REVERSE

逆向工程

逆向环境和工具

IDA :
https://pan.baidu.com/s/15k25qBem2XozN_AdKTVfAg?pwd=et1i

Exeinfo PE:
https://www.lanzouw.com/iL8nh1z2kokd

X64dbg:
https://github.com/x64dbg/x64dbg/releases/tag/snapshot

Jadx:
https://github.com/skylot/jadx/releases

汇编语言基础

- AX是主要的累加器；它用于输入/输出和大多数算术指令。例如，在乘法运算中，根据操作数的大小，将一个操作数存储在EAX或AX或AL寄存器中。

- BX被称为基址寄存器，因为它可以用于索引寻址。

- CX被称为计数寄存器，因为ECX，CX寄存器在迭代操作中存储循环计数。

- DX被称为数据寄存器。它也用于输入/输出操作。它还与AX寄存器以及DX一起使用，用于涉及大数值的乘法和除法运算。

汇编语言基础

- AX是主要的累加器；它用于输入/输出和大多数算术指令。例如，在乘法运算中，根据操作数的大小，将一个操作数存储在EAX或AX或AL寄存器中。

- BX被称为基址寄存器，因为它可以用于索引寻址。

- CX被称为计数寄存器，因为ECX，CX寄存器在迭代操作中存储循环计数。

- DX被称为数据寄存器。它也用于输入/输出操作。它还与AX寄存器以及DX一起使用，用于涉及大数值的乘法和除法运算。

汇编语言基础

- 溢出标志（OF） -指示有符号算术运算后数据的高阶位（最左位）的溢出。
- 方向标记（DF） -它确定向左或向右移动或比较字符串数据的方向。DF值为0时，字符串操作为从左至右的方向；当DF值为1时，字符串操作为从右至左的方向。
- 中断标志（IF） -确定是否忽略或处理外部中断（例如键盘输入等）。当值为0时，它禁用外部中断，而当值为1时，它使能中断。
- 陷阱标志（TF） -允许在单步模式下设置处理器的操作。我们使用的DEBUG程序设置了陷阱标志，因此我们可以一次逐步执行一条指令。
- 符号标志（SF） -显示算术运算结果的符号。根据算术运算后数据项的符号设置此标志。该符号由最左位的高位指示。正结果将SF的值清除为0，负结果将其设置为1。
- **零标志（ZF）** -指示算术或比较运算的结果。非零结果将零标志清零，零结果将其清零。
- 辅助进位标志（AF） -包含经过算术运算后从位3到位4的进位；用于专业算术。当1字节算术运算引起从第3位到第4位的进位时，将设置AF。
- 奇偶校验标志（PF） -指示从算术运算获得的结果中1位的总数。偶数个1位将奇偶校验标志清为0，奇数个1位将奇偶校验标志清为1。
- 进位标志（CF） -在算术运算后，它包含一个高位（最左边）的0或1进位。它还存储移位或旋转操作的最后一位的内容。

