

# Documentazione Ghidra W03

## Funzione applicazione

Demo che deve spegnere il computer dopo un tempo impostato dall'utente. In particolare l'applicazione essendo una demo si chiude dopo 3 minuti senza spegnere PC.

### Obiettivo

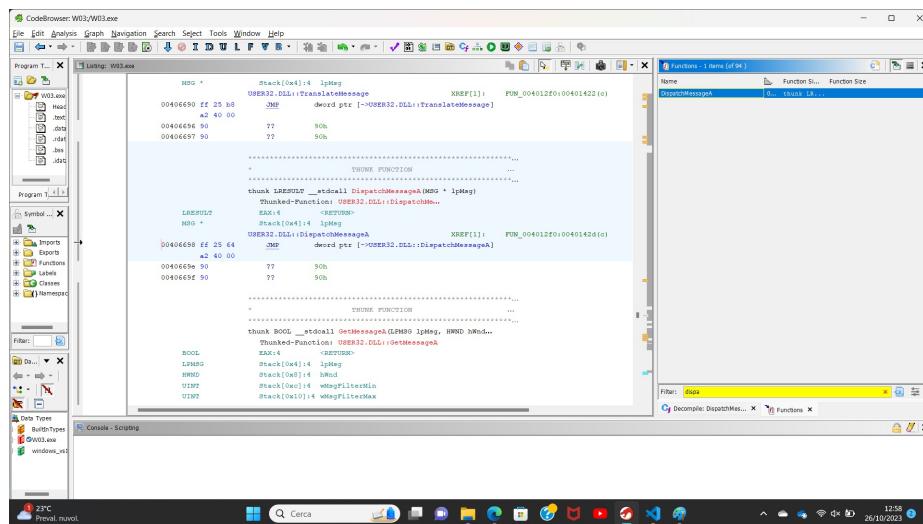
Si vuole trovare il punto del codice della demo che prevede la chiusura dell'applicazione e toglierlo per evitare che si chiuda.

### Funzionamento atteso

Mi aspetto di trovare un MsgLoop dopo la creazione della finestra (funzionamento solito app Windows).

## Trovare il main

Per trovare il main provo innanzitutto a cercare il **Message Loop**, per fare questo cerco la funzione **DispatchMessage** tra le funzioni dell'applicazione (**Window -> Functions**).

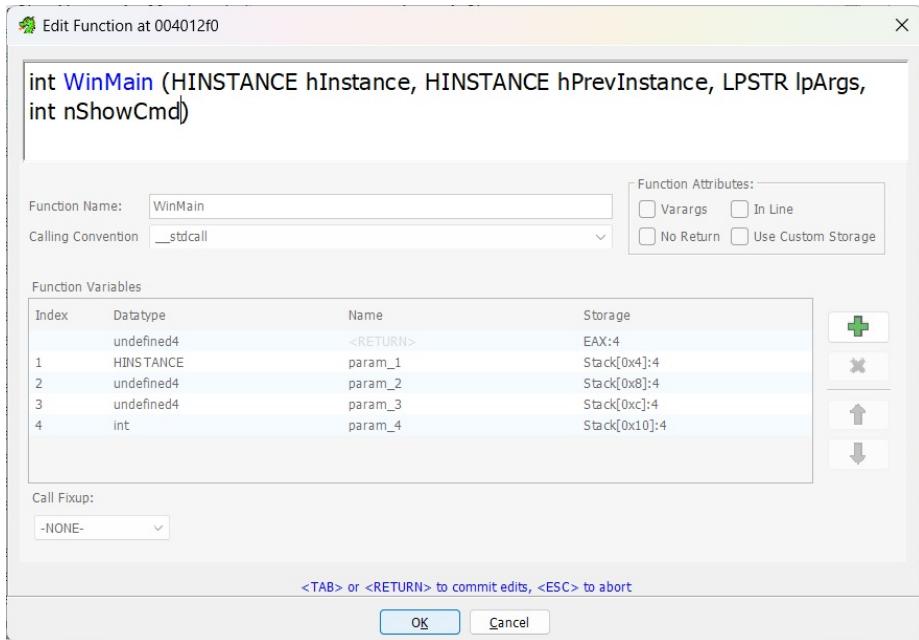


Questa mi porterà a un punto del codice macchina dove è presente il riferimento alla riga di questa funzione.

A questo punto doppio clic sul riferimento per arrivare al punto del codice in cui è presente il loop. Posso anche premere: **tasto destro->references->show references to ...** usando il riferimento rispetto ad una funzione (non indirizzo).

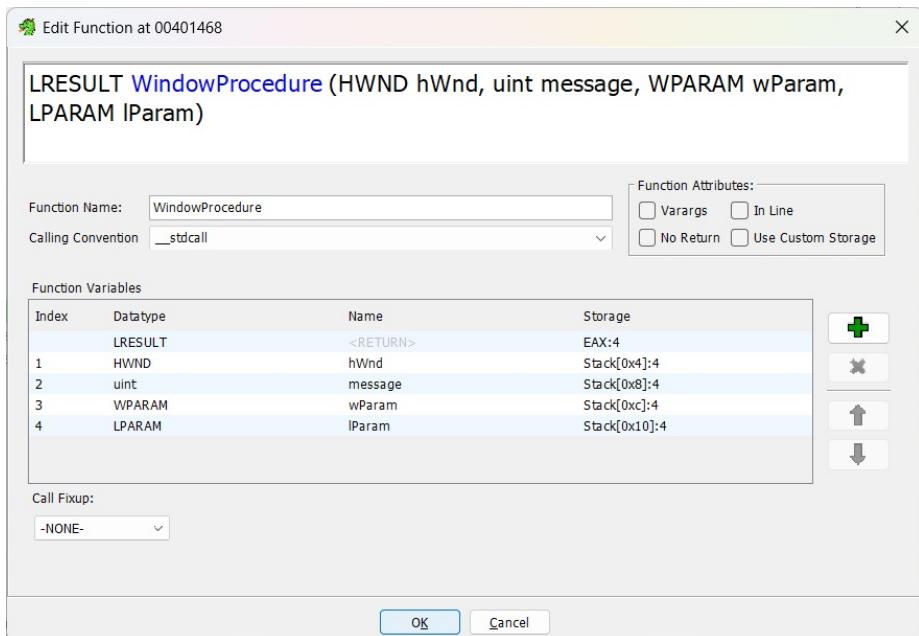
La funzione che contiene questo loop la rinominiamo **WinMain** (basta cliccare tasto **L** per rinominare con il cursore sul nome della funzione).

A questo punto cliccare con tasto destro sulla funzione e selezionare **Edit Function** (o premere **F**) per poi impostare i parametri della funzione **WinMain** (come si vede nella documentazione su msdn).



Decompilando il **WinMain**, mi accorgo della struttura **WNDCLASSEX** che contiene tra le varie cose il puntatore a **WNDPROC** ossia la procedura che gestisce i messaggi della finestra.

Di conseguenza su Ghidra la vado a rinominare l'etichetta **LAB\_0040146** come **WindowProcedure** e premo **F** per dire a Ghidra che da lì inizia una funzione. A questo punto (**rif 00401468**) impostiamo anche il prototipo (di nuovo **F**, come fatto in precedenza con WinMain).

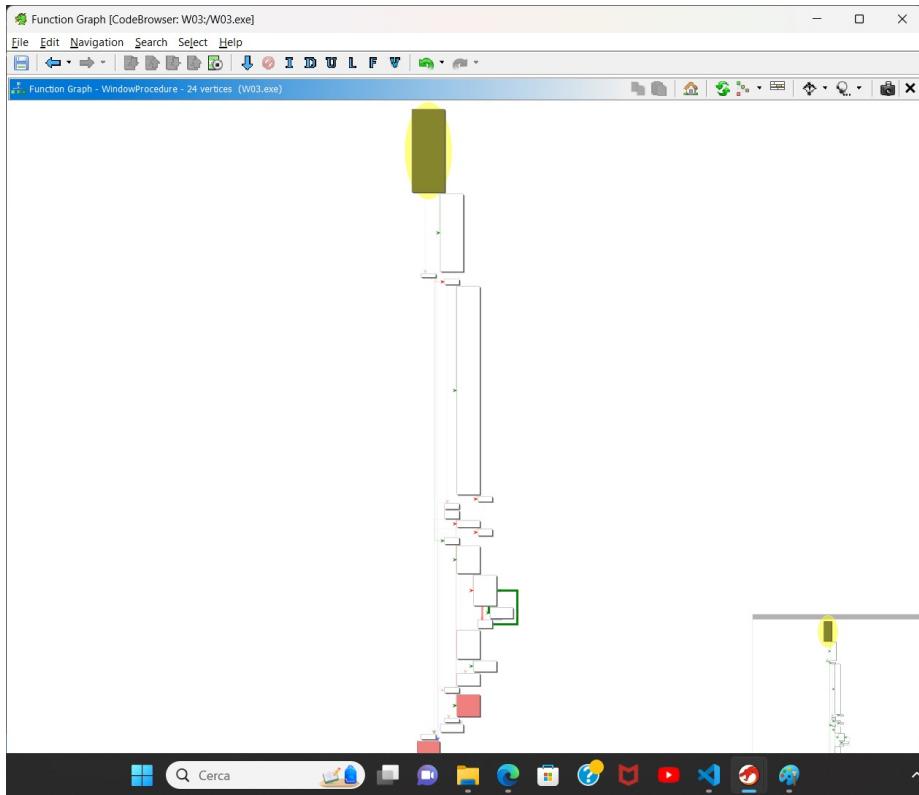


- Primo parametro è l'handle della finestra che riceve il messaggio
- secondo parametro: codice/tipo di messaggio
- WPARAM
- LPARAM

Questi ultimi due parametri variano di messaggio in messaggio.

A questo punto ci ricordiamo del nostro **obiettivo** (rimuovere il codice per la chiusura della demo).

Proviamo a capire dal grafo delle funzioni cosa sta succedendo (**Window -> Function Graph**).



Poichè sto cercando di andare a capire quali sono i messaggi gestiti quindi cerco degli **if** o uno **switch**.

Mi accorgo nel blocco della WindowProcedure di un **compare**, a riga ...1493, tra il valore di **EBX** e 5. Questo sembra promettente poiché il valore di EBX viene preso dal messaggio (come si vede dal codice macchina con indirizzo ...1474 nella seguente figura).

