### **Homework 2 Malware Analysis**

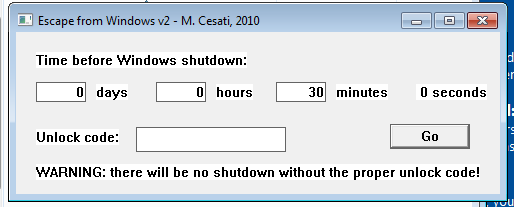
Danilo Dell’Orco

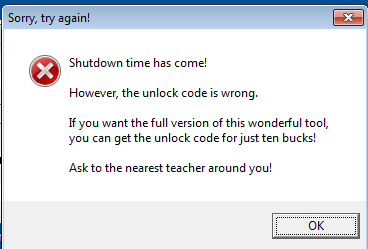
0300229

Novembre 14, 2021

# **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.

# **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,

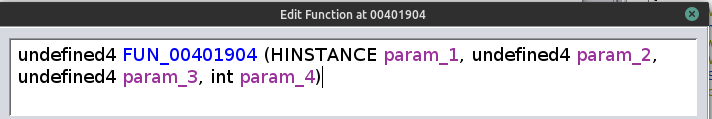
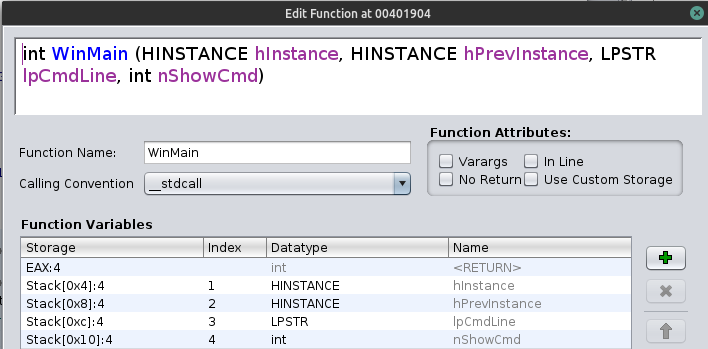
[in] HINSTANCE hPrevInstance,

[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*.

# 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;

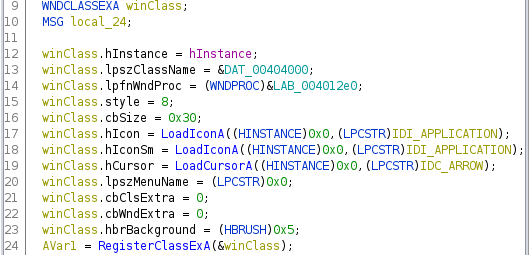
LPCSTR lpszClassName;

HICON hIconSm;

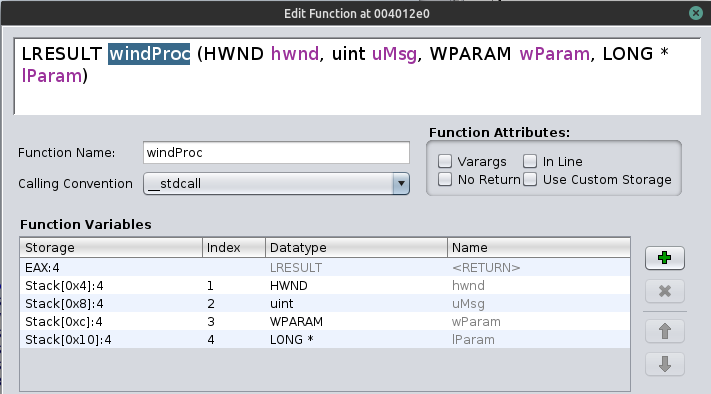
} WNDCLASSEXA, \*PWNDCLASSEXA, \*NPWNDCLASSEXA, \*LPWNDCLASSEXA;

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.

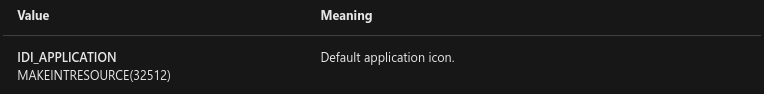
Dal codice possiamo osservare come vengano assegnati tutti i parametri di questa strutura.



