



北京邮电大学

Beijing University of Posts and Telecommunications

分 析 报 告

题	目	DES 逆向分析
姓	名	李昊伦
学	号	2023211595
班	级	2023211805
指 导 教 师		付俊松

目 录

1 DES 算法概述.....	3
2 DES 算法流程图.....	3
2.1 DES 算法的整体结构	3
2.2 DES 算法的轮函数	5
2.3 DES 算法的密钥编排算法	7
3 DES 逆向分析.....	8
3.1 Main 函数	8
3.2 get_subkey 函数	14
3.2.1 get_subkey 主函数	14
3.2.2 shift_left 函数	18
3.2.3 pc2_replace 函数	21
3.3 encryption 函数.....	22
3.3.1 encryption 主函数	22
3.3.2 byte2Bit 函数	27
3.3.3 ip_replace 函数.....	28
3.3.4 f_func 函数.....	30
3.3.5 byteXOR 函数	32
3.3.6 fp_replac 函数	34
4 DES 解密与总结.....	35
4.1 解密结果	35
4.2 总结	36

1 DES 算法概述

DES 算法为密码体制中的对称密码体制，又被称为美国数据加密标准。DES 是一个分组加密算法，典型的 DES 以 64 位为分组对数据加密，加密和解密用的是同一个算法。

DES 密钥长 64 位，密钥事实上是 56 位参与 DES 运算（第 8、16、24、32、40、48、56、64 位是校验位，使得每个密钥都有奇数个 1），分组后的明文组和 56 位的密钥按位替代或交换的方法形成密文组。

2 DES 算法流程图

DES 算法的主要流程如下图所示，接下来我将按照流程依次介绍每个模块并对其进行逆向分析。

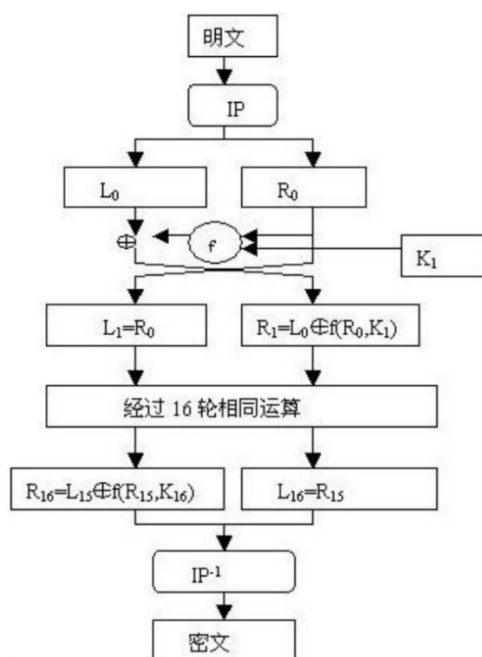


图 1 DES 算法流程图

2.1 DES 算法的整体结构

2.1.1 IP 置换

给定明文，通过一个固定的初始置换 IP 来重排输入明文块 P 中的比特，得到比特串 $P_0 = IP(P) = L_0R_0$ ，这里 L₀ 和 R₀ 分别是 P₀ 的前 32 比特和后 32 比特。

IP							
58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

图2 初始置换 IP

2.1.2 16 轮迭代

Feistel 结构：按下述规则进行 16 次迭代，即 $1 \leq i \leq 16$ 。

$$L_i = R_{i-1}, R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$$

这里是对应比特的模 2 加， f 是一个函数（称为轮函数）；16 个长度为 48 比特的子密钥 $K_i (1 \leq i \leq 16)$ 是由密钥 k 经密钥编排函数计算出来的。

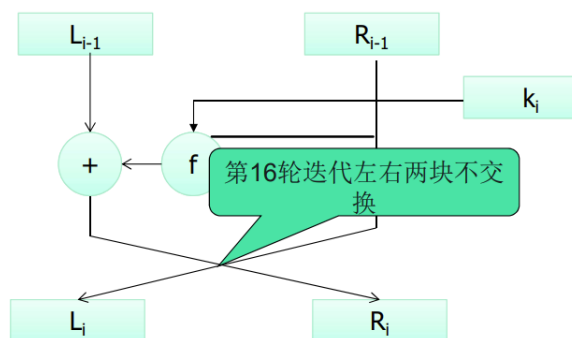


图3 16 轮迭代图

2.1.3 IP 逆置换

对比特串 $R_{16}L_{16}$ 使用逆置换 IP^{-1} 得到密文 C ，即 $C = IP^{-1}(R_{16}L_{16})$ 。（注意 L_{16} 和 R_{16} 的相反顺序）

IP^{-1}							
40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

图4 初始置换的逆置换 IP^{-1}

2.2 DES 算法的轮函数

在 16 次迭代中，函数 f 以长度为 32 比特串 R_{i-1} 作为第一输入，以长度为 48 比特串 K_i 作为第二个输入，产生长度为 32 比特的输出。

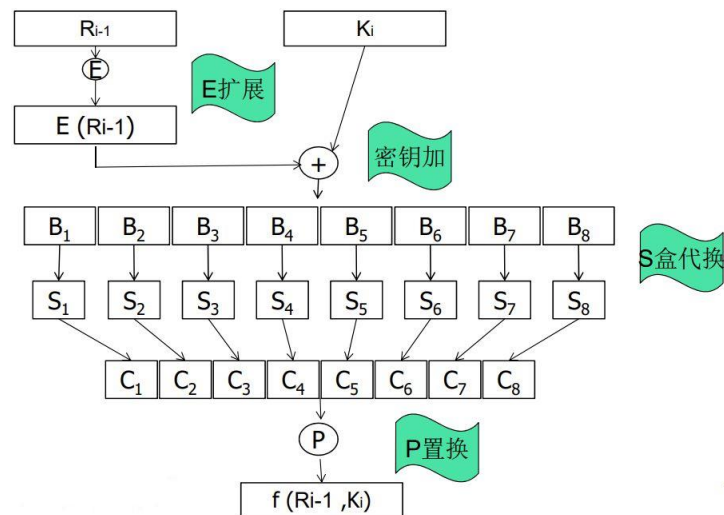


图 5 DES 轮函数示意图

2.2.1 E 扩展

R_{i-1} 根据扩展规则扩展为 48 比特长度的串。

扩展置换目标是 IP 置换后获得的右半部分 R_0 ，将 32 位输入扩展为 48 位 (分为 4 位 x 8 组) 输出。

扩展置换目的有两个：生成与密钥相同长度的数据以进行异或运算；提供更长的结果，在后续的替代运算中可以进行压缩。

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

图 6 E 扩展示意图

2.2.2 密钥加

计算 $E(R_{i-1}) \oplus K_i$ ，并将结果写成 8 个比特串，每个 6 比特，
 $B = B_1 B_2 B_3 B_4 B_5 B_6 B_7 B_8$

2.2.3 S 盒代换

使用 8 个 S 盒 S_1, \dots, S_8 . 每个 S_i 是一个固定的 4×16 阶矩阵, 其元素取 0~15 之间的整数. 给定长度为 6 的比特串, 如 $B_j = b_1b_2b_3b_4b_5b_6$, $S_j(B_j)$ 计算如下:

- (1) b_1b_6 两个比特确定了 S_j 的行 r 的二进制表示 ($0 \leq r \leq 3$)
- (2) $b_2b_3b_4b_5$ 四个比特确定了 S_j 的列 c 的二进制表示 ($0 \leq c \leq 15$)
- (3) $S_j(B_j)$ 定义成长度为 4 的比特串的值 $S_j(r, c)$ 。由此可以算出 $C_j = S_j(B_j)$, $1 \leq j \leq 8$ 。

S ₁															
14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
S ₂															
15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
S ₃															
10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
S ₄															
7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
12	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14
S ₅															
2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3
S ₆															
12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13
S ₇															
4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
S ₈															
13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

图 7 S 盒表

例如, 假设 S 盒 8 的输入为 110011, 第 1 位和第 6 位组合为 11, 对应于 S 盒 8 的第 3 行; 第 2 位到第 5 位为 1001, 对应于 S 盒 8 的第 9 列。S 盒 8 的第 3 行第 9 列的数字为 12, 因此用 1100 来代替 110011。注意, S 盒的行列计数都是从 0 开始。

代替过程产生 8 个 4 位的分组, 组合在一起形成 32 位数据。

S 盒代替时 DES 算法的关键步骤, 所有的其他的运算都是线性的, 易于分析, 而 S 盒是非线性的, 相比于其他步骤, 提供了更好安全性。

2.2.4 P 置换

P 置换: 长度为 32 比特串 $C = C_1C_2C_3C_4C_5C_6C_7C_8$, 根据固定置换 $P(*)$ 进行置

换，得到比特串 $P(C)$ 。

P置换			
16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

图 8 P 置换表

2.3 DES 算法的密钥编排算法

给定 64 比特密钥 K ，根据固定的置换 $PC-1$ 来处理 K 得到 $PC-1(K) = C_0D_0$ ，其中 C_0 和 D_0 分别由最前和最后 28 比特组成

计算 $C_i = LS_i(C_{i-1})$ 和 $D_i = LS_i(D_{i-1})$ ，且 $K_i = PC-2(C_iD_i)$ ， LS_i 表示循环左移两个或一个位置，具体地，如果 $i=1,2,9,16$ 就移一个位置，否则就移两个位置， $PC-2$ 是另一个固定的置换。

PC1							
57	49	41	33	25	17	9	
1	58	50	42	34	26	18	
10	2	59	51	43	35	27	
19	11	3	60	52	44	36	
above for C_i ; below for D_i							
63	55	47	39	31	23	15	
7	62	54	46	38	30	22	
14	6	61	53	45	37	29	
21	13	5	28	20	12	4	

PC2					
14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

图 9 密钥生成方案

对 $1 \leq i \leq 16$ ，DES 的每一轮中使用 K 的 56 比特中的 48 个比特，具体选取位置由下表确定。

轮 1															
10	51	34	60	49	17	33	57	2	9	19	42				
3	35	26	25	44	58	59	1	36	27	18	41				
22	28	39	54	37	4	47	30	5	53	23	29				
61	21	38	63	15	20	45	14	13	62	55	31				

轮 2															
2	43	26	52	41	9	25	49	59	1	11	34				
60	27	18	17	36	50	51	58	57	19	10	33				
14	20	31	46	29	63	39	22	28	45	15	21				
53	13	30	55	7	12	37	6	5	54	47	23				

轮 3															
51	27	10	36	25	58	9	33	43	50	60	18				
44	11	2	1	49	34	35	42	41	3	59	17				
61	4	15	30	13	47	23	6	12	29	62	5				
37	28	14	39	54	63	21	53	20	38	31	7				

轮 4															
35	11	59	49	9	42	58	17	27	34	44	2				
57	60	51	50	33	18	19	26	25	52	43	1				
45	55	62	14	28	31	7	53	63	13	46	20				
21	12	61	23	38	47	5	37	4	22	15	54				

轮 5;															
19	60	43	33	58	26	42	1	11	18	57	51				
41	44	35	34	17	2	3	10	9	36	27	50				
29	39	46	61	12	15	54	37	47	28	30	4				
5	63	45	7	22	31	20	21	55	6	62	38				

轮 6															
3	44	27	17	42	10	26	50	60	2	41	35				
25	57	19	18	1	51	52	59	58	49	11	34				
13	23	30	45	63	62	38	21	31	12	14	55				
20	47	29	54	6	15	4	5	39	53	46	22				

