

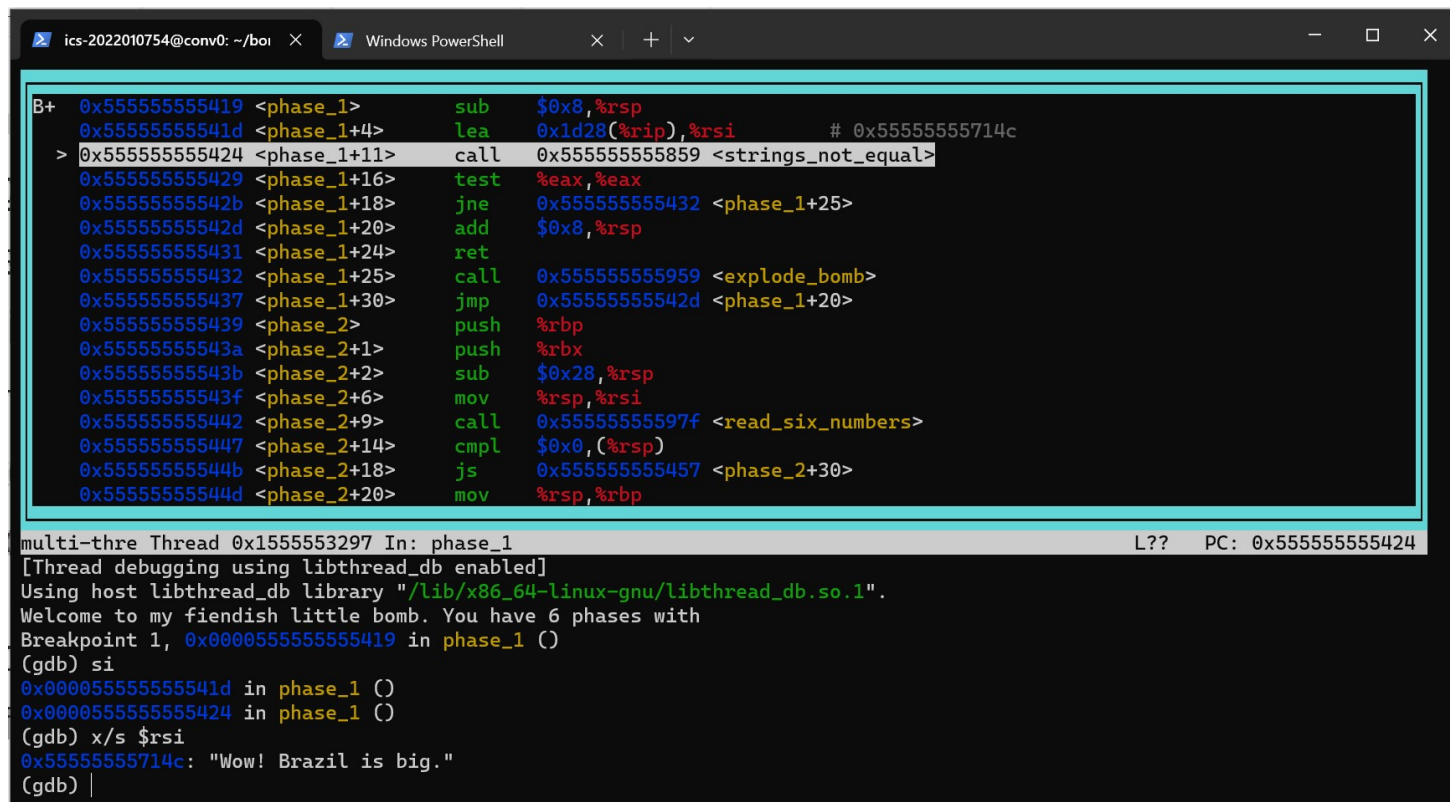
ICS Bomblab

张弛 2022010754 zhang-ch22@mails.tsinghua.edu.cn

Phase 1

首先用 objdump 将可执行文件转为汇编代码，可以见到，在 main 函数中，所有的输入都被储存在 (%rdi) 里，而在调用子函数时，%rdi 会作为第一个传入的参数。

