DOCUMENTO SUL TERZO HOMEWORK – MATTEO FANFARILLO

Il terzo homework consiste nel trovare, mediante gli strumenti forniti da Ghidra e OllyDbg, il codice segreto che rende funzionante il programma hw3.exe. Per far ciò, ho seguito i passaggi qui riportati:
1) Descrizione preliminare del programma.
2) Formalizzazione dell'obiettivo.
3) Ottenimento e disassemblaggio del codice macchina.
4) Localizzazione dei frammenti assembly di interesse.
5) Analisi dei frammenti assembly e delle relative strutture dati impiegate.
6) Verifica del risultato.
7) Riepilogo delle informazioni ottenute (questo passaggio lo sto portando a termine ora mentre scrivo il qui presente documento).

1) DESCRIZIONE PRELIMINARE DEL PROGRAMMA

Prima di iniziare l'analisi dell'eseguibile, ero a conoscenza solamente delle seguenti informazioni:
- Si tratta di un programma Windows a 32 bit basato su interfaccia e scritto con il linguaggio C.
- Il programma, per funzionare, ha bisogno dell'immissione di un codice segreto.

2) FORMALIZZAZIONE DELL'OBIETTIVO

L'obiettivo dell'homework è analizzare il programma con Ghidra e col debugger OllyDbg, in modo tale da ottenere il codice segreto che permette di sbloccare la funzionalità del programma stesso.

3) OTTENIMENTO E DISASSEMBLAGGIO DEL CODICE MACCHINA

Almeno inizialmente mi sono limitato a caricare il file eseguibile su Ghidra, il quale ha provveduto al disassemblaggio, generando così il codice assembly. Eventuali offuscamenti (anche riguardanti il debugger) verranno dunque affrontati durante la fase di analisi dei frammenti assembly e delle relative strutture dati impiegate.

4) LOCALIZZAZIONE DEI FRAMMENTI ASSEMBLY DI INTERESSE

L'obiettivo di questa fase (che coinvolge esclusivamente l'utilizzo di Ghidra) è trovare i frammenti assembly relativi al codice scritto dal programmatore.
Poiché abbiamo a che fare con un'applicazione Windows basata su interfaccia, è molto probabile che il suo codice contenga la funzione WinMain (che tipicamente è la prima scritta dal programmatore). Tale funzione è probabilmente caratterizzata dal cosiddetto message loop, che è un ciclo infinito contenente tre chiamate a funzione: GetMessage, TranslateMessage e DispatchMessage. Perciò, è stato sufficiente cercare il punto in cui una di queste tre WinAPI (ad esempio DispatchMessage) viene invocata: tale punto appartiene alla funzione FUN_004024e0, che potrebbe essere il WinMain. Il sospetto per cui si tratta effettivamente del WinMain viene alimentato dalla sua struttura all'interno del codice generato dal decompilatore: in effetti, si può notare che inizialmente viene inizializzata una struttura di tipo WNDCLASSEXA, poi vengono invocate alcune funzioni tra cui CreateWindowExA e, infine, si entra all'interno del message loop. Un altro inidizio molto importante è dato dalla segnatura della funzione: ci sono 4 parametri, tra cui il primo è stato già riconosciuto da Ghidra come un HINSTANCE, mentre l'ultimo è stato riconosciuto come un intero.
Tuttavia, la funzione ha una particolarità: non viene invocata direttamente dalla funzione entry, bensì da FUN_00403800 (ed è lei a essere chiamata da entry). Per ora trascuriamo quest'ultimo aspetto e assumiamo che FUN_004024e0 sia la WinMain.

Ho impostato quindi il prototipo di FUN_004024e0 nel seguente modo:
int WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)

5) ANALISI DEI FRAMMENTI ASSEMBLY E DELLE RELATIVE STRUTTURE DATI IMPIEGATE

Prima di proseguire con l'analisi dell'eseguibile su Ghidra, ho utilizzato il tool VirusTotal per determinare se l'applicazione contiene malware o meno, e ho ottenuto che 64 antivirus su 66 considerano hw3.exe safe, per cui, con ogni probabilità, non si tratta di un malware. Ho dunque lanciato il programma su macchina virtuale e ho realizzato che si tratta di un eseguibile identico a hw2.exe almeno agli occhi dell'utente; tuttavia, ci saranno sicuramente delle differenze all'interno del codice.

A valle di queste considerazioni, ho iniziato l'analisi statica del programma a partire dal WinMain. Qui, come accennato precedentemente, viene inizializzata la struttura WNDCLASSEXA, in cui è degno di nota il campo lpfnWndProc che viene inizializzato all'indirizzo della Window Procedure, che su Ghidra (e OllyDbg) ho chiamato WindowProc.

Ho impostato subito il prototipo di WindowProc:
LRESULT WindowProc (HWND hWnd, uint uMsg, WPARAM wParam, LPARAM lParam)

Dopodiché, anche con l'aiuto del comando Function Graph di Ghidra, ho potuto vedere quali sono i tipi di messaggio gestiti e selezionare quello (o quelli) che sembrano più pertinenti con il nostro scopo.
Il tipo di messaggio corrisponde al parametro uMsg di WindowProc e viene memorizzato nel registro EAX; poi, EAX è soggetto a una serie di controlli finché non verrà stabilito quale sarà l'handler da eseguire in base a qual è il tipo del messaggio. In particolare, EAX viene confrontato coi valori corrispondenti alle seguenti macro:
- WM_SIZE: variazione delle dimensioni della finestra.
- WM_PAINT: ridisegnazione della finestra.
- WM_COMMAND: tipo di messaggio inviato quando l'utente seleziona un comando da un menù, quando un controllo invia un messaggio di notifica alla finestra parent oppure quando viene tradotto un tasto di scelta rapida.
- WM_CREATE: creazione della finestra.
- WM_DESTROY: distruzione della finestra.
Tutti gli altri messaggi vengono passati al gestore di default.

Tra quelli sopra elencati, il tipo di messaggio che sembra più interessante è WM_COMMAND, per cui ho iniziato ad analizzare la porzione di codice di WindowProc relativo a esso.
Qui, per prima cosa, viene effettuato un controllo: se i 16 bit più significativi di wParam sono non nulli, viene invocata una return 0.
Poi c'è un controllo analogo sull'indirizzo memorizzato nel registro EBX aumentato di 184 byte. EBX era stato precedentemente inizializzato al valore di ritorno della funzione GetWindowLongA (la quale è stata invocata con GWL_USERDATA come secondo parametro). Ciò implica che EBX è uguale a un valore relativo a hWnd assegnato dal programmatore: per capire quanto vale questo valore, dobbiamo andare a ispezionare una qualche SetWindowLongA. A tal proposito, ho cercato SetWindowLongA all'interno di USER32.DLL tra gli imports, e sono andato a recuperare i riferimenti a questa funzione. Fortunatamente, risulta che l'API viene chiamata una sola volta, per cui ho raggiunto a colpo sicuro la zona di codice in cui avviene l'invocazione e che è relativa al gestore di WM_CREATE. Osservando i parametri passati a SetWindowLongA, ho dedotto che il valore relativo a hWnd assegnato dal programmatore corrisponde a lParam (o al primo campo di lParam). Poiché ora siamo all'interno del gestore di WM_CREATE, lParam è pari al puntatore a una struttura CREATESTRUCT che contiene le informazioni sulla finestra che sta per essere creata. La locazione di memoria a cui punta tale puntatore è pari all'indirizzo del primo campo di CREATESTRUCT, il quale viene comunemente chiamato lpCreateParams ed è di tipo LPVOID. Questo campo

