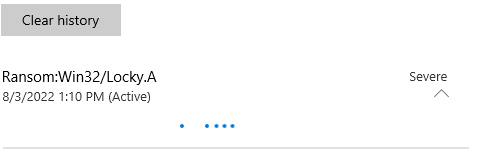
Malware Analysis Homework 4

Danilo Dell’Orco

0300229

# 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**.

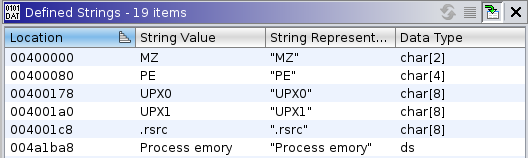
Creiamo una nuova istantanea della macchina, disabilitiamo Windows Defender e proseguiamo con l’analisi dell’eseguibile.

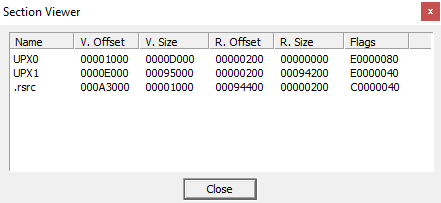
# Analisi di Base

## Analisi Statica di Base

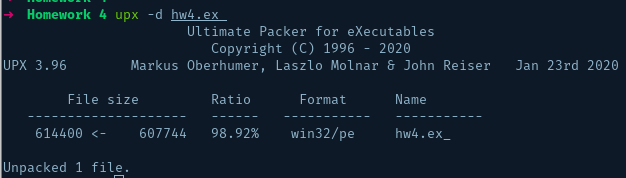
Effettuiamo innanzitutto un’*analisi statica delle stringhe*, sfruttando lo strumento *Defined Strings* di Ghidra. Dalla lista, vediamo sono presenti le seguenti stringhe:

* UPX0
* UPX1

Questo ci suggerisce che l’eseguibile è impacchettato tramite il **packer UPX**, come ci viene confermato anche eseguendo il comando file hw4.ex\_.

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

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 spacchettato correttamente.

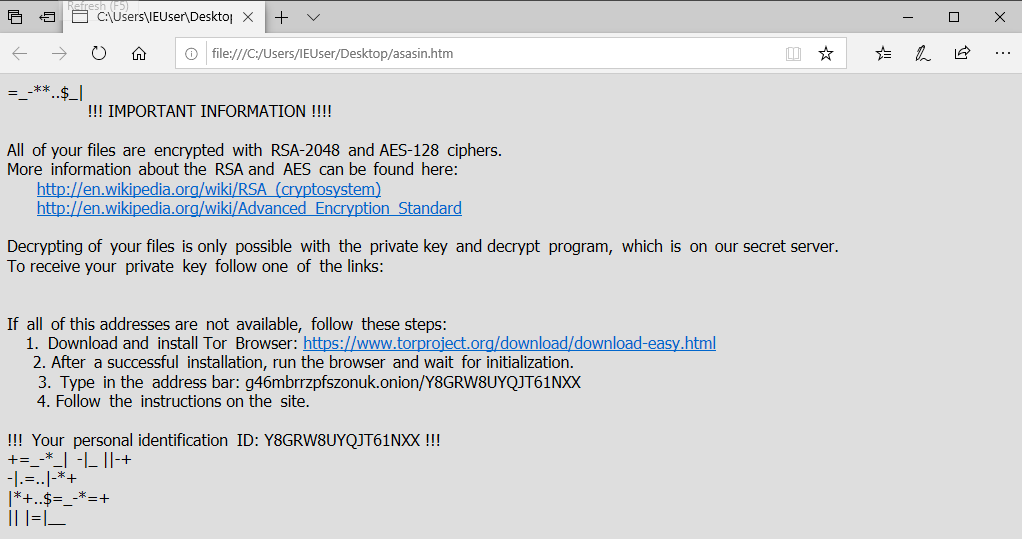
Infatti importando il nuovo eseguibile su *Ghidra* la situazione resta invariata rispetto a prima, e nessuna funzione risulta visibile in chiaro; risulta quindi necessario effettuare l’unpacking dell’eseguibile sfruttando altre tecniche.

## Analisi Dinamica di Base

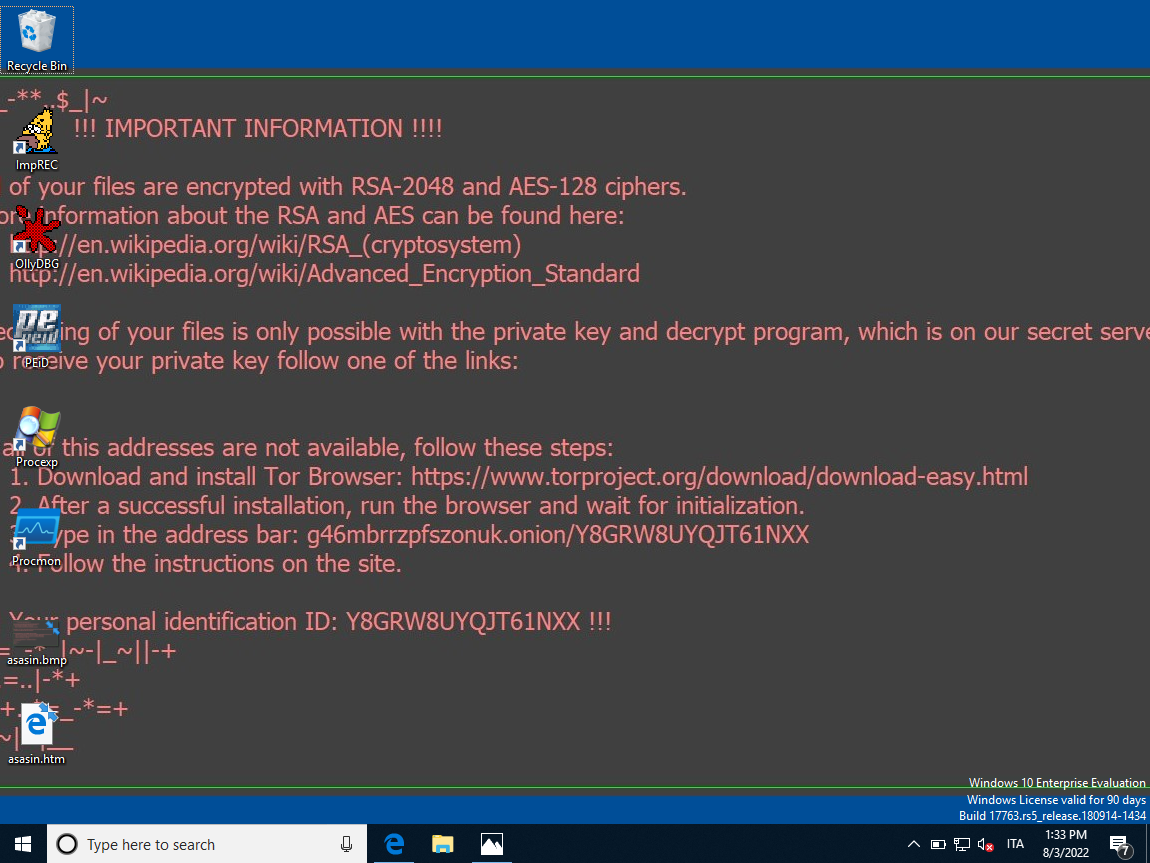
Prima di proseguire con l’unpacking dell’eseguibile effettuiamo l’*analisi dinamica di base*, in modo tale da ricavare il maggior numero di informazioni possibili sul comportamento del ransomware.

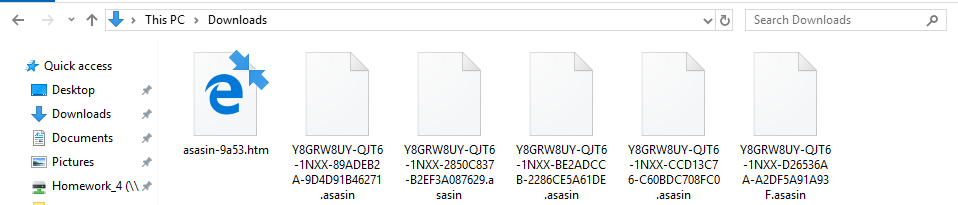
Per avere un’idea generale su cosa faccia effettivamente il malware sul sistema rinominiamo hw4.ex\_ in hw4.exe e **lanciamo l’eseguibile**.

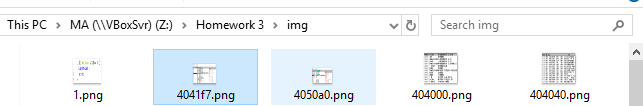
Dopo un certo tempo di attesa, vediamo che vengono aperte automaticamente una pagina web e un’immagine; entrambe riportano un messaggio, informando l’utente che tutti i suoi file sono stati **cifrati** tramite gli algoritmi di cifratura *RSA-2048* e *AES-128.* Viene inoltre indicato il procedimento necessario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

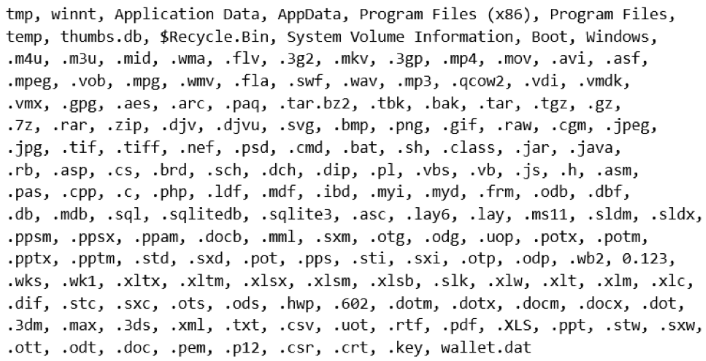
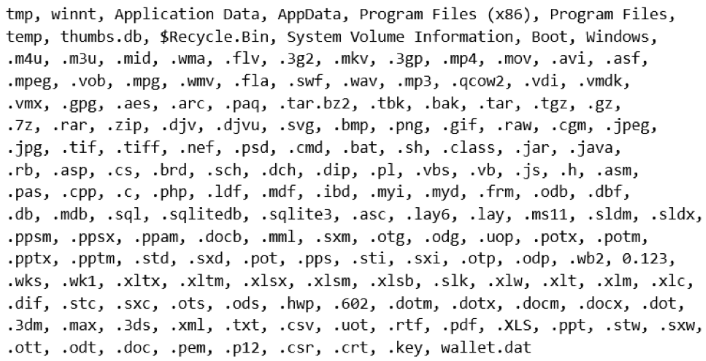


Possiamo inoltre vedere che viene modificato lo sfondo del *Desktop*, impostando la stessa immagine mostrata in precedenza.

