SHA256: eea0083ac04f8bfb3042acd615cabd7be77dee410c8db229b5980379b224e93f

Hancitor is typically delivered through phishing emails containing malicious attachments. These attachments are often weaponized Microsoft Office documents that have macros embedded within them.

Spread through emails, Hancitor tells users to enable macros on the document which allows it to connect to its C2 server to download its second stage loader.

Analysis

We unpack the zip file and analyze the contents on TrID, DiE, and PeStudio to gather information through an initial investigation on the contents of the binary.

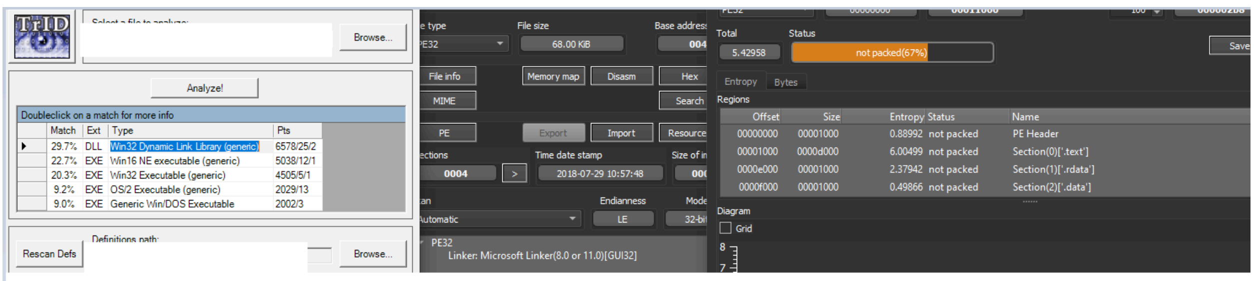


Fig 1. Initial Scan through TrID and DiE.

Looking at the results on Fig 1. It shows that they have detected this as a 32-bit DLL and has a relatively low entropy suggesting that there is a chance that this is neither packer nor encrypted.

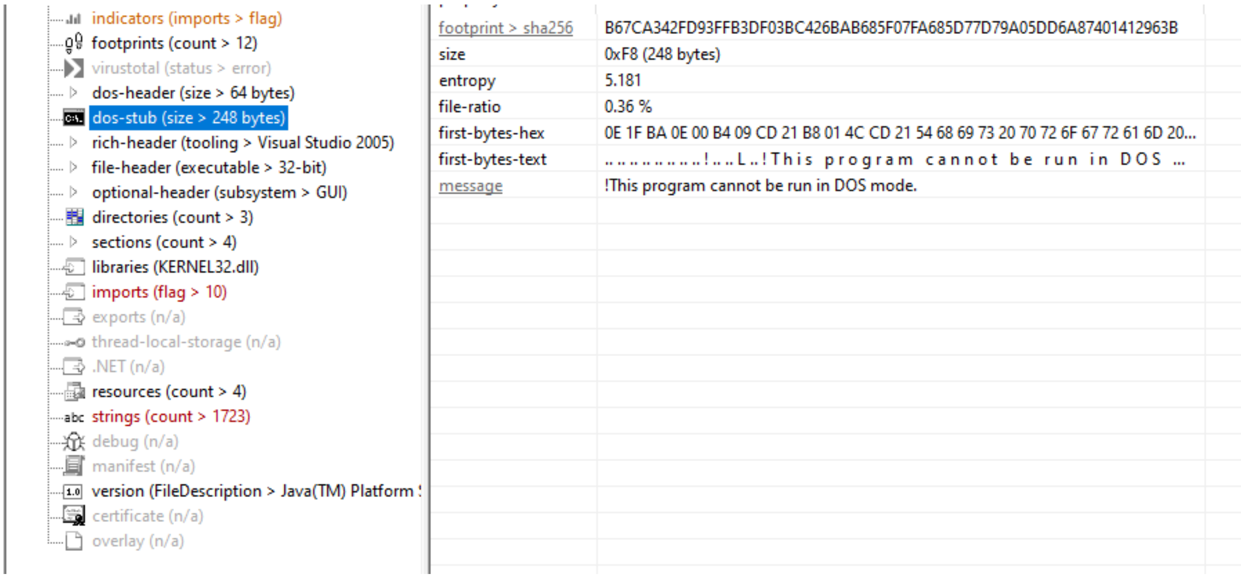


Fig 2. Screenshot of PeStudio Findings.

