# CSAPP的二进制炸弹实验

原创

the_truth_i 于 2018-06-13 11:13:55 发布 
分类专栏： CSAPP 文章标签： CSAPP



最近在看Coursera上的软硬件接口，学习CSAPP，一直都听说这个二进制炸弹实验非常有趣，跟着课程设置就自己动手做了做。打算把实验的过程和结果都纪录一下。

熟悉Linux系统确实花了一番功夫。大二的渣渣表示开始真的好难啊。。。

系统是用的Coursera的课程里面提供的VM+Fedora 64位。

课程附带的VM安装和下载：

https://class.coursera.org/hwswinterface-002/wiki/VirtualMachine

题目提供了一长串的instructions。大意就是要用gdb的反汇编功能查出6个关卡的过关指令？待会一关一关写好了。

开始很没有头绪的时候参考了一下这篇博文http://www.cnblogs.com/remlostime/archive/2011/05/21/2052708.html

也是非常感谢。不过自己做的过程中发现似乎每个人的过关指令都不一样？要是有人也做的话欢迎和我交流一下

根据提示用objdump -d bomb指令把所有函数的汇编代码打印了一下，然后发现有phase_1~6，就是这6个了。

========phase_1===========

```
Dump of assembler code for function phase_1:
   0x0000000000400e70 <+0>:     sub    $0x8,%rsp
   0x0000000000400e74 <+4>:     mov    $0x401af8,%esi
   0x0000000000400e79 <+9>:     callq  0x40123d <strings_not_equal>
   0x0000000000400e7e <+14>:    test   %eax,%eax
   0x0000000000400e80 <+16>:    je     0x400e87 <phase_1+23>
   0x0000000000400e82 <+18>:    callq  0x40163d <explode_bomb>
   0x0000000000400e87 <+23>:    add    $0x8,%rsp
   0x0000000000400e8b <+27>:    retq
End of assembler dump.
```

在0x400e74处设置一个断点，地址0x401af8里面存储的就是预设好的字符串。

后面的test指令用来检测strings_not_equal的返回值。后来仔细查了一下才知道函数的返回值基本都是放在eax里面的。

```
(gdb) p (char *) 0x401af8
$9 = 0x401af8 "Science isn't about why, it's about why not?"
```

然后把这段话打进去吧~

```
(gdb) c
Continuing.
Phase 1 defused. How about the next one?
```

========phase_2==========

```
Dump of assembler code for function phase_2:
   0x0000000000400e8c <+0>:     mov    %rbx,-0x20(%rsp)
   0x0000000000400e91 <+5>:     mov    %rbp,-0x18(%rsp)
   0x0000000000400e96 <+10>:    mov    %r12,-0x10(%rsp)
   0x0000000000400e9b <+15>:    mov    %r13,-0x8(%rsp)
   0x0000000000400ea0 <+20>:    sub    $0x48,%rsp
   0x0000000000400ea4 <+24>:    mov    %rsp,%rsi
   0x0000000000400ea7 <+27>:    callq  0x401743 <read_six_numbers>
   0x0000000000400eac <+32>:    mov    %rsp,%rbp
   0x0000000000400eaf <+35>:    lea    0xc(%rsp),%r13
   0x0000000000400eb4 <+40>:    mov    $0x0,%r12d
   0x0000000000400eba <+46>:    mov    %rbp,%rbx
   0x0000000000400ebd <+49>:    mov    0xc(%rbp),%eax
   0x0000000000400ec0 <+52>:    cmp    %eax,0x0(%rbp)
   0x0000000000400ec3 <+55>:    je     0x400eca <phase_2+62>
   0x0000000000400ec5 <+57>:    callq  0x40163d <explode_bomb>
   0x0000000000400eca <+62>:    add    (%rbx),%r12d
   0x0000000000400ecd <+65>:    add    $0x4,%rbp
   0x0000000000400ed1 <+69>:    cmp    %r13,%rbp
   0x0000000000400ed4 <+72>:    jne    0x400eba <phase_2+46>
   0x0000000000400ed6 <+74>:    test   %r12d,%r12d
   0x0000000000400ed9 <+77>:    jne    0x400ee0 <phase_2+84>
   0x0000000000400edb <+79>:    callq  0x40163d <explode_bomb>
---Type <return> to continue, or q <return> to quit---
   0x0000000000400ee0 <+84>:    mov    0x28(%rsp),%rbx
   0x0000000000400ee5 <+89>:    mov    0x30(%rsp),%rbp
   0x0000000000400eea <+94>:    mov    0x38(%rsp),%r12
   0x0000000000400eef <+99>:    mov    0x40(%rsp),%r13
   0x0000000000400ef4 <+104>:   add    $0x48,%rsp
   0x0000000000400ef8 <+108>:   retq
End of assembler dump.
```

