

# Re: ゼロから始める新世界の内核勉強

## 前言

好难。另外这次改了用户名之后还是挺清静的，前期基本没有人找我问题目。

直到w把我的ID改成了“大屁股眼子”

## 先河长长长[zhang]

cnss{xianhe\_zhang\_zhang\_zhang}

一看到这题，背包问题嘛，最简单的，01背包。

数据范围m比较大，n比较小， $O(mn)$ 比一亿稍微大点。本来考虑到bitset可以优化常数（ $O(mn/64)$ ），结果仍然在第五组数据TLE。

然后深度优先搜索啊，当然加上了几个简单的剪枝，从大的开始枚举、搜到刚好一样高就退出，反而过了，比较搞笑.....

几个先前的代码好像都找不到了，下面是重新打的。

++++++，更新了数据，还是有错的，故意卡getchar()，害得我重新提交了一遍DP代码。

[这是重新打的DP代码](#)

```
1  #include <algorithm>
2  #include <bitset>
3  #include <cstdio>
4  using namespace std;
5
6  int n, m, v[105];
7  int M;
8  bitset<20200000> dp;
9
10 int main() {
11     scanf("%d%d", &n, &m);
12     for (int i = 0; i < n; ++i)
13         scanf("%d", &v[i]), M = max(M, v[i]);
14     M += m; dp.set(0);
15     for (int i = 0; i < n; ++i)
16         for (int j = M; j >= v[i]; --j)
17             if (dp.test(j - v[i])) {
18                 dp.set(j);
19                 if (j == m) {
20                     puts("0");
21                     return 0;
22                 }
23             }
24     for (int i = m; i <= M; ++i)
25         if (dp.test(i)) { printf("%d\n", i - m); break; }
26     return 0;
27 }
```

C++

这是重新打的深搜+剪枝代码

```
1 #include <algorithm>
2 #include <cstdio>
3 #include <functional>
4 using namespace std;
5
6 int n, m, v[105];
7 int ans = ~0u >> 1;
8
9 void dfs(int next, int sum) {
10     if (sum > m) ans = min(ans, sum);
11     else if (sum == m) { puts("0"); exit(0); }
12     else for (int i = next; i < n; ++i) dfs(i + 1, sum + v[i]);
13 }
14
15 int main() {
16     scanf("%d%d", &n, &m);
17     for (int i = 0; i < n; ++i) scanf("%d", &v[i]);
18     sort(v, v + n, greater<int>());
19     dfs(0, 0);
20     printf("%d\n", ans - m);
21     return 0;
22 }
```

C++

嘛，后来室友发现还有优化的算法，上十万百万都可以，大致是.....把相同的并成同一个一起考虑。小数据跑得快，大数据重复多，所以可过。

## Hello, World! #1

cnss{fdsjh3@%\$%gkk}

没什么好说的，直接上代码好了。

[AC记录](#)

```
1 #include <cstdio>
2 using namespace std;
3
4 int main() {
5     puts("Hello, World!");
6     return 0;
7 }
```

C++

## Hello, World! #2

cnss{8573489rhfjbfjdshj}

.....不就是不用引号吗.....ASCII字符本身就是数值，于是：

[上代码](#)

C++

```

1  #include <cstdio>
2  using namespace std;
3
4  char hw[] = {
5      0x48, 0x65, 0x6c, 0x6c, // Hell
6      0x6f, 0x2c, 0x20, 0x57, // o, W
7      0x6f, 0x72, 0x6c, 0x64, // orld
8      0x21, 0x00           // !
9  };
10
11 int main() {
12     puts(hw);
13     return 0;
14 }

```

## Hello, World! #3

cnss{DSG543kks}

if啊while啊之类的语句有大括号没有分号。

代码

```

1  #include <cstdio>
2  using namespace std;
3
4  int main() {
5      if (puts("Hello, World!")) {}
6      return 0;
7  }