汇编语言基础

通用数据传送指令：MOV、MOVSX、MOVZX、PUSH、POP

输入输出端口传送指令：IN、OUT

目的地址传送指令：LEA、LDS

标志传送指令：LAHF、SAHF、PUSHF、POPF

算术运算指令：ADD、SUB、INC、DEC、MUL、DIV

逻辑运算指令：AND、OR、XOR、NOT、TEST、CMP

跳转指令：JMP、JZ、JNZ、JC、JNC、JA、JAE、JB、JBE、JO、JNO、JS、JNS、JP、JNP、JE、JNE

其他常用指令：NOP、HLT、INT、IRET、CLC、STC、CLI、STI、ESC、WAIT、LOCK

汇编语言基础

通用数据传送指令：MOV、MOVSX、MOVZX、PUSH、POP

输入输出端口传送指令：IN、OUT

目的地址传送指令：LEA、LDS

标志传送指令：LAHF、SAHF、PUSHF、POPF

算术运算指令：ADD、SUB、INC、DEC、MUL、DIV

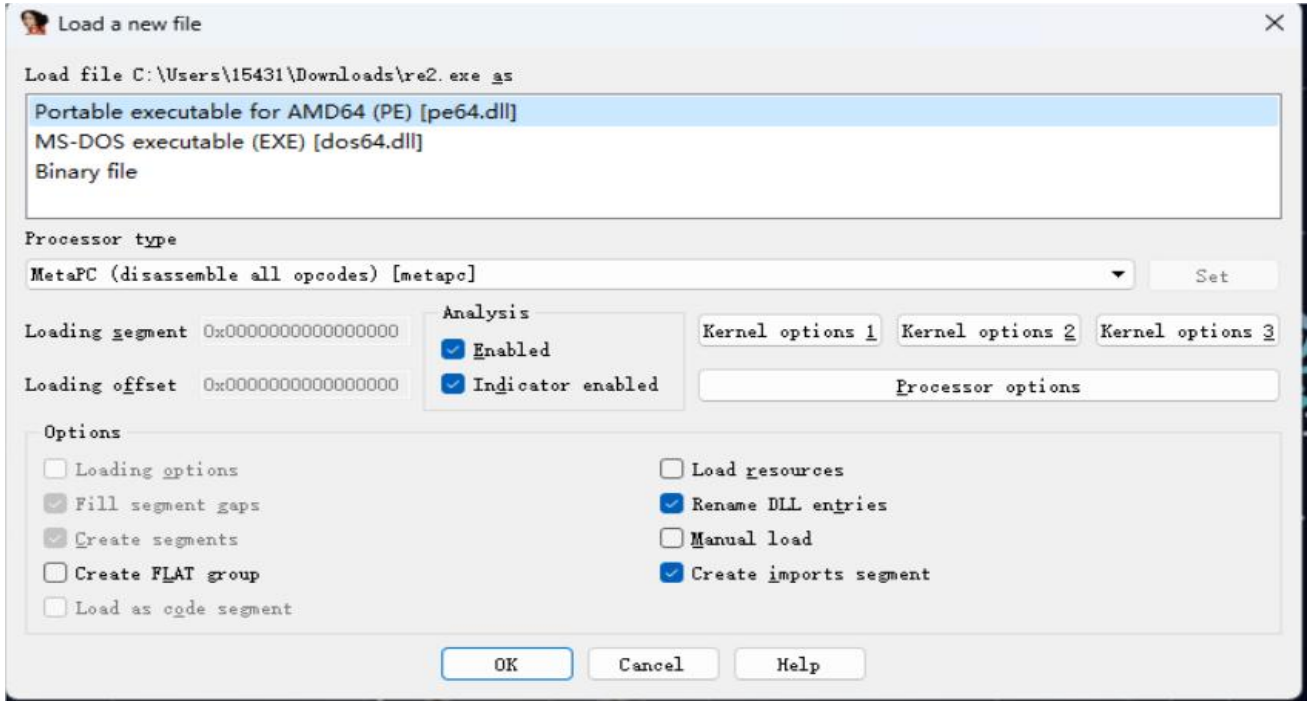
逻辑运算指令：AND、OR、XOR、NOT、TEST、CMP

跳转指令：JMP、JZ、JNZ、JC、JNC、JA、JAE、JB、JBE、JO、JNO、JS、JNS、JP、JNP、JE、JNE

其他常用指令：NOP、HLT、INT、IRET、CLC、STC、CLI、STI、ESC、WAIT、LOCK

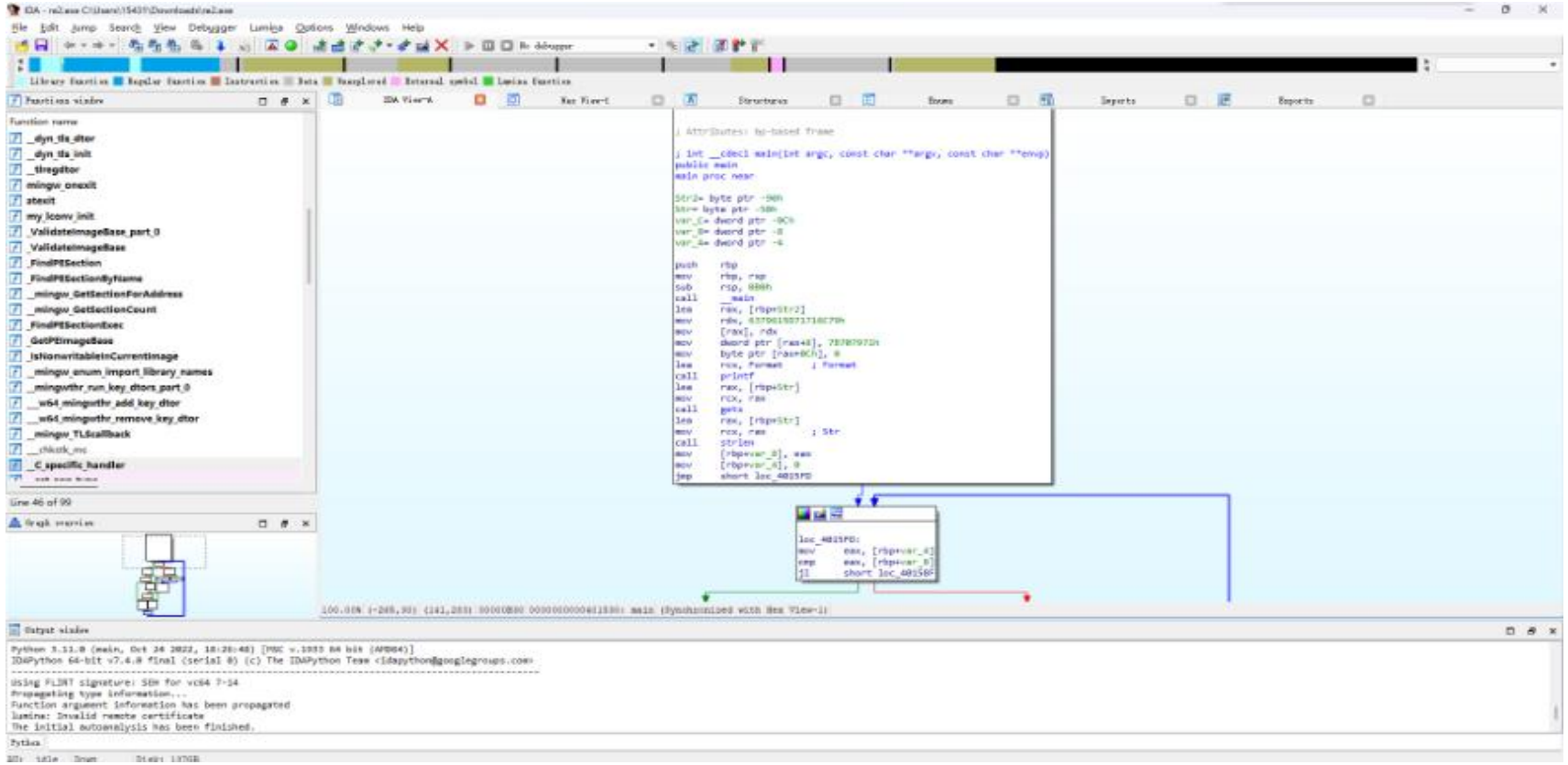
IDA 使用

最简单的使用方式是将待逆向的二进制可执行文件直接拖放到 IDA 上，IDA 会自动识别文件类型，这一步通常不需要做改动，直接点击 OK 即可：



IDA 使用

接下来我们就会看到这样一个界面：



IDA 使用

介绍一下几个窗口

Function Windows ： IDA 所识别出来的函数列表，通过该目录可以直接跳转到对应函数

IDA-View ： 以汇编形式呈现的由 IDA 进行反编译所得的单个函数结果，默认是以由基本块构成的控制流图的形式进行展示，也可以通过 空格 键切换到内存布局中的原始汇编代码

Hex View ： 二进制文件的原始数据视图

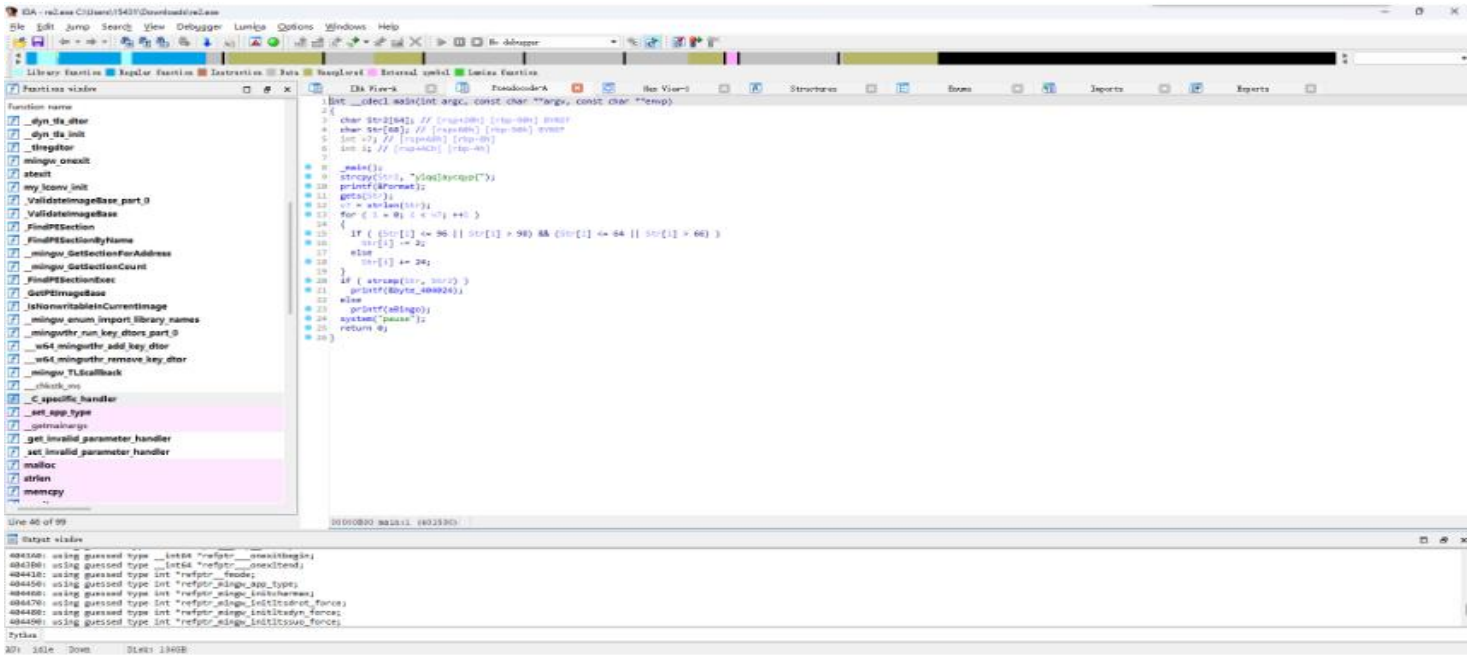
Structures ： 由 IDA 所识别出来的程序中可能存在的结构体信息

Imports ： 该二进制文件运行时可能需要从外部导入的符号

Exports ： 该二进制文件可以被导出到外部的符号

IDA 使用

除了反汇编以外，IDA 也支持将汇编代码反编译为 C/C++ 形式的源代码，我们只需要在待反编译的函数位置按下 F5 即可获得反编译后的程序代码：



IDA 使用

信息收集:

