**Analisi del Malware – Relazione Homework 4**

Studente: Andrea Pepe Matricola: 0315903

# Obiettivo

Determinare informazioni riguardanti le funzionalità del programma “hw4.ex\_”.

# Analisi di base

In primo luogo, è stata effettuata una ricerca delle stringhe utilizzate dall’eseguibile tramite il comando *strings* su Sistema Operativo Linux; tra le varie stringhe ottenute, di maggiore rilevanza sono le seguenti:

UPX0

UPX1

.rsrc

UPX!

cmutil.dll

KERNEL32.DLL

shell32.dll

shimeng.dll

user32.dll

CmAtolA

ExitProcess

GetProcAddress

LoadLibraryA

VirtualProtect

StrChrW

SE\_IsShimDll

GetPropA

Dalle stringhe che contengono “UPX”, si desume che l’eseguibile sia stato impacchettato usando un packer e, molto probabilmente, tale packer è proprio UPX. Inoltre, sono visibile i nomi delle DLL importate e delle API usate, tra cui figurano *LoadLibraryA* e *GetProcAddress.* Ciò avvalora l’ipotesi che il programma sia impacchettato e che userà tali API per ricostruire la IAT una volta eseguito.

Analizzando l’eseguibile con *PeStudio*, risulta evidente che i nomi delle testate siano UPX0 e UPX1, chiaro segno dell’uso di un packer, così come il valore elevato dell’entropia delle sezioni e il fatto che tali sezioni risultano come self-modifying.

Utilizzando le firme dell’eseguibile fornite da PeStudio, si è effettuata una ricerca su *VirusTotal* che ha prodotto il seguente esito: il file è stato indicato come malevolo da 43 security vendors su 68 e, coloro che lo hanno rilevato come tale, lo descrivono perlopiù come un Trojan.Crypt o Ransom.Locky. Molto probabilmente, l’eseguibile è un ransomware.

Si analizza il file anche con *PEBear*, concentrandosi maggiormente sulla sezione .rsrc vista da *PeStudio*. Si scopre che nella sezione risorse sono presenti due files, entrambi di grandezza pari a 1024 bytes, ma il cui contenuto sembra essere cifrato o comunque non human-readable. Anche utilizzando *ResourceHacker* non si riescono ad ottenere maggiori informazioni a riguardo.

Immagine che contiene tavolo

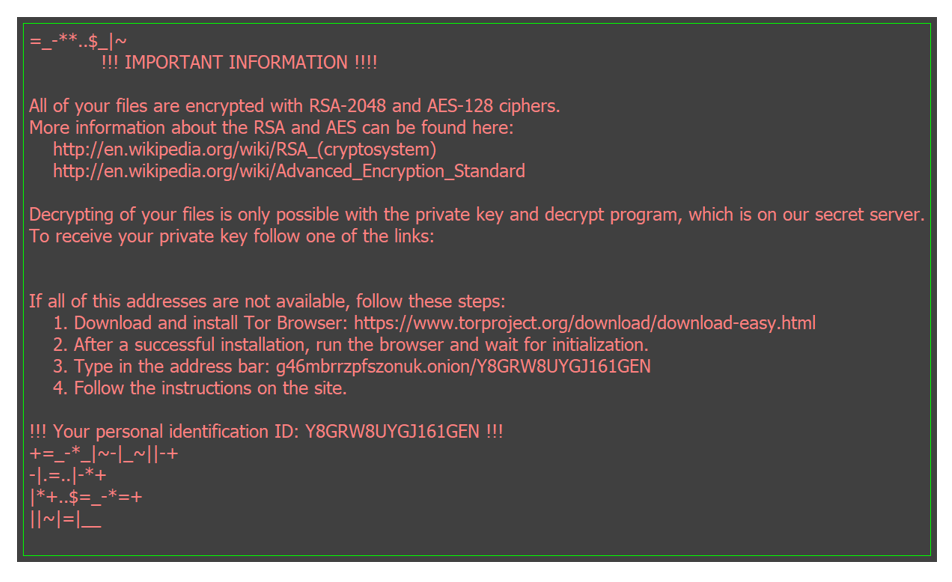
Descrizione generata automaticamente

1 Analisi di hw4.ex\_ con PeStudio

Essendo che l’eseguibile è impacchettato, si prova a farne l’unpacking. Come primo tentativo, si utilizza UPX stesso per spacchettarlo e, apparentemente, risulta essere efficace, in quanto le sezioni viste con *PeStudio* risultano essere state ridenominate come “.text”, “.adata”, “.mdata”, “.rsrc”. Tuttavia, le dimensioni del file unpacked sono le stesse del file packed e la sezione “.text” è ancora etichettata come self-modifying. Inoltre, non è riconosciuta alcuna API utilizzata dal programma. Molto probabilmente l’unpacking non ha funzionato.

Analogamente, l’utilizzo di *PEExplorer* produce risultati non soddisfacenti. Si decidere di procedere ad un unpacking manuale, tentando come prima cosa di individuare l’OEP del programma.

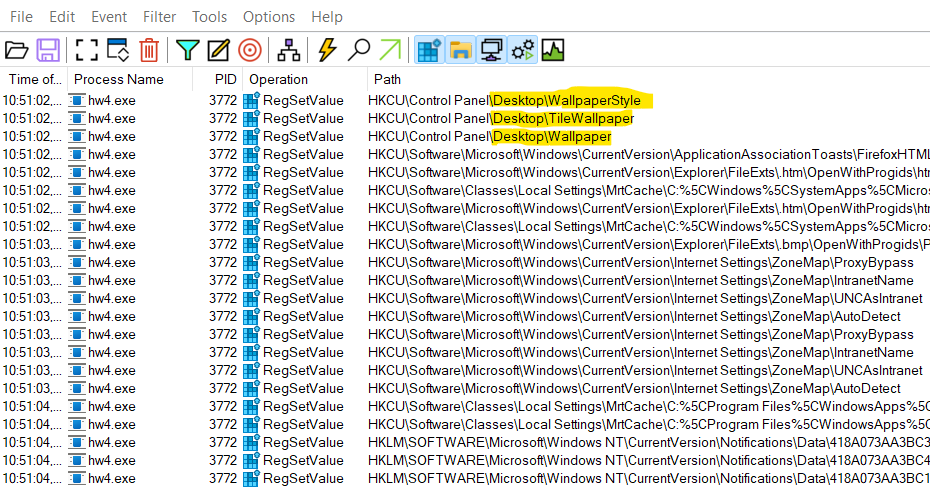
Analizzando il file con *PeID*, innanzitutto non viene rilevato l’utilizzo di alcun packer, ma ciò sembra molto improbabile. Utilizzando il plugin per l’individuazione dell’OEP del programma, l’eseguibile viene lanciato e, nonostante non venga dato alcun OEP come risultato, il malware ha operato e si manifesta nel seguente modo: viene aperta una pagina html ed una bitmap, settata anche come sfondo del desktop, contenenti il messaggio riportato di seguito:



2 Bitmap contenente le informazioni lasciate all'utente dal ransomware

Oltre a ciò, viene creato sul desktop anche un file il cui nome inizia con la stringa indicata come ID personale e che ha estensione “.asasin”, così come tutti i files che il malware ha criptato. Dopodiché, l’eseguibile si cancella.

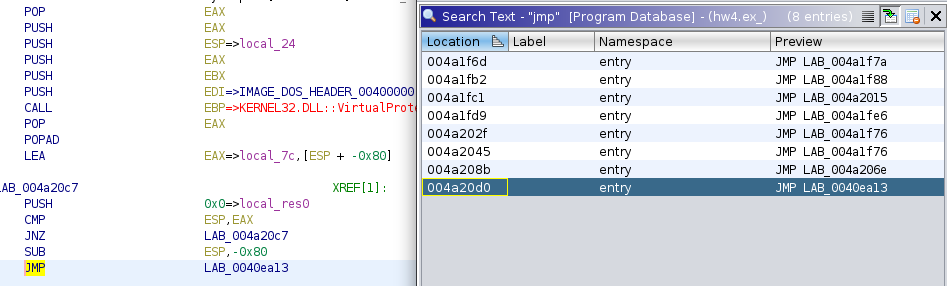
Prima di procedere con i tentativi di unpacking, si tenta di fare anche analisi dinamica di base, utilizzando *ProcessExplorer* e *ProcessMonitor*, insieme a *RegShot*. Tuttavia, il malware compie numerosissime operazioni e accede moltissime volte al registro di sistema, anche se il più delle volte probabilmente a causa di API che utilizza. Filtrando per le operazioni di tipo SetValue nel registro, si vede che modifica alcuni valori che hanno a che fare con lo sfondo del desktop, ma il resto non sembra essere interessante.



3 Analisi con ProcessMonitor. Filtraggio per RegSetValue

# Unpacking

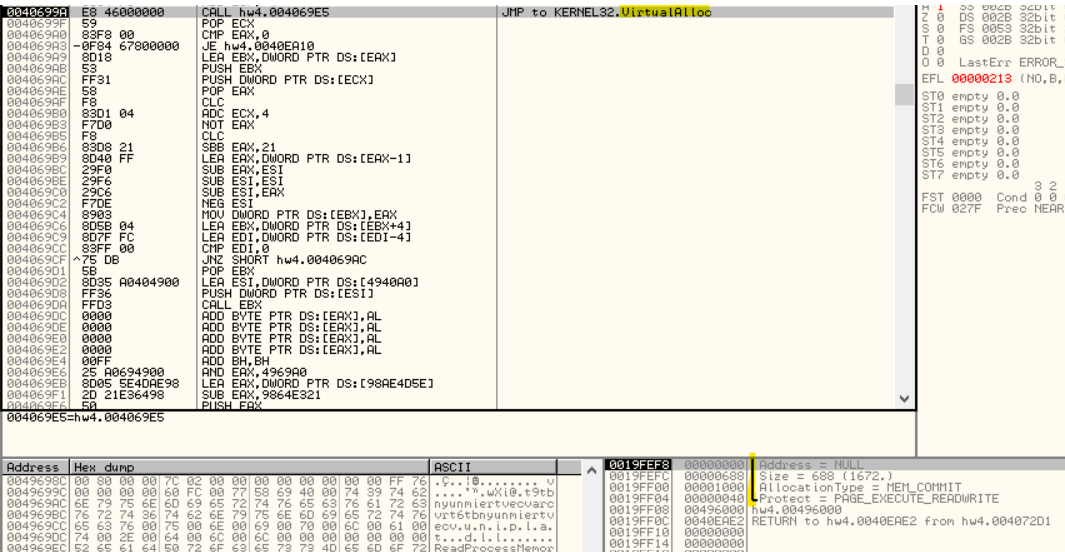
Per poter analizzare il vero malware, è necessario spacchettarlo. Si tenta di ottenere l’OEP guardando le istruzioni di jump dal disassembler di *Ghidra*, per poter individuare possibili tail jump.



4 Long jump ad inidirizzo 0x0040EA13

L’unico jump lontano è quello che porta ad indirizzo 0x0040EA13. Sembra essere un buon candidato essendo effettuato dopo una istruzione POPAD. Tuttavia, tale indirizzo è lo stesso entry point ottenuto effettuando l’unpacking con UPX, molto probabilmente inesatto.

Infatti, seguendo il jump con *OllyDbg*, si cade in un’area di memoria che non dovrebbe contenere codice. Forzando il debugger a disassemblare, si nota che, in realtà, si tratta di istruzioni valide. Seguendo il comportamento col debugger, si vede che, attraverso l’uso di *LoadLibrary* e *GetProcAddress*, il programma ottiene l’indirizzo dell’API *VirtualAlloc*, che utilizza per allocare un’area di memoria dinamicamente, nella quale va a scrivere istruzioni e vi salta dentro con una CALL.



