

# 第八章 缓冲区溢出攻击实验报告

## 一、实验内容及环境

### 1.1 实验内容

在 32 位的 Ubuntu16.04 系统中用 `gcc -fno-stack-protector` 编译该程序，通过 GDB 调试，对 `f00()`，`f01()`，`f02()` 进行分析：

- (1) 函数 `f00()`，`f01()`，`f02()` 是否存在缓冲区溢出错误。
- (2) 如果存在缓冲区溢出错误，计算出被攻击的缓冲区与函数的返回地址所在的栈的距离（即偏移 OFFSET），给出溢出后的返回地址（用十六进制数表示）。

### 1.2 实验环境

HOST: Archlinux 64bit Version: 6.0.6-arch1-1

虚拟机环境: VirtualBox 7.0.2

虚拟机: Ubuntu16.04-i386

## 二、实验原理

缓冲区是一块用于存取数据的内存，其位置和大小在编译时确定或在程序运行时分配，缓冲区可以设置在栈和堆中。

当向缓冲区拷贝数据时，如果数据长度超过了缓冲区的长度，则多出的数据就会覆盖掉该缓冲区之外的高低之内存，从而就会覆盖邻近的内存，造成缓冲区溢出错误。

如果该错误被攻击者利用，溢出的数据覆盖了函数的返回地址，则攻击者就可以发动缓冲区溢出攻击，执行攻击者期望的代码。

典型的缓冲区溢出可以由 C 语言中的字符串拷贝操作实现。

由于函数里局部变量的内存分配是发生在栈帧里的，所以如果在某一个函数内部定义了局部变量，则这个缓冲区变量所占用的内存空间是在该函数被调用时所建立的栈帧里。

由于对缓冲区的潜在操作(比如字串的复制)都是从内存低地址到高地址的,而内存中所保存的函数返回地址往往就在该缓冲区的上方(高地址)——这是由于栈的特性决定的,这就为覆盖函数的返回地址提供了条件。

当用大于目标缓冲区大小的内容来填充缓冲区时,就可以改写保存在函数栈帧中的返回地址,从而改变程序的执行流程,执行攻击者的代码。

下面我们将对典型的字符串拷贝操作造成的缓冲区溢出进行实验。

### 三、实验过程

#### 3.1 关闭栈底地址随机化技术

```
sudo /sbin/sysctl -w kernel.randomize_va_space=0
```

#### 3.2 f00() 函数的情况

首先直接运行 homework08 程序，我们有

```
geng@geng-VirtualBox:~/Documents$ ./homework08
Segmentation fault (core dumped)
geng@geng-VirtualBox:~/Documents$
```

可见会发生段错误，接下来使用 gdb 对 homework08 程序进行调试：

首先反汇编 main () 和 f00 ()

```
(gdb) disassemble main
Dump of assembler code for function main:
0x0804853c <+0>:    lea    0x4(%esp),%ecx
0x08048540 <+4>:    and    $0xfffffffff0,%esp
0x08048543 <+7>:    pushl  -0x4(%ecx)
0x08048546 <+10>:   push   %ebp
0x08048547 <+11>:   mov    %esp,%ebp
0x08048549 <+13>:   push   %ebx
0x0804854a <+14>:   push   %ecx
0x0804854b <+15>:   mov    %ecx,%ebx
0x0804854d <+17>:   sub    $0x8,%esp
0x08048550 <+20>:   push   $0x400
0x08048555 <+25>:   push   $0x804a040
0x0804855a <+30>:   call   0x804843b <init_buf>
0x0804855f <+35>:   add    $0x10,%esp
0x08048562 <+38>:   mov    (%ebx),%eax
0x08048564 <+40>:   cmp    $0x2,%eax
0x08048567 <+43>:   je     0x804857a <main+62>
0x08048569 <+45>:   cmp    $0x3,%eax
0x0804856c <+48>:   je     0x8048581 <main+69>
0x0804856e <+50>:   cmp    $0x1,%eax
0x08048571 <+53>:   jne    0x8048588 <main+76>
0x08048573 <+55>:   call   0x80484a1 <f00>
0x08048578 <+60>:   jmp    0x804858e <main+82>
---Type <return> to continue, or q <return> to quit---
0x0804857a <+62>:   call   0x80484c4 <f01>
0x0804857f <+67>:   jmp    0x804858e <main+82>
0x08048581 <+69>:   call   0x8048500 <f02>
0x08048586 <+74>:   jmp    0x804858e <main+82>
0x08048588 <+76>:   call   0x80484a1 <f00>
0x0804858d <+81>:   nop
0x0804858e <+82>:   mov    $0x0,%eax
0x08048593 <+87>:   lea    -0x8(%ebp),%esp
0x08048596 <+90>:   pop    %ecx
0x08048597 <+91>:   pop    %ebx
0x08048598 <+92>:   pop    %ebp
0x08048599 <+93>:   lea    -0x4(%ecx),%esp
0x0804859c <+96>:   ret
End of assembler dump.
```

