

# 汇编语言与逆向工程

北京邮电大学  
崔宝江

北邮网安学院 崔宝江



# 第五章 常见加密算法逆向分析

- ⑤ 5.1 简单加密算法逆向分析
- ⑤ 5.2 对称加密算法逆向分析
- ⑤ 5.3 单向散列算法逆向分析



# 5.1 简单加密算法逆向分析

## ① 1 异或加密

- ❑ (1) 原理介绍
- ❑ (2) 加解密步骤
- ❑ (3) 逆向分析

## ① 2 仿射加密

- ❑ (1) 原理介绍
- ❑ (2) 加解密步骤
- ❑ (3) 逆向分析



# 1 异或加密

- ❑ 异或运算符常作为更为复杂加密算法的一个组成部分
- ❑ 如果使用不断重复的密钥，利用频率分析就可以破解这种简单的异或密码
- ❑ 如果消息的内容被猜出或知道，密钥就会泄露
- ❑ 异或密码值得使用的原因主要是其易于实现，而且计算成本小
- ❑ 简单重复异或加密有时用于不需要特别安全的情况下隐藏信息



# 1 异或加密

## ④ (1) 原理介绍

- ❑ 异或是一种运算，数学运算符为**XOR**
- ❑ 总结起来就是相同的数（取**0**或**1**）异或得到的结果为**0**，不同则为**1**

$$A \text{ XOR } 0 = A$$

$$A \text{ XOR } A = 0$$

$$(A \text{ XOR } B) \text{ XOR } B = A$$

- ❑ 或，OR，有1即为1（包括两个1和1个1），其他则为0



# 1 异或加密

## □ (2) 加解密步骤

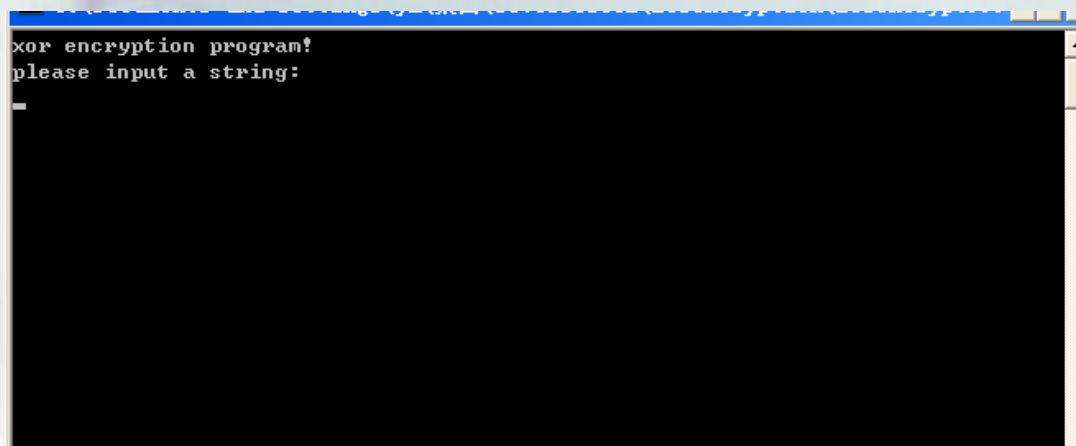
- 加密过程只需要将明文和密钥逐字节异或，而解密过程则只需要将密文和密钥逐字节异或



# 1 异或加密

## □ (3) 逆向分析

- 运行程序xorencryption.exe，通过逆向分析了解这个逆向CTF程序具有什么功能



```
xor encryption program!  
please input a string:  
-
```



# 1 异或加密

❑ 使用IDA打开xorencryption.exe，定位到main函数，分析程序的流程

○ 程序首先获取了用户的输入，并计算输入的长度，如果长度为10，则跳转到0x40107E处继续运行，否则输出wrong，并退出运行。

```
.text:00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401000 _main      proc near      ; CODE XREF: start+AF1p
.text:00401000
.text:00401000 var_C      = byte ptr -0Ch
.text:00401000 argc      = dword ptr  4
.text:00401000 argv     = dword ptr  8
.text:00401000 envp     = dword ptr 0Ch
.text:00401000
.text:00401000 sub      esp, 0Ch
.text:00401000 push     edi
.text:00401000 push     offset aXorEncryptionP ; "xor encryption program?"
.text:00401000 call     _puts
.text:00401000 push     offset aPleaseInputAST ; "please input a string:"
.text:00401000 call     _puts
.text:00401000 lea      eax, [esp+18h+var_C]
.text:00401000 push     eax
.text:00401000 push     offset aS      ; "%s"
.text:00401000 call     _scanf
.text:00401000 lea      edi, [esp+20h+var_C]
.text:00401000 or       ecx, 0FFFFFFFh
.text:00401000 xor      eax, eax
.text:00401000 add      esp, 10h
.text:00401000 repne scasb
.text:00401000 not      ecx
.text:00401000 dec      ecx
.text:00401000 pop      edi
.text:00401000 cmp      ecx, 0Ah
.text:00401000 jz       short loc_40107E
.text:00401000 push     offset aWrong   ; "wrong?"
.text:00401000 call     _puts
.text:00401000 mov      eax, stru 400000h, cmt
```

获取用户输入

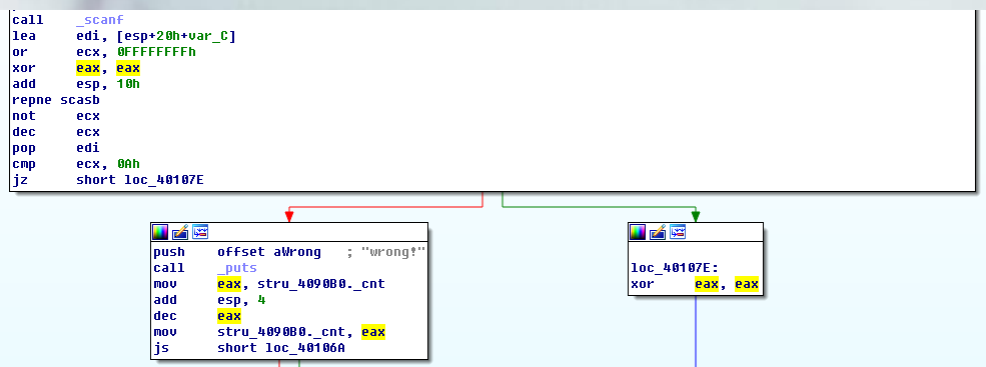
计算用户输入的长度





# 1 异或加密

- 程序首先获取了用户的输入，并计算输入的长度，如果长度为10，则跳转到0x40107E处继续运行，否则输出wrong，并退出运行。



# 1 异或加密

- ❑ 跟进到**0x40107E**代码段，程序首先逐字节的将用户输入的内容，与字节数组**0x409030**处的值（密钥）进行异或操作；



```
mov     ecx, 0Ah
jz      short loc_40107E
```

```
push     offset aWrong ; "wrong!"
call     _puts
mov      eax, stru_4090B0._cnt
add      esp, 4
dec      eax
mov      stru_4090B0._cnt, eax
js       short loc_40106A
```

```
mov      eax, stru_4090B0._ptr
inc      eax
mov      stru_4090B0._ptr, eax
or       eax, 0FFFFFFFh
add      esp, 0Ch
retn
```

```
loc_40106A: ; FILE *
push     offset stru_4090B0
call     _filbuf
add      esp, 4
or       eax, 0FFFFFFFh
add      esp, 0Ch
retn
```

```
loc_40107E:
xor      eax, eax
```

```
loc_401080:
mov      cl, byte_409030[eax]
mov      dl, [esp+eax+0Ch+var_C]
xor      dl, cl
mov      [esp+eax+0Ch+var_C], dl
inc      eax
cmp      eax, 0Ah
j1       short loc_401080
```

```
push     esi
xor      esi, esi
```

```
loc_401099:
movsx    eax, [esp+esi+10h+var_C]
xor      edx, edx
mov      dl, byte_40903C[esi]
cmp      eax, edx
jnz      short loc_4010B2
```

```
inc      esi
cmp      esi, 0Ah
j1       short loc_401099
```

```
jmp      short loc_4010BF
```

```
loc_4010B2: ; "wrong answer!!!!"
push     offset aWrongAnswer
```

# 1 异或加密

- ❑ 接着将加密得到的结果与字节数组**0x40903C**处的值（密文）进行比较，不相等则输出“**wrong answer!!!!**”，相等则输出“**goodjob!!!!**”



```

mov     eax, stru_409080._ptr
inc     eax
mov     stru_409080._ptr, eax
or      eax, 0FFFFFFFh
add     esp, 0Ch
retn

```

```

loc_40106A:                                ; FILE *
push    offset stru_409080
call    _filbuf
add     esp, 4
or      eax, 0FFFFFFFh
add     esp, 0Ch
retn

```

```

loc_401080:
mov     cl, byte_409030[eax]
mov     dl, [esp+eax+0Ch+var_C]
xor     dl, cl
mov     [esp+eax+0Ch+var_C], dl
inc     eax
cmp     eax, 0Ah
jl      short loc_401080

```

```

push    esi
xor     esi, esi

```

```

loc_401099:
movsx   eax, [esp+esi+10h+var_C]
xor     edx, edx
mov     dl, byte_40903C[esi]
cmp     eax, edx
jnz     short loc_4010B2

```

```

inc     esi
cmp     esi, 0Ah
jl      short loc_401099

```

```

jmp     short loc_4010BF

```

```

loc_4010B2:                                ; "wrong answer!!!"
push    offset aWrongAnswer
call    _puts
add     esp, 4

```

```

loc_4010BF:
cmp     esi, 0Ah
pop     esi
jnz     short loc_4010B2

```

```

push    offset aGoodJob ; "good job!!!"
call    _puts

```



# 1 异或加密

```
.text:00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401000 _main                proc near                ; CODE XREF: start+AF↓p
.text:00401000
.text:00401000 var_C                = byte ptr -0Ch
.text:00401000 argc                = dword ptr  4
.text:00401000 argv                = dword ptr  8
.text:00401000 envp                = dword ptr 0Ch
.text:00401000
.text:00401000 sub     esp, 0Ch
.text:00401003 push    edi
.text:00401004 push    offset aXorEncryptionP ; "xor encryption program?"
.text:00401009 call     _puts
.text:0040100E push    offset aPleaseInputAStr ; "please input a string:"
.text:00401013 call     _puts
.text:00401018 lea     eax, [esp+18h+var_C]
.text:0040101C push    eax
.text:0040101D push    offset aS          ; "%s"
.text:00401022 call     _scanf
.text:00401027 lea     edi, [esp+20h+var_C]
.text:0040102B or      ecx, 0FFFFFFFh
.text:0040102E xor     eax, eax
.text:00401030 add     esp, 10h
.text:00401033 repne scasd
.text:00401035 not     ecx
.text:00401037 dec     ecx
.text:00401038 pop     edi
.text:00401039 cmp     ecx, 0Ah
.text:0040103C jz      short loc_40107E
.text:0040103E push    offset aWrong      ; "wrong?"
.text:00401043 call     _puts
.text:00401048 mov     eax, stru_4090B0._cnt
.text:0040104D add     esp, 4
.text:00401050 dec     eax
.text:00401051 mov     stru_4090B0._cnt, eax
.text:00401056 js      short loc_40106A
.text:00401058 mov     eax, stru_4090B0._ptr
```

获取用户输入

计算用户输入的长度

# 1 异或加密

## □ 每条说明

- 用户输入的内容是保存在地址`esp+20h+var_C`处的，指令`lea edi,[esp+20h+var_C]`功能就是将`esp+20h+var_C`的值放入寄存器`edi`中，所以现在`edi`寄存器是指向用户输入的内容的。
- 指令`or ecx,0FFFFFFFFh`是将寄存器`ecx`的值设置为`0xFFFFFFFF`（也就是所有位设置为1）
- 指令`xor eax, eax`将`eax`的值设置为0（0在C语言中就是字符串的结尾）
- 指令`add esp,10h`是用来平衡栈的（这里不用关心）



# 1 异或加密

```
.text:00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401000 _main                proc near                ; CODE XREF: start+AF↓p
.text:00401000
.text:00401000 var_C                = byte ptr -0Ch
.text:00401000 argc                = dword ptr  4
.text:00401000 argv                = dword ptr  8
.text:00401000 envp                = dword ptr 0Ch
.text:00401000
.text:00401000 sub     esp, 0Ch
.text:00401003 push    edi
.text:00401004 push    offset aXorEncryptionP ; "xor encryption program?"
.text:00401009 call     _puts
.text:0040100E push    offset aPleaseInputAStr ; "please input a string:"
.text:00401013 call     _puts
.text:00401018 lea     eax, [esp+18h+var_C]
.text:0040101C push    eax
.text:0040101D push    offset aS          ; "%s"
.text:00401022 call     _scanf
.text:00401027 lea     edi, [esp+20h+var_C]
.text:0040102B or      ecx, 0FFFFFFFh
.text:0040102E xor     eax, eax
.text:00401030 add     esp, 10h
.text:00401033 repne scasd
.text:00401035 not     ecx
.text:00401037 dec     ecx
.text:00401038 pop     edi
.text:00401039 cmp     ecx, 0Ah
.text:0040103C jz      short loc_40107E
.text:0040103E push    offset aWrong      ; "wrong?"
.text:00401043 call     _puts
.text:00401048 mov     eax, stru_4090B0._cnt
.text:0040104D add     esp, 4
.text:00401050 dec     eax
.text:00401051 mov     stru_4090B0._cnt, eax
.text:00401056 js      short loc_40106A
.text:00401058 mov     eax, stru_4090B0._ptr
```

获取用户输入

计算用户输入的长度



# 1 异或加密

❑ 下面就是最关键的指令 **repne scasb**

❑ **repne**

- **repne** 是一个串操作指令中的条件重复前缀指令，加在串操作指令前，使串操作重复进行。
- **repne** 可检查两个字符串是否不同，发现相同立即停止比较。
- **repne** 的重复条件是 **CX≠0** 且 **ZF=0**，每执行一次，CX 的内容就减 1，直到 CX 减为 0 时，结束串指令操作。若重复条件满足，重复前缀先使 **CX←CX-1**，然后执行后面的串指令。



# 1 异或加密

❑ 指令 **repne scasb** 表示如果 **ecx** 的值不为 0 就继续执行后面的内容 --> **scasb**

❑ **scasb**

- **scasb** 则是串扫描指令，比较 **edi** 寄存器指向的值与 **eax** 寄存器中的值是否相等，每次将 **edi** 的值增加 1
- 如果相等就退出循环（执行该指令后面的指令），不相等就继续比较



# 1 异或加密

- ❑ **repne scasb** (**repeat not equal**) 该指令的功能是比较寄存器**edi**指向的值（用户输入值）和寄存器**eax**的值是否相等，如果不相等则将**edi**的值增一，并继续比较，相等则结束循环。
- ❑ 这里**eax**寄存器的值为**0**，将**edi**寄存器指向的值与**0**比较，是在判断是否到达了用户输入字符串的末尾。
- ❑ 每比较一次，寄存器**ecx**的值会减一。
- ❑ 因此寄存器**ecx**减少的值即为字符串的长度。



# 1 异或加密

```
.text:00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401000 _main                proc near                ; CODE XREF: start+AF↓p
.text:00401000
.text:00401000 var_C                = byte ptr -0Ch
.text:00401000 argc                = dword ptr  4
.text:00401000 argv                = dword ptr  8
.text:00401000 envp                = dword ptr 0Ch
.text:00401000
.text:00401000 sub     esp, 0Ch
.text:00401003 push    edi
.text:00401004 push    offset aXorEncryptionP ; "xor encryption program?"
.text:00401009 call     _puts
.text:0040100E push    offset aPleaseInputAStr ; "please input a string:"
.text:00401013 call     _puts
.text:00401018 lea     eax, [esp+18h+var_C]
.text:0040101C push    eax
.text:0040101D push    offset aS          ; "%s"
.text:00401022 call     _scanf
.text:00401027 lea     edi, [esp+20h+var_C]
.text:0040102B or      ecx, 0FFFFFFFh
.text:0040102E xor     eax, eax
.text:00401030 add     esp, 10h
.text:00401033 repne scasd
.text:00401035 not     ecx
.text:00401037 dec     ecx
.text:00401038 pop     edi
.text:00401039 cmp     ecx, 0Ah
.text:0040103C jz      short loc_40107E
.text:0040103E push    offset aWrong      ; "wrong?"
.text:00401043 call     _puts
.text:00401048 mov     eax, stru_4090B0._cnt
.text:0040104D add     esp, 4
.text:00401050 dec     eax
.text:00401051 mov     stru_4090B0._cnt, eax
.text:00401056 js      short loc_40106A
.text:00401058 mov     eax, stru_4090B0._ptr
```

获取用户输入

计算用户输入的长度

# 1 异或加密

## □再解释一遍

- 将**edi**指向了用户的输入，并将**eax**设置为了字符串结束的标志（**0**），所以这里就相当于逐字节的将输入的内容与**eax**（**0**）进行比较，如果相等了就表示到字符串结束了。
- 注意这里每比较一次**ecx**的值都会减少1，因此最后只需要看**ecx**的值减少了多少，用户输入的长度就是多少。



# 1 异或加密

- ❑ 所以 `repne scasb` 指令后面的 `not ecx` 以及 `dec ecx` 就是在计算 `ecx` 减少了多少，经过这两条指令后 `ecx` 就是字符串的长度了
- ❑ 将 `ecx` 与 `0xAH` (即二进制 `10`) 比较，看是否相等，不相等就输出 `wrong`



# 1 异或加密

- ❑ **not ecx**以及**dec ecx**是在计算**ecx**减少了多少
  - 一开始**ecx**的值为**0xFFFFFFFF**，假设循环了**5**次，也就是减了**5**。
  - **0xFFFFFFFF**对应的是**-1**，**-1-5=-6**，**-6**对应**0xFFFFF8**，然后对其取反，得到的值为**5**
  - 还有一个减**1**的操作，这样得到的结果就为**4**



# 1 异或加密

- 之所以不是5，这里需要看一下**scasb**指令具体的流程，

- scasb** :

```
inc edi  
dec ecx  
je loopdone  
cmp byte [edi-1],al  
jne scans  
loopdone
```

- 其中在循环开始处**ecx**减了1，相当于在扫描到结尾‘\0’处也将**ecx**减了1，多减了一个1（也就是将字符串长度多算了一个）





# 1 异或加密

- ❑ 讲了这么多汇编代码，其实这就是C语言库函数**strlen()**的汇编语言实现方式
- ❑ 因为使用的是**release**版，这里编译器做了许多优化（可以看到整个程序的汇编代码中都没有使用到**ebp**寄存器，全部使用**esp+xxx**代替了）



IDA - C:\Documents and Settings\Administrator\桌面\课内练习\第5章\第五章\随书代码-2\5-1\程序\xorencryption.exe

File Edit Jump Search View Debugger Options Windows Help

Library function Data Regular function Unexplored Instruction External symbol

Functions window

Function name  
\_main  
\_system  
\_filbuf  
\_scanf  
\_puts  
\_start  
\_amsg\_exit  
\_fast\_error\_exit  
\_spawnvpe  
\_cinit  
\_exit  
\_doexit  
\_initterm  
\_spawnve  
\_comexecd  
\_access  
\_getenv  
\_ioinit  
\_read  
\_getbuf  
sub\_401D63  
\_fcloseall  
\_fflush  
\_flush  
sub\_401F03  
\_flsall  
\_input  
\_hexdec  
\_fgetc  
\_un\_inc  
\_whiteout  
\_stbuf  
\_ftbuf  
\_flsbuf  
\_fwrite  
\_strlen  
\_XcptFilter  
\_xcptlookup  
\_setenvp  
\_setargv  
\_parse\_cmdline  
\_crtGetEnvironmentStringsA  
sub\_403357  
sub\_403384

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     int result; // eax@3
4     signed int u4; // eax@5
5     int u5; // esi@7
6     char v6[12]; // [sp+4h] [bp-Ch]@1
7
8     puts(aXorEncryptionP);
9     P1e; puts
10    scanf(aS, v6);
11    if ( strlen(v6) == 10 )
12    {
13        u4 = 0;
14        do
15        {
16            v6[u4] ^= byte_409838[u4];
17            ++u4;
18        }
19        while ( u4 < 10 );
20        u5 = 0;
21        while ( v6[u5] == (unsigned __int8)byte_40983C[u5] )
22        {
23            if ( ++u5 >= 10 )
24                goto LABEL_12;
25        }
26        puts(aWrongAnswer);
27    LABEL_12:
28        if ( u5 == 10 )
29            puts(aGoodJob);
30        system(aPause);
31        result = 0;
32    }
33    else
34    {
35        puts(aWrong);
36        if ( --stru_4098B0._cnt < 0 )
37        {
38            _filbuf(&stru_4098B0);
39            result = -1;
40        }
41        else
42        {
43            ++stru_4098B0._ptr;
44            result = -1;
45        }
46    }
47    return result;
48 }
```

Line 1 of 130

401000: using guessed type char var\_C[12];  
401000: using guessed type char var\_C[12];

Python

AU: idle Up Disk: 35GB

开始 程序 IDA\_Pro\_v6.8 第5章 IDA - C:\Do...

北邮网安学院 崔宝江



# 1 异或加密

## @ 异或加密过程

- ❑ 看一下**0x40107E**代码段
- ❑ 程序首先逐字节的将用户输入的内容与字节数组**0x409030**处的值（密钥）进行异或操作；



# 1 异或加密

```
.text:0040107E loc_40107E: ; CODE XREF: _main+3C1j
.text:0040107E xor     eax, eax
.text:00401080
.text:00401080 loc_401080: ; CODE XREF: _main+941j
.text:00401080 mov     cl, byte_409030[eax]
.text:00401086 mov     dl, [esp+eax+0Ch+var_C]
.text:0040108A xor     dl, cl
.text:0040108C mov     [esp+eax+0Ch+var_C], dl
.text:00401090 inc     eax
.text:00401091 cmp     eax, 0Ah
.text:00401094 jl      short loc_401080
.text:00401096 push    esi
.text:00401097 xor     esi, esi
.text:00401099
.text:00401099 loc_401099: ; CODE XREF: _main+AE1j
.text:00401099 movsx   eax, [esp+esi+10h+var_C]
.text:0040109E xor     edx, edx
.text:004010A0 mov     dl, byte_40903C[esi]
.text:004010A6 cmp     eax, edx
.text:004010A8 jnz     short loc_4010B2
.text:004010AA inc     esi
.text:004010AB cmp     esi, 0Ah
.text:004010AE jl      short loc_401099
.text:004010B0 jmp     short loc_4010BF
.text:004010B2 ; -----
.text:004010B2
.text:004010B2 loc_4010B2: ; CODE XREF: _main+A81j
.text:004010B2 push    offset aWrongAnswer ; "wrong answer!!!"
.text:004010B7 call    _puts
.text:004010BC add     esp, 4
.text:004010BF
.text:004010BF loc_4010BF: ; CODE XREF: _main+B01j
.text:004010BF cmp     esi, 0Ah
.text:004010C2 pop     esi
```

逐字节异或加密部分

判断是否是期望的加密结果



# 1 异或加密

- ❑ 第一条指令**mov**指令从指定的字节数组**0x409030**处取出一个字节，放到寄存器**ecx**中
- ❑ 看第一条指令**mov cl,byte\_409030[eax]**，其中地址**0x409030**指向的是一片内存，**byte\_409030[eax]**的意思就是取地址**0x409030+eax**处的一个字节的意思
- ❑ 这样不断将**eax**的值增加1，实现逐字节取数据



# 1 异或加密

```
.text:0040107E loc_40107E: ; CODE XREF: _main+3C1j
xor     eax, eax

.text:00401080 loc_401080: ; CODE XREF: _main+941j
mov     cl, byte_409030[eax]
mov     dl, [esp+eax+0Ch+var_C]
xor     dl, cl
mov     [esp+eax+0Ch+var_C], dl
inc     eax
cmp     eax, 0Ah
jl      short loc_401080
push    esi
xor     esi, esi

.text:00401099 loc_401099: ; CODE XREF: _main+AE1j
movsx   eax, [esp+esi+10h+var_C]
xor     edx, edx
mov     dl, byte_40903C[esi]
cmp     eax, edx
jnz     short loc_4010B2
inc     esi
cmp     esi, 0Ah
jl      short loc_401099
jmp     short loc_4010BF

.text:004010B2 loc_4010B2: ; CODE XREF: _main+A81j
push    offset aWrongAnswer ; "wrong answer!!!"
call    _puts
add     esp, 4

.text:004010BF loc_4010BF: ; CODE XREF: _main+B01j
cmp     esi, 0Ah
pop     esi
```

逐字节异或加密部分

判断是否是期望的加密结果



# 1 异或加密

- ❑ 第二条**mov**指令则是从用户输入的内容中取一个字节放入**edx**寄存器中
- ❑ 第三条指令**xor dl,cl**就是异或操作指令
  - 异或操作**xor dl,cl**，会将寄存器**edx**和**ecx**的异或后的结果放入到第一个操作数**edx**中
- ❑ 第四条指令，该指令将异或操作后得到的结果又放入到了用户输入的地址处。



# 1 异或加密

- ❑ 这里为什么会是逐字节进行异或的，就是通过  
在最开始的**xor eax,eax**指令将**eax**寄存器置为  
**0**，然后用**eax**当做数组的下标，每次增加**1**来  
实现循环取数据并进行异或操作
- ❑ 可以看到第四条**mov**下面的**inc eax**就是将**eax**  
的值增加**1**的指令，然后在下面一条指令**cmp**  
**eax, 0xA**中将**eax**的值与**10**进行比较，判断是  
否要退出循环了
- ❑ 最后在将**eax**的值与**10**比较，小于**10**就跳转到  
地址**0x401080**处（也就是循环的开始）继续执  
行--->实现循环





# 1 异或加密

```
.text:0040107E loc_40107E: ; CODE XREF: _main+3C1j
.text:0040107E xor     eax, eax
.text:00401080
.text:00401080 loc_401080: ; CODE XREF: _main+941j
.text:00401080 mov     cl, byte_409030[eax]
.text:00401086 mov     dl, [esp+eax+0Ch+var_C]
.text:0040108A xor     dl, cl
.text:0040108C mov     [esp+eax+0Ch+var_C], dl
.text:00401090 inc     eax
.text:00401091 cmp     eax, 0Ah
.text:00401094 jl      short loc_401080
.text:00401096 push    esi
.text:00401097 xor     esi, esi
.text:00401099
.text:00401099 loc_401099: ; CODE XREF: _main+AE1j
.text:00401099 movsx   eax, [esp+esi+10h+var_C]
.text:0040109E xor     edx, edx
.text:004010A0 mov     dl, byte_40903C[esi]
.text:004010A6 cmp     eax, edx
.text:004010A8 jnz     short loc_4010B2
.text:004010AA inc     esi
.text:004010AB cmp     esi, 0Ah
.text:004010AE jl      short loc_401099
.text:004010B0 jmp     short loc_4010BF
.text:004010B2 ; -----
.text:004010B2
.text:004010B2 loc_4010B2: ; CODE XREF: _main+A81j
.text:004010B2 push    offset aWrongAnswer ; "wrong answer!!!"
.text:004010B7 call    _puts
.text:004010BC add     esp, 4
.text:004010BF
.text:004010BF loc_4010BF: ; CODE XREF: _main+B01j
.text:004010BF cmp     esi, 0Ah
.text:004010C2 pop     esi
```

逐字节异或加密部分

判断是否是期望的加密结果



# 1 异或加密

- ❑ 理解了第一个方框的内容后第二个方框的内容就很好理解了，也是一个循环，只不过这里是利用**esi**寄存器来实现循环的（通过**xor esi,esi**将**esi**置0）
- ❑ 将用户输入的数据取出一字节，放入寄存器**eax**，将字节数组**0x40903C**处的数据（正确的密文）取出一字节放入**edx**中，接着比较**eax**与**edx**的值是否相等，不相等（**jnz short loc\_4010B2**）就跳到**0x4010B2**处执行（这里是输出**wrong answer**），否则继续循环，直到循环了**10**次为止



# 1 异或加密

- ❑ 从第一个方框是将输入的内容取出一个字节，加密后放回原处，所以在第二个方框里面用户输入的内容其实就是密文了，通过比较该密文是否和标准的密文（**0x40903C**处的数据）相等来判断用户输入的内容是否正确
- ❑ 因为在经过第一个方框的处理后用户输入的内容已经经过加密了（变成了密文），而在第二个方框又将该数据与字节数组**0x40903C**处的数据比较，看是否相等，所以**0x40903C**处的数据应该就是正确的密文了



# 1 异或加密

❑ 0x409030处的密钥和0x40903C处的正确密文值为

```
.data:00409030 aAbcdefg123      db 'abcdefg123',0      ; DATA XREF: _main:loc_401080↑r
.data:0040903B                align 4
.data:0040903C byte_40903C db 28h                ; DATA XREF: _main+A0↑r
.data:0040903D                db 3Dh ; =
.data:0040903E                db 24h ; $
.data:0040903F                db 54h ; T
.data:00409040                db 0Ah
.data:00409041                db 12h
.data:00409042                db 38h ; 8
.data:00409043                db 7Ah ; Z
.data:00409044                db 57h ; W
.data:00409045                db 4Ah ; J
.data:00409046                db 0
.data:00409047                db 0
.data:00409048 char aPause[1]
```



# 1 异或加密

□至此，整个程序的流程就很清楚了

- 首先获取用户的输入
- 如果输入的长度符合要求，便对其进行异或加密，加密的密钥为**0x409030**处的字节数组
- 最后将得到的密文与**0x40903C**处的字节数组进行比较，判断是否是期望的密文。



# 1 异或加密

- ❑ 在知道了程序使用的加密算法、加密密钥以及密文后，便可以对其进行解密
- ❑ 根据异或加密的原理，使用加密时的密钥来与密文进行异或即可完成解密



# 1 异或加密

## □ 解题程序

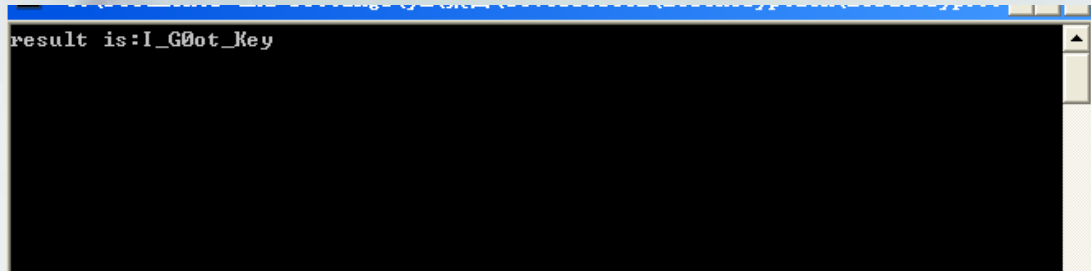
```
#include<stdio.h>
int main(){
    char token[11]="abcdefgh123";
    unsignedchar
ciphertext[]={0x28,0x3d,0x24,0x54,0x0a,0x12,0x38,0x7a,0x57,0x4a};
    char result[11];
    int i;

    for(i=0;i<10;++i){
        result[i]= token[i]^ciphertext[i];
    }
    result[i]='\0';
    printf("result is:%s\n",result);
    getchar();
    return0;
}
```

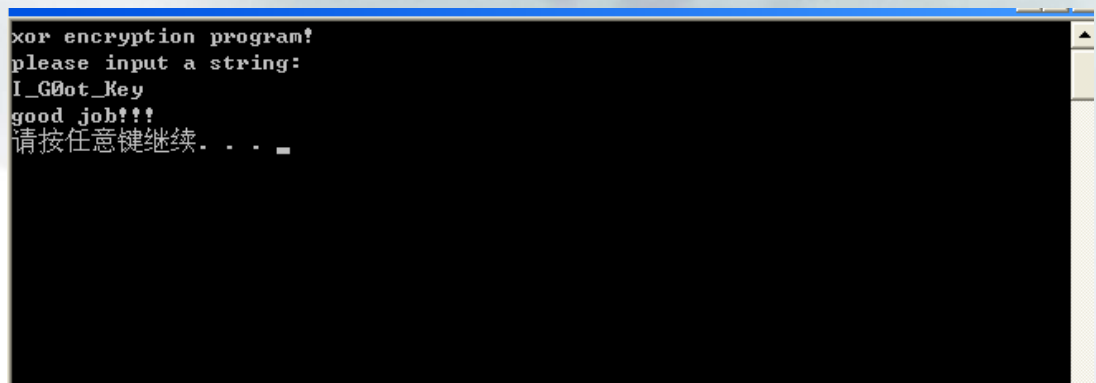


# 1 异或加密

- ❑ 运行该解密函数，得到正确的输入，并对结果进行验证



```
result is:I_G0ot_Key
```



```
xor encryption program!  
please input a string:  
I_G0ot_Key  
good job!!!  
请按任意键继续. . .
```





# 5.1 简单加密算法逆向分析

## @1 异或加密

## @2 仿射加密

- (1) 原理介绍
- (2) 加解密步骤
- (3) 逆向分析



## 2 仿射加密

### @ 古典密码

#### □ 置换密码

- 根据一定的规则重新排列明文

#### □ 代换密码

- 将明文中的字符串替换为其他字符
- 仿射加密便是代换密码中的一种



# 2 仿射加密

## @ 原理介绍

- 仿射加密的加密算法是一个线性变换，即对任意的明文字符 $x$ ，对应的密文字符 $y$ 为

$$y \equiv ax + b \pmod{26}$$

- 其中 $a$ 、 $b$ 为整数，且  $\gcd(a, 26) = 1$

- 上述变换是一一对应的

- 对于任意一个明文 $x$ ，有且仅有一个密文 $y$ 与之对应。  
对于任意一个密文 $y$ ，有且仅有一个明文 $x$ 与之对应

$x \equiv (y - b) * (a^{-1} \pmod{26})$  与之对应（ $a$  与 26 互素，故 $a^{-1}$ 存在）



## 2 仿射加密

### @ 加解密步骤

根据仿射密码的原理，加密过程为使用已选定的符合条件的密钥(a,b)，逐字符的对明文  $x$  进行  $y \equiv ax + b \pmod{26}$  的运算即可完成加密。解密过程则逐字符的每一个密文  $y$  进行  $x \equiv (y - b) * a^{-1} \pmod{26}$  的运算即可。

(注：如果  $a = 1$ ， $b = 3$  时，这种加密就是著名的凯撒密码)



## 2 仿射加密

### @ 逆向分析

- 用一个仿射加密的实例来进行分析，逆向分析程序fangsheenc.exe，查看程序的功能

```
please input a string:  
aaaaaadddddddd  
sorry  
请按任意键继续. . .
```

## 2 仿射加密

□ 使用IDA来分析该程序，定位到main函数的位置

```
.text:00401000
.text:00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401000 _main          proc near          ; CODE XREF: start+AF↓p
.text:00401000
.text:00401000 var_64          = byte ptr -64h
.text:00401000 var_63          = byte ptr -63h
.text:00401000 argc           = dword ptr  4
.text:00401000 argv           = dword ptr  8
.text:00401000 envp           = dword ptr 0Ch
.text:00401000
.text:00401000 sub          esp, 64h
.text:00401003 push         edi
.text:00401004 mov          ecx, 18h
.text:00401009 xor          eax, eax
.text:0040100B lea          edi, [esp+68h+var_63]
.text:0040100F mov          [esp+68h+var_64], 0
.text:00401014 push         offset aPleaseInputASt ; "please input a string:"
.text:00401019 rep stosd
.text:0040101B stosw
.text:0040101D stosb
.text:0040101E call         _puts
.text:00401023 lea          eax, [esp+6Ch+var_64]
```



## 2 仿射加密

- 程序首先获取用户的输入，并计算输入内容的长度，如果长度等于0，则直接跳转到0x40105A处执行，否则先执行完0x401047处的循环后再执行0x40105A处的代码

```
.text:00401027      push     eax
.text:00401028      push     offset aS          ; "%5"
.text:0040102D      call     _scanf
.text:00401032      lea      edi, [esp+74h+var_64]
.text:00401036      or       ecx, 0FFFFFFFh
.text:00401039      xor      eax, eax
.text:0040103B      add      esp, 0Ch
.text:0040103E      repne    scasb
.text:00401040      not      ecx
.text:00401042      dec      ecx
.text:00401043      test     ecx, ecx
.text:00401045      jle      short loc_40105A
.text:00401047
loc_401047:          ; CODE XREF: _main+58↓j
.text:00401047      mov      dl, [esp+eax+68h+var_64]
.text:00401048      cmp      dl, 61h
.text:0040104E      jl       short loc_4010B3
.text:00401050      cmp      dl, 7Ah
.text:00401053      jg       short loc_4010B3
.text:00401055      inc      eax
.text:00401056      cmp      eax, ecx
.text:00401058      jl       short loc_401047
.text:0040105A
loc_40105A:          ; CODE XREF: _main+45↓j
.text:0040105A      push     ebx
.text:0040105B      push     esi
.text:0040105C      xor      esi, esi
.text:0040105E      test     ecx, ecx
.text:00401060      jle      short loc_401082
.text:00401062
```



## 2 仿射加密

- ❑ 分析上述汇编代码，可以知道**0x401047**处的循环为判断用户输入的内容是否有小于**0x61**（对应字符为‘a’）或大于**0x7A**（对应字符为‘z’）的情况，有则跳转到**0x4010B3**处

```
.text:004010B3 loc_4010B3:                                ; CODE XREF: _main+4E1j  
.text:004010B3                                ; _main+531j  
.text:004010B3          or      eax, 0FFFFFFFh  
.text:004010B6          pop     edi  
.text:004010B7          add     esp, 64h  
.text:004010BA          retn  
.text:004010BD
```

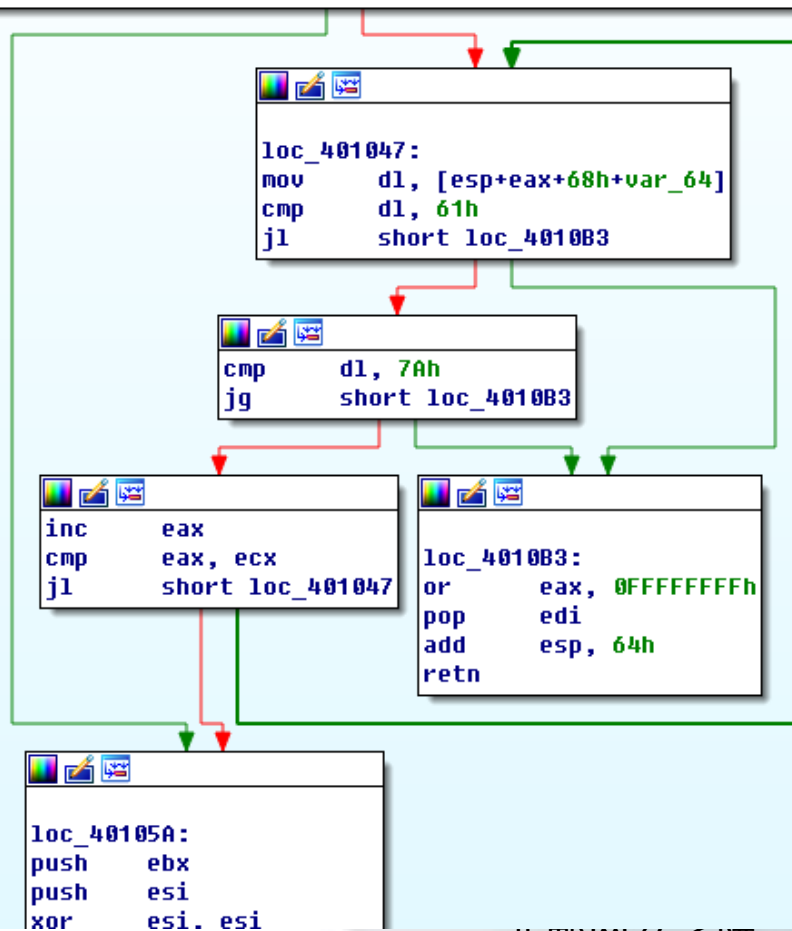




```

call    _scanf
lea     edi, [esp+74h+var_64]
or      ecx, 0FFFFFFFh
xor     eax, eax
add     esp, 0Ch
repne scasb
not     ecx
dec     ecx
test    ecx, ecx
jle     short loc_40105A

```



## 2 仿射加密

- ❑ 在对输入进行判断之后便来到仿射加密的关键部分了，这里可以看到几个特殊的值**0x1A**、**0x61**等。首先将用户输入的内容逐字节取出，放入寄存器**eax**中，并将 **$eax+eax*2-0x11C=3*eax-0x11C$** 的结果放入**eax**中；用**eax**当做被除数，**edi**当做除数（**1A**即**26**），得到的余数再加上**0x61**（**a**的**ASCII**码）后写入内存

