**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

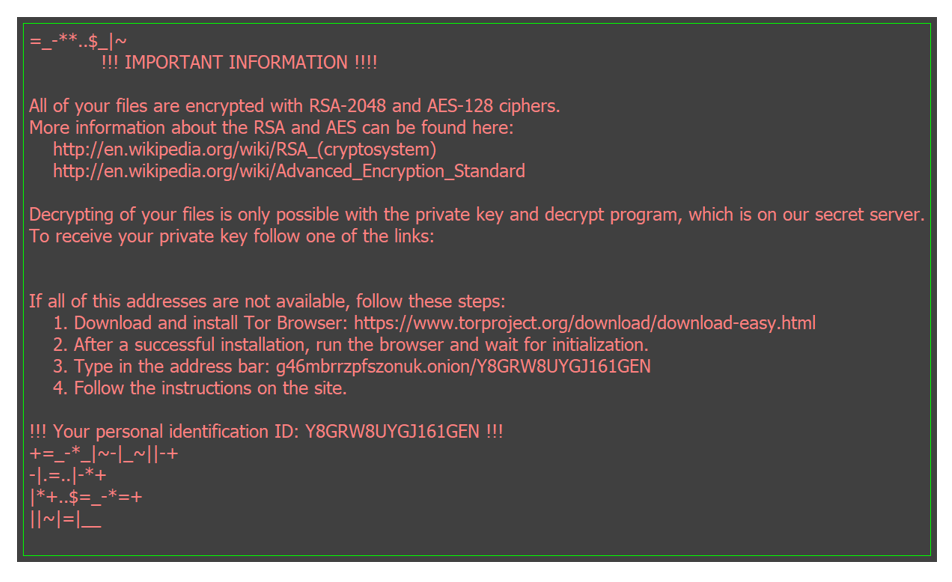
Descrizione generata automaticamente

1 Analisi di hw4.ex\_ con PeStudio

Essendo che l’eseguibile è impacchettato, si prova a farne il depacking. 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 “.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.

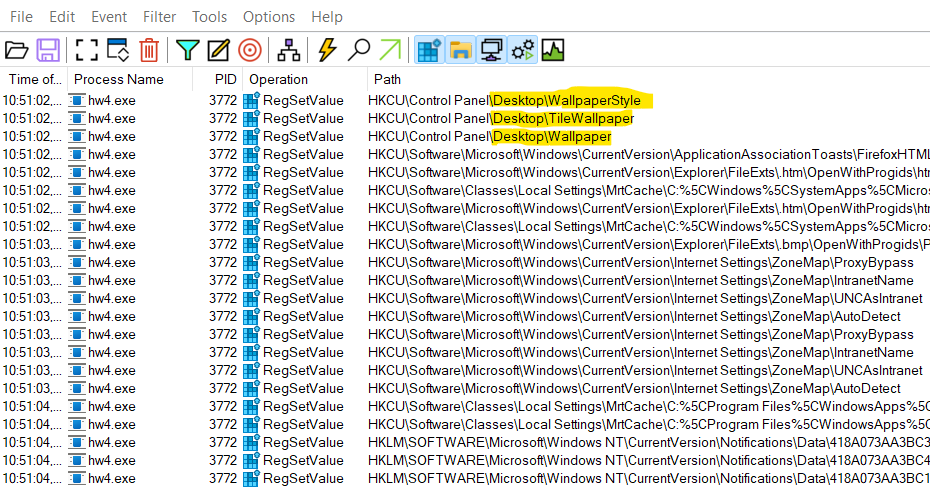
Analizzando il file con *PeID*, innanzitutto non viene rilevato l’utilizzo di alcun packer; tuttavia, ciò sembra molto improbabile. Utilizzando il plugin per l’individuazione dell’OEP del programma, l’eseguibile viene lanciato e, nonostante non venga dato alcun OEP come risultato, il malware ha operato 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.

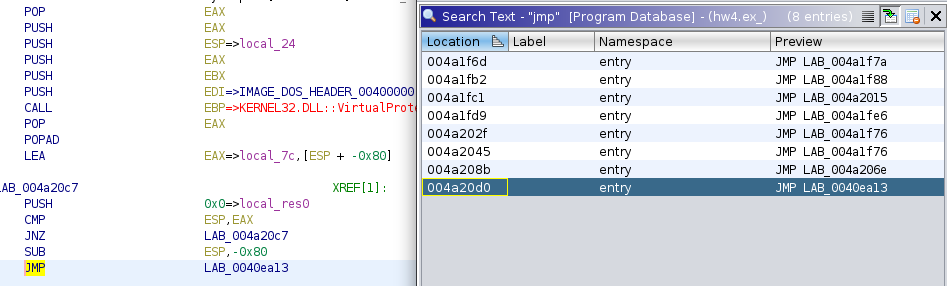
Prima di procedere con i tentativi di depacking, si tenta di fare anche analisi dinamica di base, utilizzando *ProcessExplorer* e *ProcessMonitor*, insieme a *RegShot*. Tuttavia, il malware compie numerosissime operazioni e accede moltissime volte al registro di sistema, anche se il più delle volte probabilmente a causa di API che utilizza. Filtrando per le operazioni di 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.



