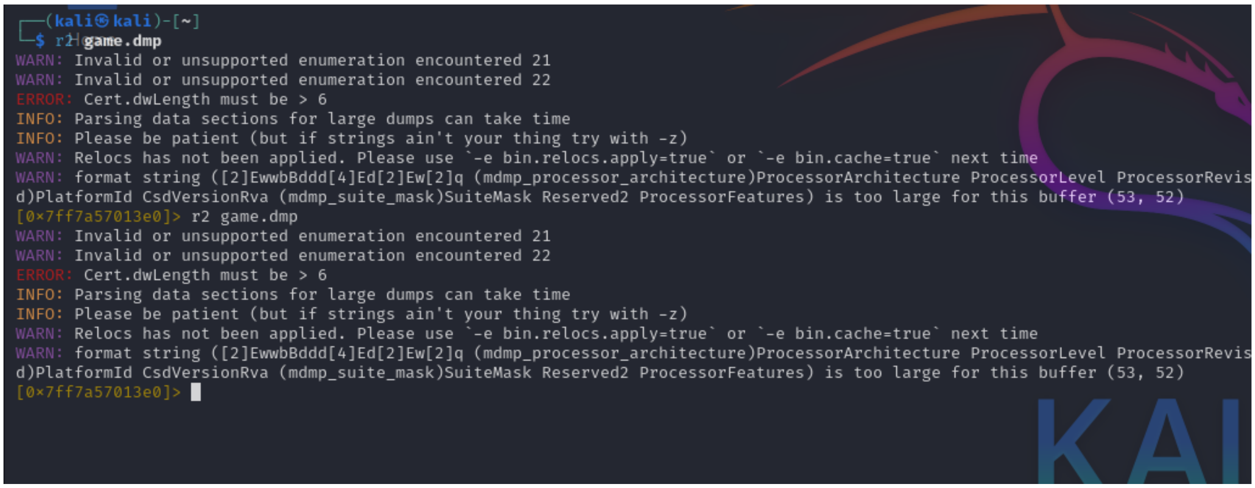
In this challenge, we will dig deeply into Crash Dump Forensic which is very different with Full Dump Forensic. The difference here is with crash dump we cannot use tool like volatility to resolve the problem since crash dump is just the memory dump from moment that program is ran.

On Linux we have a strong tool: r2 which belongs to radare, very suitable for analysing crash dump file

First, load the crash dump file:



```
┌──(kali㉿kali)-[~]
└─$ r2 game.dmp
WARN: Invalid or unsupported enumeration encountered 21
WARN: Invalid or unsupported enumeration encountered 22
ERROR: Cert.dwLength must be > 6
INFO: Parsing data sections for large dumps can take time
INFO: Please be patient (but if strings ain't your thing try with -z)
WARN: Relocs has not been applied. Please use `-e bin.relocs.apply=true` or `-e bin.cache=true` next time
WARN: format string ([2]EwwbBddd[4]Ed[2]Ew[2]q (mdmp_processor_architecture)ProcessorArchitecture ProcessorLevel ProcessorRevis
d)PlatformId CsdVersionRva (mdmp_suite_mask)SuiteMask Reserved2 ProcessorFeatures) is too large for this buffer (53, 52)
[0x7ff7a57013e0]> r2 game.dmp
WARN: Invalid or unsupported enumeration encountered 21
WARN: Invalid or unsupported enumeration encountered 22
ERROR: Cert.dwLength must be > 6
INFO: Parsing data sections for large dumps can take time
INFO: Please be patient (but if strings ain't your thing try with -z)
WARN: Relocs has not been applied. Please use `-e bin.relocs.apply=true` or `-e bin.cache=true` next time
WARN: format string ([2]EwwbBddd[4]Ed[2]Ew[2]q (mdmp_processor_architecture)ProcessorArchitecture ProcessorLevel ProcessorRevis
d)PlatformId CsdVersionRva (mdmp_suite_mask)SuiteMask Reserved2 ProcessorFeatures) is too large for this buffer (53, 52)
[0x7ff7a57013e0]>
```

Now when a program run, it will call DLLs or any libraries that need to ensure the stability of the program, and these information will be stored in the memory. We will use command **iS** to list out how many DLLs or any files contributed to the process:

```
[0x7ff7a57013e0]> iS
[Sections]

nth paddr          size vaddr             vsize perm type name
---------------------------------------------------------------
0   0x0000733a    0x1000 0x7ffe0000        0x1000 -r-- ---- Memory_Section
1   0x0000833a    0x1000 0x7ffeb000        0x1000 -r-- ---- Memory_Section_1
2   0x0000933a    0x7000 0x592cc9b000      0x7000 -rw- ---- Memory_Section_2
3   0x0001033a    0x5000 0x592cffb000      0x5000 -rw- ---- Memory_Section_3
4   0x0001533a    0x2000 0x592d1fe000      0x2000 -rw- ---- Memory_Section_4
5   0x0001733a    0x1000 0x592d3ff000      0x1000 -rw- ---- Memory_Section_5
6   0x0001833a    0x1000 0x16aaceb0000     0x1000 -rw- ---- Memory_Section_6
7   0x0001933a   0x10000 0x16aacec0000    0x10000 -rw- ---- Memory_Section_7
8   0x0002933a   0x20000 0x16aaced0000    0x20000 -r-- ---- Memory_Section_8
9   0x0004933a    0x4000 0x16aacef0000     0x4000 -r-- ---- Memory_Section_9
10  0x0004d33a    0x1000 0x16aacf00000     0x1000 -r-- ---- Memory_Section_10
11  0x0004e33a    0x2000 0x16aacf10000     0x2000 -rw- ---- Memory_Section_11
12  0x0005033a   0x11000 0x16aacf20000    0x11000 -r-- ---- Memory_Section_12
13  0x0006133a   0x11000 0x16aacf40000    0x11000 -r-- ---- Memory_Section_13
14  0x0007233a    0x3000 0x16aacf60000     0x3000 -r-- ---- Memory_Section_14
15  0x0007533a    0x7000 0x16aacf70000     0x7000 -r-- ---- Memory_Section_15
16  0x0007c33a    0x7000 0x16aacf80000     0x7000 -r-- ---- Memory_Section_16
17  0x0008333a    0x1000 0x16aacf90000     0x1000 -r-- ---- Memory_Section_17
18  0x0008433a    0x1000 0x16aacfa0000     0x1000 -r-- ---- Memory_Section_18
19  0x0008533a    0x2000 0x16aacfb0000     0x2000 -rw- ---- Memory_Section_19
20  0x0008733a    0x3000 0x16aacfd0000     0x3000 -r-- ---- Memory_Section_20
21  0x0008a33a   0x11000 0x16aacfe0000    0x11000 -r-- ---- Memory_Section_21
22  0x0009b33a   0x11000 0x16aad000000    0x11000 -r-- ---- Memory_Section_22
23  0x000ac33a    0x1000 0x16aad020000     0x1000 -rw- ---- Memory_Section_23
```

Scroll down and you will see the exe file and every essential DLLs or if you want to get only exe file you can use command **iS~exe** which will filter the output with the “exe” term:

```
[0x7ff7a57013e0]> iS~exe
176 0x0048933a  0x5de000 0x7ff7a5700000  0x5de000 ---- ---- C:\Users\Admin\Documents\game.exe
[0x7ff7a57013e0]> iS
[Sections]

nth paddr          size vaddr             vsize perm type name
---------------------------------------------------------------
0   0x0000733a    0x1000 0x7ffe0000        0x1000 -r-- ---- Memory_Section
1   0x0000833a    0x1000 0x7ffeb000        0x1000 -r-- ---- Memory_Section_1
2   0x0000933a    0x7000 0x592cc9b000      0x7000 -rw- ---- Memory_Section_2
```

Now the problem here is: we cannot extract the orignal exe file and we can just recover significant parts of it so to get them, we cannot just extract the part like the image above, the best way is extracting until you feel it’s good enough to analyse.