5 Utilizzo di VirtualAlloc

Proseguendo l’esecuzione tramite debugger, si nota che vengono caricate diverse DLL ed APIs, segno che il programma stia ricostruendo la Import Address Table. Ad un certo punto, l’esecuzione tenta di modificare il contenuto delle sezioni di cui è composto, compresa la testata, in cui riscrive anche i nomi corretti delle sezioni:

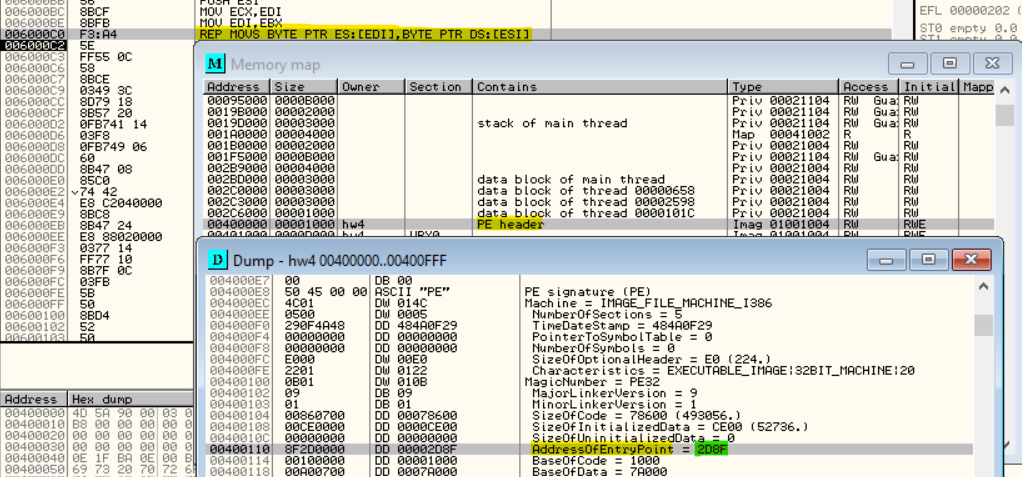


6 Nomi corretti delle sezioni: text, rdata, data, reloc, cdata

Le varie sezioni vengono riscritte con delle istruzioni assembly del tipo:

REP MOVS BYTE PTR ES:[EDI], BYTE PTR DS:[ESI]

Avendo riscritto gli headers, si presuppone che abbia messo a posto anche il valore dell’entry point presente nella testata. Quindi, da *OllyDbg*, dalla finestra Memory Map, si accede ad un dump della testata per ottenere tale entry point, che molto probabilmente sarà il vero OEP.



7 Dump della testata dell'eseguibile dopo essere stata modificata dal malware. L'entry point è ad offset 0x2D8F, quindi ad indirizzo 0x00402D8

Si individua come OEP l’indirizzo 0x00402D8F. Effettivamente, dopo aver concluso le operazioni dello stub e aver ricostruito la IAT, viene effettuato un long jump a tale indirizzo. Inoltre, poco dopo nel flusso d’esecuzione, vengono invocate APIs quali *GetStartupInfoW*, *HeapSetInformation* e *GetCommandLineA*, chiaramente segno di essere nella entry.

Tuttavia, effettuando un dump con il plugin *OllyDump* e aprendo il file di output con il disassembler di *Ghidra*, non appaiono le sezioni ricostruite dal malware, ma sono ancora rimaste le sezioni UPX0, UPX1 e rsrc. Infatti, provando a lanciare l’eseguibile ottenuto facendo il dump, si ottiene un errore. Si utilizza allora il plugin *OllyDumpEx*, il quale permette di effettuare un dump del processo in memoria, ricostruendo bene le testate. Stavolta l’eseguibile viene lanciato correttamente.

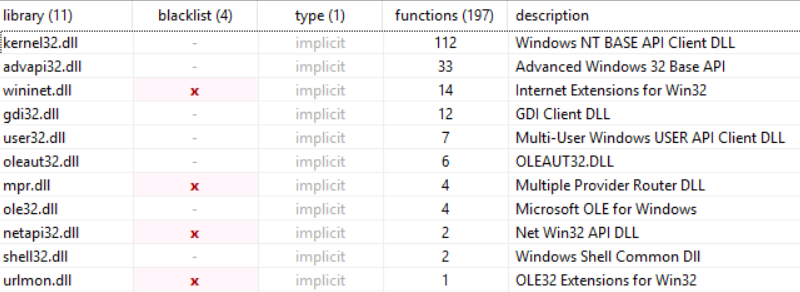
# Nuovi Obiettivi

Ottenuto un dump pulito e funzionante dell’eseguibile e noto che esso contenga un ransomware, ci si pone i seguenti obiettivi da raggiungere nel processo di analisi:

* Determinare quali files e quali drives vengono cifrati e attaccati dal malware
* Determinare come ogni file viene cifrato e con quali chiavi di cifratura, se utilizzate e se individuabili
* Come il malware riesce a fare privilege escalation
* Capire come viene generato l’ID dell’utente
* Capire se eventuali chiavi di cifratura sono generate localmente od ottenute connettendosi alla rete
* Capire se il malware tenta di evadere all’analisi
* Capire se il malware, dopo aver operato, rimane persistente nel sistema e, se sì, se si nasconde
* Comprendere come il malware ottiene la pagina html che viene mostrata e la bitmap. Inoltre, capire come fa ad impostare la bitmap come sfondo del desktop
* Capire se il malware comunica con un server di controllo

# Analisi di base ulteriore

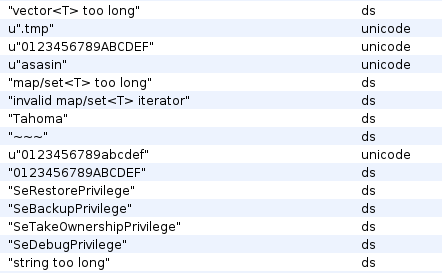
Analizzando il dump precedentemente ottenuto con PeStudio, si vede che sono importate le seguenti DLL:



8 DLL utilizzate dal programma

Da ciò e dalle varie API utilizzate, si comprende che molto probabilmente il malware si connette alla rete.

È stato utilizzato *Ghidra* per individuare le stringhe del programma. Tra tutte quelle ottenute, quelle maggiormente rilevanti che non siano nomi di API o DLL sono le seguenti:



9 Stringhe di hw4\_dump.exe viste da Ghidra

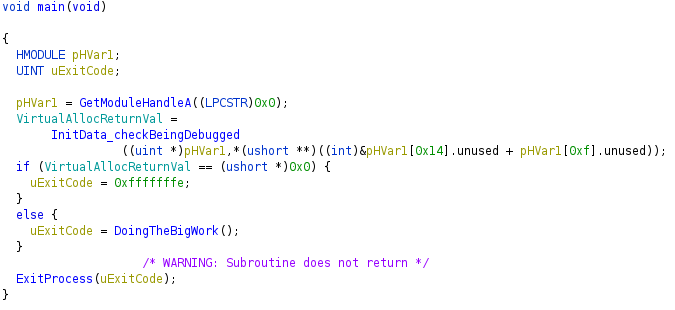
È interessante notare la presenza della stringa “asasin” in formato UNICODE, che è quella utilizzata come estensione dei file criptati. Inoltre, è importante notare la presenza delle stringhe “SeRestorePrivilege”, “SeBackupPrivilege”, “SeTakeOwnershipPrivilege” e “SeDebugPrivilege”, le quali potrebbero descrivere i privilegi che il malware tenta di ottenere nella fase di privilege escalation.

Una ulteriore informazione ottenuta aprendo il file dumpato con *Ghidra* è che il programma è stato compilato con un compilatore VisualStudio e sembra essere stato scritto in C++.

Infine, si fa notare che il file originale contenente il malware, ovvero “hw4.ex\_”, non è appunto un file in formato .exe, quindi non eseguibile direttamente. È in un formato compresso, ma la versione .exe è facilmente ottenibile utilizzando il comando *expand* da linea di comando in Windows.

# Analisi avanzata

Aprendo il file dumpato con il disassembler di *Ghidra*, si è partiti dalla entry e si è subito individuata la funzione main del programma, essendo una delle ultime ad essere chiamata da entry. Il codice disassemblato è molto contorto da seguire, a causa delle numerose istruzioni di jump che lo rendono quasi illeggibile. Tuttavia, il decompilatore fornisce un grosso aiuto nella comprensione delle istruzioni eseguite.



10 Funzione main del programma

Con la chiamata a *GetModuleHandleA(NULL)*, viene ottenuto un handle al file usato per creare il processo chiamante, quindi un handle al file dell’eseguibile.

In seguito, la funzione *InitData\_checkBeingDebugged*, alloca memoria dinamicamente invocando *VirtualAlloc* e popola dati in tale area e in variabili globali. L’indirizzo di tale area è ritornato dalla funzione e salvato in una variabile globale. L’area di memoria viene popolata in modo diverso a seconda se il processo in esecuzione si accorge di essere sotto il controllo di un debugger o meno: infatti, tramite il registro FS, si accede al byte BeingDebugged della struttura Process Environment Block (PEB); se tale byte è diverso da zero, quindi se il processo è debuggato, esso ottiene un handle verso *kernel32.dll* e lo usa per caricare l’API *AllocConsole*. Dopodiché, non inizializza l’area di memoria allocata dinamicamente come dovrebbe, bensì vi copia il codice della *AllocConsole*.



11 Misura anti-debugger: controllo del byte BeingDebugged della PEB

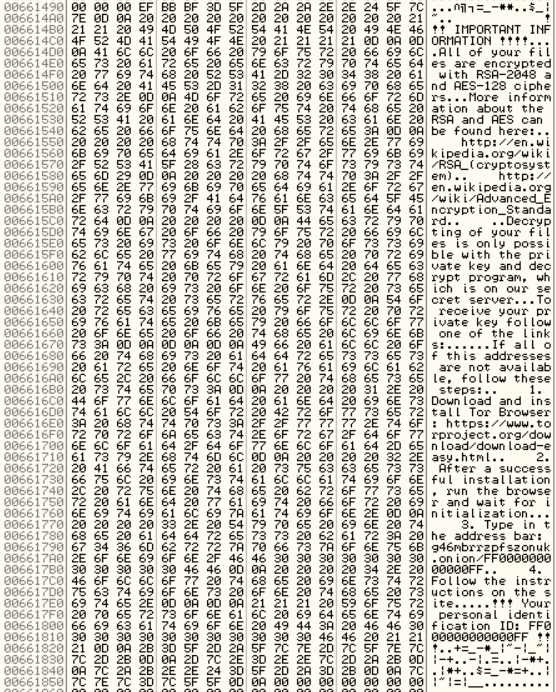
Tramite debugger, modificando l’istruzione che controlla se il valore di BeingDebugged è diverso da zero (facendo sì che risulti zero), si vede che l’area di memoria allocata contiene diverse informazioni: in primis, c’è un’area popolata di bytes a zero, fatta eccezione per tre bytes; ci sono poi altri tre blocchi di bytes. Nel primo c’è, all’inizio, la stringa ASCII “RSA1”, il che fa pensare che il resto dei bytes possano rappresentare una chiave RSA. Nel secondo blocco è presente il testo ASCII delle informazioni che vengono lasciate all’utente dopo che il ransomware ha svolto il suo lavoro. È da notare che al posto dell’ID dell’utente c’è un placeholder pari alla stringa “FF000000000000FF”. Il terzo blocco contiene il codice HTML della pagina contenente le stesse istruzioni di cui prima.



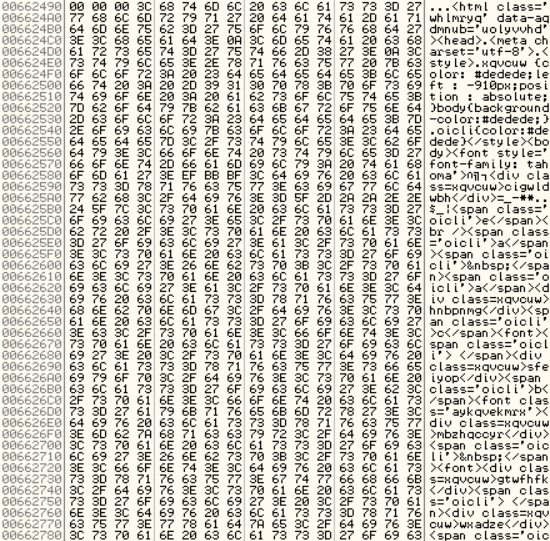
12 Prima area popolata: sono evidenziati gli unici bytes diversi da zero



Seconda area, contenente la stringa "RSA1"



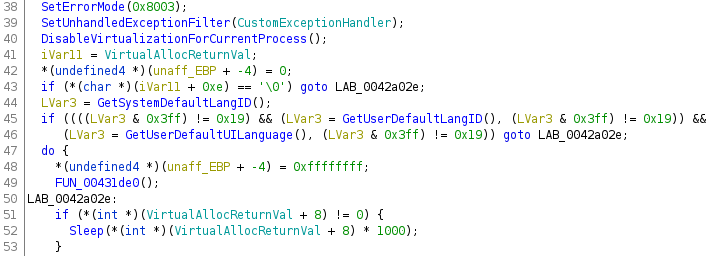
Terza area: è presente il messaggio di istruzioni da lasciare all'utente



Quarta area: è ben visibile in ASCII il testo della pagina html

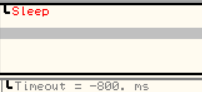
Inoltre, un byte ad offset 0x6493 dall’indirizzo base è impostato a 1.

