1 Introduzione ed obiettivi

Il seguente documento riporta i passi logici seguiti e le informazioni raccolte per quanto riguarda l'analisi del file eseguibile **hw3.exe**. Il file è stato analizzato mediante l'uso dei seguenti tools

- disassemblatore Ghidra
- debugger OllyDbg

L'obiettivo dell'analisi è quello di **trovare il codice di sblocco che rende funzionante l'applicazione**, nel rapporto vengono riassunti i passi logico-deduttivi fatti e le informazioni raccolte durante le attività di reverse code engineering.
La stesura del documento utilizza varie assunzioni che derivano dalla precedente analisi dell'eseguibile **hw2.exe**, in quanto questo presenta diverse analogie con il file **hw3.exe**.

2 Raccolta informazioni generali sul file

In una prima fase, sono state ricostruite tutte le informazioni che erano già state trovare durante l'analisi del file **hw2.exe**, comunque necessarie per ricostruire il funzionamento generale del programma. Anche in questo caso, lanciando il file eseguibile, compare una finestra che permette di impostare un timeout per lo spegnimento della macchina host. Se però non si fornirà il codice corretto che rende funzionale l'applicativo, lo spegnimento effettivo della macchina non avverrà. Sono quindi stati analizzati i blocchi fondamentali del programma e tutte le informazioni trovate, come ad esempio l'associazione dei nomi delle funzioni inserite in Ghidra, sono stati riportati anche in OllyDbg, mettendo le label sui corrispettivi indirizzi di memoria.

2.1 Ricerca del WinMain

La ricerca del `WinMain`, che è il punto d'ingresso per le applicazioni Windows basate su GUI, è avvenuta mediante Ghidra individuando il blocco di codice in cui è presente il **Messsage Loop**.
Consultando le funzioni importate dalla DLL `User32`, sono state trovate le API

- `GetMessageA`
- `TranslateMessage`
- `DispatchMessageA`

e consultando i riferimenti a tali API si vede che queste vengono chiamate una sola volta, nella funzione `FUN_004024e0` e quindi tale funzione è la `WinMain`.

2.1.1 Analisi del file mapping

La prima funzione chiamata all'interno della `WinMain` è la `FUN_00401560`. Tale funzione è stata analizzata avvalendosi del tool di de-compilazione di Ghidra:

- la prima funzione chiamata è la API `GetModuleFileNameA`, a cui viene passato come primo parametro il valore `NULL`. Questo ha l'effetto di recuperare il full path del file eseguibile del processo corrente
- viene poi invocata la `CreateFileA`, una API usata per creare o accedere in I/O un file. In questo caso, dati i parametri che vengono passati in input, si richiede di aprire il file in lettura solo se questo esiste già
- la `GetModuleFileNameA` in caso di successo, restituisce un handle al file. Se tale handle è valido, segue la chiamata alla API `GetFileInformationByHandle`. Tale API permette di recuperare le informazioni legate ad un file, che vengono restituite nel secondo parametro, di tipo struttura `LPBY_HANDLE_FILE_INFORMATION`

A questo punto, viene salvata la parte bassa della taglia del file nella variabile globale `DAT_004070e8` che è stata rinominata come `low_file_size`, per poi proseguire con altre 2 chiamate ad API:

- prima, viene invocata la `CreateFileMappingA`, che crea o apre un oggetto file mapping per un file. Il 3° parametro della funzione è `flProtect`, che specifica la protezione delle regioni di memoria del file mapping ed in questo caso i flag specificati dicono che il file corrispondente all'handle è eseguibile ed inoltre che le pagine siano mappate per accessi read-only o copy-on-write.

