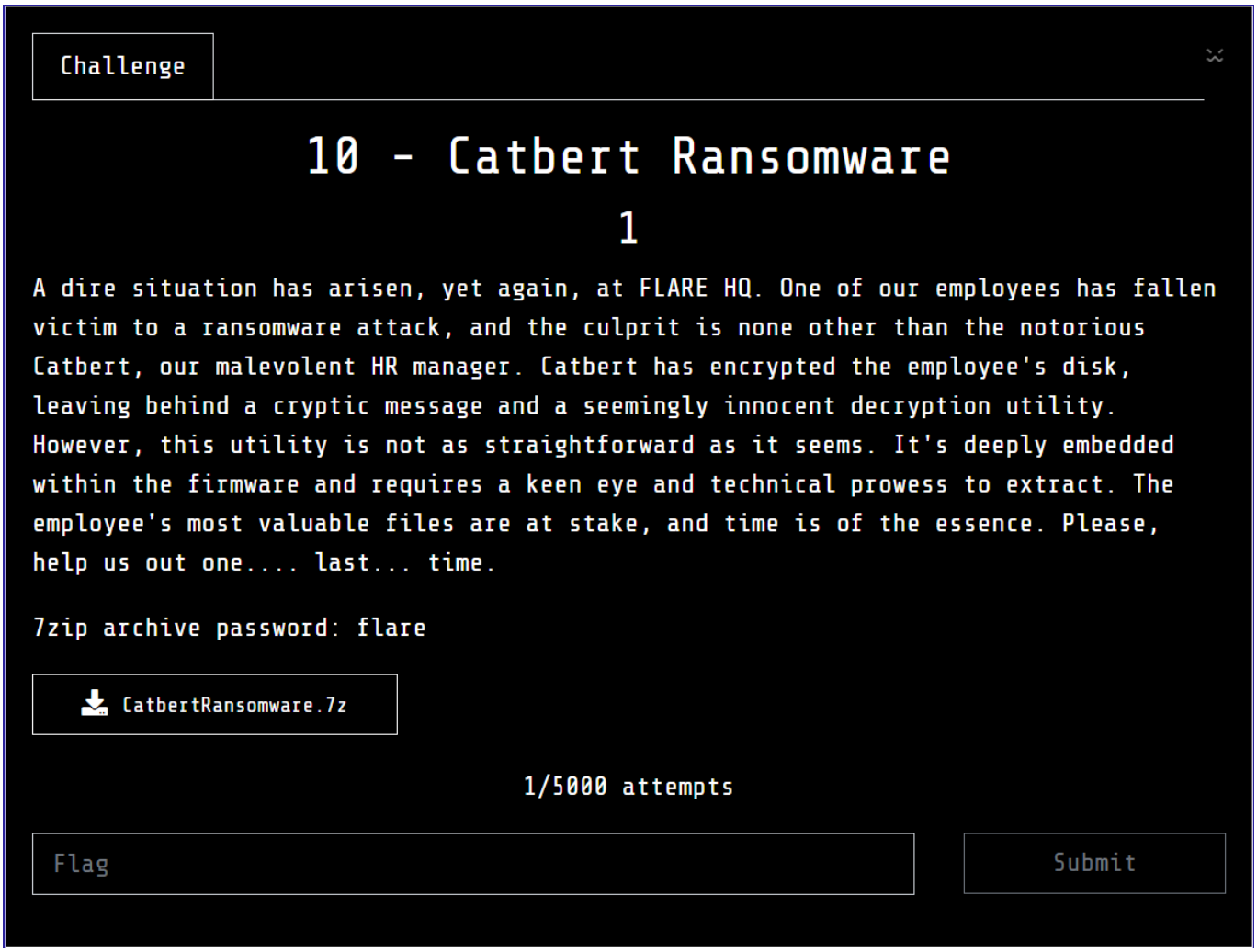
10 - Catbert Ransomware

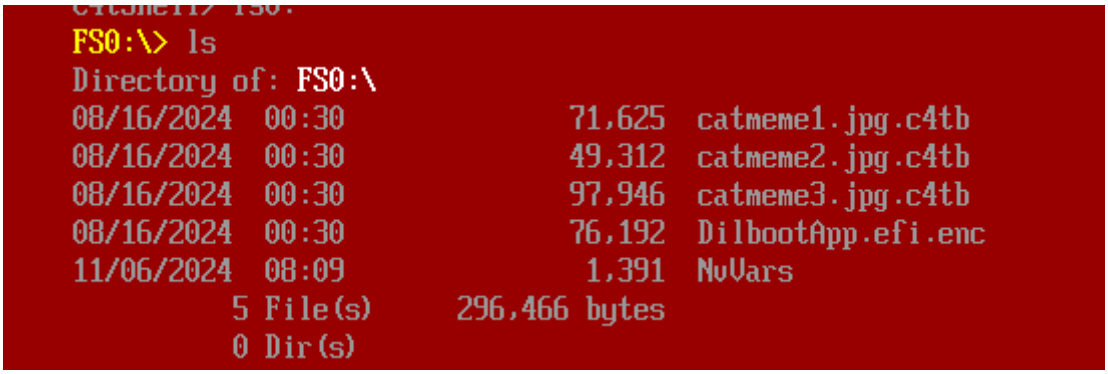


Initial analysis

The given files include a UEFI boot image and an encrypted disk image. First, i tried booting this with QEMU with command:

```
qemu-system-x86_64 -bios bios.bin -drive file=disk.img,format=raw
```

