Bomblab Report

phase_1

观察汇编，

```
0000000000001607 <phase_1>:
    1607:   f3 0f 1e fa             endbr64
    160b:   48 83 ec 08             sub    $0x8,%rsp
    160f:   48 8d 35 ca 1c 00 00    lea    0x1cca(%rip),%rsi        # 32e0
<_IO_stdin_used+0x2e0>
    1616:   e8 12 06 00 00          call   1c2d <strings_not_equal>
    161b:   85 c0                   test   %eax,%eax
    161d:   75 05                   jne    1624 <phase_1+0x1d>
    161f:   48 83 c4 08             add    $0x8,%rsp
    1623:   c3                      ret
    1624:   e8 27 07 00 00          call   1d50 <explode_bomb>
    1629:   eb f4                   jmp    161f <phase_1+0x18>
```

此处逻辑为，将 `0x1cca(%rip)` 赋给 %rsi，然后调用 `strings_not_equal` 字符比较函数。注意到 `phase_1` 没有修改 %rdi，因此传给 `strings_not_equal` 的两个待比较字符串分别是 `phase_1` 的第一个参数（我们输入给炸弹的字符串），和 %rsi 地址处的字符串。

因此使用 `b *phase_1+15` 给调用比较函数的地方下断点，然后使用 `x/s $rsi` 将 %rsi 中地址对应的内存以字符串的形式输出，得到答案。

```
For NASA, space is still a high priority.
```

phase_2

`phase_2` 代码较长。我首先观察函数开始部分。

```
000000000000162b <phase_2>:
    162b:   f3 0f 1e fa             endbr64
    162f:   55                      push   %rbp
    1630:   53                      push   %rbx
    1631:   48 83 ec 28             sub    $0x28,%rsp
    1635:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
    163c:   00 00
    163e:   48 89 44 24 18          mov    %rax,0x18(%rsp)
    1643:   31 c0                   xor    %eax,%eax
    1645:   48 89 e6                mov    %rsp,%rsi
    1648:   e8 2f 07 00 00          call   1d7c <read_six_numbers>
```

前面的压栈操作保存了被调用者保存寄存器，然后通过减栈指针扩大栈。`mov %fs:0x28,%rax` 用于设置保护栈帧的金丝雀值，与拆弹并没有太大关系。接下来，清空 %eax，将栈指针的值赋给 %rsi，然后调用 `read_six_numbers` 函数。

观察 `read_six_numbers` 函数：

```
0000000000001d7c <read_six_numbers>:
```

```
    1d7c:   f3 0f 1e fa             endbr64
    1d80:   48 83 ec 08             sub    $0x8,%rsp
    1d84:   48 89 f2                mov    %rsi,%rdx
    1d87:   48 8d 4e 04             lea    0x4(%rsi),%rcx
    1d8b:   48 8d 46 14             lea    0x14(%rsi),%rax
    1d8f:   50                      push   %rax
    1d90:   48 8d 46 10             lea    0x10(%rsi),%rax
    1d94:   50                      push   %rax
    1d95:   4c 8d 4e 0c             lea    0xc(%rsi),%r9
    1d99:   4c 8d 46 08             lea    0x8(%rsi),%r8
    1d9d:   48 8d 35 35 13 00 00    lea    0x1335(%rip),%rsi        # 30d9
<_IO_stdin_used+0xd9>
    1da4:   b8 00 00 00 00          mov    $0x0,%eax
    1da9:   e8 62 f5 ff ff          call   1310 <_init+0x310>
    1dae:   48 83 c4 10             add    $0x10,%rsp
    1db2:   83 f8 05                cmp    $0x5,%eax
    1db5:   7e 05                   jle    1dbc <read_six_numbers+0x40>
    1db7:   48 83 c4 08             add    $0x8,%rsp
    1dbb:   c3                      ret
    1dbc:   e8 8f ff ff ff          call   1d50 <explode_bomb>
```