- se la funzione ha successo, viene restituito un handle al file mapping appena creato, utilizzato come primo parametro nella chiamata alla `MapViewOfFile`. L'API mappa la vista del file mapping nello spazio d'indirizzamento del processo corrente. Il secondo parametro della funzione specifica i permessi di accesso alle pagine mappate in memoria, che in questo caso viene settato con la macro `FILE_MAP_READ`, ovvero sono possibili solo accessi read-only.
  Il risultato di questa API è l'indirizzo iniziale della vista della file map, che viene salvata nella variabile globale `DAT_004070e4`. Tale variabile è stata rinominata `file_map_startaddr`

Questa funzione è stata rinominata come `create_file_map`

2.1.2 Inizializzazione della struttura dati

Esattamente come accadeva per il file **hw2.exe**, vi è la funzione `FUN_00401830` in cui avviene l'inizializzazione della struttura di dati che verrà usata nel programma, rinominata quindi come `app_ds`. Infatti, all'interno di tale funzione, vi è l'inizializzazione:

- del valore del timeout corrente, inizializzato a 0, ad offset 0 della struttura dati
- della lunghezza del tick, inizializzato a 1000, ad offset 4
- del timeout per l'applicazione, inizializzato a 1800, ad offset 12
- dei due buffer di caratteri, uno di 128 byte ad offset 24 ed uno di 16 caratteri ad offset 152
- del capo ad offset 20, a cui viene associato il valore del parametro passato come input dalla WinMain, ovvero la variabile globale `DAT_004040e0`.

La struttura dati, a cui è stato assegnato il nome `app_ds`, è stata inizialmente creata con una taglia di 192 byte, come avvenuto per il file precedente.
Consultando le variabili globali nel .bss, è possibile notare che le due variabili definite in precedenza, ovvero `low_file_size` e `file_map_startaddr` si trovano esattamente sotto l'ultimo campo della struct, quindi è ragionevole supporre che tali variabili siano in realtà due campi della struttura, rispettivamente ad offset 200 e 196.

2.1.3 Caricamento dinamico della `OutputDebugString`

Dopo l'inizializzazione della struttura dati, vi è una chiamata alla funzione `FUN_004016f0`. Aprendo tale funzione con Ghidra, sono presenti diversi meccanismi anti-disassembling tutti basati sull'utilizzo della sequenza di istruzioni `eb ff c0 48` che realizza uno dei modi per ottenere una misura di disassembling impossibile. Per risolvere il problema, si è proceduto come segue:

- si pulisce il codice disassemblato, tramite l'opzione "clear code bytes"
- si riprende il disassemblaggio dalla prima istruzione dopo la "48", così da ottenere il corretto codice
- si sostituisce ad ognuna delle istruzioni che compongono il codice `eb ff c0 48` con una `NOP` (codice operativo 90) per poter ottenere un codice de-compilato correttamente leggibile.

Si arriva quindi al decompilato in 1.
Consultando tale funzione, si capisce che

- viene scritta nella variabile automatica `local_2d`, rinominata `local_str` la stringa "kernel32.dll"
- chiamata la `LoadLibraryA`, quindi caricata la libreria `kernel32.dll` dinamicamente
- viene poi messa nello stesso array di prima la stringa "OutputDebugStringA"
- viene infine chiamata la `GetProcAddress`: questa API restituisce l'indirizzo della funzione `OutputDebugString`

Tale risultato viene salvato nella variabile globale `DAT_004070ec`, anche in questo caso la variabile è stata inclusa come campo della struttura dati, ad offset 204.

2.2 Analisi della `WinProc`

È stata poi definita la `WinProc`, ad indirizzo 00401de0. Aprendo la funzione ed avvalendosi del tool di decompilazione di Ghidra, è possibile vedere in prima analisi che la struttura del codice è molto simile a quella del file **hw2.exe**: i messaggi gestiti sono sempre i soliti, relativi alle varie fasi che compongono il ciclo di vita di un'applicazione basata su finestra per Windows:

- `WM_CREATE`, con codice 1;
- `WM_DESTROY`, con codice 2;
- `WM_SIZE`, con codice 5;
- `WM_PAINT`, con codice 15;
- `WM_COMMAND`, con codice 273

Ad esempio, all'interno del blocco della `WM_CREATE` vengono inizializzati tutti gli handles delle finestre e del bottone che compongono l'applicazione, per inserirli all'interno di un array nella struttura dati, ad offset 168. Sono stati quindi ricostruiti tutti i campi mancanti della struttura dati, riassunti in 2