```
.text:00401062
.text:00401062 loc_401062:                                ; CODE XREF: _main+80↓j
.text:00401062      movsx   eax, [esp+esi+70h+var_64]
.text:00401067      mov     edi, 1Ah
.text:0040106C      lea     eax, [eax+eax*2-11Ch]
.text:00401073      cdq
.text:00401074      idiv    edi
.text:00401076      add     dl, 61h
.text:00401079      mov     [esp+esi+70h+var_64], dl
.text:0040107D      inc     esi
.text:0040107E      cmp     esi, ecx
.text:00401080      jl      short loc_401062
.text:00401082
```



## 2 仿射加密

那么这里如何根据  $3 * \text{eax} - 0x11C$  来推算出加密密钥  $a$ 、 $b$  的值呢？结合加密算法： $a * (\text{eax} - 0x61) + b$ ，将其进行因式分解得到： $a * \text{eax} - a * 0x61 + b$ 。这样就能很容易的得到  $a$  的值为 3，再将  $a = 3$  带入即可得到  $b$  的值为 7。

☐  $3 * \text{eax} - 0x11C = a * \text{eax} - a * 0x61 + b$

☐  $a = 3, b = 7$



## 2 仿射加密

- ❑ 程序在完成加密后，便将加密得到的结果与 0x401082处的字符串（密文）进行比较，如果相等，则输出“ok, you really know”

```
.text:00401082
.text:00401082 loc_401082: ; CODE XREF: _main+60↑j
.text:00401082 mov     esi, offset aQxbxpluxvwhuzj ; "qxbxpluxvwhuzjct"
.text:00401087 lea     eax, [esp+70h+var_64]
.text:0040108B loc_40108B: ; CODE XREF: _main+AD↓j
.text:0040108B mov     dl, [eax]
.text:0040108D mov     bl, [esi]
.text:0040108F mov     cl, dl
.text:00401091 cmp     dl, bl
.text:00401093 jnz     short loc_4010BB
.text:00401095 test    cl, cl
.text:00401097 jz      short loc_4010AF
.text:00401099 mov     dl, [eax+1]
.text:0040109C mov     bl, [esi+1]
.text:0040109F mov     cl, dl
.text:004010A1 cmp     dl, bl
.text:004010A3 jnz     short loc_4010BB
.text:004010A5 add     eax, 2
.text:004010A8 add     esi, 2
.text:004010AB test    cl, cl
.text:004010AD jnz     short loc_40108B
.text:004010AF loc_4010AF: ; CODE XREF: _main+97↑j
.text:004010AF xor     eax, eax
.text:004010B1 jmp     short loc_4010C0
.text:004010B2
```



## 2 仿射加密

- 因此，我们知道了程序加密采用的密钥为 $a=3$ ， $b=7$ ；密文为0x401082处的字符串“qxbxpluxvwhuzjct”。



## 2 仿射加密

对于仿射密码的解密，根据解密公式 $x \equiv (y - b) * (a^{-1} \bmod 26)$ ，需要计算出来 $a^{-1}$ ，这里根据数论相关知识计算出来 $a^{-1} = 9$ 。



# 2 仿射加密

## □编写的解密程序如下

```
#include<stdio.h>
#include<string.h>
int main()
{
    int key_a =3;
    int key_b =7;
    int re_key_a =9;
    char ciphertext[]="qxbxpluxvwhuzjct";
    char result[20]={0};
    int i;
    int len;
    int temp;

    len = strlen(ciphertext);
    for(i=0;i<len;++i){
        temp=ciphertext[i]-'a';
        temp=((temp-key_b+26)*re_key_a)%26;
        result[i]=temp+'a';
    }
    printf("plaintext is:\n%s\n",result);
    system("pause");
}
```

北邮网安学院 崔宝江



## 2 仿射加密

- ❑ 运行该解密函数，得到解密结果

```
plaintext is:  
doyouknowfangshe  
请按任意键继续. . .
```

- ❑ 验证其正确性

```
please input a string:  
doyouknowfangshe  
ok, you really know  
请按任意键继续. . .
```





# 第五章 常见加密算法逆向分析

④ 5.1 简单加密算法逆向分析

④ 5.2 对称加密算法逆向分析

④ 5.3 单向散列算法逆向分析



## 5.2 对称加密算法逆向分析

- ④ 加密和解密时使用相同的密钥，或是使用两个简单的可以相互推算的密钥
- ④ 本节将对两种对称加密算法进行介绍
  - ❑ RC4
  - ❑ DES



# 对称加密算法 RC4

## @ 原理介绍

- ❑ RC4本质上是流密码算法，利用生成的密钥流序列和输入明文进行异或完成加密。
- ❑ 密钥流的生成以一个足够大的数组为基础，对其进行非线性变换，把这个大数组称为**S**盒。
- ❑ RC4的处理包括两个过程
  - 一个是密钥调度算法来置乱**S**盒的初始排列
  - 另一个是伪随机生成算法，来输出随机序列并修改**S**盒的当前排列顺序



# 对称加密算法 RC4

## ④ 密钥调度算法

□ 密钥调度算法是根据用户选定的密钥**K**:

$$K(0 \leq \text{len}(k) \leq 256)$$

□ 依次对数组中的数据进行换位，进而打乱**S**盒的初始排序

○ 其中，如果**K**的长度小于**256**，则将**K**重复拼接起来，直到长度为**256**为止。



# 对称加密算法 RC4

## ④ 伪随机生成算法

- 是利用初始化后的**S**盒，按照一定的规则从中选取数据输出，同时更新**S**盒的排列顺序，达到生成伪随机序列的目的。



# 对称加密算法 RC4

## ④ 加解密步骤

- ❑ RC4加解密的关键步骤在于按照密钥生成伪随机序列，得到伪随机序列后直接与明/密文进行异或操作即可。

## ④ 生成伪随机序列的过程

- ❑ 由密钥初始化S盒
- ❑ 生成密钥流



# 对称加密算法 RC4

## □ 由密钥初始化S盒

- S盒的长度为**256**，程序首先会将**0**到**255**的互不重复的元素装入S盒，使得

$$S[i] = i (0 \leq i \leq 255)$$

- 同时建立一个长度为**256**的临时数组T，如果密钥K的长度等于**256**字节，那么直接将密钥的值赋给T，否则将K的元素依次赋给T，并不断重复的将K的值赋给T，直到T被填满。



# 对称加密算法 RC4

❑ 伪代码如下

```
for i from 0 to 255
    S[i] := i
end for
j := 0
for( i=0; i<256; i++)
    j := (j + S[i] + key[i mod keylength]) % 256
    swap values of S[i] and S[j]
end for
```





# 对称加密算法 RC4

## □ 生成密钥流

- 利用第一步中得到的S盒便可开始生成用于加密的密钥流了
- 生成密钥流的过程中，根据i、j的值来确定选取S盒的哪个元素，并更新S盒中元素的排列顺序



# 对称加密算法 RC4

□ 下列伪代码描述了生成密钥流的过程

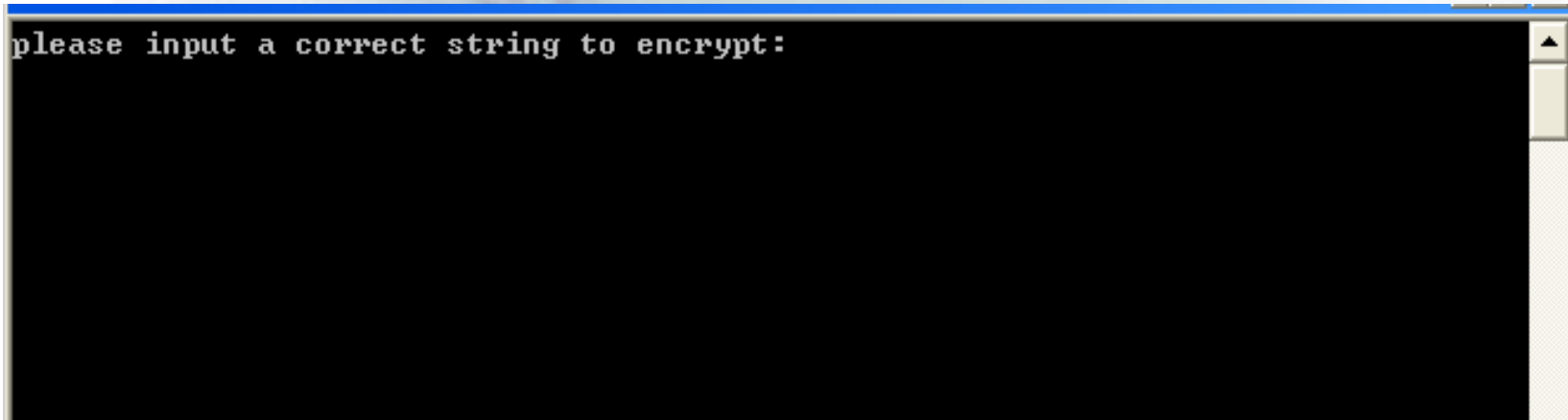
```
i := 0
j := 0
while GeneratingOutput:
    i := (i + 1) mod 256
    j := (j + S[i]) mod 256
    swap values of S[i] and S[j]
    t := (S[i] + S[j]) mod 256
    k := inputByte ^ S[t]
    output k
end while
```



# 对称加密算法 RC4

## @ 逆向分析

- ❑ 运行程序rc4enc.exe，查看程序的功能
- ❑ 首先需要输入一个正确的字符串来加密



```
please input a correct string to encrypt:
```



# 对称加密算法 RC4

- ❑ 使用**IDA**打开该程序进行分析，定位到**main**函数
- ❑ 由于**RC4**加密算法较复杂，可使用**IDA**的**F5**插件来反编译程序，在反编译出来的伪代码基础上进行分析



# 对称加密算法 RC4

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     unsigned int v3; // kr04_4@1
4     int v4; // eax@1
5     int result; // eax@4
6     char v6; // [sp+8h] [bp-C8h]@1
7     char v7; // [sp+9h] [bp-C7h]@1
8     __int16 v8; // [sp+69h] [bp-67h]@1
9     char v9; // [sp+6Bh] [bp-65h]@1
10    char v10; // [sp+6Ch] [bp-64h]@1
11
12    v6 = 0;
13    memset(&v7, 0, 0x60u);
14    v8 = 0;
15    v9 = 0;
16    puts(aPleaseInputACo);
17    scanf(aS, &v10);
18    v3 = strlen(&v10) + 1;
19    sub_4011C0(&v10, &v6);
20    v4 = 0;
21    if ( (signed int)(v3 - 1) <= 0 )
22    {
23    LABEL_4:
24        puts(aGreat);
25        system(aPause);
26        result = 0;
27    }
28    else
29    {
30        while ( *(&v6 + v4) == byte_409130[v4] )
31        {
32            if ( ++v4 >= (signed int)(v3 - 1) )
33                goto LABEL_4;
34        }
35        result = -1;
36    }
37    return result;
```



# 对称加密算法 RC4

- ❑ 这里可以很清楚的看到main函数主要是
  - 获取用户的输入并将其保存到v10中;
  - 接着将v10作为参数调用了函数0x4011C0;
  - 最后还有一个比较的过程, 将v6指向的内容与字节数组0x409130处的内容进行比较
  - 同时发现v6也作为参数传递给了函数0x4011C0
  - 于是猜测这里v6就是用户的输入加密后得到的密文



# 对称加密算法 RC4

❑ 字节数组0x409130处的正确密文

```
.data:00409130 ; char byte_409130[]  
.data:00409130 byte_409130      db 1Bh                ; DATA XREF: _main+62↑r  
.data:00409131                db 0CAh ;  
.data:00409132                db 0AEh ;  
.data:00409133                db 0EFh ;  
.data:00409134                db 1Eh  ;  
.data:00409135                db 95h  ;  
.data:00409136                db 4Bh  ; K  
.data:00409137                db 0C2h ;  
.data:00409138                db 0D5h ;  
.data:00409139                db 0E3h ;  
.data:0040913A                db 33h  ; 3  
.data:0040913B                db 76h  ; U  
.data:0040913C                db 4Fh  ; 0  
.data:0040913D                db 0F9h ;  
.data:0040913E                db 4Fh  ; 0  
.data:0040913F                db 0D2h ;  
.data:00409140                db 0FCh ;  
.data:00409141                db 60h  ;  
.data:00409142                db 96h  ;  
.data:00409143                db      0
```



# 对称加密算法 RC4

- ❑ 下面跟进函数 **0x4011C0**，发现函数中存在着两个函数调用，并且第二个函数调用 **0x401130** 的返回值还与 **v5[v6]** 进行了异或操作
- ❑ 这里即用户输入的内容

$$v5[v6] = v5 + v6 = a2 + a4 - a2 = a4 = a1$$





# 对称加密算法 RC4

```
1 int __cdecl sub_4011C0(const char *a1, _BYTE *a2)
2 {
3     unsigned int v2; // kr04_4@1
4     int result; // eax@1
5     const char *v4; // edi@2
6     _BYTE *v5; // esi@2
7     int v6; // edi@2
8     int v7; // [sp+10h] [bp+4h]@2
9
10    dword_40BE90 = 0;
11    dword_40BE94 = 0;
12    v2 = strlen(a1) + 1;
13    sub_4010A0();
14    result = 0;
15    if ( (signed int)(v2 - 1) <= 0 )
16    {
17        *a2 = 0;
18    }
19    else
20    {
21        v4 = a1;
22        v7 = v2 - 1;
23        v5 = a2;
24        v6 = v4 - a2;
25        do
26        {
27            *v5 = v5[v6] ^ sub_401130();
28            ++v5;
29            result = v7-- - 1;
30        }
31        while ( v7 );
32        a2[v2 - 1] = 0;
33    }
34    return result;
35 }
```



# 对称加密算法 RC4

- ❑ 跟进函数**0x4010A0**，该函数主要有两个循环
  - 第一个循环是将地址**0x40BA90**处开始的值赋值为**0,1,2,3.....255**;
  - 第二个循环主要是根据字符串“**RC4key**”的值来对这**256**个值进行交换操作。
- ❑ 看到这里，应该要很快的反应过来这里是**RC4**加密中的初始化**S**盒过程
  - 其中密钥为字符串“**RC4key**”，而地址**0x40BA90**指向的内存便是**S**盒



# 对称加密算法 RC4

```
1 int sub_4010A0()
2 {
3     unsigned int v0; // kr04_4@1
4     int v1; // edx@1
5     int *v2; // eax@1
6     int v3; // edi@3
7     signed int v4; // ebx@3
8     int *v5; // esi@3
9     int result; // eax@4
10    unsigned __int8 v7; // ST0C_1@4
11
12    v0 = strlen(aRc4key) + 1;
13    v1 = 0;
14    v2 = dword_40BA90;
15    do
16    {
17        *v2 = v1;
18        ++v2;
19        ++v1;
20    }
21    while ( (signed int)v2 < (signed int)&dword_40BE90 );
22    v3 = 0;
23    v4 = 0;
24    v5 = dword_40BA90;
25    do
26    {
27        v3 = (*v5 + (unsigned __int8)aRc4key[v4 % (signed int)(v0 - 1)] + v3) % 256;
28        result = dword_40BA90[v3];
29        v7 = *(_BYTE *)v5;
30        *v5 = result;
31        ++v5;
32        ++v4;
33        dword_40BA90[v3] = v7;
34    }
35    while ( (signed int)v5 < (signed int)&dword_40BE90 );
36    return result;
37 }
```



# 对称加密算法 RC4

- ❑ 现在知道了初始化S盒的函数，且知道函数 **0x401130** 的返回值与用户的输入进行了异或操作
- ❑ 则很容易的猜到函数 **0x401130** 就是生成密钥流的函数了
- ❑ 利用 **IDA** 的快捷键 ‘N’ 来对变量重新命名，以便于更好的分析



# 对称加密算法 RC4

```
1 char Generate_Key()
2 {
3     int v0; // eax@1
4     signed int v1; // ecx@1
5     unsigned __int8 v2; // dl@1
6
7     v0 = (pos_i + 1) % 256;
8     v1 = pos_j + *(_DWORD *)&Sbox[4 * v0];
9     pos_i = (pos_i + 1) % 256;
10    v1 %= 256;
11    v2 = Sbox[4 * v0];
12    pos_j = v1;
13    *(_DWORD *)&Sbox[4 * v0] = *(_DWORD *)&Sbox[4 * v1];
14    *(_DWORD *)&Sbox[4 * v1] = v2;
15    return Sbox[4 * ((v2 + *(_DWORD *)&Sbox[4 * v0]) % 256)];
16 }
```



# 对称加密算法 RC4

- ❑ 分析生成密钥流的函数，可以看到：利用两个变量 **pos\_i**, **pos\_j** 来选择一个 **S** 盒中的值作为函数的返回值，并打乱 **S** 盒的数据（交换）。
- ❑ 因此，知道了加密算法为 **RC4**，加密的密钥为字符串 “**RC4key**”，最终的密文为字节数组 **0x409130** 处的值，便可以编写解密函数来解密了。
- ❑ 对于 **RC4** 加密算法来说，解密只需要生成和加密过程一样的密钥流即可。



# 对称加密算法 RC4

❑ 解密函数如下（因算法较长，仅列出关键函数，其中初始化S盒以及生成密钥流的部分与加密算法相同）：

```
int main()  
{  
    char result[MAX_STR]={0};  
    int len;  
    unsigned char  
cipher[MAX_STR]={0x1b,0xca,0xae,0xef,0x1e,0x95,0x4b,0xc2,0xd5,0x  
e3,0x33,0x76,0x4f,0xf9,0x4f,0xd2,0xfc,0x60,0x96,0x0};  
    decryption((unsigned char*)cipher,(unsigned char*)result);  
    printf("%s",result);  
    system("pause");  
    return 0;  
}
```



# 对称加密算法 RC4

```
void decryption(unsignedchar* ciphertext,unsignedchar*result){
    pos_i =0;
    pos_j =0;
    int len = strlen((constchar*)ciphertext);
    int i=0;
    init_sbox();
    for(i=0;i<len;++i)
        result[i]= ciphertext[i]^generate_key();
    result[i]='\0';
}
```





# 对称加密算法 RC4

❑ 运行解密函数，得到正确的输入

```
Is_Th13_Simple_Rc4?请按任意键继续. . .
```

❑ 进行验证

```
please input a correct string to encrypt:  
Is_Th13_Simple_Rc4?  
Great!!!  
请按任意键继续. . .
```



## 5.2 对称加密算法逆向分析

### @ 两种对称加密算法进行介绍

□ RC4

□ DES



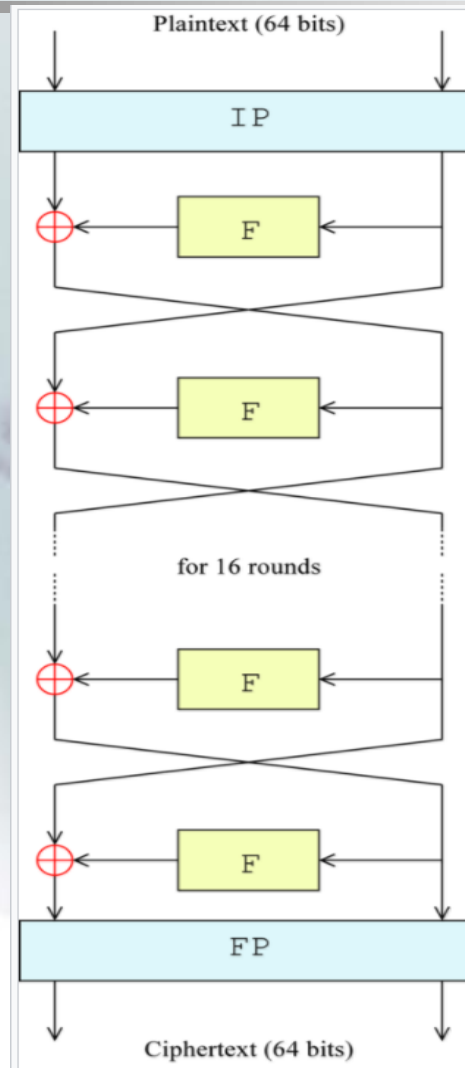
# 对称加密算法 DES

## @ 原理介绍

- ❑ **DES**属于对称密码体制，加解密使用相同的密钥，有效密钥的长度为**56**位。
- ❑ **DES**为分组密码算法，分组长度为**64**位，使用**Feistel**的结构作为加解密的基本结构。



# 对称加密算法 DES



# 对称加密算法 DES

- ❑ 首先,将输入的**64**位明文进行一个初始置换（**IP**），并将得到的结果分为左右两个分组，各为**32**位。
- ❑ 进行初始置换后，对左右两个分组进行**16**轮相同轮函数的迭代，每轮迭代都有置换和代换。需要注意的是最后一轮迭代输出不进行左右两个分组的交换。
- ❑ 经**16**轮迭代后，得到的结果再经逆初始置换（**FP**）的作用后,作为加密的最终输出。



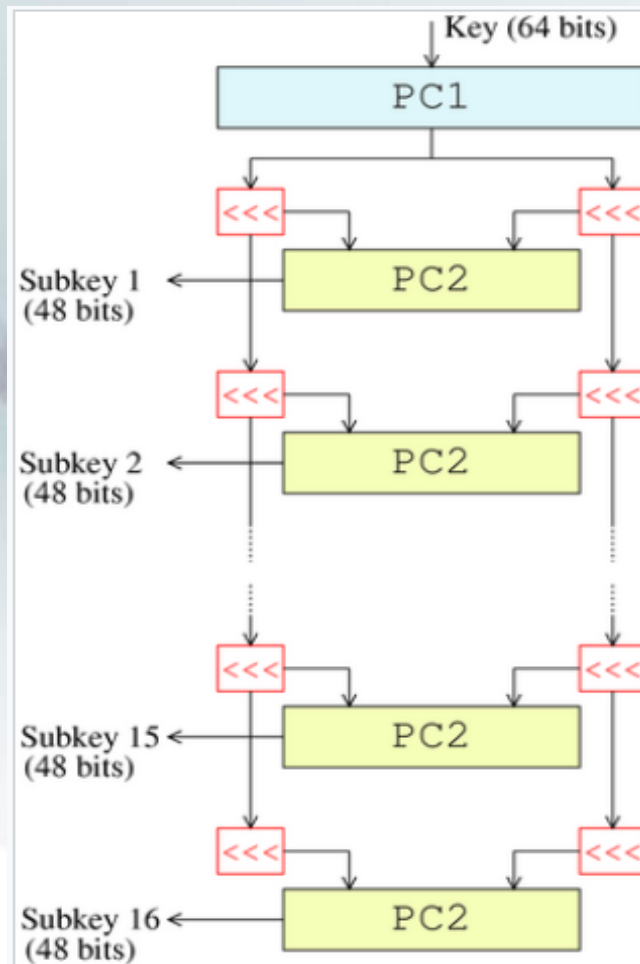
# 对称加密算法 DES

## ④ 密钥生成算法

- ❑ 在加密过程的每一轮迭代中，轮函数 $F$ 需要在右边分组和每一轮的子密钥的控制下得到输出，并与左边分组进行异或操作。
- ❑ 每一轮子密钥就是在密钥生成算法的控制下产生的。



# 对称加密算法 DES



# 对称加密算法 DES

- ❑ 从图中可以看到，对于用户输入的**64**位密钥，先经过置换选择**PC-1**得到有效的**56**位密钥——剩下的**8**位要么直接丢弃，要么作为奇偶校验位。
- ❑ 然后将得到的**56**位密钥分为左右两个**28**位的半密钥。在接下来的**16**轮中，左右两个半密钥都被左移**1**或**2**位（具体由轮数决定），然后通过置换选择**PC-2**产生**48**位的子密钥。





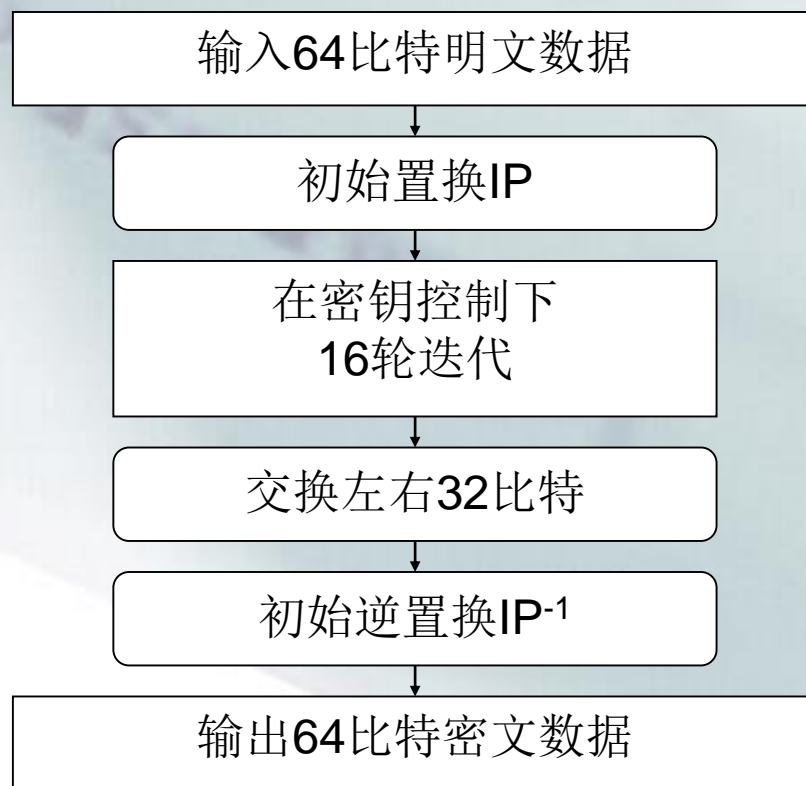
# 对称加密算法 DES

## ④ 加解密步骤



# 对称加密算法 DES

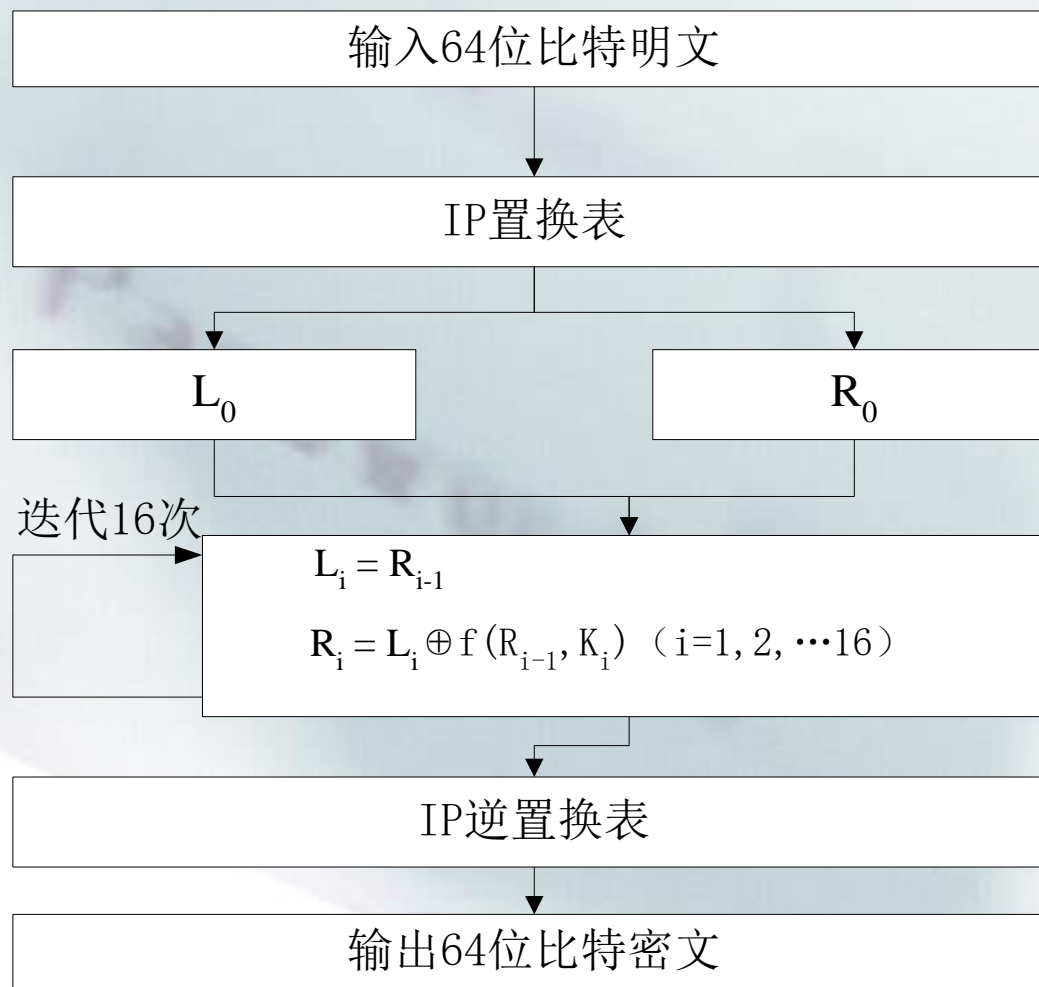
DES利用56比特串长度的密钥K来加密长度为64位的明文，得到长度为64位的密文



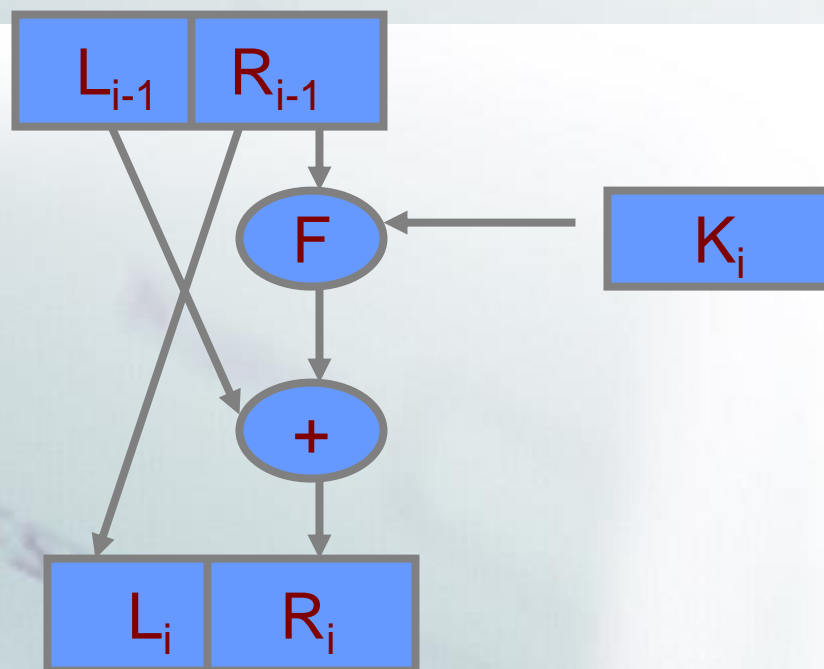
崔宝江



# 对称加密算法 DES



# 对称加密算法 DES



一轮加密的简图

# 对称加密算法 DES

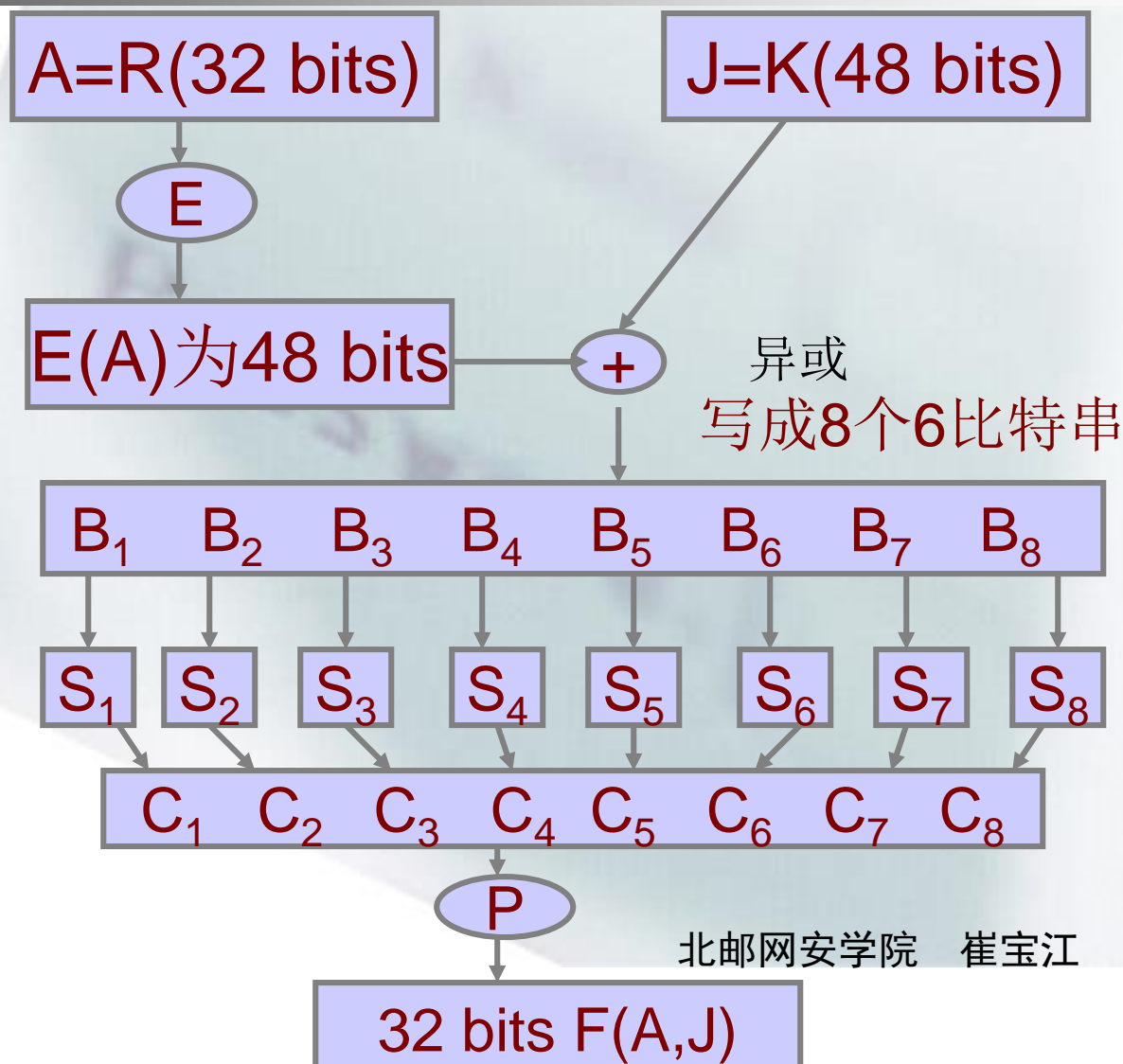
## F函数说明

$F(R_{i-1}, K_i)$  函数F以长度为32的比特串 $A=R$  (32bits) 作第一个输入, 以长度为48的比特串变元 $J=K$  (48bits) 作为第二个输入。产生的输出为长度为32的位串。

- (1) 对第一个变元A, 由给定的扩展函数E, 将其扩展成48位串 $E(A)$ ;
- (2) 计算 $E(A) + J$ , 并把结果写成连续的8个6位串,  $B=b_1b_2b_3b_4b_5b_6b_7b_8$ ;
- (3) 使用8个S盒, 每个 $S_j$ 是一个固定的 $4 \times 16$ 矩阵, 它的元素取0~15的整数。给定长度为6个比特串, 如  $B_j=b_1b_2b_3b_4b_5b_6$ , 计算 $S_j(B_j)$ 如下:  $b_1b_6$ 两个比特确定了 $S_j$ 的行数,  $r(0 \leq r \leq 3)$ ; 而 $b_2b_3b_4b_5$ 四个比特确定了 $S_j$ 的列数 $c(0 \leq c \leq 15)$ 。最后 $S_j(B_j)$ 的值为S-盒矩阵 $S_j$ 中 $r$ 行 $c$ 列的元素 $(r, c)$ , 得 $C_j=S_j(B_j)$ ;
- (4) 最后, 进行固定置换P。



# 对称加密算法 DES



# 对称加密算法 DES

(1) 给定64位的密钥K，放弃奇偶校验位（8，16，...，64）并根据固定置换PC1来排列K中剩下的位。我们写

$$PC1(K)=C_0D_0$$

其中 $C_0$ 由PC1(K)的前28位组成； $D_0$ 由后28位组成；

(2) 对 $1 \leq i \leq 16$ ，计算

$$C_i = LS_i(C_{i-1})$$

$$D_i = LS_i(D_{i-1})$$

$LS_i$ 表示循环左移2或1个位置，取决于 $i$ 的值。  
 $i=1,2,9$ 和 $16$  时移1个位置，否则移2位置；

(3)  $K_i = PC2(C_iD_i)$ ，PC2为固定置换。北邮网安学院 崔宝江



# 对称加密算法 DES





# 对称加密算法 DES

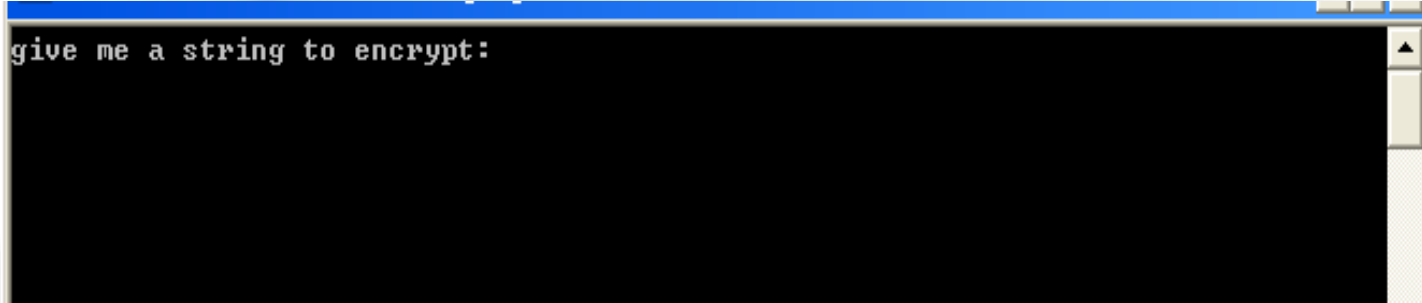
## @ 逆向分析

- ❑ 通过分析一个**DES**加密的示例来加深一下对**DES**加密算法的理解
- ❑ 由于**DES**加密算法较为复杂，仍然在**F5**插件反编译得到的伪代码基础上进行分析，并且主要对一些关键的函数进行分析



# 对称加密算法 DES

- ❑ 运行 **desenc.exe**，发现程序需要我们输入一个字符串来加密



```
give me a string to encrypt:
```



# 对称加密算法 DES

- ❑ 使用IDA打开desenc.exe，定位到main函数，查看程序大致流程
  - 首先获取用户的输入，将其存放到分配的栈空间v9中；
  - 接着判断输入内容的长度是否符合要求；
  - 在长度符合要求的情况下将地址0x409070处的数据作为参数传入函数sub\_401560中；
  - 接着将用户的输入作为参数传递给函数sub\_4010B0；
  - 最后还有一个字节数组之间的比较



# 对称加密算法 DES

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    int v4; // eax@3
    char v5[8]; // [sp+4h] [bp-28h]@3
    int v6; // [sp+Ch] [bp-20h]@1
    int v7; // [sp+10h] [bp-1Ch]@1
    char v8; // [sp+14h] [bp-18h]@1
    char v9; // [sp+18h] [bp-14h]@1

    v6 = dword_409070;
    v7 = dword_409074;
    v8 = byte_409078;
    puts(aGiveMeAStringT);
    scanf(aS, &v9);
    if ( strlen(&v9) == 8 )
    {
        sub_401560(&v6);
        sub_4010B0(&v9, v5);
        v4 = 0;
        while ( v5[v4] == byte_409030[v4] )
        {
            if ( ++v4 >= 8 )
            {
                puts(aG00dJob);
                system(aPause);
                return 0;
            }
        }
    }
    return -1;
}
```



# 对称加密算法 DES

- 下面跟进函数 `sub_401560`，该函数的参数是一块固定的8字节数据，猜测是产生密钥的函数。该函数中又存在着多处函数调用

```
int __cdecl sub_401560(int a1)
{
    int v1; // ebx@1
    void *v2; // ebp@1
    int result; // eax@2
    char *v4; // edi@2
    char v5; // [sp+10h] [bp-A8h]@1
    char v6; // [sp+2Ch] [bp-8Ch]@2
    char v7; // [sp+48h] [bp-70h]@2
    char v8; // [sp+78h] [bp-40h]@1

    sub_4011E0(a1, (int)&v8, 8);
    sub_401490(&v8, &v5);
    v1 = 0;
    v2 = &unk_40B930;
    do
    {
        sub_4014F0(&v5, &v5, byte_408208[v1]);
        sub_4014F0(&v6, &v6, byte_408208[v1]);
        result = sub_4014C0(&v5, &v7);
        v4 = (char *)v2;
        v2 = (char *)v2 + 48;
        ++v1;
        qmemcpy(v4, &v7, 0x30u);
    }
    while ( (signed int)v2 < (signed int)& dword_40BC30 );
    return result;
}
```



# 对称加密算法 DES

- 对于函数调用**sub\_4011E0**，将固定的数据作为参数传递给它，并将处理后的结果放入栈空间**v8**中，跟进该函数发现是该函数取出参数中每一字节数据的每一位，并将每一位数据再放入一个字节中；