可以观察到程序通过 cmp 对比传入参数的个数，来确定运行

f00, f01, f02 中的其中一个函数

接下来在关键位置设置断点：

在函数 f00 的入口、对 strcpy 函数的调用、ret 出口处设置断点分析

```
End of assembler dump.
(gdb) b *(f00+0)
Breakpoint 1 at 0x80484a1
(gdb) disa
disable      disassemble
(gdb) disa
disable      disassemble
(gdb) disassemble f00
Dump of assembler code for function f00:
   0x080484a1 <+0>:    push    %ebp
   0x080484a2 <+1>:    mov     %esp,%ebp
   0x080484a4 <+3>:    sub     $0xe8,%esp
   0x080484aa <+9>:    sub     $0x8,%esp
   0x080484ad <+12>:   push    $0x804a040
   0x080484b2 <+17>:   lea     -0xdc(%ebp),%eax
   0x080484b8 <+23>:   push    %eax
   0x080484b9 <+24>:   call    0x8048300 <strcpy@plt>
   0x080484be <+29>:   add     $0x10,%esp
   0x080484c1 <+32>:   nop
   0x080484c2 <+33>:   leave
   0x080484c3 <+34>:   ret
End of assembler dump.
(gdb) b *(f00+24)
Breakpoint 2 at 0x80484b9
(gdb) b *(f00+34)
Breakpoint 3 at 0x80484c3
(gdb) r
```

随后运行程序并在断点处查看寄存器的值

函数的入口处的栈顶指针 esp 指向的栈（地址为 0xbfffeec）保存了函数 f00 返回到调用函数(main)的地址（0x08048578），即 f00 函数的返回地址，记 A=\$esp 目前的值

```
(gdb) r
Starting program: /home/geng/Documents/homework08

Breakpoint 1, 0x080484a1 in f00 ()
(gdb) display/i $eip
1: x/i $eip
=> 0x080484a1 <f00>:    push    %ebp
(gdb) x/x $esp
0xbfffeec:    0x08048578
(gdb) A=$esp
Ambiguous command "A=$esp": .
(gdb) c
Continuing.

Breakpoint 2, 0x080484b9 in f00 ()
1: x/i $eip
=> 0x080484b9 <f00+24>: call    0x8048300 <strcpy@plt>
(gdb) x/x $esp
0xbfffedd0:    0xbfffedec
(gdb) x/x $esp+4
0xbfffedd4:    0x0804a040
(gdb) x/x 0x0804a040
0x804a040 <Lbuffer>:    0x44434241
(gdb)
```

继续执行，执行到调用 strcpy 之前，此时该函数的参数已经入栈，由于 C 语言默认将参数逆序入堆栈，所以 x/x \$esp 查看到的是 strcpy 的第一个参数，即 buffer 的首地址(0xbffffedec)

记 B=buffer 的首地址，计算  $A-B=0xe0=224$ 。

因此，如果 Lbuffer 的内容超过 224 字节，将会发生缓冲区溢出，并且返回地址被改写。

可以通过观察反汇编的结果得知程序的 LEN 参数为 0x400, 即 1024。

```

a.c (~/.Documents) - gedit
Open Save

804853a:      c9          leave
804853b:      c3          ret

804853c <main>:
804853c:      8d 4c 24 04    lea    0x4(%esp),%ecx
8048540:      83 e4 f0       and    $0xffffffff0,%esp
8048543:      ff 71 fc       pushl  -0x4(%ecx)
8048546:      55            push   %ebp
8048547:      89 e5          mov    %esp,%ebp
8048549:      53            push   %ebx
804854a:      51            push   %ecx
804854b:      89 cb          mov    %ecx,%ebx
804854d:      83 ec 06       sub    $0x8,%esp
8048550:      68 00 04 00 00 push    $0x400
8048555:      68 40 a0 04 00 push    $0x804a040
804855a:      e8 dc fe ff ff call    804843b <init_buf>
804855f:      83 c4 10       add    $0x10,%esp
8048562:      8b 03          mov    (%ebx),%eax
8048564:      83 f8 02       cmp    $0x2,%eax
8048567:      74 11          je     804857a <main+0x3e>
8048569:      83 f8 03       cmp    $0x3,%eax
804856c:      74 13          je     8048581 <main+0x45>
804856e:      83 f8 01       cmp    $0x1,%eax
8048571:      75 15          jne    8048588 <main+0x4c>
8048573:      e8 29 ff ff ff call    80484a1 <f00>
8048578:      eb 14          jmp     804858e <main+0x52>
804857a:      e8 45 ff ff ff call    80484c4 <f01>
804857f:      eb 0d          jmp     804858e <main+0x52>
8048581:      e8 7a ff ff ff call    8048500 <f02>
8048586:      eb 06          jmp     804858e <main+0x52>
8048588:      e8 14 ff ff ff call    80484a1 <f00>
804858d:      90            nop
804858e:      b8 00 00 00 00 mov     $0x0,%eax
8048593:      8d 65 f8       lea    -0x8(%ebp),%esp
8048596:      59            pop     %ecx
8048597:      5b            pop     %ebx
8048598:      5d            pop     %ebp

