Analisi del Malware – Relazione Homework 2

Studente: Andrea Pepe Matricola: 0315903

OBIETTIVO

Analizzare il programma eseguibile "hw2.exe", determinare il codice di sblocco che rende funzionale il programma e acquisire informazioni sul suo funzionamento e sulla sua struttura.

ANALISI PRELIMINARE

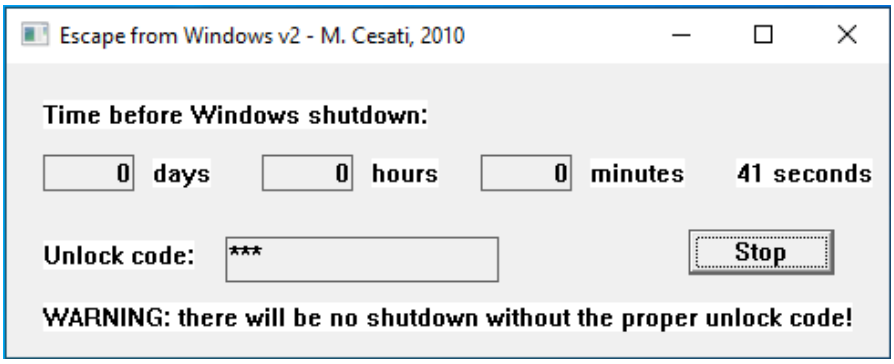
Il file eseguibile oggetto dell'analisi è in formato Portable Executable (PE), dunque un eseguibile per sistemi operativi Windows. Come prima operazione di analisi, è stata effettuata una ricerca delle stringhe utilizzate dal programma, in particolare usando il comando *strings* su sistema operativo Linux. Si riportano di seguito alcune delle stringhe più significative tra quelle ottenute:

- `Escape from Windows v2 - M. Cesati, 2010`
- `EDIT`
- `CreateWindow failed`
- `BUTTON`
- `Time before Windows shutdown:`
- `days`
- `hours`
- `minutes`
- `Unlock code:`
- `WARNING: there will be no shutdown without the proper unlock code!`
- `  0 seconds`
- `%2ld seconds`
- `Stop`
- `Sorry, try again!`
- `Shutdown time has come!`
- `However, the unlock code is wrong.`
- `If you want the full version of this wonderful tool,`
  `you can get the unlock code for just ten bucks!`
  `Ask to the nearest teacher around you!`
- `SeShutdownPrivilege`
- `AdjustTokenPrivileges`
- `LookupPrivilegeValueA`
- `GetCurrentProcess`
- `CreateWindowExA`
- `DefWindowProcA`
- `DispatchMessageA`
- `ExitWindowsEx`
- `GetMessageA`
- `KillTimer`
- `PostQuitMessage`
- `RegisterClassExA`

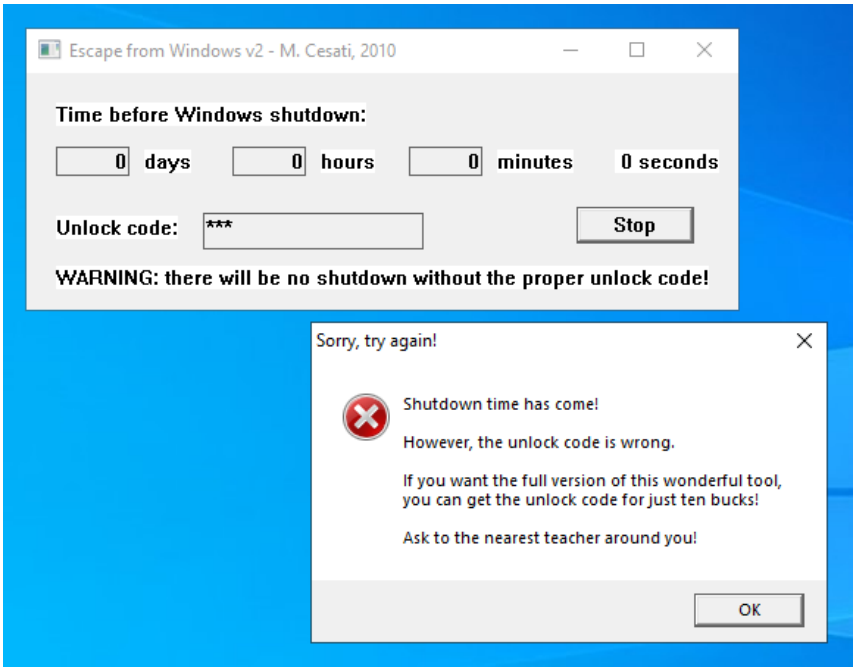
- `SetTimer`
- `TranslateMessage`
- `ADVAPI32.DLL`
- `GDI32.dll`
- `KERNEL32.dll`
- `msvcrt.dll`
- `USER32.dll`