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://](../../../../../../../Z://) non sono stati toccati in alcun modo.

Possiamo inoltre osservare che i file eseguibili, anche se sul disco [C:\](../../../../../../../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.

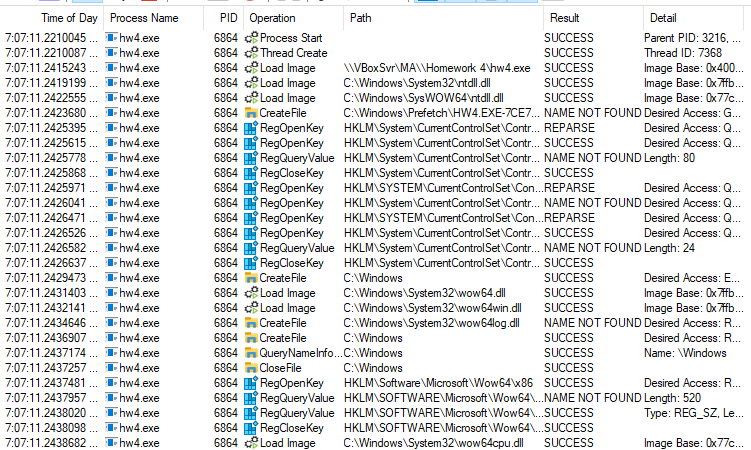
Anche i file in cartelle critiche del sistema come Windows e Program Files non sembrano essere attaccate in nessun modo, e tutti i software installati nel sistema mantengono la loro operatività.

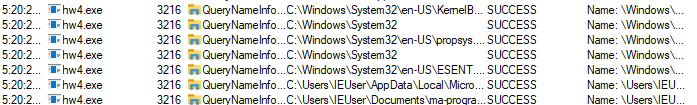
Infine è possibile vedere che l’eseguibile hw4.exe precedentemente lanciato è stato automaticamente eliminato al termine dell’esecuzione del ransomware.

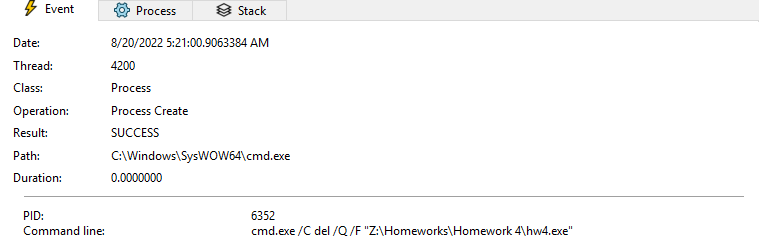
### Process Monitor

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.

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.

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.



### Informazioni Ricavate

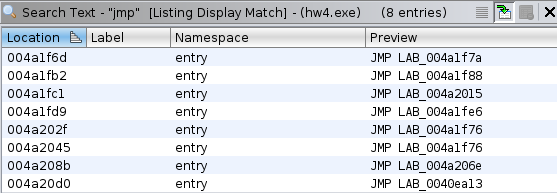
Riassumendo, tramite l’analisi dinamica sono state individuate le seguenti azioni che il ransomware effettua:

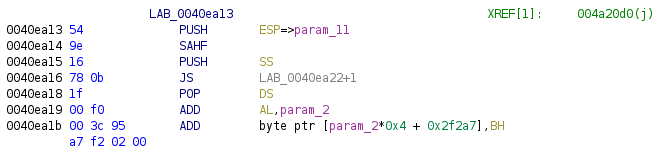
* Creazione dei file asasin.htm e asasin.bmp sul desktop.
* Apertura automatica di asasin.htm e asasin.bmp.
* Impostazione di asasin.bmp come sfondo del desktop.
* Cifratura dei dati utente tramite gli algoritmi *RSA-2048* e *AES-128*. Non vengono cifrati invece i file sul disco remoto Z:\.
* Creazione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

# Unpacking

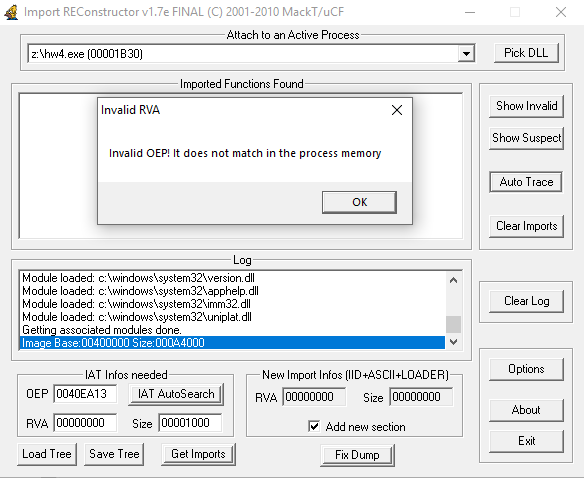
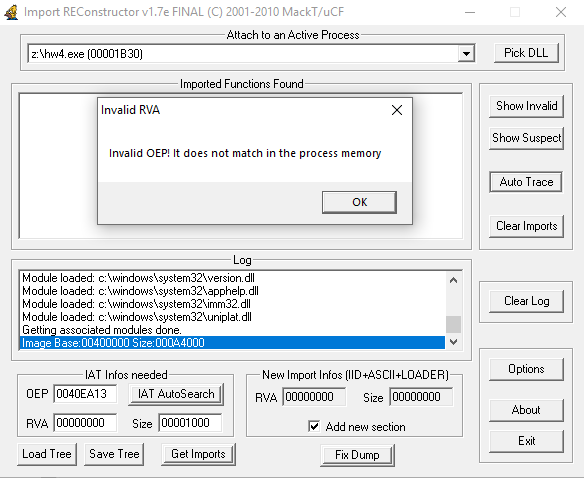
Per effettuare l’unpacking dell’eseguibile è necessario individuare innanzitutto l’**Original Entry Point** *(OEP)*. A tale scopo cerchiamo innanzitutto istruzioni di *long jump*, che possono rappresentare un salto verso l’OEP.

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.

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.

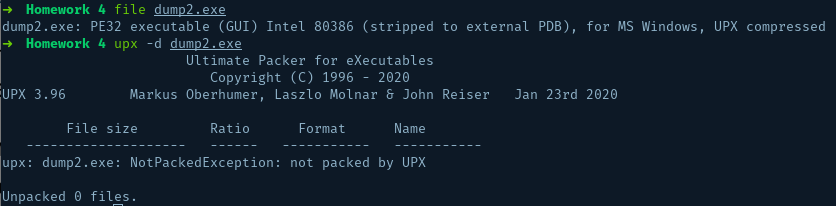
Ad ulteriore conferma di ciò, proviamo ad utilizzare **OllyDbg** e **ImpRec** per ricavare la **Import Address Table** (IAT).

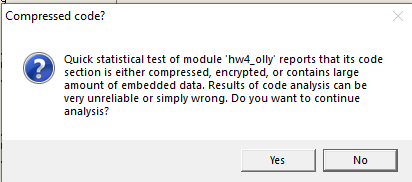
Da *OllyDbg* apriamo l’eseguibile, inseriamo un breakpoint all’indirizzo *0040ea13*, ed seguiamo il programma; quando il flusso di esecuzione raggiunge il breakpoint procediamo con il memory dump tramite il plugin *OllyDump*. Se tale indirizzo fosse l’OEP, allora tramite debugger sarebbe stato eseguito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

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.

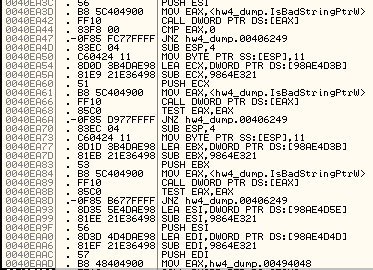
## Unpacking manuale con OllyDbg

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

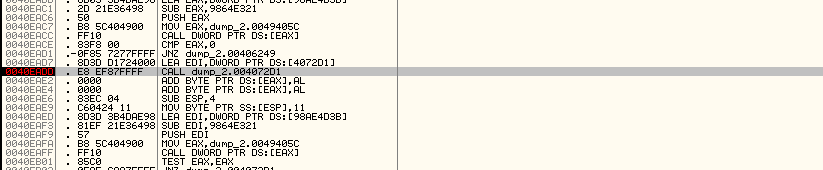
Ritorniamo quindi su *OllyDbg* per analizzare il nuovo dump2.exe. Aprendo l’eseguibile ci viene mostrato un messaggio secondo cui il programma presenta una *code section* compressa.

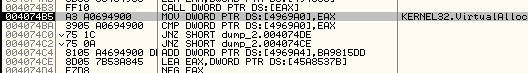
Proseguiamo comunque con l’analisi, partendo dall’indirizzo 0040EA13, ed eseguendo singolarmente le singole istruzioni macchina. All’indirizzo 0040ea42 viene invocata l’API IsBadStringPtrW, che prende in input un puntatore alla stringa ASCII “lwpqabklbeiutlcti” (puntata dal registro ESI), e verifica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

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

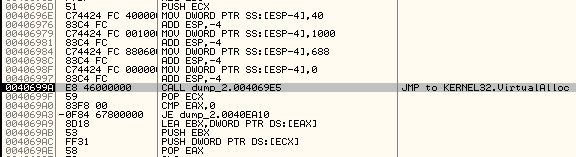


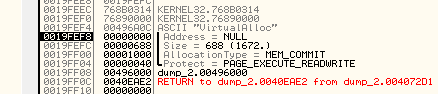
Procedendo con l’analisi, osserviamo all’indirizzo 0040EADD una CALL 004072D1. Tramite *step into* passiamo alla funzione chiamata.

