writeup_AeroCTF2021

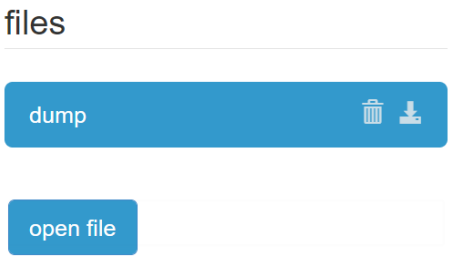
Challenge 1: Dummyper

Link containing challenge's source can be found here

Tìm hiểu sơ lược file nhận được

- Challenge chứa một file dump với size file siêu lớn(~ 36,6 MB), down về giải nến và check file type của nó thì nó là một ELF(ngoài ra còn phải check thêm hex dump của nó để phát hiện, phần sau của nó chứa một đống null-byte, gần 36,3MB là null-byte)

```
$ file dump
dump: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
```



- Ngoài ra, muốn lấy thêm một vài strings hữu ích trước khi drop file ***dump*** này vào ida, ta sử dụng thêm một tool luôn đi kèm với file đó là ***strings*** command

```
$ strings dump | grep Aero
# no output returns
$ strings dump -n 10
/lib64/ld-linux-x86-64.so.2
__stack_chk_fail
getpagesize
__cxa_finalize
__libc_start_main
GLIBC_2.2.5
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
[]A\A]A^A_
/dev/urandom
./flag.txt
/proc/self/maps
/proc/self/mem
/dev/urandom
./flag.txt
/proc/self/maps
/proc/self/mem
```

- Quan sát kết quả thu được ta thấy: quen có mà lạ cũng có. Có lẽ flag sẽ được đọc từ **flag.txt** sau đó in ra, hoặc đem đi xử lí, làm gì đó bla.bla, lạ thì có: ***/proc/self/maps, /proc/self/mem***; ***/dev/urandom*** thì có liên quan đến random một giá trị trong c: srand(), rand, seedv.v.

> Qua challenge này tui rút ra được một bài học xương máu là: phải kết hợp cả IDA vs ghidra, ghidra có thể ngu trong một vài trường hợp decompile, nhưng khi decompile các tên hàm hệ thống như fread(), fopen()... thì nó giữ nguyên, còn thằng IDA thì biến mất tiêu luôn.

Đi sâu vào phân tích challenge

Sau khi đã đọc qua 3 write-up được viết cho challenge này(đó là các write-up của: Midas, aaSSfxxx và fug1t1v31), có thể tóm tắt lại challenge như sau:

- Đi đến hàm main của file trên ida ta sẽ thấy có 3 hàm nhỏ lần lượt tại ***loc_1691***(có vẻ nó không phải một hàm), ***sub172a()*** và ***sub_188b()***

> Không thể tìm thấy được hàm **main()** ở cửa sổ decompile của ghidra, nhưng nhờ vào địa chỉ main trên ida mà ta có thể nhảy đến đó, nhấn **d** để decompile đoạn code tại đó, ngoài ra nhấn **f** để giúp ghidra nhận diện đó là một function.