```

C++

## Hello, World! #4

cnss{!!!@@@###9283466}

解法 0x00

一开始想直接gcc -E预处理后生成的代码能不能直接用（当然预处理之后也有一些#，可以手动删）。本地测试通过了，结果在OJ上CE了。后来自己声明了一下函数，本地通过，OJ上CE，不知道什么原因。

还好我知道GCC有builtin函数不用库，这就简单了。

日常上代码

```

1  int main() {
2      __builtin_printf("Hello, World!");
3  }

```

C++

解法 0x01

我知道为什么gcc -E预处理不可以了，因为OJ平台的编译器是g++，一口老血。题目处还有提示后缀是.cpp，一口老血。

用了g++预处理成功，然后用vim删除多余行就可以了。

```
1 | :g/[ ]*#.* /d
```

vim

以上是vim命令，没什么。然后其实这样还是有很多多余的函数的，实际上只需要一行声明puts函数即可。貌似忘了平台提交的record了.....抱歉没代码了.....

```
1 | extern "C" {
2 |     extern int puts(const char *__s);
3 | }
4 |
5 | int main() {
6 |     puts("Hello, World!");
7 |     return 0;
8 | }
```

C++

## 解法 0x02

一开始就有的想法，内联汇编。之前用int 80h不知道为什么一直失败，于是就先放着了。后来在爆栈网查到说x86\_64可以用syscall而且建议用syscall，于是我试了一下.....直接AC了.....

```
1 | char hewo[] = "Hello, World!";
2 | int main() {
3 |     asm("mov    $0x01, %rax"); // syscall调用系统函数编号，此处为sys_write
4 |     asm("mov    $0x01, %rdi"); // sys_write输出到stdout，其编号为1
5 |     asm("mov    $hewo, %rsi"); // 输出字符串的地址
6 |     asm("mov    $0x0d, %rdx"); // 输出字符串的长度
7 |     asm("syscall");
8 |     return 0;
9 | }
```

C++

AT&T用着真不舒服.....

## 解法 0x03

并没有提交这种做法.....我在做HW5的时候有搜到过。详细的可以去搜索Digraphs and Trigraphs，[英文维基](#)上有很详细的介绍。

可以看到，我们可以在C语言中用`??=`或者`?:`来替代`#`

不打算交，不过writeup里写一下吧，代码是最近测试的。

### AC代码

```
1 | ??=include <cstdio>
2 | using namespace std;
3 |
4 | int main() {
5 |     puts("Hello, World!");
6 |     return 0;
7 | }
```

C++

```
1 %:include <cstdio>
2 using namespace std;
3
4 int main() {
5     puts("Hello, World!");
6     return 0;
7 }
```

## Hello, World! #5

cnss{\_\_01010101010101010101\_\_}

一开始在网上搜过，只搜到两个地方有，分别是[吾爱破解](#)和[看雪论坛](#) .....还是同一个人问的.....而且也是别人给TA的任务.....不管怎样算是给了我思路。

你不用想到什么了，没错，后来在群里得知就是那个出了渗透Boss题的14级学长。

哈哈哈哈哈不许我学一会儿，神TM的黑历史啊。

不过后来我是看长者做了才去赶的.....因为之前已经有思路了，但是打出来不能运行然而完全不知道问题出在哪里.....后来去找了w和锐锐，最后才明白为什么我的不能运行.....原来是学长编译参数打错了.....

更坑爹的就是，Windows下的bash即使 `-zexecstack` 也无法正常运行，导致我以为我的就是错的，不过最后的最后我在锐锐指导下在虚拟机里跑了一遍可以正常输出，才明白这又是win的坑。

长者做出了HW5的时候我大概还泡在渗透的Boss里无法自拔，发现他做出来之后我又去搜了很多资料，就是关于 `-z` 的，编译选项 `#param` 的，最后往后走甚至涉及到了ELF文件格式、可执行文件格式的内容.....所以虽然花了不少时间，现在好歹也掌握了一点知识.....