轮 7															
52	57	11	1	26	59	10	34	44	51	25	19				
9	41	3	2	50	35	36	43	42	33	60	18				
28	7	14	29	47	46	22	5	15	63	61	39				
4	31	13	38	53	62	55	20	23	37	30	6				

轮 8															
36	41	60	50	10	43	59	18	57	35	9	3				
58	25	52	51	34	19	49	27	26	17	44	2				
12	54	61	13	31	30	6	20	62	47	45	23				
55	15	28	22	37	46	39	4	7	21	14	53				

轮 9												
57	33	52	42	2	35	51	10	49	27	1	60	
50	17	44	43	26	11	41	19	18	9	36	59	
4	46	53	55	23	22	61	12	54	39	37	15	
47	7	20	14	29	38	31	63	62	13	6	45	

轮 13												
58	34	17	43	3	36	52	11	50	57	2	35	
51	18	9	44	27	41	42	49	19	10	1	60	
7	45	20	39	22	21	28	15	53	38	4	14	
46	6	23	13	63	37	30	62	61	47	5	12	

轮 10												
41	17	36	26	51	19	35	59	33	11	50	44	
34	1	57	27	10	60	25	3	2	58	49	43	
55	30	37	20	7	6	45	63	38	23	21	62	
31	34	4	61	13	22	15	47	46	28	53	29	

轮 14												
42	18	1	27	52	49	36	60	34	41	51	9	
35	2	58	57	11	25	26	33	3	59	50	44	
54	29	4	23	6	5	12	62	37	22	55	61	
30	53	7	28	47	21	14	46	45	31	20	63	

轮 11												
25	1	49	10	35	3	19	43	17	60	34	57	
18	50	41	11	59	44	9	52	51	42	33	27	
39	14	21	4	54	53	29	47	22	7	5	46	
15	38	55	45	28	6	62	31	30	12	37	13	

轮 15												
26	2	50	11	36	33	49	44	18	25	35	58	
19	51	42	41	60	9	10	17	52	43	34	57	
38	13	55	7	53	20	63	46	21	6	39	45	
14	37	54	12	31	5	61	30	29	15	4	47	

轮 12												
9	50	33	59	19	52	3	27	1	44	18	41	
2	34	25	60	43	57	58	36	35	26	17	11	
23	61	5	55	38	37	13	31	6	54	20	30	
62	22	39	29	12	53	46	15	14	63	21	28	

轮 16												
18	59	42	3	57	25	41	36	10	17	27	50	
11	43	34	33	52	1	2	9	44	35	26	49	
30	5	47	62	45	12	55	38	13	61	31	37	
6	29	46	4	23	28	53	22	21	7	63	39	

图 10 每轮 K 选取图

3 DES 逆向分析

3.1 Main 函数

```

; Attributes: bp-based frame

; int __fastcall main(int argc, const char **argv, const char **envp)
public main
main proc near

var_40= byte ptr -40h
Str= byte ptr -30h
var_10= byte ptr -10h

push    rbp
mov     rbp, rsp
sub     rsp, 60h
call    __main
mov     rax, 43316E455F334544h
mov     qword ptr [rbp+var_10], rax
mov     [rbp+var_10+8], 0
mov     dword ptr [rbp+var_10+0Ch], 0
lea     rcx, Buffer      ; "give me a string to encrypt:"
call    puts
lea     rax, [rbp+Str]
mov     rdx, rax
lea     rcx, Format      ; "%s"
call    scanf
lea     rax, [rbp+Str]
mov     rcx, rax        ; Str
call    strlen
cmp     rax, 8
jz      short loc_40159D

```

图 11 main 函数主程序

首先进入的是主函数 `main`。`main` 函数的这段程序的主要作用是输出一个提示信息让用户输入字符串，然后计算这个字符串的长度。如果输入的字符串长度恰好为 8，就会跳转到下一段逻辑处理。

函数使用基于 `rbp` 的栈帧管理，在一开始先将当前的 `rbp` 压栈保存原栈帧，再将 `rsp` 的值赋给 `rbp`，以建立新的栈帧。随后使用 `sub rsp, 60h` 分配了 96 字节的栈空间，为局部变量预留位置。接着调用了 `__main`。

然后，程序通过 `mov rax, 43316E455F334544h` 将一个 64 位立即数赋值给 `rax` 寄存器，该值其实是一个十六进制编码的 ASCII 字符串，代表着一段密文或预定义值，它随后被保存到了局部变量区域中，即 `var_10` 所表示的位置，也就是 `rbp-10h` 开始的内存区域。此外，该值之后 8 字节位置被清零，再接着的 4 字节也被设置为 0，这三行指令总共初始化了 `var_10` 所代表的结构，总共占用了 16 字节以上的空间。

接下来，程序使用 `puts` 输出了一行提示信息，内容是“give me a string to encrypt:”，它通过 `lea rcx, Buffer` 载入字符串地址到 `rcx`，然后调用 `puts` 来显示这个提示。之后使用 `lea rax, [rbp+Str]` 把指向 `Str`（即变量 `rbp-30h`）的地址赋给 `rax`，再通过 `rdx = rax` 和 `rcx = Format` 准备好参数，调用 `scanf` 读取用户输入的字符串。`Format` 程序期望从标准输入中读取一个字符串，并存入 `Str` 所代表的内存位置。

读取完输入字符串后，程序再一次把字符串地址放到 `rcx` 中，调用 `strlen` 计算输入字符串的长度。通过 `cmp rax, 8` 比较字符串长度是否为 8 个字符，如果相等，即当 `rax` 的值等于 8 时，程序将跳转到 `loc_40159D` 所标记的代码段继续执行。



```

loc_40159D:
lea     rax, [rbp+var_10]
mov     rcx, rax ; unsigned __int8 *
call    _Z10get_subkeyPh ; get_subkey(uchar *)
lea     rdx, [rbp+var_40] ; unsigned __int8 *
lea     rax, [rbp+Str]
mov     rcx, rax ; unsigned __int8 *
call    _Z10encryptionPhS ; encryption(uchar *,uchar *)
mov     dword ptr [rbp+var_10+0Ch], 0
jmp     short loc_401604

```

图 12 main 函数用户输入的字符串长度恰好为 8 个字符

接下来这段代码继续延续了前面主函数的流程，loc_40159D 开始的代码块是当用户输入的字符串长度恰好为 8 个字符时所执行的逻辑。它主要进行密钥派生和加密操作，涉及两个自定义函数 get_subkey 和 encryption。

整体来看，这段代码体现出一个典型的加密处理流程：先使用固定密钥数据派生出子密钥，然后使用输入数据进行加密，并将结果存储到局部缓冲区中，之后进行一些清理或状态更新，最终进入后续逻辑。

首先，程序通过 lea rax, [rbp+var_10] 加载了之前初始化过的变量 var_10 的地址到 rax 寄存器中，这个变量之前存储了一段预定义的 64 位十六进制数据，可能是加密相关的种子或主密钥结构。随后将这个地址传入 rcx，用于作为参数调用 get_subkey 函数。通过名字，我推测这个函数用于从主密钥或某些初始数据中生成子密钥，返回值可能通过指针方式直接写入传入地址对应的内存区域中，因此 var_10 很可能既用于输入也用于保存输出。

完成子密钥生成后，程序紧接着就是准备加密操作。lea rdx, [rbp+var_40] 是把目标输出缓冲区（用于存储加密结果）的地址加载到 rdx 中；接着 lea rax, [rbp+Str] 获取用户输入字符串的地址，然后将其赋给 rcx，作为第二个参数。此时两个寄存器分别指向明文字符串和输出密文的缓冲区。接着调用 encryption 函数，传入这两个参数。通过名字，我觉得它是对输入字符串进行加密，并将结果写入目标缓冲区。

完成加密之后，程序将变量 var_10 中偏移 0Ch 处的 4 字节设置为 0，这通常用于状态标志位或重置密钥结构中的某个字段，可能是为了清除之前计算中用于辅助处理的标志信息或避免残留状态对后续操作造成干扰。最后，通过一

个无条件跳转 `jmp` 指令，程序跳转至标签 `loc_401604`。

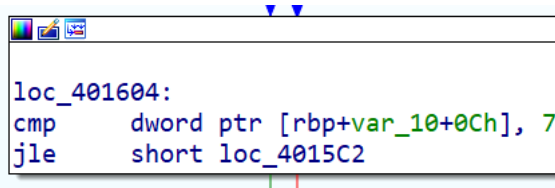


图 13 比较大小关系

程序首先通过 `cmp` 指令比较位于 `rbp+var_10+0Ch` 处的 4 字节整数与常数值 7 的大小关系。这个地址指向的是前面 `var_10` 结构中偏移为 `0Ch` 的部分。`cmp` 指令实际上不会修改任何变量，而只是设置 CPU 的标志位，以便供后续的条件跳转指令使用。接着，程序通过 `jle` 指令判断，如果该值小于或等于 7（也就是 ≤ 7 ），则跳转到 `loc_4015C2` 所指向的位置。

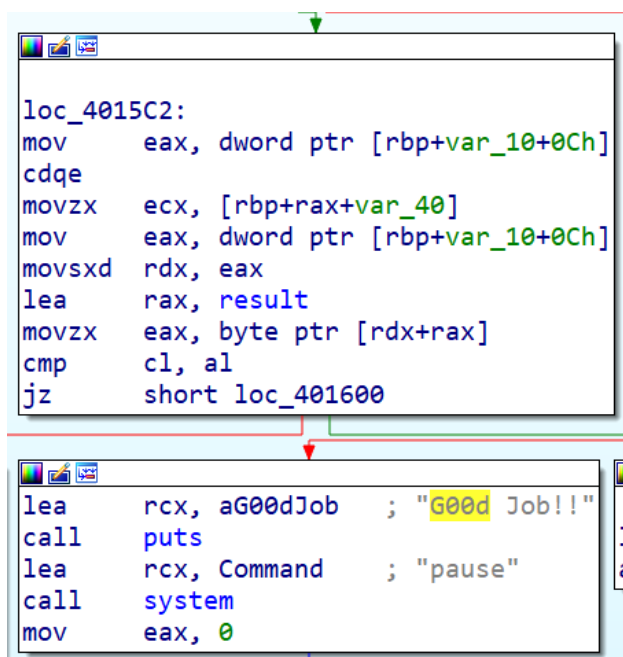


图 14 输入验证的流程

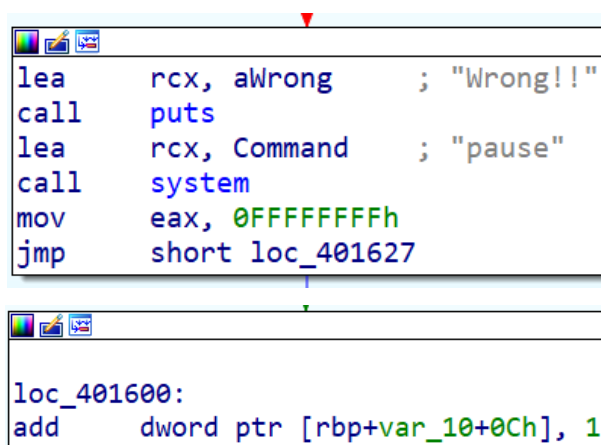
这两段代码继续沿着主函数执行路径，构成了一个完整的逻辑判断与条件性反馈机制。这一系列指令是一种典型的用于输入验证的流程：程序读取用户输入，进行加密，然后逐字节地将加密结果与某个预定义的目标密文进行比对。如果每一字节都匹配，就输出成功信息并结束程序；否则就跳出检查逻辑，可能返回错误或不予反馈。

