& S_2$	测试
testb testw testl	Test byte Test word Test double word	

比较和测试 只修改条件码寄存器

通过set指令访问条件码，用处是设置值 or 跳转 or 传送数据：

指令	同义名	效果	设置条件
sete <i>D</i>	setz	$D \leftarrow ZF$	相等/零
setne <i>D</i>	setnz	$D \leftarrow \sim ZF$	不等/非零
sets <i>D</i>		$D \leftarrow SF$	负数
setns <i>D</i>		$D \leftarrow \sim SF$	非负数
setg <i>D</i>	setnle	$D \leftarrow \sim (SF \wedge OF) \wedge \sim ZF$	大于 (有符号>)
setge <i>D</i>	setnl	$D \leftarrow \sim (SF \wedge OF)$	大于等于 (有符号>=)

注：set指令后缀是表示不同的操作数

跳转指令（对于理解链接非常重要）：

指令	同义名	跳转条件	描述
jmp <i>Label</i>		1	直接跳转
jmp <i>*Operand</i>		1	间接跳转
je <i>Label</i>	jz	ZF	相等/零
jne <i>Label</i>	jnz	$\sim ZF$	不相等/非零
js <i>Label</i>		SF	负数
jns <i>Label</i>		$\sim SF$	非负数
jg <i>Label</i>	jnle	$\sim (SF \wedge OF) \wedge \sim ZF$	大于 (有符号>)
jge <i>Label</i>	jnl	$\sim (SF \wedge OF)$	大于或等于 (有符号>=)
jl <i>Label</i>	jnge	$SF \wedge OF$	小于 (有符号<)
jle <i>Label</i>	jng	$(SF \wedge OF) \mid ZF$	小于或等于 (有符号<=)
ja <i>Label</i>	jnb	$\sim CF \wedge \sim ZF$	超过 (无符号>)
jae <i>Label</i>	jnb	$\sim CF$	超过或相等 (无符号>=)
jb <i>Label</i>	jnae	CF	低于 (无符号<)
jbe <i>Label</i>	jna	$CF \mid ZF$	低于或相等 (无符号<=)

根据条件的不同进行条件，用于改变程序的执行顺序

直接跳转用：‘.’

间接跳转用：‘*’

理解跳转指令的目标编码：

```
1      8: 7e 0d          jle 17 <silly+0x17> Target = dest2
2      a: 89 d0          mov %edx,%eax dest1:
```

0xd并不是目标地址，而是0xd+0xa

jle跳转指令中的0d并不是目标地址，而真正的地址是通过计算0d+0a来确定的，这样做的优点是：通过使用与机器相关目标使得代码简洁，可以使目标代码移到存储器中而不是简单的地址，执行的是程序计数器与目标代码的加法。

b 翻译条件分支

通过将C代码翻译成不良的goto语句可以方便我们理解汇编代码的执行方式。汇编程序通过条件测试和跳转来实现循环，我们常见的循环语句其实都是翻译成了do-while形式的：

```

1 int fact_do(int n)
2 {
3     int result = 1;
4     do {
5         result *= n;
6         n = n-1;
7     } while (n > 1);
8     return result;
9 }

```

a) C 代码

寄存器	变量	初始值
-----	----	-----

```

Argument: n at %ebp+8
Registers: n in %edx, result in %eax
1  movl    8(%ebp), %edx    Get n
2  movl    $1, %eax        Set result = 1
3  .L2:                                loop:
4  imull   %edx, %eax       Compute result *= n
5  subl    $1, %edx        Decrement n

```

do-while 循环

while 循环会先转成 do-while 形式：

```

1 int fact_while(int n)
2 {
3     int result = 1;
4     while (n > 1) {
5         result *= n;
6         n = n-1;
7     }
8     return result;
9 }

```

a) C 代码

```

1 int fact_while_goto(int n)
2 {
3     int result = 1;
4     if (n <= 1)
5         goto done;
6     loop:
7         result *= n;
8         n = n-1;
9         if (n > 1)
10            goto loop;
11    done:
12        return result;
13 }

