BombLab Report By 2019201408陈志朋

Preparation

命令行：反汇编，通过可执行文件生成汇编文件。

```
~/$ objdump -d -S ./bomb > bomb.s
```

命令行：打开带有图形化界面的gdb调试器。

```
~/$ gdb ./bomb -tui
```

gdb：开启对汇编代码的单步调试。

```
(gdb) set disassemble-next-line on
```

gdb：将寄存器的值打印在屏幕上。

```
(gdb) layout regs
```

gdb：防止炸弹爆炸，在调用爆炸函数之前设置断点。

```
(gdb) b explode_bomb
```

Phase 1

> *For NASA, space is still a high priority.*

通过汇编代码可得，该炸弹在读入后，将读入的字符串和内存中的某个字符串进行比较。如果二者相等，则拆弹成功；否则，炸弹引爆。

```
143d:   48 8d 35 0c 1d 00 00    lea    0x1d0c(%rip),%rsi        # 3150 <_IO_stdin_used+0x150>
1444:   e8 b1 04 00 00          callq  18fa <strings_not_equal>
```

调试时，在phase_1的位置设置断点，通过打印指令（p (char *) + 地址）即可得到目标字符串。

Phase 2

> *1 2 4 8 16 32*

```
1472:   e8 59 07 00 00          callq  1bd0 <read_six_numbers>
```

通过汇编代码可以知道，该炸弹需要读入6个32位整数，这6个整数将作为拆弹的密码。通过gdb可知，读入的六个整数依次储存在%rbx，0x4(%rbx)，0x4(%rbx)，0x4(%rbx)，0x4(%rbx)，0x4(%rbx)中。

```
149b:   39 43 04                cmp    %eax,0x4(%rbx)
149e:   74 ee                   je     148e <phase_2+0x35>
14a0:   e8 ef 06 00 00          callq  1b94 <explode_bomb>
```

后面的汇编代码显示，需要将每一个数分别于寄存器%eax中的值进行比较，如果相等，即可避免炸弹爆炸。

接下来需要得到的是寄存器%eax中值的计算方式。有汇编代码得，每次将%eax与%eax相加，即将%eax中之前储存的值乘2。通过其他部分的汇编可得，%eax的初值为1。所以，输入的六个数需要是以1为首项，2为公比的等比数列。

Phase 3

> *1 544*

```
14df:   48 8d 35 1f 1f 00 00    lea    0x1f1f(%rip),%rsi        # 3405 <array.0+0x235>
14e6:   e8 65 fc ff ff          callq  1150 <__isoc99_sscanf@plt>
```

通过打印0x1f1f(%rip)位置的字符串得，该炸弹需要输入两个32位整数，并且依次储存在%rsp，0x4(%rsp)这两个位置上。从汇编代码整体来看，该代码可能是一个switch语句，对于不同的第一个输入，需要对应不同的第二个输入。

```
14f0:   83 3c 24 07             cmpl   $0x7,(%rsp)
14f4:   77 64                   ja     155a <phase_3+0x97>
```

由该段代码得知，输入的第一个数需要小于或等于7。先输入"1 1"，然后通过gdb调试查看第二个输入的要求。

```
1510:   b8 20 02 00 00          mov    $0x220,%eax
1515:   39 44 24 04             cmp    %eax,0x4(%rsp)
1519:   75 52                   jne    156d <phase_3+0xaa>
151b:   48 8b 44 24 08          mov    0x8(%rsp),%rax
1520:   64 48 2b 04 25 28 00    sub    %fs:0x28,%rax
1527:   00 00
1529:   75 49                   jne    1574 <phase_3+0xb1>
152b:   48 83 c4 18             add    $0x18,%rsp
152f:   c3                      retq
```

根据gdb调试器的信息，当第一个数是1的时候，会跳转到偏移量位0x1510的代码（即上图的第一行代码）。在这个分支中，将544（即0x220）赋值到%eax中，然后将%eax和第二个数比较大小，如果二者相等，那么可以避免爆炸，并且结束程序。所以，从这里可以得到该炸弹的一个解：1 544。