1.shift+f12 定位关键字符串

2.Function 查看主函数 一般为main函数

3.查看调用堆栈（使用较少）

4.动态调试获取逻辑

Exeinfo PE使用

拖入即可

主要观察壳的信息

花指令

花指令：目的是干扰ida和od等软件对程序的静态分析。使这些软件无法正常反汇编出原始代码。

常用的两类反汇编算法：

线性扫描算法：逐行反汇编（无法将数据和内容进行区分）

递归行进算法：按照代码可能的执行顺序进行反汇编程序。

1.jmp

2.jnz和jz条件跳转

3.永真条件跳转

4.call&ret构造花指令

花指令

1.jmp

```
jmp Label1
  db thunkcode1;垃圾数据
Label1:
```

花指令

2.jnx和jx

```
asm
{
  Jz Label
  Jnz Label
  Db thunkcode;垃圾数据
Label:
}
```

花指令

3.永真条件跳转

```
__asm {
    push ebx;
    xor ebx, ebx;
    test ebx, ebx;
    jnz LABEL7;
    jz    LABEL8;
LABEL7:
    _emit 0xC7;
LABEL8:
    pop ebx;
}
```

花指令

4.call&ret

```
__asm {
    call LABEL9;
    _emit 0x83;
LABEL9:
    add dword ptr ss : [esp] , 8;
    ret;
    __emit 0xF3;
}
```

花指令题目练习

基础题：

[NSSRound#3 Team]jump_by_jump

[HNCTF 2022 WEEK2]e@sy_flower

[MoeCTF 2022]chicken_soup

进阶题：

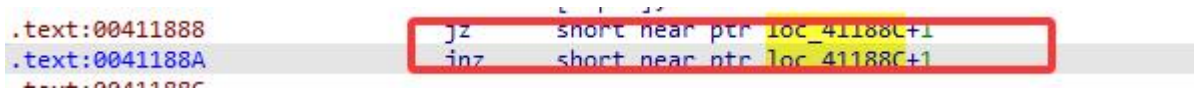
Wordy：利用idapython批量解花指令

[SWPUCTF 2023 秋季新生赛]Junk Code:利用idapython 批量解花指令

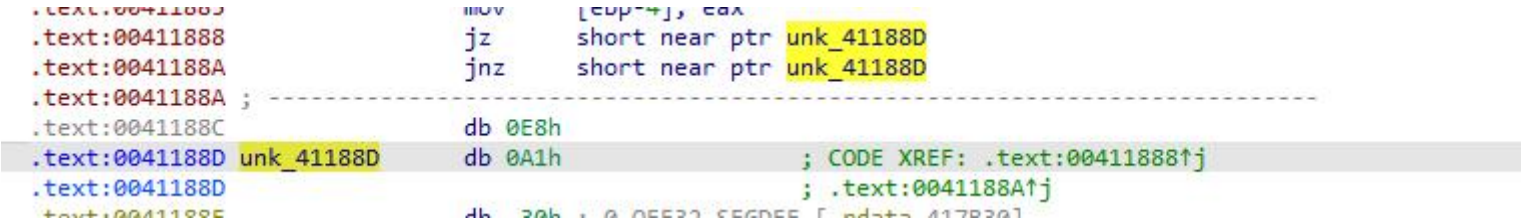
花指令题目练习

[NSSRound#3 Team]jump_by_jump

明显的构造永真条件跳转花指令 无论怎么样都会跳转到 41188c+1的地址



只需要将从411888地址到41188c+1全部nop掉



花指令题目练习

[NSSRound#3 Team]jump_by_jump

```
.text:00411860                push    ebp
.text:00411861                mov     ebp, esp
.text:00411863                sub     esp, 0F0h
.text:00411869                push    ebx
.text:0041186A                push    esi
.text:0041186B                push    edi
.text:0041186C                lea     edi, [ebp+var_F0]
.text:00411872                mov     ecx, 3Ch
.text:00411877                mov     eax, 0CCCCCCCCh
.text:0041187C                rep stosd
.text:0041187E                mov     eax, ___security_cookie
.text:00411883                xor     eax, ebp
.text:00411885                mov     [ebp+var_4], eax
.text:00411888                nop
.text:00411889                nop
.text:0041188A                nop
.text:0041188B                nop
.text:0041188C                nop
.text:0041188D                mov     eax, ds:dword_417B30
.text:00411892                mov     dword ptr [ebp+var_20], eax
```

修复后效果 可以直接得到flag

```
1  int __cdecl main_0(int argc, const char **argv, const char **envp)
2  {
3    int i; // [esp+D0h] [ebp-2Ch]
4    char v5[28]; // [esp+DCh] [ebp-20h] BYREF
5
6    strcpy(v5, "NSSCTF{Jump_b9_jump!}");
7    for ( i = 0; i < 21; ++i )
8      v5[i] = (v5[i] + v5[(i * i + 123) % 21]) % 128;
9    sub_4110CD("%s", (char)v5);
10   return 0;
11 }
```

花指令题目练习

[HNCTF 2022 WEEK2]e@sy_flower

```
.text:004010CB                  sub     eax, edx
.text:004010CD                  mov     [ebp-38h], eax
.text:004010D0                  jz      short near ptr loc_4010D4+1
.text:004010D2                  jnz     short near ptr loc_4010D4+1
.text:004010D4
.text:004010D4 loc_4010D4:                              ; CODE XREF: .text:004010D0↑j
.text:004010D4                                          ; .text:004010D2↑j
.text:004010D4                  jmp     near ptr 9A085664h
.text:004010D9 ; ---------------------------------------------------------------------------
.text:004010D9                  sub     eax, edx
```

仍然是构造恒真跳转 继续nop

```
.text:004010D0                  jz      short near ptr unk_4010D5
.text:004010D2                  jnz     short near ptr unk_4010D5
.text:004010D2 ; ---------------------------------------------------------------------------
.text:004010D4                  db 0F9h
.text:004010D5 unk_4010D5       db  8Bh                 ; CODE XREF: .text:004010D0↑j
.text:004010D5                                          ; .text:004010D2↑j
.text:004010D6                  db  45h ; E
.text:004010D7                  db 0C8h
.text:004010D8                  db  99h
.text:004010D9 ; ---------------------------------------------------------------------------
```

花指令题目练习

[HNCTF 2022 WEEK2]e@sy_flower

修复后效果

```
int __cdecl __noreturn main(int argc, const char **argv, const char **envp)
{
  signed int v3; // kr00_4
  int i; // edx
  char v5; // cl
  unsigned int j; // edx
  int v7; // eax
  char v8; // [esp+0h] [ebp-44h]
  char v9; // [esp+0h] [ebp-44h]
  char Arglist[48]; // [esp+10h] [ebp-34h] BYREF

  sub_401020("please input flag\n", v8);
  sub_401050("%s", Arglist);
  v3 = strlen(Arglist);
  for ( i = 0; i < v3 / 2; ++i )
  {
    v5 = Arglist[2 * i];
    Arglist[2 * i] = Arglist[2 * i + 1];
    Arglist[2 * i + 1] = v5;
  }
  for ( j = 0; j < strlen(Arglist); ++j )
    Arglist[j] ^= 0x30u;
  v7 = strcmp(Arglist, "c~scvdzKCEoDEZ[^roDICUMC");
  if ( v7 )
    v7 = v7 < 0 ? -1 : 1;
  if ( !v7 )
  {
    sub_401020("yes", v9);
    exit(0);
  }
  sub_401020("error", v9);
  exit(0);
}
```