Procedendo nell’analisi, nel main viene chiamata la funzione che darà inizio al vero lavoro del malware e che è stata ridenominata *DoingTheBigWork*. Essa compie alcune operazioni preliminari:



16 Inizio della funzione DoingTheBigWork

Come prima cosa, invoca la *SetErrorMode* con parametro 0x8003, corrispondente a SEM\_FAILCRITICALERRORS | SEM\_NOPAGFAULTERRORBOX | SEM\_NOOPENFILEERRORBOX. In questo modo evita che il sistema mostri finestre di dialogo riportanti eventuali errori critici ed errori dovuti a chiamate alla *OpenFile*, restituendoli invece al processo chiamante. Successivamente, invoca la *SetUnhandledExceptionFilter* per settare un gestore delle eccezioni custom. Oltre a chiamare delle APIs per ottenere la lingua di default del sistema e quella settata per l’utente, è interessante notare che effettua dei controlli su alcuni byte dell’area di memoria precedentemente allocata (*VirtualAllocReturnVal*). In particolare, a riga 51 nell’immagine precedente, controlla che l’intero ad offset 8 sia diverso da zero e, nel caso, effettua una *Sleep* del valore presente in tale intero moltiplicato per 1000. Se l’area di memoria allocata precedentemente è quella della *AllocConsole*, quindi il processo si è reso conto di essere debuggato, la *Sleep* viene invocata con un valore negativo, pari a -800, il che equivale a fare una *Sleep* all’infinito. È chiaramente una misura anti-debugger.



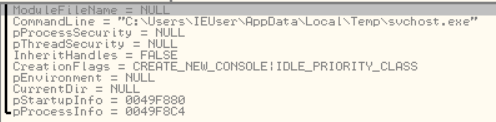
17 Sleep invocata con valore negativo

Sostituendo la *Sleep* con delle NOP, si procede nel flusso d’esecuzione col debugger. Tuttavia, ci si imbatte in un successivo if-statement che controlla di nuovo alcuni bytes dell’area di memoria allocata dinamicamente. In questo caso, se siamo sotto effetto di un debugger, si percorre il ramo else. Qui si scopre che il malware, se si accorge di essere debuggato, tenta di evadere dal controllo del debugger creando una copia di sé stesso sotto il nome di *svchost.exe* nella cartella “C:\Users\<USER NAME>\AppData\Local\Temp”, invocando l’API *CopyFileW*:



18 Parametri dell'invocazione di CopyFileW

Dopodiché, crea un nuovo processo lanciando la sua copia appena creata con la *CreateProcessW*:



19 Parametri della CreateProcessW

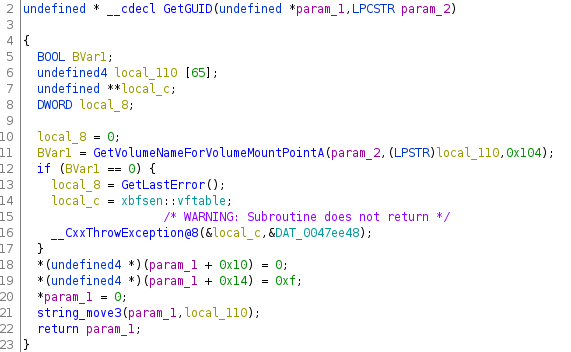
# Prima patch

Si decide dunque di effettuare una patch nel punto in cui viene controllato se il byte BeingDebugged della PEB è diverso da zero, sostituendo la compare tra EDX e BeingDebugged con CMP EDX, 0. In questo modo, il processo allocherà nella memoria sempre ciò che si aspetta di avere.

Procedendo nell’analisi della funzione *DoingTheBigWork*, se invece si entra nel ramo if dell’if-statement discusso precedentemente, viene chiamata una funzione ridenominata *DoHashOfGUID*, in cui si ottiene il GUID del volume del sistema, a partire dalla directory Windows e se ne calcola l’MD5 hash, utilizzando le API Crypto di Windows, presenti nella DLL *advapi32.dll*. Dopodiché, l’hash viene convertito in ASCII utilizzando i caratteri “0123456789ABCDEF”. Della stringa risultante, vengono presi solo i primi 16 caratteri, che verranno successivamente utilizzati per calcolare l’ID che verrà assegnato all’utente. Per ottenere il GUID, il programma invoca l’API *GetVolumeNameForVolumeMountPointA*. Tuttavia, alla prima invocazione, le viene passata la stringa “C:\Windows\”, restituendo errore e lanciando un’eccezione. Controllando il codice dell’eccezione lanciata e dicendo ad *OllyDbg* di passare tale eccezione al programma, si riesce ad arrivare al punto in cui l’API viene invocata con parametro la stringa “C:\”, restituendo il nome univoco del volume.



20 Ramo if in DoingTheBigWork



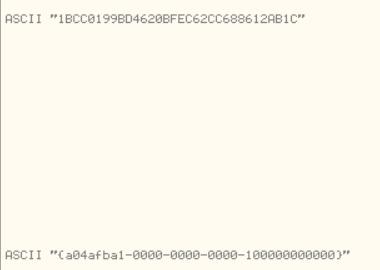
21 Chiamata a GetVolumeNameForVolumeMountPointA



22 Parametri di RaiseException invocando GetVolumeNameForVolumeMountPointA con parametro "C:\Windows\"

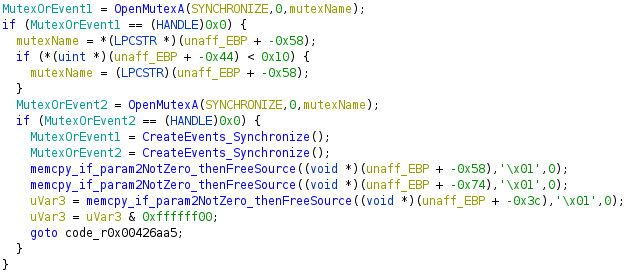


23 Volume name ottenuto



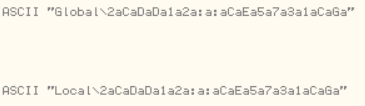
24 In basso, la parte del volume name su cui si calcola l'MD5. In alto, il risultato convertito in ASCII

Continuando nell’analisi della funzione *DoingTheBigWork*, viene invocata una funzione ridenominata *CreateMutex*:



25 Operazioni importanti della CreateMutex

Lo scopo di questa funzione è cercare di ottenere degli handle a degli oggetti di tipo mutex con il diritto d’accesso di tipo SYNCHRONIZE. Viene indicato un nome del mutex, generato a partire dai caratteri ottenuti in precedenza (“1BCC0199BD4620BF”). Si tenta di aprire prima un mutex del namespace globale; se si fallisce, si tenta nel namespace locale. Se anche questa chiamata fallisce, il malware crea due eventi con permessi d’accesso SYNCHRONIZE e con i nomi specificati precedentemente per i mutex, invocando la funzione *CreateEvents\_Synchronize*. Gli handle ottenuti vengono salvati in variabili globali e, molto probabilmente, verranno utilizzati in seguito per sincronizzare dei thread.



26 Nomi dei mutex



27 Funzione CreateEvents\_Synchronize, usata per creare eventi da usare come mutex

Se la *CreateMutex* è andata a buon fine, si prosegue nell’esecuzione della *DoingTheBigWork*, andando ad invocare la funzione *GenerateUserID*. Tale funzione, a partire dai primi 6 caratteri dell’hash precedentemente calcolato e trasformato in ASCII (quindi da “1BCC01”), costruisce l’ID dell’utente attraverso varie operazioni e parametri del sistema che ottiene invocando le APIs *GetUserDefaultUILanguage*, *GetVersionExA*, *GetSystemMetrics*, *DsRoleGetPrimaryDomainInformation*. Inoltre, invoca l’API *IsWOW64Process,* caricata con *GetProcAddress,* per discernere se il processo sta eseguendo su un processore x64 o meno.



28 UserID generato

Lo UserID generato viene salvato in una variabile globale.

Successivamente, nella *DoingTheBigWork*, si vanno a leggere le informazioni presenti nella memoria allocata dinamicamente e scritta nella fase di inizializzazione e vengono salvati dei riferimenti ai blocchi che contiene in delle variabili globali. In particolare, si ha un riferimento per il blocco contenente la stringa “RSA1”, un riferimento per il blocco contenente il messaggio da mostrare all’utente e un riferimento al testo della pagina HTML. Dopodiché, si modificano gli ultimi due blocchi (messaggio e pagina HTML), sostituendo ai placeholder “FF00 0000 0000 00FF” presenti lo UserID calcolato, che, nel caso specifico dell’analisi, risulta essere “Y8GRW8UYGJ161GEN”.



29 Continuazione della DoingTheBigWork

Arrivati a questo punto, il malware inizia a prepararsi per la cifratura dei dati. Come prima cosa, invoca la funzione ridenominata *ExploreLogicalDrives*, la quale a sua volta invoca una funzione ridenominata *EnumNetResources*.



30 Funzione EnumNetResources

Quest’ultima, tenta di fare una enumerazione delle risorse di rete, invocando l’API *WNetOpenEnumW* con parametro RESOURCETYPE\_DISK. Tenta quindi di enumerare le risorse di tipo disco (dischi remoti). Per ogni risorsa ottenuta con la *WNetEnumResourceW*, verifica se si tratti di una risorsa contenente altre risorse (CONTAINER) oppure di una risorsa a cui è possibile connettersi (CONNECTABLE): nel primo caso, invoca la *EnumNetResources* in maniera ricorsiva; nel secondo caso tenta invece di connettersi alla risorsa invocando l’API *WnetAddConnection2W*.

Tuttavia, nell’esecuzione tramite debugger, non essendo presenti risorse di rete di tipo disco, la *WNetEnumResourcesW* ritorna ERROR\_NO\_MORE\_ITEMS, il che porta l’esecuzione ad uscire dal ciclo while e la funzione *EnumNetResources* a ritornare il controllo alla funzione chiamante *ExploreLogicalDrives*.

A questo punto, viene invocata l’API *GetLogicalDrives* per ottenere una bitmask rappresentante le unità disco disponibili. Essendo disponibile il solo drive C, il valore di ritorno è 4. Adesso, per ogni drive rilevato, viene invocata la *GetDriveTypeW* per comprendere la tipologia del drive. Nel caso del disco C (hard disk), essa ritorna il valore 3, corrispondente alla macro DRIVE\_FIXED.



31 ExploreLogicalDrives: tipi di drives esaminati

Il programma analizza tutti i drive disponibili di tipo remoti, rimovibili, hard drive e dischi RAM, salvando informazioni ottenute su di essi.

A questo punto, ritornato il controllo alla *DoingTheBigWork,* si effettua una chiamata ad una funzione ridenominata *PrivilegeEscalation\_and\_CreateThread*.

# Privilege Escalation ed uso di thread



32 Parte iniziale della funzione PrivilegeEscalation\_and\_CreateThread



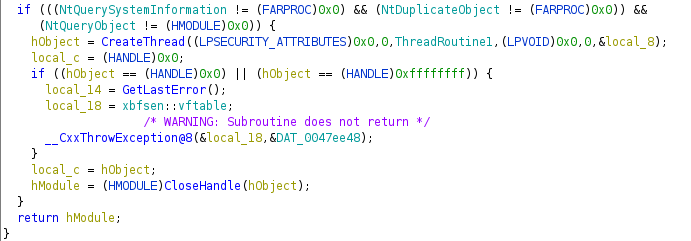
33 Funzione Get\_Privilege

Il processo invoca più volte la funzione *Get\_Privileges*, la quale gli permette di ottenere tutti i privilegi di cui necessita. In particolare, il processo ottiene i seguenti privilegi:

* *SeDebugPrivilege*: consente di debuggare e modificare la memoria di un processo posseduto da un altro account;
* *SeTakeOwnershipPrivilege*: richiesto per ottenere il possesso di files o altri oggetti;
* *SeBackupPrivilege*: richiesto per poter effettuare operazioni di backup. Garantisce permessi di accesso in lettura ad ogni file, a prescindere da quanto specificato nella Access Control List (ACL) dei files;
* *SeRestorePrivilege*: richiesto per effettuare operazioni di ripristino. Garantisce permessi in scrittura su ogni file, a prescindere dalle ACL.

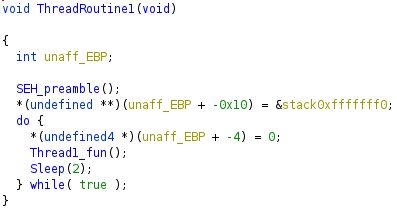
Ottenuti tali privilegi, il malware ha i diritti d’accesso per poter leggere e scrivere tutti i files del sistema.