```
1 int __cdecl sub_4011E0(int a1, int a2, int a3)
2 {
3     int result; // eax@1
4     int v4; // esi@1
5     int v5; // edi@2
6     signed int v6; // eax@3
7     signed int v7; // edx@3
8
9     result = a3;
10    v4 = 0;
11    if ( a3 > 0 )
12    {
13        v5 = a2;
14        do
15        {
16            v6 = 0;
17            v7 = 7;
18            do
19            {
20                *(_BYTE *)(v5 + v6++) = (*(_BYTE *)(v4 + a1) >> v7--) & 1;
21                while ( v6 < 8 );
22                result = a3;
23                ++v4;
24                v5 += 8;
25            } while ( v4 < a3 );
26        }
27        return result;
28    }
```



# 对称加密算法 DES

- 函数调用**sub\_401490**则是将栈中的数据**v8**作为参数进行处理，一共进行了**56**轮循环，每一轮循环都是根据字节数组**0x4081A0**处的值来将**v8**中的数据放入输出地址**a2**处，经过该函数，输入的**64**字节数据变成了**56**字节数据，联想到**DES**生成密钥的过程我们知道这是**PC-1**置换选择

```
int __cdecl sub_401490(int a1, int a2)
{
    int result; // eax@1
    signed int v3; // esi@1
    int v4; // edx@2

    result = a2;
    v3 = 56;
    do
    {
        v4 = (&byte_4081A0[result++] - a2);
        --v3;
        *(_BYTE *)(result - 1) = *(_BYTE *)(v4 + a1 - 1);
    }
    while ( v3 );
    return result;
}
```



# 对称加密算法 DES

❑ 字节数组0x4081A0处的数据如下，与标准的PC-1表进行对照发现该处就是PC-1置换表

```
.rdata:004081A0 ; _BYTE byte_4081A0[56]
.rdata:004081A0 byte_4081A0      db 39h, 31h, 29h, 21h, 19h, 11h, 9, 1, 3Ah, 32h, 2Ah, 22h
.rdata:004081A0                                     ; DATA XREF: sub_401490+510
.rdata:004081A0      db 1Ah, 12h, 0Ah, 2, 3Bh, 33h, 2Bh, 23h, 1Bh, 13h, 0Bh
.rdata:004081A0      db 3, 3Ch, 34h, 2Ch, 24h, 3Fh, 37h, 2Fh, 27h, 1Fh, 17h
.rdata:004081A0      db 0Fh, 7, 3Eh, 36h, 2Eh, 26h, 1Eh, 16h, 0Eh, 6, 3Dh, 35h
.rdata:004081A0      db 2Dh, 25h, 1Dh, 15h, 0Dh, 5, 1Ch, 14h, 0Ch, 4
```





# 对称加密算法 DES

- 知道了函数sub\_401560为生成子密钥的函数
  - 便能知道函数sub\_4014F0就是循环移位的函数，分别对56位密钥中的左右两个子密钥进行移位。
  - 同样函数sub\_4014C0应该就是PC-2置换选择函数了，跟进该函数进行验证，发现其中也有一个查表的操作，而表中的数据刚好是PC-2置换选择表

```
.rdata:004081D8 ; _BYTE byte_4081D8[48]
.rdata:004081D8 byte_4081D8      db 0Eh, 11h, 0Bh, 18h, 1, 5, 3, 1Ch, 0Fh, 6, 15h, 0Ah
.rdata:004081D8                                     ; DATA XREF: sub_4014C0+5↑o
.rdata:004081D8      db 17h, 13h, 0Ch, 4, 1Ah, 8, 10h, 7, 1Bh, 14h, 0Dh, 2
.rdata:004081D8      db 29h, 34h, 1Fh, 25h, 2Fh, 37h, 1Eh, 28h, 33h, 2Dh, 21h
.rdata:004081D8      db 30h, 2Ch, 31h, 27h, 38h, 22h, 35h, 2Eh, 2Ah, 32h, 24h
.rdata:004081D8      db 1Dh, 20h
```

# 对称加密算法 DES

## @ 函数sub\_4014F0就是循环移位的函数

```
1 int __cdecl sub_401560(int a1)
2 {
3     int v1; // ebx@1
4     void *v2; // ebp@1
5     int result; // eax@2
6     char *v4; // edi@2
7     char v5; // [sp+10h] [bp-A8h]@1
8     char v6; // [sp+2Ch] [bp-8Ch]@2
9     char v7; // [sp+48h] [bp-70h]@2
10    char v8; // [sp+78h] [bp-40h]@1
11
12    sub_4011E0(a1, (int)&v8, 8);
13    sub_401490(&v8, &v5);
14    v1 = 0;
15    v2 = &unk_40B930;
16    do
17    {
18        sub_4014F0(&v5, &v5, byte_408208[v1]);
19        sub_4014F0(&v6, &v6, byte_408208[v1]);
20        result = sub_4014C0(&v5, &v7);
21        v4 = (char *)v2;
22        v2 = (char *)v2 + 48;
23        ++v1;
24        qmemcpy(v4, &v7, 0x30u);
25    }
26    while ( (signed int)v2 < (signed int)&dword_40BC30 );
27    return result;
28 }
```



# 对称加密算法 DES

□ 下面返回main，看一下加密的函数sub\_4010B0

```
int __cdecl main(int argc, const char **argv, const char **envp)
```

```
{  
    int v4; // eax@3  
    char v5[8]; // [sp+4h] [bp-28h]@3  
    int v6; // [sp+Ch] [bp-20h]@1  
    int v7; // [sp+10h] [bp-1Ch]@1  
    char v8; // [sp+14h] [bp-18h]@1  
    char v9; // [sp+18h] [bp-14h]@1
```

```
    v6 = dword_409070;  
    v7 = dword_409074;  
    v8 = byte_409078;  
    puts(aGiveMeAStringT);
```

```
    scanf(aS, &v9);
```

```
    if ( strlen(&v9) == 8 )
```

```
    {
```

```
        sub_401560(&v6);
```

```
        sub_4010B0(&v9, v5);
```

```
        v4 = 0;
```

```
        while ( v5[v4] == byte_409030[v4] )
```

```
        {
```

```
            if ( ++v4 >= 8 )
```

```
            {
```

```
                puts(aG00dJob);
```

```
                system(aPause);
```

```
                return 0;
```

```
            }
```

```
        }
```

```
    }
```



# 对称加密算法 DES

- ❑ 下面跟进进行加密的函数**sub\_4010B0**，函数的开始调用了函数**byte2Bits (sub\_4011E0)**将用户的输入中的每一位（一共有位）提取出来转换成一个**64**字节的数组；
- ❑ 接下来调用函数**sub\_401270**对这**64**字节数组进行处理，这就是加密过程中的**IP**置换；
- ❑ 接着调用了两次**memcpy**将置换后的数据分成两个**32**字节的数组，也即**16**轮迭代中的左右两个分组；



# 对称加密算法 DES

```
int __cdecl sub_4010B0(int a1, int a2)
{
    void *v2; // ebx@1
    char v4; // [sp+Ch] [bp-A0h]@1
    char v5; // [sp+2Ch] [bp-80h]@2
    char v6; // [sp+4Ch] [bp-60h]@1
    char v7; // [sp+6Ch] [bp-40h]@1
    char v8; // [sp+8Ch] [bp-20h]@1

    byte2Bits(a1, (int)&v7, 8);
    sub_401270(&v7, (int)&v7);
    qmemcpy(&v6, &v7, 0x20u);
    v2 = &unk_40B930;
    qmemcpy(&v4, &v8, 0x20u);
do
{
    sub_401430((int)&v4, &v5, (int)v2);
    sub_401400(&v5, (int)&v6, 32);
    v2 = (char *)v2 + 48;
    qmemcpy(&v6, &v4, 0x20u);
    qmemcpy(&v4, &v5, 0x20u);
}
while ( (signed int)v2 < (signed int)&unk_40BC00 );
sub_401430((int)&v4, &v5, (int)&unk_40BC00);
sub_401400(&v6, (int)&v5, 32);
qmemcpy(&v7, &v6, 0x20u);
qmemcpy(&v8, &v4, 0x20u);
sub_4012B0(&v7, &v7);
return sub_401230(&v7, a2, 8);
}
```



# 对称加密算法 DES

- ❑ 在将数据分成左右两个分组后应该就要进入**16**轮循环迭代了，观察到该函数有一个**do...while**循环，循环的起点是**0x40B930**，增加的步长为**48**，循环的终点为**0x40BC00**，经计算刚好是进行了**15**轮循环，而缺少的一轮加密则出现在了**do...while**结束之后的地方；
- ❑ **16**轮循环迭代，这也是**DES**加密的一个特征



# 对称加密算法 DES

```
int __cdecl sub_4010B0(int a1, int a2)
{
    void *v2; // ebx@1
    char v4; // [sp+Ch] [bp-A0h]@1
    char v5; // [sp+2Ch] [bp-80h]@2
    char v6; // [sp+4Ch] [bp-60h]@1
    char v7; // [sp+6Ch] [bp-40h]@1
    char v8; // [sp+8Ch] [bp-20h]@1

    byte2Bits(a1, (int)&v7, 8);
    sub_401270(&v7, (int)&v7);
    qmemcpy(&v6, &v7, 0x20u);
    v2 = &unk_40B930;
    qmemcpy(&v4, &v8, 0x20u);
    do
    {
        sub_401430((int)&v4, &v5, (int)v2);
        sub_401400(&v5, (int)&v6, 32);
        v2 = (char *)v2 + 48;
        qmemcpy(&v6, &v4, 0x20u);
        qmemcpy(&v4, &v5, 0x20u);
    }
    while ( (signed int)v2 < (signed int)&unk_40BC00 );
    sub_401430((int)&v4, &v5, (int)&unk_40BC00);
    sub_401400(&v6, (int)&v5, 32);
    qmemcpy(&v7, &v6, 0x20u);
    qmemcpy(&v8, &v4, 0x20u);
    sub_4012B0(&v7, &v7);
    return sub_401230(&v7, a2, 8);
}
```



# 对称加密算法 DES

- 下面分析最关键的函数 `sub_401430`，对于其中的函数 `sub_4012F0`，通过观察其参数，我们可以进行猜测，`a1` 为一个 32 字节的数据，`v5` 则是分配的栈上的空间，大小为 `0x30`（48）字节，因此该函数可能是一个扩展置换，可以跟进该函数进行验证

```
int __cdecl sub_401430(int a1, char *a2, int a3)
{
    int result; // eax@1
    char v4; // [sp+8h] [bp-50h]@1
    char v5; // [sp+28h] [bp-30h]@1

    sub_4012F0((const void *)a1, (int)&v5);
    sub_401400(&v5, a3, 48);
    sub_401330((int)&v5, (int)&v4);
    result = sub_4013C0(v4, (int)&v4);
    memcpy(a2, &v4, 0x20u);
    return result;
}
```





# 对称加密算法 DES

- ❑ 分析函数sub\_401400，发现其主要做的是：  
将第一个参数和第二个参数逐字节的相加，并将结果‘与’上0x1（也即取最低位），这其实就是异或的操作，也是标准**DES**加密过程中右边分组经扩展置换后与密钥进行异或的部分



# 对称加密算法 DES

- ❑ 根据**DES**加密的步骤不难知道，函数 **sub\_401330** 是**S**盒代换的部分，主要是根据 **6bit** 数据查表得到一个 **4bit** 数据的过程。
- ❑ 对于函数 **sub\_4013C0**，跟进去后发现仍然有一个查表的操作，也即**P**盒置换的过程。
- ❑ 后续的处理则是**DES**最后的逆**IP**置换，以及从字节中提取 **bit** 位的逆操作（这里为了查表方便将字节中的每一位都提取出来放入一个字节中，构成一个字节数组）。



# 对称加密算法 DES

- ❑ **DES**加密的主要过程就分析完了，在加密完成之后将得到的密文与**0x409030**处的字节数组（密文）进行比较。
- ❑ 我们现在知道了加密算法为**DES**，密钥为地址**0x409070**处的8字节数据“**DE3\_En1C**”，在编写解密函数时，只需要将生成的**16**轮子密钥倒过来使用即。
- ❑ 因解密代码较长，仅列出关键部分



# 对称加密算法 DES

```
int main()
{
    unsignedchar key[9]="DE3_En1C";
    unsignedchar plaintext[20];
    unsignedchar
ciphertext[8]={0xef,0x34,0xd4,0xa3,0xc6,0x84,0xe4,0x23};
    get_subkey(key);
    decryption(ciphertext,plaintext);
    plaintext[8]='\0';
    printf("%s\n",plaintext);
    system("pause");
return0;
}
```



# 对称加密算法 DES

```
void decryption(unsignedchar* ciphertext,unsignedchar* plaintext){
int i;
unsignedchar array_ciphertext[64];
unsignedchar f_result[32];
unsignedchar left_array[32];
unsignedchar right_array[32];
    byte2Bit(ciphertext,array_ciphertext,8);
    ip_replace(array_ciphertext,array_ciphertext);
    memcpy(left_array,array_ciphertext,32);
    memcpy(right_array,array_ciphertext+32,32);
    for(i=15;i>0;--i){
        f_func(right_array,f_result,&subkey[i][0]);
        byteXOR(f_result,left_array,32);
        memcpy(left_array,right_array,32);
        memcpy(right_array,f_result,32);
    }
    f_func(right_array,f_result,&subkey[i][0]);
    byteXOR(left_array,f_result,32);
    memcpy(array_ciphertext,left_array,32);
    memcpy(array_ciphertext+32,right_array,32);
    fp_replace(array_ciphertext,array_ciphertext);
    bit2Byte(array_ciphertext,plaintext,8);
}
```



# 对称加密算法 DES

- ❑ 运行该解密函数，得到解密结果并对其进行验证

```
HardD3s?  
请按任意键继续. . .
```

```
give me a string to encrypt:  
HardD3s?  
G00d Job!!  
请按任意键继续. . .
```



# 第五章 常见加密算法逆向分析

⑤ 5.1 简单加密算法逆向分析

⑤ 5.2 对称加密算法逆向分析

⑤ 5.3 单向散列算法逆向分析



## 5.3 单向散列算法逆向分析

- ① 1. MD5算法
- ② 2. SHA 算法





# 1. MD5算法

- ❑ (1) 算法原理
- ❑ (2) 逆向分析



# MD系列哈希函数

- **Ron Rivest**设计的系列哈希函数系列:
  - **MD5** 是MD4的改进型 [RFC1321]
  - **MD4** [RFC1320]
  - **MD2** [RFC1319], 已被Rogier等于1995 年攻破
- 较早被标准化组织**IETF**接纳, 并已获得广泛应用
- **Hash**值长度为**128bits**



# MD5 算法逻辑

- ④ 输入：任意长度的消息
- ④ 输出：**128**位消息摘要
- ④ 处理：以**512**位输入数据块为单位

MD5 (RFC 1321) developed by Ron Rivest at MIT

北邮网安学院 崔宝江



$$L \times 512 \text{ bits} = N \times 32 \text{ bits}$$

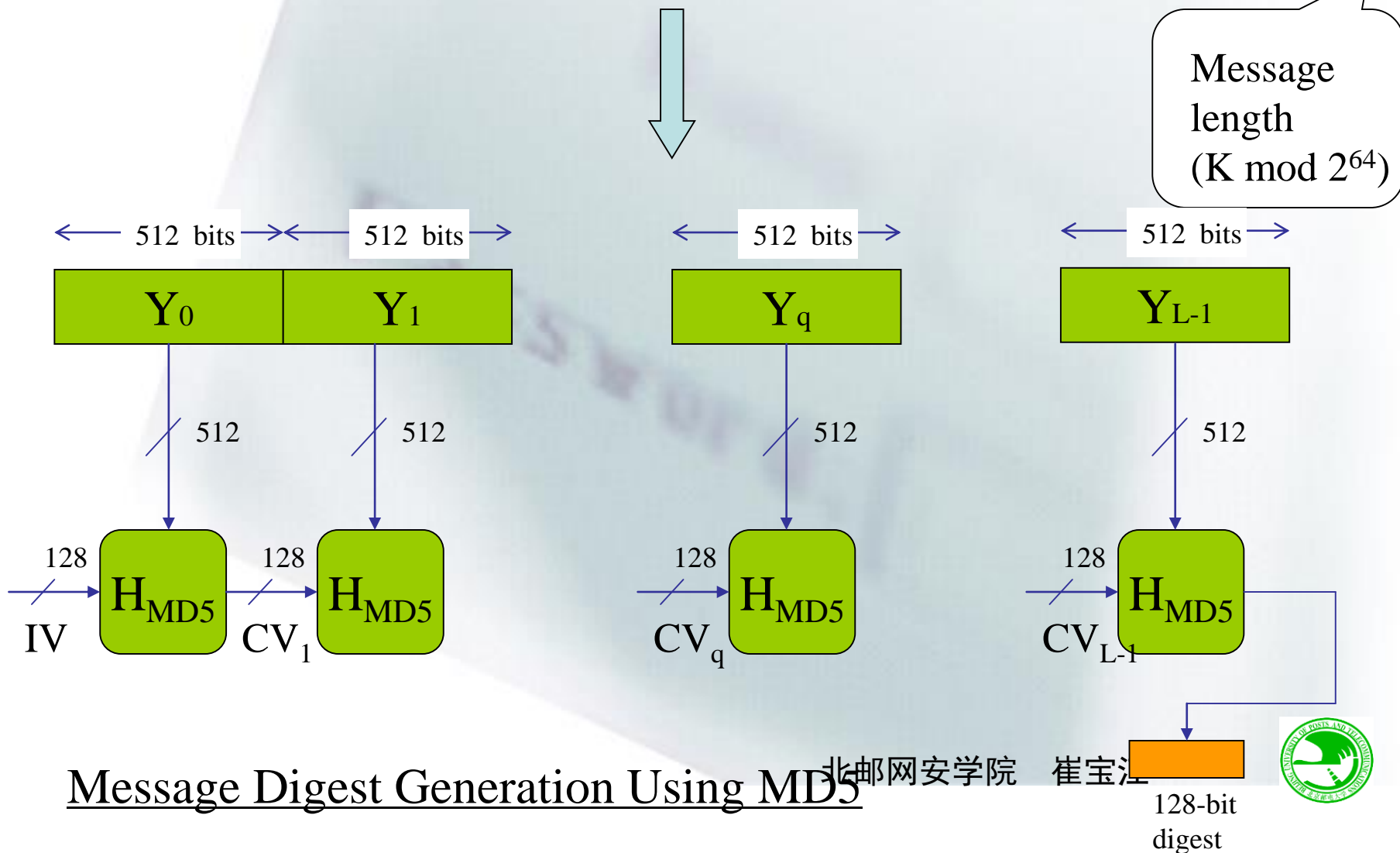
K bits

Padding  
(1 to 512  
bits)

**Message**

100...0

Message  
length  
( $K \bmod 2^{64}$ )



Message Digest Generation Using MD5

北邮网安学院

崔宝江



# MD5 算法逻辑

## MD5 Logic

步骤1：分组和填充：把明文消息按512位分组，最后填充一定长度的1000...使得每个消息的长度满足 $\text{length} \equiv 448 \pmod{512}$ 。填充的方法是先将比特“1”添加到消息的末尾，再添加k个零。

步骤2：附加消息：最后加上64位的消息摘要长度字段，整个明文恰好为512的整数倍。

步骤3：初始化MD缓冲区。一个128位MD缓冲区用以保存中间和最终散列函数的结果。置4个32比特长的缓冲区ABCD分别为

A: 01 23 45 67

B: 89 AB CD EF

C: FE DC BA 98

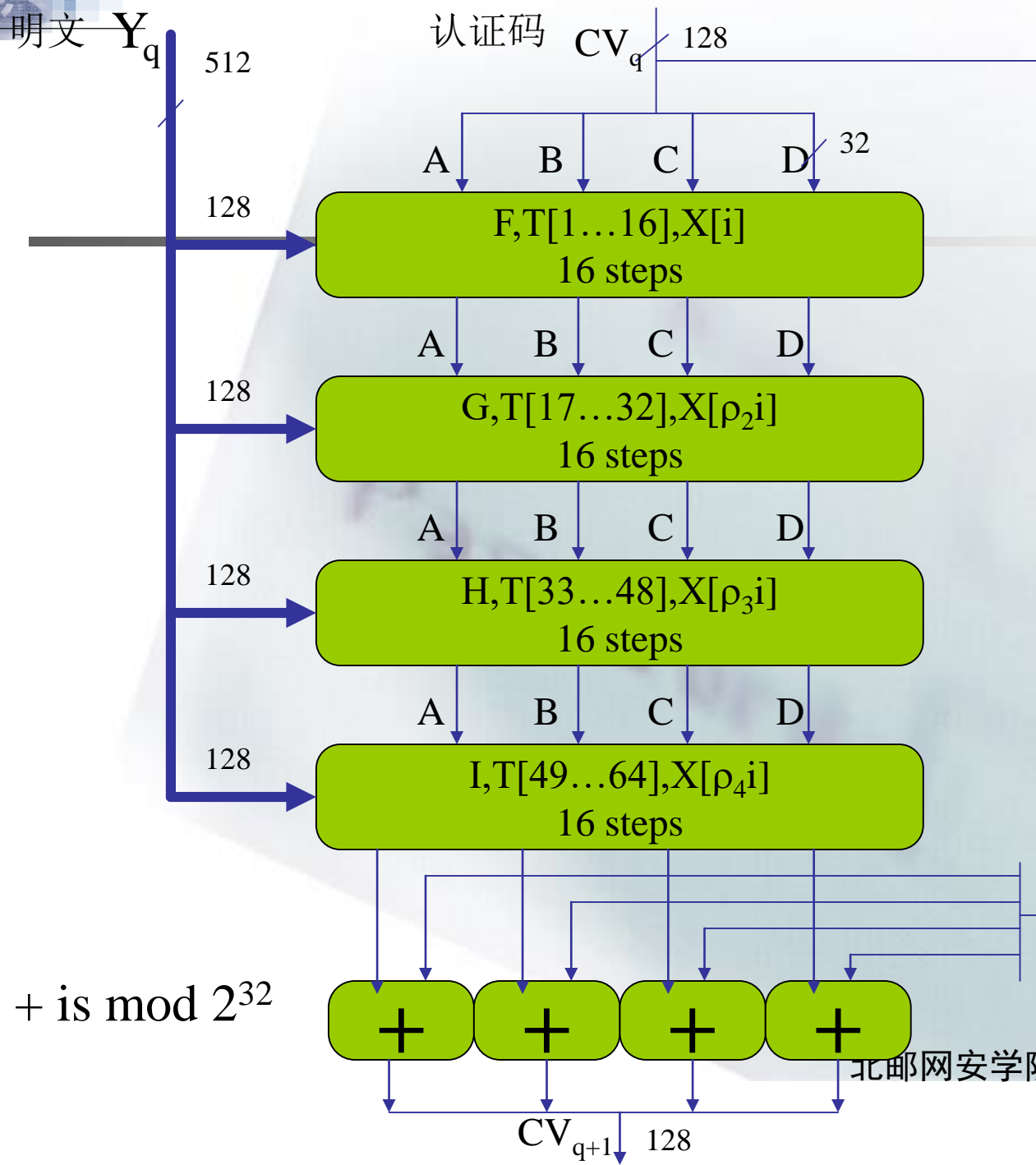
D: 76 54 32 10

步骤4：处理消息块（512位 = 16个32位字）。一个压缩函数是本算法的核心( $H_{MD5}$ )。它包括4轮处理。四轮处理具有相似的结构，但每次使用不同的基本逻辑函数，记为F,G,H,I。



明文  $Y_q$

认证码  $CV_q$



每一轮以当前的512位数据块( $Y_q$ )和128位缓冲值ABCD作为输入，并修改缓冲值的内容。每次使用64元素表 $T[1 \dots 64]$ 中的四分之一， $T[]$ 由正弦函数 $\sin$ 构造而成。 $T$ 的第 $i$ 个元素表示为 $T[i]$ ，其值等于 $2^{32} \times \text{abs}(\sin(i))$ ，其中 $i$ 是弧度。由于 $\text{abs}(\sin(i))$ 是一个0到1之间的数， $T$ 的每一个元素是一个可以表示成32位的整数。 $T$ 表提供了随机化的32位模板，消除了输入数据中的任何规律性的特征。



# MD5 算法逻辑

步骤5：输出结果。所有L个512位数据块处理完毕后，最后的结果就是128位消息摘要。

$$CV_0 = IV$$

$$CV_{q+1} = \text{SUM}_{32}(CV_q, RF_I[Y_q, RF_H[Y_q, RF_G[Y_q, RF_F[Y_q, CV_q]]]])$$

$$MD = CV_L$$

其中：IV = ABCD的初始值（见步骤3）

$Y_q$  = 消息的第q个512位数据块

L = 消息中数据块数；

$CV_q$  = 链接变量，用于第q个数据块的处理

$RF_x$  = 使用基本逻辑函数x的一轮功能函数。

MD = 最终消息摘要结果

$\text{SUM}_{32}$  = 分别按32位字计算的模 $2^{32}$ 加法结果。



# MD5 Compression Function

每一轮包含对缓冲区ABCD的16步操作所组成的一个序列。

$$a \leftarrow b + ((a + g(b,c,d) + X[k] + T[i]) \lll s)$$

其中，

$a, b, c, d$  = 缓冲区的四个字，以一个给定的次序排列；

$g$  = 基本逻辑函数F,G,H,I之一；

$\lll s$  = 对32位字循环左移s位

$X[k]$  =  $M[q \times 16 + k]$  = 在第q个512位数据块中的第k个32位字

$T[i]$  = 表T中的第i个32位字；

$+$  = 模  $2^{32}$  的加；







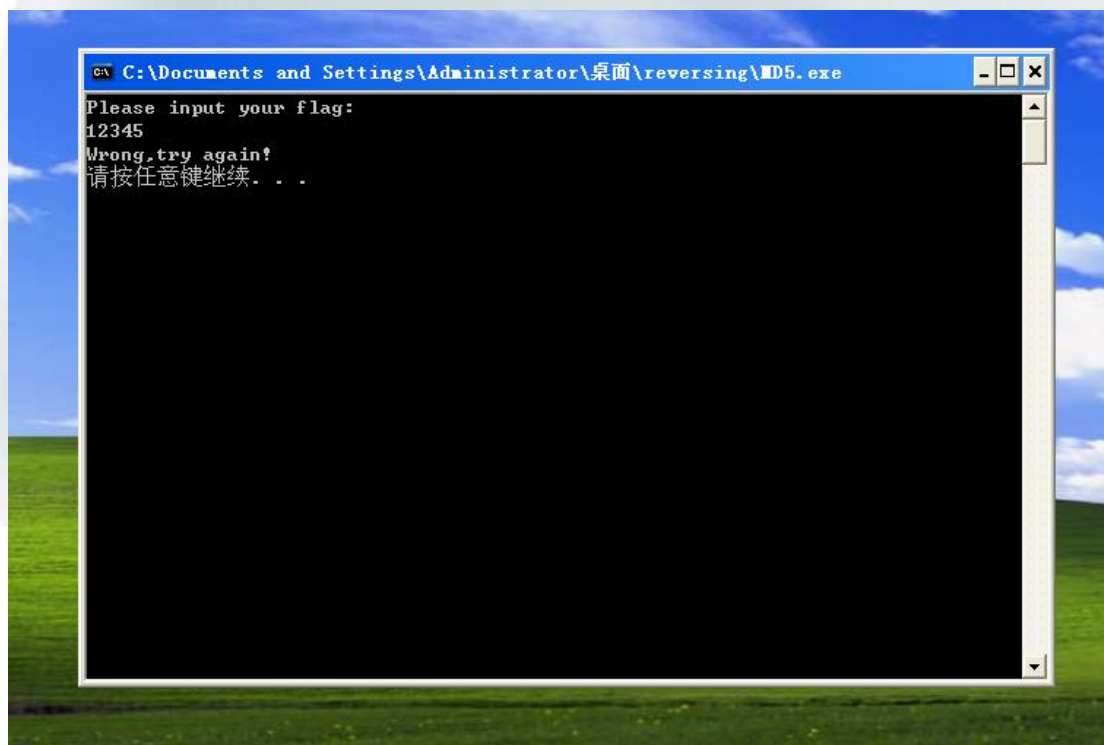
# 1. MD5算法

- ❑ (1) 算法原理
- ❑ (2) 逆向分析



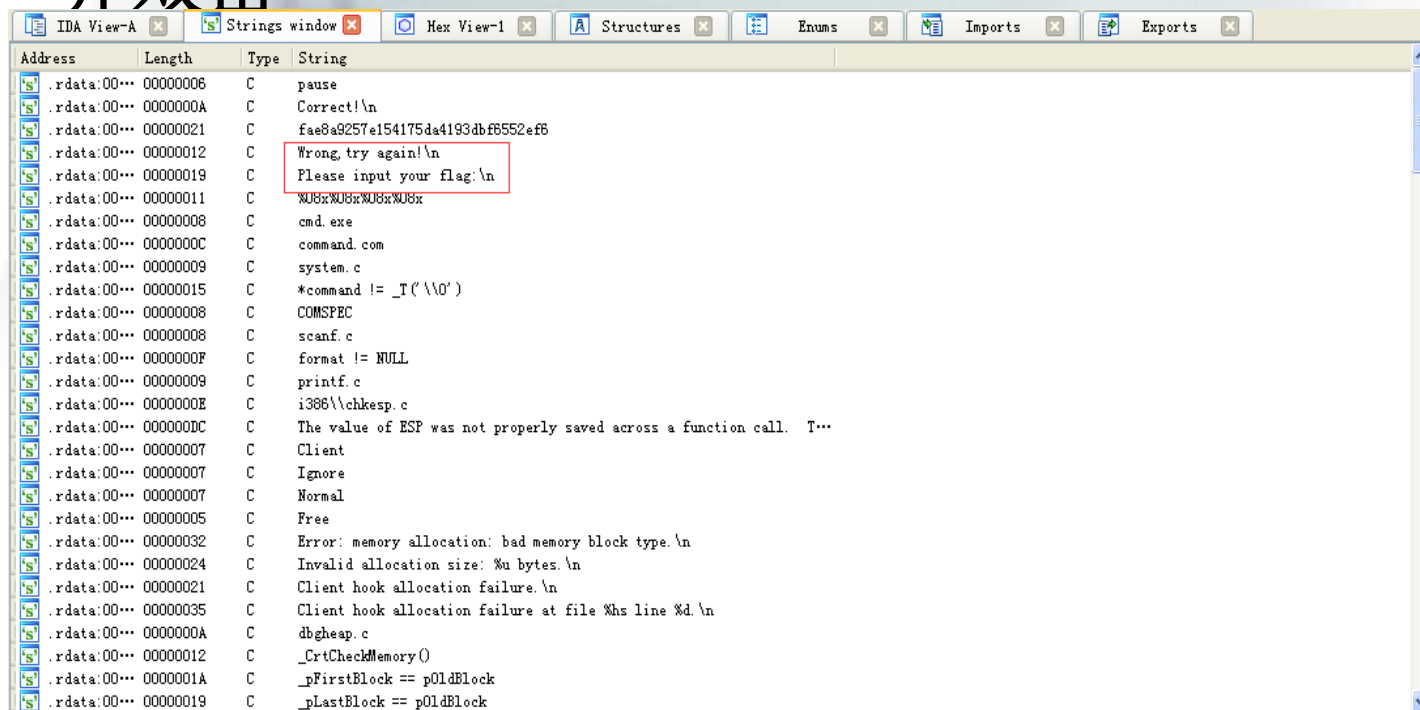
# 1. MD5算法

□ 运行示例程序MD5.exe，了解程序基本流程



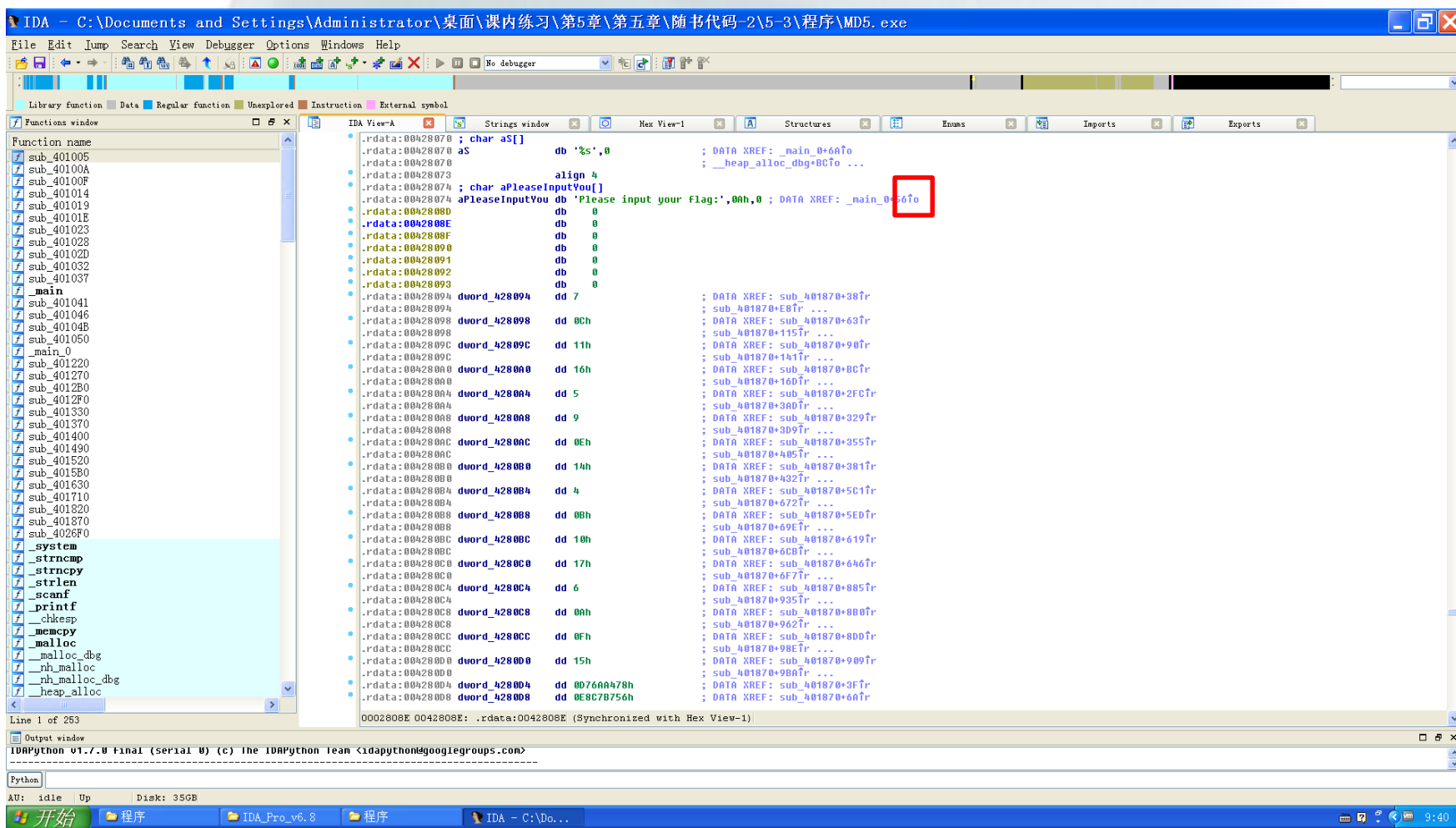
# 1. MD5算法

- ❑ 使用IDA打开MD5.exe
- ❑ 使用快捷键Shift+F12，找到Strings window并双击



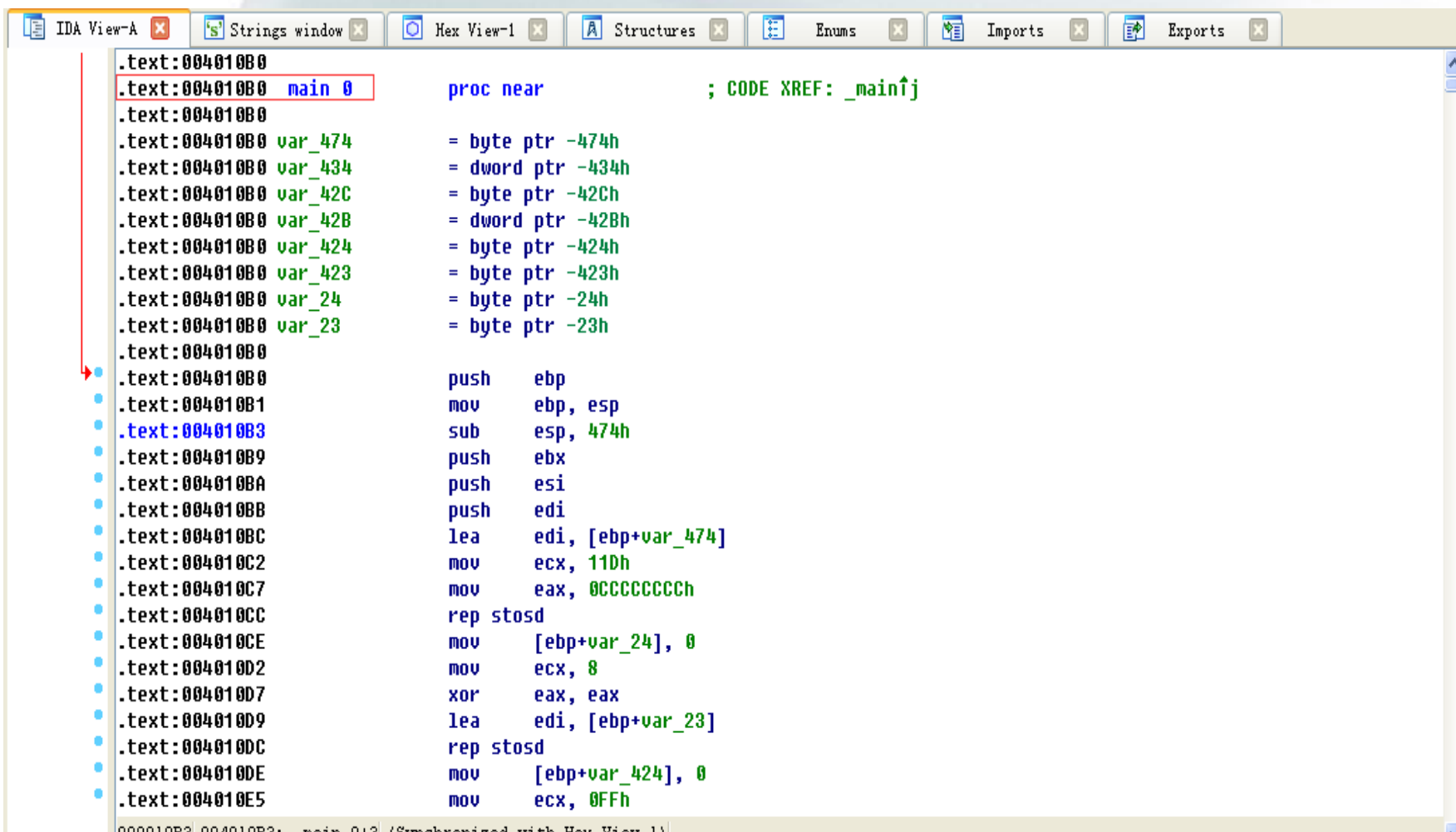
# 1. MD5算法

□ 点字符串的交叉引用，就是后面的蓝色的箭头



# 1. MD5算法

□这样就根据关键字字符串定位到main函数

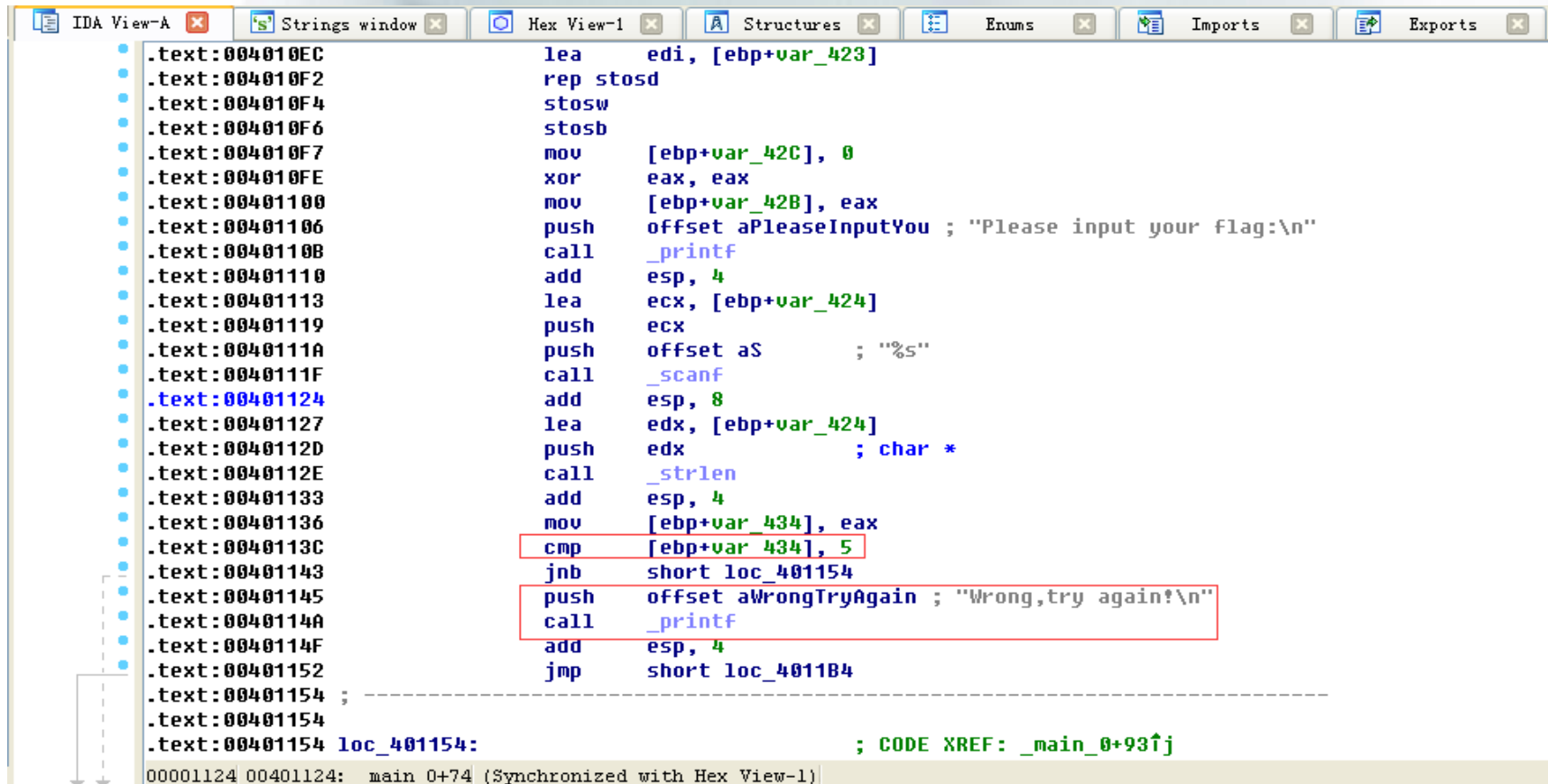


The screenshot shows the IDA Pro interface with the 'main' function selected. The 'Strings window' is open, showing the string 'main 0' at address 004010B0. The 'Hex View-1' window shows the assembly code for the function. The code starts with a 'proc near' directive and a cross-reference to '\_main'. It then defines several local variables (var\_474 to var\_23) as pointers to specific memory locations. The function body begins with a 'push ebp' instruction, followed by 'mov ebp, esp' and 'sub esp, 474h'. It then pushes several registers (ebx, esi, edi) and loads 'edi' with the address '[ebp+var\_474]'. The function continues with 'mov ecx, 110h', 'mov eax, 0CCCCCCCCh', and 'rep stosd'. It then moves '0' to '[ebp+var\_24]', sets 'ecx' to 8, XORs 'eax' with itself, loads 'edi' with '[ebp+var\_23]', and finally performs another 'rep stosd' before moving '0' to '[ebp+var\_424]' and setting 'ecx' to 0FFh.

