

Relazione homework 1

Luca Mastrobattista

Indice

1	Traccia dell'homework	2
1.1	Testo	2
1.2	Scadenza	2
1.3	Consegna	2
2	Ambiente di lavoro	3
3	Metodologia	4
3.1	Informazioni note a priori	4
3.2	Finalizzazione dell'obiettivo	4
3.3	Ottenimento del codice macchina	4
3.4	Osservazione del funzionamento	4
3.5	Disassemblaggio del codice macchina	5
3.5.1	Riepilogo risultati dell'import	6
3.5.2	Informazioni aggiuntive	6
3.6	Ricerca del main	8
3.6.1	Ricerca tramite invocazioni di funzioni user	8
4	Analisi con Ghidra	11
4.1	FUN_00402a30_main	11
4.2	FUN_004019a0_set_on_exit_actions	11
4.3	FUN_00401500_in_my_struct	13
4.4	FUN_00401900_given_to_set_before_exit	13
4.5	FUN_004014e0_set_before_exit	13
4.6	FUN_004018a0_open_handle_to_reg_key	15
4.7	FUN_00401530_retrieve_subkeys_and_values	16
4.8	FUN_00401790_print_catched_info	24

1 Traccia dell'homework

1.1 Testo

Analizzare con Ghidra, utilizzando lo strumento disassemblatore / decompilatore, il programma eseguibile hw1.exe contenuto nell'archivio hw1.zip (password: "AMW21"). Riassumere in un documento tutte le informazioni acquisite sul programma, con particolare riguardo alle strutture di dati fondamentali utilizzate. Descrivere anche la metodologia ed i passi logici deduttivi utilizzati nel lavoro di analisi.

1.2 Scadenza

Due settimane dalla data di assegnazione del lavoro: 28/10/2021

1.3 Consegna

Documento in formato PDF inviato come allegato ad un messaggio di posta elettronica all'indirizzo del docente ("`<cognome>@uniroma2.it`"), con subject: "[AMW21] HW1: `<matricola studente>`"

2 Ambiente di lavoro

Il file eseguibile è stato caricato su Ghidra istallato su un sistema operativo Linux.

3 Metodologia

3.1 Informazioni note a priori

Eseguibile ottenuto per l'esecuzione dell'homework e, quindi, non sono note informazioni preliminari.

3.2 Finalizzazione dell'obiettivo

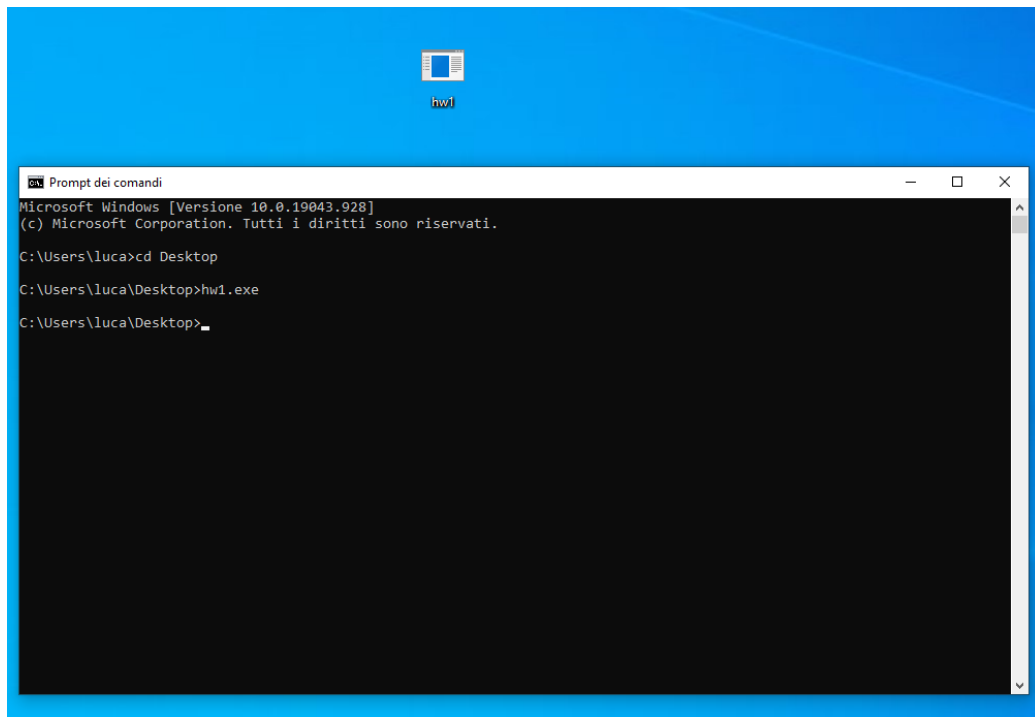
Il reversing dell'applicazione ha come obiettivo quello di comprendere il funzionamento dell'eseguibile e di completare l'attività di erverse code engineering, quindi l'obiettivo dello studio è la ricostruzione dell'intero programma.

3.3 Ottenimento del codice macchina

Codice macchina fornito dal professore.

