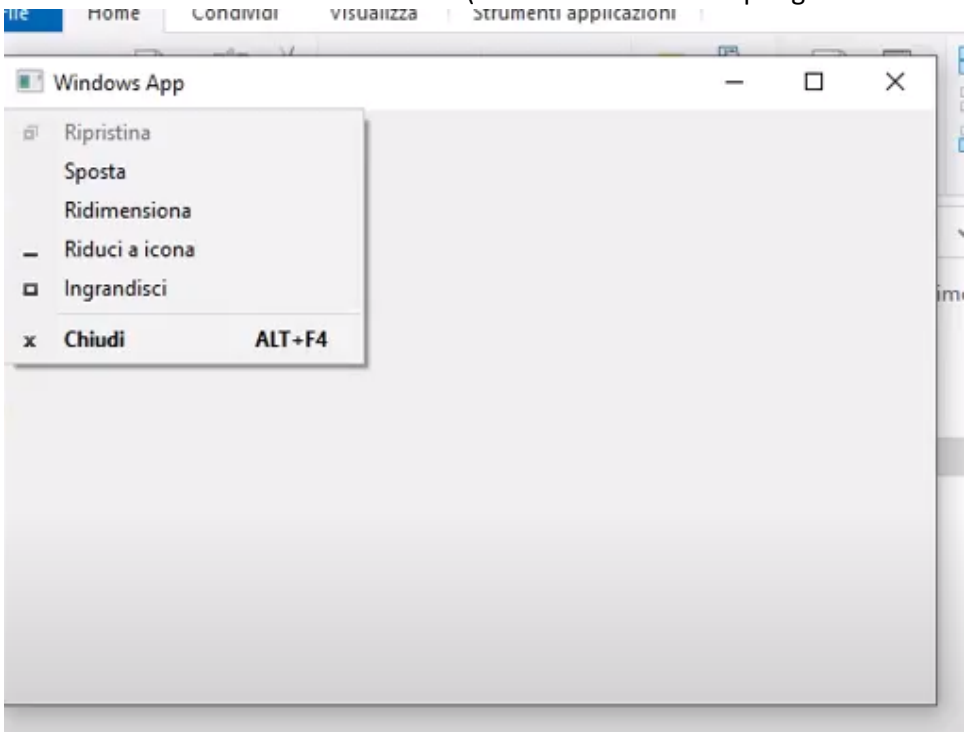
ANALISI DEL FILE W02.EXE

- Importando il programma con Ghidra, ci vengono fornite info sull'eseguibile, come ad esempio il formato 32 bit, l'architettura intel, l'utilizzo di componenti grafiche, e *stripped to external PDB:*
Ciò ci dice che potrebbero esserci delle istruzioni non necessarie ai fini dell'esecuzione, e che è possibile snellire mediante *strip*, e salvare queste cose in più in un *file.pdb*.
Tuttavia, questo non è il nostro caso.

Comportamento del file EXE

L'esecuzione su Windows riporta una schermata vuota, di cui posso modificare in dimensione.
È anche fornito da Windows un menù (ovvero la barra in alto per gestire la finestra).

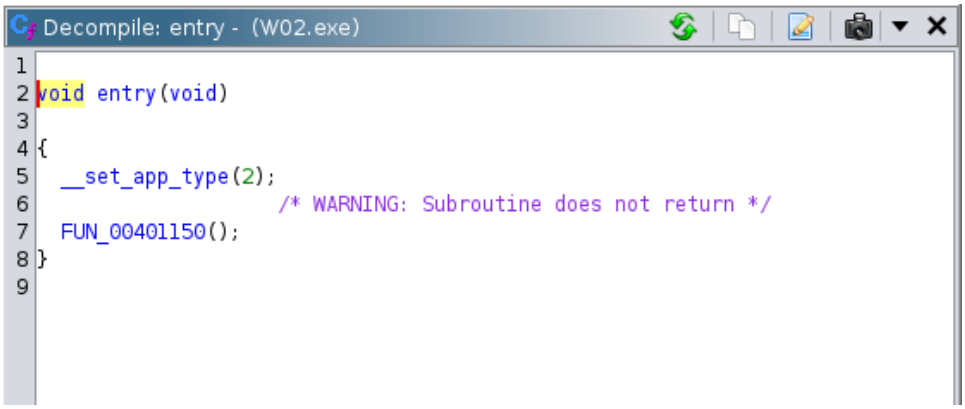


Ambiente Ghidra

Passati a Ghidra, ciò che dobbiamo fare è cercare un *entrypoint*.
Possiamo cercarne uno premendo semplicemente il tasto "G", oppure "Navigation -> Go to..."