The disk contains three encrypted images and an EFI file



After some exploration, I found the command `decrypt_file`, which can be used to decrypt the three .c4tb files shown above

```
FS0:\> help
alias          - Displays, creates, or deletes UEFI Shell aliases.
attrib         - Displays or modifies the attributes of files or directories.
bcfg           - Manages the boot and driver options that are stored in NVRAM.
cd             - Displays or changes the current directory.
cls            - Clears the console output and optionally changes the background and foreground color
.
comp           - Compares the contents of two files on a byte-for-byte basis.
connect        - Binds a driver to a specific device and starts the driver.
cp             - Copies one or more files or directories to another location.
date           - Displays and sets the current date for the system.
dblk           - Displays one or more blocks from a block device.
decrypt_file   - Decrypts a user chosen .c4tb file from a mounted storage, given a decryption key.
devices        - Displays the list of devices managed by UEFI drivers.
devtree        - Displays the UEFI Driver Model compliant device tree.
dh             - Displays the device handles in the UEFI environment.
```

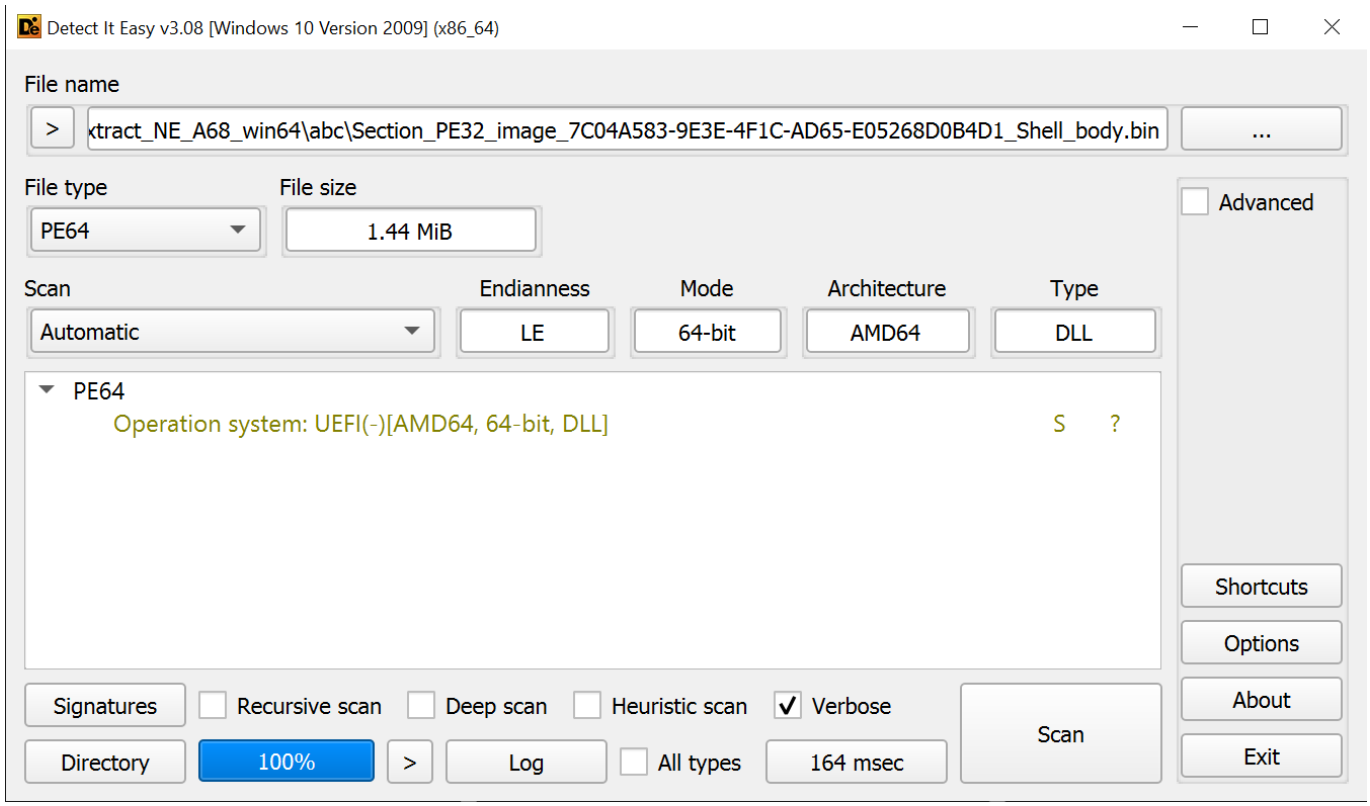
```
FS0:\> help decrypt_file
Decrypts a user chosen .c4tb file from a mounted storage, given a decryption key.

decrypt_file [filename] [decryption_key]

  filename - ought to reside in a mounted storage.
             make sure to mount the storage first.
  decryption_key - consists of printable characters.


EXAMPLES:
  * To see the mounted storage devices names, type 'map -r'.
    For a storage disk named 'FS0' type 'fs0:' in the shell to enter it.
    You can then list the files using shell commands like 'ls'
    (see 'help' command for a full ist of shell commands).

  * Within the storage you can then try to decrypt a file by passing its name
    and the corresponding decryption key, e.g.:
    fs0:\> decrypt_file some_filename1.txt.c4tb someD3cryptionK3y
```