在 phase1 中，函数调用了 strings_not_equal 函数，调用时 %rdi 的值未改变，仍为我们在主函数的输入，而 %rsi 则为 0x1d28(%rip)。如果两个返回值为不同，则引爆炸弹。因此我们只需要通过gdb调试找到 %rsi 的值即可。经过检查，如图：



```
ics-2022010754@conv0: ~/boi x Windows PowerShell
B+ 0x55555555419 <phase_1>      sub    $0x8,%rsp
0x5555555541d <phase_1+4>      lea    0x1d28(%rip),%rsi      # 0x55555555714c
> 0x55555555424 <phase_1+11>   call   0x555555555859 <strings_not_equal>
0x55555555429 <phase_1+16>      test   %eax,%eax
0x5555555542b <phase_1+18>      jne    0x55555555432 <phase_1+25>
0x5555555542d <phase_1+20>      add    $0x8,%rsp
0x55555555431 <phase_1+24>      ret
0x55555555432 <phase_1+25>      call   0x555555555959 <explode_bomb>
0x55555555437 <phase_1+30>      jmp    0x5555555542d <phase_1+20>
0x55555555439 <phase_2>          push   %rbp
0x5555555543a <phase_2+1>        push   %rbx
0x5555555543b <phase_2+2>        sub    $0x28,%rsp
0x5555555543f <phase_2+6>        mov    %rsp,%rsi
0x55555555442 <phase_2+9>        call   0x55555555597f <read_six_numbers>
0x55555555447 <phase_2+14>       cmpl   $0x0,(%rsp)
0x5555555544b <phase_2+18>       js     0x55555555457 <phase_2+30>
0x5555555544d <phase_2+20>       mov    %rsp,%rbp

multi-thre Thread 0x1555553297 In: phase_1 L?? PC: 0x55555555424
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Welcome to my fiendish little bomb. You have 6 phases with
Breakpoint 1, 0x000055555555419 in phase_1 ()
(gdb) si
0x00005555555541d in phase_1 ()
0x000055555555424 in phase_1 ()
(gdb) x/s $rsi
0x55555555714c: "Wow! Brazil is big."
(gdb) |
```

故，我们只需输入"Wow! Brazil is big."即可。

Phase 2

在 phase2 中，函数先扩大了40个字节的栈空间，把 %rsi 赋为 %rsp，然后调用了 read_six_numbers 函数。此时，%rdi 为我们的输入的地址，%rsi 为 %rsp。

`read_six_numbers` 函数把 `%rdx` , `%rcx` , `%r8` , `%r9` , 和栈上的两个元素分别取为 `%rsi` 所指向的地址的后若干位置。由于 `%rsi` 为 `%rsp` , 因此 `%rdx` , `%rcx` , `%r8` , `%r9` , 和栈上的两个元素分别指向调用者栈帧上的六个位置。之后调用了 `__isoc99_sscanf@plt` , 猜测是把我们的输入 (`%rdi` 中) 读入它的参数到 `%rdx` , `%rcx` , `%r8` , `%r9` , 和栈上的两个元素所指向的地址, 并返回元素个数 (从返回数小于5则引爆炸弹可以验证)。

返回 phase2，此时读入的六个数分别在 %rsp 的上面六个位置。之后，函数进行循环，让 %ebx 从1开始，每次加1，%rbp 指向第一个数，每次加4（指向下一个数），然后检验是否有 $\%ebx + (\%rbp) == (\%rbp + 4)$ ，即6个数 a_1, a_2, \dots, a_6 应该有关系 $a_i + i = a_{i+1}$ 。从而我们输入 "1 2 4 7 11 16"（或其他满足要求的6个数，答案不唯一）即可。

