题目

- 名称：Babyffi
- 类型：rust [AES、ffi、Linux 基础 init，fini 作用、SMC、花指令]
- 难度：[入门 | 简单 | **中等** | 较难 | 极难]

分析过程

解压程序，发现有两个文件：

Babyffi1	2025/9/2 17:51	文件
libencode.so	2025/9/2 16:42	SO 文件

分析 Babyfii1 文件：

为 64 位 ELF 文件，用 IDA 打开，直接定位到 mian 函数：

```
__int64 __fastcall main(int a1, char **a2, char **a3)
{
  const char *v3; // rax

  puts("Please input right flag!");
  fgets(byte_40E0, 40, stdin);
  v3 = (const char *)encode1(byte_40E0, aTnysRxnsNFvFvu);
  strncmp(v3,
"794e5d4e59584e0b4a454f0b6d150b0a12525c55435e0d430d4b5b0d4b5b586a", 0x40uLL);
  return 0LL;
}
```

分析发现程序会将输入 `byte_40E0`，通过 `encode1` 加密后和 `794e5d4e59584e0b4a454f0b6d150b0a12525c55435e0d430d4b5b0d4b5b586a` 进行比较。

其中 `aTnysRxnsNFvFvu` 为：`++++++++++++.tnys rxns n fv fvuG`，分析 encode1 函数：

```
// attributes: thunk
__int64 __fastcall encode1(__int64 a1, __int64 a2)
{
  return encode1(a1, a2);
}
```

这里发现其为库函数，结合该名字，猜测该函数的实现应该在 `libencode.so` 库里，分析该函数；用 IDA 打开 .so 文件，定位到 `encode1` 函数：

```
__int64 __fastcall encode1(const char *a1, const char *a2)
{
  size_t v2; // rax
  __int64 v3; // r15
  unsigned __int64 v4; // r12
  size_t v5; // rax
  __int64 v6; // r13
  __int64 i; // rbp
  char v8; // r14
  __int64 result; // rax
  unsigned __int64 v10; // rbx
  __int64 v11; // rdx
  __int64 v12; // r14
  void **v13; // r8
  _BYTE v14[24]; // [rsp+0h] [rbp-88h] BYREF
  __int64 v15; // [rsp+18h] [rbp-70h]
  __int128 v16; // [rsp+28h] [rbp-60h] BYREF
  __int64 v17; // [rsp+38h] [rbp-50h]
  __int128 v18; // [rsp+40h] [rbp-48h] BYREF
  unsigned __int64 v19; // [rsp+50h] [rbp-38h]

  v2 = strlen(a1);
  core::ffi::c_str::CStr::to_str::hbc05b41c671bd761(v14, a1, v2 + 1);
  if ( *(_DWORD *)v14 == 1 )
  {
    v13 = &off_542B0;
    goto LABEL_26;
  }
  v3 = *(_QWORD *)&v14[8];
  v4 = *(_QWORD *)&v14[16];
  v5 = strlen(a2);
  core::ffi::c_str::CStr::to_str::hbc05b41c671bd761(v14, a2, v5 + 1);
  if ( *(_DWORD *)v14 == 1 )
  {
    v13 = &off_542C8;
LABEL_26:
    v16 = *(_OWORD *)&v14[8];
    core::result::unwrap_failed::he8e27e02739cd3d2(&unk_4705D, 43LL, &v16,
&unk_541D0, v13);
  }
  v6 = *(_QWORD *)&v14[8];
  *(_QWORD *)&v16 = 0LL;
```

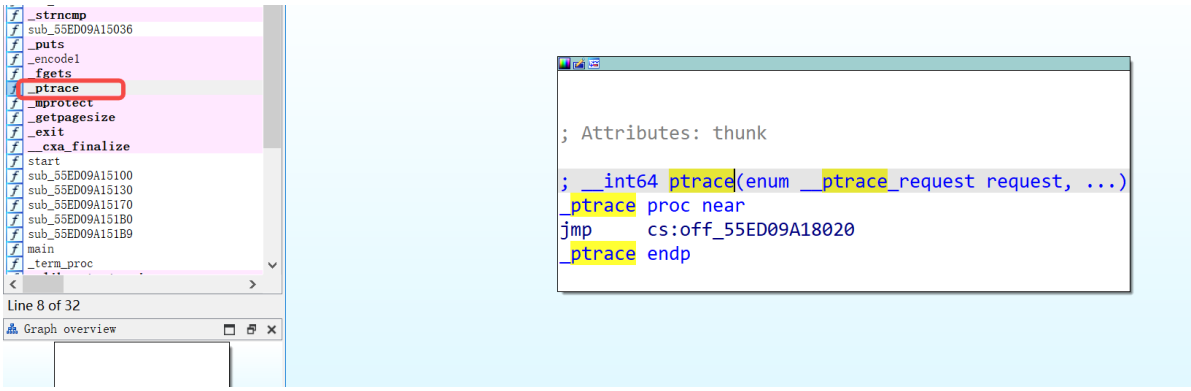
```
  *((_QWORD *)&v16 + 1) = 1LL;
  v17 = 0LL;
  if ( *(_QWORD *)&v14[16] < v4 )
    v4 = *(_QWORD *)&v14[16];
  if ( v4 )
  {
    for ( i = 0LL; i != v4; v17 = i )
    {
      v8 = *(_BYTE *)(v3 + i) ^ *(_BYTE *)(v6 + i);
      if ( i == (_QWORD)v16 )
        alloc::raw_vec::RawVec$LT$T$C$A$GT$::grow_one::h3090820664dc9481(&v16,
&off_54298);
      *(_BYTE *)(*((_QWORD *)&v16 + 1) + i++) = v8;
    }
  }
  *(_QWORD *)&v14[16] = v17;
  *(_OWORD *)v14 = v16;
  _$LT$T$u20$as$u20$hex..ToHex$GT$::encode_hex::h959eee524f1a40fc(&v18, v14);
  if ( *(_QWORD *)v14 )
    RNvCscSpY9JukOHT_7___rustc14___rust_dealloc(*(_QWORD *)&v14[8], *(_QWORD
*)v14, 1LL);
  result = *((_QWORD *)&v18 + 1);
  v10 = v19;
  if ( v19 <= 0xF )
  {
    if ( v19 )
    {
      v11 = 0LL;
      while ( *(_BYTE *)(*((_QWORD *)&v18 + 1) + v11) )
      {
        if ( v19 == ++v11 )
          goto LABEL_22;
      }
      goto LABEL_20;
    }
LABEL_22:
    *(_QWORD *)&v14[16] = v19;
    *(_OWORD *)v14 = v18;
    return
alloc::ffi::c_str::CString::_from_vec_unchecked::h6fd1f8c4256369f7(v14);
  }
  v12 = *((_QWORD *)&v18 + 1);
  if ( (core::slice::memchr::memchr_aligned::h9672377a6eaa3e7e(0LL, *((_QWORD
*)&v18 + 1), v19) & 1) == 0 )
    goto LABEL_22;
  result = v12;
LABEL_20:
  if ( !__OFSUB__(-(__int64)v18, 1LL) )
  {
    *(_QWORD *)v14 = v18;
    *(_QWORD *)&v14[8] = result;
    *(_QWORD *)&v14[16] = v10;
    v15 = v11;
    core::result::unwrap_failed::he8e27e02739cd3d2(&unk_4705D, 43LL, v14,
&off_541B0, &off_542E0);
```