```

b) 等价的 goto 版本

```

Argument: n at %ebp+8
Registers: n in %edx, result in %eax
1  movl    8(%ebp), %edx    Get n
2  movl    $1, %eax        Set result = 1
3  cmpl    $1, %edx        Compare n:1
4  jle     .L7             If <=, goto done
5  .L10:                                loop:
6  imull   %edx, %eax       Compute result *= n
7  subl    $1, %edx        Decrement n
8  cmpl    $1, %edx        Compare n:1
9  jg      .L10            If >, goto loop
10 .L7:                                done:
    Return result

```

c) 对应的汇编代码

while 循环

for 循环也是一样的道理，先转成 do-while 形式：

```

1 int fact_for(int n)
2 {
3     int i;
4     int result = 1;
5     for (i = 2; i <= n; i++)
6         result *= i;
7     return result;
8 }

```

```

Argument: n at %ebp+8
Registers: n in %ecx, i in %edx, result in %eax
1  movl    8(%ebp), %ecx    Get n
2  movl    $2, %edx        Set i to 2 (init)
3  movl    $1, %eax        Set result to 1
4  cmpl    $1, %ecx        Compare n:1 (test)
5  jle     .L14            If <=, goto done
6  .L17:                                loop:
7  imull   %edx, %eax       Compute result *= i (body)
8  addl    $1, %edx        Increment i (update)
9  cmpl    %edx, %ecx       Compare n:i (test)
10 jge     .L17            If >=, goto loop
11 .L14:                                done:

```

```

1 int fact_for_goto(int n)
2 {
3     int i = 2;
4     int result = 1;
5     if (i <= n)
6         goto loop;
7     loop:
8     result *= i;
9     i++;
10    if (i <= n)
11        goto loop;
12    done:
13    return result;
14 }

```


for循环

switch语句：使用一个数组作为跳转表

```
1 int switch_eg(int x, int n) {
2     int result = x;
3
4     switch (n) {
5
6     case 100:
7         result += 13;
8         break;
9
10    case 102:
11        result += 10;
12        /* Fall through */
13
14    case 103:
15        result += 11;
16        break;
17
18    case 104:
19    case 106:
20        result += result;
21        break;
22
23    default:
24        result = 0;
25    }
26
27    return result;
28 }
```

a) switch 语句

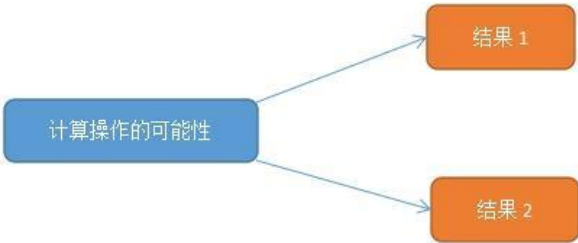
```
1 int switch_eg_impl(int x, int n) {
2     /* Table of code pointers */
3     static void *jt[7] = {
4         &loc_A, &loc_def, &loc_B,
5         &loc_C, &loc_D, &loc_def,
6         &loc_D
7     };
8
9     unsigned index = n - 100;
10    int result;
11
12    if (index > 6)
13        goto loc_def;
14
15    /* Multiway branch */
16    goto *jt[index];
17
18    loc_def: /* Default case */
19        result = 0;
20        goto done;
21
22    loc_C: /* Case 103 */
23        result = x;
24        goto rest;
25
26    loc_A: /* Case 100 */
27        result = x * 13;
28        goto done;
29
30    loc_B: /* Case 102 */
31        result = x + 10;
32        /* Fall through */
33
34    rest: /* Finish case 103 */
35        result += 11;
36        goto done;
37
38    loc_D: /* Cases 104, 106 */
39        result = x * x;
40        /* Fall through */
41
42    done:
43        return result;
44 }
```