3Analisi con ProcessMonitor. Filtraggio per RegSetValue

# Unpacking

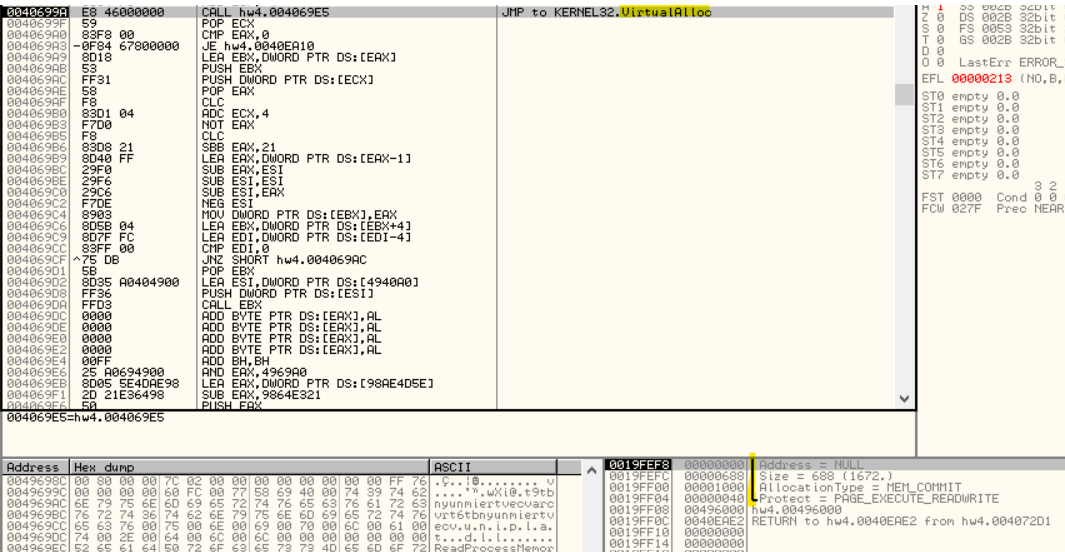
Per poter analizzare il vero malware, è necessario spacchettarlo. Si tenta di ottenere l’OEP guardando le istruzioni di jump dal disassembler di *Ghidra*, per poter individuare possibili tail jump.



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 non contenente 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 allocara memoria dinamicamente, in cui 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:

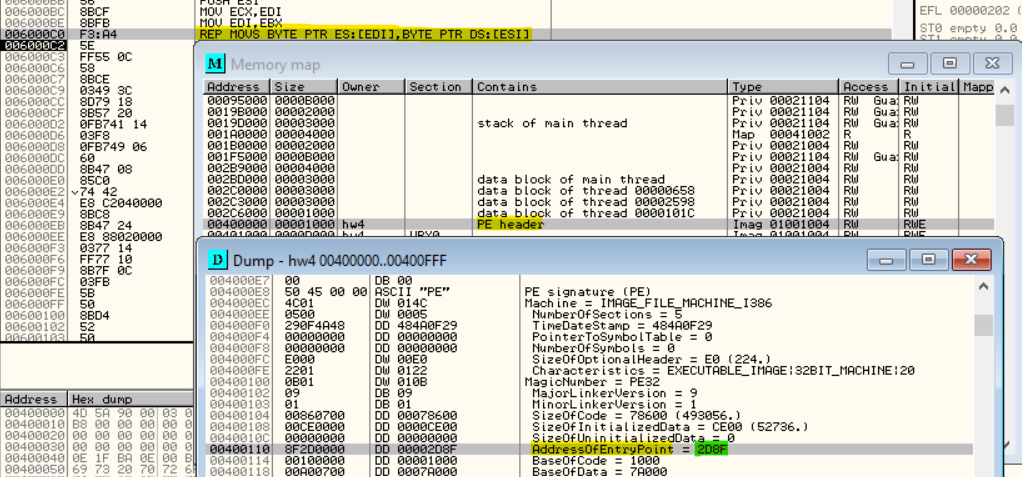


6 Nomi corretti delle sezioni: text, rdata, data, reloc, cdata

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. Il tutto funziona correttamente.

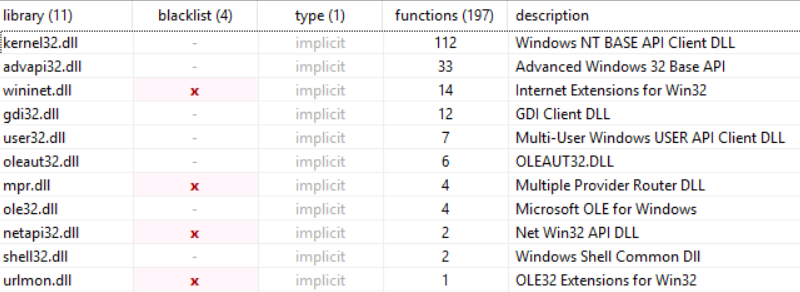
# Nuovi Obiettivi

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
* Capire se il malware è in grado di diffondersi verso altre macchine

# Analisi di base ulteriore

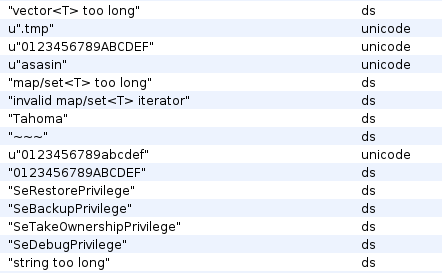
Analizzando il dump precedentemente ottenuto con PeStudio, si vede che sono importate le seguenti DLL:



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

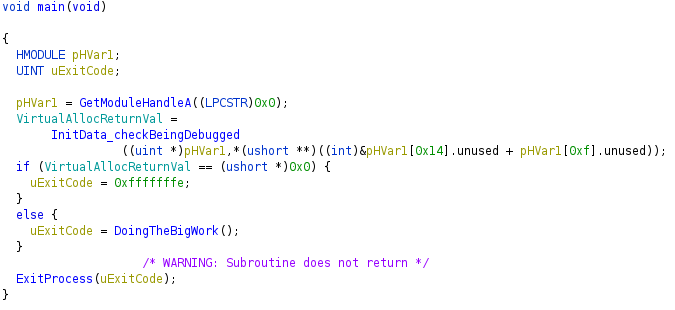
È interessante notare la presenza della stringa “asasin” in formato UNICODE, che è quella utilizzata come estensione dei file criptati. Inoltre, è importante notare la presenza delle stringhe “SeRestorePrivilege”, “SeBackupPrivilege”, “SeTakeOwnershipPrivilege” e “SeDebugPrivilege”, le quali potrebbero 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.