汇编代码如上。还挺长的。。其实只要关注关键的cmp指令就可以了。

```
  0x0000000000400ea4 <+24>:    mov    %rsp,%rsi
  0x0000000000400ea7 <+27>:    callq  0x401743 <read_six_numbers>
  0x0000000000400eac <+32>:    mov    %rsp,%rbp
```

这个函数的名称说明的很明显了。输入6个数字。

经过测试可以发现 rbp里面存放的是6个数的数组的首地址。

比较有趣的是

```
  0x0000000000400ebd <+49>:    mov    0xc(%rbp),%eax
  0x0000000000400ec0 <+52>:    cmp    %eax,0x0(%rbp)
```

之后rbp +0xc是第4个数的地址，这个数被存在了eax里面然后和(rbp)里面的数进行了比较。

再看之后的一个比较

```
  0x0000000000400ecd <+65>:    add    $0x4,%rbp
  0x0000000000400ed1 <+69>:    cmp    %r13,%rbp
```

rbp+0x4是下一个数字的地址，这个地址又被存储在了rbp里面，r13里面存储的又是什么呢？

```
  0x0000000000400eaf <+35>:    lea    0xc(%rsp),%r13
```

从这一行得知r13里面存储的就是第4个数的地址。

所以这个循环比较的是第1和4,2和5,3和6是否相同。

再看下一个比较

```
  0x0000000000400ed6 <+74>:    test   %r12d,%r12d
  0x0000000000400ed9 <+77>:    jne    0x400ee0 <phase_2+84>
  0x0000000000400edb <+79>:    callq  0x40163d <explode_bomb>
```

测试r12d里面的值是否为0,0的话炸弹还是会爆炸。所以往上看看r12d里面究竟是什么

```
  0x0000000000400eb4 <+40>:    mov    $0x0,%r12d
```

```
 0x0000000000400eca <+62>:    add    (%rbx),%r12d
```

这2行可以看出每次循环r12d都会加上(rbx)的值，而rbx里面存储的地址就是rbp里面的地址。

所以r12d里面最后的值就是第1,2,3个数的总和。这样的话只要总和不是0就能过了。

```
Phase 1 defused. How about the next one?
2 4 5 2 4 5
```

用 info register指令看一下猜的对不对。

```
(gdb) i r
rax            0x5      5
rbx            0x7fffffffe5e8   140737488348648
rcx            0x7fffffffe5d0   140737488348624
rdx            0x31a43b0f98     213208731544
rsi            0x0      0
rdi            0xfdf    4063
rbp            0x7fffffffe5ec   0x7fffffffe5ec
rsp            0x7fffffffe5e0   0x7fffffffe5e0
r8             0x1      1
r9             0x0      0
r10            0x5      5
r11            0x0      0
r12            0xb      11
r13            0x7fffffffe5ec   140737488348652
r14            0x0      0
r15            0x0      0
rip            0x400ed6 0x400ed6 <phase_2+74>
eflags         0x246    [ PF ZF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
---Type <return> to continue, or q <return> to quit---
```

