# fifo

- 首先直接執行 fifo 觀察行為, 發現沒有任何反應
- 以 IDA pro / gdb 分析, 接下來若未特別標註的記憶體位址, 皆為程式的 offset
- 在 0x149d, 0x14bb, 0x14d9, 0x14f9 呼叫了同一個 function, 以 gdb 在上面設斷點執行觀察第一個參數內容, 發現執行後, 第一個參數內容被解密成明文字串, 因此能判斷呼叫的 function 在做字串解密
- 因此在 IDA 中能先把一些資訊記錄下來, 包含了 function rename、參數型別之類的資訊, 如下圖
  - Before:

```
28  v10 = 0xCE5F6D84CCD05DB3LL;
29  v11 = 0x7F9B9A42B2CADA5DLL;
30  v12 = 0xA24Au;
31  v13 = 0;
32  v17 = 0xCE5F6D84CCD05DB3LL;
33  v18 = 0x7F9B9A42B2CADA5DLL;
34  v19 = 0x91AB22CDDD92A24ALL;
35  v20 = 0x7388DA9D25D8CCF8LL;
36  v21 = 0;
37  v14 = 0xD1596484CCD05DB3LL;
38  v15 = 0x62989648A4D1CC52LL;
39  v16 = 0xD8FE46;
40  sub_12E9(&v10, 18, &unk_2020, 0x41);
41  sub_12E9(&v17, 32, &unk_2020, 0x41);
42  sub_12E9(&v14, 19, &unk_2020, 0x41);
43  sub_12E9(&unk_4040, dword_4020, &unk_2020, 65);
```

  - After:

```
21  *str_v10 = 0xCE5F6D84CCD05DB3LL;
22  *&str_v10[8] = 0x7F9B9A42B2CADA5DLL;
23  *&str_v10[16] = 0xA24Au;
24  v11 = 0;
25  *str_v17 = 0xCE5F6D84CCD05DB3LL;
26  *&str_v17[8] = 0x7F9B9A42B2CADA5DLL;
27  *&str_v17[16] = 0x91AB22CDDD92A24ALL;
28  *&str_v17[24] = 0x7388DA9D25D8CCF8LL;
29  v14 = 0;
30  *str_v14 = 0xD1596484CCD05DB3LL;
31  *&str_v14[8] = 0x62989648A4D1CC52LL;
32  *&str_v14[16] = 0xD8FE46;
33  str_decrypt(str_v10, 0x12, &unk_2020, 0x41);
34  str_decrypt(str_v17, 0x20, &unk_2020, 0x41);
35  str_decrypt(str_v14, 0x13, &unk_2020, 0x41);
36  str_decrypt(str_unk_4040, num_3880, &unk_2020, 0x41);
```

- 解讀 str_decrypt

- Before:

```
__int64 __fastcall str_decrypt(char *enc, int enc_len, char *key, int key_len)
{
  __int64 result; // rax
  __int64 v5; // [rsp-8h] [rbp-8h]

  __asm { endbr64 }
  *(&v5 - 3) = enc;
  *(&v5 - 7) = enc_len;
  *(&v5 - 5) = key;
  *(&v5 - 8) = key_len;
  *(&v5 - 2) = 0;
  for ( *(&v5 - 1) = 0; ; ++*(&v5 - 1) )
  {
    result = *(&v5 - 1);
    if ( result >= *(&v5 - 7) )
      break;
    **(&v5 - 3) ^= *((*(&v5 - 2))++ % *(&v5 - 8) + *(&v5 - 5));
    **(&v5 - 3) ^= *((*(&v5 - 2))++ % *(&v5 - 8) + *(&v5 - 5));
    **(&v5 - 3) ^= *((*(&v5 - 2))++ % *(&v5 - 8) + *(&v5 - 5));
    *(*(&v5 - 3))++ ^= *((*(&v5 - 2))++ % *(&v5 - 8) + *(&v5 - 5));
  }
  return result;
}
```