首先，利用 %rsi（保存的是栈指针的值）将参数寄存器都赋了栈中的地址。由于最多有 6 个参数寄存器，因此多余的参数需要压栈。这里利用压 %rax 来完成。

完成参数准备后，修改 %rsi 的值，接着使用 `call 1310 <_init+0x310>` 调用 `sscanf` 系统调用。查看其 man page，获得其签名：

```
int sscanf(const char *str, const char *format, ...);
```

`phase_2` 并没有修改 %rdi，因此第一个参数 `str` 是拆弹密码。通过在调用之前打断点，然后查看 %rsi，得知 `format` 的内容是 `%d %d %d %d %d %d`，即读入 6 个整数。然后检查 `sscanf` 的返回值（%eax，成功匹配并赋值的数据项数），若小于等于 5 则引爆炸弹。若读入成功，则读入的 6 个整数此时位于栈中，最先读入的在栈顶。

返回来看调用 `read_six_numbers` 之后的部分：

```
    1648:   e8 2f 07 00 00          call   1d7c <read_six_numbers>
    164d:   83 3c 24 01             cmpl   $0x1,(%rsp)
    1651:   75 0a                   jne    165d <phase_2+0x32>
    1653:   48 89 e3                mov    %rsp,%rbx
    1656:   48 8d 6c 24 14          lea    0x14(%rsp),%rbp
    165b:   eb 10                   jmp    166d <phase_2+0x42>
    165d:   e8 ee 06 00 00          call   1d50 <explode_bomb>
    1662:   eb ef                   jmp    1653 <phase_2+0x28>
    1664:   48 83 c3 04             add    $0x4,%rbx
    1668:   48 39 eb                cmp    %rbp,%rbx
    166b:   74 10                   je     167d <phase_2+0x52>
    166d:   8b 03                   mov    (%rbx),%eax
    166f:   01 c0                   add    %eax,%eax
    1671:   39 43 04                cmp    %eax,0x4(%rbx)
    1674:   74 ee                   je     1664 <phase_2+0x39>
    1676:   e8 d5 06 00 00          call   1d50 <explode_bomb>
    167b:   eb e7                   jmp    1664 <phase_2+0x39>
    167d:   48 8b 44 24 18          mov    0x18(%rsp),%rax
```

```
    1682:   64 48 2b 04 25 28 00    sub    %fs:0x28,%rax
    1689:   00 00
    168b:   75 07                   jne    1694 <phase_2+0x69>
    168d:   48 83 c4 28             add    $0x28,%rsp
    1691:   5b                      pop    %rbx
    1692:   5d                      pop    %rbp
    1693:   c3                      ret
    1694:   e8 c7 fb ff ff          call   1260 <_init+0x260>
```

这段代码实现了一个循环。首先判断栈顶所指向的内存值（刚刚读到的第 1 个整数）是否为 1，若不是则引爆炸弹，然后将栈指针 %rsp 的值赋给 %rbx，将 %rbx 作为一个循环变量来维护，每次循环 +4，然后读取。此外，每次循环都将 %rbx 处的值赋给 %eax 并乘 2，再与栈中的下一个数字比较，若不同则引爆炸弹。

综合分析，这里首先对比用户输入的第一个数字是否是 1，接着比较 5 次，看每次的数字是否是上次的 2 倍。从而得到 phase_2 的答案。

```
1 2 4 8 16 32
```

phase_3

开始部分，调用 `sscanf` 读了 3 个东西。利用与 phase_2 相同的方法分析，得到读入的格式是 `%d %c %d`。%rdx 储存第 3 个参数，因而是第一个读入的整数。%rcx 是字符。%r8 是最后一个整数。它们都位于栈中，整体读入逻辑与 phase_2 相同。

在读入之后，利用

```
16d1:   83 7c 24 10 07          cmpl   $0x7,0x10(%rsp)
16d6:   0f 87 0d 01 00 00       ja     17e9 <phase_3+0x150>
```

对第一个输入的数字进行判断，若大于 7 则引爆炸弹。我输入 1，然后继续追踪程序运行。

```
16dc:   8b 44 24 10             mov    0x10(%rsp),%eax
16e0:   48 8d 15 f9 1d 00 00    lea    0x1df9(%rip),%rdx        # 34e0
<_IO_stdin_used+0x4e0>
16e7:   48 63 04 82             movslq (%rdx,%rax,4),%rax
16eb:   48 01 d0                add    %rdx,%rax
16ee:   3e ff e0                notrack jmp *%rax
```