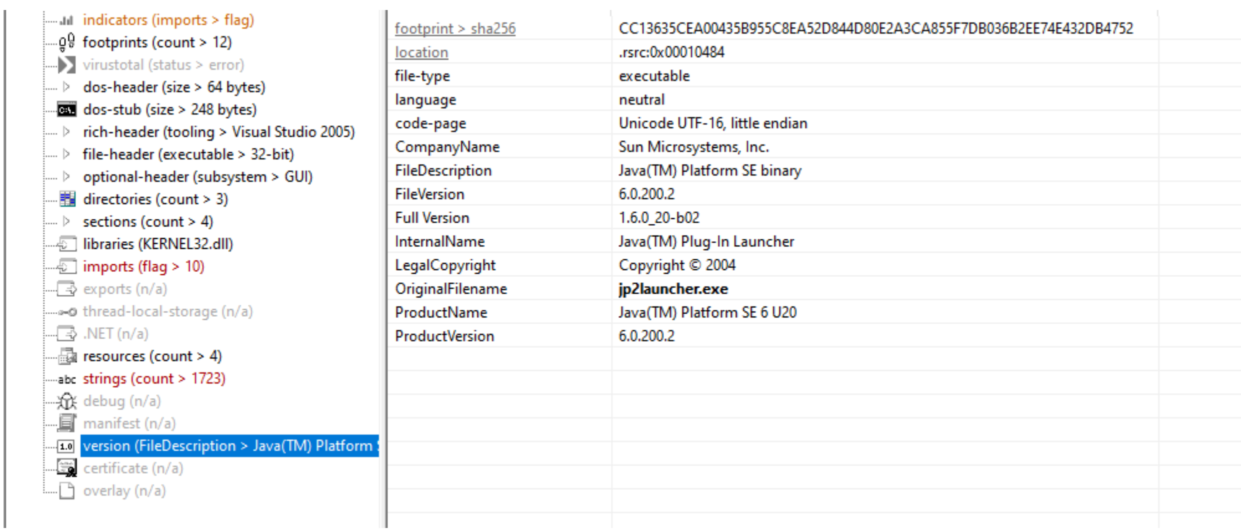


Fig 3. Versions Tab on PEStudio

Analyzing the file through PEStudio shows a few flags such as the low import count. It is also possible to look at the available strings but it does not show much as the data is possibly obfuscated or encrypted.

Running the file on Virustotal shows that this is already a well documented piece of malware and anti-viruses should not have any struggles identifying this.

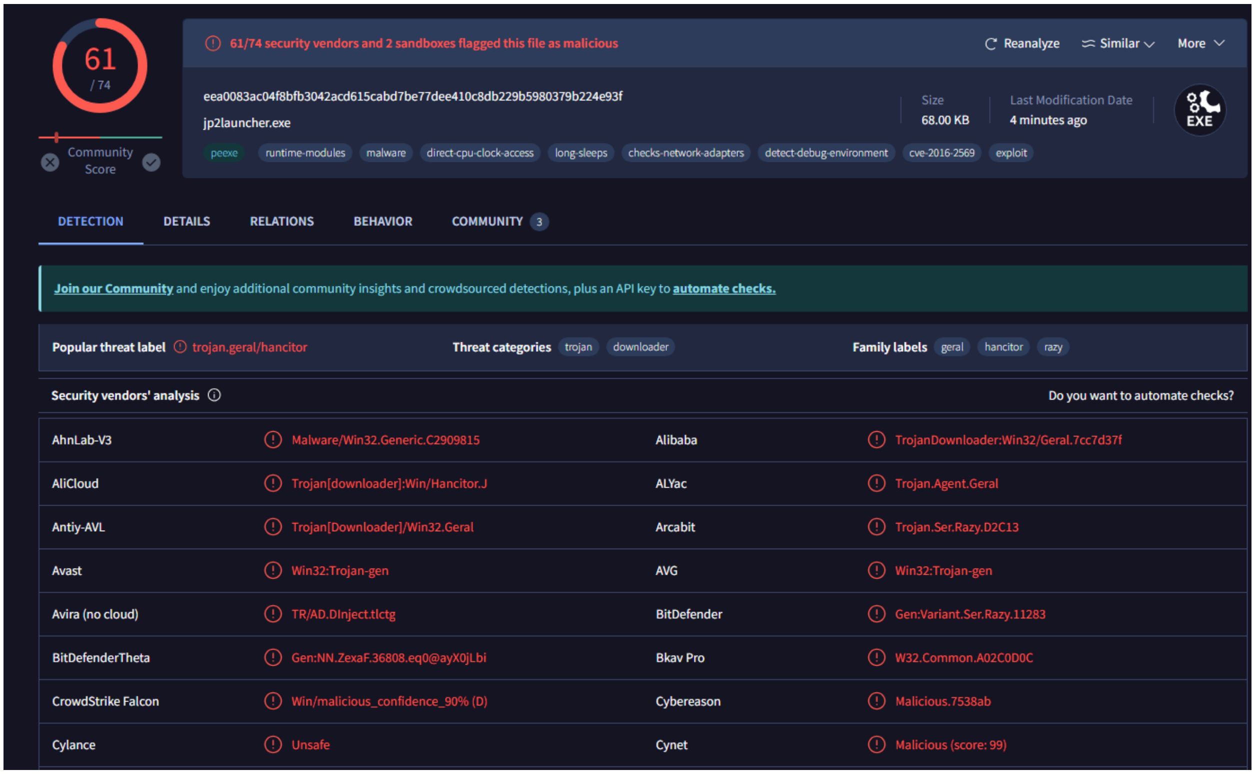


Fig 4. Virustotal Scan

- **Dynamic Analysis:** The next section shows the dynamic analysis on x32dbg and ProcessHacker to monitor if any files are created.

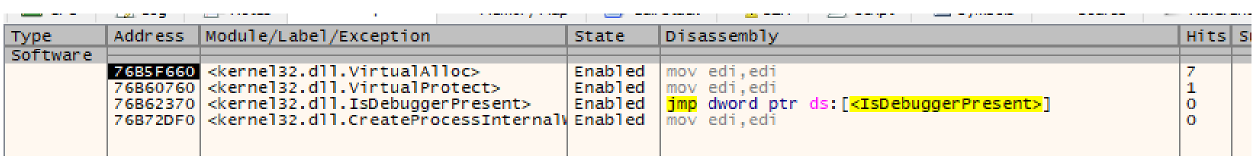


Fig 5. Breakpoints Set

Following from Fig 5. Only VirtualAlloc and VirtualProtect were hit on the four breakpoints that were set Following this on the debugger will allow us to unpack the second stage shown below.