r12b里面的值是11说明猜对了。所以第二关也过了~

最开始并没有太仔细的看明白代码，所以输了6个1，也照样过了~

========phase_3===========

开始第三关~

```
Dump of assembler code for function phase_3:
   0x0000000000400ef9 <+0>:     sub    $0x18,%rsp
   0x0000000000400efd <+4>:     lea    0x8(%rsp),%rcx
   0x0000000000400f02 <+9>:     lea    0xc(%rsp),%rdx
   0x0000000000400f07 <+14>:    mov    $0x401ebe,%esi
   0x0000000000400f0c <+19>:    mov    $0x0,%eax
   0x0000000000400f11 <+24>:    callq  0x400ab0 <__isoc99_sscanf@plt>
   0x0000000000400f16 <+29>:    cmp    $0x1,%eax
   0x0000000000400f19 <+32>:    jg     0x400f20 <phase_3+39>
   0x0000000000400f1b <+34>:    callq  0x40163d <explode_bomb>
   0x0000000000400f20 <+39>:    cmpl   $0x7,0xc(%rsp)
   0x0000000000400f25 <+44>:    ja     0x400f63 <phase_3+106>
   0x0000000000400f27 <+46>:    mov    0xc(%rsp),%eax
   0x0000000000400f2b <+50>:    jmpq   *0x401b60(,%rax,8)
   0x0000000000400f32 <+57>:    mov    $0x217,%eax
   0x0000000000400f37 <+62>:    jmp    0x400f74 <phase_3+123>
   0x0000000000400f39 <+64>:    mov    $0xd6,%eax
   0x0000000000400f3e <+69>:    jmp    0x400f74 <phase_3+123>
   0x0000000000400f40 <+71>:    mov    $0x153,%eax
   0x0000000000400f45 <+76>:    jmp    0x400f74 <phase_3+123>
   0x0000000000400f47 <+78>:    mov    $0x77,%eax
   0x0000000000400f4c <+83>:    jmp    0x400f74 <phase_3+123>
   0x0000000000400f4e <+85>:    mov    $0x160,%eax
---Type <return> to continue, or q <return> to quit---
   0x0000000000400f53 <+90>:    jmp    0x400f74 <phase_3+123>
   0x0000000000400f55 <+92>:    mov    $0x397,%eax
   0x0000000000400f5a <+97>:    jmp    0x400f74 <phase_3+123>
   0x0000000000400f5c <+99>:    mov    $0x19c,%eax
   0x0000000000400f61 <+104>:   jmp    0x400f74 <phase_3+123>
   0x0000000000400f63 <+106>:   callq  0x40163d <explode_bomb>
   0x0000000000400f68 <+111>:   mov    $0x0,%eax
```

```
   0x0000000000400f6d <+116>:   jmp    0x400f74 <phase_3+123>
   0x0000000000400f6f <+118>:   mov    $0x39e,%eax
   0x0000000000400f74 <+123>:   cmp    0x8(%rsp),%eax
   0x0000000000400f78 <+127>:   je     0x400f7f <phase_3+134>
   0x0000000000400f7a <+129>:   callq  0x40163d <explode_bomb>
   0x0000000000400f7f <+134>:   add    $0x18,%rsp
   0x0000000000400f83 <+138>:   retq
End of assembler dump.
```

又是好长的汇编代码。呼。

查了一下

```
0x0000000000400f11 <+24>:    callq  0x400ab0 <__isoc99_sscanf@plt>
0x0000000000400f16 <+29>:    cmp    $0x1,%eax
```

这个isoc99_sscanf用法和scanf一样，成功读入多少个数返回值就会是多少

不过很明显就是一个switch语句

```
=> 0x0000000000400f2b <+50>:    jmpq   *0x401b60(,%rax,8)
```