至少知道了几个有名的段有什么作用，知道了为什么我全局声明的值为0的变量会被移走移到 `.bss` 段（加编译参数 `-fno-zero-initialized-in-bss` 就不会被优化到那边去了），知道了为什么 `.data` 段里的指令默认不能执行（也是加个编译选项 `-zexecstack` 就有x权限），也知道了怎么强行把数据、函数保存在其它段 `__attribute__((section(".text")))`，以及 `attribute` 的一些其它用法。

除此之外，我还学会了objdump的使用和更加熟练使用gdb的技术。

**然而题目还是没有做出来。**

所以在花了一晚上之后，我去找了w（聊天记录显示是10月14日），并且在w的指引下找了锐锐进行了一场激情四射的交易，最后发现是oj上编译选项错误。

然后就进入了后来的我使劲催w改参数的历程。

大约一个星期过后，也就是今（其实已经是昨）天10月22日，终于在我眼皮底下改了参数。

回寝室之后，稍微重打了一下代码，然后就过了。

这题的原理其实还是很好理解的，GCC在看到函数的時候，會把這個函數編譯出來的機器碼和函數名一起扔到 `.text` 段里，然後順序執行下去，而看到一個變量的時候，也會把變量的值和變量名扔到 `.data`，不過默認 `.data` 不可執行。上面的話想說明的是，其實程序和值都一樣的儲存在elf文件里，不過是因為存的段不一樣而導致有的可執行，有的只可讀寫。

这题那个编译参数本意是让本来默认不可执行的.data段可执行，所以其实就让main开始是机器码即可。又因为变量连续保存，所以就会一直执行下去。

此处是代码，恐怕很不好读，我用我原来没有压过的版本拿来逐行解释一下。

```
1 // 下面两行是Hello, World!\n的二进制形式
2 long hw0 = 0x57202c6f6c6c6548; // W ,olleH
3 long hw1 = 0x00000a21646c726f; // ...!dlro
4 // 下面开始是代码，注释为机器码翻译后的汇编
5 long main = 0x9090909090909090; // nop
6 long push = 0x9090909090909055; // push %rbp
7 long mov0 = 0x9090909090e58948; // mov %rsp, %rbp
8 long mov1 = 0x9000000001c0c748; // mov $0x1, %rax
9 long mov2 = 0x9000000001c7c748; // mov $0x1, %rdi
10 long mov3 = 0x9000600850c6c748; // mov $0x600850, %rsi
11 long mov4 = 0x90000000dc2c748; // mov $0xd, %rdx
12 long sysc = 0x90909090909050f; // syscall
13 long mov5 = 0x9090900000000b8; // mov $0x0, %rax
14 long pop = 0x909090909090905d; // pop %rbp
15 long retq = 0x909090909090c3; // retq
```

加了边上的注释之后，显然可以看出来我这个程序是HW4那题改过来的。

还是有几个地方需要解释一下的。

首先，因为ELF是小端序，所以定义的字符必须全部反着来，比如那个Hello, World!\n，本来二进制表示应该是0x48656c6c6f2c20576f726c64210a，然而代码中看得到，我这个全部是倒着来的，另外最后填充了几位无用的0。

我之所以不用char改用long的原因之前说过，编译器没开-fno-zero-initialized-in-bss参数，一些初始化为0的量会被直接优化到.bss段导致机器码断开无法解释运行。

然后还有个0x90填充的问题，稍微反编译一下就知道的，0x90相应的汇编指令是NOP，所以用来填充空余不会影响到指令执行。而我之所以要填充空白是因为一个不满64位的数字高位是默认0x00填充（废话），对应的汇编是add %al, (%rax)，导致Segment Fault。