Next, I extracted the `bios.bin` file using UEFITool with the `unpack` option, which yielded numerous files. Since I am not very familiar with UEFI and wasn't sure what to do next , I used a "super" tool — `strings | grep` — to search for the `decrypt_file` string in all the extracted files, hoping to locate the file that processes this command. And boom! Look what we found — a PE file!

```
:~/Downloads/bios.bin.dump$ find . -type f -exec strings -el  -f {} + | grep "decrypt_file"
./Section_PE32_image_7C04A583-9E3E-4F1C-AD65-E05268D0B4D1_Shell_body.bin: decrypt_file
./Section_PE32_image_7C04A583-9E3E-4F1C-AD65-E05268D0B4D1_Shell_body.bin: .TH decrypt_file 0 "Decrypt a CATBERT ransomware encrypted file."
./Section_PE32_image_7C04A583-9E3E-4F1C-AD65-E05268D0B4D1_Shell_body.bin: decrypt_file [filename] [decryption_key]
./Section_PE32_image_7C04A583-9E3E-4F1C-AD65-E05268D0B4D1_Shell_body.bin:     fs0:\> decrypt_file some_filename1.txt.c4tb someD3cryptionK3y
```

Shell_body.bin

It's a PE file running in UEFI



Let's load it into IDA with plugin efiXplorer. It's easy to find the function `sub_31BC4()` that handles the `decrypt_file` command.

```
__int64 sub_31BC4()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  result = sub_14FE4();
  if ( result < 0 )
    return result;
  result = sub_1BFA4();
  if ( result < 0 )
    return result;
  v2 = sub_16578((__int64)&off_71550, (__int64)&v36, (int)&v39, v1, 0);
  if ( v2 < 0 )
  {
    if ( v2 == 0x800000000000000Aui64 )
    {
      v3 = v39;
      if ( v39 )
      {
        sub_170FC(0xFFFFFFFF, 0xFFFFFFFF, 0i64, 5u, qword_E8538, L"decrypt_file", v39);
        Error(v3);
        LODWORD(v2) = 2;
      }
    }
    return (unsigned int)v2;
  }
  v4 = v36;
  if ( !sub_168D4(v36, 2i64) )
  {
    v5 = 3;
    v35 = qword_E8538;
LABEL_9:
    sub_170FC(0xFFFFFFFF, 0xFFFFFFFF, 0i64, v5, v35, L"decrypt_file");
```

By running the `efiXplorer` plugin, we can easily resolve many library function names

```
result = OpenFile((const CHAR16 *)filename, (SHELL_FILE_HANDLE *)&fileHandle, 3i64, 0i64);
if ( result < 0 )
  return result;
result = GetFileSize(fileHandle, &size);
if ( result < 0 )
  return result;
fileBuffer = (c4tb *)AllocMem(size);
enc_file = fileBuffer;
if ( !fileBuffer )
  return 9i64;
result = ReadFile(fileHandle, &size, fileBuffer);
if ( result < 0 )
  return result;
((void (__fastcall *)(EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *, __int64))gST->ConOut->SetAttribute)(gST->ConOut, 79i64
printf((const char *)0xFFFFFFFFi64, 0xFFFFFFFFi64, L"Successfully read %d bytes from %s\r\n", size, filename);
((void (__fastcall *)(EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *, __int64))gST->ConOut->SetAttribute)(gST->ConOut, 71i64
structX = (x *)AllocMem(0x20ui64);
pStruct = structX;
if ( !structX )
  return 9i64;
enc = enc_file;
signature = enc_file->signature;
structX->signature = enc_file->signature;
if ( signature != 'BT4C' )                    // signature
{
  ((void (__fastcall *)(EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *, __int64))gST->ConOut->SetAttribute)(gST->ConOut, 64i
  printf(
```

First, it checks if the signature of the encrypted file is `C4TB`

```
if ( signature != 'BT4C' )                    // signature
{
  ((void (__fastcall *)(EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *, __int64))gST->ConOut->SetAttribute)(gST->ConOut, 64i64);
  printf(
    (const char *)0xFFFFFFFFi64,
    0xFFFFFFFFi64,
    L"Oh, you thought you could just waltz in here and decrypt ANY file, did you?\r\n");
  printf(
    (const char *)0xFFFFFFFFi64,
    0xFFFFFFFFi64,
    L"Newsflash: Only .c4tb encrypted JPEGs are worthy of my decryption powers.\r\n");
  ((void (__fastcall *)(EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *, __int64))gST->ConOut->SetAttribute)(gST->ConOut, 71i64);
  return 2i64;
}
```

