Analisi del Malware – Relazione Homework 1

Studente: Andrea Pepe Matricola: 0315903

OBIETTIVO

Acquisire informazioni sul funzionamento del programma "hw1.exe" ed individuare le strutture di dati fondamentali utilizzate.

DESCRIZIONE PRELIMINARE

Il file eseguibile oggetto dell'analisi è in formato Portable Executable (PE), dunque un eseguibile per sistemi operativi Windows. Le finalità e la contestualizzazione del programma sono inizialmente ignote, dunque, a priori, è impossibile effettuare un'analisi mirata a specifiche funzionalità del programma o ipotesi sul suo funzionamento.

In seguito ad un'analisi preliminare dell'eseguibile effettuata con *Ghidra*, sono state recuperate alcune importanti informazioni, tra cui:

- Processore: x86 (a 32 bit);
- Endianess: Little Endian;
- Compilatore: visualstudio: unknown

INDIVIDUAZIONE DEL MAIN

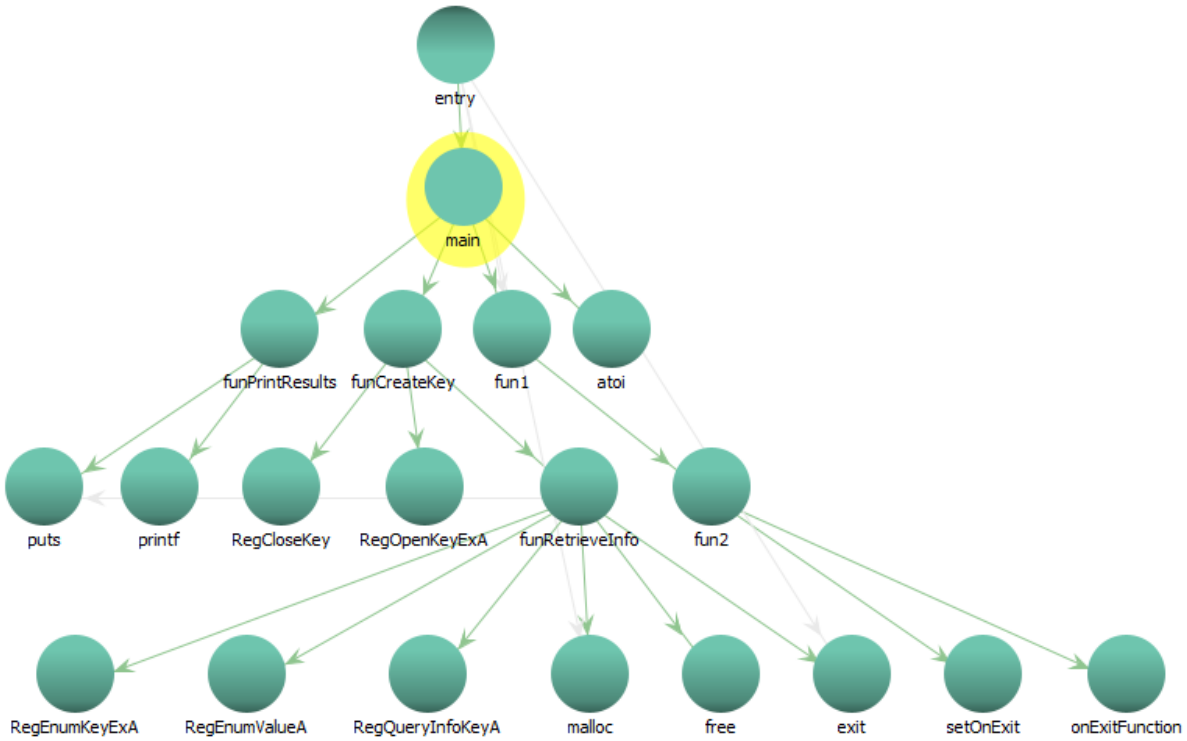
È stato utilizzato il tool di disassemblaggio messo a disposizione da *Ghidra* per disassemblare l'eseguibile, così da poterne eseguire un'analisi statica, interpretando le istruzioni in linguaggio assembly.

