DOCUMENTO SUL SECONDO HOMEWORK – MATTEO FANFARILLO

Il secondo homework consiste nel trovare, mediante gli strumenti disassemblatore e decompilatore di Ghidra, il codice segreto che rende funzionante il programma hw2.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 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 con Ghidra, utilizzando il disassemblatore e il decompilatore, il programma, in modo tale da ottenere il codice segreto che permette di sbloccare la funzionalità del programma stesso.

3) OTTENIMENTO E DISASSEMBLAGGIO DEL CODICE MACCHINA

Poiché, almeno all'inizio, non è stato necessario decriptare o deoffuscare il codice, è stato sufficiente caricare il file eseguibile su Ghidra, il quale ha provveduto al disassemblaggio, generando così il codice assembly su cui si baserà il resto dell'analisi.

4) LOCALIZZAZIONE DEI FRAMMENTI ASSEMBLY DI INTERESSE

L'obiettivo di questa fase è trovare i frammenti assembly relativi al codice scritto dal programmatore. Poiché abbiamo a che fare con una normale applicazione Windows, 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 potrebbe appartenere al WinMain. Si può avere conferma del fatto che si tratta effettivamente del WinMain andando a verificare che è una delle ultime funzioni invocate da entry.
Constatato ciò, ho impostato il prototipo della funzione:
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.

VirusTotal prende in input un file e controlla se ne ha già memorizzata la firma hash. Se sì, vuol dire che il file era già stato analizzato in precedenza e VirusTotal restituisce direttamente i risultati; altrimenti, il file viene fatto analizzare a 66 antivirus differenti, e i suoi risultati con la sua firma hash vengono memorizzati. Nel caso di hw2.exe, 63 antivirus (quindi la stragrande maggioranza) non hanno rilevato problemi, mentre solo 3 di loro hanno considerato l'eseguibile sospetto: con ogni probabilità, non abbiamo a che fare con un malware, per cui è possibile eseguirlo senza troppi problemi; questo può essere molto utile sia per avere una prima idea su com'è fatta l'applicazione, ma anche nella fase di verifica del risultato dell'analisi.

Ho dunque lanciato hw2.exe (meglio farlo su una macchina virtuale per precauzione) e ho visto che si occupa dello spegnimento automatico del sistema allo scadere di un certo timer impostato dall'utente. All'interno della finestra compare il seguente messaggio: "WARNING: there will be no shutdown without the proper unlock code!". Questa è un'ulteriore conferma di come l'eseguibile funzioni solo con l'inserimento della password corretta.

Dopodiché ho iniziato l'analisi statica vera e propria partendo dal WinMain. Qui ho potuto subito riconoscere la struttura WNDCLASSEXA, che è la struct inizializzata all'inizio del WinMain. Di particolare interesse è il campo lpfnWndProc (il terzo) di tale struttura, che viene inizializzato all'indirizzo della funzione WindowProc (la Window Procedure). Questa funzione è importante perché processa e gestisce i messaggi che vengono inviati a una finestra, per cui mi sono spostato subito su di lei.

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 che parte dall'etichetta LAB_004014e3 (che ho ridenominato CASE_WM_COMMAND).
Qui, per prima cosa, viene effettuato un controllo: se i 16 bit più significativi di wParam sono non nulli, si salta a una parte di codice in cui si invoca subito una return 0 (con rilascio di 16 byte sullo stack).
Poi c'è un controllo analogo sulla variabile locale local_b0 (o meglio, all'offset 184 di local_b0). Per capire a cosa corrisponde local_b0, bisogna andare a vedere a cosa è stato inizializzato prima della jump verso l'etichetta CASE_WM_COMMAND, e si deduce che è pari al valore di ritorno di GetWindowLongA.
Sono andato a recuperare le informazioni di GetWindowLongA nella documentazione della Microsoft, e ho dedotto che si tratta di una funzione che restituisce un valore riguardante una particolare finestra. I parametri sono riassunti nella seguente tabella:

#	TIPO	NOME	DESCRIZIONE	VALORE
1	HWND	hWnd	Handle alla finestra.	hWnd, ovvero l'handle alla finestra corrente (che sarebbe quella principale)
2	int	nIndex	Intero che indica qual è il valore che deve essere restituito.	GWL_USERDATA, ovvero un valore definito dal programmatore