I created two structs and renamed some parameters to make it easier to read

```
struct c4tb
{
  DWORD signature;
  DWORD enc_length;
  DWORD vmcode_offset;
  DWORD length_vmcode;
  char *enc_data_offset;
};

struct x
{
  DWORD signature;
  DWORD encrypted_data_length;
  DWORD vmcode_offset;
  DWORD len_vmcode;
  __int64 decrypt_buffer;
```

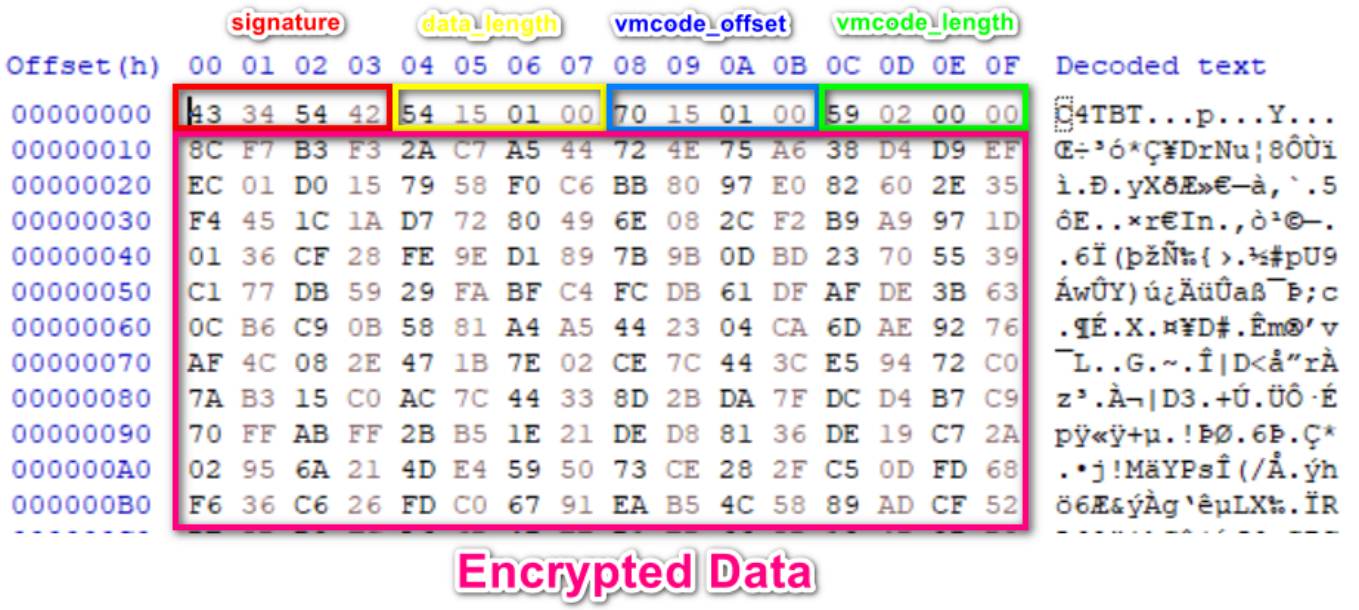
```
    __int64 enc_buffer;
};
```

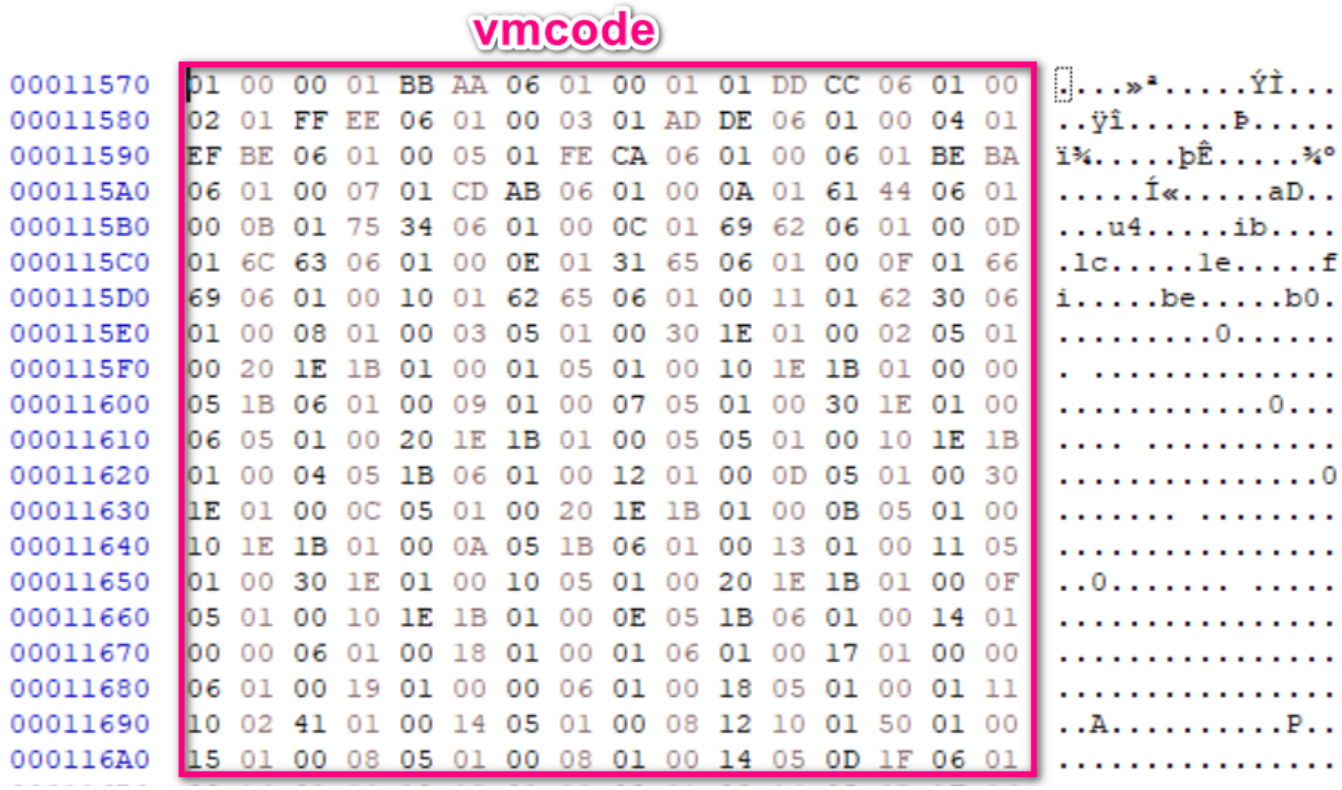
This code snippet reads the `c4tb` file and copies the input to vmcode:

```
v17 = v15[1];
v14[1] = v17;
v18 = v15[2];
v14[2] = v18;
v14[3] = v15[3];
v19 = sub_13050(v18, v17);
v20 = qword_E8588;
*(_QWORD *)(qword_E8588 + 24) = v19;
if ( !v19 )
  return 9i64;
sub_F9BC(v19, qword_E8598 + 16, *(unsigned int *)(v20 + 4));
v22 = sub_13050(v21, *(unsigned int *)(qword_E8588 + 12));
v23 = qword_E8588;
*(_QWORD *)(qword_E8588 + 16) = v22;
if ( !v22 )
  return 9i64;
sub_F9BC(v22, qword_E8598 + *(unsigned int *)(v23 + 8), *(unsigned int *)(v23 + 12));
if ( sub_112BC(qword_E8560) != 16 )
  goto LABEL_73;
v25 = sub_13050(v24, *(unsigned int *)(qword_E8588 + 12));
qword_E85A8 = v25;
if ( !v25 )
  return 9i64;
sub_F9BC(v25, *(_QWORD *)(qword_E8588 + 16), *(unsigned int *)(qword_E8588 + 12));
