# Homework 1 Malware Analysis

Danilo Dell'Orco, 0300229

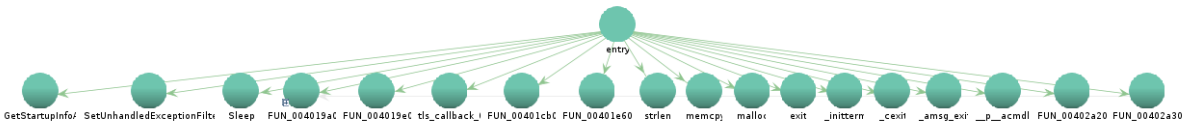
2021-10-27

# Homework 1 Malware Analysis - Danilo Dell'Orco 0300229

Il punto di partenza per capire il funzionamento di questo programma, è quello di individuare il *main*, in modo tale da poter seguire il flusso di esecuzione e le strutture dati che vengono man mano utilizzate.

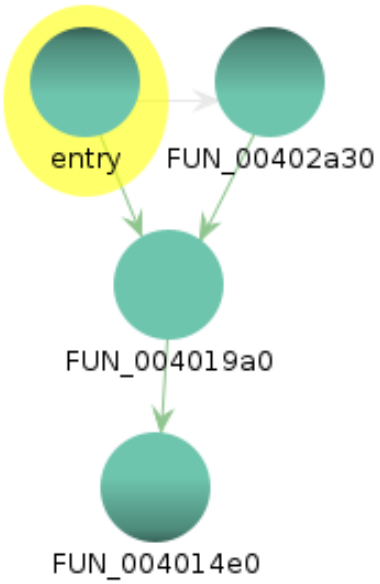
## Ricerca del main

Utilizziamo quindi lo strumento **Function Call Graph** sull'entry point per avere una prima panoramica sulle funzioni invocate.



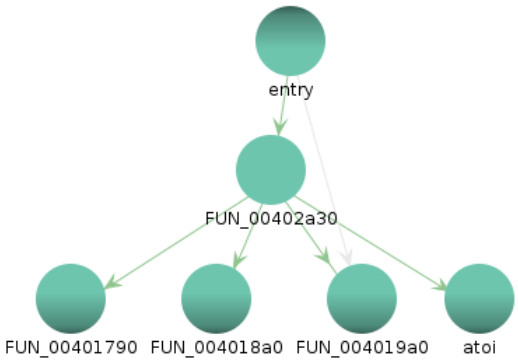

Sono presenti diverse funzioni non riconosciute da Ghidra (FUN_xxxxxxxx) che dobbiamo quindi analizzare singolarmente per individuare quale rappresenta il main del programma.

### FUN 004019a0




Questa funzione **non è il main**, in quanto viene invocata sia dall'entry point che da un'altra funzione *00402a30*. Ci si aspetta infatti che il main non venga invocato da altre funzioni.

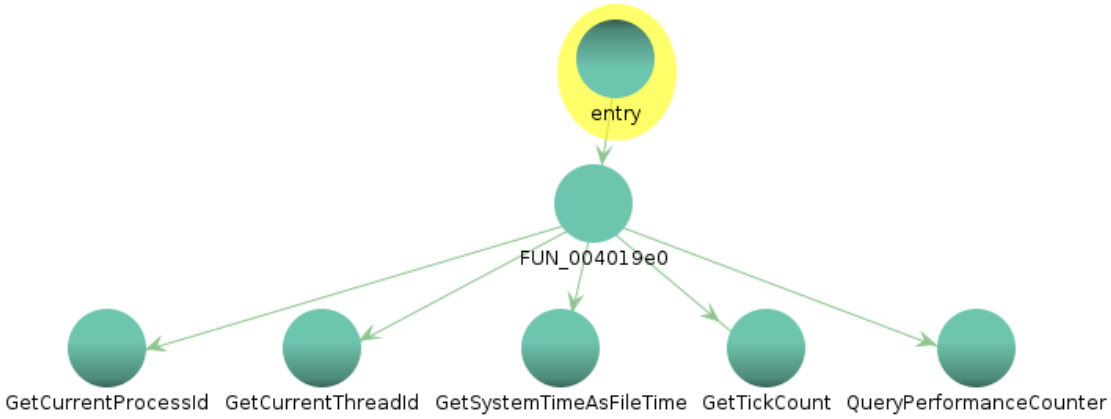
**FUN_00402a30**




Questa funzione rappresenta un **probabile main**, in quanto:

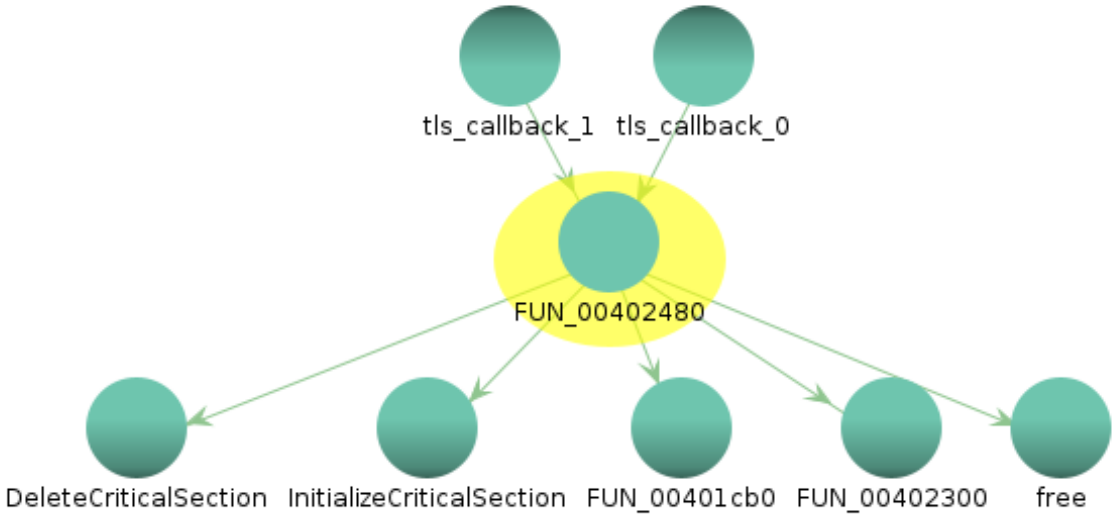
- Viene invocata direttamente dall'entry point
- Invoca altre 3 funzioni definite dal programmatore
- Invoca la funzione di libreria **atoi**, che effettua il parsing di una stringa in un intero. Questa potrebbe essere ad esempio utilizzata per convertire un argomento da riga di comando in un valore numerico.

