ReverseLab 实验报告

1. Easystr

听说第一题可以一眼看出来
简单来说就是 [0x140004000] 中存了一个字符串Welcome_to_the_reverse_world!
对于前0x14个字节，将 l 改为 1，将 o 改为 0，即为flag

2. Xorrr

获取答案程序如下

```
#include<cstdio>
#include<iostream>
#include<string>

using namespace std;

string _str[3] = { "X1j3y5a7u9t;`=|",";5w7n9 ;l=n?)A-","s9?;}=j?|AxChEj"};

int main()
{
    int i = 0;
    string ans;
    for (i = 0;i < 24;i++) {
        char a = _str[i % 3][(i / 3) << 1];
        a ^= (i + 8);
        ans.push_back((a - 2));
    }
    int size = ans.size();
    cout << size << endl;
    for (int j = 0;j < size;j++) {
        printf("%X ", ans[j]);
    }
    cout << endl;
    cout << ans << endl;
    return 0;
}
```

答案

> 4E 30 77 5F 79 30 75 5F 6B 6E 30 77 5F 77 68 61 74 5F 78 30 72 5F 31 73
> N0w_y0u_kn0w_what_x0r_1s

3.Maze

输入一个长度限定为10的字符串，下面简称str
通过[ebp + var_8]计数，下面简称i，循环10次
限定str[i] = a,s,d,w

```
mov     ecx, [ebp+var_C]
movzx   edx, ds:byte_401160[ecx]
jmp     ds:jpt_4010A1[edx*4] ; switch jump
```

控制一个变量[ebp + var_4],下面简称var_4
str[i] == 'a', var_4 += -1
str[i] == 's', var_4 += 5
str[i] == 'd', var_4 += 1
str[i] == 'w', var_4 += -5
要求var_4始终不等 3 4 5 6 9 10 12 14 15 16 19 23 24

```
mov     ecx, [ebp+var_4]
movsx   edx, ds:byte_402180[ecx]
cmp     edx, 30h ; '0'
jnz     short loc_401113
```

且位于[0,24]区间
最终要求var_4 == 20

```
loc_401118:
mov     eax, [ebp+var_4]
movsx   ecx, ds:byte_402180[eax]
cmp     ecx, 2Ah ; '*'
jnz     short loc_401135
```

从var_4 == 20开始推理?

> 0 -> 1 -> 2 -> 7 -> 8 -> 13 -> 18 -> 17 -> 22 -> 21 -> 20

对应字符串

> ddsdssasaa

加密答案

> 0e6321aa4d31bfc66b83c0406885ce86

4. Array

读入一个长度为20的字符串，下面简称str，调用 sub_401080 处的函数检查
对于两个数组，下面简称 Array1 Array2
要求 Array1[str[i]] == Array2[i]，i 从 0 循环到 19
Array1显然会越界，观察内存

```
.data:00403000                 ;org 403000h
.data:00403000 byte_403000     db 0FFh                 ; DATA XREF: sub_401080+4F↑r
.data:00403001                 db 0FFh
.data:00403002                 db 0FFh
.data:00403003                 db 0FFh
.data:00403004 dword_403004    dd 1                    ; DATA XREF: sub_40175D+2↑r
.data:00403008                 align 10h
.data:00403010 dword_403010    dd 1                    ; DATA XREF: sub_401A14+D↑w
.data:00403010                                         ; sub_401A14:loc_401B2B↑r ...
.data:00403014 ; uintptr_t __security_cookie
.data:00403014 ___security_cookie dd 0BB40E64Eh        ; DATA XREF: sub_4016AF↑r
.data:00403014                                         ; sub_4016AF:loc_4016E9↑w ...
.data:00403018 dword_403018    dd 44BF19B1h            ; DATA XREF: sub_4016AF+43↑w
.data:00403018                                         ; sub_401C2A+E4↑r
.data:0040301C dword_40301C    dd 1                    ; DATA XREF: sub_401BE8+2↑r
.data:00403020 aZyxwvutsrqponm db '~}|{zyxwvutsrqponmlkjihgfedcba`_^]\[ZYXWVUTSRQPONMLKJIHGFEDCBA@?>'
.data:00403061                 db '=<;:9876543210/.-,+*)(',27h,'&%$#"! ',0
.data:00403080 byte_403080     db 38h                  ; DATA XREF: sub_401080+7E↑r
.data:00403081 a2715099        db '2=7#+1;?50:9&?9=+%!',0
.data:00403095                 align 4
```