v26 = (_BYTE *)qword_E8560;
v27 = (_BYTE *)qword_E85A8;
*(_BYTE *)(qword_E85A8 + 5) = *(_BYTE *)qword_E8560;
v27[4] = v26[2];
v27[12] = v26[4];
v27[11] = v26[6];
v27[19] = v26[8];
v27[18] = v26[10];
v27[26] = v26[12];
```

```
enc_data_length = enc->enc_length;
pStruct->encrypted_data_length = enc_data_length;
pStruct->vmcode_offset = enc->vmcode_offset;
pStruct->len_vmcode = enc->length_vmcode;
MemAlloc_2 = AllocMem(enc_data_length);
v17 = structX;
structX->enc_buffer = (__int64)MemAlloc_2;
if ( !MemAlloc_2 )
  return 9i64;
memcpy(MemAlloc_2, &enc_file->enc_data_offset, v17->encrypted_data_length);
MemAlloc = AllocMem(structX->len_vmcode);
v19 = structX;
structX->decrypt_buffer = (__int64)MemAlloc;
if ( !MemAlloc )
  return 9i64;
memcpy(MemAlloc, (char *)enc_file + v19->vmcode_offset, v19->len_vmcode);
if ( StrLen(input_0) != 16 )
  goto Cry;
MemAlloc_1 = (char *)AllocMem(structX->len_vmcode);
vmCode_0 = MemAlloc_1;
if ( !MemAlloc_1 )
  return 9i64;
memcpy(MemAlloc_1, (const void *)structX->decrypt_buffer, structX->len_vmcode);
input = (char *)input_0;
vmCode = vmCode_0;
vmCode_0[5] = *(_BYTE *)input_0;
vmCode[4] = input[2];
vmCode[0xC] = input[4];
vmCode[0xB] = input[6];
vmCode[0x13] = input[8];
vmCode[0x12] = input[10];
vmCode[0x1A] = input[12];
```