```

所以上述 f00 肯定会发生溢出，继续执行

```
Breakpoint 3, 0x080484c3 in f00 ()
1: x/i $eip
=> 0x080484c3 <f00+34>: ret
```

即将执行 ret, ret 会把栈顶 (\$esp 指的) 的数据弹到 eip, 接着跳转到 eip 执行

此时查看\$esp 对应的数据

```
(gdb) x/x $esp
0xbfffeecc:    0x54535251
(gdb) x/s $esp
0xbfffeecc:     "QRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCD
FGHIJKLMNOPQRSTUVWXYZABCFGH"...
(gdb)
```



可见执行 ret 前 esp 指的内容为 0x54535251, 函数将跳到该处执行, 再执行就要越界报段错误

```
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x54535251 in ?? ()
1: x/i $eip
=> 0x54535251: <error: Cannot access memory at address 0x54535251>
(gdb)
```

f00() 函数会溢出, 偏移为 224, 返回地址更改为 0x54535251。

### 3.3 f01() 函数的情况

此时需要为 gdb --args ./homework08 x 为程序传入一个参数，使其执行 f01 ()

```

Type "apropos word" to search for commands related to "word"...
Reading symbols from ./homework08...(no debugging symbols found)...done.
(gdb) show args
Argument list to give program being debugged when it is started is "x".
(gdb)

```

接着同 2 中的步骤，我们需要在先反汇编查看 main 和 f01 函数，接着在关键位置设置断点

```

geng@geng-VirtualBox: ~/Documents
0x08048586 <+74>:    jmp     0x0804858e <main+82>
0x08048588 <+76>:    call   0x080484a1 <f00>
0x0804858d <+81>:    nop
0x0804858e <+82>:    mov     $0x0,%eax
0x08048593 <+87>:    lea     -0x8(%ebp),%esp
0x08048596 <+90>:    pop     %ecx
0x08048597 <+91>:    pop     %ebx
0x08048598 <+92>:    pop     %ebp
0x08048599 <+93>:    lea     -0x4(%ecx),%esp
0x0804859c <+96>:    ret
End of assembler dump.
(gdb) b *(f01+0)
Breakpoint 1 at 0x080484c4
(gdb) disas f01
Dump of assembler code for function f01:
0x080484c4 <+0>:    push    %ebp
0x080484c5 <+1>:    mov     %esp,%ebp
0x080484c7 <+3>:    sub     $0x4d8,%esp
0x080484cd <+9>:    sub     $0x8,%esp
0x080484d0 <+12>:   push    $0x400
0x080484d5 <+17>:   lea     -0x4d7(%ebp),%eax
0x080484db <+23>:   push    %eax
0x080484dc <+24>:   call    0x0804843b <init_buf>
0x080484e1 <+29>:   add     $0x10,%esp
0x080484e4 <+32>:   sub     $0x8,%esp
0x080484e7 <+35>:   lea     -0x4d7(%ebp),%eax
0x080484ed <+41>:   push    %eax
0x080484ee <+42>:   lea     -0xd7(%ebp),%eax
0x080484f4 <+48>:   push    %eax
0x080484f5 <+49>:   call    0x08048300 <strcpy@plt>
0x080484fa <+54>:   add     $0x10,%esp
0x080484fd <+57>:   nop
0x080484fe <+58>:   leave
0x080484ff <+59>:   ret
End of assembler dump.
(gdb) b *(f01+49)
Breakpoint 2 at 0x080484f5
(gdb) b *(f01+59)
Breakpoint 3 at 0x080484ff
(gdb)

```

函数的入口处的栈顶指针 esp 指向的栈（地址为 0xbfffeec）保存了函数 f01 返回到调用函数(main)的地址（0x0804857f），即 f01 函数的返回地址，记 A=\$esp 目前的值

```

error: no registers.
(gdb) r
Starting program: /home/geng/Documents/homework08 x

Breakpoint 1, 0x080484c4 in f01 ()
1: x/i $eip
=> 0x080484c4 <f01>:      push    %ebp
(gdb) x/x $esp
0xbfffecc:      0x0804857f
(gdb)