Proseguendo, nella *PrivilegeEscalation\_and\_CreateThread*, il sistema invoca l’API *GetModuleHandleA* passando come parametro la stringa “ntdll.dll” ed ottenendo così un oggetto di tipo HMODULE verso la dll specificata. Tale handle viene salvato in una variabile globale ed è successivamente utilizzato in 3 chiamate alla API *GetProcAddress* per ottenere gli indirizzi delle funzioni *NtQuerySystemInformation*, *NtDuplicateObject* ed *NtObjectQuery*. A loro volta, tali indirizzi sono salvati in delle variabili globali.



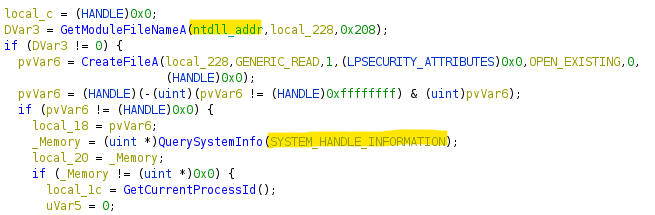
34 Invocazione di CreateThread nella PrivilegeEscalation\_and\_CreateThread

A questo punto, se il processo è riuscito ad ottenere correttamente gli indirizzi delle funzioni della ntdll, viene istanziato un nuovo thread che ha il compito di eseguire la seguente routine:



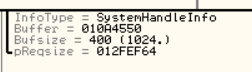
35 Routine del thread lanciato

In un ciclo infinito, viene eseguita la funzione *Thread1\_fun*.



36 Parte iniziale della Thread1\_fun

In primis, con la chiamata a *GetModuleFileNameA*, viene ottenuto il filepath di ntdll, ovvero “C:\Windows\System32\ntdll.dll”. Tale file viene aperto in lettura con la *CreateFileA*, ottenendo un handle verso il file. A questo punto, viene invocata la *QuerySystemInfo*, la quale, al suo interno, alloca memoria in cui poter scrivere il risultato della *NtQuerySystemInformation*, cui viene passato come primo parametro la macro SYSTEM\_HANDLE\_INFO.



37 Parametri della chiamata a NtQuerySystemInformation

Il processo sta cercando di ottenere tutti gli handle attivi nel sistema. La struttura di ritorno è chiamata SYSTEM\_HANDLE\_INFORMATION, ma non è documentata nella documentazione online ufficiale. Tuttavia, è stato possibile reperire informazioni su di essa ai seguenti link:

* <https://www.geoffchappell.com/studies/windows/km/ntoskrnl/api/ex/sysinfo/handle.htm>
* <https://www.geoffchappell.com/studies/windows/km/ntoskrnl/api/ex/sysinfo/handle_table_entry.htm?ts=0,956>

Ottenuta questa lista di handles, il processo invoca *GetCurrentProcessId* per ottenere il proprio PID. Dopodiché, scorre la struttura, controllando se il PID associato all’handle che sta analizzando corrisponde al proprio e, in tal caso, se il valore dell’handle è lo stesso di quello ottenuto invocando *CreateFileA* su ntdll.dll. Dunque, esplora gli altri handles presenti e tenta di ottenere, duplicandoli, degli handle verso i processi proprietari degli handles esaminati, tramite la *DuplicateProcessHandle.*

**

38 DuplicateProcessHandle

Se riesce ad avere l’handle al processo, invoca la *NtDuplicateObject* per avere un handle duplicato alla risorsa posseduta dal processo di cui ha appena ottenuto l’handle. Questa operazione viene effettuata solo per handle ad oggetti di tipo file, ovvero lo stesso di ntdll.dll.

A questo punto, se è riuscito ad ottenere gli handle cercati, viene invocata la *CreateEventsThread*, passandole come parametro l’handle al file appena ottenuto.

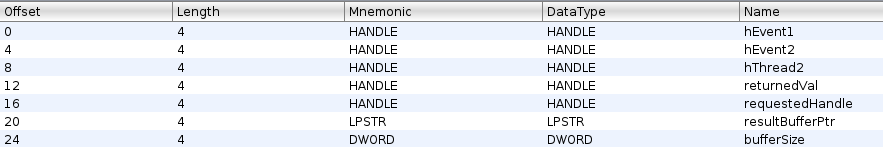


39 Continuazione della Thread1\_fun



40 Funzione CreateEventsThread

Notando delle operazioni di lettura e scrittura tramite offset a partire da una variabile globale, si identifica una struttura che viene utilizzata dai thread per sincronizzarsi. Tale struttura viene ridenominata *ThreadStruct* e i suoi campi sono riportati di seguito:



41 Campi della struttura ThreadStruct

Se ancora non sono stati creati, vengono istanziati due eventi, usati per la sincronizzazione tra i thread, e un altro thread, incaricato di eseguire la *ThreadRoutine2.* Gli handles a tali oggetti sono salvati nei primi tre campi della struttura. A questo punto, il thread 1 (quello che crea il thread che esegue la routine 2, cui ci si riferirà in seguito come thread 2) va in *WaitForsSingleObject* sull’evento 2 e verrà sbloccato dal thread 2 prima di poter proseguire.



42 ThreadRoutine2

Il thread 2 lo sblocca effettuando una *SetEvent* e va a sua volta in *WaitForsSingleObject* sull’evento 1. Il thread 1 popola la struttura globale vista in precedenza, scrivendo l’handle su cui effettuare la *NtQueryObject*, l’indirizzo del buffer dove salvare il risultato e la taglia del buffer. A questo punto, invocando una SetEvent sull’evento 1, il thread 2 viene sbloccato ed invoca la *NtQueryObject* sull’handle specificato, passando il valore 1 come secondo parametro, ovvero PUBLIC\_OBJECT\_TYPE\_INFORMATION. Quindi, nel buffer viene popolata una struttura di tipo PUBLIC\_OBJECT\_TYPE\_INFORMATION, contenente la stringa UNICODE del pathname del file associato all’handle e alcuni bytes riservati per il sistema operativo. Alla fine delle operazioni effettuate, il thread 2 scrive il campo *returnedVal* della struttura per indicare se le operazioni sono andate o meno a buon fine.



43Parametri passati in una chiamata alla NtQueryObject

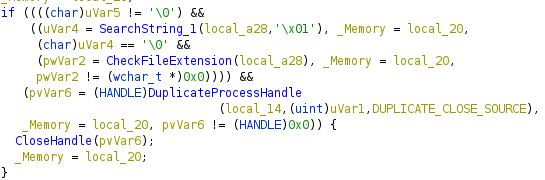
Nel frattempo, il thread 1 si era messo nuovamente in *WaitForSingleObject* sull’evento 2 e viene sbloccato dal thread 2, una volta che quest’ultimo ha effettuato la query. Tuttavia, tale *WaitForSingleObject*, a differenza delle altre chiamate, è effettuata con un valore del timeout diverso da INFINITE; infatti, il valore del timeout è hardcoded e pari a 200 millisecondi, scaduti i quali il thread 1 decide di uccidere il thread 2 invocando una *TerminateThread*.

In altri termini, se il thread 2 impiega più di 200 ms a restituire un risultato, esso viene ucciso e ne viene creato uno nuovo in ogni ciclo. Quindi, non è possibile vedere il flusso di esecuzione corretto se ci si ferma a debuggare il thread 2. Sembra essere un modo per ostacolare l’analisi. Dunque, è stata rilevata una ulteriore misura antidebugger che fa uso del multithreading.

# Seconda patch

Viene effettuata una patch all’eseguibile, sostituendo il valore 200 del timeout della *WaitForSingleObject* con -1, ovvero INFINITE, così che il thread 2 non venga mai ucciso prima di aver restituito un risultato. Si sostituisce la PUSH del parametro sullo stack con una PUSH -1.

Continuando ad analizzare l’esecuzione della *CreateEventsThread*, essa ritorna il valore presente nella struttura globale chiamato *returnedVal*. Esso è 0 oppure 1, a seconda se la query sia andata o meno a buon fine.



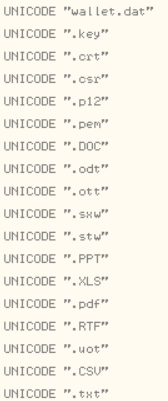
44 Thread1\_fun: codice dopo la CreateEventsThread

Se viene ritornato 1, si invoca la funzione *SearchString\_1*, passandole come primo parametro l’indirizzo della stringa ottenuta in seguito alla query e come secondo parametro il valore 1. Questo comporta che la funzione invochi la *wcsstr* per verificare se la stringa contiene delle occorrenze di altre stringhe che vengono costruite sullo stack e sono ben visibili dal debugger:



45 Stringhe controllate dalla SearchString\_1

Se almeno una di queste stringhe è presenti, viene ritornato l’indirizzo della prima occorrenza, altrimenti NULL. Quindi, nel caso in cui non ci siano occorrenze nella stringa, viene invocata anche la *CheckFileExtension*. Tale funzione svolge un lavoro simile alla precedente, invocando la *wcsicmp* per confrontare l’estensione del file con una lista di possibili stringhe costruite sullo stack.



46 Elenco delle estensioni sullo stack; è presentata solo una vista parziale, l'elenco contiene molte altre estensioni

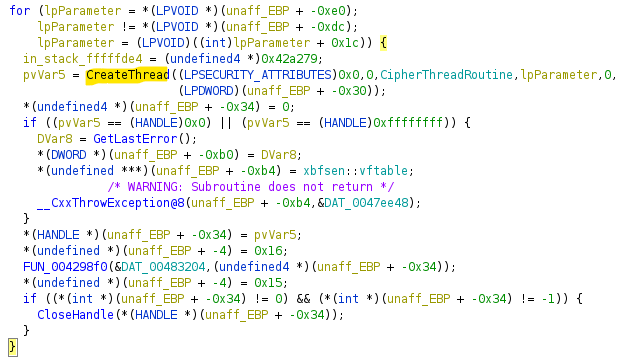
Se la stringa del file risulta avere una estensione corrispondente ad una di quelle nella lista, viene invocata la *DuplicateProcessHandle* sull’handle del file, ma con terzo parametro il flag DUPLICATE\_CLOSE\_SOURCE. In questo modo, il malware sta ottenendo un handle al file ma cancellando l’handle al medesimo file posseduto dal processo originario.

Dunque, l’intero meccansimo del mutithreading è volto a filtrare tutti gli handles ai file posseduti dagli altri processi nel sistema e, se tali files non hanno nel proprio pathname una delle stringhe controllate dalla *SearchString\_1* e hanno una estensione presente nella lista costruita dalla *CheckFileExtension,* chiudere gli handle degli altri processi verso tali files. Sembra che il malware stia facendo in modo di ottenere l’accesso esclusivo ai files che intende cifrare.