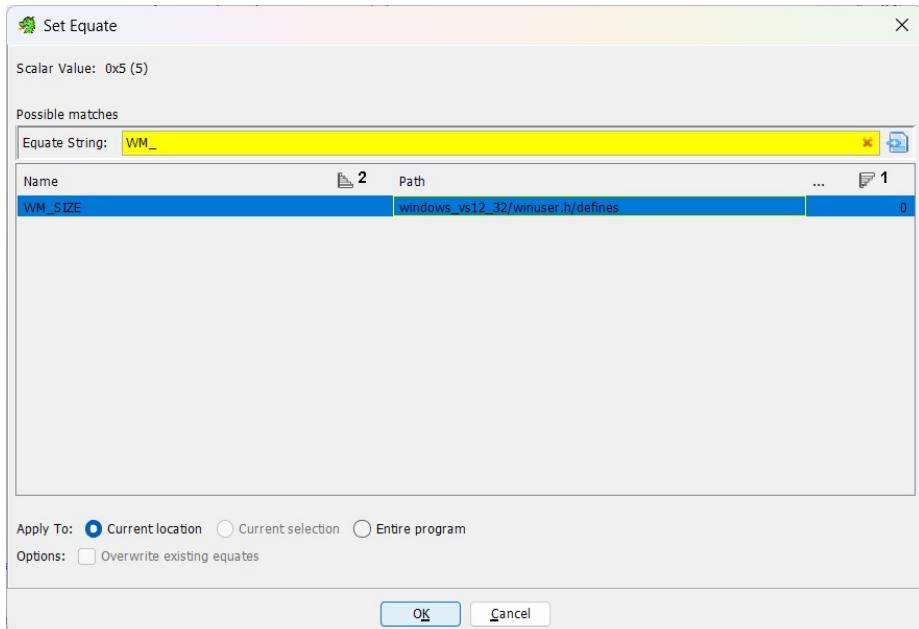
```

Function Graph [CodeBrowser: W03/W03.exe]
File Edit Navigation Search Select Help
Function Graph - WindowProcedure - 24 vertices (W03.exe)
...1468 PUSH EBP
...1469 MOV EBP,ESP
...146b PUSH EDI
...146c PUSH ESI
...146d PUSH EBX
...146e SUB ESP,0xfc
...1474 MOV EBX,dword ptr [EBP + mess...
...1477 MOV ESI,dword ptr [EBP + wParam...
...147a MOV EDI,dword ptr [EBP + lParam...
...147d MOV dword ptr [ESP + local_108...
...1485 MOV EAX,dword ptr [EBP + hWnd]
...1488 MOV dword ptr [ESP]>>local_10c...
...148b CALL USER32.DLL::GetWindowLongA
...1490 SUB ESP,0x8
...1493 CMP EBX,0x5
...1496 MOV dword ptr [EBP + local_b8...
...149c JZ LAB_00401507

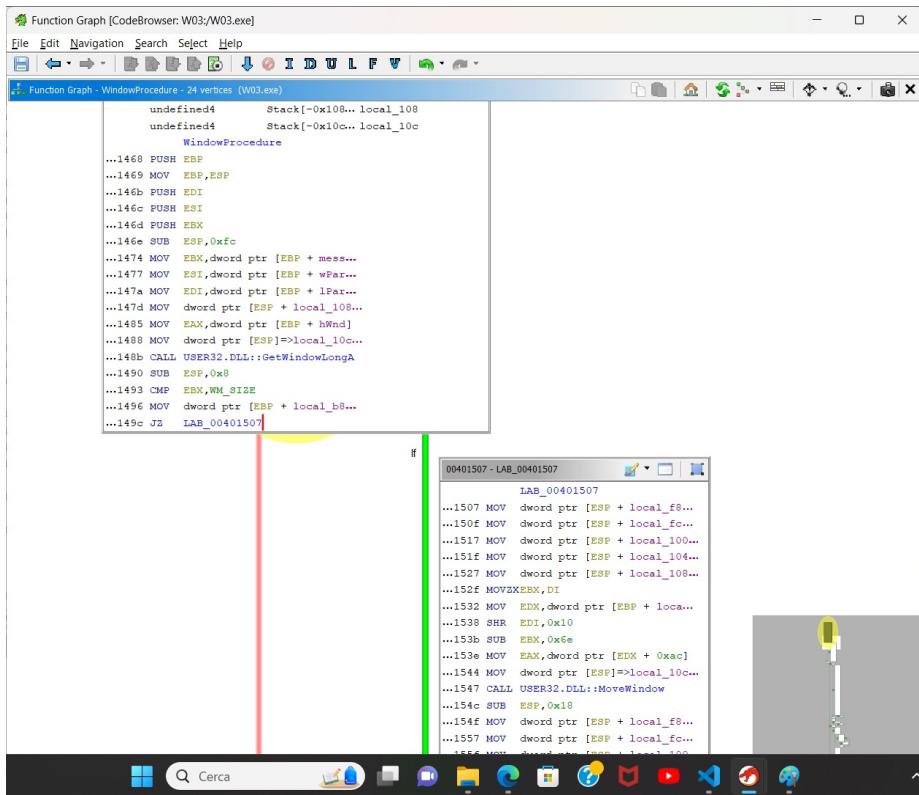
```

**LAB\_00401507**  
...1507 MOV dword ptr [ESP + local\_f8...
...150f MOV dword ptr [ESP + local\_fc...
...1517 MOV dword ptr [ESP + local\_100...
...151f MOV dword ptr [ESP + local\_108...
...1527 MOV dword ptr [ESP + local\_10...
...152f MOVZX EBX,DI
...1532 MOV EDX,dword ptr [EBP + loca...
...1538 SHR EDI,0x10
...153b SUB EBX,0x6
...153e MOV EAX,dword ptr [EDX + 0xac]
...1544 MOV dword ptr [ESP]>>local\_10c...
...1547 CALL USER32.DLL::MoveWindow
...154c SUB ESP,0x10
...154f MOV dword ptr [ESP + local\_f8...
...1557 MOV dword ptr [ESP + local\_fc...
...155f MOV dword ptr [ESP + local\_100...
...1567 MOV dword ptr [ESP + local\_104...

Per capire di quale messaggio si tratta cerco la macro che inizia con **WM\_** tra quelle che ottengo cliccando il tasto **E** oppure tramite tasto destro -> **Set Equate**.

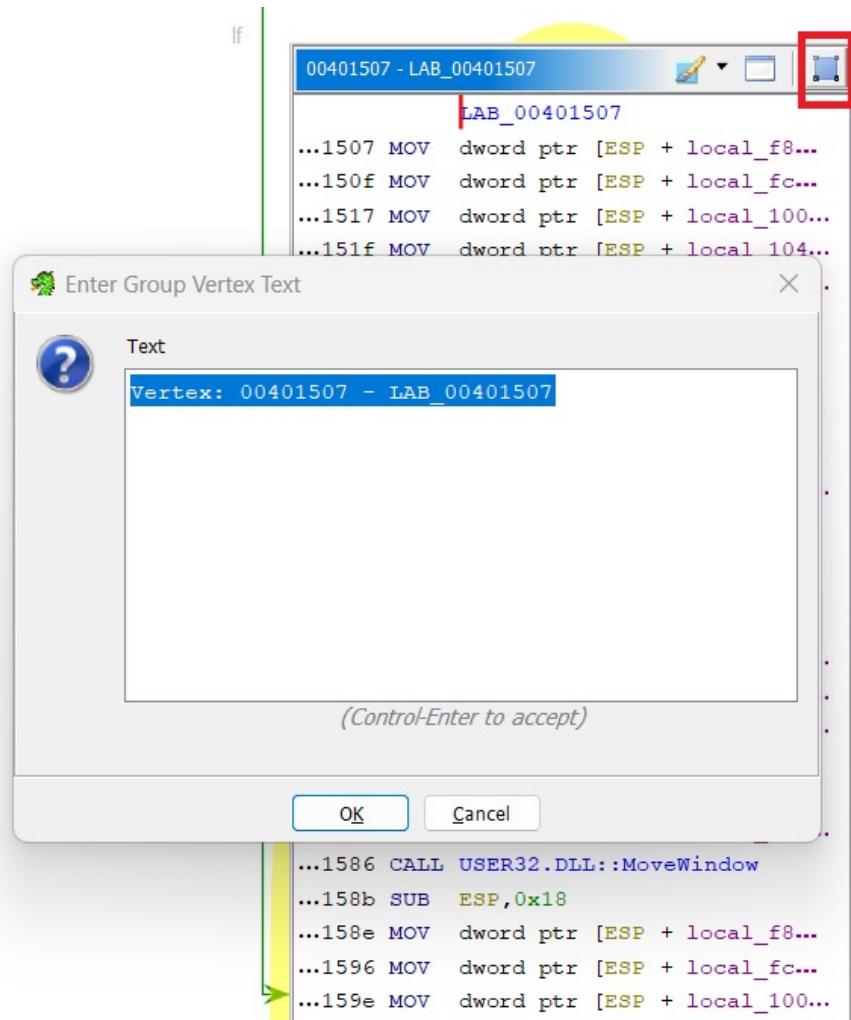


Mi accorgo quindi che tornando nel grafo ho due frecce che escono dal blocco, una verde se il salto viene preso (codice messaggio=5) e freccia rossa se il salto non viene preso.



In particolare il valore 5 come codice di messaggio indica **WM\_SIZE**, ossia indica che il messaggio serve per ridimensionare la finestra. Non è utile nel nostro caso.

Poiché non ci interessa lo collassiamo premendo nel quadratino in alto a destra nel blocco (evidenziato con un rettangolo rosso in figura) e scrivendo nella finestra di dialogo che esce fuori il nome che vogliamo dare al blocco collassato.



Procendo sui rami if/else si incontra prima un salto con un minore uguale (serve a differenziare meglio i casi), continuando a scendere incontriamo un **CMP** con 1 che indica il messaggio **WM\_CREATE** (cliccare sempre sulla costante tasto **E** come visto in precedenza per sostituire la costante con la macro) [[rif 004014d9](#)]

A questo punto bisogna prendere sempre la documentazione (msdn) e vedere cosa viene passato quando viene generato quel messaggio.

In particolare vediamo che questo viene generato quando Windows disegna per la prima volta la finestra (Per ora sembra non interessarci, quindi lo colllassiamo insieme a tutti i consequenti; per collassarli tutti insieme tenere premuto **ctrl** mentre si selezionano premendo poi sul quadratino in alto a destra di uno dei blocchi selezionati).

Tornando a lavorare con i **CMP** troviamo il messaggio con **0xf** [[rif 004014a0 func graph](#)] che ci indica in particolare una **WM\_PAINT** (inserire sempre la macro). Indica che Windows deve ridisegnare la finestra quando questa passa da nascosta a parzialmente/totalmente visibile.

Proseguendo incontriamo **WM\_COMMAND**, [[rif 004014a9 func graph](#)] dalla documentazione vediamo che questo messaggio viene mandato quando l'utente seleziona o una voce di un menu oppure un controllo manda una notifica alla sua finestra genitore (tutti gli elementi dentro una finestra sono a loro volta finestre, le finestre sono gerarchiche).

Tra gli altri comandi che gestisce l'applicazione troviamo **WM\_DESTROY**, [[rif 004014e2](#)] si ha quando la finestra viene distrutta.

Tutto ciò che è rimasto è il default. Quello che chiama `DefWindowProc`.

Una volta collassati tutti i blocchi dello switch bisogna chiedersi dove potrebbe essere il punto in cui la funzione fa le cose specifiche che cerchiamo.

Si può pensare che quello che ci interessa si trovi in `WM_COMMAND`, quindi andiamo al codice che gestisce questo messaggio ("sembra più esotico" cit.).

Conviene cambiare nome all'etichetta locale a cui si salta nel caso in cui il messaggio sia `WM_COMMAND`, in particolare la chiamiamo `handle_command_msg` (procedimento L per cambiare nome all'etichetta).

```
004014a9 81 fb 11      CMP      EBX, WM_COMMAND
                      01 00 00
004014af 0f 84 7f      JZ       LAB_00401634
                      01 00 00
```

(Sono due parti di codice separate)

```
handle_command_msg
00401634 c1 ee 10      SHR      ESI, 0x10
00401637 85 f6      TEST     ESI, ESI
00401639 0f 85 bc      JNZ      LAB_004014fb
                      fe ff ff
0040163f 8b 95 4c      MOV      EDX, dword ptr [EBP + local_b8]
                      ff ff ff
00401645 39 ba b8      CMP      dword ptr [EDX + 0xb8], EDI
                      00 00 00
0040164b 0f 85 aa      JNZ      LAB_004014fb
                      fe ff ff
00401651 e8 9b 06      CALL     FUN_00401cf1
                      00 00
00401656 31 c0      XOR      EAX, EAX
00401658 e9 a0 fe      JMP      LAB_004014fd
                      ff ff
```

Questo parte lavorando su registri già caricati, in particolare la prima istruzione lavora sul registro `ESI`. Devo allora capire chi ha impostato per ultimo questo registro. In particolare salendo nel codice vediamo che il registro `ESI` è stato impostato al valore di `wParam`.

```
file:///h
WindowProcedure
00401468 55      PUSH    EBP
00401469 89 e5      MOV     EBP, ESP
0040146b 57      PUSH    EDI
0040146c 56      PUSH    ESI
0040146d 53      PUSH    EBX
0040146e 81 ec fc      SUB     ESP, 0xfc
                      00 00 00
00401474 8b 5d 0c      MOV     EBX, dword ptr [EBP + message]
00401477 8b 75 10      MOV     ESI, dword ptr [EBP + wParam]
0040147a 8b 7d 14      MOV     EDI, dword ptr [EBP + lParam]
0040147d c7 44 24      MOV     dword ptr [ESP + local_108], 0xfffffffffeb
```

A questo punto si va a vedere il significato di questa variabile. Vedendo sulla documentazione ci accorgiamo che `wParam` ha un significato diverso a seconda di chi ha generato questo comando (menu, acceleratore o controllo). Questi casi possono essere distinti utilizzando il valore di `wParam` (in particolare i bit più significativi, per questo si ha uno shift right):

- 0: utente seleziona una voce di menu

- 1: acceleratore di tastiera
  - control-defined notification code

In quest'ultimo caso `lParam` è l'handle alla finestra di controllo cioè quella che ha generato la notifica che ha portato a questa finestra il messaggio.