In pratica, local_b0 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. Dopodiché ho analizzato i parametri passati a SetWindowLongA:

#	TIPO	NOME	DESCRIZIONE	VALORE
1	HWND	hWnd	Handle alla finestra.	hWnd, ovvero l'handle alla finestra corrente (che sarebbe quella principale)
2	int	nIndex	Intero che indica qual è il valore che deve essere impostato.	GWL_USERDATA, ovvero un valore definito dal programmatore
3	LONG	dwNewLong	Intero contenente il valore.	Valore di lParam (o del primo campo di lParam)

Ora l'obiettivo è capire a cosa corrisponde lParam nei messaggi di tipo WM_CREATE. Cercando nella documentazione della Microsoft, ho capito che 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. La documentazione riporta che lpCreateParams contiene il valore del parametro lpParam specificato nella chiamata alla funzione CreateWindowEx (o CreateWindow). Cercando i riferimenti a CreateWindowEx, ho dedotto che l'API viene invocata all'interno di WinMain; si può anche notare che il parametro lpParam (l'ultimo di CreateWindowEx) è uguale al valore di ritorno di un'altra funzione: FUN_00401aab.

Dando un primo sguardo a FUN_00401aab, 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_00406010.
Rimettendo insieme i pezzi del puzzle, mi sono accorto che la variabile local_b0 all'interno della WindowProc viene inizializzata esattamente all'indirizzo DAT_00406010 tramite la chiamata a GetWindowLongA. Non solo: local_b0 viene acceduta con degli spiazzamenti costanti, come ad esempio avviene nelle seguenti istruzioni in prossimità dell'etichetta CASE_WM_COMMAND:

```
MOV    EAX, dword ptr [EBP + local_b0]
CMP    EDI, dword ptr [EAX + 0xb8]
```

Questo fa pensare che DAT_00406010 sia l'indirizzo base di una struct, e lo ridenomino APP_STRUCT; local_b0 invece diventerà l_struct (che sta per local struct).

Soffermandoci sulle due istruzioni qui sopra riportate, capiamo che il campo al byte 0xb8 (= 184) della struct viene confrontato con un registro a 4 byte (EDI), 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 predenza 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 alla variabile locale local_a4, che ho ridenominato l_hAppInstance.
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, mentre di seguito sono riportati i parametri passati in input:

#	TIPO	NOME	DESCRIZIONE	VALORE
1	DWORD	dwExStyle	Stile esteso della finestra.	0 (= default)
2	LPCSTR	lpClassName	Classe della finestra.	"EDIT"
3	LPCSTR	lpWindowName	Nome della finestra.	NULL
4	DWORD	dwStyle	Stile della finestra.	0x50802002
5	int	X	Coordinata X iniziale della posizione della finestra.	0
6	int	Y	Coordinata Y iniziale della posizione della finestra.	0
7	int	nWidth	Larghezza della finestra.	0
8	int	nHeight	Altezza della finestra.	0
9	HWND	hWndParent	Handle alla finestra parent.	hWnd
10	HMENU	hMenu	- Se si tratta di una finestra senza parent, è il menù che verrà usato all'interno di essa. - Se si tratta di una finestra child (ed è questo il caso), è il suo identificativo.	Valore di EBX

11	HINSTANCE	hInstance	Handle all'istanza del modulo che verrà associato alla finestra.	l_hAppInstance
12	LPVOID	lpParam	Campo lpCreateParams della struttura CREATESTRUCT.	NULL

All'interno del ciclo (che nel frattempo ho etichettato con CREATE_CHLD_WND_LOOP), l'handle alla finestra appena creata viene memorizzato nel byte 168+4*EBX di APP_STRUCT; dopodiché, il valore del registro EBX (che inizialmente era impostato a 1) viene incrementato di un'unità e viene controllato: nel momento in cui diviene uguale a 4, si esce dal ciclo. Ciò vuol dire che EBX è l'indice all'interno di CREATE_CHL_WND_LOOP e che vengono effettuate in tutto 3 iterazioni, per cui vengono create 3 finestre child.
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. Per tentare di capire di cosa si tratta, sono andato al punto di WinMain in cui avviene la chiamata a init_struct. Purtroppo però non sono riuscito a ottenere particolari informazioni, salvo il fatto che si tratta dell'indirizzo di una variabile locale (DAT_00403000).
- Viene invocata per due volte la funzione FUN_004023c0, 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 ivoca una funzione di libreria, ovvero MSVCRT.DLL::_vsnprintf.
Ciò ci induce a pensare che FUN_004023c0 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 solo invocate due funzioni: FUN_00401b74 e FUN_00401b20; proviamo a darvi uno sguardo.

