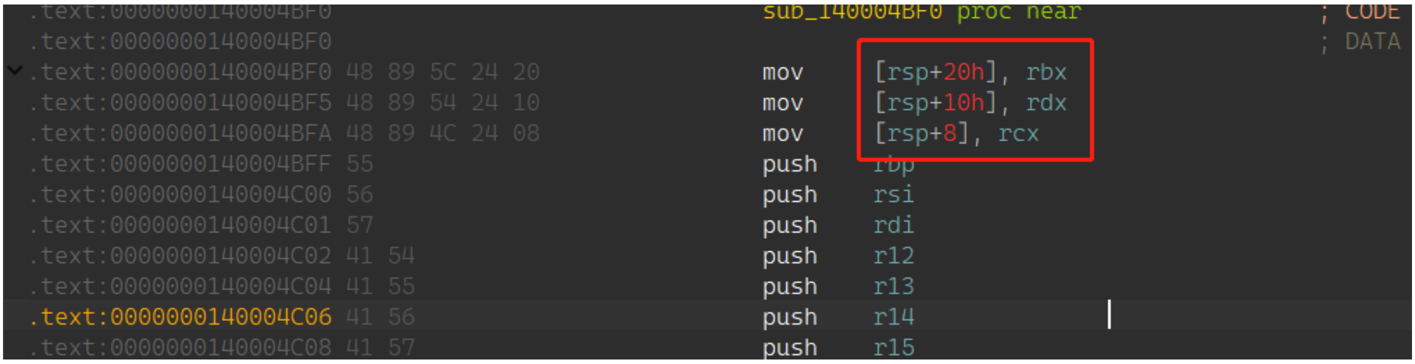
m1_read

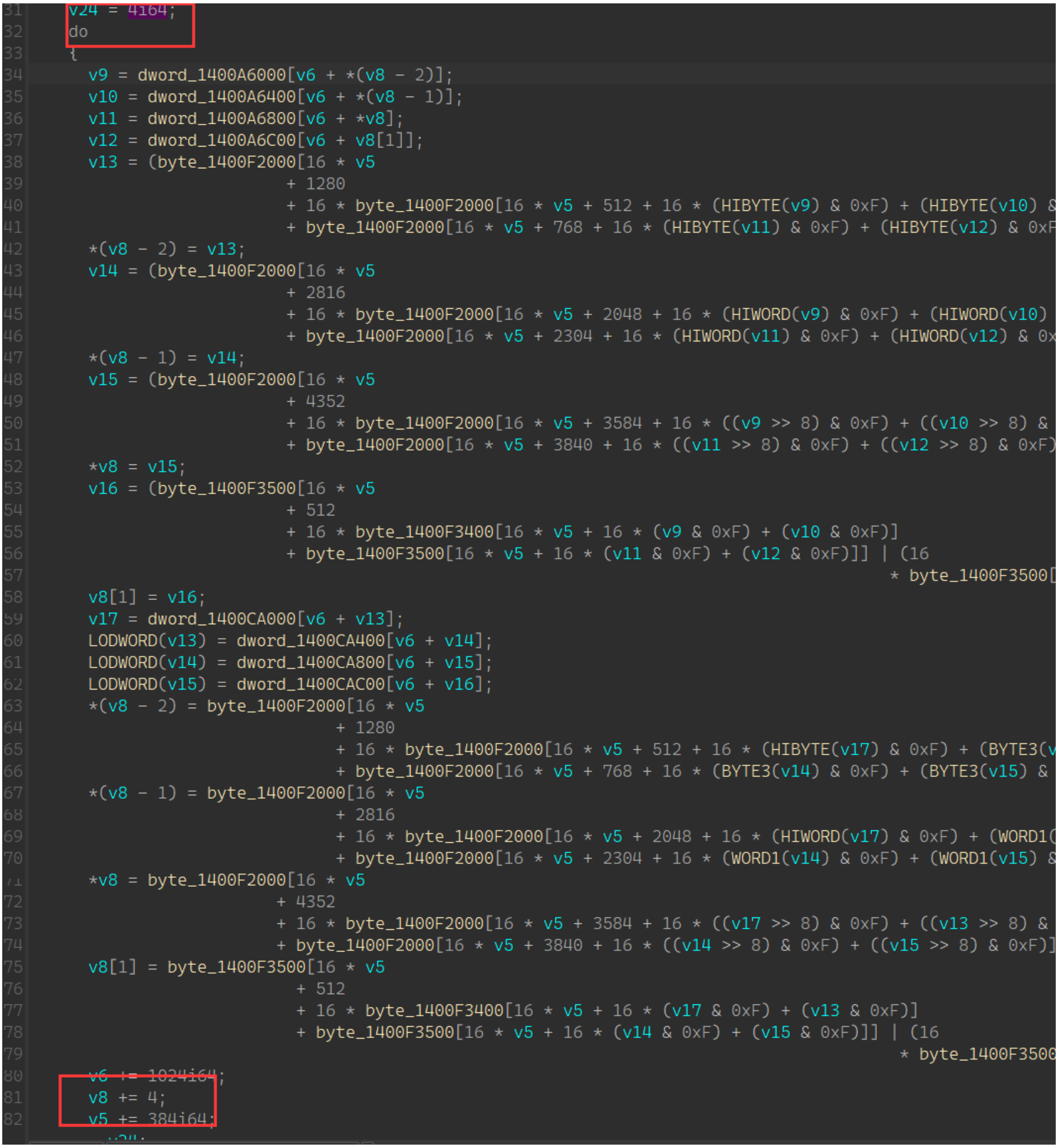
m1_read

> ### 参考
>
> 找回消失的密钥 --- DFA分析白盒AES算法_dfa攻击_奋飞安全的博客-CSDN博客
>
> REtard👨的模拟执行脚本(大爹用的Unicorn，不想写Unicorn只能用他的儿子`Qiling`了，但是Qiling需要加载很多dll所以速度会慢一点)
>
> ### 密钥获取
>
> 根据文章所讲，我们的目标就是要找到第8轮结束的位置，之后插入缺陷数据并获取最终的错误密文。而这样的错误密文需要`16`个。
>
> 由于该函数并没有过多依赖外部的一些库，所以可以选择模拟执行的方式插入缺陷数据和获取错误密文
>
> ### 参数传入
>
> 函数开头传入参数，`rcx`存储输入的地址
>
> 
>
> ```
> .text:0000000140004BF0                          sub_140004BF0 proc near      ; CODE
> .text:0000000140004BF0                                                       ; DATA
> .text:0000000140004BF0 48 89 5C 24 20           mov     [rsp+20h], rbx
> .text:0000000140004BF5 48 89 54 24 10           mov     [rsp+10h], rdx
> .text:0000000140004BFA 48 89 4C 24 08           mov     [rsp+8], rcx
> .text:0000000140004BFF 55                       push    rbp
> .text:0000000140004C00 56                       push    rsi
> .text:0000000140004C01 57                       push    rdi
> .text:0000000140004C02 41 54                    push    r12
> .text:0000000140004C04 41 55                    push    r13
> .text:0000000140004C06 41 56                    push    r14
> .text:0000000140004C08 41 57                    push    r15
> ```
>
> 由于我们只需要模拟执行该函数，所以需要在函数起始地址修改寄存器的值使其指向在内存中分配并初始化的输入，代码如下：其中`ql.mem.write(0x500000000, b"\x01" * 16)`向指定地址写入需要加密的内容。
>
> ```python
> def hook_args(ql: Qiling):
>     ql.mem.write(0x500000000, b"\x01" * 16)
>     ql.arch.regs.write("rcx", 0x500000000)
>     # print(ql.mem.read(0x500000000, 16))
>     ql.mem.write(0x500000000 + 0x10, b"\x00" * 16)
>     ql.arch.regs.write("rdx", 0x500000000 + 0x10)
>     ql.mem.write(0x500000000 + 0x20, b"\x00" * 16)
> ```

```
    ql.arch.regs.write("rbx", 0x500000000 + 0x20)
    # print("Hook Success")
    return
start_addr = 0x140004BF0
ql.hook_address(hook_args, start_addr)
```