Il primo obiettivo che ci si è posto di raggiungere nel processo di analisi è stato l'individuazione della funzione main del programma. A tale scopo, è stato di fondamentale importanza l'utilizzo dello strumento *Function Call Graph* di *Ghidra*, in quanto, in particolare, ha fornito una panoramica del grafo delle chiamate a funzione della funzione "entry", identificata dal disassembler. Espandendo i livelli del grafo delle chiamate, si è cercato di identificare delle possibili candidate ad essere la funzione main. È saltata subito all'occhio una parte del grafo in cui figurano chiamate alle funzioni "printf", "puts" e "atoi", buoni indizi per individuare funzioni scritte dal programmatore. Dunque, si è utilizzato il tool *Defined Strings* per appurare se, tra le varie stringhe utilizzate dal programma, ce ne fossero alcune che potessero essere considerate con buona probabilità delle stringhe user-defined. Sono state individuate le seguenti stringhe:

- "RegQueryInfoKey failed: key not found";
- "Memory allocation error";
- "Class: %s\n";
- "Security descriptor: 0x%lx\n";
- "Time: %08lx%08lx\n";
- "Sub-keys:";
- "\t%s\n";

- “Values:”
- “\t%s: [%lu] ”;
- “ %02x”;
- “ (%s)\n”;
- “SYSTEM\\ControlSet001\\Control”.

Effettivamente, tali stringhe sono impiegate dalle funzioni presenti in quella parte di *Function Call Graph* presa in esame, che, per concretezza, viene riportata di seguito (figurano già delle ridenominazioni delle funzioni che sono state effettuate in seguito):



Inoltre, il disassembler di *Ghidra* ha individuato che alla funzione rinominata “main” vengono passati due parametri di ingresso, entrambi di tipo int. Ciò ha avvalorato ulteriormente l’ipotesi che si potesse trattare realmente della funzione main: infatti, cambiando il tipo di dato del secondo parametro da int a char**, il codice della funzione ottenuto utilizzando il decompilatore è risultato molto più ragionevole che in precedenza.

STRUTTURA E FINALITÀ DEL PROGRAMMA

Si è cercato di acquisire il prima possibile degli indizi sulle finalità del programma, così da poter agevolare la formulazione di ipotesi plausibili sul funzionamento delle varie componenti del programma stesso.

A tale scopo, si è partiti ad analizzare la funzione che invoca “printf” e “puts” (la funzione denominata “funPrintResults” nel grafo), poiché probabilmente è stata utilizzata per stampare in output dei dati acquisiti dal programma. Dunque, l’eventuale individuazione di tali dati, avrebbe di certo favorito la ricerca delle strutture di dati utilizzate e rivelato auspicabilmente qualcosa riguardo le intenzioni del programmatore. Da una prima analisi delle istruzioni assembly della funzione, si è compreso che essa stampa su standard output i seguenti valori:

- Una stringa che rappresenta una classe, desunta dalla stringa “Class: %s\n” data in input alla printf come primo parametro;
- Un valore numerico che rappresenta un security descriptor;
- Dei valori numerici associati ad un tempo, desunti dalla stringa “Time: %08lx%08lx\n”;
- Se presenti, delle stringhe associate ad un insieme di *Sub-keys*, all’interno di un while loop;
- Se presenti, delle stringhe e dei valori numerici associati ad un insieme di *Values* all’interno di un while loop; inoltre, è presente un ulteriore loop annidato in cui avviene la stampa di bytes.

Di certo, i dati più interessanti sono le *Sub-keys* e i *Values*. Dunque, l’analisi è proseguita cercando di capire cosa fossero tali dati e in che modo venissero recuperati.

Riguardando il grafo delle chiamate a funzione illustrato in precedenza, si nota la presenza di funzioni della DLL ADVAPI32 che, nella segnatura, fanno riferimento a *Keys* e *Values*:

- RegOpenKeyExA;
- RegCloseKey;
- RegEnumKeyExA;
- RegEnumValueA;
- RegQueryInfoKeyA.

