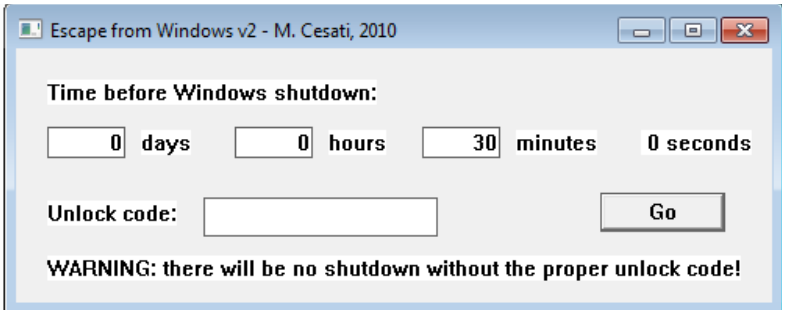
Homework 2
Malware Analysis

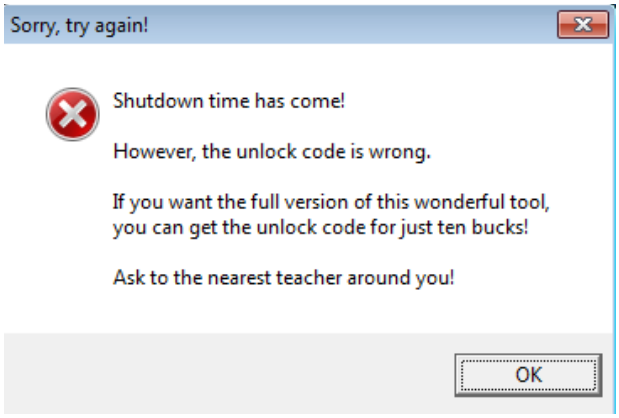
Danilo Dell'Orco
0300229
Novembre 14, 2021

1 Introduzione

Per analizzare il comportamento del programma, lanciamo l'eseguibile su una macchina virtuale Windows 7. Vediamo che stiamo trattando un'applicazione grafica, che permette di schedulare lo shutdown della macchina.



Ci viene indicato tramite un messaggio di warning che il programma per funzionare correttamente necessita di un certo *Unlock Code*. Non siamo in possesso di tale codice, per cui proviamo innanzitutto ad avviare il timer inserendo un codice qualsiasi. Il countdown viene eseguito correttamente, ma allo scadere viene mostrato il seguente messaggio di errore:



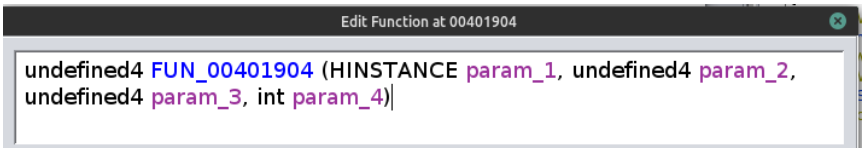
Il nostro obiettivo è quindi quello di analizzare l'eseguibile, cercando in particolare il *codice di sblocco* che rende funzionale il programma.

2 Ricerca del WinMain

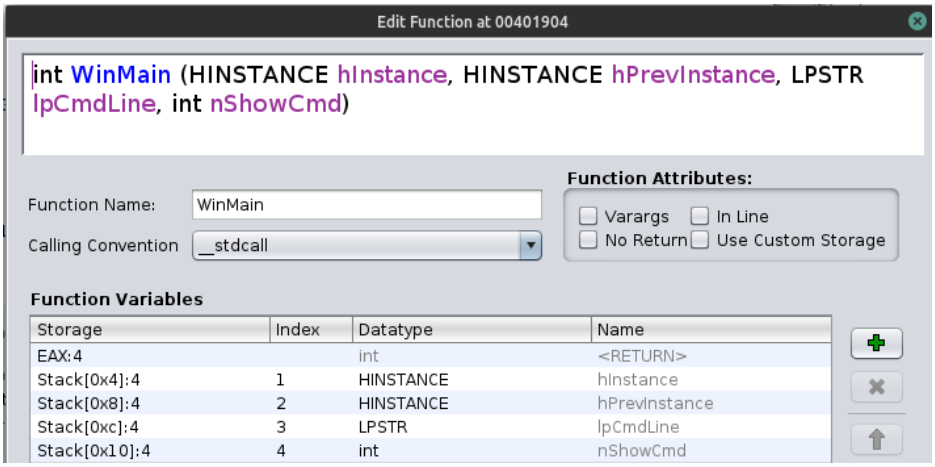
Analizzando il codice possiamo trovare molte funzioni non riconosciute da Ghidra. Come abbiamo visto eseguendo il programma, questo presenta una GUI, per cui ci aspettiamo la presenza di un **WinMain**.

```
int __clrcall WinMain(
[in] HINSTANCE hInstance,
[in] HINSTANCE hPrevInstance,
[in] LPSTR lpCmdLine,
[in] int nShowCmd
);
```

Dobbiamo quindi cercare una funzione che presenti i parametri standard visti dalla documentazione. La candidata più probabile è **FUN_00401904.**



I parametri di questa funzione non vengono correttamente rilevati da Ghidra, che li indica come *undefined*. Tramite lo strumento **Edit Function**, dobbiamo quindi effettuare il *retype* dei parametri di input e ritorno, così da allinearli a quelli standard di *WinMain*.



3 Analisi del WinMain

La gestione dell'applicazioni grafiche in Windows è basata su *eventi* o *messaggi*. Ogni finestra ha una funzione, chiamata **window procedure**, che il sistema chiama ogni volta che ha un *messaggio* per la finestra. Questa procedura elabora il messaggio ricevuto, esegue le azioni richieste, e restituisce il controllo al sistema.

All'interno del *WinMain* la prima funzione invocata è *RegisterClassExA*. Questa funzione permette di registrare una *classe*, che verrà successivamente utilizzata per istanziare una finestra.