b) 翻译到扩展的C语言

着重理解跳转的创建与使用

在switch语句的汇编代码中，我们使用的是一个数组jt来表示所以可能的7种情况，使用n-100将范围缩小到了0-6区间。其中102到103中间没有break，桤木的扩展C代码中也巧妙的实现了这样的效果。

c 条件传送指令



先计算出可能的多种不同结果

如条件语句： $x < y ? y - x : x - y$ ；就会先计算两种结果 $y - x$ 和 $x - y$ 的值，然后再判断 x, y 的大小

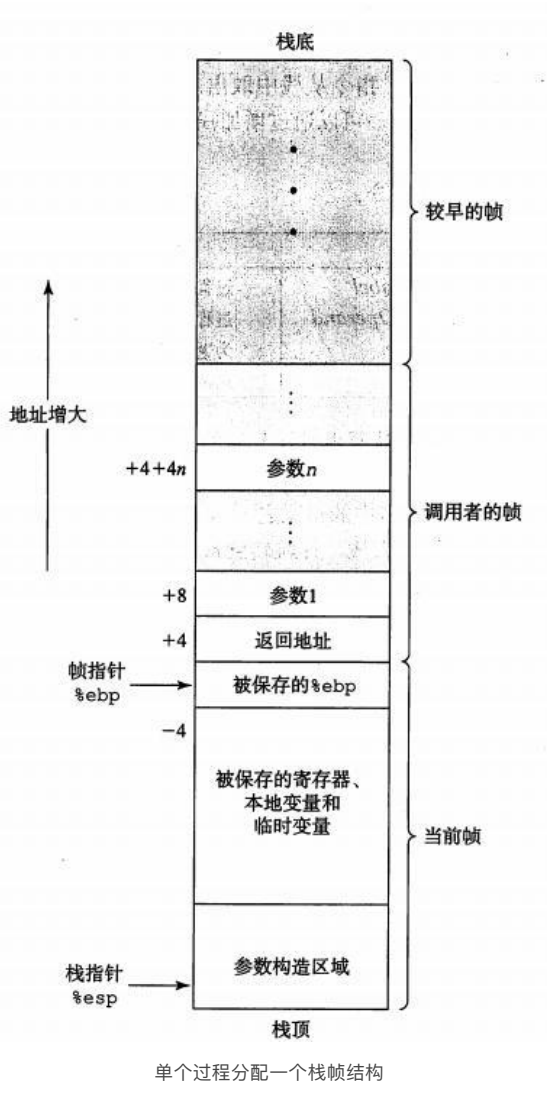
现代计算机CPU使用流水线方式实现性能的最优化：

举个例子，你去一家快餐店点餐，想要吃一个鸡蛋、一碗稀饭和一个包子，如果都是现场马上给你做的话，味道肯定最好，但花费的时间我估计你不会再去吃第二次了。然而真实的情况是，服务员将你的要求传达下去，宾果~~很快的时间就准备好了你需要的食物。这是因为快餐店已经从很早开始就将大家喜欢吃的都做好了，所以的结果（鸡蛋、稀饭、包子）都已经提前准备好了，这时候只需要根据你的需要（x, y的大小）马上就能给你上菜了。

在这个过程中，我们提前处理了一部分指令，如制作包子过程中的和面、包肉、上蒸笼我们成功的预测了90%的人早餐喜欢吃包子。就大大的节约了时间。记住所以的结果都提前准备好了的。

三、如何调用函数

我们如果要调用一个函数，实现将数据和控制代码从一个部分到另一个部分的跳转。我们如何来分配执行函数的变量空间，并在返回的时候释放空间，将返回值返回。用什么样的数据结构实现这一系列的操作：



帧指针与栈指针的不公之处：ebp放与参数与返回地址的最下方，方便计算参数的偏移位置；而esp一直在栈顶，可以通过push将数据压入，通过pop取出，增加指针来释放空间。

1) 转移控制

指令	描述
call <i>Label</i>	过程调用
call <i>*Operand</i>	过程调用
leave	为返回准备栈
ret	从过程调用中返回

常用转移控制指令

其中call先将返回地址入栈，然后跳转到函数开始的地方执行。（返回地址是开始调用者执行call的后面那条指令的地址）当遇到ret指令的时候，弹出返回地址，并跳转到该处继续执行调用者剩余部分。