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

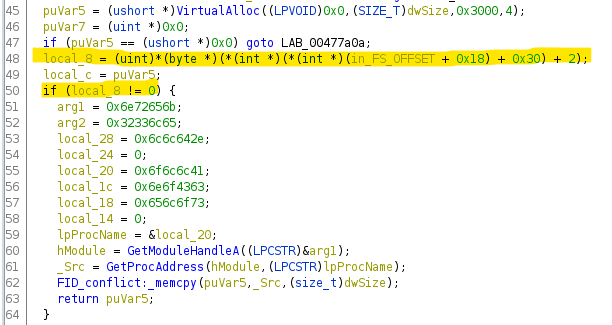
Aprendo il file dumpato con il disassembler di *Ghidra*, si è partiti dalla entry e si è subito individuata la funzione main del programma, essendo una delle ultime ad essere chiamata da entry. Il codice disassemblato è molto contorto da seguire, a causa delle numerose istruzioni di jump che lo rendono 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. Si nota che tale 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: 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

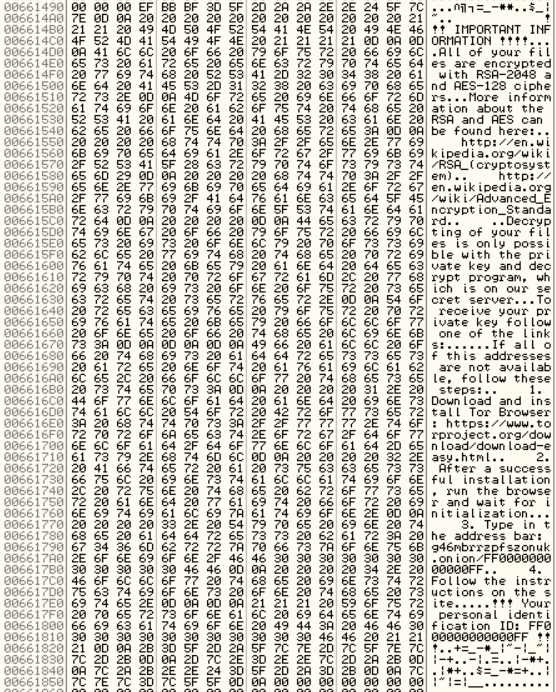
Tramite debugger, modificando l’istruzione che controlla se il valore di BeingDebugged è diverso da zero (facendo sì che risulti zero), si vede che l’area di memoria allocata contiene diverse informazioni: in primis, c’è un’area popolata di byte a zero, fatta eccezione per tre bytes; ci sono poi altri 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.



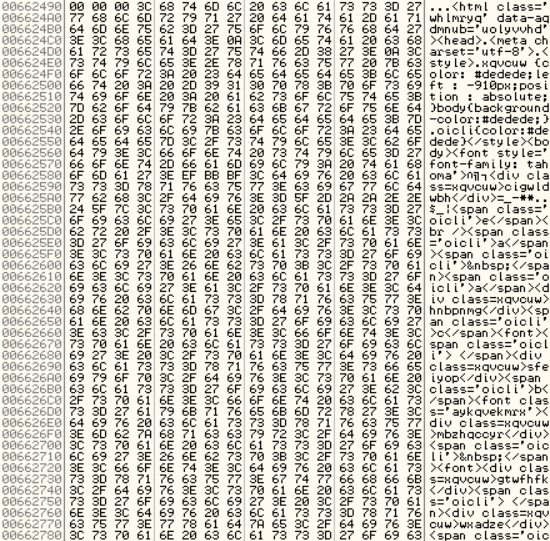
12 Prima area popolata: sono evidenziati gli unici bytes diversi da zero



Seconda area, contenente la stringa "RSA1"



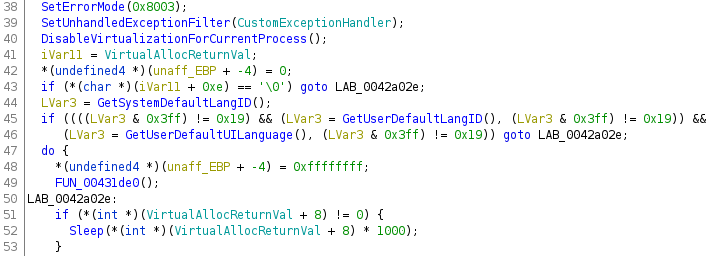
Terza area: è presente il messaggio di istruzioni da lasciare all'utente



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

Inoltre, un byte ad offset 0x6490 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.



