

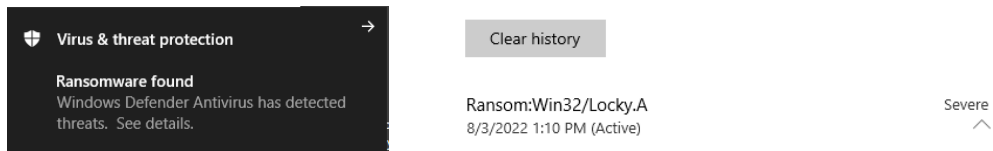
Malware Analysis Homework 4

Danilo Dell'Orco

0300229

1 Introduzione

Per analizzare il comportamento del programma, lanciamo l'eseguibile su una macchina virtuale Windows 10. Dopo aver copiato l'eseguibile sulla macchina, vediamo che viene riconosciuto come **Ransomware** da *Windows Defender*, in particolare di tipo **LockyA**.



Creiamo una nuova istantanea della macchina, disabilitiamo Windows Defender e proseguiamo con l'analisi dell'eseguibile.

2 Analisi di Base

2.1 Analisi Statica di Base

Effettuiamo innanzitutto un'analisi statica delle stringhe, sfruttando lo strumento *Defined Strings* di Ghidra. Dalla lista, vediamo sono presenti le seguenti stringhe:

- UPX0
- UPX1

A screenshot of the 'Defined Strings' window in Ghidra. The window title is '0101 DAT Defined Strings - 19 items'. It contains a table with four columns: 'Location', 'String Value', 'String Represent...', and 'Data Type'.

Location	String Value	String Represent...	Data Type
00400000	MZ	"MZ"	char[2]
00400080	PE	"PE"	char[4]
00400178	UPX0	"UPX0"	char[8]
004001a0	UPX1	"UPX1"	char[8]
004001c8	.rsrc	".rsrc"	char[8]
004a1ba8	Process emory	"Process emory"	ds

Questo ci suggerisce che l'eseguibile è impacchettato tramite il **packer UPX**, come ci viene confermato anche eseguendo il comando `file hw4.ex_`.

```
→ Homework 4 file hw4.ex_
hw4.ex_: PE32 executable (GUI) Intel 80386 (stripped to external PDB), for MS Windows, UPX compressed
```

PeiD infatti indica la presenza di due sezioni UPX0 e UPX1.

A screenshot of the 'Section Viewer' window. It contains a table with six columns: 'Name', 'V. Offset', 'V. Size', 'R. Offset', 'R. Size', and 'Flags'.

Name	V. Offset	V. Size	R. Offset	R. Size	Flags
UPX0	00001000	0000D000	00000200	00000000	E0000080
UPX1	0000E000	00095000	00000200	00094200	E0000040
.rsrc	000A3000	00001000	00094400	00000200	C0000040

Proviamo quindi ad effettuare l'unpacking tramite il comando `upx -d`. Si può vedere come l'eseguibile ottenuto ha una dimensione circa uguale a quello precedente, con un ratio di 98.92%. Molto probabilmente l'eseguibile non è stato spaccettato correttamente.

```
→ Homework 4 upx -d hw4.ex_
Ultimate Packer for eXecutables
Copyright (C) 1996 - 2020
UPX 3.96 Markus Oberhumer, Laszlo Molnar & John Reiser Jan 23rd 2020

-----
File size      Ratio      Format      Name
-----
614400 <-    607744    98.92%    win32/pe    hw4.ex_

Unpacked 1 file.
```

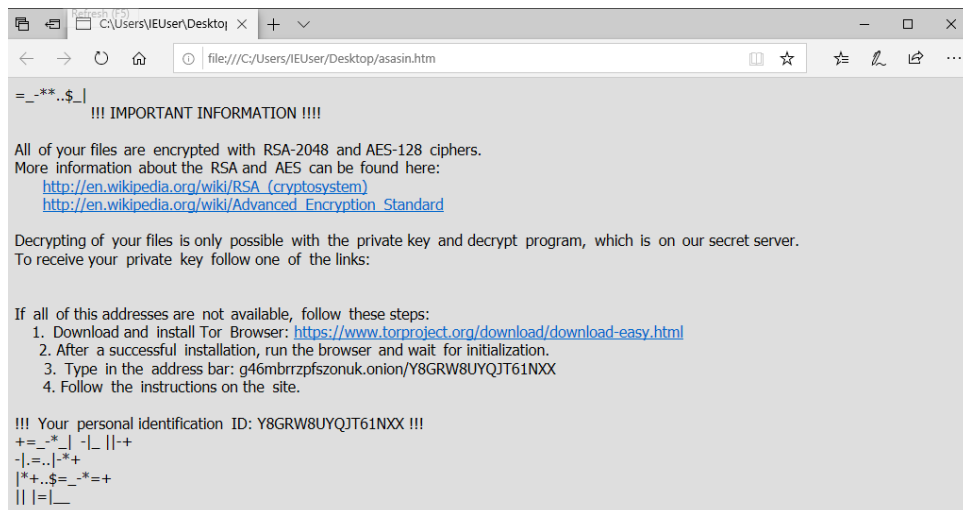
Infatti importando il nuovo eseguibile su *Ghidra* la situazione resta invariata rispetto a prima, e nessuna funzione risulta visibile in chiaro; risulta quindi necessario effettuare l'unpacking dell'eseguibile sfruttando altre tecniche.

2.2 Analisi Dinamica di Base

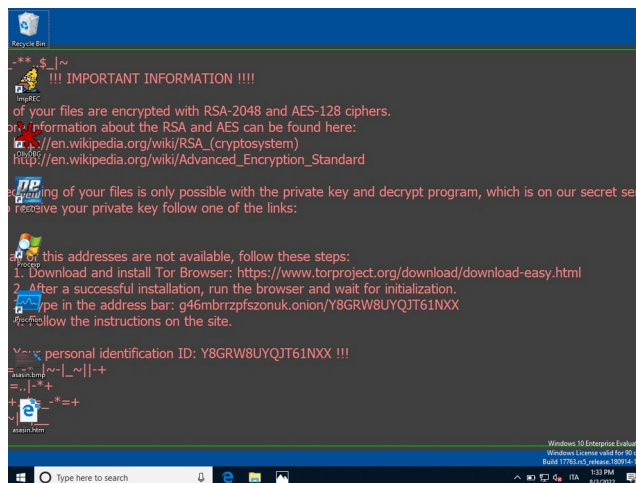
Prima di proseguire con l'unpacking dell'eseguibile effettuiamo l'*analisi dinamica di base*, in modo tale da ricavare il maggior numero di informazioni possibili sul comportamento del ransomware.

Per avere un'idea generale su cosa faccia effettivamente il malware sul sistema rinominiamo `hw4.ex_` in `hw4.exe` e **lanciamo l'eseguibile**.

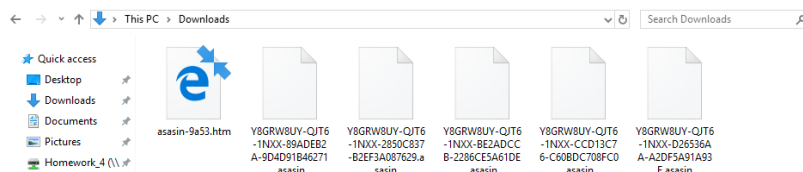
Dopo un certo tempo di attesa, vediamo che vengono aperte automaticamente una pagina web e un'immagine; entrambe riportano un messaggio, informando l'utente che tutti i suoi file sono stati **cifrati** tramite gli algoritmi di cifratura *RSA-2048* e *AES-128*. Viene inoltre indicato il procedimento necessario per ottenere la *secret key* usata per la cifratura e recuperare i file.



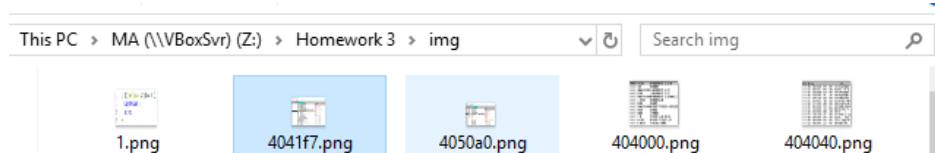
Possiamo inoltre vedere che viene modificato lo sfondo del *Desktop*, impostando la stessa immagine mostrata in precedenza.



Sul desktop sono stati generati i file **asasin.htm** e **asasin.bmp**. Accedendo alla cartella *Downloads*, possiamo verificare, come specificato nel messaggio, che i file sono stati effettivamente cifrati con estensione **".asasin"**. Viene generato un file **asasin-xxxx.htm** in ogni directory infettata.



Invece, i file presenti sul disco di rete **Z: //** non sono stati toccati in alcun modo.



Possiamo inoltre osservare che i file eseguibili, anche se sul disco **C: **, non vengono cifrati dal ransomware. Infatti documentandoci sul web vediamo che i ransomware Locky non cifrano tutti i file, ma solo quelli con specifiche estensioni.

```
.m4u, .m3u, .mid, .wma, .flv, .3g2, .mkv, .3gp, .mp4, .mov, .avi, .asf,
.mpeg, .vob, .mpg, .wmv, .fla, .swf, .wav, .mp3, .qcow2, .vdi, .vmdk,
.vmx, .gpg, .aes, .arc, .paq, .tar.bz2, .tbk, .bak, .tar, .tgz, .gz,
.7z, .rar, .zip, .djv, .djvu, .svg, .bmp, .png, .gif, .raw, .cgm, .jpeg,
.jpg, .tif, .tiff, .nef, .psd, .cmd, .bat, .sh, .class, .jar, .java,
.rb, .asp, .cs, .brd, .sch, .dch, .dip, .pl, .vbs, .vb, .js, .h, .asm,
.pas, .cpp, .c, .php, .ldf, .mdf, .ibd, .myi, .myd, .frm, .odb, .dbf,
.db, .mdb, .sql, .sqldbt, .sqlite3, .asc, .lay6, .lay, .ms11, .sldm, .sldx,
.ppsm, .ppsx, .ppam, .docb, .mml, .sxm, .otg, .odg, .uop, .potx, .potm,
.pptx, .pptm, .std, .sxd, .pot, .pps, .sti, .sxi, .otp, .odp, .wb2, 0.123,
.wks, .wk1, .xltx, .xltm, .xlsx, .xlsm, .xlsb, .slk, .xlw, .xlt, .xlm, .xlc,
.dif, .stc, .sxc, .ots, .ods, .hwp, .602, .dotm, .dotx, .docm, .docx, .dot,
.3dm, .max, .3ds, .xml, .txt, .csv, .uot, .rtf, .pdf, .XLS, .ppt, .stw, .sxw,
.ott, .odt, .doc, .pem, .p12, .csr, .crt, .key, wallet.dat
```

Anche i file in cartelle critiche del sistema come **Windows** e **Program Files** non sembrano essere attaccate in nessun modo, e tutti i software installati nel sistema mantengono la loro operatività.

Infine è possibile vedere che l'eseguibile **hw4.exe** precedentemente lanciato è stato automaticamente eliminato al termine dell'esecuzione del ransomware.

2.2.1 Process Monitor

Per avere un'idea più specifica di come il malware opera nel sistema analizziamo le informazioni raccolte tramite *Process Monitor*.

Il ransomware impiega circa *1 minuto* per eseguire tutte le sue azioni. Come previsto, il programma manipola ampiamente i registri di sistema, oltre a leggere e scrivere numerosi file.

Time of Day	Process Name	PID	Operation	Path	Result	Detail
7:07:11.2210045 ...	hw4.exe	6864	Process Start		SUCCESS	Parent PID: 3216, ...
7:07:11.2210087 ...	hw4.exe	6864	Thread Create		SUCCESS	Thread ID: 7368
7:07:11.2415243 ...	hw4.exe	6864	Load Image	\\BoxSvr\MA\Homework 4\hw4.exe	SUCCESS	Image Base: 0x400...
7:07:11.2419199 ...	hw4.exe	6864	Load Image	C:\Windows\System32\ntdll.dll	SUCCESS	Image Base: 0x7fb...
7:07:11.2422555 ...	hw4.exe	6864	Load Image	C:\Windows\SysWOW64\ntdll.dll	SUCCESS	Image Base: 0x77c...
7:07:11.2423680 ...	hw4.exe	6864	CreateFile	C:\Windows\Prefetch\HW4.EXE-7CE7...	NAME NOT FOUND	Desired Access: G...
7:07:11.2425395 ...	hw4.exe	6864	RegOpenKey	HKLM\System\CurrentControlSet\Contr...	REPARSE	Desired Access: Q...
7:07:11.2425615 ...	hw4.exe	6864	RegOpenKey	HKLM\System\CurrentControlSet\Contr...	SUCCESS	Desired Access: Q...
7:07:11.2425778 ...	hw4.exe	6864	RegQueryValue	HKLM\System\CurrentControlSet\Contr...	NAME NOT FOUND	Length: 80
7:07:11.2425868 ...	hw4.exe	6864	RegCloseKey	HKLM\System\CurrentControlSet\Contr...	SUCCESS	
7:07:11.2425971 ...	hw4.exe	6864	RegOpenKey	HKLM\SYSTEM\CurrentControlSet\Con...	REPARSE	Desired Access: Q...
7:07:11.2426041 ...	hw4.exe	6864	RegOpenKey	HKLM\System\CurrentControlSet\Contr...	NAME NOT FOUND	Desired Access: Q...
7:07:11.2426471 ...	hw4.exe	6864	RegOpenKey	HKLM\SYSTEM\CurrentControlSet\Con...	REPARSE	Desired Access: Q...
7:07:11.2426526 ...	hw4.exe	6864	RegOpenKey	HKLM\System\CurrentControlSet\Contr...	SUCCESS	Desired Access: Q...
7:07:11.2426582 ...	hw4.exe	6864	RegQueryValue	HKLM\System\CurrentControlSet\Contr...	NAME NOT FOUND	Length: 24
7:07:11.2426637 ...	hw4.exe	6864	RegCloseKey	HKLM\System\CurrentControlSet\Contr...	SUCCESS	
7:07:11.2429473 ...	hw4.exe	6864	CreateFile	C:\Windows	SUCCESS	Desired Access: E...

Alcune informazioni rilevanti riguardano la creazione di 2 thread oltre a quello principale, ed il frequente utilizzo di *QueryNameInformationFile* su varie cartelle e file del sistema.

5:20:2...	hw4.exe	3216	QueryNameInfo...	C:\Windows\System32\en-US\KemelB...	SUCCESS	Name: \Windows\...
5:20:2...	hw4.exe	3216	QueryNameInfo...	C:\Windows\System32	SUCCESS	Name: \Windows\...
5:20:2...	hw4.exe	3216	QueryNameInfo...	C:\Windows\System32\en-US\propsys...	SUCCESS	Name: \Windows\...
5:20:2...	hw4.exe	3216	QueryNameInfo...	C:\Windows\System32	SUCCESS	Name: \Windows\...
5:20:2...	hw4.exe	3216	QueryNameInfo...	C:\Windows\System32\en-US\ESNT...	SUCCESS	Name: \Windows\...
5:20:2...	hw4.exe	3216	QueryNameInfo...	C:\Users\IEUser\AppData\Local\Micro...	SUCCESS	Name: \Users\IEU...
5:20:2...	hw4.exe	3216	QueryNameInfo...	C:\Users\IEUser\Documents\ma-progra...	SUCCESS	Name: \Users\IEU...

Il processo termina chiamando la *ExitProcess*, e vediamo che poco prima viene chiamata la *CreateProcess* eseguendo un'istruzione da terminale per cancellare l'eseguibile. Quindi sappiamo che probabilmente l'eseguibile viene cancellato con questo comando alla fine dell'esecuzione del ransomware.

Event	Process	Stack
Date:	8/20/2022 5:21:00.9063384 AM	
Thread:	4200	
Class:	Process	
Operation:	Process Create	
Result:	SUCCESS	
Path:	C:\Windows\SysWOW64\cmd.exe	
Duration:	0.0000000	
PID:	6352	
Command line:	cmd.exe /C del /Q /F "Z:\Homeworks\Homework 4\hw4.exe"	

2.2.2 Informazioni Ricavate

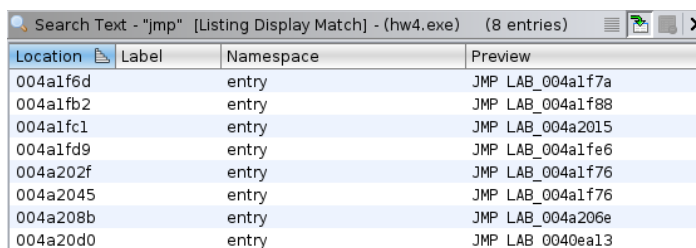
Riassumendo, tramite l'analisi dinamica sono state individuate le seguenti azioni che il ransomware effettua:

- Creazione dei file `asasin.htm` e `asasin.bmp` sul desktop.
- Apertura automatica di `asasin.htm` e `asasin.bmp`.
- Impostazione di `asasin.bmp` come sfondo del desktop.
- Cifratura dei dati utente tramite gli algoritmi *RSA-2048* e *AES-128*. Non vengono cifrati invece i file sul disco remoto `Z:\`.
- Creazione del file `asasin-xxxx.htm` nelle cartelle dove vengono cifrati file.
- Eliminazione del file `hw4.exe` lanciato, probabilmente tramite `cmd`.

3 Unpacking

Per effettuare l'unpacking dell'eseguibile è necessario individuare innanzitutto l'**Original Entry Point (OEP)**. A tale scopo cerchiamo innanzitutto istruzioni di *long jump*, che possono rappresentare un salto verso l'OEP.

Tramite lo strumento *Search Program Text* di *Ghidra* individuiamo tutte le occorrenze delle istruzioni di `jmp`. Vengono trovate 8 jump, ma i primi 7 risultano ad un indirizzo vicino e quindi non interessanti.



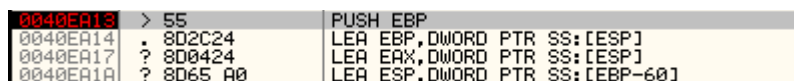
Location	Label	Namespace	Preview
004a1f6d		entry	JMP LAB_004a1f7a
004a1fb2		entry	JMP LAB_004a1f88
004a1fc1		entry	JMP LAB_004a2015
004a1fd9		entry	JMP LAB_004a1fe6
004a202f		entry	JMP LAB_004a1f76
004a2045		entry	JMP LAB_004a1f76
004a208b		entry	JMP LAB_004a206e
004a20d0		entry	JMP LAB_0040ea13

L'ultima istruzione invece è un *long jump* da 004a20d0 verso 0040ea13. Analizzando tale indirizzo da *Ghidra* vediamo che in quest'area sono presenti alcune istruzioni ASM (probabilmente codice dello stub), mentre se fosse realmente l'OEP l'area dovrebbe tipicamente essere vuota per essere successivamente riempita dallo stub con le istruzioni del programma originale.

```
LAB_0040ea13
0040ea13 54      PUSH     ESP=>param_11
0040ea14 9e      SAHF
0040ea15 16      PUSH     SS
0040ea16 78 0b   JS       LAB_0040ea22+1
0040ea18 1f      POP      DS
0040ea19 00 f0   ADD     AL,param_2
0040ea1b 00 3c 95 ADD     byte ptr [param_2*0x4 + 0x2f2a7],BH
a7 f2 02 00
```

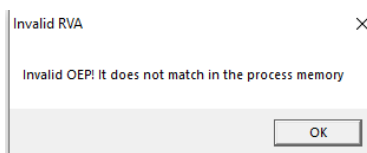
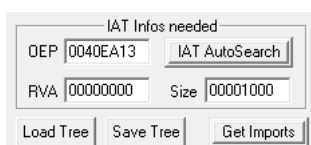
Ad ulteriore conferma di ciò, proviamo ad utilizzare **OllyDbg** e **ImpRec** per ricavare la **Import Address Table (IAT)**.

Da *OllyDbg* apriamo l'eseguibile, inseriamo un breakpoint all'indirizzo 0040ea13, ed seguiamo il programma; quando il flusso di esecuzione raggiunge il breakpoint procediamo con il memory dump tramite il plugin *OllyDump*. Se tale indirizzo fosse l'OEP, allora tramite debugger sarebbe stato eseguito tutto lo stub.



Address	Disassembly
0040EA13	> 55 PUSH EBP
0040EA14	. 8D2C24 LEA EBP,DWORD PTR SS:[ESP]
0040EA17	? 8D0424 LEA EAX,DWORD PTR SS:[ESP]
0040EA1A	? 8D65A0 LEA ESP,DWORD PTR SS:[EBP-60]

A questo punto utilizziamo *ImpRec* per cercare di ricavare la IAT. Avviamo `hw4.exe`, inseriamo 0040EA13 come OEP e sfruttiamo *IAT Autosearch*. Viene ritornato un messaggio di errore secondo cui 0040ea13 è un OEP non valido.



3.1 Unpacking manuale con OllyDbg

Nessuna delle tecniche automatiche utilizzate ha prodotto un eseguibile spaccettato. Analizzando l'eseguibile dumped prodotto, vediamo che risulta ancora compresso con UPX, ma provando nuovamente a spaccettarlo ci viene detto che non è in realtà impacchettato.

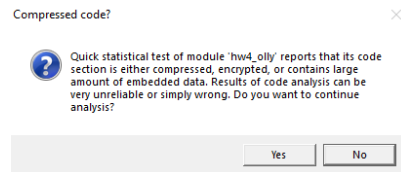
```
→ Homework 4 file dump2.exe
dump2.exe: PE32 executable (GUI) Intel 80386 (stripped to external PDB), for MS Windows, UPX compressed
→ Homework 4 upx -d dump2.exe

Ultimate Packer for eXecutables
Copyright (C) 1996 - 2020
UPX 3.96 Markus Oberhumer, Laszlo Molnar & John Reiser Jan 23rd 2020

File size      Ratio      Format      Name
-----
upx: dump2.exe: NotPackedException: not packed by UPX

Unpacked 0 files.
```

Ritorniamo quindi su *OllyDbg* per analizzare il nuovo `dump2.exe`. Aprendo l'eseguibile ci viene mostrato un messaggio secondo cui il programma presenta una *code section* compressa.