Phase 3

在 phase3 中，函数先扩大了24个字节的栈空间，然后将读入的两个参数分别放在了 `%rsp+12` 和 `%rsp+8` 这两个位置上。随后，函数检查了第一个数小于等于7，否则引爆炸弹。然后函数通过 `(%rsp+12)` 和 `%rdx` 确定了要跳转到的位置 `*%rax`。其中，`%rdx` 存储的内容是设置好的。经过调试可以找到，`%rdx` 的值（此时，输入了一个满足小于等于7的第一位输入"5")：

```

ics-2022010754@conv0: ~/boi x Windows PowerShell x + v
B+ 0x555555552c9 <main>          push    %rbx
    0x55555555482 <phase_3>      sub     $0x18,%rsp
    0x55555555486 <phase_3+4>    lea     0x8(%rsp),%rcx <main+258>
    0x5555555548b <phase_3+9>    lea     0xc(%rsp),%rdx
    0x55555555490 <phase_3+14>   lea     0x1e40(%rip),%rsi # 0x5555555572d7
    0x55555555497 <phase_3+21>   mov     $0x0,%eax b <main+306>
    0x5555555549c <phase_3+26>   call   0x55555555140 <__isoc99_sscanf@plt>
    0x555555554a1 <phase_3+31>   cmp     $0x1,%eax
    0x555555554a4 <phase_3+34>   jle     0x555555554c1 <phase_3+63>
    0x555555554a6 <phase_3+36>   cmpl    $0x7,0xc(%rsp) 0x555555559770 <infile>
    0x555555554ab <phase_3+41>   ja      0x55555555502 <phase_3+128>
    0x555555554ad <phase_3+43>   mov     0xc(%rsp),%eax <main+277>
    0x555555554b1 <phase_3+47>   lea     0x1cb8(%rip),%rdx # 0x555555557170
    0x555555554b8 <phase_3+54>   movslq  (%rdx,%rax,4),%rax
    0x555555554bc <phase_3+58>   add     %rdx,%rax
    > 0x555555554bf <phase_3+61>   jmp     *%rax # 0x5555555570c8
    0x555555554c1 <phase_3+63>   call   0x55555555599 <explode_bomb>
    4c6 <phase_3+68>   jmp     0x555555554a6 <phase_3+36>

exec No process In:
multi-thre Thread 0x1555553297 In: phase_3 L?? PC: ??
0x0000555555554b1 in phase_3 () L?? PC: 0x555555554bf
0x0000555555554b8 in phase_3 ()
0x0000555555554bc in phase_3 ()
0x0000555555554bf in phase_3 ()
(gdb) x/s $rdx
0x555555557170: "X\343\377\377\236\343\377\377h\343\377\377o\343\377\377v\343\377\377}\343\377\377\204\343\377\377\213\3
43\377\377maduiersnfotvbylWow! You've defused the secret stage!"
(gdb) x/s $rax
0x555555554ed <phase_3+107>: "\270\264\002"
(gdb) |

```

可以看到，此时 `*rax` 为 `<phase_3+107>`，转化为16进制是14ed。这里将 `&eax` 赋值为2b4，然后跳转回14cd处判断 `%eax` 与 `%rsp+8` 是否相等，相等则返回，若不等则引爆炸弹。故我们只需要将第二个参数设置为2b4（十进制：692）即可。故输入"5 692"即可。此题答案不唯一。

Phase 4

phase4 的大体逻辑如下：首先扩大栈空间存放输入，这里假设我们的两个输入分别为 a 和 b，phase4 先准备好了一些寄存器数值，使得 %edx=14，%esi=0，%edi=a，然后调用函数 func4。

在 func4 中，函数进行了下面的计算：

```
%ecx = [(%edx - %esi) + sign(%edx - %esi)] / 2 + %esi
```

其中，"/2"操作并不准确，应该是算数右移1位。如果被除数是奇数，那么第一位还需要补一个1。然而接下来的答案里不需要用到这一种情况，所以简单起见写作"/2"。

然后函数判断了计算结果 %ecx 和我们的输入 a 的大小。如果 %ecx 大于 a，则 %edx 被赋值为 %ecx - 1，递归调用 func4 并让 %eax 翻倍。若小于，则 %esi 被赋值为 %ecx + 1，递归调用 func4 并让 %eax 乘2加1。如果相等，则把 %eax 赋值为0，然后返回。

注意到，我们若想最后结束递归返回，必须要让判断器判等才能终止递归。同时，回到 phase4 中，我们可以看到题目要求 %eax 和 b 都为2才能通过，否则引爆炸弹。所以，我们需要构造出一个递归情况使得 %eax 最终变为2。注意到，比较 %ecx 和 b 大小时三种情况里最后一种将 %eax 归0，前两种都只会将 %eax 放大。所以我们最后需要的递归调用逻辑如下：

```
func4:
    ...// 计算%ecx
    %ecx > a: %edx = %ecx - 1
    func4:
        ...// 计算%ecx
        %ecx < a: %edx = %ecx - 1
        ...// 计算%ecx
        func4:
            ...// 计算%ecx
            %ecx == a: %eax = 0
            %eax = 2 * %eax + 1 = 1
        %eax = 2 * %eax = 2
```