**FUN_004019e0**



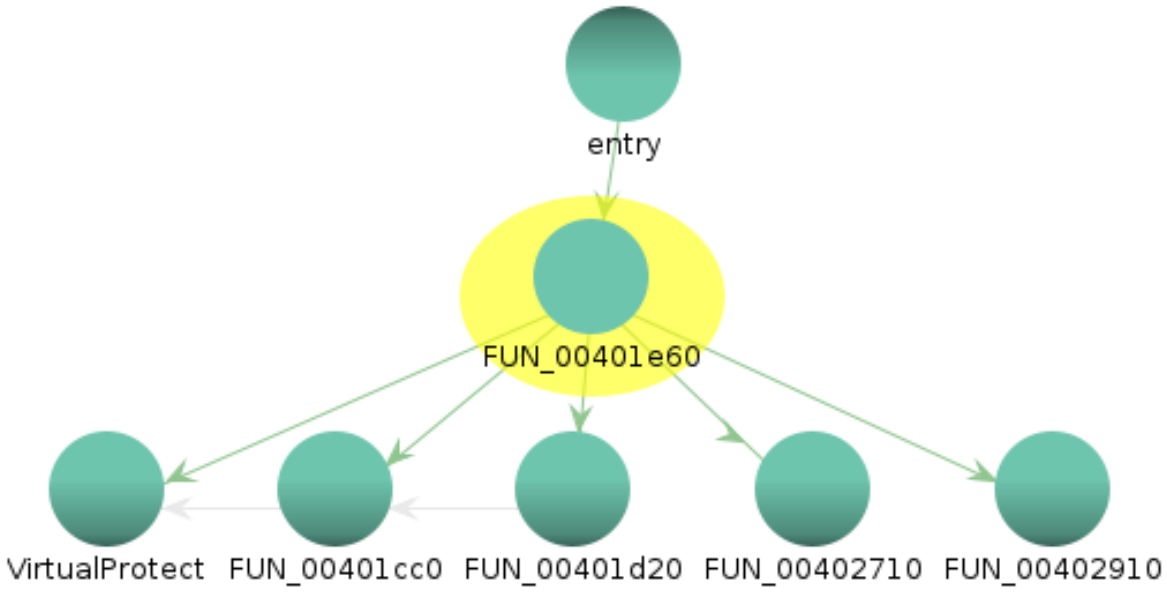

Questa funzione **non è il main** del programma, in quanto invoca soltanto funzioni di libreria, ed è probabilmente utilizzata per inizializzare l'ambiente di esecuzione.

**FUN_00402480**




Questa funzione sicuramente **non è il main**, in quanto viene invocata da delle callback.

**FUN_00401e60**




Questa funzione è un **probabile main**, in quanto invoca 4 funzioni user-defined ed una funzione **VirtualProtect**.

**Main: FUN_00402a30**

Le funzioni *00402e60* e *00402a30* sono due possibili main, per cui è necessario analizzarle più nel dettaglio. **FUN_00401e60** invoca *VirtualProtect*, che modifica la protezione su un'area di indirizzi virtuali del processo chiamante. Essendo una chiamata di basso livello, è molto più probabile che venga invocata dall'entry point per il setup dell'ambiente, rispetto ad una chiamata esplicita nel main da parte del programmatore.

Analizziamo quindi il codice di **FUN_00402a30** per vedere se rappresenta effettivamente il main del programma.

```
                     *************************************************************
                     *                          FUNCTION                         *
                     *************************************************************
                     undefined4 __cdecl FUN_00402a30(int param_1, int param_2)
undefined4         EAX:4          <RETURN>
int                Stack[0x4]:4   param_1                                 XREF[1]:     00402a3b(R)
int                Stack[0x8]:4   param_2                                 XREF[1]:     00402a3e(R)
undefined1         Stack[-0xc]:1  local_c                                 XREF[1]:     00402a93(*)
undefined4         Stack[-0x1c]:4 local_1c                                XREF[1]:     00402a77(W)
undefined4         Stack[-0x20]:4 local_20                                XREF[3]:     00402a55(*),
                                                                                       00402a71(*),
                                                                                       00402a89(*)
                     FUN_00402a30                                    XREF[1]:     entry:00401381(c)
```

In input prende due parametri, che potrebbero corrispondere ad **argc** ed **argv**, ovvero gli argomenti tradizionali della funzione *main* (che potrebbero essere argc e argv)

- **int param_1** corrisponde ad **argc** ed ha il tipo di dato corretto
- **int param_2** corrisponde ad **argv**, ma il tipo di dato è **int** invece che **char**** delle

Analizziamo quindi il **codice assembly** per capire meglio come vengono utilizzati i parametri di questa funzione, e se effettimvamente param_2 può corrispondere ad argv.

```
00402a3b 8b 75 08        MOV        ESI,dword ptr [EBP + param_1]
00402a3e 8b 5d 0c        MOV        EBX,dword ptr [EBP + param_2]
                                 ...
                     LAB_00402a52
00402a52 8b 43 04        MOV        EAX,dword ptr [EBX + 0x4]
00402a55 89 04 24        MOV        dword ptr [ESP]=>local_20,EAX
00402a58 e8 47 ff        CALL       MSVCRT.DLL::atoi
         ff ff
```

Viene scritto il contenuto dell'indirizzo *[EBP+param_2]* nel registro *EBX*. In questo contesto ghidra com param_2 non indica il valore di *param_2*, ma il suo offset all'interno dello stack. In pratica quindi viene caricato param_2 in EBX.

Possiamo quindi dire che:

- ESI contiene param_1
- EBX contiene param_2

Successivamente nella label **LAB_00402a52**, viene copiato il valore contenuto all'indirizzo *[EBX+4]* nel *registro EAX*. Il contenuto di EAX viene poi copiato tramite *MOV* all'indirizzo puntato dallo *stack pointer*. Seguendo la convenzione cdecl si sta quindi passando un parametro alla funzione **atoi**.

Fatte queste considerazioni, risulta evidente che *param_2* debba essere necessariamente un **array di char***, in quanto:

- ptr [EBX+4] corrisponde a argv[1]
- argv[1] viene passato come parametro ad *atoi*
- atoi accetta in input una stringa (quindi *char**), da convertire in intero.

Un'ulteriore conferma di ciò si ha analizzando il codice fornito dal **decompilatore**, in cui param_2 viene castato a **(char**)** prima di essere passato ad *atoi*

```
pHVar1 = (HKEY)atoi(*(char **)(param_2 + 4));
if (pHVar1 == (HKEY)0x0) {
  pHVar1 = (HKEY)0x80000002;
}
```

Possiamo quindi affermare che effettivamente **FUN_00402a30** è il **main** del programma. Per questo andiamo a:

- Modificare il tipo di dato di param_2 in char**
- Rinominare *param_1* in *argc* e *param_2* in *argv*
- Rinominare *FUN_00402a30* in *main*

## Analisi delle Funzioni

Una volta individuato il main, possiamo proseguire seguendo la logica del programma, analizzando la catena di funzioni che vengono invocate.

### Main (1)

Iniziamo proprio con l'analisi del main, andando a studiare nel dettaglio il suo comportamento.

```
                        main
00402a30 55             PUSH    EBP
00402a31 89 e5          MOV     EBP,ESP
00402a33 56             PUSH    ESI
00402a34 53             PUSH    EBX
00402a35 83 e4 f0       AND     ESP,0xfffffff0
00402a38 83 ec 10       SUB     ESP,0x10
                        ********************
00402a3b 8b 75 08       MOV     ESI,dword ptr [EBP + argc]
00402a3e 8b 5d 0c       MOV     EBX,dword ptr [EBP + argv]
00402a41 e8 5a ef       CALL    onexit_routine
         ff ff
```

#### FUN_004019a0 (onexit_routine)

Dopo il prologo vengono eseguite due *MOV* che copiano *argc in ESI* e *argv in EBX*. Successivamente viene invocata la funzione *004019a0*. Questa, come visto in precedenza, viene chiamata anche dall'entry point, ed analizzandone il codice si vede che si occupa di impostare un'attività da eseguire all'*uscita dal programma*.

```
FUN_004014e0((_onexit_t)&LAB_00401900);
return;
```

```
2 int __cdecl FUN_004014e0(_onexit_t param_1)
3
4 {
5   _onexit_t p_Var1;
6
7   p_Var1 = _onexit(param_1);
8   return -(uint)(p_Var1 == (_onexit_t)0x0);
9 }
```

Non abbiamo tuttavia informazioni sufficienti per capire quale funzione venga settata, per cui ci limitiamo al momento a rinominare **FUN_004019a0** in **onexit_routine**.