Sono elencate di seguito tutte le estensioni dei files controllate dalla *CheckFileExtension*:

* "wallet.dat"
* ".key"
* ".crt"
* ".csr"
* ".p12"
* ".pem"
* ".DOC"
* ".odt"
* ".ott"
* ".sxw"
* ".stw"
* ".PPT"
* ".XLS"
* ".pdf"
* ".RTF"
* ".uot"
* ".CSV"
* ".txt"
* ".xml"
* ".3ds"
* ".max"
* ".3dm"
* ".DOT"
* ".docx"
* ".docm"
* ".dotx"
* ".dotm"
* ".602"
* ".hwp"
* ".ods"
* ".ots"
* ".sxc"
* ".stc"
* ".dif"
* ".xlc"
* ".xlm"
* ".xlt"
* ".xlw"
* ".slk"
* ".xlsb"
* ".xlsm"
* ".xlsx"
* ".xltm"
* ".xltx"
* ".wk1"
* ".wks"
* ".123"
* ".wb2"
* ".odp"
* ".otp"
* ".sxi"
* ".sti"
* ".pps"
* ".pot"
* ".sxd"
* ".std"
* ".pptm"
* ".pptx"
* ".potm"
* ".potx"
* ".uop"
* ".odg"
* ".otg"
* ".sxm"
* ".mml"
* ".docb"
* ".ppam"
* ".ppsx"
* ".ppsm"
* ".sldx"
* ".sldm"
* ".ms11"
* ".ms11 (Security copy)"
* ".lay"
* ".lay6"
* ".asc"
* ".onetoc2"
* ".pst"
* ".001"
* ".002"
* ".003"
* ".004"
* ".005"
* ".006"
* ".007"
* ".008"
* ".009"
* ".010"
* ".011"
* ".SQLITE3"
* ".SQLITEDB"
* ".sql"
* ".mdb"
* ".db"
* ".dbf"
* ".odb"
* ".frm"
* ".MYD"
* ".MYI"
* ".ibd"
* ".mdf"
* ".ldf"
* ".php"
* ".c"
* ".cpp"
* ".pas"
* ".asm"
* ".h"
* ".js"
* ".vb"
* ".vbs"
* ".pl"
* ".dip"
* ".dch"
* ".sch"
* ".brd"
* ".cs"
* ".asp"
* ".rb"
* ".java"
* ".jar"
* ".class"
* ".sh"
* ".bat"
* ".cmd"
* ".py"
* ".psd"
* ".NEF"
* ".tiff"
* ".tif"
* ".jpg"
* ".jpeg"
* ".cgm"
* ".raw"
* ".gif"
* ".png"
* ".bmp"
* ".svg"
* ".djvu"
* ".djv"
* ".zip"
* ".rar"
* ".7z"
* ".gz"
* ".tgz"
* ".tar"
* ".bak"
* ".tbk"
* ".tar.bz2"
* ".PAQ"
* ".ARC"
* ".aes"
* ".gpg"
* ".apk"
* ".asset"
* ".asset"
* ".bik"
* ".bsa"
* ".d3dbsp"
* ".das"
* ".forge"
* ".iwi"
* ".lbf"
* ".litemod"
* ".litesql"
* ".ltx"
* ".re4"
* ".sav"
* ".upk"
* ".wallet"
* ".vmx"
* ".vmdk"
* ".vdi"
* ".qcow2"
* ".mp3"
* ".wav"
* ".swf"
* ".fla"
* ".wmv"
* ".mpg"
* ".vob"
* ".mpeg"
* ".asf"
* ".avi"
* ".mov"
* ".mp4"
* ".3gp"
* ".mkv"
* ".3g2"
* ".flv"
* ".wma"
* ".mid"
* ".m3u"
* ".m4u"
* ".m4a"
* ".n64"
* ".contact"
* ".dbx"
* ".doc"
* ".docx"
* ".jnt"
* ".jpg"
* ".mapimail"
* ".msg"
* ".oab"
* ".ods"
* ".pdf"
* ".pps"
* ".ppsm"
* ".ppt"
* ".pptm"
* ".prf"
* ".pst"
* ".rar"
* ".rtf"
* ".txt"
* ".wab"
* ".xls"
* ".xlsx"
* ".xml"
* ".zip"
* ".1cd"
* ".3ds"
* ".3g2"
* ".3gp"
* ".7z"
* ".7zip"
* ".accdb"
* ".aoi"
* ".asf"
* ".asp"
* ".aspx"
* ".asx"
* ".avi"
* ".bak"
* ".cer"
* ".cfg"
* ".class"
* ".config"
* ".css"
* ".csv"
* ".db"
* ".dds"
* ".dwg"
* ".dxf"
* ".flf"
* ".flv"
* ".html"
* ".idx"
* ".js"
* ".key"
* ".kwm"
* ".laccdb"
* ".ldf"
* ".lit"
* ".m3u"
* ".mbx"
* ".md"
* ".mdf"
* ".mid"
* ".mlb"
* ".mov"
* ".mp3"
* ".mp4"
* ".mpg"
* ".obj"
* ".odt"
* ".pages"
* ".php"
* ".psd"
* ".pwm"
* ".rm"
* ".safe"
* ".sav"
* ".save"
* ".sql"
* ".srt"
* ".swf"
* ".thm"
* ".vob"
* ".wav"
* ".wma"
* ".wmv"
* ".xlsb"
* ".3dm"
* ".aac"
* ".ai"
* ".arw"
* ".c"
* ".cdr"
* ".cls"
* ".cpi"
* ".cpp"
* ".cs"
* ".db3"
* ".docm"
* ".dot"
* ".dotm"
* ".dotx"
* ".drw"
* ".dxb"
* ".eps"
* ".fla"
* ".flac"
* ".fxg"
* ".java"
* ".m"
* ".m4v"
* ".max"
* ".mdb"
* ".pcd"
* ".pct"
* ".pl"
* ".potm"
* ".potx"
* ".ppam"
* ".ppsm"
* ".ppsx"
* ".pptm"
* ".ps"
* ".pspimage"
* ".r3d"
* ".rw2"
* ".sldm"
* ".sldx"
* ".svg"
* ".tga"
* ".wps"
* ".xla"
* ".xlam"
* ".xlm"
* ".xlr"
* ".xlsm"
* ".xlt"
* ".xltm"
* ".xltx"
* ".xlw"
* ".act"
* ".adp"
* ".al"
* ".bkp"
* ".blend"
* ".cdf"
* ".cdx"
* ".cgm"
* ".cr2"
* ".crt"
* ".dac"
* ".dbf"
* ".dcr"
* ".ddd"
* ".design"
* ".dtd"
* ".fdb"
* ".fff"
* ".fpx"
* ".h"
* ".iif"
* ".indd"
* ".jpeg"
* ".mos"
* ".nd"
* ".nsd"
* ".nsf"
* ".nsg"
* ".nsh"
* ".odc"
* ".odp"
* ".oil"
* ".pas"
* ".pat"
* ".pef"
* ".pfx"
* ".ptx"
* ".qbb"
* ".qbm"
* ".sas7bdat"
* ".say"
* ".st4"
* ".st6"
* ".stc"
* ".sxc"
* ".sxw"
* ".tlg"
* ".wad"
* ".xlk"
* ".aiff"
* ".bin"
* ".bmp"
* ".cmt"
* ".dat"
* ".dit"
* ".edb"
* ".flvv"
* ".gif"
* ".groups"
* ".hdd"
* ".hpp"
* ".log"
* ".m2ts"
* ".m4p"
* ".mkv"
* ".mpeg"
* ".ndf"
* ".nvram"
* ".ogg"
* ".ost"
* ".pab"
* ".pdb"
* ".pif"
* ".png"
* ".qed"
* ".qcow"
* ".qcow2"
* ".rvt"
* ".st7"
* ".stm"
* ".vbox"
* ".vdi"
* ".vhd"
* ".vhdx"
* ".vmdk"
* ".vmsd"
* ".vmx"
* ".vmxf"
* ".3fr"
* ".3pr"
* ".ab4"
* ".accde"
* ".accdr"
* ".accdt"
* ".ach"
* ".acr"
* ".adb"
* ".ads"
* ".agdl"
* ".ait"
* ".apj"
* ".asm"
* ".awg"
* ".back"
* ".backup"
* ".backupdb"
* ".bank"
* ".bay"
* ".bdb"
* ".bgt"
* ".bik"
* ".bpw"
* ".cdr3"
* ".cdr4"
* ".cdr5"
* ".cdr6"
* ".cdrw"
* ".ce1"
* ".ce2"
* ".cib"
* ".craw"
* ".crw"
* ".csh"
* ".csl"
* ".db\_journal"
* ".dc2"
* ".dcs"
* ".ddoc"
* ".ddrw"
* ".der"
* ".des"
* ".dgc"
* ".djvu"
* ".dng"
* ".drf"
* ".dxg"
* ".eml"
* ".erbsql"
* ".erf"
* ".exf"
* ".ffd"
* ".fh"
* ".fhd"
* ".gray"
* ".grey"
* ".gry"
* ".hbk"
* ".ibank"
* ".ibd"
* ".ibz"
* ".iiq"
* ".incpas"
* ".jpe"
* ".kc2"
* ".kdbx"
* ".kdc"
* ".kpdx"
* ".lua"
* ".mdc"
* ".mef"
* ".mfw"
* ".mmw"
* ".mny"
* ".moneywell"
* ".mrw"
* ".myd"
* ".ndd"
* ".nef"
* ".nk2"
* ".nop"
* ".nrw"
* ".ns2"
* ".ns3"
* ".ns4"
* ".nwb"
* ".nx2"
* ".nxl"
* ".nyf"
* ".odb"
* ".odf"
* ".odg"
* ".odm"
* ".orf"
* ".otg"
* ".oth"
* ".otp"
* ".ots"
* ".ott"
* ".p12"
* ".p7b"
* ".p7c"
* ".pdd"
* ".pem"
* ".plus\_muhd"
* ".plc"
* ".pot"
* ".pptx"
* ".psafe3"
* ".py"
* ".qba"
* ".qbr"
* ".qbw"
* ".qbx"
* ".qby"
* ".raf"
* ".rat"
* ".raw"
* ".rdb"
* ".rwl"
* ".rwz"
* ".s3db"
* ".sd0"
* ".sda"
* ".sdf"
* ".sqlite"
* ".sqlite3"
* ".sqlitedb"
* ".sr2"
* ".srf"
* ".srw"
* ".st5"
* ".st8"
* ".std"
* ".sti"
* ".stw"
* ".stx"
* ".sxd"
* ".sxg"
* ".sxi"
* ".sxm"
* ".tex"
* ".wallet"
* ".wb2"
* ".wpd"
* ".x11"
* ".x3f"
* ".xis"
* ".ycbcra"
* ".yuv"

# Cifratura dei files



47 Prosieguo della DoingTheBigWork: creazione dei thread che criptano i files

Dopo aver fatto privilege escalation e aver lanciato i thread esaminati in precedenza, il main thread invoca una *CreateThread* in un ciclo for, in cui, ad ogni thread creato viene passato come argomento per la routine da eseguire uno dei drive individuati in precedenza tramite la *ExploreLogicalDrives*. Molto probabilmente, si tratta dei thread che andranno ad eseguire i task di cifratura. Gli handle ai thread sono salvati in un’area di memoria il cui indirizzo viene salvato in una variabile globale.

Si nota, analizzando la funzione dal decompilatore, che il thread principale si metterà in attesa degli altri thread in un for loop, invocando una *WaitForSingleObject* su ognuno di essi. Tuttavia, prima di fare ciò, effettua delle chiamate ad altre funzioni e svolge ulteriori operazioni, ma si decide di rimandarne l’analisi ad un successivo momento, preferendo concentrarsi sulla routine svolta dai thread lanciati, tentando di acquisire maggiori informazioni sul processo di cifratura.

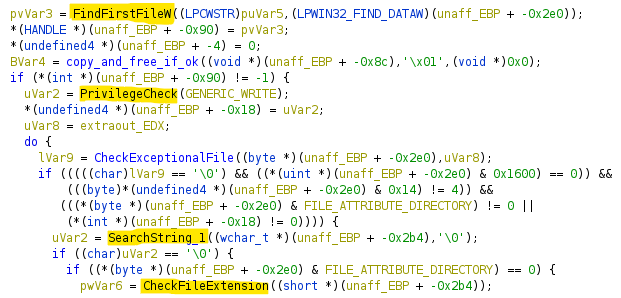


48 Routine eseguita dai thread che hanno il compito di cifrare i dati

Innanzitutto, si nota che i thread, invocando *GetCurrentThread* e *SetThreadPriority,* modificano il proprio livello di priorità lavorando in modalità background, fatta eccezione per il thread che riceve come argomento il drive con nome “C”, il quale imposta la propria priorità a THREAD\_PRIORITY\_LOWEST.

Settato ciò, viene invocata una funzione ridenominata *EnumFiles*, la quale ha il compito di preparare una lista di files che dovranno essere cifrati. Infatti, al suo interno invoca una ulteriore funzione ridenominata *WorkWithFiles*, la quale esegue le seguenti operazioni:

1. Scandisce i file e le directories del drive assegnato al thread, invocando all’inizio *FindFirstFileW* e poi, in un loop, *FindNextFileW,* così da ottenere informazioni sui files;
2. Per ogni file o directory, verifica di disporre dei privilegi d’accesso di cui necessita, invocando una funzione ridenominata *PrivilegeCheck;*
3. Se si tratta di una directory, la funzione *WorkWithFile* si invoca ricorsivamente; altrimenti, se si tratta di un file, viene invocata la *SearchString\_1* sul suo pathname;
4. Nel caso in cui nel pathname non sia presente una delle stringhe viste in precedenza analizzando la *SearchString\_1*, allora viene invocata la *CheckFileExtension*. Se l’esensione del file è tra quelle indicate nell’elenco visto in precedenza, allora il file viene aggiunto ad una lista.