contiene il valore del parametro lpParam specificato nella chiamata alla funzione CreateWindowExA (o CreateWindow). Cercando i riferimenti a CreateWindowExA, ho dedotto che l'API viene invocata all'interno di WinMain; si può anche notare che il parametro lpParam (l'ultimo di CreateWindowExA) è uguale al valore di ritorno di un'altra funzione: FUN_00401830.

Dando un primo sguardo a FUN_00401830, si intuisce che si tratta di una funzione molto piccola che inizializza delle strutture di dati. Per questo motivo, l'ho ridenominata init_struct. Tra l'altro, init_struct restituisce l'indirizzo DAT_00407020.
Rimettendo insieme i pezzi del puzzle, mi sono accorto che il registro EBX all'interno della WindowProc viene inizializzato esattamente all'indirizzo DAT_00407020 tramite la chiamata a GetWindowLongA.
Non solo: EBX viene acceduto con degli spiazzamenti costanti (come nell'istruzione CMP [EBX+184], ESI all'interno della porzione di codice di WindowProc relativa al tipo di messaggio WM_COMMAND).
Di conseguenza, possiamo concludere che DAT_00407020 è l'indirizzo base di una struct, che ho denominato APP_STRUCT.

Soffermandoci sull'istruzione CMP [EBX+184], ESI, capiamo che il campo al byte 0xb8 (= 184) della struct viene confrontato con un registro a 4 byte (ESI), che contiene l'indirizzo di lParam. Di conseguenza, si tratta di un campo di APP_STRUCT a 4 byte (molto probabilmente di un puntatore).
È giunta l'ora di definirci la struttura nella sezione Data Type Manager di Ghidra. Il suo tipo di dato l'ho chiamato struct app_struct e l'ho assegnato ad APP_STRUCT. Per il momento la struttura si presenta così:

OFFSET	LENGTH	MNEMONIC	DATA TYPE	NAME
0	184	??[184]	undefined[184]	
184	4	UINT	UINT	field_184

In precedenza abbiamo ricavato alcune informazioni interessanti grazie all'invocazione di SetWindowLongA in prossimità dell'etichetta CASE_WM_CREATE all'interno di WindowProc: torniamo dunque in questa zona di codice e vediamo cos'altro riusciamo a scoprire.
Prima di SetWindowLongA viene invocata nuovamente GetWindowLongA, ma stavolta con GWL_HINSTANCE come secondo parametro: il suo valore di ritorno è dunque l'handle all'istanza dell'applicazione, che verrà assegnato al registro EBX.
Dopo la chiamata a SetWindowLongA, invece, il byte 168 di APP_STRUCT viene inizializzato all'handle della finestra (hWnd):

OFFSET	LENGTH	MNEMONIC	DATA TYPE	NAME
0	168	??[168]	undefined[168]	
168	4	HWND	HWND	hWnd
172	1	??	undefined	
173	1	??	undefined	
174	1	??	undefined	
175	1	??	undefined	
176	1	??	undefined	
177	1	??	undefined	
178	1	??	undefined	
179	1	??	undefined	
180	1	??	undefined	
181	1	??	undefined	
182	1	??	undefined	
183	1	??	undefined	
184	4	UINT	UINT	field_184

Dopodiché c'è un loop in cui viene invocata CreateWindowExA: vengono quindi istanziate altre finestre. Il valore di ritorno della funzione è un handle alla nuova finestra che viene memorizzato nel byte 168+4*EBP di APP_STRUCT; dopodiché, il valore del registro EBP (che inizialmente era impostato a 1) viene incrementato di un'unità e viene controllato: nel momento in cui diviene strettamente maggiore di 3, si esce dal ciclo (che nel frattempo ho ridenominato CREATE_CHLD_WND_LOOP). Ciò vuol dire che EBX è l'indice all'interno di CREATE_CHLD_WND_LOOP e che vengono effettuate in tutto tre iterazioni, per cui vengono create tre finestre child, che appartengono alla classe "EDIT" e hanno l'identificativo rispettivamente pari a 1, 2 e 3.
A questo punto APP_STRUCT si presenta così:

OFFSET	LENGTH	MNEMONIC	DATA TYPE	NAME
0	168	??[168]	undefined[168]	
168	4	HWND	HWND	hWnd
172	4	HWND	HWND	hEdit1
176	4	HWND	HWND	hEdit2
180	4	HWND	HWND	hEdit3
184	4	UINT	UINT	field_184

Dopo il loop vengono create altre 2 finestre child tramite la CreateWindowExA, e i loro handle vengono memorizzati nel byte 184 e nel byte 188 della nostra struttura. In particolare:
- APP_STRUCT[184] appartiene alla classe "BUTTON", ha il nome "Go" e ha l'identificativo pari a 4;
- APP_STRUCT[188] appartiene alla classe "EDIT" e ha l'identificativo pari a 5.

Perciò la struttura diventa:

OFFSET	LENGTH	MNEMONIC	DATA TYPE	NAME
0	168	??[168]	undefined[168]	
168	4	HWND	HWND	hWnd
172	4	HWND	HWND	hEdit1
176	4	HWND	HWND	hEdit2
180	4	HWND	HWND	hEdit3
184	4	HWND	HWND	hButton
188	4	HWND	HWND	hEdit4

A questo punto potrebbe essere utile completare la struttura e, in base a ciò che avevo visto in precedenza, la funzione init_struct può essere d'aiuto: perciò sono andato ad analizzarla per scoprire quali altri campi vengono inizializzati. Al termine di tale operazione, APP_STRUCT diventa:

OFFSET	LENGTH	MNEMONIC	DATA TYPE	NAME
0	4	int	int	init_0
4	4	int	int	init_1000
8	1	??	undefined	
9	1	??	undefined	
10	1	??	undefined	
11	1	??	undefined	
12	4	int	int	init_1800
16	4	int	int	init_0_bis
20	1	??	undefined	
21	1	??	undefined	
22	1	??	undefined	
23	1	??	undefined	
24	144	??[144]	undefined[144]	

168	4	HWND	HWND	hWnd
172	4	HWND	HWND	hEdit1
176	4	HWND	HWND	hEdit2
180	4	HWND	HWND	hEdit3
184	4	HWND	HWND	hButton
188	4	HWND	HWND	hEdit4

La funzione init_struct ha inoltre due particolarità:
- Il campo di APP_STRUCT che si trova al byte 20 viene inizializzato col parametro di input della funzione stessa, che si tratta dell'indirizzo di una variabile locale (DAT_004040e0).
- Viene invocata per due volte la funzione FUN_00401530, che presenta due caratteristiche rilevanti.
In primo luogo, accetta come parametri di input un buffer, un intero e una stringa.
In secondo luogo, al suo interno invoca una funzione di libreria, ovvero MSVCRT.DLL::_vsnprintf.
Ciò ci induce a pensare che FUN_00401530 non è altro che una snprintf. In particolare, entrambe le volte in cui viene invocata, ha come primo parametro un buffer differente definito in APP_STRUCT, in cui verrà inserita la stringa passata come terzo parametro. Tali buffer si trovano rispettivamente al byte 24 e al byte 152 della nostra struttura, e le loro dimensioni sono specificate nel secondo parametro delle due chiamate a snprintf.