Dalle stringhe si riescono a trarre alcune informazioni importanti: in primis, sia dalla stringa "GDI32.dll" che specifiche chiamate ad API come ad esempio "RegisterClassExA", "CreateWindowExA", "BUTTON" e "TranslateMessage" si desume che il programma abbia una GUI. Inoltre, ci sono diversi riferimenti allo shutdown di Windows e figurano le stringhe "SetTimer" e "KillTimer" che suggeriscono che l'applicazione abbia come obiettivo quello di realizzare lo spegnimento del sistema.

Per avere una conferma delle supposizioni fatte fino a questo punto, il programma è stato eseguito su una macchina virtuale con Sistema Operativo Windows 10 ed effettivamente, al lancio dell'eseguibile, appare una finestra per programmare lo shutdown del sistema operativo che richiede anche di inserire un codice di sblocco. Si è provato a far partire il countdown di 1 minuto con un codice di sblocco casuale ("abc") e il conto alla rovescia parte:



Tuttavia, allo scadere del timer, viene mostrato un messaggio di errore:



Ciò permette di desumere la seguente importante informazione: il controllo sulla correttezza del codice di sblocco viene effettuato una volta scaduto il timeout. Inoltre, premendo sul pulsante “Ok” presente nella finestra che riporta il messaggio d’errore, il programma viene terminato.

Queste informazioni suggeriscono di andare a cercare la TimerProc che viene passata alla funzione SetTimer, ovvero la procedura da chiamare allo scadere del timeout. Dunque, analizzando l’eseguibile con *Ghidra* e in particolare con il disassemblatore, si cercano i riferimenti alla API SetTimer ottenendo un unico riferimento. In effetti, esso è presente in una funzione che non fa altro che chiamare la set timer e assegnarne il valore di ritorno, quindi il timerID, ad una variabile globale. Tale funzione è stata denominata *fn_SetTimer*. L’ultimo paramentro passato alla SetTimer è appunto l’indirizzo di una funzione, ovvero la TimerProc. Dunque, tale funzione viene denominata *TimerProc*. La particolarità di tale funzione è la presenza di una istruzione CALL ad un indirizzo di memoria presente nel segmento dati dell’eseguibile, che *Ghidra* non riconosce come funzione né tantomeno ne riconosce il valore o il tipo di dato. Date le complicazioni riscontrate e la mancanza di ulteriori informazioni che potessero sostenere le deduzioni fatte fino a quel momento, si è deciso di acquisire maggiori dettagli sul funzionamento e sulle strutture dati del programma, andando ad individuare come prima cosa la WinMain.

INDIVIDUAZIONE DELLA WINMAIN

Per individuare la WinMain, sono stati cercati i riferimenti alla API DispatchMessageA, così da trovare il message loop. Anche in questo caso, figura un solo riferimento all’interno di una funzione che risulta così essere la funzione *WinMain*.

```
2  int WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nShowCmd)
3
4  {
5    ATOM AVar1;
6    program_data *lpParam;
7    HWND hWnd;
8    BOOL BVar2;
9    WNDCLASSEXA wndclassex_struct;
10   MSG local_24;
11
12   wndclassex_struct.hInstance = hInstance;
13   wndclassex_struct.lpszClassName = className;
14   wndclassex_struct.lpfnWndProc = WndProc;
15   wndclassex_struct.style = 8;
16   wndclassex_struct.cbSize = 48;
17   wndclassex_struct.hIcon = LoadIconA((HINSTANCE)0x0,(LPCSTR)0x7f00);
18   wndclassex_struct.hIconSm = LoadIconA((HINSTANCE)0x0,(LPCSTR)0x7f00);
19   wndclassex_struct.hCursor = LoadCursorA((HINSTANCE)0x0,(LPCSTR)0x7f00);
20   wndclassex_struct.lpszMenuName = (LPCSTR)0x0;
21   wndclassex_struct.cbClsExtra = 0;
22   wndclassex_struct.cbWndExtra = 0;
23   wndclassex_struct.hbrBackground = (HBRUSH)0x5;
24   AVar1 = RegisterClassExA(&wndclassex_struct);
25   if (AVar1 == 0) {
26     return 0;
27   }
28   lpParam = fn_init_data(struct_method);
29   hWnd = CreateWindowExA(0,className,s_Escape_from_Windows_v2_-_M._Cesa_00404020,WS_TILEDWINDOW,
30                          CW_USEDEFAULT,CW_USEDEFAULT,500,200,(HWND)0x0,(HMENU)0x0,hInstance,lpParam)
31   ;
32   ShowWindow(hWnd,nShowCmd);
33   while( true ) {
34     BVar2 = GetMessageA((LPMSG)&local_24,(HWND)0x0,0,0);
35     if (BVar2 == 0) break;
36     TranslateMessage(&local_24);
37     DispatchMessageA(&local_24);
38   }
39   return local_24.wParam;
40 }
```