第一段程序从标签 `loc_4015C2` 开始执行。首先，它从局部变量 `var_10` 中偏移 `0Ch` 的位置读取一个 4 字节整数到 `eax` 寄存器中，这个值我们之前提到是一个计数器或者索引，用于跟踪当前正在处理的步骤。接着使用 `cdqe` 指令将 `eax` 扩展为 64 位 `rax`，这是为了后续作为地址偏移做准备。然后，程序使用 `movzx ecx, [rbp+rax+var_40]`，将 `var_40` 缓冲区中偏移 `rax` 字节处的一个字节零扩展后加载到 `ecx`，这表示它正在逐字节检查加密结果中的某一项，`var_40` 此时是存储加密结果的缓冲区。

紧接着，程序再次读取 `var_10+0Ch` 的值到 `eax`，然后用 `movsxd rdx, eax` 将它符号扩展为 64 位 `rdx`，这个值作为偏移用于接下来的地址运算。接下来，程序通过 `lea rax, result` 得到某个名为 `result` 的全局或静态数组的地址，并在此基础上通过 `[rdx+rax]` 的方式取出数组中对应位置的一个字节，零扩展后放入 `eax`。可以看出，`result` 是一个预定义的正确密文序列或答案数据，而程序正在逐字节地对比加密结果与这个答案。

接着，程序使用 `cmp cl, al` 比较当前加密结果中的字节（`cl`）和答案字节（`al`），如果两者相等，就执行 `jz` 指令跳转到 `loc_401600`，否则将继续往下执行或退出。

第二段汇编程序是发生在所有比对操作成功后、即验证通过时的处理逻辑。程序首先通过 `lea rcx, aG00dJob` 把字符串“G00d Job!!”的地址加载到 `rcx`，然后调用 `puts` 输出它，作为对用户输入正确的肯定反馈。程序将 `eax` 设置为 0，表示正常结束 `main` 函数并返回成功状态。



```
lea    rcx, aWrong      ; "Wrong!!"
call   puts
lea    rcx, Command     ; "pause"
call   system
mov     eax, 0FFFFFFFh
jmp     short loc_401627

loc_401600:
add     dword ptr [rbp+var_10+0Ch], 1
```

图 15 用户输入不正确时的错误反馈逻辑

这两段代码分别对应用户输入不正确时的错误反馈逻辑，以及比对流程中的一个关键控制点，继续沿着前面加密验证流程的结构延伸，类似 if/else 和 for 循环的部分。如果用户输入通过了当前字节的比对，程序就把计数器加一，进入下一轮比对（即类似于循环结构）；如果某一位不匹配，就立即跳到 "Wrong!!" 分支，反馈错误并终止程序。整段流程体现出了一种典型的“逐位校验 + 条件反馈”的加密验证机制，用来判断输入是否能够通过自定义加密逻辑后匹配上预定义密文。成功路径会一路跳转到 "G00d Job!!"，失败路径则中断流程，进入 "Wrong!!" 并返回失败码。

首先来看第一段代码，也就是当加密结果中某个字节与预期答案不匹配时执行的分支。程序先加载一个指向字符串 "Wrong!!" 的地址到 rcx 中，然后调用 puts 向用户表明输入错误。接下来，它加载字符串 "pause" 的地址到 rcx 并调用 system 让终端暂停，防止程序立刻退出，让用户有时间看到错误信息。随后程序将 eax 设置为 0xFFFFFFFF，即-1，用作失败返回码。最后通过 jmp short loc_401627 无条件跳转到标签 loc_401627，这是程序退出或清理资源的地方。

接着来看第二段逻辑，即 loc_401600 标签下的单行指令：add dword ptr [rbp+var_10+0Ch], 1。这行指令表示 var_10 中偏移 0Ch 的字段加 1，这正是前面作为当前索引的那个变量。也就是说，在成功比对当前字节的情况下，程序会来到这里将计数器加一，然后重新跳回到上面的检查逻辑，继续比对下一个字节。这说明程序是在逐个字符地验证用户输入加密后的输出与预设值 result 是否完全一致。

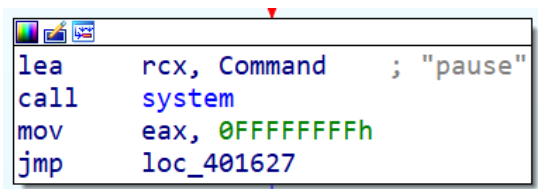


图 16 执行失败路径

这段代码是程序在执行失败路径时的结尾部分，它负责向用户显示错误提示后暂停终端，再设置一个错误返回值并退出函数。

```

loc_401627:
add     rsp, 60h
pop     rbp
retn
main endp

```

图 17 main 收尾

这段代码是函数 main 的收尾部分，它完成清理栈空间和恢复栈帧的工作，属于典型的函数退出逻辑。程序到达 loc_401627 标签时，说明所有逻辑（无论成功、失败或中途退出）都已经执行完毕。

3.2 get_subkey 函数

3.2.1 get_subkey 主函数

```

; Attributes: bp-based frame

; __int64 __fastcall get_subkey(unsigned __int8 *)
public _Z10get_subkeyPh
_Z10get_subkeyPh proc near

var_C0= byte ptr -0C0h
var_B8= qword ptr -0B8h
var_B0= qword ptr -0B0h
var_A8= qword ptr -0A8h
var_A0= qword ptr -0A0h
var_98= qword ptr -98h
var_90= byte ptr -90h
var_50= byte ptr -50h
var_4= dword ptr -4
arg_0= qword ptr 10h

push    rbp
mov     rbp, rsp
sub     rsp, 0E0h
mov     [rbp+arg_0], rcx
lea     rax, [rbp+var_50]
mov     r8d, 8 ; int
mov     rdx, rax ; unsigned __int8 *
mov     rcx, [rbp+arg_0] ; unsigned __int8 *
call    _Z8byte2BitPhS_i ; byte2Bit(uchar *,uchar *,int)
lea     rdx, [rbp+var_90] ; unsigned __int8 *
lea     rax, [rbp+var_50]
mov     rcx, rax ; unsigned __int8 *
call    _Z11pc1_replacePhS_ ; pc1_replace(uchar *,uchar *)
mov     [rbp+var_4], 0
jmp     loc_402098

```

图 18 get_subkey 函数开始部分

这一段是 `get_subkey` 函数的开始部分，其作用是从传入的密钥数据中生成一个“子密钥”，为后续的加密操作做准备的步骤。这段代码完成了子密钥生成的初始化阶段：将输入的密钥字节转换为位表示，并进行初始置换，为后续的多轮处理做准备。这种结构在实现对称加密时非常典型，从这里推测是 DES 加密算法。

函数开始的 `push rbp` 和 `mov rbp, rsp` 是设置基址帧指针，方便访问局部变量和参数。`sub rsp, 0E0h` 也就是分配了 224 字节的栈空间用于本函数的局部变量。接下来 `mov [rbp+arg_0], rcx` 是保存传入的参数（调用者通过 `rcx` 传进来的第一个参数，也就是一个字节数组的指针）。这应该是原始密钥或者部分密钥数据。然后程序准备调用一个名为 `byte2Bit` 的函数，也就是把原始的字节数据转换成位表示。为了这个转换，先通过 `lea rax, [rbp+var_50]` 把一个缓冲区地址加载到 `rax`（这将用于存储转换后的结果），并设置第三个参数为 8（可能表示转换 8 个字节），接着设置 `rcx` 为源数据地址（密钥），`rdx` 为目标地址，然后调用 `byte2Bit` 函数进行转换。

转换完成后，程序又准备调用另一个名为 `pc1_replace` 的函数，可以推断是 DES 加密中的 PC-1 步骤。它将转换后的位数组再进行重排或提取有效位。先通过 `lea rdx, [rbp+var_90]` 和 `lea rax, [rbp+var_50]` 准备参数，表示源是位转换后的结果，目标是新的缓冲区。`rcx` 被设为源缓冲区地址，然后调用 `pc1_replace`，完成位级别的置换或裁剪。

最后一行 `mov [rbp+var_4], 0` 是将一个整型变量清零，看上去像是为一个即将开始的循环做准备，`var_4` 很可能是一个索引、轮次计数器或某种标记位。最后通过 `jmp loc_402098` 跳转到接下来的主处理逻辑（即子密钥生成的核心部分）开始执行。



图 19 get_subkey 函数主要部分

接下来是 get_subkey 函数的主循环和收尾部分，整体是一个子密钥生成过程。整个 get_subkey 函数其实就是做以下事情：接收原始密钥（8 字节）、把它转换成位表示、执行一次 PC1 初始置换、然后进行 16 轮处理，每轮根据

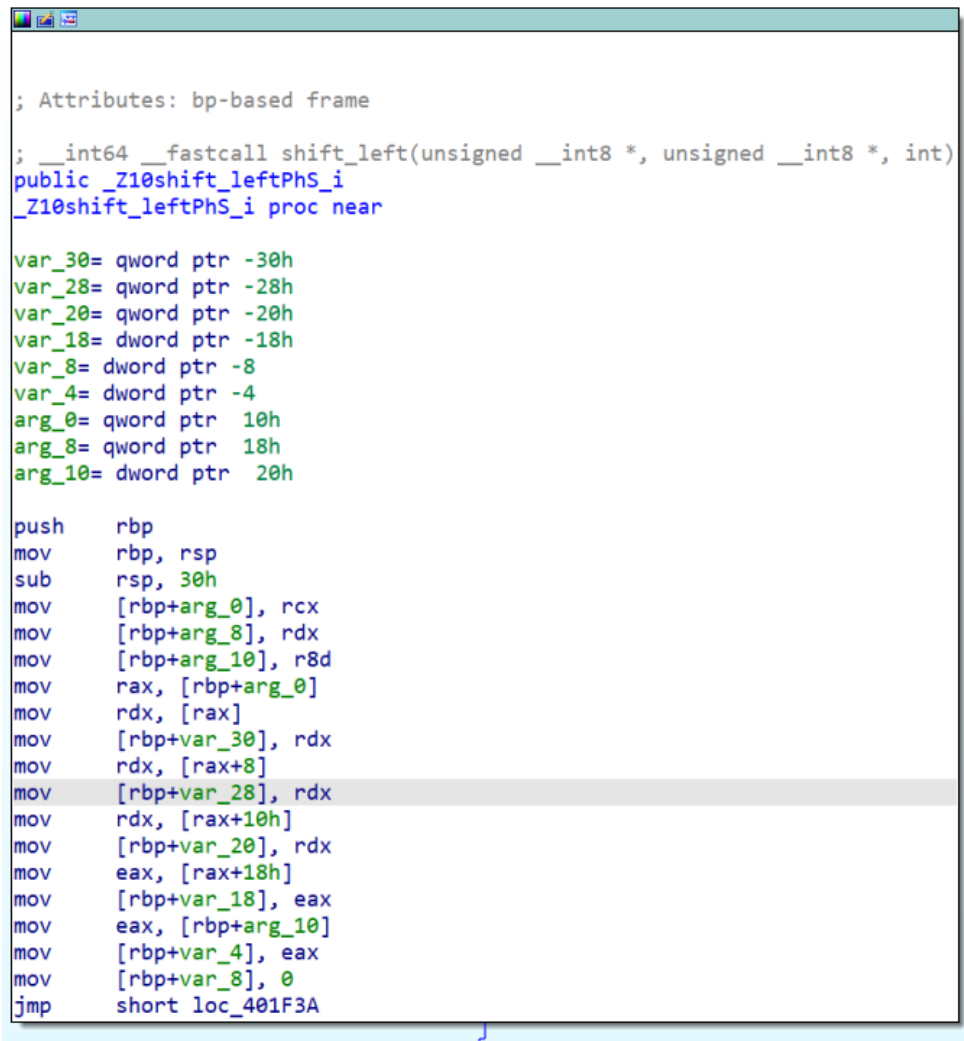
LOOP_Table 决定移位数；左移密钥的左半和右半；执行 **PC2** 置换得到本轮子密钥；写入 **subkey** 表的对应位置；最终在全局的 **subkey** 中构造好一个完整的 16 轮子密钥序列，每轮 48 位。从这结构上看，其实就是 DES 加密算法中的子密钥。