Quindi, è risultato naturale proseguire l’analisi dalla funzione che invoca la RegOpenKeyExA, denominata dunque “funCreateKey”. Quest’ultima viene invocata dal main e prende come parametri in ingresso un handle ad una chiave (HKEY) ed una stringa. Sostanzialmente, la funCreateKey non fa altro che invocare la RegOpenKeyExA passandole come:

- 1° parametro: il valore di tipo HKEY che ha ricevuto in input, che rappresenta un handle ad una chiave del registro di sistema, che sia aperta oppure una delle chiavi predefinite;
- 2° parametro: la stringa che ha ricevuto in input, la quale indica il nome di una sottochiave del registro che si intende aprire;
- 3° parametro: impostato a zero;
- 4° parametro: il codice 0xf003f, che corrisponde alla macro KEY_ALL_ACCESS per aprire la sottochiave con tutti i permessi d’accesso;
- 5° parametro: l’indirizzo di una variabile di tipo HKEY dove verrà restituito un nuovo handle alla sottochiave aperta.

Dopodiché, l’handle ottenuto viene passato ad un’altra funzione (denominata in seguito “funRetrieveInfo”), la quale si presume possa utilizzarlo per recuperare i dati che verranno successivamente stampati dalla funPrintResults, essendo che una chiave di registro può contenere al suo interno sia altre sottochiavi che valori. Infine, viene invocata la RegCloseKey per chiudere l’handle ottenuto in precedenza. A questo punto, si è cercato di capire quale sottochiave il programmatore intendesse aprire e, quindi, sono stati analizzati i parametri che la funzione main passa alla funCreateKey.

Dalle istruzioni in assembly e dal codice generato dal decompilatore, si è appreso che, nel caso in cui il parametro argc del main fosse minore di 3, alla funCreateKey viene passato come primo parametro il valore 0x80000002, che corrisponde alla macro HKEY_LOCAL_MACHINE, e come secondo parametro la stringa “SYSTEM\\ControlSet001\\Control”. Dunque, il programmatore cerca di ottenere un handle alla sottochiave della chiave di registro predefinita HKEY_LOCAL_MACHINE, il cui nome corrisponde alla stringa passata come parametro. Tale sottochiave conserva la configurazione del sistema operativo corrispondente all’ultima volta in cui ne è stato eseguito il boot.

Il passo successivo è stato l'analisi della funzione ridenominata "funRetrieveInfo" e l'individuazione delle strutture dati utilizzate dal programmatore.

INDIVIDUAZIONE DELLE STRUTTURE DI DATI

La funzione funRetrieveInfo prende come unico parametro di input un handle ad una chiave di registro, che, come visto in precedenza, sarà l'handle della nuova chiave aperta dalla RegOpenKeyExA. La prima operazione effettuata all'interno della funzione è una chiamata alla funzione di libreria "malloc" passandole come parametro il valore 52. Successivamente, i primi 4 bytes dell'area di memoria appena allocata nello heap vengono occupati da un ulteriore puntatore ad una nuova area di memoria contigua di 260 bytes, allocata sempre utilizzando la malloc, di cui il primo byte viene inizializzato con il carattere terminatore di stringa '\0'. I secondi 4 bytes dell'area di memoria da 52, sono inizializzati con il valore intero 260, proprio la taglia dell'altra area di memoria allocata; invece, i 4 bytes di memoria successivi sono impostati a 0.

Dunque, viene allocata una porzione contigua di 52 bytes in cui il primo campo è un puntatore ad un array di caratteri e i successivi due sono valori interi, quindi tipi di dato differenti. Ciò inizia a far pensare che i 52 bytes possano essere la taglia di un tipo di dato *struct* definito dal programmatore. Tale ipotesi trova ulteriori conferme andando avanti nell'analisi delle istruzioni assembly della funzione: infatti, vengono caricati sullo stack gli indirizzi delle locazioni di memoria dei vari campi della presunta *struct*, ottenendoli tramite offset dall'indirizzo base, per poterli passare come parametri alla funzione di libreria RegQueryInfoKeyA. Tale funzione prende come unico parametro di input un handle ad una chiave di registro di cui ne recupera le informazioni e le restituisce agli indirizzi passati come parametri di output. Utilizzando la documentazione, sono stati individuati i tipi di dato di alcuni dei campi della *struct*, che è stata definita anche in *Ghidra* per agevolare l'analisi, assegnandole il nome "struct_query_info":