In tale funzione, viene invocata la RegisterClassExA per registrare una classe di finestra: dunque, identificando la struttura di tipo WNDCLASSEX, il cui indirizzo viene passato come parametro alla API, è stato possibile individuare tra i campi, con l'ausilio del decompilatore, il class name della finestra, che corrisponde a "W04", e l'indirizzo della WindowProc incaricata della gestione degli eventi inviati alla finestra, ridenominata *WndProc*.

In seguito, viene invocata la CreateWindowExA per creare la finestra dell'applicazione ed ottenerne un handle. A tale API viene passato come ultimo parametro (lpParam) il valore di ritorno di una funzione chiamata appena prima e ridenominata *fn_init_data*, in quanto si presume che inizializzi una struttura contenente i dati dell'applicazione. Oltre a scrivere dei valori interi in una struct globale presente nella sezione dati, inserisce anche delle stringhe in alcuni buffer, usando una funzione che con buone probabilità corrisponde alla snprintf e che dunque viene ridenominata *fn_snprintf*. Si riporta di seguito il decompilato della funzione *fn_init_data*, ma in cui figurano già dei riferimenti più chiari alla struttura dati in questione che sono stati definiti in un secondo momento.

```
program_data * __cdecl fn_init_data(undefined *param_1)

{
  progData.current_time = 0;
  progData.clock_1000 = 1000;
  progData.target_time = 0x708;
  fn_snprintf(progData.warning_msg,0x80,
              "WARNING: there will be no shutdown without the proper unlock code!");
  fn_snprintf(progData.seconds_str,0x10," 0 seconds");
  progData.isRunning = 0;
  progData.checking_function = param_1;
  return &progData;
}
```

In effetti, il parametro lpParam della CreateWindowExA rappresenta un puntatore ad un valore passato alla finestra tramite il campo lpCreateParams di una struttura di tipo CREATESTRUCT e puntato dal parametro lParam del messaggio WM_CREATE. Tale parametro contiene per l'appunto dati addizionali da passare alla finestra al momento della sua creazione, con ogni probabilità raggruppati in una struct.

ANALISI DELLA WNDPROC

Quindi, si è passati ad analizzare la *WndProc* per desumerne informazioni utili a ricostruire la struttura dati in questione, da integrare con le informazioni già ottenute in precedenza dalla *fn_init_data*. Innanzitutto, è stato utilizzato il tool function graph per suddividere la funzione in blocchi logici e raggrupparli in base alla gestione del tipo di messaggio ricevuto dalla finestra; la *WndProc* gestisce in maniera non di default i messaggi:

- WM_SIZE;
- WM_PAINT;
- WM_COMMAND;
- WM_CREATE;
- WM_DESTROY.

Come prima operazione, prima di discernere il tipo di messaggio ricevuto dalla finestra, viene invocata l'API GetWindowLongA passandole come secondo parametro la macro GWL_USERDATA. In questo modo si ottiene un riferimento alla struttura dati passata alla finestra con la SetWindowLongA al momento della creazione della finestra e viene salvato in una variabile locale.

Analizzando la parte della funzione che gestisce il messaggio WM_CREATE, si identifica un array di 6 HANDLE inserito come campo della struttura dati, ridenominata *program_data*. Tale array, cui è stato dato il nome di *handles* è così composto:

- handles[0] → hWnd, handle alla finestra principale del programma;
- handles[1], handles[2], handles[3] → Sono gli handles alle edit boxes corrispondenti rispettivamente ai campi giorni, ore e minuti per l'input dell'utente;
- handles[4] → handle alla finestra di tipo bottone per far partire il countdown;
- handles[5] → handle alla edit box in cui scrivere il codice di sblocco.

```
if (uMsg == WM_CREATE) {
  hInstance = (HINSTANCE)GetWindowLongA(hWnd,GWL_HINSTANCE);
  SetWindowLongA(hWnd,GWL_USERDATA,*(LONG *)lParam);
  temp_5f4bbd694b = *(program_data **)lParam;
  temp_5f4bbd694b->handles[0] = hWnd;
  hMenu = (HMENU)0x1;
  do {
    pHVar3 = CreateWindowExA(0,"EDIT",(LPCSTR)0x0,0x50802002,0,0,0,0,hWnd,hMenu,hInstance,
                             (LPVOID)0x0);
    temp_5f4bbd694b->handles[(int)hMenu] = pHVar3;
    if (pHVar3 == (HWND)0x0) {
      FatalAppExitA(0,"CreateWindow failed");
    }
    hMenu = (HMENU)((int)&hMenu->unused + 1);
  } while (hMenu != (HMENU)0x4);
  pHVar3 = CreateWindowExA(0,"EDIT",(LPCSTR)0x0,0x50800020,0,0,0,0,hWnd,(HMENU)0x5,hInstance,
                           (LPVOID)0x0);
  temp_5f4bbd694b->handles[5] = pHVar3;
  pHVar3 = CreateWindowExA(0,"BUTTON","Go",0x50000001,0,0,0,0,hWnd,(HMENU)0x4,hInstance,
                           (LPVOID)0x0);
  temp_5f4bbd694b->handles[4] = pHVar3;
  if (pHVar3 == (HWND)0x0) {
    FatalAppExitA(0,"CreateWindow failed");
  }
  updateTime();
  fn_SetTimer(hWnd);
  return 0;
}
```

Alla fine della gestione del messaggio WM_CREATE, vengono invocate due funzioni: la prima, ridenominata *updateTime*, è utilizzata per cambiare i valori di giorni, ore, minuti e secondi rimanenti allo spegnimento. In questo contesto, essendo chiamata per la prima volta, setta tali valori a quelli impostati dalla *fn_init_data*, che corrispondono a 30 minuti (0x708 = 1800 secondi). Grazie all'analisi di questa funzione, si riesce a dare un significato ai valori della struct *program_data* assegnati dalla funzione di inizializzazione; in particolare si riconosce un valore che conta il tempo trascorso, uno che identifica il tempo target per lo spegnimento e uno che rappresenta la lunghezza del clock, pari a 1000 (millisecondi, ovvero un secondo). Inoltre, viene identificato un buffer di 16 bytes contenente la stringa "%2ld seconds" che viene aggiornata periodicamente al passare di ogni secondo.

```
void updateTime(void)

{
  HWND hDlg;
  uint uValue;
  UINT uValue_00;
  UINT uValue_01;

  hDlg = progData.handles[0];
  uValue = (uint)((progData.target_time - progData.current_time) * 1000) /
           (uint)(progData.clock_1000 * 0x3c);
  fn_snprintf(progData.seconds_str,0x10,"%2ld seconds",
              (progData.target_time - progData.current_time) + uValue * -0x3c);
  uValue_00 = 0;
  while (1439 < uValue) {
    uValue_00 = uValue_00 + 1;
    uValue = uValue - 1440;
  }
  uValue_01 = 0;
  while (59 < uValue) {
    uValue_01 = uValue_01 + 1;
    uValue = uValue - 60;
  }
  SetDlgItemInt(hDlg,1,uValue_00,0);
  SetDlgItemInt(hDlg,2,uValue_01,0);
  SetDlgItemInt(hDlg,3,uValue,0);
  return;
}
```

La seconda funzione è la *fn_SetTimer* incontrata all'inizio dell'analisi, che non fa altro che invocare l'API SetTimer per creare un timer associato alla finestra del programma, specificando come valore del clock quello inserito nella struct *program_data* e indicando la TIMERPROC da eseguire per processare messaggi di tipo WM_TIMER. Il valore di ritorno della SetTimer viene assegnato ad un campo della struct globale individuata in precedenza, dunque, tale campo sarà il *timerID*.

```
void __cdecl fn_SetTimer(HWND param_1)

{
  progData.timerID = SetTimer(param_1,0,progData.clock_1000,TimerProc);
  return;
}
```

STRUTTURA DATI E CODICE DI SBLOCCO

A questo punto dell'analisi, si è tornati sulla *TimerProc* e, con le informazioni ottenute sulla struttura dati usata dal programma, ci si rende conto che viene effettuata una call ad un campo della struct. Dunque, tale campo contiene l'indirizzo di una funzione, alla quale viene passato come parametro un riferimento alla struttura dati globale. Per indagare su quale funzione fosse, si ritorna indietro alla *WinMain*, che passa come parametro alla *fn_init_data* una variabile globale. Tale variabile viene attribuita al campo della struct di cui si effettua la call nella *TimerProc*.

A questo punto, la struttura dati *program_data* usata dall'applicazione è ricostruita. Con ogni probabilità, la funzione che viene invocata è quella che effettua il controllo sulla correttezza del codice di sblocco ed esegue lo shutdown di Windows.