3.4 Osservazione del funzionamento

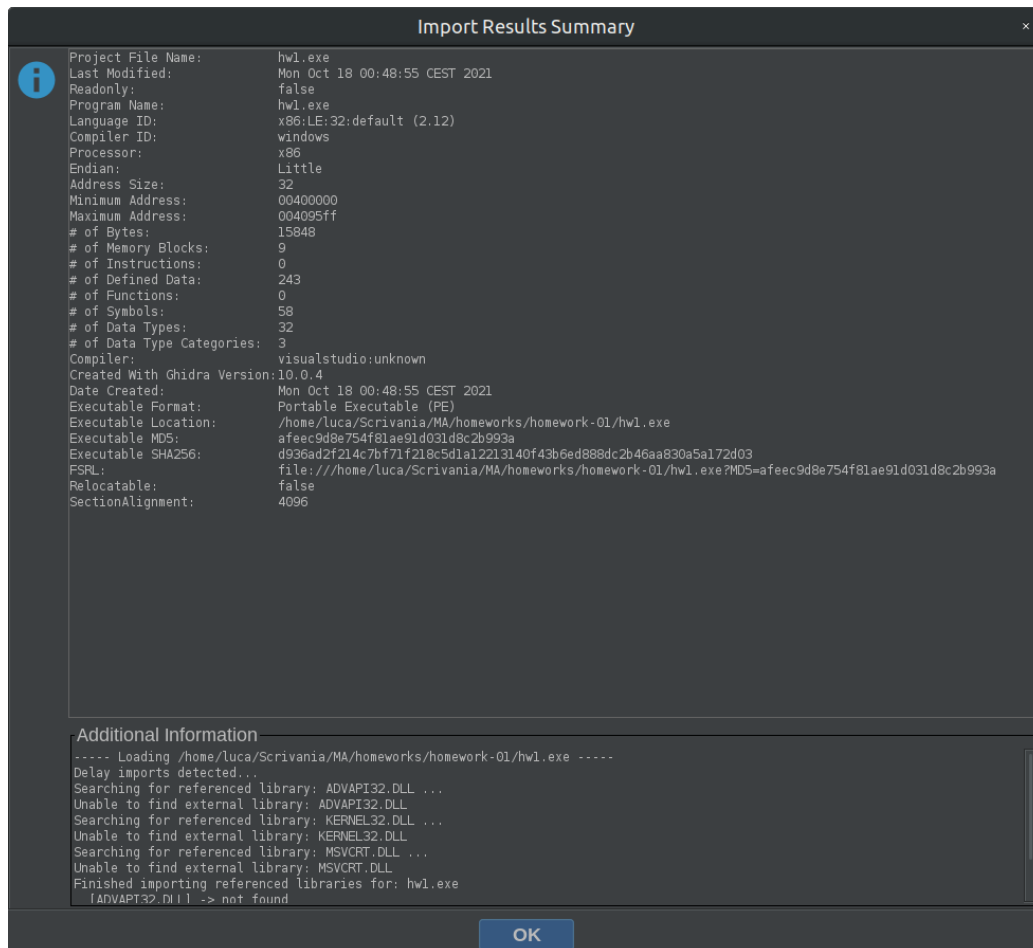
Provando ad eseguire l'eseguibile su una macchina virtuale del sistema operativo Windows 10 non è stato ottenuto alcun output. Neanche nel gestore attività viene riportata alcuna informazione. L'applicazione sembra avviarsi e terminare in poco tempo.



3.5 Disassemblaggio del codice macchina

Lo strumento che si è utilizzato è il software *Ghidra*.

3.5.1 Riepilogo risultati dell'import



3.5.2 Informazioni aggiuntive

```
----- Loading /home/luca/Scrivania/MA/homeworks/homework-01/hw1.exe
-----
Delay imports detected...
Searching for referenced library: ADVAPI32.DLL ...
Unable to find external library: ADVAPI32.DLL
Searching for referenced library: KERNEL32.DLL ...
Unable to find external library: KERNEL32.DLL
Searching for referenced library: MSVCRT.DLL ...
Unable to find external library: MSVCRT.DLL
Finished importing referenced libraries for: hw1.exe
```

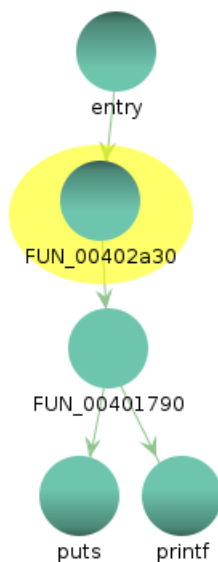
[ADVAPI32.DLL] -> not found
[KERNEL32.DLL] -> not found
[MSVCRT.DLL] -> not found

3.6 Ricerca del main

3.6.1 Ricerca tramite invocazioni di funzioni user

Tentativo di ricerca del `main` sfruttando le invocazioni a funzioni di libreria. Nella libreria `MSVCRT.DLL` viene invocata la funzione `printf`. Cercando le referenze a quest'ultima, ne sono state trovate 7. Queste invocazioni sono tutte all'interno della stessa funzione: `FUN_00401790`. Questa funzione, però, non può essere il nostro `main`, perché prende in input un parametro che viene usato per indicizzare elementi da passare alle varie `printf`. Tra i vari elementi indicizzati, cerca di prendere anche `param_1[0xc]`; questa istruzione viene eseguita sempre, e perciò avrebbe dovuto sollevare un errore in caso di avvio dell'applicazione senza parametri. Inoltre, proprio questo valore `param_1[0xc]` è usato per indicizzare a sua volta un indirizzo con un offset di `0x400c`, corrispondente a 16396. Infine, questa funzione non viene invocata da `entry`, ma da un'altra funzione definita dal programmatore. È quindi difficile credere che `param_1` sia un vettore di stringhe, ma è più probabile che sia una struttura dati più complessa.

In ogni caso, si può essere sicuri che questa funzione sia stata definita dal programmatore, e in questo modo è possibile sfruttare il grafo delle invocazioni delle funzioni per risalire al `main`.



La funzione `FUN_00402a30` è un buon candidato ad essere il `main`.

Una conferma che quella funzione sia effettivamente il `main` arriva dalla seguente osservazione. Nella funzione `entry`, viene invocata due volte

la funzione `MSVCRT.DLL::_initterm`, che prende in input due puntatori. Questi puntatori identificano l'inizio e la fine di una tabella di puntatori a funzioni e vengono inizializzati. Delle due invocazioni, è interessante la seconda: gli indirizzi dati in input identificano una tabella di un solo puntatore a funzione che, una volta definito tale, punta alla `LAB_00401120` che va quindi definita come funzione. Al suo interno viene invocata la funzione `MSVCRT.DLL: __getmainargs`. Questa funzione richiama l'analisi della riga di comando e copia gli argomenti del `main`. Riceve in input 5 parametri che sono tutti variabili globali e rappresentano, in ordine:

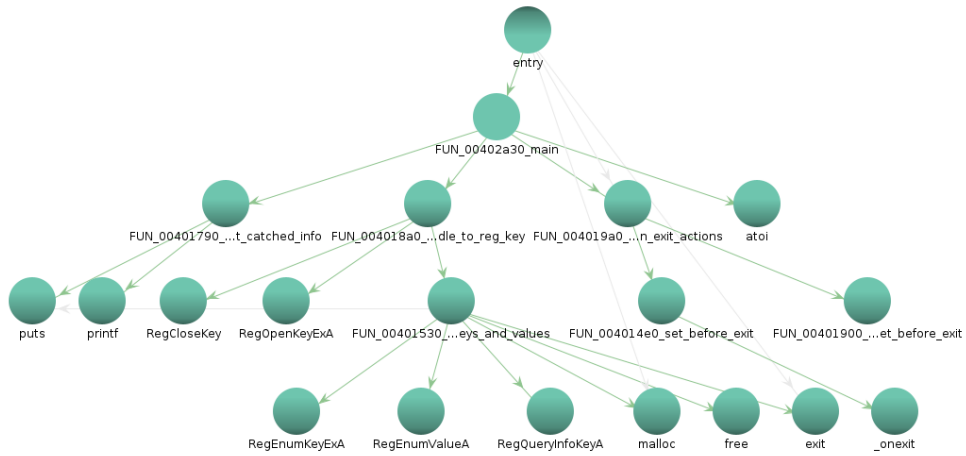
- L'intero `argc`
- La matrice di stringhe `argv`
- Una matrice di stringhe `_env` che rappresenta le variabili impostate nell'ambiente dell'utente.
- Un intero che, se impostato su 1, espande i caratteri jolly negli argomenti della riga di comando o, se impostato su 0, non esegue alcuna operazione.
- Un puntatore a `_startupinfo`, che contiene altre informazioni da passare alla DLL CRT.

La cosa interessante è che i primi due parametri sono passati in input anche alla funzione `FUN_00402a30`, proprio in quell'ordine: quella funzione prende in input `argc` e `argv`, e non può che essere il `main`.

Function call graph del main prima dell'analisi



Function call graph del main dopo l'analisi



4 Analisi con Ghidra

4.1 FUN_00402a30__main

Questa funzione, dopo aver fatto allineamento a 16 bytes, ne invoca subito un'altra: `FUN_004019a0_set_on_exit_actions`, analizzata in seguito.

Poi continua con dei controlli sulla riga comando: effettua `argv[argc]` e il risultato viene utilizzato per fare dei controlli. In particolare, se il valore così indicizzato è `NULL`, la funzione ritorna 0, altrimenti verifica il numero dei parametri passati: se è minore di 3, incluso il nome del programma, si imposta a `NULL` il valore di `argv[1]` e di `argv[2]`. Poi prepara l'invocazione di `MSVCRT.DLL::atoi`, che prende in input una stringa e ne restituisce il valore decimale rappresentato. La stringa data in input è il primo parametro a riga comando. Se il valore di ritorno è 0, si memorizza il valore della macro `HKEY_LOCAL_MACHINE` in `EAX`. Si continua controllando il secondo parametro a riga comando: se è `NULL` si salva in `EDX` il valore della stringa `"SYSTEM\\ControlSet001\\Control"`. Questi due registri conterranno i parametri da passare all'invocazione di `FUN_004018a0_open_handle_to_reg_key`. Dopo l'invocazione, si controlla il valore ritornato dalla funzione: se è 0, termina, altrimenti salva questo valore nello stack e invoca la funzione `FUN_004018a0_open_handle_to_reg_key`. Quando quest'ultima termina, viene ritornato il valore 0.

Nota: il primo controllo che viene fatto è:

```
if (argv[argc] == (char *) 0)
    uvar4 = 0;
else{
    ...
}
return uvar4
```

Ma la condizione dell'`if` è sempre vera: infatti, se al programma sono dati `argc` parametri, l'`argc -esimo` sarà sempre `NULL`.

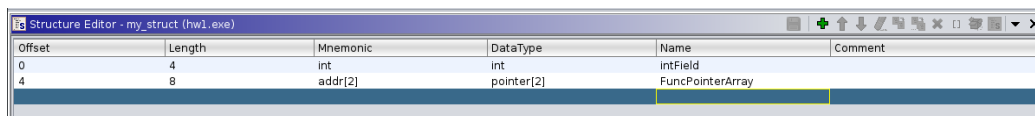
4.2 FUN_004019a0_set_on_exit_actions

La funzione prende il valore della variabile globale `DWORD_00405020_one_time_control_label` e controlla se è 0: se non lo è, termina, altrimenti continua impostandolo a 1. Sembra quindi essere una variabile di controllo per evitare che questo codice venga eseguito più di una volta: infatti gli unici riferimenti a questa variabile sono

all'interno di questa funzione. Ha senso perché questa funzione è invocata anche dentro `entry`. Dopodiché, controlla il valore di un'altra variabile, `DAT_00402ac0_my_struct`. In particolare, viene controllato se il suo valore è uguale a -1. Se non lo è, azzerà il registro `EAX` con un'istruzione di `XOR`, e inizia un ciclo. Ad ogni iterazione, il valore di `EAX` viene prima salvato in `EBX`, poi incrementato di 1, poi moltiplicato per 4 e infine viene usato come offset dal punto base fissato all'indirizzo di `DAT_00402ac0_my_struct`. Questo fatto lascia pensare che, in quell'indirizzo, sia memorizzata una struttura dati con componenti tutti della stessa taglia; potrebbe, a questo punto, essere anche un array di interi: sappiamo infatti che il primo elemento è stato confrontato col valore -1.

Il ciclo termina quando viene trovato un offset, multiplo di 4 byte, che indicizza `NULL` a partire da `DAT_00402ac0_my_struct`. Alla fine di questo ciclo, in `EBX` è memorizzato il numero di indirizzi diversi da `NULL` successivi a `DAT_00402ac0_my_struct`. Se questo valore è diverso da 0, cioè si è trovato almeno un indirizzo valido, parte un altro ciclo in cui vengono effettuate delle `CALL`. Le funzioni invocate sono tutte quelle memorizzate negli indirizzi che sono diversi da `NULL` trovati precedentemente.

A questo punto, la nostra `my_struct` è definita così:



Offset	Length	Mnemonic	DataType	Name	Comment
0	4	int	int	intField	
4	8	addr[2]	pointer[2]	FuncPointerArray	

Il vettore di puntatori a funzioni è definito di dimensione 2 perché è il minimo valore che permette di avere una funzione da invocare e il `NULL` per indicarne la fine. A questo punto dell'analisi non è nota la sua dimensione reale. Inoltre, la struttura dati all'indirizzo `DAT_00402ac0_my_struct`, contiene un intero al primo campo, l'indirizzo della funzione `FUN_00401500_in_my_struct`, e poi un `NULL`: effettivamente l'array ha solo 2 elementi.