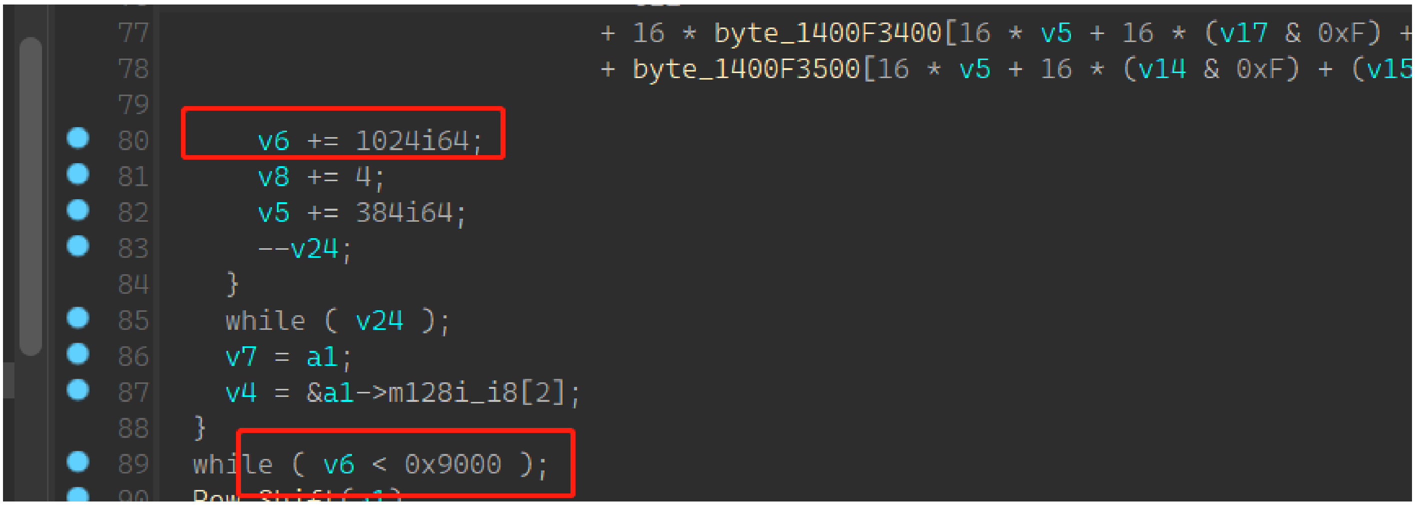
定位插入缺陷数据的位置

这一段以十六个字符为一组进行进行加密，那么这一段应该就是列混合

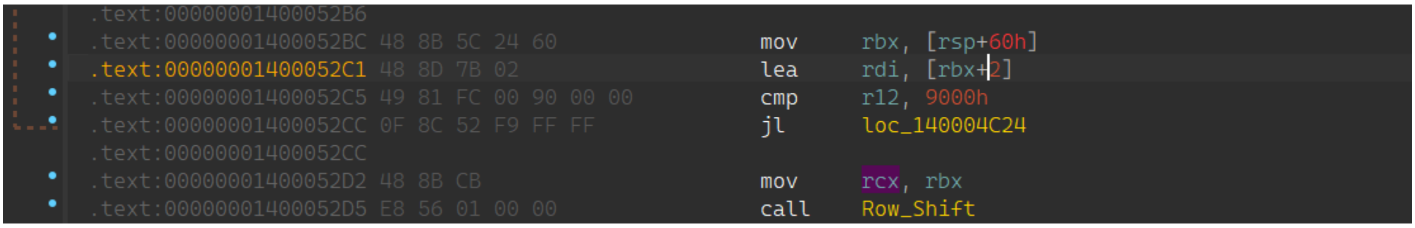


```
  v24 = 4164;
  do
  {
    v9 = dword_1400A6000[v6 + *(v8 - 2)];
    v10 = dword_1400A6400[v6 + *(v8 - 1)];
    v11 = dword_1400A6800[v6 + *v8];
    v12 = dword_1400A6C00[v6 + v8[1]];
    v13 = (byte_1400F2000[16 * v5
                        + 1280
                        + 16 * byte_1400F2000[16 * v5 + 512 + 16 * (HIBYTE(v9) & 0xF) + (HIBYTE(v10) &
                        + byte_1400F2000[16 * v5 + 768 + 16 * (HIBYTE(v11) & 0xF) + (HIBYTE(v12) & 0xF
    *(v8 - 2) = v13;
    v14 = (byte_1400F2000[16 * v5
                        + 2816
                        + 16 * byte_1400F2000[16 * v5 + 2048 + 16 * (HIWORD(v9) & 0xF) + (HIWORD(v10)
                        + byte_1400F2000[16 * v5 + 2304 + 16 * (HIWORD(v11) & 0xF) + (HIWORD(v12) & 0x
    *(v8 - 1) = v14;
    v15 = (byte_1400F2000[16 * v5
                        + 4352
                        + 16 * byte_1400F2000[16 * v5 + 3584 + 16 * ((v9 >> 8) & 0xF) + ((v10 >> 8) &
                        + byte_1400F2000[16 * v5 + 3840 + 16 * ((v11 >> 8) & 0xF) + ((v12 >> 8) & 0xF)
    *v8 = v15;
    v16 = (byte_1400F3500[16 * v5
                        + 512
                        + 16 * byte_1400F3400[16 * v5 + 16 * (v9 & 0xF) + (v10 & 0xF)]
                        + byte_1400F3500[16 * v5 + 16 * (v11 & 0xF) + (v12 & 0xF)]] | (16