Proseguiamo comunque con l'analisi, partendo dall'indirizzo `0040EA13`, ed eseguendo singolarmente le singole istruzioni macchina. All'indirizzo `0040ea42` viene invocata l'API `IsBadStringPtrW`, che prende in input un puntatore alla stringa ASCII `"lwpqabklbeiutlcti"` (puntata dal registro `ESI`), e verifica se il processo ha permessi di lettura per accedere a quell'area.

			Registers (FPU)
0040EA13	55	PUSH EBP	EAX 0049405C hw4_dump.0049405C
0040EA14	802C24	LEA EBP,DWORD PTR SS:[ESP]	ECX 00000000
0040EA17	80424	LEA EAX,DWORD PTR SS:[ESP]	EDX 0040EA13 hw4_dump.<ModuleEntryPoint>
0040EA1A	8065 00	LEA ESP,DWORD PTR SS:[EBP-60]	EBX 00357000
0040EA1D	8000 8B000000	LEA ECX,DWORD PTR DS:[EBI]	ESP 0019FF00
0040EA23	66:30 FEFD	CMPL AX,0F0F	EBP 0019FF70
0040EA27	72 E7	JB SHORT hw4_dump.0040EA10	ESI 00496A1A ASCII "lwpqabklbeiutlcti"
0040EA29	83EC 04	SUB ESP,4	EDI 0040EA13 hw4_dump.<ModuleEntryPoint>
0040EA2C	C60424 11	MOV BYTE PTR SS:[ESP],11	EIP 0040EA42 hw4_dump.0040EA42
0040EA30	8035 3B4DAE98	LEA ESI,DWORD PTR DS:[98AE4D3B]	C 0 ES 002B 32bit 0(FFFFFFFF)
0040EA36	81EE 21E36498	SUB ESI,9864E321	P 0 CS 0023 32bit 0(FFFFFFFF)
0040EA3C	56	PUSH ESI	
0040EA3D	B8 5C404900	MOV EAX,hw4_dump.0049405C	
0040EA42	FF10	CALL DWORD PTR DS:[EAX]	

In particolare `IsBadStringPtrW` viene invocata più volte, verificando ogni volta che questa ritorni zero (quindi che il processo abbia i permessi adatti).

0040EA3D	B8 5C404900	MOV EAX,<hw4_dump.IsBadStringPtrW>
0040EA42	FF10	CALL DWORD PTR DS:[EAX]
0040EA44	83F8 00	CMPL EAX,0
0040EA47	0F85 FC77FFFF	JNZ hw4_dump.00406249
0040EA4D	83EC 04	SUB ESP,4
0040EA50	C60424 11	MOV BYTE PTR SS:[ESP],11
0040EA54	800D 3B4DAE98	LEA ECX,DWORD PTR DS:[98AE4D3B]
0040EA5A	81E9 21E36498	SUB ECX,9864E321
0040EA60	51	PUSH ECX
0040EA61	B8 5C404900	MOV EAX,<hw4_dump.IsBadStringPtrW>
0040EA66	FF10	CALL DWORD PTR DS:[EAX]
0040EA68	85C0	TEST EAX,EAX
0040EA6A	0F85 D977FFFF	JNZ hw4_dump.00406249
0040EA70	83EC 04	SUB ESP,4
0040EA73	C60424 11	MOV BYTE PTR SS:[ESP],11
0040EA77	801D 3B4DAE98	LEA EBX,DWORD PTR DS:[98AE4D3B]
0040EA7D	81EB 21E36498	SUB EBX,9864E321
0040EA83	53	PUSH EBX
0040EA84	B8 5C404900	MOV EAX,<hw4_dump.IsBadStringPtrW>
0040EA89	FF10	CALL DWORD PTR DS:[EAX]
0040EA8B	85C0	TEST EAX,EAX
0040EA8D	0F85 B677FFFF	JNZ hw4_dump.00406249

Procedendo con l'analisi, osserviamo all'indirizzo 0040EADD una CALL 004072D1. Tramite *step into* passiamo alla funzione chiamata.

0040EAD1	.-0F85 7277FFFF	JNZ dump_2.00406249
0040EAD7	. 8D3D 01724000	LEA EDI,DWORD PTR DS:[4072D1]
0040EAD9	. E8 EF87FFFF	CALL dump_2.004072D1
0040EAE2	. 0000	ADD BYTE PTR DS:[EAX],AL
0040EAE4	. 0000	ADD BYTE PTR DS:[EAX],AL

Qui vengono eseguite ulteriori chiamate a IsBadStringPtrW, e viene caricato sullo stack il puntatore alla funzione di libreria VirtualAlloc.

004074B3	FF10	CALL DWORD PTR DS:[EAX]	
004074B5	A3 A0694900	MOV DWORD PTR DS:[4969A0],EAX	KERNEL32.VirtualAlloc
004074BA	3905 A0694900	CMP DWORD PTR DS:[4969A0],EAX	
004074C0	75 1C	JNZ SHORT dump_2.004074DE	
004074C2	75 0A	JNZ SHORT dump_2.004074CE	
004074C4	8105 A4694900	ADD DWORD PTR DS:[4969A4],BA9815DD	
004074CE	8D05 7B53A845	LEA EAX,DWORD PTR DS:[45A8537B]	

Successivamente si effettua un jump all'indirizzo 00406958, puntato da DS[4969A4].

004074D6	2905 A4694900	SUB DWORD PTR DS:[4969A4],EAX	
004074DC	75 00	JMP SHORT dump_2.004074DE	
004074DE	FF25 A4694900	JMP DWORD PTR DS:[4969A4]	dump_2.00406958
004074E4	0000	ADD BYTE PTR DS:[EAX],AL	

Qui si effettua la CALL a VirtualAlloc, passando i parametri tramite lo stack.

0040696E	C74424 FC 400000	MOV DWORD PTR SS:[ESP-4],40	
00406976	83C4 FC	ADD ESP,-4	
00406979	C74424 FC 001000	MOV DWORD PTR SS:[ESP-4],1000	
00406981	83C4 FC	ADD ESP,-4	
00406984	C74424 FC 880600	MOV DWORD PTR SS:[ESP-4],688	
0040698C	83C4 FC	ADD ESP,-4	
0040698F	C74424 FC 000000	MOV DWORD PTR SS:[ESP-4],0	
00406997	83C4 FC	ADD ESP,-4	
0040699A	E8 46000000	CALL dump_2.004069E5	JMP to KERNEL32.VirtualAlloc

In particolare i parametri passati sono i seguenti.

0019FEF4	00496A0C	ASCII "VirtualAlloc"
0019FEF8	00000000	Address = NULL
0019FEFC	00000688	Size = 688 (1672.)
0019FF00	00001000	AllocationType = MEM_COMMIT
0019FF04	00000040	Protect = PAGE_EXECUTE_READWRITE

Viene quindi allocata una nuova area di memoria di *688 bytes* con permessi di esecuzione, lettura e scrittura. Il parametro **address** viene passato NULL, e dalla documentazione sappiamo che in questo modo viene delegata al sistema la scelta della regione da allocare. L'indirizzo base della regione allocata (00030000¹), viene ritornato dalla funzione in EAX.

Registers (FPU)	
EAX	00030000
ECX	7A840000
EDX	00030000
EBX	00496A1A ASCII "lwpgabklbeutloti"
ESP	0019FF08
EBP	0019FF70
ESI	66C20560
EDI	00000688

Successivamente vengono eseguite una serie di istruzioni per scrivere sulla nuova area allocata, il cui puntatore viene copiato sul registro EBX, e all'indirizzo 004069DA si effettua la CALL EBX, accedendo quindi al nuovo blocco di istruzioni appena scritto in 00030000.

004069D1	5B	POP EBX	
004069D2	8D35 A0404900	LEA ESI,DWORD PTR DS:[4940A0]	
004069D8	FF36	PUSH DWORD PTR DS:[ESI]	
004069DA	FFD3	CALL EBX	

Registers (FPU)	
EAX	046A241D
ECX	00496688 dump2.00496688
EDX	00030000
EBX	00030000
ESP	0019FF08
EBP	0019FF70
ESI	004940A0 dump2.004940A0
EDI	00000000

¹ Su esecuzioni differenti l'indirizzo dell'area allocata potrebbe variare in termini di 4° e 5° byte (da destra).

All'indirizzo 0003001D inizia un ciclo con le seguenti istruzioni

0003001D	AD	LODS DWORD PTR DS:[ESI]	loop start
0003001E	E8 C020000	CALL 000302EF	
00030023	8946 FC	MOV DWORD PTR DS:[ESI-4],EAX	
00030026	E2 F5	LOOPD SHORT 0003001D	loop end

Viene effettuata una CALL 000302EF; analizzando il contenuto dei registri vediamo che questa ritorna in EAX un puntatore ad una funzione della libreria Kernel32.dll.

Successivamente il puntatore a tale funzione viene copiato tramite MOV nell'indirizzo puntato da DS:[ESI-4]. Eseguendo le diverse iterazioni del ciclo si può vedere che viene caricata ogni volta una libreria differente.

Registers (FPU)	Registers (FPU)	Registers (FPU)
EAX 75891430 KERNEL32.GetModuleHandleA	EAX 75890300 KERNEL32.GetProcAddress	EAX 75890970 KERNEL32.VirtualProtect
ECX 00000010	ECX 0000000F	ECX 0000000E
EDX 00580000	EDX 00580000	EDX 00580000
EBX 75870000 KERNEL32.75870000	EBX 75870000 KERNEL32.75870000	EBX 75870000 KERNEL32.75870000
ESP 0019FF00	ESP 0019FF00	ESP 0019FF00
EBP 00030608	EBP 00030608	EBP 00030608
ESI 00030610	ESI 00030614	ESI 00030618
EDI 00000000	EDI 00000000	EDI 00000000

In questo ciclo vengono quindi caricate alcune librerie utilizzate dall'eseguibile originale, ed è quindi probabilmente la parte dello stub in cui vengono **risolti gli import**.

A 00030065 viene eseguita una nuova VirtualAlloc. Vengono allocati indirizzi a partire da 00580000 ed il puntatore a tale area viene spostato su EDI.

00030051	FF55 08	CALL DWORD PTR SS:[EBP+8]	
00030054	8946 08	MOV DWORD PTR DS:[ESI+8],EAX	
00030057	6A 40	PUSH 40	
00030059	68 00100000	PUSH 1000	
0003005E	68 30600000	PUSH 680	
00030063	6A 00	PUSH 0	
00030065	FF55 10	CALL DWORD PTR SS:[EBP+10]	KERNEL32.VirtualAlloc

Registers (FPU)
EAX 00580000
ECX A5D80000
EDX 00580000
EBX 75870000 KERNEL32.75870000
ESP 0019FF00
EBP 00030608
ESI 0003064C
EDI 000306F5 ASCII "GetTickCount"

Tramite RETN si jumpa alla nuova area allocata.

0003007B	F3:A4	REP MOVSB BYTE PTR ES:[EDI],BYTE PTR DS:
0003007D	C3	RETN
0003007E	E8 82050000	CALL 00030605
00030083	5D	POP EBP
Return to 0058007E		

In particolare qui all'indirizzo 005800DC c'è un PUSHAD, vediamo che inizia un ciclo in cui vengono scritte diverse sezioni del programma nel registro EDI.

005800C7	8BCE	MOV ECX,ESI	
005800C9	0349 3C	ADD ECX,DWORD PTR DS:[ECX+3C]	
005800CC	8D79 18	LEA EDI,DWORD PTR DS:[ECX+18]	
005800CF	8B57 20	MOV EDI,DWORD PTR DS:[EDI+20]	
005800D2	0FB741 14	MOVZX EAX,WORD PTR DS:[ECX+14]	
005800D6	03F8	ADD EDI,EAX	
005800D9	0FB749 06	MOVZX ECX,WORD PTR DS:[ECX+6]	
005800DD	60	PUSHAD	
005800E0	8B47 08	MOV EAX,DWORD PTR DS:[EDI+8]	
005800E3	85C0	TEST EAX,EAX	
005800E2	74 42	JE SHORT 00580126	

Registers (FPU)
EAX 00001000
ECX 00000001
EDX 00001000
EBX 00400000 dump2.00400000
ESP 0019FEE0
EBP 00580608
ESI 006A0000
EDI 006A0208 ASCII ".pdata"
EIP 005800E4

Nell'ordine vengono copiate in EDI le stringhe ".text", ".rdata", ".data", ".reloc", ".pdata", che indicano le sezioni dell'address space dell'eseguibile *non-packed*;

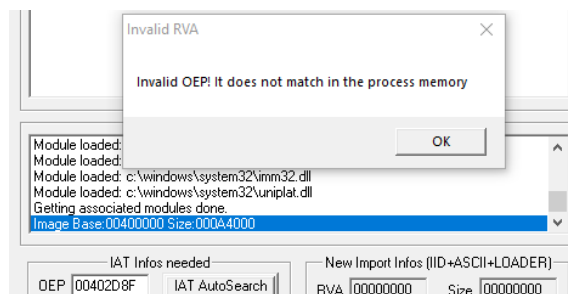
Invece all'indirizzo 005801BE viene eseguita una JMP ESI che porta all'indirizzo 00402D8F.

005801BE	-FFE6	JMP ESI	dump2.00402D8F
005801C0	60	PUSHAD	

Questo è molto probabilmente il *tail jump* cercato. Infatti analizzando da Ghidra l'eseguibile originale, l'area a tale indirizzo risulta vuota (a differenza del primo OEP individuato).

00402d8f	??	??
00402d90	??	??
00402d91	??	??

Eseguiamo quindi il *tail jump* tramite *step into* ed effettuiamo il dump tramite *OllyDump*. Testiamo il funzionamento con entrambe le euristiche offerte dal plugin, generando due eseguibili *dump_f1.exe* e *dump_f2.exe*. Proviamo a ricostruire la IAT tramite *ImpREC*, ma anche in questo caso non è stato riconosciuto l'OEP individuato.



Aperto i due eseguibili con *PeStudio* osserviamo che le due euristiche hanno comunque risolto alcuni import, ma sono state riconosciute librerie differenti.

Nel dump prodotto con il primo metodo:

library (5)	flag (2)	type (1)	functions (123)	description
kernel32.dll	-	implicit	112	Windows NT BASE API Client DLL
mpr.dll	x	implicit	4	Multiple Provider Router DLL
dsrole.dll	-	implicit	2	n/a
combase.dll	-	implicit	4	n/a
urlmon.dll	x	implicit	1	OLE32 Extensions for Win32

Nel dump prodotto con il secondo metodo:

library (4)	flag (1)	type (1)	functions (161)	description
advapi32.dll	-	implicit	33	Advanced Windows 32 Base API
gdi32.dll	-	implicit	12	GDI Client DLL
kernel32.dll	-	implicit	112	Windows NT BASE API Client DLL
mpr.dll	x	implicit	4	Multiple Provider Router DLL

Nessuno dei due metodi è riuscito dunque a ripristinare la IAT, e questo ci viene confermato provando a lanciare i due eseguibili, che vanno entrambi in errore.



Non si è riusciti quindi ad ottenere un programma spaccettato correttamente eseguibile, in quanto non è stata ripristinata la IAT nella sua interezza.

Tuttavia importando i due nuovi eseguibili su *Ghidra*, si è verificato che sono state riconosciute molte funzioni di libreria, e che nel nuovo entry point è presente effettivamente del codice macchina, quindi probabilmente in entrambi i dump siamo riusciti comunque a ripristinare il codice dell'eseguibile originale.

Per questo si è stabilito in definitiva che l'OEP corrisponde a 00402D8F, terminando così la fase di unpacking dell'eseguibile.

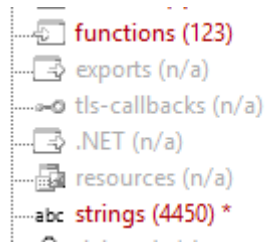
4 Analisi Statica di Base

Abbiamo ora a disposizione un eseguibile dumped che, seppur non direttamente eseguibile, ci permette di visualizzare tramite *Ghidra* il **codice decompilato**, potendo quindi analizzare con maggiore semplicità le istruzioni eseguite del ransomware.

Aprendo con *OllyDbg* il programma generato da *OllyDump* si hanno errori, per cui la strategia adottata è quella di usare **hw4.exe** per il debugger, inserendo un breakpoint al **main** individuato e proseguendo da qui l'analisi a runtime. In parallelo viene poi utilizzato il decompilato fornito da *Ghidra*, utilizzando in questo caso **dump_f1.exe**.

A differenza di quanto fatto nell'*analisi preliminare*, adesso si ha a disposizione un eseguibile con il codice originale in chiaro, per cui è possibile procedere con l'effettiva **analisi statica di base** del programma.

Tramite *Defined Strings* vengono ora riconosciute *577 stringhe* relative ai dll importati, messaggi di errore, e funzioni specifiche utilizzate dal programma. Anche usando **PEStudio** vediamo che vengono identificate ben *123* funzioni e *4450* stringhe.



Tra le API più interessanti troviamo:

- **CreateThread / TerminateThread**, quindi vengono spawnati e terminati alcuni thread dal ransomware, come visto tramite *Process Monitor*.
- **VirtualAlloc / VirtualFree**, che come già visto permettono di allocare e deallocare varie aree di memoria.
- **GetTickCount / IsDebuggerPresent / QueryPerformanceCounter**, suggeriscono la possibile presenza di meccanismi di antidebug.
- **MoveFileExW / DeleteFileW / CreateFileW / WriteFile**, che vengono utilizzate probabilmente per creare **asasin.htm** e **asasin.bmp** e manipolare file in generale.
- **CryptEncrypt / CryptDestroyKey / CryptGenRandom / CryptHashData**, per generare la chiave e cifrare i file dell'utente.
- **RegDeleteValueA / RegSetValueExW / RegSetValueExA / RegOpenKeyExA**, per manipolare i registri di sistema.
- **LookupPrivilegesA** forse utilizzata per modificare i privilegi del ransomware ed eseguire operazioni altrimenti non consentite.
- **CreateEvent / SetEvent**, per la creazione e segnalazione di eventi.

5.2 FUN_00477050 (alloc_anti_dbg)

La prima funzione invocata dal `main` è `FUN_00477050`. Un'analisi a grana grande del decompilato mostra come venga eseguite una `VirtualAlloc`, `GetProcAddress`, e poi una `memcpy` nella nuova area di memoria. Infine viene liberata l'area allocata con `VirtualFree`.

```

43 if (((local_10 == (ushort**)0x0) || (param_1 < dwSize)) ||
44     (uVar2 < (uint)((int)local_10 + (int)dwSize))) goto LAB_00477a0a;
45 puVar5 = (ushort *)VirtualAlloc((LPVOID)0x0, (SIZE_T)dwSize, 0x3000, 4);
46 puVar7 = (uint *)0x0;
60 hModule = GetModuleHandleA((LPCSTR)&local_30);
61 _Src = GetProcAddress(hModule, (LPCSTR)lpProcName);
62 FID_conflict_memcpy(puVar5, _Src, (size_t)dwSize);
63 return puVar5;
79 if (dwSize != param_1) {
80     param_2 = (ushort *)VirtualAlloc((LPVOID)0x0, (SIZE_T)param_1, 0x3000, 4);
81     if (param_2 != (ushort *)0x0) {
82         FUN_0046e870();
83         iVar4 = FUN_0046e940((undefined *)param_2, (int *)&param_1, puVar5);
84         if (iVar4 == 0) {
85             VirtualFree(puVar5, 0, 0x8000);
86             return param_2;
87         }
88         VirtualFree(param_2, 0, 0x8000);
89     }
90     VirtualFree(puVar5, 0, 0x8000);

```

Tuttavia risulta complesso comprendere il comportamento di tale funzione solo tramite decompilatore, per cui procediamo con un'analisi più dettagliata sfruttando *OllyDbg*.

Posizionandoci sull'indirizzo 00477050 corrispondente al `main` vediamo che non vengono mostrate istruzioni. Questo perché le istruzioni del `main` vengono caricate a runtime, mentre noi stiamo utilizzando l'eseguibile impacchettato.

0042C820	6A	DB 6A	Main FUN_0042C820
0042C821	00	DB 00	
0042C822	90	DB 90	
0042C823	EB	DB EB	
0042C824	74	DB 74	CHAR 'z'
0042C825	8D	DB 8D	
0042C826	36	DB 36	CHAR '6'
0042C827	EB	DB EB	
0042C828	60	DB 60	CHAR ' '
0042C829	90	DB 90	
0042C82A	E8	DB E8	
0042C82B	21	DB 21	CHAR 't'
0042C82C	A8	DB A8	
0042C82D	04	DB 04	
0042C82E	00	DB 00	
0042C82F	EB	DB EB	

Andiamo quindi a forzare l'analisi del codice tramite l'opzione 'Analyse Code'.

Analysis	>	Analyse code	Ctrl+A
Bookmark	>	Remove analysis from module	
Dump debugged process		Scan object files	Ctrl+O
Appearance	>	Remove object scan from module	
		Remove analysis from selection	BkSp
		During next analysis, treat selection as	>

Come spiegato in precedenza l'analisi tramite debugger parte dal `main` (0042C820). Qui viene eseguita la `GetModuleHandleA`, che ritorna in `EAX` l'handle all'eseguibile.

