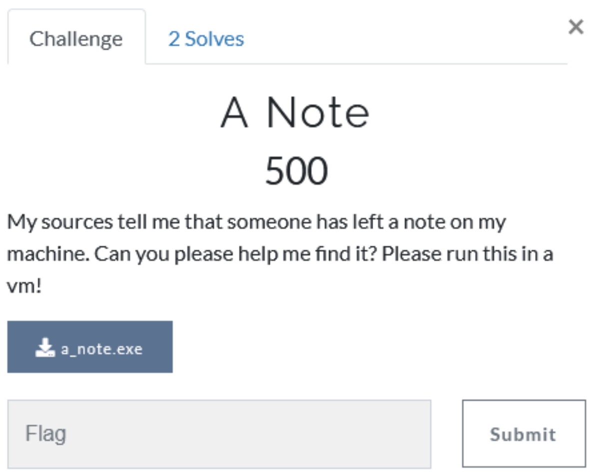
A Note - Greyhats CTF

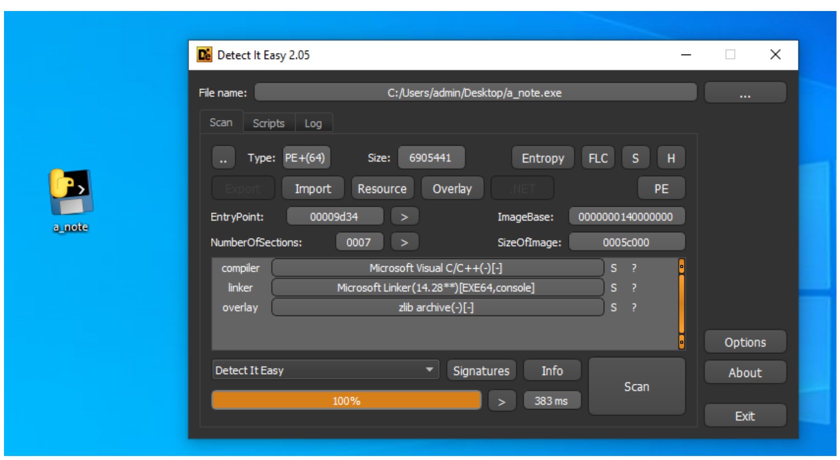
Overview

This is a relatively simple challenge with the right tool and technique. However, due to the layers of obscurity, not many people managed to solve this challenge.



Initial Analysis

This is a windows executable file. Since the author has asked us to run it in a vm, we will just put it in a windows 10 vm for analysis.



The first hint we get is from the icon and DiE. It is clearly a python executable. So it will need to unpack the python runtime and then execute the code. Also, it means that analysing the executable directly in disassembler / debugger will be very difficult as it involves decompression and then execution. We will need to get the core files somehow.

Retrieving the core files

To retrieve the files, there are 2 methods. `pyinstxtractor` and `temp file extraction`

method 1: pyinstxtractor

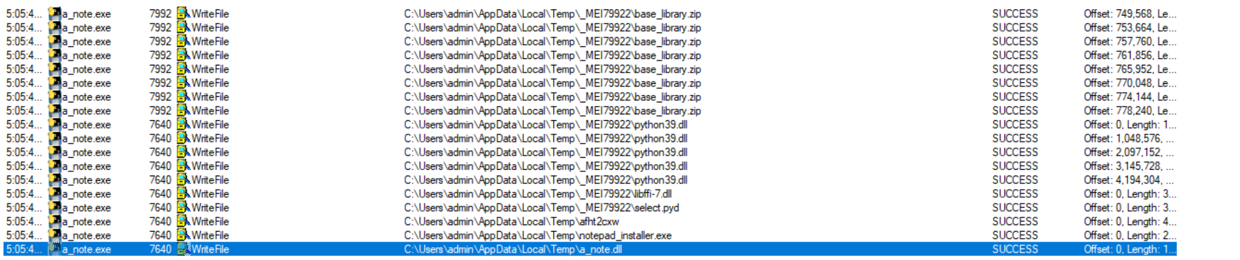
Using pyinstxtractor is simple, just grab the file from https://github.com/extremecoders-re/pyinstxtractor and run it accordingly.