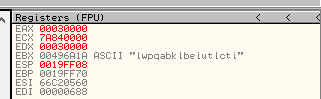
Qui vengono eseguite ulteriori chiamate a IsBadStringPtrW, e viene caricato sullo stack il puntatore alla funzione di libreria VirtualAlloc.

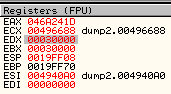
Successivamente si effettua un jump all’indirizzo 00406958, puntato da DS[4969A4].

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

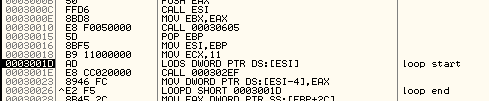
In particolare i parametri passati sono i seguenti.

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 (00030000[[1]](#footnote-2)), viene ritornato dalla funzione in EAX.

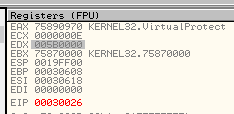
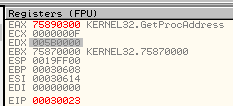
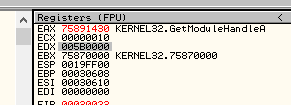
Successivamente vengono eseguite una serie di istruzioni per scrivere sulla nuova area allocata, il cui puntatore viene copiato sul registro EBX, e all’indirizzo 004069DA si effettua la CALL EBX, accedendo quindi al nuovo blocco di istruzioni appena scritto in 00030000.



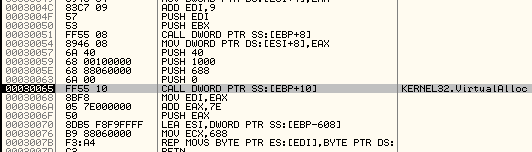
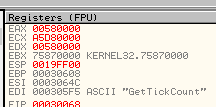
All’indirizzo 0003001D inizia un ciclo con le seguenti istruzioni

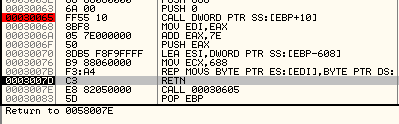
Viene effettuata una CALL 000302EF; analizzando il contenuto dei registri vediamo che questa ritorna in EAX un puntatore ad una funzione della libreria Kernel32.dll.

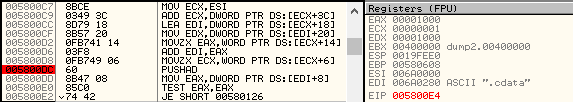
Successivamente il puntatore a tale funzione viene copiato tramite MOV nell’indirizzo puntato da DS:[ESI-4]. Eseguendo le diverse iterazioni del ciclo si può vedere che viene caricata ogni volta una libreria differente.

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**.

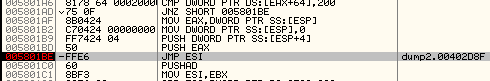
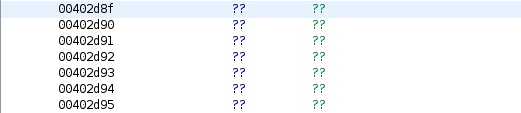
A 00030065 viene eseguita una nuova VirtualAlloc. Vengono allocati indirizzi a partire da 00580000 ed il puntatore a tale area viene spostato su EDI.

Tramite RETN si jumpa alla nuova area allocata.

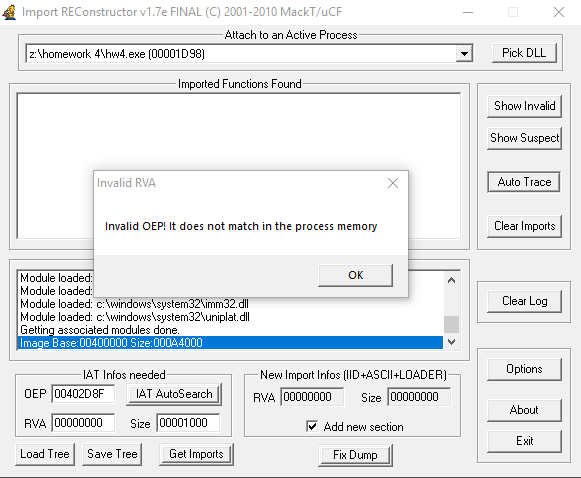
In particolare qui all’indirizzo 005800DC c’è un PUSHAD, vediamo che inizia un ciclo in cui vengono scritte diverse sezioni del programma nel registro EDI.

Nell’ordine vengono copiate in EDI le stringhe “.text”, “.rdata”, “.data”, “.reloc”, “.cdata”, che indicano le sezioni dell’address space dell’eseguibile *non-packed*;

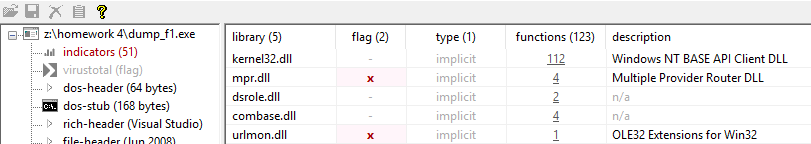
Invece all’indirizzo 005801BE viene eseguita una JMP ESI che porta all’indirizzo 00402D8F.

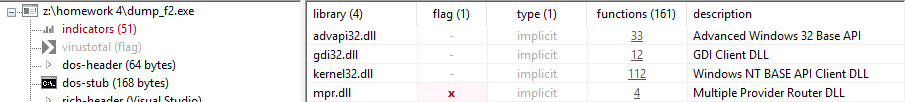
Questo è molto probabilmente il *tail jump* cercato. Infatti analizzando da *Ghidra* l’eseguibile originale, l’area a tale indirizzo risulta vuota (a differenza del primo OEP individuato).

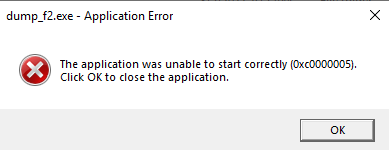
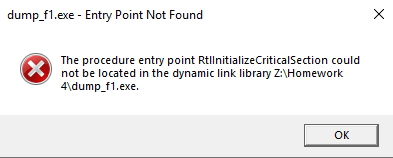
Eseguiamo quindi il *tail jump* tramite *step into* ed effettuaiamo il dump tramite *OllyDump*. Testiamo il funzionamento con entrambe le euristiche offerte dal plugin, generando due eseguibili dump\_f1.exe e dump\_f2.exe. Proviamo a ricostruire la IAT tramite *ImpREC*, ma anche in questo caso non è stato riconosciuto l’OEP individuato.

Aprendo 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:

Nel dump prodotto con il secondo metodo:

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 spacchettato correttamente eseguibile, in quanto non è stata ripristinata al 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.

# 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.

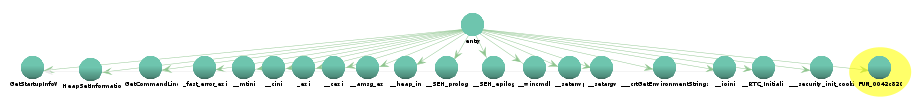
Tra le API più interessanti troviamo:

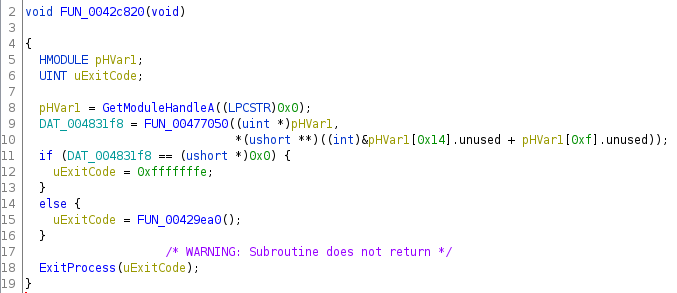
* CreateThread / TerminateThread, quindi vengono spawnati e terminati alcuni thread dal ransomware, come visto tramite *Process Monitor*.
* VirtualAlloc / VirtualFree, che come già visto permettono di allocare e deallocare varie aree di memoria.
* GetTickCount / IsDebuggerPresent / QueryPerformanceCounter, 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.

# Analisi Dinamica Avanzata

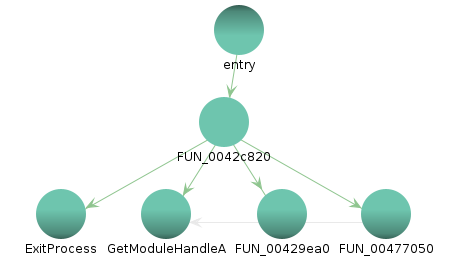
## Individuazione del main

Partiamo dall’entry point, analizzando il *Function Call Graph* per vedere la lista di funzioni invocate.

Vediamo la presenza di numerose funzioni di libreria (non particolarmente interessanti), ed una sola chiamata ad una funzione definita dallo sviluppatore del malware, FUN\_0042c820. Passiamo dunque ad analizzare il disassemblato di tale funzione.

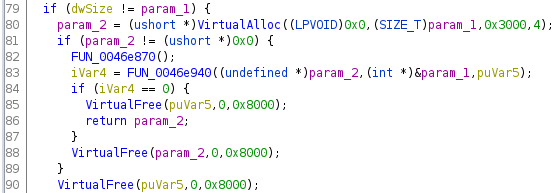
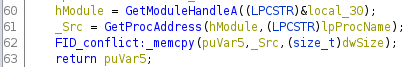
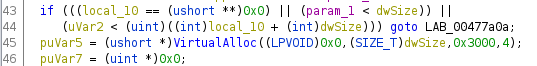
Vediamo che non riceve parametri in input; tipicamente in un main ben strutturato sono presenti i classici parametri argc e argv, ma questo comunque non vieta che non siano presenti se l’applicativo non utilizza argomenti da linea di comando (come nel caso di questo malware).

Analizziamo quindi il function call graph di tale funzione.