首先是轮次循环逻辑 (**loc_402098** 与 **loc_401FBE**)。这部分以 **cmp [rbp+var_4], 0Fh** 开头，检查当前轮数是否小于等于 15，总共执行 16 轮（正好与 DES 的 16 个子密钥生成轮数一致）。如果条件满足，就跳转到 **loc_401FBE** 执行一整轮子密钥生成。

进入 **loc_401FBE** 后。首先将当前轮数 **var_4** 读入 **eax**，通过 **movsxd** 拓展成 64 位的索引值；接着加载 **LOOP_Table** 的地址，这个表可能定义了每一轮的左移位数（在 DES 中每轮有不同的左移策略）；接着获取当前轮的左移位数 **al**，符号扩展成 **ecx**；接着准备对 **var_90** 中的密钥进行左移（即 **shift_left**（左半部分，左半部分，**n**））；接着再次取出本轮移位数，调用 **shift_left** 第二次，用于右半部分；之后调用 **pc2_replace**，对左移后的 56 位密钥做第二次置换，从而生成本轮实际用于加密的 48 位子密钥。结果保存在 **var_C0** 这块大缓冲区中。接着，程序根据当前轮数 **var_4** 计算一个偏移值，以便将本轮生成的子密钥写入 **subkey** 数组中正确的位置。这个偏移的计算方式是：**offset = round * 48**（即 **round * 6 * 8** 字节）。用了一种比较手动的方式（先乘 2，加，加，最后左移 4 位），然后加到 **subkey** 的起始地址上。之后程序把 **var_C0** 中的子密钥 48 字节内容逐段（按 8 字节）写入 **subkey** 表中。最后一条 **add [rbp+var_4], 1** 把轮数加 1，为下一轮做准备，程序随后会跳回上方继续下一轮。

接着分析函数结束 (**add rsp, 0E0h** 到 **retn**)。这一段是 **get_subkey** 的退出流程：**add rsp, 0E0h** 为回收之前为局部变量分配的栈空间；**pop rbp** 为恢复之前保存的基址指针；**retn** 为返回调用者处，整个子密钥生成流程完成。

3.2.2 shift_left 函数



```
; Attributes: bp-based frame

; __int64 __fastcall shift_left(unsigned __int8 *, unsigned __int8 *, int)
public _Z10shift_leftPhS_i
_Z10shift_leftPhS_i proc near

var_30= qword ptr -30h
var_28= qword ptr -28h
var_20= qword ptr -20h
var_18= dword ptr -18h
var_8= dword ptr -8
var_4= dword ptr -4
arg_0= qword ptr 10h
arg_8= qword ptr 18h
arg_10= dword ptr 20h

push    rbp
mov     rbp, rsp
sub     rsp, 30h
mov     [rbp+arg_0], rcx
mov     [rbp+arg_8], rdx
mov     [rbp+arg_10], r8d
mov     rax, [rbp+arg_0]
mov     rdx, [rax]
mov     [rbp+var_30], rdx
mov     rdx, [rax+8]
mov     [rbp+var_28], rdx
mov     rdx, [rax+10h]
mov     [rbp+var_20], rdx
mov     eax, [rax+18h]
mov     [rbp+var_18], eax
mov     eax, [rbp+arg_10]
mov     [rbp+var_4], eax
mov     [rbp+var_8], 0
jmp     short loc_401F3A
```

图 20 shift_left 起始部分

这段代码是 `shift_left` 函数的起始部分，它的作用是对输入的位数组执行循环左移操作。它先把源数据分段读取出来放在局部变量中，然后设置移位的目标值和计数器，即将进入核心的左移循环逻辑。在 DES 加密中，这通常是对密钥进行分轮左移（为了增加密钥变化性）的关键步骤。

函数开始时通过 `push rbp` 和 `mov rbp, rsp` 建立当前函数的帧结构，便于访问局部变量和参数。接着 `sub rsp, 30h` 为函数的局部变量分配了 48 字节的栈空间，表明函数的中间处理会用到一些缓存或状态变量。接下来的三条 `mov` 指令将传入的三个参数保存到栈上的对应位置：`arg_0`（`rcx`）是原始的位数组指针；`arg_8`（`rdx`）是左移后存储结果的目标指针；`arg_10`（`r8d`）是表示左移的位数。然后函数从 `arg_0` 指针所指向的内存中，按结构读取多个部分（总共 72 位

或更多)，并将其分段存入本地变量中。具体而言：从源地址读取 8 字节放入 var_30；再读取下一个 8 字节放入 var_28；然后再读取 8 字节放入 var_20；最后从偏移+18h 处读取 4 字节数据放入 var_18。接着，函数将 arg_10 的值（即要左移的位数）保存到 var_4 中，作为循环计数或移位依据。同时将 var_8 设置为 0，通常表示一个循环变量或临时计数器。最后，执行跳转到 loc_401F3A，显然是接下来的主处理循环部分，它将基于上述准备的数据块和移位值开始执行循环左移操作。

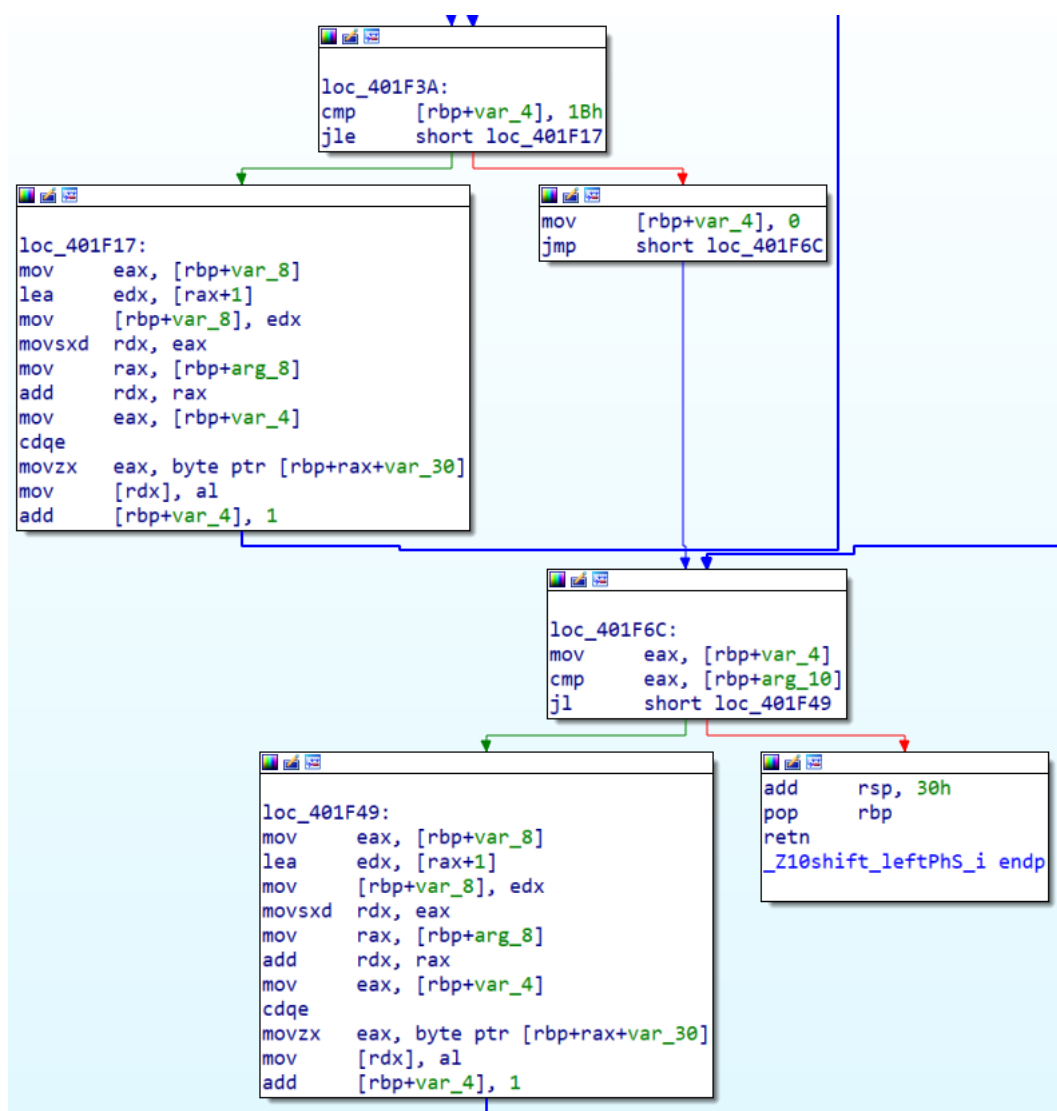


图 21 shift_left 主要部分

现在分析 `shift_left` 函数中从 `loc_401F3A` 开始的程序。整个逻辑就是把原数组的内容循环左移了 `n` 位，通过两个阶段来完成目标数组的填充，先处理“尾部移动到前”，再处理“头部补到尾”。每一行都在为正确搬移一位比特服

务，目标是构造出一个左移结果，供后续 DES 子密钥生成使用。

首先是 loc_401F3A。先从 [rbp+var_4] 中取出当前的读取偏移量（初始是你左移的位数）。然后和 1Bh（十进制的 27）做比较，判断是否还在源数组的尾部范围内（28 位以内）。如果是，就跳转到 loc_401F17，继续把剩下的内容搬到新数组最前面。否则说明尾部搬完了，跳出这一阶段，去搬头部数据。

接着是 loc_401F17。这段就是从偏移位置开始，依次把原数组的“尾巴”搬到目标数组前面。mov eax, [rbp+var_8]：从 var_8 取出当前写入偏移（新数组写入到哪里了）。lea edx, [rax+1]：计算新的写入偏移（下一位置）。mov [rbp+var_8], edx：更新写入偏移。movsxd rdx, eax：把写入偏移 sign-extend 成 64 位，用于地址计算。mov rax, [rbp+arg_8]：取出目标数组的地址（输出数组）。add rdx, rax：将写入偏移加到数组基地址，计算出写入目标位置。mov eax, [rbp+var_4]：从 var_4 获取读取偏移。cdqe：把 eax 扩展成 64 位 rax。movzx eax, byte ptr [rbp+rax+var_30]：从源数组中读取偏移 var_4 的字节。mov [rdx], al：把这个字节写入目标数组当前位置。add [rbp+var_4], 1：读取偏移加 1，为下一个字节做准备。

接着是搬头部数据准备阶段。将读取偏移 var_4 归零，准备重新从原数组最前面开始读取。跳转到下一阶段逻辑 loc_401F6C，那是搬头部数据的开始。

接下来是 loc_401F6C。mov eax, [rbp+var_4]：获取当前读取偏移。cmp eax, [rbp+arg_10]：将读取偏移与目标数组长度进行比较（目标还没填满吗？）。jl short loc_401F49：如果目标还没填满，跳转去搬头部内容（接下来那段）。