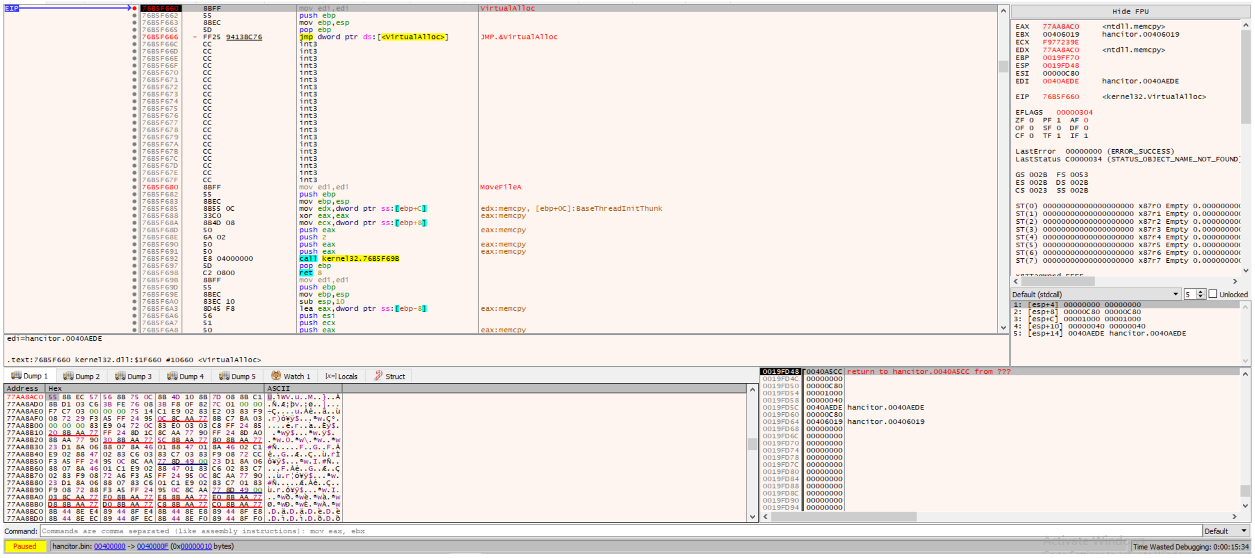


Fig 6. First VirtualAlloc

Hitting the first VirtualAlloc and following EAX to the return shows a bunch of allocated data on Dump1.

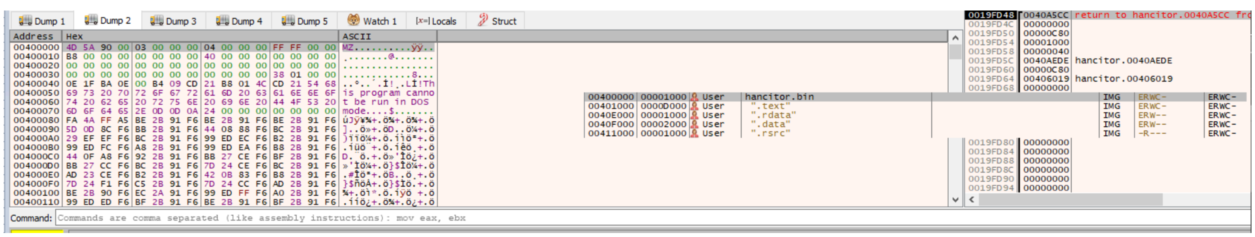


Fig 7. MZ on Dump2

Viewing Dump2 actually shows that it contains an MZ header but following that in memory it is probably just the original entrypoint for Hancitor.

Fig 8. Following through VirtualAlloc Calls.

Continuing on the unpacking process by following VirtualAlloc shows that the data is interestingly unpacked on even rows of data per call. This process is repeated seven times as seen on Fig 5.