**Argomenti**

Successivamente si controlla se **argv[argc]** è **uguale a zero**.

```
10  onexit_routine();
11  if (argv[argc] == (char *)0) {
12    result = 0;
13  }
```

Analizziamo quindi l'assembly nel dettaglio:

```
00402a46 8b 04 b3       MOV     EAX,dword ptr [EBX + ESI*0x4]
00402a49 85 c0          TEST    EAX,EAX
00402a4b 74 5f          JZ      LAB_00402aac
```

- *ESI* contiene argc
- *[EBX + ESI*4]* è l'indirizzo di *argv[argc]*
- tramite *MOV*, *argv[argc]* viene copiato in *EAX*
- con *TEST EAX, EAX* si verifica se EAX è uguale a zero, settando *Zero Flag* di conseguenza
- con JZ si effettua il salto a **LAB_00402aac** se è lo ZF è impostato a 1

```
                        LAB_00402aac
00402aac 31 d2          XOR     EDX,EDX
00402aae eb e3          JMP     LAB_00402a93
```

Zero Flag

```
                        LAB_00402a93
00402a93 8d 65 f8       LEA     ESP=>local_c,[EBP + -0x8]
00402a96 89 d0          MOV     EAX,EDX
00402a98 5b             POP     EBX
00402a99 5e             POP     ESI
00402a9a 5d             POP     EBP
00402a9b c3             RET
```

Questa condizione risulta essere sempre verificata. Infatti, se vengono passati *N argomenti* da riga di comando, si avrà **argc = N+1** *(nome del programma + N argomenti)*. Quindi, **argv** avrà elementi *argv[0] … argv[N]*, e di conseguenza **argc[N+1]** non avrà mai un valore diverso da zero.

Ciò vuol dire che a prescindere dal numero e tipo di parametri passati, il programma terminerà sempre. Ovviamente questo controllo non fà parte del flusso di istruzioni originale, ma è una condizione inserita per non permettere esplicitamente l'esecuzione del programma.

Continuando con l'analisi si effettua un ulteriore controllo sul numero di argomenti, verificando se argc <3.

```
if (argc < 3) {
  argv[1] = (char *)0x0;
  argv[2] = (char *)0x0;
}
```

In particolare, questo controllo viene effettuato tramite le istruzioni **CMP** e **JLE**.

```
00402a4d 83 fe 02        CMP        ESI,0x2
00402a50 7e 4a           JLE        LAB_00402a9c
```

Si effettua il **compare** tra il valore *2* ed il contenuto del *registro ESI*. Solo se questo è inferiore o uguale a 2 viene effettuato il Jump alla label **LAB_00402a9c**.

```
                    LAB_00402a9c
00402a9c c7 43 04       MOV        dword ptr [EBX + 0x4],0x0
         00 00 00 00
00402aa3 c7 43 08       MOV        dword ptr [EBX + 0x8],0x0
         00 00 00 00
00402aaa eb a6          JMP        LAB_00402a52
```

Questo ci indica quindi che il programma **accetta 2 argomenti da riga di comando**. Tuttavia se vengono passati meno di 2 argomenti (*argc<3*), non si và in errore, ma vengono settati argv[1] ed argv[2] a 0.

Vengono successivamente inizializzate due variabili **pHVar1** e **pcVar3**, passate poi in input alla funzione *FUN_004018a0*

```
else {
  if (argc < 3) {
    argv[1] = (char *)0x0;
    argv[2] = (char *)0x0;
  }
  pHVar1 = (HKEY)atoi(argv[1]);
  if (pHVar1 == (HKEY)0x0) {
    pHVar1 = (HKEY)0x80000002;
  }
  pcVar3 = argv[2];
  if (pcVar3 == (LPCSTR)0x0) {
    pcVar3 = "SYSTEM\\ControlSet001\\Control";
  }
  ppCVar2 = FUN_004018a0(pHVar1,pcVar3);
```

**pHVar1**

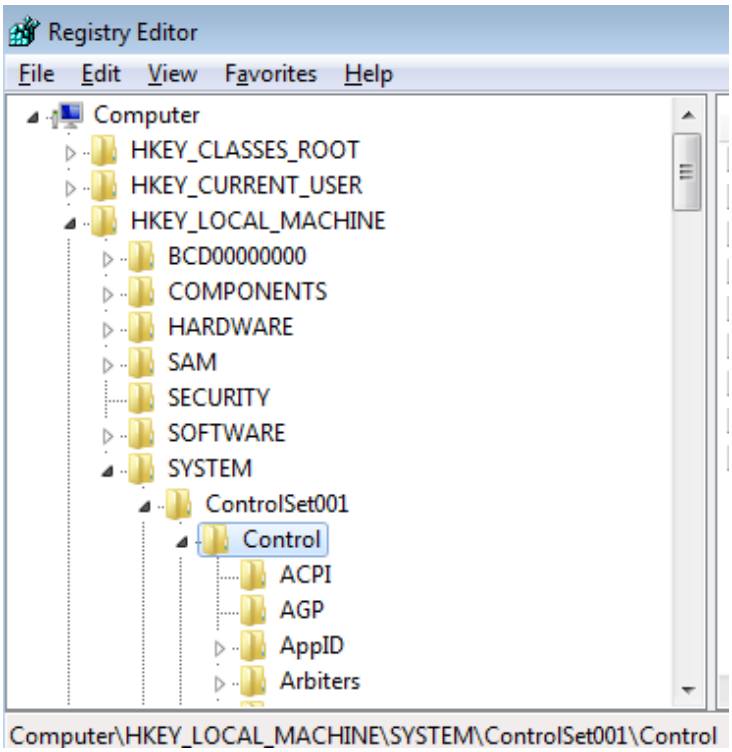
HKEY associato ad argv[1]. Nel caso in cui argv[1] sia zero, *pHVar1* viene impostata di default a *0x80000002*. Dalla documentazione vediamo che tale codice corrisponde a **HKEY_LOCAL_MACHINE**, per cui possiamo assegnarlo su *ghidra* tramite la funzione **Set Equate**. Questo ci indica quindi come il primo parametro di FUN_004018a0 sia una Handle ad una **chiave di primo livello** del *Registro di Sistema Windows*

**HKEY_LOCAL_MACHINE =**

`0x80000002`

**pcVar3**

Stringa associata ad argv[2]. Nel caso in cui argv[2] sia zero, questa viene impostata al valore **"SYSTEM\ControlSet001\Control"**. Ciò ci indica come il secondo parametro di *FUN_004018a0* rappresenti il **path della subkey**, partendo dal registro di primo livello specificato tramite il primo parametro.



### FUN_004018a0 (open_key)

Analizziamo ora il codice della *prima funzione* chiamata nel *main*

```
LPSTR * __cdecl FUN_004018a0(HKEY param_1,LPCSTR param_2)

{
  LSTATUS LVar1;
  LPSTR *ppCVar2;
  HKEY local_10 [3];

  ppCVar2 = (LPSTR *)0x0;
  LVar1 = RegOpenKeyExA(param_1,param_2,0,0xf003f,local_10);
  if (LVar1 == 0) {
    ppCVar2 = FUN_00401530(local_10[0]);
    RegCloseKey(local_10[0]);
  }
  return ppCVar2;
}
```

Analizzando il decompilato di tale funzione vediamo che viene invocata al suo interno la funzione di libreria **RegOpenKeyExA**. Dalla documentazione vediamo che questa consente di aprire la chiave di registro specificata. In particolare, i parametri che accetta sono:

```
LSTATUS RegOpenKeyExA(
  [in]           HKEY   hKey,
  [in, optional] LPCSTR lpSubKey,
  [in]           DWORD  ulOptions,
  [in]           REGSAM samDesired,
  [out]          PHKEY  phkResult
);
```

I primi due parametri che passiamo alla funzione sono *param_1* e *param_2*, che corrispondono a *pHVar1* e *pcVar3* definiti nel main. Per uniformarci alla documentazione possiamo quindi rinominare questi parametri rispettivamente in **hkey** e **lpSubKey**. Il terzo parametro **ulOptions** viene passato pari a 0.