Quando queste invocazioni terminano e il ciclo finisce, la funzione inserisce l'indirizzo di una funzione nello stack (`FUN_00401900_given_to_set_before_exit`), alla posizione `-0x1c` e invocherà poi `FUN_004014e0_set_before_exit` prima di terminare.

Nota: lo stack pointer viene cambiato come prima istruzione di `FUN_004014e0_set_before_exit`, che alloca proprio `0x1c` bytes; quindi il primo parametro di quella funzione corrisponderà al valore inserito in questo offset.

4.3 FUN_00401500_in_my_struct

Questa funzione era marcata come `UndefinedFunction` dal decompilatore perché non viene trovata una `CALL` che invoca direttamente questa funzione ed è l'unica presente nell'array di vettori nell'istanza di `my_struct` all'indirizzo `DAT_00402ac0_my_struct`. L'unica cosa che fa è invocare `FUN_004014e0_set_before_exit`, a cui viene passato in input il valore di `DAT_00401520`. Poiché, come vedremo, `FUN_004014e0_set_before_exit` prende in input un puntatore a funzione, possiamo dedurre che il valore della variabile `DAT_00401520` è un puntatore a funzione.

Problema: convertendo la variabile `DAT_00401520` in tipo di dato *pointer*, viene indicizzato un indirizzo che non è presente nell'address space. Tuttavia questo valore è passato a `_onexit`, che prende in input un puntatore a funzione e non può quindi essere di un altro tipo.

4.4 FUN_00401900_given_to_set_before_exit

Questa funzione, per prima cosa, carica un indirizzo dentro il registro `EAX`. Questo indirizzo, chiamato `PTR_PTR_00403004`, è un puntatore doppio. Infatti, subito dopo, viene preso l'indirizzo puntato e si controlla se questo sia `NULL`. Se lo è, termina, altrimenti inizia un ciclo che, come prima istruzione, prevede una `CALL` proprio a quell'indirizzo. Di conseguenza, il valore puntato da `PTR_DAT_00403004` è un puntatore a funzione. Nelle iterazioni si sposta di 4 bytes in avanti rispetto all'ultimo indirizzo usato nella `CALL` e, se non è `NULL`, procede con la sua invocazione. Si sta praticamente scorrendo un ipotetico array di puntatori a funzioni. Il ciclo termina quando viene trovato un puntatore a `NULL`; quando ciò accade la funzione ritorna.

Nota: il primo controllo che viene fatto è che l'indirizzo base puntato da `PTR_PTR_00403004` non sia `NULL`, ma nel codice questo puntatore punta proprio a `NULL`. La funzione dovrebbe quindi terminare semplicemente, senza fare altro.

4.5 FUN_004014e0_set_before_exit

Questa funzione invoca al suo interno la funzione di libreria `MSVCRT.DLL::_onexit`. Questa funzione prende in input un puntatore alla funzione che deve essere invocata prima che il programma termini, una sorta di *wrapper* per l'evento di uscita. In caso di più invocazioni di questa

funzione, l'ordine di esecuzione delle varie funzioni è quello *LIFO*: infatti la documentazione riporta il seguente esempio:

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

/* Prototypes */
int fn1(void), fn2(void), fn3(void), fn4 (void);

int main( void )
{
    _onexit( fn1 );
    _onexit( fn2 );
    printf( "This is executed first.\n" );
}

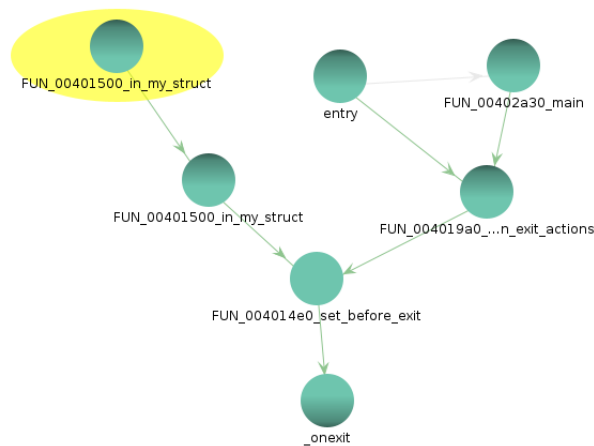
int fn1()
{
    printf( "executed next.\n" );
    return 0;
}

int fn2()
{
    printf( "This is " );
    return 0;
}
```

L'output è:

```
This is executed first.
This is executed next.
```

È utile quindi ricercare tutti i punti in cui questa funzione viene invocata. Ci sono solo 2 riferimenti, entrambi nella funzione `FUN_004019a0_set_on_exit_actions`: uno è diretto, l'altro invece è nascosto: si invoca tramite la struttura dati `my_struct`



Nota: la funzione `FUN_00401500_in_my_struct` compare 2 volte perchè la prima, quella evidenziata nell'immagine, è una *Thunk*.

Prima di terminare l'applicazione, quindi, le funzioni verranno eseguite nel seguente ordine:

1. `FUN_00401900_given_to_set_before_exit`, che però, come detto, nella **Nota**, dovrebbe terminare subito.
2. `FUN_00401500_in_my_struct`, che però utilizza un indirizzo non presente nell'address space: `PTR_DAT_00401520` ha come valore `DAT_909090c3`, segnato in rosso da Ghidra.

Nessuna funzione viene impostata come *wrapper* per l'evento di *exit*.

4.6 FUN_004018a0_open_handle_to_reg_key

Questa è un'altra funzione invocata da `main` che, per prima cosa, azzerà il registro `EBX`, poi prepara l'invocazione di `ADVAPI32.DLL::RegOpenKeyExA`. Questa funzione prende in input i seguenti parametri:

1. un `HKEY`, passato in input alla funzione;
2. un `LPCSTR`, anche questo dato in input alla funzione
3. un `DWORD`, che in questo caso è 0;
4. un `REGSAM`, una maschera di bit che rappresenta i diritti di accesso, e in questo caso è impostato a `0xf003f` che corrisponde a `KEY_ALL_ACCESS`;

5. un PHKEY, un puntatore che riceve l'handle verso la key aperta.

Quando la funzione ritorna, si controlla il suo valore di ritorno: infatti, se tutto è andato bene, ritorna il valore 0, corrispondente a `ERROR_SUCCESS`. Se in `EAX` c'è effettivamente il valore 0, si mette sullo stack l'handle verso la key aperta e si invoca `FUN_00401530_retrieve_subkeys_and_values`. Il valore di ritorno di questa funzione si memorizza in `EBX`. Poi si invoca `ADVAPI32.DLL::RegCloseKey`, passandogli come parametro l'handle verso la key ottenuto precedentemente, in modo da chiuderlo. Infine, la funzione termina restituendo il valore contenuto in `EBX`, che sarà `NULL` se c'è stato un errore oppure il risultato di `FUN_00401530_retrieve_subkeys_and_values`.

Riassumendo: si apre il *registry key path* dato da riga comando ottenendo un *handle* verso il registro tramite l'invocazione di `ADVAPI32.DLL::RegOpenKeyExA`; se non viene passato nulla, il default è impostato a: `HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control`. Si invoca poi `FUN_00401530_retrieve_subkeys_and_values` passandogli l'handle ottenuto, il cui valore di ritorno viene passato al `main` dopo aver chiuso l'handle con `ADVAPI32.DLL::RegCloseKey`.

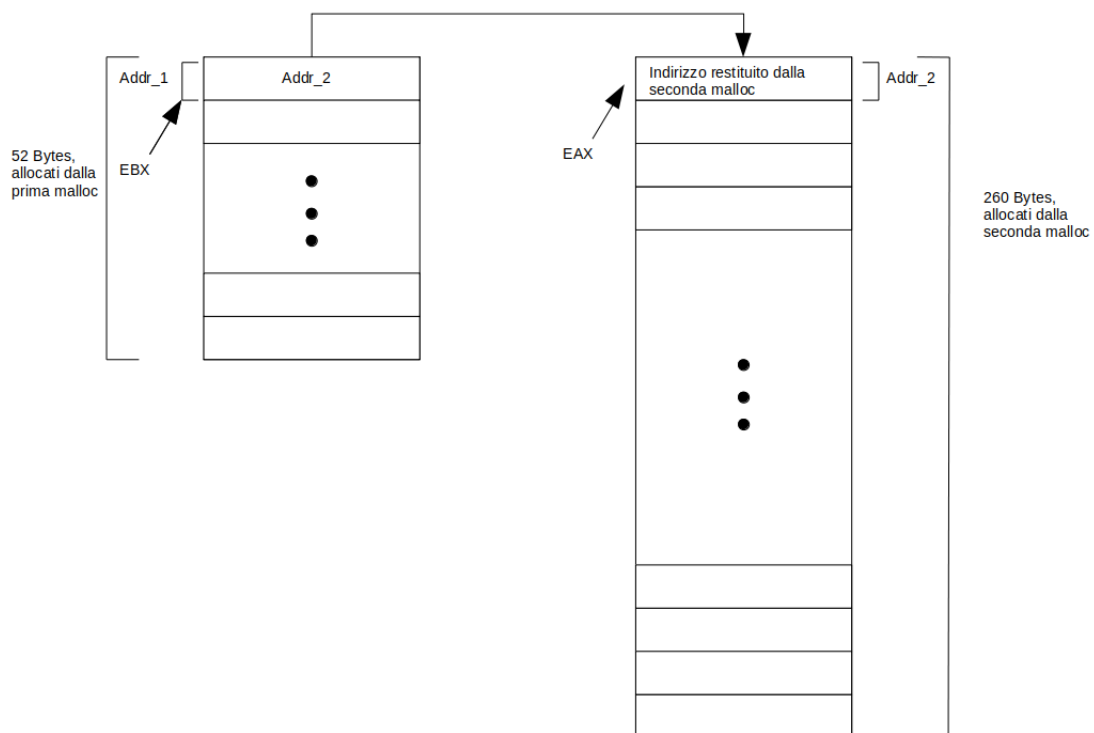
4.7 FUN_00401530_retrieve_subkeys_and_values

La prima cosa che fa questa funzione è allocare 52 bytes con un'invocazione di `MSVCRT.DLL::malloc`. Quindi viene controllato il valore di ritorno di questa invocazione e, se è `NULL`, viene invocata una `MSVCRT.DLL::puts` per stampare la stringa "Memory allocation error" prima di terminare con una chiamata a `MSVCRT.DLL::exit`, con *exit code* pari a 1.

Se le cose vanno bene, invece, si salva il valore ritornato dentro il registro `EBX`. Si invoca poi una nuova `malloc` per allocare 260 bytes. Anche qui, si controlla il valore di ritorno e si eseguono le stesse istruzioni dell'invocazione precedente nel caso questo sia `NULL`.

Se anche la seconda `malloc` va a buon fine, l'indirizzo di ritorno viene salvato come valore dell'indirizzo restituito dalla prima invocazione.

Si è ottenuto quindi che la prima `malloc` alloca 60 bytes e, il valore dei primi 4 sarà l'indirizzo restituito dalla seconda:



In **Addr_1**, che è l'indirizzo a cui punta **EBX**, è memorizzato un puntatore ad **Addr_2**

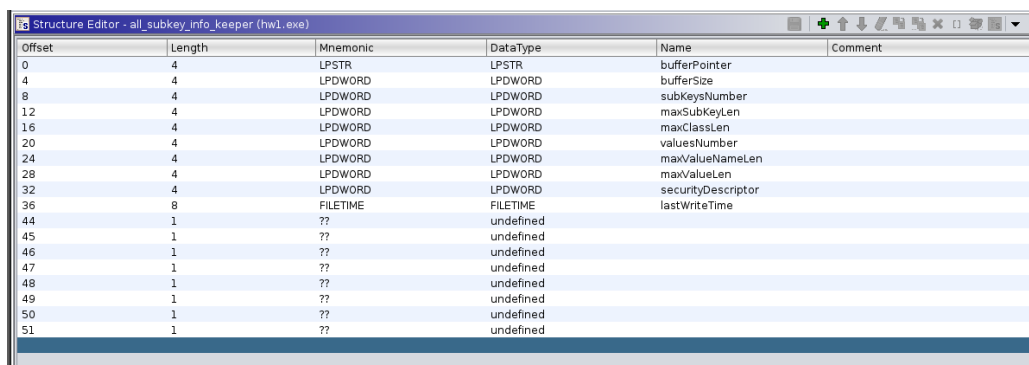
In seguito, viene salvato in **EDX** il valore contenuto all'offset *32* con base **EBX**; supponendo sia un array di puntatori, si prende l'ottavo. Poi c'è una cosa interessante: la zona di memoria puntata da **EAX**, che conserva ancora l'indirizzo restituito dalla seconda **malloc**, viene inizializzata a un **byte ptr** con valore 0; si sta settando quell'area di memoria a **'\0'**. Si procede caricando in **ECX** il valore contenuto in **EBX+36**: si sta memorizzando il nono elemento. Nell'area puntata da **EBX+4** si memorizza il valore 260, in quella puntata da **EBX+8** il valore 0.

Ipotesi: sembra che, quella puntata da **EBX**, sia una struttura dati in cui il primo campo memorizza un buffer, il secondo la sua grandezza e il terzo i byte effettivamente usati.

Il codice continua preparando l'invocazione della funzione **ADVAPI32.DLL:RegQueryInfoKeyA**: vengono messi sullo stack i seguenti parametri (riportati qui nell'ordine in cui li vede la funzione invocata):

1. Il parametro passato a questa funzione. La documentazione dice che questo è, infatti, l'*handle* verso un registro aperto. È quello ottenuto nella funzione `FUN_004018a0_open_handle_to_reg_key`.
