DOCUMENTO SUL PRIMO HOMEWORK – MATTEO FANFARILLO

Il primo homework consiste nell'effettuare il Reverse Code Engineering sul programma eseguibile hw1.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 di dati impiegate.
6) Analisi dei risultati e 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, non conoscevo nulla riguardo il suo funzionamento, ma sapevo comunque che si sarebbe trattato di un programma Windows a 32 bit scritto con il linguaggio C.

2) FORMALIZZAZIONE DELL'OBIETTIVO

L'obiettivo dell'homework è analizzare con Ghidra, utilizzando il disassemblatore e il decompilatore, il programma, raccogliendo quante più informazioni possibili sul suo funzionamento e, in particolare, sulle strutture di dati usate. Da qui si deduce che:
- è opportuno effettuare l'analisi white-box del programma;
- è necessario concentrarsi su tutta e sola la porzione di programma implementata dallo sviluppatore.

3) OTTENIMENTO E DISASSEMBLAGGIO DEL CODICE MACCHINA

Poiché non è stato necessario decriptare o deoffuscare il codice, questo passaggio è risultato banale: è stato sufficiente caricare il file eseguibile su Ghidra, il quale ha provveduto al disassemblaggio, generando così il codice assembly su cui si baserà tutto il resto dell'analisi.
Da qui, inoltre, ho potuto ricavare che il file sorgente di partenza è stato compilato con gcc.

4) LOCALIZZAZIONE DEI FRAMMENTI ASSEMBLY DI INTERESSE

L'obiettivo principale di questa fase è trovare i frammenti assembly relativi al codice scritto dal programmatore e, in particolare, riconoscere la funzione main. Per far ciò, ho adottato la seguente strategia:

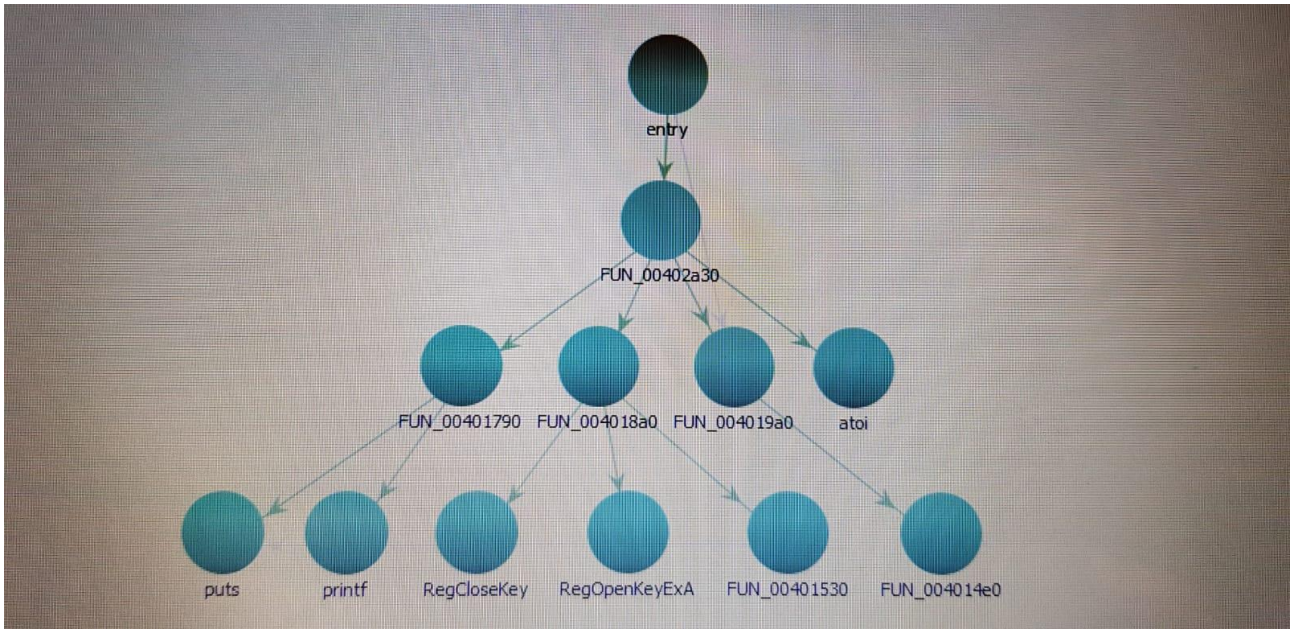
- A partire dalla sezione Symbol Tree di Ghidra, ho selezionato la cartella Imports e poi MSVCRT.DLL, che è il modulo contenente le funzioni di libreria C Standard, tipicamente invocate dal programmatore. Di queste funzioni ho preso in considerazione in particolar modo printf e puts, che sono molto facilmente utilizzate dallo sviluppatore e molto difficilmente invocate dalla funzione entry o da una qualunque funzione atta a preparare l'ambiente di esecuzione.

- All'interno della sezione Listing: hw1.exe di Ghidra, ho raggiunto la definizione di printf, da cui ho selezionato References -> Show references to printf. Da qui ho visto che la funzione printf è stata invocata sette volte.

- Ho selezionato una delle sette locazioni in cui è stata chiamata la printf e ho notato che tale locazione si trova all'interno della funzione FUN_00401790. Osservando meglio, si deduce che FUN_00401790 chiama printf tutte e sette le volte e, dunque, è con ogni probabilità una funzione definita dal programmatore.

- Ho ripetuto lo stesso procedimento per puts che, alla fine dei conti, è stata invocata da FUN_00401790 ma anche da FUN_00401530. Anche quest'ultima è dunque quasi certamente una funzione definita dal programmatore.

- Ho sfruttato la finestra Function Call Graph per avere una visione d'insieme delle chiamate a funzione, e mi sono concentrato su FUN_00401790 e FUN_00401530.



Da qui ho dedotto che l'antenato comune a queste due funzioni è FUN_00402a30.

- Sono andato a dare uno sguardo a FUN_00402a30 e ho notato che si tratta di una funzione che invoca atoi; quest'ultima, come printf e puts, viene tipicamente utilizzata dal programmatore. Inoltre, come si può anche vedere dal grafo delle chiamate a funzione, FUN_00402a30 viene invocata soltanto da entry: ha tutta l'aria di la funzione main.

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

A questo punto non ci resta che capire il funzionamento del programma.
Dando un primo sguardo alla funzione main, balza subito all'occhio che, sotto determinate condizioni, vengono passati alla funzione FUN_004018a0 dei parametri molto particolari: il valore 0x80000002 e la stringa "SYSTEM\\ControlSet001\\Control".
Per questo motivo, sono andato a visionare FUN_004018a0 e ho notato che questa, a sua volta, oltre alla funzione FUN_00401530, invoca due particolari API: RegOpenKeyExA e RegCloseKey. La prima apre una registry key, mentre la seconda la chiude.
Effettivamente, anche FUN_00401530 invoca delle funzioni riguardanti le registry key: RegQueryInfoKeyA, RegEnumKeyExA e RegEnumValueA.
Per capire meglio di cosa stiamo parlando, sono andato a recuperare tutte le informazioni utili riguardanti le registry key, che riporterò nella seguente tabella.