3 Risoluzione delle misure anti-debugging

Una volta ricostruite le informazioni fondamentali per il programma all'interno di Ghidra, queste sono state riportate su OllyDbg ed è stato utilizzato questo tool per riuscire a trovare il codice di sblocco. Il primo problema che è stato affrontato riguarda la presenza di una o più tecniche anti-debugging presenti all'interno del codice: difatti, se si tenta di aprire il programma mediante OllyDbg e si lancia la sua esecuzione (f9), ci si aspetterebbe che venga mostrata la finestra per inserire il timeout e premere il bottone che lo faccia partire. Ma in realtà, non appena si lancia, il debugger termina la sua esecuzione.

3.1 Patching per la `IsDebuggerPresent`

Fra le funzioni chiamate nel WinMain, è possibile individuare la `FUN_004024a0`, al cui interno vi è la chiamata alla API `IsDebuggerPresent`:

- si valuta il risultato di ritorno di tale API e se è pari a 0, si termina l'applicazione
- se invece il valore è diverso da 0, si chiama l'API `ShowWindow` e si ritorna al WinMain

Siccome l'applicazione viene eseguita all'interno del debugger, il valore di ritorno sarà sempre 0, quindi per risolvere tale problema in OllyDbg è stata applicata una patch modificando alcune delle istruzioni del codice Assembly, come viene mostrato in 3.
A questo punto, è stato possibile analizzare tutta la struttura del programma mediante l'utilizzo di OllyDbg: se si fa "run" tramite debugger, anche in questo caso l'esecuzione rimane running ma senza mostrare la finestra mentre ci si aspetterebbe di vederla comparire.

3.2 Accesso alla PEB

L'analisi si è spostata sulla WinProc, per verificare se ci fosse qualche altro meccanismo anti-debugging. La scelta è stata quella di analizzare la WinProc in quanto prima di mostrare la finestra, verranno sicuramente gestiti alcuni messaggi dalla procedura, per creare la finestra ed inserivi gli elementi all'interno.
È stato messo un breakpoint sulla prima istruzione della WinProc, per poi procedere con l'analisi step-by-step dopo aver rilanciato il programma. La prima funzione che si trova all'interno del codice è la `FUN_00401dc0` a cui viene passato come parametro il codice del messaggio ricevuto. Analizzando la funzione, sia con Ghidra che con OllyDbg, è possibile capire che

- viene acceduto il segmento FS ad offset 0x30. Tale offset contiene la struttura dati PEB
- all'offset 0x2 della PEB c'è un campo `BeingDebugged`, pari a 0 se è presente un debugger

Il risultato della funzione è quindi quello di incrementare il valore del codice del messaggio di uno: l'effetto di tale funzione è quindi di impedire a ciascun messaggio per cui vi è una gestione esplicita, come la creazione della finestra o il paint della stessa, di venire correttamente gestito, in quanto il valore verrebbe sempre incrementato di 1, andando a finire sulla gestione di default.
Quindi è stata applicata una patch in OllyDbg che andasse a mascherare tale chiamata a funzione mediante

una serie di NOP, come mostrato in 4.
A questo punto, se si apre il nuovo file patchato e si prova a far partire il programma, viene mostrata la finestra, ma senza contenuto e dopo qualche secondo il debugger crasha. Se si prova ad aprire il programma normalmente, quindi fuori da un debugger, viene mostrato il message box riassunta in 5.
In ogni caso, una volta risolto questo problema, sono stati messi dei breakpoint su ognuna delle prime istruzioni dei blocchi di codice per la gestione dei messaggi nella WinProc, in modo da poter analizzare singolarmente la gestione di ogni messaggio.