With r2, we can use **wtf <name_of_file> <size>** to extract data. But with these data we cannot analyse it yet since it’s still broken, so radare gave us a binary carver for fixing the binary approximately:

https://github.com/WithSecureLabs/radare2-scripts/blob/master/r2_bin_carver.py

```
d)PlatformId CsdVersionRva (mdmp_suite_mask)SuiteMask Reserved2 ProcessorFeatures) is too large
[0x7ff7a57013e0]> iS~exe
176 0x0048933a  0x5de000 0x7ff7a5700000  0x5de000 ---- ---- C:\Users\Admin\Documents\game.exe
[0x7ff7a57013e0]> s 0x7ff7a5700000
[0x7ff7a5700000]> px @ 0x7ff7a5700000
- offset -       0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7ff7a5700000  4d5a 9000 0300 0000 0400 0000 ffff 0000  MZ..............
0x7ff7a5700010  b800 0000 0000 0000 4000 0000 0000 0000  ........@.......
0x7ff7a5700020  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ff7a5700030  0000 0000 0000 0000 0000 0000 8000 0000  ................
0x7ff7a5700040  0e1f ba0e 00b4 09cd 21b8 014c cd21 5468  ........!..L.!Th
0x7ff7a5700050  6973 2070 726f 6772 616d 2063 616e 6e6f  is program canno
0x7ff7a5700060  7420 6265 2072 756e 2069 6e20 444f 5320  t be run in DOS
0x7ff7a5700070  6d6f 6465 2e0d 0d0a 2400 0000 0000 0000  mode....$.......
0x7ff7a5700080  5045 0000 6486 1500 9b4a 8468 0012 5d00  PE..d....J.h..].
0x7ff7a5700090  d602 0100 f000 2600 0b02 022b 00b8 4100  ......&....+..A.
0x7ff7a57000a0  0072 5800 003e 0000 e013 0000 0010 0000  .rX..>..........
0x7ff7a57000b0  0000 70a5 f77f 0000 0010 0000 0002 0000  ..p.............
0x7ff7a57000c0  0400 0000 0000 0000 0500 0200 0000 0000  ................
0x7ff7a57000d0  00e0 5d00 0006 0000 e418 8600 0300 6001  ..]...........`.
0x7ff7a57000e0  0000 2000 0000 0000 0010 0000 0000 0000  ................
0x7ff7a57000f0  0000 1000 0000 0000 0010 0000 0000 0000  ................
[0x7ff7a5700000]>
```

With the address of exe inside, we can build the query like this (the size for extraction is based on you):

```
┌──(kali㉿kali)-[~]
└─$ python3 r2_bin_carver.py game.dmp 0x7ff7a5700000 0x85EDB6 -p
WARN: Invalid or unsupported enumeration encountered 21
WARN: Invalid or unsupported enumeration encountered 22
ERROR: Cert.dwLength must be > 6
INFO: Parsing data sections for large dumps can take time
INFO: Please be patient (but if strings ain't your thing try with -z)
WARN: Relocs has not been applied. Please use `-e bin.relocs.apply=true` or `-e bin.cache=true` next time
WARN: format string ([2]EwwbBddd[4]Ed[2]Ew[2]q (mdmp_processor_architecture)ProcessorArchitecture ProcessorLevel
)PlatformId CsdVersionRva (mdmp_suite_mask)SuiteMask Reserved2 ProcessorFeatures) is too large for this buffer (5
INFO: Dumped 8777142 bytes from 0x7ff7a5700000 into game.dmp.0x7ff7a5700000
[+] Carving to game.dmp.0x7ff7a5700000
WARN: Relocs has not been applied. Please use `-e bin.relocs.apply=true` or `-e bin.cache=true` next time
[+] Patching ...
[+] Found 21 sections to patch
[+] Patching Section 0.
        Setting VirtualSize to 0x41b790
        Setting PointerToRawData to 0x1000
[+] Patching Section 1.
        Setting VirtualSize to 0x5720
        Setting PointerToRawData to 0x41d000
[+] Patching Section 2.
        Setting VirtualSize to 0x2d180
        Setting PointerToRawData to 0x423000
[+] Patching Section 3.
        Setting VirtualSize to 0xd4760
        Setting PointerToRawData to 0x451000
[+] Patching Section 4.
        Setting VirtualSize to 0x29e44
        Setting PointerToRawData to 0x526000
[+] Patching Section 5.
```

You can see that it will patch the exe and from now on we can analyse it using IDA pro:

```
.text:00007FF7A5701000 ;
.text:00007FF7A5701000 ; +-------------------------------------------------------------------------+
.text:00007FF7A5701000 ; |      This file was generated by The Interactive Disassembler (IDA)      |
.text:00007FF7A5701000 ; |           Copyright (c) 2024 Hex-Rays, <support@hex-rays.com>           |
.text:00007FF7A5701000 ; |                      License info: 01-2345-6789-AB                      |
.text:00007FF7A5701000 ; |                     IDA User <support@hex-rays.com>                     |
.text:00007FF7A5701000 ; +-------------------------------------------------------------------------+
.text:00007FF7A5701000 ;
.text:00007FF7A5701000 ; Input SHA256 : 43DA93A33EC5C7F2C63E2F3D70D5F74F40F465F843440DF0956945543B69BD9C
.text:00007FF7A5701000 ; Input MD5    : 6E52284D2171DE8D368B777F1FDAF3B6
.text:00007FF7A5701000 ; Input CRC32  : CABA8379
.text:00007FF7A5701000 ; Compiler     : Visual C++ (guessed)
.text:00007FF7A5701000
.text:00007FF7A5701000 ; File Name   : C:\Users\Admin\Downloads\game.exe
.text:00007FF7A5701000 ; Format      : Portable executable for AMD64 (PE)
.text:00007FF7A5701000 ; Imagebase   : 7FF7A5700000
.text:00007FF7A5701000 ; Timestamp   : 68844A9B (Sat Jul 26 03:25:15 2025)
.text:00007FF7A5701000 ; Section 1. (virtual address 00001000)
.text:00007FF7A5701000 ; Virtual size                  : 0041B790 (4306832.)
.text:00007FF7A5701000 ; Section size in file          : 0041B790 (4306832.)
.text:00007FF7A5701000 ; Offset to raw data for section: 00001000
.text:00007FF7A5701000 ; Flags 60000060: Text Data Executable Readable
.text:00007FF7A5701000 ; Alignment     : default
.text:00007FF7A5701000
.text:00007FF7A5701000                 .686p
.text:00007FF7A5701000                 .mmx
.text:00007FF7A5701000                 .model flat
.text:00007FF7A5701000
.text:00007FF7A5701000 ; ===========================================================================
.text:00007FF7A5701000
.text:00007FF7A5701000 ; Segment type: Pure code
```