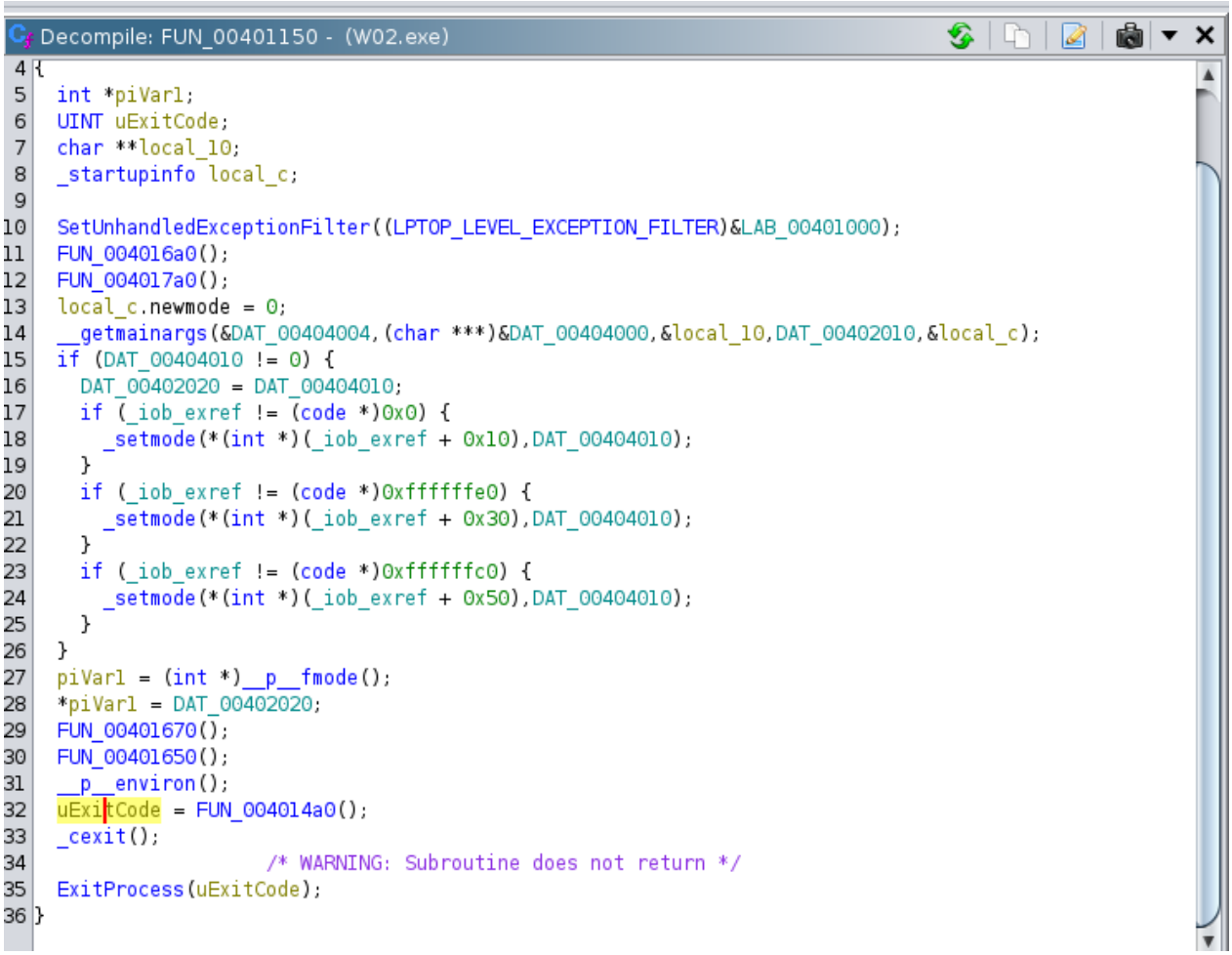
Cerchiamo semplicemente “entry”, ed arriviamo a:

Andando in “Window -> decompile” (oppure CTRL + E) avremo alla nostra destra:

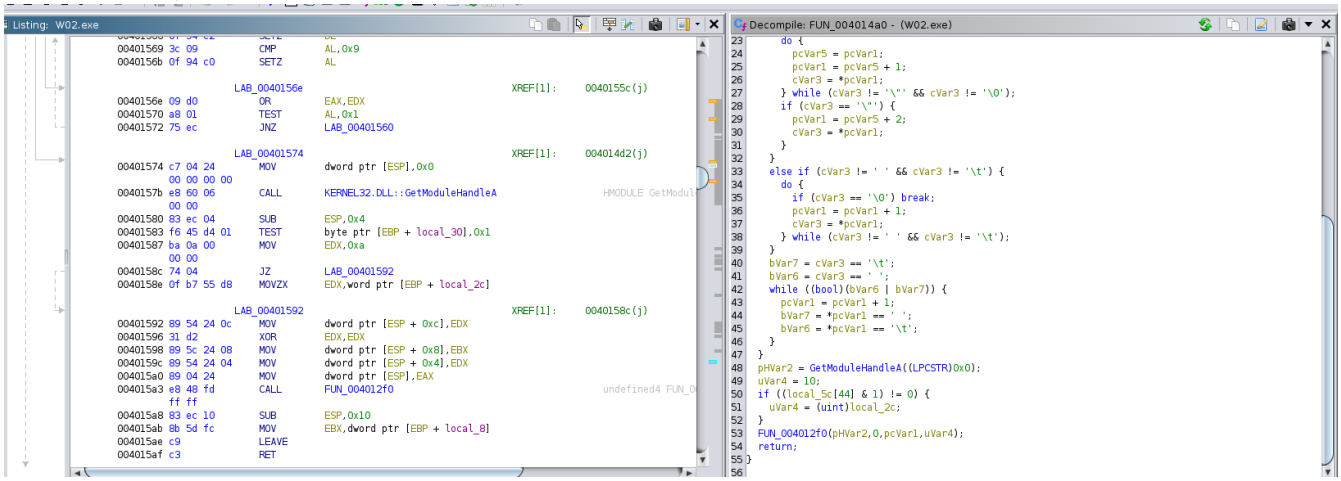


Ciò ci suggerisce di vedere *FUN_00401150*, perché a riga 5 viene fatto un setup di qualcosa, e vedendo la controparte assembly notiamo la presenza di DLL; quindi, librerie dinamiche, che per ora lasciamo in disparte. Clicchiamo sul nome della funzione a riga 7:

Aperta tale funzione, cerchiamo di vedere cosa fa:



Saltiamo alla funzione identificata a riga 32, perché precede l'exit. (il resto sembra configurazione dell'ambiente)