WINDOWS REGISTRY	È un insieme di database che memorizzano la maggior parte delle impostazioni e delle informazioni di programmi software, dispositivi hardware, preferenze di utente e configurazione del sistema operativo Windows.
REGISTRY HIVE	È la sezione principale del Windows Registry che contiene registry keys e si trova in cima alla gerarchia nel Windows Registry (root).
REGISTRY KEY	Può essere pensata come una cartella di file ma esiste solo nel Windows Registry. Contiene i registry values così come le cartelle contengono file. Può anche contenere altre registry keys, che sono spesso indicate come subkeys. Anche un registry hive è una cartella nel Windows Registry. L'unica differenza sta nel fatto che un registry hive è la prima cartella nel Windows Registry e contiene solo registry keys.
REGISTRY VALUE	Si trova all'interno di una registry key e contiene specifiche istruzioni a cui Windows e le applicazioni fanno riferimento (e.g. impostazioni sul funzionamento della tastiera).

Acquisite queste informazioni, possiamo approfondire le istruzioni della funzione main. Notiamo che inizialmente vengono utilizzati i parametri di input che, con ogni probabilità, sono argc (un intero caricato nel registro ESI) e argv (un doppio puntatore a carattere caricato nel registro EBX). In particolare, si effettuano due controlli con due salti condizionati:

```
MOV    EAX, dword ptr [EBX + ESI*4]
TEST   EAX, EAX
JZ     LAB_00402aac
CMP    ESI, 2
JLE    LAB_00402a9c
```

Banalmente, l'etichetta LAB_00402aac è relativa a una porzione di codice che si traduce con "return 0" (per cui l'ho ridenominata RETURN_0), mentre l'etichetta LAB_00402a9c è relativa a una porzione di codice che non fa altro a impostare a NULL i valori di argv[1] e argv[2]. Di conseguenza, il primo controllo si traduce nel seguente modo:

```
if (argv[argc] == NULL) return 0;
```

D'altra parte, il secondo controllo si traduce nel seguente altro modo:

```
if (argc <= 2) {
        argv[1] = NULL;
        argv[2] = NULL;
}
```

Per quanto riguarda il primo controllo, notiamo che in realtà argv[argc] è sempre NULL. Infatti, il valore di argc è pari al numero di entry valide dell'array argv, e tali entry vanno da argv[0] a argv[argc-1]. Perciò, di fatto, il programma non fa nulla: semplicemente termina con codice 0 (dopo che il main ha restituito 0 al chiamante, ovvero alla entry). Tuttavia, noi fingeremo che questo controllo non esista e vedremo quale sarebbe stato il comportamento del programma senza di esso.

Successivamente nel main si ha una chiamata alla funzione atoi con parametro argv[1] e un controllo sul valore restituito da atoi (che è pari al registro EAX). Se tale valore è 0, allora EAX viene aggiornato al valore 0x80000002. Inoltre, il registro EDX viene inizializzato al valore di argv[2] e anch'esso è soggetto a un controllo: in particolare, se è NULL, viene aggiornato alla stringa "SYSTEM\\ControlSet001\\Control".

Dopodiché, i valori dei registri EAX e EDX vengono passati come parametri alla funzione FUN_004_02a30, che certamente è stata scritta dal programmatore, per cui l'ho momentaneamente ridenominata f1.

f1 invoca subito RegOpenKeyExA dopo averne sistemato i parametri sullo stack. Ricordandoci che lo stack del processore Intel è full descending e facendoci aiutare dalla documentazione ufficiale della Microsoft, siamo in grado di stabilire quali sono i parametri di RegOpenKeyExA e quali sono i relativi valori:

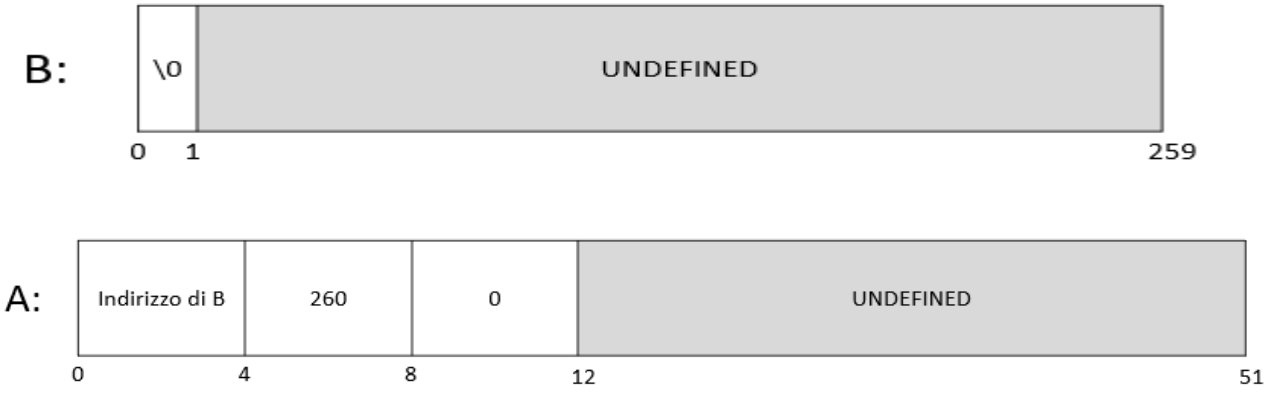
#	TIPO	NOME	DESCRIZIONE	VALORE
1	HKEY	hkey	Handle a una registry key già aperta.	Valore assunto dal primo parametro di f1
2	LPCSTR	lpSubKey	Nome della registry subkey da aprire.	Valore assunto dal secondo parametro di f1
3	DWORD	ulOptions	Opzioni da applicare quando si apre la registry key.	0
4	REGSAM	samDesired	Maschera che indica i diritti di accesso desiderati per la registry subkey da aprire.	0xf003f = KEY_ALL_ACCESS
5	PHKEY	phkResult	Puntatore a una variabile che riceverà un handle alla registry subkey da aprire.	[out] Indirizzo di una variabile locale hkResult

In Ghidra ho potuto vedere che il valore 0x80000002 dell'handle a una registry key già aperta corrisponde alla macro HKEY_LOCAL_MACHINE, che identifica un registry hive contenente:
- la maggior parte delle informazioni di configurazione per il software installato, nonché per il sistema operativo Windows stesso;
- molte configurazioni riguardanti l'hardware e i driver attualmente identificati;
- le informazioni sulla configurazione del boot.

Effettuando un'ulteriore ricerca, ho imparato che HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control è una registry key contenente le informazioni necessarie per controllare lo startup del sistema e alcuni aspetti di configurazione del dispositivo.

Tornando all'analisi del codice, dopo l'invocazione di RegOpenKeyExA, viene valutato il suo valore di ritorno (0 in caso di successo, un valore differente altrimenti) e, se è diverso da 0, f1 restituisce 0 (o NULL); in caso contrario, viene invocata la funzione FUN_00401530, che ho momentaneamente ridenominato f2, che accetta come parametro l'handle alla registry key aperta da RegOpenKeyExA (ovvero hkResult).