49 Funzione WorkWithFiles



50 Funzione PrivilegeCheck

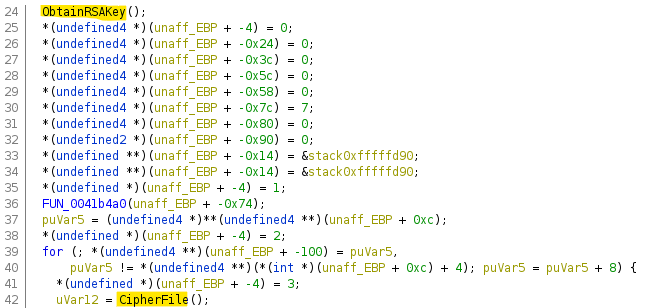
Dopo aver invocato l‘API *GetFileSecurityW* per ottenere specifiche informazioni sulla sicurezza del file o della directory, nella funzione *PrivilegeCheck,* vengono effettuate le operazioni visibili nella figura precedente per controllare che si abbiano i permessi d’accesso a tale file o directory. In particolare, ciò viene controllato invocando l’API *AccessCheck*.

Dopo aver scandito sostanzialmente tutto il drive assegnato al thread, la funzione *WorkWithFiles* ha creato una lista contenente i files da cifrare. L’indirizzo di tale lista viene salvato dalla *EnumFiles* all’interno di una classe, cui si accederà in seguito per poterli criptare.



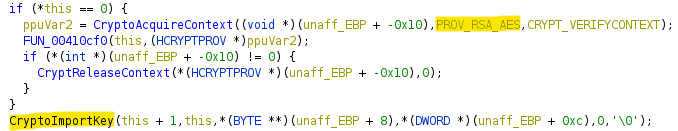
51 Lista dei file da criptare costruita

La routine del thread, a questo punto, può procedere con l’encryption dei files. La successiva funzione che viene chiamata è stata ridenominata *CryptoFiles*.



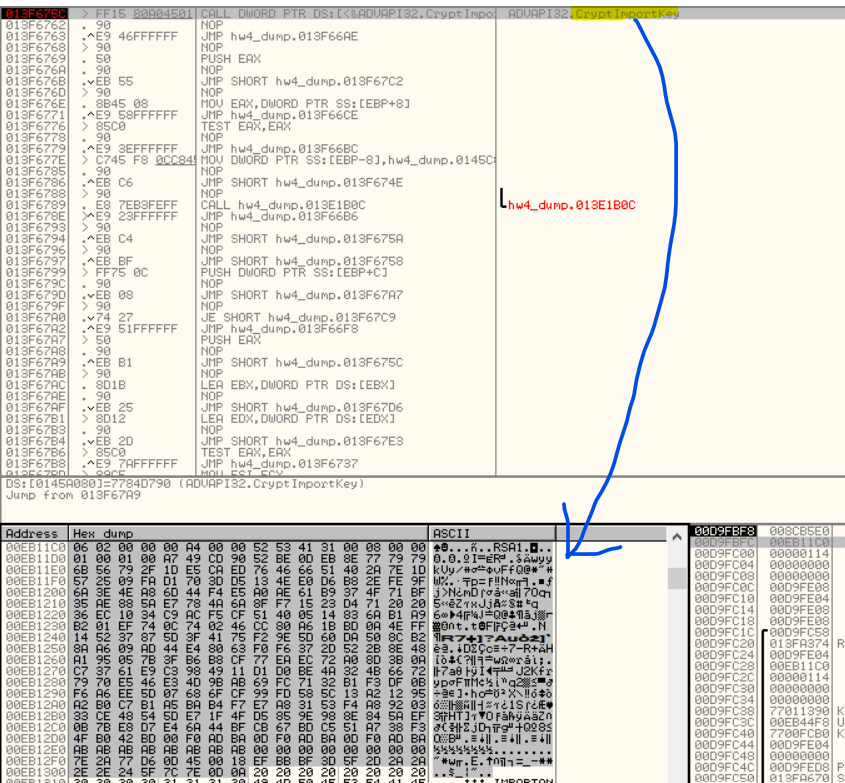
52 Inizio della funzione CryptoFiles

La prima operazione che viene effettuata è una chiamata alla funzione *ObtainRSAKey*, la quale, al suo interno, invoca le API Crypto della dll *advapi32.dll* per ottenere una chiave RSA-2048.



53 ObtainRSAKey

Viene acquisito un CSP (Crypto Service Provider) di tipo RSA\_AES, esattamente lo stesso tipo di crittografia che il malware dichiara di usare nel messaggio lasciato all’utente una volta che sono stati criptati i files. Tuttavia, la chiave non viene generata ex novo, bensì viene invocata la *CryptImportKey*, per importare nel CSP una chiave esistente sotto forma di KEY\_BLOB. Tale chiave risulta essere la seguente:



54 KEY\_BLOB della chiave RSA importata

A questo punto, in un ciclo in cui vengono scanditi i vari files, si procede a cifrarli uno alla volta, invocando la funzione *CipherFile*.

Il processo di cifratura del file si svolge nel seguente modo: come prima cosa viene invocata l’API *GetFileAttributesExW* per ottenere informazioni sul file da cifrare in una struttura di tipo WIN32\_FILE\_ATTRIBUTE\_DATA. In particolare, essa contiene: i file attributes, i timestamp di creazione del file, ultimo accesso ed ultima scrittura e infine la dimensione del file.

Dopodiché, si procede a creare il nuovo nome che verrà assegnato al file cifrato, generando delle stringhe UNICODE in maniera random. Ciò è fatto invocando due volte la funzione *GenerateRandom\_Filename\_Part*, la quale fa uso dell’API *CryptGenRandom* per generare bytes casuali, per poi mapparli nei caratteri UNICODE che rappresentano le cifre esadecimali (“0123456789ABCDEF”). Alla prima invocazione della *GenerateRandom\_Filename\_Part*, viene creata una stringa di 12 caratteri, mentre alla seconda invocazione una stringa di 8 caratteri.



55 CipherFile: gathering di informazioni sul file e creazione del filename casuale

In seguito, è invocata 3 volte la funzione *Piece\_of\_UserID\_fromASCII\_toUNICODE*, per scindere la stringa dello UserID, generata nella fase iniziale dell’esecuzione del programma dal thread main, in altrettante stringhe di caratteri UNICODE. La prima stringa contiene i primi 8 caratteri dell’ID, la seconda i caratteri dal nono al dodicesimo, la terza dal tredicesimo al sedicesimo.

Il nuovo filename è costruito mettendo all’inizio le tre parti dello UserID così costruite, separate da un trattino (“-”). Si appendono a tale stringa le due generate randomicamente in precedenza, ponendo prima quella lunga 8 caratteri e successivamente quella lunga 12, anch’esse separate da un trattino. Infine, si pone come estensione del file la stringa “.asasin”.

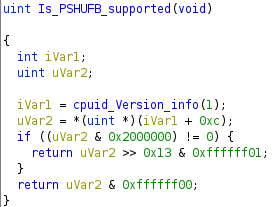
Dopodiché, il malware verifica se il file è compresso e, se è readonly, vengono modificati i suoi attributi per poterlo rendere scrivibile. Sostituisce il filename con quello precedentemente creato, invocando l’API *MoveFileExW*.



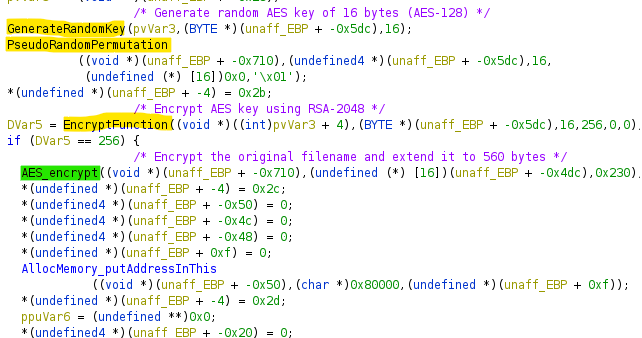
56 Parametri passati alla MoveFileExW

A questo punto, dopo aver aperto il file con permessi d’accesso in lettura e scrittura, si procede a generare una chiave AES-128 random per poter criptare il file. La chiave è generata invocando la funzione *GenerateRandomKey*, la quale usa ancora una volta l’API *CryptGenRandom*.

Successivamente, in una funzione chiamata *PseudoRandomPermutation*, effettua una permutazione casuale della chiave. Prima di farlo, verifica con una istruzione CPUID, all’interno di una funzione ridenominata *Is\_PSHUFB\_supported*, se il processore ha abilitate determinate features per poter effettuare tali operazioni a livello hardware, invocando istruzioni del tipo PSHUFB.



57 Is\_PSHUFB\_supported



58 Generazione della chaive AES e cifratura della stessa

Dopo aver generato la chiave AES, essa viene cifrata usando la chiave RSA nella funzione *EncryptFunction*, la quale fa uso dell’API *CryptEncrypt*, producendo un output di 256 bytes.

A questo punto, viene cifrato il filename originale del file invocando la funzione *AES\_encrypt* ed estendendo il risultato a 560 bytes (0x230). Molto probabilmente il malware è interessato a cifrare anche il nome originale del file oltre che il contenuto del file stesso, così da poterlo ripristinare con un programma decrypter.

Successivamente, viene allocata un’area di memoria di 0x80000 bytes in cui si scriveranno i dati letti dal file, per poterli cifrare.

La funzione *AES\_encrypt* cifra i dati che le vengono passati, ma utilizza il valore ritornato dalla *Is\_PSHUFB\_supported* per decidere come operare. Se il processore supporta operazioni AES in hardware, vengono invocate le istruzioni supportate, quali AESENC, AESENCLAST e PSHUFB. Altrimenti, la cifratura è fatta a livello software.

Immagine che contiene testo

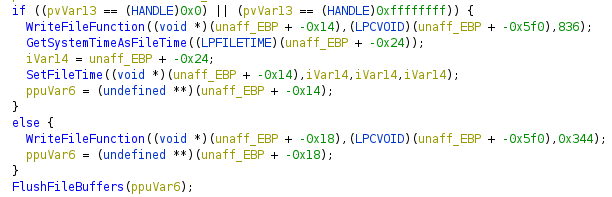
Descrizione generata automaticamente

59 Istruzioni PSHUFB e AESENC in AES\_encrypt



60 CipherFile: cifratura del contenuto del file

Come visibile dalla precedente figura, in un ciclo while si procede a leggere il file, scrivendo quanto letto nell’area di memoria precedentemente allocata (il cui indirizzo è salvato ad EBP - 0x50). Se le dimensioni del file sono maggiori dell’area di memoria, il file viene letto in più iterazioni. Ad ogni ciclo, i bytes letti vengono cifrati con AES-128, invocando la funzione *AES\_encrypt*; a questo punto, l’output della cifratura viene scritto nel file tramite la funzione *WriteFileFunction*, la quale invoca internamente l’API *WriteFile*.



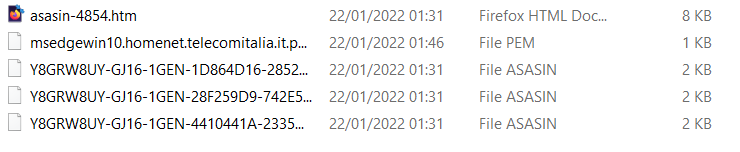
61 Scrittura nel file del nome cifrato, la chiave AES cifrata, dimensione e altri metadati

Alla fine, vengono scritti nel file altri 836 bytes (0x344) che comprendono la lungheza originaria del file, il nome del file originario cifrato con AES ed esteso a 560 bytes e la chiave AES cifrata con RSA. Inoltre, se il file non era originariamente compresso, viene settato il tempo di creazione, di ultimo accesso e di ultima scrittura del file al tempo attuale, ottenuto tramite una chiamata all’API *GetSystemTimeAsFileTime*.

Tornando alla funzione più esterna (*CryptoFiles)*, nel ciclo in cui vengono cifrati i files, viene creato un file con estensione .htm in ogni sottocartella in cui è avvenuta la cifratura. Il nome del file è generato sempre in maniera pseudo-random, essendo formato dalla stringa “asasin” separata con un trattino da una stringa di 4 caratteri UNICODE casuali. Il file contiene sempre la pagina web che mostra le informazioni che il ransomware lascia all’utente.



62 Creazione del file .htm e generazione del nome pseudo-random

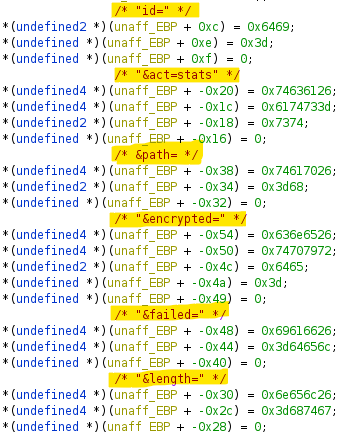


63 File .htm in una sottocartella con file cifrati

# Comunicazione con un server di controllo

Uscito dal loop della cifratura dei file sul drive, il flusso di esecuzione procede in un ramo if di un if-statement in cui il programma sembra preparare delle stringhe per una comunicazione verso un server. In particolare, sembra che il malware tenti di inviare al server delle statistiche sulla procedura di encryption appena terminata, comunicando il numero di file criptati con successo e il numero di fallimenti sullo specifico drive.