```
.text:004010B0
.text:004010B0 main 0
.text:004010B0
.text:004010B0 var_474 = byte ptr -474h
.text:004010B0 var_434 = dword ptr -434h
.text:004010B0 var_42C = byte ptr -42Ch
.text:004010B0 var_42B = dword ptr -42Bh
.text:004010B0 var_424 = byte ptr -424h
.text:004010B0 var_423 = byte ptr -423h
.text:004010B0 var_24 = byte ptr -24h
.text:004010B0 var_23 = byte ptr -23h
.text:004010B0
.text:004010B0 push ebp
.text:004010B1 mov ebp, esp
.text:004010B3 sub esp, 474h
.text:004010B9 push ebx
.text:004010BA push esi
.text:004010BB push edi
.text:004010BC lea edi, [ebp+var_474]
.text:004010C2 mov ecx, 110h
.text:004010C7 mov eax, 0CCCCCCCCh
.text:004010CC rep stosd
.text:004010CE mov [ebp+var_24], 0
.text:004010D2 mov ecx, 8
.text:004010D7 xor eax, eax
.text:004010D9 lea edi, [ebp+var_23]
.text:004010DC rep stosd
.text:004010DE mov [ebp+var_424], 0
.text:004010E5 mov ecx, 0FFh
```

# 1. MD5算法

❑ 程序首先对输入进行了判断，若输入的长度小于5，则输出Wrong，程序退出



```
.text:004010EC      lea     edi, [ebp+var_423]
.text:004010F2      rep     stosd
.text:004010F4      stosw
.text:004010F6      stosb
.text:004010F7      mov     [ebp+var_42C], 0
.text:004010FE      xor     eax, eax
.text:00401100      mov     [ebp+var_42B], eax
.text:00401106      push    offset aPleaseInputYou ; "Please input your flag:\n"
.text:00401108      call    _printf
.text:00401110      add     esp, 4
.text:00401113      lea     ecx, [ebp+var_424]
.text:00401119      push    ecx
.text:0040111A      push    offset aS ; "%5"
.text:0040111F      call    _scanf
.text:00401124      add     esp, 8
.text:00401127      lea     edx, [ebp+var_424]
.text:0040112D      push    edx ; char *
.text:0040112E      call    _strlen
.text:00401133      add     esp, 4
.text:00401136      mov     [ebp+var_434], eax
.text:0040113C      cmp     [ebp+var_434], 5
.text:00401143      jnb     short loc_401154
.text:00401145      push    offset aWrongTryAgain ; "Wrong, try again!\n"
.text:0040114A      call    _printf
.text:0040114F      add     esp, 4
.text:00401152      jmp     short loc_4011B4
.text:00401154 ; -----
.text:00401154
.text:00401154 loc_401154: ; CODE XREF: _main_0+93↑j
00001124 00401124: _main_0+74 (Synchronized with Hex View-1)
```

# 1. MD5算法

❑ 若输入的长度大于或等于5，程序跳转到0x00401154处继续执行

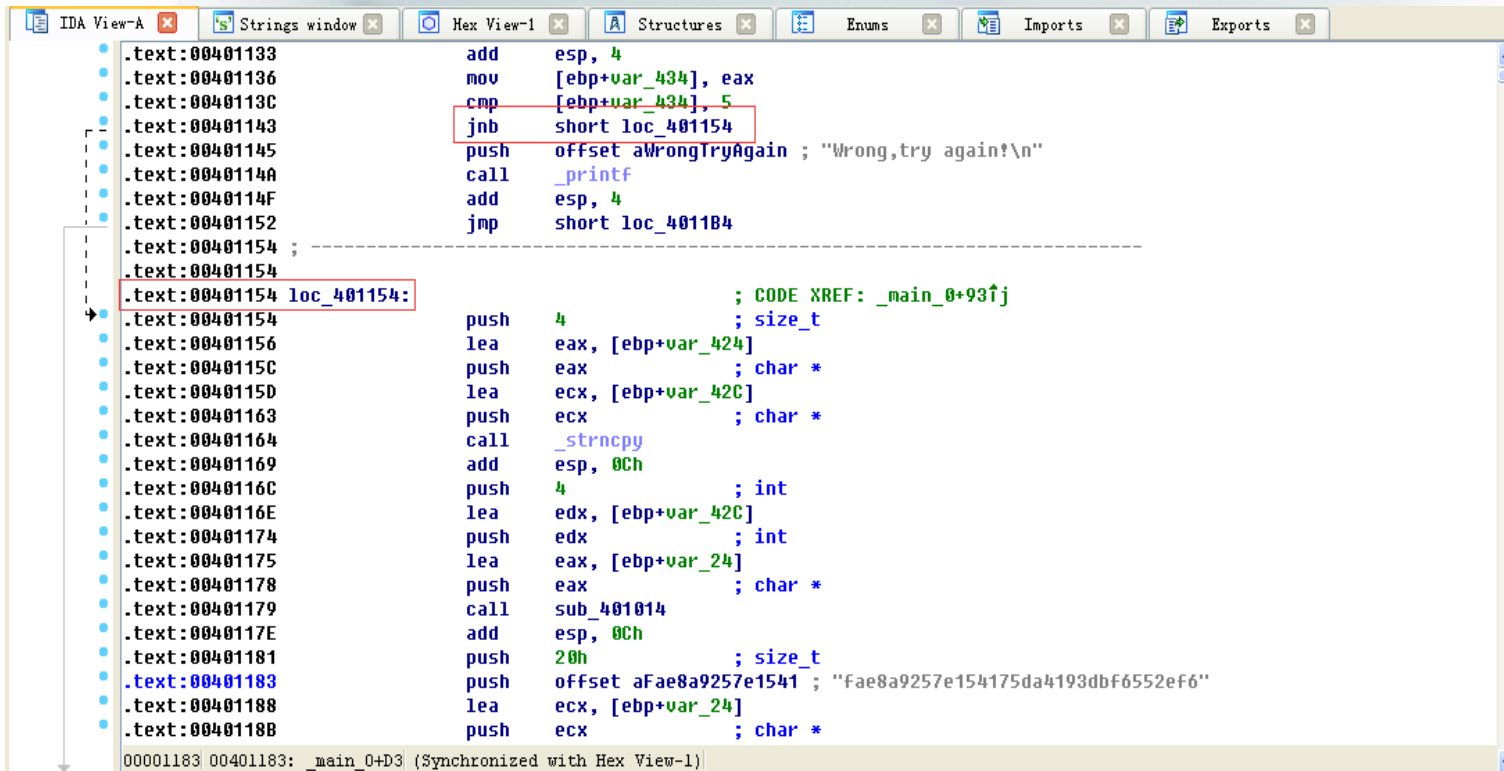
```
.text:00401133      add     esp, 4
.text:00401136      mov     [ebp+var_434], eax
.text:0040113C      cmp     [ebp+var_434], 5
.text:00401143      jnb     short loc_401154
.text:00401145      push    offset aWrongTryAgain ; "Wrong, try again!\n"
.text:0040114A      call    _printf
.text:0040114F      add     esp, 4
.text:00401152      jmp     short loc_4011B4
.text:00401154      ; -----
.text:00401154      loc_401154:
.text:00401154      push    4 ; CODE XREF: _main_0+93↑j ; size_t
.text:00401156      lea     eax, [ebp+var_424]
.text:0040115C      push    eax ; char *
.text:0040115D      lea     ecx, [ebp+var_42C]
.text:00401163      push    ecx ; char *
.text:00401164      call    _strncpy
.text:00401169      add     esp, 0Ch
.text:0040116C      push    4 ; int
.text:0040116E      lea     edx, [ebp+var_42C]
.text:00401174      push    edx ; int
.text:00401175      lea     eax, [ebp+var_24]
.text:00401178      push    eax ; char *
.text:00401179      call    sub_401014
.text:0040117E      add     esp, 0Ch
.text:00401181      push    20h ; size_t
.text:00401183      push    offset aFae8a9257e1541 ; "Fae8a9257e154175da4193dbf6552ef6"
.text:00401188      lea     ecx, [ebp+var_24]
.text:0040118B      push    ecx ; char *
```

00001183 00401183: \_main\_0+D3 (Synchronized with Hex View-1)



# 1. MD5算法

❑ 若输入的长度大于或等于5，程序跳转到0x00401154处继续执行

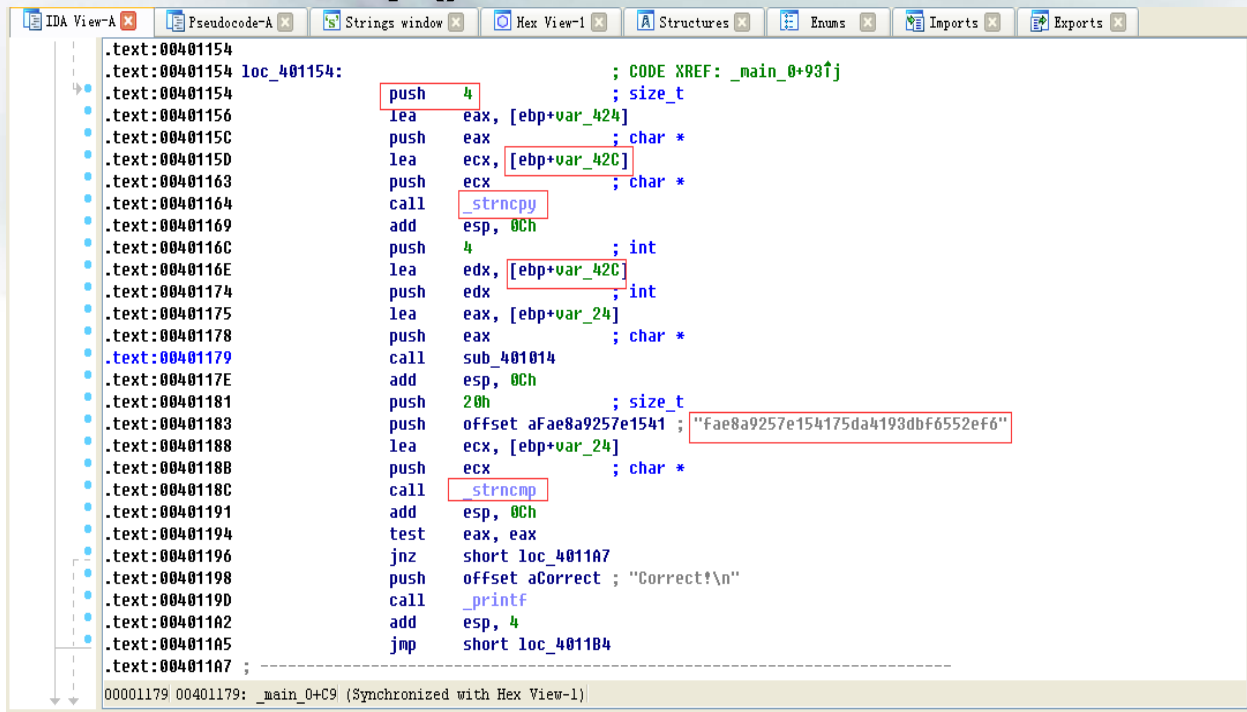


```
.text:00401133      add     esp, 4
.text:00401136      mov     [ebp+var_434], eax
.text:0040113C      cmp     [ebp+var_434], 5
.text:00401143      jnb     short loc_401154
.text:00401145      push    offset aWrongTryAgain ; "Wrong,try again!\n"
.text:0040114A      call    _printf
.text:0040114F      add     esp, 4
.text:00401152      jmp     short loc_401184
.text:00401154      ; -----
.text:00401154      loc_401154:
.text:00401154      push    4 ; CODE XREF: _main_0+93↑j ; size_t
.text:00401156      lea     eax, [ebp+var_424]
.text:0040115C      push    eax ; char *
.text:0040115D      lea     ecx, [ebp+var_42C]
.text:00401163      push    ecx ; char *
.text:00401164      call    _strncpy
.text:00401169      add     esp, 0Ch
.text:0040116C      push    4 ; int
.text:0040116E      lea     edx, [ebp+var_42C]
.text:00401174      push    edx ; int
.text:00401175      lea     eax, [ebp+var_24]
.text:00401178      push    eax ; char *
.text:00401179      call    sub_401014
.text:0040117E      add     esp, 0Ch
.text:00401181      push    20h ; size_t
.text:00401183      push    offset aFae8a9257e154175da4193dbf6552ef6 ; "Fae8a9257e154175da4193dbf6552ef6"
.text:00401188      lea     ecx, [ebp+var_24]
.text:0040118B      push    ecx ; char *
```



# 1. MD5算法

- 程序首先调用**strncpy**取出输入的前4字节并作为函数**sub\_401014()**的第二个参数，最后将**sub\_401014()**的返回结果与字符串“**fae8a9257e154175da4193dbf6552ef6**”进行比较，根据**strncmp()**的比较结果输出 **Correct**或**Wrong**。

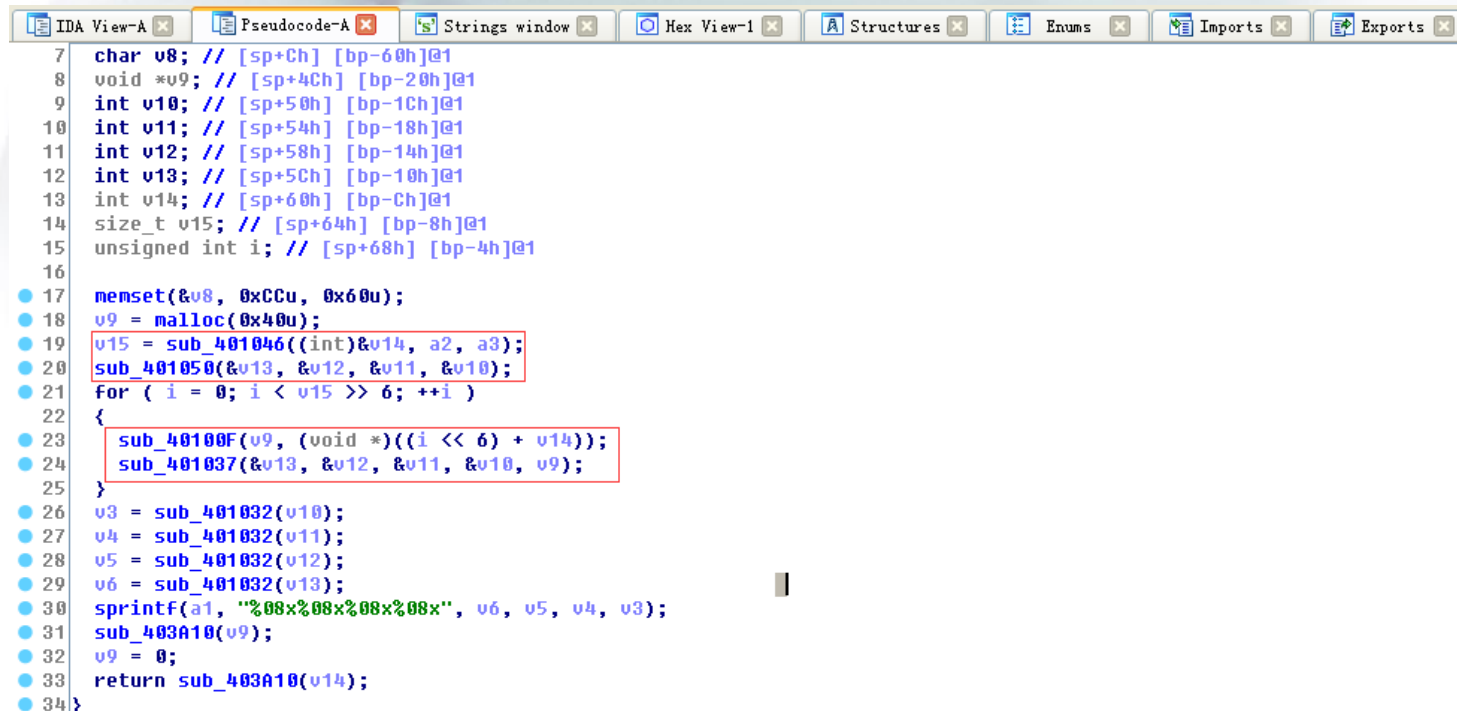


```
.text:00401154
.text:00401154 loc_401154:                ; CODE XREF: _main_0+93↑j
.text:00401154                push     4                ; size_t
.text:00401156                lea     eax, [ebp+var_424]
.text:0040115C                push     eax              ; char *
.text:0040115D                lea     ecx, [ebp+var_42C]
.text:00401163                push     ecx              ; char *
.text:00401164                call    _strncpy
.text:00401169                add     esp, 0Ch
.text:0040116C                push     4                ; int
.text:0040116E                lea     edx, [ebp+var_42C]
.text:00401174                push     edx              ; int
.text:00401175                lea     eax, [ebp+var_24]
.text:00401178                push     eax              ; char *
.text:00401179                call    sub_401014
.text:0040117E                add     esp, 0Ch
.text:00401181                push     20h              ; size_t
.text:00401183                push     offset aFae8a9257e1541 ; "fae8a9257e154175da4193dbf6552ef6"
.text:00401188                lea     ecx, [ebp+var_24]
.text:0040118B                push     ecx              ; char *
.text:0040118C                call    _strncmp
.text:00401191                add     esp, 0Ch
.text:00401194                test    eax, eax
.text:00401196                jnz     short loc_4011A7
.text:00401198                push     offset aCorrect ; "Correct!\n"
.text:0040119D                call    _printf
.text:004011A2                add     esp, 4
.text:004011A5                jmp     short loc_4011B4
.text:004011A7 ; -----
00001179 00401179: _main_0+C9 (Synchronized with Hex View-1)
```



# 1. MD5算法

- ❑ 对函数sub\_401014()进行详细分析，使用快捷键F5对sub\_401014()进行反编译
- ❑ 此函数的执行流程涉及到四个函数：sub\_401046()、sub\_401050()、sub\_40100F()、和sub\_401037()



```
7  char v8; // [sp+Ch] [bp-60h]@1
8  void *u9; // [sp+4Ch] [bp-20h]@1
9  int v10; // [sp+50h] [bp-1Ch]@1
10 int v11; // [sp+54h] [bp-18h]@1
11 int v12; // [sp+58h] [bp-14h]@1
12 int v13; // [sp+5Ch] [bp-10h]@1
13 int v14; // [sp+60h] [bp-Ch]@1
14 size_t v15; // [sp+64h] [bp-8h]@1
15 unsigned int i; // [sp+68h] [bp-4h]@1
16
17 memset(&v8, 0xCCu, 0x60u);
18 v9 = malloc(0x40u);
19 v15 = sub_401046((int)&v14, a2, a3);
20 sub_401050(&v13, &v12, &v11, &v10);
21 for ( i = 0; i < v15 >> 6; ++i )
22 {
23     sub_40100F(v9, (void *)((i << 6) + v14));
24     sub_401037(&v13, &v12, &v11, &v10, v9);
25 }
26 v3 = sub_401032(v10);
27 v4 = sub_401032(v11);
28 v5 = sub_401032(v12);
29 v6 = sub_401032(v13);
30 sprintf(a1, "%08x%08x%08x%08x", v6, v5, v4, v3);
31 sub_403A10(v9);
32 v9 = 0;
33 return sub_403A10(v14);
34 }
```



# 1. MD5算法

- ❑ 对于函数sub\_401046(), 调用了sub\_401710()

```
1 size_t __cdecl sub_401046(int a1, void *a2, size_t a3)
2 {
3     return sub_401710(a1, a2, a3);
4 }
```

- ❑ 根据sub\_401710()一些特殊的语句, 推测出这个函数的作用可能是数据填充
- ❑ 因为sub\_401710()函数中出现了56、64以及模64这些数值和运算, 联想到某些算法的数据填充规则, 数据填充后使得数据的比特长度对512取模等于448, 换算为字节运算, 即填充后的长度对64取模等于56。



# 1. MD5算法

- ❑ 联系到：填充的方法是先将被“1”添加到数据的末尾，再添加若干0。填充完毕后再添加一个64比特长的块来存储消息长度，其值等于填充前消息长度的二进制表示。

```
1 size_t __cdecl sub_401710(int a1, void *a2, size_t a3)
2 {
3     char v4; // [sp+Ch] [bp-58h]@1
4     void *v5; // [sp+4Ch] [bp-18h]@1
5     size_t v6; // [sp+50h] [bp-14h]@1
6     size_t v7; // [sp+54h] [bp-10h]@1
7     size_t v8; // [sp+58h] [bp-Ch]@1
8     int v9; // [sp+5Ch] [bp-8h]@1
9     int i; // [sp+60h] [bp-4h]@1
10
11     memset(&v4, 0xCCu, 0x58u);
12     v8 = 8 * a3;
13     v7 = 56 - a3 % 0x40;
14     v6 = a3 + 56 - a3 % 0x40 + 8;
15     v5 = calloc(v6, 1u);
16     memcpy(v5, a2, a3);
17     *((_BYTE *)v5 + a3) = -128;
18     v9 = 4;
19     for ( i = 0; i < v9; ++i )
20         memcpy((char *)v5 + v6 + i - 8, (char *)v8 + i, 1u);
21     *(_DWORD *)a1 = v5;
22     return v6;
23 }
```

v8为数据的比特长度

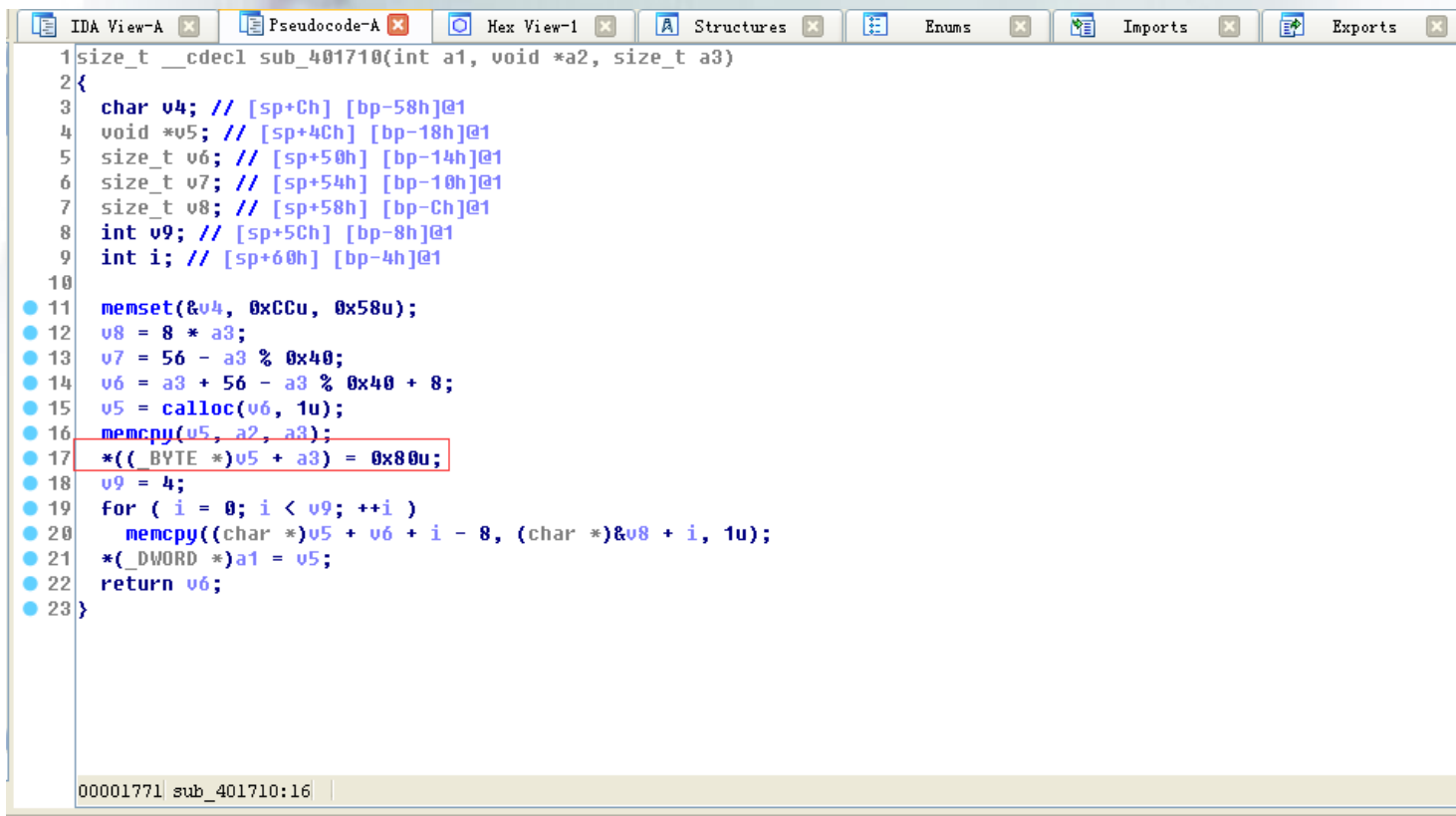
v7为需要填充的数据长度

v6为数据的总长度



# 1. MD5算法

❑ 另外0x80其实是比特串“10000000”，与数据填充规则中追加一个比特1，再填充0的填充规则相符合。



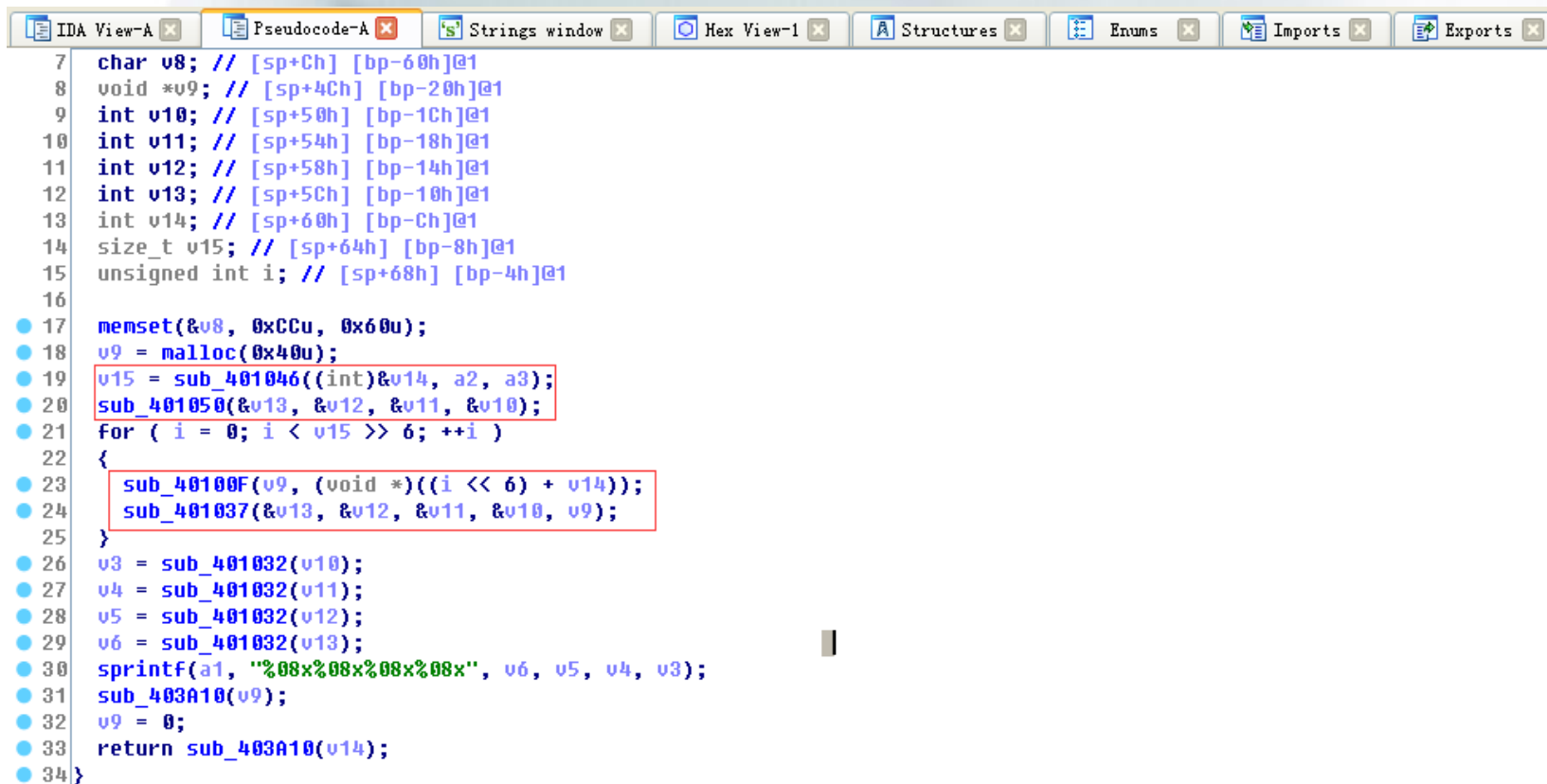
```
1 size_t __cdecl sub_401710(int a1, void *a2, size_t a3)
2 {
3     char v4; // [sp+Ch] [bp-58h]@1
4     void *v5; // [sp+4Ch] [bp-18h]@1
5     size_t v6; // [sp+50h] [bp-14h]@1
6     size_t v7; // [sp+54h] [bp-10h]@1
7     size_t v8; // [sp+58h] [bp-Ch]@1
8     int v9; // [sp+5Ch] [bp-8h]@1
9     int i; // [sp+60h] [bp-4h]@1
10
11     memset(&v4, 0xCCu, 0x58u);
12     v8 = 8 * a3;
13     v7 = 56 - a3 % 0x40;
14     v6 = a3 + 56 - a3 % 0x40 + 8;
15     v5 = calloc(v6, 1u);
16     memcpy(v5, a2, a3);
17     *((BYTE *)v5 + a3) = 0x80u;
18     v9 = 4;
19     for ( i = 0; i < v9; ++i )
20         memcpy((char *)v5 + v6 + i - 8, (char *)&v4 + i, 1u);
21     *(_DWORD *)a1 = v5;
22     return v6;
23 }
```

00001771 sub\_401710:16



# 1. MD5算法

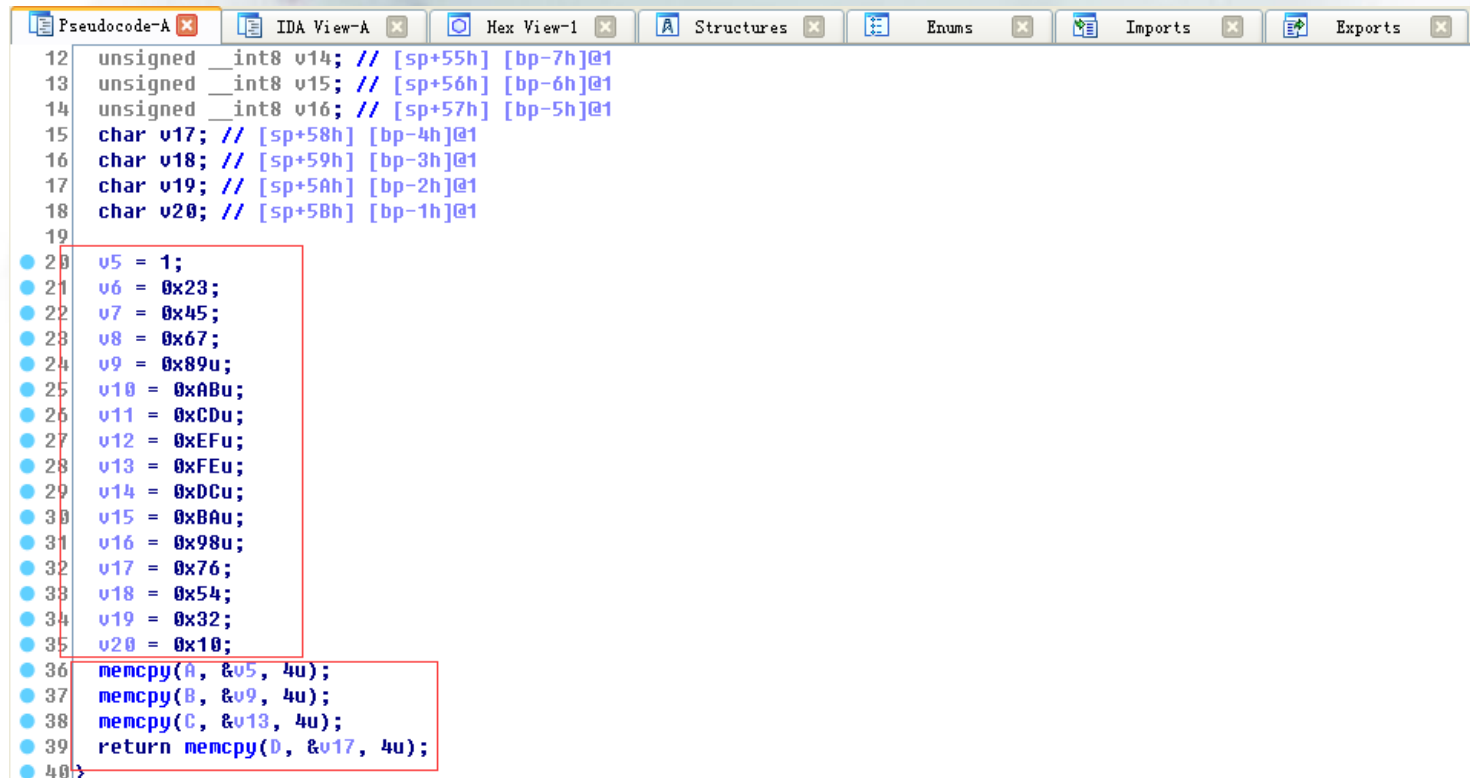
□ 结合上文的分析，sub\_401046()函数完成了数据填充



```
7 char v8; // [sp+Ch] [bp-60h]@1
8 void *v9; // [sp+4Ch] [bp-20h]@1
9 int v10; // [sp+50h] [bp-1Ch]@1
10 int v11; // [sp+54h] [bp-18h]@1
11 int v12; // [sp+58h] [bp-14h]@1
12 int v13; // [sp+5Ch] [bp-10h]@1
13 int v14; // [sp+60h] [bp-Ch]@1
14 size_t v15; // [sp+64h] [bp-8h]@1
15 unsigned int i; // [sp+68h] [bp-4h]@1
16
17 memset(&v8, 0xCCu, 0x60u);
18 v9 = malloc(0x40u);
19 v15 = sub_401046((int)&v14, a2, a3);
20 sub_401050(&v13, &v12, &v11, &v10);
21 for ( i = 0; i < v15 >> 6; ++i )
22 {
23     sub_40100F(v9, (void *)((i << 6) + v14));
24     sub_401037(&v13, &v12, &v11, &v10, v9);
25 }
26 v3 = sub_401032(v10);
27 v4 = sub_401032(v11);
28 v5 = sub_401032(v12);
29 v6 = sub_401032(v13);
30 sprintf(a1, "%08x%08x%08x%08x", v6, v5, v4, v3);
31 sub_403A10(v9);
32 v9 = 0;
33 return sub_403A10(v14);
34 }
```

# 1. MD5算法

- 接下来分析函数 `sub_401050()`，它将 `v5~v20` 这16长度为1个字节的变量赋值给四个整型变量，分别设为 `A,B,C,D`，使用快捷键 `N` 修改变量名，使用快捷键 `H` 可以将 `v5~v20` 修改为16进制显示。



```
Pseudocode-A x IDA View-A x Hex View-1 x Structures x Enums x Imports x Exports x
12 unsigned __int8 v14; // [sp+55h] [bp-7h]@1
13 unsigned __int8 v15; // [sp+56h] [bp-6h]@1
14 unsigned __int8 v16; // [sp+57h] [bp-5h]@1
15 char v17; // [sp+58h] [bp-4h]@1
16 char v18; // [sp+59h] [bp-3h]@1
17 char v19; // [sp+5Ah] [bp-2h]@1
18 char v20; // [sp+5Bh] [bp-1h]@1
19
20 v5 = 1;
21 v6 = 0x23;
22 v7 = 0x45;
23 v8 = 0x67;
24 v9 = 0x89u;
25 v10 = 0xABu;
26 v11 = 0xCDu;
27 v12 = 0xEFu;
28 v13 = 0xFEu;
29 v14 = 0xDCu;
30 v15 = 0xBAu;
31 v16 = 0x98u;
32 v17 = 0x76;
33 v18 = 0x54;
34 v19 = 0x32;
35 v20 = 0x10;
36 memcpy(A, &v5, 4u);
37 memcpy(B, &v9, 4u);
38 memcpy(C, &v13, 4u);
39 return memcpy(D, &v17, 4u);
40 }
```





# MD5 算法逻辑

## MD5 Logic

步骤1：分组和填充：把明文消息按512位分组，最后填充一定长度的1000...使得每个消息的长度满足 $\text{length} \equiv 448 \pmod{512}$ 。填充的方法是先将比特“1”添加到消息的末尾，再添加k个零。

步骤2：附加消息：最后加上64位的消息摘要长度字段，整个明文恰好为512的整数倍。

步骤3：初始化MD缓冲区。一个128位MD缓冲区用以保存中间和最终散列函数的结果。置4个32比特长的缓冲区ABCD分别为

A: 01 23 45 67

B: 89 AB CD EF

C: FE DC BA 98

D: 76 54 32 10

步骤4：处理消息块（512位 = 16个32位字）。一个压缩函数是本算法的核心( $H_{MD5}$ )。它包括4轮处理。四轮处理具有相似的结构，但每次使用不同的基本逻辑函数，记为F,G,H,I。



# 1. MD5算法

□ 经过函数sub\_401050()后，设这A~D这4个变量在内存中的存储为

Startup		Untitled1*																
▼ Edit As: Hex ▼		Run Script ▼		Run Template ▼														
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000h:		01	23	45	67	89	AB	CD	EF	FE	DC	BA	98	76	54	32	10	.#Eg%«İipÜ°~vT2.
0010h:																		

□ 由于计算机为小端存储模式，因此A~D真正的数值为

A = 0x67452301;

B = 0xefcdab89;

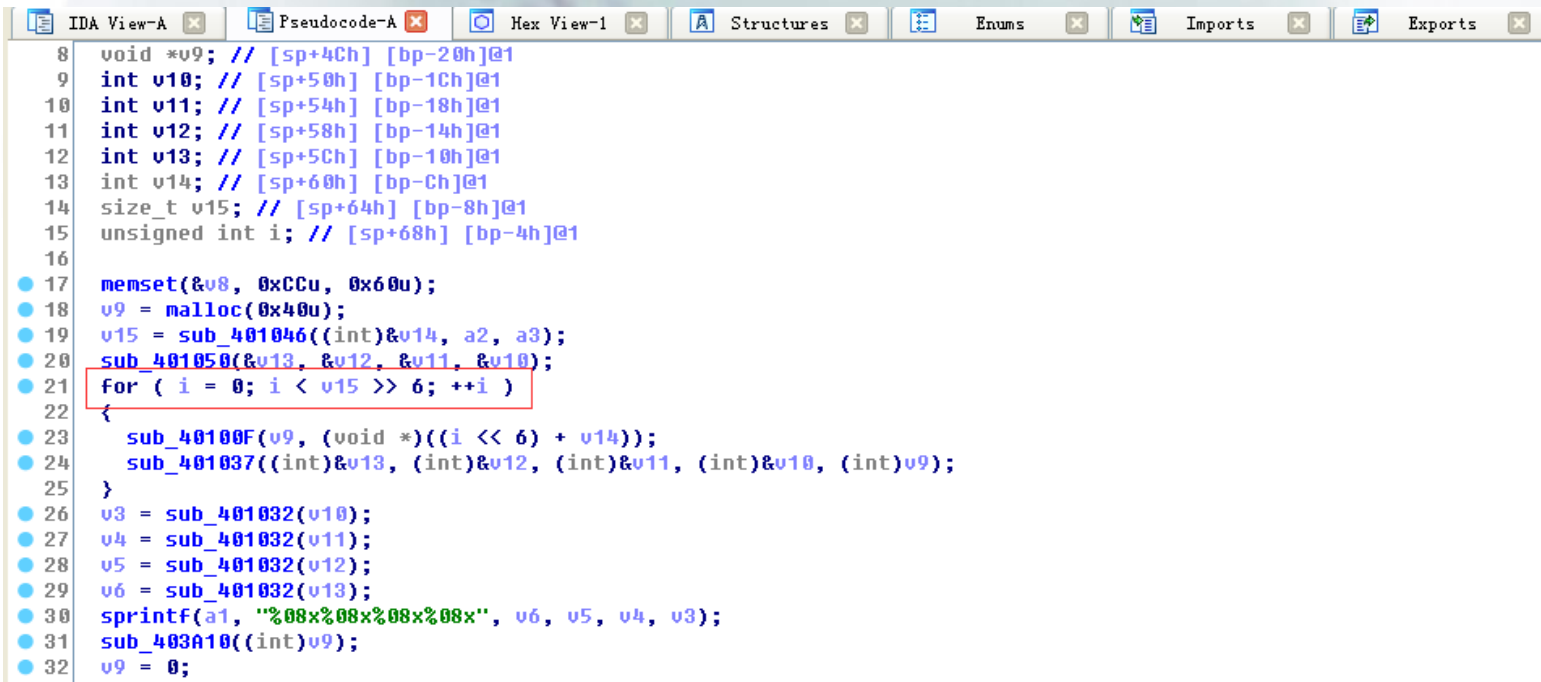
C = 0x98badcfe;

D = 0x10325476;



# 1. MD5算法

- ❑ 完成赋值操作后，函数进入for循环，循环次数为 **sub\_401046()** 的返回值右移6位(相当于除以64，就是对每个分组进行加密)，上文分析得到 **sub\_401046()** 函数的返回值为整个数据长度，这里可以看到数据的分组长度是64个字节（512bit）