- Challenge được tạo thành từ việc tác giả dump bộ nhớ của các section và heap của một tiếp trình đang chạy --> tạo thành file dump(chính vì có heap nên mới lí giả sao lại có nhiều byte \x00 đến thế), có thể nhận ra được điều này từ việc phân tích **sub_188b()**
- Sau đó một phần của data của ***dump*** file sẽ được mã hóa bởi phép XOR với một đoạn dữ liệu32 bytes(pseudo-code của loop cuối trong hàm ***sub_172a()***)
- '*Hàm*' thứ nhất(***loc_1691***) có vẻ như nằm trong vùng dữ liệu bị mã hóa, vì vậy nên lúc ida-disassem ra kết quả sai
- Từ các giả thuyết trên, tìm được XOR-key và khôi phục lại file **dump** như ban đầu(lưu với tên mới **new_dump**)
- Phân tích file **new_dump** sẽ phát hiện flag được đọc từ flag.txt(đọc 128 byte từ file và lưu vào ***heap tại 0x5060,*** có thể phát hiện ra điều này dựa vào hexeditor, tại đó bắt đầu theo sau là một loạt các null-byte), sau đó đem đi mã hóa bằng AES-128(key dài 128 bytes).
- Để có thể tìm được flag ban đầu, ta phải tìm được key và iv, có 2 cách để làm điều này:
  - cách 1:(cách của midas) vì a ấy tìm thấy được github repo thuật toán AES mà tác giả sử dụng, thuật toán này chỉ ra key, và IV sẽ xuất hiện 2 lần trong file dump memory(1 lần trong chính biến key hoặc value, và một lần trong biến ctx), dựa vào đó tìm được 2 mảng nào xuất hiện 2 lần trong heap(từ 0x5060 --> 0x5b6d8) thì hoặc là key, hoặc là IV. Thử lần lượt, và tìm được flag
  - cách 2: vì key và iv sẽ được lưu sau khi thực hiện lưu các byte ngẫu nhiên. T có thể retrieve lại vị trí lưu aeskey và iv bằng cách xác định chính xác được giá trị random(biến size) từ đó sinh ra các position chính xác cho key và iv. Trích xuất được key và iv, lấy đi decrypte

> Bài có 2 quá trình mã hóa mà chúng ta cần phải random đó chính là: xor key(mã hóa một đoạn dữ liệu trong chương trình), thứ 2 là mã hóa nội dung đọc từ flag.txt

Retrieve the original snipped-code using XOR Key

Preview qua các hàm nhỏ trong main

Như bình thường, nhảy đến hàm main để kiểm tra. main() gọi lần lượt 3 hàm nhỏ hơn nó tại các địa chỉ: **loc1691**, **sub_172a()**, **sub_188b()**

```
; int __cdecl main(int, char **, char **)
main proc near
endbr64
push    rbp
mov     rbp, rsp
call    loc_1691
call    sub_172A
call    sub_188B
mov     eax, 0
pop     rbp
retn
main endp
```

Đọc sơ lượt qua cả 3 hàm, tại địa chỉ đầu tiên(0x1691) xuất hiện các câu lệnh sai, lệnh lạ:

```
; ---------------------------------------------------------------------------

loc_1691:                               ; CODE XREF: main+8↓p
                out     53h, al
                pop     rbp
                out     1, eax          ; DMA controller, 8237A-5.
                                        ; channel 0 base address and word count
                std
                adc     [rdi+65h], cl
                movsb
                mov     ebx, 58D66E0Ah
; ---------------------------------------------------------------------------
                dq 91870DBC4BC97160h, 1FEC1165698C4247h, 26B5D424EA599C8Ah
                dq 6D5B26A257272DAAh, 0D337F0BB0838000h, 0D59EA23A7E7761B6h
                dq 0A188D955435C0CF2h, 92A4DD6ABF272DB3h, 0CABA004BBB8CE148h
```

Không phân tích được hàm thứ nhất nên t chạy đến hàm thứ 2 để thực hiện examine chúng, dưới đây làm một đoạn pseudo-code tại **sub_172A()**:

```
một đoạn pseudo-code của
 //...
 result = &loc_13A9;
  for ( j = 0; j <= 895; ++j )
  {
    result = (char *)&loc_13A9 + j;
    *result ^= *(_BYTE *)(j % 32 + v7);
  }
 return result;
```

Lại xor nữa, lần này lấy nguyên một phần data ở segment LOAD đi XOR, từ **0x13a9**, 896 phần tử là đến 0x1719--> như vậy phần code rác ở hàm nhỏ đầu tiên **loc_1691** cũng nằm trong dữ liệu mã hóa bởi phép xor.(với keyXor length = 32)