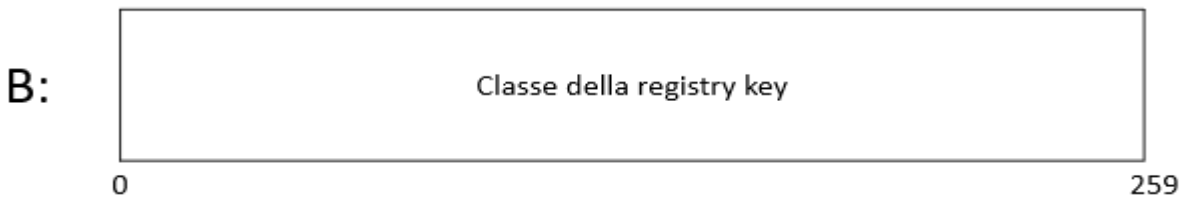
In f2 entrano in gioco le strutture di dati fondamentali utilizzate nel programma, tutte memorizzate nell'heap a partire dall'invocazione della funzione malloc.
Le prime due, che chiameremo provvisoriamente A e B, vengono istanziate dall'esecuzione rispettivamente di malloc(52) e di malloc(260). Se una chiamata a malloc fallisce (e questo vale anche successivamente), f2 termina stampando con puts un messaggio di errore ("Memory allocation error") e invocando exit(1).
Comunque sia, solo alcuni byte di A e B vengono inizializzati subito e, in particolare:



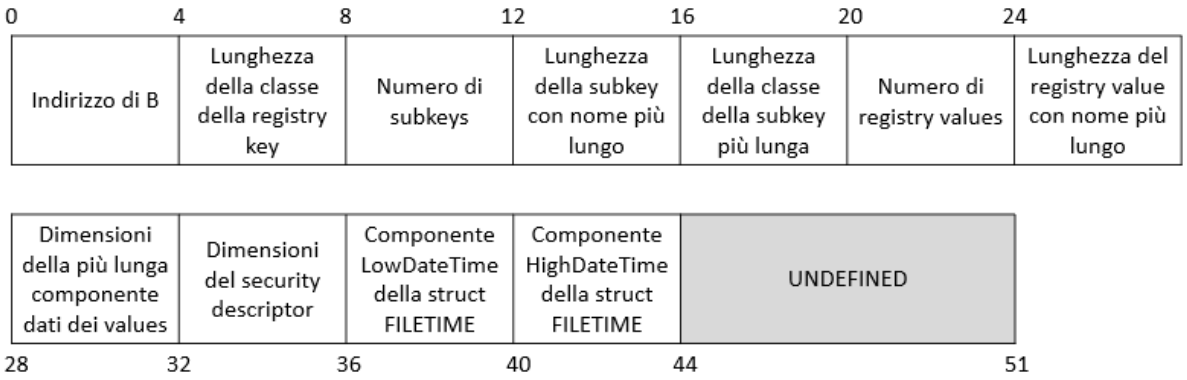
A questo punto viene invocata la funzione RegQueryInfoKeyA di cui, come al solito, sono andato a recuperare le caratteristiche nella documentazione della Microsoft. In particolare, il suo scopo è ottenere le informazioni riguardanti la registry key specificata, mentre il suo valore di ritorno è 0 in caso di successo, un valore differente altrimenti. I parametri sono riassunti nella seguente tabella:

#	TIPO	NOME	DESCRIZIONE	VALORE
1	HKEY	hKey	Handle a una registry key aperta.	Valore assunto dal parametro di f2
2	LPSTR	lpClass	Puntatore a un buffer che riceverà la classe della registry key.	[out] Indirizzo di B
3	LPDWORD	lpcchClass	Puntatore a una variabile che indica le dimensioni del buffer puntato da lpClass; dopo l'esecuzione della funzione, tale variabile conterrà la lunghezza della stringa memorizzata nel buffer.	[in, out] (Indirizzo di A) + 4
4	LPDWORD	lpReserved	Parametro riservato.	NULL
5	LPDWORD	lpcSubKeys	Puntatore a una variabile che riceverà il numero di subkeys contenute in hKey.	[out] (Indirizzo di A) + 8 byte
6	LPDWORD	lpcbMaxSubKeyLen	Puntatore a una variabile che riceverà le dimensioni della subkey col nome più lungo.	[out] (Indirizzo di A) + 12 byte
7	LPDWORD	lpcbMaxClassLen	Puntatore a una variabile che riceverà la lunghezza della stringa più lunga che descrive la classe di una subkey.	[out] (Indirizzo di A) + 16 byte
8	LPDWORD	lpcValues	Puntatore a una variabile che riceverà il numero di registry values associati a hKey.	[out] (Indirizzo di A) + 20 byte
9	LPDWORD	lpcbMaxValueNameLen	Puntatore a una variabile che riceverà la lunghezza del registry value col nome più lungo.	[out] (Indirizzo di A) + 24 byte
10	LPDWORD	lpcbMaxValueLen	Puntatore a una variabile che riceverà le dimensioni della più lunga componente dati dei registry value.	[out] (Indirizzo di A) + 28 byte
11	LPDWORD	lpcbSecurityDescriptor	Puntatore a una variabile che riceverà le dimensioni del security descriptor di hKey.	[out] (Indirizzo di A) + 32 byte
12	PFILETIME	lpftLastWriteTime	Puntatore a una struttura FILETIME che riceverà l'istante di tempo dell'ultimo aggiornamento di hKey.	[out] (Indirizzo di A) + 36 byte

Se la chiamata a RegQueryInfoKeyA fallisce, f2 termina stampando con puts un messaggio di errore ("RegQueryInfoKey failed: key not found") e restituendo 0 (o NULL).
Se invece va a buon fine, le nostre strutture di dati si presenteranno così:



A:



Dopodiché, viene inizializzata una variabile locale lpftLastWriteTime a (indirizzo di A) + 36 byte, e viene effettuato un controllo sul terzo campo di A (ovvero sul numero di subkeys della nostra registry key): se questo è uguale a 0, si salta verso LAB_004016a0 (etichetta che ho ridenominato NO_SUBKEYS); altrimenti viene inizializzata un'altra variabile locale fun_ret al valore di ritorno di RegQueryInfoKeyA (quindi a 0) e si salta verso LAB_00401604. Notiamo che, successivamente a quest'ultima etichetta, è presente un salto condizionato che riporta indietro esattamente all'istruzione di indirizzo 0x00401604: siamo con ogni probabilità entrati all'interno di un ciclo. Per tale ragione, ho ridenominato l'etichetta in LOOP.
È conveniente notare che, subito prima del ciclo, i registri EDI e ESI vengono azzerati.

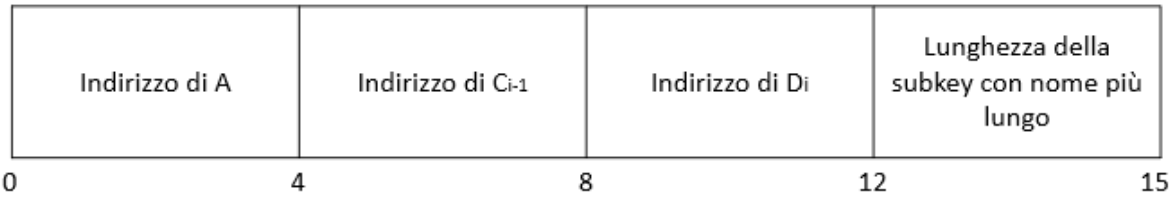
All'interno del ciclo LOOP, vengono dapprima istanziate due ulteriori strutture C_i e D_i (dove i è l'indice dell'iterazione nel ciclo) tramite l'esecuzione rispettivamente di malloc(16) e di malloc(L), dove L è il valore contenuto nel quarto campo di A e, quindi, è pari alla lunghezza della subkey con nome più lungo. Successivamente, vengono inizializzati tutti e 16 i byte di C_i nel seguente modo:

C_i:

Notiamo che all'interno di LOOP, dopo l'invocazione delle due malloc, il registro EDI viene posto uguale all'indirizzo di memoria di C_i e, normalmente, questo valore rimane memorizzato in EDI fino all'iterazione successiva di LOOP. Ciò vuol dire che:
- il secondo campo di C_0 è pari a NULL;
- il secondo campo di C_i (per i>0) è pari all'indirizzo di C_{i-1}.