2) 寄存器使用惯例

1] eax edx ecx 调用者保存，可以被调用者使用。

举个例子：这里的调用者就像很有票子的王健林一样，儿子王思聪可以无偿的使用王健林的票子

2] ebx esi edi 被调用者保存，在使用前被调用者要把这里面的值保存好，用完之后还回去

举个例子：这里就像有我有一辆豪车，可以把车子借给朋友使用，但是一定要把钥匙保存好，用完了之后还回来

3) 递归的过程

我们可以理解，递归的调用其实与其他函数的调用是一样的，因为计算机使用的是栈帧结构，为每个单独的调用创建了一个栈帧，每次调用都有私有的状态信息。



每个调用都有独立的栈帧结构

五、数组的分配与访问

1) 基本原则

数组的声明就不用多说了，来看看声明过后数组的具体位置

```
char    A[12];
char    *B[8];
double  C[6];
double  *D[5];
```

这些声明产生的数组带下列参数：

数组	元素大小	总的大小	起始地址	元素 i
A	1	12	x_A	$x_A + i$
B	4	32	x_B	$x_B + 4i$
C	8	48	x_C	$x_C + 8i$
D	4	20	x_D	$x_D + 4i$

x_A 代表的是起始地址

汇编代码使用move指令来简化访问：

```
movl (%edx, %ecx, 4), %eax
```

假设E是一个int类型的数组，我们要计算E[i]的值，在此，E的地址放于edx中，而i放于ecx中，我们通过上面的指令就完成了 $x_e + 4i$ 来读取其中的值，放在了eax中去。

2) 指针运算：对指针的运算其实际是按照相应的数据大小进行了伸缩

$point + i = X_p + (\text{数据大小}) \cdot L \cdot i$

如何计算二维数组的大小呢？

我们定义一个int D[5][3]的数组，形如：



如果我们要计算D[4, 2]的地址，就可以使用

$$D[i][j] = Xd + L(C * i + j) = D[0,0] + 4 * (3 * 4 + 2)$$

由于每组有3个数据，所以跳过一组就要乘以3，跳过4组就12个，再加上偏移的2，就是最后一个数据的地址了。

3) 理解指针：数组与指针关系密切

①指针用&符号创造、用*符号间接引用

②指针从一个类型 转为另外一个类型，只是伸缩因子变化，不改变它的值

③指针可以指向函数：int (*f) (int *) 从f开始由内往外阅读，首先f代表的是一个指向函数的指针，这个函数的参数是int * 返回值是int

六、结构与联合

1) 结构：所有的组成部分在存储器中连续存放，指向结构的指针指向结构的第一个字节；结构的各个字段的选取是在编译时处理，机器代码不包含字段的声明或字段名字的信息。

2) 联合：一个联合的总大小等于它最大字段的大小，而指向一个联合的指针，引用的是数据结构的起始位置。应用在：

a.如果两个数据互斥，减少空间；

b.访问不同数据的位模式；

```
1 unsigned float2bit(float f)
2 {
3     union {
4         float f;
5         unsigned u;
6     } temp;
7     temp.f = f;
8     return temp.u;
9 };
```

使用不同的位模式访问数据

用有符号数据存储，而返回的确实无符号的数据。特别注意的是，如果用联合将不同大小的数据组合到一起的时候要注意字节的顺序。

3) 数据对齐：要求某个类型对象的数据地址必须是（2.4.8）的倍数

```
1      .section      .rodata
2      .align 4      Align address to multiple of 4
3      .l7:
```

其中的.align 4要求数组开始的位置为4的倍数，由于每个单个数据的长度也是4的倍数，也就保证了后续的数据是4的倍数，数据对齐了。这种设计简化了，处理器与存储器之间接口的硬件设计。这种设计，编译器甚至会在字段中间、后面插入间隙，以保证每个结构满足上述要求。如下图所示：