                                                                                   * byte_1400F3500[

    v8[1] = v16;
    v17 = dword_1400CA000[v6 + v13];
    LODWORD(v13) = dword_1400CA400[v6 + v14];
    LODWORD(v14) = dword_1400CA800[v6 + v15];
    LODWORD(v15) = dword_1400CAC00[v6 + v16];
    *(v8 - 2) = byte_1400F2000[16 * v5
                             + 1280
                             + 16 * byte_1400F2000[16 * v5 + 512 + 16 * (HIBYTE(v17) & 0xF) + (BYTE3(v
                             + byte_1400F2000[16 * v5 + 768 + 16 * (BYTE3(v14) & 0xF) + (BYTE3(v15) &
    *(v8 - 1) = byte_1400F2000[16 * v5
                             + 2816
                             + 16 * byte_1400F2000[16 * v5 + 2048 + 16 * (HIWORD(v17) & 0xF) + (WORD1(
                             + byte_1400F2000[16 * v5 + 2304 + 16 * (WORD1(v14) & 0xF) + (WORD1(v15) &
    *v8 = byte_1400F2000[16 * v5
                       + 4352
                       + 16 * byte_1400F2000[16 * v5 + 3584 + 16 * ((v17 >> 8) & 0xF) + ((v13 >> 8) &
                       + byte_1400F2000[16 * v5 + 3840 + 16 * ((v14 >> 8) & 0xF) + ((v15 >> 8) & 0xF)]
    v8[1] = byte_1400F3500[16 * v5
                         + 512
                         + 16 * byte_1400F3400[16 * v5 + 16 * (v17 & 0xF) + (v13 & 0xF)]
                         + byte_1400F3500[16 * v5 + 16 * (v14 & 0xF) + (v15 & 0xF)]] | (16
                                                                                      * byte_1400F3500

    v6 += 1024i64;
    v8 += 4;
    v5 += 384i64;
```

