

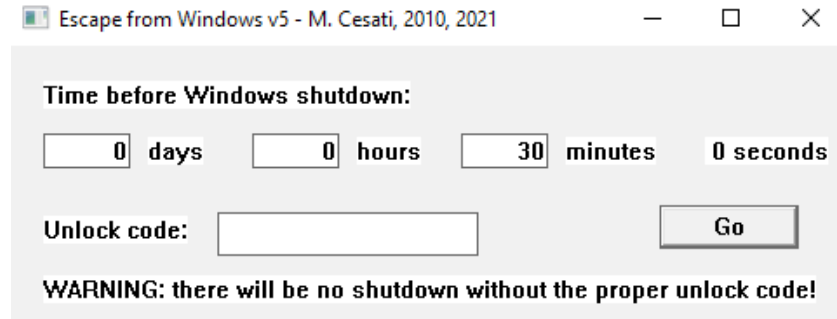
Malware Analysis Homework 3

Danilo Dell'Orco

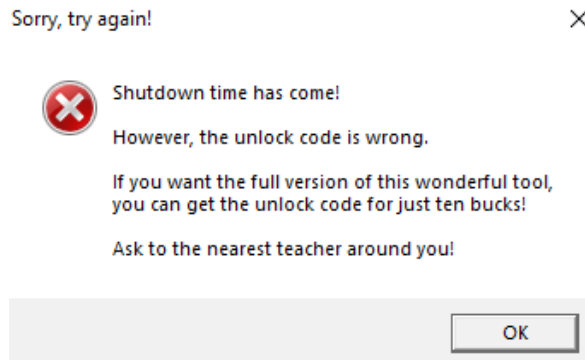
0300229

1 Introduzione

Per analizzare il comportamento del programma, lanciamo l'eseguibile su una macchina virtuale Windows 10. Vediamo che stiamo trattando un'applicazione grafica, che permette di schedare lo shutdown della macchina.



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



Il nostro obiettivo è quindi quello di analizzare l'eseguibile, cercando in particolare il *codice di sblocco* che rende funzionale il programma, patchando l'eseguibile dove necessario per bypassare i meccanismi di *antidebugging* presenti.

2 Analisi statica

Effettuiamo innanzitutto un'analisi statica delle stringhe, sfruttando lo strumento *Defined Strings* di Ghidra. Dalla lista, vediamo sono presenti le seguenti stringhe:

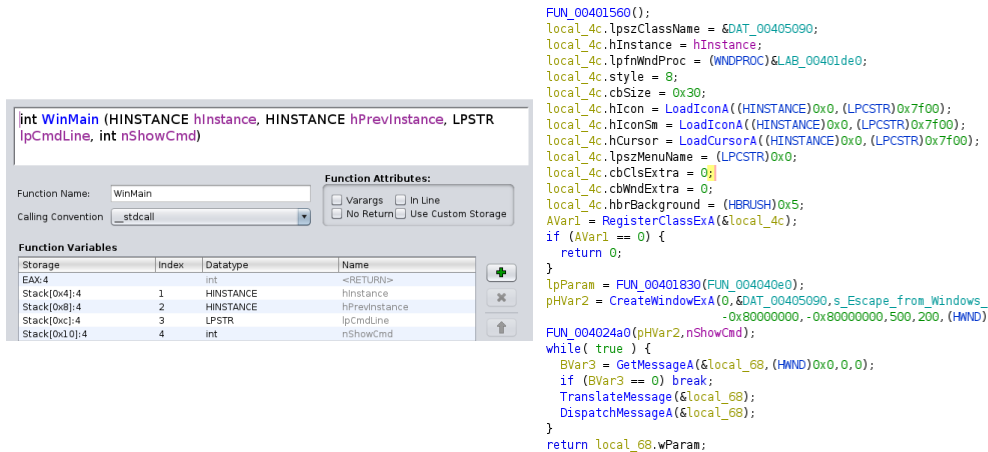
- *IsDebuggerPresent*
- *GetTickCount*
- *QueryPerformanceCounter*.

Questo ci suggerisce la presenza di meccanismi di *antidebugging* all'interno dell'eseguibile. Prima di proseguire con l'analisi dinamica sfruttando *OllyDbg*, è necessario quindi individuare dove questi meccanismi sono implementati, e bypassarli andando a patchare l'eseguibile in maniera opportuna.

3 Analisi del WinMain

Analizzando il codice possiamo trovare molte funzioni non riconosciute da Ghidra. Come abbiamo visto eseguendo il programma, questo presenta una GUI, per cui ci aspettiamo la presenza di un **WinMain**. La candidata più probabile è **FUN_004024e0**.

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



The screenshot shows the 'Edit Function' window in Ghidra for the function `WinMain`. The function signature is `int WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)`. The 'Function Variables' table lists the following variables:

Storage	Index	Datatype	Name
EAX:4		int	<RETURN>
Stack[0x4]:4	1	HINSTANCE	hInstance
Stack[0x8]:4	2	HINSTANCE	hPrevInstance
Stack[0xc]:4	3	LPSTR	lpCmdLine
Stack[0x10]:4	4	int	nShowCmd

The assembly code for the function is as follows:

```
FUN_00401560():
local_4c.lpszClassName = 6DAT_00405090;
local_4c.hInstance = hInstance;
local_4c.lpfWndProc = (WNDPROC)&LAB_00401de0;
local_4c.style = 8;
local_4c.cbSize = 0x30;
local_4c.hIcon = LoadIconA((HINSTANCE)0x0, (LPCSTR)0x7f00);
local_4c.hIconSm = LoadIconA((HINSTANCE)0x0, (LPCSTR)0x7f00);
local_4c.hCursor = LoadCursorA((HINSTANCE)0x0, (LPCSTR)0x7f00);
local_4c.lpszMenuName = (LPCSTR)0x0;
local_4c.cbClsExtra = 0;
local_4c.cbWndExtra = 0;
local_4c.hbrBackground = (HBRUSH)0x5;
AVar1 = RegisterClassExA(&local_4c);
if (AVar1 == 0) {
    return 0;
}
lpParam = FUN_00401830(FUN_004040e0);
pHVar2 = CreateWindowExA(0, 6DAT_00405090, s_Escape_from_Windows_
                        0x80000000, -0x80000000, 500, 200, (HWND)
FUN_004024a0(pHVar2, nShowCmd);
while( true ) {
    BVar3 = GetMessageA(&local_68, (HWND)0x0, 0, 0);
    if (BVar3 == 0) break;
    TranslateMessage(&local_68);
    DispatchMessageA(&local_68);
}
return local_68.wParam;
```

Analizzando il `WinMain`, vediamo che vengono utilizzate le funzioni *RegisterClassExA* e *CreateWindowExA* rispettivamente per creare ed istanziare

una classe finestra. Alla funzione *RegisterClassExA* viene passato come unico parametro il puntatore ad una struttura *WINDCLASSEXA*, inizializzata direttamente nel *WinMain*. Il campo *lpfnWndProc* di questa struttura è di tipo *WNDPROC*, e rappresenta un puntatore alla *window procedure*.

La Window Procedure viene interpretata erroneamente da Ghidra come una label, per cui andiamo a definire una nuova funzione a partire da tale label, denominandola *WindProc*. Analizzeremo questa funzione più avanti.