这一行代码就是switch的跳转语句

rax里面存储的就是输入的第一个数。

仔细看下面的汇编代码，可以看到每个switch分支都会把一个值存储在eax里面然后跳转到0x400f74，然后进行rsp+0x8和eax的比较。那只要输入的第二个数等于存储在eax里面的值就可以了。

```
  0x0000000000400f20 <+39>:    cmpl   $0x7,0xc(%rsp)
  0x0000000000400f25 <+44>:    ja     0x400f63 <phase_3+106>
```

这两行可以看到要是输入的数超过7的话炸弹会直接爆炸。

```
0x0000000000400f55 <+92>:    mov    $0x397,%eax
0x0000000000400f5a <+97>:    jmp    0x400f74 <phase_3+123>
```

我选的是第六个数，397是十六进制的，换成10进制是919。

输入6 919。

```
(gdb) c
Continuing.
Halfway there!
```

正确。继续下一关。

========phase_4===========

汇编代码如下

```
Dump of assembler code for function phase_4:
   0x0000000000400fc1 <+0>:     sub    $0x18,%rsp
   0x0000000000400fc5 <+4>:     lea    0xc(%rsp),%rdx
   0x0000000000400fca <+9>:     mov    $0x401ec1,%esi
   0x0000000000400fcf <+14>:    mov    $0x0,%eax
   0x0000000000400fd4 <+19>:    callq  0x400ab0 <__isoc99_sscanf@plt>
   0x0000000000400fd9 <+24>:    cmp    $0x1,%eax
   0x0000000000400fdc <+27>:    jne    0x400fe5 <phase_4+36>
   0x0000000000400fde <+29>:    cmpl   $0x0,0xc(%rsp)
   0x0000000000400fe3 <+34>:    jg     0x400fea <phase_4+41>
   0x0000000000400fe5 <+36>:    callq  0x40163d <explode_bomb>
   0x0000000000400fea <+41>:    mov    0xc(%rsp),%edi
=> 0x0000000000400fee <+45>:    callq  0x400f84 <func4>
   0x0000000000400ff3 <+50>:    cmp    $0x37,%eax
   0x0000000000400ff6 <+53>:    je     0x400ffd <phase_4+60>
   0x0000000000400ff8 <+55>:    callq  0x40163d <explode_bomb>
   0x0000000000400ffd <+60>:    add    $0x18,%rsp
   0x0000000000401001 <+64>:    retq
End of assembler dump.
```

输入一个数之后可以gdb调试一下发现这个数被存储在了($rsp+0xc)处，然后这个数被存在了$edi里面之后作为参数放进了func4进行了运算，最后的结果是0x37也就是十进制的55。也就是说输入一个数经过func4的运算之后要等于55。

下面仔细看一下func4

```
Dump of assembler code for function func4:
=> 0x0000000000400f84 <+0>:     mov    %rbx,-0x10(%rsp)
   0x0000000000400f89 <+5>:     mov    %rbp,-0x8(%rsp)
   0x0000000000400f8e <+10>:    sub    $0x18,%rsp
   0x0000000000400f92 <+14>:    mov    %edi,%ebx
   0x0000000000400f94 <+16>:    mov    $0x1,%eax
   0x0000000000400f99 <+21>:    cmp    $0x1,%edi
   0x0000000000400f9c <+24>:    jle    0x400fb2 <func4+46>
   0x0000000000400f9e <+26>:    lea    -0x1(%rbx),%edi
   0x0000000000400fa1 <+29>:    callq  0x400f84 <func4>
   0x0000000000400fa6 <+34>:    mov    %eax,%ebp
   0x0000000000400fa8 <+36>:    lea    -0x2(%rbx),%edi
   0x0000000000400fab <+39>:    callq  0x400f84 <func4>
   0x0000000000400fb0 <+44>:    add    %ebp,%eax
   0x0000000000400fb2 <+46>:    mov    0x8(%rsp),%rbx
   0x0000000000400fb7 <+51>:    mov    0x10(%rsp),%rbp
   0x0000000000400fbc <+56>:    add    $0x18,%rsp
   0x0000000000400fc0 <+60>:    retq
End of assembler dump.
```

```
0x0000000000400f92 <+14>:    mov    %edi,%ebx
0x0000000000400f94 <+16>:    mov    $0x1,%eax
0x0000000000400f99 <+21>:    cmp    $0x1,%edi
0x0000000000400f9c <+24>:    jle    0x400fb2 <func4+46>
```