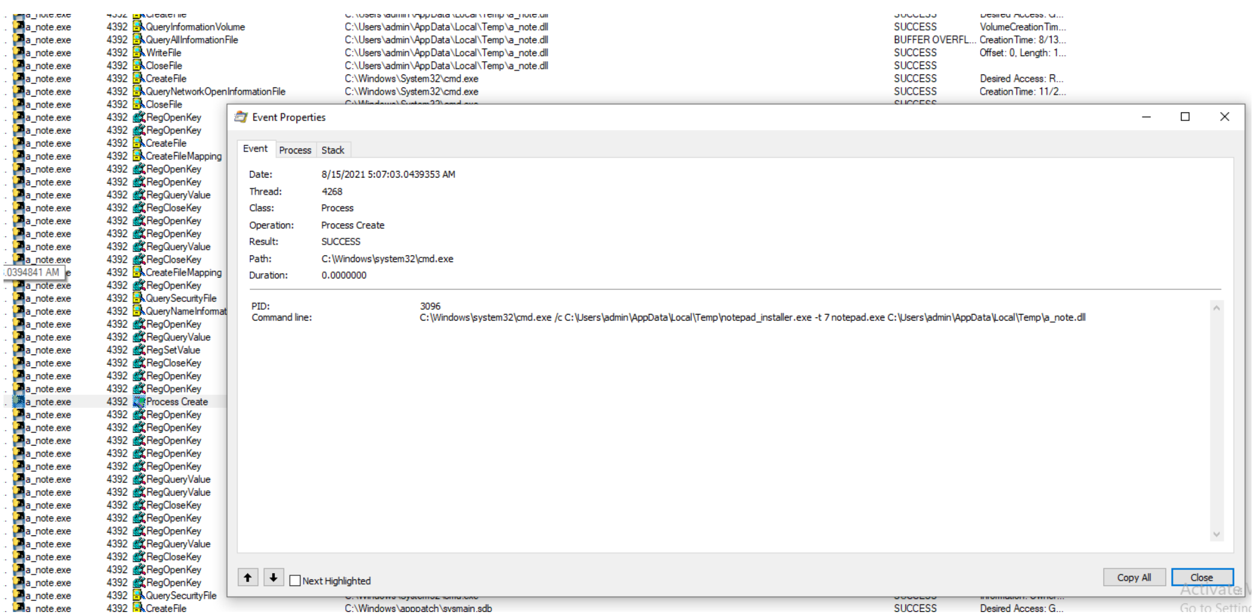
```
PYZ-00.pyz            _bz2.pyd      _lzma.pyd             _socket.pyd          base_library.zip   pyexpat.pyd                                               pyi_rth_pkgutil       pyimod03_importers   struct
PYZ-00.pyz_extracted  _ctypes.pyd   _multiprocessing.pyd  _ssl.pyd             libcrypto-1_1.dll  'pyi-windows-manifest-filename a_note.exe.manifest'   pyiboot01_bootstrap   pyimod04_ctypes      unicodedata.pyd
VCRUNTIME140.dll      _decimal.pyd  _overlapped.pyd       a_note               libffi-7.dll       pyi_rth_inspect                                           pyimod01_os_path      python39.dll
_asyncio.pyd          _hashlib.pyd  _queue.pyd            a_note.exe.manifest  libssl-1_1.dll     pyi_rth_multiprocessing                                   pyimod02_archive      select.pyd
```

The a_note file is the core program. However, it does not have a valid header and I am lazy to fix it. Running strings on the a_note file reveals that it is using injectallthings to inject a dll into notepad.exe. And there are 2 very long base64 strings that can be decoded into binary files that are `injectallthings` and `injected.dll` respectively. From here we can just analyse the dll for the flag. However, this method of using strings to make educated guesses made me a little uneasy, so I went to verify with method 2.

method 2: temp file extraction

This is a common technique in malware analysis. First, we use `process monitor` to check what API calls this program make and why is it flagged as malware by windows defender.





These API calls basically confirms that the program is unpacking the files into temp folder then using Process Create `cmd.exe` to call the injector. Now we can retrieve the a_note.dll from the temp folder and analyze it with confidence.

