Unpacking Malware Manually

blog.cyber5w.com/the-most-known-unpacking-technique

Objectives

In this blog post, we will go through a famous packing technique which is the use of VirualAlloc and VirtualProtect to decrypt data in memory and execute it, and how to unpack it manually, we are going to apply it to **Death Ransomware** malware

Introduction

What is packed malware?

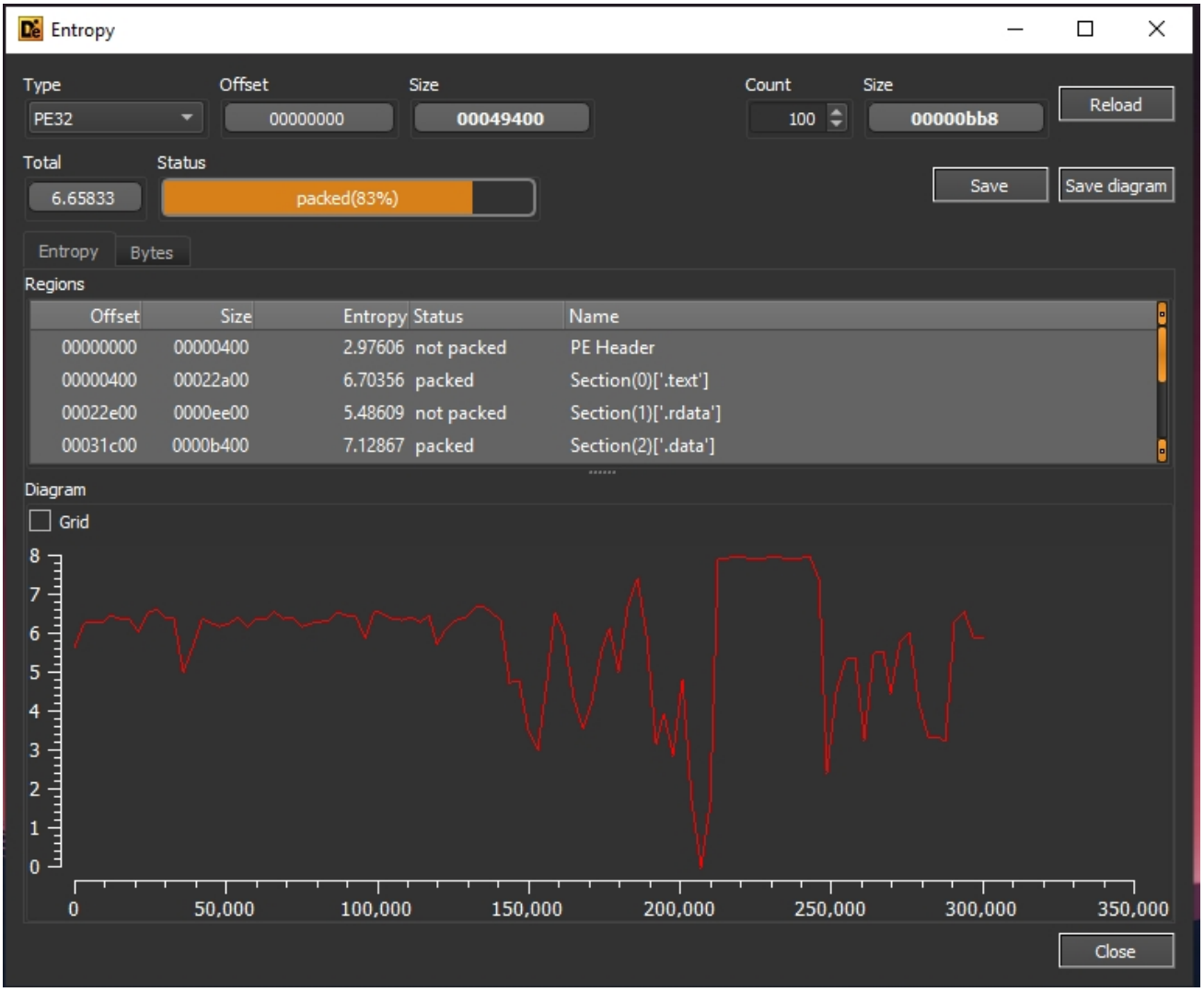
packed malware refers to malicious software that has been compressed and/or encrypted to obfuscate its code and make it more difficult to detect by antivirus or other security solutions.