每轮V6增长的值为`1024*4=0x1000`，所以第八轮结束的时V6应该为`0x8000`，对应汇编的`r12==0x8000`，在此之后插入缺陷数据



```
                              + 16 * byte_1400F3400[16 * v5 + 16 * (v17 & 0xF) +
                              + byte_1400F3500[16 * v5 + 16 * (v14 & 0xF) + (v15
      v6 += 1024i64;
      v8 += 4;
      v5 += 384i64;
      --v24;
    }
    while ( v24 );
    v7 = a1;
    v4 = &a1->m128i_i8[2];
  }
  while ( v6 < 0x9000 );
```



```
.text:00000001400052B6
.text:00000001400052BC 48 8B 5C 24 60                mov     rbx, [rsp+60h]
.text:00000001400052C1 48 8D 7B 02                   lea     rdi, [rbx+2]
.text:00000001400052C5 49 81 FC 00 90 00 00          cmp     r12, 9000h
.text:00000001400052CC 0F 8C 52 F9 FF FF             jl      loc_140004C24
.text:00000001400052CC
.text:00000001400052D2 48 8B CB                      mov     rcx, rbx
.text:00000001400052D5 E8 56 01 00 00                call    Row_Shift
```

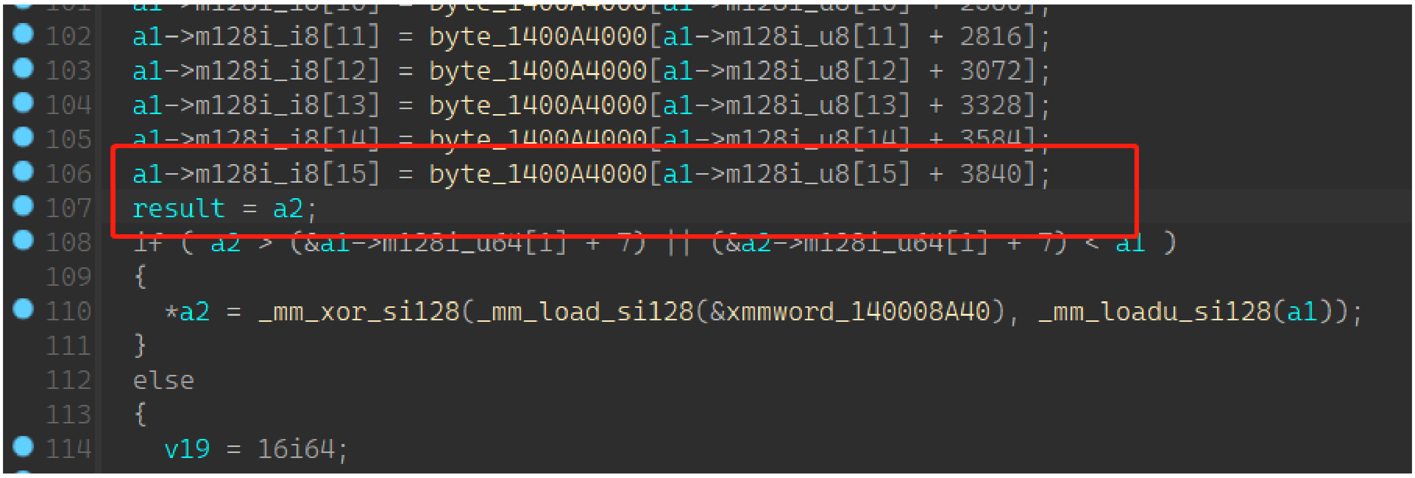
代码如下，这里的`b"\x00"`就是要插入的缺陷数据。

注：由于还需要获取正确的密文，所以第一次跑的时候不需要加上这一段`ql.mem.write(0x500000000 + index, b"\x00")`

```python
def hook_code(ql: Qiling):
    if ql.arch.regs.read("r12") == 0x8000:
        global index
        ql.mem.write(0x500000000 + index, b"\x00")
        index += 1
        # print(ql.mem.read(0x500000000, 16).hex())
    # print(ql.mem.read(0x500000000 + 0x10, 16))
    return
index_addr = 0x1400052C5
ql.hook_address(hook_code, index_addr)
```