DLL analysis

I skimmed through the injector and it does not seem to be very interesting. The core of the challenge should be in the DLL itself.

```
__int64 __fastcall sub_1800011C0(__int64 a1, int a2, _QWORD *a3)
{
  __int64 v6; // rbx
  unsigned int v7; // er14
  int v8; // edi
  CHAR *v9; // rax
  int v10; // ecx
  int v11; // edx
  _DWORD *v12; // rdi
  char v13; // r9
  unsigned __int64 v14; // rcx
  LPCSTR *v15; // rax
  const CHAR *v16; // rdx
  CHAR *v17; // rcx
  DWORD nSize; // [rsp+20h] [rbp-E0h] BYREF
  int v20; // [rsp+24h] [rbp-DCh]
  char v21[16]; // [rsp+28h] [rbp-D8h] BYREF
  LPCSTR lpText[2]; // [rsp+38h] [rbp-C8h] BYREF
  unsigned __int64 v23; // [rsp+48h] [rbp-B8h]
  unsigned __int64 v24; // [rsp+50h] [rbp-B0h]
  __int128 v25[3]; // [rsp+58h] [rbp-A8h]
  char v26; // [rsp+88h] [rbp-78h]
  CHAR Buffer[65536]; // [rsp+90h] [rbp-70h] BYREF

  v6 = 0i64;
  v20 = 0;
  strcpy(v21, "hackerman");
  v25[0] = xmmword_1800032E8;
  v25[1] = xmmword_1800032F8;
  v25[2] = xmmword_180003308;
```

After a few steps from the DLLMain, we get to the main challenge function. The string "hackerman" also gives me confidence that this is not one of the standard library functions.

```
else if ( GetComputerNameA(Buffer, &nSize) )
{
  qword_1800056E0 = a1;
  if ( v7 == 1628938800 )                     // epoch time stamp
    goto LABEL_27;
  v9 = Buffer;
  do
  {
    v10 = (unsigned __int8)v9[v21 - Buffer];
    v11 = (unsigned __int8)*v9 - v10;
    if ( v11 )
      break;
    ++v9;
  }
  while ( v10 );
  if ( v11 )
  {
LABEL_27:
    MessageBoxA(0i64, "Hello there!", "Greetings ", 0);
  }
  else
  {
    srand(v7);
    otp_table = otp_byte_1800056F0;
    do
      *otp_table++ = rand() % 100 + 1;
    while ( (__int64)otp_table < (__int64)&unk_1800057B0 );
    lpText[0] = 0i64;
    v23 = 0i64;
    v24 = 15i64;
    ((void (__fastcall *)(void *))sub_1800014C0)(lpText);// string constructor?
    v20 = 1;
    do
    {
      v13 = *((_BYTE *)v25 + v6) ^ otp_byte_1800056F0[4 * v6];
      v14 = v23;
```

The core of the algorithm is in the red boxes. Essentially, what this function is doing is a one-time-pad (OTP) encryption. However, we know how the padding is generated and the seed for the random generation. We made an educated guess here because of the if check on `v7.` The seed is 1628938800 and the generation is using rand()%100+1. The encrypted blob is located at v25 which is embedded in the DLL.

So, with all these information, we are able to recreate the program and apply the OTP to get back the plaintext flag.

```
#include <cstdio>
#include <cstdlib>

int main() {
  srand(1628938800);
  char table[50] = { 0x3A, 0x7A, 0x58, 0x66, 0x75, 0x5C, 0x27, 0x3B, 0x58, 0x20, 0x22, 0x59, 0x3D, 0x6C, 0x5A, 0x4C, 0x77, 0x6F, 0x41, 0x7D
    0x64, 0x4F, 0x25, 0x6, 0x50, 0x6C, 0x4E, 0x78, 0x16, 0x13, 0x2A, 0x23, 0x52, 0x51, 0x17, 0x00, 0x5E, 0x26, 0x58};

  for (int i = 0; i < 48; i++) {
    printf("%c", table[i] ^ (rand() % 100 + 1));
  }
  return 0;
}
```

There was 1 little caveat that I found out during the recreation of the program, linux gcc rand() must be using a different algorithm from Microsoft Visual C++ compiler. Since the same program yields very different result. Windows compiler ends up giving me the right result.

Conclusion

This is quite a simple and straight forward challenge, I am surprised that not many people have managed to solve this. I think the difficulty comes from 2 aspects

1. The realization that this is a pyinstaller executable and not a regular py2exe or windows executable. Using the wrong tools will result in a lot of wasted time
2. Extraction of the files, which is based upon understanding the type of executable we are dealing with and how to extract them

Once we have the DLL file, this reverse engineering challenge becomes rather simple. Overall, I think this is quite an interesting challenge to expose people to the versatility of windows executables.