0042C89F	> 90	NOP	Registers (FPU)
0042C8A0	. FF15 18A24700	CALL DWORD PTR DS:[47A218]	EAX 00400000 hw4.00400000

Vengono poi eseguiti una serie di `JMP` in diverse locazioni di memoria, fino a giungere in 0042C82A dove avviene la call alla `FUN_00477050` che vogliamo analizzare.

0042C829	> 90	NOP
0042C82A	. E8 21A80400	CALL hw4.00477050

Procediamo tramite *step into*. Come visto dal decompilato, è presente un `while(true)` all'indirizzo 00477722 dove vengono configurati alcuni parametri locali della funzione. Superiamo questo ciclo spostandoci all'istruzione successiva 0047729A tramite *run to selection* (F4).

All'indirizzo 00477793 viene chiamata la `VirtualAlloc`, che alloca una nuova zona di memoria scelta dal sistema operativo, che parte dagli indirizzi 00590000.

Proseguendo all'indirizzo 00477573 si ha un accesso al segmento FS con offset 18h, accedendo quindi al **Thread Environment Block** (TEB). Il puntatore alla TEB viene copiato in EAX tramite una MOV.

Bytes/ Type	offset (32-bit, FS)	offset (64-bit, GS)	Windows Versions	Description
pointer	FS:[0x00]	GS:[0x00]	Win9x and NT	Current Structured Exception Handling (SEH) frame Note: the 64-bit version of Windows uses stack unwinding done in kernel mode instead.
pointer	FS:[0x04]	GS:[0x08]	Win9x and NT	Stack Base / Bottom of stack (high address)
pointer	FS:[0x08]	GS:[0x10]	Win9x and NT	Stack Limit / Ceiling of stack (low address)
pointer	FS:[0x0C]	GS:[0x18]	NT	SubSystemTib
pointer	FS:[0x10]	GS:[0x20]	NT	Fiber data
pointer	FS:[0x14]	GS:[0x28]	Win9x and NT	Arbitrary data slot
pointer	FS:[0x18]	GS:[0x30]	Win9x and NT	Linear address of TEB

```
00477573 > 64:A1 18000000 MOV EAX,DWORD PTR FS:[18]
00477579 . 90 NOP
0047757A .^E9 09FEFFFF JMP hw4.00477388
```

Da questo punto le varie istruzioni macchina vengono intervallate da una serie di NOP e JMP, che verranno omessi (ma comunque eseguiti) per alleggerire la trattazione.

All'indirizzo 00477388 viene effettuata una MOV in EAX, spiazzandosi di 0x30 da EAX stesso. Si accede quindi all'offset 0x30 della TEB, dove è mantenuto il puntatore al **Process Environment Block** (PEB). Tale puntatore viene messo proprio in EAX.

Name	Offset
DWORD EnvironmentPointer	+1C
DWORD ProcessId	+20
DWORD threadId	+24
DWORD ActiveRpcInfo	+28
DWORD ThreadLocalStoragePointer	+2C
PEB* Peb	+30

```
00477388 > 8B40 30 MOV EAX,DWORD PTR DS:[EAX+30]
0047738B . 90 NOP
0047738C .^E9 A9020000 JMP hw4.0047763A
```

Proseguendo, all'indirizzo 004774BE si accede all'offset 2 della PEB ovvero al campo `BeingDebugged`. Il valore assunto da tale campo viene copiato in EAX tramite una MOVZX.

```
004774BE . 0FB640 02 MOVZX EAX,BYTE PTR DS:[EAX+2]
004774C2 .^E9 9B050000 JMP hw4.00477A62
004774C7 > 53 PUSH EBX
004774C8 . 90 NOP
004774C9 .^E9 89FBFFFF JMP hw4.00477057
```


Questo ci suggerisce la probabile presenza di un meccanismo di *Anti Debug*. Ciò viene confermato immediatamente osservando il codice disassemblato in corrispondenza di questo indirizzo. Vediamo infatti che il valore di `BeingDebugged` viene salvato nella variabile `local_8` (rinominata `being_dbg`), e successivamente si controlla proprio il valore assunto da tale variabile.

```

50  if (being_dbg != 0) {
51      local_30 = 0x6e72656b;
52      local_2c = 0x32336c65;
53      local_28 = 0x6c642e;
54      local_24 = 0;
55      local_20 = 0x6f6c641;
56      local_1c = 0x6e6f4363;
57      local_18 = 0x656c6f73;
58      local_14 = 0;
59      lpProcName = &local_20;
60      hModule = GetModuleHandleA((LPCSTR)&local_30);
61      .Src = GetProcAddress(hModule, (LPCSTR)lpProcName);
62      FTD_conflict; memcpy(allocated_1, .Src, (size_t)dwSize);
63      return allocated_1;
64  }

```

Se il processo è eseguito tramite debugger, `being_dbg` varrà 1, si entra nell'`if`, ed a seguito della `memcpy` viene ritornato al `main` l'area di memoria allocata; nel `main` il processo non termina in quanto l'area ritornata è comunque non nulla, per cui verrà comunque invocata la seconda funzione.

```

9  DAT_004831f8 = FUN_00477050((uint *)pHVar1,
10                               *(ushort **)((int)&pHVar1[0x14].unused + pHVar1[0xf].unused));
11  if (DAT_004831f8 == (ushort *)0x0) {
12      uExitCode = 0xffffffff;
13  }
14  else {
15      uExitCode = FUN_00429ea0();
16  }
17  /* WARNING: Subroutine does not return */
18  ExitProcess(uExitCode);

```

Tuttavia l'utilizzo di un debugger varia sicuramente il flusso di esecuzione del malware, per cui andiamo a patchare l'eseguibile; l'idea è quella di porre in `EAX` il valore 0 invece di copiare il valore 1 di `BeingDebugged`, e a tale scopo inseriamo l'istruzione `XOR EAX,EAX` all'indirizzo `004774BE`.

<pre> 004774BE . 0FB640 02 MOVZX EAX, BYTE PTR DS:[EAX+2] 004774C2 .vE9 9B050000 JMP hw4.00477A62 </pre>	→	<pre> 004774BE 33C0 XOR EAX,EAX 004774C0 90 NOP 004774C1 90 NOP 004774C2 .vE9 9B050000 JMP hw4.00477A62 </pre>
---	---	---

Aiutandoci nuovamente con *Ghidra*, vediamo che successivamente viene eseguito un ciclo *do-while* dove vengono settate alcune variabili locali; ricaviamo l'indirizzo dell'`if()` appena fuori dal ciclo (`00477747`) e nuovamente tramite *run to selection* andiamo a bypassarlo muovendoci direttamente su quella zona di memoria.

```

68  do {
69      bVar5 = (byte)puVar6 & 0xf;
70      param_2 = (ushort *)
71          (((int)param_2 << bVar5 | (uint)param_2 >> 0x20 - bVar5) +
72           ((uint)param_2 >> 3 | (int)param_2 << 0x1d) ^
73           ((uint)puVar6 >> 0xb | (int)puVar6 << 0x15) + 0x72462828);
74      pbVar3 = (byte *)((int)puVar6 + (int)allocated_1);
75      puVar6 = (uint *)((int)puVar6 + 1);
76      *pbVar3 = (byte *)((int)local_10 + (int)pbVar3) ^ (byte)param_2;
77  } while (puVar6 < dwSize);
78  }
79  if (dwSize != param_1) {

```

		LAB_00477747	
00477747	90	NOP	
00477748	0f 84 b3	JZ	LAB_00477801
	00 00 00		
0047774e	90	NOP	
0047774f	e9 98 fb	JMP	LAB_004772ec
	ff ff		

Dentro l'`if` tramite `CALL EDI` viene nuovamente eseguita una `VirtualAlloc`. L'area di memoria allocata parte dagli indirizzi `005A0000`; tale indirizzo viene scritto in `DS`: `[004831f8]`.

Registers (FPU)	
EAX	005A0000
ECX	03060000
EDX	005A0000
EBX	00590000
ESP	0019FE9C
EBP	0019FED4
ESI	00000B07
EDI	757BFC60 KERNEL32.VirtualAlloc

Address	Hex dump	ASCII
004831F8	00 00 5A 00 00 00 00 00	..Z....
00483200	00 00 00 00 00 00 00 00
00483208	00 00 00 00 00 00 00 00
00483210	00 00 00 00 00 00 00 00

Proseguendo all'indirizzo 004772BF si effettua la CALL ad una funzione FUN_0046E940. Da Ghidra vediamo che qui vengono effettuate alcune operazioni sulla prima area di memoria allocata (indirizzo 00590000, indicata con `allocated_1`), per poi ritornare al chiamante.

```
iVar4 = FUN_0046E940((undefined *)param_2, (int *)&param_1, allocated_1);
```

Dal debugger quindi seguiamo oltre con *step over* senza analizzare nel dettaglio. Infine, viene invocata la `VirtualFree` all'indirizzo 0047767F per liberare la prima area di memoria allocata, e poi si restituisce il controllo al `main`.

```
0047767F . FF15 94A14700 CALL DWORD PTR DS:[47A194] VirtualFree
00477685 . vE9 FE020000 JMP hw4.00477988
```

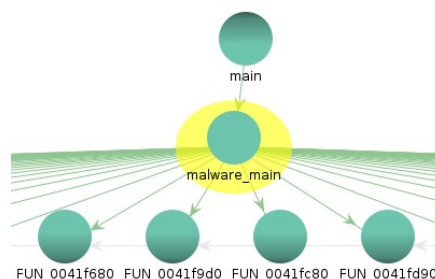
In definitiva la funzione FUN_00477050, oltre ad allocare e lavorare su alcune aree di memoria, implementa un semplice meccanismo di *Anti Debug*, che è stato bypassato con successo (o almeno sembra). Rinominiamo quindi per semplicità tale funzione su *Ghidra* come `alloc_anti_dbg`.

Inoltre tornando rapidamente al `main`, rinominiamo anche la variabile globale DAT_004831F8 in `allocated_mem`, in quanto contiene il valore di ritorno di `alloc_anti_dbg`, e quindi l'indirizzo dell'area di memoria allocata tramite `VirtualAlloc`.

5.3 FUN_00429EA0 (malware_main)

A questo punto possiamo proseguire con l'analisi dell'altra funzione invocata dal `main`, ovvero FUN_00429EA0. In `alloc_anti_dbg` non risultava essere presente nessuna delle vere funzionalità tra quelle identificate nel malware, per cui ci aspettiamo che la vera implementazione del ransomware inizi a partire da FUN_00429EA0. Visualizzando il *function call graph*, vediamo in effetti che a partire da questa vengono invocate altre 55 funzioni, sia di libreria che *user-defined*.

Presumibilmente dovremo analizzare molte funzioni del tipo FUN_004xxxxx, quindi per semplicità rinominiamo FUN_00429EA0 in `malware_main`.



Tornando su *OllyDbg*, eseguiamo *step into* all'indirizzo 0042C861 dove viene chiamata `malware_main`, ed iniziamo con l'effettiva fase di analisi.

```
0042C861 . E8 3AD6FFFF CALL <hw4.malware_main>
0042C866 . vEB 26      JMP SHORT hw4.0042C88E
0042C868 . > 90        NOP
```

```
else {
    uExitCode = malware_main();
}
/* WARNING: Subroutine does not return */
ExitProcess(uExitCode);
}
```

Il CMP a 0042AF16 verifica se l'indirizzo puntato da DS:[004831F8] è pari a 0 e in quel caso ritorna al chiamante. Come abbiamo già visto alloc_anti_dbg scriveva l'indirizzo 005A0000 dell'area allocata in 004831F8, quindi il programma prosegue.

```

0042AF16 . 391D F8314800 CMP DWORD PTR DS:[4831F8],EBX
0042AF1C . 90 NOP
0042AF1D . ^E9 52F6FFFF JMP hw4.0042A574
EBX=00000000
DS:[004831F8]=005A0000

```

```

if (allocated_mem == 0) {
    *in_FS_OFFSET = *(undefined4 *) (unaff_EBP + -0xc);
    return 0xffffffff;
}

```

Dopo una serie di JMP e NOP, all'indirizzo 0042C17B viene invocata l'API SetLastError, passando come parametro i seguenti flag.

```

0019FCC8 005C0065
0019FCCC 00008003 SetLastError = SEM_FAILCRITICALERRORS;SEM_NOGPFAULTERRORBOX;SEM_NOOPENFILEERRORBOX

```

Dalla documentazione vediamo che questi permettono di mascherare all'utente eventuali errori. In particolare:

- SEM_FAILCRITICALERRORS: Il sistema non mostra il *message box* di errori critici, ma passa l'errore al processo chiamante.
- SEM_NOGPFAULTERRORBOX: Il sistema non mostra il *Window Error Reporting dialog*.
- SEM_NOOPENFILEERRORBOX: La funzione *OpenFile* non mostra un *message box* in caso di errore, ma passa l'errore al processo chiamante.

Proseguendo viene chiamata l'API SetUnhandledExceptionFilter, che setta un handler che verrà chiamato quando il processo incontra eccezioni che non è in grado di gestire. Il parametro lpTopLevelExceptionFilter che viene passato è un puntatore alla funzione definita in 0041F820.

```

0019FCC8 0042C181 hw4.0042C181
0019FCCC 0041F820 lpTopLevelFilter = <hw4.unh_exception_handler>
0019FCD0 00000001

```

Ghidra riconosce tale funzione come una label, quindi utilizziamo *create function* per ridefinire LAB_0041F820 nella funzione unh_exception_handler.

```

*****
* FUNCTION
*****
undefined __stdcall unh_exception_handler(void)
AL:1 <RETURN>
Stack[-0x4]:1 local_4
XREF[2]: 0041f856(*),
0041f885(*)
Stack[-0x8]:4 local_8
XREF[1]: 0041f8d1(W)
unh_exception_handler
XREF[1]: malware_main:0042a341(*)
0041f820 55 PUSH EBP
0041f821 e9 8f 00 JMP LAB_0041f8b5
00 00

```

Successivamente, sempre nel malware_main, viene effettuata la CALL a FUN_0042CF00, che andiamo quindi ad analizzare nel dettaglio.

```

37 | SetLastError(0x8003);
38 | SetUnhandledExceptionFilter(unh_exception_handler);
39 | FUN_0042cf00();

```

5.3.1 FUN_0042CF00 (token_setup)

Analizzando il decompilato da Ghidra vediamo che si ottiene un handle al processo corrente, e successivamente si invocano due funzioni non riconosciute DAT_0047A058 e DAT_0047A05C. Dopo aver chiuso l'handle precedentemente aperto, il valore di quest'ultima funzione viene ritornato al chiamante `malware_main`.

```
13 | ppvVar5 = &local_8;  
14 | uVar3 = 0;  
15 | uVar4 = 0x80;  
16 | local_c = 0;  
17 | pvVar1 = GetCurrentProcess();  
18 | iVar2 = (*DAT_0047A058)(pvVar1,uVar4,ppvVar5);  
19 | if (iVar2 != 0) {  
20 |     uVar3 = (*DAT_0047A05C)(local_8,0x18,&local_c,4);  
21 |     CloseHandle(local_8);  
22 | }  
23 | return uVar3;  
24 | }
```

Procediamo dunque con un'analisi più dettagliata tramite *OllyDbg* per capire il funzionamento delle due funzioni non riconosciute. Essendo queste variabili globali, sono molto probabilmente funzioni di libreria.

La `GetCurrentProcess` viene invocata all'indirizzo 0042CF0C, mentre all'indirizzo 0042CFF2 scopriamo che la `CALL` verso DAT_0047A058 corrisponde all'invocazione dell'API `OpenProcessToken`.

0042CFF2	> FF15 58A04700	CALL DWORD PTR DS:[47A058]	OpenProcessToken
0042CFF8	. 90	NOP	
0042CFF9	.^FR 99	IMP SHORT hw4.0042CF94	

Tale API apre un accesso token associato al processo. I parametri utilizzati sono i seguenti:

0019FCB0	FFFFFFFF	hProcess = FFFFFFFF
0019FCB4	00000080	DesiredAccess = TOKEN_ADJUST_DEFAULT
0019FCB8	0019FCC4	phToken = 0019FCC4

- `ProcessHandle`: handle al processo corrente.
- `DesiredAccess`: permessi di accesso di default.
- `TokenHandle`: puntatore a un handle che identifica l'access token appena aperto quando la funzione ritorna.

`OpenProcessToken` ritorna 0 se l'operazione ha successo, oppure un valore *non-nullo*. Dal codice decompilato vediamo che si effettua un controllo sul valore ritornato, e solo in caso di successo si chiama DAT_0047A05C.

```
18 | token_succ = (*OpenProcessToken)(pvVar1,uVar3,ppvVar4);  
19 | if (token_succ != 0) {  
20 |     uVar2 = (*DAT_0047A05C)(local_8,0x18,&local_c,4);  
21 |     CloseHandle(local_8);  
22 | }
```

Dal debugger vediamo che l'API invocata è la `SetTokenInformation` che permette di impostare alcune informazioni relative al token d'accesso.

0042CFA2	.^EB E8	JMP SHORT hw4.0042CF8C	
0042CFA4	> 90	NOP	
0042CFA5	. FF15 5CA04700	CALL DWORD PTR DS:[47A05C]	SetTokenInformation

In particolare i parametri passati sono i seguenti:

0019FCAC	00000258	hToken = 00000258 (window)
0019FCB0	00000018	InfoClass = 24.
0019FCB4	0019FCC0	Data = 0019FCC0
0019FCB8	00000004	DataSize = 4

InfoClass = 24 corrisponde a `TokenVirtualizationEnabled`, che permette di scrivere un valore non-nullo se la virtualizzazione è abilitata sul token. Questo valore viene scritto su un buffer tramite puntatore passato come terzo parametro.

Infine la funzione ritorna al chiamante, chiudendo l'handle al processo corrente e ritornando al `malware_main` il risultato della `SetTokenInformation`.

```
18 token_succ = (*OpenProcessToken)(pvVar1,uVar2,ppvVar3);
19 if (token_succ != 0) {
20     info_succ = (*SetTokenInformation)(local_8,24,&local_c,4);
21     CloseHandle(local_8);
22 }
23 return info_succ;
24 }
```

In definitiva questa funzione non sembra implementare nessuna funzionalità diretta dal malware, ma teniamo conto del fatto che sono stati modificati alcuni parametri per ambienti virtualizzati. Quindi rinominiamo `FUN_0042CF00` in `token_setup` e procediamo con l'analisi del `malware_main`.

5.3.2 Language APIs

Proseguendo con l'esplorazione del codice decompilato, vediamo che vengono successivamente invocate le API `GetSystemDefaultLangID`, `GetUserDefaultLangID` e `GetUserDefaultUILanguage`. Queste API ritornano gli identificativi del linguaggio di sistema, del linguaggio utente e dell'interfaccia utente.

```
43 lang_id = GetSystemDefaultLangID();
44 if (((lang_id & 0x3ff) != 0x19) && (lang_id = GetUserDefaultLangID(), (lang_id & 0x3ff) != 0x19))
45     && (lang_id = GetUserDefaultUILanguage(), (lang_id & 0x3ff) != 0x19)) goto LAB_0042a02e;
```

Ognuno di questi identificativi viene salvato nella variabile locale `LVar3` (rinominata `lang_id`), si effettua un AND con `0x3ff`, e si verifica che questo sia differente da `0x19`.

Recuperiamo la lista dei possibili *Language Ids* dalla documentazione² e scriviamo un *semplice programma python*³ con cui testiamo tutti i possibili id, ricavando quali tra questi verificano la condizione `lang_id & 0x3ff != 0x19`.

Vediamo che gli id individuati sono 1049 e 2073, che corrispondono a “*Russian*” e “*Russian – Moldava*”.

```
danilo@pop-os:~/Documenti/uni/corsi/MA/Homework 4 81x24
→ Homework 4 python3 find_lang.py
Il Language ID cercato é: 1049
Il Language ID cercato é: 2073
```

² https://docs.microsoft.com/en-us/openspecs/office_standards/ms-oe376/6c085406-a698-4e12-9d4d-c3b0ee3dbc4a

³ <https://pastebin.com/VBUqQhip>

Quindi se si utilizza la lingua russa non viene effettuato il goto alla label `not_russian`, e viene eseguita la funzione `FUN_00421DE0`. Altrimenti si effettua il `JMP` e si skippa di fatto l'esecuzione di tale funzione.

```

44 | if (((lang_id & 1023) != 25) && (lang_id = GetUserDefaultLangID(), (lang_id & 0x3ff) != 0x19)) &&
45 | (lang_id = GetUserDefaultUILanguage(), (lang_id & 0x3ff) != 0x19)) goto not_russian;
46 | do {
47 |     *(undefined4 *) (unaff_EBP + -4) = 0xffffffff;
48 |     FUN_00431de0();
49 | not_russian:
50 |     if (*(int *) (allocated_mem + 8) != 0) {
51 |         Sleep(*(int *) (allocated_mem + 8) * 1000);
52 |     }

```

5.3.3 FUN_00421DE0 (malware_exe_remove)

Passiamo ad analizzare la funzione `FUN_00421DE0`, che viene invocata solo nel caso in cui sia selezionata la lingua russa nel sistema. Da *Ghidra* vediamo che il function graph risulta molto profondo (61 vertici) e potrebbe risultare molto dispendioso procedere direttamente con l'analisi tramite debugger, per cui cerchiamo innanzitutto di capire il comportamento generale di questa funzione dal decompilato.

