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，从结果看到输入的值应该为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。

**Phase\_5**

待输入值为%d %d。查看汇编，发现输入并进行一些操作后%rsi处于一个地址值0x5555555567a0 <array.3095>，之后进入了一个循环：

0x0000555555555449 <+69>: add $0x1,%edx

0x000055555555544c <+72>: cltq

0x000055555555544e <+74>: mov (%rsi,%rax,4),%eax

0x0000555555555451 <+77>: add %eax,%ecx

0x0000555555555453 <+79>: cmp $0xf,%eax

0x0000555555555456 <+82>: jne 0x555555555449 <phase\_5+69>

%edx为循环变量（由之前内容知其初值为0），%ecx为累加变量，到%eax值为0xf时停止（由之前内容知%eax初值为输入的第一个数和0xf的and），循环体主要操作类似于eax=rsi[eax];ecx+=eax;，循环伪代码如下：

edx=0;

:a

edx++;

eax=rsi[eax];

ecx+=eax;

if(eax==0xf)

goto :a

而array.3095内容查看如下：

(gdb) x/20dw 0x5555555567a0

0x5555555567a0 <array.3095>: 10 2 14 7

0x5555555567b0 <array.3095+16>: 8 12 15 11

0x5555555567c0 <array.3095+32>: 0 4 1 13

0x5555555567d0 <array.3095+48>: 3 9 6 5

0x5555555567e0: 2032168787 1948284271 1802398056 1970239776

可见为一个16个元素的数组（最后4个数已经超出范围）

循环结束后进行判断，条件为%edx等于0xf，即循环运行15次，且%ecx为输入的第二个数，才能过关。为了使循环运行15次，查看array.3095可见要输入的第一个值为5，运行过程中eax的值为5,12,7,3,11,13,9,4,8,0,10,1,2,14,6,15，第二个值在循环结束时从调试器看到应为115，输入5 115即可过关。

**Phase\_6**

Phase\_6开头调用了read\_six\_numbers，读入6个数存在栈上，以%r12为其头指针。假设输入的值为4 3 5 2 6 1。之后进入一个二层循环，循环变量分别为%r13d，范围0-6；%ebx，范围%r13d-5。外循环每次将%r12加四，即头指针指向下一个数，而后内循环判断外循环对应的数（%r12处地址的值）是否与其后面的值（即输入的6个值中后面的、外循环未走过值）不等，如果相等则爆炸。最终结果是要求6个值都不相等。而外循环还有各个值减一之后<=5的要求，于是输入的6个值应该是1到6的一个排列。

完成输入检查后是另一个二层循环，外侧循环变量为%rsi，从0x0开始，步长0x4，结束0x18，循环6次。第i次循环中先取出输入的第i个数，如果这个数是1则跳过内层循环，如果不是1则进入内层循环；内层循环%eax为循环变量，从1开始到%ecx结束，而%ecx为输入的第i个数：

#loop : %eax 0x1:0x1:%ecx

0x00005555555554f7 <+123>: mov 0x8(%rdx),%rdx

0x00005555555554fb <+127>: add $0x1,%eax

0x00005555555554fe <+130>: cmp %ecx,%eax

0x0000555555555500 <+132>: jne 0x5555555554f7 <phase\_6+123>

其作用为将%rdx+0x8地址处的值（还是一个地址）赋值给%rdx，如同取链表的下一个元素。%rdx初值一直是0x5555557582e0<node1>：

0x00005555555554eb <+111>: lea 0x202dee(%rip),%rdx # 0x5555557582e0<node1>

所以内层循环是将%rdx移动到合适的位置，然后在0x0000555555555502 <+134>: mov %rdx,(%rsp,%rsi,2)

外循环最后将rdx存储起来：

0x0000555555555502 <+134>: mov %rdx,(%rsp,%rsi,2)

这个循环完成后内存从node1开始的状态为：

0x5555557582e0 <node1>: 0x000001ea 0x00000001 0x557582f0 0x00005555

0x5555557582f0 <node2>: 0x0000015d 0x00000002 0x55758300 0x00005555

0x555555758300 <node3>: 0x000002f3 0x00000003 0x55758310 0x00005555

0x555555758310 <node4>: 0x00000299 0x00000004 0x55758320 0x00005555

0x555555758320 <node5>: 0x00000188 0x00000005 0x557581f0 0x00005555

以及node6的状态：

0x5555557581f0 <node6>: 0x000001b9 0x00000006 0x00000000 0x00000000

每个node为一个结构体，包含了0到5的数值和指向下一个node的地址。可见在分配内存时node1到node5被分到了连续的一段，而node6没有和它们连续。

栈的状态为：

0x7fffffffd750: 0x55758310 0x00005555 0x55758300 0x00005555

0x7fffffffd760: 0x55758320 0x00005555 0x557582f0 0x00005555

0x7fffffffd770: 0x557581f0 0x00005555 0x557582e0 0x00005555

0x7fffffffd780: 0x00000004 0x00000003 0x00000005 0x00000002

0x7fffffffd790: 0x00000006 0x00000001

其中%rsp+30之后是输入的6个数，每个数占4字节，前面是6个地址，每个占8字节，分别指向6个node结构体，顺序为输入的顺序，比如第一个值0x0000555555758310指向node4，因为第一个输入的数为4。