Il quarto parametro **samDesired** indica i permessi con cui si vuole aprire la chiave. Questo è impostato a *0xf003f*, che corrisponde alla modalità *KEY_ALL_ACCESS*. Anche in questo caso utilizziamo quindi *Set Equate* per specificare il nome invece del codice numerico.

| Value | Meaning |
|---|---|
| KEY_ALL_ACCESS (0xF003F) | Combines the STANDARD_RIGHTS_REQUIRED, KEY_QUERY_VALUE, KEY_SET_VALUE, KEY_CREATE_SUB_KEY, KEY_ENUMERATE_SUB_KEYS, KEY_NOTIFY, and KEY_CREATE_LINK access rights. |

L'ultimo argomento **phkResult** invece è un parametro di output, e contiene un puntatore ad una variabile che riceve un handle verso la chiave aperta.

Possiamo quindi ridefinire alcune variabili utilizzate dal decompilatore, sfruttando le informazioni ricavate dalla documentazione. In particolare:

```
param_1      --> hkey
param_2      --> lpSubKeys
LVar1        --> openKeyRes
0xf003f      --> KEY_ALL_ACCESS
FUN_004018a0 --> open_key()
```

Il parametro **HKEY local_10 [3]** viene interpretato da Ghidra come un *array di HKEY*. Tuttavia possiamo osservare come nel **codice assembly** non viene mai effettuato un accesso del tipo *local_10 + offset*, e anche nel codice decompilato viene utilizzato solo il primo elemento *local_10[0]*. Pertanto utilizziamo la funzione di Ghidra **Retype Variable** per cambiare il tipo di dato da **HKEY [3]** in **HKEY**. Rinominiamo inoltre tale variabile in **phkResult**.

```
LPSTR * __cdecl open_key(HKEY hkey,LPCSTR lpSubKey)

{
  LSTATUS openKeyRes;
  LPSTR *ppCVar1;
  HKEY phkResult;

  ppCVar1 = (LPSTR *)0x0;
  openKeyRes = RegOpenKeyExA(hkey,lpSubKey,0,KEY_ALL_ACCESS,&phkResult);
  if (openKeyRes == 0) {
    ppCVar1 = FUN_00401530(phkResult);
    RegCloseKey(phkResult);
  }
  return ppCVar1;
}
```

Dopo aver aperto la chiave, si controlla tramite un *if statement* il valore di ritorno di *RegOpenKeyExA*. Questa funzione ritorna **0** in caso di successo, ed in tal caso viene invocata la funzione **FUN_00401530**.

Questa funzione prende come parametro l'handler *phkResult* alla chiave aperta, che verrà successivamente chiusa tramite la chiamata di libreria **RegCloseKey**

#### FUN_00401530 (read_registry)

Questa funzione risulta molto complessa, ed al suo interno vengono chiamate tre funzioni di libreria. Procediamo quindi analizzando separatamente i tre blocchi di codice.

- RegQueryInfoKeyA
- RegEnumKeyExA
- RegEnumValueA

### RegQueryInfoKeyA

```
LPSTR * __cdecl FUN_00401530(HKEY param_1)

{
  LPSTR *ppCVar1;
  LPSTR pCVar2;
  LPSTR *ppCVar3;
  LPSTR pCVar4;
  LSTATUS LVar5;
  LPSTR *ppCVar6;
  LPSTR dwIndex;

  ppCVar1 = (LPSTR *)malloc(0x34);
  if (ppCVar1 != (LPSTR *)0x0) {
    pCVar2 = (LPSTR)malloc(0x104);
    *ppCVar1 = pCVar2;
    if (pCVar2 != (LPSTR)0x0) {
      *pCVar2 = '\0';
      ppCVar1[1] = (LPSTR)0x104;
      ppCVar1[2] = (LPSTR)0x0;
      pCVar2 = (LPSTR)RegQueryInfoKeyA(param_1,pCVar2,(LPDWORD)(ppCVar1 + 1),(LPDWORD)0x0,
                                       (LPDWORD)(ppCVar1 + 2),(LPDWORD)(ppCVar1 + 3),
                                       (LPDWORD)(ppCVar1 + 4),(LPDWORD)(ppCVar1 + 5),
                                       (LPDWORD)(ppCVar1 + 6),(LPDWORD)(ppCVar1 + 7),
                                       (LPDWORD)(ppCVar1 + 8),(PFILETIME)(ppCVar1 + 9));
```

In questa porzione di codice:

- Viene allocata un'area di memoria di 52 byte (0x34) tramite **malloc**. L' indirizzo base di quest'area viene salvato in *ppCVar1*.
- Viene chiamata la funzione di libreria **RegQueryInfoKeyA**, che ottiene tutte le informazioni sulla chiave specificata.

Analizzando i parametri passati in input a *RegQueryInfoKeyA*, possiamo osservare che la variabile **ppCVar** viene utilizzata come una base per diversi accessi a memoria del tipo *ppCVar1 + offset*

Questo ci suggerisce che questa variabile rappresenta in realtà un **array** o una **struttura dati**. Analizziamo quindi i parametri di input di *RegQueryInfoKeyA*:

```
1    [in]                HKEY      hKey,
2    [out, optional]     LPSTR     lpClass,
3    [in, out, optional] LPDWORD   lpcchClass,
4                        LPDWORD   lpReserved,
5    [out, optional]     LPDWORD   lpcSubKeys,
6    [out, optional]     LPDWORD   lpcbMaxSubKeyLen,
7    [out, optional]     LPDWORD   lpcbMaxClassLen,
8    [out, optional]     LPDWORD   lpcValues,
9    [out, optional]     LPDWORD   lpcbMaxValueNameLen,
10   [out, optional]     LPDWORD   lpcbMaxValueLen,
11   [out, optional]     LPDWORD   lpcbSecurityDescriptor,
12   [out, optional]     PFILETIME lpftLastWriteTime
```

I tipi di dato non sono omogonei, ragione per cui *ppCVar1* rappresenta sicuramente una *struttura dati*. Definiamo quindi una nuova struttura chiamata **RegistryInfo**, avente dimensione pari a 52 bytes. Definiamo inoltre anche un puntatore a tale struttura (**RegistryInfo***).

**Struttura RegistryInfo (1)**

Dobbiamo ora definire i diversi **campi** di *RegistryInfo* ed i relativi **tipi di dato**. La funzione *RegQueryInfoKeyA*, ad eccezione del primo parametro *hkey*, richiede dei puntatori (LPSTR, LPDWORD, PFILETIME).

Andiamo quindi ad analizzare il codice assembly per vedere come vengono inseriti i campi della struttura sullo stack per il passaggio dei parametri a *RegQueryInfoKeyA*

```
0040157a 89 54 24 28     MOV        dword ptr [ESP + local_34],EDX
0040157e 8d 53 1c        LEA        EDX,[EBX + 28]
00401581 89 54 24 24     MOV        dword ptr [ESP + local_38],EDX
00401585 8d 53 18        LEA        EDX,[EBX + 24]
00401588 89 54 24 20     MOV        dword ptr [ESP + local_3c],EDX
0040158c 8d 53 14        LEA        EDX,[EBX + 20]
0040158f 89 54 24 1c     MOV        dword ptr [ESP + local_40],EDX
00401593 8d 53 10        LEA        EDX,[EBX + 16]
00401596 89 54 24 18     MOV        dword ptr [ESP + local_44],EDX
0040159a 8d 53 0c        LEA        EDX,[EBX + 12]
0040159d 89 54 24 14     MOV        dword ptr [ESP + local_48],EDX
004015a1 8d 53 08        LEA        EDX,[EBX + 8]
004015a4 89 44 24 04     MOV        dword ptr [ESP + local_58],EAX
004015a8 8b 44 24 60     MOV        EAX,dword ptr [ESP + param_1]
004015ac 89 54 24 10     MOV        dword ptr [ESP + local_4c],EDX
004015b0 8d 53 04        LEA        EDX,[EBX + 4]
```

Il primo parametro *param_1* è già presente sullo stack, poiché passato dal chiamante.