可推知要求字符串为

> flag{smc_index_easy}

5. Rome

存储一个字符串，下面简称 Des
Des = "Zxb3xo_qe4_Dob@q"
读入一个长度为16的字符串，下面简称str
令计数变量 [ebp+var_4]，下面简称i，从0开始循环16次
若 str[i] 不是英文字母，直接比较 str[i] 与 Des[i]，若相等 i++
若str[i] 是大写英文字母，str[i] - 'D' (若小于0 +26直到大于等于0) + 'A' 与 Des[i] 比较，若相等 i++
若str[i] 是小写英文字母，操作类似

```
loc_BD1197:
mov     edx, [ebp+var_4]
movsx   eax, [ebp+edx+var_A0]
mov     ecx, [ebp+var_4]
movsx   edx, [ebp+ecx+Destination]
cmp     eax, edx
jz      short loc_BD11C3
```

最终要求 i >= 16
推算知答案为

> Cae3ar_th4_Gre@t

6. Equation

函数 sub_140001B80 ， 猜测是输入函数，经调试确实是输入函数?
读入一个长度为32的字符串，下面简称str
通过阅读 loc_140001099 这一段冗长而乏味的汇编代码
可列关于 str[i]，i ∈ {0,1,2,3} 的四元一次方程组

$$\begin{cases} 12a + 20b + 17c + 9d = 4518 \\ 7a + 9b + 7c + 8d = 2422 \\ 13a + 12b + 15c + 5d = 3481 \\ 19a + 11b + 19c + 16d = 5006 \end{cases}$$

求解 a, b, c, d 的值

$$a = 81$$
$$b = 84$$
$$c = 69$$
$$d = 77$$

随后向 qword_140004040 数组中存入数据以验证str后28位，下面简称qword
代码大意

```
for (int i = 0;i < 28;i++) {
    if (str[i % 4] ^ str[i + 4] != qword[2 * i])
        break; // Wrong
}
```

求解代码

```
#include<iostream>
#include<cstdio>

using namespace std;

int Array[56] = {...};

int str[4] = {81,84,69,77};

int main()
{
    for (int i = 0;i < 28;i++)
        cout << (char)(Array[2 * i] ^ str[i % 4]);
    cout << endl;
    return 0;
}
```

最终答案

> QTEM5D91sCjyBGNvNOJMOQ7q3KsINzwr

7. Click

没有发现输入函数
但是注意到程序存储了五个字符串

```
lea     rdx, aAhfrcwstmje4mj ; "aHFrcWstMjE4MjMtamNoZGts"
call    cs:??0QString@@QEAA@PEBD@Z ; QString::QString(char const *)
nop
lea     rcx, [rbx+50h]
lea     rdx, aYwjjzgutmzg0od ; "YWJjZGUtMzg0ODMta2RraHly"
call    cs:??0QString@@QEAA@PEBD@Z ; QString::QString(char const *)
nop
;   try {
lea     rcx, [rbx+68h]
lea     rdx, aYwjjzgutmtiznd ; "YWJjZGUtMTIzNDUtZ2hpamts"
call    cs:??0QString@@QEAA@PEBD@Z ; QString::QString(char const *)
nop
;   } // starts at 7FF75B7710FC
;   try {
lea     rcx, [rbx+80h]
lea     rdx, aEhh4exktmtiznd ; "eHh4eXktMTIzNDUtamtvcG1w"
call    cs:??0QString@@QEAA@PEBD@Z ; QString::QString(char const *)
nop
;   } // starts at 7FF75B77110E
;   try {
lea     rcx, [rbx+98h]
lea     rdx, aUxrmdw4tmtawod ; "UXRmdW4tMTAwODYtR1VJdG9v"
call    cs:??0QString@@QEAA@PEBD@Z ; QString::QString(char const *)
nop
;   } // starts at 7FF75B771123
```