这里对输入的第一个数字进行计算，然后依据计算结果跳转。观察到之后存在大量结构相似的代码段，推测这里实现了一个 switch case 结构。无需关心这里的分支选择是如何进行的，只需在 0x16ee 打断点，读取 %rax 的值，就知道跳转到了何处。

```
(gdb) x $rax
0x55555555571a <phase_3+129>:   0x000075b8
```

```
171a:   b8 75 00 00 00          mov    $0x75,%eax
171f:   81 7c 24 14 24 03 00    cmpl   $0x324,0x14(%rsp)
1726:   00
1727:   0f 84 c6 00 00 00       je     17f3 <phase_3+0x15a>
172d:   e8 1e 06 00 00          call   1d50 <explode_bomb>
```

这里的逻辑是比较输入的最后一个数字是否等于 0x324，即十进制的 804，若不等则引爆炸弹，相等则继续跳转。

查看 0x17f3 处的代码。

```
    17f3:   38 44 24 0f             cmp    %al,0xf(%rsp)
    17f7:   75 15                   jne    180e <phase_3+0x175>
    17f9:   48 8b 44 24 18          mov    0x18(%rsp),%rax
    17fe:   64 48 2b 04 25 28 00    sub    %fs:0x28,%rax
    1805:   00 00
    1807:   75 0c                   jne    1815 <phase_3+0x17c>
    1809:   48 83 c4 28             add    $0x28,%rsp
    180d:   c3                      ret
    180e:   e8 3d 05 00 00          call   1d50 <explode_bomb>
    1813:   eb e4                   jmp    17f9 <phase_3+0x160>
    1815:   e8 46 fa ff ff          call   1260 <_init+0x260>
```

这里检查输入中间的那个字母是否与 %al 寄存器相等。只需打断点并读取 %al 寄存器的值即可，按照 ASCII 码转换即可得到答案的最后一部分。

因此，phase_3 的答案为：

```
1 u 804
```

phase_4

观察函数的开始部分，利用与前几个阶段相同的方法，获知这里从标准输入读取 2 个整数，第 2 个保存在栈顶，第 1 个保存在栈顶的下一个位置。

```
0000000000001855 <phase_4>:
    1855:   f3 0f 1e fa             endbr64
    1859:   48 83 ec 18             sub    $0x18,%rsp
    185d:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
    1864:   00 00
    1866:   48 89 44 24 08          mov    %rax,0x8(%rsp)
    186b:   31 c0                   xor    %eax,%eax
    186d:   48 89 e1                mov    %rsp,%rcx
    1870:   48 8d 54 24 04          lea    0x4(%rsp),%rdx
    1875:   48 8d 35 69 18 00 00    lea    0x1869(%rip),%rsi        # 30e5
<_IO_stdin_used+0xe5>
    187c:   e8 8f fa ff ff          call   1310 <_init+0x310>  # sscanf($rsp+4,
$rsp)
    1881:   83 f8 02                cmp    $0x2,%eax
    1884:   75 0b                   jne    1891 <phase_4+0x3c>
```

读入之后，先经过以下处理：

```
    1886:   8b 04 24                mov    (%rsp),%eax
    1889:   83 e8 02                sub    $0x2,%eax
    188c:   83 f8 02                cmp    $0x2,%eax
    188f:   76 05                   jbe    1896 <phase_4+0x41>
    1891:   e8 ba 04 00 00          call   1d50 <explode_bomb>
```