Perciò, se definiamo (indirizzo di C_{-1}) := NULL, possiamo rappresentare così la struttura di dati C_i:

C_i:



Nel frattempo, viene invocata la funzione RegEnumKeyExA, la quale recupera le informazioni di una specifica subkey, il che lascia capire che il ciclo LOOP itera sul numero delle subkeys della nostra registry key. Il valore di ritorno di RegEnumKeyExA è 0 in caso di successo, un valore differente altrimenti.
I parametri sono riassunti nella seguente tabella:

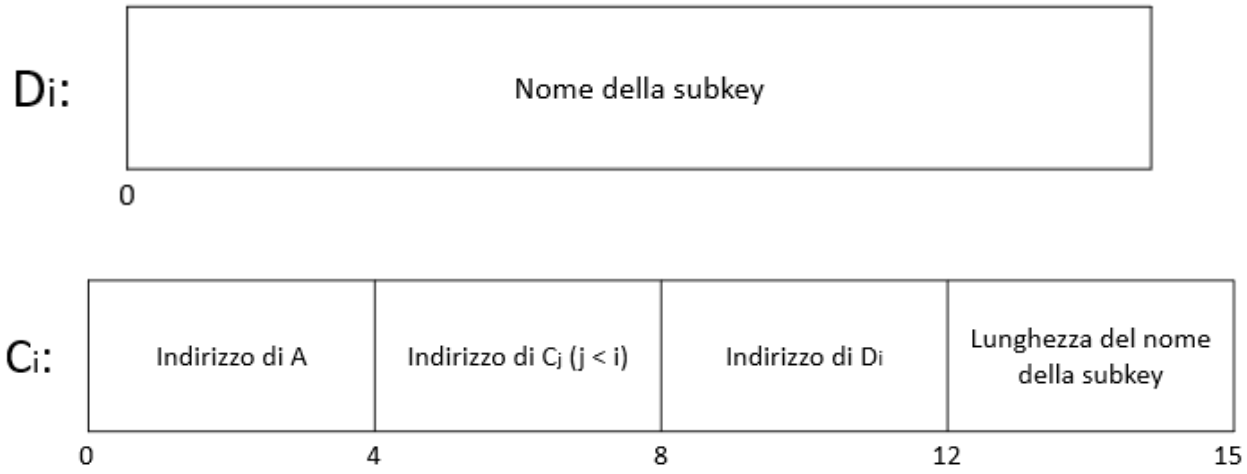
#	TIPO	NOME	DESCRIZIONE	VALORE
1	HKEY	hKey	Handle a una registry key aperta.	Valore assunto dal parametro di f2
2	DWORD	dwIndex	Indice della subkey da estrarre.	Valore assunto dal registro ESI
3	LPSTR	lpName	Puntatore a un buffer che riceverà il nome della subkey.	[out] Indirizzo di D_i
4	LPDWORD	lpcchName	Puntatore a una variabile che indica le dimensioni del buffer puntato da lpName; dopo l'esecuzione della funzione, tale variabile conterrà il numero di caratteri inseriti nel buffer.	[in, out] (Indirizzo di C_i) + 12 byte
5	LPDWORD	lpReserved	Parametro riservato.	NULL
6	LPSTR	lpClass	Puntatore a un buffer che riceverà la classe della subkey.	NULL
7	LPDWORD	lpcchClass	Puntatore a una variabile che indica le dimensioni del buffer puntato da lpClass; dopo l'esecuzione della funzione, tale variabile conterrà la lunghezza della stringa memorizzata nel buffer.	NULL
8	PFILETIME	lpftLastWriteTime	Puntatore a una struttura FILETIME che riceverà l'istante di tempo dell'ultimo aggiornamento della subkey.	[out] Valore assunto dalla variabile locale lpftLastWriteTime

Dopo l'esecuzione di RegEnumKeyExA si effettua un controllo sul valore di ritorno di questa stessa API, per cui si hanno due casi:

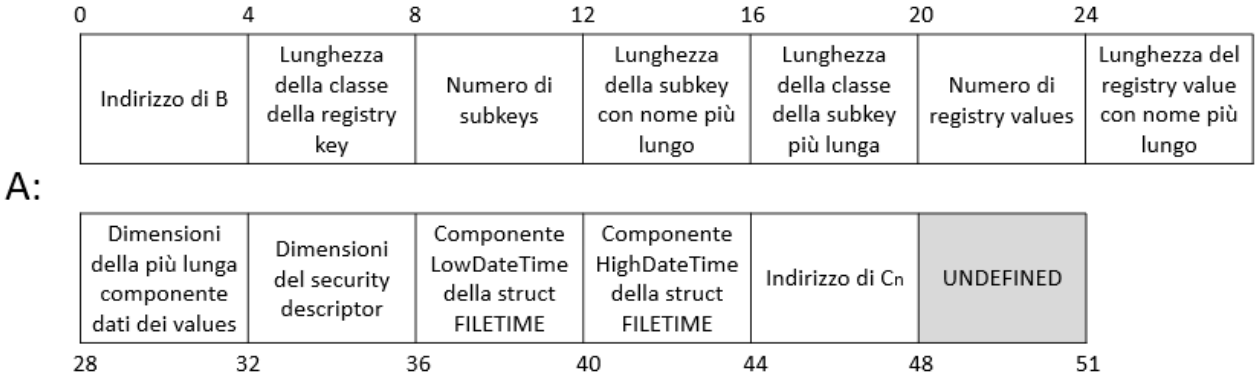
1) Il valore di ritorno è pari a 0 => si effettua un salto verso l'etichetta LAB_004015f8 e si incrementa di un'unità il valore del registro ESI (che nel frattempo ho capito essere l'indice delle iterazioni del ciclo). Dopodiché, se ESI ha raggiunto il numero delle subkeys contenute nella nostra registry key, allora si effettua un altro salto verso l'etichetta LAB_00401698, che si trova all'esterno di LOOP; altrimenti, si torna al comando contrassegnato dall'etichetta LOOP. In altre parole, se abbiamo già iterato su tutte le subkeys, abbandoniamo il ciclo, altrimenti effettuiamo un'altra iterazione.
Ho dunque ridenominato l'etichetta LAB_004015f8 in REPEAT_LOOP e l'etichetta LAB_00401698 in EXIT_LOOP.