只需要异或和换位就可以得到flag

花指令题目练习

[MoeCTF 2022]chicken_soup

主函数

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char v4[100]; // [esp+10h] [ebp-68h] BYREF

  puts("I poisoned the program... Can you reverse it?!");
  puts("Come on! Give me your flag:");
  sub_4012A0("%s", v4);
  if ( strlen(v4) == 38 )
  {
    (loc_401000)(v4);
    (loc_401080)(v4);
    if ( sub_401110(v4, &unk_403000) )
      puts("\nTTTTTTTTTTQQQQQQQQQQQQQLLLLLLLLL!!!!");
    else
      puts("\nQwQ, please try again.");
    return 0;
  }
  else
  {
    puts("\nQwQ, please try again.");
    return 0;
  }
}
```

调用了两个其他函数

花指令题目练习

[MoeCTF 2022]chicken_soup

主函数

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char v4[100]; // [esp+10h] [ebp-68h] BYREF

  puts("I poisoned the program... Can you reverse it?!");
  puts("Come on! Give me your flag:");
  sub_4012A0("%s", v4);
  if ( strlen(v4) == 38 )
  {
    (loc_401000)(v4);
    (loc_401080)(v4);
    if ( sub_401110(v4, &unk_403000) )
      puts("\nTTTTTTTTTTQQQQQQQQQQQQLLLLLLLLL!!!!");
    else
      puts("\nQwQ, please try again.");
    return 0;
  }
  else
  {
    puts("\nQwQ, please try again.");
    return 0;
  }
}
```

调用了两个其他函数 依旧是这种构造手法
继续修复

```
.text:00401000
.text:00401000 sub_401000      proc near               ; CODE XREF: _main+85↓p
.text:00401000
.text:00401000 var_14          = dword ptr -14h
.text:00401000 var_10          = dword ptr -10h
.text:00401000 var_C           = dword ptr -0Ch
.text:00401000 var_8           = dword ptr -8
.text:00401000 var_1           = byte ptr -1
.text:00401000 arg_0           = dword ptr  8
.text:00401000
.text:00401000                 push    ebp
.text:00401001                 mov     ebp, esp
.text:00401003                 sub     esp, 14h
.text:00401006                 push    ebx
.text:00401007                 push    esi
.text:00401008                 push    edi
.text:00401009                 jz      short loc_40100E
.text:0040100B                 jnz     short loc_40100E
.text:0040100D                 nop
.text:0040100E
.text:0040100E loc_40100E:                             ; CODE XREF: sub_401000+9↑j
.text:0040100E                                         ; sub_401000+B↑j
.text:0040100E                 mov     [ebp+var_8], 0
.text:00401015                 jmp     short loc_401020
.text:00401017 ; ---------------------------------------------------------------------------
.text:00401017
.text:00401017 loc_401017:                             ; CODE XREF: sub_401000+72↓j
.text:00401017                 mov     eax, [ebp+var_8]
.text:0040101A                 add     eax, 1
.text:0040101D                 mov     [ebp+var_8], eax
.text:00401020
.text:00401020 loc_401020:                             ; CODE XREF: sub_401000+15↑j
.text:00401020                 mov     ecx, [ebp+arg_0]
.text:00401023                 mov     [ebp+var_C], ecx
.text:00401026                 mov     edx, [ebp+var_C]
.text:00401029                 add     edx, 1
```

花指令题目练习

[MoeCTF 2022]chicken_soup

得到的结果

```
unsigned int __cdecl sub_401000(const char *a1)
{
  unsigned int result; // eax
  unsigned int i; // [esp+18h] [ebp-8h]

  for ( i = 0; ; ++i )
  {
    result = strlen(a1) - 1;
    if ( i >= result )
      break;
    a1[i] += a1[i + 1];
  }
  return result;
}
```

```
  for ( i = 0; ; ++i )
  {
    result = i;
    if ( i >= strlen(a1) )
      break;
    a1[i] = (16 * a1[i]) | (a1[i] >> 4);
  }
  return result;
}
```

花指令题目练习

[MoeCTF 2022]chicken_soup

只需要将密文位移和倒着减回去

```
a=[0xCD, 0x4D, 0x8C, 0x7D, 0xAD, 0x1E, 0xBE, 0x4A, 0x8A, 0x7D, 0xBC, 0x7C, 0xFC, 0x2E, 0x2A, 0x79, 0x9D, 0x6A, 0x1A, 0xCC, 0x3D, 0x4A, 0xF8, 0x3C, 0x79, 0x69, 0x39, 0xD9, 0xDD, 0x9D, 0xA9, 0x69, 0x4C, 0x8C, 0xDD, 0x59, 0xE9, 0xD7]

for i in range(len(a)):
    a[i] = a[i] <<4 | a[i] >>4
for i in range(len(a)-2,-1,-1):
    a[i] = a[i] -a[i+1]
print(''.join([chr(x&0xff) for x in a]))
```

花指令题目练习

[SWPUCTF 2023 秋季新生赛]Junk Code

但是这次的花指令变得比较多　查看特征

```
:xt:0000000140001583                jz      short near ptr loc_140001587+1
:xt:0000000140001585                jnz     short near ptr loc_140001587+1
:xt:0000000140001587
:xt:0000000140001587 loc_140001587:                         ; CODE XREF: main+133↑j
:xt:0000000140001587                                        ; main+135↑j
:xt:0000000140001587                jmp     near ptr 1B2C34ED4h
:xt:0000000140001587 ; ---------------------------------------------------------------------------
:xt:000000014000158C                db  80h
:xt:000000014000158D ; ---------------------------------------------------------------------------
```

显示机器码 就是74 3 后面3的意思就是当前地址+3

```
0000000140001583                db 74h
0000000140001584                db    3
0000000140001585 ; ---------------------------------------------------------------------------
0000000140001585                jnz     short near ptr loc_140001587+1
```

下面那一条也一样 我们只需要匹配 74 3 75 1 E9这组十六进制然后 全部nop

花指令题目练习

[SWPUCTF 2023 秋季新生赛]Junk Code

如何nop?