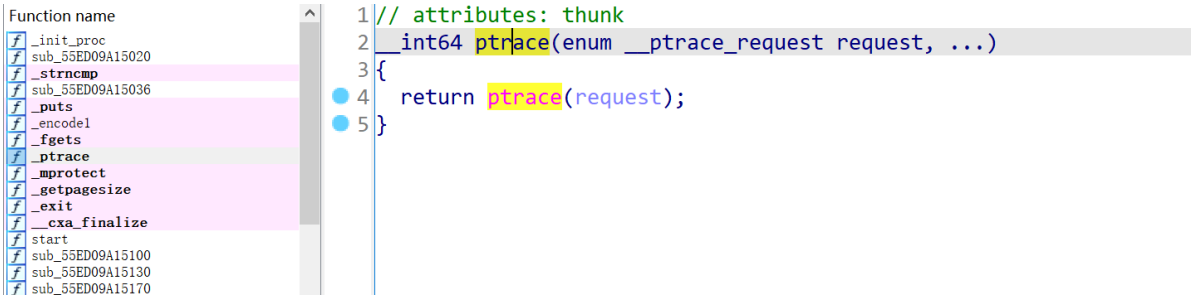
```
    }
    return result;
  }
```

发现为 rust 写的代码，出题人真坏，搞 rust 的代码。尝试调试该代码：

```
┌──(za0㊙ kali)-[/tmp/Babyffi]
└─$ ./linux_server64
IDA Linux 64-bit remote debug server(ST) v7.7.27. Hex-Rays (c) 2004-2022
Listening on 0.0.0.0:23946...
2025-09-02 18:52:32 [1] Accepting connection from 192.168.64.1...
Incorrect!
2025-09-02 18:52:33 [1] Closing connection from 192.168.64.1...
```