> Ngoài ra ta cũng phải chú ý, segment LOAD này rõ ràng nằm trong vùng code không có quyền write, nhưng nếu để ý decompile hàm **sub_172a** ở ghidra, chúng ta sẽ thấy, chương trình đã gọi system call **mprotect(*addr, len, protect_value)** để thay đổi thành vùng r-w-x

LOAD	0000000000000000	0000000000000A58	R . . .	L	mempa...	0001	public	DATA
LOAD	0000000000001000	0000000000002FF5	R . X .	L	mempa...	0002	public	CODE
LOAD	0000000000003000	00000000000037A8	R . . .	L	mempa...	0003	public	DATA
LOAD	0000000000004D20	0000000000805068	R W . .	L	mempa...	0004	public	DATA
extern	0000000000805068	0000000000805130	? ? ? .	L	qword	0008	public	

---------->hàm **sub_188b()**

Đầu tiên hai file hệ thống mà t cần chú ý đó là **/proc/self/maps** và **/proc/self/mem**, có thể hiểu gọn như thế này: thằng **maps** sẽ là file ánh xạ các virtual memory segment của tiến trình hiện tại, còn thằng **mem** sẽ là content của segment nào đó được trỏ đến trong bộ nhớ

```
acez@debian-armel:~$ cat /proc/self/maps
00008000-00012000 r-xp 00000000 08:01 380        /bin/cat
00019000-0001a000 r-xp 00009000 08:01 380        /bin/cat
0001a000-0001b000 rwxp 0000a000 08:01 380        /bin/cat
01c86000-01ca7000 rwxp 00000000 00:00 0          [heap]
b6d07000-b6e7e000 r-xp 00000000 08:01 7366       /usr/lib/locale/locale-archive
b6e7e000-b6fa8000 r-xp 00000000 08:01 761        /lib/arm-linux-gnueabi/libc-2.13.so
b6fa8000-b6fb0000 ---p 0012a000 08:01 761        /lib/arm-linux-gnueabi/libc-2.13.so
b6fb0000-b6fb2000 r-xp 0012a000 08:01 761        /lib/arm-linux-gnueabi/libc-2.13.so
b6fb2000-b6fb3000 rwxp 0012c000 08:01 761        /lib/arm-linux-gnueabi/libc-2.13.so
b6fb3000-b6fb6000 rwxp 00000000 00:00 0
b6fb6000-b6fd3000 r-xp 00000000 08:01 1381       /lib/arm-linux-gnueabi/ld-2.13.so
b6fd3000-b6fd5000 rwxp 00000000 00:00 0
b6fd9000-b6fda000 rwxp 00000000 00:00 0
b6fda000-b6fdb000 r-xp 0001c000 08:01 1381       /lib/arm-linux-gnueabi/ld-2.13.so
b6fdb000-b6fdc000 rwxp 0001d000 08:01 1381       /lib/arm-linux-gnueabi/ld-2.13.so
bed47000-bed68000 rw-p 00000000 00:00 0          [stack]
ffff0000-ffff1000 r-xp 00000000 00:00 0          [vectors]
acez@debian-armel:~$
```

Một ví dụ về nội dung của file /proc/self/maps

Trước khi đi tìm lại xor_key và retrive lại code, chúng ta cùng xem xem cách mà hàm dump được hình thành:

func_188b() được decompile sử dụng ghidra

```
1  //đổi tên một số biến để phù hợp cho việc RE
2  void FUN_0010188b(void) {
3     //..variable declaration part
4
5    local_10 = *(ulong *)(in_FS_OFFSET + 0x28);
6    proc_map_file = fopen("/proc/self/maps","r");
7    local_60 = 0;
8    local_70 = 0;
9    //đọc 12 byte đầu tiên của file mapping bộ nhớ
10   //của process hiện tại
11   fread(ptr_memMapsFile,0xc,1,proc_map_file);
12   local_1e = 0;
13   local_60 = strtoll(ptr_memMapsFile,(char **)0x0,0x10);
14   //sau câu lệnh này local_60 sẽ lưu giá trị hex của 12 byte đầu
```

```
  //tiên trong file maps, chính là địa chỉ bắt đầu của tiến trình
  local_80 = (char *)0x0;
  //đi qua các line trong file mapping đó và tìm phần
  //ánh xạ liên quan đến heap
  do {
    getline(&local_80,&local_78,proc_map_file);
    pcVar1 = strstr(local_80,"[heap]");
  } while (pcVar1 == (char *)0x0);
  ptr_chr_0x2d = strchr(local_80,'-');
  if (ptr_chr_0x2d != (char *)0x0) {
    ptr_chr_0x2d = ptr_chr_0x2d + 1;
    strncpy(12_charac_after_0x2d,ptr_chr_0x2d,0xc);
    local_10 = local_10 & 0xffffffffffffff00;
    local_70 = strtoll(12_charac_after_0x2d,(char **)0x0,0x10);
  }
  //để có thể hiểu được đoạn code trong phạm vi này
  //chúng ta cần nhìn vào hình minh họa ở trên và thấy rằng
  //nó sẽ tìm đến segment heap và copy kí tự đằng sau dấu '-'(chính
  //là ngụ ý copy address kết thúc của segment heap)
  //------------------------------

  //--> Như vậy t rút ra được kết luận rằng, file dump của chúng ta
  //sẽ là phần nội dung được dump từ bộ nhớ ra từ đầu( local_60)
  //đến kết thúc segment [heap](local_70)
  sizeFileDump = local_70 - local_60;
  memMappingFile = fopen("/proc/self/mem","r");
  if (memMappingFile != (FILE *)0x0) {
    fseek(memMappingFile,local_60,0);
    ptr = malloc(sizeFileDump);
    fread(ptr,sizeFileDump,1,memMappingFile);
    fclose(memMappingFile);
    dumpFile = fopen("dump","w");
    fwrite(ptr,sizeFileDump,1,dumpFile);
    fclose(dumpFile);
    if (local_10 == *(ulong *)(in_FS_OFFSET + 0x28)) {
      return;
    }
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  puts("NULL!");
                    /* WARNING: Subroutine does not return */
  exit(1);
}
```

Tìm xor_key và retrieve phần code rác

Một mảng 32 phần tử sẽ được đọc vào v7 từ một mảng data trong file(chú ý pseudo code dòng 17 của sub172a()). v7 lúc này như là một xor_key, vì được đọc dữ liệu từ file dump, nên t có thể tìm trong file để suy ra key. Các giả thuyết để retrieve xor_key:

- địa chỉ 13a9 được gọi như một hàmở dòng 16 trong pseudo code--> origial format của nó có vẻ là một hàm
- tất cả các hàm trong file đều được bắt đầu với 3 câu lệnh:
    + enbr64: F3 0F 1E FA
    + push rbp:55
    + mov rbp, rsp: 48 89 E5

==> Như vậy nếu đúng như dự đoán addr 0x13a9 cũng là một hàm thì 8 byte trên chính là plaintext trước khi mã hóa, ta lấy 8 byte này đi xor với cipher hiện tại tại 0x13a9 sẽ có phần của xorkey

```
truyennng@DESKTOP-5FNKINH:/mnt/c/Users/truye/Desktop/CTF challenges/AeroCTF/dump$ xxd dump | grep '428c 81c5 ea13 e0c2'
0004ba70: b881 5c7c 428c 81c5 ea13 e0c2 155c 431d  ..\|B........\C.
```