以及一个可疑的随机数函数

```
lea     rcx, [rsp+48h+var_28] ; void *
call    cs:__imp_??1QString@@QEAA@XZ ; QString::~QString(void)
lea     rcx, [rsp+48h+arg_10]
call    cs:?currentTime@QTime@@SA?AV1@XZ ; QTime::currentTime(void)
mov     rcx, rax
call    cs:?msec@QTime@@QEBAHXZ ; QTime::msec(void)
mov     ecx, eax        ; Seed
call    cs:srand
call    cs:rand
and     eax, 80000003h
jge     short loc_7FF75B7711C1
```

根据题目提示，查找 0x3e8 (1000)
调试时通过修改ZF = 1

```
cmp     dword ptr [rdi+30h], 3E8h
jnz     loc_7FF75B7715C2
```

得到假的flag
注意到一个函数

```
call    cs:?fromBase64@QByteArray@@SA?AV1@AEBV1@V?$QFlags@W4Base64Option... ;
QByteArray::fromBase64(QByteArray const &,QFlags<QByteArray::Base64Option>)
```

猜测假flag可能与之前五个字符串存在Base64加密关系
检验正确

> abcde-12345-ghijkl --> YWJjZGUtMTIzNDUtZ2hpamts

但经测试只有4个flag出现，分别由4个字符串Base64解码而来
猜测 UXRmdW4tMTAwODYtR1VJdG9v 解码可得真flag

> Qtfun-10086-GUItoo

检验正确

8. Junkcode

汇编代码中存在两句花指令，修改为nop

```
.text:007313C3                jz      short loc_7313C8
.text:007313C5                jnz     short loc_7313C8
.text:007313C7                nop


.text:007313F6                jz      short near ptr loc_7313F9+2
.text:007313F8                nop
```

之后代码可读
读入一个字符串到 [ebp-50h]，下面简称str
通过调用 sub_731160 第一次加密到 [ebp-0B4h]，下面简称B4
计数器 i 从0循环4次，每次从 B4[11 * i] 开始调用 sub_731000 加密，密文拷贝到 [ebp+ 11 * i -118h]
最终比较 [ebp-118h] 与 "P1Ekb1UxW9ErWC6ZVUKiKgMaLSEgS5gpyZOrSQG3tP8g" 判断flag是否正

确

首先逆向分析 sub_731000 ， 显然是一个简单的恺撒密码，求解代码如下

```cpp
#include<iostream>
#include<cstdio>
#include<string>
#define PerSize 11

using namespace std;

string Str = "P1Ekb1UxW9ErWC6ZVUKiKgMaLSEgS5gpyZOrSQG3tP8g";
string ans;

int main()
{
    ans.resize(Str.size());
    int sig = 1;
    int id = 0;
    for (int i = 0;i < 4;i++) {
      for (int j = id;j < id + PerSize;j++) {
          if (Str[j] < 'A' || Str[j] > 'Z') {
              if (Str[j] < 'a' || Str[j] > 'z') {
                  if (Str[j] < '0' || Str[j] > '9') {
                      ans[j] = Str[j];
                      cout << "not num or alpha" << endl;
                  }
                  else {
                      cout << "num" << endl;
                      if (Str[j] - '0' <= 9 - sig)
                          ans[j] = Str[j] + sig;
                      else
                          ans[j] = Str[j] - 10 + sig;
                  }
              }
              else {
                  cout << "alpha" << endl;
                  if (Str[j] - 'a' <= 25 - sig)
                      ans[j] = Str[j] + sig;
                  else
                      ans[j] = Str[j] - 26 + sig;
              }
          }
          else {
              cout << "ALPHA" << endl;
              if (Str[j] - 'A' <= 25 - sig)
                  ans[j] = Str[j] + sig;
```

```
                else
                    ans[j] = Str[j] - 26 + sig;
            }
        }
        sig += 2;
        id += PerSize;
    }
    cout << ans << endl;
    cout << ans.size() << endl;
    int size = ans.size();
    for (int i = 0;i < size;i++)
        printf("%02x ",ans[i]);
    cout << endl;

    return 0;
}
```