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**



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

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

HICON LoadIconA(

[in, optional] HINSTANCE hInstance,

[in] LPCSTR lpIconName

);

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

);

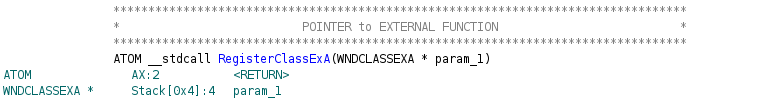
Anche in questo caso viene impostata un cursore di default tramite il codice *0x7f00*, che corrisponde a *IDC\_ARROW.* Specifichiamo tale valore utilizzando *Set Equate*

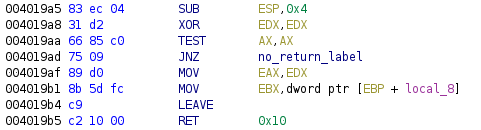


Successivamente viene invocata la funzione *RegisterClassExA*, passando come parametro il puntatore alla struttura appena definita.

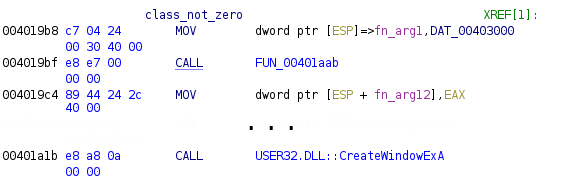
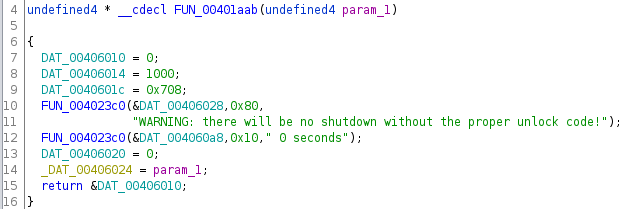


Questa funzione ritorna un *ATOM* che identifica la classe registrata in AX, quindi nei 16 bit meno significativi del registro EAX.

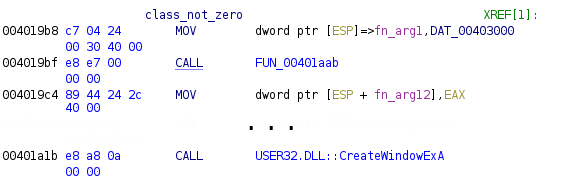
 Si effettua un controllo sulla classe registrata, eseguendo la procedura di ritorno in caso di errore.



Successivamente, viene chiamata la funzione *FUN\_00401aab*, passando in input un certo dato *DAT\_0040300*.

 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.

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*.

**

*CreateWindowExA* crea una istanza della classe di finestra specificata.

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*.

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

);

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*.

# Window Procedure

## **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.

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.

0x5 → WM\_SIZE

0x1 → WM\_CREATE

0x2 → WM\_DESTROY

0xf → WM\_PAINT

0x111 → WM\_COMMAND

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.

## **WindProc**

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

 Successivamente viene chiamata la funzione *GetWindowLongA*.

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

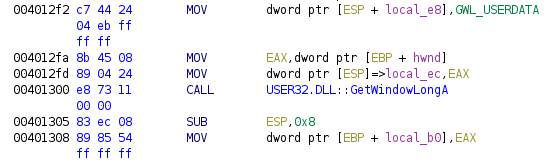
LONG GetWindowLongA(

[in] HWND hWnd,

[in] int nIndex

);

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*

Questo valore definito dal programmatore, viene settato tramite l’API *SetWindowLongA*, utilizzata nella gestione del messaggio *WM\_CREATE*.

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*.

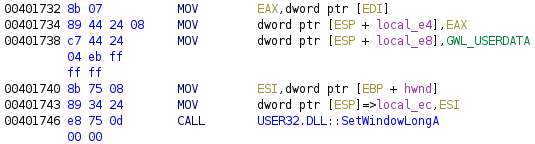
LONG SetWindowLongA(

[in] HWND hWnd,

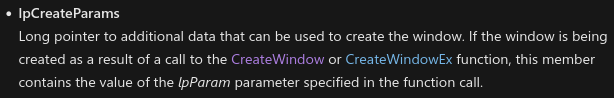
[in] int nIndex,

[in] LONG dwNewLong

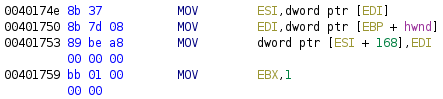
);

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*.