Per quanto riguarda FUN_00401b74, 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_00401b74 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_00401b74:
- uValue → minutes
- uValue_00 → days
- uValue_01 → hours
- hDlg → hWnd

Inoltre, ho ridenominato la funzione in calculate_time.

Per quanto invece riguarda FUN_00401b20, 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. I parametri di SetTimer sono i seguenti:

#	TIPO	NOME	DESCRIZIONE	VALORE
1	HWND	hWnd	Handle alla finestra che verrà associata al timer.	hWnd, ovvero l'handle alla finestra parent
2	UINT_PTR	nIDEvent	Identificatore del timer.	0
3	UINT	uElapse	Valore iniziale del timeout in millisecondi.	APP_STRUCT.tick_length
4	TIMEPROC	lpTimerFunc	Puntatore alla funzione che verrà attivata allo scadere del timeout.	FUN_00401c7b

Il valore restituito 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

A questo punto possiamo ridenominare la funzione FUN_00401b20 in invoke_SetTimer e possiamo analizzare FUN_00401c7b (che a sua volta ho ridenominato TimerProc).
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 in redraw_flag.
In TimerProc, inoltre, c'è una coppia di istruzioni molto particolare:

```
MOV     dword ptr [ESP]=>fun_arg1, APP_STRUCT
CALL    dword ptr [APP_STRUCT.init_struct_param]
```

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 0x00403000. 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 macchina. In tal modo, è comparsa una funzione non del tutto ben formata: in diversi punti presenta sequenze di istruzioni del tipo:

```
TEST    EAX, EAX
JZ      LAB_ADDRESS+2
CALL    SUBROUTINE
```

L'istruzione CALL SUBROUTINE si trova esattamente nell'indirizzo di memoria indicato da LAB_ADDRESS+2, il che rende insensato il controllo di TEST svolto. Tale anomalia si aggiunge al fatto che la subroutine invocata risulta essere definita in un indirizzo di memoria in realtà non esistente, per cui Ghidra non la riconosce e la segna in rosso.
Tutto ciò fa pensare che le istruzioni generate da Ghidra tramite il comando Disassemble per la funzione FUN_00403000 siano in realtà errate. In particolare, sembra necessario applicare il comando Clear Code Bytes (che è l'inverso di Disassemble) a tutte le sequenze di istruzioni che si presentano come quella sopra riportata, e anche a tutte le istruzioni strane come:
- INC ESP
- AND AL, VALORE_NUMERICO.

Successivamente bisogna istruire Ghidra in modo tale da interpretare i byte relativi alle istruzioni di FUN_00403000 correttamente: dopo aver fatto alcune prove, ho dedotto che le sequenze di byte pari a E8 63 componessero dead code e, conseguentemente, fossero le uniche da non disassemblare nuovamente e da trasformare in delle NOP (= no operation).

Ora il codice assembly risultante è apparentemente sistemato: in particolare, non si presentano più le chiamate a delle subroutine non identificate ma, piuttosto, Ghidra ha riconosciuto delle istruzioni che prima non risultavano. Osservando meglio tali istruzioni, però, notiamo che si ripresenta il seguente pattern:

```
MOV     EAX, [DAT_004060d0]
TEST    EAX, EAX
JZ      LAB_00403019
NOP
        LAB_00403019
MOV     dword ptr [EDX + 0x10], 0x0
                …
```

Come prima, qui abbiamo un salto condizionale verso l'istruzione immediatamente successiva, il che rende del tutto ridondante il controllo effettuato. Perciò, ogni volta che compare questo pattern all'interno della funzione, possiamo sostituire le istruzioni di MOV, TEST e JZ con delle NOP.