第二次加密的明文，也是第一次加密的密文为

> "Q2Flc2VyX0FuZF9CYXNlNjRfQXJlX0ludGVyZXN0aW5n"

分析第一个加密函数，大意为通过对输入字符的位操作索引到内存中的一个数组（简称lib）来获得密文，由之后的flag可知这是一个Base64加密算法

```cpp
#include<iostream>
#include<cstdio>
#include<string>
#include<unordered_map>

using namespace std;

string Target_Str = "Q2Flc2VyX0FuZF9CYXNlNjRfQXJlX0ludGVyZXN0aW5n";
string ans;
string lib = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";

unordered_map<char,int> lib_map;

void init_lib_map() {
    for (int i = 0;i < lib.size();i++)
        lib_map[lib[i]] = i;
}

int main()
{
    int id = 0,sig = 0;
    int ans_i = 0;
    ans.resize(Target_Str.size());
    int size = Target_Str.size();
    cout << size << endl;
    init_lib_map();
    int mask_3 = 3 << 4;
    int mask_f = 0xf << 2;
    for (ans_i = 0;ans_i < size;ans_i++) {
        if (id >= size)
            break;
        if (sig) {
            if (sig == 1) {
                sig = 2;
                int tar_id = lib_map[Target_Str[id]];
                if ((tar_id & mask_3) > 0) {
                    ans[ans_i - 1] += ((tar_id & mask_3) >> 4); // 获得sig==0时的低2位
                    tar_id = tar_id & 0xf;
                }
                char ans_ch = tar_id << 4; // 仅获得高4位
                ans[ans_i] = ans_ch;
            }
            else {
```

```
                sig = 0;
                int tar_id = lib_map[Target_Str[id]];
                if ((tar_id & mask_f) > 0) {
                    ans[ans_i - 1] += ((tar_id & mask_f) >> 2); // 获得sig==1时的低4位
                    tar_id = tar_id & 0x3;
                }
                char ans_ch = tar_id << 6; // 仅获得高2位
                ans[ans_i] = ans_ch;
                id++;
                int next_tar_id = lib_map[Target_Str[id]];
                ans[ans_i] += (next_tar_id & 0x3f); // 获得低6位
                ans_ch = ans[ans_i];
                if (Target_Str[id] != lib[ans_ch & 0x3f])
                    puts("error");
            }
            id++;
        }
        else {
            sig = 1;
            int tar_id = lib_map[Target_Str[id]];
            if (tar_id > 0x3f)
                puts("error");
            else {
                char ans_ch = tar_id << 2;
                ans[ans_i] = ans_ch; // 仅获得高6位
                id++;
            }
        }
    }
    cout << sig << endl;
    cout << ans << endl;
    return 0;
}
```

获得明文

> Caeser_And_Base64_Are_Interesting

9. Multithreading

这一题有很多花指令
找到IDA标红的部分一一修改
首先是一些简单的 call，jmp指令造成的code xref，对于这一类花指令修改字节为0x90(nop) 即可，由于

太多就不依次放图了
然后是两处 sp-analysis failed
观察堆栈指针位置，发现出现堆栈不平衡的问题
第一处修改比较简单，修改指针位置使得 retn 一行 stack pointer 为 000，观察局部变量数量，修改函数的 Local variables area 为 0x10 即可

```
.text:004010E0     sub_4010E0      proc near               ; DATA XREF: .text:0040112F↓o
.text:004010E0                                             ; .text:00401143↓o
.text:004010E0 000                 push    ebp
.text:004010E1 004                 mov     ebp, esp
.text:004010E3 004                 push    ebx
.text:004010E4 008                 push    esi
.text:004010E5 00C                 push    edi
.text:004010E6 010                 mov     ebx, large fs:30h
.text:004010ED 010                 movzx   ebx, byte ptr [ebx+2]
.text:004010F1 010                 xor     eax, eax
.text:004010F3 010                 jnz     short loc_4010FA
.text:004010F5 010                 call    loc_4010FB
.text:004010FA
.text:004010FA     loc_4010FA:                             ; CODE XREF: sub_4010E0+13↑j
.text:004010FA 010                 nop
.text:004010FB
.text:004010FB     loc_4010FB:                             ; CODE XREF: sub_4010E0+15↑p
.text:004010FB 010                 pop     eax
.text:004010FC 00C                 add     ebx, 9
.text:004010FF 00C                 add     eax, ebx
.text:00401101 00C                 push    eax
.text:00401102 000                 retn
.text:00401102     sub_4010E0      endp
```