调试时，发现一直断，猜测可能存在反调试，查看函数发现存在 `ptrace` 函数：





跟一下，定位到反调试相关代码：

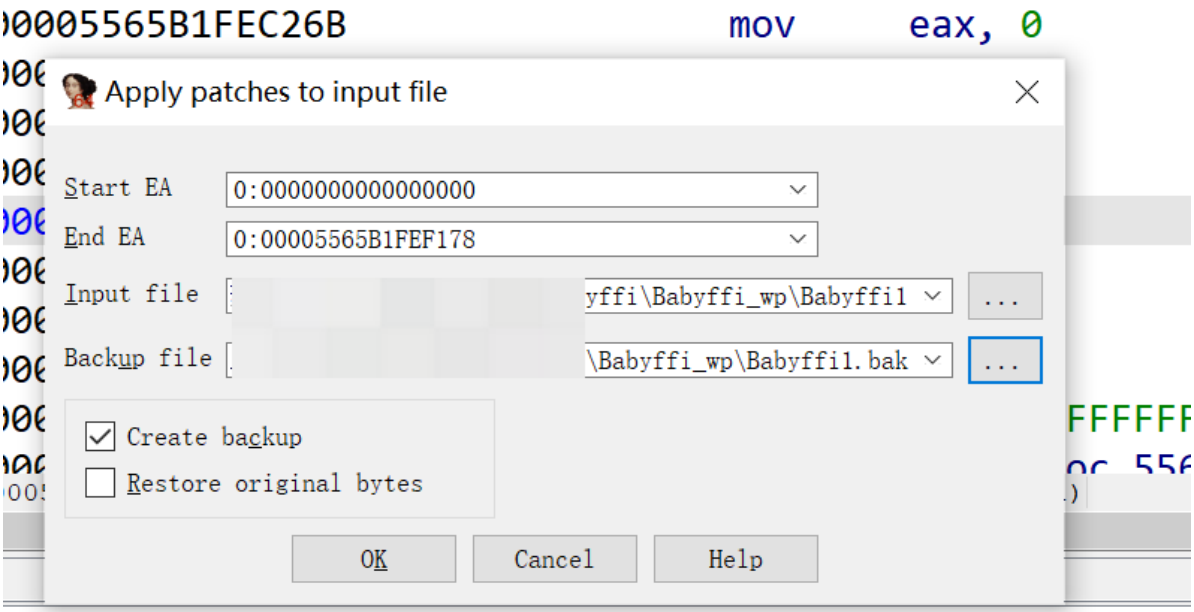
```
__int64 sub_55ED09A151B9()
{
  int v0; // eax
  int v1; // eax
  __int64 result; // rax
  void *addr; // [rsp+0h] [rbp-20h]
  char *v4; // [rsp+18h] [rbp-8h]

  v4 = (char *)&loc_55ED09A15288;
  addr = (void *)(-getpagesize() & (unsigned __int64)&loc_55ED09A15288);
  v0 = getpagesize();
  mprotect(addr, v0, 7);
  while ( &byte_55ED09A153BB[2] >= v4 )
    *v4++ ^= 0x25u;
  v1 = getpagesize();
  mprotect(addr, v1, 5);
  result = ptrace(PTRACE_TRACEME, 0LL, 0LL, 0LL);
  if ( result == -1 )
    exit(0);
  return result;
}
```

```
result = ptrace(PTRACE_TRACEME, 0LL, 0LL, 0LL);
if ( result == -1 )
  exit(0);
```

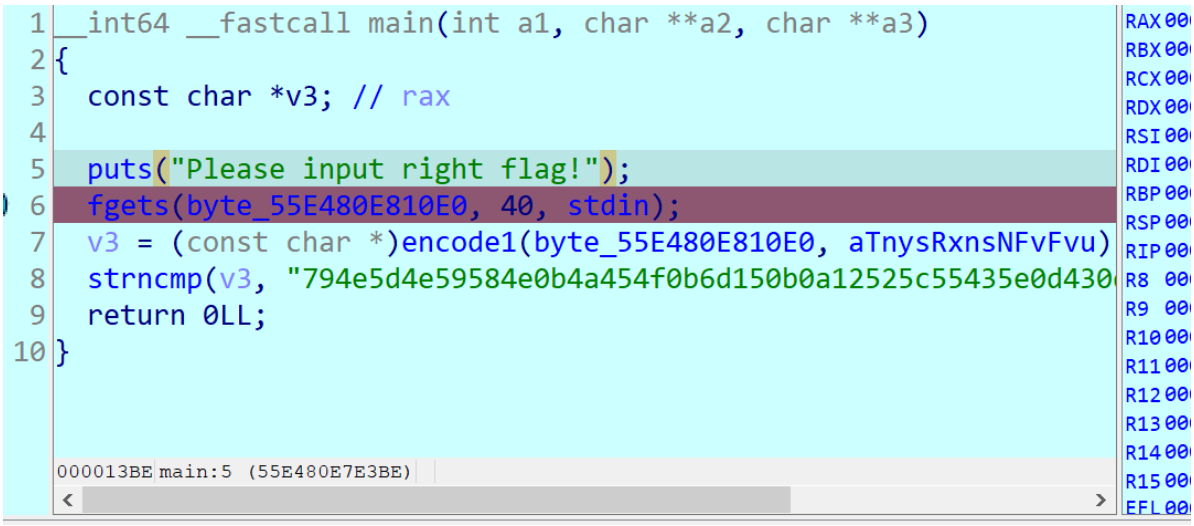
将程序相关调用 nop 掉，阻止程序反调试：

```
55ED09A1524F    mov     rdi, rax        ; addr
55ED09A15252    call    _mprotect
55ED09A15257    mov     ecx, 0
55ED09A1525C    mov     edx, 0
55ED09A15261    mov     esi, 0
55ED09A15266    mov     edi, 0          ; request
55ED09A1526B    mov     eax, 0
55ED09A15270    nop                     ; Keypatch modified this from:
55ED09A15270                            ;   call _ptrace
55ED09A15270                            ; Keypatch padded NOP to next boundary: 4
55ED09A15271    nop
55ED09A15272    nop
55ED09A15273    nop
55ED09A15274    nop
55ED09A15275    cmp     rax, 0FFFFFFFFFFFFFFFFh
```

保存程序修改，并再次尝试调试（注意将修改后的程序复制到 Linux 下）：



程序可以正常调试：



调试 encode1 函数：

第一个函数为，其会将字符串转换为类似于数组的数据：

经过分析，发现该函数基本上就是将传递的两个字符串进行异或的函数：

```
{
  for ( i = 0LL; i != v4; v16 = i )
  {
    v8 = *(_BYTE *)(v3 + i) ^ *(_BYTE *)(v6 + i);
    if ( i == (_QWORD)v15 )
      ((void (__fastcall *)(__int128 *, void *))ZN5alloc7raw_vec19RawVec_LT_T_C_A_GT_
        &v15,
        &unk_7FB1066E2298);
    *(_BYTE *)(*((_QWORD *)&v15 + 1) + i++) = v8;
  }
```