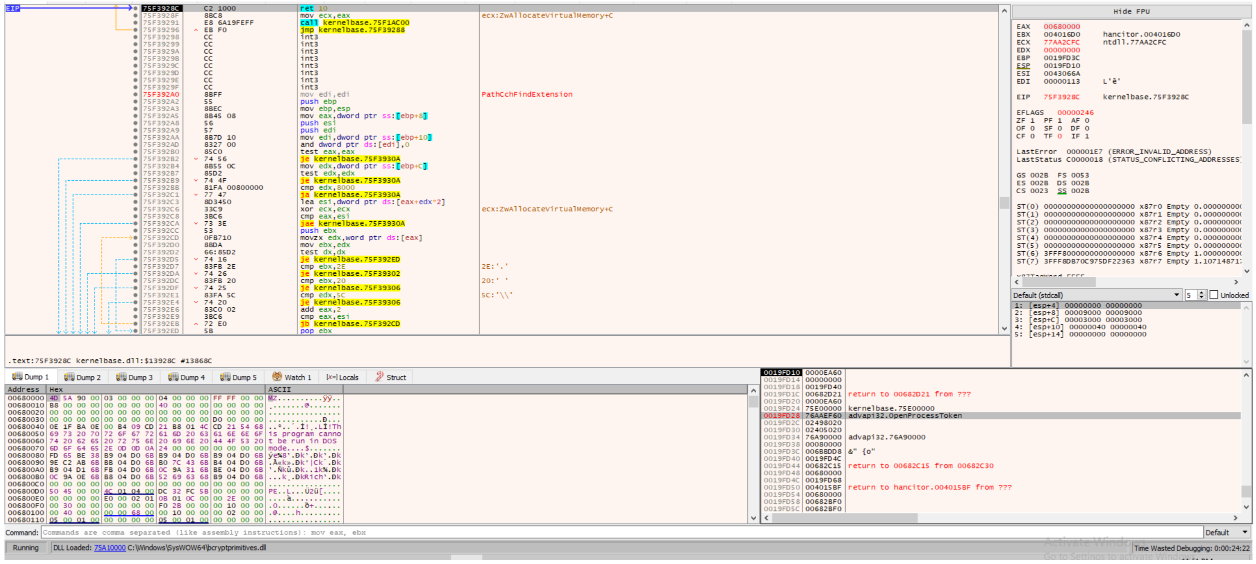


Fig 9. MZ on Last VirtualAlloc Call

After hitting the return on the final VirtualAlloc Call and following EAX in dump shows that data has been unpacked at this address.

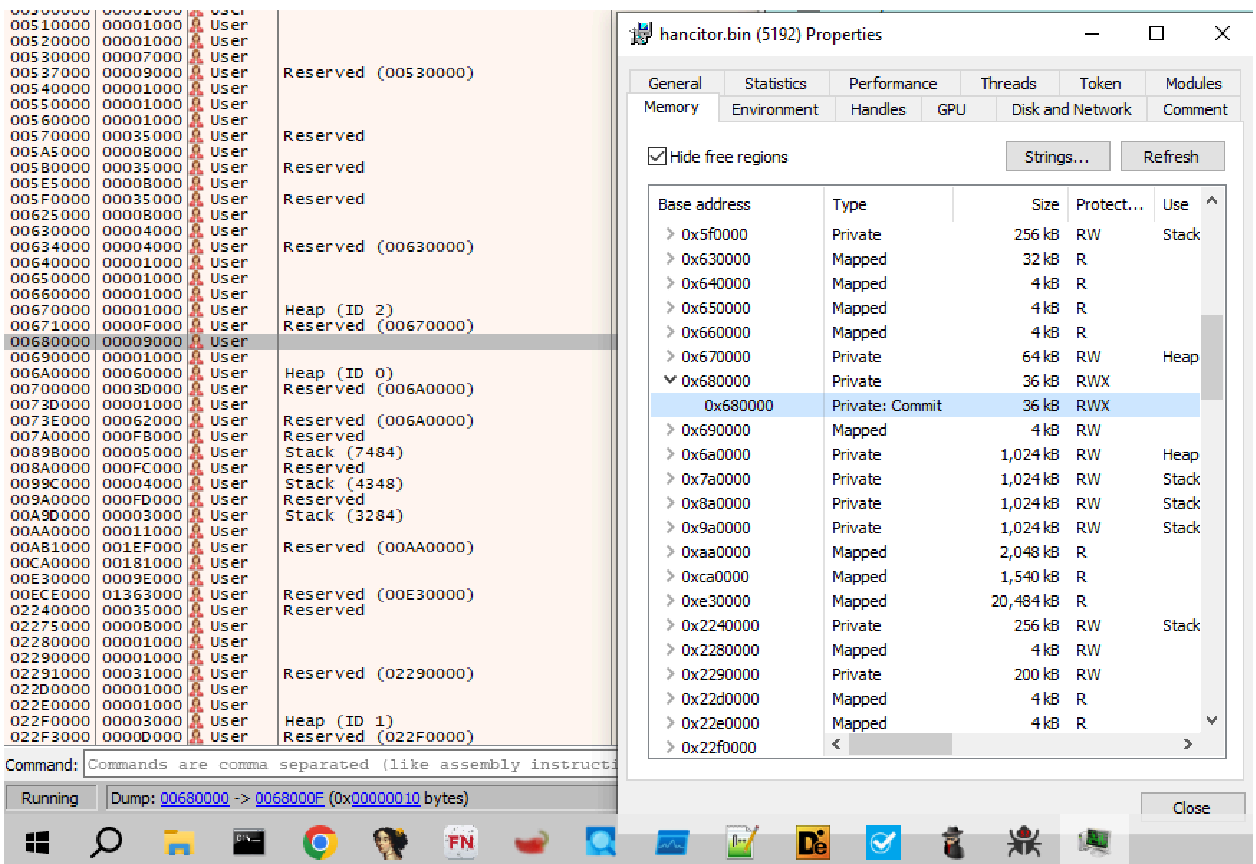


Fig 10. Dumping from Memory.

After finding the MZ file, we dump the binary to a file from memory through x32dbg or Process Hacker. However, attempting to open the binary on IDA comes up with issues and it looks like it is necessary to fix the import address table and to rebase the binary on PE-Bear first.