Esaminando le altre funzioni utilizzate nel main, vediamo che *FUN_00401830* inizializza una certa struttura dati. Analizzandone il codice osserviamo che la struttura è la stessa definita nel precedente *Homework*, per cui definiamo una nuova struttura *ProgStruct* analoga a quella già descritta in precedenza.

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

Rinominiamo *FUN_00401830* in *InitProgStruct*, e ne modifichiamo il tipo di ritorno in *ProgStruct**. Possiamo osservare che in questa funzione, oltre ad effettuare il setup della struttura, viene anche invocata *FUN_004016f0*; rimandiamo l'analisi di questa funzione ad un secondo momento.

```

2 ProgStruct * __cdecl InitProgStruct(void *param_1)
3
4 {
5     ProgStruct_00407020.timeElapsed = 0;
6     ProgStruct_00407020.updateInt = 1000;
7     ProgStruct_00407020.timerValue = 0x708;
8     FUN_00401530(ProgStruct_00407020.warningStr,0x80,
9         "WARNING: there will be no shutdown without the proper unlock code!");
10    FUN_00401530(ProgStruct_00407020.remainingSecStr,0x10," 0 seconds");
11    ProgStruct_00407020.timerFlag = 0;
12    ProgStruct_00407020.shutdownProc = param_1;
13    _DAT_004070ec = FUN_004016f0();
14    return &ProgStruct_00407020;
15 }

```

Come possiamo vedere dal *WinMain*, ad *InitProgStruct* viene passato in ingresso il parametro *DAT_004040e0*, che è un puntatore ad una funzione non riconosciuta correttamente da Ghidra. Effettuiamo quindi il Disassembling su *004040e0* e definiamo sulla nuova label la funzione *ShutdownProcedure*.

Questa funzione viene invocata allo scadere del timer, per cui implementa la logica sul controllo del codice di sblocco e sull'esecuzione effettiva dello shutdown. Analizziamo quindi il contenuto di *ShutdownProcedure* per cercare informazioni esplicite riguardo il codice di sblocco da utilizzare. Tuttavia, notiamo che a differenza dell'eseguibile *hw2.exe*, in questo caso Ghidra non riconosce molte funzioni, e non fornisce dunque informazioni utili riguardo il codice di sblocco.

Ciò significa che è necessario procedere con un'analisi più approfondita sfruttando il debugger per studiare nel dettaglio il comportamento a runtime del programma.

4 Analisi Dinamica

Utilizziamo *OllyDbg* per effettuare il debugging del programma. Sappiamo già dall'analisi statica che sono presenti dei meccanismi di *antidebugging*, per cui prima di procedere con l'analisi effettiva, è necessario bypassare i diversi meccanismi presenti nell'eseguibile.

A conferma di ciò, possiamo vedere che avviando il programma da *OllyDbg* non viene mostrata la finestra principale.



4.1 FUN_004024a0 – AntiDebugger

Cerchiamo innanzitutto la porzione di codice in cui è stato implementato il meccanismo di antidebugging. Analizzando più nel dettaglio il *WinMain*, vediamo che non è presente una chiamata esplicita a *ShowWindow()*, che viene invece invocata all'interno della funzione *FUN_004024a0*.

```
1 void __cdecl FUN_004024a0(HWND param_1,int param_2)
2 {
3     BOOL BVar1;
4     BVar1 = IsDebuggerPresent();
5     if (BVar1 == 0) {
6         ShowWindow(param_1,param_2);
7         return;
8     }
9     /* WARNING: Subroutine does not return */
10    ExitProcess(0);
11 }
```

Questa funzione internamente chiama *isDebuggerPresent()*, che ritorna 0 solo se non è in corso il debug del processo chiamante. Quindi eseguendo il programma con OllyDbg, *isDebuggerPresent* ritorna sempre 1 e non verrà eseguita la chiamata *ShowWindow()*.

Rinominiamo *FUN_004024a0* in *AntiDebugger()* e proseguiamo con il patching dell'eseguibile per bypassare questo meccanismo.

4.1.1 IsDebuggerPresent Patching

Da Ghidra, vediamo che la chiamata alla funzione *IsDebuggerPresent()* viene effettuata all'indirizzo *004024a3*. Spostandoci su *OllyDbg*, inseriamo quindi un breakpoint su tale indirizzo ed avviamo il programma.

004024A0	83EC 1C	SUB ESP,1C	
004024A3	FF15 44824000	CALL DWORD PTR DS:[<&KERNEL32.IsDebuggerPresent]	IsDebuggerPresent
004024A9	85C0	TEST EAX,EAX	
004024AB	75 1C	JNZ SHORT hw3_dbg_.004024C9	
004024AD	8B4424 24	MOV EAX,DWORD PTR SS:[ESP+24]	
004024B1	894424 04	MOV DWORD PTR SS:[ESP+4],EAX	
004024B5	8B4424 20	MOV EAX,DWORD PTR SS:[ESP+20]	
004024B9	890424	MOV DWORD PTR SS:[ESP],EAX	
004024BC	FF15 44834000	CALL DWORD PTR DS:[<&USER32.ShowWindow]	ShowWindow

Applichiamo una patch in questa porzione di codice, sostituendo le istruzioni *004024a3*, *004024a* e *004024ab* con delle *NOP* in modo da bypassare completamente sia la chiamata che il controllo su *IsDebuggerPresent*. Così facendo verrà sempre effettuata la chiamata a *ShowWindow()*.

004024A0	83EC 1C	SUB ESP,1C	
004024A3	90	NOP	
004024A4	90	NOP	
004024A5	90	NOP	
004024A6	90	NOP	
004024A7	90	NOP	
004024A8	90	NOP	
004024A9	90	NOP	
004024AA	90	NOP	
004024AB	90	NOP	
004024AC	90	NOP	
004024AD	8B4424 24	MOV EAX,DWORD PTR SS:[ESP+24]	
004024B1	894424 04	MOV DWORD PTR SS:[ESP+4],EAX	
004024B5	8B4424 20	MOV EAX,DWORD PTR SS:[ESP+20]	
004024B9	890424	MOV DWORD PTR SS:[ESP],EAX	
004024BC	FF15 44834000	CALL DWORD PTR DS:[<&USER32.ShowWindow]	ShowWindow

Inseriamo un breakpoint su *004024bc* e verifichiamo a runtime il comportamento del programma. Vediamo che la finestra non viene mostrata nonostante *ShowWindow()* venga effettivamente eseguita.

ShowWindow ritorna infatti *ERROR_INVALID_WINDOW_HANDLE*, per cui non viene correttamente passato al programma l'handle della finestra creata.