通过类似的方法，第一个数可以取$2, 3, \ldots, 7$等其他值，通过gdb调试，可以找到第二个数的取值要求。因此，这个炸弹有多个解。

Phase 4

> *4 19*

```
15c7:   48 8d 35 37 1e 00 00    lea    0x1e37(%rip),%rsi        # 3405 <array.0+0x235>
15ce:   e8 7d fb ff ff          callq  1150 <__isoc99_sscanf@plt>
```

通过打印0x1e37(%rip)位置的字符串得，该炸弹需要输入两个32位整数，并且依次储存在%rsp，0x4(%rsp)这两个位置上。

该炸弹涉及到func4函数。容易发现，这个函数不会对输入的第二个数进行判断和修改，所以猜测这个函数是用于判断第一个数的。

由于func4函数的代码长度较短，可以直接阅读。该函数实现了一个类似二分查找的功能。

通过阅读得知，该函数有三个参数：x, l, r。其中，x为输入的第一个数，l的初值为0，r的初值为14。该函数求l和r的平均值mid（下取整），然后比较平均值和x的大小。如果平均值与x相等，结束程序。否则，如果平均值比x大，那么将r调整为mid-1；如果平均值比x小，那么将l调整为mid+1。每次都将平均值累加到寄存器%eax上。

```
15f0:   e8 84 ff ff ff          callq  1579 <func4>
15f5:   83 f8 13                cmp    $0x13,%eax
15f8:   75 07                   jne    1601 <phase_4+0x56>
```

通过这段汇编得，要求func4中所有平均数的值为19（即0x13），通过计算得，当输入的第一个数为4时，func4函数的知行过程可以由下表表示：

L	R	MID
0	14	7
0	6	3
4	6	5
4	4	4

```
15fa:   83 7c 24 04 13          cmpl   $0x13,0x4(%rsp)
15ff:   74 05                   je     1606 <phase_4+0x5b>
1601:   e8 8e 05 00 00          callq  1b94 <explode_bomb>
```

继续往后阅读汇编代码，由上图的代码可以发现第二个数需要为19（即0x13）。由此可以得到该炸弹的解。

Phase 5

> *ionefg*

```
1671:   48 8d 35 2e 1b 00 00    lea    0x1b2e(%rip),%rsi        # 31a6 <_IO_stdin_used+0x1a6>
1678:   e8 7d 02 00 00          callq  18fa <strings_not_equal>
```

阅读汇编代码发现，该炸弹在最后通过调用strings_not_equal来对输入进行判断，所以输入是一个字符串。同时，打印对应地址的内容，可以知道目标字符串为：flyier。

```
1638:   e8 a0 02 00 00          callq  18dd <string_length>
163d:   83 f8 06                cmp    $0x6,%eax
```

此处的代码调用了string_length，所以输入字符串的长度为6。

```
1642:   b8 00 00 00 00          mov    $0x0,%eax
1647:   48 8d 0d 82 1b 00 00    lea    0x1b82(%rip),%rcx        # 31d0 <array.0>
164e:   0f b6 14 03             movzbl (%rbx,%rax,1),%edx
1652:   83 e2 0f                and    $0xf,%edx
1655:   0f b6 14 11             movzbl (%rcx,%rdx,1),%edx
1659:   88 54 04 01             mov    %dl,0x1(%rsp,%rax,1)
165d:   48 83 c0 01             add    $0x1,%rax
1661:   48 83 f8 06             cmp    $0x6,%rax
1665:   75 e7                   jne    164e <phase_5+0x2e>
```

接下来是汇编的主要部分，在这个部分中，先将%eax赋值为0并且在最后加一且跳转，因此这部分是一个循环，循环变量是%eax。这个循环从0到5，依次处理输入字符串中的每一个字符。