Bytes	Tipo di dato	Nome campo	Descrizione
4	LPSTR	className	Classe user-defined della chiave
4	DWORD	classNameLen	Lunghezza del className
4	DWORD	numSubKeys	Numero di sottochiavi della chiave
4	DWORD	maxSubKeyLen	Lunghezza della più lunga delle sottochiavi
4	DWORD	maxClassLen	Lunghezza della più lunga stringa che specifica la classe di una sottochiave
4	DWORD	numValues	Numero di values della chiave
4	DWORD	maxValueNameLen	Lunghezza del più lungo nome di un valore
4	DWORD	maxValueLen	Dimensione del più lungo componente dati tra i valori della chiave, in bytes
4	DWORD	securityDescriptor	Lunghezza del security descriptor della chiave
8	FILETIME	lastWriteTime	Struttura che contiene il tempo in cui la chiave o uno dei suoi valori sono stati modificati l'ultima volta
8	undefined	???	???

Ci sono ancora 8 bytes da ricoprire, ma, di fatto le *Sub-keys* e i *Values* non sono ancora stati recuperati, quindi questi 8 bytes potrebbero fare riferimento a tali dati. Infatti, continuando ad analizzare le istruzioni assembly della funzione funRetrieveInfo, con l'ausilio del tool *Function Graph* di *Ghidra*, si

vede che viene valutato se il campo numSubKeys è maggiore di zero e, in tal caso, si entra in un ciclo while in cui, ad ogni iterazione, viene chiamata la funzione di libreria RegEnumKeyExA, per recuperare informazioni su una singola sottochiave ogni volta che viene invocata. Prima di ciò, nel ciclo while viene effettuata una malloc di 16 bytes passandone gli indirizzi a offset di 8 e 12 bytes alla RegEnumKeyExA per salvare le informazioni prodotte in output, in particolare il nome della sottochiave e la sua lunghezza. Inoltre, i primi 4 bytes dei 16 allocati sono un puntatore alla *struct* di tipo struct_query_info allocata in precedenza, mentre i successivi 4 vengono settati con il riferimento alla area di memoria da 16 bytes allocata nella precedente iterazione del loop. Rappresentano quindi un campo next di una struttura dati di tipo lista collegata. Quindi, viene individuata una ulteriore *struct* che rappresenta un nodo della lista contenente informazioni su una sottochiave; tale struttura è stata denominata "struct_subkey":

Bytes	Tipo di dato	Nome campo	Descrizione
4	struct_query_info*	queryInfo	Puntatore alla struttura con le informazioni della query sulla chiave
4	struct_subkey*	next	Puntatore al successivo nodo nella lista di sottochiavi
4	LPSTR	subKeyName	Puntatore al nome della sottochiave
4	DWORD	nameLen	Lunghezza del nome della sottochiave

La lista viene costruita effettuando degli inserimenti in testa: ogni nuovo nodo inserito viene fatto puntare alla testa della lista e, di conseguenza, diviene esso stesso la nuova testa della lista. Qualora la RegEnumKeyExA restituisca un valore di errore, il nodo appena inserito in testa alla lista viene rimosso e il suo successore diviene nuovamente la testa della lista.

Infine, una volta terminate le iterazioni nel while loop, il penultimo campo della struttura "struct_query_info" viene settato con il puntatore alla testa della lista di sottochiavi, mantenendo così un riferimento alla lista di *Sub-keys*.

In maniera analoga, in un successivo while loop vengono recuperati dati riguardanti i *Values* invocando la funzione di libreria RegEnumValueA e viene costruita un'altra lista. Anche in questo caso è stata indivisuata una *struct* per i nodi della lista, denominata "struct_regValue":

Bytes	Tipo di dato	Nome campo	Descrizione
4	struct_query_info*	queryInfo	Puntatore alla struttura con le informazioni della query sulla chiave
4	struct_regValue*	next	Puntatore al successivo nodo nella lista di valori
16384	CHAR[16384]	valueName	Buffer contenente il nome del valore
4	DWORD	valueNameLen	Lunghezza del nome del valore
4	DWORD	type	Codice che identifica il tipo dei dati conservati nel valore
256	BYTE[256]	data	Buffer contenente i dati del valore
4	DWORD	dataLen	Lunghezza in bytes dei dati