Si riportano di seguito il decompilato della *TimerProc* e la struttura dati *program_data* così come definita in *Ghidra*, dopo averne denominato il campo della funzione come “checking_function”:

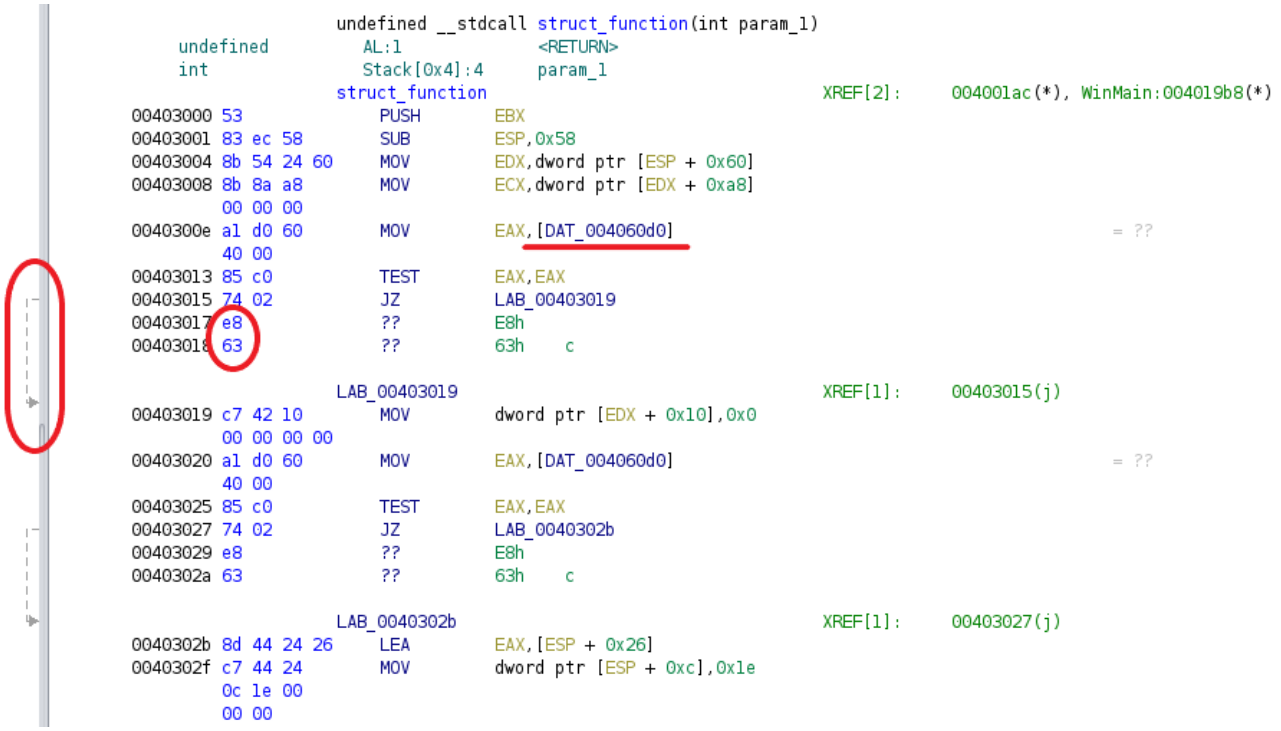
```
void TimerProc(HWND param_1)

{
  uint uVar1;

  uVar1 = progData.current_time + 1;
  progData.current_time = uVar1;
  if (progData.isRunning != 0) {
    updateTime();
  }
  RedrawWindow(param_1,(RECT *)0x0,(HRGN)0x0,5);
  if ((progData.isRunning != 0) && ((uint)progData.target_time <= uVar1)) {
    (*(code *)progData.checking_function)(&progData);
    return;
  }
  return;
}
```

Offset	Length	Mnemonic	DataType	Name	Comment
0	4	int	int	current_time	tempo trascorso
4	4	int	int	clock_1000	unità di clock
8	4	int	int	timerID	id del timer
12	4	int	int	target_time	tempo dopo il quale effettuare lo shutdown
16	4	int	int	isRunning	indica se il countdown è attivo
20	4	addr	pointer	checking_function	funzione che controlla il codice
24	128	char[128]	char[128]	warning_msg	messaggio di warning window principale
152	16	char[16]	char[16]	seconds_str	stringa indicante i secondi del countdown
168	24	HWND[6]	HWND[6]	handles	handles alla finestra e sottofinestre