Tuttavia, l’esecuzione lancia una eccezione nel momento in cui si tenta di generare delle chiavi crittografiche random, in quanto si prova a farlo su un CSP (Crypto Service Provider) mai inizializzato. Probabilente, tali chiavi vengono usate per cifrare la comunicazione col server. In effetti, guardando i riferimenti alla variabile globale che mantiene un handle al presunto CSP, si vede che avrebbe dovuto essere inizializzato all’interno del thread main, in particolare nella funzione *DoingTheBigWork*, ma ciò non è avvenuto perché facente parte di un ramo else di un if-statement che non è stato eseguito. La condizione per entrare nel ramo else dipende da alcuni bytes letti dall’area di memoria allocata in fase di inizializzazione dalla funzione *InitData\_checkBeingDebugged*.

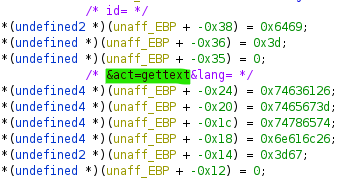


64 Stringhe da cui è stata desunta una probabile comunicazione col server

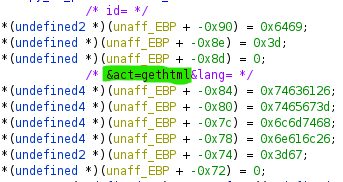
È possibile che l’area di memoria allocata dinamicamente sia stata scritta opportunamente per evitare la comunicazione con il server in seguito ad un rilevamento di esecuzione sotto debugger. Può essere sfuggita qualche misura anti-debugger. Si decide dunque di fare sniffing del traffico di rete utilizzando *Wireshark*, lanciando il malware senza il controllo del debugger su macchina virtuale. Dall’analisi non sembra che esso comunichi via rete con un server.

Si provano a seguire le istruzioni che il ransomware ha lasciato, scaricando il browser TOR e provando ad accedere all’indirizzo Web indicato. Si scopre che il server è offline o inesistente, in quanto non viene mostrata alcuna pagina Web valida.

È dunque probabile che il malware sia stato configurato appositamente per non comunicare con alcun server. In ogni caso, esso potenzialmente può farlo. Analizzando staticamente il codice, sono state trovate altre stringhe che danno un’idea di cosa il malware cerchi di fare nella comunicazione: tenta di ottenere le istruzioni per l’utente nella lingua impostata sul sistema, così come la pagina html; inoltre tenta di ottenere una chiave, probabilmente quella RSA, inviando anche informazioni sul sistema attaccato. Infine, come già visto in precedenza, per ogni drive su cui si è operato, vengono inviate statistiche sul processo di cifratura.



65 Nei commenti il significato dei bytes da interpretare come stringhe; testo localizzato



66 HTML localizzato



67 GetKey e invio di alcune informazioni sul sistema

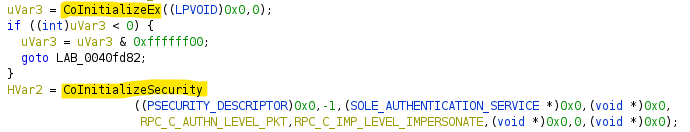
# Operazioni della DoingTheBigWork dopo aver lanciato i thread

Conclusa l’analisi dei thread incaricati di cifrare i dati sui vari drive, si ritorna ad analizzare il main thread, in particolare la funzione *DoingTheBigWork*. Come era già stato notato in precedenza, dopo aver lanciato i thread, svolge ulteriori operazioni prima di mettersi in attesa che il lavoro di encryption dei thread sia completato.

In particolare, viene effettuata una chiamata ad una funzione, dopodiché, dalle API usate e visibili dall’analisi statica, sembra accedere al registro di sistema per scrivere dei valori sotto una determinata chiave. Si decide di procedere per ordine, analizzando la funzione che viene chiamata inizialmente. Essa è stata ridenominata *BackupFunction.*



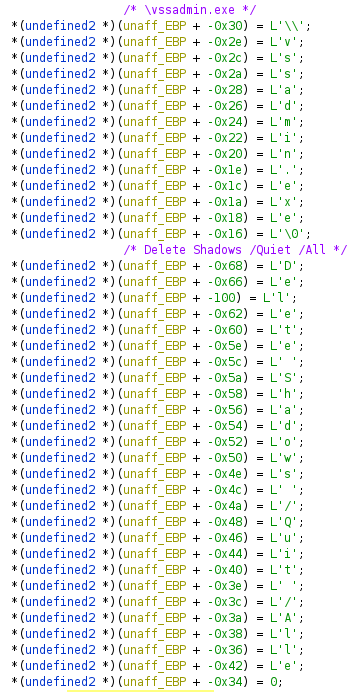
68 Funzione DoingTheBigWork: operazioni svolte prima di mettersi in attesa dei thread



69 Inizio della BackupFunction

La *BackupFunction* utilizza API della dll *ole32.dll*: usa *CoInitializeEx* per inizializzare la libreria per usare oggetti della tecnologia COM (Component Object Model). In seguito, costruisce carattere per carattere delle stringhe sullo stack, che rivelano molto dello scopo della funzione: vengono costruite le stringhe “\vssadmin.exe” e “Delete Shadows /Quiet /All”.

Sembra essere abbastanza chiaro che il malware stia cercando di eliminare le copie shadows, i backup presenti sul sistema, così da non rendere possibile un ripristino dei dati ad uno stato precedente, dopo che essi saranno stati cifrati.



70 Stringhe costruite sullo stack nella BackupFunction

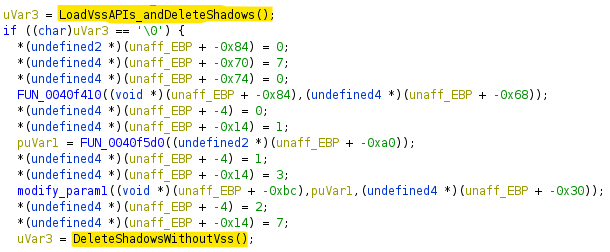
Dopo aver costruito tali stringhe, viene invocata una funzione, ridenominata *LoadVssAPIs\_andDeleteShadows*, la quale, invocando *LoadLibraryA*, carica dinamicamente la DLL *vssapi.dll*. A questo punto, tramite *GetProcAddress*, vengono ottenuti gli indirizzi delle API *CreateVssBackupComponentsInternal* e *VssFreeSnapshotPropertiesInternal*.



71 Chiamata delle API di vssapi.dll: invocazione di metodi su classi tramite offset

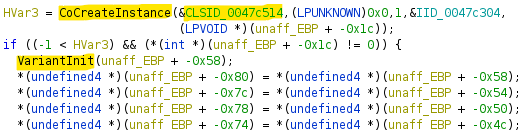
Viene invocata in primis la *CreateVssBackupComponentsInternal,* la quale crea un oggetto con interfaccia *IVssBackupComponents*. Su tale oggetto, vengono invocati dei metodi tramite offset, il che rende complicata l’individuazione dei metodi stessi, non conoscendo la struttura interna della dll. Non è d’aiuto né il debugger, né la documentazione online ufficiale (<https://docs.microsoft.com/en-us/windows/win32/api/vsbackup/nl-vsbackup-ivssbackupcomponents>). Tuttavia, dalla struttura del costrutto, si può intuire che il malware stia tentando di ottenere un riferimento a tutte le copie shadow di backup presenti nel sistema, per andare ad eliminarle una per volta all’interno di un while loop.

Ad ogni modo, seguendo il flusso di esecuzione col debugger, si nota che una delle chiamate ai metodi della classe fallisce, quindi, il flusso non riesce ad entrare nel ramo if dell’if-statement e la funzione ritorna il valore 0.



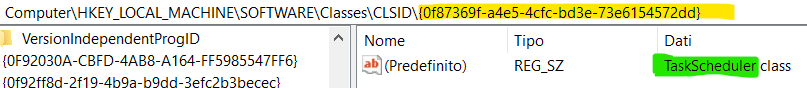
72 Eliminazione delle copie shadows senza usare le API di vssapi.dll

Se la *LoadVssAPIs\_andDeleteShadows* fallisce, ritornando 0, il programma prova ad eliminare le copie di backup in altro modo, nella funzione *DeleteShadowsWithoutVss*, in cui viene invocata come segue la *CoCreateInstance*, con un CLSID hardcoded:



73 Invocazione a CoCreateInstance, con un CLSID hardcoded (variabile globale)

Cercando il CLSID tra le chiavi del registro di sistema, si riesce ad intuire il tipo dell’oggetto che si sta cercando di inizializzare e risulta essere un oggetto di tipo TaskScheduler.



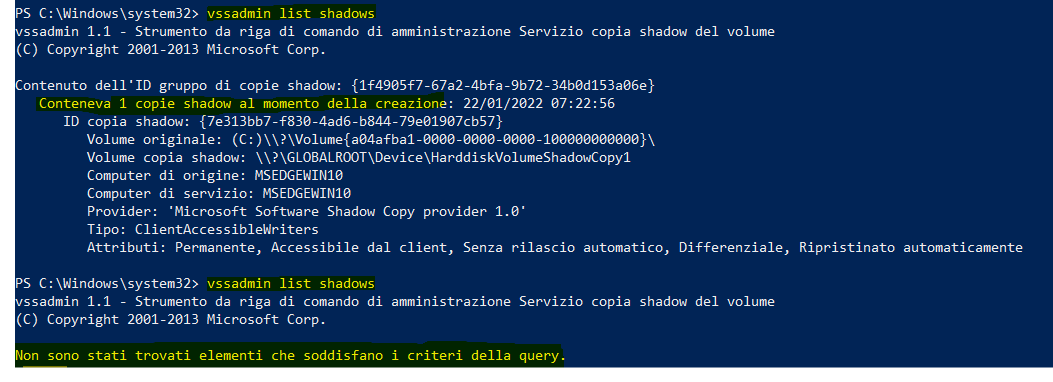
74 CLSID e corrispondente classe nel registro di sistema

Nel prosieguo della funzione *DeleteShadowsWithoutVss* vengono invocate le API della libreria *oleaut32.dll*, accedute tramite ordinale. Tuttavia, anche sull’oggetto di tipo TaskScheduler vengono invocati dei metodi tramite offset che non si riescono a riconoscere facilmente. Dal debugger, però, si nota che ad alcuni di essi vengono passate come parametri le stringhe costruite in precedenza sullo stack: in particolare, viene prima invocato un metodo passando come parametro la stringa “vssadmin.exe” e successivamente se ne invoca un secondo passandogli la stringa “Delete Shadows /Quiet /All”.

Molto probabilmente sarà stato lanciato un task che esegua tale comando.

Per verificare empiricamente che le assunzioni fatte siano veritiere, è stata generata una copia shadow sulla VM e se ne è verificata l’esistenza da terminale tramite il comando “vssadmin list shadows”. Dopodiché, lanciando il malware con i permessi di amministratore, si è atteso che svolgesse il suo lavoro, per poi verificare se la copia di backup fosse ancora presente nel sistema o meno.

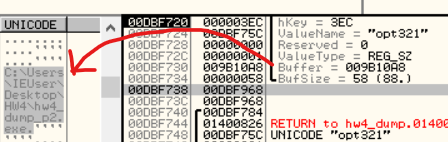
Effettivamente, la copia di backup è stata eliminata dal malware. Se però quest’ultimo non esegue con i permessi di amministratore, allora non è in grado di cancellare le copie shadows.



75 Esecuzione del comando "vssadmin list shadows" prima e dopo l'esecuzione del malware

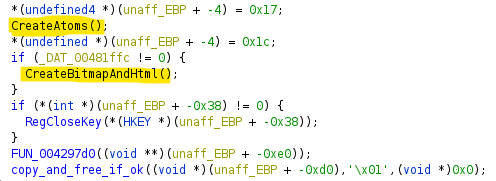
Per quanto riguarda l’accesso al registro di sistema, il codice visto in precedenza sembra essere dead code, in quanto non viene eseguito. Tuttavia, nel caso in cui venisse eseguito, è facile vedere, analizzandolo staticamente, che il programma tenta di aprire la chiave HKCU\Software\Microsoft\Windows\CurrentVersion\Run, tentando di scrivere un valore sotto il nome “opt321”. Inoltre, dopo l’esecuzione dei thread che cifrano i dati, tale valore viene rimosso dal registro di sistema.

Forzando l’esecuzione con il debugger, si vede che il valore associato non è altro che la stringa del pathname del file eseguibile del malware.