From here many guys will not know exactly where to start so from my experience, it's good to start at finding **start** function since a program always runs from top to bottom so IDA will try to display that program flow. In sub_7FF7A570143B you will see it will process an alphabet order and something that we actually don't know:

```
__int64 sub_7FF7A570143B()
{
  __int64 v0; // rax
  __int64 v1; // rbx
  __int64 v2; // rax
  __int64 v3; // rax
  _BYTE v5[4]; // [rsp+2Ch] [rbp-14h] BYREF
  __int64 v6; // [rsp+30h] [rbp-10h]
  __int64 v7; // [rsp+38h] [rbp-8h]

  v7 = sub_7FF7A5703660();
  v0 = sub_7FF7A5713E20();
  sub_7FF7A5705010(v7, v0, 0LL);
  v1 = sub_7FF7A5A5A5A0(&unk_7FF7A5C7C080);
  v2 = sub_7FF7A5A5A5C0(&unk_7FF7A5C7C080);
  sub_7FF7A5703740(v7, v2, v1);
  sub_7FF7A57039F0(v7, &unk_7FF7A5C7C040, v5);
  sub_7FF7A5703680(v7);
  v6 = sub_7FF7A5703660();
  v3 = sub_7FF7A57139C0();
  sub_7FF7A5705010(v6, v3, 0LL);
  sub_7FF7A5703740(v6, "ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuv", 54LL);
  sub_7FF7A57039F0(v6, &unk_7FF7A5C7C060, v5);
  return sub_7FF7A5703680(v6);
}
```

If you dig into **sub_7FF7A5703740**, you will see the it will use EVP_DigestUpdate to process, it means it will hash the data into many small part (likely SHA256 and MD5):

```
    else
    {
      if ( v5 != 256 )
      {
        sub_7FF7A5886410();
        v6 = 410LL;
LABEL_9:
        sub_7FF7A5886540("../openssl-3.4.0/crypto/evp/digest.c", v6, "EVP_DigestUpdate");
        sub_7FF7A5886660(6LL, 189LL, 0LL);
        return 0LL;
      }
      return sub_7FF7A5714C50();
    }
  }
  else
  {
    v8 = *(_QWORD *)(a1 + 8);
    if ( v8 && *(_QWORD *)(v8 + 104) && (v3 & 0x100) == 0 )
    {
      v9 = *(__int64 (__fastcall **)(_QWORD))(v8 + 136);
      if ( !v9 )
      {
        sub_7FF7A5886410();
```

And basically we have a EVP_MD structure like this:

```
typedef struct env_md_st {
  int type;              // hash NID → 4 bytes
  int pkey_type;         // 4 bytes
  int md_size;           // output digest size → 4 bytes
  unsigned long flags;   // 4 or 8 bytes (platform dependent)
  int block_size;        // block size → 4 bytes
  int ctx_size;          // size of the context structure → 4 bytes
  int (*init)();         // function pointer → 8 bytes (on 64-bit)
  int (*update)();
  int (*final)();
  int (*copy)();
  int (*cleanup)();
  int (*ctrl)();
  ...
} EVP_MD;
```

Moreover, even you use any type of hashes or something same, this is the typical flow for them (maybe when you deploy it might be different but in general it has a mechanism like this):

```
EVP_MD_CTX *ctx = EVP_MD_CTX_new();
EVP_DigestInit_ex(ctx, EVP_sha256(), NULL);  // example for SHA-256

// Feed first chunk
EVP_DigestUpdate(ctx, data1, len1);
// Feed second chunk
EVP_DigestUpdate(ctx, data2, len2);

// Keep feeding more chunks if needed...

// Finalize & get output
unsigned char md[EVP_MAX_MD_SIZE];
unsigned int md_len = 0;
EVP_DigestFinal_ex(ctx, md, &md_len);
EVP_MD_CTX_free(ctx);
```

Now look again to previous picture:

```
__int64 sub_7FF7A570143B()
{
  __int64 v0; // rax
  __int64 v1; // rbx
  __int64 v2; // rax
  __int64 v3; // rax
  _BYTE v5[4]; // [rsp+2Ch] [rbp-14h] BYREF
  __int64 v6; // [rsp+30h] [rbp-10h]
  __int64 v7; // [rsp+38h] [rbp-8h]

  v7 = sub_7FF7A5703660();
  v0 = sub_7FF7A5713E20();
  sub_7FF7A5705010(v7, v0, 0LL);
  v1 = sub_7FF7A5A5A5A0(&unk_7FF7A5C7C080);
  v2 = sub_7FF7A5A5A5C0(&unk_7FF7A5C7C080);
  sub_7FF7A5703740(v7, v2, v1);
  sub_7FF7A57039F0(v7, &unk_7FF7A5C7C040, v5);
  sub_7FF7A5703680(v7);
  v6 = sub_7FF7A5703660();
  v3 = sub_7FF7A57139C0();
  sub_7FF7A5705010(v6, v3, 0LL);
  sub_7FF7A5703740(v6, "ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuv", 54LL);
  sub_7FF7A57039F0(v6, &unk_7FF7A5C7C060, v5);
  return sub_7FF7A5703680(v6);
}
```

If we compare with it, you can see the similarity. First, when you access function in **v7**, you will find it imported **digest.c:**

```
__int64 sub_7FF7A5703660()
{
  return sub_7FF7A572B080(72LL, "../openssl-3.4.0/crypto/evp/digest.c", 131LL);
}
```

For **v0**, it's an unk data and if you access you will see this data:

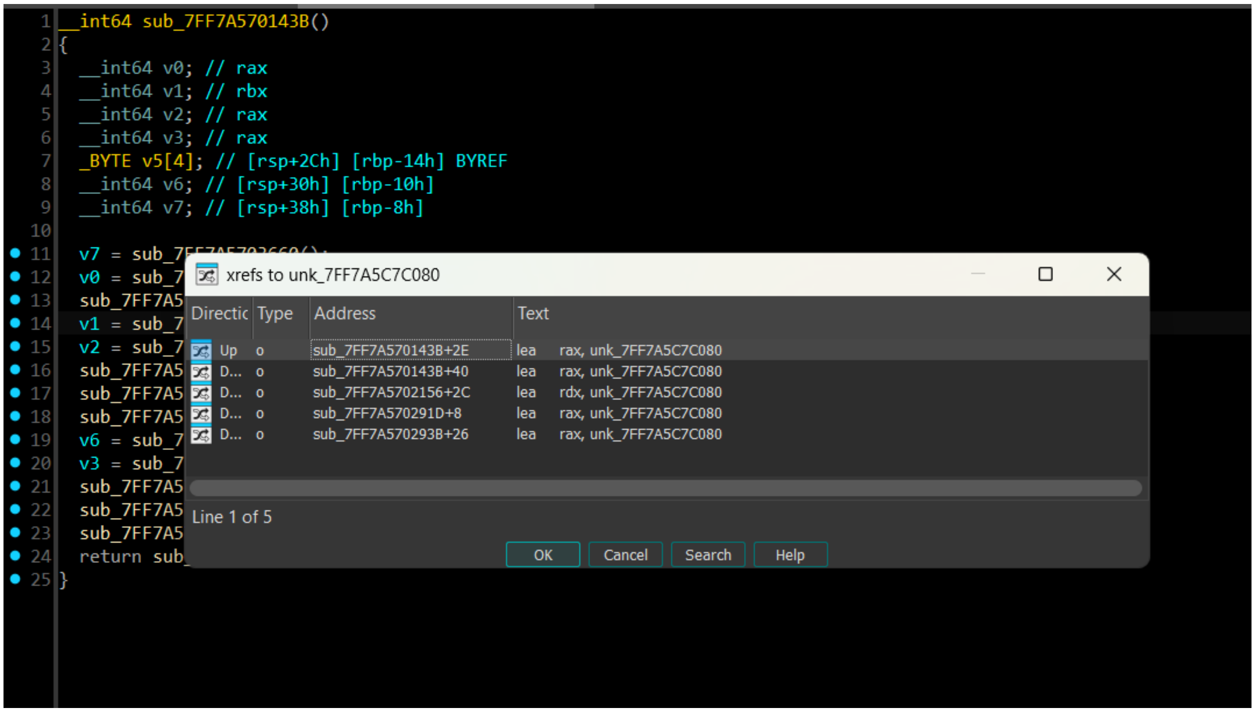
```
.rdata:00007FF7A5B57400 unk_7FF7A5B57400 db 0A0h                 ; DATA XREF: sub_7FF7A5713E20↑o
.rdata:00007FF7A5B57401                  db    2
.rdata:00007FF7A5B57402                  db    0
.rdata:00007FF7A5B57403                  db    0
.rdata:00007FF7A5B57404                  db  9Ch
.rdata:00007FF7A5B57405                  db    2
.rdata:00007FF7A5B57406                  db    0
.rdata:00007FF7A5B57407                  db    0
.rdata:00007FF7A5B57408                  db  20h
.rdata:00007FF7A5B57409                  db    0
.rdata:00007FF7A5B5740A                  db    0
.rdata:00007FF7A5B5740B                  db    0
.rdata:00007FF7A5B5740C                  db    8
.rdata:00007FF7A5B5740D                  db    0
.rdata:00007FF7A5B5740E                  db    0
.rdata:00007FF7A5B5740F                  db    0
.rdata:00007FF7A5B57410                  db    1
.rdata:00007FF7A5B57411                  db    0
.rdata:00007FF7A5B57412                  db    0
.rdata:00007FF7A5B57413                  db    0
.rdata:00007FF7A5B57414                  db    0
.rdata:00007FF7A5B57415                  db    0
.rdata:00007FF7A5B57416                  db    0
.rdata:00007FF7A5B57417                  db    0
```