Dalla documentazione vediamo che *RegisterClassExA* accetta come input un solo parametro, che è un puntatore ad una struttura `WNDCLASSEXA`

```
ATOM RegisterClassExA(
  [in] const WNDCLASSEXA *unnamedParam1
);
```

Questa, mantiene tutti gli attributi della *classe di finestra* che si vuole registrare, e presenta i seguenti campi:

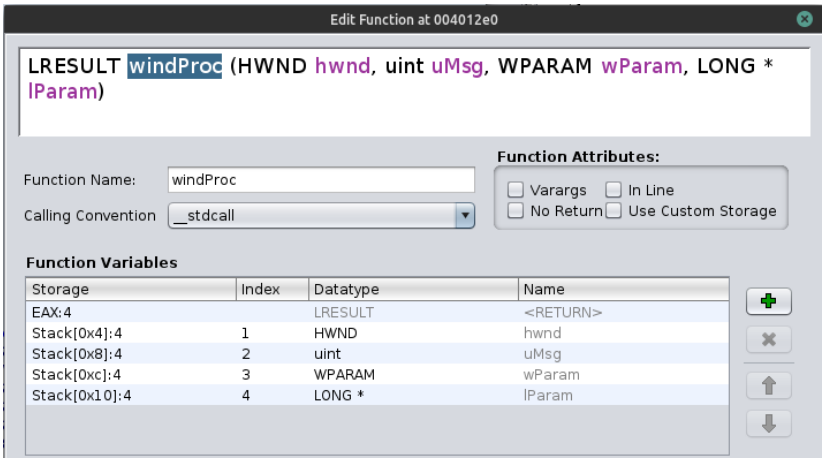
```
typedef struct tagWNDCLASSEXA {
  UINT      cbSize;
  UINT      style;
  WNDPROC   lpfnWndProc;
  int       cbClsExtra;
  int       cbWndExtra;
  HINSTANCE hInstance;
  HICON     hIcon;
  HCURSOR   hCursor;
  HBRUSH    hbrBackground;
  LPCSTR    lpszMenuName;
  LPCSTR    lpszClassName;
  HICON     hIconSm;
} WNDCLASSEXA, *PWNDCLASSEXA, *NPWNDCLASSEXA, *LPWNDCLASSEXA;
```

Tale struttura viene riconosciuta automaticamente da Ghidra, in quanto è definita all'interno della libreria *winuser.h*. Andiamo quindi a rinominarla in *winClass*, usando la funzione rename variable.

Dal codice possiamo osservare come vengano assegnati tutti i parametri di questa strutura.

```
9   WNDCLASSEXA winClass;
10  MSG local_24;
11
12  winClass.hInstance = hInstance;
13  winClass.lpszClassName = &DAT_00404000;
14  winClass.lpfnWndProc = (WNDPROC)&LAB_004012e0;
15  winClass.style = 8;
16  winClass.cbSize = 0x30;
17  winClass.hIcon = LoadIconA((HINSTANCE)0x0,(LPCSTR)IDI_APPLICATION);
18  winClass.hIconSm = LoadIconA((HINSTANCE)0x0,(LPCSTR)IDI_APPLICATION);
19  winClass.hCursor = LoadCursorA((HINSTANCE)0x0,(LPCSTR)IDC_ARROW);
20  winClass.lpszMenuName = (LPCSTR)0x0;
21  winClass.cbClsExtra = 0;
22  winClass.cbWndExtra = 0;
23  winClass.hbrBackground = (HBRUSH)0x5;
24  AVar1 = RegisterClassExA(&winClass);
```

Il campo *lpfnWndProc* è di tipo *WINDPROC*, e rappresenta un puntatore alla window procedure. Essendo questa una callback, Ghidra non ha trovato nessuna istruzione esplicita di CALL verso la funzione, ed ha interpretato questa come una label. Andiamo quindi a definire una nuova funzione a partire da tale label, denominandola **WindProc**




```
13  winClass.lpszClassName = &DAT_00404000;
14  winClass.lpfnWndProc = windProc;
15  winClass.style = 8;
```

I campi *hIcon* e *hIconSm* specificano un handle ad una classe icona. Questi vengono impostati utilizzando la funzione *LoadIconA*.

```
HICON LoadIconA(
  [in, optional] HINSTANCE hInstance,
  [in]           LPCSTR    lpIconName
);
```

In particolare come parametro lpIconName viene passato il codice *0x7f00* (=32512), che corrisponde all'icona di default *IDI_APPLICATION*. Utilizziamo quindi *set equate* per settare tale valore

Value	Meaning
IDI_APPLICATION MAKEINTRESOURCE(32512)	Default application icon.

Analogamente, viene impostato anche il campo *hCursor*, che specifica la classe di cursore per questa specifica finestra. In questo caso si utilizza la funzione *LoadCursorA*.

```
HCURSOR LoadCursorA(
  [in, optional] HINSTANCE hInstance,
  [in]           LPCSTR    lpCursorName
);
```

Anche in questo caso viene impostata un cursore di default tramite il codice *0x7f00*, che corrisponde a *IDC_ARROW*. Specifichiamo tale valore utilizzando *Set Equate*

IDC_ARROW MAKEINTRESOURCE(32512)	Standard arrow

Successivamente viene invocata la funzione *RegisterClassExA*, passando come parametro il puntatore alla struttura appena definita.

```
0040199a 8d 45 b0        LEA       EAX=>winClass,[EBP + -0x50]
0040199d 89 04 24        MOV       dword ptr [ESP]=>fn_arg1,EAX
004019a0 e8 3b 0b        CALL      USER32.DLL::RegisterClassExA
         00 00
```

Questa funzione ritorna un *ATOM* che identifica la classe registrata in AX, quindi nei 16 bit meno significativi del registro EAX.

```
                **************************************************************************
                *                          POINTER to EXTERNAL FUNCTION                  *
                **************************************************************************
                ATOM __stdcall RegisterClassExA(WNDCLASSEXA * param_1)
ATOM              AX:2           <RETURN>
WNDCLASSEXA *     Stack[0x4]:4   param_1
```

Si effettua un controllo sulla classe registrata, eseguendo la procedura di ritorno in caso di errore.

```
004019a5 83 ec 04       SUB         ESP,0x4
004019a8 31 d2          XOR         EDX,EDX
004019aa 66 85 c0       TEST        AX,AX
004019ad 75 09          JNZ         no_return_label
004019af 89 d0          MOV         EAX,EDX
004019b1 8b 5d fc       MOV         EBX,dword ptr [EBP + local_8]
004019b4 c9             LEAVE
004019b5 c2 10 00       RET         0x10
```

Successivamente, viene chiamata la funzione *FUN_00401aab*, passando in input un certo dato *DAT_0040300*.

```
                        class_not_zero                                  XREF[1]:
004019b8 c7 04 24       MOV         dword ptr [ESP]=>fn_arg1,DAT_00403000
         00 30 40 00
004019bf e8 e7 00       CALL        FUN_00401aab
         00 00
```

