

实验五：内存分配、缓冲区溢出实验

实验五：内存分配、缓冲区溢出实验

实验5.1

- 一、实验目的
- 二、实验内容
 - 1. 实验现象
 - 2. 反汇编源程序回答问题
 - 3. 解释原因

实验5.2

- 一、实验目的
- 二、实验内容
 - 1. 运行
 - 2. 反汇编源程序
 - 3. 未污染前原始存放数据
 - 4. fgets函数和gets函数的异同
 - 5. 查阅资料阐述防御方法（资料来源CSAPP）

实验5.3

- 一、实验目的
- 二、实验内容
 - 1. 运行程序观察现象
 - 2. 回答32位平台size_t类型的范围，计算两次分配内存大小的不同
 - 3. 解释出现的实验现象
 - 4. 现有Linux允许malloc 0个字节的合理性

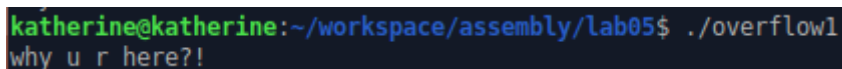
实验5.1

一、实验目的

- 1. 掌握基本的函数调用规则
- 2. 掌握局部变量在函数栈上的存放原则
- 3. 理解数组越界的危害

二、实验内容

1. 实验现象



```
katherine@katherine:~/workspace/assembly/lab05$ ./overflow1
why u r here?!
```

实验现象如上图，屏幕上打印出了字符串"why u r here?!"

2. 反汇编源程序回答问题

foo()函数的反汇编代码如下：

```

0000057b <foo>:
57b: 55          push    %ebp
57c: 89 e5       mov     %esp,%ebp
57e: 83 ec 10    sub     $0x10,%esp
581: e8 2e 00 00 00 call    5b4 <_x86.get_pc_thunk.ax>
586: 05 4e 1a 00 00 add     $0x1a4e,%eax
58b: 8d 80 79 e5 ff ff lea     -0x1a87(%eax),%eax
591: 89 45 04    mov     %eax,0x4(%ebp)
594: b8 00 00 00 00 mov     $0x0,%eax
599: c9         leave  %eax
59a: c3         ret

```

(1)从地址58b处的指令可以看出-0x1a87(%eax)是函数why_here的首地址，那么根据地址591处的指令，0x4(%ebp)就是buff[2]的位置。又因为buff是int型数组，那么buff[0]的地址就是%ebp-4=0xbffef30-4=0xbffef28

(2)变量buff[2]在内存中的开始地址就是函数foo的返回地址的起始位置，即0xbffef30

3. 解释原因

从2中的分析可以看出，在执行foo函数中的buff[2]=(int)why_here时，发生了数组越界，把foo函数的返回地址改成了why_here的首地址，因此函数返回时开始执行why_here函数，在屏幕上输出why u r here?!

实验5.2

一、实验目的

1. 掌握函数调用规则
2. 理解标准输入函数gets的缺陷
3. 掌握缓冲区溢出的基本原理
4. 了解防止缓冲区溢出的防御方法

二、实验内容

1. 运行

结果如下图：

```

katherine@katherine:~/workspace/assembly/lab05$ ./overflow2
So...The End...
aaaaaaaaaaaa
aaaaaaaaaaaa
or...maybe not?
katherine@katherine:~/workspace/assembly/lab05$ ./overflow2
So...The End...
aaaaaaaaaaaa
aaaaaaaaaaaa
Segmentation fault (core dumped)

```

可以看到当连续输入11个'a'时，没有异常；当连续输入12个'a'时，产生segmentation fault.

由此猜测需要连续输入12个字符才能产生segmentation fault.

2. 反汇编源程序

反汇编后函数doit()的汇编代码如下：

```

0000054d <doit>:
54d: 55          push    %ebp
54e: 89 e5       mov     %esp,%ebp
550: 53          push    %ebx
551: 83 ec 14    sub     $0x14,%esp
554: e8 f7 fe ff call    450 <__x86.get_pc_thunk.bx>
559: 81 c3 7b 1a 00 00 add     $0x1a7b,%ebx
55f: 83 ec 0c    sub     $0xc,%esp
562: 8d 45 f0    lea     -0x10(%ebp),%eax
565: 50          push    %eax
566: e8 65 fe ff call    3d0 <gets@plt>
56b: 83 c4 10    add     $0x10,%esp
56e: 83 ec 0c    sub     $0xc,%esp
571: 8d 45 f0    lea     -0x10(%ebp),%eax
574: 50          push    %eax
575: e8 66 fe ff call    3e0 <puts@plt>
57a: 83 c4 10    add     $0x10,%esp
57d: 90          nop
57e: 8b 5d fc    mov     -0x4(%ebp),%ebx
581: c9          leave
582: c3          ret

```

验证1中的猜测：从地址562、571两行可以看出数组开始的地址为-0x10(%ebp),又因为%ebp下四个字存放的是%ebx的值（地址550处指令），所以存放数组的空间一共是12字节，又因为字符串末尾是'\0'，所以最多可以存放除'\0'外的11个字符，当连续输入12个字符时产生segmentation fault.