这里将第 2 个数字先减去 2，再与 2 比较，如果小于等于 2 则继续执行，否则引爆炸弹。

我先将第 2 个输入设为 3，然后在跳转之后的位置打断点。

```
    1896:   8b 34 24                mov    (%rsp),%esi
    1899:   bf 08 00 00 00          mov    $0x8,%edi  # func4(8, $rsp) (8, 3)
    189e:   e8 77 ff ff ff          call   181a <func4>
    18a3:   39 44 24 04             cmp    %eax,0x4(%rsp) # compare with sscanf's
first number
    18a7:   75 15                   jne    18be <phase_4+0x69>
    18a9:   48 8b 44 24 08          mov    0x8(%rsp),%rax
    18ae:   64 48 2b 04 25 28 00    sub    %fs:0x28,%rax
    18b5:   00 00
    18b7:   75 0c                   jne    18c5 <phase_4+0x70>
    18b9:   48 83 c4 18             add    $0x18,%rsp
    18bd:   c3                      ret
    18be:   e8 8d 04 00 00          call   1d50 <explode_bomb>
    18c3:   eb e4                   jmp    18a9 <phase_4+0x54>
    18c5:   e8 96 f9 ff ff          call   1260 <_init+0x260>
```

这里调用了 `func_4(8, 3)`，然后检查用户输入的第 1 个数是否与该函数的返回值相同。

观察 `func_4`，发现它是一个多入口的递归函数。然而，由于其对于特定用户输入的行为是确定性的，且只与作为参数传入的第 2 个用户输入有关，因此无需关心该函数是如何运行的。只需要在 0x18a3 的指令处打断点，抓取其返回值，作为答案的第 1 个数字即可。

phase_4 的答案为：

```
162 3
```

phase_5

首先，将输入的字符串地址保存在 %rbx 中，然后检测输入的串长度是否等于 6，若不等则引爆炸弹。

```
00000000000018ca <phase_5>:
    18ca:   f3 0f 1e fa             endbr64
    18ce:   53                      push   %rbx
    18cf:   48 83 ec 10             sub    $0x10,%rsp
    18d3:   48 89 fb                mov    %rdi,%rbx
    18d6:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
    18dd:   00 00
    18df:   48 89 44 24 08          mov    %rax,0x8(%rsp)
    18e4:   31 c0                   xor    %eax,%eax
    18e6:   e8 21 03 00 00          call   1c0c <string_length>
    18eb:   83 f8 06                cmp    $0x6,%eax
    18ee:   75 55                   jne    1945 <phase_5+0x7b>
```

接着，读取一个字符串的地址，保存在 %rcx 中。

```
    18f0:   b8 00 00 00 00          mov    $0x0,%eax
    18f5:   48 8d 0d 04 1c 00 00    lea    0x1c04(%rip),%rcx        # 3500
<array.0>
```

利用与 phase_1 一样的手法，得到这个字符串为

```
maduiersnfotvbyl
```

记为 array.0。

接下来的代码实现了一个循环，遍历了输入串。

```
    18fc:   0f b6 14 03             movzbl (%rbx,%rax,1),%edx
    1900:   83 e2 0f                and    $0xf,%edx
    1903:   0f b6 14 11             movzbl (%rcx,%rdx,1),%edx
    1907:   88 54 04 01             mov    %dl,0x1(%rsp,%rax,1)
    190b:   48 83 c0 01             add    $0x1,%rax
    190f:   48 83 f8 06             cmp    $0x6,%rax
    1913:   75 e7                   jne    18fc <phase_5+0x32>
```

每次将输入串的一个字符的 ASCII 码和 0xf 做与运算，相当于模 16，得到一个 0 到 15 的整数，保存在 %edx 中。接着，以这个整数为下标访问 array.0，将得到的字符保存在栈中。

```
    1915:   c6 44 24 07 00          movb   $0x0,0x7(%rsp)
    191a:   48 8d 7c 24 01          lea    0x1(%rsp),%rdi
    191f:   48 8d 35 6b 17 00 00    lea    0x176b(%rip),%rsi        # 3091
<_IO_stdin_used+0x91>
    1926:   e8 02 03 00 00          call   1c2d <strings_not_equal>
```

在这个循环结束之后，将 0 放入栈中作为字符串结束标志，然后比较得到的新字符串与 `0x176b(%rip)` 处的目标字符串，不等则引爆炸弹。查看该内存地址，得到目标字符串为

```
flyers
```