2) Il valore di ritorno è diverso da 0 => il valore del registro EDI viene posto pari al secondo campo di C_i, ovvero all'indirizzo di C_{i-1}; il valore del registro ESI viene incrementato di 1 e viene invocata la funzione free con parametro C_i (in altre parole, viene liberata l'area di memoria relativa a C_i, che non esiste più). Dopodiché, se ESI è strettamente minore del numero delle subkeys contenute nella nostra registry key, allora si effettua un salto verso LOOP; altrimenti, si prosegue, arrivando all'istruzione contrassegnata dall'etichetta EXIT_LOOP.

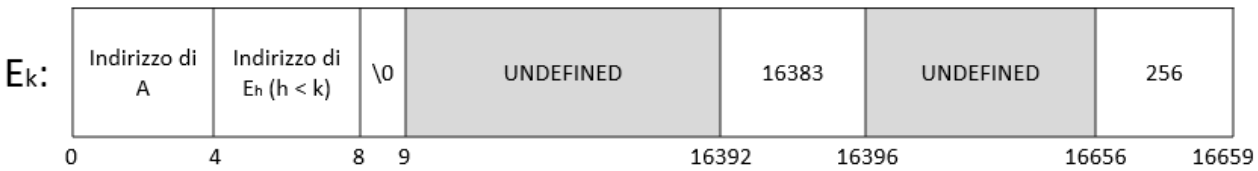
A valle di queste ultime osservazioni, possiamo notare che le varie strutture di dati C_i, insieme, formano una lista collegata, in cui ciascun nodo C_i non punta necessariamente al suo predecessore C_{i-1} (che potrebbe essere stato deallocato con una free) ma, più in generale, punta a un altro nodo C_j con $j < i$. Di conseguenza, la nostra rappresentazione di D_i e di C_i diventa la seguente:



Dopo il ciclo LOOP, i penultimi 4 byte della struttura A vengono inizializzati all'indirizzo di C_n, che corrisponde all'area di memoria relativa all'ultima subkey valida estratta all'interno di LOOP. Abbiamo dunque:



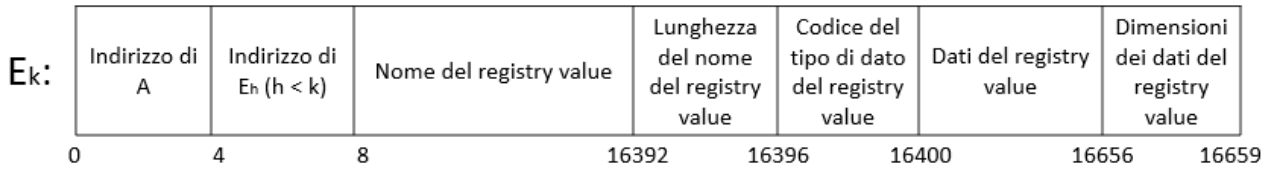
Successivamente viene effettuato un controllo sul campo "numero di registry values" di A: se esso vale 0, vengono inizializzati gli ultimi 4 byte di A a NULL e viene restituito l'indirizzo di A. Altrimenti, si effettua un salto verso l'etichetta LAB_004016d3. Anche in questo caso, dopo tale etichetta, è presente un salto condizionato che riporta indietro esattamente all'istruzione di indirizzo 0x004016d3: effettivamente, si tratta di un altro ciclo, per cui ho ridenominato l'etichetta in LOOP2. Inoltre, anche qui, subito prima del ciclo, i registri EDI e ESI vengono azzerati.
Se diamo un primo sguardo a LOOP2, ci accorgiamo che ha una struttura del tutto analoga a quella di LOOP, con l'unica differenza sostanziale che i ruoli di EDI e ESI sono invertiti (qui EDI funge da contatore, mentre ESI memorizza l'indirizzo dell'ultima area di memoria allocata dinamicamente con malloc). Per questo motivo, i ragionamenti fatti per il ciclo LOOP valgono anche in questa sede.
Entrando più nel dettaglio, all'interno di LOOP2 viene dapprima istanziata una struttura di dati E_k (dove k è l'indice delle iterazioni nel ciclo) tramite l'invocazione di malloc(16660). Dopodiché vengono inizializzati alcuni campi di E_k nel seguente modo:



Poi viene invocata la funzione RegEnumValueA, la quale recupera le informazioni di uno specifico registry value, il che lascia capire che il ciclo LOOP itera sul numero dei registry values della nostra registry key. Il valore di ritorno di RegEnumValueA è 0 in caso di successo, un valore differente altrimenti.
I parametri sono riassunti nella seguente tabella:

#	TIPO	NOME	DESCRIZIONE	VALORE
1	HKEY	hKey	Handle a una registry key aperta.	Valore assunto dal parametro di f2
2	DWORD	dwIndex	Indice del registry value da estrarre.	Valore assunto dal registro EDI
3	LPSTR	lpValueName	Puntatore a un buffer che riceverà il nome del registry value.	[out] (Indirizzo di E_k) + 8 byte
4	LPDWORD	lpcchValueName	Puntatore a una variabile che indica le dimensioni del buffer puntato da lpValueName; dopo l'esecuzione della funzione, tale variabile conterrà il numero di caratteri inseriti nel buffer.	[in, out] (Indirizzo di E_k) + 16392 byte
5	LPDWORD	lpReserved	Parametro riservato.	NULL
6	LPDWORD	lpType	Puntatore a una variabile che riceverà il codice relativo al tipo di dato memorizzato nel registry value.	[out] (Indirizzo di E_k) + 16396 byte
7	LPBYTE	lpData	Puntatore a un buffer che riceverà i dati del registry value.	[out] (Indirizzo di E_k) + 16400 byte
8	LPDWORD	lpcbData	Puntatore a una variabile che indica le dimensioni del buffer puntato da lpData; dopo l'esecuzione della funzione, tale variabile conterrà il numero di byte inseriti nel buffer.	[in, out] (Indirizzo di E_k) + 16656 byte

A seguito dell'invocazione di RegEnumValueA, possiamo dire che la struttura di dati E_k è siffatta:



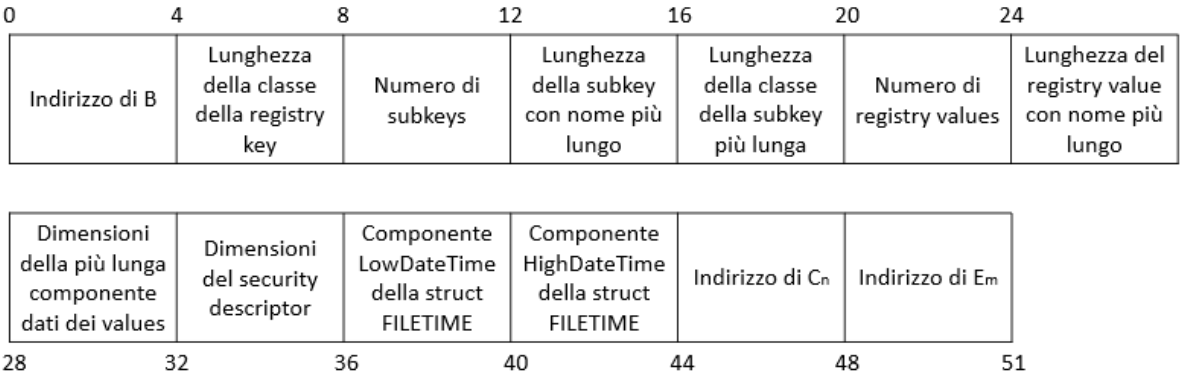
Dopodiché si effettua un controllo sul valore di ritorno di tale API, per cui si hanno i soliti due casi:

1) Il valore di ritorno è pari a 0 => si incrementa di un'unità il valore del registro EDI; se quest'ultimo ha raggiunto il numero dei registry values contenuti nella nostra registry key, allora si esce da LOOP2, altrimenti si effettua un'altra iterazione.