第二处较为复杂，出现了一个奇怪的jnz

```
.text:004011C1 01C                 jnz     short loc_4011C8
.text:004011C3 01C                 call    loc_4011C9
.text:004011C8
.text:004011C8     loc_4011C8:                             ; CODE XREF: StartAddress+A1↑j
.text:004011C8 018                 nop
```

这一行跳转指令永远不会发生，但接下来堆栈指针发生了莫名其妙的变动
尝试将其修改为 nop，下面的数据转换为代码后出现了一些奇怪的内存访问操作
同时注意到下面一段位于loc_4011F1的代码有一个跳转指向，这一段代码将一个类似循环计数器的变量 [ebp+var_4] 与0x2A比较后跳转，值得注意的是主函数读入字符串时指定长度为42，即0x2A，猜测 loc_4011F1 是一段有用的代码

```
.text:00401167 01C                 cmp     [ebp+var_4], 2Ah ; '*'
.text:0040116B 01C                 jge     loc_4011F1
```

但函数end地址在其上方，即无法访问这段代码
且end地址下方有一段 jmp loc_40115E 的代码，loc_40115E 代码段执行了对 [ebp+var_4] （怀疑是循坏计数器）的自增操作，与循环体相关的汇编代码类似，这一段代码也是看似有用但无法被访问的

```
.text:004011D3 01C                inc     eax
.text:004011D5 01C                dec     eax
.text:004011D6 01C                mov     ecx, [ebp+var_4]
.text:004011D9 01C                movzx   edx, byte_40336C[ecx]
.text:004011E0 01C                add     edx, 23h ; '#'
.text:004011E3 01C                mov     eax, [ebp+var_4]
.text:004011E6 01C                mov     byte_40336C[eax], dl
.text:004011EC 01C                jmp     loc_40115E
```

猜想①：刚才nop掉的jnz指令到函数end地址之间的数据为花指令
这一永远不会实现的跳转误导了IDA的反汇编
全部nop后设置函数end地址到 .text:004011F9，即loc_4011F1下方

```
.text:004011AC 01C                push    6               ; dwMilliseconds
.text:004011AE 020                call    ds:Sleep
.text:004011B4 01C                mov     ebx, large fs:30h
.text:004011BB 01C                movzx   ebx, byte ptr [ebx+2]
.text:004011BF 01C                xor     eax, eax
.text:004011C1 01C                nop
.text:004011C2 01C                nop
.text:004011C3 01C                nop
.text:004011C4 01C                nop
.text:004011C5 01C                nop
.text:004011C6 01C                nop
.text:004011C7 01C                nop
.text:004011C8 01C                nop
.text:004011C9 01C                nop
.text:004011CA 01C                nop
.text:004011CB 01C                nop
.text:004011CC 01C                nop
.text:004011CD 01C                nop
.text:004011CE 01C                nop
.text:004011CF 01C                nop
.text:004011D0 01C                nop
.text:004011D1 01C                nop
.text:004011D2 01C                nop
.text:004011D3 01C                inc     eax
.text:004011D5 01C                dec     eax
.text:004011D6 01C                mov     ecx, [ebp+var_4]
.text:004011D9 01C                movzx   edx, byte_40336C[ecx]
.text:004011E0 01C                add     edx, 23h ; '#'
.text:004011E3 01C                mov     eax, [ebp+var_4]
.text:004011E6 01C                mov     byte_40336C[eax], dl
.text:004011EC 01C                jmp     loc_40115E
.text:004011F1                    ; ---------------------------------------------------------------------------
.text:004011F1
.text:004011F1                    loc_4011F1:                             ; CODE XREF: StartAddress+4B↑j
.text:004011F1 01C                xor     eax, eax
.text:004011F3 01C                pop     edi
.text:004011F4 018                pop     esi
.text:004011F5 014                pop     ebx
.text:004011F6 010                mov     esp, ebp
.text:004011F8 004                pop     ebp
.text:004011F9 000                retn    4
.text:004011F9                    StartAddress    endp
```