接下来为 loc_401F49。这段代码执行的工作是把原数组前 n 位的数据复制到目标数组后部，完成循环左移的“补尾”。流程仍然是：从原数组读取、写入目标数组、更新两个偏移。

最后是函数收尾。add rsp, 30h：恢复函数运行前的栈空间。pop rbp：恢复旧的基址指针。retn：函数返回。

3.2.3 pc2_replace 函数

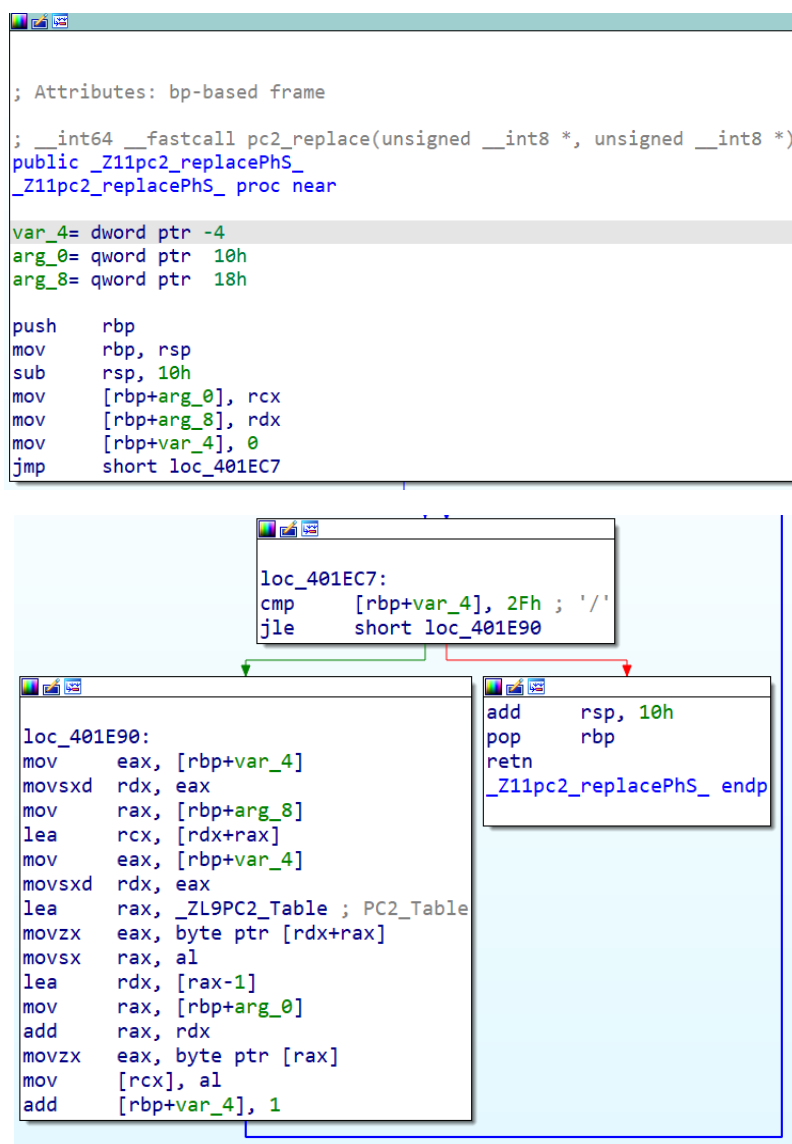


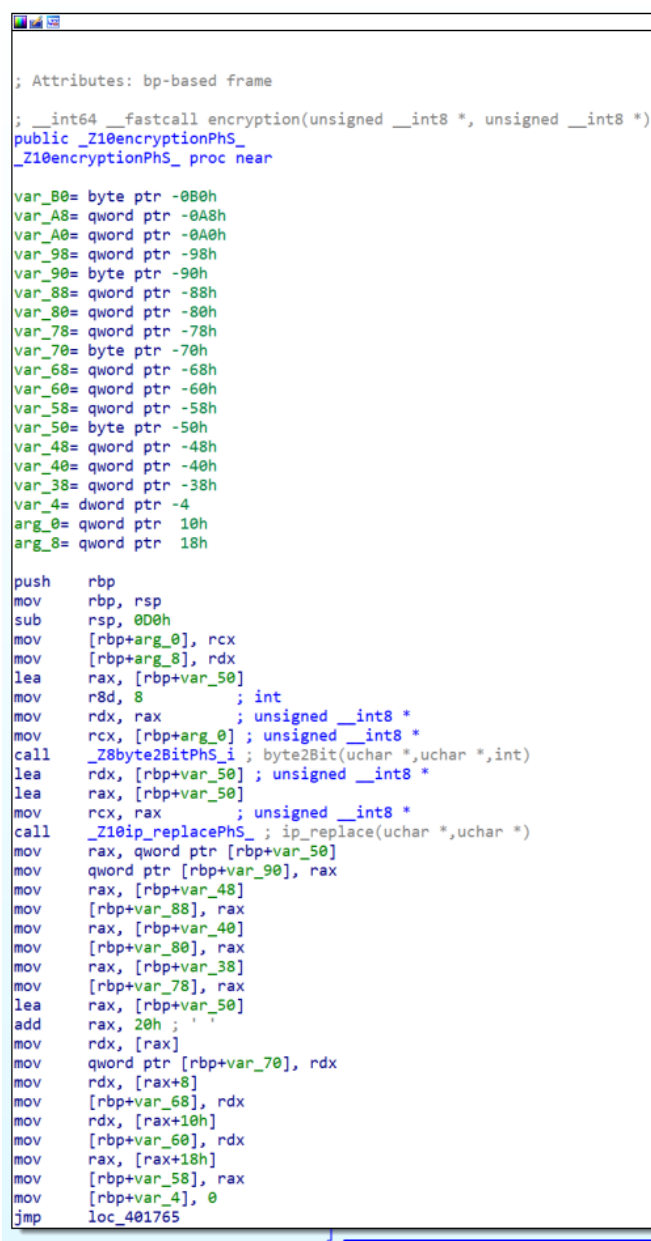
图 22 DES 算法的 PC2 置换过程

这段汇编代码实现的是 DES 算法中的 PC2 置换过程，它将 56 位密钥压缩成 48 位子密钥。函数一开始保存当前的基址指针 rbp，为局部变量和参数分配栈空间，并将两个输入参数保存下来——第一个参数是原始的 56 位密钥（以字节表示），第二个是用于存放置换后结果的目标地址。随后初始化一个整型局部变量为 0，作为循环索引。紧接着进入循环判断阶段，检查当前索引是否小于或等于 47，也就是确保总共执行 48 轮置换操作。每一轮开始时，先根据当前的索引计算出目标输出缓冲区中要写入的地址。接下来，根据当前循环索引从 PC2 置换表中取出一个值，这个值代表原始 56 位密钥中要取哪一位参与生成子密钥。由于 PC2 表是 1-based 编号的，需要减 1 得到 0-based 索引。然后使用这

个偏移从原始密钥中读取相应的字节值。最后把这个字节写入目标位置，也就是子密钥中的一位就生成了。完成写入后，将循环索引加 1，准备下一轮迭代。这个过程会一直重复直到 48 位都被处理完。循环结束后，程序清理栈帧，恢复之前的 rbp，然后返回，整个 PC2 置换就结束了，调用者将拿到已经格式化好的 48 位子密钥结果。

3.3 encryption 函数

3.3.1 encryption 主函数



```
; Attributes: bp-based frame

; __int64 __fastcall encryption(unsigned __int8 *, unsigned __int8 *)
public _Z10encryptionPhS_
_Z10encryptionPhS_ proc near

var_80= byte ptr -080h
var_A8= qword ptr -0A8h
var_A0= qword ptr -0A0h
var_98= qword ptr -98h
var_90= byte ptr -90h
var_88= qword ptr -88h
var_80= qword ptr -80h
var_78= qword ptr -78h
var_70= byte ptr -70h
var_68= qword ptr -68h
var_60= qword ptr -60h
var_58= qword ptr -58h
var_50= byte ptr -50h
var_48= qword ptr -48h
var_40= qword ptr -40h
var_38= qword ptr -38h
var_4= dword ptr -4
arg_0= qword ptr 10h
arg_8= qword ptr 18h

push    rbp
mov     rbp, rsp
sub     rsp, 0D0h
mov     [rbp+arg_0], rcx
mov     [rbp+arg_8], rdx
lea     rax, [rbp+var_50]
mov     r8d, 8
mov     rdx, rax
mov     rcx, [rbp+arg_0]
call    _Z8byte2BitPhS_i
lea     rdx, [rbp+var_50]
lea     rax, [rbp+var_50]
mov     rcx, rax
call    _Z10ip_replacePhS_
mov     rax, qword ptr [rbp+var_50]
mov     qword ptr [rbp+var_90], rax
mov     rax, [rbp+var_48]
mov     [rbp+var_88], rax
mov     rax, [rbp+var_40]
mov     [rbp+var_80], rax
mov     rax, [rbp+var_38]
mov     [rbp+var_78], rax
lea     rax, [rbp+var_50]
add     rax, 20h
mov     rdx, [rax]
mov     qword ptr [rbp+var_70], rdx
mov     rdx, [rax+8]
mov     [rbp+var_68], rdx
mov     rdx, [rax+10h]
mov     [rbp+var_60], rdx
mov     rax, [rax+18h]
mov     [rbp+var_58], rax
mov     [rbp+var_4], 0
jmp     loc_401765
```

图 23 encryption 主函数开头部分

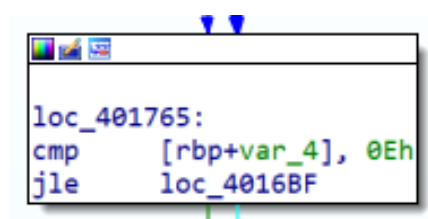
这段汇编是 `encryption` 函数的前半部分，它实现的是 DES 加密流程中的初始化阶段。程序一开始保存旧的栈帧指针，并为局部变量在栈上开辟了 `0xD0` 字节的空间以供后续使用。随后将传入的两个参数，也就是明文输入和加密输出的内存地址，分别保存在局部变量中。

接下来，它构造参数准备调用 `byte2Bit` 函数，也就是说将 8 字节（64 位）的明文数据从字节表示转换成位表示，转换结果存储在当前栈帧的 `var_50` 区域。转换完成后，立即准备并调用 `ip_replace` 函数，对那块刚生成的位数据执行初始置换（IP 置换），这也是 DES 加密的标准步骤之一，目的是打乱输入比特的顺序增加扩散性。

之后程序将 `var_50` 中经过 IP 置换后的前 32 字节数据拆成两部分，分别存入 `var_90` 到 `var_78` 等变量，实际是把 64 位数据的前 32 位（左半部分）和后 32 位（右半部分）分开存储。这相当于为接下来的 Feistel 结构迭代做准备，因为 DES 在加密中会循环操作左右两部分。

紧接着，它又对 `var_50 + 0x20` 偏移地址的数据进行处理，取出了后续可能是密钥相关或中间结果的数据，并分别放进 `var_70` 到 `var_58` 的局部变量区域，用于后续的循环计算。