2) Il valore di ritorno è diverso da 0 => il valore del registro ESI viene posto pari al secondo campo di E_k, quello del registro EDI viene incrementato di 1 e viene liberata l'area di memoria relativa a E_k tramite un'invocazione alla funzione free. Se poi EDI è maggiore o uguale al numero dei registry values contenuti nella nostra registry key, allora si esce dal ciclo; altrimenti, si prosegue con l'iterazione successiva, arrivando all'istruzione contrassegnata dall'etichetta LOOP2.

Dopo il ciclo LOOP2, gli ultimi 4 byte della struttura A vengono inizializzati all'indirizzo di E_m, che corrisponde all'area di memoria relativa all'ultimo registry value valido estratto all'interno di LOOP2.
Abbiamo dunque:

A:



Infine, la funzione f2 restituisce l'indirizzo di A.

Ricapitoliamo ora brevemente cosa indicano le strutture di dati istanziate in f2:
- STRUTTURA A: memorizza tutte le informazioni riguardanti la nostra registry key (che ricordiamo essere stata aperta mediante la funzione RegOpenKeyExA).
- STRUTTURA B: è un buffer contenente semplicemente una stringa che indica la classe della nostra registry key.
- STRUTTURE C_i: formano una lista collegata riguardante le subkeys della nostra registry key.
- STRUTTURE D_i: sono dei buffer associati alle relative strutture C_i e contengono semplicemente una stringa che indica il nome di una subkey.
- STRUTTURE E_k: formano una lista collegata riguardante i registry value della nostra registry key.

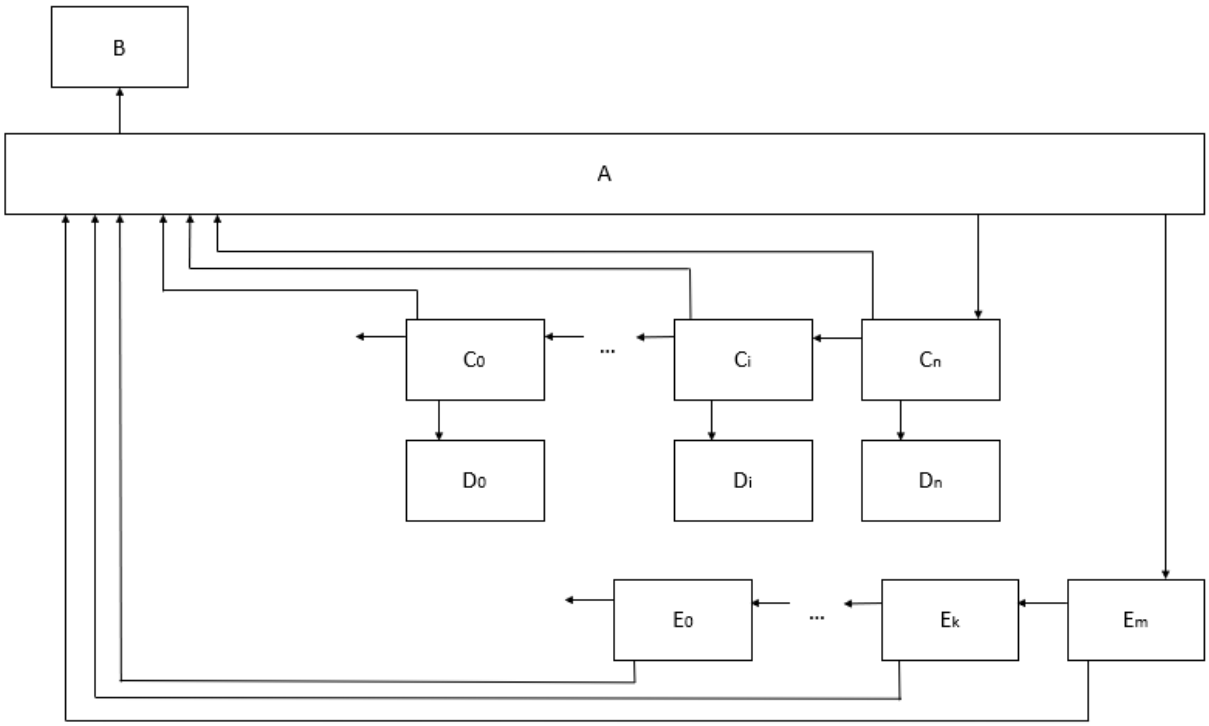
Poiché le strutture A, C_i e E_k contengono molteplici campi di dimensione fissa, e poiché tali campi vengono acceduti tramite uno spiazzamento rispetto all'indirizzo base delle strutture stesse, queste ultime, con ogni probabilità, contengono delle struct del linguaggio C. In particolare:
- All'interno di A possiamo trovare una struct denominata REGISTRY_KEY_INFO.
- All'interno di C_i possiamo trovare una struct denominata SUBKEY_INFO.
- All'interno di E_k possiamo trovare una struct denominata REGISTRY_VALUE_INFO.
- B non è altro che un array di caratteri lungo 260 byte.
- D_i non è altro che un array di caratteri lungo L byte (dove ricordiamo che L è pari alla lunghezza della subkey con nome più lungo).

Possiamo dunque definire delle struct rispettivamente relative a E_k, C_i e A nel seguente modo (in cui ho anche specificato i tipi di dato di tutti i campi di queste tre strutture):

```
struct _registry_value_info;

struct _subkey_info;

struct _registry_key_info;


typedef struct _registry_value_info {

        struct _registry_key_info* regKeyInfo;

        struct _registry_value_info* prevRegValueInfo;

        char regValueName[16384];

        DWORD regValueNameLen;
```

```
	DWORD regValueType;
	BYTE regValueData[256];
	DWORD regValueDataLen;
} REGISTRY_VALUE_INFO;

typedef struct _subkey_info {
	struct _registry_key_info* regKeyInfo;
	struct _subkey_info* prevSubkeyInfo;
	LPSTR subkeyName;
	DWORD subkeyNameLen;
} SUBKEY_INFO;

typedef struct _registry_key_info {
	LPSTR keyClass;
	DWORD keyClassLen;
	DWORD numSubkeys;
	DWORD maxSubkeyNameLen;
	DWORD maxSubkeyClassLen;
	DWORD numRegValues;
	DWORD maxRegValueNameLen;
	DWORD maxRegValueDataLen;
	DWORD secDescriptorLen;
	FILETIME lastWriteTime;
	struct _subkey_info* lastSubkeyInfo;
	struct _registry_value_info* lastRegValueInfo;
} REGISTRY_KEY_INFO;
```

Ora, per avere una visione più d'insieme delle strutture di dati principali utilizzate nella funzione f2, riporterò qui di seguito uno schema che mostri i loro riferimenti:



Tornando alla funzione f1, dopo aver invocato f2, se tutto è andato bene, riceve come valore di ritorno da parte di f2 un puntatore alla struct REGISTRY_KEY_INFO (altrimenti riceve NULL), invoca la funzione RegCloseKey per chiudere la registry key precedentemente aperta con RegOpenKeyExA, e dopo restituisce al main il medesimo puntatore alla struct REGISTRY_KEY_INFO ricevuta da f2.