2. Il puntatore al buffer memorizzato in `EBX`. Questo parametro è opzionale: è un `LPSTR` che punta a un buffer che riceve la classe della chiave.
3. Il valore memorizzato nell'area indicizzata da `EBX+4`, cioè 260, la dimensione del buffer allocato con la seconda `malloc`. Questo è un `LPDWORD` che punta a una variabile contenente la dimensione del buffer passato come parametro precedente, come da `ipotesi`.
4. Il valore 0. Questo campo è riservato e deve essere `NULL`.
- Vengono passati poi i byte che vanno da `EBX+8` a `EBX+32`: sono 6 elementi di 4 bytes ciascuno. Sono infatti 6 `LPDWORD` che memorizzano, in ordine:
 5. Un puntatore a una variabile che riceve il numero di sotto-chiavi contenute nella chiave specificata. Non è, come si pensava nell'`ipotesi`, il numero di bytes utilizzati.
 6. Un puntatore a una variabile che riceve la dimensione della sotto-chiave con il nome più lungo, senza il terminatore di stringa
 7. Un puntatore a una variabile che riceve la dimensione della stringa più lunga che specifica una classe, anche qui senza terminatore di stringa.
 8. Un puntatore a una variabile che riceve il numero di valori associati alla chiave passata.
 9. Un puntatore a una variabile che riceve la dimensione del nome più lungo tra tutti i valori associati alla chiave.
 10. Un puntatore a una variabile che riceve la dimensione del componente dati più grande tra tutti i valori associati alla chiave, espresso in bytes.
11. Il valore di `EBX+32`. Questo è un `LPDWORD` e punta a una variabile che riceve la dimensione del *security descriptor* della chiave, in bytes.
12. L'indirizzo di `EBX+36`. Questo parametro è un `PFILETIME`, che punta a una struttura che riceve l'ultimo istante di tempo in cui la chiave o uno dei suoi valori è stato modificato. Di conseguenza, in `EBX+36` è memorizzata una struttura `FILETIME`.

Si può quindi definire una struttura dati di 52 byte, che chiamiamo `all_subkey_info_keeper`. Questa struttura è, per ora, definita nel seguente modo:



Offset	Length	Mnemonic	DataType	Name	Comment
0	4	LPSTR	LPSTR	bufferPointer	
4	4	LPDWORD	LPDWORD	bufferSize	
8	4	LPDWORD	LPDWORD	subKeyNumber	
12	4	LPDWORD	LPDWORD	maxSubKeyLen	
16	4	LPDWORD	LPDWORD	maxClassLen	
20	4	LPDWORD	LPDWORD	valuesNumber	
24	4	LPDWORD	LPDWORD	maxValueNameLen	
28	4	LPDWORD	LPDWORD	maxValueLen	
32	4	LPDWORD	LPDWORD	securityDescriptor	
36	8	FILETIME	FILETIME	lastWriteTime	
44	1	??	undefined		
45	1	??	undefined		
46	1	??	undefined		
47	1	??	undefined		
48	1	??	undefined		
49	1	??	undefined		
50	1	??	undefined		
51	1	??	undefined		

Rimangono ancora altri bytes da definire.

Quando la funzione termina, si controlla il valore di ritorno: se non è 0, viene stampata la stringa "RegQueryInfoKey failed: key not found" tramite la funzione `MSVCRT.DLL::puts` e viene poi ritornato il valore 0.

Se la funzione ha avuto successo, invece, si controlla il terzo campo della struttura, e cioè il numero di sotto-chiavi.

Se questo valore non è 0 inizia un ciclo che, come prima istruzione, alloca 16 bytes con la funzione `malloc`; si salva il valore di ritorno nel registro `EBP` e si controlla il suo valore.

Nota: i valori restituiti da `malloc` sono salvati tutti: in `EBX` c'è il primo, ed è la struttura dati di tipo `all_subkey_info_keeper`; nel primo campo di `EBX` c'è il buffer; e infine questi 16 bytes sono salvati in `EBP`.

Se `malloc` fallisce, l'esecuzione termina con codice di uscita 1, stampando il messaggio di errore "Memory allocation error". Se tutto va bene, invece, si inizializza il valore dei bytes allocati.

Anche questa sembra essere una struttura dati: a suggerirlo è il fatto che vengono inizializzati gli indirizzi ottenuti con un offset a partire dall'indirizzo base restituito da `malloc`. I campi vengono inizializzati nel seguente modo:

Offset	Length	Mnemonic	DataType	Name	Comment
0	4	all_subkey_info_keeper *	all_subkey_info_keeper *	regQueryInfoKey_helper	
4	4	undefined4 *	undefined4 *	init_null	
8	1	??	undefined		
9	1	??	undefined		
10	1	??	undefined		
11	1	??	undefined		
12	4	LPDWORD	LPDWORD	maxSubkeyLen	

Anche qui rimangono ancora altri bytes da definire e non è ancora chiaro a cosa serva il secondo campo.

Questa struttura dati, come detto, viene memorizzata in EBP, e viene rinominata in `helper_retrieve_subkeys`.

Il codice continua invocando ancora `malloc` e il numero di byte da allocare è pari a `maxSubKeyLen`. Come al solito, si controlla se la funzione ha avuto successo e in caso di fallimento termina l'esecuzione dell'applicazione. Il valore di ritorno viene memorizzato nel terzo campo della struttura in EBP, ma anche sullo stack, nella variabile `local_54`. Il riferimento alla struttura dati in EBP viene salvato anche in EDI. Il terzo campo, che era rimasto indefinito, è quindi un puntatore a un'area di memoria della dimensione specificata nel quarto campo, cioè `maxSubKeyLen`.

Si prepara l'invocazione della funzione `ADVAPI32.DLL::RegEnumKeyExA`, inserendo sullo stack i seguenti valori:

1. Il primo parametro della funzione, cioè l'handle al registro aperto.