Based on EVP_MD structure, you will see NID is 0x2A0 which is 672, matched with NID of SHA-256 following by OpenSSL document, output of digest size is 0x20 which equals to 32 bytes. From line 11 to 18 you could find easily that it matched with the flow we mentioned before. Do the same with other guys, and you will see this data in **sub_7FF7A57139C0**:

```
void *sub_7FF7A57139C0()
{
  return &unk_7FF7A5B56900;
}
```

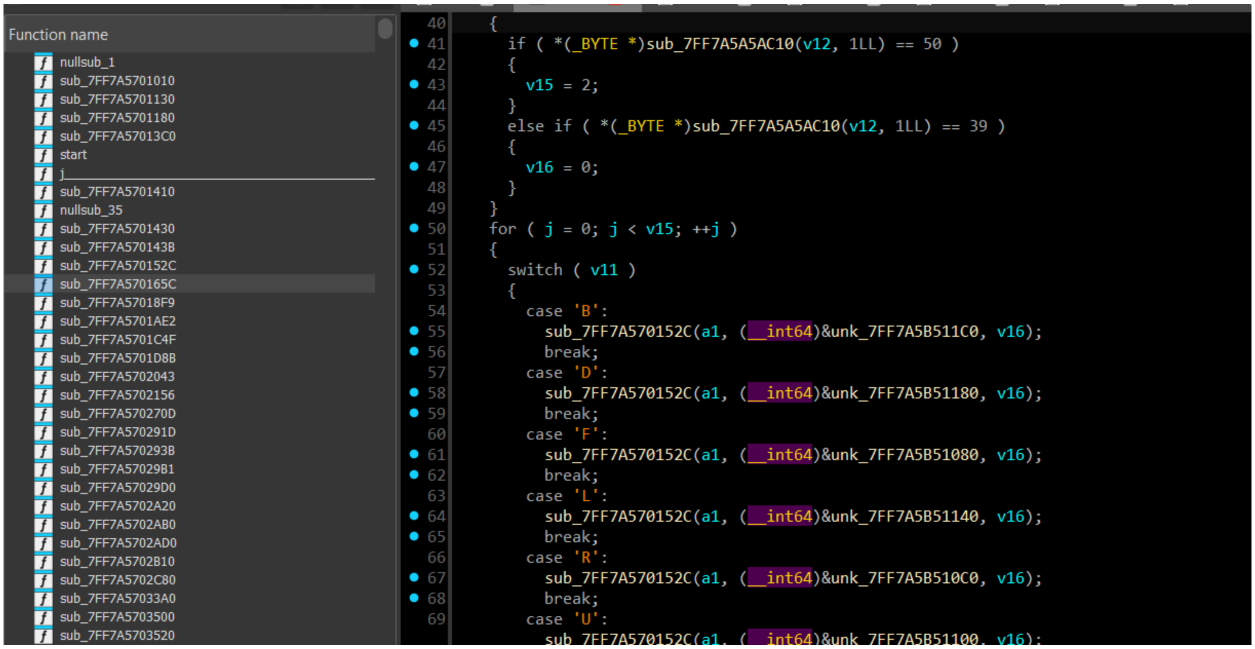
```
.rdata:00007FF7A5B568E6                 align 20h
.rdata:00007FF7A5B56900 unk_7FF7A5B56900 db    4                 ; DATA XI
.rdata:00007FF7A5B56901                 db    0
.rdata:00007FF7A5B56902                 db    0
.rdata:00007FF7A5B56903                 db    0
.rdata:00007FF7A5B56904                 db    8
.rdata:00007FF7A5B56905                 db    0
.rdata:00007FF7A5B56906                 db    0
.rdata:00007FF7A5B56907                 db    0
.rdata:00007FF7A5B56908                 db  10h
.rdata:00007FF7A5B56909                 db    0
.rdata:00007FF7A5B5690A                 db    0
.rdata:00007FF7A5B5690B                 db    0
.rdata:00007FF7A5B5690C                 db    0
.rdata:00007FF7A5B5690D                 db    0
.rdata:00007FF7A5B5690E                 db    0
.rdata:00007FF7A5B5690F                 db    0
.rdata:00007FF7A5B56910                 db    1
.rdata:00007FF7A5B56911                 db    0
.rdata:00007FF7A5B56912                 db    0
.rdata:00007FF7A5B56913                 db    0
.rdata:00007FF7A5B56914                 db    0
.rdata:00007FF7A5B56915                 db    0
.rdata:00007FF7A5B56916                 db    0
.rdata:00007FF7A5B56917                 db    0
```

This output of digest size is 0x10 which equals to 16 bytes and NID is 4 so in summary, they use sha256 and md5 for encryption something and you will notice above that md5 was used for alphabet, and how about other? Well you can track easily by choosing variable you want to track, then typing **X** it will display relevant functions:

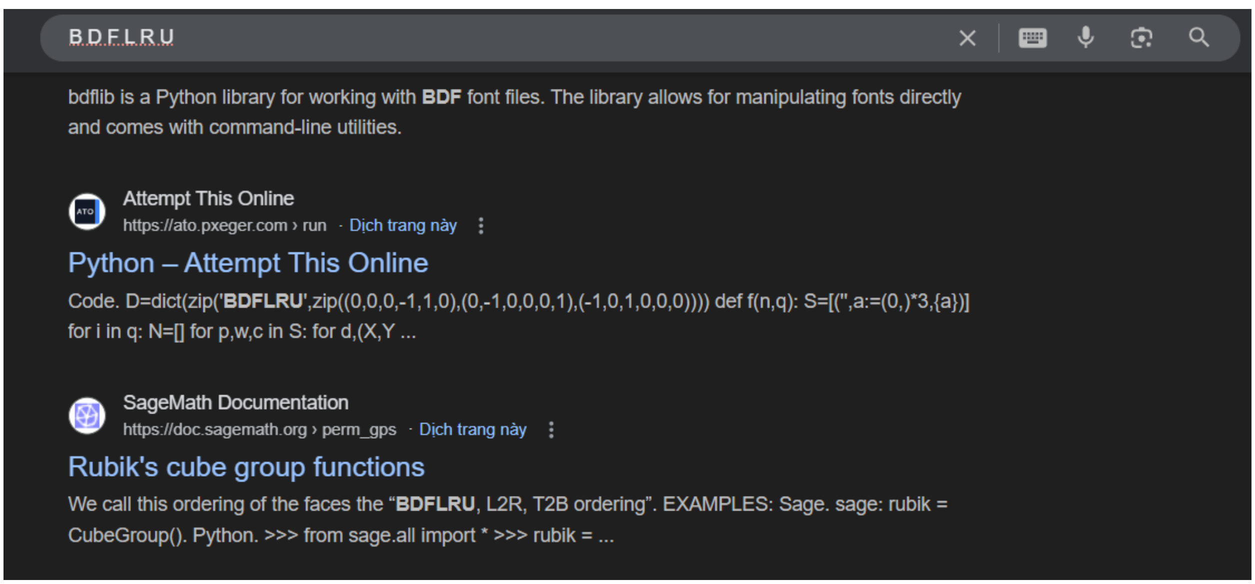


Those are functions using that variable, and navigating to the last guy, we will find their moves:

In summary, it will apply SHA-256 for this string and MD5 for alphabet. Next, we need to find how it used these moves, this function below will display fully:



What does this mean of B D F L R U? If you search on Google, you will know that they are types of move on Rubik:



So now basically, we will have these things:

- The type of moving they use is: D R2 F2 D B2 D2 R2 B2 D L2 D' R D B L2 B' L' R' B' F2 R2 D R2 B2 R2 D L2 D2 F2 R2 F' D' B2 D' B U B' L R' D'
- They use SHA256 and MD5 for encrypting movings and a

Now we dig into this function:

```
__int64 __fastcall sub_7FF7A570152C(__int64 a1, __int64 a2, char a3)
{
  char *v3; // rax
  __int64 result; // rax
  char v5; // bl
  unsigned __int8 v6; // [rsp+26h] [rbp-1Ah]
  char v7; // [rsp+27h] [rbp-19h]
  char v8; // [rsp+28h] [rbp-18h]
  char v9; // [rsp+29h] [rbp-17h]
  char v10; // [rsp+2Ah] [rbp-16h]
  char v11; // [rsp+2Bh] [rbp-15h]
  char v12; // [rsp+2Ch] [rbp-14h]
  char v13; // [rsp+2Dh] [rbp-13h]
  unsigned __int8 v14; // [rsp+2Eh] [rbp-12h]
  char v15; // [rsp+2Fh] [rbp-11h]
  char v16; // [rsp+30h] [rbp-10h]
  unsigned __int8 v17; // [rsp+31h] [rbp-Fh]
  char v18; // [rsp+32h] [rbp-Eh]
  char v19; // [rsp+33h] [rbp-Dh]
  char v20; // [rsp+34h] [rbp-Ch]
  unsigned __int8 v21; // [rsp+35h] [rbp-Bh]
  char v22; // [rsp+36h] [rbp-Ah]
  char v23; // [rsp+37h] [rbp-9h]
  int j; // [rsp+38h] [rbp-8h]
  int i; // [rsp+3Ch] [rbp-4h]

  for ( i = 0; i <= 8; ++i )
  {
    v3 = (char *)sub_7FF7A5AF5DF0(a1, *(int *)(4LL * i + a2));
    *(&v15 + i) = *v3;
  }
  if ( a3 )
  {
```

```
  if ( a3 )
  {
    v6 = v21;
    v7 = v18;
    v8 = v15;
    v9 = v22;
    v10 = v19;
    v11 = v16;
    v12 = v23;
    v13 = v20;
    result = v17;
    v14 = v17;
  }
  else
  {
    v6 = v17;
    v7 = v20;
    v8 = v23;
    v9 = v16;
    v10 = v19;
    v11 = v22;
    v12 = v15;
    v13 = v18;
    result = v21;
    v14 = v21;
  }
  for ( j = 0; j <= 8; ++j )
  {
    v5 = *(&v6 + j);
    result = sub_7FF7A5AF5DF0(a1, *(int *)(4LL * j + a2));
    *(_BYTE *)result = v5;
  }
  return result;
}
```

You can see that they did something like swapping, and based on information we get before: RUBIK, we can understand like this:

	v6	v7	v8	
v9	v15	v16	v17	
	v18	v19	v20	v11.
	v21	v22	v23	
	v12	v13	v14.	

<u>Before</u>

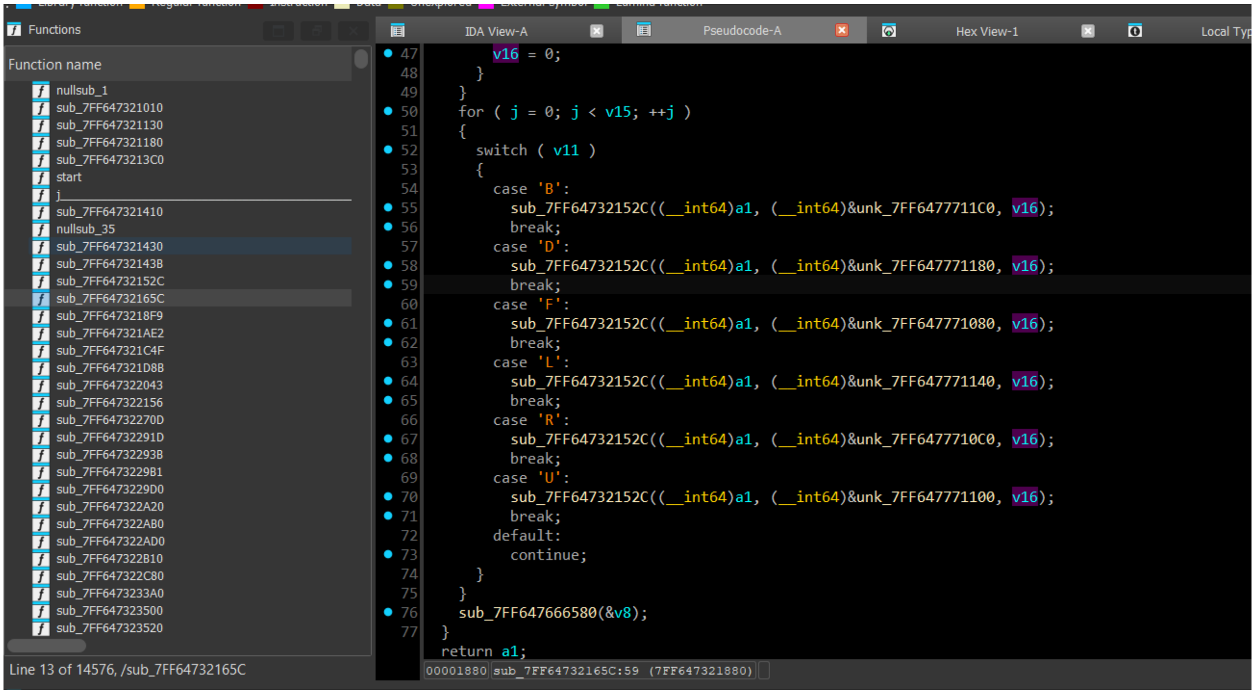
: rotate 90° 1 time

	v6	v7	v8	
	v21	v18	v15	
v9	v22	v19	v16	v11
	v23	v20	v17	
	v12	v13	v14.	