```

继续执行到 strcpy 之前，查看 esp 寄存器指代的值，可以知道 buff 的首地址为 0xbfffedf1，记为 B，计算 A-B=0xdb=219

```

(gdb) c
Continuing.

Breakpoint 2, 0x080484f5 in f01 ()
1: x/i $eip
=> 0x080484f5 <f01+49>:  call    0x8048300 <strcpy@plt>
(gdb) x/x $esp
0xbfffe9e0:      0xbfffedf1
(gdb)

```

继续执行查看返回地址的修改

```

Breakpoint 3, 0x080484ff in f01 ()
1: x/i $eip
=> 0x080484ff <f01+59>:  ret
(gdb) x/x $esp
0xbfffecc:      0x4f4e4d4c
(gdb) x/s $esp
0xbfffecc:      "LMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNO
WXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNO
YZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABC" ...
(gdb)

```

由此可以知道

f01() 函数发生了缓冲区溢出，偏移量为 219，返回地址修改为 0x4f4e4d4c



### 3.4 f02()函数的情况

首先传入 2 个参数

```
Type 'show configuration' for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./homework08...(no debugging symbols found)...done.
(gdb) show args
Argument list to give program being debugged when it is started is "x y".
(gdb)
```

接着通过反汇编 main 和 f02，在关键位置上设置断点。

```
0x08048581 <+69>: call 0x8048500 <f02>
0x08048586 <+74>: jmp 0x804858e <main+82>
0x08048588 <+76>: call 0x80484a1 <f00>
0x0804858d <+81>: nop
0x0804858e <+82>: mov $0x0,%eax
0x08048593 <+87>: lea -0x8(%ebp),%esp
0x08048596 <+90>: pop %ecx
0x08048597 <+91>: pop %ebx
0x08048598 <+92>: pop %ebp
0x08048599 <+93>: lea -0x4(%ecx),%esp
0x0804859c <+96>: ret
End of assembler dump.
(gdb) b *(f02+0)
Breakpoint 1 at 0x8048500
(gdb) disassemble f02
Dump of assembler code for function f02:
0x08048500 <+0>: push %ebp
0x08048501 <+1>: mov %esp,%ebp
0x08048503 <+3>: sub $0x4e8,%esp
0x08048509 <+9>: sub $0x8,%esp
0x0804850c <+12>: push $0x400
0x08048511 <+17>: lea -0x408(%ebp),%eax
0x08048517 <+23>: push %eax
0x08048518 <+24>: call 0x804843b <init_buf>
0x0804851d <+29>: add $0x10,%esp
0x08048520 <+32>: sub $0x8,%esp
0x08048523 <+35>: lea -0x408(%ebp),%eax
0x08048529 <+41>: push %eax
0x0804852a <+42>: lea -0x4dc(%ebp),%eax
0x08048530 <+48>: push %eax
0x08048531 <+49>: call 0x8048300 <strcpy@plt>
0x08048536 <+54>: add $0x10,%esp
0x08048539 <+57>: nop
0x0804853a <+58>: leave
0x0804853b <+59>: ret
End of assembler dump.
(gdb) b *(f02+49)
Breakpoint 2 at 0x8048531
(gdb) b *(f02+59)
Breakpoint 3 at 0x804853b
```

运行程序，在函数入口处读取返回地址的位置 0xbfffecc 记为 A

```
(gdb) r
Starting program: /home/geng/Documents/homework08 x y

Breakpoint 1, 0x08048500 in f02 ()
(gdb) x/x $esp
0xbfffeecc: 0x08048586
(gdb) █
```

接着执行

```
(gdb) c
Continuing.

Breakpoint 2, 0x08048531 in f02 ()
(gdb) x/x $esp
0xbfffe9d0: 0xbfffe9ec
(gdb) █
```

查看 buffer 的首地址为 0xbfffe9ec, 记为 B

计算  $A-B=0x4e=1248$ , 很明显小于 LENS 长度 1024, 不会溢出, 继续执行

```
Value can't be converted to integer.
(gdb) x/x $esp
0xbfffeecc: 0x08048586
(gdb) █
```

此时的返回地址是正常的返回地址

```
0xbfffeecc: 0x08048586
(gdb) c
Continuing.
[Inferior 1 (process 4105) exited normally]
(gdb) █
```

进程自然结束

所以 f02() 函数不会溢出, 偏移 1248, 返回地址是 0x08048586 正常的返回地址

#### 四、实验心得

通过本次实验，熟悉了解了缓冲区攻击的大致过程，同时简单学习到 linux 下对缓冲区攻击的防护措施，十分有收获。