Possiamo vedere che vengono invocate le seguenti API:

- `GetModuleFileNameW`: ritorna il path relativo all'eseguibile del processo corrente, poiché viene passato 0 come handle.
- `GetTempPathW`: ritorna il path della cartella "C:\Users\IEUser\AppData\Local\Temp\" usata da Windows per i file temporanei.
- `SetFileAttributesW`: imposta gli attributi del file eseguibile settando l'attributo `0x80`, cioè `FILE_ATTRIBUTES_NORMAL`.
- `GetTempFileNameW`: genera un nome casuale per un file nella cartella `Temp`.
- `MoveFileExW`: si utilizza come flag '0x9', quindi le opzioni `MOVEFILE_REPLACE_EXISTING` e `MOVEFILE_WRITE_THROUGH`; il malware viene copiato nella cartella temporanea, nel path ottenuto in precedenza.
- `MoveFileExW`: si utilizza come flag '0x4' che corrisponde a `MOVEFILE_DELAY_UNTIL_REBOOT`, e 0 come secondo parametro. Con questa combinazione di parametri la `MoveFileExW` registra l'eseguibile per essere eliminato al riavvio del sistema.

Successivamente notiamo la probabile presenza di una stringa offuscata:

```

55 | *(undefined2 *) (unaff_EBP + -0x44) = 99;
56 | *(undefined2 *) (unaff_EBP + -0x42) = 0x6d;
57 | *(undefined2 *) (unaff_EBP + -0x40) = 100;
58 | *(undefined2 *) (unaff_EBP + -0x3e) = 0x2e;
59 | *(undefined2 *) (unaff_EBP + -0x3c) = 0x65;
60 | *(undefined2 *) (unaff_EBP + -0x3a) = 0x78;
61 | *(undefined2 *) (unaff_EBP + -0x38) = 0x65;
62 | *(undefined2 *) (unaff_EBP + -0x36) = 0x20;
63 | *(undefined2 *) (unaff_EBP + -0x34) = 0x2f;
64 | *(undefined2 *) (unaff_EBP + -0x32) = 0x43;
65 | *(undefined2 *) (unaff_EBP + -0x30) = 0x20;
66 | *(undefined2 *) (unaff_EBP + -0x2e) = 100;
67 | *(undefined2 *) (unaff_EBP + -0x2c) = 0x65;
68 | *(undefined2 *) (unaff_EBP + -0x2a) = 0x6c;
69 | *(undefined2 *) (unaff_EBP + -0x28) = 0x20;
70 | *(undefined2 *) (unaff_EBP + -0x26) = 0x2f;
71 | *(undefined2 *) (unaff_EBP + -0x24) = 0x51;
72 | *(undefined2 *) (unaff_EBP + -0x22) = 0x20;
73 | *(undefined2 *) (unaff_EBP + -0x20) = 0x2f;
74 | *(undefined2 *) (unaff_EBP + -0x1c) = 0x20;
75 | *(undefined2 *) (unaff_EBP + -0x1a) = 0x22;
76 | *(undefined2 *) (unaff_EBP + -0x1e) = 0x46;
77 | *(undefined2 *) (unaff_EBP + -0x18) = 0;

```

Convertiamo tale sequenza di caratteri in caratteri sfruttando l'apposito strumento di Ghidra.

```

55 *(undefined2 *) (unaff_EBP + -0x44) = L'c';
56 *(undefined2 *) (unaff_EBP + -0x42) = L'm';
57 *(undefined2 *) (unaff_EBP + -0x40) = L'd';
58 *(undefined2 *) (unaff_EBP + -0x3e) = L'.';
59 *(undefined2 *) (unaff_EBP + -0x3c) = L'e';
60 *(undefined2 *) (unaff_EBP + -0x3a) = L'x';
61 *(undefined2 *) (unaff_EBP + -0x38) = L'e';
62 *(undefined2 *) (unaff_EBP + -0x36) = L' ';
63 *(undefined2 *) (unaff_EBP + -0x34) = L'/';
64 *(undefined2 *) (unaff_EBP + -0x32) = L'C';
65 *(undefined2 *) (unaff_EBP + -0x30) = L' ';
66 *(undefined2 *) (unaff_EBP + -0x2e) = L'd';
67 *(undefined2 *) (unaff_EBP + -0x2c) = L'e';
68 *(undefined2 *) (unaff_EBP + -0x2a) = L'l';
69 *(undefined2 *) (unaff_EBP + -0x28) = L'.';
70 *(undefined2 *) (unaff_EBP + -0x26) = L'/';
71 *(undefined2 *) (unaff_EBP + -0x24) = L'Q';
72 *(undefined2 *) (unaff_EBP + -0x22) = L' ';
73 *(undefined2 *) (unaff_EBP + -0x20) = L'/';
74 *(undefined2 *) (unaff_EBP + -0x1c) = L' ';
75 *(undefined2 *) (unaff_EBP + -0x1a) = L'\"";
76 *(undefined2 *) (unaff_EBP + -0x1e) = L'F';
77 *(undefined2 *) (unaff_EBP + -0x18) = L'\0';

```

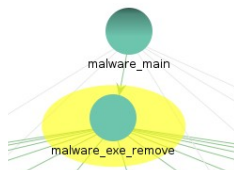
Vediamo che viene composta la stringa "cmd.exe /C del /Q /F", già notata nell'analisi dinamica di base con *Process Monitor*.

- Il flag Q corrisponde alla modalità *quiet*.
- Il flag F corrisponde a *forced*.

Proseguendo, viene chiamata la FUN_0042EC10, passando la stringa appena ricavata. Qui viene poi invocata la `CreateProcessW` che genera un processo che esegue il comando passato, e quindi spawna un terminale per eliminare il malware. Vediamo tramite *function call graph* che FUN_0042EC10 viene utilizzata anche dal `malware_main`, per cui la rinominiamo `run_process` in modo da riconoscerla successivamente.

Dopo aver eseguito il comando viene chiamata la `ExitProcess` con codice 0, senza proseguire di fatto con nessuno dei comportamenti osservati dal malware.

In definitiva sembrerebbe che FUN_00421DE0 si occupa di copiare l'eseguibile del malware in un file temporaneo, per poi eliminare il file `.exe` originale; rinominiamo quindi tale funzione in `malware_exe_remove`. Tramite analisi del *function call graph*, vediamo che viene invocata soltanto dal `malware_main`, e solo nel caso in cui si utilizzi la lingua russa.



Questo ci permette di fare due osservazioni:

1. Il malware sembra innocuo se si utilizza la lingua russa all'interno del sistema.
2. La funzione `malware_exe_remove` non viene utilizzata anche per rimuovere l'eseguibile dopo aver infettato il sistema; ciò significa che bisogna ancora individuare il blocco di codice in cui è implementato tale meccanismo.

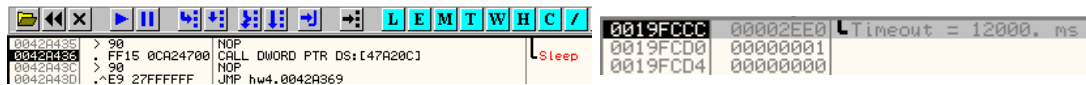
5.3.4 FUN_00431440 (gen_md5_hash)

Dopo il controllo sul linguaggio utilizzato, si entra all'interno di un ciclo do-while molto lungo, che nel decompilato include ben 304 linee di codice. Da questo punto in poi vengono chiamate moltissime funzioni user-defined, per cui l'analisi si limita a quelle più interessanti che implementano le reali funzionalità del ransomware.

Ripartiamo con l'analisi dalla label `not_russian` (0042A02E). Se non si utilizza la lingua russa nel sistema viene eseguita una `Sleep`.

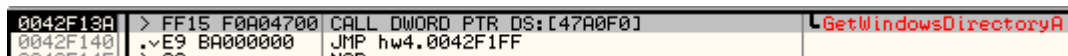
```
49 |not_russian:
50 |    if (*(int *)(allocated_mem + 8) != 0) {
51 |        Sleep(*(int *)(allocated_mem + 8) * 1000);
52 |    }
```

Con *OllyDbg* raggiungiamo la `Sleep` tramite *step over*, e vediamo che viene effettivamente invocata con un timeout di 12 secondi (12000 ms).

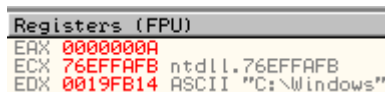


Proseguendo vengono invocate le funzioni `FUN_0041F680`, `FUN_0041F560`, `FUN_0042E4C20` che non sembrano particolarmente interessanti, per cui se ne rimanda una eventuale analisi ad un secondo momento. La `FUN_00431440` invocata successivamente sembra invece molto corposa, per cui passiamo ad analizzarla.

La prima funzione interessante chiamata è `FUN_0042F110`, che a sua volta chiama l'API `GetWindowsDirectoryA`. Questa API recupera il percorso della cartella 'Windows', scrivendolo all'interno del buffer passato tramite puntatore nel primo parametro.



Come atteso la funzione ritorna la stringa "C:\Windows".



Proseguendo si arriva poi ad invocare la `FUN_0042F430`, che chiama a sua volta la `FUN_0042F2E0`. Qui si utilizza l'API `GetVolumeNameForVolumeMountPointA`, che restituisce il *GUID path* del volume fornito come primo parametro di input.

```
2 | undefined * __cdecl FUN_0042f2e0(undefined *param_1,LPCSTR param_2)
3 |
4 | {
5 |     BOOL BVar1;
6 |     undefined4 local_110 [65];
7 |     undefined **local_c;
8 |     DWORD local_8;
9 |
10 |     local_8 = 0;
11 |     BVar1 = GetVolumeNameForVolumeMountPointA(param_2, (LPSTR)local_110,0x104);
```

Proseguiamo con *OllyDbg*, e vediamo che come parametro viene passato proprio la stringa "C:\Windows\".

```
0042F384 | > FF15 ECA04700 | CALL DWORD PTR DS:[47A0EC] | KERNEL32.GetVolumeNameForVolumeMountPointA
0042F38A | . 90 | NOP

0019FAF0 | 0019FC70 | ASCII "C:\Windows\"
0019FAF4 | 0019FAFC |
```

Tuttavia l'API fallisce, causando un'eccezione e restituendo come errore `ERROR_NOT_A_REPARSE_POINT`.

```
EIP 0042F38A hw4.0042F38A
C 0 ES 002B 32bit 0(FFFFFFFF)
P 1 CS 0023 32bit 0(FFFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFFF)
Z 1 DS 002B 32bit 0(FFFFFFFF)
S 0 FS 0053 32bit 343000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
O 0 LastErr ERROR_NOT_A_REPARSE_POINT (00001126)
```

Proseguendo con *step over*, ad un certo punto il programma entra in pausa.

```
74C6C632 | 8B4C24 54 | MOV ECX,DWORD PTR SS:[ESP+54]
74C6C636 | 33CC | XOR ECX,ESP
74C6C638 | E8 C34A0000 | CALL KERNELBA.74C71100
74C6C63D | 8BE5 | MOV ESP,EBP
74C6C63F | 5D | POP EBP
```

Procediamo con `SHIFT+F9` passando l'eccezione all'applicazione, e vediamo che viene risolta correttamente. Successivamente si ritorna alla chiamata della `FUN_0042F2E0`.

```
0042F445 | > E8 96FEFFFF | CALL hw4.0042F2E0
0042F44A | . 90 | NOP
0042F44B | .^EB F1 | JMP SHORT hw4.0042F43E
```

Viene quindi invocata nuovamente la `GetVolumeNameForVolumeMountPointA`, passando ora come parametro "C:\". Eseguendo la chiamata vediamo che si ha nuovamente un `ERROR_MORE_DATA`, ma il GUID sembra comunque essere stato scritto all'indirizzo `0019FAF4`.

```
Registers (FPU)
EAX 00000001
ECX 757FBF6A KERNEL32.757FBF6A
EDX 00620000
EBX 00000000
ESP 0019FAFC ASCII "\\?\Volume{a04afb1-0000-0000-0000-100000000000}\\"
EBP 0019FC08
ESI 0019FC08
EDI 00000001
EIP 0042F38A hw4.0042F38A
C 0 ES 002B 32bit 0(FFFFFFFF)
P 1 CS 0023 32bit 0(FFFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFFF)
Z 1 DS 002B 32bit 0(FFFFFFFF)
S 0 FS 0053 32bit 343000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
O 0 LastErr ERROR_MORE_DATA (000000EA)

0019FAF0 | 0019FC70 | ASCII "C:\"
0019FAF4 | 0019FAFC | ASCII "\\?\Volume{a04afb1-0000-0000-0000-100000000000}\\"
```

Ora è possibile continuare con l'esecuzione. Da *Ghidra* vediamo che in `FUN_00416570` viene probabilmente invocata una funzione di libreria, non correttamente riconosciuta dal decompilatore.

```
12 | iVar1 = (*DAT_0047a07c)(this,0,0,param_1,param_2);
13 | if (iVar1 == 0) {
14 |     local_8 = (void *)GetLastError();
15 |     local_c = &PTR_LAB_0047c80c;
16 |     /* WARNING: Subroutine does not return */
```


Vediamo da *OllyDbg* che questa corrisponde a **CryptAcquireContextA**, che permette di acquisire un handle ad un container di chiavi all'interno di uno specifico *Cryptographic Service Provider*⁴. Definiamo tale funzione anche sul codice decompilato.

00416504	> FF15 7CA04700	CALL DWORD PTR DS:[47A07C]	advapi32.CryptAcquireContextA
0041650A	. 90	NOP	
0041650B	. ^EB C9	JMP SHORT hw4.004165A5	
0041650D	> 8D12	LEA EDX, DWORD PTR DS:[EDX]	
0041650F	. ^EB E5	JMP SHORT hw4.004165C6	
004165E1	> 90	NOP	

Anche in FUN_0042CB70 vi è una funzione non riconosciuta, che vediamo tramite *OllyDbg* corrispondere a **CryptCreateHash**.

0042CB0F	. FF15 4CA04700	CALL DWORD PTR DS:[47A04C]	advapi32.CryptCreateHash
0042CB0E	. ^EB 69	JMP SHORT hw4.0042CC50	
0042CB07	> 51	PUSH ECX	
0042CB08	. 90	NOP	
0042CB09	. ^EB 75	JMP SHORT hw4.0042CC60	
0042CB0B	> 90	NOP	
0042CB0C	. 56	PUSH ESI	

Questa API inizializza l'hashing di uno stream di dati, restituendo al chiamante un handle ad un *hash object*. Il secondo parametro *AlgId* specifica l'algoritmo di hashing da utilizzare, e viene passato come **CALG_MD5**. Questo ci permette di fare due osservazioni:

1. In questo blocco di codice viene utilizzato MD5 come funzione hash.
2. Probabilmente non stiamo ancora analizzando le funzioni che cifrano i file dell'utente, in quanto, almeno secondo il messaggio del ransomware, questo dovrebbe far uso di RSA-2048 e AES-128.

Successivamente si invoca la FUN_0042CDD0; anche al suo interno viene chiamata una funzione non riconosciuta, che da *OllyDbg* risulta essere **CryptHashData**, che inserisce dati all'*hash object* aperto in precedenza.

0042CDE0	> FF15 00A04700	CALL DWORD PTR DS:[47A000]	advapi32.CryptHashData
0042CDF3	. 90	NOP	
0042CDF4	. ^EB 2E	JMP SHORT hw4.0042CE24	

I dati aggiunti all'oggetto corrispondono proprio al GUID del volume C:// ottenuto in precedenza.

```
0019FC1C|0019FCC4|
0019FC20|0043161C|RETURN to hw4.0043161C from hw4.0042CDD0
0019FC24|022911B0|ASCII "(a04afba1-0000-0000-0000-100000000000)"
```

Proseguendo si ha la seguente catena di invocazioni FUN_00430010 → FUN_0042fA00 → FUN_0042CCB0. Quest'ultima funzione chiama la API **CryptGetHashParam**, che recupera i dati che governano le operazioni sull'oggetto hash aperto precedentemente. Anche in questo caso la funzione non è riconosciuta dal decompilatore, ed è stata individuata tramite debugger.

0042CD08	> FF15 54A04700	CALL DWORD PTR DS:[47A054]	advapi32.CryptGetHashParam
0042CD0E	. 90	NOP	
0042CD0F	. ^EB 45	JMP SHORT hw4.0042CD56	

Ritornando alla FUN_00431440, vediamo che vengono invocate due funzioni, ancora non riconosciute da Ghidra.

```
59 | (*DAT_0047a050)(*(undefined4 *) (unaff_EBP + -0x18));
50 | }
51 | *(undefined4 *) (unaff_EBP + -4) = 0x0;
52 | if (*(int *) (unaff_EBP + -0x1c) != 0) {
53 |     (*DAT_0047a018)(*(undefined4 *) (unaff_EBP + -0x1c), 0);
54 | }
```

4 CSP: Modulo software che esegue algoritmi crittografici per autenticazione, cifratura ed encoding

Tramite *OllyDbg* queste funzioni risultano essere la `CryptDestroyHash` per rimuovere l'oggetto hash precedentemente generato ed infine la `CryptReleaseContext` per rilasciare l'handle al *CSP*.

```

00431828 . FF15 50A04700 CALL DWORD PTR DS:[47A050] advapi32.CryptDestroyHash
0043182E | > 90                NOP

004318B8 . FF15 18A04700 CALL DWORD PTR DS:[47A018] advapi32.CryptReleaseContext
004318C1 | > E9 88F0FFFF JMP hw4.0043194E

```

Come effetto visibile di `FUN_00431440` possiamo osservare che è stata scritta sulla locazione `0019FC54` la stringa `1BCC0199BD4620BF`, che probabilmente corrisponde ad un qualche hash MD5 generato a partire dal GUID di `C://`.

```

0019FC54 022911F8 ASCII "1BCC0199BD4620BF"
0019FC58 00000000
0019FC5C 00000000
0019FC60 00000000
0019FC64 00000000

```

Prima di proseguire con l'analisi dell'eseguibile rinominiamo dunque `FUN_00431440` in `gen_md5_hash`. Inoltre potremmo nuovamente incontrare le funzioni analizzate in questo paragrafo, quindi per riconoscerle più facilmente in futuro andiamo a rinominare anche quelle.

- `FUN_0042f110` in `get_win_dir`
- `FUN_0042F430` in `get_guid`
- `FUN_00416570` in `get_crypt_context`
- `FUN_0042CDD0` in `crypt_hash_data`
- `FUN_00430010`, `FUN_0042fA00`, `FUN_0042CCB0` in `get_hash_param0/1/2`.

5.3.5 Altre funzioni

A questo punto vengono invocate molte altre funzioni definite dallo sviluppatore. Non potendo analizzarle tutte nel dettaglio sfruttiamo il *function call trees* su ognuna di queste e ci appuntiamo le principali API che queste utilizzano. In questo modo teniamo teniamo d'occhio queste funzioni, così da poterle analizzare in seguito se necessario.

- `FUN_00426A40` utilizza `OpenMutexA` e `CreateEventA`
- `FUN_004270F0` utilizza `FindAtomA` e `GlobalFindAtomA`

Successivamente viene copiata nella variabile locale `puVar8` il contenuto di `(allocated_mem + 15)`. Da qui si ha un costrutto `if/else`, in cui si verifica se `puVar8 == '\0'`. Prima di proseguire con l'analisi vediamo con *OllyDbg* quale è la decisione presa rispetto a questo branch.

```

0042BDC7 . ^0F84 85EBFFFF JE hw4.0042A952
0042BDCD | . 90                NOP

```

L'istruzione `JE` effettua il `JMP` a `0042A952` se `ZF==1`. Vediamo che in effetti il `JMP` viene effettuato, per cui il flusso di esecuzione continua nel ramo `if` invece che nel ramo `else`.

Questo ci semplifica (*almeno per ora*) l'analisi di `malware_main`, in quanto possiamo analizzare le sole funzioni presenti nell'`if`, ignorando tutte quelle invocate nell'`else`.

Analizzando tali funzioni, nessuna sembra implementare i meccanismi principali del malware, quindi anche in questo caso ci limitiamo per il momento ad elencare le principali API che utilizzano:

- FUN_00423740: `GetCurrentProcess`, `GetProcessAddress`, `GetLastError` e `GetVersionExA`
- FUN_004203a0: `HeapFree` e `GetLastError`
- FUN_0041F9D0 viene invocata 3 volte, e presenta una chiamata a `RaiseException`, quindi in caso di errori con il debugger torneremo ad indagare su questa funzione.
- FUN_00418CE0 utilizza `_memchr`.
- FUN_0041E440 viene chiamata più volte nel ramo `if`, e chiama a sua volta molte funzioni definite dallo sviluppatore. Inoltre utilizza `RaiseException`, per cui la teniamo d'occhio.

Subito fuori dall' `if/else` viene invocata la FUN_0046C640. Sempre sfruttando il *Function Call Trees* vediamo che utilizza internamente le API `GetDiskFreeSpaceExW`, `GetDriveTypeW`, `GetVolumeInformationW` e `GetLogicalDrives`, per cui passiamo ad un'analisi più dettagliata.

5.3.6 FUN_0046C150 (network_work)

Analizzando la FUN_0046C640, vediamo che vengono azzerati alcuni parametri sullo stack, e successivamente viene invocata tre volte la FUN_0046C150. Questa funzione internamente utilizza alcune API per la comunicazione internet (`WnetOpenEnumW`, `WnetCloseEnum`, `WNetAddConnection2W`) per cui diamo uno sguardo più approfondito.

Viene innanzitutto invocata la `WnetOpenEnumW`. Dalla documentazione vediamo che questa API inizia l'enumerazione delle risorse di rete o delle connessioni aperte; l'enumerazione può continuare chiamando la `WnetEnumResource`. Successivamente si alloca un'area di *8192 bytes* tramite `calloc`, ed il puntatore all'area allocata viene salvato in una variabile locale `lpNetResource`.