堆栈指针自然平衡且逻辑自洽

分析加密算法，发现有两个线程并行对读入字符串（简称str）写入，这显然是不可解密的
但是注意到其中一个线程出现了函数

> NtCurrentPeb()->BeingDebugged

猜想在非debug模式下该值为0，该线程实际上令str[i] += 0, 并未修改字符串
只关注另一个线程，解密代码如下

```
#include<iostream>

using namespace std;

int Tar[42] = {0xdd,0x5b,0x9e,0x1d,0x20,0x9e,0x90,0x91,0x90,0x90,0x91,0x92,0xde,0x8b,0x11,0xd1,(

char ans[42];

int main()
{
    for (int i = 0;i < 42;i++) {
        int tmp = Tar[i] - 35;
        tmp ^= 0x23u;
        int low = (tmp >> 6) & 3;
        tmp = (tmp << 2) + low;
        ans[i] = tmp;
    }
    cout << ans << endl;
}
```

获得答案

> flag{a959951b-76ca-4784-add7-93583251ca92}

检验正确，说明猜想①亦正确

10. Babyvm

一道附加题的汇编代码过少，显然是反常识的
事实上也是不正确的，因为在十六进制View中发现了很多与并未在汇编代码里出现的字符
对于这种奇怪的现象，结合题目提示中的“字节码”云云，查阅资料知，程序可能经过了一种名为加壳的操作
另外注意到 text view 下，指令地址并没有像前面几题一样显示 ".text:"，而是 "UPX1"

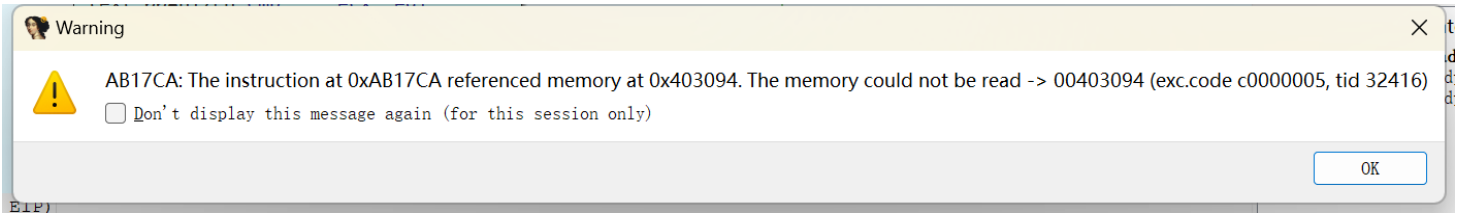
查阅资料知，这是一种压缩壳，需要使用 UPX 工具脱壳
下载 UPX 脱壳后分析代码，获得加密算法，解密代码如下

```
#include<iostream>

using namespace std;

int Tar[24] = {0x7e,0x78,0x75,0x7f,0x6b,0x52,0x75,0x72,0x6d,0x77,0x4e,0x79,0x79,0x79,0x77,0x44,0
char Ans[24];

int main()
{
    for (int i = 0;i < 24;i++) {
        int ans = (Tar[i] ^ 0x16) - 2;
        if (ans < 0x20 || ans > 0x7e)
            cout << "Error" << endl;
        Ans[i] = ans;
    }
    cout << Ans << endl;
    return 0;
}
```

一道附加题的加密算法过于简单，也是反常识的（虽然第9题的加密算法也不难）
但是输出的 flag 实在是太像正确答案了

> flag{Baby_Vmmm_Pr0tect!}

脱壳之后运行 exe 出现了内存非法访问，原因未知



无所谓，春秋云境会帮我评测
果然是正确的

End