Un altro passaggio da effettuare per sistemare FUN_00403000 consiste nell'aggiungere delle variabili locali sullo stack che, finora, mancavano. Per far ciò, è sufficiente lanciare il comando Analyze Stack affinché Ghidra tenti di correggere lo stack automaticamente.
Successivamente si può iniziare a correggere la segnatura della funzione, specificando che il suo unico parametro di input è di tipo struct app_struct * e attribuendole provvisoriamente il nome di funzione_sistemata.

Una volta concluso tutto questo lavoro, funzione_sistemata si presenta nel seguente modo (per brevità ho omesso gli indirizzi e i byte relativi alle istruzioni macchina che compaiono alla sinistra di ogni riga):

```
undefined __stdcall funzione_sistemata (struct app_struct * app_struct)

undefined               AL:1            <RETURN>
struct app_struct *     Stack[0x4]:4    app_struct
undefined1              Stack[-0x2e]:1  local_2e
```

```
undefined1              Stack[-0x2f]:1   local_2f
undefined1              Stack[-0x30]:1   local_30
undefined1              Stack[-0x31]:1   local_31
undefined1              Stack[-0x32]:1   local_32
undefined1              Stack[-0x33]:1   local_33
undefined1              Stack[-0x34]:1   local_34
undefined1              Stack[-0x35]:1   local_35
undefined1              Stack[-0x36]:1   local_36
undefined4              Stack[-0x50]:4   fun_arg4
undefined4              Stack[-0x54]:4   fun_arg3
undefined4              Stack[-0x58]:4   fun_arg2
undefined4              Stack[-0x5c]:4   fun_arg1

                        funzione_sistemata
        PUSH    EBX
        SUB     ESP, 0x58
        MOV     EDX, dword ptr [ESP + app_struct]
        MOV     ECX, dword ptr [EDX + 168]
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        MOV     dword ptr [EDX + 16], 0
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        LEA     EAX=>local_36, [ESP + 0x26]
        MOV     dword ptr [ESP + fun_arg4], 30
        MOV     dword ptr [ESP + fun_arg3], EAX
        MOV     dword ptr [ESP + fun_arg2], 5
        MOV     dword ptr [ESP]=>fun_arg1, ECX
        CALL    USER32.DLL::GetDlgItemTextA
        SUB     ESP, 0x10
        MOV     EDX, EAX
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        CMP     EDX, 9
        JZ      LAB_00403075

                        LAB_00403060
        NOP
```

```
NOP
NOP
NOP
NOP
NOP
CALL    FUN_00401ff0

        LAB_00403070
ADD     ESP, 0x58
POP     EBX
RET

LAB_00403075
CMP     byte ptr [ESP + local_36], '3'
JNZ     LAB_00403060
CMP     byte ptr [ESP + local_35], 'R'
JNZ     LAB_00403060
CMP     byte ptr [ESP + local_34], 'n'
JNZ     LAB_00403060
CMP     byte ptr [ESP + local_33], 'E'
NOP
JNZ     LAB_00403060
CMP     byte ptr [ESP + local_32], 'S'
JNZ     LAB_00403060
CMP     byte ptr [ESP + local_31], 't'
NOP
JNZ     LAB_00403060
CMP     byte ptr [ESP + local_30], '0'
JNZ     LAB_00403060
CMP     byte ptr [ESP + local_2f], '!'
NOP
JNZ     LAB_00403060
CMP     byte ptr [ESP + local_2e], '?'
JNZ     LAB_00403060
NOP
NOP
NOP
NOP
NOP
NOP
CALL    KERNEL32.DLL::GetCurrentProcess
LEA     EDX, [ESP + 0x54]
MOV     dword ptr [ESP + 0x8], EDX
MOV     dword ptr [ESP + 0x4], 0x28
MOV     dword ptr [ESP], EAX
CALL    ADVAPI32.DLL::OpenProcessToken
SUB     ESP, 0xc
TEST    EAX, EAX
JNZ     LAB_004030f7
MOV     dword ptr [ESP], 0x0
```

```
CALL    USER32.DLL::PostQuitMessage
SUB     ESP, 0x4
NOP
NOP
NOP
NOP
NOP
NOP
LEA     EBX, [ESP + 0x44]
LEA     EAX, [ESP + 0x48]
MOV     dword ptr [ESP + 0x8], EAX
MOV     dword ptr [ESP + 0x4], s_SeShutdownPrivilege_004051be
MOV     dword ptr [ESP], 0x0
CALL    ADVAPI32.DLL::LookupPrivilegeValueA
SUB     ESP, 0xc
NOP
NOP
NOP
NOP
NOP
NOP
MOV     dword ptr [ESP + 0x44], 0x1
MOV     dword ptr [ESP + 0x50], 0x2
MOV     dword ptr [ESP + 0x14], 0x0
MOV     dword ptr [ESP + 0x10], 0x0
MOV     dword ptr [ESP + 0xc], 0x0
MOV     dword ptr [ESP + 0x8], EBX
MOV     dword ptr [ESP + 0x4], 0x0
MOV     EAX, dword ptr [ESP + 0x54]
MOV     dword ptr [ESP], EAX
CALL    AdjustTokenPrivileges
SUB     ESP, 0x18
NOP
NOP
NOP
NOP
NOP
NOP
CALL    KERNEL32.DLL::GetLastError
TEST    EAX, EAX
JNZ     LAB_004031a9
NOP
NOP
NOP
NOP
NOP
NOP
MOV     dword ptr [ESP + 0x4], 0x0
MOV     dword ptr [ESP], 0x5
```

```
CALL    USER32.DLL::ExitWindowsEx
SUB     ESP, 0x8

        LAB_004031a9
NOP
NOP
NOP
NOP
NOP
NOP
MOV     dword ptr [ESP], 0x0
CALL    USER32.DLL::PostQuitMessage
SUB     ESP, 0x4
JMP     LAB_00403070
```