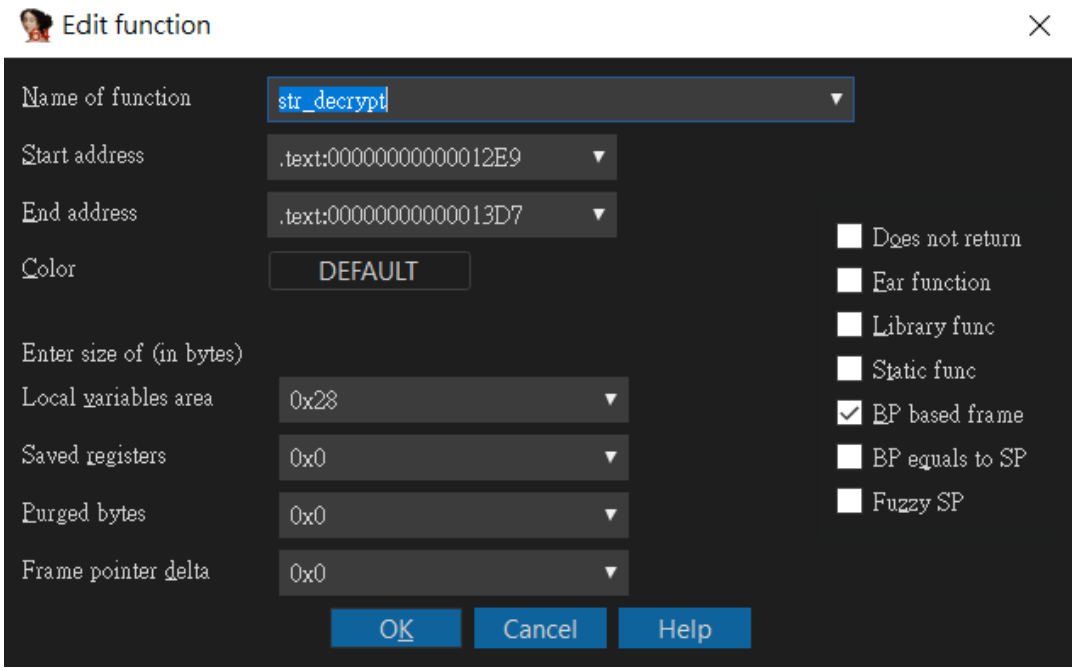
- 發現 decompile 結果都用 v5 的 offset 來定義變數, 由於此 function 為 leaf function, 沒有再呼叫其他 functions 了, 這類的 function 不需要再通過減 rsp 來分配一塊區域用於放置區域變數, 直接以 `rbp - xxx` 的形式即可, 可以觀察 prolog:

```
str_decrypt     proc near

; __unwind {
                endbr64
                push    rbp
                mov     rbp, rsp
                mov     [rbp-18h], rdi
```

  並沒有 `sub rsp, xxx` 的指令, 而 IDA decompiler 預設以 `rsp + xxx` 的形式來辨認區域變數的方式就發生了點誤會

- 對著 function name `str_decrypt` 右鍵後, 按 edit function, 將此 function 標為 `BP based frame`




- 再次 decompile, IDA 就能改以 `rbp - xxx` 的形式辨識區域變數

- After:

```
__int64 __fastcall str_decrypt(char
{
  __int64 result; // rax
  char *v5; // [rsp+10h] [rbp-18h]
  int v6; // [rsp+20h] [rbp-8h]
  int v7; // [rsp+20h] [rbp-8h]
  int i; // [rsp+24h] [rbp-4h]

  __asm { endbr64 }
  v5 = enc;
  v6 = 0;
  for ( i = 0; ; ++i )
  {
    result = i;
    if ( i >= enc_len )
      break;
    *v5 ^= key[v6 % key_len];
    v7 = v6 + 1;
    *v5 ^= key[v7++ % key_len];
    *v5 ^= key[v7++ % key_len];
    *v5 ^= key[v7 % key_len];
    v6 = v7 + 1;
    ++v5;
  }
  return result;
}
```

- 函數邏輯就等校於下方的 python code:

```python
#!/usr/bin/env python3

def str_decrypt(enc, enc_len, key, key_len):
    enc = list(enc)
    ki = 0
    for i in range(enc_len):
        enc[i] ^= key[ki % key_len]
        enc[i] ^= key[(ki+1) % key_len]
        enc[i] ^= key[(ki+2) % key_len]
        enc[i] ^= key[(ki+3) % key_len]
        ki += 4
    return bytes(enc)

with open('key.bin', 'rb') as f:
    key = f.read()

str_v10  = bytes.fromhex('CE5F6D84CCD05DB3')[::-1]
str_v10 += bytes.fromhex('7F9B9A42B2CADA5D')[::-1]
str_v10 += bytes.fromhex('A24A')[::-1]
```

```
dec_v10 = str_decrypt(str_v10, 0x12, key, 0x41)

print(dec_v10)
```

- 其中 `key.bin` 取得方式能將以下 python script 貼到 IDA python 視窗中執行後取得:

```
# run in IDA python interpreter

bb = get_bytes(0x2020, 0x41, False)

with open('key.bin', 'ab') as f:
    f.write(bb)
```