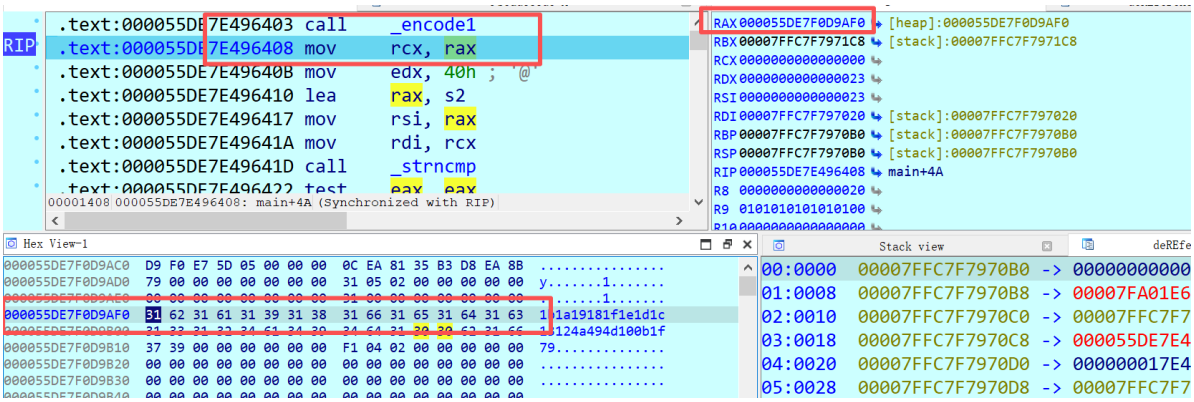
v3:

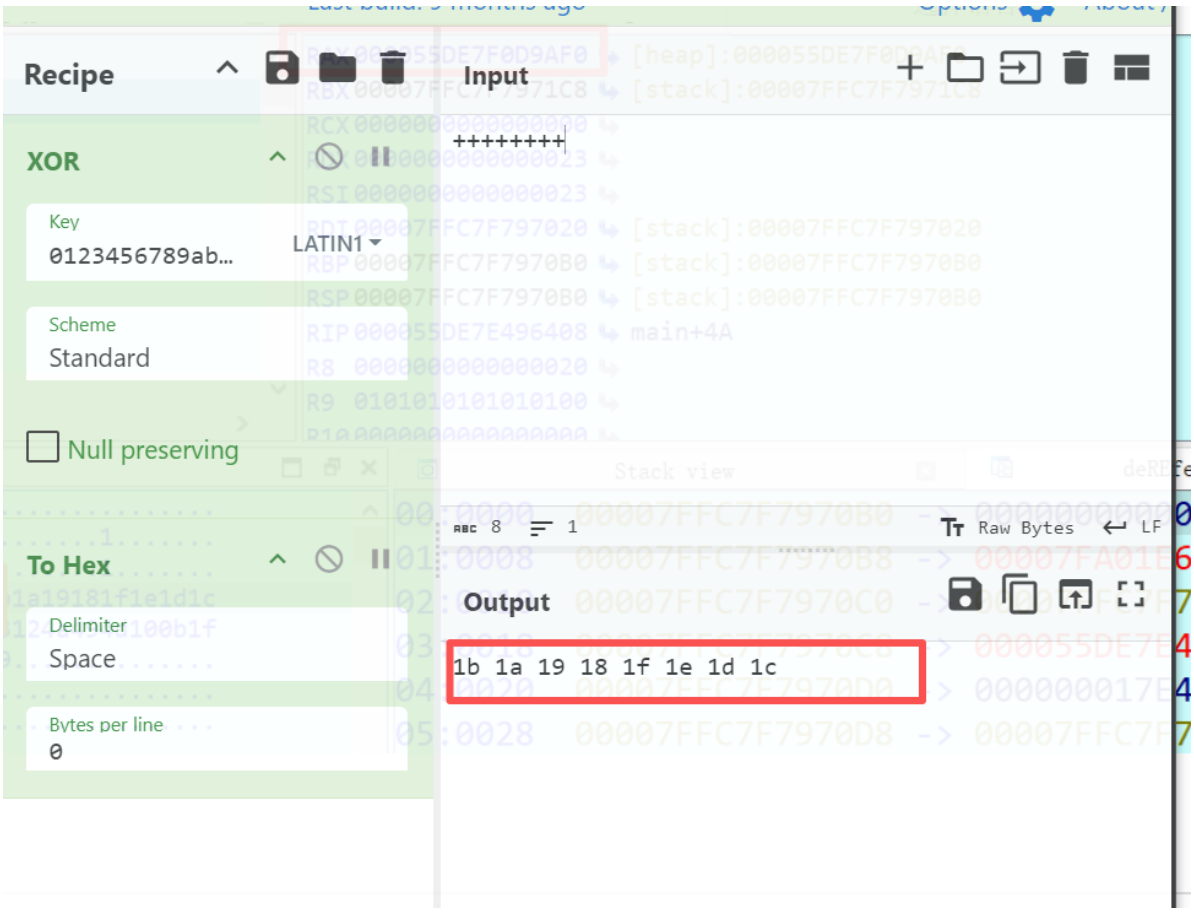
```
0E810C9 align 20h
0E810E0 ; char byte_55E480E810E0[40]
0E810E0 byte_55E480E810E0 db 31h, 32h, 33h, 33h, 34h, 35h, 36h, 37h, 38h, 39h, 30h, 6
0E810E0                                                 ; DATA XREF: main+22↑o
0E810E0                                                 ; main+3B↑o
0E810E0 db 64h, 65h, 66h, 67h, 68h, 0Ah, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0E810E0 db 0, 0, 0, 0, 0, 0, 0
```

v6:

```
ta:000055E480E81068                                     ; DATA XREF: sub_55E48
ta:000055E480E81068                                     ; .data:off_55E480E810
ta:000055E480E81070 align 20h
ta:000055E480E81080 aTnysRxnsNFvFvu db '++++++++++++.tnys rxns n fv fvuG',0
ta:000055E480E81080                                     ; DATA XREF: main+31↑o
ta:000055E480E81080                                     ; [stack]:00007FFC4007
ta:000055E480E81080 _data ends
```