最后，它将循环控制变量初始化为 0，准备进入主加密循环逻辑，该循环起始位置跳转至 `loc_401765`，从这里开始正式进入 DES 的 16 轮 Feistel 加密过程。



```

mov     eax, [rbp+var_4]
movsxd  rdx, eax
mov     rax, rdx
add     rax, rax
add     rax, rdx
shl     rax, 4
lea     rdx, subkey
lea     rcx, [rax+rdx]
lea     rdx, [rbp+var_80] ; unsigned __int8 *
lea     rax, [rbp+var_70]
mov     r8, rcx ; unsigned __int8 *
mov     rcx, rax ; unsigned __int8 *
call    _Z6f_funcPhS_S_ ; f_func(uchar *,uchar *,uchar *)
lea     rdx, [rbp+var_80] ; unsigned __int8 *
lea     rax, [rbp+var_90]
mov     r8d, 20h ; ' ' ; int
mov     rcx, rax ; unsigned __int8 *
call    _Z7byteXORPhS_i ; byteXOR(uchar *,uchar *,int)
mov     rax, qword ptr [rbp+var_90]
mov     qword ptr [rbp+var_50], rax
mov     rax, [rbp+var_88]
mov     [rbp+var_48], rax
mov     rax, [rbp+var_80]
mov     [rbp+var_40], rax
mov     rax, [rbp+var_78]
mov     [rbp+var_38], rax
lea     rax, [rbp+var_50]
add     rax, 20h ; ' '
mov     rdx, qword ptr [rbp+var_70]
mov     [rax], rdx
mov     rdx, [rbp+var_68]
mov     [rax+8], rdx
mov     rdx, [rbp+var_60]
mov     [rax+10h], rdx
mov     rdx, [rbp+var_58]
mov     [rax+18h], rdx
lea     rdx, [rbp+var_50] ; unsigned __int8 *
lea     rax, [rbp+var_50]
mov     rcx, rax ; unsigned __int8 *
call    _Z10fp_replacePhS_ ; fp_replace(uchar *,uchar *)
mov     rdx, [rbp+arg_8] ; unsigned __int8 *
lea     rax, [rbp+var_50]
mov     r8d, 8 ; int
mov     rcx, rax ; unsigned __int8 *
call    _Z8bit2BytePhS_i ; bit2Byte(uchar *,uchar *,int)
nop
add     rsp, 000h
pop     rbp
retn
_Z10encryptionPhS_ endp

```

图 24 DES 的 16 轮 Feistel 加密结构中的一轮逻辑处理

这段汇编代码继续承接 encryption 函数的主体部分，也就是 DES 的 16 轮 Feistel 加密结构中的一轮逻辑处理。这是 DES 结构中一轮的典型流程：使用当前轮次对应的子密钥对右半部分做 Feistel 运算，然后和左半部分异或，完成后左右互换，继续下一轮。

程序首先比较当前轮数变量是否小于等于 14（也就是总共 15 轮，从 0 到 14），如果是，就进入循环体内部继续执行当前这一轮的加密操作。在这一轮中，先根据当前轮数 var_4 计算子密钥在 subkey 数组中的偏移位置。具体计算逻辑是：将轮数值乘以 3 再乘以 16（即每轮子密钥占用 48 字节），通过移位和加法得到实际地址，再将这个地址加到 subkey 基地址上，形成当前轮要用的子密钥指针 rcx。随后准备调用 f_func 函数，这个函数是 DES 的核心——Feistel

函数，它会对当前轮的右半部分数据与子密钥进行复杂操作（扩展、S 盒替换、P 置换等），输出 32 位结果，放入 var_B0 开始的区域中。

紧接着调用 byteXOR 函数，把 f_func 的输出与左半部分的数据进行异或（XOR）操作，结果更新原来的右半部分数据。异或之后，当前轮加密结束，需要交换左右两部分的值，为下一轮准备。所以它将旧右半部分的数据（从 var_70 开始的四个 qword，也就是 32 字节）复制到左半部分的存储区（var_90 开始），然后把刚才异或得到的新右半部分（在 var_B0 区域）重新填回右半部分的变量区域（var_70 开始）。这些操作确保左右数据在每轮都互换，为下轮继续处理创造条件。最后，将轮数变量加一，准备进入下一轮。



```
mov     eax, [rbp+var_4]
movsxd  rdx, eax
mov     rax, rdx
add     rax, rax
add     rax, rdx
shl     rax, 4
lea     rdx, subkey
lea     rcx, [rax+rdx]
lea     rdx, [rbp+var_B0] ; unsigned __int8 *
lea     rax, [rbp+var_70]
mov     r8, rcx ; unsigned __int8 *
mov     rcx, rax ; unsigned __int8 *
call    _Z6f_funcPhS_S_ ; f_func(uchar *,uchar *,uchar *)
lea     rdx, [rbp+var_B0] ; unsigned __int8 *
lea     rax, [rbp+var_90]
mov     r8d, 20h ; ' ' ; int
mov     rcx, rax ; unsigned __int8 *
call    _Z7byteXORPhS_i ; byteXOR(uchar *,uchar *,int)
mov     rax, qword ptr [rbp+var_90]
mov     qword ptr [rbp+var_50], rax
mov     rax, [rbp+var_88]
mov     [rbp+var_48], rax
mov     rax, [rbp+var_80]
mov     [rbp+var_40], rax
mov     rax, [rbp+var_78]
mov     [rbp+var_38], rax
lea     rax, [rbp+var_50]
add     rax, 20h ; ' '
mov     rdx, qword ptr [rbp+var_70]
mov     [rax], rdx
mov     rdx, [rbp+var_68]
mov     [rax+8], rdx
mov     rdx, [rbp+var_60]
mov     [rax+10h], rdx
mov     rdx, [rbp+var_58]
mov     [rax+18h], rdx
lea     rdx, [rbp+var_50] ; unsigned __int8 *
lea     rax, [rbp+var_50]
mov     rcx, rax ; unsigned __int8 *
call    _Z10fp_replacePhS_ ; fp_replace(uchar *,uchar *)
mov     rdx, [rbp+arg_8] ; unsigned __int8 *
lea     rax, [rbp+var_50]
mov     r8d, 8 ; int
mov     rcx, rax ; unsigned __int8 *
call    _Z8bit2BytePhS_i ; bit2Byte(uchar *,uchar *,int)
nop
add     rsp, 0D0h
pop     rbp
retn
_Z10encryptionPhS_ endp
```

图 25 encryption 函数的尾部处理阶段

这段汇编完成了 DES 的最后一轮计算、左右数据交换、最终置换和输出恢复，是标准的 DES 加密流程最后收尾阶段。

首先，从当前轮数变量 `var_4` 中取出值，进行符号扩展，接着通过几步加法和左移，算出当前轮次对应的子密钥在 `subkey` 表中的地址偏移量。这个计算逻辑等价于 $\text{index} * 48$ ，因为每一轮子密钥是 48 字节。之后把这个偏移量加到 `subkey` 的地址上，得到当前子密钥的指针 `rcx`。接下来将 `var_70` 区域中保存的右半部分明文数据作为输入之一，`rcx` 为子密钥地址，`var_B0` 区域为输出缓冲区，调用 `f_func` 函数执行 Feistel 核心加密逻辑。函数结束后，在 `var_B0` 中得到 Feistel 函数的输出结果。

随后是一个异或操作 `byteXOR`，它将 Feistel 输出与左半部分的数据（`var_90` 区域）按位异或，结果仍存入 `var_B0` 区域，也就是生成新的右半部分数据。

接下来执行一次左右部分的**最终交换操作**，把左半部分（`var_90 ~ var_78`）复制到输出缓冲区（`var_50` 开头），而右半部分（`var_70 ~ var_58`）复制到 `var_50 + 0x20` 开始的后半部分，实现左、右互换后的组合。

交换完成后，程序调用 `fp_replace`，也就是 DES 的逆初始置换函数，它对整个 64 位组合密文位串做最后的排列变换，结果仍存入 `var_50` 区域。

最后，调用 `bit2Byte` 函数，将最终排列后的位数组转换回 8 字节（64 位）密文输出，写入由第二个参数 `arg_8` 指向的内存中。整个 `encryption` 函数至此结束，恢复栈空间并返回。

3.3.2 byte2Bit 函数

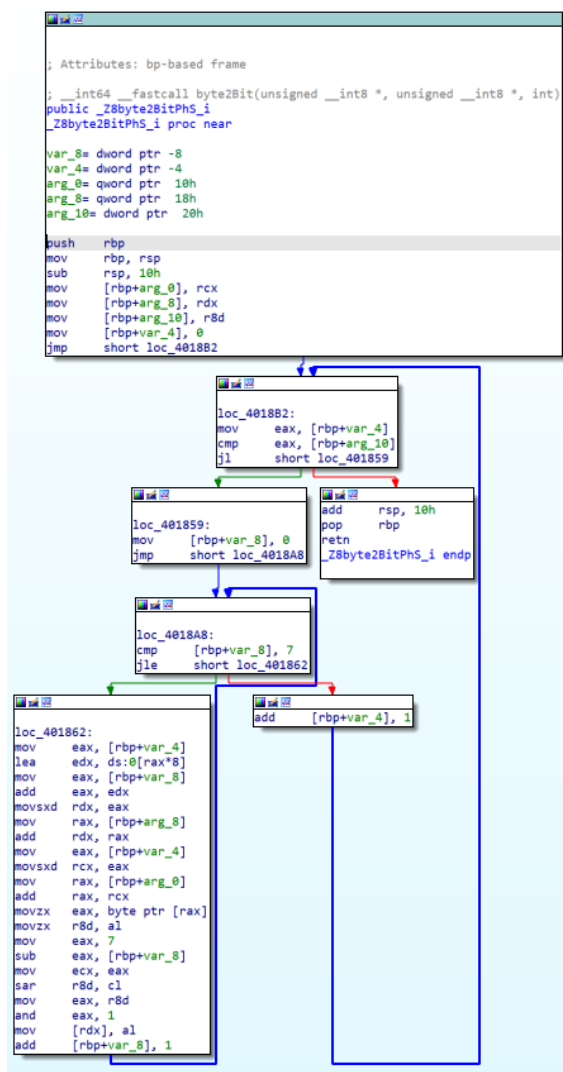


图 26 byte2Bit 函数

这段汇编实现的是 byte2Bit 函数，它的作用是逐字节地将输入字节数组中的每一个字节拆解为 8 个独立的 bit 并顺序填入输出数组中，常用于 DES 等加密算法中做初始位排列、或是明文处理阶段的预处理步骤。

函数开始时设置栈帧并分配 16 字节局部空间，然后将传入的三个参数依次保存在栈中：输入字节数组的地址、输出位数组的地址、要处理的字节数量。var_4 初始化为 0，用作外层循环的计数器，表示当前处理第几个字节。

进入外层循环，判断是否处理完所有字节。每轮外层循环对应处理一个字节。开始后，var_8 被初始化为 0，用于内层循环，它表示当前处理这个字节的第几位 bit（从第 0 位到第 7 位）。

内层循环每次处理当前字节的一个 bit 位，一共要处理 8 位。当 var_8 <= 7

时继续循环。我们先通过 `var_4`（字节序号）乘以 8 计算当前字节展开到 bit 时在输出数组中的起始偏移位置，再加上 `var_8` 得到当前 bit 在输出数组中的总偏移，从而确定输出位置。

接下来根据当前字节索引从输入数组中读出这个字节（通过 `arg_0 + var_4` 访问），并将它转换为整型值。然后通过 `7 - var_8` 计算当前要提取的 bit 在字节中的位置（因为高位优先），将字节右移相应位数，再通过 `& 1` 提取最低位，也就是当前 bit 的值。

将得到的 bit 值写入输出数组的对应位置，然后 `var_8` 加 1，继续处理当前字节的下一位；8 位处理完后，`var_4` 加 1，转向处理下一个字节。如此循环直到全部字节都处理完毕。函数最后恢复栈空间并返回。