2. Il valore del registro ESI, impostato precedentemente a 0. Questo valore è un `DWORD` che rappresenta l'indice della sottochiave da recuperare. Nella prima invocazione di questa funzione, questo parametro deve essere 0, mentre deve essere poi incrementato per invocazioni successive.
3. Il buffer di `maxSubKeyLen` bytes memorizzato al terzo campo della struttura in EBP. Infatti questo è un puntatore a un buffer che riceve il nome della sotto-chiave, includendo il terminatore di stringa.
4. Il quarto campo della struttura di tipo `helper_retrieve_subkeys`, che rappresenta il puntatore alla dimensione del buffer e, in questo caso, vale `maxSubKeyLen`. La documentazione prevede infatti un pointer a una variabile che specifica la dimensione del buffer passato nel parametro precedente.
5. Il valore 0. Parametro riservato: come da documentazione, è `NULL`.
6. Il valore 0. Qui si può avere `NULL` oppure un puntatore a un buffer che riceve una classe della sotto-chiave definita dall'utente.

7. Il valore 0. Anche qui si può avere `NULL`. Se, però, nel parametro precedente c'è un buffer, qui va inserita la sua dimensione.
8. Il valore di `ECX`, che contiene l'indirizzo dell'ottavo campo della struttura `all_subkey_info_keeper`. Questo parametro è un `PFILETIME` in cui verrà inserito l'istante di tempo in cui la sotto-chiave è stata scritta l'ultima volta.

Si controlla il valore di ritorno della funzione: se tutto è andato bene, si incrementa il valore in `ESI`: questo registro è usato come secondo parametro di `RegEnumKeyExA` e, come detto, deve essere un valore incrementale per indicizzare le varie sotto-chiavi; può essere interpretato come un indice che parte da 0. Si effettua quindi il confronto tra il valore del campo `subKeysNumber` della struttura dati in `EBX` e questo valore incrementato. Se `subKeysNumber` è minore o uguale al contatore si esce dal ciclo, perché tutte le chiavi sono state controllate, altrimenti si riparte con una nuova iterazione. Se invece c'è stato un errore, e quindi il valore di ritorno è diverso da 0, si salva il valore contenuto in `EBP+0x4` nel registro `EDI` e ci si prepara a un'invocazione della funzione `MSVCRT.DLL::free`. A questa funzione è passato di `EBP`, che punta all'ultima istanza allocata della struttura dati `helper_retrieve_subkeys`. Prima della sua invocazione, si incrementa comunque il contatore in `ESI`. Dopo la `free`, si confronta il valore del campo `subKeysNumber` della struttura dati in `EBX` con il valore di `ESI`, quindi dell'indice incrementato. Se `subKeysNumber` è maggiore stretto, si riparte con il ciclo, altrimenti si esce.

Nota: quando le cose vanno bene, il ciclo ricomincia con `EDI` impostato al valore di `EBP`, cioè al valore dell'istanza corrente della struttura dati. All'iterazione successiva, viene allocata una nuova istanza della struttura e il secondo campo viene impostato a `EDI`, cioè all'istanza precedente. Si sta quindi costruendo una *lista collegata*.

Quando le cose vanno male, invece, l'attuale istanza della classe non è inclusa nella catena, e viene anzi rilasciata. Il registro `EDI` viene impostato al valore dell'istanza creata all'iterazione precedente, che è appunto memorizzata nel secondo campo della struttura: si sta escludendo dalla catena l'istanza attuale.

Una volta usciti dal ciclo, si memorizza nella struttura dati di `EBX`, all'offset 44, il valore di `EAX`: attualmente, in questo registro è salvato il valore di `EDI`, che contiene la base della lista collegata. Possiamo quindi definire il campo della struttura dati `all_subkey_info_keeper` all'offset 44 come la base della lista collegata delle strutture di tipo `helper_retrieve_subkeys`.

Se invece nel ciclo non ci si entra proprio, perché il valore di *subKeyNumber* restituito dalla funzione `RegQueryInfoKeyA` è pari a 0, allora si memorizzerà il valore `NULL`.

Il codice continua con un controllo sul campo *valuesNumber* della struttura dati `all_subkey_info_keeper`. Se il numero dei valori non è 0 inizia un ciclo, dove, per prima cosa, vengono allocati con `malloc` la bellezza di 16660 bytes, che corrispondono a circa 16 Megabites e l'indirizzo di ritorno viene memorizzato in `EBP`; ovviamente c'è un controllo sul valore restituito e, in caso di errore, si termina mostrando la solita stringa *"Memory allocation error"*.

Nota: prima sono state controllate tutte le sotto-chiavi, adesso sta preparando il controllo su tutti i valori associati alla sotto-chiave. È lecito pensare che anche questa sia una struttura dati e che anche qui verrà costruita una lista collegata.

In effetti, la logica è la stessa: viene memorizzato all'offset 4 il valore di `ESI`, che contiene inizialmente un puntatore a `NULL`, ma poi conterrà l'istanza della struttura creata all'iterazione precedentemente. In seguito si imposta come primo valore il valore di `EBX`, cioè il puntatore alla struttura dati di tipo `all_subkey_info_keeper`. Si memorizza poi all'offset 16392 il valore 16383, mentre all'offset 8 si inizializza il terminatore di stringa. Si inserisce all'offset 16656 il valore 256; si sono inizializzati i suoi campi. Chiamiamo questa struttura `helper_retrieve_values`.