Attualmente non abbiamo nessuna informazione su questo parametro, e anche andando ad esaminare il codice decompilato non riusciamo a ricavare nessuna informazione, in quanto vengono utilizzati altri dati non ancora definiti.

```
 4 undefined4 * __cdecl FUN_00401aab(undefined4 param_1)
 5
 6 {
 7   DAT_00406010 = 0;
 8   DAT_00406014 = 1000;
 9   DAT_0040601c = 0x708;
10   FUN_004023c0(&DAT_00406028,0x80,
11                "WARNING: there will be no shutdown without the proper unlock code!");
12   FUN_004023c0(&DAT_004060a8,0x10," 0 seconds");
13   DAT_00406020 = 0;
14   _DAT_00406024 = param_1;
15   return &DAT_00406010;
16 }
```

Possiamo tuttavia osservare che questa funzione ritorna in *EAX*, ed il valore di ritorno viene successivamente copiato in *fn_arg12*, ovvero l'ultimo parametro passato alla funzione *CreateWindowExA*.

```
                    class_not_zero                                  XREF[1]:
004019b8 c7 04 24       MOV       dword ptr [ESP]=>fn_arg1,DAT_00403000
         00 30 40 00
004019bf e8 e7 00       CALL      FUN_00401aab
         00 00
004019c4 89 44 24 2c    MOV       dword ptr [ESP + fn_arg12],EAX
         40 00
                                  . . .
00401a1b e8 a8 0a       CALL      USER32.DLL::CreateWindowExA
         00 00
```

CreateWindowExA crea una istanza della classe di finestra specificata.

```
HWND CreateWindowExA(
  [in]            DWORD     dwExStyle,
  [in, optional] LPCSTR     lpClassName,
  [in, optional] LPCSTR     lpWindowName,
  [in]            DWORD     dwStyle,
  [in]            int       X,
  [in]            int       Y,
  [in]            int       nWidth,
  [in]            int       nHeight,
  [in, optional] HWND       hWndParent,
  [in, optional] HMENU      hMenu,
  [in, optional] HINSTANCE hInstance,
  [in, optional] LPVOID     lpParam
);
```

Il valore ritornato da *FUN_00401aab* corrisponde quindi al parametro *lpParam* della funzione *CreateWindowExA*. Dalla documentazione vediamo che il valore puntato da questo parametro dipende dal parametro *lParam* del messaggio *WM_CREATE*.

Riguardo *FUN_00401aab* non abbiamo quindi informazioni precise sul suo comportamento, per cui ne rimandiamo l'analisi ad un secondo momento. Possiamo proseguire andando ad esaminare la *Window Procedure*.

4 Window Procedure

4.1 Descrizione

All'interno della *Window Procedure* sono implementati tutti i gestori dei messaggi per cui non si vuole un comportamento di default. Analizzando il codice vediamo che vengono gestiti i seguenti messaggi: *0x5*, *0x1*, *0x2*, *0xf*, *0x111*.

Sfruttando la documentazione, sostituiamo questi codici numerici con il nome del messaggio, tramite la funzione *set equate* di Ghidra.

```
0x5       →     WM_SIZE
0x1       →     WM_CREATE
0x2       →     WM_DESTROY
0xf       →     WM_PAINT
0x111     →     WM_COMMAND
```

`WM_SIZE` viene inviato al ridimensionamento della finestra. `VM_DESTROY` viene inviato quando una finestra viene distrutta. `WM _ PAINT` viene inviato quando il sistema o un'altra applicazione effettua una richiesta per disegnare una parte della finestra di un'applicazione.

Questi messaggi non risultano di particolare interesse nel nostro studio, in quanto sicuramente non vanno a definire il comportamento per il controllo sul codice di sblocco.

`WM_CREATE` viene inviato quando un'applicazione richiede la creazione di una finestra chiamando la funzione *CreateWindowExA*. Nella gestione di questo messaggio viene creata l'interfaccia effettiva dell'applicazione e viene definito il comportamento dell'applicazione all'avvio.

`WM_COMMAND` viene inviato quando l'utente seleziona una voce di comando da un menu, quando un controllo invia un messaggio di notifica alla finestra padre, o quando viene convertita una shortcut da tastiera. Gestendo questo messaggio viene quindi definito il comportamento dell'applicazione in risposta agli input dell'utente.

4.2 WindProc

All'inizio della *WindProc*, vengono salvati i parametri di input *wParam* ed *lParam* rispettivamente nei registri EBX ed EDI.

```
004012ec 8b 5d 10        MOV        EBX,dword ptr [EBP + wParam]
004012ef 8b 7d 14        MOV        EDI,dword ptr [EBP + lParam]
```

Successivamente viene chiamata la funzione *GetWindowLongA*.

```
LONG GetWindowLongA(
  [in] HWND hWnd,
  [in] int  nIndex
);
```

Il primo parametro è un handle alla finestra, mentre il secondo parametro è un offset al valore che si vuole ottenere.

Come *nIndex* viene passato `-21`, che corrisponde a `GWL_USERDATA`. Ciò significa che la finestra consulta un valore definito dal programmatore per decidere cosa fare. Il valore di ritorno viene scritto in *EAX*, che viene poi copiato nella variabile *local_b0*

```
004012f2 c7 44 24        MOV        dword ptr [ESP + local_e8],GWL_USERDATA
         04 eb ff
         ff ff
004012fa 8b 45 08        MOV        EAX,dword ptr [EBP + hwnd]
004012fd 89 04 24        MOV        dword ptr [ESP]=>local_ec,EAX
00401300 e8 73 11        CALL       USER32.DLL::GetWindowLongA
         00 00
00401305 83 ec 08        SUB        ESP,0x8
00401308 89 85 54        MOV        dword ptr [EBP + local_b0],EAX
         ff ff ff
```

Questo valore definito dal programmatore, viene settato tramite l'API *SetWindowLongA*, utilizzata nella gestione del messaggio *WM_CREATE*.

```
LONG SetWindowLongA(
  [in] HWND hWnd,
  [in] int  nIndex,
  [in] LONG dwNewLong
);
```

In input a questa funzione viene passato come primo parametro l'handler alla finestra; come secondo parametro parametro il codice *-21*, e quindi `GWL_USERDATA`; come terzo parametro il valore puntato dal contenuto del registro *EDI*.

```
00401732 8b 07           MOV        EAX,dword ptr [EDI]
00401734 89 44 24 08     MOV        dword ptr [ESP + local_e4],EAX
00401738 c7 44 24        MOV        dword ptr [ESP + local_e8],GWL_USERDATA
         04 eb ff
         ff ff
00401740 8b 75 08        MOV        ESI,dword ptr [EBP + hwnd]
00401743 89 34 24        MOV        dword ptr [ESP]=>local_ec,ESI
00401746 e8 75 0d        CALL       USER32.DLL::SetWindowLongA
         00 00
```

Come visto in precedenza, *EDI* contiene il puntatore a *lParam*. Il valore di questo parametro dipende dal messaggio specifico che *WindProc* sta gestendo, in questo caso *WM_CREATE*.