004024A7	90	NOP	
004024A8	90	NOP	
004024A9	90	NOP	
004024AA	90	NOP	
004024AB	90	NOP	
004024AC	90	NOP	
004024AD	8B4424 24	MOV EAX,DWORD PTR SS:[ESP+24]	
004024B1	894424 04	MOV DWORD PTR SS:[ESP+4],EAX	
004024B5	8B4424 20	MOV EAX,DWORD PTR SS:[ESP+20]	
004024B9	890424	MOV DWORD PTR SS:[ESP],EAX	
004024BC	FF15 44834000	CALL DWORD PTR DS:[<&USER32.ShowWindow]	ShowWindow
004024C4	83EC 08	SUB ESP,8	
004024C5	83C4 1C	ADD ESP,1C	

Registers (FPU)		
EIP	004024C2	hw3_dbg_.004024C2
C 0	ES 002B 32bit	0(FFFFFFFF)
F 1	CS 0023 32bit	0(FFFFFFFF)
A 0	SS 002B 32bit	0(FFFFFFFF)
Z 1	DS 002B 32bit	0(FFFFFFFF)
S 0	FS 0053 32bit	3D1000(FFF)
T 0	GS 002B 32bit	0(FFFFFFFF)
D 0		
O 0	LastErr	ERROR_INVALID_WINDOW_HANDLE (00000578)
EFL	00000246	(NO,NB,E,BE,NS,PE,GE,LE)

Analizzando la memoria, possiamo vedere che l'errore è causato dal fatto che stiamo passando a *ShowWindow* un parametro *hWnd* nullo.

0060FDEC	00000000	
0060FDF0	00000000	hWnd = NULL
0060FDF4	0000000A	ShowState = SW_SHOWDEFAULT
0060FDF8	00000000	

Questo ci suggerisce come siano presenti anche altri meccanismi di *antidebug*, per cui è necessario proseguire con l'analisi dell'eseguibile.

4.2 FUN_00401dc0 – AntiDbgPEB

Andiamo ad analizzare la *Window Procedure*, e vediamo che viene chiamata la funzione *FUN_00401dc0*, passando come parametro il puntatore al messaggio *uMsg* attualmente in gestione.

```
41 | lParam_00 = lParam;
42 | LVar1 = GetWindowLongA(hwnd, -0x15);
43 | FUN_00401dc0((int *)&uMsg);
```

Esaminando questa funzione, vediamo che viene effettuato un accesso diretto al *Process Environment Block*, ovvero una struttura del Sistema Operativo, che mantiene diverse informazioni riguardo il processo in esecuzione. Un'informazione rilevante nella nostra analisi è quella contenuta nel secondo campo *BeingDebugged*, che specifica tramite un byte la presenza di un debugger. Questa struttura viene mantenuta all'offset 48 del registro di segmento *FS*. Il secondo campo del PEB (*BeingDebugged*) sarà quindi mantenuto in *FS:[0x30+2]*.

In *FUN_00401dc0* viene copiato nel registro *EDX* l'indirizzo base della PEB e poi si copia sempre in *EDX* il valore del campo *BeingDebugged* [*EDX+2*].

```

                                undefined __cdecl AntiDbgPEB(int * uMsg)
                                AL:1 <RETURN>
                                Stack[0x4]:4 uMsg
                                FS_OFFSET:4 in_FS_OFFSET
                                XREF[1]: 00401dc0(R)
                                AntiDbgPEB
                                XREF[1]: WindProc:00401e1a(c)
00401dc0 8b 44 24 04 MOV EAX,dword ptr [ESP + uMsg]
00401dc4 64 8b 15 MOV EDX,dword ptr FS:[0x30]
                                30 00 00 00
00401dcb 8b 52 02 MOV EDX,dword ptr [EDX + 2]
```

Successivamente si esegue l'*AND* tra il contenuto di *EDX* (quindi *BeingDebugged*) ed il valore 7. Lo *ZeroFlag* viene settato a 1 se il risultato dell'*AND* è 0, altrimenti non viene settato. Tramite *SETNZ* si imposta il byte sul registro *DL* soltanto se lo *ZeroFlag* vale 0.

```
00401dce 83 e2 07 AND EDX,7
00401dd1 0f 95 c2 SETNZ DL
```

Quindi se il debugger è attivo, si avrà in *DL* il valore 1:

```
EDX: BeingDebug = 0001
EDX: 1111 ^ 0001 = 0001 ==> ZF = 0
SETNZ DL ==> DL = 1
```

Invece, se non è attivo nessun processo di debugging, si avrà in *DL* il valore 0:

```
EDX: BeingDebug = 0000
EDX: 1111 ^ 0000 = 0000 ==> ZF = 1
SETNZ DL ==> DL = 0
```

Tramite *MOVZX* viene copiato il contenuto di *DL* (1 byte) in *EDX* (4 byte) riempiendo i restanti 24 bit più significativi con tutti 0. Il valore viene poi sommato al valore puntato da *EAX*, quindi il messaggio *uMsg* passato come parametro.

```
00401dd4 0f b6 d2      MOVZX     EDX,DL
00401dd7 01 10         ADD      dword ptr [EAX],EDX
```

In definitiva questa funzione, nel caso in cui sia attivo il debugger, incrementa di 1 il codice numerico del messaggio gestito dalla *WindProc*. Ciò significa, ad esempio, che quando viene processato il messaggio *WM_CREATE* (*uMsg=1*) per la creazione della finestra, questo verrà incrementato di 1 e diventerà *WM_DESTROY* (*uMsg=2*). Di fatto quindi se utilizziamo un debugger, non verrà mai gestita la creazione della finestra, e per questo motivo *CreateWindowExA* ritorna 0 in *EAX*.

Registers (FPU)	
EAX	00000000
ECX	00640000
EDX	00640000
EBX	74C7D790 USER32.LoadIconA
ESP	0060FE40
EBP	0060FEC8
ESI	00400000 h*3.00400000
EDI	00BB0D5C

Questo meccanismo di *antidebugging* è dunque il motivo per cui al momento della chiamata *ShowWindow* viene passato un handle nullo alla finestra. Rinominiamo la funzione *FUN_00401dc0* in *AntiDbgPEB* e proseguiamo con il patching dell'eseguibile per bypassare questo controllo.

4.2.1 ProcessEnvironmentBlock Patching

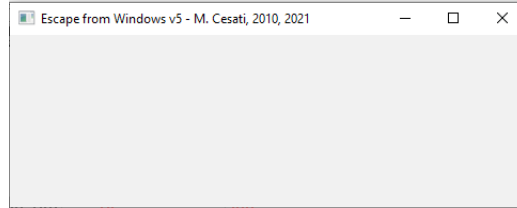
Dobbiamo rimuovere dal programma la logica in cui si verifica il campo *BeingDebugged* per incrementare il valore del messaggio gestito. Vediamo che la chiamata alla funzione *AntiDbgPEB()* viene effettuata all'indirizzo *00401e1a*.

```
00401e17 89 04 24      MOV      dword ptr [ESP]=>local_ec,EAX
00401e1a e8 al ff      CALL     AntiDbgPEB
          ff ff
```

Sfruttando *OllyDbg* bypassiamo questo meccanismo di *antidebug* andando a rimuovere l'istruzione *CALL AntiDbgPEB*, inserendo al suo posto dei *No-Operation*.

00401E17	. 890424	MOV DWORD PTR SS:[ESP],EAX
00401E1A	90	NOP
00401E1B	90	NOP
00401E1C	90	NOP
00401E1D	90	NOP
00401E1E	90	NOP
00401E1F	. 8B8424 F40000	MOV EAX,DWORD PTR SS:[ESP+F4]

Verifichiamo il comportamento a runtime dopo aver patchato questa istruzione. Vediamo che viene mostrata la finestra principale del programma, ma prima di creare le finestre relative ai *Button* e agli *EditBox* il programma v  in crash.