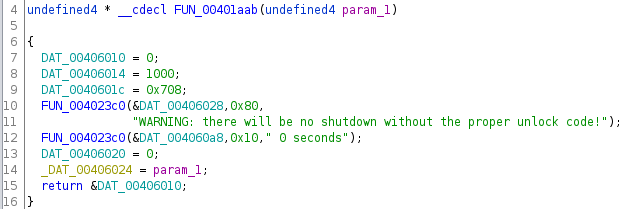
Quindi in definitiva, il valore associato a *GWL\_USERDATA* corrisponde a *lpParam* che è stato passato come ultimo parametro nella *CreateWindowExA* invocata nel *WinMain*.

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.

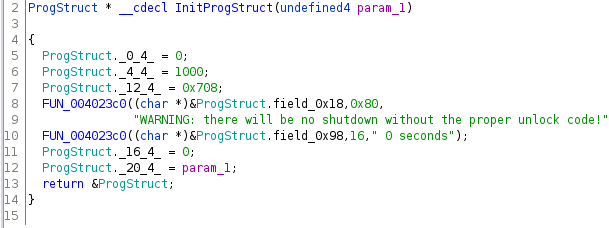
In definitva, *GetWindowLongA* ritorna un puntatore alla struttura *ProgStruct*. Questo valore viene ritornato in *EAX*, e successivamente tramite *MOV* viene copiato in *local\_b0*.

## **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*.

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*.

 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)*.

## 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,

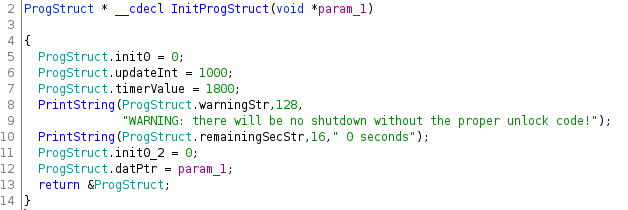
va\_list argptr

);

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.

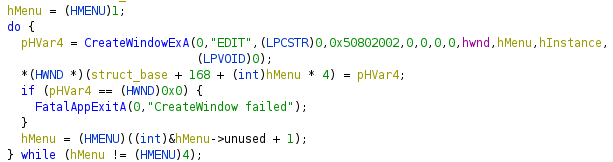
 Il primo parametro è un *char\**. Viene passato alla funzione l’indirizzo di *field\_0x18*, che sarà quindi un *array di caratteri*. Come secondo parametro viene passato il valore numerico *128*. Quindi sappiamo che il campo della struttura al *byte 24* è un *CHAR[128]*. Il terzo parametro passato è 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]*

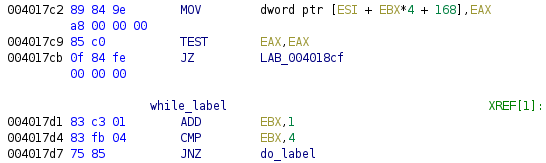
Seguendo lo stesso ragionamento possiamo anche definire il campo al *byte 152*, come un *CHAR[16]*. Denominiamo questo campo *remainingSecStr*, in quanto contiene una stringa che specifica il valore dei secondi rimanenti.

 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*

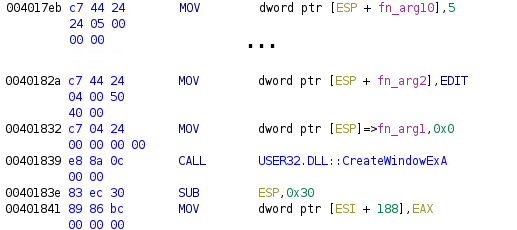
## **WindProc - WM\_CREATE**

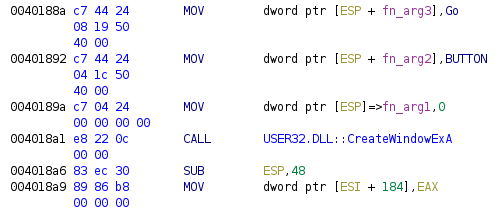
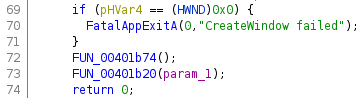
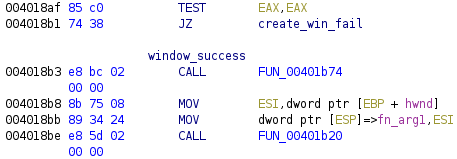
Vediamo dal codice che vengono create 3 Finestre di classe *Edit*, iterando la chiamata *CreateWindowExA* in un ciclo do-while. Viene ad ogni iterazione incrementato il parametro *hMenu*, avendo così degli *EditBox* con identificativi 1,2 e 3.