In tale scenario *lParam* contiene un puntatore alla struttura *CREATESTRUCT*, per cui stiamo copiando in *EAX* il primo parametro di *CREATESTRUCT*. Dalla documentazione vediamo che il primo parametro di questa struttura è *lpCreateParam* che contiene il valore di *lpParam* passato in input alla funzione *CreateWindowExA*.

- **lpCreateParams**

  Long pointer to additional data that can be used to create the window. If the window is being created as a result of a call to the CreateWindow or CreateWindowEx function, this member contains the value of the *lpParam* parameter specified in the function call.

Quindi in definitiva, il valore associato a *GWL_USERDATA* corrisponde a *lpParam* che è stato passato come ultimo parametro nella *CreateWindowExA* invocata nel *WinMain*.

Proseguendo con l'analisi del codice vediamo che il valore dell'indirizzo puntato da *EDI* viene copiato tramite *MOV* nel registro *ESI*. Quindi *ESI* contiene *lpParam*.

Questo registro viene utilizzato per un accesso del tipo *base+offset*. Ciò ci suggerisce come *lpParam* sia un puntatore ad una struttura dati. Definiamo quindi una nuova struttura **ProgStruct** ed associamo al *byte 168* un campo di tipo *HWND*, che rappresenta l'handler della finestra.

```
0040174e 8b 37           MOV        ESI,dword ptr [EDI]
00401750 8b 7d 08        MOV        EDI,dword ptr [EBP + hwnd]
00401753 89 be a8        MOV        dword ptr [ESI + 168],EDI
         00 00 00
00401759 bb 01 00        MOV        EBX,1
         00 00
```

In definitva, *GetWindowLongA* ritorna un puntatore alla struttura *ProgStruct*. Questo valore viene ritornato in *EAX*, e successivamente tramite *MOV* viene copiato in *local_b0*.

```
00401300 e8 73 11        CALL       USER32.DLL::GetWindowLongA
         00 00
00401305 83 ec 08        SUB        ESP,0x8
00401308 89 85 54        MOV        dword ptr [EBP + local_b0],EAX
         ff ff ff
```

4.3 InitProgStruct (FUN_00401aab)

Ora che abbiamo descritto il parametro *lParam* della *WindProc*, possiamo tornare ad analizzare più nel dettaglio la funzione *FUN_00401aab* prececedentemente incontrata nel *WinMain*.

```
4  undefined4 * __cdecl FUN_00401aab(undefined4 param_1)
5
6  {
7    DAT_00406010 = 0;
8    DAT_00406014 = 1000;
9    DAT_0040601c = 0x708;
10   FUN_004023c0(&DAT_00406028,0x80,
11                "WARNING: there will be no shutdown without the proper unlock code!");
12   FUN_004023c0(&DAT_004060a8,0x10," 0 seconds");
13   DAT_00406020 = 0;
14   _DAT_00406024 = param_1;
15   return &DAT_00406010;
16 }
```

Ora sappiamo che il valore di ritorno è un puntatore ad una struttura *ProgStruct*. Possiamo osservare che viene ritornato l'indirizzo di *DAT_00406010*, e che tutti gli altri *DAT* sono collocati su indirizzi contigui in memoria; ciò ci suggerisce che questi fanno chiaramente riferimento ai vari campi della struttura *ProgStruct*.

Andiamo quindi a modificare questa funzione rinominandola in *InitProgStruct*, e modificando il tipo di ritorno in **ProgStruct*.

```
2  ProgStruct * __cdecl InitProgStruct(undefined4 param_1)
3
4  {
5    ProgStruct._0_4_ = 0;
6    ProgStruct._4_4_ = 1000;
7    ProgStruct._12_4_ = 0x708;
8    FUN_004023c0((char *)&ProgStruct.field_0x18,0x80,
9                 "WARNING: there will be no shutdown without the proper unlock code!"
10   FUN_004023c0((char *)&ProgStruct.field_0x98,16," 0 seconds");
11   ProgStruct._16_4_ = 0;
12   ProgStruct._20_4_ = param_1;
13   return &ProgStruct;
14 }
15
```

Possiamo ora sfruttare il codice fornito dal decompilatore per definire alcuni campi della struttura. In particolare, i campi ai byte *0*, *4*, *12* e *16* sono tipi di dato interi (*init0*, *init1000*, *init1800*, *init0_2*).

Il campo al *byte 20* viene assegnato al parametro di input *param_1*, ma non avendo ancora informazioni sul suo tipo di dato lo settiamo come un puntatore generico (*void* datPtr)*.

4.4 PrintString (FUN_004023c0)

Altri due campi della struttura vengono passati in input alla funzione *FUN_004023c0,* che al suo interno invoca a sua volta la funzione *_vsnprintf.*

```
int _vsnprintf(
   char *buffer,
   size_t count,
   const char *format,
   va_list argptr
);
```

Questa accetta un puntatore ad un elenco di argomenti, formatta i dati e scrive fino a `count` caratteri nella memoria a cui punta `buffer`. Rinominiamo quindi questa funzione in *PrintString*, ed analizziamo i parametri che gli vengono passati.

```
 8   PrintString((char *)&ProgStruct.field_0x18,128,
 9               "WARNING: there will be no shutdown without the proper unlock code!");
10   PrintString((char *)&ProgStruct.field_0x98,16," 0 seconds");
```

Il primo parametro è un *char**. Viene passato alla funzione l'indirizzo di *field_0x18*, che sarà quindi un *array di caratteri*. Come secondo parametro viene passato il valore numerico *128*. Quindi sappiamo che il campo della struttura al *byte 24* è un *CHAR[128]*. Il terzo parametro passato è la stringa contenente il messaggio di warning che viene mostrato sulla finestra dell'applicazione. Definiamo quindi questo campo della struttura come *warningStr* di tipo *CHAR[128]*

Seguendo lo stesso ragionamento possiamo anche definire il campo al *byte 152*, come un *CHAR[16]*. Denominiamo questo campo *remainingSecStr*, in quanto contiene una stringa che specifica il valore dei secondi rimanenti.

```
 2 ProgStruct * __cdecl InitProgStruct(void *param_1)
 3
 4 {
 5   ProgStruct.init0 = 0;
 6   ProgStruct.updateInt = 1000;
 7   ProgStruct.timerValue = 1800;
 8   PrintString(ProgStruct.warningStr,128,
 9               "WARNING: there will be no shutdown without the proper unlock code!");
10   PrintString(ProgStruct.remainingSecStr,16," 0 seconds");
11   ProgStruct.init0_2 = 0;
12   ProgStruct.datPtr = param_1;
13   return &ProgStruct;
14 }
```