76 Valore scritto nel registro di sistema

In seguito, nella *DoingTheBigWork*, vengono invocate altre due funzioni, le quali sono state ridenominate *CreateAtoms* e *CreateBitmapAndHtml*.



77 Invocazioni a CreateAtoms e CreateBitmapAndHtml



78 Funzione CreateAtoms

La funzione *CreateAtoms* costruisce una stringa utilizzando lo UserID e la stringa “~~~” nel seguente modo: alla stringa “~~~” viene fatta una append dell’identificatore e poi nuovamente una append della stringa “~~~”.

A questo punto, la stringa così costruita viene aggiunta alle tabelle atom locali e globali, assegnandole così due ATOM che la identificano.



79 Parametro passato alla AddAtomA

Molto probabilmente, questa operazione è effettuata dal malware per controllare se ha già operato sul sistema su cui sta eseguendo. In effetti, tutta l’esecuzione fin qui descritta, non avviene se la creazione dei mutex (o eventi) usati per la sincronizzazione dei thread fallisce oppure se si riescono a trovare degli ATOMs associati alla stringa appena descritta, invocando la funzione *SearchAtom*.



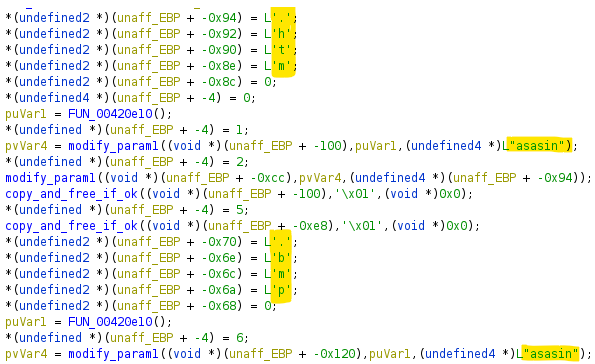
80 Chiamata alla funzione SearchAtom



81 Funzione SearchAtom

La funzione *SearchAtom*, infatti, verifica prima nella tabella degli atom globali se esiste un atom associato alla stringa costruita; se non lo trova, lo cerca nella tabella degli atom locali. Nel caso in cui tale atom viene trovato, la funzione ritorna true, altrimenti false. Quindi, se l’atom viene trovato, si segue il flusso dell’istruzione *goto*, e si va direttamente a terminare l’esecuzione.

Si procede con l’analisi della funzione *CreateBitmapAndHtml*



82 Nomi dei files htm e bmp che vengono creati

In questa funzione, vengono creati due files sul Desktop contenenti le informazioni da seguire per poter ottenere un programma di decifratura dei dati in cambio di un pagamento in cryptocurrency. I due file sono in formato .htm e .bmp; il loro filename è rispettivamente “asasin.htm” e “asasin.bmp”, come visibile anche da debugger.



83 Filename del file htm



84 Filename della bitmap

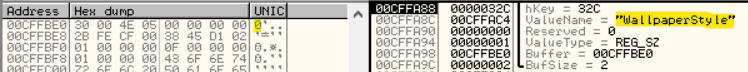
La bitmap viene creata all’interno di una funzione chiamata *CreateBitmap*, la quale fa uso di API della DLL *gdi32.dll*, tra cui *CreateFontA*, *CreateSolidBrush*, *CreateCompatibleDC*, *CreateCompatibleBitmap*, *SetTextColor*, *SetBkMode*.

A questo punto, il programma accede a chiavi di registro di sistema per poter scrivere dei valori.



85 Accesso alla chiave HKCU\Control Panel\Desktop e scrittura dei valori WallpaperStyle e TileWallpaper

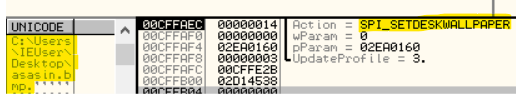
Viene aperta la chiave HKCU\Control Panel\Desktop e vengono modificati i valori WallpaperStyle e TileWallpaper, impostandoli a 0, come visibile dal debugger. Infine, viene invocata l’API *SystemParametersInfoW* passandole come primo parametro SPI\_SETDESKWALLPAPER per impostare la bitmap creata in precedenza come sfondo del desktop.



86 Impostazione del valore WallpaperStyle

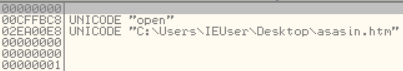


87 Impostazione del valore TileWallpaper



88 Invocazione di SystemParametersInfoW: settaggio della bitmap come sfondo del desktop

I due files creati sul desktop vengono aperti con due successive invocazioni all’API *ShellExecuteW*, passandole come stringa comando “open”. In questo modo, l’utente vede apparire a schermo la bitmap e vede aprirsi un browser con la pagina web.



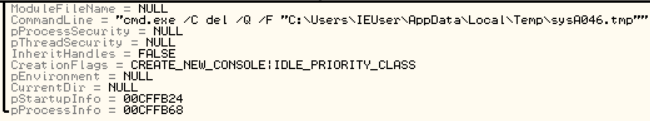
89 Parametri passati alla ShellExecuteW per aprire il file htm

Infine, come ultima operazione, il programma chiama una funzione ridenominata *TerminationEpilogue*, in cui, invocando *GetModuleFileNameW* ottiene il proprio filepath e con una chiamata a *GetTempPathW* ottiene il path designato sul sistema per i files temporanei (e.g. “C:Users\<NomeUtente>\AppData\Local\Temp”). Dopodiché, invocando l’API *SetFileAttributeW*, setta il file attribute del proprio eseguibile a FILE\_ATTRIBUTE\_NORMAL. A questo punto, invocando *GetTempFileNameW*, passando come primo parametro il path della directory contenente i files temporanei e come stringa prefisso “sys”, viene creato in tale cartella un file temporaneo (e.g “sysA046.tmp”).

Invocando una *MoveFileExW* con flag MOVEFILE\_REPLACE\_EXISTING | MOVEFILE\_WRITE\_THROUGH, viene spostato il contenuto dell’eseguibile del malware all’interno del file temporaneo.

Dopodiché, viene invocata nuovamente una *MoveFileExW*, passando stavolta come primo parametro il path del file temporaneo e come flag MOVEFILE\_DELAY\_UNTIL\_REBOOT: in questo modo, il sistema registra il file temporaneo come un file da cancellare al prossimo riavvio del sistema stesso.

Infine, lancia un nuovo processo invocando la *CreateProcessW* con i seguenti parametri:



90 Parametri della CreateProcessW: comandi nel parametro CommandLine

Tale processo ha il compito di cancellare in modalità non interattiva (/Q) e forzando l’eliminazione di file di sola lettura (/F) il file temporaneo creato precedentemente.

Infine il malware termina, invocando *ExitProcess(*0*)*.

È interessante notare che la funzione appena descritta, che fa terminare il processo, viene invocata all’inizio, prima di eseguire l’intero processo di scanning dei drives e di cifratura dei files (sostanzialmente prima che il malware effettui operazioni malevole) nel caso in cui la lingua di default del sistema o quella di default per l’utente sia il Russo.

# Conclusione e riepilogo dell’analisi

Il risultato dell’analisi effettuata è stato innanzitutto confermare che il file eseguibile “hw4.ex\_” sia un file malevolo, in particolare un ransomware, noto come *Locky*, che attacca il sistema della vittima cifrandone i files e richiedendo un pagamento in criptovalute per poter ottenere un decrypter. Inoltre, sono state ottenute molteplici altre informazioni, tra cui il fatto che l’eseguibile è impacchettato.

È ora possibile rispondere a gran parte delle domande cui si era interessati a dare una risposta all’inizio del processo di analisi:

1. Determinare quali files e quali drives vengono cifrati e attaccati dal malware:
   * Il ransomware tenta di cifrare dischi fissi, dischi remoti, RAM disk e drive rimovibili; inoltre, evita di cifrare files che abbiano le stringhe “Windows”, “Boot”, “System Volume Information”, “$Recycle.Bin”, “thumbs.db”, “temp”, “Program Files”, “Program Files (x86)”, “AppData”, “Application Data”, “winnt”, “tmp”, “\_Locky\_recover\_instruction.txt”, “\_Locky\_recover\_instruction.bmp”, “\_HELP\_instructions.txt”, “\_HELP\_instructions.bmp” e “\_HELP\_instructions.html” nel loro pathname. Inoltre, una lista di tutte le estensioni di file che il ransomware cifra è stata presentata in precedenza;
2. Determinare come ogni file viene cifrato e con quali chiavi di cifratura, se utilizzate e se individuabili:
   * Come dichiarato dal malware stesso, usa RSA-2048 e AES-128 per cifrare i file. Viene lanciato un thread per ogni drive su cui bisogna effettuare operazioni di cifratura. Ogni file, così come il suo filepath originario, è cifrato con una chiave AES generata randomicamente per ogni file. Viene cifrata anche la chiave AES, utilizzando la chiave RSA ed il tutto è salvato all’interno del “nuovo” file criptato, che avrà come nome l’ID generato per l’utente e ulteriori caratteri casual. La nuova estensione dei file cifrati è “.asasin”.
3. Come il malware riesce a fare privilege escalation:
   * Il malware acquisisce privilegi di debug, di backup sia in lettura che scrittura e privilegi per poter prendere possesso di files, invocando l’API *AdjustTokenPrivileges* per costruire un token con i privilegi appena descritti per il proprio processo.
4. Capire come viene generato l’ID dell’utente:
   * L’ID dell’utente è generato facendo l’hash MD5 del GUID del sistema e mappandolo in una stringa ASCII; dopodiché, si prendono solo alcuni caratteri di tale stringa e si genera l’ID effettuando ulteriori operazioni e basandosi su parametri del sistema, quali la lingua di default, la versione del Sistema Operativo, le metriche del sistema.
5. Capire se eventuali chiavi di cifratura sono generate localmente od ottenute connettendosi alla rete:
   * Come già risposto parzialmente al punto 2, le chiavi AES sono generate casualmente per ogni file da cifrare e in maniera locale al sistema. Per quanto riguarda la chiave RSA, il malware sembra essere stato scritto per poter ottenerla da un server, tuttavia, non comunicando con alcun server, la chiave RSA sembra essere hardcoded nell’eseguibile.
6. Capire se il malware tenta di evadere all’analisi:
   * Il malware, se si accorge di essere lanciato sotto il controllo di un debugger, tenta di evadere all’analisi, terminando l’esecuzione dopo essersi lanciato sotto il nome di “svchost.exe”. Inoltre, sono presenti ulteriori misure anti-debugger che bloccano l’esecuzione del programma, come la *Sleep* invocata con un parametro negativo, o che ostacolano l’analisi, come l’utilizzo di thread con un meccanismo di sincronizzazione.
7. Capire se il malware, dopo aver operato, rimane persistente nel sistema e, se sì, se si nasconde:
   * Il malware, dopo aver eseguito, cancella il suo eseguibile originale, per spostarsi sotto il nome di un file temporaneo nella cartella designata dal sistema per contenere files temporanei. Inoltre, viene programmata una cancellazione anche di quest’ultimo al prossimo reboot del sistema. In ogni caso, il malware lascia una traccia della sua esecuzione, salvando una stringa contenente l’ID univoco dell’utente nelle tabelle ATOM globali e locali.
8. Comprendere come il malware ottiene la pagina html che viene mostrata e la bitmap. Inoltre, capire come fa ad impostare la bitmap come sfondo del desktop:
   * L’html è hardcoded, viene semplicemente sostituito un placeholder con l’ID dell’utente. In realtà, il malware sembra in grado di ottenere dal server la pagina html localizzata nella lingua dell’utente o del sistema, tuttavia, ciò non avviene. La bitmap è costruita a partire dal messaggio di istruzioni per l’utente, utilizzando le API della DLL *gdi32.dll.* Per impostarla come sfondo del desktop, viene acceduto il registro di sistema per impostare a 0 i valori TileWallpaper e WallpaperStyle sotto la chiave HKCU\Control Panel\Desktop. Inoltre, viene effettuata una invocazione all’API *SystemParametersInfoW*.
9. Capire se il malware comunica con un server di controllo:
   * Facendo sniffing del traffico di rete relativo al processo del malware, non è stata rilevata alcuna comunicazione in rete con un server. Tuttavia, dall’analisi statica avanzata, il malware risulta essere in grado di comunicare con un server di controllo, in particolare per ottenere una chiave (molto probabilmente di cifratura, RSA), il testo del messaggio e l’html localizzati e per inviare statistiche sul processo di encryption di ogni drive. Si presume che l’eseguibile “hw4.ex\_” sia stato configurato per agire senza cimunicazioni in rete con alcun server.