计算机系统详解课程实验与心得体会

古宜民 PB1700002

2018年6月

第一部分：二进制炸弹实验报告

1. 介绍及准备

二进制炸弹为一个Linux下可执行C语言程序，包含6个阶段，每个阶段要输入特定字符串才能解除炸弹，不同阶段需要不同方法破解，通过反汇编及gdb指令级调试来获得所需内容。

运行环境：Linux 4.9.0-6-amd64 Debian 9, gcc 6.3.0, gdb 7.12.0

建造炸弹：按照README中指示，创建单机bomb，

./makebomb.pl -s ./src -b ./bombs

生成的文件中bomb（可执行）和bomb.c为主文件，其中bomb.c描述了6个阶段分别为phase\_1到phase\_6共6个函数。

1. 逐个阶段破解

**Phase\_1**

gdb加断点到phase\_1并运行，disas反汇编查看代码：

Dump of assembler code for function phase\_1:

=> 0x00005555555551b0 <+0>: sub $0x8,%rsp

0x00005555555551b4 <+4>: lea 0x1555(%rip),%rsi # 0x555555556710

0x00005555555551bb <+11>: callq 0x555555555693 <strings\_not\_equal>

0x00005555555551c0 <+16>: test %eax,%eax

0x00005555555551c2 <+18>: jne 0x5555555551c9 <phase\_1+25>

0x00005555555551c4 <+20>: add $0x8,%rsp

0x00005555555551c8 <+24>: retq

0x00005555555551c9 <+25>: callq 0x55555555579f <explode\_bomb>

0x00005555555551ce <+30>: jmp 0x5555555551c4 <phase\_1+20>

End of assembler dump.

可见以rsi、rip为参数调用了 strings\_not\_equal，之后查看返回值%eax，为0则返回，否则explode\_bomb

进入strings\_not\_equal进一步查看，发现分别直接或间接以%rsi和%rdi为参数调用了 string\_length：

0x0000555555555697 <+4>: mov %rdi,%rbx

0x000055555555569a <+7>: mov %rsi,%rbp

0x000055555555569d <+10>: callq 0x555555555675 <string\_length>

0x00005555555556a2 <+15>: mov %eax,%r12d

0x00005555555556a5 <+18>: mov %rbp,%rdi

0x00005555555556a8 <+21>: callq 0x555555555675 <string\_length>

看汇编知string\_length为求字符串长度的函数。之后比较两个string\_length的返回值：

0x00005555555556b2 <+31>: cmp %eax,%r12d

如果相等则继续，否则返回0x1，进而炸弹爆炸。之后一段代码相对较难理解，但可以看出包含循环和比较，可猜测为两字符串长度相等时依次比较每个字符。由于两个字符串的首地址是函数的参数，只要在gdb中看对应内存值即可。

(gdb) x/1s $rsi

0x555555556710: "Border relations with Canada have never been better."

(gdb) x/1s $rdi

0x555555758780 <input\_strings>: 'a' <repeats 17 times>

(gdb)

其中17个a是输入的字串。

于是phase\_1的答案就是Border relations with Canada have never been better.

**Phase\_2**

断点到phase\_2，看寄存器可见只有一个参数%rdi保存着输入的字串。之后函数调用了read\_six\_numbers函数（这时参数仍为%rdi，即输入的字串）。而read\_six\_numbers中有调用了sscanf库函数，用x/1s $rsi可见第二个参数为指向字串"%d %d %d %d %d %d"的指针，即让是sscanf读入6个整数。

回到phase\_2，仔细查看寄存器值可见，输入的6个整数存在%rsp为头指针的数组中，当从read\_six\_number返回后%rsp被复制到%rbp。一个数占4个字节。

(gdb) x/6d $rbp

0x7fffffffd790: 12 32 23 43

0x7fffffffd7a0: 34 4334

而在汇编中可以看到一个循环

0x00005555555551f5 <+37>: add $0x1,%ebx

0x00005555555551f8 <+40>: add $0x4,%rbp

0x00005555555551fc <+44>: cmp $0x6,%ebx

0x00005555555551ff <+47>: je 0x555555555212 <phase\_2+66>

0x0000555555555201 <+49>: mov %ebx,%eax

0x0000555555555203 <+51>: add 0x0(%rbp),%eax

0x0000555555555206 <+54>: cmp %eax,0x4(%rbp)

0x0000555555555209 <+57>: je 0x5555555551f5 <phase\_2+37>

0x000055555555520b <+59>: callq 0x55555555579f <explode\_bomb>

以%ebx为变量循环，初始为1，每次+1，直到%ebx为6结束并完成任务，最后一个je为判断指令，如果没有跳转，则向下执行explode\_bomb，失败。分析代码，可见每个循环过程为：