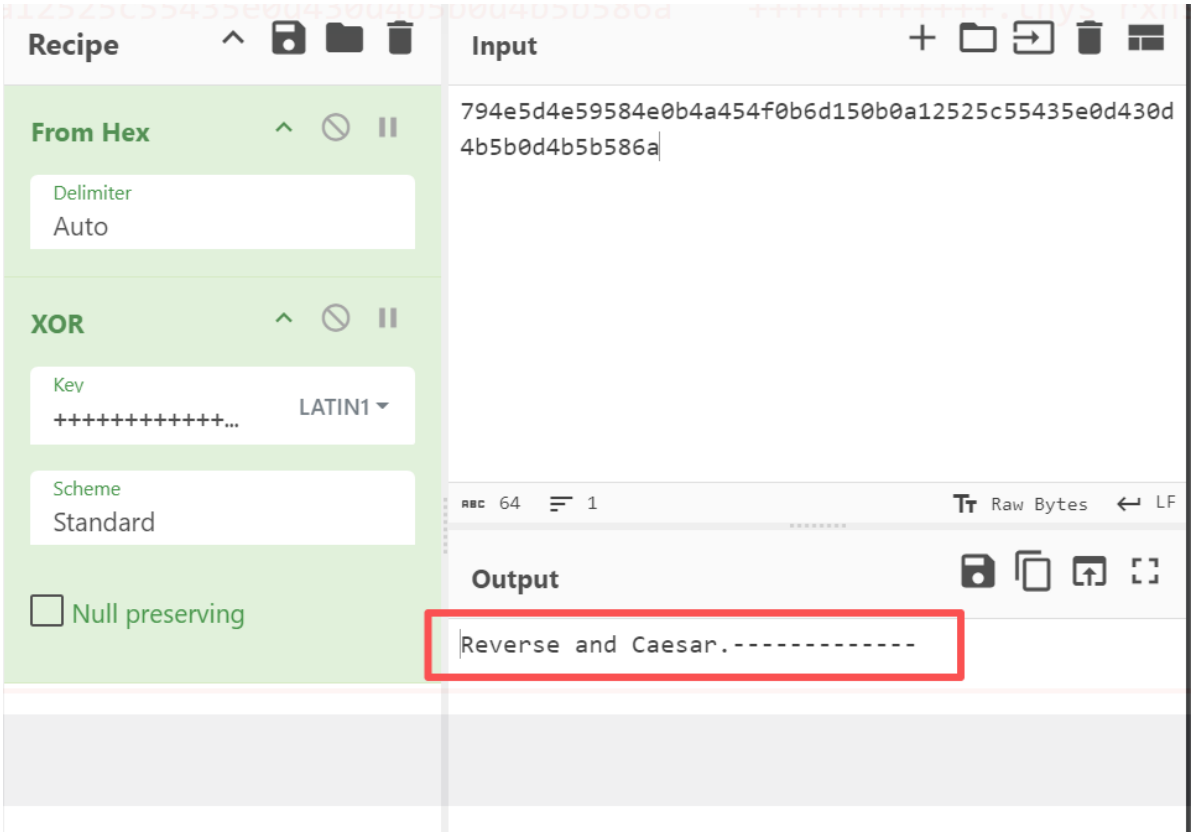
分析发现其会直接返回 hex 字符串：





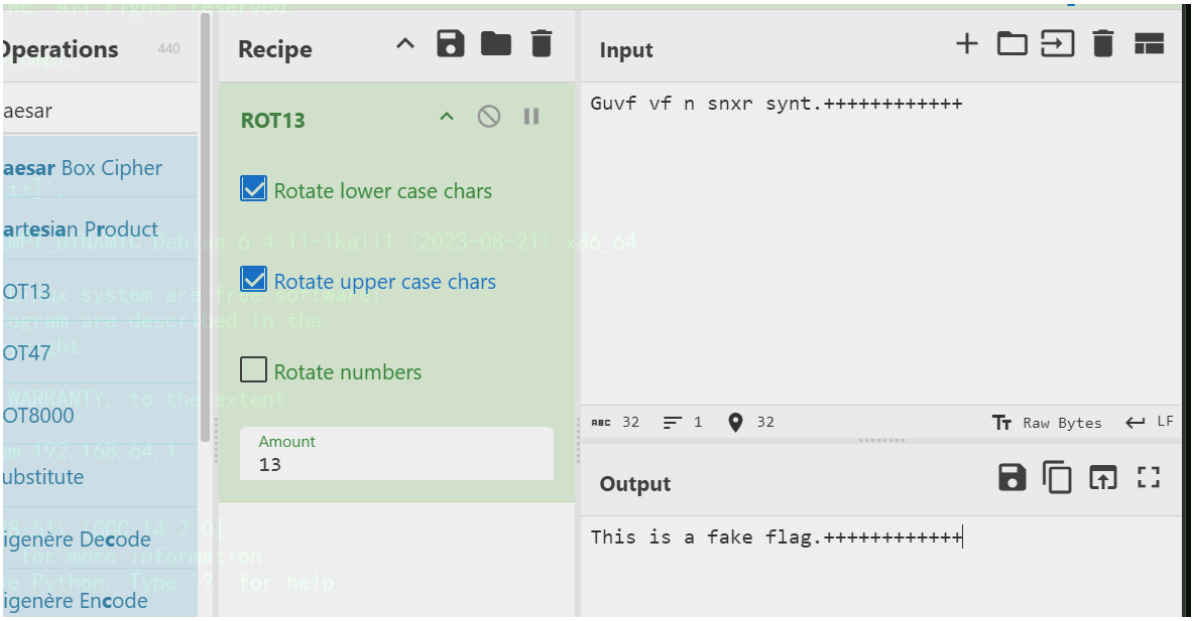
尝试解密 hex 并进行 xor：

input = `794e5d4e59584e0b4a454f0b6d150b0a12525c55435e0d430d4b5b0d4b5b586a` ^ `++++++++++++.tnys rxns n fv fvuG`




```
Reverse and Caesar.----------
```

尝试提交发现存在问题，发现 `++++++++++++.tnys rxns n fv fvuG` 很可疑猜测解密内容，可能是其提示，按照提示翻转并解密凯撒：



解密出来发现是一个假的密钥。

回想前面找到过 ptrace 的代码：

```
__int64 sub_55DE7E4961B9()
{
  int v0; // eax
  int v1; // eax
  void *addr; // [rsp+0h] [rbp-20h]
  char *v4; // [rsp+18h] [rbp-8h]

  v4 = (char *)&loc_55DE7E496288;
  addr = (void *)(-getpagesize() & (unsigned __int64)&loc_55DE7E496288);
  v0 = getpagesize();
  mprotect(addr, v0, 7);
  while ( &byte_55DE7E4963BB[2] >= v4 )
    *v4++ ^= 0x25u;
  v1 = getpagesize();
  mprotect(addr, v1, 5);
  return 0LL;
}
```

发现以下位置很像 smc 自解密模板，尝试解密该内容：

```
  mprotect(addr, v0, 7);
  while ( &byte_55DE7E4963BB[2] >= v4 )
    *v4++ ^= 0x25u;
  v1 = getpagesize();
  mprotect(addr, v1, 5);
```

解密前：

```
.text:000055DE7E496288                                          ; .fini_array:000055DE7E498DB8↓
.text:000055DE7E496288 ; __unwind { // 55DE7E495000
.text:000055DE7E496288                 jo      short loc_55DE7E4962F7
.text:000055DE7E49628A                 lodsb
.text:000055DE7E49628B                 shr     byte ptr [rbp-5Ch], 0C9h
.text:000055DE7E49628F                 mov     ch, 25h ; '%'
.text:000055DE7E496291                 and     eax, 419D6D25h
.text:000055DE7E496291 ; ---------------------------------------------------------------
.text:000055DE7E496296                 dw 1217h
.text:000055DE7E496298                 dq 1C9F6D1512174047h, 6D11174744114714h, 6DBD70AC6DB560
.text:000055DE7E496298                 dq 461214444717479Dh, 17401515169F6D12h, 6D8560AC6D4115
.text:000055DE7E496298                 dq 1140479D6D8D70ACh, 1D9F6D1714104414h, 6D151316164012
.text:000055DE7E496298                 dq 6D9D70AC6D9560ACh, 401D12151D16409Dh
.text:000055DE7E4962F0                 db 40h, 6Dh, 9Fh, 1Dh, 13h, 40h, 47h