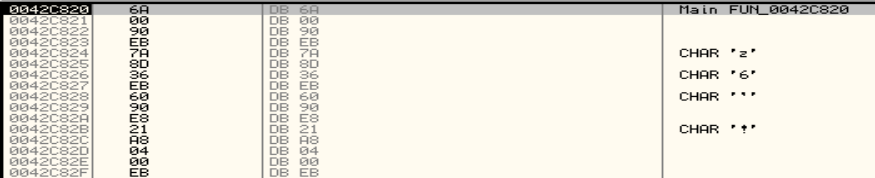
Vediamo che FUN\_0042c820 viene invocata esclusivamente dall’entry point, e a sua volta invoca altre due funzioni user-defined FUN\_00429EA0 e FUN\_00477050. Inoltre tramite GetModuleHandleA si ottiene un handle all’eseguibile hw4.exe, e con ExitProcess() si chiude il processo corrente. Questa funzione dunque non ritorna in nessun’altra, ma una volta effettuate tutte le sue operazioni termina semplicemente programma. Fatte queste osservazioni possiamo affermare che **FUN\_0042C820** è sicuramente il **main** cercato.

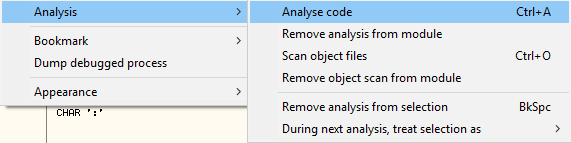
## FUN\_00477050 (alloc\_anti\_dbg)

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.

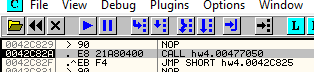
Tuttavia risulta complesso comprendere il comportamento di tale funzione solo tramite decompilatore, per cui procediamo con un’analisi più dettagliata sfruttando *OllyDbg*.

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.

Andiamo quindi a forzare l’analisi del codice tramite l’opzione *‘Analyse Code’.*

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.

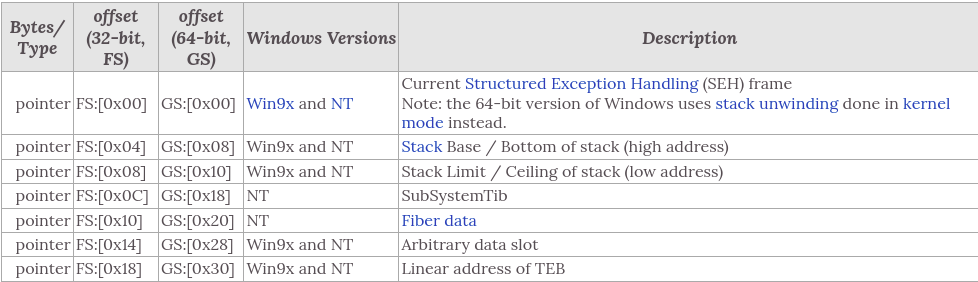
Vengono poi eseguiti una serie di JMP in diverse locazioni di memoria, fino a giungere in 0042C82A dove avviene la call alla FUN\_00477050 che vogliamo analizzare.



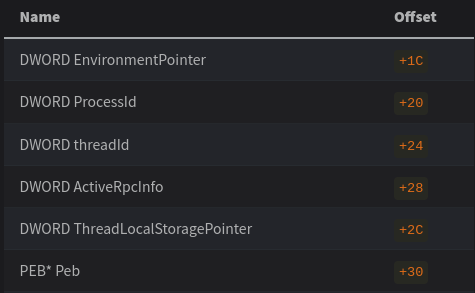
Procediamo tramite *step into*. Come visto dal decompilato, è presente un while(true) all’indirizzo 00477722 dove vengono configurati alcuni parametri locali della funzione. Superiamo questo ciclo spostandoci all’istruzione successiva 0047729A tramite *run to selection* (F4).

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.

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



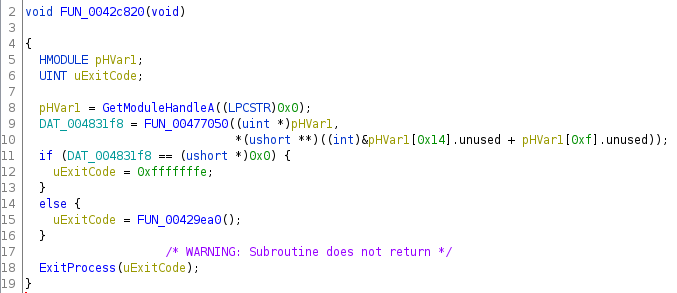


Proseguendo, all’indirizzo 004774BE si accede all’offset 2 della PEB ovvero al campo BeingDebugged. Il valore assunto da tale campo viene copiato in EAX tramite una MOVZX.



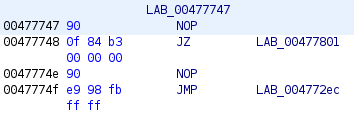
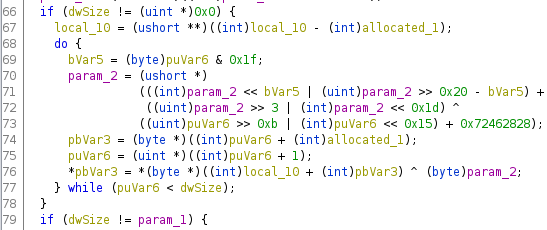
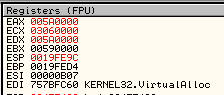
Questo ci suggerisce la probabile presenza di un meccanismo di *Anti Debug*. Ciò viene confermato immediatamente osservando il codice disassemblato in corrispondenza di questo indirizzo. Vediamo infatti che il valore di BeingDebugged viene salvato nella variabile local\_8 (rinominata being\_dbg), e successivamente si controlla proprio il valore assunto da tale variabile.

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.

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.

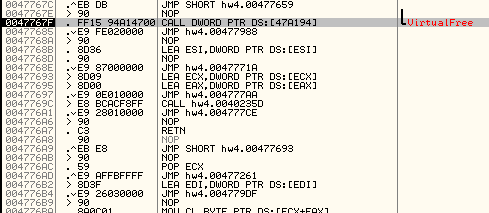


Aiutandoci nuovamente con *Ghidra*, vediamo che successivamente viene eseguito un ciclo *do-while* dove vengono settate alcune variabili locali; ricaviamo l’indirizzo dell’if() appena fuori dal ciclo (00477747) e nuovamente tramite *run to selection* andiamo a bypassarlo muovendoci direttamente su quella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

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

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.

Dal debugger quindi proseguiamo 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.

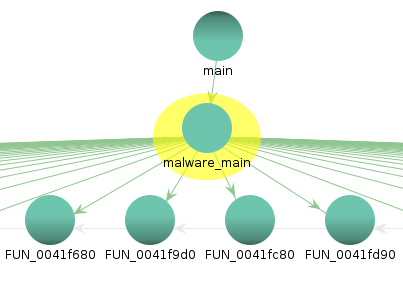
In definitiva la funzione FUN\_00477050, oltre ad allocare e lavorare su alcune aree di memoria, implementa un semplice meccanismo di *Anti Debug*, che è stato bypassato con successo (o almeno sembra). Rinominiamo quindi per semplicità tale funzione su *Ghidra* come alloc\_anti\_dbg.

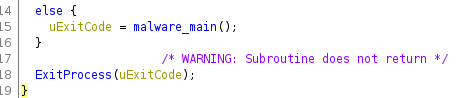
Inoltre tornando rapidamente al main, rinominiamo anche la variabile globale DAT\_004831F8 in allocated\_mem, in quanto contiene il valore di ritorno di alloc\_anti\_dbg, e quindi l’indirizzo dell’area di memoria allocata tramite VirtualAlloc.

## FUN\_00429EA0 (malware\_main)

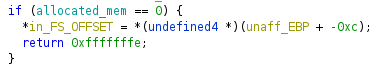
A questo punto possiamo proseguire con l’analisi dell’altra funzione invocata dal main, ovvero FUN\_00429EA0. In alloc\_anti\_dbg non risultava essere presente nessuna delle vere funzionalità tra quelle identificate nel malware, per cui ci aspettiamo che la vera implementazione del ransomware inizi a 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.



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.

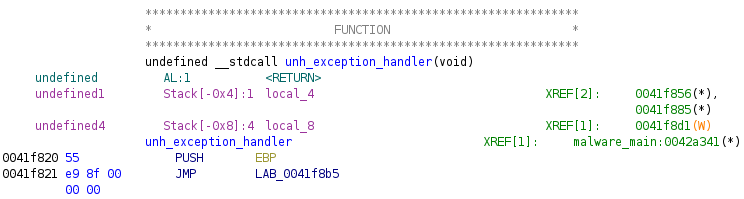
Dopo una serie di JMP e NOP, all’indirizzo 0042C17B viene invocata l’API SetErrorMode, passando come parametro i seguenti flag.

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.

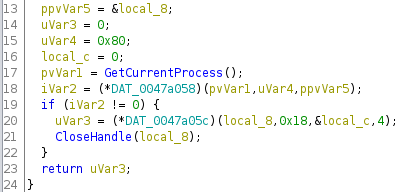
*Ghidra* riconosce tale funzione come una label, quindi utilizziamo *create function* per ridefinire LAB\_0041F820 nella funzione unh\_exception\_handler.

Successivamente, sempre nel malware\_main, viene effettuata la CALL a FUN\_0042CF00, che andiamo quindi ad analizzare nel dettaglio.



### 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.

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.

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

* 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.

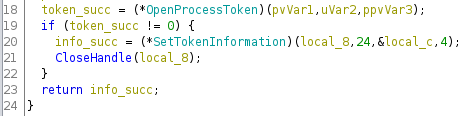
Dal debugger vediamo che l’API invocata è la SetTokenInformation che permete di impostare alcune informazioni relative al token d’accesso.



In particolare i parametri passati sono i seguenti:

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.

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.

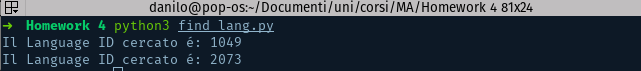
### Language APIs

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.

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 documentazione[[2]](#footnote-3) e scriviamo un *semplice programma python[[3]](#footnote-4)* 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*”.



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.