循环体中，发现每次将输入字符的ASCII码和0xf取与后，取出内存地址上对应位置的字符和目标字符进行比较。用过打印该内存地址上的字符串得，该字符串为maduiersnfotvbyl（下标从0开始）。该程序可以理解为**密文解密为明文**的过程。密文'a'对应的明文是'a'，密文'b'对应的明文是'd'等等。依次类推，即可得到该炸弹的解。

Phase 6

> *364521*

```
16cc:   e8 ff 04 00 00          callq  1bd0 <read_six_numbers>
```

输入部分，通过汇编代码可知，该炸弹需要输入6个整数。

这段汇编代码的剩下较为复杂，但是可以将其分为4个部分。

```
16e1:   eb 30                   jmp    1713 <phase_6+0x69>
16e3:   48 83 c3 01             add    $0x1,%rbx
16e7:   83 fb 05                cmp    $0x5,%ebx
16ea:   7f 10                   jg     16fc <phase_6+0x52>
16ec:   41 8b 04 9c             mov    (%r12,%rbx,4),%eax
16f0:   39 45 00                cmp    %eax,0x0(%rbp)
16f3:   75 ee                   jne    16e3 <phase_6+0x39>
16f5:   e8 9a 04 00 00          callq  1b94 <explode_bomb>
16fa:   eb e7                   jmp    16e3 <phase_6+0x39>
16fc:   49 83 c6 01             add    $0x1,%r14
1700:   49 83 c5 04             add    $0x4,%r13
1704:   4c 89 ed                mov    %r13,%rbp
1707:   41 8b 45 00             mov    0x0(%r13),%eax
170b:   83 e8 01                sub    $0x1,%eax
170e:   83 f8 05                cmp    $0x5,%eax
1711:   77 c9                   ja     16dc <phase_6+0x32>
1713:   41 83 fe 05             cmp    $0x5,%r14d
1717:   7f 05                   jg     171e <phase_6+0x74>
1719:   4c 89 f3                mov    %r14,%rbx
171c:   eb ce                   jmp    16ec <phase_6+0x42>
```

这个是主要汇编代码的第一部分。用过各类jmp指令可以大致看出，这是一个两重循环。第一重循环枚举每一个数，第二个循环是枚举其他的数，并且判断这两个数是否相等，如果相等会引起炸弹的爆炸。所以，输入的六个整数需要是六个不同的整数。

```
1723:   8b 0c b4                mov    (%rsp,%rsi,4),%ecx
1726:   b8 01 00 00 00          mov    $0x1,%eax
172b:   48 8d 15 fe 3b 00 00    lea    0x3bfe(%rip),%rdx        # 5330 <node1>
1732:   83 f9 01                cmp    $0x1,%ecx
1735:   7e 0b                   jle    1742 <phase_6+0x98>
1737:   48 8b 52 08             mov    0x8(%rdx),%rdx
173b:   83 c0 01                add    $0x1,%eax
173e:   39 c8                   cmp    %ecx,%eax
1740:   75 f5                   jne    1737 <phase_6+0x8d>
1742:   48 89 54 f4 20          mov    %rdx,0x20(%rsp,%rsi,8)
1747:   48 83 c6 01             add    $0x1,%rsi
174b:   48 83 fe 06             cmp    $0x6,%rsi
174f:   75 d2                   jne    1723 <phase_6+0x79>
```

这个是主要汇编代码的第二部分。该部分是一个循环，第一眼看不懂该循环在干什么。但是，通过阅读可以发现，每次循环会将内存某地址上的进行拷贝。因此，将该内存中的内容打印出来应该会有助于解题。打印结果如下图所示：

```
(gdb) x /24x 0x555555559330
0x555555559330 <node1>: 0x000000ba      0x00000001      0x55559340      0x00005555
0x555555559340 <node2>: 0x000001c3      0x00000002      0x55559350      0x00005555
0x555555559350 <node3>: 0x000003b9      0x00000003      0x55559360      0x00005555
0x555555559360 <node4>: 0x0000031c      0x00000004      0x55559370      0x00005555
0x555555559370 <node5>: 0x000002bb      0x00000005      0x55559210      0x00005555
0x555555559380 <host_table>:    0x5555745f      0x00005555      0x5555746b      0x00005555
(gdb) x /4x 0x555555559210
0x555555559210 <node6>: 0x00000327      0x00000006      0x00000000      0x00000000
(gdb)
```