.text:000055DE7E4962F7 ; ---------------------------------------------------------------
```

解密脚本:

```python
start = 0x1288

for i in range(0x13BD-0x1288+1):
    value = idc.get_wide_byte(start+i) ^ 0x25
    ida_bytes.patch_byte(start+i, value)
print("END!")
```

解密后:

```
.text:000055DE7E496288                                          ; .fini_array:000055DE7E498DB8↓o
.text:000055DE7E496288 ; __unwind { // 55DE7E495000
.text:000055DE7E496288                 push    rbp
.text:000055DE7E496289                 mov     rbp, rsp
.text:000055DE7E49628C                 sub     rsp, 90h
.text:000055DE7E496293                 mov     rax, 3037326562373264h
.text:000055DE7E49629D                 mov     rdx, 3432626134623139h
.text:000055DE7E4962A7                 mov     [rbp-70h], rax
.text:000055DE7E4962AB                 mov     [rbp-68h], rdx
.text:000055DE7E4962AF                 mov     rax, 3763373161623262h
.text:000055DE7E4962B9                 mov     rdx, 6430613265303033h
.text:000055DE7E4962C3                 mov     [rbp-60h], rax
.text:000055DE7E4962C7                 mov     [rbp-58h], rdx
.text:000055DE7E4962CB                 mov     rax, 3231356131346562h
.text:000055DE7E4962D5                 mov     rdx, 3036333365373438h
.text:000055DE7E4962DF                 mov     [rbp-50h], rax
```

但是解密后发现存在 花指令:

```
.text:000055DE7E49633F                 call    loc_55DE7E496345
.text:000055DE7E49633F ; ---------------------------------------------------------------
.text:000055DE7E496344                 db  83h
.text:000055DE7E496345 ; ---------------------------------------------------------------
.text:000055DE7E496345
.text:000055DE7E496345 loc_55DE7E496345:                       ; CODE XREF: .text:000055
.text:000055DE7E496345                 add     qword ptr [rsp], 8
.text:000055DE7E49634A                 retn
.text:000055DE7E49634A ; ---------------------------------------------------------------
.text:000055DE7E49634B                 db 0F3h
```