After swapped clockwise.

Do the same with counter-clockwise: . . . (do by yourself)

This is 90 degree rotation and based on their movings, the process will be different on each block. For the loop at the bottom: for each slot j = 0..8 it will get the rotated value v6 to v14 and call **sub_7FF7A5AF5DF0** to get a new destination and finally write that byte. And you will find that the rotate face function will be called by this function, it's very easy that it will do rotation based on each position and you can extract the indices by clicking on unk data:



After extracted, you're expected to get these data and the value of **v16** will decide which direction will be rotated: **clockwise** or **counter-clockwise**:

F[9] = {6,7,8,15,16,17,24,25,26};

R[9] = {2,5,8,11,14,17,20,23,26};

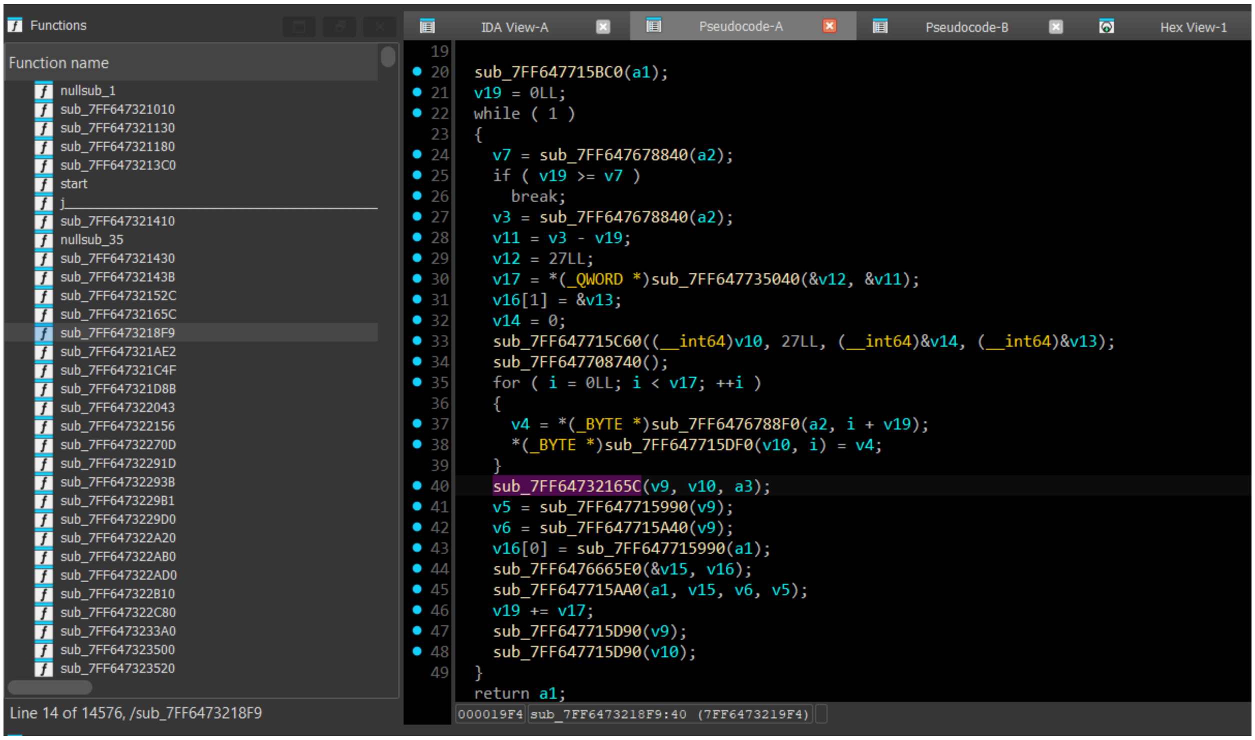
U[9] = {0,1,2,9,10,11,18,19,20};

L[9] = {0,3,6,9,12,15,18,21,24};

D[9] = {18,19,20,21,22,23,24,25,26};

B[9] = {0,1,2,3,4,5,6,7,8};

Also the function will be called by **sub_7FF6473218F9,** this cuts the data into **27-bytes blocks**, permutes each block, and puts them back together.



Next it's this function, it will do AES-256-CBC encryption which key and iv we found before. How do we know? Basically as I said before, any type of hash or encryption you're using must obey this general format: Initilize -> Encrypt -> Final blocks (handle padding) -> Clean up and AES encryption belongs to EVP_CIPHER this part of code displayed completedly:

```
// Create context
EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();


// Init encrypt operation
EVP_EncryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, key, iv);


// Encrypt the data
EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, plaintext_len);
ciphertext_len = len;


// Final block (handles padding)
EVP_EncryptFinal_ex(ctx, ciphertext + len, &len);
ciphertext_len += len;
```

// Clean up

EVP_CIPHER_CTX_free(ctx);

```
v9 = 0;
v12 = 0;
v3 = (unsigned int)sub_7FF647328E40();
sub_7FF64732DA50(v13, v3, 0, (unsigned int)&unk_7FF64789C040, (__int64)&unk_7FF64789C060);
v4 = sub_7FF647678840(a2);
v5 = sub_7FF647678810(a2);
v6 = sub_7FF647715A10(a1);
sub_7FF647329F10(v13, v6, (unsigned int)&v9, v5, v4);
v12 = v9;
v7 = sub_7FF647715A10(a1);
sub_7FF64732A360(v13, v7 + v9, &v9);
v12 += v9;
sub_7FF647329D40(v13);
sub_7FF647715B40(a1, v12);
return a1;
}
```

More precisely, in **v3** it contains unk data and we have the EVP_CIPHER structure generally:

EVP_CIPHER {

int nid; // Cipher NID

int block_size; // Block size (AES block size = 16)

int key_len; // Key length (16, 24, or 32 bytes)

int iv_len; // IV length (AES CBC = block size)

unsigned long flags;

function pointers...