Như vậy XORKEY ở tại vị trí 0x4ba74, giờ đây có vị trí rồi, chúng ta sẽ viết script python nhỏ để khôi phục lại đoạn code 896 phần tử bị mã hóa:

retrieveCode.py

```python
dumpData = open('dump', 'rb').read()
key = dumpData[0x4ba74:0x4ba74 + 32]
encryptedData = dumpData[0x13a9: 0x13a9 + 896]
decryptedData = []
for i in range(0, len(encryptedData)):
    decryptedData.append(key[i%32] ^ encryptedData[i])
decryptedData = bytearray(decryptedData)
newDumpData = dumpData[:0x13a9] + decryptedData + dumpData[0x13a9+896:]
with open('new_dump', 'wb') as file:
    file.write(newDumpData)
    file.close()
```

Dưới đây là kết quả pseudo code của hàm 1691 sau khi được retrieve code lại như ban đầu:

```
__int64 __usercall sub_1691@<rax>(__int64 a1@<rbp>)
{
  unsigned int v1; // eax
  __int64 v2; // ST00_8
  __int64 v3; // ST08_8
  __int64 v5; // [rsp-8h] [rbp-8h]

  __asm { endbr64 }
  v5 = a1;
  qword_805060 = sub_1250("/dev/urandom", "r");
  v1 = sub_1210(0LL);
  sub_11F0(v1);
  v2 = sub_1250("./flag.txt", "r");
  v3 = sub_13A9(128LL);
  sub_11A0(v3, 128LL, 1LL, v2);
  sub_11B0(v2);
  return sub_13FE(v3);
}
```

Get flag with AES Crypto stuff

Hàm readFlag() tại 0x1691

Sau khi có được file new_dump, import file này vào ghidra, cùng lúc đó, đổi tên một vài hàm mà IDA không xác định được như fread(), fopen()..., ta được hàm **readFlag()** tại 0x1691 như sau:

```
1  void readFlag()(void)
2
3  {
4    time_t seed;
5    FILE *flag_stream;
6    void *__ptr;
7
8    _random_file = fopen("/dev/urandom","r");
9    seed = time((time_t *)0x0);
10   srand((uint)seed);
11   flag_stream = fopen("./flag.txt","r");
```

```
  __ptr = (void *)allocate_memFunc(0x80);
  fread(__ptr,0x80,1,flag_stream);
  fclose(flag_stream);
  AESCryptoStuff(__ptr);
  return;
}
```

Như vậy hàm này sẽ cấp phát 128 byte đầu tiên của file flag vào vị trí đầu tiên của một heap trỏ tới bởi ptr. Xem xét hàm allocate_memFunc, ta sẽ xác định được nó cấp phát bắt đầu tại 0x5060 và sau đó cứ cộng dần lên theo size mà chúng ta cung cấp

Hàm AESCryptoStuff(tại 0x13fe)