一个很典型的花指令，经过分析需要在 `0x00055DE7E496344 + 8` 位置处恢复花指令，恢复后的效果:

```
.text:000055DE7E49633F                 call    loc_55DE7E496345
.text:000055DE7E49633F ; ---------------------------------------------------------------------------
.text:000055DE7E496344                 db  83h
.text:000055DE7E496345 ; ---------------------------------------------------------------------------
.text:000055DE7E496345
.text:000055DE7E496345 loc_55DE7E496345:                       ; CODE XREF: .text:000055DE7E49633F
.text:000055DE7E496345                 add     qword ptr [rsp], 8
.text:000055DE7E49634A                 retn
.text:000055DE7E49634A ; ---------------------------------------------------------------------------
.text:000055DE7E49634B                 db 0F3h
.text:000055DE7E49634C ; ---------------------------------------------------------------------------
.text:000055DE7E49634C                 lea     rax, [rbp-70h]
.text:000055DE7E496350                 mov     rsi, rax
.text:000055DE7E496353                 lea     rax, byte_55DE7E4990E0
.text:000055DE7E49635A                 mov     rdi, rax
```

将中间无用指令 nop 掉，发现后边还有一条花指令：

```
text:000055DE7E496362                 mov     [rbp-4], eax
text:000055DE7E496365                 push    rbx
text:000055DE7E496366                 xor     rbx, rbx
text:000055DE7E496369                 jz      short loc_55DE7E49636D
text:000055DE7E49636B                 or      al, 22h
text:000055DE7E49636D
text:000055DE7E49636D loc_55DE7E49636D:                       ; CODE XREF: .text:00005
text:000055DE7E49636D                 pop     rbx
text:000055DE7E49636E                 mov     rax, 2174636572726F43h
```

但是基本不影响分析，直接选中该函数所有代码，按 p 定义为函数：

```
int sub_55DE7E496288()
{
  _BYTE v1[20]; // [rsp+Ch] [rbp-84h] BYREF
  char v2[108]; // [rsp+20h] [rbp-70h] BYREF
  int v3; // [rsp+8Ch] [rbp-4h]

  strcpy(v2, "d27be27091b4ab24b2ba17c7300e2a0dbe41a512847e3360e38078ee86eb656cdb7a25527f7f817257031cf8e571e9b4")
  v3 = encode2(byte_55DE7E4990E0, v2);
  strcpy(&v1[11], "Correct!");
  strcpy(v1, "Incorrect!");
  if ( v3 == 1 )
    return puts(&v1[11]);
  else
    return puts(v1);
}
```

分析发现该逻辑很简单了，程序会将 `d27be27091b4ab24b2ba17c7300e2a0dbe41a512847e3360e38078ee86eb656cdb7a25527f7f817257031cf8e571e9b4` 复制给 v2，经过 encode2 函数加密后，判断返回是否为 1，如果为 1 则提示正确，否则则提示错误。所以基本得知，`d27be27091b4ab24b2ba17c7300e2a0dbe41a512847e3360e38078ee86eb656cdb7a25527f7f817257031cf8e571e9b4` 为密文，`encode2` 为加密函数，并且分析发现 `byte_55DE7E4990E0` 即为输入数据：

```
fgets(byte_55DE7E4990E0, 40, stdin);
encode1();
```

所以可知，需要重点分析 `encode2` 函数：

```
_BOOL8 __fastcall encode2(const char *a1, const char *a2)
{
  size_t v2; // rax
  const void *v3; // r13
  signed __int64 v4; // rbx
  size_t v5; // rax
  const void *v6; // r14
```