Analogamente, la lista viene costruita con degli inserimenti in testa e il riferimento alla testa della lista è assegnato all'ultimo campo della *struct* "struct_query_info", la quale adesso risulta completa. Si riporta di seguito una definizione delle struct citate in precedenza il linguaggio C, così da ricapitolare le informazioni ottenute:

```
struct struct_query_info{
	LPSTR className,
	DWORD classNameLen,
	DWORD numSubKeys,
	DWORD maxSubKeyLen,
	DWORD maxClassLen,
	DWORD numValues,
	DWORD maxValueNameLen,
	DWORD maxValueLen,
	DWORD securityDescriptor,
	FILETIME lastWriteTime,
	struct_subkey *listOfSubKeys,
	struct_regValue *listOfValues
};

struct struct_subkey{
	struct_query_info *queryInfo,
	struct_subkey *next,
	LPSTR subKeyName,
	DWORD nameLen
};

struct struct_regValue{
	struct_query_info *queryInfo,
	struct_regValue *next,
	CHAR[16384] valueName,
	DWORD valueNameLen,
	DWORD type,
	BYTE[256] data,
	DWORD dataLen
};
```

La funzione funRetrieveInfo restituisce al chiamante il puntatore alla struttura di dati di tipo struct_query_info allocata. A sua volta, la funzione funCreateKey, lo restituisce al main dopo aver

invocato la RegCloseKey e, quindi, questo puntatore viene passato in input alla funPrintResults per poterne stampare i dati contenuti.

In particolare, vengono stampati su standard output:

- Il nome della classe user-defined della chiave;
- La lunghezza in bytes del security descriptor della chiave;
- Il valore temporale dell'ultima modifica delle sottochiavi o dei valori;
- Per ogni sottochiave, navigando la lista, ne viene stampato il nome;
- Per ogni valore, navigando la lista, ne viene stampato: il nome, il codice rappresentante il tipo di valore e, se presenti, i dati contenuti, sia come bytes che come stringa (tipicamente, un valore contiene dati in formato stringa o sequenze di bytes, ma talvolta può contenere anche un intero a 32 bit, un gruppo di stringhe, una stringa con riferimenti a variabili d'ambiente o un valore amorfo).

ANALISI APPROFONDITA DELLA FUNZIONE MAIN

Come ultimo passo del processo di reversing, vengono analizzati dei dettagli della funzione main che non sono stati approfonditi inizialmente. Innanzitutto, come prima operazione, il main invoca una funzione ridenominata "fun1". Tale funzione è invocata sia dal main ma anche dalla entry. Inoltre, il suo comportamento è determinato dal valore di una variabile globale di tipo intero: se il valore di tale variabile è diverso da zero, allora la fun1 non fa nulla e restituisce il controllo al chiamante; invece, se il valore della variabile globale è pari a zero, esso viene settato ad 1 e viene invocata un'altra funzione (ridenominata "fun2" nel grafo illustrato precedentemente). Tuttavia, si nota che, nella entry, la fun1 è invocata prima di chiamare il main: dunque, quando il main invocherà nuovamente fun1, il valore della variabile globale cui si è fatto riferimento in precedenza sarà di certo diverso da zero e, quindi, fun1 restituirà immediatamente il controllo al main senza fare nulla. È importante notare che, guardando i riferimenti alla variabile globale, si evince che essa viene acceduta sia in lettura che scrittura dalla sola funzione fun1, quindi il suo valore non può essere modificato in nessun'altra zona del codice.

Molto probabilmente, la chiamata a fun1 inserita nel main è frutto di una modifica del file eseguibile successiva alla sua generazione e il suo scopo era semplicemente quello di ostacolare il processo di analisi e reversing del codice, cercando di depistare l'analista.

Inoltre, immediatamente dopo la chiamata della fun1, nel main viene effettuato il controllo se argv[argc] è pari a NULL, che, se verificato, porta il main a terminare ritornando il valore 0. Questo controllo è sempre verificato, in quanto argv[argc] è sempre posto pari a NULL; dunque, di fatto, il programma quando viene eseguito non fa altro che terminare. Molto probabilmente questa operazione è una misura precauzionale legata esclusivamente allo svolgimento dell'homework, inserita per impedire l'analisi dinamica dell'eseguibile e l'utilizzo di debugger che avrebbero facilitato l'operazione di reversing.