计算其中每个字母在 array.0 中的下标，反推用户输入的 ASCII 码，得到 phase_5 的答案为

```
ionefg
```

phase_6

phase_6 开始时读入了 6 个整数，全部保存在栈中 %rsp + 16 到 %rsp + 36 的位置。

紧接着，检查了这 6 个整数中是否有相等的数，且是否都小于等于 6。若否，则引爆炸弹。

```
    19ae:   48 83 c3 01             add    $0x1,%rbx  # {
    19b2:   83 fb 05                cmp    $0x5,%ebx  #  loop var
    19b5:   0f 8f d5 00 00 00       jg     1a90 <phase_6+0x138>  # examine ends
    19bb:   41 8b 44 9d 00          mov    0x0(%r13,%rbx,4),%eax
    19c0:   39 45 00                cmp    %eax,0x0(%rbp)
    19c3:   75 e9                   jne    19ae <phase_6+0x56> # }
    19c5:   e8 86 03 00 00          call   1d50 <explode_bomb>  # explode if
input numbers are same
```

```
    19cc:   49 83 c6 01             add    $0x1,%r14  # loop 6 times?
    19d0:   49 83 fe 07             cmp    $0x7,%r14
    19d4:   0f 85 96 00 00 00       jne    1a70 <phase_6+0x118>
    ...
    1a70:   49 83 c7 04             add    $0x4,%r15  # ### each time r15 += 4
```

```
    1a74:   eb 22                   jmp    1a98 <phase_6+0x140>
    ...
    1a98:   4c 89 fd                mov    %r15,%rbp  # ###
    1a9b:   41 8b 07                mov    (%r15),%eax
    1a9e:   83 e8 01                sub    $0x1,%eax
    1aa1:   83 f8 05                cmp    $0x5,%eax
    1aa4:   0f 87 f3 fe ff ff       ja     199d <phase_6+0x45>  # explode if
input number > 6
    1aaa:   41 83 fe 05             cmp    $0x5,%r14d  # ### r14d = 1 at first
time
    1aae:   0f 8f 18 ff ff ff       jg     19cc <phase_6+0x74>
```

由上述代码可以推断，我们要输入的内容是 1-6 的一个排列。

接着阅读代码，我们发现了一个双重循环。

```
    19fb:   be 00 00 00 00          mov    $0x0,%esi
    1a00:   8b 4c b4 10             mov    0x10(%rsp,%rsi,4),%ecx  # {  rsp + 4 *
rsi + 16
    1a04:   b8 01 00 00 00          mov    $0x1,%eax
    1a09:   48 8d 15 00 38 00 00    lea    0x3800(%rip),%rdx        # 5210
<node1>
    1a10:   83 f9 01                cmp    $0x1,%ecx
    1a13:   7e 0b                   jle    1a20 <phase_6+0xc8>
    1a15:   48 8b 52 08             mov    0x8(%rdx),%rdx  # [
    1a19:   83 c0 01                add    $0x1,%eax
    1a1c:   39 c8                   cmp    %ecx,%eax
    1a1e:   75 f5                   jne    1a15 <phase_6+0xbd>  # ]
    1a20:   48 89 54 f4 30          mov    %rdx,0x30(%rsp,%rsi,8) # 48 + rsp + 8
* rsi
    1a25:   48 83 c6 01             add    $0x1,%rsi
    1a29:   48 83 fe 06             cmp    $0x6,%rsi
    1a2d:   75 d1                   jne    1a00 <phase_6+0xa8>  # }
```

这里读取了一处 node1 的地址到 %rdx。猜测是某种线性的数据结构。

将其打印出来，观察到这是一个链表，每个节点内有 16 个字节的数据。猜测第一个 4 字节数据为链表节点包含的数据值，第二个 4 字节数据为链表节点 id，最后两个 4 位数据，按照第三个对应低位，第四个对应高位，组合成一个 64 位的地址，为该链表节点的 next 指针。

```
Breakpoint 3, 0x0000555555555a10 in phase_6 ()
(gdb) x/24x $rdx
0x555555559210 <node1>: 0x00000194      0x00000001      0x55559220
0x00005555
0x555555559220 <node2>: 0x0000029c      0x00000002      0x55559230
0x00005555
0x555555559230 <node3>: 0x0000019f      0x00000003      0x55559240
0x00005555
0x555555559240 <node4>: 0x0000036e      0x00000004      0x55559250
0x00005555
0x555555559250 <node5>: 0x000000f0      0x00000005      0x55559110
0x00005555
0x555555559260 <host_table>:    0x5555714f      0x00005555      0x55557169
0x00005555
```