这几行可以看出如果edi这个参数小于等于1的话返回值就是1。

```
  0x0000000000400f9c <+24>:    jle    0x400fb2 <func4+46>
  0x0000000000400f9e <+26>:    lea    -0x1(%rbx),%edi
  0x0000000000400fa1 <+29>:    callq  0x400f84 <func4>
  0x0000000000400fa6 <+34>:    mov    %eax,%ebp
  0x0000000000400fa8 <+36>:    lea    -0x2(%rbx),%edi
  0x0000000000400fab <+39>:    callq  0x400f84 <func4>
```

然后是这几行。其实已经很明显了。。

分别计算func4(edi-1)和func4(edi-2)的值之后加到eax里面。

func4的作用是计算斐波那契数列。不过是不过少了开头的一个1。

数一数就知道了n=9的时候正好func4(n)=55。

输入 9

```
(gdb) c
Continuing.
So you got that one.  Try this one.
```

========phase_5===========

```
   0x0000000000401002 <+0>:     sub    $0x18,%rsp
   0x0000000000401006 <+4>:     lea    0x8(%rsp),%rcx
   0x000000000040100b <+9>:     lea    0xc(%rsp),%rdx
   0x0000000000401010 <+14>:    mov    $0x401ebe,%esi
   0x0000000000401015 <+19>:    mov    $0x0,%eax
   0x000000000040101a <+24>:    callq  0x400ab0 <__isoc99_sscanf@plt>
   0x000000000040101f <+29>:    cmp    $0x1,%eax
   0x0000000000401022 <+32>:    jg     0x401029 <phase_5+39>
   0x0000000000401024 <+34>:    callq  0x40163d <explode_bomb>
   0x0000000000401029 <+39>:    mov    0xc(%rsp),%eax
   0x000000000040102d <+43>:    and    $0xf,%eax
   0x0000000000401030 <+46>:    mov    %eax,0xc(%rsp)
   0x0000000000401034 <+50>:    cmp    $0xf,%eax
   0x0000000000401037 <+53>:    je     0x401065 <phase_5+99>
   0x0000000000401039 <+55>:    mov    $0x0,%ecx
   0x000000000040103e <+60>:    mov    $0x0,%edx
   0x0000000000401043 <+65>:    add    $0x1,%edx
   0x0000000000401046 <+68>:    cltq
   0x0000000000401048 <+70>:    mov    0x401ba0(,%rax,4),%eax
   0x000000000040104f <+77>:    add    %eax,%ecx
   0x0000000000401051 <+79>:    cmp    $0xf,%eax
   0x0000000000401054 <+82>:    jne    0x401043 <phase_5+65>
---Type <return> to continue, or q <return> to quit---
   0x0000000000401056 <+84>:    mov    %eax,0xc(%rsp)
   0x000000000040105a <+88>:    cmp    $0xc,%edx
   0x000000000040105d <+91>:    jne    0x401065 <phase_5+99>
   0x000000000040105f <+93>:    cmp    0x8(%rsp),%ecx
   0x0000000000401063 <+97>:    je     0x40106a <phase_5+104>
   0x0000000000401065 <+99>:    callq  0x40163d <explode_bomb>
   0x000000000040106a <+104>:   add    $0x18,%rsp
   0x000000000040106e <+108>:   retq
```

第五个按照提示是字符串和指针。

```
 0x000000000040105a <+88>:    cmp    $0xc,%edx
 0x000000000040105d <+91>:    jne    0x401065 <phase_5+99>
 0x000000000040105f <+93>:    cmp    0x8(%rsp),%ecx
 0x0000000000401063 <+97>:    je     0x40106a <phase_5+104>
```