Infine, si nota che i parametri passati alla funzione funCreateKey, che quindi rappresentano sostanzialmente la chiave del registro di sistema di cui si vogliono recuperare valori e sottochiavi, sono pari a HKEY_LOCAL_MACHINE e "SYSTEM\\ControlSet001\\Control" solo se argc è minore di 3. Quindi, passando in input al programma due parametri diversi, si può utilizzare il programma per ottenere dati riguardanti un'altra chiave.

RICOSTRUZIONE DEL PROGRAMMA IN LINGUAGGIO C

Si riporta di seguito una descrizione ad alto livello in linguaggio C del programma, frutto della ricostruzione del comportamento del programma, dedotta analizzando le istruzioni assembly e avvalendosi del supporto fornito dal decompilatore di *Ghidra*:

```
struct struct_query_info{
        LPSTR className,
        DWORD classNameLen,
        DWORD numSubKeys,
        DWORD maxSubKeyLen,
        DWORD maxClassLen,
        DWORD numValues,
        DWORD maxValueNameLen,
        DWORD maxValueLen,
        DWORD securityDescriptor,
        FILETIME lastWriteTime,
        struct_subkey *listOfSubKeys,
        struct_regValue *listOfValues
};

struct struct_subkey{
        struct_query_info *queryInfo,
        struct_subkey *next,
        LPSTR subKeyName,
        DWORD nameLen
};

struct struct_regValue{
        struct_query_info *queryInfo,
        struct_regValue *next,
        CHAR[16384] valueName,
        DWORD valueNameLen,
        DWORD type,
        BYTE[256] data,
        DWORD dataLen
```

```
};

typedef struct struct_query_info struct_query_info;
typedef struct struct_subkey struct_subkey;
typedef struct struct_regValue struct_regValue;


DWORD global_int;

void fun1(){

	/*
	This function is also called by the entry function.
	In particular, the entry function calls fun1() before calling the main():
	this implies that, when the main calls fun1(), the value of the global variable
	"global_int" has already been set to 1 or has a value that differs from 0.

	SO THIS FUNCTION DOES ABSOLUTELY NOTHING WHEN IT'S CALLED BY MAIN!!!

	Indeed, the only cross references to the variable "global_int" are made in the fun1() function.
	This means that global_int can be modified only by fun1(), so, after the first call to fun1(),
	global_int's value always differs from 0 !!!
	*/

	if(global_int != 0){
		return;
	}
	global_int = 1;
	// calling other stuff
	//fun2();
	return;
}

void funPrintResults(struct_query_info* info){
```

```
	printf("Class: %s\n",info->className);
	printf("Security descriptor: 0x%lx\n",info->securityDescriptor);
	printf("Time: %08lx%08lx\n",(info->lastWriteTime).dwLowDateTime, (info->last-
WriteTime).dwHighDateTime);

	// print subKeys names, if any
	struct_subkey* currSubKey = info->listOfSubKeys;
	if(currSubKey != NULL){
		puts("SubKeys:");
		do{
			printf("\t%s\n", currSubKey->subKeyName);
			currSubKey = currSubKey->next;
		} while(currSubKey != NULL);
	}

	// print values, if any
	struct_regValue* currValue = info->listOfValues;
	if(currValue != NULL){
		puts("Values:");
		do{
			printf("\t%s: [%lu] ", currValue->valueName, currValue->type);
			// print data, if any
			if(currValue->dataLen != 0){
				for(int i=0; i < currValue->dataLen; i++){
					printf(" %02x", currValue->data[i]);
				}
			}
			printf(" (%s)\n", currValue->data);
			currValue = currValue->next;
		} while(currValue != NULL)
	}
	return;
}
```

```
struct_query_info* funRetrieveInfo(HKEY param1){
	struct_subkey* subkeys = NULL;
	struct_regValue* values = NULL;

	struct_query_info* info = malloc(sizeof(struct_query_info)); // 52 bytes
	if(info == NULL) goto malloc_failed;
	info -> className = malloc(260);
	if(info -> className == NULL) goto malloc_failed;
	*(info -> className) = '\0';
	info->classNameLen = 260;
	info->numSubKeys = 0;

	// getting info
	DWORD queryRes = RegQueryInfoKeyA(param1, info->className, &info->classNameLen,