- 解出了以下字串
  - /tmp/bnpkevsekfpk3
  - /tmp/bnpkevsekfpk3/aw3movsdirnqw
  - /tmp/khodsmeogemgoe
- 重新命名變數:

```
21  *str_bnpkevsekfpk3 = 0xCE5F6D84CCD05DB3LL;
22  *&str_bnpkevsekfpk3[8] = 0x7F9B9A42B2CADA5DLL;
23  *&str_bnpkevsekfpk3[16] = 0xA24Au;
24  v11 = 0;
25  *str_aw3movsdirnqw = 0xCE5F6D84CCD05DB3LL;
26  *&str_aw3movsdirnqw[8] = 0x7F9B9A42B2CADA5DLL;
27  *&str_aw3movsdirnqw[16] = 0x91AB22CDDD92A24ALL;
28  *&str_aw3movsdirnqw[24] = 0x7388DA9D25D8CCF8LL;
29  v14 = 0;
30  *str_khodsmeogemgoe = 0xD1596484CCD05DB3LL;
31  *&str_khodsmeogemgoe[8] = 0x62989648A4D1CC52LL;
32  *&str_khodsmeogemgoe[16] = 0xD8FE46;
33  str_decrypt(str_bnpkevsekfpk3, 0x12, &key, 0x41);
34  str_decrypt(str_aw3movsdirnqw, 0x20, &key, 0x41);
35  str_decrypt(str_khodsmeogemgoe, 0x13, &key, 0x41);
36  str_decrypt(str_payload_4040, num_3880, &key, 0x41);
```

- 後續執行步驟如下
  - open 了 /tmp/khodsmeogemgoe, 並將解密過的 str_payload_4040 寫入
  - fork
    - child process
      - 執行 /tmp/khodsmeogemgoe
    - parent process
      - 創造目錄 /tmp/bnpkevsekfpk3
      - mkfifo /tmp/bnpkevsekfpk3/aw3movsdirnqw
      - 將此 fifo 開啟, 向此 fifo 寫入了 0xd8 個 bytes
- 接下來逆向 /tmp/khodsmeogemgoe
  - open /tmp/bnpkevsekfpk3/aw3movsdirnqw
  - 從此 fifo read 0xd8 的 bytes, 以下稱 fifo_data
  - 將此 fifo unlink
  - 使用與之前的解密字串相同的函數進行資料解密, key 為 fifo_data, 解出來的字串如下
    - FLAG{FIFO_1s_D1sGVsTln9}

        - (其實就是這題的 flag 了)
    - 執行 fifo_data (此 0xd8 bytes 其實為能執行的 shellcode)
- 分析 fifo_data
    - 其位址與 key 相同, 但 key 長度只為 0x41, fifo_data 長度為 0xd8
    - 用以下腳本將其取出來:

```python
# run in IDA python interpreter

bb = get_bytes(0x2020, 0xd8, False)

with open('fifo_data.bin', 'ab') as f:
    f.write(bb)
```

    - 將 fifo_data.bin 拖進 IDA, 當作 64-bit x86_64 來分析, 部分截圖如下:

```
seg000:0000000000000000 seg000          segment byte public 'CODE' use64
seg000:0000000000000000                 assume cs:seg000
seg000:0000000000000000                 assume es:nothing, ss:nothing, ds:nothing, fs:noth
seg000:0000000000000000                 mov     rbp, rsp
seg000:0000000000000003                 mov     eax, 29h ; ')'
seg000:0000000000000008                 mov     edi, 2
seg000:000000000000000D                 mov     esi, 2
seg000:0000000000000012                 mov     edx, 11h
seg000:0000000000000017                 syscall                 ; Low latency system call
seg000:0000000000000019                 mov     [rbp-8], eax
seg000:000000000000001C                 mov     rdi, rax
seg000:000000000000001F                 mov     eax, 36h ; '6'
seg000:0000000000000024                 mov     esi, 1
seg000:0000000000000029                 mov     edx, 6
seg000:000000000000002E                 lea     r10, [rbp-10h]
seg000:0000000000000032                 mov     dword ptr [rbp-10h], 1
seg000:0000000000000039                 mov     r8d, 4
seg000:000000000000003F                 syscall                 ; Low latency system call
```

    - 流程大致是
        - 0x29 syscall (socket)
        - 0x36 syscall (setsockopt)
        - 進入以下迴圈
            - 0x2c syscall (sendto)
            - 0xe6 syscall (clock_nanosleep)

- 其將剛剛解密的 flag 字串以 udp 向 192.168.130.1 發送, 以 wireshark 來觀察, 如下圖