Static Analysis

Let’s open the sample in DIE to see if it is packed or not

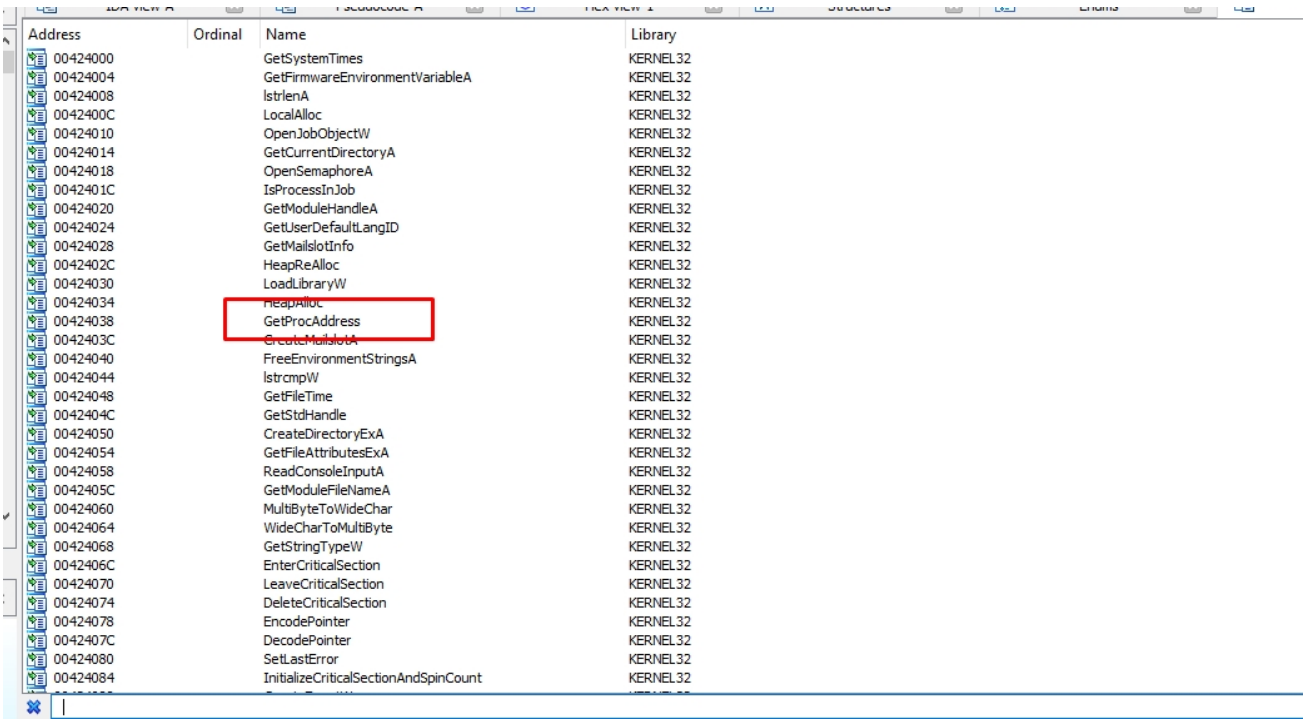
DIE is a tool that detects if the malware is packed or not. It does this by measuring the entropy of the file, which is a measure of randomness. If the data in a file is more random, it usually means that the file is packed.

When the entropy of a file is greater than 7, it generally indicates that the file is likely compressed or encrypted.



Yeah It’s packed

Let’s see its imports in IDA



Address	Ordinal	Name	Library
00424000		GetSystemTimes	KERNEL32
00424004		GetFirmwareEnvironmentVariableA	KERNEL32
00424008		lstrlenA	KERNEL32
0042400C		LocalAlloc	KERNEL32
00424010		OpenJobObjectW	KERNEL32
00424014		GetCurrentDirectoryA	KERNEL32
00424018		OpenSemaphoreA	KERNEL32
0042401C		IsProcessInJob	KERNEL32
00424020		GetModuleHandleA	KERNEL32
00424024		GetUserDefaultLangID	KERNEL32
00424028		GetMailslotInfo	KERNEL32
0042402C		HeapReAlloc	KERNEL32
00424030		LoadLibraryW	KERNEL32
00424034		HeapAlloc	KERNEL32
00424038		GetProcAddress	KERNEL32
0042403C		CreateMailslotA	KERNEL32
00424040		FreeEnvironmentStringsA	KERNEL32
00424044		lstrcmpW	KERNEL32
00424048		GetFileTime	KERNEL32
0042404C		GetStdHandle	KERNEL32
00424050		CreateDirectoryExA	KERNEL32
00424054		GetFileAttributesExA	KERNEL32
00424058		ReadConsoleInputA	KERNEL32
0042405C		GetModuleFileNameA	KERNEL32
00424060		MultiByteToWideChar	KERNEL32
00424064		WideCharToMultiByte	KERNEL32
00424068		GetStringTypeW	KERNEL32
0042406C		EnterCriticalSection	KERNEL32
00424070		LeaveCriticalSection	KERNEL32
00424074		DeleteCriticalSection	KERNEL32
00424078		EncodePointer	KERNEL32
0042407C		DecodePointer	KERNEL32
00424080		SetLastError	KERNEL32
00424084		InitializeCriticalSectionAndSpinCount	KERNEL32