获取密文

这就是加密部分的结束(暂时忽略最后的`Xor 0x66`)，所以只需要Hook该地址并打印出密文即可



```
a1->m128i_i8[10] = byte_1400A4000[a1->m128i_u8[10] + 2560];
a1->m128i_i8[11] = byte_1400A4000[a1->m128i_u8[11] + 2816];
a1->m128i_i8[12] = byte_1400A4000[a1->m128i_u8[12] + 3072];
a1->m128i_i8[13] = byte_1400A4000[a1->m128i_u8[13] + 3328];
a1->m128i_i8[14] = byte_1400A4000[a1->m128i_u8[14] + 3584];
a1->m128i_i8[15] = byte_1400A4000[a1->m128i_u8[15] + 3840];
result = a2;
if ( a2 > (&a1->m128i_u64[1] + 7) || (&a2->m128i_u64[1] + 7) < a1 )
{
  *a2 = _mm_xor_si128(_mm_load_si128(&xmmword_140008A40), _mm_loadu_si128(a1));
}
else
{
  v19 = 16i64;
```

```
def hook_enc(ql: Qiling):
    print(ql.mem.read(0x500000000, 16).hex())
    return

enc_after = 0x1400053CA
ql.hook_address(hook_enc, enc_after)
```

结果验证

正确的密文结果为 `e14d5d0ee27715df08b4152ba23da8e0` ，而在下标为0处插入缺陷数据的密文为 `e14d5d73e27708df0878152b843da8e0` ，符合文章提及的结果

```
start_addr = 0x140004BF0
end_addr = 0x14000542D
# e14d5d0ee27715df08b4152ba23da8e0
# e14d5d73e27708df0878152b843da8e0
ql.run(begin=start_addr, end=end_addr)
```

获取所有错误密文

接下来只需要逐位来插入缺陷数据即可得到所有的错误密文

```
from qiling import *
from qiling.const import QL_VERBOSE

index = 0
ql = Qiling(
    ["D:\\new\\x8664_windows\\m1_read.exe"],
    r"D:\\new\\x8664_windows",
    verbose=QL_VERBOSE.OFF,
)
```

```

for i in range(16):
    ql.run(begin=start_addr, end=end_addr)

```

结果如下

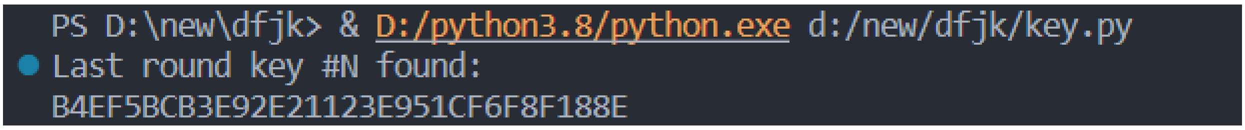
```
d24d5d0ee27715ac08b4bf2ba272a8e0
e14d5d73e27708df0878152b843da8e0
e14dd50ee23415df7fb4152ba23da890
e16f5d0e537715df08b415e7a23dc6e0
e11a5d0e057715df08b4151ba23d99e0
574d5d0ee277157508b4df2ba234a8e0
e14d5d49e27785df0840152bff3da8e0
e14db80ee2d215dfceb4152ba23da868
e14dc60ee2bf15dfc4b4152ba23da8bf
e1425d0e5e7715df08b415b6a23d4ce0
5d4d5d0ee277159608b42f2ba297a8e0
e14d5d6ce2773ddf089d152ba93da8e0
e14d5dcde2772adf084b152bba3da8e0
e14df40ee27115df96b4152ba23da881
e11b5d0e337715df08b41544a23df3e0
fa4d5d0ee27715af08b42e2ba2c2a8e0
```