A seguito di queste considerazioni, APP_STRUCT appare così:

OFFSET	LENGTH	MNEMONIC	DATA TYPE	NAME
0	4	int	int	init_0
4	4	int	int	init_1000
8	1	??	undefined	
9	1	??	undefined	
10	1	??	undefined	
11	1	??	undefined	
12	4	int	int	init_1800
16	4	int	int	init_0_bis
20	4	??[4]	undefined[4]	init_struct_param
24	128	char[128]	char[128]	str1
152	16	char[16]	char[16]	str2
168	4	HWND	HWND	hWnd
172	4	HWND	HWND	hEdit1
176	4	HWND	HWND	hEdit2
180	4	HWND	HWND	hEdit3
184	4	HWND	HWND	hButton
188	4	HWND	HWND	hEdit4

Torniamo ora al codice sottostante all'etichetta CASE_WM_CREATE in WindowProc: qui, prima di giungere all'istruzione RET, vengono invocate tre funzioni: FUN_004018b0, FUN_00401b30 e FUN_004016b0; proviamo a darvi uno sguardo.

Per quanto riguarda FUN_004018b0, il decompilatore sembra essere d'aiuto. Infatti, ci suggerisce che all'interno del campo str2 di APP_STRUCT, tramite snprintf, viene inserita la stringa "%2ld seconds", dove %2ld sta per il risultato del seguente calcolo:

$$(init\_1800 - init\_0) - 60 \times \frac{1000 \times (init\_1800 - init\_0)}{60 \times init\_1000}$$

Si deduce facilmente che questa espressione dà luogo al campo dei secondi nel countdown e che, in particolare, init_1800 – init_0 è uguale al numero di secondi totali rimanenti allo spegnimento della macchina. Effettivamente, quando l'applicazione viene lanciata, il timer viene impostato di default a 30 minuti, che corrispondono proprio a 1800 secondi. Perciò:
- init_1800 è uguale al numero di secondi da cui parte il countdown e l'ho ridenominato shutdown_time;
- init_0 è uguale al numero di secondi trascorsi dall'inizio del countdown e l'ho ridenominato time_passed;
- init_1000 è un valore che dovrebbe in qualche modo tener traccia del trascorrere del tempo e rappresenta il numero di tick in un secondo: l'ho dunque ridenominato tick_length.

All'interno di FUN_004018b0 sono definite 4 variabili locali:
UINT uValue;
UINT uValue_00;
UINT uValue_01;
HWND hDlg;
Inizialmente, la variabile uValue viene inizializzata al numero totale di minuti rimanenti allo shutdown mediante la seguente operazione:
uValue = ((APP_STRUCT.shutdown_time-APP_STRUCT.time_passed)*1000) / (APP_STRUCT.tick_length*60);

Dopodiché, uValue_00 viene inizializzata a 0 e si itera sul seguente ciclo:
```
for (; 1439 < uValue; uValue = uValue - 1440) {
   uValue_00 = uValue_00 + 1;
}
```
Ciò equivale nella pratica a effettuare una divisione di uValue per 1440 e memorizzare il risultato in uValue_00 e il resto in uValue. Poiché 1440 sono esattamente i minuti in un giorno, la variabile uValue_00 sarà uguale al numero di giorni rimanenti che figureranno all'interno del countdown.

Anche uValue_01 viene inizializzata a 0 e si itera sul seguente ciclo:
```
for (; 59 < uValue; uValue = uValue - 60) {
   uValue_01 = uValue_01 + 1;
}
```
Si tratta di un'operazione del tutto analoga alla precedente, e ha come effetto finale quello di impostare il numero di ore all'interno del countdown uguale a uValue_01 e il numero di minuti uguale a uValue.

La variabile hDlg è invece inizializzata a APP_STRUCT.hWnd.

In conclusione, ho applicato le seguenti ridenominazioni alle variabili locali di FUN_004018b0:
- uValue → minutes
- uValue_00 → days
- uValue_01 → hours
- hDlg → hWnd

Inoltre, ho ridenominato la funzione calculate_time.

Per quanto invece riguarda FUN_00401b30, è una funzione molto semplice, che riceve come parametro in ingresso l'handle della finestra hWnd e invoca la funzione SetTimer, la quale restituisce un handle al timer che viene impostato e che viene inserito all'interno del terzo campo di APP_STRUCT:

OFFSET	LENGTH	MNEMONIC	DATA TYPE	NAME
0	4	int	int	time_passed
4	4	int	int	tick_length
8	4	UINT_PTR	UINT_PTR	timerID
12	4	int	int	shutdown_time
16	4	int	int	init_0_bis
20	4	??[4]	undefined[4]	init_struct_param
24	128	char[128]	char[128]	str1
152	16	char[16]	char[16]	str2
168	4	HWND	HWND	hWnd
172	4	HWND	HWND	hEdit1
176	4	HWND	HWND	hEdit2
180	4	HWND	HWND	hEdit3
184	4	HWND	HWND	hButton
188	4	HWND	HWND	hEdit4

Possiamo ridenominare la funzione FUN_00401b30 in invoke_SetTimer e possiamo notare che il quarto parametro passato all'API SetTimer (che corrisponde al puntatore alla funzione che verrà attivata allo scadere del timeout, ovvero la TimerProc) è pari all'indirizzo LAB_004019a0.
Osservando il codice di TimerProc generato dal decompilatore, possiamo accorgerci che il campo init_0_bis di APP_STRUCT è diverso da 0 solo ogni volta che il tempo rimanente allo shutdown va ricalcolato e la finestra va ridisegnata a seguito del passare del tempo durante il countdown. Possiamo così ridenominare questo campo della struttura redraw_flag.
In TimerProc, inoltre, c'è una coppia di istruzioni molto particolare:

```
MOV     dword ptr [ESP]=>local_1c, APP_STRUCT
CALL    dword ptr [APP_STRUCT.init_struct_param[0]]
```