c4tb File Format Structure





It then checks the key we input against `vmcode` in the `CheckKey()` function (we'll discuss this later). If the key is correct, it will decrypt the image using the RC4 algorithm

```
  vmCode[0x2E] = input[26];
  vmCode[0x36] = input[28];
  vmCode[0x35] = input[30];
  CheckKey();
  if ( !CorrectKey )
  {
Cry:
    Fail();
    return 2i64;
  }
  qword_E8578 = (__int64)AllocMem(0x208ui64);
  if ( !qword_E8578 )
    return 9i64;
  key_jpeg = AllocMem(0x104ui64);
  if ( !key_jpeg )
    return 9i64;
  CopyKeyJPEG();
  StrcpyS((_WORD *)qword_E8578, 0x104ui64, filename);
  v23 = SearchString((const CHAR16 *)qword_E8578, (const CHAR16 *)L".c4tb");
  if ( v23 )
    *v23 = 0;
  Image_Decrypt_Key_crc32 = CRC32_MPEG2();
  if ( Image_Decrypt_Key_crc32 == 0x8AE981A5 )  // key_img1
  {
    v25 = (void *)AllocNZeroMem(256i64);
    qword_E85B0 = (__int64)v25;
    goto LABEL_45;
  }
  if ( Image_Decrypt_Key_crc32 == 0x92918788 )  // key_img2
  {
    v25 = (void *)AllocNZeroMem(256i64);
    qword_E85B8 = (__int64)v25;
    goto LABEL_45;
  }
  if ( Image_Decrypt_Key_crc32 != 0x80076040 )  // key_img3
  {
LABEL_47:
    PrintYay();
    v26 = structX;
    RC4((char *)key_jpeg, v27, structX->enc_buffer, structX->encrypted_data_length, (_BYTE *)structX->enc_buffer);
    enc_buffer = (_BYTE *)v26->enc_buffer;
    if ( enc_buffer[6] != 'J' || enc_buffer[7] != 'F' || enc_buffer[8] != 'I' || enc_buffer[9] != 'F' )
    {
```

After successfully decrypting three images, it will decrypt the EFI using the key that was used to decrypt images 1 and 3:

```
if ( key_img1 && key_img2 && key_img3 && !byte_E8590 )
{
  ((void (__fastcall *)(EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *, __int64))gST->ConOut->SetAttribute)(gST->ConOut, 64i64);
  printf(
    (const char *)0xFFFFFFFFi64,
    0xFFFFFFFFi64,
    L"Oh, you think you're so smart, huh? Decrypting JPEGs? Big deal.\r\n");
  printf(
    (const char *)0xFFFFFFFFi64,
    0xFFFFFFFFi64,
    L"As a special favor, I'll let you enjoy the thrill of watching me\r\n");
  printf((const char *)0xFFFFFFFFi64, 0xFFFFFFFFi64, L"decrypt the UEFI driver. Consider yourself lucky.\r\n");
  ((void (__fastcall *)(EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *, __int64))gST->ConOut->SetAttribute)(gST->ConOut, 71i64);
  v29 = AllocMem(0x100ui64);
  key_efi = (__int64)v29;
  if ( !v29 )
    return 9i64;
  key_img3_0 = (_BYTE *)key_img3;
  *v29 = *(_BYTE *)(key_img3 + 7);
  v29[1] = key_img3_0[5];
  v29[2] = key_img3_0[2];
  v29[3] = key_img3_0[1];
  key_img1_0 = (_BYTE *)key_img1;
  v29[4] = *(_BYTE *)(key_img1 + 1);
  v29[5] = key_img3_0[5];
  v29[6] = key_img1_0[6];
  v29[7] = key_img3_0[2];
  v29[8] = key_img1_0[1];
  v29[9] = key_img1_0[6];
  v29[10] = key_img3_0[3];
  v29[11] = key_img1_0[13] + 3;
  v29[12] = key_img1_0[9];
  v29[13] = key_img1_0[10];
  v29[14] = key_img1_0[11];
  v29[15] = key_img1_0[12];
  v2 = sub_17CC0((__int64)L"DilbootApp.efi.enc");
  if ( v2 >= 0 )
  {
```

So, our task now is to find the key to decrypt the three images.

VMCode Compiler

Now, let's go back to the CheckKey() function. It's a compiler for vmcode, consisting of opcodes from `0x00` to `0x26` that perform operations similar to `push`, `pop`, `xor`, `and`, `shl`, and others.

```
__int64 CheckKey()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v0 = vmCode_0;
  StackPointer = (__int64 *)&Stack;
  CorrectKey = 0i64;
  pKeymap[0] = (__int64)vmCode_0;
LABEL_2:
  StackPointer_0 = (__int64)StackPointer;
  while ( 1 )
  {
    while ( 1 )
    {
      c = *(unsigned __int8 *)pKeymap[0]++;
      if ( c > 0x13 )
        break;
      if ( c == 0x13 )
      {
        v7 = StackPointer - 1;
        StackPointer_0 = (__int64)v7;
        v8 = *(v7 - 1) <= (unsigned __int64)*v7;
LABEL_37:
        *(v7 - 1) = v8;
        goto LABEL_87;
      }
      if ( c > 9 )
      {
        switch ( c )
        {
          case 0xAu:
            pKeymap[0] += 2i64;
            *(StackPointer - 1) += *(unsigned __int8 *)(pKeymap[0] - 1)
                                 + ((unsigned __int64)*(unsigned __int8 *)(pKeymap[0] - 2) << 8);
            goto LABEL_87;
          case 0xBu:
            StackPointer_0 = (__int64)(StackPointer - 1);
            *(StackPointer - 2) -= *(StackPointer - 1);
            goto LABEL_87;
          case 0xCu:
            StackPointer_0 = (__int64)--StackPointer;
```