同样的，机器代码是从右往左读的。

最后我AC之后向w要了一份另一个学长的方法，就是在改编译参数前就AC的，代码在这儿。从注释里看得出来.....被0x00坑得很惨.....之前用了int 80h不能运行的原因就是高位是默认0x00填充，出现了一堆add %al, (%rax)而Segment Fault。

objdump分析过，虽然中间代码也出现了0x00，而且很神奇地没有移到.bss，我不是很清楚为什么，大概是因为const吗。另一方面，学长不需要用到很多0x00的原因是因为在很多地方用了32位寄存器，于是舍去了很多0x00。

虽然还是看不懂学长代码.....感觉还是我的直观，虽然有运气成分在——我的编译器和服务器上的是同一个，所以最后Hello, World!保存的地址也都在同一处，直接就可以输出了。

不过，我现在大概也是知道为什么学长的能运行了.....因为他的是const，const的会把数据移动到可执行段，虽然我也用过const，但是当时因为Windows的原因结果照样segment fault，被坑了。

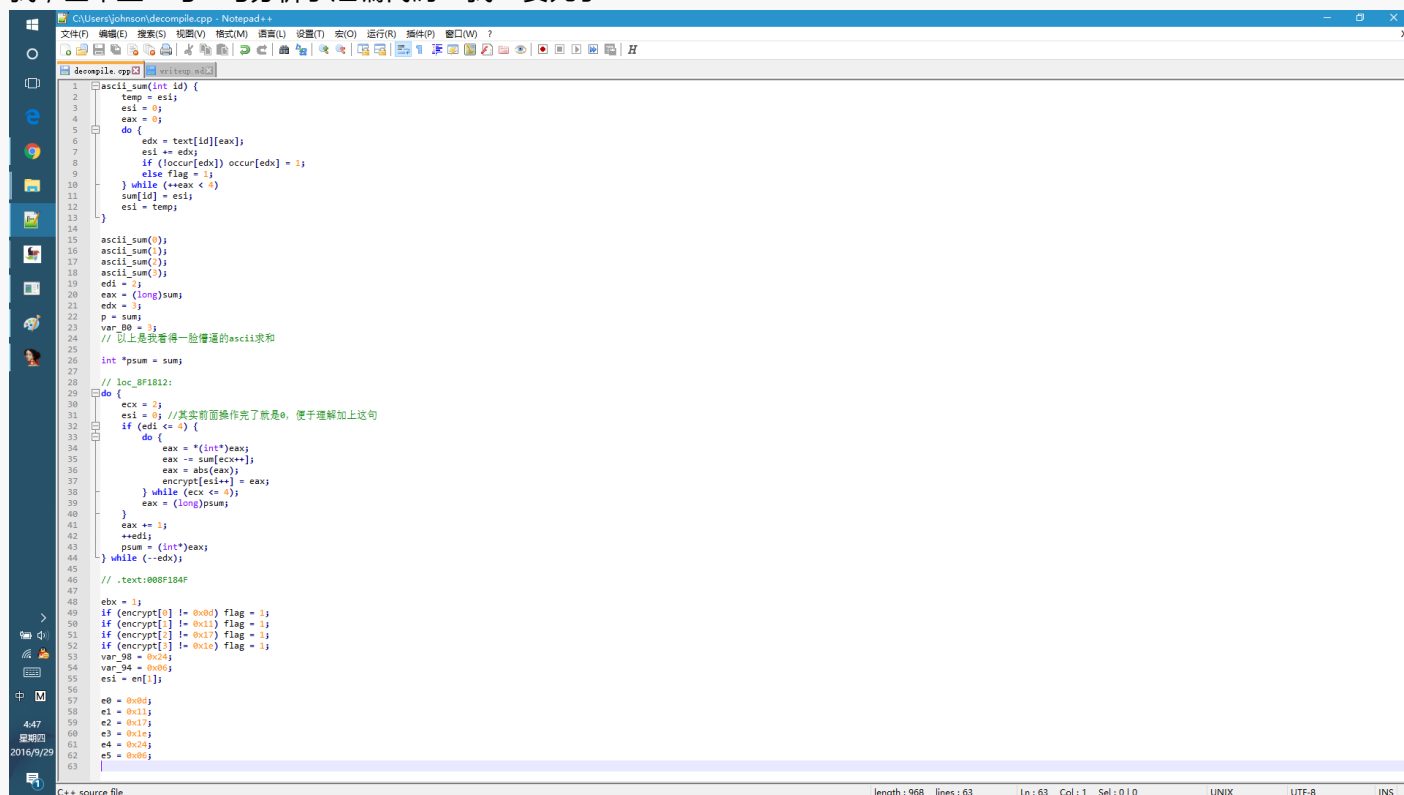
## 试试福昕吧

见试试福昕吧.zip包。

# 先河的小秘密

cnss{gkqx~`}

我，基本上一句一句分析了汇编代码.....我.....要死了.....



```
1  ascii_sum(int id) {
2      temp = esi;
3      esi = 0;
4      eax = 0;
5      do {
6          edx = text[id][eax];
7          esi += edx;
8          if (!occure[edx]) occur[edx] = 1;
9          else flag = 1;
10     } while (++eax < 4);
11     sum[id] = esi;
12     esi = temp;
13 }
14
15 ascii_sum(0);
16 ascii_sum(1);
17 ascii_sum(2);
18 ascii_sum(3);
19 edi = -1;
20 eax = (long)sum;
21 edx = 3;
22 p = sum;
23 var_80 = 3;
24 // 以上是我看得一些懵逼的ascii求和
25
26 int *psum = sum;
27
28 // loc_BF1812:
29 do
30 {
31     ecx = 2;
32     esi = 0; // 其实前面操作完了就是0, 便于理解加上这句
33     if (edi <= 4) {
34         do {