Il campo init_struct_param dovrebbe dunque contenere l'indirizzo di una funzione (che accetta l'indirizzo di APP_STRUCT come parametro). Tale indirizzo dovrebbe corrispondere a quello passato come parametro alla funzione init_struct da parte di WinMain, ovvero 0x004040e0. Ma attenzione: è un indirizzo che appartiene alla sezione *data* dove apparentemente non c'è nulla!
Ove possibile ho usato il comando Disassemble, in modo tale che Ghidra interpretasse quei byte misteriosi con delle istruzioni assembly. Così facendo, è comparsa una funzione in cui è stata utilizzata una tecnica anti-disassembler: più volte viene letto un dato nella sezione bss (in particolare DAT_00407104) mediante un'istruzione del tipo MOV EDX, dword ptr [DAT_00407104] e, successivamente, compare la seguente sequenza di istruzioni:

```
TEST    EDX, EDX
JZ      LAB_004040fa+2
LAB_004040fa+2
CALLF   0x0: SUB_1040c742
```

Questo è il classico esempio in cui compare un salto apparentemente condizionato ma che in realtà viene sempre preso; in tal modo, Ghidra assume che subito dopo l'istruzione di salto ci siano necessariamente altre istruzioni (il che non vale per JMP) e può interpretare in maniera errata in modo particolare i byte successivi a JZ (ad esempio con uno sfasamento).
Tuttavia, non sono riuscito in un primo momento a effettuare una patch sul database di Ghidra in modo tale che comparissero le istruzioni corrette, per cui ho momentaneamente ridenominato la funzione anti_disassembler1 e ho valutato di tornarci sopra successivamente con OllyDbg.

A proposito di OllyDbg, ho effettuato un primissimo controllo su eventuali tecniche anti-debugger adottate all'interno dell'eseguibile andando a ispezionare la sezione Defined Strings di Ghidra. In effetti qui compaiono almento tre stringhe interessanti: "IsDebuggerPresent", "LoadLibraryA" e "GetProcAddress". Esse sono relative a delle particolari API che, effettivamente, compaiono anche nella lista delle funzioni della dll KERNEL32.DLL che è stata importata all'interno del programma.

Innanzitutto sono andato a vedere dove viene invocata IsDebuggerPresent: ciò viene fatto una volta sola, all'interno della funzione FUN_004024a0 che ho ridenominato subito anti_debugger1.
Tale funzione, prima di invocare ShowWindow, rileva se l'eseguibile è stato lanciato su un debugger: se sì, invoca l'API ExitProcess. Per ovviare a tale problema, ho effettuato una patch direttamente su hw3.exe tramite OllyDbg, modificando anti_debugger1 in modo tale da rimpiazzare le istruzioni relative al rilevamento del debugger con delle NOP. Mettendo in atto tale modifica, ho generato un secondo eseguibile che ho denominato hw3-p1.exe e che ho caricato su OllyDbg al posto di hw3.exe.

D'altra parte, ho realizzato che, momentaneamente, a Ghidra non risulta alcuna invocazione alle funzioni LoadLibraryA e GetProcAddress: non posso ancora assumere con certezza che vengano caricate DLL a runtime nello spazio di indirizzamento.

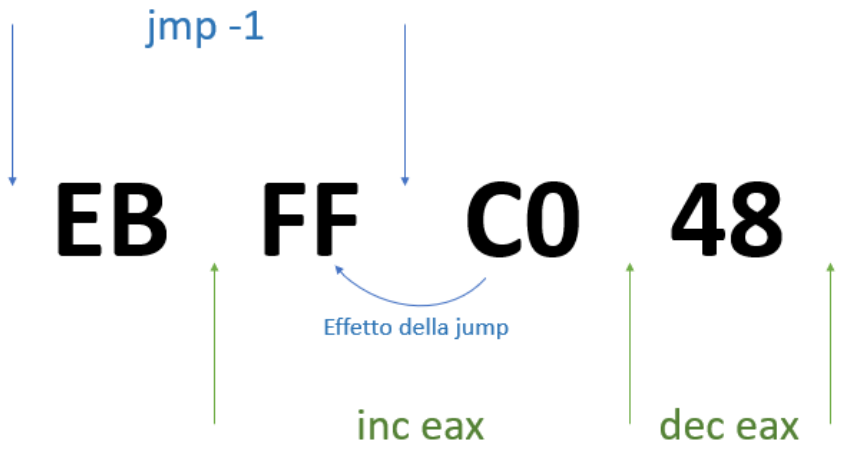
Dopodiché ho ritenuto utile andare a vedere in quali altri punti del codice viene invocata l'API ExitProcess per vedere se anche altrove è utilizzata in ambito anti-debugger: c'è solo un'altra chiamata oltre a quella già incontrata e avviene all'interno di una particolare funzione (LAB_004042a0) che, all'interno dell'address space, si trova nella sezione *data* immediatamente dopo anti_disassembler1 (per cui ho avuto modo di disassemblarla già in precedenza, ottenendo tuttavia lo stesso identico risultato di anti_disassembler1, ovvero una funzione in cui è stata applicata la tecnica anti-disassembler di cui sopra).
All'interno di tale funzione, che ho ridenominato anti_disassembler2, l'unica porzione di codice apparentemente interpretata in modo corretto da Ghidra contiene appunto l'invocazione a ExitProcess e poi due invocazioni a quella che sembra una piccola variante di snprintf, con la differenza che utilizza come secondo parametro - che indica la dimensione del buffer in cui si vanno a scrivere i dati - il valore 128 in modo deterministico. Per questo motivo, ho chiamato tale variante snprintf128. Se ci facciamo caso, alla prima invocazione di snprintf128, all'interno di un buffer viene inserita la stringa formato "InternalError=%lu/0x%lx" (dove il valore da inserire al posto di %lu e %lx non viene esplicitato), mentre, alla seconda invocazione, all'interno di un altro buffer viene inserita una stringa formato che si traduce in "DEBUG InternalError:148".
Purtroppo non sono riuscito a fare ulteriori osservazioni riguardanti anti_disassembler2 con solo Ghidra.