3.3 Path per la `OutputDebugString`

Precedentemente, era stato caricato in una variabile globale l'indirizzo della API `OutputDebugString`, quindi ci si aspetta che tale funzione venga richiamata in qualche altro punto del codice.
Tornando sempre ai breakpoints nella `WinProc`, è stato analizzato il blocco di codice per la gestione del comando `WM_PIANT` (codice 5): dopo aver chiamato le diverse API per ridisegnare la finestra, vi è una CALL alla funzione `FUN_00404000`:

- dentro OllyDbg, è stato messo un breakpoint alla prima istruzione di tale funzione
- una volta che l'esecuzione arriva al breakpoint, proseguendo con "step over", si arriva al punto in cui viene caricato sullo stack la stringa "%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s"
- viene poi preparata la CALL alla funzione `OuputDebugString`

La `OuputDebugString` è una funzione che permette di inviare una stringa al debugger, utilizzando una formattazione "printf-like", quindi la stringa di formato costituita dai 16 "%s" avrà l'effetto di dire al debugger di dover andare a leggere 16 parametri dallo stack, questo porterà ad accedere ad un indirizzo di memoria invalido, mandando quindi in crash il debugger stesso, in quanto verrà generato un segmentation fault.
Per risolvere è stata applicata una patch, andando a mettere una serie di NOP al posto della CALL alla funzione `FUN_00404000`.

3.4 Risoluzione del Message Box "Internal Error"

3.4.1 Manipolazione del file mapping

All'interno del blocco di codice che gestisce il messaggio `WM_CREATE`, dopo aver creato tutti gli handles alle finestre, aver inizializzato il timeout ed aver registrato tale timeout chiamando la `SetTimer`, vi è un'ultima CALL alla funzione `FUN_004016b0`. In tale funzione, si accedono le due variabili globali inizializzate precedentemente nella `init_ds` della `WinMain`, ovvero `file_map_startaddr` e `low_file_size`. Arrivando con un breakpoint in OllyDbg alla prima istruzione di tale funzione, è stato possibile analizzarne il contenuto:

- La funzione accede al campo ad offset 192 della struttura dati, impostandolo a 0
- il registro EAX contiene la taglia del file, che viene divisa per 4 mediante uno shift a desta e viene sottratta di 256
- EDX contiene l'indirizzo iniziale del file mapping, a cui viene aggiunto il valore 400
- all'interno di un ciclo for, viene messo in XOR l'ultimo campo della struttura dati con 4 byte alla volta della file mapping, quindi con un indirizzo alla volta di quelli che costituiscono tale mapping
- al termine del ciclo, il valore accumulato viene scritto nell'ultimo campo della struttura dati.

Quello che avviene in questa funzione è quindi il calcolo di un checksum per il file, per andare probabilmente a verificare più avanti se sono avvenuti cambiamenti al file stesso.

3.4.2 Analisi della `TimerProc`

La funzione `FUN_00401b30`, chiamata nel WinMain, chiama al suo interno la `SetTimer` di cui l'ultimo parametro è la procedura di callback `TimerProc`, chiamata per la gestione del timeout. All'interno di tale funzione, così come avveniva per il file **hw2.exe**, vi è una chiamata a funzione al campo della struttura dati ad offset 20, che è un dato globale nel segmento bss.
Tale dato va quindi convertito in una funzione, e disassemblando le istruzioni che seguono, vi sono diversi meccanismi anti-disassembling che rendono quindi complicato seguire il flusso del codice anche avvalendosi del de-compilatore. Per questo motivo, si ci è avvalsi di OllyDbg per poter seguire con più facilità: è stato posto un breakpoint sulla `TimerProc`:

- eseguendo con "step over", si arriva alla CALL della `FUN_004042a0`
- entrando in tale funzione con "step into", è stato analizzato il flusso di esecuzione della funzione
- nella funzione, si accede alla struttura di dati, ad offset 192, quindi si recupera il checksum che era stato precedentemente calcolato nella `FUN_004016b0`
- viene confrontato tale valore con il valore della variabile globale `DAT_004050a4`, il cui contenuto è 74EE8F1Fh e tale confronto non risulta essere uguale. Quindi, il programma si rende conto che sono state effettuate delle patch alle istruzioni, che rendono il checksum calcolato staticamente non valido e portano l'esecuzione a terminare
- vengono caricate le stringhe che verranno poi mostrate nel message box riportante un internal error, come mostrato in 6.