Il secondo parametro *lpClass* rappresenta un puntatore ad un'area di 260 bytes. Questo valore è contenuto nel registro EAX (malloc ritorna in EAX), e viene scritto sullo stack tramite una MOV.

```
0040154b c7 04 24        MOV        dword ptr [ESP]=>local_5c,260
         04 01 00 00
00401552 89 c3           MOV        EBX,EAX
00401554 e8 1b 14        CALL       MSVCRT.DLL::malloc
```

Quindi il primo parametro della struttura, che corrisponde a *lpClass*, è un **LPSTR**

Tutti i restanti parametri vengono invece caricati sullo stack utilizzando l'istruzione **LEA** (*Load Effective Address*). Questa istruzione carica l'indirizzo sorgente nel registro di destinazione.

Nel *registro EBX* è salvato l'indirizzo base della struttura, per cui con *[EBX+offset]* si accede ai diversi campi. All'istruzione *RegQueryInfoKeyA* vengono quindi passati i puntatori ai campi di *RegistryInfo*.

Questa considerazione ci indica che, ad eccezione del primo, tutti i campi della struttura non sono dei puntatori (LPDWORD, PFILETIME), ma dei valori effettivi (DWORD, FILETIME) di cui verranno poi passati i puntatori a *RegQueryInfoKeyA*

Possiamo quindi definire **RegistryInfo** come segue:

| Offset | Length | Mnemonic | DataType | Name |
|---|---|---|---|---|
| 0 | 4 | LPSTR | LPSTR | base |
| 4 | 4 | DWORD | DWORD | class |
| 8 | 4 | DWORD | DWORD | subKeysNumber |
| 12 | 4 | DWORD | DWORD | maxSubKeyLen |
| 16 | 4 | DWORD | DWORD | maxClassLen |
| 20 | 4 | DWORD | DWORD | valuesNumber |
| 24 | 4 | DWORD | DWORD | maxValueNam... |
| 28 | 4 | DWORD | DWORD | maxValueLen |
| 32 | 4 | DWORD | DWORD | securityDescri... |
| 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 | |

Gli *ultimi 8 byte* sono per ora lasciati come *undefined*, in quanto non si hanno ancora sufficienti informazioni per definirli.

Modifichiamo a questo punto il tipo di dato della variabile *pcCVar1*, specificando *RegistryInfo**. Vediamo che ora Ghidra riconosce correttamente la struttura dati.

```
regInfo = (RegistryInfo *)malloc(52);
if (regInfo != (RegistryInfo *)0) {
  var1 = (LPSTR)malloc(260);
  regInfo->base = var1;
  if (var1 != (LPSTR)0) {
    *var1 = 0;
    regInfo->class = 260;
    regInfo->subKeysNumber = 0;
    index2 = RegQueryInfoKeyA(hkey,var1,&regInfo->class,(LPDWORD)0,&regInfo->subKeysNumber,
                              &regInfo->maxSubKeyLen,&regInfo->maxClassLen,&regInfo->valuesNumber,
                              &regInfo->maxValueNameLen,&regInfo->maxValueLen,
                              &regInfo->securityDescriptor,(PFILETIME)&regInfo->lastWriteTime);
```

A seguito della chiamata di funzione, vengono scritte nei parametri di output le diverse informazioni riguardo la chiave di registro specificata. In particolare:

- **regInfo->subKeysNumber** contiene il numero di subkeys della chiave letta.
- **regInfo->valuesNumber** contiene il numero di values della chiave letta.

Successivamente si effettua un controllo sul valore di ritorno di *RegQueryInfoKeyA*. In caso di successo questa ritorna zero, altrimenti uno specifico codice di errore.

Nel caso in cui la funzione *non legga correttamente* le informazioni del registro (*index2!=0*) si effettua un **jump** alla label *info_key_exception* (ramo else). Qui viene stampato un codice di errore (puts), si azzera il puntatore alla struttura (EBX=0) e si effettua un altro jump alla label per il ritorno al chiamante.

Se invece la chiave è stata letta correttamente, si cercano di leggere tutte le **subkeys** e tutti **values** di quest'ultima.

**RegEnumKeyExA**

Si esegue innanzitutto un controllo sul numero di subkeys lette.

```
if (regInfo->subKeysNumber == 0) {
  pRVar1 = (RegistryInfo *)0x0;
}
```

Analizziamo quindi il caso in cui ci siano delle subkeys (ramo *else*)

```
else {
  dwIndex = 0;
  pRVar3 = (RegistryInfo *)0x0;
  do {
    while( true ) {
      pRVar1 = (RegistryInfo *)malloc(0x10);
      if (pRVar1 == (RegistryInfo *)0x0) goto LAB_00401776;
      pRVar1->base = (LPSTR)regInfo;
      pRVar1->class = (DWORD)pRVar3;
      _Size = regInfo->maxSubKeyLen;
      pRVar1->maxSubKeyLen = _Size;
      index3 = (LPSTR)malloc(_Size);
      pRVar1->subKeysNumber = (DWORD)index3;
      if (index3 == (LPSTR)0x0) goto LAB_00401776;
      LVar2 = RegEnumKeyExA(param_1,dwIndex,index3,&pRVar1->maxSubKeyLen,(LPDWORD)0x0,
                            (LPSTR)0x0,(LPDWORD)0x0,(PFILETIME)&regInfo->lastWriteTime);
      if (LVar2 == 0) break;
      pRVar3 = (RegistryInfo *)pRVar1->class;
      dwIndex = dwIndex + 1;
      free(pRVar1);
      pRVar1 = pRVar3;
      if (regInfo->subKeysNumber < dwIndex || regInfo->subKeysNumber == dwIndex)
      goto no_subkeys_1;
    }
    dwIndex = dwIndex + 1;
    pRVar3 = pRVar1;
  } while (dwIndex <= regInfo->subKeysNumber && regInfo->subKeysNumber != dwIndex);
}
```

Viene eseguito un **ciclo while**, annidato in un costrutto **do-while**, tramite il quale si ottengono tutte le *subkeys* della chiave considerata.

All'interno del *while* viene allocata un'area di memoria da 16 byte. L'indirizzo base di tale area viene assegnato alla variabile **pRVar1**. Ghidra identifica automaticamente tale variabile come un puntature alla struttura *RegistryInfo*. Tuttavia questa associazione è erronea in quanto l'indirizzo base è associato ad un'area di 16 byte e non di 52.

Questo ci suggerisce la presenza di una *nuova struttura* di 16 bytes. Definiamo quindi una nuova struttura **Subkey**, e definiamone i campi sfruttando le informazioni presenti nel codice e nella documentazione.

**Struttura Subkey**

Dal codice assembly possiamo intuire il tipo di dato dei primi due campi della struttura

```
0040161a 89 18           MOV        dword ptr [EAX],EBX
0040161c 89 78 04        MOV        dword ptr [EAX + 4],EDI
0040161f 8b 43 0c        MOV        EAX,dword ptr [EBX + 0xc]
00401622 89 45 0c        MOV        dword ptr [EBP + 12],EAX
00401625 89 04 24        MOV        dword ptr [ESP]=>local_5c,EAX
```

*EAX* è associato a *pRVar1*, in quanto la malloc di default scrive il valore di ritorno in EAX. Nella prima *MOV* viene scritto *EBX* nell'indirizzo puntato da *EAX*, quindi nel primo campo della struttura. In EBX è contenuto l'indirizzo base della variabile *regInfo*, per cui il primo campo della nuova struttura sarà un **puntatore ad una struttura *RegistryInfo***.