查阅gets函数的资料

DESCRIPTION
Never use this function.

gets() reads a line from `stdin` into the buffer pointed to by `s` until either a terminating newline or EOF, which it replaces with a null byte ('\0'). No check for buffer overrun is performed (see BUGS below).

gets函数从输入流中读入字符串直到一行结束或者EOF，不检查溢出。

存放情况

这N+6的字节分别存放buf[0]-buf[N+5]

3. 未污染前原始存放数据

根据2中对汇编代码的推断，[0xbffef30+N,0xbffef30+N+3]原始存放的数据是调用者保存寄存器%ebx中的值，[0xbffef30+N+4,0xbffef30+N+7]原始存放的数据是旧的%ebp

4. fgets函数和gets函数的异同

用man命令查看fgets

fgets() reads in at most one less than `size` characters from `stream` and stores them into the buffer pointed to by `s`. Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer. A terminating null byte ('\0') is stored after the last character in the buffer.

fgets与gets相比有更多的参数，其中参数n指定了读取的最大字符数（包括最后的空字符），所以fgets函数最多读取n-1个有效字符。因为有参数n的限制，编译器可以根据n合理分配栈中给buf数组的空间，所以fgets函数不会发生缓冲区溢出的现象，用fgets函数无法完成上述攻击。

5. 查阅资料阐述防御方法（资料来源CSAPP）

栈随机化

栈随机化的思想使得栈的位置在程序每次运行时都有变化。程序开始时，在栈上分配一段0~n字节之间的随即大小的空间，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。当程序的栈地址不确定的时候，攻击者就很难知道入侵代码从哪个地址植入，就难以产生指向破坏代码的指针。

栈破坏检测

其思想是在栈帧中任何局部缓冲区与栈状态之间存储一个特殊的金丝雀(canary)值，是在程序每次运行时随机产生的，攻击者没有办法知道它的具体值。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了。如果是的，那么程序异常中止。

实验5.3

一、实验目的

1. 掌握不同机器平台不同类型整数的大小限制
2. 理解整数溢出的危害
3. 了解常用的抵御整数溢出的防御方法

二、实验内容

1. 运行程序观察现象

结果如图：

```
katherine@katherine:~/workspace/assembly/lab05$ ./overflow3 20
malloc 80 bytes
loop time:20[0x14]
katherine@katherine:~/workspace/assembly/lab05$ ./overflow3 1073741824
malloc 0 bytes
loop time:1073741824[0x40000000]
Segmentation fault (core dumped)
```

./main 20出现的现象：

```
malloc 80 bytes
loop time:20[0x14]
```

./main 1073741824出现的现象：

```
malloc 0 bytes
loop time:1073741824[0x40000000]
Segmentation fault(core dumped)
```

2. 回答32位平台size_t类型的范围，计算两次分配内存大小的不同

在32位系统中size_t是4字节的，它的范围是[0,4294967295]

第一次分配内存80 bytes

第二次分配内存 $\text{len} * \text{sizeof}(\text{int}) = 1073741824 * 4 = 4294967296$,超出了size_t的范围，发生整数溢出，实际分配0 bytes

3. 解释出现的实验现象

第一次正常分配80 bytes内存，并正常打印出loop time

由2可知，第二次分配0 bytes内存，由于现有Linux允许malloc 0个字节，因此不会返回空指针，if语句不会执行，因此继续执行下一步printf loop time。但是因为整数溢出，分配的空间是0 bytes，而len是一个非常大的值，因此在for循环的数组复制中出现了数组越界和缓冲区溢出，所以出现了Segmentation fault.

4. 现有Linux允许malloc 0个字节的合理性

尝试malloc 0个字节，得到的结果如下：

```
malloc 0 bytes  
loop time:0[0x0]
```

malloc函数的作用是在堆区分配一块指定大小的空间，再返回一个指向起始位置的指针。malloc()函数和free()函数配套使用，在使用完空间之后释放空间。我觉得malloc(0)就类似于释放了malloc(n)之后的结果，指针指向的位置不变，但是分配的空间为0 bytes，即这块空间的起始位置和终止位置地址相同。malloc 0实际上就是malloc n的特殊化，它也具备指针、起始位置、终止位置等要素，只不过分配的空间为0，但它返回的指针还是有效指针。我觉得在Linux中允许malloc 0可以将malloc 0和malloc n两种情况统一起来，简化了对malloc的定义。