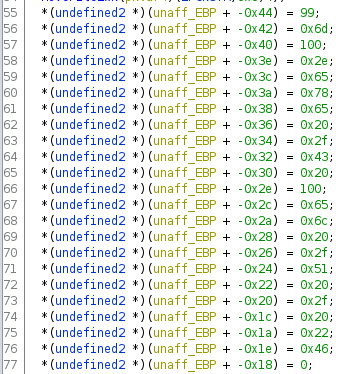
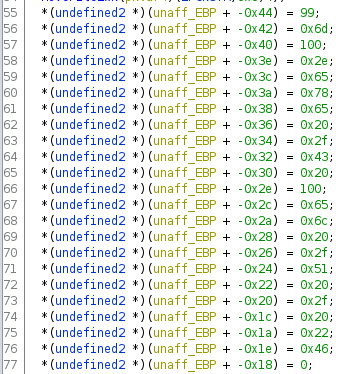
### FUN\_00421DE0 (malware\_exe\_remove)

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.

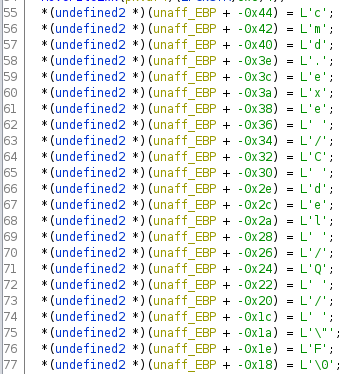
Possiamo vedere che vengono invocate le seguenti API:

* 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:



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

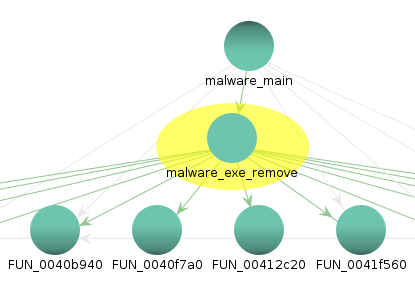
Vediamo che viene composta la stringa ”cmd.exe /C del /Q /F”, già notata nell’analisi dinamica di base con *Process Monitor*.

* 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.

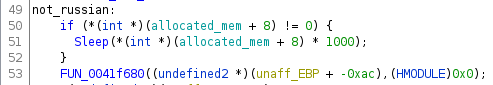
Questo ci permette di fare due osservazioni:

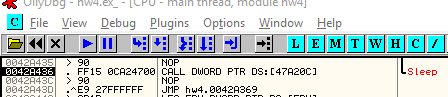
1. Il malware sembra innocuo se si utilizza la lingua russa all’interno del sistema.
2. La funzione malware\_exe\_remove non viene utilizzata anche per rimuovere l’eseguibile dopo aver infettato il sistema; ciò significa che bisogna ancora individuare il blocco di codice in cui è implementato tale meccanismo.

### 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.

Con *OllyDbg* raggiungiamo la Sleep tramite *step over*, e vediamo che viene effettivamente invocata con un timeut 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.

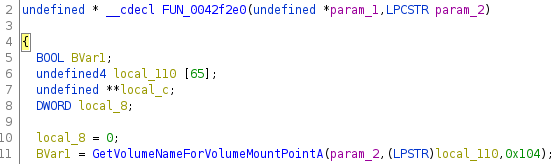
La prima funzione interessante chiamata è FUN\_0042F110, che a sua volta chiama l’API GetWindowsDirectoryA. Questa API recupera il percorso della cartella ‘Windows’, scrivendolo all’interno del buffer passato tramite puntatore nel primo parametro.



Come atteso la funzione ritorna la stringa "[C:\Windows](../../../../../../../C:/Windows)".

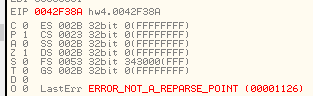


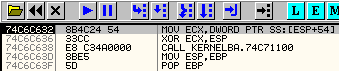
Proseguendo si arriva poi ad invocare la FUN\_0042F430, che chiama a sua volta la FUN\_0042F2E0. Qui si utilizza l’API GetVolumeNameForVolumeMountPointA, che restituisce il *GUID path* del volume fornito come primo parametro di input.



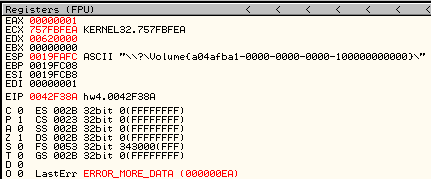
Proseguiamo con *OllyDbg*, e vediamo che come parametro viene passato proprio la stringa “C:\Windows\”.

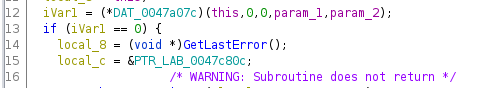
Tuttavia l’API fallisce, causando un’eccezione e restituendo come errore ERROR\_NOT\_A\_REPARSE\_POINT.

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

Procediamo con SHIFT+F9 passando l’eccezione all’applicazione, e vediamo che viene risolta correttamente. Succesivamente si ritorna alla chiamata della FUN\_0042F2E0.

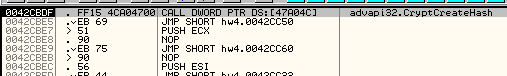
Viene quindi invocata nuovamente la GetVolumeNameForVolumeMountPointA, passando ora come parametro "[C:\](../../../../../../../C:/)". Eseguendo la chiamata vediamo che si ha nuovamente un ERROR\_MORE\_DATA, ma il GUID sembra comunque essere stato scritto all’indirizzo 0019FAF4.

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.



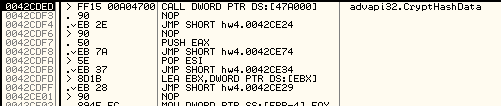
Vediamo da *OllyDbg* che questa corrisponde a CryptAcquireContextA, che permette di acquisire un handle ad un container di chiavi all’interno di uno specifico *Cryptographic Service Provider[[4]](#footnote-5)*. Definiamo tale funzione anche sul codice decompilato.

Anche in FUN\_0042CB70 vi è una funzione non riconosciuta, che vediamo tramite OllyDbg corrispondere a CryptCreateHash.

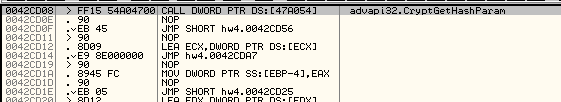
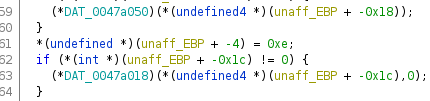
Questa API inizializza l’hashing di uno stream di dati, restituendo al chiamante un handle ad un *hash object*. Il secondo parametro Algid specifica l’algoritmo di hashing da utilizzare, e viene passato come CALG\_MD5. Questo ci permette di fare due osservazioni:

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.

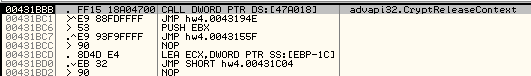
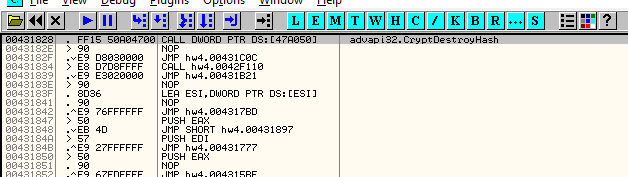
Successivamente si invoca la FUN\_0042CDD0; anche al suo interno viene chiamata una funzione non riconosciuta, che da *OllyDbg* risulta essere CryptHashData, che inserisce dati all’*hash object* aperto in precedenza.

I dati aggiunti all’oggetto corrispondono proprio al GUID del volume [C://](../../../../../../../C://) ottenuto in precedenza.

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.

Ritornando alla FUN\_00431440, vediamo che vengono invocate due funzioni, ancora non riconosciute da Ghidra.

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*.

Come effetto visibile di FUN\_00431440 possiamo osservare che è stata scritta sulla locazione 0019FC54 la stringa 1BCC0199BD42620BF, che probabilmente corrisponde ad un qualche hash MD5 generato a partire dal GUID di C://.

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.

### 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.

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.

### 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 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 effettuano alcuni controlli sul campo dwUsage della struttura, che indica come è possibile effettuare l’enumerazione.

La WNetAddConnection2W effettua una connessione ad una risorsa di rete, indicata all’interno della struttura NETRESOURCE, e viene invocata se (dwUsage & 1) != 0.

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 rinomindando FUN\_00461200 in enum\_net\_res ed il chiamante FUN\_0046C150 in network\_work.

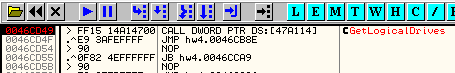
Analizzando il function call graph vediamo che viene invocata soltanto tre volte nella FUN\_0046C640, per cui continuiamo con l’analisi di quest’ultima.

### FUN\_0046C640 (get\_drives\_info)

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.

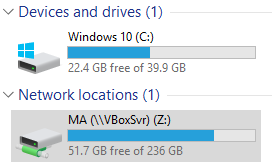
Verificando con *OllyDbg* il valore ritornato da questa funzione, notiamo che vengono sollevate tre eccezioni, in particolare a seguito delle tre CALL network\_work con errore ERROR\_NO\_MORE\_ITEM.

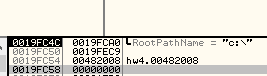
Passando comunque sempre il controllo all’applicazione con SHIFT+F9, tutte le eccezioni vengono raggiunte correttamente e riusciamo a raggiungere l’indirizzo 0046CD49 desiderato.

Procediamo con *step over*, e vediamo che in EAX viene scritto il valore 02000004.



Attualmente sulla VM sono montati i dischi [C://](../../../../../../../C://) e [Z://](../../../../../../../Z://), ed infatti convertendo in binario il valore ritornato dall’API otteniamo 0010000000000000000000000100, 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:\](../../../../../../../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 restutito 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.

### FUN\_0046C640 (priv\_spawn\_thread)

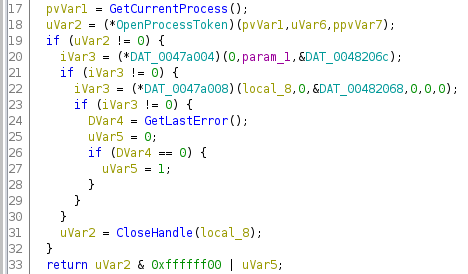
Tornando al malware\_main si effettua un nuovo controllo su allocated\_mem, e se all’offset 0x6493 è presente un valore diverso da ‘\0’ viene invocata la FUN\_0046C640. Vediamo che questa invoca quattro volte FUN\_00474820, che prende una sola stringa in input.