Ora la funzione ha effettivamente tutta l'aria di essere ben formata e può essere analizzata correttamente.

Inizialmente, il campo redraw_flag di APP_STRUCT viene impostato a 0 e viene invocata la funzione GetDlgItemTextA, che memorizza il contenuto di una data finestra di dialogo all'interno di un buffer.
I parametri sono di seguito riportati:

#	TIPO	NOME	DESCRIZIONE	VALORE
1	HWND	hDlg	Handle alla finestra di dialogo.	hWnd, ovvero l'handle alla finestra parent
2	int	nIDDlgItem	Identificatore del controllo (= della finestra child) il cui testo dovrà essere recuperato.	5
3	LPSTR	lpString	Buffer in cui verrà memorizzato il testo.	[out] Indirizzo di local_36
4	int	cchMax	Dimensione massima della stringa che verrà memorizzata nel buffer.	30

Il valore restituito da GetDlgItemTextA è il numero di caratteri copiati nel buffer, escluso '\0'. Se questo valore è uguale a 9, allora si salta verso l'etichetta LAB_00403075; altrimenti si accede alla zona di codice etichettata da LAB_00403060, in cui viene semplicemente invocata la funzione FUN_00401ff0 prima dell'istruzione di return.
Analizziamo dapprima la porzione di codice corrispondente a LAB_00403075. Qui viene effettuato un controllo carattere per carattere tra il contenuto del buffer di indirizzo local_36 e una particolare stringa, "3RnESt0!?". Al primo carattere che non corrisponde, si salta verso LAB_00403060.
Comunque sia, ho ritenuto opportuno ridenominare local_36 in pw_buff e impostare il suo tipo a char[30]: effettivamente, "3RnESt0!?" ha tutta l'aria di essere la password di sblocco della nostra applicazione.
Inoltre, ho ridenominato l'etichetta LAB_00403075 in CMP_STR e l'etichetta LAB_00403060 in NO_MATCH.

Se il confronto tra "3RnESt0!?" e la stringa contenuta in pw_buff va a buon fine, si procede con le istruzioni successive e, in particolare, si invoca GetCurrentProcess, che è una funzione che non accetta parametri in input e che restituisce semplicemente uno (pseudo-)handle al processo corrente.