Message Source	wParam (high word)	wParam (low word)	lParam
Menu	0	Menu identifier (IDM_*)	0
Accelerator	1	Accelerator identifier (IDM_*)	0
Control	Control-defined notification code	Control identifier	Handle to the control window

In particolare nel codice macchina ci si chiede se la parte più significativa di `ESI` (primi 16 bit) è 0 o meno, in particolare se non è 0 salta all'istruzione `004014fb` che è un'istruzione di uscita perché azzera `EAX` [questo lo vediamo tornando a `handle_command_msg`] e va a fare la return (e che quindi rinominiamo `existing_with_0`). Troveremo questo richiamo due volte in `handle_command_msg`

**LAB\_004014fb** XREF[2]: 00401639(j), 0040164b(j)  
004014fb 31 c0 XOR EAX,EAX

Altrimenti si continua e si va a fare un test su **EDI** (che rappresentava **lParam**, l'handle) confrontandolo con il contenuto di **EDX** più un offset fisso, dove **EDX** è letto dalla variabile locale **local\_b8**. Se non sono uguali torna ad uscire altrimenti viene invocata un'altra funzione (funzione per animare il bottone al clic la chiamo **animate\_btn**).

```
0040163f 8b 95 4c        MOV      EDX,dword ptr [EBP + local_b8]
                        ff ff ff
00401645 39 ba b8        CMP      dword ptr [EDX + 0xb8],EDI
                        00 00 00
0040164b 0f 85 aa        JNZ      exiting_with_0
                        fe ff ff
00401651 e8 9b 06        CALL     FUN_00401cf1
                        00 00
```

Facendo doppio clic su `local\_b8`, Ghidra ci porta nella zona dello stack da cui riusciamo a capire dove questo viene impostato.

Quando parte la funzione, Ghidra riassume tutto lo stack ma anche i riferimenti allo stack compreso. E' utile perchè ci permette di capire dove avviene la lettura (R) o la scrittura (W).

Facendo quindi doppio clic sul primo riferimento, Ghidra ci porta alla zona di codice macchina dove questo viene scritto.

undefined4	Stack[-0xb8]:4local_b8	XREF [9]:	00401496 (W), 00401532 (R), 00401577 (R), 004015b6 (R), 004015f0 (R), 0040163f (R), 00401813 (R), 00401879 (R), 004018a9 (R)
------------	------------------------	-----------	--

In particolare vediamo che su `local_b8` viene scritto il valore dell'accumulatore `EAX`, (questo lo vediamo facendo un doppio clic su `00401496`, cioè il primo riferimento verde, acceduto in scrittura). Ora dobbiamo vedere dove viene impostato quest'ultimo.

In particolare **l'accumulatore è utilizzato per restituire un valore da una funzione (es. le API di Windows)**.

```

0040147d c7 44 24      MOV      dword ptr [ESP + local_108],-21
                      04 eb ff
                      ff ff
00401485 8b 45 08      MOV      EAX,dword ptr [EBP + hWnd]
00401488 89 04 24      MOV      dword ptr [ESP]=>local_10c,EAX
0040148b e8 18 52      CALL    USER32.DLL::GetWindowLongA
                      00 00
00401490 83 ec 08      SUB     ESP,0x8
00401493 83 fb 05      CMP     EBX,WM_SIZE
00401496 89 85 4c      MOV     dword ptr [EBP + local_b8],EAX
                      ff ff ff

```

Salendo a ritroso in particolare incontriamo la API di Windows `GetWindowLong` (rif. `0040148b`).

Bisogna vedere nella documentazione cosa fa questa API.

Questa restituisce l'informazione legata ad una certa finestra, restituisce inoltre una parola a 32 bit (`DWORD`) ad uno specifico offset di una memoria extra allocata alla finestra.

In realtà ha diverse funzionalità perché a seconda della costante passata si possono avere diversi possibili valori (il primo parametro è l'handle alla finestra, il secondo è il cosa si vuole avere dalla finestra come si vede in figura).

Valore	Significato
<code>GWL_EXSTYLE</code> -20	Recupera gli stili di finestra estesi.
<code>GWL_HINSTANCE</code> -6	Recupera un handle nell'istanza dell'applicazione.
<code>GWL_HWNDPARENT</code> -8	Recupera un handle nella finestra padre, se presente.
<code>GWL_ID</code> -12	Recupera l'identificatore della finestra.
<code>GWL_STYLE</code> -16	Recupera gli stili della finestra.
<code>GWL_USERDATA</code> -21	Recupera i dati utente associati alla finestra. Questi dati sono destinati all'uso da parte dell'applicazione che ha creato la finestra. Il valore è inizialmente zero.
<code>GWL_WNDPROC</code> -4	Recupera l'indirizzo della routine della finestra o un handle che rappresenta l'indirizzo della routine della finestra. È necessario usare la funzione <code>CallWindowProc</code> per chiamare la routine della finestra.

In particolare si possono ricevere `user data`, ossia dati associati alla finestra che vengono decisi dal programmatore.

Questo è quello che succede in questa applicazione, infatti, andando a vedere i parametri sullo stack ci accorgiamo che come primo parametro viene passato l'handle alla finestra mentre come secondo parametro viene passato `-21` (rif. `0040147d`, il numero inizialmente si trova in esadecimale, per convertirlo cliccare su esso con il tasto destro -> Convert -> Signed Decimal).

A questo punto cliccare **E** mentre si ha il cursore sulla costante per sostituirla con la macro (si può fare anche quando è ancora in esadecimale), poiché non la conosce inserire il nome della macro trovata dalla documentazione (immagine precedente **GWL\_USERDATA**) nella finestra che si è aperta e cliccare **OK**.

```

0040147d c7 44 24      MOV      dword ptr [ESP + local_108],-21
                        04 eb ff
                        ff ff
00401485 8b 45 08      MOV      EAX,dword ptr [EBP + hWnd]
00401488 89 04 24      MOV      dword ptr [ESP]=>local_10c,EAX
0040148b e8 18 52      CALL    USER32.DLL::GetWindowLongA
                        00 00
00401490 83 ec 08      SUB      ESP,0x8
00401493 83 fb 05      CMP      EBX,WM_SIZE
00401496 89 85 4c      MOV      dword ptr [EBP + local_b8],EAX
                        ff ff ff

```

A questo punto cerchiamo nel codice macchina una **SetWindowLong** (se il valore viene preso da qualche parte è stato impostato, trovo nella documentazione di **GetWindowLong** il nome della funzione che serve per impostare quel dato).

Per trovarla da Ghidra cliccare **Window->Functions** e cercare se esiste effettivamente.



Una volta trovata cliccare su essa e fare doppio clic sul riferimento (evidenziato in giallo) che si trova vicino alla funzione che si apre come in figura.



In questo modo si trova la funzione in cui questa viene invocata ma soprattutto come viene impostato.

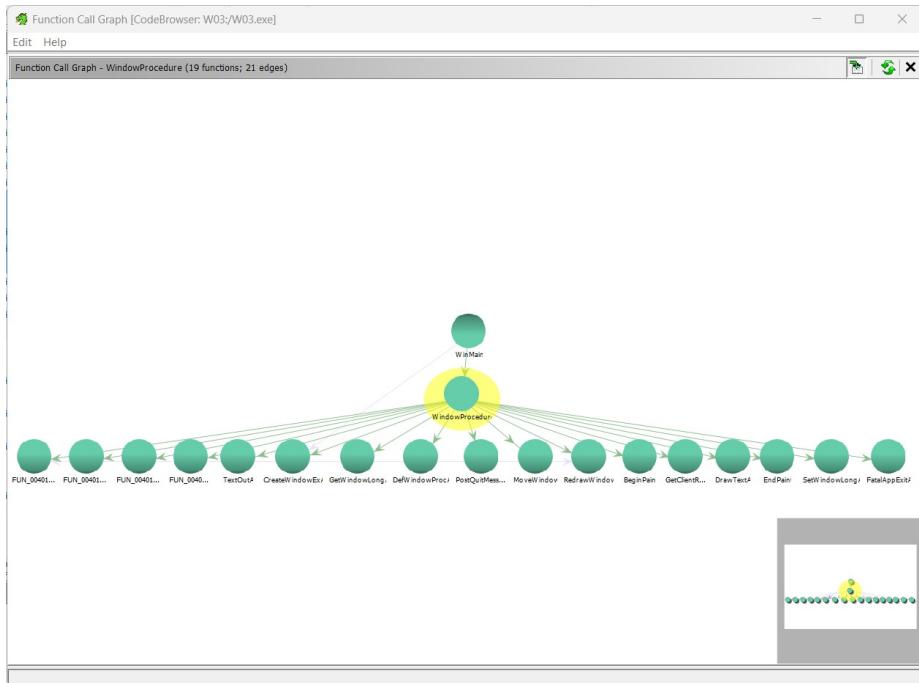
```

file:///h
004018f4 8b 45 08      MOV      EAX,dword ptr [EBP + hWnd]
004018f7 89 04 24      MOV      dword ptr [ESP]=>local_10c,EAX
004018fa e8 f1 4d      CALL    USER32.DLL::SetWindowLongA
                        00 00

```

Prima di tutto vediamo attraverso **Window->Function Call Graph** quale funzione invoca la **SetWindowLong**.

Ci si accorge che la funzione che la invoca è sempre la **WindowProcedure**.



Quindi la `WindowProcedure` ha fatto la `GetWindowLong` ma anche la `SetWindowLong`. Ora bisogna capire quando è il momento più sensato per farla, in particolare quando la finestra viene creata ossia gestendo l'evento `WM_CREATE` (lo si vede a partire dall'etichetta con cui si è arrivati ad eseguire questa parte di codice, ad esempio, decompilando nuovamente `WindowProcedure`, clicco su `WM_CREATE`).

file:///h

Andando a vedere da dove viene invocato (doppio clic sul riferimento) ci si accorge che viene invocato dalla `WM_CREATE` e quindi la label locale la si rinomina `handle_create_msq`.

	LAB_004014d9	XREF[1]:	0040149e (j)
004014d9	83 fb 01	CMP	EBX,WM_CREATE
004014dc	0f 84 e3	JZ	handle_create_msg
	03 00 00		

Si può anche vedere che `SetWindowLong` come secondo parametro prende sempre `-21` ossia `GWL_USERDATA`.