Per poter risolvere il problema, è stata nuovamente applicata una patch, andando ad evitare che venisse effettuata la CALL alla funzione, come mostrato in 7.
Ora, eseguendo il programma in OllyDbg, viene mostrata la finestra che aspetta l'input utente, come mostrato in 8

4 Ricerca del codice di sblocco

Come già detto, cercare di capire la logica per controllare la validità del codice inserito dall'utente ed eventualmente spegnere la macchina è complicato avvalendosi di Ghidra.
Si è quindi proceduto come segue:

- è stato messo un breakpoint sull'istruzione ad indirizzo 4040e0
- è stato fatto ripartire il programma, che si ferma in attesa che venga inserito l'intput dall'utente, ovvero il valore del timeout e che venga premuto il bottone "Go"
- è stata inserita una sequenza casuale di caratteri per il codice di sblocco ed impostato il timeout a 0:00:00:00, così da far scattare immediatamente il breakpoint

A questo punto, è stato possibile analizzare tutto il flusso d'esecuzione mediante "step over":

- vi sono le CALL alle diverse API usate per preparare l'ambiente per spegnere la macchina:
  - `GetCurrentProcess`
  - `LookupPrivilegeValueA`, in cui viene richiesto il privilegio "SeShutdownPrivilege"
  - `LookupPrivilegeValueA`
  - `AdjustTokenPrivileges`
- la chiamata alla API `GetDlgItemTextA`, che recupera quindi il codice passato dall'utente per poi confrontarlo con quella che sarà la reale stringa di sblocco all'atto della decisione se spegnere la macchina o meno
- vi è infine una CALL alla funzione `FUN_004050C0`, ed in seguito a tale funzione l'applicazione termina con una CALL alla `FUN_00401D80` la quale mostra la Message Box che informa che il codice di sblocco è errato e che quindi lo spegnimento della macchina non è avvenuto.

Quindi, l'ultima funzione chiamata è la `FUN_004050C0`, per cui è stato messo un breakpoint sulla CALL a tale procedura, ad indirizzo 40423e. Una volta che il flusso di esecuzione è arrivato alla CALL, si è proceduti mediante "step into" ad analizzare il contenuto della funzione:

- consultando OllyDbg, in particolare vedendo il contenuto dei registri, si nota che viene messo in EDX il contenuto della stringa passata in input dall'utente come codice di sblocco
- successivamente si procede a prendere una serie di salti, fino ad arrivare ad una CMP del contenuto di [ESP +24] col valore 9. Se tale confronto fallisce, il programma esegue la ret e termina. Questo porta quindi a pensare che la stringa di sblocco debba avere una lunghezza di 9 caratteri
- infatti, se tale CMP viene rispettata, si salta in una serie di istruzioni che precedono come segue:
  - si carica con una MOVZX il contenuto del registro ESP, spiazzandosi sempre di 1 byte alla volta, dentro il registro EAX

– si effettua lo XOR degli 8 bit meno significativi di EAX (registro AL) con il byte del registro EDX, spiazzandosi in EDX ogni volta di un byte alla volta
– si valuta il risultato di tale XOR e se non è pari al risultato atteso si esce dalla funzione

Quindi, in questa serie di istruzioni viene verificato se la stringa immessa dall'utente contiene il reale codice di sblocco, dove per ogni byte deve valere la seguente relazione:

$$AL_i \oplus X_i = byte_i \; \forall i = 1, .., 9 \tag{1}$$

dove

- AL_i è il byte caricato dal segmento ESP
- X_i è il byte che compone la stringa
- $byte_i$ è il valore atteso.

La tabella 1 contiene tutti i confronti effettuati per i 9 caratteri che compongono la stringa di sblocco. I caratteri sono stati ricavati considerando il valore che, effettuato lo XOR bit a bit col byte letto da AL, permetteva di ottenere il risultato atteso

Carattere letto da AL	**Risultato atteso dallo XOR**	**Carattere del codice**
3f	0c	33
28	5a	72
2f	4e	61
a5	c0	65
5d	2e	73
47	13	54
3d	0d	30
4f	70	3f
3f	1e	21

Tabella 1: Tabella che riassume i valori esadecimali fra cui vengono fatti gli XOR per determinare la correttezza del codice di sblocco

Quindi, il codice che rende funzionale il programma è il seguente, ottenuto dopo aver convertito i byte dell'ultima colonna della tabella dal formato esadecimale in ASCII:

3rNesT0?!