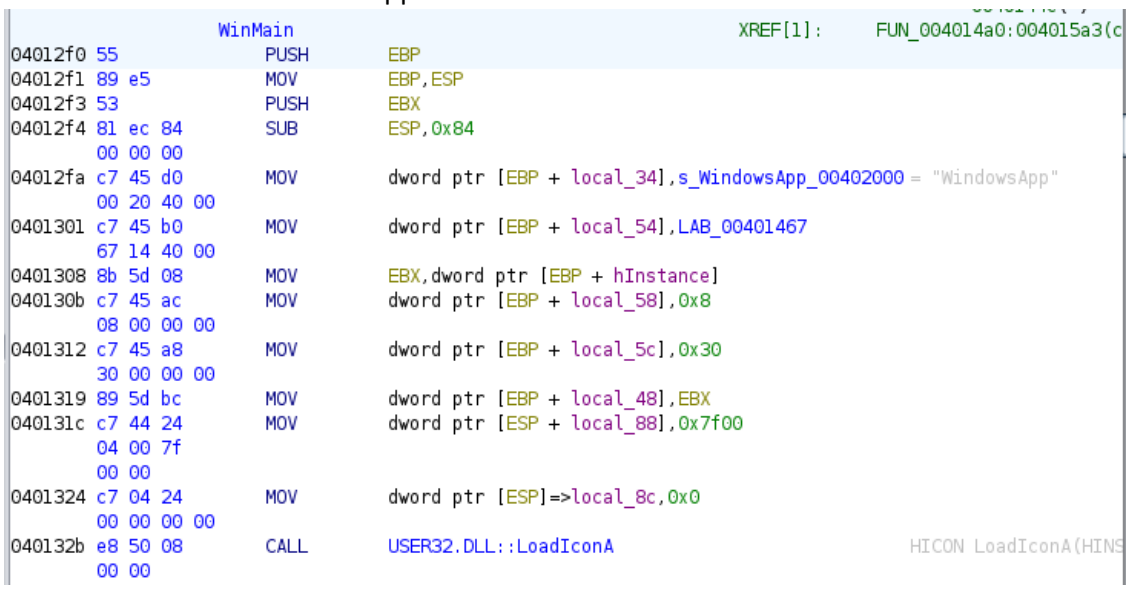
In questa nuova funzione, l'ultima chiamata è *FUN_004012fo,* che prende quattro parametri.
Il fatto che ci siano quattro parametri ci dà una informazione essenziale: abbiamo trovato il nostro **win-main**, infatti, dal manuale di Windows, vediamo che il main deve avere quattro parametri:
https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-winmain
Questo è un nome convenzionale, per cui possiamo riportarlo su Ghidra, chiamandolo proprio così:
Andiamo sulla parte assembly di Ghidra, e facciamo "Edit Function", mettendo come nome:

int WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)

Tale funzione ritorna sempre un intero, i quattro parametri ci dicono:

- oggetto che riferisce ad un altro oggetto, ma è opaco, cioè non risolvibile.
- ad oggi non si usa, settato sempre a null.
- long pointer a string, 4 byte.
- è intero, specifica come parte la finestra all'avvio.

Dopo la sostituzione, abbiamo una vista più chiara. Viene fatto il solito lavoro con lo stack, preparando quello che serve, e lasciando spazio per qualcosa che potrebbe servire.
Notiamo la dicitura "WindowsApp":



```
                    WinMain                                         XREF[1]:     FUN_004014a0:004015a3(c
04012f0 55              PUSH        EBP
04012f1 89 e5           MOV         EBP,ESP
04012f3 53              PUSH        EBX
04012f4 81 ec 84        SUB         ESP,0x84
        00 00 00
04012fa c7 45 d0        MOV         dword ptr [EBP + local_34],s_WindowsApp_00402000 = "WindowsApp"
        00 20 40 00
0401301 c7 45 b0        MOV         dword ptr [EBP + local_54],LAB_00401467
        67 14 40 00
0401308 8b 5d 08        MOV         EBX,dword ptr [EBP + hInstance]
040130b c7 45 ac        MOV         dword ptr [EBP + local_58],0x8
        08 00 00 00
0401312 c7 45 a8        MOV         dword ptr [EBP + local_5c],0x30
        30 00 00 00
0401319 89 5d bc        MOV         dword ptr [EBP + local_48],EBX
040131c c7 44 24        MOV         dword ptr [ESP + local_88],0x7f00
        04 00 7f
        00 00
0401324 c7 04 24        MOV         dword ptr [ESP]=>local_8c,0x0
        00 00 00 00
040132b e8 50 08        CALL        USER32.DLL::LoadIconA                           HICON LoadIconA(HINS
        00 00
```

Essa è la stringa che identifica il nome della finestra. Vediamo il richiamo a **LoadIconA**: essa ha due parametri, *hInstance* e *Lpcstr* (sempre cercando sul manuale). Vediamo che possono avere dei valori predefiniti dal manuale. Quindi, a riga 0401324, abbiamo che *local_8c* è il primo parametro, *(nb: su stack mettiamo i parametri dall'ultimo al primo, quindi, partendo dalla CALL, salendo troviamo i parametri in ordine "crescente")* che chiamo "*fn_arg1*", a cui passo 0x0 cioè NULL (dal manuale vuol dire che passo una icona standard), e quello sopra è il secondo, a cui passo il valore 0x7f00, cioè 32512. Però così è inutile, non mi dà info, ha più senso associarvi un nome. Ghidra ha lista nomi già pronti da associare. Basta premere il tasto "e" (rename equate), in particolare gli associamo "IDI_APPLICATION" (l'alternativa è ARROW, ma come vediamo sotto c'è un *LoadCursor*, potrebbe essere appropriato lì).

```
SUB             ESP,0x8
MOV             dword ptr [EBP + wclsex.hIcon],EAX
MOV             dword ptr [ESP + fn_arg2],IDI_APPLICATION

MOV             dword ptr [ESP]=>fn_arg1,0x0

CALL            USER32.DLL::LoadIconA
```

Ricordiamo la policy di gestione dello stack, in particolare con *SUB ESP,0x8* stiamo sottraendo 8 byte, cioè aggiungendo ad ESP il valore 8 (stiamo allocando).
Viene richiamata nuovamente la stessa funzione (probabilmente usata due volte).