35             eax = *(int*)eax;
36             eax += sum[ecx++];
37             eax = abs(eax);
38             encrypt[esi++] = eax;
39             while (ecx <= 4);
40             eax = (long)psum;
41         }
42         ++edi;
43         psum = (int*)eax;
44     } while (--edx);
45     // .text:008F184F
46
47     ebx = 1;
48     if (encrypt[0] != 0x00) flag = 1;
49     if (encrypt[1] != 0x11) flag = 1;
50     if (encrypt[2] != 0x17) flag = 1;
51     if (encrypt[3] != 0x1e) flag = 1;
52     var_98 = 0x24;
53     var_94 = 0x00;
54     esi = en[1];
55
56     e0 = 0x0d;
57     e1 = 0x11;
58     e2 = 0x17;
59     e3 = 0x1e;
60     e4 = 0x24;
61     e5 = 0x06;
62
63 }
```

做出来之后发现.....卧槽怎么这么简单.....TNND

首先本来是 `shutdown -s -t 100` 的，题目更新了一下变成了 `shutdown -s -t 100`。我在这里先赞美一下ET学长。（然并软，根本没执行到

故事从5个连续的 `GetWindowText` 函数开始，到 `MessageBoxW` 和 `DestroyWindows` 和最后的 `SetWindowTextW` 结束，具体怎么找到的，搜索WARNING!或者INVALID KEY!就可以了。

不用查MSDN都知道，`GetWindowTextW` 是获取输入的几个字符，`SetWindowTextW` 是为了在cnss{}里填充真正的flag。重点在于中间的一长串加密算法，我试过F4反编译，代码极度不可读，于是开始了我手工反编译的历程。

具体的汇编分析就免了，因为我已经给出了我反编译的结果，就说说看怎么用最简单的方法得到吧。

其实在中间就应该留意到，有很多处给ecx和它在内存中一块临时寄存（我那个代码中全部改为变量flag）的区域赋值为1的语句以及判断flag是不是1然后跳转的语句。显然，源代码中有一个标记，记录输入数据满足不满足条件，使真正的flag能够被打印出来。