La `WnetEnumResource` viene invocata successivamente nella FUN_00461200, e continua l'enumerazione delle risorse iniziata in precedenza. Come terzo parametri si passa il puntatore all'area allocata in precedenza, su cui verrà scritta una struttura `NETRESOURCEW` con i risultati dell'enumerazione.

La FUN_00461200 ritorna `true` se `WnetEnumResource` ritorna `NO_ERROR`, e ritornando al chiamante si effettua un controllo proprio sul valore restituito; se questo è `true`, vengono effettuati alcuni controlli sul campo `dwUsage` della struttura, che indica come è possibile effettuare l'enumerazione.

Se invece la FUN_00461200 ritorna `false` viene chiamata la `WnetCloseEnum` che chiude l'enumerazione iniziata in precedenza. In definitiva questa funzione non sembra implementare particolari funzionalità del malware, per cui proseguiamo rinominando FUN_00461200 in `enum net_res` ed il chiamante FUN_0046C150 in `network work`.

A network diagram showing three nodes: `malware_main`, `FUN_0046c640`, and `network_work`. `malware_main` is connected to `FUN_0046c640` by a green arrow. `FUN_0046c640` is connected to `network_work` by a green arrow. `network_work` has a self-loop and is connected to several other nodes (represented by green lines) that are not labeled.

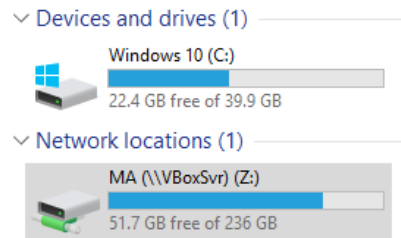
Come già visto questa funzione interagisce con i dischi di windows. La prima API che viene chiamata è `GetLogicalDrives`, che restituisce una *bitmask* sui dischi attualmente disponibili nel sistema.

```
0 0 LastErr ERROR_NO_MORE_ITEMS (00000103)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
```

0046CD49 > FF15 14A14700 CALL DWORD PTR DS:[47A114] cGetLogicalDrives
0046CD4F ^E9 3AFEFFFF JMP hw4.0046CB8E
0046CD54 > 90 NOP

Registers (FPU)	
EAX	02000004

Attualmente sulla VM sono montati i dischi C:// e Z://, ed infatti convertendo in binario il valore ritornato dall'API otteniamo 001000000000000000000000100, che specifica proprio la presenza dei due dischi.



Successivamente viene chiamata l'API `GetDriveTypeW`: come parametro `lpRootPathName` questa richiede la directory root del disco, e ritorna in output la tipologia di drive. Nel nostro caso viene passata la stringa "c:\\" ricavata a partire dalla bitmask, e ci viene restituito il valore 3 che indica un *disco fisso*.



Si effettua poi un controllo sul valore ritornato: le API `GetDiskFreeSpace` e `GetVolumeInformation` vengono invocate soltanto se viene restituito come valore 2, quindi solo se il disco è un *disco rimovibile*. Nel nostro caso i dischi C e Z sono rispettivamente `DISK_FIXED` e `DRIVE_REMOTE`, per cui queste API non vengono di fatto invocate.

Nel caso in cui sia montato un disco rimovibile, con `GetDiskFreeSpace` si ottiene lo spazio libero, mentre con `GetVolumeInformation` si ricavano informazioni sul file system e sul volume stesso.

In definitiva la `FUN_0046C640` ottiene informazioni sui dischi installati sul sistema, per cui la rinominiamo in `get_drives_info`.

5.3.8 FUN_0046C640 (priv_spawn_thread)

Tornando al `malware_main` si effettua un nuovo controllo su `allocated_mem`, e se all'offset `0x6493` è presente un valore diverso da `'\0'` viene invocata la `FUN_0046C640`. Vediamo che questa invoca quattro volte `FUN_00474820`, che prende una sola stringa in input.

Come parametri nelle 4 chiamate vengono passati i valori:

- `s_SeDebugPrivilege_0047ef28` → `SeDebugPrivilege`
- `s_SeTakeOwnershipPrivilege_0047ef08` → `SeTakeOwnershipPrivilege`
- `s_SeBackupPrivilege_0047eef0` → `SeBackupPrivilege`
- `s_SeRestorePrivilege_0047eed8` → `SeRestorePrivilege`

Questo ci suggerisce che `FUN_00474820` va in qualche modo a manipolare i privilegi del processo, per cui la rinominiamo `set_privileges`.

Analizziamone il codice decompilato.

```

17 |   pvVar1 = GetCurrentProcess();
18 |   uVar2 = (*OpenProcessToken)(pvVar1,uVar6,ppvVar7);
19 |   if (uVar2 != 0) {
20 |       iVar3 = (*DAT_0047a004)(0,param_1,&DAT_0048206c);
21 |       if (iVar3 != 0) {
22 |           iVar3 = (*DAT_0047a008)(local_8,0,&DAT_00482068,0,0,0);
23 |           if (iVar3 != 0) {
24 |               DVar4 = GetLastError();
25 |               uVar5 = 0;
26 |               if (DVar4 == 0) {
27 |                   uVar5 = 1;
28 |               }
29 |           }
30 |       }
31 |       uVar2 = CloseHandle(local_8);
32 |   }
33 |   return uVar2 & 0xffffffff00 | uVar5;

```

Viene invocata innanzitutto la `GetCurrentProcess` per ottenere l'handle al processo corrente, e poi la `OpenProcessToken` (già vista in precedenza). Vengono poi invocate altre due funzioni di libreria non riconosciute da *Ghidra*, per cui procediamo step-by-step con *OllyDbg*. Le due API sono la `LookupPrivilegeValueA` e `AdjustTokenPrivileges`.

0047493B	. FF15 04A04700	CALL DWORD PTR DS:[47A004]	LookupPrivilegeValueA
00474941	. > EB 2E	JMP SHORT hw4.00474971	
00474943	. > 90	NOP	

00474893	. FF15 08A04700	CALL DWORD PTR DS:[47A008]	AdjustTokenPrivileges
00474899	. 90	NOP	
0047489A	. > E9 B6000000	JMP hw4.00474955	

`LookupPrivilegeValueA` fornisce il *locally unique identifier* (LUID) usato sul sistema per rappresentare un determinato privilegio, passato come secondo parametro. Viene in questo caso richiesto quindi il LUID del privilegio passato a `set_privileges`, in particolare per i quattro privilegi visti in precedenza.

`AdjustTokenPrivileges`, permette di abilitare o disabilitare i privilegi su un token di accesso, e quindi di effettuare l'effettiva *privilege escalation* del ransomware.

Tornando alla `FUN_0046C640`, viene invocata una `GetModuleHandleA`, seguita da tre `GetProcAddress`, probabilmente per ottenere gli indirizzi di alcune funzioni. Ci aiutiamo nuovamente con *OllyDbg*, e constatiamo che si ottiene l'handle a `ntdll.dll`, e poi l'indirizzo delle API `NtQuerySystemInformation`, `NtDuplicateObject` e `NtQueryObject`. Riportiamo questa informazione su Ghidra modificando opportunamente le variabili globali su cui tali funzioni vengono caricate.

```

40 |   hModule = GetModuleHandleA((LPCSTR)&local_28);
41 |   ntdll_handle = hModule;
42 |   if (hModule != (HMODULE)0x0) {
    |       ...
50 |       NtQuerySystemInformation = GetProcAddress(hModule,(LPCSTR)&local_58);
    |       ...
57 |       NtDuplicateObject = GetProcAddress(ntdll_handle,(LPCSTR)&local_3c);
    |       ...
63 |       hModule = (HMODULE)GetProcAddress(ntdll_handle,(LPCSTR)&local_1c);
64 |       NtQueryObject = hModule;
65 |   }

```

Se tutte le API vengono caricate correttamente, viene invocata `CreateThread`, che spawna un thread, specificando come routine da eseguire `LAB_004768D0`. Per mantenere la discussione di `malware_main` più lineare, rimandiamo ad un secondo momento l'analisi sul funzionamento di questo thread.

Infine la `FUN_0046C640` chiude l'handle al modulo e ritorna al `malware_main`. Tale funzione complessivamente modifica i privilegi del malware, acquisendo in particolare i e spawna un certo thread probabilmente malevolo, per cui la rinominiamo in `priv_spawn_thread`.

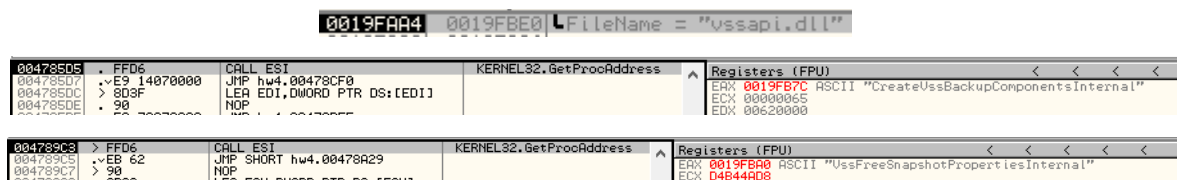
5.3.9 FUN_0040FA10 (vss_backup)

Tornando al `malware_main`, notiamo un ciclo `for`, all'interno del quale viene nuovamente invocata `CreateThread`. Come routine viene in questo caso specificato `LAB_00429B40`, e come fatto in precedenza ne **rimandiamo l'analisi** ad un secondo momento.

La prima funzione fuori dal ciclo `for` è `FUN_0040FA10`. Vediamo che vengono invocate numerose API come `CoInitializeEx`, `CoInitializeSecurity`, e `GetSystemDirectoryW`. Specialmente quest'ultima potrebbe indicare una funzionalità principale del ransomware, per cui analizziamo tale funzione.

Viene innanzitutto invocata `CoInitializeEx` per inializzare la libreria COM, che permette di creare oggetti per la comunicazione fra processi. Successivamente viene chiamata la `CoInitializeSecurity`, che imposta valori di default sulla la sicurezza del processo corrente.

Viene poi chiamata la `FUN_00478420`, che non viene riconosciuta correttamente da *Ghidra*, per cui proseguiamo tramite il debugger. Complessivamente viene chiamata `LoadLibraryA`, per caricare la libreria `vssapi.dll`, e poi `GetProcAddress` per caricare le funzioni `CreateVssBackupComponentsInternal` e `VssFreeSnapshotPropertiesInternal`. Queste due API permettono di gestire operazioni di backup su alcuni oggetti, e sembrano non essere direttamente legate al ransomware, per cui proseguiamo oltre.



Infine viene invocata la `FUN_0040F5D0`, che utilizza l'API `GetSystemDirectoryW` per recuperare il path `C:/Windows` (come visto in precedenza), e poi la stringa `'vssadmin.exe'`. Questo eseguibile va proprio ad utilizzare le librerie precedentemente caricate per visualizzare i backup correnti delle copie shadow.

```
0019FBF8 0019FC98 UNICODE "\vssadmin.exe"
0019FBFC 0019FC28
```

Anche questa funzione non sembra particolarmente interessante in relazione al ransomware, per cui rinominiamo `FUN_0040FA10` in `vss_backup` e torniamo al `malware_main`.

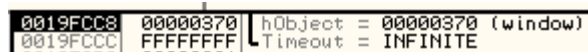
5.3.10 WaitForSingleObject Bypass

Continuando nel `malware_main` si ha un ciclo, in cui viene invocata più volte `WaitForSingleObject`. Abbiamo già incontrato questa API durante la fase di unpacking, e mette il processo in attesa finché non viene segnalato un certo oggetto.

Proseguendo con l'analisi del decompilato si incontra la `FUN_004273C0`, al cui interno vengono invocate ancora `AddAtomA` e `GlobalAtomA`. Come fatto in precedenza ignoriamo questa funzione, e la analizziamo successivamente se necessario.

Più avanti si incontra la `FUN_00425870`. Analizzando il decompilato si può vedere che vengono invocate moltissime funzioni user-defined, e all'interno di queste molte non vengono riconosciute da Ghidra. Ci spostiamo quindi su *OllyDbg* per cercare di capire le API utilizzate.

Ci posizioniamo con F4 su `0042B04F` dove avviene la `CALL` a `FUN_00425870`. Tuttavia vediamo che l'esecuzione entra in stallo, ed è necessario capire dove. Utilizzo a tale scopo *animate over* partendo dal `malware_main`, e vediamo che il debugger resta bloccato sulla chiamata a `WaitForSingleObject`. Ovviamente questo non dovrebbe succedere nell'esecuzione normale del malware, per cui molto probabilmente è presente un altro meccanismo di antidebug, che impedisce la segnalazione dell'oggetto di cui `WaitForSingleObject` è in attesa.



```
0019FCC8 00000370 hObject = 00000370 (window)
0019FCCC FFFFFFFF Timeout = INFINITE
```

Per il momento ci limitiamo a bypassare la `WaitForSingleObject` inserendo un NOP al suo posto, tenendo a mente che eventuali errori futuri potrebbero dipendere proprio da questa modifica.

5.3.11 FUN_00425870 (malware_desktop_asasin)

Possiamo proseguire ora con l'analisi di `FUN_00425870` con il debugger. La prima funzione chiamata è `FUN_00420A70`, che vediamo invoca `SHGetFolderPathW` (non riconosciuta da Ghidra). Questa API ottiene il path di una cartella mediante il suo valore `CSIDL`, e lo copia nel buffer passato come primo parametro. Vediamo che a seguito della chiamata viene copiato il percorso del desktop all'indirizzo `0019F94C`.



```
0019F948 0019F94C UNICODE "C:\Users\IEUser\Desktop"
0019F94C 003A0043
0019F950 0055005C
```

Dal decompilato vediamo che vengono caricate in due zone di memoria differenti le stringhe `' .htm'` e `' .bmp'`.

```
*(undefined2 *) (unaff_EBP + -0x94) = L'.';
*(undefined2 *) (unaff_EBP + -0x92) = L'h';
*(undefined2 *) (unaff_EBP + -0x90) = L't';
*(undefined2 *) (unaff_EBP + -0x8e) = L'm';
*(undefined2 *) (unaff_EBP + -0x8c) = 0;

*(undefined2 *) (unaff_EBP + -0x70) = L'.';
*(undefined2 *) (unaff_EBP + -0x6e) = L'b';
*(undefined2 *) (unaff_EBP + -0x6c) = L'm';
*(undefined2 *) (unaff_EBP + -0x6a) = L'p';
*(undefined2 *) (unaff_EBP + -0x68) = 0;
```


All'indirizzo 0047CBE0 vediamo che è presente la stringa 'asasin'.

Address	Hex dump	ASCII
0047CBE0	61 00 73 00 61 00 73 00	a.s.a.s.
0047CBE8	69 00 6E 00 00 00 00 00	i.n.....
0047CBF0	00 00 00 00 00 00 00 00

Seguono poi diverse chiamate alla funzione FUN_0040F9B, che complessivamente vanno a concatenare varie stringhe fino a generare i path "C:\Users\IEUser\Desktop\asasin.htm" e "C:\Users\IEUser\Desktop\asasin.bmp", che vengono salvate rispettivamente agli indirizzi 0019FBF4 e 0019FC10. Rinominiamo quindi FUN_0040F9B in asasin_path.

```

0019FBF4 022949E8 UNICODE "C:\Users\IEUser\Desktop\asasin.htm"
0019FBF8 022946C0
0019FBFC 76EE50A0 RETURN to ntdll.76EE50A0 from ntdll.76EE5420
0019FC00 00000000
0019FC04 00000022
0019FC08 0000002E
0019FC0C 00D9C9F2
0019FC10 02294530 UNICODE "C:\Users\IEUser\Desktop\asasin.bmp"

```

Viene poi invocata due volte FUN_0042E6B0.

```

44 | uVar3 = FUN_0042e6b0((undefined4 *) (unaff_EBP + -0xcc));
45 | if ((char)uVar3 == '\0') {
46 |     FUN_0042e7d0();
47 | }
48 | uVar3 = FUN_0042e6b0((undefined4 *) (unaff_EBP + -0xb0));

```

Questa internamente chiama l'API FindFirstFileW che restituisce un handle al file specificato nel primo parametro lpFileName. La funzione ritorna un handle se il file è stato trovato, INVALID_HANDLE_VALUE altrimenti. Se il file non viene trovato si ha un errore ERROR_FILE_NOT_FOUND, e viene ritornato 0.

```

Registers (FPU)
EAX FFFFFFF0
ECX 9B6A6B14
EDX 00000000
EBX 00000000
ESP 0019FB74
EBP 0019FCC0
ESI 00000001
EDI 0019FC5A
EIP 00426322 hw4.00426322
C 0 ES 002B 32bit 0(FFFFFFFF)
P 1 CS 0023 32bit 0(FFFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFFF)
Z 1 DS 002B 32bit 0(FFFFFFFF)
S 0 FS 0053 32bit 343000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
O 0 LastErr ERROR_FILE_NOT_FOUND (00000002)

```

Quindi FUN_0042E6B0 verifica la presenza di un file 'asasin' sul desktop, per cui la rinominiamo in asasin_exists.

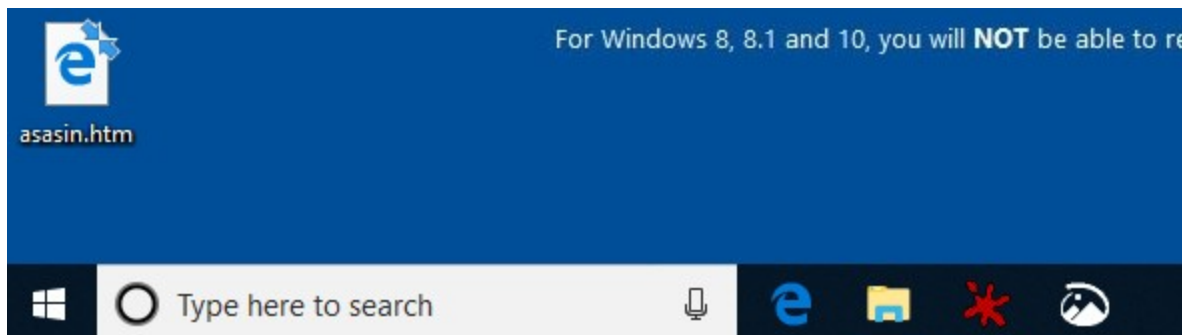
Se il file cercato non esiste, si entra nell'if, e viene invocata FUN_0042E7D0. Questa funzione utilizza internamente l'API CreateFileW, quindi presumibilmente dovrebbe creare i file asasin.htm e asasin.bmp sul desktop. Rinominiamo quindi FUN_0042E7D0 in create_asasin_file.

Senza analizzarla troppo nel dettaglio, nel nostro caso non sono presenti i file asasin.bmp e asasin.htm sul desktop, per cui create_asasin_file viene invocata due volte.

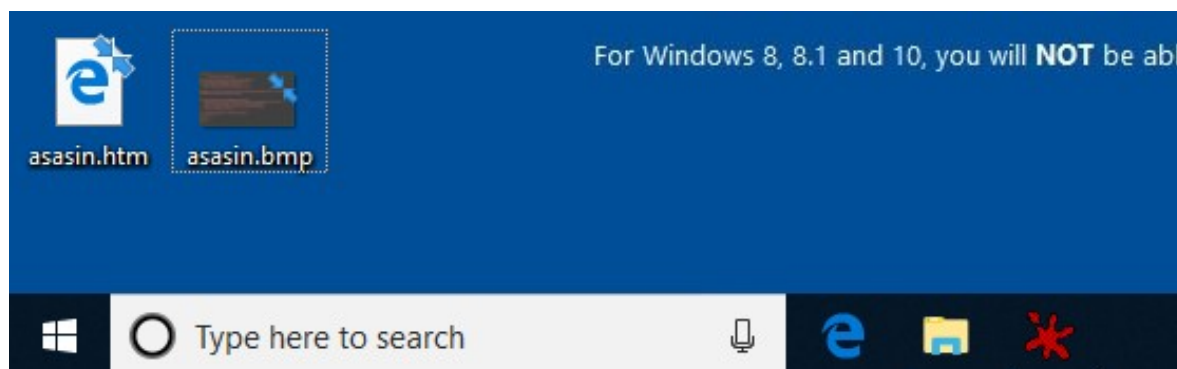
Procediamo quindi con il debugger senza entrare nella function call con *step over*, e vediamo che l'operazione viene eseguita con successo.

```
C 0 ES 002B 32bit 0(FFFFFFFF)
P 1 CS 0023 32bit 0(FFFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFFF)
Z 1 DS 002B 32bit 0(FFFFFFFF)
S 0 FS 0053 32bit 343000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
```

La prima chiamata genera "C:\Users\IEUser\Desktop\asasin.htm".



La seconda chiamata genera "C:\Users\IEUser\Desktop\asasin.bmp".



Proseguendo, vengono eseguite alcune operazioni che recuperano la stringa "Control Panel\Desktop", che viene copiata in EAX e poi passata come parametro alla FUN_00416290.

Registers (FPU)		Memory Dump		ASCII
EAX	0019FC78	0019FB70	0019FC78	ASCII "Control Panel\Desktop"
ECX	04006020	0019FB74	0002001F	
EDX	02290000	0019FB78	0019FEAF	
EBX	00000000	0019FB7C	022916B8	
		0019FB80	00000000	
		0019FB84	00000000	

Internamente viene chiamata a 00416296 una funzione non riconosciuta da *Ghidra*, che tramite *OllyDbg* vediamo essere la *RegOpenKeyExA*.