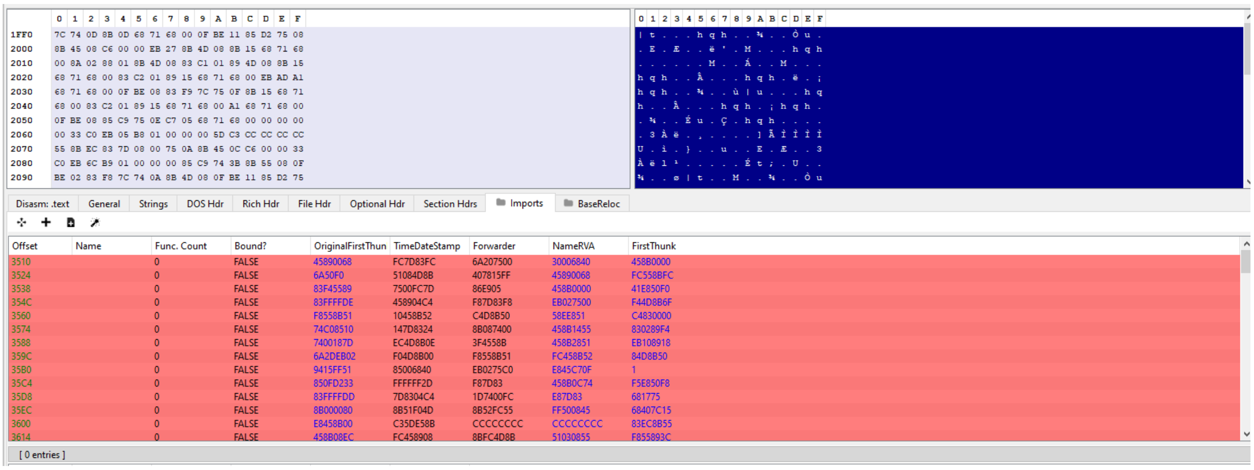


Fig 11. Broken IAT

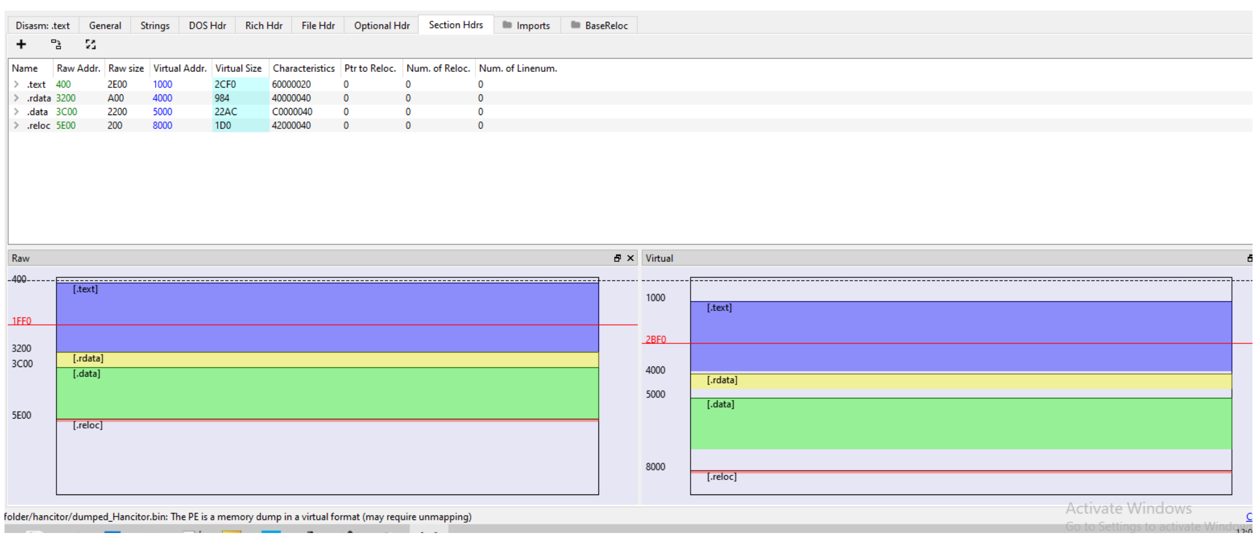
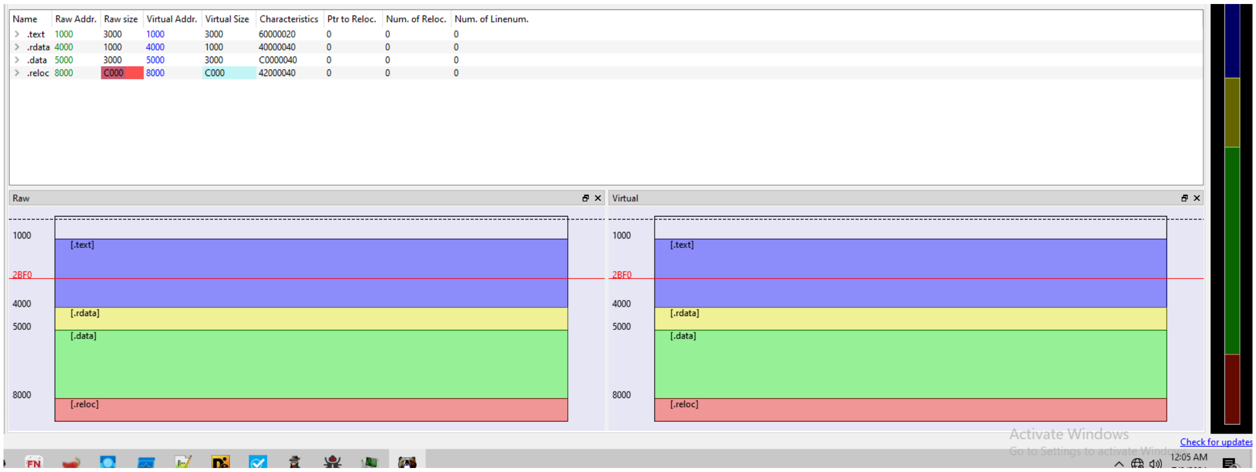