获取第十轮密钥

得到十六个错误密文加上一个正确密文，再搭配上文章中的脚本即可得到第十轮密钥

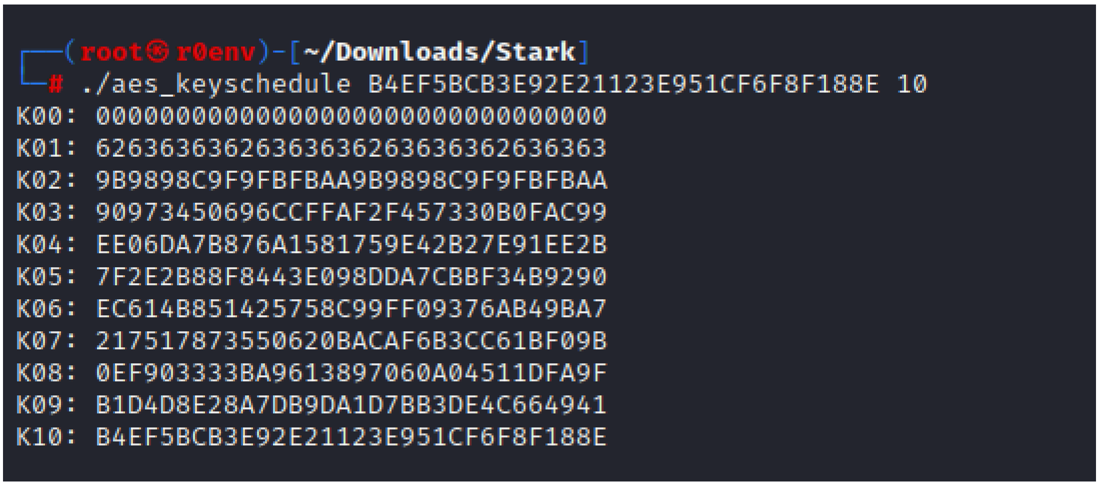
`B4EF5BCB3E92E21123E951CF6F8F188E`

```
import phoenixAES

with open("tracefile", "wb") as t:
    t.write(
        """
e14d5d0ee27715df08b4152ba23da8e0
d24d5d0ee27715ac08b4bf2ba272a8e0
e14d5d73e27708df0878152b843da8e0
e14dd50ee23415df7fb4152ba23da890
e16f5d0e537715df08b415e7a23dc6e0
e11a5d0e057715df08b4151ba23d99e0
574d5d0ee277157508b4df2ba234a8e0
e14d5d49e27785df0840152bff3da8e0
e14db80ee2d215dfceb4152ba23da868
e14dc60ee2bf15dfc4b4152ba23da8bf
e1425d0e5e7715df08b415b6a23d4ce0
5d4d5d0ee277159608b42f2ba297a8e0
e14d5d6ce2773ddf089d152ba93da8e0
e14d5dcde2772adf084b152bba3da8e0
e14df40ee27115df96b4152ba23da881
e11b5d0e337715df08b41544a23df3e0
fa4d5d0ee27715af08b42e2ba2c2a8e0
    """.encode(
            "utf8"
        )
    )
phoenixAES.crack_file("tracefile", verbose=0)

# B4EF5BCB3E92E21123E951CF6F8F188E

```



```
PS D:\new\dfjk> & D:/python3.8/python.exe d:/new/dfjk/key.py
Last round key #N found:
B4EF5BCB3E92E21123E951CF6F8F188E
```

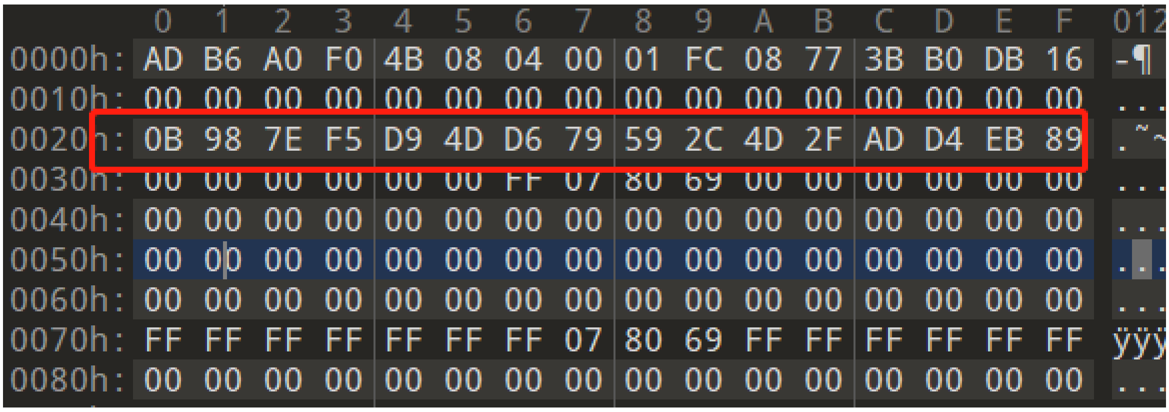
获取原始密钥



```
┌──(root㉿r0env)-[~/Downloads/Stark]
└─# ./aes_keyschedule B4EF5BCB3E92E21123E951CF6F8F188E 10
K00: 00000000000000000000000000000000
K01: 62636363626363636263636362636363
K02: 9B9898C9F9FBFBAA9B9898C9F9FBFBAA
K03: 90973450696CCFFAF2F457330B0FAC99
K04: EE06DA7B876A1581759E42B27E91EE2B
K05: 7F2E2B88F8443E098DDA7CBBF34B9290
K06: EC614B851425758C99FF09376AB49BA7
K07: 217517873550620BACAF6B3CC61BF09B
K08: 0EF903333BA9613897060A04511DFA9F
K09: B1D4D8E28A7DB9DA1D7BB3DE4C664941
K10: B4EF5BCB3E92E21123E951CF6F8F188E
```