LoadCursor si comporta nello stesso modo, sempre leggendo dal manuale:
in questo caso quel codice 32512 è associato a *IDC_Arrow*.

Arriviamo a *RegisterClassExA*, vediamo cosa fa:
E' una funzione che registra una classe finestra, non è una istanza, solo dopo sarà usabile.
Ha un solo parametro, un puntatore ad una struttura. Cosa passiamo?

```
0040137b c7 45 b8 00 00        MOV             dword ptr [EBP + wclsex.cbWndExtra],0x0
         00 00
00401382 c7 45 c8 05 00        MOV             dword ptr [EBP + wclsex.hbrBackground],0x
         00 00
00401389 89 04 24              MOV             dword ptr [ESP]=>fn_arg1,EAX
0040138c e8 ff 07 00 00        CALL            USER32.DLL::RegisterClassExA
00401391 31 d2                 XOR             EDX,EDX
```

Vediamo cosa viene fatto:

- *MOV dword ptr [ESP]=>fn_arg1, EAX:*
  Questa istruzione sposta il valore contenuto nel registro EAX in una posizione di memoria nello stack puntata da ESP. In altre parole, sta copiando il valore contenuto in EAX nella prima posizione della pila, che è utilizzata come argomento fn_arg1 da passare alla funzione RegisterClassExA. Quindi stiamo **passando EAX. Dove l'ho impostato?**
- EAX viene impostato a riga 00401378 "LEA *EAX =>local_5c,[EBP+ -0x58]"*. In particolare: *Local_5c* è l'indirizzo della struttura che l'API si aspetta. Allora saliamo su tra le linee di codice, e vediamo cosa c'è in questo local_5c:
  Ghidra lo mette come undefined4, ma noi sappiamo cosa è, vedendo il manuale!

  ```
  undefined4        Stack[-0x5c]:4 local_5c
  ```

  Mettiamo "Set data type", "choose data type" e ci metto *WNDCLASSEXA* (perché così è scritto sul manuale). Appare messaggio di warning: Ghidra ci dice che la struttura è grande e va a cancellare variabili locali, devo farlo? Si, perché sono sicuro che ci vada una struttura e non variabili prese singolarmente.

```
                **************************************************************
                *                          FUNCTION                          *
                **************************************************************
                int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPr...
int               EAX:4          <RETURN>
HINSTANCE         Stack[0x4]:4   hInstance                          XREF[1]:
HINSTANCE         Stack[0x8]:4   hPrevInstance
LPSTR             Stack[0xc]:4   lpCmdLine
int               Stack[0x10]:4  nShowCmd                           XREF[1]:
undefined4        Stack[-0x8]:4  local_8                            XREF[2]:

undefined4        Stack[-0x24]:4 local_24                           XREF[1]:
undefined1        Stack[-0x2c]:1 local_2c                           XREF[1]:
WNDCLASSEXA       Stack[-0x5c]:48 local_5c                          XREF[2,11]:
```

Notiamo che Ghidra ha riempito i campi della struttura, possiamo rinominare "local_5c" con il tasto L (rename Local Variable), chiamandola "*wclsex*" per promemoria. Cosa c'è nella struttura?
Dal manuale vediamo la struttura e definizione:
https://learn.microsoft.com/en-us/windows/win32/api/winuser/ns-winuser-wndclassexa

Rinominiamo la label a riga *004012fa* con "*wndProc*" (comando edit Label), in quanto vediamo che viene puntato *wclsex.lpfnProc.*
Per il parametro "style" leggiamo che potremmo avere anche combinazioni di stili, che sono valori costanti. Le macro che si usano per tali stili iniziano con CS, e Ghidra suggerisce *CS_DBLCLKS*

(ricezione eventi doppio clic). Per la size non abbiamo costanti vere e proprie, è semplicemente la dimensione della struttura, e non ci interessa.

```
004012fa c7 45 d0 00 20 40 00    MOV     dword ptr [EBP + wclsex.lpszClassName],s_WindowsApp_00402000   = "WindowsApp"
00401301 c7 45 b0 67 14 40 00    MOV     dword ptr [EBP + wclsex.lpfnWndProc],WndProc
00401308 8b 5d 08                MOV     EBX,dword ptr [EBP + hInstance]
0040130b c7 45 ac 08 00 00 00    MOV     dword ptr [EBP + wclsex.style],CS_DBLCLKS
00401312 c7 45 a8 30 00 00 00    MOV     dword ptr [EBP + wclsex.cbSize],0x30
00401319 89 5d bc                MOV     dword ptr [EBP + wclsex.hInstance],EBX
0040131c c7 44 24 04 00 7f 00 00 MOV     dword ptr [ESP + fn_arg2],IDI_APPLICATION
00401324 c7 04 24 00 00          MOV     dword ptr [ESP]=>fn_arg1,0x0
```

Procediamo scorrendo il codice:

```
                                         USER32.DLL::LoadIconA
0040132b e8 50 08 00 00          CALL    HICON LoadIconA...
                                         ESP,0x8
00401330 83 ec 08                SUB
                                         dword ptr [EBP + wclsex.hIcon],EAX
00401333 89 45 c0                MOV
                                         dword ptr [ESP + fn_arg2],IDI_APPLICATION
00401336 c7 44 24                MOV
```

Una volta presa la risorsa LoadIconA, essa viene messa in *wclsex.hIcon*, e più sotto in *wclsex.hIconSm*. (riga 0040134d).
Sarebbero icona grande ed icona minimizzata.
Scendendo a riga *0040138c* c'è "*RegisterClassExA*" che ritorna unico parametro.