之后将栈指针以及其处的值赋值到其他寄存器后又是一个循环，循环结束的条件为%rax与%rsi相等，而%rsi之前被赋值为栈上最后一个结构体的地址，如上面栈则该地址为0x7fffffffd778，指向node的地址为0x00005555557582e0。循环的主要内容为借用%rdx为临时寄存器将链表6个元素的指针指向改变了：

#nodes before

0x5555557582e0 <node1>: 0x000001ea 0x00000001 0x557582f0 0x00005555

0x5555557582f0 <node2>: 0x0000015d 0x00000002 0x55758300 0x00005555

0x555555758300 <node3>: 0x000002f3 0x00000003 0x55758310 0x00005555

0x555555758310 <node4>: 0x00000299 0x00000004 0x55758320 0x00005555

0x555555758320 <node5>: 0x00000188 0x00000005 0x557581f0 0x00005555

0x5555557581f0 <node6>: 0x000001b9 0x00000006 0x00000000 0x00000000

#nodes after

0x5555557582e0 <node1>: 0x000001ea 0x00000001 0x00000000 0x00005555

0x5555557582f0 <node2>: 0x0000015d 0x00000002 0x557581f0 0x00005555

0x555555758300 <node3>: 0x000002f3 0x00000003 0x55758320 0x00005555

0x555555758310 <node4>: 0x00000299 0x00000004 0x55758300 0x00005555

0x555555758320 <node5>: 0x00000188 0x00000005 0x557582f0 0x00005555

0x5555557581f0 <node6>: 0x000001b9 0x00000006 0x557582e0 0x00005555

指向顺序有1 2 3 4 5 6变为4 3 5 2 6 1，即输入的值顺序。可见该循环的作用为按输入数字的顺序重新排列链表，而将最后一个元素的next指针置0（NULL）是循环结束后的一条指令做的：

0x0000555555555533 <+183>: movq $0x0,0x8(%rdx)

然后是最后一个循环。 %ebp为循环变量，初始为5，每次减一，到0则结束并成功。循环每次%rbx->0x8(%rbx)，%rbx开始为链表头指针。而循环中进行比较的是结构体中第一个区域，即0x1ea，0x15d等这个值，并要求链表后一个数值要比前一个值小，否则爆炸。

于是phase\_6的要求为：输入一组顺序值，使得链表中的一组数值按递减顺序排列，这组数值为：

0x000001ea

0x0000015d

0x000002f3

0x00000299

0x00000188

0x000001b9

于是输入的值应为3 4 1 6 5 2。

至此，二进制炸弹6个phase结束。

**secret\_phase**

从实验的readme信息中，知还有一个secret phase存在，而在每次defuse一个关卡后调用的phase\_defused函数中，可以看到该phase的入口：

0x00005555555559ca <+127>: callq 0x5555555555a4 <secret\_phase>

查看phase\_defused函数。首先是比较全局比例num\_input\_strings，如果为0x6则进一步行动，否则跳过返回，即最后一次运行defuse时才行动，前几次均直接返回。最后一次运行了callq调用sscanf函数，参数为%d %d %s，如果输入的值是3个，则向下跳转，如果不是，则输出（puts）：

0x555555556878: "Congratulations! You've defused the bomb!"

然后程序结束。否则比较输入的%s和：

0x555555556942: "DrEvil"

如果相等则依次输出：

(gdb) x/1s 0x555555556818

0x555555556818: "Curses, you've found the secret phase!"

(gdb) x/1s 0x555555556840

0x555555556840: "But finding it and solving it are quite different..."

(gdb)

然后进入secret\_phase：

0x00005555555559ca <+127>: callq 0x5555555555a4 <secret\_phase>

但sscanf输入的是什么？查看调用之前的%rdi处地址值发现是”4 19”，即第四关输入的字符串：

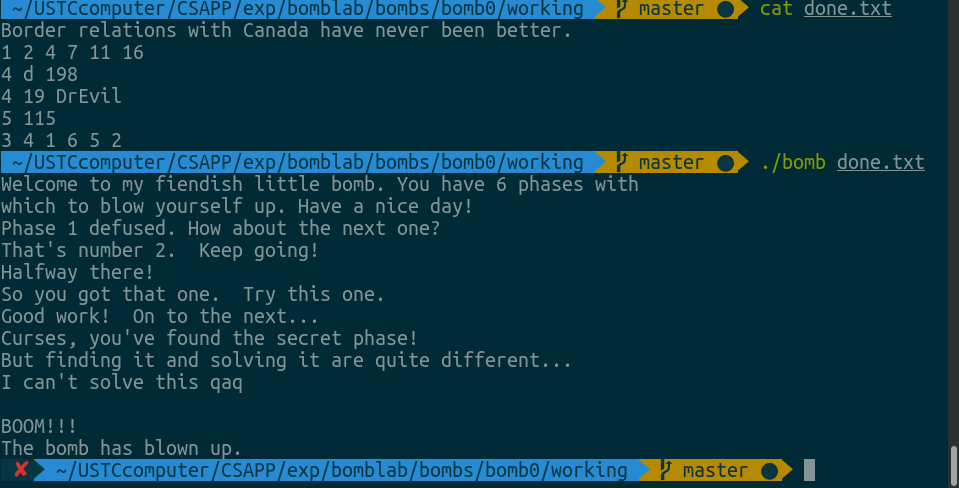
(gdb) x/1s 0x555555758870

0x555555758870 <input\_strings+240>: "4 19"