```
8 void *v9; // [sp+4Ch] [bp-20h]@1
9 int v10; // [sp+50h] [bp-1Ch]@1
10 int v11; // [sp+54h] [bp-18h]@1
11 int v12; // [sp+58h] [bp-14h]@1
12 int v13; // [sp+5Ch] [bp-10h]@1
13 int v14; // [sp+60h] [bp-Ch]@1
14 size_t v15; // [sp+64h] [bp-8h]@1
15 unsigned int i; // [sp+68h] [bp-4h]@1
16
17 memset(&v8, 0xCCu, 0x60u);
18 v9 = malloc(0x40u);
19 v15 = sub_401046((int)&v14, a2, a3);
20 sub_401050(&v13, &v12, &v11, &v10);
21 for ( i = 0; i < v15 >> 6; ++i )
22 {
23     sub_40100F(v9, (void *)((i << 6) + v14));
24     sub_401037((int)&v13, (int)&v12, (int)&v11, (int)&v10, (int)v9);
25 }
26 v3 = sub_401032(v10);
27 v4 = sub_401032(v11);
28 v5 = sub_401032(v12);
29 v6 = sub_401032(v13);
30 sprintf(a1, "%08x%08x%08x%08x", v6, v5, v4, v3);
31 sub_403A10((int)v9);
32 v9 = 0;
```

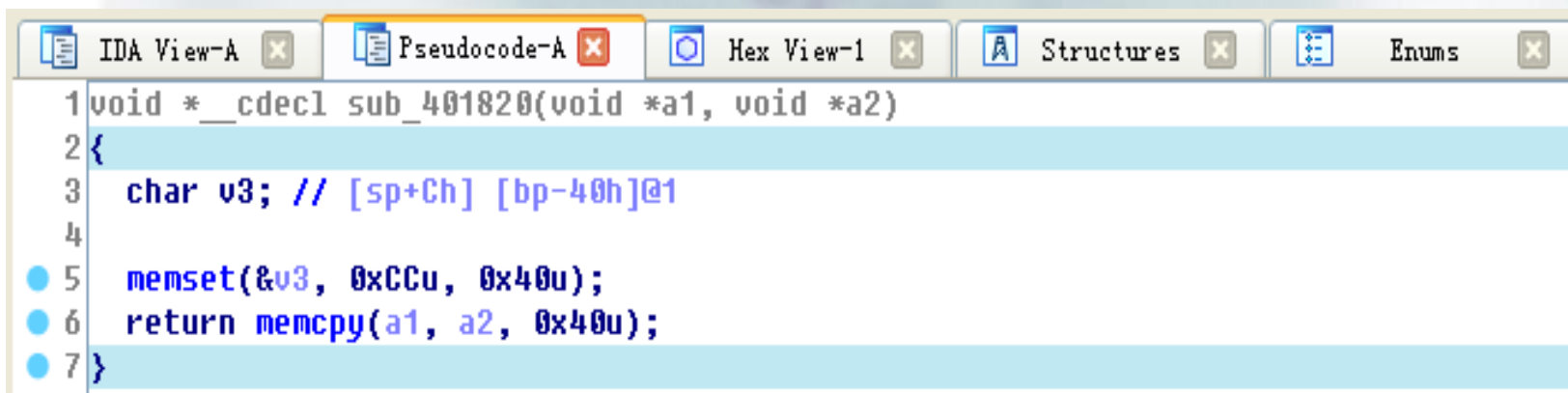


# 1. MD5算法

□ 首先分析for循环中的第一个函数 **sub\_40100F()**。该函数比较简单，调用了 **memcpy()** 函数，将input复制到变量v9中

```
23 | sub_40100F(v9, (void *)((i << 6) + v14));
```

```
1 void *__cdecl sub_40100F(void *a1, void *a2)
2 {
3   return sub_401820(a1, a2);
4 }
```



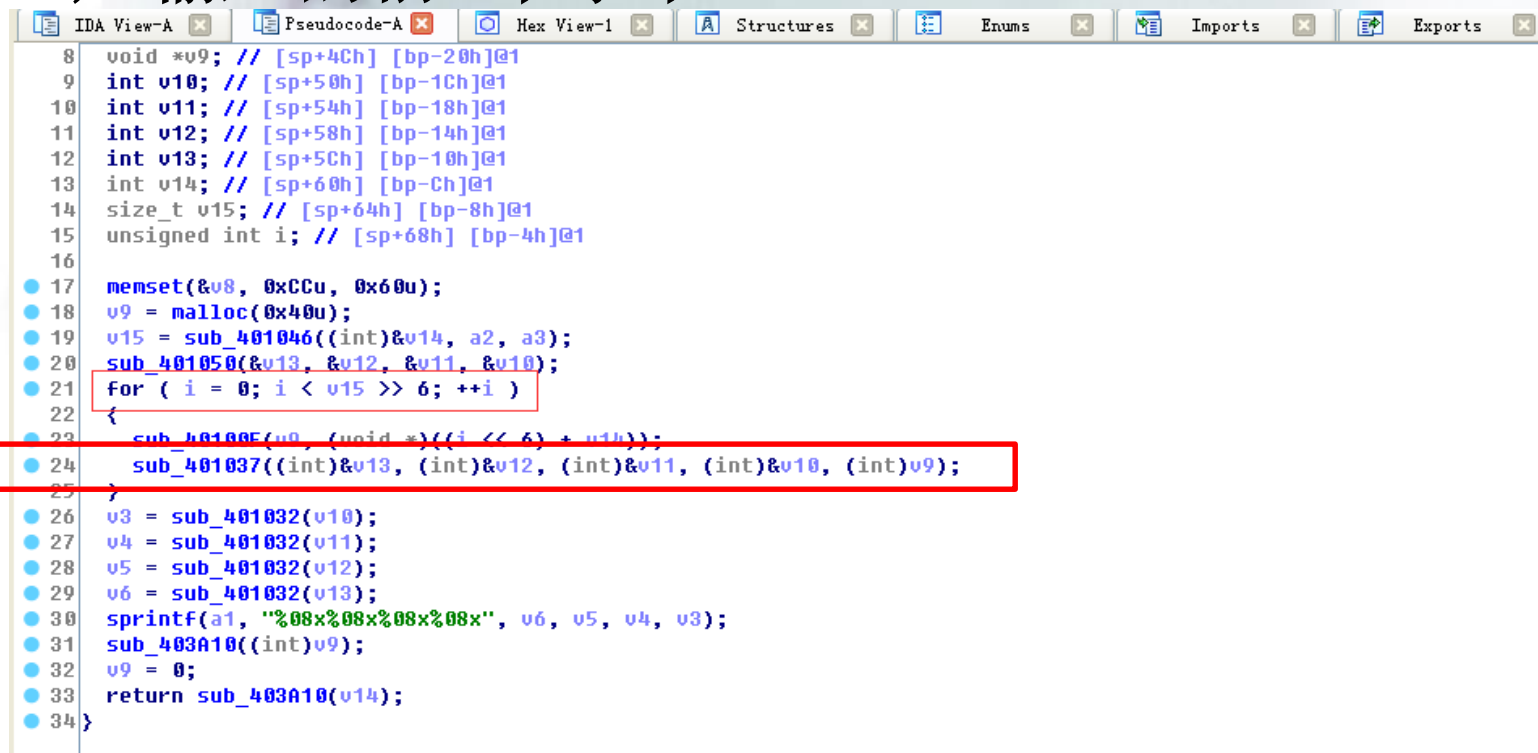
The screenshot shows the IDA Pro interface with the 'Pseudocode-A' window active. The function sub\_401820 is displayed with the following code:

```
1 void *__cdecl sub_401820(void *a1, void *a2)
2 {
3   char v3; // [sp+Ch] [bp-40h]@1
4
5   memset(&v3, 0xCCu, 0x40u);
6   return memcpy(a1, a2, 0x40u);
7 }
```



# 1. MD5算法

- ❑ 重点分析一下sub\_401037()这个函数
- ❑ 该函数的参数为前面被赋值的4个变量以及用户输入的前4个字节



```
8 void *v9; // [sp+4Ch] [bp-20h]@1
9 int v10; // [sp+50h] [bp-1Ch]@1
10 int v11; // [sp+54h] [bp-18h]@1
11 int v12; // [sp+58h] [bp-14h]@1
12 int v13; // [sp+5Ch] [bp-10h]@1
13 int v14; // [sp+60h] [bp-Ch]@1
14 size_t v15; // [sp+64h] [bp-8h]@1
15 unsigned int i; // [sp+68h] [bp-4h]@1
16
17 memset(&v8, 0xCCu, 0x60u);
18 v9 = malloc(0x40u);
19 v15 = sub_401046((int)&v14, a2, a3);
20 sub_401050(&v13, &v12, &v11, &v10);
21 for ( i = 0; i < v15 >> 6; ++i )
22 {
23     sub_40100E(v8, (void *)((i << 6) + v14));
24     sub_401037((int)&v13, (int)&v12, (int)&v11, (int)&v10, (int)v9);
25 }
26 v3 = sub_401032(v10);
27 v4 = sub_401032(v11);
28 v5 = sub_401032(v12);
29 v6 = sub_401032(v13);
30 sprintf(a1, "%08x%08x%08x%08x", v6, v5, v4, v3);
31 sub_403A10((int)v9);
32 v9 = 0;
33 return sub_403A10(v14);
34 }
```



# 1. MD5算法

- ❑ 根据反编译结果可以看到，该函数对变量A~D进行了**64**轮的轮函数
  - 1~16轮调用了函数sub\_401028()
  - 17~32轮调用了函数sub\_401005()
  - 33~48轮调用了函数sub\_40104B()
  - 49~64轮调用了函数sub\_40101E()。
- ❑ 经过**64**轮的轮函数之后，变量A~D分别与初始值相加，函数返回。

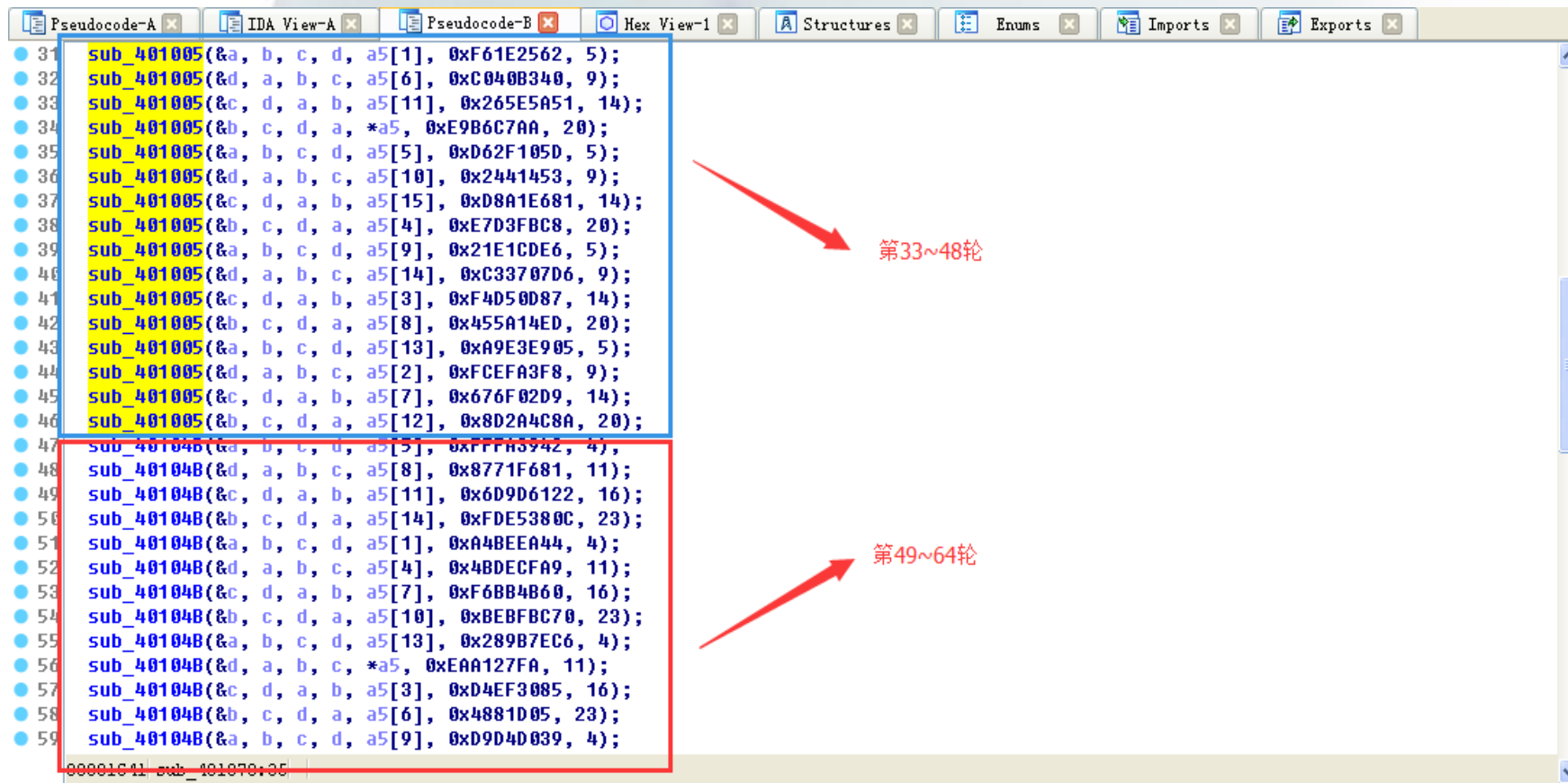


# 1. MD5算法

```
Pseudocode-A x Pseudocode-B x Hex View-1 x Structures x Enums x Imports x Exports x
16 sub_401028((int)&d, a, b, c, a5[1], 0xE8C7B756, 12);
17 sub_401028((int)&c, d, a, b, a5[2], 0x242070DB, 17);
18 sub_401028((int)&b, c, d, a, a5[3], 0xC1BDCEEE, 22);
19 sub_401028((int)&a, b, c, d, a5[4], 0xF57C0FAF, 7);
20 sub_401028((int)&d, a, b, c, a5[5], 0x4787C62A, 12);
21 sub_401028((int)&c, d, a, b, a5[6], 0xA8304613, 17);
22 sub_401028((int)&b, c, d, a, a5[7], 0xFD469501, 22);
23 sub_401028((int)&a, b, c, d, a5[8], 0x69809808, 7);
24 sub_401028((int)&d, a, b, c, a5[9], 0x8B44F7AF, 12);
25 sub_401028((int)&c, d, a, b, a5[10], 0xFFFF5BB1, 17);
26 sub_401028((int)&b, c, d, a, a5[11], 0x895CD7BE, 22);
27 sub_401028((int)&a, b, c, d, a5[12], 0x6B901122, 7);
28 sub_401028((int)&d, a, b, c, a5[13], 0xFD987193, 12);
29 sub_401028((int)&c, d, a, b, a5[14], 0xA679438E, 17);
30 sub_401028((int)&b, c, d, a, a5[15], 0x49B40821, 22);
31 sub_401005(&a, b, c, d, a5[1], 0x5A122552, 5);
32 sub_401005(&d, a, b, c, a5[6], 0xC040B340, 9);
33 sub_401005(&c, d, a, b, a5[11], 0x265E5A51, 14);
34 sub_401005(&b, c, d, a, *a5, 0xE9B6C7AA, 20);
35 sub_401005(&a, b, c, d, a5[5], 0xD62F105D, 5);
36 sub_401005(&d, a, b, c, a5[10], 0x2441453, 9);
37 sub_401005(&c, d, a, b, a5[15], 0xD8A1E681, 14);
38 sub_401005(&b, c, d, a, a5[4], 0xE7D3FBC8, 20);
39 sub_401005(&a, b, c, d, a5[9], 0x21E1CDE6, 5);
40 sub_401005(&d, a, b, c, a5[14], 0xC33707D6, 9);
41 sub_401005(&c, d, a, b, a5[3], 0xF4D50D87, 14);
42 sub_401005(&b, c, d, a, a5[8], 0x455A14ED, 20);
43 sub_401005(&a, b, c, d, a5[13], 0xA9E3E905, 5);
44 sub_401005(&d, a, b, c, a5[2], 0xFCEFA3F8, 9);
00001A2E sub_401070:23
```



# 1. MD5算法



```
31 sub_401005(&a, b, c, d, a5[1], 0xF61E2562, 5);
32 sub_401005(&d, a, b, c, a5[6], 0xC040B340, 9);
33 sub_401005(&c, d, a, b, a5[11], 0x265E5A51, 14);
34 sub_401005(&b, c, d, a, *a5, 0xE9B6C7AA, 20);
35 sub_401005(&a, b, c, d, a5[5], 0xD62F105D, 5);
36 sub_401005(&d, a, b, c, a5[10], 0x2441453, 9);
37 sub_401005(&c, d, a, b, a5[15], 0xD8A1E681, 14);
38 sub_401005(&b, c, d, a, a5[4], 0xE7D3FBC8, 20);
39 sub_401005(&a, b, c, d, a5[9], 0x21E1CDE6, 5);
40 sub_401005(&d, a, b, c, a5[14], 0xC33707D6, 9);
41 sub_401005(&c, d, a, b, a5[3], 0xF4D50D87, 14);
42 sub_401005(&b, c, d, a, a5[8], 0x455A14ED, 20);
43 sub_401005(&a, b, c, d, a5[13], 0xA9E3E905, 5);
44 sub_401005(&d, a, b, c, a5[2], 0xFCEFA3F8, 9);
45 sub_401005(&c, d, a, b, a5[7], 0x676F02D9, 14);
46 sub_401005(&b, c, d, a, a5[12], 0x8D2A4C8A, 20);
47 sub_401048(&a, b, c, d, a5[5], 0xFFFFA3942, 4);
48 sub_401048(&d, a, b, c, a5[8], 0x8771F681, 11);
49 sub_401048(&c, d, a, b, a5[11], 0x6D9D6122, 16);
50 sub_401048(&b, c, d, a, a5[14], 0xFDE5380C, 23);
51 sub_401048(&a, b, c, d, a5[1], 0xA4BEEA44, 4);
52 sub_401048(&d, a, b, c, a5[4], 0x4BDECF A9, 11);
53 sub_401048(&c, d, a, b, a5[7], 0xF6BB4B60, 16);
54 sub_401048(&b, c, d, a, a5[10], 0xBEBFC70, 23);
55 sub_401048(&a, b, c, d, a5[13], 0x289B7EC6, 4);
56 sub_401048(&d, a, b, c, *a5, 0xEAA127FA, 11);
57 sub_401048(&c, d, a, b, a5[3], 0xD4EF3085, 16);
58 sub_401048(&b, c, d, a, a5[6], 0x4881D05, 23);
59 sub_401048(&a, b, c, d, a5[9], 0xD9D4D039, 4);
```

第33~48轮

第49~64轮





# 1. MD5算法

- ❑ 每一轮轮函数的第6个参数均为常量值，经对比发现，这64个常量值与MD5算法中的64个加法常数相对应，第7个参数和MD5算法的移位的位数相对应。

```
31 sub_401005(&a, b, c, d, a5[1], 0xF61E2562, 5);
32 sub_401005(&d, a, b, c, a5[6], 0xC040B340, 9);
33 sub_401005(&c, d, a, b, a5[11], 0x265E5A51, 14);
34 sub_401005(&b, c, d, a, *a5, 0xE9B6C7AA, 20);
35 sub_401005(&a, b, c, d, a5[5], 0xD62F105D, 5);
36 sub_401005(&d, a, b, c, a5[10], 0x2441453, 9);
37 sub_401005(&c, d, a, b, a5[15], 0xD8A1E681, 14);
38 sub_401005(&b, c, d, a, a5[4], 0xE7D3FBC8, 20);
39 sub_401005(&a, b, c, d, a5[9], 0x21E1CDE6, 5);
40 sub_401005(&d, a, b, c, a5[14], 0xC33707D6, 9);
41 sub_401005(&c, d, a, b, a5[3], 0xF4D50D87, 14);
42 sub_401005(&b, c, d, a, a5[8], 0x455A14ED, 20);
43 sub_401005(&a, b, c, d, a5[13], 0xA9E3E905, 5);
44 sub_401005(&d, a, b, c, a5[2], 0xFCE5A2F8, 9);
45 sub_401005(&c, d, a, b, a5[7], 0x676F02D9, 14);
46 sub_401005(&b, c, d, a, a5[12], 0x8D2A4C8A, 20);
47 sub_401048(&a, b, c, d, a5[5], 0xFFFA3942, 4);
48 sub_401048(&d, a, b, c, a5[8], 0x8771F681, 11);
49 sub_401048(&c, d, a, b, a5[11], 0x6D9D6122, 16);
50 sub_401048(&b, c, d, a, a5[14], 0xFDE5380C, 23);
51 sub_401048(&a, b, c, d, a5[1], 0xA4BEEA44, 4);
52 sub_401048(&d, a, b, c, a5[4], 0x4BDECF09, 11);
53 sub_401048(&c, d, a, b, a5[7], 0xF6BB4B60, 16);
54 sub_401048(&b, c, d, a, a5[10], 0xBEBFBC70, 23);
55 sub_401048(&a, b, c, d, a5[13], 0x289B7EC6, 4);
56 sub_401048(&d, a, b, c, *a5, 0xEAA127FA, 11);
57 sub_401048(&c, d, a, b, a5[3], 0xD4EF3085, 16);
58 sub_401048(&b, c, d, a, a5[6], 0x4881D05, 23);
59 sub_401048(&a, b, c, d, a5[9], 0xD9D4D039, 4);
```

第33~48轮

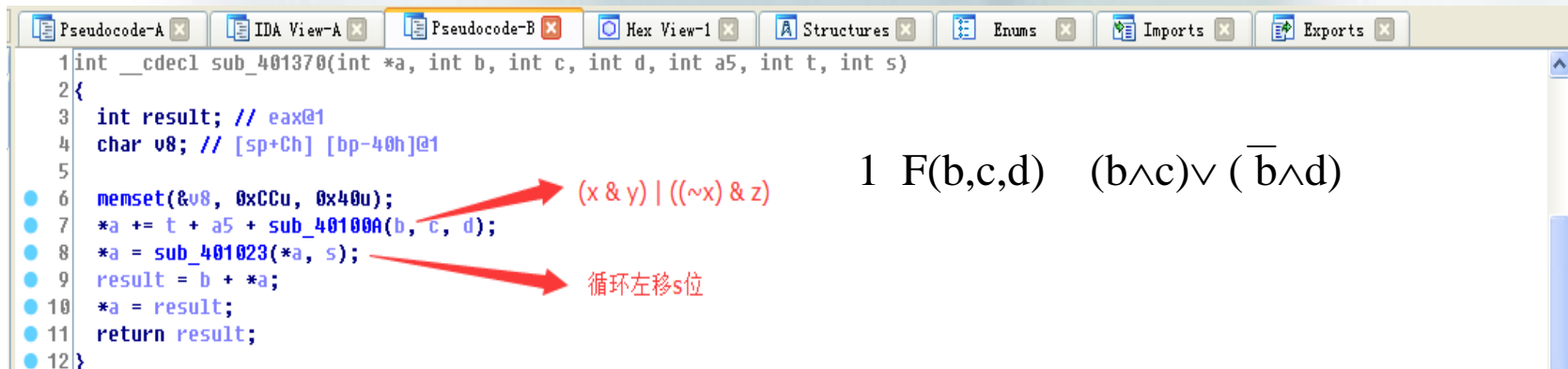
第49~64轮

# 1. MD5算法

❑ 为了进一步验证该程序的核心算法是MD5算法的猜想，可以选择一个轮函数进行分析，比如 **sub\_401028** 函数，与MD5算法中循环运算中的第一个轮函数相对应

FF(a,b,c,d,Mj,Ti,s)表示  $a = b + ((a + F(b,c,d) + Mj + Ti) \ll s)$

```
1 int __cdecl sub_401028(int a1, int a2, int a3, int a4, int a5, int a6, int a7)
2 {
3     return sub_401370(a1, a2, a3, a4, a5, a6, a7);
4 }
```



```
1 int __cdecl sub_401370(int *a, int b, int c, int d, int a5, int t, int s)
2 {
3     int result; // eax@1
4     char v8; // [sp+Ch] [bp-40h]@1
5
6     memset(&v8, 0xCCu, 0x40u);
7     *a += t + a5 + sub_40100A(b, c, d);
8     *a = sub_401023(*a, s);
9     result = b + *a;
10    *a = result;
11    return result;
12 }
```

$(x \& y) \mid ((\sim x) \& z)$

循环左移s位

1  $F(b,c,d) \quad (b \wedge c) \vee (\bar{b} \wedge d)$

# MD5 Compression Function

每一轮包含对缓冲区ABCD的16步操作所组成的一个序列。

$$a \leftarrow b + ((a + g(b,c,d) + X[k] + T[i]) \lll s)$$

其中，

$a, b, c, d$  = 缓冲区的四个字，以一个给定的次序排列；

$g$  = 基本逻辑函数F,G,H,I之一；

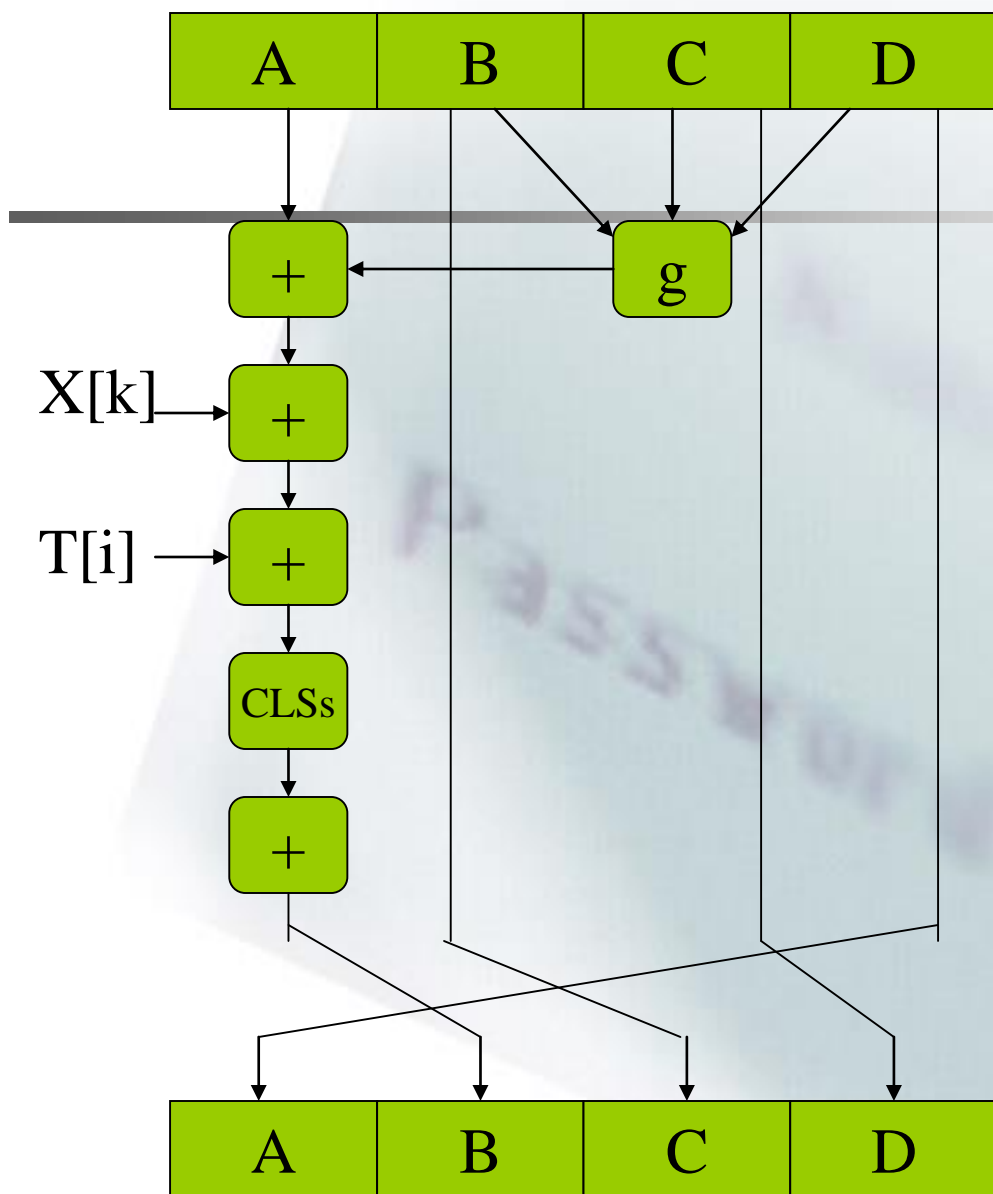
$\lll s$  = 对32位字循环左移s位

$X[k]$  =  $M[q \times 16 + k]$  = 在第q个512位数据块中的第k个32位字

$T[i]$  = 表T中的第i个32位字；

$+$  = 模  $2^{32}$  的加；





Function g	$g(b,c,d)$
1 F(b,c,d)	$(b \wedge c) \vee \overline{(b \wedge d)}$
2 G(b,c,d)	$(b \wedge d) \vee (c \wedge \overline{d})$
3 H(b,c,d)	$b \oplus c \oplus d$
4 I(b,c,d)	$c \oplus (b \vee d)$

$$\rho_2 i = (1 + 5i) \bmod 16$$

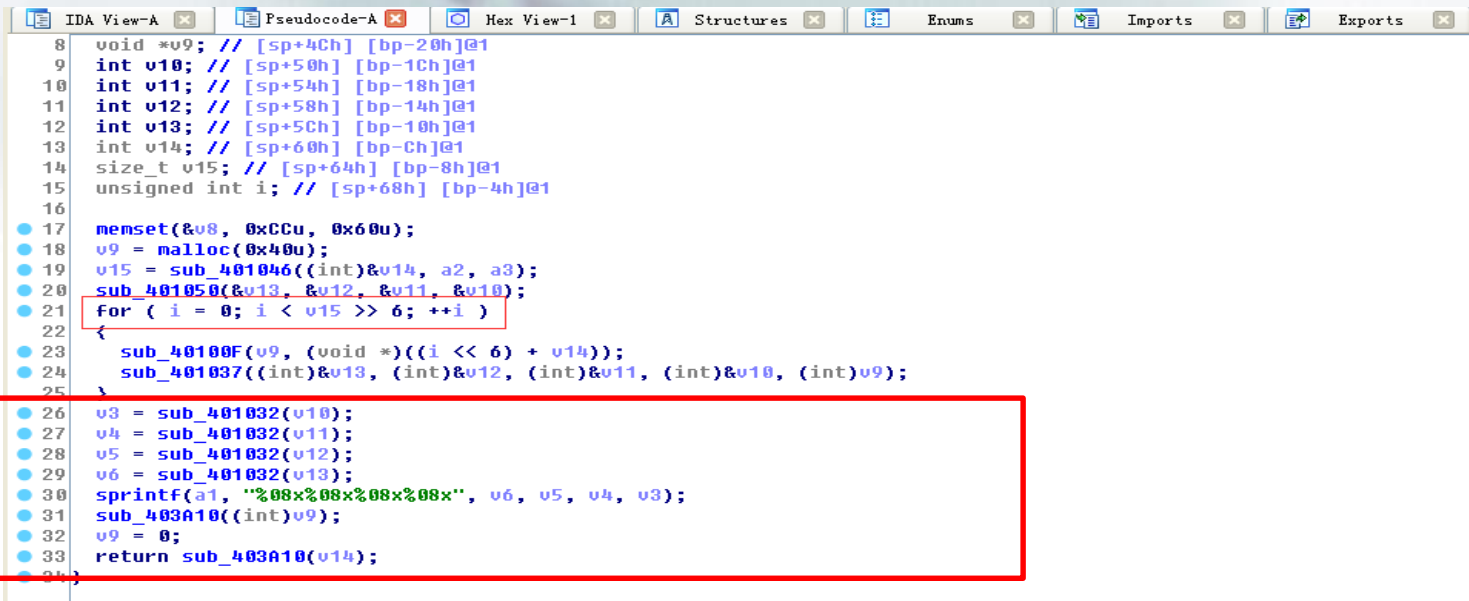
$$\rho_3 i = (5 + 3i) \bmod 16$$

$$\rho_2 i = 7i \bmod 16$$



# 1. MD5算法

- 分析至此，无论是数据填充还是轮函数，可以确定改程序的核心算法是MD5算法，完成轮函数的计算后，程序又调用了函数sub\_401032()对4个变量进行了移位操作，然后再进行链接得到最后的散列值。



```
8 void *v9; // [sp+4Ch] [bp-20h]@1
9 int v10; // [sp+50h] [bp-1Ch]@1
10 int v11; // [sp+54h] [bp-18h]@1
11 int v12; // [sp+58h] [bp-14h]@1
12 int v13; // [sp+5Ch] [bp-10h]@1
13 int v14; // [sp+60h] [bp-Ch]@1
14 size_t v15; // [sp+64h] [bp-8h]@1
15 unsigned int i; // [sp+68h] [bp-4h]@1
16
17 memset(&v8, 0xCCu, 0x60u);
18 v9 = malloc(0x40u);
19 v15 = sub_401046((int)&v14, a2, a3);
20 sub_401050(&v13, &v12, &v11, &v10);
21 for ( i = 0; i < v15 >> 6; ++i )
22 {
23     sub_40100F(v9, (void *)((i << 6) + v14));
24     sub_401037((int)&v13, (int)&v12, (int)&v11, (int)&v10, (int)v9);
25 }
26 v3 = sub_401032(v10);
27 v4 = sub_401032(v11);
28 v5 = sub_401032(v12);
29 v6 = sub_401032(v13);
30 sprintf(a1, "%08x%08x%08x%08x", v6, v5, v4, v3);
31 sub_403A10((int)v9);
32 v9 = 0;
33 return sub_403A10(v14);
34 }
```

# 1. MD5算法

- ❑ 由输入的前4个字节计算得出的散列值与字符串“**fae8a9257e154175da4193dbf6552ef6**”进行比较，然后根据**strncmp()**的比较结果输出**Correct**或**Wrong**。由于参与计算散列值的字符串长度只有4个字节，因此可以通过暴力破解的方式得到正确的**flag**。



# 1. MD5算法

IDA - C:\Documents and Settings\Administrator\桌面\课内练习\第5章\第五章\随书代码-2\5-3\程序\MD5.exe

File Edit Jump Search View Debugger Options Windows Help

No debugger

Library function Data Regular function Unexplored Instruction External symbol

Functions window

Function name  
sub\_401005  
sub\_40100A  
sub\_40100F  
sub\_401014  
sub\_401019  
sub\_40101E  
sub\_401023  
sub\_401028  
sub\_40102D  
sub\_401032  
sub\_401037  
main  
sub\_401041  
sub\_401046  
sub\_40104B  
sub\_401050  
main\_0  
sub\_401220  
sub\_401270  
sub\_4012B0  
sub\_4012F0  
sub\_401330  
sub\_401370  
sub\_401400  
sub\_401490  
sub\_401520  
sub\_4015B0  
sub\_401630  
sub\_401710  
sub\_401820  
sub\_401870  
sub\_4026F0  
\_system  
\_strncmp  
\_strcpy  
\_strlen  
\_scanf  
\_printf  
\_chkecp  
\_memcpy  
\_malloc  
\_malloc\_dbg  
\_nh\_malloc  
\_nh\_malloc\_dbg  
\_heap\_alloc

Line 1 of 253

Output window

42DC80: using guessed type int dword\_42DC80;  
42DC84: using guessed type int dword\_42DC84;

Python

AU: idle Up Disk: 35GB

开始 程序 IDA\_Pro\_v6.8 程序 IDA - C:\Do...

12:20