Get Flag

取出密文进行解密即可



```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   012
0000h: AD B6 A0 F0 4B 08 04 00 01 FC 08 77 3B B0 DB 16   -¶
0010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...
0020h: 0B 98 7E F5 D9 4D D6 79 59 2C 4D 2F AD D4 EB 89   .˜~
0030h: 00 00 00 00 00 00 FF 07 80 69 00 00 00 00 00 00   ...
0040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...
0050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...
0060h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...
0070h: FF FF FF FF FF FF FF 07 80 69 FF FF FF FF FF FF   ÿÿÿ
0080h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...
```

```python
from Crypto.Cipher import AES

enc = bytearray(bytes.fromhex("0B 98 7E F5 D9 4D D6 79 59 2C 4D 2F AD D4 EB 89
enc = bytes([enc[i] ^ 0x66 for i in range(16)])
key = bytes.fromhex("00000000000000000000000000000000")
aes = AES.new(key=key, mode=AES.MODE_ECB)
print(aes.decrypt(enc))
#b'cddc8d28dabb4ea9'
```

完整脚本

```python
from qiling import *
from qiling.const import QL_VERBOSE

index = 0
ql = Qiling(
    ["D:\\new\\x8664_windows\\m1_read.exe"],
    r"D:\\new\\x8664_windows",
    verbose=QL_VERBOSE.OFF,
)

def hook_args(ql: Qiling):
    ql.mem.write(0x500000000, b"\x01" * 16)
    ql.arch.regs.write("rcx", 0x500000000)
    # print(ql.mem.read(0x500000000, 16))
    ql.mem.write(0x500000000 + 0x10, b"\x00" * 16)
    ql.arch.regs.write("rdx", 0x500000000 + 0x10)
    ql.mem.write(0x500000000 + 0x20, b"\x00" * 16)
    ql.arch.regs.write("rbx", 0x500000000 + 0x20)
    # print("Hook Success")
    return

def hook_code(ql: Qiling):
    if ql.arch.regs.read("r12") == 0x8000:
        global index
        ql.mem.write(0x500000000 + index, b"\x00")
        index += 1
        # print(ql.mem.read(0x500000000, 16).hex())
    # print(ql.mem.read(0x500000000 + 0x10, 16))
    return

def hook_enc(ql: Qiling):
    print(ql.mem.read(0x500000000, 16).hex())
    return

index_addr = 0x1400052C5
start_addr = 0x140004BF0
end_addr = 0x14000542D
enc_after = 0x1400053CA
ql.hook_address(hook_args, start_addr)
ql.hook_address(hook_code, index_addr)
ql.hook_address(hook_enc, enc_after)

# e14d5d0ee27715df08b4152ba23da8e0
# e14d5d73e27708df0878152b843da8e0
for i in range(16):
    ql.run(begin=start_addr, end=end_addr)

```

1