I have reimplemented it in Python (here). With an input of aaaaaaaaaaaaaaa, it will print out as follows:

```
push 0x0
push 0x6161
MOV STORE[0x0],0x6161
push 0x1
push 0x6161
MOV STORE[0x1],0x6161
push 0x2
push 0x6161
MOV STORE[0x2],0x6161
push 0x3
push 0x6161
MOV STORE[0x3],0x6161
push 0x4
push 0x6161
MOV STORE[0x4],0x6161
push 0x5
push 0x6161
```

```
...
...
...
```

Image 1

The vmcode for checking the key to decrypt Image 1 performs operations like this

```
input =
[0x44,0x61,0x43,0x75,0x62,0x69,0x63,0x6c,0x65,0x4c,0x69,0x66,0x65,0x31,0x30,0x31]
cipher =
[0x44,0x61,0x34,0x75,0x62,0x69,0x63,0x6c,0x65,0x31,0x69,0x66,0x65,0x62,0x30,0x62]
def lose():
    print("wrong")
    exit()
def win():
    print("correct")
    exit()
for i in range(16):
    if i == 2:
        if(input[i]!=0xff&((cipher[i] >> 0x4) | (cipher[i] <<0x4))):
            lose()
        continue
    if i == 9:
        if(input[i]!=0xff&((cipher[i] << 0x6) | (cipher[i] >> 0x2))):
            lose()
        continue
    if i == 13 or i == 15:
        if(input[i]!=0xff&((cipher[i] >> 0x1) | (cipher[i] << 0x7))):
            lose()
        continue
    if (input[i] != cipher[i]):
        lose()
win()
```

The first key is easy to find: `DaCubicleLife101`

We've obtained the first part of the flag

Image 2

Key 2 verification algorithm

```
input = [0x47, 0x33, 0x74, 0x44, 0x61, 0x4a, 0x30, 0x62, 0x44, 0x30, 0x6e, 0x65,
0x4d, 0x34, 0x74, 0x65]
value = [0x1e, 0x93, 0x39, 0x2e, 0x42, 0x94, 0xf0, 0x46, 0xa6, 0x54, 0xdf, 0x3c,
0x4a, 0x46, 0x28, 0x1a]
cipher = [0x59, 0xa0, 0x4d, 0x6a, 0x23, 0xde, 0xc0, 0x24, 0xe2, 0x64, 0xb1, 0x59,
0x07, 0x72, 0x5c, 0x7f]

def lose():
    print("wrong")
    exit()
def win():
    print("correct")
    exit()

for i in range(len(input)):
    if(input[i]^value[i]!=cipher[i]):
        lose()
win()
```

Decrypt image 2 with Key `G3tDaJ0bD0neM4te`

Image 3

The first 4 characters of the key are checked using the DJB2 hash algorithm:

```
def djb2_hash(string):
    hash_value = 0x1505
    for char in string:
        hash_value = (hash_value *33) + ord(char)
    return (hash_value & 0xFFFFFFFF) == 0x7c8df4cb
```

I used brute force and got many plausible results, but only `VerY` seemed meaningful.

The next 4 characters are simply a rotation by 13 (ror 13) added to the input:

```
def ror13AddHash32(string):
    val = 0
    for i in string:
        val = ror(val, 0xd, 32)
        val += i
    return (val & 0xffffffff) == 0x8b681d82
```

Easy to find that an input that satisfies the condition is DumB

The last 8 characters are checked using Adler-32

```
def adler32(input_vals):
    MOD_ADLER = 0xFFF1
    s1 = 1
    s2 = 0

    for byte in input_vals:
        s1 = (s1 + byte) % MOD_ADLER
        s2 = (s2 + s1) % MOD_ADLER

    return (s2 << 16) | s1 == 0xf910374
```

I found the plaintext of the hash on google is `password`. Now we have the full password to decrypt image 3 : `VeryDumBpassword` Let's decrypt