Poich  eseguendo il programma normalmente la finestra viene mostrata correttamente, abbiamo almeno un altro meccanismo di *antidebug* all'interno del codice.

4.3 FUN_004016f0 - GetOutputDebugString

Continuiamo l'analisi con la funzione *FUN_004016f0* precedentemente incontrata in *InitProgStruct*. Questa internamente chiama la funzione di libreria *LoadLibraryA* che permette di caricare il modulo specificato in input all'interno dell'address space del processo chiamante.

```

5 void FUN_004016f0(void)
6
7 {
29     CStack45 = 'k';
30     uStack41 = 0x65;
31     uStack44 = 0x65;
32     uStack43 = 0x72;
33     uStack42 = 0x6e;
34     uStack40 = 0x6c;
35     uStack39 = 0x33;
36     uStack38 = 0x32;
37     uStack37 = 0x2e;
38     uStack36 = 100;
39     uStack34 = 0x6c;
40     uStack35 = 0x6c;
41     uStack33 = 0;
42     hModule = LoadLibraryA(&CStack45);

```

Risulta evidente che le variabili *undefined* presenti in questa funzione rappresentino un *array di caratteri*, che specificano il nome del modulo da caricare. Ridefiniamo quindi *CStack45* da *CHAR* in *CHAR[19]* ed analizziamo nuovamente il codice disassemblato.

```

11     CStack45[0] = 'k';
12     CStack45[4] = 'e';
13     CStack45[11] = 'e';
14     CStack45[2] = 'r';
15     CStack45[3] = 'n';
16     CStack45[5] = 'l';
17     CStack45[6] = '3';
18     CStack45[7] = '2';
19     CStack45[8] = '.';
20     CStack45[9] = 'd';
21     CStack45[11] = 'l';
22     CStack45[10] = 'l';
23     CStack45[12] = '\0';
24     hModule = LoadLibraryA(CStack45);

```

La variabile *CStack45* contiene la stringa *'kernel32.dll'*, ovvero il modulo che verrà caricato in memoria tramite *LoadLibraryA*. Successivamente viene chiamata la funzione *GetProcAddress* che ritorna l'indirizzo di una funzione o di una variabile della libreria *dll* specificata.

I parametri passati sono la libreria *kernel32.dll* caricata precedentemente in memoria, e la funzione *'OutputDebugStringA'*, che invia una stringa al debugger.

```

25 | CStack45[0] = '0';
26 | CStack45[9] = 'u';
27 | CStack45[4] = 'u';
28 | CStack45[1] = 'u';
29 | CStack45[12] = 't';
30 | CStack45[5] = 't';
31 | CStack45[2] = 't';
32 | CStack45[3] = 'p';
33 | CStack45[6] = 'D';
34 | CStack45[7] = 'e';
35 | CStack45[8] = 'b';
36 | CStack45[16] = 'g';
37 | CStack45[10] = 'g';
38 | CStack45[11] = 'S';
39 | CStack45[13] = 'r';
40 | CStack45[14] = 'i';
41 | CStack45[15] = 'n';
42 | CStack45[17] = 'A';
43 | CStack45[18] = '\0';
44 | GetProcAddress(hModule, CStack45);
45 | return;

```

Dalla documentazione vediamo che *GetProcAddress* ritorna un *FARPROC*, ovvero un puntatore all'indirizzo in cui viene caricata la funzione specificata. Analizzando inoltre *InitProgStruct* vediamo che il valore ritornato da *FUN_004016f0* viene assegnato alla variabile globale *__DAT_004070ec*. Ciò significa che in realtà *FUN_004016f0* non ritorna un *void* ma un *FARPROC*.

Andiamo quindi a rinominare *FUN_004016f0* in *GetOutputDebugString* modificando il valore di ritorno in *FARPROC*.

```

5 | FARPROC GetOutputDebugString(void)
6 |
7 | {
    ...
45 | outputDebugAddr = GetProcAddress(hModule, CStack45);
46 | return outputDebugAddr;
47 | }

```

Rinominiamo inoltre la variabile globale *DAT_004070ec* in *OutputDebugStringPtr*.

```

13 | OutputDebugStringPtr = GetOutputDebugString();
14 | return &ProgStruct_00407020;

```

Proseguiamo quindi l'analisi con OllyDbg, andando a patchare questo meccanismo di antidebug.

4.3.1 OutputDebugString Patching

Sappiamo che all'indirizzo *004070ec* è salvato il puntatore alla funzione *OutputDebugString*. Dobbiamo quindi ricavare l'istruzione in cui questo indirizzo è acceduto in modo da poter patchare l'istruzione effettiva che invoca *OutputDebugString*. Ghidra in tal senso non fornisce nessuna informazione utile, in quanto non rileva nessun accesso in lettura su *004070ec*.

Sfruttiamo quindi *OllyDbg* per intercettare questa chiamata. Inseriamo un breakpoint all'indirizzo *0040189c*, ed analizziamo l'evoluzione della memoria e dei registri.

0040189C <GetOutputDebugString>	E8 4FFFFFFF	CALL hw3_dbg.004016F0
004018A1	A3 EC704000	MOV DWORD PTR DS:[4070EC],EAX
004018A6	B8 20704000	MOV EAX,hw3_dbg.00407020
004018AB	83C4 1C	ADD ESP,1C

GetOutputDebugString ritorna in *EAX*, e vediamo che *EAX* contiene effettivamente l'indirizzo di *OutputDebugString*.

Registers (FPU)	<	<	<	<
EAX 754F9DA0 JMP to KERNELBA.OutputDebugStringA				
ECX E8BDD139				

Successivamente il contenuto di *EAX* viene copiato all'indirizzo *004070ec*.

Address	Hex dump
004070EC	A0 9D 2B 75 00 00 00 00 00 00 00 00 00 00 00 00
004070FC	00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00

Individuiamo quindi l'indirizzo *004070ec* in memoria, ed inseriamo un *Breakpoint sul Memory Access*. In questo modo quando verrà effettuato un qualsiasi accesso in lettura/scrittura su questa locazione, verrà messo in pausa il flusso di esecuzione.

Avviamo il programma, e vediamo che in lettura *004070ec* viene acceduto all'indirizzo *0040504a*. Qui viene effettuata la call di *OutputDebugString*.

00405046	8B4424 50	MOV EAX,DWORD PTR SS:[ESP+50]	
0040504A	FF90 CC000000	CALL DWORD PTR DS:[EAX+CC]	KERNEL32.OutputDebugStringA

Proseguendo con l'istruzione successiva su *OllyDbg* il programma termina, e ciò conferma che effettivamente il meccanismo di *antidebug* consiste nella chiamata ad *OutputDebugString*. Tuttavia questa chiamata viene effettuata fuori dalla *code section*, e non possiamo patchare direttamente l'istruzione in questo punto.

Vediamo che la funzione in cui viene effettuata la call ad *OutputDebugString* inizia all'indirizzo *00405020*.