关键的比较是这几行。

```
 0x000000000040103e <+60>:    mov    $0x0,%edx
 0x0000000000401043 <+65>:    add    $0x1,%edx
```

这两行可以猜测出来edx是用来记录循环次数的值的。从上面可以看出来edx最后要等于12，所以循环里面的运算一共进行了12次。

```
 0x0000000000401048 <+70>:    mov    0x401ba0(,%rax,4),%eax
 0x000000000040104f <+77>:    add    %eax,%ecx
```

这两行可以看出ecx会用来记录循环中的数的和。而eax就是数组的下标。

每次进行的运算都是eax=array[eax]

```
 0x0000000000401029 <+39>:    mov    0xc(%rsp),%eax
 0x000000000040102d <+43>:    and    $0xf,%eax
```

这两行看出，很明显这个数组一共不会超过16个数。输入的第一个数是第一个数组下标。

用gdb把这个数组打印出来看看就可以了。

```
(gdb) p *0x401ba0@16
$11 = {10, 2, 14, 7, 8, 12, 15, 11, 0, 4, 1, 13, 3, 9, 6, 5}
```

```
 0x0000000000401051 <+79>:    cmp    $0xf,%eax
```

可以看到eax=15的是这个循环就会停止。

那只要从下标15开始往前数12个数就可以得到开始的下标了，然后把这些数加起来就可以得到第二个数了。

最后的结果是 7 93

```
(gdb) c
Continuing.
Congratulations! You've (mostly) defused the bomb!
Hit Control-C to escape phase 6 (for free!), but if you want to
try phase 6 for extra credit, you can continue.  Just beware!
```

做完5个之后其实就可以收工了，不过还是看了看第六个的。

========phase_6===========

```
(gdb) disas phase_6
Dump of assembler code for function phase_6:
   0x00000000004010d9 <+0>:     sub    $0x8,%rsp
   0x00000000004010dd <+4>:     mov    $0xa,%edx
   0x00000000004010e2 <+9>:     mov    $0x0,%esi
   0x00000000004010e7 <+14>:    callq  0x400b80 <strtol@plt>
   0x00000000004010ec <+19>:    mov    %eax,0x20168e(%rip)
   0x00000000004010f2 <+25>:    mov    $0x602780,%edi
   0x00000000004010f7 <+30>:    callq  0x40106f <fun6>
   0x00000000004010fc <+35>:    mov    0x8(%rax),%rax
   0x0000000000401100 <+39>:    mov    0x8(%rax),%rax
   0x0000000000401104 <+43>:    mov    0x8(%rax),%rax
   0x0000000000401108 <+47>:    mov    0x201672(%rip),%edx
   0x000000000040110e <+53>:    cmp    %edx,(%rax)
   0x0000000000401110 <+55>:    je     0x401117 <phase_6+62>
   0x0000000000401112 <+57>:    callq  0x40163d <explode_bomb>
   0x0000000000401117 <+62>:    add    $0x8,%rsp
   0x000000000040111b <+66>:    retq
End of assembler dump.
```

其实我并没有看func6是干了些什么。。

输入1之后在0x40110e处打了个断点看了看这个比较是怎么回事

```
(gdb) p *(int *) $rax
$14 = 600
(gdb) p $edx
$15 = 1
```

就会发现(rax)里面的值是600。

然后只要在开始的时候输入600就能过了~也算偷了个懒吧。

```
(gdb) c
Continuing.
Congratulations! You've defused the bomb! Again!
```

用objdump -d bomb指令可以看出phase_defused函数里面有

call secret_phase可以进入秘密关卡。不够我并没有做这个。

========过关秘籍==========

gdb自带有set指令可以改寄存器的值，所以在做的时候遇到比较可以直接修改寄存器的值可以直接过关。。。这个比较。。呵呵。。

做完之后也是感觉gdb调试功底大涨，好像更有信心了。