结构的中间插入间隙，保证数据对齐

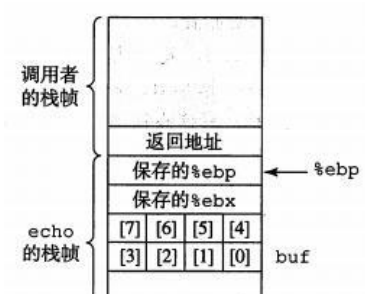
七、存储器越界引用和缓冲区溢出

由于C对于数组不进行边界检查，在栈帧结构中局部变量和状态信息，特别是返回地址也是在栈中存放的，对越界数据的访问和修改将破坏掉这些数据，当ret试图返回的时候，错误的地址（甚至是被修改的恶意目的地地址）会带来严重的安全隐患。

```
1  /* Sample implementation of library function gets() */
2  char *gets(char *s)
3  {
4      int c;
5      char *dest = s;
6      int gotchar = 0; /* Has at least one character been read? */
7      while ((c = getchar()) != '\n' && c != EOF) {
8          *dest++ = c; /* No bounds checking! */
9          gotchar = 1;
10     }
11     *dest++ = '\0'; /* Terminate string */
12     if (c == EOF && !gotchar)
13         return NULL; /* End of file or error */
14     return s;
15 }
16
17 /* Read input line and write it back */
18 void echo()
19 {
20     char buf[8]; /* Way too small! */
21     gets(buf);
22     puts(buf);
23 }
```

越界访问

上图两段代码展示了一个get函数在只有8个字节的空间中，存入了太多的数据，使得栈数据不断被破坏的过程。



在计算机中存储的 顺序

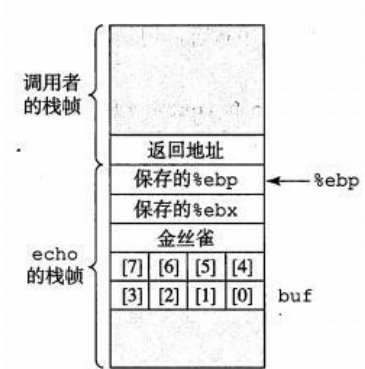
输入的字符数量	附加的被破坏的状态
0-7	无
8-11	保存的 %ebx 的值
12-15	保存的 %ebp 的值
16-19	返回地址
20+	caller 中保存的状态

被破坏的过程

常见的攻击方式是覆盖返回地址，使得程序跳转到所插入的恶意代码部分。

对抗方式：

- ① 栈随机化：在程序开始时，随机分配一段0-n的空间，使得栈的位置每次运行都不同。栈地址随机化，即使在一台机器上运行同样的程序，地址都是不同的。
- ② 栈破坏检测：插入栈保护者，俗称金丝雀的一段随机大小



在数组buf和保存状态之间放入一个特殊的金丝雀，代码检查该值，确定栈状态是否被改变

- ③ 限制可执行代码区域

八、x86-64：将IA32扩展到64位

- 1) 数据类型的比较：

C声明	Intel数据类型	汇编代码后缀	x86-64大小 (字节)	IA32大小
char	字节	b	1	1
short	字	w	2	2
int	双字	l	4	4

指针和长整数是64位

2) 访问信息：

callee被调用者保存， caller调用者保存

注：pc相对寻址-立即数+下条指令地址

- 3) 算术指令：当大小不同的操作数混在一起的时候，必须进行正确的扩展
- 4) 控制指令：增加了cmpq、testq指令

增加cmpq与testq指令

a 过程

由于寄存器翻了一倍，64位中不需要栈帧来存储参数，而是直接使用寄存器：

- 参数（最多是前六个）通过寄存器传递到过程，而不是在栈上。这消除了栈上存储和检索值的开销。
- `callq` 指令将一个 64 位返回地址存储在栈上。
- 许多函数不需要栈帧。只有那些不能将所有的局部变量都放在寄存器中的函数才需要在栈上分配空间。
- 函数最多可以访问超过当前栈指针值 128 个字节的栈上存储空间（地址低于当前栈指针的值）。这允许一些函数将信息存储在栈上而无需修改栈指针。
- 没有帧指针。作为替代，对栈位置的引用相对于栈指针。大多数函数在调用开始时分配所需要的整个栈存储，并保持栈指针指向固定位置。
- 同 IA32 一样，有些寄存器被指定为被调用者保存寄存器。任何要修改这些寄存器的过程都必须保存并恢复它们。