这里我们使用ida自带的idapython

IDA-Python 就是「把 IDA Pro 的 C/C++ 接口打包成 Python 脚本」，让你用几行 Python 就能自动完成原本要点几十下鼠标的逆向任务。

我们选择开始和结束位置 匹配然后替换

```python
import idc
def clear(start_ea,end_ea):
    s_o_h=[0x74,0x03,0x75,0x01,0xe9]
    while start_ea<end_ea:
        if idc.get_bytes(start_ea,5)==bytes(s_o_h):
            for i in range(5):
                idc.patch_byte(start_ea+i,0x90)
        start_ea+=1
start_ea=0x1450
end_ea=0x15C8
clear(start_ea,end_ea)
print("ok")
```

花指令题目练习

[SWPUCTF 2023 秋季新生赛]Junk Code

修复后如图

```
8
9   _main();
10  v8 = 1;
11  strcpy(v6, "NRQ@PC}Vdn4tHV4Yi9cd#\\}jsXz3LMuaaY0}nj]`4a5&WoB4glB7~u");
12  printf("Input your flag:\n");
13  scanf("%100s", Str);
14  for ( i = 0; i < strlen(Str); ++i )
15  {
16    if ( (v6[i] ^ (i % 9)) != Str[i] )
17    {
18      v8 = 0;
19      break;
20    }
21  }
22  if ( v8 == 1 )
23    v3 = "Right! Congratulation!";
24  else
25    v3 = "Wrong! Try agian!";
26  printf("%s", v3);
27  return 0;
28 }
```

花指令题目练习

[GFCTF 2021]wordy

自己练习

花指令题目练习

[GFCTF 2021]wordy

匹配 0xeB 和0xff

Nop掉0xeB 还要提取0xbf后面的值

Z3约束求解

z3作为一种约束求解器，在CTF中可以用来解一些密码学，二进制逆向等问题 如求数组 和逆向求一些复制表达式
安装：
pip install z3-solver

用法:
1.变量声明

2.常规使用

3.约束求解器

Z3约束求解

2.常规使用

Solver()
Solver()命令会创建一个通用求解器，创建后我们可以添加我们的约束条件，进行下一步的求解

add()
add()命令用来添加约束条件，通常在solver()命令之后，添加的约束条件通常是一个逻辑等式

check()
该函数通常用来判断在添加完约束条件后，来检测解的情况，有解的时候会回显sat，无解的时候会回显unsat

model()
在存在解的时候，该函数会将每个限制条件所对应的解集的交集，进而得出正解。

Z3约束求解

3.约束求解器

```
from z3 import *
a,b = Ints('a b')
solver = Solver()#创建一个求解器对象
solver.add(a+b==10)#用add方法添加约束条件
solver.add(a-b==6)
if solver.check() == sat: #check()方法用来判断是否有解，
sat(satisify)表示满足有解
    ans = solver.model() #model()方法得到解
    print(ans)
    #也可以用变量名作为下标得到解
    print(ans[a])
else:
    print("no ans!")
```

Z3约束求解

例一
假设有
30x+15y=675
12x+5y=265

如何求解?

```
from z3 import *
x = Real('x')
y = Real('y')      #设未知数
s = Solver()       #创建约束求解器
s.add(30*x+15*y==675)
s.add(12*x+5*y==265)    #添加约束条件
if s.check() == sat:    #检测是否有解
    result = s.model()
    print result        #若有解则得出解，注意这里的解是等式
else:
        print 'no result'     #无解则打印no result
```

Z3约束求解

例一
假设有
30x+15y=675
12x+5y=265

如何求解?

```
from z3 import *
x = Real('x')
y = Real('y')      #设未知数
s = Solver()       #创建约束求解器
s.add(30*x+15*y==675)
s.add(12*x+5*y==265)    #添加约束条件
if s.check() == sat:    #检测是否有解
    result = s.model()
    print( result      ) #若有解则得出解，注意这里的解是等式
else:
    print ('no result' )
```

动态调试基础

IDA作为一款强大的逆向集成工具，对于动态调试也有比较好的支持。利用IDA自带的动态调试功能可以完成大部分的动态调试任务。

动态调试是一种观察程序运行状态的一种手段。

逆向工程中动态调试的目的主要有：

验证静态分析的结果
观察程序运行时的数据
IDA调试器支持的特性

软件断点/硬件断点/条件断点/脚本断点
步入/步过/步出函数/执行到光标位置
汇编级/伪代码级/源码级 调试代码支持
寄存器/内存 读写
启动进程调试/附加调试

动态调试基础

IDA作为一款强大的逆向集成工具，对于动态调试也有比较好的支持。利用IDA自带的动态调试功能可以完成大部分的动态调试任务。

动态调试是一种观察程序运行状态的一种手段。

逆向工程中动态调试的目的主要有：

验证静态分析的结果
观察程序运行时的数据
IDA调试器支持的特性

软件断点/硬件断点/条件断点/脚本断点
步入/步过/步出函数/执行到光标位置
汇编级/伪代码级/源码级 调试代码支持
寄存器/内存 读写
启动进程调试/附加调试

动态调试基础

软件断点:
实现方式: 软件断点是通过修改目标程序的指令来实现的。IDA 会将目标指令替换为一个特殊的指令(通常是 INT 3 指令)来创建软件断点。
触发方式: 当执行到这个特殊指令时,CPU 会触发一个异常,从而触发断点。
优缺点: 软件断点设置简单,但会修改目标程序的代码,可能会影响程序的行为。软件断点也容易被目标程序检测到。

硬件断点:
实现方式: 硬件断点是利用 CPU 内部的特殊寄存器来实现的。IDA 会设置这些寄存器,指定断点的地址和条件。
触发方式: 当执行到指定的地址或满足特定条件时,CPU 会自动触发一个异常,从而触发断点。
优缺点: 硬件断点不会修改目标程序的代码,不会影响程序的行为。但是每个 CPU 只有有限个硬件断点寄存器,资源是有限的。硬件断点设置也相对复杂一些。

动态调试基础

步入 (Step Into):
作用: 当程序执行到一个函数调用语句时,步入操作会进入该函数内部,并停在函数的第一行代码处。
用途: 当我们需要深入分析函数内部的执行逻辑时,步入操作非常有用。它可以帮助我们更好地理解程序的执行流程。

步过 (Step Over):
作用: 当程序执行到一个函数调用语句时,步过操作会执行整个函数,并停在下一行语句处。
用途: 如果我们不需要深入分析函数内部的执行逻辑,而只想关注程序的整体执行流程,步过操作就很有帮助。它可以让我们快速地跳过函数调用,关注更高层次的执行过程。

步出 (Step Out):
作用: 当程序执行到一个函数内部时,步出操作会执行完该函数,并停在函数返回后的下一行语句处。
用途: 当我们已经进入一个函数内部进行分析,但发现不需要深入分析时,步出操作可以帮助我们快速地返回到调用函数的上下文中。

执行到光标位置 (Run to Cursor):
作用: 当程序执行到一个断点或步骤时,执行到光标位置操作会让程序执行到您在代码编辑器中设置的光标位置,并在那里停止。
用途: 这个操作非常有用,当您已经定位到感兴趣的代码区域时,可以使用它来快速执行到该区域,而不需要一步一步地执行。这可以节省大量的调试时间。

动态调试基础

在ida里面

F2下断点（在程序执行的时候是软件　在字符串或者其他地方下断点是硬件断点）

F4执行到光标位置

F7步入

F8步过

F9执行到下一断点

动态调试基础

异常处理机制

在动态调试中,异常处理机制也是非常重要的一部分。当程序运行时发生异常时,调试器可以帮助我们快速定位和解决问题。
异常捕获: 断点异常 除0异常 等等
作用: 当程序运行时发生异常时,调试器会自动捕获这些异常,并暂停程序的执行。这样可以让我们查看异常的详细信息,并分析导致异常的原因。
用途: 通过异常捕获,我们可以快速定位程序中的错误点,并进一步进行调试和修复。

异常处理逻辑:
作用: 在代码中添加适当的异常处理逻辑,可以让程序在发生异常时能够优雅地处理和恢复,而不是直接崩溃。

动态调试基础

反调试
1.IsDebuggerPresent

```
*this = (*(_BYTE *)(__readfsdword(0x30u) + 2) != 0) + 50;// _PEB+0x02-->BeingDebugged,0为未调试
v2 = IsDebuggerPresent() == 0;                  // 查询进程环境块(PEB)中的IsDebugged标志。
                                                // 如果进程没有运行在调试器环境中，函数返回0；如果调试附加了进程，函数返回一个非零值。
```