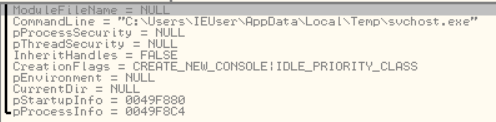
17 Sleep invocata con valore negativo

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

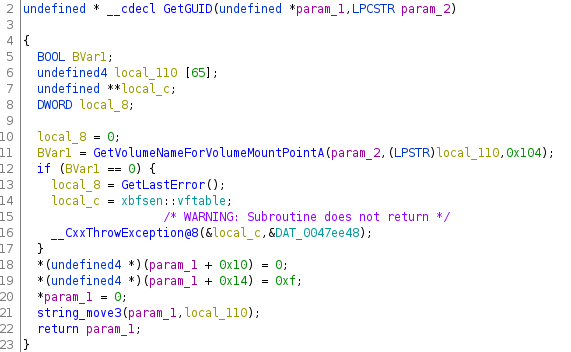
# Prima patch

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 a partire dalla directory Windows del sistema 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 tali eccezioni al programma, si riesce ad arrivare al punto in cui l’API viene invocata con parametro la stringa “C:\” e restituisce il nome del volume.



20 Ramo if in DoingTheBigWork



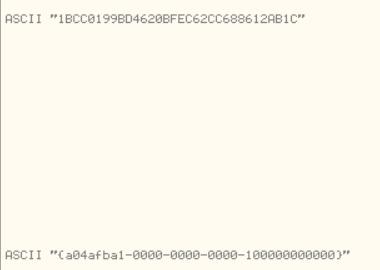
21 Chiamata a GetVolumeNameForVolumeMountPointA



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

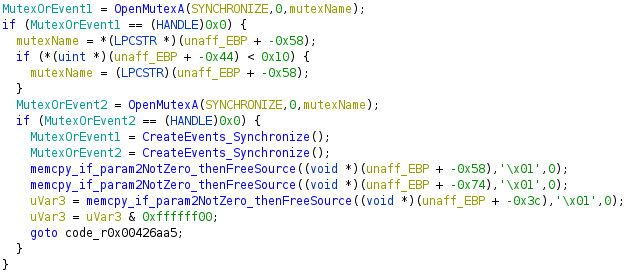


23 Volume name ottenuto



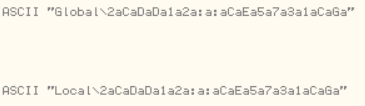
24 In basso, la parte del volume name su cui si calcola l'MD5. In alto, il risultato convertito in ASCII

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



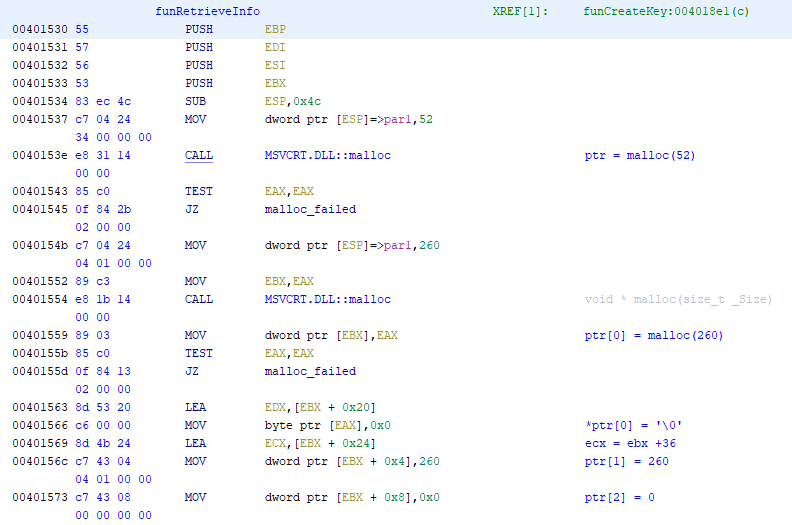
27 Funzione CreateEvents\_Synchronize, usata per creare eventi da usare come mutex

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.*



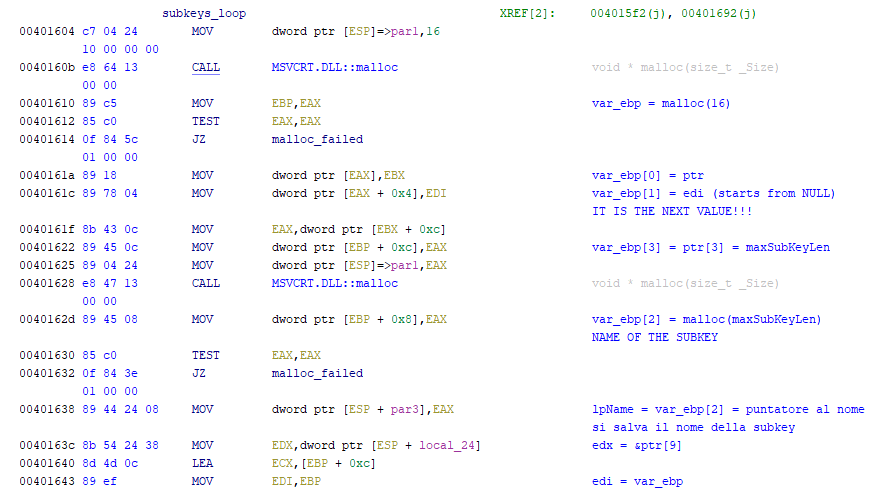
28 UserID generato

# Individuazione delle strutture di dati

La funzione funRetrieveInfo prende come unico parametro di input un handle ad una chiave di registro, che, come visto in precedenza, sarà l’handle della nuova chiave aperta dalla RegOpenKeyExA. La prima operazione effettuata all’interno della funzione è una chiamata alla funzione di libreria “malloc” passandole come parametro il valore 52. Successivamente, i primi 4 bytes dell’area di memoria appena allocata nello heap vengono occupati da un ulteriore puntatore ad una nuova area di memoria contigua di 260 bytes, allocata sempre utilizzando la malloc, di cui il primo byte viene inizializzato con il carattere terminatore di stringa ‘\0’. I secondi 4 bytes dell’area di memoria da 52, sono inizializzati con il valore intero 260, proprio la taglia dell’altra area di memoria allocata; invece, i 4 bytes di memoria successivi sono impostati a 0. Si riporta di seguito un’immagine del punto nel listato dell’eseguibile in *Ghidra* dove tali informazioni sono state desunte:

Dunque, viene allocata una porzione contigua di 52 bytes in cui il primo campo è un puntatore ad un array di caratteri e i successivi due sono valori interi, quindi tipi di dato differenti. Ciò inizia a far pensare che i 52 bytes possano essere la taglia di un tipo di dato *struct* definito dal programmatore. Tale ipotesi trova ulteriori conferme andando avanti nell’analisi delle istruzioni assembly della funzione: infatti, vengono caricati sullo stack gli indirizzi delle locazioni di memoria dei vari campi della presunta *struct*, ottenendoli tramite offset dall’indirizzo base, per poterli passare come parametri alla funzione di libreria RegQueryInfoKeyA. Tale funzione prende come unico parametro di input un handle ad una chiave di registro di cui ne recupera le informazioni e le restituisce agli indirizzi passati come parametri di output. Utilizzando la documentazione, sono stati individuati i tipi di dato di alcuni dei campi della *struct*, che è stata definita anche in *Ghidra* per agevolare l’analisi, assegnandole il nome “struct\_query\_info”:

|  |  |  |  |
| --- | --- | --- | --- |
| **Bytes** | **Tipo di dato** | **Nome campo** | **Descrizione** |
| 4 | LPSTR | className | Classe user-defined della chiave |
| 4 | DWORD | classNameLen | Lunghezza del className |
| 4 | DWORD | numSubKeys | Numero di sottochiavi della chiave |
| 4 | DWORD | maxSubKeyLen | Lunghezza della più lunga delle sottochiavi |
| 4 | DWORD | maxClassLen | Lunghezza della più lunga stringa che specifica la classe di una sottochiave |
| 4 | DWORD | numValues | Numero di values della chiave |
| 4 | DWORD | maxValueNameLen | Lunghezza del più lungo nome di un valore |
| 4 | DWORD | maxValueLen | Dimensione del più lungo componente dati tra i valori della chiave, in bytes |
| 4 | DWORD | securityDescriptor | Lunghezza del security descriptor della chiave |
| 8 | FILETIME | lastWriteTime | Struttura che contiene il tempo in cui la chiave o uno dei suoi valori sono stati modificati l’ultima volta |
| 8 | undefined | ??? | ??? |

Ci sono ancora 8 bytes da ricoprire, ma, di fatto le *Sub-keys* e i *Values* non sono ancora stati recuperati, quindi questi 8 bytes potrebbero fare riferimento a tali dati. Infatti, continuando ad analizzare le istruzioni assembly della funzione funRetrieveInfo, con l’ausilio del tool *Function Graph* di *Ghidra*, si vede che viene valutato se il campo numSubKeys è maggiore di zero e, in tal caso, si entra in un ciclo while in cui, ad ogni iterazione, viene chiamata la funzione di libreria RegEnumKeyExA, per recuperare informazioni su una singola sottochiave ogni volta che viene invocata. Prima di invocarla, sempre nel ciclo while, viene effettuata una malloc di 16 bytes passandone gli indirizzi a offset di 8 e 12 bytes alla RegEnumKeyExA per salvare le informazioni prodotte in output, in particolare il nome della sottochiave e la sua lunghezza. Inoltre, i primi 4 bytes dei 16 allocati sono un puntatore alla *struct* di tipo struct\_query\_info allocata in precedenza, mentre i successivi 4 vengono settati con il riferimento alla area di memoria da 16 bytes allocata nella precedente iterazione del loop. Rappresentano quindi un campo next di una struttura dati di tipo lista collegata. Di seguito è riportata un’immagine delle istruzioni assembly in *Ghidra* che hanno permesso tali deduzioni:

Quindi, viene individuata una ulteriore *struct* che rappresenta un nodo della lista contenente informazioni su una sottochiave; tale struttura è stata denominata “struct\_subkey”:

|  |  |  |  |
| --- | --- | --- | --- |
| **Bytes** | **Tipo di dato** | **Nome campo** | **Descrizione** |
| 4 | struct\_query\_info\* | queryInfo | Puntatore alla struttura con le informazioni ottenute dalla query sulla chiave |
| 4 | struct\_subkey\* | next | Puntatore al successivo nodo nella lista di sottochiavi |
| 4 | LPSTR | subKeyName | Puntatore al nome della sottochiave |
| 4 | DWORD | nameLen | Lunghezza del nome della sottochiave |

La lista viene costruita effettuando degli inserimenti in testa: ogni nuovo nodo inserito viene fatto puntare alla testa della lista e, di conseguenza, diviene esso stesso la nuova testa della lista. Qualora la RegEnumKeyExA restituisca un valore di errore, il nodo appena inserito in testa alla lista viene rimosso e il suo successore diviene nuovamente la testa della lista.

Infine, una volta terminate le iterazioni nel while loop, il penultimo campo della struttura “struct\_query\_info” viene settato con il puntatore alla testa della lista di sottochiavi, mantenendo così un riferimento alla lista di *Sub-keys*.