Ora, non resta altro che indagare sulla funzione che viene invocata e di cui la struct conserva l’indirizzo. Si nota che tale funzione è presente nel segmento dati ed è la stessa trovata all’inizio dell’analisi. *Ghidra* non è riuscito a riconoscerla come funzione e quindi i bytes delle istruzioni che la compongono non sono stati disassemblati. Si forza *Ghidra* a disassemblare e a riconoscere la funzione; il risultato ottenuto è il seguente:

```
                        undefined __stdcall struct_function(int param_1)
     undefined            AL:1            <RETURN>
     int                  Stack[0x4]:4    param_1                                  XREF[1]:     00403004(R)
                        struct_function                                  XREF[2]:     004001ac(*), WinMain:004019b8(*)
00403000 53             PUSH    EBX
00403001 83 ec 58       SUB     ESP,0x58
00403004 8b 54 24 60    MOV     EDX,dword ptr [ESP + param_1]
00403008 8b 8a a8       MOV     ECX,dword ptr [EDX + 0xa8]
         00 00 00
0040300e a1 d0 60       MOV     EAX,[DAT_004060d0]                                                = ??
         40 00
00403013 85 c0          TEST    EAX,EAX
00403015 74 02          JZ      LAB_00403017+2
                        LAB_00403017+2                                   XREF[0,1]:   00403015(j)
00403017 e8 63 c7       CALL    SUB_1082f77f
         42 10
0040301c 00 00          ADD     byte ptr [EAX],AL
0040301e 00 00          ADD     byte ptr [EAX],AL
00403020 a1 d0 60       MOV     EAX,[DAT_004060d0]                                                = ??
         40 00
00403025 85 c0          TEST    EAX,EAX
00403027 74 02          JZ      LAB_00403029+2
                        LAB_00403029+2                                   XREF[0,1]:   00403027(j)
00403029 e8 63 8d       CALL    SUB_2484bd91
         44 24
0040302e 26 c7 44       MOV     dword ptr ES:[ESP + 0xc],0x1e
         24 0c 1e
         00 00 00
```

Come si può notare, vengono segnalate delle istruzioni CALL in rosso, ad indicare che gli indirizzi specificati non sono presenti all'interno del file eseguibile. Analizzando le informazioni fornite dal disassembler di Ghidra, si nota che prima di ognuna di queste CALL, c'è un'istruzione JZ ad una label al terzo byte dei 5 che compongono una istruzione di CALL (1 byte per l'opcode + 4 bytes per l'indirizzo a 32 bit a cui saltare). Si desume che, molto probabilmente, il disassemblatore vada in confusione e quindi si decide di disassemblare il codice poco alla volta, per blocchi, seguendo le indicazioni fornite dalle labels. Il risultato ottenuto è il seguente:



Nell'immagine, sono evidenziate delle coppie di bytes [E8, 63] che non rappresentano nessuna istruzione, ma che confondevano il disassemblatore di *Ghidra*, inducendolo ad interpretarli come facenti parte di una istruzione di CALL, in quanto l'opcode di tale istruzione è appunto il byte E8. Si nota inoltre che il flusso d'esecuzione giunge a tali bytes solo se il valore della variabile globale DAT_004060d0 è diverso da zero. Tuttavia, tale variabile è situata nel segmento BSS, dunque è inizializzata a 0; inoltre, non ci sono riferimenti in scrittura a tale variabile e il suo valore sarà quindi sempre zero. In conclusione, il salto condizionale corrispondente all'istruzione JZ prima dei due bytes in questione verrà sempre preso e il flusso di esecuzione non si ritroverà mai ad eseguire tali bytes.

Si suppone che tale struttura sia stata ottenuta utilizzando un offuscatore di codice per ostacolare il reversing del programma.

Si decide di effettuare per ogni occorrenza della coppia di bytes [E8, 63] una Patch Instruction all'interno di *Ghidra*, inserendo una coppia di NOP [90, 90]. A questo punto, la funzione, ridenominata *struct_method*, diventa molto più leggibile sfruttando il decompilatore.