Si prepara poi l'invocazione di `ADVAPI32.DLL::RegEnumValueA`: vengono passati sullo stack i seguenti parametri:

1. Il parametro della funzione, cioè l'handle alla chiave.
2. Il valore 0, contenuto nel registro `EDI`. Questo è come il parametro 2 della chiama a `RegEnumKeyExA`: è un valore incrementale che indicizza il valore da recuperare.
3. L'indirizzo dell'offset 8 a partire dall'indirizzo allocato con `malloc`. Rappresenta un buffer che riceve il nome del valore. Include il terminatore di stringa.
4. L'indirizzo dell'offset 16392 a partire dall'indirizzo della struttura dati `helper_retrieve_values`. È un puntatore a un'area di memoria contenente la dimensione del buffer precedente, ma non include il terminatore di stringa. Il valore puntato è inizializzato al valore 16383,

quindi all'offset 8 è contenuto un buffer di 16834 caratteri. Torna con ciò che è stato inserito nella struttura precedentemente.

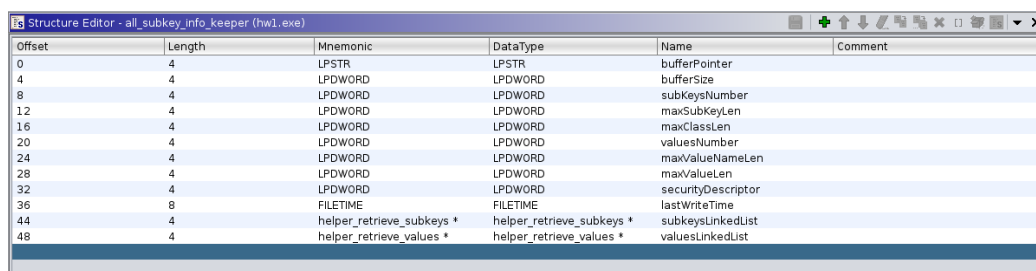
5. Il valore 0. Parametro riservato e deve essere **NULL**
6. L'indirizzo dell'offset 16396. Puntatore a una variabile che riceve un codice che indica il tipo di dato memorizzato nel valore specificato. Questo sarà, nella struttura, un **LPDWORD**.
7. L'indirizzo dell'offset 16400. Puntatore a un buffer che riceve i dati relativi al valore.
8. L'indirizzo dell'offset 16656, il cui valore puntato è inizializzato a 256. Puntatore a una variabile che specifica la dimensione del buffer precedente, in bytes. Quindi l'allocazione di memoria all'offset 16400 è un buffer di 256 bytes. Le dimensioni tornano.

Quando l'invocazione ritorna, si controlla il suo valore di ritorno. Se è andato tutto bene, il valore **EDI**, che è l'indice del valore che viene recuperato, viene incrementato di 1; si controlla quindi che questo valore non sia maggiore del campo all'offset 20 di **EBX**, cioè *valuesNumber*. Se il contatore è minore, si riparte nel ciclo, altrimenti si esce.

Se invece il valore di ritorno è diverso da zero, si salva il valore di **EBP+0x4** nel registro **ESI** prima di inserire **EBP** sullo stack per invocare la **free**: in questo modo si esclude dalla lista collegata l'istanza attuale, che ha generato errore. Si continua poi incrementando il contatore e facendo il controllo necessario a determinare se uscire dal ciclo o meno.

Una volta usciti dal ciclo, si memorizza il valore salvato nel registro **ESI**, che contiene la lista collegata appena creata, all'interno dell'offset 0x30 di **EBX**, che memorizza ancora la struttura dati di tipo **RegQueryInfoKey_helper_struct**. Infine, il valore contenuto in questo registro viene ritornato dalla funzione.

Le strutture dati, al termine della funzione, sono così definite:



Offset	Length	Mnemonic	DataType	Name	Comment
0	4	LPSTR	LPSTR	bufferPointer	
4	4	LPDWORD	LPDWORD	bufferSize	
8	4	LPDWORD	LPDWORD	subKeysNumber	
12	4	LPDWORD	LPDWORD	maxSubKeyLen	
16	4	LPDWORD	LPDWORD	maxClassLen	
20	4	LPDWORD	LPDWORD	valuesNumber	
24	4	LPDWORD	LPDWORD	maxValueNameLen	
28	4	LPDWORD	LPDWORD	maxValueLen	
32	4	LPDWORD	LPDWORD	securityDescriptor	
36	8	FILETIME	FILETIME	lastWriteTime	
44	4	helper_retrieve_subkeys *	helper_retrieve_subkeys *	subkeysLinkedList	
48	4	helper_retrieve_values *	helper_retrieve_values *	valuesLinkedList	

Offset	Length	Mnemonic	DataType	Name	Comment
0	4	all_subkey_info_keeper *	all_subkey_info_keeper *	regQueryInfoKey_helper	
4	4	helper_retrieve_subkeys *	helper_retrieve_subkeys *	next	
8	4	LPSTR	LPSTR	buffer	
12	4	LPDWORD	LPDWORD	bufferSize	

Offset	Length	Mnemonic	DataType	Name	Comment
0	4	all_subkey_info_keeper *	all_subkey_info_keeper *	helperRegQueryInfoKey	
4	4	helper_retrieve_values *	helper_retrieve_values *	next	
8	16384	char[16384]	char[16384]	buffer	inizializzato a '\0'
16392	4	LPDWORD	LPDWORD	bufferSize	16383
16396	4	LPDWORD	LPDWORD	dataType	
16400	256	BYTE[256]	BYTE[256]	data	
16656	4	DWORD	DWORD	dataBufferSize	256

Riassumendo: la funzione esplora la chiave passata in input per cercare i valori e le sotto-chiavi ad essa collegate. Memorizza il tutto in una struttura dati che viene poi ritornata. In particolare, nella struttura dati ritornata ci sono due campi che memorizzano le *liste collegate* rispettivamente di sotto-chiavi e valori che non hanno generato errori durante la loro analisi. Questa funzione, terminando, restituisce il controllo a `FUN_004018a0_open_handle_to_reg_key` che, a questo punto, non fa altro che restituire al `main` la struttura dati `all_subkey_info_keeper` creata.

4.8 FUN_00401790_print_catched_info

Questa funzione è l'ultima invocata dal `main` che non è stata ancora analizzata. Prende in input il valore ritornato da `FUN_004018a0_open_handle_to_reg_key`, che è un puntatore a `all_subkey_info_keeper`.

Per analizzare questa funzione, conviene guardare il decompilato: risulta infatti molto chiaro che questa funzione ha lo scopo di stampare tutto ciò che è stato precedentemente creato e salvato nella struttura dati passata come primo parametro.