E' necessario ora andare a definire i restanti campi della struttura *ProgStruct*; a tale scopo continuiamo con l'analisi della *WindProc*, in particolare riguardo la gestione del messaggio *WM_CREATE*

4.5 WindProc - WM_CREATE

Vediamo dal codice che vengono create 3 Finestre di classe *Edit*, iterando la chiamata *CreateWindowExA* in un ciclo do-while. Viene ad ogni iterazione incrementato il parametro *hMenu*, avendo così degli *EditBox* con identificativi 1,2 e 3.

```
hMenu = (HMENU)1;
do {
  pHVar4 = CreateWindowExA(0,"EDIT",(LPCSTR)0,0x50802002,0,0,0,0,hwnd,hMenu,hInstance,
                           (LPVOID)0);
  *(HWND *)(struct_base + 168 + (int)hMenu * 4) = pHVar4;
  if (pHVar4 == (HWND)0x0) {
    FatalAppExitA(0,"CreateWindow failed");
  }
  hMenu = (HMENU)((int)&hMenu->unused + 1);
} while (hMenu != (HMENU)4);
```

CreateWindowExA ritorna in *EAX*, e tale valore viene poi copiato nell'indirizzo `[ESI + EBX*4 + 168]`. *EBX* è il registro che mantiene l'identificativo dell' *EditBox* creato, mentre *ESI* tiene traccia della base della struttura *ProgStruct*.

```
004017c2 89 84 9e        MOV        dword ptr [ESI + EBX*4 + 168],EAX
         a8 00 00 00
004017c9 85 c0           TEST       EAX,EAX
004017cb 0f 84 fe        JZ         LAB_004018cf
         00 00 00

                     while_label                                    XREF[1]:
004017d1 83 c3 01        ADD        EBX,1
004017d4 83 fb 04        CMP        EBX,4
004017d7 75 85           JNZ        do_label
```

Stiamo quindi assegnando al byte `168+4*i` l'handle relativo all' *EditBox* con identificativo *i*. Quindi nella struttura *ProgStruct*, i campi ai byte *172*, *176* e *180* saranno di tipo *HWND* e contengono l'handle delle *EditBox* con identificativi *1*,*2* e *3*. Denominiamo tali campi come *hEdit1*, *hEdit2* e *hEdit3*.

Successivamente viene creata un'altra *EditBox* tramite *CreateWindowExA*, avente identificativo *5*. L'handle a tale finestra viene salvato nel campo al *byte 188* della struttura *ProgStruct*; definiamo quindi questo campo come *hEdit5* di tipo *HWND*.

```
004017eb c7 44 24       MOV       dword ptr [ESP + fn_arg10],5
         24 05 00
         00 00
                                   ...
0040182a c7 44 24       MOV       dword ptr [ESP + fn_arg2],EDIT
         04 00 50
         40 00
00401832 c7 04 24       MOV       dword ptr [ESP]=>fn_arg1,0x0
         00 00 00 00
00401839 e8 8a 0c       CALL      USER32.DLL::CreateWindowExA
         00 00
0040183e 83 ec 30       SUB       ESP,0x30
00401841 89 86 bc       MOV       dword ptr [ESI + 188],EAX
         00 00 00
```

Successivamente viene creata un'ulteriore finestra, questa volta di tipo *BUTTON*. Questo bottone viene istanziato con *id 4* e *titolo "Go"*, tramite *CreateWindowsExA*. L'handler ritornato viene salvato nel *byte 184* della struttura dati. Definiamo questo campo come *hButton4* di tipo *HWND*.

```
0040188a c7 44 24       MOV       dword ptr [ESP + fn_arg3],Go
         08 19 50
         40 00
00401892 c7 44 24       MOV       dword ptr [ESP + fn_arg2],BUTTON
         04 1c 50
         40 00
0040189a c7 04 24       MOV       dword ptr [ESP]=>fn_arg1,0
         00 00 00 00
004018a1 e8 22 0c       CALL      USER32.DLL::CreateWindowExA
         00 00
004018a6 83 ec 30       SUB       ESP,48
004018a9 89 86 b8       MOV       dword ptr [ESI + 184],EAX
         00 00 00
```

Se tutte le finestre sono state create correttamente, si prosegue sulla label *window_success*, in cui vengono chiamate le funzioni `FUN_00401b74` e `FUN_00401b20`.

```
004018af 85 c0          TEST      EAX,EAX
004018b1 74 38          JZ        create_win_fail

                        window_success
004018b3 e8 bc 02       CALL      FUN_00401b74
         00 00
004018b8 8b 75 08       MOV       ESI,dword ptr [EBP + hwnd]
004018bb 89 34 24       MOV       dword ptr [ESP]=>fn_arg1,ESI
004018be e8 5d 02       CALL      FUN_00401b20
         00 00
```

```
if (pHVar4 == (HWND)0x0) {
  FatalAppExitA(0,"CreateWindow failed");
}
FUN_00401b74();
FUN_00401b20(param_1);
return 0;
```

SetRemainingTime (FUN_00401b20)

La funzione *FUN_00401b20*, quando invocata, aggiorna i valori mostrati a schermo relativi al countdown. In particolare, tramite *PrintString* salva nel campo *remainingSecStr* il messaggio sui secondi rimanenti. Successivamente calcola il tempo rimanente in termini di Giorni, Ore e Minuti ed aggiorna il valore degli *EditBox* ad essi associati tramite l'API *SetDlgItemInt*

```
2  void SetRemainingTime(void)
3
4  {
5    HWND hDlg;
6    uint uVar1;
7    UINT uValue;
8    uint iVar2;
9    UINT uValue_00;
10
11   hDlg = ProgStruct.hwnd;
12   iVar2 = ProgStruct.init1800 - ProgStruct.init0;
13   uVar1 = ProgStruct.init1000 * 60;
14   PrintString(ProgStruct.remainingSecStr,16,"%2ld seconds");
15   uValue = 0;
16   for (uVar1 = (iVar2 * 1000) / uVar1; 1439 < uVar1; uVar1 = uVar1 - 1440) {
17     uValue = uValue + 1;
18   }
19   uValue_00 = 0;
20   for (; 59 < uVar1; uVar1 = uVar1 - 60) {
21     uValue_00 = uValue_00 + 1;
22   }
23   SetDlgItemInt(hDlg,1,uValue,0);
24   SetDlgItemInt(hDlg,2,uValue_00,0);
25   SetDlgItemInt(hDlg,3,uVar1,0);
26   return;
27 }
```