IsDebuggerPresent 是一个 Windows API 函数,用于检查当前进程是否正在被调试器监控。这个函数在动态调试中非常有用,因为它可以让程序检测到自己是否正在被调试。

动态调试基础

反调试
1.IsDebuggerPresent

```
#include <Windows.h>

int main() {
    if (IsDebuggerPresent()) {
        // 如果正在被调试器监控,则执行这里的代码
        MessageBox(NULL, "The program is being debugged.", "Debug Mode", MB_OK);
    } else {
        // 如果不在被调试器监控,则执行这里的代码
        MessageBox(NULL, "The program is not being debugged.", "Release Mode", MB_OK);
    }

    return 0;
}
```

动态调试基础

反调试
2.CheckRemoteDebuggerPresent
这个函数检查的是你获取的进程是否被另一个进程调试。

```
v4 = GetCurrentProcess();
CheckRemoteDebuggerPresent(v4, &pbDebuggerPresent);// 非零值表示该进程正在ring3调试器的控制下运行。
```

动态调试基础

反调试
2.CheckRemoteDebuggerPresent
这个函数检查的是你获取的进程是否被另一个进程调试。
原型
NtQueryInformationProcess (
IN HANDLE ProcessHandle, // 获取进程的句柄
IN PROCESSINFOCLASS InformationClass, // 信息类型
OUT PVOID ProcessInformation, // 缓冲区的指针
IN ULONG ProcessInformationLength, // 缓冲区大小
OUT PULONG ReturnLength OPTIONAL // 写入缓冲区的字节数
);

其中ProcessInformationClass中的ProcessDebugPort，它来检索调试器的端口号，只要是非0则有调试器。

动态调试基础

反调试
时钟检测
采用的是__rdtsc进行的检测。

Windows系列操作系统的时间间隔10 - 20 毫秒，软件正常运行时的速度比我们分析代码时快得多，所以可以比较上下两句代码的时间戳，来判断程序是否被调试。

```
sub_401580((int)v1, &v17);
v12 = __rdtsc();
pbDebuggerPresent = v12;
v13 = __rdtsc();
```

动态调试基础

反调试
TLS (Thread Local Storage) 回调函数是 Windows 操作系统中一种特殊的机制,它允许开发者在程序的线程生命周期中执行自定义的代码。

TLS 回调函数的主要特点如下:
执行时机:
TLS 回调函数会在线程创建、退出和进程退出时被调用。
这意味着开发者可以在线程的关键阶段执行一些初始化、清理或其他自定义操作。

也就是在主进程开始和结束时执行一次

可以用作反调试

动态调试练习题目

[LitCTF 2023]程序和人有一个能跑就行了

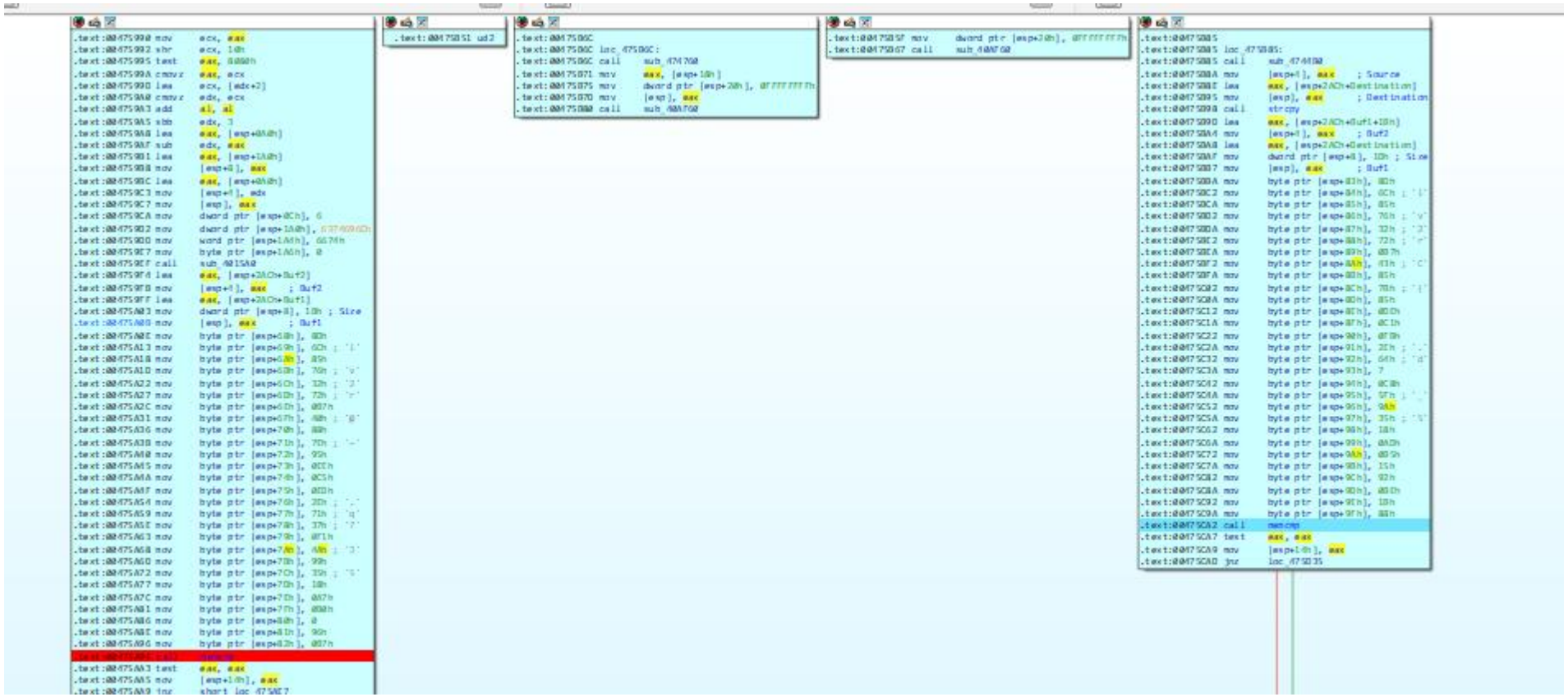
```
28  sub_409B80();
29  v11 = -1;
30  sub_472810(&dword_47DD80, Buf2);
31  strcpy(Destination, "litctf");
32  sub_4015A0(Buf2, strlen(Buf2), Destination, 6);
33  Buf1[0] = -115;
34  Buf1[1] = 108;
35  Buf1[2] = -123;
36  Buf1[3] = 118;
37  Buf1[4] = 50;
38  Buf1[5] = 114;
39  Buf1[6] = -73;
40  Buf1[7] = 64;
41  Buf1[8] = -120;
42  Buf1[9] = 126;
43  Buf1[10] = -107;
44  Buf1[11] = -18;
45  Buf1[12] = -59;
46  Buf1[13] = -19;
47  Buf1[14] = 46;
48  Buf1[15] = 113;
49  Buf1[16] = 55;
50  Buf1[17] = -15;
51  Buf1[18] = 74;
52  Buf1[19] = -103;
53  Buf1[20] = 53;
54  Buf1[21] = 24;
55  Buf1[22] = -89;
56  Buf1[23] = -80;
57  Buf1[24] = 0;
58  Buf1[25] = -106;
59  Buf1[26] = -73;
60  v8 = memcmp(Buf1, Buf2, 0x1Bu);
61  if ( v8 )
62  {
63    v11 = 1;
64    v5 = print(&dword_47DF60, "U are wrong?");
65    sub_46FBA0(v5);
```