Arrivato a questo punto, mi sono accorto che Ghidra, all'interno delle istruzioni MOV e LEA, non fornisce più i riferimenti espliciti sullo stack in funzione delle variabili locali, ma devo calcolarli io al suo posto. A tal proposito, potrebbe essere utile tenere traccia di una tabella che descriva accuratamente il posizionamento delle variabili locali di funzione_sistemata sullo stack:

OFFSET	LENGTH	DATA TYPE	NAME
-0x5c	4	dword	fun_arg1
-0x58	4	dword	fun_arg2
-0x54	4	dword	fun_arg3
-0x50	4	dword	fun_arg4
-0x4c	1	undefined	
-0x4b	1	undefined	
-0x4a	1	undefined	
-0x49	1	undefined	
-0x48	1	undefined	
-0x47	1	undefined	
-0x46	1	undefined	
-0x45	1	undefined	
-0x44	1	undefined	
-0x43	1	undefined	
-0x42	1	undefined	
-0x41	1	undefined	
-0x40	1	undefined	
-0x3f	1	undefined	
-0x3e	1	undefined	
-0x3d	1	undefined	
-0x3c	1	undefined	
-0x3b	1	undefined	
-0x3a	1	undefined	
-0x39	1	undefined	
-0x38	1	undefined	
-0x37	1	undefined	
-0x36	30	char[30]	pw_buff
-0x18	1	undefined	
-0x17	1	undefined	
-0x16	1	undefined	
-0x15	1	undefined	
-0x14	1	undefined	
-0x13	1	undefined	
-0x12	1	undefined	
-0x11	1	undefined	
-0x10	1	undefined	
-0xf	1	undefined	
-0xe	1	undefined	
-0xd	1	undefined	
-0xc	1	undefined	
-0xb	1	undefined	
-0xa	1	undefined	
-0x9	1	undefined	
-0x8	1	undefined	
-0x7	1	undefined	
-0x6	1	undefined	
-0x5	1	undefined	
-0x4	1	undefined	
-0x3	1	undefined	
-0x2	1	undefined	

-0x1	1	undefined	
0x0	1	undefined	
0x1	1	undefined	
0x2	1	undefined	
0x3	1	undefined	
0x4	4	struct app_struct *	app_struct

Sappiamo che il registro ESP punta all'indirizzo sullo stack più piccolo e, quindi, a quello con offset -0x5c.

Dopo GetCurrentProcess, all'interno di funzione_sistemata viene invocato OpenProcessToken, che apre l'access token associato al processo specificato.
L'access token contiene le informazioni di sicurezza per una sessione di accesso, e identifica l'utente, i suoi gruppi e i suoi privilegi. Il sistema usa il token per controllare l'accesso a determinati oggetti e i permessi dell'utente sull'invocazione di varie operazioni di sistema.
OpenProcessToken restituisce un valore non nullo in caso di successo, 0 altrimenti; i suoi parametri sono qui riportati:

#	TIPO	NOME	DESCRIZIONE	VALORE
1	HANDLE	ProcessHandle	Handle al processo.	Valore restituito da GetCurrentProcess.
2	DWORD	DesiredAccess	Access mask che specifica i tipi di accesso richiesti all'access token.	L'access mask esprimibile col numero 40.
3	PHANDLE	TokenHandle	Puntatore all'handle che riceverà l'access token.	[out] Indirizzo ESP + 0x54, che corrisponde all'offset -0x8 dello stack.

Se la chiamata a OpenProcessToken non va a buon fine, viene invocato PostQuitMessage, che consiste nella richiesta di terminazione da parte di un thread. In caso contrario, si salta direttamente alle istruzioni successive a PostQuitMessage, che preparano all'invocazione di LookupPrivilegeValueA. Quest'ultima è una funzione che recupera il LUID (locally unique identifier) usato sul sistema specificato per rappresentare un certo privilegio. Il valore di ritorno è non nullo in caso di successo, 0 altrimenti; i suoi parametri sono qui riportati:

#	TIPO	NOME	DESCRIZIONE	VALORE
1	LPCSTR	lpSystemName	Nome del sistema.	NULL (= sistema locale)
2	LPCSTR	lpName	Nome del privilegio.	"SeShutdownPrivilege"
3	PLUID	lpLuid	Puntatore all'handle che riceverà il LUID associato al privilegio specificato.	[out] Indirizzo ESP + 0x48, che corrisponde all'offset -0x14 dello stack.