%ebx加一；

%rbp加四，即用户输入的、待判断的数组指针（设为a）向前一个数，a=a+1；

判断是否结束，如果结束则返回；

将%ebx移动到%eax；

将a[0]加到%eax；

比较%eax与a[1]，不等则爆炸，相等则下一次循环。

由这一过程，可知如果要满足需求，需要有a[0]+i==a[1]，i为循环变量%ebx。则只要知道数组第一个元素，就可以推出其他所以元素，而第一个元素并没有要求。于是任意指定一个第一个元素，如1 2 4 7 11 16，即可通过。

**Phase\_3**

同样断点在phase\_3，看到使用了sscanf从输入的字符串中读入了%d %c %d三个内容，之后判断sscanf的返回值，如果小于2，即没有输入正确，则炸弹爆炸。

之后是一个比较跳转指令

0x0000555555555242 <+41>: cmpl $0x7,0xc(%rsp)

0x0000555555555247 <+46>: ja 0x555555555359 <phase\_3+320>

0x0000555555555359 <+320>: callq 0x55555555579f <explode\_bomb>

看内存知0xc(%rsp)为输入的第一个%d对应的值，0x8(%rsp)为第二个%d值，0x7(%rsp)为%c值（ASCII值），而该指令表明第一个值应小于7，否则爆炸。  
 之后看到一系列操作后有一条指令jmpq \*%rax，是跳转表跳转。跳转表需要跳转到<phase\_3+330>，然后对0x7(%rsp)，即字符进行比较：

0x0000555555555363 <+330>: cmp 0x7(%rsp),%al

而字符串应有的值，即%al，%eax中的值并不确定，因为%eax在跳转表中可能被赋有不同的值。

对于跳转表，当第一个整数%d为4时，待跳转的$rax为0x5555555552ed，跳转到：

0x00005555555552ed <+212>: mov $0x64,%eax

0x00005555555552f2 <+217>: cmpl $0xc6,0x8(%rsp)

0x00005555555552fa <+225>: je 0x555555555363 <phase\_3+330>

0x00005555555552fc <+227>: callq 0x55555555579f <explode\_bomb>

可见第二个整数的值为0xc6=18，字符的值为chr(0x64)=d，输入4 d 198即可过关，而答案并不唯一，如5 o 342也可过关。

**Phase\_4**

同样，可见sscanf的输入为两个%d，读入的值存在0xc(%rsp)和0x8(%rsp)。之后判断返回值是否正确，并将0xe和0xc(%rsp)比较，如果0xc(%rsp)小于等于0xe即15则继续跳转，否则在下一条指令处跳转到explode\_bomb：

0x00005555555553f2 <+75>: cmpl $0xe,0xc(%rsp)

0x00005555555553f7 <+80>: jbe 0x5555555553d0 <phase\_4+41>

=> 0x00005555555553f9 <+82>: jmp 0x5555555553cb <phase\_4+36>

继续跳转后可以看到准备参数后调用了func4，三个参数为0xc(%rsp),0x0,0xe：

=> 0x00005555555553d0 <+41>: mov $0xe,%edx

0x00005555555553d5 <+46>: mov $0x0,%esi

0x00005555555553da <+51>: mov 0xc(%rsp),%edi

0x00005555555553de <+55>: callq 0x555555555373 <func4>

进入func4，看到func4是递归调用，而且代码不长，于是决定手动将其汇编码翻译成C代码并运行，枚举可能的输入值，查看输出。

// csapp bomblab phase\_4 func4

#include <stdio.h>

// a:edi b:esi c:edx

int func4(int a, int b, int c)

{

int d = c;//d:eax

d = d - b;

int e = d;//e:ebx

e = e >> 31;

e = e + d;

e = e >> 1;

e = e + b;

if(a < e){

c = e - 1;

return func4(a, b, c) + e;

}

else{

if(a > e){

b = e + 1;

return func4(a, b, c) + e;

}

else{

return e;

}

}

}

int main(){

int i;

for(i = 0; i < 14; i++){

printf("%d:\n", i);

printf("%d\n", func4(i, 0x0, 0xe));

}

}

得到结果：

0:

11

1:

11

2:

13

3:

10

4:

19

5:

15

6:

21

7:

7

8:

35

9:

27

10:

37

11:

18

12:

43

13:

31

看到phase\_4中在func返回后比较了返回值，如果返回值不是0x13=19即失败，于是输入的值应该为4才能返回19；之后比较了0x8(%rsp)：

0x00005555555553fb <+84>: cmpl $0x13,0x8(%rsp)

0x0000555555555400 <+89>: jne 0x5555555553e8 <phase\_4+65>

0x0000555555555402 <+91>: jmp 0x5555555553ed <phase\_4+70>

0x00005555555553e8 <+65>: callq 0x55555555579f <explode\_bomb>

0x00005555555553ed <+70>: add $0x18,%rsp

0x00005555555553f1 <+74>: retq

可见第二个输入的数即0x8(%rsp)为0x13=19。

所以输入4 19可过关。

第二部分：课程心得体会