Ho così pensato di introdurre l'utilizzo di OllyDbg all'interno dell'analisi. La prima cosa che ho fatto è stata eseguire interamente il programma su OllyDbg tramite il comando run. Tuttavia, non si è aperta la finestra dell'applicazione, il che vuol dire che esistono ancora delle misure anti-debbuger da eliminare in una delle prime funzioni scritte dal programmatore.
Sono dunque tornato in WindowProc, dove ho notato che, subito dopo l'invocazione a GetWindowLongA, viene chiamata la funzione FUN_00401dc0, che è molto semplice: controlla se l'applicazione viene eseguita con un debugger controllando l'offset 2 della struttura Process Environment Block, la quale si trova all'offset 30 del segmento puntato dal registro FS. Se la risposta è affermativa, allora viene cambiato il valore puntato dall'indirizzo passato come parametro di ingresso alla funzione, ovvero il valore di uMsg (il tipo di messaggio che ha colpito la finestra). Perciò, ho ridenominato la funzione in anti_debugger2 e ho patchato nuovamente l'eseguibile mediante OllyDbg, in modo tale che anti_debugger2 non avesse più effetto. In tal modo, ho generato una terza versione dell'eseguibile, hw3-p2.exe.
Eseguendo hw3-p2.exe su OllyDbg, ho ottenuto un esito differente ma non quello sperato: dopo alcuni secondi dall'avvio, OllyDbg crasha dopo la creazione della finestra principale e prima della creazione delle 5 finestre child. Se invece eseguo l'eseguibile normalmente, appare una finestra di errore e il processo termina. La finestra di errore ha come intestazione e come contenuto proprio le due stringhe che abbiamo

incontrato precedentemente in anti_disassembler2 durante l'analisi su Ghidra, ovvero rispettivamente "DEBUG InternalError:148" e "InternalError=1514324433/0x5a42c1d1".
A questo punto ho provato a eseguire il programma in maniera più controllata: ho inserito un breakpoint in prossimità dell'invocazione alla funzione anti_disassembler2 all'interno di TimerProc, e ho lanciato il comando run affinché l'esecuzione arrivasse a tale breakpoint. Dopodiché, tramite il comando Step Into, ho osservato il comportamento del programma all'interno di anti_disassembler2: qui succede ben poco a parte la comparsa a schermo della finestra d'errore (DEBUG InternalError:148) e della terminazione del processo.
Successivamente, ho ispezionato il decompilatore di Ghidra relativamente alla funzione TimerProc, e ho visto che l'invocazione ad anti_disassembler2 è l'unica operazione effettuata se una certa condizione (APP_STRUCT.time_passed & 7 == 0) è verificata.
Ho provato così a patchare nuovamente l'eseguibile annullando gli effetti di anti_disassembler2 e generando un nuovo programma che ho denominato hw3-p3.exe. Tale eseguibile, se lanciato normalmente, sembra funzionare correttamente, mentre OllyDbg continua a crashare: ci sono quindi altre misure anti-debugger a cui far fronte.
Eseguendo nuovamente l'applicazione su OllyDbg in modo controllato e posizionando dei breakpoint in modo opportuno, ho capito che il crash di OllyDbg avviene all'invocazione dell'API ShowWindow che si ha all'interno di anti_debugger1. Ciò vuol dire che, prima della chiamata di anti_debugger1 da parte di WinMain, è stata adottata una misura che serve a sfruttare una vulnerabilità di OllyDbg. Analizzando più attentamente il codice disassemblato di WinMain e delle funzioni che essa invoca prima di anti_debugger1, ho notato l'esistenza di una funzione che avevo precedentemente trascurato: FUN_004016f0, la cui invocazione avviene all'interno di init_struct. Di tale funzione, non c'era codice disassemblato; inoltre, è balzato subito all'occhio l'utilizzo della seguente sequenza di byte di istruzioni:

EB FF C0 48

L'effetto di questa sequenza è schematizzato qui di seguito:



Se ci facciamo caso, il programma si comporta esattamente come se tale sequenza non esistesse. Questi 4 byte dunque servono solo a confondere il disassemblatore il quale, per sua natura, non assume mai che un qualche byte (come FF nel nostro caso) possa far parte di due istruzioni differenti (JMP -1; INC EAX).
Per ovviare a questo inconveniente, ho effettuato una patch all'interno del database di Ghidra, andando a sostituire, nella funzione FUN_004016f0, ciascuna occorrenza della sequenza di byte **EB FF C0 48** con delle NOP. Dopodiché, ho invocato il comando Disassemble altrove all'interno di FUN_004016f0, ottenendo così tantissime istruzioni MOV e delle chiamate a due API: LoadLibraryA e GetProcAddress. Inoltre, poiché Ghidra non aveva riconosciuto alcuna variabile automatica per FUN_004016f0, ho eseguito il comando Re-create Function di Ghidra in modo tale da generare lo stack.

Le istruzioni MOV hanno lo scopo di salvare singoli byte sullo stack; l'indirizzo del primo di questi byte viene poi dato in input a LoadLibraryA / GetProcAddress. Ma poiché entrambe le API accettano una stringa come parametro, i byte in questione sono sicuramente dei caratteri, per cui su Ghidra ho effettuato le opportune conversioni. In definitiva, FUN_004016f0 compare come segue (per brevità ho omesso gli indirizzi e i byte relativi alle istruzioni macchina che compaiono alla sinistra di ogni riga):

```
undefined_stdcall FUN_004016f0 (void)

undefined       AL:1            <RETURN>
char            Stack[-0x1b]:1  local_1b
char            Stack[-0x1c]:1  local_1c
char            Stack[-0x1d]:1  local_1d
char            Stack[-0x1e]:1  local_1e
char            Stack[-0x1f]:1  local_1f
char            Stack[-0x20]:1  local_20
char            Stack[-0x21]:1  local_21
char            Stack[-0x22]:1  local_22
char            Stack[-0x23]:1  local_23
char            Stack[-0x24]:1  local_24
char            Stack[-0x25]:1  local_25
char            Stack[-0x26]:1  local_26
char            Stack[-0x27]:1  local_27
char            Stack[-0x28]:1  local_28
char            Stack[-0x29]:1  local_29
char            Stack[-0x2a]:1  local_2a
char            Stack[-0x2b]:1  local_2b
char            Stack[-0x2c]:1  local_2c
char            Stack[-0x2d]:1  local_2d
undefined4      Stack[-0x48]:4  fun_arg2
undefined4      Stack[-0x4c]:4  fun_arg1

                FUN_004016f0
        PUSH    EBX
        SUB     ESP, 0x48
        NOP
        NOP
        MOV     byte ptr [ESP + local_2d], 'k'
        NOP
        NOP
        MOV     byte ptr [ESP + local_29], 'e'
        MOV     byte ptr [ESP + local_2c], 'e'
        NOP
        NOP
        MOV     byte ptr [ESP + local_2b], 'r'
        NOP
        NOP
        MOV     byte ptr [ESP + local_2a], 'n'
        NOP
        NOP
        MOV     byte ptr [ESP + local_28], 'l'
```

```
NOP
NOP
MOV     byte ptr [ESP + local_27], '3'
NOP
NOP
MOVZX  EAX, byte ptr [ESP + local_27]
SUB     EAX, 1
MOV     byte ptr [ESP + local_26], AL
NOP
NOP
MOV     byte ptr [ESP + local_25], '.'
NOP
NOP
MOV     byte ptr [ESP + local_24], 'd'
NOP
NOP
MOV     byte ptr [ESP + local_22], 'l'
MOV     byte ptr [ESP + local_22], 'l'
NOP
NOP
MOV     byte ptr [ESP + local_21], '\0'
NOP
NOP
LEA     EBX=>local_2d, [ESP + 0x1f]
MOV     dword ptr [ESP]=>fun_arg1, EBX
CALL    dword ptr [->KERNEL32.DLL::LoadLibraryA]
SUB     ESP, 0x4
NOP
NOP
MOV     byte ptr [ESP + local_2d], 'O'
NOP
NOP
MOV     byte ptr [ESP + local_24], 'u'
MOV     byte ptr [ESP + local_29], 'u'
MOV     byte ptr [ESP + local_2c], 'u'
NOP
NOP
MOV     byte ptr [ESP + local_21], 't'
MOV     byte ptr [ESP + local_28], 't'
MOV     byte ptr [ESP + local_2b], 't'
NOP
NOP
MOV     byte ptr [ESP + local_2a], 'p'
NOP
NOP
MOV     byte ptr [ESP + local_27], 'D'
NOP
NOP
MOV     byte ptr [ESP + local_26], 'e'
```

```
NOP
NOP
MOV    byte ptr [ESP + local_25], 'b'
NOP
NOP
MOV    byte ptr [ESP + local_1d], 'g'
MOV    byte ptr [ESP + local_23], 'g'
NOP
NOP
MOV    byte ptr [ESP + local_22], 'S'
NOP
NOP
MOV    byte ptr [ESP + local_20], 'r'
NOP
NOP
MOV    byte ptr [ESP + local_1f], 'i'
NOP
NOP
DEC    EAX
MOV    byte ptr [ESP + local_1e], 'n'
NOP
NOP
MOV    byte ptr [ESP + local_1c], 'A'
NOP
NOP
MOV    byte ptr [ESP + local_1b], '\0'
NOP
NOP
DEC    EAX
MOV    dword ptr [ESP + fun_arg2], EBX
MOV    dword ptr [ESP]=>fun_arg1, EAX
CALL   dword ptr [->KERNEL32.DLL::GetProcAddress]
SUB    ESP, 0x8
ADD    ESP, 0x48
POP    EBX
RET
```