Immagini

```
void FUN_004016f0(void)

{
  HMODULE hModule;
  char local_str [19];

  local_str[0] = 'k';
  local_str[1] = 'e';
  local_str[2] = 'r';
  local_str[3] = 'n';
  local_str[5] = 'l';
  local_str[6] = '3';
  local_str[7] = '2';
  local_str[8] = '.';
  local_str[9] = 'd';
  local_str[11] = 'l';
  local_str[10] = 'l';
  local_str[12] = '\0';
  hModule = LoadLibraryA(local_str);
  local_str[0] = 'O';
  local_str[9] = 'u';
  local_str[4] = 'u';
  local_str[1] = 'u';
  local_str[12] = 't';
  local_str[5] = 't';
  local_str[2] = 't';
  local_str[3] = 'p';
  local_str[6] = 'D';
  local_str[7] = 'e';
  local_str[8] = 'b';
  local_str[16] = 'g';
  local_str[10] = 'g';
  local_str[11] = 'S';
  local_str[13] = 'r';
  local_str[14] = 'i';
  local_str[15] = 'n';
  local_str[17] = 'A';
  local_str[18] = '\0';
  GetProcAddress(hModule,local_str);
  return;
}
```

Figura 1: Risultato della de-compilazione dopo aver risolto il disassembling della `FUN_004016f0`