Nell'immagine che si riporta di seguito, figura già una chiamata alla API AdjustTokenPrivileges. In realtà, essa non era stata riconosciuta da *Ghidra* ed è stato necessario effettuare una ulteriore forzatura a disassemblare dopo averne pulito il codice erroneamente disassemblato in precedenza. A questo punto, ne è venuta fuori una semplice istruzione di jump alla API della DLL ed è quindi stata indicata come una thunk function. Allo stesso modo, è stato necessario disassemblare la funzione indicata come *fn_wrong_code_inserted*, che corrisponde alla funzione che fa apparire la finestra contenente il messaggio d'errore quando il codice di sblocco inserito dall'utente risulta essere sbagliato.

```
void struct_method(program_data *program_data)

{
  UINT UVar1;
  HANDLE ProcessHandle;
  BOOL BVar2;
  DWORD DVar3;
  CHAR user_input;
  char cStack53;
  char cStack52;
  char cStack51;
  char cStack50;
  char cStack49;
  char cStack48;
  char cStack47;
  char cStack46;
  _TOKEN_PRIVILEGES tk_privileges;
  HANDLE tokenHandle;

  program_data->isRunning = 0;
  UVar1 = GetDlgItemTextA(program_data->handles[0],5,&user_input,30);
  if ((((((UVar1 == 9) && (user_input == '3')) && (cStack53 == 'R')) &&
       ((cStack52 == 'n' && (cStack51 == 'E')))) &&
      ((cStack50 == 'S' && ((cStack49 == 't' && (cStack48 == '0')))))) &&
     ((cStack47 == '!' && (cStack46 == '?')))) {
    ProcessHandle = GetCurrentProcess();
    BVar2 = OpenProcessToken(ProcessHandle,0x28,&tokenHandle);
    if (BVar2 == 0) {
      PostQuitMessage(0);
    }
    LookupPrivilegeValueA((LPCSTR)0x0,"SeShutdownPrivilege",(PLUID)tk_privileges.Privileges);
    tk_privileges.PrivilegeCount = 1;
    tk_privileges.Privileges[0].Attributes = 2;
    AdjustTokenPrivileges
              (tokenHandle,0,(PTOKEN_PRIVILEGES)&tk_privileges,0,(PTOKEN_PRIVILEGES)0x0,(PDWORD)0x0)
    ;
    DVar3 = GetLastError();
    if (DVar3 == 0) {
      ExitWindowsEx(5,0);
    }
    PostQuitMessage(0);
  }
  else {
    fn_wrong_code_inserted();
  }
  return;
}
```

Dunque, la *struct_method* chiama la GetDlgItemTextA per ottenere l'input inserito dall'utente nella edit box del codice di sblocco. Dopodiché, controlla che la lunghezza di tale input sia pari a 9 e, char per char, effettua il confronto con il codice di sblocco corretto, il quale risulta essere la stringa "3RnESt0!?". Se il controllo non va a buon fine, viene invocata la *fn_wrong_code_inserted* per mostrare il messaggio d'errore:

```
void fn_wrong_code_inserted(void)

{
  MessageBoxA((HWND)0x0,

              "Shutdown time has come!\n\nHowever, the unlock code is wrong.\n\nIf you want the
              full version of this wonderful tool,\nyou can get the unlock code for just ten
              bucks!\n\nAsk to the nearest teacher around you!"
              ,"Sorry, try again!",MB_ICONERROR);
  PostQuitMessage(0);
  return;
}
```

Invece, qualora il codice di sblocco inserito fosse corretto, la funzione si prepara ad effettuare lo shutdown del sistema nel seguente modo:

1. Chiama l'API GetCurrentProcess per ottenere uno pseudo handle al processo corrente;
2. Invoca la OpenProcessToken per aprire un access token associato con il processo corrente, specificando una access mask pari a 0x28, ovvero l'or delle macro TOKEN_ADJUST_PRIVILEGES (0x20) e TOKEN_QUERY (0x08). Non essendo presenti sulla documentazione Microsoft, tali valori sono stati reperiti al seguente link: https://www.pinvoke.net/default.aspx/advapi32.openprocesstoken. Tali macro, indicano che è possibile effettuare delle query sull'access token e che si possono abilitarne o disabilitarne privilegi;
3. Effettua una chiamata all'API LookupPrivilegeValueA con la quale recupera il Locally Unique Identifier (LUID) del privilegio SE_SHUTDOWN_NAME, identificato dalla stringa "SeShutdownPrivilege", e lo salva nel campo Privileges di una struct locale di tipo TOKEN_PRIVILEGES. In questo modo, ha ottenuto l'identificatore sulla macchina locale del privilegio per effettuare lo shutdown del sistema;
4. Il campo Privileges della struct TOKEN_PRIVILEGES è a sua volta una struct, di tipo LUID_AND_ATTRIBUTES, che ha come primo campo il LUID del privilegio e come secondo campo un attributo, che viene settato a 2, ovvero a SE_PRIVILEGE_ENABLED.
5. L'indirizzo della struct TOKEN_PRIVILEGES viene passato, insieme all'handle al token ottenuto con la OpenProcessToken, alla API AdjustTokenPrivileges per abilitare il privilegio necessario ad effettuare lo shutdown per l'access token appena aperto.
6. Se tutto è andato a buon fine, viene effettuato lo spegnimento del sistema invocando l'API ExitWindowsEx passando come primo parametro 5 e come secondo parametro 0. Il primo parametro corrisponde all'or dei due flag EWX_SHUTDOWN (0x01) ed EWX_FORCE (0x04): ciò vuol dire che il sistema viene arrestato forzando tutte le applicazioni a terminare, comportando una potenziale perdita di dati da parte delle applicazioni. Il secondo parametro, pari a 0, indica che non viene indicato alcun motivo per l'inizio dello shutdown.