函数参数过多，很难静态看 我们来动态调试

动态调试练习题目

[LitCTF 2023]程序和人有一个能跑就行了

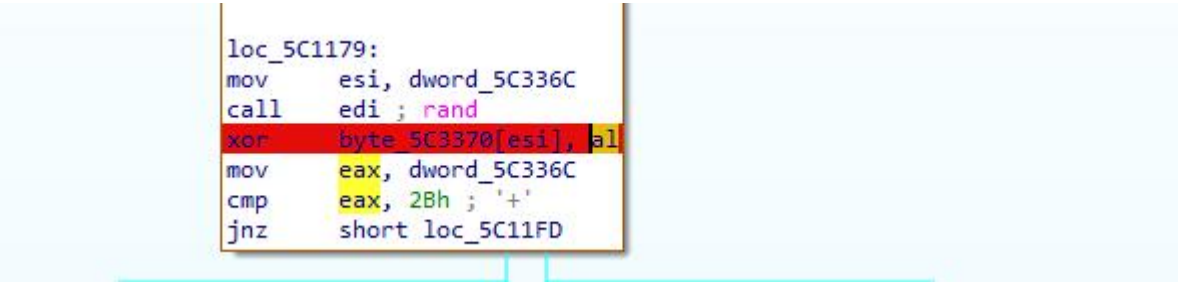
其实是密文被替换了



动态调试练习题目

[MTCTF 2021]Random
异常处理机制

只需要自己调试出al的值然后xor回去



动态调试练习题目

[MoeCTF 2022]fake_key

眼见不一定为真，就像你看到的密钥。
程序中出现不确定的数字怎么办呢？调试就完事。
仍然是调试出*key*

```
7FF99CB10000: loaded C:\Windows\System32\KernelBase.dll
7FF99F500000: loaded C:\WINDOWS\System32\msvcrt.dll
7FF99F9D2220: thread has started (tid=51708)
121,117,110,122,104,49,106,117,110,84,67,76,44,116,114,97,99,107,89,89,68,83,121,117,110,122,104,49,106,117,110,84,67,76,44,116,1,7,4,0,9,4,8,8,2,4,5,5,1,7,1,1,5,2,7,6,1,4,2,3,2,2,1,6,8,5,7,6,1,8,9,2,Debugger: thread 51708 has exited (code 0)
Debugger: process has exited (exit code 0)
```

动态调试练习题目

[MoeCTF 2022]fake_key

眼见不一定为真，就像你看到的密钥。
程序中出现不确定的数字怎么办呢？调试就完事。
仍然是调试出*key*

```
7FF99CB10000: loaded C:\Windows\System32\KernelBase.dll
7FF99F500000: loaded C:\WINDOWS\System32\msvcrt.dll
7FF99F9D2220: thread has started (tid=51708)
121,117,110,122,104,49,106,117,110,84,67,76,44,116,114,97,99,107,89,89,68,83,121,117,110,122,104,49,106,117,110,84,67,76,44,116,1,7,4,0,9,4,8,8,2,4,5,5,1,7,1,1,5,2,7,6,1,4,2,3,2,2,1,6,8,5,7,6,1,8,9,2,Debugger: thread 51708 has exited (code 0)
Debugger: process has exited (exit code 0)
```

```python
add=[1,7,4,0,9,4,8,8,2,4,5,5,1,7,1,1,5,2,7,6,1,4,2,3,2,2,1,6,8,5,7,6,1,8,9,2,7,9,5,4,3,1,2,3,3,4,1,1,3,8]
xor_table=[121,117,110,122,104,49,106,117,110,84,67,76,44,116,114,97,99,107,89,89,68,83,121,117,110,122,104,49,106]
enc=[0x15, 0x21, 0x0F, 0x19, 0x25, 0x5B, 0x19, 0x39, 0x5F, 0x3A, 0x3B, 0x30, 0x74, 0x07, 0x43, 0x3F, 0x09, 0x5A, 0x34, 0x0C, 0x74, 0x3F, 0x1E, 0x2D, 0x27, 0x21, 0x12, 0x16, 0x1F]
for i in range(len(enc)):
    enc[i] = enc[i] - add[i % len(add)]
    enc[i] = enc[i] ^ xor_table[i % len(xor_table)]
print("".join([chr(i) for i in enc]))
```

动态调试练习题目

[LitCTF 2024]ezrc4

反调试？！

有两处反调试

```
73CE20008 public key
73CE20008 ; char key[]
73CE20008 key db 'litctf!',0                    ; DATA XREF: X_X↑o
73CE20008                                       ; main+11F↑o
73CE20010 p_0 dq offset qword_7FF73CE1F090      ; DATA XREF: __do_global_dtors+4↑r
73CE20010                                       ; __do_global_dtors+1A↑r ...
73CE20018 align 20h
73CE20020 public __native_vcclrit_reason
73CE20020 __native_vcclrit_reason db 0EFh
```

动态调试练习题目

[LitCTF 2024]ezrc4

反调试？！

有两处反调试

```
73CE20008 public key
73CE20008 ; char key[]
73CE20008 key db 'litctf!',0                    ; DATA XREF: X_X↑o
73CE20008                                       ; main+11F↑o
73CE20010 p_0 dq offset qword_7FF73CE1F090      ; DATA XREF: __do_global_dtors+4↑r
73CE20010                                       ; __do_global_dtors+1A↑r ...
73CE20018 align 20h
73CE20020 public __native_vcclrit_reason
73CE20020 __native_vcclrit_reason db 0EFh
```

动态调试练习题目

[HNCTF 2022 WEEK2]Try2Bebug_Plus

ELF文件　需要配置相关环境
需要一个linux 然后把linux_server放进去跑

```
[stack]:00007FFFFFFFE14B db    0
[stack]:00007FFFFFFFE14C db  74h ; t
[stack]:00007FFFFFFFE14D db  68h ; h
[stack]:00007FFFFFFFE14E db  31h ; 1
[stack]:00007FFFFFFFE14F db  73h ; s
[stack]:00007FFFFFFFE150 db  5Fh ; _
[stack]:00007FFFFFFFE151 db  31h ; 1
[stack]:00007FFFFFFFE152 db  73h ; s
[stack]:00007FFFFFFFE153 db  5Fh ; _
[stack]:00007FFFFFFFE154 db  66h ; f
[stack]:00007FFFFFFFE155 db  6Ch ; l
[stack]:00007FFFFFFFE156 db  61h ; a
[stack]:00007FFFFFFFE157 db  67h ; g
[stack]:00007FFFFFFFE158 db    0
[stack]:00007FFFFFFFE159 db  60h ; `
[stack]:00007FFFFFFFE15A db  9Eh
[stack]:00007FFFFFFFE15B db  34h ; 4
[stack]:00007FFFFFFFE15C db  32h ; 2
[stack]:00007FFFFFFFE15D db  26h ; &
```

APP逆向分析流程

结构分析 · Java层 · Native层 · ARM汇编

APP结构分析

- [APK 文件结构]
- - AndroidManifest.xml：应用权限、组件、入口Activity等信息
- - classes.dex：Java编译后的Dalvik字节码
- - lib/：Native库（.so 文件），包含多个平台的实现（如arm64-v8a）
- - assets/：原始资源文件（如配置、html、json）
- - res/：UI布局、图片等资源（已编译）
- - META-INF/：APK签名信息