3.3.3 ip_replace 函数

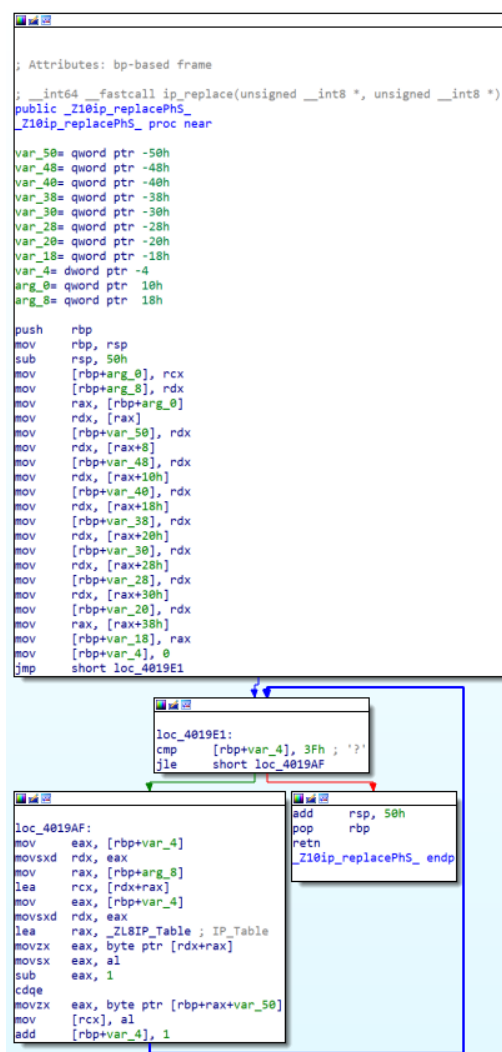


图 27 ip_replace 函数

这段汇编代码实现的是 `ip_replace` 函数，对应于 DES 中的初始置换操作，是整个加密过程的第一步。这段函数把输入明文按 IP 表顺序重排 bit。具体表 `IP_Table` 没在这段代码中定义，但我们可以确定它是一个 64 字节长的查找表，每个值表示原数据中一个 bit 的编号，按这个顺序生成新的 64-bit 输出流。这正是加密流程的第一步，旨在打乱原始数据的顺序，为后续的轮函数处理做好准备。

函数接收两个参数：`unsigned __int8 *input` 和 `unsigned __int8 *output`。我们知道 DES 在加密开始前，会先对 64 位输入明文数据进行一个固定的置换，这就是 IP 置换。

首先建立栈帧，并为局部变量分配了 0x50 (80) 字节空间，然后将函数的两个参数分别保存在 `[rbp+arg_0]` 和 `[rbp+arg_8]` 中。

接下来开始从 `arg_0` (也就是输入数据) 中读取内容。`mov rax, [rbp+arg_0]` 是把输入数组地址取出来，然后每隔 8 个字节一次读取，依次将输入的 64 位数据 (按 8 个字节划分) 保存在 `var_50` 到 `var_18` 这些局部变量里。这段代码相当于把输入的 64 位 (8 字节) 拆分为 8 个 qword (即 8 个字节单元)，为后续按位读取做准备。

然后设置 `[rbp+var_4] = 0`，作为一个循环计数器，这个循环将会进行 64 次，因为 IP 置换对输入的 64 个 bit 每一个都做了重新排列。

接下来的主循环做了下面几件事：检查是否处理满 64 个 bit (即 `cmp var_4, 0x3F`)，没满就继续；把当前处理的 bit 索引 `var_4` 转为 `rdx`，再加上输出数组地址，得到当前要写入的输出数组位置 `rcx`；然后再把当前 bit 索引加到 IP 表的地址上，读取当前 bit 应该对应的是输入的哪一位；IP 表中存储的是从 1 到 64 的位置，所以 `sub eax, 1` 将它减一；得到要读取输入数组中的哪一个 bit，接着在 `[rbp+var_50]` 开头的的数据中读取该 bit 的值 (通过偏移量访问)；将该 bit 写入输出数组对应位置。

最后计数器加一，继续循环，直到 64 个 bit 都被置换完毕。函数结束前恢复栈空间、弹出栈帧返回。

3.3.4 f_func 函数

```
; Attributes: bp-based frame

; __int64 __fastcall f_func(unsigned __int8 *, unsigned __int8 *, unsigned __int8 *)
public _Z6f_funcPhS_S_
_Z6f_funcPhS_S_ proc near

var_50= byte ptr -50h
var_48= qword ptr -48h
var_40= qword ptr -40h
var_38= qword ptr -38h
var_30= byte ptr -30h
arg_0= qword ptr 10h
arg_8= qword ptr 18h
arg_10= qword ptr 20h

push    rbp
mov     rbp, rsp
sub     rsp, 70h
mov     [rbp+arg_0], rcx
mov     [rbp+arg_8], rdx
mov     [rbp+arg_10], r8
lea     rax, [rbp+var_30]
mov     rdx, rax ; unsigned __int8 *
mov     rcx, [rbp+arg_0] ; unsigned __int8 *
call    _Z8e_expandPhS_ ; e_expand(uchar *,uchar *)
mov     rdx, [rbp+arg_10] ; unsigned __int8 *
lea     rax, [rbp+var_30]
mov     r8d, 30h ; '0' ; int
mov     rcx, rax ; unsigned __int8 *
call    _Z7byteXORPhS_i ; byteXOR(uchar *,uchar *,int)
lea     rdx, [rbp+var_50] ; unsigned __int8 *
lea     rax, [rbp+var_30]
mov     rcx, rax ; unsigned __int8 *
call    _Z9s_replacePhS_ ; s_replace(uchar *,uchar *)
lea     rdx, [rbp+var_50] ; unsigned __int8 *
lea     rax, [rbp+var_50]
mov     rcx, rax ; unsigned __int8 *
call    _Z9p_replacePhS_ ; p_replace(uchar *,uchar *)
mov     rax, [rbp+arg_8]
mov     rdx, qword ptr [rbp+var_50]
mov     [rax], rdx
mov     rdx, [rbp+var_48]
mov     [rax+8], rdx
mov     rdx, [rbp+var_40]
mov     [rax+10h], rdx
mov     rdx, [rbp+var_38]
mov     [rax+18h], rdx
add     rsp, 70h
pop     rbp
ret     0
_Z6f_funcPhS_S_ endp
```

图 28 f_func 函数

这段汇编代码实现了一个典型的 DES 加密中使用的 Feistel 轮函数，它接受三个参数：一个输入数据指针、一个输出数据指针和一个子密钥指针。函数的核心是对输入数据进行一系列的加密转换，并将最终结果写入输出位置。

函数开始时，设置好函数栈帧并保存三个参数。接着，它准备一个临时缓冲区用来存储扩展后的输入数据。然后调用 `e_expand` 函数，该函数将输入的 32 位数据块（一般是右半部分）从 32 位扩展成 48 位，这一步是 DES 中标准的 E 扩展操作。扩展后的数据存在一个局部变量数组中。

接下来，它对扩展后的数据和当前轮的 48 位子密钥进行逐字节异或操作，

这通过调用 `byteXOR` 来实现。这一步是为了引入密钥的作用，从而改变输入数据的内容。异或后的结果仍然是 48 位，被存储回原来的缓冲区中。

然后，它调用 `s_replace` 函数进行 S 盒代换操作，将 48 位数据分成 8 个 6 位小块，并分别代入 8 个 S 盒，每个 S 盒将 6 位转换为 4 位，总共输出 32 位。这一操作是 DES 中最复杂、最关键的非线性部分，也是其安全性的重要来源。S 盒代换后的结果被写入另一个缓冲区中。

之后，它调用 `p_replace` 函数将这 32 位的数据按预定义的顺序进行置换，进一步打乱比特的顺序，提高扩散性。这个结果仍然保存在原缓冲区中。

最后，它将这个最终变换后的 32 位结果写入输出参数所指向的内存中。由于目标输出位置是以 64 位为单位的，它通过四次写操作（每次写入 8 字节）将结果完整写出。这一步确保了函数的输出能被后续的加密轮或其它处理函数使用。

3.3.5 byteXOR 函数

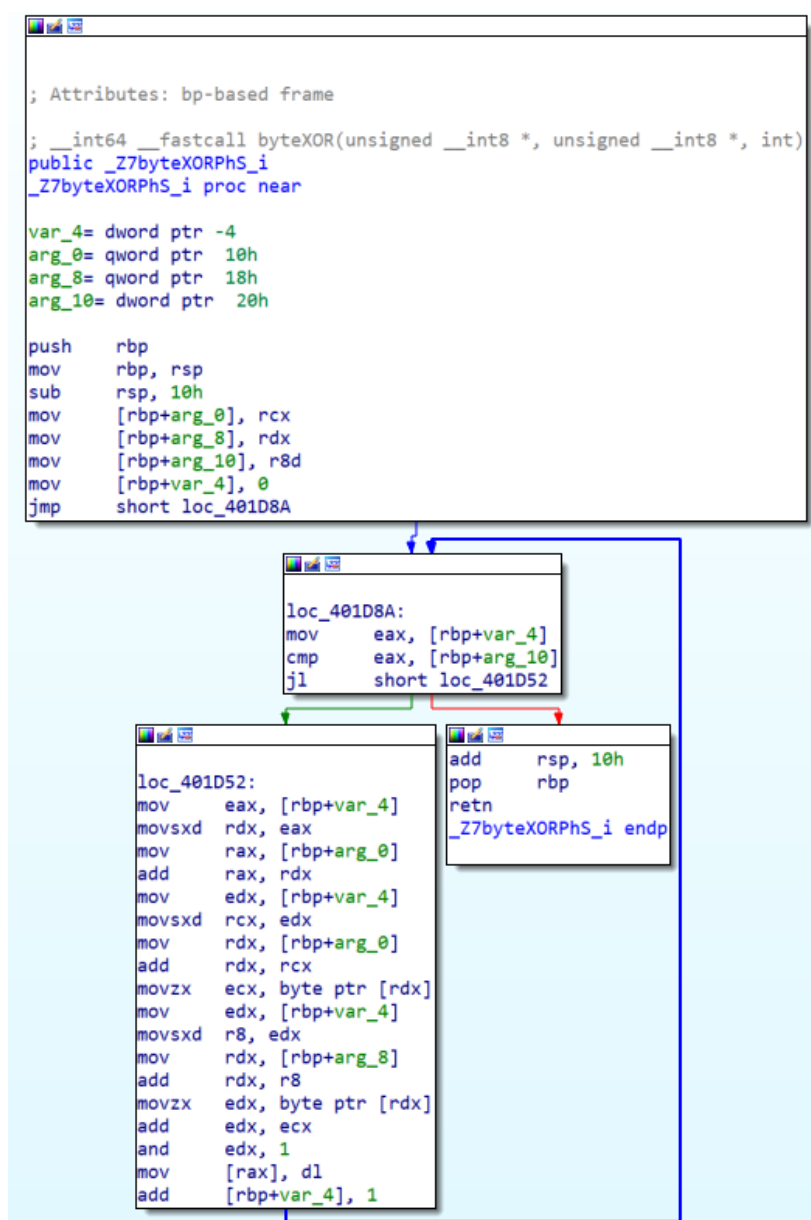


图 29 byteXOR 函数

这段汇编代码实现的是一个逐位异或操作的函数 **byteXOR**，它的作用是将两个等长的字节数组中的每一位（bit）进行按位异或，然后把结果写回第一个数组的位置。