Il terzo parametro che viene passato è il valore da settare. (Possiamo rinominare i parametri passati `fn_argx`, con x numero del parametro). Identifico i parametri dalle `MOV dword ptr [ESP + x]`, situate sopra `SetWindowLongA`, riferito a riga [004018fa](#).

```

004018e6 8b 07      MOV     EAX,dword ptr [EDI]
004018e8 89 44 24 08 MOV     dword ptr [ESP + local_104],EAX
004018ec c7 44 24    MOV     dword ptr [ESP + local_108],GWL_USERDATA
04 eb ff
ff ff
004018f4 8b 45 08    MOV     EAX,dword ptr [EBP + hWnd]
004018f7 89 04 24    MOV     dword ptr [ESP]=>local_10c,EAX
004018fa e8 f1 4d    CALL   USER32.DLL::SetWindowLongA
00 00

```

Si è interessati a questo punto a capire da dove viene preso questo terzo valore, in questo caso viene preso dall'accumulatore **EAX**, che a sua volta lo prende da **EDI**.

Per vedere a quale valore era impostato **EDI** torniamo all'inizio, in particolare era stato impostato a **lParam** e non è stato più modificato (va verificata questa ultima cosa, dal codice si vede che non è più stato modificato).

**Attenzione:** Non basta mai prendere il codice, scorrere all'indietro ed arrivare all'ultima modifica di **EDI**, poichè potrei trovare una istruzione facente parte di costrutti IF o casi particolari. Come mi aiuto? Con il *function graph*.

```

0040147a 8b 7d 14      MOV     EDI,dword ptr [EBP + lParam]
0040147d c7 44 24      MOV     dword ptr [ESP + fn_arg2],GWL_USERDATA
04 eb ff
ff ff
00401485 8b 45 08      MOV     EAX,dword ptr [EBP + hWnd]
00401488 89 04 24      MOV     dword ptr [ESP]=>local_10c,EAX
0040148b e8 18 52      CALL   USER32.DLL::GetWindowLongA
00 00
00401490 83 ec 08      SUB    ESP,0x8
00401493 83 fb 05      CMP    EBX,WM_SIZE
00401496 89 85 4c      MOV    dword ptr [EBP + local_b8],EAX
ff ff ff
0040149c 74 69        JZ    LAB_00401507
0040149e 76 39        JBE   LAB_004014d9
004014a0 83 fb 0f      CMP    EBX,WM_PAINT
004014a3 0f 84 b4      JZ    LAB_0040165d
01 00 00
004014a9 81 fb 11      CMP    EBX,WM_COMMAND
01 00 00
004014af 0f 84 7f      JZ    handle_command_msg
01 00 00

```

Quindi il terzo parametro è **lParam**, per capire cosa è **lParam** per il messaggio **WM\_CREATE** bisogna vedere la documentazione.

In particolare è un puntatore ad una struttura **CREATESTRUCT** (vedi immagine) che contiene informazioni sulla finestra che sta venendo creata.

```

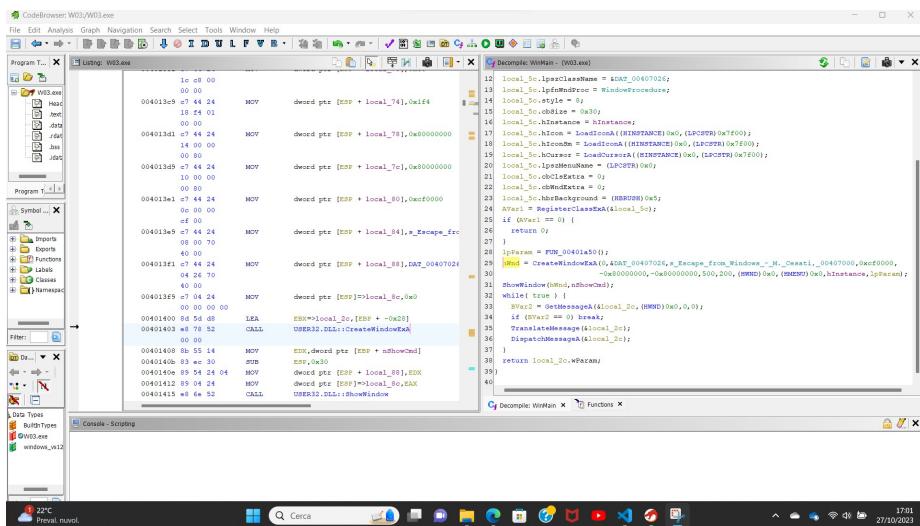
typedef struct tagCREATESTRUCTA {
    LPVOID    lpCreateParams;
    HINSTANCE hInstance;
    HMENU     hMenu;
    HWND      hwndParent;
    int       cy;
    int       cx;
    int       y;
    int       x;
    LONG      style;
    LPCSTR    lpszName;
    LPCSTR    lpszClass;
    DWORD     dwExStyle;
} CREATESTRUCTA, *LPCREATESTRUCTA;

```

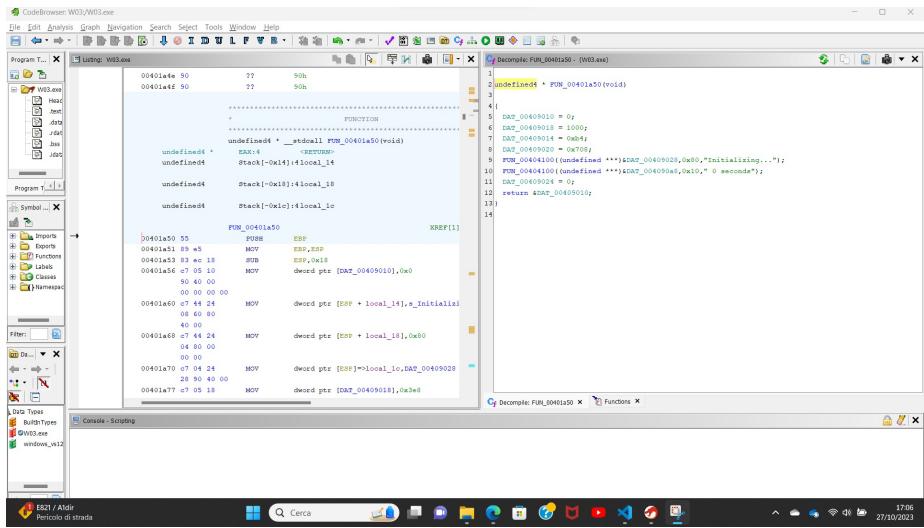
In particolare attraverso il comando `MOV EAX, dword ptr [EDI]` si sta leggendo il primo campo della struttura in particolare `lpCreateParams` che contiene dati addizionali che possono essere usati per creare la finestra, se la finestra è stata creata come risultato di una chiamata a `CreateWindowEx` (questo caso), contiene il valore del parametro `lpParam` specificato nella function call.

Quindi, per trovare quello che si sta cercando bisogna tornare a `CreateWindowEx` dentro `WinMain` perché li si è deciso il valore di quel parametro che poi è passato automaticamente qui (tornare a `WinMain`, cercare etichetta con tasto `G`).

file:///h



Questa funzione si può vedere meglio guardandola dal decompilatore ([Window->Decompile](#)), in particolare `lpParam` è l'ultimo parametro. Questo è stato impostato alla riga precedente come risultato della funzione `FUN_00401a50` (che è l'unica differenza rispetto al programma precedente). Adesso bisogna vedere cosa c'è in questa funzione, per farlo faccio doppio clic sulla label.



La si può vedere anche dal decompilatore perché è semplice, in particolare ci si accorge che inizializza alcune strutture di dati (si possono convertire i valori in decimale per vedere i corrispondenti valori) per poi invocare funzioni (possono essere o semplici o molto complesse).

Bisogna rinominarla per ricordare cosa fa, chiamiamola **initDS** (tutte le istruzioni aggiornano indirizzi adiacenti e restituisce il puntatore al primo indirizzo, sta ritornando un puntatore ad una struttura).

**initDS** è la funzione che inizializza la struttura dati, e poi passa il primo indirizzo, associato alla prima variabile assegnata, ovvero il pointer a **DAT00409010**

Se si arriva a funzioni troppo complesse non bisogna perdere tempo a cercare di capire cosa fa, meglio capirlo a posteriori. Bisogna capire prima quello che ci interessa, altrimenti si rischia di perdere tempo a cercare cose che non interessano (potrebbe essere una funzione di libreria).

Bisogna seguire le strutture di dati, non il codice.

Inoltre, si è visto che tutto dipende dal valore di ritorno di questa funzione. Si va a vedere quindi cosa restituisce la funzione e ci si accorge che è un indirizzo di una certa variabile a **00409010**.

```

file:///h
00401ab1 e8 4a 26      CALL      FUN_00404100
00 00
00401ab6 c7 05 24      MOV       dword ptr [DAT_00409024],0x0
90 40 00
00 00 00 00
00401ac0 b8 10 90      MOV       EAX,DAT_00409010
40 00
00401ac5 c9             LEAVE
00401ac6 c3             RET

```

Cliccando sul riferimento due volte Ghidra porta a dove si trova questa variabile.

In particolare, questa si trova nel segmento **.bss**, ed è inizializzata a 0.

```

// .bss
// ram:00409000-ram:00409b0f
//

DAT_00409000          XREF[4]:    004001fc(*),
                                FUN_00401150:00401195(*),
                                FUN_00401150:0040119a(*),
                                FUN_00401150:00401235(R)

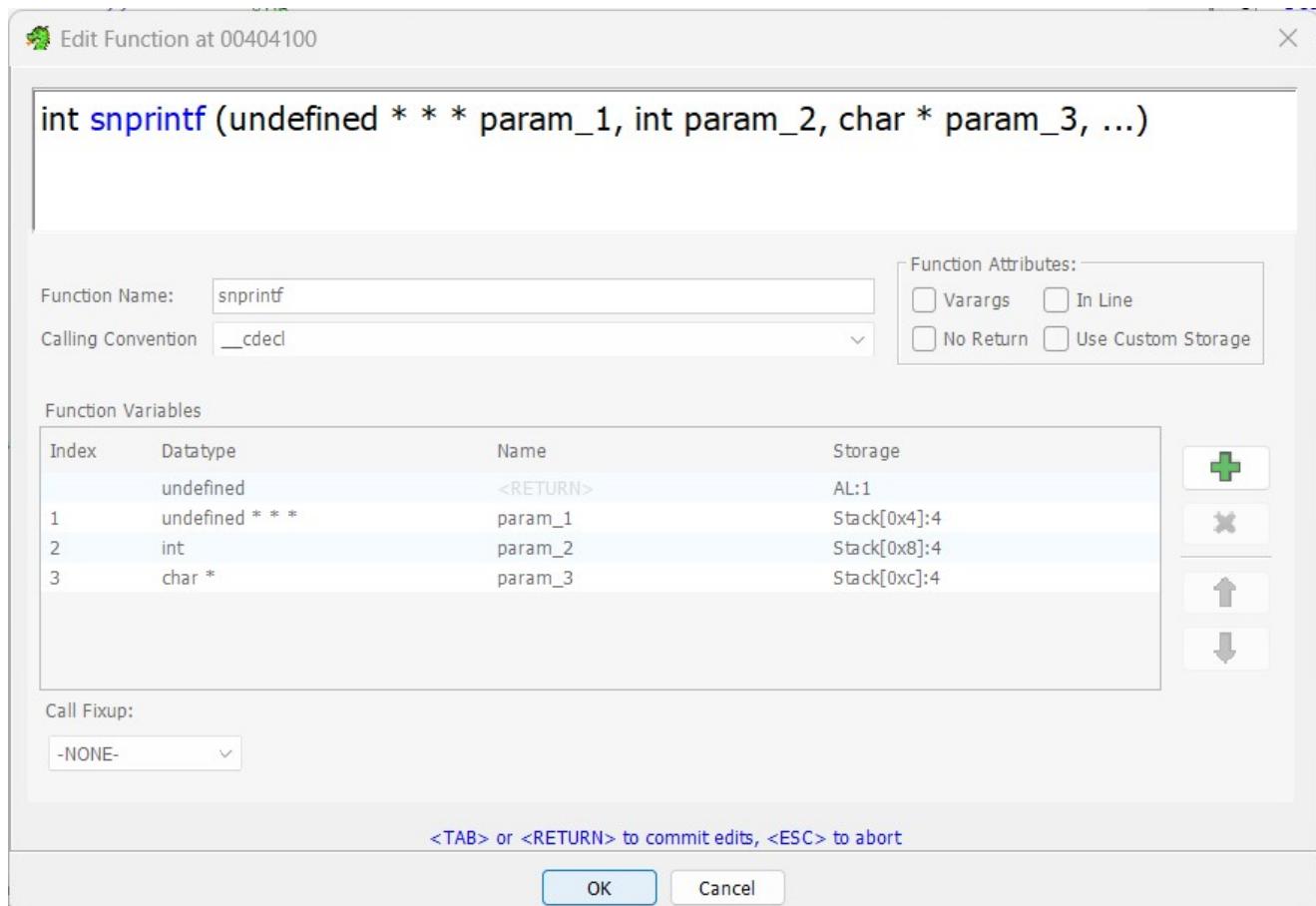
00409000      undefined... ??          XREF[2]:    FUN_00401150:00401183(*),
                                                FUN_00401150:0040123e(R)

00409004      undefined... ??          XREF[0]:    initDS:00401a56(W),
                                                initDS:00401ac0(*),
                                                FUN_00401ac7:00401aea(R),
                                                FUN_00401bc5:00401c84(R),
                                                FUN_00401ecd:00401ede(R),
                                                00402053(R), 00402063(W),
                                                004020b5(R)
00409008      ??      ??          XREF[0]:
00409009      ??      ??          XREF[0]:
0040900a      ??      ??          XREF[0]:
0040900b      ??      ??          XREF[0]:
0040900c      ??      ??          XREF[0]:
0040900d      ??      ??          XREF[0]:
0040900e      ??      ??          XREF[0]:
0040900f      ??      ??          XREF[0]:

```

Potremmo chiederci perché proprio questa variabile restituisce il valore e non tutte le altre, l'idea potrebbe essere che questa è una **struct**, (bisogna verificarlo).

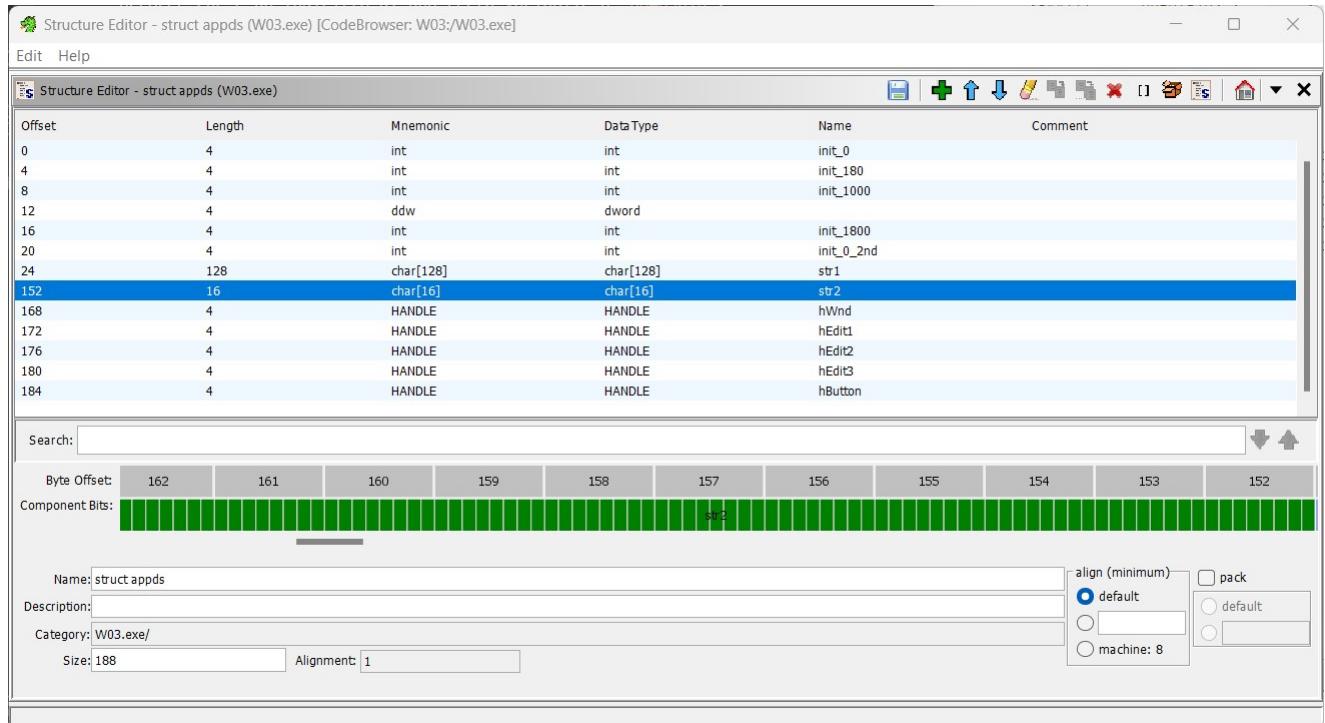
(AGGIUNTA) La funzione **FUN\_00404100** prende vari argomenti di cui l'ultimo è una stringa di formato, vedendo la stringa di formato posso pensare sia una **printf** allora cerco il **man printf** (in particolare il 3), mi accorgo che questa funzione è una **snprintf** (guardando i vari prototipi). Una volta capito questo andiamo ad aggiornare la funzione su Ghidra aggiornando il prototipo.



Da questo si capisce che **DAT\_004090a8** è una stringa **0x10** mi dà indicazioni sulla lunghezza della stringa (**16**).

Inoltre si capisce che la stringa che viene scritta precedentemente è una nuova stringa lunga **0x80** (128). Aggiorno queste informazioni alla struttura (oltre alle informazioni riguardanti i numeri aggiunti all'inizio di **initDS**). Come lo faccio?

**Window->Data Type Manager -> new Structure**, poi vado su **DAT\_00409010** e gli associo **struct appds** premendo tasto destro, **choose data type**, premo i tre puntini in alto a destra, e cerco la struttura in questione.



(La seconda parte dopo le stringhe **str2** è già aggiornata perché vista da videolezione).

Di conseguenza, la variabile locale da cui è partita l'analisi **local\_b8** (cercarla in **handle\_command\_msg**, tasto **G** per cercare), la si può rinominare **l\_appds**, e conterrà in locale l'indirizzo di questa ipotetica struttura inizializzata da **initDS**.

```

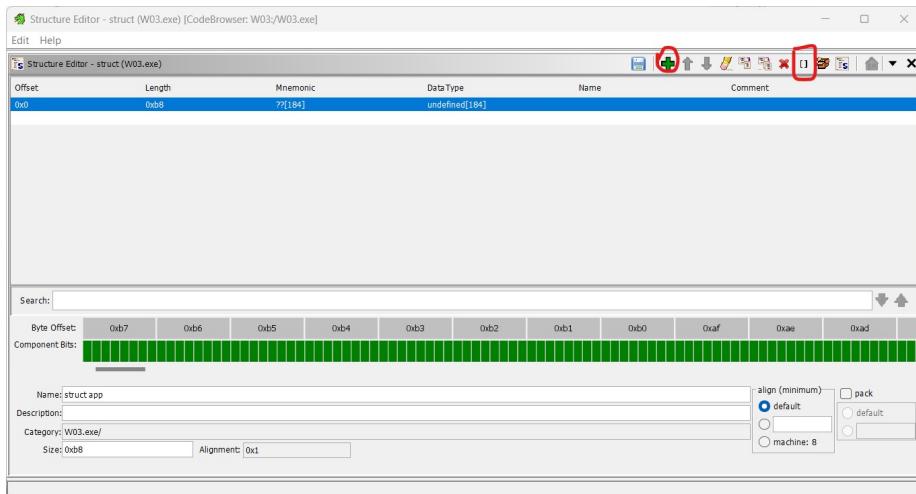
file:///h
0040163f 8b 95 4c      MOV      EDX,dword ptr [EBP + l_appds]
                      ff ff ff
00401645 39 ba b8      CMP      dword ptr [EDX + 0xb8],EDI
                      00 00 00
0040164b 0f 85 aa      JNZ      exiting_with_0
                      fe ff ff

```

**EDI** viene confrontato con **[EDX + 0xb8]** e quindi sembra molto probabile che sia una struttura (sto andando a prendere un offset rispetto all'inizio della struttura, un campo della struttura).

A questo punto tornare a **initDS** e rinominare il campo di inizio struttura (chiamarlo ad esempio **AppDS**), dopo di che siccome è una struttura conviene cominciare a riempirla (**Window->Data Type Manager** e nella finestra che si apre **tasto destro sul nome dell'applicazione -> New -> Structure...**; nella finestra che si apre dare un nome alla struttura **struct appds** e cominciare a riempirla -> ad esempio si sa che all'offset **184 (0xb8 in decimale)** di questa struttura si ha un elemento di tipo puntatore (si mette tipo **UINT** sicuramente è 32 bit) e lo si chiama **field\_184** (perché ancora non si sa cosa è), poiché prima non si sa cosa ci sia si mette un vettore di 184 byte non definiti cliccando prima sul '+' e poi sulle '[]').

Salvare le modifiche fatte alla struttura (**floppy** in alto) e chiuderla (altrimenti chiudere direttamente e nella finestra di dialogo che si apre dire di voler salvare).



A questo punto si può tornare sul codice a **AppDS** e selezionare il tipo struttura **struct appds** (tasto destro -> Data -> Choose Data Type..., scorciatoia **T**).

Adesso tutti i riferimenti ad appds danno riferimenti rispetto all'offset che Ghidra conosce (se serve si può allungare e aggiungere campi).

Tornare ora, poiché sembrava promettente al messaggio **handle\_create\_msg**.

Si vede che c'è una **GetWindowLong** anche se la set è sotto, questo perché il codice di comando in questo caso è **-6** (tradotto già in decimale).

```
004018c5 c7 44 24      MOV     dword ptr [ESP + fn_arg2], -6
                        04 fa ff
                        ff ff
004018cd be 01 00      MOV     ESI, 0x1
                        00 00
004018d2 bb 55 08      MOV     EDX, dword ptr [EBP + hWnd]
004018d5 89 14 24      MOV     dword ptr [ESP] => local_10c, EDX
004018d8 e8 cb 4d      CALL    USER32.DLL::GetWindowLongA
                                         LONG Get
```

E **-6** rappresenta, come si vede dalla documentazione, un handle all'istanza dell'applicazione in esecuzione. Allora si fa un **Rename Equate (E)** e si imposta il valore a **GWL\_HINSTANCE**.

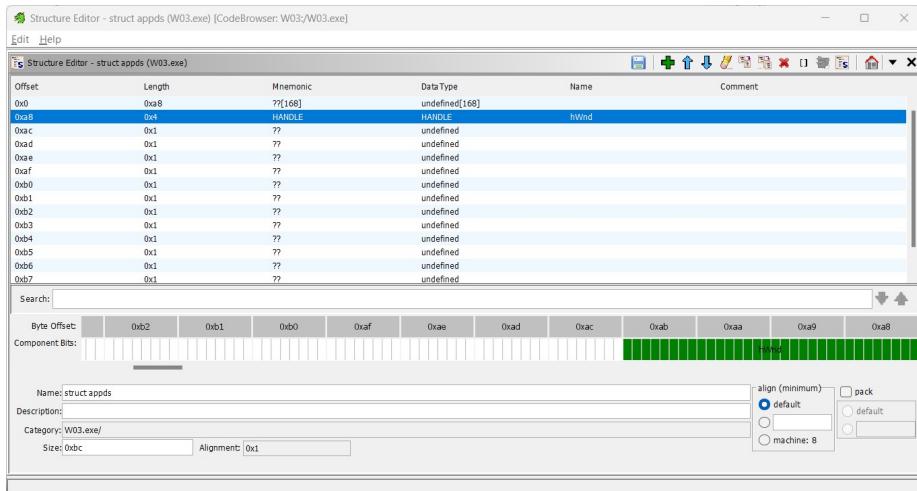
file:///h

L'handle viene messo in **local\_bc** che a questo punto rinominiamo **l\_hInstance** (tasto **L**). (Dobbiamo salire sopra). Infatti, la funzione torna un qualcosa che viene copiato da **EAX** a **local\_bc**.