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.

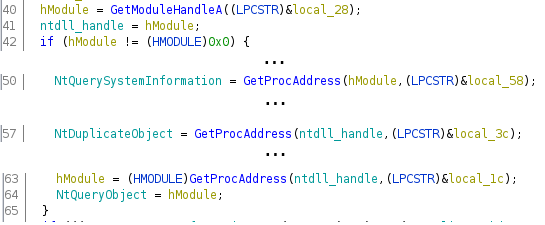
Analizziamone il codice decompilato.

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.

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.



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.

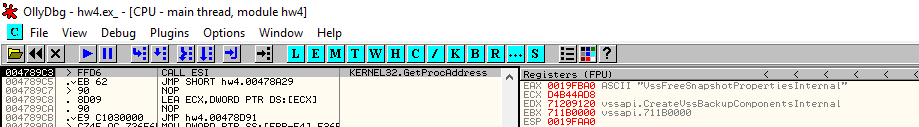
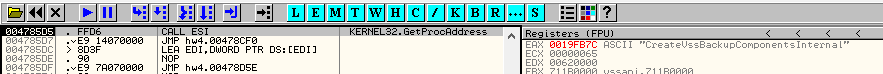
### FUN\_0040FA10 (vss\_backup)

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 inzializzare 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](../../../../../../../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.

Anche questa funzione non sembra particolarmente interessante in relazione al ransomware, per cui rinominiamo FUN\_0040FA10 in vss\_backup e torniamo al malware\_main.

### 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 posizionamo 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.

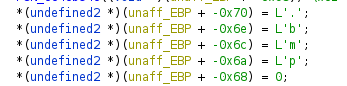
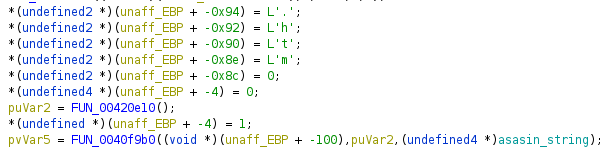


Per il momento ci limitiamo a bypassare la WaitForSingleObject inserendo un NOP al suo posto, tenendo a mente che eventuali errori futuri potrebbero dipendere proprio da questa modifica.

### FUN\_00425870 (malware\_desktop\_asasin)

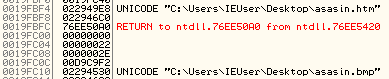
Possiamo proseguire ora con l’analisi di FUN\_00425870 con il debugger. La prima funzione chiamata è FUN\_00420A70, che vediamo invoca SHGetFolderPathW (non riconosciuta da *Ghidra*). Questa API ottiene il path di una cartella mediante il suo valore CSIDL, e lo copia nel buffer passato come primo parametro. Vediamo che a seguito della chiamata viene copiato il percorso del desktop all’indirizzo 0019F94C.

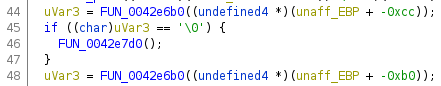
Dal decompilato vediamo che vengono caricate in due zone di memoria differenti le stringhe ‘.htm’ e ‘.bmp’.

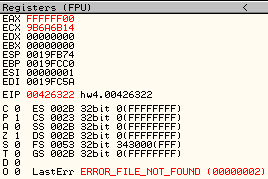


All’indirizzo 0047CBE0 vediamo che è presente la stringa ‘asasin’.

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.

Viene poi invocata due volte FUN\_0042E6B0.

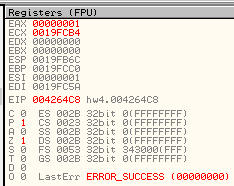
Questa internamente chiama l’API FindFirstFileW che restituisce un handle al file specificato nel primo parametro lpFileName. La funzione ritorna un handle se il file è stato trovato, INVALID\_HANDLE\_VALUE altrimenti. Se il file non viene trovato si ha un errore ERROR\_FILE\_NOT\_FOUND, e viene ritornato 0.

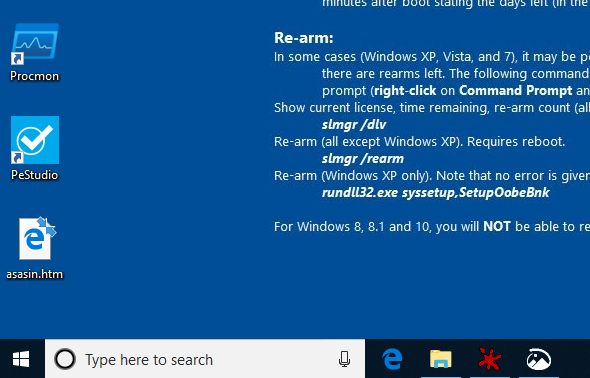
Quindi FUN\_0042E6B0 verifica la presenza di un file ‘asasin’ sul desktop, per cui la rinominiamo in asasin\_exists.

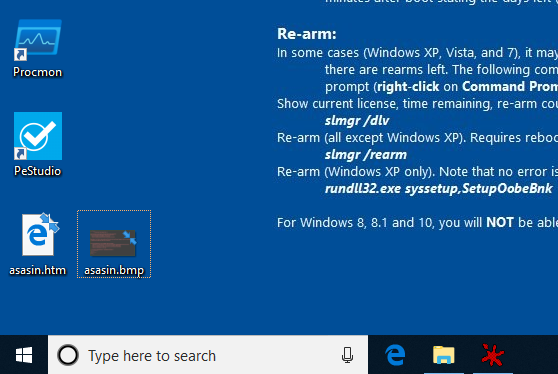
Se il file cercato non esiste, si entra nell’if, e viene invocata FUN\_0042E7D0. Questa funzione utilizza internamente l’API CreateFileW, quindi presumibilmente dovrebbe creare i file asasin.htm e asasin.bmp sul desktop. Rinominiamo quindi FUN\_0042E7D0 in create\_asasin\_file.

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.

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.

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



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.

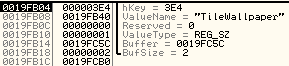
Rinominiamo quindi FUN\_00416290 in open\_desk\_key e continuiamo con l’analisi della FUN\_00425870.

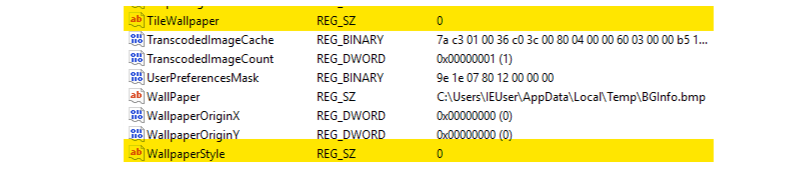
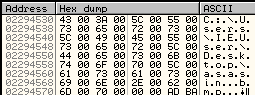
Similmente a quanto fatto in precedenza, viene copiata la stringa “WallpaperStyleQ” in EAX, per invocare poi la FUN\_004205A0. All’interno di tale funzione viene invocata FUN\_00416390 che utilizza l’API RegSetValueExA, permettendo di impostare i dati e il tipo di uno specifico valore di una registry 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.

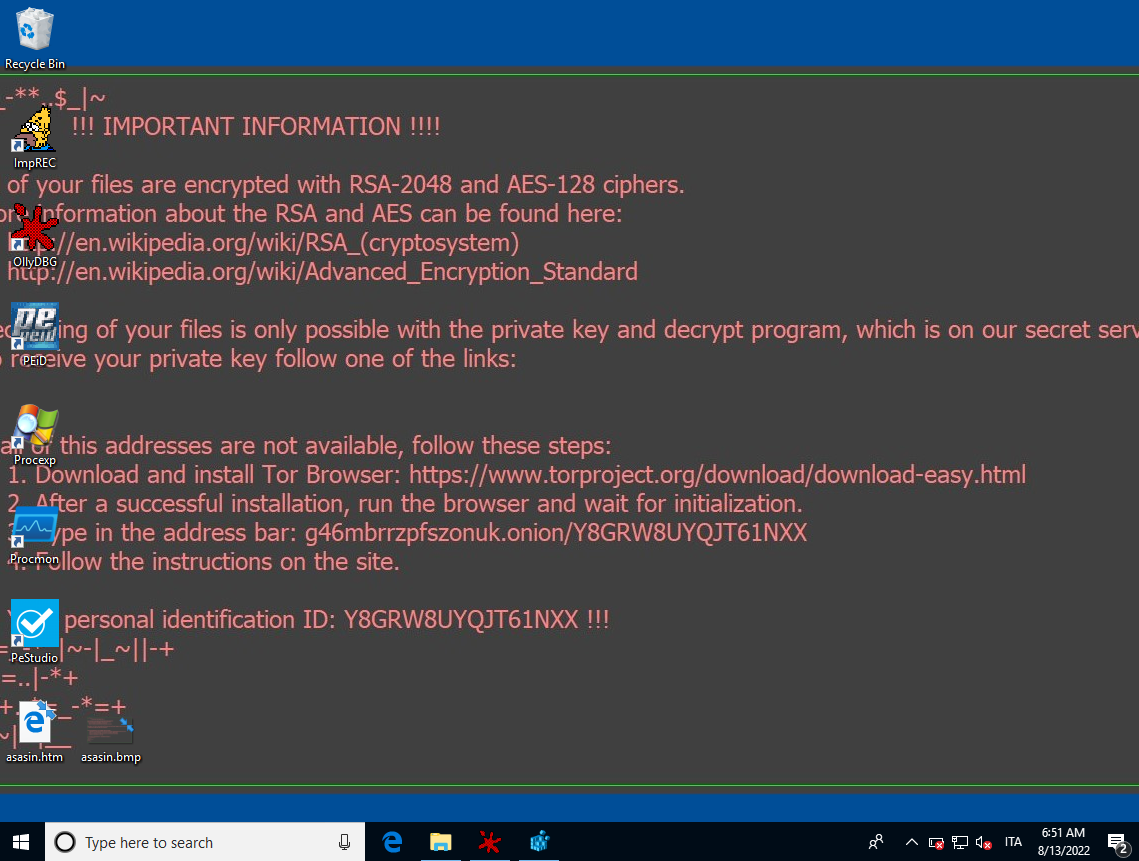
Rinominiamo quindi FUN\_004205A0 in edit\_key e FUN\_00416390 in set\_key\_value.