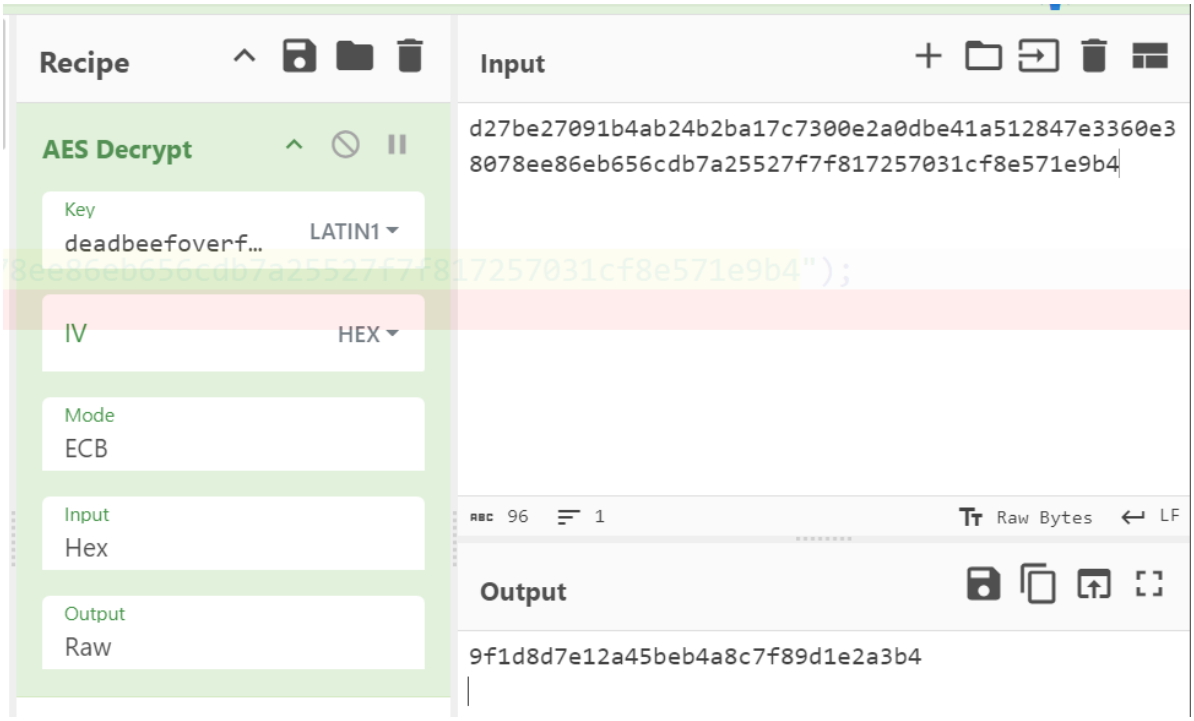
```
  size_t v7; // r15
  __int64 *v8; // r12
  __int64 v9; // rbp
  __int64 v10; // rax
  void *v11; // r12
  _BOOL4 v12; // ebx
  void **v14; // r8
  void *v15; // rcx
  __int64 v16; // [rsp+8h] [rbp-5E0h] BYREF
  void *s1; // [rsp+10h] [rbp-5D8h]
  __int64 v18; // [rsp+18h] [rbp-5D0h]
  __int64 v19; // [rsp+20h] [rbp-5C8h] BYREF
  char src[712]; // [rsp+28h] [rbp-5C0h] BYREF
  __int128 dest[47]; // [rsp+2F0h] [rbp-2F8h] BYREF

  v2 = strlen(a1);
  core::ffi::c_str::CStr::to_str::hbc05b41c671bd761(&v19, a1, v2 + 1);
  if ( (_DWORD)v19 == 1 )
  {
    dest[0] = *(_OWORD *)src;
    v14 = &off_542F8;
LABEL_20:
    v15 = &unk_541D0;
    v8 = (__int64 *)dest;
    goto LABEL_21;
  }
  v3 = *(const void **)src;
  v4 = *(_QWORD *)&src[8];
  v5 = strlen(a2);
  core::ffi::c_str::CStr::to_str::hbc05b41c671bd761(&v19, a2, v5 + 1);
  if ( (_DWORD)v19 == 1 )
  {
    dest[0] = *(_OWORD *)src;
    v14 = &off_54310;
    goto LABEL_20;
  }
  v6 = *(const void **)src;
  v7 = *(_QWORD *)&src[8];
  v8 = &v19;
  cipher::block::NewBlockCipher::new_from_slice::h7f8be4bd7ed56014(
    &v19,

 "deadbeefoverflow0123456789abcdef/rustc/17067e9ac6d7ecb70e50f92c1944e545188d235
9/library/alloc/src/string.rs/home/za0/.cargo/registry/src/index.crates.io-
1949cf8c6b5b557f/block-modes-0.8.1/src/traits.rsenough space for padding is
allocated",
    16LL);
  if ( (v19 & 1) != 0 )
  {
    v14 = &off_54280;
    v15 = &unk_54190;
LABEL_21:
    core::result::unwrap_failed::he8e27e02739cd3d2(&unk_4705D, 43LL, v8, v15,
v14);
  }
```

```
  memcpy(dest, &src[8], 0x2C0uLL);
  if ( v4 < 0 )
  {
    v9 = 0LL;
    goto LABEL_24;
  }
  if ( v4 )
  {
    v9 = 1LL;
    v10 = RNvCscSpY9JukOHT_7___rustc12___rust_alloc(v4, 1LL);
    if ( v10 )
    {
      v11 = (void *)v10;
      goto LABEL_9;
    }
LABEL_24:
    alloc::raw_vec::handle_error::h5b039796a4ecc373(v9, v4, &off_54268);
  }
  v11 = &dword_0 + 1;
LABEL_9:
  memcpy(v11, v3, v4);
  block_modes::traits::BlockMode::encrypt_vec::hc081db034e3fa9df(&v19, dest,
v11, v4);
  _$LT$T$u20$as$u20$hex..ToHex$GT$::encode_hex::h959eee524f1a40fc(&v16, &v19);
  if ( v19 )
    RNvCscSpY9JukOHT_7___rustc14___rust_dealloc(*(_QWORD *)src, v19, 1LL);
  if ( v4 )
    RNvCscSpY9JukOHT_7___rustc14___rust_dealloc(v11, v4, 1LL);
  v12 = 0;
  if ( v18 == v7 )
    v12 = bcmp(s1, v6, v7) == 0;
  if ( v16 )
    RNvCscSpY9JukOHT_7___rustc14___rust_dealloc(s1, v16, 1LL);
  return v12;
}
```

分析发现该代码为 AES 代码，其密钥为：

```
cipher::block::NewBlockCipher::new_from_slice::h7f8be4bd7ed56014(
  &v19,
  "deadbeefoverflow0123456789abcdef/rustc/17067e9ac6d7ecb70e50f92c1944e545188d2359/l
  16LL);
if ( (v19 & 1) != 0 )
{
  v14 = &off_54280;
```

解密 AES 得：




```
9f1d8d7e12a45beb4a8c7f89d1e2a3b4
```

验证 flag：

exp