Address	Disassembly	Comment
00416296	> FF15 6CA04700	CALL DWORD PTR DS:[47A06C]
0041629C	. 90	NOP

Questa API permette di *aprire una chiave di registro*. Analizzando i parametri passati alla funzione vediamo che la chiave è HKEY_CURRENT_USER e la sottochiave è "Control Panel\Desktop". Tra i flag di accesso viene specificato KEY_CREATE_SUB_KEY, per cui la sottochiave verrà creata se non esiste.

```

0019FB44  00000001  hKey = HKEY_CURRENT_USER
0019FB48  0019FC78  Subkey = "Control Panel\Desktop"
0019FB4C  00000000  Reserved = 0
0019FB50  0002001F  Access = KEY_QUERY_VALUE|KEY_SET_VALUE|KEY_CREATE_SUB_KEY|KEY_ENUMERATE_SUB_KEYS|KEY_NOTIFY|20000
0019FB54  0019FCB0  hHandle = 0019FCB0

```

Rinominiamo quindi FUN_00416290 in open_desk_key e continuiamo con l'analisi della FUN_00425870.

Similmente a quanto fatto in precedenza, viene copiata la stringa "WallpaperStyleQ" in EAX, per invocare poi la FUN_004205A0. All'interno di tale funzione viene invocata FUN_00416390 che utilizza l'API RegSetValueExA, permettendo di impostare i dati e il tipo di uno specifico valore di una registry key.

In particolare qui viene utilizzato l'handle alla chiave aperta in precedenza, e si imposta proprio il campo WallpaperStyle della chiave Control Panel\Desktop a 0.

```

0019FB04  000003E4  hKey = 3E4
0019FB08  0019FB40  ValueName = "WallpaperStyle"
0019FB0C  00000000  Reserved = 0
0019FB10  00000001  ValueType = REG_SZ
0019FB14  0019FC5C  Buffer = 0019FC5C
0019FB18  00000002  BufSize = 2

```

Rinominiamo quindi FUN_004205A0 in edit_key e FUN_00416390 in set_key_value.

Successivamente viene copiata in EAX la stringa "TileWallpaper", e viene poi chiamata nuovamente edit_key. In questo caso la RegSetValueExA andrà a impostare il valore di TileWallpaper a 0.

```

0019FB04  000003E4  hKey = 3E4
0019FB08  0019FB40  ValueName = "TileWallpaper"
0019FB0C  00000000  Reserved = 0
0019FB10  00000001  ValueType = REG_SZ
0019FB14  0019FC5C  Buffer = 0019FC5C
0019FB18  00000002  BufSize = 2
0019FB1C  0019FCB0

```

Avendo sia TileWallpaper che WallpaperStyle settati a 0, lo sfondo del desktop sarà *centrato, non-tiled e non-stretched*.

TileWallpaper	REG_SZ	0
TranscodedImageCache	REG_BINARY	7a c3 01 00 36 c0 3c 00 80 04 00 00 60 03 00 00 b5 1...
TranscodedImageCount	REG_DWORD	0x00000001 (1)
UserPreferencesMask	REG_BINARY	9e 1e 07 80 12 00 00 00
WallPaper	REG_SZ	C:\Users\IUser\AppData\Local\Temp\BGInfo.bmp
WallpaperOriginX	REG_DWORD	0x00000000 (0)
WallpaperOriginY	REG_DWORD	0x00000000 (0)
WallpaperStyle	REG_SZ	0

Proseguendo con FUN_00425870, a 0042601 vi è una CALL a SystemParameterInfoW, che permette di recuperare o impostare il valore di uno dei parametri di sistema. In questo caso viene specificata l'opzione SPI_SETDESKWALLPAPER, che imposta come sfondo del desktop l'immagine passata come terzo parametro, ovvero asasin.bmp creata in precedenza.

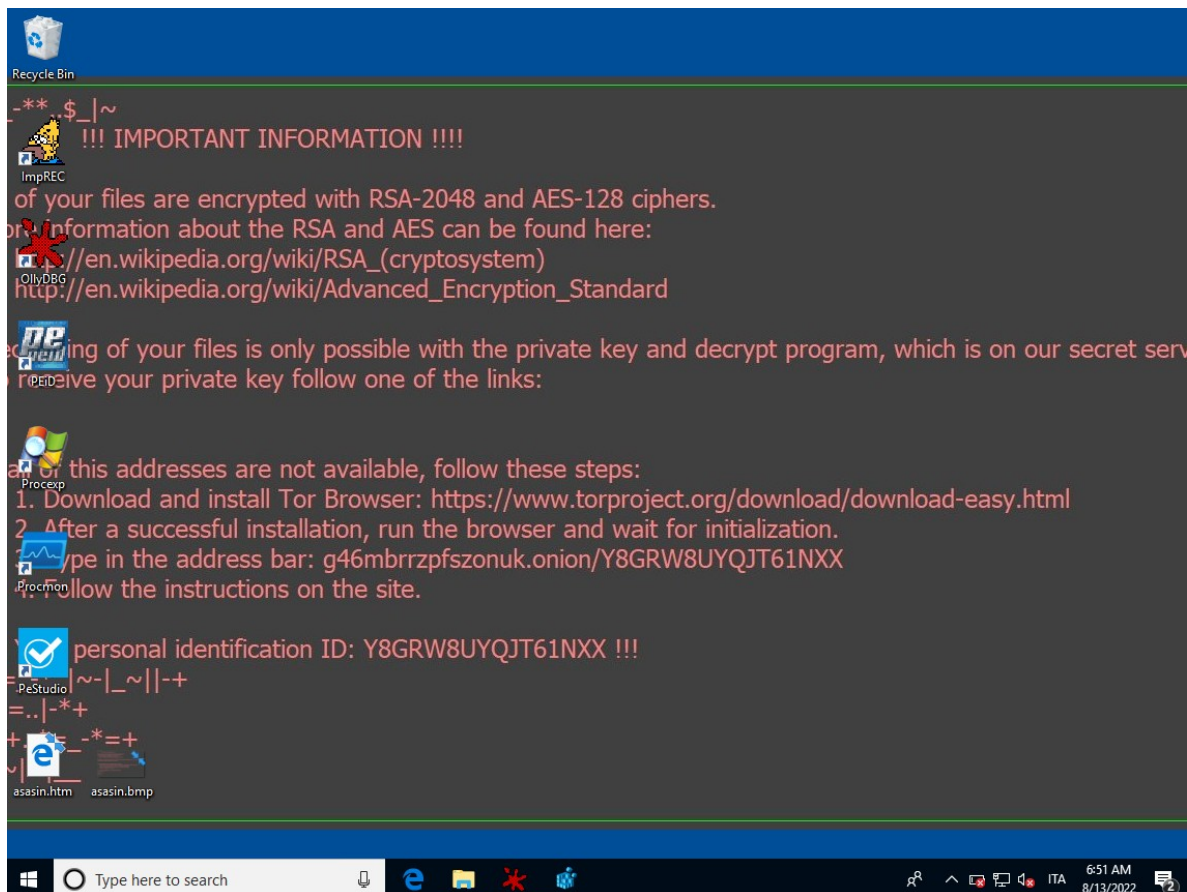
```

0019FB68  00000014  Action = SPI_SETDESKWALLPAPER
0019FB6C  00000000  wParam = 0
0019FB70  02294530  pParam = 02294530
0019FB74  00000003  UpdateProfile = 3.
0019FB78  0019FFDF

```

Address	Hex	dump	ASCII
02294530	43 00 3A 00 5C 00 55 00		C.:. \.
02294538	73 00 65 00 72 00 73 00		s.e.r.s.
02294540	5C 00 49 00 45 00 55 00		\.I.E.U.
02294548	73 00 65 00 72 00 5C 00		s.e.r.\.
02294550	44 00 65 00 73 00 68 00		D.e.s.k.
02294558	74 00 6F 00 70 00 5C 00		t.o.p.\.
02294560	61 00 73 00 61 00 73 00		a.s.a.s.
02294568	69 00 6E 00 2E 00 62 00		l.n...b.
02294570	6D 00 70 00 00 00 AD BA		n.p...ll

Verifichiamo che a seguito dell'esecuzione di tale API viene effettivamente modificato lo sfondo del desktop.



Successivamente, sempre nella FUN_00425870, viene chiamata l'API ShellExecuteW, che permette di eseguire un comando da shell. Tale API viene invocata due volte a 004260C5 e 004261C3; analizzando lo stack vediamo che tramite il comando open va ad aprire prima il file asasin.htm e poi il file asasin.bmp.

004260C5	> FFD6	CALL ESI	shell32.ShellExecuteW
004260C7	> 7E9 7D060000	JMP hw4.00426749	
004260CC	> C645 FC 0E	MOV BYTE PTR SS:[EBP-4],0E	
004260D0	. 90	NOP	
0019FB4C	0019FC38	UNICODE "open"	
0019FB50	02294530	UNICODE "C:\Users\IEUser\Desktop\asasin.bmp"	
0019FB54	00000000		
0019FB58	00000000		
0019FB5C	00000001		
0019FB60	00000000		
0019FB64	0019FC44	UNICODE "open"	
0019FB68	022949E8	UNICODE "C:\Users\IEUser\Desktop\asasin.htm"	

L'ultima API chiamata è la RegCloseKey, che va a chiudere la chiave di sistema precedentemente aperta prima di restituire il controllo al malware_main.

In definitiva, la FUN_00425870 implementa molti dei comportamenti osservati del malware, in quanto crea i file asasin.bmp e asasin.htm, modifica lo sfondo del desktop settando asasin.bmp, ed infine apre i due file creati in precedenza. Rinominiamo quindi questa funzione in malware_desktop_asasin.

5.3.12 Rimozione dell'eseguibile (malware_exe_remove)

Dopo l'esecuzione di `malware_desktop_asasin` si torna al `malware_main`, e proseguendo con l'analisi si giunge al termine del ciclo `do-while`. Il flusso di esecuzione riprende quindi dalla prima istruzione dopo il `do`, ovvero la chiamata a `malware_exe_remove` già analizzata in precedenza (5.3.3).

```
349     }
350     asasin_path((void *) (unaff_EBP + -0xac), '\x01', (void *) 0x0);
351 } while( true );
```

Al contrario di quanto detto in 5.3.3, `malware_exe_remove` non viene quindi invocata soltanto nel caso di lingua russa, ma anche dopo aver eseguito tutte le istruzioni della prima iterazione `do-while`.

```
44 if (((lang_id & 1023) != 25) && (lang_id = GetUserDefaultLangID(), (lang_id & 0x3ff) != 0x19)) &&
45     (lang_id = GetUserDefaultUILanguage(), (lang_id & 0x3ff) != 0x19)) goto not_russian;
46 do {
47     *(undefined4 *) (unaff_EBP + -4) = 0xffffffff;
48     malware_exe_remove();
49 not_russian:
```

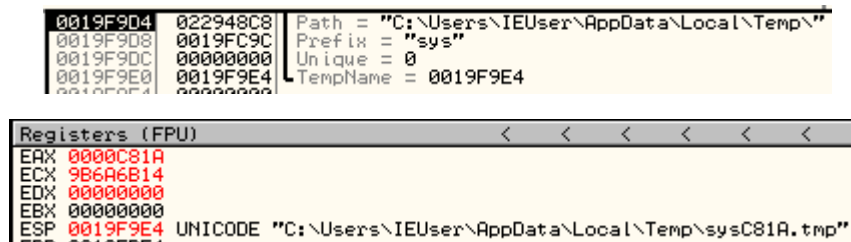
Quindi `malware_exe_remove` è l'effettiva *funzione in cui viene rimosso l'eseguibile* del malware, e viene invocata in due situazioni:

1. Prima di eseguire le azioni del malware, se nel sistema è selezionata la lingua russa.
2. Dopo aver eseguito il `malware_main`, se nel sistema non è selezionata la lingua russa.

Una analisi più approfondita di tale funzione mostra un particolare non rilevato in precedenza: prima di chiamare `FUN_0042FB40` viene recuperata la stringa `sys`, e poi si invoca `GetTempFileNameW`.

```
25 *(undefined2 *) (unaff_EBP + -0x14) = L's';
26 *(undefined2 *) (unaff_EBP + -0x12) = L'y';
27 *(undefined2 *) (unaff_EBP + -0x10) = L's';
28 *(undefined2 *) (unaff_EBP + -0xe) = 0;
29 pWVar4 = (LPCWSTR *) (unaff_EBP + -0x98);
30 if (*(uint *) (unaff_EBP + -0x84) < 8) {
31     pWVar4 = (LPCWSTR) (unaff_EBP + -0x98);
32 }
33 FUN_0042f4b0((void *) (unaff_EBP + -0x7c), pWVar4, (LPCWSTR) (unaff_EBP + -0x14), 0);
```

Proseguendo ora con *OllyDbg* vediamo che viene passato come parametro `prefix` proprio `sys`, andando a generare un file il cui casuale avrà sempre tale prefisso.



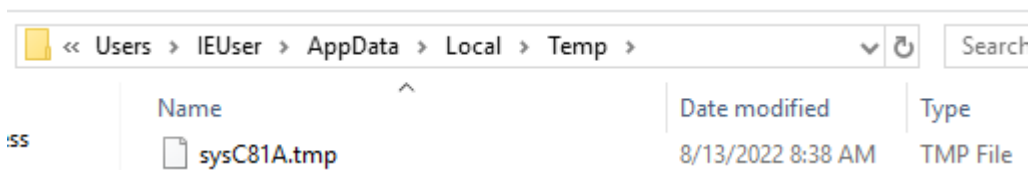
Come già visto i precedenza, viene chiamata la `run_process`, che tramite l'API `CreateProcessW` si occupa di eliminare l'eseguibile.

```

0019FB64 00000000 ModuleFileName = NULL
0019FB68 022949E8 CommandLine = "cmd.exe /C del /Q /F "Z:\Homework 4\hw4.ex_""
0019FB6C 00000000 pProcessSecurity = NULL
0019FB70 00000000 pThreadSecurity = NULL
0019FB74 00000000 InheritHandles = FALSE
0019FB78 00000050 CreationFlags = CREATE_NEW_CONSOLE|IDLE_PRIORITY_CLASS
0019FB7C 00000000 pEnvironment = NULL
0019FB80 00000000 CurrentDir = NULL
0019FB84 0019FB90 pStartupInfo = 0019FB90
0019FB88 0019FBD4 pProcessInfo = 0019FBD4
0019FB8C 00000000

```

Verifichiamo che il file temporaneo viene effettivamente creato, e l'eseguibile `hw4.ex_` eliminato a seguito di `malware_exe_remove`.



Infine viene chiamata `ExitProcess` passando come `ExitCode` il valore 0 terminando di fatto l'esecuzione del programma.

Con questo si è conclusa l'analisi del `malware_main`. Sono stati dunque analizzati quasi tutti i comportamenti osservati del malware, in particolare:

- `malware_desktop_asasin`:
 - Creazione dei file `asasin.htm` e `asasin.bmp`.
 - Apertura automatica di `asasin.htm` e `asasin.bmp`.
 - Impostazione di `asasin.bmp` come sfondo del desktop.
- `malware_exe_remove`:
 - Eliminazione del file `hw4.exe` lanciato.
 - Creazione di una copia dell'eseguibile in Temp.

A questo punto dobbiamo ancora individuare due dei comportamenti osservati del malware, e già descritti nell'introduzione:

- Effettiva *cifratura dei dati utente* in file `.asasin` tramite *RSA-2048* e *AES-128*.
- Creazione del file `asasin-xxxx.htm` nelle cartelle in cui vengono cifrati i file.

Molto probabilmente queste azioni sono implementate nelle routine `LAB_004768D0` e `LAB_00429B40` eseguite dai thread, già incontrate in [5.3.8](#) e [5.3.9](#) e di cui non è stato ancora esaminato il comportamento.

6 Analisi dei Thread Spawnati

Nella descrizione del `malware_main` sono state individuate le seguenti due funzioni eseguite dai due thread creati:

- LAB_004768D0
- LAB_00429B40

Anche in questo caso l'analisi viene effettuata utilizzando sia *Ghidra* che *OllyDbg*, mettendo un breakpoint all'indirizzo della funzione associata al thread, e poi eseguendo con F9 la relativa `CreateThread`.

6.1 Routine 1 (LAB_004768D0)

La LAB_004768D0 è una funzione non correttamente riconosciuta da Ghidra, quindi definiamo una funzione a quell'indirizzo e la rinominiamo in `thread_routine_1`.

```
2 void thread_routine_1(void)
3
4 {
5     int unaff_EBP;
6
7     ignore();
8     *(undefined **)(unaff_EBP + -0x10) = &stack0xffffffff0;
9     do {
10         *(undefined4 *)(unaff_EBP + -4) = 0;
11         FUN_00476150();
12         Sleep(2);
13     } while( true );
14 }
```

Analizzandone il codice vediamo che invoca ogni due secondi, in un ciclo infinito, la FUN_00476150.

6.1.1 FUN_00476150 (periodic_work)

Questa funzione invoca `GetModuleFileNameA`, con cui si ottiene il path "C:\Windows\SYSTEM32\ntdll.dll" tramite l'handle a `ntdll` specificato nel primo parametro.

0241F518	76EA0000	hModule = 76EA0000 (ntdll)
0241F51C	0241FD28	PathBuffer = 0241FD28
0241F520	00000208	BufSize = 208 (520.)
0241F524	00000000	
0241F528	00000000	

Successivamente viene chiamata la `CreateFileA` sul path ottenuto in precedenza, che in base ai flag specificati permette di aprire in lettura la libreria `ntdll.dll`.

0241F504	0241FD28	FileName = "C:\Windows\SYSTEM32\ntdll.dll"
0241F508	80000000	Access = GENERIC_READ
0241F50C	00000001	ShareMode = FILE_SHARE_READ
0241F510	00000000	oSecurity = NULL
0241F514	00000003	Mode = OPEN_EXISTING
0241F518	00000000	Attributes = 0
0241F51C	00000000	hTemplateFile = NULL

Proseguendo viene invocata l'API `GetCurrentProcessId`, ottenendo l'identificatore del processo corrente, per poi entrare in un ciclo `do-while` annidato. All'interno di questo ciclo vengono chiamate molte funzioni differenti, per cui passiamo all'analisi di quelle più interessanti.

- `FUN_00474C30`: utilizza `OpenProcess` e `CreateEventA`, per cui non sembra legata alla cifratura dei file.
- `FUN_004750E0`: utilizza `CreateThread` e `TerminateThread`, per cui spawna a sua volta un altro thread. Tuttavia analizzando la routine `LAB_00474ED0` eseguita dal nuovo thread non sembra esserci nessuna API interessante.
- `FUN_00475540`: utilizza `GetLogicalDriveStringsW` e `QueryDosDeviceW`. Queste API permettono di ottenere le unità valide del sistema, e recuperare informazioni sui nomi dei dispositivi MS-DOS. Anche in questo caso sembra non esserci nulla relativo alla cifratura dei files.

In definitiva la `FUN_00476150` eseguita dal primo thread non implementa la cifratura dei file effettuata dal ransomware, quindi per adesso ci limitiamo a rinominarla `periodic_work` e proseguiamo con l'analisi del secondo thread.

6.2 Routine 2 (LAB_00429B40)

Il secondo thread viene spawnato direttamente nel `malware_main`, subito dopo la chiamata a `priv_spawn_thread`. La `CreateThread` viene chiamata all'interno di un ciclo `for`, in cui viene passato ogni volta un valore differente di `lpParameter`.

```

207     for (lpParameter = *(LPVOID *) (unaff_EBP + -0xe0);
208         lpParameter != *(LPVOID *) (unaff_EBP + -0xdc);
209         lpParameter = (LPVOID)((int)lpParameter + 0x1c)) {
210     in_stack_ffffde4 = (undefined4 *)0x42a279;
211     pvVar4 = CreateThread((LPSECURITY_ATTRIBUTES)0x0,0,thread_routine_2,lpParameter,0,
212                          (LPDWORD)(unaff_EBP + -0x30));

```

Con il debugger vediamo che al primo thread viene passato il puntatore a `004244F8`, che contiene la stringa `'c:.'`.

0019FCB8	00000000	pSecurity = NULL
0019FCBC	00000000	StackSize = 0
0019FCC0	00429B40	ThreadFunction = hw4.00429B40
0019FCC4	022944F8	pThreadParm = 022944F8
0019FCC8	00000000	CreationFlags = 0
0019FCCC	0019FEAC	pThreadId = 0019FEAC

Address	Hex dump	ASCII
022944F8	63 00 3A 00 00 00 AD BA	c: : . . . #

Provando ad inserire un breakpoint sulla `CreateThread`, vediamo che non viene eseguita una seconda volta la chiamata ed il ransomware completa la sua esecuzione. Ciò vuol dire che di fatto viene spawnato un solo thread nel ciclo, passandogli come parametro `'c:.'`. Questo spiega anche perché non vengono cifrati i file sul volume remoto `Z://`, molto probabilmente viene spawnato un thread solo per ogni disco rigido o rimovibile.

Come fatto in precedenza, andiamo a definire su *Ghidra* la funzione `thread_routine_2` al posto della label `LAB_00429B40` e analizziamone il decompilato.

Vengono invocate le API `GetCurrentThread`, per ottenere un handle al thread corrente, e `SetThreadPriority`, che permette di modificare il livello di priorità di un thread. I parametri passati a quest'ultima sono l'handle del thread corrente, ed il livello di priorità che si vuole impostare, pari a `-2`. Ciò vuol dire che viene diminuita di 2 unità la priorità del thread corrente.

```
14 | iVar4 = -2;
15 | pvVar3 = GetCurrentThread();
16 | SetThreadPriority(pvVar3,iVar4);
```

Successivamente viene invocata nuovamente la `SetThreadPriority`, questa volta specificando priorità `0x100000` per settare la `THREAD_MODE_BACKGROUND_BEGIN`; in questo modo il sistema riduce la priorità del thread in modo che possa eseguire lavoro in background senza influire sulle attività in primo piano.

Successivamente vengono invocate diverse funzioni definite dallo sviluppatore, che dovranno essere analizzate singolarmente.