> Trong Crypto hay dùng ctx để chỉ context, như trong bài này một struct AES-ctx sẽ gồm 2 trường aeskey và aesiv
> AES code này tác giả lấy từ Github repo tiny-AES-c

```
void AESCryptoStuff(undefined8 param_1)
//param_1 chính là ptr đến flag
{
 //..variable declaration part
  i = 0;
  while (i < 0x40) {
    size= rand();
    pvVar = (void *)allocate_memFunc((long)(size% 0x7ff));
    fread(pvVar,(long)(size % 0x7ff),1,_random_file);
    i = i + 1;
  }
  //chúng ta sẽ thấy vòng lặp như vậy nhiều lần
  //sau khi đọc qua nhiều writeup, phát hiện ra rằng
  //các vòng lặp như vậy được cho vào để chèn các byte
  //rỗng được đọc từ file /dev/urandom/, sau đó chèn vào với
  //số lượng ngẫu nhiên( biến size = rand())

  //Sau các vòng lặp như thế này, lần lượt một key,
  //một iv( tất nhiên cũng được lấy từ /dev/urandom)
  //và một ctx(data stuct chứa key và value)
  //----------------------------
  pvVar = (void *)allocate_memFunc(0x20);
  j = 0;
  while (j < 0x40) {
```

```
      size= rand();
      pvVar1 = (void *)allocate_memFunc((long)(size% 0x7ff));
  //đọc từ file /dev/urandom dữ liệu ngẫu nhiên với số lượng bằng
  //size( cũng ngẫu nhiên luôn) vào trong heap được tạo
  //tại hàm readFlag()( heap tại 0x5060 và cứ cộng lên mỗi lần gọi
  //gọi hàm allocate_memFunc )
      fread(pvVar1,(long)(size% 0x7ff),1,_random_file);
      j = j + 1;
    }
  //Sau lặp xong thì cấp phát một giá trị quan trọng vào
  //ở đây là key chẳng hạn( lúc này chỉ mới cấp phát
  //trỏ đến bằng con trỏ, chứ chưa nạp data vào)
    pvVar = (void *)allocate_memFunc(0x20);
    j = 0;
    while (j < 0x40) {
      size = rand();
      pvVar1 = (void *)allocate_memFunc((long)(size % 0x7ff));
      fread(pvVar1,(long)(size % 0x7ff),1,_random_file);
      j = j + 1;
    }
  //cấp phát bộ nhớ cho iv trỏ đến bởi biến pvVar1
    pvVar1 = (void *)allocate_memFunc(0x10);
    k = 0;
    while (k < 0x40) {
      size = rand();
      pvVar3 = (void *)allocate_memFunc((long)(size % 0x7ff));
      fread(pvVar3,(long)(size % 0x7ff),1,_random_file);
      k = k + 1;
    }
  //nạp data lấy từ /dev/urandom vào 2 con trỏ:
  //pvVar là Key, pvVar1 là iv
    fread(pvVar,1,0x20,_random_file);
    fread(pvVar1,1,0x10,_random_file);
  //cấp phát vùng nhớ cho ctx
    uVar2 = allocate_memFunc(192);
    cnt = 0;
    while (cnt < 0x40) {
      size = rand();
      pvVar3 = (void *)allocate_memFunc((long)(size % 0x7ff));
      fread(pvVar3,(long)(size % 0x7ff),1,_random_file);
      cnt = cnt + 1;
    }
    aes_setupkey(uVar2,pvVar,pvVar);
    aes_setupiv(uVar2,pvVar1,pvVar1);
    AESEncrypt(uVar2,param_1,128,param_1);
    return;
  }
```

Sau khi đọc github chứa code AES trong C tham khảo được, t sẽ dần dần hình dung ra được 3 hàm cuối cùng sẽ làm gì: hai hàm đầu một là setupkey, một cái để setupiv, cái còn lại để encrypt. Như vậy chỉ cần khôi phục lại key và iv, nói cách khác là tìm được chúng trong đống dump đó thì set recover lại 128 byte đầu của flag.txt

Get Flag- Hướng 1: trick

đọc code github thấy key, iv của aes ngoài việc được lưu riêng lẻ trong heap, chúng còn được lưu chung trong biến ctx. Như vậy lúc dump phần heap, sẽ có 2 chuỗi byte được lặp lại lớn hơn 1 lần, một cái key, cái còn lại là iv tùy thuộc vào giá trị nào cho ra flag hợp lí

Get Flag- Viết C program để xác định position của aeskey và iv

Trước đó t đã biết vị trí của xor key, thằng này cũng dùng hàm allocate_memFunc(), giá trị base sẽ tăng lên theo size, reimplement lại các size(size = rand(), size dựa vào random, tức là dựa vào một seed giống như compile file dump). seed nào thỏa mãn vị trí xorkey = 0x4ba74 thì seed đó đúng–> vị trí của aeskey, aesiv, get chúng và suy ra flag.
Viết một chương trình bằng c(bruteforce time tại ngày 25-2 làngày modify file dump, bruteforce giờ phút giây 24*3600 + time(25-2))