Da questa funzione possiamo inoltre ridefinire alcuni campi della struttura. In *iVar2* viene sicuramente salvato il tempo rimanente, per cui *init1800* rappresenta il valore iniziale del timer (*timerValue*), e *init0* mantiene il tempo trascorso in secondi (*timeElapsed*).

```
11   hwnd = ProgStruct.hwnd;
12   remainingTime = ProgStruct.timerValue - ProgStruct.timeElapsed;
13   minutes = ProgStruct.init1000 * 60;
14   PrintString(ProgStruct.remainingSecStr,16,"%2ld seconds");
15   days = 0;
16   for (minutes = (remainingTime * 1000) / minutes; 1439 < minutes; minutes = minutes - 1440) {
17     days = days + 1;
18   }
19   hours = 0;
20   for (; 59 < minutes; minutes = minutes - 60) {
21     hours = hours + 1;
22   }
23   SetDlgItemInt(hwnd,1,days,0);
24   SetDlgItemInt(hwnd,2,hours,0);
25   SetDlgItemInt(hwnd,3,minutes,0);
```

StartTimer (FUN_0040b20)

La funzione *FUN_0040b20* si occupa dell'inizializzazione del *Timer*, utilizzando la funzione di libreria *SetTimer*.

```
UINT_PTR SetTimer(
  [in, optional] HWND      hWnd,
  [in]           UINT_PTR  nIDEvent,
  [in]           UINT      uElapse,
  [in, optional] TIMERPROC lpTimerFunc
);
```

Il parametro *uElapse* specifica l'intervallo di timeout in millisecondi, mentre l'ultimo parametro rappresenta un puntatore ad una funzione che verrà notificata allo scadere del periodo *uElapse* specificato.

```
2 void __cdecl StartTimer(HWND hwnd)
3
4 {
5   ProgStruct.timer = SetTimer(hwnd,0,ProgStruct.init1000,(TIMERPROC)&LAB_00401c7b);
6   return;
7 }
```

Nella funzione viene passato come terzo parametro il campo *init1000* della struttura, per cui *SetTimer* invierà una notifica alla funzione ogni secondo.

Per quanto riguarda il parametro *lpTimerFunc*, Ghidra interpreta erroneamente la funzione passata come una label, in quanto non rileva nessuna CALL o JUMP esplicita verso di essa.

Andiamo quindi a definire manualmente una funzione *TimerHandler* tramite *Create Function* su *LAB_00401c7b*.

Inoltre, ora conosciamo più nello specifico il significato del campo *init1000*, che specifica l'intervallo di notifica della funzione *TimerHandler*. Rinominiamo quindi tale campo della struttura in *updateInt*.

```
2 void __cdecl StartTimer(HWND hwnd)
3
4 {
5   ProgStruct.timer = SetTimer(hwnd,0,ProgStruct.updateInt,TimerHandler);
6   return;
7 }
```

TimerHandler (LAB_00401c7b)

Continuiamo ora l'analisi andando ad esaminare la funzione *TimerHandler*. Qui vediamo che viene incrementato di *1* il valore del campo *timeElapsed*; ciò risulta coerente con il comportamento descritto, in quanto *TimerHandler* viene effettivamente notificata ogni secondo.

```
2  void TimerHandler(HWND hwnd)
3
4  {
5    uint uVar1;
6
7    uVar1 = ProgStruct.timeElapsed + 1;
8    ProgStruct.timeElapsed = uVar1;
9    if (ProgStruct.init0_2 != 0) {
10     SetRemainingTime();
11   }
12   RedrawWindow(hwnd,(RECT *)0x0,(HRGN)0x0,5);
13   if ((ProgStruct.init0_2 != 0) && ((uint)ProgStruct.timerValue <= uVar1)) {
14     (*(code *)ProgStruct.datPtr)(&ProgStruct);
15     return;
16   }
17   return;
18 }
```

Qui vediamo che vengono effettuati alcuni controlli sul campo *init0_2* della struttura *ProgStruct*. Non abbiamo ancora esaminato il contenuto effettivo di tale campo, per cui non possiamo ancora definire con chiarezza che tipo di condizione questo determina.

A tale scopo, torniamo quindi alla *WindProc*, in particolare riguardo la gestione del messaggio *WM_COMMAND* finora non analizzato.

4.6 WindProc - WM_COMMAND

Viene effettuato un controllo sul valore di *lParam*, e se questo corrisponde all'handler del Button, viene invocata la funzione *FUN_00401dd6*

```
125     if (uMsg == WM_COMMAND) {
126       if (((short)(wParam_00 >> 0x10) == 0) && ((HWND)lParam_00 == LVar2->hButton4)) {
127         FUN_00401dd6();
128         return 0;
129       }
```

ButtonHandler (FUN_00401dd6)

Vediamo che se *init0_2* vale 0, viene settato a *1* e viene scritto sul *Button* la stringa *"Stop"*. Al contrario se *init0_2* vale *1*, viene settato a *0* e viene scritto sul *Button* il valore *"Go"*.

```
 8   if (ProgStruct.init0_2 == 0) {
 9     SetDlgItemTextA(ProgStruct.hwnd,4,"Stop");
10     ProgStruct.init0_2 = 1;
                    ...
17   else {
18     ProgStruct.init0_2 = 0;
19     SetDlgItemTextA(ProgStruct.hwnd,4,"Go");
```

Risulta evidente quindi che questa funzione si occupa di gestire l'evento in cui viene cliccato il bottone. Possiamo osservare che il campo *init2_0* viene utilizzato sostanzialmente come un *flag* che specifica se in un certo istante il *timer* è in esecuzione oppure no. Rinominiamo quindi *init2_0* in *timerFlag.*

Quindi se il timer non è ancora attivo (*timerFlag == 0*) e viene premuto il pulsante, si utilizza l'API *SetDlgItemMessageA* per impostare i messaggi sugli *EditBox* e sul *Button*. Successivamente viene invocata la funzione *FUN_00401cf4*

```
 2 void ButtonHandler(void)
 3
 4 {
 5   HWND hDlg;
 6
 7   hDlg = ProgStruct.hwnd;
 8   if (ProgStruct.timerFlag == 0) {
 9     SetDlgItemTextA(ProgStruct.hwnd,4,"Stop");
10     ProgStruct.timerFlag = 1;
11     SendDlgItemMessageA(hDlg,1,0xcf,1,0);
12     SendDlgItemMessageA(hDlg,2,0xcf,1,0);
13     SendDlgItemMessageA(hDlg,3,0xcf,1,0);
14     SendDlgItemMessageA(hDlg,5,0xcf,1,0);
15     FUN_00401cf4();
16   }
```

StartCountdown (FUN_00401cf4)

Questa funzione utilizza l'API *GetDlgItemInt* per leggere i valori di *Giorni*, *Ore* e *Minuti* inserite dall'utente negli appositi *EditBox*.