A seguito della **SetWindowLong** il codice si prende l'offset a cui punta **EDI**, primo parametro della struttura **CREATESTRUCT** e quindi è quello passato nella **CreateWindowEx** (stessa istruzione di riga **004018e6**), dove siamo saliti, e lo mette in **EBX** (in **EBX** c'è sempre **AppDS**).

```
004018e6 8b 07      MOV     EAX, dword ptr [EDI] ←
004018e8 89 44 24 08  MOV     dword ptr [ESP + fn_arg3], EAX
004018ec c7 44 24      MOV     dword ptr [ESP + fn_arg2], GWL_USERDATA
                        04 eb ff
                        ff ff
004018f4 bb 45 08      MOV     EAX, dword ptr [EBP + hWnd]
004018f7 89 04 24      MOV     dword ptr [ESP] => local_10c, EAX
004018fa e8 f1 4d      CALL   USER32.DLL::SetWindowLongA
                        00 00
004018ff bb 1f      MOV     EBX, dword ptr [EDI] ←
```

A questo punto viene memorizzato nel campo ad offset **0xa8** [rif **00401907**] di questa struttura **EDX**, dove **EDX** è l'handle della finestra. Allora all'offset **0xa8** (168 in decimale) c'è l'handle della finestra (ce lo vado a mettere: **Window -> Data Type Manager -> struct appds** al posto di **[184]** nella prima riga inserisco **[168]** ossia 168 byte li lascio indefiniti, per poi andare a modificare il **Data Type** del byte seguente inserendo **HANDLE** e il campo **Name** inserendo **hWnd**; gli altri byte li lascio ancora non definiti). (NOTA dato che il registro in quella zona vale sempre l'inizio della struttura posso selezionare quella parte di codice **tasto destro -> set register values** a questo punto immettere nel registro il nome del registro che si sta modificando e in value il valore esadecimale dell'indirizzo della variabile a cui è settata).



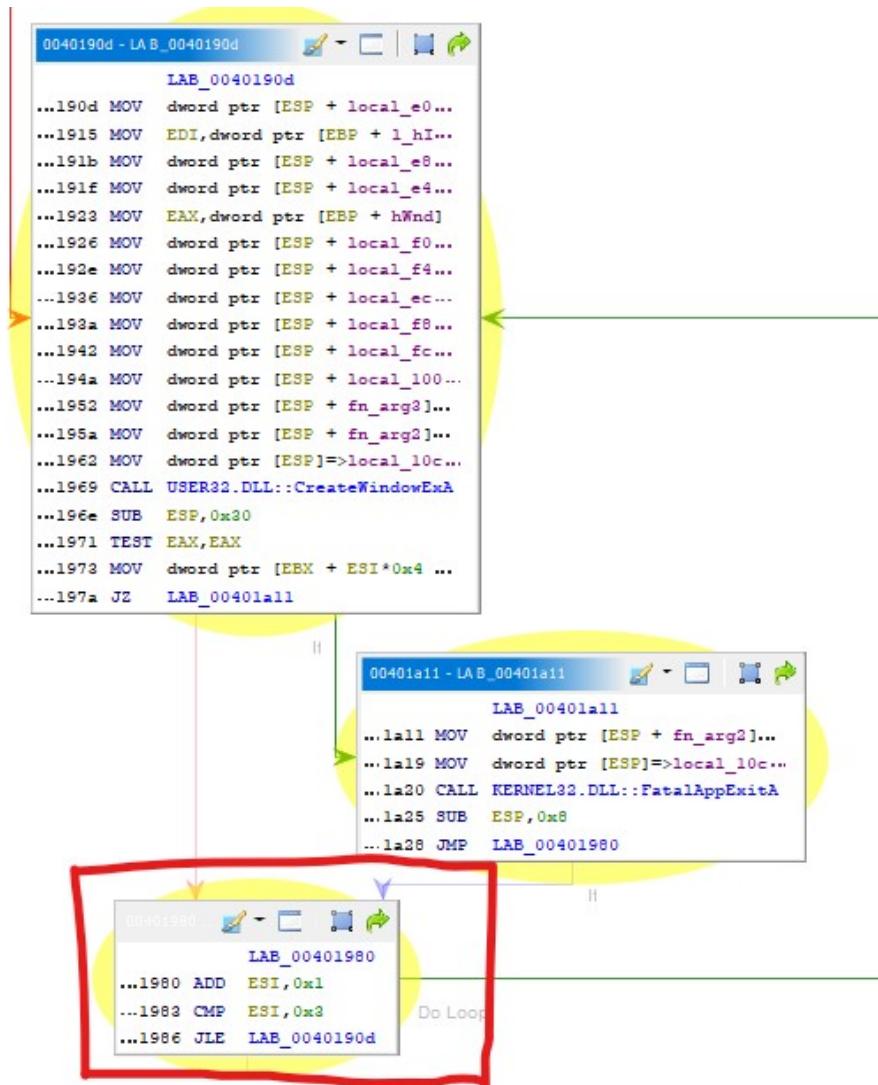
A partire da riga **0040191b** si sta preparando l'invocazione di una funzione, in particolare della **CreateWindowEx**.

```

0040191b 89 74 24 24      MOV      dword ptr [ESP + local_e8], ESI
0040191f 89 7c 24 28      MOV      dword ptr [ESP + local_e4], EDI
00401923 8b 45 08          MOV      EAX, dword ptr [EBP + hWnd]
00401926 c7 44 24          MOV      dword ptr [ESP + local_f0], 0x0
    1c 00 00
    00 00
0040192e c7 44 24          MOV      dword ptr [ESP + local_f4], 0x0
    18 00 00
    00 00
00401936 89 44 24 20      MOV      dword ptr [ESP + local_ec], EAX
0040193a c7 44 24          MOV      dword ptr [ESP + local_f8], 0x0
    14 00 00
    00 00
00401942 c7 44 24          MOV      dword ptr [ESP + local_fc], 0x0
    10 00 00
    00 00
0040194a c7 44 24          MOV      dword ptr [ESP + local_100], 0x50802002
    0c 02 20
    80 50
00401952 c7 44 24          MOV      dword ptr [ESP + fn_arg3], 0x0
    08 00 00
    00 00
0040195a c7 44 24          MOV      dword ptr [ESP + fn_arg2], DAT_00408031
    04 31 80
    40 00
00401962 c7 04 24          MOV      dword ptr [ESP] => local_10c, 0x0
    00 00 00 00
00401969 e8 12 4d          CALL    USER32.DLL::CreateWindowExA
    00 00

```

Bisogna chiamare una nuova funzione per creare gli `editBox`, la finestra ancora non ha nulla dentro, bisogna crearne almeno 3 (se vado a prendere `Window -> Function Graph` e vado al codice dedicato a `CREATE` e espando, mi accorgo che c'è un loop eseguito 3 volte in cui si va ogni volta a fare una `CreateWindowEx`; il numero di volte è data dalla `CMP` a riga `00401983`).



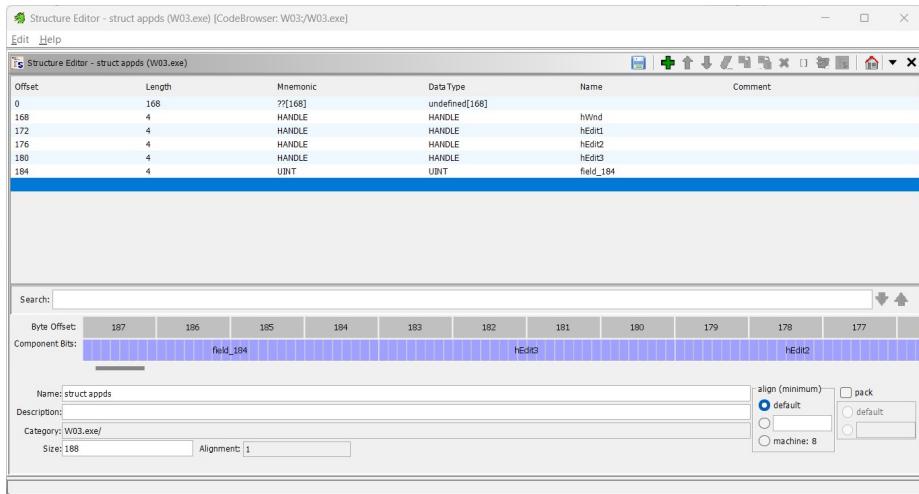
Il risultato della `CreateWindowEx` è l'handle del controllo che viene restituito in `EAX` viene inserito  
file:///h sempre all'interno della struttura `appds` ad un offset che dipende dall'indice del controllo (la base è `EBX`, ci somma `a8` che è la base, e poi ci somma `ESI * 4` dove `ESI` parte da 1, quindi non è `168` ma è quello sotto).

```

00401969 e8 12 4d      CALL    USER32.DLL::CreateWindowExA
          00 00
0040196e 83 ec 30      SUB    ESP,0x30
00401971 85 c0          TEST   EAX,EAX
00401973 89 84 b3      MOV    dword ptr [EBX + ESI*0x4 + 0xa8],EAX
          a8 00 00 00

```

Si può modificare di nuovo la struttura `struct appds` inserendo all'offset 172 4 byte di tipo `HANDLE` e lo chiameremo `hEdit1`; i 4 byte successivi sarà un nuovo `HANDLE` che chiamiamo `hEdit2`; anche i 4 byte successivi sono un `HANDLE`, in particolare, `hEdit3`. Ne metto tre perchè ho un loop che viene eseguito tre volte.



**NOTA:** Per arrivare all'obiettivo bisogna seguire il dato non il codice, anche se sembra una perdita di tempo spesso analizziamo codice non direttamente collegato perché ci dà informazioni su come sono strutturati i dati.

Andando avanti troviamo una nuova [CreateWindowEx](#) [rif. 004019e8], questa in particolare è più facile da analizzare perché sta dando dei messaggi (es. rif. 004019d9 dove si vede [stringa button](#)).

```

004019b5 89 7c 24 20      MOV      dword ptr [ESP + local_ec],EDI
004019b9 c7 44 24      MOV      dword ptr [ESP + local_f8],0x0
    14 00 00
    00 00
004019c1 c7 44 24      MOV      dword ptr [ESP + local_fc],0x0
    10 00 00
    00 00
004019c9 c7 44 24      MOV      dword ptr [ESP + local_100],0x50000001
    0c 01 00
    00 50
004019d1 c7 44 24      MOV      dword ptr [ESP + fn_arg3],DAT_00408036
    08 36 80
    40 00
004019d9 c7 44 24      MOV      dword ptr [ESP + fn_arg2],s_BUTTON_00408039
    04 39 80
    40 00
004019e1 c7 04 24      MOV      dword ptr [ESP] => local_10c,0x0
    00 00 00 00
004019e8 e8 93 4c      CALL     USER32.DLL::CreateWindowExA
    00 00

```

file:///h

Anche [DAT\\_00408036](#) è una stringa, in particolare la stringa [Go](#), Ghidra non l'ha riconosciuta perché è troppo corta (questa cosa si nota perché andando al riferimento [doppio clic](#) notiamo i caratteri nei byte uno di seguito all'altro, per ottenere la stringa intera basta cliccare con il tasto destro sul primo carattere della stringa e poi [Data -> string](#); la stessa stringa la si nota anche nell'interfaccia dell'app).