Virtual Alloc, **Virtual protect** are not listed, but I think that the malware resolves them dynamically

As we can see the sample resolves **Virtual protect**

VirtualAlloc and **VirtualProtect** are two Windows API functions commonly used by the malware to unpack itself.

Malware uses **VirtualAlloc** to allocate memory for the unpacked malware code then uses **VirtualProtect** to change the protection to mark the memory allocated as executable, writable, or both to be able to execute the dynamically unpacked code.

```
196  for ( i = 0; i < 7239525; ++i )
197  {
198    GetSystemTimes(&IdleTime, &KernelTime, &UserTime);
199    if ( i < 235234 )
200      CreateMailslotA(0, 0, 0, 0);
201  }
202  sub_423179();
203  strcpy(String, "Virtual");
204  v13 = hModule;
205  strcat(String, "Protect");
206  ProcAddress = GetProcAddress(v13, String);
207  dword_456254 = (int)ProcAddress;
208  v9 = 0;
209  do
210  {
211    if ( v9 == 3444570 )
212    {
213      ((void (__stdcall *)(int (*)(void), SIZE_T, int, char *))ProcAddress)(dword_45A0E4, uBytes, 64,
214      ProcAddress = (FARPROC)dword_456254;
215    }
216    ++v9;
```

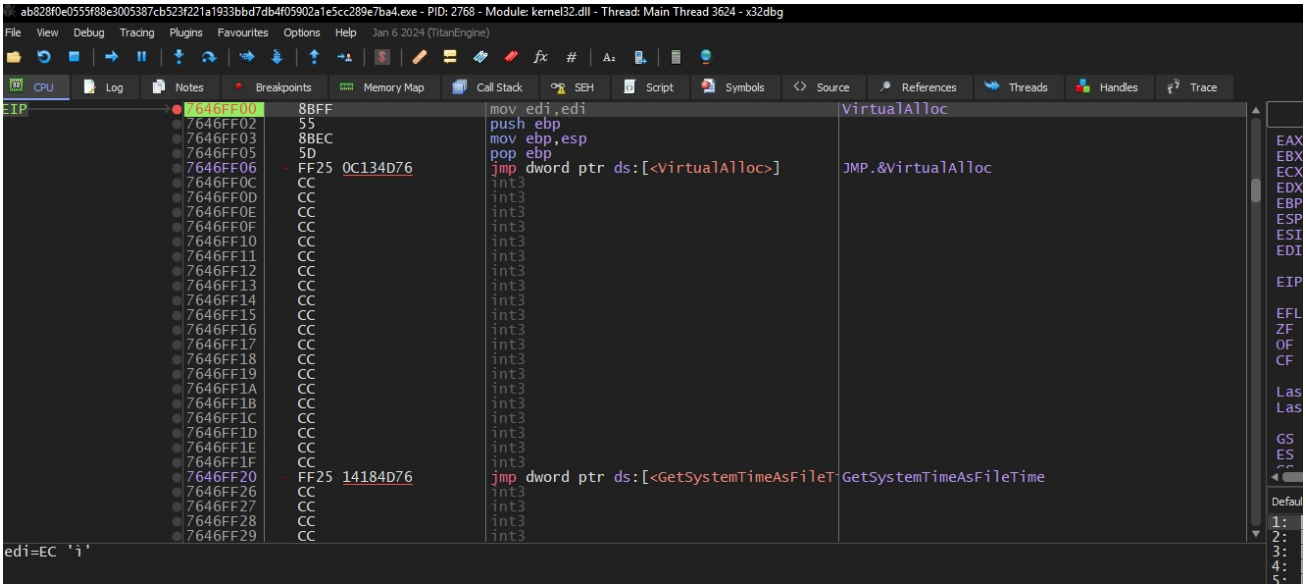
Let’s open our sample into x64dbg

I’ll put a breakpoint in **VirtualAlloc**

Press ctrl+g and write in the search bar “VirtualAlloc” and click ok.