00405020	83EC 4C	SUB ESP,4C	
00405023	31C0	XOR EAX,EAX	
00405025	8D76 00	LEA ESI,DWORD PTR DS:[ESI]	
00405028	C64404 1F 25	MOV BYTE PTR SS:[ESP+EAX+1F],25	
0040502D	C64404 20 73	MOV BYTE PTR SS:[ESP+EAX+20],73	
00405032	83C0 02	ADD EAX,2	
00405035	83F8 20	CMP EAX,20	
00405038	75 EE	JNZ SHORT hw3_dbg.00405028	
0040503A	8D4424 1F	LEA EAX,DWORD PTR SS:[ESP+1F]	
0040503E	C64424 3F 00	MOV BYTE PTR SS:[ESP+3F],0	
00405043	890424	MOV DWORD PTR SS:[ESP],EAX	
00405046	8B4424 50	MOV EAX,DWORD PTR SS:[ESP+50]	
0040504A	FF90 CC000000	CALL DWORD PTR DS:[EAX+CC]	KERNEL32.OutputDebugStringA

Sfruttando Ghidra, andiamo quindi all'indirizzo *00405020* e vediamo che è presente una label.

```

LAB_00405020
00405020 49      DEC     ECX
00405021 fc      CLD
00405022 7a 10   JP      LAB_0040502f+5
XREF[3]: 00404010(R), 0040401f(W), 00404032(j)

```

Definiamo una nuova funzione a partire da questo indirizzo e sfruttiamo il **Function Call Graph** per individuare in quale porzione di codice viene effettivamente invocata *00405020*, e quindi anche *GetOutputDebugString*. Vediamo che *00405020* viene invocata da *00404000*, chiamata a sua volta all'interno della Window Procedure.



In particolare l'istruzione *CALL FUN_00404000* viene eseguita all'indirizzo *004020f0* durante la gestione del messaggio *WM_SIZE* (uMsg=5).

```

004020ed 89 1c 24   MOV     dword ptr [ESP]=local_ec,EBX
004020f0 e8 0b 1f   CALL   FUN_00404000
00 00
004020f5 81 c4 dc   ADD     ESP,0xdc
00 00 00

```

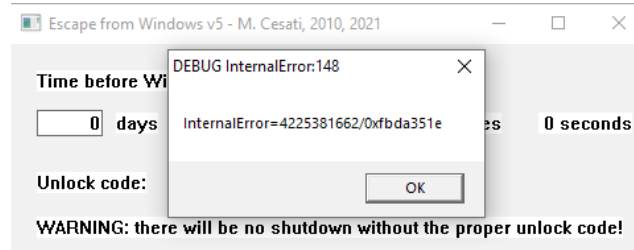
Analizzando *FUN_00404000*, vediamo che vengono eseguite una serie di operazioni di XOR e ROR verso i byte presenti in memoria a partire dall'indirizzo *00405020*, per poi effettuare un *JMP 00405020*. Di fatto il codice effettivo di *00405020* sarà visibile solo a runtime.

00404000	8B4C24 04	MOV ECX,DWORD PTR SS:[ESP+4]
00404004	31D2	XOR EDX,EDX
00404006	8DB426 00000000	LEA ESI,DWORD PTR DS:[ESI]
0040400D	8D76 00	LEA ESI,DWORD PTR DS:[ESI]
00404010	8B0495 20504000	MOV EAX,DWORD PTR DS:[EDX*4+405020]
00404017	35 2BFAA389	XOR EAX,89A3FA2B
0040401C	C1C8 09	ROR EAX,9
0040401F	890495 20504000	MOV DWORD PTR DS:[EDX*4+405020],EAX
00404026	83C2 01	ADD EDX,1
00404029	83FA 0E	CMP EDX,0E
0040402C	75 E2	JNZ SHORT hw3_dbg.00404010
0040402E	894C24 04	MOV DWORD PTR SS:[ESP+4],ECX
00404032	E9 E90F0000	JMP hw3_dbg.00405020

Possiamo quindi patchare questo meccanismo semplicemente inserendo un NOP all'indirizzo *004020f0* dove si effettua la call verso *00404000*.

004020ED	891C24	MOV DWORD PTR SS:[ESP],EBX
004020F0	E8 0B1F0000	CALL hw3_dbg.00404000
004020F5	81C4 DC000000	ADD ESP,0DC
004020FB	31C0	XOR EAX,EAX
004020ED	891C24	MOV DWORD PTR SS:[ESP],EBX
004020F0	90	NOP
004020F1	90	NOP
004020F2	90	NOP
004020F3	90	NOP
004020F4	90	NOP

Esportiamo l'eseguibile per tenere in modo permanente le patch applicate, ed eseguiamolo con *OllyDbg*. Vediamo che la finestra viene mostrata correttamente, ma viene mostrato un messaggio di errore, al seguito del quale l'applicazione viene chiusa.



E' presente quindi un altro meccanismo di *antidebugging* all'interno del codice, per cui proseguiamo con l'analisi.

4.4 InternalError patching

Sappiamo che l'errore è *InternalError*, per cui andiamo a cercare dei riferimenti a questa stringa analizzando su Ghidra le funzioni finora non considerate. Una di queste è la *Timer Procedure*, passata come parametro all'API *SetTimer*. Al suo interno notiamo la funzione *FUN_004042a0*, chiamata all'indirizzo *0040a121*.

```

00401a21 20 70 40 00          CALL     FUN_004042a0
          e8 7a 28          00 00

```

Analizzando questa funzione vediamo che viene effettivamente utilizzata la stringa *InternalError* come parametro di una funzione non riconosciuta da Ghidra.

```

39 | func_0x004026a0(aCStack270,0x80,"%s=%lu/0x%x", "InternalError",uVar4,u
40 | func_0x004026a0(aCStack142,0x80,"DEBUG %s:%d", "InternalError",0x94);

```

Ci spostiamo quindi su *OllyDbg*, inseriamo un breakpoint su *00401a21* dove viene invocata la funzione e procediamo con l'analisi a runtime. La funzione *00401a21* viene invocata due volte; la prima volta chiama *MessageBoxA* per mostrare il messaggio di errore.

```

00404399 C70424 00000000 MOV DWORD PTR SS:[ESP],0
004043A0 FF15 1C834000 CALL DWORD PTR DS:[<&USER32.MessageBoxA,USER32.MessageBoxA

```

La seconda volta chiama *ExitProcess* per terminare il programma.

```

0040431F C70424 00000000 MOV DWORD PTR SS:[ESP],0
00404326 FF15 10824000 CALL DWORD PTR DS:[<&KERNEL32.ExitProcess,KERNEL32.ExitProcess

```

Bypassiamo quindi questo meccanismo inserendo un NOP su *00401a21* dove viene chiamata *00401a21*.

```

00401A1A > C70424 20704000 MOV DWORD PTR SS:[ESP],h\3_dbg.00407020
00401A21 90 NOP
00401A22 90 NOP
00401A23 90 NOP
00401A24 90 NOP
00401A25 90 NOP
00401A26 . 83C4 14 ADD ESP,14

```

Esportiamo il nuovo eseguibile patchato, e verifichiamo che eseguendolo non viene più mostrato il *MessageBox* relativo a *InternalError*.

Abbiamo quindi terminato il patching dell'eseguibile, bypassando tutti i meccanismi di *antidebugging* presenti al suo interno. Possiamo dunque caricare l'eseguibile patchato su *OllyDbg* e proseguire con la ricerca effettiva del codice di sblocco.