6.2.1 FUN_0046BF70 (files_lookup_outer)

La prima funzione invocata dopo aver impostato la priorità del thread è la `FUN_0046BF70`, che a sua volta invoca diverse funzioni al suo interno. La prima tra queste funzioni (nonché la più interessante) è la `FUN_0046B310`.

FUN_0046B310 (1)

Qui viene chiamata la `FindFirstFileW`, già incontrata in precedenza, e restituisce un handle al file specificato nel primo parametro. In questo caso si passa come `FileName` la stringa `"c:*"`. Osserviamo che viene utilizzata la wildcard `*` per cui è necessario che il thread abbia i permessi di accesso oltre che a `c:\`, anche a tutte le sottocartelle.

```
025DFC00 025DFE64 | FileName = "c:\*"
025DFC04 025DFC10 | pFindFileData = 025DFC10
```

Successivamente `FUN_0046B310` chiama `FUN_00462B10`, per cui passiamo a descrivere quest'ultima.

FUN_00462B10 (thread_privileges)

La funzione presenta un ciclo `while` in cui viene invocata la `GetFileSecurityW`, che permette di ottenere informazioni sulla sicurezza di un file o una directory. In particolare viene passato come primo parametro la stringa `'c:'`, per cui si richiedono informazioni sul disco rigido.

```
00462E2C > FF15 40A04700 CALL DWORD PTR DS:[47A040] advapi32.GetFileSecurityW
00462E32 . 90 NOP
025DFBA0 022944F8 | UNICODE "c:"
025DFBA4 00000007 | 00000000
025DFBA8 00000000 | 00000000
```

Continuando nel `while` viene fatto un controllo sul valore di ritorno dell'API, e si verifica anche l'ultimo errore tramite `GetLastError`; se il codice di errore è diverso da `0x7A` (`ERROR_INSUFFICIENT_BUFFER`) allora si esegue un `break`, uscendo il ciclo `while`.

```
ERROR_INSUFFICIENT_BUFFER
122 (0x7A)
L'area dati passata a una chiamata di sistema è troppo piccola.
```

Eseguendo con il debugger vediamo che la prima chiamata a `GetFileSecurity` ritorna 0, quindi fallisce e con errore proprio `ERROR_INSUFFICIENT_BUFFER`.

```
C 0 ES 002B 32bit 0(FFFFFFFF)
P 1 CS 0023 32bit 0(FFFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFFF)
Z 1 DS 002B 32bit 0(FFFFFFFF)
S 0 FS 0053 32bit 355000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
0 0 LastErr ERROR_INSUFFICIENT_BUFFER (0000007a)
```

Il ciclo quindi continua finché l'API non ritorna un valore diverso da 0, oppure l'errore è diverso da 0x7a. Ad ogni iterazione viene passata una stringa differente come primo parametro, andando ad esplorare tutte le sottocartelle a partire da `C:\`. Poiché il numero di iterazioni risulta eccessivamente elevato saltiamo il `while(true)`, tramite *run to selection* sull'istruzione successiva al ciclo.

```
Registers (MMX) < < <
EAX 02294588 UNICODE "c:\ProgramData\chocolatey\bin"
ECX 025DF2B4
```

Proseguendo vengono invocate le API `GetCurrentThread` e `OpenThreadToken`, per aprire il token di accesso del thread corrente. Tuttavia questa API fallisce, restituendo l'errore `ERROR_NO_TOKEN`.

```
58 | pvVar2 = GetCurrentThread();
59 | ret = (*OpenThreadToken)(pvVar2,uVar5,uVar6,ppvVar4);
60 | if (ret == 0) {
61 |     ppvVar4 = &local_8;
62 |     uVar5 = 0x2000e;
63 |     pvVar2 = GetCurrentProcess();
64 |     ret = (*OpenProcessToken)(pvVar2,uVar5,ppvVar4);
65 |     if (ret == 0) goto code_r0x00462bc7;
66 | }

EIP 00462C6F hw4.00462C6F
C 0 ES 002B 32bit 0(FFFFFFFF)
P 1 CS 0023 32bit 0(FFFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFFF)
Z 1 DS 002B 32bit 0(FFFFFFFF)
S 0 FS 0053 32bit 355000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
0 0 LastErr ERROR_NO_TOKEN (000003f0)
```

Poiché il valore di ritorno è 0 si entra nel ramo `if`, dove vengono invocate `GetCurrentProcess` e `OpenProcessToken`, già viste in precedenza. Il token di accesso viene questa volta acquisito con successo e viene restituito l'handle a tale token in `025DFBF8`.

```
025DFBA8 FFFFFFFF hProcess = FFFFFFFF
025DFBAC 0002000E DesiredAccess = STANDARD_RIGHTS_READ|TOKEN_DUPLICATE|TOKEN_IMPERSONATE|TOKEN_QUERY
025DFBB0 025DFBF8 phToken = 025DFBF8
```

Successivamente viene invocata `DuplicateToken`, che crea una copia dell'access token specificato tramite l'handle nel primo parametro. Se la copia viene creata con successo, si passa ad invocare `MapGenericMask` che mappa i diritti di accesso generici in una maschera di accesso, a diritti di accesso specifici e standard.

```
75 | local_1c = 0x14;
76 | local_2c = 0x120089;
77 | local_28 = 0x120116;
78 | local_24 = 0x1200a0;
79 | local_20 = 0x1f01ff;
80 | (*MapGenericMask)(&param_1,&local_2c);
```

Dalla documentazione vediamo che il secondo parametro è un puntatore ad una struttura `GENERIC_MAPPING`, composta dai seguenti campi:

```
C++ Copy
typedef struct _GENERIC_MAPPING {
    ACCESS_MASK GenericRead;
    ACCESS_MASK GenericWrite;
    ACCESS_MASK GenericExecute;
    ACCESS_MASK GenericAll;
} GENERIC_MAPPING;
```

Possiamo definire quindi questa struttura su *Ghidra*, a partire da `local_2c`.

Offset	Length	Mnemonic	DataType	Name
0	4	ACCESS_MASK	ACCESS_MASK	GenericRead
4	4	ACCESS_MASK	ACCESS_MASK	GenericWrite
8	4	ACCESS_MASK	ACCESS_MASK	GenericExecute
12	4	ACCESS_MASK	ACCESS_MASK	GenericAll

```

gen_map_struct.GenericRead = 0x120089;
gen_map_struct.GenericWrite = 0x120116;
gen_map_struct.GenericExecute = 0x1200a0;
gen_map_struct.GenericAll = 0x1f01ff;
(*MapGenericMask)(&param_1, &gen_map_struct);

```

Infine viene invocata l'API `AccessCheck`, che determina se un security descriptor concede un determinato insieme di diritti di accesso al client identificato dall'access token.

```

79 | ret = (*AccessCheck)(_Memory, local_c, param_1, &gen_map_struct, &local_40, &local_1c, &local_18);
80 | if (ret == 0) {
81 |     local_14 = 0;
82 | }

```

Il quinto parametro è un puntatore ad una struttura `PRIVILEGE_SET`, che anche in questo caso andiamo a definire come fatto in precedenza.

Offset	Length	Mnemonic	DataType	Name
0	4	DWORD	DWORD	PrivilegeCount
4	4	DWORD	DWORD	Control
8	12	LUID_AND_ATTRIBUTES	LUID_AND_ATTRIBUTES	Privilege

```

priv_set_struct.PrivilegeCount = 0;
priv_set_struct.Control = 0;
priv_set_struct.Privilege.Luid.LowPart = 0;
priv_set_struct.Privilege.Luid.HighPart = 0;
priv_set_struct.Privilege.Attributes = 0;

```

Da *OllyDbg* vediamo che questa API viene eseguita con successo (valore di ritorno 1). In definitiva `FUN_00462B10` ritorna un booleano che indica l'esito delle operazioni sui permessi del thread, per cui la rinominiamo `thread_privileges`.

FUN_0046B310 (files_lookup)

Tornando al chiamante `FUN_0046B310`, vediamo che viene utilizzata l'API `FindNextFileW`, che continua la ricerca di file e directory iniziata con `FindFirstFileW`; se la funzione esegue correttamente, il valore restituito è diverso da zero e il parametro `lpFindFileData` contiene informazioni sul file o sulla directory successiva trovata. Se la funzione ha invece esito negativo, il valore restituito è zero e il contenuto di `lpFindFileData` è indeterminato.

Questa API viene chiamata più volte all'interno del `do-while`, e ad ogni iterazione viene passato un handle ad un file differente. Il ciclo termina quando `FindNextFileW` fallisce e ritorna 0. Prima di invocare questa API vengono effettuati diversi controlli e chiamate a varie funzioni user defined.

La prima tra queste è `FUN_0043C160`, al cui interno sono definite ben 272 variabili locali e varie stringhe offuscate. In input questa funzione prende come primo parametro il nome della cartella correntemente analizzata, e tramite il debugger vediamo che le stringhe caricate corrispondono a nomi specifici di cartelle e 4 file relativi al ransomware.

025DF988	025DF9A8	UNICODE	"Windows"
025DF98C	025DF9E0	UNICODE	"Boot"
025DF990	025DFA4C	UNICODE	"System Volume Information"
025DF994	025DFB78	UNICODE	"\$Recycle.Bin"
025DF998	025DFB94	UNICODE	"thumbs.db"
025DF99C	025DFBD4	UNICODE	"temp"
025DF9A0	025DFB5C	UNICODE	"Program Files"
025DF9A4	025DFB10	UNICODE	"Program Files (x86)"
025DF9A8	025DFB88	UNICODE	"AppData"
025DF9AC	025DFB38	UNICODE	"Application Data"
025DF9B0	025DFBC8	UNICODE	"winnt"
025DF9B4	025DFBEC	UNICODE	"tmp"
025DF9B8	025DFA0C	UNICODE	"_Locky_recover_instructions.txt"
025DF9BC	025DF9CC	UNICODE	"_Locky_recover_instructions.bmp"
025DF9C0	025DFAE0	UNICODE	"_HELP_instructions.txt"
025DF9C4	025DFAB0	UNICODE	"_HELP_instructions.bmp"
025DF9C8	025DFA80	UNICODE	"_HELP_instructions.html"

Queste cartelle rappresentano probabilmente una **blacklist** delle directory in cui il ransomware non deve andare a cifrare i file, in quanto abbiamo già osservato nell'introduzione come *Windows* e *Program Files* non vengano di fatto attaccate.

Dopo aver ottenuto tutte queste stringhe viene eseguito un `while(true)` in cui si utilizza l'API `wcsstr` o `wsicmp` per verificare se il nome della cartella correntemente analizzata equivale ad una di quelle caricate in precedenza.

```
025DF974 025DFC3C| UNICODE "$Recycle.Bin"
025DF978 025DFB88| UNICODE "Windows"
025DF974 025DFC3C| UNICODE "$Recycle.Bin"
025DF978 025DFB78| UNICODE "$Recycle.Bin"
```

Se le cartelle sono uguali `wsicmp` ritorna 0 e `FUN_0043C160` ritorna 1 al chiamante, altrimenti un valore differente. Rinominiamo quindi questa in `check_blacklist_directory`.

Se si sta analizzando un file in una cartella consentita si entra nell'`if` successivo e viene chiamata la `FUN_0043AED0`. Questa ha un comportamento simile a `check_blacklist_directory`, poiché anche in questo caso vengono caricate in memoria molte stringhe, relative tuttavia a specifiche *estensioni* anziché cartelle.

025DCF04	025DF8AC	UNICODE ".py"	025DCF4C	025DF37C	UNICODE ".gif"
025DCF08	00000004		025DCF50	FFFFFFFFC	
025DCF0C	025DF3C4	UNICODE ".psd"	025DCF54	025DE8D8	UNICODE ".png"
025DCF10	00000001		025DCF58	FFFFFFFFC	
025DCF14	025DE6B0	UNICODE ".NEF"	025DCF5C	025DF364	UNICODE ".bmp"
025DCF18	FFFFFFFFE		025DCF60	FFFFFFFFC	
025DCF1C	025DE338	UNICODE ".tiff"	025DCF64	025DE698	UNICODE ".svg"
025DCF20	FFFFFFFFD		025DCF68	FFFFFFFFC	
025DCF24	025DF3AC	UNICODE ".tif"	025DCF6C	025DE320	UNICODE ".djvu"
025DCF28	FFFFFFFFD		025DCF70	FFFFFFFFC	
025DCF2C	025DE8F0	UNICODE ".jpg"	025DCF74	025DF34C	UNICODE ".djv"
025DCF30	FFFFFFFFD		025DCF78	FFFFFFFFC	
025DCF34	025DE11C	UNICODE ".jpeg"	025DCF7C	025DE8C0	UNICODE ".zip"
025DCF38	FFFFFFFFD		025DCF80	FFFFFFFF6	
025DCF3C	025DF394	UNICODE ".cgm"	025DCF84	025DF334	UNICODE ".rar"
025DCF40	FFFFFFFFD		025DCF88	FFFFFFFF6	
025DCF44	025DE488	UNICODE ".raw"	025DCF8C	025DF864	UNICODE ".7z"

Abbiamo già visto che `.jpg`, `.png` e `.rar` vengono tutti cifrati dal ransomware, per cui queste estensioni rappresentano una **whitelist** dei file da andare a cifrare, lasciando intatti ad esempio i file eseguibili `.exe`.

La funzione `FUN_0043AED0` infatti prende in input il nome del file attualmente considerato, ne ricava l'estensione, e tramite `wsicmp` verifica se tale estensione è uguale ad una tra quelle della *whitelist*. Rinominiamo quindi `FUN_0043AED0` in `check_whitelist_extension`.

```
025DCB08 025DF93C| UNICODE ".jpg"
025DCB0C 025DE8F0| UNICODE ".jpg"
```

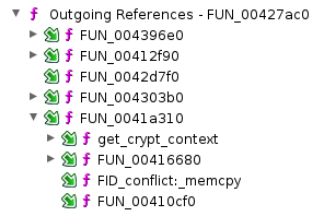
Successivamente vengono effettuate altre operazioni che non sembrano rilevanti, per invocare infine la `FindNextFileW` descritta in precedenza e continuare con la ricerca di files.

```
87 |code_r0x0046b3d6:
88 |    ret = FindNextFileW(*(HANDLE *) (unaff_EBP + -0x90), (LPWIN32_FIND_DATAW) (unaff_EBP + -0x2e0));
89 |    uVar7 = extraout_EDX_00;
90 |    } while (ret != 0);
```

Complessivamente quindi `FUN_0046B310` si occupa di individuare tutti i file che devono essere cifrati, utilizzando una *whitelist* per le estensioni ed una *blacklist* per le directory; la rinominiamo quindi `files_lookup`. Le altre funzioni `FUN_0046BF00`, `FUN_0046AC50` e `FUN_00429A70` invocate dopo `files_lookup` non risultano particolarmente interessanti, quindi rinominiamo `FUN_0046BF70` in `files_lookup_outer` e proseguiamo con l'analisi di `thread_routine_2`.

6.2.2 FUN_00427AC0 (ransomware_crypt_core)

La prima funzione invocata in `thread_routine_2` dopo `files_lookup_outer` è `FUN_00427AC0`. Analizzando innanzitutto il *function call trees* possiamo notare tra le moltissime funzioni invocate una chiamata a `get_crypt_context`, funzione definita da noi in precedenza che internamente chiama la `CryptAcquireContext`.



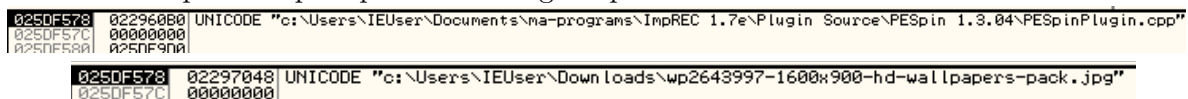
Ciò suggerisce che in `FUN_00427AC0` avvenga l'effettiva cifratura dei file, o comunque operazioni ad essa correlate, per cui proseguiamo con l'analisi dettagliata delle funzioni invocate al suo interno. La prima funzione invocata è `FUN_0041A310`, che come già visto invoca `get_crypt_context` e quindi `CryptAcquireContextA` con gli stessi parametri visti in precedenza. Viene poi chiamata la `CryptReleaseContext`.

Proseguendo viene invocata `FUN_0041B4A0` che effettua alcune allocazioni di memoria e non sembra contenere particolari funzioni per cifrare i file, quindi almeno per ora passiamo oltre.

Successivamente, all'interno di un ciclo `for`, viene chiamata la `FUN_00413BE0` che effettua a sua volta numerose chiamate a funzioni e API note come `CreateFileW`, `SetFileAttributesW`, `MoveFileExW`. Passiamo dunque ad analizzare tale funzione.

FUN_00413BE0 (aes_file_encrypt)

La prima API chiamata è `GetFileAttributesExW`, che permette di recuperare gli attributi per un file o una directory passata come primo parametro. Le informazioni ottenute verranno scritte nel buffer passato come terzo parametro. Dal codice decompilato vediamo che vengono effettuate diverse operazioni sui dati, quindi ci aiutiamo con il debugger per vedere nello specifico quali parametri vengono passati.



Nelle diverse chiamate a `GetFileAttributesExW` (all'interno del ciclo `for`) vengono passate stringhe di file differenti, quindi molto probabilmente i file che verranno cifrati.

Successivamente viene invocata due volte la `FUN_0040F410`, che prende come secondo parametro la stringa `0123456789ABCDEF` e tramite `memcpy` la scrive nella locazione di memoria specificata nel primo parametro. Rinominiamo `FUN_0040F410` in `string_copy`.

Proseguendo viene chiamata due volte `FUN_00412CA0`, che non sembra interessante, e poi la `FUN_00430450`, che utilizza internamente `MultiByteToWideChar`. Senza analizzarne l'implementazione nel dettaglio vediamo che viene chiamata tre volte, e ritorna due stringhe: `1NXX` e `QJ76`.

Vengono poi invocate più volte le funzioni FUN_00412F90, FUN_0040F9B0 e `asasin_path`, che complessivamente generano la stringa "Y8GRW8UY-QJT6-1NXX-62C0CA56-72559000AF40.asasin". Questa stringa ha il formato del nome di un file cifrato, per cui è molto probabilmente il nome di `PESpinPlugin.cpp` una volta che verrà colpito dal ransomware.

```
025DEF6C 02297D30 UNICODE "Y8GRW8UY-QJT6-1NXX-62C0CA56-72559000AF40.asasin"
025DEF70 FFFFFFF4
```

Rinominiamo quindi FUN_00430450, FUN_00412F90 e FUN_0040F9B0 in `gen_rans_name_1/2/3` e proseguiamo con l'analisi.

Le FUN_0042F660, FUN_00412C2, FUN_00413130 non sembrano per ora interessanti, mentre la FUN_00411DA0, invoca internamente `CreateFileW`, per cui ne studiamo il comportamento. Tramite il debugger vediamo che vengono passati i seguenti parametri.

```
025DF540 022960B0 FileName = "c:\Users\IEUser\Documents\na-programs\ImpREC 1.7e\Plugin Source\PESpin 1.3.04\PESpinPlugin.cpp"
025DF544 C0000000 Access = GENERIC_READ|GENERIC_WRITE
025DF548 00000004 ShareMode = 4
025DF54C 00000000 pSecurity = NULL
025DF550 00000003 Mode = OPEN_EXISTING
025DF554 00000000 Attributes = 0
025DF558 00000000 hTemplateFile = NULL
```

Il file viene aperto soltanto in lettura/scrittura soltanto se esiste, tramite `Mode = OPEN_EXISTING`. Il parametro `ShareMode=4` corrisponde a `FILE_SHARE_DELETE`, e abilita operazioni successive per l'eliminazione del file. Rinominiamo FUN_00411DA0 in `open_rans_target_file`.

Tornando a FUN_00413BE0, viene invocata l'API `MoveFileExW`; sempre con *OllyDbg* vediamo i parametri che vengono passati:

```
025DF570 022960B0 ExistingName = "c:\Users\IEUser\Documents\na-programs\ImpREC 1.7e\Plugin Source\PESpin 1.3.04\PESpinPlugin.cpp"
025DF57C 02297F50 NewName = "c:\Users\IEUser\Documents\na-programs\ImpREC 1.7e\Plugin Source\PESpin 1.3.04\Y8GRW8UY-QJT6-1NXX-62C0CA56-72559000AF40.asasin"
025DF580 00000009 Flags = REPLACE_EXISTING
025DF584 757C1390 KERNEL32.SetThreadPriority
```

Risulta evidente che con questa API si va a rinominare il file, cambiando il suo nome da quello originale a quello cifrato con estensione `.asasin` ottenuto in precedenza. Osserviamo comunque che il file seppur rinominato, ha ancora la stessa dimensione del file originale, per cui il contenuto non è stato ancora effettivamente cifrato.

<div> <div><< ImpREC 1.7e > Plugin Source > PESpin 1.3.04</div> <div>Search PESpin 1.3.04</div> </div>			
Name	Date modified	Type	Size
Y8GRW8UY-QJT6-1NXX-62C0CA56-7255...	6/17/2006 1:33 PM	ASASIN File	4 KB

Successivamente viene invocata FUN_00410D90, che contiene una `CryptGenRandom`; questa API genera dei bytes crittograficamente casuali, e prende in input un handle al CSP (ottenuto in precedenza con `get_crypt_context`), il numero di bytes casuali da generare, ed un buffer in cui verranno scritti i bytes generati. Il buffer passato come parametro si trova all'indirizzo 025DF6C4, e vediamo infatti che qui vengono scritti dei bytes casuali.