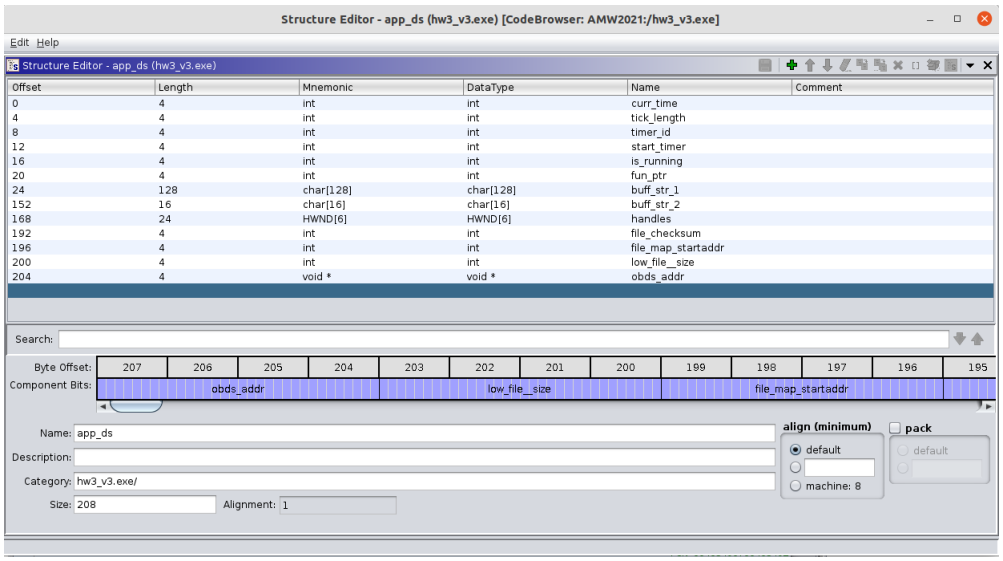

Figura 2: Struttura dati per il file hw3.exe

Figura 3: Patch della funzione `IsDebuggerPresent`

Figura 4: Patch del campo `BeingDebugged` acceduto dalla PEB

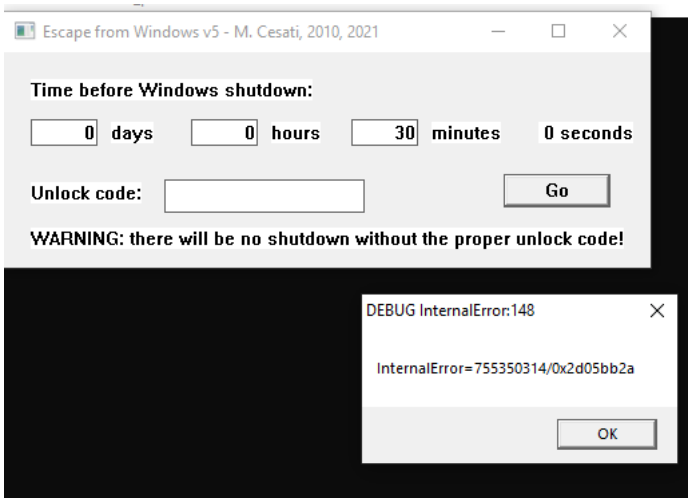

Figura 5: Message Box dell' "Internal Error"

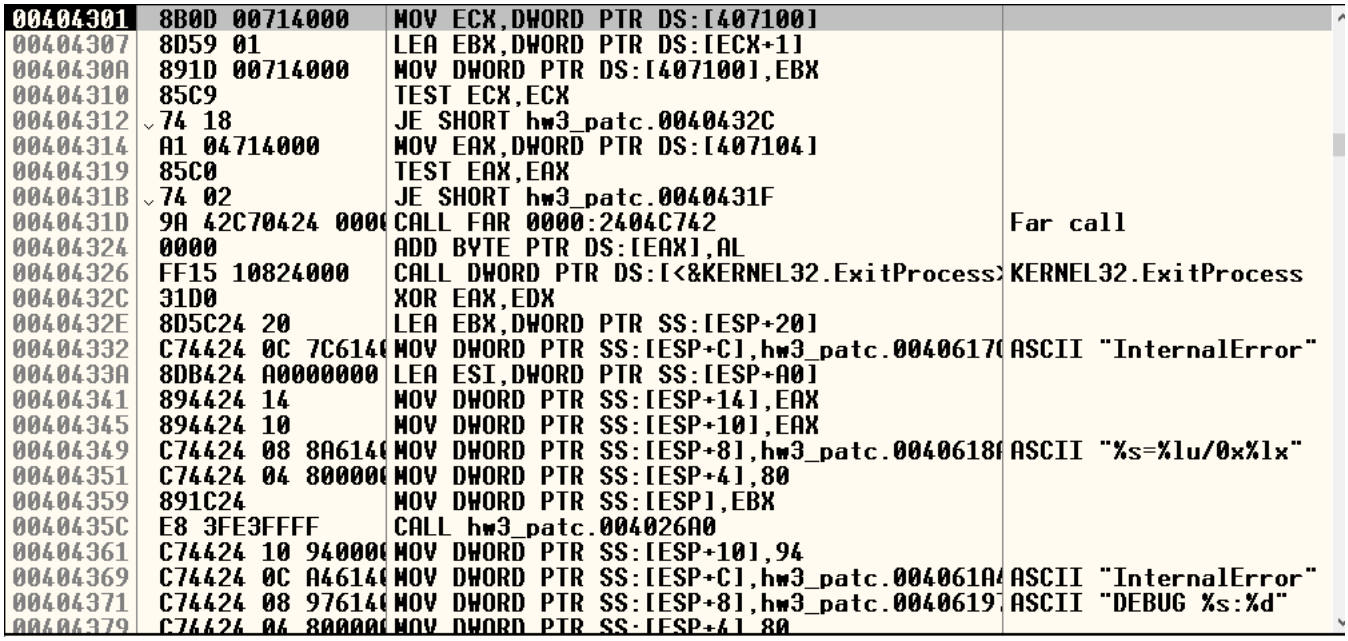

Figura 6: Stringhe relative al messaggio di "Internal Error"

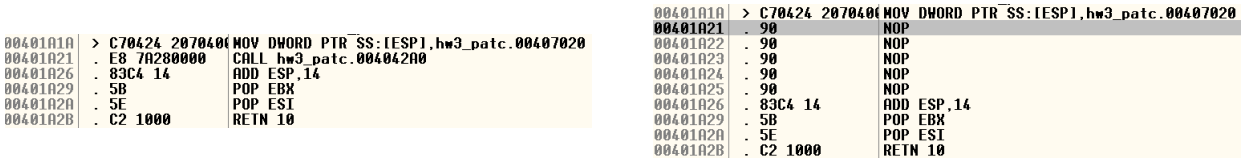

Figura 7: Patch per la Message Box "Internal Error"