D'altra parte, il main, dopo aver ricevuto questo puntatore, effettua un controllo su di esso: se è NULL, allora restituisce immediatamente 1; altrimenti, invoca un'altra funzione (FUN_00401790) che ho ridenominato f3, passandole come parametro proprio il puntatore alla struct REGISTRY_KEY_INFO.

La funzione f3, nell'ordine, esegue le seguenti operazioni:

1) Invocazione di printf con la stringa "Class: %s\n" e il primo campo (keyClass) di REGISTRY_KEY_INFO come parametri: viene stampata a schermo la classe della nostra registry key.

2) Invocazione di printf con la stringa "Security descriptor: 0x%1x\n" e il nono campo (secDescriptorLen) di REGISTRY_KEY_INFO come parametri: viene stampata a schermo la lunghezza del security descriptor della nostra registry key.

3) Invocazione di printf con la stringa "Time: %081x%081x\n" e il decimo (lastWriteTime.dwLowDateTime) e l'undicesimo (lastWriteTime.dwHighDateTime) campo di REGISTRY_KEY_INFO come parametri: viene stampato a schermo il timestamp dell'istante dell'ultima modifica della nostra registry key.

4) Viene effettuato un controllo sul penultimo campo (lastSubkeyInfo) di REGISTRY_KEY_INFO: se è NULL, si effettua un salto in avanti verso l'etichetta LAB_0040180a e si passa direttamente allo step 5. Altrimenti, viene invocato puts("Sub-keys:") e si entra all'interno di un breve ciclo che ho ridenominato SUBKEY_INFO_LOOP. In tale ciclo viene semplicemente chiamata una printf con la stringa "\t%s\n" e il terzo campo (subkeyName, ovvero il puntatore al nome della subkey che si trova all'interno del buffer che avevo chiamato D_i) della struct SUBKEY_INFO come parametri. Dopodiché, viene effettuato un controllo sul secondo campo (prevSubkeyInfo) di SUBKEY_INFO: se questo è diverso da NULL, vuol dire che ci sono altre informazioni da stampare e si incomincia una nuova iterazione all'interno del ciclo; altrimenti, si esce da

SUBKEY_INFO_LOOP.
In poche parole, questo ciclo itera sul numero delle subkeys della nostra registry key e, per ciascuna subkey, ne stampa il nome.

5) Viene effettuato un controllo sull'ultimo campo (lastRegValueInfo) di REGISTRY_KEY_INFO: se è NULL, si effettua un salto in avanti verso l'etichetta LAB_00401890 da cui parte l'epilogo con la terminazione della funzione f3.
Altrimenti, viene invocato puts("Values:") e si entra all'interno di un altro ciclo che ho ridenominato REGISTRY_VALUE_INFO_LOOP. In questo ciclo, viene chiamata una printf con la stringa "\t%s: [%lu] " e il terzo campo (regValueName) e il quinto campo (regValueType) della struct REGISTRY_VALUE_INFO come parametri.
Dopodiché, si effettua un controllo sull'ultimo campo (regValueDataLen) della struct REGISTRY_VALUE_INFO: se questo è diverso da 0, vuol dire che nella struttura sono riportati dei dati relativi a un registry value (in particolare al k-esimo registry value su cui stiamo iterando) e si prosegue con le istruzioni assembly immediatamente successive (di cui possiamo già immaginare che hanno lo scopo di stampare questi dati relativi al registry value), che sono contrassegnate dall'etichetta LAB_00401850; altrimenti, si effettua un salto in avanti verso l'etichetta LAB_00401873, che si trova comunque all'interno di REGISTRY_VALUE_INFO_LOOP.
In corrispondenza dell'etichetta LAB_00401850, possiamo notare che si ha un ciclo annidato all'interno di REGISTRY_VALUE_INFO_LOOP: possiamo ridenominare tale ciclo (e l'etichetta) in DATI_REG_VAL_LOOP.
Per quanto riguarda DATI_REG_VAL_LOOP, possiamo osservare che il registro EBX viene inizialmente impostato a zero e poi viene incrementato di 1 a ogni iterazione (per cui EBX è l'indice delle iterazioni del ciclo), mentre il registro EAX viene ogni volta impostato all'EBX-esimo byte del campo regValueData di REGISTRY_VALUE_INFO. Ciò che si fa all'interno di questo ciclo annidato è chiamare printf con la stringa " %02x" e l'EBX-esimo byte del campo regValueData di REGISTRY_VALUE_INFO come parametri. Poi viene effettuato un controllo su EBX: se ha raggiunto il valore del campo regValueDataLen della struct REGISTRY_VALUE_INFO, vuol dire che sono stati stampati tutti i byte del campo regValueData e si esce da DATI_REG_VAL_LOOP; altrimenti si procede col byte successivo.
Una volta che siamo usciti da DATI_REG_VAL_LOOP, arriviamo proprio all'etichetta LAB_00401873 (la stessa che avremmo raggiunto direttamente se il campo regValueDataLen di REGISTRY_VALUE_INFO fosse stato nullo). Ciò che si fa qui è invocare printf con la stringa " (%s)\n" e il sesto campo (regValueData) della struct REGISTRY_VALUE_INFO come parametri. Dopodiché, viene effettuato un controllo sul secondo campo (prevRegValueInfo) di REGISTRY_VALUE_INFO: se questo è diverso da NULL, vuol dire che ci sono altre informazioni da stampare e si incomincia una nuova iterazione all'interno del ciclo; altrimenti, si esce da REGISTRY_VALUE_INFO_LOOP.
In pratica, questo ciclo itera sul numero dei registry values della nostra registry key e, per ciascuno di essi, ne stampa il nome, il tipo, i dati byte per byte e, infine, gli stessi dati in formato stringa con un'unica invocazione a printf.

Alla fine dello step 5 qui sopra descritto, la funzione f3 termina e restituisce il controllo al chiamante (che è il main). A questo punto, il main a sua volta termina chiamando una return 0.

CODICE C DEL PROGRAMMA

Una volta giunto alla fine dell'analisi, ho ridenominato le funzioni f1, f2, f3 con dei nomi che descrivono brevemente la loro funzionalità principale:
f1 -> openRegistryKey
f2 -> retrieveRegistryKeyInfo
f3 -> printRegistryKeyInfo

Di seguito ho riportato il codice C che ho ricavato dall'analisi del programma con Ghidra alla luce delle considerazioni fatte fino a questo momento.