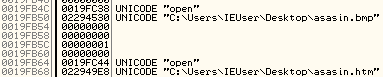
Successivamente viene copiata in EAX la stringa "TileWallpaper", e viene poi chiamata nuovamente edit\_key. In questo caso la RegSetValueExA andrà a impostare il valore di TileWallpaper a 0.

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

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.

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.

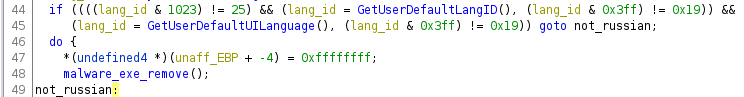
L’ultima API chiamata è la RegCloseKey, che va a chiudere la chiave di sistema precedentemente aperta prima di restituire il controllo al malware\_main.

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.

### Rimozione dell’eseguibile (malware\_exe\_remove)

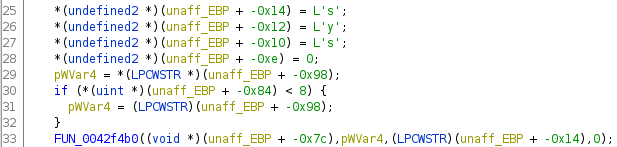
Dopo l’esecuzione di malware\_desktop\_asasin si torna al malware\_main, e proseguendo con l’analisi si giunge al termine del ciclo do-while. Il flusso di esecuzione riprende quindi dalla prima istruzione dopo il do, ovvero la chiamata a malware\_exe\_remove già analizzata in precedenza (5.3.3).

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.

Quindi malware\_exe\_remove è l’effettiva *funzione in cui viene rimosso l’eseguibile* del malware, e viene invocata in due situazioni:

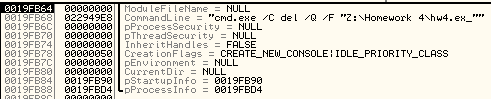
1. Prima di eseguire le azioni del malware, se nel sistema è selezionata la lingua russa.
2. Dopo aver eseguito il malware\_main, se nel sistema non è selezionata la lingua russa.

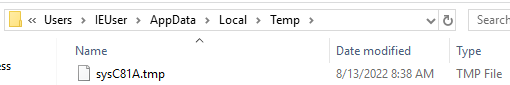
Una analisi più approfondita di tale funzione mostra un particolare non rilevato in precedenza: prima di chiamare FUN\_0042FB40 viene recuperata la stringa sys, e poi si invoca GetTempFileNameW.

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.

Verifichiamo che il file temporaneo viene effettivamente creato, e l’esegubile 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.

# Analisi dei Thread Spawnati

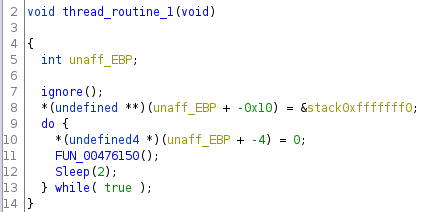
Nella descrizione del malware\_main sono state individuate le seguenti due funzioni eseguite dai due thread creati:

* 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.

## Routine 1 (LAB\_004768D0)

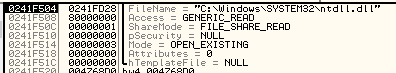
La LAB\_004768D0 è una funzione non correttamente riconosciuta da Ghidra, quindi definiamo una funzione a quell’indirizzo e la rinominiamo in thread\_routine\_1.

Analizzandone il codice vediamo che invoca ogni due secondi, in un ciclo infinito, la FUN\_00476150.

### 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.

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.



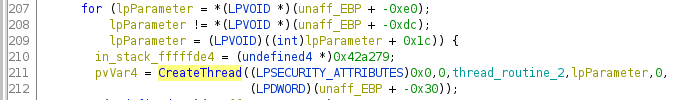
Proseguendo viene invocata l’API GetCurrentProcessId, ottenendo l’identificatore del processo corrente, per poi entrare in un ciclo do-while annidato. All’interno di questo ciclo vengono chiamate molte funzioni differenti, per cui passiamo all’analisi di quelle più interessanti.

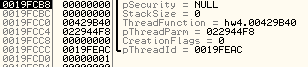
* FUN\_00474C30: utilizza OpenProcess e CreateEventA, per cui non sembra legata alla cifratura dei file.
* FUN\_004750E0: utilizza CreateThread e TerminateThread, per cui spawna a sua volta un altro thread. Tuttavia analizzando la routine LAB\_00474ED0 eseguita dal nuovo thread non sembra esserci nessuna 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.

## Routine 2 (LAB\_00429B40)

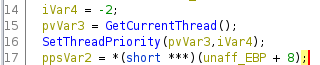
Il secondo thread viene spawnato direttamente nel malware\_main, subito dopo la chiamata a priv\_spawn\_thread. La CreateThread viene chiamata all’interno di un ciclo for, in cui viene passato ogni volta un valore differente di lpParameter.

Con il debugger vediamo che al primo thread viene passato il puntatore a 004244F8, che contiene la stringa ‘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://](../../../../../../../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 invocatate le API GetCurrentThread, per ottenere un handle al thread corrente, e SetThreadPriority, che permette di modificare il livello di proprità 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.

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.

### 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.

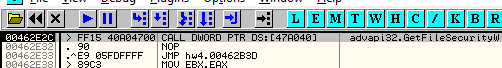
#### FUN\_0046B310 (1)

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:\](../../../../../../../c:/)\*”. Osserviamo che viene utilizzata la wildcard \* per cui è necessario che il thread abbia i permessi di accesso oltre che a [c:\](../../../../../../../c:/), anche a tutte le sottocartelle.

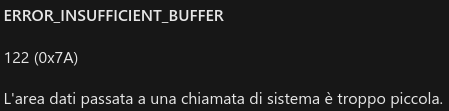


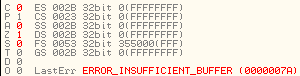
Successivamente FUN\_0046B310 chiama FUN\_00462B10, per cui passiamo a descrivere quest’ultima.

#### 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.

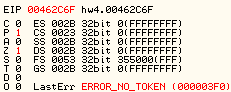
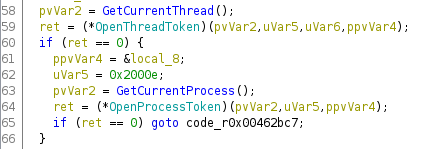
Continuando nel while viene fatto un controllo sul valore di ritorno dell’API, e si verifica anche l’ultimo errore tramite GetLastError; se il codice di errore è diverso da 0x7a (ERROR\_INSUFFICIENT\_BUFFER) allora si esegue un break, uscendo il ciclo while.



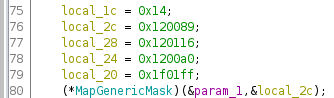
Eseguendo con il debugger vediamo che la prima chiamata a GetFileSecurity ritorna 0, quindi fallisce e con errore proprio ERROR\_INSUFFICIENT\_BUFFER.

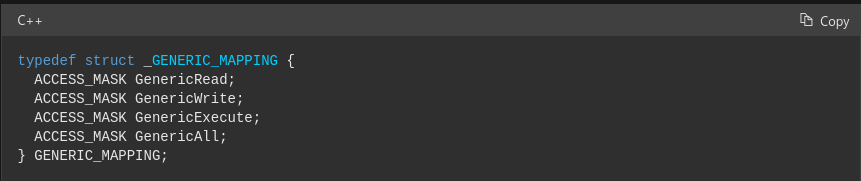
Il ciclo quindi continua finché l’API non ritorna un valore diverso da 0, oppure l’errore è diverso da 0x7a. Ad ogni iterazione viene passata una stringa differente come primo parametro, andando ad esplorare tutte le sottocartelle a partire da [C:\](../../../../../../../C:/). Poiché il numero di iterazioni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

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.

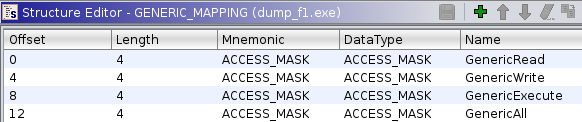
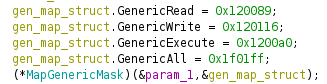
Poichè il valore di ritorno è 0 si entra nel ramo if, dove vengono invocate GetCurrentProcess e OpenProcessToken, già viste in precedenza. Il token di accesso viene questa volta acquisito con successo e viene restituito l’handle a tale token in 025DFBF8.

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.

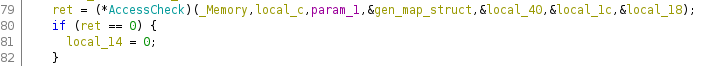
Dalla documentazione vediamo che il secondo parametro è un puntatore ad una struttura GENERIC\_MAPPING, composta dai seguenti campi:

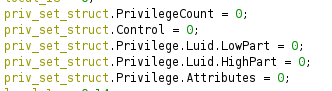
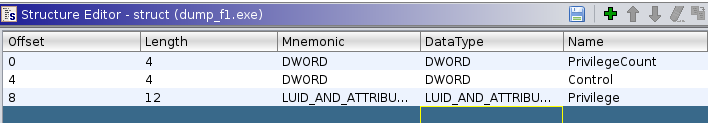


Possiamo definire quindi questa struttura su *Ghidra*, a partire da local\_2c.



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.

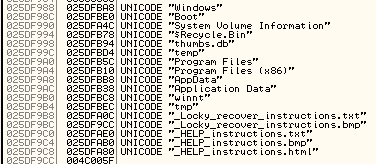
Il quinto parametro è un puntatore ad una struttura PRIVILEGE\_SET, che anche in questo caso andiamo a definire come fatto in precedenza.

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 rinominaimo 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.