但因为只有2个值，正常情况下不会进入secret\_phase。于是在第四关输入时加入一个DrEvil字串，则成功进入secret\_phase。

因时间和能力关系，secret\_phase的破解未能完成。

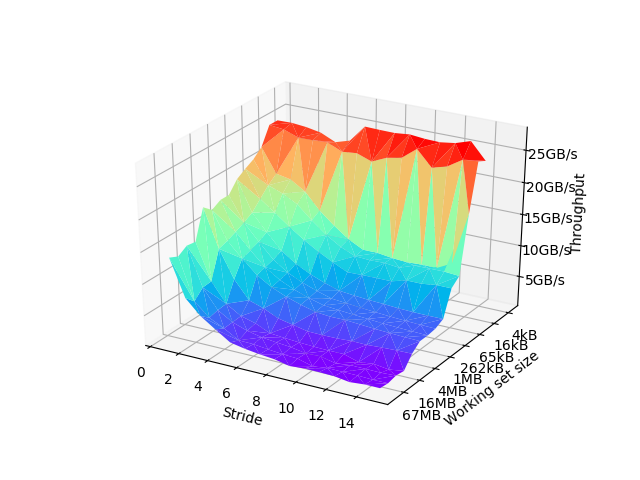
最终结果：



第二部分：课程心得体会

经过计算机系统详解课程的学习，我体会到了仅有上学期的C语言课程对写出好的代码是远远不够的，好的代码既要逻辑正确，又要有高的效率。因为代码是在系统软、硬件上运行的，所以想写好代码必须对运行代码的系统有一定了解，比如说内存管理、异常处理、系统调用；对编译代码的编译器能做的优化也要了解，比如说像函数调用、内存访问等操作的优化是编译器力不能及的；而存储器层次结构更是优化的重点，充分利用空间、时间的局部性能极大提升程序效率；至于CPU的流水线深度等优化，则更为复杂，感觉比较专业，一般是想不到也做不到的。我回想起以前写代码时只有最基本的优化意识，而对于很多可以优化的方法都没有任何概念。虽然我是物理专业的学生，但编程能力还是非常重要的，而对于实验产生的大规模数据的处理还是很依赖C/C++一类运行效率高的语言，而看到经过不同层次的优化后代码运行效率能有数十倍的提高也是我从未见过的。

给我留下印象很深的内容比如说反应存储器层次结构的存储器山，以前我并不了解除了内存、硬盘之外还有多级的cache。而可视化的结果也实实在在地反应了时间、空间的局部性速度差距有多大，这些都会直接影响程序运行效率。如图在我电脑上运行得到的为Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz处理器的存储器山，三级cache分别为256K/1M/8M。



而历史上有名的缓冲区溢出问题我也是第一次听说，以前只知道scanf等函数可能不安全但完全不知道为什么，看到了实实在在的攻击效果和各种编译器的如stack canary等预防措施等确实让人“脑洞大开”，我也意识到很多我以前写的代码都有这个安全隐患，而我并未发现。

程序的机器级表示，即汇编代码也让我对理解程序有了更深的认识，尤其是对“程序员杀手”C语言指针的理解。而二进制炸弹实验极大地锻炼了gdb调试器的使用，克服了在此之前我对调试器的“畏惧”。但实验中也有一部分代码我并未真正理解，是看者gdb给出的内存值来“猜测”一段代码是干什么的，可见我的逻辑能力和对代码的掌握仍需提高。

作为Linux初级用户，我对Linux的认识也有很大提升，比如说进程管理、信号量、异常控制流、虚拟内存、shell以及编程中用到的fork和exec系统调用。尤其是虚拟内存是计算机系统中的重要思想，也是比较难于理解的部分，内存寻址的页表查询、缺页异常处理、多级页表优化、用TLB加速访问等一系列多级设计反应了计算机的基本思想。多进程编程利用fork和exec以及进程号进行控制，也是很有趣的内容。

总之，这个课程开拓了我的视野，加深了我对计算机的理解，对我编写代码、优化代码的能力有一定提升，我学到了以前一直想学又没能下定决心学习的内容，而进行各种实验和在系统中探索也是一大乐趣。但我仍然有很多内容未能透彻理解，而系统方面的编程也从未自己做过，还需要不断学习不断提升，我也希望有时间是完成CSAPP的其他实验内容，如缓冲区溢出、shell的编写等。