5 Codice di Sblocco

Per individuare il codice di sblocco, è necessario innanzitutto localizzare la porzione di programma in cui è implementata la logica di verifica di tale codice. Sappiamo già che il controllo viene effettuato all'interno della procedura di shutdown definita in precedenza; tuttavia come abbiamo già visto le funzioni e le istruzioni utilizzate al suo interno non vengono riconosciute correttamente da Ghidra.

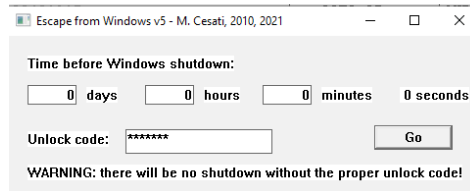
Proseguiamo quindi analizzando il comportamento a runtime della *Shutdown Procedure*. Da Ghidra vediamo che questa viene invocata all'indirizzo *00401a0f*.

```
00401a0f ff 15 34      CALL     dword ptr [ProgStruct_00407020.shutdownProc]
```

Inseriamo quindi un breakpoint su questo indirizzo e lanciamo il programma.

```
00401a08 > C70424 20704000 MOV DWORD PTR SS:[ESP],hw3_patc.00407020
00401a0f . FF15 34704000 CALL DWORD PTR DS:[407034]
```

Viene mostrata correttamente la finestra; inseriamo un timeout di 0 minuti, inseriamo il codice “*my_code*” e premiamo il pulsante “*Go*”



Una volta raggiunto il breakpoint, il programma entra in pausa. Continuiamo l'esecuzione analizzando la procedura di shutdown, tramite *Step Into (F7)*.

Vediamo che all'indirizzo *004041f7* viene chiamata l'API *GetDlgItemTextA*, utilizzata per prendere il codice inserito nell'*EditBox* e copiarlo nel registro di destinazione ESI, passato come terzo parametro. Il valore di ritorno di questa funzione è la lunghezza del codice inserito, che viene scritta invece in EAX.

```
004041dc 8d7424 42      LEA ESI,DWORD PTR SS:[ESP+42]
004041e0 c74424 0c 1e0001 MOV DWORD PTR SS:[ESP+C],1E
004041e8 897424 08      MOV DWORD PTR SS:[ESP+8],ESI
004041ec c74424 04 050001 MOV DWORD PTR SS:[ESP+4],5
004041f4 893c24        MOV DWORD PTR SS:[ESP],EDI
004041f7 <get inserted code> CALL DWORD PTR DS:[<USER32.GetDlgItemTextA>]
004041fd 8b15 04714000 MOV EDX,DWORD PTR DS:[407104]
00404203 83ec 10        SUB ESP,10
```

Analizzando il contenuto dei registri possiamo verificare che in ESI viene caricato il codice da noi inserito (*my_code*), ed in EAX la sua lunghezza (7)

Registers (FPU)	
EAX	00000007
ECX	766A9766 USER32.766A9766
EDX	00000007
EBX	766AFB10 USER32.PostQuitMessage
ESP	0060FC14
EBP	0060FCB8
ESI	0060FC46 ASCII "my_code"
EDI	00150046
EIP	004041FD hw3_dbg.004041FD

Successivamente, vediamo che viene eseguito un ciclo con 52 iterazioni dove vengono effettuati XOR e ROR dei byte contenuti a partire dall'indirizzo *004050c0*. Possiamo osservare che all'indirizzo *0040423e* si esegue la CALL verso *004050c0*, per cui all'interno di questo ciclo viene modificato il codice della funzione che viene invocata.

00404210 <start_loop>	8B148D C0504000	MOV EDI,DWORD PTR DS:[ECX*4+4050C0]
00404217	81F2 2BFAA389	XOR EDI,89A3FA2B
0040421D	C1CA 09	ROR EDI,9
00404220	89148D C0504000	MOV DWORD PTR DS:[ECX*4+4050C0],EDI
00404227	83C1 01	ADD ECX,1
0040422A	83F9 34	CMP ECX,34
0040422D <end_loop>	75 E1	JNZ SHORT <hw3_patc.start_loop>
0040422F	C74424 08 40404000	MOV DWORD PTR SS:[ESP+8],hw3_patc.00404040
00404237	894424 04	MOV DWORD PTR SS:[ESP+4],EAX
0040423B	893424	MOV DWORD PTR SS:[ESP],ESI
0040423E	E8 7D0E0000	CALL hw3_patc.004050C0
00404243	A1 84714000	MOV EAX,DWORD PTR DS:[407104]
00404248	85C0	TEST EAX,EAX
0040424A <jump_error_code>	74 02	JE SHORT hw3_patc.0040424E

A conferma di ciò, analizziamo la funzione all'indirizzo *004050c0*, prima e dopo l'esecuzione del ciclo for. Prima di entrare nel ciclo vediamo che non viene eseguita nessuna particolare operazione per la verifica del codice.

004050C0 <4050c0>	69FD 7A802B3A	IMUL EDI,EBP,3A2BB07A
004050C6	E2 08	LOOPD SHORT hw3_dbg.004050D0
004050C8	68EC 0A	IMUL EBP,ESP,0A
004050CB	C12F F0	SHR DWORD PTR DS:[EDI],0F0
004050CE	2260 22	AND AH,BYTE PTR DS:[EAX+22]
004050D1	2A6407 75	SUB AH,BYTE PTR DS:[EDI+EAX+75]
004050D5	B2 DD	MOV DL,0DD
004050D7	D963 B0	FILDENV (28-BYTE) PTR DS:[EBX-50]
004050DA	2C 00	SUB AL,0
004050DC	51	PUSH ECX
004050DD	F2	PREFIX REPNE

Invece, dopo aver eseguito completamente il ciclo, entrando nella CALL *004050c0* possiamo vedere come vengano eseguite diverse operazioni di XOR e CMP, proprio per la verifica dei singoli caratteri del codice di sblocco. Inseriamo quindi una label *check_code* su *004050c0*.

Analizziamo ora la chiamata alla funzione *check_code*.

0040422F	C74424 08 40404000	MOV DWORD PTR SS:[ESP+8],hw3_patc.00404040
00404237	894424 04	MOV DWORD PTR SS:[ESP+4],EAX
0040423B	893424	MOV DWORD PTR SS:[ESP],ESI
0040423E <call_check_code>	E8 7D0E0000	CALL <hw3_patc.check_code>

Possiamo vedere che vengono passati 3 parametri: ESI, quindi il codice inserito. EAX, quindi la lunghezza del codice inserito. Puntatore a *00404040*, che contiene una serie di byte.

Address	Hex dump	ASCII
00404040	83 EC 1C A1 04 71 40 00 85 C0 74 02 9A 42 C7 44	iiiiqq. iAt iBCD
00404050	24 04 00 00 00 00 C7 04 24 05 00 00 00 FF 15 F8	\$. . . C\$\$. . . y\$
00404060	82 40 00 A1 04 71 40 00 83 EC 08 85 C0 74 02 9A	i. . . ii iAt i
00404070	42 C7 04 24 FF FF FF FF FF 15 5C 82 40 00 A1 04	Bc\$yyyy\i. i
00404080	71 40 00 83 EC 04 85 C0 74 02 9A 42 83 C4 1C C3	q. i iAt iBiAA
00404090	56 31 D2 53 8B 4C 24 0C 8B 5C 24 10 8B 74 24 14	VtQSIL\$. iN\$ii\$

Andiamo quindi ad eseguire passo passo *check_code* con *OllyDbg*. Viene innanzitutto copiato il puntatore al codice da noi inserito sul registro EDX.