Il valore è dentro AX, in cui faccio un TEST. La funzione ritorna un ATOM, un oggetto che identifica in modo univoco la classe registrata, che userò per crearla. (Sempre da manuale!)
Prima del TEST abbiamo lo XOR che azzera il contenuto di EDX:

```
00401382 c7 45 c8 05 00 00 00    MOV     dword ptr [EBP + wclsex.hbrBackground],0x5
00401389 89 04 24                MOV     dword ptr [ESP]=>fn_arg1,EAX
0040138c e8 ff 07 00 00          CALL    USER32.DLL::RegisterClassExA
00401391 31 d2                   XOR     EDX,EDX
00401393 83 ec 04                SUB     ESP,0x4
00401396 66 85 c0                TEST    AX,AX
00401399 75 09                   JNZ     LAB_004013a4
0040139b 89 d0                   MOV     EAX,EDX
0040139d 8b 5d fc                MOV     EBX,dword ptr [EBP + local_8]
004013a0 c9                      LEAVE
004013a1 c2 10 00                RET     0x10
```

L'API successiva è "createWindow", cerchiamo sulla documentazione:

Syntax

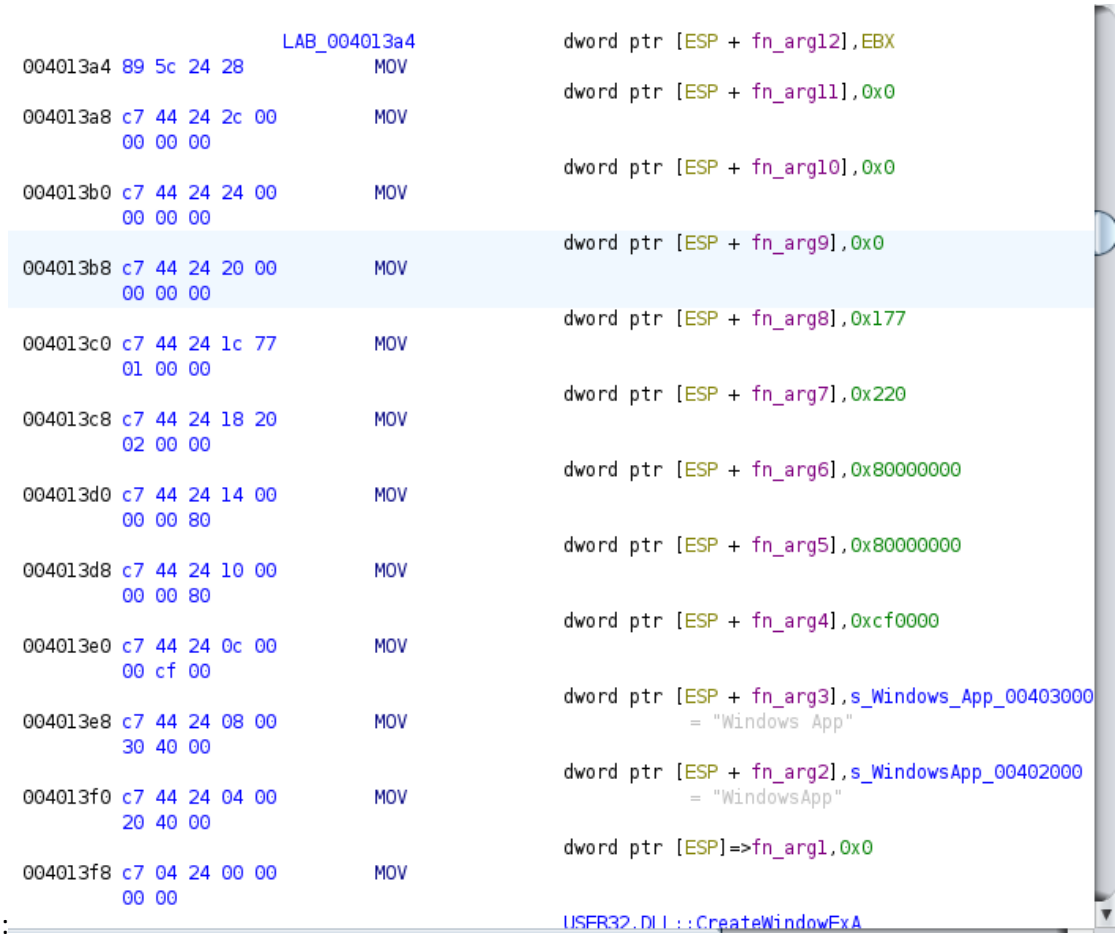
C++ Copy

```
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
);
```