In maniera analoga, in un successivo while loop vengono recuperati dati riguardanti i *Values* invocando la funzione di libreria RegEnumValueA e viene costruita un’altra lista. Anche in questo caso è stata individuata una *struct* per i nodi della lista, denominata “struct\_regValue”:

|  |  |  |  |
| --- | --- | --- | --- |
| **Bytes** | **Tipo di dato** | **Nome campo** | **Descrizione** |
| 4 | struct\_query\_info\* | queryInfo | Puntatore alla struttura con le informazioni ottenute dalla query sulla chiave |
| 4 | struct\_regValue\* | next | Puntatore al successivo nodo nella lista di valori |
| 16384 | CHAR[16384] | valueName | Buffer contenente il nome del valore |
| 4 | DWORD | valueNameLen | Lunghezza del nome del valore |
| 4 | DWORD | type | Codice che identifica il tipo dei dati conservati nel valore |
| 256 | BYTE[256] | data | Buffer contenente i dati del valore |
| 4 | DWORD | dataLen | Lunghezza in bytes dei dati |

Analogamente, la lista viene costruita con degli inserimenti in testa e il riferimento alla testa della lista è assegnato all’ultimo campo della *struct* “struct\_query\_info”, la quale adesso risulta completa. Si riporta di seguito una definizione in linguaggio C delle *struct* citate in precedenza, così da ricapitolare le informazioni ottenute:

**typedef** **struct** struct\_query\_info struct\_query\_info**;**

**typedef** **struct** struct\_subkey struct\_subkey**;**

**typedef** **struct** struct\_regValue struct\_regValue**;**

**struct** struct\_query\_info**{**

LPSTR className**;**

DWORD classNameLen**;**

DWORD numSubKeys**;**

DWORD maxSubKeyLen**;**

DWORD maxClassLen**;**

DWORD numValues**;**

DWORD maxValueNameLen**;**

DWORD maxValueLen**;**

DWORD securityDescriptor**;**

FILETIME lastWriteTime**;**

struct\_subkey **\***listOfSubKeys**;**

struct\_regValue **\***listOfValues**;**

**};**

**struct** struct\_subkey**{**

struct\_query\_info **\***queryInfo**;**

struct\_subkey **\***next**;**

LPSTR subKeyName**;**

DWORD nameLen**;**

**};**

**struct** struct\_regValue**{**

struct\_query\_info **\***queryInfo**;**

struct\_regValue **\***next**;**

CHAR valueName**[**16384**];**

DWORD valueNameLen**;**

DWORD type**;**

BYTE data**[**256**];**

DWORD dataLen**;**

**};**

La funzione funRetrieveInfo restituisce al chiamante il puntatore alla struttura di dati di tipo struct\_query\_info allocata. A sua volta, la funzione funCreateKey, lo restituisce al main dopo aver invocato la RegCloseKey. Infine, questo puntatore viene passato in input alla funPrintResults per poter proseguire con la stampa dei dati recuperati.

In particolare, vengono stampati su standard output:

* Il nome della classe user-defined della chiave;
* La lunghezza in bytes del security descriptor della chiave;
* Il valore temporale dell’ultima modifica delle sottochiavi o dei valori;
* Per ogni sottochiave, navigando la lista, ne viene stampato il nome;
* Per ogni valore, navigando la lista, ne viene stampato: il nome, il codice rappresentante il tipo di valore e, se presenti, i dati contenuti, sia come bytes che come stringa (tipicamente, un valore contiene dati in formato stringa o sequenze di bytes, ma talvolta può contenere anche un intero a 32 bit, un gruppo di stringhe, una stringa con riferimenti a variabili d’ambiente o un valore amorfo).

# Analisi approfondita della funzione main

Come ultimo passo del processo di reversing, vengono analizzati dei dettagli della funzione main che non sono stati approfonditi inizialmente. Innanzitutto, come prima operazione, il main invoca una funzione ridenominata “fun1”. Tale funzione è invocata sia dal main ma anche dalla entry. Inoltre, il suo comportamento è determinato dal valore di una variabile globale di tipo intero: se il valore di tale variabile è diverso da zero, allora la fun1 non fa nulla e restituisce il controllo al chiamante; invece, se il valore della variabile globale è pari a zero, esso viene settato ad 1 e viene invocata un’altra funzione (ridenominata “fun2” nel grafo illustrato precedentemente). Tuttavia, si nota che, nella entry, la fun1 è invocata prima di chiamare il main: dunque, quando il main invocherà nuovamente fun1, il valore della variabile globale cui si è fatto riferimento in precedenza sarà di certo diverso da zero e, quindi, fun1 restituirà immediatamente il controllo al main senza fare nulla. È importante notare che, guardando i riferimenti alla variabile globale, si evince che essa viene acceduta sia in lettura che scrittura dalla sola funzione fun1, quindi il suo valore non può essere modificato in nessun’altra zona del codice.

Molto probabilmente, la chiamata a fun1 inserita nel main è frutto di una modifica del file eseguibile avvenuta successivamente alla generazione dell’eseguibile stesso e il suo scopo era semplicemente quello di ostacolare il processo di analisi e reversing del codice, cercando di depistare l’analista.

Inoltre, immediatamente dopo la chiamata alla fun1, nel main viene effettuato il controllo se argv[argc] è pari a NULL, che, se verificato, porta il main a terminare ritornando il valore 0. Questo controllo è sempre verificato, in quanto argv[argc] è sempre posto pari a NULL; dunque, di fatto, il programma quando viene eseguito non fa altro che terminare. Molto probabilmente questa operazione è una misura precauzionale legata esclusivamente allo svolgimento dell’homework, inserita per impedire l’analisi dinamica dell’eseguibile e l’utilizzo di debugger che avrebbero facilitato l’operazione di reversing.