很容易发现，这个内存中储存了若干个结构体，结构体的第一个元素（32位整数）是该结构体的值，第二个元素（32位整数）是该结构体的id，第三个元素（64位指针）是下一个结构体的地址。那么，这个循环时用于将这6个结构体按照输入的顺序重新排列。

```
1751:   48 8b 5c 24 20          mov    0x20(%rsp),%rbx
1756:   48 8b 44 24 28          mov    0x28(%rsp),%rax
175b:   48 89 43 08             mov    %rax,0x8(%rbx)
175f:   48 8b 54 24 30          mov    0x30(%rsp),%rdx
1764:   48 89 50 08             mov    %rdx,0x8(%rax)
1768:   48 8b 44 24 38          mov    0x38(%rsp),%rax
176d:   48 89 42 08             mov    %rax,0x8(%rdx)
1771:   48 8b 54 24 40          mov    0x40(%rsp),%rdx
1776:   48 89 50 08             mov    %rdx,0x8(%rax)
177a:   48 8b 44 24 48          mov    0x48(%rsp),%rax
177f:   48 89 42 08             mov    %rax,0x8(%rdx)
1783:   48 c7 40 08 00 00 00    movq   $0x0,0x8(%rax)
```

接下来是第三部分。容易发现，这个部分是将之前构造好的结构体进行拷贝，用处不是很大。

```
178b:   bd 05 00 00 00          mov    $0x5,%ebp
1790:   eb 09                   jmp    179b <phase_6+0xf1>
1792:   48 8b 5b 08             mov    0x8(%rbx),%rbx
1796:   83 ed 01                sub    $0x1,%ebp
1799:   74 11                   je     17ac <phase_6+0x102>
179b:   48 8b 43 08             mov    0x8(%rbx),%rax
179f:   8b 00                   mov    (%rax),%eax
17a1:   39 03                   cmp    %eax,(%rbx)
17a3:   7d ed                   jge    1792 <phase_6+0xe8>
```

接下来的最后一个部分，是一个循环，需要每个数比链表上的下一个数大才不会引爆炸弹。因此，只需要根据结构体的值从大到小构造输入即可。

Secret Phase

> 35

通过ctrl-f查找secret_phase，可以发现，在phase_defused中，调用了这个函数。阅读代码容易发现，每当拆开一个炸弹后，会调用phase_defused表示该炸弹已经成功拆除。

```
1d66:   83 3d a3 3a 00 00 06    cmpl   $0x6,0x3aa3(%rip)        # 5810 <num_input_strings>
```

该代码显示出，当0x3aa3(%rip)等于6时，程序并不会直接退出，而是会继续判断是否满足进入隐藏关的条件。

```
1d88:   48 8d 4c 24 0c          lea    0xc(%rsp),%rcx
1d8d:   48 8d 54 24 08          lea    0x8(%rsp),%rdx
1d92:   4c 8d 44 24 10          lea    0x10(%rsp),%r8
1d97:   48 8d 35 b1 16 00 00    lea    0x16b1(%rip),%rsi        # 344f <array.0+0x27f>
1d9e:   48 8d 3d 6b 3b 00 00    lea    0x3b6b(%rip),%rdi        # 5910 <input_strings+0xf0>
1da5:   b8 00 00 00 00          mov    $0x0,%eax
1daa:   e8 a1 f3 ff ff          callq  1150 <__isoc99_sscanf@plt>
1daf:   83 f8 03                cmp    $0x3,%eax
```