Address	Hex dump	ASCII
025DF6C4	67 77 34 64 9F 28 73 EB	gw4df(s\$
025DF6CC	26 C4 2E C6 67 20 4C A9	&- .Pg Lr
025DF6D4	00 00 00 00 00 00 00 00
025DF6DC	00 00 00 00 00 00 00 00

Rinominiamo FUN_00410D90 in `gen_crypt_bytes` e proseguiamo.

Continuando l'analisi viene invocata FUN_00410FE0, che internamente chiama CryptEncrypt. Questa API permette di cifrare i dati, utilizzando l'algoritmo specificato dal CSP nel primo parametro. Più nel dettaglio i parametri interessanti che si utilizzano sono:

- **hKey**: handle all'encryption key, ottenuta tramite CryptGenKey o CryptImportKey.
- **hHash**: handle ad un oggetto hash, ottenuto tramite CryptCreateHash.
- **pbData**: puntatore al buffer che contiene il plaintext da cifrare prima della chiamata, e che conterrà il ciphertext risultante dopo la chiamata.

025DF570	025DF6C4
025DF574	00000010
025DF578	00000100
025DF57C	00000000
025DF580	00000000

Il primo parametro passato a FUN_00410FE0 corrisponde al quinto parametro della CryptEncrypt, ovvero pbData. Quindi viene passato come buffer l'indirizzo dove sono stati scritti i bytes casuali generati in precedenza da gen_crypt_bytes. Allo stesso indirizzo 0025DF6C4 verranno quindi scritti quei bytes cifrati.

Address	Hex dump	ASCII
025DF6C4	67 77 34 64 9F 28 73 EB	gw4df(s\$
025DF6CC	26 C4 2E C6 67 20 4C A9	&- .fg L-
025DF6D4	00 00 00 00 00 00 00 00
025DF6DC	00 00 00 00 00 00 00 00

Address	Hex dump	ASCII
025DF6C4	2C F9 6D 71 F4 78 3E E0	.mqrx>α
025DF6CC	17 A1 E0 E8 D6 12 06 2D	!iα&pf&~
025DF6D4	34 0D 5C 3F B3 26 B8 EF	4? 8q n
025DF6DC	B3 48 C2 C0 6B 5D 36 EE	Hr'k]6e
025DF6E4	5E 27 66 5F 74 0F 34 C5	^f_t*4+
025DF6EC	4C C9 D2 65 7A 73 3E 5F	Lmres>
025DF6F4	05 06 63 E9 53 27 15 24	5oc8X's\$
025DF6FC	A8 6F FA 34 2B AF 5D 3E	do 4+>I>
025DF704	C4 9E A8 94 CB 35 8E D9	-h&8f5&f
025DF70C	A7 71 87 1B 11 FE 04 D5	9q\$+&♦f
025DF714	14 A9 C0 D1 30 A9 32 24	9rL0r2\$

Possiamo rinominare FUN_00410FE0 in enc_crypt_bytes.

Viene successivamente invocata la FUN_00473830. Come possiamo vedere dal decompilato vengono invocate molte volte le funzioni aesenc, che esegue una singola iterazione dell'algoritmo di cifratura AES, ed aesenclast, che esegue l'ultima iterazione di AES.

```

115 |         auVar11 = aesenc(auVar10,pauVar3[9]);
116 |         auVar10 = aesenc(auVar8,pauVar3[9]);
117 |         auVar8 = pauVar3[10];
118 |         auVar11 = aesenclast(auVar11,auVar8);
119 |         *pauVar5 = *pauVar5 ^ auVar11;
120 |         auVar8 = aesenclast(auVar10,auVar8);

```

Per vedere cosa viene effettivamente cifrato, analizziamo con OllyDbg quali sono i parametri passati a FUN_00473830. Il primo parametro è l'indirizzo 0025DF7C4, che contiene il nome del file corrente. Eseguendo la function call vediamo che viene cifrato proprio il filename originale.

Address	Hex dump	ASCII
025DF7C4	2A A1 1B D4 50 00 45 00	*i*P.E.
025DF7CC	53 00 70 00 69 00 6E 00	S.p.i.n.
025DF7D4	50 00 6C 00 75 00 67 00	P.l.u.g.
025DF7DC	69 00 6E 00 2E 00 63 00	i.n...c.
025DF7E4	70 00 70 00 00 00 00 00	p.p.....
025DF7EC	00 00 00 00 00 00 00 00

Address	Hex dump	ASCII
025DF7C4	D3 2D 75 6A 3E 11 4A 28	u-u3>4J(
025DF7CC	DE FF 02 18 4A 68 1F 63	! 0tJhWc
025DF7D4	E0 62 92 E8 ED 8D D3 2C	αbIEφ(α
025DF7DC	D5 0B 2B F8 DF 52 7F 96	f&+oR&u
025DF7E4	79 48 BC 87 16 61 60 9F	uHc_~>~
025DF7EC	82 1B 9D CD 47 12 01 D1	e+&B0T
025DF7F4	62 BC CF D0 23 C0 9A 05	2# (+&
025DF7FC	1E F8 76 E3 1F 1B 25 8D	ΔouW&Z\
025DF804	DC 5C 7B 12 51 65 7E 48	~\&0e~H
025DF80C	1E A0 34 8B 10 8C 11 D3	Δ&4i &Δ
025DF814	14 17 FF 38 BB 84 97 9B	9& 8q&+u
025DF81C	2A FE 38 E0 07 74 6A D2	:#8&~Jm
025DF824	F7 DF 85 42 E6 04 3E AE	~#1Bp&α<
025DF82C	48 E2 CB 87 4D 82 0A 93	Hf&CMe,6
025DF834	63 79 1C 16 D9 3F AE 9A	cuL_~?<u
025DF83C	9E FA 17 81 7F 98 1B A1	R_ 9u0u+i
025DF844	17 A0 FC C6 45 AD BC 23	Δ&1 FE4#
025DF84C	06 B9 DB 71 3E 5D 46 0E	:#8&~Jm
025DF854	48 E6 75 64 AE 5A 3E BA	9u0u~&2&u
025DF85C	D3 DF 42 AF 43 97 6F 71	u#B>Cu0q
025DF864	65 20 C9 89 09 2F F9 FE	e f&e./>#

In definitiva FUN_00473830 cifra tramite AES il contenuto del buffer passato come parametro, per cui possiamo rinominare questa funzione in `aes_encrypt`.

Possiamo ignorare FUN_004129D0, entrando direttamente nel ciclo `while` successivo. Qui vediamo che viene invocata FUN_004106E0, che chiama internamente la `ReadFile`. Tale API permette di leggere i dati da un file, e viene utilizzata per ottenere i dati relativi al file originale (finora soltanto rinominato ma non cifrato). I dati ricavati vengono scritti nel buffer all'indirizzo 02761020, passato come secondo parametro.

```
025DF558 000003A8 hFile = 000003A8 (window)
025DF55C 02761020 Buffer = 02761020
025DF560 00000FCD BytesToRead = FCD (4045.)
025DF564 025DF580 pBytesRead = 025DF580
025DF568 00000000 pOverlapped = NULL
025DF56C 025DF58C
```

Verifichiamo che viene effettivamente letto il contenuto originale del file `PEspinPlugin.cpp` attualmente considerato. Rinominiamo quindi FUN_004106E0 in `read_original_file`.

Address	Hex dump	ASCII
02761020	2F 2F 2F 2F 2F 2F 2F 2F	////////
02761028	2F 2F 2F 2F 2F 2F 2F 2F	////////
02761030	2F 2F 2F 2F 2F 2F 2F 2F	////////
02761038	2F 2F 2F 2F 2F 2F 2F 2F	////////
02761040	2F 2F 2F 2F 2F 2F 2F 2F	////////
02761048	2F 2F 2F 2F 2F 2F 2F 2F	////////
02761050	2F 2F 2F 2F 2F 2F 2F 2F	////////
02761058	2F 2F 2F 2F 2F 2F 2F 2F	////////
02761060	2F 2F 2F 2F 2F 2F 2F 2F	////////
02761068	2F 2F 2F 2F 2F 2F 2F 2F	////////
02761070	2F 2F 2F 2F 2F 2F 2F 2F	////////
02761078	2F 2F 00 0A 2F 2F 00 0A	.../...
02761080	2F 2F 00 0A 2F 2F 20 20	.../
02761088	20 50 45 53 70 69 6E 20	PEspin
02761090	5B 31 2E 33 2E 30 34 5D	[1.3.04]
02761098	20 41 50 49 20 54 72 61	API Tra
027610A0	63 65 72 20 70 6C 75 67	cer plug
027610A8	69 6E 20 66 6F 72 20 49	in for I
027610B0	6D 70 6F 72 74 52 65 63	mpportRec
027610B8	00 0A 2F 2F 20 20 20 20	.../
027610C0	00 0A 2F 2F 20 20 20 41	.../ A
027610C8	75 74 68 6F 72 20 3A 20	uthor :
027610D0	4E 61 67 61 72 65 73 68	Nagares
027610D8	77 21 73 3A 5A 3A 54 21	... U T

Subito dopo aver caricato in memoria il contenuto del file viene invocata di nuovo la funzione `aes_encrypt`, proprio per andare a cifrarne il contenuto. Vediamo che come buffer viene passato proprio il puntatore 02761020, e che a seguito della chiamata il contenuto allo stesso indirizzo risulta infatti cifrato.

Address	Hex dump	ASCII
02761020	2F 2F 2F 2F 2F 2F 2F 2F	////////
02761028	2F 2F 2F 2F 2F 2F 2F 2F	////////
02761030	2F 2F 2F 2F 2F 2F 2F 2F	////////
02761038	2F 2F 2F 2F 2F 2F 2F 2F	////////
02761040	2F 2F 2F 2F 2F 2F 2F 2F	////////
02761048	2F 2F 2F 2F 2F 2F 2F 2F	////////
02761050	2F 2F 2F 2F 2F 2F 2F 2F	////////
02761058	2F 2F 2F 2F 2F 2F 2F 2F	////////
02761060	2F 2F 2F 2F 2F 2F 2F 2F	////////
02761068	2F 2F 2F 2F 2F 2F 2F 2F	////////
02761070	2F 2F 2F 2F 2F 2F 2F 2F	////////
02761078	2F 2F 00 0A 2F 2F 00 0A	.../...
02761080	2F 2F 00 0A 2F 2F 20 20	.../
02761088	20 50 45 53 70 69 6E 20	PEspin
02761090	5B 31 2E 33 2E 30 34 5D	[1.3.04]
02761098	20 41 50 49 20 54 72 61	API Tra
027610A0	63 65 72 20 70 6C 75 67	cer plug
027610A8	69 6E 20 66 6F 72 20 49	in for I
027610B0	6D 70 6F 72 74 52 65 63	mpportRec
027610B8	00 0A 2F 2F 20 20 20 20	.../
027610C0	00 0A 2F 2F 20 20 20 41	.../ A
027610C8	75 74 68 6F 72 20 3A 20	uthor :
027610D0	4E 61 67 61 72 65 73 68	Nagares
027610D8	77 21 73 3A 5A 3A 54 21	... U T

→

Address	Hex dump	ASCII
02761020	F7 F3 F2 F9 FA 30 47 F4	??s2..0Gf
02761028	A2 13 0D 3F 25 8A 49 C4	o!!l ?%e1-
02761030	18 4C FF 57 90 EC 7A 61	tl Weoza
02761038	9C 0B 05 76 73 AC 6E A4	8 fvs%an
02761040	B3 7F B0 93 7A DF 64 73	l0%0zds
02761048	02 C0 8C 67 96 AF C8 2A	0Lq0u%*
02761050	04 58 55 7C 63 27 BC 9E	0XUic%*A
02761058	26 08 F8 0F 24 99 F6 FE	80%*50+*
02761060	FD CE BA 25 9D 3C F6 C7	2fll%*<-lt
02761068	F3 D5 B8 A2 68 94 E3 23	sf 0h0m#
02761070	BA 84 A6 80 BB 65 08 B3	llaa0le0l
02761078	0C BE 09 20 A7 19 66 CD	. . 00f=
02761080	67 86 E5 08 18 B4 41 30	g00q+tlA=
02761088	3B 15 75 89 82 D6 41 D6	:Su0emAr
02761090	4C 21 88 EE D0 28 18 26	L+0e0(+%
02761098	E9 D3 6D C2 D6 BF 71 3C	04mrm q<
027610A0	FE F5 79 71 AE ED C4 2E	mJyq00-
027610A8	4D 88 B0 D3 37 89 D6 B9	Me%47eml
027610B0	3A 51 F3 01 9F 02 05 EB	:00f0+3
027610B8	85 45 B2 E2 43 C3 88 C9	0E0fC0f
027610C0	B6 09 09 21 54 DE 96 86	ll...tTl00
027610C8	12 02 4B 8D 7A FA 27 3C	00Kiz-?<

Ora non resta che individuare la porzione di codice in cui il contenuto del file cifrato (presente a 02761020) viene scritto sul file Y8GRW8UY-QJ76-1NXX-62C0CA56-72559000AF40.asasin creato in precedenza.

Nella FUN_004107E0 si utilizza `WriteFile`, che permette di scrivere dati su un file. I parametri principali sono:

- `hFile`: handle al file o dispositivo di I/O su cui scrivere.
- `lpBuffer`: puntatore al buffer contenente i dati da scrivere.

Come atteso viene passato l'handle al file `.asasin` come primo parametro, e il puntatore al buffer con i dati cifrati come secondo parametro.

025DF558	000003A8	hFile = 000003A8 (window)
025DF55C	0276D020	Buffer = 0276D020
025DF560	00000FCD	nBytesToWrite = FCD (4045.)
025DF564	025DF580	pBytesWritten = 025DF580
025DF568	00000000	pOverlapped = NULL

Come possiamo verificare, il contenuto del file è stato effettivamente modificato, andando a scrivere il contenuto precedentemente cifrato tramite algoritmo AES.

Concludendo, FUN_00413BE0 cifra il contenuto del file attualmente aperto per cui la rinominiamo `aes_file_encrypt`. Vediamo che successivamente vengono chiamate altre funzioni che effettuano operazioni di minore rilevanza:

- `GetSystemTimeAsFileTime`: restituisce la data e l'ora di sistema.
- FUN_00410AD0: utilizza `SetFileTime` per modificare la data e l'ora in cui il file è stato creato, modificato o acceduto. Rinominata `set_filetime`.
- FUN_004108B0: utilizza `FlushFileBuffers`, che prende come unico parametro l'handle al file aperto e flusha tutti i buffer proprio sul file. Rinominata `flush_file`.
- `CloseHandle`: chiude l'handle al file aperto.
- Il controllo torna infine al chiamante FUN_00427AC0, che esegue una nuova chiamata di FUN_00413BE0 andando a cifrare un nuovo file.

E' stato quindi individuato ed analizzato il blocco di codice in cui viene cifrato il file, e resta da individuare l'ultimo comportamento del ransomware, ovvero la creazione del file `asasin-xxxx.htm` nelle directory dove sono stati individuati file da infettare.

Creazione dei file asasin-xxxx.htm

Tornando a FUN_00427AC0, cerchiamo tramite *function call trees* una chiamata a `CreateFile` o simili, per individuare rapidamente dove viene creato il file `.htm`. Vediamo che viene invocata la funzione `create_asasin_file` (FUN_0042E7D0) già analizzata precedentemente, dopo aver caricato in memoria proprio la stringa `' .htm'`.

```
94 | *(__int64 *)(&__vrbp - 0x20) = L'.'; | 119 | asasin_path((void *)(&__vrbp - 0x16c), '\x01', (void *)0x0); |
95 | *(__int64 *)(&__vrbp - 0x1e) = L'h'; | 120 | *(__int64 *)(&__vrbp - 4) = 0x11; |
96 | *(__int64 *)(&__vrbp - 0x1c) = L't'; | 121 | asasin_path((void *)(&__vrbp - 200), '\x01', (void *)0x0); |
97 | *(__int64 *)(&__vrbp - 0x1a) = L'm'; | 122 | create_asasin_file(); |
| | 123 | asasin_path((void *)(&__vrbp - 0xac), '\x01', (void *)0x0); |
```

Spostandoci di nuovo sul debugger analizziamo i parametri che vengono passati alla `CreateFileW`.

```
0250FC5C 02299750 FileName = "C:\Users\IEUser\Documents\ma-programs\ImpREC 1.7e\Plugin Source\PESpin 1.3.04\asasin-2cce.htm"
0250FC60 C0000000 Access = GENERIC_READ|GENERIC_WRITE
0250FC64 00000000 ShareMode = 0
0250FC68 00000000 pSecurity = NULL
0250FC6C 00000002 Mode = CREATE_ALWAYS
0250FC70 00000000 Attributes = 0
0250FC74 00000000 hTemplateFile = NULL
```

Vediamo che effettivamente viene specificato come `FileName` proprio il file `asasin-2cce.htm` nella directory del file che è stato appena cifrato.

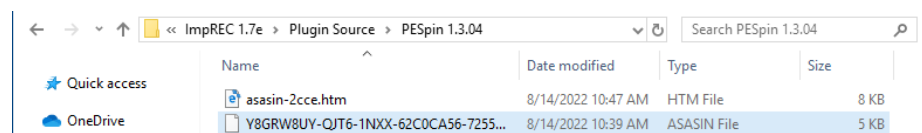
Successivamente sempre in `create_asasin_file` viene chiamata `write_enc_file` per scrivere il contenuto del file. Questa come sappiamo utilizza internamente `WriteFile`, per cui analizziamo i parametri che vengono passati.

```
0250FC4C 000003A8 hFile = 000003A8 (window)
0250FC50 02292520 Buffer = 02292520
0250FC54 00001FBC nBytesToWrite = 1FBC (8124.)
0250FC58 0250FC74 pBytesWritten = 0250FC74
0250FC5C 00000000 pOverlapped = NULL
```

Il buffer specificato contiene il codice `html` che verrà scritto nel file `.htm` creato in precedenza.

Address	Hex dump	ASCII
02292520	3C 68 74 6D 6C 20 63 6C	<html cl
02292528	61 73 73 3D 27 77 68 6C	ass='whl
02292530	6D 72 79 71 27 20 64 61	nyq' da
02292538	74 61 2D 61 71 64 6D 6E	ta=agdm
02292540	75 62 3D 27 75 6F 6C 79	ub='uoly
02292548	76 76 68 64 27 3E 3C 68	uvhd'><h
02292550	65 61 64 3E 0A 3C 6D 65	ead>.<me
02292558	74 61 2D 63 68 61 72 73	ta chars
02292560	65 74 3D 27 75 74 66 2D	et='utf-
02292568	38 27 3E 0A 3C 73 74 79	8'>.<sty
02292570	6C 65 3E 2E 78 71 76 63	le>.<quc
02292578	75 77 2D 7B 63 6F 6C 6F	uw fcolo
02292580	72 3A 2D 23 64 65 64 65	z: &dede
02292588	64 65 3E 6C 65 66 74 2D	de:left
02292590	3A 2D 2D 39 31 30 70 78	: -910px
02292598	3B 70 6F 73 69 74 69 6F	;positio
022925A0	6E 2D 3A 2D 61 62 73 6F	n : abso

Verifichiamo che il file `asasin-2cce.htm` è stato correttamente creato e contiene la pagina `html` vista nell'introduzione.



Con questo abbiamo analizzato anche l'ultimo comportamento osservato del ransomware, e possiamo concludere così l'analisi dell'eseguibile. Per completezza rinominiamo infine FUN_00427AC0 in `ransomware_crypt_core`, in quanto implementa tutte le funzionalità di cifratura del ransomware.

7 Conclusioni

Il malware analizzato viene immediatamente riconosciuto da windows come un ransomware della famiglia **Locky A**. L'eseguibile è impacchettato con UPX, per cui è stato necessario trovare l'OEP con una ricerca manuale tramite debugger. Una volta individuato l'OEP è stata ricostruita l'IAT tramite la prima euristica di *OllyDump*, in modo da avere a disposizione il codice decompilato in chiaro e poter sfruttare sia *Ghidra* che *OllyDbg*. Si è individuato innanzitutto il main del programma, e poi è stato bypassato un meccanismo di anti debug implementato tramite il campo **BeingDebugged** del PEB. Successivamente si è individuato il cosiddetto **malware_main**, a partire da cui vengono invocate le principali funzioni del malware. Tramite un controllo sulla lingua di sistema, se si utilizza il russo il malware termina e l'eseguibile viene eliminato prima ancora di infettare il sistema; ciò significa che nei *sistemi con lingua russa il ransomware è innocuo*.

Per tutte le altre lingue invece l'esecuzione continua. Il malware ottiene innanzitutto informazioni sui dischi presenti nel sistema, tramite **get_drives_info**. Successivamente tramite **priv_spawn_thread** il processo effettua la privilege escalation, e procede a spawnare un thread che va ad eseguire la **thread_routine_1**; questo lavora in background utilizzando la libreria **ntdll.dll**.

Successivamente viene spawnato un altro thread che esegue **thread_routine_2**. Anche questo thread lavora in background, ed implementa la funzionalità principale del malware, ovvero quella di cifratura dei file utilizzando AES. In particolare tramite **files_lookup** effettua individua tutti i file nel disco **C:** che devono essere cifrati, sfruttando una blacklist di directory ed una whitelist di estensioni. I file target vengono rinominati in **.asasin**, e viene sovrascritto il contenuto originale con quello cifrato. Infine il thread crea un file **.htm** in ogni directory che ha colpito.

Mentre i file vengono cifrati, continua in parallelo l'esecuzione del **malware_main**. Questo si occupa di generare i file **asasin.htm** e **asasin.bmp** sul desktop, per poi andare a modificare lo sfondo con la **SystemParameterInfoW**. Infine vengono aperti automaticamente i due file **asasin** creati sul desktop, e viene rimosso l'eseguibile del malware tramite **malware_exe_remove**.