Nella seconda *MOV* viene scritto il valore di *EDI* all'interno dell'indirizzo di memoria puntato da *EAX+4*. *EDI* contiene *0* alla prima iterazione.

```
004015e8 31 c0           XOR        EAX,EAX
004015ea 89 7c 24 3c     MOV        dword ptr [ESP + local_20],EDI
004015ee 31 f6           XOR        ESI,ESI
004015f0 89 c7           MOV        EDI,EAX
```

Nelle iterazioni successive, *EDI* contiene invece un puntatore all'*ultima struttura allocata*. Infatti *EAX*, contenente l'indirizzo base dell'area allocata, viene copiato in *EBP*, e successivamente *EBP* viene copiato in *EDI*.

```
00401604 c7 04 24        MOV        dword ptr [ESP]=>local_5c,0x10
         10 00 00 00
0040160b e8 64 13        CALL       MSVCRT.DLL::malloc
         00 00
00401610 89 c5           MOV        EBP,EAX
         ...
00401643 89 ef           MOV        EDI,EBP
```

Il secondo campo della struttura *subkey* contiene quindi un puntatore ad un'altra struttura *subkey*, ed in particolare un puntatore alla **subkey precedente**.

Questo ci indica che l'*insieme di tutte le subkeys* viene mantenuto tramite una **Linked List**, in cui ogni subkey mantiene il riferimento alla subkey precedente.

Per definire gli altri valori della struttura osserviamo che questi vengono pushati sullo stack e passati come input alla funzione **RegEnumKeyExA**.

```
_Size = regInfo->maxSubKeyLen;
*(size_t *)&subkey->field_12 = _Size;
index3 = (LPSTR)malloc(_Size);
*(LPSTR *)&subkey->field_0x8 = index3;
if (index3 == (LPSTR)0x0) goto malloc_exception;
LVar1 = RegEnumKeyExA(param_1,dwIndex,index3,(LPDWORD)&subkey->field_12,(LPDWORD)0x0,
                      (LPSTR)0x0,(LPDWORD)0x0,(PFILETIME)&regInfo->lastWriteTime);
```

Tramite la documentazione di questa funzione possiamo quindi intuire i campi della struttura.

```
[in]                HKEY      hKey,
[in]                DWORD     dwIndex,
[out]               LPSTR     lpName,
[in, out]           LPDWORD   lpcchName,
                    LPDWORD   lpReserved,
[in, out]           LPSTR     lpClass,
[in, out, optional] LPDWORD   lpcchClass,
[out, optional]     PFILETIME lpftLastWriteTime
```

Come parametro **dwIndex** viene passato il campo al byte 8 della struttura. Possiamo quindi definire il **terzo campo** di **subkeys** come un *DWORD* che corrisponde all'indice della subkey corrente.

Come parametro **lpcchName** viene passato il campo al byte 12 della struttura. Possiamo quindi definire il **quarto campo** di **subkeys** come un *DWORD*, che corrisponde alla size del nome della subkey.

In definitiva, definiamo la struttura come segue:

| Offset | Length | Mnemonic | DataType | Name |
|---|---|---|---|---|
| 0 | 4 | RegistryInfo * | RegistryInfo * | infoKeyPtr |
| 4 | 4 | Subkey * | Subkey * | prevSubkey |
| 8 | 4 | DWORD | DWORD | dwIndex |
| 12 | 4 | DWORD | DWORD | chName |

Continuando con l'analisi del codice, vediamo che quando una chiave non ha altre subkey, si ottiene la lista di tutti i values ad essa collegati.

```
  if (var2 == 0) break;
  prevSubk = subkey->prevSubkey;
  index1 = index1 + 1;
  free(subkey);
  subkey = prevSubk;
  if (regInfo->subKeysNumber < index1 || regInfo->subKeysNumber == index1)
  goto no_subkeys1;
}
```

Infatti *RegEnumKeyExA* ritorna 0 solo se la chiamata ha *avuto successo* e *ci sono altre subkey*; in quel caso si esegue il *break* e si passa alla subkey successiva. Se invece il valore di ritorno è un altro (*var2!=0*) si verifica se effettivamente sono state lette tutte le subkeys (*regInfo->subKeysNumber <= index1*) e si jumpa alla label **no_subkeys1**.

**RegEnumValueA**

In questa porzione di programma vengono letti tutti i **values** presenti all'interno dell'*hkey* specificata.

```
no_subkeys_1:
        *(Subkey **)&regInfo->field_0x2c = subkey;
        pRVar2 = (RegistryInfo *)0x0;
        if (regInfo->valuesNumber != 0) {
          do {
            _Memory = (RegistryInfo *)malloc(16660);
            if (_Memory == (RegistryInfo *)0x0) goto malloc_exception;
            _Memory->class = (DWORD)pRVar2;
            _Memory->base = (LPSTR)regInfo;
            *(DWORD *)((int)(_Memory + 0x13b) + 0xc) = 0x3fff;
            *(undefined *)&_Memory->subKeysNumber = 0;
            *(LPDWORD)((int)(_Memory + 0x140) + 0x10U) = 0x100;
            LVar1 = RegEnumValueA(param_1,index2,(LPSTR)&_Memory->subKeysNumber,
                                  (LPDWORD)((int)(_Memory + 0x13b) + 0xc),(LPDWORD)0x0,
                                  (LPDWORD)((int)(_Memory + 0x13b) + 0x10),
                                  (LPBYTE)((int)(_Memory + 0x13b) + 0x14),
                                  (LPDWORD)((int)(_Memory + 0x140) + 0x10U));
            pRVar2 = _Memory;
            if (LVar1 != 0) {
              pRVar2 = (RegistryInfo *)_Memory->class;
              free(_Memory);
            }
            index2 = index2 + 1;
          } while (index2 <= regInfo->valuesNumber && regInfo->valuesNumber != index2);
        }
        *(RegistryInfo **)&regInfo->field_0x30 = pRVar2;
      }
      else {
        regInfo = (RegistryInfo *)0x0;
        puts("RegQueryInfoKey failed: key not found");
      }
      return &regInfo->base;
    }
  }
malloc_exception
```

Viene allocata nuovamente un'area di memoria, in questo caso di 16660 byte. Ciò ci indica la presenza di una *terza struttura*, che presumibilmente conterrà tutte le informazioni relative ad un *value*. Andiamo quindi a definire una nuova struttura denominata **Value**.

```
*(Value **)&_Memory->field_0x4 = pVVar2;
*(RegistryInfo **)_Memory = regInfo;
*(undefined4 *)&_Memory->field_0x4008 = 16383;
*(undefined *)&_Memory->field_0x8 = 0;
*(LPDWORD)&_Memory->field_0x4110 = 256;
LVar1 = RegEnumValueA(param_1,index2,(LPSTR)&_Memory->field_0x8,
                      (LPDWORD)&_Memory->field_0x4008,(LPDWORD)0x0,
                      (LPDWORD)&_Memory->field_0x400c,(LPBYTE)&_Memory->field_0x4010,
                      (LPDWORD)&_Memory->field_0x4110);
pVVar2 = _Memory;
```

**Struttura Value**

Alcuni campi di questa struttura vengono passati come parametro alla funzione **RegEnumValueA**, quindi sfruttando la documentazione di questa funzione possiamo ricavare alcune informazioni sui campi della struttura *Value*.

```
[in]                HKEY    hKey,
[in]                DWORD   dwIndex,
[out]               LPSTR   lpValueName,
[in, out]           LPDWORD lpcchValueName,
                    LPDWORD lpReserved,
[out, optional]     LPDWORD lpType,
[out, optional]     LPBYTE  lpData,
[in, out, optional] LPDWORD lpcbData
```