```
1 int main_0()
2 {
3     char v1; // [sp+Ch] [bp-h74h]@1
4     size_t v2; // [sp+4Ch] [bp-h34h]@1
5     char v3; // [sp+54h] [bp-h2Ch]@1
6     int v4; // [sp+55h] [bp-h2Bh]@1
7     char v5; // [sp+5Ch] [bp-h24h]@1
8     char v6; // [sp+5Dh] [bp-h23h]@1
9     __int16 v7; // [sp+459h] [bp-27h]@1
10    char v8; // [sp+458h] [bp-25h]@1
11    char v9; // [sp+45Ch] [bp-24h]@1
12    char v10; // [sp+45Dh] [bp-23h]@1
13
14    memset(&v1, 0xCCu, 0x474u);
15    v9 = 0;
16    memset(&v10, 0, 0x20u);
17    v5 = 0;
18    memset(&v6, 0, 0x3FCu);
19    v7 = 0;
20    v8 = 0;
21    v3 = 0;
22    v4 = 0;
23    printf("Please input your flag:\n");
24    scanf("%s", &v5);
25    v2 = strlen(&v5);
26    if ( v2 >= 5 )
27    {
28        strcpy(&v3, &v5, 4u);
29        sub_401014(&v9, (int)&v3, 4);
30        if ( !_strncmp(&v9, "fae8a9257e154175da4193dbf6552ef6", 0x20u) )
31            printf("Correct!\n");
32        else
33            printf("Wrong,try again!\n");
34    }
35    else
36    {
37        printf("Wrong,try again!\n");
38    }
39    system("pause");
40    return 0;
41 }
```

401014()函数将用户输入数据v3的前4个字节计算出HASH值，保存在v9

即计算4个字节的hash值是32个字符（128bit）的  
“fae8a9257e154175da4193dbf6552ef6”

# 1. MD5算法

@练习:

❑ 逆向MD5.EXE, 提交flag。





## 5.3 单向散列算法逆向分析

① 1. MD5算法

② 2. SHA 算法



## 2. SHA 算法

- ❑ (1) 算法原理
- ❑ (2) 逆向分析



## 2. SHA 算法

- ❑ **SHA-0**: 1993年发布，发布之后很快就被**NSA**撤回，是**SHA-1**的前身。
- ❑ **SHA-1**: 1995年发布，但**SHA-1**的安全性在**2000**年以后已经不被大多数的加密场景所接受，**2017**年荷兰密码学研究小组**CWI**和**Google**正式宣布攻破了**SHA-1**。



## 2. SHA 算法

- ❑ **SHA-2**: 2001年发布, 包括**SHA-224**、**SHA-256**、**SHA-384**、**SHA-512**、**SHA-512/224**、**SHA-512/256**。至今尚未出现对**SHA-2**有效的攻击, 它的算法跟**SHA-1**基本上仍然相似。
- ❑ **SHA-3**: 2015年正式发布, 由于对**MD5**出现成功的破解, 以及对**SHA-0**和**SHA-1**出现理论上破解的方法, **NIST**感觉需要一个与之前算法不同的, 可替换的杂凑算法, 也就是现在的**SHA-3**。



## 2. SHA 算法

④ 以SHA-256为例，对SHA系列算法介绍

- ❑ SHA-256算法输入消息的最大长度不超过比特，以512比特分组来处理输入的消息，产生的输出是256比特的消息摘要。
- ❑ 消息的分组长度为512比特。



## 2. SHA 算法

### □ 初始化链接变量

- 链接变量的中间及最终结果存储于**256**比特的缓冲区中，缓冲区可用**8**个**32**位的寄存器**A-H**表示，首先对链接变量进行初始化
- 这些初值是取自前**8**个素数(**2, 3, 5, 7, 11, 13, 17, 19**)的平方根的小数部分，用其**二进制**表示的前**32**位。

A = 0x6a09e667;

B = 0xbb67ae85;

C = 0x3c6ef372;

D = 0xa54ff53a;

E = 0x510e527f;

F = 0x9b05688c;

G = 0x1f83d9ab;

H = 0x5be0cd19;



## 2. SHA 算法

### □ 压缩函数

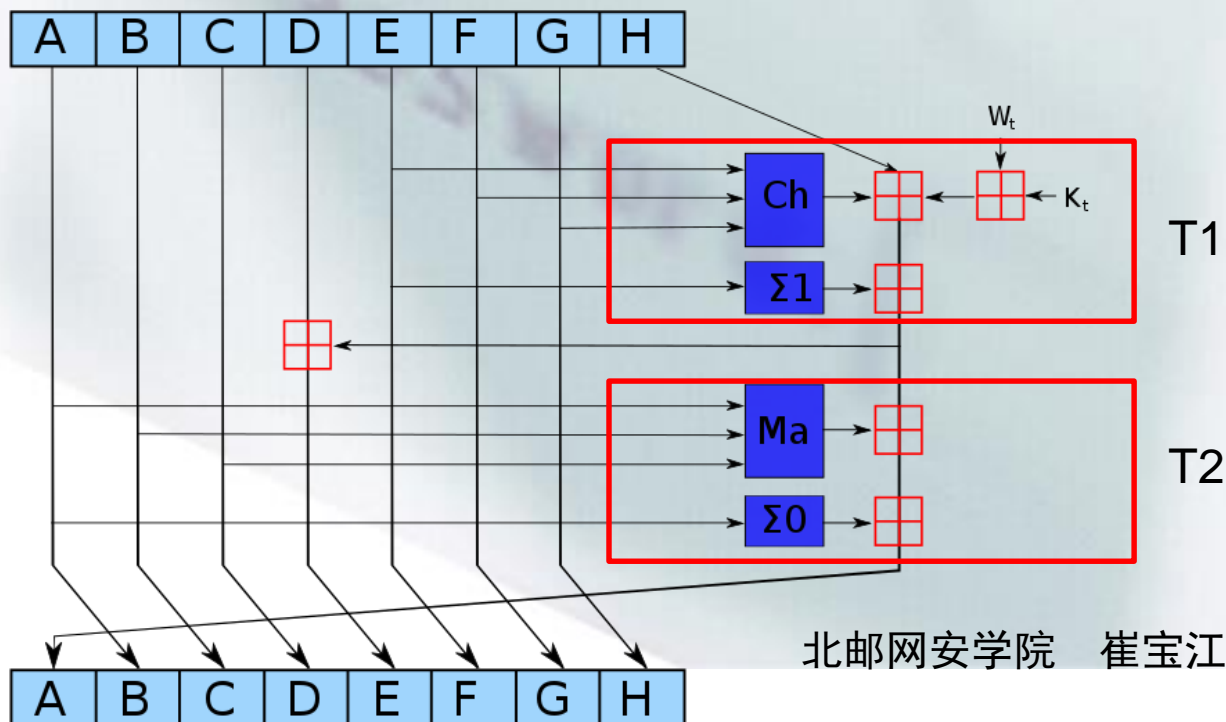
- 消息块以**512**位分组为单位进行处理，需要进行**64**步循环操作(步函数)
- 每一轮的输入均为当前处理的消息分组和上一轮输出的**256**位缓冲区**A-H**的值
- 每一步中均采用了不同的消息字 **$W_t$** 和**32**位常数 **$K_t$**



## 2. SHA 算法

### □ 步函数

- 步函数的输入为寄存器A-H的当前值、消息字 $W_t$ 和32位常数 $K_t$ ，输出为更新后的A-H值。





## 2. SHA 算法

□ 每一步都会生成两个临时的变量:

$$T_1 = \Sigma_1(E) + \text{Ch}(E, F, G) + H + W_t + K_t$$

$$T_2 = \Sigma_0(A) + \text{Maj}(A, B, C)$$

$$A = T_1 + T_2;$$

$$B = A;$$

$$C = B;$$

$$D = C;$$

$$E = D + T_1;$$

$$F = E;$$

$$G = F;$$

$$H = G;$$



## 2. SHA 算法

○其中

$$\text{Ch}(E, F, G) = (E \wedge F) \oplus (\bar{E} \wedge G)$$

$$\Sigma_1(E) = \text{ROR}^6(E) \oplus \text{ROR}^{11}(E) \oplus \text{ROR}^{25}(E)$$

$$\Sigma_0(A) = \text{ROR}^2(A) \oplus \text{ROR}^{13}(A) \oplus \text{ROR}^{22}(A)$$

$$\text{Maj}(A, B, C) = (A \wedge B) \oplus (A \wedge C) \oplus (B \wedge C)$$

其中,  $\text{ROR}^n(x)$ 表示对 32 位的变量  $x$  循环右移  $n$  位。



## 2. SHA 算法

### □ 消息字 $W_t$

消息字  $W_t$

对于  $0 \leq t \leq 15$ ,  $W_t$  直接按照消息输入分组对应的 16 个 32 位字;

对于  $16 \leq t \leq 63$ :

$$W_t = W_{t-16} + s0 + W_{t-7} + s1$$

其中

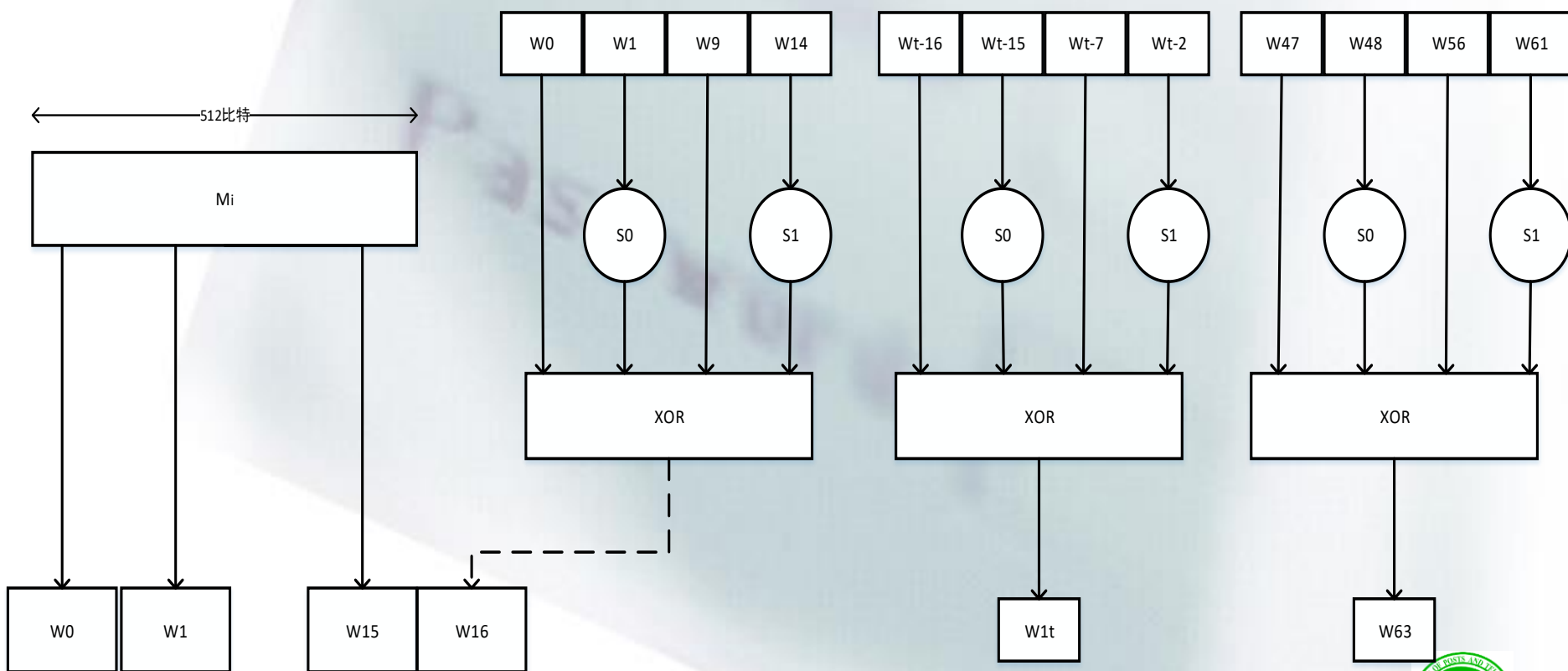
$$s0 = ROR^7(W_{t-15}) \oplus ROR^{18}(W_{t-15}) \oplus SHR^3(W_{t-15})$$

$$s1 = ROR^{17}(W_{t-2}) \oplus ROR^{19}(W_{t-2}) \oplus SHR^{10}(W_{t-2})$$

$SHR^n(x)$  表示对 32 位的变量  $x$  右移  $n$  位。

## 2. SHA 算法

□ SHA-256的64个消息字的生成过程可以表示为



## 2. SHA 算法

### □ 常数Kt

- 共使用了64个32位字长的常数，它们分别由最小的64个素数的立方根的小数部分的前32位产生(二进制表示)

K[64]={

```
0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,  
0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,  
0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,  
0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,  
0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,  
0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,  
0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,  
0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,  
0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,  
0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,  
0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,  
0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,  
0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,  
0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,  
0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,  
0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
```

}



## 2. SHA 算法

### @ 输出

- 最后一轮迭代输出的链接变量值与初始链接变量值相加，即为散列值，长度为**256**比特。



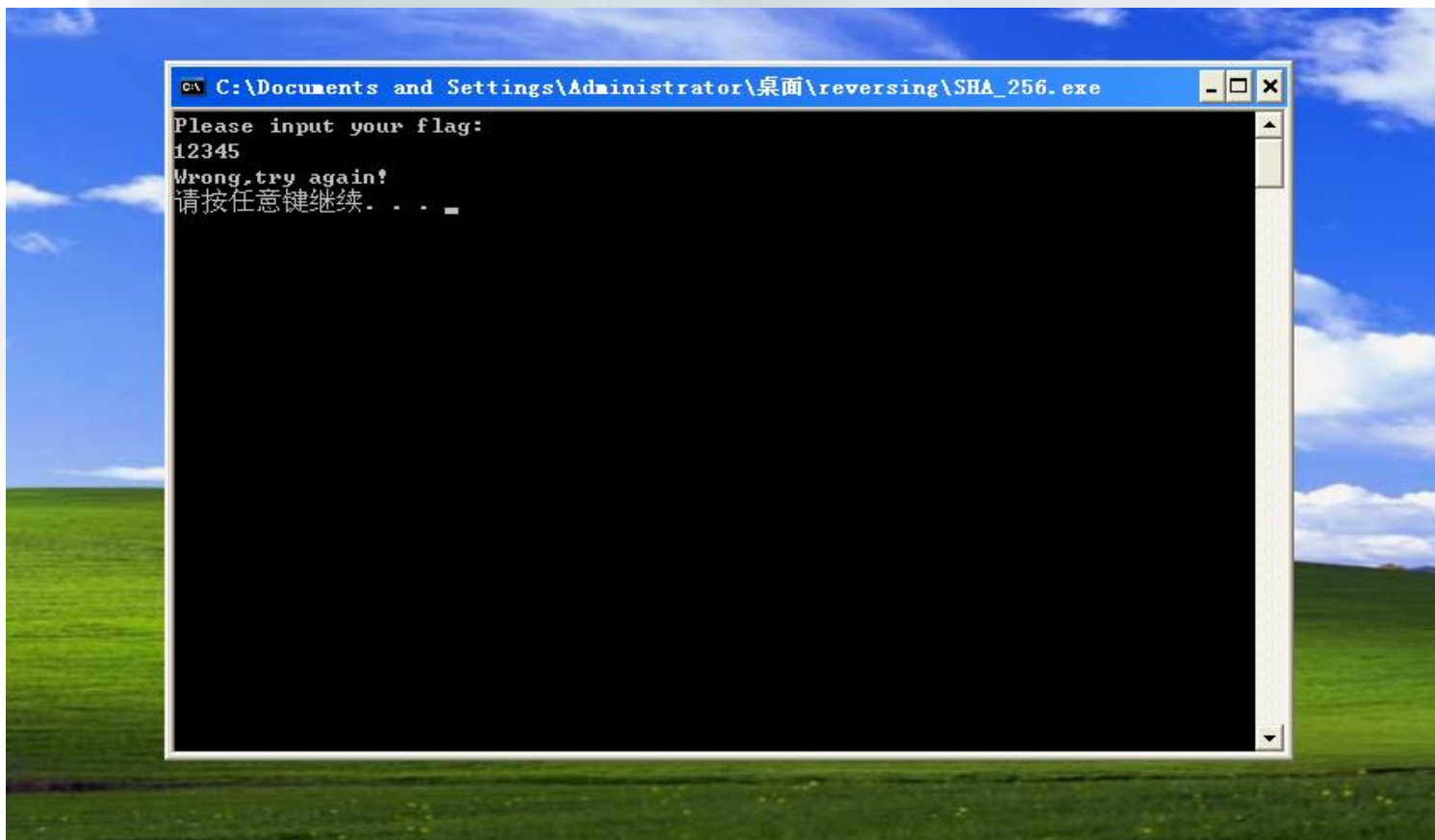
## 2. SHA 算法

- (1) 算法原理
- (2) 逆向分析



## 2. SHA 算法

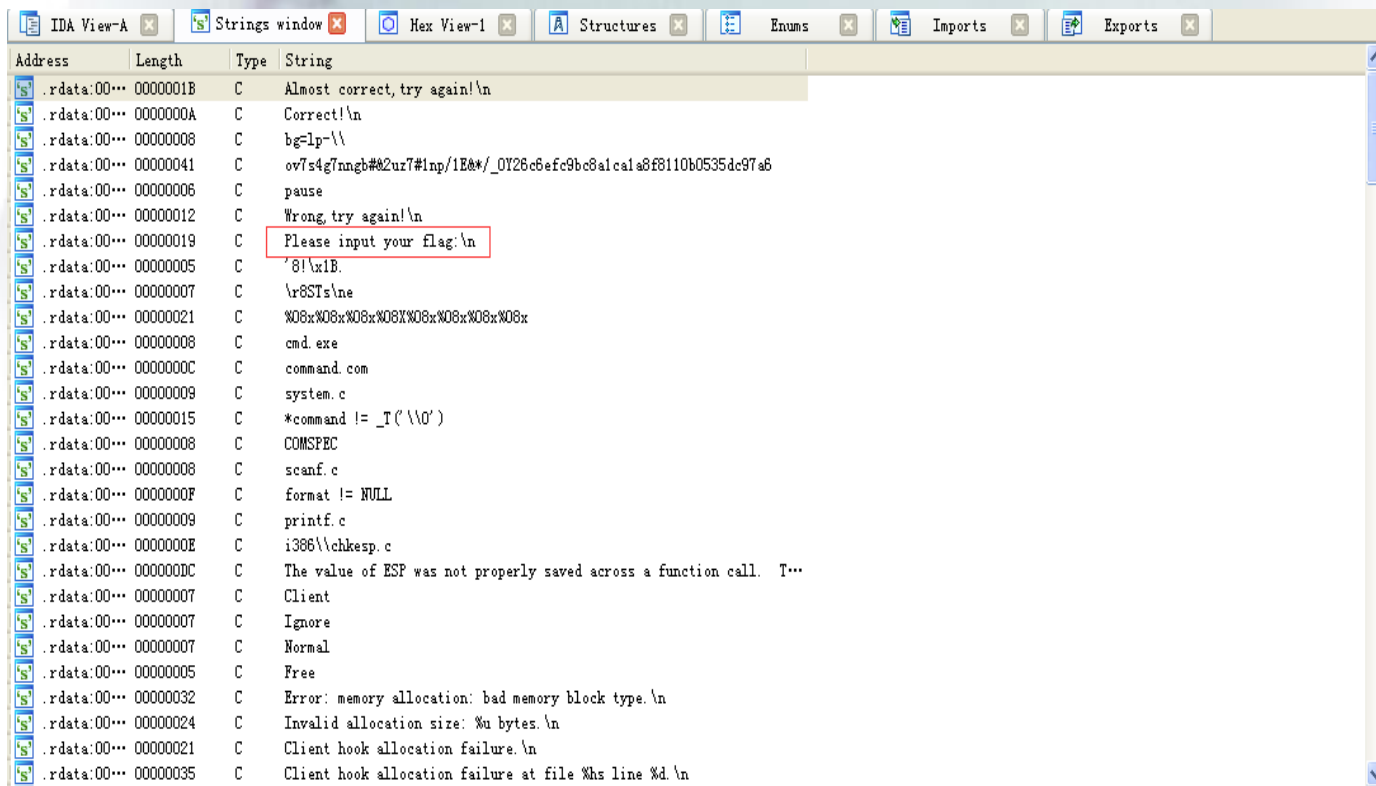
❑ 运行示例程序SHA\_256.exe，了解程序基本流程





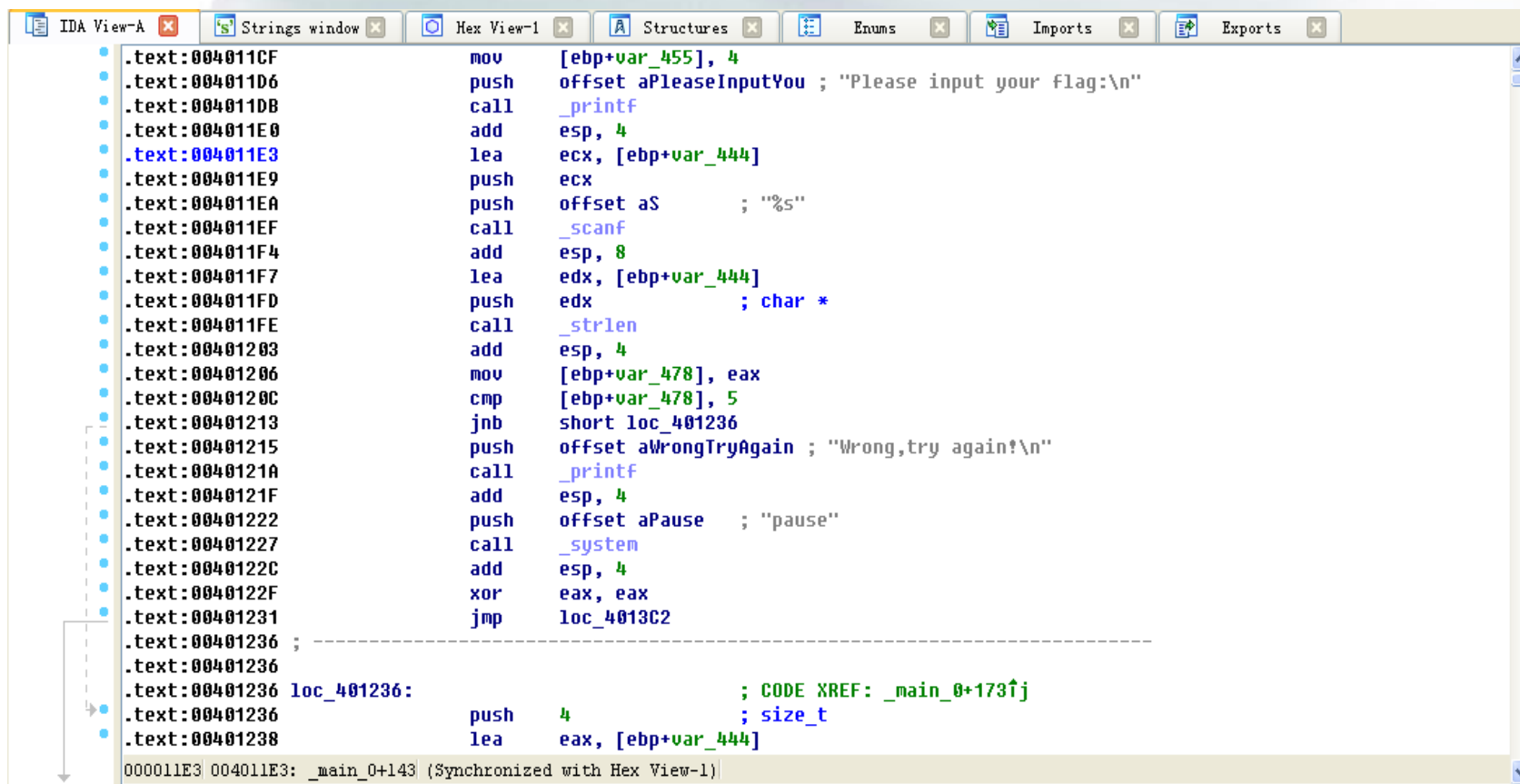
## 2. SHA 算法

- ❑ 使用IDA打开SHA\_256.exe，使用快捷键Shift+F12打开字符串窗口，根据关键字字符串定位到main函数



# 2. SHA 算法

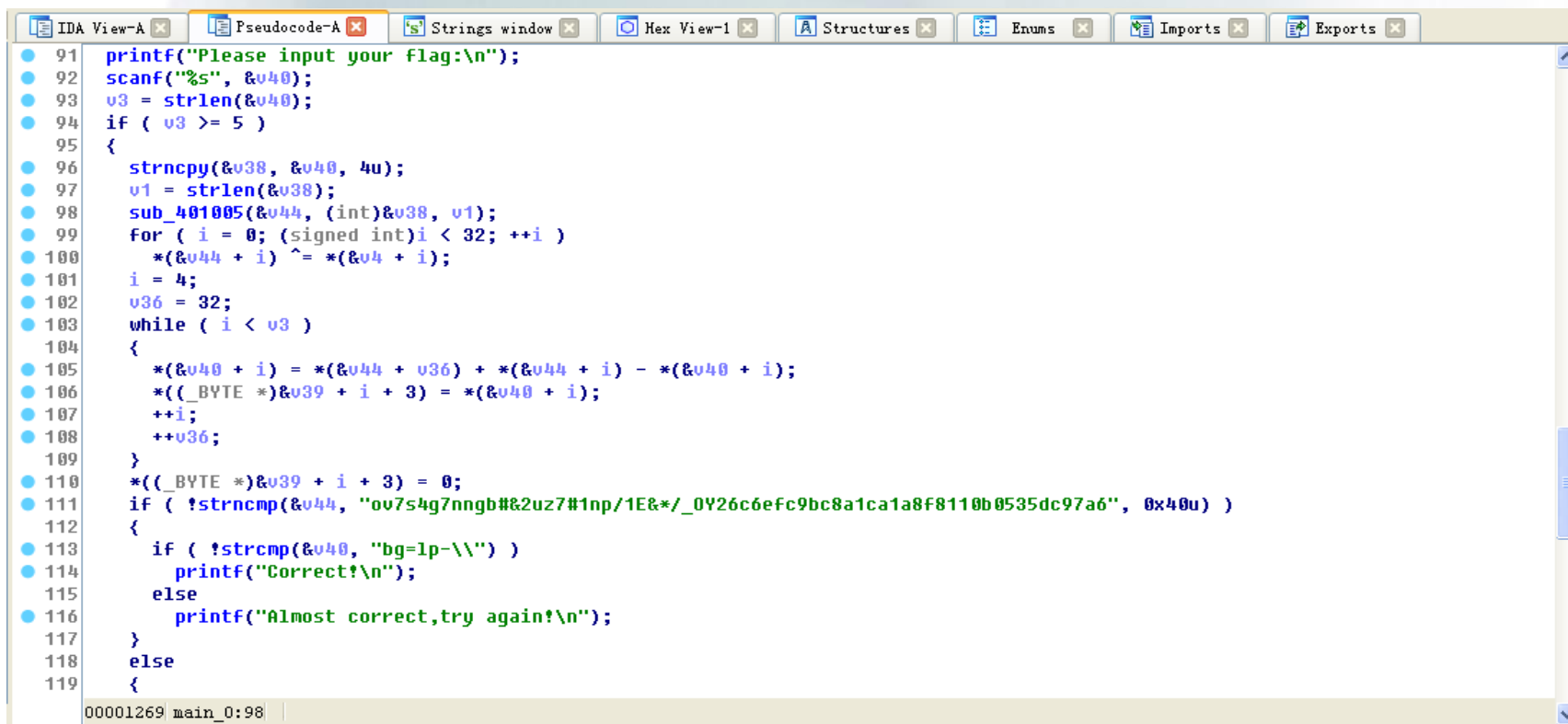
## □ Main函数



```
.text:004011CF      mov     [ebp+var_455], 4
.text:004011D6      push    offset aPleaseInputYou ; "Please input your flag:\n"
.text:004011D8      call   _printf
.text:004011E0      add     esp, 4
.text:004011E3      lea     ecx, [ebp+var_444]
.text:004011E9      push    ecx
.text:004011EA      push    offset aS ; "%5"
.text:004011EF      call   _scanf
.text:004011F4      add     esp, 8
.text:004011F7      lea     edx, [ebp+var_444]
.text:004011FD      push    edx ; char *
.text:004011FE      call   _strlen
.text:00401203      add     esp, 4
.text:00401206      mov     [ebp+var_478], eax
.text:0040120C      cmp     [ebp+var_478], 5
.text:00401213      jnb     short loc_401236
.text:00401215      push    offset aWrongTryAgain ; "Wrong, try again!\n"
.text:0040121A      call   _printf
.text:0040121F      add     esp, 4
.text:00401222      push    offset aPause ; "pause"
.text:00401227      call   _system
.text:0040122C      add     esp, 4
.text:0040122F      xor     eax, eax
.text:00401231      jmp     loc_4013C2
.text:00401236      ; -----
.text:00401236      loc_401236: ; CODE XREF: _main_0+173↑j
.text:00401236      push    4 ; size_t
.text:00401238      lea     eax, [ebp+var_444]
000011E3 004011E3: _main_0+143 (Synchronized with Hex View-1)
```

## 2. SHA 算法

□ 用快捷键F5对main函数进行反编译，了解程序整体的处理逻辑




```
91 printf("Please input your flag:\n");
92 scanf("%s", &v40);
93 v3 = strlen(&v40);
94 if ( v3 >= 5 )
95 {
96     strncpy(&v38, &v40, 4u);
97     v1 = strlen(&v38);
98     sub_401005(&v44, (int)&v38, v1);
99     for ( i = 0; (signed int)i < 32; ++i )
100         *(&v44 + i) ^= *(&v4 + i);
101     i = 4;
102     v36 = 32;
103     while ( i < v3 )
104     {
105         *(&v40 + i) = *(&v44 + v36) + *(&v44 + i) - *(&v40 + i);
106         *((_BYTE *)&v39 + i + 3) = *(&v40 + i);
107         ++i;
108         ++v36;
109     }
110     *((_BYTE *)&v39 + i + 3) = 0;
111     if ( !strcmp(&v44, "ov7s4g7nngb#&2uz7#1np/1E&*/_0Y26c6efc9bc8a1ca1a8f8110b0535dc97a6", 0x40u) )
112     {
113         if ( !strcmp(&v40, "bg=lp-\\\" )
114             printf("Correct!\n");
115         else
116             printf("Almost correct, try again!\n");
117     }
118     else
119     {
```

00001269 main\_0:98

## 2. SHA 算法

❑ 使用快捷键N修改变量名称，方便分析



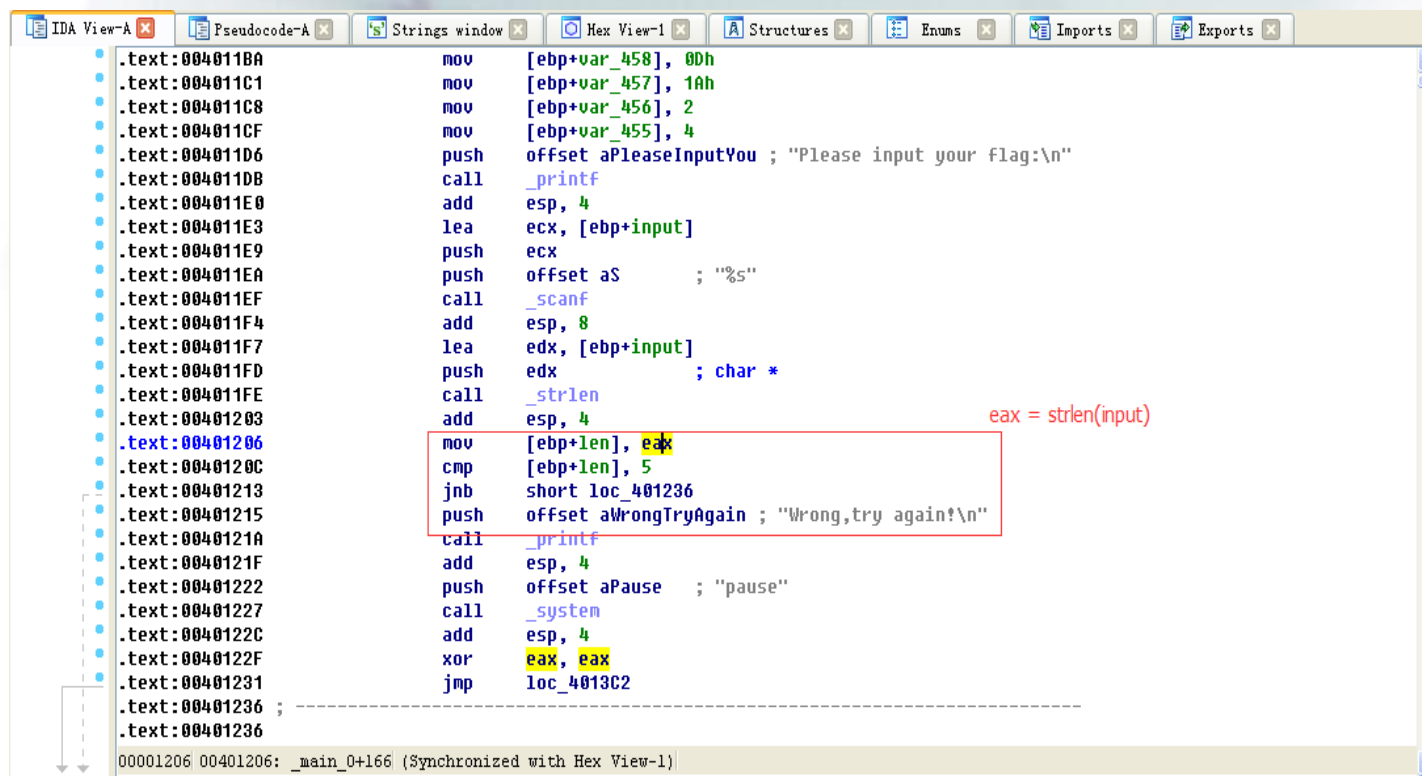
The screenshot shows the IDA Pro interface with the following windows open: IDA View, Pseudocod..., Strings win..., Pseudocod..., Hex Vie..., Structu..., Enums, Impo..., and Expo... The main window displays C code with several variables highlighted in red boxes, indicating they are ready to be renamed using the 'N' shortcut. The code is as follows:

```
88 v33 = 26;
89 v34 = 2;
90 v35 = 4;
91 printf("Please input your flag:\n");
92 scanf("%s", &input);
93 len = strlen(&input);
94 if ( len >= 5 )
95 {
96     strncpy(&v38, &input, 4u);
97     v1 = strlen(&v38);
98     sub_401005(&out, (int)&v38, v1);
99     for ( i = 0; (signed int)i < 32; ++i )
100         *(&out + i) ^= *(&v4 + i);
101     i = 4;
102     v36 = 32;
103     while ( i < len )
104     {
105         *(&input + i) = *(&out + v36) + *(&out + i) - *(&input + i);
106         *((_BYTE *)&v39 + i + 3) = *(&input + i);
107         ++i;
108         ++v36;
109     }
110     *((_BYTE *)&v39 + i + 3) = 0;
111     if ( !strcmp(&out, "ov7s4g7nngb#&2uz7#1np/1E*/_0Y26c6efc9bc8a1ca1a8f8110b0535dc97a6", 0x40u) )
112     {
113         if ( !strcmp(&input, "bg=lp-\\") )
114             printf("Correct!\n");
115         else
116             printf("Almost correct,try again!\n");
```

The status bar at the bottom shows the address 00001268 and the function name main\_0:98.

## 2. SHA 算法

❑ 在反汇编的代码中可以看到，程序首先对输入的长度进行了判断，若输入的长度小于5，则程序输出为wrong，在汇编代码里也可以看到



```
.text:004011BA    mov     [ebp+var_458], 0Dh
.text:004011C1    mov     [ebp+var_457], 1Ah
.text:004011C8    mov     [ebp+var_456], 2
.text:004011CF    mov     [ebp+var_455], 4
.text:004011D6    push    offset aPleaseInputYou ; "Please input your flag:\n"
.text:004011DB    call    _printf
.text:004011E0    add     esp, 4
.text:004011E3    lea     ecx, [ebp+input]
.text:004011E9    push    ecx
.text:004011EA    push    offset aS ; "%s"
.text:004011EF    call    _scanf
.text:004011F4    add     esp, 8
.text:004011F7    lea     edx, [ebp+input]
.text:004011FD    push    edx ; char *
.text:004011FE    call    _strlen
.text:00401203    add     esp, 4
.text:00401206    mov     [ebp+len], eax
.text:0040120C    cmp     [ebp+len], 5
.text:00401213    jnb     short loc_401236
.text:00401215    push    offset aWrongTryAgain ; "Wrong, try again!\n"
.text:0040121A    call    _printf
.text:0040121F    add     esp, 4
.text:00401222    push    offset aPause ; "pause"
.text:00401227    call    _system
.text:0040122C    add     esp, 4
.text:0040122F    xor     eax, eax
.text:00401231    jmp     loc_4013C2
.text:00401236    ; -----
.text:00401236
```

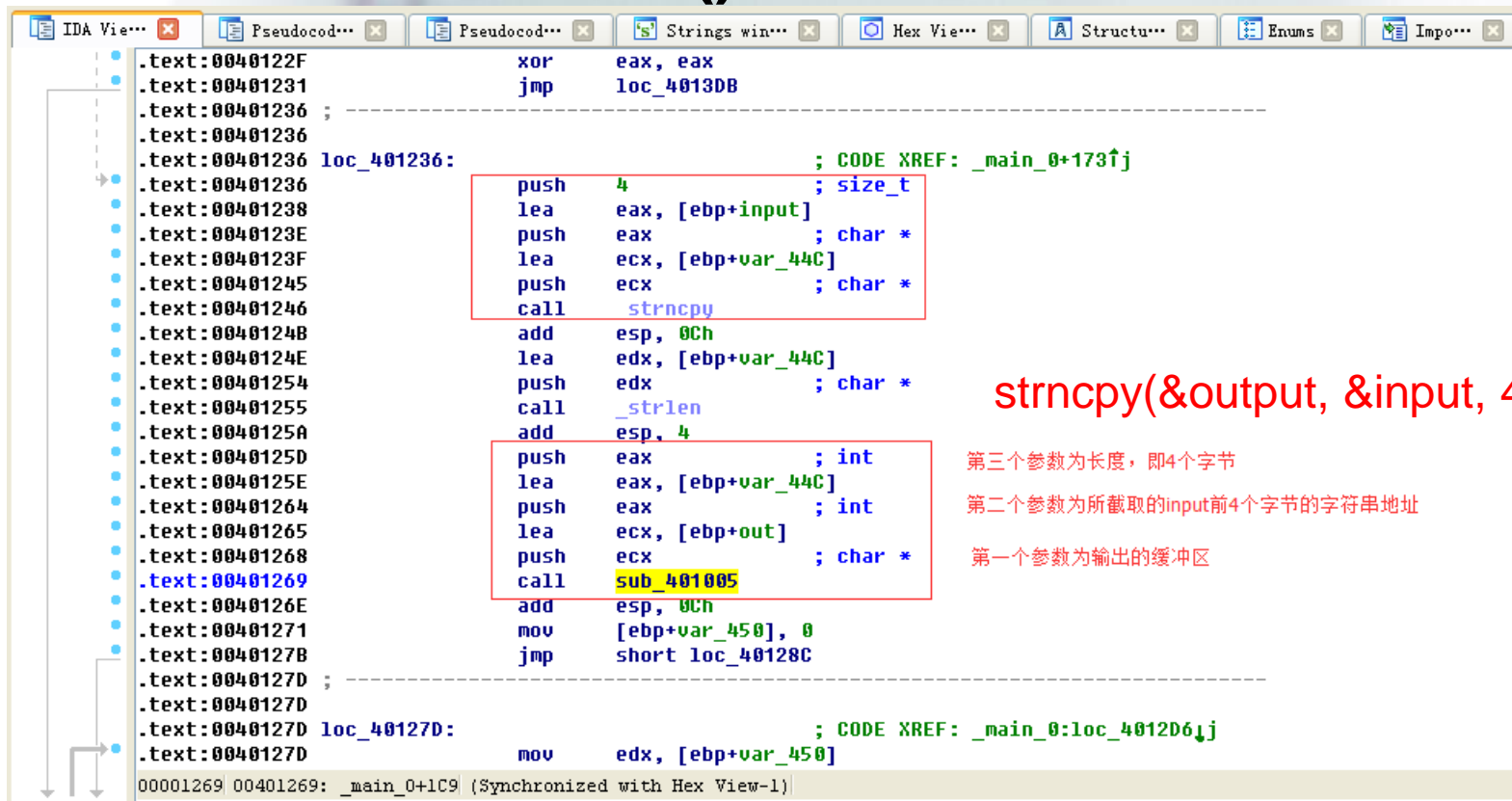
eax = strlen(input)

00001206 00401206: \_main\_0+166 (Synchronized with Hex View-1)



## 2. SHA 算法

❑ 若 `strlen(input)` 大于或等于 5，则程序跳转到 `0x00401236` 执行，截取 `input` 的前 4 个字节，作为 `sub_401005()` 的参数之一传入



```
.text:0040122F      xor     eax, eax
.text:00401231      jmp     loc_4013DB
.text:00401236      ; -----
.text:00401236      loc_401236:                                     ; CODE XREF: _main_0+173↑j
.text:00401236      push    4                                     ; size_t
.text:00401238      lea     eax, [ebp+input]
.text:0040123E      push    eax                                     ; char *
.text:0040123F      lea     ecx, [ebp+var_44C]
.text:00401245      push    ecx                                     ; char *
.text:00401246      call    strncpy
.text:00401248      add     esp, 0Ch
.text:0040124E      lea     edx, [ebp+var_44C]
.text:00401254      push    edx                                     ; char *
.text:00401255      call    _strlen
.text:0040125A      add     esp, 4
.text:0040125D      push    eax                                     ; int
.text:0040125E      lea     eax, [ebp+var_44C]
.text:00401264      push    eax                                     ; int
.text:00401265      lea     ecx, [ebp+out]
.text:00401268      push    ecx                                     ; char *
.text:00401269      call    sub_401005
.text:0040126E      add     esp, 0Ch
.text:00401271      mov     [ebp+var_450], 0
.text:0040127B      jmp     short loc_40128C
.text:0040127D      ; -----
.text:0040127D      loc_40127D:                                     ; CODE XREF: _main_0:loc_4012D6↓j
.text:0040127D      mov     edx, [ebp+var_450]
```

`strncpy(&output, &input, 4u);`

第三个参数为长度，即4个字节  
第二个参数为所截取的input前4个字节的字符串地址  
第一个参数为输出的缓冲区

00001269 00401269: \_main\_0+1C9 (Synchronized with Hex View-1)



## 2. SHA 算法

@分析函数调用

@main()

□sub\_401005()

○sub\_401F30()

❖(1)sub\_40100A()

❖(2) sub\_40100F()

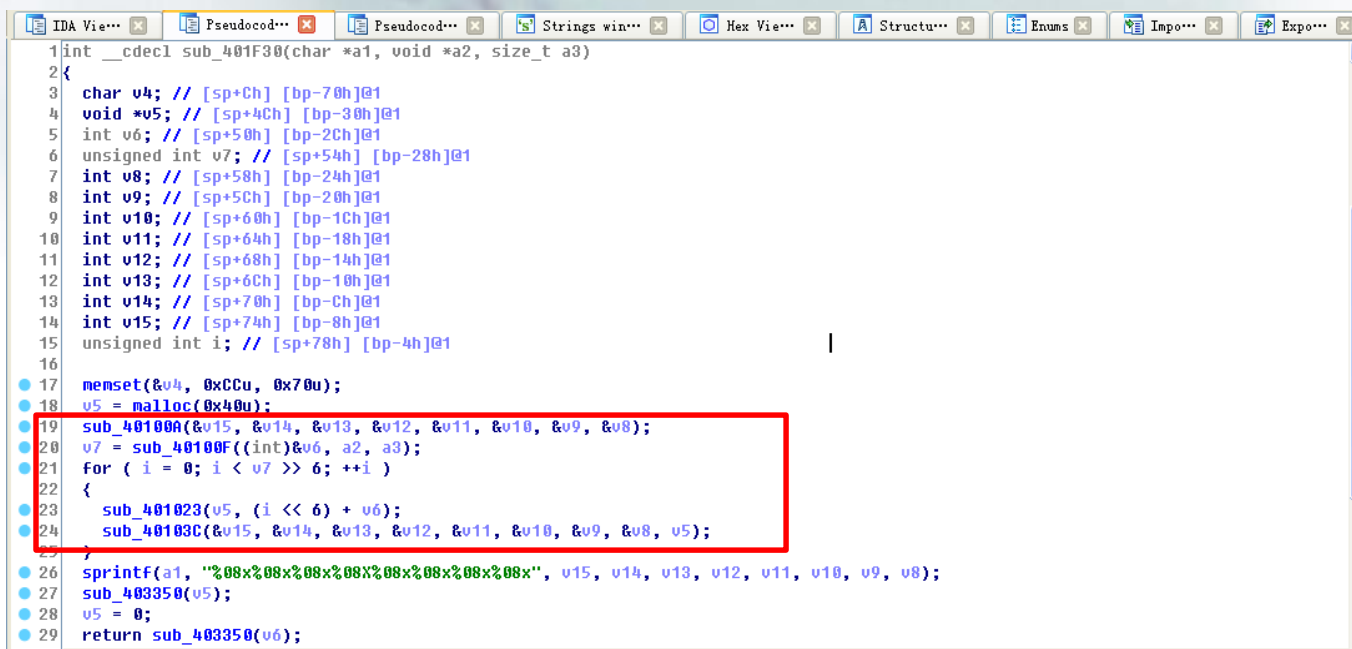
❖(3) sub\_401023()

❖(4) sub\_40103C()



## 2. SHA 算法

- 跟踪sub\_401005(), 定位至函数sub\_401F30()
- IDA反编译的结果可以看到, 该函数是程序的关键处理逻辑, 涉及到函数sub\_40100A()、sub\_40100F()、sub\_401023()、sub\_40103C()
- 下面逐一分析这4个函数