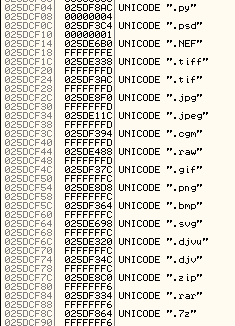
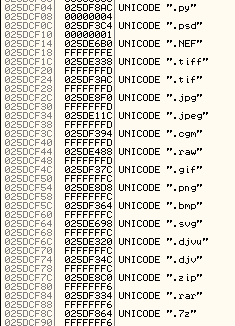
La prima tra queste è FUN\_0043C160, al cui interno sono definite ben 272 variabili locali e varie stringhe offuscate. In input questa funzione prende come primo parametro il nome della cartella correntemente analizzata, e tramite il debugger vediamo che le stringhe caricate corrispondono a nomi specifici di cartelle e 4 file relativi al ransomware.

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.

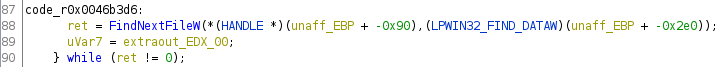
Se le cartelle sono uguali wcsicmp ritorna 0 e FUN\_0043C160 ritorna 1 al chiamante, altrimenti un valore differente. Rinominiamo quindi questa in check\_blacklist\_directory.

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.

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.

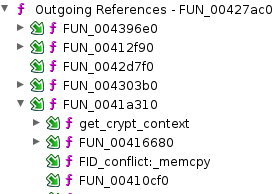
La funzione FUN\_0043AED0 infatti prende in input il nome del file attualmente considerato, ne ricava l’estensione, e tramite wcsicmp verifica se tale estensione è uguale ad una tra quelle della *whitelist*. Rinominiamo quindi FUN\_0043AED0 in check\_whitelist\_extension.

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

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.

### 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.

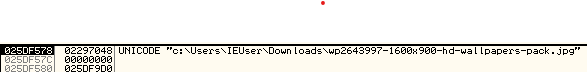
Ciò suggerisce che in FUN\_00427AC0 avvenga l’effettiva cifratura dei file, o comunque operazioni ad essa correlate, per cui proseguiamo con l’analisi dettagliata delle funzioni invocate al suo interno. La prima funzione invocata è FUN\_0041A310, che come già visto invoca get\_crypt\_context e quindi 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.

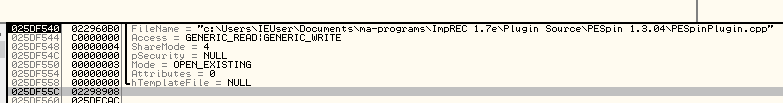
Successivamente viene invocata due volte la FUN\_0040F410, che prende come secondo parametro la stringa 0123456789ABCDEF e tramite memcpy la scrive nella locazione di memoria specificata nel primo parametro. Rinominiamo FUN\_0040F410 in string\_copy.

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-QJ76-1NXX-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.

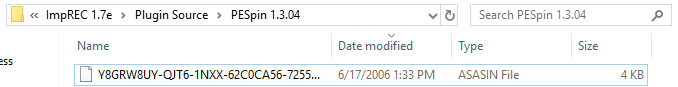
Rinominiamo quindi FUN\_00430450, FUN\_00412F90 e FUN\_0040F9B0 in gen\_rans\_name\_1/2/3 e proseguiamo con l’analisi.

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.

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:

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.

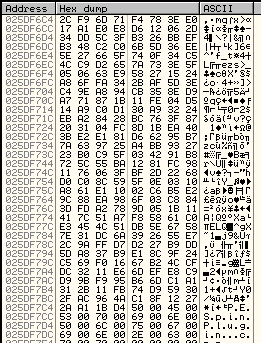
Successivamente viene invocata FUN\_00410D90, che contiene una CryptGenRandom; questa API genera dei bytes crittograficamente casuali, e prende in input un handle al CSP (ottenuto in precedenza con get\_crypt\_context), il numero di bytes casuali da generare, ed un buffer in cui verranno scritti i bytes generati. Il buffer passato come parametro si trova all’indirizzo 025DF6C4, e vediamo infatti che qui vengono scritti dei bytes casuali.

Rinominiamo FUN\_00410D90 in gen\_crypt\_bytes e proseguiamo.

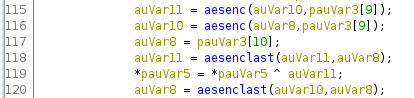
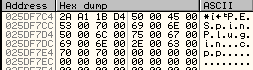
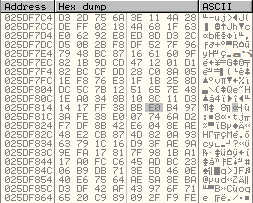
Continuando l’analisi viene invocata FUN\_00410FE0, che internamente chiama CryptEncrypt. Questa API permette di cifrare i dati, utilizzando l’algoritmo specificato dal CSP nel primo parametro. Più nel dettaglio i parametri interessanti che si utilizzano sono:

* 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 plaitext da cifrare prima della chiamata, e che conterrà il ciphertext risultante dopo la chiamata.

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.

Possiamo rinominare FUN\_ 00410FE0 in enc\_crypt\_bytes.

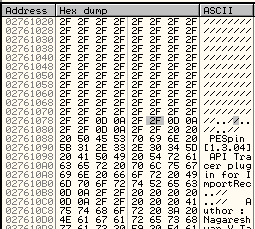
Viene successivamente invocata la FUN\_00473830. Come possiamo vedere dal decompilato vengono invocate molte volte le funzioni aesenc, che esegue una singola iterazione dell’algoritmo di cifratura AES, ed aesenclast, che esegue l’ultima iterazione di AES.

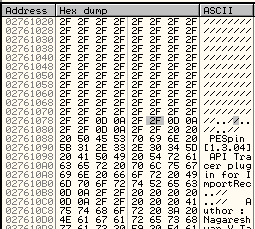
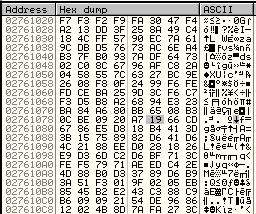
Per vedere cosa viene effettivamente cifrato, analizziamo con *OllyDbg* quali sono i parametri passati a FUN\_00473830. Il primo parametro è l’indirizzo 0025DF7C4, che contiene il nome del file corrente. Eseguendo la function call vediamo che viene cifrato proprio il filename originale.

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.

Verifichiamo che viene effettivamente letto il contenuto originale del file PEspinPlugin.cpp attualmente considerato. Rinominiamo quindi FUN\_004106E0 in read\_original\_file.

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.



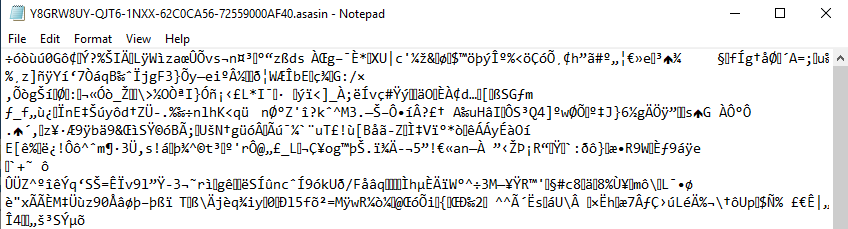
Ora non resta che individuare la porzione di codice in cui il contenuto del file cifrato (presente a 02761020) viene scritto sul file Y8GRW8UY-QJ76-1NXX-62C0CA56-72559000AF40.asasin creato in precedenza.

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.

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

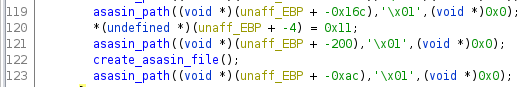
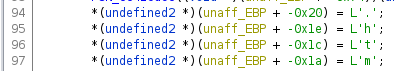
Concludendo, FUN\_00413BE0 cifra il contenuto del file attualmente aperto per cui la rinominiamo aes\_file\_encrypt. Vediamo che successivamente vengono chiamate altre funzioni che effettuano operazioni di minore rilevanza:

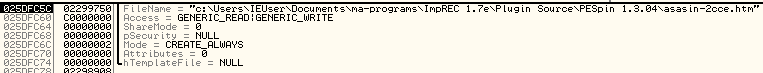
* GetSystemTimeAsFileTime: restituisce la data e l’ora di sistema.
* FUN\_00410AD0: utilizza SetFileTime per modificare la data e l’ora in cui il file è stato creato, modificato o acceduto. Rinominata set\_filetime.
* FUN\_004108B0: utilizza FlushFileBuffers, che prende come unico parametro l’handle 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

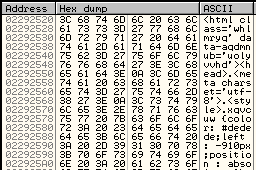
Tornando a FUN\_00427AC0, cerchiamo tramite *function call trees* una chiamata a CreateFile o simili, per individuare rapidamente dove viene creato il file .htm. Vediamo che viene invocata la funzione create\_asasin\_file (FUN\_0042E7D0) già analizzata precedentemente, dopo aver caricato in memoria proprio la stringa ‘.htm’.

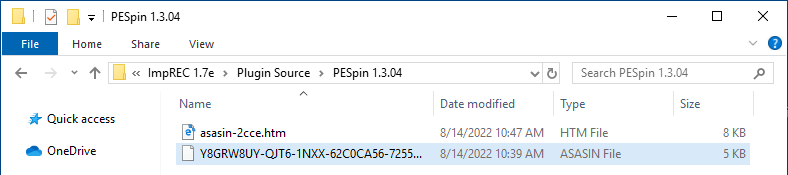
Spostandoci di nuovo sul debugger analizziamo i parametri che vengono passati alla CreateFileW.

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.

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

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.

# 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 cosidetto 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:\](../../../../../../../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.

1. Su esecuzioni differenti l’indirizzo dell’area allocata potrebbe variare in termini di 4° e 5° byte (da destra). [↑](#footnote-ref-2)
2. <https://docs.microsoft.com/en-us/openspecs/office_standards/ms-oe376/6c085406-a698-4e12-9d4d-c3b0ee3dbc4a> [↑](#footnote-ref-3)
3. <https://pastebin.com/VBUqQhip> [↑](#footnote-ref-4)
4. CSP: Modulo software che esegue algoritmi crittografici per autenticazione, cifratura ed encoding [↑](#footnote-ref-5)