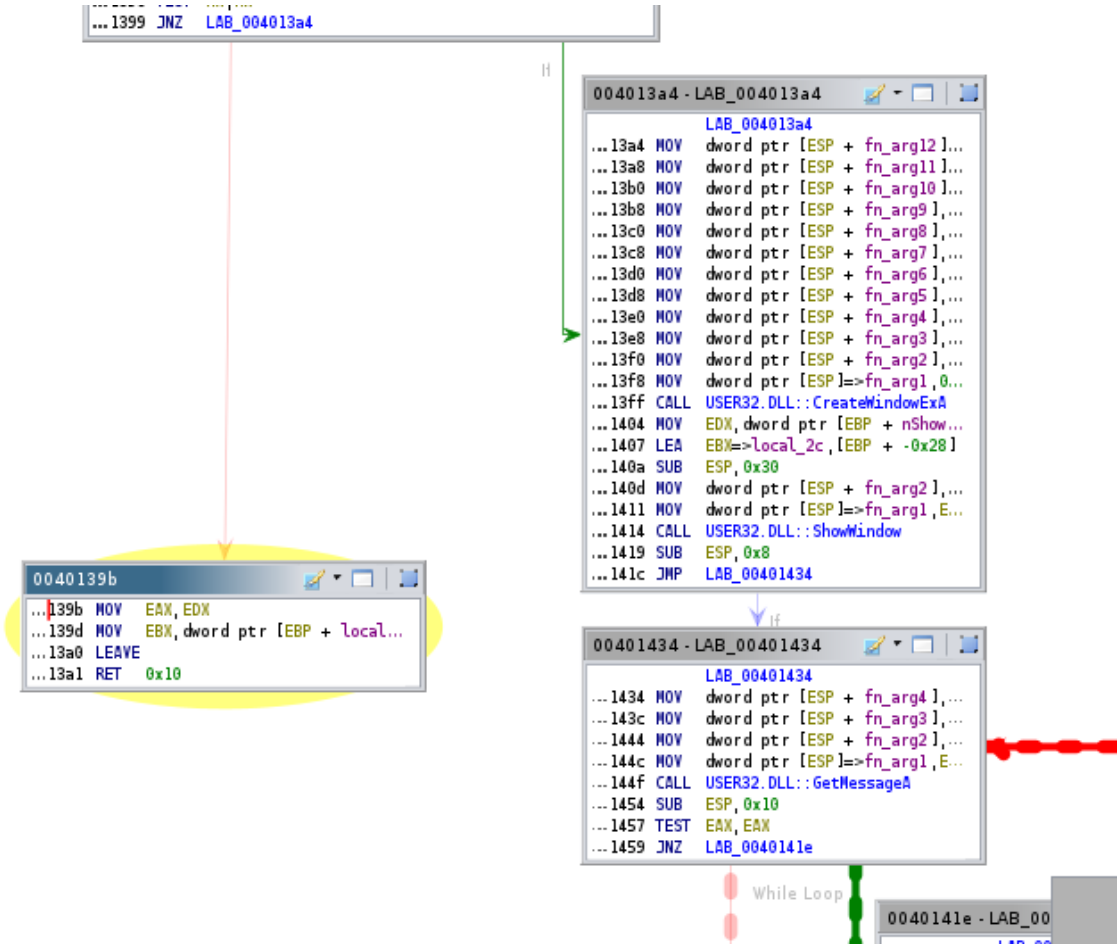
Abbiamo 12 argomenti, allora i 12 puntatori sopra tale API devo rinominarli da così:

```
004013b8 c7 44 24 20 00  MOV
         00 00 00
                                  dword ptr [ESP + local_70],0x177
004013c0 c7 44 24 1c 77  MOV
         01 00 00
                                  dword ptr [ESP + local_74],0x220
004013c8 c7 44 24 18 20  MOV
         02 00 00
                                  dword ptr [ESP + local_78],0x80000000
004013d0 c7 44 24 14 00  MOV
         00 00 80
                                  dword ptr [ESP + local_7c],0x80000000
004013d8 c7 44 24 10 00  MOV
         00 00 80
                                  dword ptr [ESP + local_80],0xcf0000
004013e0 c7 44 24 0c 00  MOV
         00 cf 00
                                  dword ptr [ESP + local_84],s_Windows_App_00403...
004013e8 c7 44 24 08 00  MOV                = "Windows App"
         30 40 00
                                  dword ptr [ESP + fn_arg2],s_WindowsApp_00402000
004013f0 c7 44 24 04 00  MOV                = "WindowsApp"
         20 40 00
                                  dword ptr [ESP]=>fn_arg1,0x0
004013f8 c7 04 24 00 00  MOV
         00 00
                                  JUSER32.DLL::CreateWindowExA
004013ff e8 94 07 00 00  CALL               HWND CreateWindowExA(...
                                  EDX dword ptr [EBP + nShowCmd]
```

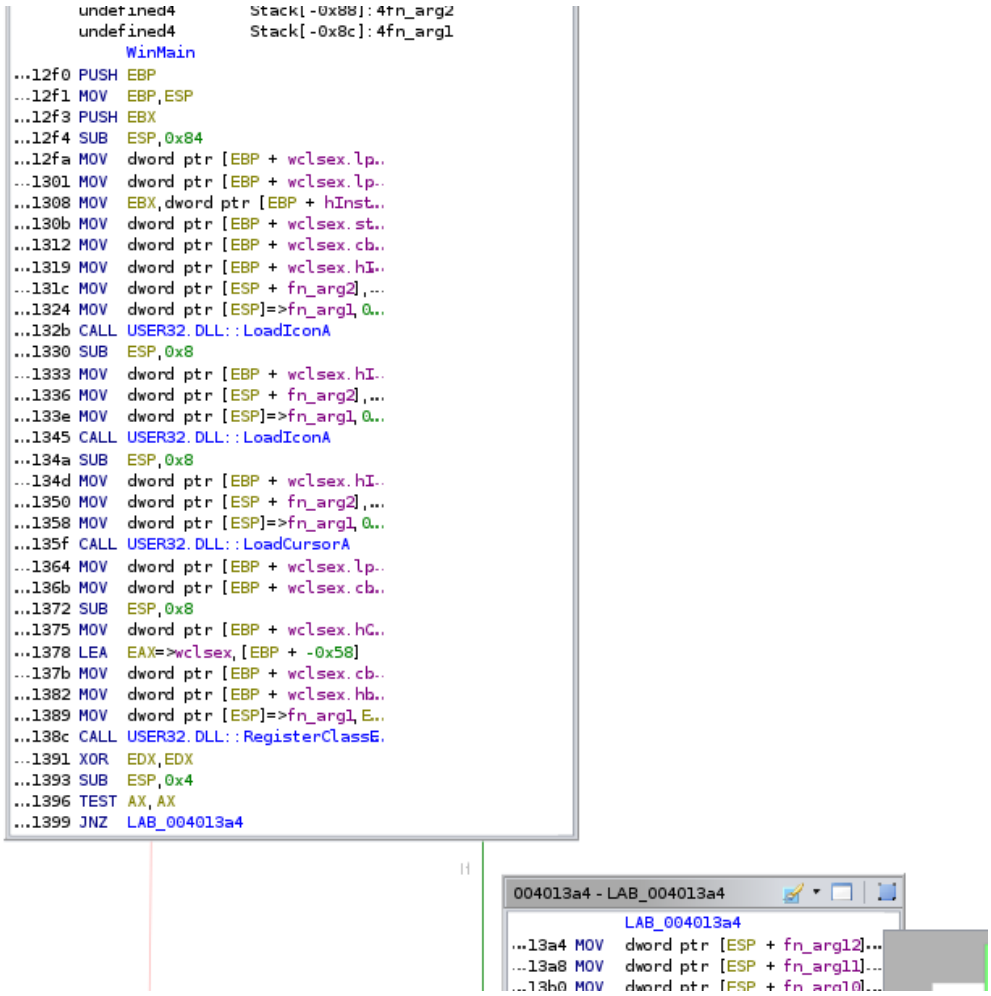
A così: (tasto "L" sui vari "local" e poi Rename Function Variable")