函数一开始建立栈帧，并为局部变量预留空间。接着它将三个输入参数保存到栈中，第一个参数是目标数组指针（结果会写入这里），第二个是源数组指针（要与第一个进行异或），第三个是处理的字节数长度（即需要异或的字节数）。

初始化一个循环变量为 0，准备进入主循环。循环会从索引 0 开始处理，直到达到给定的长度为止。在循环体内，它先从当前索引位置获取第一个数组中的一个字节，并提取这个字节的所有比特（虽然这里看似是字节操作，实际上是将每个位都单独异或）。然后从第二个数组中读取相同位置的字节。同样，只取出其比特。

将两个字节逐位相加后，再与 1 相与，得到最终异或结果的最低有效位，并写回到第一个数组当前位置中。这个操作每次只写入一个 bit（最低位），而不是整个字节。说明这个函数是在进行“按位”异或，而不是传统的按字节异或，这在处理经过扩展和 S-box 等步骤之后的 bit-level 数据时是非常典型的做法。

整个函数完成一个循环之后会自增索引，继续处理下一个字节（但每次只修改一个 bit），直到处理完所有的目标字节数为止。最后，它恢复栈帧并返回。

3.3.6 fp_replac 函数

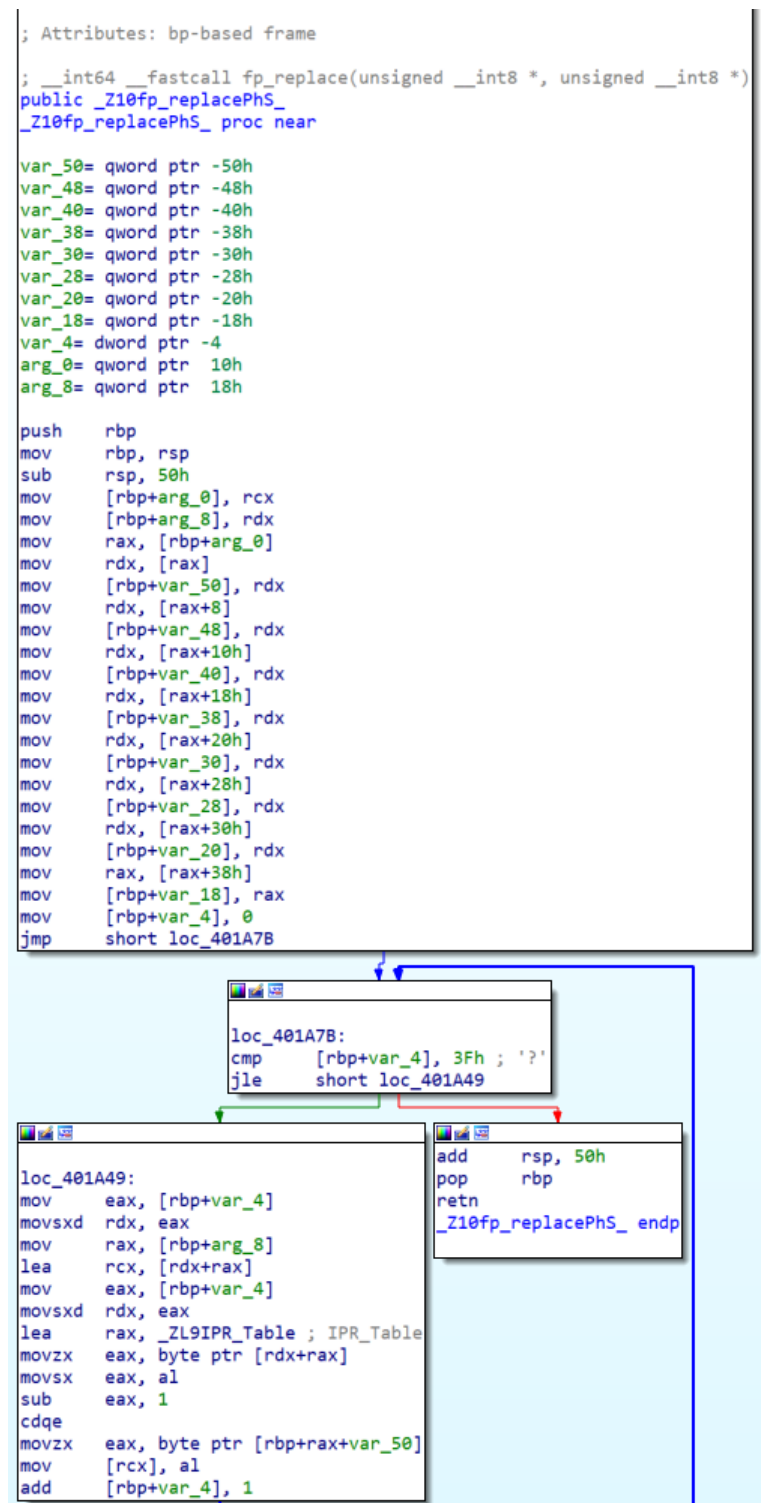


图 30 fp_replac 函数

这段汇编代码是一个实现置换操作的函数。这个函数对一个长度为 64 位的输入数据执行**最终置换**操作，它会按照一张固定的置换表把输入数据的位重新排列，结果写入另一个内存地址。这是 DES 算法中的标准步骤之一，用于将加

密轮后的结果再次打乱顺序以增加加密复杂度。

函数开始通过建立栈帧并分配 80 字节的局部变量空间。然后它将三个参数依次存入栈中：`arg_0` 是输入指针，也就是需要被置换的 64 位比特流；`arg_8` 是输出指针，用于存储置换后的结果。

紧接着，它从输入指针地址中依次取出 64 位一组的数据，共取了 8 次，把整整 64 位（8 字节）的输入数据内容拆解并保存在局部变量区域 `var_50` 到 `var_18` 中。这是为了方便后续按位操作。

然后初始化一个循环变量 `var_4` 为 0，用于遍历 64 位比特数据。进入循环，它判断当前索引是否小于等于 `0x3F`（即 63），也就是要处理 64 个比特。

在循环体内部，它计算当前目标比特位的地址（输出指针+当前索引），然后根据 `IPR_Table` 来查找当前比特在原始数据中的位置。这张表通过 `_ZL9IPR_Table` 符号引用，表中的每个条目表示输入数据的某个位将被映射到输出数据中的哪个位置。

最后将这个选中的比特值写入目标地址中，也就是输出数据的当前索引位置。这个过程持续重复 64 次，直到所有比特位完成置换。

4 DES 解密与总结

4.1 解密结果

我们可得到密钥和密文。如下所示。

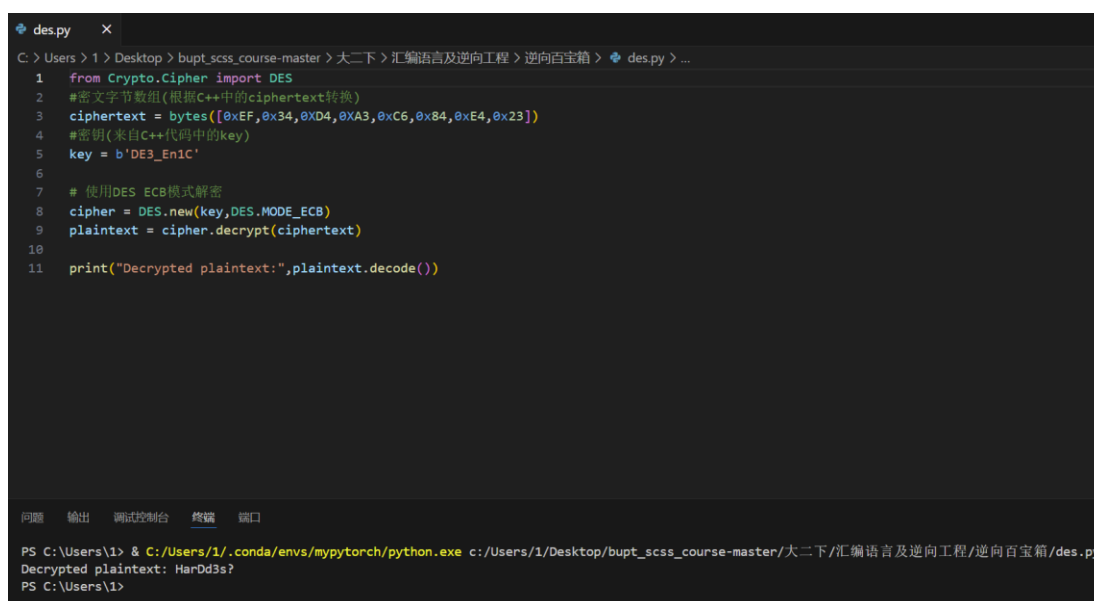
"DE3_En1C"

图 31 密钥

```
.data:0000000000404000      public __mingw_winmain_nShowCmd
.data:0000000000404000      ; DWORD __mingw_winmain_nShowCmd
.data:0000000000404000      __mingw_winmain_nShowCmd dd 0Ah          ; DATA XREF: __tmainCRTStartup:loc_401330↑w
.data:0000000000404004      align 10h
.data:0000000000404010      public result
.data:0000000000404010      ; _BYTE result[16]
.data:0000000000404010      result      db 0EFh, 34h, 0D4h, 0A3h, 0C6h, 84h, 0E4h, 23h, 8 dup(0)
.data:0000000000404010      ; DATA XREF: main+A21o
.data:0000000000404020      ; Function-local static variable
.data:0000000000404020      ; func_ptr *p_73208
.data:0000000000404020      p_73208      dq offset __DTOR_LIST__+8
.data:0000000000404020      ; DATA XREF: __do_global_dtors+4↑r
.data:0000000000404020      ; __do_global_dtors+15↑r ...
```

图 32 密文

通过解密程序，可以得到明文为：**HarDd3s?**



```
des.py
C:\Users\1\Desktop> bupt_scss_course-master > 大二下 > 汇编语言及逆向工程 > 逆向百宝箱 > des.py > ...
1 from Crypto.Cipher import DES
2 #密文字节数组(根据C++中的ciphertext转换)
3 ciphertext = bytes([0xEF,0x34,0XD4,0XA3,0xC6,0x84,0xE4,0x23])
4 #密钥(来自C++代码中的key)
5 key = b'DE3_En1C'
6
7 # 使用DES ECB模式解密
8 cipher = DES.new(key,DES.MODE_ECB)
9 plaintext = cipher.decrypt(ciphertext)
10
11 print("Decrypted plaintext:",plaintext.decode())

问题 输出 调试控制台 终端 窗口
PS C:\Users\1> & C:/Users/1/.conda/envs/mypytorch/python.exe c:/Users/1/Desktop/bupt_scss_course-master/大二下/汇编语言及逆向工程/逆向百宝箱/des.py
Decrypted plaintext: HarDd3s?
PS C:\Users\1>
```

图 33 解密 python 程序

4.2 总结

在实验过程中，我遇到了不少挑战，起初对各类参数和指令都感到相当陌生。通过持续学习和实践，我逐渐掌握了汇编指令的基本用法，现在对大多数操作都有了初步的理解。不过目前阅读代码的速度还不够快，对指令的熟悉程度也有待提升，还需要通过大量练习来加深理解。学习之路任重道远，我会继续保持努力。

同时，通过这次实验，我对 DES 加密算法及其逆向分析过程有了更深入的理解和掌握。接下来，我会通过更多实践案例来巩固和提升这方面的技能，为后续的深入研究打下更扎实的基础。