```
int __cdecl sub_401F30(char *a1, void *a2, size_t a3)
{
    char v4; // [sp+Ch] [bp-70h]@1
    void *v5; // [sp+4Ch] [bp-30h]@1
    int v6; // [sp+50h] [bp-2Ch]@1
    unsigned int v7; // [sp+54h] [bp-28h]@1
    int v8; // [sp+58h] [bp-24h]@1
    int v9; // [sp+5Ch] [bp-20h]@1
    int v10; // [sp+60h] [bp-1Ch]@1
    int v11; // [sp+64h] [bp-18h]@1
    int v12; // [sp+68h] [bp-14h]@1
    int v13; // [sp+6Ch] [bp-10h]@1
    int v14; // [sp+70h] [bp-Ch]@1
    int v15; // [sp+74h] [bp-8h]@1
    unsigned int i; // [sp+78h] [bp-4h]@1

    memset(&v4, 0xCCu, 0x70u);
    v5 = malloc(0x40u);
    sub_40100A(&v15, &v14, &v13, &v12, &v11, &v10, &v9, &v8);
    v7 = sub_40100F((int)&v6, a2, a3);
    for ( i = 0; i < v7 >> 6; ++i )
    {
        sub_401023(v5, (i << 6) + v6);
        sub_40103C(&v15, &v14, &v13, &v12, &v11, &v10, &v9, &v8, v5);
    }
    sprintf(a1, "%08x%08x%08x%08x%08x%08x%08x%08x", v15, v14, v13, v12, v11, v10, v9, v8);
    sub_403350(v5);
    v5 = 0;
    return sub_403350(v6);
}
```



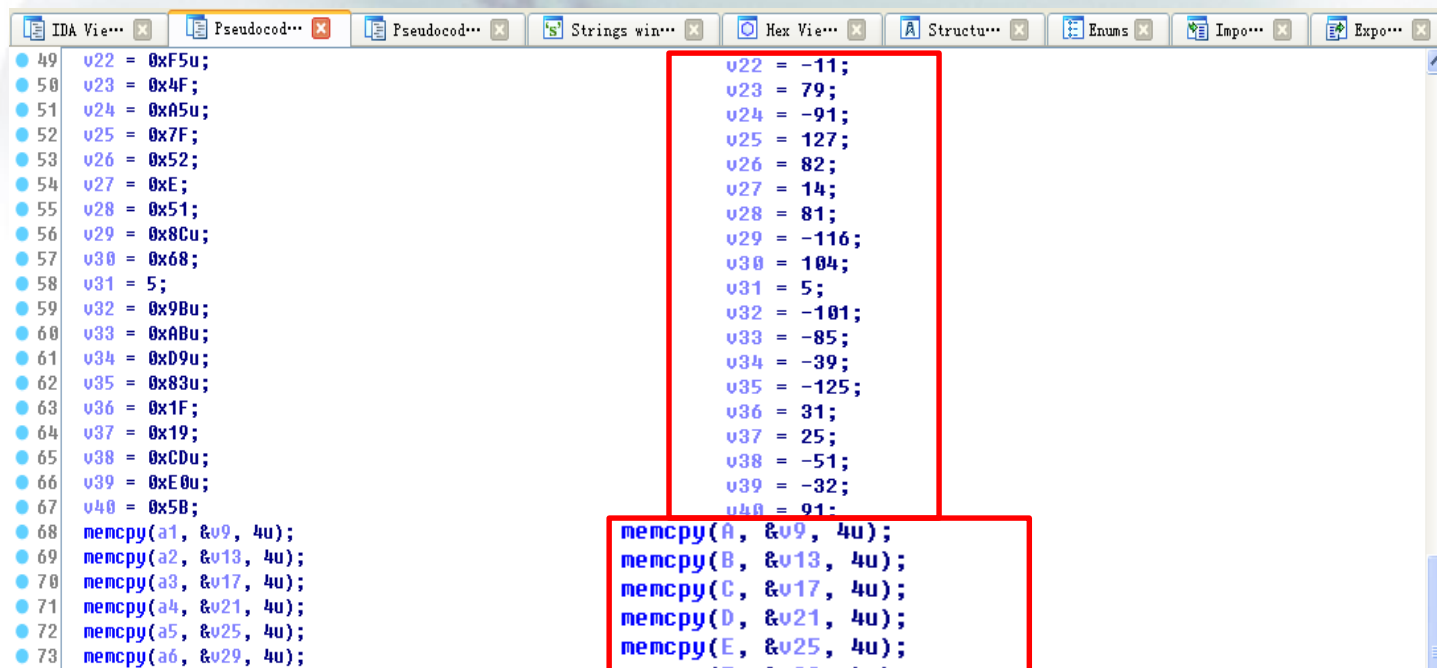


## 2. SHA 算法

### ❑ (1) 函数sub\_40100A()

○sub\_40100A()函数比较简单，是对8个变量进行赋值，为方便分析，这里使用快捷键N将这8个变量名称修改为A~H

○使用快捷键H可将数字转换为十六进制显示



```
49  v22 = 0xF5u;
50  v23 = 0x4F;
51  v24 = 0xA5u;
52  v25 = 0x7F;
53  v26 = 0x52;
54  v27 = 0xE;
55  v28 = 0x51;
56  v29 = 0x8Cu;
57  v30 = 0x68;
58  v31 = 5;
59  v32 = 0x9Bu;
60  v33 = 0xABu;
61  v34 = 0xD9u;
62  v35 = 0x83u;
63  v36 = 0x1F;
64  v37 = 0x19;
65  v38 = 0xCDu;
66  v39 = 0xE0u;
67  v40 = 0x5B;
68  memcpy(a1, &v9, 4u);
69  memcpy(a2, &v13, 4u);
70  memcpy(a3, &v17, 4u);
71  memcpy(a4, &v21, 4u);
72  memcpy(a5, &v25, 4u);
73  memcpy(a6, &v29, 4u);
```

```
v22 = -11;
v23 = 79;
v24 = -91;
v25 = 127;
v26 = 82;
v27 = 14;
v28 = 81;
v29 = -116;
v30 = 104;
v31 = 5;
v32 = -101;
v33 = -85;
v34 = -39;
v35 = -125;
v36 = 31;
v37 = 25;
v38 = -51;
v39 = -32;
v40 = 91;

memcpy(A, &v9, 4u);
memcpy(B, &v13, 4u);
memcpy(C, &v17, 4u);
memcpy(D, &v21, 4u);
memcpy(E, &v25, 4u);
```



## 2. SHA 算法

❑ 执行完sub\_40100A，变量A~H在内存中的存储为

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000h:	67	E6	09	6A	85	AE	67	BB	72	F3	6E	3C	3A	F5	4F	A5
0010h:	7F	52	0E	51	8C	68	05	9B	AB	D9	83	1F	19	CD	E0	5B
0020h:																



## 2. SHA 算法

□ 由于计算机存储为小端模式，所以变量A~H真正的数值为

A = 0x6a09e667;

B = 0xbb67ae85;

C = 0x3c6ef372;

D = 0xa54ff53a;

E = 0x510e527f;

F = 0x9b05688c;

G = 0x1f83d9ab;

H = 0x5be0cd19;



## 2. SHA 算法

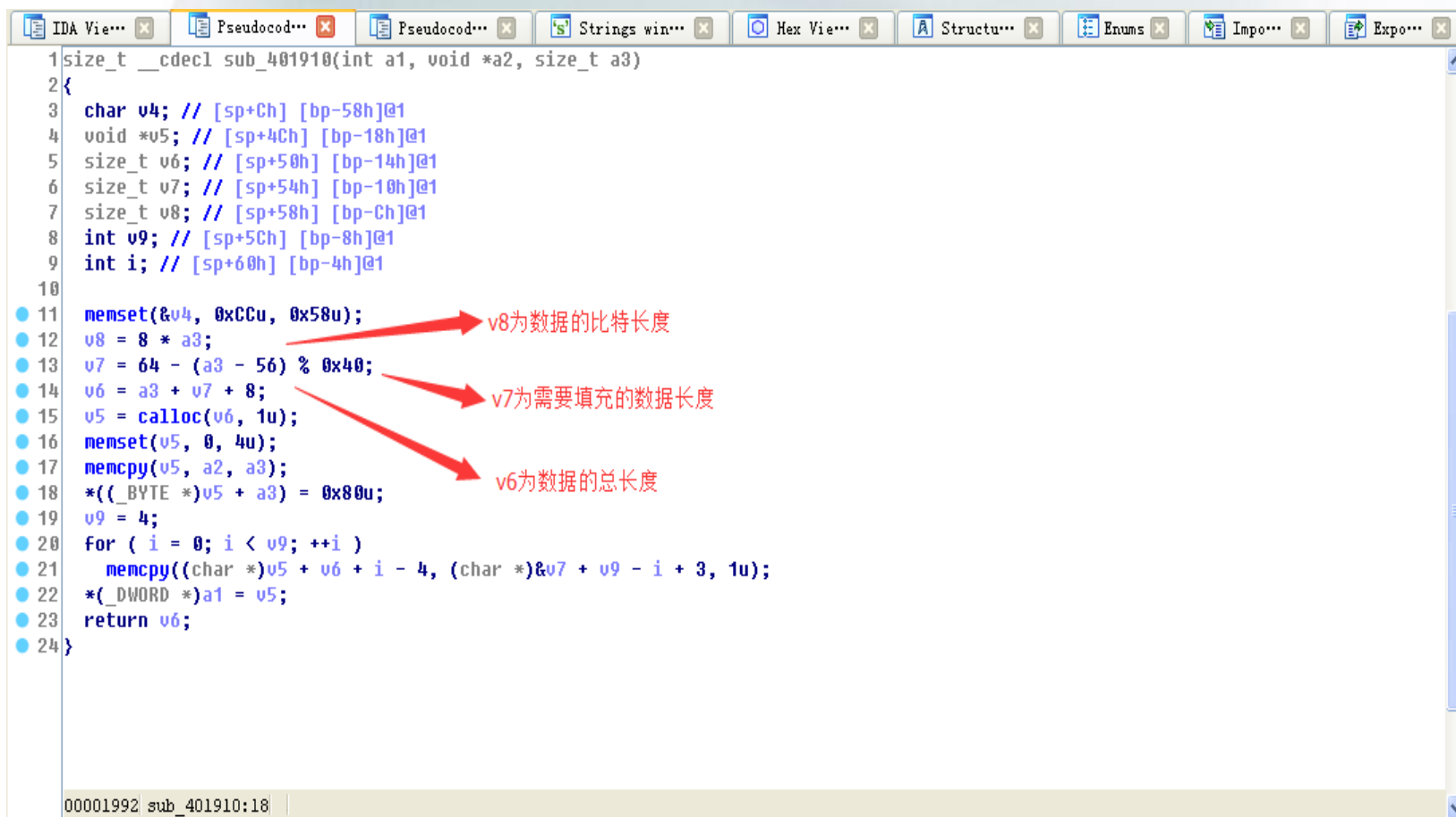
### ❑ (2) sub\_40100F()函数

- 根据一些特殊的语句，推测出这个函数的作用可能是数据填充，因为函数中出现了**56**、**64**以及模**64**这些数值和运算，联想到某些算法的填充规则，数据填充后使得数据的比特长度对**512**取模等于**448**，换算为字节运算，即填充后的长度对**64**取模等于**56**。
- 填充的方法是先将比特“**1**”添加到数据的末尾，再添加若干**0**。填充完毕后再添加一个**64**比特长的块来存储消息长度，其值等于填充前消息长度的二进制表示



## 2. SHA 算法

### ❑ (2) sub\_40100F()函数



```
1 size_t __cdecl sub_40100F(int a1, void *a2, size_t a3)
2 {
3     char v4; // [sp+Ch] [bp-58h]@1
4     void *v5; // [sp+4Ch] [bp-18h]@1
5     size_t v6; // [sp+50h] [bp-14h]@1
6     size_t v7; // [sp+54h] [bp-10h]@1
7     size_t v8; // [sp+58h] [bp-Ch]@1
8     int v9; // [sp+5Ch] [bp-8h]@1
9     int i; // [sp+60h] [bp-4h]@1
10
11     memset(&v4, 0xCCu, 0x58u);
12     v8 = 8 * a3;
13     v7 = 64 - (a3 - 56) % 0x40;
14     v6 = a3 + v7 + 8;
15     v5 = calloc(v6, 1u);
16     memset(v5, 0, 4u);
17     memcpy(v5, a2, a3);
18     *((_BYTE *)v5 + a3) = 0x80u;
19     v9 = 4;
20     for ( i = 0; i < v9; ++i )
21         memcpy((char *)v5 + v6 + i - 4, (char *)&v7 + v9 - i + 3, 1u);
22     *(_DWORD *)a1 = v5;
23     return v6;
24 }
```

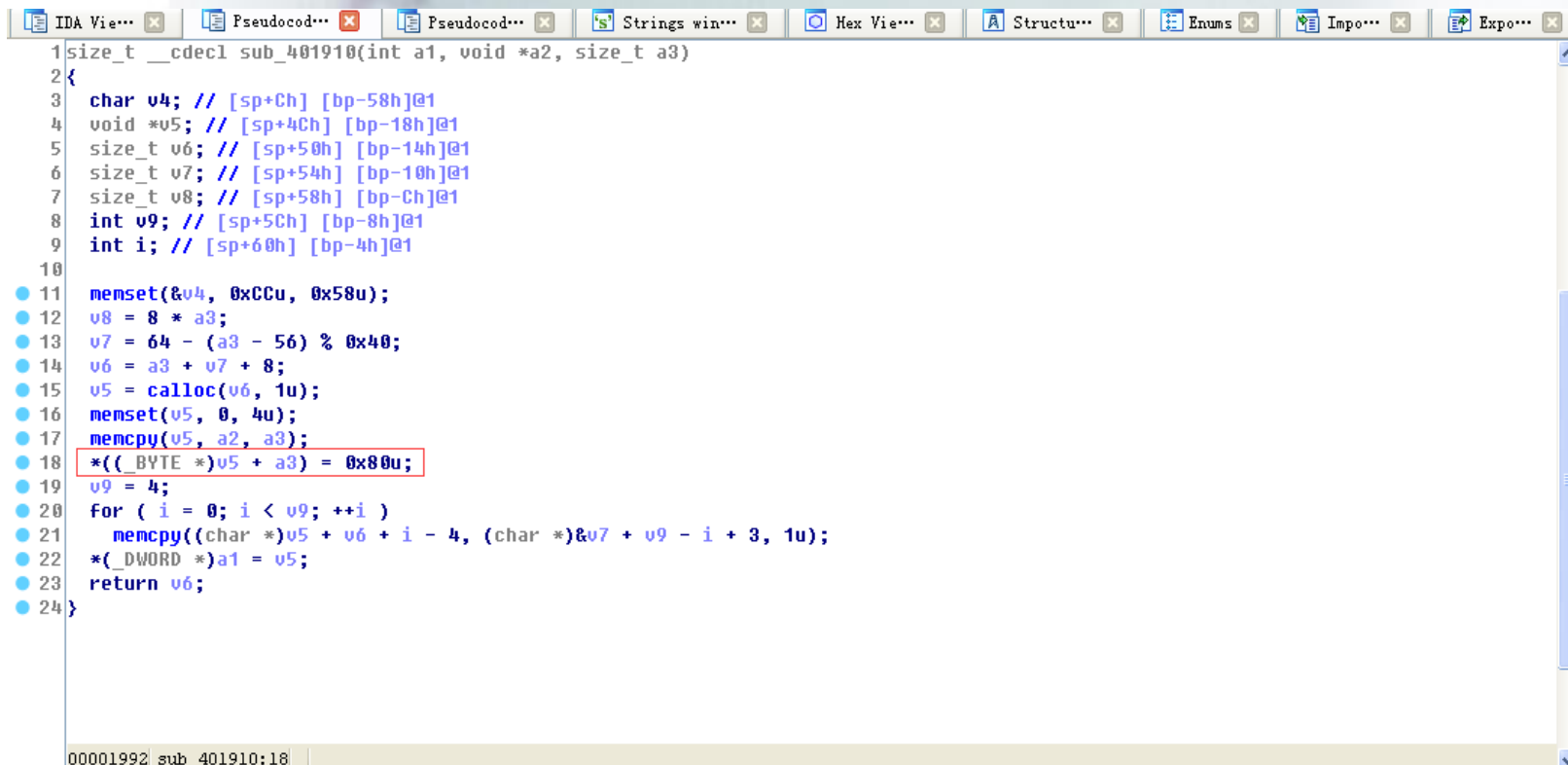
Annotations:

- v8为数据的比特长度 (points to line 12)
- v7为需要填充的数据长度 (points to line 13)
- v6为数据的总长度 (points to line 14)

00001992 sub\_40100F:18

## 2. SHA 算法

❑ 另外0x80其实是比特串“10000000”，与数据填充规则中追加一个比特1，再填充0的填充规则相符合



```
1 size_t __cdecl sub_401910(int a1, void *a2, size_t a3)
2 {
3     char v4; // [sp+Ch] [bp-58h]@1
4     void *v5; // [sp+4Ch] [bp-18h]@1
5     size_t v6; // [sp+50h] [bp-14h]@1
6     size_t v7; // [sp+54h] [bp-10h]@1
7     size_t v8; // [sp+58h] [bp-Ch]@1
8     int v9; // [sp+5Ch] [bp-8h]@1
9     int i; // [sp+60h] [bp-4h]@1
10
11     memset(&v4, 0xCCu, 0x58u);
12     v8 = 8 * a3;
13     v7 = 64 - (a3 - 56) % 0x40;
14     v6 = a3 + v7 + 8;
15     v5 = calloc(v6, 1u);
16     memset(v5, 0, 4u);
17     memcpy(v5, a2, a3);
18     *((_BYTE *)v5 + a3) = 0x80u;
19     v9 = 4;
20     for (i = 0; i < v9; ++i)
21         memcpy((char *)v5 + v6 + i - 4, (char *)&v7 + v9 - i + 3, 1u);
22     *(_DWORD *)a1 = v5;
23     return v6;
24 }
```

00001992 sub\_401910:18

## 2. SHA 算法

- ❑ 因此，函数sub\_40100F()的功能是数据填充
- ❑ 此外，sub\_401005()函数中的for循环的循环长度为填充后的总长度右移6位
  - 相当于除以64，这也证实了该程序中数据分组长度为64字节(512比特)

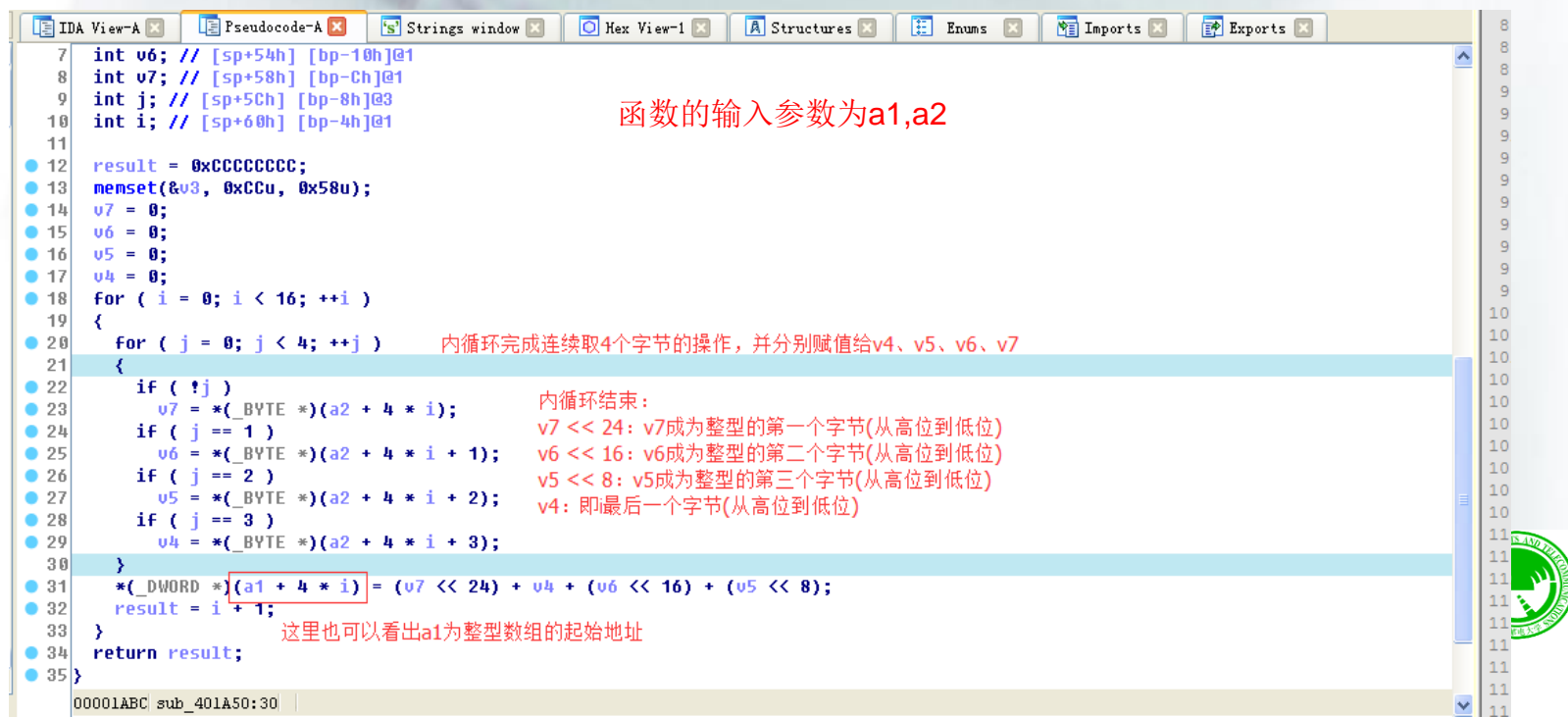
```
2{
3  char v4; // [sp+Ch] [bp-70h]@1
4  void *v5; // [sp+4Ch] [bp-30h]@1
5  int v6; // [sp+50h] [bp-2Ch]@1
6  size_t v7; // [sp+54h] [bp-28h]@1
7  int v8; // [sp+58h] [bp-24h]@1
8  int v9; // [sp+5Ch] [bp-20h]@1
9  int v10; // [sp+60h] [bp-1Ch]@1
10 int v11; // [sp+64h] [bp-18h]@1
11 int v12; // [sp+68h] [bp-14h]@1
12 int v13; // [sp+6Ch] [bp-10h]@1
13 int v14; // [sp+70h] [bp-Ch]@1
14 int v15; // [sp+74h] [bp-8h]@1
15 unsigned int i; // [sp+78h] [bp-4h]@1
16
17 memset(&v4, 0xCCu, 0x70u);
18 v5 = malloc(0x40u);
19 sub_40100A(&v15, &v14, &v13, &v12, &v11, &v10, &v9, &v8);
20 v7 = sub_40100F((int)&v6, a2, a3);
21 For ( i = 0; i < v7 >> 6; ++i )
22 {
23     sub_401023(v5, (i << 6) + v6);
24     sub_40103C(&v15, &v14, &v13, &v12, &v11, &v10, &v9, &v8, v5);
25 }
26 sprintf(a1, "%08x%08x%08x%08x%08x%08x%08x%08x", v15, v14, v13, v12, v11, v10, v9, v8);
27 sub_403350(v5);
28 v5 = 0;
29 return sub_403350(v6);
30 }
```



## 2. SHA 算法

### ❑ (3) sub\_401023()函数

- 根据函数的反编译结果我们可以看出该函数对数据做了一些比特移位操作，本质上是将char类型的输入转化为整型(int)



```
7 int v6; // [sp+54h] [bp-10h]@1
8 int v7; // [sp+58h] [bp-Ch]@1
9 int j; // [sp+5Ch] [bp-8h]@3
10 int i; // [sp+60h] [bp-4h]@1
11
12 result = 0xCCCCCCCC;
13 memset(&v3, 0xCCu, 0x58u);
14 v7 = 0;
15 v6 = 0;
16 v5 = 0;
17 v4 = 0;
18 for ( i = 0; i < 16; ++i )
19 {
20     for ( j = 0; j < 4; ++j )
21     {
22         if ( !j )
23             v7 = *(_BYTE *) (a2 + 4 * i);
24         if ( j == 1 )
25             v6 = *(_BYTE *) (a2 + 4 * i + 1);
26         if ( j == 2 )
27             v5 = *(_BYTE *) (a2 + 4 * i + 2);
28         if ( j == 3 )
29             v4 = *(_BYTE *) (a2 + 4 * i + 3);
30     }
31     *(_DWORD *) (a1 + 4 * i) = (v7 << 24) + v4 + (v6 << 16) + (v5 << 8);
32     result = i + 1;
33 }
34 return result;
35 }
```

函数的输入参数为a1,a2

内循环完成连续取4个字节的操作，并分别赋值给v4、v5、v6、v7

内循环结束：  
v7 << 24: v7成为整型的第一个字节(从高位到低位)  
v6 << 16: v6成为整型的第二个字节(从高位到低位)  
v5 << 8: v5成为整型的第三个字节(从高位到低位)  
v4: 即最后一个字节(从高位到低位)

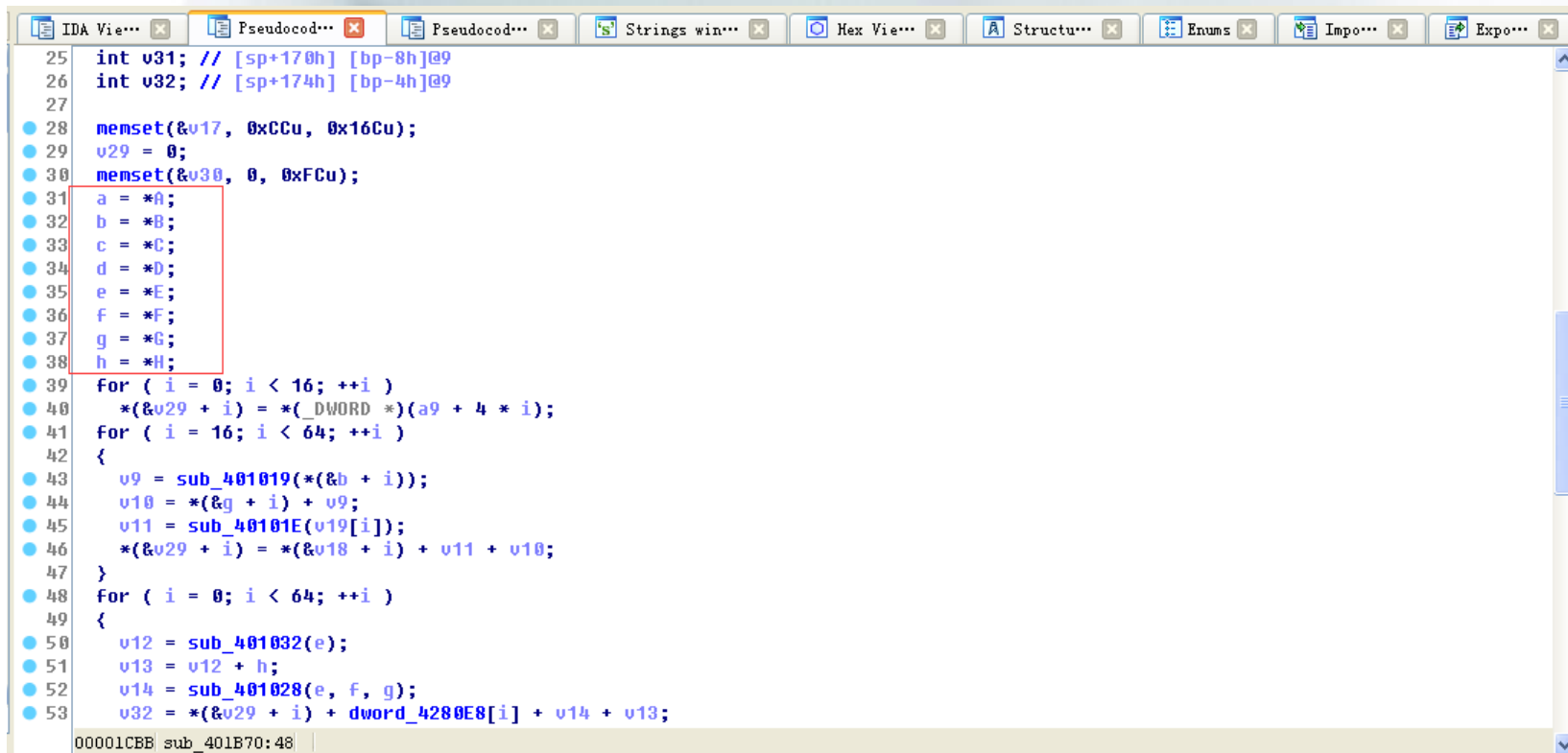
这里也可以看出a1为整型数组的起始地址



## 2. SHA 算法

### ❑ (4) 函数sub\_40103C()

○ 首先使用快捷键N修改相关变量的名称，便于分析



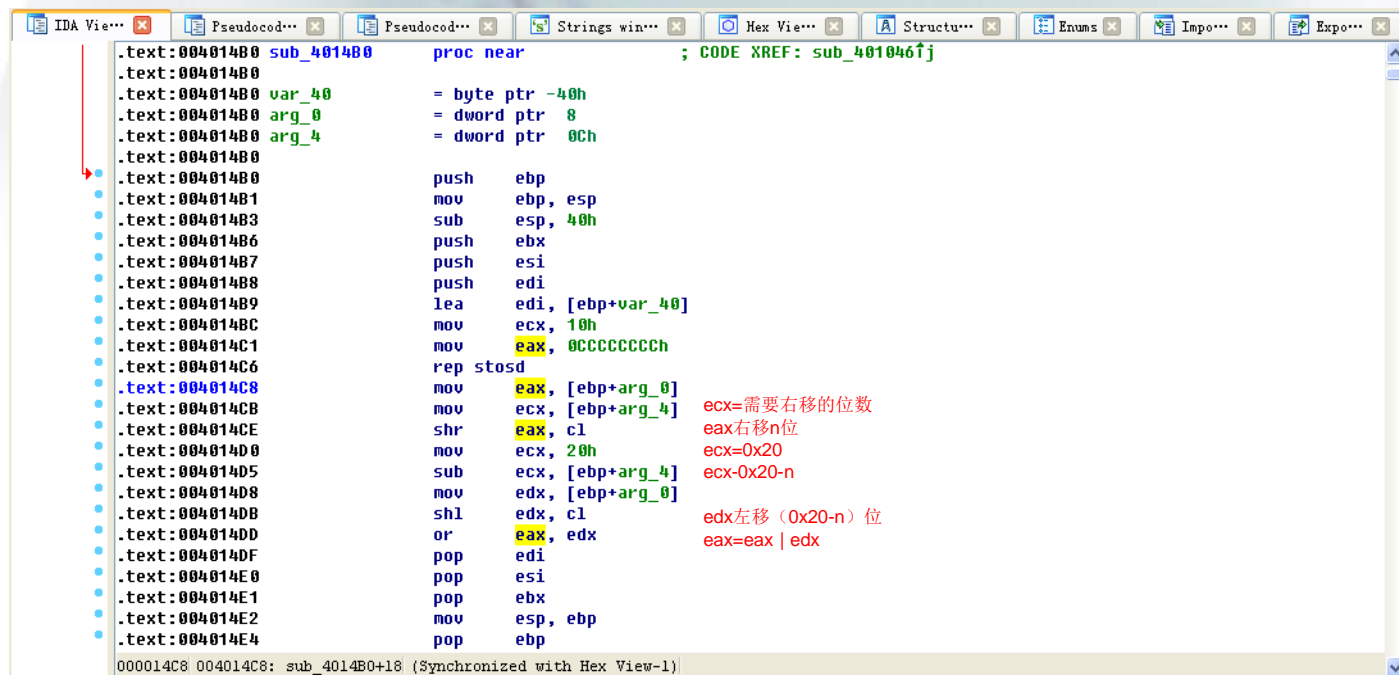
```
25 int v31; // [sp+170h] [bp-8h]@9
26 int v32; // [sp+174h] [bp-4h]@9
27
28 memset(&v17, 0xCCu, 0x16Cu);
29 v29 = 0;
30 memset(&v30, 0, 0xFCu);
31 a = *A;
32 b = *B;
33 c = *C;
34 d = *D;
35 e = *E;
36 f = *F;
37 g = *G;
38 h = *H;
39 for ( i = 0; i < 16; ++i )
40     *(&v29 + i) = *(_DWORD *) (a9 + 4 * i);
41 for ( i = 16; i < 64; ++i )
42 {
43     v9 = sub_401019(*(&b + i));
44     v10 = *(&g + i) + v9;
45     v11 = sub_40101E(v19[i]);
46     *(&v29 + i) = *(&v18 + i) + v11 + v10;
47 }
48 for ( i = 0; i < 64; ++i )
49 {
50     v12 = sub_401032(e);
51     v13 = v12 + h;
52     v14 = sub_401028(e, f, g);
53     v32 = *(&v29 + i) + dword_4280E8[i] + v14 + v13;
```

00001CBB sub\_401B70:48

## 2. SHA 算法

□ 根据反编译以及汇编代码可以看出

○ 该函数对变量A~H进行了64轮的计算操作，其中涉及到相关逻辑运算和移位操作

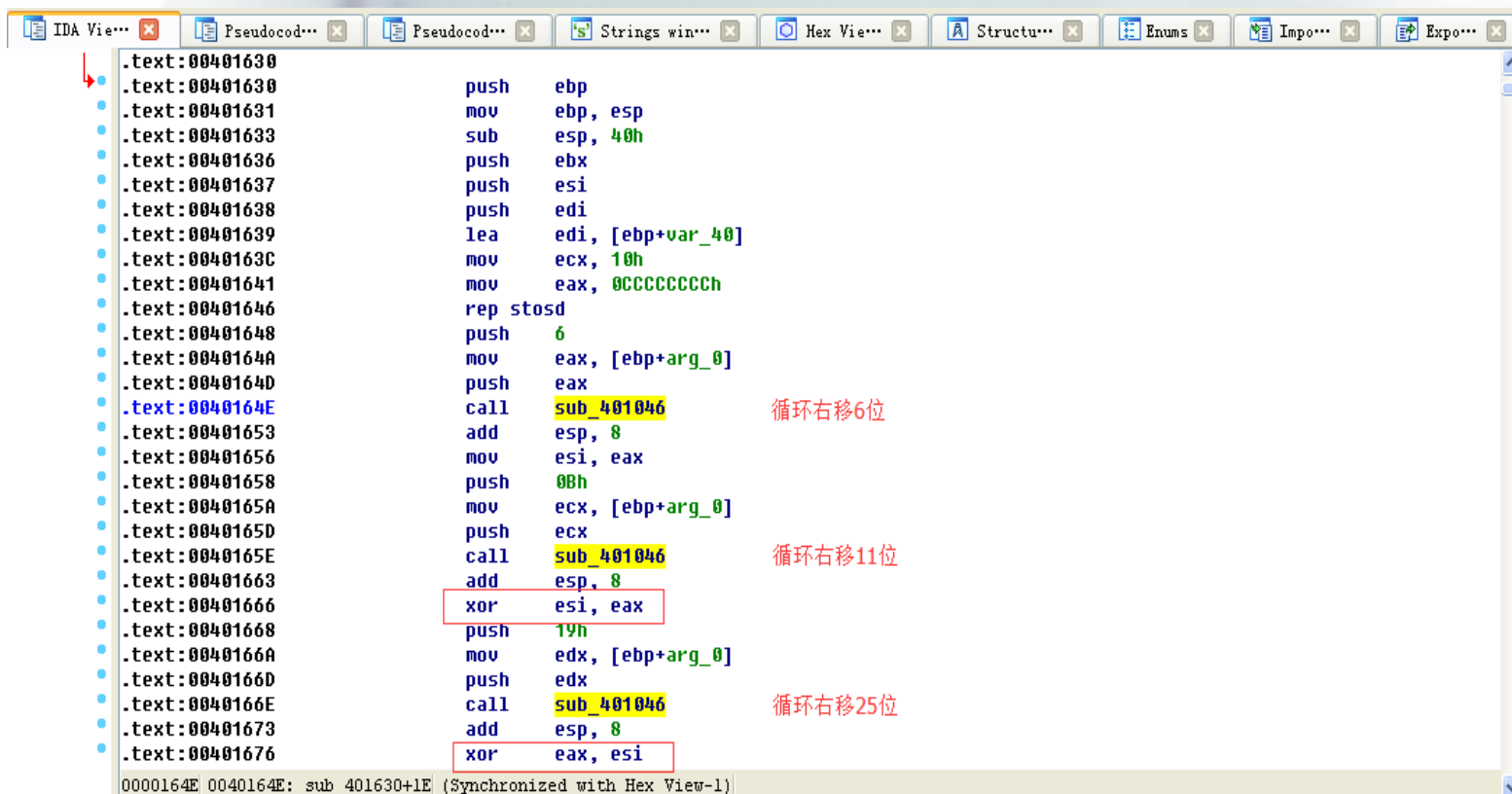


```
.text:00401480 sub_401480 proc near ; CODE XREF: sub_401046fj
.text:00401480
.text:00401480 var_40 = byte ptr -40h
.text:00401480 arg_0 = dword ptr 8
.text:00401480 arg_4 = dword ptr 0Ch
.text:00401480
.text:00401480 push ebp
.text:00401481 mov ebp, esp
.text:00401483 sub esp, 40h
.text:00401486 push ebx
.text:00401487 push esi
.text:00401488 push edi
.text:00401489 lea edi, [ebp+var_40]
.text:0040148C mov ecx, 10h
.text:004014C1 mov eax, 0CCCCCCCCh
.text:004014C6 rep stosd
.text:004014C8 mov eax, [ebp+arg_0]
.text:004014CB mov ecx, [ebp+arg_4] ecx=需要右移的位数
.text:004014CE shr eax, cl eax右移n位
.text:004014D0 mov ecx, 20h ecx=0x20
.text:004014D5 sub ecx, [ebp+arg_4] ecx=0x20-n
.text:004014D8 mov edx, [ebp+arg_0]
.text:004014DB shl edx, cl edx左移(0x20-n)位
.text:004014DD or eax, edx eax=eax | edx
.text:004014DF pop edi
.text:004014E0 pop esi
.text:004014E1 pop ebx
.text:004014E2 mov esp, ebp
.text:004014E4 pop ebp
000014C8 004014C8: sub_401480+18 (Synchronized with Hex View-1)
```



## 2. SHA 算法

❑ 例如sub\_401032()的功能为三个变量分别循环右移n位后再异或



```
.text:00401630
.text:00401630      push    ebp
.text:00401631      mov     ebp, esp
.text:00401633      sub     esp, 40h
.text:00401636      push    ebx
.text:00401637      push    esi
.text:00401638      push    edi
.text:00401639      lea     edi, [ebp+var_40]
.text:0040163C      mov     ecx, 10h
.text:00401641      mov     eax, 0CCCCCCCCh
.text:00401646      rep stosd
.text:00401648      push    6
.text:0040164A      mov     eax, [ebp+arg_0]
.text:0040164D      push    eax
.text:0040164E      call    sub_401046      循环右移6位
.text:00401653      add     esp, 8
.text:00401656      mov     esi, eax
.text:00401658      push    0Bh
.text:0040165A      mov     ecx, [ebp+arg_0]
.text:0040165D      push    ecx
.text:0040165E      call    sub_401046      循环右移11位
.text:00401663      add     esp, 8
.text:00401666      xor     esi, eax
.text:00401668      push    19h
.text:0040166A      mov     edx, [ebp+arg_0]
.text:0040166D      push    edx
.text:0040166E      call    sub_401046      循环右移25位
.text:00401673      add     esp, 8
.text:00401676      xor     eax, esi