- primo argomento: stile 0, cioè" semplice".
- secondo argomento e terzo argomento: sono stringhe con nomi di classi, faccio collegamento tra nome classe e finestra.
- Quarto argomento: di nuovo stile, proviamo con "equate" e mettiamo "WS_TILEDWINDOW", suggerita da Ghidra (c'è anche Overlapped, ma nella run del prof aveva solo questa).
  Se avessi dubbi, dovrei vedere header_file.h il valore specifico, meglio se lo fa Ghidra al posto mio.
- Quinto argomento e sesto: 0x800000000 è un numero molto grande, "particolare", nella documentazione si fa riferimento a "CW_USEDEFAULT", vediamo se con "equate" lo trova... c'è corrispondenza, allora è lui.
- Settimo argomento ed ottavo argomento: sono dimensioni iniziali della finestra quando creata, solo valori veri.
- Nono argomento: 0 corrisponde al fatto che la finestra "sia figlia del desktop".
- Decimo argomento: per creare menù, noi non lo abbiamo, allora è 0.
- Undicesimo argomento: hIstance, posto a 0.
- Dodicesimo argomento: puntatore che vogliamo passare all'applicazione quando crea la finestra. C'è "EBX", impostato a riga 0040139d? NO, è **un altro percorso**. Usiamo Function Graph per vedere il flusso tra i blocchi di codice, e trovare quello realmente collegato alla nostra funzione.



Dobbiamo vedere invece dal codice che parte, quindi in realtà saliamo sopra.



La troviamo a riga *00401308*, e sta passando *hInstance*. (è una scelta del programmatore).
Andiamo avanti:
ShowWindow, sopra passiamo due parametri:

- uno per il riferimento alla finestra.
- il secondo come deve apparire tale finestra (massimizzata, minimizzata, EDX è costante, parametro passato a winmain).

La finestra non è ancora viva, non sappiamo come funziona! Come la gestisce Windows?
Se prendessi il decompiler vedrei la versione decompilata del programma in test.

Clò che abbiamo prodotto fino ad ora è:

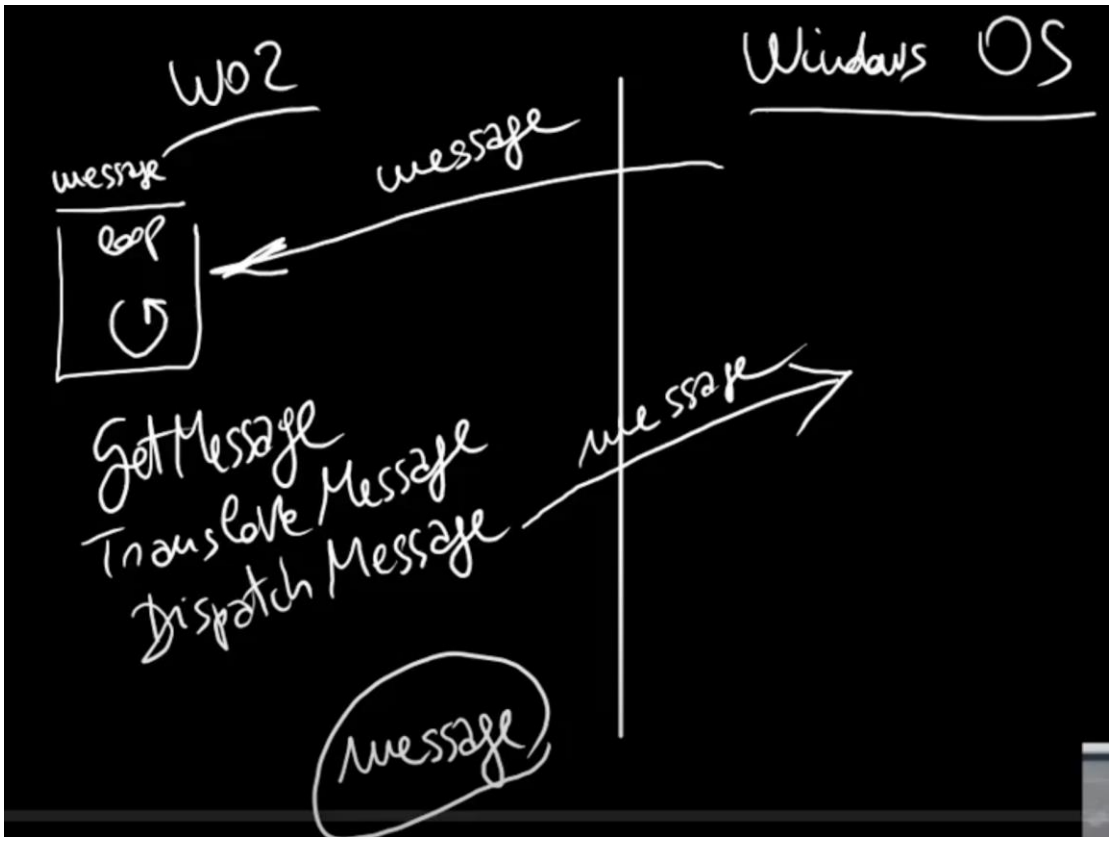
Decompile: WinMain - (W02.exe)

```
1
2  int WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nShowCmd)
3
4  {
5    ATOM AVar1;
6    HWND hWnd;
7    BOOL BVar2;
8    WNDCLASSEXA wclsex;
9    MSG local_2c;
10
11   wclsex.lpszClassName = s_WindowsApp_00402000;
12   wclsex.lpfnWndProc = (WNDPROC)&wndProc;
13   wclsex.style = CS_DBLCLKS;
14   wclsex.cbSize = 0x30;
15   wclsex.hInstance = hInstance;
16   wclsex.hIcon = LoadIconA((HINSTANCE)0x0,(LPCSTR)IDI_APPLICATION);
17   wclsex.hIconSm = LoadIconA((HINSTANCE)0x0,(LPCSTR)IDI_APPLICATION);
18   wclsex.hCursor = LoadCursorA((HINSTANCE)0x0,(LPCSTR)IDI_APPLICATION);
19   wclsex.lpszMenuName = (LPCSTR)0x0;
20   wclsex.cbClsExtra = 0;
21   wclsex.cbWndExtra = 0;
22   wclsex.hbrBackground = (HBRUSH)0x5;
23   AVar1 = RegisterClassExA(&wclsex);
24   if (AVar1 == 0) {
25     return 0;
26   }
27   hWnd = CreateWindowExA(0,s_WindowsApp_00402000,"Windows App",WS_TILEDWINDOW,CW_USEDEFAULT,
28                          CW_USEDEFAULT,0x220,0x177,(HWND)0x0,(HMENU)0x0,hInstance,(LPVOID)0x0);
29   ShowWindow(hWnd,nShowCmd);
30   while( true ) {
31     BVar2 = GetMessageA(&local_2c,(HWND)0x0,0,0);
32     if (BVar2 == 0) break;
33     TranslateMessage(&local_2c);
34     DispatchMessageA(&local_2c);
35   }
36   return local_2c.wParam;
37 }
```