Infine, si nota che i parametri passati alla funzione funCreateKey, che quindi rappresentano sostanzialmente la chiave del registro di sistema di cui si vogliono recuperare valori e sottochiavi, sono pari a HKEY\_LOCAL\_MACHINE e “SYSTEM\\ControlSet001\\Control” solo se argc è minore di 3. Quindi, passando in input al programma due parametri diversi, si può utilizzare il programma per ottenere dati riguardanti un’altra chiave.

# Conclusioni

In conclusione, il processo di reversing ha rivelato che lo scopo del programma è quello di recuperare informazioni contenute nel registro di sistema e riversarle su standard output. In particolare, di default vengono recuperati dati sui valori e sulle sottochiavi della chiave “HKEY\_LOCAL\_MACHINE\\SYSTEM\\ControlSet001\\Control”, la quale contiene i parametri di configurazione del sistema operativo l’ultima volta che ne è stato eseguito il boot. Tuttavia, passando al programma opportuni parametri, esso può essere usato per recuperare informazioni su altre chiavi del registro di sistema.

Le strutture dati utilizzate per il salvataggio delle informazioni recuperate su sottochiavi e valori sono delle single linked lists, costruite iterativamente con degli inserimenti (ed eventuali rimozioni in caso di errore) in testa alla lista e per le quali sono stati definiti dei tipi di dato strutturati che ne rappresentassero i nodi.

# Ricostruzione del programma in linguaggio C

Si riporta di seguito, a conclusione del processo di reversing del codice, una descrizione ad alto livello in linguaggio C del programma, ottenuta basandosi sull’analisi effettuata in precedenza e avvalendosi del supporto fornito dal decompilatore di *Ghidra*:

#include <stdio.h>

#include <windows.h>

#include <winreg.h>

**typedef** **struct** struct\_query\_info struct\_query\_info**;**

**typedef** **struct** struct\_subkey struct\_subkey**;**

**typedef** **struct** struct\_regValue struct\_regValue**;**

**struct** struct\_query\_info**{**

LPSTR className**;**

DWORD classNameLen**;**

DWORD numSubKeys**;**

DWORD maxSubKeyLen**;**

DWORD maxClassLen**;**

DWORD numValues**;**

DWORD maxValueNameLen**;**

DWORD maxValueLen**;**

DWORD securityDescriptor**;**

FILETIME lastWriteTime**;**

struct\_subkey **\***listOfSubKeys**;**

struct\_regValue **\***listOfValues**;**

**};**

**struct** struct\_subkey**{**

struct\_query\_info **\***queryInfo**;**

struct\_subkey **\***next**;**

LPSTR subKeyName**;**

DWORD nameLen**;**

**};**

**struct** struct\_regValue**{**

struct\_query\_info **\***queryInfo**;**

struct\_regValue **\***next**;**

CHAR valueName**[**16384**];**

DWORD valueNameLen**;**

DWORD type**;**

BYTE data**[**256**];**

DWORD dataLen**;**

**};**

DWORD global\_int**;**

**void** fun1**(){**

/\*

This function is also called by the entry function.

In particular, the entry function calls fun1() before calling the main():

this implies that, when the main calls fun1(), the value of

"global\_int" has already been set to 1.

SO THIS FUNCTION DOES ABSOLUTELY NOTHING WHEN IT'S CALLED BY MAIN!!!

Indeed, the variable "global\_int" is referenced only by fun1().

This means that global\_int can be modified only by fun1(), so, after the first call to fun1(), global\_int's value always differs from 0 !!!

\*/

**if(**global\_int **!=** 0**){**

**return;**

**}**

global\_int **=** 1**;**

// calls other stuff

//fun2();

**return;**

**}**

**void** funPrintResults**(**struct\_query\_info**\*** info**){**

printf**("Class: %s\n",**info**->**className**);**

printf**("Security descriptor: 0x%lx\n",**info**->**securityDescriptor**);**

printf**("Time: %08lx%08lx\n",(**info**->**lastWriteTime**).**dwLowDateTime**,** **(**info**->**lastWriteTime**).**dwHighDateTime**);**

// print subKeys names, if any

struct\_subkey**\*** currSubKey **=** info**->**listOfSubKeys**;**

**if(**currSubKey **!=** **NULL){**

puts**("SubKeys:");**

**do{**

printf**("\t%s\n",** currSubKey**->**subKeyName**);**

currSubKey **=** currSubKey**->**next**;**

**}** **while(**currSubKey **!=** **NULL);**

**}**

// print values, if any

struct\_regValue**\*** currValue **=** info**->**listOfValues**;**

**if(**currValue **!=** **NULL){**

puts**("Values:");**

**do{**

printf**("\t%s: [%lu] ",** currValue**->**valueName**,** currValue**->**type**);**

// print data, if any

**if(**currValue**->**dataLen **!=** 0**){**

**int** i**;**

**for(**i**=**0**;** i **<** currValue**->**dataLen**;** i**++){**

printf**(" %02x",** currValue**->**data**[**i**]);**

**}**

**}**

printf**(" (%s)\n",** currValue**->**data**);**

currValue **=** currValue**->**next**;**

**}** **while(**currValue **!=** **NULL);**

**}**

**return;**

**}**

struct\_query\_info**\*** funRetrieveInfo**(**HKEY param1**){**

struct\_subkey**\*** subkeys **=** **NULL;**

struct\_regValue**\*** values **=** **NULL;**

struct\_query\_info**\*** info **=** malloc**(sizeof(**struct\_query\_info**));** // 52 bytes