这段代码可以理解为下面这段代码：

```
1   sscanf ("%d %d %s"); //from phase4
```

从phase_4的答案中读入两个整数和一个字符串。所以，要进入隐藏关，phase_4的答案需要在后面加上一个字符串。由于在phase_4中只读入两个整数，所以这样并不会影响phase_4的正确性。

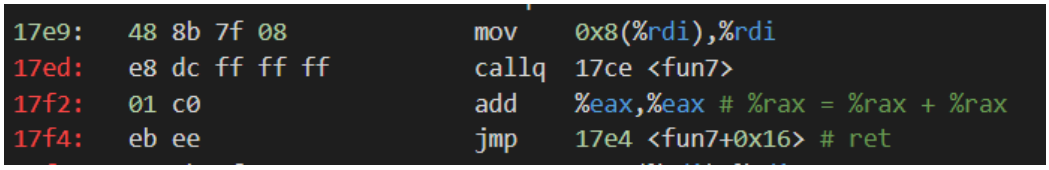
```
1dd3:   48 8d 35 7e 16 00 00    lea    0x167e(%rip),%rsi        # 3458 <array.0+0x288>
1dda:   e8 1b fb ff ff          callq  18fa <strings_not_equal>
```

从后文的字符串比较函数中，容易发现，目标字符串为：DrEvil。

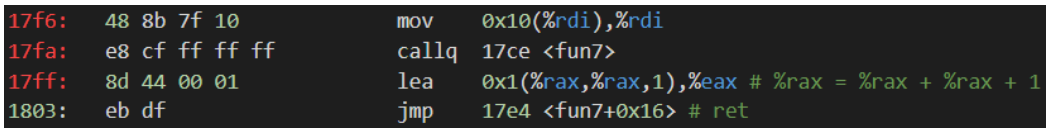
在phase_4的答案后面加上DrEvil后，成功进入隐藏关。

```
180c:   e8 00 04 00 00          callq  1c11 <read_line>
1811:   48 89 c7                mov    %rax,%rdi
1814:   ba 0a 00 00 00          mov    $0xa,%edx
1819:   be 00 00 00 00          mov    $0x0,%esi
181e:   e8 0d f9 ff ff          callq  1130 <strtol@plt>
1823:   48 89 c3                mov    %rax,%rbx
1826:   8d 40 ff                lea    -0x1(%rax),%eax
1829:   3d e8 03 00 00          cmp    $0x3e8,%eax
```

隐藏关需要读入一行东西，并且通过strtol函数把这行东子最开始的数字进行转化，并且要求这个数字减一后要小于或等于1000（即0x3e8）。然后把这个数字作为fun7的参数，调用fun7。

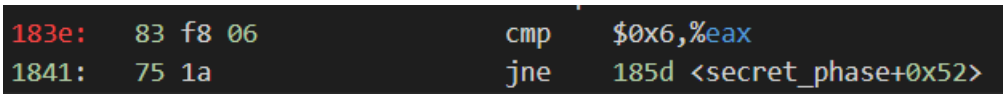


fun7将输入的数字和内存中的数比较。如果内存中的数较大，则进行递归，在回溯的时候，将%eax中的值乘2。



如果内存中的数较小，则进行递归，在回溯的时候，将%eax中的值乘2加1。

如果内存中的数和输入的数字相等，那么将%eax清零后结束程序。



结合fun7结束时，函数对%eax的值的要求，需要fun7运行中是的%eax的值为6。那么，在fun7中，需要第一个比内存中的数小，第二、三次比内存中的数大，第四次等于内存中的数。经过测试发现，该情况下，内存中的数依次为36 8 22 35。所以，只需要输入35即可解开隐藏关。

Summary

BombLab需要熟练使用gdb设置断点、查询内存中的值等功能。通过这些功能，可以快读地读懂汇编代码。