004050C0 <check_code>	83EC 1C	SUB ESP,1C	^ Registers (FPU)
004050C3	A1 E0A04000	MOV EAX,DWORD PTR DS:[40A0E0]	EAX 00000000
004050C8	8B5424 20	MOV EDX,DWORD PTR SS:[ESP+20]	ECX 00000034
004050CC	85C0	TEST EAX,EAX	EDX 0060FC46 ASCII "mv code"

Vengono poi copiati 9 bytes nell'indirizzo *0060fbe4* puntato dallo stack pointer.

004050D2	C70424 3F282FA5	MOV DWORD PTR SS:[ESP],A52F283F
004050D9	C74424 04 5D473D4F	MOV DWORD PTR SS:[ESP+4],4F3D475D
004050E1	C74424 08 3F000000	MOV DWORD PTR SS:[ESP+8],3F
0060FBE4	A52F283F	
0060FBE8	4F3D475D	
0060FBE9	0000003F	

Successivamente si verifica se il codice inserito ha lunghezza pari a 9, tramite una operazione di CMP. Se è stato inserito un codice di lunghezza diversa da 9 non viene eseguito il jump a *004050ff*, e si ritorna immediatamente al chiamante.

004050F4 <check_length>	837C24 24 09	CMP DWORD PTR SS:[ESP+24],9
004050F9	74 04	JE SHORT <hw3_patc.check_bytes>
004050FB	83C4 1C	ADD ESP,1C
004050FE	C3	RETN
004050FF <check_bytes>	0FB60424	MOVZX EAX,BYTE PTR SS:[ESP]

Avendo inserito *my_code* di lunghezza pari a 7, andiamo a patchare l'istruzione di jump sostituendo *JE* con *JNZ*; in questo modo possiamo saltare a *004050ff* dove inizia il check effettivo sul codice.

004050F4	837C24 24 09	CMP DWORD PTR SS:[ESP+24],9	^ Registers (FPU)
004050F9	75 04	JNZ SHORT hw3_patc.004050FF	EAX 00000000
004050FB	83C4 1C	ADD ESP,1C	ECX 00000034
004050FE	C3	RETN	EDX 0060FC46 ASCII "my_code"
004050FF	0FB60424	MOVZX EAX,BYTE PTR SS:[ESP]	EBX 750AFB10 USER32.PostQuitMessage
00405103	3202	XOR AL,BYTE PTR DS:[EDX]	ESP 0060FBE4

Il codice viene verificato controllando uno per volta i 9 byte che lo compongono. Considerando il *j-esimo* byte del codice di sblocco, vengono eseguite le seguenti operazioni:

- Si copia in EAX il *j-esimo* byte tra quelli copiati in precedenza su *0060fbe4* (*ESP+j*).
- Si effettua lo XOR tra EAX ed il *j-esimo* carattere del codice da noi inserito (*EDX+j*).
- Si verifica con un CMP che il risultato dello XOR sia uguale ad un certo byte atteso.

004050FF	0FB60424	MOVZX EAX,BYTE PTR SS:[ESP]	00405132	3242 04	XOR AL,BYTE PTR DS:[EDX+1]
00405103	3202	XOR AL,BYTE PTR DS:[EDX]	00405135	3C 2E	CMP AL,2E
00405105	3C 0C	CMP AL,0C	00405137	75 C2	JNZ SHORT hw3_patc.004050FB
00405107	75 F2	JNZ SHORT hw3_patc.004050FB	00405139	0FB64424 05	MOVZX EAX,BYTE PTR SS:[ESP+5]
00405109	0FB64424 01	MOVZX EAX,BYTE PTR SS:[ESP+1]	0040513E	3242 05	XOR AL,BYTE PTR DS:[EDX+5]
0040510E	3242 01	XOR AL,BYTE PTR DS:[EDX+1]	00405141	3C 13	CMP AL,13
00405111	3C 5A	CMP AL,5A	00405143	75 B6	JNZ SHORT hw3_patc.004050FB
00405113	75 E6	JNZ SHORT hw3_patc.004050FB	00405145	0FB64424 06	MOVZX EAX,BYTE PTR SS:[ESP+6]
00405115	0FB64424 02	MOVZX EAX,BYTE PTR SS:[ESP+2]	0040514A	3242 06	XOR AL,BYTE PTR DS:[EDX+6]
0040511A	3242 02	XOR AL,BYTE PTR DS:[EDX+2]	0040514D	3C 0D	CMP AL,0D
0040511D	3C 61	CMP AL,61	0040514F	75 AA	JNZ SHORT hw3_patc.004050FB
0040511F	75 DA	JNZ SHORT hw3_patc.004050FB	00405151	0FB64424 07	MOVZX EAX,BYTE PTR SS:[ESP+7]
00405121	0FB64424 03	MOVZX EAX,BYTE PTR SS:[ESP+3]	00405156	3242 07	XOR AL,BYTE PTR DS:[EDX+7]
00405126	3242 03	XOR AL,BYTE PTR DS:[EDX+3]	00405159	3C 70	CMP AL,70
00405129	3C C0	CMP AL,C0	0040515D	0FB64424 08	JNZ SHORT hw3_patc.004050FB
0040512B	75 CE	JNZ SHORT hw3_patc.004050FB	00405162	3242 08	XOR AL,BYTE PTR DS:[EDX+8]
0040512D	0FB64424 04	MOVZX EAX,BYTE PTR SS:[ESP+4]	00405165	3C 1E	CMP AL,1E
			00405167	75 92	JNZ SHORT hw3_patc.004050FB

Ad esempio, riguardo la verifica del *primo carattere*, viene portato in *EAX* il primo byte associato al codice di sblocco. Successivamente viene effettuato lo *XOR* tra il primo byte del nostro codice (quindi il byte puntato da *EDX*) ed il primo byte relativo al codice di sblocco (quindi *EAX*). Si effettua infine il *CMP* tra il registro *AI* (lower bytes di *EAX*) ed il valore *0c*.

Analizzando il contenuto dei registri, verifichiamo che in *EAX* è stato portato il valore *3F* ovvero il primo byte contenuto all'indirizzo *0060fbe4* puntato da *ESP*.

Registers (FPU)		
EAX	0000003F	
ECX	00000034	
EDX	0060FC46	ASCII "my_code"
EBX	750AFB10	USER32.PostQuitMessage
ESP	0060FBE4	
EBP	0060FCB8	
ESI	0060FC46	ASCII "my_code"
EDI	001D03A6	

0060FBE4	A52F283F
0060FBE8	4F3D475D
0060FBEC	0000003F

Quindi, indicando con *XX* il primo byte del nostro codice, dobbiamo trovare il valore di *XX* tale che:

$$3F \oplus XX = 0C.$$

Sfruttando la proprietà per cui $B \oplus B = 0$, eseguendo lo *XOR* a destra e sinistra di *3F* abbiamo

$$3F \oplus 3F \oplus XX = 0C \oplus 3F$$

$$XX = 0C \oplus 3F = 33$$

Quindi il carattere da noi inserito deve essere uguale a $0C \oplus 3F$, ovvero 33 esadecimale.