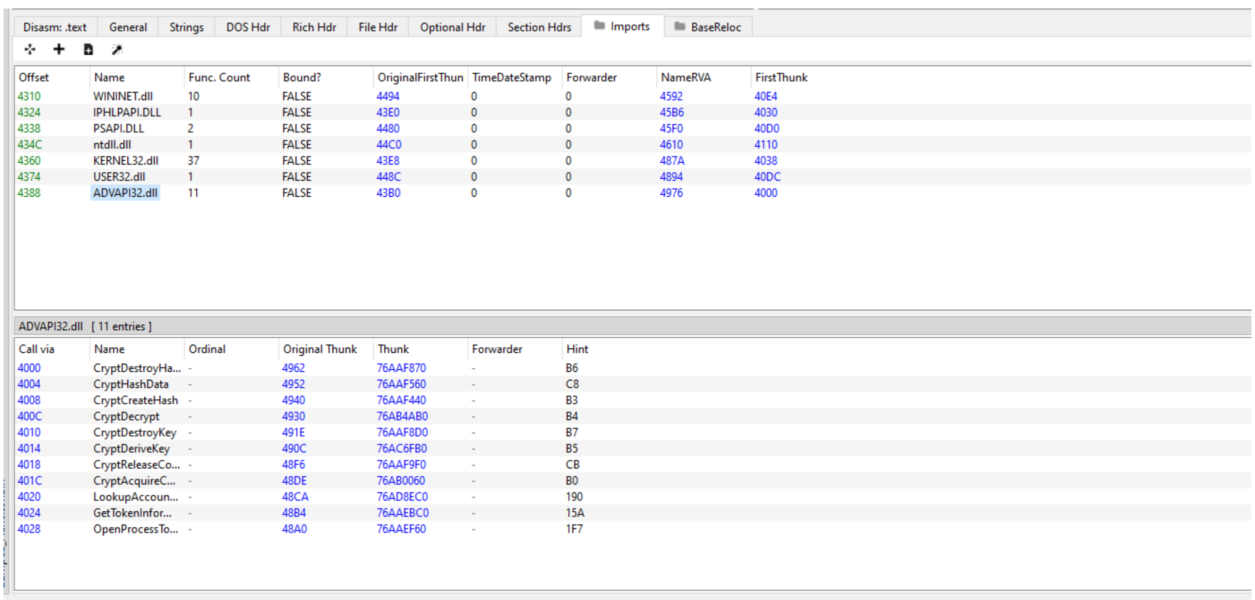
Fig 12. Fixing the IAT

To fix the import address table; the Raw address is initially changed to match the virtual address and subtracting the values of the raw address and applying the value to raw size. Afterwards, match the virtual size with the raw size and try to fill in the memory for the last slot in the raw size. The image base in the Optional Hdr tab may also be needed to match the address from where we dumped.



Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
.text	1000	3000	1000	3000	60000020	0	0	0
.rdata	4000	1000	4000	1000	40000040	0	0	0
.data	5000	3000	5000	3000	C0000040	0	0	0
.reloc	8000	C000	8000	C000	42000040	0	0	0

Fig 13. Fixed IAT



Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA	FirstThunk
4310	WININET.dll	10	FALSE	4494	0	0	4592	40E4
4324	IPHLPAPI.DLL	1	FALSE	43E0	0	0	45B6	4030
4338	PSAPI.DLL	2	FALSE	4480	0	0	45F0	40D0
434C	ntdll.dll	1	FALSE	44C0	0	0	4610	4110
4360	KERNEL32.dll	37	FALSE	43E8	0	0	487A	4038
4374	USER32.dll	1	FALSE	448C	0	0	4894	40DC
4388	ADVAPI32.dll	11	FALSE	43B0	0	0	4976	4000