这样就能满足 %eax =2的要求。经过计算，得到中间一层的 %ecx 应该为5。所以我们只需要输入"5 2"即可。

Phase 5

phase5 的首先接纳了一个输入，然后调用了 `string_length` 函数，将返回值存入 `%eax` 并与6比较。可以猜测 phase5 接纳一个长度为6的字符串。赋值后，`%rbx` 指向字符串首尾。同时阅读 `string_length` 的代码可以知道，调用完这个函数后本来指向字符串首位的指针 `%rdi` 指向了字符串的末尾。随后，程序将一个字符串的指针存到 `%rcx` 里面，经过调试，可以看到这个字符串为"maduiersnfotvbylYou've defused the secret stage!"。之后程序进行循环，设输入的字符串为 `s`，`%rcx` 存储的字符串为 `t`，则程序依次进行：

```
for i in range(6):  
    x = s[i] & 0xf // 取16进制的最后一位  
    rsp[i+9] = t[x]
```

然后，程序将 `%rsp[i+15]` 赋为0，此时 `%rsp[i+9] ~ %rsp[i+14]` 形成一个6为字符串。程序将这个字符串与 `%rsi` 进行比较，若不等则引爆炸弹。经过调试可以发现，`%rsi` 的值为"oilers"。

所以，我们只需要在 `%rcx` 指向的字符串中依次找到这些字母即可。这六个字母分别在"maduiersnfotvbylYou've defused the secret stage!"的第10 (0xa)、4、15 (0xf)、5、6、7位出现。我们只需要找到六个字母，使得它们的ASCII码的最后一个16进制位分别为A、4、F、5、6、7即可。我在0x61（小写字母"a"）之后的范围内进行寻找，发现字母j、d、o、e、f、g满足要求。于是输入"jdoefg"即可。此题答案不唯一。

附：调试截图

```
ics-2022010754@conv0: ~/boi x + v
0x5555555555c9 <phase_5+23> lea 0x1bc0(%rip),%rcx # 0x555555557190 <array.0>
0x5555555555d0 <phase_5+30> movzbl (%rbx,%rax,1),%edx
0x5555555555d4 <phase_5+34> and $0xf,%edx
0x5555555555d7 <phase_5+37> movzbl (%rcx,%rdx,1),%edx
0x5555555555db <phase_5+41> mov %dl,0x9(%rsp,%rax,1)
0x5555555555df <phase_5+45> add $0x1,%rax
0x5555555555e3 <phase_5+49> cmp $0x6,%rax
0x5555555555e7 <phase_5+53> jne 0x5555555555d0 <phase_5+30>
0x5555555555e9 <phase_5+55> movb $0x0,0xf(%rsp)
0x5555555555ee <phase_5+60> lea 0x9(%rsp),%rdi
0x5555555555f3 <phase_5+65> lea 0x1b66(%rip),%rsi # 0x555555557160
> 0x5555555555fa <phase_5+72> call 0x5555555555859 <strings_not_equal>
0x5555555555ff <phase_5+77> test %eax,%eax
0x555555555601 <phase_5+79> jne 0x555555555610 <phase_5+94>
0x555555555603 <phase_5+81> add $0x10,%rsp
0x555555555607 <phase_5+85> pop %rbx
0x555555555608 <phase_5+86> ret

multi-thre Thread 0x1555553297 In: phase_5 L?? PC: 0x5555555555fa
0x000055555555e7 in phase_5 ()
0x000055555555e9 in phase_5 ()
0x000055555555ee in phase_5 ()
0x000055555555f3 in phase_5 ()
0x000055555555fa in phase_5 ()
(gdb) x/s %rcx
0x555555557190 <array.0>: "maduiersnfotvbylWow! You've defused the secret stage!"
(gdb) x/s %rsi
0x555555557160: "oilers"
(gdb) |
```

Phase 6

phase6 与 phase2 使用相同的函数接受了六个整数作为输入，把输入的6个整数存储到了栈空间 %rsp+48 到 %rsp+72 这个地址区间里，并使用 %rsi、%rbp、%r12、%r14 同时标记第一个数所在位置。然后跳转到了16eb行，进入16c0行到1706行这段代码区间。

在16c0行到1706行这段代码区间里，程序做了这样一件事：依次让 %eax 取便输入的六个整数，%rbp 取遍存储六个整数的地址，然后判断：1. %eax-1 无符号地不大于5，即 %eax 需要在16之间；2.在`%rbp`后面存储的所有数与`%rbp`存储的数不相等。即，程序检查了输入的留个整数都在16范围内且互不相同。如果不满足这两个条件，则引爆炸弹。如果满足，则跳转到164d行，进入164d到167e行这段代码区间。