容易发现缺失了 node6。打印出 node5 next 指针处的数据可以找到它。

```
(gdb) x/4x 0x0000555555559110
0x555555559110 <node6>: 0x000000aa      0x00000006      0x00000000
0x00000000
```

将数据部分转换为十进制：

id	value
1	404
2	668
3	415
4	878
5	240
6	170

继续观察循环。外层循环中，每次从栈中取出一个用户输入到 %ecx 内，然后在内层循环中，按照用户输入的数字大小跳链表节点。每次内层循环结束之后，将跳到的链表节点的地址存在栈中。

在 0x1a20 处打断点，观察被读出链表节点的顺序。经过几次试验可以发现，记输入的 6 个数分别为 $a_1, a_2, \cdots, a_6$，则链表节点按照 id 为 $6 - a_1 + 1, 6 - a_2 + 1, \cdots, 6 - a_6 + 1$ 的顺序被读出。

接下来有一大串 `mov` 指令。

```
   1a2f:   48 8b 5c 24 30          mov    0x30(%rsp),%rbx
   1a34:   48 8b 44 24 38          mov    0x38(%rsp),%rax
   1a39:   48 89 43 08             mov    %rax,0x8(%rbx)
   1a3d:   48 8b 54 24 40          mov    0x40(%rsp),%rdx
   1a42:   48 89 50 08             mov    %rdx,0x8(%rax)
   1a46:   48 8b 44 24 48          mov    0x48(%rsp),%rax
   1a4b:   48 89 42 08             mov    %rax,0x8(%rdx)
   1a4f:   48 8b 54 24 50          mov    0x50(%rsp),%rdx
   1a54:   48 89 50 08             mov    %rdx,0x8(%rax)
   1a58:   48 8b 44 24 58          mov    0x58(%rsp),%rax
   1a5d:   48 89 42 08             mov    %rax,0x8(%rdx)
   1a61:   48 c7 40 08 00 00 00    movq   $0x0,0x8(%rax)
```

模拟代码执行可以发现，这里先把链表头的地址存在了 %rbx 寄存器中，然后逐项遍历链表节点，把链表通过修改 next 指针的方法，按照刚刚被读出的顺序重新排列了一遍。

紧接着，开始最后一次循环。

```
    1a69:   bd 05 00 00 00          mov    $0x5,%ebp
    1a6e:   eb 0f                   jmp    1a7f <phase_6+0x127>
    ...
    1a76:   48 8b 5b 08             mov    0x8(%rbx),%rbx  # [
    1a7a:   83 ed 01                sub    $0x1,%ebp
    1a7d:   74 3d                   je     1abc <phase_6+0x164>
    1a7f:   48 8b 43 08             mov    0x8(%rbx),%rax
    1a83:   8b 00                   mov    (%rax),%eax
    1a85:   39 03                   cmp    %eax,(%rbx)
    1a87:   7d ed                   jge    1a76 <phase_6+0x11e>  # ]
    1a89:   e8 c2 02 00 00          call   1d50 <explode_bomb>  # ?
```

这里维护 %ebp 作为循环变量，一共循环 6 次。循环开始之前先取出第一个链表节点作为基准，每次循环，取出一个重排之后的链表节点，并与上一个链表节点比较。若当前节点数据值大于等于上一个节点数据值，则继续循环，否则跳出循环引爆炸弹。

综合分析可以得出，本阶段的目标是依据用户输入，得到一个从大到小完成排序的链表。

将链表按数据值从大到小排序，id 顺序为：

```
4 2 3 1 5 6
```

按照先前得出的输入与链表 id 间的映射关系，得到 phase_6 答案为

```
3 5 4 6 2 1
```