Partendo da tali valori, calcola il totale in secondi e lo assegna al campo *timerValue* della struttura *ProgStruct*

```
2  void StartCountdown(void)
3
4  {
5    HWND hDlg;
6    UINT UVar1;
7    int iVar2;
8    int local_14;
9
10   hDlg = ProgStruct.hwnd;
11   UVar1 = GetDlgItemInt(ProgStruct.hwnd,1,&local_14,0);
12   iVar2 = 0;
13   if (local_14 != 0) {
14     iVar2 = UVar1 * 0x5a0;
15   }
16   UVar1 = GetDlgItemInt(hDlg,2,&local_14,0);
17   if (local_14 != 0) {
18     iVar2 = iVar2 + UVar1 * 0x3c;
19   }
20   UVar1 = GetDlgItemInt(hDlg,3,&local_14,0);
21   if (local_14 != 0) {
22     iVar2 = iVar2 + UVar1;
23   }
24   ProgStruct.timerValue = (uint)(iVar2 * ProgStruct.updateInt * 60) / 1000 + ProgStruct.timeElapsed;
25   SetRemainingTime();
26   return;
27 }
```

Di fatto quindi in questa funzione viene *inizializzato il countdown*, andando a settare il campo contenente il valore del timer, allo scadere del quale il computer verrà spento.

Completata quindi l'analisi sulla gestione del messaggio *WM_COMMAND*, possiamo tornare ad esaminare la funzione *TimerHandler*.

4.7 TimerHandler

Ora sappiamo che viene effettuato un controllo su *timerFlag*, per verificare se il timer è attivo oppure no.

```
2  void TimerHandler(HWND hwnd)
3
4  {
5    uint uVar1;
6
7    uVar1 = ProgStruct.timeElapsed + 1;
8    ProgStruct.timeElapsed = uVar1;
9    if (ProgStruct.timerFlag != 0) {
10     SetRemainingTime();
11   }
12   RedrawWindow(hwnd,(RECT *)0x0,(HRGN)0x0,5);
13   if ((ProgStruct.timerFlag != 0) && ((uint)ProgStruct.timerValue <= uVar1))
14     (*(code *)ProgStruct.datPtr)(&ProgStruct);
15     return;
16   }
17   return;
18 }
```

Se il countdown è attivo (*timerFlag!=0*) viene invocata *SetRemainingTime*, che come già visto và ad aggiornare nella finestra il valore del tempo rimanente.

Successivamente si verifica in un costrutto *if* se il timer è stato avviato, e se il suo valore iniziale *timerValue* è inferiore al tempo trascorso *uvar1*. Si sta in pratica controllando se il timer, dopo essere stato avviato, è scaduto.

Ghidra non riconosce correttamente la funzione invocata allo scadere del timer, per cui andiamo ad analizzare direttamente il codice assembly. Qui vediamo che è presente una *CALL* alla funzione puntata dal campo *datPtr* di *ProgStruct*

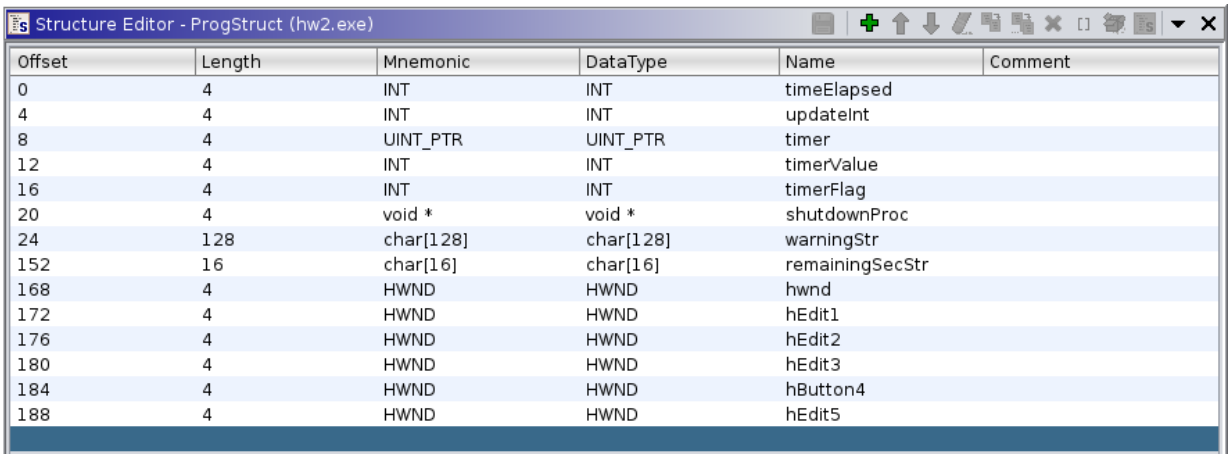
```
00401cca 3b 1d 1c        CMP        EBX,dword ptr [ProgStruct.timerValue]
         60 40 00
00401cd0 73 07           JNC        timer_expired
                                 ...
                     timer_expired                                   XREF
00401cd9 c7 04 24        MOV        dword ptr [ESP]=>local_1c,ProgStruct
         10 60 40 00
00401ce0 ff 15 24        CALL       dword ptr [ProgStruct.datPtr]
         60 40 00
00401ce6 8b 5d fc        MOV        EBX,dword ptr [EBP + local_8]
00401ce9 c9              LEAVE
00401cea c2 10 00        RET        0x10
```

Tale campo non è stato ancora descritto completamente, ma conoscendo il funzionamento del programma sappiamo che allo scadere del timer dovrà essere eseguita una qualche funzione che andrà a spegnere la macchina.

Possiamo quindi ridefinire il campo *datPtr* della struttura come *shutdownProc* di tipo *VOID**, in quanto punterà sicuramente alla funzione in cui viene eseguito lo shutdown.