**field_0x8** corrisponde al terzo parametro *lpValueName*. Sfruttando la documentazione, vediamo che questo questo campo è quindi un **array di caratteri**, che contiene *nome del valore + carattere di terminazione*.

```
[out] lpValueName

A pointer to a buffer that receives the name of the value as a null-terminated string.
```

**field_0x4008** corrisponde al quarto parametro *lpcchValueName* ed è quindi un DWORD

**field_0x400c** corrisponde al sesto parametro *lpType* ed è quindi un DWORD

**field_0x4010** corrisponde al settimo parametro *lpData*. La documentazione ci indica come questo campo sia un *Array di Byte*

```
[out, optional] lpData

A pointer to a buffer that receives the data for the value entry. This parameter can be
    NULL if the data is not required.
```

**field_0x4110** corrisponde all'ottavo parametro *lpcbData* ed è quindi un DWORD

Inoltre vediamo che, analogamente a quanto visto con la struttura *subkeys*, anche in questo caso abbiamo una **Linked List**. In particolare **field_0x4** viene assegnato sempre alla variabile *pVVar2*, che contiene l'ultimo *value* letto.

Il primo campo della struttura, mantiene invece un puntatore alla struttura *regInfo*.

Possiamo quindi definire *Value* nel seguente modo:

| Offset | Length | Mnemonic | DataType | Name |
|---|---|---|---|---|
| 0 | 4 | RegistryInfo * | RegistryInfo * | InfoKeyPtr |
| 4 | 4 | Value * | Value * | prevValuePtr |
| 8 | 16384 | CHAR[16384] | CHAR[16384] | name |
| 16392 | 4 | DWORD | DWORD | nameSize |
| 16396 | 4 | DWORD | DWORD | type |
| 16400 | 256 | BYTE[256] | BYTE[256] | data |
| 16656 | 4 | DWORD | DWORD | dataSize |

### Struttura RegistryInfo (2)

Continuando con l'analisi del codice possiamo osservare altre due istruzioni interessanti, che ci permettono di definire gli *ultimi due parametri* della struttura *RegistryInfo*

```
61 |        *(Subkey **)&regInfo->field_0x2c = subkey;
                                ...
84 |        *(Value **)&regInfo->field_0x30 = pVVar2;
```

- **field_0x2c** contiene il puntatore all'*ultima subkey* che è stata letta
- **field_0x30** contiene il puntatore all'*ultimo value* che è stato letto

Possiamo quindi completare la struttura *RegistryInfo*, inserendo anche gli ultimi due campi precedentemente non definiti.

| Offset | Length | Mnemonic | DataType | Name |
|---|---|---|---|---|
| 0 | 4 | LPSTR | LPSTR | base |
| 4 | 4 | DWORD | DWORD | class |
| 8 | 4 | DWORD | DWORD | subKeysNumber |
| 12 | 4 | DWORD | DWORD | maxSubKeyLen |
| 16 | 4 | DWORD | DWORD | maxClassLen |
| 20 | 4 | DWORD | DWORD | valuesNumber |
| 24 | 4 | DWORD | DWORD | maxValueNameLen |
| 28 | 4 | DWORD | DWORD | maxValueLen |
| 32 | 4 | DWORD | DWORD | securityDescriptor |
| 36 | 8 | FILETIME | FILETIME | lastWriteTime |
| 44 | 4 | Subkey * | Subkey * | lastSubkey |
| 48 | 4 | Value * | Value * | lastValue |

```
no_subkeys1:
        regInfo->lastSubkey = subkey;
        prevValue = (Value *)0x0;
                        /* Controllo se l'hkey non ha nessun valore */
        if (regInfo->valuesNumber != 0) {
          do {
            value = (Value *)malloc(16660);
            if (value == (Value *)0) goto malloc_exception;
            value->prevValuePtr = prevValue;
            value->InfoKeyPtr = regInfo;
            value->nameSize = 16383;
            value->name[0] = '\0';
            value->dataSize = 256;
            var2 = RegEnumValueA(hkey,index2,value->name,&value->nameSize,(LPDWORD)0,&value->type,
                                 value->data,&value->dataSize);
                        /* Controllo se ha fallito o se non ci sono altri Values */
            prevValue = value;
            if (var2 != 0) {
              prevValue = value->prevValuePtr;
              free(value);
            }
            index2 = index2 + 1;
          } while (index2 <= regInfo->valuesNumber && regInfo->valuesNumber != index2);
        }
        regInfo->lastValue = prevValue;
      }
      else {
        regInfo = (RegistryInfo *)0x0;
        puts("RegQueryInfoKey failed: key not found");
      }
      return regInfo;
    }
  }
malloc_exception:
  puts("Memory allocation error");
                        /* WARNING: Subroutine does not return */
  exit(1);
}
```

### open_key - Valore di ritorno

Una volte lette tutte le subkeys e tutti i value, se non ci sono stati errori viene ritornato **regInfo**. Questa variabile è il puntatore alla struttura che contiene tutte le informazioni del registro, compresi i puntatori alle linked list delle subkeys e dei values.

Ritornando quindi alla funzione *open_key*, vediamo che viene chiamata la funzione di libreria **RegCloseKey**, per chiudere la chiave precedentemente aperta.

```
LPSTR * __cdecl open_key(HKEY hkey,LPCSTR lpSubKey)

{
  LSTATUS openKeyRes;
  RegistryInfo *readRegistryRes;
  HKEY phkResult;

  readRegistryRes = (RegistryInfo *)0;
  openKeyRes = RegOpenKeyExA(hkey,lpSubKey,0,KEY_ALL_ACCESS,&phkResult);
  if (openKeyRes == 0) {
    readRegistryRes = read_registry(phkResult);
    RegCloseKey(phkResult);
  }
  return &readRegistryRes->base;
}
```

Come possiamo vedere dalla specifica, sia **open_key** che **read_registry** scrivono il proprio valore di ritorno nel registro *EAX*.

```
                     *************************************************
                     LPSTR * __cdecl open_key(HKEY hkey, LPCSTR lpSubKey)
LPSTR *                EAX:4          <RETURN>

                      ************************************************
                      RegistryInfo * __cdecl read_registry(HKEY hkey)
RegistryInfo *          EAX:4          <RETURN>
```

Nella funzione **open_key**, *EAX* conterrà quindi il valore ritornato da *read_key*, e quindi il puntatore alla struttura *RegistryInfo*. Questo valore viene copiato nel *registro EBX*

```
004018de 89 04 24        MOV        dword ptr [ESP]=>local_3c,EAX
004018e1 e8 4a fc        CALL       read_registry
         ff ff
004018e6 89 c3           MOV        EBX,EAX
```

Nella label per il *ritorno al main*, viene copiato *EBX* in *EAX*.

```
                     return_lab
004018f8 83 c4 38        ADD        ESP,0x38
004018fb 89 d8           MOV        EAX,EBX
004018fd 5b              POP        EBX
004018fe c3              RET
004018ff 90              NOP
```

Questo ci indica come il *valore ritornato da open_key* sia di tipo **RegistryInfo***. Ghidra tuttavia interpreta tale valore come un **LPSTR***, per cui andiamo modificarlo tramite **Retype Return**.

```
2  RegistryInfo * __cdecl open_key(HKEY hkey,LPCSTR lpSubKey)
3
4  {
5    LSTATUS openKeyRes;
6    RegistryInfo *readRegistryRes;
7    HKEY phkResult;
8
9    readRegistryRes = (RegistryInfo *)0;
10   openKeyRes = RegOpenKeyExA(hkey,lpSubKey,0,KEY_ALL_ACCESS,&phkResult);
11   if (openKeyRes == 0) {
12     readRegistryRes = read_registry(phkResult);
13     RegCloseKey(phkResult);
14   }
15   return readRegistryRes;
16 }
```

#### FUN_00401790 - print_registry

Completata la descrizione di *read_key*, possiamo procedere con l'analisi del main.