*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*.

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*.

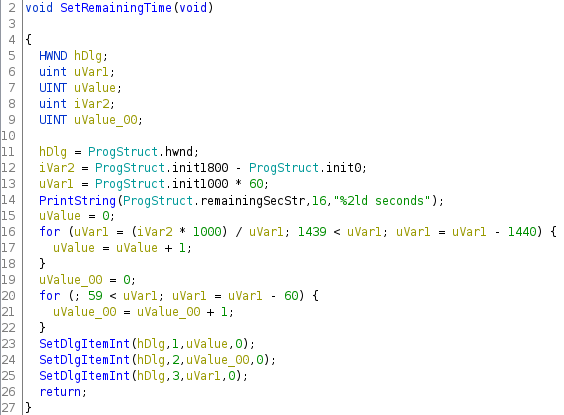
Successivamente viene creata un’altra *EditBox* tramite *CreateWindowExA*, avente identificativo *5*. L’handle a tale finestra viene salvato nel campo al *byte 188* della struttura *ProgStruct*; definiamo quindi questo campo come *hEdit5* di tipo *HWND*.

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*.

 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.

## **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*

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*).

## **StartTimer (FUN\_0040b20)**

La funzione *FUN\_0040b20* si occupa dell’inizializzazione del *Timer*, utilizzando la funzione di libreria *SetTimer*.

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.

UINT\_PTR SetTimer(

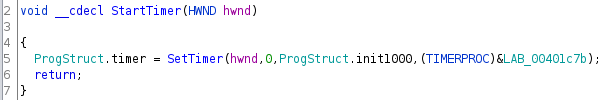
[in, optional] HWND hWnd,

[in] UINT\_PTR nIDEvent,

[in] UINT uElapse,

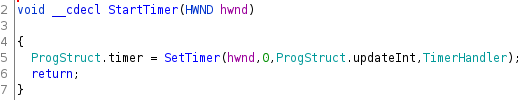
[in, optional] TIMERPROC lpTimerFunc

);

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

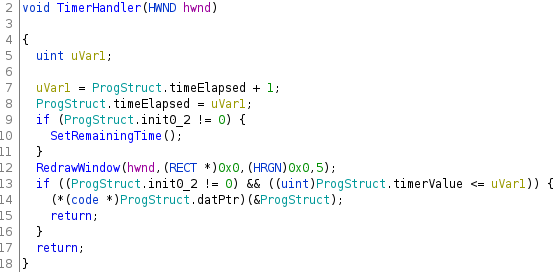
Per quanto riguarda il parametro *lpTimerFunc*, Ghidra interpreta erroneamente la funzione passata come una label, in quanto non rileva nessuna CALL o JUMP esplicita verso di essa.

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*.

## 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.

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.

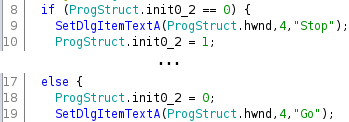
A tale scopo, torniamo quindi alla *WindProc*, in particolare riguardo la gestione del messaggio *WM\_COMMAND* finora non analizzato.

## 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*

## 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”*.

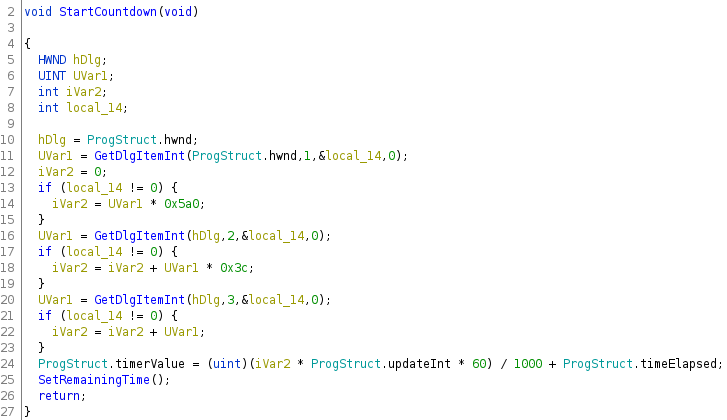
Risulta evidente quindi che questa funzione si occupa di gestire l’evento in cui viene cliccato il bottone. Possiamo osservare che il campo *init2\_0* viene utilizzato sostanzialmente come un *flag* che specifica se in un certo istante il *timer* è in esecuzione oppure no. Rinominiamo quindi *init2\_0* 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*