Dopodiché, il valore all'indirizzo ESP + 0x44 (che corrisponde all'offset -0x18 dello stack) viene impostato a 1, mentre il valore all'indirizzo ESP + 0x50 (che corrisponde all'offset -0xc dello stack) viene impostato a 2. Segue l'invocazione alla funzione AdjustTokenPrivileges, che abilita (o disabilita) i privilegi specificati nell'access token, e restituisce un valore non nullo in caso di successo, 0 altrimenti; i suoi parametri sono:

#	TIPO	NOME	DESCRIZIONE	VALORE
1	HANDLE	TokenHandle	Handle all'access token.	TokenHandle (che era stato fornito da OpenProcessToken)
2	BOOL	DisableAllPrivileges	Booleano che specifica se si devono disabilitare tutti i privilegi dell'access token.	FALSE

3	PTOKEN_PRIVILEGES	NewState	Puntatore a una struttura TOKEN_PRIVILEGES che contiene un array di privilegi e i loro attributi. Se DisableAllPrivileges == FALSE, la funzione opererà solo su questi privilegi, i quali verranno abilitati, disabilitati o rimossi in base al valore dei loro attributi.	Indirizzo ESP + 0x44 (che corrisponde all'offset -0x18 dello stack)
4	DWORD	BufferLength	Dimensione in byte del buffer puntato da PreviousState.	0
5	PTOKEN_PRIVILEGES	PreviousState	Puntatore a un buffer che riceverà una struttura TOKEN_PRIVILEGES che conterrà lo stato precedente dei privilegi.	[out] NULL
6	PDWORD	ReturnLength	Puntatore a una variabile che riceverà la dimensione in byte richiesta del buffer puntato da PreviousState.	[out] NULL

Analizzando AdjustTokenPrivileges, scopriamo che all'offset -0x18 dello stack inizia una struttura TOKEN_PRIVILEGES, che è definita nel seguente modo:

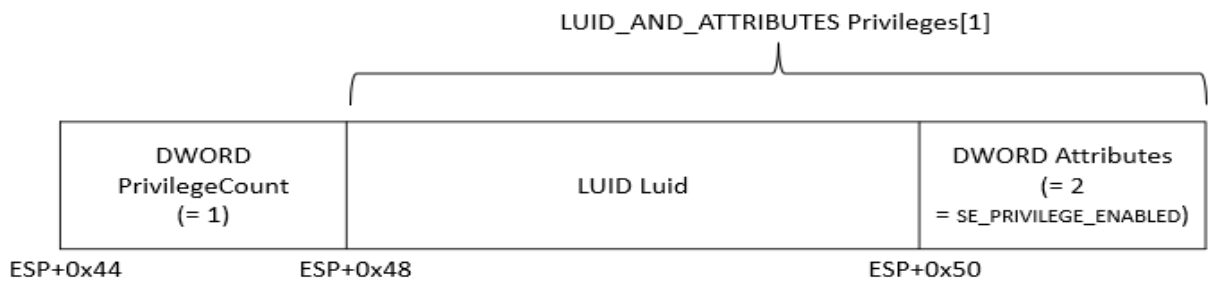
```
typedef struct _TOKEN_PRIVILEGES {
        DWORD                   PrivilegeCount;
        LUID_AND_ATTRIBUTES  Privileges[ANYSIZE_ARRAY];
} TOKEN_PRIVILEGES, *PTOKEN_PRIVILEGES;
```

Il campo PrivilegeCount indica il numero di entry all'interno dell'array Privileges. Essendo il primo campo della struttura, si trova all'offset -0x18 dello stack: ciò vuol dire che la seguente istruzione:
`MOV dword ptr [ESP + 0x44], 1` ha imposto che Privileges debba avere una sola entry.
Se ci facciamo caso, Privileges è un array di strutture LUID_AND ATTRIBUTES, che sono definite nel seguente modo:

```
typedef struct _LUID_AND_ATTRIBUTES {
        LUID            Luid;
        DWORD           Attributes;
} LUID_AND_ATTRIBUTES, *PLUID_AND_ATTRIBUTES;
```

Si tratta di una struttura composta da un valore e un attributo riguardanti un determinato privilegio. Poiché il tipo LUID comprende valori a 64 bit, abbiamo il campo Luid all'offset -0x14 e il campo Attributes all'offset -0xc dello stack.