Il risultato di questa create viene inserito all'interno dell'offset [0xb8](#) (184), mediante [MOV ptr \[EBX+0xb8\], EAX](#). Modificare la struttura (come prima) per inserire all'offset [184](#) un HANDLE con nome [hButton](#).

```

004019e8 e8 93 4c      CALL     USER32.DLL::CreateWindowExA
    00 00
004019ed 83 ec 30      SUB      ESP, 0x30
004019f0 85 c0          TEST    EAX, EAX
004019f2 89 83 b8      MOV     dword ptr [EBX + 0xb8], EAX
    00 00 00

```

Questo offset già l'avevamo impostato precedentemente come **UINT**, ci si potrebbe chiedere se prima si era sbagliato.

In particolare non si era sbagliato perché avevamo inserito questo campo gestendo il messaggio **COMMAND**, che veniva mandato in tre casi, uno di questi tre casi era il clic di un pulsante (aggiungo info alla struttura anche sovrascrivendo quelle che c'erano perché si sta capendo sempre di più).

Prima c'è un **TEST** che ci si aspetta sia diverso da 0, quindi si continua di seguito e si arriva ad un paio di call (una a **00401ac7** ed una a **00401cbd**).

```

004019fa e8 c8 00      CALL     FUN_00401ac7
    00 00
004019ff 8b 45 08      MOV     EAX, dword ptr [EBP + hWnd]
00401a02 89 04 24      MOV     dword ptr [ESP] => local_10c, EAX
00401a05 e8 b3 02      CALL    FUN_00401cbd
    00 00

```

Giusto per curiosità si prova a vedere come è fatta la **00401ac7**. In questo caso il decompilatore aiuta perché mostra che la funzione sembra fare un po' di conti (va approfondita ma non è complicata, in particolare aggiorna le **edit\_text**, quindi la chiamo **update\_text\_boxes**).

```

void FUN_00401ac7(void)
{
    HANDLE hDlg;
    uint uVar1;
    int iVar2;
    UINT uValue;
    UINT uValue_00;

    hDlg = AppDS.hWnd;
    uValue_00 = 0;
    uValue = 0;
    iVar2 = AppDS.field0_0x0._16_4_ - AppDS.field0_0x0._0_4_;
    uVar1 = AppDS.field0_0x0._8_4_ * 0x3c;
    FUN_00404100((undefined **)(AppDS.field0_0x0 + 0x98), 0x10, "%2ld seconds");
    for (uVar1 = (uint)(iVar2 * 1000) / uVar1; 0x59f < uVar1; uVar1 = uVar1 - 0x5a0) {
        uValue = uValue + 1;
    }
    for (; 0x3b < uVar1; uVar1 = uVar1 - 0x3c) {
        uValue_00 = uValue_00 + 1;
    }
    SetDlgItemInt((HWND)hDlg, 1, uValue, 0);
    SetDlgItemInt((HWND)hDlg, 2, uValue_00, 0);
    SetDlgItemInt((HWND)hDlg, 3, uVar1, 0);
    return;
}

```

L'altra funzione **00401cbd** è più semplice ancora (decompilare anche questa), in particolare in questa funzione si trova la funzione **SetTimer** (ci si poteva arrivare direttamente cercando il nome della funzione, se si fosse saputo).

```

void __cdecl FUN_00401cbd(HWND param_1)
{
    AppDS.field0_0x0._12_4_ = SetTimer(param_1, 0, AppDS.field0_0x0._8_4_, (TIMERPROC) &LAB_0040204c);
    return;
}

```

In particolare, mi accorgo che questa `SetTimer` aggiorna il campo della struttura `struct appds` ad offset `8` (quello che prima non sapevo cosa fosse), in particolare questo sarebbe un puntatore al timer (lo aggiungo nella struttura come `UINT *` e lo chiamo `timer_id`).

Mi accorgo che viene chiamata periodicamente la funzione `LAB_0040204c` periodicamente, quindi sarà questa la funzione che si preoccuperà di chiudere l'applicazione dopo un certo timeout.

Poiché è una funzione e Ghidra non l'ha riconosciuta, faccio doppio clic sull'etichetta assegnata e clicco `F` su essa in modo che Ghidra la riconosca come funzione.

Offset	Length	Mnemonic	Data Type	Name	Comment
0	4		int	init_0	
4	4		int	init_180	
8	4		int	init_1000	
12	4	UINT *	UINT *	timer_id	
16	4		int	init_1800	
20	4		int	init_0_2nd	
24	128	char[128]	char[128]	str1	
152	16	char[16]	char[16]	str2	
168	4	HANDLE	HANDLE	hWnd	
172	4	HANDLE	HANDLE	hEdit1	
176	4	HANDLE	HANDLE	hEdit2	
180	4	HANDLE	HANDLE	hEdit3	
184	4	HANDLE	HANDLE	hButton	

A questo punto andiamo a cercare la documentazione della funzione `SetTimer`, sembra tutto abbastanza facile.

In particolare prende come primo argomento l'handle della finestra a cui è associato il timer, come secondo un timer identifier, come terzo parametro un timeout in millisecondi (campo della struttura inizializzato con 1000, quindi c'è un timer che scatterà ogni secondo) e infine il puntatore alla funzione da notificare quando il timer scade.

Possiamo quindi rinominare il campo della struttura che viene passato come terzo parametro `tick_length` e rinominare la funzione invocata `TimerProc`.

Lo faccio andando in `struct appds` e da lì cambio il nome di `int_1000`

Questa funzione incrementa ogni secondo un certo campo della struttura (mantiene il tempo passato). Possiamo cambiare il nome del campo di questa struttura `ticks`.

```
1
2 void TimerProc(HWND param_1)
3 {
4     uint uVar1;
5
6     uVar1 = AppDS.ticks + 1;
7     AppDS.ticks = uVar1;
8     sprintf((undefined *** )AppDS.str1,0x80,"This demo version will terminate in %lu s"
9             ,AppDS.init_180 - uVar1);
10    if (AppDS.init_0_2nd != 0) {
11        update_text_boxes();
12    }
13
14    RedrawWindow(param_1,(RECT *)0x0,(HRGN)0x0,5);
15    if ((uint)AppDS.init_180 < (uint)AppDS.ticks) {
16        PostQuitMessage(0);
17    }
18    if ((AppDS.init_0_2nd != 0) && ((uint)AppDS.init_1800 <= uVar1)) {
19        FUN_00401ecd();
20        return;
21    }
22    return;
23}
24
```

In seguito segue in una delle stringhe il messaggio aggiornato riguardo la scadenza della demo. Il campo della struttura che mantiene 180 lo chiamiamo allora `demo_length` perché è da qui che viene preso il tempo di demo.

Una volta al secondo viene controllato il valore di `run_flag` (vale 0 se non stai facendo il timeout per spegnere, 1 se il timeout sta andando avanti, lo vedo perché cliccando su `Go` vedo aggiornarsi le edit box) per vedere se aggiornare o meno le edit boxes.

In seguito viene ridisegnata la finestra e poi viene controllato se il tempo della demo è scaduto per poi andare in uscita.

file:///h

A questo punto si hanno tre strategie (che possono funzionare o meno a seconda di quanto è stato paranoico il programmatore):

- aumentare il tempo di timer prima della chiusura
- disabilitare il codice dell'if
- disabilitare direttamente il timer, questa non funziona perché il timer serve anche a spegnere la macchina

In seguito c'è un campo inizializzato a 1800 misterioso utilizzato in and con un'altra condizione (tempo scaduto) in un if per chiamare una certa funzione di shutdown.

Abbiamo capito dove lavorare, bisogna a questo punto modificare il programma.

Andiamo a riferimento `004020c3` a vedere cosa succede quando si arriva al primo if e modificare questa funzionalità per disabilitarla. Mettendo il cursore sull'riferimento questo tra le varie informazioni

ci dice anche **Byte Source Offset** che è pari a **14c3h** (h sta per esadecimale) e indica nel codice macchina quale è l'offset dei byte di quella determinata istruzione (aiuta a capire il punto in cui devo modificare il codice macchina).

004020ba	83 ec 10	SUB	ESP, 0x10
004020bd	39 05 14	CMP	dword ptr [AppDS.init_180], EAX
	90 40 00		
004020c3	72 18	JC	LAB_004020dd
004020c5		<b>Imagebase Offset</b>	+20c3h
		<b>Memory Block Offset</b>	.text +10c3h
004020ca		<b>Function Offset</b>	TimerProc +77h
004020cc		<b>Byte Source Offset</b>	File: W03.exe +14c3h
004020ce			20d6
			[AppDS.init_1800], EBX

Nel codice macchina abbiamo che la funzione che non fa nulla è la NOP 90. L'idea è mettere NOP al posto della jump a `postQuitMessage`, nella zona identificata da `LAB_004020dd`.

A questo punto andiamo a modificare l'istruzione per mettere una NOP, questo può essere fatta sia da Ghidra che da programmi esterni.

Per fare questo esistono diversi tool (editor binario), in Linux [okteta](#). Dopo aver creato una copia dell'eseguibile, aprirlo con il tool.

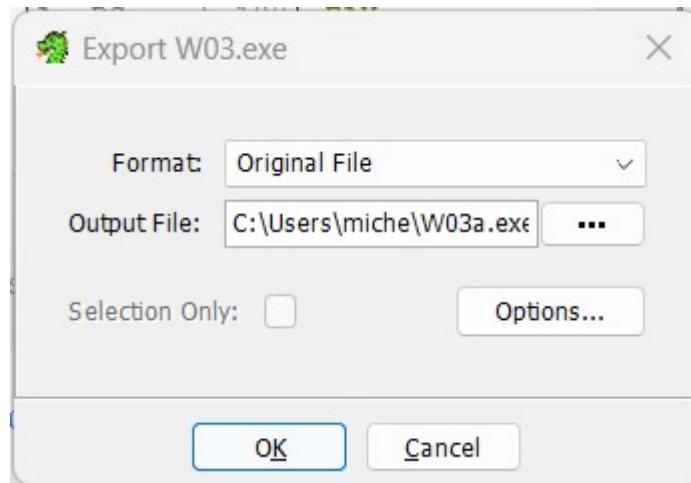
Salto al punto che mi interessa (**ctrl+G**) e inserisco **14c3**, questo mi porta a **72 18** che è il codice dell'operazione di **JC** vista su Ghidra e al loro posto inserisco **90 90** (un 90 per ogni byte che voglio sostituire, sono nop, istruzione inefficace). Quando tocco i byte in questo modo, devo essere sicuro che i byte visti su okteta siano quelli che vedo anche su Ghidra, in corrispondenza dell'istruzione che devo annullare.

004020c3	90	NOP
004020c4	90	NOP

file:///h

A questo punto bisogna fare una verifica (non sono sicuro di aver raggiunto l'obiettivo), per verificare se si è raggiunto davvero l'obiettivo.

(Stessa cosa ma in Ghidra) selezionare l'istruzione da patchare e premere **tasto destro**, **selezionare patch instruction** (è una funzione sperimentale, potrebbe non funzionare o non esistere proprio). A questo punto Ghidra deve inizializzare un suo ambiente per sostituire l'istruzione macchina e inserire NOP al posto della **JC**. Una volta patchata l'istruzione bisogna ricreare il programma, ricostruendo il file eseguibile **File -> Export** modificare primo campo con original file e modificare il nome del file per evitare si sostituirlo.