Soffermiamoci sul codice while(true), esso si chiama **MessageLoop**, in cui si eseguono tre API:

- GetMessage,
- TranslateMessage,
- DispatchMessage,

il funzionamento è il seguente:



Quando il sistema operativo invia messaggi, il programma rimane in attesa, e quando lo riceve lo processa e poi tramite dispatch lo ripassa al sistema operativo. Questo dispatch che torna al sistema operativo dove va a finire? Alla **procedura della finestra**, la quale gestisce il messaggio.
C'è **una Windows Procedure** per ogni classe, e noi la specifichiamo nel secondo parametro, dove c'è **WindProc (**riga 12).
Il win main è globale per l'applicazione, con il dispatch eseguo la **win proc.**
Esaminiamola!
Questa winProc è chiamata dal sistema operativo, non c'è una call, anche se viene invocata dal sistema operativo. Ghidra la etichetta come label, ma è una funzione, quindi la creo su Ghidra (riga tra 00401464 e 00401467)
Cerchiamo sul manuale Win Proc, troviamo:
(https://learn.microsoft.com/en-us/windows/win32/winmsg/using-window-procedures):

```
LRESULT CALLBACK MainWndProc(
    HWND hwnd,        // handle to window
    UINT uMsg,        // message identifier
    WPARAM wParam,    // first message parameter
    LPARAM lParam)    // second message parameter
{

    switch (uMsg)
    {
        case WM_CREATE:
            // Initialize the window.
            return 0;

        case WM_PAINT:
            // Paint the window's client area.
            return 0;

        case WM_SIZE:
            // Set the size and position of the window.
            return 0;

        case WM_DESTROY:
            // Clean up window-specific data objects.
            return 0;

        //
        // Process other messages.
        //

        default:
            return DefWindowProc(hwnd, uMsg, wParam, lParam);
    }
    return 0;
}
```

Gli ultimi due parametri sono generici perché cambiano a seconda del messaggio che uso.
La winProc della mia applicazione deve solo intercettare i messaggi per i quali è interessata ad avere un comportamento che si discosta dal default.

```
LRESULT WndProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam)

{
  LRESULT LVar1;

  if (uMsg != 2) {
    LVar1 = DefWindowProcA();
    return LVar1;
  }
  PostQuitMessage(0);
  return 0;
}
```

Cosa è il messaggio 2? Vediamo con ”equate”, i messaggi iniziamo con WM... WM_DESTROY, ovvero chiudere finestra, diverso da chiudere applicazione (potrebbe lavorare in background).

```
EDX,dword ptr [EBP + uMsg]

EDX,WM_DESTROY
```

Qui quello che facciamo è infatti accedere a ”uMsg”, metterlo in EDX, e fare una compare con WM_DESTROY.

Attenzione: DefWIndowProcA() non ha alcun parametro, ma quando ho fatto JMP non ho tolto nessuno dei parametri passati, in quanto non ho fatto PUSH.
Nb: Perchè parliamo di Push e non di POP per togliere i parametri?
molto semplicemente, l'idea è che basta rimuovere spazio sullo stack, quindi
"abbassare il puntatore ESP" per non far vedere questi quattro parametri.
Nbb: Raramente vedremo l'operazione Pop, perché toglie un parametro alla volta, ciò che si fa è fare una add ad ESP, per abbassarlo di una certa quantità.
Ghidra non si è accorto di questo, non ha visto che non ho svuotato nulla; quindi,
in realtà sto passando i parametri passati da WndProc, cioè:

- hwnd è in Stack[0x4]
- uMsg è in Stack[0x8]
- wParam è in Stack[0xc]
- lParam è in Stack[0x10]

Prima di saltare a "*USER32.DLL::DefWindowProcA*", questi parametri sono presenti nello stack.
Possiamo allora modificare la funzione aggiungendo questi parametri.
Quando il messaggio è di Destroy, andrò a PostQuitMessage(0), usata dal thread per fare richiesta di terminare l'applicazione.
Nota: EBX è registro che viene preservato da quando entro a quando esco da una funzione.
Abbiamo ora una visione totale di ciò che fa il programma!