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <winreg.h>

struct _registry_value_info;
struct _subkey_info;
struct _registry_key_info;

typedef struct _registry_value_info {
	struct _registry_key_info* regKeyInfo;
	struct _registry_value_info* prevRegValueInfo;
	char regValueName[16384];
	DWORD regValueNameLen;
	DWORD regValueType;
	BYTE regValueData[256];
	DWORD regValueDataLen;
} REGISTRY_VALUE_INFO;

typedef struct _subkey_info {
	struct _registry_key_info* regKeyInfo;
	struct _subkey_info* prevSubkeyInfo;
	LPSTR subkeyName;
	DWORD subkeyNameLen;
} SUBKEY_INFO;

typedef struct _registry_key_info {
	LPSTR keyClass;
	DWORD keyClassLen;
```

```
	DWORD numSubkeys;
	DWORD maxSubkeyNameLen;
	DWORD maxSubkeyClassLen;
	DWORD numRegValues;
	DWORD maxRegValueNameLen;
	DWORD maxRegValueDataLen;
	DWORD secDescriptorLen;
	FILETIME lastWriteTime;
	struct _subkey_info* lastSubkeyInfo;
	struct _registry_value_info* lastRegValueInfo;
} REGISTRY_KEY_INFO;



REGISTRY_KEY_INFO* retrieveRegistryKeyInfo(HKEY regKey) {

	REGISTRY_KEY_INFO* regKeyInfo = malloc(52);
	if (regKeyInfo == NULL) {
		puts("Memory allocation error");
		exit(1);
	}

	LPSTR kClass = malloc(260);
	if (kClass == NULL) {
		puts("Memory allocation error");
		exit(1);
	}
	kClass[0] = '\0';

	regKeyInfo->keyClass = kClass;
	regKeyInfo->keyClassLen = 260;
	regKeyInfo->numSubkeys = 0;
```

```
	LSTATUS fun_ret = RegQueryInfoKeyA(regKey, regKeyInfo->keyClass, &(regKeyInfo->keyClassLen),
NULL, &(regKeyInfo->numSubkeys), &(regKeyInfo->maxSubkeyNameLen), &(regKeyInfo->
maxSubkeyClassLen), &(regKeyInfo->numRegValues), &(regKeyInfo->maxRegValueNameLen),
&(regKeyInfo->maxRegValueDataLen), &(regKeyInfo->secDescriptorLen), &(regKeyInfo->lastWriteTime));
	if (fun_ret != 0) {
		puts("RegQueryInfoKey failed: key not found");
		return NULL;
	}
	PFILETIME lpftLastWriteTime = &(regKeyInfo->lastWriteTime);

	if (regKeyInfo->numSubkeys != 0) {
		SUBKEY_INFO* previous = NULL;

		for (DWORD i = 0; i < regKeyInfo->numSubkeys; i++) {

			SUBKEY_INFO* subkeyInfo = malloc(16);
			if (subkeyInfo == NULL) {
				puts("Memory allocation error");
				exit(1);
			}

			LPSTR name = malloc(regKeyInfo->maxSubkeyNameLen);
			if (name == NULL) {
				puts("Memory allocation error");
				exit(1);
			}

			subkeyInfo->regKeyInfo = regKeyInfo;
			subkeyInfo->prevSubkeyInfo = previous;
			subkeyInfo->subkeyName = name;
			subkeyInfo->subkeyNameLen = regKeyInfo->maxSubkeyNameLen;
```

```
                if (RegEnumKeyExA(regKey, i, subkeyInfo->subkeyName, &(subkeyInfo->
subkeyNameLen), NULL, NULL, NULL, lpftLastWriteTime) != 0) free(subkeyInfo);
                else previous = subkeyInfo;
            }
            regKeyInfo->lastSubkeyInfo = previous;
        }
        else regKeyInfo->lastSubkeyInfo = NULL;

        if (regKeyInfo->numRegValues != 0) {
            REGISTRY_VALUE_INFO* previous2 = NULL;

            for (DWORD i = 0; i < regKeyInfo->numRegValues; i++) {

                REGISTRY_VALUE_INFO* regValInfo = malloc(16660);
                if (regValInfo == NULL) {
                    puts("Memory allocation error");
                    exit(1);
                }

                regValInfo->regKeyInfo = regKeyInfo;
                regValInfo->prevRegValueInfo = previous2;
                regValInfo->regValueName[0] = '\0';
                regValInfo->regValueNameLen = 16383;
                regValInfo->regValueDataLen = 256;

                if (RegEnumValueA(regKey, i, regValInfo->regValueName, &(regValInfo->
regValueNameLen), NULL, &(regValInfo->regValueType), regValInfo->regValueData, &(regValInfo->
regValueDataLen)) != 0) free(regValInfo);
                else previous2 = regValInfo;
            }
            regKeyInfo->lastRegValueInfo = previous2;
        }
        else regKeyInfo->lastRegValueInfo = NULL;
```

```
	return regKeyInfo;
}



REGISTRY_KEY_INFO* openRegistryKey(HKEY param_1, LPCSTR param_2) {

	HKEY hkResult;
	if (RegOpenKeyExA(param_1, param_2, 0, KEY_ALL_ACCESS, &hkResult) != 0) return NULL;

	REGISTRY_KEY_INFO* regKeyInfo = retrieveRegistryKeyInfo(hkResult);
	RegCloseKey(hkResult);

	return regKeyInfo;
}



void printRegistryKeyInfo(REGISTRY_KEY_INFO* info) {

	printf("Class: %s\n", info->keyClass);
	printf("Security descriptor: 0x%1x\n", info->secDescriptorLen);
	printf("Time: %081x%081x\n", info->lastWriteTime.dwLowDateTime, info->
lastWriteTime.dwHighDateTime);

	if (info->lastSubkeyInfo != NULL) {
		SUBKEY_INFO* current = info->lastSubkeyInfo;
		SUBKEY_INFO* previous = info->lastSubkeyInfo->prevSubkeyInfo;
		puts("Sub-keys:");

		while (1) {
```

```
                printf("\t%s\n", current->subkeyName);
                current = previous;
                if (current == NULL) break;
                previous = previous->prevSubkeyInfo;
            }
        }

        if (info->lastRegValueInfo != NULL) {
            REGISTRY_VALUE_INFO* current2 = info->lastRegValueInfo;
            REGISTRY_VALUE_INFO* previous2 = info->lastRegValueInfo->prevRegValueInfo;
            puts("Values:");

            while (1) {
                printf("\t%s: [%lu] ", current2->regValueName, current2->regValueType);

                for (unsigned int i = 0; i < current2->regValueDataLen; i++) {
                    printf(" %02x", current2->regValueData[i]);
                }
                printf(" (%s)\n", current2->regValueData);

                current2 = previous2;
                if (current2 == NULL) break;
                previous2 = previous2->prevRegValueInfo;
            }
        }

        return;
}
```

```
int main(int argc, char** argv) {

	if (argv[argc] == NULL) return 0;

	if (argc <= 2) {
		argv[1] = NULL;
		argv[2] = NULL;
	}

	HKEY openRegKey;
	LPCSTR regKeyToOpen;
	if (atoi(argv[1]) == 0) openRegKey = HKEY_LOCAL_MACHINE;
	else openRegKey = (HKEY)atoi(argv[1]);
	if (argv[2] == NULL) regKeyToOpen = "SYSTEM\\ControlSet001\\Control";
	else regKeyToOpen = argv[2];

	REGISTRY_KEY_INFO* regKeyInfo = openRegistryKey(openRegKey, regKeyToOpen);
	if (regKeyInfo == NULL) return 1;

	printRegistryKeyInfo(regKeyInfo);

	return 0;
}
```