}

```
.rdata:00007FF647774280 unk_7FF647774280 db 0ABh              ; DATA XREF: sub_7FF647328E40+12↑o
.rdata:00007FF647774281                db    1
.rdata:00007FF647774282                db    0
.rdata:00007FF647774283                db    0
.rdata:00007FF647774284                db  10h
.rdata:00007FF647774285                db    0
.rdata:00007FF647774286                db    0
.rdata:00007FF647774287                db    0
.rdata:00007FF647774288                db  20h
.rdata:00007FF647774289                db    0
.rdata:00007FF64777428A                db    0
.rdata:00007FF64777428B                db    0
.rdata:00007FF64777428C                db  10h
.rdata:00007FF64777428D                db    0
.rdata:00007FF64777428E                db    0
.rdata:00007FF64777428F                db    0
.rdata:00007FF647774290                db    2
.rdata:00007FF647774291                db    0
.rdata:00007FF647774292                db    0
.rdata:00007FF647774293                db    0
.rdata:00007FF647774294                db    1
.rdata:00007FF647774295                db    0
.rdata:00007FF647774296                db    0
.rdata:00007FF647774297                db    0
```

Following the EVP_CIPHER structure, we will have the NID is 0x1AB and if you search on Google about what is NID of AES_256_CBC, it’s 472 and matching with this hex value:

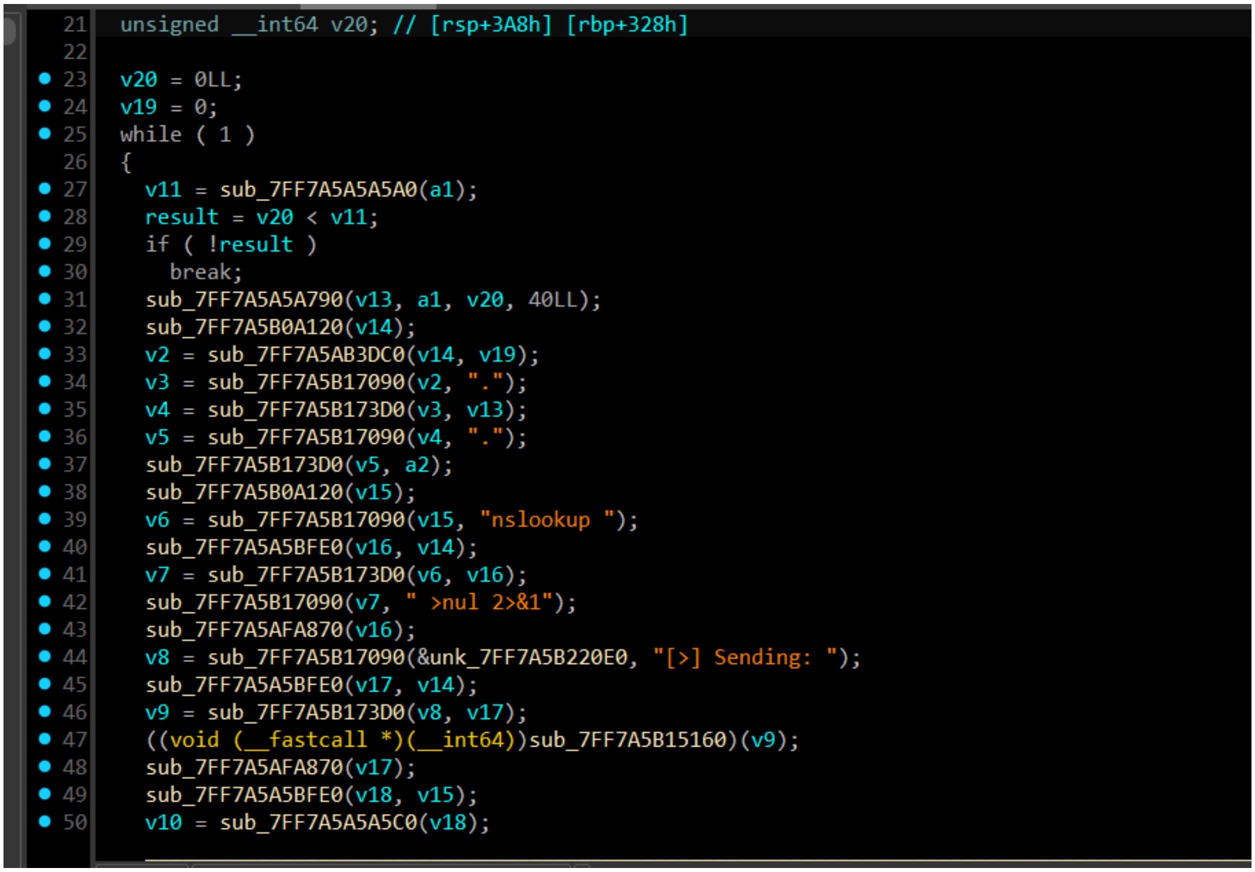
```
#define SN_aes_256_ecb          "AES-256-ECB"
#define LN_aes_256_ecb          "aes-256-ecb"
#define NID_aes_256_ecb         426
#define OBJ_aes_256_ecb         OBJ_aes,41L

#define SN_aes_256_cbc          "AES-256-CBC"
#define LN_aes_256_cbc          "aes-256-cbc"
#define NID_aes_256_cbc         427
#define OBJ_aes_256_cbc         OBJ_aes,42L
```

Finally, the last function will encode data into base32 format and the function after that will send the data by applying DNS exfiltration:

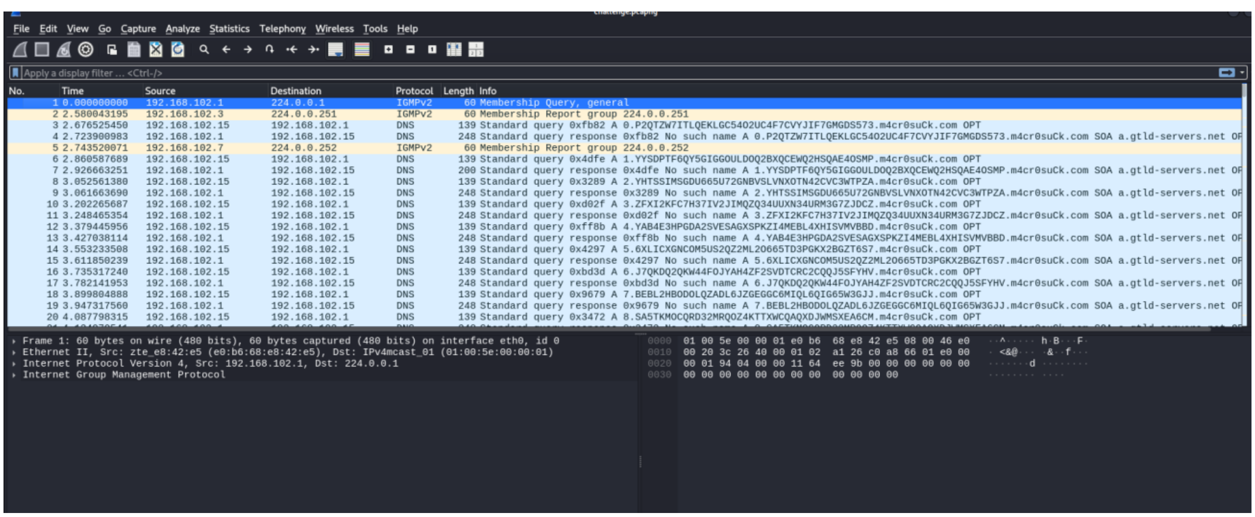
```
__int64 v3; // [rsp+28h] [rbp-28h] BYREF
__int64 v4; // [rsp+30h] [rbp-20h] BYREF
unsigned __int8 v5; // [rsp+3Fh] [rbp-11h]
__int64 v6; // [rsp+40h] [rbp-10h]
int v7; // [rsp+48h] [rbp-8h]
int v8; // [rsp+4Ch] [rbp-4h]

sub_7FF7A5AFA0E0(a1);
v8 = 0;
v7 = 0;
v6 = a2;
v4 = sub_7FF7A5A58860(a2);
v3 = sub_7FF7A5A587E0(a2);
while ( (unsigned __int8)sub_7FF7A5A47790(&v4, &v3) )
{
  v5 = *(_BYTE *)sub_7FF7A5A491B0(&v4);
  v8 <<= 8;
  v8 |= v5;
  for ( v7 += 8; v7 > 4; v7 -= 5 )
    sub_7FF7A5AFAB30(a1, (unsigned int)off_7FF7A5B1D000[(v8 >> (v7 - 5)) & 0x1F]);
  sub_7FF7A5A46610(&v4);
}
if ( v7 > 0 )
{
  v8 <<= 5 - v7;
  sub_7FF7A5AFAB30(a1, (unsigned int)off_7FF7A5B1D000[v8 & 0x1F]);
}
return a1;
```