主要不同的地方

简书

首页

下载APP

搜索

Q

Aa

beta

登录

注册

以下原因会使用栈帧结构：

- 局部变量太多，不能都放在寄存器中。
- 有些局部变量是数组或者结构。
- 函数用取地址操作符（&）来计算一个局部变量的地址。
- 函数必须将栈上的某些参数传递到另一个函数。
- 在修改一个被调用者保存寄存器之前，函数需要保存它的状态。



c 寄存器保存惯例

被调用者保存： rbp 12 13 14 15号寄存器

调用者保存： 16-31 号寄存器

d 数据结构：严格对齐要求



9人点赞 >



《深入理解计算机系统》



"小礼物走一走，来简书关注我"

赞赏支持

还没有人赞赏，支持一下



唐鱼的学习探索 如果我像一般人一样读那么多书，我就跟他们一样愚蠢了。
总资产29 (约2.85元) 共写了10.4W字 获得530个赞 共463个粉丝

关注

推荐阅读

三面字节跳动被虐得“体无完肤”，
15天读完这份pdf，终拿下美团研发
阅读 79,835

对于二本渣渣来说，面试阿里P6也太难了！（两年crud经验，已拿
阅读 45,316

利用python爬取微博数据
阅读 22

字体反爬
阅读 6

2020 高校战“疫”网络安全分享赛
pwn
阅读 762



Download a VPN & Try Risk-Free

With a VPN You Can Surf the Internet with No Censorship. Blazing-Fast Speeds!

写下你的评论...

全部评论 1

只看作者

按时间倒序 按时间正序



立己达人

2楼 2017.05.03 22:50






写下你的评论...

评论1

赞9

...

被以下专题收入，发现更多相似内容


-  编程
-  操作系统
-  《深入理解计算...

推荐阅读

更多精彩内容>

《深入理解计算机系统》-程序的机器级表示

在高级语言横行的现在，能看懂机器语言的程序员并不多。了解了寄存器，汇编等知识后，才能对进程，线程有更深入的认识，而不...

 gatsby_dhn 阅读 253 评论 0 赞 0



汇编入门（长文多图，流量慎入！！）

8086汇编 本笔记是笔者观看小甲鱼老师（鱼C论坛）《零基础入门学习汇编语言》系列视频的笔记，在此感谢他和像他一样...

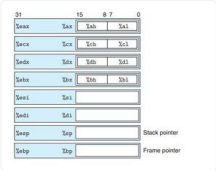
 Gibbs基 阅读 16,249 评论 7 赞 87



计算机系统之汇编

程序编码 对于机器级编程来说,两种抽象比较重要,一种是机器级程序的格式和行为,为指令集体系结构(ISA),包括IA...

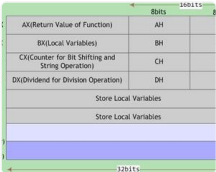
 fredal 阅读 1,739 评论 2 赞 18



[转载]C语言函数调用栈

原文地址：C语言函数调用栈(一)C语言函数调用栈(二) 0 引言 程序的执行过程可看作连续的函数调用。当一个函数执...

 小猪啊呜 阅读 2,593 评论 1 赞 17



读书笔记 - 《程序员的自我修养》

一、温故而知新 1. 内存不够怎么办 内存简单分配策略的问题地址空间不隔离内存使用效率低程序运行的地址不确定 关于...



SeanCST 阅读 3,726 评论 0 赞 21