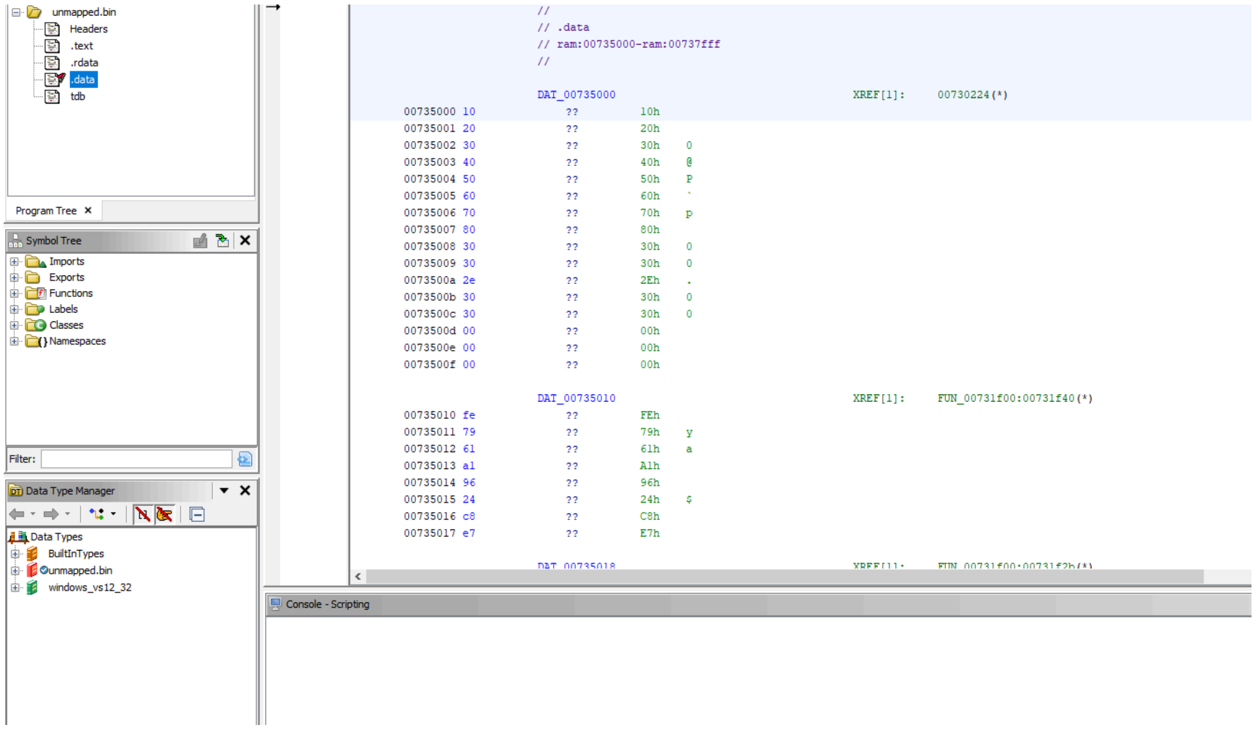
ADVAPI32.dll [11 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
4000	CryptDestroyHa...	-	4962	76AAF870	-	B6
4004	CryptHashData	-	4952	76AAF560	-	C8
4008	CryptCreateHash	-	4940	76AAF440	-	B3
400C	CryptDecrypt	-	4930	76AB4AB0	-	B4
4010	CryptDestroyKey	-	491E	76AAF8D0	-	B7
4014	CryptDeriveKey	-	490C	76AC6FB0	-	B5
4018	CryptReleaseCo...	-	48F6	76AAF9F0	-	CB
401C	CryptAcquireC...	-	48DE	76AB0060	-	B0
4020	LookupAccoun...	-	48CA	76AD8EC0	-	190
4024	GetTokenInfor...	-	48B4	76AAEBC0	-	15A
4028	OpenProcessTo...	-	48A0	76AAEF60	-	1F7

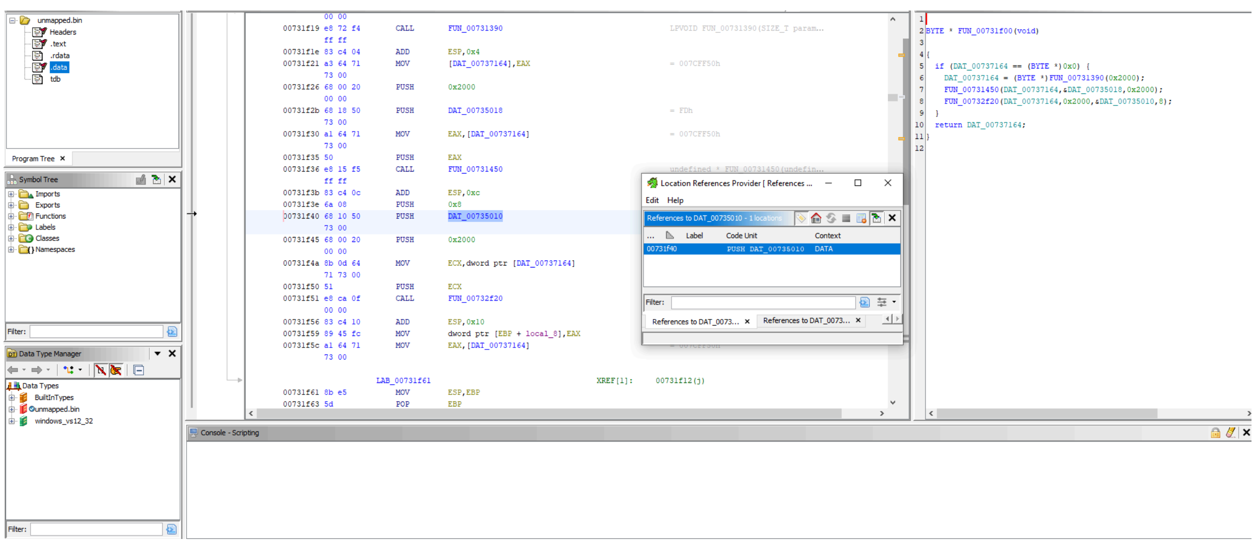
Fig 14. Fixed Imports Header

After this process is done. We then dump the unmapped file and open the unmapped file for the next stages of the analysis.

Viewing the data section we see upfront two suspicious data symbols.