- [常用反编译工具]
- - apktool：反编译为smali并提取资源
- - JADX / JEB：将DEX反编译为Java源码
- - IDA / Ghidra：分析.so文件
- - AXMLPrinter2 / baksmali：辅助反编译与修改

例题1：babyapk

一般都在com目录下，找到主函数

例题1：babyapk

解压后查看这个目录 用ida打开 xor了key

```
ArrayLength = _JNIEnv::GetArrayLength(a1, a3);
IntArrayElements = _JNIEnv::GetIntArrayElements(a1, a3, 0LL);
for ( i = 0; i < ArrayLength; ++i )
  *(IntArrayElements + 4LL * i) ^= key[i % 4];
_JNIEnv::SetIntArrayRegion(a1, a3, 0LL, ArrayLength, IntArrayElements);
return a3;
```

查看左边函数发现有其他的， 最后xor就可以了

```
_DWORD *hide_key(void)
{
  _DWORD *result; // rax

  result = key;
  key[0] ^= 0x47u;
  key[1] ^= 0x32u;
  key[2] ^= 0x11u;
  key[3] ^= 0x12u;
  return result;
}
```

例题1：babyapk

定位到关键逻辑

```
public static final void m67onCreate$lambda0(MainActivity this$0, int[] c, View it) {
    Intrinsics.checkNotNullParameter(this$0, "this$0");
    Intrinsics.checkNotNullParameter(c, "$c");
    IntStream flag = ((EditText) this$0._$_findCachedViewById(C0477R.id.input)).getText().toString().chars();
    int[] array = flag.toArray();
    Intrinsics.checkNotNullExpressionValue(array, "flag.toArray()");
    int[] cipher = this$0.baby_xor(array);
    if (Arrays.equals(cipher, c)) {
        Toast.makeText(this$0, "Success", 1).show();
    } else {
        Toast.makeText(this$0, "Failed", 0).show();
    }
}

static {
    System.loadLibrary("createso");
}
```

函数定位在native层就是 在so里面

```
public final native int[] baby_xor(int[] x);
```

例题2：goodluck

这个时候我们用jeb ooxx是一个算法

```
public class OOXX {
    private static final int A = 0x67452301;
    private static final int B = 0xEFCDAB89;
    private static final int C = 0x98BADCFE;
    private static final int D = 0x10325476;
    private static final int[] T;
    private static final int[] xxooxx;

    // This method was un-flattened
    static {
        OOXX.xxooxx = new int[]{7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 5, 9, 14, 20, 5, 9, 14, 20, 5,
        OOXX.T = new int[0x40];
        for(int v = 0; v < 0x40; ++v) {
            OOXX.T[v] = (int)(((long)(Math.abs(Math.sin(v + 1)) * 4294967296.0)));
        }
    }

    // This method was un-flattened
    private static String bytesToHex(byte[] arr_b) {
        StringBuilder stringBuilder0 = new StringBuilder();
        for(int v = 0; v < arr_b.length; ++v) {
            stringBuilder0.append(String.format("%02x", ((int)(arr_b[v] & 0xFF))));
        }

        return stringBuilder0.toString();
    }

    // This method was un-flattened
    private static int bytesToInt(byte[] arr_b, int v) {
        return arr_b[v] & 0xFF | (arr_b[v + 1] & 0xFF) << 8 | (arr_b[v + 2] & 0xFF) << 16 | (arr_b[v + 3] & 0xFF) << 24;
    }

    // This method was un-flattened
    private static byte[] longToBytes(long v) {
        byte[] arr_b = new byte[8];
        for(int v1 = 0; v1 < 8; ++v1) {
            arr_b[v1] = (byte)(((int)(v >> v1 * 8)));
        }
```

例题2：goodluck

md5算法:**MD5（Message-Digest Algorithm 5）** 是一种**单向哈希函数**，目的是将任意长度的数据转换成一个固定长度的128位（16字节）哈希值，常用于数据完整性校验。

特征：
A = 0x67452301
B = 0xefcdab89
C = 0x98badcfe
D = 0x10325476

求解：[md5在线解密破解,md5解密加密]

例题2：goodluck

发现是md5算法

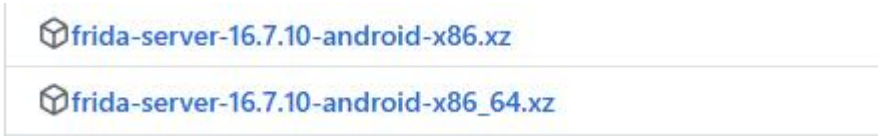
```
public class OOXX {
    private static final int A = 0x67452301;
    private static final int B = 0xEFCDAB89;
    private static final int C = 0x98BADCFE;
    private static final int D = 0x10325476;
    private static final int[] T;
    private static final int[] xxooxx;

    // This method was un-flattened
    static {
        OOXX.xxooxx = new int[]{7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 5, 9, 14, 20, 5, 9, 14, 20, 5,
        OOXX.T = new int[0x40];
        for(int v = 0; v < 0x40; ++v) {
            OOXX.T[v] = (int)(((long)(Math.abs(Math.sin(v + 1)) * 4294967296.0)));
        }
    }

    // This method was un-flattened
    private static String bytesToHex(byte[] arr_b) {
        StringBuilder stringBuilder0 = new StringBuilder();
        for(int v = 0; v < arr_b.length; ++v) {
            stringBuilder0.append(String.format("%02x", ((int)(arr_b[v] & 0xFF))));
        }

        return stringBuilder0.toString();
    }

    // This method was un-flattened
    private static int bytesToInt(byte[] arr_b, int v) {
        return arr_b[v] & 0xFF | (arr_b[v + 1] & 0xFF) << 8 | (arr_b[v + 2] & 0xFF) << 16 | (arr_b[v + 3] & 0xFF) << 24;
    }

    // This method was un-flattened
    private static byte[] longToBytes(long v) {
        byte[] arr_b = new byte[8];
        for(int v1 = 0; v1 < 8; ++v1) {
            arr_b[v1] = (byte)(((int)(v >> v1 * 8)));
        }
```

扩展：frida的基本使用

动态插桩Frida 允许你在不修改目标二进制文件的情况下，在进程运行时对函数进行 hook（钩子）操作，可以拦截或修改函数调用、返回值等，适用于调试、逆向、黑盒测试、安全研究等场景。跨平台支持Frida 支持多种平台，包括 Windows、macOS、Linux、iOS、Android 等。
特别是在移动平台上，Frida 是研究应用防护机制、绕过反调试检测的常用利器。
脚本语言Frida 使用 JavaScript 作为脚本语言，这让你可以利用熟悉的语法来描述 hook 逻辑。同时，Frida 提供了 Python 包（frida-python），使得你可以通过 Python 脚本来加载、管理和控制 Frida 注入过程。

pip install frida-tools

adb push 到data/local/tmp 目录



扩展：frida的基本使用

pip install frida-dexdump

frida-dexdump -U -f com.app.pkgname

在 Android 平台上，Java 源代码在编译后会被转换为字节码，这些字节码不会直接生成 .class 文件而是会统一汇编成一个或多个 Dalvik Executable（DEX）文件。DEX 文件是 Android 的核心文件格式，用于在 Dalvik 或 ART（Android Runtime）虚拟机中执行程序代码。

使用 Frida 注入相应的 JavaScript 脚本到目标 Android 进程中，通过访问应用内部加载的 dex 模块（通常是在 ART 虚拟机中），将内存中的 dex 数据块 dump 出来。Frida-dexdump 会解析这些内存数据，并以较为可读的形式输出 dex 文件中各个部分的信息。