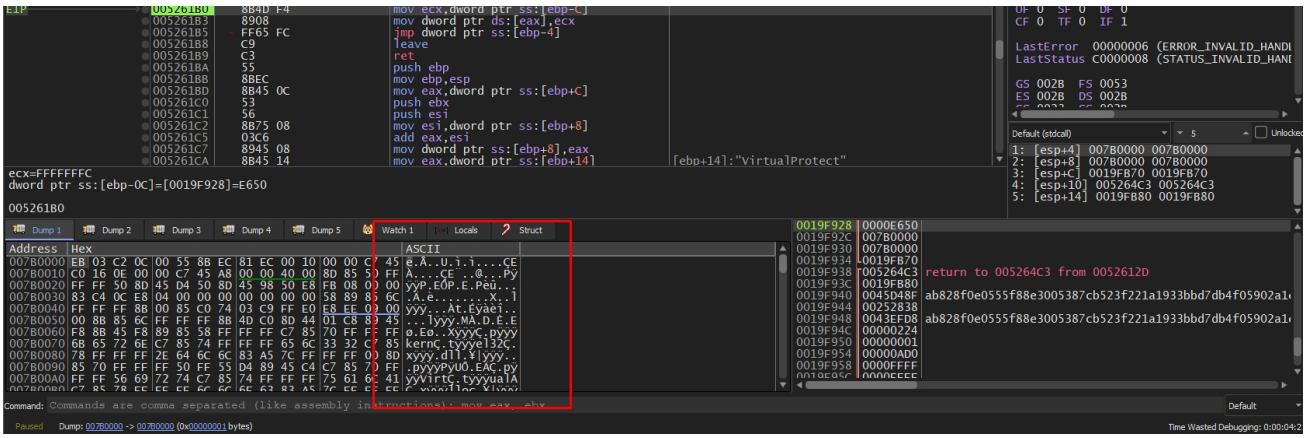
To put a breakpoint in **VirtualAlloc** we need to click on the circle on the left side of the **VirtualAlloc** instruction

Let’s run the sample until we hit the breakpoint



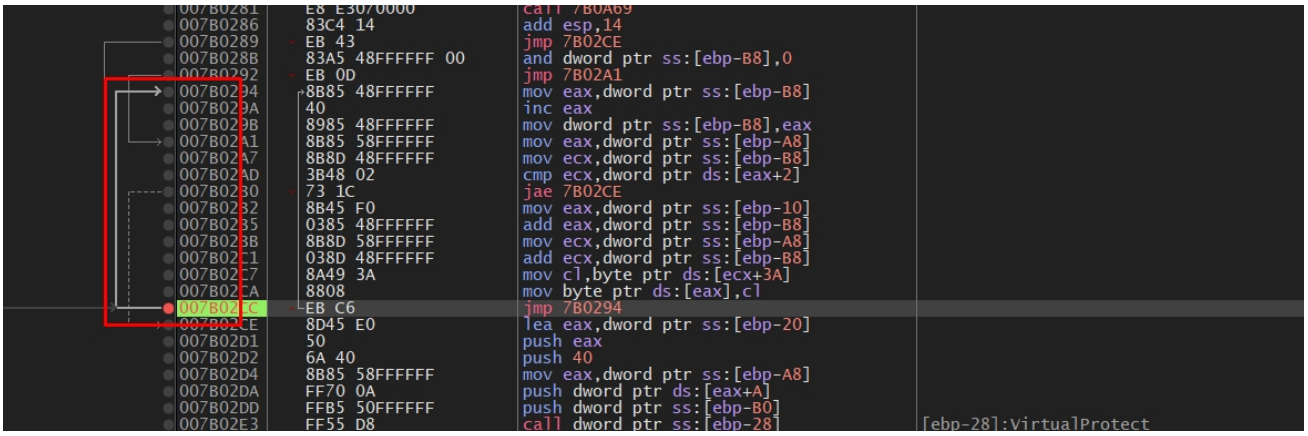
Let’s go to the return of the function and step over it and follow **EAX** In a dump.