在这短代码区间里，程序将 %rdx 赋予了一个值，并且不断地将访问 0x8(%rdx)。可以推测，%rdx 存储了一个链表的首地址，然后程序不断地执行访问链表下一个节点的操作。具体来说，对存储的每一个输入的数 i ，程序通过循环 i 次访问了链表的第 i 个节点，并把结果存储在栈中 %rsp+8*(i -1) 的位置上。

执行完167e行后，在1680到16b1行，程序根据栈中内容，重新排列了链表的各个节点，将原来的链接顺序改成栈中从下到上的顺序。

随后，程序跳转到1711行。在这里，程序再次遍历链表，同时检查是否每一个节点存储的数值的后八位都**大于等于**后一个节点的数值的后八位。若否，引爆炸弹。若新链表节点存储的数值依次递减，则结束并返回。

那么我们需要做的就是找到存储在 %rdx 位置的初始链表的数。经过调试发现：

```
ics-2022010754@conv0: ~/boi X + v
0x555555552df <main+22>      mov     0x8(%rsi),%rdi
0x555555555652 <phase_6+59>    mov     0x30(%rsp,%rsi,4),%ecx
0x555555555656 <phase_6+63>      mov     $0x1,%eax
0x55555555565b <phase_6+68>      lea     0x3c8e(%rip),%rdx      # 0x55555555592f0 <node1>
0x555555555662 <phase_6+75>      cmp     $0x1,%ecx
0x555555555665 <phase_6+78>      jle     0x555555555672 <phase_6+91>
0x555555555667 <phase_6+80>      mov     0x8(%rdx),%rdx
0x55555555566b <phase_6+84>      add     $0x1,%eax
0x55555555566e <phase_6+87>      cmp     %ecx,%eax
0x555555555670 <phase_6+89>      jne     0x555555555667 <phase_6+80>
0x555555555672 <phase_6+91>    mov     %rdx,(&rsp,%rsi,8)
> 0x555555555676 <phase_6+95>    add     $0x1,%rsi
0x55555555567a <phase_6+99>      cmp     $0x6,%rsi
0x55555555567e <phase_6+103>     jne     0x555555555652 <phase_6+59>
b+ 0x555555555680 <phase_6+105>    mov     (%rsp),%rbx
0x555555555684 <phase_6+109>    mov     0x8(%rsp),%rax
0x555555555689 <phase_6+114>    mov     %rax,0x8(%rbx)
0x55555555568d <phase_6+118>    mov     0x10(%rsp),%rdx

The program being debugged has been started already.
Start it from Thread 0x1555553297 In: phase_6
(gdb) x/12xg $rdx
0x55555555592f0 <node1>: 0x000000001000001ca      0x00005555555559300
0x5555555559300 <node2>: 0x0000000020000020d      0x00005555555559310
0x5555555559310 <node3>: 0x000000003000002b9      0x00005555555559320
0x5555555559320 <node4>: 0x000000004000001b5      0x00005555555559330
0x5555555559330 <node5>: 0x000000005000003e3      0x000055555555591f0
0x5555555559340 <host_table>: 0x0000555555557331      0x000055555555734b
(gdb) x/xg0x000055555555591f0
0x55555555591f0 <node6>: 0x000000006000002b1
(gdb) |
```

可以看到，链表的前五个节点都存储在一片连续空间里，可以直接查到。而通过查找最后一个地址也能找到第六个节点。比较这些节点存储的数值的后八位，按照大小排序顺序位node5、node3、node6、node2、node1、node4，所以我们输入"5 3 6 2 1 4"即可。

感想

整个实验还是很有趣味的。六个关卡的设计都很精妙，而且难度也相对合理（刚开始做的时候有一些畏惧情绪，但随着做实验逐渐熟悉了汇编语言之后好了很多。而且拆掉炸弹还是很有成就感的（））。通过这次实验，我对汇编语言的熟练度大大增加，能够更快地理解这些汇编代码在干什么了。想吐槽一下的就是如果能够更详细地讲一下gdb的使用方法就好了……在试图使用很多调试功能以及在查内存地址的时候遇到了不少麻烦，不过后来根据网上的资料也解决了。感觉助教们以后可以把常用gdb命令汇总到实验文档里。