0000164E 0040164E: sub_401630+1E (Synchronized with Hex View-1)
```



## 2. SHA 算法

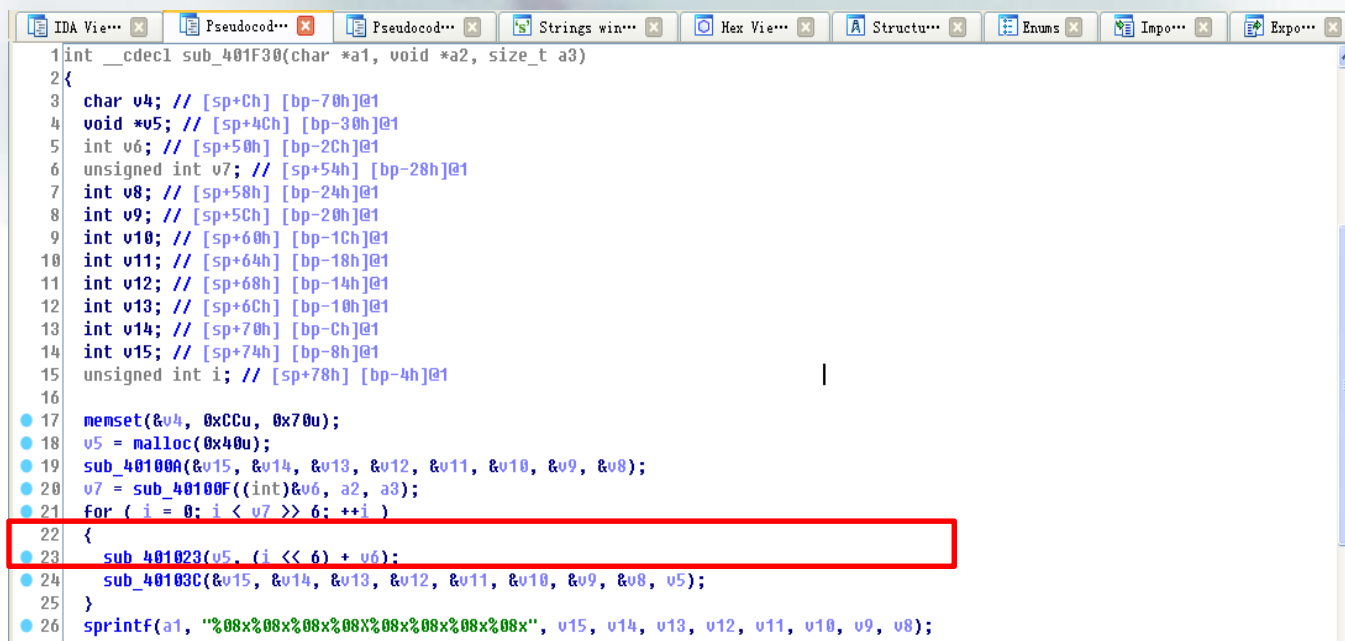
- ❑ 每一轮进行一系列计算后，变量A~H被重新赋值，循环结束后变量A~H与初始值相加，函数返回

```
47 }
48 for ( i = 0; i < 64; ++i )
49 {
50     v12 = sub_401032(e);
51     v13 = v12 + h;
52     v14 = sub_401028(e, f, g);
53     v32 = *(v29 + i) + dword_4280E8[i] + v14 + v13;
54     v15 = sub_40102D(a);
55     v31 = sub_401014(a, b, c) + v15;
56     h = g;
57     g = f;
58     f = e;
59     e = v32 + d;
60     d = c;
61     c = b;
62     b = a;
63     a = v31 + v32;
64 }
65 *A += a;
66 *B += b;
67 *C += c;
68 *D += d;
69 *E += e;
70 *F += f;
71 *G += g;
72 result = H;
73 *H += h;
74 return result;
75 }
```

00001DA2 sub\_401E70:60

## 2. SHA 算法

- ❑ `sub_401005`函数的for循环结束后，变量A~H链接并以十六进制的格式(“%08x%08x%08x%08X%08x%08x%08x%08x”)写入到a1，注意第4个为%08X。
- ❑ 注意：第4个为%08X，即大写输出，这是一个混淆项，写解题程序时需注意



```
1 int __cdecl sub_401F30(char *a1, void *a2, size_t a3)
2 {
3     char v4; // [sp+Ch] [bp-70h]@1
4     void *v5; // [sp+4Ch] [bp-30h]@1
5     int v6; // [sp+50h] [bp-2Ch]@1
6     unsigned int v7; // [sp+54h] [bp-28h]@1
7     int v8; // [sp+58h] [bp-24h]@1
8     int v9; // [sp+5Ch] [bp-20h]@1
9     int v10; // [sp+60h] [bp-1Ch]@1
10    int v11; // [sp+64h] [bp-18h]@1
11    int v12; // [sp+68h] [bp-14h]@1
12    int v13; // [sp+6Ch] [bp-10h]@1
13    int v14; // [sp+70h] [bp-Ch]@1
14    int v15; // [sp+74h] [bp-8h]@1
15    unsigned int i; // [sp+78h] [bp-4h]@1
16
17    memset(&v4, 0xCCu, 0x70u);
18    v5 = malloc(0x40u);
19    sub_40100A(&v15, &v14, &v13, &v12, &v11, &v10, &v9, &v8);
20    v7 = sub_40100F((int)&v6, a2, a3);
21    for ( i = 0; i < v7 >> 6; ++i )
22    {
23        sub_401023(u5, (i << 6) + u6);
24        sub_40103C(&v15, &v14, &v13, &v12, &v11, &v10, &v9, &v8, u5);
25    }
26    sprintf(a1, "%08x%08x%08x%08x%08X%08x%08x%08x", v15, v14, v13, v12, v11, v10, v9, v8);
```



## 2. SHA 算法

@分析函数调用

@main()

□sub\_401005() **SHA256函数**

○sub\_401F30()

- ❖(1)sub\_40100A() 对A~H 8个变量进行赋值
- ❖(2) sub\_40100F() 数据填充
- ❖(3) sub\_401023() 将char类型的输入转化为整型(int)
- ❖(4) sub\_40103C() 64轮步函数的运算规则



## 2. SHA 算法

④ 猜想该程序的核心算法为**SHA-256**散列算法

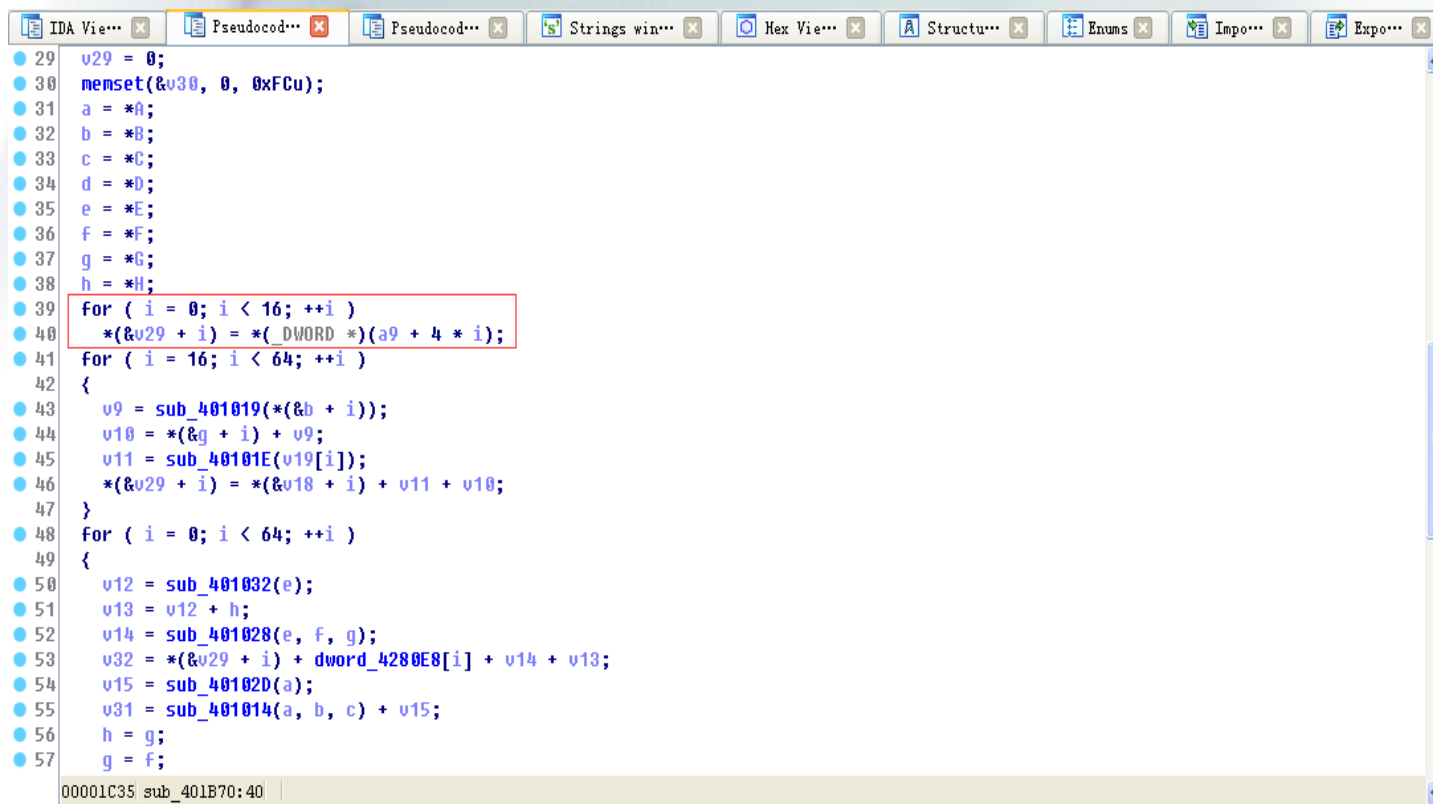
□ 根据

- 数据填充规则
- 变量A~H的初始值
- 算法的输出长度为**32字节(8个整型变量、256位)**
- **64**轮步函数的运算规则



## 2. SHA 算法

❑ 为了印证我们的猜想，可以进一步分析轮函数 **sub\_40103C**，在**SHA-256**算法的步函数中，前**16**个消息字的产生为明文的**16**个子分组



```
29  v29 = 0;
30  memset(&v30, 0, 0xFCu);
31  a = *0;
32  b = *0;
33  c = *0;
34  d = *0;
35  e = *E;
36  f = *F;
37  g = *G;
38  h = *H;
39  For ( i = 0; i < 16; ++i )
40      *(&v29 + i) = *(_DWORD *) (a9 + 4 * i);
41  For ( i = 16; i < 64; ++i )
42  {
43      v9 = sub_401019(*(&b + i));
44      v10 = *(&g + i) + v9;
45      v11 = sub_40101E(v19[i]);
46      *(&v29 + i) = *(&v18 + i) + v11 + v10;
47  }
48  For ( i = 0; i < 64; ++i )
49  {
50      v12 = sub_401032(e);
51      v13 = v12 + h;
52      v14 = sub_401028(e, f, g);
53      v32 = *(&v29 + i) + dword_4280E8[i] + v14 + v13;
54      v15 = sub_40102D(a);
55      v31 = sub_401014(a, b, c) + v15;
56      h = g;
57      g = f;
```

00001C35 sub\_401B70:40





## 2. SHA 算法

❑ 第17~64个消息字的产生，与sub\_40103C()的第2个for循环相对应

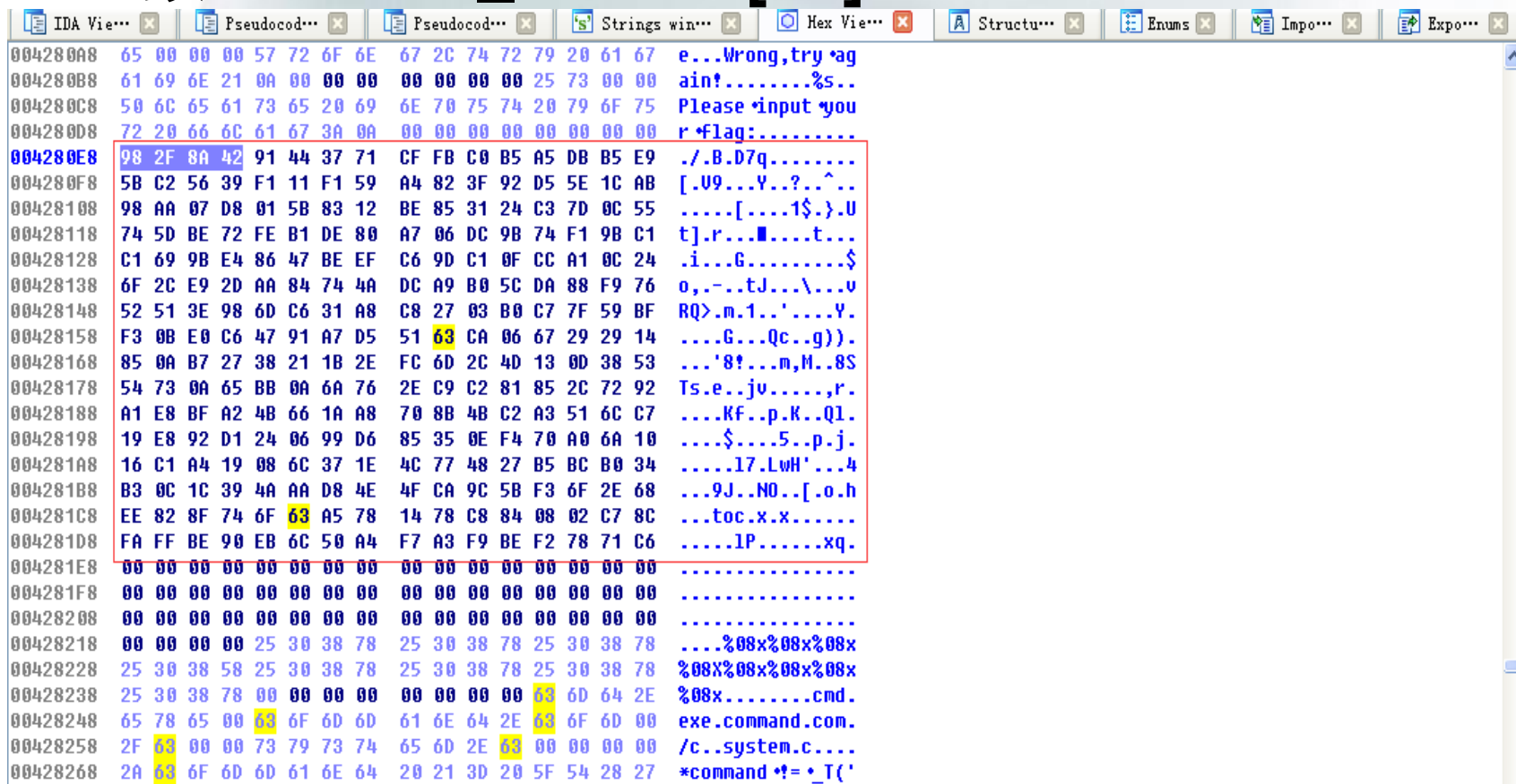
```
31 a = *A;  
32 b = *B;  
33 c = *C;  
34 d = *D;  
35 e = *E;  
36 f = *F;      W[i] (bp - 108h) *(&v29+i)  
37 g = *G;  
38 h = *H;  
39 for ( i = 0; i < 16; ++i )  
40   *(&v29 + i) = *(DWORD *) (a9 + 4 * i);  
41 for ( i = 16; i < 64; ++i )  
42 {  
43   v9 = sub_401019(*(&b + i));  
44   v10 = *(&g + i) + v9;  
45   v11 = sub_40101E(v19[i]);  
46   *(&v29 + i) = *(&v18 + i) + v11 + v10;  
47 }  
48 for ( i = 0; i < 64; ++i )  
49 {  
50   v12 = sub_401032(e);  
51   v13 = v12 + h;  
52   v14 = sub_401028(e, f, g);  
53   v32 = *(&v29 + i) + dword_4280E8[i] + v14 + v13;  
54   v15 = sub_40102D(a);  
55   v31 = sub_401014(a, b, c) + v15;  
56   h = g;  
57   g = f;  
58   f = e;  
59   e = v32 + d;
```

W[i - 2] (bp - 110h) = W[i] - 8 = W[i - 2]      \*(&b + i)  
W[i - 7] (bp - 124h) = W[i] - 28 = W[i - 7]      \*(&g + i)  
W[i - 15] (bp - 144h) = W[i] - 60 = W[i - 15]      v19[i]  
W[i - 16] (bp - 148h) = W[i] - 64 = W[i - 16]      \*(&v18 + i)

00001CBB sub\_401B70:48

## 2. SHA 算法

❑ SHA-256算法使用了64个32位字长的常数，即数组dword\_4280E8[64]



```
004280A8 65 00 00 00 57 72 6F 6E 67 2C 74 72 79 20 61 67 e...Wrong,try *ag
004280B8 61 69 6E 21 0A 00 00 00 00 00 00 00 25 73 00 00 ain!.....%s..
004280C8 50 6C 65 61 73 65 20 69 6E 70 75 74 20 79 6F 75 Please input you
004280D8 72 20 66 6C 61 67 3A 0A 00 00 00 00 00 00 00 00 r *flag:.....
004280E8 98 2F 8A 42 91 44 37 71 CF FB C0 B5 A5 DB B5 E9 ./..B.D7q.....
004280F8 5B C2 56 39 F1 11 F1 59 A4 82 3F 92 D5 5E 1C AB [.U9...Y...?..^..
00428108 98 AA 07 D8 01 5B 83 12 BE 85 31 24 C3 7D 0C 55 .....[....1$.}.U
00428118 74 5D BE 72 FE B1 DE 80 A7 06 DC 9B 74 F1 9B C1 t].r...■...t...
00428128 C1 69 9B E4 86 47 BE EF C6 9D C1 0F CC A1 0C 24 .i...G.....$
00428138 6F 2C E9 2D AA 84 74 4A DC A9 B0 5C DA 88 F9 76 o,-..tJ...\....v
00428148 52 51 3E 98 6D C6 31 A8 C8 27 03 B0 C7 7F 59 BF RQ>.m.1...'....Y.
00428158 F3 0B E0 C6 47 91 A7 D5 51 63 CA 06 67 29 29 14 ...G...Qc..g)).
00428168 85 0A B7 27 38 21 1B 2E FC 6D 2C 4D 13 0D 38 53 ...'8?...m,M..8S
00428178 54 73 0A 65 BB 0A 6A 76 2E C9 C2 81 85 2C 72 92 Ts.e..ju.....,r.
00428188 A1 E8 BF A2 4B 66 1A A8 70 8B 4B C2 A3 51 6C C7 ....KF..p.K..Q1.
00428198 19 E8 92 D1 24 06 99 D6 85 35 0E F4 70 A0 6A 10 ....$.5..p.j.
004281A8 16 C1 A4 19 08 6C 37 1E 4C 77 48 27 B5 BC B0 34 .....17.LwH'...4
004281B8 B3 0C 1C 39 4A AA D8 4E 4F CA 9C 5B F3 6F 2E 68 ...9J..N0..[.o.h
004281C8 EE 82 8F 74 6F 63 A5 78 14 78 C8 84 08 02 C7 8C ...toc.x.x.....
004281D8 FA FF BE 90 EB 6C 50 A4 F7 A3 F9 BE F2 78 71 C6 .....1P.....xq.
004281E8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
004281F8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00428208 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00428218 00 00 00 00 25 30 38 78 25 30 38 78 25 30 38 78 ....%08x%08x%08x
00428228 25 30 38 58 25 30 38 78 25 30 38 78 25 30 38 78 %08X%08x%08x%08x
00428238 25 30 38 78 00 00 00 00 00 00 00 00 6D 64 2E %08x.....cmd.
00428248 65 78 65 00 63 6F 6D 6D 61 6E 64 2E 63 6F 6D 00 exe.command.com.
00428258 2F 63 00 00 73 79 73 74 65 6D 2E 63 00 00 00 00 /c..system.c....
00428268 2A 63 6F 6D 6D 61 6E 64 20 21 3D 20 5F 54 28 27 *command *!= *_T('
```

## 2. SHA 算法

### @分析函数调用

### @main()

#### □sub\_401005() SHA256函数

#### ○sub\_401F30()

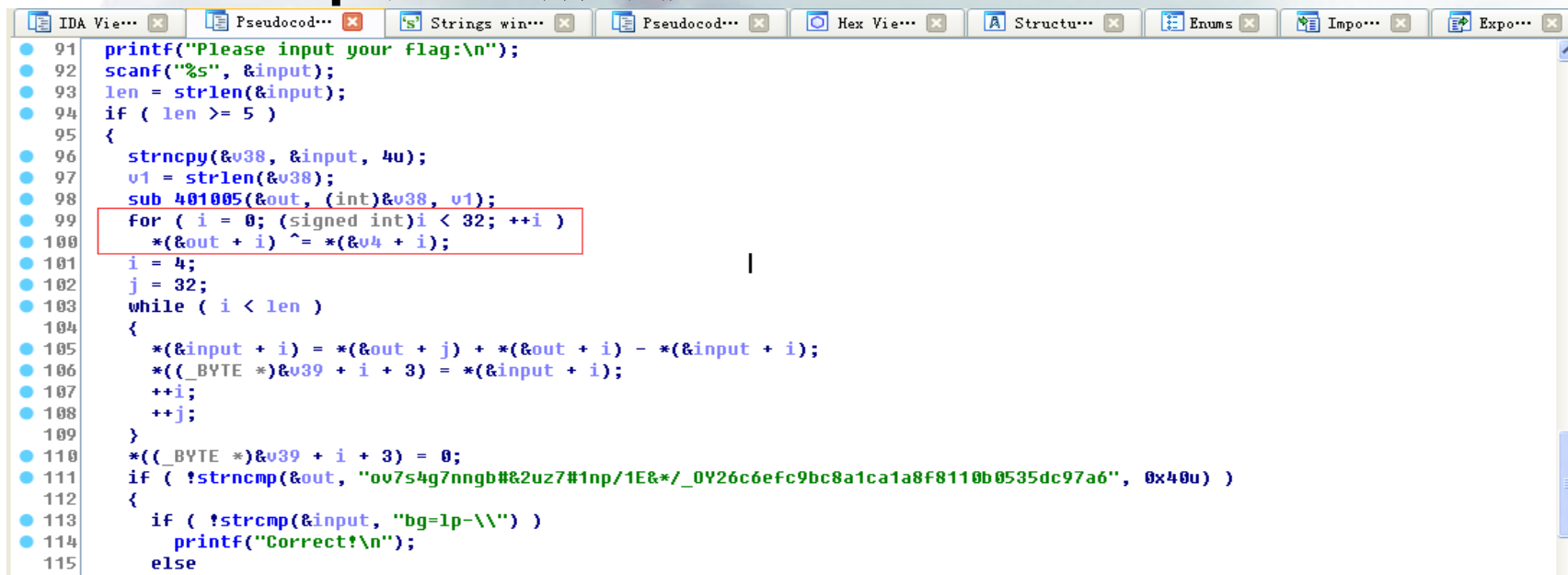
- ❖(1)sub\_40100A() 对A~H 8个变量进行赋值
- ❖(2) sub\_40100F() 数据填充
- ❖(3) sub\_401023() 将char类型的输入转化为整型(int)
- ❖(4) sub\_40103C() 64轮步函数的运算规则

#### ○分析函数sub\_401005()之后的处理流程，以写出解题代码



## 2. SHA 算法

- ❑ 接下来分析函数 `sub_401005()` 之后的处理流程，这中间加了多个混淆项
- ❑ 首先第一个混淆项，是对 `sub_401005()` 函数的第一个参数，即输出的哈希值 `out`，进行逐字节进行异或操作后赋值给 `out`。`out` 被混淆后给后面参与的第一个 `strcmp` 增加破解难度。



```
91 printf("Please input your flag:\n");
92 scanf("%s", &input);
93 len = strlen(&input);
94 if ( len >= 5 )
95 {
96     strncpy(&v38, &input, 4u);
97     v1 = strlen(&v38);
98     sub_401005(&out, (int)&v38, v1);
99     for ( i = 0; (signed int)i < 32; ++i )
100         *(&out + i) ^= *(&v4 + i);
101     i = 4;
102     j = 32;
103     while ( i < len )
104     {
105         *(&input + i) = *(&out + j) + *(&out + i) - *(&input + i);
106         *((_BYTE *)&v39 + i + 3) = *(&input + i);
107         ++i;
108         ++j;
109     }
110     *((_BYTE *)&v39 + i + 3) = 0;
111     if ( !strcmp(&out, "ov7s4g7nngb#&2uz7#1np/1E&*/_0Y26c6efc9bc8a1ca1a8f8110b0535dc97a6", 0x40u) )
112     {
113         if ( !strcmp(&input, "bg=lp-\\") )
114             printf("Correct!\n");
115         else
```

## 2. SHA 算法

□参与异或运算的变量即v4~v35共32个整理得到

```
char key[32]=
```

```
{0x0D,0x13,0x04,0x11,0x02,0x01,0x03,0x0D,  
0x0C,0x02,0x04,0x12,0x11,0x06,0x14,0x1F,  
0x07,0x16,0x09,0x0F,0x15,0x19,0x03,0x26,  
0x13,0x1E,0x1E,0x1A,0x0D,0x1A,0x02,0x04};
```



## 2. SHA 算法

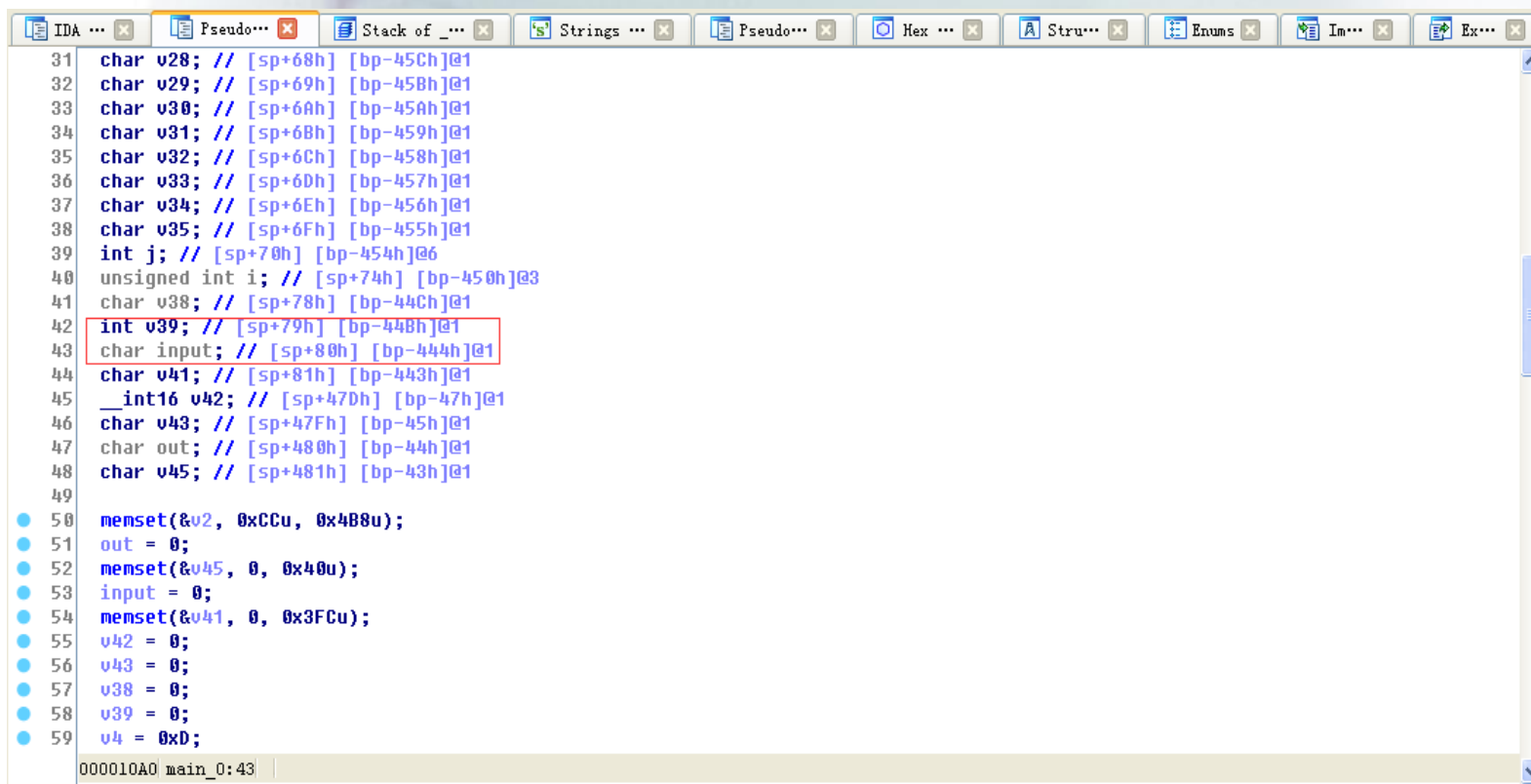
- 然后第二个混淆项，是对输入input进行逐字节处理。input[i](从下标4开始，即未参与散列值计算的前四个字节后部分)与输出out[j](从下标32开始)、out[i]进行加法运算，再写入到input[i-4]（注：相当于对前四个字节后的input输入，通过与out进行运算，实现混淆。混淆后等于“bg=lp-\\”）



```
97  v1 = strlen(&v38);
98  sub_401005(&out, (int)&v38, v1);
99  for ( i = 0; (signed int)i < 32; ++i )
100    *(&out + i) ^= *(&v4 + i);
101  i = 4;
102  j = 32;
103  while ( i < len )
104  {
105    *(&input + i) = *(&out + j) + *(&out + i) - *(&input + i);
106    *((_BYTE *)&v39 + i + 3) = *(&input + i);
107    ++i;
108    ++j;
109  }
110  *((_BYTE *)&v39 + i + 3) = 0;
111  if ( !strcmp(&out, "ov7s4g7nngb#&2uz7#1np/1E&*/_0Y26c6efc9bc8a1ca1a8f8110b0535dc97a6", 0x40u) )
112  {
113    if ( !strcmp(&input, "bg=lp-\\") )
114      printf("Correct!\n");
115    else
116      printf("Almost correct,try again!\n");
117  }
118  else
119  {
```

## 2. SHA 算法

- ❑ 因为v39是[bp-44B], input是[bp-444]
- ❑ 所以 $\&\text{v39} = \&\text{input} - 7$ , 因此把 $\&\text{input} - 7$ 代入到 $\text{*(\&\text{v39} + i + 3)}$ , 则 $\text{*(\&\text{v39} + i + 3)} = \text{input}[i - 4]$ .

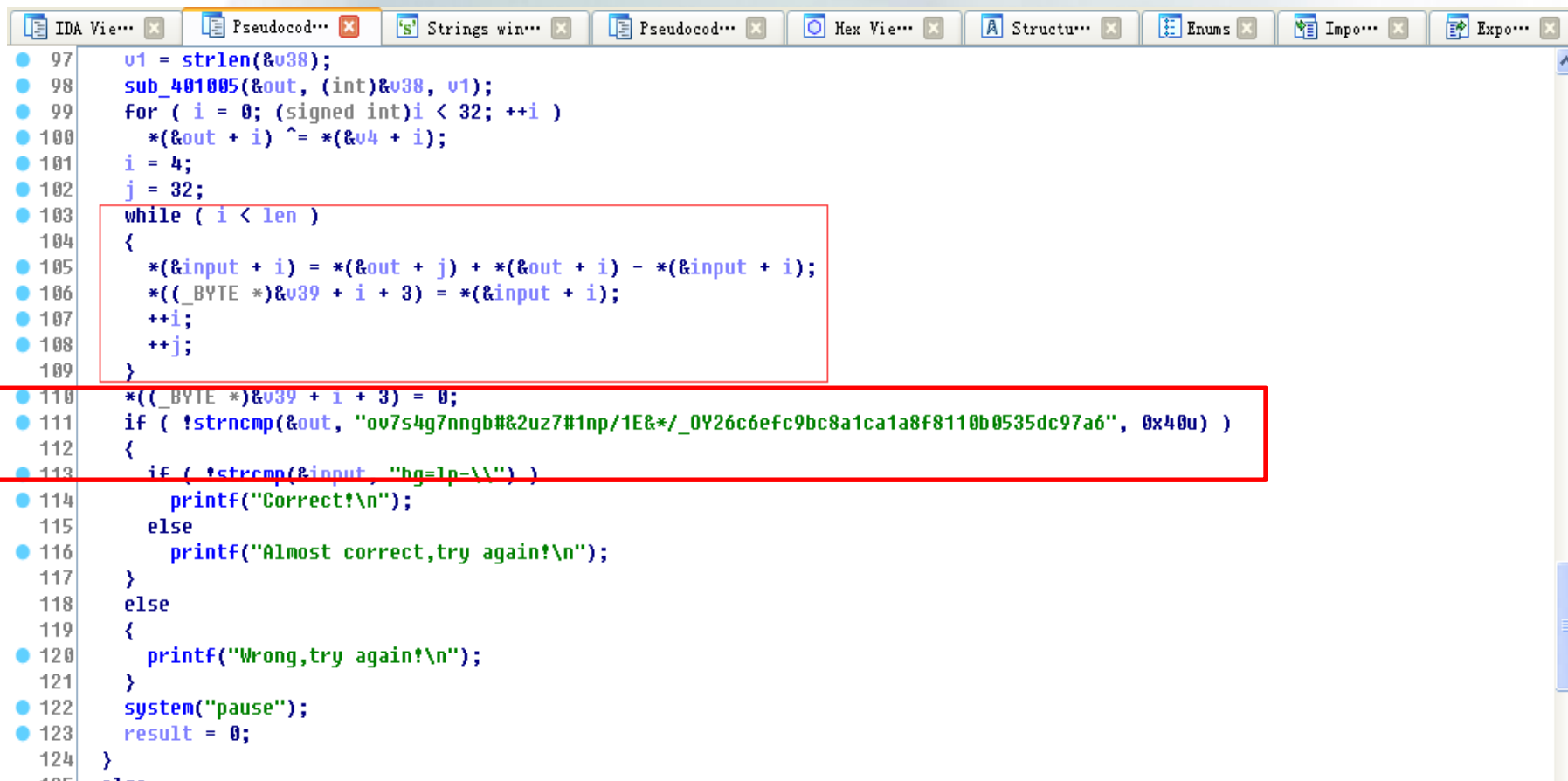


```
31 char v28; // [sp+68h] [bp-45Ch]@1
32 char v29; // [sp+69h] [bp-458h]@1
33 char v30; // [sp+6Ah] [bp-450h]@1
34 char v31; // [sp+6Bh] [bp-449h]@1
35 char v32; // [sp+6Ch] [bp-458h]@1
36 char v33; // [sp+6Dh] [bp-457h]@1
37 char v34; // [sp+6Eh] [bp-456h]@1
38 char v35; // [sp+6Fh] [bp-455h]@1
39 int j; // [sp+70h] [bp-454h]@6
40 unsigned int i; // [sp+74h] [bp-450h]@3
41 char v38; // [sp+78h] [bp-44Ch]@1
42 int v39; // [sp+79h] [bp-44Bh]@1
43 char input; // [sp+80h] [bp-444h]@1
44 char v41; // [sp+81h] [bp-443h]@1
45 __int16 v42; // [sp+470h] [bp-47h]@1
46 char v43; // [sp+47Fh] [bp-45h]@1
47 char out; // [sp+480h] [bp-44h]@1
48 char v45; // [sp+481h] [bp-43h]@1
49
50 memset(&v2, 0xCCu, 0x4B8u);
51 out = 0;
52 memset(&v45, 0, 0x40u);
53 input = 0;
54 memset(&v41, 0, 0x3FCu);
55 v42 = 0;
56 v43 = 0;
57 v38 = 0;
58 v39 = 0;
59 v4 = 0x0;
```

000010A0 main\_0:43

## 2. SHA 算法

❑最后，再调用函数**strncmp()**进行字符串比较



```
97  v1 = strlen(&v38);
98  sub_401005(&out, (int)&v38, v1);
99  for ( i = 0; (signed int)i < 32; ++i )
100    *(&out + i) ^= *(&v4 + i);
101  i = 4;
102  j = 32;
103  while ( i < len )
104  {
105    *(&input + i) = *(&out + j) + *(&out + i) - *(&input + i);
106    *((_BYTE *)&v39 + i + 3) = *(&input + i);
107    ++i;
108    ++j;
109  }
110  *((_BYTE *)&v39 + i + 3) = 0;
111  if ( !strcmp(&out, "ov7s4g7nngb#&2uz7#1np/1E&*/_0Y26c6efc9bc8a1ca1a8f8110b0535dc97a6", 0x40u) )
112  {
113    if ( !strcmp(&input, "hg=lp-\\") )
114      printf("Correct!\n");
115    else
116      printf("Almost correct,try again!\n");
117  }
118  else
119  {
120    printf("Wrong,try again!\n");
121  }
122  system("pause");
123  result = 0;
124 }
```



## 2. SHA 算法

- ❑ 第一个比较为散列值经过混淆后与字符串  
“**ov7s4g7nngb#&2uz7#1np/1E&\*/\_OY26c6  
efc9bc8a1ca1a8f8110b0535dc97a6**” 比较，  
由于参与散列值计算的字符串长度只有4个字  
节，可以通过暴力破解的方式得到



## 2. SHA 算法

- ❑ 第二个比较为，输入值input经过while循环取4个字节之后的值，并经过与out[]运算后，再与字符串“bg=lp-\\”进行比较，这里的输入指的是未参与散列值计算的输入部分，由于前面对于输入的运算是逐字节运算，可以推测出输入的长度为 $4 + \text{strlen}(\text{"bg=lp-\\"}) = 11$ 字节。
- ❑ 注意：在比较前，要考虑根据两个干扰项生成out后，再进行比较



## 2. SHA 算法

### @练习

- ❑ 写出SHA\_256.exe解题的脚本
- ❑ 解出flag
- ❑ 说明：可调用python的itertools库进行4字节的排列组合；可调用hashlib库计算SHA\_256的哈希值。



# MD5.EXE

```
ubuntu@ubuntu:~/Documents$ python md5.py  
Flag: Hash  
ubuntu@ubuntu:~/Documents$
```

A screenshot of a Windows application window titled "C:\Documents and Settings\Administrator\桌面\reversing\reverse\MD5.exe". The window has a black background with white text. It prompts the user to "Please input your flag:", the user has entered "Hash123", and the application responds with "Correct!". Below this, it says "请按任意键继续. . .".

```
C:\Documents and Settings\Administrator\桌面\reversing\reverse\MD5.exe  
Please input your flag:  
Hash123  
Correct!  
请按任意键继续. . .
```



# SHA\_256.exe

```
ubuntu@ubuntu:~/Documents$ python sha.py  
flag_1 sha2  
Flag: sha256_hash  
ubuntu@ubuntu:~/Documents$
```

```
C:\Documents and Settings\Administrator\桌面\reversing\reverse\SHA_256...  
Please input your flag:  
sha256_hash  
Correct!  
请按任意键继续. . .
```



# 第五章 常见加密算法逆向分析

- @5.1 简单加密算法逆向分析
- @5.2 对称加密算法逆向分析
- @5.3 单向散列算法逆向分析
- @5.4 其他算法逆向分析



## 5.4 其他算法逆向分析

### @Base64算法

□ **Base64**是一种基于**64**个可打印字符来表示二进制数据的表示方法。

- 由于**2**的**6**次方等于**64**，所以每**6**个比特为一个单元，对应某个可打印字符。
- 三个字节有**24**个比特，对应于**4**个**6**比特**Base64**单元，即**3**个字节**24**个比特可表示**4**个可打印字符。
- 在**Base64**中的可打印字符包括字母**A-Z**、**a-z**、数字**0-9**，这样共有**62**个字符，另外两个可打印符号在不同的系统中而不同。



## 5.4 其他算法逆向分析

### @Base64算法

- Base64常用于在通常处理文本数据的场合，表示、传输、存储一些二进制数据。包括MIME的email、在XML中存储复杂数据





# Base64算法

- ❑ (1) 算法原理
- ❑ (2) 逆向分析



# Base64算法

## □ 数据转换为Base64

- 将3个字节的数据先后放入一个24比特的缓冲区中，第一个字节占高位，然后每次取缓冲区的6比特，按照其值选择

“**ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/'**” 中的字符作为编码后的输出

- 如果要编码的字节数不能被3整除，最后会多出1个或2个字节，使用下面的方法进行处理：

- ❖ 先使用0补足24位缓冲区，然后再进行编码，在编码后的base64文本后加一个或两个“=”号，代表补足的字节数。以上过程不断进行，直到全部输入数据转换完成



# Q & A

---

谢谢!