ProgStruct

Avendo descritto anche il campo relativo alla procedura di shutdown, abbiamo terminato l'analisi della struttura *ProgStruct*, che in definitiva è stata definita come segue:



Offset	Length	Mnemonic	DataType	Name	Comment
0	4	INT	INT	timeElapsed	
4	4	INT	INT	updateInt	
8	4	UINT_PTR	UINT_PTR	timer	
12	4	INT	INT	timerValue	
16	4	INT	INT	timerFlag	
20	4	void *	void *	shutdownProc	
24	128	char[128]	char[128]	warningStr	
152	16	char[16]	char[16]	remainingSecStr	
168	4	HWND	HWND	hwnd	
172	4	HWND	HWND	hEdit1	
176	4	HWND	HWND	hEdit2	
180	4	HWND	HWND	hEdit3	
184	4	HWND	HWND	hButton4	
188	4	HWND	HWND	hEdit5	

ShutdownProcedure

Procediamo quindi con l'analisi della procedura di shutdown. Il campo *shutdownProc* della struttura viene settato tramite l'input della funzione *InitProgStruct*. Tale funzione è stata invocata all'interno del *WinMain*, ed il parametro passato è l'indirizzo del dato *DAT_00403000*.

Questo dato mantiene sicuramente una funzione, che però è stata definita nel campo *data* della memoria invece che in *.text*.

```
                    //
                    // data
                    // ram:00403000-ram:004031ff
                    //

                    DAT_00403000                                    XREF[2]:     004001ac(*),
                                                                                 WinMain:00401
00403000 53              ??         53h    S
00403001 83              ??         83h
00403002 ec              ??         ECh
00403003 58              ??         58h    X
```

Andiamo quindi ad effettuare il **Disassembling**, per poterne analizzare effettivamente il contenuto, e definiamo una nuova funzione *ShutdownProcedure*.

```
....
                    //
                    // data
                    // ram:00403000-ram:004031ff
                    //

                    ****************************************************************************
                    *                                FUNCTION                                  *
                    ****************************************************************************
                    void __stdcall ShutdownProcedure(int param_1)
    void              <VOID>        <RETURN>
    int               Stack[0x4]:4  param_1                          XREF[1]:     00403004(R)
                    ShutdownProcedure                          XREF[2]:        00400lac(*),
                                                                               WinMain:004019b8(*)
```

5 Codice di Sblocco

Analizzando il codice di *ShutdownProcedure*, possiamo vedere che viene effettuato un controllo su una serie di caratteri all'interno di un costrutto *if*. Solo se tutti i campi corrispondono ai caratteri specificati si entra nel body, in cui vengono invocate diverse funzioni.

```
48  if ((((((bVar8) && (cStack54 == '3')) && (cStack53 == 'R')) &&
49        ((cStack52 == 'n' && (cStack51 == 'E')))) &&
50       ((cStack50 == 'S' && ((cStack49 == 't' && ((char)uStack48 == '0')))))) &&
51      ((uStack48._1_1_ == '!' && (uStack48._2_1_ == '?')))) {
```

Ghidra non ci permette di analizzare il codice di tali funzioni, ma conoscendo il funzionamento del programma, siamo certi che tra queste ci sarà anche quella che andrà ad eseguire effettivamente lo shutdown del sistema.

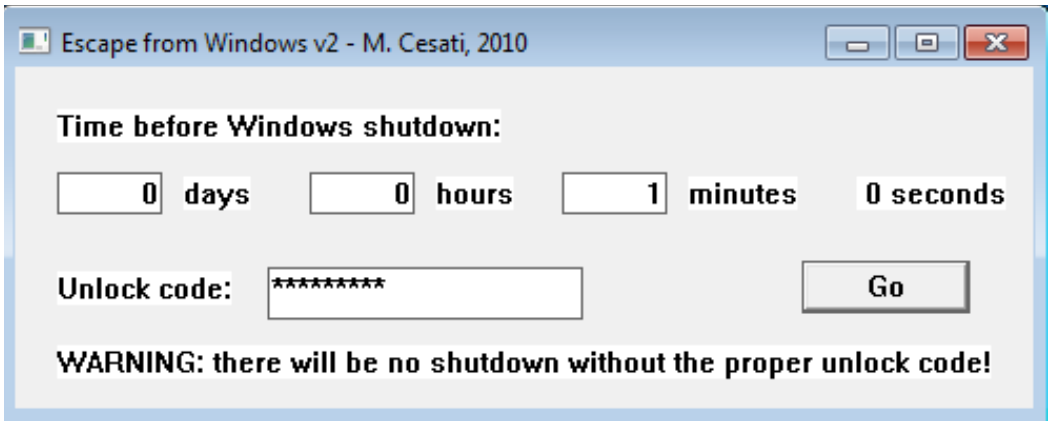
Questo ci indica che i caratteri controllati nell'*if*, rappresentano molto probabilmente il codice di sblocco che permette di spegnere la macchina allo scadere del timer.

Tali caratteri, compongono la stringa **3RnESt0!?**, per cui non resta che verificare se questo è effettivamente l'*Unlock Code* richiesto dal programma.

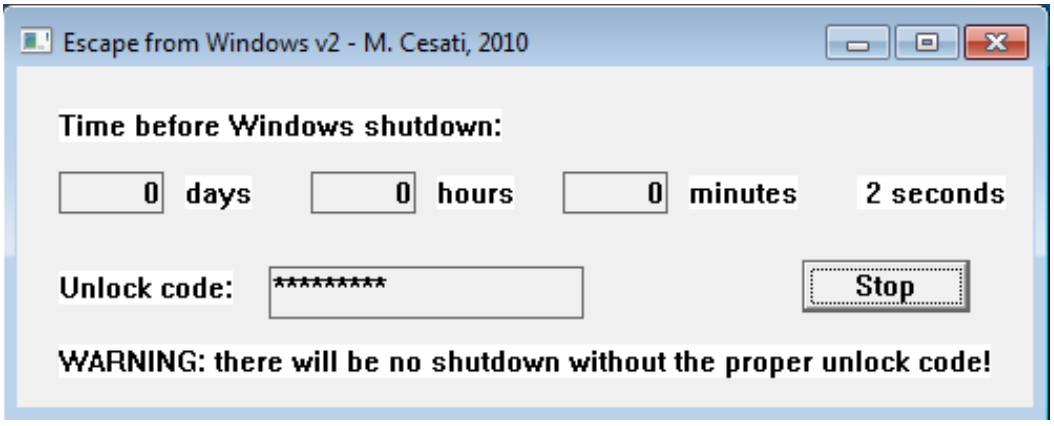
6 Verifica a Runtime

Vogliamo verificare se inserendo il codice *3RnESt0!?* viene eseguito lo shutdown del sistema invece di mostrare l'errore relativo al codice errato.

Inseriamo tale codice nell'*EditBox*, impostiamo un timer di 1 minuto, e lanciamo il programma cliccando sull'apposito *Bottone* "Go".



Attendiamo lo scadere del timer.



Vediamo che viene correttamente eseguito lo shutdown del sistema.

Questo ci conferma che il codice di sblocco che rende funzionale il programma è effettivamente ***3RnESt0!?***