**if(**info **==** **NULL)** **goto** malloc\_failed**;**

info **->** className **=** malloc**(**260**);**

**if(**info **->** className **==** **NULL)** **goto** malloc\_failed**;**

**\*(**info **->** className**)** **=** **'\0';**

info**->**classNameLen **=** 260**;**

info**->**numSubKeys **=** 0**;**

// getting info

DWORD queryRes **=** RegQueryInfoKeyA**(**param1**,** info**->**className**,** **&**info**->**classNameLen**,** **NULL,** **&**info**->**numSubKeys**,** **&**info**->**maxSubKeyLen**,** **&**info**->**maxClassLen**,** **&**info**->**numValues**,** **&**info**->**maxValueNameLen**,** **&**info**->**maxValueLen**,** **&**info**->**securityDescriptor**,** **&**info**->**lastWriteTime**);**

**if(**queryRes **!=** ERROR\_SUCCESS**){**

puts **("RegQueryInfoKey failed: key not found");**

**return** **NULL;**

**}else{**

//retrieving subkeys and building the first linked list

**if(**info**->**numSubKeys **!=** 0**){**

struct\_subkey**\*** next **=** **NULL;** // last inserted subKey

**int** i**;**

**for(**i **=** 0**;** i **<** info **->** numSubKeys**;** i**++){**

subkeys **=** malloc**(sizeof(**struct\_subkey**));** // 16 bytes

**if(**subkeys **==** **NULL)** **goto** malloc\_failed**;**

subkeys **->** queryInfo **=** info**;**

subkeys **->** next **=** next**;**

subkeys **->** nameLen **=** info **->** maxSubKeyLen**;**

subkeys **->** subKeyName **=** malloc**(**info**->**maxSubKeyLen**);**

**if(**subkeys **->** subKeyName **==** **NULL)** **goto** malloc\_failed**;**

// getting subkey info

**if(**RegEnumKeyExA**(**param1**,** i**,** subkeys**->**subKeyName**,** **&**subkeys**->**nameLen**,**

**NULL,** **NULL,** **NULL,** **&**info**->**lastWriteTime**)** **!=** ERROR\_SUCCESS**){**

// removing the item from the list: something went wrong

next **=** subkeys**->**next**;** //head of the list remains the same

free**(**subkeys**);**

subkeys **=** next**;**

**}else{**

next **=** subkeys**;** // new head

**}**

**}**

**}**

info**->**listOfSubKeys **=** subkeys**;**

//retrieving values and building the second linked list

**if(**info**->**numValues **!=** 0**){**

struct\_regValue**\*** nextVal **=** **NULL;**

**int** j**;**

**for(**j**=**0**;** j **<** info**->**numValues**;** j**++){**

values **=** malloc**(sizeof(**struct\_regValue**));** //16660 bytes

**if(**values **==** **NULL)** **goto** malloc\_failed**;**

values**->**queryInfo **=** info**;**

values**->**next **=** nextVal**;**

values**->**valueNameLen **=** 16383**;** // MAX POSSIBLE

values**->**valueName**[**0**]** **=** **'\0';**

values**->**dataLen **=** 256**;**

**if(**RegEnumValueA**(**param1**,** j**,** values**->**valueName**,** **&**values**->**valueNameLen**,** **NULL,**

**&**values**->**type**,** **&**values**->**data**,** **&**values**->**dataLen**)** **!=** ERROR\_SUCCESS**){**

nextVal **=** values**->**next**;**

free**(**values**);**

**}**

nextVal **=** values**;**

**}**

**}**

info**->**listOfValues **=** values**;**

**return** info**;**

**}**

malloc\_failed**:**

puts**("Memory allocation error");**

exit**(**1**);**

**}**

struct\_query\_info**\*** funCreateKey**(**HKEY hKey**,** LPCSTR subKey**){**

struct\_query\_info**\*** ret **=** **NULL;**

HKEY newKey**;**

**if** **(**RegOpenKeyExA**(**hKey**,** subKey**,** 0**,** KEY\_ALL\_ACCESS**,** **&**newKey**)** **==** ERROR\_SUCCESS**){**

ret **=** funRetrieveInfo**(**newKey**);**

RegCloseKey**(**newKey**);** // close the handle to the previuosly opened key

**}**

**return** ret**;**

**}**

**int** main**(int** argc**,** **char** **\*\***argv**){**

// function that typically does nothing

fun1**();**

**if(**argv**[**argc**]** **==** **'\0'){**

/\*

always true; maybe to avoid leaking of useful informations

for the reversing process by executing the program

\*/

**return** 0**;**

**}**

**if(**argc **<** 3**){**

argv**[**1**]** **=** **'\0';**

argv**[**2**]** **=** **'\0';**

**}**

HKEY hKey **=** atoi**(**argv**[**1**]);**

**if(**hKey **==** 0**){**

hKey **=** HKEY\_LOCAL\_MACHINE**;**

**}**

**char\*** subKeyString **=** argv**[**2**];**

**if(**argv**[**2**]** **==** **'\0'){**

subKeyString **=** **"SYSTEM\\ControlSet001\\Control";**

**}**

struct\_query\_info**\*** info **=** funCreateKey**(**hKey**,** subKeyString**);**

**int** ret **=** 1**;**

**if(**info **!=** **NULL){**

funPrintResults**(**info**);**

ret **=** 0**;**

**}**

**return** ret**;**

**}**