La funzione LoadLibraryA carica a runtime un modulo (una DLL) nello spazio di indirizzamento del processo. Se ha successo, restituisce un handle al modulo caricato. Accetta in input un solo parametro:

#	TIPO	NOME	DESCRIZIONE	VALORE
1	LPCSTR	lpLibFileName	Nome del modulo	"kernel32.dll"

GetProcAddress, invece, recupera l'indirizzo di una funzione esportata da una data DLL. Se ha successo, restituisce l'indirizzo della funzione. I suoi parametri sono riassunti nella seguente tabella:

#	TIPO	NOME	DESCRIZIONE	VALORE
1	HMODULE	hModule	Handle del modulo (della DLL)	Handle di kernel32.dll
2	LPCSTR	lpProcName	Nome della funzione da recuperare	"OutputDebugStringA"

OutputDebugStringA è dunque l'API i cui riferimenti erano nascosti all'interno dell'eseguibile. Effettivamente, il suo scopo è inviare un messaggio (una stringa) al debugger. Se sfruttata in modo opportuno, questa funzione può potenzialmente fare in modo che il debugger acceda sullo stack a un indirizzo non valido, vada in segmentation fault e crashi, chiudendosi in modo anomalo.
Questo meccanismo potrebbe fare al caso nostro: ho dunque ridenominato FUN_004016f0 in anti_debugger3 (consapevole comunque del fatto che qui è stata adottata anche una tecnica anti-disassembler, della quale ho disquisito precedentemente).

Avevo in precedenza notato che il crash di OllyDbg avviene all'interno della chiamata all'API ShowWindow, il che non mi è stato molto d'aiuto poiché il mio obiettivo era patchare del codice scritto dal programmatore. Ho dovuto così passare alla WindowProc e, in particolare, alla porzione di codice che gestisce l'evento WM_SIZE. Ho posizionato un breakpoint lì e, da quel punto in poi, ho avviato l'esecuzione un'istruzione per volta col comando Step Over di OllyDbg. Il debugger è crashato nuovamente nell'esecuzione della funzione FUN_00404000. Ghidra, di tale funzione, è riuscito a disassemblare poco o niente, per cui ho ritenuto opportuno ripercorrerla istruzione per istruzione con OllyDbg: dopo alcune istruzioni preliminari e alcuni loop, effettivamente sono giunto alla definizione della stringa "%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s" con conseguente chiamata alla funzione OutputDebugStringA.
In definitiva, ho ridenominato la funzione FUN_00404000 in anti_debugger4 e l'ho patchata su OllyDbg inserendo una RETN come prima istruzione, in modo tale che non abbia più alcun effetto. Ho generato così un nuovo eseguibile che ho chiamato hw3-p4.exe.

Ho provato a eseguire hw3-p4.exe sia normalmente che su OllyDbg: adesso funziona correttamente, per cui apparentemente le misure anti-debugger sono state superate tutte.

A questo punto ho fissato un breakpoint alla prima istruzione del ramo if di TimerProc che viene preso quando scade il timeout, ovvero in prossimità di:

```
MOV     dword ptr [ESP]=>local_1c, APP_STRUCT
CALL    dword ptr [APP_STRUCT.init_struct_param[0]]
```

È già possibile intuire che APP_STRUCT.init_struct_param[0] indica la funzione che ha la responsabilità di verificare che la password inserita nell'apposita finestra sia corretta ed, eventualmente, spegnere la macchina. Ho perciò ridenominato check_pw_fun questo campo della struttura.

Dopo aver fissato il breakpoint, ho lanciato l'applicazione su OllyDbg, ho impostato il timer a 1 minuto, l'ho fatto partire e ne ho atteso la terminazione. Poi, a partire dal breakpoint, ho eseguito un'istruzione per volta col comando Step Into, notando che l'istruzione CALL dword ptr [APP_STRUCT.init_struct_param[0]] invoca la funzione anti_disassembler1. Stavolta, osservando attentamente su OllyDbg le istruzioni che vengono eseguite all'interno di anti_disassembler1, sono riuscito a ricostruirne il codice assembly corretto. In particolare, è stato più volte necessario applicare la seguente trasformazione in Ghidra:

PRIMA:

```
TEST        EAX, EAX
JZ          LAB_00404182+2
                LAB_00404182+2
CALLF       0xc730:SUB_24448d42
INC         ESP
AND         AL, 0x14
ADD         byte ptr [EAX], AL
ADD         byte ptr [EAX], AL
```

DOPO:

```
        TEST        EAX, EAX
        JZ          LAB_00404184
        ??
        ??
                                LAB_00404184
        LEA         EAX, [ESP + 0x30]
        MOV         dword ptr [ESP + 0x14], 0x0
```