Follow the references of these data



Can also see that it pushes a 0x8

Following the function that called the data leads us to an the malwars encryption routine

```

BYTE * FUN_00731f00(void)

{
  if (DAT_00737164 == (BYTE *)0x0) {
    DAT_00737164 = (BYTE *)FUN_00731390(0x2000);
    FUN_00731450(DAT_00737164,&DAT_00735018,0x2000);
    FUN_00732f20(DAT_00737164,0x2000,&DAT_00735010,8);
  }
  return DAT_00737164;
}

```

Notable parameters in the encryption calls are 0x8004 in CyptHashData, 0x6801, and 0x280011 on CryptDeriveKey. Where 0x8004 is the ALG_ID(from Wincrypt.h) for SHA1 and 0x6801 indicates that it will be using RC4. The parameter 0x280011 is interesting because Windows uses this to store two variables that it will use based from the most and least significant bit. We are more interested in the most significant word 0x0028h because this will determine the byte size which can be computed by converting 28h to decimal then dividing by 8 which gives us 40/8 or 5 bytes.

```
local_14 = 0;
BVar1 = CryptAcquireContextA(&local_c,(LPCSTR)0x0,(LPCSTR)0x0,1,0xf0000000);
if ((((BVar1 != 0) && (BVar1 = CryptCreateHash(local_c,0x8004,0,0,&local_8), BVar1 != 0)) &&
    (BVar1 = CryptHashData(local_8,param_3,param_4,0), BVar1 != 0)) &&
   ((BVar1 = CryptDeriveKey(local_c,0x6801,local_8,0x280011,&local_10), BVar1 != 0 &&
    (BVar1 = CryptDecrypt(local_10,0,1,0,param_1,&param_2), BVar1 != 0)))) {
  local_14 = param_2;
}
if (local_8 != 0) {
```

Decrypting the config on cyberchef

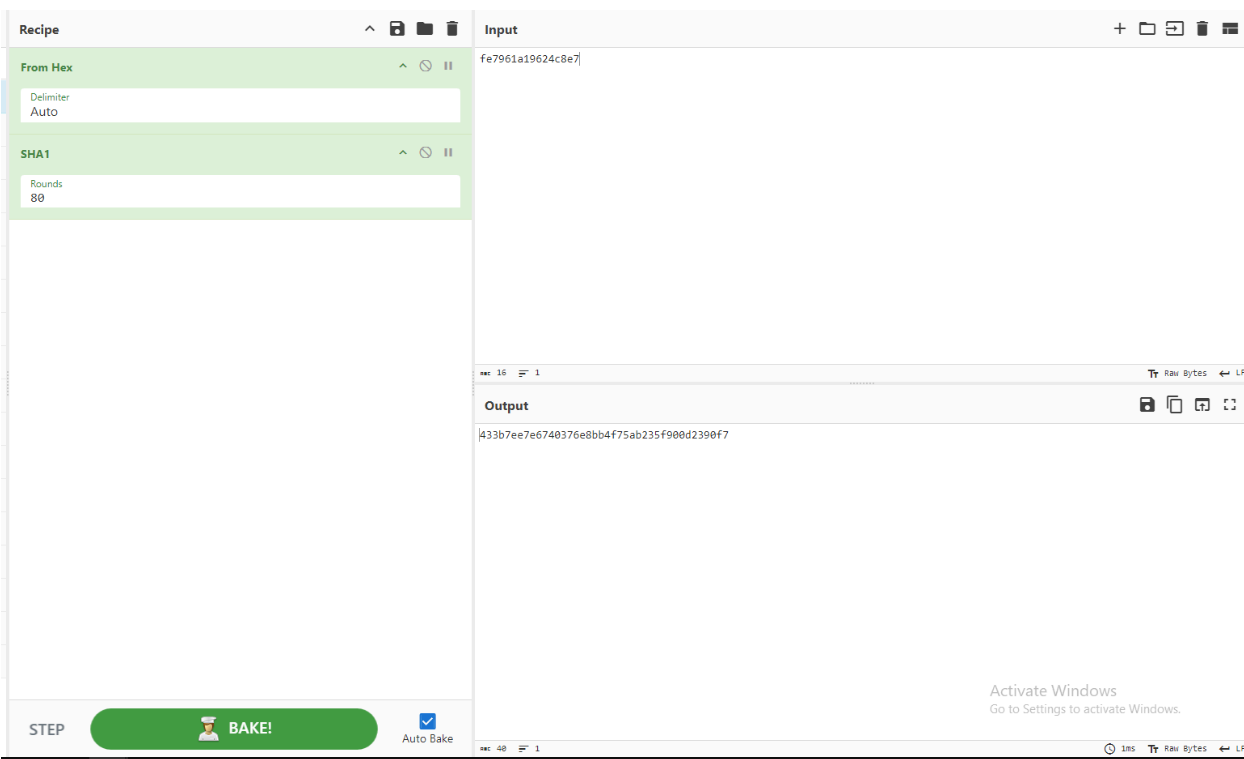
Copy the suspected key and the suspected encrypted configuration.

```
0073500f 00          ??          00h

                     DAT_00735010                        XREF[1]:     FUN_007:
00735010 fe          ??          FEh
00735011 79          ??          79h    y
00735012 61          ??          61h    a
00735013 a1          ??          A1h
00735014 96          ??          96h
00735015 24          ??          24h    $
00735016 c8          ??          C8h
00735017 e7          ??          E7h

                     DAT_00735018                        XREF[1]:     FUN_007:
00735018 fd          ??          FDh
00735019 52          ??          52h    R
0073501a 8a          ??          8Ah
0073501b ac          ??          ACh
0073501c 0a          ??          0Ah
0073501d 7f          ??          7Fh
0073501e 7f          ??          7Fh
0073501f 7c          ??          7Ch    |
```

Convert from hex then sha1 which gives us 433b7ee7e6740376e8bb4f75ab235f900d2390f7. We will only take 5 bytes from this as our decryption key as set on the encryption routine which leaves us with 433b7ee7e6 (two characters per byte).



We then input the encrypted configuration as a hex input on the RC4 recipe on cyberchef and put our key back as hex on the same recipe which gives us the decrypted configuration.