因此研究一下那几个置flag为1的地方即可。具体参照我的人工反编译代码。首先在那个给输入数据的ascii码求和的函数出现过，大概是ascii码的各项和不能重复，重复就置标记为1，这点在后面基本没影响。其次，最重要的地方，在下面我的四个连续的encrypt[] != 0x里面（我大幅简化了汇编，直接算了出来几个加密的值），出现了4个flag赋值的情况，并且在下面就有个判断flag并且跳转到1.给四个寄存器赋值并继续跳转到 `SetWindowTextW` 函数调用的地方2.弹出WARNING窗口的地方。

然而曾经令我意外的是，虽然有判断var\_98和var\_94的值的的地方，但是这两个值根本没有被初始化，也就是说，这个程序本来是根本不可能填充那个cnss的，这也是我在群里咆哮看不懂程序逻辑的那个时候。

我突然意识到var\_98和var\_94在内存中和encrypt[]原先的几个是连续的，而且甚至代码都是这么类似，我想到会不会这本来就是一个坑，故意不让我们通过乱输数字凑对.....

我大概明白了些什么.....

最后在调试进入 `SetWindowTextW` 函数push参数的时候改了几个寄存器的值，最后搞了出来。

居然又是乱码。

现在完全理解了加密算法，我可以用python打出来了。

```
1 | for x in (0xf2, 0xee, 0xe8, 0xe1, 0xdb, 0xf9): print(chr((x ^ 0xff) + 0x5a), end='') Python
```

上面这段代码里的常量完全就是程序里的常量。运行结果就是cnss{ }大括号里的东西。

## 神秘代码

cnss{she\_stayed\_at\_the\_July}

纯意译，讲真，把那三段内存里的值给翻译出来就大致明白了程序在干什么。不过安全起见我还是人工翻译了一遍。

只能说有时候编译成汇编反而更复杂了。

```
1 | lea eax, [num]
2 | push eax
3 | push 0x01046BD4
4 | call dword ptr [scanf]
5 | add esp, 8
6 | mov eax, dword ptr [num] NASM
```

这五句是为了调用scanf函数，将num的地址给eax，push eax和"%d"后call scanf函数，最后恢复栈顶指针，因为push了两个32位指针所以+8。然后又从内存中把num的值给了eax。

```
1 | cdq
2 | sub eax, edx
3 | sar eax, 1
4 | shl eax, 1
5 | cmp eax, dword ptr [num] NASM
```

cdq从某种意义上来说类似于数学的sgn()函数，每次cdq的时候如果`eax>=0`（`eax`的最高位为0）则`edx`为0，如果`eax<0`（`eax`的最高位为1）则`edx`为-1，其实就是每一位为`eax`的最高位，以便连接起来之后`edx:eax`的值和`eax`一样。

所以当`eax<0`的时候，`++eax`（我感觉这段代码是完全没有必要的，要不是编译器抽风，要不是故意出题混淆我们）。

接下来先右移再左移，实际上只是将`eax`的最低位强制置为0，再和原数比较是不是相等，实际上就是判断奇偶。

```
1 | jne 0x344cac
2 | push 0x01046BE4
3 | call dword ptr [printf]
4 | add esp, 4
5 | jmp 0x344cb9
6 | 0x344cac:
7 | push 0x01046CE4
8 | call dword ptr [printf]
9 | NASM
```



```
10 | add esp, 4
    | 0x344cb9:
```

几次反向经验告诉我，遇上这种

```
1 | jxx else
2 | ; Do some `then` things here
3 | jmp endif
4 | else:
5 | ; Do some `else` things here
6 | endif:
```

NASM

就是if-else代码。因为if-else语句块中都是各种push和call，所以也比较显然，是输出那两句even和odd的语句。

最后贴代码吧

```
1 | #include
2 | using namespace std;
3 |
4 | // %d
5 | // even number.
6 | // odd number.
7 |
8 | int num;
9 |
10 | int main() {
11 |     scanf("%d", &num);
12 |     printf(num & 1 ? "odd" : "even");
13 |     puts(" number.");
14 |     return 0;
15 | }
```

C++