So the workflow will be: Raw -> Rubik -> AES -> base32 -> encrypted

From here we can write a simple script to decrypt:




```python
import subprocess
import re
import sys
import hashlib
from Crypto.Cipher import AES
```

```python
import base64

# === CONFIG ===
PCAP_FILE = "challenge.pcapng"   # your .pcapng file
DOMAIN = "m4cr0suCk.com"        # your exfil domain
OUTPUT_BIN = "output.bin"       # output file

RUBIK_KEY = "D R2 F2 D B2 D2 R2 B2 D L2 D' R D B L2 B' L' R' B' F2 R2 D R2 B2 R2 D L2 D2 F2 R2 F' D' B2 D' B U B' L R' D'"
RUBIK_IV = b"ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuv"

# === Rubik cube faces ===
F = [6,7,8,15,16,17,24,25,26]
R = [2,5,8,11,14,17,20,23,26]
U = [0,1,2,9,10,11,18,19,20]
L = [0,3,6,9,12,15,18,21,24]
D = [18,19,20,21,22,23,24,25,26]
B = [0,1,2,3,4,5,6,7,8]

def rotate_face(cube, face_indices, clockwise):
    face = [cube[i] for i in face_indices]
    rotated = [0]*9
    if clockwise:
        rotated[0]=face[6]; rotated[1]=face[3]; rotated[2]=face[0]
        rotated[3]=face[7]; rotated[4]=face[4]; rotated[5]=face[1]
        rotated[6]=face[8]; rotated[7]=face[5]; rotated[8]=face[2]
    else:
        rotated[0]=face[2]; rotated[1]=face[5]; rotated[2]=face[8]
        rotated[3]=face[1]; rotated[4]=face[4]; rotated[5]=face[7]
        rotated[6]=face[0]; rotated[7]=face[3]; rotated[8]=face[6]
    for i in range(9):
        cube[face_indices[i]] = rotated[i]

def rubik_permute_block(block, moves):
    cube = bytearray(27)
    for i in range(len(block)):
        cube[i] = block[i]
    for move in moves:
        face = move[0]
        clockwise = True
        times = 1
        if len(move) > 1:
            if move[1] == '2': times = 2
            elif move[1] == "'": clockwise = False
        for _ in range(times):
```

```
            if face == 'F': rotate_face(cube, F, clockwise)
            elif face == 'R': rotate_face(cube, R, clockwise)
            elif face == 'U': rotate_face(cube, U, clockwise)
            elif face == 'L': rotate_face(cube, L, clockwise)
            elif face == 'D': rotate_face(cube, D, clockwise)
            elif face == 'B': rotate_face(cube, B, clockwise)
    return bytes(cube)

def rubik_inverse(data, moves):
    inverse_moves = []
    for move in reversed(moves):
        face = move[0]
        if len(move) > 1 and move[1] == '2':
            inverse_moves.append(move)
        elif len(move) > 1 and move[1] == "'":
            inverse_moves.append(face)
        else:
            inverse_moves.append(face + "'")
    result = b''
    pos = 0
    while pos < len(data):
        block = data[pos:pos+27]
        block += b'\x00'*(27-len(block))
        result += rubik_permute_block(block, inverse_moves)[:len(block)]
        pos += 27
    return result

def decrypt_aes(ciphertext, key, iv):
    if len(ciphertext) % 16 != 0:
        raise ValueError(f"Ciphertext length {len(ciphertext)} is not
multiple of AES block size (16)")
    cipher = AES.new(key, AES.MODE_CBC, iv)
    plaintext = cipher.decrypt(ciphertext)
    pad_len = plaintext[-1]
    if pad_len > 0 and pad_len <= AES.block_size:
        plaintext = plaintext[:-pad_len]
    return plaintext

def main():
    print(f"[*] Extracting DNS queries from {PCAP_FILE}...")

    cmd = [
        "tshark",
        "-r", PCAP_FILE,
        "-Y", f"dns.qry.name contains \"{DOMAIN}\"",
```

```
        "-T", "fields",
        "-e", "dns.qry.name"
    ]

    result = subprocess.run(cmd, capture_output=True, text=True)
    if result.returncode != 0:
        print("[-] tshark failed!")
        print(result.stderr)
        sys.exit(1)

    lines = result.stdout.strip().split("\n")
    print(f"[+] Found {len(lines)} DNS queries")

    pattern = re.compile(r"^(\d+)\.([A-Z0-9]+)\." + re.escape(DOMAIN))
    chunks = {}

    for line in lines:
        match = pattern.match(line)
        if match:
            cid = int(match.group(1))
            data = match.group(2)
            if cid in chunks:
                if chunks[cid] == data:
                    continue  # exact duplicate → skip
                else:
                    print(f"[!] Warning: conflicting data for chunk {cid}
— using first seen.")
            else:
                chunks[cid] = data
        else:
            print(f"[-] Skipped: {line}")

    if not chunks:
        print("[-] No valid chunks found. Check your PCAP and domain.")
        sys.exit(1)

    print(f"[+] Unique valid chunks: {len(chunks)} → IDs:
{sorted(chunks.keys())}")

    # === Join base32 ===
    b32data = "".join(chunks[i] for i in sorted(chunks.keys()))
    missing_padding = len(b32data) % 8
    if missing_padding:
        b32data += '=' * (8 - missing_padding)
```

```python
    ciphertext = base64.b32decode(b32data)
    print(f"[+] Decoded ciphertext: {len(ciphertext)} bytes (mod 16:
{len(ciphertext) % 16})")

    if len(ciphertext) % 16 != 0:
        print("[-] ERROR: Ciphertext corrupted. Check chunks.")
        sys.exit(1)

    # === AES decrypt ===
    key = hashlib.sha256(RUBIK_KEY.encode()).digest()
    iv = hashlib.md5(RUBIK_IV).digest()

    decrypted = decrypt_aes(ciphertext, key, iv)

    # === Rubik inverse ===
    moves = RUBIK_KEY.split()
    plaintext = rubik_inverse(decrypted, moves)

    with open(OUTPUT_BIN, "wb") as f:
        f.write(plaintext)

    print(f"[+] Decryption complete → {OUTPUT_BIN}")

if __name__ == '__main__':
    main()
```

```
┌──(kali㉿kali)-[~]
└─$ python3 decrypt.py flag.txt.bruh
[*] Extracting DNS queries from challenge.pcapng...
[+] Found 420 DNS queries
[+] Unique valid chunks: 210 → IDs: [0, 1, 2, 3, 4, 5, 6, 7, 8,
2, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 6
109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121,
56, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168,
3, 204, 205, 206, 207, 208, 209]
[+] Decoded ciphertext: 5248 bytes (mod 16: 0)
[+] Decryption complete → output.bin

┌──(kali㉿kali)-[~]
└─$ 
```

Grep file and you find the flag:

```
┌──(kali㉿kali)-[~]
└─$ cat output.bin | grep -a WWF
WWF{wHAt_Is-TH3_l3V3l_40918401934801948109481094104}

┌──(kali㉿kali)-[~]
└─$ 
```