In definitiva, la nostra struttura TOKEN_PRIVILEGES si presenta così:



Da qui pare evidente che, tramite AdjustTokenPrivileges, se tutto è andato a buon fine, è stato abilitato il privilegio “SeShutdownPrivilege” all’interno del sistema locale. È possibile però che il privilegio rimanga disabilitato anche a seguito della chiamata ad AdjustTokenPrivileges. Per controllare ciò, si invoca anche GetLastError che, nel nostro caso specifico, restituisce 0 se il privilegio è stato affettivamente abilitato, un valore differente (corrispondente alla macro ERROR_NOT_ALL_ASSIGNED) altrimenti. Nel primo caso, funzione_sistemata chiama ExitWindowsEx (che porta effettivamente allo spegnimento della macchina), PostQuitMessage e dopo termina; nel secondo caso, invece, si salta direttamente all’invocazione di PostQuitMessage prima della terminazione.

Ci rimane ora da analizzare la funzione FUN_00401ff0, che viene invocata nel caso in cui la stringa inserita all’interno della finestra child di hWnd con identificativo pari a 5 non corrisponda a “3RnESt0!?”.
È una funzione semplicissima: invoca MessageBoxA, PostQuitMessage e poi termina.
MessageBoxA è un’API che mostra a schermo una finestra di dialogo e che accetta i seguenti parametri:

#	TIPO	NOME	DESCRIZIONE	VALORE
1	HWND	hWnd	Handle alla finestra che includerà la message box.	NULL
2	LPCTSTR	lpText	Messaggio da mostrare.	"Shutdown time has come!\n\nHowever, the unlock code is wrong.\n\nIf you want the full version of this wonderful tool,\nyou can get the unlock code for just ten bucks!\n\nAsk to the nearest teacher around you!"
3	LPCTSTR	lpCaption	Titolo della finestra di dialogo.	“Sorry, try again!”
4	UINT	uType	Contenuto e comportamento della finestra di dialogo.	MB_ICONERROR

A questo punto possiamo fare le seguenti considerazioni:

I) funzione_sistemata può essere ridenominata in check_pw_and_exit, mentre FUN_00401ff0 può essere ridenominata in show_error_msg.

II) Lo stack, relativamente a check_pw_and_exit, possiamo vederlo come segue:

OFFSET	LENGTH	DATA TYPE	NAME
-0x5c	4	dword	fun_arg1
-0x58	4	dword	fun_arg2
-0x54	4	dword	fun_arg3
-0x50	4	dword	fun_arg4
-0x4c	4	dword	fun_arg5
-0x48	4	dword	fun_arg6
-0x44	1	undefined	
-0x43	1	undefined	
-0x42	1	undefined	
-0x41	1	undefined	
-0x40	1	undefined	
-0x3f	1	undefined	
-0x3e	1	undefined	
-0x3d	1	undefined	
-0x3c	1	undefined	
-0x3b	1	undefined	

-0x3a	1	undefined	
-0x39	1	undefined	
-0x38	1	undefined	
-0x37	1	undefined	
-0x36	30	char[30]	pw_buff
-0x18	16	TOKEN_PRIVILEGES	local_esp44
-0x8	4	PHANDLE	local_esp54
-0x4	1	undefined	
-0x3	1	undefined	
-0x2	1	undefined	
-0x1	1	undefined	
0x0	1	undefined	
0x1	1	undefined	
0x2	1	undefined	
0x3	1	undefined	
0x4	4	struct app_struct *	app_struct

III) La struttura APP_STRUCT può in definitiva essere vista così:

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	redraw_flag
20	4	addr	pointer	check_pw_fun
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

IV) “3RnESt0!?” è effettivamente il codice di sblocco (ovvero la password) che consente alla nostra applicazione di funzionare correttamente.

6) VERIFICA DEL RISULTATO

Per provare la correttezza dell’analisi e della password trovata, ho avviato l’applicazione, fatto partire il timer e inserito la stringa “3RnESt0!?” all’interno dell’apposita finestra. Poiché, allo scadere del timeout, il sistema è andato effettivamente in shutdown, si può dire che la verifica del risultato ha prodotto un esito positivo.