Per proseguire con l'analisi dei restanti caratteri è necessario patchare le varie istruzioni di jump a seguito del controllo su ogni byte, in quanto avendo inserito un codice errato torneremmo subito alla procedura di ritorno. Sostituiamo tutte le istruzioni *JNZ* con istruzioni *JE*.

00405107	^74 F2	JE SHORT hw3_patc.004050FB
00405109	0FB64424 01	MOVZX EAX, BYTE PTR SS:[ESP+1]
0040510E	3242 01	XOR AL, BYTE PTR DS:[EDX+1]
00405111	3C 5A	CMP AL, 5A
00405113	^75 E6	JNZ SHORT hw3_patc.004050FB
00405115	0FB64424 02	MOVZX EAX, BYTE PTR SS:[ESP+2]
0040511A	3242 02	XOR AL, BYTE PTR DS:[EDX+2]
0040511D	3C 61	CMP AL, 61
0040511F	^74 DA	JE SHORT hw3_patc.004050FB

Utilizzando lo stesso approccio descritto per il primo byte, possiamo quindi ricavare tutti i caratteri del codice di sblocco:

```
EAX contiene 3F. Nel CMP il valore atteso è 0c → 3F ⊕ 0C = 33
EAX contiene 28. Nel CMP il valore atteso è 5A → 28 ⊕ 5A = 72
EAX contiene 2F. Nel CMP il valore atteso è 61 → 2F ⊕ 61 = 4E
EAX contiene A5. Nel CMP il valore atteso è C0 → A5 ⊕ C0 = 65
EAX contiene 5D. Nel CMP il valore atteso è 2E → 5D ⊕ 2E = 73
EAX contiene 47. Nel CMP il valore atteso è 13 → 47 ⊕ 13 = 54
EAX contiene 3D. Nel CMP il valore atteso è 0D → 3D ⊕ 0D = 30
EAX contiene 4F. Nel CMP il valore atteso è 70 → 4F ⊕ 70 = 3F
EAX contiene 3F. Nel CMP il valore atteso è 1E → 3F ⊕ 1E = 21
```

Andiamo successivamente a convertire in ASCII questa serie di valori esadecimali, ricavando il codice di sblocco: **3rNesT0?!**

Se tutti i caratteri del codice inserito corrispondono a quelli di **3rNesT0?!**, allora tutti i CMP daranno risultato zero, e verrà eseguito il jump a *00405174* piuttosto che JNZ verso la procedura di ritorno.

00405169	A1 E0A04000	MOV EAX,DWORD PTR DS:[40A0E0]
0040516E	85C0	TEST EAX,EAX
00405170	74 02	JE SHORT hw3_patc.00405174

A questo indirizzo, viene chiamata la funzione *puntata da ESP+28*, ovvero l'indirizzo *00404040* passato come parametro a *check_code*.

00405174	FF5424 28	CALL DWORD PTR SS:[ESP+28]	hw3_patc.00404040
00405178	A1 E0A04000	MOV EAX,DWORD PTR DS:[40A0E0]	
0040517D	85C0	TEST EAX,EAX	

Continuando con l'analisi a runtime, vediamo che in *00404040* viene chiamata *ExitWindowsEx*. Eseguendo tale istruzione viene effettuato lo shutdown del sistema.

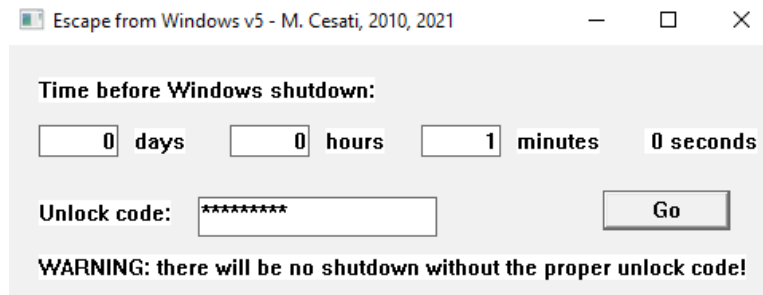
00404056	C70424 05000000	MOV DWORD PTR SS:[ESP],5	
0040405D	FF15 F8824000	CALL DWORD PTR DS:[<USER32.ExitWindowsEx>]	USER32.ExitWindowsEx
00404063	A1 04714000	MOV EAX,DWORD PTR DS:[407104]	
00404068	83EC 08	SUB ESP,8	

In questo caso lo shutdown è stato eseguito anche avendo inserito un codice errato, poichè sono state patchate tutte le istruzioni sul controllo del codice. Non abbiamo quindi ancora verificato se il codice ricavato è effettivamente quello corretto atteso dal programma.

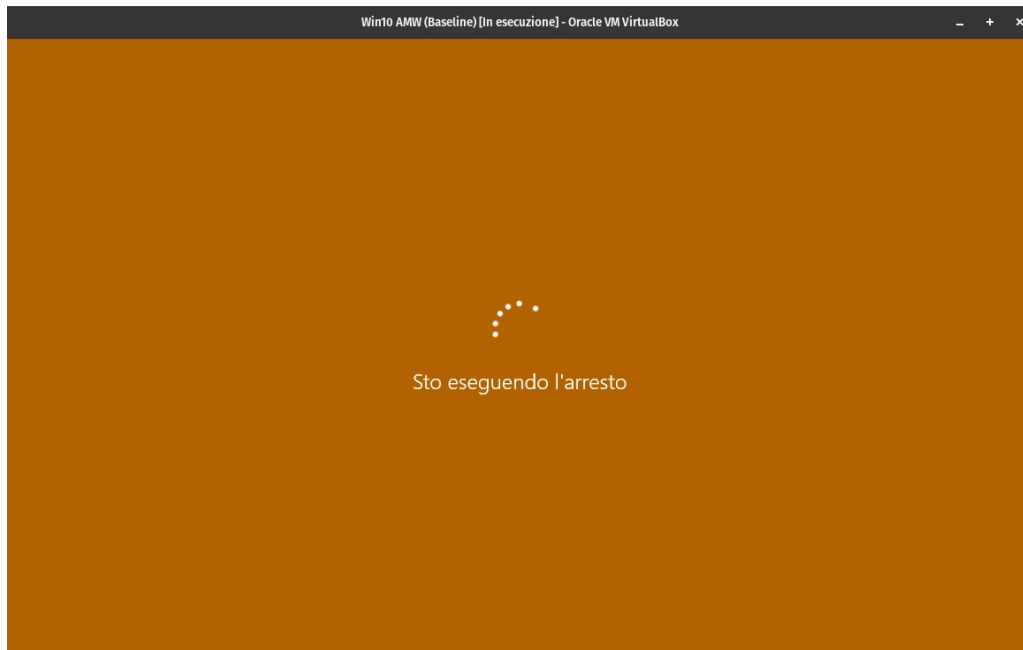
6 Verifica a Runtime

Vogliamo verificare se inserendo il codice *3rNesT0?!* viene eseguito lo shutdown del sistema invece di mostrare il messaggio di errore.

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



Vediamo che allo scadere del timer viene correttamente eseguito lo shutdown del sistema.



Questo ci conferma che il codice di sblocco che rende funzionale il programma è effettivamente *3rNesT0?!*