CONTROMISURE ANTI-DEBUGGING

Come ultima operazione nel processo di analisi e reversing dell'eseguibile, si ritorna su una funzione chiamata all'interno della *WinProc* inizialmente ignorata:

```
l_programData = (program_data *)GetWindowLongA(hWnd,GWL_USERDATA);
FUN_00401a90((int *)&uMsg);
```

A questa funzione FUN_00401a90, viene passato come parametro di ingresso l'indirizzo di uMsg, ovvero l'argomento della funzione *WinProc* che rappresenta il codice identificativo del messaggio ricevuto dalla finestra. Di seguito viene riportato il disassemblato della funzione:

```
                    undefined __cdecl FUN_00401a90(int * param_1)
     undefined          AL:1           <RETURN>
     int *              Stack[0x4]:4   param_1                          XREF[1]:     00401a93(R)
                    FUN_00401a90                                    XREF[1]:     WndProc:00401314(c)
00401a90 55             PUSH        EBP
00401a91 89 e5          MOV         EBP,ESP
00401a93 8b 55 08       MOV         EDX,dword ptr [EBP + param_1]
00401a96 64 a1 30       MOV         EAX,FS:[0x30]
         00 00 00
00401a9c 8b 40 02       MOV         EAX,dword ptr [EAX + 0x2]
00401a9f a8 07          TEST        AL,0x7
00401aa1 0f 95 c0       SETNZ       AL
00401aa4 0f b6 c0       MOVZX       EAX,AL
00401aa7 01 02          ADD         dword ptr [EDX],EAX
00401aa9 5d             POP         EBP
00401aaa c3             RET
```

In sistemi con architettura x86, il registro FS è usato per accedere tramite offset ad una struttura dati detta TIB (Thread Information Block), o anche TEB (Thread Envirnment Block), contenente informazioni sul thread attualmente in esecuzione. In particolare, con FS:[0x30], ovvero con offset 0x30 a partire da FS, si ottiene l'indirizzo di una ulteriore struttura dati detta PEB (Process Environment Block), come suggerito dalla pagina Wikipedia: https://en.wikipedia.org/wiki/Win32_Thread_Information_Block#Accessing_the_TIB.

La struttura dati PEB contiene informazioni sul processo corrente, ma non è completamente documentata da Windows. Tuttavia, la funzione tenta di recuperare il valore ad offset 2 di tale struttura, che, come desunto dalle risorse web https://www.aldeid.com/wiki/PEB-Process-Environment-Block/BeingDebugged e https://docs.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb, corrisponde al campo BeingDebugged. Tale campo è un byte che è pari a 1 nel caso in cui il processo corrente è eseguito in modalità di debbugging, 0 altrimenti.

Dunque, ciò che la funzione chiamata dalla *WndProc* fa, è di consultare tale valore e, nel caso in cui l'applicazione è stata lanciata con un debugger, modifica il valore di uMsg aumentandolo di 1 prima che ne venga fatto il confronto con le varie macro corrispondenti ai tipi di messaggi che la finestra deve gestire. Altrimenti, il valore di uMsg resta inalterato. In questo modo, si ostacola l'analisi dinamica avanzata del programma, facendogli assumere un comportamento completamente diverso quando eseguito tramite un debugger.

CONCLUSIONE

In conclusione, il codice di sblocco del programma è "3RnESt0!?". Esso è stato anche testato su macchina virtuale ed ha riportato il risultato atteso, ovvero lo spegnimento del sistema.

L'applicazione usa una struttura dati un po' "anomala", in quanto è una struct che contiene un indirizzo di una funzione, facendo pensare ad una classe in un contesto Object Oriented. Inoltre, tale funzione è contenuta nella sezione dati dell'eseguibile e probabilmente è stata modificata dall'utilizzo di un offuscatore per complicarne il reversing.

Infine, si riporta che l'applicazione tenta di proteggersi da un'analisi effettuata con un debugger, rilevando tale situazione e mutando il suo comportamento a runtime, così da ingannare l'analista.