After stepping over some code there is some data written into the dump

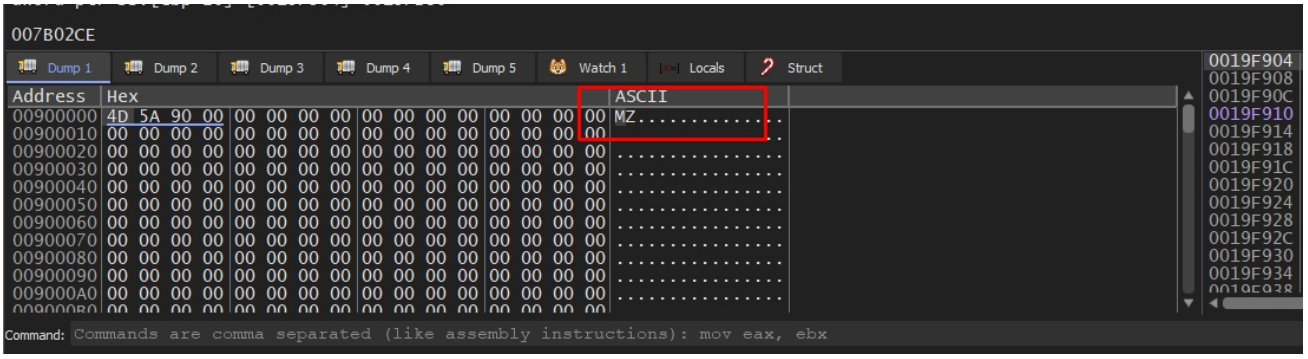


Let’s run the debugger to hit the second **VirtualAlloc** function and do the same thing we did above.

After some stepping over we can see a loop. I’ll put a breakpoint at the end of it.