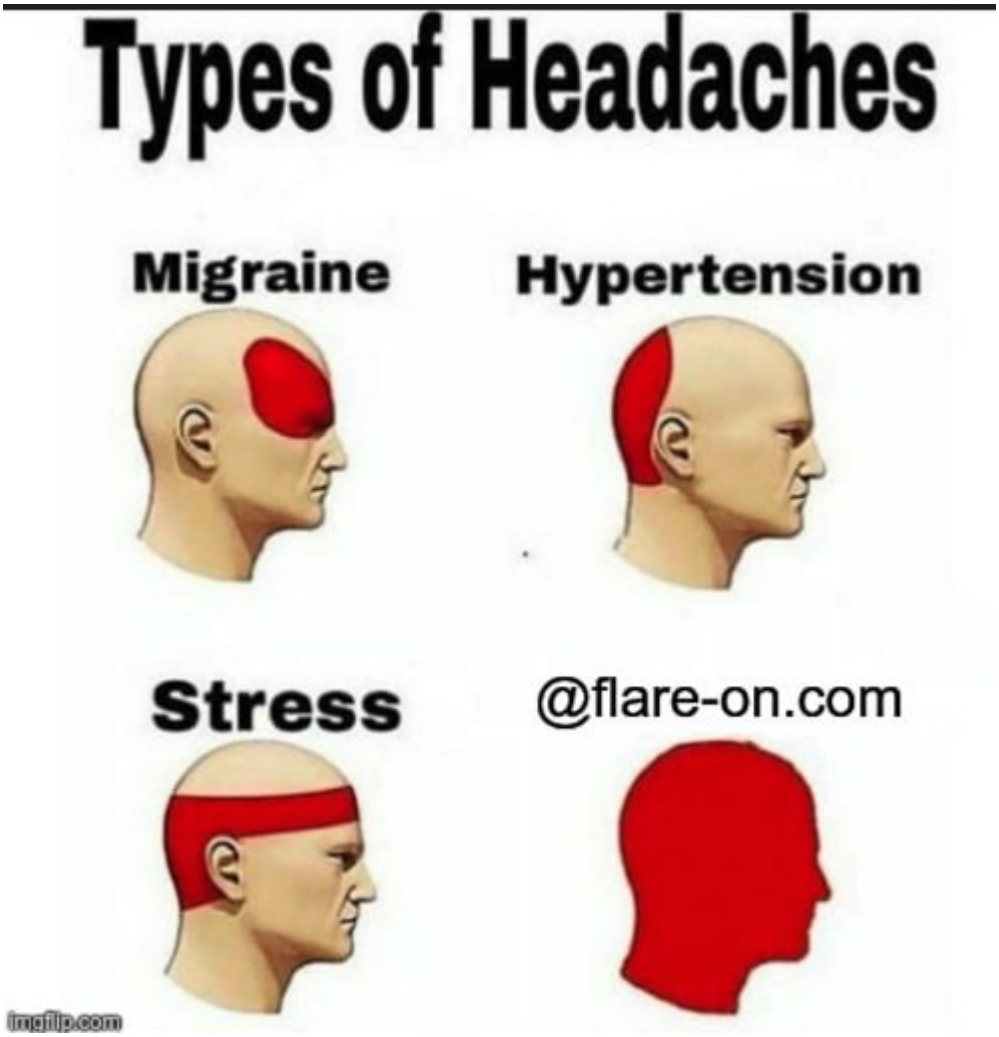
```
FS0:\> decrypt_file catmeme3.jpg.c4tb VerYDumBpassword
Successfully read 97946 bytes from catmeme3.jpg.c4tb
  /\_/\  (
 ( ^.^ ) _)
   \"/  (
 ( | | )
(__d b__)

  Yay!
0x17AB3 bytes successfully written to catmeme3.jpg.
Oh, you think you're so smart, huh? Decrypting JPEGs? Big deal.
As a special favor, I'll let you enjoy the thrill of watching me
decrypt the UEFI driver. Consider yourself lucky.
Successfully read 76192 bytes from DilbootApp.efi.enc
0x129A0 bytes successfully written to Dilboot.efi.
you've made it this far, have you? Pat yourself on the back.
Want to know what real fun is? Go ahead, run the .efi file.
Just don't say I didn't warn you.
```

After decrypting the three images, it automatically decrypts the Dilboot.efi file. Try running it

```
0x10 bytes successfully written to your_mind.jpg.c4tb.
You thought you were clever, huh?
Thought you'd find your precious answer here? Well, tough luck.
You were almost right, but not quite.
I've left another little surprise for you on disk.
Your reward is the password:'BrainNumbFromVm!'. Enjoy your headache, human.
And remember, I'm always watching.
```

Lets decrypt final image:

```
FS0:\> decrypt_file your_mind.jpg.c4tb BrainNumbFromVm!
Successfully read 51261 bytes from your_mind.jpg.c4tb
  _   _   _   _   _   _
 / \ / \ / \ / \ / \ / \
( u )( n )( d )( 3 )( r )( _ )
 \_/ \_/ \_/ \_/ \_/ \_/

  _   _   _   _   _   _   _   _   _   _   _   _
 / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \
( c )( 0 )( n )( s )( t )( r )( u )( c )( t )( i )( 0 )( n )
 \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/
  /\_/\  (
 ( ^.^ ) _)
   \"/  (
 ( | | )
(__d b__)

   Yay!
0xAF22 bytes successfully written to your_mind.jpg.
```



Flag: th3_ro4d_t0_succ3ss_1s_alw4ys_und3r_c0nstructi0n@flare-on.com