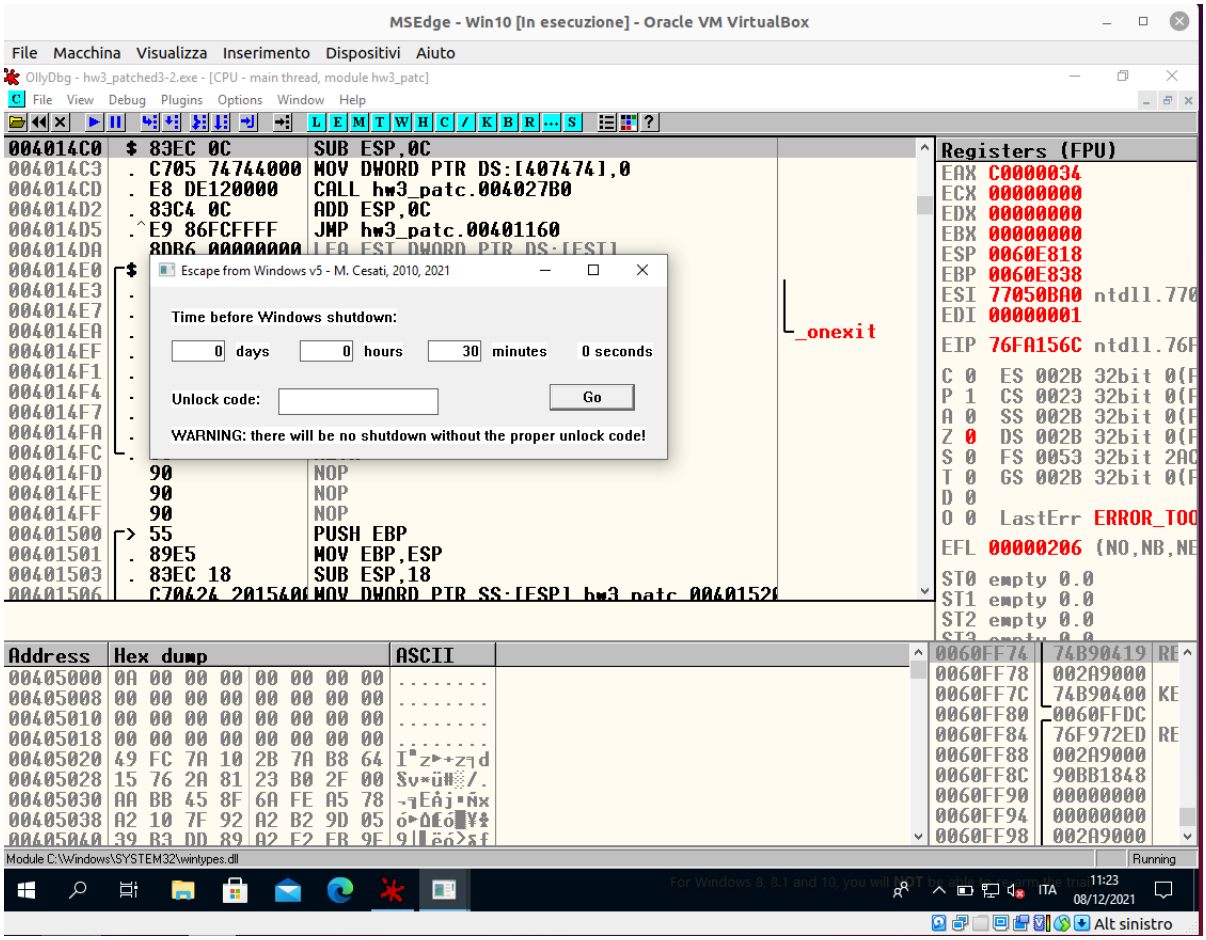

Figura 8: Finestra correttamente mostrata nel debugger