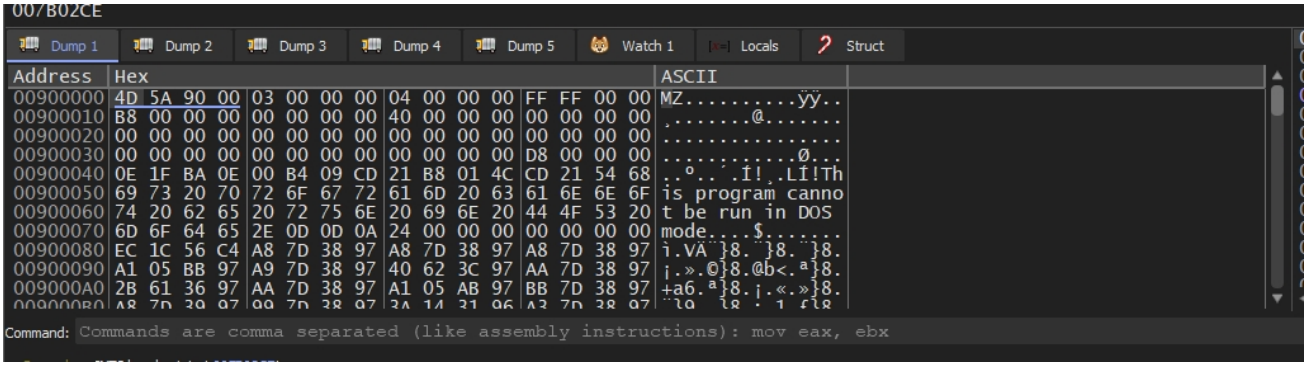


Let's run the malware

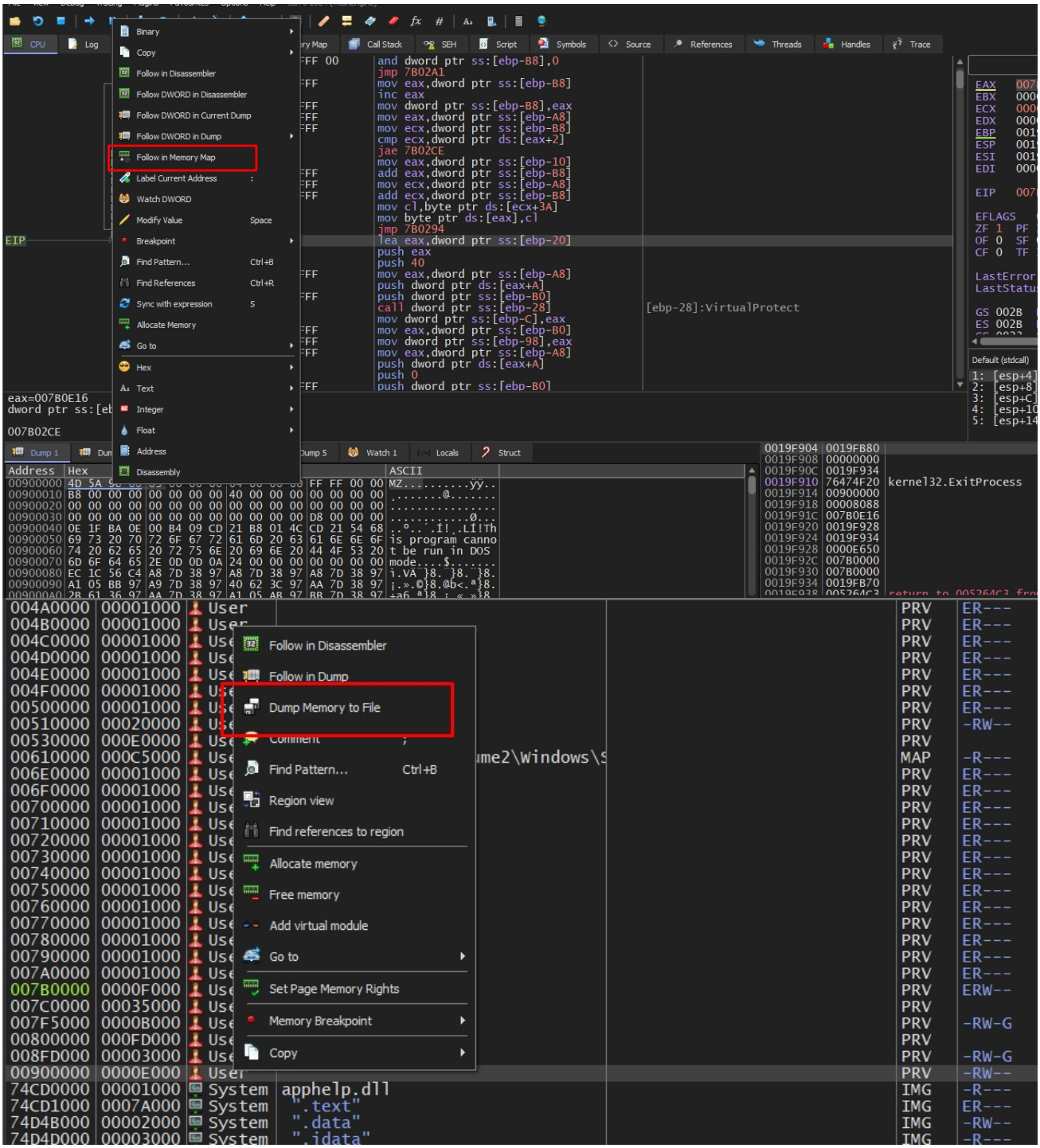


A **PE** file is being written in the dump.

This is the final result



Let's follow this in the memory map and dump it into a file



Let’s see the dumped file in IDA

```
24  int v21; // [esp-Ch] [ebp-1012Ch]
25  int v22; // [esp-Ch] [ebp-1012Ch]
26  int v23; // [esp-8h] [ebp-10128h]
27  int v24; // [esp-8h] [ebp-10128h]
28  DWORD cbData; // [esp+10h] [ebp-10110h] BYREF
29  int v26; // [esp+18h] [ebp-10108h]
30  int v27; // [esp+1Ch] [ebp-10104h]
31  BYTE Data[256]; // [esp+20h] [ebp-10100h] BYREF
32  WCHAR Buffer[32768]; // [esp+120h] [ebp-10000h] BYREF
33
34  UserDefaultLangID = GetUserDefaultLangID();
35  v26 = 1049;
36  if ( UserDefaultLangID == 1049 )
37    goto LABEL_31;
38  if ( UserDefaultLangID == 1087 )
39    goto LABEL_31;
40  v27 = 1059;
41  if ( UserDefaultLangID == 1059 )
42    goto LABEL_31;
43  cbData = 1058;
44  if ( UserDefaultLangID == 1058 || UserDefaultLangID == 1092 )
45    goto LABEL_31;
46  KeyboardLayoutList = GetKeyboardLayoutList(0, 0);
47  ProcessHeap = GetProcessHeap();
48  v3 = (HKL *)HeapAlloc(ProcessHeap, 8u, 4 * KeyboardLayoutList);
49  v4 = v3;
0000AB60 start:13 (40B760)
```

The malware is successfully unpacked

SHA256:ab828f0e0555f88e3005387cb523f221a1933bbd7db4f05902a1e5cc289e7ba4

This blog is authored by **Mostafa Farghaly(M4lcode)**.

Previous Post

Hard disk structure and analysis

Next Post

Email Forensics