## StartCountdown (FUN\_00401cf4)

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

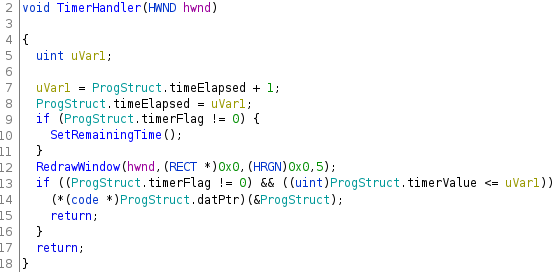
Partendo da tali valori, calcola il totale in secondi e lo assegna al campo *timerValue* della struttura *ProgStruct*

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.*

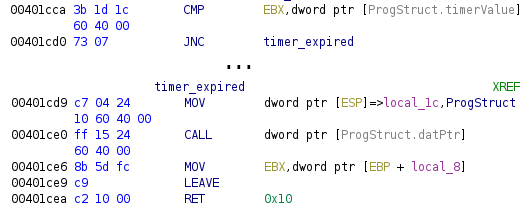
## TimerHandler

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

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*

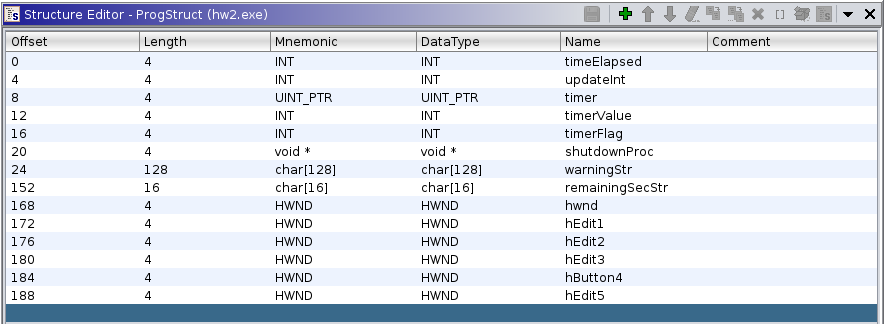


Tale campo non è stato ancora descritto completamente, ma conoscendo il funzionamento del programma sappiamo che allo scadere del timer dovrà essere eseguita una qualche funzione che andrà a spegnere la macchina.

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.

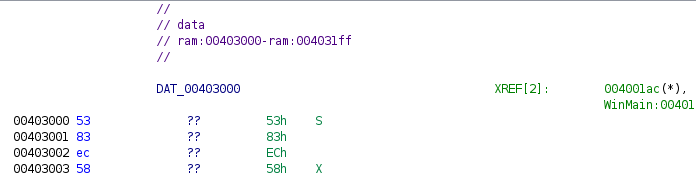
## **ProgStruct**

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:



## ShutdownProcedure

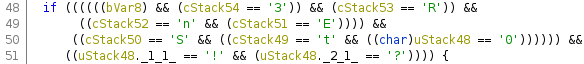
Procediamo quindi con l’analisi della procedura di shutdown. Il campo *shutdownProc* della struttura viene settato tramite l’input della funzione *InitProgStruct*. Tale funzione è stata invocata all’interno del *WinMain*, ed il parametro passato è l’indirizzo del dato *DAT\_00403000*.

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

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

## 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.

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.

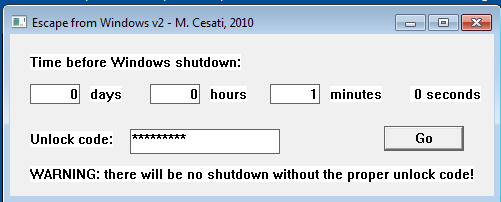
Questo ci indica che i caratteri controllati nell’*if*, rappresentano molto probabilmente il codice di sblocco che permette di spegnere la macchina allo scadere del timer.

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

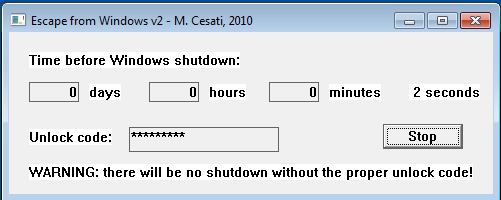
## Verifica a Runtime

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!?***