Così facendo per tutte le istruzioni di anti_disassembler1, ho ottenuto del codice ben formato. A questo punto, ho selezionato tale codice e ho eseguito l'istruzione Re-create Function di Ghidra in modo tale che comparissero tutte le variabili locali (e, quindi, lo stack).
In definitiva, anti_disassembler1 si presenta nel seguente modo:

undefined __cdecl anti_disassembler1 (struct app_struct * app_struct)

```
undefined               AL:1            <RETURN>
struct app_struct*      Stack[0x4]:4    app_struct
char[30]                Stack[-0x2a]…   pw_buff
undefined4              Stack[-0x30]:4  local_30
undefined1              Stack[-0x38]:1  local_38
undefined4              Stack[-0x3c]:4  local_3c
undefined4              Stack[-0x40]:4  local_40
undefined4              Stack[-0x58]:4  fun_arg6
undefined4              Stack[-0x5c]:4  fun_arg5
undefined4              Stack[-0x60]:4  fun_arg4
undefined4              Stack[-0x64]:4  fun_arg3
undefined4              Stack[-0x68]:4  fun_arg2
undefined4              Stack[-0x6c]:4  fun_arg1

                        anti_disassembler1
        PUSH    EDI
        PUSH    ESI
        PUSH    EBX
        SUB     ESP, 0x60
        MOV     EDX, dword ptr [DAT_00407104]
        MOV     EAX, dword ptr [ESP + app_struct]
        MOV     EDI, dword ptr [EAX + 168]
        TEST    EDX, EDX
        JZ      LAB_004040fc
        ??
        ??
                        LAB_004040fc
        MOV     dword ptr [EAX + 0x10], 0x0
        MOV     EAX, [DAT_00407104]
        TEST    EAX, EAX
        JZ      LAB_0040410e
        ??
        ??
                        LAB_0040410e
        CALL    dword ptr [->KERNEL32.DLL::GetCurrentProcess]
```

```
LEA     EDX=>local_40, [ESP + 0x2c]
MOV     dword ptr [ESP + fun_arg2], 0x28
MOV     dword ptr [ESP + fun_arg3], EDX
MOV     dword ptr [ESP]=>fun_arg1, EAX
CALL    dword ptr [->ADVAPI32.DLL::OpenProcessToken]
MOV     EBX, dword ptr [->USER32.DLL::PostQuitMessage]
SUB     ESP, 0xc
TEST    EAX, EAX
JZ      LAB_00404280
                LAB_0040413e
MOV     EAX, [DAT_00407104]
TEST    EAX, EAX
JZ      LAB_00404149
??
??
                LAB_00404149
LEA     EAX=>local_38, [ESP + 0x34]
MOV     dword ptr [ESP + fun_arg2], s_SeShutdowmPrivilege_00406168
MOV     dword ptr [ESP + fun_arg3], EAX
MOV     dword ptr [ESP]=>fun_arg1, 0x0
CALL    dword ptr [->ADVAPI32.DLL::LookupPrivilegeValueA]
MOV     EAX, [DAT_00407104]
SUB     ESP, 0xc
MOV     dword ptr [ESP + local_3c], 0x1
MOV     dword ptr [ESP + local_30], 0x2
TEST    EAX, EAX
JZ      LAB_00404184
??
??
                LAB_00404184
LEA     EAX=>local_3c, [ESP + 0x30]
MOV     dword ptr [ESP + fun_arg6], 0x0
MOV     dword ptr [ESP + fun_arg3], EAX
MOV     EAX, dword ptr [ESP + local_40]
MOV     dword ptr [ESP + fun_arg5], 0x0
MOV     dword ptr [ESP + fun_arg4], 0x0
MOV     dword ptr [ESP + fun_arg2], 0x0
MOV     dword ptr [ESP]=>fun_arg1, EAX
CALL    dword ptr [->ADVAPI32.DLL::AdjustTokenPrivileges]
MOV     EAX, [DAT_00407104]
SUB     ESP, 0x18
TEST    EAX, EAX
JZ      LAB_004041c7
??
??
                LAB_004041c7
CALL    dword ptr [->KERNEL32.DLL::GetLastError]
TEST    EAX, EAX
JNZ     LAB_00404243
```

```
MOV     EAX, [DAT_00407104]
TEST    EAX, EAX
JZ      LAB_004041dc
??
??
                    LAB_004041dc
LEA     ESI=>pw_buff, [ESP + 0x42]
MOV     dword ptr [ESP + fun_arg4], 30
MOV     dword ptr [ESP + fun_arg3], ESI
MOV     dword ptr [ESP + fun_arg2], 5
MOV     dword ptr [ESP]=>fun_arg1, EDI
CALL    dword ptr [->USER32.DLL::GetDlgItemTextA]
MOV     EDX, dword ptr [DAT_00407104]
SUB     ESP, 0x10
TEST    EDX, EDX
JNZ     LAB_0040427c
                    LAB_0040420a
XOR     ECX, ECX
LEA     ESI=>pw_buff, [ESI]
                    LAB_00404210
MOV     EDX, dword ptr [ECX*0x4 + FUN_004050c0]
XOR     EDX, 0x89a3fa2b
ROR     EDX, 0x9
MOV     dword ptr [ECX*0x4 + FUN_004050c0], EDX=>DAT_0040a0e0
ADD     ECX, 0x1
CMP     ECX, 0x34
JNZ     LAB_00404210
MOV     dword ptr [ESP + fun_arg3], DAT_00404040
MOV     dword ptr [ESP + fun_arg2], EAX
MOV     dword ptr [ESP]=>fun_arg1, ESI
CALL    FUN_004050c0
                    LAB_00404243
MOV     EAX, [DAT_00407104]
TEST    EAX, EAX
JZ      LAB_0040424e
??
??
                    LAB_0040424e
CALL    FUN_00401d80
MOV     EAX, [DAT_00407104]
TEST    EAX, EAX
JZ      LAB_0040425e
??
??
                    LAB_0040425e
MOV     dword ptr [ESP]=>fun_arg1, 0x0
CALL    EBX=>USER32.DLL::PostQuitMessage
MOV     EAX, [DAT_00407104]
SUB     ESP, 0x4
```

```
TEST    EAX, EAX
JZ      LAB_00404275
??
??
                LAB_00404275
ADD     ESP, 0x60
POP     EBX
POP     ESI
POP     EDI
RET
                LAB_0040427c
??
??
JMP     LAB_0040420a
                LAB_00404280
MOV     dword ptr [ESP]=>fun_arg1, 0x0
CALL    EBX=>USER32.DLL::PostQuitMessage
SUB     ESP, 0x4
JMP     LAB_0040413e
```

Analizzando il codice assembly di anti_disassembler1, notiamo che a un certo punto viene invocata l'API GetDlgItemTextA che, nel nostro caso specifico, recupera il contenuto della finestra relativa al codice di sblocco dell'applicazione e lo inserisce all'interno di un buffer (che ho chiamato pw_buff). Più avanti, il buffer e la lunghezza del codice di sblocco inserito dall'utente (che non è altro che il valore di ritorno di GetDlgItemTextA) vengono passati in input alla funzione FUN_004050c0, che potrebbe avere il compito di controllare se la password inserita è corretta o meno. Quest'ultima ipotesi viene rafforzata dal fatto che successivamente, sotto determinate condizioni, viene invocata un'altra funzione, FUN_00401d80, che semplicemente mostra un messaggio di errore relativo all'inserimento di una password errata tramite un'invocazione a MessageBoxA. Di conseguenza, ho ridenominato quest'ultima funzione show_error_msg.