Se open_key è stata eseguita correttamente, viene chiamata la funzione **FUN_00401790**, passando come argomento un puntatore al campo *base* della struttura ritornata da *open_key*.

```
27      registryData = open_key(hkey,lpSubKey);
28      uVar1 = 1;
29      if (registryData != (RegistryInfo *)0) {
30        FUN_00401790(&registryData->base);
31        uVar1 = 0;
32      }
33    }
34    return uVar1;
```

Tuttavia il passaggio di tale campo della struttura è descritto erroneamente da *ghidra*, in quanto non è riuscito a definire il tipo di dato del parametro di input.

```
2  void __cdecl FUN_00401790(undefined4 *param_1)
3
4  {
5    byte *pbVar1;
6    int iVar2;
7    uint uVar3;
8    undefined4 uVar4;
```

Analizziamo dunque nel dettaglio come il main passa l'argomento alla funzione *00401790*.

```
00402a7b e8 20 ee ff ff    CALL    open_key
00402a80 ba 01 00 00 00    MOV     EDX,1
00402a85 85 c0             TEST    EAX,EAX
00402a87 74 0a             JZ      return_label
00402a89 89 04 24          MOV     dword ptr [ESP]=>hkey,EAX
00402a8c e8 ff ec ff ff    CALL    print_registry
```

Capiamo che questa funzione prende in input un parametro di tipo *RegistryInfo**, in quanto:

- *open_key* ritorna il *puntatore a RegistryInfo* in *EAX*.
- *EAX* viene messo sullo stack tramite *MOV* per passare l'argomento a *FUN_00401790*

Possiamo quindi ridefinire il parametro **undefined *param_1** di **FUN_00401790** in **RegistryInfo* registryData**.

```
2  void __cdecl print_registry(RegistryInfo *registryData)
3
4  {
5    byte *pbVar1;
6    Subkey *subkeys;
7    uint index;
8    Value *values;
9    DWORD var;
10
11   printf("Class: %s\n",registryData->base);
12   printf("Security descriptor: 0x%lx\n",registryData->securityDescriptor);
13   var = (registryData->lastWriteTime).dwHighDateTime;
14   printf("Time: %08lx%08lx\n",(registryData->lastWriteTime).dwLowDateTime,var);
```

Vediamo che anche nel main viene riconosciuto correttamente il parametro passato alla funzione:

```
if (registryData != (RegistryInfo *)0) {
  FUN_00401790(registryData);
  result = 0;
}
```

Analizzando più nel dettaglio questa funzione, vediamo che viene utilizzata la funzione di libreria **printf** per scrivere su standard output tutte le informazioni sul registro. Nell'ordine, i valori che vengono vengono stampati sono:

- **registryData->base**, contenente le *user-defined class* della chiave.
- **registryData->securityDescriptor**
- **registryData->lastWriteTime**, contenente il timestamp dell'ultima modifica sulla chiave
- tutte le **subkeys**
- tutti i **values**

Dopo aver stampato tutti questi valori, la funzione semplicemente *ritorna al main*. Focalizziamoci quindi su come vengono stampate queste informazioni.

**Subkeys**

Viene eseguito un *ciclo do-while*, partendo dall'*ultima subkey*, acceduta tramite il campo *lastSubkey* del parametro di input. Viene stampato quindi l'indice della subkey corrente, e si scorre l'intera linked list tramite il puntatore alla sottochiave precedente.

```
17  subkeys = registryData->lastSubkey;
18  if (subkeys != (Subkey *)0x0) {
19    puts("Sub-keys:");
20    do {
21      printf("\t%s\n",subkeys->dwIndex,var);
22      subkeys = subkeys->prevSubkey;
23    } while (subkeys != (Subkey *)0x0);
24  }
```

Dopo aver stampato la prima subkey (*prevSubkey==0*), termina il ciclo while e si procede andando a stampare tutti i values.

```
                    print_subkeys                                   XREF[1]:
004017f0 8b 43 08        MOV        EAX,dword ptr [EBX + 0x8]
004017f3 c7 04 24        MOV        dword ptr [ESP]=>local_1c,s__%s_00404081
         81 40 40 00
004017fa 89 44 24 04     MOV        dword ptr [ESP + local_18],EAX
004017fe e8 61 11        CALL       MSVCRT.DLL::printf
         00 00
00401803 8b 5b 04        MOV        EBX,dword ptr [EBX + 0x4]
00401806 85 db           TEST       EBX,EBX
00401808 75 e6           JNZ        print_subkeys
```

**Values**

Si esegue anche in questo caso un *ciclo do-while*, partendo dall'ultimo value individuato tramite il campo *lastValue*. Vengono stampati il *nome* ed il *tipo* del valore corrente, per poi stampare l'intero campo *data* scorrendo uno per uno i singoli byte dell'array.

```
27  values = registryData->lastValue;
28  if (values != (Value *)0) {
29    puts("Values:");
30    do {
31      var = values->type;
32      printf("\t%s: [%lu] ",values->name,var);
33      if (values->dataSize != 0) {
34        index = 0;
35        do {
36          pbVar1 = values->data + index;
37          index = index + 1;
38          printf(" %02x",(uint)*pbVar1,var);
39        } while (index <= values->dataSize && values->dataSize != index);
40      }
41      printf(" (%s)\n",values->data,var);
42      values = values->prevValuePtr;
43    } while (values != (Value *)0x0);
44  }
```

Successivamente si accede al *value precedente* tramite il puntatore, scorrendo in questo modo l'*intera linked list* dei valori. Dopo aver stampato il primo value, il controllo torna al *main*, eseguendo la procedura di ritorno al chiamante.

**main (2)**

Dopo aver terminato la descrizione della funzione *print_registry*, possiamo analizzare qual è il valore di ritorno del main.

```
27      registryData = open_key(hkey,lpSubKey);
28      result = 1;
29      if (registryData != (RegistryInfo *)0) {
30        print_registry(registryData);
31        result = 0;
32      }
33    }
34    return result;
35  }
```

Escludendo il primo controllo su argv[argc], vediamo che result viene settato a 1 dopo la chiamata open_key. Se questa ha avuto successo, (registryData!=0), viene settato result a 0. Quindi il programma ritorna 0 se il registro è stato letto correttamente, 1 altrimenti.

## Conclusione

Dopo aver completato l'analisi di tutte le funzioni invocate a partire dal main, è possibile descrivere il comportamento complessivo del programma.

Il main accetta due parametri:

- Il **codice** di una **root key**
- Il **path** della **subkey** che vogliamo analizzare.

Nel programma sono state definite 3 strutture dati:

- **Subkey**: Mantiene le informazioni su una subkey, ed il riferimento alla subkey precedente.
- **Value**: Mantiene le informazioni su un value, ed il riferimento al value precedente.
- **RegistryInfo**: Mantiene le informazioni sul registro. In particolare tiene traccia delle linked list relative alle subkey ed ai values.

La prima funzione invocata dal main è **open_key**, che permette di aprire un handle verso il registro, e di leggere tutti i dati al suo interno. Questa funzione restituisce un puntatore ad una struttura **RegistryInfo**.

Tale puntatore viene passato in input alla funzione **print_registry**, che stampa tutte le informazioni precedentemente lette dal registro. In particolare stampa anche la lista di subkeys e values, scorrendo le rispettive linked list.

Infine il programma termina, ritornando **1** se ci sono stati errori nella lettura del registro, **0** altrimenti.