Si ha il vantaggio di non dover conoscere le istruzioni macchina associate all'istruzione che vogliamo inserire, si ha lo svantaggio che non sempre funziona (dipende dal supporto del processore).

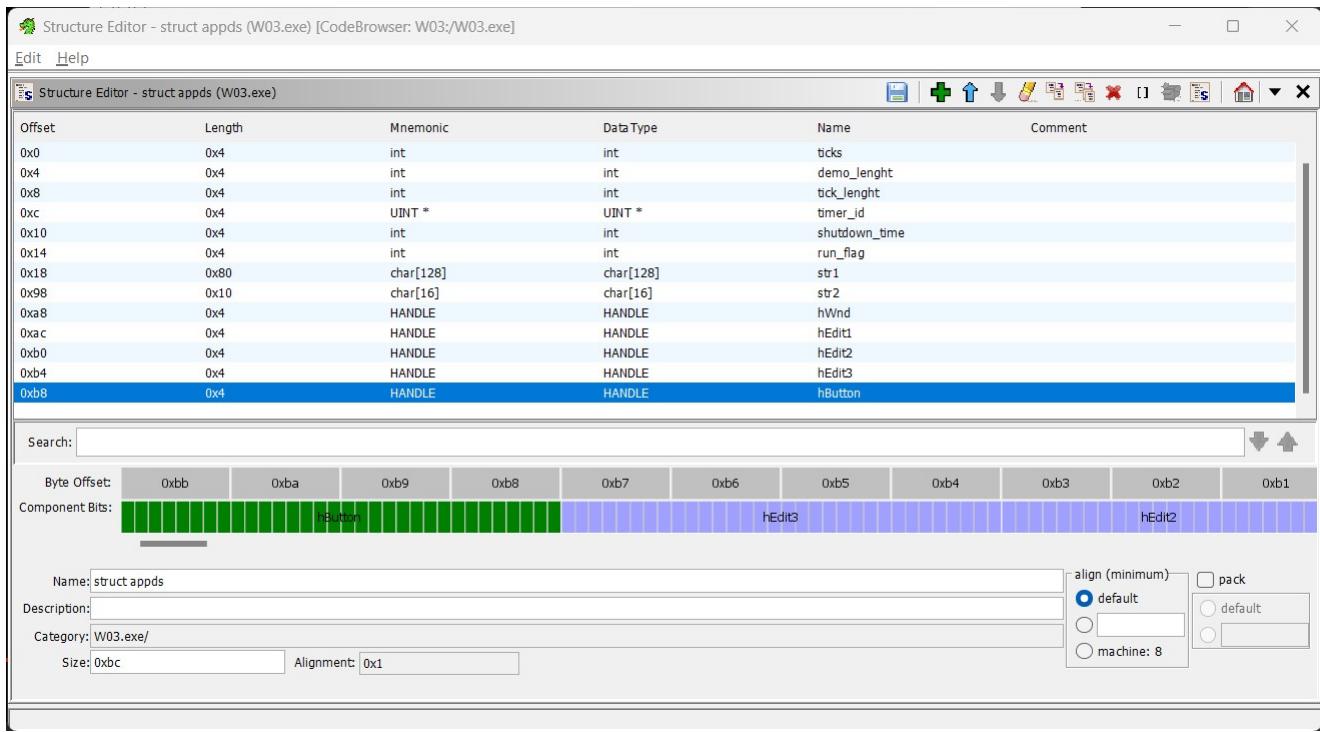
Quello che succede è che il programma si chiude lo stesso senza spegnere la macchina, **NON abbiamo risolto il nostro problema** (gran parte del lavoro è intuizione, non so se ho fatto bene).

**Cerchiamo di capire dove potrebbe essere il problema.** Potrebbe essere che questo controllo è ripetuto in altri punti.

Se vado in `struct appDS` posso modificare i campi di questa struttura in base alle nuove informazioni scoperte.

Facendo doppio clic su `appDS`, mi porta a riga `0040900f`, se faccio tasto destro → references, posso vedere i riferimenti a tale struct.

In particolare, il primo campo lo rinominiamo `ticks` (viene aggiornato ogni secondo per mantenere il tempo passato), il secondo campo è la `demo_length` (tempo di durata della demo), `init_1000` è il `tick_length` (ogni quanto viene eseguita la funzione del clock), `init_1800` è il valore in cui ci sarà lo shutdown (ogni volta viene confrontato con ticks per sapere se devo fare lo shutdown).



A partire dalla struttura in memoria si può vedere tutti i posti in cui viene fatto l'accesso al campo della struttura che contiene 180. Come ci sono arrivato? Da `references` di appDS → vedo riga `00401ecd`, ovvero la funzione di seguito.

00409014	int	??	demo_lenght	XREF[4]:	initDS:00401a81(R), FUN_00401ecd:00401ee3(R), TimerProc:00402059(R), TimerProc:004020bd(R)
----------	-----	----	-------------	----------	---

In `TimerProc` lo leggo 2 volte, la **prima** per scrivere all'utente il messaggio esplicitando il numero di secondi che mancano prima della chiusura, la **seconda** per controllare se è arrivato il momento di chiudere.

Sicuramente quindi questi punti non ci interessano, sono già coperti.

Di conseguenza vado ad analizzare l'altro punto in cui ho un accesso in lettura a questa area di memoria **FUN\_00401ecd**.

FUN_00401ecd			XRE
00401ecd 55	PUSH	EBP	
00401ece 89 e5	MOV	EBP, ESP	
00401ed0 53	PUSH	EBX	
00401ed1 83 ec 44	SUB	ESP, 0x44	
00401ed4 c7 05 24	MOV	dword ptr [AppDS.run_flag], 0x0	
90 40 00			
00 00 00 00			
00401ede a1 10 90	MOV	EAX, [AppDS]	
40 00			
00401ee3 39 05 14	CMP	dword ptr [AppDS.demo_length], EAX	
90 40 00			
00401ee9 0f 82 c4	JC	LAB_00401fb3	
00 00 00			

In questa funzione, viene letto il valore di `demo_length` e confrontato con il primo campo della struttura (`ticks`) per poi fare un `JC` (rif. 00401ee9) che se viene preso porta a `PostQuitMessage`. Sostituisco anche i 6 byte (come capisco quanti byte? Basta vedere i byte che riporta Ghidra in prossimità di tale operazione, in questo caso c'era 0f 84 d4 00 00 00) che trovo a questo punto (ad offset 12e9h nel file di byte) con le `NOP` necessarie.

NB: okteta non prende tutte le cifre dell'offset, ne posso mettere solo tre.

Anche in questo caso mi accorgo che non funziona perché si è chiusa prima dello shutdown poco tempo dopo i 3 minuti. Questo è strano perché nel controllo del timer non c'è nessun altro meccanismo. C'è un altro posto dove fa il controllo ma non lo vedo.

Questo perché salvo il puntatore di `struct_appDS` in `IParam` che passo come dati utente attraverso `SetWindowLong` (Ghidra non lo può vedere perché è un offset rispetto ad un registro).

Vado a vedere tutti i posti in cui può essere recuperato il puntatore attraverso una `GetWindowLong`. Come lo faccio?

search → search program text → `getWindowLongA`, con la spunta su All fields. A noi non interessano le tipologie `GLOBAL` nè `USER32.DLL` (scritte in Namespace, quelle in Preview posso essere interessanti invece, come vedremo).

Search Text - "getWindowLongA" [Listing Display Match] [CodeBrowser(2): Malware:/W03.exe]			
Location	Label	Namespace	Preview
0040148b		WindowProcedure	LONG GetWindowLongA(HWND hWnd, int nIndex)
0040148b		WindowProcedure	CALL USER32.DLL::GetWindowLongA
004018d8		WindowProcedure	CALL USER32.DLL::GetWindowLongA
004018d8		WindowProcedure	LONG GetWindowLongA(HWND hWnd, int nIndex)
004066a8	GetWindowLongA	USER32.DLL	JMP dword ptr [->USER32.DLL::GetWindowLongA]
004066a8	GetWindowLongA	USER32.DLL	thunk LONG __stdcall GetWindowLongA(HWND ...)
004066a8	GetWindowLongA	USER32.DLL	GetWindowLongA
004066a8	GetWindowLongA	USER32.DLL	JMP dword ptr [->USER32.DLL::GetWindowLongA]
0040a280	PTR_GetWindowL...	Global	340 GetWindowLongA <>not bound>
0040a280	PTR_GetWindowL...	Global	addr USER32.DLL::GetWindowLongA
0040a280	PTR_GetWindowL...	Global	addr USER32.DLL::GetWindowLongA
0040a280	PTR_GetWindowL...	Global	LONG __stdcall GetWindowLongA(HWND hWnd, ...)
0040a280	PTR_GetWindowL...	Global	PTR_GetWindowLongA_0040a280
0040a5ae			ds "GetWindowLongA"

/

Sono due posti entrambi nella [WindowProc](#), chiamato con due parametri diversi (il primo è [GWL\\_USERDATA](#) il secondo [GWL\\_HINSTANCE](#)).

Il secondo non ci interessa, non è [USERDATA](#), preleva una istanza infatti.

Controllo il primo, questo aggiorna una certa zona nello stack ([l\\_appds](#)). Devo vedere tutti gli accessi in lettura fatti a questa variabile nello stack. Lo vedo salendo su per il programma, dove ho tutti gli accessi nello stack, e ritrovo [l\\_appds](#).

undefined4	Stack[-0xb8]:4 l_appds	XREF[9]:	00401496 (W), 00401532 (R), 00401577 (R), 004015b6 (R), 004015f0 (R), 0040163f (R), 00401813 (R), 00401879 (R), 004018a9 (R)
------------	------------------------	----------	--

Il primo è in scrittura è quello della [GetWindowLong](#), non mi interessa.

file:///h

Il secondo è un accesso al campo a spiazzamento 0xac, quindi è un HANDLE. Non mi interessa. Fa parte della label [handle\\_size\\_msg](#) a riga [00401507](#)

Il terzo è un accesso al campo a spiazzamento 0xb0, un altro HANDLE, non mi interessa.

Fino al sesto accesso sono accessi a HANDLE, quindi non ci interessano.

Settimo e ottavo sono accessi rispettivamente alla seconda e alla prima stringa, non ci interessano. Andando a vedere l'ultimo accesso in lettura mi accorgo che c'è un accesso al quarto campo della struttura che è proprio [demo\\_length](#). Lo vedo perché viene eseguita [MOV EAX, dword ptr\[EDI\]](#), con cui referenzio il primo campo di [AppDS](#) e lo metto in [EAX](#), e poi lo confronto con [dword ptr\[EDI+4\]](#), ovvero quarto campo.

In particolare vedendo il codice macchina vedo che c'è un confronto tra [demo\\_length](#) e [ticks](#). Se il primo è superiore si salta a [PostQuitMessage](#). Tale gestione appartiene al comando [paint](#), ovvero

quando Windows ridisegna una finestra, allora viene fatto un controllo.

004018a9	8b bd 4c	MOV	EDI, dword ptr [EBP + l_appds]
	ff ff ff		
004018af	8b 07	MOV	EAX, dword ptr [EDI]
004018b1	83 ec 08	SUB	ESP, 0x8
004018b4	39 47 04	CMP	dword ptr [EDI + 0x4], EAX
004018b7	2e 0f 82	JC	LAB_004014ec
	2e fc ff ff		

Questo è il punto da patchare (rif. 004018b7), sono 7 byte di istruzione da sostituire con 90. Questo è all'offset del file cb7e e questi inziano con 2e 0f 82, una volta trovati li sostituiamo con 90 (NOP).

Salvo l'eseguibile ottenuto e riprovo per vedere se ora funziona (FUNZIONA, il tempo continua ad essere decrementato quindi diventa negativo, ma la demo continua ad eseguire finché Windows si spegne).