Rimane ora da analizzare per bene FUN_004050c0. Purtroppo Ghidra non ci aiuta in alcun modo a riguardo, per cui un'idea può essere eseguire la funzione istruzione per istruzione con OllyDbg.
Qui, dopo alcune istruzioni preliminari (come il salvataggio della password inserita dall'utente all'interno del registro EDX), OllyDbg riconosce i seguenti controlli:

```
CMP     DWORD PTR SS:[ESP+24], 9
JE      SHORT hw3-p4.004050FF
ADD     ESP, 1C
RETN
MOVZX   EAX, BYTE PTR SS:[ESP]
XOR     AL, BYTE PTR DS:[EDX]
CMP     AL, 0C
JNZ     SHORT hw3-p4.004050FB
MOVZX   EAX, BYTE PTR SS:[ESP+1]
XOR     AL, BYTE PTR DS:[EDX+1]
CMP     AL, 5A
JNZ     SHORT hw3-p4.004050FB
```

```
MOVZX  EAX, BYTE PTR SS:[ESP+2]
XOR    AL, BYTE PTR DS:[EDX+2]
CMP    AL, 61
JNZ    SHORT hw3-p4.004050FB
MOVZX  EAX, BYTE PTR SS:[ESP+3]
XOR    AL, BYTE PTR DS:[EDX+3]
CMP    AL, C0
JNZ    SHORT hw3-p4.004050FB
MOVZX  EAX, BYTE PTR SS:[ESP+4]
XOR    AL, BYTE PTR DS:[EDX+4]
CMP    AL, 2E
JNZ    SHORT hw3-p4.004050FB
MOVZX  EAX, BYTE PTR SS:[ESP+5]
XOR    AL, BYTE PTR DS:[EDX+5]
CMP    AL, 13
JNZ    SHORT hw3-p4.004050FB
MOVZX  EAX, BYTE PTR SS:[ESP+6]
XOR    AL, BYTE PTR DS:[EDX+6]
CMP    AL, 0D
JNZ    SHORT hw3-p4.004050FB
MOVZX  EAX, BYTE PTR SS:[ESP+7]
XOR    AL, BYTE PTR DS:[EDX+7]
CMP    AL, 70
JNZ    SHORT hw3-p4.004050FB
MOVZX  EAX, BYTE PTR SS:[ESP+8]
XOR    AL, BYTE PTR DS:[EDX+8]
CMP    AL, 1E
JNZ    SHORT hw3-p4.004050FB
```

Come si può vedere, il primo controllo riguarda la lunghezza della password (che infatti è stata passata come parametro di input a FUN_004050c0), che deve essere di 9 caratteri. Successivamente, vengono controllati i caratteri uno per volta, col supporto dei registri EAX, AL (che contiene esattamente gli 8 bit meno significativi di EAX) ed EDX; ricordiamo che quest'ultimo contiene il codice di sblocco inserito dall'utente.
I controlli sui caratteri vengono effettuati nel seguente modo:

<u>1° carattere</u>
- EAX = 0x3f
- Lo XOR tra EAX e il primo byte di EDX deve dare luogo a 0x0c.

Perciò, il primo byte di EDX deve essere pari a 0x33 = 51, ovvero al carattere '3'.

<u>2° carattere</u>
- EAX = 0x28
- Lo XOR tra EAX e il secondo byte di EDX deve dare luogo a 0x5a.

Perciò, il secondo byte di EDX deve essere pari a 0x72 = 114, ovvero al carattere 'r'.

<u>3° carattere</u>
- EAX = 0x2f
- Lo XOR tra EAX e il terzo byte di EDX deve dare luogo a 0x61.

Perciò, il terzo byte di EDX deve essere pari a 0x4e = 78, ovvero al carattere 'N'.

4° carattere
- EAX = 0xa5
- Lo XOR tra EAX e il quarto byte di EDX deve dare luogo a 0xc0.
Perciò, il quarto byte di EDX deve essere pari a 0x65 = 101, ovvero al carattere 'e'.

5° carattere
- EAX = 0x5d
- Lo XOR tra EAX e il quinto byte di EDX deve dare luogo a 0x2e.
Perciò, il quinto byte di EDX deve essere pari a 0x73 = 115, ovvero al carattere 's'.

6° carattere
- EAX = 0x47
- Lo XOR tra EAX e il sesto byte di EDX deve dare luogo a 0x13.
Perciò, il sesto byte di EDX deve essere pari a 0x54 = 84, ovvero al carattere 'T'.

7° carattere
- EAX = 0x3d
- Lo XOR tra EAX e il settimo byte di EDX deve dare luogo a 0x0d.
Perciò, il settimo byte di EDX deve essere pari a 0x30 = 48, ovvero al carattere '0'.

8° carattere
- EAX = 0x4f
- Lo XOR tra EAX e l'ottavo byte di EDX deve dare luogo a 0x70.
Perciò, l'ottavo byte di EDX deve essere pari a 0x3f = 63, ovvero al carattere '?'.

9° carattere
- EAX = 0x3f
- Lo XOR tra EAX e il nono byte di EDX deve dare luogo a 0x1e.
Perciò, il nono byte di EDX deve essere pari a 0x21 = 33, ovvero al carattere '!'.

Notiamo che il controllo sull'i-esimo carattere viene effettuato solo se tutti i precedenti sono andati a buon fine; ciò vuol dire che è possibile scoprire un solo nuovo carattere del codice di sblocco ogni volta che viene lanciata l'applicazione col debugger. In altre parole, dopo aver capito che la password deve essere composta da 9 caratteri, ho dovuto eseguire hw3-p4.exe per altre dieci volte, inserendo via via le seguenti password:

1° tentativo: "999999999"
2° tentativo: "399999999"
3° tentativo: "3r9999999"
4° tentativo: "3rN999999"
5° tentativo: "3rNe99999"
6° tentativo: "3rNes9999"
7° tentativo: "3rNesT999"
8° tentativo: "3rNesT099"
9° tentativo: "3rNesT0?9"
10° tentativo: "3rNesT0?!"

Al decimo tentativo, tutti i controlli sono andati a buon fine, e successivamente è stata invocata l'API ExitWindowsEx che ha causato lo spegnimento del sistema.

Avendo così capito lo scopo della funzione FUN_004050c0, ho ridenominato quest'ultima in check_and_shut_down.

In definitiva, il codice di sblocco che consente a hw3.exe di funzionare correttamente è “3rNesT0?!”.

6) VERIFICA DEL RISULTATO

Possiamo già dire che il risultato trovato è corretto grazie all’invocazione dell’API ExitWindowsEx nel momento in cui è stato inserito il codice di sblocco “3rNesT0?!” all’interno dell’apposita finestra dell’applicazione.