NULL, &info->numSubKeys, &info->maxSubKeyLen, &info->maxClassLen, &info->numValues, &info-
>maxValueNameLen, &info->maxValueLen, &info->securityDescriptor, &info->lastWriteTime);
	if(queryRes != ERROR_SUCCESS){
		puts ("RegQueryInfoKey failed: key not found");
		return NULL;
	}else{

		//retrieving subkeys and building the first linked list
		if(info->numSubKeys != 0){
			struct_subkey* next = NULL;   // last inserted subKey
			for(int i = 0; i < info -> numSubKeys; i++){
				subkeys = malloc(sizeof(struct_subkey));          // 16 bytes
				if(subkeys == NULL) goto malloc_failed;

				subkeys -> queryInfo = info;
				subkeys -> next = next;
				subkeys -> nameLen = info -> maxSubKeyLen;
				subkeys -> subKeyName = malloc(info->maxSubKeyLen);
				if(subkeys -> subKeyName == NULL) goto malloc_failed;
				// getting subkey info
```

```
                        if(RegEnumKeyExA(param1, i, subkeys->subKeyName, &subkeys-
>nameLen,
                        NULL, NULL, NULL, &info->lastWriteTime) != ERROR_SUCCESS){
                                // removing the item from the list: something went wrong

                                next = subkeys->next;   //the head of the list remains the same
                                free(subkeys);
                                subkeys = next;  // the head of the list is the last inserted struct
                        }else{
                                next = subkeys;                 // new head
                        }
                }
        }
        info->listOfSubKeys = subkeys;

        //retrieving values and building the second linked list
        if(info->numValues != 0){
                struct_regValue* nextVal = NULL;
                for(int j=0; j < info->numValues; j++){
                        values = malloc(sizeof(struct_regValue));       //16660 bytes
                        if(values == NULL) goto malloc_failed;

                        values->queryInfo = info;
                        values->next = nextVal;
                        values->valueNameLen = 16383;                   // MAX POSSIBLE
                        values->valueName[0] = '\0';
                        values->dataLen = 256;

                        if(RegEnumValueA(param1, j, values->valueName, &values->val-
ueNameLen, NULL,
                        &values->type, &values->data, &values->dataLen) != ERROR_SUC-
CESS){
                                nextVal = values->next;
                                free(values);
                        }
                        nextVal = values;
```

```
                        }
                }
                info->listOfValues = values;
                return info;
        }

malloc_failed:
        puts("Memory allocation error");
        exit(1);
}



struct_query_info* funCreateKey(HKEY hKey, LPCSTR subKey){
        struct_query_info* ret = NULL;
        HKEY newKey;
        if(RegOpenKeyExA(hKey, subKey, NULL, KEY_ALL_ACCESS, &newKey) == ERROR_SUC-
CESS){
                ret = funRetrieveInfo(newKey);
                RegCloseKey(newKey);            // close the handle to the previuosly opened key
        }
        return ret;
}




int main(int argc, char **argv){
        // function that typically does nothing
        fun1();

        if(argv[argc] == '\0'){
                /*
                always true; maybe to avoid leaking of useful informations
                for the reversing process by executing the program
                */
```

```
		return 0;
	}

	if(argc < 3){
		argv[1] = '\0';
		argv[2] = '\0';
	}

	HKEY hKey = atoi(argv[1]);
	if(HKEY == 0){
		hKey = HKEY_LOCAL_MACHINE;
	}

	char* subKeyString = argv[2];
	if(argv[2] == '\0'){
		subKeyString = "SYSTEM\\ControlSet001\\Control";
	}

	struct_query_info* info = funCreateKey(hKey, subKeyString);
	int ret = 1;
	if(info != NULL){
		funPrintResults(info);
		ret = 0;
	}
	return ret;
}
```